Estructura de computadors

Departament d'Arquitectura de Computadors

Grau en Enginyeria Informàtica
Facultat d'informàtica de Barcelona
Universitat politècnica de Catalunya

Professors de l'assignatura

14 d'octubre de 2026 (ae6c355)

Continguts

Presentació **1**

- Qualificació . . . 1
  - Nota final . . . 1
  - Examen parcial — temari . . . 1
  - Examen final — temari . . . 2
  - Laboratori . . . 2
  - Competència transversal . . . 2
- Material docent . . . 2
  - Apunts, problemes i solucions . . . 2
  - Laboratori . . . 3
- Recursos per a l'estudi . . . 3
  - El simulador (RARS) . . . 3
  - El compilador (GNU/GCC) . . . 4
  - El Racó . . . 4
  - Pàgina de l'assignatura . . . 4
  - Guia docent (FIB) . . . 4
- Referències tècniques . . . 4
- Bibliografia . . . 5
  - Bibliografia complementària . . . 6

I Apunts **7**

1 Tema 1: Introducció **8**

1.1 La gestió de la complexitat: capes i interfícies . . . 8
- 1.1.1 Nivells d'abstracció . . . 8
- 1.1.2 Interfícies: els contractes del sistema . . . 9

1.2 El cicle de vida del programa: generació vs. execució . . . 9
- 1.2.1 El flux de generació: la visió del programador . . . 10
- 1.2.2 El flux d'execució: la visió del processador . . . 10
- 1.2.3 El rol de les interfícies . . . 11
- 1.2.4 La dualitat de l'entorn: amfitrió vs. *target* . . . 12
- 1.2.5 El llenguatge C: el pont de l'abstracció . . . 13
- 1.2.6 Traducció: de C al llenguatge màquina . . . 14
- 1.2.7 C vs. C++ . . . 16
- 1.2.8 Llenguatges interpretats . . . 18

1.3 Arquitectura física: el model de von Neumann . . . 18
- 1.3.1 El concepte de programa emmagatzemat: instruccions i dades a la mateixa memòria 18
- 1.3.2 Els components del model . . . 19
- 1.3.3 El cicle d'instrucció (*fetch-decode-execute*) . . . 19

1.4 Codificació de nombres naturals en base 2 . . . 20
- 1.4.1 Interpretació: bits → natural . . . 20
- 1.4.2 Representació: natural → bits . . . 20
- 1.4.3 Rang de representació . . . 21

1.4.4 Extensió de zeros . . . 21
1.4.5 Notació hexadecimal . . . 21
1.5 Codificació de nombres enters en base 2 . . . 22
1.5.1 Enters en complement a 2 (Ca2) . . . 22
1.5.2 Enters en excés . . . 23
1.5.3 Enters en signe i magnitud . . . 24
1.5.4 Enters en complement a 1 (Ca1) . . . 24
1.6 Operacions aritmètiques en complement a 2 . . . 24
1.6.1 Suma en Ca2 . . . 24
1.6.2 Inversió de signe . . . 25
1.6.3 Resta en Ca2 . . . 26
1.6.4 Multiplicació en Ca2 . . . 27
1.6.5 Divisió en Ca2 . . . 27

2 Tema 2: Instruccions i tipus de dades bàsics 29
2.1 Introducció a l'ISA de RISC-V . . . 29
2.1.1 *Instruction Set Architecture*, ISA . . . 29
2.1.2 Arquitectura de tipus RISC (vs. CISC) . . . 30
2.1.3 L'arquitectura RISC-V . . . 31
2.1.4 La base RV32I . . . 33
2.1.5 L'extensió M . . . 35
2.1.6 L'extensió F . . . 35
2.2 Format de les instruccions RV32I . . . 36
2.3 Estructura d'un programa en assemblador . . . 39
2.3.1 Segments . . . 39
2.3.2 Etiquetes . . . 40
2.3.3 Directives . . . 41
2.3.4 Constants simbòliques . . . 41
2.3.5 Instruccions, pseudoinstruccions i macros . . . 42
2.3.6 Comentaris . . . 48
2.3.7 Format del codi . . . 48
2.3.8 Esquelet d'un programa . . . 49
2.4 Variables escalars . . . 50
2.4.1 Declaració . . . 50
2.4.2 Tipus . . . 50
2.4.3 Visibilitat i durada . . . 51
2.4.4 Mida i rang de valors representables . . . 52
2.5 La memòria . . . 53
2.5.1 Ordre dels bytes (*endianness*) . . . 54
2.5.2 Restriccions d'alineació . . . 56
2.5.3 Directives d'alineació de memòria . . . 57
2.6 Operands i modes d'adreçament . . . 58
2.6.1 Operands en mode registre . . . 58
2.6.2 Operands en mode immediat . . . 60
2.6.3 Operands en mode base + desplaçament . . . 61
2.7 Punters . . . 65
2.7.1 C — Un tipus de punter per cada tipus de dades . . . 65
2.7.2 Declaració . . . 66
2.7.3 C — &: l'operador «adreça de» . . . 67
2.7.4 Assignació i inicialització en la declaració . . . 67
2.7.5 Desreferència o indirecció . . . 68
2.7.6 Aritmètica de punters . . . 70

2.7.7 Absència de control en l'accés a memòria . . . 71
2.8 Vectors . . . 71
2.8.1 Declaració i inicialització . . . 71
2.8.2 C — Tipus `<tipus_elements> nom_vector[<quantitat_elements>]` . . . 73
2.8.3 Indexació . . . 74
2.8.4 C — Relació entre punters i vectors . . . 75
2.8.5 Accés a vectors amb bucles . . . 77
2.9 Caràcters i cadenes de caràcters . . . 79
2.9.1 Codificació de caràcters . . . 79
2.9.2 Representació de caràcters en C i RV32I . . . 80
2.9.3 Declaració i inicialització de caràcters . . . 80
2.9.4 Cadenes de caràcters . . . 81

3 Tema 3: Traducció de programes 83
3.1 Desplaçaments de bits . . . 83
3.1.1 Desplaçaments lògics . . . 83
3.1.2 Desplaçament aritmètic a la dreta . . . 84
3.1.3 Traducció dels operadors C `<<` i `>>` . . . 85
3.1.4 Aplicació d'`sll`/`slli`: multiplicació per potències de 2 . . . 86
3.1.5 Aplicació d'`srl`/`srli` i `sra`/`srai`: divisió entera per potències de 2 . . . 86
3.2 Operacions lògiques bit a bit . . . 86
3.2.1 Operadors `&`, `|` i `^` (`and`, `or` i `xor`) . . . 86
3.2.2 Aplicació d'`and`: selecció de bits (màscara) . . . 87
3.2.3 Aplicació d'`or`: posar bits a `1` . . . 87
3.2.4 Aplicació d'`xor`: complementar bits . . . 88
3.3 Comparacions i operacions booleanes . . . 88
3.3.1 Instruccions de comparació: `slt` i `sltu` . . . 88
3.3.2 Traducció de `==` i `!=` . . . 89
3.3.3 Traducció de la negació booleana `!` . . . 90
3.3.4 Traducció de `<`, `>`, `<=`, `>=` . . . 91
3.3.5 Traducció de `&&` i `||`: avaluació tardana . . . 91
3.4 Salts . . . 92
3.4.1 Salts condicionals . . . 92
3.4.2 Salts incondicionals relatius al PC . . . 95
3.4.3 Salts incondicionals indirectes . . . 95
3.5 Revisió dels modes d'adreçament . . . 96
3.6 Sentències alternatives: `if-then-else` i `switch` . . . 97
3.6.1 Sentència `if-then-else` . . . 97
3.6.2 Avaluació tardana als salts condicionals encadenats . . . 98
3.6.3 Sentència `switch` . . . 99
3.7 Sentències iteratives: `while`, `do-while` i `for` . . . 100
3.7.1 Sentència `while` . . . 100
3.7.2 Sentència `do-while` . . . 102
3.7.3 Sentència `for` . . . 103
3.8 Estructura de la memòria . . . 104
3.8.1 RARS — Mapa de memòria . . . 105
3.8.2 Accés a variables globals . . . 106
3.9 Subrutines . . . 107
3.9.1 Crida i retorn (el registre `ra`) . . . 107
3.9.2 Pas de paràmetres i retorn de resultats (els registres `a0`–`a7`) . . . 108
3.9.3 Variables locals: el bloc d'activació (el registre `sp`) . . . 112
3.9.4 Subrutines multinivell: la pila del programa (els registres `s0`–`s11`) . . . 116

3.10 Compilació, assemblatge, enllaçament i càrrega . . . 124
3.10.1 Compilació separada . . . 125
3.10.2 Compilació . . . 126
3.10.3 Assemblatge . . . 126
3.10.4 Enllaçament . . . 128
3.10.5 Càrrega en memòria . . . 129
3.10.6 GCC/Clang — El flux de generació complet . . . 130

4 Tema 4: Aritmètica entera i matrius 132
4.1 Suma, resta i canvi de signe . . . 132
4.2 Desbordament en suma i resta d'enters . . . 132
4.2.1 Condicions de desbordament . . . 133
4.2.2 Detecció del desbordament per maquinari . . . 133
4.2.3 Detecció del desbordament per programari . . . 134
4.3 Multiplicació entera . . . 135
4.3.1 Algorisme . . . 135
4.3.2 Circuit seqüencial de multiplicació . . . 136
4.3.3 Instruccions de multiplicació (extensió M) . . . 138
4.3.4 Desbordament en la multiplicació . . . 139
4.4 Divisió entera . . . 140
4.4.1 Algorisme . . . 140
4.4.2 Circuit seqüencial de divisió . . . 141
4.4.3 Instruccions de divisió entera a RV32IM . . . 142
4.4.4 Divisió per potències de 2 . . . 143
4.5 Matrius . . . 143
4.5.1 Declaració i emmagatzematge . . . 144
4.5.2 Accés aleatori . . . 145
4.6 Optimitzacions de bucle . . . 148
4.6.1 Optimització #1 — Accés seqüencial . . . 148
4.6.2 Optimització #2 — Avaluació de la condició al final . . . 149
4.6.3 Optimització #3 — Eliminació de la variable d'inducció . . . 150
4.6.4 Recorreguts de matrius amb accés seqüencial . . . 151

5 Tema 5: Coma flotant 153
5.1 Representació en coma flotant . . . 153
5.1.1 Notació científica normalitzada . . . 153
5.1.2 Representació binària . . . 153
5.1.3 L'estàndard IEEE 754 . . . 154
5.1.4 Rang i precisió . . . 158
5.1.5 Error de representació i subdesbordament . . . 158
5.1.6 Codificacions especials . . . 160
5.1.7 Arrodoniment . . . 162
5.2 Operacions en coma flotant . . . 164
5.2.1 Suma i resta . . . 164
5.2.2 Multiplicació i divisió . . . 165
5.2.3 No-associativitat . . . 167
5.3 Coma flotant a RISC-V (RV32F) . . . 168
5.3.1 Registres i ABI . . . 168
5.3.2 Instruccions . . . 170
5.3.3 Exemple de traducció . . . 173

6 Tema 6: Rendiment i potència **175**
6.1 Rendiment . . . 175
6.1.1 Definicions . . . 175
6.1.2 Factors que influeixen en el temps d'execució . . . 176
6.2 Llei d'Amdahl . . . 179
6.3 Potència . . . 180
6.3.1 Potència dinàmica . . . 181
6.3.2 Potència estàtica . . . 182
6.3.3 Miniaturització i límit tèrmic . . . 183

7 Tema 7: Memòria cau **185**
7.1 La jerarquia de memòria . . . 185
7.1.1 Tecnologies de memòria . . . 185
7.1.2 El problema de la bretxa (*gap*) entre processador i memòria . . . 185
7.1.3 Localitat dels programes . . . 188
7.1.4 La jerarquia de memòria . . . 188
7.2 La memòria cau . . . 189
7.2.1 Principi de funcionament . . . 189
7.2.2 Organització: blocs, línies, etiquetes i bit de validesa . . . 189
7.2.3 Terminologia de rendiment . . . 191
7.2.4 Gestió d'una lectura . . . 191
7.3 Polítiques d'emplaçament . . . 192
7.3.1 Correspondència directa . . . 192
7.3.2 Memòria cau associativa per conjunts . . . 195
7.3.3 Memòria cau completament associativa . . . 195
7.4 Polítiques de reemplaçament . . . 198
7.5 Polítiques d'escriptura . . . 199
7.5.1 Gestió de les escriptures en cas d'encert . . . 200
7.5.2 Polítiques en cas de fallada . . . 200
7.5.3 Combinacions habituals . . . 201
7.5.4 Exemples . . . 201
7.6 Polítiques — Resum . . . 204
7.7 Rendiment . . . 204
7.7.1 Model de temps simplificat . . . 205
7.7.2 Temps d'accés mitjà (TAM) . . . 206
7.7.3 Impacte de les fallades en el temps d'execució . . . 207
7.8 Tipologia de les fallades . . . 208
7.8.1 Exemple: fallades de conflicte i d'arrencada en fred . . . 210
7.8.2 Exemple: fallades de capacitat . . . 212
7.9 Memòries cau multinivell . . . 212
7.9.1 Estructura i funcionament . . . 212
7.9.2 Disseny diferencial L1 i L2 . . . 214
7.9.3 Memòries cau separades d'instruccions i dades . . . 215
7.9.4 Jerarquies actuals . . . 215

8 Tema 8: Memòria virtual **217**
8.1 Introducció . . . 217
8.1.1 El problema: reubicació i mida limitada . . . 217
8.1.2 Espais d'adreçament lògic i físic . . . 217
8.1.3 El disc com a nivell addicional de la jerarquia . . . 218
8.2 Paginació . . . 218
8.2.1 Pàgines i marcs de pàgina . . . 218

8.2.2 Traducció d'adreces: VPN, PPN i desplaçament . . . 220
8.3 La taula de pàgines . . . 221
8.3.1 Estructura d'una entrada (PTE) . . . 221
8.3.2 El registre de taula de pàgines i el canvi de context . . . 223
8.3.3 Taules de pàgines multinivell . . . 224
8.4 Fallada de pàgina . . . 225
8.5 Traducció ràpida: el TLB . . . 226
8.5.1 Estructura i funcionament . . . 227
8.5.2 Encert de TLB . . . 227
8.5.3 Fallada de TLB . . . 228
8.5.4 Entrades lliures i invalidació del TLB . . . 229
8.5.5 El bit D: consistència amb la taula de pàgines . . . 229
8.5.6 Flux complet de traducció . . . 230
8.6 Protecció i compartició . . . 231
8.6.1 Protecció entre processos . . . 231
8.6.2 Protecció contra escriptura . . . 231
8.6.3 Compartició de pàgines . . . 232
8.7 Integració del TLB i la memòria cau . . . 233
8.7.1 Memòria cau indexada físicament . . . 233
8.7.2 Memòria cau indexada i etiquetada virtualment . . . 233
8.7.3 VIPT: la solució habitual . . . 234
8.8 Exemple pràctic . . . 235

9 Tema 9: Excepcions i interrupcions 240
9.1 Introducció . . . 240
9.1.1 Necessitat . . . 240
9.1.2 Tipologia . . . 240
9.2 Suport de maquinari: registres de control i estat . . . 241
9.2.1 El model de privilegis de RISC-V . . . 241
9.2.2 `mcause` — Codi de causa . . . 243
9.2.3 `mepc` — Adreça de retorn . . . 244
9.2.4 `mstatus` — Estat global . . . 244
9.2.5 `mtvec` — Vector d'excepcions . . . 245
9.2.6 `mtval` — Informació addicional de l'excepció . . . 245
9.2.7 `mip` / `mie` — Peticions i màscares d'interrupció . . . 246
9.2.8 Els registres equivalents en mode S . . . 246
9.3 Flux de maquinari en detectar una excepció o interrupció . . . 247
9.3.1 Accions automàtiques del processador . . . 247
9.3.2 Diferències entre excepció i interrupció . . . 247
9.4 La rutina de servei d'excepcions . . . 248
9.4.1 Responsabilitats . . . 248
9.4.2 La instrucció `mret` . . . 249
9.4.3 Ajust de `mepc` . . . 249
9.4.4 Exemple: RSE mínima en RISC-V . . . 250
9.5 Crida al sistema (`ecall`) . . . 252
9.5.1 Protecció: mode U vs. mode M . . . 252
9.5.2 `ecall` com a porta d'entrada controlada a l'SO . . . 252
9.5.3 Convenció de crides al sistema a RARS . . . 252
9.5.4 Convenció ABI de Linux . . . 254
9.6 Interrupcions de dispositius d'E/S . . . 257
9.6.1 Sincronització per enquesta vs. per interrupcions . . . 257
9.6.2 El cicle de vida d'una interrupció . . . 257

9.6.3 Permisos: `MIE` global i màscares `mie` . . . 258
9.6.4 Identificació del dispositiu a l'RSE . . . 258
9.7 La fallada de pàgina . . . 259
9.7.1 Del TLB a la fallada de pàgina . . . 259
9.7.2 Codi de causa i adreça de la fallada . . . 260
9.7.3 Implementació en mode S . . . 260

II Problemes **263**

1 Tema 1: Introducció **264**
1.1 El cicle de vida del programa: Generació vs. execució . . . 264
1.2 Codificació de nombres naturals i enters . . . 264
1.3 Operacions aritmètiques en complement a 2 . . . 267

2 Tema 2: Instruccions i tipus de dades bàsics **268**
2.1 Operands en registre . . . 268
2.2 Constants i immediats . . . 268
2.3 Operands en memòria . . . 270
2.4 Representació de naturals i enters . . . 272
2.5 Representació de caràcters . . . 272
2.6 Format de les instruccions . . . 274
2.7 Punters . . . 275
2.8 Vectors . . . 278
2.9 Cadenes de caràcters . . . 279

3 Tema 3: Traducció de programes **281**
3.1 Operacions lògiques i desplaçaments . . . 281
3.2 Sentències condicionals i bucles . . . 282
3.3 Subrutines: introducció . . . 287
3.4 Subrutines: context i bloc d'activació . . . 291
3.5 Compilació, assemblatge, enllaçament i càrrega . . . 293

4 Tema 4: Aritmètica entera i matrius **294**
4.1 Suma i resta . . . 294
4.2 Multiplicació . . . 295
4.3 Matrius . . . 296
4.4 Accés seqüencial . . . 299
4.5 Divisió . . . 304

5 Tema 5: Coma flotant **306**
5.1 Representació IEEE 754 . . . 306
5.2 Operacions . . . 307
5.3 No-associativitat . . . 310

6 Tema 6: Rendiment i potència **311**
6.1 Rendiment . . . 311
6.2 Llei d'Amdahl . . . 313
6.3 Potència . . . 314

7 Tema 7: Memòria cau **316**
7.1 Disseny bàsic . . . 316
7.2 Rendiment . . . 317

7.3 Associativitat i multinivell . . . 319
7.4 Tipologia de les fallades . . . 321

8 Tema 8: Memòria virtual 323
8.1 Traducció d'adreces i fallada de pàgina . . . 323
8.2 TLB . . . 325
8.3 Protecció . . . 328
8.4 Integració del TLB i la memòria cau . . . 328

9 Tema 9: Excepcions i interrupcions 330
9.1 Excepcions i interrupcions . . . 330
9.2 Crides al sistema . . . 331
9.3 Entrada/sortida . . . 331
9.4 Fallada de pàgina . . . 332

III Solucions 333

1 Tema 1: Introducció 334
1.1 El cicle de vida del programa: Generació vs. execució . . . 334
1.2 Codificació de nombres naturals i enters . . . 335
1.3 Operacions aritmètiques en complement a 2 . . . 339

2 Tema 2: Instruccions i tipus de dades bàsics 342
2.1 Operands en registre . . . 342
2.2 Constants i immediats . . . 343
2.3 Operands en memòria . . . 344
2.4 Representació de naturals i enters . . . 347
2.5 Representació de caràcters . . . 348
2.6 Format de les instruccions . . . 349
2.7 Punters . . . 354
2.8 Vectors . . . 355
2.9 Cadenes de caràcters . . . 358

3 Tema 3: Traducció de programes 359
3.1 Operacions lògiques i desplaçaments . . . 359
3.2 Sentències condicionals i bucles . . . 362
3.3 Subrutines: introducció . . . 366
3.4 Subrutines: context i bloc d'activació . . . 369
3.5 Compilació, assemblatge, enllaçament i càrrega . . . 371

4 Tema 4: Aritmètica entera i matrius 373
4.1 Suma i resta . . . 373
4.2 Multiplicació . . . 376
4.3 Matrius . . . 377
4.4 Accés seqüencial . . . 380
4.5 Divisió . . . 387

5 Tema 5: Coma flotant 391
5.1 Representació IEEE 754 . . . 391
5.2 Operacions . . . 397
5.3 No-associativitat . . . 407

6 Tema 6: Rendiment i potència **414**
6.1 Rendiment . . . 414
6.2 Llei d'Amdahl . . . 418
6.3 Potència . . . 420

7 Tema 7: Memòria cau **423**
7.1 Disseny bàsic . . . 423
7.2 Rendiment . . . 427
7.3 Associativitat i multinivell . . . 429
7.4 Tipologia de les fallades . . . 433

8 Tema 8: Memòria virtual **436**
8.1 Traducció d'adreces i fallada de pàgina . . . 436
8.2 TLB . . . 438
8.3 Protecció . . . 440
8.4 Integració del TLB i la memòria cau . . . 441

9 Tema 9: Excepcions i interrupcions **444**
9.1 Excepcions i interrupcions . . . 444
9.2 Crides al sistema . . . 447
9.3 Entrada/sortida . . . 448
9.4 Fallada de pàgina . . . 450

IV Laboratori **452**

1 Laboratori 1: Introducció al simulador RARS **453**
Lliuraments . . . 453
Lectura prèvia . . . 453
1.1 El primer programa . . . 453
1.2 Exploració del simulador . . . 454
1.3 Execució i inspecció . . . 456
1.4 Punts de ruptura . . . 457
1.5 Depuració . . . 458

2 Laboratori 2: Assemblador RISC-V i tipus bàsics de dades **461**
Lliuraments . . . 461
Lectura prèvia . . . 461
2.1 Representació en memòria . . . 462
2.2 Pseudoinstruccions `la` i `li` . . . 465
2.3 Punters . . . 465
2.4 Vectors . . . 468
2.5 Cadenes de caràcters . . . 470

3 Laboratori 3: Traducció de programes **473**
Lliuraments . . . 473
Lectura prèvia . . . 473
3.1 Operacions lògiques i desplaçaments . . . 473
3.2 Sentències condicionals . . . 474
3.3 Bucles . . . 476
3.4 Subrutines . . . 477
3.5 Depuració de codi . . . 482

4 Laboratori 4: Matrius i subrutines **486**
Lliuraments . . . 486
Lectura prèvia . . . 486
4.1 Declaració de matrius . . . 486
4.2 Accés a elements . . . 488
4.3 Accés seqüencial . . . 491

5 Laboratori 5: Coma flotant i compilació separada **495**
Lliuraments . . . 495
Lectura prèvia . . . 495
5.1 Compilació separada a RARS . . . 495
5.2 Subrutina `descompon` . . . 499
5.3 Subrutina `compon` i programa complet . . . 503
5.4 Errors de precisió . . . 505

6 Laboratori 6: Memòria cau **507**
Lliuraments . . . 507
Lectura prèvia . . . 507
6.1 Descomposició d'adreces . . . 508
6.2 Recorregut seqüencial d'un vector . . . 509
6.3 Escriptures i polítiques d'escriptura . . . 513
6.4 Accés a múltiples estructures . . . 518
6.5 Optimització: *loop tiling* . . . 524

Calendari **531**

Referència tècnica **532**
RV32I . . . 532
Registres . . . 532
Format de les instruccions . . . 533
Instruccions . . . 533
Pseudoinstruccions . . . 536
RV32M . . . 538
Instruccions . . . 538
RV32F . . . 539
Registres . . . 539
Instruccions . . . 539
Pseudoinstruccions . . . 541
Zicsr . . . 541
Instruccions . . . 541
Pseudoinstruccions . . . 541
Excepcions i interrupcions . . . 542
RARS . . . 545
Mapa de memòria . . . 545
Directives . . . 546
Crides al sistema . . . 547

Sigles, símbols i notació **548**
Sigles . . . 548
Símbols . . . 550
Notació . . . 555
Termes . . . 557

Contribueix-hi **565**

Llicència i reconeixements **566**

Llicència 566

Reconeixements 566

Projecte d'innovació docent 566

Materials elaborats per professors d'EC 566

Intel · ligències artificials generatives 567

Programari 567

Presentació

Departament d'Arquitectura de Computadors

Estructura de Computadors (EC) és una assignatura **obligatòria** de 7,5 crèdits del Grau en Enginyeria Informàtica (GEI) de la Facultat d'Informàtica de Barcelona (FIB), impartida pel Departament d'Arquitectura de Computadors (DAC) de la Universitat Politècnica de Catalunya (UPC).[1]

L'assignatura tracta de l'organització bàsica en què s'estructura un computador per donar suport als llenguatges de programació d'alt nivell: tant els tipus de dades que utilitzen els programes com les sentències de control més habituals. L'eix vertebrador és la relació entre el **maquinari** i el **programari** a través del nivell d'abstracció de l'**arquitectura del repertori d'instruccions** (*instruction set architecture*, ISA).[2]

☑ Qualificació

◎ Nota final

La **nota final** (NF) s'obté a partir de:

- **Nota de teoria i problemes** (NT): 80 % de NF. Distribució:
  - Examen parcial (EP): $0{,}20 \cdot \max\{\text{EP}, \text{EF}\}$.
  - Examen final (EF): $0{,}60 \cdot \text{EF}$.
- **Nota de laboratori** (NL): 20 % de NF. Distribució:
  - Examen de laboratori (EL): 85 % de NL.
  - Avaluació continuada (AC): 15 % de NL.

$$\text{NF} = 0{,}20 \cdot \max\{\text{EP}, \text{EF}\} + 0{,}60 \cdot \text{EF} + 0{,}20 \cdot (0{,}85 \cdot \text{EL} + 0{,}15 \cdot \text{AC})$$

🕮 Examen parcial — temari

- Teoria: T1–T5.
- Laboratori: L1–L3.

[1] Totes les assignatures esmentades en aquests apunts són del Grau en Enginyeria Informàtica (GEI) de la Facultat d'Informàtica de Barcelona (FIB), si no s'especifica explícitament el contrari.

[2] A partir del curs 2025/26, l'arquitectura de referència de l'assignatura és **RISC-V** (variant **RV32I**, la base entera de 32 bits, i extensions **M**, **F** i **Zicsr**); anteriorment havia estat l'arquitectura *Microprocessor without Interlocked Pipeline Stages* (MIPS).

Examen final — temari

- Teoria: Tot.
- Laboratori: Tot.

Laboratori

- **Avaluació continuada**: mitjana de les notes de les 6 sessions.
  - **Treball previ**: Treball que s'ha de lliurar al Racó abans de l'inici de la sessió per **poder ser avaluat a la pràctica**. Consisteix a lliurar els fitxers de cada sessió. No s'exigeix que les respostes siguin totalment correctes, però sí suficientment completes per assegurar que durant la pràctica s'obtingui una nota positiva.
  - **Nota de la sessió**: L'avaluació es fa individualment **durant la pràctica** a través de la verificació de la validesa de les respostes als exercicis i de les respostes a les preguntes que pugui fer el professor. **És responsabilitat de l'estudiant d'avisar amb temps suficient el professor perquè l'avaluï.**
- **Examen de laboratori**: Es realitza **individualment** en un **ordinador dels laboratoris docents**. És responsabilitat de l'estudiant assegurar-se:
  - Abans de l'examen, de disposar d'almenys 10 MB d'espai lliure en la seva zona de disc per poder baixar i editar els fitxers de l'examen.
  - Durant l'examen, de lliurar els fitxers de la solució **dins del termini** temporal establert.
    - **No hi ha límit de lliuraments**. Sempre s'avaluarà només la darrera versió de cada fitxer.

Competència transversal

La competència transversal treballada a EC és **Sostenibilitat i Compromís Social** (**G2**), amb èmfasi en la relació entre rendiment i consum elèctric dels computadors. S'avalua mitjançant un problema específic de l'EF. La nota d'aquest problema també es té en compte per al càlcul de la nota de l'EF.

Material docent

Apunts, problemes i solucions

Cada tema, **T1**–**T9**, està format pel conjunt d'apunts, **A1**–**A9**, problemes, **P1**–**P9** i solucions **S1**–**S9**.

Taula 1: Temes de l'assignatura, amb els apunts, els problemes i les solucions de cadascun.

Tema	Apunts	Problemes	Solucions
Tema 1: Introducció	A1	P1	S1
Tema 2: Instruccions i tipus de dades bàsics	A2	P2	S2
Tema 3: Traducció de programes	A3	P3	S3
Tema 4: Aritmètica entera i matrius	A4	P4	S4
Tema 5: Coma flotant	A5	P5	S5
Tema 6: Rendiment i potència	A6	P6	S6
Tema 7: Memòria cau	A7	P7	S7
Tema 8: Memòria virtual	A8	P8	S8

Tema	Apunts	Problemes	Solucions
Tema 9: Excepcions i interrupcions	A9	P9	S9

Taula 2: Requadres de text: significat i rellevància.

Color	Icona	Tipus	Matèria avaluable	Contingut
		RISC-V	**Sí**	Referències tècniques
		EC	**Sí**	Criteris de l'assignatura
		Essencial	**Sí**	Contingut indispensable
		Exemple	**Sí**	Exemples il · lustratius
		Aprofundiment	No	Contingut opcional

Laboratori

El laboratori es numera per sessions (L1–L6).

Taula 3: Sessions del laboratori, amb els enunciats i les solucions de cadascuna.

Laboratori	Enunciats i solucions
Laboratori 1: Introducció al simulador RARS	L1
Laboratori 2: Assemblador RISC-V i tipus bàsics de dades	L2
Laboratori 3: Traducció de programes	L3
Laboratori 4: Matrius i subrutines	L4
Laboratori 5: Coma flotant i compilació separada	L5
Laboratori 6: Memòria cau	L6

Recursos per a l'estudi

El simulador (RARS)

El simulador del laboratori és el *RISC-V Assembler and Runtime Simulator* (RARS).

- **Indispensable** per al laboratori.
- **Recomanat** per a l'assignatura en general.
  - Davant de qualsevol dubte relatiu al llenguatge d'assemblador, és recomanable desenvolupar un exemple mínim (*minimal working example*) i experimentar-hi.
  - Als materials, hi ha fragments de codi que poden servir d'orientació.

Treballar amb codi de manera habitual és una manera molt efectiva d'aclarir i consolidar conceptes.

Instal · lació

1. Descarregueu i guardeu el fitxer `rars1_6.jar` de la versió de referència (referències tècniques).
2. Descarregueu i instal · leu *Java Runtime Environment* (JRE) versió 8.0 o superior.

Inicialització

1. Obriu un terminal (GNU/Linux: Ctrl+Alt+T).
2. Aneu al directori de `rars1_6.jar` (per exemple amb l'ordre `cd <DIR_FULL_PATH>`).
3. Executeu: `java -jar rars1_6.jar`.

El compilador (GNU/GCC)

El compilador de C de referència és el de la *GNU Compiler Collection* (GCC).

- **Recomanat** per a l'assignatura en general.
  - Pels mateixos motius que per al codi d'assemblador.
  - Als materials, també hi ha fragments de codi com a orientació.

El Racó

- Avisos/Notícies
  - Convocatòria d'exàmens
  - Comunicacions dels professors i distribució de material (diapositives, problemes addicionals, etc.)
- Laboratori
  - Estudis previs
  - Examen
- Exàmens d'anys anteriors
  - Enunciats
  - Solucions

Pàgina de l'assignatura

- Grups
- Professors
- Dates de laboratori

Guia docent (FIB)

- Competències
- Objectius
- Rol de l'assignatura dins del pla d'estudis

Referències tècniques

Les referències següents estableixen els criteris tècnics segons els quals s'avaluarà.

! EC 0.1: Referències tècniques de l'assignatura

Taula 4: Llenguatges, eines i estàndards de referència.

Ítem	Nom	Estàndard
Llenguatge d'alt nivell	C23 (ISO/IEC 9899:2024)	[1]
Compilador C[1]	GCC 16	[2]
Llenguatge d'assemblador	RISC-V de 32 bits: RV32I, M, F i Zicsr (especificació de l'ISA v20260120)	[8]
Simulador[1]	RARS 1.6 (32 bits)	[9]
Cadena d'eines	`riscv32-unknown-elf-gcc` (RISC-V GNU Compiler Toolchain)	[10]
Sistema operatiu	GNU/Linux	Qualsevol distribució generalista

[1] En cas de conflicte RARS preval sobre el compilador.

Bibliografia

[1] ISO/IEC JTC1/SC22/WG14, «ISO/IEC 9899:2024 — Information technology — Programming languages — C», International Organization for Standardization, ISO/IEC 9899:2024, oct. 2024. Disponible a: https://www.open-std.org/jtc1/sc22/wg14/

[2] GCC Developers, «GCC 16 Release Series». https://gcc.gnu.org/gcc-16/, 2026.

[3] RISC-V International, «RV32I Base Integer Instruction Set, Version 2.1», RISC-V International, gen. 2026. Disponible a: https://docs.riscv.org/reference/isa/unpriv/rv32.html

[4] RISC-V International, «Control and Status Registers (CSRs)», RISC-V International, gen. 2026. Disponible a: https://docs.riscv.org/reference/isa/priv/priv-csrs.html

[5] RISC-V International, «"M" Standard Extension for Integer Multiplication and Division, Version 2.0», RISC-V International, gen. 2026. Disponible a: https://docs.riscv.org/reference/isa/unpriv/m-st-ext.html

[6] RISC-V International, «"F" Standard Extension for Single-Precision Floating-Point, Version 2.2», RISC-V International, gen. 2026. Disponible a: https://docs.riscv.org/reference/isa/unpriv/f-st-ext.html

[7] RISC-V International, «"Zicsr" Extension for Control and Status Register (CSR) Instructions, Version 2.0», RISC-V International, gen. 2026. Disponible a: https://docs.riscv.org/reference/isa/unpriv/zicsr.html

[8] RISC-V International, «RISC-V Assembly Programmer's Manual». https://github.com/riscv-non-isa/riscv-asm-manual, 2024.

[9] B. Landers, «RARS: RISC-V Assembler and Runtime Simulator, v1.6». https://github.com/TheThirdOne/rars/tree/v1.6, 2023.

[10] RISC-V Collab, «RISC-V GNU Compiler Toolchain». https://github.com/riscv-collab/riscv-gnu-toolchain, 2024.

[11] D. A. Patterson i J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*, 2nd ed. Morgan Kaufmann, 2020.

[12] S. L. Harris i D. M. Harris, *Digital Design and Computer Architecture: RISC-V Edition*. Morgan Kaufmann, 2021.

[13] G. M. Amdahl, «Validity of the single processor approach to achieving large scale computing capabilities», dins *AFIPS Spring Joint Computer Conference*, AFIPS, 1967, pp. 483-485.

[14] F. Fritz i JmsDoug, «Intel Core i9-13900K Labelled Die Shot». Wikimedia Commons, 2023. Disponible a: https://commons.wikimedia.org/wiki/File:Intel_Core_i9-13900K_Labelled_Die_Shot.jpg

[15] J. M. Parcerisa, «Tema 1: Introducció». Material docent intern. Departament d'Arquitectura de Computadors, FIB-UPC, març de 2020.

[16] J. M. Parcerisa, «Tema 2: Instruccions i tipus de dades bàsics». Material docent intern. Departament d'Arquitectura de Computadors, FIB-UPC, febrer de 2022.

[17] J. M. Parcerisa, R. Tous, i J. Tubella, «Tema 3: Traducció de Programes». Material docent intern. Departament d'Arquitectura de Computadors, FIB-UPC, setembre de 2019.

[18] J. M. Parcerisa, «Tema 4: Matrius». Material docent intern. Departament d'Arquitectura de Computadors, FIB-UPC, març de 2016.

[19] J. M. Parcerisa, «Tema 5: Aritmètica d'enters i coma flotant». Material docent intern. Departament d'Arquitectura de Computadors, FIB-UPC, març de 2020.

[20] J. M. Parcerisa i R. Tous, «Tema 6: Memòria Cache». Material docent intern. Departament d'Arquitectura de Computadors, FIB-UPC, febrer de 2023.

[21] J. M. Parcerisa, R. Tous, i J. Tubella, «Tema 7: Memòria Virtual». Material docent intern. Departament d'Arquitectura de Computadors, FIB-UPC, juny de 2018.

[22] J. M. Parcerisa, «Tema 8: Excepcions i Interrupcions». Material docent intern. Departament d'Arquitectura de Computadors, FIB-UPC, juny de 2018.

[23] E. Castillo *et al.*, «Quadern de laboratori, Sessió 0: Introducció». Material docent intern. Departament d'Arquitectura de Computadors, FIB-UPC, 2015.

[24] E. Castillo *et al.*, «Quadern de laboratori, Sessió 1: Assemblador MIPS i tipus bàsics de dades». Material docent intern. Departament d'Arquitectura de Computadors, FIB-UPC, 2015.

[25] E. Castillo *et al.*, «Quadern de laboratori, Sessió 2: Traducció de Programes». Material docent intern. Departament d'Arquitectura de Computadors, FIB-UPC, 2015.

[26] E. Castillo *et al.*, «Quadern de laboratori, Sessió 3: Tipus de dades estructurats». Material docent intern. Departament d'Arquitectura de Computadors, FIB-UPC, 2015.

[27] E. Castillo *et al.*, «Quadern de laboratori, Sessió 4: Codificació en coma flotant». Material docent intern. Departament d'Arquitectura de Computadors, FIB-UPC, 2015.

[28] E. Castillo *et al.*, «Quadern de laboratori, Sessió 5: Memòria cache». Material docent intern. Departament d'Arquitectura de Computadors, FIB-UPC, 2015.

[29] M. Fernández, D. López, J. M. Parcerisa, A. Toribio, R. Tous, i J. Tubella, «Col · lecció de Problemes». Material docent intern. Departament d'Arquitectura de Computadors, FIB-UPC, 2015.

[30] M. Fernández, D. López, J. M. Parcerisa, A. Toribio, R. Tous, i J. Tubella, «Solucionari de la Col · lecció de Problemes». Material docent intern. Departament d'Arquitectura de Computadors, FIB-UPC, 2015.

Bibliografia complementària

- RISC-V Reader: An Open Architecture Atlas (David Patterson i Andrew Waterman, 2017). Versió en castellà: RISC-V Reader: Un atlas de arquitectura abierta (David Patterson y Andrew Waterman, 2017. Traducció d'Alí Lemus, Eduardo Corpeño).

Part I

Apunts

Tema 1: Introducció

1.1 La gestió de la complexitat: capes i interfícies

Un **computador** és un sistema d'una complexitat extraordinària, però que, en el seu nivell més fonamental, només és capaç d'executar **operacions** aritmètiques, lògiques i de control **elementals**. La distància conceptual que separa aquestes instruccions bàsiques de les aplicacions tan sofisticades actuals és tan gran que és impossible dissenyar qualsevol sistema operatiu (SO) o programa modern que interactuï directament amb els components físics del processador.

1.1.1 Nivells d'abstracció

Per gestionar aquesta distància conceptual tan gran i fer viable el disseny de sistemes computacionals, s'aplica el mecanisme de l'**abstracció**. Aquesta estratègia permet estructurar el conjunt del computador, entès com la unió de **maquinari** (***hardware***) i **programari** (***software***), en una **jerarquia de nivells** clarament diferenciats. En aquest model, cada nivell ofereix una descripció simplificada del nivell immediatament inferior, ocultant-ne la complexitat interna i definint interfícies precises que permeten la comunicació entre els diferents components del sistema.

- **Nivell de components físics**: Constitueix el fonament material del computador. Inclou els elements de l'electrònica de base com transistors, resistències, condensadors, díodes, interconnexions físiques, etc.
- **Nivell d'unitats funcionals i lògica digital**: En aquest estadi s'agrupen els components per formar portes lògiques, descodificadors, sumadors, comparadors, busos, etc. Aquest nivell d'abstracció és l'objecte d'estudi de l'assignatura **Introducció als computadors** (IC).
- **Nivell de microarquitectura**: És la implementació específica de l'arquitectura del repertori d'instruccions, que és el nivell següent. Defineix com s'organitzen i s'interconnecten les unitats funcionals (com la unitat aritmeticològica, els busos o les memòries cau) i com es dissenya el camí de dades (*datapath*) per executar les instruccions. Dos processadors poden implementar el mateix repertori d'instruccions i comportar-se de manera idèntica per al programador, però tenir microarquitectures completament diferents (per exemple, Intel i AMD amb x86).
- **Nivell de l'arquitectura del repertori d'instruccions** (*instruction set architecture*, ISA): És la **frontera** crítica entre el **maquinari** i el **programari**. Estableix, entre d'altres, el conjunt d'instruccions que la unitat central de processament (*central processing unit*, CPU) és capaç d'executar, el model de memòria i els registres visibles per al programador. A **Estructura de computadors** (EC), l'**ISA** de referència és **RISC-V** (vegeu EC 0.1).
- **Nivell de sistema (baix nivell)**: És l'àmbit d'estudi principal d'**EC**. Aquí el programari té un control directe sobre els recursos definits per l'ISA. Aquest nivell inclou:
  - **Llenguatge màquina**: Instruccions codificades en binari que la unitat de control del processador interpreta i executa.
  - **Llenguatge d'assemblador**: Una representació simbòlica del llenguatge màquina que utilitza **mnemònics**. Un programa anomenat **assemblador** s'encarrega de fer la **traducció directa** a codi **binari**.

- **Nivell d'aplicació (alt nivell)**: Nivell on es desenvolupa el programari d'usuari utilitzant llenguatges d'alt nivell com el **C** (**el de referència a EC**), C++ o Rust. Un programa anomenat **compilador** tradueix el codi d'alt nivell (codi font) a instruccions d'assemblador. L'abstracció d'aquest nivell permet al programador gestionar dades i algorismes sense haver de conèixer els detalls de la microarquitectura o el mapa de registres específic.
- **Nivell d'usuari**: Representa el punt de contacte amb l'usuari final, que interacciona amb el sistema mitjançant aplicacions i la **interfície d'usuari** (*user interface*, UI), ja sigui a través de línies d'ordres o d'entorns gràfics.

1.1.2 Interfícies: els contractes del sistema

L'abstracció només és efectiva si les interfícies (fronteres) actuen com a contractes rígids que cap de les parts pot violar sense trencar la funcionalitat del sistema. En un entorn **allotjat** (***hosted***) amb **SO** —el cas més habitual, per exemple el de Programació I (PRO1)—, n'hi ha tres:

- *Application programming interface* (API): Contracte en l'àmbit del codi font. Garanteix la portabilitat del programari entre diferents sistemes si es respecten els estàndards, com l'**ISO C**, el C de la *International Organization for Standardization* (ISO), o la *Portable Operating System Interface* (POSIX). **Defineix *com* s'ha d'escriure el codi**.
- *Application binary interface* (ABI): Contracte en l'àmbit del binari. És el protocol que defineix com interactuen les peces ja compilades. Estableix les convencions de crida (pas de paràmetres per registres), la gestió de la pila (*stack*) i el format dels fitxers executables. L'ABI depèn del binomi **processador-SO** (p. ex. GNU/Linux sobre RISC-V). **Defineix com parlen els binaris**.
- **Arquitectura del repertori d'instruccions** (*instruction set architecture*, ISA): Contracte en l'àmbit del maquinari. Defineix el vocabulari d'instruccions, els registres visibles per al programador i el model de memòria. **Defineix què sap fer el processador**.

> **!** EC 1.1
>
> Els callouts **EC**, com aquest, recullen els **criteris i les convencions de l'assignatura**: la notació, el format del codi i els termes. **No seguir una convenció no penalitza l'avaluació**, tret que l'enunciat ho indiqui explícitament.

> **!** EC 1.2
>
> A **EC**, s'assumeix que es **treballa directament sobre el processador**, tret que s'indiqui el contrari. Per tant, recursos vistos en aquest tema, com ara les API o l'SO, no estan disponibles per defecte.
> El **RARS** és una **excepció important** d'aquesta assumpció, ja que disposa d'un simulador de **crides al sistema** (*syscall*), un recurs bàsic de qualsevol SO, el qual es fa servir en algunes sessions de laboratori.

1.2 El cicle de vida del programa: generació vs. execució

Per entendre la relació entre les interfícies (**API**, **ABI**, **ISA**), cal diferenciar dues etapes totalment separades en el temps i, sovint, en l'espai (màquines diferents o, fins i tot, arquitectures diferents).

1.2.1 El flux de generació: la visió del programador

El procés de transformació d'una idea abstracta en un fitxer executable no és un pas directe, sinó una **cadena d'eines** (***toolchain***) altament coordinada on intervenen diferents eines de sistema. Aquest flux, que va des de l'escriptura del codi fins a la generació del binari executable, es pot desglossar en quatre (més una) etapes fonamentals on les interfícies **API**, **ABI** i **ISA** (vegeu Secció 1.1.2) actuen com a garanties de compatibilitat:

0. **Desenvolupament**: El programador escriu el codi font (`.c`, `.h`) d'acord amb l'**API**.
1. **Preprocessament** (`cpp`): El preprocessador és el primer a actuar. Neteja el text i resol directives (com `#include` o `#define`) per generar el codi expandit (`.i`), una versió completa però encara en llenguatge d'alt nivell.
2. **Compilació** (`cc1`): El compilador tradueix la lògica de l'alt nivell a un llenguatge de baix nivell anomenat assemblador (`.s`). En aquest punt, el compilador actua com a pont: ha d'entendre l'**API** escrita pel programador i, alhora, començar a aplicar les regles de l'**ABI** per determinar com s'estructuraran les dades en memòria.
3. **Assemblatge** (`as`): L'assemblador agafa el fitxer de text (`.s`) i el converteix en codi objecte (`.o`). Aquí és on l'**ISA** pren protagonisme, ja que el programari s'ha de traduir a les instruccions específiques del processador diana, com x86 o *Advanced RISC Machines* (ARM), mantenint l'estructura binària que dicta l'**ABI**.
4. **Enllaçament** (`ld`, *linking*): L'enllaçador és el pas final i crític de síntesi. La seva funció és combinar el codi objecte generat pel programador amb les biblioteques del sistema, tant estàtiques (`.a`) com dinàmiques (`.so`). Per fer-ho, l'enllaçador ha de tenir un coneixement profund de l'**ISA** (per resoldre salts i adreces de memòria) i de l'**ABI** (per assegurar que totes les peces «parlen» el mateix llenguatge binari), i produeix l'**executable final** llest per ser carregat a la memòria.

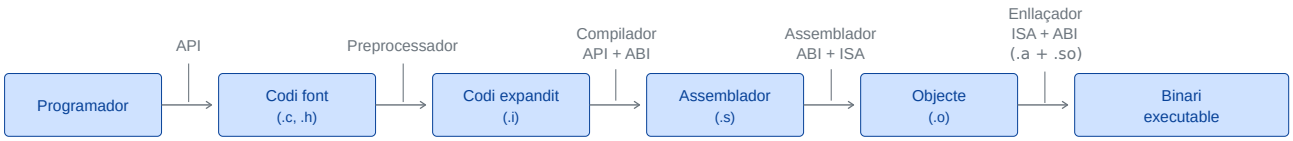


Figura 1.1: El flux de generació del programari: les quatre etapes de la cadena d'eines de la *GNU Compiler Collection* (GCC).

> Exemple 1.1: GCC i Clang
>
> La **GCC toolchain** és l'estàndard *de facto* de la indústria per a sistemes oberts. És un projecte de codi obert que ha evolucionat durant dècades, oferint un suport ampli per a múltiples arquitectures i estàndards de llenguatge.
> La **Clang toolchain** és una alternativa moderna que ha guanyat popularitat per la seva arquitectura modular i per uns missatges de diagnòstic més clars.

1.2.2 El flux d'execució: la visió del processador

Un cop obtingut el binari, aquest s'ha d'executar. El camí que segueixen les dades depèn de l'entorn de treball:

1.2.2.1 Entorn allotjat (*hosted*)

És el model d'un ordinador personal convencional. El binari no és autònom:

1. El **sistema operatiu** carrega el binari a la memòria.
2. El binari s'executa i, quan necessita interactuar amb l'exterior, fa una crida al sistema (`ecall`, *environment call*; Secció 9.5), seguint les convencions de l'**ABI**.
3. L'**SO** rep la petició i utilitza l'**ISA** (instruccions privilegiades) per gestionar el maquinari.

1.2.2.2 Entorn autònom (*bare-metal*)

L'**entorn autònom** (*bare-metal*) és el **model** dels **sistemes encastats** (*embedded systems*) simples i el d'**EC**:

1. El binari es carrega directament a la memòria del processador (o via simulador).
2. El programa té el control total i executa les instruccions de l'**ISA** sense intermediaris.
3. Aquí l'**ABI** no serveix per parlar amb un SO, sinó per garantir que el codi és compatible amb les biblioteques i per interactuar amb el **simulador** (per exemple, mitjançant `ecall` per imprimir per consola).

1.2.3 El rol de les interfícies

Les interfícies defineixen el binari durant la **generació**, però també regeixen el seu comportament durant l'**execució**. Són el contracte que garanteix que el binari i la màquina (o simulador) s'entenguin en temps real.

1.2.3.1 Les interfícies en la «visió del programador» (generació)

Des del punt de vista del programador, les interfícies actuen com a **normes de construcció**.

- **API**: El programador escriu `printf()`. El compilador sap què vol dir perquè l'API ho defineix.
- **ABI**: El compilador decideix posar el primer argument a `a0` perquè l'ABI de RISC-V ho mana.
- **ISA**: L'assemblador codifica la suma `add a0, a0, a1` (Exemple 1.3) com a `0x00B50533` perquè l'ISA ho especifica.

1.2.3.2 Les interfícies en la «visió del processador» (execució)

És al processador on les interfícies esdevenen el **protocol de funcionament real**:

- **L'ABI en execució**: Quan el programa executa `ecall`, el processador cedeix el control a l'entorn d'execució (l'SO o, a EC, el simulador), que llegeix el codi de servei del registre `a7` i l'interpreta segons la convenció de crides al sistema de l'**ABI**. Si la convenció diu que el codi `1` és «imprimir un enter» (és el cas de RARS), el sistema l'imprimeix. L'ABI és el que permet que el binari i l'entorn es basin en la mateixa «llengua» mentre el programa s'executa.
- **L'ISA en execució**: És el moment de la veritat. La unitat de control del processador llegeix els bits del binari. Si aquests bits no coincideixen amb l'**ISA** del maquinari, el processador no sabrà quines portes lògiques obrir i el sistema fallarà (instrucció il · legal).

La Taula 1.1 resumeix l'impacte que tenen els canvis a cada una de les interfícies.

Taula 1.1: Impacte del canvi d'interfícies

Si canvia la...	El codi s'ha de...	Perquè...
API	Modificar el codi i recompilar	Han canviat les interfícies de les funcions d'alguna biblioteca usada.
ABI	Recompilar	El binari ha de canviar com gestiona els registres o les crides al sistema.
ISA	Recompilar	El processador nou no entén els «zeros i uns» de l'antic.

1.2.4 La dualitat de l'entorn: amfitrió vs. *target*

En el desenvolupament de sistemes de baix nivell i, especialment, en els encastats, on el dispositiu que executarà el codi sol mancar dels recursos necessaris per allotjar eines de programació, cal diferenciar clarament entre dos ordinadors que fan papers diferents:

1. **Màquina amfitriona** (***host***):
   - **Què és**: L'ordinador que utilitza el programador per escriure el codi (per exemple, un ordinador de la facultat o un portàtil).
   - **Arquitectura**: Habitualment, **x86-64** o **ARM**.
   - **Funció**: Executar l'editor de text i el **compilador**. El compilador és un programa «nadiu» de l'amfitrió que s'encarrega de «fabricar» el binari.
2. ***Target*** (la màquina destí):
   - **Què és**: El sistema on realment s'executarà el programa final.
   - **Arquitectura**: A **EC**, l'**ISA** de destí és **RISC-V**.
   - **Funció**: Executar el binari generat per l'amfitrió. En aquest cas, el *target* no és una placa física (inicialment), sinó un **simulador** (com el RARS) que reprodueix el comportament d'una CPU RISC-V sobre l'amfitrió.

1.2.4.1 Per què és important aquesta separació?

- **Incompatibilitat de binaris**: Un binari generat per a RISC-V és una seqüència de bits inintel · ligible per a un processador Intel/AMD. Si s'intenta executar-lo directament al terminal de Linux/Windows, el sistema operatiu advertirà que el format de l'executable és invàlid (tret que l'amfitrió tingui configurat un emulador, com `qemu-user`, que el sistema invoqui automàticament per a aquest format).
- **El paper del simulador**: El simulador actua com una «bombolla» de l'arquitectura de destí. El simulador s'executa en l'amfitrió (x86-64 o ARM), però l'interior de la bombolla entén l'**ISA** i l'**ABI** de RISC-V.
- **Portabilitat del codi font**: El codi en C és l'únic element portable entre tots dos mons: es pot compilar el mateix fitxer `.c` per a l'amfitrió (per provar la lògica) o fer una **compilació creuada** (*cross-compilation*) per al *target* (Secció 1.2.4.2).

1.2.4.2 Compilació creuada (*cross-compilation*)

S'utilitza un entorn allotjat, per exemple, amb arquitectura **x86** i sistema **Linux**, per generar un binari destinat a una arquitectura diferent (per exemple, **RISC-V**). El compilador s'executa en l'**amfitrió**, però genera codi per al ***target***.

> 💡 Exemple 1.2: RISC-V GNU Compiler Toolchain
>
> La RISC-V GNU Compiler Toolchain permet crear codi des de diversos amfitrions GNU/Linux o macOS per a moltes ISA RISC-V.

Un cop obtingut a l'amfitrió, el binari es carrega al *target* mitjançant un cable i un protocol apropiat.

1.2.4.3 Depuració creuada (*cross-debugging*)

La depuració s'ha de fer en l'entorn del *target*, el qual pot ser:

- **Amb simulador**: Un programa (el simulador) en un ordinador personal recrea per programari el comportament dels registres i la memòria de la CPU de destí i fa visibles els efectes sobre els components del maquinari simulat a través d'interfícies gràfiques. És el cas del **RARS**, el **simulador d'EC**.
- **Amb *target* real**: El binari s'envia des de l'amfitrió al xip real (el *target*) a través d'una sonda de depuració (*debug probe*). D'aquesta manera, el codi s'executa en l'entorn real, amb els seus temps de resposta i perifèrics físics: díodes emissors de llum (*light-emitting diodes*, LED), sensors i motors.

1.2.5 El llenguatge C: el pont de l'abstracció

A **EC**, el llenguatge **C** és l'eina de referència perquè ocupa una posició estratègica única: és prou «alt» per ser llegible, però prou «baix» per reflectir fidelment el que passa a l'**ISA**.

1.2.5.1 Origen i dualitat

Creat per Dennis Ritchie als laboratoris Bell (anys 70) per reescriure el sistema operatiu **UNIX**, el C va néixer amb una missió: **substituir el llenguatge d'assemblador**.

- **Com a llenguatge d'alt nivell** (API): Ofereix estructures de control (`if`, `while`) i tipus de dades que permeten programar sense pensar en processadors concrets.
- **Com a «assemblador portàtil»** (ABI/ISA): Permet manipular directament adreces de memòria mitjançant **punters**. Una variable en C té una correspondència quasi directa amb una posició de memòria o un registre de l'ISA.

1.2.5.2 Control del maquinari amb C

A diferència de llenguatges com Java o Python, el C permet accedir a capes molt més pròximes al maquinari:

1. **Gestió de memòria**: Mitjançant els punters, el programador decideix exactament on resideixen les dades. Això és vital en sistemes **autònoms** per accedir a registres de control del maquinari que estan «mapats» en memòria (*Memory-Mapped I/O*).
2. **Alineament i estructures**: El C permet definir com s'empaqueten els bits (estructures `struct`). L'**ABI** determina com s'ordenen aquests bytes en memòria, i el C té les eines per respectar aquest ordre.
3. **Eficiència**: Un bon compilador tradueix el C a instruccions de l'**ISA** de manera extremadament eficient.

1.2.5.3 La dependència de l'entorn

Tot i que el codi C és portable, el seu comportament final depèn de l'**ABI**:

- La mida d'un `int` (16, 32 o 64 bits) no l'estableix el llenguatge, sinó l'ABI del sistema on s'executarà el programa.
- El pas de paràmetres a les funcions el gestiona el compilador seguint l'ABI, però el programador de C ha de ser conscient d'aquestes regles quan interactua amb codi escrit directament en **assemblador**.

1.2.6 Traducció: de C al llenguatge màquina

El procés de traducció no és un pas únic, sinó una cadena de transformacions, la cadena d'eines, on cada etapa acosta el codi una mica més al maquinari (vegeu les quatre etapes a Secció 1.2.1 i la visió general a Secció 1.1.2 i Figura 1.1). **El compilador de referència a EC és el GCC**, amb les eines `cpp` (preprocessament), `cc1` (compilació), `as` (assemblatge) i `ld` (enllaçament).

1.2.6.1 El llenguatge d'assemblador: la visió de l'ISA

El llenguatge d'assemblador és rellevant a EC perquè és la representació textual de l'**ISA**: cada instrucció d'assemblador correspon a una instrucció de llenguatge màquina (llevat de les pseudoinstruccions, que l'assemblador expandeix). La **compilació** hi és rellevant perquè és el pas que tradueix el C a aquest llenguatge (Secció 1.2.1).

Exemple 1.3: Traducció de **C** a **assemblador RISC-V**

C

```c
int suma(int a, int b) {
    return a + b;
}
```

RV32I

```
# L'ABI de RISC-V diu que els arguments van a a0 i a1
suma:
        add     a0, a0, a1  # Instrucció de l'ISA: a0 <- a0 + a1
        ret                 # Pseudoinstrucció: retorna el control al cridant
```

En aquest exemple, el compilador ha seguit les regles de l'**ABI** per decidir que els paràmetres `a` i `b` es passen a través dels registres `a0` i `a1` (registres d'argument). La instrucció `add` és una operació definida per l'**ISA** de RISC-V que realitza la suma i en deixa el resultat a `a0` (registre de retorn). Finalment, la instrucció `ret` és una pseudoinstrucció (Secció 2.3.5.2) que s'expandeix a `jalr zero, 0(ra)` (`jalr`, *jump and link register*): un salt indirecte (Secció 3.4.3) a l'adreça de retorn continguda al registre `ra`.

⚠ Aprofundiment 1.1: Treballant amb maquinari real: Raspberry Pi Pico 2 + Raspberry Pi Debug Probe

Un **microcontrolador** és un sistema informàtic complet integrat en un sol xip de silici, dissenyat específicament per controlar tasques concretes en sistemes encastats. A diferència d'un **microprocessador** convencional (com el d'un ordinador personal), que només conté la CPU i necessita components externs per funcionar, el microcontrolador incorpora dins del mateix encapsulat la memòria —una *random access memory* (RAM) i, sovint, memòria flaix (*flash*)—, els circuits de rellotge i una gran varietat de perifèrics d'entrada/sortida (E/S), com *general-purpose input/output* (GPIO), *universal asynchronous receiver-transmitter* (UART) o *analog-to-digital converter* (ADC).
El **RP2350** és un microcontrolador de *Raspberry Pi* (RP) de tipus *system-on-chip* (SoC) que inclou dos nuclis ARM Cortex-M33 i dos nuclis **Hazard3** basats en l'**ISA RISC-V**. El programador tria l'arquitectura en compilar el programa, i la memòria d'arrencada, una *read-only memory* (ROM), activa els nuclis que corresponen a la imatge que hi troba.
La **Raspberry Pi Pico 2** és una placa de desenvolupament basada en el xip **RP2350**. Com a plataforma d'experimentació exposa els pins del microcontrolador per connectar-hi sensors, actuadors i depuradors.
La **Raspberry Pi Debug Probe** és una sonda de depuració (***debug probe***). La interfície *Serial Wire Debug* (SWD) li permet controlar la CPU del *target*, carregar-hi binaris i inspeccionar registres o memòria en temps real. L'estàndard **CMSIS-DAP**, compatible amb la majoria d'eines de depuració de codi obert, com OpenOCD, al qual es connecta el *GNU Debugger* (GDB), li permet la interacció amb l'ordinador del desenvolupador.

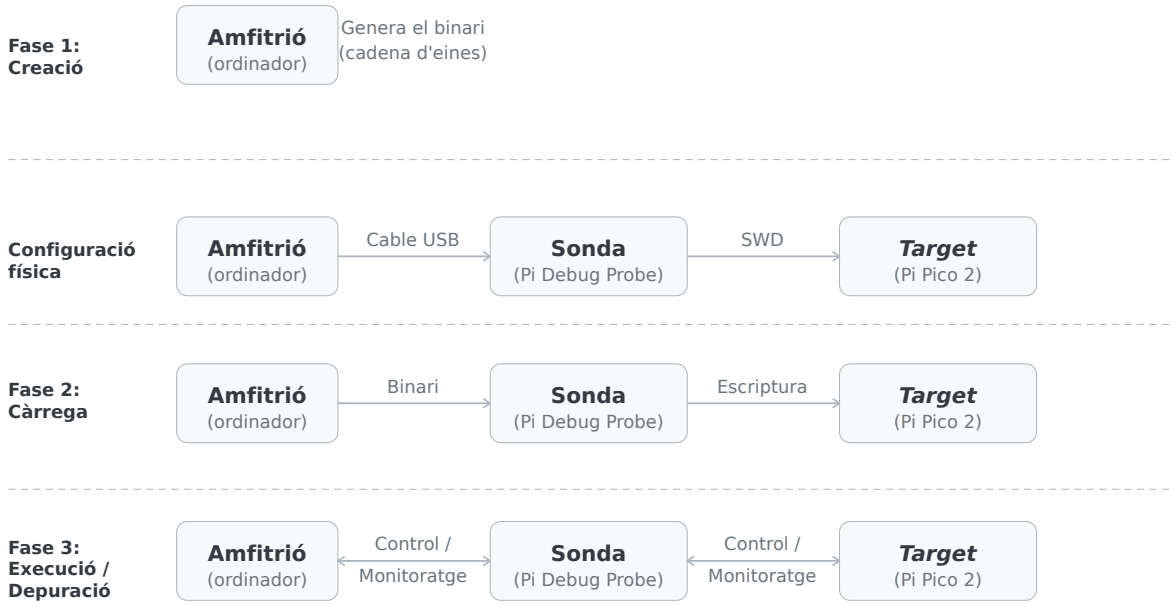


Figura 1.2: Configuració física i fases d'ús del conjunt Amfitrió + Sonda (*Pi Debug Probe*) + *Target* (*Pi Pico 2*).

- **Amfitrió** (ordinador):
  - Executa el programa **depurador** (GDB/OpenOCD), que, a través de CMSIS-DAP, dona **ordres** a la sonda i en rep les **respostes**.
- **Sonda** (**Pi Debug Probe**):

  - Actua de **pont**. Tradueix els missatges de l'amfitrió a senyals elèctrics de baix nivell que el *target* pot entendre i viceversa.

- **Target** (**Pi Pico 2**):
  - És on hi ha el processador RISC-V que **executa** el binari.

Fase 1: Creació

- **Amfitrió** (ordinador):
  - Genera el binari (riscv-gnu-toolchain).

Fase 2: Càrrega (***Flashing*** / ***Deployment***)

- **Amfitrió** (ordinador):
  - El depurador envia el binari a la sonda.
- **Target** (**Pi Pico 2**):
  - Rep el binari i l'emmagatzema a la **memòria flaix** (***flash***), que és permanent.

Fase 3: Execució / Depuració (***Runtime*** / ***Debugging***)

- **Amfitrió** (ordinador):
  - Envia ordres com «Atura't (`break`)», «Executa la instrucció següent (`stepi`)» o «Escriu el valor `5` al registre `a0`».
- **Target** (**Pi Pico 2**):
  - Respon enviant la informació en temps real: «Estic aturat a l'adreça `0x00000100`», «El registre `t0` val `42`» o «S'ha produït un error de memòria».

1.2.7 C vs. C++

La majoria d'estudiants arriben a EC havent après **C++**, el llenguatge de referència a PRO1. Tot i que el C++ és un descendent directe del C i en conserva gairebé tota la sintaxi bàsica, la seva orientació és radicalment diferent de la requerida a **EC**.

1.2.7.1 Diferències fonamentals: què es «perd» en passar a C?

El pas de C++ a C és un exercici de **simplificació**. En C no existeixen les capes d'abstracció d'alt nivell que faciliten el desenvolupament de programari complex:

- **Sense classes ni objectes**: El C és un llenguatge **procedimental**. No hi ha `class`, `public`, `private` ni herència. Les dades s'agrupen en `struct` (estructures de dades) i les funcions operen sobre elles de manera externa.
- **Sense la biblioteca estàndard de plantilles** (*Standard Template Library*, STL): No hi ha `std::vector`, `std::stack` o `std::map`. En C, si, per exemple, es vol una llista enllaçada, cal programar-la gestionant els punters manualment.
- **Gestió manual de l'entrada/sortida** (E/S): No hi ha els **fluxos** (*streams*) de C++ (`cin >>`, `cout <<`), sinó funcions de biblioteca com `scanf` i `printf`. Això obliga a especificar el **format de**

les dades (`%d` per a enters, Exemple 1.4), el qual està íntimament lligat a la manera com es representen en bits.
- **Gestió explícita de la memòria**: En C++ sovint la memòria s'allibera automàticament (destructors). En C, cada `malloc()` (reserva) requereix un `free()` (alliberament) manual.

1.2.7.2 Per què C i no C++ a Estructura de computadors?

A **arquitectura de computadors** s'usa **C** com a referència perquè dona un **accés al maquinari** molt més **transparent**:

1. **Correspondència directa amb l'ISA**: El C++ introdueix mecanismes com el polimorfisme o la gestió automàtica d'objectes (constructors i destructors) que generen molt «codi ocult» en el binari. En C, gairebé cada línia de codi té una traducció directa i previsible a **instruccions de l'ISA**. És el llenguatge perfecte per aprendre com funciona el processador per dins.
2. **Control de l'ABI**: L'**ABI** de C és l'estàndard universal. Gairebé tots els sistemes operatius i simuladors estan dissenyats per entendre's amb el binari que genera el C. El C++ té una ABI molt més complexa (especialment pel que fa als noms de funcions) que dificultaria molt les pràctiques de baix nivell.
3. **Determinisme**: En sistemes de baix nivell (com els **encastats** o els **autònoms**), cal saber exactament quant de temps triga una operació i quanta memòria ocupa. El C no té «sorpreses» en temps d'execució; el C++ pot tenir sobrecàrregues ocultes que un microcontrolador petit no podria gestionar.
4. **Accés al maquinari**: El C permet tractar una adreça de memòria com si fos una variable. Això és el que es fa per parlar amb els perifèrics del simulador.

💡 Exemple 1.4: L'abstracció vs. la transparència

Imprimir un nombre enter en base decimal per pantalla pot semblar una tasca trivial, però implica diversos conceptes d'**ABI** i **format de dades**:
En C++, l'èmfasi en l'abstracció del flux:

C++

```
int n = 42;
std::cout << n;
```

- **Què passa aquí?** El programador no diu *com* és el nombre. L'objecte `cout` «endevina» que `n` és un enter (gràcies a la sobrecàrrega d'operadors) i decideix internament com transformar els bits de `n` en caràcters de l'*American Standard Code for Information Interchange* (ASCII).
- **Problema per a EC**: Hi ha molta «màgia» oculta. Per exemple, tant el pas de paràmetres com la gestió dels tipus de dades als registres resten ocults.

En C, l'èmfasi en la transparència del format:

C

```
int n = 42;
printf("%d", n);
```

- **Què passa aquí?** C obliga a ser explícit:
    1. **`"%d"`**: Indica a la funció exactament com ha d'interpretar els bits. Com a enter en decimal, en aquest cas. O `%x`, per imprimir-lo en hexadecimal.

2. `n`: Passa el valor directament.

- **Avantatge per a EC**: Aquesta estructura de `printf` es tradueix gairebé directament a l'**ABI** de RISC-V: el punter a la cadena de format anirà al registre `a0` i el valor de `n` anirà al registre `a1`. **No hi ha màgia, hi ha registres.**

⚠ Aprofundiment 1.2: Exemple de correspondències entre C i C++

Concepte	C++ (PRO1)	C (EC)	Per què en C?
Sortida de dades	`cout << x;`	`printf("%d", x);`	Cal explicitar el format de les dades (bits).
Entrada de dades	`cin >> x;`	`scanf("%d", &x);`	Obliga a usar l'adreça de memòria (`&`).
Reserva de memòria	`new int[10];`	`malloc(10 * sizeof(int));`	Cal gestionar la mida exacta en bytes.
Alliberar memòria	`delete[] v;`	`free(v);`	Gestió manual del *heap* (Secció 3.8).

1.2.8 Llenguatges interpretats

Mentre que C es **compila** (es tradueix tot el codi font abans d'executar-lo), altres llenguatges com Python s'**interpreten**: un programa anomenat **intèrpret** llegeix el codi font (o una forma intermèdia, com el **codi de bytes** (*bytecode*) de Python) i l'executa directament, sentència a sentència, sense generar cap binari per a l'ISA.

Els llenguatges interpretats no són apropiats per a EC perquè, en general, fan molt més opacs els detalls d'arquitectura, que és precisament l'àmbit d'interès de l'assignatura.

1.3 Arquitectura física: el model de von Neumann

La pràctica totalitat dels computadors actuals segueixen el model de programa emmagatzemat proposat per **John von Neumann** l'any 1945 (alguns microcontroladors separen la memòria d'instruccions de la de dades: arquitectura Harvard). L'aportació conceptual més important d'aquest model és el **programa emmagatzemat**.

1.3.1 El concepte de programa emmagatzemat: instruccions i dades a la mateixa memòria

Abans de von Neumann, per «reprogramar» una màquina calia canviar cables i interruptors físicament. En el model de von Neumann:

- Les **instruccions** (el codi binari, Secció 1.2.3.1) i les **dades** (els nombres amb què s'opera) resideixen al mateix lloc: la **memòria principal**.
- El processador llegeix la memòria, interpreta què és una instrucció i l'executa.

1.3.2 Els components del model

L'arquitectura es divideix en quatre blocs funcionals connectats entre si (Figura 1.3):

1. **Unitat central de processament** (*central processing unit*, CPU): El «cervell» que es divideix en:
   - **Unitat aritmeticològica** (*arithmetic logic unit*, ALU): On es fan les operacions matemàtiques i lògiques.
   - **Unitat de control** (*control unit*, CU): L'orquestra que llegeix la memòria i diu a la resta de components què han de fer.
   - **Registres**: L'emmagatzematge intern de la CPU, el més ràpid (entre els quals, el comptador de programa).
2. **Memòria principal** (MP): Un vector gegant de cel · les on cada posició té una **adreça** única.
3. **Sistema d'entrada/sortida** (E/S): Permet la comunicació amb el món exterior (teclat, pantalla, disc dur).
4. **Busos**: Les «autopistes» de dades que connecten tots els components.

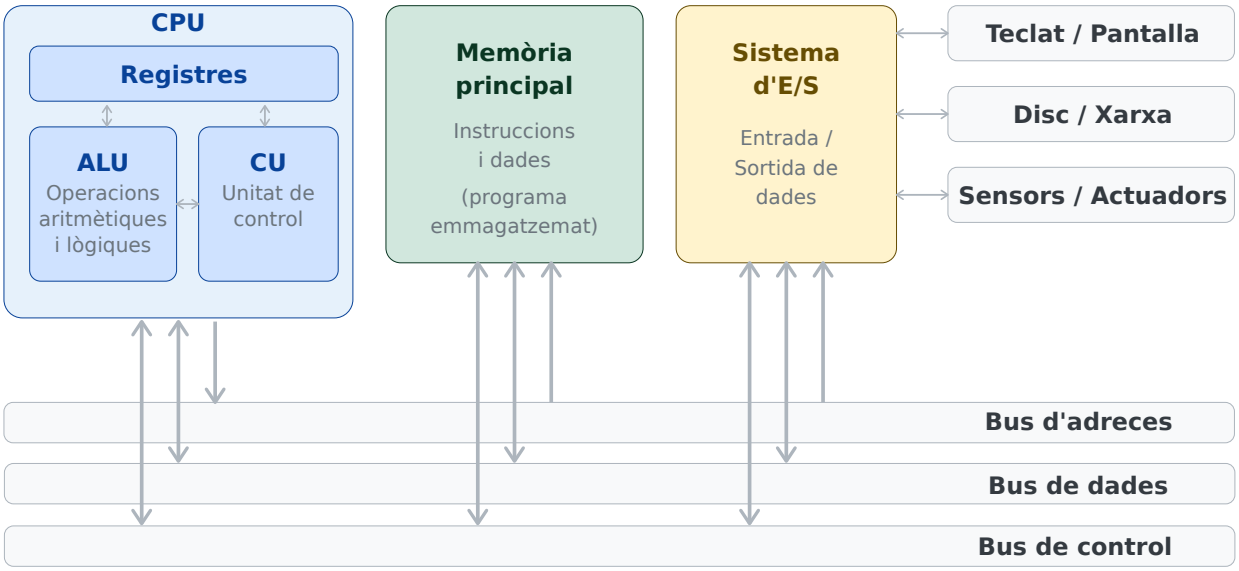


Figura 1.3: Arquitectura de von Neumann: CPU (ALU, CU i registres), memòria principal i sistema d'E/S interconnectats pels busos de dades, d'adreces i de control. Sovint se'n representa una versió simplificada amb un únic bus bidireccional que connecta CPU, Memòria i Sistema d'E/S.

1.3.3 El cicle d'instrucció (*fetch-decode-execute*)

La unitat de control repeteix indefinidament tres passos:

1. **Captura** (***fetch***): Es llegeix de la memòria la instrucció següent, la de l'adreça que guarda un registre especial, el comptador de programa (*program counter*, PC).
2. **Descodificació** (***decode***): La unitat de control analitza els bits de la instrucció per saber què cal fer (és una suma? És un salt? Cal llegir dades de la memòria?).
3. **Execució** (***execute***): S'activen els circuits de l'ALU o es mouen dades entre registres i memòria.

Aprofundiment 1.3: El «coll d'ampolla» del model de von Neumann

Com que les dades i les instruccions viatgen pel mateix bus cap a la memòria, el processador sovint ha d'esperar que la memòria li doni el que demana. Aquest límit de velocitat es coneix com el **coll d'ampolla de von Neumann**.

1.4 Codificació de nombres naturals en base 2

Un nombre natural x_u de n bits es representa com una cadena de bits $X = X_{n-1} \dots X_1 X_0$, on cada bit $X_i \in \{0, 1\}$.

1.4.1 Interpretació: bits → natural

Donat un vector de bits X, el natural representat és:

$$x_u = \sum_{i=0}^{n-1} X_i \cdot 2^i \tag{1.1}$$

Exemple 1.5: Interpretació de bits com a natural

Pregunta: S'ha d'interpretar $X = 0001\,1001_2$ (8 bits) com a natural.
Solució:

$$x_u = 1{\cdot}2^4 + 1{\cdot}2^3 + 1{\cdot}2^0 = 16 + 8 + 1 = 25$$

Resposta: $x_u = 25$

1.4.2 Representació: natural → bits

Donat un natural x_u, la seva representació en n bits s'obté dels residus de dividir successivament x_u entre 2, del bit de menys pes al de més pes.

Exemple 1.6: Representació d'un natural en binari

Pregunta: S'ha de representar $x_u = 25$ amb 8 bits.
Solució:

Divisió	Quocient	Residu (bit)
$25 \div 2$	12	$X_0 = 1$
$12 \div 2$	6	$X_1 = 0$
$6 \div 2$	3	$X_2 = 0$
$3 \div 2$	1	$X_3 = 1$
$1 \div 2$	0	$X_4 = 1$
$0 \div 2$	0	$X_5 = 0$
$0 \div 2$	0	$X_6 = 0$
$0 \div 2$	0	$X_7 = 0$

Resposta: $X = 0001\,1001_2$
Un cop el quocient és zero, les divisions restants només afegeixen zeros a l'esquerra (extensió de zeros, vegeu Secció 1.4.4).

1.4.3 Rang de representació

Amb n bits es poden representar 2^n naturals diferents:

$$x_u \in [0,\ 2^n - 1] \tag{1.2}$$

Per exemple, amb 8 bits: $x_u \in [0, 255]$.

1.4.4 Extensió de zeros

Per representar un natural de n bits amb $m > n$ bits, s'afegeixen zeros a l'esquerra. El valor representat no canvia.

Exemple 1.7: Extensió de zeros

El natural $x_u = 25$ es representa amb 8 bits com $0001\,1001_2$.
Amb 16 bits: $0000\,0000\,0001\,1001_2$. El valor és el mateix.

1.4.5 Notació hexadecimal

La representació binària és poc pràctica per a nombres de mida gran: és llarga i propensa a errors de transcripció. La **notació hexadecimal** (base 16) ofereix una manera compacta d'escriure patrons de bits, amb el prefix `0x` per identificar-la.

Cada dígit hexadecimal (0–9, A–F) representa exactament un **quartet** (*nibble*, 4 bits), ja que $2^4 = 16$. Aquesta correspondència directa és el que fa que la conversió binari↔hexadecimal sigui immediata, a diferència de la conversió a decimal: només cal agrupar els bits de 4 en 4 (des del bit menys significatiu) i traduir cada quartet pel seu dígit hexadecimal equivalent.

Exemple 1.8: Conversió entre binari i hexadecimal

Pregunta: S'ha d'expressar $X = 0001\,1001_2$ en hexadecimal.
Solució: S'agrupa en quartets: $0001_2 = 1$, $1001_2 = 9$.
Resposta: $X =$ `0x19`

Essencial 1.1

La notació hexadecimal és només una manera d'**escriure** un patró de bits de forma compacta; no en canvia la interpretació. Un mateix patró, com `0x19`, pot representar un natural, un enter en complement a 2 o qualsevol altra codificació segons el context (vegeu Secció 1.5).

1.5 Codificació de nombres enters en base 2

Un nombre enter x_s de n bits es representa com una cadena de bits $X = X_{n-1} \dots X_1 X_0$, on, en complement a 2 (Ca2), en complement a 1 (Ca1) i en signe i magnitud, el bit de més pes X_{n-1} indica el signe: 0 si no és negatiu i 1 si ho és.

En aquestes tres codificacions, els enters no negatius es codifiquen igual que els naturals equivalents. La codificació en excés (Secció 1.5.2) segueix una altra regla.

1.5.1 Enters en complement a 2 (Ca2)

És la codificació estàndard dels enters en els processadors moderns, inclòs **RISC-V**. El rang representable amb n bits és:

$$x_s \in [-2^{n-1},\ 2^{n-1} - 1] \tag{1.3}$$

Per exemple, amb 8 bits: $x_s \in [-128, +127]$.

1.5.1.1 Interpretació: bits → enter

$$x_s = -X_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} X_i \cdot 2^i \tag{1.4}$$

Exemple 1.9: Interpretació de bits com a enter en Ca2

Pregunta: S'ha d'interpretar $X = 1100\,1101_2$ (8 bits) com a enter en Ca2.
Solució: Aplicant Equació 1.4:

$$\begin{aligned} x_s &= -1{\cdot}2^7 + 1{\cdot}2^6 + 1{\cdot}2^3 + 1{\cdot}2^2 + 1{\cdot}2^0 \\ &= -128 + 64 + 8 + 4 + 1 = -51 \end{aligned}$$

Resposta: $x_s = -51$

1.5.1.2 Representació: enter → bits

Per a enters **positius**, el procediment és idèntic al dels naturals (Secció 1.4.2).

Per a enters **negatius**, s'aplica la **regla del canvi de signe**: es complementen tots els bits i es suma 1.

Exemple 1.10: Representació d'un enter negatiu en Ca2

Pregunta: S'ha de representar $x_s = -51$ amb 8 bits.
Solució: Representem l'oposat $+51$ per divisions successives:

$$51 \to 0011\,0011_2$$

Apliquem el canvi de signe (complement i suma 1):

$$\overline{0011\,0011_2} = 1100\,1100_2$$

$$1100\,1100_2 + 1 = 1100\,1101_2$$

Resposta: $X = 1100\,1101_2$

1.5.1.3 Extensió de signe

Per representar un enter en Ca2 de n bits amb $m > n$ bits, es replica el bit de signe X_{n-1} a l'esquerra. El valor representat no canvia.

Exemple 1.11: Extensió de signe

L'enter $x_s = -51$ es representa amb 8 bits com $1100\,1101_2$.
Amb 16 bits: $1111\,1111\,1100\,1101_2$. El valor és el mateix.

Essencial 1.2

Com a drecera pràctica per interpretar directament un patró de bits, sense passar per Equació 1.4: si $X_{n-1} = 1$ (bit de signe actiu), el valor en Ca2 s'obté restant 2^n al valor llegit com a natural: $x_s = x_u - 2^n$. Recíprocament, per representar un enter negatiu en Ca2: $\text{Ca2}(x_s) = x_s + 2^n$ (com a natural de n bits).

1.5.2 Enters en excés

La codificació en excés assigna la representació $000\ldots0_2$ al valor més negatiu i $111\ldots1_2$ al més positiu, de manera que l'ordre de les representacions coincideix amb l'ordre numèric dels enters. Això permet **comparar enters amb el mateix circuit que compara naturals**.

Donada una constant arbitrària K (l'**excés**):

$$x_s = x_u - K \tag{1.5}$$

$$x_u = x_s + K \tag{1.6}$$

L'elecció habitual és $K = 2^{n-1} - 1$, amb un rang $x_s \in [-(2^{n-1}-1),\ 2^{n-1}]$ (amb 8 bits, $[-127, +128]$):

Exemple 1.12: Interpretació i representació en excés

Amb $n = 8$ bits, $K = 2^7 - 1 = 127$.
Interpretació: $X = 1000\,0000_2 \Rightarrow x_u = 128 \Rightarrow x_s = 128 - 127 = +1$
Representació: $x_s = -127 \Rightarrow x_u = -127 + 127 = 0 \Rightarrow X = 0000\,0000_2$

EC 1.3

A **EC**, la codificació en excés s'utilitza per representar l'exponent dels nombres en **coma flotant** (*floating point*), que es presenten a Capítol 5.

1.5.3 Enters en signe i magnitud

El bit de més pes indica el signe (0 positiu, 1 negatiu) i els $n-1$ bits restants representen el valor absolut com un natural.

Rang: $x_s \in [-(2^{n-1}-1),\ +2^{n-1}-1]$

Inconvenients: doble representació del zero ($+0$ i -0) i circuit de suma diferent del dels naturals. No s'usa per als enters en els processadors moderns (sí, en canvi, a la coma flotant, Capítol 5).

1.5.4 Enters en complement a 1 (Ca1)

L'enter negatiu oposat de x_s s'obté complementant tots els bits de la seva representació.

Rang: $x_s \in [-(2^{n-1}-1),\ +2^{n-1}-1]$

Inconvenients: doble representació del zero i algorisme de suma diferent del dels naturals. No s'usa en processadors moderns.

1.6 Operacions aritmètiques en complement a 2

La propietat més important del **complement a 2** és que permet realitzar la suma i la resta d'enters amb el **mateix circuit** que opera amb naturals. Això simplifica notablement el disseny del maquinari i és el motiu principal pel qual Ca2 és la codificació estàndard dels enters als processadors moderns.

1.6.1 Suma en Ca2

Per sumar dos enters en Ca2, s'aplica directament la suma binària, ignorant el **ròssec de sortida** (*carry-out*) que depassi el bit de signe ($n-1$):

$$a+b \quad \Rightarrow \quad \text{Ca2}(a)+\text{Ca2}(b) \pmod{2^n} \tag{1.7}$$

El resultat és correcte si no hi ha **desbordament** (***overflow***). El desbordament es produeix quan el resultat matemàtic exacte cau fora del rang $[-2^{n-1},\ 2^{n-1}-1]$. La regla de detecció és: **hi ha desbordament si i només si els dos operands tenen el mateix signe i el resultat té signe contrari**.

> Exemple 1.13: Suma en Ca2
>
> Amb $n=8$ bits (rang $[-128,+127]$):
> **Sense desbordament**: $45+30=75$
>
> $$\underbrace{0010\,1101}_{+45}+\underbrace{0001\,1110}_{+30}=\underbrace{0100\,1011}_{+75}\ \checkmark$$
>
> **Amb desbordament**: $100+80=180$, però $180>127$:
>
> $$\underbrace{0110\,0100}_{+100}+\underbrace{0101\,0000}_{+80}=\underbrace{1011\,0100}_{?}$$
>
> El resultat $1011\,0100_2$ s'interpreta com a -76 en Ca2, no com a $+180$. Els dos operands eren positius i el resultat sembla negatiu: **desbordament**.

Exemple 1.14: Suma en Ca2 — visualització bit a bit

Els mateixos exemples amb la fila de **ròssec** (*carry*) explícita:
Sense desbordament: $45 + 30 = 75$

```
Ròssec:     0 1 1 1 1 0 0 0
            0 0 1 0 1 1 0 1   (+45)
          + 0 0 0 1 1 1 1 0   (+30)
          ---------------------
            0 1 0 0 1 0 1 1   (+75) ✓
```

Amb desbordament: $100 + 80$

```
Ròssec:     1 0 0 0 0 0 0 0
            0 1 1 0 0 1 0 0   (+100)
          + 0 1 0 1 0 0 0 0   (+80)
          ---------------------
            1 0 1 1 0 1 0 0   (-76 en Ca2, no +180)
```

El ròssec entra al bit de signe (posició 7), però **no en surt**: aquesta asimetria és exactament la condició de desbordament detectable per maquinari.

1.6.2 Inversió de signe

Per obtenir l'**oposat** d'un enter en Ca2 s'aplica la regla següent: complementar tots els bits i sumar 1.

$$-x \quad \Rightarrow \quad \overline{\text{Ca2}(x)} + 1 \pmod{2^n} \tag{1.8}$$

La regla és la seva pròpia inversa: aplicar-la dues vegades retorna el valor original.

Exemple 1.15: Inversió de signe

Primera inversió — de -45 a $+45$:

```
Original:   1101 0011   (-45)
Complement: 0010 1100
          +          1
          ------------
Resultat:   0010 1101   (+45) ✓
```

Segona inversió — de $+45$ de tornada a -45:

```
Original:   0010 1101   (+45)
Complement: 1101 0010
          +          1
          ------------
Resultat:   1101 0011   (-45) ✓
```

Aplicar la inversió dues vegades retorna el valor inicial.

Aprofundiment 1.4: El cas límit -2^{n-1}

El valor mínim de Ca2 (-2^{n-1}, representat com $1000\ldots0$) **no té oposat representable** en n bits. Aplicant la regla:

```
Original:   1000 0000   (-128)
Complement: 0111 1111
          +          1
          ------------
Resultat:   1000 0000   (-128)  ← desbordament: el resultat és el mateix!
```

El valor $+2^{n-1}$ ($+128$ amb 8 bits) requeriria $n+1$ bits per ser representat en Ca2. És l'única excepció del format.

1.6.3 Resta en Ca2

La resta s'implementa com una suma amb l'**oposat del subtrahend** (vegeu Secció 1.6.2 per a la regla d'inversió de signe).

$$a - b \equiv a + (-b) \quad \Rightarrow \quad \text{Ca2}(a) + \text{Ca2}(-b) \pmod{2^n} \tag{1.9}$$

Gràcies a aquesta equivalència, el maquinari necessita **un sol circuit** per a la suma i la resta.

Exemple 1.16: Resta en Ca2

Amb $n = 8$ bits:
Sense desbordament: $45 - 30 = 45 + (-30) = 15$

$$\text{Ca2}(-30) : \overline{0001\,1110} + 1 = 1110\,0010$$

$$\underbrace{0010\,1101}_{+45} + \underbrace{1110\,0010}_{-30} = \underbrace{0000\,1111}_{+15} \checkmark$$

Amb desbordament: $-100 - 45 = -100 + (-45) = -145$, però $-145 < -128$:

$$\text{Ca2}(-45) : \overline{0010\,1101} + 1 = 1101\,0011$$

$$\underbrace{1001\,1100}_{-100} + \underbrace{1101\,0011}_{-45} = \underbrace{0110\,1111}_{?}$$

El resultat $0110\,1111_2 = +111$ en Ca2, no -145: **desbordament** (dos operands negatius, resultat positiu).

Exemple 1.17: Resta en Ca2 — visualització bit a bit

Els mateixos exemples amb la fila de **préstec** (*borrow*) explícita:
Sense desbordament: $45 - 30 = 15$

```
Préstec:    0 0 1 1 1 1 0 0
            0 0 1 0 1 1 0 1   (+45)
          - 0 0 0 1 1 1 1 0   (+30)
            ------------------
            0 0 0 0 1 1 1 1   (+15) ✓
```

Amb desbordament: $-100 - 45 = -145$

```
Préstec:    1 1 0 1 1 1 1 0
            1 0 0 1 1 1 0 0   (-100)
          - 0 0 1 0 1 1 0 1   (+45)
          ---------------
            0 1 1 0 1 1 1 1   (+111 en Ca2, no -145)
```

El minuend és negatiu i el subtrahend positiu (signes contraris), però el resultat surt amb signe **contrari al del minuend** (positiu, quan el resultat exacte, -145, és negatiu): **desbordament**.

1.6.4 Multiplicació en Ca2

Per obtenir el producte complet de $2n$ bits en Ca2, **no** es pot aplicar directament l'algorisme dels naturals, perquè la codificació del signe interfereix amb els bits de més pes del producte (els n bits de menys pes, en canvi, coincideixen: Aprofundiment 4.7). El mètode pràctic és:

1. Determinar el **signe del resultat**: el producte de dos nombres del mateix signe és positiu; de signes contraris, negatiu.
2. Obtenir les **magnituds** (valor absolut de cada operand).
3. **Multiplicar les magnituds** com a naturals (sumes i desplaçaments).
4. Aplicar el **signe** al resultat i codificar-lo en Ca2.

El resultat és correcte si no hi ha desbordament; el rang representable en n bits és $[-2^{n-1},\ 2^{n-1}-1]$, que per al producte s'esgota ràpidament (calen $2n$ bits per garantir que no hi hagi desbordament).

Exemple 1.18: Multiplicació en Ca2

Amb $n = 8$ bits:
Sense desbordament: $(-6) \times 5 = -30$
Magnituds: $6 \times 5 = 30$. Signe: $(-) \times (+) \to (-)$.

$$\text{Ca2}(-30) : \overline{0001\,1110} + 1 = 1110\,0010$$

Resposta: $(-6) \times 5 = -30 \to 1110\,0010_2 =$ `0xE2`
Comprovació: $6 \times 5 = 30$; -30 en Ca2 (8 bits) $= 256 - 30 = 226 = 1110\,0010_2$ ✓

EC 1.4

A **EC**, les instruccions de multiplicació i divisió com a operacions de l'**ISA** (extensió **M**: `mul`, `mulh`, `div`, `divu`, etc.) s'estudien al Tema 4 (Secció 4.3.3).

1.6.5 Divisió en Ca2

La divisió en Ca2 segueix el mateix esquema de signe i magnituds que la multiplicació. El resultat consta de dues parts: el **quocient** i el **residu**. La convenció estàndard (seguida per C i per les instruccions `div`/`rem` de RISC-V) és la **truncació cap a zero**: el quocient s'arrodoneix cap a zero, i el residu té el mateix signe que el dividend.

El mètode pràctic és:

1. Determinar el **signe del quocient**: igual que en la multiplicació (positiu si els signes coincideixen, negatiu si difereixen).
2. Obtenir les **magnituds** (valor absolut de dividend i divisor).
3. **Dividir les magnituds** com a naturals (divisió entera), obtenint quocient i residu de les magnituds.
4. Aplicar el **signe del quocient** al quocient i codificar-lo en Ca2.
5. El **residu** pren el signe del **dividend** i es codifica en Ca2.

L'única divisió que produeix desbordament és $-2^{n-1} \div (-1)$, pel mateix motiu que Aprofundiment 1.4. Els casos especials de RISC-V, inclosa la divisió per zero, són a Secció 4.4.3.1.

Exemple 1.19: Divisió en Ca2

Amb $n = 8$ bits:
Sense residu: $(-30) \div 5$
Magnituds: $30 \div 5 = 6$, residu 0. Signe: $(-) \div (+) \to (-)$.

$$\text{Ca2}(-6) : \overline{0000\,0110} + 1 = 1111\,1010$$

Resposta: quocient $= -6 \to 1111\,1010_2$; residu $= 0 \to 0000\,0000_2$.
Amb residu: $(-7) \div 2$
Magnituds: $7 \div 2 = 3$, residu 1. Signe quocient: $(-) \div (+) \to (-)$. Signe residu: igual que el dividend $\to (-)$.

$$\text{Ca2}(-3) : \overline{0000\,0011} + 1 = 1111\,1101 \qquad \text{Ca2}(-1) : \overline{0000\,0001} + 1 = 1111\,1111$$

Resposta: quocient $= -3 \to 1111\,1101_2$; residu $= -1 \to 1111\,1111_2$.
Comprovació: $2 \times (-3) + (-1) = -6 - 1 = -7$ ✓

Tema 2: Instruccions i tipus de dades bàsics

2.1 Introducció a l'ISA de RISC-V

A l'assignatura IC, s'estudia en detall el disseny d'un computador simple, analitzant la relació entre el **llenguatge màquina** i el disseny dels **circuits lògics digitals**. A **EC**, s'aprofundeix en el **llenguatge d'assemblador** i s'estudia la relació entre aquest i un **llenguatge d'alt nivell**.

La taula d'EC 0.1 recull les especificacions més rellevants de les eines i llenguatges d'**EC**.

2.1.1 *Instruction Set Architecture*, ISA

L'especificació de la **interfície** entre el **maquinari** (*hardware*) i el **programari** (*software*) d'un processador s'anomena **conjunt** d'instruccions, **repertori** d'instruccions o **joc** d'instruccions, i també **arquitectura del repertori d'instruccions** (*instruction set architecture*, ISA). L'**ISA**, en tant que **interfície**, constitueix un **contracte** entre els fabricants del maquinari i els de programari. Al llarg dels anys, el fabricant d'un processador pot introduir en el seu disseny successives millores, a fi d'augmentar-ne el **rendiment**. Però s'ha demostrat que comercialment resulta crític mantenir la **compatibilitat** dels nous processadors amb els programes existents, i per a això cal que el nou disseny respecti punt per punt totes les **especificacions** de l'**ISA**.

L'**ISA** descriu tècnicament tots els aspectes del processador visibles al programador d'assemblador, incloent-hi:

- El format de les **instruccions** que el processador pot executar.
- La manera com l'execució de cada instrucció afecta l'**estat del processador**.
- Els **tipus de dades** suportats.
- Els **registres**.
- L'**arquitectura i gestió de la memòria**.
- El model d'**excepcions** i **interrupcions**.

Aquests aspectes constitueixen la matèria principal d'estudi d'**EC**.

> Essencial 2.1
>
> El terme *Instruction Set* pot induir a pensar que l'**ISA** es limita al conjunt d'instruccions. Però no és així: l'ISA descriu **tots** els aspectes del processador visibles al programador d'assemblador.

> Essencial 2.2
>
> Els termes **arquitectura de computadors** i **estructura de computadors** s'usen sovint com a sinònims, però tenen matisos diferenciats [11]:
>
> - **Arquitectura de computadors**: Els aspectes visibles al programador: el conjunt d'instruccions, els tipus de dades, els modes d'adreçament, els registres accessibles i el model de memòria. En definitiva, allò que defineix l'**ISA**.

- **Estructura de computadors**: La implementació interna que fa possible l'arquitectura: la unitat de control, l'ALU, la memòria cau, els busos i la microarquitectura en general. És el **com** s'implementa el **què** que defineix l'arquitectura.

A la pràctica, tots dos aspectes s'estudien de forma conjunta, ja que les decisions d'arquitectura condicionen l'estructura i viceversa. El nom de l'assignatura, **Estructura de Computadors**, reflecteix l'èmfasi en la implementació, però el seu contingut cobreix els dos nivells.

2.1.2 Arquitectura de tipus RISC (vs. CISC)

El terme *reduced instruction set computer* (RISC) apareix a principis dels anys 80 per designar l'arquitectura de certs processadors dissenyats amb criteris innovadors respecte a la tendència dels processadors existents fins llavors, els quals van passar a anomenar-se per contraposició *complex instruction set computers* (CISC).

Abans que els processadors **RISC** fossin populars, els ràpids avenços en la tecnologia d'integració a escala molt gran (*very large scale integration*, VLSI) havien permès als arquitectes integrar circuits cada cop més complexos, i adoptar llenguatges d'assemblador cada cop més potents, amb l'objectiu de reduir la **diferència semàntica** amb els llenguatges d'alt nivell. Així, per exemple, en alguns processadors es podien realitzar amb una sola instrucció de llenguatge màquina operacions tan complexes com avaluar un polinomi en coma flotant, o copiar una cadena de caràcters traduint-los prèviament per mitjà d'una taula de conversió. Per abastar tal varietat i sofisticació de tasques calia un repertori amplíssim d'instruccions, amb un nombre variable d'operands, cosa que obligava a codificar-les amb un **format de mida variable**. Per especificar la ubicació dels operands es necessitaven molts **modes d'adreçament** i **indireccions de memòria complexes**. Algunes instruccions involucraven tantes operacions que per interpretar-les es necessitava executar cada cop un petit programa escrit en **microcodi**, en una *read-only memory* (ROM) interna. Aquestes arquitectures són les que van ser més tard conegudes com **CISC**.

Però aquesta tendència va començar a canviar quan alguns estudis estadístics van mostrar que les **instruccions més complexes a penes es feien servir** en la pràctica, ja que responien a necessitats tan particulars que els compiladors trobaven moltes dificultats per aplicar-les. Aviat es va fer evident que tot i que les instruccions complexes podien resoldre problemes particulars de manera molt eficient, la seva implementació augmentava la complexitat del processador de tal manera que comportava sovint que l'execució de les instruccions més simples s'alentís. La filosofia **RISC** va tendir doncs a suprimir les instruccions complexes i lentes, seguint el principi que la mateixa tasca es pot realitzar amb una seqüència d'**instruccions simples**, i que suprimir-les del repertori permet dissenyar un **maquinari** més simple, capaç d'executar més ràpidament les altres instruccions (***make the common case fast***, una de les grans idees de [11]). La principal característica de les arquitectures **RISC** (**ARM**, **PowerPC**, **MIPS**, **RISC-V**, **Alpha**, **SPARC**, etc.) consistia en un joc d'instruccions dissenyat amb el criteri bàsic del **rendiment**. El més característic dels **RISC** era el fet de tenir **instruccions de mida fixa** (normalment 32 bits), pocs **modes d'adreçament** (les operacions aritmètiques i lògiques, només amb operands en registre o immediats), i que l'**accés a memòria** es feia només amb instruccions específiques de lectura i escriptura de memòria.

Originàriament, els nous microprocessadors de tipus **RISC** eren estructuralment més simples que els de tipus **CISC** i estaven formats per un nombre molt més reduït de transistors, característica que els feia **més barats, ràpids i energèticament eficients**. Moltes de les arquitectures **CISC** van desaparèixer, però algunes que estaven fermament instal · lades en el mercat (com la **x86** d'Intel i **AMD**) simplement van evolucionar, i amb els anys, les diferències entre **CISC** i **RISC** s'han anat difuminant. Per una banda, els processadors **x86** actuals converteixen internament cada una de les instruccions en una o vàries **microoperacions** del tipus **RISC**. Per l'altra, a mesura que la dissipació

de calor ha frenat l'augment de les freqüències de rellotge, aquestes arquitectures han anat incorporant als seus repertoris noves instruccions per augmentar-ne la funcionalitat. Amb aquestes ampliacions, el repertori ha deixat de ser «reduït», però no per això s'ha desvirtuat el criteri **RISC**, que no és pas la llargada del repertori sinó la **millora de rendiment global**.

2.1.3 L'arquitectura RISC-V

A **EC**, s'estudien els aspectes bàsics de l'arquitectura **RISC-V**. Aquesta arquitectura va ser proposada l'any 2010 per l'equip de la Universitat de Califòrnia a Berkeley liderat per Krste Asanović i David Patterson. **RISC-V** destaca per ser un **estàndard obert** (***open standard***), gestionat per l'associació **RISC-V International**, que qualsevol fabricant pot implementar sense pagar llicències. RISC-V està inspirada en **MIPS**[1], arquitectura desenvolupada a la Universitat de Stanford per John L. Hennessy.

RISC-V s'ha dissenyat de forma **modular** mitjançant **extensions** d'una **base** (el mínim per ser considerat RISC-V) com **RV32I** o **RV64I**, on els registres són de 32 bits o de 64 bits, respectivament (a RISC-V, però, el terme *word* designa sempre 32 bits, també a RV64, on una dada de 64 bits és una *doubleword*: Aprofundiment 2.2). Les extensions s'identifiquen amb lletres afegides a la nomenclatura de la base, per exemple **RV32IMF** indica que el processador implementa la base RV32I i les extensions M i F. L'ordre de les lletres és fix: RV32IMAC és vàlid; RV32ICAM, no.

Essencial 2.3

La **paraula nativa** (***word***) és la mida en bits de dada que el processador maneja de manera més natural i eficient:

- Mida dels registres.
- Operand directe de les instruccions aritmètiques.
- Mida per defecte en operacions de lectura i escriptura de memòria.

És una de les característiques determinants de qualsevol arquitectura.

La modularitat de RISC-V s'expressa mitjançant sufixos.

RISC-V 2.1: RV32I ISA — Extensions més comunes (G)

La lletra **G** (**RV32G**) és l'abreviatura de **propòsit general** (***General-purpose***). Indica que el processador inclou el conjunt estàndard d'extensions **IMAFD + Zicsr + Zifencei**, que són les més habituals i suficients per executar sistemes operatius i aplicacions modernes.

- **M** (**Multiply**): Multiplicació i divisió per maquinari.
- **A (Atomic)**: Instruccions de sincronització i memòria atòmica.
- **F (Single-precision Floating-point)**: Coma flotant de precisió simple.
- **D (Double-precision Floating-point)**: Coma flotant de precisió doble.
- **Zicsr**: Accés als registres de control i estat (*control and status registers*, CSR).
- **Zifencei**: Instruccions de sincronització d'instruccions.

A més d'aquestes extensions, és molt habitual afegir-hi la identificada amb la lletra **C** (***Compressed***), que redueix la mida del codi per millorar la **densitat de memòria**, de manera que la nomenclatura completa més habitual és **RV32GC/RV64GC**. En el compilador **GNU/GCC**, el paràmetre `-march` permet especificar la combinació d'extensions.

[1] MIPS va ser l'arquitectura de referència d'EC fins al curs 2025/26.

Aquesta flexibilitat és molt potent, però genera un problema pràctic: dos processadors que compleixen l'ISA RISC-V poden tenir conjunts d'extensions tan diferents que el programari compilat per a un d'ells no funcioni en l'altre. Per resoldre aquest problema de portabilitat, RISC-V International defineix els **perfils** (***profiles***): conjunts estandarditzats d'extensions que un processador ha de suportar obligatòriament per poder-se anunciar sota un determinat perfil.

Essencial 2.4

No s'ha de confondre la lletra G amb els perfils oficials de RISC-V:

- G (propòsit general): Històricament, la lletra G es va crear com una abreviatura primerenca i pràctica per a I + M + A + F + D, juntament amb les extensions Zicsr i Zifencei. Simplement indica a un compilador (com GCC o Clang) que activi un paquet específic i predefinit d'extensions d'una sola lletra.
- Perfils oficials de RISC-V: Els perfils són els fulls de ruta estandarditzats i versionats que gestiona RISC-V International (per exemple, RVA20, RVA22, RVA23). Defineixen requisits molt més estrictes per a la portabilitat del programari d'aplicació, agrupant no només extensions estàndard d'una sola lletra, sinó també definicions de models de memòria, nivells de privilegi, gestió de memòria cau i subvariants específiques d'extensions Z.

Aprofundiment 2.1: RISC-V — *Profiles*

Els perfils ratificats més rellevants són:

- **RVI20U32 / RVI20U64**: Perfil mínim genèric, equivalent a RV32I i RV64I respectivament. És el punt de partida per a eines de programari.
- **RVA20U64, RVA22U64**: Perfils per a processadors d'aplicació de 64 bits. Defineixen un conjunt d'extensions suficient per executar sistemes operatius estàndard.
- **RVA23U64**: Ratificat l'octubre de 2024, fa obligatòria l'extensió vectorial (**V**), entre d'altres. És el perfil de referència per a distribucions Linux modernes.

Els perfils es diferencien de la nomenclatura d'extensions (`rv32imf`, etc.) perquè estan pensats en primer lloc per a acords entre fabricants de maquinari i de programari (tot i que els compiladors recents també els accepten a `-march`).

EC 2.1: Base i extensions de l'assignatura

A **EC**, s'estudia la base i les extensions següents:

- La base **RV32I**, Tema 2: Instruccions i tipus de dades bàsics i Tema 3: Traducció de programes.
- L'extensió **M**, Tema 4: Aritmètica entera i matrius.
- L'extensió **F**, Tema 5: Coma flotant.
- L'extensió **Zicsr** Tema 9: Excepcions i interrupcions.

Aprofundiment 2.2: Per què RV32 i no RV64?

Pràcticament tots els processadors RISC-V d'aplicació disponibles actualment (SiFive, StarFive, etc.) i les distribucions Linux sobre RISC-V utilitzen la variant de 64 bits (**RV64**). Per tant, és legítim preguntar-se per què a **EC** s'estudia RV32I.
La raó és pedagògica. En RV32I, registres, adreces de memòria i dades escalars bàsiques (`int`)

tenen tots la mateixa amplada de 32 bits. Aquesta coherència elimina una font important de confusió quan s'està aprenent la relació entre bits, registres i memòria per primera vegada.
En canvi, RV64 introdueix asimetries que compliquen l'aprenentatge inicial:

- Els registres i les adreces són de 64 bits, però el tipus `int` de C continua sent de 32 bits. Cal distingir constantment entre operacions de 32 i de 64 bits amb instruccions diferents (`add` vs. `addw`, `lw` vs. `ld`, etc.).
- Els mapes de memòria, les representacions hexadecimals i els exemples numèrics (Ca2, coma flotant, adreces) són més grans i menys manejables manualment.

Un cop assentats els conceptes fonamentals amb RV32I, la transició a RV64 és relativament senzilla: essencialment consisteix a ampliar l'amplada dels registres i ajustar alguns detalls de l'ABI. La dificultat real de l'assignatura no és la distinció 32 vs. 64 bits, sinó entendre la relació entre C, assemblador, ISA i maquinari.

2.1.4 La base RV32I

RISC-V 2.2 recull les característiques principals de la base **RV32I**. En acabar l'assignatura s'hauran tractat tots els conceptes que hi ha involucrats.

RISC-V 2.2: RV32I ISA — Característiques bàsiques i instruccions

Característiques bàsiques:

- Tot processador RISC-V implementa una base entera: RV32I, RV64I o les variants reduïdes RV32E i RV64E.
- **Paraula nativa** (***word***) de **32 bits**.
- **32 registres** de **32 bits** (**x0 fins a x31**, segons l'ISA).
- El registre **x0** (també anomenat **zero**) està fixat per **maquinari** al valor 0 i, per tant, no es pot modificar.
- **Espai d'adreces** de **32 bits**: 2^{32} adreces (`0x00000000`–`0xFFFFFFFF`); una adreça per **byte**; 4 GiB (2^{32} bytes).
- L'espai de memòria és **unificat** (instruccions i dades comparteixen el mateix espai d'adreces).
- **Model lectura/escriptura** (*load/store*): accés a memòria exclusivament mitjançant instruccions de lectura i escriptura (`lw`/`sw` i variants).
- Nombre d'instruccions: **40**.

Totes les instruccions:

- **Aritmètiques i lògiques**:
  - **Mode registre**: `add`, `sub`, `and`, `or`, `xor`, `sll`, `srl`, `sra`, `slt`, `sltu`.
  - **Mode immediat**: `addi`, `andi`, `ori`, `xori`, `slli`, `srli`, `srai`, `slti`, `sltiu`.
- **Càrrega d'immediats**: `lui`, `auipc`.
- **Lectura/escriptura**: `lw`, `lh`, `lb`, `lhu`, `lbu`, `sw`, `sh`, `sb`.
- **Salts condicionals**: `beq`, `bne`, `blt`, `bge`, `bltu`, `bgeu`.
- **Salts incondicionals**: `jal`, `jalr`.
- **Sistema**: `ecall`, `ebreak`, `fence` (`fence.tso` és una variant codificada dins `fence`).

A **EC**, s'estudien totes excepte les de sistema, de les quals només es tracta `ecall` (Secció 9.5).

Essencial 2.5

Taula 2.1: Prefixos de mida: amb «i» (binaris, potències de 2) vs. sense «i» (decimals, potències de 10)

Prefix	Símbol	Valor	Origen
Kilo / Kibi	`kB` (sovint escrit `KB`) / `KiB`	$10^3 = 1000$ / $2^{10} = 1024$ bytes	SI (decimal) / IEC (binari)
Mega / Mebi	`MB` / `MiB`	10^6 / $2^{20} = 1\,048\,576$ bytes	SI (decimal) / IEC (binari)
Giga / Gibi	`GB` / `GiB`	10^9 / 2^{30} bytes	SI (decimal) / IEC (binari)
Tera / Tebi	`TB` / `TiB`	10^{12} / 2^{40} bytes	SI (decimal) / IEC (binari)

El prefix **amb `i`** (`KiB`, `MiB`, `GiB`, `TiB`), definit per la *International Electrotechnical Commission* (IEC), denota sempre una potència de **2**: 1 KiB = 1024 bytes. El prefix **sense `i`** (`KB`, `MB`, `GB`, `TB`), el del Sistema Internacional d'Unitats (SI), denota, estrictament, una potència de **10**: 1 KB = 1000 bytes. A **EC**, gairebé totes les mides que apareixen són potències de 2 exactes (espais d'adreçament, pàgines, memòries cau), també les del *translation lookaside buffer* (TLB) de T8, de manera que s'utilitzen sistemàticament els prefixos **amb `i`**. Els prefixos **sense `i`** només s'usen quan la mida és genuïnament decimal, com en capacitats comercials de disc anunciades pels fabricants.

Aprofundiment 2.3: RISC-V GNU Compiler Toolchain — Paràmetres rellevants i traducció de C a RV32I comprensible

A **EC**, els paràmetres més rellevants del compilador creuat `riscv32-unknown-elf-gcc` (RISC-V GNU Compiler Toolchain) són:

Taula 2.2: GCC, paràmetres per a la generació de codi assemblador sense optimitzacions

Paràmetre	Valor	Significat
`-march`	`rv32i`	ISA de destí: només la base entera (**I**)
`-march`	`rv32imf`	ISA de destí: base entera (**I**) + multiplicació (**M**) + coma flotant simple (**F**)
`-mabi`	`ilp32`	ABI de destí: enters i punters de 32 bits; arguments i resultats de coma flotant pels registres enters
`-mabi`	`ilp32f`	ABI de destí: enters i punters de 32 bits; arguments en coma flotant passats per registres de coma flotant
`-S`		Atura la compilació després de generar el fitxer assemblador (`.s`); no assembla ni enllaça
`-O0`		Desactiva totes les optimitzacions del compilador
`-fno-omit-frame-pointer`		Preserva el punter de marc (*frame pointer*) per facilitar la lectura del codi assemblador generat
`-fsigned-char`		`char` amb signe, la convenció d'EC (EC 2.10)

Per obtenir el fitxer d'assemblador el més comprensible possible, cal deshabilitar qualsevol optimització:

`Terminal`

```
# Fins a T3 (només RV32I):
riscv32-unknown-elf-gcc -march=rv32i -mabi=ilp32 -S -O0 -fno-omit-frame-pointer -fsigned-char
↪  fitxer.c

# T4 (extensió M):
riscv32-unknown-elf-gcc -march=rv32im -mabi=ilp32 -S -O0 -fno-omit-frame-pointer -fsigned-char
↪  fitxer.c

# A partir de T5 (extensions M i F):
riscv32-unknown-elf-gcc -march=rv32imf -mabi=ilp32f -S -O0 -fno-omit-frame-pointer -fsigned-char
↪  fitxer.c
```

El resultat és un fitxer `fitxer.s` amb el codi assemblador RV32I generat pel compilador, sense optimitzacions i amb el marc de pila preservat, proper a la traducció manual que es practica a EC.

2.1.5 L'extensió M

RISC-V 2.3: RV32IM ISA — Característiques bàsiques

- Afegeix **multiplicació i divisió entera** per **maquinari**.
- Sense l'extensió M, aquestes operacions s'han d'implementar per **programari** (seqüències d'instruccions RV32I), amb un cost en temps d'execució molt superior.
- Els operands i el resultat són de **32 bits** (registres de propòsit general).
- La multiplicació de 32 bits pot produir un resultat de **64 bits**; instruccions específiques (`mulh`, `mulhsu`, `mulhu`) permeten obtenir els **32 bits superiors** del resultat.
- Nombre d'instruccions: **8**.

RISC-V 2.4: RV32IM ISA — Totes les instruccions

- **Multiplicació**: `mul`, `mulh`, `mulhsu`, `mulhu`.
- **Divisió**: `div`, `divu`, `rem`, `remu`.

2.1.6 L'extensió F

RISC-V 2.5: RV32F ISA — Característiques bàsiques

- Afegeix aritmètica en **coma flotant** de **precisió simple** (*single precision*, 32 bits), seguint l'estàndard **IEEE 754**, de l'*Institute of Electrical and Electronics Engineers* (IEEE).
- Incorpora **32 registres de coma flotant** de 32 bits addicionals: `f0`–`f31`, independents dels registres enters d'RV32I.
- Inclou instruccions de **lectura i escriptura** (`flw`/`fsw`), **aritmètica** (`fadd.s`, `fsub.s`, `fmul.s`, `fdiv.s`, `fsqrt.s`), **comparació**, **conversió** entre enters i flotants, i operacions de **multiplicació i suma fusionades** (*fused multiply-add*: `fmadd.s`, `fmsub.s`, etc.).
- Nombre d'instruccions: **26**.

> **RISC-V 2.6: RV32F ISA — Totes les instruccions**
>
> L'extensió **F** afegeix **26 instruccions** de coma flotant de precisió simple, distribuïdes en aquests grups:
>
> - **Lectura/escriptura**: `flw`, `fsw`.
> - **Aritmètica bàsica**: `fadd.s`, `fsub.s`, `fmul.s`, `fdiv.s`, `fsqrt.s`.
> - **Fusionades (*fused*)**: `fmadd.s`, `fmsub.s`, `fnmadd.s`, `fnmsub.s`.
> - **Signe**: `fsgnj.s`, `fsgnjn.s`, `fsgnjx.s`.
> - **Comparació i mínim/màxim**: `feq.s`, `flt.s`, `fle.s`, `fmin.s`, `fmax.s`.
> - **Conversió**: `fcvt.w.s`, `fcvt.wu.s`, `fcvt.s.w`, `fcvt.s.wu`.
> - **Moviment i classificació**: `fmv.x.w`, `fmv.w.x`, `fclass.s`.

2.2 Format de les instruccions RV32I

En qualsevol arquitectura, les **instruccions** es representen amb **cadenes de bits**, s'emmagatzemen a la **memòria**, i són llegides (i escrites si es tenen els privilegis), igual que si fossin dades (model de memòria unificat). Es codifiquen seguint uns determinats patrons, que s'anomenen **formats**.

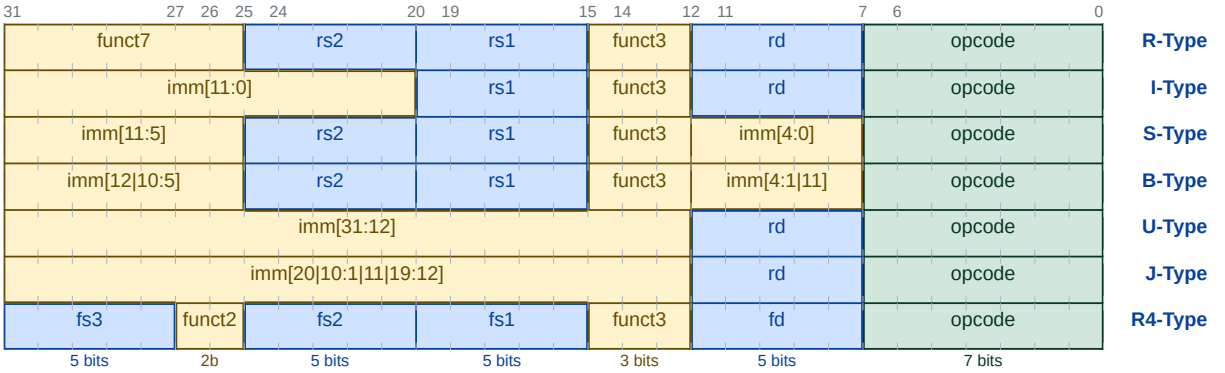
Un **format d'instrucció** especifica la subdivisió de la instrucció (en aquest cas de **32 bits**) en una sèrie de **camps** (***fields***) de bits corresponents al **codi d'operació** (`opcode`) i als **operands**.

Idealment, per satisfer els **principis de disseny** de **regularitat** i **simplicitat**, hi hauria un únic format d'instrucció, però hi ha casos que no ho permeten. D'altra banda, hi ha un conflicte entre el desig de mantenir totes les instruccions de la mateixa mida i el desig de tenir un únic format d'instrucció. El **Principi de Disseny 3** estableix que «un bon disseny requereix compromisos» (*good design demands good compromises*). El compromís escollit a **RISC-V** és mantenir totes les instruccions de la mateixa mida, un **word**[2] (a **RV32I**, 32 bits, 4 bytes), i definir **quatre formats nuclears d'instrucció** [3].

L'arquitectura **RV32I** defineix quatre formats nuclears d'instrucció [3]: **R** (*Register*), **I** (*Immediate*), **S** (*Store*) i **U** (*Upper*). Dos formats addicionals, **B** (*Branch*) i **J** (*Jump*), són variants de S i U, respectivament: reorganitzen els mateixos bits d'immediat per codificar desplaçaments sempre parells (el bit 0 és implícit), i s'introdueixen a Tema 3 amb les instruccions de salt (RISC-V 3.6, RISC-V 3.10). RISC-V 2.7 mostra els **camps de bits** de tots els formats, com a visió de conjunt: en aquest tema es fan servir R, I i S, i el format U s'introdueix més endavant, amb `lui` (RISC-V 2.27); B i J, a T3; i R4, el de les instruccions de coma flotant amb tres registres font, a T5 (Figura 5.7). La mateixa figura és al compendi de referència tècnica (RISC-V 6.3).

[2] Amb l'extensió C (*Compressed*), les instruccions poden ser de 16 bits. L'extensió C defineix versions comprimides de les instruccions més comunes, codificades en 16 bits en lloc de 32, per reduir la mida del codi.

RISC-V 2.7: RV32I ISA — Tipus de format de les instruccions



Els camps `funct3` i `funct7` s'utilitzen per estendre l'`opcode`, especificant l'operació concreta dins d'una mateixa categoria. Els camps `rs1`, `rs2` (registres font, *source registers*) i `rd` (registre destinació, *destination register*) codifiquen operands en **mode registre**. El camp `imm` codifica operands en **mode immediat**; en **RV32I** els bits dels immediats estan distribuïts de manera que els registres font i destinació sempre ocupen posicions fixes, la qual cosa simplifica la descodificació en el **maquinari**.

Exemple 2.1: Codificació amb mnemònics vs. codificació en codi màquina

L'objectiu d'aquest exemple és il · lustrar la correspondència entre la sintaxi d'algunes instruccions i els camps de bits de les seves codificacions. La sintaxi i semàntica d'aquestes i moltes altres instruccions s'aniran estudiant progressivament durant el curs.
Codi font (a títol purament il · lustratiu):

`RV32I`

```
add     t0, s2, zero
sub     t1, s3, s4
addi    t2, s5, 10
lw      t3, 4(s6)
sw      s7, 8(s8)
jal     ra, label
```

Associació als camps de bits de la codificació de les instruccions:

Instrucció	Operands	Tipus	`opcode`	`rd`	`funct3`	`rs1`	`rs2`	`funct7` / `imm`
`add`	`rd, rs1, rs2`	R	`0x33`	`rd`	`0x0`	`rs1`	`rs2`	`0x00`
`sub`	`rd, rs1, rs2`	R	`0x33`	`rd`	`0x0`	`rs1`	`rs2`	`0x20`
`addi`	`rd, rs1, imm12`	I	`0x13`	`rd`	`0x0`	`rs1`	`-`	`imm[11:0]`
`lw`	`rd, offset(rs1)`	I	`0x03`	`rd`	`0x2`	`rs1`	`-`	`offset[11:0]`
`sw`	`rs2, offset(rs1)`	S	`0x23`	`-`	`0x2`	`rs1`	`rs2`	`offset[11:0]`
`jal`	`rd, label`	J	`0x6F`	`rd`	`-`	`-`	`-`	`offset[20:1]`

La darrera columna recull l'immediat sencer, encara que ocupi altres posicions: a les instruccions de format I, els bits 31–20 (les posicions de `rs2` i `funct7`); a `sw` (format S), els bits 31–25 (`offset[11:5]`) i els bits 11–7, la posició de `rd` (`offset[4:0]`); i a `jal` (format J, RISC-V 3.10), els bits 31–12.
Abocament de memòria de les instruccions a RARS, en el format *Text/Data Segment Window* (RISC-V 2.8):

```
Abocament de memòria (Text/Data Segment Window)

Address     Code        Basic                    Line Source

0x00400000  0x000902b3  add x5,x18,x0            2           add     t0, s2, zero
0x00400004  0x41498333  sub x6,x19,x20           3           sub     t1, s3, s4
0x00400008  0x00aa8393  addi x7,x21,10           4           addi    t2, s5, 10
0x0040000c  0x004b2e03  lw x28,4(x22)            5           lw      t3, 4(s6)
0x00400010  0x017c2423  sw x23,8(x24)            6           sw      s7, 8(s8)
0x00400014  0x004000ef  jal x1,0x00000004        7           jal     ra, label
```

RISC-V 2.8: RARS — Abocament de memòria

A **RARS**, per obtenir un **abocament de memòria** (*memory dump*) de `.text` o `.data` (RISC-V 2.9): *File* → *Dump Memory* (*Text/Data Segment Window*, per exemple).

EC 2.2

A **EC**, es codifica sempre en mnemònics, si no s'especifica el contrari.
Recomanació: Convé aprendre's de memòria el significat dels mnemònics i fer-los servir quan llegiu codi d'assemblador.

Exemple 2.2: Codificació d'instruccions en binari i hexadecimal

```
RV32I

        add     t0, s2, zero
```

A partir de RISC-V 2.24:

- **t0**: `x5` = 00101_2.
- **s2**: `x18` = 10010_2.
- **zero**: `x0` = 00000_2.

Camps de la instrucció `add` (format **Tipus-R**):

- **opcode**: `0x33` = 0110011_2.
- **rd** (`t0`): 00101_2.
- **funct3** (`0x0`): 000_2.
- **rs1** (`s2`): 10010_2.
- **rs2** (`zero`): 00000_2.
- **funct7** (`0x00`): 0000000_2.

Resposta:

- **Binari**: `0000000 00000 10010 000 00101 0110011`.
- **Hexadecimal**: `0x000902B3`.

Comprovació: A RARS, a *Execute* (F3) o fent un abocament de memòria de la secció `.text` (RISC-V 2.8).

⚠ Aprofundiment 2.4: Nomenclatura alternativa a Patterson & Hennessy

Algunes edicions de [11] anomenen el format B **SB** i el format J **UJ**, reflectint que B és una variant de S (*store*) i J és una variant de U (*upper immediate*). Harris & Harris [12] fa servir la nomenclatura oficial de l'especificació RISC-V, **B** i **J**, que és la que s'usa a **EC**.

2.3 Estructura d'un programa en assemblador

Un programa en assemblador es compon de:

- **Segments**: Regions del programa que agrupen contingut del mateix tipus.
- **Etiquetes**: Noms simbòlics que representen adreces de memòria.
- **Directives**: Instruccions per a l'assemblador que no generen codi màquina.
- **Constants simbòliques**: Noms significatius associats a valors literals constants, que l'assemblador substitueix abans de generar el codi.
- **Instruccions, pseudoinstruccions i macros**: Les operacions que el processador executa o que l'assemblador expandeix en operacions reals.
- **Comentaris**: Anotacions per al programador que l'assemblador ignora completament.

2.3.1 Segments

Els principals segments d'un programa són: `.text` (instruccions) i `.data` (variables globals). Es declaren amb una **directiva de segment** i tot el contingut que els segueix fins al segment següent hi pertany.

ℹ RISC-V 2.9: Directives — Segments de memòria: `.text` i `.data`

Les directives `.text` i `.data` són **estàndard** de l'assemblador RISC-V de GNU (GNU Assembler) i del format de fitxer *Executable and Linkable Format* (ELF); no són específiques de RARS. El que sí que depèn de la configuració de **RARS** són les **adreces inicials** de cada segment:

Segment	Directiva	Adreça inicial	Contingut
Codi	`.text`	`0x00400000`	Instruccions executables
Dades	`.data`	`0x10010000`	Variables globals

L'execució comença a la primera instrucció del segment `.text`, és a dir, a l'adreça `0x00400000` de la taula: RARS hi posa el comptador de programa (*program counter*, PC) en començar i, amb la configuració per defecte, no cerca cap etiqueta d'entrada. Per això, als programes d'aquests apunts, el punt d'entrada **no porta etiqueta** i el `.text` del programa principal va **abans** de qualsevol subrutina. El motiu de fons —que a EC es programa sense sistema operatiu— és a RISC-V 3.23.
Forma llarga de la directiva. L'assemblador de GNU admet també `.section .data` i `.section .text`, que és la forma que genera `gcc` i la que trobareu en codi real. **A EC s'usa sempre la forma curta** (`.data`, `.text`): RARS no accepta la forma llarga per a aquests dos segments —dona l'error `.section must be followed by a section name`— i, en les variants que sí que assembla, la directiva **no commuta de segment**, de manera que les dades o el codi que la segueixen no van on sembla. És una limitació del simulador, no una diferència de l'arquitectura.

⚠ Aprofundiment 2.5: Seccions addicionals a l'ABI ELF

En sistemes Linux reals, l'ABI ELF defineix seccions addicionals que RARS no suporta explícitament:

Secció	Directiva	Contingut
Dades de només lectura	`.rodata`	Constants globals; protegides d'escriptura en temps d'execució
Dades no inicialitzades	`.bss`	Variables globals sense valor inicial; inicialitzades a zero per l'SO. No ocupa espai al fitxer executable

A RARS, tant `.rodata` com `.bss` es substitueixen per `.data`: les constants no estan protegides d'escriptura i les variables no inicialitzades es declaren amb `.space` (RISC-V 2.31).

2.3.2 Etiquetes

Una **etiqueta** (***label***) és un nom simbòlic que representa una adreça de memòria. Es declara escrivint el nom seguit de dos punts (`:`), sense sagnat. L'assemblador la substitueix per l'adreça definitiva durant l'assemblatge o, si és d'un altre mòdul, durant l'enllaçament (Secció 3.10).

ℹ RISC-V 2.10: RARS — Noms reservats

A **RARS**, una constant simbòlica (`.eqv`) **no pot tenir el mateix nom que cap de les paraules que l'assemblador ja coneix**: cap mnemònic (`add`, `b`, `j`, `li`, `mv`, `ret`, `nop`…) ni cap nom de registre (`t0`, `a0`, `zero`…); i una etiqueta no pot tenir el nom d'un registre. **El missatge d'error no us ho diu**, i és per això que costa de trobar.
Els dos missatges amb què hi arribareu:

On	Nom que hi poseu	Missatge
`.eqv`	mnemònic o registre	`Malformed .eqv directive`
Etiqueta a `.text`	registre en minúscules	`"t0" is not a recognized operator`

Exemple concret: si voleu una constant per a la mida d'un bloc i escriviu

⚠`codi_erroni__nom_reservat.s`⚠

```
.eqv    B, 16           # ✗ Malformed .eqv directive
```

falla perquè `B` és el salt incondicional `b`, i **els mnemònics no distingeixen majúscules de minúscules**. La solució és un nom més descriptiu, que a més es llegeix millor:

`RV32I`

```
.eqv    BLK, 16         # ✓
```

Dos detalls que fan el diagnòstic més difícil:

- **Les dues cares no es comporten igual.** Una *etiqueta* sí que pot tenir el nom `add` o `b` (els mnemònics només estan reservats per a `.eqv`); en canvi, un nom de *registre* en minúscules com a etiqueta només falla al segment `.text`.
- **Al segment `.data`, una etiqueta anomenada `t0` assembla**, però és inservible: l'error apareix més tard, on la useu (`la t1, t0` → `"t0": operand is of incorrect type`).

En majúscules (`T0`, `ZERO`) no hi ha xoc, perquè els noms de registre sí que distingeixen majúscules; però **no és una solució recomanable**: un nom com `T0` confon el lector. Trieu noms descriptius i el problema no apareix.

2.3.3 Directives

Essencial 2.6

Les **directives** són instruccions per a l'**assemblador** (o el simulador); no generen codi màquina ni afegeixen codi d'execució.

Les directives cobreixen funcions molt diverses: reservar espai en memòria, inicialitzar dades, declarar constants simbòliques, especificar visibilitat de símbols, etc. Es detallen a les seccions següents i a mesura que apareixen al llarg del curs.

2.3.4 Constants simbòliques

Les **constants simbòliques** faciliten l'escriptura i la lectura dels programes: permeten donar un nom significatiu a un valor que pot aparèixer en múltiples llocs, de manera que una sola modificació actualitza totes les ocurrències.

A **RARS**, les constants simbòliques es declaren mitjançant la directiva `.eqv` (GNU `as` hi afegeix `.equ` i `.set`). El nom que se'ls dona no pot coincidir amb cap mnemònic ni cap nom de registre (RISC-V 2.10).

Exemple 2.3: C i RARS — Constants simbòliques

`C`

```
#define N 10
```

RV32I

```
        .eqv    N, 10
```

Un cop definida, es pot usar en qualsevol context on s'usaria el literal equivalent:

RV32I

```
.text
        li      t0, N    # equival a li t0, 10
```

RISC-V 2.11: RARS — Operands sense expressions aritmètiques

A **RARS**, una constant simbòlica (`.eqv`) només és vàlida com a substitució d'un **literal únic**; l'assemblador **no avalua expressions aritmètiques** en cap operand, ni de directives ni d'instruccions. Així, `.space N*4`, `li t0, N*4` o `la t0, vec + 4` produeixen un error d'assemblatge, encara que `N` estigui definida amb `.eqv`.
Per tant, quan una mida o un valor és el resultat d'un càlcul (p. ex. el nombre de bytes d'un vector o d'una matriu), cal escriure el **literal ja calculat** a l'operand i deixar l'**expressió aritmètica al comentari**, com a documentació:

RV32I

```
.eqv    NF, 5
.eqv    NC, 6
.data
        .align  2
mat:    .space  120      # int mat[NF][NC]: NF*NC*4 = 120 bytes
```

EC 2.3

A **EC**, a la teoria, als problemes i als exàmens, si no s'indica explícitament el contrari, **s'admeten expressions aritmètiques als operands** (`.space NF*NC*4`, `li t0, NC*4`, `la t0, vec+8`): fan visible l'estructura del càlcul, que és el que s'hi vol ensenyar. Un `NC*4` diu què és el valor; un `24`, no.
Al **laboratori**, en canvi, cal escriure-hi sempre el **literal ja calculat**, amb l'expressió aritmètica al comentari (RISC-V 2.11). El cas de `la rd, etiqueta+desplaçament` té una substitució pròpia —la parella `la` + `addi`— i es detalla a EC 2.6.

2.3.5 Instruccions, pseudoinstruccions i macros

2.3.5.1 Instruccions

Les instruccions són les operacions que el processador executa directament. Cada **instrucció** té una codificació binària única i es **correspon amb una operació real del maquinari**, tal com s'ha vist a la Secció 2.2. RISC-V 2.12 mostra les **instruccions d'aritmètica** entera bàsica d'**RV32I**. Es presenten en aquest punt perquè n'hi ha **només tres** i són força **intuïtives**. D'aquesta manera es disposarà, d'ara endavant, d'un joc d'instruccions mínim per il · lustrar conceptes i altres instruccions a mesura que es vagin introduint. L'aritmètica entera es completa al Tema 4, amb la pseudoinstrucció

`neg` i l'extensió M (multiplicació i divisió).

RISC-V 2.12: RV32I ISA — Instruccions d'aritmètica entera

Mnemònic	Operands	Operació	Nom	Tipus
`add`	`rd, rs1, rs2`	$rd \leftarrow rs1 + rs2$	*add*	R
`addi`	`rd, rs1, imm12`	$rd \leftarrow rs1 + SignExt(imm12)$	*add immediate*	I
`sub`	`rd, rs1, rs2`	$rd \leftarrow rs1 - rs2$	*subtract*	R

En aquestes instruccions tots els enters són en **Ca2**. A `add` i `sub`, en ser de Tipus-R, tant els operands (`rs1` i `rs2`) com el resultat (`rd`) són registres i, per tant, tots tres són de 32 bits. A `addi`, el primer operand (`rs1`) és un registre, però el segon és un **immediat de 12 bits** (`imm12`) que s'estén amb signe a 32 bits ($SignExt(imm12)$) abans de ser sumat al registre font. El resultat també és un registre (`rd`).

EC 2.4

L'**extensió de rang** és l'operació de pas de la representació d'un nombre a una altra de més bits. Per exemple, a RV32I és freqüent l'extensió de rang de 12 bits a 32.

EC 2.5

A **EC**, es fa l'extensió per l'**esquerra** i només de dues maneres:

- **Extensió per zero** ($ZeroExt()$): Afegeix tants **zeros** que calgui.
- **Extensió de signe** ($SignExt()$): **Replica** el bit més significatiu (el de signe en Ca2) tantes vegades que calgui.

Essencial 2.7

A **RV32I**, les operacions de suma i resta **ignoren** el desbordament (*overflow*): el resultat és simplement els 32 bits de menys pes de l'operació, independentment de si el resultat és correcte o no en el rang dels enters representables.

Essencial 2.8

A **RV32I** no existeix la instrucció `subi`. La resta amb immediat es fa amb `addi` i un immediat negatiu: `addi rd, rs1, -N`.

Exemple 2.4: Operacions aritmètiques bàsiques

Sigui la sentència en C:

C

```c
f = (g + h) - (i - 100);
```

Suposem que `f`, `g`, `h`, `i` són variables locals enteres emmagatzemades als registres `t0`, `t1`, `t2`, `t3`:

RV32I

```
add     t4, t1, t2      # t4 = g + h
addi    t5, t3, -100    # t5 = i - 100
sub     t0, t4, t5      # f = (g + h) - (i - 100)
```

RISC-V 2.13: RV32I ISA — Instrucció `lui`

Mnemònic	Operands	Operació	Nom	Tipus
`lui`	`rd, imm20`	$rd \leftarrow imm20 << 12$	*load upper immediate*	U

`lui` carrega un immediat de 20 bits als **20 bits superiors** del registre destinació i posa els 12 bits inferiors a zero. Aquesta instrucció és especialment útil per a la càrrega d'immediats de 32 bits, que es poden construir combinant `lui` amb una instrucció de suma immediata `addi` per afegir els bits inferiors, que és el que fa la pseudoinstrucció `li` (RISC-V 2.16).

RISC-V 2.14: RV32I ISA — Instrucció `auipc`

Mnemònic	Operands	Operació	Nom	Tipus
`auipc`	`rd, imm20`	$rd \leftarrow PC + (imm20 << 12)$	*add upper immediate to PC*	U

`auipc` col · loca l'immediat de 20 bits als **20 bits superiors**, com `lui`, però en lloc d'escriure'l directament al registre destinació **hi suma el valor del PC**, que és l'adreça de la mateixa instrucció `auipc`. El desplaçament codificat és **relatiu al PC**: la mateixa instrucció dona l'adreça correcta, es carregui on es carregui el programa en memòria.

2.3.5.2 Pseudoinstruccions

El programa assemblador, seguint les recomanacions de l'especificació de RISC-V, pot definir **pseudoinstruccions** (***pseudoinstructions***) que amplien el repertori (d'instruccions) amb una sintaxi més llegible. En el moment de ser assemblada, cada pseudoinstrucció s'**expandeix en una o més instruccions de llenguatge màquina**.

RISC-V 2.15: Pseudoinstruccions — Còpia entre registres: `mv`

La pseudoinstrucció `mv` (***move***) copia el valor d'un registre a un altre:

Pseudoinstrucció	Operació	Expansió
`mv rd, rs`	$rd \leftarrow rs$	`addi rd, rs, 0`

`RV32I`

```
mv      t1, t2
```

Nota: **RARS** expandeix `mv` com `add rd, zero, rs` en lloc d'`addi rd, rs, 0`. Les dues expansions són funcionalment idèntiques: la primera usa una suma entera amb el registre `zero`, la segona una suma amb immediat zero. La diferència és visible al panell *Basic* de RARS, però no té cap efecte sobre el resultat.

RISC-V 2.16: Pseudoinstruccions — Càrrega d'immediat a registre: `li`

La pseudoinstrucció `li` (***load immediate***) inicialitza un registre amb un valor literal de fins a 32 bits. S'expandeix de manera diferent segons la magnitud del valor:

Pseudoinstrucció	Operació	Condició	Expansió
`li rd, imm12`	$rd \leftarrow imm12$	$-2048 \leq imm12 \leq 2047$	`addi rd, zero, imm12`
`li rd, imm32`	$rd \leftarrow imm32$	$lo_{12}(imm32) = 0$	`lui rd, hi_20(imm32)`
`li rd, imm32`	$rd \leftarrow imm32$	$lo_{12}(imm32) \neq 0$	`lui rd, hi_20(imm32)` `addi rd, rd, lo_12(imm32)`

On $hi_{20}(imm32)$ són els 20 bits superiors i $lo_{12}(imm32)$ els 12 bits inferiors de l'immediat. Si $lo_{12}[11] = 1$, l'assemblador incrementa hi_{20} en 1 per compensar l'extensió de signe d'`addi`.

- **Exemple 12 bits**: `li t1, 100` → `addi t1, zero, 100`
- **Exemple 32 bits,** $lo_{12} = 0$: `li t1, 0x11223000` → `lui t1, 0x11223`
- **Exemple 32 bits,** $lo_{12} \neq 0$: `li t1, 0x11223344` → dues instruccions:
  1. `lui t1, 0x11223` — carrega els 20 bits superiors; els 12 bits baixos queden a zero.
  2. `addi t1, t1, 0x344` — afegeix els 12 bits inferiors.
- **Nota**: RARS no aplica el cas $lo_{12} = 0$: genera sempre `lui` + `addi`, també quan l'`addi` suma 0, i cal comptar-hi dues instruccions.
- **Exemple amb ajust**: `li t1, 0x11223800` — com que $lo_{12}[11] = 1$, l'assemblador genera:
  1. `lui t1, 0x11224` — valor ajustat (0x11223 + 1) per compensar l'extensió de signe.
  2. `addi t1, t1, -2048` — el valor immediat de 12 bits és -2048 (0x800 en Ca2 de 12 bits); l'extensió de signe produeix el resultat correcte.

RISC-V 2.17: Pseudoinstruccions — Càrrega d'adreça de memòria: `la`

La pseudoinstrucció **`la`** (***load address***) inicialitza un registre amb una adreça de memòria constant, expressada com una etiqueta:

Pseudoinstrucció	Operació	Expansió

`la rd, label`	$rd \leftarrow @label$	`auipc rd, hi_20(@label - PC)`
		`addi rd, rd, lo_12(@label - PC)`
		(amb l'ajust de hi_{20} si $lo_{12}[11] = 1$)

RV32I

```
.data
...
y:      .word   14, -1 # Es suposa que y està a l'adreça 0x10010024

.text
        la      t0, y           # t0 <- @y        (0x10010024)
        addi    t1, t0, 4       # t1 <- @y + 4    (0x10010028)
```

`la` s'expandeix en dues instruccions amb adreces **relatives al PC**: `auipc`, que suma al PC la part alta del desplaçament fins a l'etiqueta, i `addi`, que hi suma la part baixa:

RV32I

```
        auipc   rd, hi_20(offset)     # rd ← PC + (hi_20(offset) << 12)
        addi    rd, rd, lo_12(offset) # rd ← rd + lo_12(offset)
```

on `offset = @label - PC` és la distància entre l'etiqueta i la instrucció `auipc`, i `hi_20` i `lo_12` en són la part alta i baixa respectivament (amb l'ajust de signe habitual si `lo_12[11] == 1`).
Exemple: `la t0, AA` amb `@AA = 0x10010000` i `auipc` al PC `0x00400000`:

- `offset = 0x10010000 - 0x00400000 = 0x0fc10000`
- `hi_20 = 0x0fc10`, `lo_12 = 0x000`
- `auipc x5, 0x0fc10` → `x5 = 0x00400000 + 0x0fc10000 = 0x10010000`
- `addi x5, x5, 0` → `x5 = 0x10010000`

La raó d'usar `auipc` (RISC-V 2.14) en lloc de `lui` (RISC-V 2.13) és que, amb adreces relatives al PC, el codi enllaçat és **independent de la posició** (*position-independent*): el desplaçament entre la instrucció i la dada no canvia, es carregui on es carregui el programa en memòria. L'enllaçador l'ha de calcular igualment, com ho fa amb l'adreça absoluta de `lui` (vegeu Secció 3.10).

! EC 2.6

Cas particular d'EC 2.3. A **EC**, a la teoria, als problemes i als exàmens, si no s'indica explícitament el contrari, s'admet la sintaxi més llegible següent:

Notació EC	Operació	Equivalent a RARS (laboratori)
`la rd, label+offset`	$rd \leftarrow @label + offset$	`la rd, label` `addi rd, rd, offset`

Però **RARS no la suporta**. Per tant, al **laboratori** cal substituir-la per `la` + `addi`:

RV32I

```
        # Notació EC (pissarra i exàmens):
        la      t1, y+4         # t1 <- @y + 4

        # Equivalent a RARS (laboratori):
        la      t0, y
        addi    t1, t0, 4       # t1 <- @y + 4
```

RISC-V 2.18: Pseudoinstruccions — Llistes i expansions

Les pseudoinstruccions no formen part de l'ISA. Estan documentades al RISC-V Assembly Programmer's Manual [8], *Chapter 29. A listing of standard RISC-V pseudoinstructions* i *Chapter 30. Pseudoinstructions for accessing control and status registers* (Tema 9: Excepcions i interrupcions). La mateixa especificació de l'ISA no privilegiada remet explícitament a aquest manual per a la sintaxi del llenguatge assemblador. El manual cobreix les pseudoinstruccions, els noms de registre de l'ABI i les directives d'assemblador.

Aprofundiment 2.6: RISC-V — Pseudoinstruccions: estatus de l'especificació

L'Assembly Programmer's Manual es manté sota `riscv-non-isa` (especificacions no-ISA), cosa que significa que no és una especificació ISA ratificada en el mateix sentit que l'ISA no privilegiada. És un document de referència normatiu, però no figura a la Ratified Specifications Library. Defineix un conjunt estàndard de pseudoinstruccions, però el seu estatus és més aviat estandarditzat per convenció i consens de les cadenes d'eines (*toolchains*) que no pas ratificat pel *Technical Steering Committee*.

2.3.5.3 Macros

Una **macro** és una seqüència d'instruccions amb nom i paràmetres, definida pel programador dins del seu programa. Cada cop que s'invoca, l'assemblador la **substitueix** pel seu contingut, amb els paràmetres formals reemplaçats pels reals. A diferència d'una pseudoinstrucció, una macro no forma part del repertori de l'assemblador: cal definir-la explícitament al programa.

Aprofundiment 2.7: Directives — Macros: `.macro` i `.end_macro`

A **RARS**, les macros es defineixen amb les directives `.macro` i `.end_macro`:

RV32I

```
.macro nom_macro (%param1, %param2)
    # cos de la macro
.end_macro
```

Exemple: macro que inicialitza un element d'un vector global d'enters a zero, donat l'índex. L'escalat de l'índex es fa amb un desplaçament de bits (Secció 3.1):

RV32I

```
.macro zero_elem (%base, %idx)
        slli    t6, %idx, 2         # desplaçament = idx * 4 (slli, T3)
        add     t6, %base, t6       # adreça de l'element
        sw      zero, 0(t6)         # mem[base + idx*4] = 0
.end_macro

.data
v:      .word   1, 2, 3, 4, 5

.text
        la      t0, v
        li      t1, 2
        zero_elem (t0, t1)      # v[2] = 0
        li      t1, 4
        zero_elem (t0, t1)      # v[4] = 0
```

Les macros són útils quan una mateixa seqüència d'instruccions apareix repetidament amb paràmetres variables. Si la seqüència és sempre idèntica, és preferible una subrutina: les macros dupliquen el codi a cada invocació, mentre que una subrutina s'executa des d'un sol punt en memòria.
Restriccions de RARS:

- No es poden definir macros dins d'altres macros.
- Una macro només pot invocar macros definides **anteriorment** al codi.
- Dues macros amb el mateix nom però diferent nombre de paràmetres es consideren macros diferents.

2.3.6 Comentaris

En l'assemblador de RISC-V (RARS i GNU `as`), els comentaris s'introdueixen amb el caràcter `#`. Tot el text a partir del `#` fins al final de la línia és ignorat per l'assemblador.

2.3.7 Format del codi

Essencial 2.9

Un **format acurat** facilita la comprensió i la gestió del codi.

EC 2.7: Format del codi

A **EC**, es recomana:

- Seguir sempre els mateixos criteris.
- Escriure, en C, els comentaris línia per línia amb `//`, també quan n'ocupen més d'una (atenció: el parell `/* */`, que desactiva un bloc sencer de codi, no admet la **imbricació** (*nesting*): `/* … /* … */ … */` no compila).
- Afegir els comentaris necessaris per facilitar la lectura del codi. Dos criteris informals: afegir els comentaris que al programador li agradaria trobar si el codi fos d'un altre programador, o si ell mateix hi hagués de fer modificacions al cap d'un cert temps.
- En assemblador, seguir els criteris de sagnat següents:
  - **Sense sagnat** (marge esquerre):
    - Directives de segment: `.text`, `.data`.
    - Directives de visibilitat: `.globl`, `.extern`.
    - Etiquetes: `vec:`, `main:`, `bucle:`.
  - **Amb sagnat** (8 espais):
    - Directives de dades: `.word`, `.byte`, `.asciz`, `.space`.
    - Instruccions: `add`, `lw`, `li`.
  - **Indistint**:
    - Directiva `.eqv`.
    - Directiva `.align` (sovint es sagna per indicar que afecta la dada o instrucció següent).
- En assemblador, alinear els operands en columna: el mnemònic o la directiva ocupen un camp de 8 caràcters, de manera que els operands comencen 16 caràcters després del marge,

com amb dos tabuladors de 8. Si el mnemònic en té 8 o més (`fcvt.w.s`), n'hi ha prou amb un espai. Els operands se separen amb una coma i un espai.

Aquests criteris no s'avaluen per si mateixos (EC 1.1): als exàmens el codi s'escriu a mà, i a les pràctiques es valora que el codi sigui llegible, sagnat i comentat.

Exemple 2.5: Fragment de codi d'assemblador RV32I

RV32I

```
        .eqv    MIDA_VECTOR, 100

.data
var:    .half   42
        .align  2
vect:   .space  400

.text
.globl  suma
suma:
        add     a0, a0, a1
        ret
```

2.3.8 Esquelet d'un programa

L'esquelet mínim d'un programa en assemblador per a RARS és el següent.

RISC-V 2.19: RARS — Esquelet mínim d'un programa

esquelet.s

```
.data                   # Declaració (i inicialització) de variables globals

.text
                        # Instruccions

        li      a7, 93          # syscall 93: exit2
        li      a0, 0           # codi de sortida: 0
        ecall
```

El `.text` no porta cap etiqueta d'entrada: l'execució comença a la seva **primera instrucció** (Secció 2.3.1).

2.4 Variables escalars

Essencial 2.10

En C i en RV32I, les variables es classifiquen en dos grans grups segons la seva estructura:

- **Variables escalars**:
  - Contenen un **únic valor** d'un tipus bàsic (`char`, `int`, `float`, etc.).
  - La mida és fixada pel tipus (EC 2.8).
  - Emmagatzemades preferentment en **registres** per qüestions d'eficiència; a **memòria** quan no hi ha registres disponibles.
- **Variables estructurades**:
  - Agrupen **múltiples valors** en una sola entitat (`int v[10]`, `struct`, etc.)
  - A **EC**, de variables estructurades només s'estudien els **vectors** (Secció 2.8) i les **matrius** (Secció 4.5).

2.4.1 Declaració

En **C**, tota variable s'ha de **declarar** abans d'usar-la. La declaració especifica el **tipus** i el **nom** de la variable, i opcionalment un **valor inicial**:

C

```
<tipus> <nom> [= valor_inicial];
```

El **tipus** determina la mida en memòria (EC 2.8, EC 2.9) i les operacions vàlides sobre la variable. El **nom** és l'identificador simbòlic que el compilador substitueix per una adreça de memòria o un registre.

2.4.2 Tipus

En **C**, el **tipus** d'una variable determina:

- La **mida** en memòria (en bytes).
- El **rang** de valors representables.
- Les **operacions** aplicables.

Els tipus bàsics d'**EC** es presenten a les taules EC 2.8 i EC 2.9.

Aprofundiment 2.8: Truc brut per esbrinar el tipus d'una variable

En **C**, **no existeix** un mecanisme en temps d'execució (com el `typeof` de JavaScript o el `type()` de Python) per imprimir el nom d'un tipus, ja que C és un llenguatge de tipus estàtics on la informació de tipus desapareix després de la compilació.
Un **truc brut** (*dirty hack*) és cometre errades al codi deliberadament (conflicte de tipus en aquest cas) perquè el compilador (que sí que sap el tipus) emeti avisos (*warnings*) o errors durant la compilació.
Exemple — ⚠forçar un error per esbrinar un tipus⚠

Codi 2.1: Un `int` passat a `printf` amb el format `%f`.
⚠`codi_erroni__gcc_tipus.c`⚠

```c
#include <stdio.h>

int main(void) {
    int x = 42;
    printf("%f\n", x);  // ⚠ ERROR: format '%f' espera 'double', l'argument és 'int' ⚠
    return 0;
}
```

El Codi 2.1, en ser compilat,

`Terminal`

```
gcc codi_erroni__gcc_tipus.c -Wall -Wextra -Wpedantic -o /dev/null
```

genera un avís similar al del Codi 2.2:

Codi 2.2: Avís de `gcc` sobre el format `%f` i l'argument `int`.
`Terminal`

```
codi_erroni__gcc_tipus.c: In function ‘main’:
codi_erroni__gcc_tipus.c:5:14: warning: format ‘%f’ expects argument of type ‘double’, but
↪  argument 2 has type ‘int’ [-Wformat=]
    5 |     printf("%f\n", x);  // ⚠ ERROR: format '%f' espera 'double', l'argument és 'int'
     ↪ ⚠
      |             ~^     ~
      |              |     |
      |              |     int
      |              double
      |             %d
```

2.4.3 Visibilitat i durada

En **C**, les variables s'han de declarar explícitament abans d'emprar-les. La declaració ha d'incloure el **tipus de dades** (*data type*; en **C**, els tipus bàsics són: `char`, `short`, `int`, `long`, `long long`, `float`, `double` i `long double`) i el **nom**, i opcionalment un **valor inicial** (inicialització de variable). Una variable que es declara fora de qualsevol funció és una **variable global** i una que es declara dins d'un bloc o funció és una **variable local** (***local variable***) al bloc o funció (l'àmbit, *scope*). Aquestes dues classes de variables tenen assignades per defecte propietats de **visibilitat** i de **durada** (o **categoria d'emmagatzemament**) següents:

- **Variables globals**: Tenen **visibilitat global** i **durada permanent** (en **C**, categoria d'emmagatzemament *static*). Existeixen durant tota l'execució del programa i s'emmagatzemen en posicions de memòria fixes. En **assemblador**, la seva declaració ha d'aparèixer al segment de dades `.data`.
- **Variables locals**: Tenen **visibilitat local** i **durada temporal**. Es creen cada cop que s'executa el bloc on estan declarades i deixen d'existir en acabar l'àmbit on s'han declarat. No tenen un espai d'emmagatzematge fix. Poden residir en registres o en memòria.

⚠ Aprofundiment 2.9: C: variable local amb durada permanent (`static`)

Una variable local declarada amb la paraula clau `static` és una excepció a la regla general: té **visibilitat local** (només és accessible dins del bloc on està declarada) però **durada permanent** (es crea en iniciar el programa i no es destrueix en sortir del bloc).

C

```c
void f() {
    static int comptador = 0;  // inicialitzat una sola vegada
    comptador++;
}
```

Cada invocació de `f()` incrementa el mateix `comptador`, que persisteix entre crides.

2.4.4 Mida i rang de valors representables

En **C**, la **mida** en bytes d'una variable **depèn** del seu **tipus** i de l'**ABI**. Per a l'ABI d'EC (`ilp32`), les mides són les de les taules següents:

! EC 2.8: Mida dels tipus naturals

Tipus C	Mida	Rang	Rang
`unsigned char`	1 byte	$[0, 2^8 - 1]$	[0, 255]
`unsigned short`	2 bytes	$[0, 2^{16} - 1]$	[0, 65 535]
`unsigned int`	4 bytes	$[0, 2^{32} - 1]$	[0, 4 294 967 295]
`unsigned long`	4 bytes	$[0, 2^{32} - 1]$	[0, 4 294 967 295]
`unsigned long long`	8 bytes	$[0, 2^{64} - 1]$	[0, 18 446 744 073 709 551 615]

! EC 2.9: Mida dels tipus enters

Tipus C	Mida	Rang	Rang
`char`	1 byte	$[-2^7, 2^7 - 1]$	[-128, 127]
`short`	2 bytes	$[-2^{15}, 2^{15} - 1]$	[-32 768, 32 767]
`int`	4 bytes	$[-2^{31}, 2^{31} - 1]$	[-2 147 483 648, 2 147 483 647]
`long`	4 bytes	$[-2^{31}, 2^{31} - 1]$	[-2 147 483 648, 2 147 483 647]
`long long`	8 bytes	$[-2^{63}, 2^{63} - 1]$	[-9 223 372 036 854 775 808, 9 223 372 036 854 775 807]

! EC 2.10

A **EC**, **`char` és un tipus enter, amb signe** (com `signed char`). L'ABI de RISC-V el defineix sense signe; amb `gcc`, l'opció `-fsigned-char` hi aplica la convenció d'EC.

Essencial 2.11

Com que els registres són de 32 bits, per allotjar-hi un `long long` en calen dos, un per als bits de menys pes, l'altre pels de més pes; cal gestionar-los corresponentment.

Aprofundiment 2.10: C — Tuples

En **C**, no tots els tipus de dades tenen una mida potència de 2. Les tuples (representades en C pel tipus `struct`) són agrupacions d'elements («camps») de tipus heterogenis, on la mida de la tupla depèn dels tipus dels camps que la formen. Cada camp de la tupla ha d'estar alineat en memòria segons el seu tipus, i, per tant, poden aparèixer no solament espais de memòria no usats entre els camps, sinó també entre elements consecutius d'un vector de tuples. A EC, no es tracten les tuples.

2.5 La memòria

RISC-V 2.20: RV32I ISA — Model de memòria — Característiques bàsiques

- És **unificat**: Instruccions i dades comparteixen el mateix espai d'adreces (model de von Neumann, Secció 1.3).
- L'**espai d'adreces** és de **32** bits. Per tant, el rang adreçable és [`0x00000000`, `0xFFFFFFFF`].
- Adreçament per **bytes**: Cada adreça identifica **una posició** de memòria que conté **1 byte** (8 bits).
- És de **lectura/escriptura**: Accés a memòria exclusivament mitjançant instruccions de lectura i escriptura (`lw`/`sw` i variants).
- És **little-endian**: El byte de menys pes ocupa l'adreça més baixa.
- Les adreces del programa **poden ser virtuals**: en un sistema amb sistema operatiu, la unitat de gestió de memòria (*memory management unit*, MMU) les tradueix a adreces físiques (Secció 8.1.2, Tema 8). A **RARS** no hi ha traducció: les adreces del programa **són** directament les adreces físiques.

Aprofundiment 2.11: Model de memòria relaxat i instruccions `FENCE`

RISC-V defineix un **model de memòria relaxat** (*RISC-V Weak Memory Ordering*, RVWMO) que regula l'ordre en què esdevenen visibles els accessos a memòria entre múltiples nuclis o **fils d'execució de maquinari** (***harts***). Les instruccions `fence` i `fence.tso` permeten imposar restriccions d'ordenació en aquests contextos concurrents, i `fence.i` (extensió **Zifencei**) sincronitza la lectura d'instruccions després de modificar el codi en temps d'execució.
A **EC** s'estudia un **únic processador**, sense concurrència, sense dispositius que accedeixin a la memòria pel seu compte i sense codi que es modifiqui a si mateix: en aquest context, aquestes instruccions no tenen cap efecte observable. Per aquest motiu queden fora de l'abast de l'assignatura.

Essencial 2.12

Per operar amb una dada emmagatzemada en memòria, cal **sempre** llegir-ne primer el valor, abans de qualsevol instrucció de l'operació, i deixar-lo en un registre mitjançant la **instrucció de lectura** adient.

Per escriure qualsevol resultat a memòria, abans de qualsevol **instrucció d'escriptura** primer cal **sempre** obtenir el resultat.

! EC 2.11: @ — l'operador «adreça de»

A **EC**, en **pseudocodi**, s'empra el símbol arrova, **@** (es llegeix com **adreça de**) seguit del nom d'una **variable** per referir-se a l'**adreça de memòria** d'aquesta variable. Per exemple, **@x** representa l'adreça de la variable `x`.
No s'ha de confondre amb l'operador de C `&` (Secció 2.7.3).

Essencial 2.13

Habitualment, les **adreces** de memòria s'expressen en **hexadecimal** (prefix **0x**).

! EC 2.12: Sentit creixent de les adreces

A **EC**, les adreces de memòria creixen **de dalt cap a baix** en tots els diagrames.[3]

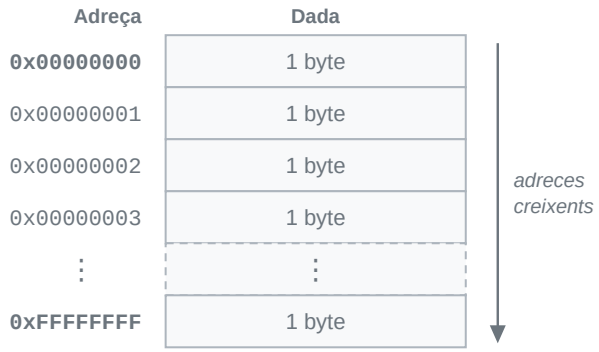


Figura 2.1: Representació gràfica de la memòria.

Essencial 2.14

Les **adreces** només poden ser **nombres naturals**.

2.5.1 Ordre dels bytes (*endianness*)

L'**ordre dels bytes** (***endianness***, o *byte ordering*) estableix com s'emmagatzemen a la memòria els bytes d'una dada de més d'un byte. En ***little-endian***, el byte de menys pes (*least significant byte*, LSB) s'emmagatzema a l'adreça més baixa, mentre que el byte de més pes (*most significant byte*, MSB) s'emmagatzema a l'adreça més alta. En ***big-endian***, és al revés: l'MSB ocupa l'adreça més baixa i l'LSB l'adreça més alta.

[3] Aquesta és una convenció gràfica arbitrària, i altres fonts, com el llibre recomanat [11], adopten justament la convenció contrària: les adreces dels diagrames creixen de baix cap a dalt.

Essencial 2.15

Taula 2.3: Terminologia: la b minúscula, per al bit, i la B majúscula, per al byte

Terme	Àmbit
Bit de menys pes (*least significant bit*, LSb)	Bit
Byte de menys pes (*least significant byte*, LSB)	Byte
Bit de més pes (*most significant bit*, MSb)	Bit
Byte de més pes (*most significant byte*, MSB)	Byte

Exemple 2.6: *Little-endian* vs. *Big-endian*, ocupació de les posicions de memòria de la paraula **0x11223344**

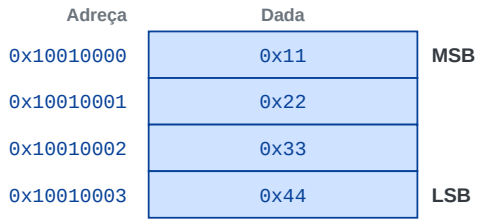


Figura 2.2: *Big-endian*.

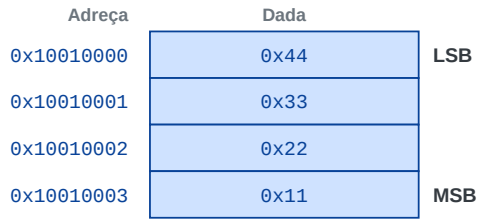


Figura 2.3: *Little-endian*.

Ordre creixent d'adreces cap a baix (criteri EC).

EC 2.13: Només *Little-endian*

A **EC**, **només** s'utilitza l'ordre ***little-endian*** (en tots els exemples, problemes i laboratoris).

Aprofundiment 2.12: Ordre dels bytes — Regla mnemotècnica (regla de la lletra pi)

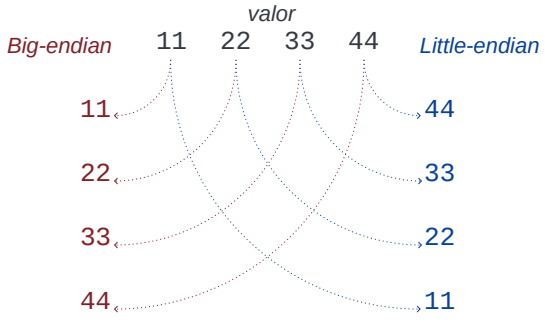


Figura 2.4: Regla mnemotècnica de la lletra grega pi (Π) per recordar l'ordre dels bytes *little-endian* i *big-endian*.

2.5.2 Restriccions d'alineació

Les **restriccions d'alineació** (***alignment constraints***) són regles que imposen condicions sobre l'**adreça inicial** de les dades emmagatzemades en memòria.

Essencial 2.16

A **EC**, es segueixen les restriccions definides per l'**ABI** `ilp32` de RISC-V.

RISC-V 2.21: RV32I ABI — Restriccions d'alineació

Tipus C	Mida	Adreça inicial múltiple de…
`char`, `unsigned char`	1 B	1 (cap restricció)
`short`, `unsigned short`	2 B	2
`int`, `unsigned int`, `float`	4 B	4
`long long`, `unsigned long long`, `double`	8 B	8

A **RARS**, quan es declaren variables globals consecutivament, les directives `.half` i `.word` insereixen automàticament bytes de **farciment** (*padding*) entre elles per respectar aquestes restriccions (l'assemblador de GNU no ho fa: és el compilador qui hi posa un `.align`). El valor dels bytes de farciment no té cap significat, i el programa no n'ha de dependre.

Exemple 2.7: Restricció d'alineació, correcte vs. incorrecte

Al Codi 2.3, a les línies **L12** i **L13** hi ha **errors** d'alineació.

Codi 2.3: Tres lectures alineades i dues que no ho són.
⚠`codi_erroni__alineacio_incorrecte.s`⚠

```
.data
AA:     .word   0x41424344
BB:     .word   0x45464748

.text
        la      t0, AA          # t0 <- @AA  (0x10010000)
        lb      t1, 1(t0)       # t1 <- 0x00000043  ('C')
        lh      t2, 2(t0)       # t2 <- 0x00004142
        lw      t3, 4(t0)       # t3 <- 0x45464748

###   CODI INCORRECTE   ################################################
        lh      t4, 5(t0)       # ⚠ ERROR: excepció d'alineació ⚠
        lw      t4, 1(t0)       # ⚠ ERROR: excepció d'alineació ⚠
```

En executar el Codi 2.3 a RARS (`F7`: execució línia a línia), en arribar a la **L12** salta una **excepció d'alineació** i s'atura l'execució:

```
RARS → Execute → Messages
```

```
Error in /.../codi.s line 12: Runtime exception at 0x00400014: Load address not aligned on
↪  halfword boundary 0x10010005
```

Essencial 2.17

No respectar l'alineació:

- A **EC**, es considera un error greu.
- A **RARS**, genera una excepció i atura l'execució.

2.5.3 Directives d'alineació de memòria

RARS té les **directives** de RISC-V 2.22 per gestionar les restriccions d'alineació de memòria. Aquestes directives apliquen automàticament la restricció d'alineació, descartant tantes posicions de memòria com sigui necessari, tret de `.dword` (vegeu-ne la nota a RISC-V 2.22). Opcionalment, la directiva pot anar **precedida** d'una **etiqueta** amb el **nom de la variable**, la qual representa de forma simbòlica l'adreça inicial de la dada definida.[4]

RISC-V 2.22: Directives — Inicialització de variables globals: `.byte`, `.half`, `.word`, `.dword`

Directiva	Operands	Emmagatzema `v1, v2, ...` en...
`.byte`	`v1, v2, ...`	bytes de 8 bits
`.dword`	`v1, v2, ...`	paraules dobles de 64 bits
`.half`	`v1, v2, ...`	mitges paraules de 16 bits
`.word`	`v1, v2, ...`	paraules de 32 bits

Cal especificar almenys un operand; per inicialitzar a zero s'usa p. ex. `.word 0`. RARS no avisa si no n'hi ha cap: no reserva res, i l'etiqueta queda amb l'adreça de la dada següent.
A **RARS**, `.dword` només s'alinea a **4 bytes**, no a 8: per a un `long long` o un `double`, que l'ABI alinea a 8 (RISC-V 2.21), cal la directiva `.align 3` al davant.
(Vegeu també RISC-V 2.31.)

Exemple 2.8: Traducció de declaracions de variables globals

C

```c
unsigned char p;
short q = 13;
char r = -1;
char s = 10;
int t = 0x10AA00FF;
long long u = 0x7766554433221100;
```

RV32I

```
.data
p:      .byte   0               # unsigned char p;
q:      .half   13              # short q = 13;
r:      .byte   -1              # char r = -1;
s:      .byte   10              # char s = 10;
t:      .word   0x10AA00FF      # int t = 0x10AA00FF;
```

[4] L'assemblador, quan pot (Secció 3.10), substitueix totes les etiquetes (variables, funcions, destinacions de salts, etc.) per adreces de memòria.

```
        .align  3                   # long long: alineació a 8
u:      .dword  0x7766554433221100  # long long u = 0x7766554433221100;
```

! EC 2.14: Noms dels tipus de variable, tant C com RARS

A **EC**, es fa referència als tipus de variables usant tant la terminologia de C (`char`, `short`, etc.) com la de RARS (`.byte` o byte, `.half` o *half*, etc.).

2.6 Operands i modes d'adreçament

La majoria d'instruccions en **RV32I**, igual que en la majoria de processadors **RISC**, tenen tres **operands** explícits. Aquesta regularitat respon al **principi de disseny** (el primer de [11], *simplicity favors regularity*) que la **regularitat** (***regularity***) en el disseny afavoreix implementacions més simples i més ràpides, encara que algunes instruccions poden tenir-ne menys o, fins i tot, cap.

La manera com s'especifica un operand en una instrucció rep el nom de **mode d'adreçament** (***addressing mode***). Tot i que l'especificació oficial d'**RV32I** no explicita formalment els modes d'adreçament, sinó que descriu el comportament de cada instrucció individualment, aquesta caracterització és útil per a comprendre i categoritzar les diferents maneres d'especificar els operands en les instruccions.

! EC 2.15

A **EC**, es distingeixen **quatre modes d'adreçament**:

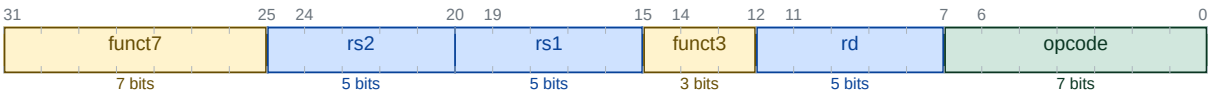
1. **Mode registre** (***register mode***): Els operands es troben en els registres de propòsit general (`x0`–`x31`). La majoria d'instruccions aritmeticològiques operen d'aquesta manera.
2. **Mode immediat** (***immediate mode***): S'utilitza un valor constant inclòs directament en la instrucció. Per exemple, `addi` suma un valor immediat de 12 bits amb signe a un registre; `lui` carrega un immediat de 20 bits als bits superiors del registre destinació.
3. **Mode base + desplaçament** (***base + offset mode***): L'adreça efectiva es calcula sumant un valor immediat, el **desplaçament** (*offset*), a un registre base. S'usa en instruccions de tipus *load*/*store* (`lw`, `sw`, etc.) i també en salts incondicionals indirectes (`jalr`), on el resultat és una adreça de salt en lloc d'una adreça de memòria.
4. **Mode relatiu al PC** (***PC-relative mode***): L'adreça efectiva es calcula sumant un desplaçament al valor actual del **PC**. S'usa en les instruccions de salt condicional (`beq`, `bne`, etc.) i la de salt incondicional relatiu al PC `jal`, a més d'`auipc`, que suma un immediat al PC.

2.6.1 Operands en mode registre

En el **mode registre**, l'operand resideix en un **registre** (***register***) del processador, i la instrucció només n'especifica el número.

Les instruccions en què **tots** els operands són en mode registre són de format **Tipus-R** (RISC-V 2.23). Són instruccions de tres operands on el primer acostuma a ser l'**operand-destinació** (**rd**) i els altres dos els **operands-font** (**rs1** i **rs2**). Els camps `funct7` i `funct3`, combinats amb l'`opcode`, determinen l'operació concreta.

RISC-V 2.23: RV32I ISA — Format Tipus-R



Essencial 2.18

Com que hi ha 32 registres, cal un camp de 5 bits per identificar-los unívocament.

