

ARHITECTURA SISTEMELOR DE CALCUL - CURS 0x08

**PIPELINING, BRANCH PREDICTION, OUT OF
ORDER EXECUTION**

Cristian Rusu

DATA TRE CUTĂ

- **Instruction Set Architecture (ISA)**
- **de la cod sursă la cod mașină**
 - sper că v-ați amintit demo-ul de la Cursul 0x00 unde am executat jocul Doom în browser
- **un exemplu simplu de *software cracking* și *shellcode execution***
- **dacă sunteți pasionați de securitate (binary exploitation):**
 - C, ASM (stiva, call-ul procedurilor)
 - cursul de SO (anul II, semestrul I)
 - J. Erickson, Hacking: The Art of Exploitation
 - exerciții:
 - protostar
 - Capture The Flag (CTF)

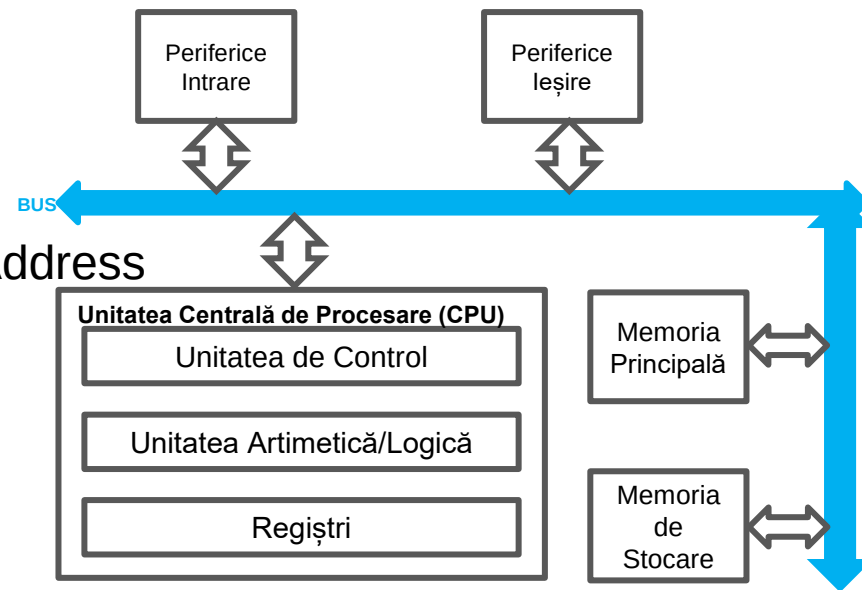
CUPRINS

- **trei mecanisme pentru execuție cu viteză sporită**
 - pipelining (conductă de date)
 - branch prediction (predicția salturilor)
 - out of order execution (execuția în ordine arbitrară)

ARHITECTURA DE BAZĂ

- **CPU execută instrucțiuni:**

- IF – Instruction Fetch
- ID – Instruction Decode
 - COA – Calculate Operand Address
 - FO – Fetch Operand
 - FOs – Fetch Operands
- EX – Execution
- MEM – Memory Access
- WB – Write Back



