ECE 552: Introduction to Computer Architecture

15. Pipelining, Hazards & Exceptions #3

Lecture notes based in part on slides created by Joshua San Miguel, Mikko Lipasti, Mark Hill, David Wood, Guri Sohi, John Shen, and Jim Smith

October 23, 2025

George Tzimpragos

Refresher 1/3

- Pipeline function for ADD (R-type):
  - Fetch: read instruction from memory
  - Decode: **read source operands from reg**
  - Execute: calculate sum
  - Memory: pass results to next stage
  - Writeback: **write sum into register file**

```
add x1, x2, x3 # write to x1
add x4, x1, x5 # read from x1
```

* RAW data dependency

Refresher 2/3

- Data Dependency: *one instruction uses the result of a previous one*
  - Doesn't necessarily cause a problem (e.g., WAR, WAW)
- Data Hazard: *one instruction has a data dependency that will cause a problem if we don't "deal with it"*

Refresher 3/3

- ❑ Avoid
  - ➢Make sure there are no hazards in the code
- ❑ **Detect and Stall**
  - ➢If hazards exist, stall the processor until they go away.
- ❑ **Detect and Forward**
  - ➢If hazards exist, fix up the pipeline to get the correct value (if possible)

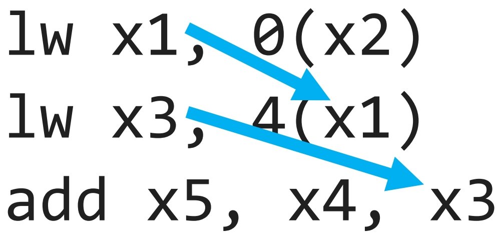
Exercise

```
lw x1, 0(x2)
lw x3, 4(x1)
add x5, x4, x3
```

(a) Assume **no RF bypassing** and **no forwarding**
(b) Assume **RF bypassing** but **no forwarding**
(c) Assume **RF bypassing** and **forwarding**

inst\cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
lw x1	F	D	X	M	W								

Exercise



```
lw x1, 0(x2)
lw x3, 4(x1)
add x5, x4, x3
```

(a) **No RF bypassing** and **no forwarding**

inst\cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
lw x1	F	D	X	M	W								
lw x3		F	D*	D*	D*	D	X	M	W				
add			F*	F*	F*	F	D*	D*	D*	D	X	M	W

Exercise

```
lw x1, 0(x2)
lw x3, 4(x1)
add x5, x4, x3
```

(b) **RF bypassing** but **no forwarding**

inst\cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
lw x1	F	D	X	M	W								
lw x3		F	D*	D*	D	X	M	W					
add			F*	F*	F	D*	D*	D	X	M	W		

Exercise

```
lw x1, 0(x2) # Result available after the MEM stage
lw x3, 4(x1) # The result of I1 is needed during the EX stage
add x5, x4, x3
```

(c) **RF bypassing** and **forwarding**

inst\cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
lw x1	F	D	X	M	W								
lw x3		F	D*	D	X	M	W						
add			F*	F	D*	D	X	M	W				

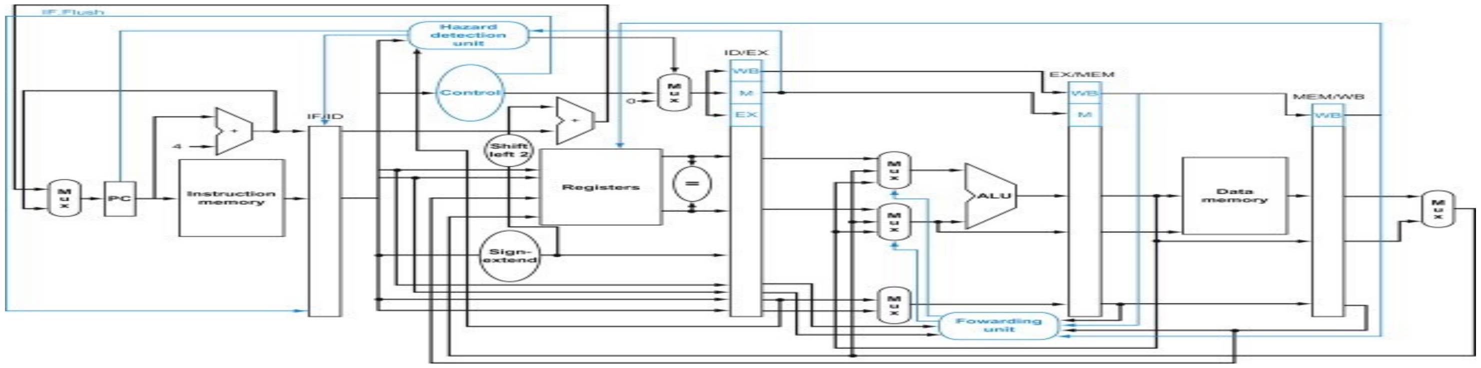
MEM-to-EX forwarding

MEM-to-EX forwarding

Exercise

inst\cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
lw x1	F	D	X	M	W								
nop				X	M	W							
lw x3		F	D*	D	X	M	W						
nop						X	M	W					
add			F*	F	D*	D	X	M	W				

cycle 3:



add

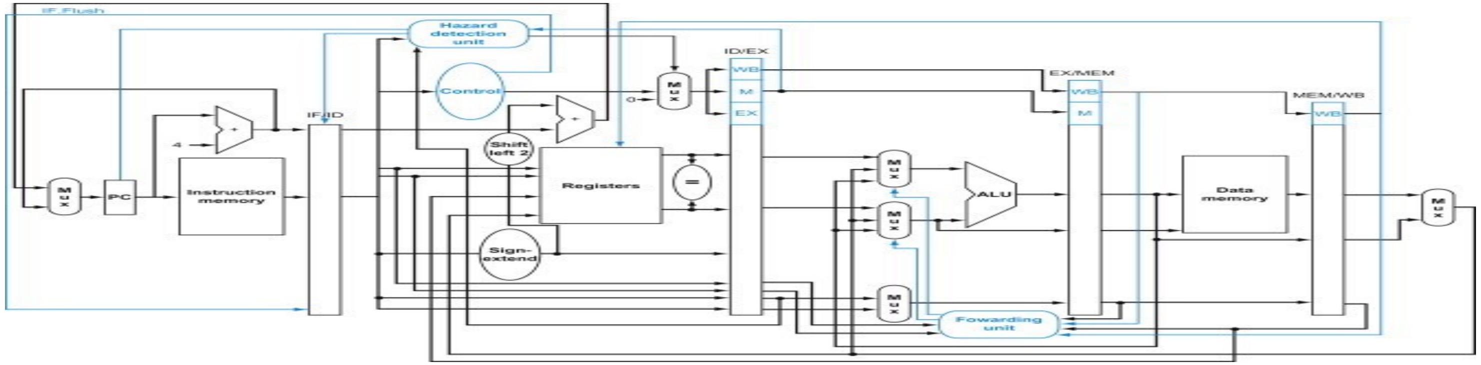
lw x3

lw x1

Exercise

inst\cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
lw x1	F	D	X	M	W								
nop				X	M	W							
lw x3		F	D*	D	X	M	W						
nop						X	M	W					
add			F*	F	D*	D	X	M	W				

cycle 4:

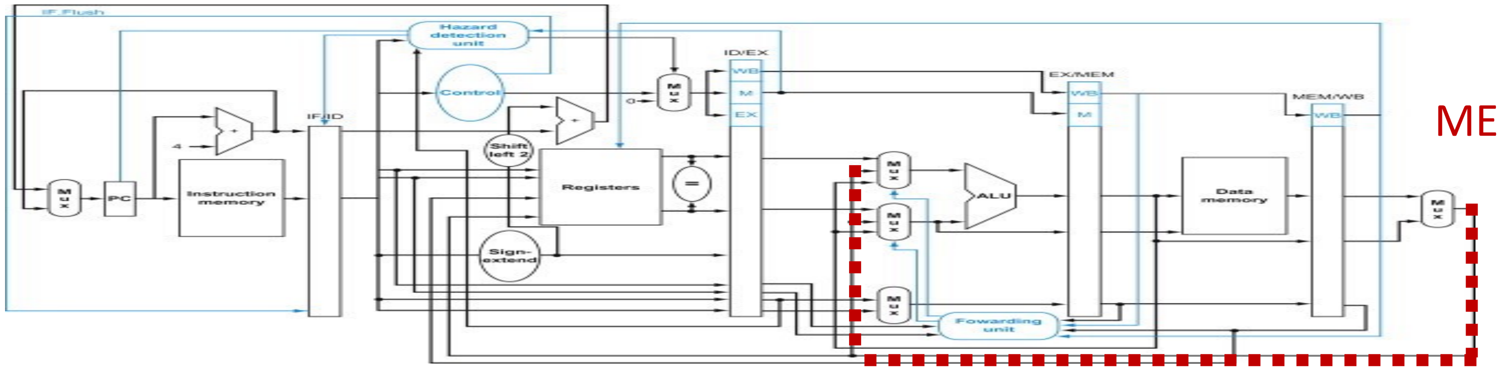


add lw x3 nop lw x1

Exercise

inst\cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
lw x1	F	D	X	M	W								
nop				X	M	W							
lw x3		F	D*	D	X	M	W						
nop						X	M	W					
add			F*	F	D*	D	X	M	W				

cycle 5:



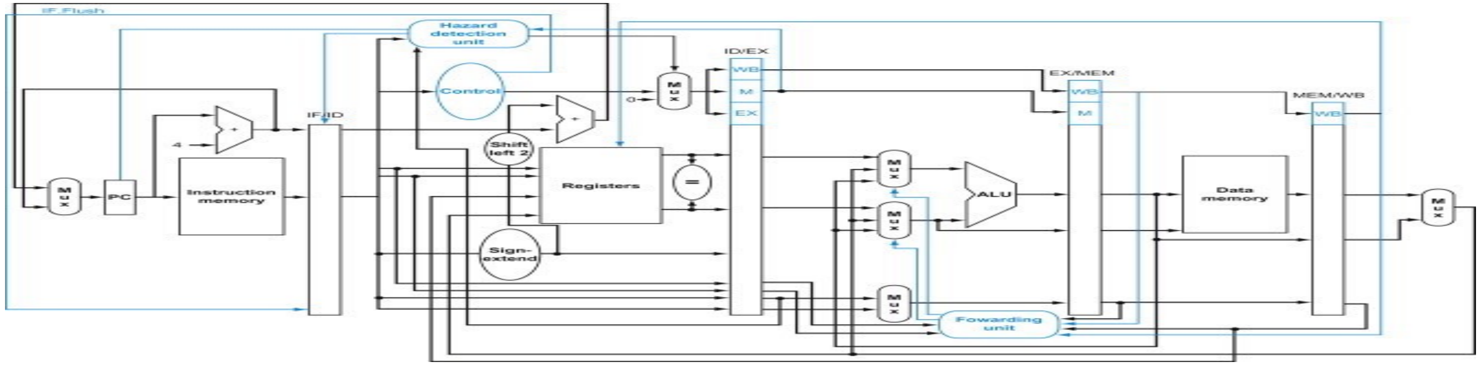
MEM-to-EX forwarding

add lw x3 nop lw x1

Exercise

inst\cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
lw x1	F	D	X	M	W								
nop				X	M	W							
lw x3		F	D*	D	X	M	W						
nop						X	M	W					
add			F*	F	D*	D	X	M	W				

cycle 6:

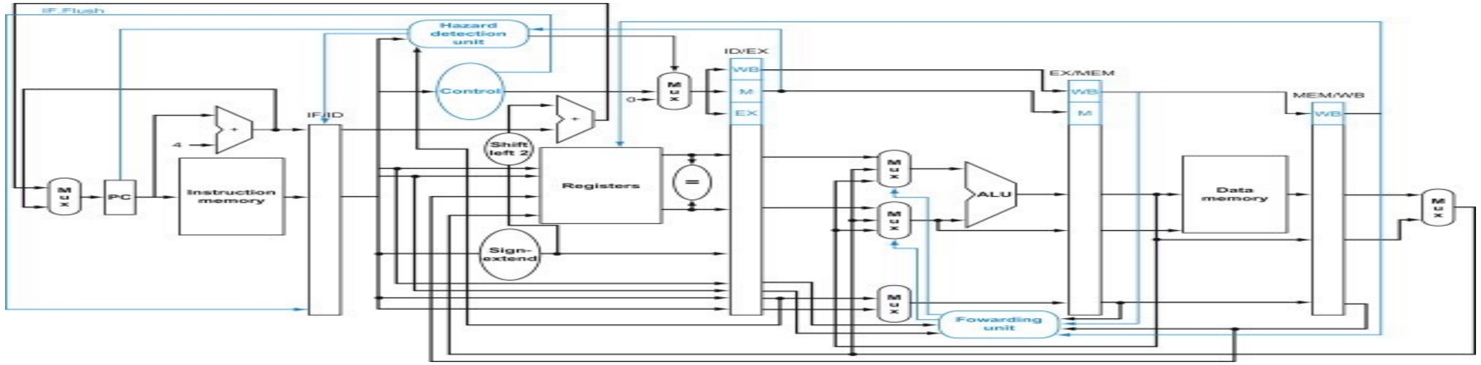


add nop lw x3 nop

Exercise

inst\cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
lw x1	F	D	X	M	W								
nop				X	M	W							
lw x3		F	D*	D	X	M	W						
nop						X	M	W					
add			F*	F	D*	D	X	M	W				

cycle 7:



add nop lw x3

One more exercise

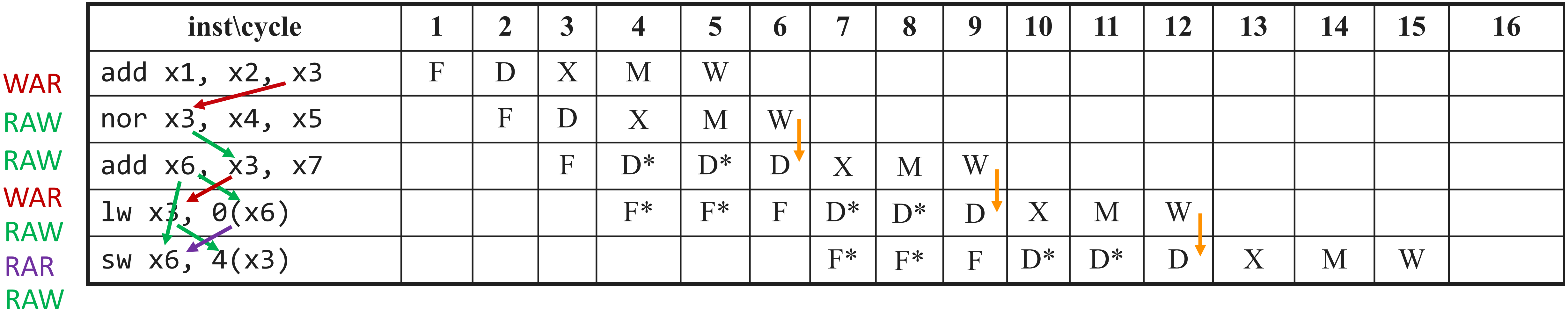
(a) Assume **RF bypassing** but **no forwarding**
(b) Assume **RF bypassing** and **forwarding**
* Annotate all RF bypassing and forwarding as appropriate

inst\cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
`add x1, x2, x3`	F	D	X	M	W											
`nor x3, x4, x5`																
`add x6, x3, x7`																
`lw x3, 0(x6)`																
`sw x6, 4(x3)`																

One more exercise

(a) Assume **RF bypassing** but **no forwarding**

* Annotate all RF bypassing and forwarding as appropriate

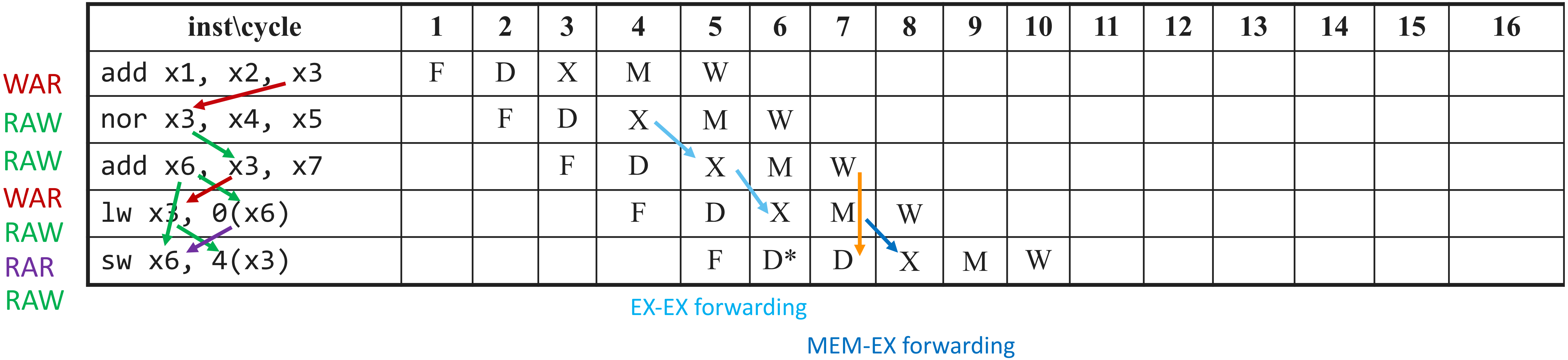


inst\cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
`add x1, x2, x3`	F	D	X	M	W											
`nor x3, x4, x5`		F	D	X	M	W										
`add x6, x3, x7`			F	D*	D*	D	X	M	W							
`lw x3, 0(x6)`				F*	F*	F	D*	D*	D	X	M	W				
`sw x6, 4(x3)`							F*	F*	F	D*	D*	D	X	M	W	

One more exercise

(b) Assume **RF bypassing** and **forwarding**

* Annotate all RF bypassing and forwarding as appropriate



WAR
RAW
RAW
WAR
RAW
RAR
RAW

inst\cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
`add x1, x2, x3`	F	D	X	M	W											
`nor x3, x4, x5`		F	D	X	M	W										
`add x6, x3, x7`			F	D	X	M	W									
`lw x3, 0(x6)`				F	D	X	M	W								
`sw x6, 4(x3)`					F	D*	D	X	M	W						