Exemple 2.9: Operands en mode registre

Instrucció	Operands	Operació
`add`	`rd, rs1, rs2`	$rd \leftarrow rs1 + rs2$
`sub`	`rd, rs1, rs2`	$rd \leftarrow rs1 - rs2$

L'**ISA** d'RV32I especifica 32 **registres de propòsit general** (*general-purpose registers*, GPR) de 32 bits: `x0`–`x31`. De tots ells, l'únic registre especial és `x0`, que sempre conté el valor zero i no es pot modificar. Així doncs, mantenint-se estrictament en el pla de l'ISA, la resta de registres poden ser usats per a qualsevol propòsit. RISC-V 2.24 mostra com l'**ABI** assigna els **31 registres** restants.

Essencial 2.19

L'**ISA** d'**RV32I** és **ortogonal** pel que fa als registres: tret de `x0` (fixat a zero per maquinari), **no imposa cap restricció sobre l'ús dels registres de propòsit general**. Qualsevol registre pot actuar com a operand font o destinació en qualsevol instrucció. És l'**ABI** qui estableix les convencions d'ús (registres de paràmetres, de valors de retorn, temporals, etc.).

RISC-V 2.24: RV32I Registres — Propòsit general

ISA	ABI	Propòsit	*Saver*
`x0`	`zero`	El valor constant zero	—
`x1`	`ra`	**Adreça de retorn** (***return address***) de subrutina	*Caller*
`x2`	`sp`	**Punter de pila** (***stack pointer***), adreça del cim de la **pila**	*Callee*
`x3`	`gp`	**Punter global** (***global pointer***)	—
`x4`	`tp`	**Punter de fil** (***thread pointer***)	—
`x5`–`x7`	`t0`–`t2`	**Temporals**	*Caller*
`x8`	`s0` / `fp`	**Segur** / **Punter de marc** (***frame pointer***)	*Callee*
`x9`	`s1`	**Segur**	*Callee*
`x10`, `x11`	`a0`, `a1`	**Arguments** de subrutines / Valors de **retorn**	*Caller*
`x12`–`x17`	`a2`–`a7`	**Arguments** de subrutines	*Caller*
`x18`–`x27`	`s2`–`s11`	**Segurs**	*Callee*
`x28`–`x31`	`t3`–`t6`	**Temporals**	*Caller*

La columna «*Saver*» estableix qui és responsable de preservar el valor d'un registre a través d'una crida a subrutina: la subrutina que fa la crida (*caller*) o la subrutina cridada (*callee*). Aquesta distinció es formalitza al Tema 3 amb la terminologia *caller-saved*/*callee-saved* (RISC-V 3.16), en el context de les subrutines.

! EC 2.16

En llenguatge assemblador, els registres es poden referenciar per l'identificador de l'**ISA** (`x0`–`x31`) o pel de l'**ABI**, i aquest darrer és el més habitual i el que s'usa a **EC**.

i RISC-V 2.25: RV32I Registres — Comptador de programa (*program counter*, PC)

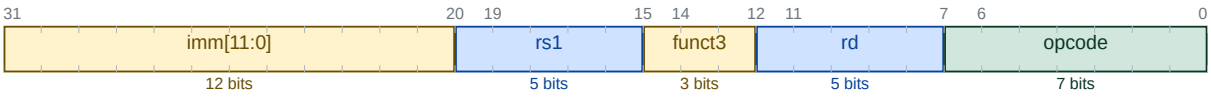
El registre `pc` no és un registre de propòsit general. És un registre que conté l'adreça de la instrucció que s'està executant; un cop finalitzada aquesta execució, el PC pren el valor de l'adreça de la instrucció següent a executar. Aquesta instrucció no ha de ser, necessàriament, la que hi ha a continuació de la que s'acaba d'executar, ja que és a través de la gestió dels valors de PC com s'implementen els salts condicionals, crides a subrutines, etc.

2.6.2 Operands en mode immediat

En el **mode immediat**, l'operand és un valor constant (l'immediat o literal) que s'inclou directament en la instrucció. En les instruccions de format Tipus-I (com `addi`), l'immediat és de **12 bits amb signe**, cosa que permet representar valors entre -2048 i 2047. Per a valors més grans, cal utilitzar una combinació d'instruccions per carregar el valor en un registre abans d'utilitzar-lo.

Les instruccions que usen el mode immediat són de format **Tipus-I** (RISC-V 2.26). En aquest format, el camp `rd` especifica el **registre destinació**, `rs1` el **registre font**, i `imm[11:0]` codifica l'**immediat de 12 bits** al qual el processador fa l'**extensió de signe** a 32 bits abans d'operar.

i RISC-V 2.26: RV32I ISA — Format Tipus-I



Essencial 2.20

A **RV32I**, als operands immediats sempre se'ls aplica l'extensió de signe, $SignExt()$, que és la **regla general** a **RISC-V**.

Les instruccions `lui` (RISC-V 2.13) i `auipc` (RISC-V 2.14) també tenen un operand immediat, però de **20 bits**, i per això no caben al format Tipus-I: són de format **Tipus-U** (RISC-V 2.27), un dels quatre formats nuclears (Secció 2.2). Pel que fa al mode d'adreçament, `auipc` és relatiu al PC (EC 2.15).

i RISC-V 2.27: RV32I ISA — Format Tipus-U



El camp `imm[31:12]` conté els 20 bits superiors de l'immediat; els 12 bits inferiors del resultat els determina la instrucció concreta (`lui` els posa a `0`, i `auipc` hi deixa els del PC). És el format amb l'immediat més gran de tota l'ISA, i no té camps `rs1`, `rs2` ni `funct`.

Exemple 2.10: Operands en mode immediat

Instrucció	Operands	Operació
`addi`	`rd, rs1, imm12`	$rd \leftarrow rs1 + SignExt(imm12)$
`ori`	`rd, rs1, imm12`	$rd \leftarrow rs1 \vee SignExt(imm12)$

Exemple 2.4 combina tots dos modes: `add` i `sub` hi fan servir operands en mode registre, i `addi`, un operand en mode immediat (-100).

2.6.3 Operands en mode base + desplaçament

Essencial 2.21: Mode base + desplaçament

En **RV32I**, el mode d'adreçament que usa la sintaxi `offset(rs1)` s'anomena **mode base + desplaçament** (*base + offset mode*) i és el terme que s'utilitza a EC. A la literatura es troba sovint com a **mode memòria** (*memory addressing mode*); tots dos noms fan referència al mateix concepte.

EC 2.17

A **EC**, els operands en **mode base + desplaçament** s'escriuen sempre **`desplaçament(base)`**, com a `lw t0, 4(s0)`, també a les instruccions que l'assemblador admet en una altra forma: el salt indirecte `jalr` (Secció 3.4.3) s'escriu `jalr rd, 0(rs1)`, i no `jalr rd, rs1, 0`. Així, la forma de l'operand identifica el mode d'adreçament a totes les instruccions que el fan servir.

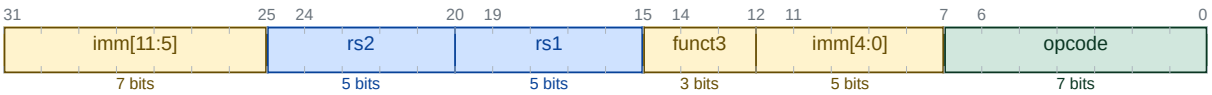
En processadors **RISC** com l'**RV32I**, només les instruccions del tipus **lectura** (***load***) i **escriptura** (***store***) admeten un **operand** resident en memòria. Aquest operand s'ha de codificar en **mode base + desplaçament**, el qual té la sintaxi `offset(rs1)`, on `rs1` és el **registre-base** que conté una adreça de memòria, i `offset` és un **desplaçament** que s'afegeix a aquesta adreça per obtenir l'**adreça efectiva** de la dada a la qual s'accedeix.

El desplaçament és un **immediat** de **12 bits** codificat en **Ca2**, que abasta el rang de valors $[-2^{11}, 2^{11}-1]$ ([-2048, +2047]). El processador **RV32I** fa **extensió de signe** a 32 bits abans de sumar-lo a la base.

Les instruccions de **lectura** (*load*) són de format **Tipus-I** (RISC-V 2.26): especifiquen un **registre destinació** (**rd**) on escriure el valor llegit, a més del registre base (**rs1**) i el desplaçament (`offset`).

Les instruccions d'**escriptura** (*store*) són de format **Tipus-S** (RISC-V 2.28): no necessiten `rd` (no escriuen a cap registre), però necessiten **dos registres font**: el registre base (`rs1`) per calcular l'adreça efectiva, i el registre font (`rs2`) que conté el valor a escriure. Els bits de l'immediat es distribueixen en dos camps separats precisament per mantenir `rs1` i `rs2` en posicions fixes, igual que en la resta de formats, cosa que simplifica la descodificació al **maquinari**.

RISC-V 2.28: RV32I ISA — Format Tipus-S



El comportament específic de moltes operacions depèn de les característiques dels operands o el resultat, sent les més habituals el tipus o la mida. Això fa que, en les arquitectures RISC, per implementar-les, calguin diverses **variants** d'un mateix tipus d'instrucció.

Per a les instruccions de lectura i escriptura, els factors rellevants i la influència en els mnemònics són:

- La direcció del flux de dades: si es tracta d'una instrucció de lectura (`l`, *load*) o d'escriptura (`s`, *store*).
- La mida de les dades a les quals s'accedeix: 4 bytes (`w`, *word*), 2 bytes (`h`, *half-word*) o 1 byte (`b`, byte).
- Si les dades a llegir són naturals (`u`, d'*unsigned*) o enters (sense lletra específica als mnemònics).

Aquesta nomenclatura es manté a totes les variants d'instruccions d'RV32I.

RISC-V 2.29 mostra totes les instruccions de **lectura** i **escriptura** d'**RV32I**.

RISC-V 2.29: RV32I ISA — Instruccions de lectura i escriptura

Mnemònic	Operands	Operació	Nom	Tipus
`lw`	`rd, offset(rs1)`	$rd \leftarrow M_w[rs1 + SignExt(offset)]$	*load word*	I
`lh`	`rd, offset(rs1)`	$rd \leftarrow SignExt(M_h[rs1 + SignExt(offset)])$	*load half*	I
`lhu`	`rd, offset(rs1)`	$rd \leftarrow ZeroExt(M_h[rs1 + SignExt(offset)])$	*load half unsigned*	I
`lb`	`rd, offset(rs1)`	$rd \leftarrow SignExt(M_b[rs1 + SignExt(offset)])$	*load byte*	I
`lbu`	`rd, offset(rs1)`	$rd \leftarrow ZeroExt(M_b[rs1 + SignExt(offset)])$	*load byte unsigned*	I
`sw`	`rs2, offset(rs1)`	$M_w[rs1 + SignExt(offset)] \leftarrow rs2$	*store word*	S
`sh`	`rs2, offset(rs1)`	$M_h[rs1 + SignExt(offset)] \leftarrow rs2[15:0]$	*store half*	S
`sb`	`rs2, offset(rs1)`	$M_b[rs1 + SignExt(offset)] \leftarrow rs2[7:0]$	*store byte*	S

EC 2.18: M_x[adreça] — Contingut del tipus x de l'*adreça*

A **EC**, M_w[adreça], M_h[adreça] i M_b[adreça] signifiquen **contingut** de la **paraula**, de la **mitja paraula** i del **byte** d'**adreça**, respectivament. En termes de bytes accedits, M_w[adreça] en són quatre ([adreça, adreça + 3]), M_h[adreça] dos ([adreça, adreça + 1]) i M_b[adreça] un (adreça).

Essencial 2.22

El mode base + desplaçament és el més lent, ja que implica un accés a memòria, però és l'únic que permet accedir a dades que no estan als registres.

Exemple 2.11: Operands en mode base + desplaçament

RV32I

```
.text
        sw      t0, 4(s1)       # Mw[s1 + SignExt(4)] <- t0
        lb      a0, 7(t1)       # a0 <- SignExt(Mb[t1 + SignExt(7)])
        lbu     a0, 7(t1)       # a0 <- ZeroExt(Mb[t1 + SignExt(7)])
```

2.6.3.1 Accessos a paraula: `lw`, `sw`

La instrucció `lw` (*load word*) copia **quatre bytes** consecutius de la **memòria**, a partir de l'**adreça efectiva** (que conté l'**LSB**, perquè RV32I és *little-endian*), al **registre** destinació. Per la seva banda, `sw` (*store word*) copia els **quatre bytes** del registre font a altretantes posicions consecutives de memòria a partir de l'adreça efectiva.

Essencial 2.23

A **RV32I** no existeix ~~`lwu`~~: el registre ja és de 32 bits i no hi ha res a estendre (sí que existeix a RV64I).

2.6.3.2 Accés a enters de mitja paraula o byte: `lh`, `lb`, `sh`, `sb`

Les instruccions `lh` (*load half*) i `lb` (*load byte*) llegeixen de la memòria, a partir de l'adreça efectiva, i copien a un registre **dos bytes** o **un byte**, respectivament. Com que el registre destinació és de 32 bits, es fa una **extensió de signe** del bit més significatiu del valor llegit (el bit de signe en el cas dels enters) per omplir els bits més significatius del registre destinació.

Per la seva banda, les instruccions `sh` (*store half*) i `sb` (*store byte*) copien els **dos bytes** o el **byte** de menys pes del registre font a memòria a partir de l'adreça efectiva.

Essencial 2.24

Les instruccions `sh` i `sb` no realitzen **cap tipus d'extensió**. Simplement copien els bits menys significatius del registre font a memòria. Per tant, si el registre font conté un valor que no s'ajusta a la mida de l'operació d'escriptura, s'escriurà un valor inesperat a memòria.

Exemple 2.12: Operands en mode base + desplaçament, accés a enters de mitja paraula o byte

Suposa que `t2` conté l'adreça `0x10010000` i que, en aquesta adreça i les següents, el contingut inicial de la memòria és el que mostra la taula següent.

Adreça	Estat inicial	Estat final
`0x10010000`	`0x11`	?
`0x10010001`	`0x22`	?
`0x10010002`	`0xCC`	?
`0x10010003`	`0xDD`	?

1. Determina el **valor del registre** resultant de l'execució de cada instrucció següent:

Instrucció	Operació	Resultat al registre
`lb t1, 1(t2)`	Lectura del byte `0x22`	`t1 = 0x00000022`
`lb t1, 2(t2)`	Lectura del byte `0xCC`	`t1 = 0xFFFFFFCC`
`lh t1, 0(t2)`	Lectura de la mitja paraula `0x2211`	`t1 = 0x00002211`
`lh t1, 2(t2)`	Lectura de la mitja paraula `0xDDCC`	`t1 = 0xFFFFDDCC`
`sb t1, 1(t2)`	Guarda el byte `0xCC` a l'adreça `0x10010001`	(Estat final de memòria)

2. Determina l'**estat final de memòria** resultant de l'execució anterior:

Adreça	Estat inicial	Estat final
`0x10010000`	`0x11`	`0x11`
`0x10010001`	`0x22`	`0xCC` ←
`0x10010002`	`0xCC`	`0xCC`
`0x10010003`	`0xDD`	`0xDD`

2.6.3.3 Lectura de naturals de mitja paraula o byte: `lhu`, `lbu`

Les instruccions `lhu` (*load half unsigned*) i `lbu` (*load byte unsigned*) copien a un registre destinació els **dos bytes** o el **byte** de memòria, respectivament, a partir de l'adreça efectiva. Com que el registre destinació és de 32 bits, es fa una **extensió per zero** afegint zeros als bits més significatius del registre destinació. Aquestes instruccions s'utilitzen per llegir valors de mida inferior a 32 bits que es consideren naturals (sense signe), com ara els `unsigned char` i els `unsigned short` de C, que no s'han d'interpretar com a enters. A **EC**, `char` és amb signe (EC 2.10), i es llegeix amb `lb`.

💡 Exemple 2.13: Operands en mode base + desplaçament, lectura de naturals de mitja paraula o byte

Considera el mateix contingut inicial de la memòria de l'exemple anterior i determina el **resultat** de l'execució de cada instrucció:

Instrucció	Operació	Resultat al registre
`lbu t1, 2(t2)`	Lectura del byte `0xCC`	`t1 = 0x000000CC`
`lhu t1, 2(t2)`	Lectura de la mitja paraula `0xDDCC`	`t1 = 0x0000DDCC`

2.6.3.4 Restricció d'alineació

Les adreces utilitzades en les instruccions `lw` i `sw` han de ser múltiples de quatre, i les adreces usades en `lh`, `lhu` i `sh` han de ser múltiples de dos. Altrament, a **RARS** es produeix una excepció per adreça no alineada i s'atura l'execució.

Exemple 2.14: Alineació

El Codi 2.4 executa un `lw` a una **adreça no alineada** i, per tant, provoca una **excepció**:

Codi 2.4: Un `lw` a una adreça que no és múltiple de 4.
`RV32I`

```
.data
x:      .word   0xDDCCAABB

.text
        la      t0, x
        addi    t0, t0, 3       # t0 = adreça base + 3
        lw      t1, 0(t0)       # Provoca una excepció d'accés no alineat!
```

2.7 Punters

Un punter és una **variable escalar** que conté una **adreça de memòria**.

RISC-V 2.30: RV32I ABI — Mida dels punters

A **RV32I**, com que l'espai d'adreces és de 32 bits (RISC-V 2.20), els punters ocupen sempre **32 bits** (**4 bytes**, una paraula) i s'alineen a 4 bytes (ABI `ilp32`).

Essencial 2.25

Els **punters** representen adreces de memòria i, per tant, **només** poden ser **nombres naturals** del **rang de memòria adreçable**.

Aprofundiment 2.13: Mida dels punters en arquitectures de 64 bits

A **x86-64** i a **RV64I**, els punters tenen 8 bytes.

2.7.1 C — Un tipus de punter per cada tipus de dades

Aritmèticament, els punters són nombres naturals. En **C**, però, no només es consideren un **tipus de dades específic**, diferent dels naturals, sinó que hi ha un **tipus de punter** per cada **tipus de dades**. Per exemple, el tipus «punter a tipus enter» és diferent del tipus «punter a tipus caràcter».

Per abús de llenguatge, es sol dir «punter a enter» o «punter a caràcter» en lloc de «punter a tipus enter» i «punter a tipus caràcter», respectivament.

2.7.2 Declaració

En **C**, en una **declaració**, el caràcter `*`, precedit del nom d'un tipus i seguit del nom d'una variable, indica «**variable de tipus punter a tipus** `<tipus>`», on `<tipus>` és el tipus que el precedeix.

En C, la sintaxi de declaració d'un punter és:

- C

```
<tipus_apuntat> *<nom_del_punter> [= adreça_d_inicialització];
```

  - **Atenció**: Els claudàtors `[]` indiquen l'opcionalitat de la inicialització (Secció 2.7.4).

Així doncs, per exemple, la declaració `int *p;` declara `p` com a «variable de tipus punter a tipus enter». Tècnicament, `p` és de tipus `int *`, això és, «punter a tipus enter» o, més habitualment, per abús de llenguatge, «punter a enter».

En les **declaracions**, el caràcter `*` és un modificador que forma part del **declarador**, `*<nom_del_punter>`, i no del nom del tipus que el precedeix.

A l'Exemple 2.15, `p1` i `p4` són del tipus `int *`, és a dir, són punters a variables de tipus `int`. Anàlogament, `p2` és un punter del tipus `char *`, és a dir, un punter a una variable de tipus `char`, mentre que `p3` és un punter del tipus `short *`, per tant, un punter a una variable de tipus `short`. Finalment, `p5` no és un punter, sinó un enter, ja que el `*` forma part del declarador de `p4` (`*p4`), no d'`int`.

Exemple 2.15: Declaració de punters en C

C

```c
int *p1;          // Punter a enter
char *p2;         // Punter a caràcter
short *p3;        // Punter a short
int *p4, p5;      // p4 és un punter a enter, però p5 és un enter!
```

Essencial 2.26

No s'ha de confondre l'ús de `*` en les **declaracions**, on denota «punter a», amb l'ús en les **sentències**, on, com a operador unari, és l'operador de desreferència o indirecció (Secció 2.7.5).

En **RV32I no hi ha una distinció explícita entre diferents tipus de punters**; tots són sempre adreces de 32 bits.

A l'Exemple 2.16 es tradueixen a RV32I les declaracions en C dels punters `p1`, `p2` i `p3` de l'Exemple 2.15 suposant que les variables `p1`, `p2` i `p3` són variables globals. S'hi inicialitzen a zero, el **punter nul** (*null pointer*) de C, per explicitar que no apunten a cap adreça vàlida, per exemple, perquè encara no es coneix l'adreça de les variables a les quals apuntaran.

Exemple 2.16: Declaració de punters en RV32I

RV32I

```
.data
p1:     .word   0     # Punter a enter (inicialitzat a null)
p2:     .word   0     # Punter a caràcter (inicialitzat a null)
p3:     .word   0     # Punter a short (inicialitzat a null)
```

2.7.3 C — `&`: l'operador «adreça de»

En **C**, l'**operador** `&`, que es llegeix **et** (*ampersand*), davant d'una variable o d'un element d'un vector, és l'**operador** «**adreça de la variable**» (o de l'element) que precedeix. Així doncs, `&x` representa l'«adreça de la variable» `x`.

> **Essencial 2.27**
>
> No s'ha de confondre l'ús de `&` en punters, on és un operador **unari**, amb l'ús en lògica bit a bit, on és l'operador **binari** AND (Secció 3.2.1).

2.7.4 Assignació i inicialització en la declaració

En **C**, a un **punter** se li pot **assignar un valor** (una adreça) principalment de tres maneres:

- Mitjançant l'operador `&` aplicat a una variable del tipus apuntat:

C

```c
ptr = &var;
```

- Assignant-li un altre punter del mateix tipus:

C

```c
p1 = p2;                    // p2 ja està inicialitzat
```

- Explícitament, mitjançant una **conversió de tipus** (*type cast*) a partir d'una adreça numèrica (ús avançat):

C

```c
ptr = (int *) <adreça>;     // per exemple, 0x10010000
```

En **C**, un **punter** es pot **inicialitzar en la declaració** amb la part `[= adreça_d_inicialització]` de la sintaxi de la declaració (`<tipus_apuntat> *<nom_del_punter> [= adreça_d_inicialització];`, Secció 2.7.2). L'adreça d'inicialització pot ser l'adreça d'una variable (`&x`) o un altre punter del mateix tipus; però una variable **global** només es pot inicialitzar amb una **expressió constant**, de manera que `int *p2 = p1;` només és vàlid en una variable local.

> **Aprofundiment 2.14: C — Declarador, inicialitzador i delimitador d'inicialització**
>
> En **C**, una declaració com `int *p1 = &x;` té tres parts:
>
> - El **tipus base** (`int`): el tipus de la dada apuntada.
> - El **declarador** (`*p1`): combinat amb el tipus base, indica que `p1` és un punter a `int`. El símbol `*` forma part del declarador, no del **tipus base**.
> - L'**inicialitzador** (`&x`): l'expressió que assigna el valor inicial al punter.
>
> El caràcter `=` és el **delimitador d'inicialització**, que separa el declarador de l'inicialitzador.

En **RV32I**, l'**assignació** d'un punter es pot fer:

- Assignant-li l'adreça d'una variable (per exemple, mitjançant la pseudoinstrucció `la` en cas d'una variable global).

- Explícitament (per exemple mitjançant la pseudoinstrucció `li`).

La **inicialització en la declaració de variables globals** es pot fer mitjançant etiquetes de dades, com es mostra a l'Exemple 2.17. En aquest exemple, `p1` s'inicialitza amb l'adreça de la variable global `x`, mentre que `p2` s'inicialitza a zero (punter nul). A la funció `main`, s'assigna `p1` a `p2`, de manera que `p2` també apunta a `x`.

Essencial 2.28

A partir d'aquí, els programes en C d'aquests apunts s'escriuen amb `void main()`: són **notació** del programa que es tradueix a mà a assemblador, no codi que es compili, i no hi ha cap SO que en reculli el valor de retorn (EC 1.2). **En un programa de C normal, compilat per a un sistema operatiu, cal `int main`.**

Exemple 2.17: Inicialització de punters a variables globals

C

```c
int x = 42;         // Variable entera
int *p1 = &x;       // p1 apunta a x (inicialització per assignació d'adreça)
int *p2;            // Punter global sense inicialitzador (val NULL)

void main() {
    p2 = p1;        // p2 apunta a x (assignació del punter p1)
}
```

RV32I

```
.data
x:      .word   42              # x <- 42
p1:     .word   x               # Punter a enter (inicialitzat a &x)
p2:     .word   0               # Punter a enter (inicialitzat a null)

.text
        la      t0, p1          # t0 <- adreça de p1
        lw      t1, 0(t0)       # t1 <- p1 (el valor del punter p1, és a dir, &x)
        la      t2, p2          # t2 <- adreça de p2
        sw      t1, 0(t2)       # p2 = p1;
```

2.7.5 Desreferència o indirecció

L'operació «**desreferència** o **indirecció** de punter» és l'operació d'accés (lectura o escriptura) a la posició de memòria apuntada pel punter.

En **C**, s'expressa precedint el nom del punter amb el caràcter `*` en les **sentències**.

Així doncs, l'expressió `*p` significa «**contingut de la posició de memòria** apuntada pel punter `p`».

Essencial 2.29

No s'ha de confondre l'ús de `*` com a operador en les **sentències** amb l'ús en les **declaracions**, on forma part del declarador (`*<nom_del_punter>`, Secció 2.7.2). En C només té sentit parlar

d'operadors en expressions, no en declaracions.

Exemple 2.18: Operació de desreferència de punters

C

```c
char *pglob;     // variable global: exemple il·lustratiu; en ús
                 // real cal inicialitzar abans de desreferenciar

void g()
{
        char tmp, *ploc;    // es guarden a t0, t1
        ...
        tmp = *ploc;
        ...
        tmp = *pglob;
        ...
}
```

RV32I

```
.data
pglob:  .word   0           # Punter a caràcter (inicialitzat a null)

.text
g:
        ...
        lb      t0, 0(t1)   # desreferencia el punter ploc
        ...
        la      t2, pglob
        lw      t3, 0(t2)   # llegeix el punter en un registre
        lb      t0, 0(t3)   # desreferencia el punter pglob
        ...
```

La primera sentència llegeix el contingut de la posició de memòria apuntada per `ploc` i el copia a la variable local `tmp`. Com que `ploc` està en un registre, n'hi ha prou amb executar una instrucció `lb` on el registre-base sigui el que conté el punter. La segona sentència és similar, però com que `pglob` és global, està emmagatzemada en memòria. Cal llegir prèviament la variable global `pglob` al registre `t3` (es carrega l'adreça de `pglob`, la de l'etiqueta, a `t2` amb `la`, i a continuació s'accedeix al punter en memòria amb `lw`). Un cop llegit el punter `pglob` en un registre, es fa la desreferència de la mateixa manera que en el cas de `ploc`, amb `lb`.

Aprofundiment 2.15: Punters a funcions

Les funcions també resideixen en memòria (al segment `.text`). Un **punter a funció** és una variable escalar que conté l'**adreça de la primera instrucció** d'una funció.
En **C** es declara indicant el tipus de retorn i els tipus dels paràmetres; per exemple, `int (*op)(int, int);` declara `op` com a punter a una funció que rep dos `int` i en retorna un. Els punters a funció permeten decidir en temps d'execució quina funció s'invoca (taules de salts, **crides de retorn** (*callbacks*), etc.).
En **RV32I**, invocar un punter a funció és una **crida a subrutina indirecta**: es carrega l'adreça en un registre i s'hi salta amb `jalr` (Secció 3.9). A **EC** no es treballen explícitament.

2.7.6 Aritmètica de punters

En **C**, quan es suma un enter `i` a un punter `p` que apunta a un tipus de mida T bytes (els vectors es presenten a Secció 2.8, i `int *p = v`, a Secció 2.8.4), en resulta un punter que apunta $i \cdot T$ bytes més endavant de l'adreça apuntada per `p`. De manera simètrica, restar `i` desplaça el punter $i \cdot T$ bytes enrere d'aquesta adreça. Aquesta regla d'escalat permet navegar per elements d'un vector independentment del tipus de dades que contingui.

Essencial 2.30

En C, es poden aplicar operacions aritmètiques als punters, però el resultat sempre ha de ser una adreça de memòria vàlida, cosa que limita les operacions permeses: per exemple, no es poden multiplicar punters (Aprofundiment 2.16).

Aprofundiment 2.16: C — Aritmètica de punters: operacions permeses

L'estàndard de C és molt restrictiu per evitar accessos a memòria sense sentit. Les úniques operacions permeses amb punters són:

- Suma d'un punter i un enter: `p + n` (desplaça el punter `n` posicions del tipus de dades apuntat).
- Resta d'un punter i un enter: `p - n`.
- Resta de dos punters: `p1 - p2` (dona el nombre d'elements entre ells, de tipus `ptrdiff_t`).
- Increment i decrement: `p++` i `p--`.
- Comparacions: `p1 < p2`, `p1 == p2`, etc.

La multiplicació, per exemple, no és permesa.

En **RV32I**, l'aritmètica de punters l'ha de gestionar el programador. Per exemple, si `p` és un punter a `short` i és a `t0`, la sentència `p = p + 1;` es tradueix a `addi t0, t0, 2`, ja que un `short` ocupa 2 bytes. De manera simètrica, `p = p - 2;` es tradueix a `addi t0, t0, -4`.

Exemple 2.19: Aritmètica de punters

C

```c
int v[10];
int *p = v;      // p apunta al primer element de v
p = p + 1;       // ara p apunta a v[1] (incrementa l'adreça en 4 bytes)
p = p - 2;       // ara p apunta a v[-1]: fora de rang! (decrementa l'adreça en 8 bytes)
```

RV32I

```
.data
        .align  2
v:      .space  40              # int v[10]
p:      .word   0               # Punter a enter

.text
        la      t1, p           # t1 <- adreça del punter p
        la      t0, v           # t0 <- adreça de v[0]
        sw      t0, 0(t1)       # p = v  (p apunta a v[0])
```

```
lw      t0, 0(t1)       # t0 <- valor de p
addi    t0, t0, 4       # p + 1 (avança 4 bytes: mida d'un int)
sw      t0, 0(t1)       # p = p + 1  (ara p apunta a v[1])

lw      t0, 0(t1)       # t0 <- valor de p
addi    t0, t0, -8      # p - 2 (retrocedeix 8 bytes: 2 * mida d'un int)
sw      t0, 0(t1)       # p = p - 2  (ara p apunta a v[-1]: fora de rang!)
```

(Per a la directiva `.space`, vegeu RISC-V 2.31.)

2.7.7 Absència de control en l'accés a memòria

L'**estàndard** de C **no especifica** què passa si s'accedeix a una àrea de memòria il · legítima. Es produeix el que es coneix com a **comportament indefinit** (*undefined behavior*). En conseqüència, l'estàndard no garanteix ni cap error específic ni cap resultat concret. A efectes pràctics, depenent del context i de la memòria, pot ocórrer una d'aquestes tres situacions, per ordre creixent de **perillositat**:

1. **Error de segmentació** (***segmentation fault***), o *segfault*: És l'error més comú en temps d'execució. L'SO detecta que el programa intenta llegir o escriure en una adreça de memòria que no li pertany (una zona protegida o no assignada) i finalitza el procés immediatament.
2. **Corrupció (silenciosa) de dades**: Si l'adreça calculada amb l'aritmètica de punters encara cau dins de la memòria que el programa té assignada (per exemple, on hi ha una altra variable veïna al segment de dades, Secció 2.5), **el programa continuarà funcionant**, però **accedint a dades indeterminades** sense cap indicació d'error.
3. **Vulnerabilitats de seguretat**: Si l'accés és a la pila (*stack*, Secció 3.9.4), es pot produir un **desbordament de memòria intermèdia** (***buffer overflow***), fet que podria arribar a permetre a un **atacant** modificar l'adreça de retorn d'una funció i **executar codi maliciós**.

En **RV32I** tampoc no es comproven els límits: el programador és responsable que els punters apuntin a adreces vàlides. Un accés fora d'un vector però dins d'un segment accessible s'executa sense cap avís; només es produeix una excepció si l'adreça no és alineada (Secció 2.6.3.4) o, a **RARS**, si cau fora de la memòria del programa (per exemple, en desreferenciar el punter nul).

2.8 Vectors

Les **variables estructurades** són **agrupacions** d'elements. Aquests elements poden ser variables escalars o estructurades.

Els **vectors** (***arrays***) són **variables estructurades unidimensionals**, on tots els elements són del **mateix** tipus i s'identifiquen per un **índex**, i.

2.8.1 Declaració i inicialització

En **C**, els elements dels **vectors** només poden ser del mateix tipus, el qual s'ha d'especificar en el moment de la declaració.

En **C**, la **declaració** de **vectors** es pot fer de tres maneres:

- Especificant el nombre d'elements sense inicialització:

– C

```
<tipus_elements> nom_vector[<quantitat_elements>];
```

- Inicialitzant-ne els elements sense explicitar-ne el nombre:

– C

```
<tipus_elements> nom_vector[] = {valor_element_1, valor_element_2, ...};
```

- Explicitant-ne el nombre d'elements i inicialitzant-los (combinació de les dues anteriors):

– C

```
<tipus_elements> nom_vector[<quantitat_elements>] = {valor_element_1, valor_element_2,
↪  ...};
```

Exemple 2.20: Declaració i inicialització de vectors i matrius

En **C**, es poden declarar i inicialitzar vectors i matrius de la manera següent:

C

```
short vec[] = {2, -1, 3, 5, 0};                 // vector
int mat[2][3] = {{11, 12, 13}, {21, 22, 23}};  // matriu [files][columnes]
```

A **RARS**, es poden inicialitzar diversos valors separats per comes (també admet espais, però GNU `as` no).

RV32I

```
.data
vec:    .half   2, -1, 3, 5, 0
mat:    .word   11, 12, 13, 21, 22, 23
```

Per a matrius, l'opció del Codi 2.5 permet visualitzar millor la seva estructura bidimensional:

Codi 2.5: La matriu `mat` declarada fila a fila.

RV32I

```
mat:    .word   11, 12, 13
        .word   21, 22, 23
```

Per a vectors llargs no inicialitzats (valor **zero** per defecte, tant a **RARS** com en **C** per a les variables globals; en **C**, només les variables locals sense inicialitzar tenen un valor **indeterminat**), s'usa la directiva `.space n`, que reserva `n` **bytes**. Per assegurar l'alineació que correspon al tipus del vector, cal precedir `.space` de la directiva `.align k`, que alinea l'adreça següent a un múltiple de 2^k.

RISC-V 2.31: Directives — Reserva de memòria: `.space`, `.align`

Directiva	Operands	Descripció
`.align`	`k`	Alinea l'element següent a la frontera de 2^k bytes
`.space`	`n`	Reserva `n` bytes al segment de dades

Exemple 2.21: Declaració d'un vector assegurant l'alineació correcta

C

```c
int v[100];
```

RV32I

```
.data
        .align 2        # Força adreça múltiple de 4 (2^2)
v:      .space  400     # Reserva 400 bytes (100 enters * 4 bytes)
```

2.8.2 C — Tipus `<tipus_elements> nom_vector[<quantitat_elements>]`

En **C**, `int vec[<quantitat_elements>]` no és només una declaració: constitueix un **tipus complet** (`int[<quantitat_elements>]`).

Per exemple, `int[10]` és diferent d'`int[5]` i d'`int *`.

Aquest fet té conseqüències pràctiques, tant en vectors (Exemple 2.22) com en matrius (Exemple 2.24, Exemple 2.25).

Exemple 2.22: Els tipus dels vectors

C

```c
int vec1[100];  // Tipus: "vector de 100 enters"
int vec2[5];    // Tipus: "vector de 5 enters"
```

Exemple 2.23: Els vectors no són assignables

A diferència dels punters, en **C** els vectors **no són assignables**, ni tan sols si són del mateix tipus i mida. Ho il · lustra el Codi 2.6:

Codi 2.6: Dues assignacions de vectors, que el compilador rebutja.
⚠`codi_erroni__vectors.c`⚠

```c
int a[5];
int b[5];
int c[10];
```

```
b = a;  // ⚠ ERROR: no es poden assignar vectors directament ⚠
c = a;  // ⚠ ERROR: + tipus incompatibles (int[5] vs. int[10]) ⚠
```

Exemple 2.24: Les matrius com a vectors de vectors

En **C**, una matriu `int mat[2][3]` és un **vector de 2 elements**, on cada element és de tipus `int[3]` (un vector de 3 enters).

C

```c
int mat[2][3] = {{1, 2, 3}, {4, 5, 60}};

int x = mat[1][2];  // equivalent a *(*(mat + 1) + 2) => x <- 60
```

L'emmagatzemament a memòria és **per files**, de dalt a baix (primer tots els elements de la fila 0, després els de la fila 1, etc.) (Secció 4.5.1).

En **C**, `sizeof(element)` és un operador unari que retorna la mida d'`element`, que pot ser un escalar o una dada estructurada.

Exemple 2.25: `sizeof()`

Taula 2.4: Exemples de retorn de `sizeof()` a RV32I

Codi	Retorna
`sizeof(int)`	La mida d'un enter (4 bytes a RV32I)
`sizeof(int[10])`	La mida d'un vector de 10 enters (40 bytes a RV32I)
`sizeof(int[2][3])`	La mida d'una matriu de 2×3 enters (24 bytes a RV32I)

2.8.3 Indexació

L'accés a un element d'un vector es fa mitjançant l'**índex** de l'element.

En **C**, l'índex comença a **0** i, en sentències, els símbols `[` i `]` precedits del nom d'un vector i envoltant una expressió entera són l'operador d'**indexació** (`[]`).

En C és el compilador qui s'encarrega de calcular l'adreça de l'element `vec[i]` multiplicant automàticament l'índex `i` per la mida de cada element, `sizeof(element)`, per obtenir el desplaçament real en bytes. Per tant, rarament cal preocupar-se pel desplaçament; n'hi ha prou amb proporcionar l'índex.

En canvi, en **RV32I** cal gestionar manualment l'accés a un element d'un vector mitjançant les instruccions de **lectura** i **escriptura**. En aquestes instruccions, el registre base conté l'adreça base i l'immediat el desplaçament, tots dos en bytes. Per tant, cal calcular el desplaçament com el producte de l'índex per la mida del tipus dels elements:

$$@\texttt{vec[i]} = @\texttt{vec[0]} + i \cdot T \tag{2.1}$$

on T és la mida en bytes de cada element ($T =$ `sizeof(tipus_element)`).

Exemple 2.26: Accés a un element d'un vector

Sigui `vec` un vector d'enters (mida de cada element = 4 bytes) i `h` una variable entera. L'operació d'assignar a `vec[12]` el resultat de sumar `h` i `vec[8]` es pot expressar en C com:

C

```c
vec[12] = h + vec[8];
```

En **RV32I**, suposant que `h` ocupa el registre `s2`, i que `vec` és una variable global, l'adreça de `vec[8]` és la suma de l'adreça base de `vec` (@vec) i el desplaçament $8 \cdot 4 = 32$. I l'adreça de `vec[12]` és la suma de @vec i el desplaçament de $12 \cdot 4 = 48$:

RV32I

```
.text
        la      t3, vec         # t3 <- Adreça base del
        ↪ vector
        lw      t0, 32(t3)      # t0 <- vec[8]
        add     t0, s2, t0      # t0 <- h + vec[8]
        sw      t0, 48(t3)      # vec[12] <- t0
```

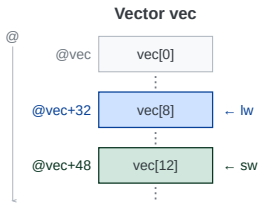


Figura 2.5: Accés a un element d'un vector.

Essencial 2.31

Com que el compilador de C no comprova si un accés es fa a una adreça il · legítima (Secció 2.7.7), no comprova si l'índex està fora del rang del vector. Simplement suma l'índex a l'adreça i accedeix a la memòria resultant, encara que sigui fora de l'àrea de memòria del vector.

2.8.4 C — Relació entre punters i vectors

En **C**, en la majoria d'expressions, el nom d'un vector **decau** automàticament a un **punter al seu primer element** (del tipus `<tipus_element> *`), excepte en dos contextos específics: quan és operand de l'operador `&` i quan és operand de `sizeof`. Aquest fenomen es coneix com a **decaïment de vectors a punters**.

Una **conseqüència** del decaïment és l'**equivalència** entre `vec` i `&vec[0]`.

Essencial 2.32

Per tant, en **C**, l'**adreça base** del vector `vec` es pot obtenir de dues maneres (equivalents) següents:

- `&vec[0]`.
- `vec`.

Exemple 2.27: Decaïment de vectors a punters

C

```c
int vec[100];
int *p = vec;
```

Anàlisi:

- Les variables `vec` i `p` tenen tipus diferents:
  - `vec` representa tot el bloc de memòria reservat.
  - `p` és una variable que pot emmagatzemar una adreça de memòria.
- Mida:
  - `sizeof(vec)` retorna 400 (100 enters × 4 bytes).
  - `sizeof(p)` retorna la mida d'una adreça (4 bytes a RV32I).
- Assignació:
  - `p = vec;` és una expressió vàlida perquè `vec` decau a un punter al seu primer element. `&vec[0]` també seria vàlid.
  - ~~`vec = p;`~~ no és una expressió vàlida: el nom d'un vector no es pot assignar (Exemple 2.23).
  - `p` és una variable, per tant, pot canviar en el temps.

Exemple 2.28: Decaïment d'una fila d'una matriu a punter

C

```c
int mat[2][3] = {{1, 2, 3},
                 {4, 5, 6}};

int *p = mat[0];    // p apunta al primer element de la fila 0
```

`mat[0]` és de tipus `int[3]` (el vector de la primera fila de `mat`). En l'assignació a `p`, decau a `int *`, que apunta al seu primer element, `mat[0][0]`.

Tècnicament, l'operador d'indexació (`[]`) és una **notació abreujada** per a l'aritmètica de punters: l'estàndard de C defineix `vec[i]` com a exactament equivalent a `*(vec + i)`.

Per tant, l'operador `[]` forma part de l'**aritmètica de punters** perquè el seu funcionament intern depèn de les mateixes regles d'escalat presentades a Secció 2.8.3.

Essencial 2.33: `[]`, l'operador d'indirecció implícit

En **C**, en sentències, l'operador `[]`, precedit del nom d'una variable de tipus vector, fa, per aquest ordre:

1. L'**aritmètica de punters**: suma l'índex, escalat per la mida de l'element, a l'adreça base (Equació 2.1).
2. La **desreferència** (accedeix a l'element apuntat; lectura o escriptura, segons correspongui).

Per tant, és una forma compacta de realitzar aritmètica de punters i desreferència simultàniament.

> ⚠ Aprofundiment 2.17: Commutativitat de l'operador `[]`
>
> Com que la suma és commutativa (`a + i` és el mateix que `i + a`), en C és perfectament vàlid escriure `3[vec]` en lloc de `vec[3]`. El compilador ho tradueix igualment a `*(3 + vec)`.

2.8.5 Accés a vectors amb bucles

Hi ha dues maneres bàsiques de recórrer els elements d'un vector amb un bucle:

- L'**accés aleatori** (Secció 2.8.5.2; per a matrius, Secció 4.5.2), que consisteix a calcular, a cada iteració, l'adreça de cada accés (Equació 2.1, Equació 4.3).
- L'**accés seqüencial** (Secció 2.8.5.3; optimització definida a Secció 4.6.1), que manté un punter que avança una quantitat fixa (*stride*) a cada iteració.

2.8.5.1 Extracció d'invariants

Un **invariant de bucle** és qualsevol valor que no canvia d'una iteració a una altra (no depèn de cap variable que el bucle modifiqui) i que, per tant, es pot calcular una sola vegada abans d'entrar al bucle. Calcular-lo dins el bucle és un error de rendiment perquè el processador el recalcularia innecessàriament a cada iteració.

> ! EC 2.19
>
> A **EC**, cal fer sempre l'**extracció d'invariants** abans d'entrar a qualsevol bucle, llevat que s'indiqui explícitament el contrari. És un prerequisit implícit de totes les optimitzacions de bucle que es veuran a la Secció 4.6.

2.8.5.2 Accés aleatori

> 🔥 Essencial 2.34
>
> L'accés aleatori **no implica atzar**[5], sinó la capacitat d'accedir directament a qualsevol element d'un vector o matriu (vegeu Secció 4.5.2) en temps constant $\mathcal{O}(1)$, independentment de la seva posició.

> 🔥 Essencial 2.35
>
> Els exemples de bucle d'aquesta secció usen instruccions de salt condicional (`beq`) i incondicional (`j`), i de desplaçament de bits (`slli`), que es presenten formalment al Tema 3 (Secció 3.1, Secció 3.4). De moment n'hi ha prou amb entendre-les intuïtivament pel comentari: `beq rs1, rs2, etq` salta a `etq` si `rs1 = rs2`; `j etq` salta sempre a `etq`; `slli rd, rs1, n` desplaça `rs1` `n` bits cap a l'esquerra (equivalent a multiplicar per 2^n).

[5] És el mateix cas d'ús que a la *random access memory* (RAM), que no significa memòria a l'atzar, sinó memòria que permet accés directe a qualsevol adreça.

Exemple 2.29: Recorregut d'un vector per accés aleatori

C

```c
#define N 4

int V[N] = {0, 1, 2, 3};

void main() {
    int suma = 0, i = N;
    while (i != 0) {
        i--;
        suma = suma + V[i];
    }
}
```

Suposem que `suma` i `i` ocupen els registres `t0` i `t1`:

RV32I

```
        .eqv    N, 4                # define N 4

.data
V:      .word   0, 1, 2, 3          # vector d'enters

.text
        mv      t0, zero            # suma <- 0
        li      t1, N
        la      t4, V               # invariant: @V[0], extret del bucle
while:
        beq     t1, zero, fiwhile   # [ctrl]
        addi    t1, t1, -1          # [ctrl] i--
        slli    t3, t1, 2           # [ctrl] i*4
        add     t3, t3, t4          # [ctrl] @V[0] + i*4
        lw      t5, 0(t3)           #        V[i]
        add     t0, t0, t5          #        suma <- suma + V[i]
        j       while               # [ctrl]
fiwhile:
```

L'adreça base `@V[0]` és un invariant i s'ha extret del bucle (`la t4, V`). La infraestructura de control del bucle ocupa **5 instruccions** per iteració (les marcades `[ctrl]`). A la Secció 4.6 es reduirà fins a **2** aplicant les tres optimitzacions, cada una de les quals comportarà l'estalvi d'una instrucció.

2.8.5.3 Accés seqüencial

Exemple 2.30: Recorregut d'un vector per accés seqüencial (optimització #1)

El Codi 2.7 fa servir un punter (`p`) per recórrer el vector i accedeix als seus elements desreferenciant-lo.

Codi 2.7: Suma dels elements de `V` amb un punter.

C

```c
#define N 4
```

```c
int V[N] = {0, 1, 2, 3};

void main() {
    int suma = 0, i = N;
    int *p = V;                 // equival a: p = &V[0]
    while (i != 0) {
        suma = suma + (*p);     // desreferència de p
        p++;                    // p apunta a l'element següent
        i--;
    }
}
```

Suposem que `suma`, `i` i `p` ocupen `t0`, `t1` i `t3`:

RV32I

```
        .eqv    N, 4                # define N 4

.data
V:      .word   0, 1, 2, 3          # vector d'enters

.text
        mv      t0, zero            # suma <- 0
        li      t1, N
        la      t3, V               # p <- @V[0]
while:
        beq     t1, zero, fiwhile   # [ctrl]
        lw      t4, 0(t3)           #       *p
        add     t0, t0, t4          #       suma <- suma + (*p)
        addi    t3, t3, 4           # [ctrl] p++ (stride = 4)
        addi    t1, t1, -1          # [ctrl] i--
        j       while               # [ctrl]
fiwhile:
```

Aplicant l'**accés seqüencial** (optimització #1, Secció 4.6.1), la infraestructura de control ha quedat en **4 instruccions** (les marcades `[ctrl]`). A la Secció 4.6 es reduirà fins a **2** avaluant la condició al final (optimització #2) i eliminant la variable d'inducció `i` (optimització #3).

2.9 Caràcters i cadenes de caràcters

2.9.1 Codificació de caràcters

Els **caràcters** es representen mitjançant codis numèrics. El sistema de codificació més utilitzat ha estat el **codi ASCII**. En ASCII, els caràcters imprimibles (com les lletres, dígits i símbols) es representen amb números entre 32 i 126 (`0x20`–`0x7E`), mentre que els caràcters de control (com el salt de línia) es representen amb números entre 0 i 31 (`0x00`–`0x1F`) i el 127 (`0x7F`, DEL). Els caràcters s'emmagatzemen usant 1 byte (8 bits), però el bit de més pes val sempre zero. La Taula 2.5 mostra els rangs de codis ASCII.

Taula 2.5: Rangs de codis ASCII

Hex	Caràcter
`0x00`–`0x1F`	Caràcters de control (no imprimibles)
`0x20`	(espai)
`0x30`–`0x39`	Dígits `0`–`9`
`0x41`–`0x5A`	Lletres majúscules `A`–`Z`
`0x61`–`0x7A`	Lletres minúscules `a`–`z`
`0x7F`	DEL (caràcter de control)

2.9.2 Representació de caràcters en C i RV32I

En C i RV32I, un literal de tipus caràcter imprimible s'escriu usant el caràcter (entre cometes simples), el seu codi ASCII en decimal, en hexadecimal precedit per `0x` o octal precedit per `0`. Per exemple, la A majúscula es pot escriure: `'A'`, `65`, `0x41` o `0101`. Els caràcters de control s'escriuen mitjançant **seqüències d'escapada** (*escape sequences*), que comencen amb una barra invertida (\) seguida d'un caràcter que indica el tipus de control. Per exemple, el salt de línia es pot escriure com `'\n'` o `0x0A`.

Exemple 2.31: Seqüències d'escapada per a caràcters de control i caràcters especials

Seqüència d'escapada	Hex	Significat en anglès	Significat
`'\n'`	`0x0A`	*Newline*	Línia nova
`'\t'`	`0x09`	*Tab*	Tabulació
`'\r'`	`0x0D`	*Carriage return*	Retorn de carro
`'\0'`	`0x00`	*Null character*	Caràcter nul
`'\\'`	`0x5C`	*Literal backslash*	Barra invertida literal
`'\''`	`0x27`	*Literal single quote*	Cometa simple literal

2.9.3 Declaració i inicialització de caràcters

Exemple 2.32: Declaració i inicialització de caràcters en C i inicialització en memòria en RV32I

C

```c
char c1 = 'A';          // Caràcter A (codi ASCII 65)
char c2 = '\n';         // Salt de línia (codi ASCII 10)
char c3 = '\\';         // Barra invertida literal (codi ASCII 92)
char alt1 = 65;         // Caràcter A (codi ASCII 65)
char alt2 = 0x41;       // Caràcter A (codi ASCII 65)
char alt3 = 0101;       // Caràcter A (codi ASCII 65)
```

RV32I

```
.data
c1:     .byte   0x41    # Caràcter 'A' (codi ASCII 65)
c2:     .byte   0x0A    # Salt de línia (codi ASCII 10)
c3:     .byte   0x5C    # Barra invertida literal (codi ASCII 92)
```

```
alt1:   .byte   65      # Caràcter 'A' (codi ASCII 65)
alt2:   .byte   0x41    # Caràcter 'A' (codi ASCII 65)
alt3:   .byte   0101    # Caràcter 'A' (codi ASCII 65; 0101 en octal)
```

Aprofundiment 2.18: Estàndard Unicode

L'estàndard Unicode és un sistema de codificació de caràcters que assigna un número únic a cada caràcter de tots els sistemes d'escriptura del món, així com a símbols, emojis i altres elements gràfics. Unicode utilitza diversos formats de codificació, el més comú dels quals és **UTF-8**. Aquest estàndard permet representar una àmplia varietat de caràcters més enllà dels 128 caràcters ASCII, incloent-hi els caràcters especials i les lletres amb accents utilitzats en molts idiomes, com el català.
En UTF-8, els caràcters ASCII es representen amb un sol byte (igual que en ASCII), però els caràcters no ASCII es representen amb múltiples bytes. Per exemple, els caràcters de l'alfabet català com *ç*, *ü* o *à* es representen amb dos bytes en UTF-8. En RV32I, les cadenes de caràcters es poden emmagatzemar en memòria seguint la codificació UTF-8, però cal tenir en compte que els caràcters no ASCII ocuparan més d'un byte, i que el final de la cadena encara ha de ser indicat amb un byte nul (`0x00`).

2.9.4 Cadenes de caràcters

Una **cadena de caràcters** (*string*) és una seqüència de caràcters.

En **C**, les cadenes de caràcters es representen com a vectors de caràcters (tipus `char[<quantitat_caracters + 1>]`, pel sentinella) i sempre acaben amb un caràcter nul (`'\0'`, el sentinella), que indica el final de la cadena. El nom de la cadena decau a un punter al seu primer element (el primer caràcter de la cadena).

En **RV32I**, les cadenes de caràcters s'emmagatzemen a memòria com a seqüències de bytes, seguint la codificació ASCII, i també han d'acabar amb un byte nul (`0x00`) per indicar el final de la cadena.

A l'assemblador (GNU `as` i RARS), la directiva `.asciz` (*ASCII zero-terminated*) s'utilitza per definir cadenes de caràcters que acaben amb un byte nul, mentre que la directiva `.ascii` defineix cadenes de caràcters sense el byte nul al final. Per tant, només les seqüències acabades amb un byte nul (`.asciz`, `.string` o `.ascii` seguida de `.byte 0`) són cadenes de caràcters en el sentit de C. `.string` és una directiva equivalent a `.asciz`.

RISC-V 2.32: Directives — Cadenes de caràcters

Directiva	Operands	Descripció
`.ascii`	`"cadena"`	Emmagatzema la cadena al segment de dades sense terminador nul
`.asciz`	`"cadena"`	Emmagatzema la cadena al segment de dades amb terminador nul
`.string`	`"cadena"`	Àlies de `.asciz`

Exemple 2.33: Declaració i inicialització de cadenes de caràcters en C i RV32I

C

```c
char str1[] = "Hola";    // Seqüència de caràcters seguida d'un caràcter nul: és una cadena
↪  (sizeof(str1) == 5)
char str2[] = {'M', 'e', 'u'};     // Seqüència de caràcters sense un caràcter nul al final
↪  (sizeof(str2) == 3)
char str3[] = {'M', 'a', 'p', 'a', '\0'}; // Seqüència de caràcters "Mapa" seguida d'un caràcter
↪  nul: és una cadena de caràcters
```

RV32I

```
.data
str1:   .asciz  "Hola"  # Seqüència de caràcters "Hola" seguida d'un byte nul: és una cadena de
↪  caràcters
str2:   .ascii  "Meu"   # Seqüència de caràcters "Meu" no seguida d'un byte nul: NO és una
↪  cadena de caràcters
str3:   .ascii  "Mapa"  # Seqüència de caràcters "Mapa" sense byte nul
        .byte   0x00    # Byte nul que permet interpretar la seqüència str3 com una cadena de
        ↪  caràcters
```

Tema 3: Traducció de programes

EC 3.1

A **EC**, les traduccions de C a assemblador han de ser **el més literals possible**, si no s'indica explícitament el contrari: cada sentència es tradueix en l'ordre del programa, **sense les transformacions d'un compilador optimitzador** (reordenar, fusionar o eliminar sentències, o les optimitzacions de bucles de Secció 4.6). Literal no vol dir ineficient: dins de cada sentència, cal fer servir els registres i les instruccions amb criteri.

3.1 Desplaçaments de bits

3.1.1 Desplaçaments lògics

Les instruccions `sll` i `srl` desplacen el contingut del registre `rs1` a l'esquerra o a la dreta el nombre de posicions especificat als **5 bits** de menys pes del **registre** `rs2`, omplint les posicions vacants amb **zeros**. Les variants `slli` i `srli` fan la mateixa operació, però el nombre de posicions es codifica directament com un **immediat** de **5 bits** (`shamt`, *shift amount*)[1]. En tots els casos, el nombre de posicions s'interpreta com un natural.

RISC-V 3.1: RV32I ISA — Instruccions de desplaçament lògic

Mnemònic	Operands	Operació	Nom	Tipus
`sll`	`rd, rs1, rs2`	$rd \leftarrow rs1 << rs2[4:0]$	*shift left logical*	R
`slli`	`rd, rs1, shamt`	$rd \leftarrow rs1 << shamt$	*shift left logical*	I
`srl`	`rd, rs1, rs2`	$rd \leftarrow rs1 >> rs2[4:0]$	*shift right logical*	R
`srli`	`rd, rs1, shamt`	$rd \leftarrow rs1 >> shamt$	*shift right logical*	I

Les posicions vacants s'omplen amb `0`.

Exemple 3.1: Desplaçaments lògics

RV32I

```
        li      t0, 0x88888888
        slli    t0, t0, 1       # t0 = 0x11111110
```

RV32I

```
        li      t0, 0x99999999
```

[1] Els 7 bits de més pes de l'immediat de 12 bits (`imm[11:5]`) no formen part del nombre de posicions: fan de `funct7` i distingeixen `srli` (`0000000`) de `srai` (`0100000`).

```
srli    t0, t0, 2       # t0 = 0x26666666
```

Aprofundiment 3.1: Conversió de Ca2 a Ca1 amb instruccions de desplaçament

Donada la representació en Ca2 d'un enter x_s, guardada a `t2`,
Pregunta: Com s'obté la seva representació en Ca1 (Secció 1.5.4) i s'assigna a `t1`?
Solució:
Si x_u i x'_u són els valors de les representacions en Ca2 i en Ca1 llegides com a naturals, les fórmules que les relacionen amb l'enter representat x_s són (la de Ca2 és la de Essencial 1.2; la de Ca1 se'n dedueix de la mateixa manera, amb $2^n - 1$ en lloc de 2^n):

- **Ca2**: $x_s = x_u - X_{n-1} \cdot 2^n$.
- **Ca1**: $x_s = x'_u - X_{n-1} \cdot (2^n - 1)$.

on X_{n-1} és el bit de signe (el mateix en els dos casos).
Igualant les dues expressions i simplificant:

$$x'_u = x_u - X_{n-1}$$

És a dir: la representació en Ca1 s'obté restant el bit de signe a la representació en Ca2. Per extreure el bit de signe de `t2` i fer-ne la resta:

`RV32I`

```
srli    t0, t2, 31      # desplaça el bit de signe a la posició 0
sub     t1, t2, t0      # resta del bit de signe a la representació Ca2
```

Anàlisi

- Si $x_s \geq 0$: el bit de signe és `0`, de manera que `t0 = 0` i `t1 = t2`. La representació no canvia, cosa que té sentit perquè els positius es codifiquen igual en Ca1 i Ca2.
- Si $x_s < 0$: el bit de signe és `1`, de manera que `t0 = 1` i `t1 = t2 - 1`. Efectivament, la representació en Ca1 d'un negatiu és sempre 1 menys que la representació en Ca2 del mateix enter (Secció 1.5).

Exemple: $x_s = -51$, representat en Ca2 amb 8 bits com `1100 1101`$_2$ (= `0xCD`) i en Ca1 com `1100 1100`$_2$ (= `0xCC`). Fent les extensions a 32 bits, `t2 = 0xFFFFFFCD` i `t1 = 0xFFFFFFCC`.

- `srli t0, t2, 31` → `t0 = 1` (bit de signe)
- `sub t1, t2, t0` → `t1 = 0xFFFFFFCD - 1 = 0xFFFFFFCC`

El mètode val per a tot el rang de Ca1, $[-(2^{n-1} - 1), 2^{n-1} - 1]$; -2^{n-1} no hi té representació, i el resultat seria `0x7FFFFFFF`.

3.1.2 Desplaçament aritmètic a la dreta

La instrucció `sra` desplaça el contingut de `rs1` a la dreta el nombre de posicions indicat, però en lloc d'omplir les posicions vacants amb zeros, les omple amb una còpia del **bit de signe** de `rs1`. Igual que `srl`, admet tant la variant d'immediat com la de registre.

EC 3.2: Notació >>> a la taula ISA

A la taula següent s'usa >>> per denotar el **desplaçament aritmètic a la dreta** (amb extensió de signe), seguint la notació de Harris & Harris. Aquesta notació **no coincideix** amb la de l'operador `>>` de C usat més avall (Secció 3.1.3), on lògic i aritmètic es distingeixen pel tipus de l'operand (`unsigned` o amb signe) en lloc d'un símbol diferent. Són dues convencions vàlides, cadascuna pròpia del context on apareix (taula ISA vs. traducció d'operadors C): cal no barrejar-les.

RISC-V 3.2: RV32I ISA — Instrucció de desplaçament aritmètic

Mnemònic	Operands	Operació	Nom	Tipus
`sra`	`rd, rs1, rs2`	$rd \leftarrow rs1 >>> rs2[4:0]$	*shift right arithmetic*	R
`srai`	`rd, rs1, shamt`	$rd \leftarrow rs1 >>> shamt$	*shift right arithmetic*	I

Les posicions vacants s'omplen amb una còpia del bit de signe.

Exemple 3.2: Desplaçament aritmètic

`RV32I`

```
        li      t0, 0x99999999
        srai    t0, t0, 2       # t0 = 0xE6666666
```

El bit de signe de `0x99999999` és `1`, de manera que les dues posicions que queden vacants a l'esquerra s'omplen amb uns.

Essencial 3.1

No existeix la instrucció ~~`sla`~~ ni la ~~`slai`~~.

3.1.3 Traducció dels operadors C << i >>

L'operador `<<` de **C** (desplaçament a l'esquerra) es tradueix sempre amb `sll`/`slli`. L'operador `>>` es tradueix segons el tipus de l'operand esquerre: amb `srl`/`srli` si és un natural (de tipus `unsigned`), o amb `sra`/`srai` si és un enter (amb un enter negatiu, el resultat depèn de la implementació de C; els compiladors habituals, com `gcc`, fan el desplaçament aritmètic).

Exemple 3.3: Traducció de << i >>

`C`

```
a = (a << b) >> 2;   // a i b són enters (int)
```

Suposem que `a` i `b` es guarden als registres `t0` i `t1`:

`RV32I`

```
        sll     t4, t0, t1   # a << b
        srai    t0, t4, 2    # resultat >> 2 (amb extensió de signe)
```

3.1.4 Aplicació d'`sll/slli`: multiplicació per potències de 2

Desplaçar a l'esquerra un enter o natural amb `sll/slli` equival a **multiplicar-lo** per 2^k (k = `shamt` o `rs2[4:0]`), sempre que el resultat sigui representable en 32 bits.

> **!** EC 3.3
>
> A **EC**, el **desplaçament** en bytes d'un element d'un vector o d'una matriu (Equació 2.1) es calcula multiplicant l'índex per la mida de l'element amb `slli` (un desplaçament de bits de $\log_2$ de la mida), no amb `mul`: a EC, les mides dels elements (1, 2, 4 o 8 bytes) són sempre potències de 2. Amb elements d'1 byte, no cal desplaçar.

3.1.5 Aplicació d'`srl/srli` i `sra/srai`: divisió entera per potències de 2

Desplaçar a la dreta naturals amb `srl/srli` o enters amb `sra/srai` equival a **dividir-los** per una potència de 2: `rd` $\leftarrow$ `rs1` / $2^{\texttt{shamt}}$.

> Essencial 3.2
>
> La semàntica de l'operador `/` de C requereix que el residu tingui el **mateix signe** que el dividend. En canvi, `sra/srai` produeix sempre un residu **no negatiu**. Per tant, `sra/srai` **no és equivalent** a `/` quan el **dividend és negatiu**.
> Per exemple, la divisió entera (-15 / 4) dona quocient -3 i residu -3, mentre que el desplaçament `-15 >> 2` dona quocient -4 i residu 1. Només cal usar `sra/srai` per traduir `/` si es pot garantir que el dividend no és negatiu.

3.2 Operacions lògiques bit a bit

3.2.1 Operadors `&`, `|` i `^` (`and`, `or` i `xor`)

Els operadors **bit a bit** `&`, `|` i `^` de **C** es tradueixen amb `and`, `or` i `xor`. Cadascuna té també una variant amb immediat (`andi`, `ori` i `xori`), on l'immediat de 12 bits s'estén a 32 bits amb **extensió de signe**, igual que `addi`, `slti` i `sltiu`.

> Essencial 3.3
>
> Els immediats d'`andi`, d'`ori` i d'`xori` s'interpreten en **complement a 2**: si el bit 11 de l'immediat val `1`, el valor estès a 32 bits té els bits [31:12] a `1`. Això pot produir màscares no desitjades. Per exemple:
>
> `RV32I`
>
> ```
> andi    t0, t1, -2048   # t0 = t1 & 0xFFFFF800  (no t1 & 0x00000800!)
> ```
>
> Per a màscares arbitràries de 32 bits, cal carregar la màscara en un registre amb `li` i usar `and/or/xor`.

RISC-V 3.3: RV32I ISA — Instruccions lògiques bit a bit

Mnemònic	Operands	Operació	Nom	Tipus
`and`	`rd, rs1, rs2`	$rd \leftarrow rs1 \wedge rs2$	*and*	R
`andi`	`rd, rs1, imm12`	$rd \leftarrow rs1 \wedge SignExt(imm12)$	*and immediate*	I
`or`	`rd, rs1, rs2`	$rd \leftarrow rs1 \vee rs2$	*or*	R
`ori`	`rd, rs1, imm12`	$rd \leftarrow rs1 \vee SignExt(imm12)$	*or immediate*	I
`xor`	`rd, rs1, rs2`	$rd \leftarrow rs1 \oplus rs2$	*exclusive or*	R
`xori`	`rd, rs1, imm12`	$rd \leftarrow rs1 \oplus SignExt(imm12)$	*exclusive or immediate*	I

RISC-V 3.4: Pseudoinstruccions — Negació bit a bit: `not`

RV32I no inclou la instrucció `not`. La **negació bit a bit** d'un registre, equivalent al `~` de **C**, s'obté amb la pseudoinstrucció `not`:

Pseudoinstrucció	Operació	Expansió
`not rd, rs`	$rd \leftarrow \overline{rs}$	`xori rd, rs, -1`

`xori rd, rs, -1` funciona perquè `-1` en Ca2 és `0xFFFFFFFF`, i fer XOR amb tots uns complementa tots els bits.

3.2.2 Aplicació d'`and`: selecció de bits (màscara)

La instrucció `and` s'usa per **seleccionar determinats bits** d'un registre posant a zero la resta. La tècnica consisteix a construir una **màscara** on els bits que es volen conservar valen `1` i la resta valen `0`.

Exemple 3.4: Selecció de bits amb màscara `and`

Selecció dels 16 bits de menys pes de `rs1`:

RV32I

```
        li      t0, 0x0000FFFF  # màscara de 16 bits
        and     rd, rs1, t0
```

`0xFFFF` no cap en l'immediat de 12 bits d'`andi` (rang $[-2^{11}, 2^{11}-1]$, és a dir, $[-2048, 2047]$), per la qual cosa cal carregar la màscara en un registre i fer servir la variant registre-registre `and`.

3.2.3 Aplicació d'`or`: posar bits a `1`

La instrucció `or` s'usa per **posar a 1** aquells bits d'un registre que valen `1` a la màscara.

Exemple 3.5: Posar bits a `1` amb `or`

Posada dels 16 bits de menys pes de `t0` a `1`:

`RV32I`

```
        li      t1, 0x0000FFFF  # màscara de 16 bits
        or      t0, t0, t1
```

`0xFFFF` no cap en l'immediat de 12 bits d'`ori`, per la qual cosa cal carregar la màscara en un registre.

3.2.4 Aplicació d'`xor`: complementar bits

La instrucció `xor` s'usa per **complementar** aquells bits d'un registre que valen `1` a la màscara.

Exemple 3.6: Complementar bits amb `xor`

Complement dels bits parells de `t0`:

`RV32I`

```
        li      t1, 0x55555555  # màscara amb uns als bits parells
        xor     t0, t0, t1
```

3.3 Comparacions i operacions booleanes

En **C**, les expressions enteres admeten els **operadors de comparació** `==`, `!=`, `<`, `<=`, `>`, `>=`. El resultat d'una comparació **no és d'un tipus booleà**, sinó un `int`: l'enter `1` si la comparació és certa i `0` en cas contrari (el tipus `_Bool` de C99 no canvia aquesta regla). Les **comparacions** seran **amb signe o sense signe** segons el tipus declarat dels operands.

En **C** existeixen també els **operadors booleans** `&&`, `||`, `!`. Cada operand s'interpreta com a **fals** si val `0`, o com a **cert** si val **diferent de `0`**. El resultat és també un enter **normalitzat**: `1` si cert, `0` si fals.

Essencial 3.4

En C, **qualsevol valor diferent de `0`** s'interpreta com a cert, de manera que dues expressions certes **no són necessàriament iguals**: `2` i `5` són totes dues certes, i `2 == 5` és fals. En canvi, les comparacions i els operadors booleans sempre donen com a resultat un enter normalitzat: `0` o `1`. Per exemple, si `y = 2` i `z = 0`, la sentència `x = (y >= z) + 5;` assigna a `x` el valor `6`.

3.3.1 Instruccions de comparació: `slt` i `sltu`

La instrucció `slt` compara dos enters i **posa a `1`** al registre destinació (LSb a `1` i la resta a `0`) si el primer **és menor** que el segon, o `0` en cas contrari. La instrucció `sltu` fa la mateixa operació, però tractant els operands com a naturals. Les variants immediates `slti` i `sltiu` comparen amb un immediat de 12 bits amb **extensió de signe**.

RISC-V 3.5: RV32I ISA — Instruccions de comparació

Mnemònic	Operands	Operació	Nom	Tipus
`slt`	`rd, rs1, rs2`	$rd \leftarrow (rs1 < rs2)\,?\,1 : 0 \quad (\mathbb{Z})$	*set less than*	R
`sltu`	`rd, rs1, rs2`	$rd \leftarrow (rs1 < rs2)\,?\,1 : 0 \quad (\mathbb{N})$	*set less than unsigned*	R
`slti`	`rd, rs1, imm12`	$rd \leftarrow (rs1 < SignExt(imm12))\,?\,1 : 0 \quad (\mathbb{Z})$	*set less than immediate*	I
`sltiu`	`rd, rs1, imm12`	$rd \leftarrow (rs1 < SignExt(imm12))\,?\,1 : 0 \quad (\mathbb{N})$	*set less than immediate unsigned*	I

Per a les comparacions freqüents amb zero, RISC-V també defineix pseudoinstruccions de salt condicional específiques (`beqz`, `bnez`, `bltz`, `bgez`, `bgtz`, `blez`), que es presenten amb la resta de pseudoinstruccions de salt a RISC-V 3.9.

3.3.2 Traducció de `==` i `!=`

Exemple 3.7: Traducció de `==` i `!=`

Si `t0`, `t1` i `t2` són enters:
Comparació amb zero:

C

```c
t2 = (t0 == 0);
```

RV32I

```
        sltiu   t2, t0, 1       # si t0 < 1 com a natural, llavors t0 == 0
```

- **Atenció**: És vàlid per a `t0 == 0` perquè comparar `t0` com a natural amb `1` dona `1` si i només si `t0 == 0` (ja que `0` és l'únic natural estrictament menor que `1`).

C

```c
t2 = (t0 != 0);
```

RV32I

```
        sltu    t2, zero, t0    # si 0 < t0 com a natural, llavors t0 != 0
```

Comparació entre dos registres:

C

```c
t2 = (t0 == t1);        // equival a: t2 = ((t0 - t1) == 0)
```

RV32I

```
        sub     t2, t0, t1
        sltiu   t2, t2, 1
```

C

```
t2 = (t0 != t1);        // equival a: t2 = ((t0 - t1) != 0)
```

RV32I

```
        sub     t2, t0, t1
        sltu    t2, zero, t2
```

3.3.3 Traducció de la negació booleana `!`

La negació booleana no es pot traduir amb `xori rd, rs, -1` (negació bit a bit), ja que aquest sempre dona un resultat diferent de zero tant si l'operand val `0` com `1`. La traducció correcta és convertir-la en una comparació amb zero:

Exemple 3.8: Traducció de `!`

Si `t0` i `t1` són enters:

C

```
t1 = !t0;               // equival a: t1 = (t0 == 0)
```

RV32I

```
        sltiu   t1, t0, 1
```

- Si se sap amb certesa que `t0` val `0` o `1` (per exemple, si és el resultat d'una comparació anterior), es pot traduir també amb:

RV32I

```
        xori    t1, t0, 1       # inverteix l'LSb
```

3.3.4 Traducció de <, >, <=, >=

Exemple 3.9: Traducció de <, >, <=, >=

Si `t0`, `t1` i `t2` són enters:

C

```
        t2 = (t0 < t1);
```

RV32I

```
        slt     t2, t0, t1
```

C

```
        t2 = (t0 > t1);         // equival a: t1 < t0
```

RV32I

```
        slt     t2, t1, t0
```

C

```
        t2 = (t0 <= t1);        // equival a: !(t0 > t1)
```

RV32I

```
        slt     t2, t1, t0
        sltiu   t2, t2, 1       # negació
```

C

```
        t2 = (t0 >= t1);        // equival a: !(t0 < t1)
```

RV32I

```
        slt     t2, t0, t1
        sltiu   t2, t2, 1       # negació
```

- **Atenció**: Per a operands de tipus `unsigned`, cal substituir `slt` per `sltu` a totes les traduccions d'aquest exemple.

3.3.5 Traducció de `&&` i `||`: avaluació tardana

En **C**, els operadors lògics `&&` i `||` s'avaluen d'esquerra a dreta de forma **tardana** (***lazy***): si el valor del primer operand ja determina el resultat de tota la condició, **el segon operand no s'avalua**. Això succeeix si el primer operand d'una `&&` és fals, o si el primer operand d'una `||` és cert. La seva traducció implica l'ús d'instruccions de salt i s'estudia a Secció 3.6.2.

> Essencial 3.5
>
> En **C**, l'avaluació tardana de `&&` i `||` **no és opcional**: la imposa la semàntica del llenguatge. Per tant, la **traducció a assemblador l'ha de respectar**.

3.4 Salts

Les instruccions de salt modifiquen el flux d'execució **desviant-lo** a una adreça diferent de la instrucció següent. A **RV32I** es distingeixen dos eixos de classificació:

- **Condicional vs. incondicional**: els **salts condicionals** (*branches*), com `beq` o `bne`, salten només si es compleix una condició; els **salts incondicionals** (*jumps*), com `jal` i `jalr`, salten sempre.
- **Relatiu al PC vs. indirecte**: els **salts relatius al PC** calculen l'adreça de destinació sumant un desplaçament al valor actual del PC; els **salts indirectes** la calculen a partir d'un registre base (mode base + desplaçament, com `jalr`).

La combinació d'aquests dos eixos dona lloc a tres categories:

	Relatiu al PC	Indirecte
Condicional	`beq`, `bne`, `blt`, `bge`, `bltu`, `bgeu`	—
Incondicional	`jal`	`jalr`

Les pseudoinstruccions de salt (`j`, `jr`, `ret` i les de RISC-V 3.8 i RISC-V 3.9) s'expandeixen a aquestes instruccions.

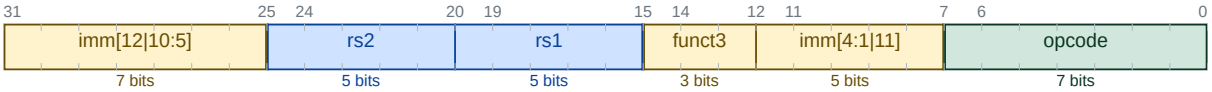
> Essencial 3.6
>
> A **RV32I**, tots els salts **condicionals** són **relatius al PC**. No existeix cap salt condicional indirecte.

3.4.1 Salts condicionals

Els salts condicionals salten a l'adreça especificada per l'etiqueta si es compleix la condició, o continuen a la instrucció següent si no es compleix. En el llenguatge màquina no es codifica l'adreça absoluta de destinació, sinó un **desplaçament relatiu al PC**. L'assemblador calcula automàticament aquest desplaçament a partir de l'etiqueta.

Les instruccions de salt condicional són de format **Tipus-B** (RISC-V 3.6), variant del format **Tipus-S** ja presentat a Tema 2 (RISC-V 2.28). El desplaçament codificat és de **13 bits amb signe**, però el bit 0 és sempre implícit (val `0`) perquè les destinacions de salt estan alineades a, com a mínim, 2 bytes. D'aquesta manera, la instrucció emmagatzema 12 bits i obté un rang efectiu de **±4 KiB** (Essencial 2.5) respecte al PC. A **RV32I** base (cas d'**EC**), les instruccions estan alineades a 4 bytes; el format B manté la granularitat de 2 bytes per compatibilitat amb l'extensió **C** (*compressed*).

RISC-V 3.6: RV32I ISA — Format Tipus-B



Els bits de l'immediat estan repartits en dos camps no contigus (`imm[12|10:5]` a [31:25] i `imm[4:1|11]` a [11:7]) per mantenir `rs1` i `rs2` en les mateixes posicions que en els formats R i S, cosa que simplifica la descodificació al maquinari.

RISC-V 3.7: RV32I ISA — Instruccions de salt condicional

Mnemònic	Operands	Operació	Nom	Tipus
`beq`	`rs1, rs2, label`	$rs1 == rs2\,?\,PC \leftarrow PC + SignExt(offset13)$	*branch equal*	B
`bne`	`rs1, rs2, label`	$rs1 != rs2\,?\,PC \leftarrow PC + SignExt(offset13)$	*branch not equal*	B
`blt`	`rs1, rs2, label`	$rs1 < rs2\,?\,PC \leftarrow PC + SignExt(offset13)$ $(\mathbb{Z})$	*branch less than*	B
`bge`	`rs1, rs2, label`	$rs1 >= rs2\,?\,PC \leftarrow PC + SignExt(offset13)$ $(\mathbb{Z})$	*branch greater than or equal*	B
`bltu`	`rs1, rs2, label`	$rs1 < rs2\,?\,PC \leftarrow PC + SignExt(offset13)$ $(\mathbb{N})$	*branch less than unsigned*	B
`bgeu`	`rs1, rs2, label`	$rs1 >= rs2\,?\,PC \leftarrow PC + SignExt(offset13)$ $(\mathbb{N})$	*branch greater than or equal unsigned*	B

RISC-V 3.8: Pseudoinstruccions — Salt condicional

Pseudoinstrucció	Operació	Expansió
`bgt rs1, rs2, label`	$rs1 > rs2\,?\,PC \leftarrow PC + SignExt(offset13)$ $(\mathbb{Z})$	`blt rs2, rs1, label`
`ble rs1, rs2, label`	$rs1 <= rs2\,?\,PC \leftarrow PC + SignExt(offset13)$ $(\mathbb{Z})$	`bge rs2, rs1, label`
`bgtu rs1, rs2, label`	$rs1 > rs2\,?\,PC \leftarrow PC + SignExt(offset13)$ $(\mathbb{N})$	`bltu rs2, rs1, label`
`bleu rs1, rs2, label`	$rs1 <= rs2\,?\,PC \leftarrow PC + SignExt(offset13)$ $(\mathbb{N})$	`bgeu rs2, rs1, label`

RISC-V 3.9: Pseudoinstruccions — Salt condicional amb zero

Pseudoinstrucció	Operació	Expansió
`beqz rs, label`	$rs == 0\,?\,PC \leftarrow PC + SignExt(offset13)$	`beq rs, zero, label`
`bnez rs, label`	$rs! = 0\,?\,PC \leftarrow PC + SignExt(offset13)$	`bne rs, zero, label`
`bltz rs, label`	$rs < 0\,?\,PC \leftarrow PC + SignExt(offset13)$ $(\mathbb{Z})$	`blt rs, zero, label`
`bgez rs, label`	$rs >= 0\,?\,PC \leftarrow PC + SignExt(offset13)$ $(\mathbb{Z})$	`bge rs, zero, label`
`bgtz rs, label`	$rs > 0\,?\,PC \leftarrow PC + SignExt(offset13)$ $(\mathbb{Z})$	`blt zero, rs, label`

`blez rs, label` $rs <= 0\,?\,PC \leftarrow PC + SignExt(offset13)$ $(\mathbb{Z})$ `bge zero, rs, label`

Exemple 3.10: Càlcul del desplaçament d'un salt condicional

Sigui el programa del Codi 3.1 i la vista que en dona la finestra *Text Segment* de RARS:

Codi 3.1: Un `beq` que salta per sobre de dues `nop`.

RV32I

```
        beq     zero, zero, etiqueta    # 0x00400000
        nop                             # 0x00400004
        nop                             # 0x00400008
etiqueta:
        sub     zero, zero, zero        # 0x0040000C
```

Adreça	Codi	Bàsic	Font
0x00400000	0x00000663	beq x0,x0,0x0000000c	beq zero, zero, etiqueta
0x00400004	0x00000013	addi x0,x0,0	nop
0x00400008	0x00000013	addi x0,x0,0	nop
0x0040000C	0x40000033	sub x0,x0,x0	sub zero, zero, zero

L'assemblador ha calculat el desplaçament com:

$$\text{desplaçament} = \text{@etiqueta} - \text{@beq} = \texttt{0x0040000C} - \texttt{0x00400000} = 12 = \texttt{0x0000000C}$$

Aquest valor apareix directament al panell *Basic* de RARS com `0x0000000c`, confirmat per la codificació binària de la instrucció: `0x00000663`.

Per què 12 i no 8?

El desplaçament es calcula respecte a l'adreça de la instrucció de salt (`0x00400000`), no respecte a la instrucció següent (`0x00400004`): de `beq` fins a `etiqueta` hi ha 3 instruccions de 4 bytes (`beq` i les dues `nop`).

Exemple 3.11: Execució condicional amb `bne`

Execució si la variable `b` té actius els bits 0 i 4, i inactius els bits 2 i 6:

C

```
a = b & 0x55;       // màscara: selecció bits 0, 2, 4 i 6
if (a == 0x11)      // bits 0 i 4 actius, 2 i 6 inactius
    { ... }
```

Suposem que `a` i `b` ocupen els registres `t0` i `t1`:

RV32I

```
        andi    t0, t1, 0x0055      # selecciona els bits 0, 2, 4 i 6
        li      t4, 0x0011
        bne     t0, t4, fisi        # if (a == 0x11) ...
        ...                         # codi si es compleix la condició
fisi:
```

Les instruccions de salt condicional són la base per implementar les sentències alternatives (Secció 3.6) i els bucles (Secció 3.7).

3.4.2 Salts incondicionals relatius al PC

La instrucció `jal` (*jump and link*) salta a una adreça especificada per una etiqueta, codificada com un desplaçament relatiu al PC, i guarda l'**adreça de retorn** (`pc + 4`) al registre `rd`. Quan `rd = ra`, s'usa per cridar subrutines (vegeu la presentació del concepte de subrutina a Secció 3.9). Si `rd = zero`, salta perdent l'adreça de retorn.

La instrucció `jal` és de format **Tipus-J** (RISC-V 3.10), variant del format **Tipus-U**, presentat a Tema 2 amb `lui` (RISC-V 2.27). Com en el format B (RISC-V 3.6), el bit 0 de l'immediat és implícit (val `0`), de manera que la instrucció codifica 20 bits i obté un rang efectiu de **±1 MiB** respecte al PC.

RISC-V 3.10: RV32I ISA — Format Tipus-J



Com en el format B, els bits de l'immediat estan reordenats (`imm[20|10:1|11|19:12]` a [31:12]) perquè cada bit coincideixi tant com sigui possible amb la posició que ocupa als altres formats —el de signe, sempre al bit 31—, cosa que simplifica el maquinari que reconstrueix l'immediat. `rd` queda a la mateixa posició que en els formats R, I i U perquè l'immediat ocupa, com al format U, els bits [31:12].

RISC-V 3.11: RV32I ISA — Instrucció de salt incondicional relatiu al PC

Mnemònic	Operands	Operació	Nom	Tipus
`jal`	`rd, label`	$rd \leftarrow PC + 4; PC \leftarrow PC + SignExt(offset21)$	*jump and link*	J

RISC-V 3.12: Pseudoinstruccions — Salt incondicional relatiu al PC: `j`

Pseudoinstrucció	Operació	Expansió
`j label`	$PC \leftarrow PC + SignExt(offset21)$	`jal zero, label`

3.4.3 Salts incondicionals indirectes

La instrucció `jalr` (*jump and link register*) salta a l'adreça calculada sumant el registre `rs1` i un immediat de 12 bits (**mode base + desplaçament**), i guarda l'adreça de retorn a `rd`. A diferència de `jal`, l'adreça de destinació no és relativa al PC sinó que depèn del contingut d'un registre; d'aquí el terme **salt indirecte**. S'usa principalment per retornar de subrutines i per saltar a adreces llunyanes desconegudes en temps de compilació. S'escriu, com les instruccions de lectura i escriptura, amb l'operand en la forma `desplaçament(rs1)` (EC 2.17): `jalr rd, desplaçament(rs1)`.

RISC-V 3.13: RV32I ISA — Instrucció de salt incondicional indirecte

Taula 3.1: {tbl-colwidths="[13,19,33,28,7]"}‘

Mnemònic	Operands	Operació	Nom	Tipus
`jalr`	`rd, offset(rs1)`	$PC \leftarrow (rs1 + SignExt(offset))\&\overline{1};\ rd \leftarrow PC_{ant} + 4$	*jump and link register*	I

RISC-V 3.14: Pseudoinstruccions — Salt incondicional indirecte: `jr` i `ret`

Pseudoinstrucció	Operació	Expansió
`jr rs`	$PC \leftarrow rs$	`jalr zero, 0(rs)`
`ret`	$PC \leftarrow ra$	`jalr zero, 0(ra)`

Exemple 3.12: Salt a adreça llunyana amb `jr`

RV32I

```
        la      t0, etiqueta_llunyana
        jalr    zero, 0(t0)             # equivalent a: jr t0
```

EC 3.4

A **EC**, els salts tenen rols ben diferenciats: els **condicionals** i `j` s'usen per implementar el control de flux local (`if-then-else`, bucles); `jal` (amb `rd = ra`), per cridar subrutines; i els salts **indirectes** (`jalr`, `jr`, `ret`), quan la destinació està en un registre, com en retorns de subrutina o en les taules de salts (Secció 3.6.3).

3.5 Revisió dels modes d'adreçament

Ara que ja s'han presentat totes les instruccions de processament i control de flux de la base **RV32I** (RISC-V 2.2), tret d'`ecall` (Secció 9.5) —les instruccions de sistema, com `ecall` i `ebreak` (presentada a Tema 9), queden fora d'aquesta classificació—, l'exercici de classificar-les segons el **mode d'adreçament** (EC 2.15) pot ajudar no només a interioritzar el repertori de l'ISA, sinó també, a un nivell més abstracte, a anar conceptualitzant el domini d'estructura de computadors com a disciplina i el rol que li pertoca dins del context maquinari-programari (Secció 1.1.1).

La classificació proposada inclou quatre decisions que cal justificar:

- **`lui`** s'ha classificat com a mode **immediat** perquè el seu únic operand font és un valor constant de 20 bits codificat directament a la instrucció; no hi ha cap registre base ni cap referència al PC.
- **`auipc`** s'ha classificat com a mode **relatiu al PC** perquè suma un immediat de 20 bits al valor actual del PC per construir adreces relatives a la posició actual del programa.

- **`jalr`** s'ha classificat com a mode **base + desplaçament** perquè calcula l'adreça de salt sumant un registre base (`rs1`) i un immediat de 12 bits, exactament com fan les instruccions de lectura i escriptura. L'única diferència és que el resultat s'usa com a adreça de salt en lloc d'adreça de memòria.
- **`ecall`** no s'ha inclòs perquè no té cap operand ni accedeix a cap dada: el servei el decideix el contingut d'`a7` (Secció 9.5).

Mode	Instruccions
Registre	`add`, `sub`, `and`, `or`, `xor`, `sll`, `srl`, `sra`, `slt`, `sltu`
Immediat	`addi`, `andi`, `ori`, `xori`, `slli`, `srli`, `srai`, `slti`, `sltiu`, `lui`
Base + desplaçament	`lw`, `lh`, `lb`, `lhu`, `lbu`, `sw`, `sh`, `sb`, `jalr`
Relatiu al PC	`beq`, `bne`, `blt`, `bge`, `bltu`, `bgeu`, `jal`, `auipc`

3.6 Sentències alternatives: `if-then-else` i `switch`

3.6.1 Sentència `if-then-else`

En **C**, la condició d'un `if` és una expressió entera qualsevol: el resultat s'avalua com a fals si val zero, o com a cert si val diferent de zero. El cas més freqüent és que la condició sigui una comparació o una operació booleana, però no és obligatori.

Sigui el cas general en C:

C

```c
if (condicio)
    sentencia_then;
else
    sentencia_else;
```

El patró de traducció a RV32I és:

RV32I

```
        # avaluar condicio
        # salt si és falsa a sino
        # traducció de sentencia_then
        j       fisi
sino:
        # traducció de sentencia_else
fisi:
```

La clàusula `sino` (`else`) és opcional. Si no existeix, el primer salt va directament a `fisi`, que es col · loca just després de la traducció de `sentencia_then`.

Exemple 3.13: `if-then-else` amb condició simple

C

```c
if (a >= b)
```

```
    d = a;
else
    d = b;
```

Suposem que `a`, `b`, `d` són enters i ocupen `t0`, `t1`, `t3`:

RV32I

```
        blt     t0, t1, sino    # salta si a < b
        mv      t3, t0          # d = a
        j       fisi
sino:
        mv      t3, t1          # d = b
fisi:
```

3.6.2 Avaluació tardana als salts condicionals encadenats

L'avaluació tardana es tradueix naturalment amb salts condicionals encadenats.

Exemple 3.14: `if` amb operador `&&`

C

```
if (a >= b && a < c)
    d = a;
else
    d = b;
```

Suposem que `a`, `b`, `c`, `d` ocupen `t0`, `t1`, `t2`, `t3`:

RV32I

```
        blt     t0, t1, sino    # salta si a < b  (1a condició falsa)
        bge     t0, t2, sino    # salta si a >= c (2a condició falsa)
        mv      t3, t0          # d = a
        j       fisi
sino:
        mv      t3, t1          # d = b
fisi:
```

Si el primer operand de `&&` és fals (`a < b`), el salt a `sino` evita avaluar el segon operand.

Exemple 3.15: `if` amb operador `||`

C

```
if (a >= b || a < c)
    d = a;
else
    d = b;
```

Suposem que `a`, `b`, `c`, `d` ocupen `t0`, `t1`, `t2`, `t3`:

RV32I

```
        bge     t0, t1, llavors # salta si a >= b (1a condició certa)
        bge     t0, t2, sino    # salta si a >= c (2a condició falsa)
llavors:
        mv      t3, t0          # d = a
        j       fisi
sino:
        mv      t3, t1          # d = b
fisi:
```

Si el primer operand de `||` és cert (`a >= b`), el salt a `llavors` evita avaluar el segon operand.

3.6.3 Sentència `switch`

La sentència `switch` selecciona una d'entre múltiples clàusules `case` comparant el resultat d'una expressió entera amb constants. Si no en coincideix cap, s'executa la clàusula `default` si existeix.

C

```c
switch (expressio) {
    case const1: sentencies1; break;
    case const2: sentencies2; break;
    ...
    default:     sentenciesN;
}
```

El mètode més senzill de traducció és una **cadena d'`if-then-else`** que comprova les etiquetes seqüencialment. Un mètode més eficient, quan les constants formen un rang dens, consisteix en una **taula de salts** (*jump table*): un vector de paraules on cada element conté l'adreça del codi d'una clàusula `case`, indexat pel resultat de l'expressió.

Exemple 3.16: `switch` amb taula de salts

Suposem que `t0` conté el resultat de l'expressió del `switch` i que `switch_vec` és un vector de paraules amb les adreces de cada clàusula:

Codi 3.2: Salt a la clàusula del `switch` a través de la taula de salts.

RV32I

```
        la      t1, switch_vec  # adreça base del vector
        slli    t2, t0, 2       # desplaçament = índex * 4 (mida d'una paraula)
        add     t1, t1, t2      # adreça efectiva de l'element
        lw      t3, 0(t1)       # llegeix l'adreça de la clàusula
        jalr    zero, 0(t3)     # salta al codi de la clàusula (equivalent a: jr t3)
```

La taula de salts es declara a la secció `.data`:

RV32I

```
.data
switch_vec:
        .word   cas_0           # apunta al codi de case 0
        .word   cas_1           # apunta al codi de case 1
        .word   cas_2           # apunta al codi de case 2
        .word   cas_3           # apunta al codi de case 3
```

La versió simplificada del Codi 3.2 assumeix que l'expressió del `switch` només pot valer `0`–`3` (rang cobert exactament pel vector) i que no hi ha clàusula `default`. El cas general —amb `default` i sense garantia que l'expressió caigui dins del rang— es tracta a continuació.

Aprofundiment 3.2: Implementació de la taula de salts en detall

La taula de salts és eficient quan les constants del `switch` formen un rang dens (per exemple, `0`, `1`, `2`, `3`). Si les constants estan disperses o el seu rang és molt gran, és preferible la cadena d'`if-then-else`, ja que la taula de salts requeriria moltes entrades per a constants que no existeixen. Cal tenir en compte que si les constants no comencen a `0`, cal restar el valor mínim al resultat de l'expressió abans d'indexar el vector. Per exemple, si les constants van de `5` a `8`:

RV32I

```
        addi    t0, t0, -5          # normalitza l'índex (resta el valor mínim)
        li      t4, 4
        bgeu    t0, t4, cas_defecte # si t0 >= 4, fora de rang
        la      t1, switch_vec
        slli    t2, t0, 2
        add     t1, t1, t2
        lw      t3, 0(t1)
        jalr    zero, 0(t3)         # equivalent a: jr t3
```

Nota: El `break` al final de cada `case` s'ha de traduir amb un salt incondicional `j` a l'etiqueta que marca el final del `switch`. Si no hi ha `break`, l'execució continua (*fall-through*) al codi del `case` següent, sense necessitat de cap instrucció addicional.

3.7 Sentències iteratives: `while`, `do-while` i `for`

3.7.1 Sentència `while`

La sentència `while` executa zero o més iteracions mentre es compleixi la condició. La condició s'avalua **abans** de cada iteració, de manera que si inicialment és falsa no s'executa cap iteració.

Sintaxi:

C

```
while (condicio)
    sentencia_cos;
```

El patró de traducció a RV32I és:

Codi 3.3: Patró de traducció d'un `while`.

RV32I

```
bucle:
        # avaluar condicio
        # salt si és falsa a fibucle
        # traducció de sentencia_cos
        j       bucle
fibucle:
```

EC 3.5

En aquesta secció s'usa l'etiqueta `bucle:` tant per al `while` com per al `do-while` i el `for` per fer explícit que tots tres tipus de bucles tenen la mateixa estructura bàsica de control de flux, i per poder parlar d'optimitzacions de bucle de manera genèrica.
En la pràctica, s'usen etiquetes més descriptives (per exemple, `while:`, `fiwhile:`, `do:`, `for:`, `fifor:`) per facilitar la lectura i el manteniment del codi.

Exemple 3.17: Divisió entera de nombres positius amb `while`

C

```c
q = 0;
while (dd >= dr) {
    dd = dd - dr;
    q++;
}
```

Suposem que `dd`, `dr`, `q` ocupen `t1`, `t2`, `t3`:

RV32I

```
        li      t3, 0           # q = 0
bucle:
        blt     t1, t2, fibucle # salta si dd < dr
        sub     t1, t1, t2      # dd = dd - dr
        addi    t3, t3, 1       # q++
        j       bucle
fibucle:
```

Exemple 3.18: Avaluació de la condició al final

El patró del Codi 3.3 conté un salt incondicional `j bucle` al final de cada iteració. Es pot eliminar avaluant la condició al **final** del bucle en lloc de fer-ho al principi. Però cal assegurar que es manté la semàntica del `while`: si la condició és inicialment falsa, no s'ha d'executar cap iteració. Per això cal una comprovació prèvia a l'entrada:

RV32I

```
        li      t3, 0           # q = 0
        blt     t1, t2, fibucle # comprovació inicial: salta si dd < dr
bucle:
        sub     t1, t1, t2      # dd = dd - dr
        addi    t3, t3, 1       # q++
        bge     t1, t2, bucle   # salta si dd >= dr
fibucle:
```

El cos del bucle s'executa ara sense cap salt incondicional: el salt condicional al final torna al principi si la condició segueix sent certa, o en cas contrari cau a `fibucle`. Això estalvia un salt per iteració. Aquesta transformació es formalitza més endavant com a Optimització #2 (Secció 4.6.2). Si la condició és complexa i escriure-la dues vegades seria costós, es pot substituir la comprovació inicial per un salt incondicional a la instrucció d'avaluació al final del bucle:

RV32I

```
        li      t3, 0           # q = 0
        j       test            # salta a la comprovació inicial
bucle:
        sub     t1, t1, t2      # dd = dd - dr
        addi    t3, t3, 1       # q++
test:
        bge     t1, t2, bucle   # salta si dd >= dr
fibucle:
```

3.7.2 Sentència `do-while`

La sentència `do-while` executa el cos **sempre almenys una vegada**, i la condició s'avalua al final de cada iteració.

C

```
do
    sentencia_cos;
while (condicio);
```

El patró de traducció és més senzill que el del `while` perquè no cal comprovació inicial: el cos s'executa primer i el salt condicional al final decideix si tornar a iterar:

RV32I

```
bucle:
        # traducció de sentencia_cos
        # avaluar condicio
        # salt si és certa a bucle
```

És el mateix patró que deixa l'avaluació de la condició al final d'un `while` (Exemple 3.18), sense la comprovació d'entrada.

Exemple 3.19: `do-while`: cerca del primer element no negatiu

C

```
int n;
do {
    n = v[i];
    i++;
} while (n < 0);
```

Suposem que `n`, `i` ocupen `t0`, `t1` i `v` és un vector global d'enters:

RV32I

```
        la      t2, v               # adreça base de v
bucle:
        slli    t3, t1, 2           # i * 4
```

```
        add     t3, t2, t3          # @v[i]
        lw      t0, 0(t3)           # n = v[i]
        addi    t1, t1, 1           # i++
        bltz    t0, bucle           # n < 0: torna a iterar
```

El bucle avança per `v` fins a trobar el primer element no negatiu. La condició s'avalua després d'executar el cos, de manera que sempre es llegeix almenys un element.

3.7.3 Sentència `for`

Al bucle `for`, el més habitual és que `sentencia1` inicialitzi la variable de control del bucle (típicament un comptador) i que `sentencia2` l'actualitzi després de cada iteració.

C

```c
for (sentencia1; condicio; sentencia2) {
    sentencia3;
}
```

Quan es fa així, el `for` agrupa tots els elements que controlen el bucle en una única capçalera, cosa que el fa més fàcil de llegir que el `while` equivalent (si el cos no conté cap `continue`):

C

```c
sentencia1;
while (condicio) {
    sentencia3;
    sentencia2;
}
```

Exemple 3.20: Traducció ingènua del bucle `for`

Codi 3.4: Suma dels elements d'un vector amb un `for`.

C

```c
int sum = 0;
for (int i = 0; i < N; i++)
    sum += v[i];
```

Suposem que `sum`, `i`, `N` ocupen `t0`, `t1`, `t2`, i que `v` és un vector global d'enters:

RV32I

```
        li      t0, 0           # sum = 0
        li      t1, 0           # i = 0
        la      t3, v           # adreça base de v
bucle:
        bge     t1, t2, fibucle # salta si i >= N
        slli    t4, t1, 2       # i * 4
        add     t4, t3, t4      # @v[i]
        lw      t4, 0(t4)       # v[i]
        add     t0, t0, t4      # sum += v[i]
```

```
        addi    t1, t1, 1       # i++
        j       bucle
fibucle:
```

Exemple 3.21: Eliminació de la variable d'inducció

Una **variable d'inducció** és una variable que s'incrementa o decrementa en una quantitat constant a cada iteració d'un bucle. El compilador pot eliminar-les substituint-les per l'aritmètica de punters directament. Aquesta transformació es formalitza més endavant com a Optimització #3 (Secció 4.6.3).
Per exemple, en el `for` del Codi 3.4, en lloc de calcular `@v[i] = @v[0] + i*4` a cada iteració, pot mantenir directament un punter `p` que avanci de 4 en 4 bytes:

C

```c
int sum = 0;
int *p = v;
int *fi = v + N;    // adreça del primer element fora del vector
while (p != fi) {
    sum += *p;
    p++;
}
```

Suposem que `sum`, `N`, `p` i `fi` ocupen `t0`, `t2`, `t3`, `t4`:

RV32I

```
        li      t0, 0           # sum = 0
        la      t3, v           # p = @v[0]
        slli    t4, t2, 2       # N * 4
        add     t4, t3, t4      # fi = @v[0] + N*4
bucle:
        beq     t3, t4, fibucle # salta si p == fi
        lw      t5, 0(t3)       # *p
        add     t0, t0, t5      # sum += *p
        addi    t3, t3, 4       # p++
        j       bucle
fibucle:
```

Aquesta optimització elimina la multiplicació `i*4` dins el bucle i redueix el nombre d'instruccions per iteració. Els compiladors moderns l'apliquen automàticament.

3.8 Estructura de la memòria

Les dades (instruccions i variables) d'un programa poden tenir un mode d'emmagatzemament **estàtic** o **dinàmic**.

A l'emmagatzemament **estàtic**, l'espai de memòria reservat per cada objecte és el mateix durant tot el temps d'execució del programa. Les reserves es fan en **temps de compilació** i no es poden modificar en temps d'execució. Les **instruccions**, les **variables globals** i les **constants globals** sempre s'emmagatzemen estàticament. Els segments de memòria estàtica principals són `.text` per a les instruccions del programa en assemblador i `.data` per a les variables i les constants globals, als quals

s'escriu en temps de compilació mitjançant les directives de segment (RISC-V 2.9).

A l'emmagatzemament **dinàmic** la gestió de memòria (reserva i alliberament) es fa en **temps d'execució**. Els segments de memòria dinàmica principals són la **pila** (***stack***) i el ***heap*** (a l'Optimot, *monticle*, un terme poc usat). La pila s'usa per a les **variables locals** i els **registres desats** (Secció 3.9.4), mentre que el *heap* s'usa per a les variables dinàmiques, on la reserva i l'alliberament es fan explícitament per part del programador (mitjançant `malloc()` i `free()`).

Exemple 3.22: Les tres ubicacions de les dades: `.data`, pila i *heap*

C

```c
int gvec[100];      // global: emmagatzemament estàtic (.data)
int *pvec;          // global: punter a dada dinàmica (heap)

int f() {
    int lvec[100];  // local: emmagatzemament dinàmic (pila)
    ...
    pvec = malloc(400);     // reserva espai al heap
    ...
    pvec[10] = gvec[10] + lvec[10];
    ...
}

int g() {
    ...
    free(pvec);     // allibera espai del heap
    ...
}
```

3.8.1 RARS — Mapa de memòria

RARS organitza l'espai d'adreces en les regions principals (segments, àrees, etc.) següents (Figura 3.1); n'hi ha d'altres, com la de l'E/S mapada en memòria, que a EC no es fan servir:

RISC-V 3.15: RARS — Segments de memòria

Regió	Tipus	Adreça d'inici	Contingut	Creix cap a …
`.text`	Estàtic	`0x00400000`	Codi màquina (instruccions)	—
`.data`	Estàtic	`0x10010000`	Variables globals	—
heap	Dinàmic	`0x10040000`	Variables dinàmiques (`malloc()`/`free()`)	adreces altes
pila (*stack*)	Dinàmic	`0x7FFFFFFC` (`sp`: `0x7FFFEFFC`, RISC-V 3.21)	Variables locals, registres desats	adreces baixes
nucli (*kernel*)	Estàtic	`0x80000000`	Codi i dades del sistema operatiu	—

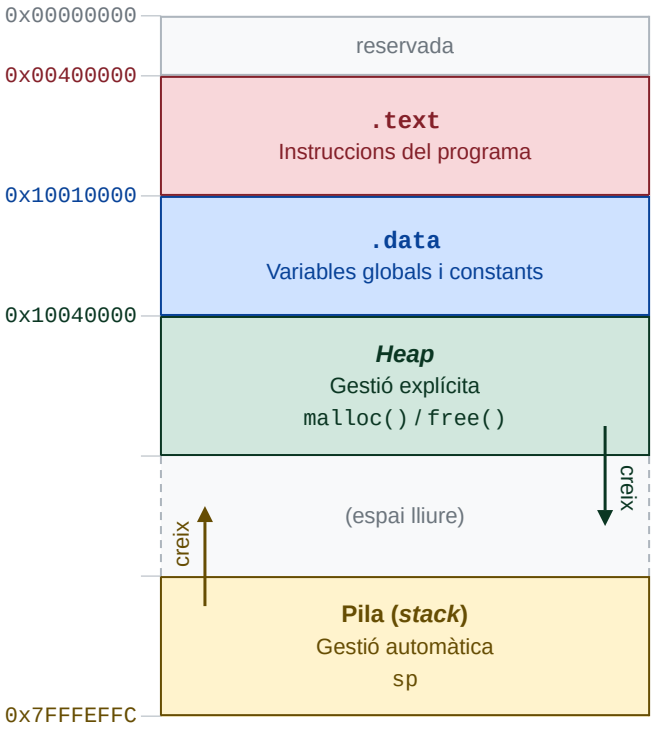


Figura 3.1: Mapa de memòria de RARS: regions `.text`, `.data`, *heap* i pila, amb les adreces d'inici de cada segment.

Aprofundiment 3.3

RARS: *Settings* → *Memory Configuration…* mostra totes les regions de memòria.

Aprofundiment 3.4: El punter global `gp` (*global pointer*)

L'ABI reserva el punter global `gp` (`x3`, *global pointer*) per optimitzar l'accés a variables globals petites situades dins d'un rang de ±2 KiB respecte al valor de `gp`. L'entorn d'execució l'inicialitza en arrencar el programa (a RARS, a `0x10008000`); el programador no l'ha de manipular mai directament. A **EC** no s'estudia aquest mecanisme.

3.8.2 Accés a variables globals

En **RV32I**, la pseudoinstrucció `la` s'expandeix en dues instruccions que carreguen l'adreça de 32 bits de la variable en un registre. Un cop carregada l'adreça, l'accés a la variable es fa amb una instrucció de `lw`/`sw` (o les variants de byte i de mitja paraula):

`RV32I`

```
        la      t0, variable    # t0 <- adreça de variable
        lw      t1, 0(t0)       # t1 <- valor de variable
        addi    t1, t1, 1       # t1++
        sw      t1, 0(t0)       # variable <- t1
```

Si s'accedeix diverses vegades a la mateixa variable, és eficient carregar l'adreça una sola vegada en un registre i reutilitzar-la.

La primera de les dues instruccions en què s'expandeix `la`, `auipc` (RISC-V 2.14), és de format **Tipus-U**, com `lui`, presentat a Tema 2 (RISC-V 2.27); la segona, `addi`, és de format Tipus-I.

3.9 Subrutines

Una **subrutina** és un subprograma que executa una tasca concreta, que pot admetre **paràmetres** d'entrada i retornar **resultats**. Pot ser invocada des de múltiples punts d'un programa sense haver de ser reescrita cada vegada. És la generalització del que en altres contextos s'anomena acció, **funció** (C), procediment, mètode, etc. Permet escriure programes estructurats aplicant conceptes com el disseny descendent, la reutilització de codi i la programació modular.

> Essencial 3.7
>
> A diferència d'una subrutina, una macro no genera cap crida: l'assemblador insereix el codi a cada punt d'invocació, de manera que si s'invoca n vegades, el codi apareix n vegades a l'executable. Una subrutina, en canvi, és codi compartit que resideix en un únic lloc de memòria i s'executa per salt, amb el cost d'una crida i un retorn (Secció 2.3.5.3).

Per tal que una subrutina i el codi que la **crida** puguin ser programats en llenguatge d'assemblador (o compilats) independentment, cal que la subrutina declari amb precisió la seva **interfície** (nom, paràmetres, resultat i tipus respectius), i que tots dos s'ajustin a les regles de l'**ABI**, la qual, per a cada ISA i sistema operatiu, defineix les convencions que permeten compartir codi emmagatzemat en biblioteques.

> Essencial 3.8
>
> Una subrutina és:
>
> - **Uninivell** o **fulla** (*leaf*), si el seu codi no conté cap crida a una altra subrutina.
> - **Multinivell** o **no fulla** (*non-leaf*), si el seu codi conté alguna crida a una altra subrutina (les subrutines recursives en són un cas particular).

3.9.1 Crida i retorn (el registre `ra`)

Des del punt de vista del control del flux d'execució, quan es crida una subrutina cal guardar l'**adreça de retorn**, és a dir, l'adreça de la instrucció que segueix la instrucció de crida. L'ABI d'RV32I reserva el registre `ra` (***return address***, `x1`) per a aquest propòsit.

La instrucció `jal ra, label` (RISC-V 3.11) salta a l'adreça indicada per `label` (primera instrucció de la subrutina cridada) i guarda l'adreça de retorn (`PC + 4`) a `ra`. Per retornar, s'usa la pseudoinstrucció `ret` (RISC-V 3.14), que salta a l'adreça continguda a `ra`.

> Exemple 3.23: Crida d'una subrutina des de dos punts diferents d'un programa

RV32I

```
# Codi que fa les crides
        jal     ra, sub         # ra = adr1; salta a sub
adr1:   ...
        jal     ra, sub         # ra = adr2; salta a sub
adr2:   ...
```

```
# Codi de la subrutina
sub:
        ...
        ret                     # salta a ra (adr1 o adr2 segons el cas)
```

Essencial 3.9

Un codi és:

- ***caller***, si fa una crida a una funció o subrutina.
- ***callee***, si és la funció o subrutina cridada.

Per analogia, és comú emprar els termes mare/pare i fill/filla per referir-se a la funció o subrutina que crida i a la funció o subrutina cridada, respectivament. Quan hi ha tres nivells de crida, l'analogia s'expandeix a tres generacions, avi-pare-fill.

RISC-V 3.16: RV32I ABI — Gestió dels registres de propòsit general: *caller-saved* vs. *callee-saved*

La columna «*Saver*» de les taules de registres de l'ISA i l'ABI (RISC-V 2.24) estableix si un registre és *caller-saved* o *callee-saved*.

- **Caller-saved**: És la subrutina que crida qui ha de preservar-lo si vol recuperar el seu valor després de la crida.
- **Callee-saved**: És la subrutina cridada qui ha de preservar-lo si vol usar-lo i, per tant, ha de restaurar-lo abans de retornar.

El registre `ra` és **caller-saved**, ja que la instrucció que el pare empra per fer la crida (`jal`, RISC-V 3.11) escriu a `ra` l'adreça de retorn que el fill utilitzarà per retornar-li el control de flux. Aquesta escriptura implica la **pèrdua per sobreescriptura** de l'adreça de retorn que ell, el pare, ha d'utilitzar per tornar el control de flux a l'avi.

3.9.2 Pas de paràmetres i retorn de resultats (els registres `a0`–`a7`)

El pas de paràmetres i el retorn de resultats impliquen una comunicació entre la subrutina que crida i la subrutina cridada. En assemblador aquesta comunicació es fa dipositant paràmetres i resultats en llocs determinats per l'ABI.

RISC-V 3.17: RV32I ABI — Pas de paràmetres i retorn de resultats

- Els **paràmetres** de tipus **escalar** (inclosos els punters) es passen pels registres `a0`–`a7`, en ordre d'aparició, d'esquerra a dreta.
- Els **resultats** es retornen pels registres `a0` i `a1` (com a màxim dos resultats escalars). El segon registre `a1` s'usa habitualment per retornar un **codi d'error** o un segon valor de retorn.
- Si un paràmetre o resultat té menys de 32 bits (`char` o `short`), cal estendre'l a 32 bits, dins el registre assignat, fent extensió de signe o de zeros segons hagi estat declarat `signed` o `unsigned`.

Exemple 3.24: Traducció d'una funció amb paràmetres i resultat

C

```c
int suma2(int a, int b) {
    return a + b;
}

void main() {
    int x, y, z;   // en t0, t1, t2
    ...
    z = suma2(x, y);
    ...
}
```

RV32I

```
        ...
        mv      a0, t0          # passa x
        mv      a1, t1          # passa y
        jal     ra, suma2
        mv      t2, a0          # z = resultat
        ...

suma2:
        add     a0, a0, a1      # a + b
        ret
```

EC 3.6

A **EC**, els modes de pas de paràmetres són:

- **Pas per valor**: La funció rep una **còpia** del paràmetre. Qualsevol modificació que faci la funció afecta només la còpia, no l'original.
- **Pas per referència**: La funció rep una **referència** a la dada original, és a dir, la seva **adreça en memòria**. Les modificacions que faci la funció afecten directament la dada original.

Essencial 3.10

Cal distingir el **mecanisme físic** de pas del paràmetre (com es transmet la dada) de la **semàntica** (què representa la dada transmesa):

Mecanisme	Semàntica	Exemple a RV32I
Registre (`a0`–`a7`)	Valor	Paràmetre escalar (`int`, `char`)
Registre (`a0`–`a7`)	Referència	Paràmetre punter (`int *`)
Pila	Valor	Paràmetre escalar quan `a0`–`a7` ja estan ocupats

En **C**, el mecanisme és sempre per registre o pila, però la **semàntica** depèn del tipus: un **escalar** és pas per **valor**, i un **punter** té un efecte equivalent al pas per **referència**.

En **C** només existeix el **pas per valor**. Per tant, la funció cridada rep una còpia dels paràmetres i, en

conseqüència, les modificacions que hi faci no afectaran els valors de les dades originals.

Per aconseguir un efecte equivalent al **pas per referència**, el que es passa com a paràmetre és un **punter** a la dada original (per tant, aquesta ha d'estar en memòria). D'aquesta manera, les modificacions que faci la funció afectaran directament la dada original.

Exemple 3.25: Pas per valor vs. pas per referència: `printf()` i `scanf()`

Les funcions `printf` (impressió formatada de dades; `%d` per a enters) i `scanf` (lectura formatada de dades) de la biblioteca `stdio.h` il · lustren bé la diferència entre pas per valor i pas per referència en C:

C

```c
int x = 42;
printf("%d", x);    // pas per valor: printf rep una còpia de x (42)
scanf("%d", &x);    // pas per referència: scanf rep l'adreça de x
                    // i pot modificar-ne el contingut directament
```

Després de `scanf`, `x` pot tenir un valor diferent de `42`. Després de `printf`, `x` segueix valent `42`: `printf` no pot modificar-lo perquè només n'ha rebut una còpia.

Exemple 3.26: Pas de paràmetres per referència amb punters

C

```c
void sub(int *p) {
    *p = *p + 10;   // actualitza el contingut de l'adreça apuntada
}

void f() {
    int a, b;
    sub(&a);        // passa un punter a a
    sub(&b);        // passa un punter a b
}
```

Les variables **escalars** (globals i locals) es poden passar per **valor** o bé per **referència** en cas que es passi la seva adreça.

Els **vectors** i **matrius** només es poden passar per **referència**.

Exemple 3.27: C — Pas de paràmetres: vector i variables globals

C

```c
short x[10], y, z;
short funcB(short *vec, short n, int i);

void funcA() {
    int k;                      // k a t0
    ...
    z = funcB(x, y, k);
    ...
}
```

```c
short funcB(short *vec, short n, int i) {
    return vec[i] - n;
}
```

`funcA` és una subrutina **multinivell**: crida `funcB` i després continua executant-se fins al seu propi `ret`, de manera que haurà de preservar `ra` abans de la crida (Secció 3.9.4). S'omet aquí per centrar l'exemple en el pas de paràmetres.

RV32I

```
.data
x:      .space  20              # vector de 10 shorts (20 bytes)
y:      .half   0               # variable short (2 bytes)
z:      .half   0               # variable short (2 bytes)
...
.text
...
funcA:
        ...
        la      a0, x           # a0 <- @x (global)
        la      t2, y           # t2 <- @y (global)
        lh      a1, 0(t2)       # a1 <- M_h[@y] (amb extensió de signe)
        mv      a2, t0          # a2 <- k (local)
        jal     ra, funcB       # crida a funcB
        la      t2, z           # t2 <- @z (global)
        sh      a0, 0(t2)       # M_h[@z] <- a0 (retorn de funcB)
        ...
        ret
...
funcB:
        slli    a2, a2, 1       # desplaçament = i * 2  (sizeof(short) = 2)
        add     a0, a0, a2      # @vec[i] = vec + desplaçament
        lh      a0, 0(a0)       # vec[i] (amb ext. de signe)
        sub     a0, a0, a1      # resultat = vec[i] - n
        slli    a0, a0, 16      # retorn: short, amb extensió
        srai    a0, a0, 16      #   de signe
        ret
...
```

Exemple 3.28: Correspondència entre crides i capçaleres de funcions

Donades les declaracions de variables i les capçaleres de funcions del Codi 3.5 (en C no hi ha sobrecàrrega: les capçaleres són alternatives i no conviuen en un mateix programa):

Codi 3.5: Declaracions de variables i cinc capçaleres de funcions alternatives.

C

```c
short V1[10];
char V2[20];
int a;
int *p;

int f1(int i);
int f2(char c);
int f3(short *pi);
int f4(char vc[]);
```

```
int f5(int vi[]);
```

Pregunta: A quina capçalera correspon cada crida?

C

```
a = f?(V1);         // (1)
a = f?(p);          // (2)
a = f?(&V2[10]);    // (3)
a = f?(V2[10]);     // (4)
a = f?(*p);         // (5)
```

Solució:

Crida	Tipus del paràmetre real	Capçalera
`f?(V1)`	`short *` (nom de vector)	`int f3(short *pi)`
`f?(p)`	`int *`	`int f5(int vi[])`
`f?(&V2[10])`	`char *` (adreça d'element)	`int f4(char vc[])`
`f?(V2[10])`	`char` (valor d'element)	`int f2(char c)`
`f?(*p)`	`int` (desreferència)	`int f1(int i)`

Anàlisi:

- El nom d'un vector sense índex equival a un punter al primer element.
- `int vi[]` com a paràmetre formal és equivalent a `int *vi`.
- `&V2[10]` és l'adreça de l'element 10 de `V2`: un `char *`.
- `V2[10]` és el valor de l'element 10: un `char`.
- `*p` desreferencia `p` (de tipus `int *`) i dona un `int`.

3.9.3 Variables locals: el bloc d'activació (el registre `sp`)

Com s'ha vist a la Secció 2.4.3, les variables locals tenen visibilitat local i durada temporal: es creen cada cop que s'executa la subrutina i deixen d'existir quan retorna. Per això, l'espai de les que resideixen en memòria es reserva **dinàmicament**, en temps d'execució, i s'allibera en retornar.

RISC-V 3.18: RV32I ABI — El bloc d'activació — Punter de pila `sp`

Les subrutines reserven tot l'espai que necessitaran a l'inici de la seva execució i l'alliberen just abans de retornar el control de flux a la subrutina que crida. Aquest espai de memòria s'anomena **bloc d'activació** (*stack frame*, BA).
La reserva i l'alliberament d'espai de memòria per al BA es fan mitjançant el punter de pila `sp` (***stack pointer***, `x2`).
El BA **creix** cap a **adreces més baixes**.
Durant l'**execució** de la subrutina, l'`sp` **apunta al primer byte** del BA, el **cim de pila**.
Per fer la **reserva** d'espai, cal **decrementar** `sp` la mida del bloc i, per fer l'**alliberament**, restituir-lo al valor original **incrementant-lo** la mateixa quantitat.

RISC-V 3.19: RV32I ABI — Alineació de la pila

L'ABI estàndard d'RV32I exigeix que la mida del bloc d'activació i el valor de `sp` siguin múltiples de **16 bytes**.

EC 3.7

A **EC**, la mida del bloc d'activació i el valor de `sp` han de ser sempre **múltiples de 4 bytes**: és una relaxació de l'ABI (RISC-V 3.19) que simplifica els exemples i la solució dels problemes. Amb aquesta relaxació, `sp` només garanteix l'alineació a 4 bytes (el fons de pila de RARS, `0x7FFFEFFC`, ja només és múltiple de 4: RISC-V 3.21). N'hi ha prou per a les variables locals dels exemples i problemes d'EC, però no per a les de 8 bytes (`long long`, `double`), que l'ABI alinea a 8 (RISC-V 2.21).

Exemple 3.29: Evolució del bloc d'activació durant la crida i retorn d'una subrutina uninivell

C

```c
void funcB() { sentB; }
```

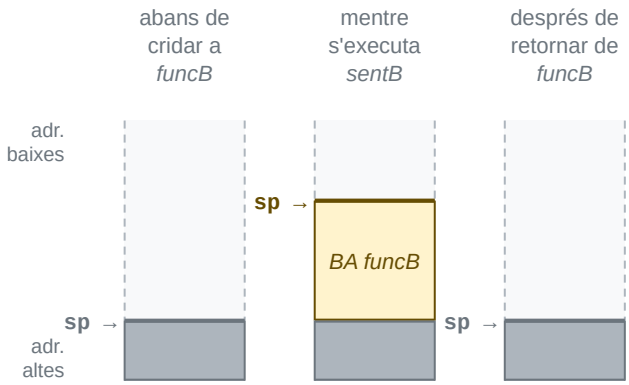


Figura 3.2: Evolució del registre `sp` durant la crida i el retorn de la subrutina fulla `funcB`: decreix en reservar el bloc d'activació de la subrutina i recupera el valor anterior en alliberar-lo.

RISC-V 3.20: RV32I ABI — Assignació de variables locals a registres o al bloc d'activació

Les variables locals s'assignen a **registres** o al **BA** seguint aquestes regles:

- Les variables **escalars** (enters, caràcters, punters) es guarden en **registres**. **Només** es guarden al BA si:
  - no hi ha prou registres disponibles, o
  - apareixen precedides de l'operador `&` (*adreça de*), per poder tenir una adreça de memòria.
- Les variables **estructurades** (vectors, matrius, etc.) es guarden **sempre** al **BA**.

El BA pot incloure:

- **Variables locals** de tipus estructurat i les escalars que no caben en registres o que requereixen adreça.
- **Registres segurs desats** (vegeu Secció 3.9.4), si la subrutina és multinivell.

La disposició dins el BA, de les adreces més baixes a les més altes (Figura 3.3):

Zona	Contingut
Primera (adreces baixes)	Variables locals, en ordre de declaració, alineades segons el seu tipus
Segona (adreces altes)	Registres segurs desats, alineats a múltiple de 4
Tercera (adreces més altes)	`ra`, alineat a múltiple de 4, si la subrutina és **multinivell**.

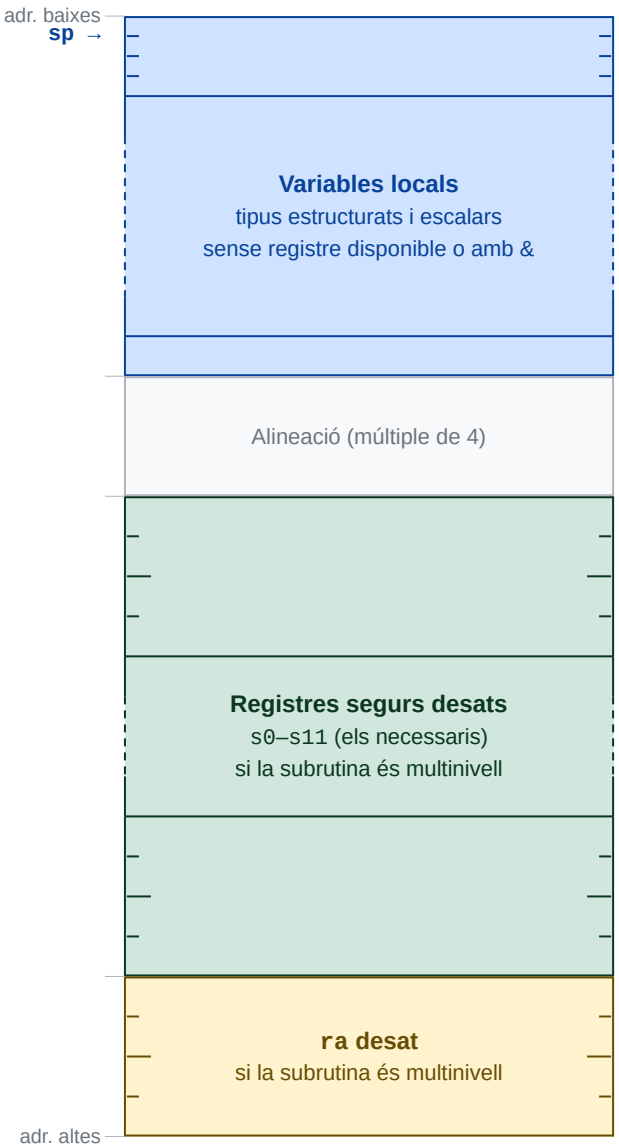


Figura 3.3: Estructura general del bloc d'activació: variables locals al cim de la pila i registres desats a continuació. El registre `sp` apunta al primer byte del bloc durant l'execució de la subrutina.

Exemple 3.30: Subrutina uninivell amb variables locals a la pila

C

```c
char func(int i) {          // i en a0
    char v[10];
    int w[10], k;           // k en t0
    ...
    return v[w[i] + k];
}
```

El bloc d'activació conté `v` (10 bytes) i `w` (40 bytes), en ordre textual, amb `k` en el registre `t0`. La mida total és 52 bytes (arrodonit a múltiple de 4).

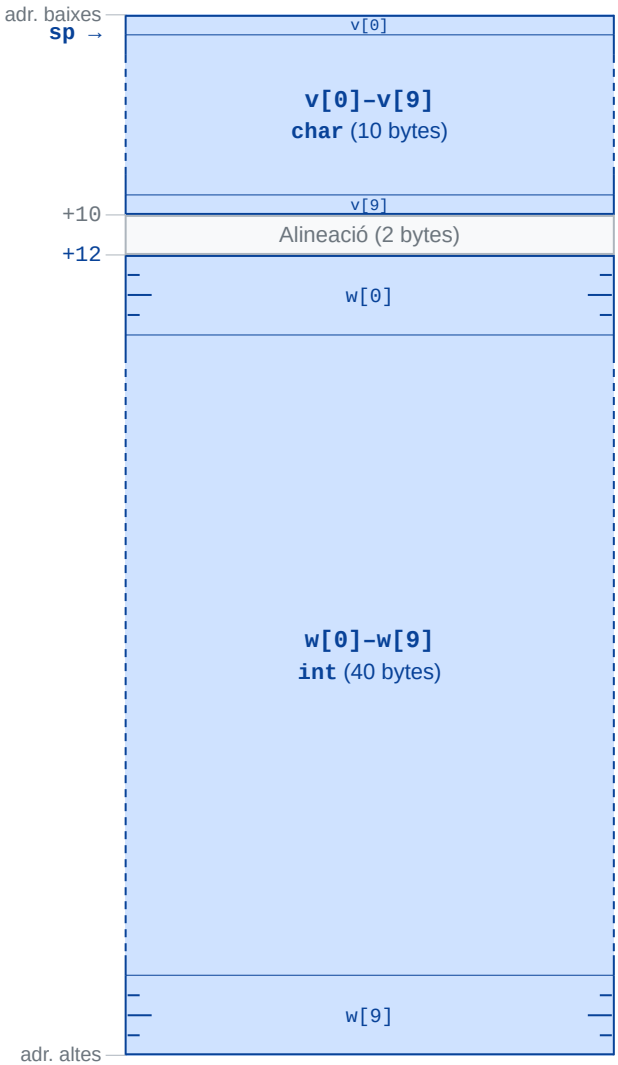


Figura 3.4: Bloc d'activació de la subrutina `func`: el vector `v` ocupa els bytes 0–9, l'alineació ocupa els bytes 10–11, i el vector `w` ocupa els bytes 12–51. El registre `sp` apunta al byte 0.

RV32I

```
func:
        addi    sp, sp, -52     # reserva espai: v(10) + alineació(2) + w(40)
        ...
```

```
# @w[i] = (sp+12) + i*4
slli    t4, a0, 2       # i*4
add     t4, t4, sp      # sp + i*4
lw      t4, 12(t4)      # w[i]
add     t4, t4, t0      # w[i] + k
# @v[w[i]+k] = sp + desplaçament
add     t5, sp, t4
lb      a0, 0(t5)       # retorna v[w[i]+k]
addi    sp, sp, 52      # allibera espai
ret
```

3.9.4 Subrutines multinivell: la pila del programa (els registres `s0`–`s11`)

Una subrutina multinivell en pot cridar una altra, que al seu torn en pot cridar una altra, i així successivament. En aquest cas, el bloc d'activació de cada subrutina s'apila sobre el de la subrutina que l'ha cridada.

La pila del programa és, doncs, una successió de blocs d'activació que creix i decreix a mesura que les subrutines es criden i retornen seguint la política d'«última en entrar, primera en sortir» (*last in, first out*, LIFO), la qual s'implementa mitjançant el punter de pila `sp`.

RISC-V 3.21: RARS — Fons de pila

A RARS, la regió de pila arriba fins a `0x7FFFFFFC`, però l'entorn reserva els 4 KiB que hi ha entre `0x7FFFEFFC` i aquesta adreça: el **fons de pila**, és a dir, el valor inicial d'`sp`, és `0x7FFFEFFC`.

Exemple 3.31: Evolució dels blocs d'activació durant la crida i retorn d'una subrutina multinivell la qual en crida una uninivell

C

```c
void funcA() { sentA1; funcB(); sentA2; }
void funcB() { sentB; }
```

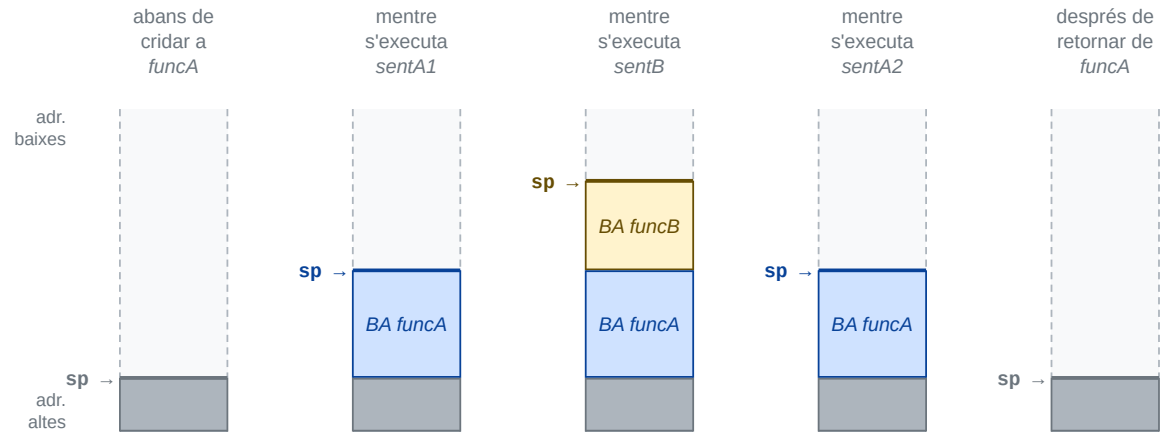


Figura 3.5: Evolució del registre `sp` durant la crida i el retorn de la subrutina multinivell `funcA` i de la subrutina fulla `funcB`: decreix en reservar el bloc d'activació de cada subrutina i recupera el valor anterior en alliberar-lo.

Una subrutina multinivell ha de preservar, com a mínim, el valor de `ra` abans de la crida a la subrutina cridada i restaurar-lo després:

RV32I

```
sub:
        ...
        sw      ra, 0(sp)       # preserva ra a la pila
        ...
        jal     ra, sub2        # crida a sub2 (per això sub és multinivell)
        ...
        lw      ra, 0(sp)       # restaurar ra abans de retornar
        ret
```

3.9.4.1 Preservació del context

El **context** d'execució d'una subrutina és el conjunt de totes les seves dades locals (incloent-hi paràmetres, adreça de retorn, punter de pila i càlculs intermedis), tant les que es guarden a la pila com en registres. En general, el context d'una subrutina no és visible, ni, per tant, accessible, a cap altra part del programa, excepte en el cas que la subrutina passi com a paràmetre a una altra subrutina un punter que apunta a una d'aquestes dades.

Els registres, però, són compartits per totes les subrutines. Per tant, hi ha d'haver algun mecanisme per garantir que el valor d'un registre que s'usa després d'una crida a una subrutina filla no es perdi per sobreescriptura. Aquest és el problema de preservar el context.

L'**ABI** d'RV32I resol aquest problema mitjançant una solució elegant i eficient, però que pot semblar contraintuïtiva a primera vista.

Aquesta solució es basa en la classificació dels registres en **temporals** i **segurs** (els *caller-saved* i els *callee-saved* de l'ABI) i en el repartiment de responsabilitats entre la subrutina que crida i la subrutina cridada per preservar-los (RISC-V 3.16). Mentre que els registres temporals poden ser modificats lliurement per qualsevol subrutina i, per tant, preservar-ne el valor és responsabilitat de la subrutina que crida, els registres segurs han de ser restaurats al seu valor original per la subrutina que els modifica, abans de retornar i, per tant, la responsabilitat de preservar-los és de la subrutina cridada.

Que el pare hagi de prendre mesures per preservar el seu context sembla raonable, ja que és qui té la necessitat de recuperar-lo després de la crida. El que pot semblar contraintuïtiu és que el fill hagi d'ajudar el pare a preservar el context d'aquest últim preservant-ne els valors dels registres segurs, tot i no tenir-ne cap dependència. No obstant això, aquesta solució és molt eficient perquè permet a **les subrutines** modificar lliurement els registres temporals sense haver de preocupar-se de preservar-los: només han de preservar **els registres segurs**, de manera que, per economia, només fan servir els registres que realment necessiten.

Dit de manera informal, el sistema no es basa en la idea que el pare s'hagi de protegir contra les modificacions del fill, sinó que es basa en el seu compromís de preservar els registres segurs del seu pare, és a dir, els de l'avi.

RISC-V 3.22: RV32I ABI — Registres *caller-saved* i *callee-saved*

De RISC-V 2.24 (vegeu RISC-V 3.16):

Categoria	Registres ABI	Qui els preserva
Temporals i arguments	`t0`–`t6`, `a0`–`a7`	*Caller*
Adreça de retorn	`ra`	*Caller*
Segurs	`s0`–`s11`, `sp`	*Callee*

Essencial 3.11

Resumint:

- Una subrutina, abans de retornar, ha de deixar els registres **segurs** (`s0`–`s11`) en el mateix estat que tenien quan ha estat invocada.
- `ra` és **caller-saved** (RISC-V 3.16), però una subrutina multinivell l'ha de desar abans de fer una crida i restaurar-lo abans de retornar.

3.9.4.2 Determinació dels registres segurs necessaris

Per determinar quins registres cal fer segurs, cal analitzar el codi i identificar les dades que compleixen **les dues condicions simultàniament**:

1. El seu valor ha estat generat per **última vegada abans** d'una crida.
2. El seu valor s'usa **després** d'aquesta mateixa crida.

A cada una d'aquestes dades se li assigna un registre segur (`s0`–`s11`), de manera que la subrutina filla els preservarà automàticament.

3.9.4.3 Estructura general d'una subrutina multinivell

La secció de codi d'una subrutina multinivell es divideix en tres parts:

- **Pròleg**: és el codi que s'executa a l'inici de la subrutina per generar el bloc d'activació.
- **Cos**: és el codi que implementa la funcionalitat de la subrutina.
- **Epíleg**: és el codi que s'executa al final de la subrutina, just abans de retornar el control de flux a la subrutina que crida, per restituir el context.

RV32I

```
sub:
        # pròleg
        addi    sp, sp, -BA_MIDA    # reserva d'espai per al BA
        sw      s0, 0(sp)           # desa els registres segurs (callee-saved)
        sw      s1, 4(sp)
        ...                         # tants com sigui necessari (màxim 12)
        sw      ra, BA_MIDA-4(sp)   # desa ra (si és multinivell)

        # cos
        ...

        # epíleg
        lw      s0, 0(sp)           # restauració de registres segurs
        lw      s1, 4(sp)
        ...                         # tants com sigui necessari (màxim 12)
        lw      ra, BA_MIDA-4(sp)   # restauració ra (si és multinivell)
        addi    sp, sp, BA_MIDA     # alliberació d'espai del BA

        ret
```

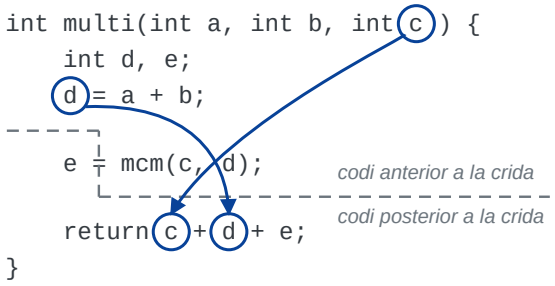
3.9.4.4 Exemples

Exemple 3.32: Subrutina multinivell 1

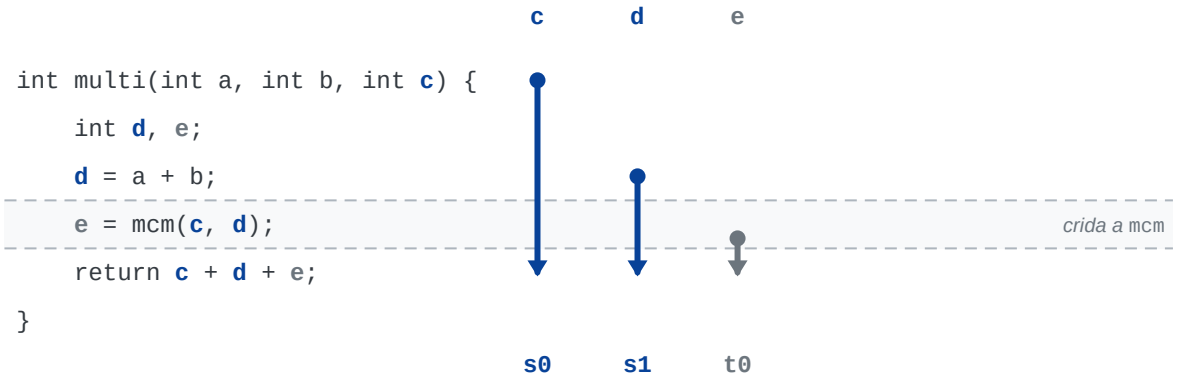
C

```c
int multi(int a, int b, int c) {
    int d, e;
    d = a + b;
    e = mcm(c, d);
    return c + d + e;
}
```

Pas 1. Registres segurs. Identificació de les dades que es fan servir **després** de la crida a `mcm` i el valor de les quals ha estat generat per **última** vegada **abans** d'aquesta crida: són `c` (paràmetre) i `d` (assignat a `a+b`). S'assignen els registres segurs `s0` i `s1`, respectivament. La variable `e` s'assigna al registre temporal `t0`, ja que s'escriu DESPRÉS de la crida.



(a) Amb fletxes, de l'escriptura a l'ús. Les línies de punts separen el codi anterior i el posterior a la crida.



(b) Amb barres de vida, de l'última escriptura a l'últim ús. La franja grisa és la crida.

Figura 3.6: Dependències de dades de la subrutina `multi`, en dues representacions. Les dades `c` i `d` travessen la crida a `mcm` i s'han de guardar en registres segurs; `e`, no.

Pas 2. Bloc d'activació. No hi ha variables locals a la pila (totes són escalars). Cal desar `s0`, `s1` i `ra`: 3 paraules = 12 bytes.

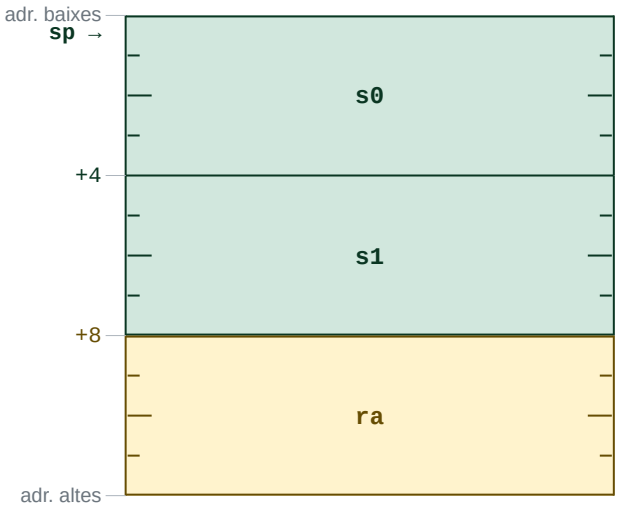


Figura 3.7: Bloc d'activació de la subrutina `multi`: 12 bytes amb `s0`, `s1` i `ra` desats als desplaçaments `+0`, `+4` i `+8`, respectivament.

Pas 3. Codi assemblador.

RV32I

```
multi:
```

```
        # pròleg
        addi    sp, sp, -12
        sw      s0, 0(sp)       # desa s0
        sw      s1, 4(sp)       # desa s1
        sw      ra, 8(sp)       # desa ra
        mv      s0, a2          # copia c a s0

        # d = a + b  → s1
        add     s1, a0, a1

        # e = mcm(c, d)
        mv      a0, s0          # passa c
        mv      a1, s1          # passa d
        jal     ra, mcm
        mv      t0, a0          # e = resultat de mcm

        # return c + d + e
        add     t1, s0, s1      # c + d
        add     a0, t1, t0      # c + d + e → a0

        # epíleg
        lw      s0, 0(sp)       # restaura s0
        lw      s1, 4(sp)       # restaura s1
        lw      ra, 8(sp)       # restaura ra
        addi    sp, sp, 12
        ret
```

Exemple 3.33: Subrutina multinivell 2

C

```c
int f(int m, int *n);
int g(char *y, char *z);

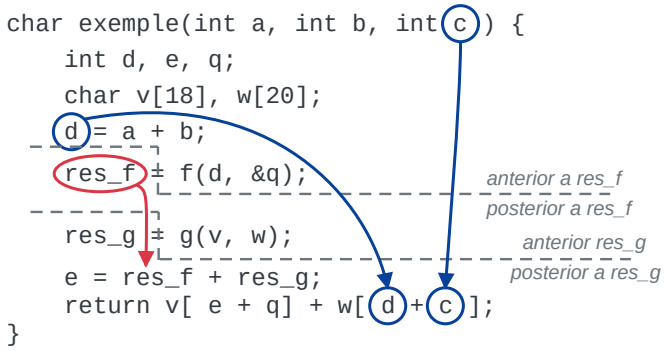
char exemple(int a, int b, int c) {
    int d, e, q;
    char v[18], w[20];
    d = a + b;
    e = f(d, &q) + g(v, w);
    return v[e + q] + w[d + c];
}
```

Pas 1. Registres segurs. Desglossant les dues crides:

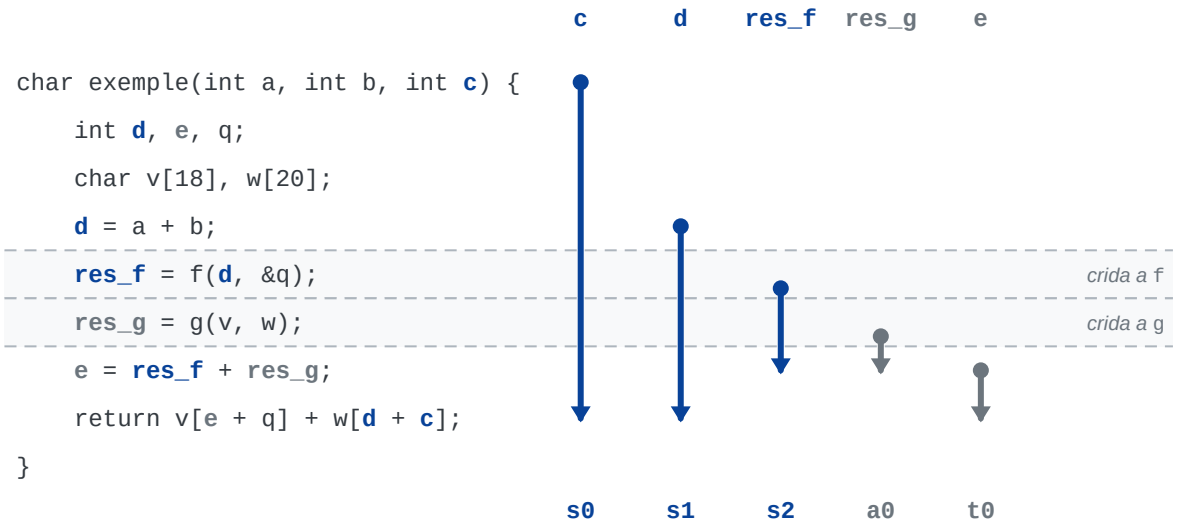
C

```c
res_f = f(d, &q);
res_g = g(v, w);
e = res_f + res_g;
```

Les dades que travessen fronteres de crides (escrites **abans**, usades **després**) són: `c`, `d` i `res_f`. S'assignen a `s0`, `s1` i `s2` respectivament. La variable `e` s'assigna a `t0`. Els paràmetres `a` i `b` resten a `a0` i `a1`.



(a) Amb fletxes, de l'escriptura a l'ús. Les línies de punts separen el codi anterior i el posterior a cada crida.



(b) Amb barres de vida, de l'última escriptura a l'últim ús. Les franges grises són les crides.

Figura 3.8: Dependències de dades de la subrutina `exemple`, en dues representacions. Les dades `c`, `d` i `res_f` travessen alguna crida i s'han de guardar en registres segurs, `s0`, `s1` i `s2` respectivament; `res_g` i `e`, no.

Anotació de registres sobre el codi:

C

```
char exemple(int a, int b, int c) { // a → a0, b → a1, c → s0
    int d,  e,  q;                  // d → s1, e → t0, q → pila
    char v[18], w[20];              // v → pila, w → pila
    ...
    res_f = f(d, &q);               // f(d, &q)  → s2
    res_g = g(v, w);                // g(v, w)   → a0 (temporal)
    e = res_f + res_g;              // e = s2 + a0
    ...
}
```

Pas 2. Bloc d'activació. Variables a la pila en ordre textual: `q` (4 bytes), `v` (18 bytes), `w` (20 bytes). Total variables: 42 bytes. Alineació fins a múltiple de 4: 2 bytes. Registres desats: `s0`, `s1`, `s2`, `ra` = 16 bytes. **Total: 60 bytes.**

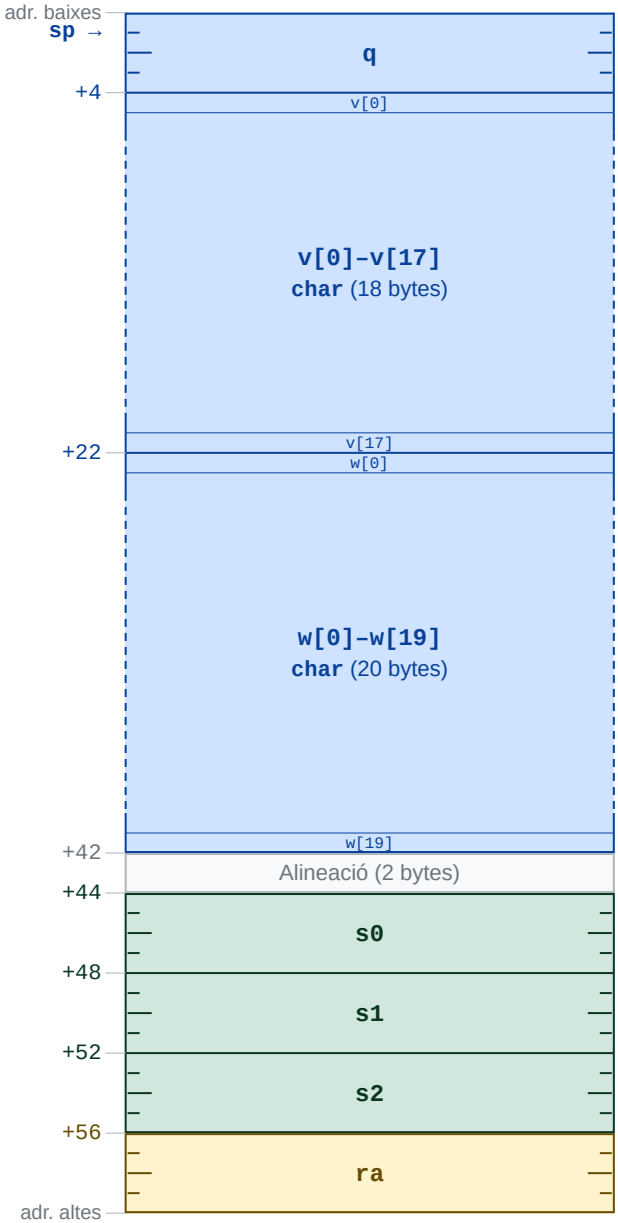


Figura 3.9: Bloc d'activació de la subrutina `exemple`: variables locals `q`, `v` i `w` als desplaçaments `+0`, `+4` i `+22`, alineació als bytes `+42`–`+43`, i registres desats `s0`, `s1`, `s2`, `ra` als desplaçaments `+44`, `+48`, `+52` i `+56`.

Pas 3. Codi assemblador.

RV32I

```
exemple:
        # pròleg
        addi    sp, sp, -60
        sw      s0, 44(sp)
        sw      s1, 48(sp)
        sw      s2, 52(sp)
        sw      ra, 56(sp)
        mv      s0, a2          # copia c a s0

        # d = a + b → s1
```

```
        add     s1, a0, a1

        # crida a f(d, &q)
        mv      a0, s1          # passa d
        mv      a1, sp          # passa &q (sp+0)
        jal     ra, f
        mv      s2, a0          # res_f → s2

        # crida a g(v, w)
        addi    a0, sp, 4       # passa @v[0]
        addi    a1, sp, 22      # passa @w[0]
        jal     ra, g
        add     t0, s2, a0      # e = res_f + res_g → t0

        # return v[e+q] + w[d+c]
        lw      t1, 0(sp)       # q
        add     t1, t0, t1      # e + q
        add     t1, t1, sp      # sp + (e+q)
        lb      t2, 4(t1)       # v[e+q]

        add     t1, s1, s0      # d + c
        add     t1, t1, sp      # sp + (d+c)
        lb      t3, 22(t1)      # w[d+c]

        add     a0, t2, t3      # v[e+q] + w[d+c]
        slli    a0, a0, 24      # retorn: char, amb extensió
        srai    a0, a0, 24      #   de signe

        # epíleg
        lw      s0, 44(sp)
        lw      s1, 48(sp)
        lw      s2, 52(sp)
        lw      ra, 56(sp)
        addi    sp, sp, 60
        ret
```

3.10 Compilació, assemblatge, enllaçament i càrrega

> ⚠ Aprofundiment 3.5: Compiler Explorer (Godbolt)
>
> *Compiler Explorer* (godbolt.org) permet introduir codi C directament al navegador i veure en temps real el codi assemblador generat per GCC, Clang i altres compiladors per a múltiples ISA, inclòs RISC-V (RV32 i RV64). És una eina útil per explorar l'efecte de les optimitzacions del compilador, comparar traduccions entre arquitectures i observar l'ús de pseudoinstruccions al codi generat.

En general, el procés pel qual un programa escrit en **C** es **converteix en** un programa **executable** consta de quatre passos:

1. **Compilació**.
2. **Assemblatge**.
3. **Enllaçament**.
4. **Càrrega**.

A la Secció 1.2.6 hi ha una introducció a aquest procés. En aquesta secció s'analitza amb més detall, posant èmfasi en com cada pas gestiona les **etiquetes** i les **adreces** del programa.

3.10.1 Compilació separada

En **C**, un **mòdul** (també anomenat unitat de traducció) és una unitat funcional de programació composta, generalment, per **un o més fitxers de codi font** (`.c`) i **un fitxer de capçalera** (`.h`), el qual conté les declaracions de les funcions i les dades que el mòdul vol fer accessibles a altres mòduls (tècnicament, la interfície). La directiva de preprocessament `#include` fa la inclusió textual dels fitxers de capçalera dins de la unitat de traducció, cosa que facilita la propagació de declaracions de tipus, prototips de funcions i macros necessàries per a la resolució de símbols.

Hi ha moltes situacions en què resulta convenient separar el codi i les dades en **mòduls**. Algunes d'aquestes situacions són:

- Fer possible la reutilització de codi (biblioteques de funcions).
- Estructurar racionalment projectes complexos.
- Permetre que equips independents de programadors treballin en paral · lel.
- Fer que només s'hagin de recompilar els mòduls modificats.

Els mòduls es compilen i assemblen **per separat**, generant fitxers objecte independents que més tard s'**enllaçaran** per crear l'executable (Figura 3.10).

> **Essencial 3.12**
>
> De vegades, per abús de llenguatge, s'anomena «compilar» tot el procés de traducció de codi font a executable, incloent-hi l'assemblatge i l'enllaçament. Cal, però, tenir present que, en realitat, cada un d'aquests passos és una fase diferent del procés de traducció, amb funcionalitats i sortides diferents.

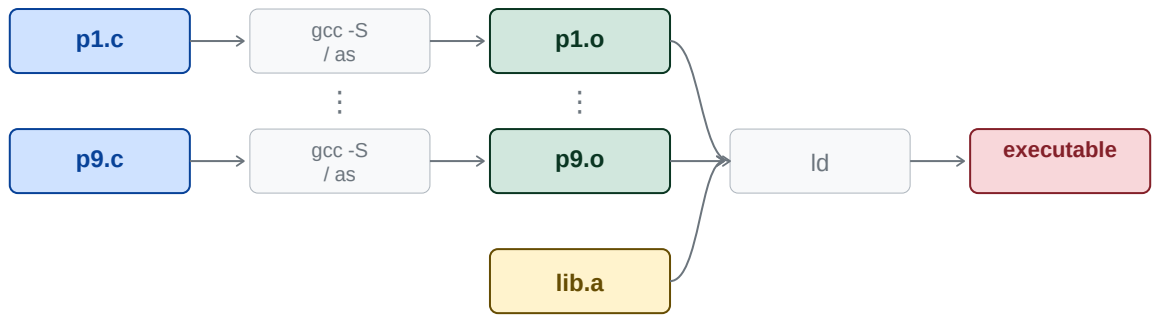


Figura 3.10: Flux de compilació separada: cada mòdul es compila i assembla independentment generant un fitxer objecte (`.o`), que l'enllaçador (`ld`) combina amb les biblioteques per produir l'executable.

> **RISC-V 3.23: RARS — El punt d'entrada i l'etiqueta `_start`**
>
> En **GNU/Linux**, el punt d'entrada d'un executable es marca amb l'etiqueta **`_start`**, declarada global amb `.globl _start`. El nom no és decoratiu: l'**enllaçador** l'ha de localitzar per escriure'l a la capçalera de l'executable, i el **carregador** del sistema operatiu hi transfereix el control en

engegar el procés. Hi ha, doncs, dos agents que llegeixen aquell nom, i per això el símbol ha de ser global i s'ha d'anomenar exactament així.
A **EC** es programa directament sobre el processador (EC 1.2), en un **entorn autònom** (*bare-metal*, Secció 1.2.2.2): no hi ha ni enllaçador ni carregador, i **cap agent no llegeix el nom del punt d'entrada**. L'execució comença sempre a la **primera instrucció del segment `.text`**, de manera que als programes d'aquests apunts el punt d'entrada **no porta etiqueta**.
No és una peculiaritat de l'assignatura, sinó la diferència entre un entorn **allotjat** (*hosted*) i un **autònom**: en un entorn autònom, la forma i el nom del punt d'entrada són **definits per la implementació**. És el mateix motiu pel qual el C d'aquests apunts declara `void main()` i no `int main` (Essencial 2.28): una etiqueta que ningú no resol seria tan fictícia com un `return` que ningú no recull.

3.10.2 Compilació

El **compilador** pren com a entrada el codi font en C i el tradueix a llenguatge assemblador (fitxer `.s`). Durant aquest procés aplica les regles de l'**ABI** per decidir quins registres usar per a cada variable, i genera les instruccions de l'**ISA** de destí.

El compilador no assigna adreces absolutes ni a les dades ni al codi, sinó que fa servir **etiquetes**. Aquestes etiquetes es resolen a adreces absolutes més tard. Durant l'assemblatge es resolen les que no depenen d'on es col · locarà el mòdul (els salts relatius al PC dins del mateix mòdul), i la resta (les adreces absolutes i les que referencien dades o codi d'altres mòduls) es deixen amb adreces provisionals que l'enllaçador corregirà quan combini tots els mòduls. D'aquesta manera, el compilador pot generar codi per a cada mòdul de manera independent, sense necessitat de conèixer els detalls dels altres mòduls amb els quals s'enllaçarà.

3.10.3 Assemblatge

L'**assemblador** pren com a entrada el **fitxer assemblador** `.s` i el tradueix a codi màquina en un **fitxer objecte** `.o`. Durant l'assemblatge es produeixen tres tasques fonamentals:

1. Expansió de macros i pseudoinstruccions.
2. Construcció de la taula de símbols.
3. Reubicació i referències externes.

3.10.3.1 Expansió de macros i pseudoinstruccions

L'assemblador expandeix les macros i pseudoinstruccions en les seqüències d'instruccions reals equivalents. Per exemple, `li t0, 0x0030D900` s'expandeix en `lui` + `addi`, i `ret` s'expandeix en `jalr zero, 0(ra)`.

3.10.3.2 Construcció de la taula de símbols

El llenguatge assemblador usa **etiquetes** per referir-se a adreces, tant al codi com a les dades. Una etiqueta es pot usar dins el fitxer fins i tot abans de ser definida (**referència anticipada**). Per gestionar-ho, l'assemblador fa **dues passades** sobre el fitxer:

- **Primera passada**: llegeix tot el codi font i construeix una **taula de símbols** que associa cada etiqueta amb la seva posició (desplaçament en bytes respecte a l'inici de la secció on està definida).

- **Segona passada**: codifica les instruccions, usant la taula de símbols per resoldre les etiquetes.

Cal notar que les adreces que assigna l'assemblador **no són les adreces definitives** en memòria, ja que l'assemblador no coneix el codi i les dades definits en altres mòduls. Les adreces definitives les assignarà l'enllaçador.

3.10.3.3 Reubicació i referències externes

Algunes instruccions necessiten l'adreça absoluta d'una etiqueta per ser codificades completament:

- Els **salts relatius al PC** (`beq`, `bne`, `blt`, `jal`, etc.) codifiquen un desplaçament relatiu al PC, que no depèn de l'adreça absoluta del codi. L'assemblador pot codificar-los completament si l'etiqueta de destinació és al mateix mòdul.
- La pseudoinstrucció `la` (`auipc` + `addi`) i els **salts indirectes** (`jalr`) que depenen d'una adreça externa necessiten una adreça absoluta que l'assemblador no coneix en temps d'assemblatge. L'assemblador hi escriu un valor provisional i anota la instrucció a la llista de reubicació perquè l'enllaçador la corregeixi.

L'assemblador genera per a cada fitxer objecte:

- Una **llista de reubicació**: posició de cada instrucció que conté una adreça provisional que caldrà corregir, el seu tipus, i el desplaçament provisional.
- Una **taula de símbols globals**: etiquetes declarades amb `.globl` i les seves adreces provisionals, visibles des d'altres mòduls.
- Una **llista de referències externes no resoltes**: instruccions que fan referència a etiquetes definides en altres mòduls.

> **RISC-V 3.24: Directives — `.globl`**
>
> Si una etiqueta ha de ser referenciada des d'un altre mòdul, cal declarar-la amb la directiva `.globl`, altrament només és visible localment dins el seu mòdul.

El fitxer objecte generat per l'assemblador conté els components següents:

> **RISC-V 3.25: RV32I ABI — Estructura d'un fitxer objecte**

Component	Contingut
Capçalera	Mida i posició dels restants components
Secció `.text`	Codi màquina de les subrutines del mòdul
Secció `.data`	Dades estàtiques definides al mòdul
Llista de reubicació	Posició, tipus i adreça provisional de cada instrucció a reubicar
Taula de símbols globals	Etiquetes globals i les seves adreces provisionals
Llista de referències externes	Instruccions que referencien etiquetes d'altres mòduls
Informació de depuració	Correspondència entre instruccions i línies del codi font

Exemple 3.34: Fitxers objecte de dos mòduls

Considerem dos mòduls A i B:

`Programa A`

```
.data
.globl X
X:      .word   -1
Y:      .word   -2

.text
f:      ...
        la      t0, Y       # adreça de Y (reubicació interna)
        ...
        jal     ra, g       # referència externa a g
        ...
```

`Programa B`

```
.data
Z:      .word   -3

.text
.globl g
g:      ...
        la      t1, X       # referència externa a X
        ...
```

Els fitxers objecte contindran:
Mòdul A:

- Reubicació: posició `+36` (text), tipus `la`, desplaçament `+4` (data).
- Símbols globals: `X` al desplaçament `+0` (data).
- Referències externes: posició `+60` (text), tipus `jal`, etiqueta `g`.

Mòdul B:

- Símbols globals: `g` al desplaçament `+16` (text).
- Referències externes: posició `+36` (text), tipus `la`, etiqueta `X`.

3.10.4 Enllaçament

L'**enllaçador** (*linker*) combina múltiples fitxers objecte i fitxers de biblioteca en un sol fitxer executable. Ho fa en quatre passos:

1. Resolució de símbols. L'enllaçador reuneix totes les taules de símbols globals i llistes de referències externes. S'assegura que cada referència externa té una definició, cercant-la primer als fitxers objecte i després als fitxers de biblioteca. Si no troba alguna definició, finalitza amb un error: `undefined reference to` `` `X' `` (GNU `ld`).

2. Combinació de seccions. Les seccions `.text` i `.data` de tots els mòduls es concatenen. Les seccions dels mòduls que no són el primer queden desplaçades. L'enllaçador calcula i anota el desplaçament sofert per les seccions de cada mòdul.

3. Reubicació d'adreces internes. Es corregeixen les instruccions de les llistes de reubicació. L'adreça definitiva s'obté sumant l'adreça absoluta inicial de la secció (`0x00400000` per a `.text` i `0x10010000` per a `.data` en RARS) amb el desplaçament del mòdul dins de la secció i el desplaçament provisional que l'assemblador hi havia codificat.

4. Resolució d'adreces externes. Es corregeixen les instruccions amb referències externes. L'adreça definitiva s'obté de manera anàloga, usant l'adreça del símbol global tal com consta a la taula de símbols.

Finalment, s'escriu al disc el **fitxer executable**, que té una estructura similar a un fitxer objecte però sense informació de reubicació ni referències externes no resoltes.

Exemple 3.35: Resultat de l'enllaçament dels mòduls A i B

Enllaçant els mòduls A i B de l'exemple anterior, el programa definitiu queda:

```
Inici codi:  0x00400000
Inici dades: 0x10010000

.text:
  codi A  (desplaçament +0  .. +79)   → adreces 0x00400000..0x0040004F
  codi B  (desplaçament +80 .. +139)  → adreces 0x00400050..0x0040008B

.data:
  X: .word -1   → adreça 0x10010000  (+0)
  Y: .word -2   → adreça 0x10010004  (+4)
  Z: .word -3   → adreça 0x10010008  (+8, desplaçat 8 bytes)

Reubicacions resoltes:
  @Y = 0x10010000 + 4           = 0x10010004
  @g = 0x00400000 + 80 + 16     = 0x00400060
  @X = 0x10010000 + 0           = 0x10010000
```

3.10.5 Càrrega en memòria

El **carregador** (*loader*) és un programa del sistema operatiu que llegeix el fitxer executable i el copia a la memòria principal per ser executat. En el cas d'UNIX/Linux segueix els passos següents:

1. Llegeix la capçalera del fitxer executable per determinar la mida de les seccions de codi i dades, i reserva memòria principal suficient.
2. Copia les instruccions i les dades del fitxer executable a la memòria.
3. Copia a la pila els paràmetres del programa principal expressats a la línia d'ordres (si n'hi ha).
4. Inicialitza els registres del processador, de manera que `sp` apunta al cim de la pila.
5. Salta a una rutina d'arrencada (*startup*) dins l'executable, la qual copia els paràmetres `argc` i `argv` de la pila als registres `a0` i `a1` i fa una crida a `main`.

Quan la subrutina `main` retorna, la rutina d'arrencada invoca la crida al sistema `exit`, que allibera tots els recursos assignats al programa (memòria, fitxers oberts, etc.).

Essencial 3.13

El mecanisme descrit en aquesta secció és el d'un **sistema real amb SO**. A **EC** no hi ha ni SO ni carregador (EC 1.2): els programes no reben `argc` ni `argv` i no tornen enlloc en acabar. El punt d'entrada és la **primera instrucció del segment `.text`**, sense cap etiqueta que l'anomeni (RISC-V 3.23), i la sortida es fa amb `li a7, 93` + `ecall` (RISC-V 2.19).
El `void main()` dels programes en C d'aquests apunts no contradiu res del que s'acaba de dir: és **notació** del programa que es tradueix a mà (Essencial 2.28), no un punt d'entrada que cap rutina d'arrencada no arriba a cridar.

3.10.6 GCC/Clang — El flux de generació complet

Aprofundiment 3.6: Flux de generació detallat amb GCC/Clang

El flux de generació d'un executable es pot desglossar en les etapes següents. A **EC** es fa servir el **compilador creuat** (*cross-compiler*) (`riscv32-unknown-elf-gcc`, en el cas de GCC), que s'executa a l'amfitrió (*host*, x86-64 o ARM), però genera codi per al *target* (RV32I), com s'ha explicat a la Secció 1.2.4.

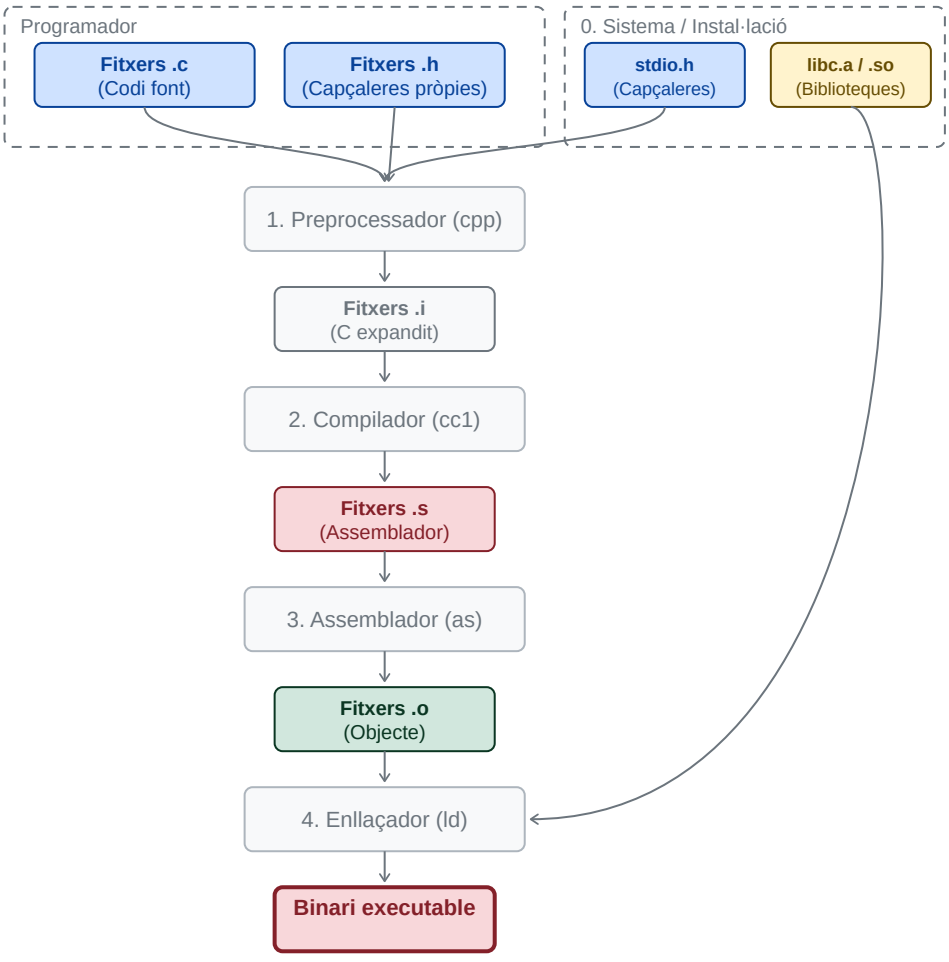


Figura 3.11: El flux de generació complet.

0. Sistema / Instal · lació: capçaleres del sistema (`stdio.h`, etc.) i biblioteques (`libc.a`, `.so`)

disponibles al sistema de compilació.

1. **Preprocessador (`cpp`) [agnòstic a ABI i ISA]**:

   - Expandeix macros i símbols.
   - Substitueix el contingut dels fitxers de capçalera en la posició de les directives `#include`.
   - Processa les directives de compilació condicional (`#ifdef`, `#ifndef`, `#if`).
   - Suprimeix tots els comentaris.
   - Resultat: un fitxer `.i` per cada fitxer `.c`.

2. **Compilador (`cc1`) [ABI + ISA]**:

   - Tradueix la lògica de C a assemblador RV32I.
   - Aplica les regles de l'ABI per decidir quins registres usar per a cada variable.
   - Genera la combinació d'instruccions més eficient per al repertori RV32I.
   - Resultat: un fitxer `.s` per cada fitxer `.i`.

3. **Assemblador (`as`) [ISA]**:

   - Expandeix pseudoinstruccions.
   - Construeix la taula de símbols (dues passades).
   - Genera el codi màquina amb adreces provisionals on calgui.
   - Resultat: un fitxer `.o` per cada fitxer `.s`, amb llistes de reubicació i símbols globals.

4. **Enllaçador (`ld`) [ISA + ABI]**:

   - Resol símbols globals i referències externes.
   - Combina les seccions de tots els mòduls.
   - Corregeix les adreces provisionals (reubicació).
   - Resultat: el binari executable, llest per ser carregat pel sistema operatiu o el simulador.

Tema 4: Aritmètica entera i matrius

4.1 Suma, resta i canvi de signe

Les instruccions de suma (`add`) i resta (`sub`) d'enters ja s'han introduït a RISC-V 2.12; aquí se'n reprèn l'ús per estudiar-ne el **desbordament** i per definir el **canvi de signe**.

