POLITECNICO MILANO 1863 | DIPARTIMENTO DI ELETTRONICA INFORMAZIONE E BIOINGEGNERIA

POLITECNICO MILANO 1863 NECST laboratory

Exercise Session 2

Pipelining, Static Branch Prediction, Dynamic Branch Prediction

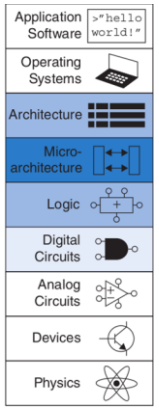
Advanced Computer Architectures

17th March 2025

Davide Conficconi <davide.conficconi@polimi.it>

POLITECNICO MILANO 1863 | DIPARTIMENTO DI ELETTRONICA INFORMAZIONE E BIOINGEGNERIA

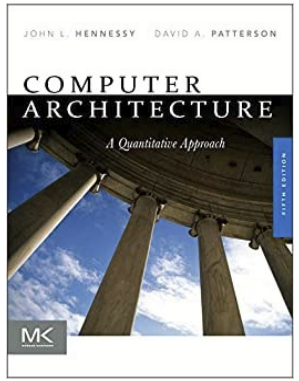
Recall: Material (EVERYTHING OPTIONAL)



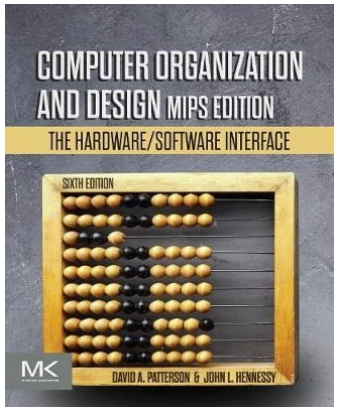
https://webeep.polimi.it/course/view.php?id=14754

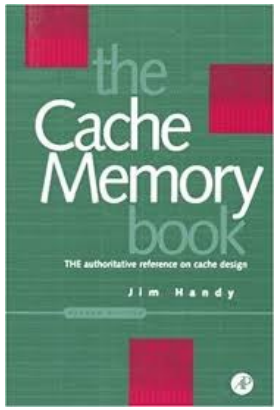
https://tinyurl.com/aca-grid25

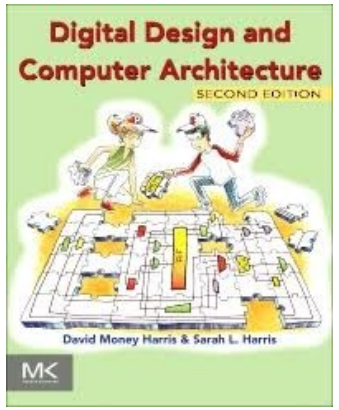
Textbook: Hennessy and Patterson, Computer Architecture: A Quantitative Approach

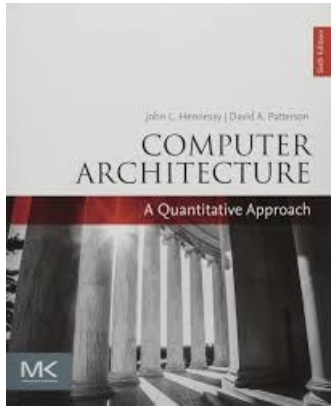


Other Interesting Reference

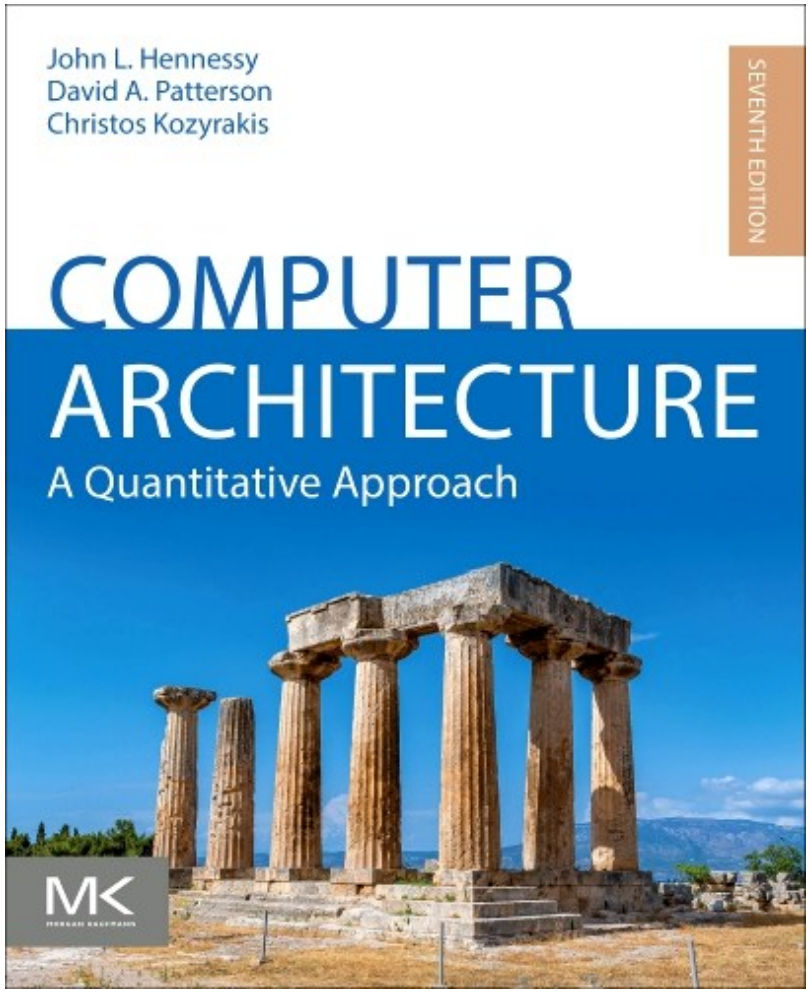


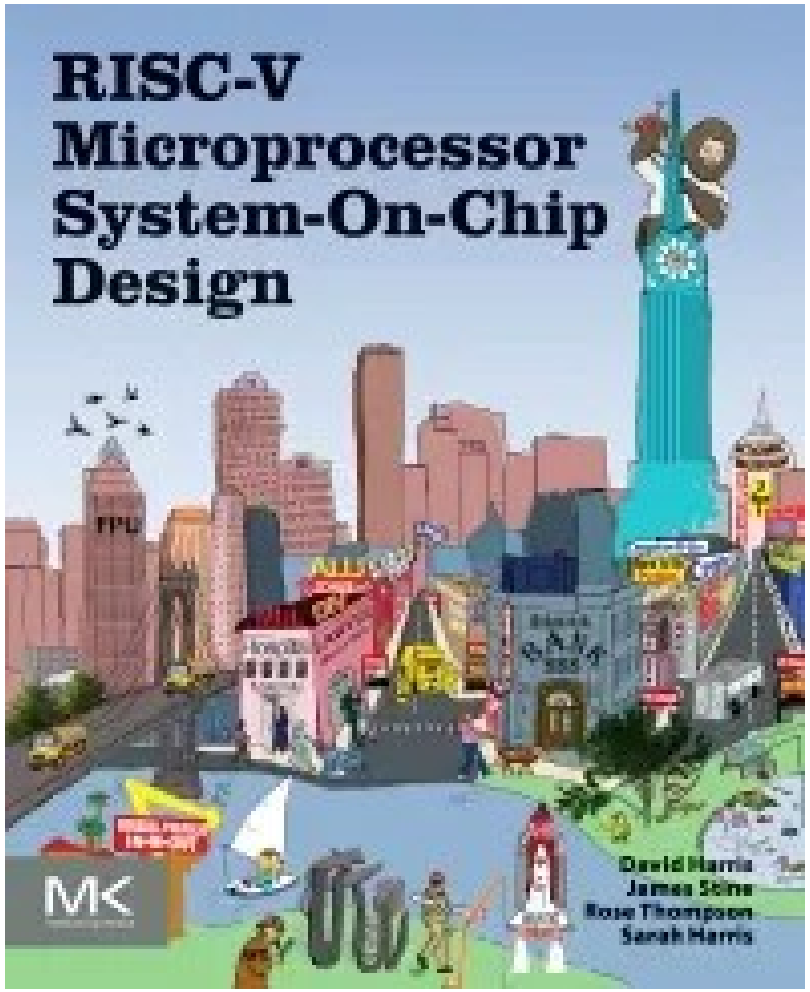






News from the outer world: new interesting books incoming!





News from the outer world: Championship Branch Prediction@ISCA2025

"the goal of this competition is to encourage researchers and practitioners to push the envelope in branch prediction."

