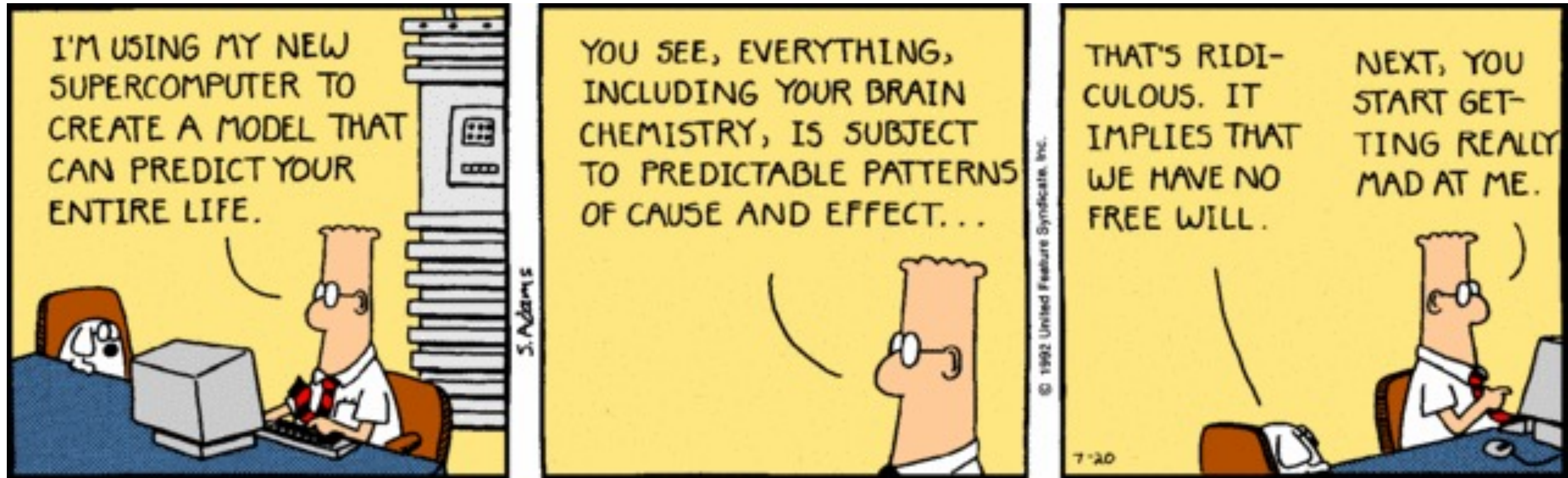
Calculatoare Numerice

– Cursul 12 –

Banda de asamblare (2)

Facultatea de Automatică și Calculatoare
Universitatea Politehnica București

Comic of the day



http://dilbert.com/strips/comic/1992-07-20/

Din cursul anterior

Embedded Systems Laboratory

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$

Crește datorită bulelor din pipeline

Scade, deoarece sunt mai puține porți logice pe căile critice dintre flip-flop-uri

- Pipelining mărește frecvența de operare iar CPI cresc mai lent -> performanță mărită
- Implementarea în b.a. este complicată de HAZARDE:
  - Hazarde structurale (două istrucțiuni vor aceeași resursă hardware)
  - Hazarde de date (instrucțiunea anterioară produce o valoare de care are nevoie instrucțiunea în curs de execuție)
  - Hazarde de control(instrucțiunile schimbă semnalele de control, e.g., branches & exceptions)
- Tehnici de rezolvare:
  1) Interlock-uri (oprim execuția noilor instrucțiuni până când vechile instrucțiuni ies din b.a. și fac write-back)
  2) Bypass (transferăm valoarea vechii instrucțiuni către cea nouă imediat ce rezultatul este disponibil)
  3) Speculează (ghicește efectul instrucțiunii prealabile)

Hazarde de control

Embedded Systems Laboratory

De ce avem nevoie să calculăm următorul PC?

- Pentru Jump-uri
  - Opcode, PC & offset
- Pentru Jump Register
  - Opcode, Register value și PC
- Pentru Conditional Branches
  - Opcode, Register (pentru condiție), PC și offset
- Pentru toate celelalte instrucțiuni
  - Opcode și PC (și trebuie să știm că nu sunt cele de sus)

Calculul PC - bule

Embedded Systems Laboratory

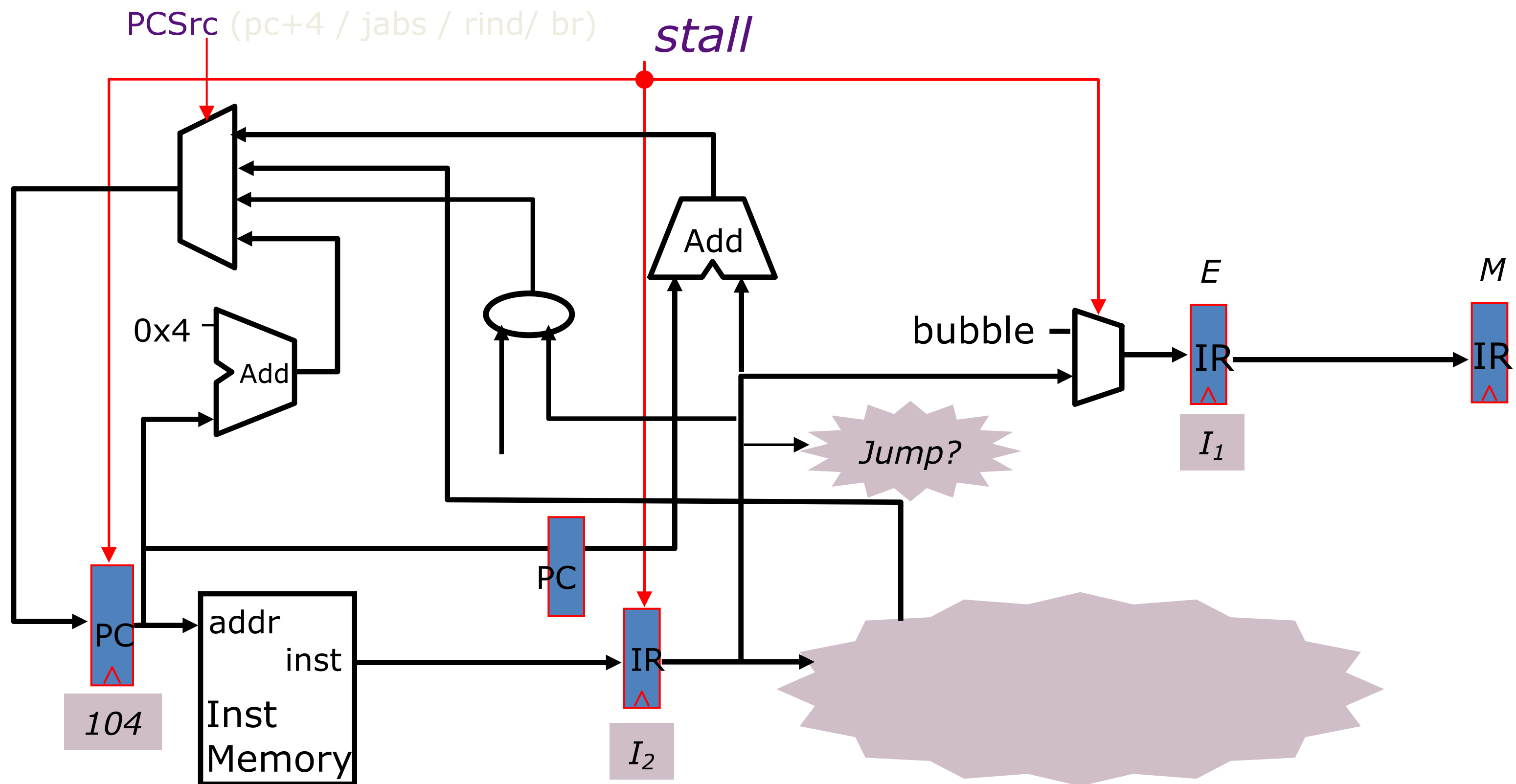
	timp										
	t0	t1	t2	t3	t4	t5	t6	t7	. . . .		
(I_1) x1 ← x0 + 10	IF_1	ID_1	EX_1	MA_1	WB_1						
(I_2) x3 ← x2 + 17		IF_2	IF_2	ID_2	EX_2	MA_2	WB_2				
(I_3)				IF_3	IF_3	ID_3	EX_3	MA_3	WB_3		
(I_4)						IF_4	IF_4	ID_4	EX_4	MA_4	WB_4

Resource Usage

	timp										
	t0	t1	t2	t3	t4	t5	t6	t7	. . . .		
IF	I_1	-	I_2	-	I_3	-	I_4				
ID		I_1	-	I_2	-	I_3	-	I_4			
EX			I_1	-	I_2	-	I_3	-	I_4		
MA				I_1	-	I_2	-	I_3	-	I_4	
WB					I_1	-	I_2	-	I_3	-	I_4