RISC-V 4.1: Pseudoinstruccions — Canvi de signe: `neg`

El **canvi de signe** d'un registre s'obté amb la pseudoinstrucció `neg`:

Pseudoinstrucció	Operació	Expansió
`neg rd, rs`	$rd \leftarrow -rs$	`sub rd, zero, rs`

Gràcies a l'equivalència entre la resta i la suma amb el negatiu del subtrahend, el maquinari necessita un sol circuit per a la suma i la resta (Secció 1.6.3).

4.2 Desbordament en suma i resta d'enters

El **rang** de representació dels enters en Ca2 (Secció 1.5.1) amb n bits és $[-2^{n-1},\ 2^{n-1}-1]$ (Equació 1.3). Es produeix **desbordament** en una operació quan el resultat exacte no pertany a aquest rang; en aquest cas, el resultat calculat amb n bits no és correcte.

Aprofundiment 4.1: Desbordament i llenguatges de programació

La necessitat de detectar el desbordament depèn del llenguatge d'alt nivell. En llenguatges com Ada, quan la suma o la resta d'enters produeix desbordament, el programa ha de llançar una excepció, mentre que els desbordaments de naturals han de ser ignorats. En canvi, C no demana cap detecció: per als naturals, el resultat es trunca a mòdul 2^n, i per als enters, el desbordament és un **comportament indefinit** (l'estàndard no en fixa el resultat, i el compilador pot suposar que no es produeix).
Pel que fa al maquinari, alguns processadors ofereixen suport per detectar el desbordament (mitjançant indicadors o instruccions específiques). D'altres no, i en aquest cas la detecció és responsabilitat del programari (com en el cas d'RV32I).

Essencial 4.1

A **RV32I**, les operacions de suma i resta **ignoren el desbordament**: el resultat és simplement els 32 bits de menys pes de l'operació, independentment de si és correcte en el rang dels enters representables (Essencial 2.7).

4.2.1 Condicions de desbordament

Per a la suma $s = a + b$, es produeix desbordament si i només si **els dos operands són del mateix signe i el resultat és de signe diferent**.

Per a la diferència $d = a - b$, es produeix desbordament si i només si **la diferència és del mateix signe que el subtrahend b però de signe diferent que el minuend a.**

⚠ Aprofundiment 4.2: Demostració de les condicions de desbordament en Ca2

Sigui la suma exacta $s = a + b$, on a i b estan representats per les cadenes de bits $A = A_{n-1} \dots A_0$ i $B = B_{n-1} \dots B_0$. La suma exacta sempre es pot representar amb $n+1$ bits com $S = S_n\, S_{n-1} \dots S_0$.
Cas 1 — Signes oposats: si $a < 0$ i $b \geq 0$, llavors $a \leq a + b < b$, de manera que s sempre és representable amb n bits: no hi ha desbordament.
Cas 2 — Mateix signe: la suma exacta ha de ser del mateix signe que els operands, de manera que $S_n = A_{n-1} = B_{n-1}$. La cadena truncada $S' = S_{n-1} \dots S_0$ és la representació correcta de s si i només si $S_{n-1} = S_n$, és a dir, si $S_{n-1} = A_{n-1} = B_{n-1}$. Si $S_{n-1} \neq A_{n-1}$, el resultat truncat és incorrecte: hi ha desbordament.
Per a la diferència $d = a - b$, es pot aplicar la mateixa regla a l'expressió $a = d + b$.

4.2.2 Detecció del desbordament per maquinari

En un sumador de n bits amb **propagació del ròssec** (*ripple carry*), es produeix desbordament quan el **ròssec d'entrada** (*carry-in*) al bit de més pes difereix del ròssec de sortida:

$$v = c_{n-1} \oplus c_n \tag{4.1}$$

⚠ Aprofundiment 4.3: Detecció de desbordament per maquinari — el sumador amb propagació del ròssec

Un sumador de n bits es construeix amb una cadena de n **sumadors complets** (*full-adders*). Cada sumador complet rep el bit a_i, el bit b_i i el bit de ròssec d'entrada (*carry-in*) c_i, i produeix el bit de suma $s_i = a_i \oplus b_i \oplus c_i$ i el bit de ròssec de sortida (*carry-out*) $c_{i+1} = ((a_i \oplus b_i) \wedge c_i) \vee (a_i \wedge b_i)$. Internament, el sumador complet es construeix amb dos **semisumadors** (*half-adders*) i una porta OR (Figura 4.1): el primer semisumador dona $a_i \oplus b_i$ i $a_i \wedge b_i$, i el segon hi suma c_i.

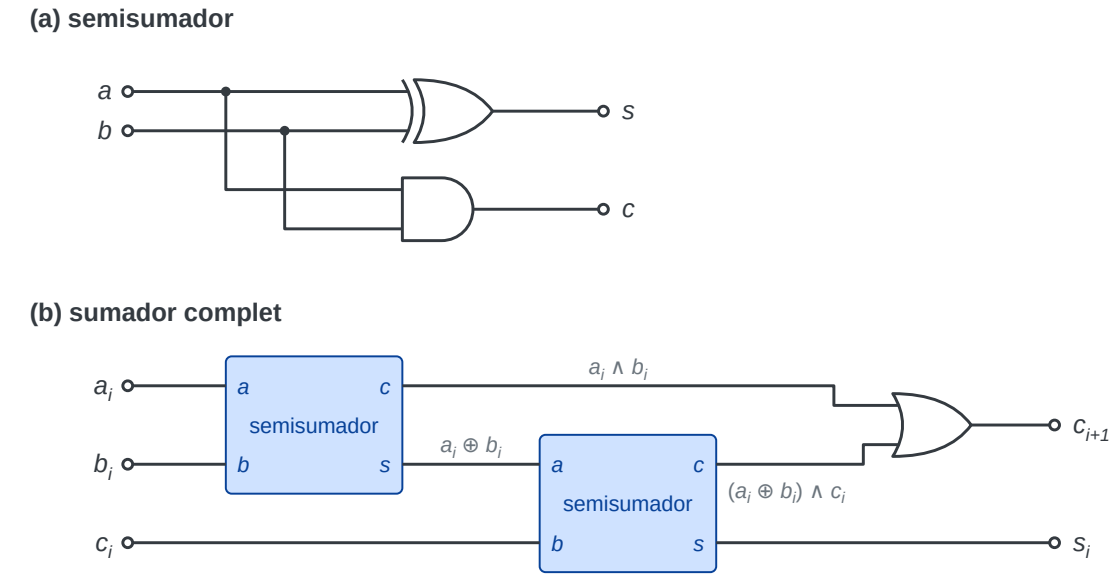


Figura 4.1: Semisumador i sumador complet: (a) el semisumador, amb una XOR per al bit de suma i una AND per al bit de ròssec; (b) el sumador complet, fet amb dos semisumadors i una porta OR.

Al darrer sumador complet (bit de més pes), el ròssec d'entrada és c_{n-1} i el de sortida és c_n. El desbordament es detecta amb una sola porta lògica afegida al darrer sumador: $v = c_{n-1} \oplus c_n$ (Figura 4.2).

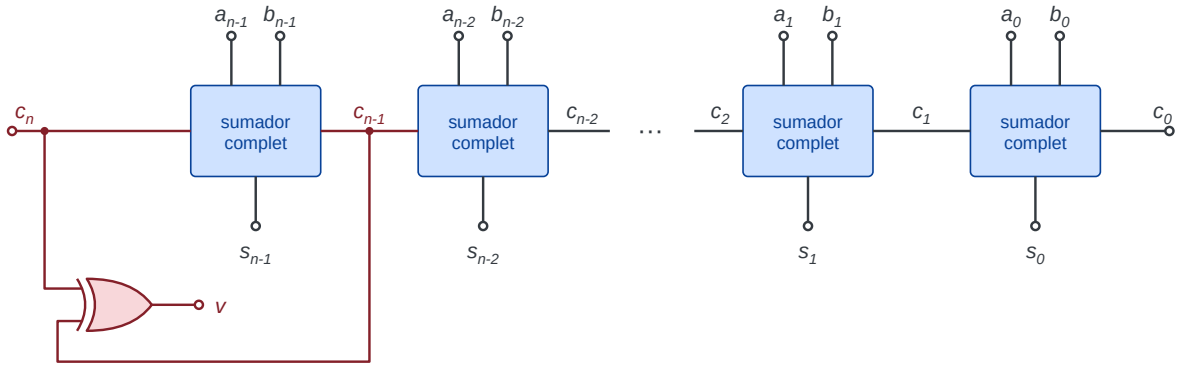


Figura 4.2: Sumador de n bits amb propagació del ròssec. El ròssec avança de dreta a esquerra, del bit de menys pes al de més pes, i una porta XOR sobre c_{n-1} i c_n dona el bit de desbordament v (Equació 4.1).

4.2.3 Detecció del desbordament per programari

Donada la suma $s = a + b$ d'enters de 32 bits, la condició de desbordament és:

$$v = \overline{(A_{31} \oplus B_{31})} \wedge (A_{31} \oplus S_{31}) \tag{4.2}$$

és a dir, hi ha desbordament si els operands tenen el mateix signe i el resultat té signe diferent.

Exemple 4.1: Càlcul de la condició de desbordament per programari

Suposem que els enters `a`, `b`, `s` estan a `t0`, `t1`, `t2`, i que s'ha executat `add t2, t0, t1` (la negació bit a bit es fa amb la pseudoinstrucció `not`, RISC-V 3.4):

RV32I

```
xor     t3, t0, t1      # A xor B  (bit 31: 0 si mateix signe)
not     t3, t3          # ~(A xor B)
xor     t4, t0, t2      # A xor S  (bit 31: 1 si signes diferents)
and     t3, t3, t4      # condició de desbordament als 32 bits
srli    t3, t3, 31      # desplaça bit 31 a la posició 0
                        # t3 = 1 si desbordament, 0 si no
```

Aprofundiment 4.4: Desbordament en naturals

Per als naturals, el rang de representació amb n bits és $[0,\ 2^n - 1]$, i es produeix desbordament quan el resultat exacte no pertany a aquest rang. La condició és més simple que en el cas dels enters: en un sumador amb propagació del ròssec, n'hi ha prou d'observar el ròssec de sortida del bit de més pes ($c_n = 1$). Per detectar-lo per programari, n'hi ha prou de comparar el resultat, com a natural (`bltu`, `sltu`), amb qualsevol dels dos operands: si la suma ha produït desbordament, el resultat serà necessàriament menor que cadascun dels operands (ja que s'ha «donat la volta» al rang).
En llenguatges com C, l'estàndard especifica que el desbordament de naturals produeix un truncament a mòdul 2^n (*wrapping around*). Per a tipus de dades de menys de 32 bits, el compilador ha de generar instruccions addicionals per truncar el resultat al rang corresponent.

4.3 Multiplicació entera

4.3.1 Algorisme

L'algorisme de multiplicació entera es basa en el mateix principi que la multiplicació manual: calcular productes parcials i acumular-los. Per a enters, es treballa amb els valors absoluts i s'ajusta el signe del resultat al final.

L'algorisme per calcular el producte $z = x \times y$ d'enters és:

1. Calcular els valors absoluts $|x|$ i $|y|$.
2. Multiplicar els valors absoluts (producte de naturals).
3. Si els operands tenen signes diferents, canviar el signe del resultat.

Per multiplicar dos naturals en base 2, s'obtenen tants **productes parcials** com bits té el multiplicador: el producte parcial i és el multiplicand desplaçat i posicions a l'esquerra si el bit i del multiplicador és 1, o zero si és 0. La suma de tots els productes parcials és el resultat.

Exemple 4.2: Algorisme de multiplicació de naturals en base 2

Multiplicar $x = 1010_2$ (10_{10}) per $y = 1101_2$ (13_{10}):

```
  1010    multiplicand (x)
× 1101    multiplicador (y)
```

```
    ------
      1010    producte parcial: x × 1 (bit 0)
     0000     producte parcial: x × 0 (bit 1)
    1010      producte parcial: x × 1 (bit 2)
   1010       producte parcial: x × 1 (bit 3)
---------
  10000010    resultat: 130₁₀ ✓
```

4.3.2 Circuit seqüencial de multiplicació

Per evitar guardar tots els productes parcials simultàniament, el circuit els va acumulant, a mesura que es calculen, en un registre acumulador de $2n$ bits, inicialitzat a zero. En total, el circuit (Figura 4.3) disposa de **tres registres**:

- **MD** ($2n$ bits): conté el multiplicand, inicialitzat amb x a la part baixa i zeros a la part alta. Es desplaça una posició a l'esquerra a cada iteració.
- **MR** (n bits): conté el multiplicador y. Es desplaça una posició a la dreta a cada iteració. El bit de menys pes s'usa per decidir si sumar o no.
- **P** ($2n$ bits): acumulador del producte. S'inicialitza a zero i s'hi suma MD si el bit de menys pes d'MR és 1.

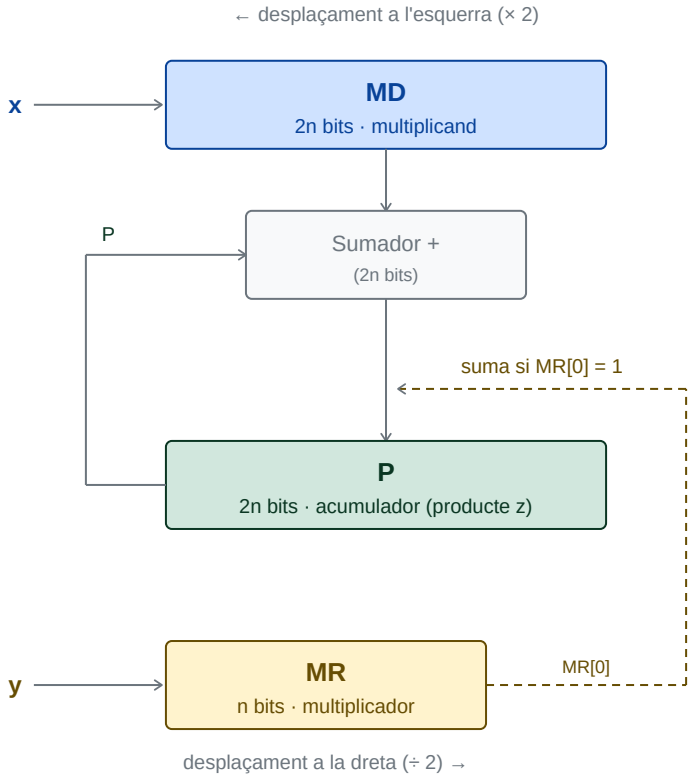


Figura 4.3: Circuit multiplicador seqüencial.

Algorisme 4.1. Multiplicació seqüencial de naturals de n bits, on x multiplicand, y multiplicador i z el resultat.

`Pseudocodi: multiplicació seqüencial`

```
// Inicialització
MD[n-1:0] = x;  MD[2n-1:n] = 0
MR = y
P = 0

per i = 1 fins a n:
    si MR[0] == 1: P = P + MD
    MD = MD << 1
    MR = MR >> 1

z = P
```

Exemple 4.3: Seguiment del circuit multiplicador (4 bits)

Multiplicar $x = 1010_2$ per $y = 1101_2$:

Iteració	P (acumulador)	MD (multiplicand)	MR (multiplicador)
inicial	`0000 0000`	`0000 1010`	`1101`
1	`0000 1010`	`0001 0100`	`0110`
2	`0000 1010`	`0010 1000`	`0011`
3	`0011 0010`	`0101 0000`	`0001`
4	`1000 0010`	`1010 0000`	`0000`

Resultat: $P = 10000010_2 = 130_{10}$ ✓

Per a $n = 32$, aquest circuit tarda 33 cicles per obtenir el producte (32 iteracions més la inicialització); en general, $n + 1$.

Aprofundiment 4.5: Circuit combinacional de multiplicació

Una alternativa al circuit seqüencial és un **circuit combinacional** que calcula tots els productes parcials en paral · lel i els suma amb una cadena de sumadors:

$$z = (x \cdot y_0) + ((x \cdot y_1) << 1) + ... + ((x \cdot y_{31}) << 31)$$

Això requereix 31 sumadors disposats en sèrie (de 33 a 63 bits). El retard total és aproximadament la suma dels retards dels 31 sumadors, cosa que no representa una millora substancial respecte del circuit seqüencial.
No obstant això, si es reestructuren les sumes en forma d'**arbre**, es pot reduir el retard a l'equivalent de 5 sumadors (aproximadament 6 vegades més ràpid que la versió en sèrie), ja que moltes sumes es fan en paral · lel.

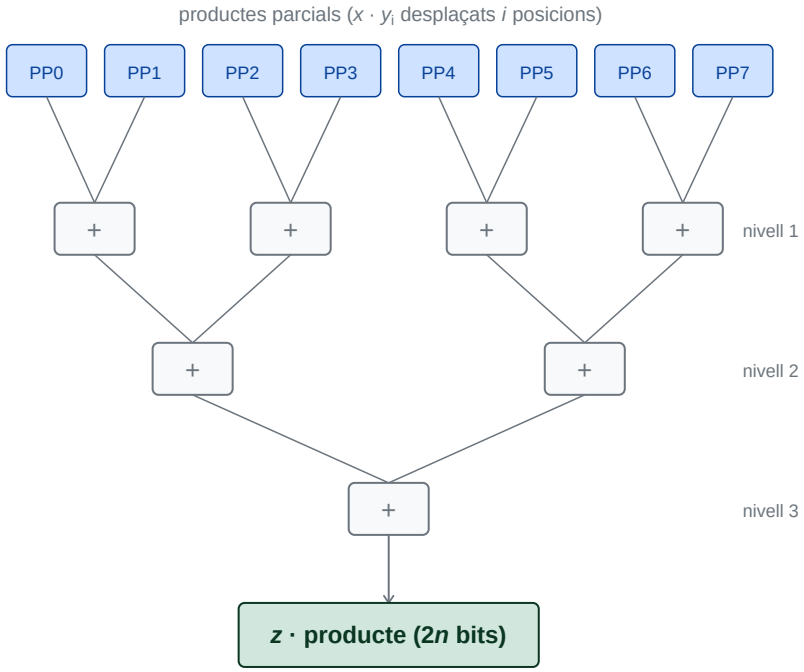


Figura 4.4: Circuit multiplicador combinacional en arbre.

4.3.3 Instruccions de multiplicació (extensió M)

Com s'ha vist a la Secció 3.1.4, `slli` permet multiplicar per potències de 2 de manera eficient; és la solució adequada per escalar índexs de vectors quan la mida de l'element és 1, 2, 4 o 8 bytes (EC 3.3). Per al cas general, però, cal una instrucció de multiplicació d'enters: per exemple, per accedir a un element qualsevol d'una matriu, cal calcular `i*NC`, on el nombre de columnes `NC` no té per què ser una potència de 2.

Aquí és on es manifesta la potència del disseny modular de RISC-V (Secció 2.1.3): l'extensió **M** (Secció 2.1.5) afegeix instruccions de multiplicació i divisió entera per maquinari. Per activar-la, cal afegir `m` al paràmetre `-march` del compilador: `-march=rv32im`.

RISC-V 4.2: RV32IM ISA — Instruccions de multiplicació entera

Mnemònic	Operands	Operació	Nom	Tipus
`mul`	`rd, rs1, rs2`	$rd \leftarrow (rs1 \times rs2)[31{:}0]$	*Multiply*	R
`mulh`	`rd, rs1, rs2`	$rd \leftarrow (rs1 \times rs2)[63{:}32]$ (enters)	*Multiply High*	R
`mulhu`	`rd, rs1, rs2`	$rd \leftarrow (rs1 \times rs2)[63{:}32]$ (naturals)	*Multiply High Unsigned*	R
`mulhsu`	`rd, rs1, rs2`	$rd \leftarrow (rs1 \times rs2)[63{:}32]$ ($rs1$ enter, $rs2$ natural)	*Multiply High Signed-Unsigned*	R

Exemple 4.4: Ús de `mul`

C

```c
int a = 5, b = 6, c;
c = a * b;                    // c = 30
```

Suposem que `a`, `b` i `c` ocupen `t0`, `t1` i `t2`:

RV32IM

```
        mul     t2, t0, t1    # c <- a * b (32 bits de menys pes)
```

Aprofundiment 4.6: Fusió de `mul` i `mulh`

`mul` i `mulh` (o `mulhu`/`mulhsu`) són instruccions independents: cadascuna calcula internament la multiplicació completa de 64 bits i en retorna una meitat. Per evitar fer el càlcul dues vegades, l'especificació de RISC-V recomana als compiladors que, quan calguin les dues meitats del resultat, posin primer `mulh` (o `mulhu`/`mulhsu`) i just després `mul`, amb els operands font en el mateix ordre i amb un registre destí de `mulh` diferent dels dos operands font; això permet al maquinari detectar el patró i fusionar-les en una sola multiplicació de 64 bits.

4.3.4 Desbordament en la multiplicació

El producte de dos enters de 32 bits pot tenir fins a 64 bits de resultat exacte. `mul` retorna només els **32 bits de menys pes**, que corresponen al resultat truncat. Si el resultat exacte no és representable amb 32 bits, **es produeix desbordament i el valor retornat per `mul` no és correcte**.

Essencial 4.2

El criteri vist per a la suma i la resta (Essencial 4.1) s'aplica a tota l'aritmètica entera de RISC-V, inclosa l'extensió M: el resultat es trunca silenciosament als 32 bits de menys pes, sense excepció ni indicador. La detecció, si cal, és responsabilitat del programari (Secció 4.2.3).

Per obtenir el resultat exacte de 64 bits cal combinar `mul` amb `mulh` (enters) o `mulhu` (naturals), que retornen els **32 bits de més pes**. A partir d'aquests es pot detectar el desbordament:

- En **naturals** (`mulhu`): hi ha desbordament si `mulhu` retorna un valor diferent de zero.
- En **enters** (`mulh`): hi ha desbordament si el resultat de `mulh` no és l'extensió de signe del resultat de `mul`.

Exemple 4.5: Detecció de desbordament en la multiplicació entera

C

```c
int a, b, s;
long long producte_exacte;
s = a * b;                      // RV32IM: mul en dona els 32 bits de menys pes
producte_exacte = (long long)a * b;
```

En RV32IM, per obtenir el producte exacte de 64 bits i detectar desbordament, suposant que `a`, `b` i `s` ocupen `t0`, `t1` i `t2`:

RV32IM

```
# a, b als registres t0, t1
mulh    t3, t0, t1            # t3 <- bits [63:32] del producte (enters)
mul     t2, t0, t1            # t2 <- bits [31:0] del producte
srai    t4, t2, 31            # t4 <- extensió de signe de t2
bne     t3, t4, desbordament  # desbordament si t3 != extensió de signe de t2
```

⚠ Aprofundiment 4.7: Per què `mul` no distingeix el signe dels operands?

A la Secció 4.3.1 s'ha presentat l'algorisme general per a enters: treballar amb els valors absoluts i ajustar el signe al final.
Estrictament, però, per a la **part baixa** del resultat aquest ajust no és necessari: el producte mòdul 2^n (els n bits de menys pes) **és idèntic** tant si els operands s'interpreten com a naturals com si s'interpreten com a enters en Ca2. Per això RV32M (l'extensió M d'RV32I) té una única instrucció `mul` per als bits [31:0], vàlida per a tots dos casos. En canvi, la **part alta** ([63:32]) sí que depèn de la interpretació dels operands, i per això existeixen `mulh`, `mulhu` i `mulhsu`.

4.4 Divisió entera

4.4.1 Algorisme

L'algorisme de divisió entera es basa en la divisió de naturals. Per a enters, es treballa amb els valors absoluts i s'ajusta el signe dels resultats al final.

L'algorisme per calcular el quocient $q = x \div y$ i el residu $r = x \bmod y$ és:

1. Calcular els valors absoluts $|x|$ i $|y|$.
2. Dividir els valors absoluts (divisió de naturals), obtenint q i r.
3. Si els operands tenen signes diferents, canviar el signe del quocient.
4. Si el dividend és negatiu, canviar el signe del residu.

La divisió del dividend x entre el divisor y consisteix a trobar el quocient q i el residu r naturals tals que $x = y \cdot q + r$, amb $0 \leq r < y$.

L'algorisme treballa els bits del quocient de més pes a menys pes. A cada pas i (de $n-1$ a 0), es compara el dividend parcial amb el divisor desplaçat i posicions a l'esquerra: si el dividend parcial és major o igual, el bit $q_i = 1$ i el divisor desplaçat es resta del dividend parcial; si no, $q_i = 0$ i el dividend parcial no canvia.

💡 Exemple 4.6: Algorisme de divisió de naturals en base 2

Dividir $x = 1011_2$ (11_{10}) entre $y = 0010_2$ (2_{10}):

Pas	Divisor desplaçat	Comparació	q_i	Residu parcial
inicial	—	—	—	`1011`
1 ($i = 3$)	`0010000`	`1011 ≥ 0010000`? No	0	`1011`

2 ($i = 2$)	`001000`	`1011` ≥ `001000`? Sí	1	`0011`
3 ($i = 1$)	`00100`	`0011` ≥ `00100`? No	0	`0011`
4 ($i = 0$)	`0010`	`0011` ≥ `0010`? Sí	1	`0001`

Quocient: $q = 0101_2 = 5_{10}$, residu: $r = 0001_2 = 1_{10}$ ✓

4.4.2 Circuit seqüencial de divisió

El circuit (Figura 4.5) implementa l'algorisme anterior amb **tres registres**:

- **R** ($2n$ bits): dividend/residu. S'inicialitza amb x a la part baixa i zeros a la part alta.
- **D** ($2n$ bits): divisor. S'inicialitza amb y a la part alta i zeros a la part baixa. Es desplaça una posició a la dreta a cada iteració.
- **Q** (n bits): quocient. S'inicialitza a zero.

Per evitar guardar el dividend original quan la comparació falla, s'utilitza l'**algorisme amb restauració**: es resta sempre el divisor del dividend parcial (la comparació es fa com una resta). Si el resultat és negatiu (bit de signe $R_{2n-1} = 1$), el bit de quocient és 0 i es restaura el dividend sumant-hi de nou el divisor; si el resultat és positiu o zero, el bit és 1.

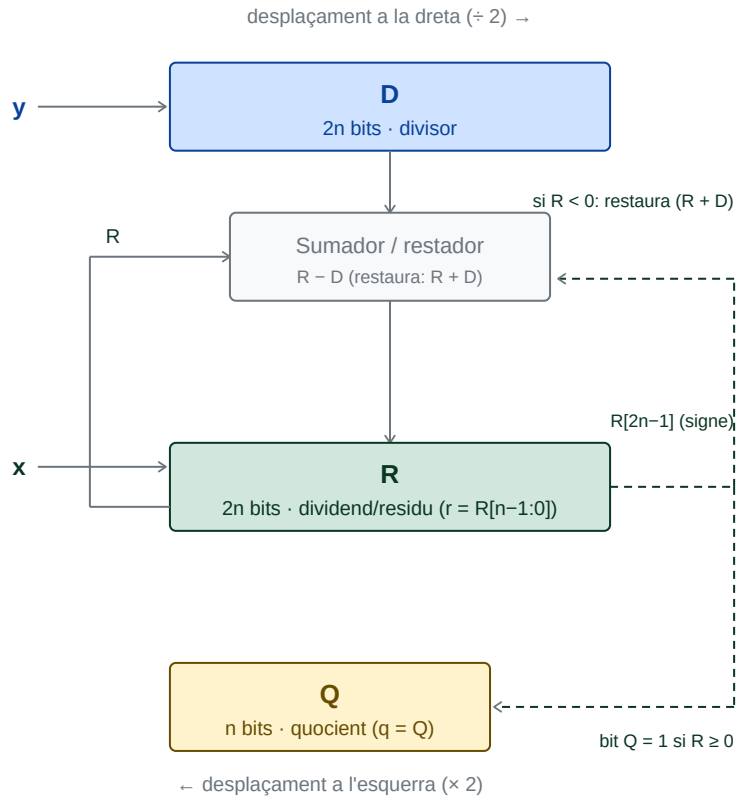


Figura 4.5: Circuit divisor seqüencial amb restauració.

Algorisme 4.2. Divisió seqüencial de naturals de n bits amb algorisme de restauració, on x és el dividend, y el divisor, q el quocient i r el residu.

`Pseudocodi: divisió amb restauració`

```
// Inicialització
R[n-1:0] = x;  R[2n-1:n] = 0
D[2n-1:n] = y;  D[n-1:0] = 0
Q = 0

per i = 1 fins a n:
    D = D >> 1
    R = R - D
    si R[2n-1] == 0:        // resultat positiu
        Q = (Q << 1) | 1
    sino:                   // resultat negatiu: restaurar
        R = R + D
        Q = Q << 1

q = Q          // quocient
r = R[n-1:0]   // residu
```

Exemple 4.7: Seguiment del circuit divisor (4 bits)

Dividir $x = 1011_2$ entre $y = 0010_2$:

Iteració	R (dividend/residu)	D (divisor)	Q (quocient)
inicial	`0000 1011`	`0010 0000`	`0000`
1	`0000 1011`	`0001 0000`	`0000`
2	`0000 0011`	`0000 1000`	`0001`
3	`0000 0011`	`0000 0100`	`0010`
4	`0000 0001`	`0000 0010`	`0101`

Quocient: $Q = 0101_2 = 5_{10}$, residu: $R[3:0] = 0001_2 = 1_{10}$ ✓

4.4.3 Instruccions de divisió entera a RV32IM

L'extensió M proporciona quatre instruccions de divisió entera:

RISC-V 4.3: RV32IM ISA — Instruccions de divisió entera

Mnemònic	Operands	Operació	Nom	Tipus
`div`	`rd, rs1, rs2`	$rd \leftarrow rs1 \div rs2$ (enters)	*Divide*	R
`divu`	`rd, rs1, rs2`	$rd \leftarrow rs1 \div rs2$ (naturals)	*Divide Unsigned*	R
`rem`	`rd, rs1, rs2`	$rd \leftarrow rs1 \bmod rs2$ (enters)	*Remainder*	R
`remu`	`rd, rs1, rs2`	$rd \leftarrow rs1 \bmod rs2$ (naturals)	*Remainder Unsigned*	R

`div` i `rem` operen amb enters. `divu` i `remu` operen amb naturals. `div` i `divu` escriuen el quocient a `rd`; `rem` i `remu` escriuen el residu.

Essencial 4.3

Les instruccions `div` i `rem` segueixen la **semàntica de C**: el residu, si no és zero, té el mateix signe que el dividend, i $|r| < |y|$. Aquesta és la mateixa semàntica que s'obté dividint els valors absoluts i ajustant el signe (algorisme de la Secció 4.4.1).
Això **difereix de la divisió euclidiana**, on el residu és sempre no negatiu. Vegeu la diferència amb `sra` a Essencial 3.2.

4.4.3.1 Casos especials

- **Divisor zero**: RISC-V no genera cap excepció, i l'especificació en fixa el resultat: `div` i `divu` retornen tots els bits a 1 (-1 en enters, $2^{32}-1$ en naturals), i `rem` i `remu` retornen el dividend.
- **Desbordament en enters**: l'únic cas possible és $x = -2^{31}$, $y = -1$, ja que el resultat exacte 2^{31} no és representable en Ca2 amb 32 bits. L'especificació també en fixa el resultat: `div` retorna -2^{31} i `rem` retorna 0.

Exemple 4.8: Traducció de divisió i mòdul d'enters

C

```c
int a, b, q, r;
q = a / b;
r = a % b;
```

RV32IM

```
        div     t2, t0, t1      # q = a / b
        rem     t3, t0, t1      # r = a % b
```

4.4.4 Divisió per potències de 2

Com s'ha vist a Essencial 3.2, la instrucció `sra` calcula el quocient de la divisió d'un enter per una potència de 2, però amb la **semàntica de la divisió euclidiana** (residu sempre no negatiu), que difereix de `div` quan el dividend és negatiu i la divisió no és exacta.

En C, l'operador `/` entre enters no es pot traduir amb `sra` sola: cal `div` o bé, si el divisor és una constant 2^k, sumar $2^k - 1$ al dividend quan és negatiu abans del desplaçament, que és el que fan els compiladors. `sra` sola només és vàlida si es pot garantir que el dividend és no negatiu.

4.5 Matrius

Amb `mul` disponible, ja és possible calcular adreces arbitràries dins d'estructures de dades multidimensionals. Una **matriu** és una agrupació multidimensional d'elements del mateix tipus, identificats per un índex per cada dimensió. Les matrius de dues dimensions es representen habitualment com una **taula d'`NF` files i `NC` columnes**:

$$\begin{pmatrix} \texttt{mat[0][0]} & \texttt{mat[0][1]} & \cdots & \texttt{mat[0][NC-1]} \\ \texttt{mat[1][0]} & \texttt{mat[1][1]} & \cdots & \texttt{mat[1][NC-1]} \\ \vdots & \vdots & \ddots & \vdots \\ \texttt{mat[NF-1][0]} & \texttt{mat[NF-1][1]} & \cdots & \texttt{mat[NF-1][NC-1]} \end{pmatrix}$$

4.5.1 Declaració i emmagatzematge

En C, una matriu global de `NF` files i `NC` columnes d'elements `int` es declara així:

C

```c
int mat1[NF][NC];
int mat2[2][3] = {{-1, 2, 0}, {1, -12, 4}};
```

Els elements s'emmagatzemen en posicions consecutives de memòria per files (Figura 4.6): primer tots els de la fila 0, després els de la fila 1, etc. Aquesta és una convenció arbitrària de C (en Fortran, les matrius s'emmagatzemen per columnes).

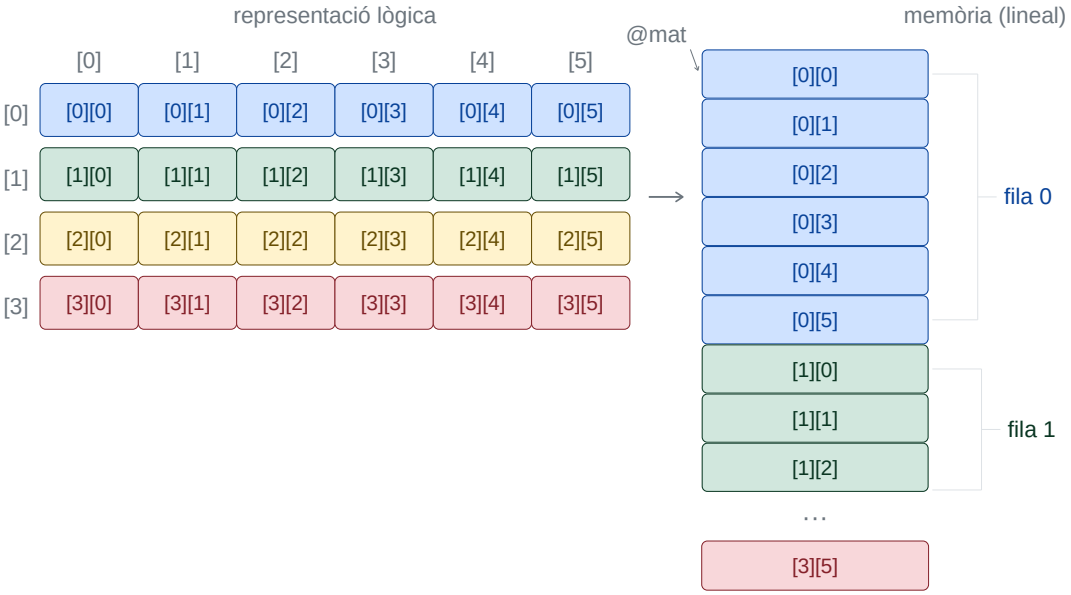


Figura 4.6: Emmagatzematge de `mat[4][6]` en memòria per files.

En RV32I, la declaració equivalent és:

RV32I

```
.data
mat1:   .space  NF*NC*4             # NF files × NC columnes × 4 bytes
mat2:   .word   -1, 2, 0, 1, -12, 4
```

> **!** EC 4.1
>
> A **EC**, es recomana definir les dimensions de la matriu (`NF`, `NC`) com a constants (`#define` en C, `.eqv` en assemblador) per facilitar la lectura del codi, el càlcul d'adreces i l'aplicació d'optimitzacions. Els blocs d'aquest tema usen aquestes constants dins d'expressions aritmètiques (`NF*NC*4`, `NC*4`, `mat + 3*NC*4`) per fer visible l'estructura del càlcul d'adreces. **RARS no les assembla**: vegeu EC 2.3 per a què se n'admet a la teoria i què cal escriure al laboratori.

4.5.2 Accés aleatori

L'accés aleatori és la capacitat d'accedir directament a qualsevol element d'una estructura de dades en temps constant $\mathcal{O}(1)$, independentment de la seva posició. Això és possible perquè tots els elements tenen una mida fixa T i estan disposats de forma contigua a la memòria (als `struct` de C, cada camp és a un desplaçament fix, conegut en compilar); per tant, l'adreça de qualsevol element es pot calcular explícitament mitjançant una fórmula aritmètica.

Presenten aquesta propietat les estructures de dades que s'emmagatzemen com a blocs compactes d'elements, com ara els vectors, les matrius i les estructures de dades complexes (`struct` en C). En canvi, les estructures de dades dinàmiques on els elements estan dispersos, com les llistes enllaçades, els arbres o els grafs, no permeten l'accés aleatori i requereixen un accés seqüencial.

Per accedir a l'element `mat[i][j]` d'una matriu d'elements de mida T, cal tenir en compte que la matriu s'emmagatzema per files: per arribar a la fila `i` cal saltar `i` files senceres de `NC` elements, i dins la fila cal avançar `j` elements més (Figura 4.7):

$$@\texttt{mat[i][j]} = @\texttt{mat[0][0]} + (i \cdot \texttt{NC} + j) \cdot T \tag{4.3}$$

L'Equació 4.3 és la generalització de l'Equació 2.1 (`NC > 1` vs. `NC = 1`).

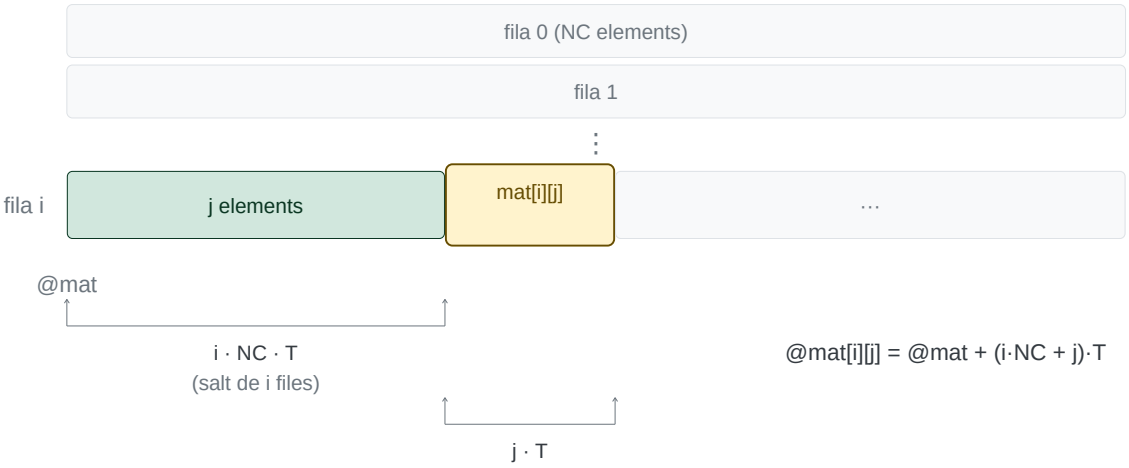


Figura 4.7: Càlcul de l'adreça de `mat[i][j]`.

! EC 4.2

A **EC**, s'usa `mul` en els càlculs d'adreces amb multiplicacions per constants que no són potències de 2. Els 32 bits de més pes s'ignoren perquè qualsevol adreça vàlida en RV32I cap en 32 bits.

Exemple 4.9: Accés a `mat[i][j]` — índexs variables

C

```c
int mat[NF][NC];

void func() {
    int i, j, k;
    k = mat[i][j];
}
```

Suposem que `i`, `j`, `k` ocupen `t0`, `t1`, `t2`:

RV32IM

```
        la      t3, mat             # invariant: @mat
        li      t4, NC              # invariant: NC
        mul     t4, t0, t4          # i*NC
        add     t4, t4, t1          # i*NC + j
        slli    t4, t4, 2           # (i*NC + j)*4
        add     t3, t3, t4          # @mat + (i*NC + j)*4
        lw      t2, 0(t3)           # k = mat[i][j]
```

Exemple 4.10: Accés a `mat[i][j]` — columna constant

Si la columna `j` és una constant (per exemple `j = 5`):

C

```c
k = mat[i][5];
```

El terme `5*T` és un invariant que el compilador pot incorporar directament a l'adreça base:

RV32IM

```
        la      t3, mat + 5*4       # invariant: @mat + 5*4
        li      t4, NC*4            # invariant: NC*4
        mul     t4, t0, t4          # i*NC*4
        add     t3, t3, t4          # @mat + i*NC*4 + 5*4
        lw      t2, 0(t3)           # k = mat[i][5]
```

Exemple 4.11: Accés a `mat[i][j]` — fila constant

Si la fila `i` és una constant (per exemple `i = 3`):

C

```c
k = mat[3][j];
```

El terme `3*NC*T` és un invariant incorporable a l'adreça base, i l'escalat de `j` es fa amb `slli`:

RV32I

```
        la      t3, mat + 3*NC*4    # invariant: @mat + 3*NC*4
        slli    t4, t1, 2           # j*4
        add     t3, t3, t4          # @mat + 3*NC*4 + j*4
        lw      t2, 0(t3)           # k = mat[3][j]
```

Exemple 4.12: Accés a `mat[i][j]` — fila i columna constants

Si tant la fila com la columna són constants, tota l'adreça és un invariant calculable en temps de compilació:

C

```
k = mat[3][5];
```

RV32I

```
        la      t3, mat + 3*NC*4 + 5*4   # adreça completa, invariant
        lw      t2, 0(t3)                 # k = mat[3][5]
```

Exemple 4.13: Pas d'una matriu per referència

Quan una matriu es passa com a paràmetre, el que es passa és la seva **adreça base**, igual que amb un vector (Exemple 3.27): el paràmetre és, de fet, un punter. L'única diferència respecte a l'accés a una matriu global és que l'adreça base **arriba en un registre d'argument** en lloc d'obtenir-se amb `la`. Com que només es passa l'adreça, la funció no coneix les dimensions de la matriu: per això el tipus del paràmetre ha de declarar el nombre de columnes (`int m[][NC]`) i no pot ser `int m[][]`: sense `NC` no es pot calcular el salt d'una fila a la següent, que és l'`i*NC` de la fórmula.

C

```
int f(int m[][NC], int i, int j) {
    return m[i][j];
}
```

Amb `@m` a `a0`, `i` a `a1` i `j` a `a2`, l'accés a `m[i][j]` aplica la mateixa fórmula que a Exemple 4.9, però prenent l'adreça base del registre d'argument `a0` en comptes de carregar-la amb `la`:

RV32IM

```
        li      t0, NC              # NC = nombre de columnes
        mul     t0, a1, t0          # i*NC
        add     t0, t0, a2          # i*NC + j
        slli    t0, t0, 2           # (i*NC + j)*4
        add     t0, a0, t0          # @m[i][j] = @m + (i*NC + j)*4
        lw      a0, 0(t0)           # return m[i][j]
```

4.6 Optimitzacions de bucle

Els bucles que recorren vectors i matrius sovint admeten optimitzacions que redueixen el nombre d'instruccions executades per iteració. Com s'ha vist a la Secció 2.8.5.1, l'extracció d'invariants és un prerequisit implícit que cal aplicar sempre abans de qualsevol altra optimització.

Les tres optimitzacions que es presenten a continuació s'apliquen de manera progressiva sobre el mateix bucle: cada una parteix del resultat de l'anterior.

Els blocs d'aquesta secció mantenen les expressions aritmètiques als operands (`li t0, NC*4`, `li t2, NC*2`) per fer visible l'*stride* de cada recorregut (EC 2.3).

4.6.1 Optimització #1 — Accés seqüencial

Quan un bucle recorre els elements d'un vector o matriu i la **distància en bytes** entre les adreces de dos elements consecutius del recorregut **és constant**, es pot aplicar l'**accés seqüencial**. Aquesta distància constant s'anomena ***stride***:

$$\text{stride} = @X_{i+1} - @X_i = \text{constant} \tag{4.4}$$

En lloc de calcular l'adreça de cada element a partir de l'adreça base i l'índex, es manté un **punter** que s'actualitza sumant-li l'*stride* a cada iteració:

$$@X_{i+1} = @X_i + \text{stride} \tag{4.5}$$

Això elimina el producte `i*T` del càlcul d'adreça i el substitueix per una suma constant.

Passos per aplicar l'accés seqüencial:

1. Calcular l'adreça del primer element del recorregut i inicialitzar un punter `p` amb aquesta adreça.
2. Calcular l'*stride* restant les adreces de dos elements consecutius.
3. A cada iteració, accedir a l'element desreferenciant `p`, i sumar-li l'*stride* per avançar al següent.

Exemple 4.14: Optimització #1 — Accés seqüencial aplicat a una fila de matriu

Considerem la funció que posa a zero una fila d'una matriu:

C

```c
void zero_fila(int mat[][NC], int fila) {
    int j;
    for (j = 0; j < NC; j++)
        mat[fila][j] = 0;
}
```

Versió d'accés aleatori sense optimitzar:

RV32IM

```
        # a0 = @mat, a1 = fila
        li      t0, NC*4            # invariant: NC*4
        mul     t1, a1, t0          # fila*NC*4
        add     t2, a0, t1          # invariant: @mat[fila][0]
        li      t0, NC              # invariant: NC (comptador)
        mv      t1, zero            # j = 0
```

```
for:
        bge     t1, t0, fifor       # [ctrl]
        slli    t3, t1, 2           # [ctrl] j*4
        add     t4, t2, t3          # [ctrl] @mat[fila][j]
        sw      zero, 0(t4)         #        mat[fila][j] = 0
        addi    t1, t1, 1           # [ctrl] j++
        j       for                 # [ctrl]
fifor:
        ret
```

La infraestructura de control ocupa **5 instruccions per iteració**.
Versió d'accés seqüencial optimitzada:
L'*stride* és: `@mat[fila][j+1]` − `@mat[fila][j]` = 4

RV32IM

```
        # a0 = @mat, a1 = fila
        li      t0, NC*4            # invariant: NC*4
        mul     t1, a1, t0          # fila*NC*4
        add     t3, a0, t1          # p <- @mat[fila][0]
        li      t0, NC              # invariant: NC (comptador)
        mv      t1, zero            # j = 0
for:
        bge     t1, t0, fifor       # [ctrl]
        sw      zero, 0(t3)         #        mat[fila][j] = 0
        addi    t3, t3, 4           # [ctrl] p += stride
        addi    t1, t1, 1           # [ctrl] j++
        j       for                 # [ctrl]
fifor:
        ret
```

La infraestructura de control queda en **4 instruccions per iteració**. Aplicant les optimitzacions #2 i #3 podrem reduir aquest nombre fins a 2.

4.6.2 Optimització #2 — Avaluació de la condició al final

En un bucle `while`, **la condició s'avalua abans** de cada iteració, cosa que implica un salt incondicional al final del cos per tornar a avaluar-la. Si es transforma en un `do-while`, **la condició s'avalua al final**, i s'elimina el salt incondicional.

No obstant això, cal garantir la semàntica del `while` original en el **cas de zero iteracions**: si el bucle pot no executar-se cap vegada, cal afegir una comprovació inicial abans d'entrar al `do-while`.

Pseudocodi: while → do-while

```
while (condició) {      →     if (condició)
    cos                 →         do {
}                       →             cos
                        →         } while (condició);
```

Exemple 4.15: Optimització #2 — Avaluació de la condició al final

Partint de la versió #1 amb accés seqüencial:

RV32IM

```
        # a0 = @mat, a1 = fila
        li      t0, NC*4
        mul     t1, a1, t0
        add     t3, a0, t1          # p <- @mat[fila][0]
        li      t0, NC
        mv      t1, zero            # j = 0
for:
        sw      zero, 0(t3)         #       mat[fila][j] = 0
        addi    t3, t3, 4           # [ctrl] p += stride
        addi    t1, t1, 1           # [ctrl] j++
        blt     t1, t0, for         # [ctrl] salta si j < NC
        ret
```

Com que `NC` és una constant positiva, el bucle sempre fa alguna iteració i no cal la comprovació inicial (sí que cal a Exemple 4.18, on el nombre de files és un paràmetre).
La infraestructura de control queda en **3 instruccions per iteració**. El salt incondicional `j` ha desaparegut, substituït pel salt condicional al final `blt`.

4.6.3 Optimització #3 — Eliminació de la variable d'inducció

Després d'aplicar la versió #1 amb accés seqüencial, el punter `p` recorre tots els elements del bucle. Si la variable d'inducció (Exemple 3.21) `j` ja no s'usa per a res més que controlar el bucle, es pot eliminar: en lloc de comparar `j` amb `NC`, es compara `p` directament amb l'adreça de l'element posterior al darrer.

Essencial 4.4

Com que es comparen adreces (naturals), cal usar `bgeu` / `bltu` en lloc de `bge` / `blt`.

Exemple 4.16: Optimització #3 — Eliminació de la variable d'inducció

Partint de la versió amb les optimitzacions #1 i #2:

RV32IM

```
        # a0 = @mat, a1 = fila
        li      t0, NC*4
        mul     t1, a1, t0
        add     t3, a0, t1          # p <- @mat[fila][0]
        add     t4, t3, t0          # t4 <- @mat[fila][NC] (sentinella)
for:
        sw      zero, 0(t3)         #       mat[fila][j] = 0
        addi    t3, t3, 4           # [ctrl] p += stride
        bltu    t3, t4, for         # [ctrl] salta si p < sentinella
        ret
```

La infraestructura de control queda en **2 instruccions per iteració**. La variable `j` ha desaparegut completament: el bucle es controla únicament amb el punter `t3` i la sentinella `t4`.

4.6.4 Recorreguts de matrius amb accés seqüencial

L'*stride* depèn del patró de recorregut, no de l'estructura de la matriu. Per a una matriu d'elements de mida T i `NC` columnes (Figura 4.8):

Recorregut	*Stride*
Una fila	T
Una columna	$\texttt{NC} \cdot T$
Diagonal principal	$(\texttt{NC} + 1) \cdot T$
Diagonal secundària	$(\texttt{NC} - 1) \cdot T$



Figura 4.8: Recorreguts de matrius, `mat[4][5]` a dalt i `mat[4][4]` a baix, i els *strides* corresponents.

Exemple 4.17: `sumacolumna` — accés aleatori

C

```c
short sumacolumna(short mat[][NC], int col, int nfiles) {
    int i;
    short suma = 0;
    for (i = 0; i < nfiles; i++)
        suma = suma + mat[i][col];
    return suma;
}
```

L'adreça de `mat[i][col]` és (amb $T = 2$):

$$@\texttt{mat[i][col]} = @\texttt{mat} + (i \cdot \texttt{NC} + \texttt{col}) \cdot 2$$

RV32IM

```
        # a0 = @mat, a1 = col, a2 = nfiles
        mv      t0, zero            # suma = 0
        mv      t1, zero            # i = 0
        li      t2, NC*2            # invariant: NC*2
        slli    t3, a1, 1           # invariant: col*2
        add     t3, t3, a0          # invariant: @mat + col*2
for:
        bge     t1, a2, fifor       # [ctrl]
        mul     t4, t1, t2          # [ctrl] i*NC*2
        add     t5, t3, t4          # [ctrl] @mat + col*2 + i*NC*2
        lh      t5, 0(t5)           #        mat[i][col]
        add     t0, t0, t5          #        suma += mat[i][col]
        addi    t1, t1, 1           # [ctrl] i++
        j       for                 # [ctrl]
fifor:
        slli    a0, t0, 16          # retorn: suma (short, amb
        srai    a0, a0, 16          #   extensió de signe)
        ret
```

Exemple 4.18: `sumacolumna` — accés seqüencial (optimitzacions #1, #2 i #3)

L'*stride* per recórrer una columna és $\texttt{NC} \cdot T = \texttt{NC} \cdot 2$.
Com que, en general, `NC` no té per què ser potència de 2, l'*stride* `NC*2` tampoc ho és: el desplaçament fins a la sentinella (`nfiles*NC*2`) s'ha de calcular amb `mul`, ja que no es pot obtenir amb `slli`.

`RV32IM`

```
        # a0 = @mat, a1 = col, a2 = nfiles
        mv      t0, zero            # suma = 0
        slli    t1, a1, 1           # col*2
        add     t3, a0, t1          # p <- @mat[0][col]
        li      t1, NC*2            # stride = NC*2
        mul     t2, a2, t1          # nfiles*NC*2
        add     t4, t3, t2          # sentinella: @mat[nfiles][col]
        bgeu    t3, t4, fifor       # comprovació inicial (0 iteracions)
for:
        lh      t5, 0(t3)           #        mat[i][col]
        add     t0, t0, t5          #        suma += mat[i][col]
        add     t3, t3, t1          # [ctrl] p += stride
        bltu    t3, t4, for         # [ctrl] salta si p < sentinella
fifor:
        slli    a0, t0, 16          # retorn: suma (short, amb
        srai    a0, a0, 16          #   extensió de signe)
        ret
```

Tema 5: Coma flotant

5.1 Representació en coma flotant

5.1.1 Notació científica normalitzada

La notació científica normalitzada és un sistema de representació de nombres reals que separa clarament la **magnitud** del nombre (els seus dígits significatius) de la seva **escala** (la posició de la coma). Un nombre v s'escriu en la forma:

$$v = m \times 10^e$$

on l'**exponent** e és un enter i la **mantissa** (*significand*) m satisfà $1 \leq |m| < 10$, amb almenys un dígit fraccionari. Aquesta restricció s'anomena **condició de normalització** i garanteix una representació única per a cada nombre no nul.

La notació científica normalitzada presenta dos avantatges respecte de la representació decimal ordinària. D'una banda, és més compacta: $2{,}34 \times 10^8$ és més llegible que $234\,000\,000$. D'altra banda, fa explícit el nombre de dígits significatius: $2{,}34 \times 10^8$ indica clarament tres dígits significatius, mentre que $234\,000\,000$ és ambigu (poden ser tres, o bé el nombre pot ser exacte fins a les unitats). El terme **coma flotant** fa referència precisament a aquest mecanisme: la coma «es desplaça» cap a l'esquerra o cap a la dreta en funció del valor de l'exponent.

En dispositius electrònics, els tres caràcters `×10` es substitueixen habitualment per la lletra `E`, que separa mantissa i exponent: 2,34E8.

Exemple 5.1: Conversió a notació científica normalitzada

Nombre	Notació científica normalitzada
$234\,000\,000$	$2{,}34 \times 10^8$
$0{,}000\,468$	$4{,}68 \times 10^{-4}$
$-0{,}0234 \times 10^{10}$	$-2{,}34 \times 10^8$

5.1.2 Representació binària

Els nombres en coma flotant en base 2 segueixen el mateix principi que la notació científica normalitzada, amb la base 2 en lloc de la base 10. La condició de normalització imposa que la part entera de la mantissa sigui exactament 1, ja que en base 2 l'únic dígit no nul és l'1:

$$v = \pm 1, \underbrace{b_{-1} b_{-2} \ldots b_{-(p-1)}}_{\text{fracció } F} \times 2^e$$

on l'exponent e és un enter, F és la part fraccionària de la mantissa i p és la **precisió**, el nombre de bits significatius de la mantissa.

> **Essencial 5.1: El bit ocult**
>
> Com que la condició de normalització garanteix que el bit a l'esquerra de la coma és sempre `1`, **aquest bit no s'emmagatzema** a la codificació: és el **bit ocult** (***hidden bit***). El maquinari el torna a afegir implícitament en qualsevol operació.
> Cal tenir-lo en compte en tots els càlculs: si el camp de fracció F té $p-1$ bits emmagatzemats, la mantissa completa té p bits significatius: els $p-1$ bits de F més el bit ocult.
> Els únics casos en què el bit ocult no s'aplica són els nombres subnormals i el zero, a més de les codificacions especials $\pm\infty$ i *not a number* (NaN), que no representen cap mantissa (Secció 5.1.6).

5.1.3 L'estàndard IEEE 754

Fins als anys 80, cada fabricant de processadors adoptava el seu propi format de coma flotant, amb decisions diverses sobre el nombre de bits de mantissa i exponent, i fins i tot sobre la base de la potència. La incompatibilitat entre sistemes dificultava l'intercanvi de dades i la portabilitat del programari. El 1985, l'*Institute of Electrical and Electronics Engineers* (IEEE) va publicar l'**estàndard IEEE 754**, que no sols defineix el format de representació sinó també les operacions aritmètiques, l'arrodoniment i el tractament de les excepcions. L'estàndard s'ha revisat i ampliat posteriorment (la darrera revisió és de 2019).

> **RISC-V 5.1: RV32F ISA — Formats IEEE 754 de precisió simple i doble**

	Precisió simple	Precisió doble (extensió D; no s'estudia a EC)
Tipus C	`float`	`double`
Mida total	32 bits	64 bits
Signe (S)	1 bit	1 bit
Exponent (E)	8 bits	11 bits
Fracció (F)	23 bits	52 bits
Precisió (amb bit ocult)	24 bits	53 bits
Biaix de l'exponent	127	1023
Rang de l'exponent e	$[-126,\ +127]$	$[-1022,\ +1023]$

> **EC 5.1**
>
> A **EC**, s'usa exclusivament el **format de precisió simple**. Totes les operacions, exemples i problemes fan referència a aquest format llevat que s'indiqui explícitament el contrari.

Els tres camps de bits del format de precisió simple s'organitzen de la manera següent (Figura 5.1):

$$\underbrace{S}_{1}\ \underbrace{E_7 E_6 \dots E_0}_{8}\ \underbrace{F_{22} F_{21} \dots F_0}_{23}$$

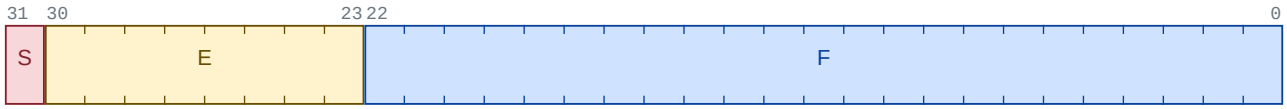


Figura 5.1: Disposició dels camps S, E i F en el format IEEE 754 de precisió simple (32 bits).

5.1.3.1 Signe (S)

El bit de signe S indica el signe del nombre: $S = 0$ per a valors positius i $S = 1$ per a valors negatius.

5.1.3.2 Fracció (F)

El camp de fracció F emmagatzema els $p - 1 = 23$ bits de la part fraccionària de la mantissa normalitzada. La mantissa completa s'obté afegint el bit ocult: $1{,}F$.

5.1.3.3 Exponent (E)

L'exponent és un enter (pot ser negatiu), però no es codifica en Ca2. S'emmagatzema com a **exponent esbiaixat** (***biased exponent***), amb **biaix 127**, és a dir, en excés a 127 en el sentit de la Secció 1.5.2: el valor emmagatzemat E_u és el natural que satisfà

$$e = E_u - 127 \qquad \Longleftrightarrow \qquad E_u = e + 127 \tag{5.1}$$

on e és el valor enter de l'exponent. Per exemple, $e = 0 \Rightarrow E_u = 127 = 0111\,1111_2$; $e = 1 \Rightarrow E_u = 128 = 1000\,0000_2$.

> Essencial 5.2: Per què l'exponent es codifica esbiaixat?
>
> La representació esbiaixada (Secció 1.5.2 i EC 1.3) garanteix que l'ordre numèric dels valors explícits E_u (interpretats com a naturals) coincideix amb l'ordre dels exponents enters e. Això permet **comparar les magnituds de dos nombres en coma flotant amb el mateix circuit que compara naturals** (si tots dos són positius, l'ordre coincideix amb el dels valors; si tots dos són negatius, és l'invers), amb dues condicions:
>
> 1. Els bits de l'exponent ocupen la part alta del nombre, immediatament després del signe, i els de la fracció ocupen la part baixa. Així, a exponents iguals guanya el de fracció major; entre exponents diferents guanya el de l'exponent major.
> 2. L'exponent es codifica esbiaixat (no en Ca2), de manera que la correspondència entre representacions i valors és monòtona creixent.

El valor complet representat per una codificació normalitzada és:

$$v = (-1)^S \times 1{,}F \times 2^{E_u - 127} \tag{5.2}$$

La Figura 5.2 mostra la correspondència entre l'exponent emmagatzemat E_u i el valor real $e = E_u - 127$, i n'indica les zones reservades per a subnormals i codificacions especials.

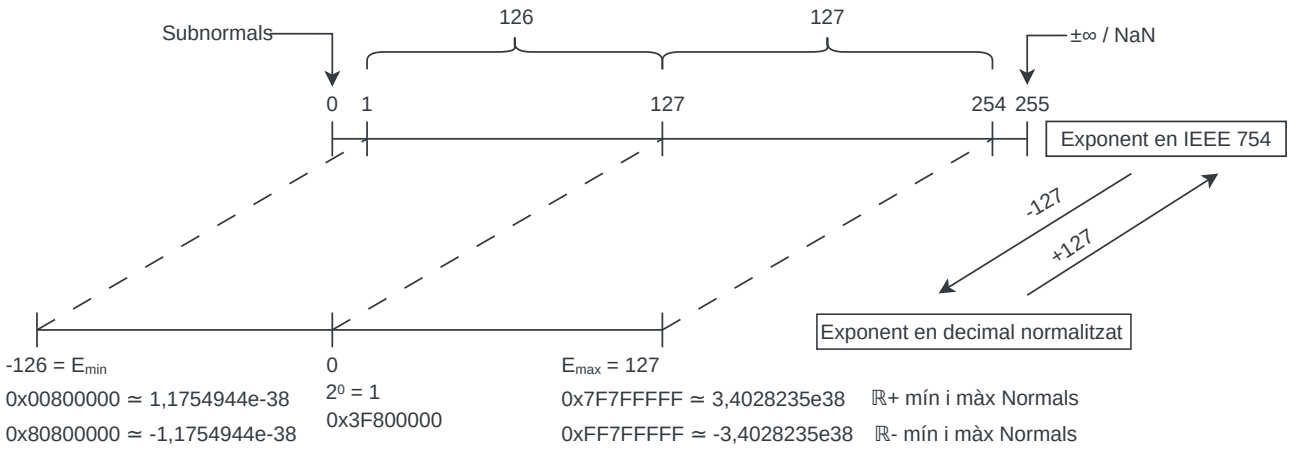


Figura 5.2: Correspondència entre l'exponent emmagatzemat E_u (0–255) i l'exponent real $e = E_u - 127$: $E_u = 0$ reservat per a subnormals i ± 0; $E_u = 255$ per a $\pm\infty$ i NaN.

Exemple 5.2: Conversió a IEEE 754 de precisió simple

Codifica el nombre $v = -1029{,}68$ en format IEEE 754 de precisió simple i calcula l'error de representació.

a) Part entera a binari (divisions successives):

$$1029 = 1000\,0000\,0101_2 \quad (11 \text{ bits})$$

b) Part fraccionària a binari (multiplicacions successives per 2 i extracció de la part entera de cada resultat com a bit de la representació binària de la part fraccionària):

$$0{,}68 \to 1{,}36 \to 0{,}72 \to 1{,}44 \to 0{,}88 \to 1{,}76 \to 1{,}52 \to 1{,}04 \to 0{,}08 \to \ldots$$

$$(0 \quad \to 1 \quad \to 0 \quad \to 1 \quad \to 0 \quad \to 1 \quad \to 1 \quad \to 1 \quad \to 0 \quad \to \ldots)$$

$$0{,}16 \to 0{,}32 \to 0{,}64 \to 1{,}28 \to 0{,}56 \to 1{,}12 \to 0{,}24 \to 0{,}48 \to 0{,}96 \to 1{,}92 \to \ldots$$

$$(0 \quad \to 0 \quad \to 0 \quad \to 1 \quad \to 0 \quad \to 1 \quad \to 0 \quad \to 0 \quad \to 0 \quad \to 1 \quad \to \ldots)$$

$0{,}68 = 0{,}1010\,1110\,0001\,0100\,01\ldots_2$. Per tant:

$$v = -1000\,0000\,101{,}1010\,1110\,0001\,0100\,01\ldots_2$$

c) Normalitzar la mantissa:

$$v = -1{,}0000\,0001\,0110\,1011\,1000\,0101\,0001\ldots_2 \times 2^{10}$$

d) Arrodonir a 24 bits significatius (23 bits de fracció + bit ocult).
Els bits a descartar comencen per `10001...`; com que el primer bit descartat és `1` i n'hi ha d'altres a la dreta, cal arrodonir **al valor següent** (sumar 1 al bit de menys pes conservat). El criteri general d'arrodoniment es formalitza a Secció 5.1.7; en aquest cas:

$$\begin{array}{r} 1{,}0000\,0001\,0110\,1011\,1000\,010 \\ + \qquad\qquad\qquad\qquad\qquad\qquad 1 \\ \hline 1{,}0000\,0001\,0110\,1011\,1000\,011 \end{array}$$

e) Codificar l'exponent en forma esbiaixada (biaix 127):

$$E_u = 10 + 127 = 137 = 1000\,1001_2$$

f) Ajuntar S, E, F (descartant el bit ocult de la fracció):

$$V = \underbrace{1}_{S}\ \underbrace{1000\,1001}_{E}\ \underbrace{0000\,0001\,0110\,1011\,1000\,011}_{F}$$

$$V = 1100\,0100\,1000\,0000\,1011\,0101\,1100\,0011_2 = \texttt{0xC480B5C3}$$

Solució:

$$V = \texttt{0xC480B5C3}$$

g) Error de representació:
L'error de representació és la diferència entre el valor realment emmagatzemat i el valor exacte original. Com que el valor exacte té infinits bits en binari, és més senzill calcular aquesta diferència en decimal. Desfent la normalització de la mantissa arrodonida (coma 10 posicions a la dreta), el valor emmagatzemat (en valor absolut) és:

$$1{,}0000\,0001\,0110\,1011\,1000\,011_2 \times 2^{10} = 1000\,0000\,101{,}1010\,1110\,0001\,1_2$$

La part entera ($1000\,0000\,101_2 = 1029$) no ha canviat amb l'arrodoniment: l'error és, doncs, la diferència entre les parts fraccionàries. La part fraccionària emmagatzemada val:

$$0{,}1010\,1110\,0001\,1_2 = 1\,0101\,1100\,0011_2 \times 2^{-13} = \frac{5571}{8192}$$

$$\varepsilon = \left|\frac{5571}{8192} - 0{,}68\right| = \frac{5571 - 0{,}68 \times 8192}{8192} = \frac{0{,}44}{8192} \approx 5{,}371 \times 10^{-5}$$

Solució:

$$\varepsilon = 5{,}371 \times 10^{-5}$$

Exemple 5.3: Conversió des d'IEEE 754 de precisió simple a base 10

Converteix a decimal el nombre `0x45814140`.
a) Descompondre en camps S, E, F:

$$\texttt{0x45814140} = 0 \mid 1000\,1011 \mid 0000\,0010\,1000\,0010\,1000\,000_2$$

b) Descodificar signe, exponent i mantissa:

- $S = 0$ (positiu).
- $E_u = 1000\,1011_2 = 139 \Rightarrow e = 139 - 127 = 12$.
- Mantissa (amb bit ocult): $1{,}0000\,0010\,1000\,0010\,1000\,000_2$.

$$v = +1{,}0000\,0010\,1000\,0010\,1000\,000_2 \times 2^{12}$$

c) Eliminar la potència (desplaçar la coma 12 posicions a la dreta):

$$v = 1\,0000\,0010\,1000{,}0010\,1000\,000_2$$

d) Convertir la part entera:

$$1\,0000\,0010\,1000_2 = 2^{12} + 2^5 + 2^3 = 4096 + 32 + 8 = 4136$$

e) Convertir la part fraccionària:

$$0{,}0010\,1000\,000_2 = 101_2 \times 2^{-5} = 5/32 = 0{,}15625$$

Solució:

$$v = 4136{,}15625$$

5.1.4 Rang i precisió

El **rang** de la representació és l'interval $[-v_{max},\ +v_{max}]$, on v_{max} és el nombre positiu més gran representable. S'obté amb la màxima fracció (tots els bits a 1) i el màxim exponent ($e = 127$ en precisió simple):

$$v_{max} = (2 - 2^{-23}) \times 2^{127} \approx 3{,}4 \times 10^{38}$$

Si el resultat d'una operació té un exponent $e > 127$, no és representable i es produeix una excepció de desbordament (*overflow*).

La **precisió** d'una representació és el seu nombre de bits significatius. En precisió simple, la mantissa normalitzada té 24 bits significatius (23 bits de fracció emmagatzemats més el bit ocult), de manera que la precisió és de 24 bits.

Essencial 5.3: Compromís rang–precisió

Donat un nombre fix de bits, augmentar la precisió (més bits de fracció) redueix el rang (menys bits d'exponent) i viceversa. El format de precisió simple (8 bits d'exponent i 23 de fracció) és un dels compromisos que fixa l'estàndard IEEE 754; el de precisió doble (11 i 52 bits, RISC-V 5.1) n'és un altre.

5.1.5 Error de representació i subdesbordament

El resultat exacte v d'una operació pot no ser representable amb un nombre finit de bits de fracció. En aquest cas cal aproximar-lo per un dels dos valors representables més pròxims $v_0 < v < v_1$. L'**error absolut** de representació es defineix com $\varepsilon = |v - \tilde{v}|$, on $\tilde{v} \in \{v_0, v_1\}$ és el valor que tria l'arrodoniment, i l'**error relatiu**, com $\eta = \varepsilon \,/\, |v|$.

Per a un nombre normalitzat qualsevol de la forma $v_0 = m \times 2^e$, el valor representable següent és $v_1 = (m + 2^{-23}) \times 2^e$, de manera que l'error absolut màxim és:

$$\varepsilon_{max} < 2^{e-23}$$

Dividint per v_0 i tenint en compte que $m \geq 1{,}0$, l'error relatiu queda acotat uniformement (Aprofundiment 5.1):

$$\eta_{max} < 2^{-23} = 1 \text{ ULP}$$

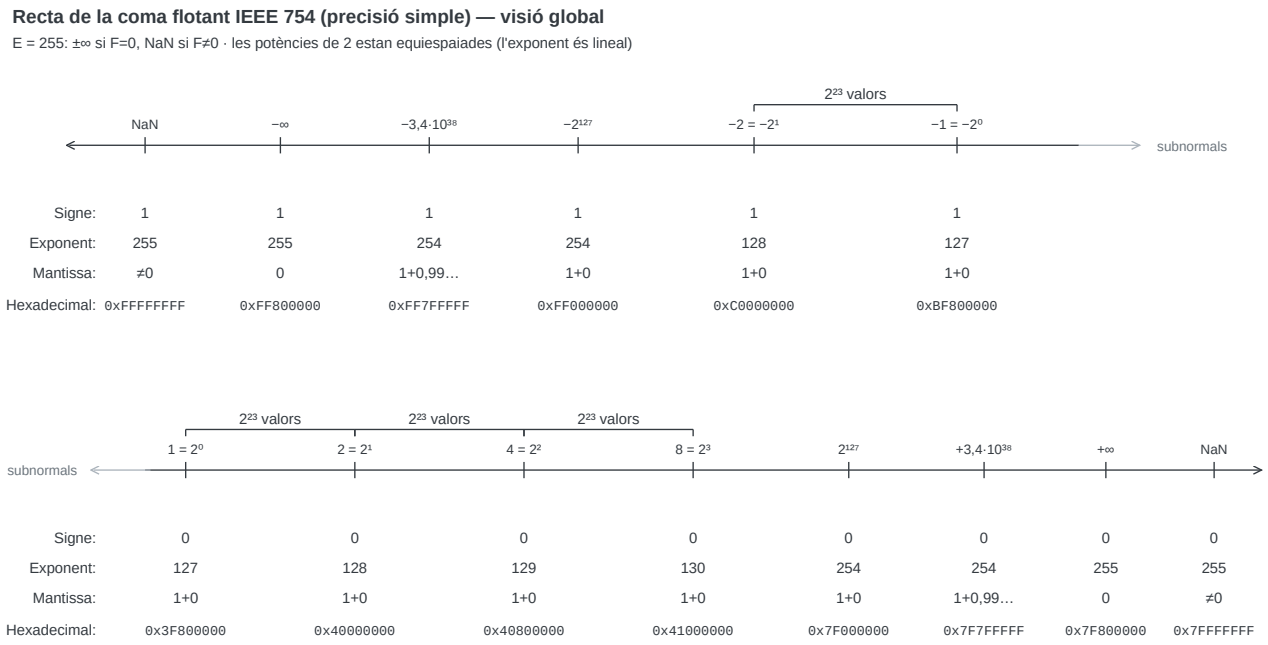


Figura 5.3: Recta de la coma flotant IEEE 754 (precisió simple): $\pm\infty$ i NaN als extrems ($E = 255$) i les potències de 2 equidistants: l'eix segueix l'ordre de les codificacions, i entre dues potències de 2 consecutives n'hi ha sempre 2^{23}.

La quantitat 1 ULP, una *unit in the last place* (ULP), és el valor $0{,}0\ldots001_2$ amb un sol bit a 1 a la posició de menys pes de la mantissa. La fita $\eta_{max} < 2^{-23}$ és independent del valor de v i depèn exclusivament de la precisió de la mantissa.

Essencial 5.4: Subdesbordament

Hi ha una excepció important a la fita d'error relatiu. Quan el valor absolut del resultat és inferior al mínim positiu normalitzat representable $2^{E_{min}} = 2^{-126}$, si s'aproxima simplement per zero l'error relatiu és $\eta = 1$, és a dir, de l'ordre del 100 %. Això pot invalidar completament el significat del resultat.
Quan el maquinari detecta aquesta situació, genera una excepció de **subdesbordament** (*underflow*), que permet al programari tractar-la adequadament. Els nombres **subnormals** (Secció 5.1.6) són el mecanisme que defineix l'IEEE 754 per reduir aquest error de forma gradual.

Aprofundiment 5.1: Càlcul formal de la fita d'error relatiu

Donat $v_0 = m \times 2^e$ i $v_1 = (m + 2^{-23}) \times 2^e$, l'error absolut màxim a l'interval (v_0, v_1) és:

$$\varepsilon_{max} < |v_1 - v_0| = 2^{-23} \times 2^e = 2^{e-23}$$

Per a l'error relatiu, el cas pitjor dins l'interval és el valor de menor magnitud, que és v_0:

$$\eta_{max} < \frac{\varepsilon_{max}}{|v_0|} = \frac{2^{e-23}}{m \times 2^e} = \frac{2^{-23}}{m}$$

Com que la mantissa normalitzada satisfà $m \geq 1{,}0$, resulta:

$$\eta_{max} < 2^{-23} = 1 \text{ ULP} \approx 1{,}19 \times 10^{-7}$$

Aquesta fita és independent de e i de m: l'error relatiu de representació és uniformement acotat per 1 ULP arreu del rang normalitzat.
Amb arrodoniment al més pròxim (Secció 5.1.7), el valor triat és a menys de mig interval: $\varepsilon \leq 2^{e-24}$ i $\eta \leq 2^{-24} = \frac{1}{2}$ ULP.

5.1.6 Codificacions especials

L'estàndard IEEE 754 reserva dos valors del camp d'exponent per a codificacions especials: $E = 0000\,0000_2$ (tot zeros) i $E = 1111\,1111_2$ (tot uns). Per tant, el **rang efectiu** de l'exponent queda reduït a $[-126,\ +127]$ en precisió simple. La Figura 5.4 en mostra el mapa complet.

! EC 5.2: IEEE 754 — Taula de codificacions

Exponent E	Fracció F	Interpretació
0000 0000	000 ... 0	**Zero** (± 0)
0000 0000	$\neq$ 000 ... 0	**Subnormal** ($\pm 0{,}F \times 2^{-126}$)
0000 0001 a 1111 1110	qualsevol	**Normalitzat** ($\pm 1{,}F \times 2^{e}$)
1111 1111	000 ... 0	**Infinit** ($\pm\infty$)
1111 1111	$\neq$ 000 ... 0	**NaN**

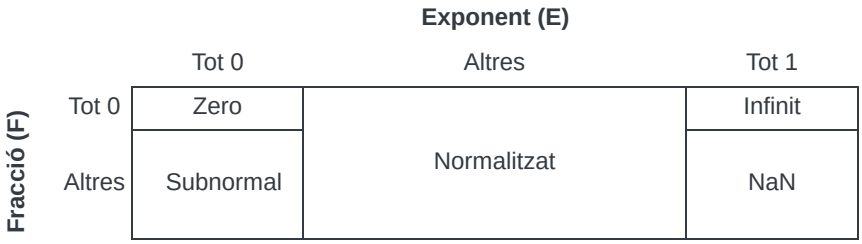


Figura 5.4: Mapa de codificacions IEEE 754 de precisió simple: els eixos E i F determinen unívocament la interpretació de cada patró de bits.

5.1.6.1 Zero

Els formats normalitzats no poden representar el zero, ja que el bit ocult imposa $m \geq 1{,}0$. El zero es codifica convencionalment amb tots els bits d'exponent i fracció a zero. A causa del bit de signe, existeixen dues representacions: $+0$ i -0. L'estàndard defineix que $+0 = -0$ en les comparacions.

5.1.6.2 Subnormals

Com s'ha vist a la Secció 5.1.5, quan $|v| < 2^{-126}$ l'error relatiu pot arribar al 100 % si s'arrodoneix a zero. Els nombres **subnormals** (*subnormals*), també anomenats **denormals** (*denormalized*), cobreixen l'interval $(0,\ 2^{-126})$ amb nombres de la forma:

$$v = \pm 0{,}F \times 2^{-126} \qquad (F \neq 0)$$

A diferència dels nombres normalitzats, **no hi ha bit ocult per als subnormals**: la part entera de la mantissa és 0 (en comptes d'1). L'exponent implícit dels subnormals és sempre $e = -126$; és a dir,

no s'obté aplicant la fórmula $e = E_u - 127$ (que amb $E_u = 0$ donaria -127), sinó que és, per definició, el mateix exponent que el del normal més petit (2^{-126}).

Amb subnormals, la fita d'error absolut dins l'interval $(0,\ 2^{-126})$ és la mateixa que a l'interval normalitzat adjacent $(2^{-126},\ 2^{-125})$:

$$\varepsilon_{max} < 2^{-149}$$

L'**error relatiu creix de forma gradual** a mesura que v s'acosta a zero, en lloc d'augmentar de cop com passaria sense subnormals. Per aquesta raó el mecanisme es coneix també com a **subdesbordament gradual** (*gradual underflow*).

En precisió simple, la fita d'error relatiu és 2^{-23} a l'extrem superior de l'interval $(0,\ 2^{-126})$ i creix progressivament fins a $2^0 = 1$ a mesura que v s'acosta a zero i la mantissa no normalitzada perd dígits significatius.

Essencial 5.5

Per a cada exponent normalitzat $E \in [1, 254]$, la fracció F pren els 2^{23} valors possibles $\{0, 1, \ldots, 2^{23} - 1\}$. Per exemple, $F = 0$ correspon a la potència de 2 exacta ($\pm 1{,}0 \times 2^e$), un nombre normalitzat vàlid.
Tanmateix, per als subnormals ($E = 0$), el rang de F és el mateix, però el cas $F = 0$ codifica ± 0, que no és un subnormal. Per tant, hi ha $2^{23} - 1$ subnormals per signe, enfront dels 2^{23} valors per exponent dels normals. Aquesta **asimetria** és visible comparant Figura 5.5 amb Figura 5.3.

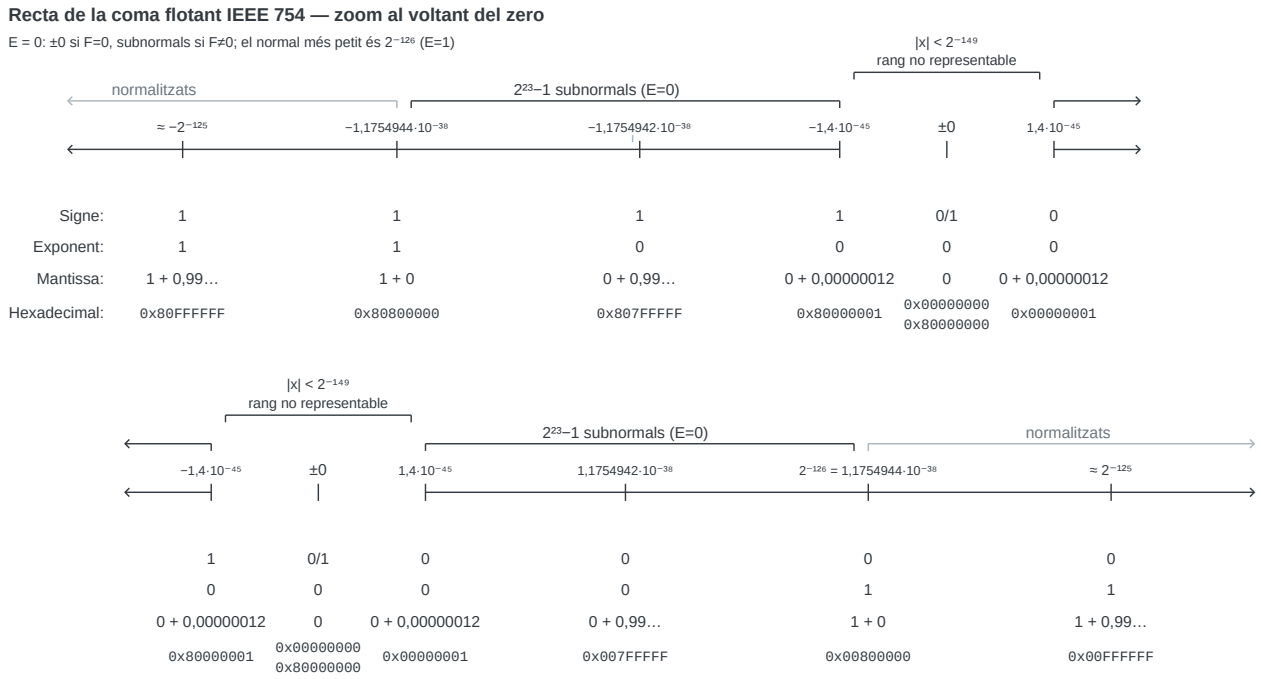


Figura 5.5: Zoom al voltant del zero: ± 0 i els subnormals ($E = 0$) omplen el buit entre el zero i el normal més petit (2^{-126}).

5.1.6.3 Infinit

L'infinit s'incorpora a l'aritmètica per raons pràctiques. Sigui l'expressió $1\,/\,(1 + 100/x)$ per a valors de x pròxims a zero: sense infinit, el càlcul $100/x$ produiria una excepció de desbordament que avortaria

el programa, tot i que el resultat final de l'expressió convergeix a zero. Amb infinit, la seqüència d'operacions és $100/x = \pm\infty$, $1 + \infty = \infty$, $1/\infty = 0$, i el programa continua sense avortar: l'excepció de desbordament només activa el seu indicador (RISC-V 5.3).

L'infinit es codifica amb tots els bits d'exponent a 1 i tots els de fracció a zero. S'apliquen regles aritmètiques senzilles: $1/0 = \infty$, $1/\infty = 0$, $x + \infty = \infty$, etc.

5.1.6.4 NaN

Algunes operacions no produeixen cap valor real vàlid: $\sqrt{-1}$, $\log(-1)$, $0/0$, $\infty - \infty$, $0 \times \infty$. En lloc d'avortar l'execució, el resultat pren la codificació especial **NaN**, i l'execució continua. Qualsevol operació aritmètica posterior amb un operand NaN produeix un altre NaN (les comparacions, en canvi, retornen 0: RISC-V 5.7), de manera que l'error es propaga fins al resultat final, on el programa pot fer una única comprovació.

Un NaN es codifica amb tots els bits d'exponent a 1 i almenys un bit de fracció no nul (per distingir-lo de l'infinit). El bit de més pes de la fracció distingeix el NaN **silenciós** (1), que es propaga sense activar cap indicador, del **senyalitzador** (0), que activa NV. A RISC-V, les operacions que generen un NaN donen el NaN canònic `0x7FC00000`.

5.1.7 Arrodoniment

Quan el resultat exacte v d'una operació no és representable, cal triar entre els dos valors representables $v_0 < v < v_1$ més pròxims. Aquesta aproximació s'anomena arrodoniment. L'IEEE 754 (revisió de 2008) defineix cinc modes; els quatre d'ús més habitual són:

Mode	Criteri
Al més pròxim (per defecte)	El valor representable més proper a v; en cas d'empat, el parell
A zero	El valor de menor magnitud (truncament)
A $+\infty$	El valor representable major
A $-\infty$	El valor representable menor

El mode per defecte és l'**arrodoniment al més pròxim** (*round to nearest, ties to even*, RNE). Proporciona l'error mínim possible i, en el cas equidistant, tria el valor parell (el que té el bit de menys pes a 0) per evitar biaixos sistemàtics en cadenes llargues de càlculs.

Els modes d'arrodoniment a $+\infty$ i a $-\infty$ s'usen conjuntament en aritmètica d'intervals, una tècnica que acota els errors d'arrodoniment per garantir la fiabilitat dels resultats numèrics. L'estàndard inclou encara un cinquè mode poc emprat: «al més pròxim amb empat cap a la magnitud major» (*round to nearest, ties to max magnitude*, RMM). RISC-V implementa els cinc modes al registre `fcsr` (RISC-V 5.3).

5.1.7.1 Bits de guarda

Quan el resultat exacte d'una operació té més bits que la mantissa, els bits de guarda (els bits sobrants) determinen com s'arrodoneix. N'hi ha prou amb tres (Figura 5.6):

- G (*Guard*): el primer bit descartat.
- R (*Round*): el segon bit descartat.

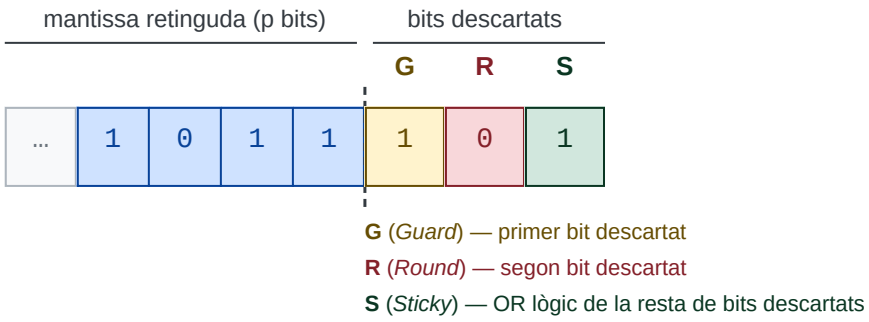


Figura 5.6: Esquema dels bits de guarda (G), arrodoniment (R) i *sticky* (S) respecte de la mantissa retinguda, amb la regla RNE.

- S (*Sticky*): l'OR lògic de la resta de bits descartats.

La regla RNE en funció de G, R i S és:

G	R	S	Acció
0	—	—	A l'anterior (truncar)
1	0	0	Empat → al parell
1	1	—	Al següent (sumar 1 ULP)
1	0	1	Al següent (sumar 1 ULP)

Exemple 5.4: Arrodoniment RNE

Arrodoneix les mantisses de 6 bits següents a 3 bits (els 3 bits descartats són directament G, R i S):

Valor exacte	$G\,R\,S$	Criteri	Resultat
101,010	0 1 0	$G = 0 \rightarrow$ a l'anterior	101
101,011	0 1 1	$G = 0 \rightarrow$ a l'anterior	101
101,101	1 0 1	$G = 1$, R o $S = 1$ $\rightarrow$ al següent	110
101,100	1 0 0	$G = 1$, $R = S = 0$ $\rightarrow$ empat $\rightarrow$ al parell (110)	110
100,100	1 0 0	$G = 1$, $R = S = 0$ $\rightarrow$ empat $\rightarrow$ al parell (100)	100

⚠ Aprofundiment 5.2: Per què 3 bits de guarda són suficients?

Durant una suma o resta, la mantissa del sumand de menor magnitud es desplaça a la dreta per igualar els exponents, i alguns bits queden fora de la mantissa de p bits. Descartar-los introdueix error, mentre que conservar-los tots requeriria un sumador de mida arbitrària. Es demostra que els tres bits G, R i S (Secció 5.1.7.1) basten per obtenir **exactament** el mateix resultat arrodonit que amb tots els bits: G pot incorporar-se a la mantissa en normalitzar, R decideix l'arrodoniment i S (l'OR de la resta) distingeix el cas equidistant.
Exemple (precisió $p = 6$ bits):

```
Resta amb G, R i S (p = 6)

  1,00000
- 0,00001 101   ← G=1, R=0, S=1 → al següent
-----------
  0,11110 011
Normalitzar:    1,11100 11
Arrodonir:      1,11101     (G=1, R=1 → al següent)
```

Amb tots els bits, el resultat arrodonit és idèntic: 1,11101.

5.2 Operacions en coma flotant

5.2.1 Suma i resta

L'algorisme de suma i resta en coma flotant segueix cinc passos:

1. **Igualar els exponents** al major dels dos operands, desplaçant a la dreta la mantissa del de menor magnitud.
2. **Sumar o restar les mantisses** segons els signes dels operands.
3. **Normalitzar** el resultat, desplaçant la mantissa fins que la part entera sigui 1, ajustant l'exponent en conseqüència; cal comprovar desbordament/subdesbordament.
4. **Arrodonir** la mantissa a p bits.
5. **Codificar** signe, exponent i fracció.

💡 Exemple 5.5: Suma en coma flotant, base 10

Calcula $z = x + y$ amb $x = 9{,}999 \times 10^{1}$ i $y = 1{,}680 \times 10^{-1}$, amb 4 dígits de precisió a la mantissa.
Pas 1. Igualar exponents (al major, $e = 1$):

$$y = 1{,}680 \times 10^{-1} = 0{,}01680 \times 10^{1}$$

Pas 2. Sumar les mantisses:

$$\begin{array}{r} 9{,}999 \\ +\ 0{,}01680 \\ \hline 10{,}01580 \end{array} \times 10^{1}$$

Pas 3. Normalitzar:

$$z = 1{,}001580 \times 10^{2} \qquad (e = 2 \in [-99, +99],\ \text{cap excepció})$$

Pas 4. Arrodonir a 4 dígits (dígits descartats en negreta: **580**; com que superen la meitat, s'arrodoneix al següent):

$$z \approx 1{,}002 \times 10^2$$

Exemple 5.6: Suma en coma flotant, precisió simple

Calcula $z = x + y$ amb $x =$ `0x3F40000D` i $y =$ `0xC0800004`.
Descomposició dels operands:

$$x = 0 \mid 0111\,1110 \mid 100\,0000\,0000\,0000\,0000\,1101_2$$
$$y = 1 \mid 1000\,0001 \mid 000\,0000\,0000\,0000\,0000\,0100_2$$

Descodificació dels operands:

- x: $S = 0$, $E_u = 126 \Rightarrow e = -1$; $x = +1{,}1000\,0000\,0000\,0000\,0001\,101_2 \times 2^{-1}$.
- y: $S = 1$, $E_u = 129 \Rightarrow e = +2$; $y = -1{,}0000\,0000\,0000\,0000\,0000\,100_2 \times 2^2$.

Pas 1. Igualar exponents (al major, $e = 2$; cal desplaçar x 3 posicions a la dreta):

$$x = +0{,}0011\,0000\,0000\,0000\,0000\,0011\,01_2 \times 2^2$$

Pas 2. Restar les magnituds (signes oposats; $|y| > |x|$; el resultat pren el signe de y):

$$\begin{array}{r} 1{,}0000\,0000\,0000\,0000\,0000\,100 \\ -\ 0{,}0011\,0000\,0000\,0000\,0000\,001101 \\ \hline 0{,}1101\,0000\,0000\,0000\,0000\,010011 \end{array} \times 2^2$$

Pas 3. Normalitzar (desplaçar 1 posició a l'esquerra, $e = 2 - 1 = 1$):

$$|z| = 1{,}1010\,0000\,0000\,0000\,0000\,10011_2 \times 2^1$$

$e = 1 \in [-126, +127]$: cap excepció.
Pas 4. Arrodonir a 24 bits significatius (bits de guarda en negreta: **11**; $G = 1$, $R = 1 \rightarrow$ al següent):

$$\begin{array}{r} 1{,}1010\,0000\,0000\,0000\,0000\,100 \\ +\qquad\qquad\qquad\qquad\qquad\qquad 1 \\ \hline 1{,}1010\,0000\,0000\,0000\,0000\,101 \end{array} \times 2^1$$

Pas 5. Codificar ($S = 1$, $E_u = 1 + 127 = 128 = 1000\,0000_2$):

$$z = 1 \mid 1000\,0000 \mid 101\,0000\,0000\,0000\,0000\,0101_2 = \texttt{0xC0500005}$$

Error de representació:

$$\varepsilon = |1{,}1010\,0000\,0000\,0000\,0000\,101 - 1{,}1010\,0000\,0000\,0000\,0000\,10011| \times 2^1$$
$$= 0{,}0000\,0000\,0000\,0000\,0000\,0000\,1 \times 2^1 = 1{,}0_2 \times 2^{-24} \approx 6{,}0 \times 10^{-8}$$

5.2.2 Multiplicació i divisió

L'algorisme de multiplicació i divisió és més simple que el de suma i resta perquè no cal igualar exponents:

1. **Sumar els exponents** (multiplicació) o **restar-los** (divisió).
2. **Multiplicar o dividir les mantisses**.
3. **Determinar el signe**: negatiu si els operands tenen signes diferents.
4. **Normalitzar** el resultat: cal comprovar desbordament/subdesbordament.
5. **Arrodonir** la mantissa a p bits.
6. **Codificar** signe, exponent i fracció.

Exemple 5.7: Multiplicació en coma flotant, base 10

Calcula $z = x \times y$ amb $x = -2{,}111 \times 10^{10}$ i $y = 9{,}200 \times 10^{-5}$, amb 4 dígits de precisió.
Pas 1. Sumar els exponents: $10 + (-5) = 5$
Pas 2. Multiplicar les mantisses:

$$2{,}111 \times 9{,}200 = 19{,}4212$$

Pas 3. Signe: negatiu (signes oposats).
Pas 4. Normalitzar:

$$|z| = 1{,}94212 \times 10^{6} \qquad (e = 6 \in [-99, +99])$$

Pas 5. Arrodonir a 4 dígits (dígits descartats: **12**; primer dígit descartat $< 5 \rightarrow$ a l'anterior):

$$z \approx -1{,}942 \times 10^{6}$$

Exemple 5.8: Multiplicació en coma flotant, precisió simple

Calcula $z = x \times y$ amb $x =$ `0x3F600000` i $y =$ `0xBED00002`.
Descomposició dels operands:

$$x = 0 \mid 0111\,1110 \mid 110\,0000\,0000\,0000\,0000\,0000_2$$
$$y = 1 \mid 0111\,1101 \mid 101\,0000\,0000\,0000\,0000\,0010_2$$

Descodificació:

- x: $S = 0$, $e = -1$, mantissa $1{,}1100\,0000\,0000\,0000\,0000\,000_2$.
- y: $S = 1$, $e = -2$, mantissa $1{,}1010\,0000\,0000\,0000\,0000\,010_2$.

Pas 1. Sumar els exponents: $e = -1 + (-2) = -3$
Pas 2. Multiplicar les mantisses (es treballa amb els valors absoluts):

$$\begin{array}{r} 1{,}11 \\ \times\ 1{,}1010\,0000\,0000\,0000\,0000\,010 \\ \hline 10{,}1101\,1000\,0000\,0000\,0000\,0111 \end{array}$$

Pas 3. Signe: negatiu (signes oposats).
Pas 4. Normalitzar (desplaçar 1 posició a la dreta, $e = -3 + 1 = -2$):

$$|z| = 1{,}0110\,1100\,0000\,0000\,0000\,0011\,1_2 \times 2^{-2}$$

$e = -2 \in [-126, +127]$: cap excepció.
Pas 5. Arrodonir a 24 bits (bits de guarda: **11**; $G = 1$, $R = 1 \rightarrow$ al següent):

$$\begin{array}{r} 1{,}0110\,1100\,0000\,0000\,0000\,001 \\ +\qquad\qquad\qquad\qquad\qquad\qquad 1 \\ \hline 1{,}0110\,1100\,0000\,0000\,0000\,010 \end{array} \times 2^{-2}$$

Pas 6. Codificar ($S = 1$, $E_u = -2 + 127 = 125 = 0111\,1101_2$):

$$z = 1 \mid 0111\,1101 \mid 011\,0110\,0000\,0000\,0000\,0010_2 = \texttt{0xBEB60002}$$

Exemple 5.9: Divisió en coma flotant, precisió simple

Calcula $z = x/y$ amb $x = \texttt{0x41000000}$ i $y = \texttt{0x40400000}$.
Descodificació:

- x: $S = 0$, $e = 3$, mantissa $= +1{,}0_2 \rightarrow x = +8{,}0$
- y: $S = 0$, $e = 1$, mantissa $= +1{,}1_2 \rightarrow y = +3{,}0$

Pas 1. Restar els exponents: $e = 3 - 1 = 2$
Pas 2. Dividir les mantisses:

$$\frac{1{,}0_2}{1{,}1_2} = 0{,}1010\,1010\,1010\ldots_2$$

A diferència de la multiplicació, que dona un producte finit de $2p$ bits com a màxim, la divisió de mantisses sol donar un quocient **periòdic que no acaba**, de manera que l'arrodoniment és gairebé sempre necessari.
Pas 3. Signe: positiu (mateix signe).
Pas 4. Normalitzar (desplaçar 1 posició a l'esquerra, $e = 2 - 1 = 1$):

$$|z| = 1{,}0101\,0101\,0101\,0101\,0101\,010\,\mathbf{1010}\ldots_2 \times 2^1$$

$e = 1 \in [-126, +127]$: cap excepció.
Pas 5. Arrodonir a 24 bits (bits de guarda: $G = 1$, $R = 0$, $S = 1 \rightarrow$ al següent):

$$\begin{array}{r} 1{,}0101\,0101\,0101\,0101\,0101\,010 \\ + \qquad\qquad\qquad\qquad\qquad\quad 1 \\ \hline 1{,}0101\,0101\,0101\,0101\,0101\,011 \end{array} \times 2^1$$

Pas 6. Codificar ($S = 0$, $E_u = 1 + 127 = 128 = 1000\,0000_2$):

$$z = 0 \mid 1000\,0000 \mid 010\,1010\,1010\,1010\,1010\,1011_2 = \texttt{0x402AAAAB} \approx 2{,}6667$$

5.2.3 No-associativitat

La suma d'enters en Ca2 és associativa: $(a + b) + c = a + (b + c)$ sempre. En coma flotant, aquesta propietat simplement no es compleix a causa dels arrodoniments intermedis.

Exemple 5.10: Contraexemple de l'associativitat

Siguin $x = -1{,}1_2 \times 2^{127}$, $y = +1{,}1_2 \times 2^{127}$, $z = 1{,}0_2 \times 2^0$. Comprova que $x + (y + z) \neq (x + y) + z$.
$x + (y + z)$:
$y + z$: cal igualar exponents; $z = 2^0$ és tan petit respecte de $y \approx 2^{127}$ que queda fora de la precisió i no contribueix a la suma:

$$y + z = 1{,}1_2 \times 2^{127} + 0{,}0\ldots0\ 1_2 \times 2^{127} \approx 1{,}1_2 \times 2^{127}$$

$$x + (y + z) = -1{,}1_2 \times 2^{127} + 1{,}1_2 \times 2^{127} = 0{,}0$$

$(x + y) + z$:

$$x + y = -1{,}1_2 \times 2^{127} + 1{,}1_2 \times 2^{127} = 0{,}0$$
$$(x + y) + z = 0{,}0 + 1{,}0 = 1{,}0$$

Per tant, $x + (y + z) = 0{,}0 \neq 1{,}0 = (x + y) + z$.

Essencial 5.6: La suma en coma flotant no és associativa

La no-associativitat de la suma en coma flotant té conseqüències pràctiques importants:

- **Paral · lelització**: dividir una suma de n elements en subcadenes i sumar els resultats parcials pot donar un resultat diferent del càlcul seqüencial. El resultat depèn del nombre de processadors i de l'ordre d'execució.
- **Comparacions d'igualtat**: comparar el resultat d'una cadena de càlculs amb un valor exacte (`== 1.0`) és perillós. És preferible comprovar si el resultat es troba dins d'un interval de tolerància $1{,}0 \pm \varepsilon$.

5.3 Coma flotant a RISC-V (RV32F)

5.3.1 Registres i ABI

L'**extensió F** de RISC-V afegeix 32 registres de coma flotant de 32 bits, **`f0`**–**`f31`**, independents dels registres enters d'RV32I. L'ABI `ilp32f` assigna noms i rols a cadascun, seguint el mateix esquema de temporals, segurs i paràmetres que els registres enters (RISC-V 2.24).

RISC-V 5.2: RV32F ABI — Registres de coma flotant

ISA	ABI	Propòsit	*Saver*
`f0`–`f7`	`ft0`–`ft7`	**Temporals**	*Caller*
`f8`–`f9`	`fs0`–`fs1`	**Segurs**	*Callee*
`f10`–`f11`	`fa0`–`fa1`	**Arguments** de subrutines / Valors de **retorn**	*Caller*
`f12`–`f17`	`fa2`–`fa7`	**Arguments** de subrutines	*Caller*
`f18`–`f27`	`fs2`–`fs11`	**Segurs**	*Callee*
`f28`–`f31`	`ft8`–`ft11`	**Temporals**	*Caller*

A més dels registres `f0`–`f31`, l'extensió F afegeix el registre de control i estat de coma flotant **`fcsr`** (*Floating-Point Control and Status Register*), que controla el mode d'arrodoniment i registra les excepcions produïdes per les operacions de coma flotant (cosa que fa obligatòria l'extensió Zicsr, que es presenta a T9: RISC-V 9.2).

RISC-V 5.3: RV32F ISA — Registre de control `fcsr`



Bits	Camp	Descripció
`31:8`	—	Reservats (llegeixen com a 0)
`7:5`	`frm`	Mode d'arrodoniment
`4:0`	`fflags`	Indicadors d'excepció (*sticky*)

Modes d'arrodoniment (`frm`):

Codi	Mnemònic	Criteri
`000`	`RNE`	Al més pròxim, empat al parell (per defecte, Secció 5.1.7)
`001`	`RTZ`	A zero (truncament)
`010`	`RDN`	A $-\infty$
`011`	`RUP`	A $+\infty$
`100`	`RMM`	Al més pròxim, empat al de major magnitud

Indicadors d'excepció (`fflags`):

Bit	Mnemònic	Excepció
`4`	`NV`	Operació invàlida ($\sqrt{-1}$, $0/0$, $\infty - \infty$…)
`3`	`DZ`	Divisió per zero
`2`	`OF`	Desbordament
`1`	`UF`	Subdesbordament
`0`	`NX`	Resultat inexacte (arrodoniment)

Els indicadors són *sticky*: un cop activats pel maquinari, romanen actius fins que el programari els esborra explícitament. A RISC-V, aquestes excepcions no interrompen l'execució: només activen l'indicador corresponent, i és el programari qui l'ha de consultar.

EC 5.3: Paral · lelisme entre registres enters i de coma flotant

Els noms ABI dels registres de coma flotant segueixen el mateix esquema que els dels registres enters:

Categoria	Enters	Coma flotant
Temporals	`t0`–`t6`	`ft0`–`ft11`
Segurs	`s0`–`s11`	`fs0`–`fs11`
Arguments / retorn	`a0`–`a7`	`fa0`–`fa7`

Essencial 5.7: Preservació dels registres de coma flotant

Les convencions de preservació s'apliquen als registres de coma flotant exactament igual que als enters:

- Els **registres temporals** (`ft0`–`ft11`) i els **d'arguments** (`fa0`–`fa7`) no es preserven: poden ser modificats lliurement per qualsevol subrutina. Si la funció que crida en necessita el valor després d'una crida, els ha de desar ella mateixa.
- Els **registres segurs** (`fs0`–`fs11`) han de ser restaurats al seu valor original per la subrutina que els modifica. Si una subrutina usa `fs0`, ha de desar-lo al pròleg i restaurar-lo a l'epíleg, exactament igual que faria amb `s0`.

El pròleg i l'epíleg d'una subrutina que usa registres segurs de coma flotant segueixen el mateix patró que per als enters (Secció 3.9.4):

RV32IF

```
sub:
        addi    sp, sp, -8
        fsw     fs0, 0(sp)      # desa fs0 (callee-saved)
        fsw     fs1, 4(sp)      # desa fs1 (callee-saved)

        # cos de la subrutina

        flw     fs0, 0(sp)      # restaura fs0
        flw     fs1, 4(sp)      # restaura fs1
        addi    sp, sp, 8
        ret
```

5.3.2 Instruccions

Les instruccions aritmètiques, de comparació i de conversió de precisió simple porten el sufix `.s` (`fadd.s`, `flt.s`, `fcvt.w.s`); les de lectura i escriptura (`flw`, `fsw`) i les de moviment de bits (`fmv.x.w`, `fmv.w.x`) porten la lletra de la mida, com les d'enters. Els operands de les instruccions **aritmètiques** són tots registres de coma flotant: **no hi ha operands immediats flotants** ni instruccions aritmètiques que combinin registres enters i flotants. Els registres enters només hi apareixen com a base d'adreça (`flw`, `fsw`), com a destí de les comparacions i com a origen o destí de les conversions i dels moviments.

RISC-V 5.4: RV32F ISA — Lectura i escriptura

Mnemònic	Operands	Operació	Nom	Tipus
`flw`	`rd, offset(rs1)`	$rd \leftarrow M_w[rs1 + SignExt(offset)]$	*Float Load Word*	I
`fsw`	`rs2, offset(rs1)`	$M_w[rs1 + SignExt(offset)] \leftarrow rs2$	*Float Store Word*	S

`rs1` és sempre un **registre enter**. L'adreça efectiva ha de ser múltiple de 4 (alineació de paraula).

RISC-V 5.5: RV32F ISA — Instruccions aritmètiques

Mnemònic	Operands	Operació	Nom	Tipus
`fadd.s`	`rd, rs1, rs2`	$rd \leftarrow rs1 + rs2$	*Float Add*	R
`fsub.s`	`rd, rs1, rs2`	$rd \leftarrow rs1 - rs2$	*Float Subtract*	R
`fmul.s`	`rd, rs1, rs2`	$rd \leftarrow rs1 \times rs2$	*Float Multiply*	R
`fdiv.s`	`rd, rs1, rs2`	$rd \leftarrow rs1/rs2$	*Float Divide*	R
`fsqrt.s`	`rd, rs1`	$rd \leftarrow \sqrt{rs1}$	*Float Square Root*	R
`fmin.s`	`rd, rs1, rs2`	$rd \leftarrow \min(rs1, rs2)$	*Float Minimum*	R
`fmax.s`	`rd, rs1, rs2`	$rd \leftarrow \max(rs1, rs2)$	*Float Maximum*	R

Aprofundiment 5.3: Instruccions fusionades i format R4

L'extensió F inclou quatre instruccions de **multiplicació i suma fusionades** (*fused multiply-add*, FMA), que calculen un producte i una suma en una sola instrucció (RISC-V 2.6). La bàsica és `fmadd.s`:

RV32IF

```
fmadd.s ft0, ft1, ft2, ft3    # ft0 <- ft1 * ft2 + ft3
```

Les altres tres només canvien els signes: `fmsub.s` calcula $rs1 \times rs2 - rs3$; `fnmadd.s`, $-(rs1 \times rs2) - rs3$, i `fnmsub.s`, $-(rs1 \times rs2) + rs3$.
Com que necessiten **tres registres font**, no caben al format R i en tenen un de propi, el **R4** (Figura 5.7): el camp `rs3` (`fs3` a la figura, perquè és un registre de coma flotant) ocupa els bits [31:27], on el format R té la part alta de `funct7`; `funct2` (bits [26:25]) indica la precisió (`00` per a la precisió simple), i `funct3` conté el mode d'arrodoniment.

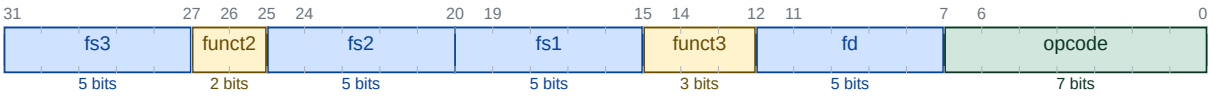


Figura 5.7: Format d'instrucció R4 (RV32F), el de les instruccions fusionades.

La diferència amb la seqüència `fmul.s` + `fadd.s` no és només de velocitat. `fmadd.s` calcula $rs1 \times rs2 + rs3$ com si tingués precisió infinita i **arrodoneix una sola vegada**, al final (Secció 5.1.7); la seqüència de dues instruccions arrodoneix dues vegades, el producte i la suma. Per això el resultat de `fmadd.s` és, en general, més precís, i pot no coincidir amb el de la seqüència: com a la Secció 5.2.3, els arrodoniments intermedis canvien el resultat.

RISC-V 5.6: Pseudoinstruccions — Còpia entre registres de coma flotant: `fmv.s`

Pseudoinstrucció	Operació	Expansió
`fmv.s rd, rs`	$rd \leftarrow rs$	`fsgnj.s rd, rs, rs`

RARS admet `fmv.s` directament.

RISC-V 5.7: RV32F ISA — Comparacions

Mnemònic	Operands	Operació	Nom	Tipus
`feq.s`	`rd, rs1, rs2`	$rd \leftarrow (rs1 = rs2)\ ?\ 1 : 0$	*Float Equal*	R
`flt.s`	`rd, rs1, rs2`	$rd \leftarrow (rs1 < rs2)\ ?\ 1 : 0$	*Float Less Than*	R
`fle.s`	`rd, rs1, rs2`	$rd \leftarrow (rs1 \leq rs2)\ ?\ 1 : 0$	*Float Less or Equal*	R

El resultat s'escriu en un **registre enter** `rd` (0 o 1). Per implementar un salt condicional, cal combinar la comparació amb `beq` o `bne`:

RV32IF

```
        flt.s   t0, fa0, fa1    # t0 = 1 si fa0 < fa1, si no 0
        bne     t0, zero, label # salta si fa0 < fa1
```

Si algun operand és NaN, totes les comparacions retornen 0.

RISC-V 5.8: RV32F ISA — Conversions i moviments

Conversions (amb conversió numèrica):

Mnemònic	Operands	Operació	Nom
`fcvt.s.w`	`rd, rs1`	$rd \leftarrow (float)\ rs1$	*Enter → flotant*
`fcvt.s.wu`	`rd, rs1`	$rd \leftarrow (float)\ rs1$	*Natural → flotant*
`fcvt.w.s`	`rd, rs1`	$rd \leftarrow (int)\ rs1$	*Flotant → enter (arrodoniment segons `rm`; `rtz` per a la conversió de tipus de C)*
`fcvt.wu.s`	`rd, rs1`	$rd \leftarrow (unsigned)\ rs1$	*Flotant → natural (arrodoniment segons `rm`; `rtz` per a la conversió de tipus de C)*

`rd` de `fcvt.s.w`/`fcvt.s.wu` és un registre de coma flotant; `rs1` és un registre enter. A `fcvt.w.s`/`fcvt.wu.s` és a l'inrevés.

Moviments (còpia de bits sense conversió numèrica):

Mnemònic	Operands	Operació	Nom
`fmv.w.x`	`rd, rs1`	$rd \leftarrow rs1$	*Enter → flotant (bits)*
`fmv.x.w`	`rd, rs1`	$rd \leftarrow rs1$	*Flotant → enter (bits)*

`fmv.w.x` i `fmv.x.w` copien el patró de 32 bits sense cap conversió: el valor numèric en coma flotant del resultat és en general diferent del valor enter de l'operand.

Exemple 5.11: Conversió entre `float` i `int`: `rtz` vs. arrodoniment

Tradueix la funció C del Codi 5.1, que converteix un `float` a `int` fent una conversió de tipus explícita:

Codi 5.1: Conversió explícita d'un `float` a `int`.

C

```c
int truncar(float x) {
    return (int) x;
}
```

La conversió `(int)` de C sempre trunca cap a zero (per als valors que caben en un `int`). Per traduir-ho cal indicar explícitament el mode d'arrodoniment `rtz` a `fcvt.w.s`:

RV32IF

```
truncar:
        fcvt.w.s a0, fa0, rtz      # a0 = (int) x  [trunca cap a zero, com el cast de C]
        ret
```

Si s'escriu `fcvt.w.s a0, fa0` (sense operand de mode) l'assemblador hi assigna `dyn` (mode dinàmic): la conversió s'arrodoneix segons el mode vigent al camp `frm` del `fcsr` (RISC-V 5.3), que per defecte és RNE (al més pròxim, empat al parell). Per a $x = 2{,}5$, tots dos modes donen el mateix resultat, però per motius diferents: `rtz` trunca i RNE desempata al parell:

Instrucció	Mode	Resultat
`fcvt.w.s a0, fa0, rtz`	Truncament (conversió `(int)` de C)	2
`fcvt.w.s a0, fa0`	`dyn` → RNE per defecte	2 (empat, al parell)

Per a $x = 3{,}5$, en canvi, la diferència és evident: `rtz` dona 3; `dyn`/RNE dona 4 (empat, al parell). La instrucció `fcvt.w.s` trunca només quan se n'indica el mode `rtz`.
RARS 1.6 no segueix l'especificació en el primer cas: amb RNE (el mode per defecte, o `rne` explícit), `fcvt.w.s` hi dona 3 per a 2,5 (i −3 per a −2,5), tot i que 0,5 i 1,5 sí que els desempata al parell (0 i 2).

EC 5.4: No hi ha immediats flotants

Cap instrucció d'RV32F admet un operand constant directament a la instrucció. La manera habitual de fer servir una constant de coma flotant és declarar-la al segment `.data` i llegir-la de memòria amb `la` + `flw` (també es pot carregar el patró de bits en un registre enter amb `li` i copiar-lo amb `fmv.w.x`):

RV32IF

```
.data
const_u: .float 1.0

.text
        la      t0, const_u     # t0 <- @const_u
        flw     ft0, 0(t0)      # ft0 <- 1.0
```

5.3.3 Exemple de traducció

Exemple 5.12: Traducció d'una funció amb `float`

Tradueix a RV32F la funció C del Codi 5.2:

Codi 5.2: Funció amb una condició i operacions de coma flotant.
C

```
float func(float x) {
    if (x < 1.0f)
        return x * x;
    else
        return 2.0f - x;
}
```

El paràmetre `x` arriba per `fa0` i el resultat es retorna per `fa0`.
Pas 1. Constants. No hi ha immediats flotants: es declaren `1.0` i `2.0` al `.data`.
Pas 2. Registres segurs. `func` és una subrutina fulla: no fa cap crida. No cal preservar `ra` ni cap registre segur.
Pas 3. Codi.

RV32IF

```
.data
const_1: .float 1.0
const_2: .float 2.0

.text
func:
        la      t0, const_1
        flw     ft0, 0(t0)      # ft0 <- 1.0
        la      t1, const_2
        flw     ft1, 0(t1)      # ft1 <- 2.0

        flt.s   t2, fa0, ft0    # t2 = (x < 1.0) ? 1 : 0
        beq     t2, zero, sino  # si no (x >= 1.0), salta

        fmul.s  fa0, fa0, fa0   # return x * x
        ret

sino:
        fsub.s  fa0, ft1, fa0   # return 2.0 - x
        ret
```

Tema 6: Rendiment i potència

6.1 Rendiment

6.1.1 Definicions

El disseny de computadors aspira a millorar-ne contínuament el rendiment. Cal, per tant, definir amb precisió aquest concepte a fi de ser capaços de mesurar si els nous dissenys aporten millores o no.

6.1.1.1 Temps d'execució, rendiment i productivitat

El terme «rendiment» s'empra per a referir-se a dos conceptes diferents:

- **Temps d'execució o de resposta** (t_{exe}): Temps transcorregut, mesurat en segons ([s]), entre l'inici i el final d'una tasca. És el temps que experimenta l'usuari.
  - En un programa consisteix en la suma de diversos temps: Temps de CPU + temps d'espera d'E/S + temps en cua mentre s'executen altres programes (concurrència).
  - **A EC es considera exclusivament el temps de CPU**, ja que és el que depèn més directament del disseny de la CPU.
- **Productivitat** (*throughput*): Nombre de tasques completades per unitat de temps. S'usa sovint per quantificar el rendiment en centres de processament de dades (servidors d'aplicacions, bases de dades, etc.).

A EC, el rendiment es mesura amb el temps d'execució, i n'és l'**invers**:

$$\text{Rendiment} = \frac{1}{t_{exe}} \qquad (\text{s}^{-1}) \tag{6.1}$$

Quina **diferència** hi ha entre **temps d'execució** i **rendiment**?

- El temps d'execució és una mesura de «quant de temps» triga una tasca.
- El rendiment n'és l'invers: quantes vegades es pot executar la tasca per unitat de temps, l'una darrere l'altra. La productivitat, en canvi, compta també les execucions en paral · lel.

Com que el temps d'execució es mesura en segons ([s]), el rendiment es mesura en tasques per segon (o instruccions per segon, o operacions per segon, etc.) ([s^{-1}]).

> 💡 Exemple 6.1: Rendiment i productivitat de dos avions
>
> Un avió A porta 100 passatgers a 1200 km/h. Un avió B porta 500 passatgers a 1000 km/h.
> **Pregunta**: Quin té més rendiment?
> **Resposta**: A té un temps d'execució menor i, per tant, més rendiment (Equació 6.1). B té una productivitat major.

> 💡 Exemple 6.2: Rendiment i productivitat: dos computadors
>
> Un computador A té 1 CPU que tarda 10 μs. B té 200 CPU que tarden cada una 20 μs.
> **Pregunta**: Quin té més rendiment?
> **Resposta**: A té un temps d'execució menor i, per tant, més rendiment (Equació 6.1). B té una productivitat major.

Quina **relació** hi ha entre **productivitat** i **temps d'execució**?

- **Sempre**: A menor temps d'execució, major productivitat.
- **Només en cas de congestió**: Augmentar la productivitat pot disminuir el temps d'execució, ja que es redueix el temps que les tasques romanen a la cua (com en el cas d'un centre de dades). Però si el sistema no està congestionat, augmentar la productivitat no té cap efecte sobre el temps d'execució.

6.1.1.2 Guany de rendiment

El **guany de rendiment** o simplement **guany** (*speedup*), denotat pel símbol s, és la **relació** (adimensional) entre el rendiment d'un sistema **nou** i el d'un sistema de **referència** (original, inicial, etc.):

$$\text{Guany} = s = \frac{\text{Rendiment}_{nou}}{\text{Rendiment}_{\text{referència}}} = \frac{t_{\text{referència}}}{t_{nou}}$$

> 💡 Exemple 6.3: Interpretació del guany
>
> Un guany de 2 significa que el sistema nou és el doble de ràpid que el de referència.
> Si $t_{\text{referència}} = 10$ s i $t_{nou} = 5$ s, el guany és $s = 10$ s/5 s $= 2$.

6.1.2 Factors que influeixen en el temps d'execució

Per tal d'entendre quins aspectes influeixen en el temps d'execució, i, per tant, conèixer com és possible millorar-lo, es desglossa en els seus components:

$$t_{exe} = n_{cicles} \cdot t_{clock} = n_{cicles}/f_{clock} \qquad [\text{s}] \tag{6.2}$$

on t_{exe} és el temps d'execució de CPU, n_{cicles} el nombre de cicles de rellotge que tarda l'execució i t_{clock} i f_{clock} són el període ([s]) i la freqüència del rellotge ([Hz]), respectivament. El rendiment depèn del nombre de cicles i de la freqüència.

6.1.2.1 Reducció del nombre de cicles

Intuïtivament, s'identifiquen dos camins per a reduir el nombre de cicles. En primer lloc, fer que el programa s'executi amb menys instruccions (n_{ins}) i, en segon lloc, fer que aquestes tardin el mínim **nombre de cicles per instrucció** (CPI). És a dir, essent CPI la mitjana de cicles per instrucció de tot el programa, el nombre de cicles total es pot expressar com el producte:

$$n_{cicles} = n_{ins} \cdot \text{CPI}$$

I, per tant, el temps d'execució és:

$$t_{exe} = n_{ins} \cdot \text{CPI} \cdot t_{clock} \tag{6.3}$$

Per a reduir n_{ins} cal millorar l'algorisme, l'eficiència del compilador o l'ISA. El CPI d'un programa determinat és una mitjana que depèn del retard de cada tipus d'instrucció (CPI_i) i de quantes n'hi ha de cada tipus (n_i): és una **mitjana ponderada**. Així, suposant que té m tipus diferents d'instruccions, el seu CPI és:

$$\text{CPI} = \frac{1}{n_{ins}} \cdot \left(\sum_{i=1}^{m} n_i \cdot \text{CPI}_i \right) \tag{6.4}$$

En conseqüència, el **temps d'execució** és:

$$t_{exe} = \left(\sum_{i=1}^{m} n_i \cdot \text{CPI}_i \right) \cdot t_{clock} \tag{6.5}$$

Es pot reduir el temps d'execució reduint el retard CPI_i de cada tipus d'instrucció, però això requereix millorar el disseny de la microarquitectura. També es pot reduir el nombre d'instruccions n_i dels tipus més lents (amb major CPI_i), encara que sigui a costa d'augmentar el nombre de les més ràpides, si la suma total de cicles resultant és inferior.

Cal anar amb compte amb el concepte CPI, perquè es pot fer referència a:

- La quantitat de **cicles per instrucció d'un programa en conjunt**, o, fins i tot, a un conjunt de programes en un cert computador.
- La quantitat, en molts casos fixa, de **cicles d'un tipus concret d'instruccions** en un cert computador (CPI_i per al tipus i).[1]

En comparar el rendiment de dues versions d'un mateix programa generades per compiladors diferents, en realitat s'està comparant l'efectivitat dels compiladors. Com que el temps de cicle és el mateix, la comparació dels temps d'execució es pot fer comparant el nombre de cicles de rellotge.

💡 Exemple 6.4: Comparació de dues versions d'un programa

Es comparen dues versions d'un programa en un mateix computador, que disposa de tres tipus d'instruccions A, B i C, amb CPI d'1, 2 i 3, respectivament. El programa 1 (P1) consta de: 2 instruccions d'A, 1 de B i 2 de C. El programa 2 (P2) consta de: 4 instruccions d'A, 1 de B i 1 de C.
Pregunta 1: Quin és més ràpid?
Pregunta 2: Quins són els CPI?
Solució 1: Calculem el nombre d'instruccions i cicles totals de cada programa:

[1] Però cal anar amb compte: no sempre és cert que un tipus d'instrucció tardi un temps fix, ja que, com es veurà al Tema 7, les instruccions de lectura (*load*) i d'escriptura (*store*) d'un mateix programa poden tardar més o menys depenent de l'estat de la memòria cau en cada instant. No obstant això, encara es pot parlar del CPI de les instruccions de lectura, cas en què es fa referència a la mitjana de les instruccions de lectura dins d'un determinat programa.

Taula 6.1: Instruccions i cicles de les dues versions del programa.

Tipus d'instrucció	CPI	Ins. P1	Ins. P2	Cicles P1	Cicles P2
A	1	2	4	2	4
B	2	1	1	2	2
C	3	2	1	6	3
Total		**5**	**6**	**10**	**9**

Resposta 1: El més ràpid és P2 (9 cicles en lloc de 10).
Solució 2: Calculem els CPI:

- $\text{CPI}_1 = 10$ cicles/5 instruccions $= 2$.
- $\text{CPI}_2 = 9$ cicles/6 instruccions $= 1{,}5$.

Resposta 2: P2 té més instruccions però menys cicles per instrucció.

6.1.2.2 Augment de la freqüència de rellotge (reducció del temps de cicle)

Sigui un circuit combinacional A que llegeix dades d'entrada d'un biestable i escriu dades de sortida en un altre biestable. En principi, reduint el temps de cicle del circuit A es pot aconseguir que calculi més resultats per unitat de temps i, doncs, que es redueixi el temps total d'execució de les tasques.

Però cal parar atenció: reduir excessivament el temps de cicle pot fer que les dades de sortida de A encara no estiguin calculades, i que el resultat correcte no s'obtingui fins al cicle de rellotge següent. En aquest cas, haurà augmentat el nombre de cicles per executar la mateixa tasca i, per tant, no és segur que s'hi guanyi.

Exemple 6.5: La reducció del temps de cicle no sempre és una millora

Per exemple, suposem dos circuits A i B de retards diferents, que formen part del mateix processador i escriuen repetidament el seu resultat en un biestable quan arriba el flanc de rellotge. Suposem que inicialment el temps de cicle és t_{clock} i volem reduir-lo a t'_{clock}:

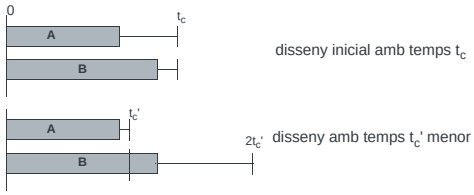


Figura 6.1: Els mateixos components, temps de cicle diferents.

Després de reduir el temps de cicle, la tasca A es realitzarà en menys temps, ja que no canvia el nombre de cicles, però la tasca B no podrà escriure en el primer cicle i necessitarà un cicle més: per tant, necessitarà el doble de cicles que abans. La reducció de t_{clock} només suposarà una millora de rendiment si la influència del cas A en el temps d'execució total és major que la de B.

Exemple 6.6: Comparació de dos processadors via el temps de cicle i el CPI

El processador A té $t_{cA} = 500$ ps i $\text{CPI}_A = 2$. El nou processador B té $t_{cB} = 250$ ps, però comporta augmentar el $\text{CPI}_B = 3$.
Pregunta: Serà més ràpid el nou disseny?
Solució:

$$\begin{aligned} s_B &= t_A/t_B = (n \cdot \text{CPI}_A \cdot t_{cA})/(n \cdot \text{CPI}_B \cdot t_{cB}) \\ &= (\text{CPI}_A \cdot t_{cA})/(\text{CPI}_B \cdot t_{cB}) = (2 \cdot 500 \text{ ps})/(3 \cdot 250 \text{ ps}) = 1{,}33 \end{aligned}$$

Resposta: El nou disseny B és 1,33 vegades més ràpid que l'original.

Exemple 6.7: Freqüència d'un disseny millorat

Pregunta: Suposem un computador A, amb $f_A = 2$ GHz, que executa un programa en $t_{exe} = 10$ s. Amb certes millores al circuit, s'executa en 6 s, però augmenta el nombre de cicles en un factor 1,2. Quina freqüència de rellotge té el nou disseny?
Solució: Tenim dues equacions amb dues incògnites, n_A i f_B:

$$\begin{aligned} f_A &= n_A/t_A \\ f_B &= n_B/t_B = (1{,}2 \cdot n_A)/t_B \end{aligned}$$

Aïllem n_A de la primera equació i la substituïm a la segona:

$$n_A = t_A \cdot f_A$$

$$t_B = 1{,}2 \cdot (t_A \cdot f_A)/f_B$$

Resposta: Aïllant f_B

$$f_B = (1{,}2 \cdot t_A \cdot f_A)/t_B = (1{,}2 \cdot 10 \text{ s} \cdot 2 \text{ GHz})/6 \text{ s} = 4 \text{ GHz}$$

6.2 Llei d'Amdahl

El **màxim guany** que es pot aconseguir minimitzant el retard d'una part (d'un circuit, d'un programa, etc.) està limitat per la fracció de temps que representa aquesta part sobre el temps total [13]. En conseqüència, cal concentrar els esforços d'optimització en aquelles parts del programa (o circuit) que comporten el major cost en temps.

Sigui, per exemple, un programa que s'executa en un temps t_0. Si s'optimitza una subrutina, que comporta una **fracció** P_x del temps total, i s'aconsegueix que aquesta subrutina tingui un **guany de rendiment** s_x, el temps d'execució total es redueix de t_0 a t_1 (Figura 6.2).

Quin ha estat el guany de rendiment total s?

$$s = \frac{t_0}{t_1} = \frac{t_0}{t_{\text{millorat}} + t_{\text{no-millorat}}}$$

on:

- $t_{\text{millorat}} = \frac{t_0 \cdot P_x}{s_x}$.
- $t_{\text{no-millorat}} = t_0 \cdot (1 - P_x)$.

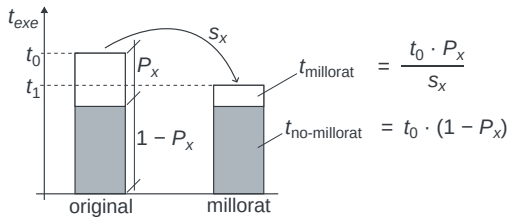


Figura 6.2: Temps total del programa original dividit en una part optimitzada (fracció P_x) i una part no optimitzada $(1 - P_x)$, i el programa millorat, amb temps t_1, on el temps de la part optimitzada queda dividit pel guany s_x.

Per tant:

$$s = \frac{t_0}{\frac{t_0 \cdot P_x}{s_x} + t_0 \cdot (1 - P_x)} = \frac{1}{\frac{P_x}{s_x} + (1 - P_x)} \tag{6.6}$$

Quin guany total màxim (s_{max}) es pot aconseguir amb la millor optimització possible d'aquesta subrutina (suposant un guany s_x infinit)?

$$s_{max} = \lim_{s_x \to \infty} s = \frac{1}{1 - P_x} \tag{6.7}$$

Exemple 6.8: Guany màxim segons la llei d'Amdahl

Pregunta: Quin és el guany total màxim assolible si es millora una part d'un programa que representa un 80 % del temps d'execució total?
Resposta: $s_{max} = 1/(1 - 0{,}80) = 5$

6.3 Potència

La potència (P) és l'energia dissipada per unitat de temps per la circulació de corrents elèctrics en el circuit:

$$P = P_d + P_s \tag{6.8}$$

La potència es pot dividir en dos tipus:

- La potència dinàmica (P_d), causada pels corrents de càrrega i descàrrega durant la commutació de les portes.
- La potència estàtica (P_s), fruit dels corrents paràsits que circulen pels transistors independentment de si commuten o no.

6.3.1 Potència dinàmica

El consum d'energia dinàmic és l'originat pels corrents de càrrega i descàrrega de les capacitats equivalents dels transistors que tenen lloc en les commutacions. En cada cicle complet de càrrega i descàrrega d'una porta es produeix un corrent elèctric, ja que hi ha un desplaçament net de càrrega del pol positiu a terra. La càrrega desplaçada en un cicle complet és:

$$q_i = C_i \cdot V_{CC}$$

on C_i és la capacitat equivalent en farads o capacitància del circuit connectat a la sortida de la porta. C_i depèn de la **connectivitat de sortida** (*fan-out*), del material i de la mida dels transistors i de les interconnexions a la sortida de la porta. V_{CC} és la tensió d'alimentació en volts. El corrent elèctric generat en tot el circuit durant un cicle de rellotge és igual a la càrrega i descàrrega agregada de totes les portes que commutin dividida pel temps de cicle de rellotge t_{clock}:

$$I = \sum q_i / t_{clock} = \sum C_i \cdot V_{CC} \cdot f_{clock}$$

No totes les portes commuten en cada cicle de rellotge. Es defineix el factor d'activitat (α) com la fracció mitjana de portes que commuten en cada cicle, per a un determinat programa o conjunt de programes representatius, de manera que $\sum C_i = \alpha \cdot C$, essent C la capacitància agregada de tot el circuit. Per tant, la potència dinàmica dissipada (consum d'energia per unitat de temps) en watts pel circuit és:

$$P_d = \alpha \cdot C \cdot V_{CC}^2 \cdot f_{clock} \tag{6.9}$$

A tall d'indicació, són bastant comuns els valors de α pròxims a 0,1. L'energia dissipada en joules durant un temps t és:

$$E = P \cdot t \tag{6.10}$$

⚠ Aprofundiment 6.1: Cicle de càrrega/descàrrega en el circuit equivalent RC d'una porta lògica

Les portes d'un circuit estan majoritàriament construïdes amb transistors de la família *complementary metal–oxide–semiconductor* (CMOS). Per exemple, una porta NOT (Figura 6.3):

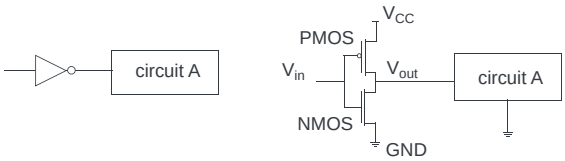


Figura 6.3: Porta NOT, representació funcional i implementació amb CMOS.

El transistor *p-channel metal–oxide–semiconductor* (PMOS) condueix quan la tensió d'entrada baixa d'un cert llindar (*threshold*) V_t, i roman tallat (circuit obert) si la tensió és superior. A la inversa, un transistor *n-channel metal–oxide–semiconductor* (NMOS) condueix quan la tensió d'entrada puja per sobre d'un cert llindar V_t, i resta obert si la tensió és inferior. Aquests llindars determinen els nivells lògics 1 i 0.
Càrrega: Quan la tensió a l'entrada de la porta V_{in} passa del valor lògic 1 al 0 (funció graó de flanc descendent), el transistor PMOS passa a conduir i l'NMOS queda obert. En conduir, el

corrent flueix a través de la seva resistència equivalent R, i la càrrega elèctrica s'acumula a la sortida de la porta, en la capacitat d'entrada de les portes que hi estan connectades (i en la dels fils que les uneixen), que es comporta com un condensador C. El circuit equivalent i la gràfica de tensió $V(t) = V_{CC}(1 - e^{-t/RC})$ són a la Figura 6.4:

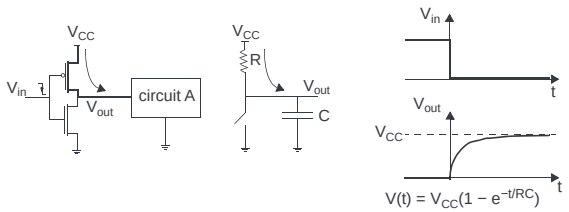


Figura 6.4: Evolució temporal de la tensió de sortida de la porta NOT quan el valor lògic de l'entrada passa d'1 a 0.

Descàrrega: Quan la tensió a l'entrada de la porta V_{in} passa del valor lògic 0 a l'1 (funció graó de flanc ascendent), el transistor NMOS passa a conduir i el PMOS queda obert. En conduir, el corrent flueix a través de la seva resistència equivalent R, i la càrrega elèctrica prèviament acumulada en les etapes d'entrada del circuit A (condensador C) es descarrega cap a terra. El circuit equivalent i la gràfica de tensió $V(t) = V_{CC} \cdot e^{-t/RC}$ són a la Figura 6.5:

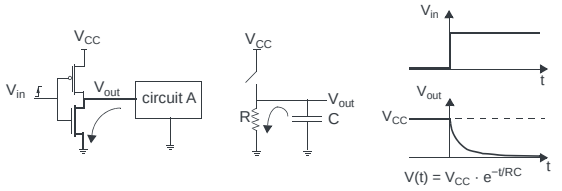


Figura 6.5: Evolució temporal de la tensió de sortida de la porta NOT quan el valor lògic de l'entrada passa de 0 a 1.

6.3.2 Potència estàtica

A diferència de la dinàmica, la potència estàtica (P_s) no depèn de la commutació de les portes. P_s es dissipa contínuament, fins i tot quan el circuit no canvia d'estat. El seu origen són els corrents de fuita (*leakage currents*): petits corrents que travessen els transistors teòricament tallats, ja que cap transistor real es comporta com un interruptor ideal.

Els corrents de fuita tenen dos orígens principals:

- **Fuita subllindar** (*subthreshold leakage*): corrent que circula entre drenador i font d'un transistor amb una tensió de porta per sota del llindar V_t. Creix a mesura que aquest llindar és més petit, i els transistors miniaturitzats el redueixen per poder funcionar a tensions V_{CC} més baixes.
- **Fuita de porta** (*gate leakage*): corrent que travessa l'òxid aïllant de la porta, cada cop més prim a mesura que es miniaturitza el transistor.

Donada la tensió d'alimentació V_{CC} i el corrent de fuita agregat de tot el circuit I_{fuita}, la potència estàtica és el seu producte:

$$P_s = V_{CC} \cdot I_{fuita} \tag{6.11}$$

A diferència de la potència dinàmica (Equació 6.9), la potència estàtica no depèn de la freqüència de rellotge: un circuit alimentat però inactiu continua dissipant P_s. A més, I_{fuita} creix amb la temperatura i amb la miniaturització dels transistors, de manera que la seva contribució a la potència total ha anat guanyant pes en cada nova generació.

6.3.3 Miniaturització i límit tèrmic

L'any 1965 Gordon Moore (cofundador d'Intel) va predir que **la densitat de transistors al xip es duplicaria cada any**, i el 1975 va revisar la previsió a **cada dos anys** (l'«any i mig» que se sol citar és una estimació posterior, que parla del rendiment i no de la densitat). Aquesta predicció es coneix com a **llei de Moore** (tot i que en realitat és una observació empírica). Durant dècades, aquesta duplicació de transistors va alimentar dos grans motors de rendiment: l'augment de la freqüència de rellotge i el creixement de les memòries cau (Tema 7).

La miniaturització que fa possible la llei de Moore permet reduir el temps de commutació dels transistors i, per tant, augmentar la freqüència de rellotge. Però a mesura que la freqüència creixia, també ho feia la **dissipació de potència** (Equació 6.9), ja que l'energia consumida es converteix en calor que cal dissipar per evitar superar els llindars de tolerància del circuit. A això cal afegir-hi la gran demanda de dispositius mòbils (telèfons intel · ligents, tauletes, portàtils ultraprims) que disposen d'un espai molt limitat per a dissipadors i que necessiten un consum molt baix per preservar l'autonomia de la bateria. El resultat és que el creixement de la freqüència de rellotge es va estancar cap al 2003–2005: el creixement dels corrents de fuita (Secció 6.3.2) va posar fi a l'**escalat de Dennard** (Aprofundiment 6.2).

> ⚠ Aprofundiment 6.2: L'escalat de Dennard i la fi del creixement de freqüència
>
> Durant dècades, la reducció de la mida dels transistors no sols permetia encabir-ne més al xip (llei de Moore), sinó que, gràcies a l'escalat de Robert Dennard (IBM), també mantenia la **densitat de potència constant**: transistors més petits consumien proporcionalment menys energia, de manera que duplicar-ne la quantitat no disparava el consum total. Això permetia augmentar la freqüència de rellotge sense problemes tèrmics.
> Cap al 2003–2005, però, l'escalat de Dennard va deixar de funcionar. En arribar a dimensions de transistor molt petites, els **corrents de fuita** (I_{fuita}, Equació 6.11) ja no eren negligibles i creixien amb cada nova generació, tot i que la freqüència no augmentés. A partir d'aquí, augmentar la freqüència implicava disparar la potència total per sobre dels límits tèrmics tolerables. Des d'aleshores, la freqüència sostinguda dels processadors s'ha estancat entre els 3 i els 5 GHz, amb pics puntuals de fins a uns 6 GHz en mode *turbo*.
> La resposta de la indústria va ser el disseny **multinucli** (*multicore*): en lloc d'un sol nucli molt ràpid, es van integrar múltiples nuclis de freqüència moderada en un mateix xip, mantenint la potència total sota control mentre es continuava augmentant la productivitat. Això explica que avui dia un portàtil d'oficina i un servidor de centre de dades tinguin freqüències de rellotge similars (3–5 GHz), però el servidor tingui molts més nuclis. La distribució física d'aquests nuclis en un xip modern es pot apreciar al Tema 7 (Figura 7.25).

> ⚠ Aprofundiment 6.3: Tècniques de reducció del consum
>
> Les equacions de la potència (Equació 6.9 i Equació 6.11) diuen on es pot actuar per reduir el consum. Els processadors actuals combinen tres tècniques, cadascuna dirigida a un terme diferent:
>
> - **Escalat dinàmic de tensió i freqüència** (*dynamic voltage and frequency scaling*, DVFS): quan la càrrega de treball és baixa, el processador redueix la freqüència de rellotge f_{clock} i,

amb ella, la tensió d'alimentació V_{CC}, que es pot rebaixar perquè a menor freqüència els transistors disposen de més temps per commutar. Com que P_d és proporcional a $V_{CC}^2 \cdot f_{clock}$, reduir totes dues alhora redueix la potència dinàmica més que proporcionalment: amb un 15 % menys de tensió i de freqüència, la potència dinàmica queda multiplicada per $0{,}85^3 \approx 0{,}61$, gairebé un 40 % menys.

- **Inhibició del rellotge** (*clock gating*): s'atura el senyal de rellotge de les unitats que no s'usen en un moment donat (per exemple, la unitat de coma flotant mentre el programa només opera amb enters). Les seves portes deixen de commutar, i es redueix el factor d'activitat α.
- **Desconnexió de l'alimentació** (*power gating*): es talla l'alimentació de les unitats inactives durant períodes llargs. És l'única de les tres que elimina la potència estàtica, perquè sense tensió no hi ha corrents de fuita (I_{fuita}).

Cap d'aquestes tècniques no és gratuïta. Reduir la freqüència allarga el temps d'execució (Equació 6.2), i reactivar una unitat desconnectada costa temps i energia. Pel que fa a l'energia que consumeix una tasca (Equació 6.10), reduir només la freqüència no n'estalvia: la potència dinàmica baixa en la mateixa proporció en què augmenta el temps. L'estalvi d'energia del DVFS ve de la reducció de la tensió.

Tema 7: Memòria cau

7.1 La jerarquia de memòria

7.1.1 Tecnologies de memòria

Tot computador necessita emmagatzemar les instruccions i les dades dels programes que executa. Aquesta funció la cobreixen diverses tecnologies, cadascuna amb característiques pròpies de velocitat, capacitat i cost:

- La **memòria secundària**, feta de discs durs magnètics o d'unitats d'estat sòlid (*solid-state drives*, SSD), és la memòria **persistent**: conserva les dades quan el computador s'apaga. Té un cost per bit molt baix i una capacitat molt gran, però un temps d'accés extremadament lent: típicament més de 5 ms en un disc dur i entre 0,05 i 0,1 ms en un SSD. Per a un període de rellotge de CPU típic de 0,25 ns, un accés a SSD tardaria centenars de milers de cicles; un accés a disc dur, desenes de milions.
- La **memòria principal** (MP), una *random access memory* (RAM), és de tipus **volàtil**: perd el contingut quan s'apaga el computador. Quan s'executa un programa, les seves instruccions i dades es carreguen des de la memòria secundària a la memòria principal, on resideixen durant l'execució. La RAM té un temps d'accés molt més curt que la memòria secundària (entre 50 i 100 ns), però el seu cost per bit és considerablement més alt.

La RAM es pot implementar com a RAM estàtica (*static random access memory*, SRAM) o com a RAM dinàmica (*dynamic random access memory*, DRAM). Cada bit d'SRAM requereix almenys sis transistors, mentre que un bit de DRAM n'ocupa un de sol més un condensador. Això fa que la DRAM tingui molta més capacitat per unitat de superfície i un cost per bit molt menor, i per aquest motiu s'usa habitualment per a la memòria principal. L'SRAM, en canvi, és molt més ràpida i s'usa en contextos on la velocitat és prioritària, com ara les memòries cau.

Taula 7.1: Capacitat, temps d'accés i cost per GB típics de les principals tecnologies d'emmagatzematge

Tecnologia	Capacitat típica	Temps d'accés	Cost per GB
Registres (SRAM)	~1 KiB	~0,25 ns	—
Memòria cau (SRAM)	32 KiB – 64 MiB	0,5 – 5 ns	~3000 $/GB
Mem. principal (DRAM)	8 – 128 GiB	50 – 100 ns	~3 $/GB
Mem. secundària (SSD)	256 GB – 8 TB	0,05 – 0,1 ms	~0,08–0,15 $/GB
Mem. secundària (disc)	1 – 20 TB	5 – 10 ms	~0,02 $/GB

7.1.2 El problema de la bretxa (*gap*) entre processador i memòria

Al llarg dels anys, el rendiment dels processadors ha crescut molt més ràpidament que la velocitat de la memòria principal. La Figura 7.1 il · lustra aquesta divergència entre 1980 i 2010: mentre el rendiment dels processadors creixia a un ritme d'entre el 35 % i el 55 % anual fins al 2000, i molt més lentament a partir del 2003–2005 (Secció 7.1.2.1), la latència de la DRAM millorava menys d'un 7 % anual. El resultat és que el temps d'accés a la memòria principal pot arribar a ser de 200 a 400

cicles de CPU, és a dir, centenars de vegades superior al temps que tarda la resta d'operacions d'una instrucció (descodificació, lectura de registres, ALU, etc.). Aquesta diferència s'anomena **bretxa** (*gap*) entre processador i memòria.

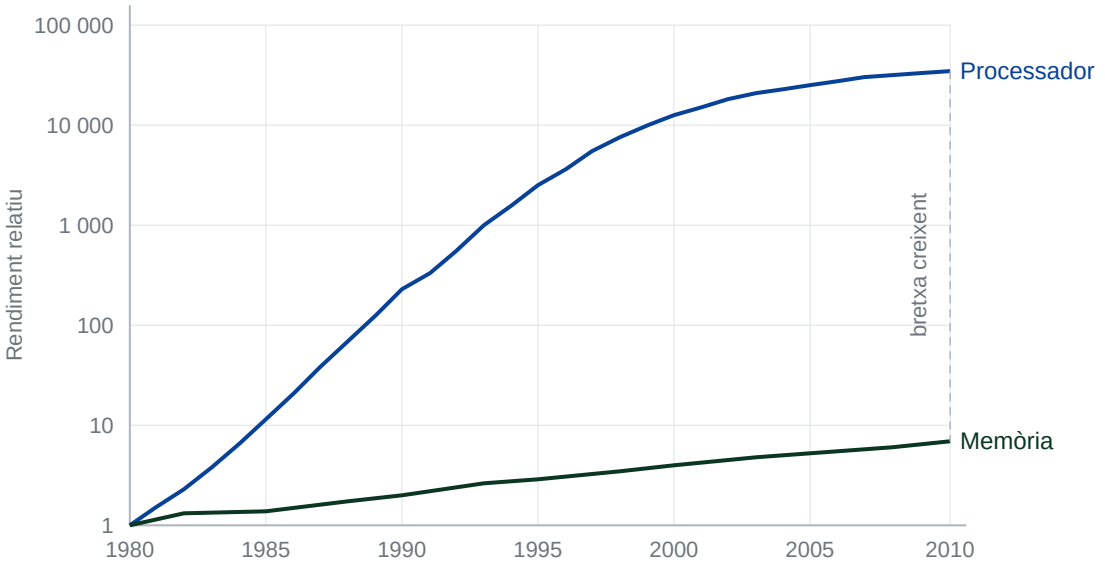


Figura 7.1: Evolució del rendiment relatiu de processadors i memòria DRAM entre 1980 i 2010 (escala logarítmica). El creixement anual dels processadors (35–55 % fins al 2000, molt més lent des del 2003–2005) supera àmpliament el de la latència de la DRAM (<7 %), cosa que genera una bretxa de més de tres ordres de magnitud.

Per veure l'impacte d'aquesta bretxa en el rendiment, s'aplica la llei d'Amdahl (Secció 6.2). Si t_{MP} és el temps d'accés a la memòria principal i t_{proc} el temps de la resta d'operacions d'una instrucció, el **CPI** és proporcional a $t_{proc} + t_{MP}$. Suposant $t_{MP} = 200 \cdot t_{proc}$, la fracció del temps d'execució que correspon al processament de la instrucció (sense l'accés a memòria) és:

$$P_x = \frac{t_{proc}}{t_{proc} + t_{MP}} \approx 0{,}005$$

Per la llei d'Amdahl, el màxim guany que es pot obtenir millorant únicament el processador (sense tocar la memòria) és:

$$s_{max} = \frac{1}{1 - P_x} \approx 1{,}005$$

És a dir, menys d'un 1 %. Millorar el processador sense atacar el problema de la memòria és pràcticament inútil.

7.1.2.1 El punt d'inflexió: la fi de l'escalat de Dennard

Cap al 2003–2005, la indústria va topar amb un límit físic fonamental: el **límit tèrmic**. Fins aleshores, reduir la mida dels transistors permetia augmentar la freqüència de rellotge mantenint la mateixa densitat de potència (un principi conegut com a escalat de Dennard, Aprofundiment 6.2). Quan aquest escalat va deixar de funcionar, augmentar la freqüència va implicar disparar el consum i la calor

generada fins a nivells insostenibles. Des d'aleshores, la freqüència sostinguda dels processadors s'ha estancat entre els 3 i els 5 GHz, amb pics puntuals de fins a uns 6 GHz en mode *turbo*.

La indústria va respondre amb dues estratègies complementàries:

- Els processadors **multinucli** (*multicore*): en lloc d'un sol nucli molt ràpid, es van integrar dos, quatre, vuit o més nuclis en un mateix xip. Això va permetre continuar aprofitant els transistors addicionals que oferia cada nova generació tecnològica per augmentar el **rendiment en paral · lel**, però sense millorar el rendiment d'un sol fil d'execució (*single-thread*), que és el que determina la bretxa amb la memòria. Des del 2010, el creixement del rendiment d'un sol fil ha caigut a un modest 5–10 % anual, molt per sota del ritme anterior.
- La **reducció del consum energètic**. La generalització dels dispositius mòbils —telèfons intel · ligents, tauletes, portàtils ultraprims— va convertir l'eficiència energètica en una prioritat de primer ordre: no es tracta només d'impacte ambiental, sinó de l'autonomia de la bateria i de la possibilitat de dissenyar dispositius sense ventilador. Avui dia, l'arquitectura de qualsevol processador modern es dissenya tant per al rendiment com per al consum per instrucció.

7.1.2.2 La diferència entre processador i memòria avui

Malgrat tot, la bretxa entre processador i memòria no ha desaparegut: la latència de la DRAM segueix sent de l'ordre de centenars de cicles, i la DRAM tampoc ha resolt el problema de fons (continua optimitzant capacitat per davant de velocitat). El creixement dels processadors multinucli augmenta, a més, la **pressió sobre la memòria**: diversos nuclis competeixen simultàniament pels mateixos recursos de memòria, de manera que l'**amplada de banda** (*bandwidth*) disponible per nucli es redueix. En aquest context, **les memòries cau continuen sent tan imprescindibles com sempre**, i les jerarquies s'han fet cada cop més profundes i complexes per absorbir aquesta pressió creixent (vegeu la Secció 7.9).

⚠ Aprofundiment 7.1: Les GPU i la memòria en l'era de la IA

Qui tingui una targeta gràfica per a jocs —o hagi llegit sobre els acceleradors que entrenen models d'intel · ligència artificial com els que hi ha darrere de ChatGPT— potser s'ha preguntat com funciona la seva memòria. La resposta és fascinant, i il · lustra fins on arriba la bretxa quan s'escalen les demandes.

Una GPU moderna per a jocs (per exemple, una NVIDIA RTX 4090) integra milers de nuclis de càlcul dissenyats per executar moltíssimes operacions en paral · lel. Per alimentar tots aquests nuclis, necessita una amplada de banda de memòria enorme: mentre que un processador convencional accedeix a la DRAM amb una amplada de banda de l'ordre de desenes de GB/s, una GPU d'alt rendiment en necessita centenars de GB/s, o fins i tot més d'un TB/s. Per aconseguir-ho s'usa memòria **GDDR6** o **GDDR6X** o, en els acceleradors d'IA professionals, *high bandwidth memory* (HBM), una tecnologia que apila diverses capes de DRAM l'una sobre l'altra, en una pila que es col · loca al costat del xip de càlcul, dins del mateix encapsulat, i s'hi connecta amb un bus de milers de fils.

La latència, però, segueix sent el problema de sempre: la GDDR6 augmenta l'**amplada de banda** amb una velocitat molt alta per fil, i l'HBM, encara més, amb la proximitat física i un bus molt ample; però la latència d'accés roman comparable a la de la DRAM convencional. Per això les GPU modernes també incorporen **jerarquies de memòria cau** pròpies: cada grup de nuclis (*streaming multiprocessor* a NVIDIA, *compute unit* a AMD) té la seva pròpia memòria cau de primer nivell (L1), i tots comparteixen una de segon nivell (L2) de desenes de MiB.

Els acceleradors d'IA porten aquesta lògica a l'extrem. En entrenar una xarxa neuronal, s'han de moure quantitats ingents de dades (pesos del model, activacions, gradients) entre la memòria i els

nuclis de càlcul milers de vegades per segon. El coll d'ampolla no sol ser la velocitat de càlcul sinó precisament l'**amplada de banda de memòria**: és el que determina quants paràmetres del model es poden processar per segon. Això ha portat a dissenyar xips com l'H100 d'NVIDIA amb fins a 80 GiB d'HBM3 i una amplada de banda de 3,35 TB/s, un valor que hauria semblat de ciència-ficció fa una dècada.
En definitiva: la bretxa entre càlcul i memòria no és un problema exclusiu dels processadors convencionals, sinó un repte fonamental de tota l'arquitectura de computadors moderna, especialment visible en les aplicacions d'IA que definiran la computació dels anys vinents.

7.1.3 Localitat dels programes

La solució a la bretxa no consisteix a construir una memòria principal tan ràpida com el processador —el cost seria prohibitiu— sinó a explotar una propietat que presenten la majoria dels programes: la **localitat**. L'estudi del comportament dels programes mitjançant simulació ha permès identificar-ne dues formes:

- **Localitat temporal**: si s'accedeix a una adreça de memòria, és probable que s'hi torni a accedir en un futur pròxim. La causa principal és l'existència de **bucles**: s'accedeix repetidament a les mateixes instruccions i dades a cada iteració.
- **Localitat espacial**: si s'accedeix a una adreça de memòria, és probable que s'accedeixi a adreces properes en un futur pròxim. En el cas de les instruccions, la causa és el **seqüenciament implícit**: llevat dels salts, el processador sempre incrementa el PC i busca la instrucció següent en memòria. En el cas de les dades, la localitat espacial és característica dels **recorreguts de vectors i matrius** emmagatzemats en posicions contigües.

7.1.4 La jerarquia de memòria

La localitat permet una solució elegant a la bretxa: interposar entre el processador i la memòria principal una memòria petita i ràpida que retingui les dades a què és més probable que s'accedeixi. Si la majoria d'accessos es resolen en aquesta memòria ràpida, el temps d'accés mitjà s'aproximarà al de la memòria ràpida, no al de la lenta.

Aplicant la mateixa idea de forma recursiva —una jerarquia de nivells, cada un més gran i més lent que l'anterior— s'obté el subsistema de memòria tal com s'implementa als processadors actuals (Figura 7.2):

Cada nivell emmagatzema un subconjunt del contingut del nivell immediatament inferior (és una jerarquia inclusiva, la que s'estudia a EC). Quan el processador necessita una dada, la cerca primer al nivell més ràpid (la memòria cau); si no hi és, baixa al nivell següent, i així successivament. La **memòria cau** (MC) és el nom que rep el nivell que s'interposa entre el processador i la memòria principal; és la peça central d'aquest tema.

A banda del rendiment, la memòria cau té també un impacte positiu en el **consum energètic**: com que la majoria dels accessos es resolen a l'MC sense necessitat d'accedir a la memòria principal, el nombre total d'operacions que activen els circuits de la DRAM disminueix, i amb ell el factor d'activitat α de la fórmula de potència dinàmica $P_d = \alpha \cdot C \cdot V_{CC}^2 \cdot f_{clock}$ (Equació 6.9).

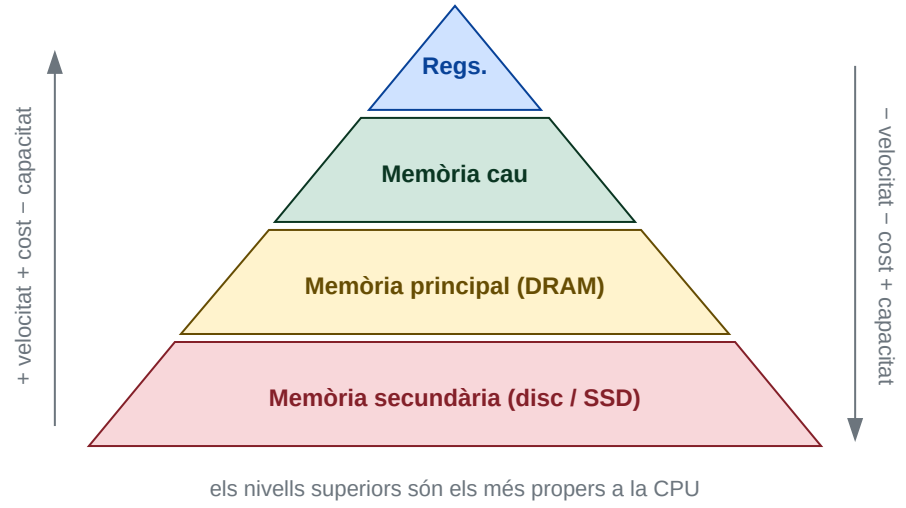


Figura 7.2: La jerarquia de memòria: cada nivell conserva les dades del nivell inferior que tinguin més probabilitat de ser accedides.

7.2 La memòria cau

7.2.1 Principi de funcionament

La **memòria cau** (*cache*) és una memòria auxiliar petita i ràpida que s'interposa entre el processador i la memòria principal. Emmagatzema un subconjunt del contingut de la memòria principal, amb l'objectiu de retenir les dades a què és més probable que s'accedeixi aviat:

- Explota la **localitat temporal**: cada cop que la CPU accedeix a una dada de la memòria principal, en guarda simultàniament una còpia a l'MC. Si la mateixa dada es torna a referenciar i encara és a l'MC, l'accés es resol sense necessitat d'anar a la memòria principal.
- Explota la **localitat espacial**: cada cop que la CPU accedeix a una dada, l'MC no en copia únicament la dada individual sinó tot el **bloc** al qual pertany (un fragment contigu de memòria de mida fixa). Si posteriorment s'accedeix a una dada propera, és probable que ja sigui al bloc copiat i l'accés es torni a resoldre ràpidament.

💡 Exemple 7.1: Analogia de la memòria cau amb l'escriptori

Un estudiant d'història va a l'arxiu de la seva ciutat per elaborar un treball de recerca. Els documents que necessita consultar es troben agrupats en carpetes a les prestatgeries de l'arxiu (la memòria principal). Cada cop que necessita un document, es desplaça a la prestatgeria, el consulta i torna al seu lloc de treball.
Com que alguns documents s'han de consultar repetidament (localitat temporal), l'estudiant decideix emportar-se'ls a l'escriptori (la memòria cau) per estalviar viatges. A més, com que els documents que necessita sovint es troben dins les mateixes carpetes (localitat espacial), decideix que cada cop que va a la prestatgeria s'endú la carpeta sencera, no tan sols el document concret.

7.2.2 Organització: blocs, línies, etiquetes i bit de validesa

L'MC no guarda dades soltes sinó **blocs** sencers de mida fixa B bytes, on B és una potència de 2 (les mides típiques oscil · len entre 16 i 128 bytes). L'espai d'adreçament de la memòria principal es

subdivideix lògicament en blocs de B bytes: l'adreça inicial de cada bloc és un múltiple de B, i cada bloc té un identificador únic anomenat **número de bloc**, que s'obté numerant correlativament tots els blocs des de zero.

Internament, l'MC s'organitza com una taula. Cada entrada d'aquesta taula s'anomena **línia** i pot contenir un bloc de dades o estar buida. Per a cada línia, l'MC emmagatzema tres elements (Figura 7.3):

- El **bit de validesa** (V, *valid bit*): indica si la línia conté un bloc vàlid ($V = 1$) o està buida ($V = 0$).
- L'**etiqueta** (*tag*): identifica quin bloc de la memòria principal hi ha guardat a la línia. Està formada pels bits del número de bloc (o una part d'aquests, com es veurà a la Secció 7.3).
- Les **dades**: el contingut del bloc de memòria que s'ha copiat a la línia.

#	V	Etiqueta	Dades (16 bytes)
0	1	0x100100F	bytes 0x100100F0–0x100100FF
1	1	0x7FFFF01	bytes 0x7FFFF010–0x7FFFF01F
2	1	0x1001010	bytes 0x10010100–0x1001010F
3	0	–	

Figura 7.3: Organització interna d'una memòria cau de quatre línies. Cada línia conté un bit de validesa (V), una etiqueta i un bloc de dades.

El **número de bloc** (identificador de bloc) i el **desplaçament** (posició de la dada dins el bloc, ***offset***) s'obtenen a partir de l'adreça A de la dada a la memòria principal mitjançant la divisió i el mòdul de B, respectivament:

$$\text{num_bloc} = A \div B$$

$$\text{desplaçament} = A \bmod B$$

Com que $B = 2^b$, aquestes operacions es redueixen a una simple descomposició de bits: els b bits de menys pes de A són el desplaçament, i la resta és el número de bloc.

Exemple 7.2

Pregunta: Suposem blocs de mida $B = 16$ bytes ($b = 4$) i una adreça $A =$ `0x100100F8`. Determina el número de bloc i el desplaçament.
Solució: Descomponem l'adreça en binari separant els 4 bits de menys pes:

$$A = \underbrace{0001\,0000\,0000\,0001\,0000\,0000\,1111}_{\text{num_bloc=0x100100F}} \quad \underbrace{1000}_{\text{desplaçament=8}}$$

Si l'accés és a una paraula de 4 bytes (`lw`), el **desplaçament de paraula** (posició de la paraula dins el bloc) és:

$$\text{desplaçament de paraula} = \text{desplaçament} \div 4 = 2$$

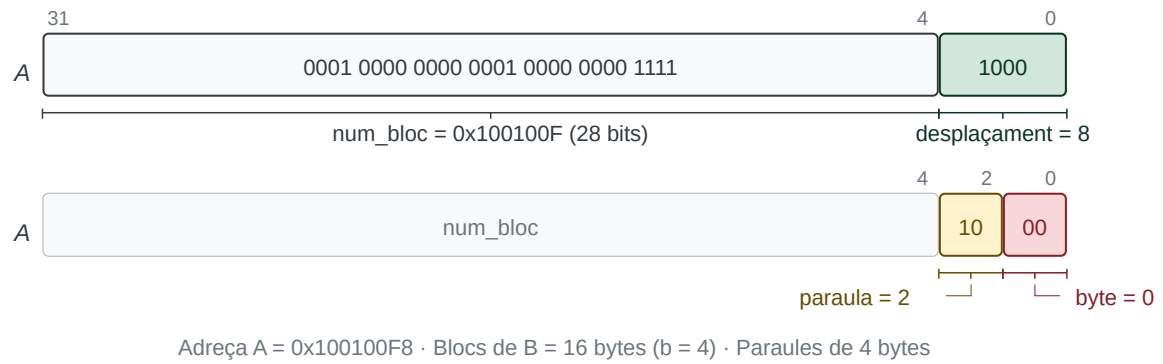


Figura 7.4: Descomposició de l'adreça `0x100100F8` amb blocs de 16 bytes: número de bloc i desplaçament; i el desplaçament, en paraula dins del bloc i byte dins de la paraula.

7.2.3 Terminologia de rendiment

En accedir a una dada, l'MC comprova si el bloc que la conté és present:

- **Encert** (*hit*): la dada és a l'MC i l'accés es resol en un temps curt t_h, característic de l'MC.
- **Fallada** (*miss*): la dada no és a l'MC. Cal accedir a la memòria principal per obtenir el bloc, cosa que afegeix un **temps de penalització** t_p al temps d'accés total:

$$t_{acc} = \begin{cases} t_h & \text{(encert)} \\ t_h + t_p & \text{(fallada)} \end{cases}$$

S'anomenen **taxa d'encerts** (h) i **taxa de fallades** (m) les fraccions d'accessos que acaben en encert i en fallada respectivament:

$$h = \frac{\text{núm. encerts}}{\text{núm. referències}} \qquad m = \frac{\text{núm. fallades}}{\text{núm. referències}} = 1 - h$$

7.2.4 Gestió d'una lectura

L'operació d'una lectura il · lustra la mecànica bàsica de l'MC. Siguin blocs de 16 bytes i una lectura de la CPU a l'adreça $A =$ `0x100100F8` (num_bloc = `0x100100F`, desplaçament = 8).

Cas d'encert (Figura 7.5): el bloc és present a l'MC.

1. L'MC consulta les etiquetes de les línies vàlides i troba que el num_bloc de A coincideix amb l'etiqueta d'una línia. S'ha produït un encert.
2. L'MC usa el desplaçament com a selector per extreure la dada sol · licitada del bloc guardat i la transfereix a la CPU.

Cas de fallada (Figura 7.6): el bloc no és present a l'MC.

1. L'MC comprova les etiquetes i no troba el num_bloc de A. S'ha produït una fallada: la CPU es bloqueja fins que l'MC resolgui l'accés.
2. L'MC sol · licita a la memòria principal el bloc sencer que conté A, enviant-li l'adreça inicial del bloc (`0x100100F0`, que s'obté posant a zero els b bits de desplaçament de A).



Figura 7.5: Encert en un accés a l'adreça `0x100100F8`: la dada es transfereix directament de l'MC a la CPU.

3. L'MC decideix en quina línia emmagatzema el bloc rebut, hi escriu les dades, n'actualitza l'etiqueta i posa el bit de validesa a 1.
4. L'MC finalitza l'accés com si fos un encert: usa el desplaçament per extreure la dada i la transfereix a la CPU.

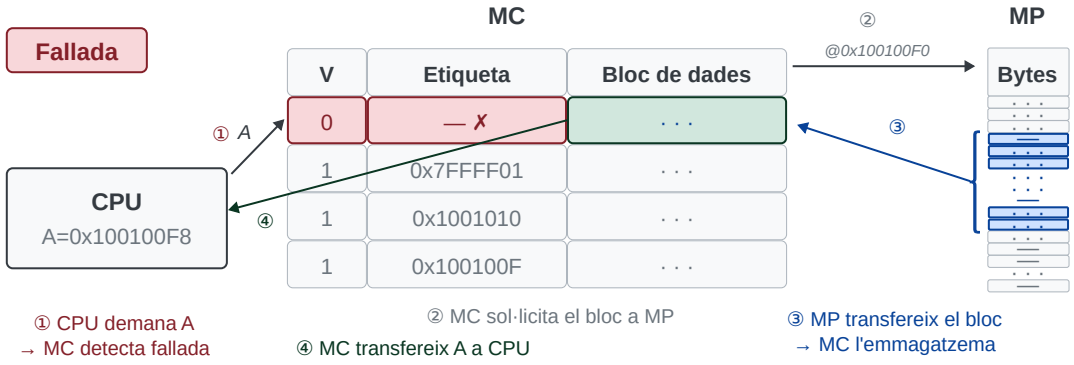


Figura 7.6: Fallada en un accés a l'adreça `0x100100F8`: el bloc es copia de l'MP a l'MC i la dada es transfereix a la CPU.

7.3 Polítiques d'emplaçament

Quan l'MC ha de guardar un bloc procedent de la memòria principal, cal decidir en quina línia l'emmagatzema. Aquest és el problema que resol la **política d'emplaçament** (*placement policy*). La política d'emplaçament determina també on cal cercar un bloc quan la CPU hi vol accedir: si se sap en quines línies pot estar, la cerca és més ràpida.

Hi ha tres polítiques possibles, que formen un espectre de menys a més flexibilitat. Com es veurà, la flexibilitat té un cost en complexitat de maquinari, i el disseny real és sempre un compromís entre els dos factors.

7.3.1 Correspondència directa

En una memòria cau de **correspondència directa** (*direct-mapped*), cada bloc de la memòria principal té assignada **una única línia** on pot ser emmagatzemat. L'assignació es fa per mòdul: si l'MC té $N_L = 2^{n_l}$ línies, el bloc amb número de bloc n va sempre a la línia:

$$\text{índex} = n \bmod N_L$$

Com que N_L és potència de 2, el mòdul és simplement els n_l bits de menys pes del número de bloc. Combinant-ho amb la descomposició de bits de l'adreça de la Secció 7.2.2, l'adreça queda dividida en tres camps:

$$A = \underbrace{\cdots}_{\text{etiqueta}} \quad \underbrace{\cdots}_{\text{índex } (n_l \text{ bits})} \quad \underbrace{\cdots}_{\text{desplaçament } (b \text{ bits})}$$

Com que tots els blocs que es mapen a una mateixa línia comparteixen el mateix índex, l'etiqueta pot prescindir d'aquests bits: n'hi ha prou amb els bits restants del número de bloc per identificar unívocament el bloc guardat (Figura 7.7).

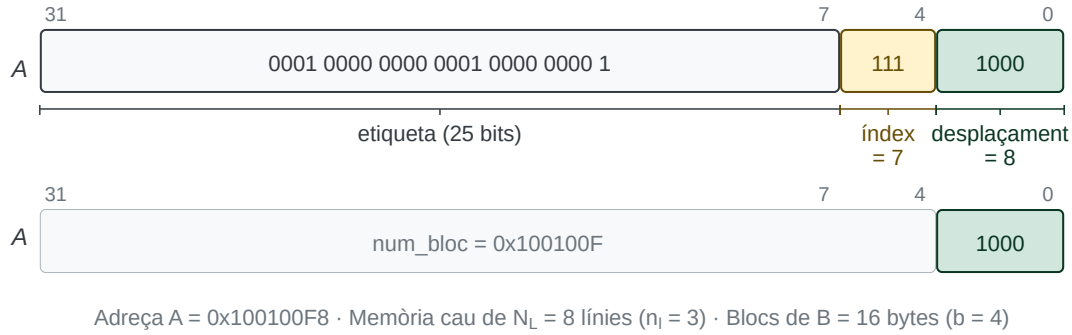


Figura 7.7: Descomposició dels bits d'una adreça en etiqueta, índex i desplaçament per a una memòria cau de correspondència directa amb $N_L = 8$ línies i blocs de $B = 16$ bytes.