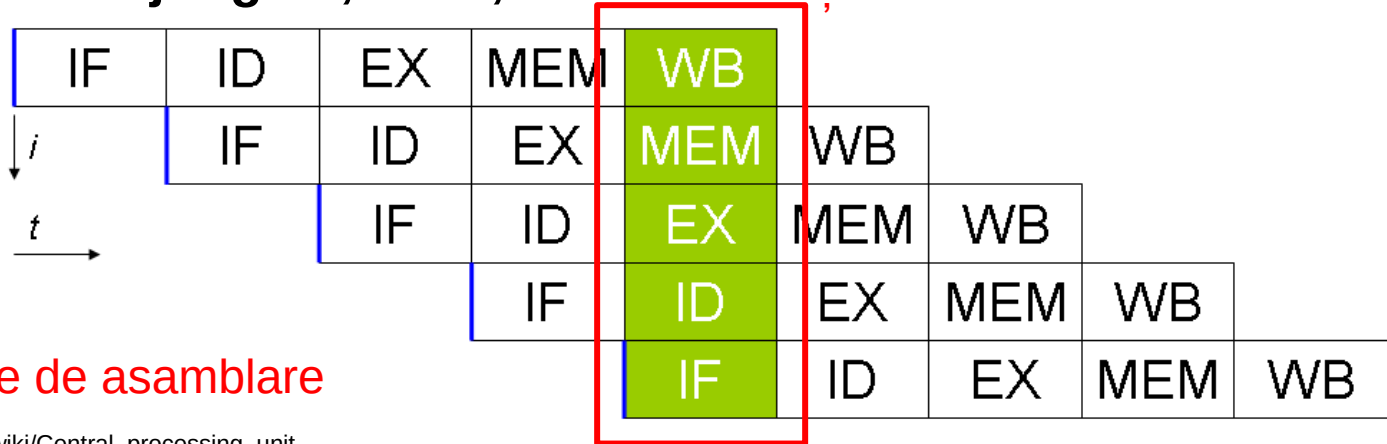
PIPELINING

- este un tip de paralelism la nivel de instrucțiune (Instruction Level Paralellism - ILP)
 - asta pentru că e paralelism diferit față de “task/process level paralellism” (thread-uri și procese)

• de la :

