Laborator 0x0C

15 Jan 2025

1. Reprezentarea binară a instrucțiunilor
2. Instrucțiuni de bază ale procesorului -> asmpa
3. Pipelines
4. Hazarduri
5. Performanța
6. Memory Cache

Reprezentarea binară a instrucțiunilor

În RISC-V, spre deosebire de x86, toate instrucțiunile sunt codificate pe dimensiune fixă, pe 32 biți = 4 B = 4 octeți

Avem, în RISC-V, 6 clase de instrucțiuni, R, I, S, B, U, J

R - type

31	30	29	28	27	26	25	24	23	22	21	20
f7	f7	f7	f7	f7	f7	f7	rs2	rs2	rs2	rs2	rs2

19	18	17	16	15	14	13	12	11	10	09	08
rs1	rs1	rs1	rs1	rs1	f3	f3	f3	rd	rd	rd	rd

07	06	05	04	03	02	01	00
rd	op	op	op	op	op	op	op

J - type imm [11:0]

31	30	29	28	27	26	25	24	23	22	21	20
im	im	im	im	im	im	im	im	im	im	im	im

19	18	17	16	15	14	13	12	11	10	09	08
rs1	rs1	rs1	rs1	rs1	f3	f3	f3	rd	rd	rd	rd

07	06	05	04	03	02	01	00
rd	op	op	op	op	op	op	op

B - type imm [12 | 10 : 5] rs imm [4 : 1 | 11]

31	30	29	28	27	26	25	24	23	22	21	20
imm	imm	imm	imm	imm	imm	imm	rs2	rs2	rs2	rs2	rs2

19	18	17	16	15	14	13	12	11	10	09	08
rs1	rs1	rs1	rs1	rs1	f3	f3	f3	imm	imm	imm	imm

07	06	05	04	03	02	01	00
imm	opc	opc	opc	opc	opc	opc	opc

J - type imm [20 | 10 : 1 | 11 | 19 : 12]

31	30	29	28	27	26	25	24	23	22	21	20
imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm

19	18	17	16	15	14	13	12	11	10	09	08
imm	imm	imm	imm	imm	imm	imm	imm	rd	rd	rd	rd

07	06	05	04	03	02	01	00
rd	opc	opc	opc	opc	opc	opc	opc

U - Type imm [31 : 12]

31	30	29	28	27	26	25	24	23	22	21	20
imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm

19	18	17	16	15	14	13	12	11	10	09	08
imm	imm	imm	imm	imm	imm	imm	imm	rd	rd	rd	rd

07	06	05	04	03	02	01	00
rd	opc	opc	opc	opc	opc	opc	opc

Câmpuri identificate în reprezentare

- func 7 = un câmp de funcție pe 7 biți
- func 3 = un câmp de funcție pe 3 biți
- rd = reg. destinație
- rs1 = reg sursă 1
- rs2 = reg sursă 2
- op = op code, merem pe 7 biți

Exemplu

add a0, a1, a2

Știm:

instrucțiunea add este repr. pe clasa R

func 7 = 0 0 0 0 0 0 0

func 3 = 0 0 0

op code = 0 1 1 0 0 1 1

Identificăm

rd = a0 = 10 = 0b 0 1 0 1 0

r1 = a1 = 11 = 0b 0 1 0 1 1

r2 = a2 = 12 = 0b 0 1 1 0 0

0b 0000 0000 1100 0101 1000 0101 0011 0011

f7 r2 r1 f3 rd op

0x 00 C5 85 33

Exemplu 2

```
addi   a0,  a1,  15
```

Știm

instrucțiunea addi → clasa I

funct7 = 0000000

funct3 = 000

opcode = 0010011

imm = 15 = 0b 0000 0000 1111

Identificăm

rd = a0 = 10 = 0b 01010

rs1 = a1 = 11 = 0b 01011

0b 0000 0000 1111 01011 000 01010 0010011

imm rs1 f3 rd op

0x 00 F5 85 13

b et → clasa I ⇒ salturi mai mici

j et → clasa J ⇒ salturi mai mari

et ↑ adresă de memorie

Instrucțiunile de bază ale procesorului

sll = shift logical left

sll rd, rs1, rs2

$\Rightarrow$ rd := rs1 << rs2

= rs1 * 2^{rs2}

0b 0100 << 1 = 0b 1000

0b 0001 1000 0100 << 2 = 0b 0110 0001 0000

0b 0110 0001 0000 >> 3 = 0b 0000 1100 0010

lui rd, imm → rd = im << 12

= im * 2^{12}

auipc rd, imm → rd = pc + (imm << 12)

↳ PC - program counter
- pleacă inițial de la zero
- la fiecare instrucțiune executată, crește cu 0x4

Exemplu

Fie PC = 0x1000

Executăm auipc a0, 0x3

Cum se execută?

a0 = PC + (imm << 12) = 0x1000 + (0x3 << 12)

= 0x1000 + 0x3000

= 0x4000

Pipelines

- lucrăm cu un procesor în 5 stadii

Fetch (se preia instrucțiunea din memorie) 00 C5 85 33

Decode (se decodifică codul hexa pentru a det. ce trebuie executat mai exact) add a0, a1, a2

Execute (se execută, folosind UAL)

Memory (citire / scriere în memoria principală)

Write Back (salvare în reg. / memorie, pentru utilizare de alte instr în viitor)

Hazarduri

- o problemă care împiedică procesorul să execute eficient instr. în pipeline

- avem hazard de
  - date
  - control
  - structurale

Hazarduri de date

data forwarding : nu așteptăm să se finalizeze instrucțiunile care datele de care avem nevoie, ci folosim valori intermediare

stalling : blocam pipeline-ul pentru a aștepta după date

out-of-order execution : se păstrează ordinea de preluare și decodificare (Fetch & Decode), dar celelalte trei nu sunt neapărat în ordine. Instrucțiunile se pun într-un buffer, pentru ca, în final, să păstrăm ordinea

Hazarduri de control

- apar în situația de branching - procesul nu știe ce instr. să preia până nu se valorează condiția saltului
- se folosește branch prediction

Hazarduri structurale

- două sau mai multe instr. încearcă să folosească aceeași resursă hardware în același timp

Exercițiu :

Avem un procesor (F, D, E, M, WB). Considerăm:

```
lw  a0, 0(gb)
add a1, a0, a2
sw  a1, 4(gb)
```

Identif. hazardul de date.

