*Introducció als Ordinadors*

# Pràctica 1

Mario Vilar (MV)

31 de març de 2022

## Índex

# 1 Introducció

Treballarem amb el simulador *Ripes-RiscV*. Per la mateixa arquitectura, Ripes ens permet treballar amb diferents versions, cada una amb diferents compromisos entre velocitat, complexitat, cost i forma de treballar. En particular treballarem amb la forma més bàsica associada a aquesta arquitectura, un Processador de un sol cicle (Single cycle processor) i un processador multicicle Tots dos tenen instruccions de mida de 32 bits ja que tenen la mateixa arquitectura. En canvi, l'estructura de la CPU (microarquitectura) és lleugerament diferent. Tots dos tenen una estructura Harvard, amb la memòria de dades per una banda i la memòria de programa per l'altre. *Ripes Single Cycle Processor* (RSCP) presenta una microarquitectura simple, que permet executar instruccions en un sol cicle de rellotge. Això fa que el temps de cicle del rellotge hagi de tenir present el pitjor cas, és a dir, que ajustarem el temps a aquella instrucció que més triga en executar-se. Per altra banda el mateix simulador ens permet treballar amb un Ripes amb instruccions multicicle i amb execució simultània de varies instruccions, el que es coneix com pipeline, *Ripes 5-Stage Processor* (R5SP). Això vol dir que mentre estàs executant una instrucció, pots anar carregant una altra instrucció i pots anar decodificant una altra de manera simultània, ja que aquests processos fant servir diferents recursos.

El conjunt d'instruccions d'un processador ens reflexa directament la seva arquitectura. La implementació d'un algorisme per tal de solucionar un problema implica la realització d'un programa que es realitzarà tenint en compte les instruccions que pot interpretar l'assemblador. La majoria dels processadors tenen el mateix tipus de grups d'instruccions, tot i que no necessàriament han de tenir el mateix format ni el mateix nombre d'instruccions per a cada grup. De fet, cada arquitectura té el seu propi llenguatge màquina i en conseqüència el seu propi conjunt d'instruccions.

## 1.1 Objectius

Els objectius d'aquesta pràctica són:

1. entendre el funcionament de la `CPU` i la seva connexió amb la memòria principal;
2. familiaritzar-se amb el simulador `Pipeline` en les seves dues versions amb què treballarem: *RSCP* i *R5SP*;
3. entendre què fan les instruccions;
4. comprendre l'analogia entre llenguatge màquina i el llenguatge assemblador;
5. analitzar i veure en la pràctica conceptes com el `pipeline` o execució pseudoparal·lela de les instruccions.

## 1.2 Exercicis diapositives

**Exercici 1.** *A partir del següents nombres binaris: A = 1001 0011 i B = 0011 1100. Feu les següents operacions* `A · B`, `A AND B`, `A+B`, `A OR B`. *Comenteu els resultats.*

*Resolució.* El nombre `A` és 147 en decimal i `B` és 60. Aleshores, `A · B` resulta la multiplicació dels dos nombres en decimal, és a dir, `8820` o, si ho transformem un altre cop a binari, `10001001110100`. Cal no confondre-ho amb l'operació `AND`, que s'ha de fer bit a bit, i `A AND B` resulta ser `0001 0000` en binari o, passant-ho a decimal, `16`.

Seguint un procediment anàleg amb la suma obtenim `207` (la suma dels dos nombres, `1100 1111` en base 2) i, en canvi, `A OR B` es resol rutinàriament en binari i dona `1011 1111`, o `191` en decimal. ◁

**Exercici 2.** *Transforma en representació de coma flotant el següent valor: 3,14. Un cop trobat el valor, feu el procés invers per recuperar el valor.*