EX-EX forwarding

MEM-EX forwarding

Control dependencies

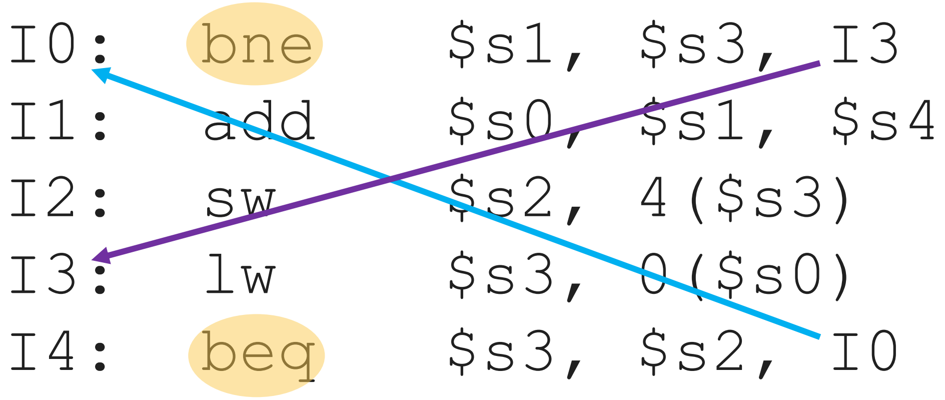
- An instruction Y is control-dependent on an instruction X if the outcome of X determines whether or not Y will be executed
  - Due to conditional branches:
    - Branch must execute to determine which instruction to fetch next
    - Instructions following a conditional branch are control-dependent on the branch instruction

Control dependencies

```
I0:   bne    $s1, $s3, I3
I1:   add    $s0, $s1, $s4
I2:   sw     $s2, 4($s3)
I3:   lw     $s3, 0($s0)
I4:   beq    $s3, $s2, I0
```

- ❑ List all control dependences

Control dependencies



```
I0:   bne    $s1, $s3, I3
I1:   add    $s0, $s1, $s4
I2:   sw     $s2, 4($s3)
I3:   lw     $s3, 0($s0)
I4:   beq    $s3, $s2, I0
```

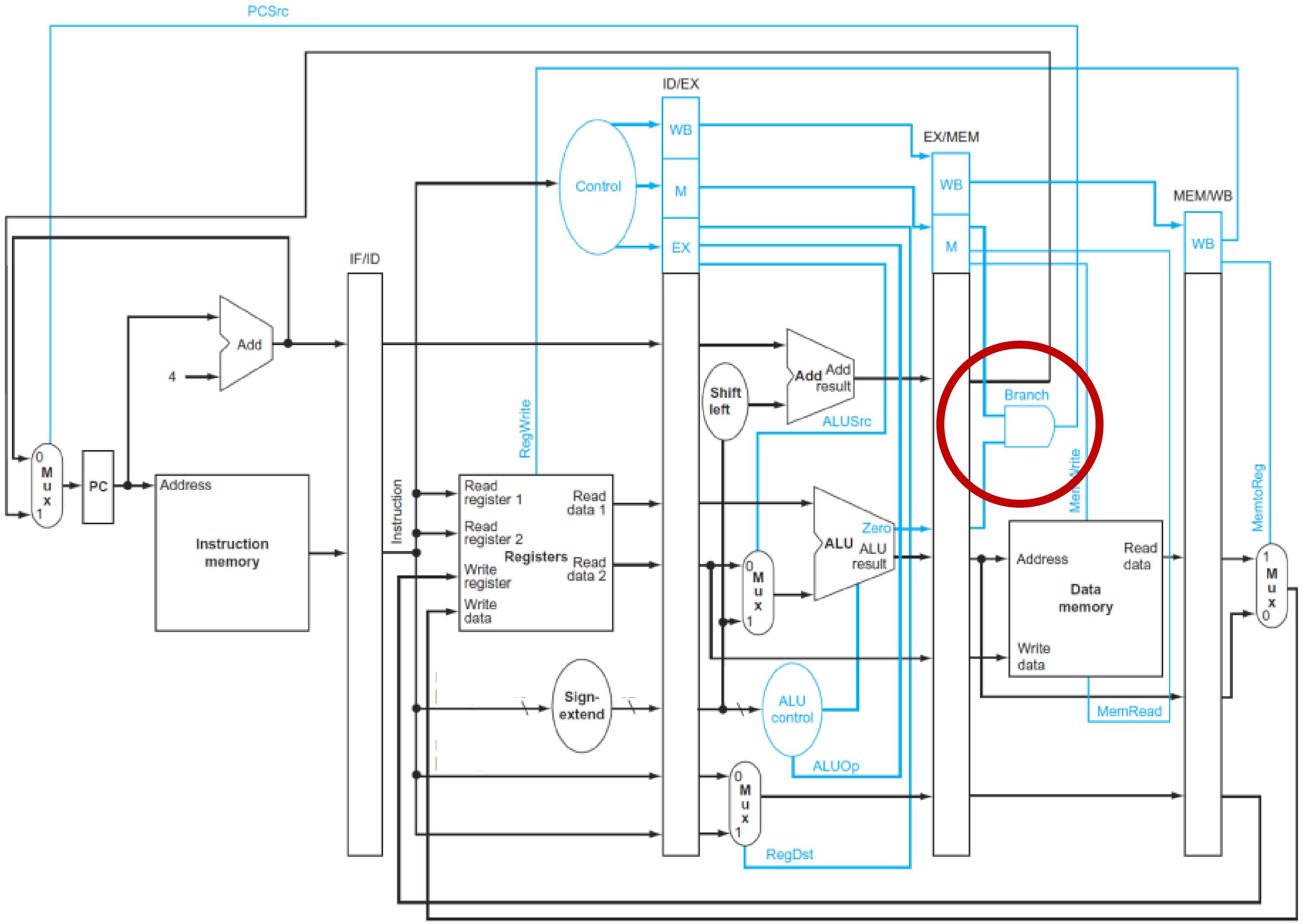
- List all control dependences:
  - `I0→I1, I0→I2`
  - `I4→I0, I4→I1, I4→I2, I4→I3, I4→I4`

Control dependencies

- ❑ Technically, control dependences need not be preserved
  - ➢Can execute instructions that should not have been executed, as long as program correctness is not affected:
    - Data dependences are preserved
    - No new exceptions incurred