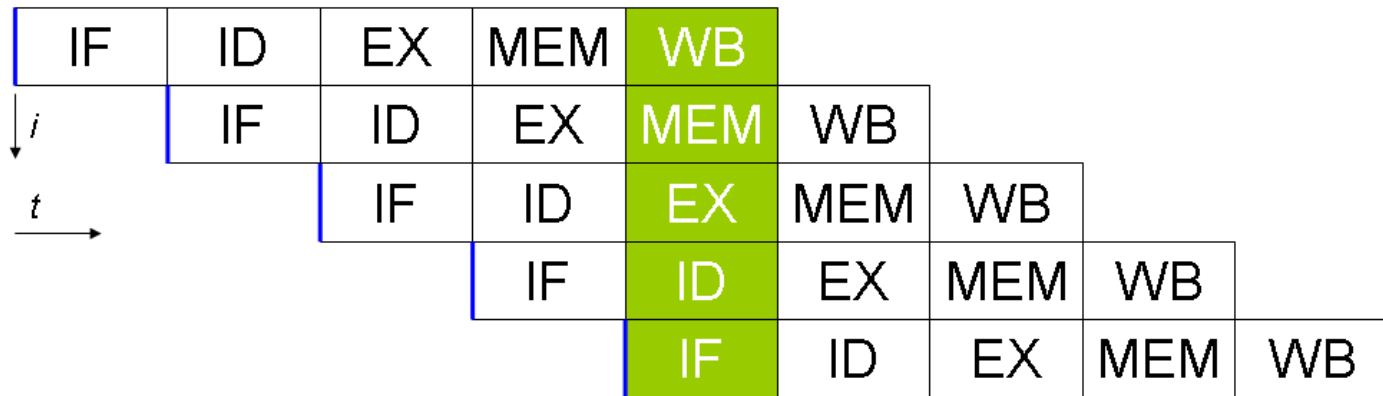


• vrem să ajungem, ideal, la: eficiență maximă



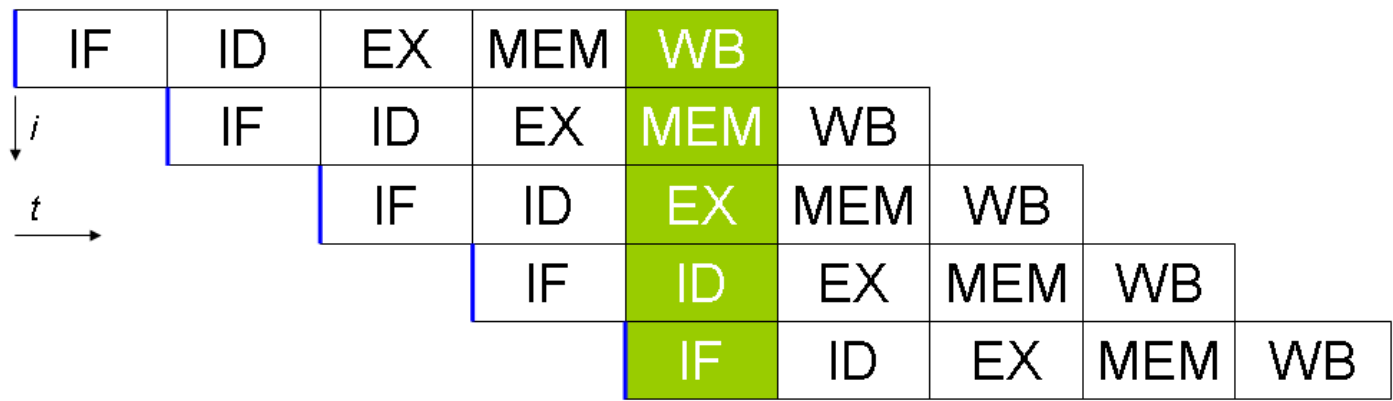
ideea: o linie de asamblare

PIPELINING

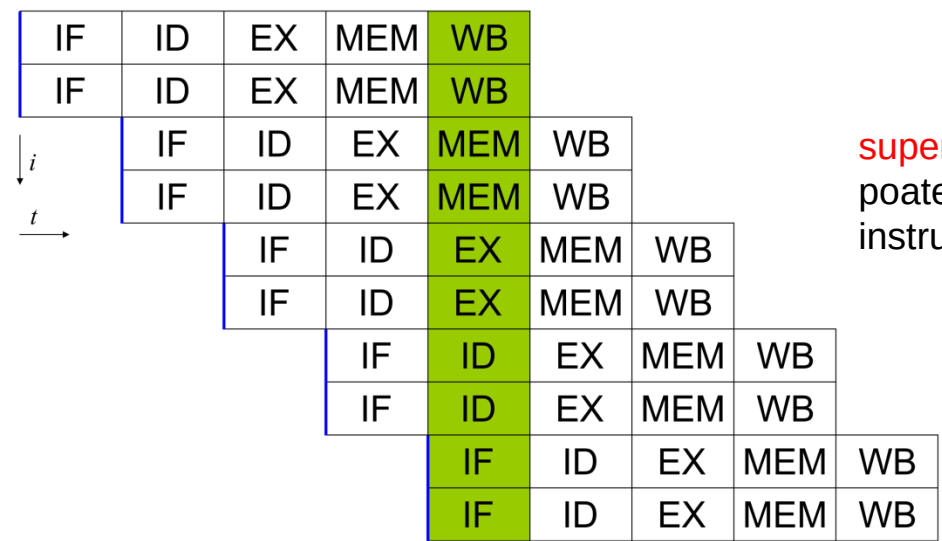


- **imaginea arată bine, din păcate nu putem face așa ceva mereu**
 - pipeline stalls (întârzieri în conducta de date)
 - aceste evenimente se numesc hazards (erori)
 - **structural hazards**: o unitate de calcul este deja utilizată
 - două instrucțiuni încearcă să acceseze aceeași unitate
 - **data hazards**: datele nu sunt pregătite pentru utilizare
 - o instrucțiune depinde de rezultatul unei instrucțiuni precedente
 - **control hazards**: nu știm următoarea instrucțiune
 - din cauza unor instrucțiuni de jump nu știm instrucțiunea următoare

PIPELINING: STRUCTURAL HAZARDS

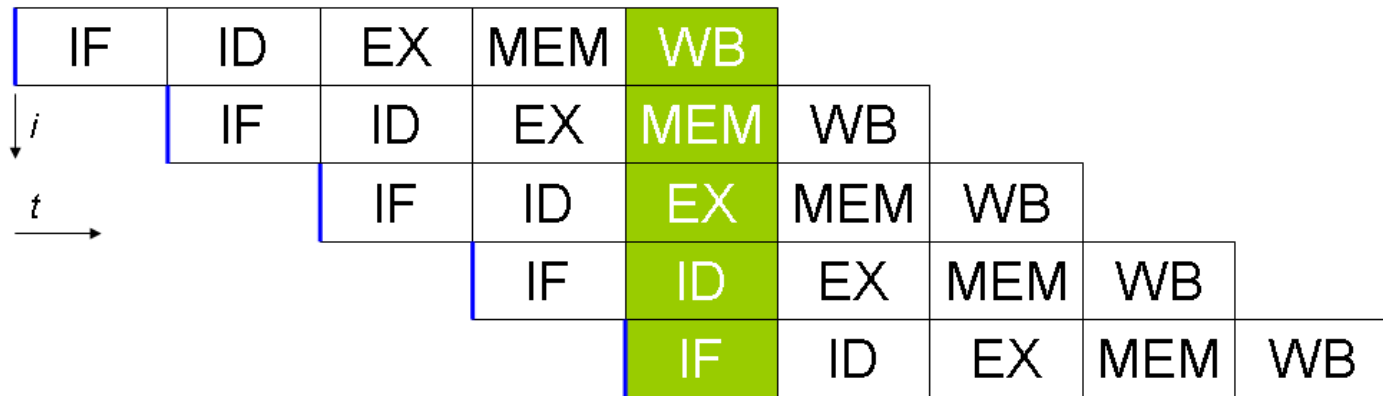


- pipeline stalls (întârzieri în conducta de date)
 - structural hazards
 - două instrucțiuni încearcă să acceseze aceeași unitate