*Resolució.*

1. Comencem representant en binari el 3, que és 11 i agafem tots els decimals que puguem de 0,14 agafant la part entera i doblant iterativament la part decimal. Desplacem la coma cap a l'esquerra multiplicant per $2^1$ i ens queda:

$$1.10010001111010111000010 1 \cdot 2^1. \tag{1.1}$$

   L'exponent és 127+1 = 128, que en binari resulta 10000000. El nombre final, inclòs el signe i la mantissa, és 01000000010010001111010111000010 1.
2. Pel que fa al procés contrari, ens adonem que el nombre és positiu ja que el primer bit és un 0, el valor de l'exponent és 1 per reciprocitat i la mantissa resulta:

$$2^{-3}+2^{-7}+2^{-8}+2^{-9}+2^{-10}+2^{-12}+2^{-14}+2^{-15}+2^{-16}+2^{-21}+2^{-23} = 0.569999694. \tag{1.2}$$

   I el resultat final és:

$$(2^0 + 2^{-1} + 2^{-4} + 2^{-10} + 2^{-18}) \cdot 2^1 = 3.139998. \tag{1.3}$$

◁

**Exercici 3.** *Feu un programa que ens indiqui si un determinat valor obtingut per una operació prèvia és positiu o negatiu fent servir màscares i operacions lògiques. Troba la màscara que has de fer servir per determinar que un número és negatiu. Considerant que el valor està donat en Ca2, passa-ho al valor positiu que correspondria (ex.* $-7 \implies 7$*). Troba diverses solucions al problema.*

*Resolució.* ◁

**Exercici 4.** *La instrucció* `MOV A, M` *guarda en el registre* `A` *el contingut que hi ha a la posició de memòria* `M`*. Indiqueu el contingut del registre* `A` *en funció dels possibles valors de* `M`*.*

*Resolució.* L'adreça de la posició de memòria de `M` pot ser diversos valors `@i`, amb $i = 1, \ldots, 6$. En particular, podem fer la següent assignació segona la taula proporcionada:

1. Si `M=0`, `A=65432`,
2. si `M=1`, `A=23457`,
3. si `M=2`, `A=67`,
4. si `M=3`, `A=33223`,
5. si `M=4`, `A=5435`,
6. si `M=5`, `A=65434` i
7. si `M=6`, `A=1234`.

◁

**Exercici 5.** *Tenim una memòria de 4kBytes. La unitat de memòria és el word (32 bits). Quants bits necessitem per adreçar-la? Analitzeu i determineu la diferència entre adreça i contingut.*

*Resolució.* `4KB` són `4000` bytes que, al seu torn, són `32000` bits. Si la distribució es dona en `32` bits, Tenim `1000` espais de memòria en total. Ara, per adreçar el nombre de bits:

$$2^n = 1000 \iff n = \frac{\ln(1000)}{\ln(2)} \approx 10. \tag{1.4}$$

Per tant, necessitem 10 bits per adreçar-la. ◁

**Exercici 6.** *La instrucció* `LOAD R1`*, A carrega el contingut que hi ha a la posició de memòria* `A` *en el registre* `R1`*. Si la memòria és la de l'exercici 1, quants bits ens calen per identificar el valor de A? Si tenim 8 registres de propòsit general* `R0, ..., R7`*. Quants bits calen per identificar el registre* `R1`*? Si el conjunt d'instruccions total que té aquest processador és de* `100` *instruccions, quants bits calen per identificar la instrucció* `LOAD`*?*

*Resolució.* Per l'exercici anterior, els bits necessaris per identificar `A` en el registre `R1` són `10` bits. Per identificar un de `8` registres, `R1`, necessitem `3` bits ($2^3$). Per identificar la instrucció `LOAD`, ens calen:

$$2^m = 100 \iff m = \frac{\ln(100)}{\ln(2)} \approx 7. \tag{1.5}$$

Per tant, necessitarem 7 bits. ◁

# 2 Cos del treball

## 2.1 Exercicis

**Exercici 7.** *Indiqueu quines de les següents instruccions són incorrectes segons les especificacions anteriorment indicades. Expliqueu el per què de cada cas:*

