RISC-V

Teodora Nae

Cuprins

1 Introducere

Până acum, ne-am familiarizat cu arhitectura x86, o arhitectură de tip CISC (Complex Instruction Set Computer). În continuare, vom studia arhitectura RISC (Reduced Instruction Set Computer), aprofundând limbajul Assembly RISC-V.
Principalele diferențe dintre cele două arhitecturi sunt prezentate atât în cadrul cursului, cât și în cadrul laboratorului.

2 Instruction Set Architecture (ISA)

2.1 Regiștri

Tabelul cu regiștrii de uz general din RISC-V poate fi consultat în suportul de laborator.
Câteva precizări:

- După cum se menționează, registrul **pc** este echivalentul lui %eip din x86. În RISC-V, instrucțiunile au o lungime fixă de 4 bytes, prin urmare valoarea registrului pc va crește cu 4 la fiecare instrucțiune executată.
- Registrul **ra** este utilizat în proceduri și stochează adresa de retur.

- Registrul **sp** este utilizat în proceduri la fel ca %esp din x86.
- Registrul **gp** stochează adresa lui .data. Putem accesa variabilele declarate în această secțiune, relativ la gp.
- **s0** sau **fp** este utilizat în proceduri la fel ca %ebp din x86.
- Prin **a0** și **a1** se returnează valorile într-o procedură. Dacă dorim să returnăm mai mult de două valori, trebuie să foolosim vârful stivei.
- Registrul **zero** are valoarea 0.
- **a7** se utilizează pentru apeluri de sistem.
- **s1-s11** - regiștri saved.
- **t0-t6** - regiștri temporary.
- **a0-a7** - regiștri argumente.
- **tp** - thread pointer.

2.2 Instrucțiuni

Spre deosebire de x86, arhitectura RISC se bazează pe principiul load-store. În toate instrucțiunile folosim regiștri, iar, pentru interacțiunea cu memoria RAM, folosim load sau store, după cum urmează:

- **load** - din memorie în registru
- **store** - din registru în memorie

Instrucțiunile LOAD și STORE sunt sufixate **obligatoriu**!
Ordinea operanzilor diferă pentru cele două instrucțiuni: pentru **load** avem **lw dest, src**, iar pentru **store** avem **sw src, dest** . Sufixul variază în funcție de tipul de date cu care lucrăm.
De asemenea, există și instrucțiunea **move reg1, reg2** care este, de fapt, tradusă în **addi reg1, reg2, 0**. Aceste instrucțiuni care nu au o encodare proprie se numesc pseudoinstrucțiuni.

Principalele instrucțiuni pentru operații aritmetice și logice pot fi consultate aici.

Ultimele două instrucțiuni prezentate în tabel sunt lui (load upper immediate) și auipc (add upper immediate to program counter). În rezolvarea exercițiilor care presupun auipc, trebuie să ținem cont de faptul că pc se modifică la fiecare instrucțiune.
Spre exemplu, pentru următoarele instrucțiuni, considerând pc inițial egal cu 0:

```
auipc a0, 0x12345
lw t1, 0(gp)
auipc a1, 0x23456
```

Prima linie: se pune pe cei mai din stânga 20 de biți valoarea immediate, a0 = 0x12345 << 12, apoi i se adaugă valoarea lui pc, la momentul actual 0: a0 = 0x12345000
A doua linie: pc crește cu 4 (dimensiunea unei instrucțiuni) și se pune în t1 valoarea aflată la 0(gp). Acum, pc=4
A treia linie: din nou, pc crește cu 4. a1 = 0x23456000 << 12, apoi i se adaugă valoarea actuală a lui pc. La final, a1=0x23456008

Salturile se numesc **branch** în RISC-V. La fel ca în x86, există salturi condiționate și salturi necondiționate. Spre deosebire de x86, instrucțiunile de branch verifică atât condiția de salt, cât și efectuează saltul dacă aceasta este îndeplinită:

Salt	Operanzi	Denumire	Explicație
j	j label	jump	Echivalent cu jal x0, label
blt	blt r1, r2, imm	branch if less than	dacă $r1 < r2$, salt la pc+imm
bltu	blt r1, r2, imm	branch if less than unsigned	la fel, dar pe numere pozitive
ble	ble r1, r2, imm	branch if less than or equal to	dacă $r1 \leq r2$, salt la pc+imm
bleu	bleu r1, r2, imm	branch if less than or equal to unsigned	la fel, dar pe numere pozitive
bgt	bgt r1, r2, imm	branch if greater than	dacă $r1 > r2$, salt la pc+imm
bgtu	bgtu r1, r2, imm	branch if greater than unsigned	la fel, dar pe numere pozitive
bge	bge r1, r2, imm	branch if greater than or equal to	dacă $r1 \geq r2$, salt la pc+imm
bgeu	bgeu r1, r2, imm	branch if greater than or equal to unsigned	la fel, dar pe numere pozitive
beq	beq r1, r2, imm	branch if equal to	dacă $r1 = r2$, salt la pc+imm
bne	bne r1, r2, imm	branch if not equal to	dacă $r1 \neq r2$, salt la pc+imm
beqz	beqz r1, imm	branch if equal to zero	dacă $r1 = 0$, salt la pc+imm
bnez	bnez r1, imm	branch if not equal to zero	dacă $r1 \neq 0$, salt la pc+imm