superscalar, hardware dublat,
poate termina mai mult de o
instrucțiune într-un ciclu

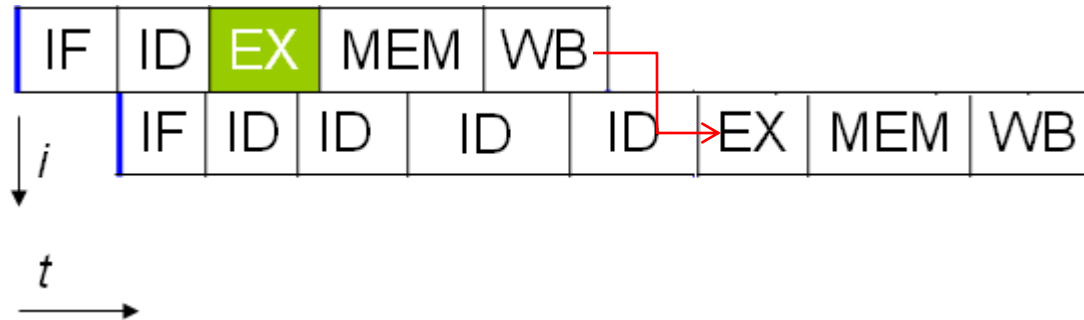
PIPELINING: DATA HAZARDS



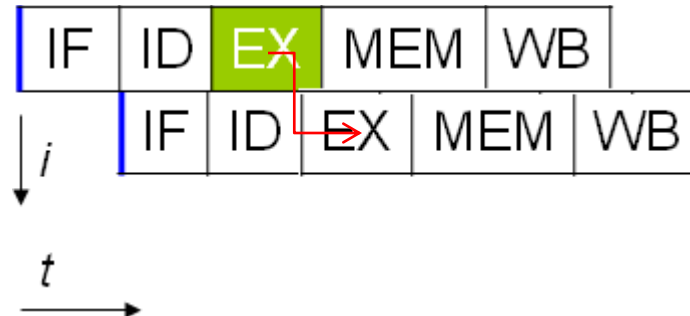
- pipeline stalls (întârzieri în conducta de date)
 - data hazards
 - o instrucțiune depinde de rezultatul unei instrucțiuni anterioare
 - **True Dependence (Read After Write – RAW)**
 - add %ebx, %eax
 - sub %eax, %ecx
 - **Anti-dependence (Write After Read – WAR)**
 - add %ebx, %eax
 - sub %ecx, %ebx
 - **Output Dependence (Write After Write – WAW)**
 - mov \$0x10, %eax
 - mov \$0x01, %eax

PIPELINING: DATA HAZARDS

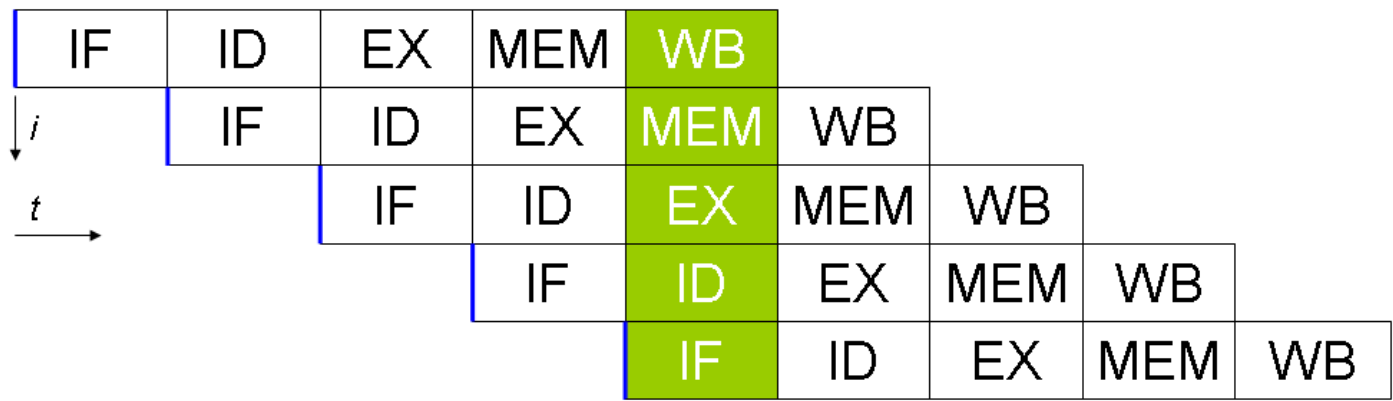
- pipeline stalls (întârzieri în conducta de date)
 - data hazards
 - True Dependence (Read After Write – RAW)
 - add %ebx, %eax
 - sub %eax, %ecx