*Resolució.* La resolució seria la següent:

| Instruccions | Correcte/incorrecte | Motiu |
|---|---|---|
| `LW a1(R0),a1` | incorrecte | `LW a1,0(a1)` |
| `SW a1,a1(3)` | incorrecte | `SW a1,3(a1)` |
| `BNE 6(t1)` | incorrecte | `BNE 6(t1)` |
| `ADDI t2,#11,5(t3)` | incorrecte | `ADDI t2,t3,11` |
| `SUB zero,a2,a3` | correcte | |
| `LOAD 3(a0),a1` | incorrecte | `LOAD a1,3(a0)` |

Amb el següent programa de Ripes fem les modificacions adequades a les instruccions incorrectes. Això cobriria la pregunta *Captura de pantalla del simulador RIPES amb les modificacions adequades a les instruccions incorrectes*.

```
.data
aa: .word 0
.text
main:
    lw a1, 0(a1)
    sw a1, 3(a1)
    bne t1, t2, etiqueta
etiqueta:
    addi t2, a3, 11
    sub zero, a2, a3
    lw a1, 3(a0)
```

◁

**Exercici 8.** *Suposem les dades de l'enunciat de la pràctica i les instruccions:*

1. `lw a1,0(a4)`
2. `sw a1,0(a5)`
3. `j` *etiqueta*
4. `addi a3,a2,11`
5. `SUB zero,a2,a3`
6. `srli a1,a1,1`

*Indiqueu què fa cadascuna de les instruccions següents, cal explicitar totes les alteracions que es produeixen a la memòria, bits de condició, registres i valor final. Per cada una de les instruccions sempre partiu de les condicions inicials descrites més a dalt. Feu una breu descripció del que fa cada instrucció.*

*Resolució.*

1. `lw a1,0(a4)`: guarda, o més ben dit, carrega en el registre `a1` el contingut apuntat per `[a4]+0`. Per tant, el seu valor final és `F45A`.

2. `sw a1,0(a5)`: instrucció de 32 bits que guarda el contingut del registre `a1`, que ja hem modificat, a la posició de memòria apuntada per `a5`. Per tant, el valor final de `M[a5] = M[address2]` és `F45A`.
3. `j` etiqueta: el PC apuntarà a l'etiqueta designada.
4. `addi a3,a2,11`: es guarda a `a3` el resultat de la suma immediata entre `a2` i `11`. Aleshores, resulta que el valor final d'`a3` és `A5EE = A5E3+11`.
5. `SUB zero,a2,a3`: `zero` ja està determinada i no es pot modificar, de tal manera que, per molt que es produeixi la operació, en teoria mai no s'emmagatzemaria al `zero`.
6. `srli a1,a1,1`: és un *shift right* immediat, així que fa un desplaçament a la dreta del contingut que hi ha a `a1` en el nombre de bits que indica el tercer valor, així que s'emmagatzemarà a `a1` el valor d'`a1` desplaçat un bit cap a la dreta, en aquest cas `7A2D`. Podem comprovar-ho passant `F45A` a decimal, dividint per 2 (desplaçament a la dreta) i tornant a hexadecimal.

◁

**Exercici 9.** *Escriviu en llenguatge màquina el següent programa, feu-ho en hexadecimal i binari:*

```
.data
xx: .word 3
yy: .word 5
oo: .word 2
zz: .word 8

.text
la a0,xx
sub a1,a1,a1
add a2,zero,zero
loop: addi a7,a1,-4
beqz a7,final
lw a3,0(a0)
add a2,a3,a3
addi a0,a0,4
addi a1,a1,1
j loop
final: sw a2,4(a0)
```