ACM SIGARCH

https://www.sigarch.org/call-contributions/championship-branch-prediction-cbp2025-isca-2025/

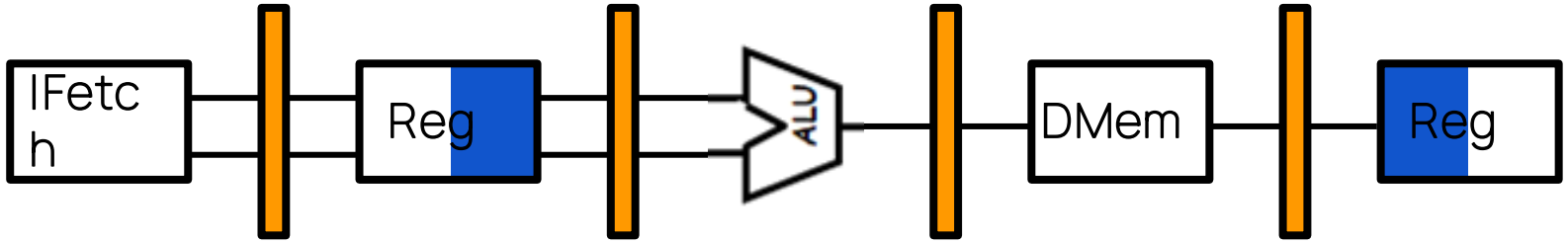
News from the outer world: LLMs use prediction (speculation) for decoding

Speculative decoding has proven to be an effective technique for faster and cheaper inference from LLMs without compromising quality.

It has also proven to be an effective paradigm for a range of optimization techniques.

https://research.google/blog/looking-back-at-speculative-decoding/

Recall: 5-Stage Pipelining



IF Instruction Fetch	ID Instruction Decode	EX Execution	ME Memory Access	WB Write Back

ALU Instructions: `op $x,$y,$z`

Instr. Fetch & PC Increm.	Read of Source Regs. `$y` and `$z`	ALU Op. (`$y op $z`)		Write Back Destinat. Reg. `$x`

Load Instructions: `lw $x,offset($y)`

Instr. Fetch & PC Increm.	Read of Base Reg. `$y`	ALU Op. (`$y+offset`)	Read Mem. `M($y+offset)`	Write Back Destinat. Reg. `$x`

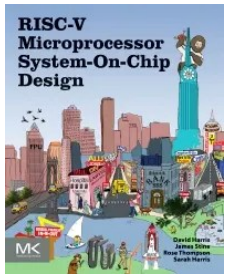
Store Instructions: `sw $x,offset($y)`

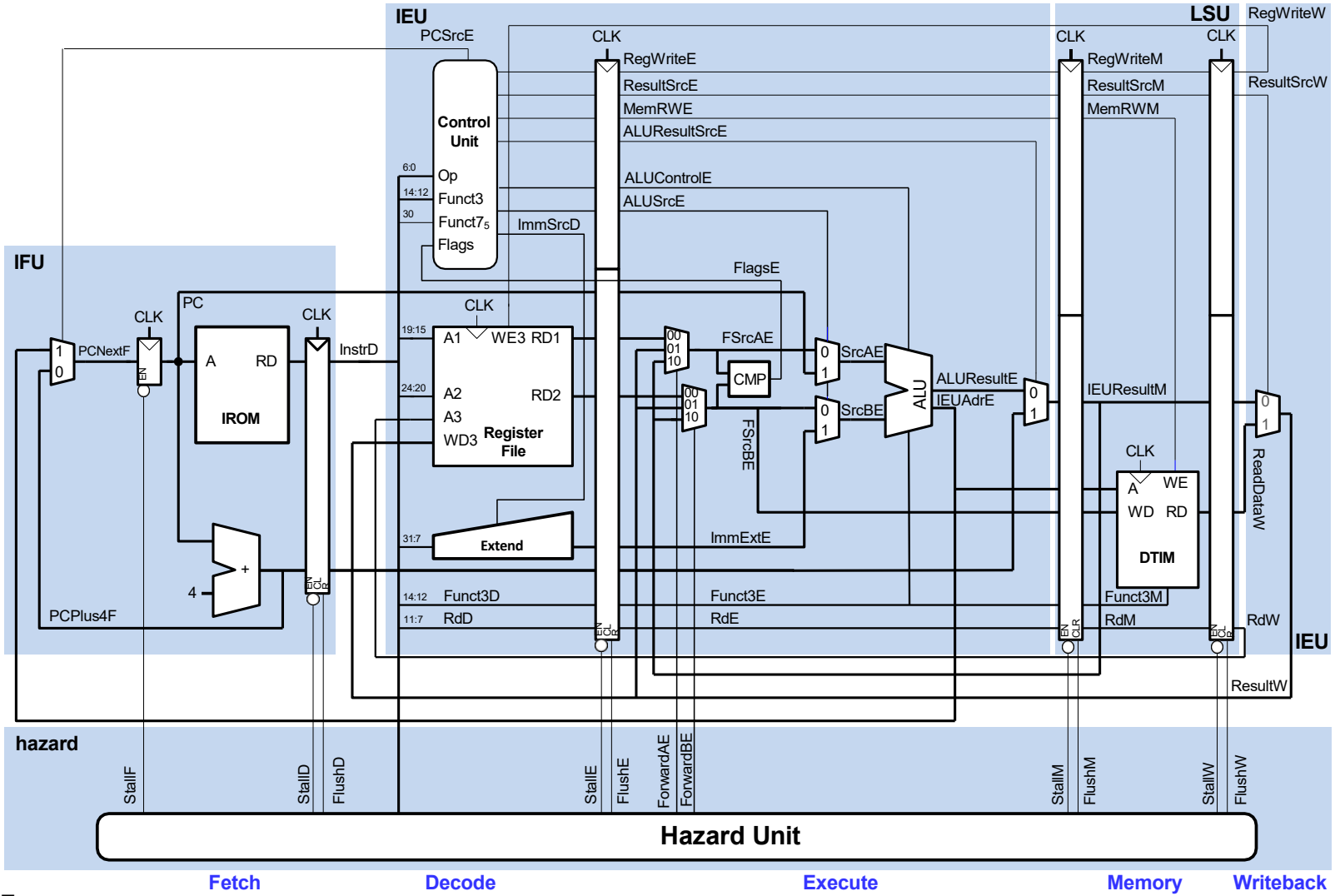
Instr. Fetch & PC Increm.	Read of Base Reg. `$y` & Source `$x`	ALU Op. (`$y+offset`)	Write Mem. `M($y+offset)`	

Conditional Branches: `beq $x,$y,offset`

Instr. Fetch & PC Increm.	Read of Source Regs. `$x` and `$y`	ALU Op. (`$x-$y`) & (`PC+4+offset`)	Write PC	

5-Stage Pipelining RISC-V Wally Educational core





Recall: Pipeline performance

Pipeline CPI = Ideal pipeline CPI + Structural Stalls + Data Hazard Stalls + Control Stalls

Ideal pipeline CPI: measure of the maximum performance attainable by the implementation

Structural hazards: HW cannot support this combination of instructions

Data hazards: Instruction depends on result of prior instruction still in the pipeline

Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches, jumps, exceptions)

Recall: Three Classes of Hazards

Structural Hazards: Attempt to use the same resource from different instructions simultaneously

Example: Single memory for instructions and data

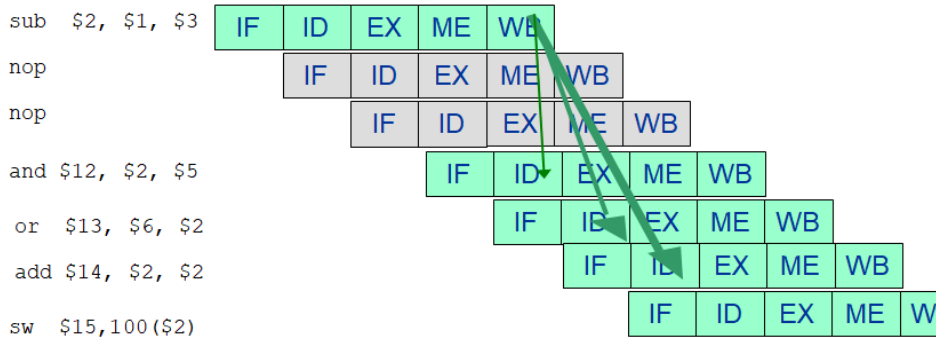
Data Hazards: Attempt to use a result before it is ready

Example: Instruction depending on a result of a previous instruction still in the pipeline

Control Hazards: Attempt to make a decision on the next instruction to execute before the condition is evaluated

Example: Conditional branch execution

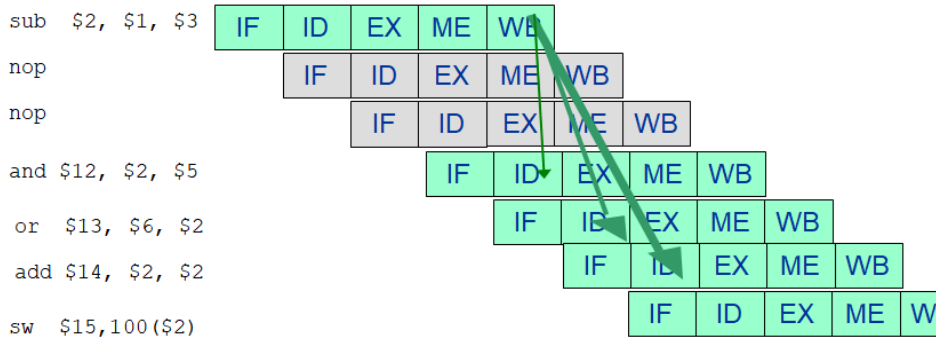
Recall: Data Hazards possible solutions




```
sub  $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15,100($2)
add $4, $10, $11
and $7, $8, $9
lw $16, 100($18)
lw $17, 200($19)
```