Exemple 7.3

Pregunta: Suposem una memòria cau de correspondència directa amb $N_L = 8$ línies ($n_l = 3$) i blocs de $B = 16$ bytes ($b = 4$). A quina línia s'emmagatzema el bloc que conté l'adreça $A =$ `0x100100F8`?
Solució:

$$\text{num_bloc} = A \div 16 = \texttt{0x100100F}$$

$$\text{índex} = \texttt{0x100100F} \bmod 8 = 7$$

El bloc s'emmagatzema a la línia 7. En binari, l'índex correspon als 3 bits de menys pes del num_bloc:

$$A = \underbrace{0001\,0000\,0000\,0001\,0000\,0000\,1}_{\text{etiqueta}} \quad \underbrace{111}_{\text{índex}=7} \quad \underbrace{1000}_{\text{desplaçament}=8}$$

El diagrama de blocs de la Figura 7.8 mostra el maquinari necessari per a una lectura. Els bits d'índex de l'adreça seleccionen una línia mitjançant un **descodificador**. Es llegeixen simultàniament l'etiqueta guardada, el bit de validesa i el bloc de dades. Els bits d'etiqueta de l'adreça es comparen amb l'etiqueta guardada: si coincideixen i $V = 1$, es produeix un encert i el **multiplexor** selecciona la dada indicada pel desplaçament.

La correspondència directa és la política més senzilla i ràpida: amb un sol accés a la taula i una sola comparació es sap si hi ha encert o fallada. El seu inconvenient és la **rigidesa**: si s'accedeix alternament

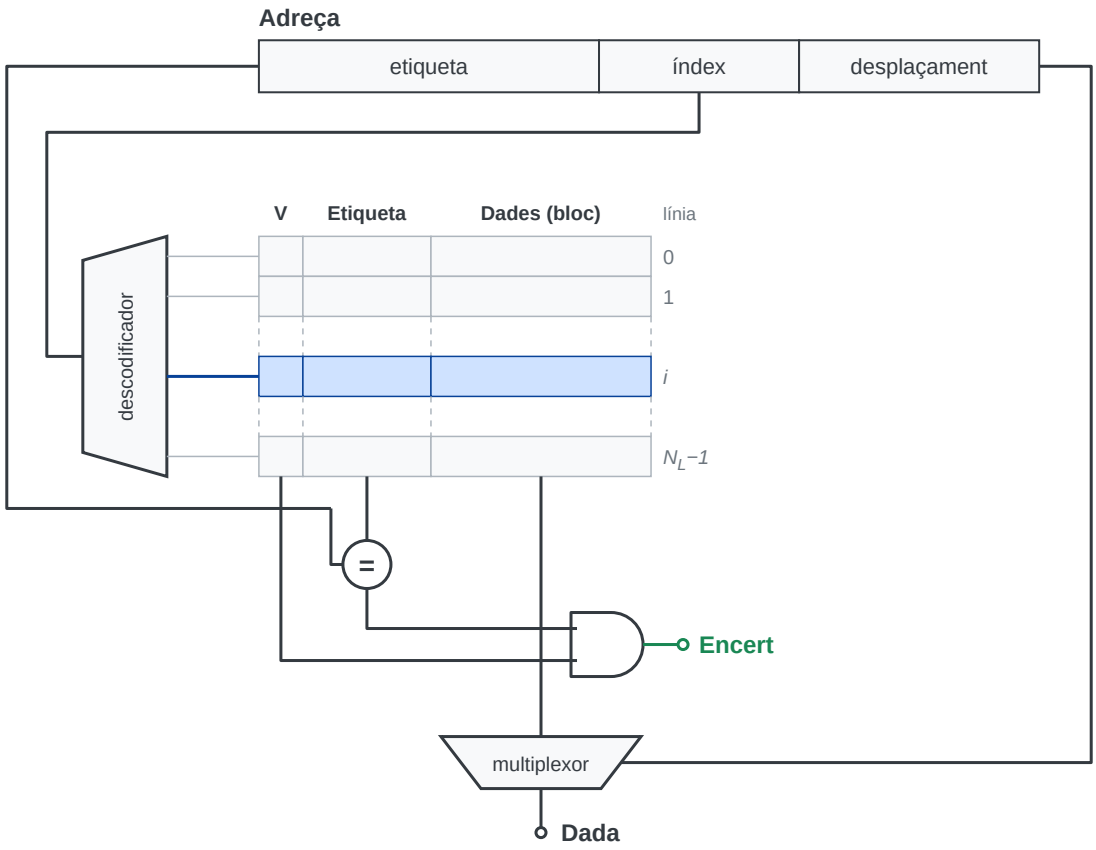


Figura 7.8: Diagrama de blocs d'una lectura en una memòria cau de correspondència directa.

a dos blocs que es mapen a la mateixa línia, s'expulsaran mútuament a cada accés i causaran fallades contínues, fins i tot si la resta de línies estan buides. Això s'anomena **conflicte** i s'estudia en detall a la Secció 7.8.

7.3.2 Memòria cau associativa per conjunts

La **memòria cau associativa per conjunts** (*set-associative*) redueix els conflictes dotant cada posició de la memòria cau de N **vies** (*ways*) en lloc d'una sola. El conjunt de N vies associades a una mateixa posició s'anomena **conjunt** (*set*), i N és el **grau d'associativitat**.

L'assignació de blocs als conjunts segueix el mateix criteri que la correspondència directa: si la memòria cau té $N_C = 2^{n_c}$ conjunts, el bloc n va al conjunt d'índex n mod N_C. Però dins del conjunt, el bloc pot ocupar **qualsevol de les N vies** que estigui lliure. Si no n'hi ha cap de lliure, s'aplica un algorisme de reemplaçament (vegeu la Secció 7.4). La descomposició de bits és anàloga a la de correspondència directa, però amb n_c bits d'índex en lloc de n_l:

$$A = \underbrace{\cdots}_{\text{etiqueta}} \quad \underbrace{\cdots}_{\text{índex } (n_c \text{ bits})} \quad \underbrace{\cdots}_{\text{desplaçament } (b \text{ bits})}$$

Exemple 7.4

Pregunta: Suposem adreces de 8 bits, blocs de $B = 4$ bytes ($b = 2$) i una memòria cau associativa per conjunts, amb 4 conjunts ($n_c = 2$) de 3 vies. A quin conjunt pertany el byte de l'adreça `0b00000100` (adreça 4)?
Solució:

$$\text{num_bloc} = 4 \div 4 = 1$$

$$\text{índex} = 1 \bmod 4 = 1$$

El bloc s'ha d'emmagatzemar al conjunt 1, però pot ocupar qualsevol de les 3 vies que estigui lliure (Figura 7.9).

$$A = \underbrace{0000}_{\text{etiqueta}} \quad \underbrace{01}_{\text{índex=1}} \quad \underbrace{00}_{\text{desplaçament=0}}$$

El diagrama de blocs de la Figura 7.10 mostra que els bits d'índex seleccionen el conjunt mitjançant un **descodificador** (com a la correspondència directa), però dins del conjunt hi ha N **comparadors en paral·lel**, un per via. Si alguna via té la mateixa etiqueta i $V = 1$, una porta OR genera el senyal d'encert i un **multiplexor N:1** selecciona les dades de la via guanyadora, que el segon multiplexor (controlat pel desplaçament) usa per extreure la dada sol·licitada.

La correspondència directa és el cas particular $N = 1$: una sola via per conjunt (cada conjunt és una línia), un sol comparador, sense necessitat de multiplexor de vies.

7.3.3 Memòria cau completament associativa

La **memòria cau completament associativa** (*fully associative*) porta la flexibilitat al límit: un bloc pot emmagatzemar-se a **qualsevol línia** de la memòria cau, sense cap restricció d'índex. Això elimina completament els conflictes (vegeu la Secció 7.8), però té un cost important: per saber si un bloc és present, cal comparar l'etiqueta amb **totes** les línies simultàniament.

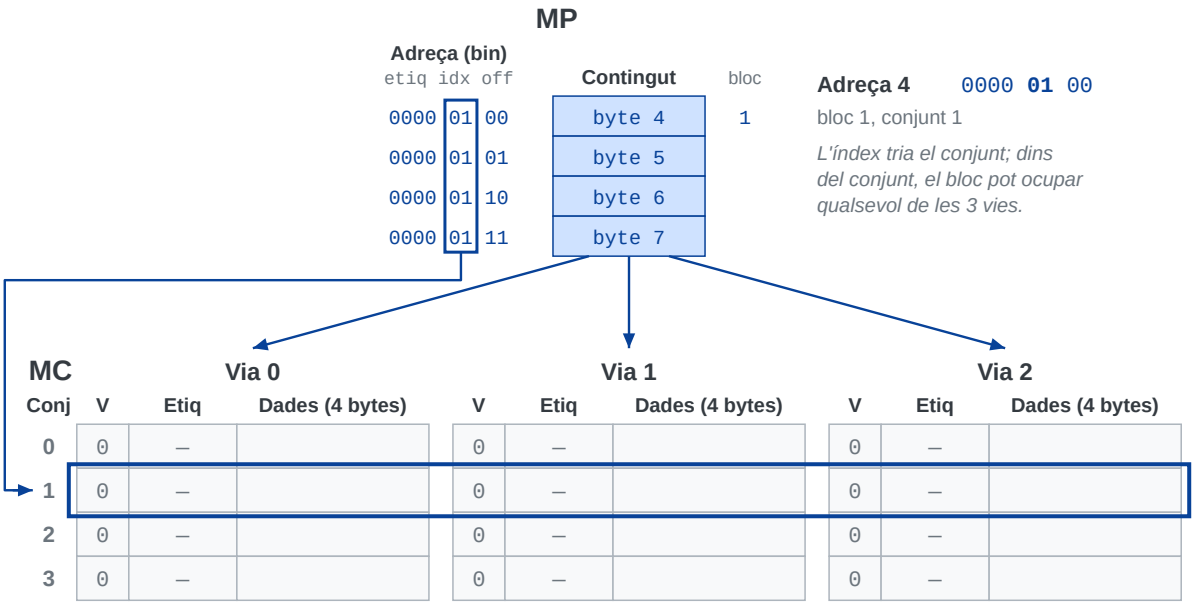


Figura 7.9: Organització d'una memòria cau associativa per conjunts, amb 4 conjunts i 3 vies: el bloc de l'adreça `0b00000100` es mapa al conjunt 1 i pot ocupar qualsevol de les seves 3 vies.

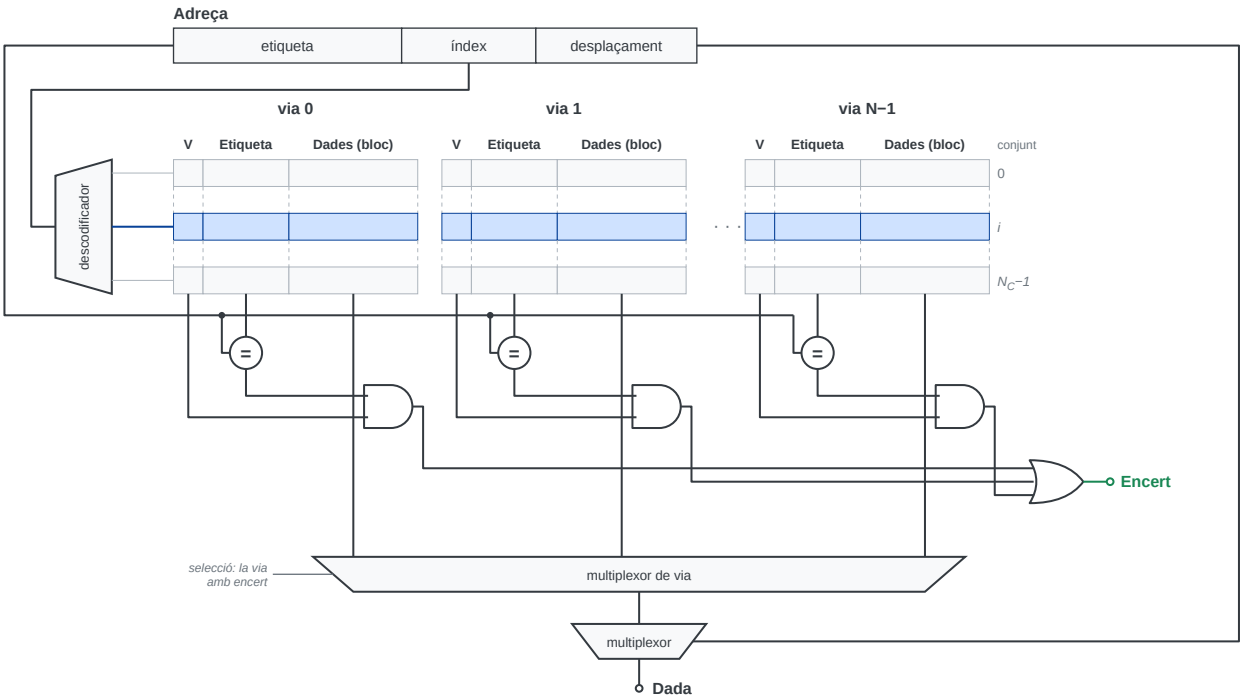


Figura 7.10: Diagrama de blocs d'una lectura en una memòria cau associativa per conjunts de N vies.

Com que no hi ha índex, l'adreça es descompon només en dos camps:

$$A = \underbrace{\cdots}_{\text{etiqueta}} \quad \underbrace{\cdots}_{\text{desplaçament } (b \text{ bits})}$$

L'etiqueta és el número de bloc sencer. El nombre de línies no cal que sigui potència de 2.

El diagrama de blocs de la Figura 7.11 mostra que ara tots els comparadors operen sobre totes les línies alhora: no hi ha descodificador. Una porta OR sobre totes les sortides dels comparadors genera el senyal d'encert, i un multiplexor N_L:1 selecciona la línia amb encert. El cost en maquinari creix **linealment** amb el nombre de línies N_L, cosa que fa que les memòries cau completament associatives pràctiques tinguin pocs centenars de línies com a màxim.

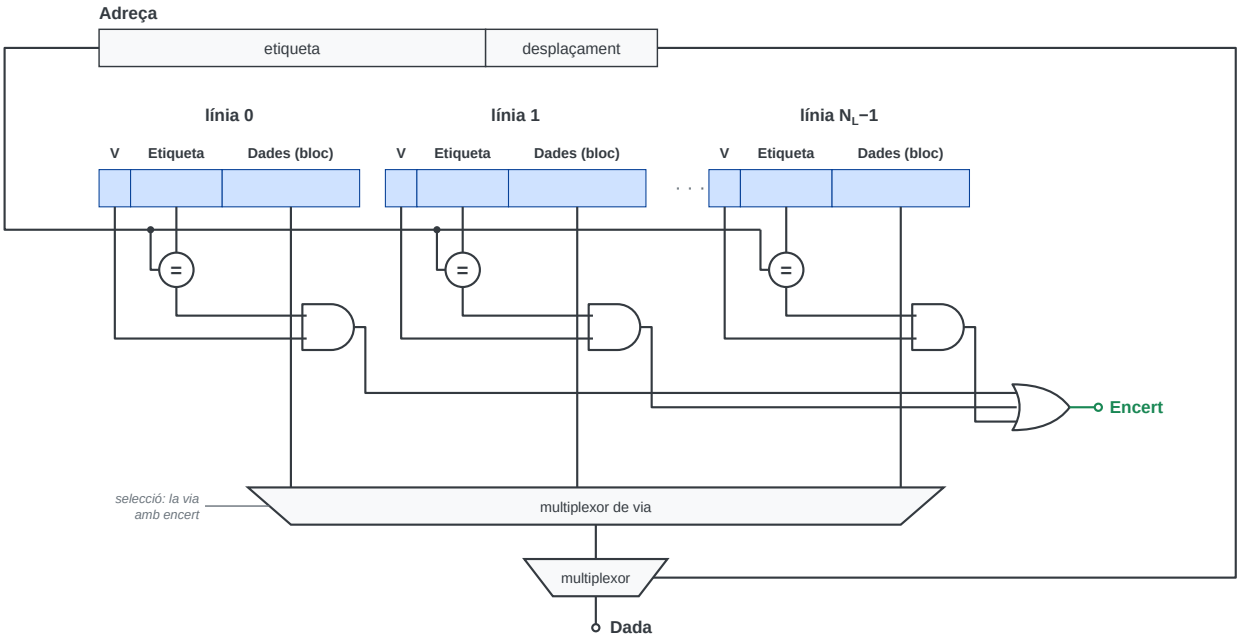


Figura 7.11: Diagrama de blocs d'una lectura en una memòria cau completament associativa: cal comparar l'etiqueta amb totes les línies en paral · lel.

La memòria cau completament associativa és el cas límit $N_C = 1$: un sol conjunt que conté totes les línies. Des d'aquest punt de vista, les tres polítiques d'emplaçament no són categories separades sinó els extrems i el punt mitjà d'un mateix espectre, parametritzat pel nombre de conjunts N_C i el grau d'associativitat N:

Taula 7.2: Relació entre les tres polítiques d'emplaçament

Política	Conjunts N_C	Vies N
Correspondència directa	N_L	1
Associativa per conjunts	N_L/N	N
Completament associativa	1	N_L

Essencial 7.1: Descomposició dels bits d'una adreça

Per a una memòria cau amb $N_C = 2^{n_c}$ conjunts i blocs de $B = 2^b$ bytes, els bits d'una adreça de w bits es distribueixen sempre de la mateixa manera, de més a menys pes (d'esquerra a dreta):

$$A = \underbrace{\overbrace{\cdots}^{w-n_c-b}\ \overbrace{\cdots}^{n_c}\ \overbrace{\cdots}^{b}}_{w}$$

$$\underbrace{\qquad}_{\text{etiqueta}}\ \underbrace{\quad}_{\text{índex}}\ \underbrace{\quad}_{\text{desplaçament}}$$

Els casos extrems encaixen naturalment: si $N_C = N_L$ (correspondència directa), $n_c = n_l$ i l'etiqueta és mínima; si $N_C = 1$ (completament associativa), $n_c = 0$ i no hi ha bits d'índex.

7.4 Polítiques de reemplaçament

Quan l'MC ha d'emmagatzemar un nou bloc i totes les vies del conjunt corresponent estan ocupades, cal alliberar-ne una. L'algorisme que decideix quina via s'elimina s'anomena **política de reemplaçament**. En una memòria cau de correspondència directa no hi ha elecció possible —cada bloc té una única línia assignada—, de manera que la política de reemplaçament només és rellevant per a les memòries cau associatives.

L'objectiu de qualsevol política de reemplaçament és minimitzar la taxa de fallades, és a dir, expulsar el bloc que tingui menys probabilitat de ser accedit en el futur. Com que el futur no és conegut, els algorismes reals intenten predir-lo a partir del comportament passat. Hi ha quatre algorismes principals:

Least recently used (LRU): Reemplaça el bloc que fa més temps que no s'ha accedit. Es basa en la localitat temporal: si un bloc no s'ha usat recentment, és poc probable que es necessiti aviat. Entre els algorismes implementables, és dels que donen la taxa de fallades més baixa en la majoria de casos pràctics, i és el que s'usarà en els problemes d'aquest curs.

La implementació d'LRU és senzilla per a 2 vies (un sol bit per conjunt n'és suficient), però es complica per a 4 o més vies. En la pràctica, per a 4 vies s'usa un **pseudo-LRU**: les vies s'agrupen en dues parelles, i tres bits per conjunt aproximen l'ordre d'ús (un bit indica quina parella s'ha usat menys recentment, i un bit per parella, quina via dins de cadascuna).

Aleatori (***Random***): Selecciona la via víctima a l'atzar, normalment mitjançant un mecanisme de maquinari senzill. No és tan eficient com LRU, però la seva implementació és molt simple i la seva taxa de fallades s'aproxima a la d'LRU quan la memòria cau és gran. És una alternativa atractiva en aquests casos.

First in, first out (FIFO): Reemplaça el bloc que va entrar al conjunt fa més temps. No s'ha de confondre amb LRU: FIFO considera quan va entrar el bloc, independentment de si s'ha accedit des d'aleshores. És menys eficient que LRU perquè un bloc al qual s'ha accedit recentment pot ser expulsat si va entrar fa molt de temps.

Òptim (*optimal*, OPT): Reemplaça el bloc que no s'usarà fins més tard. S'ha demostrat teòricament que OPT obté la taxa de fallades mínima possible. Malauradament, no és implementable en maquinari real perquè requereix conèixer els accessos futurs. La seva utilitat pràctica és establir una **cota inferior** de la taxa de fallades: en entorns de simulació, permet avaluar quant de marge de millora queda per a nous algorismes. Val la pena observar que LRU és, de fet, un intent intuïtiu d'aproximar OPT: «el bloc que fa més temps que no s'usa» és una bona predicció del bloc que s'usarà més tard.

Exemple 7.5

Pregunta: Suposem adreces de 8 bits, blocs de $B = 4$ bytes ($b = 2$) i una memòria cau associativa per conjunts, amb 4 conjunts ($n_c = 2$) de 2 vies, inicialment buida. Donada la seqüència d'accessos de lectura a les adreces 4, 38, 6, 52 (totes al conjunt 1), determina l'estat de la memòria cau i indica el tipus de cada accés (encert o fallada) aplicant LRU.
Solució:

- `@4` → bloc 1, conjunt 1 → **Fallada** (via 0 lliure → s'ocupa).
- `@38` → bloc 9, conjunt 1 → **Fallada** (via 1 lliure → s'ocupa; ara LRU = via 0, la que fa més temps que no s'hi accedeix).
- `@6` → bloc 1, conjunt 1 → **Encert** (via 0). LRU passa a ser la via 1.
- `@52` → bloc 13, conjunt 1 → **Fallada** (cap via lliure; reemplaça LRU = via 1).

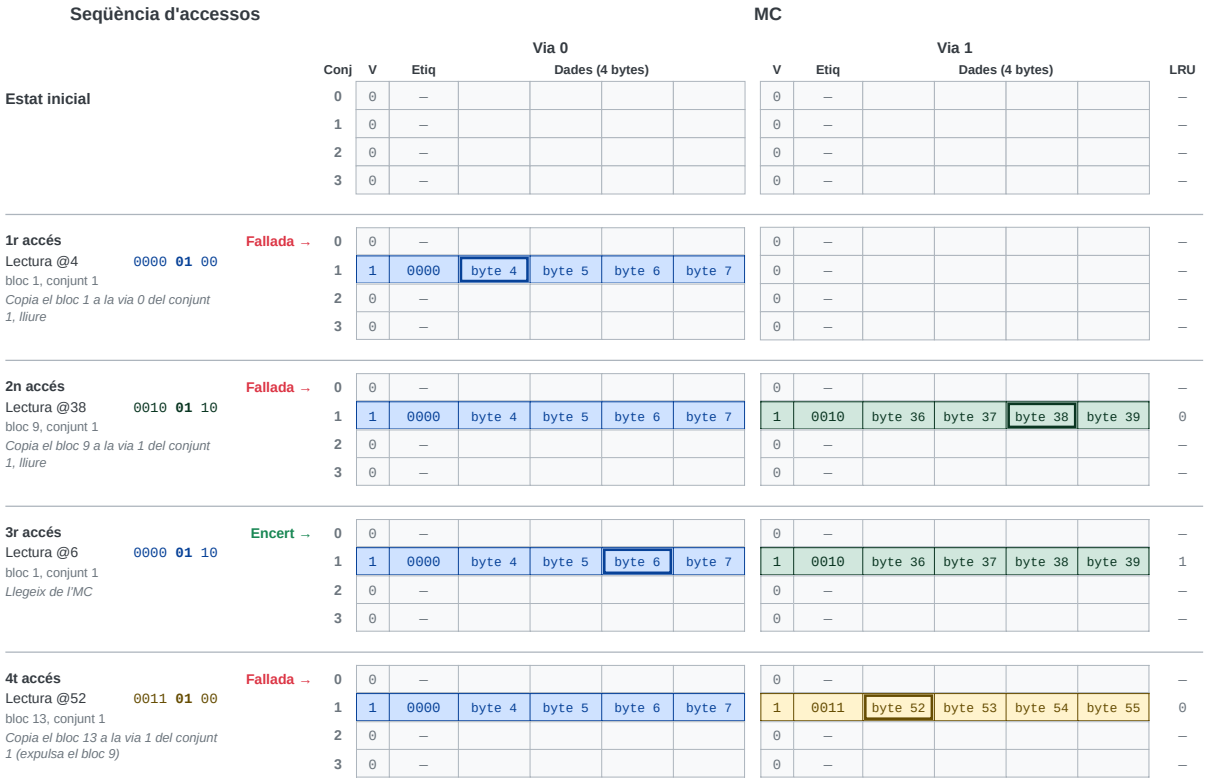


Figura 7.12: Evolució de la memòria cau associativa per conjunts de 2 vies amb reemplaçament LRU per a la seqüència de lectures `@4`, `@38`, `@6`, `@52`. Tots els accessos van al conjunt 1; els conjunts 0, 2 i 3 resten buits. La columna LRU diu, per a cada conjunt ple, quina via es reemplaçaria.

7.5 Polítiques d'escriptura

Fins ara s'ha suposat que tots els accessos a memòria són lectures. Les escriptures introdueixen un problema nou: quan la CPU modifica una dada, la còpia a l'MC i la còpia a la memòria principal divergeixen. Cal decidir com i quan sincronitzar-les. Això és el que resol la **política d'escriptura**.

La política d'escriptura té dues dimensions independents: com es gestiona un **encert** d'escriptura, i

com es gestiona una **fallada** d'escriptura.

7.5.1 Gestió de les escriptures en cas d'encert

Quan una escriptura produeix un encert, la dada és present a l'MC i cal decidir si s'actualitza també la memòria principal:

Escriptura immediata (***write-through***): S'escriu simultàniament a l'MC i a la memòria principal. La memòria principal rep totes les escriptures i mai no queda cap bloc modificat només a l'MC. Quan calgui reemplaçar un bloc, es pot fer directament sense cap escriptura prèvia a la memòria principal, ja que aquesta ja té la versió actualitzada.

L'inconvenient és que cada escriptura accedeix a la memòria principal, que és lenta. Per mitigar-ho s'usa una **memòria intermèdia d'escriptura** (*write buffer*): les escriptures pendents es posen a la cua de la memòria intermèdia i es transfereixen a la memòria principal en segon pla mentre la CPU continua executant. Si la memòria intermèdia s'omple (perquè el ritme d'escriptures supera la velocitat de buidatge), la CPU s'ha d'aturar fins que hi hagi espai.

Escriptura retardada (***write-back*** o ***copy-back***): S'escriu únicament a l'MC. La memòria principal queda temporalment desactualitzada. Per saber si un bloc ha estat modificat, cada línia incorpora un **bit de modificació** (*dirty bit*, D; Figura 7.13). $D = 1$ indica que el bloc ha estat modificat i la memòria principal no té la versió actualitzada. Quan calgui reemplaçar un bloc amb $D = 1$, caldrà copiar-lo prèviament a la memòria principal per no perdre les modificacions.

#	V	D	Etiqueta	Dades (16 bytes)
0	1	1	0x100100F	bytes 0x100100F0–0x100100FF
1	1	0	0x7FFFF01	bytes 0x7FFFF010–0x7FFFF01F
2	1	1	0x1001010	bytes 0x10010100–0x1001010F
3	0	0	–	

Figura 7.13: Estructura d'una línia de memòria cau amb escriptura retardada: el bit de modificació D indica si el bloc ha estat modificat respecte a la còpia de la memòria principal.

L'escriptura retardada té l'avantatge que múltiples escriptures al mateix bloc generen una sola escriptura a la memòria principal (quan el bloc s'expulsa), cosa que redueix el tràfic total. El desavantatge és que una fallada que requereix expulsar un bloc amb $D = 1$ allarga el temps de penalització, ja que cal copiar primer el bloc antic a la memòria principal abans de copiar el nou bloc a l'MC.

7.5.2 Polítiques en cas de fallada

Quan una escriptura produeix una fallada (el bloc no és a l'MC), hi ha dues alternatives:

Escriptura amb assignació (***write allocate***): Es copia el bloc de la memòria principal a l'MC (igual que en una fallada de lectura) i després s'escriu la dada a l'MC. Els accessos posteriors al mateix bloc es resoldran des de l'MC.

Escriptura sense assignació (***no-write allocate***): S'escriu la dada directament a la memòria principal sense copiar el bloc a l'MC. La memòria cau no s'altera. És adequada quan no s'espera que s'hi torni a accedir aviat.

7.5.3 Combinacions habituals

Combinant una política d'encert amb una política de fallada s'obté la **política d'escriptura** completa. A la pràctica, no totes les combinacions són igual de comunes:

Taula 7.3: Combinacions habituals de política d'escriptura

	Amb assignació	Sense assignació
Escriptura immediata	✓	✓
Escriptura retardada	✓	—

L'escriptura retardada sense assignació és poc habitual: les escriptures a un bloc que no és a l'MC anirien totes a la memòria principal, una per una, i es perdria precisament el que aporta l'escriptura retardada, que és agrupar-les en una sola còpia del bloc. En aquest curs s'estudien les tres combinacions marcades:

1. **Escriptura immediata amb assignació**.
2. **Escriptura immediata sense assignació**.
3. **Escriptura retardada amb assignació**.

7.5.4 Exemples

Els exemples següents usen una memòria cau de correspondència directa molt senzilla per il · lustrar el comportament de cada política. El context és comú als tres:

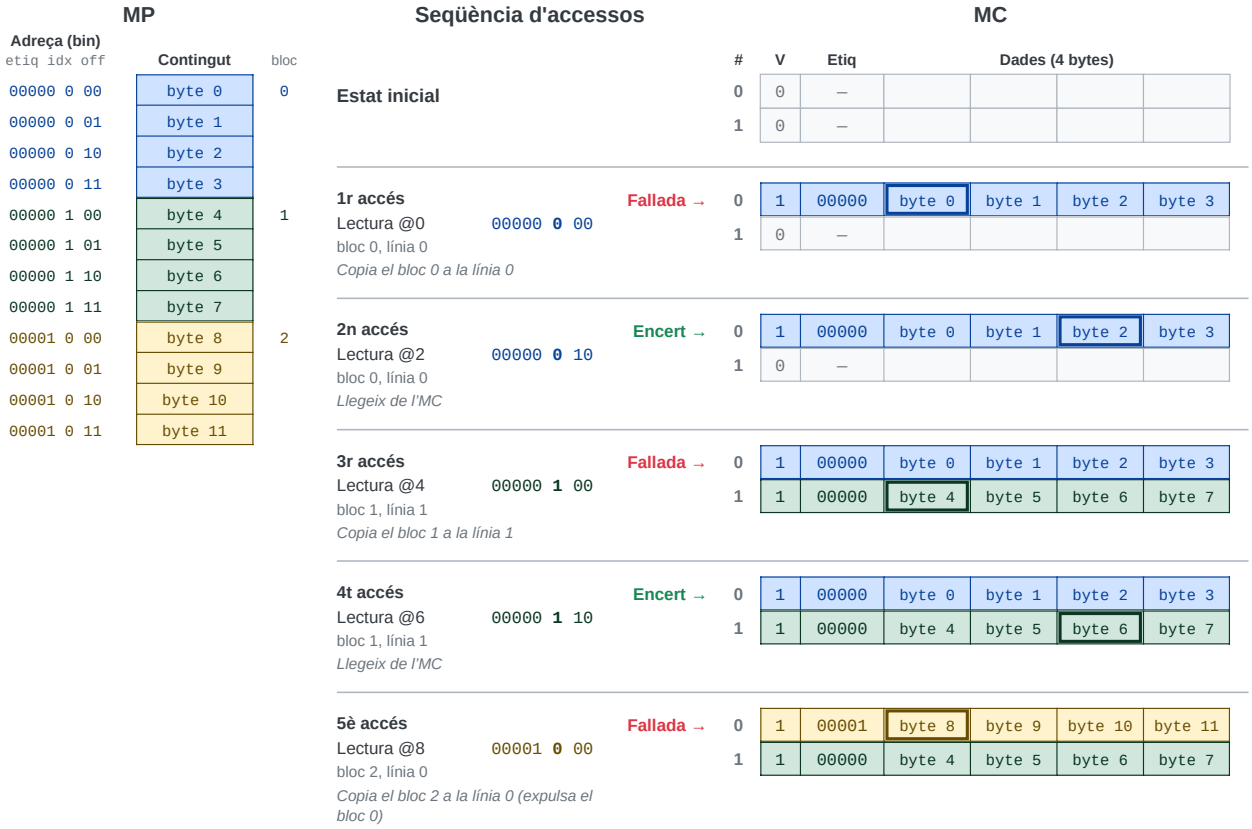
- Processador amb adreces de 8 bits (rang 0–255).
- Blocs de $B = 4$ bytes ($b = 2$): 2 bits de desplaçament, 6 bits de número de bloc.
- Memòria cau de 2 línies ($n_l = 1$): 1 bit d'índex, 5 bits d'etiqueta.

Es parteix sempre de l'estat de la memòria cau després d'executar la seqüència inicial de 5 lectures (Figura 7.14):

- `lb @0` (bloc 0, índex 0) → **Fallada**, copia el bloc 0 a la línia 0.
- `lb @2` (bloc 0, índex 0) → **Encert**.
- `lb @4` (bloc 1, índex 1) → **Fallada**, copia el bloc 1 a la línia 1.
- `lb @6` (bloc 1, índex 1) → **Encert**.
- `lb @8` (bloc 2, índex 0) → **Fallada**, reemplaça el bloc 0 a la línia 0.

Estat resultant: línia 0 conté el bloc 2 (`@8`–`@11`), línia 1 conté el bloc 1 (`@4`–`@7`).

> Exemple 7.6: Escriptura immediata amb assignació
>
> Partint de l'estat inicial, s'executen dues escriptures:
>
> - `sb @10` (bloc 2, índex 0) → **Encert**: s'escriu a l'MC i a la memòria principal simultàniament.
> - `sb @252` (bloc 63, índex 1) → **Fallada**: es copia primer el bloc 63 de la memòria principal a la línia 1 (reemplaçant el bloc 1), i després s'escriu a l'MC i a la memòria principal.



(a) Pas a pas: l'MP, cada lectura i l'estat de l'MC després de l'accés.

Accés	Bloc	Línia	Resultat	MC: bloc de cada línia 0	1
Estat inicial				—	—
Lectura @0	0	0	Fallada	**0**	—
Lectura @2	0	0	Encert	0	—
Lectura @4	1	1	Fallada	0	**1**
Lectura @6	1	1	Encert	0	1
Lectura @8	2	0	Fallada	**2**	1

(b) En forma de traça: una fila per lectura, amb el bloc que conté cada línia de l'MC després de l'accés; en color, el que acaba de canviar, i amb la vora gruixuda, la línia a què s'ha accedit.

Figura 7.14: Estat de la memòria cau al final de la seqüència inicial de 5 lectures, partint d'una MC buida, en les dues formes que fa servir aquest tema. La línia 0 conté el bloc 2 (`@8`–`@11`) i la línia 1 conté el bloc 1 (`@4`–`@7`).

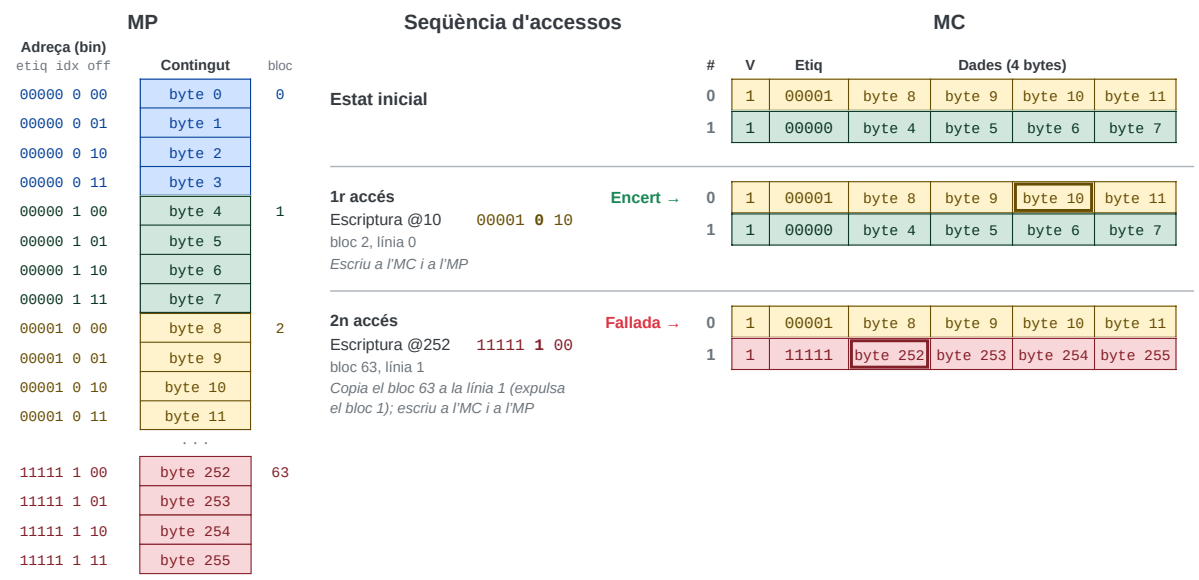


Figura 7.15: Escriptura immediata amb assignació: en cas d'encert s'escriu a l'MC i a l'MP simultàniament; en cas de fallada, es copia primer el bloc a l'MC i després s'escriu a l'MC i a l'MP.

Exemple 7.7: Escriptura immediata sense assignació

Partint del mateix estat inicial, les mateixes dues escriptures:

- `sb @10` (bloc 2, índex 0) → **Encert**: s'escriu a l'MC i a la memòria principal (igual que amb assignació).
- `sb @252` (bloc 63, índex 1) → **Fallada**: s'escriu **únicament a la memòria principal**. La memòria cau no s'altera.



Figura 7.16: Escriptura immediata sense assignació: en cas d'encert s'escriu a l'MC i a l'MP; en cas de fallada, s'escriu directament a l'MP sense modificar l'MC.

Exemple 7.8: Escriptura retardada amb assignació

En aquest cas la memòria cau incorpora el bit D. Partint del mateix estat inicial (amb $D = 0$ a totes les línies), s'executen una escriptura, una lectura i una altra escriptura:

- `sb @10` (bloc 2, índex 0) → **Encert**: s'escriu **únicament a l'MC** i es posa $D = 1$ a la línia 0.
- `lb @0` (bloc 0, índex 0) → **Fallada** amb $D = 1$ a la línia 0: cal primer copiar el bloc 2 (modificat) a la memòria principal, i després copiar el bloc 0 a la línia 0 ($D \leftarrow 0$).
- `sb @252` (bloc 63, índex 1) → **Fallada** amb $D = 0$ a la línia 1: es copia el bloc 63 a la línia 1 (reemplaça el bloc 1) i s'escriu a l'MC ($D \leftarrow 1$).

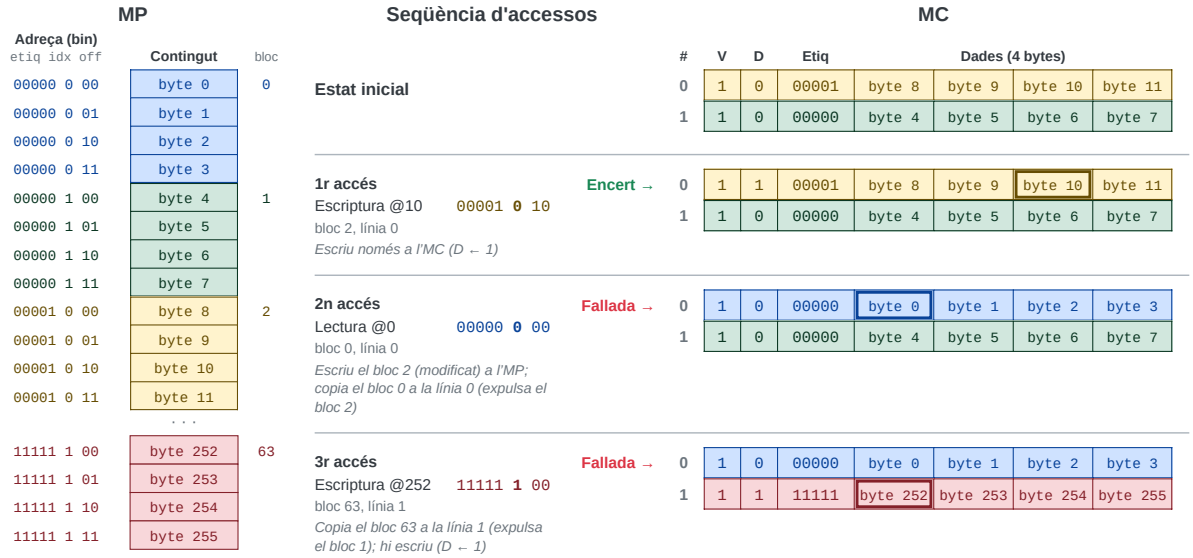


Figura 7.17: Escriptura retardada amb assignació: en cas d'encert s'escriu únicament a l'MC i es posa $D \leftarrow 1$; en cas de fallada, si el bloc expulsat té $D = 1$, cal copiar-lo primer a l'MP.

7.6 Polítiques — Resum

El disseny d'una memòria cau queda determinat per tres decisions independents: on es pot emmagatzemar cada bloc (política d'emplaçament, Secció 7.3), quin bloc s'expulsa quan no hi ha lloc (política de reemplaçament, Secció 7.4) i com se sincronitzen les escriptures amb la memòria principal (política d'escriptura, Secció 7.5). La Figura 7.18 recull les opcions estudiades per a cadascuna.

7.7 Rendiment

Les decisions de disseny estudiades fins ara —política d'emplaçament, d'escriptura i de reemplaçament— afecten el rendiment del sistema. Cal disposar d'eines per quantificar aquest impacte i comparar alternatives. En aquesta secció es defineixen les dues mètriques principals: el **temps d'accés mitjà** (TAM) a la memòria i el **temps d'execució** (t_{exe}).

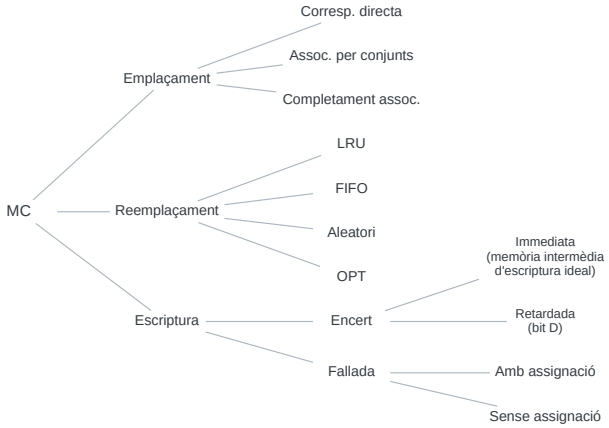


Figura 7.18: Resum de les polítiques de memòria cau.

7.7.1 Model de temps simplificat

El **temps d'accés** (t_{acc}) és el temps que transcorre entre la sol · licitud d'accés de la CPU i el moment en què la dada és efectivament disponible, de manera que la CPU pot prosseguir l'execució.

En cas d'**encert**, el temps d'accés és $t_{acc} = t_h$. Aquest temps inclou la comprovació de l'etiqueta i la transferència de la dada entre l'MC i la CPU, i és un paràmetre constructiu fix de la memòria cau.

En cas de **fallada**, cal afegir un temps de penalització t_p:

$$t_{acc} = t_h + t_p$$

El temps de penalització depèn de la política d'escriptura. Per modelar-lo de forma unívoca, es defineix t_{bloc} com el temps necessari per transferir un bloc sencer entre l'MC i la memòria principal. Amb aquest paràmetre, el temps de penalització per a cada política és el següent:

1. Escriptura immediata amb assignació

Totes les fallades (de lectura o d'escriptura) tenen la mateixa penalització: cal copiar el bloc de la memòria principal a l'MC (t_{bloc}) i transferir la dada a/de la CPU (t_h):

$$t_p = t_{bloc} + t_h$$

2. Escriptura immediata sense assignació

- **Fallada de lectura**: igual que en el cas anterior:

$$t_p = t_{bloc} + t_h$$

- **Fallada d'escriptura**: s'escriu la dada a la memòria intermèdia d'escriptura durant la comprovació de l'etiqueta, sense cap temps extra:

$$t_p = 0$$

3. Escriptura retardada amb assignació

La penalització depèn de si el bloc que cal reemplaçar ha estat modificat:

- **Bloc no modificat** ($D = 0$): igual que en l'escriptura immediata:

$$t_p = t_{bloc} + t_h$$

- **Bloc modificat** ($D = 1$): cal primer copiar el bloc antic a la memòria principal (t_{bloc}), després copiar el nou bloc de la memòria principal a l'MC (t_{bloc}), i finalment transferir la dada a/de la CPU (t_h):

$$t_p = 2 \cdot t_{bloc} + t_h$$

Essencial 7.2: Resum del model de temps

	Immediata amb assignació	Immediata sense assignació	Retardada amb assignació
Fallada de lectura	$t_{bloc} + t_h$	$t_{bloc} + t_h$	$t_{bloc} + t_h$ (si $D = 0$) $2t_{bloc} + t_h$ (si $D = 1$)
Fallada d'escriptura	$t_{bloc} + t_h$	0	$t_{bloc} + t_h$ (si $D = 0$) $2t_{bloc} + t_h$ (si $D = 1$)

7.7.2 Temps d'accés mitjà (TAM)

Com que el temps d'accés varia en cada referència (encert o fallada, lectura o escriptura, bloc modificat o no), s'usa el **temps d'accés mitjà** (TAM), també anomenat *average memory access time* (AMAT), que en dona la mitjana ponderada:

$$\text{TAM} = t_h + m \cdot t_p \tag{7.1}$$

on m és la taxa de fallades i t_p el temps de penalització mitjà.

7.7.2.1 Memòria cau unificada

Si hi ha una sola memòria cau per a instruccions i dades, l'expressió Equació 7.1 és directament aplicable. Quan la política d'escriptura implica penalitzacions diferents per a fallades de lectura i d'escriptura, cal ponderar:

- **Escriptura immediata sense assignació**: si p_e és la fracció de fallades que corresponen a escriptures:

$$t_p = (1 - p_e) \cdot (t_{bloc} + t_h) + p_e \cdot 0 = (1 - p_e)(t_{bloc} + t_h)$$

- **Escriptura retardada amb assignació**: si p_m és la fracció de fallades que reemplacen un bloc modificat ($D = 1$):

$$t_p = (1 - p_m)(t_{bloc} + t_h) + p_m(2 \cdot t_{bloc} + t_h) = (1 + p_m) \cdot t_{bloc} + t_h$$

Essencial 7.3: Fórmula del TAM per a escriptura retardada

La fórmula correcta del temps de penalització mitjà per a l'escriptura retardada amb assignació és:

$$t_p = (1 + p_m) \cdot t_{bloc} + t_h$$

Cal notar que t_h **no** es multiplica per $(1+p_m)$: el terme t_h apareix una sola vegada independentment de si el bloc és modificat o no.

7.7.2.2 TAM amb memòries cau separades d'instruccions i dades

En moltes arquitectures, la CPU disposa de dues memòries cau independents: una memòria cau d'instruccions (MC_i, de sols lectura) i una memòria cau de dades (MC_d, usada per les instruccions de lectura i escriptura: `lw`, `sw`, `lb`, `sb`…). Siguin:

- n_{ins}: nombre d'accessos a MC_i (igual al nombre d'instruccions executades).
- n_{dat}: nombre d'accessos a MC_d (igual al nombre d'instruccions de lectura i escriptura).
- $n_{refs} = n_{ins} + n_{dat}$: nombre total de referències a memòria.
- m_i, m_d: taxes de fallades de MC_i i MC_d.
- t_{pi}, t_{pd}: temps de penalització de MC_i i MC_d.

El TAM pondera les penalitzacions de cada memòria cau per la seva fracció d'accessos:

$$\text{TAM} = t_h + \frac{n_{ins}}{n_{refs}} \cdot m_i \cdot t_{pi} + \frac{n_{dat}}{n_{refs}} \cdot m_d \cdot t_{pd} \tag{7.2}$$

És habitual expressar-ho en funció del nombre de **referències a memòria per instrucció** $n_r = n_{refs}/n_{ins}$, que és sempre ≥ 1 (tota instrucció requereix almenys un accés per a la captura):

$$\text{TAM} = t_h + \frac{1}{n_r} \cdot m_i \cdot t_{pi} + \left(1 - \frac{1}{n_r}\right) \cdot m_d \cdot t_{pd} \tag{7.3}$$

7.7.3 Impacte de les fallades en el temps d'execució

El TAM mesura l'eficiència de la memòria cau de forma aïllada, però l'objectiu final és el **temps d'execució** del sistema. Les fallades de memòria cau provoquen cicles d'espera que s'afegeixen als cicles ideals:

$$\text{CPI} = \text{CPI}_{ideal} + \text{CPI}_{penal}$$

El terme CPI_{ideal} inclou tots els cicles d'execució excepte les esperes per fallades de memòria cau. El terme CPI_{penal} inclou exclusivament els cicles d'espera causats per les fallades:

$$\text{CPI}_{penal} = \frac{t_p \cdot n_{fallades}}{n_{ins}} = t_p \cdot m \cdot \frac{n_{refs}}{n_{ins}} = t_p \cdot m \cdot n_r$$

on t_p i m s'expressen en cicles i en fracció respectivament. Partint de Equació 6.3, el temps d'execució queda:

$$t_{exe} = (\text{CPI}_{ideal} + t_p \cdot m \cdot n_r) \cdot n_{ins} \cdot t_{clock} \tag{7.4}$$

on t_{clock} és el període de rellotge (Equació 6.3). La Figura 7.19 il · lustra com les fallades de memòria cau s'insereixen com a temps extra en el flux d'execució de les instruccions.

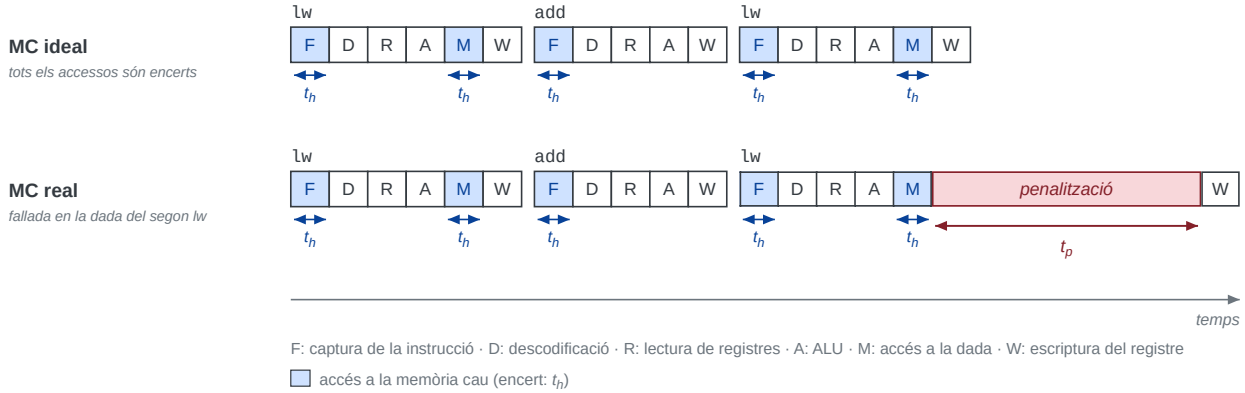


Figura 7.19: Impacte d'una fallada de memòria cau en el temps d'execució: els cicles de penalització s'afegeixen als cicles ideals de la instrucció.

De l'expressió Equació 7.4 s'extreuen dues conclusions pràctiques:

- Per reduir CPI_{penal} cal reduir m (la taxa de fallades) o t_p (la penalització). L'associativitat redueix m; les memòries cau multinivell redueixen t_p (vegeu la Secció 7.9).
- El terme CPI_{ideal} depèn de la microarquitectura del processador, mentre que CPI_{penal} depèn del disseny de la memòria cau i del programa. Millorar un sense millorar l'altre té un impacte limitat, d'acord amb la llei d'Amdahl (Secció 6.2).

7.8 Tipologia de les fallades

Les fallades de memòria cau es classifiquen segons la seva causa en tres categories, conegudes com les **tres C** (de l'anglès *compulsory*, *conflict* i *capacity*). Classificar-les permet raonar sobre per què es produeixen i sobre quines mesures de disseny poden reduir cadascuna.

Fallades d'arrencada en fred (***cold-start***), també anomenades **obligatòries** (***compulsory***) o **de primera referència** (***first-reference***): es produeixen el primer cop que s'accedeix a un bloc. Cap memòria cau no pot tenir un bloc al qual encara no s'ha accedit mai, de manera que aquestes fallades **no depenen** de la política d'emplaçament, del grau d'associativitat ni de la capacitat de la memòria cau: n'hi ha tantes com blocs diferents als quals accedeix el programa. Dels paràmetres de la memòria cau, només les redueixen els blocs més grans, perquè cada fallada porta més dades veïnes a la memòria cau (localitat espacial).

Fallades de conflicte (o **de col · lisió**, ***conflict***): es produeixen quan s'accedeix alternament a dos o més blocs que es mapen al mateix conjunt i s'expulsen mútuament, encara que hi hagi línies lliures en altres conjunts. Només n'hi ha en les memòries cau amb indexació (correspondència directa i associativa per conjunts): **una fallada de conflicte seria un encert en una memòria cau completament associativa de la mateixa capacitat**. Augmentar el grau d'associativitat les redueix.

Fallades de capacitat (***capacity***): es produeixen perquè la memòria cau no és prou gran per contenir tots els blocs que el programa necessita alhora, el ***working set***: un bloc expulsat per manca d'espai torna a fallar quan es torna a necessitar. **També fallaria en una memòria cau completament**

associativa de la mateixa capacitat, i per això no s'elimina augmentant l'associativitat, sinó només augmentant la capacitat.

Els dos criteris en negreta donen un mètode per classificar qualsevol fallada que no sigui d'arrencada en fred: es repeteix la mateixa seqüència d'accessos en una memòria cau completament associativa de la mateixa capacitat, amb reemplaçament LRU, i si l'accés també hi falla, la fallada és de capacitat; si hi encerta, és de conflicte.

La Figura 7.20 il · lustra la relació entre els tres tipus, el grau d'associativitat i la mida de la memòria cau per a un programa donat. Les fallades d'arrencada en fred no varien, perquè només depenen dels blocs als quals accedeix el programa. Per a una mida donada (la línia discontínua), les de conflicte disminueixen a mesura que augmenta l'associativitat, i en una memòria cau completament associativa no n'hi ha. Les de capacitat, en canvi, només disminueixen si augmenta la mida de la memòria cau, sigui quin sigui el grau d'associativitat.

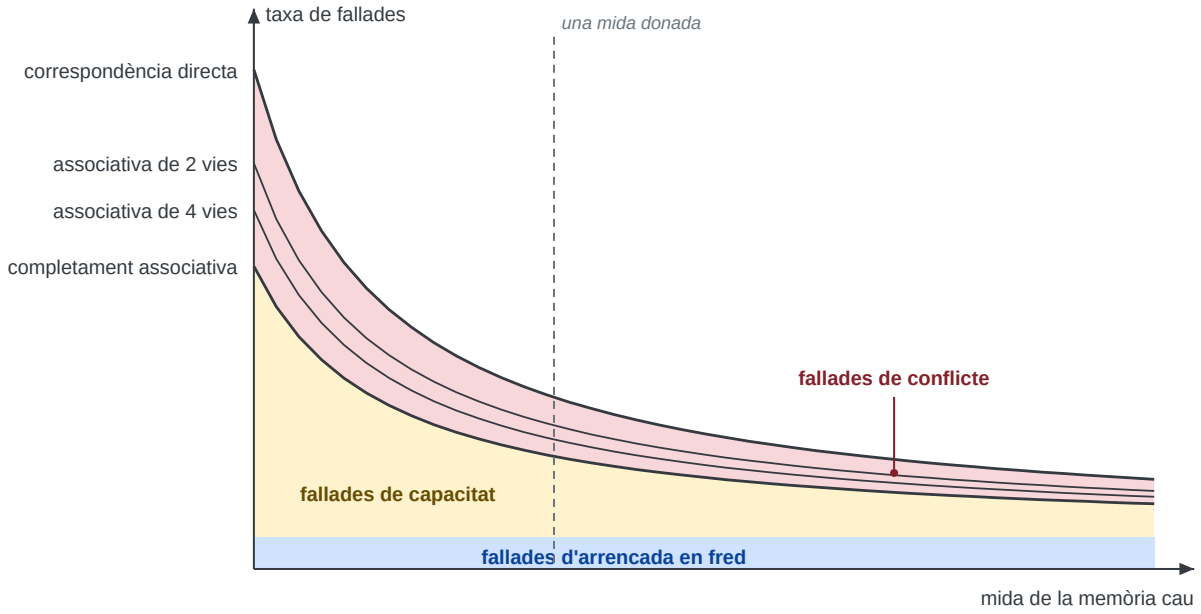


Figura 7.20: Taxa de fallades segons la mida de la memòria cau i el grau d'associativitat, per a un programa donat (gràfica qualitativa).

Essencial 7.4: Les tres C

Taula 7.4: Els tres tipus de fallades, la causa i què les redueix.

Tipus	Causa	Què les redueix
Arrencada en fred (obligatòries)	Primer accés a un bloc	Blocs més grans; no depenen de la capacitat ni de l'associativitat
Conflicte (col · lisió)	Blocs que competeixen pel mateix conjunt	Més associativitat
Capacitat	*Working set* més gran que la memòria cau	Més capacitat

Una fallada que no és d'arrencada en fred és **de conflicte** si encertaria en una memòria cau **completament associativa** de la mateixa capacitat, i **de capacitat** si també hi fallaria.

Aprofundiment 7.2: Fallades de coherència: la quarta C

En els processadors multinucli, cada nucli té la seva memòria cau privada (Secció 7.9.4), i n'hi ha un quart tipus. La classificació passa a ser de les **quatre C**.
Fallades de coherència (o **d'invalidació**, *coherence*): quan un nucli modifica una dada, les còpies d'aquell bloc a les memòries cau privades dels altres nuclis s'invaliden, perquè tots vegin el mateix valor. El pròxim accés d'un altre nucli a aquell bloc falla, i n'ha de tornar a portar la còpia actualitzada. Aquestes fallades no depenen de la capacitat ni de l'associativitat de cap memòria cau, sinó de les dades que comparteixen els nuclis, i de la mida del bloc: dos nuclis que escriuen variables diferents d'un mateix bloc també s'invaliden mútuament.

7.8.1 Exemple: fallades de conflicte i d'arrencada en fred

Considerem la funció C del Codi 7.1, executada en un processador RV32I amb blocs de 8 bytes (2 paraules per bloc, $b = 3$):

Codi 7.1: Producte escalar de dos vectors de 8 enters.

C

```c
int f(int A[8], int B[8]) {
    int i, s = 0;
    for (i = 0; i < 8; i++)
        s = s + A[i] * B[i];
    return s;
}
```

Suposem una memòria cau **de dades** de correspondència directa amb 4 línies ($n_l = 2$), inicialment buida. Els vectors `A` i `B` es troben en posicions contigües de memòria: `A` comença al bloc 0 i `B` al bloc 4.

Com que cada bloc conté 2 enters (8 bytes / 4 bytes per `int`), els elements `A[i]` i `A[i+1]` van al mateix bloc, i el mateix passa amb `B`. La taula següent mostra el mapatge complet:

`i`	Bloc d'`A[i]`	Bloc de `B[i]`	Índex (línia)
0, 1	0	4	0
2, 3	1	5	1
4, 5	2	6	2
6, 7	3	7	3

Per a cada parella d'iteracions, els blocs d'`A[i]` i `B[i]` difereixen exactament en $N_L = 4$: sumar N_L no altera els $n_l = 2$ bits de menys pes del número de bloc (l'índex), de manera que tots dos blocs es mapen a la **mateixa línia** però amb **etiquetes diferents**. El resultat és que `A[i]` i `B[i]` s'expulsen mútuament a cada accés.

La traça dels primers quatre valors d'`i` mostra els dos tipus de fallada:

- $i = 0$: `lw A[0]` (bloc 0, línia 0) → **fallada obligatòria**; carrega el bloc 0. `lw B[0]` (bloc 4, línia 0) → **fallada obligatòria**; reemplaça el bloc 0.
- $i = 1$: `lw A[1]` (bloc 0, línia 0) → **fallada de conflicte**; el bloc 0 havia estat carregat a $i = 0$ però expulsat per `B[0]`. `lw B[1]` (bloc 4, línia 0) → **fallada de conflicte**.
- $i = 2$: `lw A[2]` (bloc 1, línia 1) → **fallada obligatòria**; primer accés al bloc 1. `lw B[2]` (bloc 5, línia 1) → **fallada obligatòria**.
- $i = 3$: `lw A[3]` (bloc 1, línia 1) → **fallada de conflicte**. `lw B[3]` (bloc 5, línia 1) → **fallada de conflicte**.

El patró es repeteix per a $i \in \{4, 5\}$ i $i \in \{6, 7\}$. En total: **8 fallades obligatòries** (iteracions parelles, primer accés al bloc) i **8 fallades de conflicte** (iteracions senars, reaccés a un bloc expulsat). **Els 16 accessos fallen** (Figura 7.21).

				MC: bloc de cada línia			
Accés	Bloc	Línia	Resultat	0	1	2	3
i = 0							
Lectura A[0]	0	0	Fallada obligatòria	**0**	—	—	—
Lectura B[0]	4	0	Fallada obligatòria	**4**	—	—	—
i = 1							
Lectura A[1]	0	0	Fallada de conflicte	**0**	—	—	—
Lectura B[1]	4	0	Fallada de conflicte	**4**	—	—	—
i = 2							
Lectura A[2]	1	1	Fallada obligatòria	4	**1**	—	—
Lectura B[2]	5	1	Fallada obligatòria	4	**5**	—	—
i = 3							
Lectura A[3]	1	1	Fallada de conflicte	4	**1**	—	—
Lectura B[3]	5	1	Fallada de conflicte	4	**5**	—	—
i = 4							
Lectura A[4]	2	2	Fallada obligatòria	4	5	**2**	—
Lectura B[4]	6	2	Fallada obligatòria	4	5	**6**	—
i = 5							
Lectura A[5]	2	2	Fallada de conflicte	4	5	**2**	—
Lectura B[5]	6	2	Fallada de conflicte	4	5	**6**	—
i = 6							
Lectura A[6]	3	3	Fallada obligatòria	4	5	6	**3**
Lectura B[6]	7	3	Fallada obligatòria	4	5	6	**7**
i = 7							
Lectura A[7]	3	3	Fallada de conflicte	4	5	6	**3**
Lectura B[7]	7	3	Fallada de conflicte	4	5	6	**7**

Figura 7.21: Fallades de conflicte en el recorregut paral · lel de dos vectors amb una memòria cau de correspondència directa de 4 línies, en forma de traça: una fila per accés, amb el bloc que conté cada línia després de l'accés. `A[i]` i `B[i]` van sempre a la mateixa línia: a les iteracions amb i parell, les dues lectures són fallades obligatòries; amb i senar, de conflicte. Fallen els 16 accessos.

Una memòria cau associativa per conjunts de 2 vies i la mateixa capacitat (2 conjunts de 2 vies) eliminaria completament les fallades de conflicte d'aquest exemple: els blocs d'`A` i `B` podrien coexistir a les dues vies del mateix conjunt.

7.8.2 Exemple: fallades de capacitat

Considerem ara la funció C del Codi 7.2, executada amb la mateixa configuració però amb una memòria cau **de dades completament associativa** de 4 línies (eliminant així qualsevol conflicte):

Codi 7.2: Dos recorreguts d'un vector de 16 enters.

C

```c
int g(int V[16]) {
    int i, s = 0;
    for (i = 0; i < 16; i++) s = s + V[i];
    for (i = 0; i < 16; i++) s = s + V[i] * 3;
    return s;
}
```

El vector `V` ocupa 8 blocs de 2 paraules cadascun. La memòria cau, amb 4 línies, només pot contenir la meitat del vector simultàniament (Figura 7.22).

- **Primer bucle**: els blocs de `V[0]`–`V[7]` entren a la memòria cau amb fallades obligatòries. Quan s'accedeix a `V[8]`–`V[15]`, la memòria cau és plena i els blocs de la primera meitat s'expulsen (LRU).
- **Segon bucle**: quan es torna a accedir a `V[0]`–`V[7]`, els blocs ja no hi són. Fallen de nou: són **fallades de capacitat**, i en tornar-los a portar s'expulsen els de `V[8]`–`V[15]`, que també tornaran a fallar: el segon bucle té 8 fallades de capacitat, tantes com fallades obligatòries el primer bucle.

Augmentar l'associativitat no resoldria res: les fallades es produeixen perquè el *working set* (8 blocs) supera la capacitat de la memòria cau (4 línies), no per cap conflicte d'índex. Sense canviar el programa, l'única solució és augmentar la capacitat de la memòria cau fins a 64 bytes o més (per exemple, 8 línies de 8 bytes).

7.9 Memòries cau multinivell

A la Secció 7.7.3 s'ha vist que el terme dominant del temps d'execució pot ser $\text{CPI}_{penal} = t_p \cdot m \cdot n_r$. L'associativitat redueix m, però no aborda el problema de t_p: el temps que tarda la memòria principal a respondre una fallada no ha disminuït al llarg dels anys al mateix ritme que el temps de processament d'una instrucció. En conseqüència, t_p ha esdevingut una fracció cada cop més dominant del temps d'execució.

La solució és aplicar el mateix principi que va donar lloc a la memòria cau: interposar un nivell addicional de memòria entre l'MC i la memòria principal.

7.9.1 Estructura i funcionament

En un sistema amb dos nivells de memòria cau, **L1** i **L2**, el flux d'un accés és el següent (Figura 7.23):

1. La CPU accedeix primer a **L1**. Si hi ha encert, l'accés es resol amb temps $t_{h_{L1}}$.
2. Si hi ha fallada a L1, s'accedeix a **L2**. Si hi ha encert a L2, la penalització és $t_{h_{L2}}$, molt menor que el temps d'accés a la memòria principal.
3. Només si el bloc falla a L2 cal accedir a la memòria principal, pagant la penalització completa t_p.

Accés	Bloc	Línia	Resultat	MC: bloc de cada línia 0	1	2	3
Primer bucle							
Lectura V[0]	0	0	Fallada obligatòria	**0**	—	—	—
Lectura V[1]	0	0	Encert	0	—	—	—
Lectura V[2]	1	1	Fallada obligatòria	0	**1**	—	—
Lectura V[3]	1	1	Encert	0	1	—	—
Lectura V[4]	2	2	Fallada obligatòria	0	1	**2**	—
Lectura V[5]	2	2	Encert	0	1	2	—
Lectura V[6]	3	3	Fallada obligatòria	0	1	2	**3**
Lectura V[7]	3	3	Encert	0	1	2	3
Lectura V[8]	4	0	Fallada obligatòria	**4**	1	2	3
Lectura V[9]	4	0	Encert	4	1	2	3
Lectura V[10]	5	1	Fallada obligatòria	4	**5**	2	3
Lectura V[11]	5	1	Encert	4	5	2	3
Lectura V[12]	6	2	Fallada obligatòria	4	5	**6**	3
Lectura V[13]	6	2	Encert	4	5	6	3
Lectura V[14]	7	3	Fallada obligatòria	4	5	6	**7**
Lectura V[15]	7	3	Encert	4	5	6	7
Segon bucle, de V[0] a V[7]							
Lectura V[0]	0	0	Fallada de capacitat	**0**	5	6	7
Lectura V[1]	0	0	Encert	0	5	6	7
Lectura V[2]	1	1	Fallada de capacitat	0	**1**	6	7
Lectura V[3]	1	1	Encert	0	1	6	7
Lectura V[4]	2	2	Fallada de capacitat	0	1	**2**	7
Lectura V[5]	2	2	Encert	0	1	2	7
Lectura V[6]	3	3	Fallada de capacitat	0	1	2	**3**
Lectura V[7]	3	3	Encert	0	1	2	3

Figura 7.22: Fallades de capacitat en una memòria cau completament associativa de 4 línies amb reemplaçament LRU, en forma de traça. Al primer bucle, cada bloc nou entra amb una fallada obligatòria i, a partir del cinquè, expulsa el més antic; al segon, els blocs de `V[0]` a `V[7]` ja no hi són i tornen a fallar, ara amb fallades de capacitat (la traça s'atura a `V[7]`: la segona meitat del segon bucle es comporta igual).

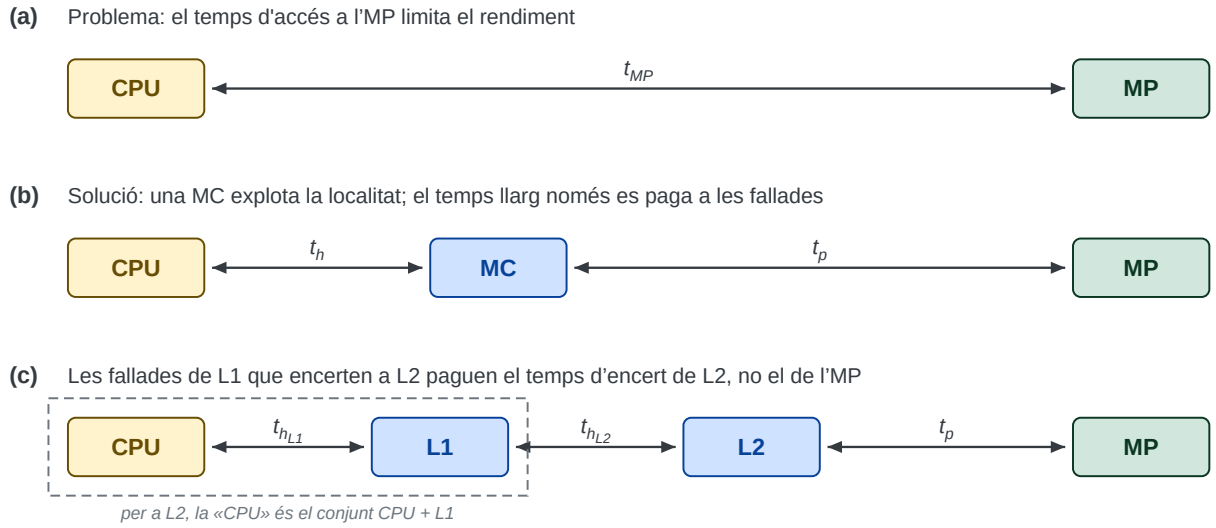


Figura 7.23: Les memòries cau multinivell redueixen la penalització de les fallades de L1: les que encerten a L2 paguen $t_{h_{L2}}$ en lloc de t_p. (a) Sense memòria cau, cada accés paga el temps de l'MP; (b) amb una MC, només les fallades; (c) amb L1 i L2, les fallades de L1 que també fallen a L2.

Les fallades de L1 que encerten a L2 paguen una penalització $t_{h_{L2}}$, molt menor que t_p. Només les fallades que arriben a la memòria principal paguen la penalització completa. El temps d'accés mitjà queda:

$$\text{TAM} = t_{h_{L1}} + m_{L1} \cdot (t_{h_{L2}} + m_{L2} \cdot t_p) \tag{7.5}$$

on m_{L1} és la taxa de fallades de L1 i m_{L2} és la taxa de fallades de L2 **entre les referències que han fallat a L1** (taxa de fallades local de L2). Aquí, $t_{h_{L2}}$ és tota la penalització d'una fallada de L1 que encerta a L2: inclou l'accés a L2 i la transferència del bloc a L1, sense cap t_{bloc} a part.

7.9.2 Disseny diferencial L1 i L2

Tenir dos nivells permet orientar el disseny de cadascun a un objectiu diferent:

L1 s'optimitza per minimitzar $t_{h_{L1}}$, ja que qualsevol cicle afegit aquí incrementa CPI_{ideal} i afecta totes les instruccions. Per aconseguir-ho:

- **Associativitat més baixa** que la de L2: menys comparadors, accés més ràpid.
- **Blocs petits**: menys dades a transferir i multiplexors més simples.
- **Capacitat moderada**: augmentar la capacitat incrementa el temps d'accés.

L2 s'optimitza per minimitzar m_{L2}, ja que una fallada a L2 accedeix a la memòria principal i paga la penalització completa. Com que $t_{h_{L2}}$ ja no afecta CPI_{ideal}, es pot admetre un $t_{h_{L2}}$ lleugerament més alt a canvi d'una taxa de fallades menor:

- **Associativitat més alta** que la de L1: redueix les fallades de conflicte.
- **Blocs grans**: exploten millor la localitat espacial.
- **Molta capacitat**: redueix les fallades de capacitat.

Essencial 7.5: Disseny diferencial L1 i L2

Taula 7.5: Objectius i paràmetres de disseny de les memòries cau L1 i L2.

	L1	L2
Objectiu	Minimitzar $t_{h_{L1}}$	Minimitzar m_{L2}
Associativitat	Més baixa	Més alta
Mida de bloc	Petita	Gran (als processadors actuals, sovint 64 B a tots els nivells)
Capacitat	Moderada (32–64 KiB)	Gran (256 KiB–diversos MiB)

7.9.3 Memòries cau separades d'instruccions i dades

Als processadors actuals, el nivell L1 es divideix habitualment en dues memòries cau independents:

- **L1i** (memòria cau d'instruccions): de sols lectura, s'usa durant la captura (*fetch*) de cada instrucció.
- **L1d** (memòria cau de dades): s'usa per a les instruccions de lectura i escriptura (`lw`, `sw`, `lb`, `sb`…).

Aquesta separació permet que ambdues memòries cau operin en paral · lel (la captura de la instrucció següent i l'accés a dades de la instrucció actual), cosa especialment rellevant en els processadors que comencen a buscar una instrucció abans d'acabar l'anterior (processadors segmentats, que no s'estudien a EC). A més, cadascuna es pot dissenyar de forma específica: L1i no necessita admetre escriptures i pot optimitzar-se per a accessos seqüencials.

A partir de L2, les memòries cau acostumen a ser **unificades** (instruccions i dades compartides).

7.9.4 Jerarquies actuals

L'augment del nombre de nuclis (*cores*) integrats en un mateix xip ha incrementat el tràfic sobre la memòria principal fins a saturar l'amplada de banda de la interconnexió. Per reduir aquest tràfic, les arquitectures actuals incorporen un tercer nivell **L3**, compartit entre tots els nuclis i integrat dins del mateix xip (Figura 7.24):

Habitualment, cada nucli disposa de L1i, L1d i L2 privades, orientades a minimitzar la latència per a aquell nucli. La L3 compartida actua com a darrer filtre abans de la memòria principal: redueix el nombre d'accessos a la DRAM i, per tant, el tràfic sobre el bus de memòria.

La incorporació de L3 respon a la mateixa lògica que va motivar L2: a mesura que la bretxa entre el processador i la memòria principal continua creixent, afegir un nivell intermedi addicional permet reduir la penalització efectiva i mantenir el CPI_{penal} sota control.

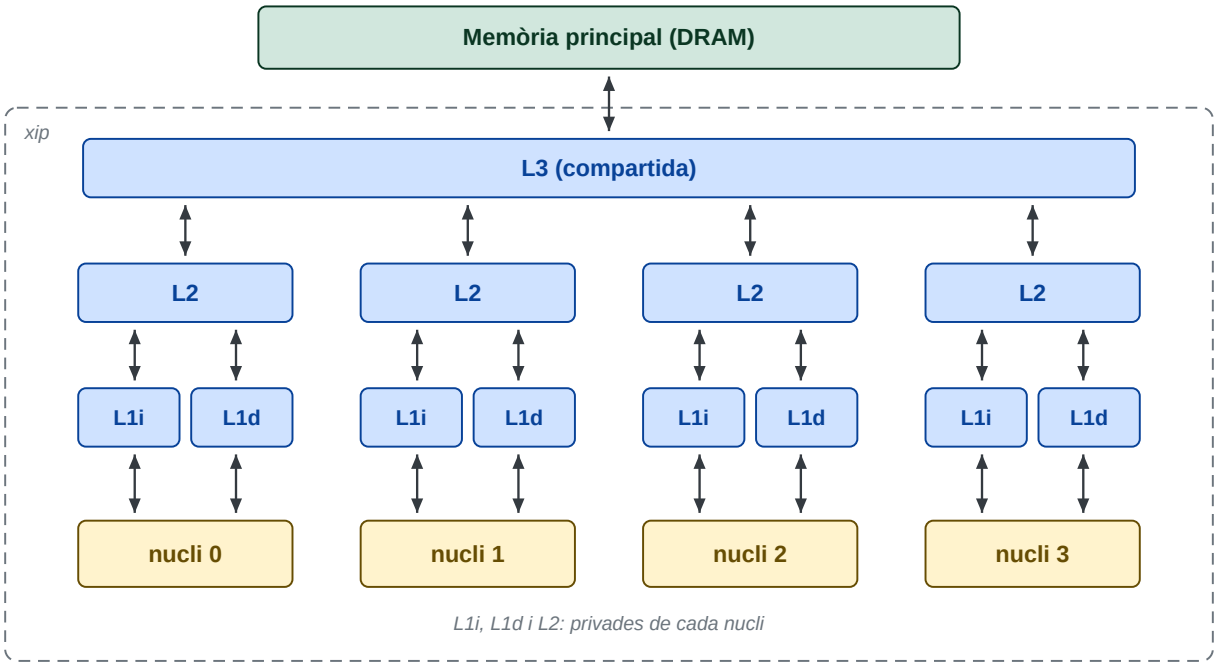


Figura 7.24: Jerarquia de memòries cau en un processador multinucli: L1 i L2 són privades de cada nucli, i L3 és compartida entre tots. L1i i L1d són les memòries cau de primer nivell d'instruccions i de dades, separades (Secció 7.9.3).

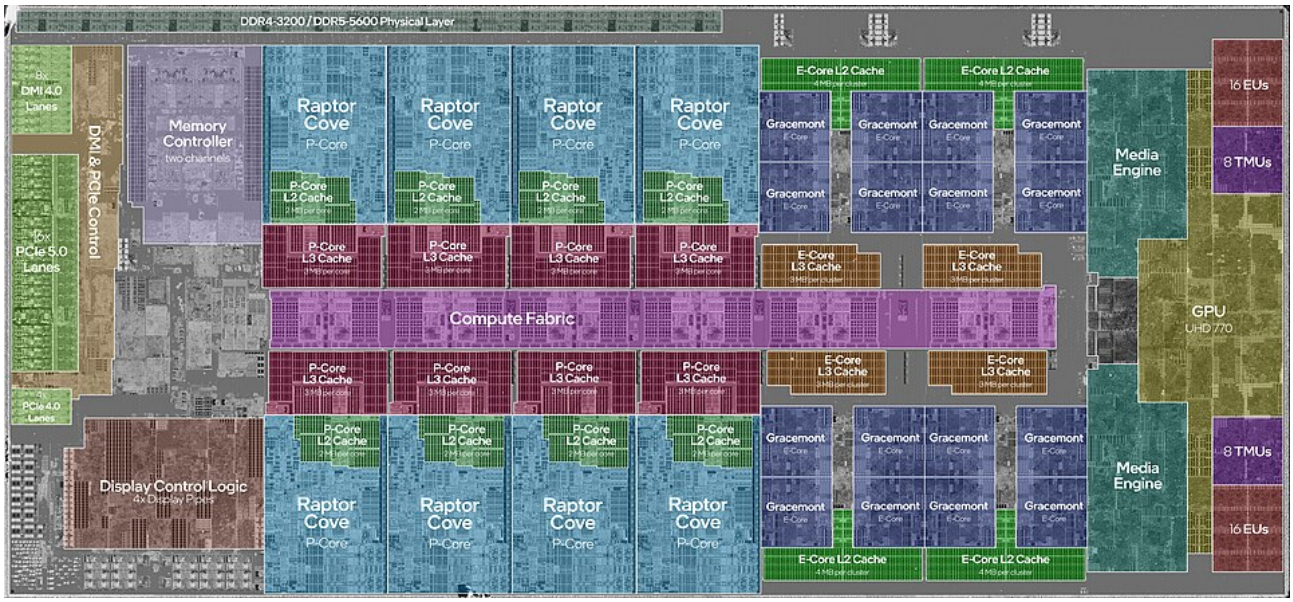


Figura 7.25: Fotografia del dau (bloc de silici, *die*) de l'Intel Core i9-13900K (*Raptor Lake*, 2022), amb els principals components etiquetats. S'hi distingeixen els nuclis de rendiment (*P-cores*) amb les seves memòries cau L2 privades, els nuclis d'eficiència (*E-cores*), agrupats en clústers de quatre que comparteixen una L2, i la memòria cau L3 compartida que ocupa una fracció significativa de la superfície del xip (Font: [14]).

Tema 8: Memòria virtual

8.1 Introducció

8.1.1 El problema: reubicació i mida limitada

Quan es compila i s'enllaça un programa, l'enllaçador assigna adreces de memòria a les instruccions i les dades. Però en un computador modern s'executen múltiples programes simultàniament, tots compartint la mateixa memòria física. Això planteja dos problemes fonamentals:

- **Reubicació.** Si dos programes han estat compilats per ocupar les mateixes adreces, no poden coexistir a la memòria sense que les seves dades i instruccions col · lideixin. Cal un mecanisme que permeti que cada programa cregui que ocupa les adreces que se li han assignat en compilar-lo, però que en realitat estigui ubicat en una zona diferent de la memòria física.
- **Mida limitada.** El conjunt d'instruccions i dades de tots els programes en execució pot superar fàcilment la capacitat de la memòria física. Sense cap mecanisme addicional, això impossibilitaria l'execució de certs programes o combinacions de programes.

La solució a tots dos problemes és la **memòria virtual**.

8.1.2 Espais d'adreçament lògic i físic

La memòria virtual es fonamenta en la distinció entre dos espais d'adreçament independents.

- L'**espai d'adreçament lògic** (o virtual) és el conjunt d'adreces que utilitza un programa: tant en temps de compilació com en temps d'execució, la CPU treballa exclusivament amb adreces lògiques. Cada procés té el seu propi espai d'adreçament lògic, independent del dels altres, amb una mida màxima determinada pel nombre de bits d'adreça del processador (per exemple, 2^{32} bytes = 4 GiB en un sistema de 32 bits; vegeu Essencial 2.5 per a la notació). Això proporciona a cada procés la il · lusió de disposar de tota la memòria per a ell sol.
- L'**espai d'adreçament físic** és el conjunt d'adreces de la memòria física real. La seva mida depèn de la quantitat de RAM instal · lada al sistema, i és totalment independent de l'espai d'adreçament lògic.
- La **unitat de gestió de memòria** (*memory management unit*, MMU) és el component de maquinari responsable de traduir, en temps d'execució, cada adreça lògica generada per la CPU en l'adreça física corresponent. Gràcies a aquesta traducció transparent, els programes no necessiten conèixer la seva ubicació real a la memòria física.

La Figura 8.1 il · lustra aquests tres elements amb dos processos.

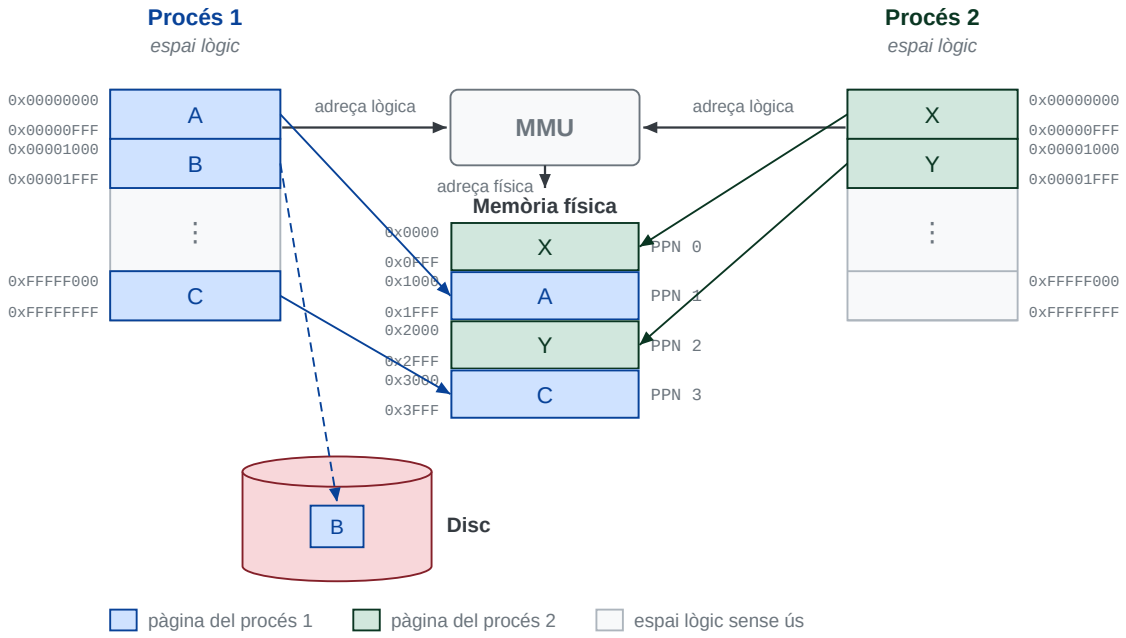


Figura 8.1: Cada procés disposa d'un espai d'adreçament lògic propi i independent, de `0x00000000` a `0xFFFFFFFF`, del qual només fa servir unes quantes pàgines (Secció 8.2.1), identificades amb lletres. L'MMU tradueix les adreces lògiques a adreces físiques en temps d'execució; la memòria física, de 16 KiB, té quatre marcs, i la pàgina B del procés 1 només és al disc. Al disc es dibuixen únicament les pàgines que no són a la memòria física.

8.1.3 El disc com a nivell addicional de la jerarquia

Una de les motivacions originals de la memòria virtual era donar solució al problema de no poder encabir tot un programa a la memòria física, la mida de la qual era antigament molt limitada. Tradicionalment, això s'havia solucionat mitjançant la tècnica de les superposicions (*overlays*): el programador divideix el programa en blocs, que va movent selectivament entre la memòria física i el disc, mantenint a la memòria únicament els blocs indispensables en cada moment. La memòria virtual funciona d'una manera semblant, ja que també fa servir el disc per emmagatzemar temporalment les parts d'un programa que no s'estan fent servir, però això ho gestiona el sistema operatiu, amb l'ajuda del maquinari, i allibera els programadors d'aquesta tasca.

En un computador amb memòria virtual, el disc (o SSD) passa a ser una capa més de la jerarquia de memòria de la Secció 7.1.4 (Figura 8.2). De la mateixa manera que la memòria cau només conté un subconjunt de les dades del nivell inferior (de la memòria principal), la memòria física d'un computador amb memòria virtual només conté un subconjunt de les dades necessàries per a l'execució d'un programa. Totes les dades són sempre al nivell inferior, el disc, que es troba a la base de la jerarquia.

8.2 Paginació

8.2.1 Pàgines i marcs de pàgina

Els fonaments de la memòria virtual són els mateixos que els de les memòries cau del Capítol 7: un nivell de la jerarquia conté una part de les dades del nivell inferior, organitzades en blocs de mida fixa, i quan la dada no hi és cal anar-la a buscar. Com que els dos mecanismes tenen orígens històrics

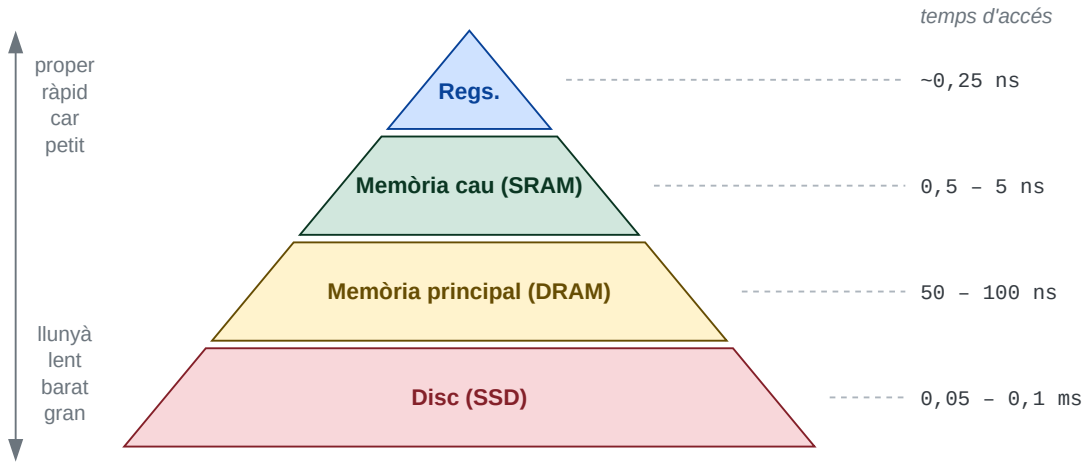
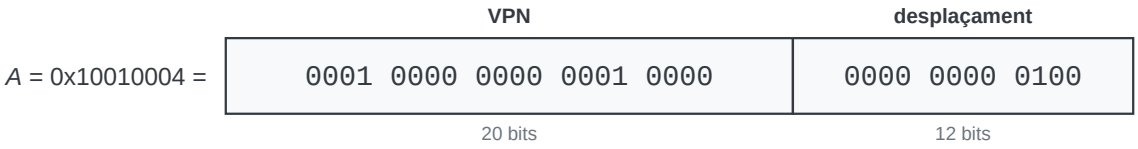


Figura 8.2: Jerarquia de memòria en un computador amb memòria virtual. Els temps d'accés són ordres de magnitud orientatius, que depenen del processador, de la memòria i del dispositiu concrets (vegeu Taula 7.1 per a dades de capacitat i cost).

diferents, però, la terminologia canvia: els blocs s'anomenen **pàgines**, i les fallades, **fallades de pàgina**.

La memòria virtual gestiona la traducció d'adreces, doncs, en blocs de mida fixa anomenats **pàgines**. L'espai d'adreçament lògic de cada procés es divideix en pàgines de T bytes (habitualment $T = 4$ KiB $= 2^{12}$ bytes), i cada pàgina rep un **número de pàgina virtual** (*virtual page number*, VPN), que indica la seva posició dins l'espai lògic, començant per 0. En aquest tema es tracta només la memòria virtual **paginada**; també existeix la memòria virtual **segmentada**, que divideix l'espai lògic en segments de mida variable, i que avui és residual.

Donada una adreça lògica A d'un programa, per exemple la d'una dada, es pot determinar a quin número de pàgina virtual correspon fent la divisió entera entre l'adreça i la mida de pàgina, VPN $= \lfloor A/T \rfloor$. Si, per exemple, $A =$ `0x10010004` i $T = 4$ KiB $= 2^{12}$ bytes, A és a la pàgina virtual amb VPN = `0x10010`. Com que es treballa amb valors de T que són potències de dos ($T = 2^t$), determinar el VPN consisteix simplement a agafar l'adreça i descartar-ne els t bits de menys pes. Aquests t bits de menys pes indiquen quina és la ubicació de la dada dins la pàgina, i s'anomenen **desplaçament** (*page offset*).



Cada computador disposa d'una determinada capacitat de memòria física, que ocupa un rang d'adreces anomenat espai d'adreçament físic, totalment independent del rang d'adreces lògiques (pot ser més petit, igual o més gran). L'espai d'adreçament físic se subdivideix en blocs de mida T, la mateixa que les pàgines lògiques, anomenats **marcs de pàgina** (*page frames*), que es poden imaginar com a «contenidors» on es pot allotjar una pàgina lògica. Cada marc de pàgina té associat un **número de pàgina física** (*physical page number*, PPN), que indica la posició que ocupa dins l'espai d'adreçament físic, començant pel 0.

Cada procés consta d'un cert nombre de pàgines virtuals, no totes necessàriament contigües, que ocupen unes adreces determinades en temps de compilació. Les adreces d'aquestes pàgines virtuals són totalment independents de les adreces físiques que ocuparan després, en temps d'execució. Inicialment, un procés resideix a l'emmagatzematge secundari (el disc), i durant l'execució el sistema operatiu haurà

d'anar carregant del disc a la memòria física les pàgines que es vagin necessitant. Si no hi queden marcs de pàgina lliures, el sistema n'haurà de reemplaçar un d'ocupat, i possiblement haurà d'escriure'n la pàgina al disc abans de carregar-hi la nova. En un moment donat, les pàgines d'un procés que no són a la memòria física són al disc. Es deixen per a més endavant els detalls de com es llegeixen i s'escriuen les pàgines al disc. Per simplificar, se suposa que **la memòria física conté un subconjunt de les pàgines d'un procés, la totalitat de les quals es troba al disc**: el disc fa, respecte de la memòria física, el mateix paper que la memòria física feia respecte de la memòria cau al Capítol 7. (En un sistema real, les pàgines que un procés crea durant l'execució, com les de la pila, no tenen còpia al disc fins que el sistema operatiu no les hi ha d'escriure per fer lloc.)

La Figura 8.3 mostra una disposició possible de les pàgines de dos processos en un moment determinat. Amb adreces de 32 bits i una mida de pàgina $T = 4$ KiB $= 2^{12}$ bytes, l'espai d'adreçament lògic de cada procés és de 2^{32} bytes $= 4$ GiB, és a dir, 2^{20} pàgines virtuals ($2^{32}/2^{12}$). A l'exemple, però, els dos processos només fan servir dues pàgines virtuals cadascun (VPN 0 i VPN 1). La memòria física és de 2^{14} bytes $= 16$ KiB, és a dir, 4 marcs de pàgina ($2^{14}/2^{12}$): PPN 0, PPN 1, PPN 2 i PPN 3. Algunes pàgines estan carregades a la memòria física (VPN 0 del procés 1, i VPN 0 i VPN 1 del procés 2) i una altra (VPN 1 del procés 1) només és al disc. L'assignació de pàgines a marcs de pàgina la fa el sistema operatiu en funció de l'espai disponible en cada moment, de manera similar a com funciona una memòria cau completament associativa (Secció 7.3.3).

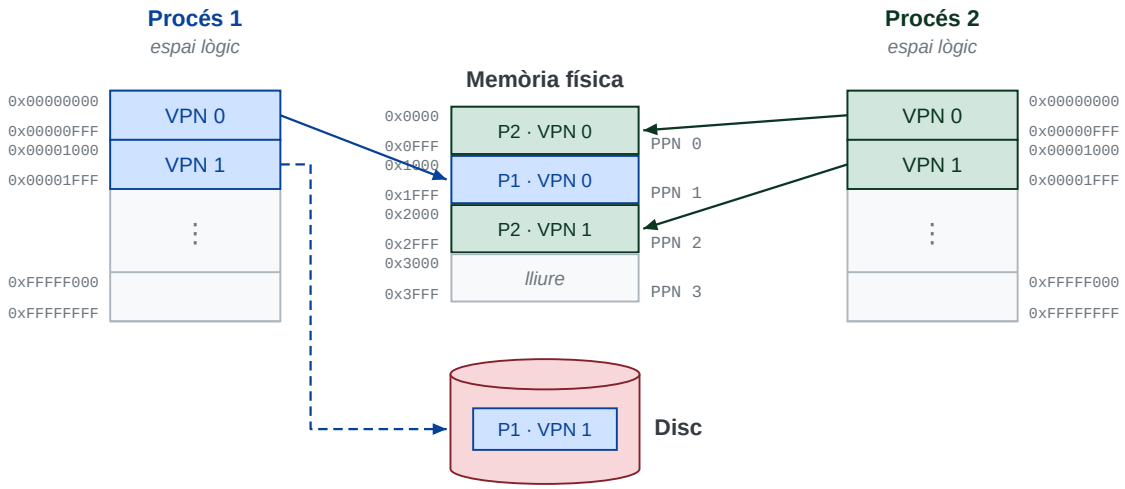


Figura 8.3: Els VPN de dos processos, a l'esquerra, i els PPN d'una memòria de 16 KiB, a la dreta. Les pàgines lògiques s'assignen als marcs de pàgina de manera completament associativa; al disc només es dibuixen les que no són a la memòria física.

8.2.2 Traducció d'adreces: VPN, PPN i desplaçament

Com que T és una potència de dos ($T = 2^t$), qualsevol adreça lògica es descompon naturalment en dos camps:

- Els t **bits de menys pes** indiquen la posició de la dada dins la pàgina: és el **desplaçament** (*page offset*).
- Els **bits restants** identifiquen la pàgina: és el **VPN**.

En un computador amb memòria virtual, el processador treballa amb adreces lògiques. Cada vegada que necessita llegir o escriure una dada de la memòria (per exemple, per fer la captura d'una instrucció), demana l'adreça lògica d'aquesta dada a l'MMU. Mitjançant un procés anomenat **traducció d'adreces**, l'MMU tradueix l'adreça lògica sol · licitada a l'adreça física on realment es troba la dada. Com que les

dades estan agrupades en pàgines, l'únic que ha de fer el sistema de traducció és, donat el número de pàgina (VPN) a què pertany l'adreça, determinar en quin marc de pàgina (PPN) de la memòria física es troba. L'adreça física de la dada s'obté reemplaçant, a l'adreça lògica, els bits corresponents al VPN pel PPN; el desplaçament es conserva intacte. La Figura 8.4 il · lustra el mecanisme de traducció amb les dades de l'exemple anterior: adreces de 32 bits, mida de pàgina $T = 4$ KiB $= 2^{12}$ bytes i memòria física de 16 KiB $= 2^{14}$ bytes.

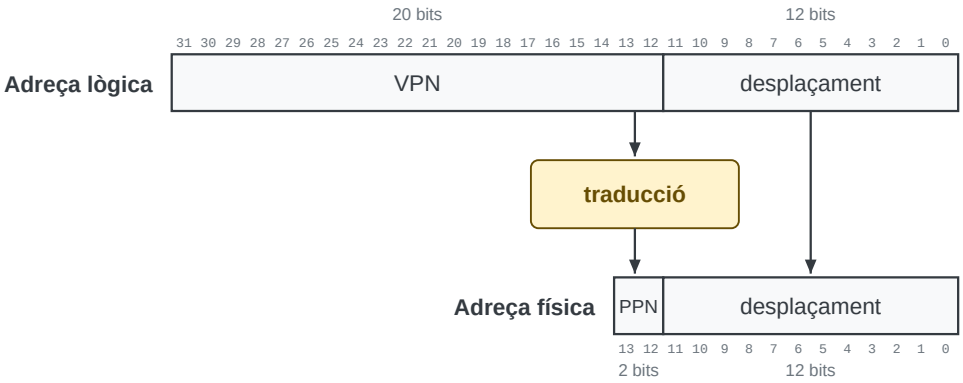


Figura 8.4: Traducció d'una adreça lògica a una adreça física: el VPN (20 bits) es tradueix a PPN (2 bits) i el desplaçament (12 bits) es conserva.

8.3 La taula de pàgines

Per poder traduir una adreça lògica cal saber en quin marc de pàgina (PPN) es troba cada pàgina virtual (VPN). Aquesta informació la manté una estructura de dades anomenada **taula de pàgines**. Cada procés té la seva pròpia taula de pàgines, emmagatzemada a la memòria principal.

La taula de pàgines s'indexa pel VPN (Figura 8.5): cada fila correspon a una pàgina virtual i s'anomena **entrada de la taula de pàgines** (*page table entry*, PTE). El nombre d'entrades és igual al nombre de pàgines virtuals de l'espai d'adreçament (2^{20} entrades en l'exemple de 32 bits amb pàgines de 4 KiB).

8.3.1 Estructura d'una entrada (PTE)

Cada PTE conté els camps següents:

- **PPN**: el número de marc de pàgina on es troba la pàgina, si és a la memòria física.
- **Bit V** (*valid*, validesa): val 1 si la PTE conté una traducció utilitzable, és a dir, si la pàgina és a la memòria física; val 0 si no hi és (perquè és al disc o perquè l'adreça no pertany a l'espai vàlid del procés).
- **Bit de modificació** (***dirty bit***), D: val 1 si la pàgina ha estat modificada des que es va carregar. Quan cal reemplaçar-la, només s'escriu al disc si D = 1.
- **Bit E** (escriptura): indica si el procés té permís d'escriptura sobre aquesta pàgina. Si una instrucció intenta escriure en una pàgina amb E = 0, es produeix una excepció de protecció (Secció 8.6.2).

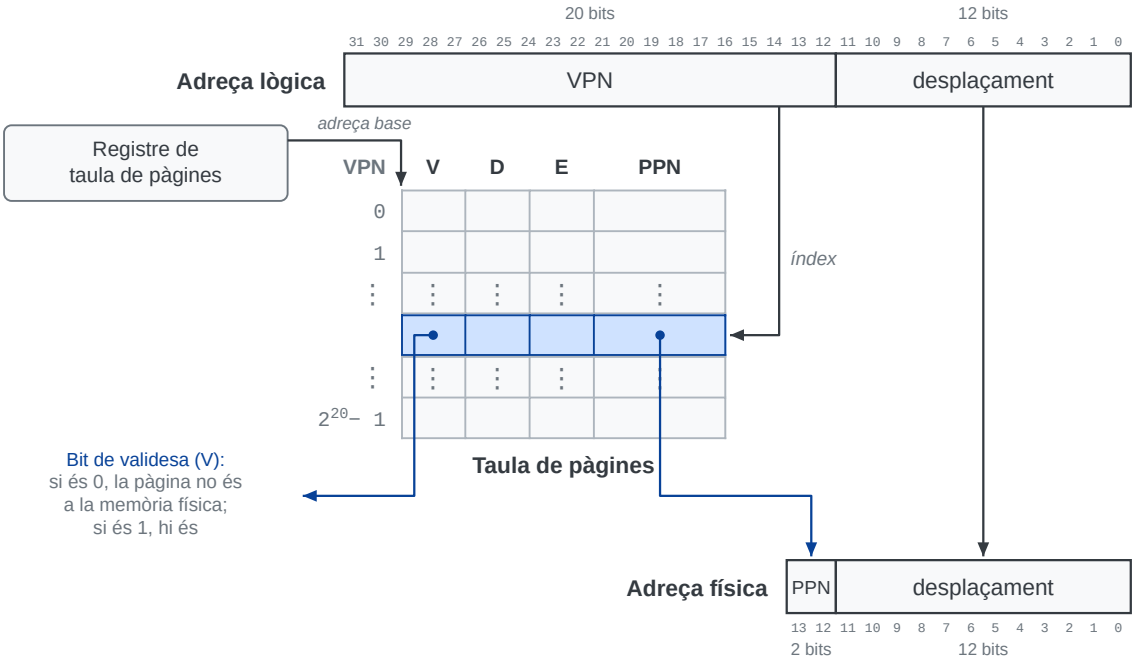


Figura 8.5: Donat un VPN, la taula de pàgines indica si la pàgina és a la memòria física (bit V) i en quin marc de pàgina (PPN) es troba; el PPN, juntament amb el desplaçament, forma l'adreça física. El registre de taula de pàgines (Secció 8.3.2) en conté l'adreça base, i cada entrada (PTE) conté, a més, els bits D i E.

Essencial 8.1: Estructura d'una PTE

Camp	Mida	Significat
PPN	p bits	Marc de pàgina física on es troba la pàgina
V	1 bit	1 = PTE vàlida (pàgina a memòria); 0 = pàgina al disc o adreça invàlida
D	1 bit	1 = pàgina modificada (*dirty*)
E	1 bit	1 = permís d'escriptura

Quan V = 0, el camp PPN no té validesa: la pàgina no és a la memòria física.

Aprofundiment 8.1: Notació: `V` i `E` a EC, `P` i `W` en altres fonts

Aquests apunts anomenen **`V`** (*valid*) el bit de validesa de la PTE, com fan l'ISA de RISC-V (esquema Sv32) i Harris & Harris. Alguns textos i els exàmens anteriors d'EC l'anomenen **`P`** (*present*), com fa l'arquitectura x86; en consultar-los, cal tenir en compte aquesta diferència de notació.

El bit **`E`**, en canvi, és una simplificació d'aquests apunts: una PTE real de Sv32 no té cap bit `E`, sinó tres bits de permís, **`R`** (lectura), **`W`** (escriptura) i **`X`** (execució), a més dels bits **`U`** (accessible en mode usuari), **`G`** (traducció global, comuna a tots els processos) i **`A`** (*accessed*, pàgina accedida). El bit `E` d'EC correspon al bit `W` de Sv32, i el bit `U` és el que fa inaccessible el nucli des del mode usuari: les pàgines del nucli són a la taula de pàgines de cada procés, però amb U = 0, de manera que qualsevol accés en mode U hi provoca una fallada de pàgina (Secció 8.6.1). La resta de

permisos no es tracten en aquest tema. Pel que fa a `V`, Sv32 només estableix que, amb V = 0, la PTE és invàlida i qualsevol accés provoca una fallada de pàgina: que la pàgina sigui al disc o que l'adreça no pertanyi al procés és informació que gestiona el sistema operatiu, no el maquinari. De fet, quan V = 0 la resta de camps de la PTE no tenen cap significat per al maquinari, i el sistema operatiu els pot fer servir per recordar, per exemple, on és la pàgina al disc.

La Figura 8.6 mostra com es faria la traducció de l'adreça lògica `0x00001801` mitjançant la taula de pàgines del procés 2 de la Figura 8.3. Amb pàgines de 4 KiB, el VPN són els 20 bits de més pes de l'adreça (VPN = 1) i el desplaçament, els 12 de menys pes (`0x801`). L'entrada 1 de la taula té V = 1, de manera que la pàgina és a la memòria física, al marc PPN = 2 (`10` en binari). L'adreça física s'obté reemplaçant el VPN pel PPN i conservant el desplaçament: `10 1000 0000 0001` = `0x2801`. Com que la memòria física és de 16 KiB, l'adreça física té 14 bits: 2 de PPN i 12 de desplaçament.

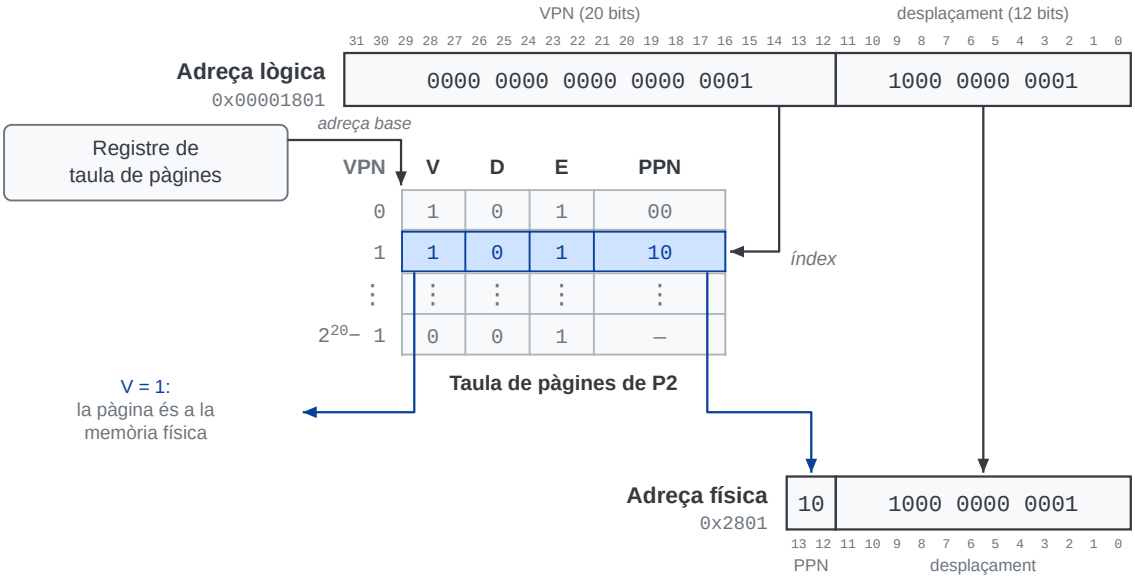


Figura 8.6: Traducció de l'adreça lògica `0x00001801` mitjançant la taula de pàgines del procés 2 de la Figura 8.3.

8.3.2 El registre de taula de pàgines i el canvi de context

En un sistema **multiprogramat**, diversos processos s'executen de manera concurrent en una sola CPU per **temps compartit** (*time sharing*): el sistema operatiu reparteix el temps de la CPU en períodes curts (fraccions de segon) i els alterna entre els processos, de manera que semblen executar-se alhora. En cada instant només hi ha un **procés actiu**, i quan el sistema operatiu en canvia un per un altre ha de desar l'estat del que deixa d'executar-se i restaurar el del que passa a ser actiu: els registres de propòsit general, el PC i altres estructures de dades. Aquest estat s'anomena **context** del procés.

En un sistema amb memòria virtual, cada procés té la seva pròpia taula de pàgines a la memòria. Per saber on és la taula del procés actiu, el maquinari incorpora un **registre de taula de pàgines** que conté l'adreça base de la taula del procés actiu. L'MMU usa aquest registre per accedir a la PTE corresponent a cada VPN que cal traduir.

Quan el sistema operatiu commuta d'un procés a un altre (**canvi de context**), actualitza el registre de taula de pàgines perquè apunti a la taula del nou procés actiu. Aquesta operació és suficient per canviar completament l'espai d'adreçament visible: a partir d'aquell moment, totes les traduccions es

fan amb la taula del nou procés, i l'espai lògic de l'anterior queda inaccessible. El valor del registre de taula de pàgines forma part, doncs, del context del procés.

8.3.3 Taules de pàgines multinivell

La taula de pàgines plana descrita fins ara ocupa una quantitat de memòria proporcional al nombre de pàgines virtuals possibles, independentment de quantes en faci servir realment el procés. En sistemes de 32 bits amb pàgines de 4 KiB i entrades de 4 bytes, això representa 4 MiB per procés en tot moment; en sistemes de 64 bits, el cost es fa inassumible. Les **taules de pàgines multinivell** resolen el problema assignant memòria només per a les regions de l'espai lògic que el procés utilitza realment (Figura 8.7).

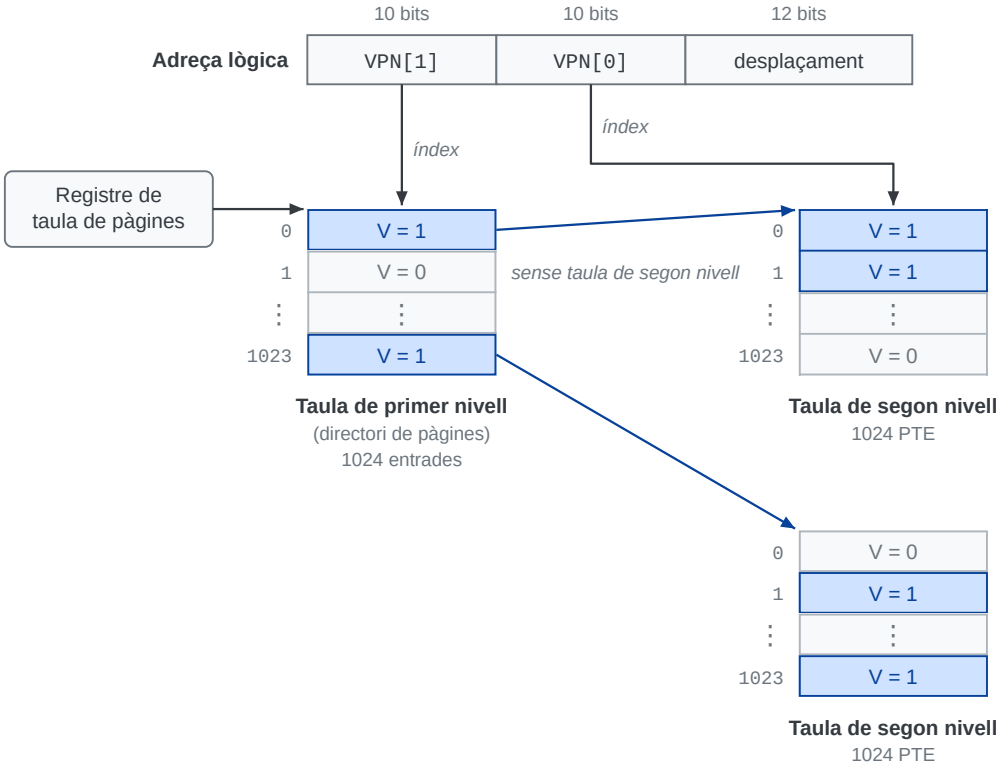


Figura 8.7: Taula de pàgines de dos nivells (Sv32). Els 10 bits de més pes del VPN, VPN[1], indexen la taula de primer nivell, i els 10 següents, VPN[0], la de segon nivell. Les taules de segon nivell només s'assignen per a les regions de l'espai lògic en ús, cosa que redueix la memòria que ocupa la taula de pàgines.

> **Aprofundiment 8.2: Taules de pàgines multinivell**
>
> En un sistema de 32 bits amb pàgines de 4 KiB, la taula de pàgines té 2^{20} entrades. Si cada entrada ocupa 4 bytes, la taula ocupa 4 MiB per procés, tots a la memòria principal. En sistemes de 64 bits el problema s'aguditza: amb adreces de 48 bits significatius i pàgines de 4 KiB, la taula tindria 2^{36} entrades, cosa inviable.
> La solució és estructurar la taula de pàgines en **múltiples nivells**. En una taula de dos nivells, els bits del VPN es divideixen en dues parts: els bits superiors indexen una **taula de primer nivell** (o directori de pàgines), on cada entrada apunta a una **taula de segon nivell**. Cada taula de segon nivell conté les PTE d'un subconjunt de l'espai lògic.

L'avantatge clau és que les taules de segon nivell només s'assignen per a les regions de l'espai lògic que el procés utilitza realment. Com que la majoria dels processos fan servir només una fracció petita dels 2^{32} o 2^{48} bytes d'espai lògic possible, la memòria ocupada per la taula de pàgines es redueix dràsticament. El nombre de nivells depèn de l'arquitectura i de la configuració del sistema: per exemple, a x86-64 Linux fa servir habitualment quatre nivells (cinc en els processadors que ho admeten), i a RISC-V de 64 bits, tres (Sv39), quatre (Sv48) o cinc (Sv57), segons el que admeti el processador i triï el nucli en arrencar.

⚠ Aprofundiment 8.3: L'esquema Sv32 de RISC-V

Sv32 és l'esquema estàndard de paginació de 32 bits definit per l'arquitectura privilegiada de RISC-V. Tradueix adreces lògiques de 32 bits a adreces físiques de **34 bits** mitjançant una taula de pàgines de **dos nivells** amb pàgines de 4 KiB. Per tant, tot i que cada procés disposa d'un espai lògic de 4 GiB (2^{32} bytes), el sistema pot adreçar fins a **16 GiB** (2^{34} bytes) de memòria física.
Per a sistemes de 64 bits, RISC-V defineix els esquemes **Sv39**, **Sv48** i **Sv57**, amb taules de pàgines de tres, quatre i cinc nivells respectivament, que permeten espais d'adreçament molt més grans.

8.4 Fallada de pàgina

Quan la CPU accedeix a una adreça lògica i el bit V de la PTE corresponent val 0, es produeix una **fallada de pàgina** (*page fault*). Això pot succeir per dos motius: la pàgina pertany a l'espai vàlid del procés, però no és a la memòria física (és al disc), o bé l'adreça és invàlida (no pertany a cap regió reservada del procés). En el segon cas, el sistema operatiu avorta l'execució del programa (a Unix i derivats, amb un error de tipus *segmentation fault*).

La fallada de pàgina és una **excepció**: la instrucció que l'ha provocat s'interromp i el processador passa el control a una rutina del sistema operatiu, que en el primer cas resol la fallada carregant la pàgina des del disc:

1. Es localitza la pàgina al disc.
2. Si hi ha un marc de pàgina lliure, s'hi carrega la pàgina. Si no n'hi ha cap de lliure, cal **reemplaçar** un marc ocupat: es tria una pàgina víctima, s'escriu al disc si ha estat modificada (D = 1) i es posa V = 0 a la seva PTE, perquè deixa de ser a la memòria física.
3. S'actualitza la PTE de la pàgina nova: es posa V = 1, D = 0 i s'escriu el PPN del marc assignat.
4. Es reexecuta la instrucció que ha causat la fallada, que ara troba la pàgina a la memòria.

Com que un accés al disc és centenars o milers de vegades més lent que un accés a la memòria principal (desenes de milers, o més, en el cas d'un disc magnètic), la fallada de pàgina és, amb diferència, l'incident més costós de tot el sistema de memòria. Per aquest motiu, les dades no es llegeixen ni s'escriuen mai al disc individualment, sinó sempre per pàgines senceres: primer es carrega la pàgina a la memòria física i després s'hi accedeix. Les fallades de pàgina que requereixen un accés al disc s'anomenen **fallades majors** (*major page faults*); els sistemes reals tenen també **fallades menors** (*minor page faults*), que el sistema operatiu resol sense accedir al disc (per exemple, quan la pàgina ja era a la memòria física, però encara no tenia PTE, com en una pàgina nova de la pila), i que no es consideren en aquest tema.

Essencial 8.2: Mecanisme de la fallada de pàgina

Una fallada de pàgina es produeix quan V = 0 en la PTE del VPN sol · licitat. És una excepció: si la pàgina és vàlida, el sistema operatiu la carrega des del disc (reemplaçant-ne una altra si no hi ha cap marc lliure), actualitza les PTE afectades i reexecuta la instrucció causant. Si l'adreça és invàlida, el procés s'avorta.

Quan no hi ha cap marc lliure, el sistema operatiu n'ha de reemplaçar un, seguint un algorisme de reemplaçament. A l'assignatura es fa servir l'algorisme *least recently used* (LRU), com a les memòries cau (Secció 7.4): es reemplaça la pàgina que fa més temps que no s'utilitza. Abans de sobreescriure el marc seleccionat, comprova el bit D de la seva PTE: si D = 1, la pàgina ha estat modificada i cal escriure-la primer al disc per no perdre els canvis; si D = 0, es pot sobreescriure directament. Aquesta política és equivalent a l'escriptura retardada amb assignació que es fa servir a les memòries cau (Secció 7.5.3). Si el processador guarda traduccions en un TLB (Secció 8.5), el sistema operatiu també n'ha d'invalidar l'entrada de la pàgina víctima, com s'explica a la Secció 8.5.4.

La zona del disc on s'emmagatzemen les pàgines reemplaçades s'anomena **espai d'intercanvi** (*swap space*).

Aprofundiment 8.4: Reemplaçament de pàgines als sistemes reals

El mecanisme descrit és el mínim que cal per fer funcionar la memòria virtual. Els sistemes operatius reals hi afegeixen diverses optimitzacions:

- **LRU aproximat**: als exemples i problemes d'aquest tema es fa servir un LRU exacte, que és determinista i fàcil de seguir a mà. Els sistemes operatius reals no poden mantenir l'ordre d'ús exacte de milions de pàgines, i l'aproximen amb un **bit de referència** (*reference bit*, el bit A d'una PTE de RISC-V) que el maquinari posa a 1 en cada accés i el sistema operatiu esborra periòdicament, o amb algorismes més elaborats.
- **Lectura anticipada** (*anticipatory paging* o *prefetching*): en resoldre una fallada de pàgina, el sistema no es limita a llegir del disc la pàgina sol · licitada, sinó que aprofita l'accés per llegir-ne algunes més de contigües, amb l'expectativa que el programa les necessiti aviat (localitat espacial).
- **Reemplaçament global o local**: la pàgina víctima es pot triar entre totes les pàgines residents de tots els processos (*global page replacement*, el que fa Linux) o només entre les del procés que ha causat la fallada (*local page replacement*), que acota la memòria que pot ocupar cada procés.
- **Neteja anticipada** (*precleaning*): en lloc d'esperar que una pàgina modificada sigui triada com a víctima per escriure-la al disc, el sistema aprofita els moments en què el disc està desocupat per escriure-hi les pàgines amb D = 1 i posar-les a D = 0; així, quan calgui reemplaçar-les, no caldrà esperar l'escriptura.

8.5 Traducció ràpida: el TLB

La taula de pàgines és a la memòria principal. Això implica que cada accés a memòria requereix, en realitat, dos accessos: un per llegir la PTE i traduir l'adreça, i un per llegir o escriure la dada (amb una taula de pàgines multinivell, encara més). Per evitar aquest cost, els processadors incorporen a l'MMU una memòria cau de traduccions anomenada *translation lookaside buffer* (TLB), que emmagatzema les PTE utilitzades més recentment. El TLB és una memòria cau com les del Capítol 7, però de traduccions en lloc de dades: en guarda poques (de l'ordre de desenes o centenars), s'hi accedeix en una fracció del temps d'un accés a memòria, i té els seus propis encerts i fallades.

EC 8.1: El model de TLB d'EC: canvi respecte del temari anterior (MIPS)

El TLB no forma part de l'ISA de RISC-V: l'arquitectura defineix com es tradueix una adreça amb la taula de pàgines, però deixa a cada implementació si guarda traduccions en una memòria cau, i amb quina organització. El TLB d'aquest tema és un **model**, el mateix per a la teoria i els problemes: completament associatiu, amb reemplaçament LRU i amb les entrades que descriu la Secció 8.5.1. L'únic que l'ISA fixa, i que aquest model respecta, és que el sistema operatiu mai no veu les fallades de TLB, i que ha d'invalidar les traduccions d'una pàgina quan en modifica la PTE; el model d'EC, a més, només hi guarda traduccions vàlides. Al temari anterior d'EC, basat en MIPS, el TLB formava part de l'ISA i podia contenir entrades de pàgines no residents (Aprofundiment 8.5).

8.5.1 Estructura i funcionament

Cada entrada del TLB conté una còpia d'una PTE **vàlida** (camps D, E i PPN) i el VPN al qual correspon, que fa de clau de cerca (Figura 8.8). A diferència de la taula de pàgines, que s'indexa directament pel VPN, el TLB funciona de manera **completament associativa**, com les memòries cau d'aquest tipus (Secció 7.3.3): per traduir una adreça, es compara el VPN amb els VPN de totes les entrades simultàniament.

Cada entrada té també un bit de validesa **V**, amb el mateix paper que el bit de validesa d'una línia de memòria cau: val 1 si l'entrada conté una traducció i 0 si és lliure. En engegar el processador totes les entrades tenen V = 0; una entrada torna a quedar lliure quan el sistema operatiu la invalida (Secció 8.5.4). No s'ha de confondre amb el bit V de la PTE: al TLB només s'hi copien PTE amb V = 1, de manera que **tota entrada vàlida del TLB correspon a una pàgina resident** a la memòria física.

Essencial 8.3: Estructura d'una entrada del TLB

Camp	Significat
V	1 = l'entrada conté una traducció; 0 = entrada lliure
VPN	Identificador de la pàgina virtual (clau de cerca)
D	1 = pàgina modificada (*dirty*), copiat de la PTE
E	1 = permís d'escriptura, copiat de la PTE
PPN	Marc de pàgina física corresponent, copiat de la PTE

El TLB només guarda traduccions de pàgines residents: una PTE amb V = 0 no es copia mai al TLB.

8.5.2 Encert de TLB

Quan cal traduir una adreça, l'MMU extreu el VPN i el cerca entre les entrades vàlides del TLB. Si en troba una amb aquest VPN, es produeix un **encert de TLB** (*TLB hit*): la pàgina és resident, i la traducció es completa directament amb el PPN d'aquella entrada, sense accedir a la taula de pàgines a la memòria. El cost de la traducció es redueix al temps d'accés al TLB, que és molt inferior al d'un accés a memòria.

Abans d'acabar l'accés, l'MMU comprova els permisos amb els bits de l'entrada: si la instrucció és una escriptura i E = 0, es produeix una excepció de protecció (Secció 8.6.2); si és una escriptura i D = 0,

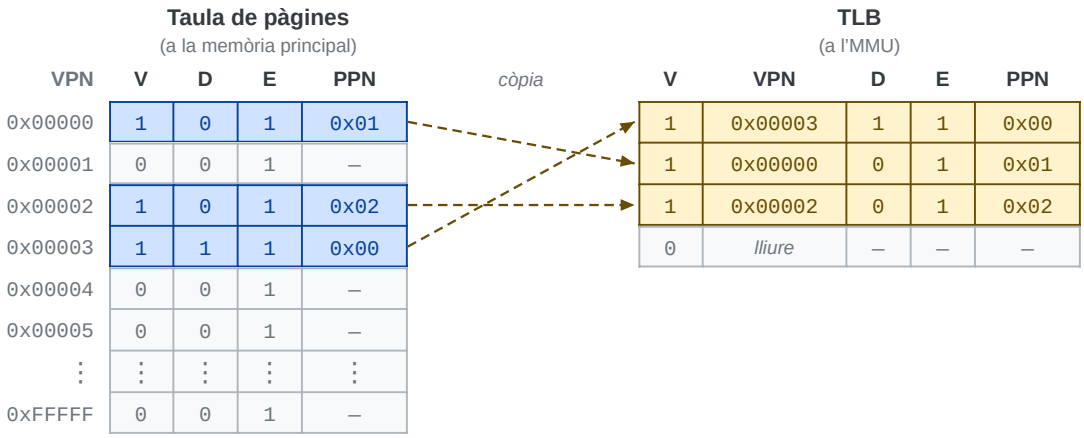


Figura 8.8: El TLB emmagatzema una còpia de les entrades de la taula de pàgines utilitzades més recentment, i cada entrada inclou el VPN com a clau de cerca. Són les dades inicials de Exemple 8.2.

és la primera vegada que s'escriu a la pàgina i cal posar D a 1, tant a l'entrada del TLB com a la PTE de la taula de pàgines (Secció 8.5.5).

8.5.3 Fallada de TLB

Si el VPN no es troba en cap entrada vàlida del TLB, es produeix una **fallada de TLB** (*TLB miss*). L'MMU inclou un circuit, el **recorregut de la taula de pàgines** (*page-table walker*), que llegeix la PTE corresponent de la taula de pàgines a la memòria principal (a partir del registre de taula de pàgines i del VPN) **sense interrompre el programa**: la fallada de TLB és transparent per al programari, igual que una fallada de memòria cau. Segons el bit V de la PTE llegida:

- Si $\mathbf{V = 1}$, la pàgina és resident. La PTE es copia en una entrada del TLB —una de lliure (V = 0) si n'hi ha, o la que triï l'algorisme de reemplaçament (LRU, aleatori, FIFO, etc.) si totes són ocupades— i la traducció continua com en un encert.
- Si $\mathbf{V = 0}$, la pàgina no és a la memòria física. **No es copia res al TLB** i es genera l'excepció de fallada de pàgina (Secció 8.4), que resol el sistema operatiu. Quan la instrucció es reexecuta, torna a fallar al TLB, però ara el recorregut troba la PTE amb V = 1 i la instal · la.

El cost d'una fallada de TLB és, doncs, el dels accessos a memòria necessaris per llegir la PTE: un amb la taula de pàgines plana d'aquest tema, i un per nivell amb una taula multinivell.

Essencial 8.4: Encert i fallada de TLB

Encert de TLB (*TLB hit*): el VPN és en una entrada vàlida del TLB. La pàgina és resident i la traducció es completa amb el PPN de l'entrada, sense accedir a la memòria principal.
Fallada de TLB (*TLB miss*): el VPN no és al TLB. L'MMU llegeix la PTE de la taula de pàgines. Si V = 1, la copia al TLB (entrada lliure o reemplaçament) i la traducció continua. Si V = 0, es produeix una fallada de pàgina i no es copia res.

> ⚠ Aprofundiment 8.5: El TLB gestionat per programari: el model de MIPS
>
> A RISC-V, a x86 i a ARM, la fallada de TLB la resol el maquinari, com s'ha explicat, i el sistema operatiu només intervé en les fallades de pàgina. L'arquitectura **MIPS**, en què es basaven els apunts anteriors d'EC, seguia un model diferent, el del **TLB gestionat per programari** (*software-managed TLB*): el TLB formava part de l'ISA, amb instruccions per llegir-ne i escriure'n les entrades, i la fallada de TLB era una **excepció**. El sistema operatiu executava una rutina que llegia la PTE de la taula de pàgines i la copiava al TLB —encara que tingués V = 0—, i reexecutava la instrucció; si la PTE copiada tenia V = 0, l'encert següent al TLB provocava la fallada de pàgina. En aquell model, doncs, el TLB podia contenir entrades de pàgines no residents, i el bit V d'una entrada del TLB era una còpia del bit V de la PTE.
> Cal tenir-ho en compte en consultar exàmens i problemes antics d'EC: hi apareixen encerts de TLB amb V = 0, que en el model d'aquests apunts no poden passar. Els resultats observables (fallades de pàgina, escriptures al disc i adreces físiques) són els mateixos en tots dos models; el que canvia és el contingut del TLB en cada moment.
> RISC-V permet que una implementació resolgui les fallades de TLB per programari, però com una extensió del mode màquina (M) invisible per al sistema operatiu: el nucli, que s'executa en mode supervisor (S), només veu fallades de pàgina. El Tema 9 (Secció 9.7) presenta la rutina de fallada de pàgina del nucli amb codi RISC-V.

8.5.4 Entrades lliures i invalidació del TLB

Una entrada del TLB pot estar lliure (V = 0) per dos motius: perquè no s'ha fet servir mai des que s'ha engegat el processador, o perquè el sistema operatiu l'ha **invalidat**.

El TLB és una còpia parcial de la taula de pàgines, i cal mantenir-les coherents. Quan el sistema operatiu modifica una PTE —en reemplaçar una pàgina, en canviar-ne els permisos o en canviar de procés—, les entrades del TLB que en tinguin una còpia queden obsoletes, i el maquinari no ho pot detectar, perquè el TLB no observa les escriptures a la taula de pàgines. Per això el sistema operatiu, després de modificar la PTE, ha d'**invalidar** les entrades afectades del TLB (a RISC-V, amb la instrucció `sfence.vma`, que presenta el Tema 9 a Secció 9.7.3), posant-les a V = 0. En el cas del reemplaçament de pàgina, l'entrada de la pàgina víctima queda lliure, i és la primera candidata a rebre la traducció de la pàgina nova. En el canvi de context, com que la taula de pàgines canvia sencera, s'invaliden totes les entrades (tret que cada entrada del TLB identifiqui el procés a què pertany, com permet el camp ASID del registre `satp` de RISC-V; vegeu RISC-V 9.15).

8.5.5 El bit D: consistència amb la taula de pàgines

El bit D és l'únic camp del TLB que l'execució del programa pot modificar: quan una instrucció d'escriptura (`sw`, `fsw`, etc.) accedeix a una pàgina amb D = 0, el bit D s'ha de posar a 1, i la taula de pàgines ho ha de saber, perquè el sistema operatiu consulta la PTE quan ha de reemplaçar la pàgina.

A EC se suposa que l'actualització del bit D **la fa el maquinari**, sense cap excepció ni intervenció del sistema operatiu: l'MMU segueix una política d'**escriptura immediata** i, a la primera escriptura en una pàgina (D = 0 al TLB), posa D = 1 simultàniament a l'entrada del TLB i a la PTE de la taula de pàgines. Això requereix un accés addicional a la memòria principal, però només succeeix la primera vegada que s'escriu en cada pàgina, no en cada instrucció d'escriptura. És una de les dues opcions que admet RISC-V per actualitzar els bits A i D de la PTE; l'altra, que ho faci el sistema operatiu a través d'una excepció, no es fa servir, i s'explica a l'aprofundiment Aprofundiment 9.7 del Tema 9.

8.5.6 Flux complet de traducció

La Figura 8.9 resumeix totes les situacions possibles durant la traducció d'una adreça en un sistema amb TLB i memòria virtual. Les dues columnes de l'esquerra, l'encert i la fallada de TLB, les resol el maquinari (l'MMU) sense interrompre el programa; la de la dreta, la fallada de pàgina, és una excepció que resol el sistema operatiu i acaba reexecutant la instrucció.

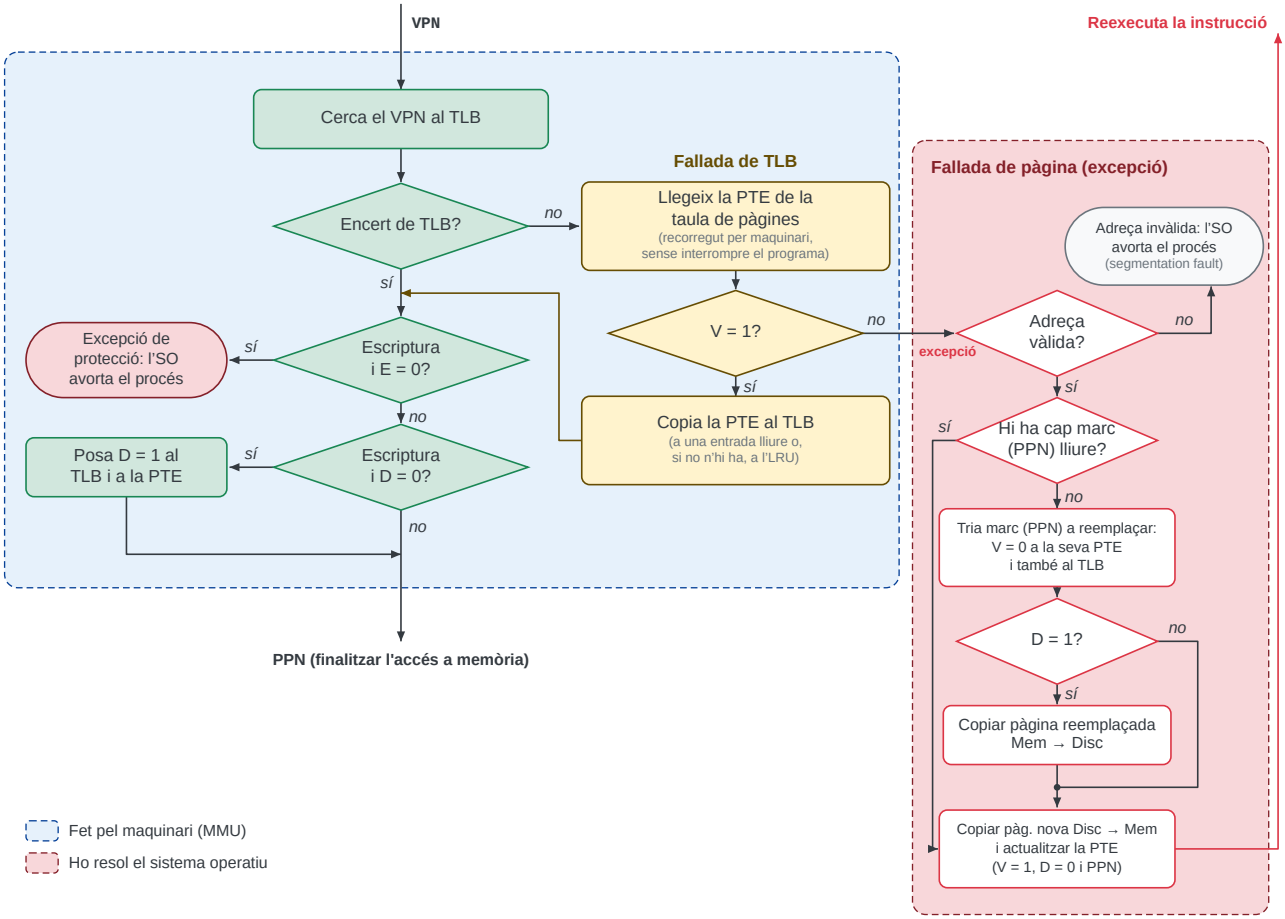


Figura 8.9: Flux complet de traducció d'una adreça en un sistema amb TLB i memòria virtual. A l'esquerra, l'encert de TLB; al centre, la fallada de TLB, que resol l'MMU sense interrompre el programa, i a la dreta, la fallada de pàgina, una excepció que resol el sistema operatiu.

> **Aprofundiment 8.6:** *TLB shootdown* en sistemes multinucli
>
> En un processador amb múltiples nuclis, cada nucli disposa del seu propi TLB. Quan el sistema operatiu modifica una PTE de la taula de pàgines d'un procés (per exemple, en reemplaçar una pàgina o revocar un permís d'escriptura), qualsevol nucli que tingui una còpia d'aquella entrada al seu TLB passa a tenir informació obsoleta.
> Per garantir la coherència, el sistema operatiu ha d'invalidar les entrades afectades als TLB de tots els nuclis que puguin tenir-les, no només al nucli on s'executa (la instrucció `sfence.vma` de RISC-V només actua sobre el TLB del nucli que l'executa). La manera habitual de fer-ho és enviar una interrupció entre processadors (*inter-processor interrupt*, IPI) a cadascun dels nuclis implicats perquè invalidin l'entrada corresponent, tot i que algunes plataformes ofereixen mecanismes d'invalidació remota que ho estalvien. Aquest procés s'anomena ***TLB shootdown***.
> El cost del *TLB shootdown* és significatiu: cada nucli afectat ha d'interrompre l'execució, processar la interrupció i invalidar l'entrada. En sistemes amb molts nuclis i processos que comparteixen

memòria, el *TLB shootdown* pot convertir-se en un coll d'ampolla de rendiment rellevant, i és una de les raons per les quals la gestió eficient de la memòria virtual en sistemes **multinucli** (*multicore*) és un problema de recerca actiu.

8.6 Protecció i compartició

Una de les funcions més importants de la memòria virtual és permetre que múltiples processos comparteixin la memòria física de manera segura. Això requereix mecanismes que garanteixin que cap procés pugui accedir a l'espai d'adreçament d'un altre de manera no autoritzada.

8.6.1 Protecció entre processos

El mecanisme de traducció d'adreces proporciona protecció entre processos de manera natural: com que cada procés té la seva pròpia taula de pàgines, i l'MMU usa exclusivament la taula del procés actiu per a totes les traduccions, un procés no pot generar cap adreça física que correspongui a una pàgina assignada a un altre procés, tret que el sistema operatiu ho hagi volgut així (Secció 8.6.3). L'aïllament no requereix cap comprovació addicional en cada accés: és conseqüència del contingut de la taula de pàgines.

Perquè aquest mecanisme sigui efectiu, cal que cap procés pugui modificar la seva pròpia taula de pàgines ni el registre de taula de pàgines. Per garantir-ho, les taules de pàgines es troben a la zona de memòria reservada al nucli del sistema operatiu (*kernel*), accessible només quan el processador opera en un mode privilegiat. A RISC-V, els programes d'usuari s'executen en **mode usuari** (U) i el nucli del sistema operatiu en **mode supervisor** (S): en mode U, qualsevol intent d'accedir a la zona del nucli provoca una fallada de pàgina; el sistema operatiu, en tractar-la, detecta que l'adreça no és vàlida per al procés i l'avorta (*segmentation fault*), com en qualsevol accés a una adreça invàlida (Secció 8.4). Qualsevol intent d'escriure el registre de taula de pàgines (`satp`) o d'invalidar el TLB (`sfence.vma`), en canvi, provoca una excepció d'instrucció il · legal. El Tema 9 descriu amb més detall els modes d'execució (Secció 9.2.1) i el mecanisme d'excepcions.

8.6.2 Protecció contra escriptura

A més de l'aïllament entre processos, la memòria virtual permet controlar el tipus d'accés permès sobre cada pàgina individual mitjançant el **bit E** de la PTE i del TLB. Si una instrucció intenta escriure en una pàgina amb E = 0, l'MMU genera una **excepció de protecció** (a RISC-V, una fallada de pàgina d'escriptura, *store page fault*) i el sistema operatiu, en el model d'aquest tema, avorta el procés amb un error de violació de protecció.

Aquest mecanisme té diverses aplicacions pràctiques:

- **Protecció del codi**: les pàgines que contenen instruccions es marquen habitualment com a només de lectura (E = 0), de manera que un error de programació no pugui sobreescriure el codi en execució.
- **Detecció d'errors**: marcar com a només de lectura pàgines que no haurien de ser modificades permet detectar errors de programació que d'altra manera passarien desapercebuts.
- **Compartició segura**: permet compartir pàgines entre processos amb permisos asimètrics (un procés pot escriure-hi i un altre només llegir-hi), com es veurà a la secció següent.

8.6.3 Compartició de pàgines

Hi pot haver situacions en què un procés (per exemple, P1) vulgui permetre a un altre (per exemple, P2) accedir al seu espai d'adreçament. El sistema operatiu, a petició de P1, pot habilitar la compartició simplement escrivint una entrada a la taula de pàgines del procés P2 que assigni una pàgina lògica de P2 a la mateixa pàgina física que P1 vol compartir (Figura 8.10). A petició del procés P1, la pàgina compartida pot tenir permís d'escriptura o no per al procés P2: només cal que el sistema operatiu inicialitzi adequadament el bit E de la pàgina lògica de P2.

> Exemple 8.1: Compartició de pàgines
>
> El procés P1 vol compartir la seva pàgina VPN = `0x00000` (carregada al marc PPN = `0x02`) amb el procés P2 en mode escriptura. El sistema operatiu escriu una entrada a la taula de pàgines de P2 que assigna VPN = `0x00001` al mateix PPN = `0x02`, amb E = 1.
> A partir d'aquell moment, quan P2 accedeix a VPN = `0x00001` i P1 accedeix a VPN = `0x00000`, tots dos llegeixen i escriuen els mateixos bytes de la memòria física: el que un dels dos hi escrigui, l'altre ho llegirà, sense cap còpia pel mig.

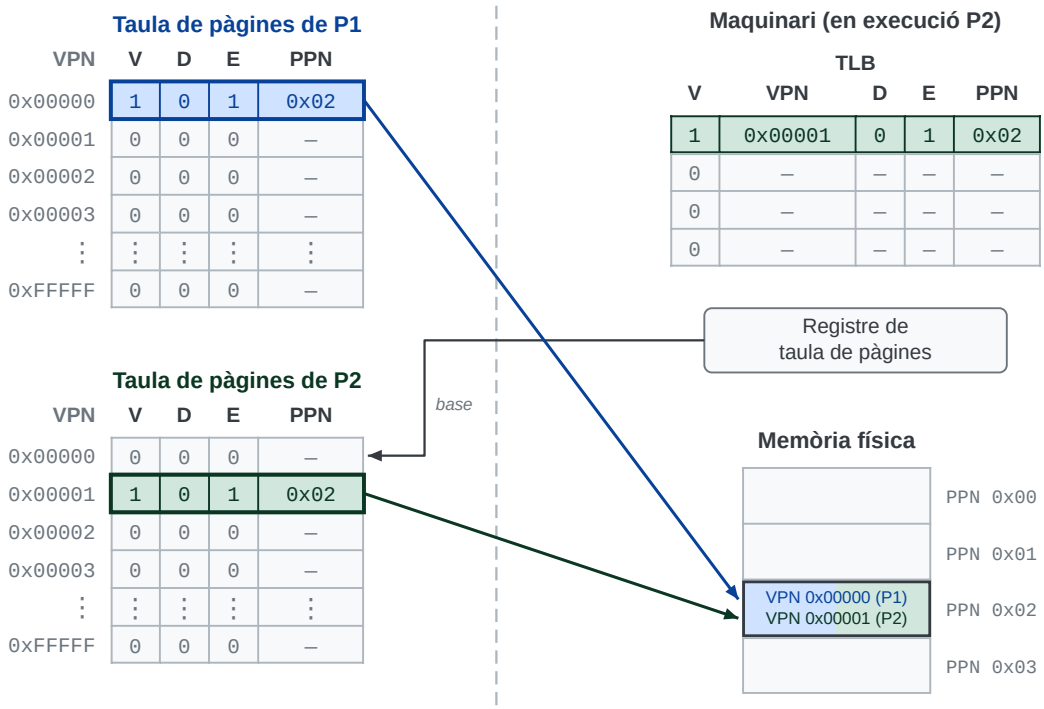


Figura 8.10: Compartició d'una pàgina física entre dos processos (Exemple 8.1): P1 la té assignada a VPN = `0x00000` i P2, a VPN = `0x00001`, totes dues al marc PPN = `0x02` i amb E = 1. La figura mostra el moment en què P2 és en execució: el registre de taula de pàgines apunta a la taula de P2, i el TLB només conté traduccions de la taula de P2.

La compartició de pàgines té dues aplicacions principals:

- **Comunicació entre processos**: dos processos poden intercanviar dades escrivint i llegint una pàgina compartida, sense necessitat de còpies.
- **Biblioteques compartides**: si múltiples processos usen simultàniament el codi d'una mateixa biblioteca, el sistema operatiu pot mantenir una sola còpia de les pàgines de codi a la memòria física i assignar-les (en mode només de lectura, E = 0) a l'espai lògic de cada procés, estalviant memòria física de manera significativa.

8.7 Integració del TLB i la memòria cau

En un sistema amb memòria virtual, cada accés a memòria implica dos passos: traduir l'adreça lògica a física (amb el TLB) i accedir a la dada (a la memòria cau). La manera com s'organitzen aquests dos passos determina el temps d'accés efectiu i la complexitat del disseny.

8.7.1 Memòria cau indexada físicament

L'esquema més senzill és la **memòria cau indexada i etiquetada físicament** (*physically indexed physically tagged*, PIPT): primer es tradueix l'adreça lògica amb el TLB i, un cop obtinguda l'adreça física, s'accedeix a la memòria cau amb aquesta adreça (Figura 8.11).

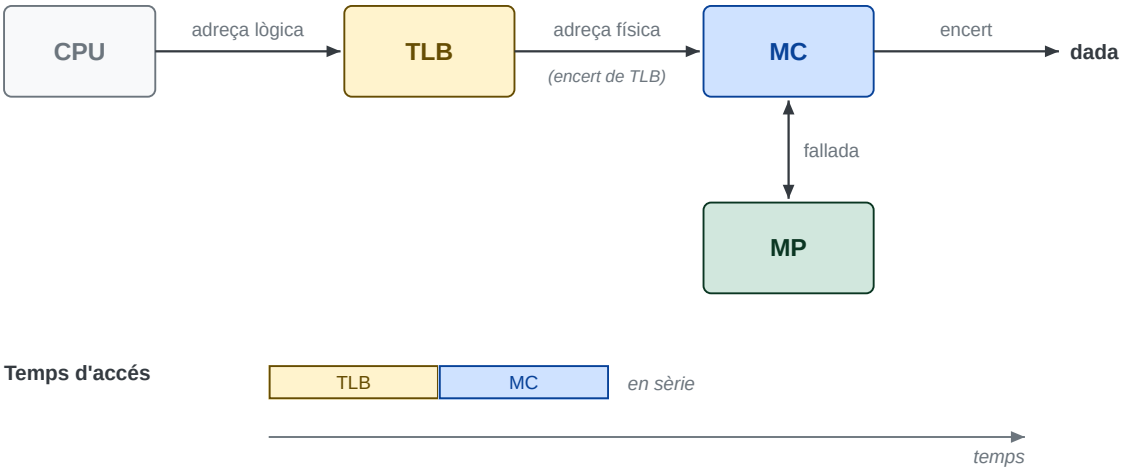


Figura 8.11: Memòria cau indexada físicament (PIPT): la traducció i l'accés a la memòria cau es fan en sèrie. A sota, el temps d'accés: el del TLB i el de la memòria cau, un darrere l'altre.

L'avantatge és la simplicitat: no hi ha ambigüitat possible, ja que cada adreça física identifica unívocament una dada. L'inconvenient és el **temps d'accés**: la memòria cau no pot començar a operar fins que el TLB hagi acabat la traducció, de manera que els dos accessos es fan en sèrie.

8.7.2 Memòria cau indexada i etiquetada virtualment

Una **memòria cau indexada i etiquetada virtualment** (*virtually indexed virtually tagged*, VIVT) redueix la latència perquè usa l'adreça lògica directament, sense esperar la traducció. Si hi ha encert, la dada s'obté sense necessitat de la traducció; només cal el TLB en cas de fallada, per anar a buscar el bloc a la memòria principal amb l'adreça física.

El problema és l'**aliàsing** (*aliasing*): si dos processos (o dues pàgines lògiques d'un mateix procés) comparteixen una mateixa pàgina física (Secció 8.6.3), el mateix bloc pot ocupar dues línies diferents de la memòria cau, una per a cada adreça lògica. Una escriptura feta a través d'una adreça lògica pot no ser visible quan es llegeix a través de l'altra, cosa que trenca la coherència.

L'altre problema són els **homònims**: la mateixa adreça lògica de dos processos correspon a dades físiques diferents. Per això, en cada canvi de context, cal invalidar la memòria cau sencera o afegir a cada etiqueta l'identificador del procés, com fa l'ASID amb el TLB (Secció 8.5.4).

8.7.3 VIPT: la solució habitual

La solució més habitual als processadors actuals, per a la memòria cau de primer nivell, és la **memòria cau indexada virtualment i etiquetada físicament** (*virtually indexed physically tagged*, VIPT). La idea clau és que l'índex per accedir a la memòria cau i la traducció del TLB es poden fer en **paral · lel** (Figura 8.12):

- L'**índex** de la memòria cau s'extreu de l'adreça lògica. Si només fa servir bits del desplaçament de pàgina (els t bits de menys pes; és la condició de més avall), coincideix exactament amb el de l'adreça física, perquè el desplaçament no es tradueix.
- Mentre la memòria cau llegeix les etiquetes i els blocs de dades corresponents a aquest índex, el TLB tradueix el VPN i obté el PPN.
- Un cop acabats els dos accessos en paral · lel, es compara l'etiqueta física de l'adreça —el PPN obtingut del TLB, més els bits del desplaçament de pàgina que l'índex i el desplaçament de bloc no fan servir, si n'hi ha— amb les etiquetes llegides de la memòria cau per determinar si hi ha encert.

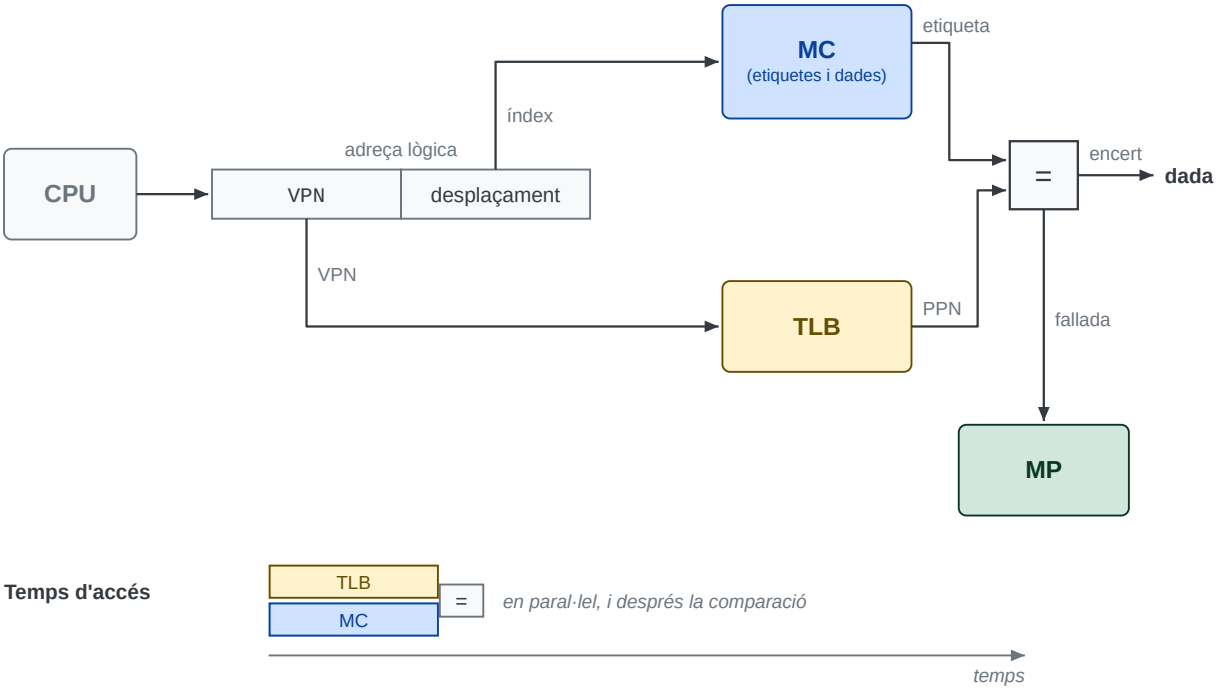


Figura 8.12: Memòria cau VIPT: la indexació de la memòria cau i la traducció del TLB es fan en paral · lel, i, com que l'índex només fa servir bits del desplaçament, no hi ha aliàsing. A sota, el temps d'accés, a la mateixa escala que a la Figura 8.11.

La condició perquè VIPT funcioni sense aliàsing és que l'índex de la memòria cau usi exclusivament bits del desplaçament (és a dir, bits que no canvien amb la traducció). En una memòria cau amb $N_C = 2^{n_c}$ conjunts i blocs de $B = 2^b$ bytes (Essencial 7.1), l'índex ocupa els n_c bits que hi ha per sobre dels b del desplaçament dins del bloc. Perquè tots siguin del desplaçament de pàgina (els $t = \log_2 T$ bits de menys pes), cal que $n_c + b \leq t$, és a dir:

$$N_C \cdot B \leq T$$

on T és la mida de pàgina. El producte $N_C \cdot B$ és la mida d'una via, la capacitat de la memòria cau dividida pel grau d'associativitat N: cap via no pot ser més gran que una pàgina. Per exemple,

amb pàgines de 4 KiB i 4 vies, la memòria cau L1 pot tenir fins a 16 KiB sense aliàsing. És una condició suficient per evitar l'aliàsing en una memòria cau VIPT, no una equivalència amb la PIPT en tots els aspectes, i els dissenys de L1 l'acostumen a respectar: una L1 més gran obliga a augmentar l'associativitat en la mateixa proporció.

Essencial 8.5: Comparació dels esquemes d'integració TLB–memòria cau

Esquema	Indexació	Etiqueta	Avantatge	Inconvenient
PIPT	Física	Física	Simple, sense aliàsing	TLB i MC en sèrie
VIVT	Virtual	Virtual	Ràpid (sense TLB en els encerts)	Aliàsing i homònims
VIPT	Virtual	Física	Ràpid i sense aliàsing	Restricció de mida

VIPT és l'esquema habitual als processadors actuals per a la memòria cau L1. Les memòries cau L2 i L3, menys crítiques en latència, usen habitualment PIPT. Cap dels tres esquemes no és un requisit de l'ISA: l'organització de les memòries cau és una decisió de cada implementació.

8.8 Exemple pràctic

Aquesta secció recorre un exemple complet: una seqüència de cinc accessos a memòria en un sistema amb TLB, amb la traducció de cada adreça, les fallades de TLB i de pàgina, el reemplaçament i el bit D, seguint el flux de la Figura 8.9.

Exemple 8.2: Traducció amb TLB

Considerem un sistema amb adreces lògiques de 32 bits, pàgines de 4 KiB ($t = 12$), una memòria física de 16 KiB (4 marcs de pàgina, adreces físiques de 14 bits) i un TLB de 4 entrades, completament associatiu, amb reemplaçament LRU, tant per al TLB com per a les pàgines. L'estat inicial és:

Taula 8.1: TLB (estat inicial)

V	VPN	D	E	PPN
1	`0x00003`	1	1	`0x00`
1	`0x00000`	0	1	`0x01`
1	`0x00002`	0	1	`0x02`
0	—	—	—	—

La taula de pàgines (entrades rellevants) és:

Taula 8.2: Taula de pàgines (entrades rellevants)

VPN	V	D	E	PPN
`0x00000`	1	0	1	`0x01`
`0x00001`	0	0	1	—
`0x00002`	1	0	1	`0x02`
`0x00003`	1	1	1	`0x00`
`0x00004`	0	0	1	—
`0x00005`	0	0	1	—

Sabem que només tres dels quatre marcs de pàgina estan ocupats (PPN = `0x03` és lliure), i que els tres darrers accessos han estat a VPN = `0x00003`, VPN = `0x00000` i VPN = `0x00002`, en aquest ordre (de més antic a més recent). Les tres pàgines residents tenen entrada al TLB, de manera que l'ordre LRU del TLB i el de les pàgines coincideixen.

Accés 1: escriptura a `0x00002A0B` → VPN = `0x00002`.

Encert de TLB. Com que és una escriptura amb E = 1 i D = 0, s'aplica la política d'escriptura immediata descrita a la Secció 8.5.5: es posa D = 1 al TLB i a la PTE. PPN = `0x02`. Adreça física (14 bits): `0x2A0B`. VPN 2 passa a ser la pàgina més recent (ja ho era).

Accés 2: lectura a `0x00001F21` → VPN = `0x00001`.

Fallada de TLB. L'MMU llegeix la PTE de VPN 1: V = 0, fallada de pàgina; no es copia res al TLB. El sistema operatiu troba el marc lliure (PPN = `0x03`), hi carrega la pàgina des del disc i actualitza la PTE (V = 1, D = 0, PPN = `0x03`). En reexecutar la lectura, torna a fallar el TLB, però ara la PTE té V = 1 i l'MMU la copia a l'entrada lliure. PPN = `0x03`. Adreça física: `0x3F21`. Ordre LRU: VPN 3, 0, 2, 1.

Accés 3: lectura a `0x0000420C` → VPN = `0x00004`.

Fallada de TLB i, en llegir la PTE, fallada de pàgina. Cap marc lliure: el sistema operatiu tria la víctima LRU, VPN 3 (al marc `0x00`), que té D = 1 i que cal escriure primer al disc. Posa V = 0 a la PTE de VPN 3 i n'**invalida l'entrada del TLB**, que queda lliure. Carrega VPN 4 al marc `0x00` i actualitza la seva PTE (V = 1, D = 0, PPN = `0x00`). En reexecutar la lectura, l'MMU copia la PTE de VPN 4 a l'entrada que acaba de quedar lliure. PPN = `0x00`. Adreça física: `0x020C`. Ordre LRU: VPN 0, 2, 1, 4.

Accés 4: lectura a `0x00003001` → VPN = `0x00003`.

Fallada de TLB (l'entrada de VPN 3 s'ha invalidat a l'accés 3) i fallada de pàgina. La víctima LRU és VPN 0 (al marc `0x01`), amb D = 0: no cal escriure-la al disc. V = 0 a la seva PTE i entrada del TLB invalidada. Es carrega VPN 3 al marc `0x01` (PTE: V = 1, D = 0, PPN = `0x01`; en tornar del disc la pàgina ja no està modificada) i, en reexecutar, s'instal · la a l'entrada lliure. PPN = `0x01`. Adreça física: `0x1001`. Ordre LRU: VPN 2, 1, 4, 3.

Accés 5: lectura a `0x00005120` → VPN = `0x00005`.

Fallada de TLB i fallada de pàgina. La víctima LRU és VPN 2 (al marc `0x02`), que té D = 1 des de l'accés 1: s'escriu al disc. V = 0 a la seva PTE i entrada del TLB invalidada. Es carrega VPN 5 al marc `0x02` (PTE: V = 1, D = 0, PPN = `0x02`) i s'instal · la al TLB. PPN = `0x02`. Adreça física: `0x2120`. Ordre LRU: VPN 1, 4, 3, 5.

Taula 8.3: Resum dels cinc accessos

	Adreça lògica	VPN	Fallada TLB	Fallada pàg.	Pàg. víctima	Escr. disc	PPN	Adreça física
E	`0x00002A0B`	`0x00002`	no	no	—	no	`0x02`	`0x2A0B`
L	`0x00001F21`	`0x00001`	sí	sí	— (marc lliure)	no	`0x03`	`0x3F21`
L	`0x0000420C`	`0x00004`	sí	sí	`0x00003`	sí (VPN 3)	`0x00`	`0x020C`
L	`0x00003001`	`0x00003`	sí	sí	`0x00000`	no	`0x01`	`0x1001`
L	`0x00005120`	`0x00005`	sí	sí	`0x00002`	sí (VPN 2)	`0x02`	`0x2120`

A la columna «Fallada TLB» es compta una fallada per accés: quan la instrucció es torna a executar després d'una fallada de pàgina, la segona consulta del TLB no compta com una fallada nova.
En acabar, la memòria física conté VPN 4 (`0x00`), VPN 3 (`0x01`), VPN 5 (`0x02`) i VPN 1 (`0x03`), totes amb D = 0, i el TLB té exactament aquestes quatre traduccions. Fixa't que, amb tants marcs com entrades de TLB, el reemplaçament LRU del TLB no arriba a actuar mai: mentre hi ha marcs lliures, també hi ha entrades lliures. Quan el TLB és ple, hi són totes les pàgines residents; per tant, una fallada de TLB és sempre una fallada de pàgina, i en invalidar l'entrada de la víctima se n'allibera una. El reemplaçament del TLB només entra en joc quan hi ha més pàgines residents que entrades, com a Exercici 8.5. La Figura 8.13 mostra l'estat del TLB i de la memòria física després de cada accés.

Estat inicial

Ordre LRU (de més antic a més recent): VPN 3, VPN 0, VPN 2

TLB

V	VPN	D	E	PPN
1	0x00003	1	1	0x00
1	0x00000	0	1	0x01
1	0x00002	0	1	0x02
0	*lliure*	–	–	–

PPN	Memòria física
0x00	VPN 3 · D = 1
0x01	VPN 0
0x02	VPN 2
0x03	*lliure*

Accés 1: Escriptura a 0x00002A0B → VPN 0x00002

Encert de TLB. Primera escriptura: D = 1 al TLB i a la PTE.
PPN 0x02 → adreça física 0x2A0B

Ordre LRU (de més antic a més recent): VPN 3, VPN 0, VPN 2

V	VPN	D	E	PPN
1	0x00003	1	1	0x00
1	0x00000	0	1	0x01
1	0x00002	1	1	0x02
0	*lliure*	–	–	–

PPN	Memòria física
0x00	VPN 3 · D = 1
0x01	VPN 0
0x02	VPN 2 · D = 1
0x03	*lliure*

Accés 2: Lectura a 0x00001F21 → VPN 0x00001

Fallada de TLB i de pàgina: es carrega la pàgina al marc lliure.
PPN 0x03 → adreça física 0x3F21

Ordre LRU (de més antic a més recent): VPN 3, VPN 0, VPN 2, VPN 1

V	VPN	D	E	PPN
1	0x00003	1	1	0x00
1	0x00000	0	1	0x01
1	0x00002	1	1	0x02
1	0x00001	0	1	0x03

PPN	Memòria física
0x00	VPN 3 · D = 1
0x01	VPN 0
0x02	VPN 2 · D = 1
0x03	VPN 1

Accés 3: Lectura a 0x0000420C → VPN 0x00004

Fallada de TLB i de pàgina: víctima VPN 3, escrita al disc (D = 1); se n'invalida l'entrada del TLB.
PPN 0x00 → adreça física 0x020C

Ordre LRU (de més antic a més recent): VPN 0, VPN 2, VPN 1, VPN 4

V	VPN	D	E	PPN
1	0x00004	0	1	0x00
1	0x00000	0	1	0x01
1	0x00002	1	1	0x02
1	0x00001	0	1	0x03

PPN	Memòria física
0x00	VPN 4
0x01	VPN 0
0x02	VPN 2 · D = 1
0x03	VPN 1

Accés 4: Lectura a 0x00003001 → VPN 0x00003

Fallada de TLB i de pàgina: víctima VPN 0 (D = 0, no cal escriure-la); se n'invalida l'entrada del TLB.
PPN 0x01 → adreça física 0x1001

Ordre LRU (de més antic a més recent): VPN 2, VPN 1, VPN 4, VPN 3

V	VPN	D	E	PPN
1	0x00004	0	1	0x00
1	0x00003	0	1	0x01
1	0x00002	1	1	0x02
1	0x00001	0	1	0x03

PPN	Memòria física
0x00	VPN 4
0x01	VPN 3
0x02	VPN 2 · D = 1
0x03	VPN 1

Accés 5: Lectura a 0x00005120 → VPN 0x00005

Fallada de TLB i de pàgina: víctima VPN 2, escrita al disc (D = 1); se n'invalida l'entrada del TLB.
PPN 0x02 → adreça física 0x2120

Ordre LRU (de més antic a més recent): VPN 1, VPN 4, VPN 3, VPN 5

V	VPN	D	E	PPN
1	0x00004	0	1	0x00
1	0x00003	0	1	0x01
1	0x00005	0	1	0x02
1	0x00001	0	1	0x03

PPN	Memòria física
0x00	VPN 4
0x01	VPN 3
0x02	VPN 5
0x03	VPN 1

Figura 8.13: Traça dels cinc accessos de Exemple 8.2: l'estat inicial i, per a cada accés, el TLB i la memòria física després de resoldre'l, amb les entrades i els marcs que han canviat amb vora gruixuda. A la versió en *HyperText Markup Language* (HTML), el navegador de passos mostra també la taula de pàgines després de cada accés.

Tema 9: Excepcions i interrupcions

9.1 Introducció

9.1.1 Necessitat

Durant l'execució normal d'un programa, el processador segueix un flux d'instruccions determinat per les instruccions de salt explícites (`beq`, `jal`, etc.). Les **excepcions** i les **interrupcions** són un mecanisme que permet alterar aquest flux de manera imprevista, quan es produeix un **esdeveniment** (*event*), sense que cap instrucció de salt ho hagi causat. La necessitat d'aquest mecanisme sorgeix de dues fonts ben diferenciades.

D'una banda, durant l'execució d'una instrucció poden aparèixer condicions errònies o especials que el processador no pot resoldre per si sol: un accés a una adreça de memòria no alineada o una instrucció amb un codi d'operació inexistent. En tots dos casos, cal interrompre el programa i transferir el control a un codi del sistema operatiu (SO) que en faci el tractament adequat.

De l'altra, els dispositius d'**entrada/sortida** (E/S) necessiten notificar al processador que han completat una operació o que requereixen atenció, perquè esperar activament que el dispositiu acabi (consultant-ne l'estat repetidament) desaprofita el processador. La solució és que el dispositiu, quan estigui llest, avisi el processador, i que el processador pugui interrompre temporalment el programa en execució per atendre'l.

En tots dos casos, el mecanisme és el mateix: el processador interromp l'execució del programa, transfereix el control a una **rutina de servei d'excepcions** (RSE) de l'SO, i un cop feta la gestió, pot reprendre el programa interromput.

Essencial 9.1

- **Excepció** (***exception***): esdeveniment causat per l'execució d'una instrucció (intern al processador).
- **Interrupció** (***interrupt***): esdeveniment causat per un dispositiu extern.

Tots dos usen el mateix mecanisme de maquinari, però difereixen en l'origen, el moment de detecció i, sovint, en el tractament.

9.1.2 Tipologia

La taula següent classifica els principals tipus d'excepcions i interrupcions, indicant-ne l'origen, el moment en què el processador les detecta i l'acció que pren en acabar l'RSE.

Taula 9.1: Tipologia d'excepcions i interrupcions

Tipus	Causa	Detecció	Acció en acabar l'RSE	Excep. / Inter.
Accés no alineat	Adreça no múltiple de la mida de dada	Durant la instrucció	Avorta[1]	Excepció
Instrucció il · legal	Codi d'operació inexistent	Durant la instrucció	Avorta[1]	Excepció
Fallada de pàgina	Pàgina absent de la memòria física	Durant la instrucció	Reexecuta la instrucció causant	Excepció
Crida al sistema (`ecall`)	Instrucció `ecall` explícita	Durant la instrucció	Continua a la instrucció següent	Excepció
Interrupció d'E/S	Senyal extern d'un dispositiu	En acabar la instrucció en curs	Continua a la instrucció següent	Interrupció

[1] En el cas general, l'RSE avorta el programa i emet un missatge d'error. Tècnicament, una RSE suficientment sofisticada podria recuperar-se d'algunes d'aquestes situacions, però això queda fora de l'abast d'EC.

Cal destacar dues diferències clau entre excepcions i interrupcions que la taula reflecteix:

- **Moment de detecció**:
  - Les **excepcions** es detecten **durant** l'execució de la instrucció causant, que s'avorta sense escriure cap resultat ni incrementar el PC.
  - Les **interrupcions** es detecten **en acabar** la instrucció en curs, quan el PC ja s'ha incrementat.
- **Acció en acabar l'RSE**:
  - En les **excepcions**, depèn de l'excepció:
    - Si **es pot resoldre** (fallada de pàgina), cal **reexecutar** la instrucció causant.
    - Si **no altera l'estat del programa** (crida al sistema), cal **continuar** a la instrucció següent.
    - Si **és irrecuperable**, el programa s'*avorta*.
  - En les **interrupcions**, el programa sempre es reprèn a la instrucció següent.

9.2 Suport de maquinari: registres de control i estat

9.2.1 El model de privilegis de RISC-V

RISC-V defineix tres **modes de privilegi**, que determinen quines **instruccions** pot executar el processador i a quines **adreces de memòria** pot accedir.

RISC-V 9.1: RV32I ISA — Modes de privilegi

Mode	Privilegi	Nom	Ús habitual	Obligatori
M	Màxim	*Machine* (màquina)	**Microprogramari** (*firmware*), arrencada, gestió d'excepcions de baix nivell	Sí
S	Mitjà	*Supervisor*	Sistema operatiu (Linux, etc.)	No
U	Mínim	*User* (usuari)	Programes d'usuari	No

EC 9.1

A **EC** s'estudia el mecanisme d'excepcions en **mode M**, que és el més directe i el que implementen tots els processadors RISC-V. El mode S s'introdueix a Secció 9.2.8 i s'aplica al tractament de fallades de pàgina a Secció 9.7.3.

El mode en què opera el processador en cada moment es controla mitjançant uns registres especials anomenats **registres de control i estat** (*control and status registers*, CSR). A aquests registres només s'hi pot accedir amb instruccions específiques.

L'extensió **Zicsr** de RISC-V defineix les instruccions d'accés als CSR. Encara que en versions antigues de l'especificació aquestes instruccions formaven part de l'ISA base, a partir de la versió ratificada del 2019 (document 20190608, amb RV32I 2.1) es van segregar en una extensió independent per permetre implementacions minimalistes sense suport CSR. En la pràctica, qualsevol processador RISC-V que implementi el mecanisme d'excepcions i interrupcions requereix Zicsr.

RISC-V 9.2: Zicsr ISA — Instruccions d'accés als CSR

Mnemònic	Operands	Operació	Nom	Tipus
`csrrw`	`rd, csr, rs1`	$rd \leftarrow csr;\ csr \leftarrow rs1$	*CSR read/write*	I
`csrrs`	`rd, csr, rs1`	$rd \leftarrow csr;\ csr \leftarrow csr \vee rs1$	*CSR read and set bits*	I
`csrrc`	`rd, csr, rs1`	$rd \leftarrow csr;\ csr \leftarrow csr \wedge \overline{rs1}$	*CSR read and clear bits*	I

Les pseudoinstruccions `csrr` i `csrw` (RISC-V 9.3) són les formes habituals per llegir i escriure un CSR sense modificar-ne l'altre operand. Les instruccions `csrrs` i `csrrc` permeten activar o desactivar bits individuals sense alterar la resta.

