Sinteză laborator ASC

Limbajul Assembly x86 (Sintaxa AT&T) & RISC-V

Chiper Ștefan & Cocu Matei-Iulian

Cuprins

1 Preliminarii - Mașina virtuală & Git

1.1 Mașina virtuală / Windows Subsystem for Linux

Acest laborator va fi predat sub Linux, astfel că dacă nu aveți o distributie de Linux ca sistem principal de operare, va trebui să vă instalați cel puțin pe o mașină virtuală. Recomandarea este să vă luați o imagine de Ubuntu și să o instalați pe VMWare sau pe VirtualBox. Imaginea sistemului de operare o găsiți pe site-ul de la Ubuntu. Două tutoriale video excelente pentru instalarea mediului de lucru, împreună cu unele sugestii de setare pentru sistem le găsiți aici:

- Windows & Ubuntu (other Linux-based - Debian) OS: https://youtu.be/XGr6_QV5moU
- MacOS Apple Silicon (Mx Normal/Pro/Max): https://youtu.be/1kENnRk39J0

Daca nu ați mai folosit Linux, o investiție bună pentru viitorul vostru este un alt tutorial puțin mai detaliat.

1.2 Git

La acest curs, tot codul sursă scris de voi va fi integrat într-un sistem pentru controlul versiunilor. Aceasta este o practică standard în industria software. Munca din timpul laboratoarelor va fi salvată pe Git. În acest sens, se recomandă să vă faceți un cont pe Github, să vă creați un repository cu numele "Laborator ASC" sau similar, iar toate problemele pe care le lucrați să le aveți centralizate acolo.
Un tutorial video pentru instalarea Git pe sistemul vostru este disponibil online.

2 Sinteză Laborator 0x00 - Introducere

2.1 Concepte de Bază ale Assembly x86

Limbajul Assembly x86 este un limbaj de programare de nivel scăzut care permite controlul direct asupra procesorului (CPU). În acest laborator, folosim **sintaxa AT&T**, care este predominantă pe sistemele Unix.

Componentele principale ale limbajului sunt:

- **Registrii**: Mici locații de stocare rapide, direct în procesor (ex: `%eax`, `%ebx`).
- **Instrucțiuni**: Operații pe care procesorul le poate executa (ex: `mov`, `add`).
- **Flaguri (Indicatori)**: Biți speciali care rețin starea ultimei operații (ex: dacă rezultatul a fost zero).
- **Adresarea Memoriei**: Modul în care accesăm datele stocate în RAM.
- **Stiva (Stack)**: O zonă specială de memorie, folosită intens pentru apelul funcțiilor.
- **Întreruperi**: Modul în care cerem sistemului de operare să execute operații pentru noi (ex: afișarea pe ecran).

2.2 Registrii, Notația și Adresarea

2.2.1 Registrii de uz general

Pe o arhitectură de 32 de biți, registrii principali de uz general au prefixul "E" (Extended). De exemplu, registrul `%eax` (32 biți) conține registrul `%ax` (16 biți), care la rândul lui este împărțit în `%ah` (High) și `%al` (Low), fiecare pe 8 biți.

Iată principalii registri și scopul lor convențional:

2.2.2 Registrul EFLAGS (Indicatori)

Acest registru reține starea programului. Câteva flag-uri importante sunt:

- **CF (Carry Flag)**: Setat dacă a avut loc transport (la adunare/scădere).
- **ZF (Zero Flag)**: Setat dacă rezultatul ultimei operații a fost 0.
- **SF (Sign Flag)**: Setat dacă rezultatul a fost negativ (bitul cel mai semnificativ e 1).

Registrul	Nume	Descriere	Restaurat după apel
AX	accumulator	este utilizat în operații aritmetice	nu
BX	base	utilizat ca pointer la date	da
CX	counter	este utilizat în instrucțiunile repetitive	nu
DX	data	este utilizat în operații aritmetice și I/O	nu
SP	stack pointer	pointer la vârful stivei	da
BP	base pointer	pointer la baza stivei	da
SI	source index	pointer la sursă în operații stream	da
DI	destination index	pointer la destinație în operații stream	da

- **OF (Overflow Flag)**: Setat dacă a avut loc o depășire *cu semn* (overflow).