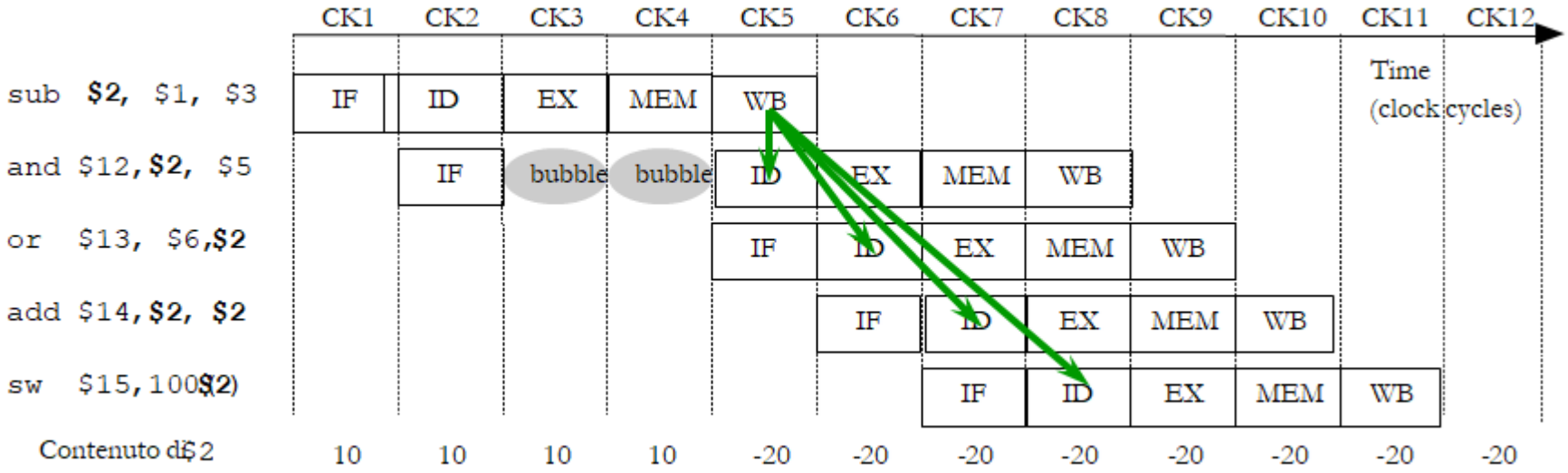
```
sub $2, $1, $3
add $4, $10, $11
and $7, $8, $9
lw $16, 100($18)
lw $17, 200($19)
and  $12, $2, $5
or $13, $6, $2
add  $14, $2, $2
sw $15,100($2)
```

Recall: Data Hazards possible solutions

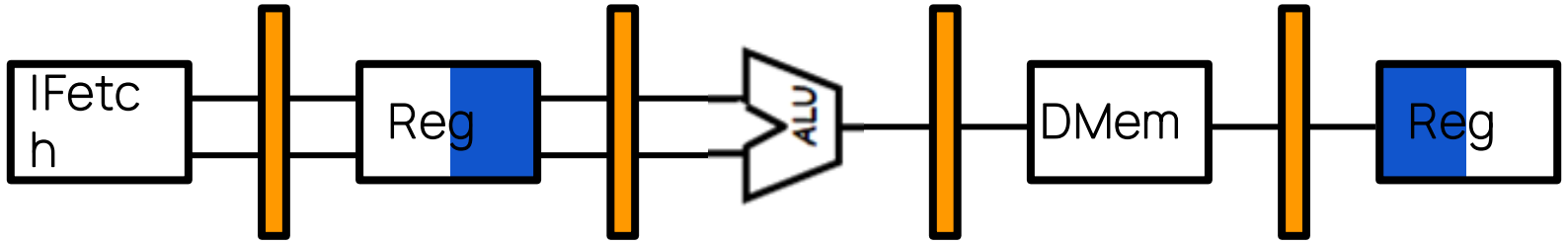


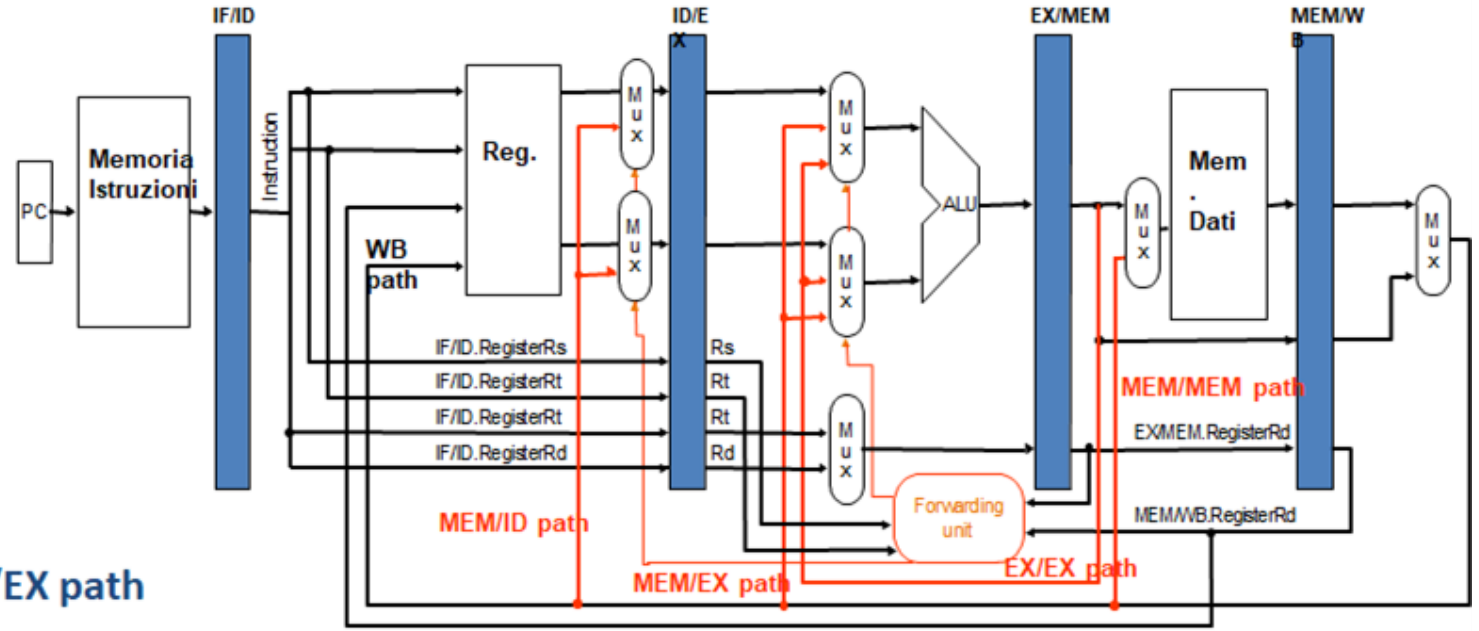

```
sub  $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15,100($2)
add $4, $10, $11
and $7, $8, $9
lw $16, 100($18)
lw $17, 200($19)
```

```
sub $2, $1, $3
add $4, $10, $11
and $7, $8, $9
lw $16, 100($18)
lw $17, 200($19)
and  $12, $2, $5
or $13, $6, $2
add  $14, $2, $2
sw $15,100($2)
```



