CSCI 210: Computer Architecture
Lecture 32: Control Hazards

Stephen Checkoway

Oberlin College

Dec. 22, 2021

Slides from Cynthia Taylor

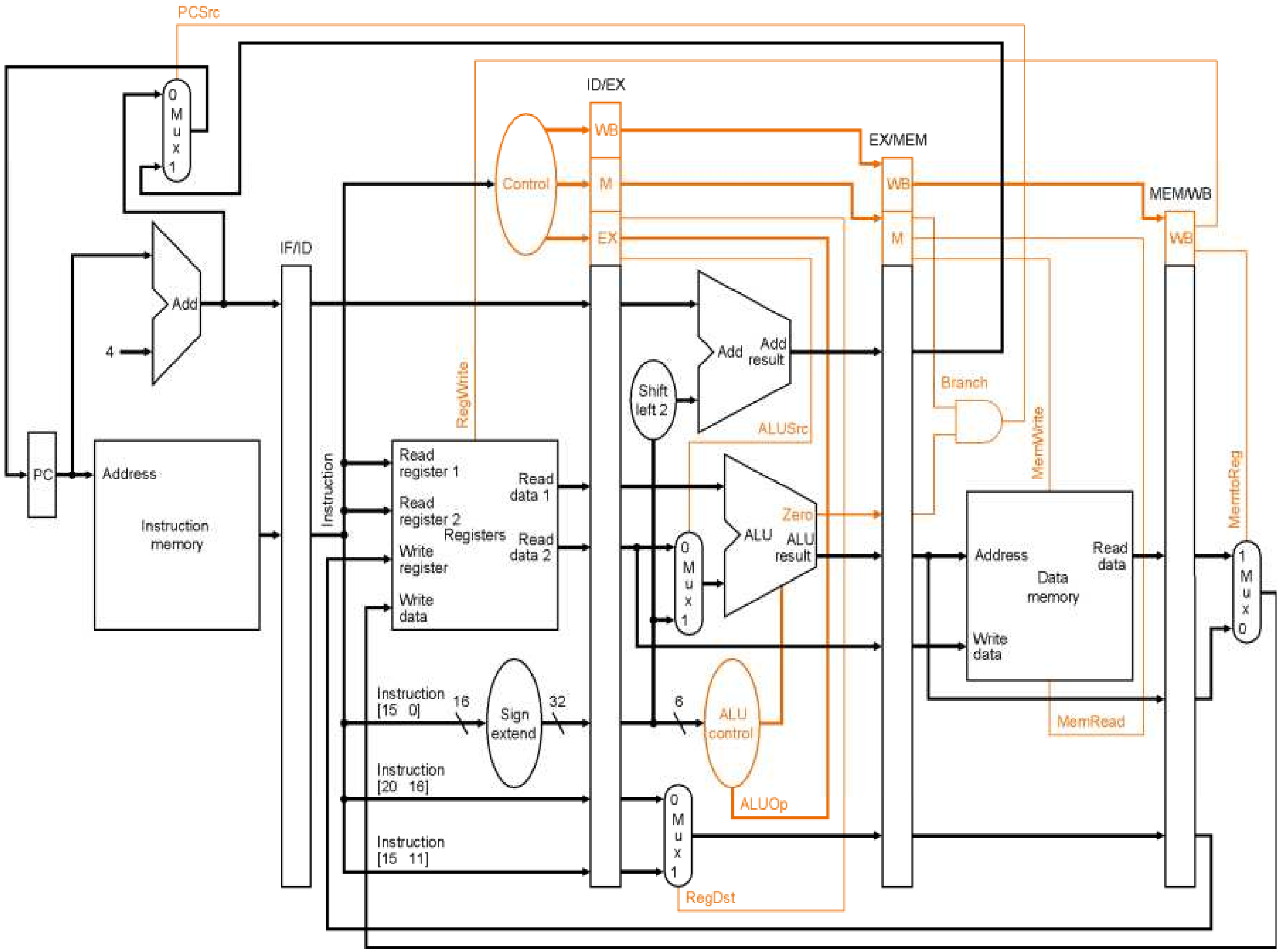
Announcements

- Problem Set 10 due Sunday, January 2
- Lab 8 due Sunday, January 2
- Office Hours tomorrow 13:30–14:30