Pipeline control hazards

- ❑ When an incorrect instruction is fetched because the outcome of a previous branch has not yet been resolved
  - e.g., Assume branches resolved in M stage



Pipeline control hazards

- ❑ When an incorrect instruction is fetched because the outcome of a previous branch has not yet been resolved
  - e.g., Assume branches resolved in M stage

```
beq x0, x1, TARGET        TARGET:
lw x3, 0(x0)              sub x5, x0, x5
add x4, x5, x6
or x2, x1, x7
```

inst\cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
beq	F	D	X	M	W								
lw		F	D	X	M	W							
add			F	D	X	M	W						
or				F	D	X	M	W					
sub					F	D	X	M	W				

Pipeline control hazards

❑ When an incorrect instruction is fetched because the outcome of a previous branch has not yet been resolved

- e.g., Assume branches resolved in M stage

Predict-not-taken:

- Fetch next instructions *speculatively*
- *flush* if branch is actually taken

```
beq x0, x1, TARGET
lw x3, 0(x0)
add x4, x5, x6
or x2, x1, x7
```

```
TARGET:
sub x5, x0, x5
```

inst\cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
beq	F	D	X	M	W								
lw		F	D	X	M	W							
add			F	D	X	M	W						
or				F	D	X	M	W					
sub					F	D	X	M	W				

Resolving control hazards

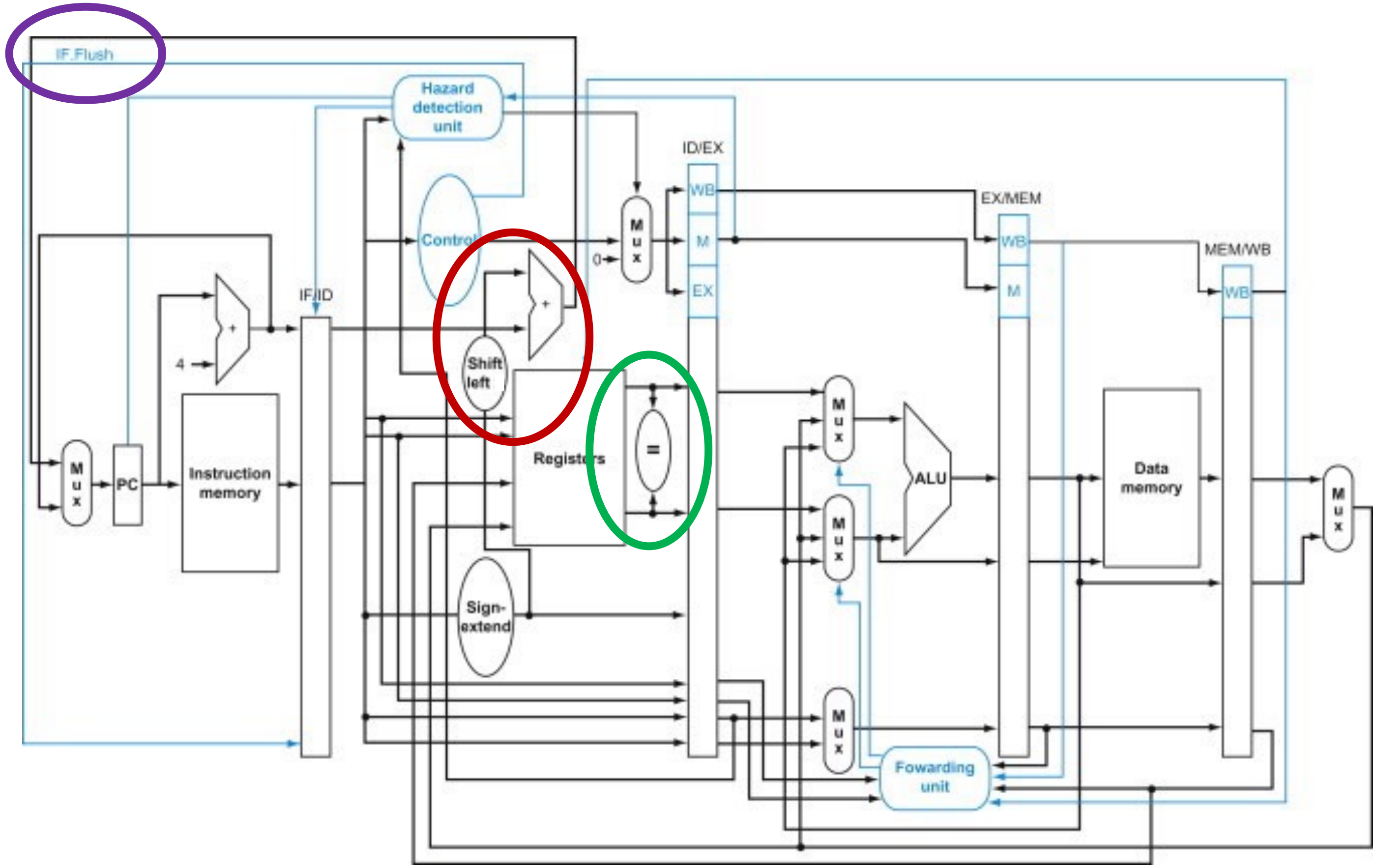
❑ Predict-not-taken + resolve in ID:

- Compute target and compare in ID stage to reduce branch penalty
- Flush: clear IF/ID.Instruction

2. Compare register operands as soon as possible

3. Clear IF/ID.Instruction

1. Compute the branch target PC in the decode stage



Pipeline control hazards

- ❑ When an incorrect instruction is fetched because the outcome of a previous branch has not yet been resolved
  - e.g., Assume branches resolved in D stage

```
beq x0, x1, TARGET          TARGET:
lw x3, 0(x0)                sub x5, x0, x5
add x4, x5, x6
or x2, x1, x7
```

inst\cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
beq	F	D											
lw		F											

* Flush 1 cycle, not 3 (if needed)