- posibilă soluție *bypassing*



PIPELINING: DATA HAZARDS

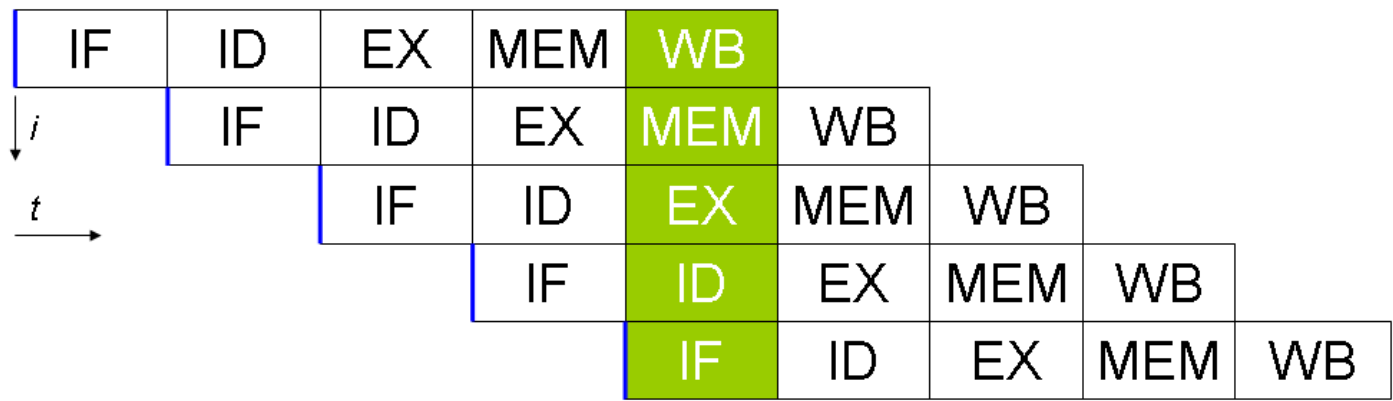


- **suplimentar, situația e complicată de faptul că instrucțiuni diferite au nevoie de un număr diferit de cicli de CPU**

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?

PIPELINING: DATA HAZARDS



- **suplimentar, situația e complicată de faptul că instrucțiuni diferite au nevoie de un număr diferit 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

avem registrii speciali

PIPELINING: DATA HAZARDS

- pentru instrucțiuni “dificile” avem regiștri speciali

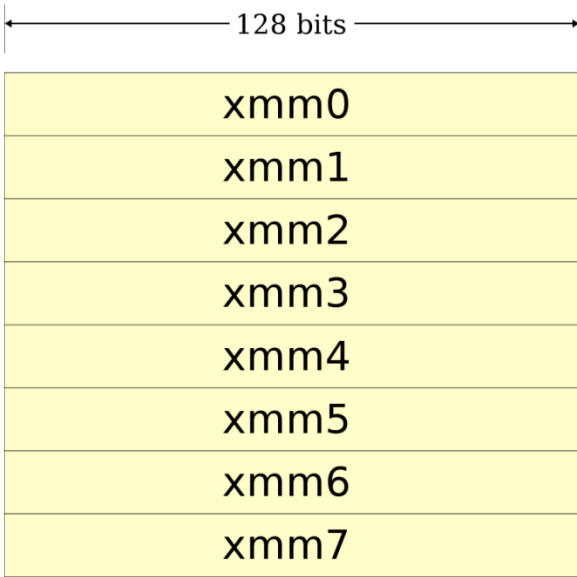
- xmm?
- xmm8-xmm15 există doar pe x64

- original, regiștri suportau
 - 4 înmulțiri pe 32-bit FP

- datorită SSE2, operații cu
 - două numere 64-bit FP
 - două numere întregi pe 64-biți
 - 4 numere pe 32 de biți
 - 8 numere pe 16 biți
 - 16 numere pe 8 biți

