Arhitectura Sistemelor de Calcul
Laborator Partea 0x02

Ruxandra Balucea
Bogdan Macovei
Cristian Rusu

Noiembrie 2024

Cuprins

1 Introducere

1.1 Context

RISC-V este un set de instructiuni(**ISA** - *Instructions Set rchitecture*) RISC dezvoltat în 2010 la Universitatea Berkeley din California ce prezintă următoarele caracteristici principale:

1. Este o arhitectură **open-source.** Spre deosebire de x86 si ARM, oricine poate utiliza si implementa RISC-V fără taxe de licentiere sau restrictii, promovând inovarea si reducând dependenta de furnizorii comerciali.
2. **Modularitate.** RISC-V defineste un set de instructiuni de bază minimalist la care se pot adăuga extensii optionale (pentru operatii în virgulă mobilă, vectori, criptografie, cache management etc).
3. **Portabilitate.** Arhitectura este neutră din punct de vedere hardware, ceea ce o face usor de implementat pe diverse tipuri de dispozitive. RISC-V poate fi utilizat atât în procesoare simple de IoT, cât si în procesoare complexe pentru servere si supercomputere.

Comunitatea globală de dezvoltatori contribuie activ la standardizarea si implementarea RISC-V, iar organizatii precum **RISC-V International**[1] coordonează aceste eforturi. Dintre membrii fondatori amintim: Andes, Antmicro, ETH Zurich, Google, IBM, lowRISC, Microchip, Qualcomm, Rambus, Rumble, SiFive etc.

Inovatia accelerată, independenta tehnologică si costurile reduse fac din RISC-V alegerea a numeroase companii pentru dispozitive embedded, mobile, centre de date si supercomputere, inteligentă artificială, automotive si chiar aplicatii militare si guvernamentale.

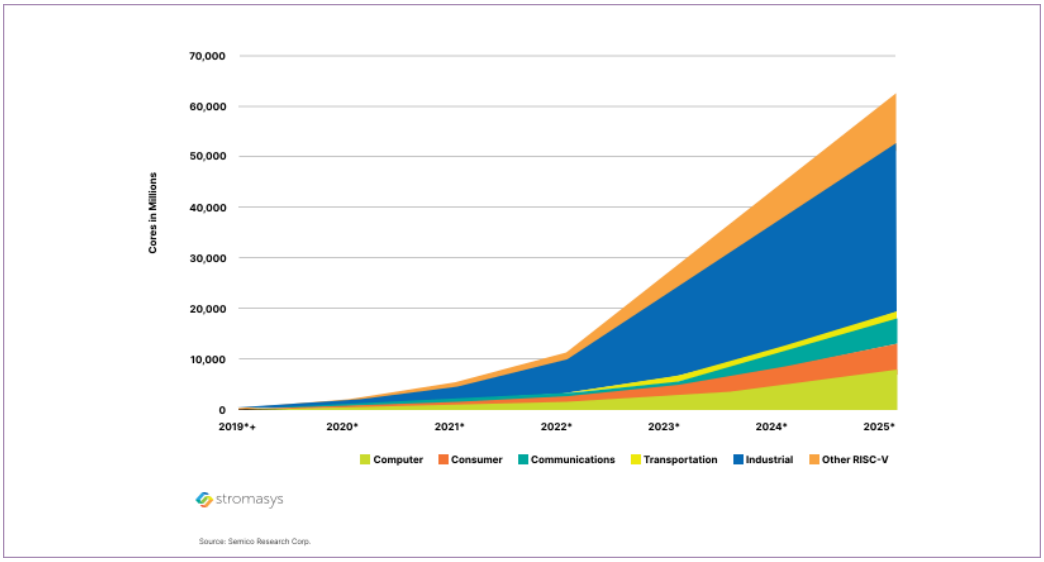


Figura 1: Evolutia RISC-V [2]

[1] https://riscv.org/
[2] https://www.stromasys.com/resources/all-about-the-risc-v-processors/

1.2 Formatul instructiunilor si encodarea

Arhitecturile RISC-V si x86 adoptă abordări fundamental diferite în ceea ce priveste formarea instructiunilor si codificarea lor, reflectând filosofia generală a designului lor. În timp ce RISC-V este simplă si modulară, cu un set fix de formate de instructiuni, x86 este o arhitectură mai complexă si mai veche, cu codificări variabile.

Codificarea instructiunilor x86 ocupă între 1 si 15 octeti si poate include diferite componente ce fac decodarea mai lentă si mai complicată. Dintre acestea amintim:

prefixuri (de exemplu pentru diferentierea dimensiunii registrului - **mov %eax, %ebx** are encodarea **89 c3**, pe cand **mov %ax, %bx**) are encodarea **66 89 c3**;

opcode-uri (mnemonica) - specifică instructiunea principală;

moduri de adresare - registru/memorie;

operanzi suplimentari.

Pe de cealaltă parte, instructiunile RISC-V au lungime fixă de 32 de biti (4 bytes) si respectă una dintre cele câteva formate standard (de exemplu, R, I, S, B, U, J), ceea ce face decodarea rapidă si simplă pentru hardware.

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

Figura 2: Formatul instructiunilor RISC-V [3]

Exemplu: Se consideră instructiunea **add a0, a1, a2** care realizează suma dintre valorile regăsite în registrii **a1**, respectiv **a2** si o depune în **a0**. Aceasta este o instructiune de tip R, având definite în manualul cu specificatiile arhitecturii[4] **func7 = 0000000**, **func3 = 000** si **opcode = 0110011**. În plus stim codurile registrilor ca fiind următoarele: (în decimal) **a0 = 10**, **a1 = 11**, **a2 = 12**. Luând în considerare informatiile anterioare, encodarea instructiunii va fi următoarea:

func7	rs2	rs1	funct3	rd	opcode
0 0 0 0 | 0 0 0	0 | 1 1 0 0 |	0 1 0 1 | 1	0 0 0 |	0 1 0 1 | 0	0 1 1 | 0 0 1 1

Asadar, grupând câte 4 biti si covertind în hexa, o să obtinem encodarea **0x00c58533**.

Exercitiu: Care este encodarea pentru instructiunea **addi a0, a1, 15** care este de tip **I** si realizează suma dintre imediatul **15** si valoarea din registrul sursă **a1**, depozitând valoarea în registrul destinatie **a0** (vezi codurile registrilor mai sus)? De asemenea, se cunosc **func7 = 0000000**, **func3 = 000** si **opcode = 0010011**.

[3] https://five-embeddev.com/riscv-user-isa-manual/Priv-v1.12/instr-table.html
[4] https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

1.3 Extensii RISC-V

Asa cum am prezentat anterior, una dintre principalele avantaje ale arhitecturii este modularitatea si împărtirea instructiunilor pe functii specifice care să poată fi folosite sau nu în acord cu scopul procesorului.

La nucleul RISC-V se află setul de instructiuni de bază, cunoscut sub denumirea de **RV32I** pentru arhitecturi pe 32 de biti si **RV64I** pentru arhitecturi pe 64 de biti. Acestea includ un set minim de instructiuni necesar pentru a implementa o arhitectură de procesor functională. Caracteristicile principale ale bazei includ:

Operatii aritmetice de bază (add, sub etc);

Operatii de acces la memorie (lw, sw etc.);

Instructiuni de control-flow (jal, beq etc)

De asemenea, dintre extensiile standardizate amintim:

RV32M - suport pentru operatii aritmetice mai avansate, cum ar fi multiplicarea (mul) si împărtirea (div). Este esentială pentru aplicatii care necesită calcul numeric intensiv.

RV32C - subset de instructiuni compresate, reducând dimensiunea codului si economisind memorie. Este utilă în sisteme integrate sau alte aplicatii cu resurse limitate.

RV32A - suport pentru operatii atomice, care sunt cruciale în programele multithread. Instructiunile precum lr (load-reserved) si sc (store-conditional) permit sincronizarea eficientă între mai multe fire de executie.

RV32F si **RV32D** - suport pentru operatii în virgulă mobilă. Acestea sunt esentiale în aplicatii precum procesarea semnalelor digitale si învătarea automată.

RV32G - este o extensie generală ce este echivalentă cu RV32IMAFD.

