Introducció als Ordinadors

Pràctica 4

Mario Vilar

28 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.

El número de cicles dividit per la freqüència de funcionament del processador ens defineix el temps d'execució de la instrucció. Les diferents accions que es realitzen en cada cicle forma el que es coneixen

com microinstruccions.

En ciència de computadors, el pipeline RISC es una solució de microarquitectura que té com a propòsit la execució d'una instrucció per cicle de rellotge. Aquest pipeline executa una instrucció en cinc etapes: Instruction Fetch, Instruction Decode, Execute, Memory Access i Writeback.

Alguns dels processadors que comparteixen aquesta microarquitectura són: MIPS (Microprocessor without Interlocked Pipelined Stages), SPARC (Scalable Processor Architecture), Motorola 88000 (o m88k) i, es clar, el RISC-V (Reduced Instruction Set Computer).

1.1 Objectius

L'objectiu de la pràctica és visualitzar els diferents passos que ha de fer un microprocessador per tal d'executar una instrucció. 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 Exercici guiat

Exercici 1. *Haurem d'executar els següents codis, cicle a cicle (microinstrucció a microinstrucció) utilitzant com a micro el 5-Stage Processor al simulador Ripes:*

Codi 1

```
.data
resultat: .word 0
.text
main:
add a3, zero, zero
add a7, zero, zero
addi a2, zero, 4
add a3, a2, a3
addi a7, a7, -1
la a0, resultat
sw a3, 0(a0)
```

Codi 2

```
.data
resultat: .word 0
.text
main:
add a3, zero, zero
add a7, zero, zero
addi a2, zero, 4
add a3, a2, a3
addi a7, a7, -1
blt a7, zero, salta
salta:
la a0, resultat
sw a3, 0(a0)
```

1. *Abans d'executar els codis tracta d'esbrinar la seva funcionalitat. Faran el mateix? Creus què trigaran el mateix nombre de cicles en executar-se?*
2. *Ves a la finestra del Ripes dedicada al processador (Processor). Busca entre les opcions del Simulator control, la Pipeline table.*

3. *Executa els dos codis cicle a cicle fixant-te com les instruccions van passant per les diferents etapes del pipeline. Compta els nombre de cicles que es necessiten per executar cadascun dels codis i compara les Pipeline tables.*
4. *Perquè les etapes de la instrucció* `auipc x10 0x 10000` *al pipeline són* `IF ID IF ID EX MEM WB` *al Codi 2?*
5. *Com afectaria al nombre total de cicles d'execució el següent canvi en el codi?*

Codi 1

```
.data
resultat: .word 0
.text
main:
add a3, zero, zero
add a7, zero, zero
addi a2, zero, 4
add a3, a2, a3
addi a7, a7, -1
la a0, resultat
sw a3, 0(a0)
```

Codi 3

```
.data
resultat: .word 0
.text
main:
add a3, zero, zero
add a7, zero, zero
addi a2, zero, 4
add a3, a2, a3
addi a7, a7, -1
bgt a7, zero, salta
salta:
la a0, resultat
sw a3, 0(a0)
```

Demostració. La funcionalitat dels dos codis serà similar, fixem-nos que les operacions aritmètiques són idèntiques i simplement difereixen en la part final del programa, amb un *branch*, que provoca un salt (que sempre es complirà) a les mateixes instruccions finals per ambdós programes. En definitiva, la funcionalitat hauria de ser la mateixa. Evidentment, el fet que hi hagi un condicional pel mig implicarà que el nombre de cicles no serà el mateix en els dos codis.

Les *pipeline tables* han sigut les següents:

	0	1	2	3	4	5	6	7	8	9	10	11	12
add x13 x0 x0	IF	ID	EX	MEM	WB								
add x17 x0 x0		IF	ID	EX	MEM	WB							
addi x12 x0 4			IF	ID	EX	MEM	WB						
add x13 x12 x13				IF	ID	EX	MEM	WB					
addi x17 x17 -1					IF	ID	EX	MEM	WB				
auipc x10 0x10000						IF	ID	EX	MEM	WB			
addi x10 x10 -20							IF	ID	EX	MEM	WB		
sw x13 0(x10)								IF	ID	EX	MEM	WB	