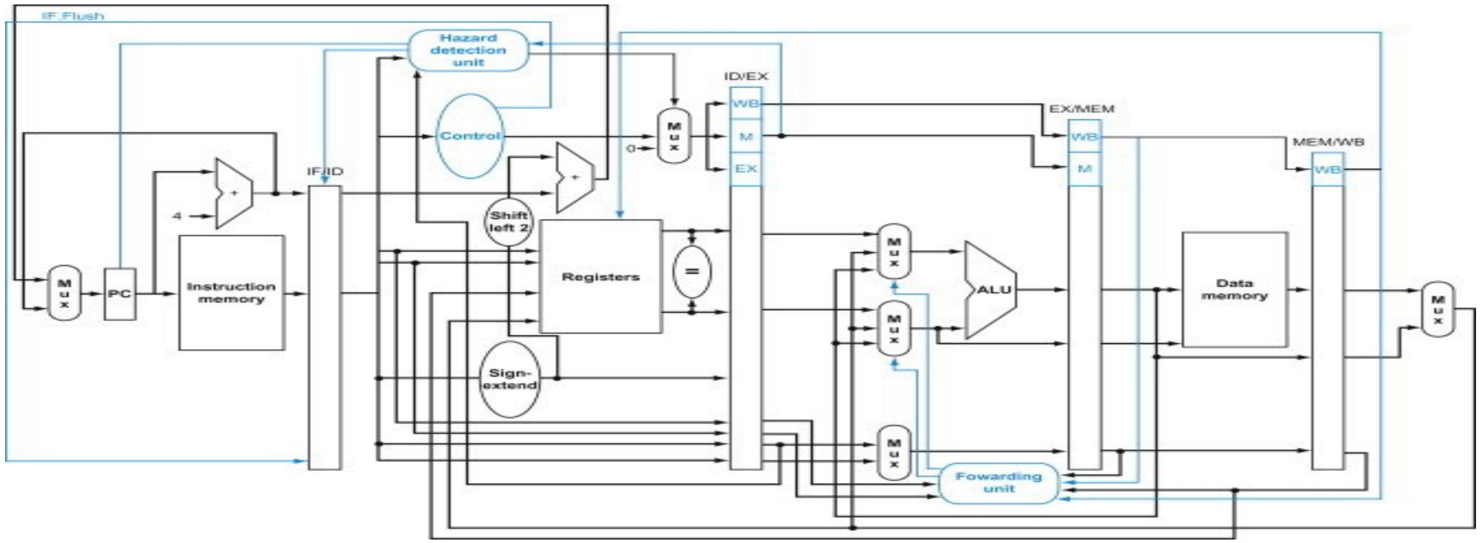
Pipeline control hazards

```
beq x0, x1, TARGET
lw x3, 0(x0)
add x4, x5, x6
or x2, x1, x7

TARGET:
sub x5, x0, x5
```

inst\cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
beq	F	D	X	M	W								
lw		F	=										
sub			F	D	X	M	W						

cycle 4:



Pipeline control + data hazards

- Predict-not-taken + resolve in ID:
  - Compute target and compare in ID stage
  - Flush: clear IF/ID.Instruction
  - **But RAW hazards in ID stage?**
    - **Stall + EX-to-ID forwarding**

```
add x1, x5, x6
beq x0, x1, TARGET
```

inst\cycle	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
add	F	D	X	M	W								
beq		F	D*	D	X	M	W						

Branch delay slots

- Branch delay slot:
  - Instruction after branch *always* executed (avoids flush logic)
    - Compiler is responsible for filling slot with *legal* instruction
  - Assume branches resolved in ID stage

```
lw x7, 0(x8)                TARGET:
beq x0, x1, TARGET          sub x5, x0, x5
add x5, x4, x6
```

inst\cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
lw	F	D	X	M	W								
beq		F	D	X	M	W							
add			F	D	X	M	W						
sub				F	D	X	M	W					

Branch delay slots

- Branch delay slot:
  - Instruction after branch *always* executed (avoids flush logic)
    - Compiler is responsible for filling slot with *legal* instruction
  - Assume branches resolved in ID stage

```
            lw x7, 0(x8)                 TARGET:
            beq x0, x1, TARGET           sub x5, x0, x5
Option 1:   nop
            add x5, x4, x6
```

inst\cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
lw	F	D	X	M	W								
beq		F	D	X	M	W							
nop													
sub				F	D	X	M	W					

Branch delay slots

- ❑ Branch delay slot:
  - Instruction after branch *always* executed (avoids flush logic)
    - Compiler is responsible for filling slot with *legal* instruction
  - Assume branches resolved in ID stage

```
             ~~lw x7, 0(x8)~~         TARGET:
             beq x0, x1, TARGET     sub x5, x0, x5
Option 2:    lw x7, 0(x8)
             add x5, x4, x6
```

inst\cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
beq	F	D	X	M	W								
lw		F	D	X	M	W							
sub			F	D	X	M	W						

Branch prediction

- Dynamic branch direction prediction:
  - *Predict* branch outcome (T or N) in IF stage
  - Assume branches resolved in ID stage

```
beq $t0, $t1, TARGET          TARGET:
add $t5, $t4, $t6             sub $t5, $t0, $t5
```

inst\cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
beq	F	D	X	M	W								
sub		F	D	X	M	W							

Predict taken (correct!)

Branch prediction

- ❑ Dynamic branch direction prediction:
  - *Predict* branch outcome (T or N) in IF stage
  - Assume branches resolved in ID stage

```
beq $t0, $t1, TARGET          TARGET:
add $t5, $t4, $t6             sub $t5, $t0, $t5
```

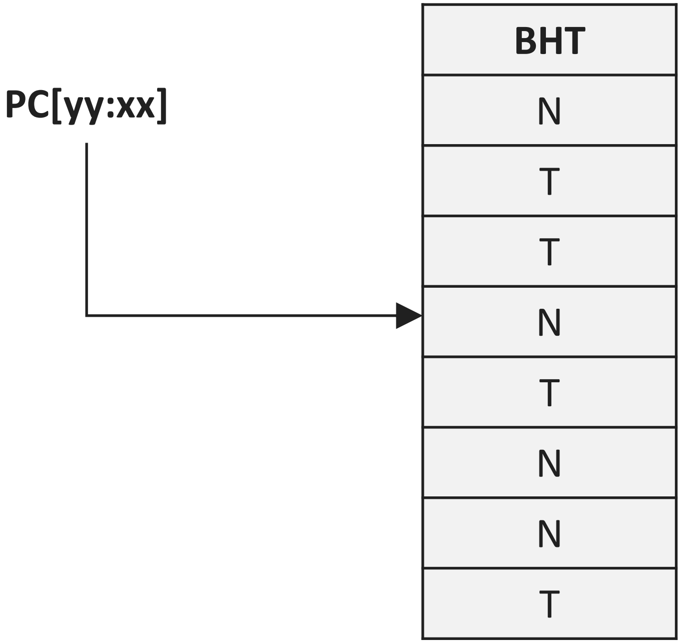
inst\cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
beq	F	D	X	M	W								
add		F	=										
sub			F	D	X	M	W						