- Streaming SIMD Extensions

- exemplu: $\text{result.x} = \text{v1.x} + \text{v2.x}$, $\text{result.y} = \text{v1.y} + \text{v2.y}$, $\text{result.z} = \text{v1.z} + \text{v2.z}$, $\text{result.w} = \text{v1.w} + \text{v2.w}$
- devine:
 - `movaps v1, xmm0` # $\text{xmm0} = \text{v1.w} \mid \text{v1.z} \mid \text{v1.y} \mid \text{v1.x}$
 - `movaps v2, xmm1` # $\text{xmm1} = \text{v2.w} \mid \text{v2.z} \mid \text{v2.y} \mid \text{v2.x}$
 - `addps xmm1, xmm0` # $\text{xmm0} = \text{v1.w} + \text{v2.w} \mid \text{v1.z} + \text{v2.z} \mid \text{v1.y} + \text{v2.y} \mid \text{v1.x} + \text{v2.x}$



PIPELINING: DATA HAZARDS

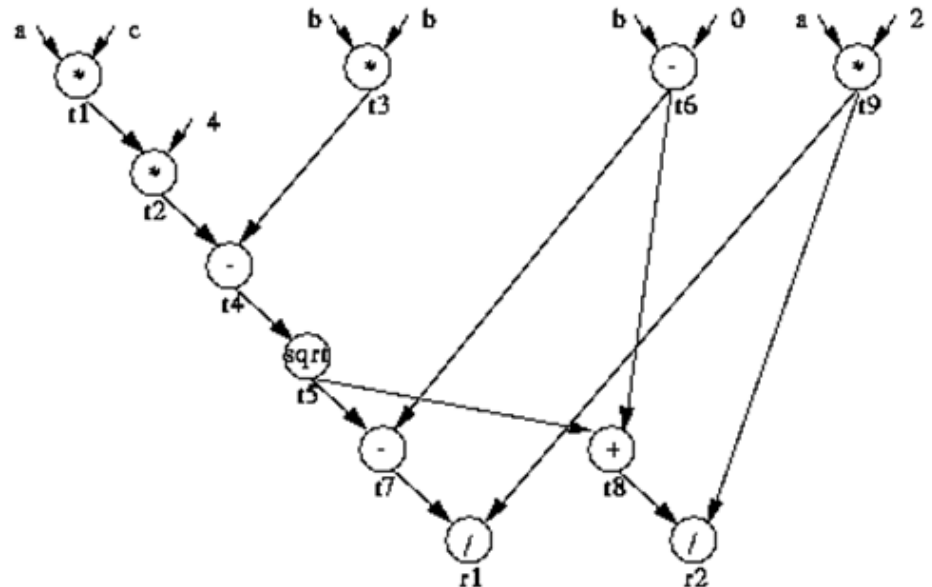
- redenumirea regiștrilor (*register renaming*)
- considerăm codul (unde folosim doar registrul R1)
 - $R1 = \text{data}[1024]$
 - $R1 = R1 + 2$
 - $\text{data}[1032] = R1$
 - $R1 = \text{data}[2048]$
 - $R1 = R1 + 4$
 - $\text{data}[2056] = R1$
- ce observați? ce putem face aici?