- ⇒ *pipeline bubble*

Speculăm că următoarea adresă e PC+4



I_1	096	ADD
I_2	100	J 304
I_3	~~104~~	~~ADD~~
I_4	304	ADD

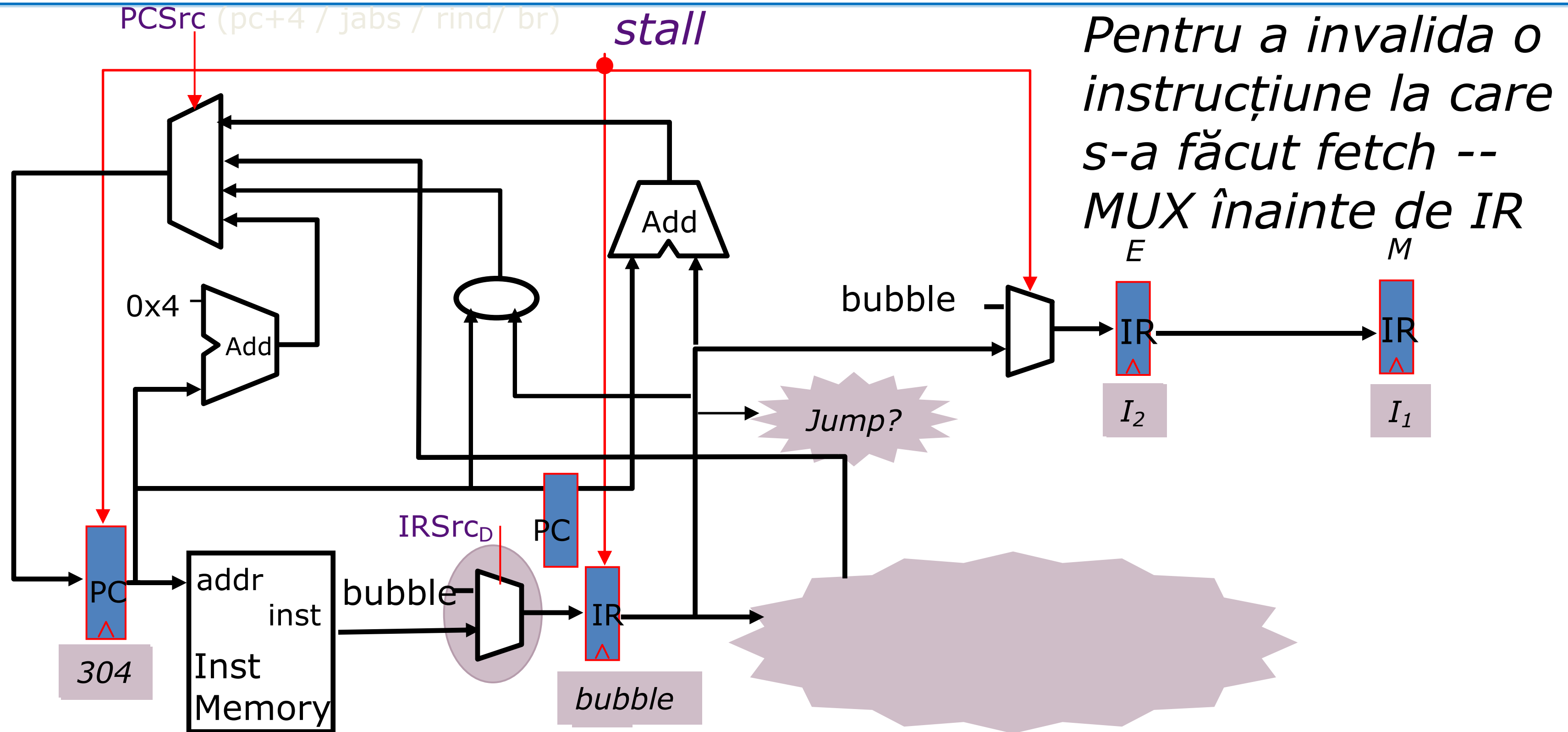
kill

Un jump invalidează (nu oprește) instrucțiunea următoare

Cum?

Implementarea de jump-uri în b.a.

Embedded Systems Laboratory

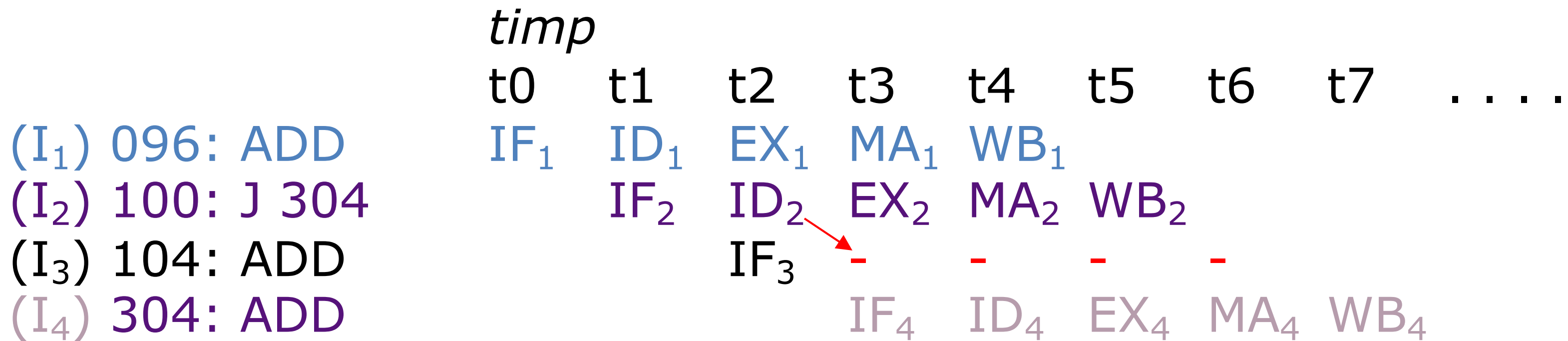


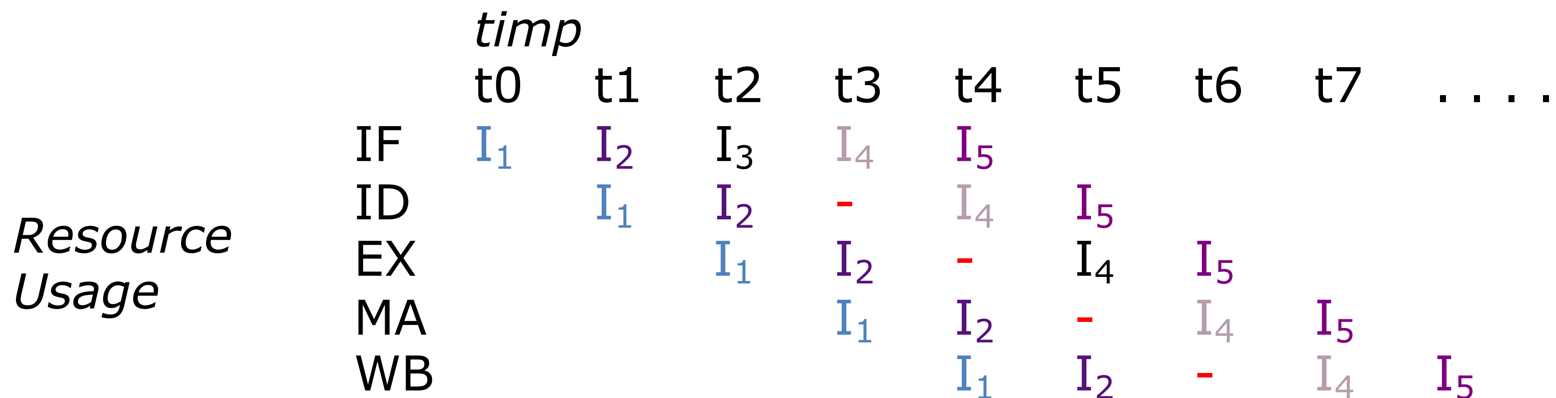
I_1	096	ADD
I_2	100	J 304
I_3	~~104~~	~~ADD~~
I_4	304	ADD

kill

$IRSrc_D$ = *Case* $opcode_D$
J, JAL ⇒ bubble
... ⇒ IM

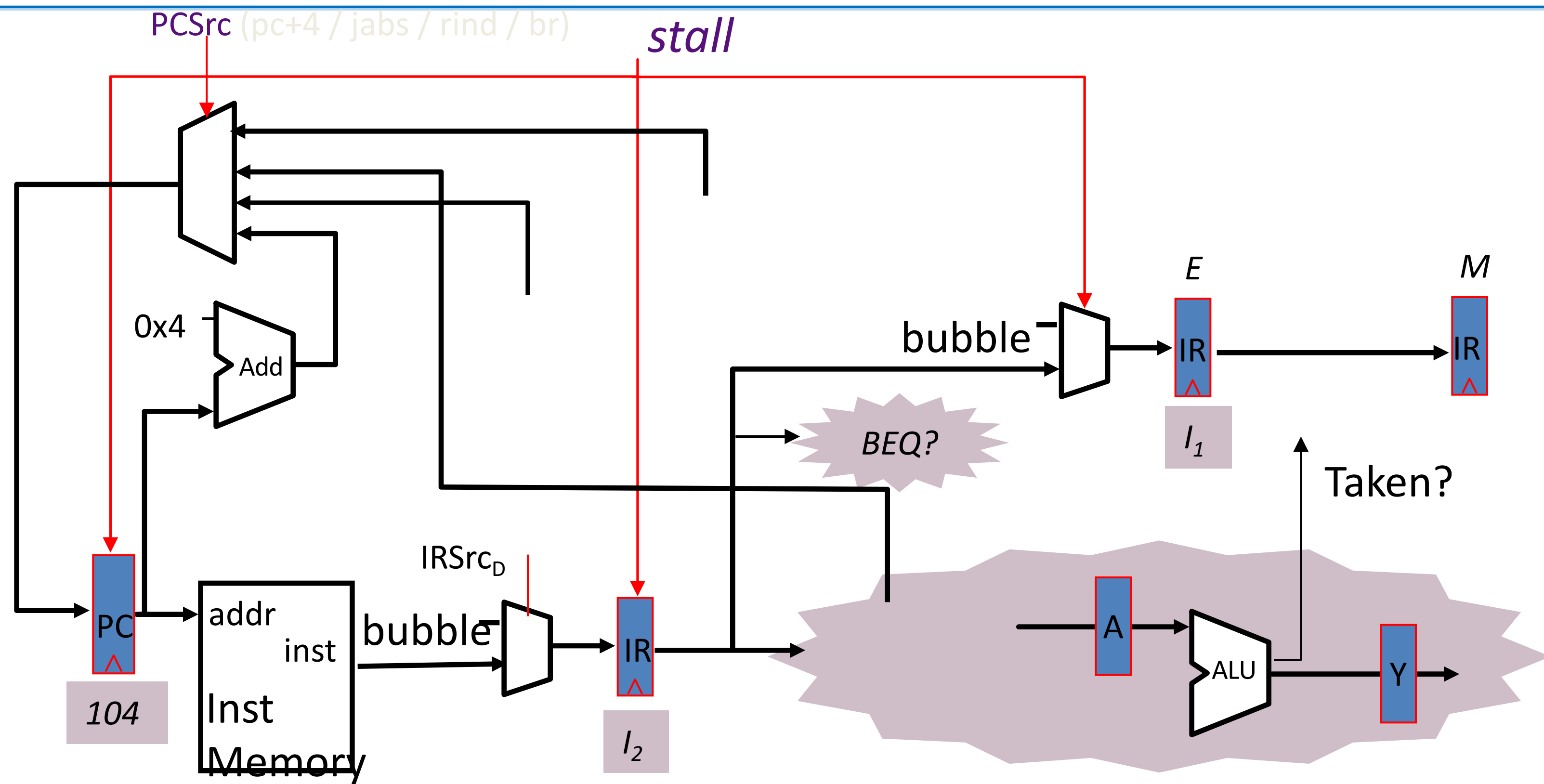
Diagrama Jump în b.a.