La ce ne referim prin "unsigned"? Dacă sufixăm o instrucțiune de branch cu **u**, atunci nu se va ține cont de semnul operanzilor comparați, fiind mereu considerați pozitivi. Cu alte cuvinte, pentru următorul cod:

```
.data
    x:  .word -1
    y:  .word 0

.text
.global main
main:
    lw t1, x          # pune valoarea lui x in t1
    lw t2, y          # pune valoarea lui y in t2
    bltu t1, t2, et_exit #salt daca t1<t2, unsigned

    # alte instructiuni

et_exit:
    # instructiunile pentru incheierea programului
```

Saltul nu va avea loc, deoarece valoarea din t1, adică 0xFFFFFFFF, va fi considerată $2^{32} - 1$, care nu este mai mică decât valoarea din t2, adică 0.

Pseudoinstrucțiuni de salt:

Salt	Operanzi	Denumire	Explicație
jal	jal rd, imm	jump and link	pune în rd valoarea pc+4; salt la adresa pc+imm
jr	jr rd	jump register	salt la adresa reținută în rd
jalr	jalr rd, rs, imm	jump and link register	pune în rd valoarea pc+4; salt la adresa din rs+imm

2.3 Tipuri de date

Tip	Spațiul ocupat
byte	1 byte
halfword	2 bytes
word	4 bytes
doubleword	8 bytes
ascii	dimensiunea = numărul de caractere
asciz	dimensiunea = numărul de caractere+1
space	dimensiunea specificată, în bytes

Valorile immediate sunt stocate pe 20 de biți.

2.4 Metode de adresare a memoriei

Am văzut în suportul de laborator că putem accesa și folosi variabilele din secțiunea .data fără a fi nevoie de un nume, ci relativ la gp, adresa .data.
În ceea ce privește tablourile, le declarăm exact ca în x86, dar cu tipurile de date corespunzătoare.
În x86, pentru a pune adresa unui tablou v într-un registru și a-i putea accesa, ulterior, elementele, avem două opțiuni: fie **mov $v, %reg**, fie **lea v, %reg**. În RISC-V folosim instrucțiunea **la dest, v** (load address).

Nu există echivalent în RISC-V pentru modalitatea de adresare din x86 a(b, c, d). Este de preferat ca, la fiecare pas, elementul curent să se afle la adresa 0(reg), unde reg este registrul în care se află adresa tabloului, prin urmare să adunăm, la fiecare pas, dimensiunea în bytes a unui element (4 pentru word, de exemplu) la acel registru. Astfel, registrul va stoca adresa elementului curent.

Următorul program afișează elementele unui tablou:

```
.data
    v:  .word 1, 2, 3, 4
    n:  .word 4
    spatiu: .asciz "␣"
.text
.global main
```

```
main:
    la t0, v # adresa
    li t1, 0 # indice
    lw t3, n # numarul de elemente

et_loop:
    beq t1, t3, et_exit

    #print INT
    li a7, 1
    lw a0, 0(t0)
    ecall

    #print STRING
    li a7, 4
    la a0, spatiu
    ecall

    addi t0, t0, 4
    addi t1, t1, 1
    j et_loop

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

3 Proceduri în RISC-V

Convențiile de apel sunt aceleași ca în x86.

- Apelul, respectiv revenirea din apel, se realizează folosind call și ret
- Argumentele se încarcă în ordine inversă pe stivă
- Accesarea elementelor din cadrul de apel se face relativ la s0
- Regiștrii callee-saved trebuie restaurați, iar cei caller-saved nu trebuie restaurați
- ra nu este pus automat pe stivă, deci trebuie să îl punem noi
- În loc de **call proc**, putem apela procedura cu **jal proc** (jump and link)

Stivă	x86	RISC-V
push op	sub $4, %esp mov op, 0(%esp)	addi sp, sp, -4 sw op, 0(sp)
pop op	mov 0(%esp), op add $4, %esp	lw op, 0(sp) addi sp, sp 4
pop	add $4, %esp	addi sp, sp, 4

Exemplu: Procedură care primește un tablou și dimensiunea sa ca parametri și returnează suma elementelor:

```
.data
    v: .word 1, 2, 3, 4
    n: .word 4
    str: .asciz "Suma este "
    sum: .word 0

.text
.global main

main:              # sum_vec(&v, n)
    # push n
    lw t1, n
    addi sp, sp, -4
    sw t1, 0(sp)