RISC-V 9.3: Pseudoinstruccions — Accés als CSR: `csrr` i `csrw`

Pseudoinstrucció	Operació	Expansió
`csrr rd, csr`	$rd \leftarrow csr$	`csrrs rd, csr, zero`
`csrw csr, rs`	$csr \leftarrow rs$	`csrrw zero, csr, rs`

RARS 1.6 espera l'ordre invers dels operands de `csrw`, `csrw rs, csr` (`csrw t1, fcsr`), i no té els CSR de mode M, com `mepc`.

L'accés als CSR està **protegit pel nivell de privilegi**: cada CSR té un nivell mínim, codificat als

bits 9:8 de la seva adreça de 12 bits, i accedir des de mode U a un CSR de mode M o S genera una excepció d'instrucció il · legal. Alguns CSR són de mode U, com `fcsr` (T5) o els comptadors `cycle` i `instret`.

Aprofundiment 9.1: Variants immediates de les instruccions CSR

Les instruccions `csrrwi`, `csrrsi` i `csrrci` operen igual que `csrrw`, `csrrs` i `csrrc` respectivament, però accepten un immediat de 5 bits sense signe (`uimm5`) en lloc del registre `rs1`. Són útils per activar o desactivar bits individuals d'un CSR sense necessitat de carregar un valor en un registre prèviament.

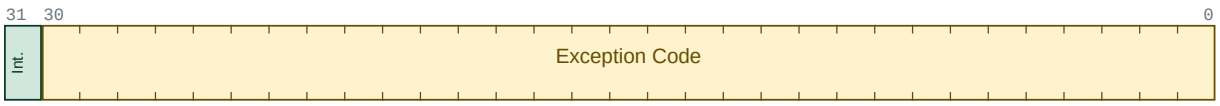
Zicsr ISA — Instruccions d'accés als CSR amb immediat

Mnemònic	Operands	Operació	Nom	Tipus
`csrrwi`	`rd, csr, uimm5`	$rd \leftarrow csr;\ csr \leftarrow uimm5$	*CSR read/write immediate*	I
`csrrsi`	`rd, csr, uimm5`	$rd \leftarrow csr;\ csr \leftarrow csr \vee uimm5$	*CSR read and set bits immediate*	I
`csrrci`	`rd, csr, uimm5`	$rd \leftarrow csr;\ csr \leftarrow csr \wedge \overline{uimm5}$	*CSR read and clear bits immediate*	I

9.2.2 `mcause` — Codi de causa

Quan es produeix una excepció o interrupció, el maquinari escriu automàticament al registre **`mcause`** un codi que n'identifica la causa. El bit de més pes (bit 31) indica si l'origen és una interrupció (bit 31 = 1) o una excepció (bit 31 = 0). Els 31 bits restants contenen el codi de causa.

RISC-V 9.4: RV32I ISA — `mcause` — Valors més rellevants



A la figura, *Int.* és el bit 31 (*Interrupt*).

Bit 31	Codi	Causa
0	0	Instrucció mal alineada
0	1	Error d'accés en cerca d'instrucció
0	2	Instrucció il · legal
0	3	Punt de parada (`ebreak`)
0	4	Lectura de dada mal alineada
0	5	Error d'accés en lectura de dada
0	6	Escriptura de dada mal alineada
0	7	Error d'accés en escriptura de dada
0	8	Crida al sistema des del mode U (`ecall`)

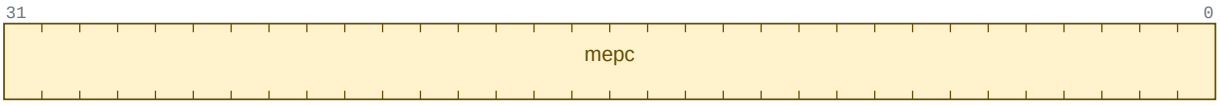
0	9	Crida al sistema des del mode S (`ecall`)
0	11	Crida al sistema des del mode M (`ecall`)
0	12	Fallada de pàgina en cerca d'instrucció
0	13	Fallada de pàgina en lectura
0	15	Fallada de pàgina en escriptura o en una *atomic memory operation* (AMO)
1	3	Interrupció de programari en mode M
1	7	Interrupció de temporitzador en mode M
1	11	Interrupció externa (E/S) en mode M

9.2.3 `mepc` — Adreça de retorn

El registre **`mepc`** (*Machine Exception Program Counter*) guarda l'adreça on l'RSE ha de retornar el control en acabar: la de la instrucció causant o la de la següent, segons el tipus d'esdeveniment:

- **Excepció**: `mepc` apunta a la instrucció causant (la que s'ha interromput sense finalitzar). En els casos en què cal continuar a la instrucció següent (per exemple, `ecall`), la mateixa RSE incrementa `mepc` en 4 abans de retornar.
- **Interrupció**: `mepc` apunta a la instrucció que *hauria* executat el processador a continuació (la instrucció causant ja ha finalitzat i el PC ja s'ha incrementat).

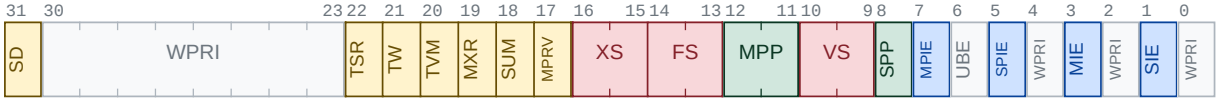
RISC-V 9.5: RV32I ISA — `mepc`



9.2.4 `mstatus` — Estat global

El registre **`mstatus`** conté els bits d'estat global del processador. Per al mecanisme d'excepcions, els camps rellevants són:

RISC-V 9.6: RV32I ISA — `mstatus`



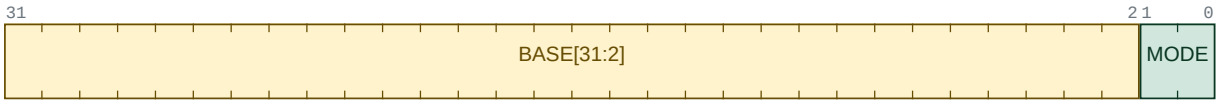
- **`MIE`** (bit 3, *Machine Interrupt Enable*): permís global d'interrupcions. Si `MIE=1`, les interrupcions estan habilitades. Si `MIE=0`, totes les interrupcions queden inhibides. Les excepcions

no es poden inhibir.
- **`MPIE`** (bit 7, *Machine Previous Interrupt Enable*): guarda el valor de `MIE` en el moment en què s'ha produït l'excepció o interrupció. Permet restaurar-lo en retornar.
- **`MPP`** (bits 12:11, *Machine Previous Privilege*): guarda el mode de privilegi anterior (M, S o U) per poder-lo restaurar en retornar amb `mret`.

9.2.5 `mtvec` — Vector d'excepcions

El registre **`mtvec`** (*Machine Trap-Vector Base-Address Register*) conté l'adreça on el processador salta quan es produeix una excepció o interrupció. Té dos camps:

RISC-V 9.7: RV32I ISA — `mtvec`



- **`BASE`** (bits 31:2): adreça base de la rutina o taula de vectors, alineada a 4 bytes.
- **`MODE`** (bits 1:0): mode de funcionament.

RISC-V defineix dos modes:

`MODE`	Nom	Comportament
0	Directe	Totes les excepcions i interrupcions salten a `BASE`
1	Vectoritzat	Les interrupcions salten a `BASE + 4 × causa`; les excepcions sempre a `BASE`

En el **mode directe**, l'RSE és responsable de llegir `mcause` i derivar el control a la rutina específica de cada causa. En el **mode vectoritzat**, el maquinari calcula directament l'adreça de la rutina per a cada tipus d'interrupció, cosa que estalvia el temps de llegir `mcause` i saltar a la rutina corresponent.

EC 9.2

A **EC** s'usa sempre el **mode directe** (`MODE=0`): totes les excepcions i interrupcions salten a l'adreça `BASE`, i és l'RSE qui consulta `mcause` per determinar-ne la causa.

9.2.6 `mtval` — Informació addicional de l'excepció

El registre **`mtval`** (*Machine Trap Value*) conté informació addicional sobre la causa de l'excepció, per ajudar l'RSE en el tractament. El valor que hi escriu el maquinari depèn del tipus d'excepció:

- **Accés no alineat o fallada de pàgina**: `mtval` conté l'adreça virtual que ha causat l'excepció.
- **Instrucció il · legal**: `mtval` pot contenir la instrucció que ha causat l'excepció, o zero (l'especificació ho deixa a la implementació).
- **Interrupció o altres excepcions**: `mtval` s'escriu a zero.

9.2.7 `mip` / `mie` — Peticions i màscares d'interrupció

Els registres **`mip`** (*Machine Interrupt Pending*) i **`mie`** (*Machine Interrupt Enable*) gestionen les interrupcions bit a bit. Cada bit correspon a un tipus d'interrupció:

RISC-V 9.8: RV32I ISA — `mip` i `mie`



Bit	Nom `mip`	Nom `mie`	Tipus d'interrupció
11	`MEIP`	`MEIE`	Externa (E/S) en mode M
9	`SEIP`	`SEIE`	Externa (E/S) en mode S
7	`MTIP`	`MTIE`	Temporitzador en mode M
5	`STIP`	`STIE`	Temporitzador en mode S
3	`MSIP`	`MSIE`	Programari en mode M
1	`SSIP`	`SSIE`	Programari en mode S

- **`mip`**: cada bit s'activa quan el dispositiu corresponent ha enviat una petició d'interrupció i la manté activa. Els bits de mode M (`MEIP`, `MTIP`, `MSIP`) són de **només lectura** per al programari (els actualitza el maquinari).
- **`mie`**: cada bit és una màscara independent que habilita o deshabilita les interrupcions d'un tipus concret. És de **lectura/escriptura**.

Perquè una interrupció sigui atesa cal que es compleixin simultàniament tres condicions: `MIE=1` (permís global), el bit corresponent de `mie` val 1 (permís específic) i el bit corresponent de `mip` val 1 (petició pendent).

Essencial 9.2

`mip` i `mie` treballen conjuntament: `mip` indica *quines* peticions han arribat, i `mie` indica *quines* d'elles estan habilitades. El permís global `MIE` de `mstatus` les inhibeix totes alhora, independentment de `mie`.

9.2.8 Els registres equivalents en mode S

Per al mode S, RISC-V defineix un conjunt de registres CSR paral · lel al de mode M, amb la lletra inicial `s` en lloc de `m`. La funció de cada registre és anàloga:

RISC-V 9.9: RV32I ISA — Registres CSR de mode M i mode S

Registre mode M	Registre mode S	Funció
`mcause`	`scause`	Codi de causa de l'excepció o interrupció

`mepc`	`sepc`	Adreça de retorn
`mstatus`	`sstatus`	Estat global (subconjunt dels bits de `mstatus`)
`mtvec`	`stvec`	Adreça de la rutina de servei
`mip`	`sip`	Peticions d'interrupció pendents
`mie`	`sie`	Màscares d'interrupció

Quan el processador opera en mode S (per exemple, executant el nucli de Linux), les excepcions i interrupcions delegades pel mode M es gestionen amb els registres `s*`. La delegació s'especifica als registres `medeleg` (excepcions) i `mideleg` (interrupcions), que el microprogramari configura durant l'arrencada. L'equivalent de `mret` en mode S és la instrucció **`sret`**, amb la mateixa semàntica sobre els camps `S*` de `sstatus` i sobre `sepc`.

9.3 Flux de maquinari en detectar una excepció o interrupció

9.3.1 Accions automàtiques del processador

Quan el processador detecta una excepció o una petició d'interrupció, executa automàticament, per maquinari, la seqüència d'accions següent:

1. **Desar el PC a `mepc`**: el processador escriu el valor actual del PC al registre `mepc`, per poder reprendre el programa interromput en acabar l'RSE.
2. **Escriure la causa a `mcause`**: s'escriu el codi de la causa (bit 31 + codi) al registre `mcause`, per tal que l'RSE pugui identificar l'origen de l'excepció o interrupció.
3. **Desar i desactivar el permís d'interrupcions**: el valor de `MIE` es copia a `MPIE` (per poder-lo restaurar més tard), i `MIE` es posa a 0. A partir d'aquest moment, totes les interrupcions queden inhibides mentre s'executa l'RSE.
4. **Desar el mode de privilegi**: el mode actual s'escriu a `MPP` de `mstatus`, i el processador passa a mode M.
5. **Saltar a l'RSE**: el PC s'escriu amb l'adreça continguda a `mtvec` (en mode directe, `BASE`; en mode vectoritzat, `BASE + 4 × causa` per a interrupcions).

> Essencial 9.3
>
> Les accions 1–5 les executa el **maquinari** de forma atòmica, sense intervenció del programari. L'RSE no comença a executar-se fins que totes s'han completat.

9.3.2 Diferències entre excepció i interrupció

Tot i que el flux de maquinari és el mateix, hi ha dues diferències importants segons el tipus d'esdeveniment:

- **Moment de detecció**:

  - En una **excepció**, la instrucció causant s'**avorta** a mitja execució: no escriu cap resultat, no accedeix a memòria i el PC no s'incrementa. El valor guardat a `mepc` és l'adreça d'aquesta instrucció (vegeu Secció 9.2.3).
  - En una **interrupció**, el processador no la comprova fins que la instrucció en curs ha **finalitzat** completament i el PC ja s'ha incrementat. El valor guardat a `mepc` és l'adreça de la instrucció que *hauria* executat a continuació (vegeu Secció 9.2.3).

- **Escriptura a `mtval`**: en algunes excepcions, el maquinari escriu informació addicional al registre `mtval` per ajudar l'RSE en el tractament (vegeu Secció 9.2.6). En les interrupcions, `mtval` s'escriu a zero.

⚠ Aprofundiment 9.2: Excepcions en un *pipeline*

Un processador amb ***pipeline*** divideix l'execució de cada instrucció en etapes, com les del cicle d'instrucció (Secció 1.3.3), i en treballa diverses alhora, cadascuna en una etapa diferent, com en una cadena de muntatge: mentre una instrucció s'executa, la següent es descodifica i la de després es captura. Així pot acabar una instrucció a cada cicle, encara que cadascuna en necessiti diversos. Quan es detecta una excepció, doncs, hi ha instruccions a mig fer a les etapes anteriors i posteriors, i cal garantir que:

- La instrucció causant i totes les posteriors en el *pipeline* s'anul · len (*flush*), sense escriure cap resultat.
- Totes les instruccions anteriors a la causant en l'ordre del programa s'han **completat**.

Aquesta propietat s'anomena **excepcions precises** (*precise exceptions*) i és un requisit de l'arquitectura RISC-V. Garantir-la en *pipelines* profunds és un dels reptes de disseny més importants dels processadors moderns.
Les interrupcions, en canvi, es comproven sempre entre instruccions (mai a mig d'una), cosa que simplifica el tractament en el *pipeline*.

9.4 La rutina de servei d'excepcions

9.4.1 Responsabilitats

Un cop el maquinari ha completat les accions de Secció 9.3.1, el processador comença a executar l'RSE. Aquesta rutina, que forma part de l'SO, té les responsabilitats següents, en ordre:

1. **Desar l'estat del programa interromput**: l'RSE ha de desar a memòria tots els registres generals (`x1`–`x31`), per poder-los restaurar en acabar. A diferència d'una subrutina normal, aquí no s'apliquen les regles de l'ABI sobre registres salvats i temporals: com que l'RSE no ha estat invocada voluntàriament pel programa, cal salvar-los tots sense excepció.
2. **Identificar la causa**: l'RSE llegeix el registre `mcause` per determinar el tipus d'excepció o interrupció i saltar a la **rutina específica** (*exception handler*) corresponent.
3. **Tractar l'excepció o interrupció**: el tractament depèn de la causa:
   - **Resoldre el problema i retornar** (fallada de pàgina): l'RSE resol la situació i el programa es reprèn reexecutant la instrucció causant.
   - **Continuar a la instrucció següent** (crida al sistema, interrupció d'E/S): l'RSE executa el servei sol · licitat i el programa es reprèn a la instrucció següent.
   - **Avortar el programa** (accés no alineat, instrucció il · legal): l'RSE emet un missatge d'error i cedeix el control a un altre procés.

4. **Ajustar `mepc` si cal**: en els casos en què el programa ha de continuar a la instrucció *següent* a la causant (per exemple, `ecall`), l'RSE incrementa `mepc` en 4. En la resta de casos no cal cap ajust (vegeu Secció 9.4.3).
5. **Restaurar l'estat**: l'RSE restaura tots els registres desats al pas 1.
6. **Retornar al programa**: l'RSE executa la instrucció `mret` per retornar al programa interromput (vegeu Secció 9.4.2).

Essencial 9.4

A l'RSE cal desar **tots** els registres generals sense excepció, independentment de les convencions de l'ABI. El programa interromput no ha cedit el control voluntàriament i no pot saber quins registres ha modificat l'RSE.

9.4.2 La instrucció `mret`

La instrucció **`mret`** (*machine return*) finalitza l'execució de l'RSE i restaura l'estat del processador anterior a l'excepció o interrupció. Executa les accions següents, per maquinari:

1. **Restaurar el permís d'interrupcions**: copia `MPIE` a `MIE`, i posa `MPIE=1`.
2. **Restaurar el mode de privilegi**: copia `MPP` al mode actiu, i posa `MPP=U`.
3. **Saltar a l'adreça de retorn**: copia `mepc` al PC.

RISC-V 9.10: RV32I ISA — `mret`

Mnemònic	Operands	Operació	Nom
`mret`	—	$MIE \leftarrow MPIE;\ MPIE \leftarrow 1;\ mode \leftarrow MPP;\ MPP \leftarrow U;\ PC \leftarrow mepc$	*Machine return*

9.4.3 Ajust de `mepc`

En acabar el tractament, l'RSE ha de decidir on reprèn el programa. La regla és:

- **Reexecutar la instrucció causant** (`mepc` sense modificar): quan l'excepció s'ha produït perquè la instrucció no ha pogut completar-se, però la situació ja s'ha resolt (fallada de pàgina). En acabar `mret`, el PC apuntarà a la mateixa instrucció, que ara s'executarà correctament.
- **Continuar a la instrucció següent** (`mepc += 4`): quan la instrucció causant ja ha complert la seva funció i no s'ha de tornar a executar. Els casos principals són:
  - `ecall`: la crida al sistema ja ha estat atesa.
  - `ebreak`, quan el programa l'hi ha posat com una crida explícita al depurador. En un punt de parada de programari, en canvi, el depurador havia substituït una instrucció del programa per `ebreak`: la restaura i hi torna, amb `mepc` sense modificar.
  - En les **interrupcions**, `mepc` ja apunta a la instrucció següent (vegeu Secció 9.3.2), de manera que no cal cap ajust.

Essencial 9.5

Si l'RSE retornés sense ajustar `mepc` en el cas d'una `ecall`, el processador reexecutaria la mateixa instrucció `ecall` indefinidament. L'increment de `mepc` en 4 és responsabilitat de l'RSE, no del maquinari.

9.4.4 Exemple: RSE mínima en RISC-V

El Codi 9.1 és un exemple **il · lustratiu** (no executable directament) d'una RSE mínima que desa tots els registres, identifica la causa i en delega el tractament a un **distribuïdor** (*dispatcher*). Com que una RSE real no viu al segment `.text` habitual d'un programa d'usuari, l'exemple usa la directiva `.section` de GNU `as`, fora de l'abast de RARS:

Codi 9.1: RSE mínima: desa els registres i delega el tractament en el distribuïdor.

`RV32IZicsr`

```
.section .text.trap, "ax"
.align 2
rse:
        # Reservar espai a la pila per a 30 registres (30 × 4 = 120 bytes; s'ometen x0 i sp)
        addi    sp, sp, -120

        # Desar tots els registres generals (excepte x0, que és sempre 0)
        sw      ra,   0(sp)
        sw      t0,   4(sp)
        sw      t1,   8(sp)
        sw      t2,  12(sp)
        sw      s0,  16(sp)
        sw      s1,  20(sp)
        sw      a0,  24(sp)
        sw      a1,  28(sp)
        sw      a2,  32(sp)
        sw      a3,  36(sp)
        sw      a4,  40(sp)
        sw      a5,  44(sp)
        sw      a6,  48(sp)
        sw      a7,  52(sp)
        sw      s2,  56(sp)
        sw      s3,  60(sp)
        sw      s4,  64(sp)
        sw      s5,  68(sp)
        sw      s6,  72(sp)
        sw      s7,  76(sp)
        sw      s8,  80(sp)
        sw      s9,  84(sp)
        sw      s10, 88(sp)
        sw      s11, 92(sp)
        sw      t3,  96(sp)
        sw      t4, 100(sp)
        sw      t5, 104(sp)
        sw      t6, 108(sp)
        # Desar també gp i tp per completesa
        sw      gp, 112(sp)
        sw      tp, 116(sp)

        # Llegir mcause i mepc i passar-los com a paràmetres al dispatcher
```

```
    csrr    a0, mcause          # a0 = mcause (causa de l'excepció/interrupció)
    csrr    a1, mepc            # a1 = mepc (adreça de retorn)
    mv      a2, sp              # a2 = punter als registres desats

    # Cridar el dispatcher, que identificarà la causa i cridarà el handler
    call    handler_dispatcher  # retorna a a0 l'adreça de retorn ajustada

    # Actualitzar mepc amb l'adreça de retorn que ha decidit el dispatcher
    csrw    mepc, a0

    # Restaurar tots els registres generals
    lw      ra,   0(sp)
    lw      t0,   4(sp)
    lw      t1,   8(sp)
    lw      t2,  12(sp)
    lw      s0,  16(sp)
    lw      s1,  20(sp)
    lw      a0,  24(sp)
    lw      a1,  28(sp)
    lw      a2,  32(sp)
    lw      a3,  36(sp)
    lw      a4,  40(sp)
    lw      a5,  44(sp)
    lw      a6,  48(sp)
    lw      a7,  52(sp)
    lw      s2,  56(sp)
    lw      s3,  60(sp)
    lw      s4,  64(sp)
    lw      s5,  68(sp)
    lw      s6,  72(sp)
    lw      s7,  76(sp)
    lw      s8,  80(sp)
    lw      s9,  84(sp)
    lw      s10, 88(sp)
    lw      s11, 92(sp)
    lw      t3,  96(sp)
    lw      t4, 100(sp)
    lw      t5, 104(sp)
    lw      t6, 108(sp)
    lw      gp, 112(sp)
    lw      tp, 116(sp)

    # Alliberar la pila
    addi    sp, sp, 120

    # Retornar al programa: MIE←MPIE, mode←MPP, PC←mepc
    mret
```

El distribuïdor (`handler_dispatcher`) rep la causa (`mcause`), l'adreça de retorn (`mepc`) i un punter als registres desats. Aquest punter li permet llegir els paràmetres de la crida al sistema i modificar els valors que es restauraran; per exemple, escriure el resultat d'una crida a la posició salvada d'`a0`, que serà la que torni al programa en acabar l'RSE. El distribuïdor consulta `mcause` per cridar la rutina específica i retorna l'adreça on ha de continuar el programa: `mepc` (per reexecutar la instrucció causant), `mepc + 4` (per continuar a la instrucció següent) o una altra adreça (per exemple, en un canvi de context).

⚠ Aprofundiment 9.3: La pila de l'RSE

L'exemple usa la pila del programa interromput (`sp`). En un sistema real això és problemàtic: si el programa tenia `sp` corromput o apuntant a una adreça invàlida, l'RSE fallaria immediatament. Els sistemes operatius reals usen una pila separada per al nucli, i el canvi de pila forma part de les accions inicials de l'RSE, sovint amb l'ajut del registre `mscratch`, que el microprogramari inicialitza amb l'adreça de la pila del nucli.

9.5 Crida al sistema (`ecall`)

9.5.1 Protecció: mode U vs. mode M

L'SO és un programa resident al computador que gestiona els recursos compartits: la memòria física, els dispositius d'E/S i l'ús del processador. Per garantir que cap programa d'usuari pugui accedir directament a aquests recursos i comprometre la seguretat o l'estabilitat del sistema, el codi de l'SO només s'executa en **mode M** (o mode S en sistemes amb Linux), mentre que els programes d'usuari s'executen en **mode U**.

Els privilegis del mode M permeten dues coses que el mode U no pot fer:

- **Accedir a adreces de memòria reservades a l'SO**: en mode U, un accés a aquestes adreces genera una excepció d'accés no permès.
- **Executar instruccions privilegiades** (com `mret`) **o accedir a CSR de mode M o S**: en mode U, intentar-ho genera una excepció d'instrucció il · legal.

9.5.2 `ecall` com a porta d'entrada controlada a l'SO

Per tal que un programa d'usuari pugui sol · licitar un servei a l'SO, cal un mecanisme que permeti passar de mode U a mode M de forma controlada, sense concedir al programa accés il · limitat als privilegis del sistema. La solució és precisament el mecanisme d'excepcions: la instrucció **`ecall`** (*environment call*) causa una excepció que activa automàticament el mode M i salta a l'RSE de l'SO.

ℹ RISC-V 9.11: RV32I ISA — `ecall` i `ebreak`

Mnemònic	Operands	Operació	Nom
`ecall`	—	Provoca una excepció (crida al sistema)	*Environment call*
`ebreak`	—	Provoca una excepció (punt de parada)	*Environment break*

Això garanteix que hi ha un únic punt d'entrada a l'SO (l'adreça de l'RSE a `mtvec`) i que el programa d'usuari no pot triar on salta ni amb quins privilegis ho fa. En acabar el tractament, l'RSE executa `mret`, que restaura el mode U i retorna el control al programa.

9.5.3 Convenció de crides al sistema a RARS

Els SO estableixen una convenció de crides al sistema per passar el codi de servei, els paràmetres de les crides i els registres que cal emprar. Per exemple, RARS estableix el codi de servei al registre **`a7`** i l'argument principal i el valor de retorn al registre **`a0`**. També estableix dos codis de servei per sortir del programa, `exit` (codi de servei 10) i `exit2` (codi de servei 93). La diferència és que `exit2` permet

especificar el codi de sortida (normalment 0 per a una terminació correcta), que es retorna al procés pare (normalment l'SO).

Els dos números tenen orígens diferents, i això explica la duplicitat. El catàleg de RARS [9] combina dues famílies de serveis: els heretats del *MIPS Assembler and Runtime Simulator* (MARS), el simulador de MIPS del qual RARS descendeix, i els compatibles amb les eines de Linux. El **10** prové de la primera. El **93**, en canvi, és el número real de la crida `exit` a **Linux/RISC-V** (`asm-generic/unistd.h`), el mateix origen que el `63` (`read`), el `64` (`write`) o el `214` (`brk`, que RARS no té: hi fa servir `sbrk`, el 9) de Secció 9.5.4. RARS conserva els dos noms heretats de MARS i adopta per a `exit2` el número de Linux; per això el mateix servei apareix com a `exit2` a la taula de RARS i com a `exit` a la de Linux.

RISC-V 9.12: RARS — Registres de crida al sistema

- **`a7`**: codi de servei.
- **`a0`**: argument principal i valor de retorn.

RISC-V 9.13: RARS — Codis de servei més rellevants

Codi	Nom	Arguments	Retorn
1	`print_int`	`a0` = enter	—
2	`print_float`	`fa0` = flotant	—
4	`print_string`	`a0` = adreça de la cadena	—
5	`read_int`	—	`a0` = enter
6	`read_float`	—	`fa0` = flotant
8	`read_string`	`a0` = adreça de la memòria intermèdia, `a1` = longitud màxima	—
10	`exit`	—	—
11	`print_char`	`a0` = caràcter (ASCII)	—
12	`read_char`	—	`a0` = caràcter (ASCII)
34	`print_hex`	`a0` = enter	—
35	`print_bin`	`a0` = enter	—
36	`print_unsigned`	`a0` = natural	—
93	`exit2`	`a0` = codi de sortida	—

Els exemples següents mostren les crides al sistema més habituals a RARS:

Exemple 9.1: Crides al sistema a RARS

Imprimir un enter (codi 1):

RV32I

```
        li      a7, 1               # codi de print_int
        li      a0, 42              # enter a imprimir
        ecall
```

Llegir un enter de l'entrada estàndard (codi 5):

RV32I

```
        li      a7, 5               # codi de read_int
        ecall                       # a0 ← enter llegit
```

Imprimir una cadena de caràcters (codi 4):

RV32I

```
        li      a7, 4               # codi de print_string
        la      a0, missatge        # adreça de la cadena (acabada en '\0')
        ecall
```

Finalitzar el programa (codi 10):

RV32I

```
        li      a7, 10              # codi de exit
        ecall                       # no retorna
```

A EC, però, la sortida dels programes es fa sempre amb `exit2` (codi 93), que permet especificar el codi de sortida a `a0` (RISC-V 2.19):

RV32I

```
        li      a7, 93              # codi d'exit2
        li      a0, 0               # codi de sortida: 0
        ecall                       # no retorna
```

9.5.4 Convenció ABI de Linux

Les crides al sistema segueixen una convenció estricta per passar el codi de servei i els paràmetres. A Linux/RISC-V:

- **`a7`**: codi de la crida al sistema (*syscall number*).
- **`a0`**–**`a5`**: arguments (fins a sis).
- **`a0`**: valor de retorn (i `a1` per a resultats de 64 bits en RV32).

Essencial 9.6

La convenció de crides al sistema **no és la mateixa** que la convenció de crida a subrutines de l'ABI. En particular, el codi de servei va a **`a7`**, no a `a0`, i hi pot haver fins a sis arguments (`a0`–`a5`), en lloc dels vuit (`a0`–`a7`) de la convenció de crida a subrutines.

Aprofundiment 9.4: Exemples de crides al sistema Linux/RISC-V (RV32)

Els exemples següents mostren les crides al sistema més habituals:
Escriure a la sortida estàndard (`write`, codi 64):

RV32I

```
li      a7, 64              # codi de write
li      a0, 1               # descriptor: stdout
la      a1, missatge        # adreça del buffer
li      a2, 13              # nombre de bytes
ecall                       # a0 ← nombre de bytes escrits (o error)
```

Llegir de l'entrada estàndard (`read`, codi 63):

RV32I

```
li      a7, 63              # codi de read
li      a0, 0               # descriptor: stdin
la      a1, buffer          # adreça del buffer
li      a2, 64              # màxim de bytes a llegir
ecall                       # a0 ← nombre de bytes llegits (o error)
```

Finalitzar el programa (`exit`, codi 93):

RV32I

```
li      a7, 93              # codi de exit
li      a0, 0               # codi de retorn (0 = èxit)
ecall                       # no retorna
```

Reservar memòria dinàmica (`brk`, codi 214):

RV32I

```
li      a7, 214             # codi de brk
li      a0, 0               # 0 = consultar l'adreça actual del heap
ecall                       # a0 ← adreça actual (o nova) del final del heap
```

Aprofundiment 9.5: Taula de crides al sistema Linux/RISC-V (RV32) més rellevants

Taula 9.2: Codi, arguments i retorn de cada crida al sistema.

Codi	Nom	Arguments	Retorn
17	`getcwd`	`a0`=buf, `a1`=size	`a0`=buf o error
23	`dup`	`a0`=fd	`a0`=nou fd o error
25	`fcntl`	`a0`=fd, `a1`=cmd, `a2`=arg	`a0`=resultat o error
29	`ioctl`	`a0`=fd, `a1`=req, `a2`=arg	`a0`=0 o error
35	`unlinkat`	`a0`=dirfd, `a1`=path, `a2`=flags	`a0`=0 o error
48	`faccessat`	`a0`=dirfd, `a1`=path, `a2`=mode	`a0`=0 o error
56	`openat`	`a0`=dirfd, `a1`=path, `a2`=flags, `a3`=mode	`a0`=fd o error
57	`close`	`a0`=fd	`a0`=0 o error

61	`getdents64`	`a0`=fd, `a1`=buf, `a2`=count	`a0`=bytes o error
62	`llseek`	`a0`=fd, `a1`=offset_hi, `a2`=offset_lo, `a3`=resultat, `a4`=whence	`a0`=0 o error
63	`read`	`a0`=fd, `a1`=buf, `a2`=count	`a0`=bytes o error
64	`write`	`a0`=fd, `a1`=buf, `a2`=count	`a0`=bytes o error
65	`readv`	`a0`=fd, `a1`=iov, `a2`=iovcnt	`a0`=bytes o error
66	`writev`	`a0`=fd, `a1`=iov, `a2`=iovcnt	`a0`=bytes o error
78	`readlinkat`	`a0`=dirfd, `a1`=path, `a2`=buf, `a3`=size	`a0`=bytes o error
93	`exit`	`a0`=codi	—
94	`exit_group`	`a0`=codi	—
96	`set_tid_address`	`a0`=tidptr	`a0`=tid
99	`set_robust_list`	`a0`=head, `a1`=len	`a0`=0 o error
129	`kill`	`a0`=pid, `a1`=sig	`a0`=0 o error
131	`tgkill`	`a0`=tgid, `a1`=tid, `a2`=sig	`a0`=0 o error
160	`uname`	`a0`=buf	`a0`=0 o error
172	`getpid`	—	`a0`=pid
173	`getppid`	—	`a0`=ppid
174	`getuid`	—	`a0`=uid
175	`geteuid`	—	`a0`=euid
176	`getgid`	—	`a0`=gid
177	`getegid`	—	`a0`=egid
178	`gettid`	—	`a0`=tid
179	`sysinfo`	`a0`=info	`a0`=0 o error
214	`brk`	`a0`=addr	`a0`=addr o error
215	`munmap`	`a0`=addr, `a1`=len	`a0`=0 o error
216	`mremap`	`a0`=old, `a1`=old_sz, `a2`=new_sz, `a3`=flags	`a0`=addr o error
222	`mmap`	`a0`=addr, `a1`=len, `a2`=prot, `a3`=flags, `a4`=fd, `a5`=off	`a0`=addr o error
261	`prlimit64`	`a0`=pid, `a1`=res, `a2`=new, `a3`=old	`a0`=0 o error
278	`getrandom`	`a0`=buf, `a1`=len, `a2`=flags	`a0`=bytes o error
291	`statx`	`a0`=dirfd, `a1`=path, `a2`=flags, `a3`=mask, `a4`=buf	`a0`=0 o error
403	`clock_gettime64`	`a0`=clkid, `a1`=tp	`a0`=0 o error

A RV32, algunes d'aquestes crides corresponen a la variant específica per a mides de 32 bits (per exemple, `fcntl64` en lloc de `fcntl`), tot i que el número i els arguments es mantenen; `mmap` rep el desplaçament en pàgines de 4 KiB (semàntica `mmap2`) en lloc de bytes. RV32 no té `fstatat` ni `fstat` (79 i 80, que només existeixen a RV64): la informació d'un fitxer s'obté amb `statx`.

9.6 Interrupcions de dispositius d'E/S

9.6.1 Sincronització per enquesta vs. per interrupcions

La comunicació amb els dispositius d'E/S està dominada per la necessitat de tolerar les seves latències llargues. Quan un programa inicia una operació en un dispositiu (per exemple, una lectura de disc), l'operació pot trigar milions de cicles en completar-se. Hi ha dues estratègies per gestionar aquesta espera:

- **Sincronització per enquesta** (*polling*): el processador consulta repetidament l'estat del dispositiu fins que l'operació acaba. És simple d'implementar, però desaprofita completament el processador durant l'espera.

`RV32I`

```
bucle:
        lw      t0, ESTAT_DISPOSITIU    # llegir l'estat del dispositiu
        li      t1, READY               # valor que indica «llest»
        bne     t0, t1, bucle           # esperar fins que estigui llest
    # continuar quan el dispositiu ha acabat
```

- **Sincronització per interrupcions**: el processador continua executant altres tasques mentre el dispositiu treballa. Quan l'operació acaba, el dispositiu activa un senyal de **petició d'interrupció** que notifica al processador. El processador interromp el programa en curs, atén el dispositiu a l'RSE i després reprèn el programa.

Essencial 9.7

La sincronització per **interrupcions** és molt més eficient que la sincronització per enquesta: permet al processador fer feina útil mentre el dispositiu treballa, en lloc d'esperar activament. L'SO pot aprofitar aquest temps per executar altres programes.

9.6.2 El cicle de vida d'una interrupció

El flux complet d'una interrupció, des de la petició fins al retorn, segueix les etapes següents (Figura 9.1):

1. **Petició**: el dispositiu activa el seu senyal de petició d'interrupció. El bit corresponent de `mip` (per exemple, `MEIP` per a una interrupció externa) s'activa i es manté actiu mentre el dispositiu mantingui el senyal.
2. **Detecció**: en acabar cada instrucció, la unitat de control comprova si hi ha algun bit actiu a `mip` amb el seu permís corresponent a `mie` habilitat i amb `MIE=1`. Si es compleixen les tres condicions, es genera una interrupció.

3. **Acceptació**: el maquinari executa les accions de Secció 9.3.1: guarda el PC a `mepc`, escriu `mcause` amb bit 31=1 i el codi de causa, desactiva `MIE` i salta a `mtvec`.
4. **Servei**: l'RSE identifica el dispositiu (llegint `mcause` i `mip`) i crida la rutina específica d'aquell dispositiu, que en fa el servei (per exemple, llegir-ne les dades).
5. **Retorn**: l'RSE executa `mret`, que restaura `MIE` i retorna al programa interromput, que reprèn l'execució a la instrucció que hauria executat a continuació (ja guardada a `mepc`).

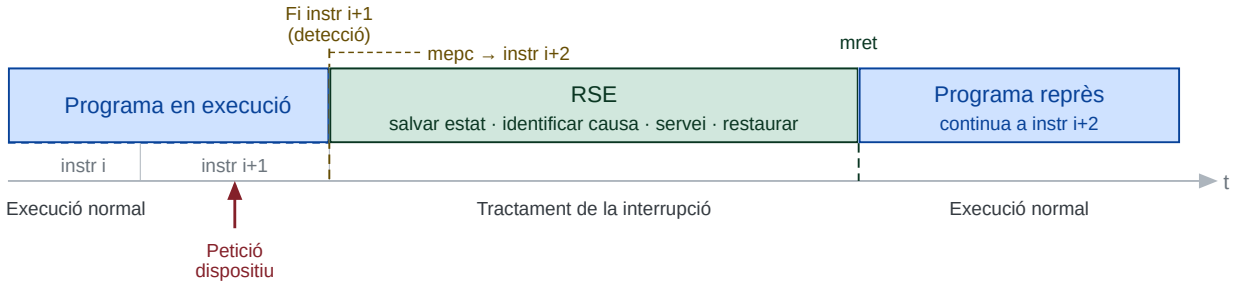


Figura 9.1: Cicle de vida d'una interrupció: el dispositiu fa la petició mentre s'executa la instrucció $i+1$; en acabar-la, el processador detecta la interrupció, guarda `mepc` apuntant a la instrucció $i+2$ i salta a l'RSE; en retornar amb `mret`, el programa es reprèn a partir de $i+2$.

9.6.3 Permisos: `MIE` global i màscares `mie`

El sistema de permisos d'interrupcions té dos nivells independents:

- **Permís global `MIE`** (bit 3 de `mstatus`): habilita o deshabilita totes les interrupcions alhora. El maquinari el posa a 0 automàticament en acceptar qualsevol excepció o interrupció, i `mret` el restaura al valor de `MPIE`.
- **Màscares individuals `mie`**: cada bit habilita o deshabilita independentment un tipus d'interrupció. L'SO els configura durant la inicialització per indicar quins dispositius poden interrompre el processador.

Essencial 9.8

Perquè una interrupció sigui atesa cal que es compleixin **simultàniament** les tres condicions de Secció 9.2.7:

1. `MIE = 1` (permís global habilitat).
2. El bit corresponent de `mie` val 1 (permís específic habilitat).
3. El bit corresponent de `mip` val 1 (petició pendent activa).

Si qualsevol d'aquestes tres condicions no es compleix, la interrupció queda pendent però no s'atén.

9.6.4 Identificació del dispositiu a l'RSE

Quan l'RSE rep el control per una interrupció (bit 31 de `mcause` val 1), ha d'identificar quin dispositiu ha fet la petició i si té permís per ser atès:

1. Llegir `mcause`: el codi de causa indica el tipus d'interrupció (per exemple, codi 11 = interrupció externa en mode M).
2. Llegir `mip`: examinar quins bits de petició estan actius.
3. Creuar amb `mie`: verificar quins d'aquests dispositius tenen el permís habilitat.
4. Si n'hi ha més d'un, aplicar un criteri de **prioritat** per decidir quin s'atén primer.

⚠ Aprofundiment 9.6: Múltiples peticions simultànies

Pot succeir que mentre les interrupcions estaven inhibides (`MIE=0`) arribin peticions de diversos dispositius. En reprendre's el permís, tots els bits corresponents de `mip` estaran actius simultàniament. L'RSE ha d'establir un ordre de prioritat per atendre'ls. Entre els tipus d'interrupció, RISC-V fixa un ordre per maquinari (en mode M, externa, de programari i de temporitzador: `MEIP`, `MSIP`, `MTIP`), i `mcause` identifica la de més prioritat. Entre els dispositius que comparteixen la interrupció externa, en canvi, l'ordre és responsabilitat de l'SO, que implementa la política que consideri adequada (per exemple, prioritat fixa, *round-robin*, etc.).
A més, pot succeir que un dispositiu hagi desactivat el senyal de petició abans que l'RSE l'atengui (per exemple, per expiració d'un temporitzador intern). En aquest cas, el bit de `mip` ja no estarà actiu quan l'RSE el comprovi, i la petició es perdrà.

9.7 La fallada de pàgina

La **fallada de pàgina** és l'exemple més complet del mecanisme d'excepcions d'aquest tema: combina la detecció per maquinari, el tractament per programari amb una RSE i la reexecució de la instrucció causant. A més, és el punt de trobada entre els mecanismes d'excepcions i la memòria virtual del Capítol 8.

9.7.1 Del TLB a la fallada de pàgina

Com s'ha vist a Secció 8.5, el *translation lookaside buffer* (TLB) és una memòria cau de la taula de pàgines que guarda les traduccions d'adreces virtuals a físiques més recents. Quan la traducció d'una adreça no és al TLB, es produeix una **fallada de TLB** (*TLB miss*).

A RISC-V, la fallada de TLB **no és una excepció**: la resol de manera transparent el recorregut de la taula de pàgines (*page-table walker*), un circuit de l'MMU que hi llegeix la PTE, la copia al TLB i deixa continuar l'accés sense interrompre el programa (Secció 8.5.3). L'excepció apareix quan el walker troba que la pàgina **no és a la memòria física** (bit V = 0 a la PTE): aleshores es genera una **fallada de pàgina** (*page fault*), que el sistema operatiu tracta per programari. Amb la PTE ja al TLB, una escriptura en una pàgina amb E = 0 també genera una excepció, la de protecció (Secció 8.6.2).

🔥 Essencial 9.9

A RISC-V, la **fallada de TLB** la resol el maquinari (el recorregut de la taula de pàgines) de forma transparent. Només la **fallada de pàgina** (V = 0) és una excepció: la instrucció causant s'avorta, `sepc` (o `mepc`) hi apunta, i l'RSE l'ha de **reexecutar** en acabar (sense incrementar `sepc`). En la reexecució, el recorregut de la taula de pàgines troba la PTE vàlida i l'accés continua normalment.

El bit V de la PTE que desencadena la fallada és el que s'ha estudiat a Secció 8.3.1. Cal recordar que l'especificació RISC-V l'anomena `V` —no `P`—, tal com s'explica a Aprofundiment 8.1.

9.7.2 Codi de causa i adreça de la fallada

Una fallada de pàgina escriu a `scause` (o `mcause` en mode M) el codi 12, 13 o 15 segons si l'accés era una cerca d'instrucció, una lectura o una escriptura de dada (vegeu RISC-V 9.4). El registre `stval` (o `mtval`) conté l'**adreça virtual** que ha causat la fallada, que l'RSE necessita per localitzar la pàgina i la PTE corresponents a la taula de pàgines del procés.

9.7.3 Implementació en mode S

9.7.3.1 Registres CSR de memòria virtual

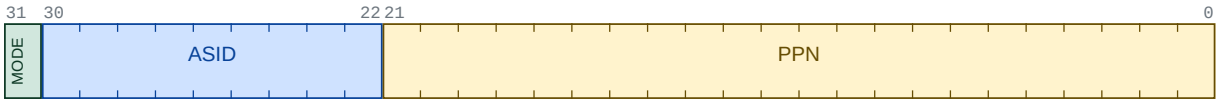
En un sistema amb Linux, la memòria virtual i les fallades de pàgina es gestionen en **mode S**. Els registres CSR específics de la traducció d'adreces són els que es mostren a RISC-V 9.14; `sepc` i `scause` ja s'han presentat com a equivalents en mode S de `mepc` i `mcause` (RISC-V 9.9).

RISC-V 9.14: RV32I ISA — Registres CSR de memòria virtual

Registre	Nom complet	Funció
`satp`	*Supervisor Address Translation and Protection*	Mode de traducció i adreça física de la taula de pàgines de primer nivell
`stval`	*Supervisor Trap Value*	Adreça virtual que ha causat la fallada de pàgina

El registre `satp` té el format següent en RV32:

RISC-V 9.15: RV32I ISA — `satp`



- **`MODE`** (bit 31): `0` = sense traducció d'adreces (*bare*); `1` = traducció Sv32 activa.
- **`ASID`** (bits 30:22, *Address Space Identifier*): identificador de l'espai d'adreces, que permet al TLB distingir traduccions de processos diferents sense haver-lo de buidar en cada canvi de context.
- **`PPN`** (bits 21:0): número de pàgina física (*Physical Page Number*) de la taula de pàgines de primer nivell.

Quan l'RSE en mode S modifica la taula de pàgines en memòria, executa la instrucció **`sfence.vma`** per invalidar les entrades del TLB afectades i garantir que el recorregut de la taula de pàgines torni a llegir la PTE actualitzada.

RISC-V 9.16: Supervisor ISA — `sfence.vma`

Mnemònic	Operands	Operació	Nom
`sfence.vma`	`rs1, rs2`	Invalida les entrades del TLB per a l'adreça `rs1` i l'ASID `rs2` (`x0` = totes)	*Supervisor fence*

`sfence.vma` forma part de l'arquitectura privilegiada de RISC-V: està disponible sempre que el processador implementi el mode S.

9.7.3.2 La rutina de fallada de pàgina

L'RSE del nucli, en rebre una fallada de pàgina (codi 12, 13 o 15 a `scause`), llegeix `stval`, localitza la PTE corresponent a la taula de pàgines del procés actiu i actua segons el bit V:

`RV32IZicsr`

```
page_fault_handler:
        csrr    t0, stval               # t0 = adreça virtual que ha causat la fallada

        # Localitzar la PTE de t0 a la taula de pàgines del procés actiu (via satp).
        # L'SO coneix l'estructura de la seva taula; aquí se n'omet el recorregut.
        call    locate_pte              # t1 = adreça de la PTE de t0
        lw      t3, 0(t1)               # t3 = contingut de la PTE

        andi    t4, t3, 1               # t4 = bit V
        bnez    t4, reexecuta           # V=1: la pàgina ja és a memòria física

        # V=0: portar la pàgina del disc a un marc lliure (o reemplaçar-ne un;
        #      si el marc víctima té D=1, escriure'l primer al disc).
        call    load_page_from_disk     # actualitza la PTE: V←1, PPN←marc, D←0

reexecuta:
        sfence.vma t0, zero             # sincronitza el TLB amb la PTE actualitzada
        sret                            # reexecuta la instrucció causant (sepc sense modificar)
```

Quan la pàgina es carrega del disc, la còpia a memòria és idèntica a la del disc, de manera que la PTE s'inicialitza amb $\mathbf{D = 0}$. Si l'adreça de `stval` no pertany a cap regió vàlida del procés, la rutina avorta el programa (*segmentation fault*) en lloc de carregar cap pàgina.

9.7.3.3 El bit D en la fallada de pàgina

El bit D (*dirty*, o **bit de modificació**) d'una PTE indica si la pàgina s'ha modificat respecte de la còpia al disc. Compleix la mateixa funció que el bit de modificació de les memòries cau d'escriptura retardada (Secció 7.5.1). La seva gestió en la memòria virtual ja s'ha descrit a Secció 8.5.5: el maquinari el posa a 1 en la primera escriptura, simultàniament al TLB i a la PTE.

En el context de la fallada de pàgina, el bit D té dos efectes:

- **En carregar** una pàgina del disc, la PTE s'inicialitza amb D = 0, ja que la còpia a memòria és idèntica a la del disc.

- **En reemplaçar** un marc, l'RSE consulta el bit D de la pàgina víctima: si D = 1, l'escriu primer al disc; si D = 0, el marc es pot sobreescriure directament (vegeu Secció 8.4).

> **Aprofundiment 9.7: Actualització dels bits A i D**
>
> L'especificació RISC-V permet que el maquinari actualitzi els bits A (*accessed*) i D de la PTE de forma autònoma (extensió Svadu). Les implementacions que no ho admeten generen una excepció de fallada de pàgina en el primer accés a una pàgina nova —o en la primera escriptura a una pàgina amb D = 0—, i l'RSE en mode S actualitza els bits A i D manualment abans de reexecutar la instrucció causant.

Part II

Problemes

Tema 1: Introducció

1.1 El cicle de vida del programa: Generació vs. execució

Vegeu Secció 1.2.

Problema 1.1. Es vol executar un programa escrit en Python en un ordinador RISC-V, però només es disposa dels elements següents:

- Un compilador de C a llenguatge màquina x86-64, escrit en llenguatge màquina x86-64.
- Un traductor binari de llenguatge màquina x86-64 a llenguatge màquina RISC-V, escrit en llenguatge màquina RISC-V.
- Un intèrpret de Python, escrit en C.

Quins passos cal fer per executar el programa en Python en l'ordinador RISC-V, usant solament els elements disponibles?

Solució 1

Problema 1.2. Es vol executar un programa escrit en C en un ordinador x86-64, però només es disposa dels elements següents:

- Un intèrpret de Python, escrit en llenguatge màquina RISC-V.
- Un compilador de C a llenguatge màquina x86-64, escrit en Python.
- Un traductor de llenguatge màquina RISC-V a llenguatge màquina x86-64, escrit en llenguatge màquina x86-64.

a) Quins passos cal fer per executar el programa en C en l'ordinador x86-64, usant solament els elements disponibles?
b) Quins passos caldria fer si es vol tornar a executar el mateix programa?

Solució 2

1.2 Codificació de nombres naturals i enters

Vegeu Secció 1.4, Secció 1.4.5 i Secció 1.5.

Problema 1.3.

a) Interpreteu els patrons de bits següents com a naturals i calculeu-ne el valor decimal:

Patró de bits	n	Valor decimal
`0001 1001`	8	
`1010 0110`	8	
`0000 0000 1010 1101`	16	

b) Representeu els naturals següents en binari amb el nombre de bits indicat. Si el valor no és representable amb el nombre de bits indicat, indiqueu-ho amb NC (no codificable).

Decimal	n	Binari
25	8	
173	8	
256	8	
173	16	

Solució 3

Problema 1.4. Quin és l'enter més petit (el més negatiu) representable en complement a 2 amb 13 bits?

a) Escriviu el patró de 13 bits i expresseu-lo en hexadecimal (completeu-lo amb zeros a l'esquerra fins a 16 bits).
b) Expresseu-lo en decimal.

Solució 4

Problema 1.5. Quin és l'enter positiu més gran representable en complement a 2 amb 13 bits?

a) Escriviu el patró de 13 bits i expresseu-lo en hexadecimal (completeu-lo amb zeros a l'esquerra fins a 16 bits).
b) Expresseu-lo en decimal.

Solució 5

Problema 1.6.

a) Completeu la taula de rang i representació del zero per als formats d'enters en n bits:

Format	Rang	Representació del zero
Signe i magnitud		
Complement a 1		
Complement a 2	$[-2^{n-1},\ 2^{n-1}-1]$	$0\ldots00$
Excés a $2^{n-1}-1$		

b) Completeu la taula de codificació en 4 bits per a aquests enters de l'interval $[-8, 8]$. Per als valors no codificables amb 4 bits, poseu NC.

Decimal	Ca2	Ca1	Excés a 7	Signe i magnitud
8				
7				
4				
1				
0				
-1				
-4				

Decimal	Ca2	Ca1	Excés a 7	Signe i magnitud
-7				
-8				

Solució 6

Problema 1.7. Convertiu els enters decimals següents als formats de representació binària per a 8 bits. Escriviu el resultat en hexadecimal:

Decimal	Ca1	Ca2	Signe i magnitud	Excés a $2^7 - 1$
-78				
125				
0				
-1				

Solució 7

Problema 1.8. Donats els patrons de bits següents, calculeu-ne el valor decimal interpretant-los primer com a **natural** i després com a **enter en Ca2**:

a) `0000 1111`
b) `1000 0000`
c) `1111 1111`
d) `1111 1111 1111 0000`

Solució 8

Problema 1.9. Apliqueu l'extensió de signe als patrons de 8 bits següents per obtenir la representació equivalent en 16 bits. Expresseu el resultat en hexadecimal i comproveu que el valor decimal representat és el mateix.

a) `0011 0101`
b) `1100 1101`

Solució 9

Problema 1.10. Convertiu els nombres enters següents, representats en complement a 2 de 16 bits, a la representació decimal amb signe:

a) `0x0123`
b) `0xFE23`

Solució 10

Problema 1.11. Codifiqueu en complement a 2 els enters següents, usant el format més petit possible (8 o 16 bits), i expresseu el resultat en hexadecimal:

a) $+123$
b) $+255$
c) -127
d) $+170$

e) $+128$
f) -230
g) -18
h) -128
i) -1

Solució 11

1.3 Operacions aritmètiques en complement a 2

Vegeu Secció 1.6.

Problema 1.12. Efectueu les sumes següents en complement a 2 amb $n = 8$ bits. Per a cada resultat, indiqueu si hi ha desbordament i justifiqueu-ho.

a) $45 + 30$
b) $100 + 80$
c) $(-45) + (-30)$
d) $(-100) + (-80)$
e) $45 + (-30)$

Solució 12

Problema 1.13. Efectueu les restes següents en complement a 2 amb $n = 8$ bits, usant el mètode de negació i suma. Per a cada resultat, indiqueu si hi ha desbordament.

a) $45 - 30$
b) $30 - 45$
c) $(-100) - 45$

Solució 13

Problema 1.14. Efectueu les multiplicacions següents en complement a 2 amb $n = 8$ bits, usant el mètode de magnituds i signe. Indiqueu si el resultat és representable en 8 bits o si hi ha desbordament.

a) 6×5
b) $(-6) \times 5$
c) $(-6) \times (-5)$
d) 20×10

Solució 14

Problema 1.15. Efectueu les divisions següents en complement a 2 amb $n = 8$ bits, usant el mètode de magnituds i signe. Per a cada resultat, indiqueu el quocient i el residu en Ca2, o bé si hi ha desbordament.

a) $30 \div 5$
b) $(-30) \div 5$
c) $(-30) \div (-5)$
d) $(-7) \div 2$
e) $(-128) \div (-1)$

Solució 15

Tema 2: Instruccions i tipus de dades bàsics

2.1 Operands en registre

Vegeu Secció 2.6.1.

Problema 2.1. Traduïu els fragments següents de C a assemblador RISC-V. Suposeu que les variables `f`, `g`, `h`, `i` i `j` són enters de 32 bits assignats als registres `t0`, `t1`, `t2`, `t3` i `t4` respectivament.

a) `f = g + h + i + j`
b) `g = f + (h + 5) - i`

Problema 2.2. Traduïu els fragments següents d'assemblador RISC-V a C. Suposeu que les variables `f`, `g`, `h`, `i` i `j` són enters de 32 bits assignats als registres `t0`, `t1`, `t2`, `t3` i `t4` respectivament.

a)

RV32I

```
        add     t0, t1, t2
        add     t1, t3, t4
        add     t1, t1, t1
        add     t0, t0, t1
```

b)

RV32I

```
        add     t1, t1, t2
        add     t3, t3, t4
        add     t0, t1, t3
        addi    t0, t0, 2
        add     t0, t0, t0
```

2.2 Constants i immediats

Vegeu Secció 2.6.2 i RISC-V 2.16.

Problema 2.3. Donades les declaracions de variables del Codi 2.1, emmagatzemades a partir de l'adreça `0x10010000`:

Codi 2.1: Quatre etiquetes, `P`, `Q`, `R` i `S`, amb paraules inicialitzades.

RV32I

```
.data
P:      .word   5, 2
Q:      .word   3
R:      .word   4, 0x0FFFF, 0x4180
S:      .word   0x10010008, 0
```

Tots els apartats parteixen del mateix estat inicial.

a) Quin és el valor final de la variable `Q` després d'executar el Codi 2.2?

Codi 2.2: Codi de l'apartat a).

RV32I

```
.text
        la      t1, R
        lw      t2, 12(t1)
        lb      t1, 8(t2)
        la      t3, Q
        lb      t4, 0(t3)
        add     t1, t1, t4
        sb      t1, 0(t3)
```

---

b) Expliqueu textualment i de forma concisa què fa el Codi 2.3:

Codi 2.3: Fragment de l'apartat b).

RV32I

```
.text
        la      t1, P+1
        lw      t1, 0(t1)
```

Nota: la sintaxi `la label+offset` és admesa a EC (teoria i exàmens), però a RARS cal substituir-la per `la` + `addi`.

Problema 2.4. Indiqueu quin és el contingut de memòria a partir de l'adreça `0x10010000`, per paraules, per a la declaració de variables globals del Codi 2.4, amb les restriccions d'alineació de l'ABI (RISC-V 2.21; un `.dword` s'alinea a 8 bytes, encara que RARS 1.6 l'alinea a 4):

Codi 2.4: Declaració de bytes, mitges paraules, paraules i una doble paraula.

RV32I

```
.data
        .byte   0, 1, 2
P:      .byte   5, -1
Q:      .half   0x66
R:      .word   4, 0x0FDF
S:      .half   3
T:      .byte   25
U:      .byte   0x08
V:      .dword  0x10010008
```

Problema 2.5. En RISC-V, la pseudoinstrucció `li rd, imm` s'expandeix en una sola instrucció real (`addi`) si l'immediat cap en 12 bits amb extensió de signe, i en dues (`lui` + `addi`) si no hi cap.

a) Quina seqüència d'instruccions genera l'assemblador per carregar el valor `0xDEADBEEF` al registre `t0`? Calculeu els valors exactes dels operands de cada instrucció.
b) Per què cal ajustar el camp immediat d'`addi` quan el bit 11 del valor original és 1? Il · lustreu-ho amb el valor `0xDEADBEEF`.
c) Quin és el rang de valors immediats que accepta `addi` (és a dir, el rang de constants que es pot carregar a un registre amb `addi rd, zero, imm`)? I quin és el rang de constants que es pot carregar a un registre amb la seqüència `lui` + `addi`?

2.3 Operands en memòria

Vegeu Secció 2.6.3.

Problema 2.6. Donada la declaració de dades del Codi 2.5, emmagatzemada a partir de l'adreça `0x10010000`:

Codi 2.5: Declaració de bytes i paraules sense etiqueta.

RV32I

```
.data
        .byte   1, 2, 3, 4
        .word   -1, 1, -2, 2, -3, 3
        .word   0x12345678
```

Indiqueu quin és el valor en hexadecimal del **byte** emmagatzemat a cada una de les adreces `0x1001000C` i `0x1001001C`. Si no es pot determinar a partir de la declaració, indiqueu-ho amb la sigla de no aplicable (NA).

Problema 2.7. Donats els continguts inicials de registres i memòria següents (en hexadecimal):

`t0 = 0x10010000`, `t2 = 0x10010008`

Adreça	Contingut (byte a byte, hex)
`0x10010000`	`00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F`
`0x10010010`	`00 00 00 00 EC FD 0E 0F 00 00 00 00 00 00 00 00`

Determineu, per a cada instrucció, quin registre o adreça de memòria es modifica i quin és el seu contingut:

RV32I

```
        lw      t1,  0(t0)
        lw      t3,  0(t2)
        sw      t3, -8(t2)
        lw      t4,  4(t0)
        sw      t1, -4(t2)
        sw      t4,  0(t2)
```

Problema 2.8. Donada la declaració del Codi 2.6:

Codi 2.6: Vector `A` de cinc bytes.

`RV32I`

```
.data
A:      .byte   0, 0, 0, 0, 0
```

Escriviu un fragment de no més de 12 línies de codi assemblador que guardi **el valor complet de `t1`** (32 bits) a partir de l'adreça `A+1`, sense produir cap excepció d'alineació. Podeu escriure valors temporals a la posició `A` de memòria si cal (és una adreça alineada a paraula).

Problema 2.9. Indiqueu quin és el contingut de memòria a partir de l'adreça `0x10010000`, byte a byte, per a la declaració de variables globals del Codi 2.7, amb les restriccions d'alineació de l'ABI (RISC-V 2.21; un `.dword` s'alinea a 8 bytes, encara que RARS 1.6 l'alinea a 4):

Codi 2.7: Declaració de bytes, mitges paraules, paraules i dobles paraules, amb caràcters.

`RV32I`

```
.data
        .byte   0x10
P:      .half   -5
Q:      .word   -1, 67
R:      .byte   -4, '5', 6
S:      .half   66, 67
T:      .dword  0x5799
U:      .byte   'C'
V:      .half   0x66
W:      .dword  579
```

Problema 2.10. RISC-V és una arquitectura *little-endian*.

a) Expliqueu què significa *little-endian* i en quin ordre s'emmagatzemen els bytes d'un enter de 32 bits a memòria.
b) La variable global `int x = 0x12345678` s'emmagatzema a partir de l'adreça `0x10010000`. Indiqueu el contingut byte a byte de les adreces `0x10010000` a `0x10010003`.
c) Després d'executar el Codi 2.8, quin valor té `t1`? Raoneu la resposta tenint en compte l'ordre dels bytes.

Codi 2.8: Codi de l'apartat c).

`RV32I`

```
.data
x:      .word   0x12345678
.text
        la      t0, x
        lb      t1, 0(t0)
```

d) Quin hauria de ser el resultat si l'arquitectura fos *big-endian*?

Problema 2.11. Suposeu que el registre `t0` conté l'adreça `0x10010000` i que la memòria té el contingut inicial següent:

Adreça	+0	+1	+2	+3
0x10010000	11	22	CC	DD

a) Determineu el valor dels registres resultants de l'execució de cada instrucció (de forma independent, partint sempre del contingut inicial de memòria). Raoneu l'extensió de signe o per zero quan sigui rellevant.

Instrucció	Resultat al registre
`lb t1, 1(t0)`	
`lb t1, 2(t0)`	
`lh t1, 0(t0)`	
`lh t1, 2(t0)`	
`lbu t1, 2(t0)`	
`lhu t1, 2(t0)`	

b) Determineu l'estat final de la memòria després d'executar les instruccions del Codi 2.9 de forma **seqüencial** (una rere l'altra, partint del contingut inicial):

Codi 2.9: Un `lb` i un `sb`.

RV32I

```
        lb      t1, 2(t0)
        sb      t1, 1(t0)
```

2.4 Representació de naturals i enters

Vegeu els problemes del Tema 1: Introducció: complement a 2, complement a 1, signe i magnitud i excés.

2.5 Representació de caràcters

Vegeu Secció 2.9.

Problema 2.12. Donades les declaracions del Codi 2.10:

Codi 2.10: Dos dígits en ASCII, `u1` i `u2`, i el byte del resultat, `d3`.

RV32I

```
.data
u1:     .byte   '9'
u2:     .byte   '3'
d3:     .byte   0
```

Cadascuna de les dades `u1` i `u2` conté un caràcter ASCII que representa un **dígit decimal individual**. Escriviu un programa en assemblador que sumi els naturals representats per `u1` i `u2` i deixi el resultat en binari (natural) a `d3`. Com que cada operand és un sol dígit (rang $[0, 9]$), la suma màxima és $9 + 9 = 18$ i sempre cap en un sol byte.

Problema 2.13. Escriviu en hexadecimal el contingut de memòria ocupat per les declaracions del Codi 2.11, suposant que comencen a emmagatzemar-se a partir de l'adreça `0x10010000`, amb les restriccions d'alineació de l'ABI (RISC-V 2.21):

Codi 2.11: Declaració de mitges paraules, bytes, paraules i una doble paraula.

`RV32I`

```
.data
p:      .half   54
q:      .byte   'C'
r:      .half   0xAAAA
s:      .word   0x4F3C
t:      .byte   -18
u:      .dword  0x00000000FFFFEEEE
```

Feu servir el format següent:

Adreça	Contingut (byte a byte, hex)
`0x10010000`	__ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __
`0x10010010`	__ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __

Problema 2.14. Escriviu en hexadecimal el contingut de memòria ocupat per les declaracions del Codi 2.12, suposant que comencen a emmagatzemar-se a partir de l'adreça `0x10010000`. Recordeu que la directiva `.align 0` desactiva l'alineació automàtica fins al final del segment `.data` actual (cap `.align k` amb $k > 0$ posterior no la reactiva):

Codi 2.12: Declaració amb `.align 0` i `.align 1`.

`RV32I`

```
.data
        .align 0
        .byte   24
        .align 1
Q:      .byte   'A', 'C', 'E'
R:      .half   512, 0xAAAA
S:      .word   0x43, 0xF3C, 'a'
T:      .byte   -18, 'A'
```

Feu servir el format següent:

Adreça	Contingut (byte a byte, hex)
`0x10010000`	__ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __
`0x10010010`	__ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __

Problema 2.15. Donada la declaració en C:

C

```c
char X = 'C';
int  Y = -1;
```

a) Traduïu la declaració a llenguatge assemblador RISC-V.
b) Especifiqueu el valor hexadecimal de `t0` després d'executar el Codi 2.13:

Codi 2.13: Codi de l'apartat b).

RV32I

```
.text
        la      t0, X
        la      t1, Y
        lb      t0, 0(t0)
        lw      t1, 0(t1)
        add     t0, t0, t1
```

2.6 Format de les instruccions

Vegeu Secció 2.2.

Problema 2.16. Convertiu les instruccions d'assemblador RISC-V següents a llenguatge màquina (format binari i hexadecimal):

Assemblador RISC-V	Binari	Hexadecimal
`add  t4, t3, t5`		
`addi t6, t5, 25`		
`lw   t3, 0(t2)`		
`sw   t0, 0(t1)`		

Problema 2.17. Desassembleu els valors hexadecimals següents a les instruccions RISC-V corresponents:

a) `0x00B50633`
b) `0xFFC50413`
c) `0x0062A023`
d) `0x0062A2B3` (instrucció `slt`, presentada al Tema 3: Traducció de programes; vegeu Secció 3.3.1)

Problema 2.18. S'estudien dues modificacions possibles a l'arquitectura RISC-V:

1. Passar de 32 registres a 8 registres.
2. Passar de 12 bits per a l'operand immediat a 6 bits.

Per a cadascuna d'elles contesteu:

a) Quin impacte tindria en la mida total d'una instrucció de tipus R?
b) I en la mida total d'una instrucció de tipus I?

2.7 Punters

Vegeu Secció 2.7.

Problema 2.19. Suposeu les declaracions en C del Codi 2.14:

Codi 2.14: Dos punters globals i dos de locals.

C

```c
int *m1, *m2;   // m1 i m2 són punters a int (en memòria)
void main() {
    int *r1, *r2;   // r1 i r2 són punters a int (en registre)
    ...
}
```

Suposant que `r1` i `r2` ocupen els registres `t1` i `t2`, traduïu a assemblador RISC-V les sentències en C següents, pertanyents a `main`:

a) `r1 = r2;`
b) `*r1 = *r2;`
c) `m1 = m2;`
d) `*m1 = *m2;`

Problema 2.20. Donada la declaració de dades global en C:

C

```c
int *pdada;
```

Traduïu a una única sentència en C el conjunt d'instruccions de cada apartat:

a)

RV32I

```
        la      t0, pdada
        lw      t0, 0(t0)
        lw      t1, 0(t0)
        addi    t1, t1, 4
        sw      t1, 0(t0)
```

b)

RV32I

```
        la      t0, pdada
        lw      t1, 0(t0)
        addi    t1, t1, 4
        sw      t1, 0(t0)
```

c)

RV32I

```
        la      t0, pdada
        lw      t0, 0(t0)
        lw      t1, 0(t0)
        addi    t0, t0, 4
        sw      t1, 0(t0)
```

Problema 2.21. Donades les declaracions en C:

C

```c
int vec[6] = {2, 4, 0, 6, 8, 0};
int *punter;
```

Traduïu a una única sentència en C el conjunt d'instruccions de cada apartat:

a)

RV32I

```
        la      t0, punter
        la      t1, vec+8
        sw      t1, 0(t0)
```

b)

RV32I

```
        la      t0, punter
        lw      t1, 0(t0)
        addi    t1, t1, 4
        sw      t1, 0(t0)
```

c)

RV32I

```
        la      t1, vec
        lw      t3, 0(t1)
        lw      t4, 4(t1)
        add     t3, t3, t4
        sw      t3, 4(t1)
```

d)

RV32I

```
        la      t1, vec
        la      t0, punter
        lw      t2, 0(t0)
        lw      t3, 0(t2)
        addi    t3, t3, 1
        sw      t3, 8(t1)
```

---

e)

RV32I

```
        la      t0, punter
        lw      t2, 0(t0)
        lw      t3, 0(t2)
        addi    t3, t3, 1
        sw      t3, 8(t2)
```

Problema 2.22. Donades les declaracions de variables globals en C:

C

```
char      *punterc;
short     *punterh;
int       *punteri;
long long *punterd;
```

Traduïu a assemblador RISC-V les sentències següents:

a) `punterc++;`
b) `punteri++;`
c) `punterh++;`
d) `punterd++;`
e) `*punteri = *punteri + 5;`
f) `*punterh = *punterh + 10;`

Problema 2.23. Donades les declaracions en C:

C

```
char          a;
int           b;
long long int c;
void main() {
    char          *p;   // guardat a t0
    int           *q;   // guardat a t1
    long long int *h;   // guardat a t2
    ...
}
```

Traduïu a assemblador RISC-V les sentències en C següents, pertanyents a `main`:

a) `q = q + 1;`
b) `a = *p;`
c) `h = &c;`
d) `b = *(q + b);`
e) `*h = *(h + b);`
f) `p[*q + 10] = a;`
g) `h = &h[*p];`

Problema 2.24. Donades les declaracions en C:

C

```c
int  dada;
int *pdada;
```

Traduïu a assemblador RISC-V les sentències en C següents:

a) `pdada = &dada;`
b) `*pdada = *pdada + 1;`
c) `pdada = pdada + 1;`
d) `dada = dada - 1;`

2.8 Vectors

Vegeu Secció 2.8 i Secció 2.8.5.

Problema 2.25. Donades les declaracions en C:

C

```c
char  indx[100];    // enters d'1 byte
short meitat[100];  // enters de 2 bytes
int   val[100], vec[100];  // enters de 4 bytes
void main() {
    char c;
    int  i, j;
    ...
}
```

Traduïu les sentències en C següents, suposant que `i`, `j` i `c` ocupen els registres `t0`, `t1` i `t2`:

a) `i = val[5] + vec[10];`
b) `i = vec[10 + val[5]];`
c) `c = indx[i + val[j]];`
d) `c = indx[meitat[i]];`
e) `i = meitat[vec[i] + val[j]];`
f) `i = val[meitat[i]] + vec[j];`
g) `i = vec[indx[i]] - val[j];`
h) `i = meitat[indx[val[j]]];`

Problema 2.26. Donades les declaracions en C:

C

```c
int v[10];
void main() {
    int i, suma = 0;  // i a t0, suma a t1
    ...
}
```

a) Escriviu en assemblador RISC-V el bucle que inicialitza tots els elements de `v` a zero.
b) Escriviu el bucle que calcula la suma de tots els elements de `v` i deixa el resultat a `suma`.
c) Escriviu el bucle que copia els elements de `v` al revés en un nou vector `w` de la mateixa longitud declarat a continuació de `v`.

Problema 2.27. Donada la declaració en C:

C

```c
int vec[100];
void main() {
    int i, val;  // i a t0, val a t1
    ...
}
```

Escriviu en assemblador RISC-V un fragment de codi que cerqui el valor de `val` dins `vec` i deixi a `t2` l'índex de la primera ocurrència, o -1 si no s'hi troba.

Problema 2.28. Donades les declaracions en C:

C

```c
short v[8] = {10, 20, 30, 40, 50, 60, 70, 80};
void main() {
    short *p = v;   // p a t0
    int i;          // i a t1
    ...
}
```

a) Escriviu en assemblador RISC-V la sentència `p = v + 3;` i expliqueu de quants bytes s'incrementa el punter.
b) Escriviu el bucle que, usant el punter `p`, dobla el valor de cada element del vector.

2.9 Cadenes de caràcters

Vegeu Secció 2.9.4.

Problema 2.29. Traduïu a assemblador RISC-V el programa en C del Codi 2.15, que escriu a la cadena `minus` els caràcters ASCII en minúscules corresponents a cada element del vector `vec`:

Codi 2.15: Programa que omple la cadena `minus` a partir del vector `vec`.

C

```c
#define N 8
char         minus[N+1];
unsigned int vec[N] = {70, 85, 78, 67, 73, 79, 78, 65};

void main() {
    int i = 0;
    while (i < N) {
        minus[i] = (char)vec[i] + 'a' - 'A';
        i++;
    }
    minus[N] = 0;
}
```

Problema 2.30. Escriviu en assemblador RISC-V una funció `longitud` que, donada una cadena de caràcters acabada en `'\0'` amb adreça base a `a0`, retorni la seva longitud (sense comptar el `'\0'`) a `a0`.

Problema 2.31. Donades les declaracions en C:

C

```c
char src[50], dst[50];
```

Escriviu en assemblador RISC-V un fragment de codi que copiï el contingut de `src` a `dst`, incloent-hi el caràcter `'\0'` de final.

Tema 3: Traducció de programes

3.1 Operacions lògiques i desplaçaments

Vegeu Secció 3.1 i Secció 3.2.

Problema 3.1. Calculeu el resultat en hexadecimal de les operacions lògiques bit a bit següents amb 16 bits:

a) `0x7654 | 0x3333`
b) `0x7654 & 0x3333`
c) `0x7654 << 2`
d) `0x7654 >> 3` (desplaçament lògic)

Problema 3.2. Donada la sentència en C:

C

```c
C = A ^ B;
```

Traduïu-la a assemblador RISC-V sense utilitzar la instrucció `xor`. Recordeu que l'operació és equivalent a:

C

```c
C = (~A & B) | (A & ~B);
```

Problema 3.3. Escriviu les seqüències de codi necessàries per deixar a `t1` el resultat de les condicions lògiques següents, sense fer servir instruccions de salt. El valor final de `t1` ha de ser 0 si la condició no es compleix, o diferent de zero altrament.

a) L'enter representat per `t1` és negatiu.
b) L'enter representat per `t1` és múltiple de 4.
c) El natural representat per `t1` és tal que el seu quadrat no es pot escriure amb 32 bits.
d) El natural representat per `t1` és tal que el seu doble no es pot escriure amb 32 bits.
e) Els bits 1, 3 i 5 de `t1` valen zero.

Problema 3.4. Programeu una seqüència de no més de 6 instruccions que posi a 0 el bit i-èsim del registre `t1`, deixant la resta de bits intactes. Suposeu que i està guardat a `t2`. Podeu usar altres registres per a resultats temporals.

Problema 3.5. Feu un programa que faci els desplaçaments següents sobre un nombre de 64 bits guardat als registres `t2` (32 bits de més pes) i `t1` (32 bits de menys pes):

a) Desplaçament a l'esquerra d'1 posició.

b) Desplaçament lògic a la dreta d'1 posició.
c) Desplaçament aritmètic a la dreta d'1 posició.
d) Desplaçament lògic a l'esquerra de n posicions ($n < 32$), on n ocupa `t3`.
e) Desplaçament lògic a la dreta de n posicions ($n < 32$), on n ocupa `t3`.

Problema 3.6. Suposeu que el registre `t0` conté dos enters de 16 bits en complement a 2: un als 16 bits de més pes i un altre als 16 bits de menys pes. Escriviu una seqüència d'instruccions que extregui aquests dos nombres, els representi en 32 bits estenent el signe, i els guardi a `t1` i `t2` respectivament, sense usar cap instrucció d'accés a memòria.

Problema 3.7. Feu un programa que compti el nombre de bits actius que hi ha al registre `t1`, executant el mínim d'instruccions possible i sense fer cap lectura de memòria. El resultat s'ha de deixar al registre `t3`.

Problema 3.8. Escriviu un fragment de codi assemblador RISC-V que intercanviï tots els bits parells amb els bits senars del registre `t0`.

Problema 3.9. Representeu el valor inicial de cada bit de `t4` amb una lletra: el bit 31 és `b31`, el bit 30 és `b30`, …, el bit 0 és `b0`. Cada lletra denota un valor binari desconegut (0 o 1).

a) Escriviu les instruccions necessàries perquè el bit de més pes passi a ser el de menys pes i la resta de bits es desplacin una posició cap a l'esquerra.
b) Seguint la mateixa notació, escriviu el contingut del registre `t4` després d'executar les instruccions del Codi 3.1:

Codi 3.1: Codi de l'apartat b).

`RV32I`

```
        li      t1, 16
        mv      t2, t4
bucle:
        srli    t4, t4, 1
        add     t2, t2, t2
        addi    t1, t1, -1
        bne     t1, zero, bucle
        or      t4, t4, t2
```

3.2 Sentències condicionals i bucles

Vegeu Secció 3.6 i Secció 3.7.

Problema 3.10. Traduïu a assemblador RISC-V el programa en C del Codi 3.2, suposant que `a`, `b` i `c` ja s'han inicialitzat als registres `t1`, `t2` i `t3`:

Codi 3.2: Sentències `if` imbricades.

`C`

```c
void main() {
    int a, b, c, i;  // i s'emmagatzema a t0
    ...
    if (a > 0) {
        i = b;
        if (((i - 4) < a) && (i > 0))
            b = b + c;
        else if (c > 0)
            b = b - c;
    } else {
        a = a + b;
    }
}
```

Problema 3.11. Traduïu a assemblador RISC-V la sentència `if` del programa en C del Codi 3.3, que converteix un dígit hexadecimal (representat per un caràcter ASCII guardat a la variable global `c`) al seu valor binari equivalent i l'escriu a la variable global `num` (enter de 8 bits). Suposeu que el valor de `c` és un dels caràcters `{'0'..'9', 'A'..'F', 'a'..'f'}`.

Codi 3.3: Conversió d'un dígit hexadecimal en ASCII al seu valor.

C

```c
char c, num;
void main() {
    ...
    if ((c >= 'A') && (c <= 'F'))
        num = c - 'A' + 10;
    else if ((c >= 'a') && (c <= 'f'))
        num = c - 'a' + 10;
    else
        num = c - '0';
}
```

Problema 3.12. Traduïu a assemblador RISC-V la sentència `switch` del Codi 3.4. La variable `op` ocupa el registre `t0` i les variables `a`, `b`, `res` ocupen `t1`, `t2`, `t3`:

Codi 3.4: Sentència `switch` amb quatre casos i `default`.

C

```c
int op, a, b, res;
...
switch (op) {
    case 0:  res = a + b;  break;
    case 1:  res = a - b;  break;
    case 2:  res = a & b;  break;
    case 3:  res = a | b;  break;
    default: res = 0;
}
```

Proposeu dues implementacions: una amb salts encadenats i una altra amb taula de salts (*jump table*). Quan és preferible cada enfocament?

Problema 3.13. Donat el programa en C:

C

```c
void main() {
    int a = 23;  // guardat a t0
    int b = 12;  // guardat a t1
    int i;       // guardat a t2
    for (i = 0; i < 10; i++)
        a += b;
}
```

a) Traduïu-lo a assemblador RISC-V.
b) Quin és el nombre total d'instruccions que s'executen?
c) Suposant un processador a 2 GHz on totes les instruccions tenen CPI = 2, si el nombre d'instruccions executades fos 10, quants segons trigaria a executar-se el programa?
d) En un processador a 2 GHz, si el nombre d'instruccions executades fos 10, com a l'apartat c), i el programa trigués 5×10^{-9} s, quin seria el CPI del programa?

Problema 3.14. Donades dues variables enteres `a` i `b` (amb `b ≥ 0`), ja inicialitzades als registres `t1` i `t2`:

a) Programeu en C una seqüència d'instruccions que multipliqui `a * b` usant un bucle `while` i operacions de suma. El resultat s'ha de deixar a `c`, ubicada al registre `t3`.
b) Traduïu-la a assemblador RISC-V.
c) Quin és el nombre màxim d'instruccions RISC-V que poden arribar a executar-se?
d) En un processador a 2 GHz on totes les instruccions usen 2 cicles de rellotge, quants segons trigaria a executar-se el programa?
e) Si en comptes de sumes s'usa una única instrucció de multiplicació que utilitza 10 cicles de rellotge, quants segons trigaria a executar-se? Quin seria el CPI del programa?

Problema 3.15. Traduïu a assemblador RISC-V el programa en C del Codi 3.5:

Codi 3.5: Bucle `while` que escriu el vector `D`.

C

```c
int D[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
void main() {
    int a = 0;   // guardat a t0
    int b = 12;  // guardat a t1
    while (a < 10) {
        D[a] = b + a;
        a += 1;
    }
}
```

Problema 3.16. Traduïu a assemblador RISC-V el programa en C del Codi 3.6, suposant que les variables globals ja s'han inicialitzat:

Codi 3.6: Suma i recompte dels elements de `V` més grans que 5.

C

```c
#define N 30
int V[N], suma, elems;
void main() {
    int i;
    ...
    i = 0;
    elems = 0;
    suma = 0;
    while (i < N) {
        if (V[i] > 5) {
            suma = suma + V[i];
            elems++;
        }
        i++;
    }
}
```

Problema 3.17. Donada la variable global (on `N` és una constant):

`C`

```c
int v[N];
```

Calculeu el temps d'execució en cicles de rellotge de cada versió del programa que suma els elements de `v`, suposant que:

- Els accessos a memòria triguen 4 cicles.
- Els salts que no salten triguen 1 cicle.
- Els salts que salten triguen 2 cicles.
- La resta d'instruccions triguen 1 cicle.

a)

`RV32I`

```
        li      t0, 0       # suma
        li      t1, 0       # i
        li      t2, 0       # desplaçament
bucle:
        li      t3, N
        bge     t1, t3, fi
        la      t4, v
        add     t4, t4, t2
        lw      t4, 0(t4)
        add     t0, t0, t4
        addi    t2, t2, 4
        addi    t1, t1, 1
        j       bucle
fi:
```

---

b)

`RV32I`

```
        li      t0, 0        # suma
        li      t1, 0        # i
bucle:
        li      t3, N
        bge     t1, t3, fi
        la      t3, v
        slli    t2, t1, 2
        add     t2, t3, t2
        lw      t2, 0(t2)
        add     t0, t0, t2
        addi    t1, t1, 1
        j       bucle
fi:
```

---

c)

RV32I

```
        li      t0, 0        # suma
        li      t1, 0        # i
bucle:
        la      t3, v
        slli    t2, t1, 2
        add     t2, t2, t3
        lw      t2, 0(t2)
        add     t0, t0, t2
        addi    t1, t1, 1
        li      t3, N
        blt     t1, t3, bucle
```

---

d)

RV32I

```
        li      t0, 0        # suma
        li      t1, N-1      # i
bucle:
        la      t3, v
        slli    t2, t1, 2
        add     t2, t3, t2
        lw      t2, 0(t2)
        add     t0, t0, t2
        addi    t1, t1, -1
        bge     t1, zero, bucle
```

---

e)

RV32I

```
        li      t0, 0        # suma
        li      t1, N        # i
        la      t2, v
bucle:
        lw      t3, 0(t2)
        add     t0, t0, t3
        addi    t2, t2, 4
        addi    t1, t1, -1
        bne     t1, zero, bucle
```

f) Programeu una nova versió de l'apartat e) que en disminueixi el temps d'execució.

Problema 3.18. Donada la declaració en C (on `N` és una constant):

C

```c
int V[2*N];
```

Escriviu un fragment de codi assemblador RISC-V que intercanviï cada element d'índex parell amb l'element següent d'índex senar.

Problema 3.19. Donades les declaracions:

RV32I

```
.data
v1:     .byte   '0', '0', '4', '1', '9'
v2:     .byte   '8', '9', '7', '2', '3'
v3:     .byte   0,   0,   0,   0,   0,  0
```

Cada vector conté caràcters ASCII que representen dígits numèrics. En conjunt, cada vector representa un natural decimal amb les unitats a l'última posició. Escriviu un programa que sumi els naturals representats per `v1` i `v2` i deixi el resultat a `v3`, que té un dígit de més per poder representar qualsevol suma sense desbordament. Feu servir l'algorisme típic de la suma dígit a dígit.

Problema 3.20. El vector d'enters `Array` està ubicat a l'adreça `0x10010000` de memòria i s'ha declarat així:

RV32I

```
.data
Array:  .word   2, 4, 6, 1
```

Feu un programa en assemblador RISC-V que ordeni els elements del vector de menor a major, deixant el valor més petit a la posició de memòria més baixa.

3.3 Subrutines: introducció

Vegeu Secció 3.9.

Problema 3.21. Escriviu en C i en assemblador RISC-V una funció que, donats dos enters de 32 bits passats per valor, retorni el mínim dels dos.

Problema 3.22. Traduïu a assemblador RISC-V la subrutina en C del Codi 3.7:

Codi 3.7: Funció `strlen`.

C

```c
int strlen(char s[]) {
    int i = 0;
    while (s[i] != '\0')
        i++;
    return i;
}
```

Problema 3.23. Donat el fragment de programa en C:

C

```c
void examen(char vi[], char vo[]) {
    int i = 0;
    while (vi[i] != '\0') {
        if (vi[i] != ' ')
            vo[i] = vi[i] - 32;
        else
            vo[i] = vi[i];
        i++;
    }
    vo[i] = '\0';
}
void main() {
    examen(vect1, vect2);
}
```

Sabent que `vect1` i `vect2` són variables globals:

a) Traduïu a assemblador RISC-V la funció `main`.
b) Traduïu a assemblador RISC-V la subrutina `examen`.

Problema 3.24. Donades les declaracions en C:

C

```c
int proc(int vector[], int *x1);

char exam(int v[], int a, int b) {
    int i, *p;
    ...
}
```

a) Traduïu a assemblador RISC-V el fragment de codi del Codi 3.8, que pertany a `exam`, suposant que `i` i `p` ocupen `t0` i `t1`:

Codi 3.8: Fragment de l'apartat a): una crida a `proc`.

C

```c
p = &v[a + 5];
i = proc(v, p);
```

b) Traduïu a assemblador RISC-V la sentència condicional del Codi 3.9, que pertany a `exam`, suposant que `i` ocupa `t0`:

Codi 3.9: Sentència condicional de l'apartat b).

C

```c
if ((i > 0) && (i < 10))
    i = 10;
else
    i = 0;
```

c) Traduïu a assemblador RISC-V el fragment de codi del Codi 3.10, que pertany a `exam`, suposant que `i` i `p` ocupen `t0` i `t1`:

Codi 3.10: Bucle de l'apartat c).

C

```c
p = v + b;
for (i = 0; i < 10; i++) {
    *p = 0;
    p++;
}
```

Problema 3.25. Donades les funcions en C:

C

```c
void swap(int *a, int *b) {
    int temp;
    temp = *a;
    *a   = *b;
    *b   = temp;
}

void examen(int vector[], char c, int j) {
    swap(&vector[j + 1], &vector[j]);
}
```

a) Traduïu a assemblador RISC-V la subrutina `swap`.
b) Traduïu a assemblador RISC-V la subrutina `examen`.

Problema 3.26. Sabent que la funció `f` té la capçalera en C:

C

```c
int f(int n);
```

i que la seva traducció a assemblador RISC-V és el Codi 3.11:

Codi 3.11: Traducció de la funció `f`.

RV32I

```
f:
        mv      t0, a0      # t0 = n
        li      a0, 0       # a0 = acumulador
buc:
        andi    t1, t0, 1
        add     a0, a0, t1
        srli    t0, t0, 1
        bne     t0, zero, buc
        li      t1, 32
        sub     a0, t1, a0
        ret
```

a) Quin valor retorna `f` per a `n = 0x6789ABCD`?
b) Expliqueu amb paraules què calcula la funció `f`.

Problema 3.27. Traduïu a assemblador RISC-V la funció recursiva del Codi 3.12:

Codi 3.12: Funció recursiva `pellnumber`.

C

```c
int pellnumber(int n) {
    if (n < 2)
        return n;
    else
        return 2 * pellnumber(n - 1) + pellnumber(n - 2);
}
```

Problema 3.28. Traduïu a assemblador RISC-V la subrutina en C del Codi 3.13:

Codi 3.13: Subrutina recursiva `maxv`.

C

```c
int maxv(int vector[], int length) {
    int temp;
    if (length == 1)
        temp = vector[0];
    else {
        temp = maxv(vector, length - 1);
        if (vector[length - 1] > temp)
            temp = vector[length - 1];
    }
    return temp;
}
```

Problema 3.29. Traduïu a assemblador RISC-V la funció `func1`:

C

```c
unsigned int func1(unsigned int v, unsigned int *p) {
    if (v == 0)
        *p = 0;
    else
        *p = func1(func1(v - 1, p), p + 1) - 1;
    return *p;
}
```

Problema 3.30. El repertori de RISC-V inclou la instrucció `jalr` per cridar subrutines i per a salts indirectes. Suposeu que s'executa el fragment de codi RISC-V del Codi 3.14:

Codi 3.14: Fragment amb `la`, `addi` i `jalr`.

RV32I

```
        la      ra, etiq
        addi    ra, ra, -4
etiq:   jalr    ra, 0(ra)
```

Quina és la resposta correcta?

a) La instrucció `jalr` s'executa 1 vegada.
b) La instrucció `jalr` s'executa 2 vegades.
c) La instrucció `jalr` s'executa 3 vegades.
d) La instrucció `jalr` s'executa infinites vegades (el programa entra en un bucle infinit).
e) Sense saber el contingut d'altres posicions de memòria no es pot saber quantes vegades s'executa.

Problema 3.31. Donades les declaracions en C:

C

```c
int s2(int c, long long *d, long long w[]);

int s1(int x, long long v[], int *p) {
    long long *k;
    k = &v[x];
    x = s2(*p, k, v);
    return *p + x;
}
```

Traduïu a assemblador RISC-V la funció `s1`.

3.4 Subrutines: context i bloc d'activació

Vegeu Secció 3.9.3 i Secció 3.9.4.

Problema 3.32. Donades les declaracions en C:

C

```c
int subr2(int a, char *b, char c, int d);

int subr1(char v[], int e2, int *var, char p) {
    int valor;
    valor = subr2(*var, &v[e2], p, e2);
    return valor + e2;
}
```

Traduïu a assemblador RISC-V la funció `subr1`.

Problema 3.33. Donades les declaracions de variables globals i de la funció `A`:

C

```c
char  x;
int   z;
char  w[20];

char A(char i, int k, char *v) {
    char r[10];
    char c;
    c    = v[3];
    r[k] = i;
    return c;
}
```

a) Traduïu a assemblador RISC-V la sentència:

C

```c
x = A(x, z, w);
```

b) Dibuixeu el bloc d'activació d'una crida a la funció `A`.
c) Traduïu a assemblador RISC-V la funció `A`.

Problema 3.34. Traduïu a assemblador RISC-V la funció `A`:

C

```c
int B(short *x, short y[], int z, char m);

int A(int a[], short b[], char c) {
    int   p1 = 0;
    short p2[5];
    p1         = B(b, p2, p1, 'A');
    p2[p1 + a[2]] = b[p1];
    if (c == 'z')
        p1 = p1 - 3;
    else
        p1 = 2;
    return p1;
}
```

Problema 3.35. Traduïu a assemblador RISC-V la funció `sub` (escrivint punts suspensius per a les sentències desconegudes):

C

```c
void sub(int p1[10], int p2, int p3) {
    int x1[10], x2;
    ...
    x1[p2]  = p1[x2] + p3;
    ...
    sub(x1, p2, p1[3]);
    ...
    x2 = p2 + p3;
    ...
}
```

3.5 Compilació, assemblatge, enllaçament i càrrega

Vegeu Secció 3.10.

Problema 3.36. La pseudoinstrucció `la rd, symbol` s'expandeix habitualment en dues instruccions reals. Expliqueu quines són i per què són necessàries dues instruccions. Quina és la limitació en el desplaçament que pot representar l'expansió?

Problema 3.37. Un projecte té dos fitxers font:

C

```c
// fitxer a.c
int x = 42;
int f(int n) { return n + x; }
```

C

```c
// fitxer b.c
extern int x;
int f(int n);
int main() { return f(x); }
```

a) Quines fases del flux de generació (compilació, assemblatge, enllaçament) intervenen en construir l'executable?
b) El fitxer objecte `b.o` conté una referència externa a `x` i a `f`. Qui resol aquestes referències i en quin moment?
c) Si `x` es declara `static int x = 42` a `a.c`, canvia alguna cosa? Per què?
d) La pseudoinstrucció `la t0, x` es tradueix en `auipc` + `addi` (Problema 3.36). Per què `auipc`, relatiu al PC, i no `lui`, que carrega una adreça absoluta?

Tema 4: Aritmètica entera i matrius

4.1 Suma i resta

Vegeu Secció 4.2 i Secció 1.6.

Problema 4.1. Convertiu a decimal els nombres `0x0D34` i `0xBA1D`:

a) Suposant que representen naturals.
b) Suposant que representen enters en complement a 2.
c) Suposant que representen enters en signe-magnitud.

Problema 4.2. Siguin els nombres: $A =$ `0x0D34`, $B =$ `0xDD17`, $C =$ `0xBA1D`, $D =$ `0x3617`.

a) Quant val, en hexadecimal, la suma $A + B$ si representen naturals de 16 bits?
b) Ídem per a la suma $C + D$.
c) Quant val, en hexadecimal, la suma $A + B$ si són enters en signe-magnitud de 16 bits?
d) Ídem per a la suma $C + D$.

Problema 4.3. Siguin els nombres: $A =$ `0xBA7C`, $B =$ `0x241A`, $C =$ `0xAADF`, $D =$ `0x47BE`.

a) Quant val, en hexadecimal, la resta $A - B$ si representen naturals de 16 bits?
b) Ídem per a la resta $C - D$.
c) Quant val, en hexadecimal, la resta $A - B$ si són enters en signe-magnitud de 16 bits?
d) Ídem per a la resta $C - D$.

Problema 4.4. Per a les operacions següents amb naturals de 8 bits, codifiqueu els operands en binari, realitzeu l'operació i indiqueu si s'ha produït ròssec (*carry*) o, en el cas de la resta, préstec (*borrow*). Convertiu el resultat a decimal:

a) $200 + 50$
b) $25 + 34$
c) $254 + 3$
d) $128 - 20$

Problema 4.5. Per a les operacions següents amb enters en Ca2 de 8 bits, codifiqueu els operands en binari, realitzeu l'operació i indiqueu si s'ha produït desbordament (*overflow*). Convertiu el resultat a decimal:

a) $(-25) + (-34)$
b) $(-25) - (-34)$
c) $(+127) - (-20)$
d) $(-128) - (-20)$

Vegeu Secció 4.2.1.

Problema 4.6. Donades dues variables naturals de 32 bits emmagatzemades a `t1` i `t2`, escriviu un fragment de codi que calculi la seva suma (`t0 = t1 + t2`) i determini si s'ha produït ròssec, desant 1 a `t3` en cas afirmatiu i 0 altrament. El fragment no ha de contenir cap instrucció de salt.

Indicació: si es produeix ròssec, el resultat `t0` és menor que qualsevol dels dos sumands.

Problema 4.7. Donades dues variables enteres de 32 bits emmagatzemades a `t1` i `t2`, escriviu un fragment de codi que calculi la seva suma (`t0 = t1 + t2`) i determini si s'ha produït desbordament (*overflow*). El fragment no ha de contenir cap instrucció de salt.

Indicació: hi ha desbordament si i només si els dos sumands tenen el mateix signe i el resultat té signe oposat. A RISC-V, la detecció es fa combinant operacions lògiques bit a bit i desplaçaments sobre els bits de signe.

Vegeu Secció 4.2.3 i Exemple 4.1.

Problema 4.8. Donada la declaració en C:

C

```c
long long x, y;
```

Traduïu a assemblador RISC-V les sentències següents (cada variable ocupa dos registres consecutius: part alta i part baixa):

a) `x = x + y;`
b) `x = x - y;`
c) `if (x > y) x = 1;`

4.2 Multiplicació

Vegeu Secció 4.3.

Problema 4.9. Donat el diagrama del multiplicador seqüencial de naturals de 32 bits estudiat a la teoria, que calcula el producte de 64 bits, completeu l'algorisme iteratiu que en descriu el funcionament cicle a cicle, indicant el valor inicial dels registres i les operacions de cada iteració.

Vegeu Algorisme 4.1.

Problema 4.10. Seguint el circuit multiplicador seqüencial, descriviu els passos per a la multiplicació dels naturals de 6 bits x (multiplicand) i y (multiplicador), calculant en cada pas el valor dels registres P, MD i MR en binari:

a) $x =$ `101000`, $y =$ `010011`.
b) $x =$ `110110`, $y =$ `000100`.

Vegeu Algorisme 4.1 i Exemple 4.3.

Problema 4.11. Seguint el mateix algorisme del circuit multiplicador, i sense usar cap instrucció de multiplicació, escriviu un fragment de codi en assemblador RISC-V que calculi el producte de dos naturals de 32 bits guardats als registres `t0` i `t2`. El resultat de 64 bits s'ha de guardar a `t3` (part baixa) i `t4` (part alta).

Vegeu Algorisme 4.1.

Problema 4.12. Siguin `x`, `y` i `z` variables de 32 bits emmagatzemades a `t1`, `t2` i `t3`. Escriviu un fragment de codi en RISC-V que calculi `z = x * y` i escrigui a `t0` el valor 0 si el resultat és correcte o 1 si s'ha produït desbordament. El fragment no ha de fer servir cap instrucció de salt. Considereu:

a) `unsigned int x, y, z;`
b) `int x, y, z;`

Vegeu Secció 4.3.4.

Problema 4.13. Completeu el contingut que han de tenir els registres operands de les instruccions següents perquè siguin coherents després de la seva execució. Considereu cada instrucció de manera independent:

a) `mul t3, t1, t2` — el resultat als 32 bits de menys pes és `0xFFFFFFFA`; `t1 = 0x00000002`; calculeu el valor de `t2`.
b) `div t3, t1, t2` (divisió entera, truncament cap a zero) — `t1 = 0x00000005`, `t2 = 0xFFFFFFFE`; calculeu el quocient a `t3` i el residu.
c) En quins casos concrets pot produir-se desbordament en una divisió entera?

4.3 Matrius

Vegeu Secció 4.5.

Problema 4.14. Donades les declaracions en C, on `NF` i `NC` són constants:

C

```c
int mat[NF][NC];
int f() {
    int i, j;
    ...
}
```

Escriviu el codi RISC-V per calcular les adreces dels elements següents, fent servir el nombre mínim d'instruccions i deixant el resultat a `a0`. Suposeu que `i` i `j` estan emmagatzemades a `t0` i `t1`:

a) `mat[3][11]`
b) `mat[i][11]`
c) `mat[3][j]`
d) `mat[i][j]`
e) `mat[i+5][j-1]`
f) `mat[i*10+4][6]`

Vegeu Equació 4.3.

Problema 4.15. Traduïu a assemblador RISC-V la funció del Codi 4.1, on `N` és una constant:

Codi 4.1: Funció `func` sobre una matriu de `N` columnes.

C

```c
void func(int mat[][N], int i, int j) {
    if (i > j)
        mat[i][j] = mat[j][i];
}
```

Problema 4.16. Donada la declaració de variable global (on `N` és una constant):

C

```c
int M[N][N];
```

Traduïu a assemblador RISC-V el fragment de codi en C del Codi 4.2, usant el nombre mínim d'instruccions, suposant que `i` i `suma` ocupen els registres `t0` i `t1`:

Codi 4.2: Sentència amb dos accessos a la matriu `M`.

C

```c
suma += M[i][i+1] - M[i+1][i];
```

Problema 4.17. Sigui la declaració en C:

C

```c
char mat[6][10];
```

Si el registre `t0` és un punter que apunta a l'element `mat[i][j]`, és possible modificar `t0` amb una sola instrucció perquè apunti a `mat[i-3][j+5]`? Si la resposta és sí, escriviu-la.

Problema 4.18. Sigui la declaració de la variable global:

C

```c
long long mat[10][50];
```

Suposant que inicialment `t0` conté l'adreça de l'element `mat[4][45]`, a quin element de la matriu apunta `t0` després d'executar la instrucció del Codi 4.3?

Codi 4.3: Instrucció `addi` sobre `t0`.

RV32I

```
        addi    t0, t0, 1384
```

Problema 4.19. Donades les declaracions en C:

C

```
void f(int n) {
    int j;
    int b[8][10];
    ...
    // Aquí va la sentència incògnita
    ...
}
```

El fragment de codi RISC-V del Codi 4.4 correspon a la traducció de la sentència incògnita. Determineu quina és aquesta sentència i expresseu-la en C, tenint en compte que `t0` conté la variable `j`:

Codi 4.4: Traducció de la sentència incògnita.

RV32IM

```
        li      t0, 0
bucle:
        li      t1, 10
        bge     t0, t1, final
        li      t1, 10
        mul     t1, t1, a0
        add     t1, t1, t0
        slli    t1, t1, 2
        add     t1, t1, sp
        sw      zero, 0(t1)
        addi    t0, t0, 1
        j       bucle
final:
```

Problema 4.20. Donades les declaracions en C:

C

```
int mati[5][4];

void func(int veci[4]) {
    int j;
    for (j = 0; j < 4; j++)
        veci[j] = j;
}

void main() {
    int *p = &mati[0][0];
    int *fi = &mati[5][0];
    while (p != fi) {
        func(p);
        p += 4;
    }
}
```

a) Traduïu a RISC-V la subrutina `func` amb accés seqüencial al vector `veci`, suposant que `j` ocupa `t0`.
b) Traduïu a RISC-V la subrutina `main` amb accés seqüencial a la matriu `mati`.

4.4 Accés seqüencial

Vegeu Secció 4.6.

Problema 4.21. Donades les declaracions en C (on `N` és una constant):

C

```c
void func(int A[N][N]) {
    int i, j, suma = 0;
    ...
}
```

Traduïu a RISC-V les sentències següents usant accés seqüencial per als accessos a `A`, suposant que `i`, `j` i `suma` ocupen `t0`, `t1` i `t2`:

a)

C

```c
for (i = 0; i < N; i++)
    suma += A[3][i];
```

b)

C

```c
for (i = 0; i < N; i++)
    suma += A[i][4];
```

c)

C

```c
for (i = 0; i < N; i++)
    suma += A[i][i];
```

d)

C

```c
for (i = 0; i < N; i += 3)
    suma += A[i][N-1-i];
```

Problema 4.22. Donades les declaracions en C:

C

```c
#define N 100
char      vecchar[N];
int       matint[N][N];
long long matlong[N][N];

void func() {
    int i;
    ...
}
```

Traduïu a RISC-V les sentències següents usant accés seqüencial, suposant que `i` ocupa `t0`:

a)

C

```c
for (i = 1; i < N; i += 2)
    vecchar[i] = N - i;
```

---

b)

C

```c
for (i = 0; i < N; i++)
    matint[i][0] = 0;
```

---

c)

C

```c
i = 4;
while (matint[3][i] != 0) {
    matint[3][i]--;
    i += 3;
}
```

---

d)

C

```c
for (i = N-1; i >= 0; i--)
    matlong[i][4] = i;
```

Problema 4.23. Siguin les declaracions en C:

C

```c
int   z[5][3];
int   vector[4];

void func(int i, int j) {
    int k;
    ...
}
```

---

a) Traduïu a RISC-V la sentència del Codi 4.5, que pertany a `func`:

Codi 4.5: Sentència de l'apartat a).

C

```c
vector[i] = z[4][j];
```

b) El Codi 4.6, en assemblador RISC-V, s'ha obtingut en traduir el bucle del Codi 4.7 implementant accés seqüencial. Completeu les quatre constants inicials perquè la traducció sigui correcta:

Codi 4.6: Traducció de l'apartat b), amb les quatre constants per completar.

RV32I

```
.eqv    offset_vector, ?
.eqv    offset_z,      ?
.eqv    max_dist,      ?
.eqv    stride,        ?

func:
        la      t2, vector + offset_vector
        la      t3, z + offset_z
        addi    t6, t3, max_dist
for:
        beq     t3, t6, fifor
        lw      t4, 0(t2)
        lw      t5, 0(t3)
        add     t4, t4, t5
        sw      t4, 0(t2)
        addi    t3, t3, stride
        j       for
fifor:
        ret
```

Codi 4.7: Bucle de l'apartat b), en C.

C

```c
for (k = 0; k < 5; k++)
    vector[2] += z[k][2];
```

Problema 4.24. Donades les declaracions en C:

C

```c
#define N 10
int mat[N][N];
```

Traduïu a assemblador RISC-V el codi del Codi 4.8 amb accés seqüencial i un sol punter:

Codi 4.8: Bucle `do-while` sobre la diagonal de `mat`.

C

```
i = 0;
do {
    mat[i][i]   = mat[i][i+1] - mat[i+1][i];
    i += 2;
} while (i < N);
```

Problema 4.25. Donades les declaracions en C:

C

```
void examen(short p1[100], int p2) {
    int    i;
    short *p;
    ...
}
```

a) Traduïu a C, amb una única sentència, el fragment d'assemblador RISC-V del Codi 4.9, sabent que forma part del cos d'`examen`:

Codi 4.9: Fragment de l'apartat a).

RV32I

```
        addi    t2, a1, 3
        slli    t2, t2, 1
        add     t2, a0, t2
        sh      zero, 0(t2)
```

b) El fragment incomplet de codi C del Codi 4.10 i el de RISC-V del Codi 4.11 pertanyen al cos d'`examen` i inicialitzen tots els elements de `p1` a zero amb accés seqüencial. Completeu les tres sentències que falten en C i les línies corresponents en assemblador, suposant que `i` i `p` es guarden a `t0` i `t1`:

Codi 4.10: Fragment incomplet en C de l'apartat b).

C

```
p = _______ ;
for (i = 0; i < 100; i++) {
    _______ ;
    _______ ;
}
```

Codi 4.11: Fragment incomplet en RISC-V de l'apartat b).

RV32I

```
        li      t0, 0
        li      t2, 100
bucle:
        bge     t0, t2, fibucle
        ____________________
        ____________________
        addi    t0, t0, 1
        j       bucle
fibucle:
```

Problema 4.26. La funció `func` recorre la matriu `matriu` amb un punter. Suposeu que `N` i `M` són constants, amb $N \leq M$:

Codi 4.12: Funció `func`, que recorre `matriu` amb un punter.

C

```c
int matriu[N][M];

void func() {
    int  i, *p;
    p = &matriu[0][0];
    for (i = 0; i < N; i++) {
        *p  = i;
        p   = p + M + 1;
    }
}
```

a) Enumereu els elements de la matriu que s'escriuen i indiqueu quin valor s'assigna a cada un.
b) Traduïu a RISC-V el bucle del Codi 4.12 usant accés seqüencial, suposant que `i` i `p` ocupen `t0` i `t1`.

Problema 4.27. La funció `fila_dispersa` escriu els elements de la fila `i` d'una matriu dispersa, representada pels vectors `A`, `IA` i `JA`, a les posicions corresponents de la matriu global `mat`:

Codi 4.13: Funció `fila_dispersa`.

C

```c
int mat[N][M];

void fila_dispersa(int *A, int *IA, int *JA, int i) {
    int k;
    for (k = IA[i]; k < IA[i+1]; k++)
        mat[i][JA[k]] = A[k];
}
```

a) Dels accessos a memòria que fa el bucle del Codi 4.13 (`IA[i]`, `IA[i+1]`, `JA[k]`, `mat[i][JA[k]]`, `A[k]`), quins es poden traduir amb accés seqüencial i quins només amb accés aleatori?
b) Traduïu la funció `fila_dispersa` usant accés seqüencial per als recorreguts que ho permetin, suposant que `k` es guarda a `t0`.

Problema 4.28. Donada la versió inicial del bucle del Codi 4.14 (on `N` és una constant i `v` és un vector global d'enters):

Codi 4.14: Versió inicial del bucle, sense optimitzar.

RV32I

```
        li      t0, 0
bucle:
        li      t1, N
        bge     t0, t1, fibucle
        la      t2, v
        slli    t3, t0, 2
        add     t2, t2, t3
```

```
        lw      t3, 0(t2)
        add     t4, t4, t3
        addi    t0, t0, 1
        j       bucle
fibucle:
```

a) Identifiqueu quantes instruccions s'executen per iteració i calculeu el temps total en cicles per a N iteracions (useu el model de costos: memòria = 4 cicles, salt pres = 2, salt no pres = 1, resta = 1).
b) Apliqueu l'optimització de condició al final del bucle (`do-while`). Escriviu la versió optimitzada i calculeu de nou el temps total.
c) Quina reducció percentual del temps s'obté per a $N = 100$?

Vegeu Secció 4.6.2.

Problema 4.29. Donada la declaració (on `N` és una constant):

C

```
int v[N], w[N];
```

El bucle `for (i = 0; i < N; i++) w[i] = v[i] * 2;` es pot optimitzar eliminant la variable d'inducció `i`.

a) Escriviu la versió assemblador RISC-V **sense** cap optimització (usant `i` explícitament, amb la condició avaluada a l'inici del bucle).
b) Apliqueu l'eliminació de la variable d'inducció: useu dos punters que avancin directament i elimineu tots els càlculs d'adreça, mantenint la condició avaluada a l'inici. Escriviu la versió optimitzada.
c) Combineu l'eliminació de la variable d'inducció (b) amb l'avaluació de la condició al final. Quantes instruccions té el cos del bucle resultant?

Vegeu Secció 4.6.1, Secció 4.6.2 i Secció 4.6.3.

4.5 Divisió

Vegeu Secció 4.4.

Problema 4.30. Donat el diagrama del divisor seqüencial de naturals de 32 bits amb restauració estudiat a la teoria, que calcula el quocient i el residu, completeu l'algorisme iteratiu que en descriu el funcionament cicle a cicle.

Vegeu Algorisme 4.2.

Problema 4.31. Seguint el circuit divisor seqüencial amb restauració, descriviu els passos per a la divisió dels naturals de 6 bits x (dividend) entre y (divisor), calculant en cada pas el valor dels registres Q, D i R en binari:

a) $x =$ `101000`, $y =$ `010011`.
b) $x =$ `010101`, $y =$ `100100`.

Vegeu Algorisme 4.2 i Exemple 4.7.

Problema 4.32. Seguint el mateix algorisme del circuit divisor, i sense usar cap instrucció de multiplicació ni divisió, escriviu un fragment de codi en assemblador RISC-V que calculi el quocient i el residu de la divisió de dos naturals de 32 bits: el dividend al registre `t3` i el divisor a `t1`. El quocient s'ha de guardar a `t2` i el residu a `t3`.

Vegeu Algorisme 4.2.

Tema 5: Coma flotant

5.1 Representació IEEE 754

Vegeu Secció 5.1.

Problema 5.1. En una arquitectura de von Neumann, els grups de bits no tenen un significat específic per si mateixos: el seu significat depèn totalment de com es facin servir. Suposeu els valors `A = 0x407302B3` i `B = 0xF9C40513`. Contesteu les preguntes següents fent diverses suposicions sobre `A` i `B`:

a) Si són naturals, quin és el seu valor en decimal?
b) Si són enters en complement a 2, quin és el seu valor en decimal?
c) Si són nombres en coma flotant de precisió simple, quin és el seu valor en decimal?
d) Si són instruccions RISC-V, quina instrucció representen en assemblador?

Problema 5.2. Convertiu a decimal els valors següents suposant que codifiquen nombres en format IEEE 754 de precisió simple:

a) `0x3FB08000`
b) `0x80000000`
c) `0xB0FC0000`
d) `0xC2968000`

Problema 5.3. La taula següent conté una llista de nombres binaris en format IEEE 754 de precisió simple. Marqueu amb una X la casella corresponent al tipus de valor de cada un, d'acord amb la notació: NRM = normalitzat; DNRM = subnormal; 0 = zero; INF = infinit; NaN = *Not a Number*.

Signe	Exponent	Fracció	NRM	DNRM	0	INF	NaN
0	`0000 0000`	`110 0010 0000 1110 1110 1011`					
0	`0000 0000`	`000 0000 0000 0000 0000 0000`					
0	`0010 0100`	`000 0000 0000 0000 0000 0000`					
1	`1111 1111`	`000 0000 0000 0000 0000 0000`					
0	`0010 0100`	`110 0010 0000 1110 1110 1011`					
1	`0000 0000`	`000 0000 0000 0000 0000 0000`					
0	`1111 1111`	`101 0001 0001 0000 1001 0100`					

Problema 5.4. En el format de coma flotant IEEE 754 de precisió simple:

a) Quin és el nombre positiu més gran representable? Expresseu el resultat en notació exponencial i en hexadecimal.
b) Quin és el nombre positiu no nul més petit representable en format normalitzat? Expresseu el resultat en notació exponencial i en hexadecimal.
c) Quin és el nombre positiu no nul més petit representable en format subnormal? Expresseu el resultat en notació exponencial i en hexadecimal.

Problema 5.5. Convertiu els nombres decimals següents al format de coma flotant IEEE 754 de precisió simple, expressant el resultat en hexadecimal. Determineu en cada cas si es comet error per pèrdua de precisió i, en cas afirmatiu, calculeu-ne el valor en decimal:

a) 340
b) 44,4
c) $-255{,}125$
d) $-10{,}75$
e) 6,5

Problema 5.6. Donats els nombres decimals $A = -1609{,}5$ i $B = -938{,}8125$:

a) Convertiu-los al format IEEE 754 de precisió simple, expressant el resultat en hexadecimal, i calculeu l'error comès per pèrdua de precisió expressat en decimal.
b) Convertiu-los al format IEEE 754 de precisió doble, expressant el resultat en hexadecimal, i calculeu l'error comès per pèrdua de precisió expressat en decimal.

Problema 5.7. Els processadors gràfics usen un format de coma flotant de 16 bits anomenat *half*, similar al de precisió simple, excepte que l'exponent té 5 bits (esbiaixat, biaix 15) i la fracció en té 10 (amb bit ocult, com a precisió simple: la mantissa en té 11). Convertiu els nombres decimals següents a la representació *half* en hexadecimal:

a) $A = -1{,}278 \times 10^{3}$, $B = -3{,}90625 \times 10^{-1}$
b) $A = 2{,}3109375 \times 10^{1}$, $B = 6{,}3916015625 \times 10^{-1}$
c) $A = 6{,}18 \times 10^{2}$, $B = 5{,}79375 \times 10^{1}$
d) $A = -1{,}6360 \times 10^{4}$, $B = 1{,}6360 \times 10^{4}$, $C = 1{,}0$

Problema 5.8. Convertiu els nombres decimals següents al format *half* de 16 bits (vegeu Problema 5.7) i expresseu el resultat en hexadecimal:

a) $A = 2{,}865625 \times 10^{1}$, $B = 4{,}140625 \times 10^{-1}$, $C = 1{,}2140625 \times 10^{1}$
b) $A = 1{,}5234375 \times 10^{-1}$, $B = 2{,}0703125 \times 10^{-1}$, $C = 9{,}96875 \times 10^{1}$
c) $A = -2{,}7890625 \times 10^{1}$, $B = -8{,}088 \times 10^{3}$, $C = 1{,}0216 \times 10^{4}$

5.2 Operacions

Vegeu Secció 5.2 i Secció 5.3.

Problema 5.9. Suposeu que els registres de coma flotant `ft2 = 0x417AC000` i `ft4 = 0x3F140000`, i que s'executa la instrucció RISC-V `fadd.s ft6, ft2, ft4`. Amb arrodoniment al més pròxim (al parell en cas equidistant):

a) Calculeu a mà, seguint l'algorisme de suma de nombres en coma flotant, el valor final de `ft6` en hexadecimal.
b) Es produeix algun error de precisió en el resultat?
c) Convertiu a decimal el valor final de `ft6`.

Problema 5.10. Suposeu que `ft2 = 0xC076C000` i `ft4 = 0x3ECA8000`, i que s'executa `fadd.s ft6, ft2, ft4`. Amb arrodoniment al més pròxim (al parell en cas equidistant):

a) Calculeu a mà el valor final de `ft6` en hexadecimal.
b) Es produeix algun error de precisió?

c) Convertiu a decimal el valor final de `ft6`.

Problema 5.11. Suposeu que `ft2 = 0x42000000` i `ft4 = 0x3D800000`, i que s'executa `fmul.s ft6, ft2, ft4`. Suposant arrodoniment al més pròxim (al parell en cas equidistant), quin és el valor final de `ft6` en hexadecimal?

Problema 5.12. Suposeu els nombres A i B en format *half* de 16 bits (vegeu Problema 5.7). Amb un sumador amb bit de guarda, d'arrodoniment i *sticky*, i arrodoniment al més pròxim (al parell en cas equidistant):

a) Calculeu a mà $A + B$ suposant `A = 0xE4FE` i `B = 0xB640`. Doneu el resultat en hexadecimal i en decimal. Quina és la precisió del resultat?
b) Ídem amb `A = 0x4DC7` i `B = 0x391D`.

Problema 5.13. Suposeu els nombres A i B en format *half* de 16 bits. Amb arrodoniment al més pròxim (al parell en cas equidistant):

a) Calculeu a mà $A \times B$ suposant `A = 0x45A9` i `B = 0x484C`. Doneu el resultat en hexadecimal i en decimal. Quina és la precisió del resultat?
b) Ídem amb `A = 0x60D4` i `B = 0x533E`.

Problema 5.14. Suposeu els nombres A i B en format *half* de 16 bits. Amb arrodoniment al més pròxim (al parell en cas equidistant):

a) Calculeu a mà A/B suposant `A = 0xEA60` i `B = 0xE000`. Doneu el resultat en hexadecimal i en decimal.
b) Ídem amb `A = 0xCD98` i `B = 0x6400`.

Problema 5.15. Suposant que `f` i `g` són variables de tipus `float` emmagatzemades als registres de coma flotant `fa2` i `fa4`, traduïu a assemblador RISC-V les sentències en C següents. Tingueu en compte que l'assemblador només admet literals fraccionaris a les directives de dades (`.float`), mai com a operands d'instrucció:

a) `f = 3.14;`
b) `f = g;`
c) `f = g + 3.14;`

Problema 5.16. Traduïu a assemblador RISC-V la subrutina `absdif`:

C

```c
float absdif(float a, float b) {
    if (a > b)
        return a - b;
    else
        return b - a;
}
```

Problema 5.17. Donades les declaracions de funcions:

C

```c
float mitjana(float p[]);

float variancia(float vec[]) {
    int   i;
    float m, q;
    float vquadrats[100];
    for (i = 0; i < 100; i++)
        vquadrats[i] = vec[i] * vec[i];
    q = mitjana(vquadrats);
    m = mitjana(vec);
    return q - m * m;
}
```

---

a) Quines variables locals és convenient guardar al bloc d'activació i quines és més convenient guardar en registres? Quins registres caldrà preservar a la pila abans de cada crida a `mitjana`?
b) Traduïu a assemblador RISC-V la funció `variancia`.

Problema 5.18. Suposeu els nombres A i B en format *half* de 16 bits emmagatzemats a `t1` i `t2`. Escriviu programes en assemblador RISC-V que:

a) Calculin la suma $A = A + B$.
b) Calculin el producte $A = A \times B$ i indiquin si s'ha produït desbordament o subdesbordament.

Problema 5.19. Donades les declaracions en C:

C

```c
float v[4] = {1.5, -2.0, 3.25, 0.5};
float resultat;
```

Escriviu en assemblador RISC-V (extensió F) el fragment de codi que:

a) Llegeixi els elements `v[0]` i `v[2]` de la memòria i els deixi als registres de coma flotant `fa0` i `fa1` respectivament.
b) Calculi la suma `fa0 + fa1` i l'escrigui a `resultat`.
c) Compari `v[0]` i `v[2]` i deixi a `t0` el valor 1 si `v[0] > v[2]`, o 0 altrament (useu `flt.s`).

Problema 5.20. Escriviu en assemblador RISC-V les sentències equivalents a les conversions en C següents. Useu les instruccions `fcvt.w.s`, `fcvt.wu.s`, `fcvt.s.w` i `fcvt.s.wu` segons convingui.

a) `int i = (int) f;` — on `f` és un `float` a `fa0` i `i` és un `int` a `a0`.
b) `float f = (float) i;` — on `i` és a `a0` i `f` ha d'anar a `fa0`.
c) `unsigned int u = (unsigned int) f;` — on `f` és a `fa0` i `u` ha d'anar a `a0`.
d) Per a `f = 3.9`, calculeu a mà el resultat de l'apartat a) i expliqueu si es produeix truncament o arrodoniment.

5.3 No-associativitat

Vegeu Secció 5.2.3.

Problema 5.21. Suposeu els nombres A, B i C en format *half* de 16 bits. Amb un sumador amb bit de guarda, d'arrodoniment i *sticky*, i arrodoniment al més pròxim (al parell en cas equidistant):

a) Suposant `A = 0xF3FD`, `B = 0x73FD` i `C = 0x3C00`, calculeu a mà $(A+B)+C$ i $A+(B+C)$. Doneu els resultats en hexadecimal i en decimal. Es pot afirmar que $(A+B)+C = A+(B+C)$?
b) Feu la mateixa comprovació amb `A = 0x4F2A`, `B = 0x36A0` i `C = 0x4A12`.

Problema 5.22. Suposeu els nombres A, B i C en format *half* de 16 bits. Amb arrodoniment al més pròxim (al parell en cas equidistant):

a) Suposant `A = 0x30E0`, `B = 0x32A0` i `C = 0x563B`, calculeu a mà $A \times (B+C)$ i $(A \times B) + (A \times C)$. Doneu els resultats en hexadecimal i en decimal. Es pot afirmar que $A \times (B+C) = (A \times B) + (A \times C)$?
b) Feu la mateixa comprovació amb `A = 0xCEF9`, `B = 0xEFE6` i `C = 0x70FD`.

Tema 6: Rendiment i potència

6.1 Rendiment

Vegeu Secció 6.1; en particular, Equació 6.3 i Equació 6.4.

Problema 6.1. Un programa en alt nivell es tradueix a llenguatge assemblador en dues versions: una en RISC-V, que s'executarà en els processadors P1 i P2; l'altra, en x86-64, que s'executarà en P3. Cada processador té les característiques següents:

Processador	Freqüència	CPI mitjà	ISA	Nombre d'instruccions
P1	2 GHz	1,5	RISC-V	$2 \cdot 10^6$
P2	2,5 GHz	1	RISC-V	$2 \cdot 10^6$
P3	3 GHz	0,75	x86-64	10^6

Contesteu les preguntes següents raonant les respostes:

a) Quin dels tres processadors executa més ràpidament el programa? Quantes vegades és més ràpid que els altres dos?
b) Sense conèixer el nombre d'instruccions, es pot calcular si P1 executa el programa més ràpid que P2? I més ràpid que P3?
c) Amb aquestes dades, es pot comparar el rendiment de P1 i P2 en general (no només per a aquest programa)? I el de P1 i P3? I si el CPI fos el CPI mitjà per a tots els programes?

Problema 6.2. Considereu dos processadors P1 i P2 que executen una seqüència de 10^6 instruccions. Cada processador té les característiques següents:

Processador	Freqüència	CPI
P1	4 GHz	1,25
P2	3 GHz	0,75

Contesteu les preguntes següents raonant les respostes:

a) Quin dels dos processadors té el millor rendiment? Quantes vegades és més ràpid?
b) Quantes instruccions pot executar P2 en el temps que triga P1 a fer les 10^6 instruccions?

Problema 6.3. Considereu un processador sense instruccions de coma flotant. Les operacions en coma flotant s'implementen mitjançant subrutines. A partir d'estadístiques se sap el següent:

- El CPI mitjà és 3.
- El 20 % de les instruccions executades per qualsevol programa formen part de les subrutines de coma flotant.

Es vol avaluar la conveniència d'afegir instruccions de coma flotant al processador. En afegir-les s'observa:

- El temps de cicle passa d'1 ns a 1,05 ns.
- El CPI mitjà passa de 3 a 4.
- El nombre d'instruccions de coma flotant executades és ara 10 vegades menor.

Contesteu les preguntes següents raonant les respostes:

a) Val la pena afegir instruccions de coma flotant al repertori?
b) Valdria la pena si les subrutines de coma flotant representessin el 40 % de totes les instruccions (en comptes del 20 % de l'apartat anterior)?

Problema 6.4. En un processador d'1 GHz s'executa un programa P distribuït de la forma següent:

Aritmètiques	Escriptura	Lectura	Salt	Total
500	50	100	50	700

a) Si les instruccions aritmètiques triguen 1 cicle, les de lectura i d'escriptura 5 cicles, i els salts 2 cicles, quin és el temps d'execució de P?
b) Quin és el CPI?
c) Si s'optimitza el codi de manera que ara només s'executin 50 instruccions de lectura, quin és el guany (*speedup*) obtingut? I el CPI?

Problema 6.5. Considereu dues implementacions diferents del mateix ISA, on hi ha quatre classes d'instruccions, A, B, C i D. La taula següent mostra la freqüència de rellotge i el CPI de cada classe per a les dues implementacions:

	Freqüència	CPI A	CPI B	CPI C	CPI D
P1	1,5 GHz	1	2	3	4
P2	2 GHz	2	2	2	2

a) Quin és el CPI mitjà de cada implementació, suposant un programa amb instruccions distribuïdes de la forma: 10 % classe A, 20 % classe B, 50 % classe C i 20 % classe D?
b) Quina de les dues implementacions és més ràpida, suposant un programa de 10^6 instruccions?

Problema 6.6. Un processador no disposa d'instrucció de multiplicació. Cada multiplicació es resol amb una crida a una subrutina de 200 instruccions. Al disseny original, totes les instruccions tenen un CPI mitjà de 3. Es vol avaluar la conveniència d'afegir un multiplicador per maquinari.

Amb el nou disseny, un 0,5 % de les instruccions executades són multiplicacions (CPI = 40 cadascuna); la resta d'instruccions tenen un CPI mitjà de 3.

a) Si no hi ha cap altre canvi en el disseny, quin guany s'obté?
b) El nou multiplicador obliga a reduir la freqüència de rellotge des dels 2 GHz originals. Quina és la freqüència mínima acceptable perquè el nou processador no sigui més lent que l'original?

Problema 6.7. Considereu dos processadors que executen un programa de la forma que mostra la taula:

	Coma flotant	Enters	Lectura/escriptura	Salt	Total
a	35 s	85 s	50 s	30 s	200 s
b	50 s	80 s	50 s	30 s	210 s

Per a cadascuna de les execucions, contesteu les preguntes següents:

a) Quina és la millora de rendiment si es redueix el temps de les operacions de coma flotant en un 20 %?
b) Quin és el percentatge de reducció del temps d'execució d'operacions d'enters necessari per aconseguir una millora del 20 % en el rendiment global?
c) Es pot reduir el temps d'execució total en un 20 % millorant únicament les operacions de salt?

6.2 Llei d'Amdahl

Vegeu Secció 6.2; en particular, Equació 6.6 i Equació 6.7.

Problema 6.8. Un programa passa el 60 % del temps executant instruccions de coma flotant i el 40 % restant en altres instruccions.

a) Si s'afegeix una unitat de coma flotant que fa les operacions de coma flotant (CF) 5 vegades més ràpides, quin és el guany global?
b) Quin seria el guany màxim teòric si les operacions CF fossin infinitament ràpides?
c) Per assolir un guany global de 3, quantes vegades més ràpides han de ser les operacions CF?

Problema 6.9. Un sistema té tres components amb els temps d'execució relatius i les millores possibles indicats a la taula:

Component	Fracció del temps	Millora disponible
Unitat entera	50 %	2×
Unitat CF	30 %	4×
Accés a memòria	20 %	1,5×

a) Calculeu el guany global si s'apliquen totes tres millores simultàniament.
b) Quina millora individual té més impacte? Raoneu la resposta amb la llei d'Amdahl.
c) Si el pressupost només permet aplicar una millora, quina trieu? Per què?

Problema 6.10. Un fabricant vol anunciar que el seu nou processador és «10 vegades més ràpid» que el model anterior.

a) Si l'única millora és que les instruccions de coma flotant s'executen 20 vegades més ràpidament, quin ha de ser el percentatge mínim del temps d'execució que ocupen les instruccions CF perquè s'assoleixi el guany de 10?
b) En un programa típic, les instruccions CF representen el 35 % del temps. Quin guany real s'obtindria?
c) Quin és el guany màxim assolible per a qualsevol programa amb un 35 % de CF, independentment de la millora aplicada a les instruccions CF?

Problema 6.11. Considereu dos processadors A i B que executen versions diferents d'un mateix programa. Per a cada tipus d'instruccions, la taula següent mostra el nombre d'instruccions executades i els cicles per instrucció:

	Coma flotant	Enters	Lectura/escriptura	Salt
Proc. A	90 instr. · 2 CPI	240 instr. · 1,5 CPI	150 instr. · 6 CPI	120 instr. · 3 CPI
Proc. B	45 instr. · 3 CPI	225 instr. · 1 CPI	135 instr. · 5 CPI	45 instr. · 2 CPI

a) Calculeu el CPI mitjà de l'execució del programa en cadascun dels processadors.
b) Suposant que els processadors A i B treballen a les freqüències f_A i f_B respectivament, indiqueu la relació f_A/f_B per tal que tots dos processadors triguin el mateix temps a executar el programa.
c) Si s'aconseguís reduir a zero el temps d'execució de les instruccions de lectura i d'escriptura del processador A, quin seria el guany de rendiment aconseguit?

6.3 Potència

Vegeu Secció 6.3; en particular, Equació 6.9 i Equació 6.11.

Problema 6.12. La taula següent mostra la freqüència de rellotge, el voltatge i la potència dinàmica de dos processadors:

Processador	Freqüència	Voltatge	Potència dinàmica
A	10 MHz	5 V	2 W
B	3 GHz	1 V	100 W

a) Calculeu la capacitat equivalent (capacitància) dels processadors A i B.
b) Quina seria la potència del processador A si, sense canviar el voltatge ni la capacitància, es volgués aconseguir la mateixa freqüència de rellotge que el processador B?

Problema 6.13. La taula següent mostra l'evolució de la freqüència de rellotge, el voltatge i la potència dinàmica en vuit generacions de processadors Intel:

Generació	Processador	Any	Freqüència	Voltatge	Potència
1	80286	1982	12,5 MHz	5 V	3,3 W
2	80386	1985	16 MHz	5 V	4,1 W
3	80486	1989	25 MHz	5 V	4,9 W
4	Pentium	1993	66 MHz	5 V	10,1 W
5	Pentium Pro	1997	200 MHz	3,3 V	29,1 W
6	Pentium 4 Willamette	2001	2 GHz	1,75 V	75,3 W
7	Pentium 4 Prescott	2004	3,6 GHz	1,25 V	103 W
8	Core 2 Kentsfield	2007	2,667 GHz	1,1 V	95 W

a) Calculeu les ràtios de potència i freqüència entre generacions consecutives.
b) Quin és el canvi (ràtio) més gran en la freqüència entre generacions? I en la potència?
c) Quina és la mitjana geomètrica de les ràtios de freqüència i potència entre generacions consecutives?

d) Quantes vegades més gran és la freqüència de la darrera generació respecte de la primera? I la potència?
e) Calculeu la capacitat equivalent de cada generació.

Problema 6.14. Tot i que la potència dinàmica és la principal font de dissipació en circuits *complementary metal–oxide–semiconductor* (CMOS), els corrents de fuita produeixen també una dissipació de potència estàtica de la forma $P_s = V_{CC} \cdot I_{fuita}$. La taula següent mostra la dissipació de potència estàtica i dinàmica per a dues generacions de processadors:

Tecnologia	Potència dinàmica	Potència estàtica	Potència total	Voltatge
250 nm	49 W	1 W	50 W	3,3 V
90 nm	75 W	45 W	120 W	1,1 V

a) Trobeu quin percentatge de potència total dissipada correspon a la potència estàtica i dinàmica en cada generació.
b) Trobeu el corrent de fuita I_{fuita} per a cada tecnologia.
c) En quin percentatge es podria reduir la potència total de cada processador reduint només la potència dinàmica?

Tema 7: Memòria cau

7.1 Disseny bàsic

Vegeu Secció 7.2, Secció 7.3 i Secció 7.5.

Problema 7.1. Considereu que la jerarquia de memòria d'un sistema computador està formada per una memòria principal de 64 bytes i una memòria cau de 16 bytes, amb correspondència directa i blocs de 4 bytes. La unitat d'adreçament és el byte.

a) Donada l'adreça `0x39` de memòria principal, indiqueu:

- Quin és l'índex de la memòria cau que li correspon.
- Quina és l'etiqueta que li correspon.
- Quin és el byte que li correspon dins el bloc.

b) Considereu la seqüència de referències de lectura a blocs de memòria principal següent: 0, 1, 0, 3, 12, 10, 3, 7, 12, 15, 3. Indiqueu per a cada referència si es produeix un encert o una fallada a la memòria cau.

Problema 7.2. Considereu un processador de 16 bits amb una memòria cau amb les característiques següents:

- Correspondència directa.
- Mida total: 256 bytes.
- Mida de bloc: 16 bytes.
- Escriptura immediata sense assignació.

a) Ompliu la taula següent a partir de la seqüència de referències donada:

Tipus	Adreça (hex)	Etiqueta (hex)	Índex MC (hex)	Encert/Fallada	Bytes llegits MP	Bytes escrits MP	Lectura dades MC	Escriptura dades MC
L	`4534`							
L	`4568`							
E	`13A4`							
E	`13A8`							
L	`3560`							
E	`453C`							
E	`60A0`							
L	`453C`							
E	`3900`							
L	`A238`							

b) Ompliu la mateixa taula suposant que la memòria cau té una política d'escriptura retardada amb assignació.

c) Indiqueu per a cada política: taxa de fallades, nombre de bytes llegits d'MP i nombre de bytes escrits a MP.

Problema 7.3. El programa del Codi 7.1 multiplica una matriu `A(32×32)` per un vector `B(32)` produint com a resultat un vector `C(32)`:

Codi 7.1: Producte d'una matriu de bytes per un vector.

`C`

```
char A[32][32], B[32], C[32];
void main() {
    int i, j;
    for (i = 0; i < 32; i++)
        for (j = 0; j < 32; j++)
            C[i] = A[i][j] * B[j] + C[i];
}
```

Els elements d'`A`, `B` i `C` són bytes. Tots els elements de `C` han estat prèviament inicialitzats a zero. `A` s'emmagatzema a partir de l'adreça 0 (adreces 0..1023), `B` a continuació (adreces 1024..1055) i `C` tot seguit (adreces 1056..1087). Les variables `i` i `j` estan en registres.

El computador disposa d'una memòria cau de correspondència directa amb 4 línies i blocs de 32 bytes, amb escriptura immediata. Suposant que en iniciar-se l'execució la memòria cau és buida, calculeu la taxa d'encerts.

Problema 7.4. Es vol dissenyar una nova versió d'un processador que disposa d'una memòria cau de correspondència directa amb 4 línies i blocs d'una paraula (32 bits). La nova versió tindrà el doble de capacitat. Es consideren dues alternatives:

- **Alternativa 1**: doblar el nombre de línies (8 línies d'una paraula).
- **Alternativa 2**: doblar la mida del bloc (4 línies de 2 paraules).

a) Proposeu una seqüència de referències a paraules de memòria principal tal que la taxa d'encerts de l'alternativa 1 sigui major que la de l'alternativa 2.
b) Proposeu una seqüència de referències tal que la taxa d'encerts de l'alternativa 2 sigui major que la de l'alternativa 1.
c) Raoneu quina característica dels programes s'explota en major mesura en cadascuna de les dues alternatives.

7.2 Rendiment

Vegeu Secció 7.7; en particular, Equació 7.1 i Equació 7.4.

Problema 7.5. Una mesura possible del rendiment d'una memòria cau és la taxa de tràfic, definida com el quocient entre el nombre de bytes transferits des de/cap a la memòria principal en un sistema amb memòria cau i el nombre de bytes transferits en un sistema sense memòria cau.

Suposeu que totes les referències a memòria són de paraules de 32 bits. Calculeu la taxa de tràfic en funció de la taxa d'encerts (h), la mida del bloc en bytes (B), el percentatge d'escriptures (p_e) i el percentatge de blocs reemplaçats que han estat modificats (p_m) per a:

a) Escriptura immediata sense assignació.

b) Escriptura retardada amb assignació.

Problema 7.6. Es vol definir la política d'escriptura de la memòria cau d'un processador. Es consideren dues alternatives: (1) escriptura immediata sense assignació i (2) escriptura retardada amb assignació.

Mitjançant simulació s'han obtingut les mesures següents:

- Percentatge d'escriptures (p_e): 20 %.
- Percentatge de blocs modificats sobre el total de blocs reemplaçats (p_m): 33,33 %.
- Taxa d'encerts en el cas (1): 0,9.
- Taxa d'encerts en el cas (2): 0,85.

El temps d'accés en cas d'encert (t_h) és 10 ns. La lectura o escriptura d'un bloc de memòria principal (t_{bloc}) requereix 100 ns.

a) Calculeu el temps d'accés mitjà (TAM) en ambdues alternatives.
b) Indiqueu quina alternativa seria la més ràpida per a un programa que només fes lectures.

Problema 7.7. Considereu una CPU amb memòria cau en què s'han observat les característiques següents:

- CPI_{ideal}: 1,5 cicles/instrucció.
- Temps de cicle (t_c): 10 ns.
- Nombre de referències per instrucció (n_r): 1,6.
- Memòries cau d'instruccions i dades separades.
- Memòria cau de dades amb escriptura retardada amb assignació.

Característica	MC d'instruccions	MC de dades
Accessos per instrucció	1	0,6
Percentatge d'escriptures (p_e)	—	40 %
Percentatge de blocs modificats (p_m)	—	20 %
Taxa de fallades (m)	4 %	10 %
Penalització bloc no modificat (t_p)	10 cicles	15 cicles
Penalització bloc modificat (t_p)	—	20 cicles
Temps d'encert (t_h)	1 cicle	1 cicle

a) Quin és el temps d'accés mitjà (TAM) en cicles?
b) Quin és el temps mitjà d'execució d'una instrucció (t_{exe}) en ns?

Problema 7.8. Considereu una CPU amb memòria cau en què s'han observat les característiques següents:

- CPI_{ideal}: 1,8 cicles/instrucció.
- Temps de cicle (t_c): 10 ns.
- Nombre de referències per instrucció (n_r): 1,4.
- Mida del bloc: 8 bytes.
- Política d'escriptura retardada amb assignació.
- Temps d'encert (t_h): 1 cicle.
- Temps de servei d'MP per llegir/escriure blocs (t_{bloc}): 15 cicles.
- Percentatge d'escriptures (p_e): 15 %.
- Percentatge de blocs modificats (p_m): 20 %.

- Taxa d'encerts (h): 90 %.

a) Quin és el temps de penalització (t_p) en cas que el bloc estigui modificat? I en cas que no ho estigui?
b) Quin és el temps d'accés mitjà (TAM) en cicles?
c) Quin és el temps mitjà d'execució d'una instrucció en ns?

Problema 7.9. A l'hora de dissenyar una memòria cau d'instruccions de 8 KiB es dubta entre dues mides de bloc, amb les taxes de fallades mesurades següents:

- 8 bytes: 16 %.
- 16 bytes: 10 %.

a) Calculeu el temps d'accés mitjà (TAM) sabent que:
   - El temps d'accés en cas d'encert (t_h) és 1 cicle.
   - El temps de servei d'MP per a blocs de 8 bytes és 6 cicles; per a blocs de 16 bytes és 12 cicles.

b) Quant ocuparan totes les etiquetes en cadascuna de les dues opcions, sabent que:
   - MP té 2^{32} bytes.
   - Les memòries cau són de correspondència directa.
   - La unitat mínima d'accés a memòria és el byte.

7.3 Associativitat i multinivell

Vegeu Secció 7.3.2, Secció 7.4 i Secció 7.9.

Problema 7.10. Considereu un processador de 16 bits (paraules i adreces de 16 bits) amb una memòria cau amb les característiques següents:

- Correspondència associativa de 2 vies.
- Mida total: 1024 bytes.
- Mida de bloc: 16 bytes.
- Política d'escriptura immediata sense assignació.
- Algorisme de reemplaçament: LRU.

Ompliu la taula següent a partir de la seqüència de referències donada (L byte = lectura d'1 byte, L paraula = lectura d'una paraula, de 2 bytes, E byte = escriptura d'1 byte, E paraula = escriptura d'una paraula):

Tipus	Adreça (hex)	Bloc MP	Conjunt MC	Encert/Fallada	Lectura MP (S/N, adreça, mida)	Escriptura MP (S/N, adreça, mida)
L byte	`A930`					
L paraula	`B930`					
E byte	`A972`					
E paraula	`A932`					
E byte	`C935`					
L paraula	`C934`					
E byte	`B976`					

Tipus	Adreça (hex)	Bloc MP	Conjunt MC	Encert/Fallada	Lectura MP (S/N, adreça, mida)	Escriptura MP (S/N, adreça, mida)
E paraula	`B936`					
L byte	`B938`					
L paraula	`A978`					

Problema 7.11. Sobre un sistema amb una memòria cau de dades amb 64 conjunts, 4 vies per conjunt, 32 bytes per bloc i algorisme LRU, s'executen dues versions d'una mateixa aplicació:

`C`

```c
int A[128][1024]; // emmagatzemada a partir de l'adreça 0

// Versió A
sumA = 0;
for (i = 0; i < 128; i++)
    for (j = 0; j < 1024; j++)
        sumA = sumA + A[i][j];

// Versió B
sumA = 0;
for (j = 0; j < 1024; j++)
    for (i = 0; i < 128; i++)
        sumA = sumA + A[i][j];
```

Indiqueu quantes fallades hi ha a la memòria cau de dades per a cadascuna de les dues versions. Considereu que `i`, `j` i `sumA` estan en registres.

Problema 7.12. El subsistema de memòria d'un computador està organitzat en tres nivells: L1 (4 línies amb blocs de 4 bytes, correspondència directa), L2 (16 línies amb blocs de 4 bytes, associativa de 2 vies, LRU) i MP (1 MiB). En cas de fallada a L1 es comprova L2; si hi és, es copia a L1 sense accedir a MP. Si no hi és a L2, s'accedeix a MP i es copia a les dues memòries cau.

La taxa d'encerts de cada nivell es defineix com h_{L1} = encerts a L1/peticions a L1, i igualment h_{L2}.

Donada la seqüència de 7 adreces de lectura següent, repetida 4 vegades (28 accessos en total):

```
0, 5, 10, 12, 34, 0, 66
```

a) Suposant les memòries cau inicialment buides, calculeu les taxes h_{L1} i h_{L2}.
b) Calculeu de nou les taxes h_{L1} i h_{L2} suposant que l'algorisme de reemplaçament de L2 és FIFO.

Problema 7.13. Es vol dissenyar una memòria cau multinivell (L1 i L2). Es coneixen les dades següents:

- El temps de servei de L1 en cas d'encert és $t_{h_{L1}}$; en cas de fallada cal afegir-hi el temps de servei de L2.
- El temps de servei de L2 en cas d'encert és $t_{h_{L2}}$; en cas de fallada cal afegir-hi la penalització t_p (temps d'accés a MP).
- m_{L1}: taxa de fallades a L1.
- m_{L2}: taxa de fallades local a L2 (fallades a L2 / accessos a L2).

Obteniu l'expressió del temps d'accés mitjà (TAM) en cicles, en funció de $t_{h_{L1}}$, $t_{h_{L2}}$, t_p, m_{L1} i m_{L2}.

Problema 7.14. Es disposa d'un processador de 8 bits amb una memòria cau completament associativa amb les característiques següents:

- Nombre de línies: 4.
- Mida del bloc: 8 bytes.
- Algorisme de reemplaçament: LRU.
- Política d'escriptura retardada amb assignació.
- Temps d'encert (t_h): 1 cicle.
- Temps de servei d'MP (t_{bloc}): 10 cicles.

a) Temps de servei en cas de fallada de lectura que reemplaça un bloc no modificat.
b) Temps de servei en cas de fallada de lectura que reemplaça un bloc modificat.
c) Temps de servei en cas de fallada d'escriptura que reemplaça un bloc no modificat.
d) Temps de servei en cas de fallada d'escriptura que reemplaça un bloc modificat.
e) Suposant la memòria cau inicialment buida i la llista d'accessos (en decimal, L = lectura, E = escriptura):

```
2L, 9L, 16L, 2L, 33E, 42E, 33L, 2L, 17L, 27E, 43L, 3L, 44L, 46L, 3L, 14L, 17L, 6L, 35L
```

```
- Indiqueu per a cada accés si produeix encert, fallada amb reemplaçament (indicant el número del bloc r
- Calculeu el temps total de servei de la llista d'accessos.
```

7.4 Tipologia de les fallades

Vegeu Secció 7.8 i Essencial 7.4.

Problema 7.15. Una memòria cau de correspondència directa té 8 línies amb blocs de 4 bytes. Considereu les seqüències d'accés a bytes següents (les adreces estan en decimal):

- Seqüència A: 0, 4, 8, 12, 16, 20, 24, 28, 0, 4, 8, 12.
- Seqüència B: 0, 4, 8, 12, 0, 4, 8, 12, 0, 4, 8, 12.
- Seqüència C: 0, 32, 0, 32, 0, 32, 0, 32.

a) Per a cada seqüència, calculeu el nombre de fallades i classifiqueu-les en fallades d'arrencada en fred (*cold-start*), de conflicte o de capacitat.
b) Per a la seqüència C, quin canvi de disseny de la memòria cau eliminaria les fallades de conflicte? Justifiqueu-ho.
c) Una memòria cau associativa de 2 vies (mateixa capacitat total) milloraria els resultats de la seqüència C? I de la seqüència B?

Problema 7.16. Considereu el programa C del Codi 7.2, on `N = 256` i els elements d'`A` i `B` són enters de 4 bytes:

Codi 7.2: Suma dels elements de `A` i de `B`.

```
C
```

```
int A[N], B[N];
void main() {
    int i, suma = 0;
    for (i = 0; i < N; i++)
        suma += A[i] + B[i];
}
```

`A` s'emmagatzema a partir de l'adreça `0x000` i `B` a partir de `0x400`. La memòria cau té correspondència directa, 16 línies amb blocs de 32 bytes i política d'escriptura immediata.

Nota: en aquest problema, excepcionalment, les adreces s'escriuen en format reduït, sense els zeros de l'esquerra. Totes són inferiors a `0x1000`, i per saber a quina línia de la memòria cau va cada bloc només calen els bits de menys pes (els 5 de desplaçament i els 4 d'índex).

a) Quantes fallades d'arrencada en fred es produeixen en el primer recorregut d'`A`? I en el de `B`?
b) Es produeixen fallades de conflicte entre `A` i `B`? Raoneu la resposta calculant a quines línies de la memòria cau es mapen.
c) Si `B` comencés a `0x600` en comptes de `0x400`, canviaria la resposta a l'apartat b)?

Tema 8: Memòria virtual

8.1 Traducció d'adreces i fallada de pàgina

Vegeu Secció 8.2, Secció 8.3 i Secció 8.4.

Problema 8.1. Es disposa d'un sistema amb memòria virtual paginada amb les característiques següents:

- Mida de la memòria virtual: 16 MiB.
- Mida de la memòria física: 16 KiB.
- Mida de les pàgines: 256 bytes.

Cada programa pot tenir a memòria física un màxim de 3 pàgines simultàniament. En cas que calgui alliberar una pàgina, s'usa l'algorisme de reemplaçament LRU.

a) Indiqueu les dimensions de la taula de pàgines de cada programa, tenint en compte que cada entrada té un bit de validesa (V) i un bit de modificació (D).
b) La taula següent mostra una llista d'adreces lògiques generades pel processador. Ompliu-la indicant, per a cada adreça, el número de pàgina (columna A), si es produeix una fallada de pàgina (columna B) i, en cas de reemplaçament, el número de pàgina reemplaçada (columna C). Considereu que inicialment no hi ha cap pàgina carregada a memòria física.

Adreça (hex)	A — Núm. pàgina (hex)	B — Fallada de pàgina	C — Pàgina reemplaçada (hex)
000023			
000101			
000102			
000200			
000010			
000111			
000416			
000520			
000101			

Problema 8.2. Un sistema té adreces lògiques de 32 bits, pàgines de 4 KiB i taules de pàgines de dos nivells. Cada entrada de la taula ocupa 4 bytes.

a) Quants bits s'usen per al desplaçament dins la pàgina, per a l'índex de primer nivell i per a l'índex de segon nivell (suposeu una divisió simètrica dels bits d'índex)?
b) Quina és la mida d'una taula de pàgines de primer nivell? I d'una de segon nivell?
c) Quin és el nombre màxim de taules de segon nivell que pot tenir un procés? Quina és la mida total de la jerarquia de taules en el pitjor cas?
d) Compareu la mida total de la jerarquia de dos nivells (pitjor cas) amb la d'una taula de pàgines de nivell únic. Quin avantatge té la jerarquia en el cas típic en què un procés usa poca memòria?

Problema 8.3. Considereu un processador amb memòria virtual paginada amb les característiques següents:

- 16 bits d'adreça lògica, 15 bits d'adreça física.
- Mida de les pàgines: 8 KiB.
- Reemplaçament LRU.

El contingut inicial de la taula de pàgines és:

VPN	V	D	PPN
0	0	0	—
1	0	0	—
2	1	0	0
3	1	0	1
4	1	0	2
5	0	0	—
6	0	0	—
7	0	0	—

El contingut inicial de la memòria física és:

PPN	VPN
0	2
1	3
2	4
3	—

L'ordre d'accés recent és: PPN 1 (VPN 3) el més recent, PPN 0 (VPN 2) el segon i PPN 2 (VPN 4) el més antic.

Ompliu la taula següent per a la seqüència de referències donada (E = escriptura, L = lectura). Indiqueu també el contingut final de la taula de pàgines i de la memòria física.

Adreça lògica (hex)	VPN	PPN	Fallada de pàgina	Lectura disc	Escriptura disc	Pàg. reemplaçada VPN	Pàg. reemplaçada PPN
E `F458`							
E `8666`							
L `1BBF`							
E `5C44`							
L `6600`							
L `4000`							

Problema 8.4. Un sistema experimental gestiona la memòria virtual amb paginació i algorisme LRU. Té 64 KiB de memòria virtual, 8 KiB de memòria física i pàgines d'1 KiB. La taula de pàgines ocupa sempre l'últim marc de pàgina.

Una aplicació s'ubica a l'espai lògic de la manera següent:

- Codi: adreces `0x0000`–`0x03FF`.

- Dades: adreces `0x0400`–`0x47FF`.

Dins les dades hi ha una matriu declarada en C com:

C

```c
short MAT[32][256];
```

emmagatzemada a partir de l'adreça base `0x0400`. Els `short` es codifiquen en 16 bits (2 bytes). La resta de variables i la pila s'emmagatzemen a partir de `0x4400`. El compilador reserva registres per a `i` i `j`.

Considereu el fragment de codi del Codi 8.1:

Codi 8.1: Bucles que escriuen la matriu `MAT`.

C

```c
for (i = 0; i < 32; i++)
    for (j = 0; j < 256; j++)
        MAT[i][j] = i + j;
```

Abans d'executar el Codi 8.1, la taula de pàgines indica que només hi ha carregada a memòria física la pàgina corresponent al codi de l'aplicació.

a) Quins elements de `MAT` es porten a memòria física en la primera fallada de pàgina?
b) Indiqueu, justificant-ho, el nombre total de fallades de pàgina que es produiran.
c) En alguns llenguatges com ara FORTRAN, les matrius s'emmagatzemen per columnes. Considerant aquest emmagatzematge, quins elements de `MAT` es portarien a memòria física en la primera fallada de pàgina?
d) Considerant l'emmagatzematge per columnes, indiqueu el nombre total de fallades de pàgina que es produirien.

8.2 TLB

Vegeu Secció 8.5.

Problema 8.5. Considereu un sistema amb memòria virtual de pàgines de 4 KiB, un TLB de 4 entrades completament associatiu i reemplaçament LRU.

El contingut inicial del TLB és:

V	VPN	PPN
1	11	12
1	7	4
1	3	6
0	—	—

(La quarta entrada és lliure.)

El contingut inicial de la taula de pàgines és:

VPN	V	PPN
0	1	5
1	0	—
2	0	—
3	1	6
4	1	9
5	1	11
6	0	—
7	1	4
8	0	—
9	0	—
10	1	3
11	1	12

A les pàgines que es portin a memòria física se'ls assignaran PPN correlatius a partir del primer de lliure (13, 14, etc.). Inicialment, l'ordre d'ús de les entrades vàlides del TLB, de la més recentment usada a la més antiga, és VPN 11, VPN 7, VPN 3.

a) Per a cadascuna de les dues llistes d'adreces lògiques següents, indiqueu l'estat final del TLB i de la taula de pàgines, i per a cada referència si es produeix una fallada de TLB i si es produeix una fallada de pàgina:

- Llista 1: 4095, 31272, 15789, 15000, 7193, 4096, 8912.
- Llista 2: 9452, 30964, 19136, 46502, 38110, 17653, 47480.

b) Raoneu, sense refer la traça completa, com canviaria el nombre total de fallades de TLB de cada llista si la mida de pàgina fos de 16 KiB en comptes de 4 KiB.

c) Calculeu la mida de la taula de pàgines per als dos sistemes següents:

Mida adreces lògiques	Mida pàgina	Mida PTE
32 bits	4 KiB	4 bytes
64 bits	16 KiB	8 bytes

Problema 8.6. Considereu el codi assemblador RISC-V del Codi 8.2, on a l'etiqueta `A` li correspon l'adreça `0x10010000`:

Codi 8.2: Bucle que llegeix i escriu paraules a partir de `A`.

`RV32I`

```
        la      s0, A
        li      s1, 0
        li      s2, 512000
for:
        bge     s1, s2, fi
        slli    t0, s1, 2
        add     t0, s0, t0
        lw      t1, 0(t0)
        li      t3, 8192
        add     t3, t0, t3
        lw      t2, 0(t3)
        add     t2, t2, t1
```

```
        sw      t2, 0(t3)
        li      t3, 16384
        add     t3, t0, t3
        sw      t2, 0(t3)
        addi    s1, s1, 512
        j       for
fi:
```

Suposant pàgines de 8 KiB i un TLB de 4 entrades completament associatiu amb reemplaçament LRU:

a) Per a cadascuna de les instruccions d'accés a dades de memòria (`lw`/`sw`) i per a les primeres 5 iteracions del bucle, indiqueu a quina pàgina de la memòria virtual s'accedeix.
b) Indiqueu el nombre total de fallades del TLB de dades en tot el bucle (només es consideren els accessos a dades; les captures d'instruccions passen per un TLB d'instruccions a part, que no es té en compte).
c) Responeu les mateixes preguntes suposant pàgines de 4 KiB.

Problema 8.7. Considereu el codi assemblador RISC-V del Codi 8.3, on a l'etiqueta `A` li correspon l'adreça `0x10010000`:

Codi 8.3: Bucle que llegeix i escriu mitges paraules a partir de `A`.

`RV32I`

```
        la      s0, A
        li      s1, 0
        li      s2, 4096
for:
        bge     s1, s2, fi
        add     t0, s1, s1
        add     t0, s0, t0
        lh      t1, 0(t0)
        lh      t2, 1024(t0)
        add     t2, t2, t1
        sh      t2, 1024(t0)
        li      t3, 4096
        add     t3, t0, t3
        sh      t2, 0(t3)
        addi    s1, s1, 1
        j       for
fi:
```

Suposant pàgines d'1 KiB i un TLB completament associatiu de 4 entrades amb reemplaçament LRU:

a) Indiqueu el nombre de pàgines diferents accedides per a cadascuna de les quatre instruccions d'accés a dades de memòria (`lh`/`sh`).
b) Calculeu el nombre de fallades del TLB de dades en tot el bucle (només es consideren els accessos a dades, no les captures d'instruccions).
c) Es vol redissenyar el TLB, mantenint el reemplaçament LRU, per minimitzar el nombre de fallades en aquest programa. Indiqueu quin és el nombre mínim de fallades assolible i quin és el nombre mínim d'entrades de TLB necessàries per obtenir-lo.
d) Repetiu els apartats anteriors considerant pàgines de 4 KiB.

8.3 Protecció

Vegeu Secció 8.6.

Problema 8.8. Un sistema de memòria virtual amb pàgines de 4 KiB disposa, a cada entrada de la taula de pàgines, d'un bit de validesa (V), un bit de modificació (D) i un bit de permís d'escriptura (E: 1 = lectura i escriptura; 0 = només lectura).

VPN	V	D	E	PPN
0	1	0	0	3
1	1	1	1	7
2	0	0	1	—
3	1	0	0	5

Per a cada accés de la llista següent, indiqueu si té èxit, provoca una fallada de pàgina o provoca una excepció de protecció:

a) Lectura de l'adreça lògica `0x0FF0`.
b) Escriptura a l'adreça lògica `0x1234`.
c) Lectura de l'adreça lògica `0x2000`.
d) Escriptura a l'adreça lògica `0x3100`.
e) Escriptura a l'adreça lògica `0x0010`.

Nota: en aquest problema, excepcionalment, les adreces lògiques s'escriuen en format reduït, amb 4 dígits hexadecimals. Amb pàgines de 4 KiB, els 3 dígits de menys pes són el desplaçament dins de la pàgina i el de més pes és el VPN, i totes les adreces són de les pàgines 0–3 de la taula: els dígits de l'esquerra, que valen zero, no hi aporten res.

8.4 Integració del TLB i la memòria cau

Vegeu Secció 8.7.

Problema 8.9. Un sistema amb adreces lògiques de 32 bits té pàgines de 4 KiB, una memòria cau de dades indexada virtualment i etiquetada físicament (VIPT), de correspondència directa, amb 64 línies amb blocs de 32 bytes, i un TLB de dades completament associatiu de 16 entrades.

a) Indiqueu quants bits de l'adreça lògica s'usen per al desplaçament dins la pàgina, quants per al VPN, quants per a l'índex de l'MC i quants per al desplaçament dins el bloc.
b) Expliqueu el flux complet de traducció i accés a dades per a una lectura amb encert al TLB i encert a l'MC.
c) Per al mateix accés, quin és el flux si hi ha fallada de TLB, però la pàgina és present a memòria física?
d) En una MC indexada físicament (PIPT), la traducció ha de completar-se abans d'accedir a l'MC. En canvi, en VIPT, les dues operacions poden fer-se alhora. Sota quina condició una MC VIPT no presenta problemes d'aliàsing?

Problema 8.10. Considereu un processador RISC-V amb una memòria cau de dades de 4 vies, 64 conjunts i 32 bytes per bloc (algorisme LRU), i un TLB de dades completament associatiu de 32 entrades, amb pàgines de 8 KiB i algorisme LRU. Considereu la memòria cau i el TLB inicialment buits.

S'executen dues versions d'una mateixa aplicació:

C

```c
int A[128][1024]; // emmagatzemada a partir de l'adreça 0

// Versió A
sumA = 0;
for (i = 0; i < 128; i++)
    for (j = 0; j < 1024; j++)
        sumA = sumA + A[i][j];

// Versió B
sumA = 0;
for (j = 0; j < 1024; j++)
    for (i = 0; i < 128; i++)
        sumA = sumA + A[i][j];
```

Suposant que `i`, `j` i `sumA` estan en registres, indiqueu quantes fallades hi ha a la memòria cau i al TLB per a cada versió.

Tema 9: Excepcions i interrupcions

9.1 Excepcions i interrupcions

Vegeu Secció 9.1, Secció 9.2, Secció 9.3 i Secció 9.4.

Problema 9.1. Responeu les preguntes següents sobre el mecanisme d'excepcions i interrupcions en RISC-V:

a) Quina diferència hi ha entre una excepció i una interrupció? Doneu un exemple de cadascuna.
b) Quan es produeix una excepció, el processador desa l'adreça de retorn al registre `mepc`. A quina instrucció apunta `mepc` en cas d'excepció? I en cas d'interrupció?
c) Quins passos realitza el maquinari automàticament quan es produeix una excepció en mode M?
d) Quina és la responsabilitat de la rutina de servei d'excepcions (RSE) pel que fa als registres generals? Per què és diferent de la convenció ABI habitual?

Problema 9.2. El registre `mcause` de RISC-V conté el codi de causa d'una excepció o interrupció. El bit de més pes indica si és una interrupció (1) o una excepció (0); els bits restants identifiquen el tipus.

a) Indiqueu el valor de `mcause` per a cadascuna de les situacions següents:

- Instrucció en adreça no alineada.
- Accés a memòria en adreça no alineada (lectura).
- Crida al sistema (`ecall`) des de mode usuari.
- Interrupció de temporitzador en mode M.

b) El registre `mtval` conté informació addicional sobre la causa. Quin valor conté en cas d'una excepció per adreça no alineada en un accés a memòria?

Problema 9.3. Escriviu en assemblador RISC-V una rutina de servei d'excepcions (RSE) mínima que:

a) Salvi tots els registres generals a la pila.
b) Llegeixi `mcause` i salti a un gestor específic si la causa és una crida al sistema (`ecall`) des de mode usuari (codi de causa 8); altrament, salti a un gestor d'error genèric.
c) Restauri tots els registres i retorni amb `mret`.

Suposeu que el processador opera en mode M i que el registre `sp` apunta a una pila vàlida en entrar a l'RSE.

Problema 9.4. Responeu les preguntes següents sobre els registres `mepc` i `mtvec`:

a) Una instrucció `lw t0, 0(t1)` a l'adreça `0x80001000` causa una excepció perquè `t1` conté una adreça no alineada. A quina adreça apunta `mepc`?
b) Una interrupció de temporitzador arriba mentre s'executa la instrucció `lw t0, 0(a0)` a l'adreça `0x80002004`. A quina adreça apunta `mepc`? Quan es reprèn l'execució amb `mret`, quina instrucció s'executa?

c) El camp `MODE` de `mtvec` (bits 1:0) pot valer 0 (mode directe) o 1 (mode vectoritzat). Expliqueu la diferència. En mode vectoritzat, a quina adreça salta el processador per a una interrupció de temporitzador de màquina (codi 7)?
d) L'RSE ha d'incrementar `mepc` en 4 per a les crides al sistema (`ecall`). Per què? Escriviu les instruccions necessàries.

9.2 Crides al sistema

Vegeu Secció 9.5.

Problema 9.5. En RISC-V, les crides al sistema es fan amb la instrucció `ecall`. El codi de servei es passa al registre `a7` i l'argument principal a `a0`.

Escriviu fragments de codi assemblador RISC-V per a les operacions següents, usant els codis de servei de RARS:

a) Imprimir l'enter emmagatzemat al registre `t0` per la sortida estàndard.
b) Llegir un enter de l'entrada estàndard i deixar el resultat a `t1`.
c) Imprimir la cadena de caràcters amb adreça base a `t2`.
d) Finalitzar el programa.

Problema 9.6. Escriviu un programa complet en assemblador RISC-V que:

a) Llegeixi dos enters de l'entrada estàndard.
b) Imprimeixi el màxim dels dos seguit d'un salt de línia.
c) Finalitzi correctament.

El programa ha de seguir les convencions del curs: el punt d'entrada és la primera instrucció del segment `.text`, i la sortida es fa amb `li a7, 93` + `ecall` (codi de sortida a `a0`).

Problema 9.7. Escriviu un programa complet en assemblador RISC-V que:

a) Declari un vector de 10 enters inicialitzats a zero.
b) Llegeixi 10 enters de l'entrada estàndard i els escrigui al vector.
c) Calculi la suma dels elements del vector.
d) Imprimeixi la suma per la sortida estàndard.
e) Finalitzi correctament.

9.3 Entrada/sortida

Vegeu Secció 9.6.

Problema 9.8. Responeu les preguntes següents sobre els mecanismes d'entrada/sortida:

a) Quina diferència hi ha entre la sincronització per enquesta (*polling*) i la sincronització per interrupcions? Quins són els avantatges i inconvenients de cadascun?
b) En un sistema amb E/S per interrupcions, quins passos cal seguir per habilitar les interrupcions en RISC-V (registres `mstatus` i `mie`)?
c) Quin és el rol del bit `MIE` del registre `mstatus` i del bit `MTIE` del registre `mie` en el control de les interrupcions de temporitzador?

Problema 9.9. Un sistema RISC-V disposa d'un port en sèrie amb els registres mapats a memòria següents:

Adreça	Registre	Descripció
`0xF0000000`	`UART_RX`	Dada rebuda (byte de menys pes)
`0xF0000004`	`UART_TX`	Dada a transmetre (byte de menys pes)
`0xF0000008`	`UART_STATUS`	Bit 0: dada disponible per llegir; bit 1: transmissor lliure

Escriviu fragments de codi assemblador RISC-V que implementin:

a) Una funció `uart_getchar` que esperi fins que hi hagi una dada disponible i la retorni a `a0`.
b) Una funció `uart_putchar` que esperi fins que el transmissor estigui lliure i transmeti el byte d'`a0`.
c) Un programa principal que llegeixi caràcters del port en sèrie i els retransmeti en majúscules fins que rebi un caràcter de nova línia (`'\n'`).

Problema 9.10. Un sistema RISC-V gestiona un teclat per interrupcions. El controlador del teclat té un registre d'estat a l'adreça `0xF0001000` (bit 0 = tecla disponible) i un registre de dades a `0xF0001004`.

a) Escriviu el codi assemblador per habilitar les interrupcions externes en mode M (bits `MIE` de `mstatus` i `MEIE` de `mie`).
b) Escriviu una RSE mínima que, en rebre una interrupció externa (bit `MEIP` de `mip` actiu), comprovi el registre d'estat del teclat (`0xF0001000`, bit 0) per confirmar-ne l'origen, llegeixi el caràcter del registre de dades i el desi a una variable global `car`, i retorni correctament.
c) Per distingir entre una interrupció de teclat i una altra interrupció externa, es pot usar el codi de causa o cal llegir directament el registre d'estat del dispositiu? Justifiqueu.

9.4 Fallada de pàgina

Vegeu Secció 9.7.

Problema 9.11. A RISC-V, la fallada de TLB (*TLB miss*) la resol el maquinari, amb el recorregut de la taula de pàgines (*page-table walker*), de manera transparent i sense generar cap excepció; quan el recorregut troba que la pàgina no és a la memòria física (bit V = 0 a la PTE), es genera una excepció de **fallada de pàgina** (Essencial 9.9), i, si l'accés no té permís (una escriptura amb E = 0), una de protecció. En canvi, en arquitectures amb TLB gestionat per **programari** (com MIPS), la mateixa fallada de TLB genera una excepció, i és una rutina de l'SO qui llegeix la taula de pàgines i actualitza el TLB (Aprofundiment 8.5).

a) En RISC-V, quin codi de causa (`mcause`) s'assigna a una fallada de pàgina en una lectura de dada? I en una escriptura? Quin valor pren `mtval` en tots dos casos?
b) Descriviu els passos que ha de seguir l'RSE de RISC-V per gestionar una fallada de pàgina: com localitza la PTE a partir de `mtval`, què fa segons el bit V, quin paper hi té `sfence.vma` i com es prepara la reexecució de la instrucció causant.
c) En un sistema amb TLB gestionat per programari, la rutina que atén una fallada de TLB accedeix a la taula de pàgines i pot provocar, al seu torn, una nova fallada de TLB si la pàgina que conté la taula tampoc no és al TLB. Expliqueu per què aquesta situació és problemàtica i com es pot evitar.

Part III

Solucions

Tema 1: Introducció

1.1 El cicle de vida del programa: Generació vs. execució

💡 Exemple 1: Solució: executar Python en RISC-V

Enunciat: Problema 1.1.
Tots els passos s'executen en la **màquina RISC-V**. El traductor binari (element 2) és executable en RISC-V i permet convertir qualsevol binari x86-64 a RISC-V.

1. **Traduir el compilador de C a RISC-V**: S'aplica el traductor x86-64→RISC-V (element 2) al compilador de C (element 1, binari x86-64). S'obté un compilador de C **executable en RISC-V**, però que genera codi x86-64 (és un **compilador creuat**: s'executa en RISC-V i produeix binaris x86-64).
2. **Compilar l'intèrpret de Python**: S'usa el compilador del pas 1 per compilar l'intèrpret de Python (element 3, escrit en C). S'obté un binari x86-64 de l'intèrpret de Python.
3. **Traduir l'intèrpret de Python a RISC-V**: S'aplica de nou el traductor x86-64→RISC-V (element 2) al binari x86-64 del pas 2. S'obté un intèrpret de Python **executable en RISC-V**.
4. **Executar el programa Python**: S'usa l'intèrpret del pas 3 per executar el programa en Python a la màquina RISC-V.

💡 Exemple 2: Solució: executar C en x86-64

Enunciat: Problema 1.2.
Tots els passos s'executen en la **màquina x86-64**, ja que el traductor (element 3) és un binari x86-64 i s'hi pot executar directament.
a)

1. **Traduir l'intèrpret de Python a x86-64**: S'aplica el traductor RISC-V→x86-64 (element 3, escrit en x86-64) a l'intèrpret de Python (element 1, escrit en RISC-V). S'obté un intèrpret de Python **executable en x86-64**.
2. **Compilar el programa en C**: El compilador de C (element 2) és un programa escrit en Python, que s'executa amb l'intèrpret del pas 1 i compila el programa en C. S'obté un binari x86-64 del programa.
3. **Executar el binari**: S'executa directament el binari resultant a la màquina x86-64.

b) Per tornar a executar **el mateix programa**: si el binari del pas 2 s'ha conservat, s'executa directament (pas 3) sense repetir cap pas anterior. Si s'ha esborrat, cal repetir únicament el pas 2 (l'intèrpret del pas 1 es pot conservar entre execucions).