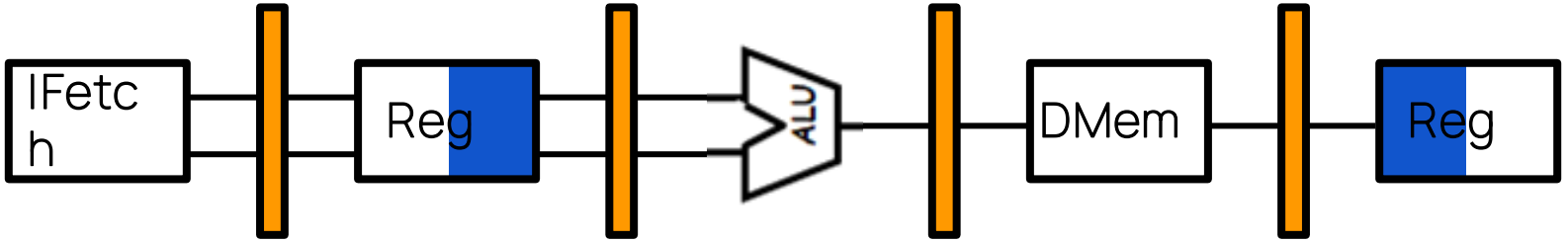
Recall: Pipelining and Forwarding



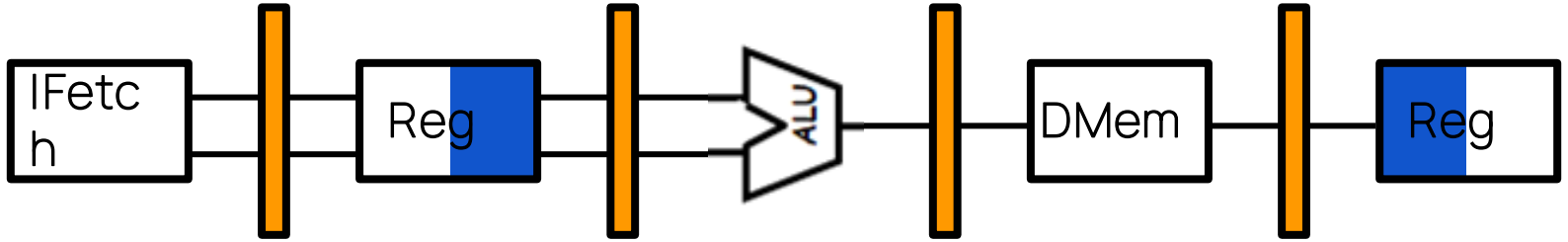


- EX/EX path
- MEM/EX path
- MEM/ID path
- MEM/MEM path

Exe : Pipelining

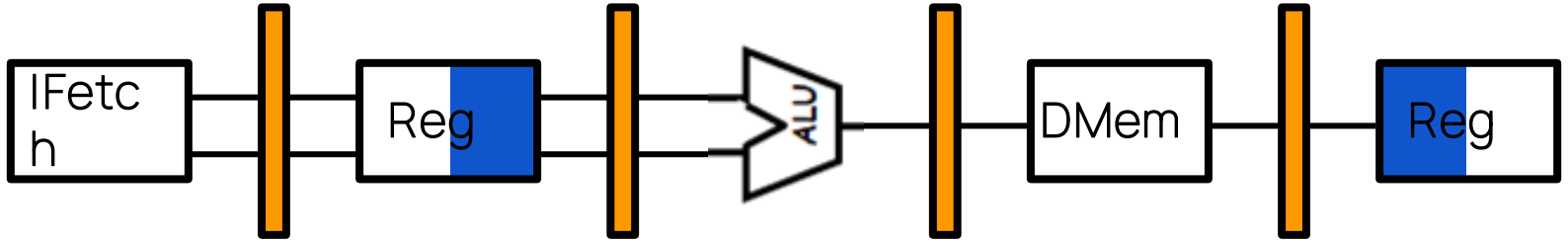


Exe : Pipelining



i1: add $t1, $t0, $t1
i2: add $t2, $t1, $t2
i3: subi $t0, $t2, 1
i4: sw $t0, 0x00BB($t2)
i5: beq $t0, $t2, 0x0089

Exe : Pipelining



i1: add $t1, $t0, $t1
i2: add $t2, $t1, $t2
i3: subi $t0, $t2, 1
i4: sw $t0, 0x00BB($t2)
i5: beq $t0, $t2, 0x0089

IF Instruction Fetch	ID Instruction Decode	EX Execution	ME Memory Access	WB Write Back

ALU Instructions: `op $x,$y,$z`

Instr. Fetch & PC Increm.	Read of Source Regs. `$y` and `$z`	ALU Op. (`$y op $z`)		Write Back Destinat. Reg. `$x`

Load Instructions: `lw $x,offset($y)`

Instr. Fetch & PC Increm.	Read of Base Reg. `$y`	ALU Op. (`$y+offset`)	Read Mem. `M($y+offset)`	Write Back Destinat. Reg. `$x`

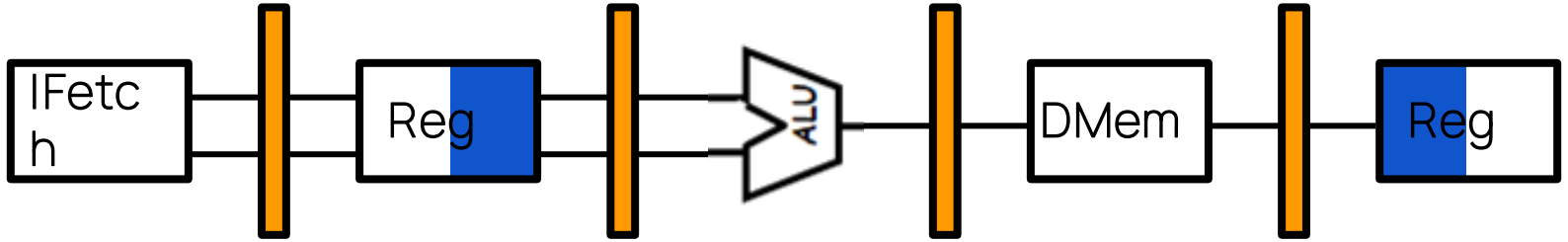
Store Instructions: `sw $x,offset($y)`

Instr. Fetch & PC Increm.	Read of Base Reg. `$y` & Source `$x`	ALU Op. (`$y+offset`)	Write Mem. `M($y+offset)`	

Conditional Branches: `beq $x,$y,offset`

Instr. Fetch & PC Increm.	Read of Source Regs. `$x` and `$y`	ALU Op. (`$x-$y`) & (`PC+4+offset`)	Write PC	

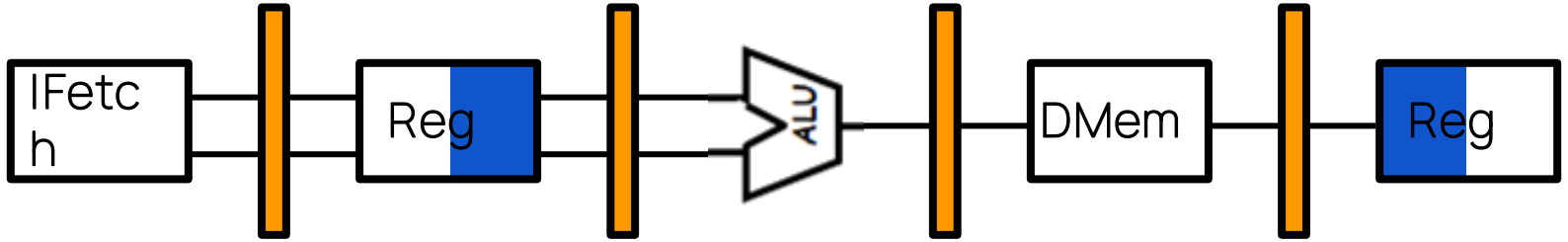
Exe : Pipelining



i1: add $t1, $t0, $t1
i2: add $t2, $t1, $t2
i3: subi $t0, $t2, 1
i4: sw $t0, 0x00BB($t2)
i5: beq $t0, $t2, 0x0089

- **No forwarding** paths
- RF access **R/W optimization**
- **Control** Hazard solved in **ID**

Exe : Pipelining



i1: add $t1, $t0, $t1
i2: add $t2, $t1, $t2
i3: subi $t0, $t2, 1
i4: sw $t0, 0x00BB($t2)
i5: beq $t0, $t2, 0x0089

- **No forwarding** paths
- RF access **R/W optimization**
- **Control** Hazard solved in **ID**

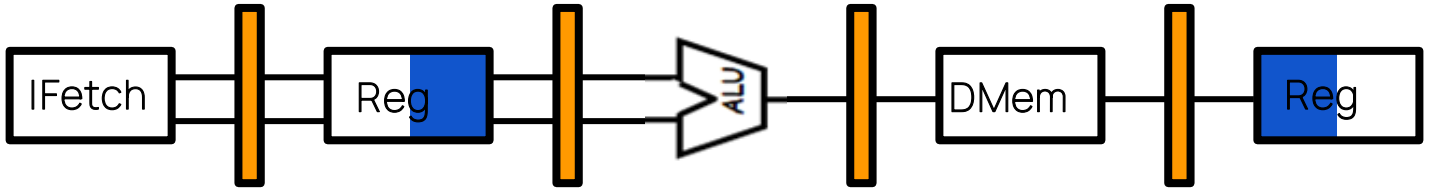
1) **Define all conflicts/dependencies**. For each of them indicate **whether** it causes an **hazard** and the **theoretical** amount of **stalls**
2) Draw the effective **pipeline schema**
3) Assuming EX/EX, MEM/EX, and MEM/MEM **forwarding paths** available + 2)
4) **Assuming** EX/ID + 3)

Exe.1 : Dependencies & Hazards

i1: add $t1, $t0, $t1
i2: add $t2, $t1, $t2
i3: subi $t0, $t2, 1
i4: sw $t0, 0x00BB($t2)
i5: beq $t0, $t2, 0x0089

Instr. #	Dependency on Instr. #	Register involved	Hazard (yes/no)	# of stalls (theoretical)