- $\Rightarrow$ *pipeline bubble*

Implementarea în b.a. a branch-urilor

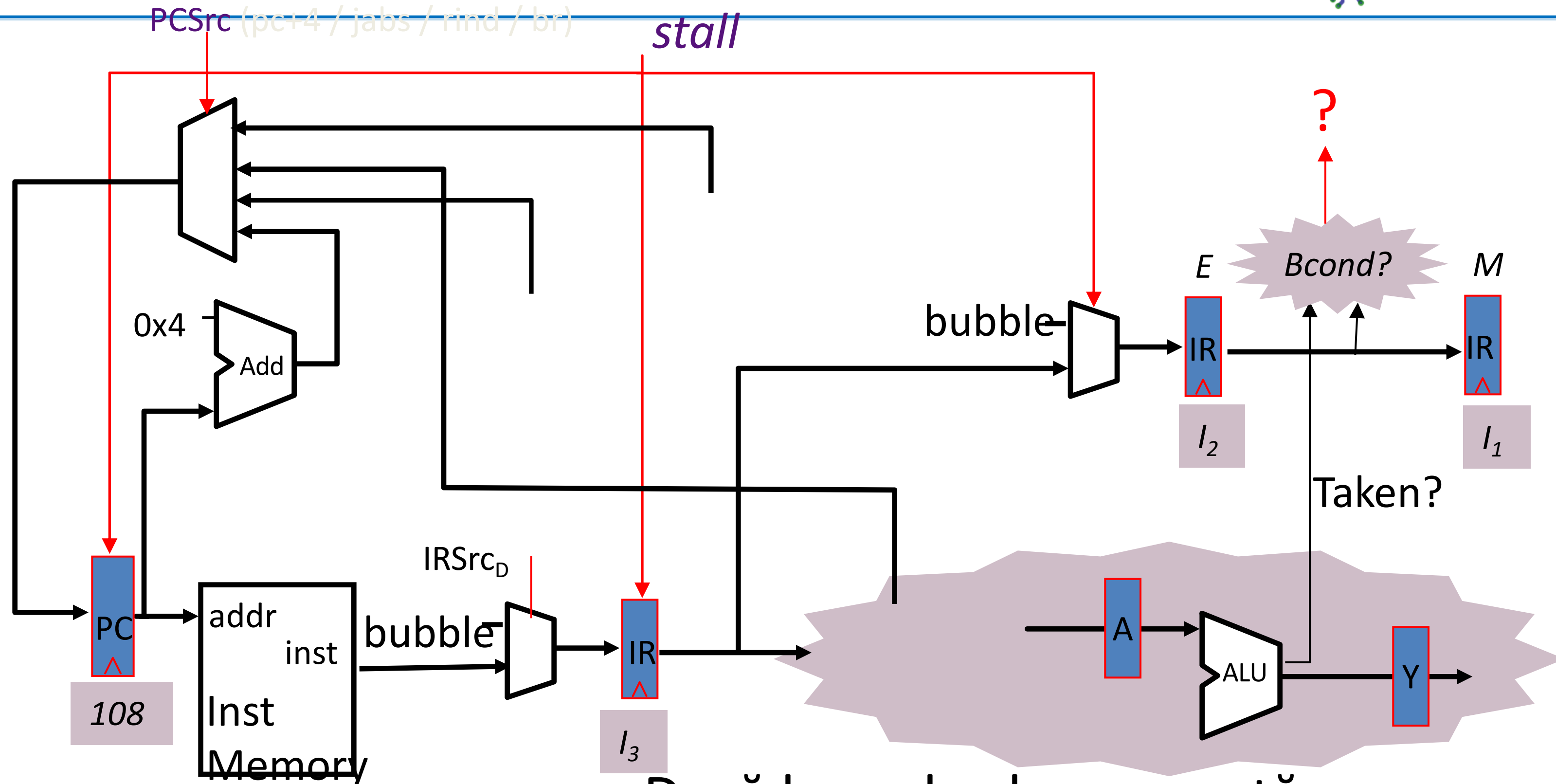


I_1	096	ADD
I_2	100	BEQ x1,x2 +200
I_3	104	ADD
I_4	300	ADD

Condiția de branch nu e cunoscută până la faza de execute

Ce putem face în faza de decode?

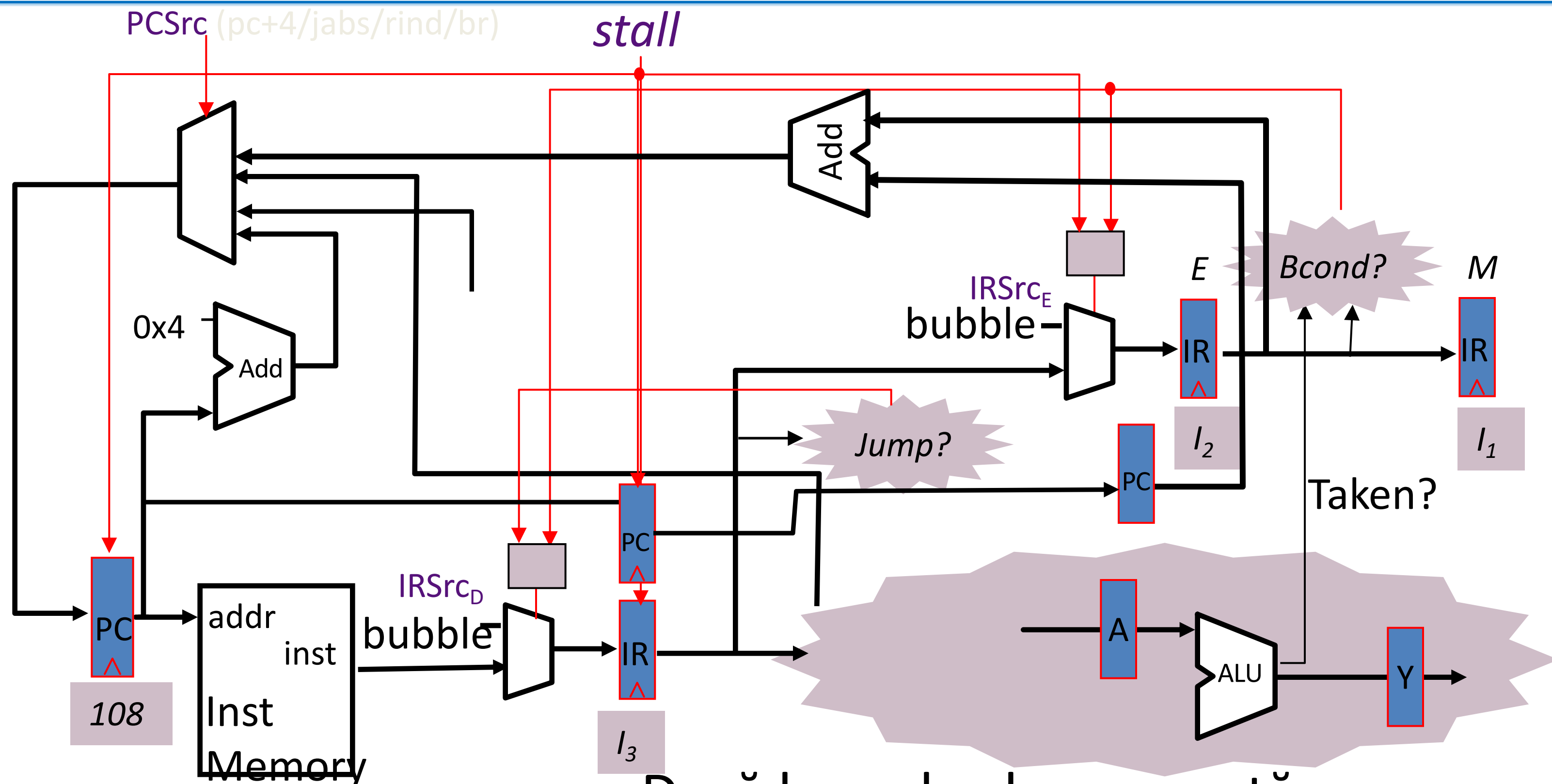
Implementarea în b.a. a branch-urilor



Dacă branch-ul se execută
- Invalidăm următoarele 2 instrucțiuni
- instrucțiunea din faza de decode nu e validă ⇒ semnalul de stall nu e valid