RV32V - suport pentru calcul vectorial, esential în aplicatii precum procesarea imaginilor, simulările stiintifice si deep-learning.

RV32B - instructiuni optimizate pentru manipularea bitilor, folosite în criptografie si procesarea semnalelor

La fel ca setul de bază, toate aceste extensii suportă si variante pe 64 de biti.

În plus, organizatiile si companiile pot defini propriile extensii pentru nevoi specifice, contribuind la diversitatea implementărilor RISC-V.

Toate extensiile ratificate, precum si statusul curent al extensiilor în curs de ratificare pot fi consultate aici[5].

[5] `https://lf-riscv.atlassian.net/wiki/spaces/HOME/pages/16154732/Ratified Extensions`

1.4 Comparatie x86 vs RISC-V

În finalul acestei sectiuni de introducere, prezentăm un tabel cu principalele avantaje si dezavantaje ale celor 2 arhitecturi studiate. Acest tabel poate fi privit chiar ca o comparatie CISC vs RISC.

	x86 (CISC)	**RISC-V (RISC)**
Avantaje	Codificare compactă pentru instructiuni simple → economisirea memoriei în aplicatii vechi. Suport pentru moduri complexe de adresare, util pentru software legacy.	Simplitate si uniformitate → usor de implementat în hardware si rapid de decodat. Scalabilitate → extensiile pot fi adăugate fără a afecta codificările existente. Pipeline eficient → formatele fixe permit instructiuni care pot fi decodate în paralel.
Dezavantaje	Decodare lentă si complicată → dificultăti în proiectarea procesoarelor moderne. Mostenirea istorică → dificultăti în a elimina instructiunile redundante sau ineficiente.	Uneori necesită mai multe instructiuni pentru operatiuni complexe, comparativ cu x86.

Tabela 1: Comparatie cu avantaje si dezavantaje între x86 si RISC-V

2 Mediul de lucru

Arhitectura RISC-V beneficiază de un ecosistem divers de tool-uri si simulatoare care acoperă nevoi educationale, de cercetare si industriale. Mai jos sunt principalele tool-uri si simulatoare utilizate în ecosistemul RISC-V, împărtite în categorii relevante.

Simulatorul oficial este **Spike**[6] si este dezvoltat de RISC-V International, având suport pentru toate extensiile standard si putând fi folosit pentru testare si validare. Acesta ese destul de lent, principalul focus fiind pe acuratete. Pe de cealaltă parte, **QEMU**[7] este un **emulator** si suportă mai multe arhitecturi (ARM, X86 etc.) si mai multe modalităti de configurare de sistem, fiind potrivit pentru dezvoltarea de sisteme de operare si aplicatii embedded. Permite integrarea de tool-uri precum **gdb** si este mult mai rapid, translatarea codului făcându-se dinamic din ISA-ul emulat în ISA-ul nativ al masinii.

Un subdomeniu extrem de important pentru dezvoltarea ecosistemului RISC-V este reprezentat de design-ul hardware. Dintre cele mai cunoscute limbaje utilizate pentru proiectarea si validarea procesoarelor amintim **Chisel**[8] care permite definirea de scheme hardware pentru design-ul de procesoare. De asemenea, ca o platformă ce înglobează multe dintre procesoarele astfel dezvoltate, **Chipyard**[9] oferă suport pentru diverse core-uri (RocketChip, BOOM etc.) si acceleratoare, oferind de asemenea posibilitatea rulării în cloud pe un **FPGA** (circuit fizic reprogramabil pe baza unei scheme a unui sistem).

De asemenea, dintre cele mai imporante tool-uri software dezvoltate amintim:

RISC-V GNU Toolchain[1] - compiler GCC, binutils, si GDB pentru RISC-V.

LLVM/Clang pentru RISC-V[11] - compilare optimizată, suport extins pentru extensii ISA.

Majoritatea acestor ustensile de lucru sunt necesare a fi build-uite si adaptate pentru mediul de lucru al fiecăruia, proces ce poate dura si câteva ore. Din acest motiv, în cadrul laboratorului vom utiliza un simulator online cu o interfată user-friendly, simulator proiectat pentru utilizarea în scop educational - **Ripes** - `https://ripes.me/`. Sursele pentru acest simulator care poate fi si instalat local se gasesc aici[12]. De asemenea, un tutorial amănuntit despre functionalitătile simulatorului pot fi regăsite în sectiunea de documentatie.

Acesta permite vizualizarea executiei instructiunilor într-un procesor simplificat, fie single-cycle, fie într-un pipeline. Oferă o interfată intuitivă pentru explorarea etapelor procesorului, modificarea registrilor, memoriei si stivei si rularea programelor în assembly RISC-V.

Ripes implementează extensia de bază RV32I, permitând si activarea extensiilor C si M.

Vom reveni asupra altor detalii despre acest simulator într-o sectiune ulterioară, punctând acum doar ce este necesar pentru a observa conceptele studiate anterior:

[6] `https://github.com/riscv-software-src/riscv-isa-sim`
[7] `https://www.qemu.org/`
[8] `https://www.chisel-lang.org/docs`
[9] `https://chipyard.readthedocs.io/en/stable/Chipyard-Basics/Chipyard-Components.html`
[1] `https://github.com/riscv-collab/riscv-gnu-toolchain`
[11] `https://github.com/llvm/llvm-project`
[12] `https://github.com/mortbopet/Ripes`

Sectiunile statice (.text, .data, .bss) pot fi navigate prin butonul `Go to section:`.

Stiva poate fi vizualizată prin folosirea butonului `Go to register:` si alegerea registrului **sp** care mentine (similar cu esp), vârful stivei. Remarcăm pozitionarea acesteia la o adresă mare 0x7 0, asa cum am discutat spatiul de adresă al unui proces în cadrul laboratorului despre proceduri pentru arhitectura x86.

Registrii - acestia pot fi vizualizati in tabelul din dreapta paginii.

Codul - scrierea se face direct în fereastră putând observa o conversie instantă în binar, precum si varianta dezasamblată (encodată si ulterior decodată prin aplicarea succesivă a unui asamblor si a unui dezasamblor). De asemenea, se pot si încărca direct fisiere scrise local.

Atentie!

1. Pentru rularea instructiunilor pas cu pas, este nevoie sa schimbati procesorul din iconita de sub *File*, într-un **Single-cycle processor**.
2. Uneori site-ul refuza sa se încarce. Aceasta problema poate fi rezolvata prin curătarea memoriei cache sau prin deschiderea simulatorului într-o fereastră incognito.

3 Limbajul Assembly RISC-V

3.1 Registrii arhitecturii

RISC-V dispune de 32 de registrii generali (x0 până la x31), fiecare având o dimensiune de 32 de biti (pentru arhitectura de 32 de biti). Acesti registri sunt utilizati pentru manipularea datelor, fiind folositi la fel ca la x86 ca niste variabile.

Encodare	**Encodare comprimată**	**Registru**	**Nume ABI**	**Descriere**	**Callee-saved**
0	-	`x0`	`zero`	Hardwired zero	-
1	-	`x1`	`ra`	Return address	Da
2	-	`x2`	`sp`	Stack pointer	Da
3	-	`x3`	`gp`	Global pointer	-
4	-	`x4`	`tp`	Thread pointer	-
5	-	`x5`	`t0`	Temporary register 0	Nu
6	-	`x6`	`t1`	Temporary register 1	Nu
7	-	`x7`	`t2`	Temporary register 2	Nu
8	0	`x8`	`s0/fp`	Saved register 0 / frame pointer	Da
9	1	`x9`	`s1`	Saved register 1	Da
10	2	`x10`	`a0`	Function argument 0 / return value 0	Nu
11	3	`x11`	`a1`	Function argument 1 / return value 1	Nu
12	4	`x12`	`a2`	Function argument 2	Nu
13	5	`x13`	`a3`	Function argument 3	Nu
14	6	`x14`	`a4`	Function argument 4	Nu
15	7	`x15`	`a5`	Function argument 5	Nu
16	-	`x16`	`a6`	Function argument 6	Nu
17	-	`x17`	`a7`	Function argument 7	Nu
18	-	`x18`	`s2`	Saved register 2	Da
19	-	`x19`	`s3`	Saved register 3	Da
20	-	`x20`	`s4`	Saved register 4	Da
21	-	`x21`	`s5`	Saved register 5	Da
22	-	`x22`	`s6`	Saved register 6	Da
23	-	`x23`	`s7`	Saved register	Da
24	-	`x24`	`s8`	Saved register 8	Da
25	-	`x25`	`s9`	Saved register 9	Da
26	-	`x26`	`s10`	Saved register 10	Da
27	-	`x27`	`s11`	Saved register 11	Da
28	-	`x28`	`t3`	Temporary register 3	Nu
29	-	`x29`	`t4`	Temporary register 4	Nu
30	-	`x30`	`t5`	Temporary register 5	Nu
31	-	`x31`	`t6`	Temporary register 6	Nu

Tabela 2: Registrii de uz general RISC-V

Un alt registru de interes este registrul **pc (program counter)** care mentine întotdeauna adresa instructiunii ce urmează a fi executată.