Figura 1: Codi usat

| @ | M[@] | Llenguatge màquina | Hex | Binari |
|---|---|---|---|---|
| .data | | | | |
| xx: | .word 3 | | | |
| yy: | .word 5 | | | |
| oo: | .word 2 | | | |
| zz: | .word 8 | | | |
| .text | | | | |
| 00h | la a0,xx | auipc x10 0x10000 | 10000517 | 00010000000000000000010100010111 |
| 08h | sub a1,a1,a1 | sub x11 x11 x11 | 40b585b3 | 01000000101101011000010110110011 |
| 0ch | add a2,zero,zero | add x12 x0 x0 | 00000633 | 00000000000000000000011000110011 |
| 10h | loop: addi a7,a1,-4 | addi x17 x11 -4 | ffc58893 | 11111111111000101100010010010011 |
| 14h | beqz a7,final | beq x17 x0 24 <final> | 00088c63 | 00000000000010001000110001100011 |
| 18h | lw a3,0(a0) | lw x13 0(x10) | 00052683 | 00000000000001010010011010000011 |
| 1ch | add a2,a3,a3 | add x12 x13 x13 | 00d68633 | 00000000110101101000011000110011 |
| 20h | addi a0,a0,4 | addi x10 x10 4 | 00450513 | 00000000010001010000010100010011 |
| 24h | addi a1,a1,1 | addi x11 x11 1 | 00158593 | 00000000000101011000010110010011 |
| 28h | j loop | jal x0 0x10 <loop> | fe9ff06f | 11111110100111111111000001101111 |
| 2ch | final: sw a2,4(a0) | sw x12 4(x10) | 00c52223 | 00000000110001010010001000100011 |

## 2.2 Qüestions

**Exercici 10.** *Pel programa de l'apartat 3 de l'estudi guiat, les posicions de memòria que van des de la* `xxh` *a la* `zzh`*, ambdues incloses, contenen una seqüència del quatre números 3, 5, 2, 8, emmagatzemats de forma consecutiva. Partint d'aquesta descripció responeu a les següents preguntes:*

1. *En quina posició de memòria escriu aquest programa el resultat?*
2. *Què fa el programa?*
3. *Quina sentència de control de flux (en llenguatge d'alt nivell) implementa el programa?*
4. *Quina és la funció d'*`a1` *al programa? I la d'*`a0`*?*

*Resolució, primer apartat.* Una vegada es dona la condició de sortida, que `a7` sigui igual a 0, s'executa una instrucció final que emmagatzema el valor de `a2`, 10, a la direcció de memòria emmagatzemada per `a0+4`. ◁

*Resolució, segon apartat.* El programa fa una sèrie de sumes, restes (immediates o no), assignacions i adreçaments, alguns d'ells dins d'un bucle. El lector em permetrà la llicència de no esplaiar-me molt en aquesta pregunta, ja que entrarem en tota mena de detalls pel que fa l'estructura del programa més endavant. Potser, a trets generals, destacar:

1. la primera assignació, que guarda al registre no un valor, sinó una adreça de memòria, fet totalment transcendental al bucle, ja que ens permetrà accedir als valors declarats a `.data` sense crear un registre explícit;
2. l'existència d'un bucle i una condició de sortida en un llenguatge de baix nivell.
3. el `final` fa una última operació `store`, que fa `Mem[4+R[a0]] = R[a2]`, on `R[a0]` és una direcció de memòria continguda en tal registre.

◁

*Resolució, tercer apartat.* La sentència de control de flux implementada pel nostre programa és un condicional que ens serveix per controlar la iteració: com hem comentat, tenim la *keyword* `loop` que ens denota l'existència d'un bucle, i l'incremental és a dins de la mateixa estructura (en un llenguatge de baix nivell no es trobarà dins la capçalera del mateix `loop`). Per tant, el condicional és un `if` que està descrit en aquest llenguatge com `beqz a7,final`, de tal manera que salta a la posició de memòria `final` en cas que `a7` sigui igual a zero. ◁