I_1	096	ADD
I_2	100	BEQ x1,x2 +200
I_3	104	ADD
I_4	300	ADD

Implementarea în b.a. a branch-urilor



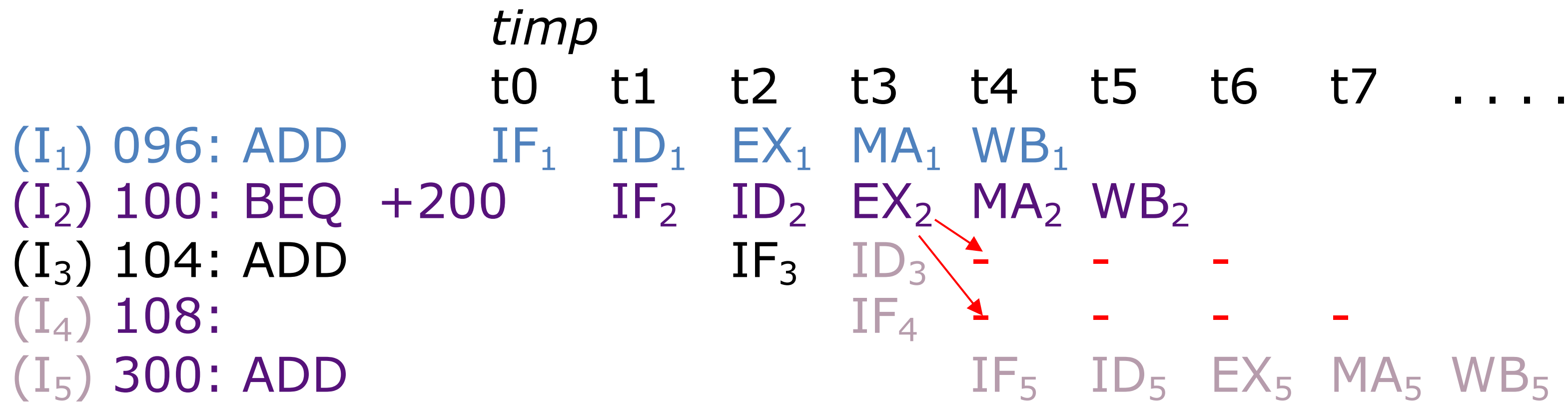
Dacă branch-ul se execută

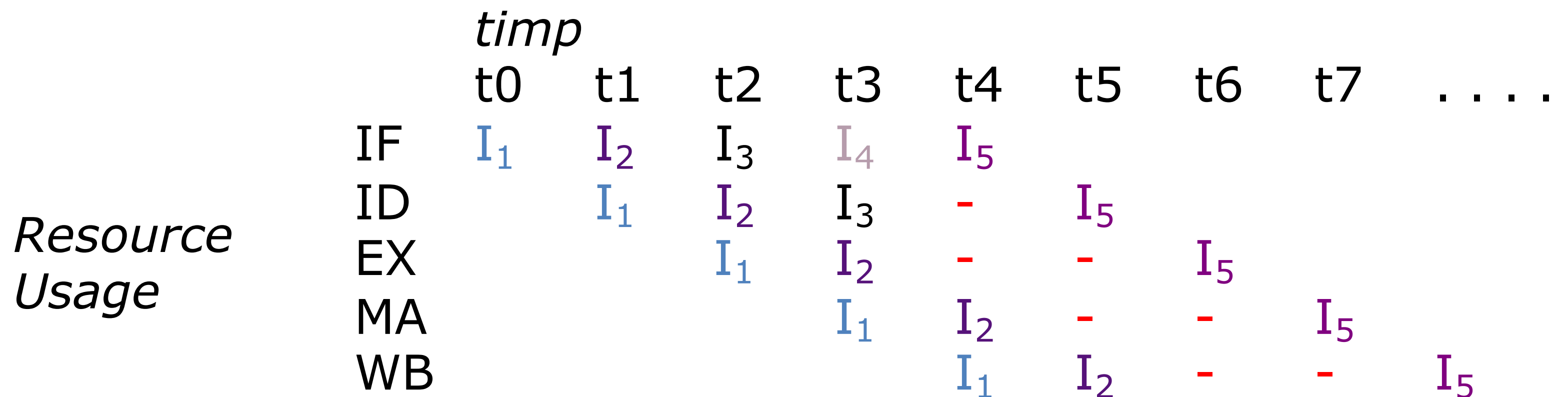
- Invalidăm următoarele 2 instrucțiuni
- instrucțiunea din faza de decode nu e validă ⇒ semnalul de stall nu e valid

I_1:	096	ADD
I_2:	100	BEQ x1,x2 +200
I_3:	104	ADD
I_4:	300	ADD

Diagrame în b.a. pentru branch

Embedded Systems Laboratory





- ⇒ *pipeline bubble*

Folosim branch-uri mai simple (e.g.,comparăm un registru cu zero) cu comparația în faza de decode

Embedded Systems Laboratory

	timp								
	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
(I_1) 096: ADD	IF_1	ID_1	EX_1	MA_1	WB_1				
(I_2) 100: BEQZ +200		IF_2	ID_2	EX_2	MA_2	WB_2			
(I_3) 104: ADD			IF_3	-	-	-	-		
(I_4) 300: ADD				IF_4	ID_4	EX_4	MA_4	WB_4	

Resource Usage

	timp								
	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
IF	I_1	I_2	I_3	I_4	I_5				
ID		I_1	I_2	-	I_4	I_5			
EX			I_1	I_2	-	I_4	I_5		
MA				I_1	I_2	-	I_4	I_5	
WB					I_1	I_2	-	I_4	I_5

- $\Rightarrow$ *pipeline bubble*

Branch Delay Slots
(expunem hazardul de control software-ului)

Embedded Systems Laboratory

- Schimbăm semnatica ISA ca instrucțiunea de după un jump sau branch este executată întotdeauna
  - Dă compilatorului oportunitatea de-a insera o instrucțiune utilă unde in mod normal ar fi fost o bulă în b.a.

I_1	096	ADD
I_2	100	BEQZ r1, +200
I_3	104	ADD ←
I_4	300	ADD

Delay slot instruction executată indiferent de rezultatul branch-ului

Diagrama în b.a. pentru branch
(branch delay slot)

Embedded Systems Laboratory

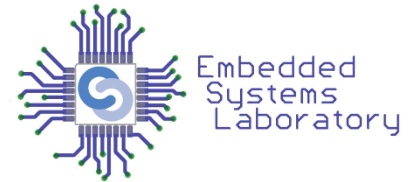
timp

	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
(I_1) 096: ADD	IF_1	ID_1	EX_1	MA_1	WB_1				
(I_2) 100: BEQZ +200		IF_2	ID_2	EX_2	MA_2	WB_2			
(I_3) 104: ADD			IF_3	ID_3	EX_3	MA_3	WB_3		
(I_4) 300: ADD				IF_4	ID_4	EX_4	MA_4	WB_4	

timp

Resource Usage		t0	t1	t2	t3	t4	t5	t6	t7	. . . .
	IF	I_1	I_2	I_3	I_4					
	ID		I_1	I_2	I_3	I_4				
	EX			I_1	I_2	I_3	I_4			
	MA				I_1	I_2	I_3	I_4		
	WB					I_1	I_2	I_3	I_4	

De ce nu putem să executăm tot timpul o instrucțiune nouă la fiecare ciclu de ceas? (CPI>1)



- Un sistem de bypass complet s-ar putea să fie prea dificil de implementat (și prea costisitor)
  - De obicei, sunt rutate cele mai utilizate căi de bypass
  - Pentru alte rute este posibil să avem nevoie de să mărim ciclul de ceas, ceea ce ar contracara beneficiile date de reducerea CPI
- Un load are latență de doi cicli
  - O instrucțiune imediat după load nu poate să utilizeze rezultatul load-ului
  - MIPS-I ISA definește *load delay slots*, un hazard vizibil software (compilatorul planifică o instrucțiune independentă sau inserează un NOP pentru a evita hazardul). Înlăturat la MIPS-II (prin adăugarea de pipeline interlocks în hardware)
    - MIPS:"**M**icroprocessor without **I**nterlocked **P**ipeline **S**tages"
- Branch-urile condiționale pot cauza bule
  - Invalidează următoarea instrucțiune dacă nu avem delay slots.

RISC-V Branches & Jumps

Embedded Systems Laboratory

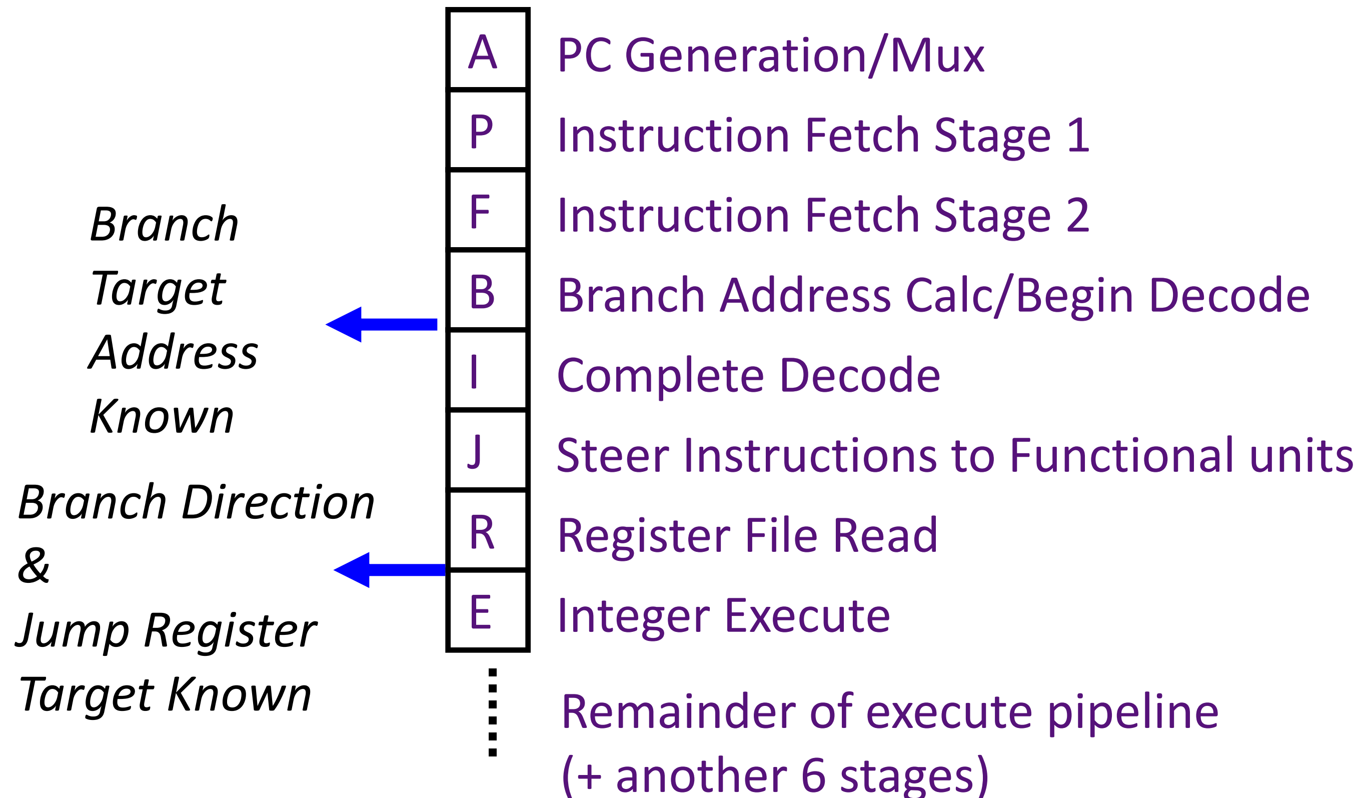
Fiecare instruction fetch depinde de una sau două informații de la instrucțiunea precedentă:

1) Este instrucțiunea precedentă un branch valid?
2) Dacă da, care e adresa țintă?

Instruction	*Taken known?*	*Target known?*
J	After Inst. Decode	After Inst. Decode
JR	After Inst. Decode	After Reg. Fetch
B<cond.>	After Execute	After Inst. Decode

Penalizările pentru branch în b.a. moderne

Embedded Systems Laboratory



Reducerea penalizărilor în lanțul de control

- Soluții software
  - Eliminarea branch-urilor - loop unrolling
    - Mărește lungimea programului
  - Reducerea timpului de decizie – instruction scheduling
    - Calculează condiția pentru branch cât de curând posibil
- Soluții hardware
  - Găsește altceva de făcut - delay slots
    - Umple bulele din b.a. cu instrucțiuni utile (necesită cooperare din partea software-ului)
  - Speculează - branch prediction
    - Execuția speculativă a instrucțiunilor de după branch

Branch Prediction

Embedded Systems Laboratory

Motivație:

Penalizările pentru branch limitează profund performanța procesoarelor în b.a.

Predictoarele moderne au o acuratețe destul de mare (>95%) și pot reduce penalizările semnificativ

Suport hardware necesar:

Structuri pentru predicție:

- Branch history tables, branch target buffers, etc.