Figura 1: Pipeline del codi 1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
add x13 x0 x0	IF	ID	EX	MEM	WB											
add x17 x0 x0		IF	ID	EX	MEM	WB										
addi x12 x0 4			IF	ID	EX	MEM	WB									
add x13 x12 x13				IF	ID	EX	MEM	WB								
addi x17 x17 -1					IF	ID	EX	MEM	WB							
blt x17 x0 4 <salta>						IF	ID	EX	MEM	WB						
auipc x10 0x10000							IF	ID	IF	ID	EX	MEM	WB			
addi x10 x10 -24								IF		IF	ID	EX	MEM	WB		
sw x13 0(x10)											IF	ID	EX	MEM	WB	

Figura 2: Pipeline del 2

El nombre de cicles que es necessiten pel primer codi és de 12; en canvi, per al segon codi en fan falta més, sense sorpreses (en particular, 15).

Les etapes de `auipc x10 0x 10000` la sortida del `branch` es determina al final de l'etapa d'execució. Això és, quan sabem cap a on fer-lo, noves instruccions es troben al *pipeline*: com a resposta, es produeix un `nop (flush)` en la segona i la tercera etapa del *pipeline* ja que no s'iterarà quan es va a executar la instrucció. En més detall, quan hi ha una instrucció de salt condicional, el processador intenta guanyar temps fent una predicció. La predicció és que bàsicament es voldrà iterar i per tant la condició no s'acomplirà, per tant, continua executant les següents instruccions. Ara bé, si la condició de salt si es compleix, el processador elimina les microinstruccions associades a la predicció (`flush`) i va a la posició indicada per la instrucció de salt. Això fa que el `PC` apunti un altre cop a `auipc x10 0x 10000`, és a dir, es reinicïi un altre cop des d'`IF`, i es necessitin dos cicles de `flush`. Tot això, resulta en aquesta conjugació d'etapes.

Cal notar la similitud entre el segon codi (*codi 2*) i el tercer que se'ns acaba de presentar. De fet, són el mateix, excepte un canvi en la condició de salt. En efecte, aquest canvi provoca que el conjunt d'instruccions de l'etiqueta `salta` no es produeixi i, en conseqüència, acabi el programa sense l'execució corresponent. En altres paraules, estem dient:

$$N_{C_3} = N_{C_1} + 3 - 2 = 13, \tag{2.1}$$

on sumem 3 a causa del `bgt` del codi 3 i restem 2 perquè no s'executen les instruccions de l'etiqueta `salta` (ja hem comentat que el programa no hi arriba). ◁

2.2 Pràctica

Exercici 2. *Què és un stall?*

Demostració. L'*stall* d'una etapa es realitza desactivant el registre del *pipeline*, perquè els continguts no canviïn. Quan una etapa està en *stall*, totes les etapes anteriors també s'han de posar en *stall*, de manera que no hi ha instruccions posteriors que es perdin. El registre del *pipeline* directament posterior a l'etapa d'*stall* ha de ser esborrat per evitar que la informació incompleta (o falsa, com es prefereixi) es propagui cap endavant. L'*stall* degrada el rendiment del processador i, per tant, del programa, de manera que només s'ha d'utilitzar quan sigui estrictament necessari. ◁

Exercici 3.

1. *Quin es l'estat de cadascuna de les cinc etapes del pipeline al cicle 6? I al 8?*
2. *Quins senyals de control s'activen en el cicle 4 a la segona etapa del pipeline? A quines instruccions del codi corresponen?*
3. *Quins són els valors a les sortides dels multiplexors assenyalats a la figura al cicle 9?*
4. *Què està calculant la ALU al cicle 9?*
5. *Llegeix amb cura la part del codi amb que s'implementa el loop. Tenint en compte el que has vist a la qüestió 3), justifica perquè el pipeline al cicle 10 presenta aquest estat.*

6. *Quan NO es produeix el salt, quantes etapes de la instrucció* `auipc x10 0x 10000` *s'executen al pipeline?*
7. *Quan s'està executant la instrucció de salt (etapa EX), però el salt NO es produeix, a quina posició apunta el program counter (PC)?*
8. *Quan es produeix el salt, quantes etapes de la instrucció* `auipc x10 0x 10000` *s'executen al pipeline?*
9. *Quan s'està executant la instrucció de salt (etapa EX), i el salt es produeix, a quina posició apunta el program counter (PC)?*