*Resolució, quart apartat.* Com ja hem esmentat succintament, `a1` seria el nostre incremental, que comença inicialitzat a zero, dins del bucle: quan `a1` sigui 4, `a7` valdrà zero i sortirem de la iteració. Per parlar d'`a0` no ho podem fer d'una forma tan concreta: com veiem, és una direcció de memòria que s'actualitza iterativament per accedir successivament al següent espai de memòria, que augmenta de 4 en 4 perquè

les direccions s'escriuen d'aquesta manera. En particular, com hem vist, aquestes dades a les quals accedia corresponien justament a les declarades a `.data`. Potser, de manera formal, podem dir que `a0` és un accessor i actualitzador recursiu a la memòria. ◁

**Exercici 11.**

1. *Estabiu la relació entre el codi que hi ha a la finestra* source code *i el que apareix a la finestra* executable code.
2. *Observeu que inicialment* `PC=0h` *(el* `PC` *apunta a la següent instrucció executable del programa). Esbrineu com ho fa.*
3. *Executeu el programa, instrucció per instrucció, observant el resultat d'executar cada una de les instruccions:*
   - *Abans d'executar cada instrucció prediu les variacions que es produiran al banc de registres i a la memòria.*
   - *Comproveu que el resultar de l'execució del programa coincideix amb el que havíeu previst.*
   - *Per cada instruccions anoteu tots els canvis.*

*Resolució, primer apartat.* El *source code* és aquell que podem escriure en llenguatge de baix nivell, `Risc-V`, o d'alt nivell, en `C`, i és traduït per l'ensamblador a llenguatge màquina per ser executat a partir d'una sèrie d'instruccions en binari. Fixem-nos: totes tres són equivalents, però la única que és reconeguda per l'ordinador és la tercera. ◁

*Resolució, segon apartat.* Inicialment `PC=0h`, és a dir, apunta a la primera instrucció i anirà apuntant a la següent instrucció del programa. Per tal d'esbrinar com va apuntant a la següent iterativament, ens cal fixar-nos en l'estructura del processador `Risc-V` que estem utilitzant. En efecte, el `PC` està connectat a una `ALU` sumador, la qual suma la sortida del `PC` amb el terme 4. Així doncs, les instruccions, tal com el registres consecutius, es poden recórrer com a múltiples de 4. En definitiva, el que estem fent és incrementar en 4 la direcció de memòria apuntada per `PC` per carregar la següent instrucció des de la memòria de programa. ◁

*Resolució, tercer apartat.* La primera instrucció ens emmagatzema l'**adreça** d'`xx` al registre `a0`. Després, fem la resta d'`a1` amb ell mateix, per la qual cosa acabem posant-hi un zero. Acte seguit, emmagatzemem un zero al registre `a2`, ja que la suma d'elements nuls és l'element nul. Comencem el bucle:

1. Primer bucle:
   1. Fem la suma immediata d'`a1`, que conté un zero, i $-4$, de manera que obtenim $-4$ i l'emmagatzemem a `a7`.
   2. Com `a7` no és zero, continuem el bucle.

3. Carreguem a a3 el valor que es troba a memòria donat per la direcció emmagatzemada a a0, un 3.
4. Sumem a3 amb ell mateix, 6, i guardem en a2.
5. Desplacem la direcció de memòria emmagatzemada en a0 4 posicions. Si busquem tal direcció a memòria, ens surt un 5, justament el valor de memòria de yy.
6. A a1, que té un 0, li sumem 1. Ara, emmagatemarà un 1.