PIPELINING: DATA HAZARDS

- redenumirea regiștrilor (*register renaming*)
- considerăm codul (unde folosim doar registrul R1)
 - $R1 = \text{data}[1024]$
 - $R1 = R1 + 2$
 - $\text{data}[1032] = R1$
 - $R1 = \text{data}[2048]$
 - $R1 = R1 + 4$
 - $\text{data}[2056] = R1$
- echivalent, dar mai bine pentru pipelining
 - $R1 = \text{data}[1024]$
 - $R1 = R1 + 2$
 - $\text{data}[1032] = R1$
 - $R2 = \text{data}[2048]$
 - $R2 = R2 + 4$
 - $\text{data}[2056] = R2$
- ultimele 3 instrucțiuni se pot executa acum în paralel cu primele 3

PIPELINING: DATA HAZARDS

- ce se întâmplă în procesoarele moderne?
 - se citesc câteva sute de instrucțiuni la un moment dat
 - în hardware, se realizează un graf (*data-flow graph*) de dependențe între aceste instrucțiuni
- exemplu: quad(a, b, c)
 - $t1 = a * c;$
 - $t2 = 4 * t1;$
 - $t3 = b * b;$
 - $t4 = t3 - t2;$
 - $t5 = \text{sqrt}(t4);$
 - $t6 = -b;$
 - $t7 = t6 - t5;$
 - $t8 = t6 + t5;$
 - $t9 = 2 * a;$
 - $r1 = t7 / t9;$
 - $r2 = t8 / t9;$

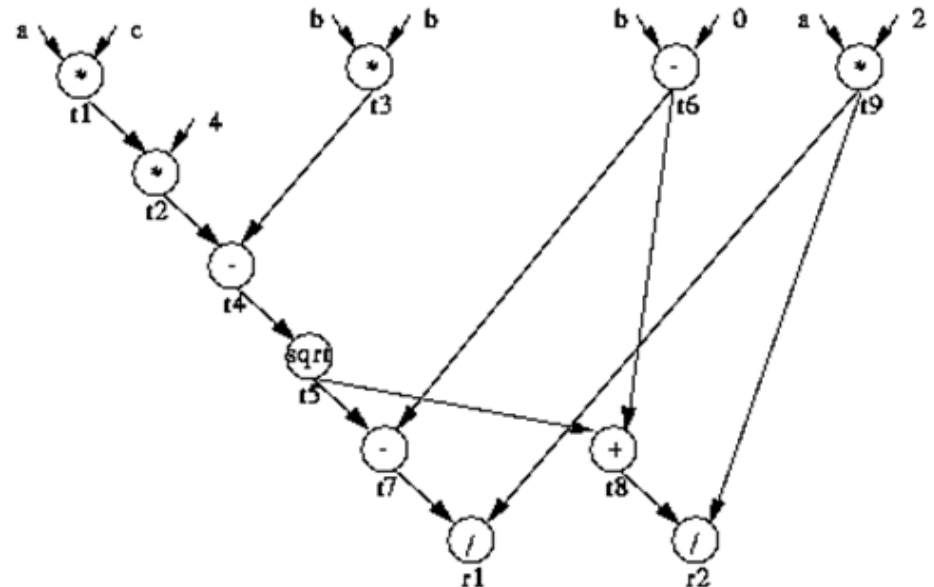


cum executăm eficient acest graf?

PIPELINING: DATA HAZARDS

- ce se întâmplă în procesoarele moderne?
 - se citesc câteva sute de instrucțiuni la un moment dat
 - în hardware, se realizează un graf (*data-flow graph*) de dependențe între aceste instrucțiuni
- exemplu: quad(a, b, c)
 - $t1 = a * c$; $t3 = b * b$; $t6 = -b$; $t9 = 2 * a$;
 - $t2 = 4 * t1$;
 - $t4 = t3 - t2$;
 - $t5 = \text{sqrt}(t4)$;
 - $t7 = t6 - t5$; $t8 = t6 + t5$;
 - $r1 = t7 / t9$; $r2 = t8 / t9$;

6 vs. 11 pași



out of order execution

PIPELINING: BRANCH PREDICTION

- considerați următoarea secvență de cod Assembly

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

- ce face codul de mai sus?

PIPELINING: BRANCH PREDICTION

- considerați următoarea secvență de cod Assembly


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

- ce face codul de mai sus?
 - for loop, apeleaza funcția *callfunc*
 - unde e problema pentru pipelining?

PIPELINING: BRANCH PREDICTION

- considerați următoarea secvență de cod Assembly

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




- ce face codul de mai sus?
 - for loop, apeleaza funcția *callfunc*
 - unde e problema pentru pipelining?
 - avem două posibilități pentru următoarea instrucțiune:
 - `popq %rbx`
 - `movl %ebx, %edi`

PIPELINING: BRANCH PREDICTION

- o potențială soluție: branch prediction (predicția saltului)

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




- în general predicția este binară (dacă sari sau nu, *then* sau *else*)
- ce propuneți voi?

