Introducció als Ordinadors

Pràctica 2

Mario Vilar (MV)

6 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

L'objectiu d'aquesta pràctica és familiaritzar-nos amb el funcionament dels registres que hi ha a la CPU de la mà de l'entorn `Ripes` i el format de les instruccions més rutinàries, les seves semblances i diferències i com això afecta a l'escriptura i execució del nostre codi. En aquest sentit, perfeccionant el nostre coneixement sobre el paper de la memòria i la seva interacció amb la resta de processos i components.

Continuar el nostre aprenentatge en el llenguatge d'assemblador és una de les metes que persegueix aquesta pràctica, acabant de polir alguns dels nostres avenços en les entregues anteriors.

2 Cos del treball

2.1 Exercicis pràctica

Exercici 1. *Determina la funció del codi que tenim a continuació.*

```
.data
Dades: .word 9, 3, 4, 1 # vector de 4 enters
Resultat: .word 0, 0 # vector de 2 enters
.text # directiva de inici de programa
main:
la a0, Dades # a0 actuara com a punter
lw a1, 0(a0) # 0(a0) = Dada[0]
lw a2, 4(a0) # 4(a0) = Dada[1]
lw a3, 8(a0) # 8(a0) = Dada[2]
lw a4, 12(a0) # 12(a0) = Dada[3]
loop:
add a5,a1,a2
add a6,a4,a2
addi a3, a3,-1
bgez a3, loop
sw a5, 16(a0) # 16(a0) = Resultat[0]
sw a6, 20(a0) # 20(a0) = Resultat[1]
end: nop #final de programa
```

I hem de respondre una sèrie de preguntes:

1. *Un cop acabat el programa quant val el contingut de* `a5`*?*
2. *Un cop acabat el programa quant val el contingut de* `a6`*?*
3. *Un cop acabat el programa quant val el contingut de* `a3`*?*
4. *Quines instruccions reals s'utilitzen per a les pseudoinstruccions* `nop` *i* `bgez`*? (mireu el codi màquina que es genera i consulteu la taula d'instruccions, utilitzant els camps* `opcode` *i* `func3/7` *si cal).*
5. *En quina posició de memòria es troben els valors de* `resultat`*? En quina posició de memòria es guarda el contingut del registre* `a6`*?*

6. *Quan executem* `bgez`*, quin registre es veu afectat?*

Resolució. Aquest programa assigna quatre valors a quatre diferents registres (notem que cadascun dels `a_i` contindrà l'$(i-1)$-èsim element de l'`array` per punters) i fa una sèrie d'operacions entre ells. En particular, `a3` funciona com a iterador, ja que la condició de sortida s'aplica: quan `a3` sigui més petit estrictament que zero. Al seu torn, `a5` i `a6` guarden successivament la mateixa suma al vector de resultats, així que en un nombre n qualsevol d'iteracions, tenim:

1. `Resultat[0] = (Dada[0]+Dada[1]) = 12`
2. `Resultat[1] = (Dada[1]+Dada[3]) = 4`

Això és, perquè els valors finals d'`a5`, `a6` seran 12 i -4, respectivament. A més, `a3` valdrà -1 quan surti del `loop` perquè la condició de sortida comporta un nombre inferior a 0. Pel que fa a les instruccions que ens pregunten:

1. `nop` genera el següent codi màquina: `addi x0, x0, 0`,
2. `bgez` genera el següent codi màquina: `bge rs, x0, offset`.

Els valors de resultats es guarden amb un `offset` de 16 i de 20 respecte l'adreça de memòria d'`a0`, així que seran `0x10000010` i `0x10000014`, respectivament. En efecte, podem consultar `Memory` al principi del programa assignant un valor determinat a aquestes dues quantitats per assegurar-nos-en. Potser la pregunta més interessant és on es guarden `a5` i `a6`: justament a les posicions `0x10000010` i `0x10000014`, les mateixes que el vector `resultat`.

Quan executem `bgez`, el registre que es veu modificat és el que control·la les instruccions a executar assenyalant-ne la següent (és a dir, el `PC`), ja que la condició `beqz` determina un canvi en l'ordre d'aquestes.

◁

Exercici 2. *Feu un programa similar als de l'exercici anterior. Inicialitzeu primer el contingut de* `A0`*. Carregueu al* `A1` *el valor* `0000110000001011b`*. Carregueu al registre* `A2` *el valor* `0000000000010001b`*. Feu l'operació* `A0<= A0+A1` *i decrementeu el valor de* `A2`*. Feu això en un bucle fins que el valor contingut en* `A2` *sigui 0.*

1. *Quina operació matemàtica està realitzant aquest codi?*
2. *Quant val el contingut d'*`A0`*,* `A1` *i* `A2` *quan acaba el programa?*
3. *Quantes iteracions del bucle executa el codi?*

Resolució. El programa resultant serà aquest:

```
.data
xx:  .word  0
yy:  .word  0b0001100000001011  # 3083
zz:  .word  0b0000000000010001  # 17
.text
```

```
main:
lw a0, xx
lw a1, yy
lw a2, zz
loop:
beqz a2,end
add a0, a0, a1
addi a2,a2,-1
j loop
end: nop
```

A l'executar-lo, veiem que el programa converteix els nombres a decimal i després procedeix a fer les operacions pertinents. Aleshores, deixant els cops de rellotge automatitzats a una freqüència determinada, s'acaba el bucle amb a2=0, a1 emmagatzema el valor inicial 3083 i a0 s'ha anat actualitzant fins arribar al 52411 final. És fàcil veure que:

$$3083 \cdot 17 = 52411, \tag{2.1}$$

justament el valor d'a2. Per tant, es dona una multiplicació executada en 17 iteracions (si ens posem puristes podem dir 18: certament el programa entra una última vegada al bucle per comprovar la condició i sortir-ne). ◁