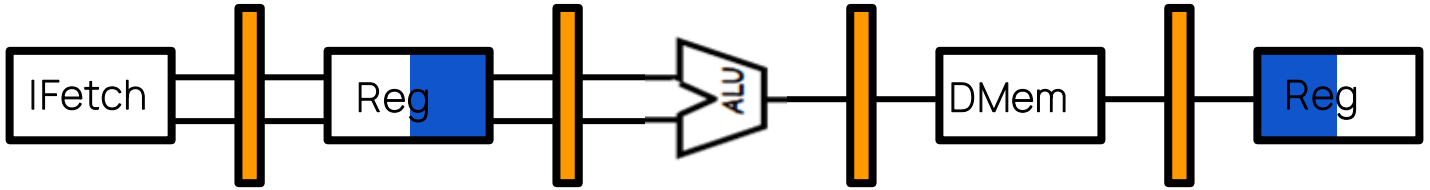
Exe.2 : Effective Pipeline Schema



CC 0

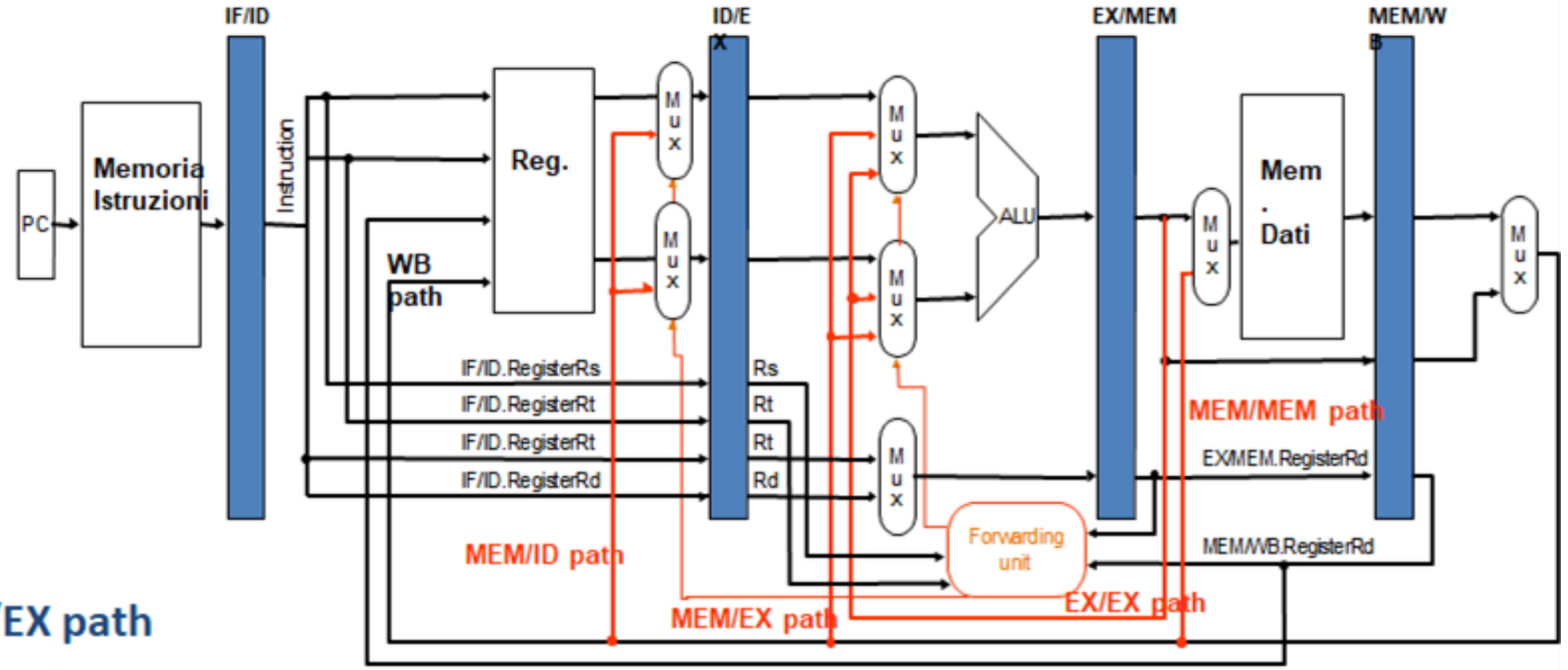
	Instruction	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16
1	add $t1, $t0, $t1																
2	add $t2, $t1, $t2																
3	subi $t0, $t2, 1																
4	sw $t0, 0x00BB($t2)																
5	beq $t0, $t2, 0x0089																

Exe.3 : Forwarding Paths



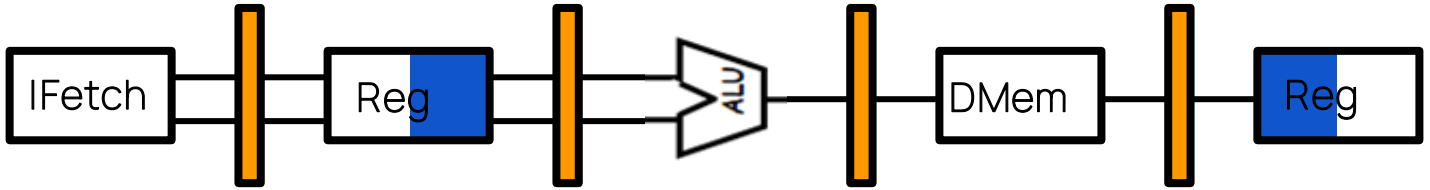
	Instruction	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16
1	add $t1, $t0, $t1																
2	add $t2, $t1, $t2																
3	subi $t0, $t2, 1																
4	sw $t0, 0x00BB($t2)																
5	beq $t0, $t2, 0x0089																

Recall MIPS with Forwarding



- EX/EX path
- MEM/EX path
- MEM/ID path
- MEM/MEM path

Exe.4 : Forwarding Paths + EXE/ID Path



	Instruction	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16
1	add $t1, $t0, $t1																
2	add $t2, $t1, $t2																
3	subi $t0, $t2, 1																
4	sw $t0, 0x00BB($t2)																
5	beq $t0, $t2, 0x0089																

Recall: Three Classes of Hazards

Structural Hazards: Attempt to use the same resource from different instructions simultaneously

Example: Single memory for instructions and data

Data Hazards: Attempt to use a result before it is ready

Example: Instruction depending on a result of a previous instruction still in the pipeline

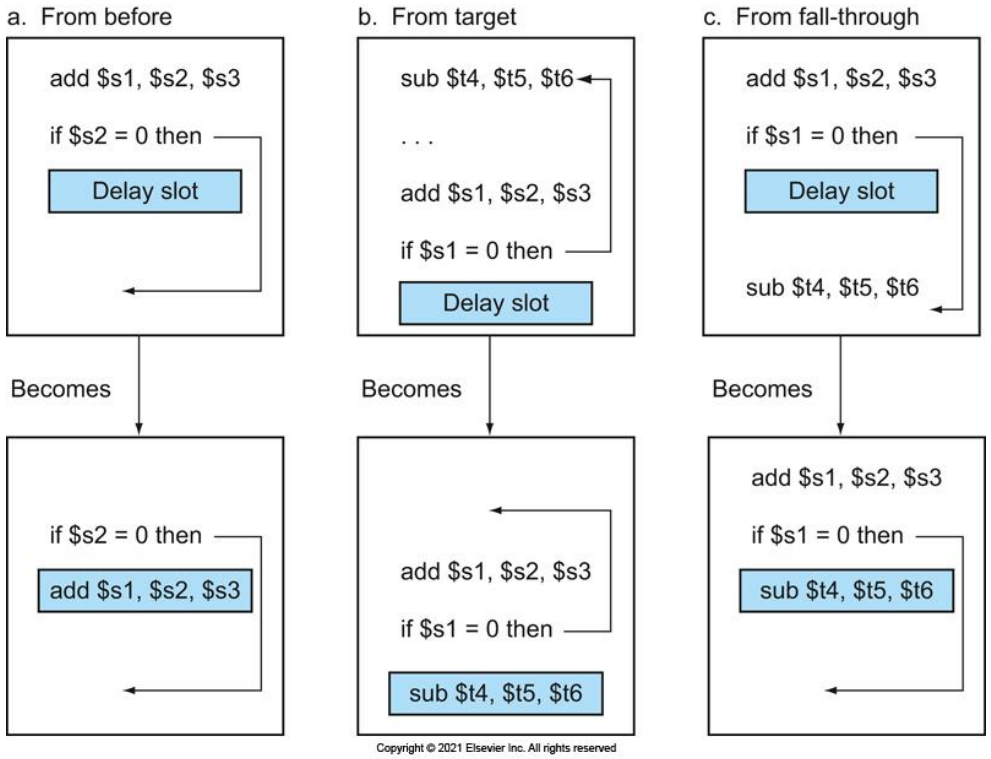
Control Hazards: Attempt to make a decision on the next instruction to execute before the condition is evaluated