2. Segon bucle:
   1. Fem la suma immediata d'a1, que conté un 1, i $-4$, de manera que obtenim $-3$ i l'emmagatzemem a a7.
   2. Com a7 no és zero, continuem el bucle.
   3. Carreguem a a3 el valor que es troba a memòria donat per la direcció emmagatzemada a a0, un 5.
   4. Sumem a3 amb ell mateix, 10, i guardem en a2.
   5. Desplacem la direcció de memòria emmagatzemada en a0 4 posicions. Si busquem tal direcció a memòria, ens surt un 2, la direcció de memòria d'oo.
   6. A a1, que té un 1, li sumem 1. Ara, emmagatzemarà un 2.
3. Tercer bucle:
   1. Fem la suma immediata d'a1, que conté un 2, i $-4$, de manera que obtenim $-2$ i l'emmagatzemem a a7.
   2. Com a7 no és zero, continuem el bucle.
   3. Carreguem a a3 el valor que es troba a memòria donat per la direcció emmagatzemada a a0, el 2 que comentàvem abans.
   4. Sumem a3 amb ell mateix, 10, i guardem en a2.
   5. Desplacem la direcció de memòria emmagatzemada en a0 4 posicions. Si busquem tal direcció a memòria, ens surt un 8, la direcció de memòria d'zz.
   6. A a1, que té un 2, li sumem 1. Ara, emmagatzemarà un 3.
4. Quart bucle:
   1. Fem la suma immediata d'a1, que conté un 3, i $-4$, de manera que obtenim $-2$ i l'emmagatzemem a a7.
   2. Com a7 no és zero, continuem el bucle.
   3. Carreguem a a3 el valor que es troba a memòria donat per la direcció emmagatzemada a a0, el 5 que comentàvem abans.
   4. Sumem a3 amb ell mateix, 10, i guardem en a2.
   5. Desplacem la direcció de memòria emmagatzemada en a0 4 posicions. Si busquem tal direc-

ció a memòria, `0x10000010`, no hauríem de trobar cap valor. Igualment, això no influirà en l'execució del programa, ja que podem consultar que per defecte se li assignarà un zero.

6. A `a1`, que té un 3, li sumem 1. Ara, emmagatzemarà un 4.

5. Cinquè bucle:
   1. Fem la suma immediata d'`a1`, que conté un 4, i $-4$, de manera que obtenim 0 i l'emmagatzemem a `a7`.
   2. Com `a7` és zero, sortim.

Si ens adonem, l'`a1` funciona com un comptador incremental, per la qual cosa el bucle s'aturarà a la quarta iteració (la cinquena és per sortir) i, en teoria, s'acaba el programa amb la posició de memòria `final`.

**Observació 12.** *Notem que hem declarat 4 valors a* `.data`*, així que està convenientment estructurat: recordem que en cas contrari* `a0` *s'aniria desplaçant per posicions de memòria que no hem assignat.*

Seguint el mateix procés per a totes les iteracions, hem trobat que el valor a l'acabar el bucle resulta ser d'`a2` és 16, i, després de passar-li la instrucció `final`, el valor de `M[a0+4] = M[0x10000014]` passa a ser `a2 = 16`. ◁

**Exercici 13.** *Executa el programa pas a pas. El programa que s'executarà és el desensamblat. La primera instrucció el que fa es incrementar el registre PC per carregar la instrucció des de la memòria de programa. Executa inicialment el programa amb el processador que hem fet servir des de l'inici: el RSCP. Tot seguit canvia de processador i posa el R5SP. El processador amb el que esteu simulant ara és un processador multicicle amb un pipeline de 5 estats. Això vol dir que, en un cicle de rellotge està executant 5 fases de 5 instruccions diferents. Analitza el comportament del programa:*