Predict not taken (incorrect)

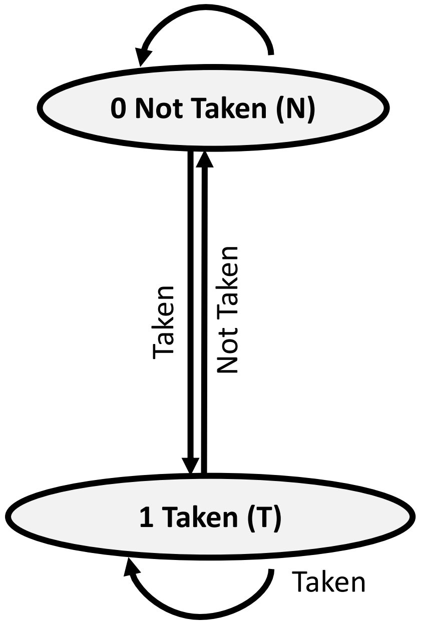
Branch history table

❑ 1-bit prediction

- *Why use PC for addressing?*
- *Why drop PC's least significant bit(s)?*



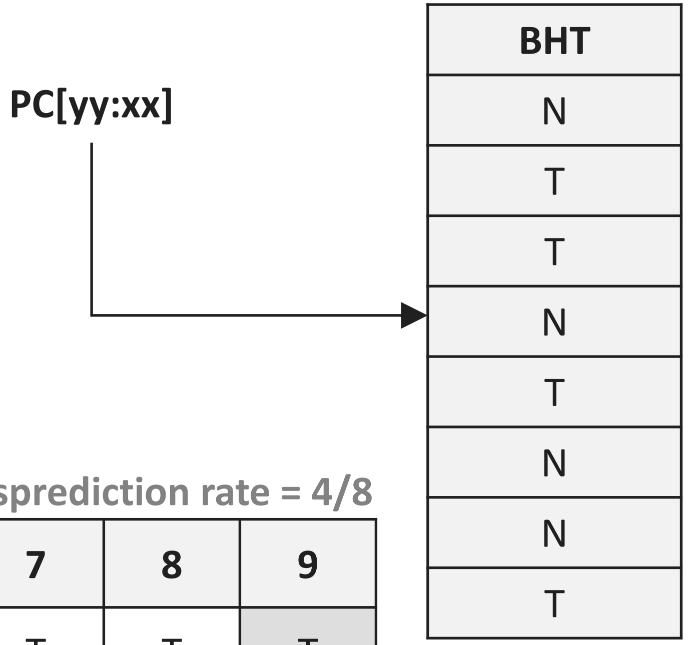
BHT
N
T
T
N
T
N
N
T



Branch history table

- ❑ 1-bit prediction

- *Why use PC for addressing?*
- *Why drop PC's least significant bit(s)?*



PC[yy:xx]