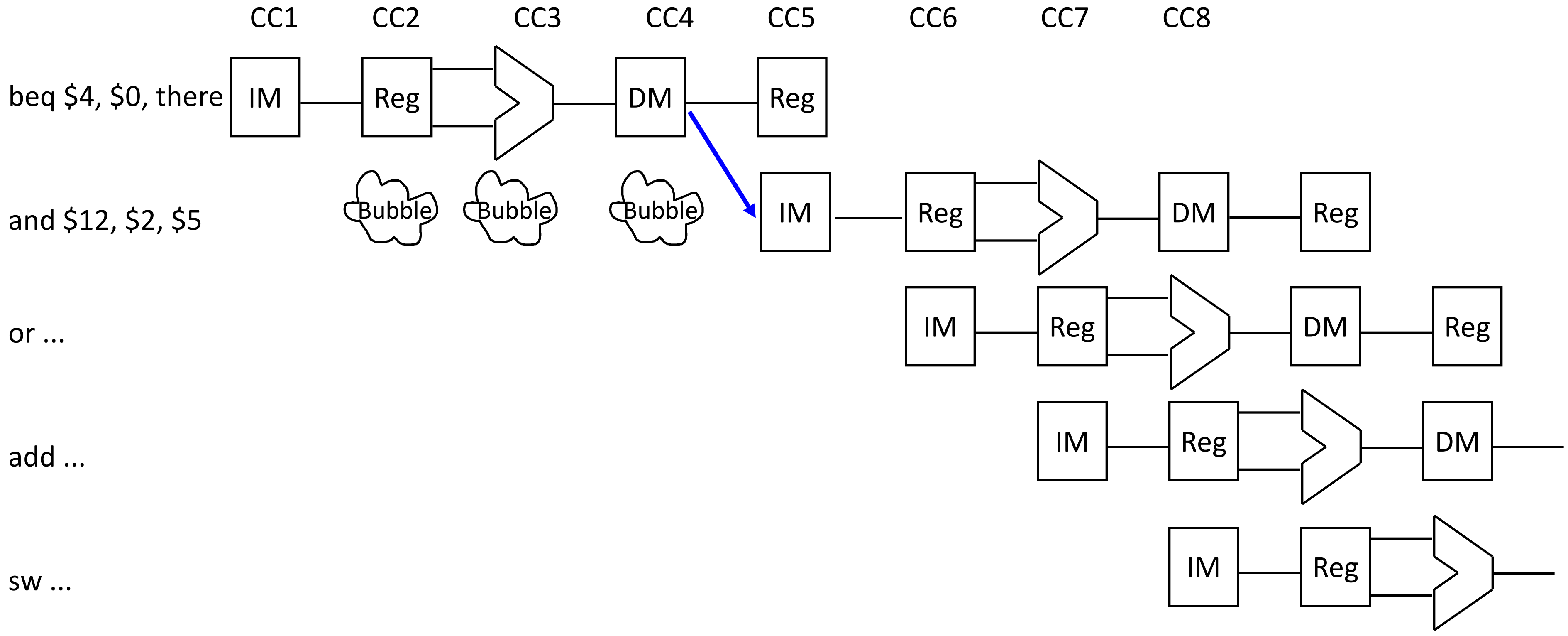
Stalling the pipeline

Given a pipeline where branches are resolved by the ALU – let's assume we stall until we know the branch outcome. How many cycles will you lose per branch?