2.2.3 Notație și Adresare

În sintaxa AT&T, regulile de bază sunt:

1. **Registrii** sunt prefixați de `%` (ex: `%eax`, `%ebx`).
2. **Constantele** (valorile imediate) sunt prefixate de `$` (ex: `$5`, `$0x80`).
3. **Adresarea Memoriei** folosește paranteze. Forma generală complexă este `offset(%baza, %index, multiplicator)`.
   - Exemplu: `4(%eax, %edx, 4)` înseamnă adresa calculată ca `%eax + (%edx * 4) + 4`.

2.3 Structura Programului și Tipurile de Date

2.3.1 Tipurile de Date

Când declarăm variabile în memorie, specificăm tipul lor:

- `.byte`: 1 byte (8 biți)
- `.word`: 2 bytes (16 biți)
- `.long`: 4 bytes (32 biți)
- `.quad`: 8 bytes (64 biți)
- `.ascii`: Șir de caractere *fără* terminator nul.
- `.asciz`: Șir de caractere *cu* terminator nul (`\0`).
- `.space N`: Rezervă `N` bytes de spațiu neinițializat.

2.3.2 Structura de Bază

Un program Assembly are, în general, două secțiuni principale:

1. `.data`: Aici se declară variabilele inițializate.
2. `.text`: Aici se scrie codul (instrucțiunile).

```
.data
    x: .long 15
    str: .asciz "String"

.text
.global main
main:
    # aici vin instructiunile
    # [...]
```

2.4 Instrucțiuni Fundamentale și Apeluri de Sistem

2.4.1 Instrucțiunea `mov`

Este instrucțiunea de bază pentru a muta date. Formatul AT&T este: `mov Sursa, Destinatia`

Exemplu: `movl $16, %eax` încarcă valoarea constantă `16` în registrul `%eax`. (Sufixul `l` vine de la `.long`).

2.4.2 Apeluri de Sistem (System Calls)

Pentru a interacționa cu sistemul de operare, folosim întreruperea `int $0x80`.

- `%eax`: Deține *codul funcției* pe care vrem să o apelăm.
- `%ebx, %ecx, %edx...`: Dețin *argumentele* funcției.

Exemplu 1: Oprirea programului (SYSTEM_EXIT)

- Cod funcție: `1` (pentru `exit`)
- Argument 1 (`%ebx`): Codul de retur (de obicei `0`)

```
movl $1, %eax    # codul pentru sys_exit
movl $0, %ebx    # codul de retur 0
int $0x80        # Apelez sistemul
```

Exemplu 2: Afișare pe ecran (SYSTEM_WRITE)

- Cod funcție: `4` (pentru `write`)
- Argument 1 (`%ebx`): unde scriem (`1` pentru `stdout` - consola)
- Argument 2 (`%ecx`): adresa de memorie a mesajului
- Argument 3 (`%edx`): lungimea mesajului

```
.data
    helloWorld: .asciz "Hello, world!\n"
    # "Hello, world!\n" are 15 caractere
.text
.global main
main:
    movl $4, %eax           # codul pentru sys_write
    movl $1, %ebx           # file descriptor 1 (stdout)
    movl $helloWorld, %ecx  # adresa mesajului
    movl $15, %edx          # lungimea mesajului
    int $0x80               # apelez sistemul

    # ... codul pentru exit ...
```

2.5 Operații Aritmetice și Logice

2.5.1 Operații Aritmetice

Instrucțiune	Efect
`add op1, op2`	`op2 = op2 + op1`
`sub op1, op2`	`op2 = op2 - op1`
`mul op`	`(EDX, EAX) = EAX * op` (fără semn)
`imul op`	`(EDX, EAX) = EAX * op` (cu semn)
`div op`	`(EDX, EAX) = (EDX, EAX) / op` (fără semn)
`idiv op`	`(EDX, EAX) = (EDX, EAX) / op` (cu semn)

Important la `imul` și `idiv`:

- Presupun *implicit* că operandul sursă este în `%eax`.
- **Înmulțirea**: Rezultatul pe 64 de biți este stocat în perechea `EDX:EAX`.
- **Împărțirea**: Deîmpărțitul pe 64 de biți este citit din `EDX:EAX`. Câtul este stocat în `%eax`, iar restul în `%edx`.

