Introducció als Ordinadors

Pràctica 3

Mario Vilar

12 d'abril 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.

Com s'ha explicat a teoria, dividim el conjunt de registres fonamentalment en dos tipus:

1. Registres de propòsit general
2. Registres específics

Els registres de propòsit general es troben al banc de registres. Són 32 registres `[x0:x31]` de propòsit general, el registre `x0` sempre val 0, per tots dos casos, mentre que la resta, el seu contingut és variable. Així una bona forma d'inicialitzar un registre (posar a 0 el seu contingut) serà per exemple:

```
ADD a0, zero, zero # Suma el contingut de x0 + el contingut de
                   # x0 i es desa en a0
```

Un registre de propòsit específic és per exemple el Program Counter (PC), on tenim l'adreça de la següent instrucció a executar. RISC-V té altres registres de propòsit específic (per gestionar el temps, per fer entrada/sortida amb el hardware, etc.) però el simulador `RIPES` no ho implementa

1.1 Objectius

Aquesta pràctica té com a principal objectiu la utilització de sentències de control de flux, generació de bucles i salts condicionals i incondicionals. Per altra banda, un altre objectiu és solidificar els coneixements adquirits per part de l'alumne en les pràctiques anteriors.

2 Cos del treball

2.1 Pràctica

Exercici 1.

1. *Quines instruccions de salt condicional hem fet servir? En quin cas salten?*
2. *Què fan les instruccions de salt condicional quan la condició no es compleix?*
3. *Per què hem utilitzat les instruccions de salt incondicional?*

Observació 2. *Fixa't que la condició per al codi d'alt nivell ($a \geq b$) és la contrària que fem servir en el codi en llenguatge assemblador. Això passa perquè en el codi d'alt nivell la condició indica que s'executa la branca certa ($a \geq b$). En canvi, en el codi màquina de l'exemple anterior indica que saltem a la branca falsa ($a < b$). També és important recordar que després d'executar la branca certa cal saltar-se el bloc de la branca falsa.*

Demostració. La solució a què s'ha arribat és la següent:

```
.data
a: .word 5
b: .word -6
resultat: .word 0
.text
la a0, a
lw a1, 0(a0)
lw a2, 4(a0)
bge a1,a2,cert
j fals
cert:
sub a3,a1,a2
j end
fals:
sub a3,a2,a1
j end
end:
sw a3, 8(a0)
nop
```

La instrucció de salt condicional que hem fet servir és `bge`, ja que volem trobar si $a \geq b$ i saltar a `cert` si es produeix tal casuística. Quan la condició no es compleixi, el PC apuntarà a la següent instrucció del programa, de manera que necessitem una condició de salt incondicional `j etiqueta` per a conduir el codi (augmentar el PC) cap a la branca `fals`. ◁

Exercici 3.

1. *Quin tipus d'estructura de control de flux indica normalment un salt cap enrere?*
2. *Omple la següent taula executant el programa.*

Observació 4. *De nou, la condició per al codi d'alt nivell (*`comptador > 0`*) és la contrària que fem servir en el codi en llenguatge assemblador. En aquest cas, el codi d'alt nivell avalua si es manté en el bucle (*`comptador > 0`*), mentre que el codi en llenguatge assemblador salta si surt del bucle (*`comptador = 0`*). A més, al final del bucle hem de tornar al principi.*

Demostració. La instrucció que indica un salt cap enrere és qualsevol instrucció de salt, tant condicional com incondicional, que es refereixi a una etiqueta que estigui situada anteriorment en l'ordre normal del codi. El codi utilitzat ha estat el següent:

```
.data
comptador: .word 10
resultat: .word 0
.text
la a0, comptador
lw a0, 0(a0)
addi a1, zero, 0
addi a2, zero, 1
loop:
beqz a0,end
add a3,a1,a2
# guardem a2 en a1 i a3 en a2
addi a1,a2,0
addi a2,a3,0
# actualitzem el comptador
addi a0,a0,-1
j loop
end:
la a0, resultat
sw a1, 0(a0)
nop
```

En aquest cas, el codi calcularia F_{10}.

I la taula resultant ha sigut aquesta:

Valor inicial	Valor resultant	Cicles	Instruccions
F_0	0	10	10
F_1	1	16	16
F_2	1	22	22
F_3	2	28	28
F_4	3	34	34
F_5	5	40	40
F_6	8	46	46
F_{25}	75025	160	160
F_{46}	1836311903	286	286
F_{47}	-1323752223	292	292

Valor inicial	Cicles	Instruccions
F_0	16	10
F_1	24	16
F_2	32	22
F_3	40	28
F_4	48	34
F_5	56	40
F_6	64	46
F_{25}	216	160
F_{46}	384	286
F_{47}	392	292

En el primer quadre, notem que els cicles i les instruccions són els mateixos perquè estem treballant per defecte sobre el processador *single-cycle* (a l'esquerra). Al segon, ho hem fet amb el *pipe-line* per fer palesa la diferència existent. El valor de F_{47} és negatiu a causa de l'*overflow* que es produeix en les sumes successives. Evidentment, el veritable F_{47} és més gran que 0 i el valor resultant és incorrecte. ◁