3.2 Tipuri de date

Tipurile principale de date pe care le vom utiliza sunt

1. **byte** - după cum îi spune si numele, ocupă 1 byte, adică 8 biti în memorie;
2. **halfword** - ocupă 2 bytes (16 de biti) si este utilizat pentru stocarea numerelor fractionare;
3. **word** - ocupă 4 bytes (32 biti) si este utilizat pentru stocarea întregilor. **long** ocupă tot 32 de biti.
4. **doubleword** - ocupa 8 bytes (64 biti);
5. **ascii**, **asciz** si **space** cu aceeasi semnificatie ca pe x86 (**space** nu este implementat de Ripes).

3.3 Instructiunile principale

Instructiune	**Operanzi**	**Descriere**
`add`	`rd, rs1, rs2`	Adună `rs1 + rs2` si salvează în `rd`.
`sub`	`rd, rs1, rs2`	Scade `rs2` din `rs1` si salvează în `rd`.
`mul`	`rd, rs1, rs2`	Înmulteste `rs1` cu `rs2` (M-extension).
`div`	`rd, rs1, rs2`	Împarte `rs1` la `rs2` (M-extension).
`rem`	`rd, rs1, rs2`	Restul împărtirii întregi între `rs1` si `rs2`.
`and`	`rd, rs1, rs2`	Realizează `and` logic între `rs1` si `rs2`.
`or`	`rd, rs1, rs2`	Realizează `or` logic între `rs1` si `rs2`.
`xor`	`rd, rs1, rs2`	Realizează `xor` logic între `rs1` si `rs2`.
`sll`	`rd, rs1, rs2`	Shift la stânga logic cu valoarea din `rs2`.
`srl`	`rd, rs1, rs2`	Shift la dreapta logic cu valoarea din `rs2`.
`addi`	`rd, rs1, imm`	Adună `rs1 + imm` si salvează în `rd`.
`slti`	`rd, rs1, imm`	Compară `rs1 < imm` si setează `rd` la 1 sau 0.
`andi`	`rd, rs1, imm`	Realizează `and` între `rs1` si `imm`.
`lui`	`rd, imm`	Încarcă imm pe cei 20 de biti superiori din `rd` (`rd = imm` $\ll$ `12`)
`auipc`	`rd, imm`	Încarcă imm pe cei 20 de biti superiori din `rd`, adunând la această valoare si pc (`rd = pc + (imm` $\ll$ `12)`).

Tabela 3: Principalele instructiuni RISC-V si operanzii lor

De asemenea, există si câteva pseudoinstructiuni importante. O pseudoinstructiune este o instructiune care poate fi folosită în asamblare, dar care nu are o encodare proprie, ci este de fapt convertită în una sau mai multe instructiuni clasice.

Pseudoinstructie	**Instructiuni echivalente**	**Descriere**
`li rd, imm`	`lui rd, imm[31:12]` `addi rd, rd, imm[11:0]`	Încarcă valoarea imediată `imm` în registrul `rd`.
`la rd, label`	`lui rd, label[31:12]` `addi rd, rd, label[11:0]`	Încarcă adresa etichetei `label` în registrul `rd`.
`move rd, rs`	`addi rd, rs, 0`	Copiază valoarea din registrul `rs` în registrul `rd`.
`nop`	`addi x0, x0, 0`	Instructiune "no operation", nu face nimic.

Tabela 4: Principalele pseudoinstructiuni RISC-V

3.4 Arhitectură load-store

RISC-V este o arhitectură bazată pe principiul load-store, ceea ce înseamnă că toate operatiile de procesare a datelor trebuie să utilizeze exclusiv registre pentru calcule, iar memoria poate fi accesată doar prin instructiuni dedicate de încărcare (load) si stocare (store). Această abordare este esentială în designul arhitecturilor RISC (Reduced Instruction Set Computing).

Deoarece toate operatiile implică doar registri, numărul de etape în pipeline-ul procesorului este redus, ceea ce duce la o executie mai eficientă a instructiunilor. Memoria este accesată doar în etapele de load si store, reducând complexitatea logicii de executie.

Pentru încărcarea unei valori din memorie vom folosi instructiunea:

```
lw reg1, offset(reg2)
```

unde `reg2` retine o adresă de memorie, `offset` reprezintă numărul de bytes adăugati pentru a ajunge pe pozitia din care vrem să citim, iar `reg1` este registrul în care se va încărca valoarea din memorie. Pentru stocarea unei valori în memorie vom folosi instructiunea:

```
sw reg1, offset(reg2)
```

unde `offset` si `reg2` sunt folositi similar pentru a calcula adresa din memorie la care se face stocarea, iar `reg1` este registrul din care se ia valoarea ce urmează a fi depozitată.

Aceste variante de instructiuni se aplică pe o dimensiune de 32 de biti. Similar există instructiuni pentru a lucra cu 16 biti (`lh` si `sh`), respectiv cu 8 biti (`lb` si `sb`).

Registrul `gp` este folosit pentru a retine adresa sectiunii `.data`. Una dintre modalitătile de accesare a datelor din memorie, este asadar prin folosirea unui o set raportat la baza sectiunii.

Exercitiu: Pentru o exemplificare, să considerăm secventa următoare de cod. Cum va arăta memoria în urma executiei celor 2 instructiuni? Ce observati legat de declaratiile din zona `.data`?

```
.data
    .long 5
    .long 6

.text
.global main
main:
    lw a0, 0(gp)
    sw a0, 4(gp)
```

3.5 Relocări

Pentru a putea folosi în continuare simbolii declarati în memorie în cadrul instructiunilor si a nu fi nevoiti să calculăm de fiecare o set-ul pentru accesarea datelor, este nevoie să ne uităm peste conceptul de relocare.

O relocare este un proces utilizat în timpul compilării, linkării sau încărcării unui program pentru a ajusta adresele de memorie sau referintele la simboluri, astfel încât programul să poată fi executat corect indiferent de pozitia sa în memorie. Majoritatea operanzilor de tip imediat acceptă o variantă de relocare pentru deducerea ulterioară a respectivului o set. Dacă pozitionarea sectiunilor este statică si simbolii sunt prezenti în sursa curentă, adresa lor se poate deduce în timpul procesului de compilare, adresa fiind calculată automat. Dacă simbolii în schimb sunt necunoscuti si sunt aflati în timpul procesului de linkare (sunt declarati în alte surse, biblioteci etc), iar pozitiile sectiunilor sunt fixe (în special pentru dispozitivele embedded unde sunt predefinite printr-un script de configurare), adresa este calculată în această etapă. Altfel, în cazul în care sectiunile sunt dinamice (pozitionate la diferite adrese la fiecare rulare), relocările vor fi rezolvate abia la rulare.

Pentru utilizarea simboluri din zonele de date, RISC-V foloseste următoarele relocări:

Relocări pentru partea superioară (%hi si %pcrel_hi). Cele două sunt folosite în cadrul instructiunilor `lui`, respectiv `auipc` pentru a prelua cei 20 de biti superior dintr-o adresă.