BHT

N

T

T

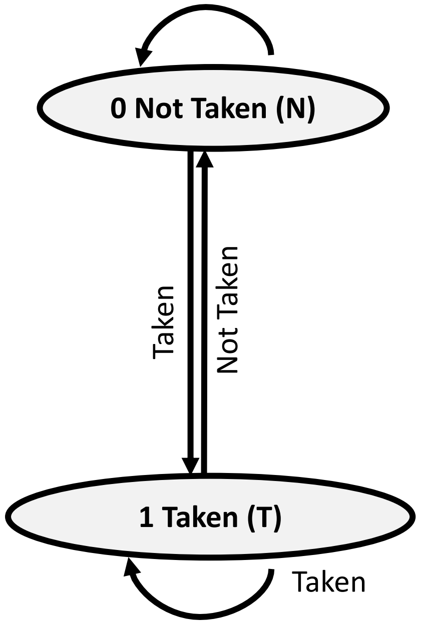
N

T

N

N

T

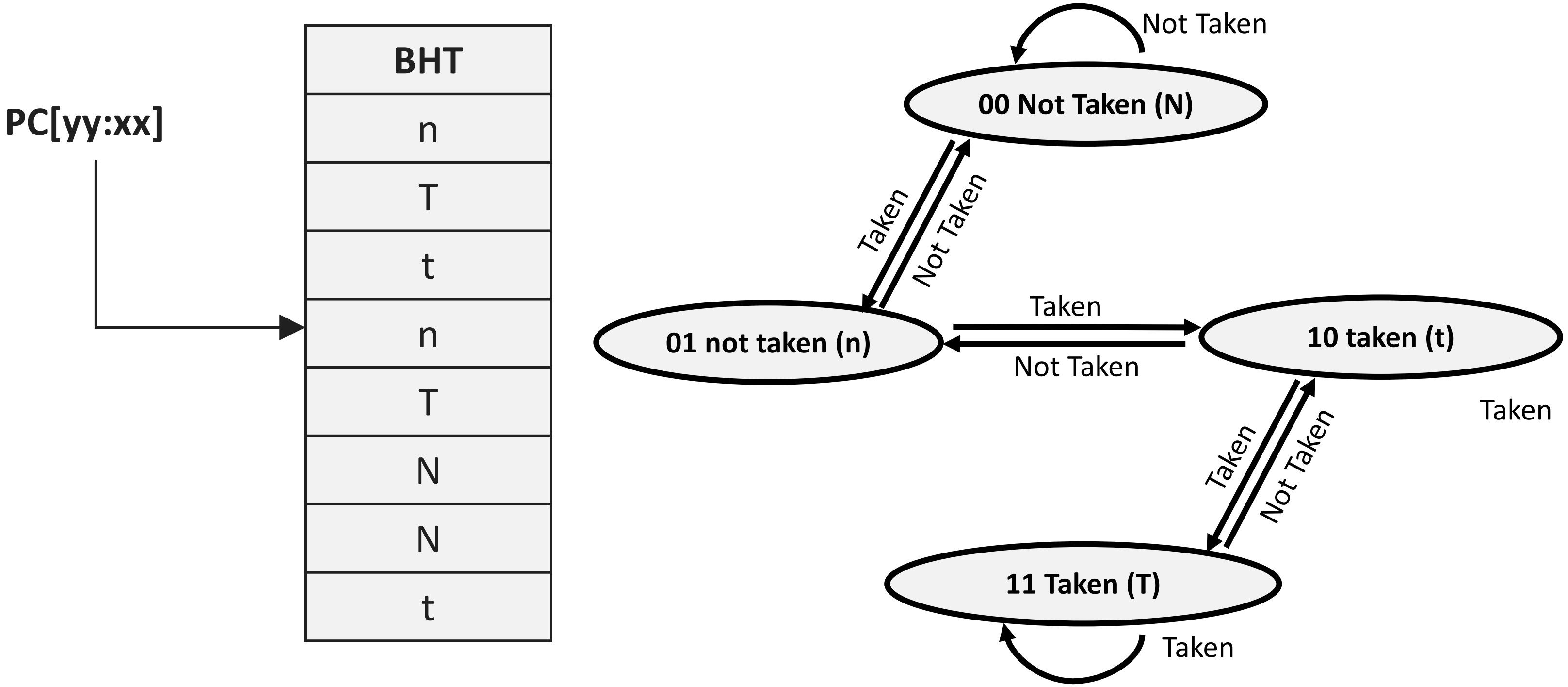


mispredicton rate = 4/8

LOOP iteration	1	2	3	4	6	7	8	9
prediction	N	T	T	T	N	T	T	T
outcome	T	T	T	N	T	T	T	N

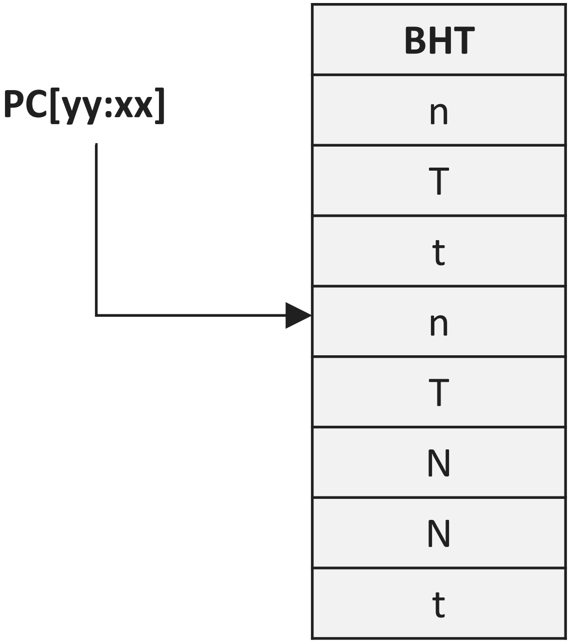
Branch history table

- ❑ 2-bit prediction



Branch history table

- ❑ 2-bit prediction



PC[yy:xx]

BHT

n

T

t

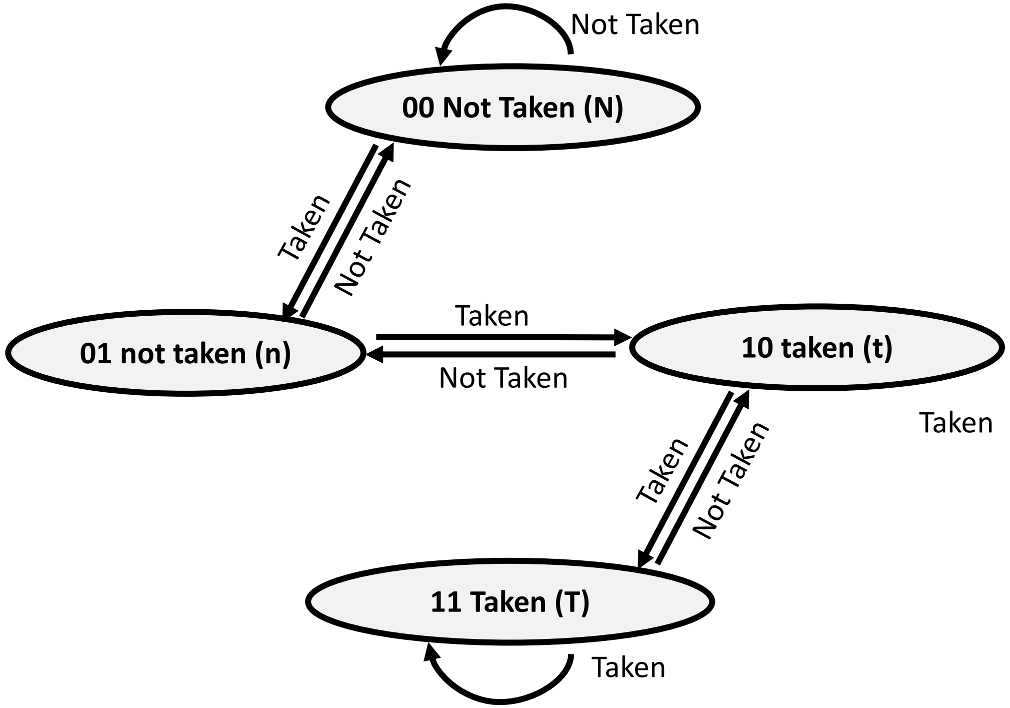
n

T

N

N

t



mispredication rate = 3/8

LOOP iteration	1	2	3	4	6	7	8	9
prediction	n	t	T	T	t	T	T	T
outcome	T	T	T	N	T	T	T	N

Exercise

❑Consider five-stage pipeline

- Assume accounting for data hazards, average CPI = 1.2
- Assume branches resolved in ID stage
- Assume 30% of instructions are branches: 60% are taken

Q. Calculate average CPI and throughput

a. Predict-not-taken with 300us clock period?

b. 90%-accurate predictor with 325us clock period?

Midterm 1 outline

Questions?