    # push &v
    la t0, v
    addi sp, sp, -4
    sw t0, 0(sp)

    # call sum_vec
    jal sum_vec
    addi sp, sp, 8

    # sw a0, sum
    la t5, sum
    sw a0, 0(t5)

    j afis

sum_vec:
        # crearea cadrului de apel

        # push ra
        addi sp, sp, -4
        sw ra, 0(sp)

        # push s0
        addi sp, sp, -4
        sw s0, 0(sp)

        #s0 = sp
        addi s0, sp, 0

        # s2 = n sau s2 = 12(sp)
        lw s2, 12(sp)

        # s1 = &v sau s1 = 8(sp)
        lw s1, 8(sp)

        addi sp, sp, -8
```

```
        # push s2
        sw s2, 4(sp)
        # push s1
        sw s1, 0(sp)

        li a0, 0     #suma
        li t2, 0     #indicele
        #vom folosi registri callee-saved, pentru a ilustra necesitatea
           restaurarii lor
fun_loop:

        beq t2, s2, ies      # prin a0 va fi returnat rezultatul,
        lw t3, 0(s1)
        add a0, a0, t3       # deci actualizam a0 la fiecare pas
        addi s1, s1, 4
        addi t2, t2, 1       # cat timp mai avem elemente de adaugat
        j fun_loop

ies:
        lw s1, -8(s0)        # restaurarea registrilor callee-saved
        lw s2, -4(s0)
        lw ra, 4(s0)
        lw s0, 0(s0)
        addi sp, sp, 16
        jr ra                # echivalent cu ret

afis:
        li a7, 4             # afisare format
        la a0, str
        ecall

        li a7, 1             # afisare suma
        lw a0, sum
        ecall


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

4 Exerciții

4.1 Laboratorul 9

Găsiți exercițiul 1 aici.

```
.data
    x:      .word 2
    y:      .word 3
    aux:    .word 0       # Ripes nu are .space;
```

```
                        # putem lucra direct cu registrii,
                        # insa vom folosi o variabila auxiliara
                        # pentru acest exemplu
    spatiu: .asciz "␣"

.text
.global main
main:

    lw t1, x  # alternativ, lw t1, 0(gp)
    lw t2, y  # alternativ, lw t2, 4(gp)

    # modificam direct valoarea lui aux, nu este nevoie de load
    #instructiunea sw reg, nume_variabila
    #nu functioneaza pe Ripes,
    #deci o vom inlocui dupa cum urmeaza

    #sw t1, aux
    la t5, aux
    sw t1, 0(t5)      # aux = x

    #sw t2, x
    la t5, x
    sw t2, 0(t5)      # x = y

    lw t3, aux        # acum punem aux (cu valoarea lui x) in registru

    # sw t3, aux
    la t5, y         # y = aux
    sw t3, 0(t5)

    li a7, 1       # afisare x
    lw a0, x
    ecall


    li a7, 4       # afisare spatiu
    la a0, spatiu
    ecall


    li a7, 1       # afisare y
    lw a0, y
    ecall


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

4.2 Laboratorul 10

Găsiți exercițiul 1 aici.

```
.data
    n: .word 6
    contor: .word 0
    str1: .asciz "Numarul␣"      # pentru afisarea
    str2: .asciz "␣are␣"         # cu formatul cerut
    str3: .asciz "␣divizori:␣"
    lista: .word 0, 0, 0, 0, 0, 0, 0, 0, 0, 0  # simulam un .space 40,
       maxim 10 divizori
    spatiu: .asciz "␣"

.text
.global main

main:
    lw t0, n
    li t1, 1                    # parcurgem toate numerele de la 1 la n
                                # (vrem si divizorii improprii)
    li t2, 0                    # contorul pentru divizori
    la t3, lista                # lista divizorilor

calc_div:
    bgt t1, t0, afis            # indicele e mai mare decat numarul,
                                # putem sa afisam
    rem t4, t0, t1              # t4 = t0 % t1
    bnez t4, div_urm            # daca restul e nenul, t1 nu e divizor


    sw t1, 0(t3)                # altfel, punem divizorul in lista
    addi t3, t3, 4              # avansam in lista
    addi t2, t2, 1              # incrementam contorul

div_urm:
    addi t1, t1, 1              #trecem la urmatorul numar
    j calc_div

afis:

    li a7, 4                    # afisare "Numarul␣"
    la a0, str1
    ecall

    li a7, 1                    # afisare n
    lw a0, n
    ecall


    li a7, 4                    # afisare "␣are␣"
    la a0, str2
    ecall
```

```
    la t5, contor                # sw t2, contor
    sw t2, 0(t5)

    li a7, 1
    lw a0, contor                # afisare contor
    ecall


    li a7, 4                     # afisare "␣divizori:␣"
    la a0, str3
    ecall


    la t3, lista
    li t1, 0

afis_div:
    beq t1, t2, et_exit          # parcurgem lista

    li a7, 1                     # afisam divizorii
    lw a0, 0(t3)
    ecall

    li a7, 4
    la a0, spatiu
    ecall

    addi t3, t3, 4
    addi t1, t1, 1
    j afis_div


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