Relocări pentru o set-uri (%lo si %pcrel_lo). Cele două sunt folosite pentru instructiunile `lw` si `sw` pentru adăugarea o set-ului de 12 biti rămasi. De asemenea, poate fi folosită si în cadrul unei instructiuni `addi` pentru a ajuta la calcularea adresei de început (pentru un vector, un string). Această relocare lo este precedată întotdeauna de o instructiune cu relocare hi, obtinând prin această combinare adresa pe 32 de biti.

Relocări pentru salturi - vor fi prezentate în sectiunea despre salturi.

Altele.

De asemenea, o diferentiere importantă o reprezintă modul în care functionează relocările absolute, respectiv relocările relative la program counter:

%lo(sym), respectiv **%hi(sym)** vizează exact bitii inferiori, respectiv bitii superiori ai adresei simbolului `sym`;

%pcrel_hi(sym) face referire `(adr_sym - pc) >> 12` (bitii superiori);

%pcrel_lo(label) unde `label` indică spre o instructiune având o relocare de tip `%pcrel_hi(sym)` va considera `(adr_sym - label) % 2`^{12} (bitii inferiori).

Astfel, pentru a obtine adresa unei variabile, putem folosi una dintre variantele de mai jos:

```
.data
    a: .long 5

.text
.global main
main:
    # Varianta 1 (statica)
    lui a0, %hi(a)
    lw a1, %lo(a)(a0)

    # Varianta 2 (relativa cu relocari)
    l0:
    auipc a2, %pcrel_hi(a)
    lw a3, %pcrel_lo(l0)(a2)

    # Varianta 3 (convenție de scriere care se expandeaza la varianta 2)
    lw a4, a

    # Varianta 4 (relativa cu global pointer)
    lw a5, 0(gp)
```

Observatie! Programul functionează si fără un apel `exit(0)` în cazul acestui simulator, neexistând un sistem de operare.

Întrucât pentru Ripes cunoastem de la început locatia sectiunilor `.data` si `.text` si de asemenea simbolul `a` este în acelasi fisier, putem aplica toate variantele de mai sus. Vom alege în continuare să folosim varianta 3 pentru programele pe care le vom avea de implementat. **Relocările relative sunt într-o variantă instabilă pe Ripes si nu vor functiona corect!**

Exercitiu: Analizati cele 4 variante, observând ce adrese sunt retinute la fiecare pas. Care este diferenta dintre relocările %hi si %pcrel_hi, respectiv %lo si %pcrel_lo? De ce variantele relative pot fi utilizate si când nu se cunosc static adresele sectiunilor? Observati comportamentul gresit al variantei 2 (comparativ cu varianta 3).

Exercitiu: Transformati variantele 1, 2 si 4 de mai sus astfel încât în registrii `a1, a3 și a5` să obtineti adresa lui `a` (în loc de valoarea lui) folosind instructiunea `addi`.

Exercitiu: Transformati variantele de mai sus pentru stocare si verificati ce variantă este corect implementată de Ripes. **Observatie!** Toate variantele de mai sus sunt corecte arhitectural vorbind, stadiul dezvoltării tool-ului fiind singurul imp ediment în folosirea lor.

3.6 Apeluri de sistem

Pentru folosirea apelurilor de sistem mecanismul este similar cu ce am văzut pe x86, diferind doar registrii. Astfel, pentru a apela o functie sistem vom folosi instructiunea

```
ecall
```

care va produce verificarea registrilor următori:

`a7` - pentru stocarea codului functiei

`a0 - a6` - pentru încărcarea argumentelor.

De exemplu, functia `exit` are codul 93. Deci pentru realizarea apelui `exit(0)`, vom avea secventa următoare de mai jos.

```
li a7, 93
li a0, 0
ecall
```

Codurile de apel pentru **Ripes** pot fi găsite în sectiunea **Help** > **System calls**.

Exercitiu: Declarati un string si un întreg în memorie si afisati-le pe ecran.

3.7 Salturi

O instructiune importantă pentru efectuarea de salturi neconditionate este instructiunea:

```
jal rd, imm
```

care stochează valoarea `pc + 4` în registrul `rd` si face saltul la adresa `pc + imm`.

În general pentru executarea de salturi neconditionate vom folosi:

```
j label
```

care este de fapt o pseudoinstructiune pentru instructiunea:

```
jal x0, label
```

Cum registrul `x0` nu poate fi modificat, asta înseamnă că valoarea `pc + 4` nu se salvează nicăieri, iar saltul se face la adresa lui `label`.

Fiind vorba din nou despre lucratul cu simboli, avem si în acest caz o relocare specifică instructiunii - **R_RISCV_JAL** care indică înlocuirea simbolului cu adresa sa relativă la `pc`.

Pentru salturile conditionate, spre deosebire de x86, RISC-V comasează instructiunea de comparare si de salt într-una singură.

În mod similar, putem folosi în loc de imediat un simbol a cărui adresă va fi calculată relativ prin rezolvarea relocării **R_RISCV_BRANCH**.

Să considerăm următorul exemplu:

Instructiune	Operanzi	Descriere
blt(u)	rs1, rs2, imm	Dacă rs1 < rs2, salt la pc + imm
ble(u)	rs1, rs2, imm	Dacă rs1 <= rs2, salt la pc + imm
bgt(u)	rs1, rs2, imm	Dacă rs1 > rs2, salt la pc + imm
bge(u)	rs1, rs2, imm	Dacă rs1 >= rs2, salt la pc + imm
beq	rs1, rs2, imm	Dacă rs1 == rs2, salt la pc + imm
bne	rs1, rs2, imm	Dacă rs1 ! = rs2, salt la pc + imm

Tabela 5: Principalele instructiuni RISC-V pentru salturi conditionate.

Pseudoinstructiune	Operanzi	Descriere
beqz	rs1, imm	Dacă rs1 == 0, salt la pc + imm
bnez	rs1, imm	Dacă rs1 ! = 0, salt la pc + imm

Tabela 6: Principalele pseudoinstructiuni RISC-V pentru salturi conditionate.

```
.data
 .long 6
 .long 7
 egale: .asciz  Numerele sunt egale\n
 diferite: .asciz  Numerele sunt diferite\n
.text
.global main
main:

  lw a0, 0(gp)
  lw a1, 4(gp)

  beq a0, a1, l0
  li a7, 4
  la a0, diferite
  ecall
  j final

l0:
  li a7, 4
  la a0, egale
  ecall

final:
  li a7, 93
  li a0, 0
  ecall
```

Exercitiu: Analizati programul si conversiile simbolilor în imediati.

3.8 Tablouri unidimensionale

Mecanismul de functionare este acelasi ca pe x86, dar nu există o grupare specifică pentru accesarea elementelor.

Pentru clarificare, să considerăm următorul exemplu care realizează suma elementelor dintr-un vector de întregi si o depozitează în memorie:

```
.data
    n: .long 5
    s: .long 0
    v: .long 1, 2, 3, 4, 5

.text
.global main
main:
    lw a0, 0(gp)     # se incarca n in a0
    addi a1, gp, 8   # se pune adresa lui v in a1
    li a2, 0         # in a2 vom calcula suma
    li a3, 0         # in a3 vom mentine indexul

    begin:
    bge a3, a0, final

    slli a4, a3, 2   # a4 = 4 * index
    add a4, a4, a1   # a4 = adresa lui v + 4 * index
    lw a4, 0(a4)     # a4 = *(adresa lui v + 4 * index)
    add a2, a2, a4   # adunam elementul la suma totala
    addi a3, a3, 1   # incrementam indexul
    j begin

    final:
    sw a2, 4(gp)

    li a7, 93
    li a0, 0
    ecall
```

Exercitiu: Să se rescrie exemplul anterior, modificând întâi valoarea lui `n` în `n * 4` si incrementând apoi indexul cu `4` la fiecare iteratie.

3.9 RV32C - instructiuni comprimate

Pentru a îmbunătăti performanta si eficienta memoriei, RISC-V include si un subset de instructiuni comprimate, care permit utilizarea a 16 biti pentru a reprezenta anumite instructiuni, reducând astfel dimensiunea programelor si economisind lătimea de bandă a memoriei.