Exercici 3. *Ens demanen calcular un algoritme que ens faci el següent: Donades dues entrades emmagatzemades en les posicions de memòria* `A` *i* `B` *fer la comparació. Si* `A>B` *calculem la suma. Si* `B>A` *fem la diferència* `(B-A)` *i si són iguals que multipliqui el seu valor. Carregueu diferents valors en memòria per tal de comprovar el funcionament del programa. Quina instrucció feu servir per fer la multiplicació? Què és més eficient, una instrucció directament que faci aquesta operació o el càlcul iteratiu? Raoneu la resposta.*

Resolució. El programa que hem creat per acomplir les directrius de l'enunciat és la següent:

```
# DADES
.data
A: .word 1
B: .word 4
Z: .word 0
# PROGRAMA
.text
main:
lw a0, A # primer valor, A
lw a1, B # segon valor, B
lw a2, Z # inicialitzem a zero, no necessari pero ho fem
la a3, A # guardara l'adreca de l inici de la seccio .data
# INSTRUCCIONS
bgt a0,a1,suma
bgt a1,a0,difer
beq a0,a1,mult
```

```
# ETIQUETES
suma:
add a2,a0,a1 # suma
j end
difer:
sub a2,a1,a0 # diferencia
j end
mult:
mul a2,a0,a1 # multiplicacio
j end
end:
sw a2,24(a3) # es guarda a la posicio desitjada
nop
```

Per a B=4 i A=1 obtenim a a2 el resultat 3, ja que B>A i s'executa la resta. Per a B=1 i A=4 obtenim a a2 el resultat 5, ja que A>B i s'executa la suma. Per a B=3 i A=3 obtenim a a2 el resultat 9, ja que A=B i s'executa la multiplicació.

Per pura lògica, una instrucció que efectuï directament la multiplicació serà més efectiva que un bucle. En efecte, una operació aritmètica simple és possible amb la comanda `mult` i, l'altra opció, és fer *n* operacions aritmètiques simples per arribar al mateix resultat. En altres paraules, la complexitat algorísmica és molt més alta en la segona.

El programa fa, donades dues entrades emmagatzemades en les posicions de memòria A i B fa la comparació; si A>B calcula la suma; si B>A fa la diferència (B-A) i si són iguals multiplica el seu valor. És prou simple i té una estructura molt visible, potser destacar l'ús d'etiquetes per avançar a la part end del programa quan es dona una condició en concret i s'executen totes les seves instruccions, per tal que no s'executi la resta de condicions. Al seu torn, fem ús d'un registre que serveix d'acumulador, ja que una vegada a end, l'utilitzem per l'assignació del resultat que se'ns demana, és a dir, que es guardi a la posició de memòria 24h bytes després de l'inici de la secció `.data`, és a dir, 0x10000018. ◁

2.2 Exercicis diapositives

Exercici 4. *Feu un programa en assemblador que carregui dos valors de una determinada posició de memòria en dos registres:* a1 *i* a0*, faci la suma i ho guardi en* a2*. Finalment guarda el resultat en memòria.*

Demostració. La resolució és el següent programa:

```
.data
xx: .word 1
yy: .word 4
.text
lw a0, xx
lw a1, yy
add a2, a0, a1
```

```
la a3, xx
sw a2, 8(a0)
nop
```

◁

Exercici 5. *Feu un programa en assemblador que carregui dos valors de la memòria en dos registres. Si els valors són iguals, que faci el producte, si un és negatiu que faci la suma, si els dos són positius que faci la resta i si tots dos són negatius, que els passi a positiu i els guardi en memòria.*

Resolució.

```
.data
xx: .word -8
yy: .word -9

.text
main:
lw a0, xx
lw a1, yy
la a2, xx

beq a0, a1, mult
bgtz a0, pos
bltz a0, negat

pos:
bgtz a1, resta
j suma
negat:
bltz a1, assignacio
j suma
mult:
mul a2, a0, a1
j end
suma:
bgt a0, a1, suma01
j suma02
suma01:
add a2, a0, a1
j end
suma02:
add a2, a1, a0
j end
resta:
bgt a0, a1, resta01
j resta02
resta01:
```

```
sub a2, a0, a1
j end
resta02:
sub a2, a1, a0
j end
assignacio:
neg a0, a0
neg a1, a1
sw a0, 0(a2)
sw a1, 4(a2)
j end

end: nop
```

◁

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.
6. familiaritzar-nos amb el funcionament dels registres que hi ha a la CPU. Per tal d'assolir aquest objectiu, farem servir el simuladors RIPES.

Hem après l'ús de vectors (s'ha de fer notar que és molt *strightforward*, tot i ser un llenguatge d'assemblador, la qual cosa és un fet d'agrair) i ens hem endinsat més en particular en els salts condicionals, però també en els incondicionals, així com l'emmagatzemament a memòria i els bucles, entre d'altres. M'ha sorprès especialment que l'exercici més complicat (en el meu parer) sigui el de les diapositives i no pas el de la mateixa pràctica; en qualsevol cas, reafirma la necessitar de fer tots els exercicis possibles per a practicar. En definitiva, seguim aprenent del món del Risc-V. Gens malament per la nostra segona pràctica!