Codi 4

```
.data
valorDada: .word 2
guardaResultat: .word 0
.text
main:
la a0, valorDada
lw a7, 0(a0)
addi a2, zero, 9
add a3, zero, zero
loop:
add a3, a2, a3
addi a7, a7, -1
bgt a7, zero, loop
la a0, guardaResultat
sw a3, 0(a0)
```

Codi 5

```
.data
valorDada: .word 2
guardaResultat: .word 0
.text
main:
la a0, valorDada
lw a7, 0(a0)
addi a2, zero, 9
add a3, zero, zero
loop:
add a3, a2, a3
addi a7, a7, -1
bgt a7, zero, loop
la a0, guardaResultat
sw a3, 0(a0)
lw a0, guardaResultat
addi a0, a0, 4
```

Resolució. Dividirem la resposta en els dos codis que se'ns demana:

1. En el codi 3, al cicle 6 i 8 tenim els següents estats:

Stage	Instruction
	`add x13 x0 x0`
	`add x17 x0 x0`
WB	`addi x12 x0 4`
MEM	`add x13 x12 x13`
EX	`addi x17 x17 -1`
ID	`blt x0 x17 4 <salta>`
IF	`auipc x10 0x10000`
	`addi x10 x10 -24`
	`sw x13 0(x10)`

Figura 3: Cicle 6, codi 3

Stage	Instruction
	`add x13 x0 x0`
	`add x17 x0 x0`
	`addi x12 x0 4`
	`add x13 x12 x13`
WB	`addi x17 x17 -1`
MEM	`blt x0 x17 4 <salta>`
EX	`auipc x10 0x10000`
ID	`addi x10 x10 -24`
IF	`sw x13 0(x10)`

Figura 4: Cicle 8, codi 3

En el quart cicle s'activa el `write-enable` del banc de registres, i s'apaga en el novè; això és, s'activa quan arriba la primera instrucció a la fase de `writeback` (`WB`) i es desactiva quan la operació d'emmagatzematge `sw x13 0(x10)` passa per la fase del *pipeline* en qüestió, la segona.
Seguint al novè cicle, la sortida dels multiplexors que després entren a l'ALU és la següent:

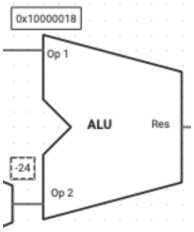

Figura 5: Sortida dels multiplexors

evidentment, està efectuant `addi x10 x10 -24` que, juntament amb la instrucció anterior, suposen el codi màquina que conforma la instrucció `la a0, resultat`. L'última qüestió no es pot contestar per al codi 3, ja que no té cap bucle. Respondrem la resta de preguntes a partir del codi 5.

2. Procedim per al codi 5. al cicle 6 i 8 tenim els següents estats:

Stage	Instruction
	`auipc x10 0x10000`
	`addi x10 x10 0`
WB	`lw x17 0(x10)`
MEM	`addi x12 x0 9`
EX	`add x13 x0 x0`
ID	`add x13 x12 x13`
IF	`addi x17 x17 -1`
	`blt x0 x17 -8 <loop>`
	`auipc x10 0x10000`
	`addi x10 x10 -28`
	`sw x13 0(x10)`
	`auipc x10 0x10000`
	`lw x10 -40(x10)`
	`addi x10 x10 4`

Figura 6: Cicle 6, codi 5

Stage	Instruction
	`auipc x10 0x10000`
	`addi x10 x10 0`
	`lw x17 0(x10)`
	`addi x12 x0 9`
WB	`add x13 x0 x0`
MEM	`add x13 x12 x13`
EX	`addi x17 x17 -1`
ID	`blt x0 x17 -8 <loop>`
IF	`auipc x10 0x10000`
	`addi x10 x10 -28`
	`sw x13 0(x10)`
	`auipc x10 0x10000`
	`lw x10 -40(x10)`
	`addi x10 x10 4`

Figura 7: Cicle 8, codi 5

En el quart cicle s'activa el `write-enable` del banc de registres; això és, s'activa quan s'escriu els resultats de la primera operació del programa en un registre, i serà a l'onzè cicle quan aquesta senyal quedi desactivada. Pel que fa al novè cicle, la sortida dels multiplexors que després entren a l'ALU és la següent:

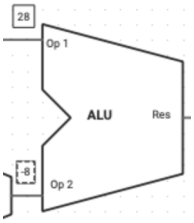

Figura 8: Sortida dels multiplexors.