1.2 Codificació de nombres naturals i enters

Exemple 3: Solució: interpretació i representació de naturals

Enunciat: Problema 1.3.
a) Aplicant Equació 1.1:

Patró de bits	n	Valor decimal
`0001 1001`	8	$2^4 + 2^3 + 2^0 = 16 + 8 + 1 = 25$
`1010 0110`	8	$2^7 + 2^5 + 2^2 + 2^1 = 128 + 32 + 4 + 2 = 166$
`0000 0000 1010 1101`	16	$2^7 + 2^5 + 2^3 + 2^2 + 2^0 = 128 + 32 + 8 + 4 + 1 = 173$

b) Per divisions successives entre 2 (residus de menor a major pes):
$25 \div 2$: residus $1, 0, 0, 1, 1, 0, 0, 0$ (d'LSb a MSb) $\rightarrow 0001\,1001_2$. ✓
$173 \div 2$: residus $1, 0, 1, 1, 0, 1, 0, 1 \rightarrow 1010\,1101_2$. ✓
256 en 8 bits: $256 > 2^8 - 1 = 255$. **NC** (no codificable en 8 bits).
173 en 16 bits: extensió de zeros (Secció 1.4.4) de $1010\,1101_2 \rightarrow 0000\,0000\,1010\,1101_2$. ✓

Decimal	n	Binari
25	8	`0001 1001`
173	8	`1010 1101`
256	8	NC
173	16	`0000 0000 1010 1101`

Exemple 4: Solució: menor enter en Ca2 de 13 bits

Enunciat: Problema 1.4.
Amb $n = 13$ bits, el rang de Ca2 és $[-2^{12},\ 2^{12} - 1] = [-4096,\ +4095]$.
L'enter més petit (el més negatiu) és $x_s = -4096 = -2^{12}$.
a) En hexadecimal: el patró de bits és $x_u = x_s + 2^{13} = -4096 + 8192 = 2^{12} = 1\,0000\,0000\,0000_2$ (13 bits; Essencial 1.2). Completant aquest **patró** amb zeros a l'esquerra fins a 16 bits per poder-lo expressar en hexadecimal: `0x1000`.
b) En decimal: -4096.
Atenció: el valor `0x1000` anterior és només el patró de 13 bits escrit en hexadecimal, **no** l'extensió de signe a 16 bits. Si es vol representar -4096 directament amb 16 bits en Ca2, cal aplicar l'extensió de signe (Secció 1.5.1.3): com que el bit de signe és 1, es repliquen tres 1 addicionals a l'esquerra, i el resultat és `0xF000` (no `0x1000`, que en Ca2 de 16 bits representaria $+4096$).

Exemple 5: Solució: major enter en Ca2 de 13 bits

Enunciat: Problema 1.5.
L'enter positiu més gran és $x_s = 2^{12} - 1 = 4095$.
a) En hexadecimal: $4095 = 0\,1111\,1111\,1111_2 =$ `0x0FFF`.
b) En decimal: $+4095$.

💡 Exemple 6: Solució: rang i taula de codificació de formats d'enters

Enunciat: Problema 1.6.
a) Rang i representació del zero per a n bits:

Format	Rang	Representació del zero
Signe i magnitud	$[-(2^{n-1}-1),\ 2^{n-1}-1]$	`0...00` i `1...00` (doble zero)
Complement a 1	$[-(2^{n-1}-1),\ 2^{n-1}-1]$	`0...00` i `1...11` (doble zero)
Complement a 2	$[-2^{n-1},\ 2^{n-1}-1]$	`0...00`
Excés a $2^{n-1}-1$	$[-(2^{n-1}-1),\ 2^{n-1}]$	`0111...1` ($n-1$ uns)

b) Taula de codificació en 4 bits ($n=4$). L'excés és $2^{n-1}-1=7$, de manera que el codi d'excés és `decimal + 7`.

Decimal	Ca2	Ca1	Excés a 7	Signe i magnitud
8	NC	NC	`1111`	NC
7	`0111`	`0111`	`1110`	`0111`
4	`0100`	`0100`	`1011`	`0100`
1	`0001`	`0001`	`1000`	`0001`
0	`0000`	`0000` i `1111`	`0111`	`0000` i `1000`
-1	`1111`	`1110`	`0110`	`1001`
-4	`1100`	`1011`	`0011`	`1100`
-7	`1001`	`1000`	`0000`	`1111`
-8	`1000`	NC	NC	NC

Observacions:

- El 8 **no** és representable en Ca2 (rang $[-8,7]$) ni en Ca1 ni en signe i magnitud (rang $[-7,7]$ en tots dos). Sí que és representable en excés a 7 (rang $[-7,8]$): el seu codi és $8+7=15=\texttt{1111}_2$.
- El -8 no és representable en Ca1, en excés a 7 ni en signe i magnitud (el mínim en tots tres formats és -7); sí que ho és en Ca2 (codi `1000`).
- En Ca1 i signe i magnitud, el 0 té dues representacions.

💡 Exemple 7: Solució: conversió a Ca1, Ca2, signe i magnitud i excés a 127 (8 bits)

Enunciat: Problema 1.7.
Per a cada valor, el procediment és:

- **Ca2**: els positius coincideixen amb la representació binària natural; els negatius, $x+2^8=x+256$ (Essencial 1.2).
- **Ca1**: els positius coincideixen amb Ca2; els negatius s'obtenen complementant tots els bits de la representació de $|x|$.
- **Signe i magnitud** (S&M): bit de signe (0 si positiu, 1 si negatiu) i $|x|$ als bits restants. Rang: $[-127,+127]$.
- **Excés a 127**: codi $=x+127$. Rang: $[-127,+128]$.

$x=-78$
$|-78|=78=0100\,1110_2$.

- Ca2: $256-78=178=1011\,0010_2\rightarrow$ `0xB2`

- Ca1: $\overline{0100\,1110} = 1011\,0001_2 \rightarrow$ `0xB1`
- S&M: bit signe $= 1$; $78 = 0100\,1110_2 \rightarrow 1100\,1110_2 \rightarrow$ `0xCE`
- Excés: $-78 + 127 = 49 = 0011\,0001_2 \rightarrow$ `0x31`

$x = +125$
$125 = 0111\,1101_2$.

- Ca2, Ca1, S&M: `0x7D` (coincideixen per als positius)
- Excés: $125 + 127 = 252 = 1111\,1100_2 \rightarrow$ `0xFC`

$x = 0$

- Ca2, Ca1, S&M: `0x00`
- Excés: $0 + 127 = 127 = 0111\,1111_2 \rightarrow$ `0x7F`

$x = -1$
$|-1| = 1 = 0000\,0001_2$.

- Ca2: $256 - 1 = 255 = 1111\,1111_2 \rightarrow$ `0xFF`
- Ca1: $\overline{0000\,0001} = 1111\,1110_2 \rightarrow$ `0xFE`
- S&M: $1000\,0001_2 \rightarrow$ `0x81`
- Excés: $-1 + 127 = 126 = 0111\,1110_2 \rightarrow$ `0x7E`

Resum:

Decimal	Ca1	Ca2	S&M	Excés a 127
-78	`0xB1`	`0xB2`	`0xCE`	`0x31`
$+125$	`0x7D`	`0x7D`	`0x7D`	`0xFC`
0	`0x00`	`0x00`	`0x00`	`0x7F`
-1	`0xFE`	`0xFF`	`0x81`	`0x7E`

💡 Exemple 8: Solució: interpretació com a natural i Ca2

Enunciat: Problema 1.8.
Aplicant la drecera $x_s = x_u - 2^n$ per als patrons amb bit de signe actiu (Essencial 1.2):
a) `0000 1111` (8 bits)
El bit de signe és $0 \rightarrow$ el valor és el mateix en Ca2 que en natural.

- Natural: $2^3 + 2^2 + 2^1 + 2^0 = 8 + 4 + 2 + 1 = 15$
- Ca2: $+15$

b) `1000 0000` (8 bits)
Natural: $2^7 = 128$.
Ca2: bit de signe $= 1 \rightarrow 128 - 2^8 = 128 - 256 = -128$.
c) `1111 1111` (8 bits)
Natural: $2^7 + 2^6 + \ldots + 2^0 = 255$.
Ca2: $255 - 256 = -1$.
Verificació: (-1) en Ca2 $= \overline{0000\,0001} + 1 = 1111\,1110 + 1 = 1111\,1111$ ✓
d) `1111 1111 1111 0000` (16 bits)
Natural: $2^{16} - 2^4 = 65\,536 - 16 = 65\,520$.
Ca2: $65\,520 - 65\,536 = -16$.

Verificació: (-16) en Ca2 (16 bits) $= 65\,536 - 16 = 65\,520 = 1111\,1111\,1111\,0000_2$ ✓

Patró	Natural	Ca2
`0000 1111`	15	$+15$
`1000 0000`	128	-128
`1111 1111`	255	-1
`1111 1111 1111 0000`	65 520	-16

Exemple 9: Solució: extensió de signe a 16 bits

Enunciat: Problema 1.9.
Per aplicar l'extensió de signe, es replica el bit de signe (X_{n-1}) a totes les posicions noves a l'esquerra. El valor representat no canvia.
a) `0011 0101` (8 bits)
Bit de signe $= 0 \rightarrow$ nombre positiu. Valor: $32 + 16 + 4 + 1 = +53$.
Extensió de signe a 16 bits: es repliquen vuit 0:

$$0011\,0101_2 \rightarrow 0000\,0000\,0011\,0101_2 = \texttt{0x0035}$$

Comprovació: `0x0035` $= 53$ ✓
b) `1100 1101` (8 bits)
Bit de signe $= 1 \rightarrow$ nombre negatiu. Valor (per Equació 1.4): $-128 + 64 + 8 + 4 + 1 = -51$.
Extensió de signe a 16 bits: es repliquen vuit 1:

$$1100\,1101_2 \rightarrow 1111\,1111\,1100\,1101_2 = \texttt{0xFFCD}$$

Comprovació: `0xFFCD` representa $65\,485 - 65\,536 = -51$ ✓

Exemple 10: Solució: Ca2 en 16 bits → decimal

Enunciat: Problema 1.10.
a) `0x0123`
`0x0123` $= 0000\,0001\,0010\,0011_2$. El bit de signe (bit 15) és $0 \rightarrow$ nombre positiu.

$$x_s = 1{\cdot}2^8 + 1{\cdot}2^5 + 1{\cdot}2^1 + 1{\cdot}2^0 = 256 + 32 + 2 + 1 = +291$$

b) `0xFE23`
`0xFE23` $= 1111\,1110\,0010\,0011_2$. El bit de signe (bit 15) és $1 \rightarrow$ nombre negatiu.
Aplicant la drecera $x_s = x_u - 2^n$ de Essencial 1.2 (equivalent a Equació 1.4), amb $x_u =$ `0xFE23` $= 15 \cdot 4096 + 14 \cdot 256 + 2 \cdot 16 + 3 = 65\,059$:

$$x_s = 65\,059 - 2^{16} = 65\,059 - 65\,536 = -477$$

Verificació: $|-477| = 477 = 0000\,0001\,1101\,1101_2$; complementant i sumant 1: $\overline{0000\,0001\,1101\,1101} + 1 = 1111\,1110\,0010\,0010 + 1 = 1111\,1110\,0010\,0011 =$ `0xFE23` ✓

Exemple 11: Solució: Ca2 en el format mínim (8 o 16 bits)

Enunciat: Problema 1.11.
El rang de Ca2 amb 8 bits és $[-128, +127]$; amb 16 bits, $[-32\,768, +32\,767]$. S'usa 8 bits si el valor hi cap; altrament, 16 bits. Un negatiu x de n bits es codifica com el natural $2^n + x$: per

exemple, $-127 \to 256 - 127 = 129 = 1000\,0001_2$.

Valor	Format	Ca2	Hex
$+123$	8 bits	0111 1011	`0x7B`
$+255$	16 bits	0000 0000 1111 1111	`0x00FF`
-127	8 bits	1000 0001	`0x81`
$+170$	16 bits	0000 0000 1010 1010	`0x00AA`
$+128$	16 bits	0000 0000 1000 0000	`0x0080`
-230	16 bits	1111 1111 0001 1010	`0xFF1A`
-18	8 bits	1110 1110	`0xEE`
-128	8 bits	1000 0000	`0x80`
-1	8 bits	1111 1111	`0xFF`

Nota sobre $+255$: calen 16 bits perquè $255 > 127$ (màxim de Ca2 en 8 bits). El valor $1111\,1111_2$ en 8 bits representaria -1 en Ca2, no $+255$.
Nota sobre -230: $-230 < -128$, per tant, calen 16 bits. Càlcul: $65\,536 - 230 = 65\,306 = 1111\,1111\,0001\,1010_2$.

1.3 Operacions aritmètiques en complement a 2

Exemple 12: Solució: sumes en Ca2 (8 bits)

Enunciat: Problema 1.12.
Rang Ca2 de 8 bits: $[-128, +127]$. Hi ha desbordament si els dos operands tenen el mateix signe i el resultat té signe contrari.
a) $45 + 30 = 75$

$$\underbrace{0010\,1101}_{+45} + \underbrace{0001\,1110}_{+30} = \underbrace{0100\,1011}_{+75}$$

Resultat $+75 \in [-128, +127]$. **Sense desbordament** ✓
b) $100 + 80$

$$\underbrace{0110\,0100}_{+100} + \underbrace{0101\,0000}_{+80} = \underbrace{1011\,0100}_{?}$$

$1011\,0100_2$ s'interpreta com a -76 en Ca2. Tots dos operands positius i resultat aparentment negatiu: **desbordament** ($100 + 80 = 180 > 127$).
c) $(-45) + (-30) = -75$
Ca2 (Secció 1.6.2): $-45 \to 1101\,0011$; $-30 \to 1110\,0010$.

$$\underbrace{1101\,0011}_{-45} + \underbrace{1110\,0010}_{-30} = \underbrace{1011\,0101}_{-75}$$

(El ròssec de sortida (*carry-out*) es descarta.) $1011\,0101_2 = -75$ en Ca2. **Sense desbordament** ✓
d) $(-100) + (-80)$
Ca2 (Secció 1.6.2): $-100 \to 1001\,1100$; $-80 \to 1011\,0000$.

$$\underbrace{1001\,1100}_{-100} + \underbrace{1011\,0000}_{-80} = \underbrace{0100\,1100}_{?}$$

$0100\,1100_2 = +76$ en Ca2. Tots dos operands negatius i resultat aparentment positiu: **desbordament** ($-180 < -128$).
e) $45 + (-30) = +15$
Ca2: $-30 \to 1110\,0010$.

$$\underbrace{0010\,1101}_{+45} + \underbrace{1110\,0010}_{-30} = \underbrace{0000\,1111}_{+15}$$

(El ròssec final es descarta.) Els operands tenen signes contraris: **no hi pot haver desbordament** quan els signes dels operands difereixen. Resultat: $+15$ ✓

Exemple 13: Solució: restes en Ca2 (8 bits)

Enunciat: Problema 1.13.
La resta $a - b$ es calcula com $a + (-b)$: es nega b (complement de tots els bits i suma 1) i se suma.
a) $45 - 30 = +15$
-30 en Ca2: $\overline{0001\,1110} + 1 = 1110\,0001 + 1 = 1110\,0010$.

$$\underbrace{0010\,1101}_{+45} + \underbrace{1110\,0010}_{-30} = \underbrace{0000\,1111}_{+15}$$

Sense desbordament ✓
b) $30 - 45 = -15$
-45 en Ca2: $\overline{0010\,1101} + 1 = 1101\,0010 + 1 = 1101\,0011$.

$$\underbrace{0001\,1110}_{+30} + \underbrace{1101\,0011}_{-45} = \underbrace{1111\,0001}_{-15}$$

$1111\,0001_2 = -15$ en Ca2. **Sense desbordament** ✓
c) $(-100) - 45 = -145$
-45 en Ca2: $1101\,0011$ (calculat a l'apartat b).

$$\underbrace{1001\,1100}_{-100} + \underbrace{1101\,0011}_{-45} = \underbrace{0110\,1111}_{?}$$

$0110\,1111_2 = +111$ en Ca2. Tots dos operands negatius i resultat aparentment positiu: **desbordament** ($-145 < -128$).

Exemple 14: Solució: multiplicacions en Ca2 (8 bits)

Enunciat: Problema 1.14.
Mètode: (1) signe del resultat (positiu si els signes coincideixen, negatiu si difereixen), (2) calcular el producte de les magnituds, (3) aplicar el signe i codificar en Ca2.
a) $6 \times 5 = +30$
Magnituds: $6 \times 5 = 30$. Signe: $(+) \times (+) \to (+)$.
$30 = 0001\,1110_2 =$ `0x1E`. $30 \in [-128, +127]$: **sense desbordament** ✓
b) $(-6) \times 5 = -30$
Magnituds: $6 \times 5 = 30$. Signe: $(-) \times (+) \to (-)$.
Ca2 de -30: $\overline{0001\,1110} + 1 = 1110\,0001 + 1 = 1110\,0010_2 =$ `0xE2`. **Sense desbordament** ✓
c) $(-6) \times (-5) = +30$
Magnituds: $6 \times 5 = 30$. Signe: $(-) \times (-) \to (+)$.
Igual que l'apartat a): `0x1E`. **Sense desbordament** ✓
d) $20 \times 10 = +200$
Magnituds: $20 \times 10 = 200$. Signe: $(+) \times (+) \to (+)$.

$200 > 127$: **desbordament**. El resultat no és representable en Ca2 de 8 bits (caldrien almenys 9 bits; en Ca2 de 16 bits: $200 = \texttt{0x00C8}$).

Exemple 15: Solució: divisions en Ca2 (8 bits)

Enunciat: Problema 1.15.
Mètode: (1) signe del quocient (positiu si els signes coincideixen, negatiu si difereixen), (2) dividir les magnituds com a naturals, (3) aplicar el signe al quocient; el residu pren el signe del **dividend**.
a) $30 \div 5$
Magnituds: $30 \div 5 = 6$, residu 0. Signe: $(+) \div (+) \to (+)$.
Quocient $= +6 \to 0000\,0110_2$; residu $= 0 \to 0000\,0000_2$. **Sense desbordament** ✓
b) $(-30) \div 5$
Magnituds: $30 \div 5 = 6$, residu 0. Signe: $(-) \div (+) \to (-)$.
Ca2(-6) : $\overline{0000\,0110} + 1 = 1111\,1010$.
Quocient $= -6 \to 1111\,1010_2$; residu $= 0 \to 0000\,0000_2$. **Sense desbordament** ✓
c) $(-30) \div (-5)$
Magnituds: $30 \div 5 = 6$, residu 0. Signe: $(-) \div (-) \to (+)$.
Quocient $= +6 \to 0000\,0110_2$; residu $= 0 \to 0000\,0000_2$. **Sense desbordament** ✓
d) $(-7) \div 2$
Magnituds: $7 \div 2 = 3$, residu 1. Signe quocient: $(-) \div (+) \to (-)$. Signe residu: igual que el dividend $\to (-)$.
Ca2(-3) : $\overline{0000\,0011} + 1 = 1111\,1101$. Ca2$(-1)$: $\overline{0000\,0001} + 1 = 1111\,1111$.
Quocient $= -3 \to 1111\,1101_2$; residu $= -1 \to 1111\,1111_2$.
Comprovació: $2 \times (-3) + (-1) = -7$ ✓
e) $(-128) \div (-1)$
Magnituds: $128 \div 1 = 128$, residu 0. Signe: $(-) \div (-) \to (+)$.
El quocient matemàtic és $+128$, però $128 > 127 = 2^7 - 1$: **desbordament**. El valor $+128$ no és representable en Ca2 de 8 bits (rang $[-128, +127]$). És l'únic cas de desbordament possible en la divisió Ca2 de n bits: quan el dividend és el mínim (-2^{n-1}) i el divisor és -1.

Tema 2: Instruccions i tipus de dades bàsics

2.1 Operands en registre

Exemple 16: Solució: traducció de C a assemblador RISC-V (operands en registre)

Enunciat: Problema 2.1.
Variables: `f`→`t0`, `g`→`t1`, `h`→`t2`, `i`→`t3`, `j`→`t4`.
a) `f = g + h + i + j`
Calen tres `add` encadenats; no existeix cap `add` de quatre operands:

RV32I

```
add     t0, t1, t2  # t0 = g + h
add     t0, t0, t3  # t0 = g + h + i
add     t0, t0, t4  # f  = g + h + i + j
```

b) `g = f + (h + 5) - i`
Es calcula primer `h + 5` (amb `addi`), s'afegeix `f` i finalment es resta `i`:

RV32I

```
addi    t5, t2, 5   # t5 = h + 5
add     t5, t0, t5  # t5 = f + (h + 5)
sub     t1, t5, t3  # g  = f + (h + 5) - i
```

El registre temporal (`t5`) no és imprescindible: com que el valor inicial de `g` (`t1`) no intervé en el càlcul, els resultats parcials es podrien acumular directament a `t1` sense sobreescriure `f` (`t0`) ni `h` (`t2`).

Exemple 17: Solució: desassemblar fragments a C

Enunciat: Problema 2.2.
a)

RV32I

```
add     t0, t1, t2
add     t1, t3, t4
add     t1, t1, t1
add     t0, t0, t1
```

Instrucció per instrucció:

1. `add t0, t1, t2` → `t0 = t1 + t2` → `f = g + h`
2. `add t1, t3, t4` → `t1 = t3 + t4` → `g = i + j`
3. `add t1, t1, t1` → `t1 = t1 + t1 = 2·t1` → `g = 2 * (i + j)`

4. `add t0, t0, t1` → `t0 = t0 + t1` → `f = (g + h) + 2 * (i + j)`

Resposta: `f = (g + h) + 2 * (i + j)`
Nota: el codi també modifica `g` (`t1`), que queda amb el valor `2 * (i + j)`. La resposta anterior dona `f` en funció dels valors **inicials** de `g`, `h`, `i`, `j`.
b)

`RV32I`

```
        add     t1, t1, t2
        add     t3, t3, t4
        add     t0, t1, t3
        addi    t0, t0, 2
        add     t0, t0, t0
```

Instrucció per instrucció:

1. `add t1, t1, t2` → `g = g + h`
2. `add t3, t3, t4` → `i = i + j`
3. `add t0, t1, t3` → `f = (g + h) + (i + j)`
4. `addi t0, t0, 2` → `f = (g + h) + (i + j) + 2`
5. `add t0, t0, t0` → `f = 2 * ((g + h) + (i + j) + 2)`

Resposta: `f = 2 * (g + h + i + j + 2)`
Nota: el codi també modifica `g` (`t1`, que queda amb `g + h`) i `i` (`t3`, que queda amb `i + j`). La resposta dona `f` en funció dels valors **inicials** de `g`, `h`, `i`, `j`.

2.2 Constants i immediats

Exemple 18: Solució: expansió de `li` en `lui` + `addi` i rang d'immediats

Enunciat: Problema 2.5.
a) Seqüència per carregar `0xDEADBEEF` a `t0`.
El valor `0xDEADBEEF` no cap en 12 bits, de manera que calen dues instruccions. Es divideix en:

- **20 bits superiors** (hi_{20}): `0xDEADB`.
- **12 bits inferiors** (lo_{12}): `0xEEF` $= 1110\,1110\,1111_2$.

El bit 11 de lo_{12} val 1, de manera que `addi` aplicarà extensió de signe i restarà `0x111` (273 en positiu). Cal incrementar hi_{20} en 1 per compensar:

- hi_{20} ajustat: `0xDEADB + 1 = 0xDEADC`.
- lo_{12} com a enter de 12 bits amb signe: `0xEEF - 0x1000 = -273`.

`RV32I`

```
        lui     t0, 0xDEADC     # t0 = 0xDEADC000
        addi    t0, t0, -273    # t0 = 0xDEADC000 - 273 = 0xDEADBEEF
```

Comprovació: `0xDEADC000 - 0x111 = 0xDEADBEEF` ✓
b) Per què cal ajustar hi_{20} quan el bit 11 de lo_{12} val 1.

`addi` fa **extensió de signe** de l'immediat de 12 bits. Si el bit 11 és 1, el valor s'interpreta com a enter negatiu: el valor efectiu és $lo_{12} - \texttt{0x1000}$ (en lloc de lo_{12}). Sense l'ajust, el resultat seria 0x1000 (4096) menys del valor desitjat.
Per `0xDEADBEEF`: sense ajust, `lui t0, 0xDEADB` → `t0 = 0xDEADB000`; `addi t0, t0, 0xEEF` → l'extensió de signe interpreta `0xEEF` com `0xEEF` − `0x1000` = −273 → `0xDEADB000 - 273 = 0xDEADAEEF` ≠ `0xDEADBEEF`. Amb l'ajust `hi20 = 0xDEADC`: `0xDEADC000 - 273 = 0xDEADBEEF` ✓
c) Rangs de valors.

- **Amb una sola instrucció `addi`**: l'immediat és de 12 bits amb extensió de signe → rang $[-2048,\ 2047]$.
- **Amb `lui` + `addi`**: `lui` carrega 20 bits als bits [31:12]; `addi` afegeix fins a 12 bits amb signe. El rang cobert és $[-2^{31},\ 2^{31}-1]$, és a dir, tots els enters representables en 32 bits.

2.3 Operands en memòria

Exemple 19: Solució: seguiment d'instruccions `lw` i `sw`

Enunciat: Problema 2.7.
Estat inicial: `t0 = 0x10010000`, `t2 = 0x10010008`.
Contingut de memòria inicial (*little-endian*):

Adreça	+0	+1	+2	+3
0x10010000	00	01	02	03
0x10010004	04	05	06	07
0x10010008	08	09	0A	0B
0x1001000C	0C	0D	0E	0F
0x10010010	00	00	00	00
0x10010014	EC	FD	0E	0F

Seguiment:
`lw t1, 0(t0)` — adreça efectiva: `0x10010000 + 0 = 0x10010000`
Llegeix 4 bytes a partir de `0x10010000` (*little-endian*): bytes `00 01 02 03` → `t1 = 0x03020100`.
`lw t3, 0(t2)` — adreça efectiva: `0x10010008 + 0 = 0x10010008`
Llegeix 4 bytes a partir de `0x10010008`: bytes `08 09 0A 0B` → `t3 = 0x0B0A0908`.
`sw t3, -8(t2)` — adreça efectiva: `0x10010008 - 8 = 0x10010000`
Escriu `t3 = 0x0B0A0908` a partir de `0x10010000` (*little-endian*): bytes `08 09 0A 0B`.
Memòria modificada: `M[0x10010000..0x10010003] = 08 09 0A 0B`.
`lw t4, 4(t0)` — adreça efectiva: `0x10010000 + 4 = 0x10010004`
Llegeix 4 bytes a partir de `0x10010004`: bytes `04 05 06 07` → `t4 = 0x07060504`.
`sw t1, -4(t2)` — adreça efectiva: `0x10010008 - 4 = 0x10010004`
Escriu `t1 = 0x03020100` a partir de `0x10010004`: bytes `00 01 02 03`.
Memòria modificada: `M[0x10010004..0x10010007] = 00 01 02 03`.
`sw t4, 0(t2)` — adreça efectiva: `0x10010008 + 0 = 0x10010008`
Escriu `t4 = 0x07060504` a partir de `0x10010008`: bytes `04 05 06 07`.
Memòria modificada: `M[0x10010008..0x1001000B] = 04 05 06 07`.
Resum de resultats:

Operació	Destí	Valor resultant
`lw t1, 0(t0)`	`t1`	`0x03020100`
`lw t3, 0(t2)`	`t3`	`0x0B0A0908`
`sw t3, -8(t2)`	`M[0x10010000]`	`08 09 0A 0B`
`lw t4, 4(t0)`	`t4`	`0x07060504`
`sw t1, -4(t2)`	`M[0x10010004]`	`00 01 02 03`
`sw t4, 0(t2)`	`M[0x10010008]`	`04 05 06 07`

Exemple 20: Solució: *little-endian* i ordre dels bytes

Enunciat: Problema 2.10.
a) Significat de *little-endian*.
En una arquitectura *little-endian*, el byte de **menys pes** (*least significant byte*, LSB) d'una dada de més d'un byte s'emmagatzema a l'adreça **més baixa**, i el byte de **més pes** (*most significant byte*, MSB) a l'adreça més alta (vegeu Secció 2.5.1). Per a un enter de 32 bits, doncs, els 4 bytes s'emmagatzemen en l'ordre invers al que s'escriu el valor en hexadecimal: el parell de dígits de la dreta (LSB) ocupa la primera adreça, i així successivament.
b) `int x = 0x12345678` a partir de `0x10010000`.
Els bytes del valor, de l'MSB a l'LSB, són `12 34 56 78`. En *little-endian* s'emmagatzemen començant per l'LSB:

Adreça	Contingut	Byte
`0x10010000`	`78`	LSB
`0x10010001`	`56`	
`0x10010002`	`34`	
`0x10010003`	`12`	MSB

c) Valor de `t1`.
`lb t1, 0(t0)` llegeix **un sol byte** de l'adreça `0x10010000`, que en *little-endian* conté el byte de menys pes de `x`: `0x78`. El bit 7 de `0x78` és 0, de manera que l'extensió de signe afegeix zeros:
`t1 = 0x00000078`
d) Resultat en *big-endian*.
En *big-endian*, l'adreça més baixa (`0x10010000`) contindria el byte de **més pes**: `0x12`. La mateixa instrucció retornaria `t1 = 0x00000012` (el bit 7 de `0x12` també és 0). El mateix codi pot produir, doncs, resultats diferents segons l'ordre dels bytes de la màquina quan s'accedeix a fragments d'una dada de més d'un byte.

Exemple 21: Solució: instruccions de lectura i escriptura de byte i mitja paraula

Enunciat: Problema 2.11.
Estat inicial: `t0 = 0x10010000`.

Adreça	+0	+1	+2	+3
0x10010000	11	22	CC	DD

a) Resultat de cada instrucció (partint sempre del contingut inicial):
`lb t1, 1(t0)` — adreça efectiva `0x10010001`, byte = `0x22`.
El bit 7 de `0x22` és 0 → extensió de signe afegeix zeros: `t1 = 0x00000022`.
`lb t1, 2(t0)` — adreça efectiva `0x10010002`, byte = `0xCC`.
El bit 7 de `0xCC` és 1 → extensió de signe afegeix uns: `t1 = 0xFFFFFFCC`.
`lh t1, 0(t0)` — adreça efectiva `0x10010000`, dos bytes (*little-endian*): `11 22` → `0x2211`.
El bit 15 de `0x2211` és 0 → extensió de signe: `t1 = 0x00002211`.
`lh t1, 2(t0)` — adreça efectiva `0x10010002`, dos bytes: `CC DD` → `0xDDCC`.
El bit 15 de `0xDDCC` és 1 → extensió de signe: `t1 = 0xFFFFDDCC`.
`lbu t1, 2(t0)` — adreça efectiva `0x10010002`, byte = `0xCC`.
Extensió **per zero** (sense signe): `t1 = 0x000000CC`.
`lhu t1, 2(t0)` — adreça efectiva `0x10010002`, dos bytes: `CC DD` → `0xDDCC`.
Extensió **per zero**: `t1 = 0x0000DDCC`.
Resum:

Instrucció	Resultat al registre
`lb t1, 1(t0)`	`t1 = 0x00000022`
`lb t1, 2(t0)`	`t1 = 0xFFFFFFCC`
`lh t1, 0(t0)`	`t1 = 0x00002211`
`lh t1, 2(t0)`	`t1 = 0xFFFFDDCC`
`lbu t1, 2(t0)`	`t1 = 0x000000CC`
`lhu t1, 2(t0)`	`t1 = 0x0000DDCC`

b) Execució seqüencial:

`RV32I`

```
lb      t1, 2(t0)   # t1 = SignExt(0xCC) = 0xFFFFFFCC
sb      t1, 1(t0)   # M[0x10010001] <- byte LSB de t1 = 0xCC
```

`sb` escriu **només el byte menys significatiu** del registre font (0xCC), sense extensió de cap mena.
Estat final de la memòria:

Adreça	+0	+1	+2	+3
`0x10010000`	`11`	`CC` ←	`CC`	`DD`

2.4 Representació de naturals i enters

Vegeu les solucions de Capítol 1.

2.5 Representació de caràcters

Exemple 22: Solució: contingut de memòria amb tipus mixtos

Enunciat: Problema 2.13.
Declaracions, a partir de `0x10010000`:

RV32I

```
p:      .half   54
q:      .byte   'C'
r:      .half   0xAAAA
s:      .word   0x4F3C
t:      .byte   -18
u:      .dword  0x00000000FFFFEEEE
```

Càlcul adreça per adreça, amb les restriccions d'alineació de l'ABI (l'assemblador afegeix **farciment** (*padding*) per respectar-les). El `.dword u` s'alinea a 8 bytes; aquí l'alineació de RARS 1.6, a 4 bytes, donaria la mateixa adreça:

Variable	Tipus	Mida	Adreça	Farciment previ
`p`	`.half`	2 B	`0x10010000`	—
`q`	`.byte`	1 B	`0x10010002`	—
`r`	`.half`	2 B	`0x10010004`	1 B
`s`	`.word`	4 B	`0x10010008`	2 B
`t`	`.byte`	1 B	`0x1001000C`	—
`u`	`.dword`	8 B	`0x10010010`	3 B

Valors byte a byte (*little-endian*):

- `p = 54 = 0x0036` → `36 00`
- `q = 'C' = 0x43` → `43`
- farciment: `??` (indeterminat)
- `r = 0xAAAA` → `AA AA`
- farciment: `?? ??`
- `s = 0x00004F3C` → `3C 4F 00 00`
- `t = -18 = 0xEE` (Ca2 de 8 bits) → `EE`
- farciment: `?? ?? ??`
- `u = 0x00000000FFFFEEEE` (`.dword` de 64 bits) → `EE EE FF FF 00 00 00 00`

Adreça	Contingut (byte a byte, hex)
`0x10010000`	`36 00 43 ?? AA AA ?? ?? 3C 4F 00 00 EE ?? ?? ??`
`0x10010010`	`EE EE FF FF 00 00 00 00`

Els bytes marcats `??` corresponen a farciment i el seu valor és indeterminat.

2.6 Format de les instruccions

Exemple 23: Solució: assemblador → codi màquina

Enunciat: Problema 2.16.
Per resoldre cada instrucció cal identificar el format, localitzar els camps i concatenar els bits. Els números de registre ABI es converteixen als corresponents ISA:

ABI	ISA	ABI	ISA
t4	x29 = 11101	t3	x28 = 11100
t5	x30 = 11110	t2	x7 = 00111
t6	x31 = 11111	t0	x5 = 00101
t1	x6 = 00110		

`add t4, t3, t5` — Tipus R, `opcode = 0x33 = 0110011`, `funct3 = 000`, `funct7 = 0000000`

funct7	rs2	rs1	funct3	rd	opcode
`0000000`	`11110`	`11100`	`000`	`11101`	`0110011`

Binari: `0000000 11110 11100 000 11101 0110011`
Hexadecimal: `0x01EE0EB3`

`addi t6, t5, 25` — Tipus I, `opcode = 0x13 = 0010011`, `funct3 = 000`
`25 = 0x019 = 000000011001` (12 bits)

imm[11:0]	rs1	funct3	rd	opcode
`000000011001`	`11110`	`000`	`11111`	`0010011`

Binari: `000000011001 11110 000 11111 0010011`
Hexadecimal: `0x019F0F93`

`lw t3, 0(t2)` — Tipus I, `opcode = 0x03 = 0000011`, `funct3 = 010`
`offset = 0 = 000000000000` (12 bits)

imm[11:0]	rs1	funct3	rd	opcode
`000000000000`	`00111`	`010`	`11100`	`0000011`

Binari: `000000000000 00111 010 11100 0000011`
Hexadecimal: `0x0003AE03`

`sw t0, 0(t1)` — Tipus S, `opcode = 0x23 = 0100011`, `funct3 = 010`
`offset = 0`; `imm[11:5]` = `0000000`, `imm[4:0]` = `00000`

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
`0000000`	`00101`	`00110`	`010`	`00000`	`0100011`

Binari: `0000000 00101 00110 010 00000 0100011`
Hexadecimal: `0x00532023`

Resum:

Assemblador RISC-V	Binari	Hexadecimal
`add  t4, t3, t5`	`0000000 11110 11100 000 11101 0110011`	`0x01EE0EB3`
`addi t6, t5, 25`	`000000011001 11110 000 11111 0010011`	`0x019F0F93`
`lw   t3, 0(t2)`	`000000000000 00111 010 11100 0000011`	`0x0003AE03`
`sw   t0, 0(t1)`	`0000000 00101 00110 010 00000 0100011`	`0x00532023`

2.7 Punters

Exemple 24: Solució: traducció de sentències amb vector i punter

Enunciat: Problema 2.21.
Declaracions:

C

```c
int vec[6] = {2, 4, 0, 6, 8, 0};
int *punter;
```

`vec` és global → adreça base accessible via `la`. `punter` és global → cal fer `la` + `lw`/`sw`.

---

a) `punter = &vec[2];`
`&vec[2] = @vec + 2*4 = @vec + 8`:

RV32I

```
        la      t0, punter
        la      t1, vec+8       # &vec[2]
        sw      t1, 0(t0)       # punter = &vec[2]
```

Nota: la sintaxi `la label+offset` és admesa a EC (EC 2.3), però a RARS cal substituir-la per `la t1, vec` seguit d'`addi t1, t1, 8` (EC 2.6).

---

b) `punter = punter + 1;`
El punter apunta a `int` (mida 4 bytes), de manera que `punter + 1` incrementa en 4:

RV32I

```
        la      t0, punter
        lw      t1, 0(t0)       # t1 = punter (adreça actual)
        addi    t1, t1, 4       # t1 = punter + 1 (desplaça 4 bytes)
        sw      t1, 0(t0)       # punter = punter + 1
```

---

c) `vec[1] = vec[0] + vec[1];`

RV32I

```
        la      t1, vec
        lw      t3, 0(t1)       # t3 = vec[0]
        lw      t4, 4(t1)       # t4 = vec[1]
        add     t3, t3, t4      # t3 = vec[0] + vec[1]
        sw      t3, 4(t1)       # vec[1] = vec[0] + vec[1]
```

---

d) `vec[2] = *punter + 1;`
Cal llegir el valor de `punter` (una adreça), desreferenciar-lo per obtenir el valor apuntat, sumar-hi 1 i escriure el resultat a `vec[2]`:

RV32I

```
        la      t1, vec
        la      t0, punter
        lw      t2, 0(t0)       # t2 = punter (adreça apuntada)
        lw      t3, 0(t2)       # t3 = *punter
        addi    t3, t3, 1       # t3 = *punter + 1
        sw      t3, 8(t1)       # vec[2] = *punter + 1
```

---

e) `*(punter + 2) = *punter + 1;`
`punter + 2` desplaça 8 bytes (2 enters × 4 bytes), de manera que l'adreça de destí és `punter + 8`:

RV32I

```
        la      t0, punter
        lw      t2, 0(t0)       # t2 = valor de punter (adreça apuntada)
        lw      t3, 0(t2)       # t3 = *punter
        addi    t3, t3, 1       # t3 = *punter + 1
        sw      t3, 8(t2)       # *(punter + 2) = *punter + 1
```

2.8 Vectors

Exemple 25: Solució: bucles sobre un vector

Enunciat: Problema 2.26.
Declaracions: `int v[10]` global; `i` a `t0`, `suma` a `t1`.

---

a) Inicialitza tots els elements de `v` a zero:

RV32I

```
        la      t2, v           # invariant: @v[0]
        li      t3, 10          # invariant: N
        li      t0, 0           # i = 0
bucle_a:
        bge     t0, t3, fibucle_a
        slli    t4, t0, 2       # t4 = i * 4
        add     t4, t4, t2      # t4 = @v[i]
        sw      zero, 0(t4)     # v[i] = 0
        addi    t0, t0, 1       # i++
        j       bucle_a
fibucle_a:
```

---

b) Calcula la suma de tots els elements:

RV32I

```
        la      t2, v           # invariant: @v[0]
        li      t3, 10          # invariant: N
        li      t0, 0           # i = 0
```

```
        li      t1, 0           # suma = 0
bucle_b:
        bge     t0, t3, fibucle_b
        slli    t4, t0, 2       # t4 = i * 4
        add     t4, t4, t2      # t4 = @v[i]
        lw      t5, 0(t4)       # t5 = v[i]
        add     t1, t1, t5      # suma += v[i]
        addi    t0, t0, 1       # i++
        j       bucle_b
fibucle_b:
```

---

c) Copia els elements de `v` al revés a `w`:

RV32I

```
        la      t2, v           # invariant: @v[0]
        la      t3, w           # invariant: @w[0]
        li      t4, 10          # invariant: N
        li      t5, 9           # invariant: N - 1
        li      t0, 0           # i = 0
bucle_c:
        bge     t0, t4, fibucle_c
        slli    t6, t0, 2       # t6 = i * 4
        add     t6, t6, t2      # t6 = @v[i]
        lw      t6, 0(t6)       # t6 = v[i]  (l'adreça ja no cal, es reutilitza t6 per al valor)
        sub     t1, t5, t0      # t1 = 9 - i
        slli    t1, t1, 2       # t1 = (9 - i) * 4
        add     t1, t1, t3      # t1 = @w[9 - i]
        sw      t6, 0(t1)       # w[9 - i] = v[i]
        addi    t0, t0, 1       # i++
        j       bucle_c
fibucle_c:
```

A la declaració cal afegir `w` a continuació de `v`:

RV32I

```
.data
v:      .space  40      # int v[10]
w:      .space  40      # int w[10]
```

Exemple 26: Solució: cerca d'un valor en un vector

Enunciat: Problema 2.27.
Declaracions: `int vec[100]` global; `i` a `t0`, `val` a `t1`; el resultat (l'índex de la primera ocurrència o -1) s'ha de deixar a `t2`.
El bucle recorre el vector `i` compara cada element amb `val`. La novetat respecte als bucles de Solució 25 és la **sortida anticipada**: en trobar la primera coincidència, se surt del bucle immediatament amb l'índex actual, sense recórrer la resta del vector.

RV32I

```
        la      t3, vec         # invariant: @vec[0]
        li      t4, 100         # invariant: N
```

```
        li      t0, 0           # i = 0
bucle:
        bge     t0, t4, notrobat # final del vector sense coincidència
        slli    t5, t0, 2       # t5 = i * 4
        add     t5, t5, t3      # t5 = @vec[i]
        lw      t5, 0(t5)       # t5 = vec[i]
        beq     t5, t1, trobat  # coincidència: surt amb l'i actual
        addi    t0, t0, 1       # i++
        j       bucle
trobat:
        mv      t2, t0          # t2 = índex de la primera ocurrència
        j       fi
notrobat:
        li      t2, -1          # t2 = -1 (val no és a vec)
fi:
```

El `j fi` després de `mv t2, t0` és imprescindible: sense aquest salt, el flux d'execució continuaria seqüencialment cap a `notrobat` i sobreescriuria l'índex trobat amb -1.

💡 Exemple 27: Solució: aritmètica de punters sobre `short`

Enunciat: Problema 2.28.
Declaracions: `short v[8]` global; `p` a `t0`, `i` a `t1`.
a) `p = v + 3;`
`p` apunta a `short`, un tipus de mida 2 bytes. Per la regla d'escalat de l'aritmètica de punters (Secció 2.7.6), `v + 3` desplaça l'adreça base $3 \times 2 = 6$ bytes: el punter s'incrementa en **6 bytes** (no en 3).

RV32I

```
        la      t0, v+6         # p = v + 3  (3 elements × 2 bytes = 6 bytes)
```

Nota: la sintaxi `la label+offset` és admesa a EC, però a RARS cal substituir-la per `la t0, v` seguit d'`addi t0, t0, 6` (EC 2.6).
b) Bucle que dobla cada element usant `p`:

RV32I

```
        la      t0, v           # p = v  (@v[0])
        li      t2, 8           # invariant: N
        li      t1, 0           # i = 0
bucle:
        bge     t1, t2, fibucle
        lh      t3, 0(t0)       # t3 = *p
        add     t3, t3, t3      # t3 = 2 * (*p)
        sh      t3, 0(t0)       # *p = 2 * (*p)
        addi    t0, t0, 2       # p++  (stride = 2 bytes: mida d'un short)
        addi    t1, t1, 1       # i++
        j       bucle
fibucle:
```

Com que els elements són `short`, l'accés a memòria es fa amb `lh`/`sh` i l'increment del punter (`p++`) és de 2 bytes per element, a diferència dels 4 bytes dels vectors d'`int` vistos fins ara.

2.9 Cadenes de caràcters

Exemple 28: Solució: funció `longitud`

Enunciat: Problema 2.30.
La funció ha de recórrer la cadena byte a byte fins a trobar el `'\0'` i retornar el nombre de caràcters llegits sense comptar-lo.

RV32I

```
# longitud: retorna la longitud d'una cadena acabada en '\0'
# a0: adreça base de la cadena
# Retorn: a0 = longitud (sense comptar el '\0')

longitud:
        mv      t0, a0          # t0 = punter al caràcter actual
        li      t1, 0           # comptador = 0
bucle:
        lb      t2, 0(t0)       # t2 = caràcter actual
        beq     t2, zero, fi    # si '\0', acabar
        addi    t0, t0, 1       # avança el punter
        addi    t1, t1, 1       # comptador++
        j       bucle
fi:
        mv      a0, t1          # retorn: a0 = longitud
        ret
```

La funció usa únicament registres temporals (`t0`–`t2`), de manera que no cal preservar cap registre.

Exemple 29: Solució: còpia de cadenes de caràcters

Enunciat: Problema 2.31.
Es recorre `src` byte a byte amb dos punters que avancen alhora, un per a cada cadena. Cada byte es copia **abans** de comprovar si és el `'\0'`; d'aquesta manera, el caràcter de final de cadena també queda copiat a `dst` sense cap instrucció addicional després del bucle:

RV32I

```
        la      t0, src         # punter a src
        la      t1, dst         # punter a dst
bucle:
        lb      t2, 0(t0)       # t2 = caràcter actual de src
        sb      t2, 0(t1)       # el copia a dst (inclòs el '\0')
        beq     t2, zero, fibucle # si era '\0', la còpia és completa
        addi    t0, t0, 1       # avança el punter de src
        addi    t1, t1, 1       # avança el punter de dst
        j       bucle
fibucle:
```

Si la comprovació del `'\0'` es fes abans de l'`sb` (com al bucle de Solució 28), el bucle acabaria sense copiar el terminador i caldria un `sb` addicional després de l'etiqueta de sortida.

Tema 3: Traducció de programes

3.1 Operacions lògiques i desplaçaments

Exemple 30: Solució: operacions lògiques bit a bit

Enunciat: Problema 3.1.

a) `0x7654 | 0x3333`

`Càlcul`

```
  0111 0110 0101 0100   (0x7654)
| 0011 0011 0011 0011   (0x3333)
= 0111 0111 0111 0111
```

Resultat: **`0x7777`**.

b) `0x7654 & 0x3333`

`Càlcul`

```
  0111 0110 0101 0100   (0x7654)
& 0011 0011 0011 0011   (0x3333)
= 0011 0010 0001 0000
```

Resultat: **`0x3210`**.

c) `0x7654 << 2`

Desplaçament lògic a l'esquerra de 2 posicions: els dos bits de més pes es perden i s'afegeixen dos zeros per la dreta.

`Càlcul`

```
  0111 0110 0101 0100   (0x7654)  << 2
= 1101 1001 0101 0000
```

Resultat: **`0xD950`**.

d) `0x7654 >> 3` (desplaçament lògic)

Desplaçament lògic a la dreta de 3 posicions: els tres bits de menys pes es perden i s'afegeixen tres zeros per l'esquerra.

`Càlcul`

```
  0111 0110 0101 0100   (0x7654)  >> 3
= 0000 1110 1100 1010
```

Resultat: **`0x0ECA`**.

Exemple 31: Solució: desplaçaments sobre 64 bits

Enunciat: Problema 3.5.
En RV32I no hi ha registres de 64 bits. El valor es reparteix entre dos registres de 32 bits: `t2` conté els 32 bits de **més pes** (bits 63–32) i `t1` conté els 32 bits de **menys pes** (bits 31–0).

a) Desplaçament lògic a l'esquerra d'1 posició
El bit 31 de `t1` (part baixa) ha de passar a ser el bit 0 de `t2` (part alta). La resta dels bits de cada registre es desplacen normalment.

RV32I

```
srli    t3, t1, 31      # extreu el bit 31 de t1 a t3[0]
slli    t2, t2, 1       # desplaça t2 a l'esquerra 1 posició
or      t2, t2, t3      # insereix el bit de ròssec provinent de t1
slli    t1, t1, 1       # desplaça t1 a l'esquerra 1 posició
```

b) Desplaçament lògic a la dreta d'1 posició
El bit 0 de `t2` (part alta) ha de passar a ser el bit 31 de `t1` (part baixa).

RV32I

```
slli    t3, t2, 31      # extreu el bit 0 de t2 i el posa a la posició 31 de t3
srli    t1, t1, 1       # desplaça t1 a la dreta 1 posició
or      t1, t1, t3      # insereix el bit de ròssec provinent de t2
srli    t2, t2, 1       # desplaça t2 a la dreta 1 posició
```

c) Desplaçament aritmètic a la dreta d'1 posició
Igual que el desplaçament lògic a la dreta, però la posició vacant de `t2` s'omple amb el bit de signe original (bit 31 de `t2`).

RV32I

```
slli    t3, t2, 31      # extreu el bit 0 de t2 i el posa a la posició 31 de t3
srli    t1, t1, 1       # desplaça t1 a la dreta 1 posició (omplint amb 0)
or      t1, t1, t3      # insereix el bit de ròssec provinent de t2
srai    t2, t2, 1       # desplaça t2 a la dreta 1 posició (omplint amb signe)
```

La diferència entre els apartats b) i c) és l'última instrucció: `srli` omple amb `0`, `srai` omple amb el bit de signe.

d) Desplaçament lògic a l'esquerra de n posicions ($n < 32$, n a `t3`)
Els bits superiors de `t1` que «surten» per la part alta han d'entrar a `t2` per la part baixa. Per obtenir-los, cal desplaçar `t1` a la dreta `32 - n` posicions.
En RV32I, les instruccions de desplaçament usen els **5 bits de menys pes** de l'operand de desplaçament (mòdul 32). Això implica que si `n = 0`, `32 - n = 32` $\equiv$ `0 (mod 32)`, de manera que `srl t5, t1, t4` desplaçaria 0 posicions en lloc de 32, donant un resultat incorrecte.
Versió restringida ($1 \leq n \leq 31$)
Si es pot garantir que `n` mai és 0, la solució és directa:

RV32I

```
li      t4, 32
sub     t4, t4, t3      # t4 = 32 - n  (rang: 1..31)
srl     t5, t1, t4      # t5 = t1 >> (32-n): bits que passen de t1 a t2
sll     t2, t2, t3      # desplaça t2 n posicions
or      t2, t2, t5      # insereix els bits provinents de t1
sll     t1, t1, t3      # desplaça t1 n posicions
```

Versió genèrica ($0 \leq n \leq 31$)
Per cobrir el cas `n = 0` sense instruccions de salt, es pot evitar el desplaçament de 32 usant dos desplaçaments consecutius d'1 i `31 - n` posicions (la suma és sempre `32 - n`, i cap dels dos mai arriba a 32):

RV32I

```
li      t4, 31
sub     t4, t4, t3      # t4 = 31 - n  (rang: 0..31)
srli    t5, t1, 1       # desplaça 1 posició (segur per a qualsevol n)
srl     t5, t5, t4      # desplaça 31-n posicions més: total = 32-n
sll     t2, t2, t3      # desplaça t2 n posicions
or      t2, t2, t5      # insereix els bits provinents de t1
sll     t1, t1, t3      # desplaça t1 n posicions
```

Quan `n = 0`: `t4 = 31`, `srli t5, t1, 1` desplaça 1 posició i `srl t5, t5, 31` desplaça 31 posicions més; total = 32 posicions, resultat correcte (`t5 = 0`). Quan `n > 0`: la suma dels dos desplaçaments és `1 + (31 - n) = 32 - n`, com cal.

e) Desplaçament lògic a la dreta de n posicions ($n < 32$, **n a `t3`**)
Simètric a l'apartat d): els bits inferiors de `t2` han d'entrar a `t1` per la part alta.

Versió restringida ($1 \leq n \leq 31$)

RV32I

```
li      t4, 32
sub     t4, t4, t3      # t4 = 32 - n  (rang: 1..31)
sll     t5, t2, t4      # t5 = t2 << (32-n): bits que passen de t2 a t1
srl     t1, t1, t3      # desplaça t1 n posicions
or      t1, t1, t5      # insereix els bits provinents de t2
srl     t2, t2, t3      # desplaça t2 n posicions
```

Versió genèrica ($0 \leq n \leq 31$)

RV32I

```
li      t4, 31
sub     t4, t4, t3      # t4 = 31 - n  (rang: 0..31)
slli    t5, t2, 1       # desplaça 1 posició (segur per a qualsevol n)
sll     t5, t5, t4      # desplaça 31-n posicions més: total = 32-n
srl     t1, t1, t3      # desplaça t1 n posicions
or      t1, t1, t5      # insereix els bits provinents de t2
srl     t2, t2, t3      # desplaça t2 n posicions
```

Exemple 32: Solució: rotació de bits

Enunciat: Problema 3.9.
Notació: el contingut inicial de `t4` és `b31 b30 ... b1 b0`.
a) Rotació a l'esquerra d'1 posició.
Cal combinar dos desplaçaments amb un `or`: el desplaçament a l'esquerra d'1 posició mou tots els bits cap a l'esquerra (perd `b31` i deixa un `0` al bit 0), i el desplaçament lògic a la dreta de 31 posicions aïlla `b31` al bit 0:

RV32I

```
        srli    t1, t4, 31      # t1 = 0 0 ... 0 b31
        slli    t4, t4, 1       # t4 = b30 b29 ... b0 0
        or      t4, t4, t1      # t4 = b30 b29 ... b0 b31
```

Contingut final: `b30 b29 ... b1 b0 b31`.
b) Seguiment del codi.
El bucle s'executa **16 vegades** (`t1` va de 16 a 0). A cada iteració:

- `srli t4, t4, 1` desplaça `t4` una posició a la dreta (lògic: entra un `0` per l'esquerra).
- `add t2, t2, t2` suma `t2` amb si mateix, és a dir, el duplica: equival a desplaçar-lo una posició a l'esquerra (Secció 3.1.4).

Després de les 16 iteracions:

- `t4` = valor inicial desplaçat 16 posicions a la dreta: `0 0 ... 0 b31 b30 ... b16` (16 zeros seguits dels 16 bits de més pes).
- `t2` = valor inicial desplaçat 16 posicions a l'esquerra: `b15 b14 ... b0 0 0 ... 0` (els 16 bits de menys pes seguits de 16 zeros).

L'`or` final combina les dues meitats, que no se superposen:

`t4 final`

```
b15 b14 ... b1 b0 b31 b30 ... b17 b16
```

El codi fa una **rotació de 16 posicions**: les dues meitats de 16 bits del registre s'intercanvien. Amb una rotació de 16 posicions sobre 32 bits, girar a l'esquerra o a la dreta dona el mateix resultat.

3.2 Sentències condicionals i bucles

Exemple 33: Solució: sentència `switch`

Enunciat: Problema 3.12.

`C`

```
switch (op) {
    case 0:  res = a + b;  break;
    case 1:  res = a - b;  break;
    case 2:  res = a & b;  break;
    case 3:  res = a | b;  break;
    default: res = 0;
}
```

`op` a `t0`, `a` a `t1`, `b` a `t2`, `res` a `t3`.
Implementació 1: cadena d'`if-then-else`

`RV32I`

```
        li      t4, 0
        beq     t0, t4, cas0
        li      t4, 1
```

```
        beq     t0, t4, cas1
        li      t4, 2
        beq     t0, t4, cas2
        li      t4, 3
        beq     t0, t4, cas3
        mv      t3, zero            # default: res = 0
        j       fiswitch
cas0:   add     t3, t1, t2          # res = a + b
        j       fiswitch
cas1:   sub     t3, t1, t2          # res = a - b
        j       fiswitch
cas2:   and     t3, t1, t2          # res = a & b
        j       fiswitch
cas3:   or      t3, t1, t2          # res = a | b
fiswitch:
```

Implementació 2: taula de salts (*jump table*)

Les constants `0`–`3` formen un rang dens, cosa que fa la taula de salts eficient.

RV32I

```
.data
switch_vec:
        .word   cas0
        .word   cas1
        .word   cas2
        .word   cas3

.text
        li      t4, 4
        bgeu    t0, t4, defecte     # si op >= 4, salta al default
        la      t4, switch_vec
        slli    t5, t0, 2           # op * 4
        add     t4, t4, t5          # @switch_vec[op]
        lw      t5, 0(t4)           # llegeix l'adreça del cas
        jr      t5                  # salta al cas corresponent
cas0:   add     t3, t1, t2          # res = a + b
        j       fiswitch
cas1:   sub     t3, t1, t2          # res = a - b
        j       fiswitch
cas2:   and     t3, t1, t2          # res = a & b
        j       fiswitch
cas3:   or      t3, t1, t2          # res = a | b
        j       fiswitch
defecte:
        mv      t3, zero            # res = 0
fiswitch:
```

Quan és preferible cada enfocament?

La cadena d'`if-then-else` és preferible quan el nombre de casos és petit (fins a 3–4) o quan les constants no formen un rang dens (per exemple, `case 0`, `case 100`, `case 1000`). La taula de salts és preferible quan les constants formen un rang dens i el nombre de casos és gran: l'accés és $\mathcal{O}(1)$ independentment del nombre de casos, mentre que la cadena té cost $\mathcal{O}(n)$ en el pitjor cas.

Exemple 34: Solució: bucle `for` simple

Enunciat: Problema 3.13.

a) Traducció a assemblador RISC-V

C

```c
void main() {
    int a = 23;  // guardat a t0
    int b = 12;  // guardat a t1
    int i;       // guardat a t2
    for (i = 0; i < 10; i++)
        a += b;
}
```

RV32I

```
        li      t0, 23          # a = 23
        li      t1, 12          # b = 12
        li      t2, 0           # i = 0
bucle:
        li      t3, 10
        bge     t2, t3, fibucle # salta si i >= 10
        add     t0, t0, t1      # a += b
        addi    t2, t2, 1       # i++
        j       bucle
fibucle:
```

b) Nombre total d'instruccions executades

Inicialització (3 instruccions): `li t0`, `li t1`, `li t2`.

Per a cada iteració $i = 0, 1, \ldots, 9$ (10 iteracions, el salt **no** es pren): `li t3` + `bge` + `add` + `addi` + `j` = **5 instruccions**.

Darrera comprovació ($i = 10$, el salt **es pren**): `li t3` + `bge` = **2 instruccions**.

Total: $3 + 10 \times 5 + 2 = \mathbf{55}$ instruccions.

c) Temps d'execució si hi hagués 10 instruccions, CPI = 2, freqüència = 2 GHz

Segons l'Equació 6.3, amb el període de rellotge expressat com a $1/f_{clock}$:

$$t_{exe} = \frac{n_{ins} \times \text{CPI}}{f_{clock}} = \frac{10 \times 2}{2 \times 10^9} = 10^{-8}\ \text{s} = 10\ \text{ns}$$

d) CPI si el programa, amb les 10 instruccions de l'apartat c), triga 5×10^{-9} s a 2 GHz

$$\text{CPI} = \frac{t_{exe} \times f_{clock}}{n_{ins}} = \frac{5 \times 10^{-9} \times 2 \times 10^9}{10} = \frac{10}{10} = 1$$

És a dir, el programa fa 10 cicles per a 10 instruccions.

Exemple 35: Solució: anàlisi de cost de bucles de suma de vector

Enunciat: Problema 3.17.

Model de costos: memòria = 4 cicles; salt pres = 2 cicles; salt no pres = 1 cicle; resta = 1 cicle. Les pseudoinstruccions es compten per les instruccions en què s'expandeixen: `la` en són dues, `auipc` + `addi` (RISC-V 2.17), i `li t3, N`, una, perquè se suposa que N cap en 12 bits

(RISC-V 2.16).
Per a cada versió s'analitzen les instruccions executades per iteració i el temps total per a N iteracions. Les instruccions fora del bucle (inicialització) es compten una sola vegada.

---

a)
Instruccions fora del bucle: `li t0` + `li t1` + `li t2` = 3 cicles.
Per iteració (N vegades, salt `j` pres): `li t3` (1) + `bge` no pres (1) + `la t4` (2) + `add t4` (1) + `lw t4` (4) + `add t0` (1) + `addi t2` (1) + `addi t1` (1) + `j` pres (2) = **14 cicles**.
Última comprovació ($i = N$): `li t3` (1) + `bge` pres (2) = 3 cicles.
Total: $3 + 14N + 3 = 14N + 6$ cicles.

---

b)
Instruccions fora del bucle: `li t0` + `li t1` = 2 cicles.
Per iteració: `li t3` (1) + `bge` no pres (1) + `la t3` (2) + `slli t2` (1) + `add t2` (1) + `lw t2` (4) + `add t0` (1) + `addi t1` (1) + `j` pres (2) = **14 cicles**.
Última comprovació: `li t3` (1) + `bge` pres (2) = 3 cicles.
Total: $2 + 14N + 3 = 14N + 5$ cicles.

---

c)
Instruccions fora del bucle: `li t0` + `li t1` = 2 cicles.
Per iteració (condició al final; última iteració: salt no pres): `la t3` (2) + `slli t2` (1) + `add t2` (1) + `lw t2` (4) + `add t0` (1) + `addi t1` (1) + `li t3` (1) + `blt` pres (2) = **13 cicles** per a les $N - 1$ primeres iteracions.
Última iteració: `blt` no pres (1) $\rightarrow$ cost $= 13 - 2 + 1 = 12$ cicles.
Total: $2 + 13(N - 1) + 12 = 13N + 1$ cicles.

---

d)
Instruccions fora del bucle: `li t0` + `li t1` = 2 cicles.
Per iteració (condició al final, comptador decreixent; salt `bge` pres $N - 1$ vegades): `la t3` (2) + `slli t2` (1) + `add t2` (1) + `lw t2` (4) + `add t0` (1) + `addi t1` (1) + `bge` pres (2) = **12 cicles** per a les $N - 1$ primeres iteracions.
Última iteració: `bge` no pres (1) $\rightarrow$ cost $= 12 - 2 + 1 = 11$ cicles.
Total: $2 + 12(N - 1) + 11 = 12N + 1$ cicles.

---

e)
Instruccions fora del bucle: `li t0` (1) + `li t1` (1) + `la t2` (2) = 4 cicles.
Per iteració (condició al final; salt `bne` pres $N - 1$ vegades): `lw t3` (4) + `add t0` (1) + `addi t2` (1) + `addi t1` (1) + `bne` pres (2) = **9 cicles** per a les $N - 1$ primeres iteracions.
Última iteració: `bne` no pres (1) $\rightarrow$ cost $= 9 - 2 + 1 = 8$ cicles.
Total: $4 + 9(N - 1) + 8 = 9N + 3$ cicles.

---

f) Nova versió que millora l'apartat e)
L'apartat e) ja avalua la condició al final i fa accés seqüencial amb un punter, però encara manté el comptador `t1` per detectar el final del bucle. La millora que fa el Codi 3.1 és eliminar la variable d'inducció, comparant directament el punter amb una sentinella que marca el final del vector:

Codi 3.1: Versió de l'apartat f), sense la variable d'inducció.
RV32I

```
        li      t0, 0           # suma = 0
        la      t2, v           # p = @v[0]
        li      t1, N
        slli    t1, t1, 2       # N*4
        add     t1, t2, t1      # sentinella = @v[N]
bucle:
        lw      t3, 0(t2)       # *p
        add     t0, t0, t3      # suma += *p
        addi    t2, t2, 4       # p++
        bltu    t2, t1, bucle   # salta si p < sentinella
```

Instruccions fora del bucle: `li t0` (1) + `la t2` (2) + `li t1` (1) + `slli` (1) + `add` (1) = 6 cicles.
Per iteració: `lw` (4) + `add` (1) + `addi` (1) + `bltu` pres (2) = **8 cicles** per a les $N-1$ primeres.
Última iteració: `bltu` no pres (1) $\rightarrow$ 7 cicles.
Total: $6 + 8(N-1) + 7 = 8N + 5$ cicles.
Respecte a l'apartat e) ($9N+3$), s'estalvia 1 cicle per iteració a cost de 2 instruccions d'inicialització addicionals (+2 cicles). Totes dues versions costen el mateix per a $N = 2$; a partir de $N = 3$, la versió f) és millor.

3.3 Subrutines: introducció

Exemple 36: Solució: funció mínim de dos enters

Enunciat: Problema 3.21.
Codi en C:
C

```c
int minim(int a, int b) {
    if (a <= b)
        return a;
    else
        return b;
}
```

Traducció a assemblador RISC-V:
La funció és **uninivell** (no fa cap crida a una altra subrutina), per tant, no cal desar `ra` ni cap registre segur.
Els paràmetres `a` i `b` arriben per `a0` i `a1`. El resultat es retorna per `a0`.
RV32I

```
minim:
        blt     a0, a1, retorn  # salta si a < b  (a és el mínim)
        mv      a0, a1          # a >= b: el mínim és b → a0 = b
retorn:
        ret                     # retorna a0 (a o b segons el cas)
```

Anàlisi: si `a < b` el salt es pren i es retorna `a0` sense modificar (el mínim és `a`). Si `a >= b` el salt no es pren, `a0` s'actualitza amb `b` i es retorna (el mínim és `b`). El cas `a == b` queda correctament cobert: `blt` no salta amb igualtat, de manera que s'executa `mv a0, a1`, però com que `a == b` el resultat és el mateix.

Exemple 37: Solució: funcions `main` i `examen`

Enunciat: Problema 3.23.

C

```c
void examen(char vi[], char vo[]) {
    int i = 0;
    while (vi[i] != '\0') {
        if (vi[i] != ' ')
            vo[i] = vi[i] - 32;
        else
            vo[i] = vi[i];
        i++;
    }
    vo[i] = '\0';
}
void main() {
    examen(vect1, vect2);
}
```

`vect1` i `vect2` són variables globals.

a) Traducció de `main`

`main` és **multinivell** (crida `examen`), per tant, ha de desar `ra`.

RV32I

```
main:
        addi    sp, sp, -4
        sw      ra, 0(sp)
        la      a0, vect1       # a0 = @vect1
        la      a1, vect2       # a1 = @vect2
        jal     ra, examen
        lw      ra, 0(sp)
        addi    sp, sp, 4
        ret
```

b) Traducció d'`examen`

`examen` és **uninivell**: no fa cap crida i no usa cap registre segur. La variable local `i` s'assigna a `t0` (temporal).

RV32I

```
examen:
        # a0 = @vi,  a1 = @vo
        li      t0, 0               # i = 0
bucle:
        add     t1, a0, t0          # @vi[i]
        lb      t2, 0(t1)           # t2 = vi[i]
        beqz    t2, fivi            # salta si vi[i] == '\0'
```

```
        li      t3, ' '
        beq     t2, t3, espai       # salta si vi[i] == ' '

        addi    t2, t2, -32         # vo[i] = vi[i] - 32  (passa a majúscula)
        j       escriu
espai:                              # vo[i] = vi[i]  (copia l'espai)
escriu:
        add     t3, a1, t0          # @vo[i]
        sb      t2, 0(t3)           # vo[i] = t2
        addi    t0, t0, 1           # i++
        j       bucle
fivi:
        add     t3, a1, t0          # @vo[i]
        sb      zero, 0(t3)         # vo[i] = '\0'
        ret
```

Anàlisi: la funció recorre `vi` byte a byte fins al `'\0'`. Per a cada caràcter que no és un espai, en resta 32 (que és la diferència entre els codis ASCII de les minúscules i les majúscules corresponents, `'a' - 'A' = 32`) i el desa a `vo`, convertint-lo a majúscula. Els espais es copien sense modificar. El `'\0'` final s'escriu explícitament a `vo` després del bucle.

Exemple 38: Solució: seguiment de `jalr`

Enunciat: Problema 3.30.
Primer cal fixar les adreces de les instruccions. La pseudoinstrucció `la` s'expandeix en dues instruccions reals, `auipc` + `addi`. Suposant que el fragment comença a l'adreça A:

Adreça	Instrucció	Observacions
A	`auipc ra, ...`	expansió de `la ra, etiq`
$A+4$	`addi ra, ra, ...`	expansió de `la ra, etiq`
$A+8$	`addi ra, ra, -4`	
$A+12$	`jalr ra, 0(ra)`	`etiq`

Per tant, `etiq` $= A + 12$.
Per fer el seguiment cal recordar la semàntica de `jalr rd, offset(rs1)`: l'adreça de destinació es calcula amb el valor **actual** de `rs1`, i **després** s'escriu `rd` $\leftarrow$ PC $+ 4$. Quan `rd` i `rs1` són el mateix registre (com aquí, `ra`), la lectura es fa abans de l'escriptura.
Seguiment pas a pas:

1. `la ra, etiq` $\rightarrow$ `ra` $= A + 12$.
2. `addi ra, ra, -4` $\rightarrow$ `ra` $= A + 8$.
3. **1a execució de `jalr`** (PC $= A + 12$): destinació $=$ `ra` $+ 0 = A + 8$; `ra` $\leftarrow$ PC $+ 4 = A + 16$. Salta a $A + 8$, la instrucció `addi ra, ra, -4`.
4. `addi ra, ra, -4` $\rightarrow$ `ra` $= A + 16 - 4 = A + 12$. El flux continua seqüencialment cap a $A + 12$.
5. **2a execució de `jalr`** (PC $= A + 12$): destinació $= A + 12$ (salta a si mateixa!); `ra` $\leftarrow A + 16$.
6. **3a execució de `jalr`** (PC $= A + 12$): destinació $=$ `ra` $= A + 16$; `ra` $\leftarrow A + 16$. Salta a $A + 16$, fora del fragment: l'execució continua amb el codi que segueix.

Resposta: **c)** la instrucció `jalr` s'executa **3 vegades**.
El resultat no depèn de quantes instruccions ocupi l'expansió de `la`: el primer `jalr` salta a `etiq` -4, que és sempre la instrucció `addi ra, ra, -4` immediatament anterior.

3.4 Subrutines: context i bloc d'activació

Exemple 39: Solució: funció `A` amb bloc d'activació

Enunciat: Problema 3.33.

C

```c
char  x;
int   z;
char  w[20];

char A(char i, int k, char *v) {
    char r[10];
    char c;
    c    = v[3];
    r[k] = i;
    return c;
}
```

a) Traducció de `x = A(x, z, w)`
Aquesta sentència forma part de codi que crida `A`. Els paràmetres s'han de preparar abans de la crida: `i` (char) per `a0`, `k` (int) per `a1`, `v` (char *) per `a2`.

RV32I

```
        la      t0, x
        lb      a0, 0(t0)       # a0 = x (char, extensió de signe)
        la      t1, z
        lw      a1, 0(t1)       # a1 = z (int)
        la      a2, w           # a2 = @w (punter)
        jal     ra, A
        la      t0, x
        sb      a0, 0(t0)       # x = resultat de A (char)
```

b) Bloc d'activació d'`A` (Figura 3.1)
Variables locals: `r[10]` (10 chars × 1 byte = 10 bytes) + `c` (escalar, s'assigna a registre). Per tant, el BA conté únicament `r`.
Mida del BA: 10 bytes, arrodonit a múltiple de 4 → **12 bytes**.
`A` és **uninivell** (no fa cap crida a cap altra subrutina): no cal desar `ra`.

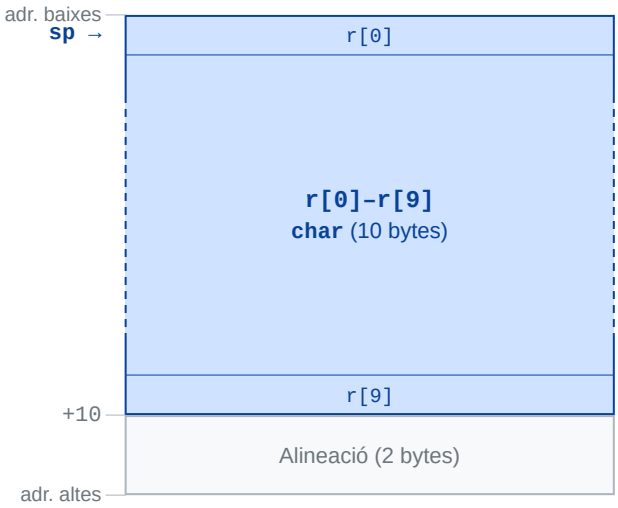


Figura 3.1: Bloc d'activació de la funció `A`: el vector `r` al desplaçament `+0` i l'alineació als bytes `+10`–`+11`.

c) Traducció d'`A`
Els paràmetres arriben per `a0` (`i`, char), `a1` (`k`, int), `a2` (`v`, char *). El resultat es retorna per `a0`. La variable `c` s'assigna a `t0` (escalar, uninivell). El vector `r` va al BA.

RV32I

```
A:
        addi    sp, sp, -12     # reserva espai per a r[10] + alineació

        # c = v[3]
        lb      t0, 3(a2)       # t0 = v[3]  (c)

        # r[k] = i
        add     t1, sp, a1      # @r[k] = sp + k  (stride = 1 byte)
        sb      a0, 0(t1)       # r[k] = i

        # return c
        mv      a0, t0          # a0 = c

        addi    sp, sp, 12      # allibera el BA
        ret
```

3.5 Compilació, assemblatge, enllaçament i càrrega

Exemple 40: Solució: expansió de `la` i rang del desplaçament

Enunciat: Problema 3.36.
Quines instruccions? `la rd, symbol` s'expandeix en `auipc` + `addi` (RISC-V 2.17):

RV32I

```
        auipc   rd, hi_20(offset)     # rd ← PC + (hi_20(offset) << 12)
        addi    rd, rd, lo_12(offset) # rd ← rd + lo_12(offset)
```

on `offset = @symbol - PC` és la distància entre la instrucció `auipc` i el símbol.
Per què en calen dues? Un desplaçament de 32 bits no cap en cap immediat d'RV32I: el més gran és el de 20 bits del format U. Calen dues instruccions per construir els 32 bits: `auipc` aporta els 20 bits superiors (sumats al PC) i `addi` els 12 bits inferiors. Com en l'expansió de `li` amb `lui` + `addi`, si el bit 11 de `lo_12` val 1 cal incrementar `hi_20` en 1 per compensar l'extensió de signe d'`addi` (el mecanisme detallat és a Problema 2.5).
Limitació del desplaçament: l'expansió codifica un desplaçament de 32 bits amb signe, $[-2^{31}, 2^{31}-1]$, de manera que el símbol ha de ser dins d'un rang de **±2 GiB** respecte al PC. En RV32I, on l'espai d'adreces és de 32 bits (4 GiB) i l'aritmètica d'adreces és mòdul 2^{32}, aquest rang cobreix a la pràctica tot l'espai d'adreces; en arquitectures de 64 bits, en canvi, és una limitació real que obliga a usar altres mecanismes per a símbols més llunyans.

Exemple 41: Solució: compilació separada i reubicació

Enunciat: Problema 3.37.

a.c

```c
int x = 42;
int f(int n) { return n + x; }
```

b.c

```c
extern int x;
int f(int n);
int main() { return f(x); }
```

a) Fases del flux de generació
Les tres fases que intervenen en construir l'executable a partir d'`a.c` i `b.c` són:

- **Compilació**: el compilador tradueix cada fitxer `.c` a assemblador (`.s`). En aquesta fase s'apliquen les regles de l'ABI i es generen les instruccions de l'ISA, però les adreces no es resolen: s'usen etiquetes.
- **Assemblatge**: l'assemblador tradueix cada fitxer `.s` a codi màquina (`.o`). Construeix la taula de símbols locals, expandeix pseudoinstruccions i genera les llistes de reubicació i de referències externes per als símbols no resolts.
- **Enllaçament**: l'enllaçador (*linker*) combina `a.o` i `b.o`, resol els símbols globals i externs (`x`, `f`), corregeix les adreces provisionals i genera l'executable.

La **càrrega** en memòria la fa el sistema operatiu i no forma part del procés de construcció de l'executable en si.

b) Qui resol les referències externes a `x` i `f` de `b.o`?
L'**enllaçador**, en la fase d'enllaçament. Durant l'assemblatge de `b.s`, `x` i `f` no estan definits en aquest mòdul, de manera que l'assemblador deixa adreces provisionals i les anota a la llista de referències externes. Quan l'enllaçador combina `a.o` i `b.o`, cerca `x` i `f` a la taula de símbols globals d'`a.o` (on estan declarats amb `.globl`) i substitueix les adreces provisionals per les adreces definitives.
c) Efecte de declarar `static int x = 42` a `a.c`
Amb `static`, `x` passa a tenir **vinculació interna** (*internal linkage*): deixa de ser visible fora del mòdul `a.c`. L'assemblador **no** l'inclou a la taula de símbols globals del fitxer objecte `a.o`. En conseqüència, quan l'enllaçador intenta resoldre la referència externa a `x` de `b.o`, no la troba i finalitza amb un error (amb el muntador de GNU, `undefined reference to 'x'`).
d) Per què `la t0, x` es tradueix amb `auipc` i no amb `lui`?
L'expansió en `auipc` + `addi` i el rang de ± 2 GiB són els de Solució 40.
La raó d'usar `auipc` (relatiu al PC) en lloc de `lui` (adreça absoluta) és que en un fitxer objecte reubicable les adreces absolutes no es coneixen fins a l'enllaçament. Amb adreces relatives al PC, el desplaçament entre la instrucció i el símbol és constant independentment d'on es carregui el programa en memòria, cosa que simplifica la reubicació.

Tema 4: Aritmètica entera i matrius

4.1 Suma i resta

Exemple 42: Solució: conversió a decimal en tres representacions

Enunciat: Problema 4.1.
Codificació en binari dels dos nombres de 16 bits:

`Codificació`

```
0x0D34 = 0000 1101 0011 0100
0xBA1D = 1011 1010 0001 1101
```

El bit de més pes (bit 15) és `0` per a `0x0D34` i `1` per a `0xBA1D`: les tres interpretacions coincideixen per al primer nombre i divergeixen per al segon.
`0x0D34`: com que el bit 15 és `0`, el valor és el mateix en les tres representacions:

$$0 \cdot 16^3 + 13 \cdot 16^2 + 3 \cdot 16 + 4 = 3380$$

`0xBA1D`:

a) Natural: $11 \cdot 16^3 + 10 \cdot 16^2 + 1 \cdot 16 + 13 = 47\,645$.

b) Enter en Ca2 (Secció 1.5.1): el bit 15 val `1`, per tant, el nombre és negatiu. El valor absolut s'obté fent el complement a 2: `0xBA1D` → invertir bits → `0x45E2` → sumar 1 → `0x45E3` = 17 891. Resultat: −17 891. Comprovació: $47\,645 - 2^{16} = -17\,891$ ✓

c) Enter en signe-magnitud (Secció 1.5.3): bit de signe `1` (negatiu) i magnitud als 15 bits restants: `0x3A1D` = 14 877. Resultat: −14 877.

Resum:

	a) Natural	b) Ca2	c) Signe-magnitud
`0x0D34`	3380	+3380	+3380
`0xBA1D`	47 645	−17 891	−14 877

Exemple 43: Solució: desbordament en enters Ca2 de 8 bits

Enunciat: Problema 4.5.
El rang dels enters en Ca2 de 8 bits és $[-128, 127]$. Es produeix desbordament quan els dos operands tenen el mateix signe i el resultat en té de diferent.
a) $(-25) + (-34)$
Codificació: -25 = `1110 0111`, -34 = `1101 1110`.

`Càlcul`

```
  1110 0111   (-25)
+ 1101 1110   (-34)
-----------
1 1100 0101   (descartant el ròssec de sortida: 0xC5)
```

El resultat truncat a 8 bits és `1100 0101` $= -59_{10}$ en Ca2. Els dos operands són negatius i el resultat també és negatiu: **no hi ha desbordament**. El resultat és correcte: $-25+(-34) = -59$.

b) $(-25) - (-34) = (-25) + 34$

Codificació: $-25 =$ `1110 0111`. El subtrahend, $-34 =$ `1101 1110`, es nega complementant-lo i sumant-hi 1 (Secció 1.6.2): `0010 0001` $+ 1 =$ `0010 0010` $(+34)$.

`Càlcul`

```
  1110 0111   (-25)
+ 0010 0010   (+34)
-----------
1 0000 1001   (descartant el ròssec de sortida: 0x09)
```

Resultat: `0000 1001` $= +9_{10}$. Els operands tenen signes oposats: **no hi ha desbordament**. Correcte: $-25 + 34 = +9$.

c) $(+127) - (-20) = (+127) + 20$

Codificació: $+127 =$ `0111 1111`. El subtrahend, $-20 =$ `1110 1100`, es nega: `0001 0011` $+ 1 =$ `0001 0100` $(+20)$.

`Càlcul`

```
  0111 1111   (+127)
+ 0001 0100   (+20)
-----------
  1001 0011   (0x93)
```

Resultat: `1001 0011` $= -109_{10}$ en Ca2. Tots dos operands eren positius, però el resultat és negatiu: **hi ha desbordament**. El resultat correcte seria $+147$, que no és representable amb 8 bits en Ca2.

d) $(-128) - (-20) = (-128) + 20$

Codificació: $-128 =$ `1000 0000`. El subtrahend negat és, com a c), $+20 =$ `0001 0100`.

`Càlcul`

```
  1000 0000   (-128)
+ 0001 0100   (+20)
-----------
  1001 0100   (0x94)
```

Resultat: `1001 0100` $= -108_{10}$ en Ca2. Els operands tenien signes oposats: **no hi ha desbordament**. Correcte: $-128 + 20 = -108$.

💡 Exemple 44: Solució: detecció del ròssec sense salts

Enunciat: Problema 4.6.

En sumar dos naturals de 32 bits, el resultat es trunca mòdul 2^{32}. Si hi ha ròssec (*carry*, $c = 1$), el valor retornat és $S = A + B - 2^{32}$ i, com que $B < 2^{32}$, es compleix $S < A$ (i, simètricament, $S < B$). Si no n'hi ha, $S = A + B \geq A$. Per tant:

$$c = 1 \iff S <_u A$$

que és exactament la indicació de l'enunciat. La comparació sense signe es fa amb `sltu` (Secció 3.3.1), que deixa el resultat booleà (0 o 1) directament en un registre, sense cap salt:

RV32I

```
add     t0, t1, t2      # t0 = t1 + t2  (truncat a 32 bits)
sltu    t3, t0, t1      # t3 = 1 si t0 < t1 (sense signe): hi ha ròssec
```

Dues instruccions en total. Comparar el resultat amb qualsevol dels dos sumands és equivalent; s'ha triat `t1`.

Exemple 45: Solució: detecció de desbordament d'enters sense salts

Enunciat: Problema 4.7.
Les variables `t1` i `t2` contenen els sumands; el resultat ha d'anar a `t0` i l'indicador de desbordament a `t3` (1 si hi ha desbordament, 0 altrament), sense cap instrucció de salt.
La condició de desbordament en la suma de dos enters és: els operands **tenen el mateix signe** i el resultat **té signe diferent** (Equació 4.2):

$$v = \overline{(A_{31} \oplus B_{31})} \wedge (A_{31} \oplus S_{31})$$

RV32I

```
add     t0, t1, t2      # t0 = t1 + t2  (resultat truncat a 32 bits)
xor     t3, t1, t2      # t1 XOR t2: bit 31 = 0 si mateix signe
not     t3, t3          # NOT(t1 XOR t2): bit 31 = 1 si mateix signe
xor     t4, t1, t0      # t1 XOR resultat: bit 31 = 1 si signes diferents
and     t3, t3, t4      # condició de desbordament als 32 bits
srli    t3, t3, 31      # desplaça el bit 31 a la posició 0
                        # t3 = 1 si desbordament, 0 altrament
```

Anàlisi pas a pas:

- `xor t3, t1, t2`: el bit 31 és `0` si `t1` i `t2` tenen el **mateix signe**, `1` si en tenen de diferent.
- `not t3, t3`: el bit 31 passa a ser `1` si els operands tenien el **mateix signe**.
- `xor t4, t1, t0`: el bit 31 és `1` si el **signe del resultat és diferent** del de `t1`.
- `and t3, t3, t4`: el bit 31 és `1` si i **només si** es compleixen les dues condicions simultàniament: mateix signe als operands i signe diferent al resultat.
- `srli t3, t3, 31`: posa el bit de condició a la posició 0.

El programa no conté cap instrucció de salt: totes les operacions són aritmètiques o lògiques de cost $\mathcal{O}(1)$.

4.2 Multiplicació

Exemple 46: Solució: seguiment del circuit multiplicador seqüencial

Enunciat: Problema 4.10.
El circuit treballa amb naturals. Per a operands de 6 bits, els registres P i MD tenen 12 bits i MR en té 6. A cada iteració:

1. Si `MR[0] == 1`: `P = P + MD`.
2. `MD = MD << 1`.
3. `MR = MR >> 1`.

Convenció de la taula: cada fila mostra l'estat **final** de la iteració; les columnes MR[0] i Acció indiquen el que s'ha **executat en aquella iteració** (decidit amb el valor de `MR[0]` **abans** del desplaçament); la fila inicial porta «—».
a) $x =$ **`101000`** (40_{10}), $y =$ **`010011`** (19_{10}). **Resultat esperat:** 760_{10}.

Iter.	P (12 bits)	MD (12 bits)	MR (6 bits)	MR[0]	Acció
inicial	`000000 000000`	`000000 101000`	`010011`	—	—
1	`000000 101000`	`000001 010000`	`001001`	1	suma
2	`000001 111000`	`000010 100000`	`000100`	1	suma
3	`000001 111000`	`000101 000000`	`000010`	0	no suma
4	`000001 111000`	`001010 000000`	`000001`	0	no suma
5	`001011 111000`	`010100 000000`	`000000`	1	suma
6	`001011 111000`	`101000 000000`	`000000`	0	no suma

Resultat: $P =$ `001011 111000`$_2 = 760_{10}$ ✓
b) $x =$ **`110110`** (54_{10}), $y =$ **`000100`** (4_{10}). **Resultat esperat:** 216_{10}.

Iter.	P (12 bits)	MD (12 bits)	MR (6 bits)	MR[0]	Acció
inicial	`000000 000000`	`000000 110110`	`000100`	—	—
1	`000000 000000`	`000001 101100`	`000010`	0	no suma
2	`000000 000000`	`000011 011000`	`000001`	0	no suma
3	`000011 011000`	`000110 110000`	`000000`	1	suma
4	`000011 011000`	`001101 100000`	`000000`	0	no suma
5	`000011 011000`	`011011 000000`	`000000`	0	no suma
6	`000011 011000`	`110110 000000`	`000000`	0	no suma

Resultat: $P =$ `000011 011000`$_2 = 216_{10}$ ✓

Exemple 47: Solució: coherència d'operands de `mul` i `div`

Enunciat: Problema 4.13.
a) **`mul t3, t1, t2` amb resultat baix `0xFFFFFFFA` i `t1 = 0x00000002`**
`mul` calcula els 32 bits de menys pes del producte, és a dir, el producte **mòdul** 2^{32} (RISC-V 4.2). Cal trobar un `t2` tal que:

$$2 \cdot \texttt{t2} \equiv \texttt{0xFFFFFFFA} \pmod{2^{32}}$$

Com que el factor 2 i el mòdul (2^{32}) no són coprimers, la solució **no és única**: n'hi ha exactament dues, separades per 2^{31}:

- `t2 = 0xFFFFFFFD`: interpretant els valors com a enters, $2 \times (-3) = -6 =$ `0xFFFFFFFA` ✓
- `t2 = 0x7FFFFFFD`: interpretant-los com a naturals, $2 \times 2\,147\,483\,645 = 4\,294\,967\,290 =$ `0xFFFFFFFA` ✓

Totes dues són coherents amb l'enunciat: els 32 bits de menys pes del producte són idèntics tant si els operands s'interpreten com a enters com si s'interpreten com a naturals — per això hi ha una única instrucció `mul`, mentre que per a la part alta cal distingir la interpretació amb `mulh`/`mulhu`/`mulhsu` (Secció 4.3.4 i Aprofundiment 4.7).

b) `div t3, t1, t2` amb `t1 = 0x00000005` i `t2 = 0xFFFFFFFE`

`t2` = `0xFFFFFFFE` $= -2$. La divisió entera trunca cap a zero (Essencial 4.3):

$$q = \text{trunc}\left(\frac{5}{-2}\right) = \text{trunc}(-2{,}5) = -2 \qquad \Rightarrow \qquad \texttt{t3} = \texttt{0xFFFFFFFE}$$

El residu té el signe del dividend i satisfà la identitat $x = y \cdot q + r$:

$$r = 5 - (-2) \times (-2) = 5 - 4 = +1$$

c) Casos de desbordament en la divisió entera

Un únic cas: $-2^{31} \div (-1)$, el resultat matemàtic del qual ($+2^{31}$) no és representable en Ca2 de 32 bits. RISC-V no genera cap excepció: `div` retorna -2^{31} i `rem` retorna 0 (Secció 4.4.3.1). La divisió per zero **no** és un cas de desbordament, sinó un cas especial amb resultat definit (`div` → -1, `rem` → dividend).

4.3 Matrius

Nota: les solucions deixen les expressions aritmètiques als operands (`NC*4`, `mat + 3*NC*4`, `mat + (3*NC + 11)*4`) per fer visible l'invariant de cada càlcul d'adreces, tal com s'admet a la teoria i als exàmens (EC 2.3).

Exemple 48: Solució: càlcul d'adreces en una matriu

Enunciat: Problema 4.14.

C

```c
int mat[NF][NC];
int f() {
    int i, j;  // i a t0, j a t1
    ...
}
```

L'adreça de `mat[i][j]` és `@mat` $+ (i \cdot$ `NC` $+ j) \cdot 4$.

a) `mat[3][11]` — fila i columna constants: tot és invariant.

RV32I

```
        la      a0, mat + (3*NC + 11)*4
```

b) `mat[i][11]` — fila variable, columna constant.

RV32IM

```
la      a0, mat + 11*4      # @mat + 11*4 (invariant)
li      t2, NC*4            # NC*4 (invariant)
mul     t3, t0, t2          # i*NC*4
add     a0, a0, t3          # @mat + i*NC*4 + 11*4
```

---

c) **`mat[3][j]`** — fila constant, columna variable.

RV32I

```
la      a0, mat + 3*NC*4    # @mat + 3*NC*4 (invariant)
slli    t2, t1, 2           # j*4
add     a0, a0, t2          # @mat + 3*NC*4 + j*4
```

---

d) **`mat[i][j]`** — fila i columna variables.

RV32IM

```
la      a0, mat             # @mat (invariant)
li      t2, NC              # NC (invariant)
mul     t3, t0, t2          # i*NC
add     t3, t3, t1          # i*NC + j
slli    t3, t3, 2           # (i*NC + j)*4
add     a0, a0, t3          # @mat + (i*NC + j)*4
```

---

e) **`mat[i+5][j-1]`**
Equivalent a `mat[i'][j']` amb `i' = i+5` i `j' = j-1`. Incorporant les constants als invariants:

RV32IM

```
la      a0, mat + (5*NC - 1)*4   # @mat + (5*NC - 1)*4 (invariant)
li      t2, NC                    # NC (invariant)
mul     t3, t0, t2               # i*NC
add     t3, t3, t1               # i*NC + j
slli    t3, t3, 2                # (i*NC + j)*4
add     a0, a0, t3               # @mat + (i*NC + j)*4 + (5*NC-1)*4
```

---

f) **`mat[i*10+4][6]`**
La fila és `i*10+4`, la columna és la constant `6`. La fila ja conté una multiplicació per 10 que no és potència de 2; les dues multiplicacions es fusionen en una: l'invariant `10*NC*4` permet resoldre-ho amb un únic `mul`.

RV32IM

```
la      a0, mat + 4*NC*4 + 6*4  # @mat + (4*NC + 6)*4 (invariant)
li      t2, 10*NC*4              # 10*NC*4 (invariant)
mul     t3, t0, t2              # i*10*NC*4
add     a0, a0, t3              # @mat + (i*10*NC + 4*NC + 6)*4
```

💡 Exemple 49: Solució: elements veïns de la diagonal amb un sol `mul`

Enunciat: Problema 4.16.
Adreces dels dos elements segons l'Equació 4.3 (amb `NC = N` i $T = 4$), reagrupant els termes:

$$@\texttt{M[i][i+1]} = @\texttt{M} + (i \cdot N + i + 1) \cdot 4 = @\texttt{M} + 1 \cdot 4 + i \cdot (N + 1) \cdot 4$$

$$@\texttt{M[i+1][i]} = @\texttt{M} + ((i+1)\cdot N + i)\cdot 4 = @\texttt{M} + N\cdot 4 + i\cdot(N+1)\cdot 4$$

Les dues adreces comparteixen el terme variable $i\cdot(N+1)\cdot 4$: amb **un únic `mul`** n'hi ha prou. La part restant de cada adreça és un invariant que s'integra a la `la` corresponent:

RV32IM

```
        # t0 = i, t1 = suma
        li      t4, (N+1)*4
        mul     t4, t0, t4          # d = i*(N+1)*4  (comú a les dues adreces)
        la      t2, M + 1*4         # invariant: @M[0][1]
        add     t2, t2, t4          # @M[i][i+1]
        lw      t2, 0(t2)           # M[i][i+1]
        la      t3, M + N*4         # invariant: @M[1][0]
        add     t3, t3, t4          # @M[i+1][i]
        lw      t3, 0(t3)           # M[i+1][i]
        sub     t2, t2, t3          # M[i][i+1] - M[i+1][i]
        add     t1, t1, t2          # suma += M[i][i+1] - M[i+1][i]
```

Deu instruccions i una sola multiplicació. La traducció directa (calcular cada adreça per separat amb l'Equació 4.3) necessitaria dues multiplicacions i més instruccions.

Exemple 50: Solució: `func` i `main` amb accés seqüencial

Enunciat: Problema 4.20.

C

```
int mati[5][4];

void func(int veci[4]) {
    int j;
    for (j = 0; j < 4; j++)
        veci[j] = j;
}

void main() {
    int *p = &mati[0][0];
    int *fi = &mati[5][0];
    while (p != fi) {
        func(p);
        p += 4;
    }
}
```

a) Traducció de `func` amb accés seqüencial a `veci`

`func` és **uninivell**. El paràmetre `veci` arriba per `a0` (adreça base del vector). `j` s'assigna a `t0`. L'*stride* del recorregut per fila és 4 bytes (mida d'un `int`). El bucle es controla amb el punter; la variable `j` es manté, perquè se'n necessita el valor (`veci[j] = j`).

RV32I

```
func:
        # a0 = @veci
        mv      t1, a0              # t1 = p = @veci[0]
        addi    t2, a0, 4*4         # t2 = sentinella = @veci[4]
        li      t0, 0               # j = 0
```

```
        bgeu    t1, t2, fibucle1    # comprovació inicial (longitud 4 > 0: mai no salta)
bucle1:
        sw      t0, 0(t1)           # veci[j] = j
        addi    t1, t1, 4           # p += stride (4 bytes)
        addi    t0, t0, 1           # j++
        bltu    t1, t2, bucle1      # salta si p < sentinella
fibucle1:
        ret
```

---

b) Traducció de `main` amb accés seqüencial a `mati`

`main` és **multinivell** (crida `func`). Com que `p` i `fi` han de sobreviure a la crida i `func` (apartat a) utilitza `t0`–`t2`, cal desar-los en **registres segurs** (`s0`, `s1`) en lloc de `t0`/`t1`: `p` → `s0`, `fi` → `s1`. Cal desar `ra`, `s0` i `s1` al bloc d'activació.

RV32I

```
main:
        addi    sp, sp, -12
        sw      ra, 8(sp)
        sw      s1, 4(sp)
        sw      s0, 0(sp)

        la      s0, mati            # s0 = p = @mati[0][0]  (segur: sobreviu a la crida)
        addi    s1, s0, 5*4*4       # s1 = fi = @mati[5][0]
        bgeu    s0, s1, fibucle2    # comprovació inicial (5 > 0: mai no salta)
bucle2:
        mv      a0, s0              # a0 = p
        jal     ra, func
        addi    s0, s0, 4*4         # p += 4  (4 ints = 16 bytes)
        bltu    s0, s1, bucle2
fibucle2:
        lw      s0, 0(sp)
        lw      s1, 4(sp)
        lw      ra, 8(sp)
        addi    sp, sp, 12
        ret
```

4.4 Accés seqüencial

Nota: com a la secció anterior, les solucions deixen les expressions aritmètiques als operands (`N*4`, `(N+1)*4`, `3*(N-1)*4`, i també les directives `.eqv`) per fer visible l'*stride* de cada recorregut, tal com s'admet a la teoria i als exàmens (EC 2.3).

💡 Exemple 51: Solució: recorreguts per fila, columna i diagonal

Enunciat: Problema 4.21.

C

```
void func(int A[N][N]) {
    int i, j, suma = 0;  // i a t0, j a t1, suma a t2
    ...
}
```

`A` arriba per `a0`. `N` és una constant. Es demana accés seqüencial: les variables d'índex desapareixen de la traducció (el control es fa amb el punter i la sentinella), tret de l'apartat d, on `i` es conserva com a comptador. `suma` ocupa `t2`.

a) Suma de la fila 3: `for (i = 0; i < N; i++) suma += A[3][i]`

L'*stride* del recorregut per fila és 4 bytes.

L'adreça del primer element és `@A[3][0] = @A + 3*N*4`.

RV32I

```
        # a0 = @A, t2 = suma (inicialitzada prèviament a 0)
        li      t3, 3*N*4
        add     t3, a0, t3          # t3 = p = @A[3][0]
        li      t4, N*4
        add     t4, t3, t4          # t4 = sentinella = @A[3][N]
        bgeu    t3, t4, fibucle     # comprovació inicial
bucle:
        lw      t5, 0(t3)           # A[3][i]
        add     t2, t2, t5          # suma += A[3][i]
        addi    t3, t3, 4           # p += stride (4 bytes)
        bltu    t3, t4, bucle
fibucle:
```

b) Suma de la columna 4: `for (i = 0; i < N; i++) suma += A[i][4]`

L'*stride* del recorregut per columna és `N*4` bytes.

L'adreça del primer element és `@A[0][4] = @A + 4*4`.

La sentinella és `@A[N][4] = @A[0][4] + N*(N*4)`. Com que `N` és una constant, el desplaçament `N*N*4` també ho és, i va directament a l'operand de `li` (EC 2.3): no cal `mul`.

RV32I

```
        # a0 = @A, t2 = suma (inicialitzada prèviament a 0)
        addi    t3, a0, 4*4         # t3 = p = @A[0][4]
        li      t5, N*4             # stride = N*4
        li      t4, N*N*4           # N²*4
        add     t4, t3, t4          # t4 = sentinella = @A[N][4]
        bgeu    t3, t4, fibucle     # comprovació inicial
bucle:
        lw      t6, 0(t3)           # A[i][4]
        add     t2, t2, t6          # suma += A[i][4]
        add     t3, t3, t5          # p += stride (N*4 bytes)
        bltu    t3, t4, bucle
fibucle:
```

c) Suma de la diagonal principal: `for (i = 0; i < N; i++) suma += A[i][i]`

D'un element `A[i][i]` al següent `A[i+1][i+1]` hi ha $N+1$ elements de distància: l'*stride* és `(N+1)*4` bytes (fila «diagonal principal» de la taula de *strides* de Secció 4.6.4).

El primer element és `@A[0][0] = @A`, i la sentinella, `@A + N*((N+1)*4)`: com a l'apartat b, el desplaçament és constant i va a l'operand de `li`.

RV32I

```
        # a0 = @A, t2 = suma (inicialitzada prèviament a 0)
        mv      t3, a0              # t3 = p = @A[0][0]
        li      t5, (N+1)*4         # stride = (N+1)*4
        li      t4, N*(N+1)*4       # N * (N+1)*4
        add     t4, t3, t4          # t4 = sentinella
        bgeu    t3, t4, fibucle     # comprovació inicial
```

```
bucle:
        lw      t6, 0(t3)           # A[i][i]
        add     t2, t2, t6          # suma += A[i][i]
        add     t3, t3, t5          # p += stride ((N+1)*4 bytes)
        bltu    t3, t4, bucle
fibucle:
```

---

d) Diagonal secundària saltant de 3 en 3: `for (i = 0; i < N; i += 3) suma += A[i][N-1-i]`
Entre dos elements accedits consecutivament, `A[i][N-1-i]` i `A[i+3][N-1-(i+3)]`, la distància en elements és $3N - 3 = 3(N - 1)$: l'*stride* és constant, `3*(N-1)*4` bytes, i l'accés seqüencial és aplicable al càlcul d'adreces.
Ara bé, el nombre d'iteracions és $\lceil N/3 \rceil$ i l'adreça de la sentinella no té una expressió senzilla. En aquest cas és més net **conservar `i` com a comptador** del bucle (la variable d'inducció sobreviu per al control) i avançar el punter només per a l'accés a `A`:

RV32I

```
        # a0 = @A, t0 = i, t2 = suma (inicialitzada prèviament a 0)
        li      t0, 0               # i = 0
        li      t3, (N-1)*4
        add     t3, a0, t3          # t3 = p = @A[0][N-1]
        li      t5, 3*(N-1)*4       # stride = 3*(N-1)*4
        li      t4, N               # invariant
        bge     t0, t4, fibucle     # comprovació inicial
bucle:
        lw      t6, 0(t3)           # A[i][N-1-i]
        add     t2, t2, t6          # suma += A[i][N-1-i]
        add     t3, t3, t5          # p += stride
        addi    t0, t0, 3           # i += 3
        blt     t0, t4, bucle
fibucle:
```

Contrast entre els recorreguts per fila i per columna (apartats a i b): el recorregut per fila (apartat a) té *stride* 4, el recorregut per columna (apartat b) té *stride* `N*4`. Per a valors grans de `N`, el recorregut per columna genera moltes fallades de memòria cau, ja que cada accés aterra en una línia de memòria cau diferent. El recorregut per fila aprofita molt millor la localitat espacial (aquests conceptes s'estudien a Capítol 7).

💡 Exemple 52: Solució: accessos mixtos a matriu i vector

Enunciat: Problema 4.23.
a) `vector[i] = z[4][j]`
Els paràmetres arriben per `a0` (`i`) i `a1` (`j`). La fila de `z[4][j]` és constant: l'invariant `@z[4][0] = @z + 4*3*4` s'integra a la `la` (com a Exemple 4.11) i només cal escalar `j`. L'adreça de `vector[i]` s'obté escalant `i`:

RV32I

```
        la      t0, z + 4*3*4       # invariant: @z[4][0]
        slli    t1, a1, 2           # j*4
        add     t0, t0, t1          # @z[4][j]
        lw      t0, 0(t0)           # z[4][j]
        la      t1, vector          # invariant: @vector
        slli    t2, a0, 2           # i*4
        add     t1, t1, t2          # @vector[i]
```

```
        sw      t0, 0(t1)           # vector[i] = z[4][j]
```

b) Constants del recorregut seqüencial
El bucle accedeix sempre a `vector[2]` (posició fixa) i recorre la columna 2 de `z` (`int z[5][3]`):

- `offset_vector`: distància de `@vector` a `@vector[2]` → `2*4` = 8.
- `offset_z`: distància de `@z` al primer element del recorregut, `z[0][2]` → `2*4` = 8.
- `stride`: d'un element `z[k][2]` al següent, `z[k+1][2]`, hi ha una fila sencera (3 `int`) → `3*4` = 12.
- `max_dist`: la sentinella `t6` s'assoleix (`beq`) després de les 5 iteracions → 5 vegades l'*stride* → `5*3*4` = 60.

RV32I

```
.eqv    offset_vector, 2*4      # 8
.eqv    offset_z,      2*4      # 8
.eqv    max_dist,      5*3*4    # 60
.eqv    stride,        3*4      # 12
```

Exemple 53: Solució: vector de `short` — traducció i inicialització

Enunciat: Problema 4.25.
a) Traducció a C del fragment
Els paràmetres arriben per `a0` (`@p1`) i `a1` (`p2`). El fragment calcula `t2 = p2 + 3`, l'escala per la mida d'un `short` ($T = 2$; per això `slli t2, t2, 1`), el suma a la base `@p1` i hi escriu un zero de 16 bits (`sh`):

C

```
p1[p2 + 3] = 0;
```

b) Inicialització de `p1` amb accés seqüencial
Sentències que falten (`i` a `t0`, `p` a `t1`):

C

```
p = p1;
for (i = 0; i < 100; i++) {
    *p = 0;
    p++;
}
```

Al fragment d'assemblador, els dos buits del cos del bucle són l'escriptura i l'avanç del punter. L'*stride* és $T = 2$ bytes (mida d'un `short`): `p++` es tradueix per `addi t1, t1, 2`, i l'escriptura de 16 bits, per `sh`:

RV32I

```
        mv      t1, a0              # p = p1  (prèvia al fragment mostrat)
        li      t0, 0
        li      t2, 100
bucle:
        bge     t0, t2, fibucle
        sh      zero, 0(t1)         # *p = 0
```

```
        addi    t1, t1, 2           # p++  (2 bytes)
        addi    t0, t0, 1
        j       bucle
fibucle:
```

Nota: la inicialització `p = p1` (`mv t1, a0`) no apareix al fragment de l'enunciat, que comença a `li t0, 0`; s'ha inclòs aquí per completesa.

Exemple 54: Solució: optimització #2 — avaluació de la condició al final

Enunciat: Problema 4.28.
Versió inicial:

`RV32I`

```
        li      t0, 0
bucle:
        li      t1, N
        bge     t0, t1, fibucle
        la      t2, v
        slli    t3, t0, 2
        add     t2, t2, t3
        lw      t3, 0(t2)
        add     t4, t4, t3
        addi    t0, t0, 1
        j       bucle
fibucle:
```

---

a) Cost de la versió inicial
10 instruccions per iteració: les pseudoinstruccions es compten per les instruccions reals, i `la` en són dues, `auipc` i `addi`. Cost per iteració (model de costos: `lw` = 4, salt pres = 2, salt no pres = 1, resta = 1):

Instrucció	Cost
`li t1, N`	1
`bge` (no pres, N vegades)	1
`la t2, v` (`auipc` + `addi`)	2
`slli t3`	1
`add t2`	1
`lw t3`	4
`add t4`	1
`addi t0`	1
`j bucle` (pres)	2
Total per iteració	**14**

Inicialització: `li t0, 0` = 1 cicle.
Última comprovació ($i = N$): `li t1, N` (1) + `bge` pres (2) = 3 cicles.
Total: $1 + 14N + 3 = 14N + 4$ cicles.

b) Versió optimitzada (condició al final)

Es pot fer la comprovació inicial una sola vegada (si `N > 0` sempre es compleix, es pot ometre si es pot garantir). Per ser fidel a la semàntica del `while` original:

RV32I

```
        li      t0, 0
        li      t1, N
        la      t2, v               # extracció d'invariants fora del bucle
        bge     t0, t1, fibucle     # comprovació inicial: salta si N <= 0
bucle:
        slli    t3, t0, 2
        add     t5, t2, t3
        lw      t3, 0(t5)
        add     t4, t4, t3
        addi    t0, t0, 1
        blt     t0, t1, bucle       # salta si i < N
fibucle:
```

S'han extret `li t1, N` i `la t2, v` com a invariants fora del bucle (extracció d'invariants, Secció 2.8.5.1, prerequisit de tota optimització).
Instruccions per iteració:

Instrucció	Cost
`slli t3`	1
`add t5`	1
`lw t3`	4
`add t4`	1
`addi t0`	1
`blt` pres ($N-1$ vegades) / no pres (última)	2 / 1
Total per iteració (menys la darrera)	**10**

Inicialització: `li t0` (1) + `li t1` (1) + `la t2` (2) + `bge` no pres (1) = 5 cicles.
Última iteració: `blt` no pres (1) → cost = $10 - 2 + 1 = 9$ cicles.
Total: $5 + 10(N-1) + 9 = 10N + 4$ cicles.

c) Reducció percentual per a $N = 100$

Versió inicial: $14 \times 100 + 4 = 1404$ cicles.
Versió optimitzada: $10 \times 100 + 4 = 1004$ cicles.

$$\text{Reducció} = \frac{1404 - 1004}{1404} \times 100 \approx \mathbf{28{,}5}\ \%$$

💡 Exemple 55: Solució: optimització #3 — eliminació de la variable d'inducció

Enunciat: Problema 4.29.

C

```
int v[N], w[N];
// for (i = 0; i < N; i++) w[i] = v[i] * 2;
```

a) Versió sense cap optimització (usant `i` explícitament, condició a l'inici)

RV32I

```
        li      t0, 0               # i = 0
        li      t1, N               # N (invariant)
        la      t2, v               # @v (invariant)
        la      t3, w               # @w (invariant)
bucle:
        bge     t0, t1, fibucle     # [ctrl]
        slli    t4, t0, 2           # i*4
        add     t5, t2, t4          # @v[i]
        lw      t5, 0(t5)           # v[i]
        slli    t5, t5, 1           # v[i] * 2
        add     t6, t3, t4          # @w[i]
        sw      t5, 0(t6)           # w[i] = v[i] * 2
        addi    t0, t0, 1           # i++
        j       bucle
fibucle:
```

9 instruccions per iteració: `bge`, `slli`, `add`, `lw`, `slli`, `add`, `sw`, `addi` i `j`. Amb el model de costos de Problema 4.28 (memòria = 4 cicles, salt pres = 2, salt no pres = 1, resta = 1), 16 cicles.

b) Eliminació de la variable d'inducció (punters directes), mantenint la condició a l'inici

Es mantenen dos punters `pv` i `pw` que avancen en paral · lel amb *stride* 4, controlats amb la sentinella de `pv`. La condició es continua avaluant a l'inici de cada iteració (amb el salt incondicional `j` corresponent), sense aplicar encara l'optimització #2.

RV32I

```
        la      t0, v               # t0 = pv = @v[0]
        la      t1, w               # t1 = pw = @w[0]
        li      t2, N*4
        add     t2, t0, t2          # t2 = sentinella = @v[N]
bucle:
        bgeu    t0, t2, fibucle     # [ctrl]
        lw      t3, 0(t0)           # *pv = v[i]
        slli    t3, t3, 1           # v[i] * 2
        sw      t3, 0(t1)           # *pw = v[i] * 2
        addi    t0, t0, 4           # pv++
        addi    t1, t1, 4           # pw++
        j       bucle
fibucle:
```

7 instruccions per iteració: `bgeu`, `lw`, `slli`, `sw`, dues `addi` i `j`. Amb el mateix model de costos, 14 cicles.

c) Combinació amb l'avaluació de la condició al final

Partint de b), s'aplica l'optimització #2: la comprovació es fa una única vegada abans d'entrar al bucle (cas de zero iteracions) i el salt incondicional `j` se substitueix pel salt condicional `bltu` al final:

RV32I

```
        la      t0, v               # t0 = pv = @v[0]
        la      t1, w               # t1 = pw = @w[0]
        li      t2, N*4
        add     t2, t0, t2          # t2 = sentinella = @v[N]
```

```
        bgeu    t0, t2, fibucle     # comprovació inicial
bucle:
        lw      t3, 0(t0)           # *pv = v[i]
        slli    t3, t3, 1           # v[i] * 2
        sw      t3, 0(t1)           # *pw = v[i] * 2
        addi    t0, t0, 4           # pv++
        addi    t1, t1, 4           # pw++
        bltu    t0, t2, bucle
fibucle:
```

El cos del bucle conté:

Instrucció	Nombre
`lw`	1
`slli`	1
`sw`	1
`addi` (pv++)	1
`addi` (pw++)	1
`bltu` (control)	1
Total	**6**

Amb el mateix model de costos, 13 cicles per iteració (12 a la darrera, on `bltu` no és pres).

4.5 Divisió

Exemple 56: Solució: algorisme del divisor seqüencial amb restauració

Enunciat: Problema 4.30.
El circuit (Secció 4.4.2) prova de restar el divisor alineat a cada posició de pes, de la més significativa a la menys significativa. Paper dels registres, per a $n = 32$:

- **R** ($2n$ bits): comença amb el dividend x a la part baixa i zeros a la part alta; en acabar, la part baixa conté el residu.
- **D** ($2n$ bits): comença amb el divisor y a la part **alta** (equival a $y \cdot 2^n$) i es desplaça una posició a la dreta a cada iteració: a la iteració i-èsima queda alineat amb el pes 2^{n-i}.
- **Q** (n bits): acumula els bits del quocient, del més significatiu al menys significatiu.

Algorisme iteratiu complet:

`Pseudocodi: divisió amb restauració`

```
// Inicialització
R[n-1:0] = x;  R[2n-1:n] = 0
D[2n-1:n] = y;  D[n-1:0] = 0
Q = 0

per i = 1 fins a n:
    D = D >> 1
    R = R - D
    si R[2n-1] == 0:        // resultat no negatiu: el divisor "hi cap"
        Q = (Q << 1) | 1
    sino:                   // resultat negatiu: restaurar
```

```
        R = R + D
        Q = Q << 1

q = Q           // quocient
r = R[n-1:0]    // residu
```

A cada iteració es fa la resta de prova R − D: si el resultat no és negatiu (bit R_{2n-1} a 0), el divisor desplaçat «hi cap» i s'anota un 1 al quocient; si és negatiu, es **restaura** el valor de R sumant-hi D de nou, i s'anota un 0. Després de les n iteracions, Q conté el quocient i la part baixa de R, el residu (Algorisme 4.2; per a un seguiment numèric complet, Exemple 4.7 i Solució 57).

Exemple 57: Solució: seguiment del circuit divisor seqüencial amb restauració

Enunciat: Problema 4.31.
El circuit treballa amb naturals. Per a operands de 6 bits, els registres R i D tenen 12 bits i Q en té 6. L'algorisme amb restauració a cada iteració:

1. `D = D >> 1`.
2. `R = R - D`.
3. Si `R[11] == 0` (resultat no negatiu): `Q = (Q << 1) | 1`.
4. Altrament (resultat negatiu): `R = R + D` (restauració); `Q = Q << 1`.

Convenció de la taula: la columna «R − D» mostra el valor intermedi de la resta (abans de decidir si cal restaurar); la columna «R (final)» mostra l'estat ja restaurat si ha calgut; la fila inicial porta «—».
a) $x =$ **`101000`** (40_{10})**,** $y =$ **`010011`** (19_{10})**. Resultat esperat:** $q = 2$**,** $r = 2$**.**

Iter.	D (12 bits)	R − D	Signe	Acció	R (final)	Q
inicial	`010011 000000`	—	—	—	`000000 101000`	`000000`
1	`001001 100000`	`110111 001000`	neg.	restaura; `Q<<1`	`000000 101000`	`000000`
2	`000100 110000`	`111011 111000`	neg.	restaura; `Q<<1`	`000000 101000`	`000000`
3	`000010 011000`	`111110 010000`	neg.	restaura; `Q<<1`	`000000 101000`	`000000`
4	`000001 001100`	`111111 011100`	neg.	restaura; `Q<<1`	`000000 101000`	`000000`
5	`000000 100110`	`000000 000010`	pos.	`Q = (Q<<1)|1`	`000000 000010`	`000001`
6	`000000 010011`	`111111 101111`	neg.	restaura; `Q<<1`	`000000 000010`	`000010`

Quocient: $Q = 000010_2 = 2_{10}$, residu: $R[5:0] = 000010_2 = 2_{10}$ ✓
Verificació: $19 \times 2 + 2 = 40$ ✓
b) $x =$ **`010101`** (21_{10})**,** $y =$ **`100100`** (36_{10})**. Resultat esperat:** $q = 0$**,** $r = 21$**.**
Com que $x < y$, el quocient és 0 i el residu és x en totes les iteracions: el divisor desplaçat sempre és més gran que el dividend parcial i cal restaurar en cada pas.

Iter.	D (12 bits)	R − D	Signe	Acció	R (final)	Q
inicial	`100100 000000`	—	—	—	`000000 010101`	`000000`
1	`010010 000000`	`101110 010101`	neg.	restaura; `Q<<1`	`000000 010101`	`000000`
2	`001001 000000`	`110111 010101`	neg.	restaura; `Q<<1`	`000000 010101`	`000000`
3	`000100 100000`	`111011 110101`	neg.	restaura; `Q<<1`	`000000 010101`	`000000`
4	`000010 010000`	`111110 000101`	neg.	restaura; `Q<<1`	`000000 010101`	`000000`
5	`000001 001000`	`111111 001101`	neg.	restaura; `Q<<1`	`000000 010101`	`000000`
6	`000000 100100`	`111111 110001`	neg.	restaura; `Q<<1`	`000000 010101`	`000000`

Quocient: $Q = 000000_2 = 0_{10}$, residu: $R[5:0] = 010101_2 = 21_{10}$ ✓

Exemple 58: Solució: divisió per restauració en programari

Enunciat: Problema 4.32.
El circuit (Algorisme 4.2) treballa amb registres R i D de $2n = 64$ bits, que no caben en un registre d'RV32I. En lloc de simular-los amb parells de registres, s'usa una **formulació equivalent**: mantenir el divisor fix i desplaçar R a l'**esquerra** (desplaçar D a la dreta respecte de R equival a desplaçar R a l'esquerra respecte del divisor), injectant a cada pas el bit següent del dividend. Així tots els valors caben en registres de 32 bits. A més, en lloc de restar i restaurar, és més senzill **comparar abans de restar** (`bltu`), que és equivalent: només es fa la resta quan el resultat no seria negatiu.
Registres: `t3` = dividend x (es va desplaçant; al final es reutilitza per al residu, com demana l'enunciat), `t1` = divisor y, `t2` = quocient (Q), `t4` = residu parcial (R), `t5` = comptador, `t6` = bit extret.

`RV32I`

```
        li      t2, 0               # Q = 0
        li      t4, 0               # R = 0
        li      t5, 32              # 32 iteracions (una per bit)
bucle:
        slli    t4, t4, 1           # R = R << 1
        srli    t6, t3, 31          # bit més significatiu de X
        or      t4, t4, t6          # R = (R << 1) | X[31]
        slli    t3, t3, 1           # X = X << 1 (consumeix el bit)
        slli    t2, t2, 1           # Q = Q << 1
        bltu    t4, t1, seguent     # si R < y: bit de quocient = 0
        sub     t4, t4, t1          # R = R - y
        ori     t2, t2, 1           # Q = (Q << 1) | 1
seguent:
        addi    t5, t5, -1
```

```
        bnez    t5, bucle
        mv      t3, t4              # residu a t3
```

El bucle s'executa exactament 32 vegades: el nombre d'iteracions és conegut i positiu, de manera que no cal comprovació inicial i el salt condicional va al final (Secció 4.6.2). En acabar, `t2` conté el quocient i `t3` el residu.

Tema 5: Coma flotant

5.1 Representació IEEE 754

Exemple 59: Solució: interpretació de patrons de bits

Enunciat: Problema 5.1.
Es treballa amb `A = 0x407302B3` i `B = 0xF9C40513`.
a) Com a naturals
Conversió directa hexadecimal → decimal:

$$A = \texttt{0x407302B3} = 4 \cdot 16^7 + 0 \cdot 16^6 + \cdots = 1\,081\,279\,155$$

$$B = \texttt{0xF9C40513} = 15 \cdot 16^7 + 9 \cdot 16^6 + \cdots = 4\,190\,373\,139$$

b) Com a enters en complement a 2
El bit de signe és el bit 31, que correspon al dígit hexadecimal de més pes.
`A = 0x407302B3`: quartet de més pes `4` = `0100` → bit 31 = `0` → valor **positiu**; coincideix amb el natural:

$$A = +1\,081\,279\,155$$

`B = 0xF9C40513`: quartet de més pes `F` = `1111` → bit 31 = `1` → valor **negatiu**. Per obtenir el valor absolut, s'inverteixen tots els bits i se suma 1:

$$\overline{\texttt{0xF9C40513}} + 1 = \texttt{0x063BFAED} = 104\,594\,157$$

$$B = -104\,594\,157$$

c) Com a nombres en coma flotant IEEE 754 de precisió simple
Es descompon cada valor en camps S (bit 31), E_u (bits 30:23) i F (bits 22:0):

`Càlcul A = 0x407302B3`

```
0 | 1000 0000 | 111 0011 0000 0010 1011 0011
S = 0  (positiu)
Eu = 1000 0000b = 128  →  e = 128 - 127 = 1
F  = 111 0011 0000 0010 1011 0011
Valor    = 1,111 0011 0000 0010 1011 0011 × 2¹
```

`Càlcul B = 0xF9C40513`

```
1 | 1111 0011 | 100 0100 0000 0101 0001 0011
S = 1  (negatiu)
Eu = 1111 0011b = 243  →  e = 243 - 127 = 116
F  = 100 0100 0000 0101 0001 0011
Valor    = 1,100 0100 0000 0101 0001 0011 × 2¹¹⁶
```

Per a `A` ($e = 1$, es desplaça la coma 1 posició a la dreta):

$$A = +11{,}1100\,1100\,0000\,1010\,1100\,11_2$$

$$= 2 + 1 + 2^{-1} + 2^{-2} + ... \approx \mathbf{3{,}797}$$

Per a `B` ($e = 116$, valor molt gran en mòdul):

$$B \approx -1{,}531 \times 2^{116} \approx \mathbf{-1{,}272 \times 10^{35}}$$

d) Com a instruccions RISC-V
S'identifica l'opcode (bits 6:0) i es descodifica el format:

```
Càlcul A = 0x407302B3
```

```
opcode = 011 0011b = 0x33  →  OP (tipus R)
funct7 = 010 0000b = 0x20
funct3 = 000b
rd  = 00101b = x5  (t0)
rs1 = 00110b = x6  (t1)
rs2 = 00111b = x7  (t2)
funct7=0x20, funct3=0 → SUB
```

A : `sub t0, t1, t2`

```
Càlcul B = 0xF9C40513
```

```
opcode = 001 0011b = 0x13  →  OP-IMM (tipus I)
funct3 = 000b  →  ADDI
rd  = 01010b = x10  (a0)
rs1 = 01000b = x8   (s0)
imm = 1111 1001 1100b = -100  (extensió de signe: bit 11 = 1)
```

B : `addi a0, s0, -100`

Atenció: el mateix patró de 32 bits representa quatre coses completament diferents: un natural, un enter, un nombre en coma flotant i una instrucció de màquina. El maquinari no distingeix entre elles: és el programari (compilador, sistema operatiu) qui decideix com s'interpreta cada zona de memòria.

Exemple 60: Solució: classificació de patrons IEEE 754

Enunciat: Problema 5.3.
La classificació només depèn dels camps E (exponent) i F (fracció), segons les codificacions especials (Secció 5.1.6):

E	F	Tipus
`0000 0000`	$= 0$	zero (± 0, segons el signe)
`0000 0000`	$\neq 0$	subnormal
`1111 1111`	$= 0$	infinit ($\pm\infty$, segons el signe)
`1111 1111`	$\neq 0$	NaN
qualsevol altre	qualsevol	normalitzat

Aplicant aquestes regles a cada fila:

Signe	Exponent	Fracció	NRM	DNRM	0	INF	NaN
0	`0000 0000`	`110 0010 0000 1110 1110 1011`		X			
0	`0000 0000`	`000 0000 0000 0000 0000 0000`			X		
0	`0010 0100`	`000 0000 0000 0000 0000 0000`	X				
1	`1111 1111`	`000 0000 0000 0000 0000 0000`				X	
0	`0010 0100`	`110 0010 0000 1110 1110 1011`	X				
1	`0000 0000`	`000 0000 0000 0000 0000 0000`			X		
0	`1111 1111`	`101 0001 0001 0000 1001 0100`					X

Observacions:

- Files 1 i 2: amb $E = 0$, el que distingeix el zero del subnormal és la fracció ($F = 0 \rightarrow$ zero; $F \neq 0 \rightarrow$ subnormal).
- Fila 3: $F = 0$ amb exponent normalitzat ($E = 0010\,0100_2 = 36$) és un nombre normalitzat vàlid: la potència de 2 exacta $+1{,}0 \times 2^{36-127} = 2^{-91}$ (Essencial 5.5).
- Files 4 i 7: amb $E = 255$, la fracció distingeix l'infinit ($F = 0$) del NaN ($F \neq 0$).
- Fila 6: el zero té signe: aquesta fila codifica -0.

Exemple 61: Solució: rang del format IEEE 754 de precisió simple

Enunciat: Problema 5.4.

a) Nombre positiu més gran representable

L'exponent vàlid més gran en format normalitzat és $e = +127$ (correspon a $E_u = 254$; $E_u = 255$ és reservat per a infinit i NaN). La fracció més gran possible és tota la fracció a `1`:

$$F = 1111\,1111\,1111\,1111\,1111\,111_2 = 1 - 2^{-23}$$

Mantissa completa (amb bit ocult): $1{,}1111\ldots1_2 = 2 - 2^{-23}$.

$$v_{max} = (2 - 2^{-23}) \times 2^{127} = 2^{128} - 2^{104}$$

`Càlcul`

```
S = 0
Eu = 1111 1110b = 254
F  = 111 1111 1111 1111 1111 1111b
→  0x7F7FFFFF
```

$$\boxed{v_{max} = (2 - 2^{-23}) \times 2^{127} \approx 3{,}403 \times 10^{38} = \texttt{0x7F7FFFFF}}$$

b) Nombre positiu no nul més petit en format normalitzat

L'exponent vàlid normalitzat més petit és $e = -126$ ($E_u = 1$). La mantissa normalitzada més petita és $1{,}000\ldots0$ (fracció a zero, bit ocult = 1):

`Càlcul`

```
S = 0
Eu = 0000 0001b = 1
F  = 000 0000 0000 0000 0000 0000b
→  0x00800000
```

$$v_{\text{min,norm}} = 1{,}0 \times 2^{-126} \approx 1{,}175 \times 10^{-38} = \texttt{0x00800000}$$

c) Nombre positiu no nul més petit en format subnormal
En format subnormal ($E_u = 0$), el valor és $0{,}F \times 2^{-126}$. El valor no nul més petit s'obté amb un sol bit a `1` a la posició menys significativa de la fracció (1 ULP):

Càlcul

```
S = 0
Eu = 0000 0000b = 0
F  = 000 0000 0000 0000 0000 0001b
→  0x00000001
```

$$v_{\text{min,denorm}} = 2^{-23} \times 2^{-126} = 2^{-149} \approx 1{,}401 \times 10^{-45} = \texttt{0x00000001}$$

Exemple 62: Solució: codificació de decimal a IEEE 754

Enunciat: Problema 5.5.
Els apartats a), c), d) i e) tenen part fraccionària que és suma **finita** de potències negatives de 2 (0; $0{,}125 = 2^{-3}$; $0{,}75 = 2^{-1} + 2^{-2}$; $0{,}5 = 2^{-1}$): la conversió és **exacta** i no cal arrodonir. L'apartat b) té part fraccionària periòdica en binari i requereix arrodoniment.
a) 340 (exacte):

a) 340

```
340 = 1 0101 0100b = 1,01010100 × 2⁸
e = 8  →  Eu = 8 + 127 = 135 = 1000 0111b
F = 010 1010 0000 0000 0000 0000
0 | 1000 0111 | 010 1010 0000 0000 0000 0000  =  0x43AA0000
```

c) −255,125 (exacte):

c) -255,125

```
255 = 1111 1111b     0,125 = 0,001b
-255,125 = -1111 1111,001b = -1,1111111001 × 2⁷
e = 7  →  Eu = 134 = 1000 0110b
F = 111 1111 0010 0000 0000 0000
1 | 1000 0110 | 111 1111 0010 0000 0000 0000  =  0xC37F2000
```

d) −10,75 (exacte):

d) -10,75

```
10 = 1010b     0,75 = 0,11b
-10,75 = -1010,11b = -1,01011 × 2³
e = 3  →  Eu = 130 = 1000 0010b
F = 010 1100 0000 0000 0000 0000
1 | 1000 0010 | 010 1100 0000 0000 0000 0000  =  0xC12C0000
```

e) 6,5 (exacte):

e) 6,5

```
6 = 110b     0,5 = 0,1b
6,5 = 110,1b = 1,101 × 2²
e = 2  →  Eu = 129 = 1000 0001b
F = 101 0000 0000 0000 0000 0000
0 | 1000 0001 | 101 0000 0000 0000 0000 0000  =  0x40D00000
```

---

b) 44,4 (amb arrodoniment):
Pas 1. Conversió a binari. Part entera: $44 = 10\,1100_2$. Part fraccionària, per multiplicacions successives per 2: $0{,}4 \to 0{,}8 \to 1{,}6 \to 1{,}2 \to 0{,}4 \to \ldots$, que dona l'expansió **periòdica**:

$$0{,}4 = 0{,}0110\,0110\,0110\ldots_2$$

Pas 2. Normalitzar:

$$44{,}4 = 10\,1100{,}0110\,0110\,0110\ldots_2 = 1{,}0110\,0011\,0011\,0011\,0011\,001\,\underbrace{1}_{G}\,\underbrace{0}_{R}\,\underbrace{0110\ldots}_{S} \times 2^5$$

Pas 3. Arrodonir (RNE). Els bits descartats són $G = 1$, $R = 0$ i $S = 1$ (la cua periòdica conté uns): com que $G = 1$ i $(R \vee S) = 1$, s'arrodoneix **al següent** (+1 al bit de menys pes):

```
Arrodoniment de la fracció (23 bits)

F truncada   = 011 0001 1001 1001 1001 1001   (G=1, R=0, S=1)
F arrodonida = 011 0001 1001 1001 1001 1010
```

Pas 4. Codificar: $e = 5 \Rightarrow E_u = 132 = 1000\,0100_2$:

$$V = 0 \mid 1000\,0100 \mid 011\,0001\,1001\,1001\,1001\,1010 = \texttt{0x4231999A}$$

Pas 5. Error de representació. Desfent la normalització de la mantissa arrodonida (coma 5 posicions a la dreta), la part entera (44) no ha canviat: l'error és la diferència entre les parts fraccionàries. La part fraccionària emmagatzemada val:

$$0{,}0110\,0110\,0110\,0110\,10_2 = 1\,1001\,1001\,1001\,1010_2 \times 2^{-18} = \frac{104\,858}{262\,144} = \frac{52\,429}{131\,072}$$

$$\varepsilon = \left|\frac{52\,429}{131\,072} - 0{,}4\right| = \frac{52\,429 - 0{,}4 \times 131\,072}{131\,072} = \frac{0{,}2}{131\,072} \approx 1{,}526 \times 10^{-6}$$

Resum:

Apartat	Valor	IEEE 754	Error
a)	340	`0x43AA0000`	exacte
b)	44,4	`0x4231999A`	$\approx 1{,}526 \times 10^{-6}$
c)	−255,125	`0xC37F2000`	exacte
d)	−10,75	`0xC12C0000`	exacte
e)	6,5	`0x40D00000`	exacte

Exemple 63: Solució: precisió simple i doble, error de representació

Enunciat: Problema 5.6.
Es treballa amb $A = -1609{,}5$ i $B = -938{,}8125$.

$A = -1609{,}5$
Part entera:

$$1609 = 1024 + 512 + 64 + 8 + 1 = 2^{10} + 2^9 + 2^6 + 2^3 + 2^0 = 11001001001_2$$

Part fraccionària:

$$0{,}5 = 0{,}1_2$$

$$A = -11001001001{,}1_2 = -1{,}10010010011_2 \times 2^{10}$$

La representació binària és **exacta** (la part fraccionària és $0{,}5 = 2^{-1}$, representable en 1 bit).
Precisió simple (23 bits de fracció):
Els 11 bits de la fracció caben àmpliament als 23 bits disponibles: no hi ha arrodoniment.

Càlcul

```
S = 1
e = 10  →  Eu = 10 + 127 = 137 = 1000 1001b
F = 100 1001 0011 0000 0000 0000  (omplim amb zeros fins a 23 bits)
→  0xC4C93000
```

$$\boxed{A_{\text{simple}} = \texttt{0xC4C93000}, \quad \varepsilon_A = 0}$$

Precisió doble (52 bits de fracció):
Igualment, sense arrodoniment (els 11 bits caben en 52).

Càlcul

```
S = 1
e = 10  →  Eu = 10 + 1023 = 1033 = 100 0000 1001b
F = 1001 0010 0110 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000  (zeros fins a 52 bits)
→  0xC099260000000000
```

$$\boxed{A_{\text{doble}} = \texttt{0xC099260000000000}, \quad \varepsilon_A = 0}$$

$B = -938{,}8125$
Part entera:

$$938 = 512 + 256 + 128 + 32 + 8 + 2 = 2^9 + 2^8 + 2^7 + 2^5 + 2^3 + 2^1 = 1110101010_2$$

Part fraccionària:

$$0{,}8125 = 0{,}5 + 0{,}25 + 0{,}0625 = 2^{-1} + 2^{-2} + 2^{-4} = 0{,}1101_2$$

$$B = -1110101010{,}1101_2 = -1{,}1101010101101_2 \times 2^9$$

La representació binària és **exacta** (la part fraccionària és 0,8125, representable en 4 bits).
Precisió simple:
La fracció té 13 bits (sense el bit ocult), que caben en els 23 bits disponibles: sense arrodoniment.

Càlcul

```
S = 1
e = 9  →  Eu = 9 + 127 = 136 = 1000 1000b
F = 110 1010 1011 0100 0000 0000  (zeros fins a 23 bits)
→  0xC46AB400
```

$$\boxed{B_{\text{simple}} = \texttt{0xC46AB400}, \quad \varepsilon_B = 0}$$

Precisió doble:
Igualment, sense arrodoniment.

`Càlcul`

```
S = 1
e = 9  →  Eu = 9 + 1023 = 1032 = 100 0000 1000b
F = 1101 0101 0110 1000 0000 ... (zeros fins a 52 bits)
→  0xC08D568000000000
```

$$\boxed{B_{\text{doble}} = \texttt{0xC08D568000000000}, \quad \varepsilon_B = 0}$$

Atenció: en aquest problema, tant A com B resulten ser exactament representables en precisió simple i doble: les seves parts fraccionàries (0,5 i 0,8125) són sumes de potències negatives de 2 amb un nombre finit de termes. En general, però, la majoria de decimals (per exemple, 0,1 o 0,3) **no** són representables exactament en cap format binari de precisió finita i introdueixen sempre un error de representació.

5.2 Operacions

Exemple 64: Solució: suma de nombres en coma flotant

Enunciat: Problema 5.9.
Descodificació dels operands

`ft2 = 0x417AC000`

```
0 | 1000 0010 | 111 1010 1100 0000 0000 0000
S=0, Eu=130 → e = 130−127 = 3
Mantissa = 1,111 1010 1100 0000 0000 0000
Valor = 1,111 1010 1100b × 2³ = 15,671875
```

`ft4 = 0x3F140000`

```
0 | 0111 1110 | 001 0100 0000 0000 0000 0000
S=0, Eu=126 → e = 126−127 = −1
Mantissa = 1,001 0100 0000 0000 0000 0000
Valor = 1,001 0100 0000b × 2⁻¹ = 0,578125
```

Pas 1. Alinear exponents
L'exponent més petit és $e = -1$ (ft4). Cal desplaçar la seva mantissa 4 posicions a la dreta per igualar $e = 3$:

`Alineació`

```
ft2:            1,1111010110 0000000000000          × 2³
ft4 original:   1,0010100000 0000000000000          × 2⁻¹
ft4 alineat:    0,0001001010 0000000000000          × 2³
                (els 4 bits que surten per la dreta són 0: G=0, R=0, S=0)
```

Pas 2. Suma de mantisses

Suma

```
  1,1111010110 0000000000000
+ 0,0001001010 0000000000000
= 10,0000100000 0000000000000
  ↑
  ròssec = 1 → cal renormalitzar
```

Pas 3. Renormalitzar

El resultat té un `1` extra a l'esquerra de la coma. Es desplaça 1 posició a la dreta i s'incrementa l'exponent:

Renormalització

```
10,0000100000 0000000000000 × 2³
→  1,0000010000 0000000000000 × 2⁴   (G=0, sense arrodoniment)
```

Pas 4. Codificar el resultat

$$e = 4 \Rightarrow E_u = 4 + 127 = 131 = 1000\,0011_2$$

$$F = 000\,0010\,0000\,0000\,0000\,0000$$

$$\texttt{ft6} = 0 \mid 1000\,0011 \mid 000\,0010\,0000\,0000\,0000\,0000 = \texttt{0x41820000}$$

a) $\boxed{\texttt{ft6} = \texttt{0x41820000}}$
b) G = 0, R = 0, S = 0 → no cal arrodonir. **No hi ha error de precisió.**
c) $e = 4$, mantissa $= 1{,}0000010_2 = 1 + 2^{-6} = 1{,}015625$:

$$\texttt{ft6} = 1{,}015625 \times 2^4 = \mathbf{16{,}25}$$

Comprovació: $15{,}671875 + 0{,}578125 = 16{,}25$ ✓

💡 Exemple 65: Solució: suma en format *half* amb bits de guarda

Enunciat: Problema 5.12.
Format *half* (Problema 5.7): 1 bit de signe, 5 bits d'exponent esbiaixat (biaix 15), 10 bits de fracció (amb bit ocult). El sumador manté els bits G, R i S (Secció 5.1.7.1) i arrodoneix amb RNE.

a) `A = 0xE4FE, B = 0xB640`

Descodificació

A = 0xE4FE = 1 11001 0011111110

```
S=1, Eu=11001b=25 → e = 25−15 = 10
Mantissa = 1,0011111110
Valor = −1,0011111110b × 2¹⁰ = −100 1111 1110b = −1278
```

```
B = 0xB640 = 1 01101 1001000000

S=1, Eu=01101b=13 → e = 13−15 = −2
Mantissa = 1,1001000000
Valor = −1,1001b × 2⁻² = −0,011001b = −0,390625
```

Pas 1. Alinear exponents
La diferència d'exponents és $10-(-2)=12$: la mantissa de B es desplaça 12 posicions a la dreta. Els seus bits cauen tots **fora** dels 10 bits de fracció:

```
Alineació (Δe = 12)

A:          1,0011111110                  × 2¹⁰
B original: 1,1001000000                  × 2⁻²
B alineat:  0,0000000000 | 0 1 | 1001…    × 2¹⁰
            ← 10 bits →    G R    S
```

El bit ocult de B arriba a la posició 2^{-12} (R); per tant, $G=0$, $R=1$ i $S=1$ (els bits `1001` restants contenen uns).
Pas 2. Sumar les mantisses
Els dos operands són negatius (mateix signe): se sumen les magnituds. Dins dels 10 bits de fracció, B només aporta zeros, així que la mantissa no canvia:

```
Suma

  1,0011111110 | 011…
= 1,0011111110   amb G=0, R=1, S=1     (sense transport: no cal renormalitzar)
```

Pas 3. Arrodonir (RNE)
$G=0 \rightarrow$ es **trunca**. Cal notar que $R=S=1$ no basta per arrodonir cap amunt: si $G=0$, el residu descartat és inferior a ½ ULP i la decisió és sempre truncar (Secció 5.1.7.1).

$$\boxed{A+B=-1278=\texttt{0xE4FE}\ (=A)}$$

Precisió: B queda **absorbit** completament pel resultat. L'error absolut és $|B|=0{,}390625$, per sota de ½ ULP (l'ULP a $e=10$ val $2^{-10}\times 2^{10}=1$): la fita d'error d'RNE es compleix.
b) `A = 0x4DC7, B = 0x391D`
Descodificació

```
A = 0x4DC7 = 0 10011 0111000111

S=0, Eu=10011b=19 → e = 19−15 = 4
Mantissa = 1,0111000111
Valor = 1,0111000111b × 2⁴ = 1 0111,000111b = 23,109375
```

```
B = 0x391D = 0 01110 0100011101

S=0, Eu=01110b=14 → e = 14−15 = −1
Mantissa = 1,0100011101
Valor = 1,0100011101b × 2⁻¹ = 0,10100011101b = 0,63916015625
```

Pas 1. Alinear exponents
Diferència d'exponents: $4-(-1)=5$; la mantissa de B es desplaça 5 posicions a la dreta:

```
Alineació (Δe = 5)
```

```
A:          1,0111000111                  × 2⁴
B original: 1,0100011101                  × 2⁻¹
B alineat:  0,0000101000 | 1 1 | 101      × 2⁴
            ← 10 bits →    G R    S
```

Per a B alineat: $G = 1$, $R = 1$ i $S = 1$ (els bits `101` que queden més enllà contenen uns).

Pas 2. Sumar les mantisses

`Suma`

```
  1,0111000111 | 000
+ 0,0000101000 | 111
= 1,0111101111 | 111        (sense transport: no cal renormalitzar)
```

Pas 3. Arrodonir (RNE)

$G = 1$ i $R = 1 \rightarrow$ s'arrodoneix **al següent** (+1 al bit de menys pes):

`Arrodoniment`

```
  1,0111101111
+ 0,0000000001
= 1,0111110000
```

Pas 4. Codificar el resultat

$$e = 4 \Rightarrow E_u = 19 = 10011_2 \qquad F = 01\,1111\,0000_2$$

$$0 \mid 10011 \mid 0111110000 = \texttt{0x4DF0}$$

$$\boxed{A + B = \texttt{0x4DF0} = 1\,0111{,}11_2 = 23{,}75}$$

Precisió: la suma exacta és $23{,}109375 + 0{,}63916015625 = 23{,}74853515625$; l'error d'arrodoniment és

$$\varepsilon = 23{,}75 - 23{,}74853515625 = 0{,}00146484375$$

inferior a ½ ULP ($= 2^{-11} \times 2^4 = 2^{-7} = 0{,}0078125$).

Exemple 66: Solució: multiplicació en format *half*

Enunciat: Problema 5.13.

Se segueix l'algorisme de multiplicació (Secció 5.2.2): sumar exponents, calcular el signe ($S_A \oplus S_B$), multiplicar les mantisses (11 bits × 11 bits → fins a 22 bits), normalitzar comprovant el desbordament, arrodonir amb RNE i codificar.

a) `A = 0x45A9, B = 0x484C`

Descodificació

`A = 0x45A9 = 0 10001 0110101001`

```
S=0, Eu=10001b=17 → e = 17−15 = 2
Mantissa = 1,0110101001  (= 1449/1024)
Valor = 1,0110101001b × 2² = 101,10101001b = 5,66015625
```

`B = 0x484C = 0 10010 0001001100`

```
S=0, Eu=10010b=18 → e = 18−15 = 3
Mantissa = 1,0001001100  (= 1100/1024)
Valor = 1,00010011b × 2³ = 1000,10011b = 8,59375
```

Pas 1. Exponent i signe: $e = 2 + 3 = 5$; signe $= 0 \oplus 0 = 0$ (positiu).
Pas 2. Multiplicar les mantisses

`Producte de mantisses`

```
1449 × 1100 = 1 593 900
(1449/1024) × (1100/1024) = 1 593 900 / 2²⁰
1 593 900 = 1 1000 0101 0010 0010 1100b          (21 bits)
Producte  = 1,1000 0101 0010 0010 1100           (1 ≤ producte < 2)
```

Pas 3. Normalitzar: el producte ja és a $[1, 2)$: no cal renormalitzar; $e = 5$.
Pas 4. Arrodonir (RNE): es conserven 10 bits de fracció; els descartats són $G = 1$, $R = 0$ i $S = 1$ (`0010 1100` conté uns) → s'arrodoneix **al següent**:

`Arrodoniment`

```
F truncada   = 1000010100   (G=1, R=0, S=1)
F arrodonida = 1000010101
```

Pas 5. Codificar

$$e = 5 \Rightarrow E_u = 20 = 10100_2$$

$$0 \mid 10100 \mid 1000010101 = \texttt{0x5215}$$

$$\boxed{A \times B = \texttt{0x5215} = 11\,0000{,}10101_2 = 48{,}65625}$$

Precisió: el producte exacte és $5{,}66015625 \times 8{,}59375 = 48{,}6419677734375$; l'error d'arrodoniment és

$$\varepsilon = 48{,}65625 - 48{,}6419677734375 = 0{,}0142822265625$$

inferior a ½ ULP ($= 2^{-11} \times 2^5 = 2^{-6} = 0{,}015625$).

b) `A = 0x60D4, B = 0x533E`

Descodificació

`A = 0x60D4 = 0 11000 0011010100`

```
S=0, Eu=11000b=24 → e = 24−15 = 9
Mantissa = 1,0011010100  (= 1236/1024)
Valor = 1,00110101b × 2⁹ = 10 0110 1010b = 618
```

`B = 0x533E = 0 10100 1100111110`

```
S=0, Eu=10100b=20 → e = 20−15 = 5
Mantissa = 1,1100111110  (= 1854/1024)
Valor = 1,1100111110b × 2⁵ = 11 1001,11110b = 57,9375
```

Pas 1. Exponent i signe: $e = 9 + 5 = 14$; signe positiu.
Pas 2. Multiplicar les mantisses

`Producte de mantisses`

```
1236 × 1854 = 2 291 544
2 291 544 = 10 0010 1111 0111 0101 1000b          (22 bits)
Producte  = 10,0010 1111 0111 0101 1000           (≥ 2: cal renormalitzar)
```

Pas 3. Normalitzar: es desplaça 1 posició a la dreta i s'incrementa l'exponent: $e = 14 + 1 = 15$. **Comprovació de desbordament**: $e = 15$ és exactament l'exponent màxim del format ($E_u = 30$; $E_u = 31$ està reservat per a $\pm\infty$ i NaN): el resultat queda **just al límit**, sense desbordament.

`Producte normalitzat`

```
1,0001 0111 1011 1010 1100 0   × 2¹⁵
```

Pas 4. Arrodonir (RNE): $G = 1$, $R = 1$ ($S = 1$) → s'arrodoneix **al següent**:

`Arrodoniment`

```
F truncada   = 0001011110   (G=1, R=1, S=1)
F arrodonida = 0001011111
```

Pas 5. Codificar

$$e = 15 \Rightarrow E_u = 30 = 11110_2$$

$$0 \mid 11110 \mid 0001011111 = \texttt{0x785F}$$

$$\boxed{A \times B = \texttt{0x785F} = 100\,0101\,1111_2 \times 2^5 = 1119 \times 2^5 = 35\,808}$$

Precisió: el producte exacte és $618 \times 57{,}9375 = 35\,805{,}375$; l'error d'arrodoniment és

$$\varepsilon = 35\,808 - 35\,805{,}375 = 2{,}625$$

Sembla un error gran, però és inferior a ½ ULP ($= 2^{-11} \times 2^{15} = 2^4 = 16$): en coma flotant l'ULP —i, per tant, l'error absolut— **creix amb l'exponent**; el que està acotat és l'error **relatiu** (aquí $2{,}625/35\,805{,}375 \approx 7{,}3 \times 10^{-5}$).

💡 Exemple 67: Solució: traducció de sentències amb `float`

Enunciat: Problema 5.15.
En RV32F **no hi ha operands immediats flotants**: cap instrucció admet una constant de coma flotant directament. Per usar un literal cal declarar-lo al segment `.data` i llegir-lo de memòria amb `la` + `flw`.
a) **`f = 3.14;`**

`RV32IF`

```
.data
const_pi:
        .float  3.14

.text
        la      t0, const_pi    # t0 ← @const_pi
        flw     fa2, 0(t0)      # fa2 ← 3.14  (f = 3.14)
```

---

b) `f = g;`
Per copiar entre registres de coma flotant s'usa la pseudoinstrucció `fmv.s`.

RV32IF

```
        fmv.s   fa2, fa4        # fa2 ← fa4  (f = g)
```

c) `f = g + 3.14;`
Cal llegir primer la constant de memòria i després fer la suma.

RV32IF

```
.data
const_pi:
        .float  3.14

.text
        la      t0, const_pi    # t0 ← @const_pi
        flw     ft0, 0(t0)      # ft0 ← 3.14
        fadd.s  fa2, fa4, ft0   # fa2 ← fa4 + 3.14  (f = g + 3.14)
```

Atenció: la directiva `.float` accepta decimals: `.float 3.14` converteix el valor a IEEE 754 de precisió simple en temps d'assemblatge. Però les instruccions de la família RV32F mai admeten literals numèrics flotants: tots els operands han d'estar en registres de coma flotant.

Exemple 68: Solució: funció `variancia` amb crides a subrutina

Enunciat: Problema 5.17.
a) Anàlisi de les variables locals
`variancia` és una subrutina no fulla (*non-leaf*): crida a `mitjana` dues vegades.

Variable	Tipus	Viu a través d'alguna crida?	Decisió
`vec` (paràmetre)	`float *`	Sí: es fa servir al bucle (abans de la 1a crida) i a la 2a crida (`m = mitjana(vec)`)	registre segur (cal preservar-lo)
`vquadrats[100]`	vector local (400 bytes)	Sí, però massa gran per a registres	al BA
`i`	`int`	No: només viu dins el bucle, s'esgota abans de la 1a crida	registre temporal
`q`	`float`	Sí: s'assigna a la 1a crida i es fa servir després de la 2a (`return q - m*m`)	registre segur
`m`	`float`	No: s'assigna després de la 2a crida i es fa servir tot seguit	registre temporal

Només `vec` i `q` han de **sobreviure a alguna crida** i, per tant, necessiten un registre **segur** (`s/fs`) o el BA; `i` i `m` es podrien allotjar en registres temporals sense cap risc. Per uniformitat i simplicitat del codi es trien igualment tots quatre en registres segurs:

- `vec` → `s0` (enter, preservat — **necessari**: sobreviu a una crida)
- `q` → `fs1` (flotant, preservat — **necessari**: sobreviu a una crida)
- `i` → `s1` (enter, preservat — no era imprescindible; un temporal `t` hauria bastat)
- `m` → `fs0` (flotant, preservat — no era imprescindible; un temporal `ft` hauria bastat)
- `vquadrats` → BA (400 bytes)

Registres a preservar a la pila: `ra` (adreça de retorn), `s0`, `s1`, `fs0`, `fs1`. Abans de cada crida no cal desar cap registre temporal: tots els valors que la travessen són en registres segurs.

b) Traducció a RISC-V

Bloc d'activació de 420 bytes (Figura 5.1), múltiple de 4:

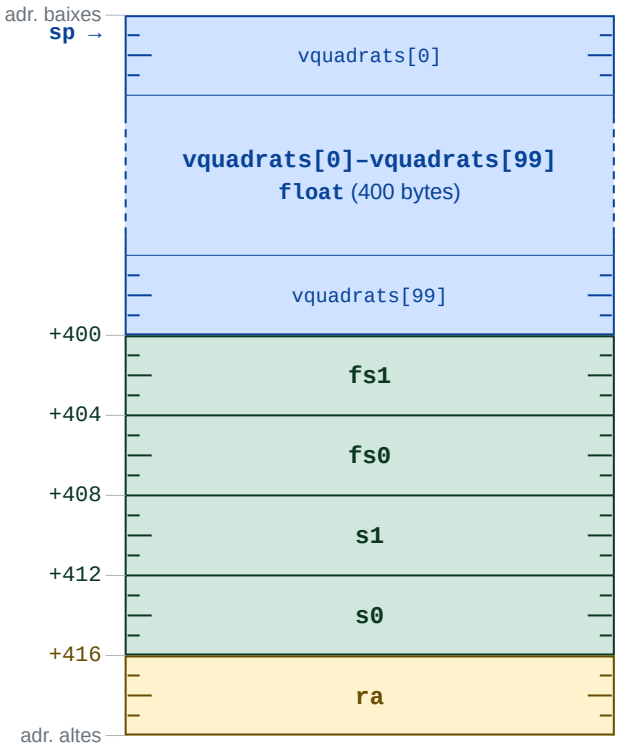


Figura 5.1: Bloc d'activació de `variancia`: el vector `vquadrats` al desplaçament `+0` i els registres desats `fs1`, `fs0`, `s1`, `s0` i `ra` als desplaçaments `+400`, `+404`, `+408`, `+412` i `+416`.

RV32IF

```
variancia:
        # Pròleg
        addi    sp, sp, -420
        sw      ra,  416(sp)
        sw      s0,  412(sp)
        sw      s1,  408(sp)
        fsw     fs0, 404(sp)
        fsw     fs1, 400(sp)

        mv      s0, a0          # s0 ← vec (preserva l'argument)
```

```
        # Bucle: vquadrats[i] = vec[i] * vec[i]  per a i = 0..99
        li      s1, 0           # i = 0
bucle:
        li      t0, 100
        bge     s1, t0, fibucle
        slli    t1, s1, 2       # t1 ← i * 4
        add     t2, s0, t1      # t2 ← &vec[i]
        flw     fa0, 0(t2)      # fa0 ← vec[i]
        fmul.s  ft0, fa0, fa0   # ft0 ← vec[i] * vec[i]
        addi    t3, sp, 0       # t3 ← &vquadrats[0]
        add     t3, t3, t1      # t3 ← &vquadrats[i]
        fsw     ft0, 0(t3)      # vquadrats[i] ← vec[i] * vec[i]
        addi    s1, s1, 1       # i++
        j       bucle
fibucle:

        # q = mitjana(vquadrats)
        mv      a0, sp          # a0 ← &vquadrats[0]
        call    mitjana
        fmv.s   fs1, fa0        # fs1 ← q

        # m = mitjana(vec)
        mv      a0, s0          # a0 ← vec
        call    mitjana
        fmv.s   fs0, fa0        # fs0 ← m

        # return q - m * m
        fmul.s  ft0, fs0, fs0   # ft0 ← m * m
        fsub.s  fa0, fs1, ft0   # fa0 ← q - m * m

        # Epíleg
        flw     fs1, 400(sp)
        flw     fs0, 404(sp)
        lw      s1,  408(sp)
        lw      s0,  412(sp)
        lw      ra,  416(sp)
        addi    sp, sp, 420
        ret
```

Exemple 69: Solució: lectura, suma i comparació de `float` en memòria

Enunciat: Problema 5.19.

C

```
float v[4] = {1.5, -2.0, 3.25, 0.5};
float resultat;
```

---

a) Lectura de `v[0]` i `v[2]`

Cada element `float` ocupa 4 bytes. `v[2]` és a un desplaçament de $2 \times 4 = 8$ bytes respecte de la base.

RV32IF

```
        la      t0, v           # t0 ← @v
        flw     fa0, 0(t0)      # fa0 ← v[0] = 1.5
        flw     fa1, 8(t0)      # fa1 ← v[2] = 3.25
```

b) Suma i escriptura a `resultat`

RV32IF

```
        fadd.s  fa0, fa0, fa1   # fa0 ← v[0] + v[2] = 4.75
        la      t1, resultat
        fsw     fa0, 0(t1)      # resultat ← 4.75
```

c) Comparació `v[0] > v[2]` amb `flt.s`

`flt.s` calcula $rd = (rs1 < rs2)\ ?\ 1 : 0$. Per obtenir $v[0] > v[2]$ cal invertir els operands: $v[0] > v[2] \Leftrightarrow v[2] < v[0]$.
Cal recarregar `fa0` i `fa1` perquè l'apartat b) ha sobreescrit `fa0`:

RV32IF

```
        la      t0, v
        flw     fa0, 0(t0)      # fa0 ← v[0] = 1.5
        flw     fa1, 8(t0)      # fa1 ← v[2] = 3.25
        flt.s   t0, fa1, fa0    # t0 ← (v[2] < v[0]) ? 1 : 0
                                # = (3.25 < 1.5) ? 1 : 0 = 0
```

Per a `v[0] = 1.5` i `v[2] = 3.25`, 1,5 $\not>$ 3,25, per tant, `t0 = 0`.
Atenció: `flt.s rd, rs1, rs2` avalua $rs1 < rs2$, no $rs1 > rs2$. Per implementar `a > b`, cal escriure `flt.s rd, b, a` (invertir l'ordre dels operands), **no** negar el resultat amb `xori`.

Exemple 70: Solució: conversions entre `int` i `float`

Enunciat: Problema 5.20.
a) `int i = (int) f;`
La conversió de tipus `(int)` de C trunca cap a zero: cal indicar-ho explícitament amb el mode `rtz`.

RV32IF

```
        fcvt.w.s a0, fa0, rtz   # a0 ← (int) fa0  [rtz: trunca cap a zero, com el cast de C]
```

b) `float f = (float) i;`

RV32IF

```
        fcvt.s.w fa0, a0        # fa0 ← (float) a0
```

c) `unsigned int u = (unsigned int) f;`

RV32IF

```
        fcvt.wu.s a0, fa0, rtz  # a0 ← (unsigned int) fa0  [rtz: trunca cap a zero, com el cast
        ↪  de C]
```

d) Càlcul manual per a `f = 3.9`
`fcvt.w.s`/`fcvt.wu.s` s'arrodoneixen segons el mode indicat: **no trunquen per si mateixes**. Sense operand de mode, l'assemblador hi assigna `dyn` (el mode vigent al `frm`, RNE per defecte). Per traduir la conversió `(int)` de C —que trunca sempre cap a zero— cal l'operand explícit `rtz`, com a l'apartat a).

3,9 en IEEE 754 de precisió simple no és exactament representable. El valor emmagatzemat és:

$$3{,}9 \approx 1{,}11110011001100110011101_2 \times 2^1 = 3{,}9000000953\ldots$$

Aplicant `fcvt.w.s a0, fa0, rtz` (truncament cap a zero, com la conversió `(int)` de C):

$$\lfloor 3{,}9000\ldots \rfloor = 3$$

Resultat: `a0 = 3`.
Es tracta d'un truncament, no d'un arrodoniment: el valor emmagatzemat ($3{,}9000000953\ldots$) ja és lleugerament superior a 3,9; amb `rtz` la part fraccionària es descarta directament. Si s'hagués escrit `fcvt.w.s a0, fa0` sense mode (`dyn` → RNE per defecte), el resultat seria `a0 = 4` (arrodoniment al més pròxim), no `3`.

5.3 No-associativitat

Exemple 71: Solució: no-associativitat de la suma en coma flotant

Enunciat: Problema 5.21.
a) `A = 0xF3FD`, `B = 0x73FD`, `C = 0x3C00` en format *half* (1 bit signe, 5 bits exponent esbiaixat (biaix 15), 10 bits fracció).
Descodificació

`A = 0xF3FD = 1 11100 1111111101`

```
S=1, Eu=11100b=28 → e = 28−15 = 13
F = 11 1111 1101b = 1021
mantissa = 1 + 1021/1024 = 2045/1024
Valor = −(2045/1024) × 2¹³ = −2045 × 2³ = −16360
```

`B = 0x73FD = 0 11100 1111111101`

```
S=0, Eu=28 → e=13  (mateixa fracció que A, signe oposat)
Valor = +16360
```

`C = 0x3C00 = 0 01111 0000000000`

```
S=0, Eu=01111b=15 → e = 15−15 = 0
F = 00 0000 0000b = 0
Valor = 1,0000000000 × 2⁰ = 1
```

Càlcul de $(A+B)+C$
Pas 1: $A+B$
A i B tenen el mateix exponent i la mateixa fracció però signes oposats: $A = -B$ exactament. La suma és zero sense cap error d'arrodoniment:

$$A + B = 0 \quad (\texttt{0x0000})$$

Pas 2: $0+C$

$$0 + 1 = 1 \quad (\texttt{0x3C00})$$

$$\boxed{(A+B)+C=1}$$

Càlcul de $A+(B+C)$
Pas 1: $B+C$
$e_B = 13$, $e_C = 0$, diferència $= 13$. Cal desplaçar la mantissa de C 13 posicions a la dreta per alinear-la amb B:

`Alineació de C amb B`

```
B:          1,1111111101  × 2¹³   (10 bits de fracció)
C original: 1,0000000000  × 2⁰
C alineat:  0,0000000000001...  × 2¹³
              ← 10 bits→↑ posició −11 (Guard)
```

Els 10 bits de fracció de B cobreixen les posicions 2^{-1} fins a 2^{-10}. El bit ocult de C desplaçat arriba a la posició 2^{-13}, tres posicions més enllà del darrer bit de fracció. Per tant:

- $G = 0$ (posició 2^{-11}: cap bit de C arriba aquí).
- $R = 0$ (posició 2^{-12}: ídem).
- $S = 1$ (posicions $\leq 2^{-13}$: hi ha el bit ocult de C).

Com que $G = 0$, la regla RNE no arrodoneix cap amunt. C es perd completament:

$$B + C = B = +16360 \quad (\texttt{0x73FD})$$

Pas 2: $A+(B+C)$

$$A + 16360 = -16360 + 16360 = 0 \quad (\texttt{0x0000})$$

$$\boxed{A+(B+C)=0}$$

Conclusió

$$(A+B)+C = 1 \neq 0 = A+(B+C)$$

La suma en coma flotant **no és associativa**. La pèrdua de precisió es produeix en calcular $B+C$: els exponents de $C = 1$ i de $B = 16360$ difereixen en 13, més que els 10 bits de fracció del format *half*, i C queda completament absorbit per l'arrodoniment. En canvi, quan es calcula $(A+B)$ primer, la cancel · lació exacta de A i B deixa zero, i llavors C és prou gran per ser representat.
b) `A = 0x4F2A`, `B = 0x36A0`, `C = 0x4A12`.

`Descodificació`

```
A = 0x4F2A = 0 10011 1100101010   e = 19−15 = 4    mantissa 1,1100101010   valor 28,65625
B = 0x36A0 = 0 01101 1010100000   e = 13−15 = −2   mantissa 1,1010100000   valor 0,4140625
C = 0x4A12 = 0 10010 1000010010   e = 18−15 = 3    mantissa 1,1000010010   valor 12,140625
```

Càlcul de $(A+B)+C$
Pas 1: $A+B$. Els exponents difereixen en 6: la mantissa de B es desplaça 6 posicions a la dreta. El bit que cau a la posició 2^{-11} és un 1 (G), i tots els de després, zeros ($R = S = 0$):

`A + B`

```
A:          1,1100101010           × 2⁴
B alineat:  0,0000011010 | 1 0 0   × 2⁴   (G R S)
Suma:       1,1101000100 | 1 0 0   × 2⁴
```

$GRS = 100$ és un empat: RNE arrodoneix al parell, i el bit de menys pes ja és 0. El resultat, $1{,}1101000100_2 times 2^4 = 29{,}0625$ (`0x4F44`), ha perdut el 2^{-7} de B (la suma exacta és 29,0703125).
Pas 2: $(A + B) + C$. Diferència d'exponents 1:

```
(A + B) + C
```

```
A+B:         1,1101000100           × 2⁴
C alineat:   0,1100001001 | 0 0 0   × 2⁴
Suma:       10,1001001101 | 0 0 0   × 2⁴
Normalitzat: 1,0100100110 | 1 0 0   × 2⁵
```

En normalitzar, el bit de menys pes surt cap a G: torna a ser un empat, i el bit de menys pes és 0.

$$\boxed{(A + B) + C = 1{,}0100100110_2 \times 2^5 = 41{,}1875 \quad (\texttt{0x5126})}$$

Càlcul de $A + (B + C)$
Pas 1: $B + C$. Diferència d'exponents 5: els cinc primers bits de la fracció de B queden dins dels 10 de C, i la resta són zeros. La suma és exacta:

```
B + C
```

```
C:          1,1000010010           × 2³
B alineat:  0,0000110101 | 0 0 0   × 2³
Suma:       1,1001000111 | 0 0 0   × 2³   = 12,5546875 (0x4A47)
```

Pas 2: $A + (B + C)$. Diferència d'exponents 1:

```
A + (B + C)
```

```
A:           1,1100101010           × 2⁴
B+C alineat: 0,1100100011 | 1 0 0   × 2⁴
Suma:       10,1001001101 | 1 0 0   × 2⁴
Normalitzat: 1,0100100110 | 1 1 0   × 2⁵
```

En normalitzar, el bit de menys pes passa a G i l'antic G, a R: $GRS = 110$, més de la meitat, i RNE arrodoneix cap amunt.

$$\boxed{A + (B + C) = 1{,}0100100111_2 \times 2^5 = 41{,}21875 \quad (\texttt{0x5127})}$$

Conclusió

$$(A + B) + C = 41{,}1875 \neq 41{,}21875 = A + (B + C)$$

Els dos resultats difereixen en una unitat de l'última posició ($2^{5-10} = 0{,}03125$). La suma exacta és 41,2109375: $A + (B + C)$ n'és més a prop, perquè $B + C$ és exacta, mentre que $A + B$ perd els bits de menys pes de B en un empat arrodonit cap avall.

Exemple 72: Solució: no-distributivitat de la multiplicació sobre la suma

Enunciat: Problema 5.22.
Els operands són els mateixos que les conversions de Problema 5.8 (apartats b i c). Cada operació s'arrodoneix amb RNE abans de la següent.
a) `A = 0x30E0, B = 0x32A0, C = 0x563B`
Descodificació

```
A = 0x30E0 = 0 01100 0011100000
```

```
S=0, Eu=01100b=12 → e = 12−15 = −3
Mantissa = 1,0011100000  (= 1248/1024)
Valor = 1,00111b × 2⁻³ = 0,00100111b = 0,15234375
```

```
B = 0x32A0 = 0 01100 1010100000
```

```
S=0, Eu=12 → e = −3
Mantissa = 1,1010100000  (= 1696/1024)
Valor = 1,10101b × 2⁻³ = 0,00110101b = 0,20703125
```

```
C = 0x563B = 0 10101 1000111011
```

```
S=0, Eu=10101b=21 → e = 21−15 = 6
Mantissa = 1,1000111011  (= 1595/1024)
Valor = 1,1000111011b × 2⁶ = 110 0011,1011b = 99,6875
```

Operació 1: $B + C$ (per al camí de l'esquerra)
Diferència d'exponents: $6 - (-3) = 9$; la mantissa de B es desplaça 9 posicions a la dreta:

```
Alineació (Δe = 9)
```

```
C:          1,1000111011                 × 2⁶
B original: 1,1010100000                 × 2⁻³
B alineat:  0,0000000011 | 0 1 | 01…     × 2⁶
            ← 10 bits →    G R    S
```

```
Suma
```

```
  1,1000111011 | 000
+ 0,0000000011 | 011
= 1,1000111110 | 011        (sense ròssec)
```

$G = 0 \rightarrow$ es **trunca**:

$$B + C = 0 \mid 10101 \mid 1000111110 = \texttt{0x563E} = 110\,0011{,}1110_2 = 99{,}875$$

(La suma exacta és 99,89453125: error $0{,}01953125 < \frac{1}{2}\,\text{ULP} = 2^{-5}$.)
Operació 2: $A \times (B + C)$
Exponent: $e = -3 + 6 = 3$; signe positiu. Producte de mantisses:

```
Producte de mantisses
```

```
1248 × 1598 = 1 994 304 = 1 1110 0110 1110 0100 0000b   (21 bits)
Producte = 1,1110 0110 1110 0100 0000                   (< 2: no cal renormalitzar)
```

$G = 1$, $R = 0$, $S = 1 \rightarrow$ **al següent**: $F = 1110011011 + 1 = 1110011100$:

$$A \times (B + C) = 0 \mid 10010 \mid 1110011100 = \texttt{0x4B9C} = 1111{,}00111_2 = \mathbf{15{,}21875}$$

Operació 3: $A \times B$ (per al camí de la dreta)
Exponent: $e = -3 + (-3) = -6$; signe positiu. Producte de mantisses:

```
Producte de mantisses
```

```
1248 × 1696 = 2 116 608 = 10 0000 0100 1100 0000 0000b  (22 bits)
Producte = 10,0000 0100 1100 0000 0000                  (≥ 2: renormalitzar)
→ 1,0000 0100 1100 0000 0000 0  × 2⁻⁵
```

Els bits descartats són $G = 1$, $R = 0$ i $S = 0$: el residu val **exactament** ½ ULP $\rightarrow$ **empat**. La regla del parell decideix: la fracció truncada `0000001001` acaba en `1` (senar), així que s'arrodoneix **amunt** perquè el resultat acabi en bit parell:

$$F = 0000001001 + 1 = 0000001010$$

$$A \times B = 0 \mid 01010 \mid 0000001010 = \texttt{0x280A} = 0{,}03155517578125$$

(És l'únic problema resolt de T5 on el desempat «al parell» entra en joc: l'error comès és exactament ½ ULP $= 2^{-16}$.)