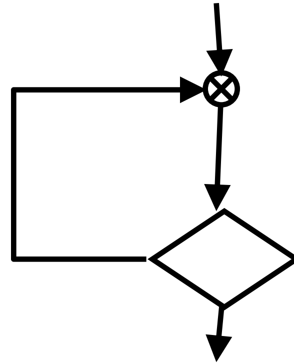
Mecanisme de recuperare în cazul unei predicții proaste:

- *Ține rezultatele calculelor separate de commit*
- Invalidează instrucțiunile de după branch din b.a.
- Reface starea pentru branch-ul respectiv

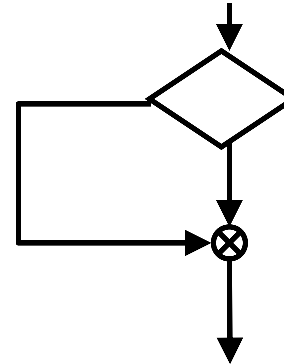
Static Branch Prediction

Probabilitatea totală ca un branch să fie valid este de ~60-70% dar:

înapoi
90%



înainte
50%

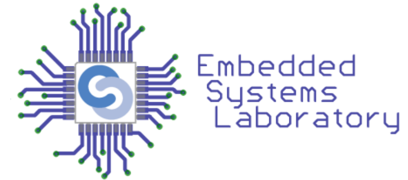


ISA poate să atașeze o semantică potrivită la direcțiile pentru branch-uri, e.g., Motorola MC88110

bne0 *(direcția preferată)* beq0 *(mai puțin folosită)*

ISA poate permite alegerea arbitrară a direcției prezise static, e.g., HP PA-RISC, Intel IA-64 sunt de obicei ~80% precise

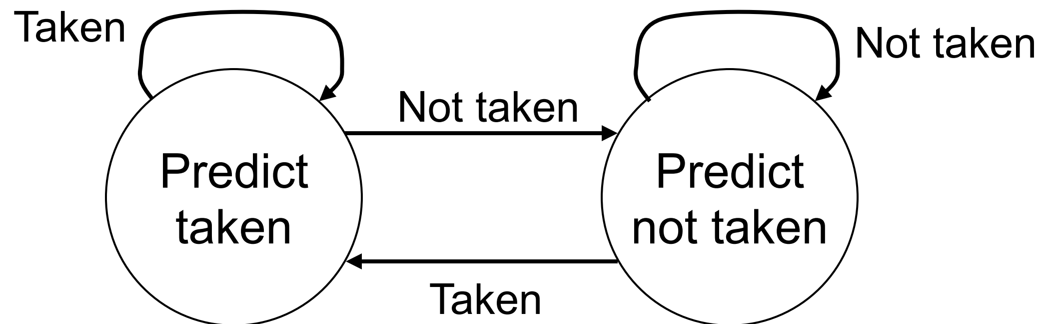
Dynamic Branch Prediction
- învățând din experiențele anterioare



- Corelație temporală
  - Felul în care se execută un branch poate fi un bun indicator despre modul în care se va executa la următoarea trecere prin program.
- Corelație spațială
  - O structură cu mai multe branch-uri se poate executa într-o manieră corelată (cale preferată de execuție)

Predicția branch-urilor

Embedded Systems Laboratory



Automat pentru predicția branch-urilor cu două stări.

Probleme cu această abordare:

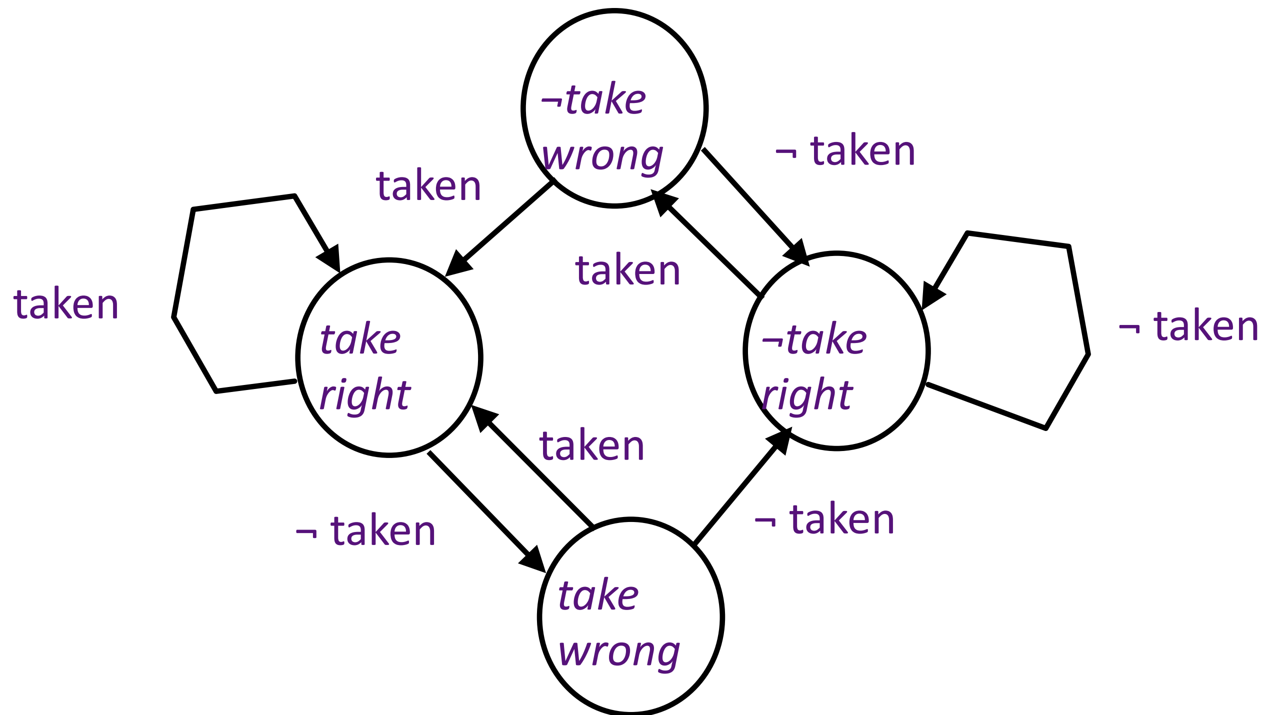
Dacă avem branch-uri într-o buclă, avem automat 2 previziuni greșite:

La prima iterație (bucla este repetată dar istoricul indică ieșirea din buclă)

La ultima interație (când bucla se termină dar istoricul arată că există repetiție)

Elemente de predicție a branch-urilor

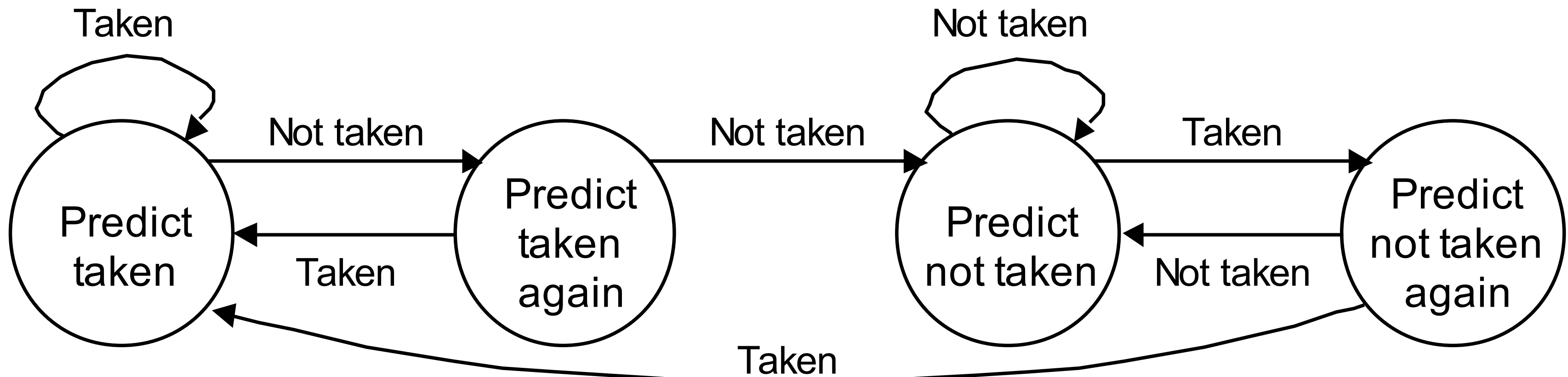
- Presupunem că avem doi biți BP per instrucțiune
- Schimbăm politica de predicție după două greșeli consecutive!



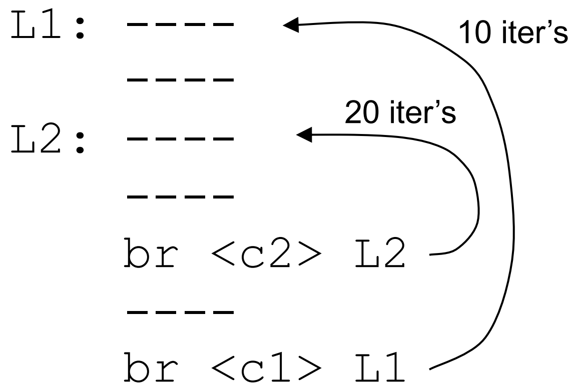
BP state:

(*predict* take/¬take) x (*last prediction* right/wrong)

Elemente de predicție a branch-urilor



Schemă de predicție cu patru stări



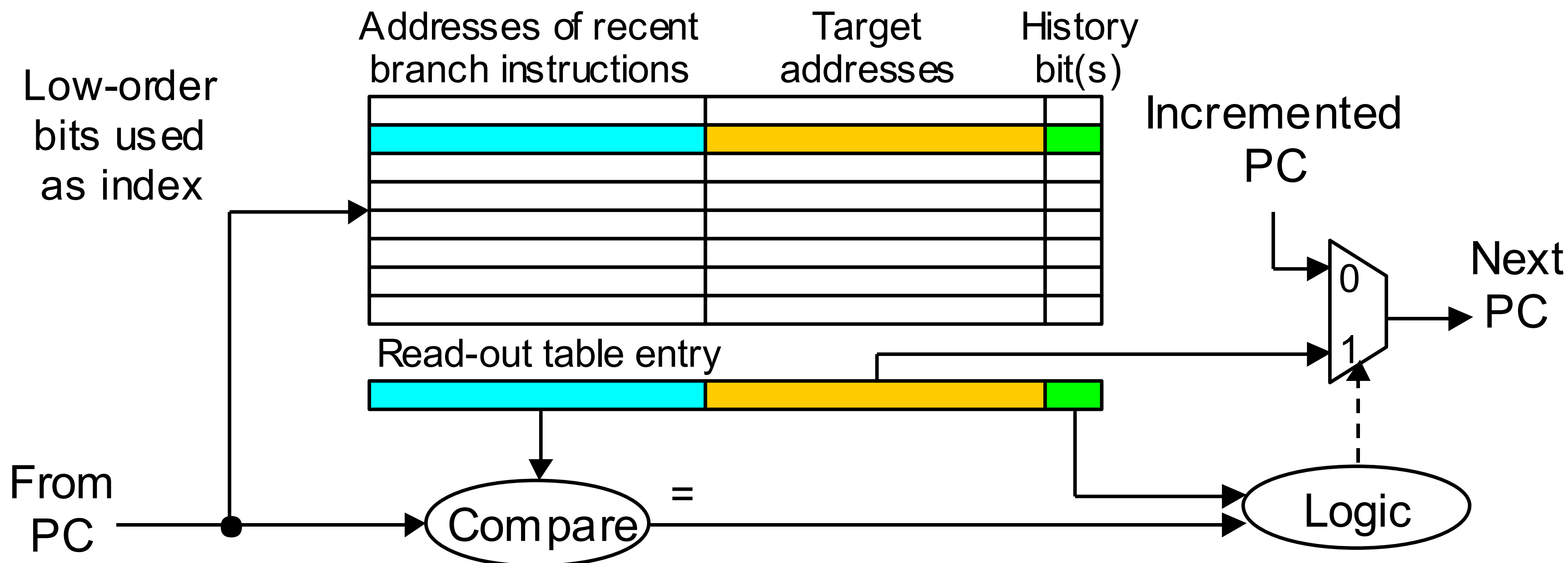
Impactul diferitelor scheme de predicție