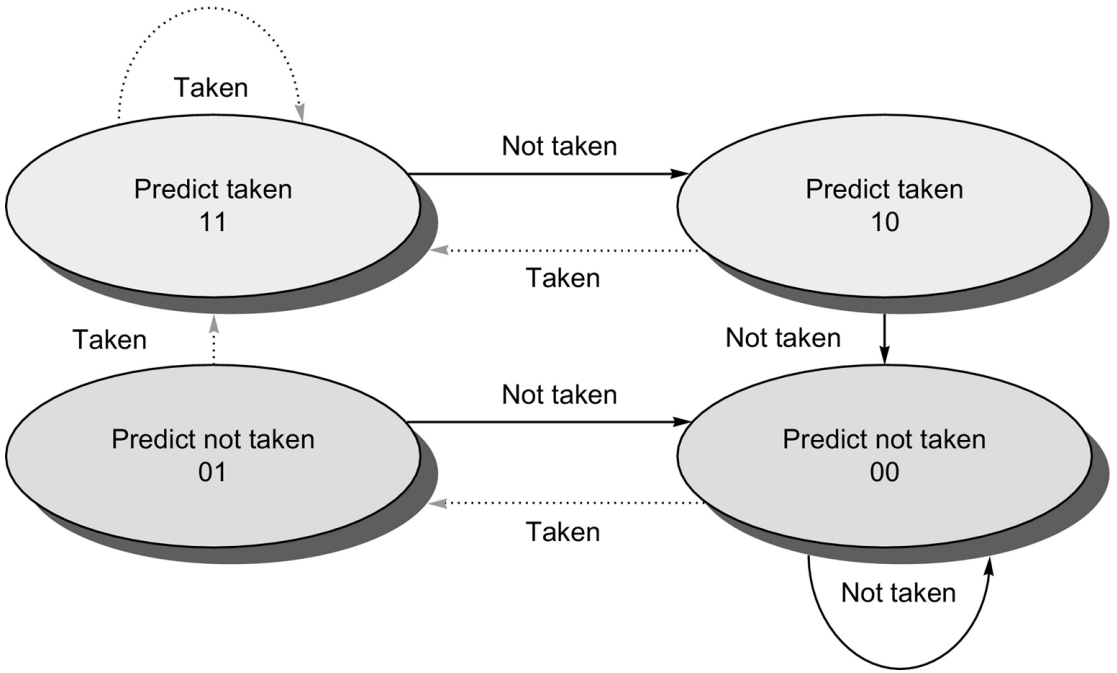
Example: Conditional branch execution

Prediction

[Branch vanguard: decomposing branch functionality into prediction and resolution instructions]

[The IBM z15 High Frequency Mainframe Branch Predictor]





Recall: Static Branch Prediction Techniques

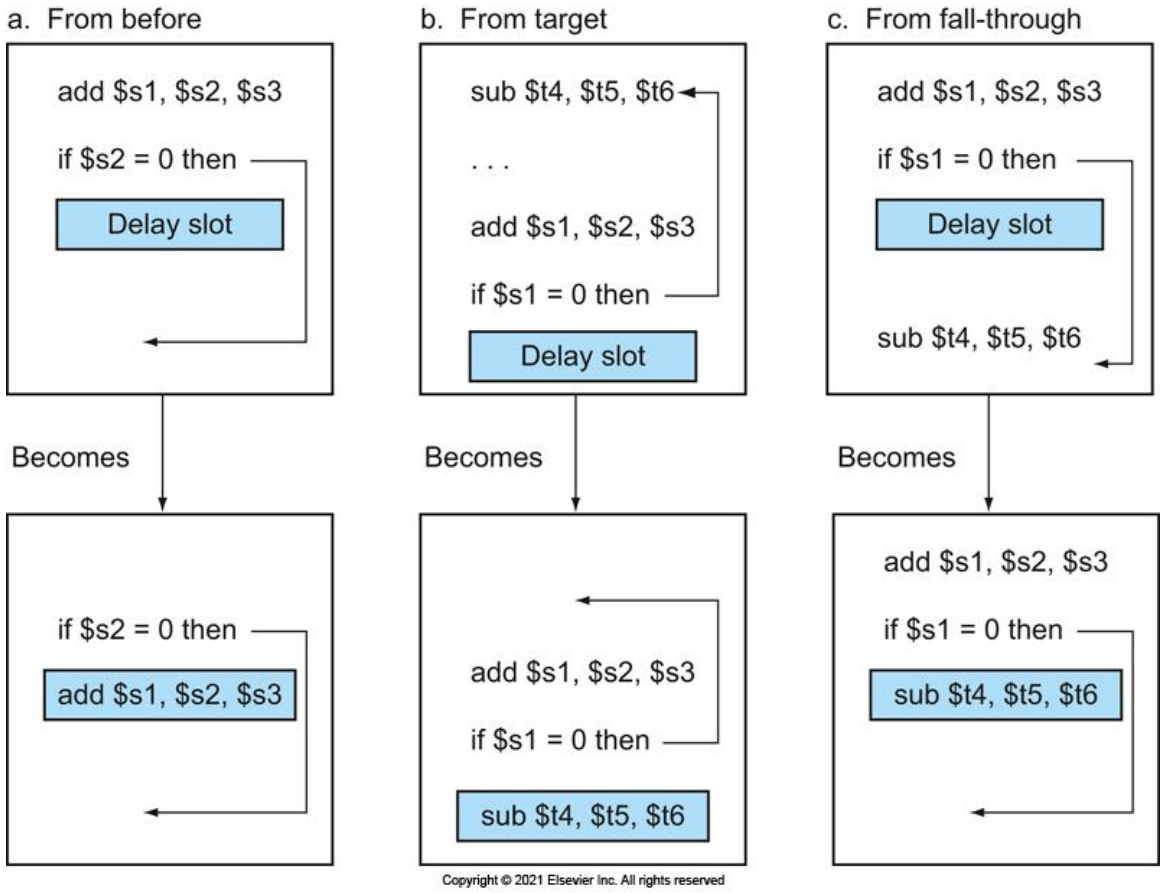
Branch Always Not Taken (Predicted-Not-Taken)

Branch Always Taken (Predicted-Taken)

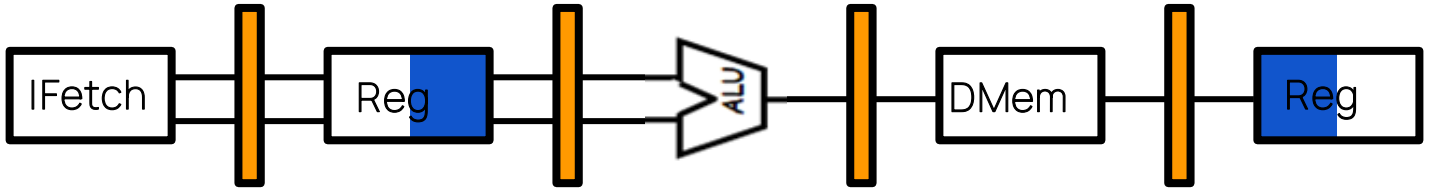
Backward Taken Forward Not Taken (BTFNT)

Profile-Driven Prediction

Delayed Branch



Exe.bonus : Pipeline Schema+Static BP



	Instruction	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16
1	add $t1, $t0, $t1																
2	add $t2, $t1, $t2																
3	subi $t0, $t2, 1																
4	sw $t0, 0x00BB($t2)																
5	beq $t0, $t2, 0x0089																
6	NEW INSTRUCTION																

Recall: Three Classes of Hazards

Structural Hazards: Attempt to use the same resource from different instructions simultaneously

Example: Single memory for instructions and data

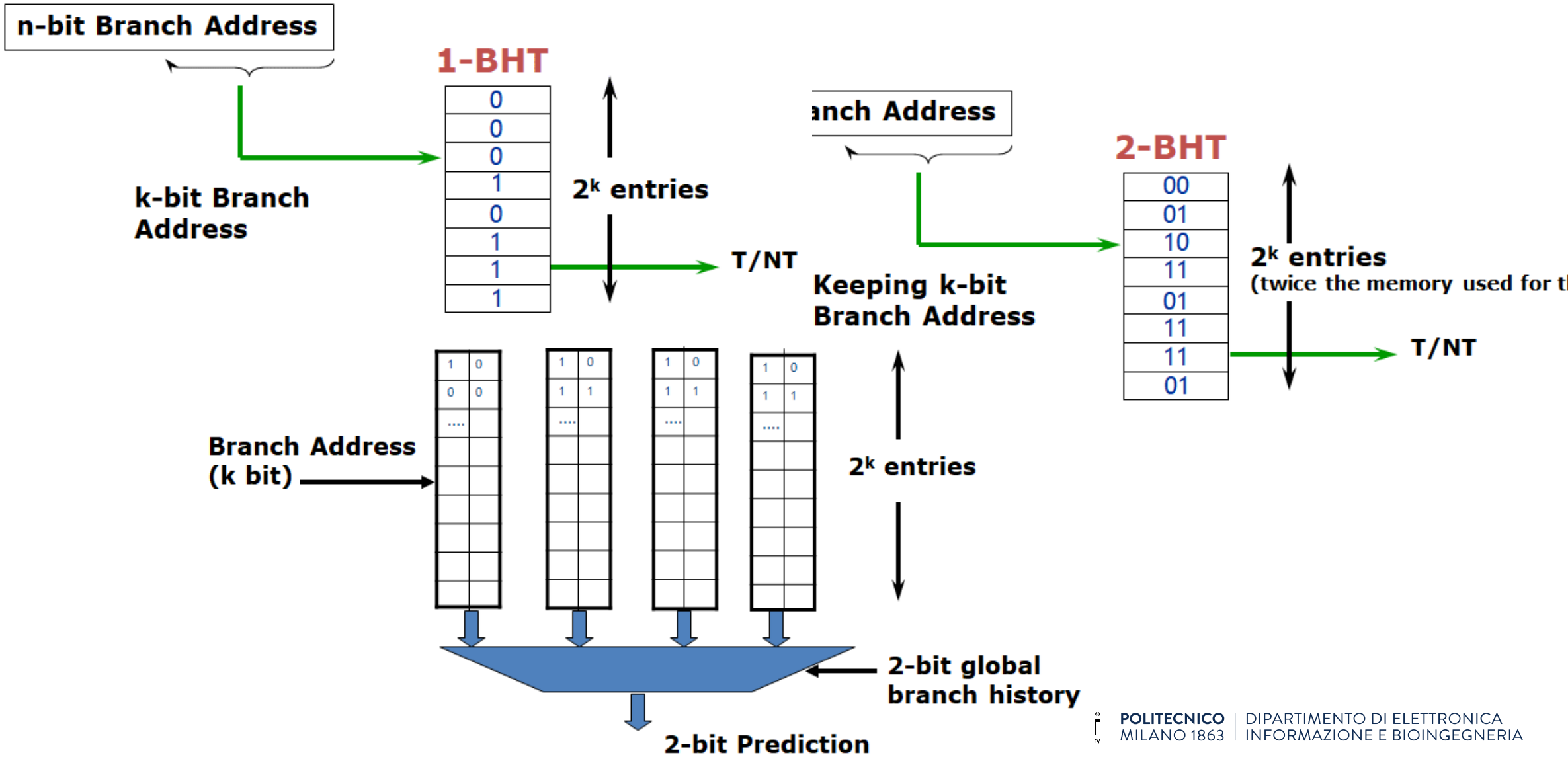
Data Hazards: Attempt to use a result before it is ready