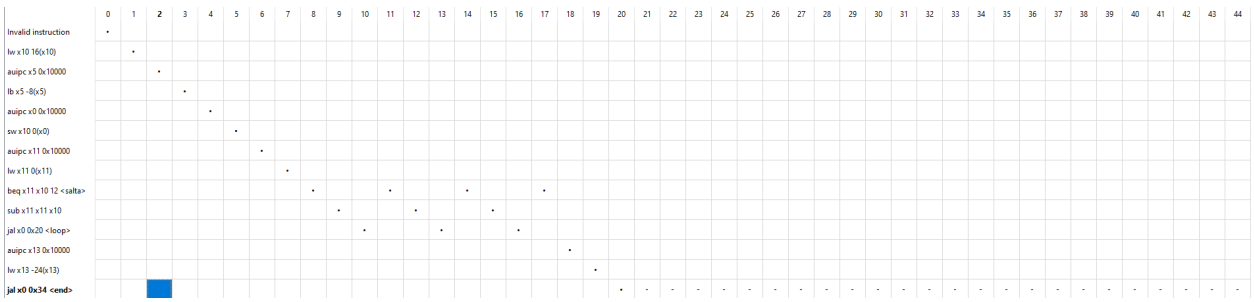
1. *Quan triguen a executar el programa? Per què?*
2. *Quin és el diagrama d'execució?*
3. *Quin és el màxim nombre d'instruccions que s'han executat simultàniament?*

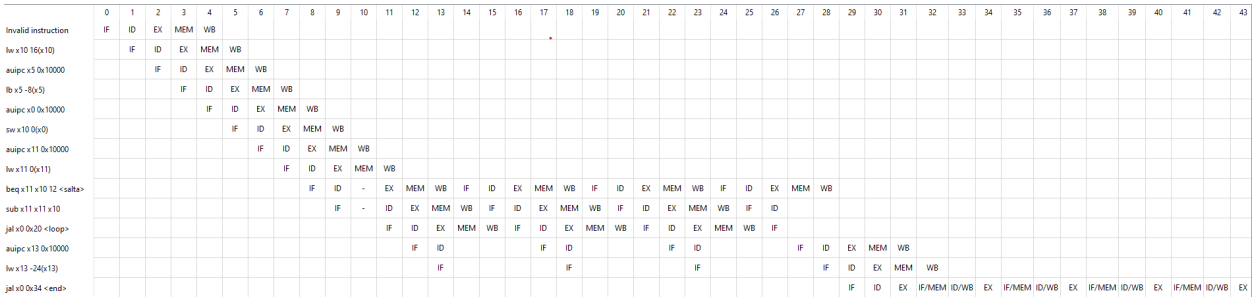
```
.data
    dataByte0: .byte 4
    databyte1: .byte 55
    dataByte2: .byte 235
    dataByte3: .byte 31
    data1: .word 1
    data2: .word 0
    data3: .word 4
.text
main:
    lw a0,data1
    lb t0,dataByte0
    sw a0,data2(zero)
    lw a1,data3
loop:
    beq a1,a0,salta
    sub a1,a1,a0
    j loop
salta:
    lw a3,data2
end:
    j end
```

Figura 2: Codi de l'exercici

*Resolució, apartat 1.* Si executem el programa amb qualsevol dels dos processadors ens trobem que mai no acaba. Això pot ser perquè entra en un bucle infinit. Fixem-nos, però, que no és cosa del `loop`: si el deixem córrer veiem que el valor d'`a1` i `a0` és el mateix, així que, efectivament, surt del bucle. Per tant, el problema està en l'`end`, de manera que salta contínuament a aquesta etiqueta des d'ella mateixa, esdevenint així un bucle infinit. ◁

*Resolució, apartat 2.* En el cas del *Single Cycle*, el procés té com a màxim una instrucció simultània, clarament. En el de cinc cicles, en tindrem com a màxim cinc. A les etapes finals, però, veiem que s'executa solament una en els dos casos a causa del bucle infinit.






◁

# 3 Conclusions

Hem après a:

1. entendre el funcionament de la CPU i la seva connexió amb la memòria principal;
2. familiaritzar-se amb el simulador Ripes en les seves dues versions amb què treballarem: RSCP i R5SP;
3. entendre què fan les instruccions;
4. comprendre l'analogia entre llenguatge màquina i el llenguatge assemblador;
5. analitzar i veure en la pràctica conceptes com el 'Pipeline' o execució pseudoparal·lela de les instruccions.

En altres paraules, hem fet les nostres primeres passes en el món del `Risc-V`. Gens malament per la nostra primera pràctica!