Soluție

Always taken: 11 predicții proaste, 94.8% precis
1-bit history: 20 predicții proaste, 90.5% precis
2-bit history: la fel ca Always taken

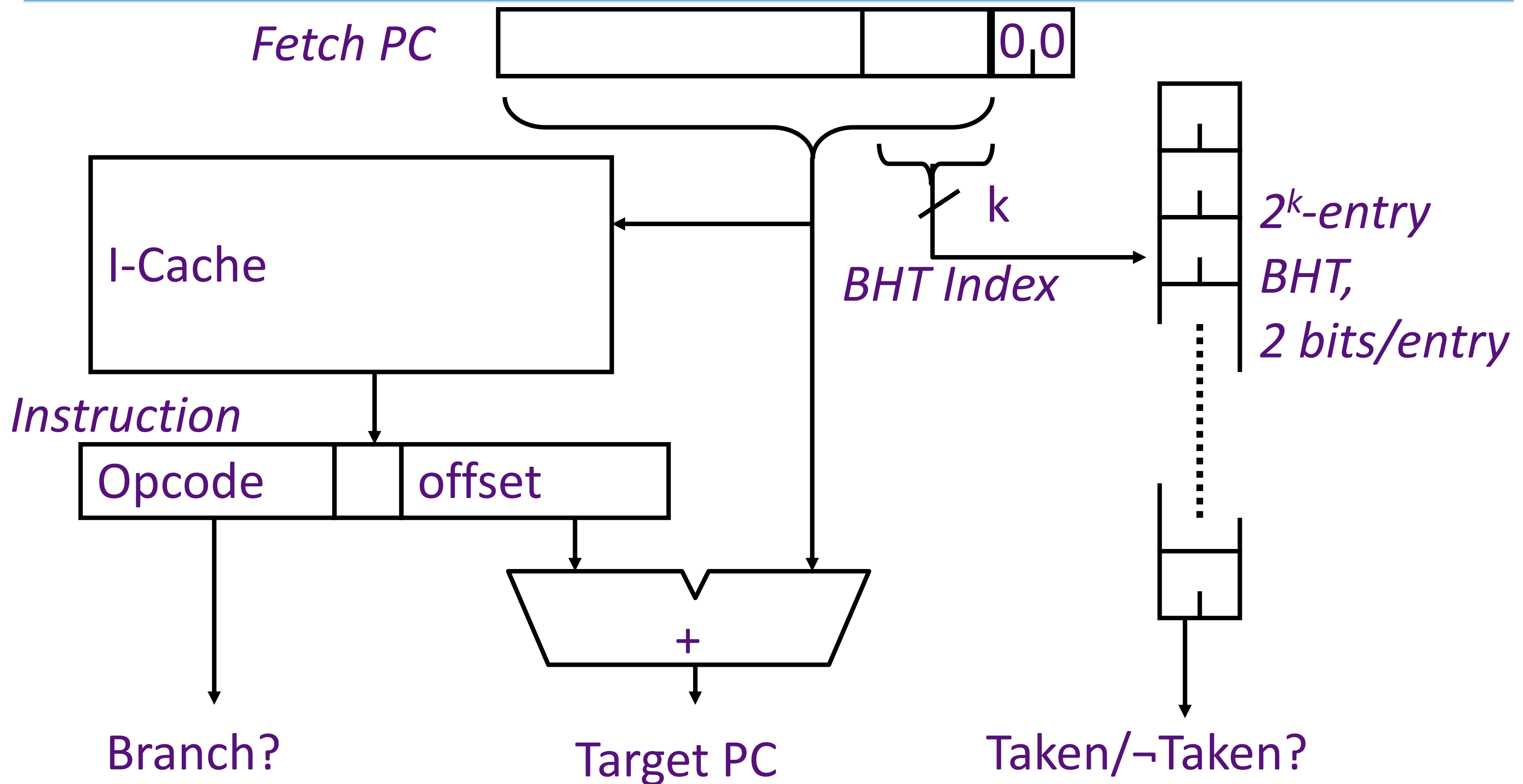
Implementarea hardware pentru branch prediction

Embedded Systems Laboratory



Elementele hardware ale unei scheme de branch prediction

Branch History Table



4K-entry BHT, 2 bits/entry, ~80-90% correct predictions

Exploatarea corelării spațiale

Yeh and Patt, 1992

Embedded Systems Laboratory

```
if (x[i] < 7) then
    y += 1;
if (x[i] < 5) then
    c -= 4;
```

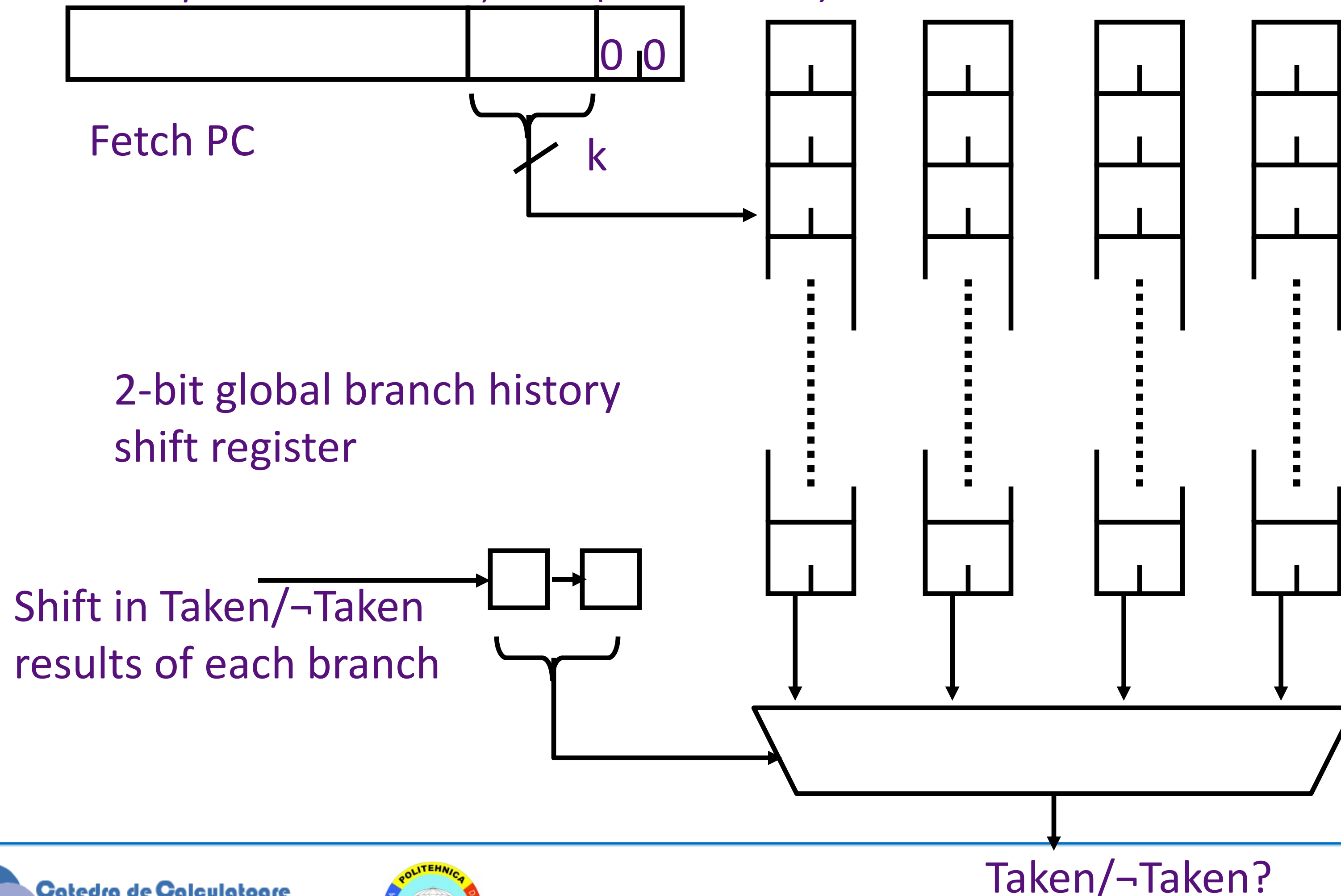
Dacă prima condiție e falsă atunci și a doua este falsă

History register, H, înregistrează direcția ultimelor N branch-uri executate de către procesor

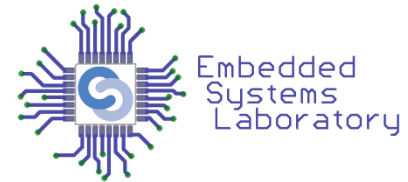
Branch prediction pe două niveluri

Embedded Systems Laboratory

Pentium Pro folosește rezultatul ultimelor două branch-uri pentru a selecta unul din cele patru seturi de biți BHT (~95% corect)