Instrucțiune	Efect
`not op`	`op = ~op` (NOT pe biți)
`and op1, op2`	`op2 = op2 & op1` (ȘI pe biți)
`or op1, op2`	`op2 = op2 | op1` (SAU pe biți)
`xor op1, op2`	`op2 = op2 ^ op1` (SAU Exclusiv pe biți)

Instrucțiune	Efect
`shl numar, op`	`op = op « numar` (shift logic stânga)
`shr numar, op`	`op = op » numar` (shift logic dreapta)
`sal numar, op`	`op = op « numar` (shift aritmetic stânga)
`sar numar, op`	`op = op » numar` (shift aritmetic dreapta, *păstrează bitul de semn*)

2.5.2 Operații Logice

2.5.3 Operații de Deplasare (Shift)

2.6 Exemple pentru Concepte "Dificile"

2.6.1 Exemplu 1: Împărțirea cu `idiv` și `cdq`

Instrucțiunea `idiv` împarte numărul de 64 de biți din `EDX:EAX` la operand. Pentru a împărți un număr de 32 de biți din `%eax`, trebuie să extindem bitul de semn al acestuia în `%edx`. Instrucțiunea `cdq` (Convert Double to Quad) face exact acest lucru.

```
.data
    x: .long -45
    y: .long 7
.text
.global main
main:
    movl x, %eax     # EAX = -45
    movl y, %ebx     # EBX = 7

    # pregatirea pentru impartire
    cdq              # extinde EAX (-45) in EDX:EAX.
                     # EDX devine 0xFFFFFFFF (adica -1)

    idivl %ebx       # imparte EDX:EAX la EBX (7)

    # rezultat:
    # EAX = -6 (catul)
    # EDX = -3 (restul)

    # ... exit ...
```

2.6.2 Exemplu 2: `lea` vs. `mov`

Aceasta este o distincție crucială:

- `movl v, %eax`: Încarcă **valoarea** de la adresa `v` în `%eax`.
- `leal v, %eax`: Încarcă **adresa** lui `v` în `%eax`.

```
.data
    v: .long 10, 20, 30, 40, 50
.text
.global main
main:
    # --- varianta cu LEA (corecta si eficienta) ---
    leal v, %edi             # %edi = adresa de inceput a lui v (baza)
    movl $1, %ecx            # %ecx = indexul dorit (1)
```

```
    # adresare: (%baza, %index, multiplicator)
    # adresa calculata = %edi + %ecx * 4
    movl (%edi, %ecx, 4), %eax  # EAX = valoarea de la adresa (v + 1*4)
                                # EAX va fi 20

    # --- varianta cu MOV (incorecta pentru adresare) ---
    # movl v, %edi          # incorect: %edi ar fi 10 (valoarea primului element)

    # ... exit ...
```

2.7 Salturi și Structuri de Control (Bucle)

2.7.1 Salturi Condiționate

- `jmp eticheta`: Sare necondiționat.
- `cmp op1, op2`: Compară `op2` cu `op1` (setează flag-urile).
- `j<conditie> eticheta`: Sare dacă condiția e îndeplinită.

Operator (cu semn)	Descriere
`je`	jump if equal (egal)
`jne`	jump if not equal (diferit)
`jg`	jump if greater (op2 > 1)
`jl`	jump if less (op2 < op1)
`jge`	jump if greater or equal (op2 >= op1)
`jle`	jump if less or equal (op2 <= op1)

2.7.2 Simularea Buclelor

Metoda 1: Manuală (cu `cmp` și `jmp`)

```
    movl $0, %ecx      # %ecx = 0 (contor, i=0)
    movl $5, %ebx      # %ebx = 5 (limita, n=5)
etloop:
    cmp %ebx, %ecx     # compara %ecx cu %ebx (i cu n)
    jge etexit         # daca %ecx >= %ebx, iesi (if i >= n, exit)

    # ... corpul buclei ...

    addl $1, %ecx      # %ecx = %ecx + 1 (i++)
    jmp etloop         # sari inapoi la inceputul buclei
etexit:
    # ... continua programul ...
```