La operació que està efectuant no és tan clara com en el cas anterior. Veiem que la instrucció en aquesta etapa d'exe és `blt x0 x17 -8 <loop>`. Com veiem, el resultat és 20 i, com la `ALU` és asíncrona, això passa en un mateix cicle de rellotge al multiplexor anterior al `PC`. D'aquesta manera, distingim que la operació efectuada ens diu la posició de la següent instrucció del programa, `0x00000014`, justament `add x13 x12 x13`. Al següent cicle escriurem aquest valor al registre `PC` i descartarem amb un `flush` (activant el `clear` de `IFID` i `IDEX`) el contingut d'aquestes dues etapes, ja que pels valors que operem en el cos de la primera iteració del bucle no sortim d'ell i, consegüentment, no es poden executar instruccions posteriors al bucle. De fet, al desè cicle es dona la següent casuística:

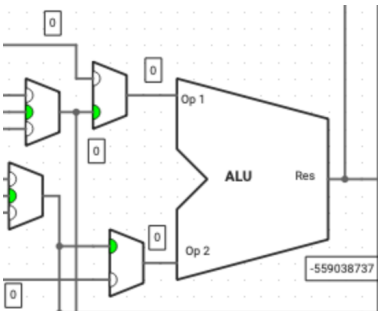

Figura 9: Sortida dels multiplexors al desè cicle.

Aquesta fenomenologia que hem comentat ve directament donada a causa del `flush`. És fàcil veure que amb els multiplexors a zero el resultat de la ALU no aportarà cap sortida significativa en aquest cicle.

Pel que fa a les últimes preguntes, les contestarem executant el codi en qüestió. En el codi 5 la instrucció `auipc x10 0x10000` apareix tres cops. Si no hi ha bucle, cada instrucció passa per les cinc etapes una sola vegada i:

$$N_{ins} = 3 \cdot 5 = 15 \text{ etapes.} \tag{2.2}$$

Si mantinguéssim el codi 5 sense modificar-lo, de manera que fes una iteració pel bucle, s'executarien $15 + 2 = 17$ etapes de la instrucció: a les 15 anteriors li hem d'afegir les dues que fa a dins del bucle abans del `flush`. Aquí, doncs, hem distingit les etapes executades de la instrucció en qüestió. Ara,

una anotació important que hauríem d'haver fet al principi: el salt no es produeix quan $a_7 \leq 0$, això és, a la segona iteració. Per tant, responent a l'apartat 7, tenim el següent:

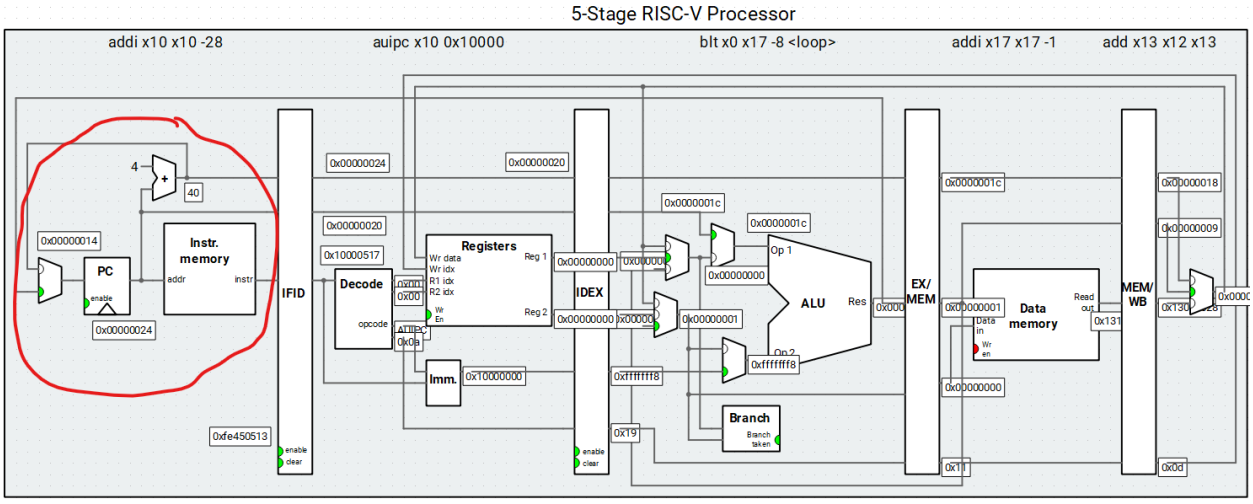

Figura 10: Valor del *program counter* quan no es produeix el salt.