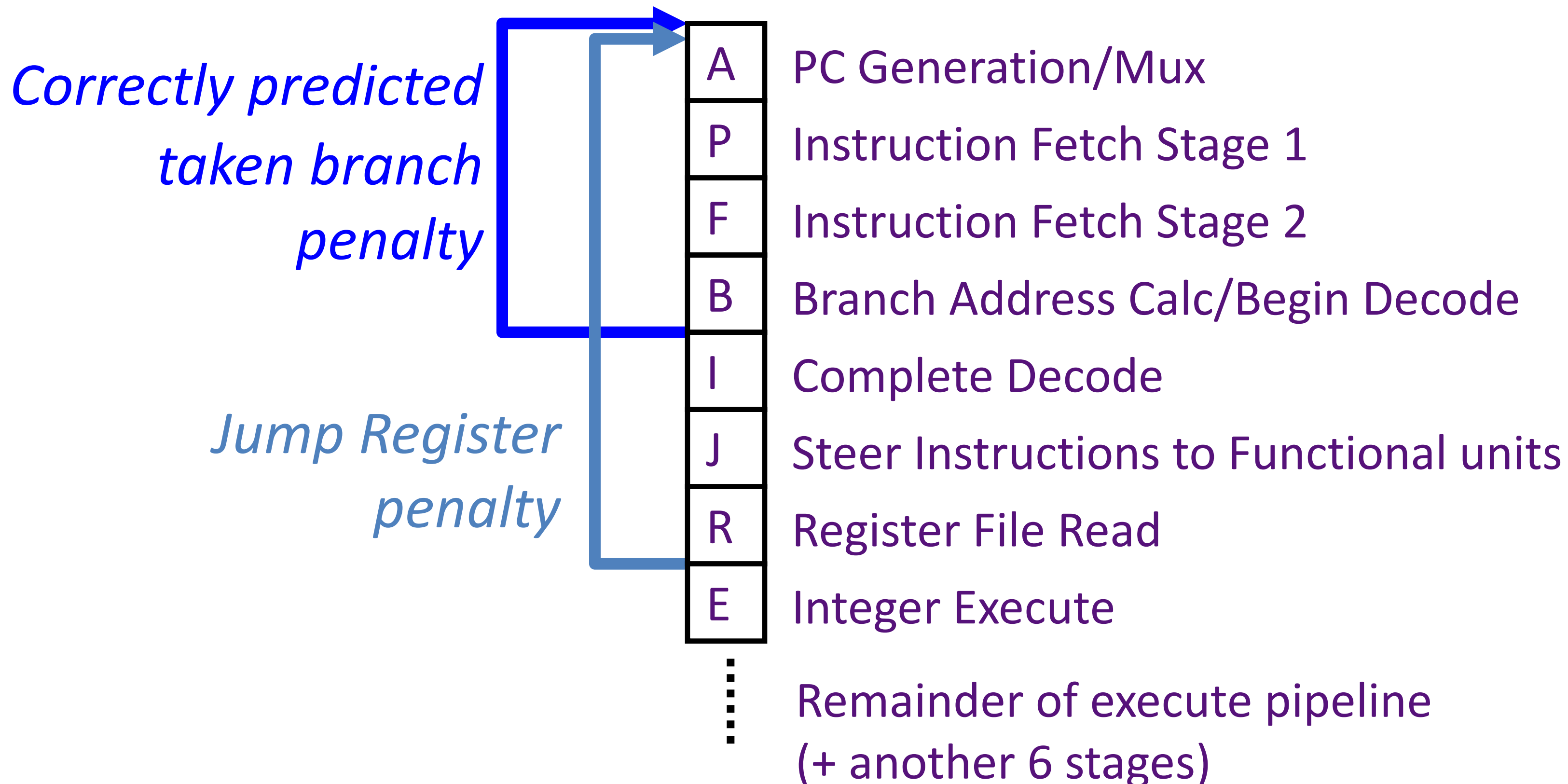
Speculații în ambele direcții



- O alternativă la branch prediction este execuția speculativă a ambelor direcții ale branch-ului
  - Cerințele de resurse sunt proporționale cu numărul de execuții speculative concurente
  - Doar jumătate din resurse vor face muncă utilă când ambele direcții ale unui branch sunt executate speculativ
  - Un branch prediction clasic folosește mai puține resurse decât o execuție speculativă concurentă
- Dacă avem un branch predictor cu o precizie foarte bună, e mai eficient din punct de vedere al costului să dedicăm toate resursele direcției prezise!

Limitările BHT

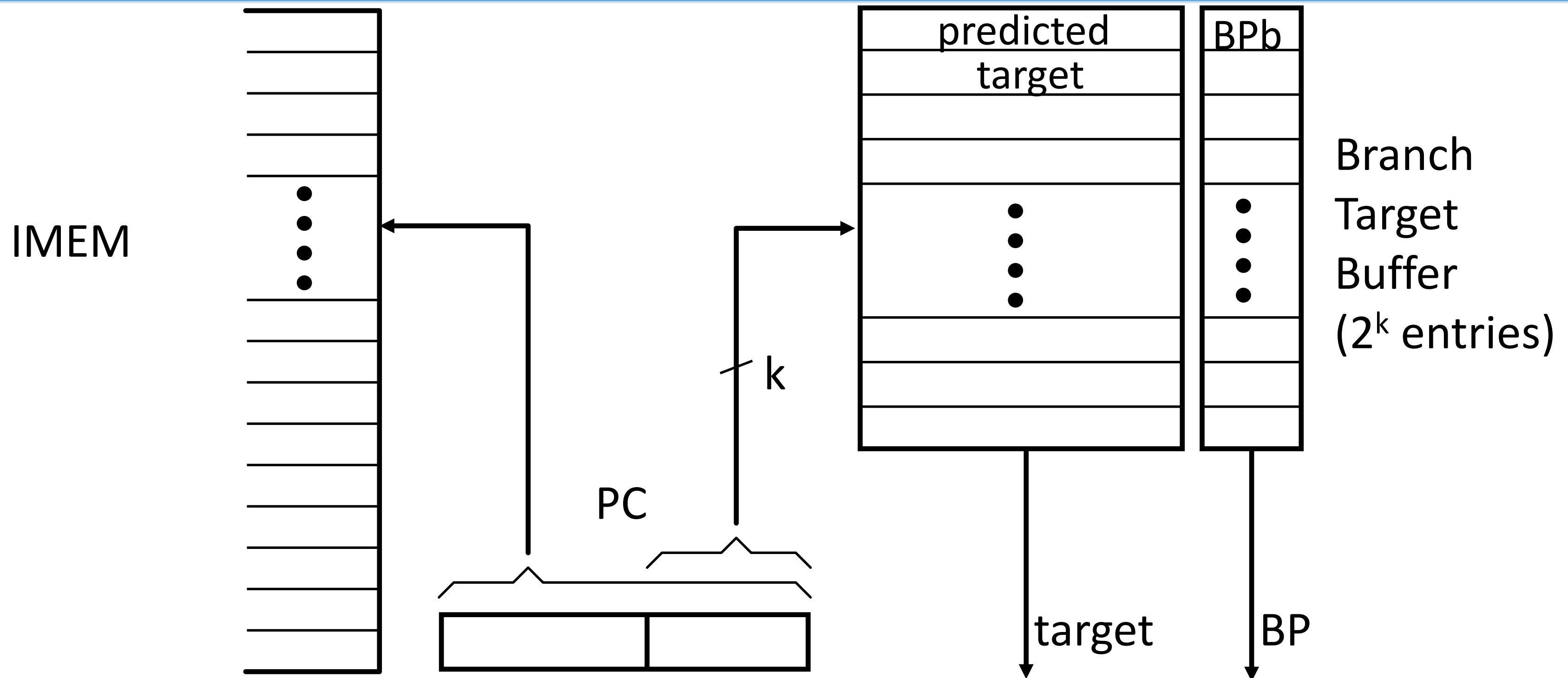
Prezice doar direcția branch-ului. Prin urmare, nu poate să redirecteze operațiile de fetch decât după ce s-a determinat ținta branch-ului.



UltraSPARC-III fetch pipeline

Branch Target Buffer

Embedded Systems Laboratory



Biții BP sunt stocați împreună cu adresa țintă prezisă pentru salt.

IF stage: *If (BP=taken) then nPC=target else nPC=PC+4*
Later: *check prediction, if wrong then kill the instruction and update BTB & BPb else update BPb*

BTB este folosit numai pentru instrucțiunile de control

- BTB conține informații utile doar pentru instrucțiunile de branch și jump

  -> Nu face update pentru celalalte instrucțiuni

- Pentru toate celalalte instrucțiuni, urmăroarea adresă e PC+4 !

- *Cum obținem un astfel de comportament fără a decodifica instrucțiunea?*

Branch Target Buffer (BTB)

Embedded Systems Laboratory

I-Cache

PC

2^k-entry direct-mapped BTB
(can also be associative)

Entry PC

Valid

predicted target PC

k

=

match

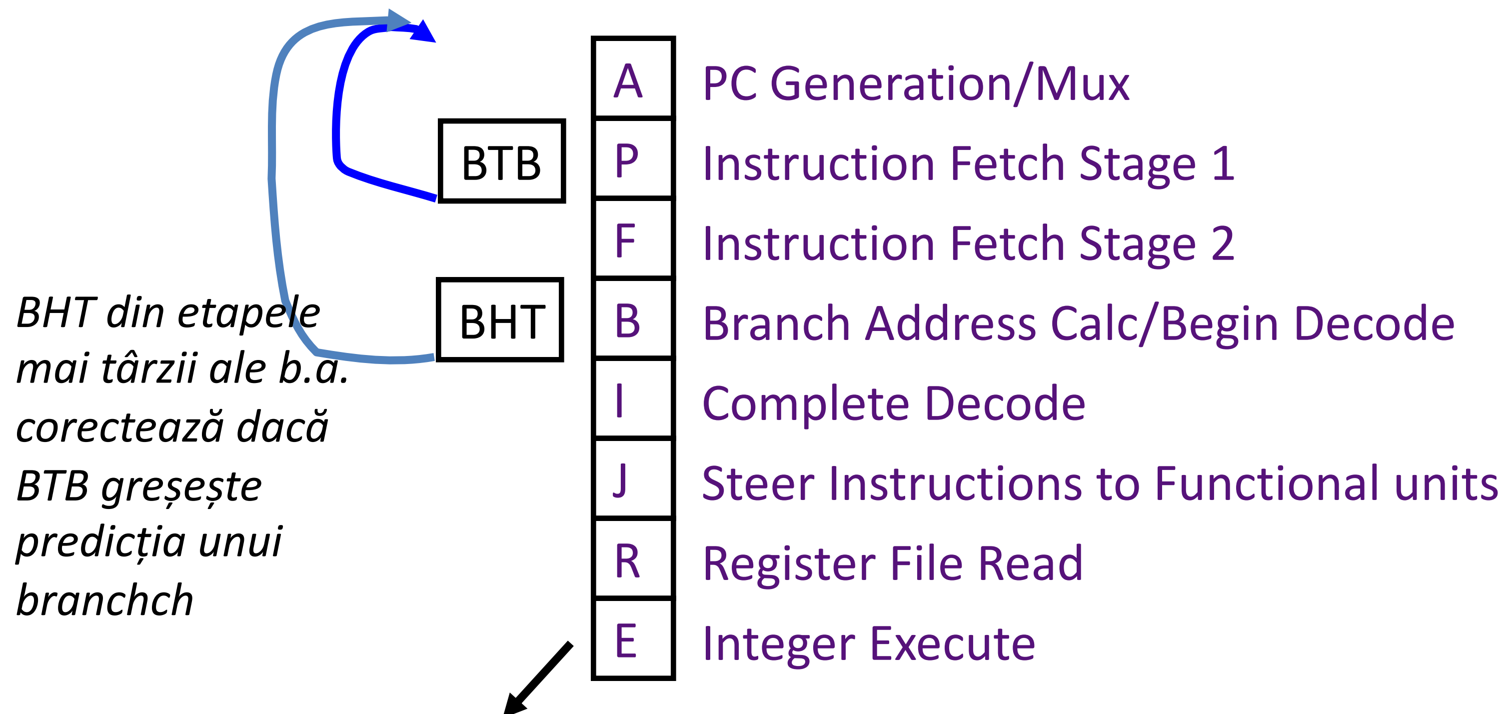
valid

target

- Ținem și PC-ul pentru branch și PC-ul țintă în BTB
- Facem fetch la PC+4 dacă cele două nu coincid
- Doar branch-urile și jump-urile active sunt ținute în BTB
- Următorul PC e determinat înainte să se facă fetch și decode la un branch