Metoda 2: Instrucțiunea `loop` Instrucțiunea `loop` folosește automat `%ecx` ca și contor. La fiecare pas, decrementează `%ecx` și sare la etichetă dacă `%ecx` nu este 0.

```
    movl $5, %ecx      # %ecx = 5 (seteaza contorul)
etloop:
    # ... corpul buclei ...

    loop etloop        # decrementeaza %ecx; daca %ecx != 0, sare la etloop

# ... continua programul ...
```

2.8 Inline Assembly în C

Putem insera cod Assembly direct într-un program C folosind directiva `__asm__`.

```c
#include <stdio.h>

int x = 1;

int main() {
    __asm__
    (
        "movl x, %eax;"    // %eax = x
        "addl $1, %eax;"   // %eax = %eax + 1
        "movl %eax, x;"    // x = %eax
    );

    printf("%d\n", x); // va afisa 2
    return 0;
}
```

3 Sinteză Laborator 0x01 - Stiva, Proceduri și Apeluri C

3.1 Stiva (Stack)

Stiva este o regiune dinamică a memoriei unui proces care funcționează pe principiul **LIFO (Last In, First Out)**. La fiecare apel de funcție, pe stivă se alocă spațiu pentru argumente, variabile locale și adresa de retur. Stiva crește spre adrese de memorie mai mici.

Operațiile de bază pe stivă sunt:

- `push operand`: Adaugă un operand pe stivă.
- `pop operand`: Scoate valoarea din vârful stivei și o stochează în operand.

Registrii principali pentru stivă sunt:

- `%esp (Stack Pointer)`: Indică mereu vârful stivei.
  - `pushl %eax`: Decrementează `%esp` cu 4 și salvează `%eax`.
  - `popl %eax`: Încarcă valoarea de la `%esp` în `%eax` și incrementează `%esp` cu 4.
- `%ebp (Base Pointer)`: Indică baza cadrului de stivă curent.

```
.data
x: .long 2
y: .long 1
z: .long 3
e: .space 4
.text
.global main
main:
    # Exemplu: Evaluare ((x+y)*(x-y)*(x+z))/(y+z) folosind stiva
    movl y, %eax
    addl z, %eax
    pushl %eax          # %esp: (y+z)

    movl x, %eax
    addl z, %eax
    pushl %eax          # %esp: (x+z), (y+z)

    movl x, %eax
    subl y, %eax
    pushl %eax          # %esp: (x-y), (x+z), (y+z)

    movl x, %eax
    addl y, %eax
    pushl %eax          # %esp: (x+y), (x-y), (x+z), (y+z)

    popl %eax           # eax = (x+y)
    popl %ebx           # ebx = (x-y)
    mull %ebx
    pushl %eax          # %esp: (x+y)*(x-y), (x+z), (y+z)

    popl %eax           # eax = (x+y)*(x-y)
    popl %ebx           # ebx = (x+z)
    mull %ebx
    pushl %eax          # %esp: (x+y)*(x-y)*(x+z), (y+z)

    popl %eax           # eax = numarator
    popl %ebx           # ebx = numitor (y+z)
    movl $0, %edx       # important pentru impartire
    divl %ebx           #
    pushl %eax          # %esp: rezultat_final

    popl e              # stocam rezultatul in var e

    movl $1, %eax
    xor %ebx, %ebx
    int $0x80
```

3.2 Proceduri în Limbaj de Asamblare

Procedurile (funcțiile) sunt blocuri de cod definite de etichete, care pot fi apelate.

3.2.1 Apelul și Revenirea

- `call eticheta_procedura`:
  1. Salvează pe stivă adresa instrucțiunii următoare (adresa de retur).
  2. Sare la `eticheta_procedura`.
- `ret`:
  1. Scoate adresa de retur din vârful stivei.
  2. Sare la acea adresă.

3.2.2 Argumentele Procedurii

Argumentele sunt transmise prin intermediul stivei. Convenția de apel C cere ca argumentele să fie puse pe stivă în **ordine inversă**.

```
# Apel C: proc(arg1, arg2, arg3)
# Cod Assembly:
pushl arg3
pushl arg2
pushl arg1
call proc
addl $12, %esp  # curata stiva (3 argumente * 4 bytes)
```

3.2.3 Crearea Cadrului de Apel (Stack Frame)

Se folosește `%ebp` (Base Pointer) ca referință stabilă.

