1 x86 AT&T (Arhitectura CISC)

1.1 Registri și Notație

Registri GPR (32-bit): **EAX** (Accumulator), **EBX** (Base), **ECX** (Counter), **EDX** (Data), **ESP** (Stack Pointer), **EBP** (Base Pointer), **ESI** (Source Index), **EDI** (Destination Index). Părțile inferioare: **AX** (16-bit), **AH** (high 8-bit), **AL** (low 8-bit).

EFLAGS (Indicatori): **ZF** (Zero Flag): Setat dacă rezultatul e 0. **SF** (Sign Flag): Setat dacă rezultatul e negativ. **CF** (Carry Flag): Setat dacă a avut loc transport. **OF** (Overflow Flag): Setat la depășire cu semn.

Sintaxa AT&T: Sursa *înaintea* Destinației: `movl %eax, %ebx`

Prefixe Sintaxă: Registri: `%eax`. Valori imediate: `$10`. Etichete/Adrese: `et`/`$et`.

Adresare Memorie: `offset(%baza, %index, scalar)` → `%baza + %index * scalar + offset`

Tipuri Date: `.byte` (1B), `.word` (2B), `.long` (4B), `.ascii` (fără null), `.asciz` (cu null), `.space N` (N bytes).

1.2 Apeluri de Sistem (Linux 32-bit)

Apel (Interrupt): Se folosește `int $0x80`. Codul funcției → `%eax`. Argumente → `%ebx, %ecx, %edx`.

```
# sys_exit(0) - Opreste programul
movl $1, %eax
movl $0, %ebx
int $0x80
```

```
# sys_write(stdout, msg, len) - Afiseaza
movl $4, %eax
movl $1, %ebx
movl $msg_addr, %ecx
movl $msg_len, %edx
int $0x80
```

1.3 Stiva (Stack) și Proceduri

Stiva (LIFO): Crește spre adrese mici. `%esp` (Stack Pointer) indică vârful stivei. `%ebp` (Base Pointer) indică baza cadrului de stivă curent (pentru referință stabilă).

Instrucțiuni Stivă:

- `pushl src` (Doar `long` (4B) sau `word` (2B)). Decrementează `%esp` cu 4, apoi scrie `src` la `(%esp)`.
- `popl dest` (Doar `long` sau `word`). Citește valoarea de la `(%esp)` în `dest`, apoi incrementează `%esp` cu 4.

Apel Procedură:

- `call etic`: 1. Pune adresa instrucțiunii următoare (adresa de retur) pe stivă. 2. Sare la `etic`.
- `ret`: 1. Ia adresa de retur de pe stivă. 2. Sare la acea adresă.

Convenție Apel C (Argumente):

- Argumentele sunt puse pe stivă de *apelant*, în ordine **inversă**.
- *Apelantul* curăță stiva după apel (ex: `addl $8, %esp` pentru 2 argumente `long`).

```
# Prolog Procedura (Setup Stack Frame)
eticheta_proc:
    pushl %ebp       # 1. Salveaza vechiul %ebp
    movl %esp, %ebp  # 2. Seteaza noul %ebp la %esp curent
    subl $8, %esp    # 3. Aloca spatiu pt 2 var locale
```

```
# Epilog Procedura (Restore Stack Frame)
    movl %ebp, %esp  # 1. Elibereaza var locale (sau leave)
    popl %ebp        # 2. Restaureaza vechiul %ebp
    ret              # 3. Revine la apelant
```

Layout Cadru Stivă (după prolog): Accesul se face relativ la `%ebp`:

- `...`
- `12(%ebp)`: Al doilea argument
- `8(%ebp)`: Primul argument
- `4(%ebp)`: Adresa de retur (salvată de `call`)
- `0(%ebp)`: Vechiul `%ebp` (salvat de prolog) ← **%ebp**
- `-4(%ebp)`: Prima variabilă locală
- `-8(%ebp)`: A doua variabilă locală
- `...` ← **%esp**

Salvare Registri: **Caller-saved** (apelantul salvează dacă are nevoie de ele): `%eax, %ecx, %edx`. **Callee-saved** (apelatul salvează dacă le folosește): `%ebx, %esi, %edi`.

Returnare Valori: Valorile sunt returnate de procedură (callee) prin registri, în această ordine: **%eax** (prima valoare), **%ecx** (a doua), **%edx** (a treia).

1.4 FPU și SSE

FPU (Stack-based): Folosește stiva `st(0)..st(7)`. `flds mem` (load float). `fstps mem` (store float + pop).