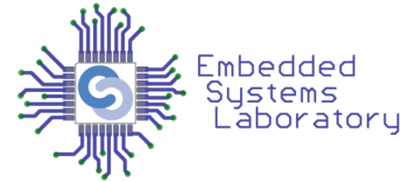
Combinăm BTB și BHT

- Intrările în BTB sunt considerabil mai costisitoare decât cele din BHT, dar pot să redirecționeze fetch-urile mai devreme în b.a.și pot să accelereze branch-urile indirecte (JR)
- BHT poate să țină mai multe înregistrări și este mai precisă



BTB/BHT sunt actualizate după ce branch-ul se execută (etapa E)

Utilizarea Jump Register (JR)



- Switch-uri (jump la adresa pentru case-ul corespunzător)

  BTB funcționează bine dacă folosim același case în mod repetat

- Apelare dinamică de funcții (jump la adresa de run-time a funcției)

  BTB funcționează bine dacă apelăm aceeași funcție în mod repetat, (e.g., în C++/Java, când obiectele au același tip într-un apel virtual de funcție)

- Întoarceri din subrutine (jump to return address)

  BTB funcționează bine dacă avem return la aceeași adresă

  $\Rightarrow$ *De obicei, aceeași funcție este apelată din mai multe locuri într-un program!*

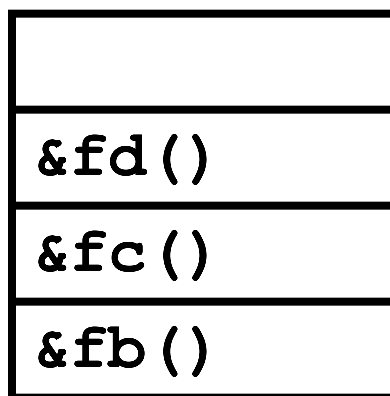
Cât de bine funcționează BTB pentru fiecare din aceste cazuri?

Stiva pentru întoarcere din subrutine

Structură de mici dimensiuni pentru accelerarea JR în cazul return-urilor din subrutine, de obicei mai precisă decât un BTB.

```
fa() { fb(); }
fb() { fc(); }
fc() { fd(); }
```

Facem push la adresa la care se face call atunci când se apelează funcția



Pop întoarce adresa atunci când este decodificată instrucțiunea de return din subrutină

k intrări
(de obicei k=8-16)

Variația CPI în funcție de diferitele arhitecturi

Embedded Systems Laboratory

Efectul pe care le au arhitectura, metodele de branch prediction și execuție speculativă asupra CPI.

Architectură	Metode utilizate în practică	CPI
Nonpipelined, multicycle	Strict in-order instruction issue and exec	5-10
Nonpipelined, overlapped	In-order issue, with multiple function units	3-5
Pipelined, static	In-order exec, simple branch prediction	2-3
Superpipelined, dynamic	Out-of-order exec, adv branch prediction	1-2
Superscalar	2- to 4-way issue, interlock & speculation	0.5-1
Advanced superscalar	4- to 8-way issue, aggressive speculation	0.2-0.5

3.3 inst / cycle × 3 Gigacycles / s
≅ 10 GIPS

Dezvoltarea procesoarelor Intel

Embedded Systems Laboratory

La început a fost 8080; urmat de 80x86 = IA32 ISA

În jur de cinci etape în banda de asamblare:

- 80286
- 80386
- 80486
- Pentium (80586)

Tehnologie mai avansată

În jur de zece etape în banda de asamblare, out-of-order execution

- Pentium Pro
- Pentium II
- Pentium III
- Celeron

Tehnologie mai avansată

Instrucțiunile sunt sparte în microinstrucțiuni care sunt executate out-of-order dar sunt retrase în ordine.

20-30 de etape în banda de asamblare

- Pentium 4

Starea curentă a performanței calculatoarelor

Embedded Systems Laboratory

Multi-GIPS/GFLOPS pentru desktop-uri și laptop-uri

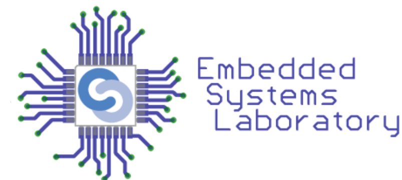
- Foarte puțini utilizatori au nevoie de mai multă putere de calcul
- Utilizatorii nu mai sunt atât de doritori să facă upgrade doar pentru a avea un procesor mai rapid
- Accentul se pune mai mult pe reducerea consumului și ergonomie

Multi-PIPS/PFLOPS în centrele mari de calcul

- Top 500 al supercalculatoarelor, http://www.top500.org
- Următoarea listă în iunie 2020; din noiembrie 2019:
- Toate 500 >> 1 PFLOPS, ≈30 > 6 PFLOPS, 3 > 90 PFLOPS
- *(în 2008) Toate 500 >> 10 TFLOPS, ≈30 > 100 TFLOPS, 1 > PFLOPS*

Multi-EIPS/EFLOPS supercalculatoare în faza de proiectare

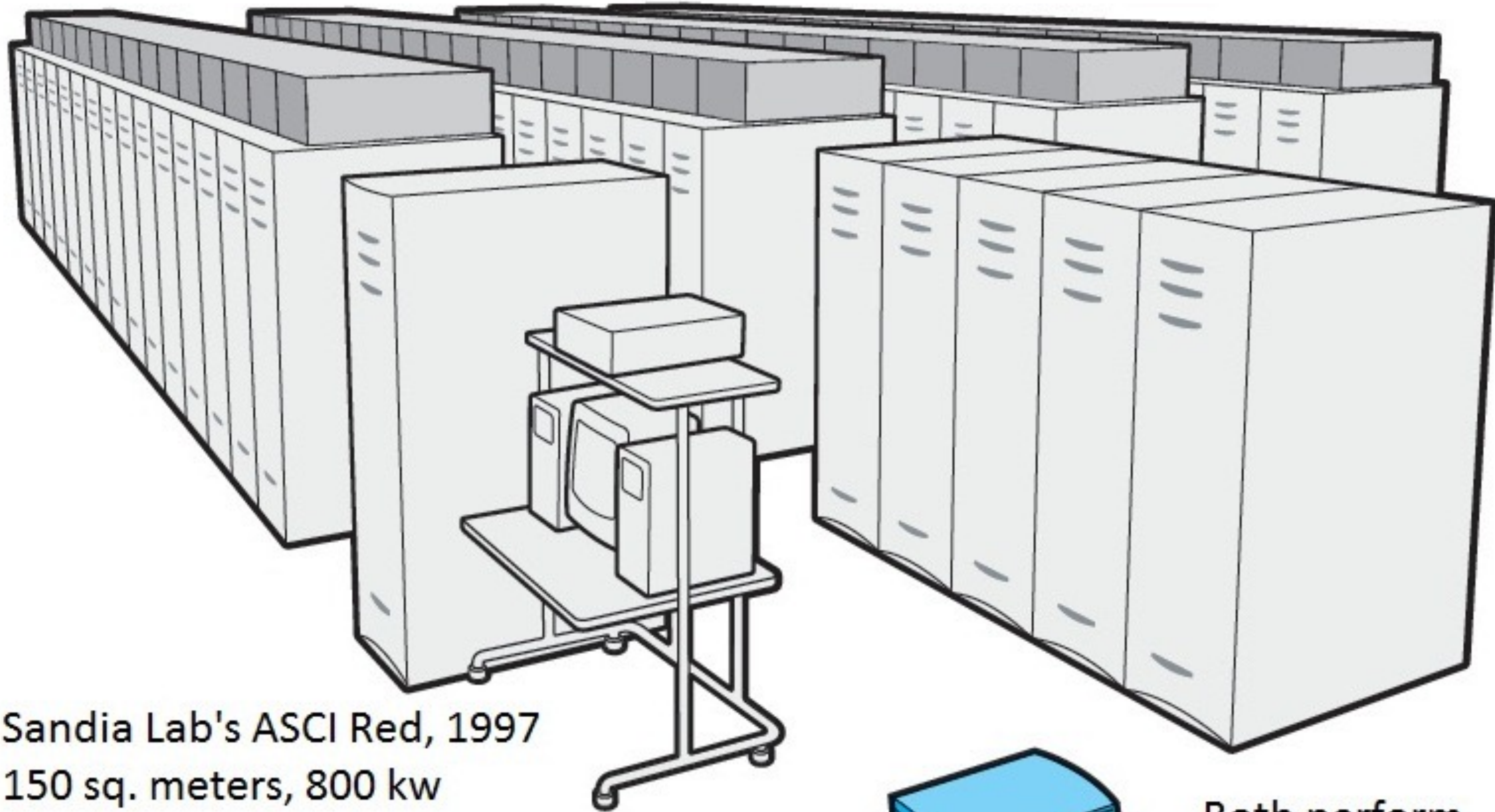
Primele trei supercalculatoare din lume (noiembrie 2021)



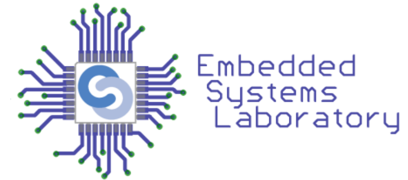
Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442,010.0	537,212.0	29,899
2	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
3	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438

The Shrinking Supercomputer

Embedded Systems Laboratory



Acknowledgements



- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)

- MIT material derived from course 6.823
- UCB material derived from course CS252