Prologul (la începutul procedurii):

```
eticheta_proc:
    pushl %ebp        # 1. salveaza vechiul %ebp (al apelantului)
    movl %esp, %ebp   # 2. %ebp pointeaza acum la baza noului cadru
    # ... (aici se pot aloca variabile locale, ex: subl $8, %esp)
```

Epilogul (la sfârșitul procedurii, înainte de 'ret'):

```
    # ... (aici se elibereaza variabilele locale, ex: movl %ebp, %esp sau leave)
    popl %ebp         # 1. restaureaza vechiul %ebp (al apelantului)
    ret               # 2. revine la apelant
```

După prolog, accesul la date se face relativ la `%ebp`:

- `8(%ebp)`: Primul argument.
- `12(%ebp)`: Al doilea argument.
- `-4(%ebp)`: Prima variabilă locală.

```
# Procedura add(x, y, &s) rescrisa cu cadru de apel
add:
    pushl %ebp
    movl %esp, %ebp

    movl 8(%ebp), %eax    # %eax = x (primul argument)
    addl 12(%ebp), %eax   # %eax = x + y (al doilea argument)
    movl 16(%ebp), %ebx   # %ebx = &s (al treilea argument)
    movl %eax, (%ebx)     # s = %eax

    popl %ebp
    ret
```

3.2.4 Registrii Salvați (Caller vs. Callee)

- **Caller-saved** (`%eax`, `%ecx`, `%edx`): Apelantul este responsabil să salveze acești registri.
- **Callee-saved** (`%ebx`, `%esi`, `%edi`, `%ebp`, `%esp`): Procedura (callee) este responsabilă să salveze și să restaureze acești registri.

3.2.5 Returnarea Valorilor

Valorile sunt returnate, în ordine, prin registrii `%eax`, `%ecx` și `%edx`.

3.3 Apelul Funcțiilor din C

Putem apela funcții C standard (ca 'printf', 'scanf') dacă linkăm programul folosind 'gcc'.

3.3.1 Apelul Funcției `printf`

- Argumentele se pun pe stivă în ordine inversă.
- Stringul de format este pasat ca adresă (ex: `pushl $formatString`).
- Trebuie curățată stiva după apel (ex: `addl $8, %esp`).
- Necesar un apel la `fflush` pentru a forța afișarea.

```
.data
x: .long 23
formatString: .asciz "Numarul este: %ld\n"
.text
.global main
main:
    pushl x                 # arg2: valoarea
    pushl $formatString     # arg1: adresa formatului
    call printf             #
    addl $8, %esp           # curata 2 argumente (2 * 4 bytes)

    pushl $0                # argumentul pentru fflush (NULL)
    call fflush
    addl $4, %esp           # curata 1 argument

    # ... cod exit ...
```

3.3.2 Apelul Funcției `scanf`

- Similar cu 'printf', argumentele se pun pe stivă invers.
- **Diferența majoră:** Variabilele se pasează prin **adresă** (ex: `pushl $x`).

```
.data
x: .space 4                 # spatiu stocare rezultat
formatScanf: .asciz "%ld"   #
.text
.global main
main:
    pushl $x                # arg2: ADRESA lui x
    pushl $formatScanf      # arg1: adresa formatului
    call scanf              #
    addl $8, %esp           # curata 2 argumente

    # ... cod exit ...
```

3.4 Manipularea Numerelor în Virgulă Mobilă

3.4.1 FPU (Floating Point Unit)

- Folosește o stivă internă de 8 registri, `st(0)` - `st(7)`.

•Majoritatea operațiilor se fac pe `st(0)`.
•Linkare cu '-lm' pentru funcții matematice.
•Rezultatul funcțiilor C (bazate pe FPU) este preluat din `st(0)`.

Instrucțiune	Efect
`flds mem`	Încarcă un float (4 bytes) din memorie în `st(0)`.
`fstps mem`	Stochează `st(0)` în memorie ca float și eliberează `st(0)`.

3.4.2 SSE (Streaming SIMD Extensions)

•Folosește 8 registri dedicați, `%xmm0 - %xmm7`.
•Metoda modernă, preferată.
•Valorile returnate de funcții C ('float'/'double') sunt plasate în `%xmm0`.