RVC realizează o schemă simplă de conversie care permite scurtarea instructiunilor din setul de bază când:

1. imediatul/o setul este suficient de mic;
2. unul dintre registri este x0 (zero register), x1 (return address) sau x2 (stack pointer-ul);
3. este folosit acelasi registru pentru destinatie si pentru sursă;
4. se folosesc registrii care au o varianta comprimabilă (vezi Tabelul 2)

Exemple

```
lw t0, 4(sp)     --> c.lwsp t0, 4(sp)    // este folosit registrul sp
addi t0, t0, 1   --> c.addi t0, 1        // sursa == destinatie == t0
lw a0, 4(s0)     --> c.lw a0, 4(s0)      // a0 si s0 au variante comprimabile
lw t0, 516(sp)   --> nu se comprima      // imediatul este prea mare
```

Astfel, sunt definite următoarele formate pentru encodarea instructiunilor comprimate:

Format	Meaning	15 14 13	12	11 10	9 8 7	6 5	4 3 2	1 0
CR	Register	funct4		rd/rs1		rs2		op
CI	Immediate	funct3	imm	rd/rs1		imm		op
CSS	Stack-relative Store	funct3	imm			rs2		op
CIW	Wide Immediate	funct3	imm				rd′	op
CL	Load	funct3	imm	rs1′		imm	rd′	op
CS	Store	funct3	imm	rs1′		imm	rs2′	op
CB	Branch	funct3	offset	rs1′		offset		op
CJ	Jump	funct3	jump target					op

Figura 3: Formatul instructiunilor RVC [13]

Opcode-urile acestor operatii au fost gândite astfel încât să nu se suprapună cu cele ale instructiunilor uzuale. Sunt verificati parte dintre bitii de identificare, iar procesorul trimite instructiunea către un decodor specializat care interpretează această varianta pe 16 biti. În rest, pipelinul rămâne neschimbat.

Principalele instructiuni sunt indicate mai jos. De asemenea, un tabel cu toate instructiunile comprimate poate fi găsit aici [14]

Pentru activarea acestor instructiuni în Ripes este nevoie să bifati si extensia C în meniul de unde ati ales tipul de procesor.

Exercitiu: Să se rescrie exemplul din Subsectiunea 3.8, parcurgând sirul de la final spre început si folosind ca index initial `4 * (n - 1)`. Ce dimensiune are zona **.text**?

Exercitiu: Să se rescrie codul anterior folosind instructiuni comprimate. Care este noua dimensiune a zonei de cod?

[13] https://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-209.pdf
[14] https://five-embeddev.com/riscv-user-isa-manual/Priv-v1.12/rvc-instr-table.html

Instructiune	Operanzi	Descriere
c.addi	rd, imm	Adaugă un imediat la registrul rd.
c.addisp	rd, imm	Adaugă un imediat (16 biti) la registrul sp (pointerul de stivă).
c.add	rd, rs	Adaugă valoarea din registrul rs la registrul rd.
c.lwsp	rd, offset(sp)	Încarcă un cuvânt din memorie la adresa sp + offset în registrul rd.
c.swsp	rd, offset(sp)	Stochează un cuvânt din registrul rd la adresa sp + offset.
c.li	rd, imm	Încărcă un imediat de 16 biti în registrul rd.
c.j	offset	Salt neconditionat la adresa specificată de offset.
c.beqz	rs, offset	Salt conditionat dacă registrul rs este zero.
c.bnez	rs, offset	Salt conditionat dacă registrul rs nu este zero.
c.not	rd, rs	Neagă (complementul pe biti) valoarea din registrul rs si o stochează în registrul rd.
c.lw	rd', offset(rs')	Încarcă un cuvânt din memorie la adresa rs' + offset în registrul rd'.
c.sw	rd', offset(rs')	Stochează un cuvânt din registrul rd' la adresa rs' + offset.

Tabela 7: Principalele instructiuni RVC si operanzii lor

4 Conventia de apel

Pentru alocarea de spatiu pe stivă este necesar să folosim instructiuni aritmetice si de operare cu memoria. Desi nu există instructiuni precum push si pop, comportamentul este acelasi.

De exemplu, pentru a stoca valoarea din registrul s0 pe stivă, vom folosi următoarea secventă:

```
addi sp, sp, -4
sw s0, 0(sp)
```

În sens invers, pentru a scoate elementul din vârful stivei si a îl depozita în registrul s0, vom folosi:

```
lw s0, 0(sp)
addi sp, sp, 4
```

4.1 Apelul unei proceduri

Pentru a apela o procedură, putem folosi două instructiuni:

```
call procedura
jal procedura
```

Aceasta este de fapt o pseudoinstructiune pentru instructiunea:

```
jal x1, procedura
```

care, ne reamintim, salvează adresa instructiunii următoare în registrul `x1`, adică registrul `ra` (return address). Asa cum îi spune si numele acest registru este utilizat pentru retinerea adresei de retur. Asadar spre deosebire de x86, revenirea dintr-o functie se face bazat pe un registru, valoarea nemaifiind transmisă prin stivă. Returul se face prin instructiunea:

```
ret
```

RISC-V mai are o pseudoinstructiune de salt pe care nu am prezentat-o si anume:

```
jr reg
```

care produce saltul la adresa retinută în registrul `reg` si este la rândul ei o pseudoinstructiune pentru:

```
jalr x0, reg, 0
```

primul operand marcând la fel ca în cazul instructiunii `jal` locatia în care se va stoca adresa instructiunii următoare, iar ultimul operand reprezentând o setul.

Cum este destul de usor de remarcat, instructiunea de retur este deci doar un alias pentru:

```
jr ra
```

Asadar ca schelet de bază pentru apelul unei proceduri vom avea următoarea secventă:

```
.data
.text
.global main
main:
    call proc   // jal proc
                // jal ra, proc

    li a7, 93
    li a0, 0
    ecall

proc:

    ret         // jr ra
                // jalr x0, ra, 0
```

Atentie! Pare că Ripes începe executia întotdeauna în prima procedură pe care o găseste, motiv pentru care vom pune mereu functia `main` prima.

4.2 Argumentele unei proceduri si valoarea de retur

În cazul RISC-V, de asemenea, argumentele sunt transmise si ele prin registrii conform conventiei pe care am văzut-o si la apelurile de sistem. Registrii `a0 - a7` sunt folositi pentru transmiterea primelor 8 argumente. Dacă există mai mult de 8 argumente, acestea vor fi depozitate pe stivă ca în cazul x86.

De asemenea, valorile returnate se vor mentine în registrii `a0`, `a1`, iar în cazul în care sunt mai mult de două, vor fi transmise prin vârful stivei.

Exemplu: Fie acum functia având declaratia `proc(x,y,&sum)` care realizează adunarea a doi întregi `x` si `y` si depozitează suma în `sum` si care de asemenea returnează diferenta celor două numere. Până în acest punct, un program minimalist în care se cheamă respectiva functie cu argumentele 2, 3, respectiv adresa unei variabile din memorie, este prezentat mai jos.

```
.data
    x: .long 3
    y: .long 2
    sum: .long 0
    dif: .long 0

.text
.global main
main:
    lw a0, 0(gp)
    lw a1, 4(gp)
    addi a2, gp, 8
    call proc

    sw a0, 12(gp)

    li a7, 93
    li a0, 0
    ecall

proc:
    add a3, a0, a1
    sw a3, 0(a2)
    sub a0, a0, a1

    ret
```

4.3 Crearea cadrului de apel

La fel ca în cazul arhitecturii x86, avem nevoie să marcăm pe stivă acele locatii de unde încep să se pună pe stivă elemente din cadrul curent. Registrul folosit pentru a marca începutul acestui "frame" este `fp` (frame pointer-ul). De asemenea, pentru a nu pierde adresa de retur în cazul unor apeluri imbricate, la intrarea în functie se salvează si registrul `ra` (ambii sunt de fapt registrii callee-saved). `fp` este mutat pentru a pointa către acea locatie de pe stivă de după aceste două salvări (Ripes acceptă doar denumirea de `s0`.

Astfel o procedură va avea următorul schelet:

```
proc:
    addi sp, sp, -8
    sw ra, 4(sp)
    sw s0, 0(sp)
    addi s0, sp, 0
```

```
    // corpul procedurii

    lw s0, 0(sp)
    lw ra, 4(sp)
    addi sp, sp, 8
    ret
```

În plus, conceptul de registrii callee-saved/caller-saved functionează la fel ca pentru x86. Tipul fiecărui registru este mentionat în Tabelul 2.