Efectivament, com que no es torna a iterar sobre el *loop*, el multiplexor emmagatzema l'adreça , que passarà al registre del program counter a la següent etapa (en aquest moment emmagatzema `0x00000024`, `addi x10 x10 -28`). A continuació, ja apuntarà a `0x00000028`, `sw x13 0(x10)` donat que hem sortit de la iteració. Anàlogament, quan es produeix el salt, executem el programa i en mirem el valor.

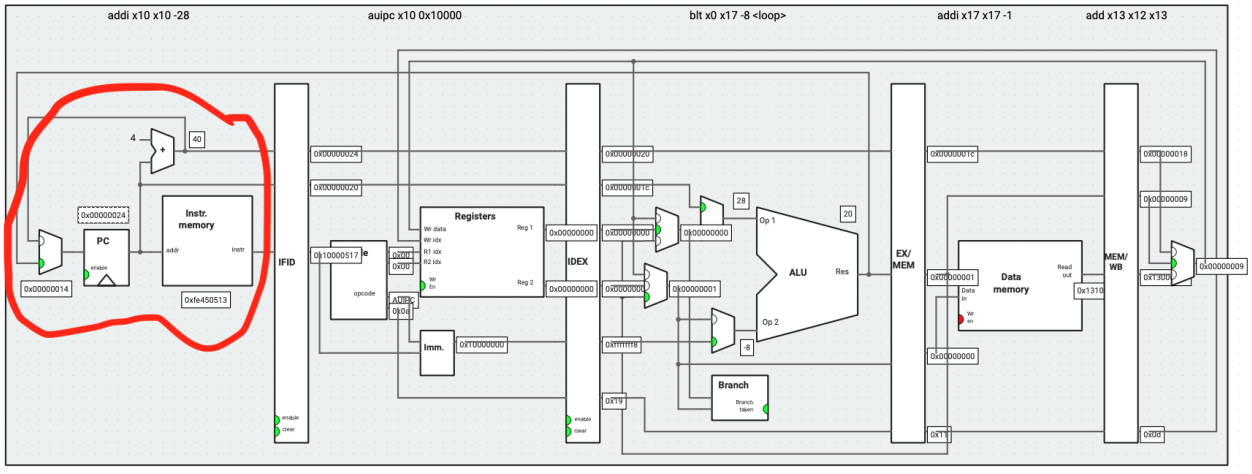

Figura 11: Valor del *program counter* quan es produeix, s'executa, el salt.

Quan s'executa la condició de salt veiem que no se surt del bucle i es torna a l'instrucció inicial d'aquest, de manera que el program counter apuntarà al resultat donat pel multiplexor que, al seu torn, agafarà en aquest cas la seva sortida de la `ALU`. Així, el `PC` emmagatzemarà en registre la posició corresponent a `addi x10 x10 -28` en el novè cicle, quan `blt` es troba a l'execu-

ció (`EX`). Un cicle de rellotge permetrà l'escriptura al registre `PC`, de manera que passarà a apuntar `add x13 x12 x13` (valor `0x00000014`).

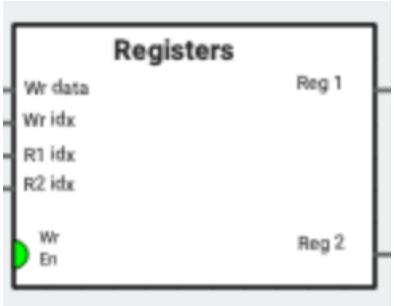

Figura 12: Captura de pantalla, quan s'activa el WrEn.

◁

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 visualitzar els diferents passos que ha de fer un microprocessador per tal d'executar una instrucció i solidificar els coneixements adquirits en les pràctiques anteriors.

D'altra banda, i ja per últim, m'ha semblat que aquesta pràctica ha donat un salt de dificultat massa alt respecte la resta. Hem just començat a entendre el funcionament intern del processador de cinc etapes, i ho hem hagut de fer en una sola sessió. L'estructura de *Ripes* és prou confusa com per dedicar-li tan poc temps. És cert que vam treballar aquest tipus de circuits ja de cara al parcial: és solament un comentari a tall de reflexió.