Instrucțiune	Efect
`movss src, dest`	Mută un scalar float (`ex: 'movss mem, %xmm0'`).
`addss src, dest`	Adună scalari float (dest = dest + src).
`subss src, dest`	Scade scalari float (dest = dest - src).
`mulss src, dest`	Înmulțește scalari float (dest = dest * src).
`divss src, dest`	Împarte scalari float (dest = dest / src).
`cvtsi2ss reg, dest`	Convertește int (din registru) în float (în xmm).
`cvtss2sd src, dest`	Convertește float (din xmm) în double (în xmm).

Afișarea unui float cu `printf` Funcția 'printf' așteaptă 'float'-uri promovate la 'double' (8 bytes).

```
.data
logResult: .float 1.0
fs: .asciz "%f\n"
.text
et_printf:
    # Presupunem ca rezultatul de afisat e in logResult
    movss logResult, %xmm0      # incarca float in xmm0

    # Pregatire apel printf
    cvtss2sd %xmm0, %xmm0       # converteste float (xmm0) la double (xmm0)
    subl $12, %esp              # aloca spatiu: 8 pt double + 4 pt format
    movsd %xmm0, 4(%esp)        # pune double-ul pe stiva (la offset 4)
    movl $fs, 0(%esp)           # pune adresa formatului (la offset 0)

    call printf                 #
    addl $12, %esp              # curata stiva
```

4 Sinteză Laborator 0x02 - RISC-V

4.1 Introducere RISC-V

RISC-V este un set de instrucțiuni (ISA) bazat pe principii **RISC (Reduced Instruction Set Computing)**.

- **Open-source**: Fără taxe de licențiere.
- **Modular**: Are un set de bază minimalist (RV32I) și extensii opționale (M, A, F, D, C, V, B).

x86 (CISC)	RISC-V (RISC)
Instrucțiuni complexe, lungime variabilă (1-15 bytes).	Instrucțiuni simple, lungime fixă (4 bytes).
Decodare lentă și complicată.	Decodare rapidă și simplă, bună pentru pipeline.
Operații (ex. `addl`) pot accesa memoria.	Arhitectură **Load-Store**. Doar `lw/sw` accesează memoria.
Multe instrucțiuni specializate.	Set redus de instrucțiuni de bază + extensii.

4.2 Registrii RISC-V (RV32I)

Sunt 32 de registri de uz general (`x0 - x31`), fiecare de 32 de biți. `x0` este hardcodat la valoarea 0. Convenția de nume (ABI) este crucială:

Registru	Nume ABI	Rol
`x0`	`zero`	Hardcodat la 0
`x1`	`ra`	Adresa de retur
`x2`	`sp`	Stack Pointer
`x8`	`s0/fp`	Frame Pointer / Callee-saved
`x10-x17`	`a0-a7`	Argumente funcție / Valori retur
`x5-x7, x28-x31`	`t0-t6`	Registri temporari (Caller-saved)
`x8-x9, x18-x27`	`s0-s11`	Registri salvați (Callee-saved)

4.3 Arhitectura Load-Store

Operațiile aritmetice/logice se fac **doar** între registri. Accesul la memorie se face **doar** cu instrucțiuni 'load' și 'store'.

- `lw rd, offset(rs1)`: Load Word (4B) → rd = mem[rs1 + offset]
- `sw rs2, offset(rs1)`: Store Word (4B) → mem[rs1 + offset] = rs2
- `lb/sb` (1B, byte), `lh/sh` (2B, halfword)

4.4 Instrucțiuni de Bază și Pseudo-instrucțiuni

Pseudo-instrucțiunile sunt "scurtături" traduse de asamblor:

4.5 Apeluri de Sistem (`ecall`)

Se folosește instrucțiunea `ecall`.

- **Codul funcției** se pune în `a7` (`x17`).
- **Argumentele** se pun în `a0-a6` (`x10-x16`).
- **Valoarea de retur** se găsește în `a0` (`x10`).