Exercitiu: Observati valorile depozitate în `s3`, respectiv `t5` la finalul executiei programului. Adăugati instructiunile necesare pentru a mentine si valoarea registrului calleer-saved `t5`.

```
.data
    x: .long 3
    y: .long 2
    sum: .long 0
    dif: .long 0

.text
.global main
main:
    li s3, 10
    li t5, 11

    lw a0, 0(gp)
    lw a1, 4(gp)
    addi a2, gp, 8
    addi a3, gp, 12
    call proc

  final:
    li a7, 93
    li a0, 0
    ecall

proc:
    addi sp, sp, -8
    sw ra, 4(sp)
    sw s0, 0(sp)
    addi s0, sp, 0
    addi sp, sp, -4
    sw s3, -4(s0)

    add s3, a0, a1
    sw s3, 0(a2)

    sub t5, a0, a1
```

```
    sw t5, 0(a3)

    lw s3, -4(s0)
    addi sp, sp, 4

    lw s0, 0(sp)
    lw ra, 4(sp)
    addi sp, sp, 8
    ret
```

Exercitiu: Este posibil să facem alocarea de spatiu pe stivă o singură dată, numai că o setii pentru salvarea lui `ra` si `fp` se modifică. Adaptati programul astfel încât să alocati de la început 12 octeti, iar la final să îi stergeti.

Mai multe informatii despre conventiile de apel puteti găsi aici [15].

[15] `https://d3s.mff.cuni.cz/files/teaching/nswi200/202324/doc/riscv-abi.pdf`

5 Pipelining

Pipeline-ul unui procesor este o tehnică utilizată pentru a spori performanta procesorului, permitând executia simultană a mai multor instructiuni în etape diferite. .

Pipeline-ul este o componentă esentială în arhitectura procesoarelor moderne, fiind prezent în procesoarele RISC (Reduced Instruction Set Computing) si CISC (Complex Instruction Set Computing). Până acum am folosit Ripes având în spate o simulare a unui proces într-un singur ciclu, în procesor putând exista la un anumit moment doar o singură instructiune. În continuare vom analiza un procesor în 5 "stage"-uri (primul dintre ele **5-stage processor w/o forwarding or hazard detection**) care este compus din următoarele unități:

Fetch (F): Se preia instructiunea din memorie.

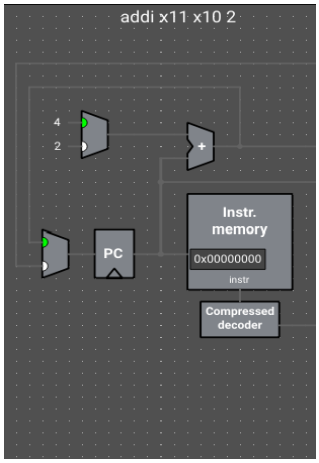
Decode (D): Se decodează instructiunea pentru a determina ce operatie trebuie executată.

Execute (E): Se execută operatia pe baza instructiunii folosind unitatea aritmetico-logică.

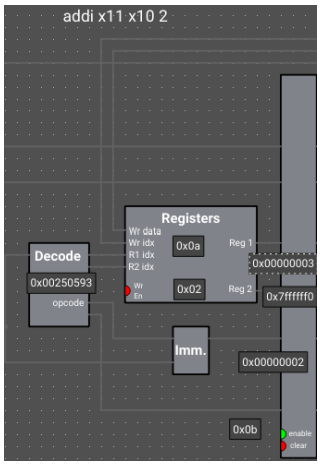
Memory (M): Citirea sau scrierea datelor în memoria principală, dacă instructiunea implică astfel de operatii.

Write Back (WB): Salvarea rezultatelor în registru sau memorie, pregătindu-le pentru instructiunile următoare.

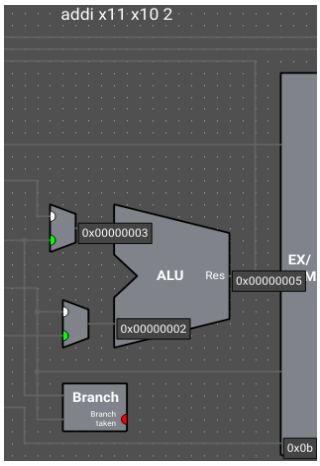
Să considerăm acum un program având o singură instructiune `addi a1, a0, 3`, iar `a0 = 2`. Să analizăm în continuare fiecare stage.



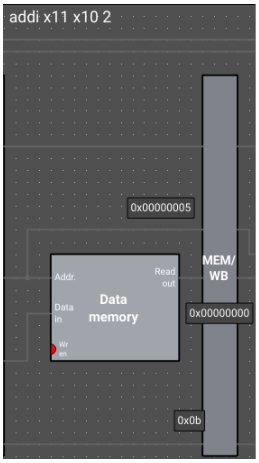
1. În etapa de **fetch**, prin PC (care este 0), este preluată instructiunea regăsită la acea adresă în memorie. Instructiunile comprimate sunt transformate în corespondentele lor necomprimate. Tot aici PC este incrementat cu 2 sau 4 pe baza acestei informatii.



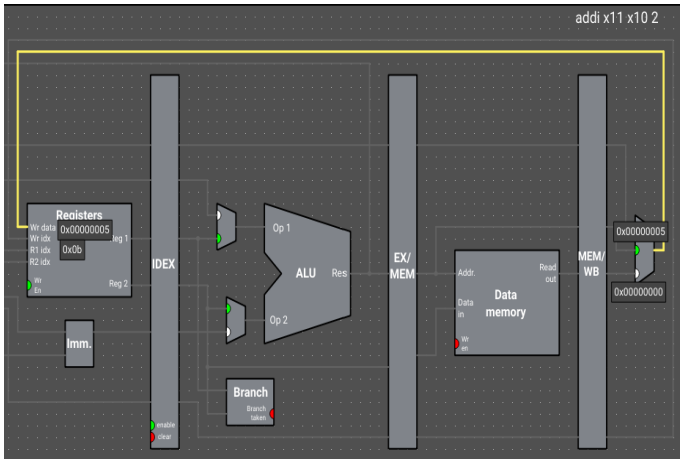
2. În etapa de **decode**, este interpretată encodarea si sunt preluate sursele si destinatia operatiei. Sunt transmise mai departe valorile curente ale celor 2 registri sursă (sau 0 dacă nu există un al doilea), respectiv valoarea imediată si encodarea pentru identificarea ulterioară a operatiei.



3. În etapa de **execute**, cele două valori sunt transmise către ALU unde se aplică operatia indicată de către opcode (adunarea cu imediat). Aceasta este o schemă simplificată, pentru a observa si ruta pe care se transmite tipul operatiei, puteti selecta o variantă de schemă extinsă.



4. În etapa de **memory**, se accesează datele din memorie (dacă ele există). După cum se poate observa, instructiunea aceasta nu foloseste nimic din memorie, deci se trece prin în această etapă doar transmitând mai departe rezultatul.



5. În etapa de **write back** se produce scrierea efectivă în destinatie - fie în memorie (în cazul nostru este 0, nu se accesează memoria), fie în register file, modificându-se valoarea registrului destinatie.

În ciuda avantajelor, performanta unui pipeline depinde de mai multi factori care pot introduce penalizări.

5.1 Hazard-uri ˆn pipeline

Un hazard este o problemă care împiedică procesorul să execute eficient instructiunile în pipeline. Există trei tipuri principale:

1. **Hazarduri de date** - Apar când o instructiune are nevoie de rezultatele unei alte instructiuni care nu a fost încă finalizată. Ca potentiale solutii remarcăm:
   (a) **Data forwarding:** Rezultatele din etapele intermediare sunt ”transmise înainte” pentru a fi utilizate fără a astepta finalizarea completă.
   (b) **Stalling (blocare):** Pipeline-ul este temporar oprit pentru a astepta disponibilitatea datelor.
   (c) **Out-of-order execution:** Un procesor OoO decuplează strict ordinea de preluare si decodificare a instructiunilor (care rămân in-order) de ordinea în care acestea sunt executate (care este out-of-order). Prin reorganizarea executiei instructiunilor, procesorul poate reduce latentele introduse de hazarduri. La final, instructiunile sunt mentinute într-un bu er, iar rezultatele lor sunt finalizate si scrise în registre în ordinea originală (in-order commit).
2. **Hazard-uri de control** - Apar în cazul instructiunilor de salt (branching), când procesorul nu stie ce instructiune să preia până când conditia saltului este evaluată. Pentru a elimina din penalităti putem folosi **Branch prediction** - prezicerea directia saltului si începerea executiei

instructiunilor prezise. Dacă predictia este gresită, instructiunile incorecte din pipeline sunt eliminate.