Folosim stalling pt. a rezolva acest hazard, adica instr. dependentă trebuie să finalizeze WB pt a intra în procesarea curentă.

Care este numărul total de cicluri în care se execută secvența ?

lw	F	D	E	M	WB				
add		F	D	S	S	E	M	WB	
sw			F	D	S	S	S	S	E	M	WB

raspuns : 11 (numără căsuțele)

Performanța

Avem mai multe măsuri :

- Throughput - cate instr. pot fi finalizate pe unitatea de timp
- Latența - timpul total necesar pt. a finaliza o singură instr.
- CPI - cicli per instrucțiune - nr. de cicluri necesari pt. a finaliza o instr
- IPC - instrucțiuni per ciclu - $\frac{1}{CPI}$

Exemplu 1

Un program execută un nr. egal de instr. aritmetice, de accesare de memorie și cu floating points.

O instr. aritmetică are 4 cicli, una de accesare de mem. are 5, respectiv una de fp are 6

Care sunt CPI, IPC ?

$CPI = 0.33 * op\_aritm + 0.33 * acc\_mem + 0.33 \cdot fp$

$= 0.33 * 4 + 0.33 \cdot 5 + 0.33 \cdot 6 = 4,95$

$IPC = \frac{1}{CPI} = \frac{1}{4,95} = 0.202$

Exemplu 2

Pp. ca 20% din cereri avem instr. aritmetice, în 50% dintre accesare de mem, iar în rest fp.

O instr. aritm. are 1 ciclu, dar necesită o accesare suplimentară de mem

O acc. de mem. are 3 cicli

O op. de fp are 5 cicli

Care este performanța ? (CPI, IPC)

$CPI = 0.2 * op\_aritm + 0.5 * acc\_mem + 0.3 * fp$

$= 0.2 * (1 \text{ ciclu aritm} + 3 \text{ cicli acc. mem}) + 0.5 * 3 \text{ cicli}$

$+ 0.3 * 5 \text{ cicli}$

$= 0.2 * 4 + 0.5 * 3 + 0.3 * 5 = 3.8$

$IPC = \frac{1}{3,8} = \dots$

Memoria cache

- este un tip de mem. rapidă, util pt. a creşte performanţa procesorului
- se foloseşte de accesarea mem. RAM
- este amplasată în apropierea procesorului
- poate fi văzută ca o mem. intermediară

Se ocupă cu maparea adreselor în cache

Avem 3 tipuri de mapare:

- mapare directă: fiecare bloc din mem. principală poate fi mapat doar într-o locaţie specifică în cache
- mapare asociativă: orice bloc din mem. principală poate fi mapat în orice locaţie din cache
- mapare x-way set asociativă - cache-ul este împărţit în seturi, şi un bloc din mem principală poate fi mapat într-un subset specific de locaţii în cache

Exemplu:

Pp. că avem un sistem de calcul cu o memorie principală de 2^{19} bytes şi un cache de 16 KB.

Memoria este împărţită în blocuri de 32 bytes.

Câte blocuri sunt în mem. principală? Câte blocuri sunt în mem. cache?

Sol:

nr blocuri mem. princ. = $2^{18} / 2^7 = 2^{11}$ blocuri

nr blocuri din cache = 16 KB / 32 bytes = $2^4 \cdot 2^{10} / 2^5$

= 2^9 blocuri

Exemplu 2

Avem o memorie principala de 2^{18} bytes, iar cache-ul are o capacitate de 8 KB, iar capacitatea unui bloc este de 128 bytes

Câți nr. de blocuri din mem., respectiv nr. de blocuri din cache?

Într-o schemă de mapare directă, unde se va mapa 0x A1F0?

Sol:

nr. blocuri mem principala = $2^{18} / 2^7 = 2^{11}$ blocuri în mem p.

nr. blocuri mem. cache = 8 KB / 128 bytes = $2^3 \cdot 2^{10} / 2^7$

= 2^6 blocuri cache

0x A1F0 = 0b 1010 0001 1111 0000

Avem de aflat urm. informații

- **offset** - se obține din dimensiunea blocului
  - dacă blocul este 2^n, offset-ul sunt **ultimii n biți din adresă**

În cazul nostru, blocul este 128 bytes => 2^7
=> sunt ultimii 7 biți din adresă

0b 111 0000

- **index** - se obține din dimensiunea cache-ului, din nr. de blocuri din cache
  - dacă am cache 2^m => **următorii m biți**, de la **dreapta la stânga**

În cazul nostru, avem 2^6 blocuri în cache => următorii 6 biți

0b 0000 11

- **tag** - ce rămâne

0b 0101

offset = 0b 111 0000 = 0x 70
index = 0b 00 0011 = 0x 3
tag = 0b 101 = 0x 5

Adresa 0x A1 F0 se mapează la linia (index) la offsetul (ofset)