SSE (Register-based): Folosește registri `%xmm0..%xmm7`. `movss s, d` (mută float). `addss s, d` (adună float). `cvtsi2ss i, f` (int → float). `cvtss2sd f, d` (float → double).

printf float: Argumentele float trebuie promovate la double (8B). Se folosește `cvtss2sd %xmm0, %xmm0` înainte de apel.

1.5 Instrucțiuni Aritmetice / Logice

Aritmetice: `addl s, d` (d=d+s), `subl s, d` (d=d-s). `imul src` (EDX:EAX = EAX * src). `idivl src` (EAX = cat, EDX = rest).

Logice / Shift: `andl s, d`, `orl s, d`, `xorl s, d`, `notl d`. `shll N, d`, `shrl N, d` (logic). `sall N, d`, `sarl N, d` (aritmetic).

1.6 Salturi Condiționate x86 (după `cmp op1, op2`)

Cu Semn	Descriere (op2 ? op1)
`je`	jump if equal (==)
`jne`	jump if not equal (!=)
`jg`	jump if greater (>)
`jl`	jump if less (<)
`jge`	jump if greater or equal (>=)
`jle`	jump if less or equal (<=)
Fără Semn	**Descriere (op2 ? op1)**
`ja`	jump if above (>)
`jb`	jump if below (<)
`jae`	jump if above or equal (>=)
`jbe`	jump if below or equal (<=)

1.7 Instrucțiuni x87 FPU

Instr.	Operanzi	Descriere
`flds`	mem	Încarcă **float** din memorie în `st(0)`
`fldl`	mem	Încarcă **double** din memorie în `st(0)`
`fstps`	mem	Stoc. **float** din `st(0)` în mem. și ex. pop
`fstpl`	mem	Stoc. **double** din `st(0)` în mem. și ex. pop
`sqrtf`	mem	`st(0) = sqrt(st(0))`
`logf`	mem	`st(0) = log(st(0))`
`expf`	mem	`st(0) = exp(st(0))`

1.8 Instrucțiuni x87 SSE

Instr.	Operanzi	Descriere
`movss`	src, dest	Mută un scalar **float** din src în dest
`movsd`	src, dest	Mută un scalar **double** din src în dest
`addss`	src, dest	`dest = dest + src` (**float**)
`subss`	src, dest	`dest = dest - src` (**float**)
`mulss`	src, dest	`dest = dest * src` (**float**)
`divss`	src, dest	`dest = dest / src` (**float**)
`sqrtss`	dest	`dest = sqrt(dest)` (**float**)
`cvtsi2ss`	src, dest	Convertește un **long** (src) în **float**
`cvtss2sd`	src, dest	Convertește un **float** (src) în **double**

2 RISC-V (Arhitectura RISC)

Filosofie: Arhitectură **open-source**, **modulară** (bază + extensii).

2.1 Registri (RV32I)

32 Registri GPR (x0-x31): Nume ABI (Convenție):

- `x0` (zero): Hardcodat la 0.
- `x1` (ra): Adresa de retur *(important: acest registru nu există în x86 - influențează)*.
- `x2` (sp): Stack Pointer.
- `x8` (s0/fp): Frame Pointer.
- `a0-a7` (x10-x17): Argumente / Valori retur.
- `s0-s11` (x8-9, x18-27): Callee-saved.
- `t0-t6` (x5-7, x28-31): Caller-saved.

2.2 Arhitectura Load-Store

Principiu: Operațiile (add, sub etc.) se fac **doar** între registri. Memoria se accesează **doar** cu instrucțiuni `load` și `store`.

Acces Memorie: `lw rd, offset(rs1)`: Load Word (4B). `sw rs2, offset(rs1)`: Store Word (4B). `lb/sb` (1B), `lh/sh` (2B).

Pseudo-Instrucțiuni: `li rd, imm` (Load Immediate). `la rd, eticheta` (Load Address). `mv rd, rs` (Move, echivalent `addi rd, rs, 0`). `nop` (No operation).

2.3 Apeluri de Sistem (ecall)

Apel (ecall): Se folosește instrucțiunea `ecall`. Codul funcției → `a7` (x17). Argumente → `a0-a6` (x10-x16).

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

```
# PrintString(a0=adresa) (Ripes)
la a0, eticheta_string
li a7, 4
ecall
```

2.4 Salturi și Proceduri

Salturi: `j eticheta` (Salt necondiționat, alias pt `jal x0, et`). `beq r1, r2, et` (Branch if equal). `bne`, `blt`, `bge` (Not equal, less than, greater/equal).

Apel Procedură: `call eticheta` (Alias pt `jal ra, eticheta`). Salvează PC+4 în `ra` (x1). `ret` (Alias pt `jalr x0, ra, 0`).