3. **Hazard-uri structurale** - Apar când două sau mai multe instructiuni încearcă să folosească aceeasi resursă hardware în acelasi timp (ex. o singură unitate de acces la memorie). Pentru a evita acest aspect, este nevoie ca proiectarea procesorului să se realizeze cu resurse hardware redundante.

Exercitiu: Considerati următoarea declaratie în memorie `.long 200, 300, 400`. Analizati pe rând pipeline-ul pentru instructiunile `lw a0, 4(gp)`, `sw a0, 8(gp)` respectiv `beq a0, a1, 12`.

Să încercăm în continuare să observăm pipeline-ul când avem un cod cu mai multe instructiuni:

```
.data
 .long 100, 200, 300
.text
.global main
main:
   lw a0, 4(gp)
   lw a1, 0(gp)
   add a0, a0, a1
   sw a0, 8(gp)
```

Exercitiu: Care este rezultatul depus în memorie? Ce s-a întâmplat?

Să observăm de asemenea codul următor:

```
.data
 .long 100, 200, 300
.text
.global main
main:
   lw a0, 0(gp)
   lw a1, 4(gp)
   blt a0, a1, label

   sw a0, 8(gp)
   j final

label:
   sw a1, 8(gp)

final:
   li a7, 93
   li a0, 0
   ecall
```

Exercitiu: Care este rezultatul depus în memorie? Ce s-a întâmplat?

Pentru rezolvarea acestor probleme, sunt oferite două variante îmbunătătite de procesor care abordează una dintre următoarele metode:

1. **Pipeline stall** (5-Stage processor w/o forwarding unit) - sunt introduse `nop`-uri (stall-uri) în conducta de date până când sursele sunt updatate.
2. **Forwarding/Bypassing** (5-stage processor w/o hazard detection) - sunt transmise datele din ALU direct către următoarea instructiune înainte de a fi scrise.

În cazul acesta însă cea de-a doua variantă nu este în continuare de folos, fiind necesar totusi mai multi cicli între accesări. Pentru eficientă si corectitudine, vom folosi deci procesorul care implementează ambele mecanisme (5-stage processor).

Exercitiu: Observati avantajul mecanismului de forwarding (5-Stage processor w/o forwarding unit vs 5-stage processor).

În plus, Ripes oferă si o schemă logică extinsă pentru un procesor superscalar - dual issue care permite intrarea simultană în pipeline a două instructiuni. În plus fată de procesorul anterior, acesta are 6 stage-uri, adăugându-se etapa **Issue** în care instructiunile sunt distribuite către unitătile functionale disponibile (ALU, unităti de acces la memorie).

5.2 Măsurarea performantei

1. **Throughput** - Măsoară câte instructiuni pot fi finalizate pe unitatea de timp. (5-stage processor - o instructiune, 6-stage dual-issue processor - 2 instructiuni).
2. **Latenta** - Timpul total necesar pentru a finaliza o singură instructiune (din Fetch până la Write Back - 5 vs 6).
3. **CPI (Cycles Per Instruction)** - numărul mediu de cicluri necesare pentru a finaliza o instructiune. Într-un procesor pipelined, CPI ideal = 1, dar în realitate poate fi mai mare din cauza hazard-urilor.
4. **IPC (Instructions Per Cycle)** - numărul efectiv de instructiuni executate pe ciclu de ceas. Un procesor superscalar (dual-issue sau quad-issue) poate avea un IPC mai mare de 1 ($IPC = 1/CPI$).
5. **Utilizarea pipeline-ului** - procentul de timp în care pipeline-ul este complet ocupat cu sarcini utile. Idle time (timp de inactivitate) înseamnă că resursele nu sunt utilizate eficient.

Exemplu: Un program execută un număr egal de instructiuni aritmetice, de accesare a memoriei si cu floating point. Stim că o instructiune aritmetică ia 4 cicli, una de accesare a memoriei 5, iar una care accesează unitatea de floating point 6. Cât este **CPI**? Dar **IPC**? **Observatie:** Acest tip de calcul ignoră multe alte variabile.

Răspuns:

$$CPI = 0\ 33 \quad 4 + 0\ 33 \quad 5 + 0\ 33 \quad 6 = 4\ 95$$

$$IPC = 1/4\ 95 = 0\ 2020$$

6 Cache

Memoria cache este un tip de memorie rapidă utilizată pentru a creste performanta procesorului, reducând timpul necesar pentru a accesa datele frecvent utilizate din memoria principală (RAM). Cache-ul este amplasat de obicei în apropierea procesorului si joacă un rol crucial într-un sistem de calcul modern.

Cache-ul functionează ca o memorie intermediară între procesor si memoria principală, stocând datele utilizate recent sau predictii de date ce vor fi necesare în viitor. Prin aceasta, el reduce latenta accesului la memorie si sporeste eficienta executiei instructiunilor.

Cache-ul exploatează principiile **localității spatiale** (datele din apropierea unei locatii utilizate recent sunt probabil necesare în viitor) si **localității temporale** (datele accesate recent sunt probabil utilizate din nou în scurt timp).

Exercitiu: Considerăm un sistem de calcul pe 32 de biti care are un cache de 16 Kbytes. Considerăm că memoriile sunt împărtite în blocuri de 32 de bytes. Câte blocuri sunt posibile în memoria principală? Câte blocuri sunt în memoria cache?

Memoria cache este organizată în mai multe niveluri ierarhice, fiecare având caracteristici distincte (L0, L1, L2 etc.). Dispozitivele cu capabilități reduse pot să nu aibă deloc sau să aibă mai putine niveluri de memorie cache. Simulatorul din Ripes de exemplu are doar memorie cache L1 pe care o vom folosi pentru a descoperi principalele caracteristici ale unei astfel de memorii.

O să începem prin a selecta o memorie cu 32 de linii sau blocuri si cu o dimensiune a liniei de 4 word-uri (16 bytes), direct-mapped (prima variantă preset). Să urmărim mai întâi algoritmul pentru descoperirea corespondentei dintre o adresă din memoria principală si locatia din cache.

Să considerăm codul de mai jos:

```
.data
 .long 0x10, 0x11, 0x12, 0x13
 .long 0x14, 0x15, 0x16, 0x17
 .long 0x18, 0x19, 0x20, 0x21
.text
.global main
main:
   lw a0, 20(gp)      #0x10000014  --> 0x15
   lw a1, 24(gp)      #0x10000018  --> 0x16
   add a0, a0, a1
   sw a0, 28(gp)      #0x1000001c  --> 0x17
   lw a1, 0(gp)       #0x10000010  --> 0x10
   add a0, a0, a1
   sw a0, 32(gp)      #0x10000020  --> 0x18

final:
   li a7, 93
   li a0, 0
   ecall
```

Exercitiu: Vom considera `gp = 0x10000000`, valoarea initială indicată de Ripes. Observati pozitionarea în memoria cache. Puteti deduce regula?

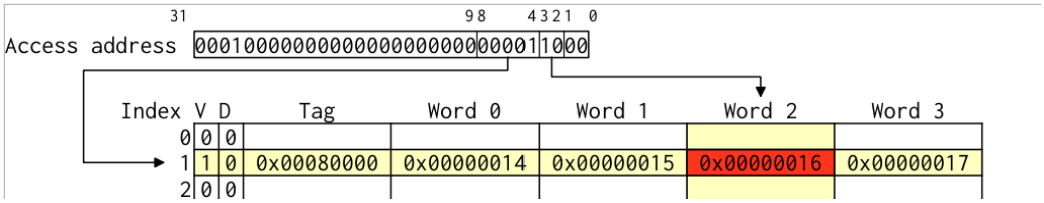


Figura 4: Linie de cache din Ripes

Algoritm

1. Unitatea de bază este word-ul, stim asadar ca ultimii 2 biti vor fi mereu 0 (adresele sunt din 4 în 4).

2. Extragerea o set-ului - Se determină câti biti sunt rezervati pentru o set pe baza dimensiunii blocului. În cazul nostru, o linie contine 16 bytes, 4 word-uri, deci o setii vor lua valori de la 0 la 3 - avem nevoie de încă 2 biti.