PIPELINING: BRANCH PREDICTION

- o potențială soluție: branch prediction (predicția saltului)

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



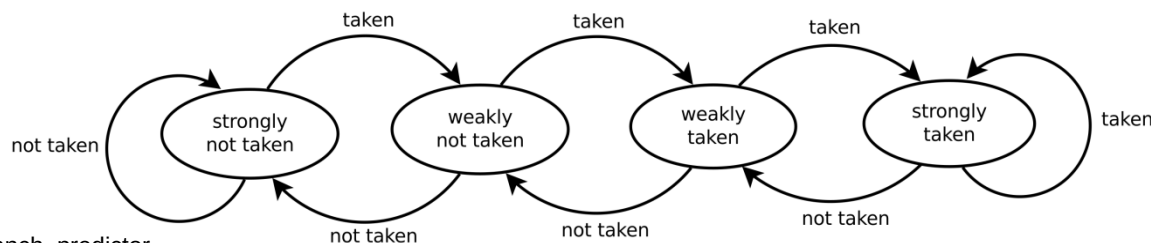
- în general predicția este binară (dacă sari sau nu, *then* sau *else*)
- ce propuneți voi?
 - predicție fixă: mereu sare / mereu nu sare
 - predicția de la pasul anterior: ce a făcut instrucțiunea ultima dată
 - predicția cu istoric: ce face de obicei instrucțiunea
 - execuție speculativă (eager execution): calculează cu și fără salt

PIPELINING: BRANCH PREDICTION

- o potențială soluție: branch prediction (predicția saltului)

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

- în general predicția este binară (dacă sari sau nu, *then* sau *else*)
- ce propuneți voi?
 - predicția cu istoric: ce face de obicei instrucțiunea



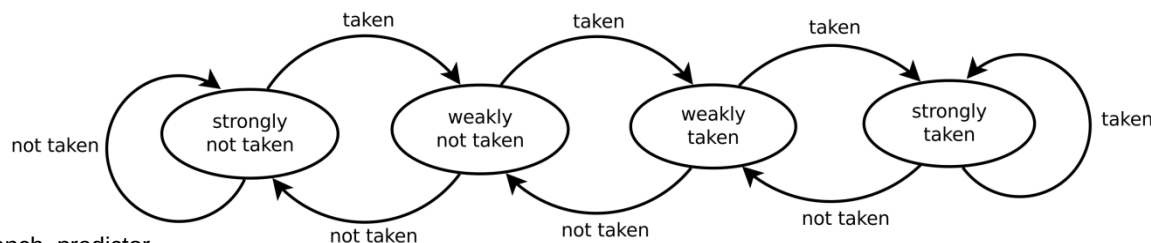
ce e asta?

PIPELINING: BRANCH PREDICTION

- o potențială soluție: branch prediction (predicția saltului)

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

- în general predicția este binară (dacă sari sau nu, *then* sau *else*)
- ce propuneți voi?
 - predicția cu istoric: ce face de obicei instrucțiunea



ce e asta?
counter 2 biți

PIPELINING

- **când complicăm hardware și software pot apărea probleme**
- **în special probleme de securitate**
 - meltdown, spectre
 - aceste două atacuri exploatează execuția speculativă și sistemul ierarhic al memoriei (cache-ul)
- **când complicăm hardware, e cu atât mai rău**
 - soluția: trebuie înlocuit hardware-ul
 - soluția: sistemul de operare trebuie să ia în considerare problema
 - *totul* va fi mai lent

CE AM FĂCUT ASTĂZI

- **pipelining**
- **branch prediction**
- **out of order execution**

DATA VIITOARE ...

- **sisteme multi-procesor**
- **ierarhia memoriei, caching**
- **performanța calculatoarelor**

LECTURĂ SUPLIMENTARĂ

- **PH book**
 - 4.5 An Overview of Pipelining
 - 4.6 Pipelined Datapath and Control
 - 4.7 Data Hazards: Forwarding versus Stalling
 - 4.8 Control Hazards
 - 4.9 Exceptions
 - 4.10 Paralellism via Instructions
- **Crash Course Computer Science**
 - Advanced CPU Designs,
<https://www.youtube.com/watch?v=rtAIC5J1U40>