Example: Instruction depending on a result of a previous instruction still in the pipeline

Control Hazards: Attempt to make a decision on the next instruction to execute before the condition is evaluated

Example: Conditional branch execution

Recall: Dynamic Branch Prediction



Dynamic Branch Predictor

- Describe (the answer has to be effectively supported) a 1-BHT and a 2-BHT able to execute the following assembly code (R0 is set to 2000, R1 is set to 0)

```
LOOP:   LD     F1     0      R0
        ADDD   F2     F1     F1
        ADDI   R1     R1     100
LOOP2:  MULTD  F2     F2     F1
        SUBI   R1     R1     1
        BNEZ   R1     LOOP2
        SUBI   R0     R0     2
        BNEZ   R0     LOOP
```

- The obtained result, in terms of mispredictions, is inline with theoretical characteristics of the two predictors? Please effectively support your answer.

A First Consideration

```
LOOP:   LD      F1   0    R0
        ADDD    F2   F1   F1
        ADDI    R1   R1   100
LOOP2:  MULTD   F2   F2   F1
        SUBI    R1   R1   1
        BNEZ    R1   LOOP2
        SUBI    R0   R0   2
        BNEZ    R0   LOOP
```

How many iterations?

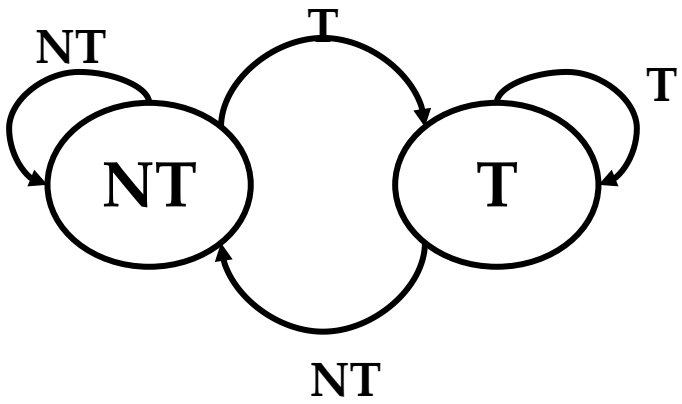
```
LOOP:   LD      F1   0    R0
        ADDD    F2   F1   F1
        ADDI    R1   R1   100
LOOP2:  MULTD   F2   F2   F1
        SUBI    R1   R1   1
        BNEZ    R1   LOOP2
        SUBI    R0   R0   2
        BNEZ    R0   LOOP
```

R0 is set to 2000
R1 is set to 0

1bit - BHT

```
LOOP:   LD      F1      0       R0
        ADDD    F2      F1      F1
        ADDI    R1      R1      100
LOOP2:  MULTD   F2      F2      F1
        SUBI    R1      R1      1
        BNEZ    R1      LOOP2
        SUBI    R0      R0      2
        BNEZ    R0      LOOP
```

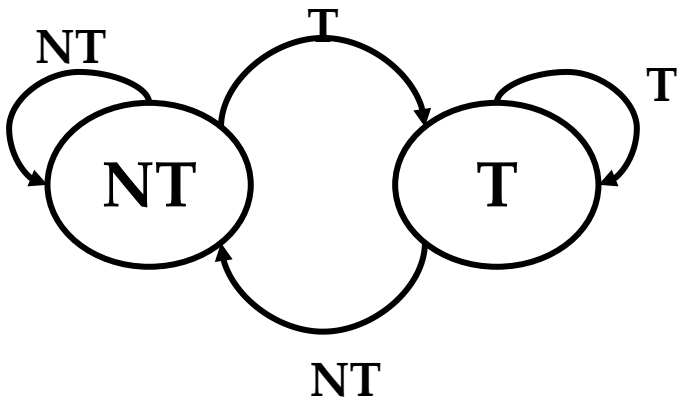
R0 is set to 2000
R1 is set to 0

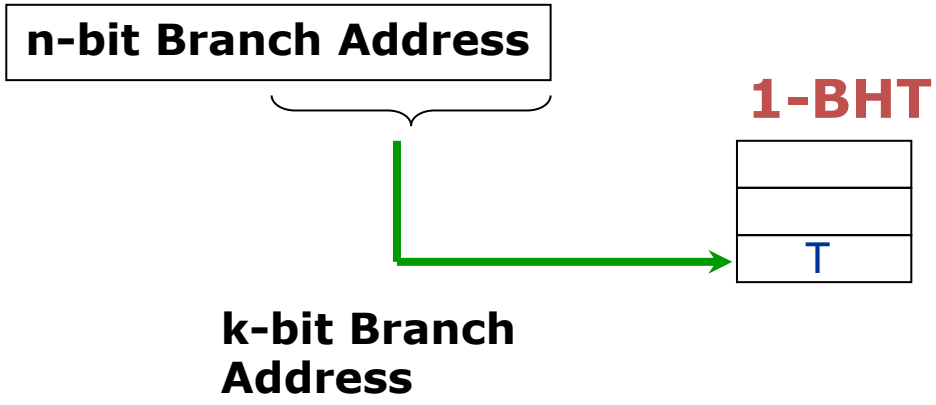


1bit - BHT

```
LOOP:   LD      F1      0       R0
        ADDD    F2      F1      F1
        ADDI    R1      R1      100
LOOP2:  MULTD   F2      F2      F1
        SUBI    R1      R1      1
        BNEZ    R1      LOOP2
        SUBI    R0      R0      2
        BNEZ    R0      LOOP
```

R0 is set to 2000
R1 is set to 0



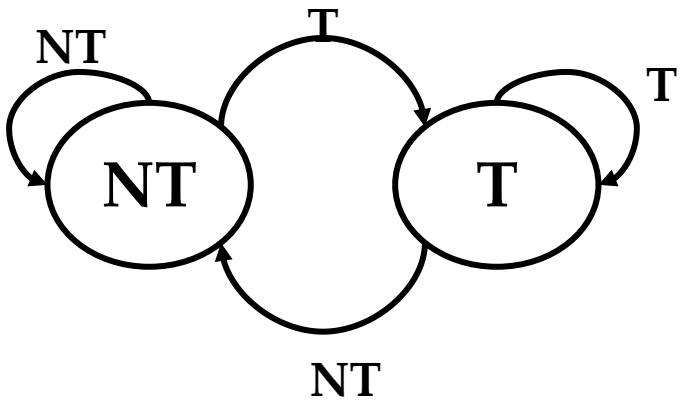


k-bit Branch Address:
Collide
Not collide

1bit - BHT - Not Collide

```
LOOP:   LD      F1      0       R0
        ADDD    F2      F1      F1
        ADDI    R1      R1      100
LOOP2:  MULTD   F2      F2      F1
        SUBI    R1      R1      1
        BNEZ    R1      LOOP2
        SUBI    R0      R0      2
        BNEZ    R0      LOOP
```

R0 is set to 2000
R1 is set to 0



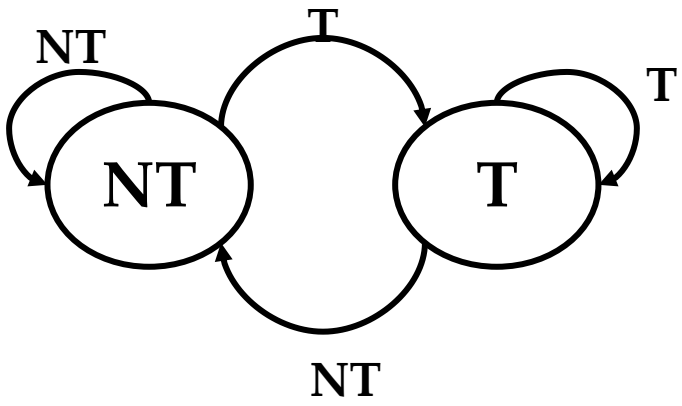
Let us consider that the branch addresses do not collide