3. Extragerea indexului - Indexul indica linia/blocul din cache, deci va depinde de numărul de linii/blocuri din cache. În acest caz, avem 32 de linii, deci avem nevoie de 5 biti pentru identificare. Acesta este folosit pentru a identifica locatia din cache în care s-ar putea afla adresa cerută din memorie.

4. Extragerea tag-ului - Tag-ul este reprezentat de restul de biti rămasi. Tag-ul este comparat cu tag-ul stocat în acea locatie din cache pentru a verifica dacă datele sunt cele căutate.

Exercitiu: Urmăriti cum se modifică memoria cache pe parcursul executiei programului. Care ar fi size-ul minim al unei linii pentru a avea cel mai bun hit rate?

Exercitiu: Care este corelatia dintre dimensiunea cache-ului ca număr de linii, si dimensiunea unei linii pe de-o parte si conceptele de localitate temporală si spatială pe de cealaltă parte?

În plus fată de aceste concepte, putem modifica si **tehnica de mapare**:

Mapare directă - Fiecare bloc din memoria principală poate fi mapat doar într-o locatie specifică din cache. Simplu de implementat, dar poate duce la conflicte frecvente.

Mapare asociativă - Orice bloc din memoria principală poate fi mapat în orice locatie din cache. Este mai flexibil, dar implică un cost mai mare pentru căutare.

Mapare x-way set asociativă - Combinatie între maparea directă si cea asociativă, unde cache-ul este împărtit în seturi, iar un bloc din memoria principală poate fi mapat într-un subset specific de locatii din cache.

Exercitiu: Schimbati tipul de mapare în mapare asociativă. Ce observati?

Să considerăm acum următorul program:

```
.data
 .long 0x10, 0x11, 0x12, 0x13
 .long 0x14, 0x15, 0x16, 0x17
 .long 0x18, 0x19, 0x20, 0x21
.text
.global main
main:
   lw a0, 8(gp)
   lw a1, 12(gp)
   add a2, a0, a1
   lw a0, 24(gp)
   lw a1, 28(gp)
   add a3, a0, a1
   lw a0, 40(gp)
   lw a1, 44(gp)
   add a4, a0, a1
final:
   li a7, 93
   li a0, 0
   ecall
```

Exercitiu: Să considerăm acum o memorie cache 2-way set associative cu un line size de 2 si cu doar 2 linii de cache. Ce observati că se întâmplă?

Când cache-ul este plin, procesorul trebuie să decidă de asemenea ce bloc să înlocuiască. Politicile obisnuite includ:

LRU (Least Recently Used) - Se înlocuieste blocul care nu a fost folosit de cel mai mult timp.

FIFO (First In First Out) - Se înlocuieste blocul care a fost adăugat primul.

Random - Blocul înlocuit este ales aleatoriu.

O altă caracteristică a cache-ului (din păcate nefunctională pe Ripes) este politica de scriere:

1. **Write-through** - Procesorul modifică datele în blocul corespunzător din cache, iar modificarea este simultan scrisă si în memoria principală.
2. **Write-back** - Procesorul scrie datele doar în blocul din cache si setează un indicator numit bitul dirty (dirty bit). Dacă blocul trebuie înlocuit din cache, datele sunt scrise în memoria principală dacă acest bit este setat.

O altă politică aditională pentru scriere este **(No) Write Allocate** - decizia de a aduce blocul în memoria cache si la scriere sau nu.

6.1 Performanta memoriei cache

Performanta cache-ului este determinată de:

1. **Rata de accesare (hit rate)** - Procentajul accesărilor care găsesc datele necesare în cache. Rata ridicată de accesare reduce semnificativ latenta memoriei.

2. **Latenta memoriei** - Timpul necesar pentru a accesa datele. Latenta este minimă pentru L1 si creste pentru L2, L3 si memoria principală.

3. **Penalty-ul ratării (miss penalty)** - Timpul suplimentar necesar pentru a obtine datele din memoria principală atunci când acestea nu sunt prezente în cache.

4. **Politici de preluare (prefetching)** - Procesorul poate prelua proactiv date care ar putea fi necesare în viitor, bazându-se pe modele de acces..

7 Exercitii

7.1 Laboratorul 9

1. Se dau doua numere naturale x si y. Să se scrie un program care sa realizeze interschimbarea lor.
2. Se dau 4 numere a, b, c si d. Să se calculeze rezultatul operatiei $(((a + b + 5) \quad 2) \ll c)|d)$ si să se afiseze pe ecran.
3. Să se indice encodarea pentru instructiunea `and t0, s1, a5`.

7.2 Laboratorul 10

1. Afisati numărul de divizori ai unui întreg stocat în memorie si apoi o listă cu acestia. ("Numărul 6 are 4 divizori: 1, 2, 3, 6").
2. Realizati o procedură `divizori(int x)` care realizează afisarea de mai sus si întoarce numărul de divizori.
3. Realizati o procedură `divizori_elemente(int *v, int n, int k)` care parcurge vectorul `v` cu `n` elemente si returnează numărul de elemente ce au exact `k` divizori. Afisati în `main` un mesaj de tipul "Sunt nr elemente cu exact k divizori". Folositi-vă de procedura din exercitiul anterior.

7.3 Laboratorul 11

1. Un procesor are un pipeline cu 5 stadii: Fetch, Decode, Execute, Memory, Write-back. Determinati câte cicluri sunt necesare pentru a executa 4 instructiuni dacă nu apar hazarduri. Ce impact ar avea un hazard de date între instructiunile 1 si 2 asupra ciclurilor totale de executie?
2. Să considerăm un procesor cu următoarea frecventă a instructiunilor si următoarele costuri:

   ALU pentru întregi - 50%, 1 ciclu

   Load - 20%, 5 cicli

   Store - 10%, 1 ciclu

   Branch - 20%, 2 cicli

   Calculati CPI si decideti care schimbare ar aduce cea mai bună performantă dintre următoarele:

   implementarea unui mecanism de branch prediction pentru a reduce costul branch-urilor la 1 ciclu;

   folosirea unui data cache care are reduce costul unui load la 3 cicli.

3. Dacă un procesor superscalar este introdus pentru a rezolva hazardurile de date, câte cicluri pot fi economisite pentru următoarea secventă de instructiuni:

```
ADD R1, R2, R3
SUB R4, R1, R5
MUL R6, R7, R8
```

4. Memoria principală are o dimensiune de 2^{16}, iar cache-ul are o capacitate de 1 KB, cu o dimensiune a blocului de 32 bytes. Calculati numărul total de blocuri din memoria principală. Calculati numărul de linii de cache. Determinati unde va fi mapată adresa 0x3A7F în cache.

5. Un cache asociativ setă cu 4 seturi si 2 linii per set utilizează politica LRU (Least Recently Used). Considerati următoarele adrese accesate: 0x0001, 0x0005, 0x0021, 0x0001, 0x0045, 0x0005. Explicati cum sunt organizate datele în cache după fiecare acces. Care bloc este înlocuit atunci când adresa 0x0045 este accesată?

8 Resurse suplimentare

`https://riscv.org/educational-resources/`

`https://www.allaboutcircuits.com/technical-articles/introductions-to-risc-v-instruction-set-understanding-this-open-instruction-set-architecture/`

`https://www.cse.iitd.ac.in/~srsarangi/archbook/chapters/riscv.pdf`

`https://passlab.github.io/CSE564/notes/lecture09_RISCV_Impl_pipeline.pdf`

`https://nsec.sjtu.edu.cn/data/MK.Computer.Organization.and.Design.4th.Edition.Oct.2011.pdf`