Exercici 5. *Descriu el programa que has implementat. Quins salts has usat? Quins són condicionals i quins són incondicionals, i per què?*

Demostració. El programa implementat és el següent:

```
.data
a: .word 252
b: .word 105
resultat: .word 0
.text
la a0,a
lw a1,0(a0)
lw a2,4(a0)
loop:
beq a1,a2,end
bgt a1,a2,cert
j fals
cert:
sub a1,a1,a2
j loop
fals:
sub a2,a2,a1
j loop
end:
sw a1,8(a0)
nop
```

Primerament, carreguem el valor de memòria d'`a` i `b` als dos registres corresponents. Després, entrem al `loop`, del qual no sortirem fins que `a=b`. En altres paraules, anem aplicant l'algorisme euclidià per al màxim comú múltiple iterativament: si `a1` és més gran que `a2`, fem `a1-a2`. Anàlogament, si `a2` és més gran que `a1`, fem `a2-a1`. Quan arribem a `a2=a1` (la condició que esmentàvem abans), fem un salt fins a `end` i guardem en memòria el valor final, `a1`. Els salts usats són els incondicionals `j loop`, destinats a mantenir-nos dins del bucle (la següent instrucció apuntarà a `loop` un altre cop), el `j fals` i el `j cert`, i el condicional `beqz a0,end`, ja que, en funció d'un valor, és a dir, quan `a0` sigui més petit que zero saltarem a l'etiqueta `end`. ◁

2.2 Teorico-pràctica

Exercici 6. *Executa un codi similar al presentat en la taula anterior. Verifica l'espai que ocupa en memòria el programa. Executa el programa en el simulador i compara el temps d'execució per al processador* single stage *i el* pipeline *de 5 etapes. Considereu que el* clock *per el primer processador i el del* pipeline *el podem conèixer a partir de la següent diapositiva.*

Demostració. Com en `RISC-V` cal operar amb registres (no podem fer-ho directament des de memòria), ens basarem en l'últim codi de la taula: *instruccions amb load i store i banc de registres*. El codi que hem usat a tall d'exemple és el següent:

```
.data
A: .word 2
B: .word 3
.text
la a0,A
lw a1,A
lw a2,B
add a3,a1,a2
sub a4,a1,a2
mul a5,a3,a4
sw a5,20(a0)
```

En binari cada instrucció ocupa 32 bits i tenim 9 instruccions, ja que ignorem la segona instrucció generada, `addi x10 x10 0`, i en total resulta:

$$\frac{288}{8} = 36 \text{ Bytes.} \tag{2.1}$$

Single-Cycle

$$\begin{aligned} T_c &= t_{pcq} + 2t_{mem} + t_{RFread} + t_{ALU} \\ &+t_{mux} + t_{RFsetup} = 30 + 2 \cdot 250 + 150 + 200 \\ &+25 + 20 = 925 \text{ ps/cycle} \\ T_1 &= 10 \text{ cycles} \cdot \frac{925 \text{ ps}}{1 \text{ cycle}} = 9250 \text{ ps.} \end{aligned}$$

5-Stage Cycle

$$\begin{aligned} T_c &= 550 \text{ ps,} \\ T_2 &= 15 \text{ cycles} \cdot T_c = 8250 \text{ ps.} \end{aligned}$$

Totes les etapes del *pipeline* prenen un sol cicle de rellotge, de manera que el cicle de rellotge ha de ser prou llarg per acomodar l'operació més lenta. El processador *pipeline* és substancialment més ràpid que els altres. Tanmateix, el seu avantatge respecte al processador d'un sol cicle no s'acosta ni a l'acceleració de cinc vegades que un espera obtenir d'un *pipeline* de cinc etapes. L'etapa de descodificació és substancialment més lenta que les altres, perquè el fitxer de registre d'escriptura, lectura i comparació de branques ha de succeir tot en mig cicle. El disseny d'un sol cicle ha de prendre el cicle de rellotge del pitjor dels casos. El processador *pipeline* té requeriments de *hardware* similars a un *single-cycle*, però augmenta en el nombre de registres, multiplexors i portes lògiques per a resoldre *hazards*.