	1-BHT	1-BHT	1-BHT	1-BHT
LOOP:	T	T	NT	NT
LOOP2:	T	NT	T	NT

1bit - BHT - Not Collide

```
LOOP:   LD      F1      0       R0
        ADDD    F2      F1      F1
        ADDI    R1      R1      100
LOOP2:  MULTD   F2      F2      F1
        SUBI    R1      R1      1
        BNEZ    R1      LOOP2
        SUBI    R0      R0      2
        BNEZ    R0      LOOP
```

R0 is set to 2000
R1 is set to 0



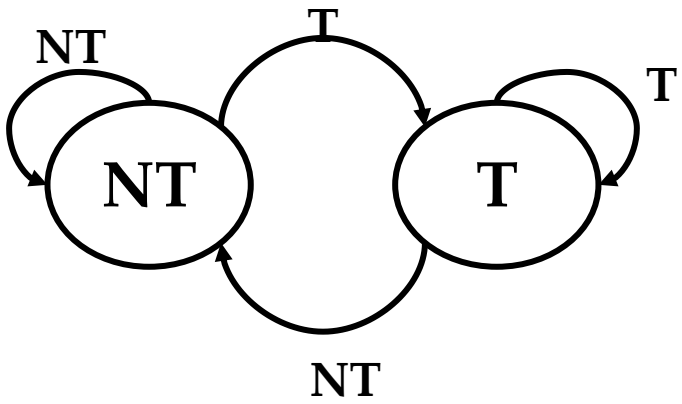
Let us consider that the branch addresses do not collide

	1-BHT	1-BHT	1-BHT	1-BHT
LOOP:	T	T	NT	NT
LOOP2:	T	NT	T	NT

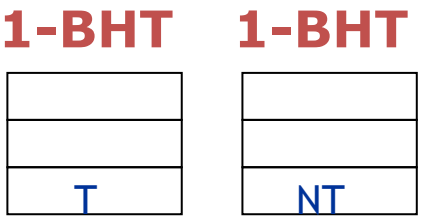
1bit - BHT - Collision

```
LOOP:   LD      F1      0       R0
        ADDD    F2      F1      F1
        ADDI    R1      R1      100
LOOP2:  MULTD   F2      F2      F1
        SUBI    R1      R1      1
        BNEZ    R1      LOOP2
        SUBI    R0      R0      2
        BNEZ    R0      LOOP
```

R0 is set to 2000
R1 is set to 0



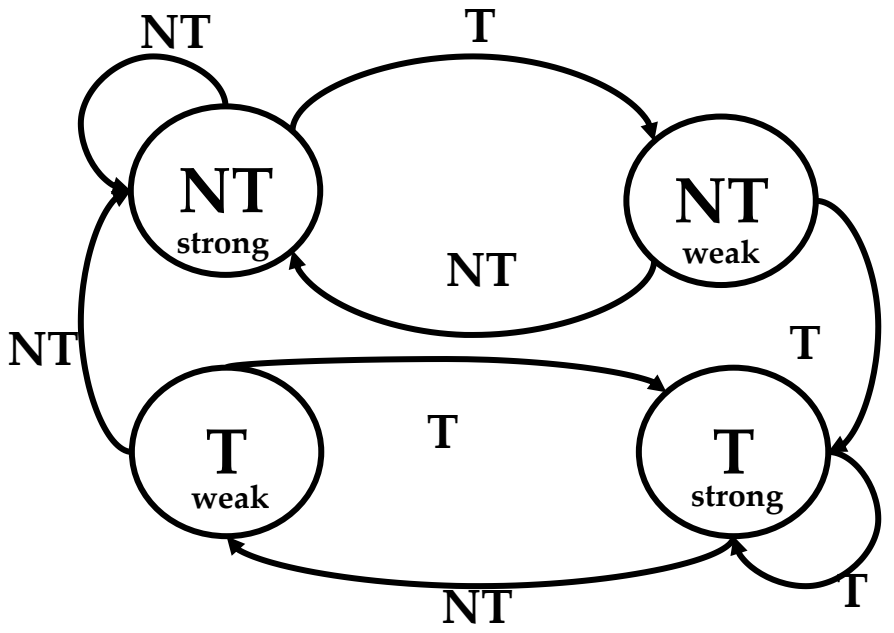
Let us consider that the branch addresses do collide



2bit - BHT

R0 is set to 2000
R1 is set to 0

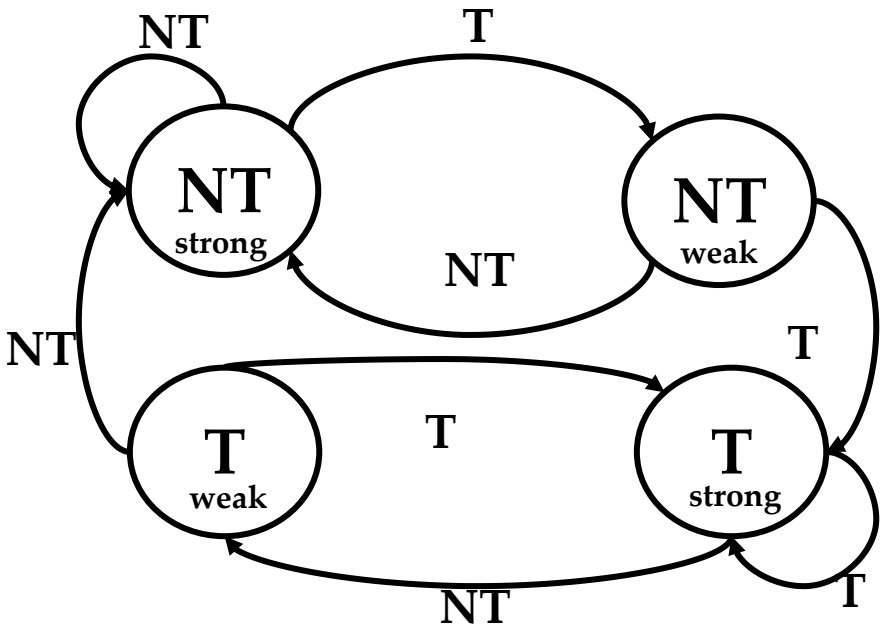
```
LOOP:   LD      F1      0       R0
        ADDD    F2      F1      F1
        ADDI    R1      R1      100
LOOP2:  MULTD   F2      F2      F1
        SUBI    R1      R1      1
        BNEZ    R1      LOOP2
        SUBI    R0      R0      2
        BNEZ    R0      LOOP
```



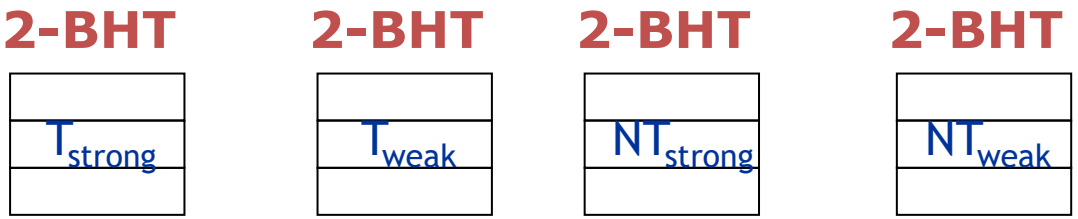
2bit - BHT

R0 is set to 2000
R1 is set to 0

```
LOOP:   LD      F1      0       R0
        ADDD    F2      F1      F1
        ADDI    R1      R1      100
LOOP2:  MULTD   F2      F2      F1
        SUBI    R1      R1      1
        BNEZ    R1      LOOP2
        SUBI    R0      R0      2
        BNEZ    R0      LOOP
```



Let us consider that the branch addresses do collide

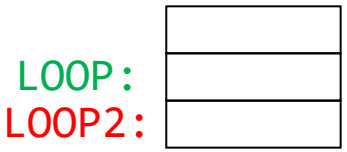


SUMMARY

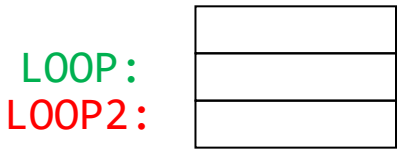
Assumption: NO collision

WORST CASES

1-BHT

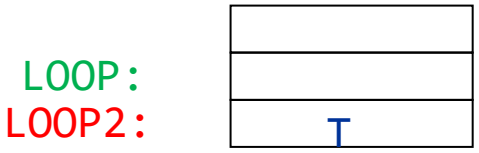


2-BHT

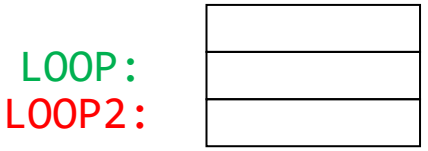


BEST CASES

1-BHT



2-BHT



Thanks for your attention

Davide Conficconi <davide.conficconi@polimi.it>

Acknowledgements

E. Del Sozzo, Marco D. Santambrogio, D. Sciuto

Part of this material comes from:

- “Computer Organization and Design” and “Computer Architecture A Quantitative Approach” Patterson and Hennessy books
- News and paper cited throughout the lecture