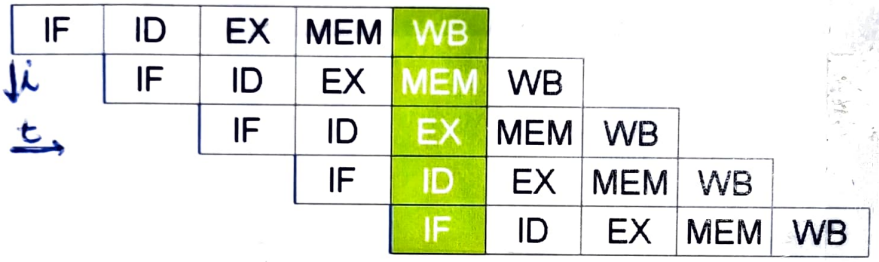
# CURS 9

- Mecanisme pt. execuții sporită:
  - pipelining
  - branch prediction
  - out of order execution
- Pipelining = un tip de paralelism la nivel de instrucțiune
- Pentru eficiență maximă:




IF = instruction fetch
ID = instruction decode
EX = execution
MEM = memory access
WB = write back

Cât timp dă decode la prima instrucțiune, se poate da fetch la următoarea instrucțiune ș.a.m.d.

- Însă, nu mereu funcționează și apar hazards / erori:
  - → structural hazards = două instrucțiuni încearcă să acceseze aceeași unitate <=> o unitate de calcul este deja utilizată
  - → data hazards = o instr. depinde de rezultatul unei instr. anterioare <=> datele nu sunt gata de utl.
  - → control hazards = nu se știe urm. instr <=> din cauza unor instr. „jump" nu se știe urm. instr.

- Pentru data hazards:
  - → True dependence (Read After Write - RAW)

    add %ebx, %eax

    sub %eax, %ecx
  - → Anti dependence (Write After Read - WAR)

    add %ebx, %eax

    sub %ecx, %ebx
  - → Output dependence (Write After Write - WAW)

    mov $0x10, %eax

    mov $0x01, %eax
- Instrucțiuni diferite au nevoie de nr. dif de cicli de CPU

| operația | instrucțiuni | # cicli |
|---|---|---|
| operații întregi/biți | add, sub, and, or, xor, sar, sal, lea, etc. | 1 |
| înmulțirea întregilor | mul, imul | 3 |
| împărțirea întregilor | div, idiv | depinde (20–80) |
| adunare floating point | addss, addsd | 3 |
| înmulțire floating point | mulss, mulsd | 5 |
| împărțire floating point | divss, divsd | depinde (20–80) |
| fused-multiply-add floating point | vfmass, vfmasd | 5 |

ce facem în aceste situații?

- În procesoarele moderne se citesc sute de instr. la un nr. dat iar în hardware se realizează un data-flow graph de dependențe între instr.
- Graful se execută eficient cu out of order execution (execuție în ordine aleatoare)

quad (a, b, c)

$t_1 = a * c$ ; $t_3 = b * b$ ; $t_6 = -b$ ; $t_9 = 2 * a$

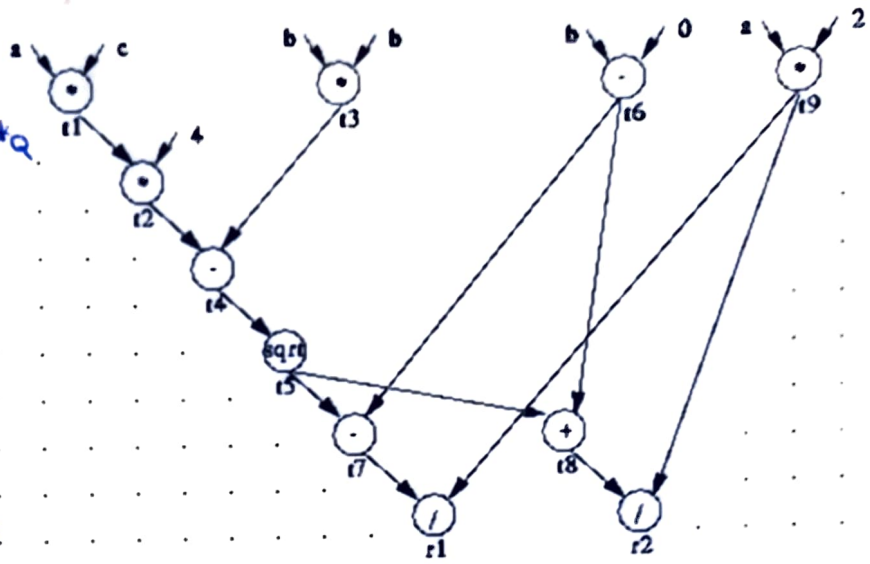
$t_2 = 4 * t_1$

$t_4 = t_3 - t_2$

$t_5 = sqrt(t_4)$

$t_7 = t_6 - t_5$ ; $t_8 = t_6 + t_5$

$r_1 = t_7 / t_9$ ; $r_2 = t_8 / t_9$




- O soluție a codului este branch prediction

Codul are 2 porțiuțăți:

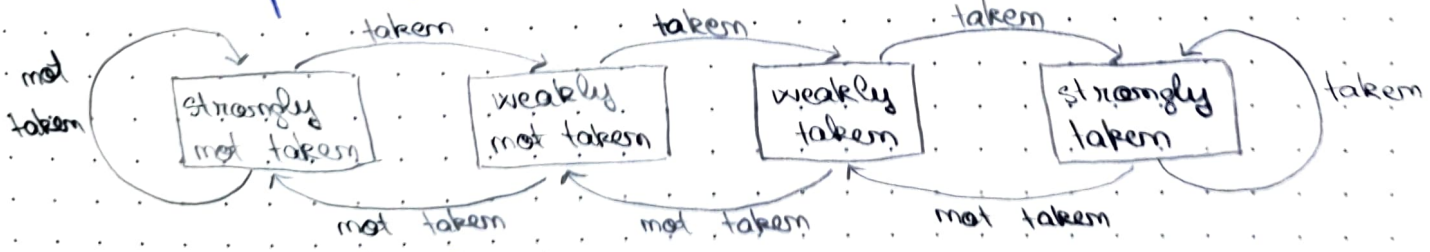
→ popq %rbx

→ movl %ebx, %edi

```
f2:
        pushq   %rbx
        xorl    %ebx, %ebx
.L3:
        movl    %ebx, %edi
        addl    $1, %ebx
        call    callfunc
        cmpl    $10, %ebx
        jne     .L3
        popq    %rbx
        ret
```

Predicțiile sunt:

- predicție fixă: mereu sare sau nu
- predicția de la pasul anterior: ce a făcut instr. ultima dată
- predicția cu istoric: ce face deobicei instr.




counter 2 biți

- execuție speculativă (eager execution): calculează cu și fără salt