Instrucțiune	Descriere
`add rd, rs1, rs2`	`rd = rs1 + rs2`
`sub rd, rs1, rs2`	`rd = rs1 - rs2`
`mul rd, rs1, rs2`	Înmulțire (Extensia M)
`div rd, rs1, rs2`	Împărțire (Extensia M)
`rem rd, rs1, rs2`	Rest (Extensia M)
`and/or/xor rd, rs1, rs2`	Operații logice pe biți
`addi rd, rs1, imm`	Adunare cu valoare imediată
`sll/srl rd, rs1, rs2`	Shift logic stânga/dreapta

Pseudo-Instrucțiune	Echivalent(e) Real(e)	Descriere
`li rd, imm`	`lui` / `addi`	Încarcă valoare imediată
`la rd, label`	`auipc` / `addi`	Încarcă adresă
`mv rd, rs`	`addi rd, rs, 0`	Mută (copiază) registru
`nop`	`addi x0, x0, 0`	No-operation
`j label`	`jal x0, label`	Salt necondiționat
`ret`	`jalr x0, ra, 0`	Retur din procedură

```
# exit(0) - Opreste programul (Ripes)
li a7, 93  # Codul pentru exit
li a0, 0   # Argumentul (0)
ecall
```

4.6 Salturi și Proceduri

- `call eticheta`: Este un alias pentru `jal ra, eticheta`. Salvează adresa următoarei instrucțiuni (PC+4) în `ra` și sare la `eticheta`.
- `ret`: Este un alias pentru `jalr x0, ra, 0`. Sare la adresa stocată în `ra`.
- `beq rs1, rs2, et`: Branch if Equal (salt la `et` dacă `rs1 == rs2`).
- `bne, blt, bge`: Branch if Not Equal, Less Than, Greater Than or Equal.

4.7 Cadrul de Apel RISC-V

Stiva este folosită pentru a salva registrii `ra` și `s0-s11` (dacă sunt modificați).

Prologul (la începutul procedurii):

```
proc:
    addi sp, sp, -12  # 1. aloca spatiu pe stiva (ex: 12 bytes)
    sw ra, 8(sp)      # 2. salveaza adresa de retur (ra)
    sw s0, 4(sp)      # 3. salveaza vechiul frame pointer (s0)
    sw s1, 0(sp)      # 4. salveaza alt registru s*
    addi s0, sp, 12   # 5. seteaza noul frame pointer (s0)
```

Epilogul (la sfârșitul procedurii):

```
    lw s1, 0(sp)      # 1. restaureaza s1
    lw s0, 4(sp)      # 2. restaureaza s0
    lw ra, 8(sp)      # 3. restaureaza adresa de retur
    addi sp, sp, 12   # 4. elibereaza spatiul de pe stiva
    ret               # 5. revine la apelant
```

4.8 Pipelining și Hazard-uri

Procesorul execută instrucțiunile în etape suprapuse pentru viteză.

- **Etape (5):** 1. Fetch, 2. Decode, 3. Execute, 4. Memory, 5. Write-Back.

- **Hazard-uri:** Probleme care opresc pipeline-ul.
  - **De Date**: O instrucțiune are nevoie de rezultatul uneia nefinalizate.
  - **De Control**: Un salt (branch) face adresa următoarei instrucțiuni necunoscută.
  - **Structurale**: Două instrucțiuni au nevoie de aceeași componentă hardware.
- **Soluții:**
  - **Stalling (nop)**: Introducerea unei "bule" (pauze) în pipeline.
  - **Forwarding (Bypassing)**: Trimite rezultatul din etapa Execute direct înapoi la intrarea ALU, ocolind scrierea în registru.
  - **Branch Prediction**: Procesorul "ghicește" dacă saltul va fi efectuat sau nu.

4.9 Memoria Cache

O memorie mică și rapidă între CPU și RAM, care stochează datele accesate frecvent.

- **Principiul Localității:**
  - **Temporală**: Datele accesate recent vor fi probabil accesate din nou.
  - **Spațială**: Datele aflate lângă o adresă accesată vor fi probabil accesate în curând.
- **Mapare Cache:**
  - **Direct-Mapped**: Fiecare bloc de memorie se poate duce într-un singur loc (linie) specific din cache.
  - **N-Way Set Associative**: Fiecare bloc se poate duce în oricare din cele N linii dintr-un set specific.
  - **Fully Associative**: Fiecare bloc se poate duce în orice linie din cache.
- **Politici de Înlocuire** (când cache-ul e plin):
  - **LRU** (Least Recently Used): Se înlocuiește blocul cel mai puțin folosit recent.
  - **FIFO** (First In First Out): Se înlocuiește blocul cel mai vechi.