Operació 4: $A \times C$

Exponent: $e = -3 + 6 = 3$; signe positiu. Producte de mantisses:

`Producte de mantisses`

```
1248 × 1595 = 1 990 560 = 1 1110 0101 1111 1010 0000b   (21 bits)
Producte = 1,1110 0101 1111 1010 0000                   (< 2: no cal renormalitzar)
```

$G = 1$, $R = 1$ $\rightarrow$ **al següent**: $F = 1110010111 + 1 = 1110011000$:

$$A \times C = 0 \mid 10010 \mid 1110011000 = \texttt{0x4B98} = 1111{,}0011_2 = 15{,}1875$$

Operació 5: $(A \times B) + (A \times C)$

Diferència d'exponents: $3 - (-5) = 8$; la mantissa de $A \times B$ es desplaça 8 posicions:

`Alineació (Δe = 8)`

```
A×C:          1,1110011000                  × 2³
A×B original: 1,0000001010                  × 2⁻⁵
A×B alineat:  0,0000000100 | 0 0 | 001010   × 2³
              ← 10 bits →    G R    S
```

`Suma`

```
  1,1110011000 | 000
+ 0,0000000100 | 001
= 1,1110011100 | 001        (sense transport)
```

$G = 0$ $\rightarrow$ es **trunca**:

$$(A \times B) + (A \times C) = 0 \mid 10010 \mid 1110011100 = \texttt{0x4B9C} = \mathbf{15{,}21875}$$

Comparació de l'apartat a)

$$\boxed{A \times (B + C) = \texttt{0x4B9C} = 15{,}21875 = (A \times B) + (A \times C)}$$

En aquest cas els dos camins **coincideixen**, tot i haver acumulat cinc arrodoniments (dos truncaments, dos «al següent» i un empat «al parell»). Els resultats exactes de l'última operació de cada camí, calculats amb els intermedis ja arrodonits, són diferents (15,21533... i 15,21905...), i el valor exacte d'$A \times (B + C)$ és 15,2183...: tots dos s'arrodoneixen casualment al mateix representable. La igualtat és **fortuïta**, no una propietat.

b) `A = 0xCEF9, B = 0xEFE6, C = 0x70FD`

Descodificació

```
A = 0xCEF9 = 1 10011 1011111001

S=1, Eu=10011b=19 → e = 19−15 = 4
Mantissa = 1,1011111001  (= 1785/1024)
Valor = −1,1011111001b × 2⁴ = −1 1011,111001b = −27,890625
```

```
B = 0xEFE6 = 1 11011 1111100110

S=1, Eu=11011b=27 → e = 27−15 = 12
Mantissa = 1,1111100110  (= 2022/1024)
Valor = −(2022/1024) × 2¹² = −2022 × 2² = −8088
```

```
C = 0x70FD = 0 11100 0011111101

S=0, Eu=11100b=28 → e = 28−15 = 13
Mantissa = 1,0011111101  (= 1277/1024)
Valor = (1277/1024) × 2¹³ = 1277 × 2³ = 10 216
```

Operació 1: $B + C$

Els signes són **diferents**: es resten les magnituds i el resultat pren el signe de l'operand més gran (C, positiu). Diferència d'exponents: $13 - 12 = 1$:

```
Alineació i resta (Δe = 1)

C:          1,0011111101 0        × 2¹³
B alineat:  0,1111110011 0        × 2¹³     (el desplaçament d'1 bit no en perd cap)
Resta:      0,0100001010 0                  (exacta: G=R=S=0)
```

El resultat es renormalitza desplaçant 2 posicions a l'**esquerra** ($e = 13 - 2 = 11$):

$$B + C = 0 \mid 11010 \mid 0000101000 = \texttt{0x6828} = 1000\,0101\,0000_2 = 2128 \quad \text{(exacte)}$$

Operació 2: $A \times (B + C)$

Exponent: $e = 4 + 11 = 15$, **just el màxim del format**: si el producte de mantisses arribés a 2, la renormalització provocaria desbordament. Signe: $(-) \cdot (+) = -$.

```
Producte de mantisses

1785 × 1064 = 1 899 240 = 1 1100 1111 1010 1110 1000b   (21 bits)
Producte = 1,1100 1111 1010 1110 1000                   (< 2: e = 15, sense desbordament)
```

$G = 1$, $R = 0$, $S = 1 \rightarrow$ **al següent**: $F = 1100111110 + 1 = 1100111111$:

$$A \times (B + C) = 1 \mid 11110 \mid 1100111111 = \texttt{0xFB3F} = -1855 \times 2^5 = \mathbf{-59\,360}$$

(El producte exacte és $-59\,351{,}25$: error $8{,}75 < \frac{1}{2}\,\text{ULP} = 2^4 = 16$.)

Operacions 3 i 4: $A \times B$ **i** $A \times C$ **— desbordament**

Per a $A \times B$, la suma d'exponents ja és $e = 4 + 12 = 16 > e_{max} = 15$. Com que el producte de dues mantisses normalitzades és sempre ≥ 1, la normalització **mai no redueix** l'exponent per sota d'aquesta suma: el resultat supera amb seguretat el màxim representable del format *half*, $(2 - 2^{-10}) \times 2^{15} = 65\,504$, sense necessitat de multiplicar les mantisses. **Desbordament**: amb RNE el resultat es converteix en infinit, amb el signe del producte:

$$A \times B \to (-)(-) = +\infty = \texttt{0x7C00} \qquad (|A \times B| \approx 2{,}3 \times 10^5)$$

$$A \times C:\ e = 4 + 13 = 17 > 15 \to (-)(+) = -\infty = \texttt{0xFC00} \qquad (|A \times C| \approx 2{,}8 \times 10^5)$$

Operació 5: $(A \times B) + (A \times C)$

$$(+\infty) + (-\infty) = \infty - \infty \to \textbf{NaN}$$

L'operació no té cap valor definit (Secció 5.1.6.4): el resultat és un NaN silenciós, amb $E_u = 31$ i el bit de més pes de la fracció a 1 (per exemple, `0x7E00`; l'estàndard no en fixa la resta de la fracció).

Comparació de l'apartat b)

$$\boxed{A \times (B + C) = -59\,360\ (\texttt{0xFB3F}) \neq \text{NaN} = (A \times B) + (A \times C)}$$

Conclusió

La multiplicació **no és distributiva** sobre la suma en coma flotant. L'apartat a) mostra que la igualtat es pot donar fortuïtament; l'apartat b) mostra la fallada extrema: **factoritzar** l'expressió (calcular primer $B + C = 2128$) manté tots els resultats intermedis representables, mentre que **expandir-la** fa desbordar els dos productes parcials cap a $\pm\infty$ i el resultat final és NaN. Com en la no-associativitat (Essencial 5.6), l'ordre de les operacions altera el resultat: reordenar algebraicament un càlcul en coma flotant no és mai «gratuït».

Tema 6: Rendiment i potència

6.1 Rendiment

Exemple 73: Solució: comparació de processadors

Enunciat: Problema 6.1.
El **temps d'execució** d'un programa és $t_{exe} = N \cdot \text{CPI} \cdot t_c = N \cdot \text{CPI}/f$ (Equació 6.3):

$$t_{P1} = \frac{2 \cdot 10^6 \cdot 1{,}5}{2 \cdot 10^9} = 1{,}5 \text{ ms} \qquad t_{P2} = \frac{2 \cdot 10^6 \cdot 1}{2{,}5 \cdot 10^9} = 0{,}8 \text{ ms} \qquad t_{P3} = \frac{10^6 \cdot 0{,}75}{3 \cdot 10^9} = 0{,}25 \text{ ms}$$

a) El processador més ràpid és P3. Les diferències:

$$\frac{t_{P1}}{t_{P3}} = \frac{1{,}5}{0{,}25} = 6\times \qquad \frac{t_{P2}}{t_{P3}} = \frac{0{,}8}{0{,}25} = 3{,}2\times$$

P3 és 6 vegades més ràpid que P1 i 3,2 vegades més ràpid que P2.
b) P1 i P2 executen el **mateix programa amb el mateix ISA** i, per tant, el mateix nombre d'instruccions (N). El guany (*speedup*) $t_{P1}/t_{P2} = \text{CPI}_{P1} \cdot f_{P2}/(\text{CPI}_{P2} \cdot f_{P1})$ no depèn de N: P1 i P2 es poden comparar directament.
En canvi, P1 i P3 usen **ISA diferent** (RISC-V vs. x86-64): el mateix programa font compila a un nombre d'instruccions diferent. Sense conèixer N, P1 i P3 no es poden comparar.
c) No, en cap dels dos casos: el CPI és una propietat del **parell processador–programa**. Per comparar dos processadors en general caldria el CPI mitjà sobre **tots** els programes. Si el CPI donat fos la mitjana sobre tots els programes:

- P1 i P2 usen el mateix ISA: si també fan servir el mateix compilador (l'enunciat no ho diu), el nombre d'instruccions és el mateix per a qualsevol programa, i la comparació és vàlida.
- P1 i P3 usen ISA diferent: el nombre d'instruccions varia per a cada programa, i la comparació general continua sense ser vàlida.

Exemple 74: Solució: dos processadors, temps i instruccions

Enunciat: Problema 6.2.

$$t_{P1} = \frac{10^6 \cdot 1{,}25}{4 \cdot 10^9} = 312{,}5 \ \mu\text{s} \qquad t_{P2} = \frac{10^6 \cdot 0{,}75}{3 \cdot 10^9} = 250 \ \mu\text{s}$$

a) P2 és més ràpid. El guany de P2 respecte de P1:

$$s = \frac{t_{P1}}{t_{P2}} = \frac{312{,}5}{250} = 1{,}25\times$$

b) En el temps $t_{P1} = 312{,}5$ μs, P2 pot executar:

$$N_{P2} = t_{P1} \cdot \frac{f_{P2}}{\text{CPI}_{P2}} = 312{,}5 \cdot 10^{-6} \cdot \frac{3 \cdot 10^9}{0{,}75} = 1{,}25 \cdot 10^6 \text{ instruccions}$$

Exemple 75: Solució: unitat de coma flotant per maquinari

Enunciat: Problema 6.3.
Es pren N_{orig} com a referència: el nombre d'instruccions **sense** unitat de coma flotant. El 20 % correspon a instruccions de les subrutines de coma flotant (CF); el 80 % restant, a instruccions d'altres tipus. Totes les instruccions amb CPI = 3 i $t_c = 1$ ns:

$$t_{orig} = N_{orig} \cdot 3 \cdot 1 \text{ ns} = 3\,N_{orig} \text{ ns}$$

Amb la unitat de coma flotant, el nombre d'instruccions de coma flotant executades es redueix 10 vegades (enunciat). Les instruccions no-CF no canvien:

$$N_{nou} = 0{,}80\,N_{orig} + \frac{0{,}20\,N_{orig}}{10} = 0{,}82\,N_{orig}$$

Amb $t_c = 1{,}05$ ns i CPI = 4 per a totes les instruccions:

$$t_{nou} = 0{,}82\,N_{orig} \cdot 4 \cdot 1{,}05 \text{ ns} = 3{,}444\,N_{orig} \text{ ns}$$

a)

$$s = \frac{t_{orig}}{t_{nou}} = \frac{3}{3{,}444} \approx \mathbf{0{,}87}$$

El processador amb unitat de coma flotant té un rendiment un 13 % **inferior** (triga un 14,8 % més: $3{,}444/3 \approx 1{,}148$): no val la pena. Tot i que el nombre d'instruccions es redueix ($\times 0{,}82$), la reducció no compensa l'augment del CPI (de 3 a 4) ni l'allargament del cicle ($\times 1{,}05$).
b) Amb CF = 40 %:

$$N_{nou} = 0{,}60\,N_{orig} + \frac{0{,}40\,N_{orig}}{10} = 0{,}64\,N_{orig}$$

$$t_{nou} = 0{,}64\,N_{orig} \cdot 4 \cdot 1{,}05 \text{ ns} = 2{,}688\,N_{orig} \text{ ns}$$

$$s = \frac{3}{2{,}688} \approx \mathbf{1{,}12}$$

Amb el 40 % de CF **sí que val la pena** afegir la unitat de coma flotant (millora del 12 %). La reducció d'instruccions és prou gran per compensar els efectes negatius sobre el CPI i el temps de cicle.

Exemple 76: Solució: distribució per tipus d'instruccions

Enunciat: Problema 6.4.
a) El nombre de cicles totals és la suma de cicles per tipus:

Tipus	Instruccions	CPI	Cicles
Aritmètiques	500	1	500
Escriptura	50	5	250
Lectura	100	5	500
Salt	50	2	100
Total	**700**		**1 350**

$$t_{exe} = 1350 \text{ cicles} \cdot 1 \text{ ns/cicle} = \mathbf{1{,}35}\ \mu\text{s}$$

b)

$$\text{CPI} = \frac{1350 \text{ cicles}}{700 \text{ instruccions}} \approx 1{,}93$$

c) Amb 50 instruccions de lectura (en comptes de 100):

$$\text{Cicles}_{opt} = 500 + 250 + 50 \cdot 5 + 100 = 1100$$

$$s = \frac{1350}{1100} \approx \mathbf{1{,}23}\times \qquad \text{CPI}_{opt} = \frac{1100}{650} \approx 1{,}69$$

Exemple 77: Solució: quatre classes d'instruccions

Enunciat: Problema 6.5.
a) El CPI mitjà es calcula com la mitjana ponderada per la freqüència de cada classe:

$$\text{CPI}_{P1} = 0{,}10 \cdot 1 + 0{,}20 \cdot 2 + 0{,}50 \cdot 3 + 0{,}20 \cdot 4 = 0{,}1 + 0{,}4 + 1{,}5 + 0{,}8 = \mathbf{2{,}8}$$

$$\text{CPI}_{P2} = 0{,}10 \cdot 2 + 0{,}20 \cdot 2 + 0{,}50 \cdot 2 + 0{,}20 \cdot 2 = \mathbf{2{,}0}$$

b) Per a $N = 10^6$ instruccions:

$$t_{P1} = \frac{10^6 \cdot 2{,}8}{1{,}5 \cdot 10^9} \approx 1{,}867 \text{ ms} \qquad t_{P2} = \frac{10^6 \cdot 2{,}0}{2{,}0 \cdot 10^9} = 1{,}000 \text{ ms}$$

P2 és més ràpid. El guany:

$$s = \frac{t_{P1}}{t_{P2}} = \frac{N \cdot \text{CPI}_{P1}/f_{P1}}{N \cdot \text{CPI}_{P2}/f_{P2}} = \frac{\text{CPI}_{P1} \cdot f_{P2}}{\text{CPI}_{P2} \cdot f_{P1}} = \frac{2{,}8 \cdot 2{,}0}{2{,}0 \cdot 1{,}5} = \frac{28}{15} \approx \mathbf{1{,}87}\times$$

Observació: P2 té un CPI uniforme de 2 per a totes les classes. Això reflecteix que un disseny amb CPI uniforme pot ser competitiu si la freqüència de rellotge és prou alta.

Exemple 78: Solució: multiplicador per maquinari

Enunciat: Problema 6.6.
Sigui N el nombre d'instruccions amb el multiplicador per maquinari. El 0,5 % de N són multiplicacions (1 instrucció, CPI = 40); el 99,5 % restant té CPI = 3.
Sense multiplicador: cada multiplicació s'expandeix a una subrutina de 200 instruccions. Les $0{,}005N$ multiplicacions passen a $0{,}005N \cdot 200 = N$ instruccions de subrutina, totes amb CPI = 3:

$$N_{orig} = 0{,}995N + 0{,}005N \cdot 200 = 1{,}995N$$

a) Comparació a freqüència idèntica ($f = 2$ GHz):

$$t_{orig} = 1{,}995N \cdot 3 \cdot t_c = 5{,}985N \cdot t_c$$

$$t_{nou} = N \cdot (0{,}005 \cdot 40 + 0{,}995 \cdot 3) \cdot t_c = N \cdot 3{,}185 \cdot t_c$$

$$s = \frac{5{,}985}{3{,}185} = \frac{1197}{637} \approx \mathbf{1{,}88}\times$$

b) Per no empitjorar el rendiment cal $t_{nou}(f_{nou}) \leq t_{orig}(2\text{ GHz})$:

$$\frac{N \cdot 3{,}185}{f_{nou}} \leq \frac{1{,}995N \cdot 3}{2 \cdot 10^9}$$

$$f_{nou} \geq \frac{2 \cdot 10^9 \cdot 3{,}185}{5{,}985} = \frac{1274}{1197} \cdot 10^9 \approx \mathbf{1{,}064}\text{ GHz}$$

La freqüència mínima acceptable és aproximadament 1,06 GHz, força per sota dels 2 GHz originals. El multiplicador per maquinari és clarament beneficiós fins i tot amb una degradació significativa de la freqüència de rellotge.

Exemple 79: Solució: millores per component

Enunciat: Problema 6.7.
Cas a ($t_{orig} = 200$ s: CF = 35 s, Enters = 85 s, lectura/escriptura (L/E) = 50 s, Salt = 30 s)
a) Es redueix el temps de CF un 20 % (35 s → 28 s):

$$t_{nou} = 200 - 35 + 28 = 193\text{ s} \qquad s = \frac{200}{193} \approx 1{,}036\times$$

b) Objectiu: $s = 1{,}20$, és a dir, $t_{nou} = 200/1{,}20 \approx 166{,}7$ s. El temps restant sense enters és $200 - 85 = 115$ s. Cal que el nou temps d'enters sigui:

$$t_{Enters,nou} = 166{,}7 - 115 = 51{,}7\text{ s} \quad \Rightarrow \quad \text{reducció} = \frac{85 - 51{,}7}{85} \approx \mathbf{39{,}2}\ \%$$

c) El guany màxim possible millorant únicament els salts (fins a reduir-ne el temps a 0) és:

$$s_{max} = \frac{200}{200 - 30} = \frac{200}{170} \approx 1{,}18\times$$

Com que $1{,}18 < 1{,}25$ (reduir un 20 % equival a $s = 1{,}25$), **no és possible**.

Cas b ($t_{orig} = 210$ s: CF = 50 s, Enters = 80 s, L/E = 50 s, Salt = 30 s)
a) Es redueix el temps de CF un 20 % (50 s → 40 s):

$$t_{nou} = 210 - 50 + 40 = 200\text{ s} \qquad s = \frac{210}{200} = 1{,}05\times$$

b) $t_{nou} = 210/1{,}20 = 175$ s; temps sense enters $= 210 - 80 = 130$ s:

$$t_{Enters,nou} = 175 - 130 = 45\text{ s} \quad \Rightarrow \quad \text{reducció} = \frac{80 - 45}{80} = \mathbf{43{,}75}\ \%$$

c) s_{max} per als salts:

$$s_{max} = \frac{210}{210 - 30} = \frac{210}{180} \approx 1{,}17\times < 1{,}25$$

Tampoc és possible en el cas b.

6.2 Llei d'Amdahl

Exemple 80: Solució: Amdahl bàsic

Enunciat: Problema 6.8.
La **llei d'Amdahl** (Equació 6.6) dona el guany global quan s'accelera una fracció P_x del temps d'execució amb un guany s_x:

$$s = \frac{1}{(1 - P_x) + P_x/s_x}$$

Aquí $P_x = 0{,}60$ (CF), fracció no millorada $= 0{,}40$.
a) Amb $s_x = 5$:

$$s = \frac{1}{0{,}40 + 0{,}60/5} = \frac{1}{0{,}40 + 0{,}12} = \frac{1}{0{,}52} = \frac{25}{13} \approx \mathbf{1{,}92}\times$$

b) Quan $s_x \to \infty$, el terme $P_x/s_x \to 0$ (Equació 6.7):

$$s_{max} = \frac{1}{1 - P_x} = \frac{1}{0{,}40} = \mathbf{2{,}5}\times$$

Independentment de com s'accelerin les instruccions CF, el guany global no pot superar 2,5, perquè el 40 % restant és un **coll d'ampolla** irreductible.
c) Amb $s = 3$:

$$3 = \frac{1}{0{,}40 + 0{,}60/s_x} \implies 0{,}40 + \frac{0{,}60}{s_x} = \frac{1}{3} \implies \frac{0{,}60}{s_x} = \frac{1}{3} - 0{,}40 = -\frac{1}{15}$$

El resultat és negatiu, cosa que és impossible. La raó és que el guany màxim teòric és 2,5, estrictament inferior a 3: **no existeix cap valor de s_x que permeti assolir un guany global de 3** amb aquesta distribució del temps.

Exemple 81: Solució: tres components simultanis

Enunciat: Problema 6.9.
Quan s'apliquen múltiples millores simultànies, la llei d'Amdahl es generalitza:

$$s = \frac{1}{\left(1 - \sum_x P_x\right) + \sum_x \frac{P_x}{s_x}}$$

on P_x és la fracció del temps original que ocupa el component x i s_x el guany d'aquest component. En aquest problema, els tres components cobreixen el 100 % del temps ($\sum_x P_x = 1$), de manera que el primer terme s'anul · la i l'expressió es redueix a $s = 1/\sum_x (P_x/s_x)$.
a) Aplicant les tres millores:

$$s = \frac{1}{\frac{0{,}50}{2} + \frac{0{,}30}{4} + \frac{0{,}20}{1{,}5}} = \frac{1}{0{,}250 + 0{,}075 + 0{,}1\overline{3}} = \frac{1}{11/24} = \frac{24}{11} \approx \mathbf{2{,}18}\times$$

b i c) Guany aplicant cada millora individualment:

Component	P_x	s_x	s (millora sola)
Unitat entera	50 %	2×	$1/(0{,}50 + 0{,}50/2) = 1{,}33\times$

Unitat CF	30 %	4×	$1/(0{,}70 + 0{,}30/4) = 1{,}29\times$
Accés a memòria	20 %	1,5×	$1/(0{,}80 + 0{,}20/1{,}5) = 1{,}07\times$

La millora individual amb **més impacte** és la de la **unitat entera** ($s = 1{,}33\times$), tot i que la seva millora disponible ($2\times$) no és la més gran (la de CF és de $4\times$). La llei d'Amdahl explica per què: el que determina l'impacte és principalment la **fracció de temps** ($P_x = 50$ %), no el factor de millora.
Si el pressupost permet una sola millora, s'ha d'escollir la **unitat entera**: maximitza el guany global perquè actua sobre el component dominant en temps.

Exemple 82: Solució: Amdahl invers i límit teòric

Enunciat: Problema 6.10.
a) Es vol $s = 10$ amb $s_x = 20$. De la llei d'Amdahl:

$$10 = \frac{1}{(1 - P_x) + P_x/20} \implies \frac{1}{10} = 1 - P_x + \frac{P_x}{20} = 1 - P_x\left(1 - \frac{1}{20}\right) = 1 - \frac{19P_x}{20}$$

$$\frac{19P_x}{20} = 1 - \frac{1}{10} = \frac{9}{10} \implies P_x = \frac{9}{10} \cdot \frac{20}{19} = \frac{18}{19} \approx \mathbf{94{,}7}\ \%$$

Per anunciar un guany de 10 basant-se exclusivament en les instruccions CF, caldria que el 94,7 % del temps d'execució fos de coma flotant — una condició molt restrictiva que pocs programes reals compleixen.
b) En un programa típic amb $P_x = 0{,}35$:

$$s = \frac{1}{0{,}65 + 0{,}35/20} = \frac{1}{0{,}65 + 0{,}0175} = \frac{1}{0{,}6675} \approx \mathbf{1{,}50}\times$$

Molt lluny del $10\times$ anunciat.
c) El guany màxim assolible amb $P_x = 0{,}35$, independentment de s_x:

$$s_{max} = \frac{1}{1 - 0{,}35} = \frac{1}{0{,}65} \approx \mathbf{1{,}54}\times$$

Amb instruccions CF infinitament ràpides, el guany tendiria a $1{,}54\times$ sense poder superar-lo. Això posa de manifest el missatge fonamental de la llei d'Amdahl: el guany màxim queda sempre limitat per la fracció de temps no millorada, per gran que sigui la millora aplicada a la part optimitzada.

Exemple 83: Solució: CPI per components i guany

Enunciat: Problema 6.11.
a) El nombre de cicles totals és la suma per tipus d'instruccions:

	CF	Enters	L/E	Salt	Total
Proc. A	$90 \cdot 2 = 180$	$240 \cdot 1{,}5 = 360$	$150 \cdot 6 = 900$	$120 \cdot 3 = 360$	**1800**
Proc. B	$45 \cdot 3 = 135$	$225 \cdot 1 = 225$	$135 \cdot 5 = 675$	$45 \cdot 2 = 90$	**1125**

$$\text{CPI}_A = \frac{1800}{600} = \mathbf{3{,}0} \qquad \text{CPI}_B = \frac{1125}{450} = \mathbf{2{,}5}$$

b) Perquè tots dos processin el programa en el mateix temps:

$$\frac{N_A \cdot \text{CPI}_A}{f_A} = \frac{N_B \cdot \text{CPI}_B}{f_B} \implies \frac{f_A}{f_B} = \frac{N_A \cdot \text{CPI}_A}{N_B \cdot \text{CPI}_B} = \frac{\text{cicles}_A}{\text{cicles}_B} = \frac{1800}{1125} = \mathbf{1{,}6}$$

El processador A ha de funcionar a una freqüència 1,6 vegades superior a la de B per igualar el seu temps d'execució.
c) Si les instruccions L/E del processador A s'executessin en 0 cicles, els cicles totals serien $1800 - 900 = 900$:

$$s = \frac{1800}{900} = \mathbf{2{,}0}\times$$

Equivalent a aplicar la llei d'Amdahl amb $P_{L/E} = 900/1800 = 50\ \%$ del temps i $s_{L/E} \to \infty$: $s_{max} = 1/(1 - 0{,}5) = 2\times$.

6.3 Potència

Exemple 84: Solució: capacitat equivalent

Enunciat: Problema 6.12.
La **potència dinàmica** en circuits *complementary metal–oxide–semiconductor* (CMOS) és $P_d = \alpha \cdot C \cdot V_{CC}^2 \cdot f$ (Equació 6.9, amb un factor d'activitat $\alpha = 1$).
a) S'aïlla la capacitat equivalent (capacitància) $C = P_d/(V_{CC}^2 \cdot f)$:

$$C_A = \frac{2\ \text{W}}{(5\ \text{V})^2 \cdot 10 \cdot 10^6\ \text{Hz}} = \frac{2}{250 \cdot 10^6} = \mathbf{8}\ \text{nF}$$

$$C_B = \frac{100\ \text{W}}{(1\ \text{V})^2 \cdot 3 \cdot 10^9\ \text{Hz}} = \frac{100}{3 \cdot 10^9} \approx \mathbf{33{,}3}\ \text{nF}$$

b) Amb la mateixa tensió i capacitància de A, però amb $f = f_B = 3$ GHz:

$$P'_{d,A} = C_A \cdot V_{CC,A}^2 \cdot f_B = 8 \cdot 10^{-9} \cdot 25 \cdot 3 \cdot 10^9 = \mathbf{600}\ \text{W}$$

Aquesta potència és absolutament inassolible a la pràctica, i il · lustra per què no n'hi ha prou d'augmentar la freqüència sense reduir el voltatge: la potència creix linealment amb f.

Exemple 85: Solució: evolució de vuit generacions Intel

Enunciat: Problema 6.13.
a i e) Ràtios entre generacions consecutives i capacitats equivalents ($C = P_d/(V_{CC}^2 \cdot f)$):

Gen	Processador	f	V_{CC}	P_d	r_f	r_P	C (nF)
1	80286	12,5 MHz	5,0 V	3,3 W	—	—	10,56
2	80386	16 MHz	5,0 V	4,1 W	1,28	1,24	10,25
3	80486	25 MHz	5,0 V	4,9 W	1,56	1,20	7,84
4	Pentium	66 MHz	5,0 V	10,1 W	2,64	2,06	6,12
5	Pentium Pro	200 MHz	3,3 V	29,1 W	3,03	2,88	13,36

6	P4 Willamette	2 GHz	1,75 V	75,3 W	10,00	2,59	12,29
7	P4 Prescott	3,6 GHz	1,25 V	103 W	1,80	1,37	18,31
8	Core 2 Kentsfield	2,667 GHz	1,1 V	95 W	0,74	0,92	29,44

Exemple de càlcul per a la generació 5 (Pentium Pro):

$$C_5 = \frac{29{,}1\ \text{W}}{(3{,}3\ \text{V})^2 \cdot 200 \cdot 10^6\ \text{Hz}} = \frac{29{,}1}{10{,}89 \cdot 2 \cdot 10^8} \approx 13{,}36\ \text{nF}$$

b) La ràtio màxima de freqüència es produeix entre les generacions 5 i 6 (de 200 MHz a 2 GHz): $r_f = 10$. La ràtio màxima de potència es produeix entre les generacions 4 i 5: $r_P \approx 2{,}88$.
c) La **mitjana geomètrica** de les 7 ràtios entre generacions consecutives:

$$\bar{r}_f = \left(\prod_{i=1}^{7} r_{f,i}\right)^{1/7} = (1{,}28 \cdot 1{,}56 \cdot 2{,}64 \cdot 3{,}03 \cdot 10 \cdot 1{,}80 \cdot 0{,}74)^{1/7} \approx \mathbf{2{,}15}\times$$

$$\bar{r}_P = \left(\prod_{i=1}^{7} r_{P,i}\right)^{1/7} = (1{,}24 \cdot 1{,}20 \cdot 2{,}06 \cdot 2{,}88 \cdot 2{,}59 \cdot 1{,}37 \cdot 0{,}92)^{1/7} \approx \mathbf{1{,}62}\times$$

El producte de les ràtios és el quocient entre la darrera generació i la primera, de manera que $\bar{r}_f = (f_8/f_1)^{1/7}$ i $\bar{r}_P = (P_8/P_1)^{1/7}$: és el que relaciona aquest apartat amb el d).
d) Respecte de la generació 1:

$$\frac{f_8}{f_1} = \frac{2{,}667 \cdot 10^9}{12{,}5 \cdot 10^6} \approx \mathbf{213}\times \qquad \frac{P_8}{P_1} = \frac{95}{3{,}3} \approx \mathbf{28{,}8}\times$$

Observació: a partir de la generació 7 (Pentium 4 Prescott, 2004), la freqüència deixa de créixer —a la 8 fins i tot baixa— i la potència es redueix lleugerament, com situa A6 cap al 2003–2005 (Aprofundiment 6.2). Això marca el final de l'era del creixement de la freqüència com a via principal per millorar el rendiment.

Exemple 86: Solució: potència estàtica i corrents de fuita

Enunciat: Problema 6.14.
a) Percentatge de cada component sobre la potència total:

Tecnologia	P_d	P_s	P	% dinàmica	% estàtica
250 nm	49 W	1 W	50 W	**98 %**	**2 %**
90 nm	75 W	45 W	120 W	**62,5 %**	**37,5 %**

Els corrents de fuita passen de ser negligibles (2 %) a representar el 37,5 % de la potència total en passar de 250 nm a 90 nm. Aquesta és una de les raons principals per les quals la miniaturització porta associada una importància creixent del consum estàtic.

b) De $P_s = V_{CC} \cdot I_{fuita}$ (Equació 6.11):

$$I_{fuita,250} = \frac{1\text{ W}}{3{,}3\text{ V}} \approx \mathbf{0{,}303}\text{ A}$$

$$I_{fuita,90} = \frac{45\text{ W}}{1{,}1\text{ V}} \approx \mathbf{40{,}9}\text{ A}$$

Malgrat que el voltatge s'ha reduït de 3,3 V a 1,1 V (factor 3), el corrent de fuita s'ha multiplicat per 135. Això reflecteix la dificultat creixent de mantenir els transistors en tall quan les dimensions disminueixen.

c) La reducció màxima de P eliminant tota la P_d és, per a cada tecnologia, el percentatge de la potència dinàmica sobre la total:

$$\Delta P_{250} = \frac{49}{50} = \mathbf{98}\ \% \qquad \Delta P_{90} = \frac{75}{120} = \mathbf{62{,}5}\ \%$$

En la tecnologia de 90 nm, la reducció màxima possible de P actuant exclusivament sobre la potència dinàmica és del 62,5 %; la potència estàtica restant (37,5 %) no es pot eliminar per aquesta via.

Tema 7: Memòria cau

7.1 Disseny bàsic

Exemple 87: Solució: adreçament i seqüència de blocs

Enunciat: Problema 7.1.
Memòria principal de 64 bytes (adreces de 6 bits), memòria cau de 16 bytes, correspondència directa, blocs de 4 bytes ($b = 2$). L'MC té $16/4 = 4$ línies ($n_l = 2$). L'adreça es descompon en:

$$A = \underbrace{\cdot\,\cdot}_{\text{2 bits etiqueta}} \quad \underbrace{\cdot\,\cdot}_{\text{2 bits índex}} \quad \underbrace{\cdot\,\cdot}_{\text{2 bits desplaçament}}$$

a) Adreça `0x39` $= 57 = 111001_2$:

$$\underbrace{11}_{\text{etiqueta=3}} \quad \underbrace{10}_{\text{índex=2}} \quad \underbrace{01}_{\text{desplaçament=1}}$$

La dada correspon a la **línia 2**, amb **etiqueta 3** i **byte 1** dins el bloc.
b) Mapatge de blocs a línies (índex = bloc mod 4):

Bloc	0	1	3	7	10	12	15
Línia	0	1	3	3	2	0	3

Evolució de la seqüència (MC inicialment buida):

#	Bloc	Línia	Contingut línia	En/F
1	0	0	buida	**F**
2	1	1	buida	**F**
3	0	0	bloc 0	**En**
4	3	3	buida	**F**
5	12	0	bloc $0 \neq 12$	**F**
6	10	2	buida	**F**
7	3	3	bloc 3	**En**
8	7	3	bloc $3 \neq 7$	**F**
9	12	0	bloc 12	**En**
10	15	3	bloc $7 \neq 15$	**F**
11	3	3	bloc $15 \neq 3$	**F**

Encerts: 3. Fallades: 8. Taxa d'encerts: $h = 3/11 \approx 27{,}3\ \%$.
De les 8 fallades, 7 són **d'arrencada en fred** (*cold-start*: el primer accés a cadascun dels blocs 0, 1, 3, 12, 10, 7 i 15) i la de l'accés 11 és de **conflicte**: el bloc 3, carregat a l'accés 4, ha estat expulsat pels blocs 7 i 15, que competeixen amb ell per la línia 3 (en una memòria cau completament associativa de la mateixa capacitat, aquest accés seria un encert). El conflicte a la línia 3 és especialment greu: els tres blocs s'hi expulsen l'un a l'altre, i el bloc 3 només hi encerta un cop (a l'accés 7).

Exemple 88: Solució: polítiques d'escriptura

Enunciat: Problema 7.2.
Processador de 16 bits. MC de correspondència directa, 256 bytes, blocs de 16 bytes. L'MC té $256/16 = 16$ línies. Descomposició de l'adreça (16 bits):

$$A = \underbrace{\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot}_{\text{8 bits etiqueta}}\ \underbrace{\cdot\cdot\cdot\cdot}_{\text{4 bits índex}}\ \underbrace{\cdot\cdot\cdot\cdot}_{\text{4 bits desplaçament}}$$

Exemples de descomposició:

- `0x4534`: etiqueta = `0x45`, índex = `0x3`, desplaçament = `0x4`.
- `0x13A4`: etiqueta = `0x13`, índex = `0xA`, desplaçament = `0x4`.
- `0x60A0`: etiqueta = `0x60`, índex = `0xA`, desplaçament = `0x0`.

Cada accés L o E transfereix una paraula de 2 bytes (processador de 16 bits).
a) Escriptura immediata sense assignació.
Comportament per tipus d'accés:

- **L encert**: es llegeix de l'MC. Cap tràfic amb MP.
- **L fallada**: es llegeix el bloc sencer d'MP (16 B), es carrega a l'MC i se'n llegeix la dada.
- **E encert**: s'escriu a l'MC i a l'MP (2 B, a través de la memòria intermèdia d'escriptura).
- **E fallada**: s'escriu directament a MP (2 B). L'MC no s'altera.

Les columnes **Lect. MC** i **Escr. MC** indiquen si l'accés llegeix o escriu la matriu de dades de l'MC (la dada lliurada a la CPU, la càrrega d'un bloc, l'escriptura d'una dada o la còpia d'un bloc modificat cap a MP).

#	Tipus	Adr.	Etiq.	Índ.	En/F	Llegits MP	Escrits MP	Lect. MC	Escr. MC
1	L	`4534`	`45`	`3`	**F**	16	0	S	S
2	L	`4568`	`45`	`6`	**F**	16	0	S	S
3	E	`13A4`	`13`	`A`	**F**	0	2	N	N
4	E	`13A8`	`13`	`A`	**F**	0	2	N	N
5	L	`3560`	`35`	`6`	**F**	16	0	S	S
6	E	`453C`	`45`	`3`	**En**	0	2	N	S
7	E	`60A0`	`60`	`A`	**F**	0	2	N	N
8	L	`453C`	`45`	`3`	**En**	0	0	S	N
9	E	`3900`	`39`	`0`	**F**	0	2	N	N
10	L	`A238`	`A2`	`3`	**F**	16	0	S	S

Justificació dels casos clau:

- #3 i #4 (`E 13A4`, `E 13A8`): ambdues escriptures fallen a la línia `A` (buida). Sense assignació, cap de les dues carrega el bloc → la línia resta buida i la segona escriptura també falla.
- #6 (`E 453C`): la línia 3 conté el bloc carregat al pas 1 (etiqueta `45`) → encert. S'escriu a MC i a MP (2 B).
- #8 (`L 453C`): la línia 3 encara conté l'etiqueta `45` → encert.
- #10 (`L A238`): etiqueta `A2` $\neq$ `45` a la línia 3 → fallada, reemplaça.

b) Escriptura retardada amb assignació.
Comportament per tipus d'accés:

- **L/E encert**: accés a MC. Cap tràfic amb MP. En cas d'escriptura, $D \leftarrow 1$.
- **L/E fallada,** $D = 0$: es llegeix el bloc d'MP (16 B) i es carrega a l'MC.
- **L/E fallada,** $D = 1$: es llegeix el bloc antic de l'MC i es copia a MP (16 B), i després es llegeix el nou bloc d'MP (16 B) i es carrega a l'MC.

A les columnes **Lect. MC** i **Escr. MC**, tota fallada escriu l'MC (càrrega del bloc i, si és E, la dada); l'MC es llegeix quan la dada es lliura a la CPU (L) o quan cal copiar un bloc modificat cap a MP. La columna **D** dona el bit de modificació de la línia abans de l'accés i, si canvia, també el de després (abans→després).

#	Tipus	Adr.	Etiq.	Índ.	En/F	D	Llegits MP	Escrits MP	Lect. MC	Escr. MC
1	L	`4534`	`45`	`3`	**F**	0	16	0	S	S
2	L	`4568`	`45`	`6`	**F**	0	16	0	S	S
3	E	`13A4`	`13`	`A`	**F**	0→1	16	0	N	S
4	E	`13A8`	`13`	`A`	**En**	1	0	0	N	S
5	L	`3560`	`35`	`6`	**F**	0	16	0	S	S
6	E	`453C`	`45`	`3`	**En**	0→1	0	0	N	S
7	E	`60A0`	`60`	`A`	**F**	1	16	16	S	S
8	L	`453C`	`45`	`3`	**En**	1	0	0	S	N
9	E	`3900`	`39`	`0`	**F**	0→1	16	0	N	S
10	L	`A238`	`A2`	`3`	**F**	1→0	16	16	S	S

Justificació dels casos clau:

- #3 (`E 13A4`): amb assignació, es carrega el bloc d'MP (16 B) a la línia `A` i s'escriu la dada a l'MC ($D \leftarrow 1$).
- #4 (`E 13A8`): encert (el bloc és a la línia `A` des del pas 3). S'escriu a MC, D ja és 1.
- #7 (`E 60A0`): la línia `A` conté l'etiqueta `13` amb $D = 1$. Fallada: cal copiar primer el bloc antic a MP (16 B *escrits*), després carregar el nou bloc d'MP (16 B *llegits*), i finalment escriure la dada ($D \leftarrow 1$).
- #10 (`L A238`): la línia 3 conté l'etiqueta `45` amb $D = 1$ (des del pas 6). Fallada amb còpia del bloc modificat a MP: 16 B escrits + 16 B llegits.

c) Resum comparatiu:

	Escr. immediata s/a	Escr. retardada a/a
Taxa de fallades	80 % (8/10)	70 % (7/10)
Bytes llegits d'MP	64	112
Bytes escrits a MP	10	32
Bytes totals MP	74	144

L'escriptura retardada amb assignació té una taxa de fallades inferior (l'accés #4 passa de fallada a encert perquè el bloc es va assignar al pas 3), però genera més tràfic total amb MP: cada assignació per escriptura carrega un bloc de 16 B, i cada escriptura a MP d'un bloc modificat n'escriu 16 B més. L'escriptura immediata sense assignació genera menys tràfic perquè les escriptures que fallen només transfereixen la paraula escrita (2 B), sense carregar blocs sencers.

7.2 Rendiment

Exemple 89: Solució: comparació de polítiques d'escriptura

Enunciat: Problema 7.6.
Dades: $p_e = 20$ %, $p_m = 1/3$, $t_h = 10$ ns, $t_{bloc} = 100$ ns, $h_1 = 0{,}9$ (immediata s/a), $h_2 = 0{,}85$ (retardada a/a).
a) Aplicant Equació 7.1, amb el temps de penalització mitjà per fallada de cada política (Essencial 7.2; per a l'escriptura retardada, Essencial 7.3). Se suposa que la taxa de fallades és independent del tipus d'accés: així, la fracció de fallades que corresponen a escriptures coincideix amb el percentatge d'escriptures p_e.
Escriptura immediata sense assignació ($m_1 = 0{,}10$): les fallades de lectura paguen $t_{bloc} + t_h$ i les d'escriptura paguen 0 (la memòria intermèdia absorbeix l'escriptura):

$$t_{p_1} = (1 - p_e)(t_{bloc} + t_h) = 0{,}80 \cdot 110 = 88 \text{ ns}$$

$$\text{TAM}_1 = t_h + m_1 \cdot t_{p_1} = 10 + 0{,}10 \cdot 88 = \mathbf{18{,}8} \text{ ns}$$

Escriptura retardada amb assignació ($m_2 = 0{,}15$): la penalització depèn de si el bloc expulsat està modificat:

$$t_{p_2} = (1 + p_m) \cdot t_{bloc} + t_h = \frac{4}{3} \cdot 100 + 10 = 133{,}3 + 10 \approx 143{,}3 \text{ ns}$$

$$\text{TAM}_2 = t_h + m_2 \cdot t_{p_2} = 10 + 0{,}15 \cdot 143{,}3 \approx 10 + 21{,}5 = \mathbf{31{,}5}\ \text{ns}$$

L'escriptura immediata sense assignació és més ràpida (18,8 vs. 31,5 ns). Té una taxa de fallades inferior (10 % vs. 15 %) i, sobretot, una penalització mitjana per fallada molt menor: les fallades d'escriptura no paguen penalització i les de lectura no generen escriptures de blocs modificats a MP. La penalització mitjana de l'escriptura retardada és un 63 % més gran (143,3 vs. 88 ns).
b) Per a un programa de només lectures ($p_e = 0$), ambdues polítiques es comporten de manera **idèntica**: en absència d'escriptures, no hi ha diferència entre escriptura immediata i retardada ($p_m = 0$, cap bloc es modifica), i ambdues polítiques carreguen blocs a l'MC exactament igual en les fallades de lectura. La taxa de fallades seria la mateixa per a ambdues i la penalització també ($t_{bloc} + t_h = 110$ ns), de manera que cap de les dues alternatives seria més ràpida que l'altra.

Exemple 90: Solució: MC separades d'instruccions i dades

Enunciat: Problema 7.7.
Dades: $\text{CPI}_{ideal} = 1{,}5$, $t_c = 10$ ns, $n_r = 1{,}6$ referències/instrucció. MC de dades amb escriptura retardada amb assignació.

	MC Instruccions	MC Dades
Accessos per instrucció	1	0,6
m	4 %	10 %
t_p (bloc no modif.)	10 cicles	15 cicles
t_p (bloc modif.)	—	20 cicles
p_m	—	20 %

a) Penalització per fallada a l'MC de dades (escriptura retardada):

$$t_{pd} = (1 - p_m) \cdot 15 + p_m \cdot 20 = 0{,}80 \cdot 15 + 0{,}20 \cdot 20 = 12 + 4 = 16 \text{ cicles}$$

El percentatge d'escriptures ($p_e = 40$ %) no és necessari per resoldre el problema: la penalització de l'MC de dades ja es dona directament en funció de si el bloc reemplaçat és modificat o no.
TAM ponderat per la fracció d'accessos de cada MC:

$$\text{TAM} = t_h + \frac{1}{n_r} \cdot m_i \cdot t_{pi} + \frac{0{,}6}{n_r} \cdot m_d \cdot t_{pd}$$

$$= 1 + \frac{1}{1{,}6} \cdot 0{,}04 \cdot 10 + \frac{0{,}6}{1{,}6} \cdot 0{,}10 \cdot 16$$

$$= 1 + 0{,}625 \cdot 0{,}4 + 0{,}375 \cdot 1{,}6 = 1 + 0{,}25 + 0{,}6 = \mathbf{1{,}85} \text{ cicles}$$

b) Cicles de penalització per instrucció:

$$\text{CPI}_{penal,i} = m_i \cdot 1 \cdot t_{pi} = 0{,}04 \cdot 10 = 0{,}4 \text{ cicles/instr.}$$

$$\text{CPI}_{penal,d} = m_d \cdot 0{,}6 \cdot t_{pd} = 0{,}10 \cdot 0{,}6 \cdot 16 = 0{,}96 \text{ cicles/instr.}$$

$$\text{CPI} = \text{CPI}_{ideal} + \text{CPI}_{penal,i} + \text{CPI}_{penal,d} = 1{,}5 + 0{,}4 + 0{,}96 = 2{,}86$$

$$t_{exe} = \text{CPI} \cdot t_c = 2{,}86 \cdot 10 = \mathbf{28{,}6} \text{ ns/instrucció}$$

Observació: les fallades de memòria cau incrementen el CPI un 90,7 % respecte del valor ideal (d'1,5 a 2,86). L'MC de dades, amb només el 37,5 % dels accessos, contribueix el 70,6 % de la penalització (0,96/1,36), a causa de la seva taxa de fallades més alta i la penalització per les escriptures de blocs modificats a MP.

7.3 Associativitat i multinivell

Exemple 91: Solució: associativa de 2 vies

Enunciat: Problema 7.10.
Processador de 16 bits. MC associativa de 2 vies, 1 KiB, blocs de 16 bytes, escriptura immediata sense assignació, LRU. L'MC té $1024/16 = 64$ línies organitzades en $64/2 = 32$ conjunts de 2 vies.
Descomposició de l'adreça (16 bits):

$$A = \underbrace{\cdots\cdots}_{\text{7 bits etiqueta}} \quad \underbrace{\cdots\cdots}_{\text{5 bits conjunt}} \quad \underbrace{\cdots\cdots}_{\text{4 bits desplaçament}}$$

El conjunt es determina pels bits [8:4] i l'etiqueta pels bits [15:9]. Descomposició de les adreces:

Adreça	Bloc MP	Conjunt	Etiqueta
`A930`	`A93`	19 (`0x13`)	`0x54`
`B930`	`B93`	19	`0x5C`
`A972`	`A97`	23 (`0x17`)	`0x54`
`A932`	`A93`	19	`0x54`

C935	C93	19	0x64
C934	C93	19	0x64
B976	B97	23	0x5C
B936	B93	19	0x5C
B938	B93	19	0x5C
A978	A97	23	0x54

Exemple de descomposició per a `0xA972`:

$$\texttt{A972}_{16} = \underbrace{1010100}_{0x54}\ \underbrace{10111}_{23}\ \underbrace{0010}_{2}$$

Els bits [8:4] són $10111_2 = 23$, no $00111_2 = 7$. El bit 8 (que forma part del dígit hex `9` $= 1001_2$) és 1.
La seqüència accedeix a dos conjunts: **19** (7 accessos) i **23** (3 accessos). Amb escriptura immediata sense assignació, les escriptures que fallen no modifiquen l'MC.
Evolució del conjunt 19 (2 vies):

#	Accés	En/F	Via 0	Via 1	LRU
1	L `A930` (0x54)	**F**	**0x54**	—	v1
2	L `B930` (0x5C)	**F**	0x54	**0x5C**	v0
4	E `A932` (0x54)	**En**	0x54	0x5C	v1
5	E `C935` (0x64)	**F**	0x54	0x5C	(s/a)
6	L `C934` (0x64)	**F**	0x54	**0x64**	v0
8	E `B936` (0x5C)	**F**	0x54	0x64	(s/a)
9	L `B938` (0x5C)	**F**	**0x5C**	0x64	v1

- #4: encert a via 0 (etiqueta `0x54`). S'escriu a MC i a MP (2 B). LRU passa a v1.
- #5: fallada, però és E sense assignació → MC inalterada, LRU inalterat.
- #6: fallada de lectura. LRU = v1 (des de #4; #5 no altera res) → s'expulsa v1 (`0x5C`) i es carrega `0x64`.
- #8: E fallada sense assignació → MC inalterada.
- #9: L fallada. LRU = v0 (des de #6) → s'expulsa v0 (`0x54`) i es carrega `0x5C`.

Evolució del conjunt 23 (2 vies):

#	Accés	En/F	Via 0	Via 1	LRU
3	E `A972` (0x54)	**F**	—	—	(s/a)
7	E `B976` (0x5C)	**F**	—	—	(s/a)
10	L `A978` (0x54)	**F**	**0x54**	—	v1

Les escriptures #3 i #7 fallen, però sense assignació no carreguen cap bloc. El conjunt 23 resta buit fins a la lectura #10.
Taula resum:

#	Tipus	Adr.	Bloc	Conj.	En/F	Lectura MP	Escriptura MP
1	L byte	`A930`	`A93`	19	**F**	S, `A930`, 16 B	N
2	L paraula	`B930`	`B93`	19	**F**	S, `B930`, 16 B	N
3	E byte	`A972`	`A97`	23	**F**	N	S, `A972`, 1 B
4	E paraula	`A932`	`A93`	19	**En**	N	S, `A932`, 2 B
5	E byte	`C935`	`C93`	19	**F**	N	S, `C935`, 1 B
6	L paraula	`C934`	`C93`	19	**F**	S, `C930`, 16 B	N
7	E byte	`B976`	`B97`	23	**F**	N	S, `B976`, 1 B
8	E paraula	`B936`	`B93`	19	**F**	N	S, `B936`, 2 B
9	L byte	`B938`	`B93`	19	**F**	S, `B930`, 16 B	N
10	L paraula	`A978`	`A97`	23	**F**	S, `A970`, 16 B	N

Encerts: 1 (pas 4). Fallades: 9. Taxa de fallades: 90 %.
La taxa de fallades és molt alta perquè la seqüència barreja accessos a tres blocs diferents dins el conjunt 19 (etiquetes `0x54`, `0x5C` i `0x64`), i les escriptures sense assignació impedeixen que els blocs escrits s'instal · lin a l'MC per a futurs accessos.

Exemple 92: Solució: TAM multinivell

Enunciat: Problema 7.13.
Un accés a memòria segueix la cadena L1 → L2 → MP:

1. S'accedeix a L1 amb temps $t_{h_{L1}}$.
2. En cas de fallada a L1 (probabilitat m_{L1}), s'accedeix a L2 amb temps $t_{h_{L2}}$.
3. En cas de fallada a L2 (probabilitat m_{L2}, local), s'accedeix a MP amb penalització t_p.

El temps mitjà d'accés és la suma del temps de L1 més les penalitzacions ponderades:

$$\boxed{\text{TAM} = t_{h_{L1}} + m_{L1} \cdot (t_{h_{L2}} + m_{L2} \cdot t_p)}$$

La fórmula coincideix amb Equació 7.5. Desenvolupant:

$$\text{TAM} = t_{h_{L1}} + m_{L1} \cdot t_{h_{L2}} + m_{L1} \cdot m_{L2} \cdot t_p$$

El terme $m_{L1} \cdot m_{L2}$ és la **taxa de fallades global** de L2 (proporció de tots els accessos que fallen tant a L1 com a L2 i arriben a MP).
Observació: en un sistema ben dissenyat, $t_{h_{L1}}$ domina el TAM perquè afecta tots els accessos. Per això L1 s'optimitza per minimitzar $t_{h_{L1}}$, mentre que L2 s'optimitza per minimitzar m_{L2}.

7.4 Tipologia de les fallades

Exemple 93: Solució: classificació de fallades

Enunciat: Problema 7.15.
MC de correspondència directa, 8 línies, blocs de 4 bytes ($b = 2$). Capacitat total: $8 \times 4 = 32$ bytes. El mapatge és línia = bloc mod 8, on bloc = adreça $\div$ 4. La classificació de les fallades segueix les tres C (Secció 7.8, Essencial 7.4).
a) Seqüència A: 0, 4, 8, 12, 16, 20, 24, 28, 0, 4, 8, 12
Blocs accedits: 0, 1, 2, 3, 4, 5, 6, 7, 0, 1, 2, 3. Cadascun ocupa una línia diferent (k mod $8 = k$ per a $k = 0 \dots 7$). El *working set* (8 blocs) cap exactament a l'MC.

Accés	Bloc	Línia	Resultat	Tipus
0	0	0	Fallada	Obligatòria
4	1	1	Fallada	Obligatòria
8	2	2	Fallada	Obligatòria
12	3	3	Fallada	Obligatòria
16	4	4	Fallada	Obligatòria
20	5	5	Fallada	Obligatòria
24	6	6	Fallada	Obligatòria
28	7	7	Fallada	Obligatòria
0	0	0	**Encert**	—
4	1	1	**Encert**	—
8	2	2	**Encert**	—
12	3	3	**Encert**	—

Total: **8 fallades** (totes obligatòries), **4 encerts**. Cap fallada de conflicte (cada bloc té línia pròpia) ni de capacitat (el *working set* cap a l'MC).
Seqüència B: 0, 4, 8, 12, 0, 4, 8, 12, 0, 4, 8, 12
Blocs: 0, 1, 2, 3 (repetits 3 vegades). Quatre blocs, cadascun amb línia diferent: el *working set* (4 blocs) és la meitat de la capacitat.

- Primer recorregut (0, 4, 8, 12): 4 fallades d'arrencada en fred.
- Segon i tercer recorreguts: tot encerts (els 4 blocs continuen a l'MC).

Total: **4 fallades** (totes obligatòries), **8 encerts**.
Seqüència C: 0, 32, 0, 32, 0, 32, 0, 32
Blocs: 0, 8, 0, 8, … El bloc 0 i el bloc 8 es mapen tots dos a la línia 0 ($0 \bmod 8 = 0$, $8 \bmod 8 = 0$).

Accés	Bloc	Línia	Resultat	Tipus
0	0	0	Fallada	Obligatòria
32	8	0	Fallada	Obligatòria (expulsa bloc 0)
0	0	0	Fallada	Conflicte
32	8	0	Fallada	Conflicte
0	0	0	Fallada	Conflicte
32	8	0	Fallada	Conflicte
0	0	0	Fallada	Conflicte
32	8	0	Fallada	Conflicte

Total: **8 fallades** (2 obligatòries, 6 de conflicte), **0 encerts**. El *working set* és de només 2 blocs (molt inferior a la capacitat de 8 línies), però tots dos competeixen per la mateixa línia. L'MC té 7 línies buides que no s'aprofiten.
b) Per eliminar les fallades de conflicte de la seqüència C, cal que els blocs 0 i 8 puguin coexistir a l'MC. La solució és **augmentar l'associativitat** a 2 o més vies: amb 2 vies, el conjunt 0 tindria dues posicions i ambdós blocs hi cabrien simultàniament.
c) Amb una MC associativa de 2 vies de la mateixa capacitat total (8 línies → 4 conjunts de 2 vies):

- **Seqüència C**: blocs 0 i 8 es mapen al conjunt $0 \bmod 4 = 0$ i $8 \bmod 4 = 0$. Amb 2 vies, ambdós hi caben. Després de les 2 fallades d'arrencada en fred, els 6 accessos restants serien encerts. **Sí, millora** (de 8 a 2 fallades).
- **Seqüència B**: blocs 0, 1, 2, 3 es mapen als conjunts 0, 1, 2, 3. Cada bloc ocupa un conjunt diferent: no hi ha conflictes ni amb correspondència directa. **No millora** (les 4 fallades d'arrencada en fred són igualment inevitables).

Exemple 94: Solució: conflictes entre dos vectors

Enunciat: Problema 7.16.
`A[256]` d'enters de 4 bytes, des de l'adreça `0x000` (1 KiB, fins a `0x3FF`). `B[256]` des de `0x400` (fins a `0x7FF`). MC de correspondència directa, 16 línies, blocs de 32 bytes. Variables `i` i `suma` en registres. Paràmetres: bloc = 32 bytes = 8 enters per bloc. A i B ocupen cadascun $256/8 = 32$ blocs.
a) En el primer recorregut, cada bloc s'accedeix per primera vegada: **32 fallades d'arrencada en fred per a A** i **32 per a B** (una per bloc, quan s'accedeix a $A[0], A[8], A[16], \ldots, A[248]$ i anàlogament per a B).

b) Mapatge de cada vector:

- Bloc k d'A ($k = 0 \dots 31$): adreça base = $32k$. Número de bloc = k. Línia = k mod 16.
- Bloc k de B ($k = 0 \dots 31$): adreça base = `0x400`$+32k$. Número de bloc = $1024/32+k = 32+k$. Línia = $(32 + k)$ mod 16 = k mod 16.

El bloc k d'A i el bloc k de B es mapen a la **mateixa línia** (k mod 16). Això és perquè la distància entre A i B (`0x400` − `0x000` = 1024 bytes, 1 KiB, que són 32 blocs) és un múltiple exacte del nombre de línies de l'MC (16).
En el bucle, els accessos s'intercalen: $A[i], B[i], A[i + 1], B[i + 1], \dots$ Dins de cada grup de 8 iteracions (que correspon a un bloc), l'accés a $A[i]$ carrega el bloc d'A a la línia k, i l'accés a $B[i]$ el reemplaça pel bloc de B; i viceversa.

- $A[0]$ (bloc 0, línia 0): **fallada** (obligatòria), carrega bloc 0.
- $B[0]$ (bloc 32, línia 0): **fallada** (obligatòria), expulsa bloc 0.
- $A[1]$ (bloc 0, línia 0): **fallada** (conflicte), expulsa bloc 32.
- $B[1]$ (bloc 32, línia 0): **fallada** (conflicte), expulsa bloc 0.
- ... i així fins a $A[7], B[7]$.

Dins de cada grup de 8 iteracions: 2 fallades d'arrencada en fred (primer accés a cada bloc) + 14 fallades de conflicte. El patró es repeteix per a cada bloc, amb 32 grups. En total: **tots els 512 accessos a dades fallen**, cosa que anul · la completament l'efecte de l'MC.
Sí, es produeixen fallades de conflicte massives, causades per l'alineament dels dos vectors respecte a l'MC.
c) Si B comença a `0x600`:

- Bloc k de B: adreça base = `0x600`$+32k = 1536+32k$. Número de bloc = $1536/32+k = 48+k$. Línia = $(48 + k)$ mod 16 = k mod 16.

El mapatge és idèntic: la distància entre A i B (`0x600` = 1536 bytes = 48 blocs) torna a ser un múltiple exacte del nombre de línies (16). El patró de conflictes **no canvia**.
Observació: perquè els conflictes desapareguin, la distància entre A i B (en nombre de blocs) **no ha de ser múltiple de 16**. Per exemple, si B comencés a `0x420` (bloc 33), el bloc k d'A aniria a la línia k mod 16 i el bloc k de B a la línia $(33+k)$ mod 16 = $(k+1)$ mod 16: línies diferents, sense conflictes. Això il · lustra que les **fallades de conflicte depenen de l'adreça base relativa dels vectors**, un efecte que el programador no controla directament, però que el compilador o l'assignador de memòria poden mitigar mitjançant **farciment** (*padding*) o alineament estratègic.

Tema 8: Memòria virtual

8.1 Traducció d'adreces i fallada de pàgina

Exemple 95: Solució: paginació bàsica i traça LRU

Enunciat: Problema 8.1.
Memòria virtual de 16 MiB (2^{24} B $\rightarrow$ 24 bits d'adreça lògica), memòria física de 16 KiB (2^{14} B $\rightarrow$ 14 bits d'adreça física) i pàgines de 256 B (2^8 B $\rightarrow$ desplaçament de 8 bits).
La descomposició VPN/desplaçament segueix la Secció 8.2.2:

- VPN: $24 - 8 = 16$ bits $\rightarrow 2^{16} = 65\,536$ pàgines virtuals.
- PPN: $14 - 8 = 6$ bits $\rightarrow 2^6 = 64$ marcs de pàgina.

a) La taula de pàgines té una entrada per pàgina virtual: $2^{16} = 65\,536$ entrades. Cada entrada conté el PPN (6 bits) més el bit V i el bit D:

$$6 + 1 + 1 = 8 \text{ bits} = 1 \text{ byte/entrada}$$

$$\text{Mida taula de pàgines} = 2^{16} \times 1 \text{ B} = 64 \text{ KiB}$$

b) El VPN és $\lfloor \text{adreça}/\texttt{0x100} \rfloor$ (els dos dígits hexadecimals inferiors són el desplaçament de 8 bits).
Cada programa pot tenir com a màxim 3 pàgines a memòria física, amb reemplaçament LRU.
Seqüència de VPN: 0, 1, 1, 2, 0, 1, 4, 5, 1.
La pila LRU s'escriu de la pàgina *most recently used* (MRU), la usada més recentment, a la menys recent.

Adreça	A — Pàg.	B — Fallada	C — Reempl.	Pila LRU (MRU→LRU)
`000023`	`0`	**Sí**	—	0
`000101`	`1`	**Sí**	—	1, 0
`000102`	`1`	No	—	1, 0
`000200`	`2`	**Sí**	—	2, 1, 0
`000010`	`0`	No	—	0, 2, 1
`000111`	`1`	No	—	1, 0, 2
`000416`	`4`	**Sí**	`2`	4, 1, 0
`000520`	`5`	**Sí**	`0`	5, 4, 1
`000101`	`1`	No	—	1, 5, 4

Hi ha **5 fallades** i 4 encerts. A `000416` s'expulsa la pàgina 2 (la menys recentment usada, desplaçada pels encerts a 0 i 1); a `000520` s'expulsa la pàgina 0.

Exemple 96: Solució: taula de pàgines de dos nivells

Enunciat: Problema 8.2.
Adreces de 32 bits, pàgines de 4 KiB (2^{12} B), entrades de 4 B, dos nivells amb repartiment simètric.
a) Desplaçament: $\log_2(4\text{ KiB}) = 12$ bits. Resten $32-12 = 20$ bits d'índex, repartits simètricament:

$$\underbrace{10\text{ bits}}_{\text{índex N1}}\ \underbrace{10\text{ bits}}_{\text{índex N2}}\ \underbrace{12\text{ bits}}_{\text{desplaçament}}$$

b) Cada taula (de qualsevol nivell) té $2^{10} = 1024$ entrades de 4 B:

$$2^{10} \times 4\text{ B} = 4\text{ KiB}$$

Tant la taula de primer nivell com cada taula de segon nivell ocupen **4 KiB** (exactament una pàgina).
c) Cada entrada del primer nivell pot apuntar a una taula de segon nivell: com a màxim $2^{10} = 1024$ taules de segon nivell. Mida total en el **pitjor cas** (totes assignades):

$$\underbrace{4\text{ KiB}}_{\text{N1}} + 1024 \times \underbrace{4\text{ KiB}}_{\text{N2}} = 4\text{ KiB} + 4\text{ MiB}$$

d) Una taula de **nivell únic** tindria 2^{20} entrades de 4 B = 4 MiB. La jerarquia de dos nivells, en el pitjor cas, ocupa lleugerament **més** (4 MiB + 4 KiB: el sobrecost del directori de primer nivell). L'avantatge apareix en el **cas típic**, en què el procés només usa una fracció de l'espai lògic: només cal assignar les taules de segon nivell de les regions realment utilitzades. Per exemple, un procés que toqui codi, dades i pila en 3 regions disjuntes necessita la taula de primer nivell (4 KiB) i tres de segon nivell (una per regió), és a dir, **16 KiB** en lloc de 4 MiB. Amb un nivell únic, en canvi, sempre cal reservar els 4 MiB sencers encara que la majoria d'entrades siguin invàlides.

Exemple 97: Solució: traça de la taula de pàgines amb accés a disc

Enunciat: Problema 8.3.
Adreça lògica de 16 bits, física de 15 bits, pàgines de 8 KiB (2^{13} → desplaçament de 13 bits). VPN: $16-13 = 3$ bits (8 pàgines); PPN: $15-13 = 2$ bits (4 marcs). El VPN és $\lfloor \text{adreça}/\texttt{0x2000} \rfloor$ (els 3 bits de més pes de l'adreça).
Estat inicial: PPN 0 = VPN 2, PPN 1 = VPN 3, PPN 2 = VPN 4, PPN 3 lliure. Ordre LRU (MRU→LRU): VPN 3, VPN 2, VPN 4. Cap pàgina està modificada ($D = 0$).

Adreça	VPN	PPN	Fallada	Lect. disc	Escr. disc	Reempl. VPN	Reempl. PPN
E `F458`	7	3	**Sí**	Sí	No	—	—
E `8666`	4	2	No	No	No	—	—
L `1BBF`	0	0	**Sí**	Sí	No	2	0
E `5C44`	2	1	**Sí**	Sí	No	3	1
L `6600`	3	3	**Sí**	Sí	**Sí**	7	3
L `4000`	2	1	No	No	No	—	—

Punts clau:

- `E F458` (VPN 7): fallada amb el marc lliure PPN 3; com que és escriptura, queda $D = 1$.
- `L 1BBF` (VPN 0) i `E 5C44` (VPN 2): cap marc lliure; s'expulsen les pàgines més antigues (VPN 2 i VPN 3), totes dues amb $D = 0$, de manera que no cal escriure-les a disc.
- `L 6600` (VPN 3): s'expulsa la pàgina del VPN 7, que té $D = 1 \rightarrow$ **cal escriure-la primer a disc** (única escriptura a disc de la traça).
- `L 4000` (VPN 2): encert, la pàgina segueix a PPN 1.

Estat final de la taula de pàgines:

VPN	V	D	PPN
0	1	0	0
1	0	0	—
2	1	1	1
3	1	0	3
4	1	1	2
5–7	0	0	—

Estat final de la memòria física: PPN 0 = VPN 0, PPN 1 = VPN 2, PPN 2 = VPN 4, PPN 3 = VPN 3.

8.2 TLB

Exemple 98: Solució: traça del TLB i de la taula de pàgines

Enunciat: Problema 8.5.
Pàgines de 4 KiB → VPN = ⌊adreça/4096⌋. TLB de 4 entrades completament associatiu, LRU (vegeu Secció 8.5). La quarta entrada ($V = 0$) és lliure i s'omple en la primera fallada; mentre n'hi hagi cap de lliure, el reemplaçament LRU no actua. Ordre LRU inicial de les vàlides (MRU→LRU): VPN 11, VPN 7, VPN 3.
a) Llista 1. VPN: 0, 7, 3, 3, 1, 1, 2.

Ref.	VPN	Fallada TLB	Fallada pàgina	PPN
4095	0	**Sí**	No	5
31272	7	No	No	4
15789	3	No	No	6
15000	3	No	No	6
7193	1	**Sí**	**Sí**	13
4096	1	No	No	13
8912	2	**Sí**	**Sí**	14

- 4095 (VPN 0): fallada de TLB; l'MMU llegeix la PTE, que té $V = 1$ (PPN 5), i la copia a l'entrada lliure, sense fallada de pàgina.
- 7193 (VPN 1): fallada de TLB i la PTE té $V = 0 \rightarrow$ **fallada de pàgina**; l'SO carrega la pàgina i li assigna PPN 13. En reexecutar l'accés, l'MMU copia la PTE al TLB reemplaçant l'entrada del VPN 11 (l'única no reutilitzada des de l'inici $\rightarrow$ la més antiga).
- 8912 (VPN 2): fallada de TLB i $V = 0 \rightarrow$ fallada de pàgina (PPN 14); en reexecutar, es reemplaça l'entrada del VPN 0 (la més antiga de la pila en aquest punt).

TLB final: (VPN 1, PPN 13), (VPN 7, PPN 4), (VPN 3, PPN 6), (VPN 2, PPN 14). Taula de pàgines: VPN 1 $\rightarrow$ V 1/PPN 13, VPN 2 $\rightarrow$ V 1/PPN 14. Total: 3 fallades de TLB, 2 de les quals són fallades de pàgina. Ordre LRU final, de la més recent a la més antiga: VPN 2, 1, 3 i 7.
Llista 2 (des de l'estat inicial). VPN: 2, 7, 4, 11, 9, 4, 11.

Ref.	VPN	Fallada TLB	Fallada pàgina	PPN
9452	2	**Sí**	**Sí**	13
30964	7	No	No	4
19136	4	**Sí**	No	9
46502	11	No	No	12
38110	9	**Sí**	**Sí**	14
17653	4	No	No	9
47480	11	No	No	12

- 9452 (VPN 2): fallada de TLB i $V = 0 \rightarrow$ fallada de pàgina (PPN 13); s'omple l'entrada lliure.
- 19136 (VPN 4): fallada de TLB però $V = 1$ (PPN 9) $\rightarrow$ sense fallada de pàgina. Es reemplaça l'entrada del VPN 3 (la més antiga segons l'ordre inicial $11 > 7 > 3$).
- 38110 (VPN 9): fallada de TLB i $V = 0 \rightarrow$ fallada de pàgina (PPN 14); es reemplaça l'entrada del VPN 2 (la menys recentment usada).
- 46502 i 47480 (VPN 11): encerts, perquè l'entrada del VPN 11 era la més recent inicialment i no s'ha expulsat.

TLB final: (VPN 11, PPN 12), (VPN 7, PPN 4), (VPN 4, PPN 9), (VPN 9, PPN 14). Taula de pàgines: VPN 2 $\rightarrow$ V 1/PPN 13, VPN 9 $\rightarrow$ V 1/PPN 14. Total: 3 fallades de TLB, 2 de les quals són fallades de pàgina. Ordre LRU final: VPN 11, 4, 9 i 7.
b) Amb pàgines de 16 KiB, VPN = $\lfloor$adreça/16 384$\rfloor$ i moltes adreces que abans queien en pàgines diferents passen a compartir pàgina:

- Llista 1: pàgines 0, 1, 0, 0, 0, 0, 0 $\rightarrow$ només **2 pàgines diferents** (abans 5).
- Llista 2: pàgines 0, 1, 1, 2, 2, 1, 2 $\rightarrow$ només **3 pàgines diferents** (abans 5).

En els dos casos el nombre de pàgines diferents accedides baixa, cosa que **redueix el nombre potencial de fallades de TLB d'arrencada en fred** (*cold-start*: una fallada per pàgina nova referenciada). No es pot donar, però, un nombre exacte de fallades comparable al de l'apartat a): l'estat inicial del TLB (VPN 11, 7, 3 vàlides) es va definir per a pàgines de 4 KiB i deixa de tenir sentit en reinterpretar-lo amb una mida de pàgina diferent — els VPN ja no designen els mateixos rangs d'adreces lògiques, de manera que no hi ha una manera coherent de «traduir» aquell estat inicial concret al nou context.
En general, augmentar la mida de pàgina **redueix el nombre de pàgines diferents** accedides i, per tant, les fallades de TLB d'arrencada en fred, a canvi de més fragmentació interna i transferències a disc més grans en cada fallada de pàgina.
c) Mida de la taula de pàgines (nivell únic):

$$\text{Nre. entrades} = \frac{2^{\text{bits adreça}}}{\text{mida pàgina}}, \qquad \text{Mida taula de pàgines} = \text{Nre. entrades} \times \text{mida entrada}$$

- 32 bits, pàgines de 4 KiB, entrades de 4 B: $\dfrac{2^{32}}{2^{12}} \times 4 = 2^{20} \times 4 = \mathbf{4}$ MiB.
- 64 bits, pàgines de 16 KiB, entrades de 8 B: $\dfrac{2^{64}}{2^{14}} \times 8 = 2^{50} \times 2^3 = 2^{53}$ B = **8** PiB.

El segon resultat (8 PiB **per procés**) és inviable i justifica les taules de pàgines multinivell de Solució 96.

8.3 Protecció

Exemple 99: Solució: bit de permís d'escriptura i excepcions

Enunciat: Problema 8.8.
Pàgines de 4 KiB $\rightarrow$ VPN = $\lfloor$adreça/`0x1000`$\rfloor$. Criteri: una **lectura** té èxit sempre que $V = 1$ (el bit E només restringeix escriptures); una **escriptura** requereix $V = 1$ **i** $E = 1$. Si $V = 0 \rightarrow$ fallada de pàgina; si l'escriptura troba $E = 0 \rightarrow$ excepció de protecció.

Accés	Adreça	VPN	V	E	Resultat
a	L `0x0FF0`	0	1	0	**Èxit** (la lectura no mira E); PPN 3
b	E `0x1234`	1	1	1	**Èxit**; PPN 7
c	L `0x2000`	2	0	—	**Fallada de pàgina** ($V = 0$)
d	E `0x3100`	3	1	0	**Excepció de protecció** (escriptura amb $E = 0$)
e	E `0x0010`	0	1	0	**Excepció de protecció** (escriptura amb $E = 0$)

Observació: els casos a) i e) accedeixen a la **mateixa pàgina** (VPN 0, només lectura). La lectura té èxit i l'escriptura provoca excepció: és el patró típic d'una pàgina de codi o de dades constants, protegida contra modificacions accidentals.

8.4 Integració del TLB i la memòria cau

Exemple 100: Solució: flux VIPT i condició antialiàsing

Enunciat: Problema 8.9.
Pàgines de 4 KiB, memòria cau de dades de correspondència directa amb 64 línies amb blocs de 32 B, TLB completament associatiu de 16 entrades. Adreces lògiques de 32 bits.
a) Descomposició de l'adreça lògica:

- Desplaçament dins la pàgina: $\log_2(4\text{ KiB}) = 12$ bits.
- VPN: $32 - 12 = 20$ bits.
- Desplaçament dins el bloc: $\log_2(32) = 5$ bits.
- Índex de la memòria cau: $\log_2(64) = 6$ bits.

$$\underbrace{b_{31}\cdots b_{12}}_{\text{VPN (20)}} \quad \underbrace{b_{11}\cdots b_{0}}_{\text{desplaçament pàgina (12)}} \quad , \quad \underbrace{b_{31}\cdots b_{11}}_{\text{etiqueta (21)}} \quad \underbrace{b_{10}\cdots b_{5}}_{\text{índex cau (6)}} \quad \underbrace{b_{4}\cdots b_{0}}_{\text{despl. bloc (5)}}$$

Cal notar que índex + desplaçament de bloc $= 6 + 5 = 11$ bits, tots dins dels 12 bits del desplaçament de pàgina.
b) Lectura amb encert de TLB i encert de memòria cau (esquema VIPT):

1. La CPU genera l'adreça lògica. **En paral · lel**: els bits d'índex (b10–b5, part del desplaçament) seleccionen la línia de la memòria cau, i el VPN (b31–b12) s'envia al TLB.
2. El TLB troba el VPN (encert) i retorna el PPN.
3. La memòria cau llegeix l'etiqueta i les dades de la línia indexada.
4. Es compara l'etiqueta física —el PPN i el bit b11, que és del desplaçament de pàgina i no canvia en traduir l'adreça— amb l'etiqueta llegida de la memòria cau → coincideixen → **encert de memòria cau**.
5. Es lliura la dada (el desplaçament de bloc, b4–b0, selecciona el byte/paraula).

Com que la traducció i la indexació es fan alhora, el temps d'accés és pràcticament el d'accedir a la memòria cau.
c) Fallada de TLB però pàgina present a memòria:

1. El VPN no és al TLB → **fallada de TLB**. Es llegeix la PTE de la taula de pàgines a memòria (adreça del registre de taula de pàgines + VPN × 4, la mida d'una PTE).

2. La PTE té $V = 1$ (pàgina present) → **no hi ha fallada de pàgina**. L'MMU copia la PTE al TLB (entrada lliure o reemplaçament LRU), sense interrompre el programa.
3. La traducció continua amb el PPN de la PTE: l'etiqueta física (el PPN i b11) es compara amb la de la memòria cau → encert de memòria cau → dada.

El cost addicional respecte de b) és **un accés a memòria** per llegir la PTE (amb la taula de pàgines plana del tema; amb una taula multinivell, un per nivell).
d) Com s'explica a Secció 8.7.3, perquè una memòria cau **VIPT** no pateixi **aliàsing** (*aliasing*) (i es comporti, pel que fa a la selecció de la línia, com una **PIPT**), l'índex de la memòria cau ha d'usar **només bits del desplaçament de pàgina** (bits que no canvien en la traducció). Això equival a:

$$N_C \cdot B \leq T$$

on N_C és el nombre de conjunts, B la mida del bloc (el producte és la mida d'una via) i T la mida de pàgina. Aquí, la memòria cau és de correspondència directa, amb $N_C = 64$ conjunts d'una línia, $B = 32$ bytes i $T = 4$ KiB:

$$64 \times 32 = 2 \text{ KiB} \leq 4 \text{ KiB} \quad \checkmark$$

es compleix (índex + desplaçament de bloc = 11 bits ≤ 12 bits de desplaçament de pàgina), de manera que la memòria cau no pateix aliàsing: l'índex selecciona la mateixa línia amb l'adreça lògica que amb la física.

Exemple 101: Solució: fallades de memòria cau i TLB per files vs. columnes

Enunciat: Problema 8.10.
Enters de 4 B. MC: 4 vies × 64 conjunts × 32 B/bloc = 8 KiB; un bloc conté $32/4 = 8$ enters. TLB de 32 entrades, pàgines de 8 KiB (2048 enters per pàgina; cada fila d'`A` ocupa 1024 enters = 4 KiB, és a dir, mitja pàgina). La matriu `A[128][1024]` ocupa 512 KiB = 64 pàgines = 16 384 blocs.
Adreça de l'element `A[i][j]`: byte $= (i \cdot 1024 + j) \cdot 4 = i \cdot 4096 + j \cdot 4$.

- Conjunt d'MC: $\left\lfloor \frac{i \cdot 4096 + j \cdot 4}{32} \right\rfloor \bmod 64 = (i \cdot 128 + \lfloor j/8 \rfloor) \bmod 64 = \lfloor j/8 \rfloor \bmod 64$ → **depèn només de** j.
- Pàgina: $\left\lfloor \frac{i \cdot 4096 + j \cdot 4}{8192} \right\rfloor = \lfloor i/2 \rfloor$ → **depèn només de** i.

Versió A (recorregut per files, accés seqüencial en memòria):

- MC: cada bloc de 8 enters es carrega un cop i s'aprofita 8 vegades. Fallades = 16 384 blocs = **16 384** (1 de cada 8 accessos).
- TLB: es recorren les 64 pàgines en ordre, una fallada per pàgina = **64**.

Versió B (recorregut per columnes, *stride* de 4096 B):

- MC: per a un j fix, els 128 elements `A[0..127][j]` són blocs diferents que mapen **tots al mateix conjunt** ($\lfloor j/8 \rfloor \bmod 64$, independent de i). Amb només 4 vies, els 128 blocs s'expulsen entre ells abans de poder-se reutilitzar a la iteració següent de j. **Cap reutilització**: totes les referències fallen = 128×1024 = **131 072**.
- TLB: per a cada j, el bucle intern recorre `A[0..127][j]`, que toca les pàgines $\lfloor i/2 \rfloor = 0, 0, 1, 1, \ldots, 63, 63$: **64 pàgines diferents**, una fallada cadascuna (la segona visita, i senar, encerta). Són 64 fallades per cada valor de j: 64×1024 = **65 536**.

	Fallades d'MC	Fallades de TLB
Versió A (per files)	16 384	64
Versió B (per columnes)	131 072	65 536

La versió B multiplica per 8 les fallades d'MC i per 1024 les de TLB. El recorregut per columnes no només destrueix la localitat espacial (cada accés toca un bloc i una pàgina nous), sinó que a més pateix un **conflicte de conjunt** sistemàtic: com que l'*stride* entre files (4096 B) és múltiple del nombre de conjunts per la mida de bloc, totes les files competeixen pel mateix conjunt de l'MC.

Tema 9: Excepcions i interrupcions

9.1 Excepcions i interrupcions

Exemple 102: Solució: conceptes bàsics d'excepcions i interrupcions

Enunciat: Problema 9.1.
a) Una **excepció** és un esdeveniment causat per l'execució d'una instrucció (origen intern): per exemple, un accés a memòria amb adreça no alineada o una instrucció il · legal. Una **interrupció** és un esdeveniment causat per un dispositiu extern (origen extern): per exemple, un teclat que notifica que s'ha premut una tecla.
b) En cas d'**excepció**, `mepc` apunta a la **instrucció causant** (la que s'ha interromput sense completar-se). En cas d'**interrupció**, `mepc` apunta a la **instrucció següent** a la instrucció que s'estava executant (la instrucció en curs ha finalitzat completament i el PC ja s'ha incrementat).
c) Quan el processador detecta una excepció o interrupció, executa automàticament, per maquinari (Secció 9.3.1):

1. Guarda el PC a `mepc`.
2. Escriu el codi de causa a `mcause` (bit 31 = 0 per a excepcions, bit 31 = 1 per a interrupcions).
3. Copia `MIE` a `MPIE` i posa `MIE = 0` (deshabilita les interrupcions).
4. Guarda el mode de privilegi actual a `MPP` i passa a mode M.
5. Salta a l'adreça de la rutina de servei d'excepcions (RSE) continguda a `mtvec`.

d) L'RSE ha de desar **tots** els registres generals (`x1`–`x31`) sense excepció. A diferència d'una subrutina normal, el programa interromput no ha cedit el control voluntàriament i no pot saber quins registres modificarà l'RSE. Les convencions ABI (registres salvats vs. temporals) no s'apliquen.

Exemple 103: Solució: RSE mínima en assemblador

Enunciat: Problema 9.3. *Vegeu l'estructura general de l'RSE a Secció 9.4.*

`RV32IZicsr`

```
rse:
        # a) Reservar espai i desar tots els registres (x1-x31, excepte sp=x2)
        addi    sp, sp, -120

        sw      ra,   0(sp)
        sw      t0,   4(sp)
        sw      t1,   8(sp)
        sw      t2,  12(sp)
        sw      s0,  16(sp)
        sw      s1,  20(sp)
        sw      a0,  24(sp)
        sw      a1,  28(sp)
        sw      a2,  32(sp)
```

```
        sw      a3,  36(sp)
        sw      a4,  40(sp)
        sw      a5,  44(sp)
        sw      a6,  48(sp)
        sw      a7,  52(sp)
        sw      s2,  56(sp)
        sw      s3,  60(sp)
        sw      s4,  64(sp)
        sw      s5,  68(sp)
        sw      s6,  72(sp)
        sw      s7,  76(sp)
        sw      s8,  80(sp)
        sw      s9,  84(sp)
        sw      s10, 88(sp)
        sw      s11, 92(sp)
        sw      t3,  96(sp)
        sw      t4, 100(sp)
        sw      t5, 104(sp)
        sw      t6, 108(sp)
        sw      gp, 112(sp)
        sw      tp, 116(sp)

        # b) Llegir mcause i comprovar si és ecall des de mode U (codi 8)
        csrr    t0, mcause
        li      t1, 8
        beq     t0, t1, handler_ecall

        # Causa desconeguda: salt al gestor d'error genèric
        j       handler_error

handler_ecall:
        # Tractament de la crida al sistema...
        # Incrementar mepc en 4 per continuar a la instrucció següent
        csrr    t0, mepc
        addi    t0, t0, 4
        csrw    mepc, t0
        j       rse_ret

handler_error:
        # Tractament d'error genèric...
        j       rse_ret

rse_ret:
        # c) Restaurar tots els registres
        lw      ra,   0(sp)
        lw      t0,   4(sp)
        lw      t1,   8(sp)
        lw      t2,  12(sp)
        lw      s0,  16(sp)
        lw      s1,  20(sp)
        lw      a0,  24(sp)
        lw      a1,  28(sp)
        lw      a2,  32(sp)
        lw      a3,  36(sp)
        lw      a4,  40(sp)
        lw      a5,  44(sp)
        lw      a6,  48(sp)
        lw      a7,  52(sp)
```

```
        lw      s2,  56(sp)
        lw      s3,  60(sp)
        lw      s4,  64(sp)
        lw      s5,  68(sp)
        lw      s6,  72(sp)
        lw      s7,  76(sp)
        lw      s8,  80(sp)
        lw      s9,  84(sp)
        lw      s10, 88(sp)
        lw      s11, 92(sp)
        lw      t3,  96(sp)
        lw      t4, 100(sp)
        lw      t5, 104(sp)
        lw      t6, 108(sp)
        lw      gp, 112(sp)
        lw      tp, 116(sp)

        addi    sp, sp, 120

        # Retornar: MIE←MPIE, mode←MPP, PC←mepc
        mret
```

Observacions:

- L'etiqueta `rse` ha de ser l'adreça que es configura a `mtvec` (camp `BASE`) durant la inicialització.
- El salt `j` és una pseudoinstrucció equivalent a `jal x0, offset`.

💡 Exemple 104: Solució: `mepc`, `mtvec` i ajust de retorn

Enunciat: Problema 9.4.
a) La instrucció `lw t0, 0(t1)` a `0x80001000` genera una excepció en executar-se (adreça no alineada). La instrucció s'avorta sense completar-se i el PC no s'incrementa. El maquinari guarda l'adreça de la instrucció causant:

$$\texttt{mepc} = \texttt{0x80001000}$$

b) La interrupció de temporitzador es detecta en **acabar** la instrucció `lw t0, 0(a0)` (el PC ja s'ha incrementat). El maquinari guarda l'adreça de la instrucció *següent*:

$$\texttt{mepc} = \texttt{0x80002008}$$

En retornar amb `mret`, el processador salta a `mepc`, és a dir, a `0x80002008`: la instrucció que seguia el `lw`.
c) En **mode directe** (`MODE = 0`), totes les excepcions i interrupcions salten a `BASE`. L'RSE llegeix `mcause` per identificar la causa i derivar el control al gestor específic.
En **mode vectoritzat** (`MODE = 1`), les *interrupcions* salten a `BASE + 4 × causa`; les *excepcions* sempre salten a `BASE`.
Per a una interrupció de temporitzador en mode M (codi de causa 7):

$$\text{adreça} = \texttt{BASE} + 4 \times 7 = \texttt{BASE} + 28$$

d) Sense l'increment, `mret` retornaria a la mateixa instrucció `ecall`, que causaria una nova excepció indefinidament. L'RSE ha d'incrementar `mepc` perquè l'`ecall` ja ha estat atesa i el programa ha de continuar a la instrucció *següent*:

`RV32IZicsr`

```
csrr    t0, mepc            # llegir l'adreça de retorn
addi    t0, t0, 4           # apuntar a la instrucció següent
csrw    mepc, t0            # actualitzar mepc
```

9.2 Crides al sistema

Exemple 105: Solució: crides al sistema bàsiques a RARS

Enunciat: Problema 9.5. *Els codis de servei de RARS són a RISC-V 9.13.*
a) Imprimir l'enter de `t0`:

RV32I

```
mv      a0, t0              # argument: enter a imprimir
li      a7, 1               # codi: print_int
ecall
```

b) Llegir un enter i deixar el resultat a `t1`:

RV32I

```
li      a7, 5               # codi: read_int
ecall                       # a0 ← enter llegit
mv      t1, a0              # moure el resultat a t1
```

c) Imprimir la cadena de caràcters amb adreça base a `t2`:

RV32I

```
mv      a0, t2              # argument: adreça del string
li      a7, 4               # codi: print_string
ecall
```

d) Finalitzar el programa:

RV32I

```
li      a7, 93              # codi: exit2
li      a0, 0               # codi de sortida: 0
ecall                       # no retorna
```

Nota: a EC la sortida es fa sempre amb `exit2`, no amb `exit` (codi 10) — vegeu RISC-V 2.19.

9.3 Entrada/sortida

Exemple 106: Solució: conceptes d'entrada/sortida

Enunciat: Problema 9.8.
a) En la **sincronització per enquesta** (*polling*), el processador consulta repetidament el registre d'estat del dispositiu fins que l'operació acaba. És simple, però desaprofita el processador durant l'espera. En la **sincronització per interrupcions**, el dispositiu notifica el processador quan ha acabat, i el processador pot fer feina útil entremig. És més eficient, però requereix configurar els permisos d'interrupció i escriure una RSE.
b) Per habilitar les interrupcions en RISC-V cal:

1. Habilitar el permís específic del tipus d'interrupció al registre `mie` (per exemple, `MEIE` = bit 11 per a interrupcions externes).
2. Habilitar el permís global `MIE` (bit 3 de `mstatus`).

`RV32IZicsr`

```
        # Habilitar interrupcions externes (MEIE = bit 11 de mie)
        li      t0, 0x800           # bit 11 (MEIE)
        csrrs   x0, mie, t0

        # Habilitar permís global (MIE = bit 3 de mstatus)
        li      t0, 0x8             # bit 3 (MIE)
        csrrs   x0, mstatus, t0
```

c) El bit `MIE` (bit 3 de `mstatus`) és el **permís global**: si val 0, cap interrupció no s'atén, independentment de la resta de bits. El bit `MTIE` (bit 7 de `mie`) és la **màscara específica** de les interrupcions de temporitzador en mode M: si val 0, les interrupcions de temporitzador queden inhibides fins i tot quan `MIE = 1`. Cal que tots dos valguin 1 (i el bit corresponent de `mip` estigui actiu) perquè una interrupció de temporitzador sigui atesa.

Exemple 107: Solució: RSE per a interrupcions de teclat

Enunciat: Problema 9.10.
a) Cal activar `MEIE` (bit 11 de `mie`) i `MIE` (bit 3 de `mstatus`):

`RV32IZicsr`

```
        # Habilitar interrupcions externes en mode M (MEIE = bit 11 de mie)
        li      t0, 0x800           # bit 11 (MEIE)
        csrrs   x0, mie, t0

        # Habilitar permís global d'interrupcions (MIE = bit 3 de mstatus)
        li      t0, 0x8             # bit 3 (MIE)
        csrrs   x0, mstatus, t0
```

b) RSE mínima per a la interrupció de teclat:

`RV32IZicsr`

```
.data
car:    .word   0
```

```
.text
rse_teclat:
        # Desar tots els registres generals (x1-x31, excepte sp=x2)
        addi    sp, sp, -120

        sw      ra,   0(sp)
        sw      t0,   4(sp)
        sw      t1,   8(sp)
        sw      t2,  12(sp)
        sw      s0,  16(sp)
        sw      s1,  20(sp)
        sw      a0,  24(sp)
        sw      a1,  28(sp)
        sw      a2,  32(sp)
        sw      a3,  36(sp)
        sw      a4,  40(sp)
        sw      a5,  44(sp)
        sw      a6,  48(sp)
        sw      a7,  52(sp)
        sw      s2,  56(sp)
        sw      s3,  60(sp)
        sw      s4,  64(sp)
        sw      s5,  68(sp)
        sw      s6,  72(sp)
        sw      s7,  76(sp)
        sw      s8,  80(sp)
        sw      s9,  84(sp)
        sw      s10, 88(sp)
        sw      s11, 92(sp)
        sw      t3,  96(sp)
        sw      t4, 100(sp)
        sw      t5, 104(sp)
        sw      t6, 108(sp)
        sw      gp, 112(sp)
        sw      tp, 116(sp)

        # Comprovar que és una interrupció externa (bit 31 de mcause = 1, codi = 11)
        csrr    t0, mcause
        li      t1, 0x8000000B      # bit31=1, codi=11 (interrupció externa mode M)
        bne     t0, t1, rse_ret_teclat

        # Comprovar bit MEIP de mip (bit 11): petició externa pendent
        csrr    t0, mip
        li      t1, 0x800           # bit 11 (MEIP)
        and     t0, t0, t1
        beqz    t0, rse_ret_teclat

        # Confirmar que la interrupció prové del teclat (registre d'estat, bit 0)
        li      t0, 0xF0001000      # adreça registre d'estat del teclat
        lw      t1, 0(t0)
        andi    t1, t1, 1           # bit 0: tecla disponible
        beqz    t1, rse_ret_teclat

        # Llegir el caràcter del registre de dades i desar-lo a 'car'
        li      t0, 0xF0001004      # adreça registre de dades del teclat
        lw      t1, 0(t0)           # llegir el caràcter
        la      t2, car
        sw      t1, 0(t2)           # desar a la variable global
```

```
rse_ret_teclat:
        # Restaurar tots els registres generals
        lw      ra,   0(sp)
        lw      t0,   4(sp)
        lw      t1,   8(sp)
        lw      t2,  12(sp)
        lw      s0,  16(sp)
        lw      s1,  20(sp)
        lw      a0,  24(sp)
        lw      a1,  28(sp)
        lw      a2,  32(sp)
        lw      a3,  36(sp)
        lw      a4,  40(sp)
        lw      a5,  44(sp)
        lw      a6,  48(sp)
        lw      a7,  52(sp)
        lw      s2,  56(sp)
        lw      s3,  60(sp)
        lw      s4,  64(sp)
        lw      s5,  68(sp)
        lw      s6,  72(sp)
        lw      s7,  76(sp)
        lw      s8,  80(sp)
        lw      s9,  84(sp)
        lw      s10, 88(sp)
        lw      s11, 92(sp)
        lw      t3,  96(sp)
        lw      t4, 100(sp)
        lw      t5, 104(sp)
        lw      t6, 108(sp)
        lw      gp, 112(sp)
        lw      tp, 116(sp)

        addi    sp, sp, 120
        mret
```

---

c) El codi de causa (`mcause`, codi 11) i el bit `MEIP` de `mip` indiquen que hi ha una **interrupció externa** pendent, però no quin dispositiu específic l'ha causada. Per identificar que la interrupció prové del teclat cal llegir directament el **registre d'estat del dispositiu** (`0xF0001000`, bit 0): és l'únic lloc on el controlador del teclat indica que té una dada disponible. En sistemes amb múltiples dispositius d'E/S, l'RSE ha d'interrogar cadascun per determinar quin ha generat la petició.

9.4 Fallada de pàgina

Exemple 108: Solució: fallada de pàgina i TLB gestionat per programari

Enunciat: Problema 9.11, apartats a–b.
a) Una fallada de pàgina escriu a `mcause` (RISC-V 9.4):

- En **lectura** de dada: `mcause = 13` (*load page fault*).
- En **escriptura**: `mcause = 15` (*store/AMO page fault*).

En tots dos casos, `mtval` conté l'**adreça virtual** que ha causat la fallada (Secció 9.7.2).
Cal recordar que, a RISC-V, la fallada de pàgina només es produeix quan la pàgina no és a la memòria física (bit V = 0 a la PTE): la fallada de TLB amb la pàgina present la resol el recorregut de la taula de pàgines (*page-table walker*) sense interrompre el programa (Essencial 9.9).
b) El flux que ha de seguir l'RSE (la rutina completa és a Secció 9.7.3.2):

1. Llegir `mtval` per obtenir l'adreça virtual que ha causat la fallada.
2. Extreure'n el VPN (els bits per sobre del desplaçament de pàgina) i localitzar la PTE corresponent a la taula de pàgines del procés actiu (a partir del registre de taula de pàgines; `satp` en mode S).
3. Si l'adreça no pertany a cap regió vàlida del procés, avortar el programa (*segmentation fault*).
4. Si la pàgina és vàlida però no present ($V = 0$): copiar-la del disc a un marc lliure (reemplaçant-ne un si cal, i escrivint-lo prèviament al disc si té $D = 1$) i actualitzar la PTE ($V \leftarrow 1$, $PPN \leftarrow$ marc assignat, $D \leftarrow 0$).
5. Executar `sfence.vma` perquè el TLB no conservi cap còpia obsoleta de la PTE i el recorregut de la taula de pàgines la torni a llegir.
6. No incrementar `mepc`: la instrucció causant s'ha de **reexecutar**; en la reexecució, el recorregut de la taula de pàgines trobarà la PTE vàlida i l'accés es completarà.

Part IV

Laboratori

Laboratori 1: Introducció al simulador RARS

Lliuraments

Cal lliurar les respostes als exercicis següents al campus virtual abans de l'inici de la sessió de laboratori:

Exercici	Nom i format del fitxer
Exercici 1.1	`s1_2_1.md`
Exercici 1.2	`s1_2_2.md`
Exercici 1.3	`s1_3_1.md`
Exercici 1.4	`s1_3_2.md`
Exercici 1.5	`s1_4_1.md`
Exercici 1.6	`s1_5_1.s` i `s1_5_1.md`
Exercici 1.7	`s1_5_2.s` i `s1_5_2.md`

Lectura prèvia

Per fer la sessió, cal haver instal · lat RARS tal com s'indica a la Presentació del curs.

També cal haver llegit i entès els continguts següents:

Concepte	On trobar-ho
Format del codi assemblador	EC 2.7
Esquelet d'un programa	RISC-V 2.19
Accessos a memòria (`lw`), desplaçaments (`slli`) i salts condicionals (`bge`, `blt`)	Secció 2.6.3.1, Secció 3.1.1, Secció 3.4.1
Pseudoinstruccions `la` i `li`, i l'expansió de `la` en `auipc` + `addi`	RISC-V 2.17, RISC-V 2.16, RISC-V 2.14

Essencial 1.1: Format del codi i esquelet

Es recomana seguir el **format del codi** (EC 2.7) i l'**esquelet** (RISC-V 2.19) en **tots** els programes lliurats al laboratori i a l'examen de laboratori, com fan els exemples. El format no s'avalua per si mateix.

1.1 El primer programa

El Codi 1.1 és un programa mínim executable a RARS: imprimeix un missatge per consola i en finalitza l'execució.

Codi 1.1: Programa mínim: escriu un missatge i acaba.

`hola.s`

```
.data
msg:    .asciz  "Hola, món!\n"

.text
        la      a0, msg         # a0 = adreça de msg
        li      a7, 4           # syscall 4: print_string
        ecall

        li      a7, 93          # syscall 93: exit2
        li      a0, 0           # codi de sortida: 0
        ecall
```

Per executar-lo a RARS:

1. Copieu el codi a l'editor de RARS i guardeu-lo com `hola.s`.
2. Assembleu amb *Run* → *Assemble* (o F3). Si hi ha errors, RARS els mostra al panell de missatges amb el número de línia.
3. Executeu amb *Run* → *Go* (o F5). El missatge ha d'aparèixer al panell de consola.

Essencial 1.2: Crides al sistema de RARS

Una **crida al sistema** és una petició de servei al sistema operatiu (o, en el cas de RARS, al simulador). Es fa carregant el codi de servei a `a7` i els paràmetres a `a0`–`a2`, i executant `ecall`. Les crides al sistema s'estudien en detall a Secció 9.5; per ara n'hi ha prou amb usar-les com a patró:

Codi de servei (`a7`)	Servei	Paràmetre (`a0`)	Retorn (`a0`)
1	`print_int`	enter a imprimir	—
4	`print_string`	adreça de la cadena	—
93	`exit2`	codi de sortida	—

1.2 Exploració del simulador

La secció treballa sobre el programa `suma.s`, que calcula la suma dels elements d'un vector d'enters i deixa el resultat al registre `t0`.

`suma.s`

```
.data
V:      .word   -1, -2, -3, -4, -5, -6, -7, -8, -9, -10

.text
        li      t0, 0           # t0 = suma = 0
        li      t1, 0           # t1 = i = 0
        li      t2, 10          # t2 = N = 10
        la      t3, V           # t3 = adreça de V

bucle:
        bge     t1, t2, fi      # si i >= N, surt
        slli    t4, t1, 2       # t4 = 4*i
```

```
        add     t4, t3, t4      # t4 = @V[i]
        lw      t4, 0(t4)       # t4 = V[i]
        add     t0, t0, t4      # suma += V[i]
        addi    t1, t1, 1       # i++
        j       bucle

fi:
        li      a7, 93
        li      a0, 0
        ecall
```

Un cop assemblat el programa (F3), la pestanya *Execute* mostra els panells que s'usaran durant la simulació:

- **Panell de codi desassemblat** (*Text Segment*, esquerra): mostra les instruccions en memòria, una per fila, amb l'adreça, la codificació en hexadecimal i el mnemònic. La instrucció que s'executarà a continuació apareix ressaltada.
- **Panell de registres** (*Registers*, dreta): mostra el contingut dels 32 registres de propòsit general i el PC. Els registres modificats a l'últim pas apareixen ressaltats.
- **Panell de memòria de dades** (*Data Segment*, centre-baix): mostra el contingut del segment `.data` en format de paraules hexadecimals.
- **Panell de consola** (*Run I/O*, baix): mostra la sortida de les crides al sistema i permet introduir dades.

Exercici 1.1 (`s1_2_1.md`)**.** Assembleu `suma.s` i responeu les preguntes següents:

a) On es mostra el valor del comptador de programa (*program counter*, PC)?
b) RARS no mostra el nombre d'instruccions executades als panells principals. Quina eina de RARS permet comptar-les? Indiqueu el camí de menús per obrir-la i el pas necessari perquè comenci a comptar.
c) El panell de codi mostra, per a cada instrucció, tres columnes de valors (a més de la casella d'activació de punt de ruptura i la columna de codi font). Quina informació conté cadascuna?

Solució 1.1.

a) Al panell de registres (*Registers*), a la fila `pc`.
b) *Tools* → *Instruction Counter*. Cal obrir-la **abans** d'executar el programa i prémer *Connect to Program* perquè comenci a comptar; si s'executa el programa abans de connectar l'eina, el recompte no es registrarà.
c) *Address* (adreça de memòria de la instrucció), *Code* (codificació de la instrucció en hexadecimal) i *Basic* (la instrucció real, amb els registres identificats amb `x0`–`x31`).

Exercici 1.2 (`s1_2_2.md`)**.** Algunes pseudoinstruccions s'expandeixen en més d'una instrucció real. Treballeu sobre `suma.s` ja assemblat.

a) Localitzeu la pseudoinstrucció `la t3, V` al panell de codi desassemblat. Quines instruccions reals ha generat? Copieu-les exactament tal com apareixen al panell.
b) Per quina raó `la` necessita dues instruccions reals en lloc d'una?
c) La pseudoinstrucció `li t2, 10`, en canvi, genera una sola instrucció. Per quina raó en aquest cas n'hi ha prou amb una? Hauria passat el mateix amb `li t2, 100000`?

Indicació: els immediats de les instruccions tipus I tenen 12 bits amb signe (rang $-2048..2047$).

Solució 1.2.

a) El panell mostra `auipc x28,0x0000fc10` seguida d'`addi x28,x28,0xffffffff4`, és a dir, `auipc t3, 0xfc10` i `addi t3, t3, -12` amb els noms ABI (els valors exactes depenen de l'adreça de `V` i de la posició de `la` al programa; l'estructura auipc+addi és sempre la mateixa).
b) Perquè una adreça de memòria ocupa 32 bits, i cap instrucció tipus I té un immediat de 32 bits: `auipc` aporta els 20 bits alts (relatius al PC) i `addi` hi suma els 12 bits baixos amb signe.
c) `10` cap dins el rang $[-2048, 2047]$ d'un immediat de 12 bits amb signe, per tant, una sola `addi` n'hi ha prou. Amb `li t2, 100000` **no** passaria el mateix: 100000 no cap en 12 bits, de manera que RARS generaria `lui t2, 0x18` seguida d'`addi t2, t2, 1696` (dues instruccions), igual que amb `la`.

1.3 Execució i inspecció

RARS ofereix tres modes d'execució:

- **Execució contínua** (F5): executa el programa fins al final (o fins al primer punt de ruptura).
- **Execució pas a pas** (F7): executa una instrucció i s'atura. Útil per seguir el flux instrucció per instrucció.
- **Reinici** (F12): torna el programa a l'estat inicial sense tornar a assemblar.

El programa `hola.s` de Secció 1.1 no té bucles: el nombre d'instruccions és fix i petit. És el programa que s'usarà per al recompte manual.

Exercici 1.3 (`s1_3_1.md`)**.** Treballeu sobre el programa `hola.s`.

a) Obriu *Tools* → *Instruction Counter*, connecteu-la al programa (*Connect to Program*) i executeu amb F5. Quin nombre d'instruccions executades reporta l'eina?
b) Feu el recompte manual: llisteu totes les instruccions que s'executen, en ordre, indicant per a cada una si és una instrucció real o l'expansió d'una pseudoinstrucció. Quantes instruccions reals s'executen en total?
c) El valor del recompte manual coincideix amb el que reporta l'Instruction Counter? Si hi ha diferència, expliqueu-ne la causa.

Indicació: la pseudoinstrucció `la` s'expandeix en dues instruccions reals. Useu el panell de codi desassemblat per veure exactament quines instruccions s'executen.

Solució 1.3.

a) 7 instruccions.
b) `la a0, msg` (auipc + addi, 2 instruccions reals), `li a7, 4` (addi, 1), `ecall` (1), `li a7, 93` (addi, 1), `li a0, 0` (addi, 1), `ecall` (1). Total: **7 instruccions reals**.
c) Sí, coincideixen (7 = 7). Tot i que el codi font té 6 instruccions, el recompte manual dona 7 perquè `la` s'expandeix en dues instruccions reals.

Exercici 1.4 (`s1_3_2.md`)**.** Assembleu `suma.s` i observeu el panell de memòria de dades.

a) En quina adreça de memòria comença el vector `V`?
b) En quina adreça exacta es troba `V[3]`? Quin és el seu contingut en hexadecimal?
c) `V[0]` conté el valor `-1`. Quin valor hexadecimal té al panell de memòria? Expliqueu com s'obté aquesta representació.
d) El panell mostra cada paraula com un valor hexadecimal de 32 bits. Quin ordre dels bytes (*endianness*) usa RV32I?

Solució 1.4.

a) `0x10010000` (adreça per defecte del segment `.data` a RARS).
b) `0x1001000C` (= 0x10010000 + 3 · 4); conté `0xFFFFFFFC`.
c) `0xFFFFFFFF`. S'obté aplicant a +1 la regla del canvi de signe en Ca2 (Secció 1.5.1): es complementen tots els bits d'1 (`0x00000001` → `0xFFFFFFFE`) i se suma 1; o, de manera equivalent, $2^{32} - 1 =$ `0xFFFFFFFF`.
d) RV32I és **little-endian**: el byte de menys pes s'emmagatzema a l'adreça més baixa.

1.4 Punts de ruptura

Un **punt de ruptura** (*breakpoint*) és una marca sobre una instrucció que fa que l'execució contínua (F5) s'aturi just abans d'executar-la. S'activa i es desactiva fent clic a la casella de l'esquerra del panell de codi desassemblat, a la fila de la instrucció triada.

Els punts de ruptura permeten inspeccionar l'estat del programa en un punt clau sense haver d'executar-lo pas a pas des del principi. Quan el programa s'atura, es pot continuar amb F5 (fins al següent punt de ruptura) o amb F7 (pas a pas).

Exercici 1.5 (`s1_4_1.md`)**.** Treballeu sobre `suma.s`.

a) Afegiu un punt de ruptura a la instrucció `add t0, t0, t4` (la que acumula la suma). Executeu el programa amb F5. En quina iteració s'atura? Quins són els valors de `t0` i `t4` en aquest moment? Tingueu en compte que l'execució s'atura just **abans** d'executar la instrucció marcada.
b) Premeu F5 repetidament fins al final. Anoteu el valor de `t4` en cada aturada. Verifiqueu que els valors corresponen als elements del vector `V` en ordre.
c) Quin és el valor final de `t0` un cop ha acabat l'execució? Era el resultat esperat? Justifiqueu-lo.

Solució 1.5.

a) S'atura a la primera iteració, amb `t0 = 0` i `t4 = -1`.
b) Els valors de `t4` en cada aturada successiva són -1, -2, -3, -4, -5, -6, -7, -8, -9, -10: coincideixen exactament amb els elements de `V` en ordre.
c) `t0 = -55`. És el resultat esperat: $-1-2-3-4-5-6-7-8-9-10 = -55$.

1.5 Depuració

Quan un programa assembla sense errors, però produeix un resultat incorrecte, cal **depurar-lo**: executar-lo pas a pas o amb punts de ruptura per identificar en quina instrucció el comportament no és l'esperat.

El procediment general és:

1. Determineu el resultat esperat abans d'executar el programa.
2. Identifiqueu un punt de control (per exemple, la instrucció d'acumulació d'un bucle) i afegiu-hi un punt de ruptura.
3. En cada aturada, comproveu que els valors dels registres rellevants coincideixen amb el que s'esperava.
4. Quan trobeu una discrepància, retrocediu un pas per identificar la instrucció causant. RARS ofereix *Run* → *Backstep* (F8) per desfer l'última instrucció executada sense haver de reiniciar.

Exercici 1.6 (`s1_5_1.s` i `s1_5_1.md`)**.** El programa `s1_5_1.s` és una versió **errònia** del programa `suma.s`. Conté un únic error que fa que el resultat final de `t0` no sigui `-55`.

`s1_5_1.s`

```
.data
V:      .word   -1, -2, -3, -4, -5, -6, -7, -8, -9, -10

.text
        li      t0, 0
        li      t1, 9
        li      t2, 10
        la      t3, V

bucle:
        bge     t1, t2, fi
        slli    t4, t1, 2
        add     t4, t3, t4
        lw      t4, 0(t4)
        add     t0, t0, t4
        addi    t1, t1, 1
        j       bucle

fi:
        li      a7, 93
        li      a0, 0
        ecall
```

a) Sense executar el programa, analitzeu el codi i preveieu quin serà el resultat incorrecte de `t0` i per quina raó.
b) Executeu el programa pas a pas per confirmar la vostra predicció. En quina iteració s'observa per primera vegada un comportament incorrecte?
c) Corregiu l'error (canvieu una única línia) i verifiqueu que el resultat és `-55`. Lliureu la versió corregida com `s1_5_1.s`.
d) Al fitxer `s1_5_1.md`, descriviu: quina és la instrucció errònia, quin valor incorrecte produïa i com ho heu detectat.

Solució 1.6.

a) L'error és `li t1, 9` (hauria de ser `li t1, 0`): l'índex `i` comença a 9 en lloc de 0, per tant, el bucle només suma `V[9] = -10`. El resultat incorrecte previst és `t0 = -10`.
b) El comportament incorrecte s'observa ja a la **primera** iteració: `t1` val 9 des de l'inici i a `t4` es llegeix `V[9] = -10` en lloc de `V[0] = -1`.
c) Cal canviar `li t1, 9` per `li t1, 0`. Amb la correcció, `t0` acaba valent `-55`.
d) La instrucció errònia és `li t1, 9`; produïa `t0 = -10` perquè només s'acumulava l'últim element del vector; es detecta comparant l'índex inicial esperat (0) amb el valor carregat a `t1` a l'inici de l'execució pas a pas.

Exercici 1.7 (`s1_5_2.s` i `s1_5_2.md`)**.** El programa `s1_5_2.s` calcula la **mitjana entera** dels elements del vector `V` (és a dir, la divisió entera de la suma entre el nombre d'elements). La divisió es fa per mitjà d'un segon bucle que resta repetidament el divisor del dividend mentre el dividend no sigui inferior al divisor.

El programa conté un únic error. El resultat correcte del quocient (registre `t5`) hauria de ser `5` (ja que $\lfloor 55/10 \rfloor = 5$, i la suma dels elements de `V = {1, 2, ..., 10}` és 55).

`s1_5_2.s`

```
.data
V:      .word   1, 2, 3, 4, 5, 6, 7, 8, 9, 10

.text
        li      t0, 0           # t0 = suma = 0
        li      t1, 0           # t1 = i = 0
        li      t2, 10          # t2 = N = 10
        la      t3, V

bucle:
        bge     t1, t2, fi
        slli    t4, t1, 2
        add     t4, t3, t4
        lw      t4, 0(t4)
        add     t0, t0, t4
        addi    t1, t1, 1
        j       bucle

fi:
        li      t5, 0           # t5 = quocient = 0
        mv      t6, t2          # t6 = divisor = N

bucle2:
```

```
        blt     t0, t6, fi2     # si suma < divisor, acaba
        sub     t0, t6, t0      # suma -= divisor
        addi    t5, t5, 1
        j       bucle2

fi2:
        li      a7, 93
        li      a0, 0
        ecall
```

---

a) Sense executar el programa, llegiu el codi del segon bucle (`bucle2`) i identifiqueu l'error. Què hauria de fer la instrucció errònia?
b) Afegiu un punt de ruptura a la primera instrucció de `bucle2`. Executeu fins al punt de ruptura amb F5 i comproveu que `t0` és `55` i `t6` és `10`.
c) Continueu amb F7 pas a pas. Observeu el valor de `t0` després de la instrucció `sub`. Es confirma el diagnòstic del punt a)?
d) Corregiu l'error i verifiqueu que `t5` val `5`. Lliureu la versió corregida com `s1_5_2.s`.
e) Al fitxer `s1_5_2.md`, descriviu: on heu col · locat el punt de ruptura i per quina raó, quina és la instrucció errònia i com l'heu corregit.

Solució 1.7.

a) L'error és `sub t0, t6, t0` (els operands estan invertits: hauria de ser `sub t0, t0, t6`). La instrucció hauria de calcular `t0 = t0 - t6` (suma restant el divisor), no `t6 - t0`.
b) A l'aturada, `t0 = 55` (suma de `V = {1, ..., 10}`) i `t6 = 10` (còpia de `N`).
c) Després del `sub` erroni, `t0 = 10 - 55 = -45`. Això confirma el diagnòstic: en comptes de restar el divisor de la suma ($55 - 10 = 45$), es resta la suma del divisor, produint un valor negatiu que fa sortir del bucle massa aviat (amb `t5 = 1` en lloc de 5).
d) Amb `sub t0, t0, t6`, el bucle itera 5 vegades ($55 \to 45 \to 35 \to 25 \to 15 \to 5$) i `t5` acaba valent `5`.
e) Exemple de resposta: el punt de ruptura es col · loca a `blt t0, t6, fi2` (primera instrucció de `bucle2`) perquè és el punt on es pot comprovar l'estat inicial de `t0` i `t6` abans de cap iteració; la instrucció errònia és `sub t0, t6, t0`, amb els operands invertits; es corregeix escrivint `sub t0, t0, t6`.

Laboratori 2: Assemblador RISC-V i tipus bàsics de dades

Lliuraments

Cal lliurar les respostes als exercicis següents al campus virtual abans de l'inici de la sessió de laboratori:

Exercici	Nom i format del fitxer
Exercici 2.1	`s2_1_1.s`
Exercici 2.2	`s2_1_2.md`
Exercici 2.3	`s2_1_3.md`
Exercici 2.4	`s2_3_1.md`
Exercici 2.5	`s2_3_2.s`
Exercici 2.6	`s2_4_1.md` i `s2_4_1.s`
Exercici 2.7	`s2_4_2.s`
Exercici 2.8	`s2_5_1.s`

Lectura prèvia

Abans de fer la sessió, cal haver llegit i entès els continguts següents:

Concepte	On trobar-ho
Mida dels tipus de dades en C i RV32I	EC 2.8, EC 2.9
Directives de declaració i inicialització de variables globals	RISC-V 2.22
Model de memòria i adreces	Secció 2.5
Ordre dels bytes (*little-endian*)	Secció 2.5.1
Restriccions d'alineació	RISC-V 2.21
Directiva `.align`	RISC-V 2.31
Instruccions d'accés a memòria (`lw`, `lh`, `lhu`, `lb`, `lbu`, `sw`, `sh`, `sb`)	RISC-V 2.29
Punters: declaració, `&`, `*`, aritmètica	Secció 2.7.2, Secció 2.7.3, Secció 2.7.5, Secció 2.7.6
Vectors: declaració, indexació, relació amb punters	Secció 2.8, Secció 2.8.3, Secció 2.8.4
Caràcters i cadenes de caràcters	Secció 2.9
Pseudoinstruccions `la` i `li`	RISC-V 2.17, RISC-V 2.16

2.1 Representació en memòria

Els tipus de dades bàsics del llenguatge C tenen una correspondència directa amb les directives de declaració de variables globals en RV32I. Les taules de mides i directives són a T2 (EC 2.8, EC 2.9, RISC-V 2.22).

Quan es declaren diverses variables globals consecutivament, l'assemblador insereix automàticament bytes de **farciment** (*padding*) entre elles per respectar les restriccions d'alineació (Secció 2.5.2), tret de les de `.dword`, que RARS només alinea a 4 bytes (RISC-V 2.22). A més, els valors s'emmagatzemen en ordre *little-endian* (Secció 2.5.1): el byte de menys pes s'emmagatzema a l'adreça més baixa.

Exercici 2.1 (`s2_1_1.s`)**.** Traduïu a RV32I la declaració de variables globals en C del Codi 2.1:

Codi 2.1: Variables globals de tipus diferents.

C

```c
char      aa = -5;
short     bb = -344;
long long cc = -3;
unsigned char dd = 0xA0;
int       ee = 5799;
short     ff = -1;
```

Afegiu-hi l'esquelet mínim del programa (RISC-V 2.19).

Solució 2.1.

s2_1_1.s

```
.data
aa:     .byte   -5
bb:     .half   -344
        .align 3          # cc és long long: alineació a 8
cc:     .dword  -3
dd:     .byte   0xA0
ee:     .word   5799
ff:     .half   -1

.text
        # Sortir del programa
        li      a7, 93
        li      a0, 0
        ecall
```

Exercici 2.2 (`s2_1_2.md`)**.** Sabent que les dades globals s'emmagatzemen a partir de l'adreça `0x10010000`, ompliu la taula de distribució de variables:

Variable	Tipus	Mida [B]	Alineació [B]	Adreça	Farciment [B]
`aa`					
`bb`					
`cc`					
`dd`					
`ee`					
`ff`					

Ara escriviu el contingut de memòria byte a byte, en ordre *little-endian*. Indiqueu cada etiqueta a la posició que li correspon i marqueu amb `--` els bytes de farciment. Deixeu en blanc els bytes lliures (no inicialitzats).

Etiqueta	Adreça	Contingut
	`0x10010000`	
	`0x10010001`	
	`0x10010002`	
	`0x10010003`	
	`0x10010004`	
	`0x10010005`	
	`0x10010006`	
	`0x10010007`	
	`0x10010008`	
	`0x10010009`	
	`0x1001000A`	
	`0x1001000B`	
	`0x1001000C`	
	`0x1001000D`	
	`0x1001000E`	
	`0x1001000F`	

Etiqueta	Adreça	Contingut
	`0x10010010`	
	`0x10010011`	
	`0x10010012`	
	`0x10010013`	
	`0x10010014`	
	`0x10010015`	
	`0x10010016`	
	`0x10010017`	
	`0x10010018`	
	`0x10010019`	
	`0x1001001A`	
	`0x1001001B`	
	`0x1001001C`	
	`0x1001001D`	
	`0x1001001E`	
	`0x1001001F`	

Solució 2.2. `cc` és un `long long`, que l'ABI alinea a 8 bytes (RISC-V 2.21). A RARS, la directiva `.dword` només l'alinea a 4 (RISC-V 2.22): per això la solució de l'Exercici 2.1 hi posa `.align 3` al davant.

Variable	Tipus	Mida [B]	Alineació [B]	Adreça	Farciment [B]
`aa`	`char`	1	1	`0x10010000`	
`bb`	`short`	2	2	`0x10010002`	1
`cc`	`long long`	8	8	`0x10010008`	4
`dd`	`unsigned char`	1	1	`0x10010010`	
`ee`	`int`	4	4	`0x10010014`	3
`ff`	`short`	2	2	`0x10010018`	

Etiqueta	Adreça	Contingut
`aa`	`0x10010000`	`0xFB`
	`0x10010001`	`--`
`bb`	`0x10010002`	`0xA8`
	`0x10010003`	`0xFE`
	`0x10010004`	`--`
	`0x10010005`	`--`
	`0x10010006`	`--`
	`0x10010007`	`--`
`cc`	`0x10010008`	`0xFD`
	`0x10010009`	`0xFF`
	`0x1001000A`	`0xFF`
	`0x1001000B`	`0xFF`
	`0x1001000C`	`0xFF`
	`0x1001000D`	`0xFF`
	`0x1001000E`	`0xFF`
	`0x1001000F`	`0xFF`

Etiqueta	Adreça	Contingut
`dd`	`0x10010010`	`0xA0`
	`0x10010011`	`--`
	`0x10010012`	`--`
	`0x10010013`	`--`
`ee`	`0x10010014`	`0xA7`
	`0x10010015`	`0x16`
	`0x10010016`	`0x00`
	`0x10010017`	`0x00`
`ff`	`0x10010018`	`0xFF`
	`0x10010019`	`0xFF`
	`0x1001001A`	
	`0x1001001B`	
	`0x1001001C`	
	`0x1001001D`	
	`0x1001001E`	
	`0x1001001F`	

Essencial 2.1: RARS — Vista de memòria

RARS mostra el contingut de la memòria agrupat en **paraules de 4 bytes**. Cada fila mostra 8 paraules consecutives (columnes `+0` fins a `+1c`) a partir d'una adreça alineada a 32 bytes (múltiple de `0x20`). Dins de cada paraula, el byte de major adreça apareix a l'esquerra i el de menor adreça a la dreta. Això és contrari a l'ordre real d'emmagatzematge (*little-endian*) i sovint és font de confusió.
Per exemple, la paraula a l'adreça `0x10010000` conté els bytes (de menor a major adreça):

`+0`	`+1`	`+2`	`+3`
`0xFB`	`--`	`0xA8`	`0xFE`

i RARS el mostra com `0xFEA800FB`.

Exercici 2.3 (`s2_1_3.md`)**.** Després d'assemblar el programa de l'Exercici 2.1, RARS mostra la vista de memòria del Codi 2.2:

Codi 2.2: Vista de memòria de RARS.

```
Address      Value (+0)  Value (+4)  Value (+8)  Value (+c)  Value (+10) Value (+14) Value (+18)
 ↪  Value (+1c)
0x10010000   0xfea800fb  0x00000000  0xfffffffd  0xffffffff  0x000000a0  0x000016a7  0x0000ffff
 ↪  0x00000000
```

Per a cada paraula, identifiqueu a quina variable correspon cada byte i expliqueu per què apareix en aquest ordre.

Solució 2.3. El byte de menor adreça ocupa la posició de la dreta; el de major adreça, la de l'esquerra:

Adreça	Valor RARS	+0	+1	+2	+3	Variable(s)
0x10010000	0xfea800fb	0xFB	--	0xA8	0xFE	aa, farciment, bb
0x10010004	0x00000000	--	--	--	--	farciment (.align 3)
0x10010008	0xfffffffd	0xFD	0xFF	0xFF	0xFF	cc (bytes 0–3)
0x1001000C	0xffffffff	0xFF	0xFF	0xFF	0xFF	cc (bytes 4–7)
0x10010010	0x000000a0	0xA0	--	--	--	dd, farciment
0x10010014	0x000016a7	0xA7	0x16	0x00	0x00	ee
0x10010018	0x0000ffff	0xFF	0xFF			ff
0x1001001C	0x00000000					

2.2 Pseudoinstruccions `la` i `li`

Els programes d'aquesta sessió operen amb variables globals emmagatzemades en memòria. Com que l'únic mode d'adreçament per accedir-hi en RV32I és el mode base + desplaçament (Secció 2.6.3), abans de cada accés cal carregar l'adreça de la variable en un registre. Per fer-ho s'usen dues pseudoinstruccions (Secció 2.3.5.2), que l'assemblador expandeix en una o dues instruccions reals en el moment d'assemblar:

- **`la rd, etiqueta`** (***load address***, RISC-V 2.17) carrega a `rd` l'**adreça** associada a `etiqueta`. Per exemple, `la t0, vec` deixa a `t0` l'adreça de `vec`.
- **`li rd, valor`** (***load immediate***, RISC-V 2.16) carrega a `rd` un **valor immediat** de fins a 32 bits. Per exemple, `li t1, 10`.

> Essencial 2.2: RARS — `la` amb desplaçament
>
> A la teoria i als exàmens d'**EC** s'admet la sintaxi `la rd, etiqueta+desplaçament`, però **RARS no la suporta** (EC 2.6): al laboratori cal substituir-la sempre per la parella `la` + `addi`.

2.3 Punters

Un **punter** és una variable que conté l'adreça de memòria d'una altra variable. En RV32I, un punter ocupa sempre 4 bytes (una paraula), ja que les adreces són de 32 bits (RISC-V 2.30). La pseudoinstrucció `la` carrega l'adreça d'una variable en un registre (RISC-V 2.17), i les instruccions `lw`, `lh`, `lhu`, `lb`, `lbu`, `sw`, `sh` i `sb` permeten accedir al valor apuntat (RISC-V 2.29).

Exercici 2.4 (`s2_3_1.md`)**.** Donada la declaració del Codi 2.3, on `pdada` és un punter inicialitzat amb l'adreça de `dada`:

Codi 2.3: Variable `dada` i punter `pdada`.

`RV32I`

```
.data
dada:   .half   3
pdada:  .word   dada
```

Sabent que les variables s'emmagatzemen a partir de `0x10010000`, ompliu els blancs de les vinyetes i calculeu el valor de cada expressió C:

- `dada` és un _____ (___ bytes, alineació ___), s'emmagatzema a `0x1001000__`.
- `pdada` és un _____ (___ bytes, alineació **), s'emmagatzema a `0x1001000__` (** bytes de farciment).

Expressió C	Valor (hex)
`&dada`	
`&pdada`	
`dada`	
`pdada`	
`*pdada`	

Comproveu que, en assemblar la declaració a RARS, la paraula a `0x10010000` mostra `0x00000003` i la paraula a `0x10010004`, `0x10010000`.

Solució 2.4.

- `dada` és un `short` (2 bytes, alineació 2), s'emmagatzema a `0x10010000`.
- `pdada` és un punter (4 bytes, alineació 4), s'emmagatzema a `0x10010004` (2 bytes de farciment).

Expressió C	Valor (hex)	Explicació
`&dada`	`0x10010000`	Adreça de `dada`
`&pdada`	`0x10010004`	Adreça de `pdada`
`dada`	`0x0003`	Contingut de `dada` (un `short`, 2 bytes)
`pdada`	`0x10010000`	Contingut de `pdada` (adreça de `dada`)
`*pdada`	`0x0003`	Contingut de l'adreça apuntada per `pdada` (un `short`)

En llegir `dada` (o `*pdada`) en un registre amb `lh`, el valor s'estén en signe a 32 bits: `0x00000003`.

> **Comprovació pràctica**
>
> No cal executar el programa: n'hi ha prou d'assemblar-lo i obrir la vista del segment de dades, on es veuen el `short dada` seguit de 2 bytes de farciment i, a continuació, el punter `pdada`.

Exercici 2.5 (`s2_3_2.s`)**.** Traduïu a RV32I el programa C del Codi 2.4. La variable temporal `temp` s'emmagatzema al registre `s0`:

Codi 2.4: Programa amb un punter al vector `A`.

`C`

```c
int A[3] = {3, 5, 7};
int *punter = 0;

void main() {
    int temp;
    punter = &A[2];
    temp = *punter + 2;
    temp = *(punter - 2) + temp;
    A[1] = temp;
}
```

Comproveu que, en acabar l'execució, `s0` val `0x0000000C` (12), que `A[1]` (`0x10010004`) conté `0x0000000C` i que `punter` (`0x1001000C`) conté `0x10010008`, l'adreça d'`A[2]`.

Solució 2.5.

`s2_3_2.s`

```
.data
A:      .word   3, 5, 7
punter: .word   0

.text
        # punter = &A[2]
        la      t0, A
        addi    t0, t0, 8   # &A[2] = &A + 2*4
        la      t1, punter
        sw      t0, 0(t1)   # punter = &A[2]

        # temp = *punter + 2
        lw      s0, 0(t0)   # s0 = *punter = A[2] = 7
        addi    s0, s0, 2   # s0 = 9

        # temp = *(punter - 2) + temp
        addi    t0, t0, -8  # punter - 2 = &A[0]
        lw      t2, 0(t0)   # t2 = A[0] = 3
        add     s0, t2, s0  # s0 = 3 + 9 = 12

        # A[1] = temp
        la      t0, A
        sw      s0, 4(t0)   # A[1] = 12

        # Sortir del programa
        li      a7, 93
        li      a0, 0
        ecall
```

> 💡 Comprovació pràctica
>
> El resultat es reparteix entre dues vistes de RARS: `s0` a la finestra de registres i els valors d'`A[1]` i `punter` al segment de dades. Avançant pas a pas es pot seguir com `punter` s'escriu abans que `A[1]`.

2.4 Vectors

Un **vector** és una seqüència d'elements del mateix tipus emmagatzemats consecutivament en memòria. L'adreça de l'element `vec[i]` es calcula a partir de l'adreça base del vector i la mida de cada element, segons l'Equació 2.1 (Secció 2.8, Secció 2.8.3):

$$@\texttt{vec[i]} = @\texttt{vec[0]} + i \cdot T$$

on T és la mida en bytes de cada element. En RV32I, l'accés indexat requereix calcular aquesta adreça explícitament, ja que no hi ha mode d'adreçament indexat (Secció 2.6.3).

Exercici 2.6 (`s2_4_1.md` i `s2_4_1.s`)**.** Donat el vector global en C del Codi 2.5:

Codi 2.5: Vector global `vec` de deu enters.

C

```c
int vec[10] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
```

1. Escriviu la fórmula per calcular l'adreça de `vec[i]` en funció de l'adreça base de `vec` i de l'índex `i`.
2. Escriviu un fragment de codi RV32I, amb la declaració corresponent, que copiï el valor de `vec[i]` al registre `s1`, suposant que `i` es troba al registre `s2`.

Comproveu que, afegint `li s2, 7` abans del fragment i executant-lo a RARS, `s1` val `0x00000002` (= `vec[7]`).

Solució 2.6. **1. Fórmula d'accés**:

$$@\texttt{vec[i]} = @\texttt{vec[0]} + i \cdot 4$$

(o, equivalentment, $@\texttt{vec} + i \cdot 4$, ja que `vec` és `&vec[0]`, Secció 2.8.4)

2. Declaració en RV32I i fragment de codi:

`s2_4_1.s`

```
.data
vec:    .word   9, 8, 7, 6, 5, 4, 3, 2, 1, 0

.text
        la      t0, vec     # t0 = @vec
        slli    t1, s2, 2   # t1 = i * 4
        add     t0, t0, t1  # t0 = @vec[i]
        lw      s1, 0(t0)   # s1 = vec[i]
```

Exercici 2.7 (`s2_4_2.s`)**.** Traduïu a RV32I el programa C del Codi 2.6, que calcula els 10 primers termes de la sèrie de Fibonacci i els escriu al vector `fib`. La variable `i` s'emmagatzema al registre `s0`:

Codi 2.6: Programa que calcula els deu primers termes de la sèrie de Fibonacci.

`C`

```
int fib[10];

void main() {
    int i = 2;
    fib[0] = 0;
    fib[1] = 1;
    while (i < 10) {
        fib[i] = fib[i-1] + fib[i-2];
        i++;
    }
}
```

Comproveu que, en acabar l'execució, el segment de dades conté els 10 termes: `0x0`, `0x1`, `0x1`, `0x2`, `0x3`, `0x5`, `0x8`, `0xD`, `0x15`, `0x22`.

Solució 2.7.

`s2_4_2.s`

```
.data
fib:    .space  40          # 10 enters * 4 bytes

.text
        # fib[0] = 0
        la      t0, fib
        sw      zero, 0(t0) # fib[0] = 0

        # fib[1] = 1
        li      t1, 1
        sw      t1, 4(t0)   # fib[1] = 1

        # i = 2
        li      s0, 2
```

```
while:
        # if i >= 10, sortir
        li      t1, 10
        bge     s0, t1, fi

        # t1 = fib[i-1]
        la      t0, fib
        slli    t2, s0, 2   # t2 = i * 4
        add     t3, t0, t2  # t3 = @fib[i]
        lw      t1, -4(t3)  # t1 = fib[i-1]

        # t2 = fib[i-2]
        lw      t2, -8(t3)  # t2 = fib[i-2]

        # fib[i] = fib[i-1] + fib[i-2]
        add     t1, t1, t2
        sw      t1, 0(t3)   # fib[i] = t1

        # i++
        addi    s0, s0, 1
        j       while

fi:
        # Sortir del programa
        li      a7, 93
        li      a0, 0
        ecall
```

2.5 Cadenes de caràcters

Una **cadena de caràcters** (*string*) és un vector de caràcters (`char`) que finalitza amb el caràcter sentinella `'\0'` (valor `0x00`). La codificació utilitzada és ASCII de 7 bits (Secció 2.9.1). Per recórrer una cadena de caràcters no cal conèixer-ne la longitud prèviament: n'hi ha prou de recórrer-la fins a trobar el sentinella.

Exercici 2.8 (`s2_5_1.s`)**.** El vector `vec` conté els dígits decimals del nombre 19865, ordenats de menys a més pes. El programa del Codi 2.7 converteix cada dígit a la seva representació ASCII i l'escriu a la cadena `cadena`; els caràcters queden ordenats de més a menys pes per poder imprimir el nombre correctament.

Codi 2.7: Conversió dels dígits de `vec` a caràcters ASCII.

C

```c
char         cadena[6];
unsigned int vec[5] = {5, 6, 8, 9, 1};

void main() {
    int i = 0;
```

```
    while (i < 5) {
        cadena[i] = vec[4-i] + '0';
        i++;
    }
    cadena[5] = 0;
}
```

Traduïu el programa a RV32I. La variable `i` s'emmagatzema al registre `s0`.

Nota: la conversió de dígit a caràcter ASCII consisteix a sumar el valor del dígit al codi ASCII de `'0'` (`0x30`). Per interpretar el contingut de `cadena` a la vista de memòria de RARS, vegeu Essencial 2.1.

Comproveu que, en acabar l'execució, la paraula a `0x10010000` mostra `0x36383931` i la paraula a `0x10010004`, `0x00000035`.

Solució 2.8.

`s2_5_1.s`

```
.data
cadena: .space  6
vec:    .word   5, 6, 8, 9, 1

.text
        # i = 0
        li      s0, 0

while:
        # if i >= 5, sortir
        li      t0, 5
        bge     s0, t0, fi

        # t1 = vec[4-i]
        li      t0, 4
        sub     t0, t0, s0  # t0 = 4 - i
        la      t1, vec
        slli    t2, t0, 2   # t2 = (4-i) * 4
        add     t1, t1, t2  # t1 = @vec[4-i]
        lw      t1, 0(t1)   # t1 = vec[4-i]

        # cadena[i] = vec[4-i] + '0'
        li      t2, 0x30    # '0' en ASCII
        add     t1, t1, t2  # t1 = vec[4-i] + '0'
        la      t2, cadena
        add     t2, t2, s0  # t2 = @cadena[i]
        sb      t1, 0(t2)   # cadena[i] = t1

        # i++
        addi    s0, s0, 1
        j       while

fi:
        # cadena[5] = 0
        la      t0, cadena
        sb      zero, 5(t0)
```

```
    # Sortir del programa
    li      a7, 93
    li      a0, 0
    ecall
```

> 💡 Comprovació pràctica
>
> Llegint els bytes d'aquestes dues paraules de menys a més adreça —és a dir, invertint l'ordre amb què RARS els mostra dins de cada paraula (Essencial 2.1)— s'obté `"19865\0"`.

Laboratori 3: Traducció de programes

Lliuraments

Cal lliurar les respostes als exercicis següents al campus virtual abans de l'inici de la sessió de laboratori:

Exercici	Nom i format del fitxer
Exercici 3.1	`s3_1_1.s`
Exercici 3.2	`s3_2_1.s`
Exercici 3.3	`s3_3_1.s`
Exercici 3.4	`s3_4_1.s`
Exercici 3.5	`s3_4_2.md` i `s3_4_2.s`
Exercici 3.6	`s3_5_1.s`

Lectura prèvia

Abans de fer la sessió, cal haver llegit i entès els continguts següents:

Concepte	On trobar-ho
Traducció literal C→RV32I	EC 3.1
Desplaçaments lògics (`sll`, `srl`, `slli`, `srli`)	RISC-V 3.1
Desplaçament aritmètic (`sra`, `srai`)	RISC-V 3.2
Operacions lògiques bit a bit (`and`, `or`, `xor`, `not`)	RISC-V 3.3, RISC-V 3.4
Instruccions de comparació (`slt`, `sltu`, `slti`, `sltiu`)	RISC-V 3.5
Salts condicionals (`beq`, `bne`, `blt`, `bge`, `bltu`, `bgeu`; pseudo: `bgt`, `ble`, `beqz`)	RISC-V 3.7, RISC-V 3.8, RISC-V 3.9
Salts incondicionals (`jal`, `j`, `ret`)	RISC-V 3.11, RISC-V 3.12, RISC-V 3.14
Traducció d'`if-then-else`	Secció 3.6
Traducció de `while` i `for`	Secció 3.7
Crida i retorn de subrutines (`ra`)	Secció 3.9.1
Pas de paràmetres i retorn de resultats (`a0`–`a7`)	Secció 3.9.2, RISC-V 3.17
Variables locals i bloc d'activació (`sp`)	Secció 3.9.3, RISC-V 3.18
Subrutines multinivell: registres temporals i segurs	Secció 3.9.4, RISC-V 3.22

3.1 Operacions lògiques i desplaçaments

Les operacions de desplaçament i lògiques bit a bit permeten construir màscares i manipular bits individuals. En aquest exercici es combinen per construir una màscara de bits de mida variable en temps d'execució (RISC-V 3.1, RISC-V 3.3).

Exercici 3.1 (`s3_1_1.s`)**.** Donats dos enters `X` ($0 \leq X \leq 31$) i `Y` als registres `s0` i `s1` respectivament, escriviu un fragment de codi RV32I que complementi els `X` bits de menys pes de `Y` i deixi el resultat a `s1`. No podeu usar instruccions de salt ni de comparació, i el fragment ha de tenir com a màxim 4 instruccions.

Indicació: la sentència C equivalent és:

`C`

```c
Y = Y ^ ((1 << X) - 1);
```

on `^` és la XOR bit a bit i `1 << X` desplaça el valor `1` exactament `X` posicions a l'esquerra.

Comproveu que, afegint `li s0, 5` i `li s1, 0x12345678` abans del fragment, `s1` acaba valent `0x12345667`.

Solució 3.1.

`s3_1_1.s`

```
.text
        li      t0, 1
        sll     t0, t0, s0  # t0 = 1 << X
        addi    t0, t0, -1  # t0 = (1 << X) - 1  (màscara dels X bits de menys pes)
        xor     s1, s1, t0  # complementa els X bits de menys pes de Y

        li      a7, 93
        li      a0, 0
        ecall
```

La clau és construir la màscara en temps d'execució: `1 << X` produeix un `1` a la posició `X` i zeros a la resta; restar-li `1` posa a `1` tots els bits de posicions 0 a `X-1`. El XOR final complementa exactament aquells bits.

3.2 Sentències condicionals

Una sentència `if-then-else` imbricada es tradueix aplicant el patró de Secció 3.6 de manera recursiva: cada `if` interior es tracta com una sentència del cos de l'`else` exterior. L'avaluació tardana (*lazy*) de `&&` i `||` (Essencial 3.5, Secció 3.6.2) s'implementa encadenant salts condicionals sense avaluar la segona condició quan la primera ja determina el resultat.

Exercici 3.2 (`s3_2_1.s`)**.** Traduïu a RV32I el programa C del Codi 3.1. Les variables globals `num` i `result` s'han de declarar a la secció `.data`. Tots els temporals van a registres `t`.

Codi 3.1: Classificació del caràcter `num`.

C

```c
char   num    = '7';
int    result = 0;

void main() {
    if (((num >= 'a') && (num <= 'z')) || ((num >= 'A') && (num <= 'Z')))
        result = num;
    else if ((num == ' ') || (num == '!'))
        result = 0;
    else
        result = -1;
}
```

El programa ha de respectar l'avaluació tardana de `&&` i `||` (Essencial 3.5). Comproveu que, per als valors `'m'`, `'Z'`, `' '`, `'!'` i `'7'` de `num` (codis ASCII: `0x6D`, `0x5A`, `0x20`, `0x21`, `0x37`), `result` val, respectivament, `0x6D`, `0x5A`, `0`, `0` i `-1` (`0xFFFFFFFF`).

Solució 3.2.

s3_2_1.s

```asm
.data
num:    .byte   '7'
result: .word   0

.text
        la      t1, num
        lb      t0, 0(t1)       # t0 = num

        # if (num >= 'a' && num <= 'z')
        li      t2, 'a'
        blt     t0, t2, comp_maj
        li      t2, 'z'
        bgt     t0, t2, comp_maj
        j       es_alfa

comp_maj:
        # || (num >= 'A' && num <= 'Z')
        li      t2, 'A'
        blt     t0, t2, comp_esp
        li      t2, 'Z'
        bgt     t0, t2, comp_esp

es_alfa:
        la      t1, result
        sw      t0, 0(t1)       # result = num
        j       fi

comp_esp:
```

```
        # else if (num == ' ' || num == '!')
        li      t2, ' '
        beq     t0, t2, es_esp  # primer operand || cert → lazy
        li      t2, '!'
        bne     t0, t2, es_altre

es_esp:
        la      t1, result
        sw      zero, 0(t1)     # result = 0
        j       fi

es_altre:
        la      t1, result
        li      t2, -1
        sw      t2, 0(t1)       # result = -1

fi:
        li      a7, 93
        li      a0, 0
        ecall
```

La primera branca del `||` exterior usa dos `&&` encadenats (dos rangs); la segona branca usa un `||` simple entre dues igualtats. En el `||` intern (`' '` o `'!'`), el salt a `es_esp` quan la primera condició és certa implementa l'avaluació tardana: si `num == ' '`, no s'avalua la comparació amb `'!'`.

3.3 Bucles

El patró de traducció d'un `while` (Secció 3.7) consisteix a avaluar la condició al principi de cada iteració i saltar al final del bucle si és falsa. El recorregut d'una cadena de caràcters fins al sentinella `'\0'` és un cas particular molt freqüent: la condició del bucle és que el caràcter actual sigui diferent de zero.

Exercici 3.3 (`s3_3_1.s`)**.** La cadena de caràcters `w` conté 31 dígits decimals seguits del sentinella `'\0'`. Escriviu un programa RV32I que compti quantes vegades apareix el caràcter `'4'` a `w` i deixi el resultat al registre `s0`.

C

```c
char w[32] = "44441857204449381672044492510 44";

void main() {
    int  i   = 0;
    int  n   = 0;
    char car = '4';
    while (w[i] != '\0') {
        if (w[i] == car)
            n++;
        i++;
    }
}
```

Nota: useu un punter (`t1`) en lloc d'un índex per recórrer la cadena: és un patró que reapareixerà, amb variacions, a l'exercici de subrutines.

Comproveu que, en acabar, `s0` val `12`.

Solució 3.3.

`s3_3_1.s`

```
.data
w:      .asciz  "4444185720444938167204449251044"

.text
        la      t1, w           # t1 = punter al caràcter actual
        li      s0, 0           # s0 = comptador d'aparicions de '4'
        li      t2, '4'         # t2 = caràcter a cercar

while:
        lb      t0, 0(t1)       # t0 = w[i]
        beqz    t0, fiwhile     # si w[i] == '\0', fi del bucle
        bne     t0, t2, seguent # si w[i] != '4', no comptem
        addi    s0, s0, 1       # s0++

seguent:
        addi    t1, t1, 1       # avança el punter
        j       while

fiwhile:
        li      a7, 93
        li      a0, 0
        ecall
```

El patró fonamental és: llegir el caràcter actual amb `lb`, comprovar el sentinella amb `beqz`, processar si escau, i avançar el punter amb `addi t1, t1, 1`. Aquest mateix esquelet apareixerà dins de `moda` a l'exercici de subrutines.

3.4 Subrutines

En una subrutina multinivell, els valors generats abans d'una crida i usats després s'han d'assignar a registres segurs (`s0`–`s11`); la subrutina ha de desar i restaurar aquests registres i `ra` (Secció 3.9.4, Essencial 3.11). Les variables locals estructurades (vectors) s'han d'allotjar al bloc d'activació (*stack frame*, BA) de la subrutina (RISC-V 3.20). Els dos exercicis d'aquesta secció s'han de fer en ordre: el segon reutilitza la traducció del primer.

Exercici 3.4 (`s3_4_1.s`)**.** La funció `update` rep un punter a l'histograma `h`, l'índex `i` del caràcter actual i l'índex `imax` del caràcter més freqüent fins ara. Incrementa `h[i]` i retorna `i` si ara és el més freqüent, o `imax` en cas contrari.

`C`

```c
char update(int *h, char i, char imax) {
    h[i]++;
    if (h[i] > h[imax])
        return i;
    else
        return imax;
}
```

Traduïu `update` a RV32I. Tingueu en compte:

- És una subrutina **uninivell**: no cal desar `ra`.
- Determineu quins valors, si n'hi ha, cal assignar a registres segurs abans d'escriure cap instrucció.
- El resultat es retorna a `a0` (RISC-V 3.17).

Solució 3.4. **Anàlisi de registres segurs.** `update` és uninivell: no hi ha cap crida, de manera que cap valor travessa una frontera de crida. Tots els temporals van a registres `t`; no cal cap registre `s`. El BA és buit.

`s3_4_1.s`

```s
update:
        # update(int *h, char i, char imax)
        # a0 = h, a1 = i, a2 = imax

        # h[i]++
        slli    t0, a1, 2       # t0 = i * 4
        add     t0, a0, t0      # t0 = &h[i]
        lw      t1, 0(t0)
        addi    t1, t1, 1
        sw      t1, 0(t0)       # h[i]++

        # if (h[i] > h[imax]) return i; else return imax
        slli    t2, a2, 2       # t2 = imax * 4
        add     t2, a0, t2      # t2 = &h[imax]
        lw      t2, 0(t2)       # t2 = h[imax]
        ble     t1, t2, ret_imax
        mv      a0, a1          # return i
        ret
ret_imax:
        mv      a0, a2          # return imax
        ret
```

Com que `update` és uninivell, els paràmetres `a0`–`a2` es poden usar directament durant tot el cos sense risc de sobreescriptura per cap crida. No cal moure'ls a registres segurs.

Exercici 3.5 (`s3_4_2.md` i `s3_4_2.s`)**.** La funció `moda` recorre la cadena `vec` de `num` caràcters, construeix un histograma local `histo[10]` i, per a cada caràcter, crida `update` (traduïda a Solució 3.4) per actualitzar el dígit més freqüent.

`C`

```c
char w[32]    = "44441857204449381672044492510 44";
char resultat;

char moda(char *vec, int num) {
    int  k, histo[10];
    char max;
    for (k = 0; k < 10; k++)
        histo[k] = 0;
    max = '0';
    for (k = 0; k < num; k++)
        max = '0' + update(histo, vec[k] - '0', max - '0');
    return max;
}

void main() {
    resultat = moda(w, 31);
}
```

Traduïu `moda` i el programa principal a RV32I, al fitxer `s3_4_2.s`. Abans d'escriure cap instrucció, responeu al fitxer `s3_4_2.md`:

1. Identifiqueu quines dades es generen **abans** de la crida a `update` i s'usen **després** (Secció 3.9.4). Assigneu-les a registres segurs.
2. Dibuixeu el bloc d'activació de `moda` amb els desplaçaments de cada element.

Comproveu que al final de l'execució la variable global `resultat` conté el caràcter `'4'`.

Nexe amb Exercici 3.3: el segon `for` de `moda` segueix el mateix patró de recorregut seqüencial caràcter a caràcter que heu fet servir a l'exercici de bucles (ara amb índex explícit en lloc de punter). Identifiqueu on apareix.

Solució 3.5. **Pas 1. Registres segurs de `moda`.**

Dades generades abans de la crida a `update` i usades després: `histo` (adreça base = `sp`; derivable en tot moment, no cal registre), `vec` → `s0`, `num` → `s1`, `k` → `s2`, `max` → `s3`. Cal desar també `ra`.

Pas 2. Bloc d'activació de `moda` (Figura 3.1).

Mida del BA: 60 bytes (EC 3.7).

`s3_4_2.s`

```
.data
w:      .asciz  "44441857204449381672044492510 44"
resultat: .byte 0

.text

        la      a0, w
```

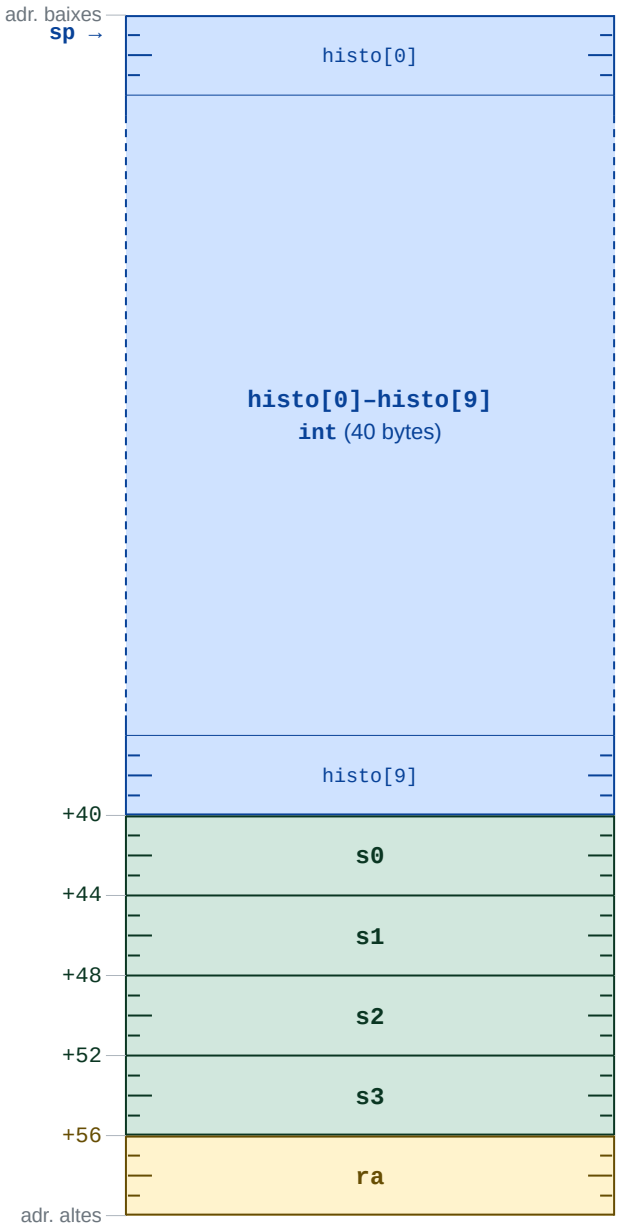


Figura 3.1: Bloc d'activació de `moda`: el vector `histo` al desplaçament `+0` i els registres desats `s0`, `s1`, `s2`, `s3` i `ra` als desplaçaments `+40`, `+44`, `+48`, `+52` i `+56`.

```
        li      a1, 31
        jal     ra, moda

        la      t0, resultat
        sb      a0, 0(t0)

        li      a7, 93
        li      a0, 0
        ecall

# update: vegeu la solució de s3_4_1.s (inseriu el codi aquí)

moda:
        # moda(char *vec, int num) — a0 = vec, a1 = num
        # pròleg
        addi    sp, sp,  -60
        sw      s0,  40(sp)
        sw      s1,  44(sp)
        sw      s2,  48(sp)
        sw      s3,  52(sp)
        sw      ra,  56(sp)
        mv      s0, a0          # s0 = vec
        mv      s1, a1          # s1 = num

        # for (k=0; k<10; k++) histo[k] = 0
        li      s2, 0           # k = 0
        li      t0, 10
for1:
        bge     s2, t0, fifor1
        slli    t1, s2, 2       # t1 = k * 4
        add     t1, sp, t1      # t1 = &histo[k]
        sw      zero, 0(t1)
        addi    s2, s2, 1
        j       for1
fifor1:

        # max = '0'
        li      s3, '0'

        # for (k=0; k<num; k++)   ← mateix patró que s3_3_1.s
        li      s2, 0           # k = 0
for2:
        bge     s2, s1, fifor2

        # vec[k] → t0; a1 = vec[k] - '0'
        add     t0, s0, s2      # t0 = &vec[k]
        lb      t0, 0(t0)       # t0 = vec[k]
        li      t1, '0'
        sub     a1, t0, t1      # a1 = vec[k] - '0'

        # a2 = max - '0'
        sub     a2, s3, t1      # a2 = max - '0'

        # a0 = &histo
        mv      a0, sp

        jal     ra, update

        # max = '0' + resultat
```

```
        li      t1, '0'
        add     s3, t1, a0      # s3 = '0' + update(...)

        addi    s2, s2, 1       # k++
        j       for2
fifor2:

        mv      a0, s3          # return max

        # epíleg
        lw      s0,  40(sp)
        lw      s1,  44(sp)
        lw      s2,  48(sp)
        lw      s3,  52(sp)
        lw      ra,  56(sp)
        addi    sp, sp, 60
        ret
```

El patró de Exercici 3.3 reapareix al segon `for`: `add t0, s0, s2` calcula l'adreça del caràcter actual (equivalent a avançar el punter), `lb t0, 0(t0)` el llegeix, i `bge s2, s1, fifor2` comprova la condició de fi (aquí amb comptador explícit en lloc de sentinella, perquè `num` és conegut). El resultat esperat a `resultat` és `'4'` (`0x34`).

3.5 Depuració de codi

La depuració és el procés d'identificar i corregir errors en un programa. En assemblador, els errors més freqüents són confusions entre registres, errors d'adreçament i una gestió incorrecta del bloc d'activació. En aquest exercici es practiquen tots tres.

Exercici 3.6 (`s3_5_1.s`)**.** El programa del Codi 3.2 declara el vector global `alfabet` amb les lletres majúscules de l'A a la Z i una cadena de caràcters d'entrada `w1`. La funció `codifica` recorre `w1` caràcter a caràcter i, per a cada lletra, crida la funció `g`, que calcula la lletra simètrica (A↔Z, B↔Y, etc.) i en retorna el codi ASCII. El resultat s'escriu al vector de sortida `w2`.

Codi 3.2: Programa amb les funcions `codifica` i `g`.

C

```c
char alfabet[27] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
char w1[13]      = "ARQUITECTURA";
char w2[16];

int codifica(char *pfrasein, char *pfraseout) {
    int i = 0;
    while (*pfrasein != 0) {
        *pfraseout = g(alfabet, pfrasein);
        pfrasein++;
        pfraseout++;
        i++;
```

```
    }
    *pfraseout = 0;
    return i;
}

char g(char *alfa, char *pfrase) {
    return alfa[25 - (*pfrase - 'A')];
}

void main() {
    codifica(w1, w2);
}
```

El Codi 3.3, la traducció a RV32I, conté **tres errors**, tots dins de `codifica`. La funció `g` és correcta. Identifiqueu cada error, expliqueu per què és incorrecte i escriviu-ne la correcció. Les explicacions van com a comentaris dins del mateix `s3_5_1.s`, a la línia de cada correcció.

Codi 3.3: Traducció amb tres errors dins de `codifica`.

`s3_5_1.s`

```
.data
alfabet: .asciz "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
w1:      .asciz  "ARQUITECTURA"
w2:      .space  16

.text

        la      a0, w1
        la      a1, w2
        jal     ra, codifica

        li      a7, 93
        li      a0, 0
        ecall

g:
        lb      t0, 0(a1)       # t0 = *pfrase
        li      t1, 'A'
        sub     t0, t0, t1      # t0 = *pfrase - 'A'
        li      t1, 25
        sub     t0, t1, t0      # t0 = 25 - (*pfrase - 'A')
        add     t0, a0, t0      # t0 = &alfa[25 - ...]
        lb      a0, 0(t0)       # a0 = alfa[25 - ...]
        ret

codifica:
        addi    sp, sp, -16     # reserva el BA
        sw      s0,  0(sp)
        sw      s1,  4(sp)
        sw      s2,  8(sp)
        mv      s0, a0          # s0 = pfrasein
        mv      s1, a1          # s1 = pfraseout
        li      s2, 0           # i = 0

while:
        lb      t0, 0(s0)
        beqz    t0, fiwhile
```

```
        la      a0, alfabet
        mv      a1, s1
        jal     ra, g

        sb      a0, 0(s0)
        addi    s0, s0, 1
        addi    s1, s1, 1
        addi    s2, s2, 1
        j       while

fiwhile:
        sb      zero, 0(s1)
        mv      a0, s2

        lw      s0,  0(sp)
        lw      s1,  4(sp)
        lw      s2,  8(sp)
        addi    sp, sp, 16
        ret
```

Comproveu que, un cop esmenats els errors, `w2` conté `"ZIJFRGVXGFIZ"` al final de l'execució.

Solució 3.6. El programa conté tres errors, tots dins de `codifica`: el pròleg i l'epíleg no tracten `ra`, la crida a `g` rep el punter equivocat i el resultat s'escriu al vector equivocat.

Error 1 — el pròleg no desa `ra` i l'epíleg no el restaura.

`codifica` és multinivell (crida `g`). La instrucció `jal ra, g` sobreescriu `ra` amb l'adreça de retorn cap a `codifica`, destruint l'adreça de retorn cap al programa principal. Quan `codifica` executa `ret`, no torna al programa principal sinó a la instrucció següent a `jal ra, g` (l'adreça de retorn de l'última crida a `g`), de manera que l'execució reprèn dins del cos del bucle de `codifica`.

El símptoma observable és immediat: l'epíleg ja ha restaurat `s0` amb el valor que tenia en entrar a `codifica` —el del programa principal, que no l'inicialitza mai i, per tant, val 0—, de manera que la primera instrucció reexecutada, `sb a0, 0(s0)`, intenta escriure a l'adreça `0x00000000`. RARS avorta amb «`address out of range 0x00000000`».

Correcció: desar `ra` al pròleg i restaurar-lo a l'epíleg.

`RV32I`

```
        # pròleg — afegir:
        sw      ra, 12(sp)      # codifica crida g, que sobreescriu ra

        # epíleg — afegir:
        lw      ra, 12(sp)      # recupera l'adreça de retorn cap al programa principal
```

Error 2 — `mv a1, s1` passa `pfraseout` (`s1`) a `g` en lloc de `pfrasein` (`s0`).

La funció `g` ha de rebre el punter al caràcter d'entrada per calcular-ne la lletra simètrica. Amb `mv a1, s1` rep el punter de sortida, que apunta a una posició de `w2` (inicialment buida), i retorna un resultat incorrecte.

Correcció:

`RV32I`

```
        mv      a1, s0          # g ha de rebre el punter d'entrada
```

Error 3 — `sb a0, 0(s0)` escriu a `pfrasein` (`s0`) en lloc de `pfraseout` (`s1`).

El caràcter codificat que retorna `g` a `a0` s'ha d'escriure al vector de sortida. Amb `sb a0, 0(s0)` s'escriu sobre el vector d'entrada `w1`, corrompent-lo i deixant `w2` sense modificar.

Correcció:

`RV32I`

```
        sb      a0, 0(s1)       # el caràcter codificat va al vector de sortida
```

Versió corregida de `codifica`:

`s3_5_1.s`

```
codifica:
        addi    sp, sp, -16
        sw      s0,  0(sp)
        sw      s1,  4(sp)
        sw      s2,  8(sp)
        sw      ra, 12(sp)      # codifica crida g, que sobreescriu ra
        mv      s0, a0
        mv      s1, a1
        li      s2, 0

while:
        lb      t0, 0(s0)
        beqz    t0, fiwhile

        la      a0, alfabet
        mv      a1, s0          # g ha de rebre el punter d'entrada
        jal     ra, g

        sb      a0, 0(s1)       # el caràcter codificat va al vector de sortida
        addi    s0, s0, 1
        addi    s1, s1, 1
        addi    s2, s2, 1
        j       while

fiwhile:
        sb      zero, 0(s1)
        mv      a0, s2

        lw      s0,  0(sp)
        lw      s1,  4(sp)
        lw      s2,  8(sp)
        lw      ra, 12(sp)      # recupera l'adreça de retorn cap al programa principal
        addi    sp, sp, 16
        ret
```

Laboratori 4: Matrius i subrutines

Lliuraments

Cal lliurar les respostes als exercicis següents al campus virtual abans de l'inici de la sessió de laboratori:

Exercici	Nom i format del fitxer
Exercici 4.1	`s4_1_1.s`
Exercici 4.2	`s4_2_1.md`
Exercici 4.3	`s4_2_2.md` i `s4_2_2.s`
Exercici 4.4	`s4_3_1.md` i `s4_3_1.s`

Lectura prèvia

Abans de fer la sessió, cal haver llegit i entès els continguts següents:

Concepte	On trobar-ho
Declaració de matrius en memòria (`.space`, `.word`, `.align`)	Secció 4.5.1, RISC-V 2.31
Càlcul de l'adreça d'un element `mat[i][j]`	Equació 4.3, Secció 4.5.2
Instruccions de lectura i escriptura (`lw`, `sw`, `lb`, `sb`, `lbu`)	RISC-V 2.29
`long long` en RV32I: dos registres per valor de 64 bits	Essencial 2.11
Instrucció de multiplicació entera (`mul`)	RISC-V 4.2
Pas de matrius per referència com a paràmetre	Exemple 4.13
Subrutines multinivell: registres segurs i bloc d'activació	Secció 3.9.4, RISC-V 3.20
Accés seqüencial i *stride*	Secció 4.6.1, Equació 4.4

4.1 Declaració de matrius

Les matrius multidimensionals s'emmagatzemen en memòria en ordre de files (*row-major order*): primer tots els elements de la fila 0, després els de la fila 1, i així successivament fins a la darrera fila (Secció 4.5.1). La mida total d'una matriu de `NF` files, `NC` columnes i elements de T bytes és $\texttt{NF} \cdot \texttt{NC} \cdot T$ bytes.

Exercici 4.1 (`s4_1_1.s`)**.** Declareu al segment `.data` les variables globals del Codi 4.1. Useu `.eqv` per a les dimensions de cada matriu (EC 4.1). Garantiu l'alineació correcta de cada variable amb la directiva `.align` on calgui.

Codi 4.1: Quatre matrius globals de tres tipus.

C

```c
int       mat1[5][6];
char      mat2[3][5];
long long mat3[2][2];
int       mat4[2][3] = {{2, 3, 1}, {2, 4, 3}};
```

Recordeu que en C els elements d'una matriu es guarden per files, i que en RV32I:

- `int` ocupa 4 bytes (alineació a 4 bytes, `.align 2`).
- `char` ocupa 1 byte (sense restricció d'alineació).
- `long long` ocupa 8 bytes (alineació a 8 bytes, `.align 3`); calen **dos registres** per operar-hi (Essencial 2.11).

Solució 4.1.

s4_1_1.s

```
.eqv NF1, 5
.eqv NC1, 6
.eqv NF2, 3
.eqv NC2, 5
.eqv NF3, 2
.eqv NC3, 2
.eqv NF4, 2
.eqv NC4, 3

.data
        .align 2
mat1:   .space  120             # int mat1[5][6]: NF1*NC1*4 = 120 bytes

mat2:   .space  15              # char mat2[3][5]: NF2*NC2*1 = 15 bytes (sense alineació)

        .align 3
mat3:   .space  32          # long long mat3[2][2]: NF3*NC3*8 = 32 bytes

        .align 2
mat4:   .word   2, 3, 1, 2, 4, 3  # int mat4[2][3] inicialitzada per files

.text
        li      a7, 93
        li      a0, 0
        ecall
```

La mida de cada `.space` s'escriu com a **literal ja calculat**, amb l'expressió aritmètica al comentari, perquè RARS no avalua expressions als operands (RISC-V 2.11); per això, tot i estar definides amb

`.eqv`, les dimensions `NF`/`NC` no es poden multiplicar dins `.space` (`NF1*NC1*4`): una constant `.eqv` sola sí que hi és vàlida, però no una expressió.

`mat2` és de tipus `char` (1 byte), de manera que no requereix cap directiva `.align`. Tot i així, la variable **següent** (`mat3`) necessita alineació a 8 bytes: la directiva `.align 3` just abans de `mat3` garanteix que comença en una adreça múltiple de 8, independentment de quants bytes ocupa `mat2`.

`mat4` s'inicialitza amb `.word` llistant els elements **per files**: primer la fila 0 (`2, 3, 1`) i després la fila 1 (`2, 4, 3`). La directiva `.align 2` és explícita a títol documental: com que `mat3` comença en una adreça múltiple de 8 i ocupa 32 bytes, l'adreça següent (la de `mat4`) ja és múltiple de 4.

4.2 Accés a elements

L'adreça d'un element `mat[i][j]` d'una matriu de `NC` columnes i elements de T bytes es calcula com (Equació 4.3):

$$@\texttt{mat[i][j]} = @\texttt{mat[0][0]} + (i \cdot \texttt{NC} + j) \cdot T$$

Exercici 4.2 (`s4_2_1.md`)**.** Donades les variables globals declarades a Solució 4.1, calculeu l'adreça de memòria de cada un dels elements següents en funció de l'adreça base de la matriu corresponent:

- `@mat1[4][3]`
- `@mat2[2][4]`
- `@mat3[1][0]`
- `@mat4[0][2]`

Solució 4.2. Aplicant la fórmula $@\texttt{mat[i][j]} = @\texttt{mat[0][0]} + (i \cdot \texttt{NC} + j) \cdot T$:

Element	`NC`	T	Càlcul	Resultat
`mat1[4][3]`	6	4	`@mat1 + (4·6 + 3)·4`	`@mat1 + 108`
`mat2[2][4]`	5	1	`@mat2 + (2·5 + 4)·1`	`@mat2 + 14`
`mat3[1][0]`	2	8	`@mat3 + (1·2 + 0)·8`	`@mat3 + 16`
`mat4[0][2]`	3	4	`@mat4 + (0·3 + 2)·4`	`@mat4 + 8`

Per a `mat3`, $T = 8$ perquè cada element és de tipus `long long` (64 bits). L'element `mat3[1][0]` ocupa les adreces `@mat3 + 16` (paraula baixa) i `@mat3 + 20` (paraula alta), en ordre *little-endian* (Secció 2.5.1).

Exercici 4.3 (`s4_2_2.md` i `s4_2_2.s`)**.** Traduïu a RV32IM el programa C del Codi 4.2. El programa principal crida `subr` dues vegades; `subr` accedeix a `mat1` per índexs variables.

Codi 4.2: Programa que crida dues vegades la funció `subr`.

`C`

```
int mat1[5][6];
int mat4[2][3] = {{2, 3, 1}, {2, 4, 3}};
int col = 2;

void main() {
    mat1[4][3] = subr(mat4, mat4[0][2], col);
    mat1[0][0] = subr(mat4, 1, 1);
}

int subr(int x[][3], int i, int j) {
    mat1[j][5] = x[i][j];
    return i;
}
```

`subr` rep una matriu per referència (`x` és l'adreça base) i accedeix a `x[i][j]` i a `mat1[j][5]` amb índexs variables: cal usar `mul` (RISC-V 4.2).

Abans d'escriure cap instrucció:

1. Determineu si `subr` és uninivell o multinivell i per què.
2. Determineu si el programa principal fa crides i quines conseqüències té per a la gestió del context. Tingueu en compte que és el punt d'entrada del programa —és el primer bloc del segment `.text`—: no és cridat per cap altra subrutina i acaba amb `ecall` (RISC-V 3.23).
3. Per a cada funció, identifiqueu quines dades es generen **abans** d'una crida i s'usen **després** (Secció 3.9.4). Assigneu-les a registres segurs.
4. Dibuixeu el bloc d'activació de cada funció amb els desplaçaments de cada element.

Un cop escrit, poseu un punt de ruptura a l'inici de `subr`, executeu fins a aquest punt i anoteu els valors (en hexadecimal) dels registres `a0`, `a1`, `a2` i `ra` en cadascuna de les dues crides.

Comproveu que, en acabar, `mat1[0][0]` = 1, `mat1[1][5]` = 4, `mat1[2][5]` = 3 i `mat1[4][3]` = 1.

Solució 4.3. **Pas 1. `subr`: uninivell o multinivell.**

`subr` és **uninivell**: no fa cap crida. No cal preservar `ra` ni cap registre segur per a valors que travessen una crida; tots els temporals van a registres `t` i el bloc d'activació queda buit.

Pas 2. El programa principal: gestió del context.

El programa principal fa dues crides a `subr`, però és el **punt d'entrada**: com que és el primer bloc del segment `.text`, no és la funció cridada per cap altra subrutina, i acaba amb `ecall` (sortida), no amb `ret`. Per tant, **no ha de preservar `ra` ni cap registre segur** per a ningú, i no li cal bloc d'activació. Sí que ha d'usar registres **segurs** (`s0`–`s11`) per a les dades pròpies que han de sobreviure a les crides: com que `subr` és responsable de restaurar-los abans de retornar (Essencial 3.11), el seu valor es conserva a través de cada crida.

Pas 3. Registres segurs del programa principal.

Entre la primera crida a `subr` i la segona, el programa principal necessita les adreces de `mat1` i `mat4`. Com que aquests valors travessen les crides, s'assignen a registres segurs:

Dada	Registre
`@mat1`	`s0`

Dada	Registre
`@mat4`	`s1`

Pas 4. Blocs d'activació.

Cap de les dues rutines no necessita bloc d'activació: `subr` és uninivell i sense variables locals, i el programa principal no ha de preservar cap registre. Tots dos BA són buits.

`s4_2_2.s`

```
.eqv NF1, 5
.eqv NC1, 6
.eqv NF4, 2
.eqv NC4, 3

.data
        .align 2
mat1:   .space  120             # int mat1[5][6]: NF1*NC1*4 = 120 bytes
        .align 2
mat4:   .word   2, 3, 1, 2, 4, 3  # int mat4[2][3]
col:    .word   2

.text

        la      s0, mat1          # s0 = @mat1 (segur: travessa les crides)
        la      s1, mat4          # s1 = @mat4 (segur: travessa les crides)

        # mat1[4][3] = subr(mat4, mat4[0][2], col)
        mv      a0, s1            # arg0: @mat4
        lw      a1, 8(s1)         # arg1: mat4[0][2] = @mat4 + 8 → 1
        la      t0, col
        lw      a2, 0(t0)         # arg2: col = 2
        jal     ra, subr          # a0 = subr(@mat4, 1, 2) → retorna 1

        # @mat1[4][3] = @mat1 + (4*NC1 + 3)*4 = @mat1 + 108
        sw      a0, 108(s0)

        # mat1[0][0] = subr(mat4, 1, 1)
        mv      a0, s1            # arg0: @mat4
        li      a1, 1             # arg1: i = 1
        li      a2, 1             # arg2: j = 1
        jal     ra, subr          # a0 = subr(@mat4, 1, 1) → retorna 1

        # @mat1[0][0] = @mat1 + 0
        sw      a0, 0(s0)

        li      a7, 93
        li      a0, 0
        ecall

# subr(int x[][3], int i, int j)
# a0 = @x, a1 = i, a2 = j
# retorna i (a0)
# uninivell: a2 = j és vàlid durant tot el cos (cap crida el destrueix)
subr:
        # t0 = x[i][j] = Mw[@x + (i*NC4 + j)*4]
        li      t1, NC4
```

```
        mul     t0, a1, t1          # t0 = i * NC4
        add     t0, t0, a2          # t0 = i*NC4 + j
        slli    t0, t0, 2           # t0 = (i*NC4 + j) * 4
        add     t0, a0, t0          # t0 = @x[i][j]
        lw      t0, 0(t0)           # t0 = x[i][j]

        # mat1[j][5] = x[i][j]
        # @mat1[j][5] = @mat1 + (j*NC1 + 5)*4
        la      t1, mat1
        li      t2, NC1
        mul     t3, a2, t2          # t3 = j * NC1
        addi    t3, t3, 5           # t3 = j*NC1 + 5
        slli    t3, t3, 2           # t3 = (j*NC1 + 5) * 4
        add     t1, t1, t3          # t1 = @mat1[j][5]
        sw      t0, 0(t1)           # mat1[j][5] = x[i][j]

        mv      a0, a1              # return i
        ret
```

> 💡 Comprovació pràctica
>
> Els quatre elements de l'enunciat són a `@mat1 + 0`, `@mat1 + 44`, `@mat1 + 68` i `@mat1 + 108`. Per localitzar-los a la vista de dades de RARS, calculeu l'adreça absoluta sumant el desplaçament a l'adreça base de `mat1` (visible al panell *Labels*).