Argumente (Convenție): Primele 8 argumente → `a0-a7`. Restul → pe stivă. Valori retur → `a0, a1`.

```
# prolog procedura
addi sp, sp, -8    # aloca spatiu
sw ra, 4(sp)       # salveaza adresa retur
sw s0, 0(sp)       # salveaza frame pointer
addi s0, sp, 8     # seteaza noul fp
```

```
# epilog procedura
lw s0, 0(sp)       # restaureaza fp
lw ra, 4(sp)       # restaureaza adr retur
addi sp, sp, 8     # elibereaza spatiu
ret
```

2.5 Instrucțiuni RISC-V (RV32I/M)

Instr.	Operanzi	Descriere
`add/sub`	rd, rs1, rs2	rd = rs1 +/- rs2
`mul/div`	rd, rs1, rs2	Inmultire/Impartire (M)
`rem`	rd, rs1, rs2	Rest (M)
`and/or/xor`	rd, rs1, rs2	Logice pe biti
`addi`	rd, rs1, imm	rd = rs1 + imm
`slti`	rd, rs1, imm	rd = (rs1 < imm) ? 1 : 0
`lui`	rd, imm	rd = imm « 12
`auipc`	rd, imm	rd = pc + (imm « 12)

`mențiune:` registrul **pc** - program counter, este incrementat automat *(convențional, acesta are la începutul programului valoarea 0, și este incrementat cu 4 bytes)*, dar în cadrul instrucțiunilor de branch, își schimbă valoarea în funcție de context

2.6 Salturi Condiționate RISC-V

Instr.	Operanzi	Descriere (Salt dacă...)
`beq`	rs1, rs2, imm	rs1 == rs2
`bne`	rs1, rs2, imm	rs1 != rs2
`blt`	rs1, rs2, imm	rs1 < rs2 (cu semn)
`bge`	rs1, rs2, imm	rs1 >= rs2 (cu semn)
`bltu`	rs1, rs2, imm	rs1 < rs2 (fără semn)
`bgeu`	rs1, rs2, imm	rs1 >= rs2 (fără semn)

2.7 Instrucțiuni RVC (Comprimate)

Instr.	Operanzi	Descriere
`c.addi`	rd, imm	rd = rd + imm (imm != 0)
`c.lwsp`	rd, offset(sp)	Load Word (din stiva)
`c.swsp`	rd, offset(sp)	Store Word (pe stiva)
`c.j`	offset	Salt neconditionat
`c.beqz`	rs, offset	Salt daca rs == 0
`c.bnez`	rs, offset	Salt daca rs != 0
`c.lw/c.sw`	rd', off(rs')	Load/Store (pt registri s0-s1, a0-a5)

3 Arhitectură (Pipeline & Cache)

Pipeline 5 Etape: **F**etch (Ia instrucțiune). **D**ecode (Decodează, citește registri). **E**xecute (Calculează în ALU). **M**emory (Accesează memoria). **W**rite-Back (Scrie rezultatul în registru).

Hazard-uri: **De Date**: O instrucțiune are nevoie de rezultatul uneia nefinalizate. **De Control**: Salt (branch) - nu se știe următoarea instrucțiune. **Structurale**: Două instrucțiuni cer aceeași resursă hardware.

Soluții Hazard: **Stalling**: Introducere `nop` (bule) în pipeline. **Forwarding**: Rezultat ALU trimis direct la intrarea ALU, ocolind scrierea în registru. **Branch Prediction**: Ghicirea direcției saltului.

Localitate (Cache): **Temporală**: Date accesate recent vor fi accesate iar. **Spațială**: Date apropiate (ex. în array) vor fi accesate curând.

Mapare Cache: **Direct-Mapped**: Adresă → o singură linie specifică (index). **N-Way Set Associative**: Adresă → oricare din N linii dintr-un set. **Fully Associative**: Adresă → orice linie din cache.

Politici aplicate de procesor atunci când memoria cache este plină:

- **LRU (Least Recently Used)** - Se înlocuiește blocul care nu a fost folosit de cel mai mult timp
- **FIFO (First In First Out)** - Se înlocuiește blocul care a fost adăugat primul
- **Random** - Blocul înlocuit este ales *aleatoriu*

Performanța memoriei cache este determinată de:

1. **Rata de accesare (hit rate)** - Procentajul accesărilor care găsesc datele necesare în cache. Rata ridicată de accesare reduce semnificativ latența memoriei
2. **Latența memoriei** - Timpul necesar pentru a accesa datele. Latența este minimă pentru L1, crește pentru L2, L3 și RAM *(memoria principală)*
3. **Penalty-ul ratării (miss penalty)** - Timpul suplimentar necesar pentru a obține 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

31 2524	2019	1514	1211	76	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 1: RISC-V Instruction Formats