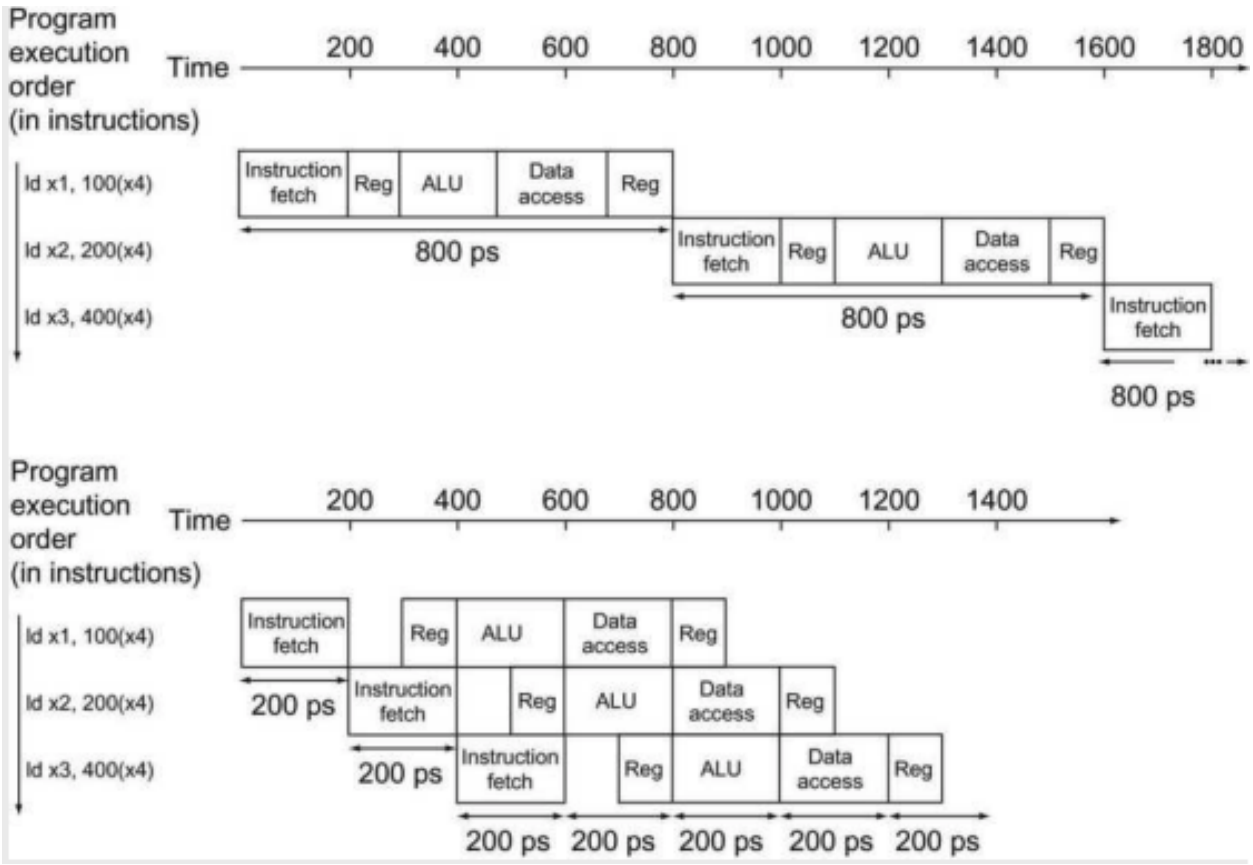

Figura 1: Comparació d'un cas general entre un *single-cycle* i un *pipeline*.

En qualsevol cas, per la segona equació, la de 5 cicles, hem usat que:

$$T_c = \max \left\{ \begin{array}{c} t_{pcq} + t_{mem} + t_{setup} \\ 2(t_{RFread} + t_{mux} + t_{eq} + t_{and} + T_{mux} + t_{setup}) \\ t_{pcq} + 2t_{mux} + t_{ALU} + t_{setup} \\ t_{pcq} + t_{memwrite} + t_{setup} \\ 2(t_{pcq} + t_{mux} + t_{RFwrite}) \end{array} \right\} = \max \left\{ \begin{array}{c} 30 + 250 + 20, \\ 2(150 + 25 + 40 + 15 + 25 + 20), \\ 30 + 25 \cdot 2 + 200 + 20, \\ 30 + 220 + 20, \\ 2(30 + 25 + 100) \end{array} \right\}.$$

◁

Exercici 7. *Feu un programa senzill en ASM que incorpori aquest tipus d'adreçament (mode d'adreçament relatiu). Identifiqueu-lo i expliqueu-lo.*

```
.data
A: .word 0
.text
la a0, A
lw a1, 0(a0) # efectuem a1 <= M[@A]
addi a1, a1, 4
sw a1, 4(a0) # M[(@A)+4] <= a1
nop
```

La primera línia del cos del programa instrueix `la`, de manera que `a0` emmagatzemarà la posició de memòria de la dada `A`. La segona i, en particular, el primer comentari que la succeeix, ens indica que estem carregant un contingut de memòria mitjançant una adreça en un registre i poder-lo fer servir en determinades parts d'un programa. El segon i últim comentari fa la operació contrària: guardem el contingut del registre `a1`, el qual hem augmentat en 4 unitats, en una posició de memòria fixada. En aquest cas és, justament, la posició consecutiva del valor `A`; a la vegada, tenim en consideració l'`offset` que hem declarat.

3 Conclusions

Hem assolit els objectius principals d'aquesta pràctica: la utilització de sentències de control de flux, generació de bucles i salts condicionals i incondicionals, així com solidificar els coneixements adquirits en les pràctiques anteriors. En definitiva, les branques i els salts no tenen cap mena de dificultat si abstraiem el concepte a un llenguatge d'alt nivell i el pensem com un condicional habitual.

D'altra banda, i ja per últim, l'exercici de Fibonacci m'ha permès repassar un tema que tenia força oblidat, que era el de l'`overflow` en qualsevol operació aritmètica i la posterior gestió que se'n fa.