4.3 Accés seqüencial

Quan un bucle recorre els elements d'una matriu i la distància en bytes entre dos elements consecutius del recorregut és constant, es pot aplicar la tècnica d'**accés seqüencial** (Secció 4.6.1): en lloc de recalcular l'adreça a cada iteració, es manté un punter `p` que s'incrementa sumant-li l'***stride*** (Equació 4.4):

$$\text{stride} = @X_{i+1} - @X_i = \text{constant}$$

L'*stride* depèn del patró de recorregut. Per a una matriu d'elements de mida T bytes i `NC` columnes (Secció 4.6.4):

Recorregut	*Stride*
Una fila (elements consecutius)	T
Una columna (salt entre files)	$\texttt{NC} \cdot T$

Exercici 4.4 (`s4_3_1.md` i `s4_3_1.s`)**.** Considereu la matriu i el programa C del Codi 4.3:

Codi 4.3: Matriu `mat` i programa que en suma una columna.

C

```c
int mat[4][6] = { {0, 0, 2, 0, 0, 0},
                  {0, 0, 4, 0, 0, 0},
                  {0, 0, 6, 0, 0, 0},
                  {0, 0, 8, 0, 0, 0} };
int resultat;

void main() {
    resultat = suma_col(mat);
}

int suma_col(int m[][6]) {
    int i, suma = 0;
    int *p;
    p = &m[0][2];       // inicialitzar el punter
    for (i = 0; i < 4; i++) {
        suma += *p;     // accés a m usant el punter
        p += 6;         // incrementar el punter
    }
    return suma;
}
```

Abans d'escriure cap instrucció, responeu al fitxer `s4_3_1.md`:

1. Calculeu l'adreça de `m[i][2]` en funció de `@m` i `i`.
2. Calculeu l'*stride* per recórrer la columna 2 (distància en bytes entre `m[i][2]` i `m[i+1][2]`).
3. Expliqueu per què `p += 6` al codi C correspon, en assemblador, a sumar l'*stride* calculat a l'apartat anterior (aritmètica de punters).
4. Determineu si `suma_col` és uninivell o multinivell i assigneu cada variable a un registre.
5. Determineu si el programa principal fa alguna crida i quines conseqüències té per a la gestió del context. Recordeu que és el punt d'entrada del programa —el primer bloc de `.text`— i que acaba amb `ecall`.

A continuació, traduïu el programa a RV32IM usant la tècnica d'accés seqüencial per recórrer la columna i eliminant la variable d'inducció (Secció 4.6.3).

Comproveu que, en acabar, la variable global `resultat` conté `20` (la suma dels elements de la columna 2: $2+4+6+8$).

Solució 4.4. **Pas 1. Adreça de `m[i][2]`.**

$$@\texttt{m[i][2]} = @\texttt{m} + (i \cdot 6 + 2) \cdot 4 = @\texttt{m} + i \cdot 24 + 8$$

Pas 2. Stride.

$$\text{stride} = @\texttt{m[i+1][2]} - @\texttt{m[i][2]} = 24 \text{ bytes}$$

Pas 3. Aritmètica de punters.

En C, `p` és de tipus `int *`. Un increment d'una unitat (`p += 1`) equival a sumar la mida de l'element apuntat en bytes (4 bytes per a `int`). Per tant, `p += 6` equival a sumar $6 \cdot 4 = 24$ bytes en assemblador, que coincideix exactament amb l'*stride* calculat.

Pas 4. Assignació de registres de `suma_col`.

`suma_col` és **uninivell**: no fa cap crida. Tots els temporals van a registres `t`; BA buit.

Variable	Registre
`p` (punter, inicialitzat a `@m[0][2]`)	`t0`
`suma`	`t1`
sentinella (`@m[NF][2]`)	`t2`
stride (24)	`t3`
`*p` (valor llegit a cada iteració)	`t4`

El registre `t4` es reutilitza abans del bucle com a temporal per calcular la sentinella (`NF`), i dins del bucle per contenir `*p`; no hi ha conflicte perquè els dos usos no coincideixen en el temps.

Com que el recorregut elimina la variable d'inducció `i` i controla el bucle comparant el punter amb una sentinella (Secció 4.6.3), la comparació es fa entre **adreces**, que són naturals: per això s'usa `bgeu`/`bltu` en lloc de `bge`/`blt` (Essencial 4.4).

Pas 5. Gestió del context del programa principal.

El programa principal crida `suma_col`, però és el **punt d'entrada**: com que és el primer bloc del segment `.text`, no és la funció cridada per cap altra subrutina, i acaba amb `ecall`. Per tant, no ha de preservar `ra` ni cap registre segur, i no li cal bloc d'activació. A més, no hi ha cap valor propi que hagi de travessar la crida (el resultat es desa directament a `resultat` en retornar), de manera que no necessita cap registre segur.

`s4_3_1.s`

```
.eqv NF, 4
.eqv NC, 6

.data
        .align 2
mat:    .word   0, 0, 2, 0, 0, 0
        .word   0, 0, 4, 0, 0, 0
        .word   0, 0, 6, 0, 0, 0
        .word   0, 0, 8, 0, 0, 0
        .align 2
resultat: .word 0

.text

        la      a0, mat
        jal     ra, suma_col    # a0 = suma_col(mat)

        la      t0, resultat
        sw      a0, 0(t0)       # resultat = a0

        li      a7, 93
        li      a0, 0
        ecall

# suma_col(int m[][6])
# a0 = @m
# retorna suma (a0)
suma_col:
```

```
        li      t1, 0           # suma = 0
        li      t3, 24          # stride = NC*4 = 6*4 = 24
        addi    t0, a0, 8       # p = @m[0][2] = @m + (0*NC + 2)*4 = @m + 8
        li      t4, NF
        mul     t2, t4, t3
        add     t2, t0, t2      # sentinella = @m[NF][2] = p + NF*stride
for:
        bgeu    t0, t2, fifor   # salta si p >= sentinella
        lw      t4, 0(t0)       # t4 = *p = m[i][2]
        add     t1, t1, t4      # suma += *p
        add     t0, t0, t3      # p += stride
        j       for
fifor:
        mv      a0, t1          # return suma
        ret
```

Comprovació pràctica

Per verificar l'accés seqüencial, poseu un punt de ruptura a la instrucció `lw t4, 0(t0)` del bucle. A cada pulsació de *Run* → *Go* (F5), observeu com `t0` avança 24 bytes i `t4` pren successivament els valors `2`, `4`, `6` i `8`.

Aprofundiment 4.1: L'*stride* i la memòria cau

L'*stride* determina l'eficiència d'ús de la memòria cau. Recórrer una **fila** (amb un *stride* de 4 bytes) és eficient: elements consecutius comparteixen el mateix bloc. Recórrer una **columna** (amb un *stride* de 24 bytes, en aquest cas) salta entre files i pot provocar moltes més fallades de memòria cau (*cache misses*), especialment si `NC` és gran.
A Capítol 6 s'analitza quantitativament aquest efecte.

Laboratori 5: Coma flotant i compilació separada

Lliurements

Cal lliurar les respostes als exercicis següents al campus virtual abans de l'inici de la sessió de laboratori:

Exercici	Nom i format del fitxer
Exercici 5.1	`s5_1_1.s` i `s5_1_2.s`
Exercici 5.2	`s5_2_1.md` i `s5_2_1.s`
Exercici 5.3	`s5_3_1.s`
Exercici 5.4	`s5_4_1.md`

Lectura prèvia

Abans de fer la sessió, cal haver llegit i entès els continguts següents:

Concepte	On trobar-ho
Format IEEE 754 de precisió simple: signe, exponent esbiaixat, fracció	Secció 5.1.3, RISC-V 5.1
El bit ocult	Essencial 5.1
Codificació i decodificació de nombres en coma flotant	Exemple 5.2, Exemple 5.3
Modes d'arrodoniment	Secció 5.1.7
Desplaçaments lògics i aritmètics (`sll`, `srl`, `sra`, `slli`, `srli`, `srai`)	RISC-V 3.1, RISC-V 3.2
Operacions lògiques bit a bit i de comparació (`and`, `or`, `slt`)	RISC-V 3.3, RISC-V 3.5
Subrutines: pas de paràmetres i retorn de resultats	Secció 3.9.2, RISC-V 3.17
Subrutines multinivell: registres segurs i bloc d'activació	Secció 3.9.4, RISC-V 3.22
Compilació separada i directiva `.globl`	Secció 3.10, RISC-V 3.24

5.1 Compilació separada a RARS

En un projecte real, el codi es divideix en **mòduls** independents que es compilen i assemblen per separat (Secció 3.10). Cadascun viu en el seu propi fitxer font, declara com a globals les etiquetes que han de ser visibles des d'altres mòduls, i referencia les etiquetes externes sense necessitat de conèixer-ne els detalls interns.

RARS reprodueix aquest flux de manera directa: amb l'opció *Settings* → *Assemble all files currently open* activada, l'assemblador processa tots els fitxers oberts alhora i resol automàticament les referències entre ells, assumint la funció combinada d'assemblador i enllaçador.

Essencial 5.1: Compilació separada — directiva `.globl`

Qualsevol etiqueta referenciada des d'un altre fitxer ha de declarar-se amb `.globl` al fitxer on es defineix. Sense aquesta declaració, l'etiqueta és invisible per als altres mòduls i RARS finalitza amb un error de símbol no resolt. Només s'hi declaren els símbols que altres mòduls necessiten: el programa principal no en porta, perquè cap fitxer no hi fa referència i RARS no el cerca per nom (vegeu l'aprofundiment següent).

Comprovació pràctica

Un cop oberts tots els fitxers i activada l'opció *Assemble all files currently open*, assembleu el projecte. Al panell *Labels* (o *Symbol Table*) de RARS hi apareixeran totes les etiquetes globals dels mòduls. Comproveu que les etiquetes esperades hi són i que les seves adreces s'han resolt correctament.

RISC-V 5.1: RARS — Ordre d'assemblatge de diversos fitxers

L'execució comença a la primera instrucció del segment `.text` (Secció 2.3.1). Quan s'assemblen **diversos fitxers alhora**, aquell `.text` és el **combinat**, i l'ordre en què s'hi encadenen els mòduls el decideix RARS: amb *Assemble all files currently open*, el primer que s'assembla és el del fitxer que té la **pestanya activa** en prémer F3.
Per tant, **la pestanya activa ha de ser la del fitxer que conté el programa principal**. Si és la d'un fitxer de subrutines, el programa arrenca dins d'una subrutina.
RARS no té enllaçador separat: assembla i resol les referències entre mòduls en una sola passada, de manera que el que es tria aquí és l'**ordre d'assemblatge** (el procés que ho fa en dues fases és a Secció 3.10.4).
L'assemblatge no falla si l'ordre és el dolent. La consola diu `operation completed successfully` en tots dos casos, i l'error només apareix en executar:

```
Assemble: assembling s5_1_2.s, s5_1_1.s
Assemble: operation completed successfully.
```

```
Assemble: assembling s5_1_1.s, s5_1_2.s
Assemble: operation completed successfully.
```

L'execució, en canvi, sí que els distingeix:

```
-- program is finished running (0) --
```

```
Go: running s5_1_1.s

Error in : Instruction load access error

Go: execution terminated with errors.
```

La primera pista és el nom del fitxer: `Go: running s5_1_1.s` diu que RARS ha començat pel fitxer de la subrutina, i ho diu **abans** del missatge d'error. Si el programa s'atura amb un error d'accés a instrucció just en començar, cal mirar aquella línia abans que res.
A la solució de l'exercici hi ha els dos segments de text comparats instrucció per instrucció, per a qui vulgui veure què hi canvia.

Exercici 5.1 (`s5_1_1.s` i `s5_1_2.s`)**.** Traduïu a RV32I el programa C del Codi 5.1, repartint el codi en **dos fitxers** tal com s'indica:

- `s5_1_1.s`: conté únicament la subrutina `abs`.
- `s5_1_2.s`: conté el programa principal, que crida `abs` i escriu el resultat a `resultat`.

Codi 5.1: Subrutina `abs` i programa principal.

`abs.c`

```c
int resultat;

int abs(int x) {
    if (x < 0)
        return -x;
    else
        return x;
}

void main() {
    resultat = abs(-7);
}
```

Tingueu en compte:

- `abs` és **uninivell**: determineu si cal algun registre segur.
- El programa principal és **multinivell**: determineu quins valors, si n'hi ha, travessen la crida i cal preservar.
- Obriu els dos fitxers a RARS, activeu *Settings* → *Assemble all files currently open* i assembleu. Comproveu que `abs` apareix al panell *Labels*.

Comproveu que, en acabar, la variable global `resultat` conté `7`, i que conté `0` si el programa principal crida `abs(0)`.

Solució 5.1. **Anàlisi de registres segurs.**

`abs` és uninivell: no fa cap crida, de manera que no hi ha cap registre en risc de ser destruït; tots els temporals van a registres `t`. BA buit.

El programa principal és multinivell (crida `abs`), però no necessita cap registre segur ni bloc d'activació: no hi ha cap valor generat abans de la crida que s'usi després (el resultat d'`abs` s'usa immediatament en tornar), i com que és el punt d'entrada i mai no torna a cap invocant (acaba amb `ecall`), tampoc cal preservar `ra`. BA buit.

`s5_1_1.s`

```
.globl abs

.text

# abs(int x)
# a0 = x; retorna |x| a a0
abs:
        bge     a0, zero, positiu  # si x >= 0, retorna x directament
        neg     a0, a0             # a0 = -x
positiu:
        ret
```

---

s5_1_2.s

```
.data
resultat: .word 0

.text

        li      a0, -7
        jal     ra, abs         # a0 = abs(-7) = 7

        la      t0, resultat
        sw      a0, 0(t0)       # resultat = 7

        li      a7, 93
        li      a0, 0
        ecall
```

💡 Comprovació pràctica

Obriu `s5_1_1.s` i `s5_1_2.s` a RARS, activeu *Settings* → *Assemble all files currently open* i assembleu. Al panell *Labels* hi ha d'aparèixer `abs` amb la seva adreça: és l'únic símbol declarat `.globl`, i per això és l'únic visible entre mòduls. Amb `abs(0)`, el salt `bge` es pren i es retorna el valor directament.

L'ordre d'assemblatge, vist al segment de text. Feu *File* → *Dump Memory…* (`Ctrl-D`) amb el segment `.text` i el format *Text/Data Segment Window*: escriu en un fitxer el llistat del segment de text, i permet comparar els dos ordres de la pestanya activa (RISC-V 5.1). Per línia d'ordres, és `dump .text SegmentWindow <fitxer>`:

Els dos segments de text, un sota l'altre (a la versió HTML es veuen de costat).

Correcte — pestanya activa: `s5_1_2.s`

```
Address     Code        Basic                    Line  Source
0x00400000  0xff900513  addi x10,x0,0xfffffff9    6    li   a0, -7
0x00400004  0x01c000ef  jal x1,0x0000001c         7    jal  ra, abs
0x00400008  0x0fc10297  auipc x5,0x0000fc10       9    la   t0, resultat
0x0040000c  0xff828293  addi x5,x5,0xfffffff8
0x00400010  0x00a2a023  sw x10,0(x5)             10    sw   a0, 0(t0)
0x00400014  0x05d00893  addi x17,x0,0x0000005d   12    li   a7, 93
0x00400018  0x00000513  addi x10,x0,0            13    li   a0, 0
0x0040001c  0x00000073  ecall                    14    ecall
0x00400020  0x00055463  bge x10,x0,0x00000008     8    bge  a0, zero, positiu
0x00400024  0x40a00533  sub x10,x0,x10            9    neg  a0, a0
```

```
0x00400028  0x00008067  jalr x0,x1,0              11    ret
```

Incorrecte — pestanya activa: `s5_1_1.s`

```
Address     Code        Basic                     Line  Source
0x00400000  0x00055463  bge x10,x0,0x00000008     8     bge  a0, zero, positiu
0x00400004  0x40a00533  sub x10,x0,x10            9     neg  a0, a0
0x00400008  0x00008067  jalr x0,x1,0             11     ret
0x0040000c  0xff900513  addi x10,x0,0xfffffff9    6     li   a0, -7
0x00400010  0xff1ff0ef  jal x1,0xfffffff0         7     jal  ra, abs
0x00400014  0x0fc10297  auipc x5,0x0000fc10       9     la   t0, resultat
0x00400018  0xfec28293  addi x5,x5,0xffffffec
0x0040001c  0x00a2a023  sw x10,0(x5)             10     sw   a0, 0(t0)
0x00400020  0x05d00893  addi x17,x0,0x0000005d   12     li   a7, 93
0x00400024  0x00000513  addi x10,x0,0            13     li   a0, 0
0x00400028  0x00000073  ecall                    14     ecall
```

Dues coses que el bolcat no diu tot sol:

1. **Amb l'ordre incorrecte, la primera instrucció de `.text` és de la subrutina `abs`** (`bge a0, zero, positiu`). És allà on RARS posa el PC, de manera que el programa comença per la primera instrucció d'`abs`, amb `a0` sense inicialitzar.
2. **La mateixa instrucció font canvia de codificació.** `jal ra, abs` és `0x01c000ef` en un cas i `0xff1ff0ef` en l'altre —un salt endavant i un enrere—, i el `la t0, resultat` passa d'`addi x5,x5,-8` a `addi x5,x5,-20` per arribar a la mateixa adreça. Són desplaçaments relatius al PC, corregits segons on ha quedat cada mòdul: és la correcció d'adreces de Secció 3.10.4, aquí visible en dos bolcats.

5.2 Subrutina `descompon`

En aquesta sessió es desenvolupa un programa que converteix un nombre real codificat en **coma fixa** al format de **coma flotant IEEE 754 de precisió simple** (Secció 5.1.3). El programa s'estructura en dues subrutines (`descompon` i `compon`) i un programa principal repartits en dos fitxers, aplicant la compilació separada de la secció anterior.

El **format de coma fixa** que s'usa en aquesta sessió té 32 bits: el bit 31 conté el signe (0 per als positius) i els bits 30–0 representen la magnitud en base 2, amb 19 bits de part entera (bits 30–12) i 12 bits de part fraccionària (bits 11–0):

$$\text{valor}(A) = (-1)^{A_{31}} \times \sum_{i=0}^{30} A_i \cdot 2^{i-12}$$

El **format de coma flotant IEEE 754 de precisió simple** (RISC-V 5.1) té 32 bits: 1 bit de signe (bit 31), 8 bits d'exponent esbiaixat amb biaix 127 (bits 30–23) i 23 bits de fracció (bits 22–0), amb bit ocult implícit (Essencial 5.1).

El programa de conversió s'organitza en dos passos:

`main.c`

```
int signe;
int exponent;
int mantissa;
int cfixa    = 0x87D18A00;
int cflotant;

void main() {
    descompon(cfixa, &signe, &exponent, &mantissa);
    cflotant = compon(signe, exponent, mantissa);
}
```

`cflotant` es declara com a `int`, no `float`: el valor que es construeix és el **patró de bits** de la representació IEEE 754, no un nombre que el compilador hagi de convertir. Si es declarés `float`, l'assignació convertiria el valor enter a coma flotant en lloc de reinterpretar-ne els bits.

La subrutina `descompon` extreu de `cfixa` els tres camps que necessita IEEE 754: el signe, la fracció (els 23 bits que segueixen el bit ocult de la mantissa normalitzada) i l'exponent esbiaixat amb biaix 127 (8 bits):

Codi 5.2: Descomposició d'un nombre de coma fixa en els camps d'IEEE 754.

`descompon.c`

```
void descompon(int cf, int *s, int *e, int *m) {
    int exp;
    *s = (cf < 0);
    cf = cf << 1;
    if (cf == 0)
        exp = 0;
    else {
        exp = 18;
        while (cf >= 0) {
            cf = cf << 1;
            exp--;
        }
        cf = (cf >> 8) & 0x7FFFFF;
        exp = exp + 127;
    }
    *e = exp;
    *m = cf;
}
```

El Codi 5.2 descriu l'algorisme bit a bit: alguns dels desplaçaments sobre valors amb signe (`cf << 1`, `cf >> 8`) queden fora de la definició estricta del llenguatge C, per això en assemblador es fa servir `srli`, el desplaçament lògic, que és el que descriu l'algorisme. (Aquí `srai` donaria el mateix resultat: la màscara `0x7FFFFF` esborra els 8 bits que entren per l'esquerra.)

L'algorisme treballa en tres fases. Primer, separa el signe de la magnitud: codifica el signe a `*s` i elimina el bit de signe de `cf` desplaçant tots els bits una posició a l'esquerra. En aquest punt la coma implícita ha avançat una posició i se situa entre els bits 13 i 12 (el bit 12 és ara el primer bit fraccionari). Segon, si la magnitud no és zero, normalitza: inicialitza l'exponent a 18 (que correspon a situar la coma a la dreta del bit 31) i desplaça `cf` a l'esquerra iterativament fins que el bit 31 sigui `1`, decrementant l'exponent a cada iteració. Tercer, alinea la mantissa normalitzada als 23 bits de menys pes fent un desplaçament a la dreta de 8 posicions i elimina el bit ocult (posició 23) amb una màscara `0x7FFFFF`; finalment, suma 127 a l'exponent per codificar-lo en forma esbiaixada.

Exercici 5.2 (`s5_2_1.md` i `s5_2_1.s`)**.** Traduïu a RV32I la subrutina `descompon`. El fitxer `s5_2_1.s` ha de contenir **únicament** la subrutina, sense cap programa principal.

Abans d'escriure cap instrucció, responeu al fitxer `s5_2_1.md`:

1. Determineu si `descompon` és uninivell o multinivell i justifiqueu-ho.
2. Assigneu cada paràmetre (`cf`, `*s`, `*e`, `*m`) i la variable local (`exp`) a un registre i justifiqueu l'elecció.
3. Dibuixeu el bloc d'activació amb els desplaçaments de cada element.

Comproveu que, amb `cfixa` = `0x87D18A00`, `descompon` deixa `signe` = `0x00000001`, `exponent` = `0x0000008D` (141 en decimal) i `mantissa` = `0x007A3140`.

Solució 5.2. **Pas 1. Uninivell o multinivell.**

`descompon` és **uninivell**: no fa cap crida. No cal preservar `ra`.

Pas 2. Assignació de registres.

Els paràmetres arriben per convenció ABI: `a0 = cf`, `a1 = @s`, `a2 = @e`, `a3 = @m`. Cap valor travessa una frontera de crida, de manera que tots els temporals van a registres `t`. BA buit.

Variable	Registre
`cf` (es modifica al llarg de la subrutina)	`t0`
`exp`	`t1`
`@s`, `@e`, `@m` (paràmetres: adreces de destinació dels resultats)	`a1`, `a2`, `a3`

Pas 3. Bloc d'activació.

`descompon` és uninivell i totes les variables s'assignen a registres: BA buit.

`s5_2_1.s`

```
.globl descompon

.text

# descompon(int cf, int *s, int *e, int *m)
# a0 = cf, a1 = @s, a2 = @e, a3 = @m
descompon:
        # *s = (cf < 0)
        slt     t2, a0, zero       # t2 = 1 si cf < 0, 0 si no
        sw      t2, 0(a1)          # *s = t2

        # cf = cf << 1  (elimina el bit de signe)
        slli    t0, a0, 1          # t0 = cf << 1

        # if (cf == 0) exp = 0
        li      t1, 0              # exp = 0 (cas zero)
        beqz    t0, desa           # si cf == 0, salta a desar
```

```
        # else: exp = 18
        li      t1, 18

        # while (cf >= 0): normalitza
while:
        blt     t0, zero, fiwhile  # si bit 31 és 1 (cf < 0), fi del bucle
        slli    t0, t0, 1          # cf = cf << 1
        addi    t1, t1, -1         # exp--
        j       while
fiwhile:

        # cf = (cf >> 8) & 0x7FFFFF
        # srli (desplaçament lògic): els 8 bits que entren per l'esquerra són
        # 0. Amb srai, que hi propagaria el bit 31 (aquí, 1), el resultat seria
        # el mateix: la màscara 0x7FFFFF esborra aquests bits i el bit ocult.
        srli    t0, t0, 8
        li      t2, 0x7FFFFF
        and     t0, t0, t2         # elimina el bit ocult i bits superiors

        # exp = exp + 127
        addi    t1, t1, 127

desa:
        sw      t1, 0(a2)          # *e = exp
        sw      t0, 0(a3)          # *m = cf

        ret
```

Comprovació pràctica

Per provar `descompon` de manera aïllada, creeu temporalment un programa principal mínim al fitxer `s5_3_1.s` (el de la secció següent) que declari les variables globals i faci la crida:

`s5_3_1.s`

```
.data
signe:  .word   0
exponent: .word 0
mantissa: .word 0
cfixa:  .word   0x87D18A00

.text
        la      a0, cfixa
        lw      a0, 0(a0)
        la      a1, signe
        la      a2, exponent
        la      a3, mantissa
        jal     ra, descompon

        li      a7, 93
        li      a0, 0
        ecall
```

Obriu `s5_2_1.s` i `s5_3_1.s` a RARS amb *Assemble all files currently open* i executeu: en acabar, els valors de `signe`, `exponent` i `mantissa` es llegeixen al panell del segment de dades.

5.3 Subrutina `compon` i programa complet

La subrutina `compon` fa el pas invers: rep els tres camps separats i els combina en un únic valor de 32 bits en format IEEE 754 de precisió simple. El signe s'ha de desplaçar al bit 31, l'exponent als bits 30–23, i la fracció (paràmetre `mantissa`) ja ocupa els bits 22–0:

`compon.c`

```
int compon(int signe, int exponent, int mantissa) {
    return (signe << 31) | (exponent << 23) | mantissa;
}
```

Els tres camps no se superposen (cada un ocupa exclusivament els seus bits), de manera que la combinació amb `or` és directa sense necessitat de màscares.

Exercici 5.3 (`s5_3_1.s`)**.** Substituïu el programa principal temporal de la secció anterior pel programa complet. El fitxer `s5_3_1.s` ha de contenir `compon` i el programa principal, que crida `descompon` (definida a `s5_2_1.s`) i `compon` en seqüència i escriu el resultat a `cflotant`.

Abans d'escriure cap instrucció:

1. Determineu si `compon` és uninivell o multinivell i justifiqueu-ho.
2. Determineu si el programa principal és uninivell o multinivell. Identifiqueu quins valors es generen abans d'una crida i s'usen després, i assigneu-los a registres segurs.
3. Dibuixeu el bloc d'activació del programa principal.

Comproveu que, en acabar, `cflotant` = `0xC6FA3140`, i que, amb els valors inicials de `cfixa` següents, pren el valor esperat:

`cfixa`	Valor decimal	`cflotant` esperat
`0x00000000`	0,0	`0x00000000`
`0x80000000`	−0,0	`0x80000000`
`0x0000CC00`	12,75	`0x414C0000`

Solució 5.3. **Pas 1. `compon`: uninivell o multinivell.**

`compon` és **uninivell**: no fa cap crida. Tots els temporals van a registres `t`; BA buit.

Pas 2. El programa principal: registres segurs.

El programa principal és **multinivell**: crida `descompon` i després `compon`. Els valors generats abans de la primera crida que s'usen després són les adreces de `signe`, `exponent` i `mantissa`, necessàries per llegir els resultats de `descompon` abans de cridar `compon`: calen registres segurs per a elles. `ra`, en canvi, no necessita preservar-se: és el punt d'entrada i mai no torna a cap invocant (acaba amb `ecall`), de manera que el valor que hi deixa cada `jal ra, ...` es pot sobreescriure lliurement en la crida següent.

Dada	Registre
`@signe`	`s0`
`@exponent`	`s1`
`@mantissa`	`s2`

Pas 3. Bloc d'activació del programa principal.

El programa principal no necessita bloc d'activació, tot i usar registres segurs: com que és el punt d'entrada i no torna mai a cap invocant, no hi ha cap valor previ de `s0`–`s2` (ni de `ra`) que calgui preservar i restaurar. BA buit.

`s5_3_1.s`

```
.data
signe:  .word   0
exponent: .word 0
mantissa: .word 0
cfixa:  .word   0x87D18A00
cflotant: .word 0

.text

        la      s0, signe
        la      s1, exponent
        la      s2, mantissa

        # descompon(cfixa, &signe, &exponent, &mantissa)
        la      a0, cfixa
        lw      a0, 0(a0)
        mv      a1, s0
        mv      a2, s1
        mv      a3, s2
        jal     ra, descompon

        # compon(signe, exponent, mantissa)
        lw      a0, 0(s0)       # a0 = signe
        lw      a1, 0(s1)       # a1 = exponent
        lw      a2, 0(s2)       # a2 = mantissa
        jal     ra, compon      # a0 = cflotant

        la      t0, cflotant
        sw      a0, 0(t0)       # cflotant = a0

        li      a7, 93
        li      a0, 0
        ecall

# compon(int signe, int exponent, int mantissa)
# a0 = signe, a1 = exponent, a2 = mantissa
# retorna el valor IEEE 754 de precisió simple a a0
compon:
        slli    t0, a0, 31      # t0 = signe << 31
        slli    t1, a1, 23      # t1 = exponent << 23
        or      t0, t0, t1      # t0 = (signe << 31) | (exponent << 23)
        or      a0, t0, a2      # a0 = t0 | mantissa
        ret
```

> 💡 Comprovació pràctica
>
> Obriu `s5_2_1.s` i `s5_3_1.s` a RARS amb *Assemble all files currently open* i executeu. Per als casos límit, canvieu el valor inicial de `cfixa` a `s5_3_1.s` i torneu a assemblar els dos fitxers abans d'executar.

5.4 Errors de precisió

El programa de conversió desenvolupat en aquesta sessió és correcte per a la majoria de valors, però en alguns casos introdueix una **pèrdua de precisió** inevitable. En aquest exercici se n'analitzen les causes i les condicions en què es produeix.

Exercici 5.4 (`s5_4_1.md`)**.** Responeu les preguntes següents sobre el programa de conversió de coma fixa a coma flotant:

1. Quina condició ha de complir el valor inicial de `cfixa` perquè es produeixi pèrdua de precisió en la conversió? Raoneu la resposta en termes del nombre de bits significatius de cada format.
2. Indiqueu un valor concret de `cfixa` per al qual es produeixi pèrdua de precisió. Indiqueu també el valor de `cflotant` resultant i expliqueu quins bits s'han perdut.
3. En quina instrucció concreta del codi C de `descompon` es produeix la pèrdua de precisió?
4. Quin dels modes d'arrodoniment (Secció 5.1.7) aplica aquest programa de conversió? Justifiqueu-ho.
5. El format de coma fixa d'aquesta sessió té un rang limitat. Indiqueu un nombre positiu que estigui dins del rang del format IEEE 754 de precisió simple però fora del rang del format de coma fixa. Indiqueu també quina és la potència de 2 més petita que compleix aquesta condició.

Solució 5.4. **Pregunta 1.**

El format de coma fixa té 31 bits de magnitud; el format IEEE 754 de precisió simple té 24 bits significatius (23 de fracció més el bit ocult). Després de la normalització, l'operació `cf >> 8` descarta els **8 bits de menys pes** de la mantissa normalitzada (posicions 0–7). Es produeix pèrdua de precisió quan algun d'aquests 8 bits és significatiu (és a dir, val `1`), cosa que passa sempre que la magnitud original tingui bits significatius en posicions prou baixes per caure en aquest rang un cop normalitzada.

De manera equivalent i exacta: sigui b_{alt} la posició del bit `1` de més pes de la magnitud i b_{baix} la posició del bit `1` de menys pes. Es produeix pèrdua de precisió si i només si $b_{\text{alt}} - b_{\text{baix}} + 1 \geq 25$, és a dir, si la magnitud «ocupa» 25 bits o més entre el seu bit significatiu més alt i el més baix (només 24 d'aquests bits caben al format IEEE 754: el bit ocult més els 23 de fracció).

Pregunta 2.

Un exemple clar és `cfixa = 0x7FFFFFFF` (magnitud màxima: tots els bits 30–0 a `1`, valor $\approx 2^{19}$). Després de `cf = cf << 1`, `cf = 0xFFFFFFFE`; el bit 31 ja és `1`, de manera que el bucle de normalització no itera i `exp` es manté a `18`. El desplaçament `cf = (cf >> 8) & 0x7FFFFF` dona `cf = 0x007FFFFF` (la fracció resultant). L'exponent codificat és `exp = 18 + 127 = 145 = 0x91`. El valor `cflotant` resultant

és `0x48FFFFFF`: els 7 bits significatius de menys pes de la magnitud original (tots amb valor `1`), més el `0` introduït pel desplaçament inicial `cf << 1`, s'han descartat, i el nombre representat és lleugerament inferior al valor original.

Pregunta 3.

La pèrdua de precisió es produeix a la instrucció:

`descompon.c`

```
cf = (cf >> 8) & 0x7FFFFF;
```

El desplaçament a la dreta de 8 posicions descarta els 8 bits de menys pes de la mantissa normalitzada. Si algun d'aquests bits era significatiu, la informació es perd de manera irreversible.

Pregunta 4.

El programa aplica el mode d'**arrodoniment a zero** (*round toward zero*): el desplaçament `>> 8` descarta els bits de menys pes sense cap ajust (truncament).

Pregunta 5.

El rang del format de coma fixa d'aquesta sessió és aproximadament $[-(2^{19} - 2^{-12}),\ +(2^{19} - 2^{-12})]$, és a dir, el valor positiu màxim és lleugerament inferior a $2^{19} = 524\,288$. Qualsevol nombre positiu $\geq 2^{19}$ és dins del rang IEEE 754 de precisió simple però fora del rang de coma fixa. El menor nombre potència de 2 que compleix aquesta condició és:

$$2^{19} = 524\,288$$

Laboratori 6: Memòria cau

Lliurements

Cal lliurar les respostes als exercicis següents al campus virtual abans de l'inici de la sessió de laboratori:

Exercici	Nom i format del fitxer
Exercici 6.1	`s6_1_1.md`
Exercici 6.2	`s6_2_1.md`
Exercici 6.3	`s6_2_2.md`
Exercici 6.4	`s6_2_3.md`
Exercici 6.5	`s6_2_4.md`
Exercici 6.6	`s6_3_1.md`
Exercici 6.7	`s6_3_2.md`
Exercici 6.8	`s6_3_3.md`
Exercici 6.9	`s6_4_1.md`
Exercici 6.10	`s6_4_2.s`
Exercici 6.11	`s6_4_3.md`
Exercici 6.12	`s6_4_4.md`
Exercici 6.13	`s6_4_5.md`
Exercici 6.14	`s6_5_1.s`

Lectura prèvia

Abans de fer la sessió, cal haver llegit i entès els continguts següents:

Concepte	On trobar-ho
Localitat temporal i espacial	Secció 7.1.3
Organització de la memòria cau: blocs, línies, etiquetes i bit de validesa	Secció 7.2.2
Terminologia: encert, fallada, taxa d'encerts	Secció 7.2.3
Gestió d'una lectura	Secció 7.2.4
Correspondència directa: descomposició d'adreces	Secció 7.3.1
Memòria cau associativa per conjunts	Secció 7.3.2
Memòria cau completament associativa	Secció 7.3.3
Política de reemplaçament (LRU)	Secció 7.4
Política d'escriptura: gestió en cas d'encert i de fallada	Secció 7.5, Taula 7.3
Exemples de polítiques d'escriptura	Exemple 7.6, Exemple 7.7, Exemple 7.8
Model de temps simplificat	Secció 7.7.1, Essencial 7.2
Impacte de les fallades en el temps d'execució	Secció 7.7.3, Equació 7.4
Recorregut d'un vector per accés seqüencial	Exemple 2.30
L'*stride* i l'eficiència de la memòria cau	Secció 7.1.3

6.1 Descomposició d'adreces

En una memòria cau de **correspondència directa** (Secció 7.3.1), cada adreça generada pel processador es divideix en tres camps: **etiqueta**, **índex** i **desplaçament** (*offset*). El nombre de bits de cada camp depèn de la geometria de la memòria cau:

- **Desplaçament**: selecciona un byte dins del bloc. Si el bloc té $N_P = 2^{n_p}$ paraules de 4 bytes, ocupa $4 \cdot N_P = 2^{n_p+2}$ bytes i el camp de desplaçament té $n_p + 2$ bits.
- **Índex** (n_l bits): identifica la línia de la memòria cau. Si la memòria cau té $N_L = 2^{n_l}$ línies, l'índex té n_l bits.
- **Etiqueta**: la resta de bits de més pes.

A la teoria (Secció 7.2.2) la mida del bloc es dona directament en bytes com $B = 2^b$; aquí, com que els programes operen amb paraules de 4 bytes, l'expressem en paraules, de manera que $B = 4 \cdot N_P = 2^{n_p+2}$ bytes i el desplaçament $b = n_p + 2$. El nombre de línies $N_L = 2^{n_l}$ és el mateix símbol en tots dos llocs.

En aquesta sessió es fa servir una geometria de referència de **32 línies amb blocs de 4 paraules** (tots els exercicis tret que s'indiqui el contrari). Al *Data Cache Simulator* de RARS, el nombre de línies es diu *Number of blocks*. Les variables globals dels programes s'emmagatzemen a partir de l'adreça `0x10010000`.

Exercici 6.1 (`s6_1_1.md`)**.** Considereu una memòria cau de correspondència directa amb **32 línies amb blocs de 4 paraules**.

1. Determineu el nombre de bits dels camps **desplaçament**, **índex** i **etiqueta** per a adreces de 32 bits. Justifiqueu cada valor.
2. El processador genera una referència de lectura a l'adreça `0x1001014C`. Calculeu:
   a. El número del bloc de memòria principal al qual pertany aquesta referència.
   b. L'índex de la línia de memòria cau corresponent.
   c. L'etiqueta corresponent.
3. En el moment de la referència, la línia de memòria cau seleccionada conté el bloc de memòria principal número `0x1001034`. Indiqueu si la referència és un **encert** o una **fallada** i justifiqueu-ho.

Solució 6.1. **Apartat 1. Descomposició de bits.**

El bloc té 4 paraules de 4 bytes = 16 bytes = 2^4 bytes → **desplaçament: 4 bits**.

La memòria cau té 32 línies = 2^5 línies → **índex: 5 bits**.

Etiqueta: $32 - 4 - 5 = 23$ **bits.**

Camp	Bits	Posicions
Etiqueta	23	31–9
Índex	5	8–4
Desplaçament	4	3–0

Apartat 2. Anàlisi de l'adreça `0x1001014C`.

En binari: `0001 0000 0000 0001 0000 0001 0100 1100`

Descomposició per camps:

Camp	Bits extraïts	Valor hex
Etiqueta (bits 31–9)	`0001 0000 0000 0001 0000 000`	`0x080080`
Índex (bits 8–4)	`1 0100`	`0x14` (20 en decimal)
Desplaçament (bits 3–0)	`1100`	`0xC`

a. **Número del bloc d'MP**: l'adreça `0x1001014C` dividida per la mida del bloc (16 bytes) és $\lfloor$`0x1001014C`$/16\rfloor =$ `0x1001014`.
b. **Índex de la línia d'MC**: `0x1001014` mod $32 = 20 =$ `0x14`.
c. **Etiqueta**: `0x080080`.

Apartat 3. Encert o fallada.

La línia d'índex `0x14` conté el bloc `0x1001034` (que hi mapa correctament: `0x1001034` mod $32 = 20$), amb etiqueta `0x080081`. La referència pertany al bloc `0x1001014`, que té el mateix índex `0x14` però etiqueta `0x080080`. Com que les dues etiquetes no coincideixen (`0x080080` $\neq$ `0x080081`), la referència és una **fallada**.

6.2 Recorregut seqüencial d'un vector

En aquesta secció s'analitza el comportament de la memòria cau durant el recorregut seqüencial d'un vector. El programa calcula la suma de tots els elements d'un vector de 100 enters:

`suma_vector.c`

```c
int vector[100];

void main() {
    int i, suma = 0;
    for (i = 0; i < 100; i++)
        suma += vector[i];
}
```

La traducció a RV32I segueix el patró d'accés seqüencial (Exemple 2.30), mantenint un punter `t2` que avança 4 bytes a cada iteració:

Codi 6.1: Suma dels elements d'un vector amb accés seqüencial.

`suma_vector.s`

```
.eqv N, 100

.data
vector: .space  400             # int vector[100]

.text

        mv      t1, zero        # suma = 0
```

```
        la      t2, vector      # t2 = @vector[0]
        li      t3, N
        slli    t3, t3, 2
        add     t3, t2, t3      # t3 = @vector[N] (sentinella)
for:
        bgeu    t2, t3, fifor   # salta si t2 >= sentinella
        lw      t4, 0(t2)       # t4 = vector[i]
        add     t1, t1, t4      # suma += vector[i]
        addi    t2, t2, 4       # t2++ (stride = 4 bytes)
        j       for
fifor:
        li      a7, 93
        li      a0, 0
        ecall
```

Carregueu el programa del Codi 6.1 a RARS i activeu el mòdul *Tools → Data Cache Simulator*. Configureu la memòria cau amb 32 línies (*Number of blocks*) amb blocs de 4 paraules i correspondència directa. Premeu *Connect to Program* abans d'executar.

Exercici 6.2 (`s6_2_1.md`)**.** Considereu la memòria cau de correspondència directa amb **32 línies amb blocs de 4 paraules** i les variables globals emmagatzemades a partir de `0x10010000`.

1. Ompliu la taula següent per a les **10 primeres iteracions** del bucle. Per a cada iteració, indiqueu l'adreça de `vector[i]`, la línia de la memòria cau a què s'accedeix i si la referència és un encert (En) o una fallada (F).

Iteració	Adreça de `vector[i]`	Línia d'MC	En / F
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			

2. Calculeu la taxa d'encerts que s'obté en l'execució completa del programa. Raoneu la resposta a partir del patró de fallades observat.

Verifiqueu les respostes amb el *Data Cache Simulator* de RARS.

Solució 6.2. **Apartat 1.**

`vector[0]` comença a `0x10010000`. Cada element ocupa 4 bytes; el bloc té 16 bytes (4 paraules), de manera que cada bloc conté 4 elements consecutius del vector. L'índex de línia és $\lfloor \text{adreça}/16 \rfloor \bmod 32$.

Iteració	Adreça de `vector[i]`	Línia d'MC	En / F
1	`0x10010000`	0	F
2	`0x10010004`	0	En
3	`0x10010008`	0	En
4	`0x1001000C`	0	En
5	`0x10010010`	1	F
6	`0x10010014`	1	En
7	`0x10010018`	1	En
8	`0x1001001C`	1	En
9	`0x10010020`	2	F
10	`0x10010024`	2	En

Apartat 2.

El vector té 100 elements. Cada bloc d'MC conté 4 elements: la primera referència a cada grup de 4 és una fallada d'arrencada en fred (*cold-start*) i les tres restants són encerts. Hi ha $\lceil 100/4 \rceil = 25$ blocs d'MP accedits, cadascun amb 1 fallada i 3 encerts.

$$\text{taxa d'encerts} = \frac{75}{100} = 0{,}75$$

Exercici 6.3 (`s6_2_2.md`)**.** Considereu el mateix programa i la mateixa geometria de referència (correspondència directa, 32 línies amb blocs de 4 paraules).

1. Si dupliqueu la mida de la memòria cau **doblant el nombre de línies** (64 línies amb blocs de 4 paraules), la taxa d'encerts variarà? Raoneu la resposta.
2. Si dupliqueu la mida de la memòria cau **doblant el nombre de paraules per bloc** (32 línies amb blocs de 8 paraules), la taxa d'encerts variarà? Raoneu la resposta.

Verifiqueu les respostes canviant la geometria al *Data Cache Simulator* i executant el programa de nou. Reinicialitzeu la memòria cau amb *Reset* abans de cada execució.

Solució 6.3. **Apartat 1. Doblar el nombre de línies (64 línies amb blocs de 4 paraules).**

La mida del bloc no canvia (4 paraules = 16 bytes), de manera que cada fallada continua portant un bloc de 4 elements. El patró de fallades és idèntic: 1 fallada cada 4 accessos. La taxa d'encerts **no varia**: 0,75.

Apartat 2. Doblar el nombre de paraules per bloc (32 línies amb blocs de 8 paraules).

Ara cada bloc conté 8 paraules = 32 bytes = 8 elements del vector. La primera referència a cada grup de 8 és una fallada i les 7 restants són encerts. Hi ha $\lceil 100/8 \rceil = 13$ blocs accedits (12 complets + 1 de parcial amb 4 elements). Fallades: 13. Encerts: $100 - 13 = 87$.

$$\text{taxa d'encerts} = \frac{87}{100} = 0{,}87$$

La taxa **millora** perquè blocs més grans aprofiten millor la localitat espacial del recorregut seqüencial.

Exercici 6.4 (`s6_2_3.md`)**.** Si el programa recorregués el vector des del darrer element fins al primer (en ordre decreixent d'índexs), la taxa d'encerts seria la mateixa o diferent respecte al recorregut en ordre creixent? Raoneu la resposta.

Modifiqueu el programa `suma_vector.s` per invertir l'ordre del recorregut i verifiqueu la resposta amb el *Data Cache Simulator*.

Solució 6.4. La taxa d'encerts és **la mateixa**: 0,75.

El patró de fallades no depèn de la direcció del recorregut, sinó de la relació entre la mida del bloc i l'*stride* d'accés (Secció 7.1.3). En ambdós casos, cada bloc d'MC conté 4 elements consecutius del vector: la primera referència a cada bloc és una fallada d'arrencada en fred i les tres restants són encerts, independentment de si el recorregut és creixent o decreixent.

Exercici 6.5 (`s6_2_4.md`)**.** Considereu un sistema amb memòria cau d'instruccions i de dades separades. La memòria cau d'instruccions és ideal (taxa d'encerts = 1). La memòria cau de dades és de correspondència directa amb 32 línies amb blocs de 4 paraules i el model de temps simplificat següent:

- Temps de cicle: $t_c = 10$ ns.
- Temps d'accés en cas d'encert: $t_h = 1$ cicle.
- Temps per copiar un bloc d'MP a MC: $t_{bloc} = 10$ cicles.

Useu el programa `suma_vector.s` de Secció 6.2. Per obtenir el nombre d'instruccions executades i el nombre de fallades, useu les eines *Tools* → *Instruction Statistics* i *Tools* → *Data Cache Simulator* de RARS.

1. Calculeu t_p (penalització per fallada) per a una lectura en política d'escriptura immediata amb assignació (Secció 7.7.1).
2. Determineu el nombre d'instruccions executades (n_{ins}) i el nombre de fallades de la memòria cau de dades ($n_{fallades}$).
3. Calculeu el temps d'execució suposant $\text{CPI}_{ideal} = 2$ (Equació 7.4).
4. Calculeu el temps d'execució si el sistema **no tingués memòria cau de dades**, sabent que el temps d'accés a la memòria principal és $t_{MP} = 6$ cicles (Secció 7.1.2). Calculeu també el guany de rendiment respecte al cas sense memòria cau.

Solució 6.5. **Apartat 1. Càlcul de t_p.**

Per a una lectura amb fallada en política d'escriptura immediata amb assignació (Essencial 7.2):

$$t_p = t_{bloc} + t_h = 10 + 1 = 11 \text{ cicles}$$

Apartat 2. Nombre d'instruccions i fallades.

El bucle `for` executa 5 instruccions per iteració (`bgeu`, `lw`, `add`, `addi`, `j`), és a dir $5 \times 100 = 500$ instruccions, més el `bgeu` final que provoca la sortida (1). Les instruccions d'inicialització són `mv`, `la` i tres per al sentinella (`li`, `slli`, `add`); cal recordar que `la` és una pseudoinstrucció que s'expandeix a **dues** instruccions reals (`auipc` + `addi`), de manera que la inicialització ocupa 6 instruccions. Finalment, la sortida (`li`, `li`, `ecall`) són 3 instruccions més. El nombre total d'instruccions executades és, doncs:

$$n_{ins} = 5 \times 100 + 1 + 6 + 3 = 510 \text{ instruccions}$$

Aquest és el valor que reporta l'eina *Tools → Instruction Statistics* de RARS (compta cada instrucció executada, inclosa l'`ecall` de sortida). El nombre de fallades de la memòria cau de dades s'obté amb el *Data Cache Simulator*: 25 fallades (una per cada bloc de 4 elements, com es calcula a Solució 6.2).

$$n_{fallades} = 25$$

Apartat 3. Temps d'execució amb memòria cau.

Aplicant Equació 7.4:

$$t_{exe} = t_c \cdot (n_{ins} \cdot \text{CPI}_{ideal} + n_{fallades} \cdot t_p)$$

$$t_{exe} = 10 \text{ ns} \cdot (510 \cdot 2 + 25 \cdot 11) = 10 \cdot (1020 + 275) = 12\,950 \text{ ns} \approx 12{,}95 \ \mu\text{s}$$

Apartat 4. Temps d'execució sense memòria cau i guany.

Sense memòria cau, tots els `lw` (100 accessos) sofreixen una penalització $t'_p = t_{MP} - t_h = 6 - 1 = 5$ cicles (Secció 7.7.3):

$$t_{exe,\text{sense cau}} = 10 \cdot (510 \cdot 2 + 100 \cdot 5) = 10 \cdot (1020 + 500) = 15\,200 \text{ ns}$$

$$\text{Guany} = \frac{t_{exe,\text{sense cau}}}{t_{exe,\text{amb cau}}} = \frac{15\,200}{12\,950} \approx 1{,}17$$

Els valors de n_{ins} i $n_{fallades}$ es poden confirmar amb les eines de RARS (*Instruction Statistics* i *Data Cache Simulator*).

6.3 Escriptures i polítiques d'escriptura

En aquesta secció s'analitza com afecten les escriptures al comportament de la memòria cau. El programa ordena **en ordre decreixent** els elements d'un vector de 5 enters pel mètode de selecció (a cada pas selecciona el màxim dels elements restants):

`ordenacio.c`

```c
int vec[5] = {20, -18, 45, 12, -1};

void main() {
    int i, j, aux, max;
    for (i = 0; i < 4; i++) {
        max = i;
        for (j = i + 1; j < 5; j++)
            if (vec[j] > vec[max]) max = j;
        aux = vec[i];
        vec[i] = vec[max];
        vec[max] = aux;
    }
}
```

La traducció a RV32I que es fa servir en aquesta secció és el Codi 6.2:

Codi 6.2: Ordenació per selecció d'un vector de 5 enters.

`ordenacio.s`

```
.eqv N, 5

.data
vec:    .word   20, -18, 45, 12, -1  # int vec[5]

.text

        la      t0, vec              # t0 = @vec[0]
        li      t6, N                # t6 = N
        li      s0, 0                # i = 0
buc_i:
        addi    t5, t6, -1
        bge     s0, t5, fibuc_i      # surt si i >= N-1

        mv      s1, s0               # max = i
        addi    s2, s0, 1            # j = i+1
buc_j:
        bge     s2, t6, fibuc_j      # surt si j >= N

        slli    t1, s2, 2
        add     t1, t0, t1           # t1 = @vec[j]
        lw      t2, 0(t1)            # t2 = vec[j]

        slli    t3, s1, 2
        add     t3, t0, t3           # t3 = @vec[max]
        lw      t4, 0(t3)            # t4 = vec[max]

        ble     t2, t4, no_canvi     # si vec[j] <= vec[max], no actualitza
        mv      s1, s2               # max = j
no_canvi:
        addi    s2, s2, 1            # j++
        j       buc_j
fibuc_j:
        slli    t1, s0, 2
        add     t1, t0, t1           # t1 = @vec[i]
        lw      t2, 0(t1)            # t2 = aux = vec[i]

        slli    t3, s1, 2
```

```
        add     t3, t0, t3          # t3 = @vec[max]
        lw      t4, 0(t3)           # t4 = vec[max]

        sw      t4, 0(t1)           # vec[i] = vec[max]
        sw      t2, 0(t3)           # vec[max] = aux

        addi    s0, s0, 1           # i++
        j       buc_i
fibuc_i:
        li      a7, 93
        li      a0, 0
        ecall
```

Carregueu el programa del Codi 6.2 a RARS i activeu el mòdul *Tools* → *Data Cache Simulator* amb 32 línies (*Number of blocks*) amb blocs de 4 paraules i correspondència directa. El simulador de RARS no té cap opció de política d'escriptura: en comptar encerts i fallades, tracta cada escriptura com una lectura (una escriptura amb fallada porta el bloc a la memòria cau), que és el comportament de la política **amb assignació** d'aquest exercici, i no compta el trànsit cap a la memòria principal.

Exercici 6.6 (`s6_3_1.md`)**.** Considereu la memòria cau de correspondència directa amb **32 línies amb blocs de 4 paraules** i política d'**escriptura immediata amb assignació** (Exemple 7.6). Les variables globals comencen a `0x10010000`.

1. Per a cada iteració del bucle exterior (`i = 0, 1, 2, 3`), empliu la taula amb els accessos a elements de `vec`: índex de l'element accedit, tipus d'accés (L = lectura, E = escriptura) i resultat (En = encert, F = fallada).
2. Compteu el nombre total d'accessos i d'encerts per a cada iteració del bucle exterior.
3. Calculeu la taxa d'encerts global de l'execució completa del programa.

Verifiqueu la taxa d'encerts amb el *Data Cache Simulator* de RARS. Comproveu també que, en acabar, el vector queda ordenat de gran a petit: `{45, 20, 12, -1, -18}`.

Solució 6.6. Els 5 elements de `vec` ocupen $5 \times 4 = 20$ bytes. Com que $20 > 16$ (la mida d'un bloc), `vec` ocupa **2 blocs consecutius**: el bloc 0 conté `vec[0]`–`vec[3]` i el bloc 1 conté `vec[4]`.

Com que `vec` comença a `0x10010000`:

Element	Adreça	Línia d'MC
`vec[0]`	`0x10010000`	0
`vec[1]`	`0x10010004`	0
`vec[2]`	`0x10010008`	0
`vec[3]`	`0x1001000C`	0
`vec[4]`	`0x10010010`	1

Iteració `i = 0` (`max` iniciat a 0; bucle `j` d'1 a 4; intercanvi `vec[0]` ↔ `vec[max]`):

Accés	Element	L/E	En/F
Lectura `vec[1]` (j=1)	1	L	F
Lectura `vec[max=0]` (j=1)	0	L	En
Lectura `vec[2]` (j=2)	2	L	En
Lectura `vec[max]` (j=2)	0	L	En
Lectura `vec[3]` (j=3)	3	L	En
Lectura `vec[max]` (j=3)	2	L	En
Lectura `vec[4]` (j=4)	4	L	F
Lectura `vec[max]` (j=4)	2	L	En
Lectura `vec[i=0]` (intercanvi)	0	L	En
Lectura `vec[max]` (intercanvi)	2	L	En
Escriptura `vec[i=0]`	0	E	En
Escriptura `vec[max]`	2	E	En

La lectura de `vec[max]` usa el valor de `max` **abans** de la seva possible actualització. Amb `vec = {20, -18, 45, 12, -1}`, `max = 0` fins que a `j = 2` es detecta el nou màxim `vec[2] = 45`; a partir d'aleshores `max = 2` durant la resta de la iteració (per això les lectures de `vec[max]` a `j = 1` i `j = 2` accedeixen a l'element 0, i les posteriors, a l'element 2).

> 💡 Comprovació pràctica
>
> Executeu `ordenacio.s` a RARS amb el *Data Cache Simulator* configurat amb 32 línies (*Number of blocks*) amb blocs de 4 paraules i correspondència directa (les escriptures hi compten com amb assignació). En acabar l'execució, el simulador mostra el nombre total de referències, encerts i fallades.
> Per seguir l'evolució accés a accés, poseu un punt de ruptura a la primera instrucció del bucle interior i avanceu pas a pas observant els canvis a la vista *Cache Block Table* del simulador.

Apartat 2. Accessos i encerts per iteració del bucle exterior.

A cada iteració `i`, el bucle interior fa $2 \times (4 - i)$ lectures (dues per cada valor de `j`, d'`i+1` a 4) i l'intercanvi hi afegeix 4 accessos més (2 lectures + 2 escriptures):

Iteració `i`	Accessos	Fallades	Encerts
0	12	2	10
1	10	0	10
2	8	0	8
3	6	0	6
Total	**36**	**2**	**34**

Apartat 3. Taxa d'encerts global.

Les úniques fallades són les dues d'arrencada en fred de la iteració `i = 0` (primer accés al bloc 0, amb `vec[1]`, i al bloc 1, amb `vec[4]`); a partir d'aleshores tots dos blocs romanen a la memòria cau (vector de 2 blocs, cap conflicte amb 32 línies). Per tant:

$$\text{taxa d'encerts} = \frac{34}{36} \approx 0{,}944$$

El *Data Cache Simulator* ha de reportar 36 referències i 2 fallades.

Exercici 6.7 (`s6_3_2.md`)**.** Considereu el mateix programa i la mateixa geometria de memòria cau (32 línies amb blocs de 4 paraules).

1. Com canviaria la taxa d'encerts si la política d'escriptura fos **immediata sense assignació** (Exemple 7.7)? Raoneu la resposta indicant quins accessos canvien de resultat.
2. Com canviaria la taxa d'encerts si la política d'escriptura fos **retardada amb assignació** (Exemple 7.8)? Raoneu la resposta.

Solució 6.7. **Apartat 1. Escriptura immediata sense assignació.**

En política d'escriptura immediata sense assignació, una escriptura amb fallada **no porta el bloc a MC**: l'escriptura es fa directament a MP i el bloc no s'emmagatzema a la memòria cau. En el programa d'ordenació, totes les escriptures (`sw`) es produeixen sobre elements que ja han estat llegits prèviament en la mateixa iteració del bucle exterior, de manera que el bloc ja és present a MC en el moment de l'escriptura (és un encert). Per tant, **la taxa d'encerts no varia** respecte a la política immediata amb assignació: el cas de fallada en escriptura no es produeix en aquest programa.

Apartat 2. Escriptura retardada amb assignació.

En política d'escriptura retardada amb assignació, les escriptures es fan únicament a MC (sense actualitzar MP immediatament) i el bloc es marca com a modificat (bit D). Quan el bloc és expulsat, s'escriu a MP. En el programa d'ordenació, el vector és tan petit (2 blocs) que cap bloc és expulsat durant l'execució. Les escriptures sobre blocs presents a MC (encerts) no generen cap penalització addicional. La taxa d'encerts és, per tant, **la mateixa** que en les polítiques anteriors.

Exercici 6.8 (`s6_3_3.md`)**.** Considereu ara un vector de **64 elements** (en lloc de 5), mantenint la mateixa geometria de memòria cau (correspondència directa, 32 línies amb blocs de 4 paraules) i la política d'escriptura immediata amb assignació.

1. Quantes fallades es produirien en l'execució del programa d'ordenació adaptat a 64 elements? Justifiqueu la resposta indicant si es produeixen fallades d'arrencada en fred, de capacitat o de conflicte (Secció 7.8, Essencial 7.4).
2. Sense que augmenti el nombre de fallades, fins a quin nombre mínim de línies es podria reduir la memòria cau? Raoneu la resposta.

Modifiqueu `ordenacio.s` per adaptar-lo a un vector de 64 elements: poseu `.eqv N, 64` i substituïu la declaració per `vec: .space 256` (els valors concrets no afecten el nombre de fallades). Verifiqueu les respostes amb el *Data Cache Simulator*.

Solució 6.8. **Apartat 1. Nombre de fallades amb 64 elements.**

El vector de 64 enters ocupa $64 \times 4 = 256$ bytes, és a dir 16 blocs de 16 bytes. Amb 32 línies disponibles, tots els blocs del vector caben simultàniament a la memòria cau: **no hi ha fallades de conflicte ni de capacitat**.

Les úniques fallades són **d'arrencada en fred**: la primera referència a cada un dels 16 blocs. En el recorregut del bucle interior, cada bloc és accedit per primera vegada en ordre creixent d'índex, i a partir d'aquell moment roman a la memòria cau durant tota l'execució (no és expulsat perquè no hi ha conflictes). El nombre total de fallades és, doncs:

$$n_{fallades} = 16$$

Apartat 2. Nombre mínim de línies.

Perquè no augmenti el nombre de fallades (que ha de continuar sent només les 16 d'arrencada en fred), cal que els 16 blocs del vector puguin conviure a la memòria cau sense expulsar-se. Al llarg de l'execució, qualsevol bloc expulsat es tornaria a referenciar en una iteració posterior del bucle exterior i provocaria una fallada addicional; per tant, els 16 blocs han de residir simultàniament a la memòria cau. Amb una memòria cau de correspondència directa de **16 línies**, els 16 blocs consecutius del vector (0–15) es mapen a 16 línies diferents (bloc $b \to$ línia $b \bmod 16$), de manera que no hi ha conflictes. El mínim és, doncs, **16 línies**.

6.4 Accés a múltiples estructures

En aquesta secció s'analitza com la presència de múltiples estructures de dades pot provocar **conflictes** a la memòria cau (Secció 7.8): dues estructures que es mapen a les mateixes línies s'expulsen mútuament, degradant la taxa d'encerts fins i tot quan la memòria cau tindria capacitat suficient per contenir-les totes.

El programa calcula la suma element a element de dos vectors i desa el resultat en un tercer:

`suma_vectors.c`

```c
int X[128], Y[128], Z[128];

void main() {
    int i;
    for (i = 0; i < 128; i++)
        Z[i] = X[i] + Y[i];
}
```

La traducció a RV32I fa servir tres punters que avancen en paral · lel:

Codi 6.3: Suma de dos vectors en un tercer, amb tres punters.

`suma_vectors.s`

```
.eqv N, 128

.data
X:      .space  512             # int X[128]
Y:      .space  512             # int Y[128]
Z:      .space  512             # int Z[128]

.text

        la      t0, X           # t0 = @X[0]
        la      t1, Y           # t1 = @Y[0]
        la      t2, Z           # t2 = @Z[0]
        li      t3, N
        slli    t3, t3, 2
        add     t3, t0, t3      # t3 = @X[N] (sentinella)
for:
        bgeu    t0, t3, fifor
        lw      t4, 0(t0)       # t4 = X[i]
        lw      t5, 0(t1)       # t5 = Y[i]
        add     t4, t4, t5      # t4 = X[i] + Y[i]
        sw      t4, 0(t2)       # Z[i] = X[i] + Y[i]
        addi    t0, t0, 4       # X[i+1]
        addi    t1, t1, 4       # Y[i+1]
        addi    t2, t2, 4       # Z[i+1]
        j       for
fifor:
        li      a7, 93
        li      a0, 0
        ecall
```

Carregueu el programa del Codi 6.3 a RARS i activeu el *Data Cache Simulator*.

Exercici 6.9 (`s6_4_1.md`)**.** Considereu una memòria cau de correspondència directa amb **16 línies amb blocs de 4 paraules** i política d'**escriptura immediata amb assignació**. Les variables globals comencen a `0x10010000`.

1. Calculeu la mida en blocs de cada vector. Determineu a quines línies d'MC es mapen els blocs d'`X`, `Y` i `Z`. Identifiqueu si hi ha conflictes entre els tres vectors i expliqueu-ne la causa.
2. Calculeu la taxa d'encerts esperada. Justifiqueu el resultat a partir de l'anàlisi de conflictes.

Verifiqueu la resposta amb el *Data Cache Simulator* configurat amb 16 línies (*Number of blocks*) amb blocs de 4 paraules.

Solució 6.9. **Apartat 1. Mapatge i conflictes.**

Cada vector té 128 enters × 4 bytes = 512 bytes = 32 blocs de 16 bytes. La memòria cau té 16 línies. El mapatge en correspondència directa assigna el bloc n d'MP a la línia n mod 16. Per simplicitat, numerem els blocs de manera relativa a l'inici d'`X` (el bloc que conté `X[0]` és el bloc 0).

`X` comença a `0x10010000` → bloc MP 0 a la línia 0, bloc MP 1 a la línia 1, …, bloc MP 15 a la línia 15, bloc MP 16 a la línia 0, …

`Y` comença a `0x10010000 + 512 = 0x10010200` → bloc MP 32 a la línia 0, bloc MP 33 a la línia 1, … Els blocs d'`Y` es mapen exactament a les **mateixes línies** que els d'`X`.

`Z` comença a `0x10010000 + 1024 = 0x10010400` → bloc MP 64 a la línia 0, bloc MP 65 a la línia 1, … Els blocs de `Z` es mapen també a les **mateixes línies** que `X` i `Y`.

Els tres vectors es mapen a les mateixes 16 línies: **conflicte total**. A cada iteració del bucle, l'accés a `X[i]`, `Y[i]` i `Z[i]` accedeix a la mateixa línia d'MC, expulsant el bloc anterior en cada accés.

Apartat 2. Taxa d'encerts.

A cada iteració es fan 3 accessos (lectura `X[i]`, lectura `Y[i]`, escriptura `Z[i]`), cadascun a la mateixa línia d'MC. Seqüència per a la iteració i:

1. `lw X[i]`: fallada (el bloc d'`X[i]` no és a MC o ha estat expulsat).
2. `lw Y[i]`: fallada (expulsa el bloc d'`X[i]` que s'acaba de portar).
3. `sw Z[i]`: fallada en escriptura amb assignació (expulsa el bloc d'`Y[i]`).

Els 3 accessos de cada iteració són fallades. Amb 128 iteracions i 3 accessos cadascuna:

$$\text{taxa d'encerts} = \frac{0}{384} = 0{,}00$$

> 💡 Comprovació pràctica
>
> Configureu el *Data Cache Simulator* amb 16 línies (*Number of blocks*) amb blocs de 4 paraules i correspondència directa (les escriptures hi compten com amb assignació). Executeu `suma_vectors.s` i observeu que la taxa d'encerts és exactament **0**: el simulador només comptabilitza els accessos a dades (les 384 lectures i escriptures als vectors), i totes són fallades.

Exercici 6.10 (`s6_4_2.s`)**.** La taxa d'encerts nul · la de l'exercici anterior es deu al fet que els tres vectors es mapen exactament a les mateixes línies. Una manera d'eliminar els conflictes sense canviar l'algorisme és afegir **farciment** (*padding*) entre vectors: un espai buit que desplaça l'adreça base del vector següent i trenca l'alineació que causa el conflicte.

Modifiqueu la declaració de dades de `suma_vectors.s` afegint farciment entre els vectors de manera que la taxa d'encerts sigui **0,75**. El fitxer resultant ha de ser `s6_4_2.s`.

Justifiqueu al mateix fitxer (com a comentaris) el valor de farciment escollit i per què produeix la taxa d'encerts desitjada.

Verifiqueu el resultat amb el *Data Cache Simulator*.

Solució 6.10. Amb 16 línies i blocs de 4 paraules (16 bytes), la memòria cau cobreix $16 \times 16 = 256$ bytes. Els conflictes es produeixen perquè `X`, `Y` i `Z` estan separats exactament 512 bytes = 32 blocs, múltiple del nombre de línies (16), de manera que es mapen a les mateixes línies.

Per aconseguir una taxa d'encerts de 0,75 cal que 3 de cada 4 accessos siguin encerts. Això s'obté quan els tres vectors es mapen a línies **diferents**: en cada iteració del bucle, els blocs d'`X[i]`, `Y[i]` i `Z[i]` van a línies distintes, de manera que no s'expulsen mútuament. Amb 128 elements i blocs de 4 paraules,

hi ha 32 blocs per vector. La primera referència a cada bloc és una fallada d'arrencada en fred: 32 fallades per vector × 3 vectors = 96 fallades. El total d'accessos és 384. Taxa d'encerts:

$$\text{taxa d'encerts} = \frac{384 - 96}{384} = \frac{288}{384} = 0{,}75$$

Per separar els vectors cal que `X`, `Y` i `Z` es mapin a línies diferents. Com que `X` comença en una adreça alineada a bloc, un farciment de `PAD` bytes (múltiple de 16) desplaça la línia base d'`Y` en `PAD/16` línies i la de `Z` en `2·PAD/16` línies (el doble, perquè `Z` acumula els dos farciments). Cal, doncs, que `0`, `PAD/16` i `2·PAD/16` siguin **tres línies diferents mòdul 16**. Compte: no serveix qualsevol valor —per exemple, `PAD = 128` desplaçaria `Z` en $2 \times 128/16 = 16 \equiv 0$ línies, fent-la coincidir de nou amb `X`. Un farciment d'un sol bloc (`PAD = 16`) porta `X`, `Y` i `Z` a les línies base 0, 1 i 2, que són diferents, i és suficient:

`s6_4_2.s`

```
.eqv N, 128
.eqv PAD, 16                    # 1 bloc de farciment entre vectors

.data
X:      .space  512             # int X[128]: 512 bytes, línies 0–15
        .space  PAD             # farciment: desplaça Y 1 bloc
Y:      .space  512             # int Y[128]: 512 bytes, línies 1–0 (rotació)
        .space  PAD             # farciment: desplaça Z 1 bloc addicional
Z:      .space  512             # int Z[128]: 512 bytes, línies 2–1 (rotació)

.text

        la      t0, X
        la      t1, Y
        la      t2, Z
        li      t3, N
        slli    t3, t3, 2
        add     t3, t0, t3
for:
        bgeu    t0, t3, fifor
        lw      t4, 0(t0)
        lw      t5, 0(t1)
        add     t4, t4, t5
        sw      t4, 0(t2)
        addi    t0, t0, 4
        addi    t1, t1, 4
        addi    t2, t2, 4
        j       for
fifor:
        li      a7, 93
        li      a0, 0
        ecall
```

> 💡 Comprovació pràctica
>
> Executeu `s6_4_2.s` amb el *Data Cache Simulator* (16 línies amb blocs de 4 paraules, correspondència directa). La taxa d'encerts ha de ser 0,75. Comproveu també que el resultat al vector `Z` coincideix amb el de `suma_vectors.s` original (els valors de `Z` han de ser iguals en ambdós casos, ja que el farciment no afecta la lògica del programa).

Exercici 6.11 (`s6_4_3.md`)**.** Considereu ara una memòria cau **completament associativa** (Secció 7.3.3) amb **16 línies amb blocs de 4 paraules** i política LRU (Secció 7.4).

Calculeu la taxa d'encerts en l'execució del programa `suma_vectors.s`. Raoneu la resposta indicant si es produeixen fallades de conflicte, de capacitat o únicament d'arrencada en fred.

Verifiqueu la resposta amb el *Data Cache Simulator* configurat en mode completament associatiu.

Solució 6.11. En una memòria cau completament associativa, qualsevol bloc d'MP pot anar a qualsevol línia: no hi ha mapatge fix i, per tant, **no hi ha fallades de conflicte**. Podria semblar que hi haurà fallades de capacitat, ja que els tres vectors sumen $3 \times 32 = 96$ blocs mentre que la memòria cau només en té 16 ($96 > 16$); però el que determina les fallades de capacitat no és l'ocupació total del programa, sinó el *working set* instantani (els blocs realment vius en un moment donat).

A cada iteració del bucle s'accedeix a un bloc d'`X`, un d'`Y` i un de `Z`. Amb política LRU i 16 línies, els blocs actius en qualsevol moment són els tres blocs corrents (un per vector). Com que $3 < 16$, els tres blocs sempre caben a la memòria cau simultàniament sense expulsar-se: la primera referència a cada bloc és una fallada d'arrencada en fred i les tres referències restants al mateix bloc (els altres 3 elements) són encerts.

El comportament és idèntic al d'accés seqüencial sense conflictes: 32 fallades per vector $\times$ 3 vectors $=$ 96 fallades, $384 - 96 = 288$ encerts.

$$\text{taxa d'encerts} = \frac{288}{384} = 0{,}75$$

> Comprovació pràctica
>
> Configureu el *Data Cache Simulator* en mode completament associatiu (seleccioneu *Fully Associative* al camp *Placement Policy*) amb 16 línies (*Number of blocks*) amb blocs de 4 paraules i política de reemplaçament LRU. Executeu `suma_vectors.s` i verifiqueu que la taxa d'encerts és 0,75.

Exercici 6.12 (`s6_4_4.md`)**.** Considereu una memòria cau **associativa per conjunts** (Secció 7.3.2) amb blocs de 4 paraules i **16 conjunts**.

Quin ha de ser el **grau mínim d'associativitat** (nombre de vies per conjunt) perquè la taxa d'encerts del programa `suma_vectors.s` sigui superior a 0? Raoneu la resposta sense usar el simulador.

Verifiqueu la resposta amb el *Data Cache Simulator* ajustant el camp *Set size* a la potència de 2 igual o immediatament superior al grau calculat (el simulador només accepta potències de 2).

Solució 6.12. Amb 16 conjunts i blocs de 4 paraules (16 bytes), el mapatge és: bloc d'MP $n \to$ conjunt n mod 16. Els tres vectors ocupen 32 blocs cadascun i es mapen als mateixos 16 conjunts amb el mateix patró que en correspondència directa: els blocs i, $i+32$ i $i+64$ (un de cada vector) van tots al conjunt i mod 16.

Amb grau 1 (correspondència directa), els 3 blocs que competeixen per cada conjunt s'expulsen contínuament: taxa d'encerts = 0.

Amb grau **2**, cada conjunt té 2 vies. A cada iteració hi ha 3 blocs actius que competeixen pel mateix conjunt (un d'`X`, un d'`Y`, un de `Z`): $3 > 2$, de manera que amb LRU un dels tres blocs és expulsat a cada accés. Encara hi haurà conflictes i la taxa d'encerts continuarà sent 0.

Amb grau **3**, cada conjunt té 3 vies: els 3 blocs actius ($X[i]$, $Y[i]$, $Z[i]$) caben simultàniament al mateix conjunt sense expulsar-se. La primera referència a cada bloc és una fallada d'arrencada en fred i les tres referències restants al bloc són encerts. La taxa d'encerts és $0{,}75 > 0$.

El **grau mínim** és, per tant, **3**.

> Comprovació pràctica
>
> El simulador RARS només accepta potències de 2. Configureu *Set size = 4* (la potència de 2 immediatament superior a 3) amb *Number of blocks = 64* (16 conjunts × 4 vies) i 4 paraules/bloc. Executeu `suma_vectors.s` i verifiqueu que la taxa d'encerts és 0,75. Comproveu també amb *Set size = 2* i *Number of blocks = 32* (16 conjunts × 2 vies) que la taxa és 0.

Exercici 6.13 (`s6_4_5.md`)**.** En aquest exercici classificareu les fallades del programa `suma_vectors.s` segons la taxonomia de les **tres C** (Secció 7.8, Essencial 7.4) usant el *Runtime Log* del *Data Cache Simulator*.

Activeu la casella *Runtime Log* (*Enabled*) del *Data Cache Simulator*: per a cada accés, el registre indica l'adreça, l'etiqueta i si és un encert o una fallada. Tingueu present que **RARS no etiqueta el tipus de fallada**; la classificació és un pas de raonament vostre, a partir del mètode següent:

- Una fallada és **d'arrencada en fred** si és la **primera** referència a aquell bloc: la primera vegada que apareix al registre una adreça del bloc. El bloc s'identifica pel número de bloc, l'adreça sense els 4 bits del desplaçament (`adreça >> 4`), i no per l'etiqueta sola, que en correspondència directa comparteixen blocs diferents.
- Les fallades que no són d'arrencada en fred són de **conflicte** o de **capacitat**. Per distingir-les, es compara amb una execució en què no hi pugui haver conflictes: una memòria cau **completament associativa** de la mateixa capacitat. Les fallades que desapareixen en passar a completament associativa eren de **conflicte**; les que persisteixen (més enllà de les d'arrencada en fred) són de **capacitat**.

1. Executeu `suma_vectors.s` amb la memòria cau de **correspondència directa** de 16 línies amb blocs de 4 paraules (la de Exercici 6.9). Amb el *Runtime Log*, determineu quantes de les fallades són d'arrencada en fred. (Pista: quants blocs diferents s'accedeixen en total?)
2. Executeu ara el mateix programa amb la memòria cau **completament associativa** de 16 línies (la de Exercici 6.11). Quantes fallades queden? De quin tipus són?
3. A partir dels dos resultats, feu el desglossament complet de les fallades de la memòria cau de correspondència directa en fallades d'arrencada en fred, de conflicte i de capacitat.

Solució 6.13. **Apartat 1. Fallades de la memòria cau de correspondència directa i d'arrencada en fred.**

Amb la memòria cau de correspondència directa (Solució 6.9), els 384 accessos són fallades. Ara bé, d'arrencada en fred només ho són les primeres referències a cada bloc diferent. Cada vector ocupa 32 blocs, i són tres vectors sense superposició, de manera que s'accedeixen $3 \times 32 = 96$ blocs diferents:

$$n_{\text{obligatòries}} = 96$$

Apartat 2. Fallades de la memòria cau completament associativa.

Amb la memòria cau completament associativa (Solució 6.11), la taxa d'encerts és 0,75: hi ha **96 fallades**. Com que en una memòria cau completament associativa no hi pot haver fallades de conflicte, i el *working set* instantani (3 blocs) és molt inferior a la capacitat (16 línies), tampoc no n'hi ha de capacitat: **les 96 fallades són totes d'arrencada en fred**.

Apartat 3. Desglossament de les fallades de la memòria cau de correspondència directa.

Les 96 fallades de la memòria cau completament associativa són el nombre irreductible de fallades d'arrencada en fred. Les fallades addicionals de la memòria cau de correspondència directa (les que desapareixen en passar a completament associativa) són de **conflicte**:

Tipus de fallada	Correspondència directa	Completament associativa
Obligatòria	96	96
Conflicte	288	0
Capacitat	0	0
Total	**384**	**96**

El cas confirma que aquí la degradació de la correspondència directa (taxa 0 enfront de 0,75) es deu **íntegrament a fallades de conflicte**, no de capacitat: la memòria cau tindria prou espai per als 3 blocs actius, però el mapatge fix els força a la mateixa línia.

6.5 Optimització: *loop tiling*

El programa de suma de vectors de la secció anterior té un patró d'accés favorable: recorre les tres estructures de manera estrictament seqüencial i en paral · lel, de manera que la localitat espacial és màxima. No tots els programes tenen aquest comportament. En alguns casos, el patró d'accés provoca que elements carregats a la memòria cau siguin expulsats abans de ser reutilitzats, malbaratant la localitat temporal.

El programa del Codi 6.4 calcula el producte d'una matriu per un vector:

Codi 6.4: Producte d'una matriu per un vector.

`matvec.c`

```c
int V1[16], M[16][16], V2[16];

void main() {
    int i, j, tmp;
    for (i = 0; i < 16; i++) {
        tmp = 0;
        for (j = 0; j < 16; j++)
            tmp += M[i][j] * V2[j];
        V1[i] = tmp;
    }
}
```

`M` s'emmagatzema per files (*row-major order*, Secció 4.5.1): els elements `M[i][0]`, `M[i][1]`, …, `M[i][15]` són contigus en memòria. El bucle interior recorre la fila `i` de `M` i tot el vector `V2` de manera seqüencial. A cada iteració del bucle exterior, `V2` es llegeix sencer (16 elements). Si la memòria cau pot mantenir alhora `V2`, la fila de `M` i el bloc de `V1` en ús, sense que es facin conflicte, els accessos a `V2` a partir de la segona iteració exterior són encerts. Si la memòria cau és massa petita, `V2` és expulsat a cada iteració i s'ha de tornar a carregar des de la memòria principal.

La traducció a RV32IM de la versió original és:

`matvec.s`

```
.eqv NI, 16
.eqv NJ, 16

.data
        .align 2
V1:     .space  64              # int V1[16]
        .align 2
M:      .space  1024            # int M[16][16]
        .align 2
V2:     .space  64              # int V2[16]

.text

        la      s3, V1          # s3 = @V1[0] (invariant)
        la      s4, M           # s4 = @M[i][0] (punter de fila)
        la      s5, V2          # s5 = @V2[0] (invariant)
        li      s2, NI          # s2 = sentinella i (invariant)
        li      s6, NJ
        slli    s6, s6, 2       # s6 = NJ*4: mida d'una fila en bytes (invariant)

        li      s1, 0           # i = 0
buc_i:
        bge     s1, s2, fibuc_i

        li      t1, 0           # tmp = 0
        mv      t2, s4          # t2 = punter a M[i][j] (j = 0)
        mv      t4, s5          # t4 = punter a V2[j] (j = 0)
        add     t5, s4, s6      # t5 = sentinella j: @M[i][NJ]
buc_j:
        bgeu    t2, t5, fibuc_j
        lw      t6, 0(t2)       # t6 = M[i][j]
        lw      a0, 0(t4)       # a0 = V2[j]
        mul     t6, t6, a0      # t6 = M[i][j] * V2[j]
        add     t1, t1, t6      # tmp += M[i][j] * V2[j]
```

```
        addi    t2, t2, 4       # punter M avança (stride = 4)
        addi    t4, t4, 4       # punter V2 avança (stride = 4)
        j       buc_j
fibuc_j:
        slli    t6, s1, 2
        add     t6, s3, t6      # t6 = @V1[i]
        sw      t1, 0(t6)       # V1[i] = tmp
        add     s4, s4, s6      # punter de fila: @M[i+1][0]
        addi    s1, s1, 1       # i++
        j       buc_i
fibuc_i:
        li      a7, 93
        li      a0, 0
        ecall
```

L'optimització ***loop tiling*** (també anomenada ***loop blocking*** o *tiling*) reorganitza el recorregut per aprofitar la localitat temporal de `V2`: en lloc de multiplicar tota una fila de `M` per tot `V2`, es processen **grups de 4 columnes** alhora. Primer s'itera per totes les files amb les columnes 0–3, després amb les columnes 4–7, i així fins a completar les 16 columnes. D'aquesta manera, els 4 elements de `V2` implicats en cada grup caben en un únic bloc de memòria cau, que es pot reutilitzar en totes les files si cap altre bloc no l'expulsa:

`matvec_tiling.c`

```
int V1[16];             // global (inicialitzat a 0 en C)
int M[16][16], V2[16];

void main() {
    int i, j, k, tmp;
    for (k = 0; k < 4; k++)
        for (i = 0; i < 16; i++) {
            tmp = 0;
            for (j = 0; j < 4; j++)
                tmp += M[i][k * 4 + j] * V2[k * 4 + j];
            V1[i] += tmp;
        }
}
```

El bucle `k` itera els 4 grups de columnes (0–3, 4–7, 8–11, 12–15). Per a cada bloc, el bucle `i` recorre totes les files de `M` acumulant parcialment el resultat a `V1[i]`. El bucle `j` recorre els 4 elements del bloc de columnes. Amb blocs de 4 elements (= 1 bloc de memòria cau de 4 paraules), els accessos a `V2[k*4 + j]` són encerts a partir del primer element del bloc, perquè els 4 elements es carreguen junts en la primera fallada.

Carregueu `matvec.s` (versió original) a RARS i mesureu amb el *Data Cache Simulator* la seva taxa d'encerts per a diverses geometries de memòria cau; registrareu aquests valors i els compareu amb els del programa amb *tiling* a la Taula 6.1 de l'exercici.

Exercici 6.14 (`s6_5_1.s`)**.** Traduïu a RV32IM el programa `matvec_tiling.c`. El fitxer `s6_5_1.s` ha de contenir el programa complet.

Abans d'escriure cap instrucció:

1. Identifiqueu totes les variables del programa (`i`, `j`, `k`, `tmp`, els punters o adreces base que calgui) i assigneu-les a registres. Justifiqueu si useu registres `t` o `s`.
2. Determineu l'*stride* d'accés a `M[i][k*4 + j]` en funció de com varien `i`, `j` i `k`. Indiqueu quin dels tres índexs varia més ràpidament al bucle interior i com avança el punter corresponent.

A continuació, escriviu el codi RV32IM complet.

Comproveu que, amb `M[i][j] = 1` per a tots `i`, `j` i `V2[j] = j`, cada element de `V1` val `120`, com amb el programa original no optimitzat.

3. Amb el *Data Cache Simulator*, mesureu la taxa d'encerts del programa amb *tiling* per a les geometries de la taula següent i compareu-les amb les del programa original `matvec.s`:

Taula 6.1: Taxa d'encerts del programa amb *tiling* per a diferents geometries de memòria cau

Capacitat	CD	Assoc. grau 2	Assoc. grau 4	Assoc. grau 8	Comp. assoc.
8 línies					
16 línies					
32 línies					
64 línies					
128 línies					

Solució 6.14. **Pas 1. Assignació de registres.**

El programa és **pla**: no fa cap crida, de manera que tot el codi és al bloc principal i no hi ha context que preservar. Tots els registres són `t` tret dels que han de sobreviure als tres nivells de bucle imbricat. Els valors que creuen el bucle interior (`i`, `k`, les adreces base) han de ser en registres que el bucle interior no destrueixi. Com que no hi ha crides, es poden usar registres `t` sempre que el compilador (o el programador) garanteixi que el bucle interior no els sobreescriu. Per claredat i robustesa s'usen `s` per als índexs dels bucles exteriors:

Variable	Registre	Justificació
`k`	`s0`	índex del bucle més exterior; ha de sobreviure als bucles `i` i `j`
`i`	`s1`	índex del bucle mitjà; ha de sobreviure al bucle `j`
sentinella `k`	`s6`	límit del bucle `k`; invariant durant tota l'execució
sentinella `i`	`s2`	límit del bucle `i`; invariant durant tota l'execució
`@V1[0]`	`s3`	adreça base de `V1`; invariant durant tota l'execució
`@M[0][0]`	`s4`	adreça base de `M`; invariant durant tota l'execució
`@V2[0]`	`s5`	adreça base de `V2`; invariant durant tota l'execució
`j`	`t0`	índex del bucle interior
`tmp`	`t1`	acumulador parcial; reinicialitzat a cada iteració d'`i`

Variable	Registre	Justificació
punter `@M[i][k*4+j]`	`t2`	avança 4 bytes a cada iteració de `j`; recalculat a cada `i`
`@V2[k*4]`	`t3`	adreça base del bloc de `V2` per al `k` actual; recalculada a cada `k`
còpia de `t3` per al bucle `j`	`t4`	avança 4 bytes a cada iteració de `j`
sentinella `j`	`t5`	límit del bucle `j`; reinicialitzat a cada `i`
temporals del bucle `j`	`t6`, `a0`	lectures i producte; no cal preservar-los entre iteracions. Com que `t0`–`t6` ja estan tots assignats i no hi ha crides, `a0` queda lliure per usar-lo com a temporal dins el bucle (i es reaprofita per a l'`ecall` de sortida)

Pas 2. Stride d'accés.

Al bucle interior (`j` de 0 a 3), `j` varia més ràpidament. L'element accedit és `M[i][k*4 + j]`: per a `i` i `k` fixos, en incrementar `j` en 1 l'adreça avança `4 bytes` (*stride* = 4, elements contigus a la mateixa fila). Es pot usar un punter `t2` que s'inicialitza a `@M[i][k*4]` i s'incrementa 4 bytes a cada iteració del bucle `j`. Anàlogament, `V2[k*4 + j]` avança 4 bytes per iteració de `j`.

`s6_5_1.s`

```
.eqv NI, 16
.eqv NJ, 16
.eqv BLK, 4                    # mida del bloc de tiling

.data
        .align 2
V1:     .word   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 # int V1[16] inicialitzat a 0
        .align 2
M:      .space  1024                            # int M[16][16]
        .align 2
V2:     .space  64                               # int V2[16]

.text

        la      s3, V1          # s3 = @V1[0]
        la      s4, M           # s4 = @M[0][0]
        la      s5, V2          # s5 = @V2[0]
        li      s6, 4           # s6 = sentinella k = NJ/BLK (invariant)
        li      s2, NI          # s2 = sentinella i (invariant)

        li      s0, 0           # k = 0
buc_k:
        bge     s0, s6, fibuc_k

        # @V2[k*4] = @V2 + k*BLK*4 = @V2 + k*16
        slli    t3, s0, 4       # t3 = k * 16
        add     t3, s5, t3      # t3 = @V2[k*4]
```

```
        li      s1, 0           # i = 0
buc_i:
        bge     s1, s2, fibuc_i

        li      t1, 0           # tmp = 0

        # @M[i][k*4] = @M + i*NJ*4 + k*16
        li      t0, NJ
        mul     t2, s1, t0      # t2 = i * NJ
        slli    t2, t2, 2       # t2 = i * NJ * 4 (bytes)
        add     t2, s4, t2      # t2 = @M[i][0]
        slli    t0, s0, 4       # t0 = k * 16 (desplaçament de columna en bytes)
        add     t2, t2, t0      # t2 = @M[i][k*4]

        mv      t4, t3          # t4 = @V2[k*4] (còpia local per al bucle j)

        li      t0, 0           # j = 0
        li      t5, BLK         # t5 = sentinella j
buc_j:
        bge     t0, t5, fibuc_j

        lw      t6, 0(t2)       # t6 = M[i][k*4 + j]
        lw      a0, 0(t4)       # a0 = V2[k*4 + j]
        mul     t6, t6, a0      # t6 = M[i][k*4+j] * V2[k*4+j]
        add     t1, t1, t6      # tmp += producte

        addi    t2, t2, 4       # punter M avança al següent element de la fila
        addi    t4, t4, 4       # punter V2 avança al següent element del bloc
        addi    t0, t0, 1       # j++
        j       buc_j
fibuc_j:
        # V1[i] += tmp
        slli    t2, s1, 2       # t2 = i * 4
        add     t2, s3, t2      # t2 = @V1[i]
        lw      t6, 0(t2)       # t6 = V1[i]
        add     t6, t6, t1      # t6 = V1[i] + tmp
        sw      t6, 0(t2)       # V1[i] += tmp

        addi    s1, s1, 1       # i++
        j       buc_i
fibuc_i:
        addi    s0, s0, 1       # k++
        j       buc_k
fibuc_k:
        li      a7, 93
        li      a0, 0
        ecall
```

💡 Comprovació pràctica

Verificació de correctesa. Inicialitzeu `M` i `V2` amb els valors de l'enunciat, executeu primer el programa original i després `s6_5_1.s`, i compareu-ne `V1` al segment de dades.
Verificació de rendiment. Executant `s6_5_1.s` (amb *tiling*) i `matvec.s` (original) amb el *Data Cache Simulator* per a les geometries de la Taula 6.1 s'obtenen les taxes d'encerts següents (format *tiling* / original):

Taula 6.2: Taxes d'encerts de referència (solució)

Capacitat	CD	Assoc. grau 2	Assoc. grau 4	Assoc. grau 8	Comp. assoc.
8 línies	0,52 / 0,43	0,84 / 0,81	0,87 / 0,78	0,87 / 0,73	0,87 / 0,73
16 línies	0,70 / 0,65	0,88 / 0,86	0,88 / 0,86	0,88 / 0,86	0,87 / 0,86
32 línies	0,80 / 0,76	0,88 / 0,86	0,88 / 0,86	0,88 / 0,86	0,89 / 0,86
64 línies	0,84 / 0,82	0,89 / 0,86	0,89 / 0,86	0,89 / 0,86	0,89 / 0,86
128 línies	0,89 / 0,86	0,89 / 0,86	0,89 / 0,86	0,89 / 0,86	0,89 / 0,86

Les taxes, però, no comparen bé els dos programes: la versió amb *tiling* fa 640 accessos a dades, i l'original, 528, perquè hi afegeix 112 accessos a `V1` (una lectura i una escriptura per a cada parella `k`, `i`), que són gairebé tots encerts i fan pujar la taxa. El que cal comparar és el **nombre de fallades** (format *tiling* / original):

Taula 6.3: Fallades de referència (solució)

Capacitat	CD	Assoc. grau 2	Assoc. grau 4	Assoc. grau 8	Comp. assoc.
8 línies	306 / 300	105 / 99	82 / 114	84 / 144	84 / 144
16 línies	190 / 184	78 / 72	78 / 72	78 / 72	84 / 72
32 línies	131 / 126	76 / 72	76 / 72	76 / 72	72 / 72
64 línies	100 / 97	73 / 72	72 / 72	72 / 72	72 / 72
128 línies	72 / 72	72 / 72	72 / 72	72 / 72	72 / 72

El *tiling* només redueix les fallades amb 8 línies i grau 4 o més (també completament associativa): és el cas en què la memòria cau no pot mantenir tot `V2`, però sí el bloc actiu de `V2` al costat de la fila de `M`. Amb correspondència directa no les redueix: el bloc de `M[i][…]` i el de `V2` que es fan servir alhora cauen a la mateixa línia i s'expulsen l'un a l'altre. Amb prou capacitat, tots dos programes es queden en les 72 fallades d'arrencada en fred (64 blocs de `M`, 4 de `V2` i 4 de `V1`), i el *tiling* no aporta res.

Calendari

El calendari de les sessions de laboratori canvia cada quadrimestre, i només és a la versió web:
https://rbaig.github.io/migracio.ec.gitlab.upc.edu/04_laboratori/Lcalendari.html

Referència tècnica

RV32I

Registres

Propòsit general

RISC-V 6.1: RV32I Registres — Propòsit general

ISA	ABI	Propòsit	*Saver*
`x0`	`zero`	El valor constant zero	—
`x1`	`ra`	**Adreça de retorn** (***return address***) de subrutina	*Caller*
`x2`	`sp`	**Punter de pila** (***stack pointer***), adreça del cim de la **pila**	*Callee*
`x3`	`gp`	**Punter global** (***global pointer***)	—
`x4`	`tp`	**Punter de fil** (***thread pointer***)	—
`x5`–`x7`	`t0`–`t2`	**Temporals**	*Caller*
`x8`	`s0` / `fp`	**Segur** / **Punter de marc** (***frame pointer***)	*Callee*
`x9`	`s1`	**Segur**	*Callee*
`x10`, `x11`	`a0`, `a1`	**Arguments** de subrutines / Valors de **retorn**	*Caller*
`x12`–`x17`	`a2`–`a7`	**Arguments** de subrutines	*Caller*
`x18`–`x27`	`s2`–`s11`	**Segurs**	*Callee*
`x28`–`x31`	`t3`–`t6`	**Temporals**	*Caller*

Vegeu RISC-V 2.24.

Caller-saved i *callee-saved*

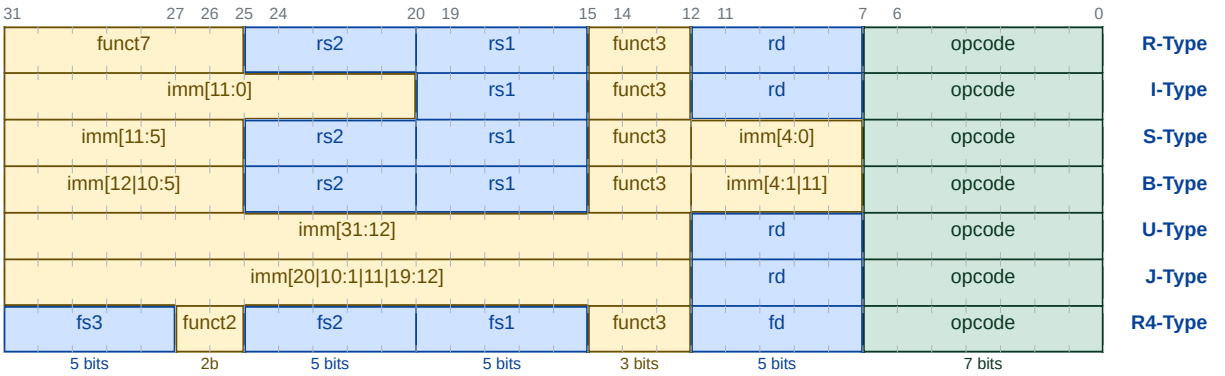
RISC-V 6.2: RV32I ABI — Registres *caller-saved* i *callee-saved*

Categoria	Registres ABI	*Saver*
Temporals i arguments	`t0`–`t6`, `a0`–`a7`	*Caller*
Adreça de retorn	`ra`	*Caller*
Segurs	`s0`–`s11`, `sp`	*Callee*

Vegeu RISC-V 3.22.

Format de les instruccions

RISC-V 6.3: RV32I ISA — Formats d'instrucció



Vegeu RISC-V 2.23, RISC-V 2.26, RISC-V 2.28, RISC-V 3.6, RISC-V 2.27 i RISC-V 3.10.

Instruccions

Aritmètica entera

RISC-V 6.4: RV32I ISA — Instruccions d'aritmètica entera

Mnemònic	Operands	Operació	Nom	Tipus
`add`	`rd, rs1, rs2`	$rd \leftarrow rs1 + rs2$	*add*	R
`addi`	`rd, rs1, imm12`	$rd \leftarrow rs1 + SignExt(imm12)$	*add immediate*	I
`sub`	`rd, rs1, rs2`	$rd \leftarrow rs1 - rs2$	*subtract*	R
`lui`	`rd, imm20`	$rd \leftarrow imm20 << 12$	*load upper immediate*	U
`auipc`	`rd, imm20`	$rd \leftarrow PC + (imm20 << 12)$	*add upper immediate to PC*	U

Vegeu RISC-V 2.12, RISC-V 2.13 i RISC-V 2.14.

Lògica bit a bit

RISC-V 6.5: RV32I ISA — Instruccions lògiques bit a bit

Mnemònic	Operands	Operació	Nom	Tipus
`and`	`rd, rs1, rs2`	$rd \leftarrow rs1 \wedge rs2$	*and*	R
`andi`	`rd, rs1, imm12`	$rd \leftarrow rs1 \wedge SignExt(imm12)$	*and immediate*	I
`or`	`rd, rs1, rs2`	$rd \leftarrow rs1 \vee rs2$	*or*	R
`ori`	`rd, rs1, imm12`	$rd \leftarrow rs1 \vee SignExt(imm12)$	*or immediate*	I

`xor`	`rd, rs1, rs2`	$rd \leftarrow rs1 \oplus rs2$	*exclusive or*	R
`xori`	`rd, rs1, imm12`	$rd \leftarrow rs1 \oplus SignExt(imm12)$	*exclusive or immediate*	I

Vegeu RISC-V 3.3.

Desplaçaments de bits

RISC-V 6.6: RV32I ISA — Instruccions de desplaçament de bits

Mnemònic	Operands	Operació	Nom	Tipus
`sll`	`rd, rs1, rs2`	$rd \leftarrow rs1 << rs2[4:0]$	*shift left logical*	R
`slli`	`rd, rs1, shamt`	$rd \leftarrow rs1 << shamt$	*shift left logical*	I
`srl`	`rd, rs1, rs2`	$rd \leftarrow rs1 >> rs2[4:0]$	*shift right logical*	R
`srli`	`rd, rs1, shamt`	$rd \leftarrow rs1 >> shamt$	*shift right logical*	I
`sra`	`rd, rs1, rs2`	$rd \leftarrow rs1 >>> rs2[4:0]$	*shift right arithmetic*	R
`srai`	`rd, rs1, shamt`	$rd \leftarrow rs1 >>> shamt$	*shift right arithmetic*	I

Vegeu RISC-V 3.1 i RISC-V 3.2. Vegeu EC 3.2 per a la notació `>>>`.

Comparació

RISC-V 6.7: RV32I ISA — Instruccions de comparació

Mnemònic	Operands	Operació	Nom	Tipus
`slt`	`rd, rs1, rs2`	$rd \leftarrow (rs1 < rs2)\,?\,1 : 0 \quad (\mathbb{Z})$	*set less than*	R
`sltu`	`rd, rs1, rs2`	$rd \leftarrow (rs1 < rs2)\,?\,1 : 0 \quad (\mathbb{N})$	*set less than unsigned*	R
`slti`	`rd, rs1, imm12`	$rd \leftarrow (rs1 < SignExt(imm12))\,?\,1 : 0 \quad (\mathbb{Z})$	*set less than immediate*	I
`sltiu`	`rd, rs1, imm12`	$rd \leftarrow (rs1 < SignExt(imm12))\,?\,1 : 0 \quad (\mathbb{N})$	*set less than immediate unsigned*	I

Vegeu RISC-V 3.5.

Lectura i escriptura

RISC-V 6.8: RV32I ISA — Instruccions de lectura i escriptura

Mnemònic	Operands	Operació	Nom	Tipus
`lw`	`rd, offset(rs1)`	$rd \leftarrow M_w[rs1+SignExt(offset)]$	*load word*	I
`lh`	`rd, offset(rs1)`	$rd \leftarrow SignExt(M_h[rs1 + SignExt(offset)])$	*load half*	I
`lhu`	`rd, offset(rs1)`	$rd \leftarrow ZeroExt(M_h[rs1 + SignExt(offset)])$	*load half unsigned*	I
`lb`	`rd, offset(rs1)`	$rd \leftarrow SignExt(M_b[rs1 + SignExt(offset)])$	*load byte*	I
`lbu`	`rd, offset(rs1)`	$rd \leftarrow ZeroExt(M_b[rs1 + SignExt(offset)])$	*load byte unsigned*	I
`sw`	`rs2, offset(rs1)`	$M_w[rs1 + SignExt(offset)] \leftarrow rs2$	*store word*	S
`sh`	`rs2, offset(rs1)`	$M_h[rs1 + SignExt(offset)] \leftarrow rs2[15:0]$	*store half*	S
`sb`	`rs2, offset(rs1)`	$M_b[rs1+SignExt(offset)] \leftarrow rs2[7:0]$	*store byte*	S

Vegeu RISC-V 2.29.

Salts condicionals

RISC-V 6.9: RV32I ISA — Instruccions de salt condicional

Mnemònic	Operands	Operació	Nom	Tipus
`beq`	`rs1, rs2, label`	$rs1 == rs2\,?\,PC \leftarrow PC + SignExt(offset13)$	*branch equal*	B
`bne`	`rs1, rs2, label`	$rs1! = rs2\,?\,PC \leftarrow PC + SignExt(offset13)$	*branch not equal*	B
`blt`	`rs1, rs2, label`	$rs1 < rs2\,?\,PC \leftarrow PC + SignExt(offset13)$ $(\mathbb{Z})$	*branch less than*	B
`bge`	`rs1, rs2, label`	$rs1 >= rs2\,?\,PC \leftarrow PC + SignExt(offset13)$ $(\mathbb{Z})$	*branch greater than or equal*	B
`bltu`	`rs1, rs2, label`	$rs1 < rs2\,?\,PC \leftarrow PC + SignExt(offset13)$ $(\mathbb{N})$	*branch less than unsigned*	B
`bgeu`	`rs1, rs2, label`	$rs1 >= rs2\,?\,PC \leftarrow PC + SignExt(offset13)$ $(\mathbb{N})$	*branch greater than or equal unsigned*	B

Vegeu RISC-V 3.7.

Salts incondicionals

RISC-V 6.10: RV32I ISA — Instruccions de salt incondicional

Mnemònic	Operands	Operació	Nom	Tipus
`jal`	`rd, label`	$rd \leftarrow PC + 4; PC \leftarrow PC + SignExt(offset21)$	*jump and link*	J
`jalr`	`rd, offset(rs1)`	$PC \leftarrow (rs1 + SignExt(offset)) \wedge \overline{1};\ rd \leftarrow PC_{ant} + 4$	*jump and link register*	I

Vegeu RISC-V 3.11 i RISC-V 3.13.

Pseudoinstruccions

Aritmètiques i de còpia

RISC-V 6.11: Pseudoinstruccions — Aritmètiques i de còpia

Pseudoinstrucció	Operació	Expansió
`mv rd, rs`	$rd \leftarrow rs$	`addi rd, rs, 0`
`neg rd, rs`	$rd \leftarrow -rs$	`sub rd, zero, rs`
`not rd, rs`	$rd \leftarrow \overline{rs}$	`xori rd, rs, -1`

Vegeu RISC-V 2.15, RISC-V 3.4 i RISC-V 4.1.

`li`

RISC-V 6.12: Pseudoinstruccions — Càrrega d'immediat a registre: `li`

Pseudoinstrucció	Operació	Condició	Expansió
`li rd, imm12`	$rd \leftarrow imm12$	$-2048 \leq imm12 \leq 2047$	`addi rd, zero, imm12`
`li rd, imm32`	$rd \leftarrow imm32$	$lo_{12}(imm32) = 0$	`lui rd, hi_20(imm32)`
`li rd, imm32`	$rd \leftarrow imm32$	$lo_{12}(imm32) \neq 0$	`lui rd, hi_20(imm32)` `addi rd, rd, lo_12(imm32)`

Vegeu RISC-V 2.16.

`la`

RISC-V 6.13: Pseudoinstruccions — Càrrega d'adreça de memòria: `la`

Pseudoinstrucció	Operació	Expansió

`la rd, label`	$rd \leftarrow @label$	`auipc rd, hi_20(@label - PC)`
		`addi rd, rd, lo_12(@label - PC)`
		(amb l'ajust de hi_{20} si $lo_{12}[11] = 1$)

Vegeu RISC-V 2.17, RISC-V 2.14 i RISC-V 2.14.

Salt condicional

RISC-V 6.14: Pseudoinstruccions — Salt condicional

Pseudoinstrucció	Operació	Expansió
`bgt rs1, rs2, label`	$rs1 > rs2\,?\,PC \leftarrow PC + SignExt(offset13)$ $(\mathbb{Z})$	`blt rs2, rs1, label`
`ble rs1, rs2, label`	$rs1 <= rs2\,?\,PC \leftarrow PC + SignExt(offset13)$ $(\mathbb{Z})$	`bge rs2, rs1, label`
`bgtu rs1, rs2, label`	$rs1 > rs2\,?\,PC \leftarrow PC + SignExt(offset13)$ $(\mathbb{N})$	`bltu rs2, rs1, label`
`bleu rs1, rs2, label`	$rs1 <= rs2\,?\,PC \leftarrow PC + SignExt(offset13)$ $(\mathbb{N})$	`bgeu rs2, rs1, label`

Vegeu RISC-V 3.8.

Salt condicional amb zero

RISC-V 6.15: Pseudoinstruccions — Salt condicional amb zero

Pseudoinstrucció	Operació	Expansió
`beqz rs, label`	$rs == 0\,?\,PC \leftarrow PC + SignExt(offset13)$	`beq rs, zero, label`
`bnez rs, label`	$rs! = 0\,?\,PC \leftarrow PC + SignExt(offset13)$	`bne rs, zero, label`
`bltz rs, label`	$rs < 0\,?\,PC \leftarrow PC + SignExt(offset13)$ $(\mathbb{Z})$	`blt rs, zero, label`
`bgez rs, label`	$rs >= 0\,?\,PC \leftarrow PC + SignExt(offset13)$ $(\mathbb{Z})$	`bge rs, zero, label`
`bgtz rs, label`	$rs > 0\,?\,PC \leftarrow PC + SignExt(offset13)$ $(\mathbb{Z})$	`blt zero, rs, label`
`blez rs, label`	$rs <= 0\,?\,PC \leftarrow PC + SignExt(offset13)$ $(\mathbb{Z})$	`bge zero, rs, label`

Vegeu RISC-V 3.9.

Salt incondicional

RISC-V 6.16: Pseudoinstruccions — Salt incondicional

Pseudoinstrucció	Operació	Expansió
`j label`	$PC \leftarrow PC + SignExt(offset21)$	`jal zero, label`
`jr rs`	$PC \leftarrow rs$	`jalr zero, 0(rs)`
`ret`	$PC \leftarrow ra$	`jalr zero, 0(ra)`

Vegeu RISC-V 3.12 i RISC-V 3.14.

RV32M

Instruccions

RISC-V 6.17: RV32IM ISA — Instruccions de multiplicació i divisió entera

Mnemònic	Operands	Operació	Nom	Tipus
`mul`	`rd, rs1, rs2`	$rd \leftarrow (rs1 \times rs2)[31{:}0]$	*Multiply*	R
`mulh`	`rd, rs1, rs2`	$rd \leftarrow (rs1 \times rs2)[63{:}32]$ (enters)	*Multiply High*	R
`mulhu`	`rd, rs1, rs2`	$rd \leftarrow (rs1 \times rs2)[63{:}32]$ (naturals)	*Multiply High Unsigned*	R
`mulhsu`	`rd, rs1, rs2`	$rd \leftarrow (rs1 \times rs2)[63{:}32]$ ($rs1$ enter, $rs2$ natural)	*Multiply High Signed-Unsigned*	R
`div`	`rd, rs1, rs2`	$rd \leftarrow rs1 \div rs2$ (enters)	*Divide*	R
`divu`	`rd, rs1, rs2`	$rd \leftarrow rs1 \div rs2$ (naturals)	*Divide Unsigned*	R
`rem`	`rd, rs1, rs2`	$rd \leftarrow rs1 \bmod rs2$ (enters)	*Remainder*	R
`remu`	`rd, rs1, rs2`	$rd \leftarrow rs1 \bmod rs2$ (naturals)	*Remainder Unsigned*	R

Vegeu RISC-V 4.2 i RISC-V 4.3.

RV32F

Registres

Coma flotant

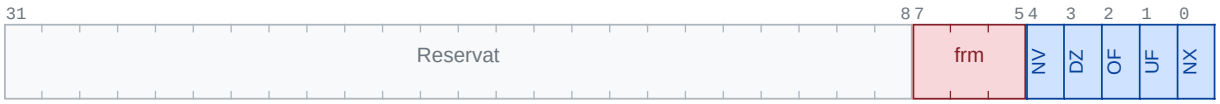
RISC-V 6.18: RV32F ABI — Registres de coma flotant

ISA	ABI	Propòsit	*Saver*
`f0`–`f7`	`ft0`–`ft7`	**Temporals**	*Caller*
`f8`–`f9`	`fs0`–`fs1`	**Segurs**	*Callee*
`f10`–`f11`	`fa0`–`fa1`	**Arguments** de subrutines / Valors de **retorn**	*Caller*
`f12`–`f17`	`fa2`–`fa7`	**Arguments** de subrutines	*Caller*
`f18`–`f27`	`fs2`–`fs11`	**Segurs**	*Callee*
`f28`–`f31`	`ft8`–`ft11`	**Temporals**	*Caller*

Vegeu RISC-V 5.2.

Control (`fcsr`)

RISC-V 6.19: RV32F ISA — Registre de control `fcsr`



Vegeu RISC-V 5.3.

Instruccions

Lectura i escriptura

RISC-V 6.20: RV32F ISA — Lectura i escriptura

Mnemònic	Operands	Operació	Nom	Tipus
`flw`	`rd, offset(rs1)`	$rd \leftarrow M_w[rs1 + SignExt(offset)]$	*Float Load Word*	I
`fsw`	`rs2, offset(rs1)`	$M_w[rs1 + SignExt(offset)] \leftarrow rs2$	*Float Store Word*	S

Vegeu RISC-V 5.4.

Aritmètica

RISC-V 6.21: RV32F ISA — Instruccions aritmètiques

Mnemònic	Operands	Operació	Nom	Tipus
`fadd.s`	`rd, rs1, rs2`	$rd \leftarrow rs1 + rs2$	*Float Add*	R
`fsub.s`	`rd, rs1, rs2`	$rd \leftarrow rs1 - rs2$	*Float Subtract*	R
`fmul.s`	`rd, rs1, rs2`	$rd \leftarrow rs1 \times rs2$	*Float Multiply*	R
`fdiv.s`	`rd, rs1, rs2`	$rd \leftarrow rs1/rs2$	*Float Divide*	R
`fsqrt.s`	`rd, rs1`	$rd \leftarrow \sqrt{rs1}$	*Float Square Root*	R
`fmin.s`	`rd, rs1, rs2`	$rd \leftarrow \min(rs1, rs2)$	*Float Minimum*	R
`fmax.s`	`rd, rs1, rs2`	$rd \leftarrow \max(rs1, rs2)$	*Float Maximum*	R

Vegeu RISC-V 5.5.

Comparacions

RISC-V 6.22: RV32F ISA — Comparacions

Mnemònic	Operands	Operació	Nom	Tipus
`feq.s`	`rd, rs1, rs2`	$rd \leftarrow (rs1 = rs2)\ ?\ 1 : 0$	*Float Equal*	R
`flt.s`	`rd, rs1, rs2`	$rd \leftarrow (rs1 < rs2)\ ?\ 1 : 0$	*Float Less Than*	R
`fle.s`	`rd, rs1, rs2`	$rd \leftarrow (rs1 \leq rs2)\ ?\ 1 : 0$	*Float Less or Equal*	R

Vegeu RISC-V 5.7.

Conversions i moviments

RISC-V 6.23: RV32F ISA — Conversions i moviments

Conversions (amb conversió numèrica):

Mnemònic	Operands	Operació	Nom
`fcvt.s.w`	`rd, rs1`	$rd \leftarrow (float)\ rs1$	*Enter → flotant*
`fcvt.s.wu`	`rd, rs1`	$rd \leftarrow (float)\ rs1$	*Natural → flotant*
`fcvt.w.s`	`rd, rs1`	$rd \leftarrow (int)\ rs1$	*Flotant → enter (arrodoniment segons* `rm`*;* `rtz` *per a la conversió de tipus de C)*
`fcvt.wu.s`	`rd, rs1`	$rd \leftarrow (unsigned)\ rs1$	*Flotant → natural (arrodoniment segons* `rm`*;* `rtz` *per a la conversió de tipus de C)*

Moviments (còpia de bits sense conversió numèrica):

Mnemònic	Operands	Operació	Nom
`fmv.w.x`	`rd, rs1`	$rd \leftarrow rs1$	*Enter → flotant (bits)*
`fmv.x.w`	`rd, rs1`	$rd \leftarrow rs1$	*Flotant → enter (bits)*

Vegeu RISC-V 5.8.

Pseudoinstruccions

RISC-V 6.24: Pseudoinstruccions — Còpia entre registres de coma flotant: `fmv.s`

Pseudoinstrucció	Operació	Expansió
`fmv.s rd, rs`	$rd \leftarrow rs$	`fsgnj.s rd, rs, rs`

Vegeu RISC-V 5.6.

Zicsr

Instruccions

RISC-V 6.25: Zicsr ISA — Instruccions d'accés als CSR

Mnemònic	Operands	Operació	Nom	Tipus
`csrrw`	`rd, csr, rs1`	$rd \leftarrow csr;\ csr \leftarrow rs1$	*CSR read/write*	I
`csrrs`	`rd, csr, rs1`	$rd \leftarrow csr;\ csr \leftarrow csr \vee rs1$	*CSR read and set bits*	I
`csrrc`	`rd, csr, rs1`	$rd \leftarrow csr;\ csr \leftarrow csr \wedge \overline{rs1}$	*CSR read and clear bits*	I

Vegeu RISC-V 9.2.

Pseudoinstruccions

RISC-V 6.26: Pseudoinstruccions — Accés als CSR: `csrr` i `csrw`

Pseudoinstrucció	Operació	Expansió
`csrr rd, csr`	$rd \leftarrow csr$	`csrrs rd, csr, zero`
`csrw csr, rs`	$csr \leftarrow rs$	`csrrw zero, csr, rs`

RARS 1.6 espera l'ordre invers dels operands de `csrw`, `csrw rs, csr` (`csrw t1, fcsr`), i no té els CSR de mode M, com `mepc`.
Vegeu RISC-V 9.3.

Excepcions i interrupcions

Instruccions

RISC-V 6.27: RV32I ISA — `ecall` i `ebreak`

Mnemònic	Operands	Operació	Nom
`ecall`	—	Provoca una excepció (crida al sistema)	*Environment call*
`ebreak`	—	Provoca una excepció (punt de parada)	*Environment break*

Vegeu RISC-V 9.11.

RISC-V 6.28: RV32I ISA — `mret`

Mnemònic	Operands	Operació	Nom
`mret`	—	$MIE \leftarrow MPIE$; $MPIE \leftarrow 1$; $mode \leftarrow MPP$; $MPP \leftarrow U$; $PC \leftarrow mepc$	*Machine return*

Vegeu RISC-V 9.10.

RISC-V 6.29: Supervisor ISA — `sfence.vma`

Mnemònic	Operands	Operació	Nom
`sfence.vma`	`rs1, rs2`	Invalida les entrades del TLB per a l'adreça `rs1` i l'ASID `rs2` (`x0` = totes)	*Supervisor fence*

Vegeu RISC-V 9.16.

Registres CSR

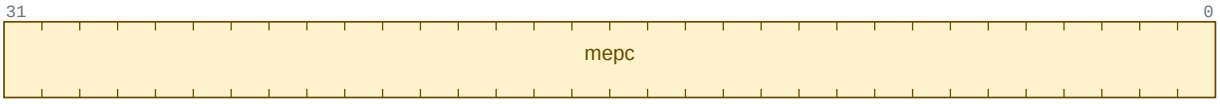
RISC-V 6.30: RV32I ISA — `mcause` — Valors més rellevants

Bit 31	Codi	Causa
0	0	Instrucció mal alineada
0	1	Error d'accés en cerca d'instrucció

0	2	Instrucció il · legal
0	3	Punt de parada (`ebreak`)
0	4	Lectura de dada mal alineada
0	5	Error d'accés en lectura de dada
0	6	Escriptura de dada mal alineada
0	7	Error d'accés en escriptura de dada
0	8	Crida al sistema des del mode U (`ecall`)
0	9	Crida al sistema des del mode S (`ecall`)
0	11	Crida al sistema des del mode M (`ecall`)
0	12	Fallada de pàgina en cerca d'instrucció
0	13	Fallada de pàgina en lectura
0	15	Fallada de pàgina en escriptura o en una *atomic memory operation* (AMO)
1	3	Interrupció de programari en mode M
1	7	Interrupció de temporitzador en mode M
1	11	Interrupció externa (E/S) en mode M

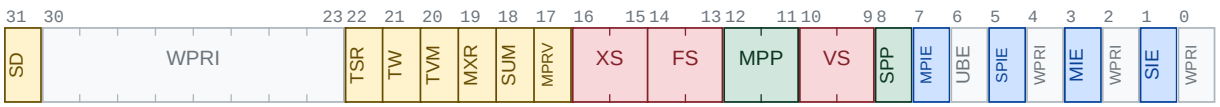
Vegeu RISC-V 9.4.

RISC-V 6.31: RV32I ISA — `mepc`



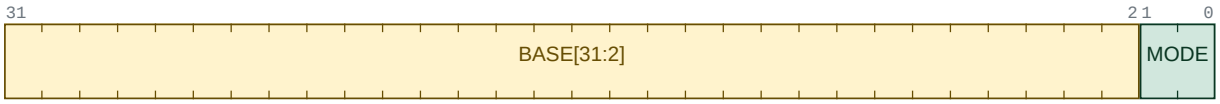
Vegeu RISC-V 9.5.

RISC-V 6.32: RV32I ISA — `mstatus`



Vegeu RISC-V 9.6.

RISC-V 6.33: RV32I ISA — `mtvec`



- **`BASE`** (bits 31:2): adreça base de la rutina o taula de vectors, alineada a 4 bytes.
- **`MODE`** (bits 1:0): mode de funcionament.

RISC-V defineix dos modes:

`MODE`	Nom	Comportament
0	Directe	Totes les excepcions i interrupcions salten a `BASE`
1	Vectoritzat	Les interrupcions salten a `BASE + 4 × causa`; les excepcions sempre a `BASE`

Vegeu RISC-V 9.7.

RISC-V 6.34: RV32I ISA — `mip` i `mie`



Bit	Nom `mip`	Nom `mie`	Tipus d'interrupció
11	`MEIP`	`MEIE`	Externa (E/S) en mode M
9	`SEIP`	`SEIE`	Externa (E/S) en mode S
7	`MTIP`	`MTIE`	Temporitzador en mode M
5	`STIP`	`STIE`	Temporitzador en mode S
3	`MSIP`	`MSIE`	Programari en mode M
1	`SSIP`	`SSIE`	Programari en mode S

Vegeu RISC-V 9.8.

RISC-V 6.35: RV32I ISA — Registres CSR de mode M i mode S

Registre mode M	Registre mode S	Funció
`mcause`	`scause`	Codi de causa de l'excepció o interrupció
`mepc`	`sepc`	Adreça de retorn
`mstatus`	`sstatus`	Estat global (subconjunt dels bits de `mstatus`)
`mtvec`	`stvec`	Adreça de la rutina de servei

`mip`	`sip`	Peticions d'interrupció pendents
`mie`	`sie`	Màscares d'interrupció

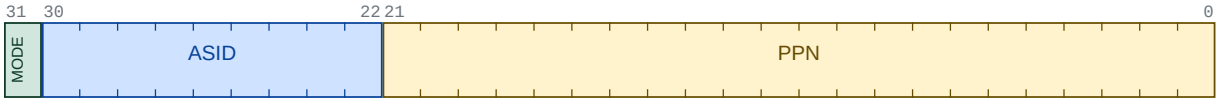
Vegeu RISC-V 9.9.

RISC-V 6.36: RV32I ISA — Registres CSR de memòria virtual

Registre	Nom complet	Funció
`satp`	*Supervisor Address Translation and Protection*	Mode de traducció i adreça física de la taula de pàgines de primer nivell
`stval`	*Supervisor Trap Value*	Adreça virtual que ha causat la fallada de pàgina

Vegeu RISC-V 9.14.

RISC-V 6.37: RV32I ISA — `satp`



Vegeu RISC-V 9.15.

RARS

Mapa de memòria

RISC-V 6.38: RARS — Segments de memòria

Segment	Tipus	Adreça inicial	Contingut	Creix cap a
`.text`	Estàtic	`0x00400000`	Codi màquina (instruccions)	—
`.data`	Estàtic	`0x10010000`	Variables globals	—
heap	Dinàmic	`0x10040000`	Variables dinàmiques (`malloc()`/`free()`)	adreces altes
pila (*stack*)	Dinàmic	`0x7FFFFFFC` (`sp`: `0x7FFFEFFC`, RISC-V 3.21)	Variables locals, registres desats	adreces baixes

nucli (*kernel*)	Estàtic	`0x80000000`	Codi i dades del sistema operatiu	—

Vegeu RISC-V 3.15.

Directives

RISC-V 6.39: Directives — RARS

Directiva	Operands	Descripció
`.align`	`k`	Alinea l'element següent a la frontera de 2^k bytes
`.ascii`	`"cadena"`	Emmagatzema la cadena al segment de dades sense terminador nul
`.asciz`	`"cadena"`	Emmagatzema la cadena al segment de dades amb terminador nul
`.byte`	`v1, v2, ...`	Emmagatzema els valors com a bytes de 8 bits
`.data`	—	Els elements següents s'emmagatzemen al segment de dades
`.double`	`v1, v2, ...`	Emmagatzema els valors com a nombres en coma flotant de precisió doble
`.dword`	`v1, v2, ...`	Emmagatzema els valors com a paraules dobles de 64 bits
`.end_macro`	—	Acaba la definició d'una macro (vegeu Secció 2.3.5.3)
`.eqv`	`símbol, literal`	Substitueix el símbol pel literal. ⚠ **No s'hi admet cap expressió aritmètica ni cap valor negatiu**: `.eqv T, 5*6*4` dona `Invalid language element` i `.eqv NEG, -4`, `Malformed .eqv directive`. Vegeu EC 2.3.
`.float`	`v1, v2, ...`	Emmagatzema els valors com a nombres en coma flotant de precisió simple
`.globl`	`etiqueta`	Declara l'etiqueta com a global (també s'accepta `.global`)
`.half`	`v1, v2, ...`	Emmagatzema els valors com a mitges paraules de 16 bits
`.macro`	`nom [(%p1, …)]`	Inicia la definició d'una macro, amb els paràmetres opcionals (vegeu Secció 2.3.5.3)
`.section`	`nom`	⚠ **No la useu**; useu `.data` i `.text`. Amb un nom reconegut (`.rodata`, `.sdata`) **descarta sense cap avís el codi que la segueix**, que no queda assemblat en cap segment. Amb `.data` o `.text` com a operand no assembla. Amb un nom desconegut, avisa i no fa res. Vegeu Secció 2.3.1.
`.space`	`n`	Reserva `n` bytes al segment de dades
`.string`	`"cadena"`	Àlies de `.asciz`
`.text`	—	Els elements següents s'emmagatzemen al segment de text
`.word`	`v1, v2, ...`	Emmagatzema els valors com a paraules de 32 bits

Crides al sistema

RISC-V 6.40: RARS — Registres de crida al sistema

- **`a7`**: codi de servei.
- **`a0`**: argument principal i valor de retorn.

Vegeu RISC-V 9.12.

RISC-V 6.41: RARS — Codis de servei més rellevants

Codi	Nom	Arguments	Retorn
1	`print_int`	`a0` = enter	—
2	`print_float`	`fa0` = flotant	—
4	`print_string`	`a0` = adreça de la cadena	—
5	`read_int`	—	`a0` = enter
6	`read_float`	—	`fa0` = flotant
8	`read_string`	`a0` = adreça de la memòria intermèdia, `a1` = longitud màxima	—
10	`exit`	—	—
11	`print_char`	`a0` = caràcter (ASCII)	—
12	`read_char`	—	`a0` = caràcter (ASCII)
34	`print_hex`	`a0` = enter	—
35	`print_bin`	`a0` = enter	—
36	`print_unsigned`	`a0` = natural	—
93	`exit2`	`a0` = codi de sortida	—

Vegeu RISC-V 9.13.

Sigles, símbols i notació

Sigles

Sigles i acrònims del llibre, per ordre alfabètic. El significat és la forma catalana, amb l'expansió anglesa en cursiva entre parèntesis quan la sigla ve de l'anglès, o només l'expansió anglesa quan no hi ha forma catalana (els noms propis, per exemple). El text expandeix cada sigla només la primera vegada que surt al llibre, amb el mateix format: «unitat d'estat sòlid (*solid-state drive*, SSD)».

Taula 6.4: Sigles i acrònims.

Sigla	Significat	Observacions
ABI	*Application binary interface*	
AC	Avaluació continuada	Del laboratori
ALU	Unitat aritmeticològica (*arithmetic logic unit*)	
AMAT	*Average memory access time*	Equivalent anglès de TAM; només apareix en prosa, no en fórmules
AMO	*Atomic memory operation*	
API	*Application programming interface*	
ARM	*Advanced RISC Machines*	
ASCII	*American Standard Code for Information Interchange*	
BA	Bloc d'activació (*stack frame*)	
Ca1	Complement a 1	
Ca2	Complement a 2	
CF	Coma flotant	
CISC	*Complex instruction set computer*	
CMOS	*Complementary metal–oxide–semiconductor*	
CPI	Cicles per instrucció	Apareix també com a variable dins de fórmules; vegeu CPI a Símbols
CPU	Unitat central de processament (*central processing unit*)	
CSR	Registre de control i estat (*control and status register*)	
CU	Unitat de control (*control unit*)	
DRAM	RAM dinàmica (*dynamic random access memory*)	
EC	Estructura de computadors	Assignatura del GEI
EF	Examen final	
EL	Examen de laboratori	
ELF	*Executable and Linkable Format*	
EP	Examen parcial	
E/S	Entrada/sortida	
FIB	Facultat d'Informàtica de Barcelona	
FIFO	*First in, first out*	

Sigla	Significat	Observacions
GCC	*GNU Compiler Collection*	
GEI	Grau en Enginyeria Informàtica	
GPR	Registre de propòsit general (*general-purpose register*)	
HTML	*HyperText Markup Language*	
IC	Introducció als computadors	Assignatura del GEI
IEC	*International Electrotechnical Commission*	
IEEE	*Institute of Electrical and Electronics Engineers*	
ISA	Arquitectura del repertori d'instruccions (*instruction set architecture*)	
ISO	*International Organization for Standardization*	
JRE	*Java Runtime Environment*	
L1	Memòria cau de primer nivell	
L2	Memòria cau de segon nivell	
L3	Memòria cau de tercer nivell	
L/E	Lectura/escriptura	
LED	Díode emissor de llum (*light-emitting diode*)	
LIFO	Última en entrar, primera en sortir (*last in, first out*)	
LRU	*Least recently used*	
LSB	Byte de menys pes (*least significant byte*)	
LSb	Bit de menys pes (*least significant bit*)	
MARS	*MIPS Assembler and Runtime Simulator*	
MC	Memòria cau	Apareix també dins de fórmules (T7); vegeu MC_i, MC_d a Símbols
MIPS	*Microprocessor without Interlocked Pipeline Stages*	
MMU	Unitat de gestió de memòria (*memory management unit*)	
MP	Memòria principal	
MRU	*Most recently used*	
MSB	Byte de més pes (*most significant byte*)	
MSb	Bit de més pes (*most significant bit*)	
NA	No aplicable	
NaN	*Not a number*	
NF	Nota final	A T4, `NF` és el nombre de files d'una matriu (vegeu Símbols)
NL	Nota de laboratori	
NT	Nota de teoria i problemes	
OPT	Òptim (*optimal*)	L'algorisme de reemplaçament òptim (T7)
PC	Comptador de programa (*program counter*)	
PIPT	Memòria cau indexada i etiquetada físicament (*physically indexed physically tagged*)	
POSIX	*Portable Operating System Interface*	
PPN	Número de pàgina física (*physical page number*)	
PRO1	Programació I	Assignatura del GEI
PTE	Entrada de la taula de pàgines (*page table entry*)	
RAM	*Random access memory*	
RARS	*RISC-V Assembler and Runtime Simulator*	

Sigla	Significat	Observacions
RDN	Arrodoniment cap a $-\infty$ (*round down*)	
RISC	*Reduced instruction set computer*	
RMM	Al més pròxim amb empat cap a la magnitud major (*round to nearest, ties to max magnitude*)	
RNE	Arrodoniment al més pròxim (*round to nearest, ties to even*)	
ROM	*Read-only memory*	
RSE	Rutina de servei d'excepcions	
RTZ	Arrodoniment cap a zero (*round towards zero*)	
RUP	Arrodoniment cap a $+\infty$ (*round up*)	
SI	Sistema Internacional d'Unitats	
S&M	Signe i magnitud	
SO	Sistema operatiu	
SRAM	RAM estàtica (*static random access memory*)	
SSD	Unitat d'estat sòlid (*solid-state drive*)	
STL	Biblioteca estàndard de plantilles (*Standard Template Library*)	
TAM	Temps d'accés mitjà	Apareix també com a variable dins de fórmules (T7); vegeu TAM a Símbols
TLB	*Translation lookaside buffer*	
UI	Interfície d'usuari (*user interface*)	
ULP	*Unit in the last place*	Apareix també com a variable dins de fórmules (T5); vegeu ULP a Símbols
VIPT	Memòria cau indexada virtualment i etiquetada físicament (*virtually indexed physically tagged*)	Condició per evitar l'aliàsing (T8): $N_C \cdot B \leq T$
VIVT	Memòria cau indexada i etiquetada virtualment (*virtually indexed virtually tagged*)	
VLSI	Integració a escala molt gran (*very large scale integration*)	
VPN	Número de pàgina virtual (*virtual page number*)	

Símbols

Recull dels símbols i variables usats a les fórmules, amb el tema on s'introdueixen. Quan un símbol té significats diferents segons el tema, s'indica a la columna «Observacions». Les sigles que fan de variable van en lletra dreta (CPI, TAM), i les constants d'un programa, en lletra de codi (`NF`).

Taula 6.5: Símbols i variables de les fórmules.

Símbol	Significat	Tema	Observacions
a, b	Operands enters de la suma i la resta	T1, T4, T5	$s = a + b$, $d = a - b$; vegeu A, B per a les cadenes de bits que els representen
a_i	Bit i-èsim d'un dels sumands	T4	Sumador de n bits: $s_i = a_i \oplus b_i \oplus c_i$. És el mateix bit que A_i: el sumador fa servir la minúscula habitual dels circuits

Símbol	Significat	Tema	Observacions
A	Adreça (de la dada a la memòria principal)	T7	
A, A_i, A_{n-1}	Cadena de bits de l'operand a / bit i-èsim / bit de més pes	T4	$S_n = A_{n-1} = B_{n-1}$ (condició de mateix signe)
b	Nombre de bits del desplaçament (*offset*) dins un bloc	T7	Vegeu t per al desplaçament dins una pàgina (T8)
b_i	Bit i-èsim d'un dels sumands	T4	Anàleg a a_i
b_i	Bit i-èsim d'una adreça lògica	T8	Descomposició $b_{n-1} \cdots b_t \mid b_{t-1} \cdots b_0$ (VPN i desplaçament); no confondre amb el b sol de T7
b_{-i}	Bit de la fracció de pes 2^{-i}	T5	$v = \pm 1{,}b_{-1}b_{-2} \ldots b_{-(p-1)} \times 2^e$; l'índex negatiu el distingeix de b_i (T4, T8)
B	Mida del bloc (en mots o bytes)	T7	
B, B_i, B_{n-1}	Cadena de bits de l'operand b / bit i-èsim / bit de més pes	T4	Anàleg a A, A_i, A_{n-1}
c_i	Bit de ròssec d'entrada (*carry-in*) del sumador i	T4	c_{i+1} n'és el ròssec de sortida (*carry-out*); c_{n-1}/c_n són el darrer ròssec d'entrada/de sortida de la cadena
C	Capacitància agregada del circuit (farads)	T6	**No confondre amb la capacitat (bytes) de la memòria cau**: concepte físic diferent
C_i	Capacitància equivalent connectada a la sortida de la porta i	T6	$q_i = C_i \cdot V_{CC}$; $\sum C_i = \alpha \cdot C$. **No confondre amb CPI$_i$**
CPI	Cicles per instrucció	T6	Vegeu «CPI» a Sigles per a la definició completa
CPI$_i$	CPI del tipus d'instrucció i	T6	Vegeu n_i per al nombre d'instruccions d'aquest tipus
CPI$_{ideal}$	CPI sense penalització per fallades de memòria cau	T7	
CPI$_{penal}$	Terme de penalització del CPI degut a fallades de memòria cau	T7	CPI$_{penal} = t_p \cdot m \cdot n_r$
d	Diferència (resta de dos operands)	T4	$d = a - b$
D	Bit de modificació (*dirty bit*)	T7, T8	Introduït a T7 (línia d'MC); reutilitzat a T8 (PTE i TLB) amb el mateix significat
e	Exponent (valor enter)	T5	$v = \pm 1{,}F \times 2^e$, i també a la notació científica decimal ($v = m \times 10^e$); $e = E_u - 127$
E, E_i	Camp d'exponent (8 bits en precisió simple) / bit i-èsim	T5	El seu valor natural és E_u; a les codificacions especials ($E = 0$, $E = 255$) el camp i el seu valor s'usen indistintament
E	Energia dissipada	T6	$E = P \cdot t$
E_{min}	Exponent mínim normalitzat	T5	$2^{E_{min}} = 2^{-126}$ en precisió simple
E_u	Valor natural del camp d'exponent (exponent esbiaixat, en excés a 127)	T5	$e = E_u - 127 \Leftrightarrow E_u = e + 127$
f	Freqüència de rellotge (abreviatura de f_{clock})	T6	Als exemples i problemes, un cop definida f_{clock}

Símbol	Significat	Tema	Observacions
f_A, f_B	Freqüència de rellotge dels computadors A i B (exemples comparatius)	T6	
f_{clock}	Freqüència de rellotge	T6	$f_{clock} = 1/t_{clock}$
F, F_i	Camp de fracció (23 bits en precisió simple) / bit i-èsim	T5	$v = (-1)^S \times 1{,}F \times 2^{E_u - 127}$
G	Bit de guarda (*guard bit*)	T5	Regla RNE amb G, R, S
h	Taxa d'encerts (*hit rate*)	T7	$h = \frac{\text{núm. encerts}}{\text{núm. referències}} = 1 - m$
i	Índex (genèric: d'instrucció, d'iteració, de bit)	T1, T4, T6, T7	Ús transversal com a índex de sumatori o de bit
I	Corrent elèctric	T6	$I = \sum q_i / t_{clock}$
I_{fuita}	Corrent de fuita	T6	$P_s = V_{CC} \cdot I_{fuita}$
K	Excés (representació en excés)	T1	L'elecció habitual és $K = 2^{n-1} - 1$; a T5, l'exponent IEEE 754 fa servir aquest mateix valor (127 en precisió simple), que s'anomena **biaix** i no porta símbol
m	Nombre de bits de la representació ampliada (T1); mantissa (T5); nombre de tipus diferents d'instruccions (T6); taxa de fallades (*miss rate*) (T7)	T1, T5, T6, T7	**Significats diferents segons el tema**: a T1, l'extensió de n a $m > n$ bits; a T5, la mantissa; a T6, el nombre de tipus d'instrucció al sumatori del CPI ponderat; a T7, la taxa de fallades
m_d, m_i	Taxa de fallades de dades / d'instruccions	T7	
m_{L1}, m_{L2}	Taxa de fallades de L1 / L2	T7	Notació multinivell
MC_i, MC_d	Memòria cau d'instruccions / de dades	T7	Vegeu «MC» a Sigles
n	Nombre de bits (T1, transversal); Número de bloc (instància concreta, T7); Nombre de bits d'una adreça lògica (T8)	T1, T5, T7, T8	Ús molt transversal; el significat concret depèn del context immediat
n_A, n_B	Nombre de cicles dels computadors A i B (exemple comparatiu)	T6	$n_A = t_A \cdot f_A$
n_c	Nombre de bits d'índex de conjunt, $N_C = 2^{n_c}$	T7	No és el nombre de cicles, que és n_{cicles} (T6)
n_i	Nombre d'instruccions del tipus i	T6	Sumatori del CPI ponderat: $\text{CPI} = \frac{1}{n_{ins}} \sum n_i \cdot \text{CPI}_i$
n_l	Nombre de bits d'índex de línia, $N_L = 2^{n_l}$ (correspondència directa)	T7	
n_{cicles}	Nombre de cicles de rellotge d'una execució	T6	$t_{exe} = n_{cicles} \cdot t_{clock}$
n_{dat}, n_{ins}	Nombre d'accessos a dades / a instruccions	T7	A T6, n_{ins} és el nombre total d'instruccions executades d'un programa; a T7 coincideix amb el nombre d'accessos a l'MC d'instruccions
$n_{fallades}$	Nombre total de fallades de memòria cau	T7	$\text{CPI}_{penal} = t_p \cdot n_{fallades}/n_{ins}$

Símbol	Significat	Tema	Observacions
n_r	Nombre de referències a memòria per instrucció	T7	$n_r = n_{refs}/n_{ins} \geq 1$
n_{refs}	Nombre total de referències a memòria	T7	$n_{refs} = n_{ins} + n_{dat}$
N	Nombre d'instruccions executades (abreviatura de n_{ins})	T6	Als exemples i problemes, un cop definit n_{ins}; **a T7, N és el grau d'associativitat**
N	Grau d'associativitat (nombre de vies)	T7	
N_C	Nombre de conjunts	T7, T8	A T8, condició VIPT: $N_C \cdot B \leq T$
N_L	Nombre de línies de la memòria cau	T7	
`NC`, `NF`	Nombre de columnes / de files d'una matriu	T4	Constants del programa (`#define` en C, `.eqv` en assemblador): per això en lletra de codi. `@mat[i][j]` = `@mat[0][0]` + $(i \cdot$ `NC` $+ j) \cdot T$. A Sigles, NF és la nota final
p	Precisió (nombre de bits de la mantissa) (T5); Nombre de bits del PPN (*physical page number*) (T8)	T5, T8	Significats diferents segons el tema. A T5, p inclou el bit ocult: la fracció en té $p-1$
p_e, p_m	Fracció de les fallades que són d'escriptura / que reemplacen un bloc modificat ($D=1$)	T7	$t_p = (1-p_e)(t_{bloc}+t_h)$ (immediata sense assignació); $t_p = (1+p_m)\,t_{bloc}+t_h$ (retardada amb assignació)
P	Potència	T6	$P = P_d + P_s$
P, MD, MR; R, D, Q	Registres dels circuits seqüencials de multiplicació (producte, multiplicand, multiplicador) i de divisió (dividend i residu, divisor, quocient)	T4	Noms de maquinari, no variables: al text, en rodona; a les fórmules, només amb el seu valor o un bit ($P = 10000010_2$, R_{2n-1})
P_d, P_s	Potència dinàmica / estàtica	T6, T7	P_d es reutilitza a T7 en l'anàlisi energètica de la memòria cau
P_x	Fracció del temps total ocupada pel component optimitzat (llei d'Amdahl)	T6, T7	**No és una potència** malgrat la lletra P: és una fracció de temps. Amb diversos components, $\sum_x P_x/s_x$. Reutilitzat a T7 ($P_x = t_{proc}/(t_{proc}+t_{acc})$)
q	Quocient	T4	$q = x \div y$; $x = y \cdot q + r$
q_i	Bit i-èsim del quocient	T4	Pseudocodi de divisió seqüencial
q_i	Càrrega elèctrica transferida en commutar la porta i	T6	$q_i = C_i \cdot V_{CC}$; **no confondre amb q_i de T4** (bit de quocient)
r	Residu	T4	$r = x \bmod y$; $x = y \cdot q + r$, amb $0 \leq r < y$
R	Bit d'arrodoniment (*round bit*)	T5	Regla RNE amb G, R, S
R	Resistència equivalent del circuit RC	T6	$V(t) = V_{CC}(1-e^{-t/RC})$; **no confondre** amb el bit d'arrodoniment R (T5) ni amb el registre R del divisor seqüencial (T4)
s	Valor exacte de la suma (abans de truncar)	T4	$s = a + b$; S' n'és la cadena de bits truncada

Símbol	Significat	Tema	Observacions
s	Guany de rendiment (*speedup*)	T6	$s = t_{\text{referència}}/t_{nou}$: relació entre el rendiment del sistema nou i el de referència
s_i	Bit de suma del sumador i	T4	$s_i = a_i \oplus b_i \oplus c_i$; és el mateix bit que S_i de la cadena de la suma
s_{max}, s_x	Guany màxim / guany del component x (llei d'Amdahl)	T6, T7	$s_{max} = 1/(1 - P_x)$; reutilitzat a T7 en l'anàlisi de rendiment amb memòria cau
stride	Distància en bytes entre les adreces de dos elements consecutius d'un recorregut	T4	stride $= @X_{i+1} - @X_i$; per a una columna d'una matriu, $\texttt{NC} \cdot T$
S	Bit de signe (representació IEEE 754)	T5	$v = (-1)^S \times 1{,}F \times 2^{E_u - 127}$
S	Bit *sticky*	T5	**Significat diferent dins del mateix tema (T5)**: a la representació és el bit de signe; als bits de guarda (G, R, S) és el bit *sticky*
S, S_i, S', S_n	Cadena de bits de la suma (completa / bit i-èsim / truncada / bit addicional)	T4	$S' = S_{n-1} \ldots S_0$ és la cadena truncada; correcta si $S_{n-1} = S_n$
t	Temps genèric, base dels símbols compostos t_{xxx} (T6, T7); Nombre de bits del desplaçament dins una pàgina (T8)	T6, T7, T8	**Significats diferents segons el tema**: a T6/T7 és la base temporal (t_h, t_p…); a T8 és un nombre de bits ($T = 2^t$), anàleg a b per als blocs a T7
t_0, t_1	Temps original / temps després de l'optimització (llei d'Amdahl)	T6	$s = t_0/t_1$
t_A, t_B	Temps d'execució dels computadors A i B (exemple comparatiu)	T6	
t_{acc}	Temps d'accés	T7	Vegeu TAM a Sigles per a la mitjana ponderada
t_{bloc}	Temps de transferència d'un bloc	T7	
t_{MP}	Temps d'accés a la memòria principal	T7	
t_c	Període de rellotge (abreviatura de t_{clock})	T6	Als exemples i problemes, un cop definit t_{clock}
t_{clock}	Període de rellotge	T6	
t_{exe}	Temps d'execució	T6	Reutilitzat a T7 en el context del càlcul de rendiment amb memòria cau
t_h	Temps d'encert (*hit time*)	T7	
$t_{h_{L1}}$, $t_{h_{L2}}$	Temps d'encert de L1 / L2	T7	Notació multinivell, anàloga a m_{L1}/m_{L2}
t_{nou}, $t_{\text{referència}}$	Temps d'execució del sistema nou / de referència	T6	$s = t_{\text{referència}}/t_{nou}$; a les solucions, el de referència també és t_{orig}
t_p	Temps de penalització per fallada	T7	
t_{pd}, t_{pi}	Temps de penalització de MC_d / MC_i	T7	Memòria cau dividida: dades / instruccions
t_{proc}	Temps de la resta d'operacions d'una instrucció (sense accés a memòria)	T7	

Símbol	Significat	Tema	Observacions
T	Mida de la pàgina (en bytes)	T8	Condició VIPT: $N_C \cdot B \leq T$
T	Mida en bytes d'un element d'un vector o matriu	T2, T4	@`vec[i]` = @`vec[0]` + $i \cdot T$
TAM	Temps d'accés mitjà	T7	Vegeu «TAM» a Sigles; TAM = $t_h + m \cdot t_p$
ULP	Valor d'un sol bit a 1 a la posició de menys pes de la mantissa	T5	Vegeu «ULP» a Sigles; s'usa com a unitat: $\eta_{max} < 2^{-23} = 1$ ULP
v	Valor representat (IEEE 754)	T5	
v	Bit de desbordament del sumador	T4	$v = c_{n-1} \oplus c_n$
v_0, v_1	Valors representables adjacents més propers a un valor exacte v	T5	$v_0 < v < v_1$
v_{max}	Valor positiu més gran representable	T5	$v_{max} = (2 - 2^{-23}) \times 2^{127}$ en precisió simple
V	Codificació (cadena de 32 bits) d'un valor v en coma flotant	T5	Majúscula per a la cadena i minúscula per al valor, com X i x_u a T1
V	Bit de validesa (*valid bit*)	T7, T8	
$V(t)$	Tensió de sortida d'una porta en funció del temps	T6	$V(t) = V_{CC}(1 - e^{-t/RC})$ en la càrrega i $V_{CC} \cdot e^{-t/RC}$ en la descàrrega
V_{CC}, V_{in}, V_t	Tensió d'alimentació / d'entrada / llindar	T6, T7	V_{CC} es reutilitza a T7 en la potència dinàmica de la memòria cau ($P_d = \alpha \cdot C \cdot V_{CC}^2 \cdot f_{clock}$)
w	Nombre de bits d'una adreça	T7	El nombre de bits de l'etiqueta ($w - n_c - b$) no té símbol propi
x, y, z	Multiplicand, multiplicador i producte ($z = x \times y$); dividend i divisor ($x = y \cdot q + r$) (T4); operands i resultat d'una operació en coma flotant ($z = x + y$) (T5)	T4, T5	A la divisió, el quocient i el residu són q i r
x_s	Valor enter (amb signe) representat	T1, T3	$x_s \in [-2^{n-1},\ 2^{n-1} - 1]$
x_u	Valor natural (sense signe) representat	T1, T3	$x_u \in [0,\ 2^n - 1]$
y_i	Bit i-èsim del multiplicador	T4	$z = \sum_i (x \cdot y_i) << i$ (circuit combinacional)
X, X_i, X_{n-1}	Cadena de bits / bit i-èsim / bit de més pes (signe en Ca2)	T1, T3	A T4 (Equació 4.4), @X_i és l'adreça de l'element i-èsim d'un recorregut, no un bit
α	Factor d'activitat (model de potència dinàmica)	T6	
ε, ε_{max}	Error absolut de representació / error absolut màxim	T5	$\varepsilon = \lvert v - \tilde{v} \rvert$, amb $\tilde{v} \in \{v_0, v_1\}$ el valor arrodonit
η, η_{max}	Error relatiu de representació / error relatiu màxim	T5	$\eta = \varepsilon / \lvert v \rvert$; $\eta_{max} < 2^{-23}$ en precisió simple

Notació

Operadors i convencions de representació. La majoria són transversals a tot el llibre, estables amb independència del tema on apareixen; quan una notació és pròpia d'un sol tema, s'indica a la columna «Tema» (en cas contrari es deixa en blanc). Els matisos o ambigüitats es documenten a la columna

«Observacions».

Taula 6.6: Operadors i convencions de notació.

Notació	Significat	Tema	Observacions
$<<$	Desplaçament a l'esquerra (`sll`/`slli`)		
$>>$	Desplaçament a la dreta		**Lògic o aritmètic segons el context**: en C es distingeix pel tipus de l'operand (`unsigned` o amb signe), no per un símbol diferent
$>>>$	Desplaçament aritmètic a la dreta (`sra`/`srai`)		A les taules ISA d'instruccions (Secció 3.1.2); **no coincideix** amb la notació `>>` de C, que distingeix lògic/aritmètic pel tipus de l'operand en lloc d'un símbol propi
$\lfloor x \rfloor$	Part entera (truncament cap avall)		
$\lceil x \rceil$	Arrodoniment cap amunt (enter més petit que no és menor que x)		
$\lvert x \rvert$	Valor absolut		
mod	Operador mòdul (resta de la divisió entera)		A les taules ISA (`rem`, `remu`), el residu de la divisió truncada cap a zero, amb el signe del dividend, com `%` de C
$\div$	Divisió entera		
$/$	Divisió (no entera)		P. ex. $rd \leftarrow rs1/rs2$ (`fdiv.s`)
$\oplus$	O-exclusiva (XOR) bit a bit		
$\wedge$	Conjunció lògica (AND) bit a bit		
$\vee$	Disjunció lògica (OR) bit a bit		
$\overline{X}$	Negació / complement bit a bit		Usat en Ca1
$<_u$	Comparació sense signe (dels valors naturals)		P. ex. $S <_u A$; a les taules ISA, la mateixa distinció es marca amb $(\mathbb{N})$
$(\mathbb{Z})$, $(\mathbb{N})$	La comparació interpreta els operands com a enters (amb signe) / com a naturals (sense signe)	T3	A les taules ISA de comparació i de salt condicional (`slt`/`sltu`, `blt`/`bltu`)
$\equiv \pmod m$	Congruència mòdul m		Aritmètica en Ca2
$\text{Ca2}(x)$	Representació en Ca2 de x	T1	P. ex. $\text{Ca2}(a) + \text{Ca2}(b) \pmod{2^n}$
x_2, x_{10}	Base de numeració, en subíndex		P. ex. $1011_2 = 11_{10}$; en hexadecimal, el prefix `0x` del codi
`@`*etiqueta*	Adreça de l'element o etiqueta indicada		P. ex. `@vec[i]`, `@etiqueta`
`[a:b]`	Rang de bits d'un registre o cadena, del bit a al bit b (inclusius)		P. ex. `R[2n-1:n]`, `lo_12[11]`, $rs2[4:0]$
$\mathcal{O}(1)$	Cost constant (accés aleatori)	T2, T4	
$rd \leftarrow x$	Assignació: el destí rep el valor x	T2	Notació de transferència de registres de les taules ISA, on registres i camps van en cursiva (rd, $rs1$, PC, $mepc$)

Notació	Significat	Tema	Observacions
$M_w[a]$, $M_h[a]$, $M_b[a]$	Paraula / mitja paraula / byte de la memòria a l'adreça a	T2	P. ex. $rd \leftarrow M_w[rs1 + SignExt(offset)]$
$(int)\,x$, $(unsigned)\,x$, $(float)\,x$	Conversió de tipus; de flotant a enter, arrodonida amb el mode rm de la instrucció (el *cast* de C trunca: $rm =$ `rtz`)	T5	Taules ISA d'RV32F (`fcvt`)
$c\,?\,x : y$	Si es compleix la condició c, x; si no, y	T3	P. ex. $rd \leftarrow (rs1 < rs2)\,?\,1 : 0$
$SignExt()$	Extensió de signe: replica el bit de signe cap a l'esquerra	T2	
$ZeroExt()$	Extensió per zero: afegeix zeros cap a l'esquerra	T2	
$hi_{20}()$	20 bits superiors d'un immediat de 32 bits	T2	
$lo_{12}()$	12 bits inferiors d'un immediat de 32 bits	T2	Si $lo_{12}[11] = 1$, l'assemblador incrementa hi_{20} en 1 per compensar l'extensió de signe

Termes

Equivalències entre els termes anglesos i els catalans que el llibre presenta amb l'anglès entre parèntesis, amb el tema on es presenten per primer cop (o el laboratori, L1–L6, o el compendi). És el lèxic del llibre: hi és cada terme tal com el presenta el llibre, amb les observacions d'ús que calgui. La columna «Fonts» diu d'on surt la forma catalana: el diccionari de l'IEC (DIEC), el Termcat, l'Optimot, el diccionari de Softcatalà o la Viquipèdia, per aquest ordre de precedència i amb l'enllaç a la consulta; «Sintagma», si cap font no la té i està feta amb mots que sí que hi són; i «Llibre», si és una decisió del llibre, explicada a les observacions. La segona taula recull els termes que el llibre manté en anglès, perquè el terme català no existeix o és molt poc usat, amb el motiu de cadascun.

Taula 6.7: Termes anglesos i la forma catalana que fa servir el llibre.

Anglès	Català	Tema	Fonts	Observacions
Addressing mode	Mode d'adreçament	T2	Viquipèdia	
Aliasing	Aliàsing	T8	Llibre	Grafia adaptada.
Alignment constraint	Restricció d'alineació	T2	Sintagma	
Ampersand	Et	T2	Termcat, Viquipèdia	El nom del signe `&`.
Anticipatory paging	Lectura anticipada	T8	Sintagma	
Array	Vector	T2	DIEC	
Bandwidth	Amplada de banda	T7	DIEC, Termcat, Softcatalà, Viquipèdia	
Bare-metal	Entorn autònom	T1	Sintagma	
Base + offset mode	Mode base + desplaçament	T2	Sintagma	

Anglès	Català	Tema	Fonts	Observacions
Biased exponent	Exponent esbiaixat	T5	Sintagma	En el format IEEE 754. El valor numèric del biaix s'anomena «biaix», no «excés»; la codificació general és la codificació en excés (Secció 1.5.2).
Borrow	Préstec	T1	Sintagma	
Branch	Salt condicional	T3	Llibre	El Termcat en diu «bifurcació», que no distingeix el salt condicional de l'incondicional (*jump*), com fa el llibre.
Breakpoint	Punt de ruptura	L1	Termcat	
Buffer overflow	Desbordament de memòria intermèdia	T2	Sintagma	
Bytecode	Codi de bytes	T1	Termcat	
Cache	Memòria cau	T7	Termcat, Softcatalà, Viquipèdia	Sempre «memòria cau», sense la forma curta «la cau» (fora de les etiquetes de les fórmules, per l'espai).
Callback	Crida de retorn	T2	Termcat	
Capacity	Fallada de capacitat	T7	Sintagma	
Carry	Ròssec	T1	Termcat, Softcatalà, Viquipèdia	
Carry-in	Ròssec d'entrada	T4	Sintagma	
Carry-out	Ròssec de sortida	T1	Sintagma	
Clock gating	Inhibició del rellotge	T6	Sintagma	
Coherence	Fallada de coherència	T7	Sintagma	
Cold-start	Arrencada en fred	T7	Llibre	
Compulsory	Obligatòria	T7	Softcatalà	
Conflict	Fallada de conflicte	T7	Sintagma	
Copy-back	Escriptura retardada	T7	Sintagma	
Cross-compilation	Compilació creuada	T1	Sintagma	
Cross-compiler	Compilador creuat	T3	Viquipèdia	
Data type	Tipus de dades	T2	Termcat	
Decode	Descodificació	T1	DIEC, Termcat, Softcatalà	
Denormalized	Denormal	T5	Llibre	El terme de l'IEEE 754 del 1985. El llibre fa servir *subnormal*, «subnormal».
Direct-mapped	Correspondència directa	T7	Sintagma	
Dirty bit	Bit de modificació	T7	Sintagma	
Dirty hack	Truc brut	T2	Sintagma	
Dispatcher	Distribuïdor	T9	Termcat	

Anglès	Català	Tema	Fonts	Observacions
Embedded system	Sistema encastat	T1	Termcat, Viquipèdia	
Endianness	Ordre dels bytes	T2	Termcat, Viquipèdia	Els dos ordres, *little-endian* i *big-endian*, en anglès (segona taula).
Escape sequence	Seqüència d'escapada	T2	Sintagma	Com «caràcter d'escapada» (*escape character*), del Termcat.
Event	Esdeveniment	T9	Termcat, Softcatalà, Viquipèdia	
Exception	Excepció	T9	Termcat	
Exception handler	Rutina específica	T9	Sintagma	La rutina que tracta una causa concreta, a la qual l'RSE salta o que el distribuïdor crida (T9).
Execute	Execució	T1	Sintagma	
Fan-out	Connectivitat de sortida	T6	Termcat	
Fetch	Captura	T1	Llibre	El Termcat només en té el verb, «recollir», i hi fa servir «captura» per a *capture*.
Field	Camp	T2	DIEC, Termcat, Viquipèdia	
Firmware	Microprogramari	T9	Termcat, Viquipèdia	
First-reference	De primera referència	T7	Sintagma	
Flash	Memòria flaix	T1	Termcat, Viquipèdia	
Floating point	Coma flotant	T1	Viquipèdia	
Frame pointer	Punter de marc	T2	Sintagma	
Full-adder	Sumador complet	T4	Sintagma	El DIEC en diu «sumador binari».
Fully associative	Memòria cau completament associativa	T7	Sintagma	
Fused multiply-add	Multiplicació i suma fusionades	T2	Sintagma	
Gap	Bretxa	T7	Softcatalà	Entre el rendiment del processador i la latència de la memòria (T7).
Gate leakage	Fuita de porta	T6	Sintagma	
General-purpose	Propòsit general	T2	Softcatalà	
Global pointer	Punter global	T2	Sintagma	
Gradual underflow	Subdesbordament gradual	T5	Sintagma	
Half-adder	Semisumador	T4	DIEC	

Anglès	Català	Tema	Fonts	Observacions
Hardware	Maquinari	T1	DIEC, Termcat, Optimot, Softcatalà, Viquipèdia	També l'abreviatura *HW*.
Hart	Fil d'execució de maquinari	T2	Sintagma	
Hidden bit	Bit ocult	T5	Sintagma	
Hit	Encert	T7	Sintagma	
Host	Màquina amfitriona	T1	Sintagma	
Hosted	Allotjat	T1	Sintagma	
Immediate mode	Mode immediat	T2	Sintagma	
Internal linkage	Vinculació interna	T3	Sintagma	
Interrupt	Interrupció	T9	Termcat	
Jump	Salt incondicional	T3	Sintagma	
Jump table	Taula de salts	T3	Viquipèdia	
Label	Etiqueta	T2	DIEC, Termcat, Softcatalà	
Lazy	Tardana	T3	Softcatalà, Viquipèdia	En «avaluació tardana» (*lazy evaluation*).
Leaf	Fulla	T3	Sintagma	
Linker	Enllaçador	T3	Viquipèdia	
Linking	Enllaçament	T1	Sintagma	
Load	Lectura	T2	Llibre	Les instruccions que llegeixen la memòria (`lw`, `flw`…) i la seva operació, també el verb (*llegir*). No ho és la càrrega d'immediats o d'adreces (`li`, `lui`, `la`), la del programa, la d'una pàgina ni la d'un bloc a l'MC.
Load/store	Lectura/escriptura	T2	Llibre	
Loader	Carregador	T3	Softcatalà, Viquipèdia	
Local variable	Variable local	T2	Termcat, Viquipèdia	
Major page fault	Fallada major	T8	Sintagma	
Memory addressing mode	Mode memòria	T2	Sintagma	
Memory dump	Abocament de memòria	T2	Termcat, Softcatalà, Viquipèdia	
Minor page fault	Fallada menor	T8	Sintagma	
Miss	Fallada	T7	Sintagma	
Multicore	Multinucli	T6	Viquipèdia	Adjectiu invariable: «processador multinucli», «disseny multinucli».

Anglès	Català	Tema	Fonts	Observacions
Nesting	Imbricació	T2	DIEC, Termcat	
Nibble	Quartet	T1	Viquipèdia	Un grup de 4 bits, un dígit hexadecimal.
No-write allocate	Escriptura sense assignació	T7	Sintagma	
Non-leaf	No fulla	T3	Sintagma	
Null pointer	Punter nul	T2	Viquipèdia	
Offset	Desplaçament	T2	Termcat	
Open standard	Estàndard obert	T2	Viquipèdia	
Overflow	Desbordament	T1	Termcat, Softcatalà	El de la família de *underflow* («subdesbordament») i *buffer overflow*.
Padding	Farciment	T2	Termcat, Softcatalà	El símbol `PAD` del codi es manté.
Page fault	Fallada de pàgina	T8	Sintagma	
Page frame	Marc de pàgina	T8	Sintagma	
Page offset	Desplaçament	T8	Termcat	
Page-table walker	Recorregut de la taula de pàgines	T8	Sintagma	També *walker*, sol.
PC-relative mode	Mode relatiu al PC	T2	Sintagma	
Placement policy	Política d'emplaçament	T7	Sintagma	
Polling	Sincronització per enquesta	T9	Sintagma	
Position-independent	Independent de la posició	T2	Sintagma	
Power gating	Desconnexió de l'alimentació	T6	Sintagma	
Precise exceptions	Excepcions precises	T9	Sintagma	
Precleaning	Neteja anticipada	T8	Sintagma	
Prefetching	Lectura anticipada	T8	Sintagma	
Profile	Perfil	T2	Sintagma	
Pseudoinstruction	Pseudoinstrucció	T2	Llibre	El Termcat i Softcatalà en diuen «directriu», que és també *directive*: el llibre distingeix les pseudoinstruccions de les directives.
Random	Aleatori	T7	DIEC, Termcat, Softcatalà	
Reference bit	Bit de referència	T8	Sintagma	
Register	Registre	T2	DIEC, Termcat, Viquipèdia	
Register mode	Mode registre	T2	Sintagma	
Regularity	Regularitat	T2	Softcatalà	
Return address	Adreça de retorn	T2	Sintagma	

Anglès	Català	Tema	Fonts	Observacions
Ripple carry	Propagació del ròssec	T4	Sintagma	
Round toward zero	Arrodoniment a zero	L5	Sintagma	
Segmentation fault	Error de segmentació	T2	Sintagma	
Set	Conjunt	T7	Softcatalà	
Set-associative	Memòria cau associativa per conjunts	T7	Sintagma	
Significand	Mantissa	T5	DIEC, Softcatalà, Viquipèdia	La part 1,F del nombre en coma flotant, bit ocult inclòs; la fracció, F, és la part emmagatzemada (13_contrib.qmd §T5, D-11). No el calc «significand».
Single precision	Precisió simple	T2	Termcat	
Software	Programari	T1	DIEC, Termcat, Optimot, Softcatalà, Viquipèdia	També l'abreviatura *SW*.
Software-managed TLB	TLB gestionat per programari	T8	Sintagma	
Speedup	Guany	T6	Llibre	Softcatalà en dona «acceleració»; el llibre en fixa «guany» (13_contrib.qmd §T6).
Stack	Pila	T3	DIEC, Termcat, Softcatalà	
Stack frame	Bloc d'activació	T3	Sintagma	
Stack pointer	Punter de pila	T2	Sintagma	
Store	Escriptura	T2	Llibre	Les instruccions que escriuen la memòria (`sw`, `fsw`…) i la seva operació, també el verb (*escriure*). No ho és l'emmagatzematge com a disposició o com a suport (secundari, per files, l'ordre dels bytes).
Stream	Flux	T1	Softcatalà, Viquipèdia	El Termcat en diu «corrent de dades»; el llibre en fa servir «flux», la forma de Softcatalà i de la Viquipèdia («flux de dades»), que és la de la programació.
String	Cadena de caràcters	T2	Sintagma	O «cadena», si surt sovint.

Anglès	Català	Tema	Fonts	Observacions
Subnormal	Subnormal	T5	Llibre	El terme de l'estàndard IEEE 754 des del 2008; abans, *denormalized*, «denormal».
Subthreshold leakage	Fuita subllindar	T6	Sintagma	
Swap space	Espai d'intercanvi	T8	Sintagma	
Syscall	Crida al sistema	T1	Sintagma	
Tag	Etiqueta	T7	DIEC	
Thread pointer	Punter de fil	T2	Sintagma	
Throughput	Productivitat	T6	Llibre	La fitxa del Termcat, «cabal», és de telecomunicacions.
Time sharing	Temps compartit	T8	DIEC, Termcat, Softcatalà, Viquipèdia	
TLB hit	Encert de TLB	T8	Sintagma	
TLB miss	Fallada de TLB	T8	Sintagma	
Toolchain	Cadena d'eines	T1	Sintagma	
Type cast	Conversió de tipus	T2	Termcat, Softcatalà, Viquipèdia	
Undefined behavior	Comportament indefinit	T2	Sintagma	
Underflow	Subdesbordament	T5	Termcat	
Valid bit	Bit de validesa	T7	Sintagma	
Way	Via	T7	Sintagma	
Word	Paraula nativa	T2	Llibre	
Write allocate	Escriptura amb assignació	T7	Sintagma	
Write buffer	Memòria intermèdia d'escriptura	T7	Sintagma	*Buffer*, sol, és «memòria intermèdia».
Write-back	Escriptura retardada	T7	Sintagma	
Write-through	Escriptura immediata	T7	Sintagma	

Taula 6.8: Termes mantinguts en anglès.

Anglès	Terme català	Motiu
Caller, *callee*, *saver*	«La funció o subrutina que crida», «la funció o subrutina cridada»; *Saver*, —	Al text, la perífrasi catalana; a les figures i a les taules de registres, l'anglès, per l'espai: la columna *Saver* hi diu qui desa el registre, *Caller* o *Callee*.
Caller-saved, *callee-saved*	—	Termes de l'ABI.
Doubleword	—	El nom de la dada de 64 bits a l'especificació de RISC-V (A2).

Anglès	Terme català	Motiu
Fall-through	—	El pas d'una clàusula `case` a la següent sense `break` (A3). Cap font no en té el terme.
Half	—	El nom del format de coma flotant de 16 bits de l'IEEE 754 (P5, S5).
Heap	«Memòria en monticle» (Termcat)	El terme català és molt poc usat; «monticle», sol, és l'estructura de dades.
Little-endian, *big-endian*	«Ordre dels bytes petit-gran», «gran-petit» (Termcat)	Els termes catalans són molt poc usats; el concepte, l'ordre dels bytes (*endianness*), és en català.
Loop tiling, *loop blocking*	—	La transformació de bucles per blocs (L6). Cap font no en té el terme; «divisió en talls» i «tessel · lació» són d'altres sentits.
Pipeline	«Canal» (Termcat)	El terme català és molt poc usat en arquitectura de computadors, i es confon amb un canal de comunicació; el Termcat admet *pipeline* com a sinònim complementari. Només surt en un aprofundiment d'A9.
Round-robin	—	La política de reemplaçament o de planificació per torns (A9). Cap font no en té el terme.
Segfault	—	El nom col · loquial de l'error de segmentació (*segmentation fault*, A2).
Sticky	—	El bit S del sumador (G, R i S) i els indicadors de `fflags`: el Softcatalà només en dona el sentit general («enganxós»), i l'Optimot no en té cap terme.
Stride	—	La distància entre dos accessos consecutius a la memòria. El Termcat només en té el sentit esportiu («gambada»), i el Softcatalà, el general («pas»).
Target	—	No és a l'Optimot amb aquest sentit. A1 el glossa com «la màquina destí».
TLB shootdown	—	La invalidació de les entrades del TLB dels altres nuclis (A8). Cap font no en té el terme.
Working set	—	El conjunt de blocs o de pàgines que un programa fa servir en un interval de temps. Cap font no en té el terme.

Contribueix-hi

Aquest document de convencions només està disponible en la versió web:

https://rbaig.github.io/migracio.ec.gitlab.upc.edu/

Llicència i reconeixements

Llicència



Creative Commons Reconeixement-NoComercial-CompartirIgual 4.0 Internacional (CC BY-NC-SA 4.0).

© 2026, Fernando Agraz, Adrià Armejach, Roger Baig, Octavio Castillo, Josep-llorenç Cruz, Agustín Fernández, Pedro José Martínez, Miquel Morató, Rubèn Tous.

Reconeixements

Projecte d'innovació docent

Aquesta obra s'ha elaborat dins del projecte d'innovació docent «Migració a RISC-V: una oportunitat per actualitzar la metodologia docent i els continguts d'Estructures de computadors (EC)», finançat per la Facultat d'Informàtica de Barcelona (FIB). La proposta del projecte és a people.ac.upc.edu/rbaig.

Materials elaborats per professors d'EC

En l'elaboració d'aquesta obra s'ha fet servir les publicacions següents, totes amb llicència Creative Commons Reconeixement-NoComercial-CompartirIgual 2.5 Espanya (CC BY-NC-SA 2.5 ES):

Referència	Títol	Data
[15]	Tema 1: Introducció	03/2020
[16]	Tema 2: Instruccions i tipus de dades bàsics	02/2022
[17]	Tema 3: Traducció de Programes	09/2019
[18]	Tema 4: Matrius	03/2016
[19]	Tema 5: Aritmètica d'enters i coma flotant	03/2020
[20]	Tema 6: Memòria Cache	02/2023
[21]	Tema 7: Memòria Virtual	06/2018
[22]	Tema 8: Excepcions i Interrupcions	06/2018
[23]	Quadern de laboratori, Sessió 0: Introducció	2014/15
[24]	Quadern de laboratori, Sessió 1: Assemblador MIPS i tipus bàsics de dades	2014/15
[25]	Quadern de laboratori, Sessió 2: Traducció de Programes	2014/15
[26]	Quadern de laboratori, Sessió 3: Tipus de dades estructurats	2014/15
[27]	Quadern de laboratori, Sessió 4: Codificació en coma flotant	2014/15
[28]	Quadern de laboratori, Sessió 5: Memòria cache	2014/15
[29]	Col · lecció de Problemes	2014/15
[30]	Solucionari de la Col · lecció de Problemes	2014/15

Intel · ligències artificials generatives

En l'elaboració d'aquesta obra s'ha fet ús d'eines d'intel · ligència artificial generativa, principalment Claude (Anthropic), per a les tasques següents:

- Extracció i estructuració de contingut a partir dels materials originals.
- Suggeriment de contingut, organització i estructura.
- Revisió tècnica, lingüística i pedagògica.

Tot el contingut generat o suggerit per eines d'IA ha estat revisat, contrastat i validat per professors de l'assignatura.

Programari

Aquesta obra s'ha elaborat exclusivament amb eines de programari lliure, essent les més rellevants:

- Quarto — sistema de publicació científica i tècnica.
- LaTeX — sistema de composició tipogràfica.
- MathML — llenguatge de marcatge matemàtic.
- Inkscape — editor de gràfics vectorials.
- Firefox — navegador web.
- Evince — visualitzador de documents.
- GNU/Linux Debian — sistema operatiu.