Selection	cycles
A	0
B	1
C	2
D	3
E	4



Stalling for Branch Hazards

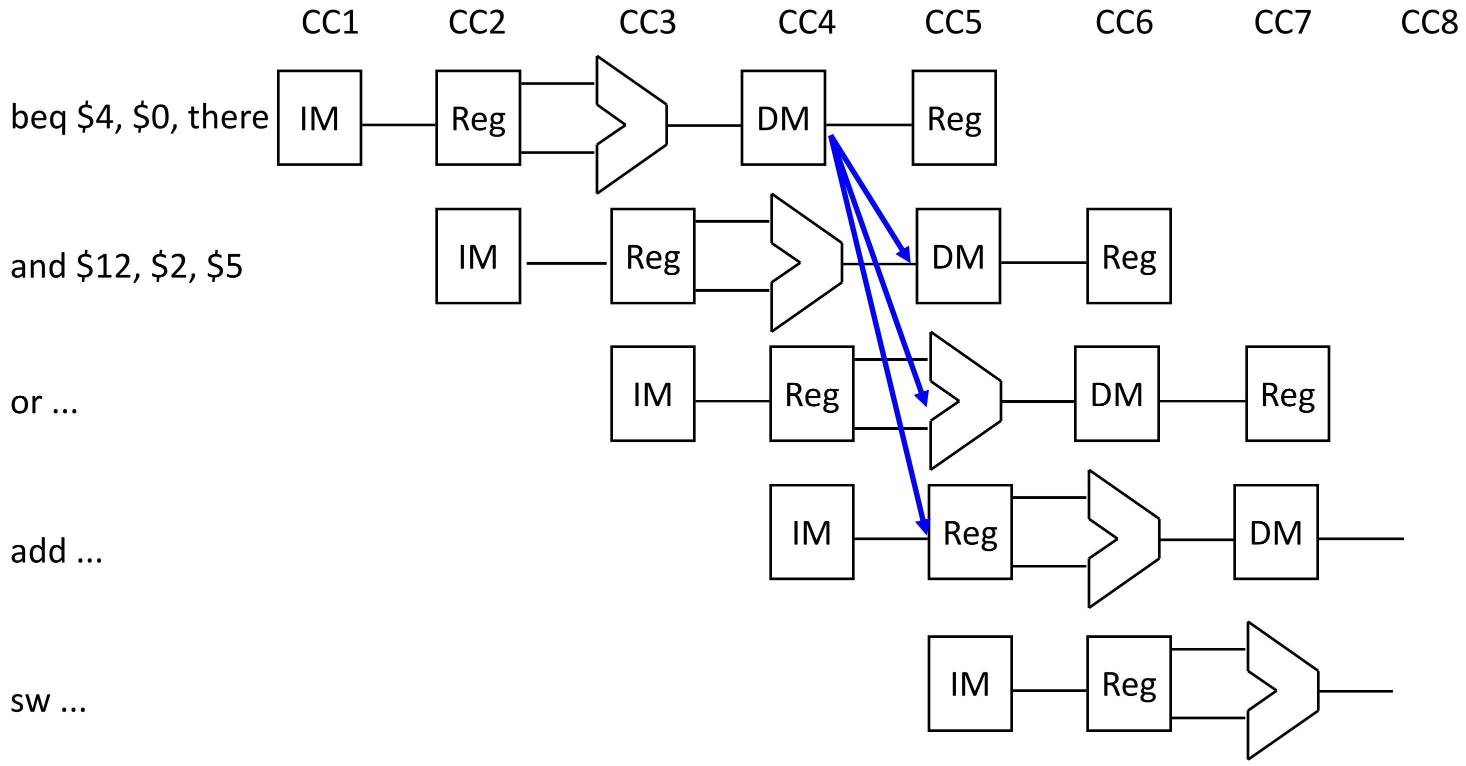


Stalling for Branch Hazards

- Seems wasteful, particularly when the branch isn’t taken.
- Makes all branches cost 4 cycles.
- What if we just assume the branch isn’t taken?

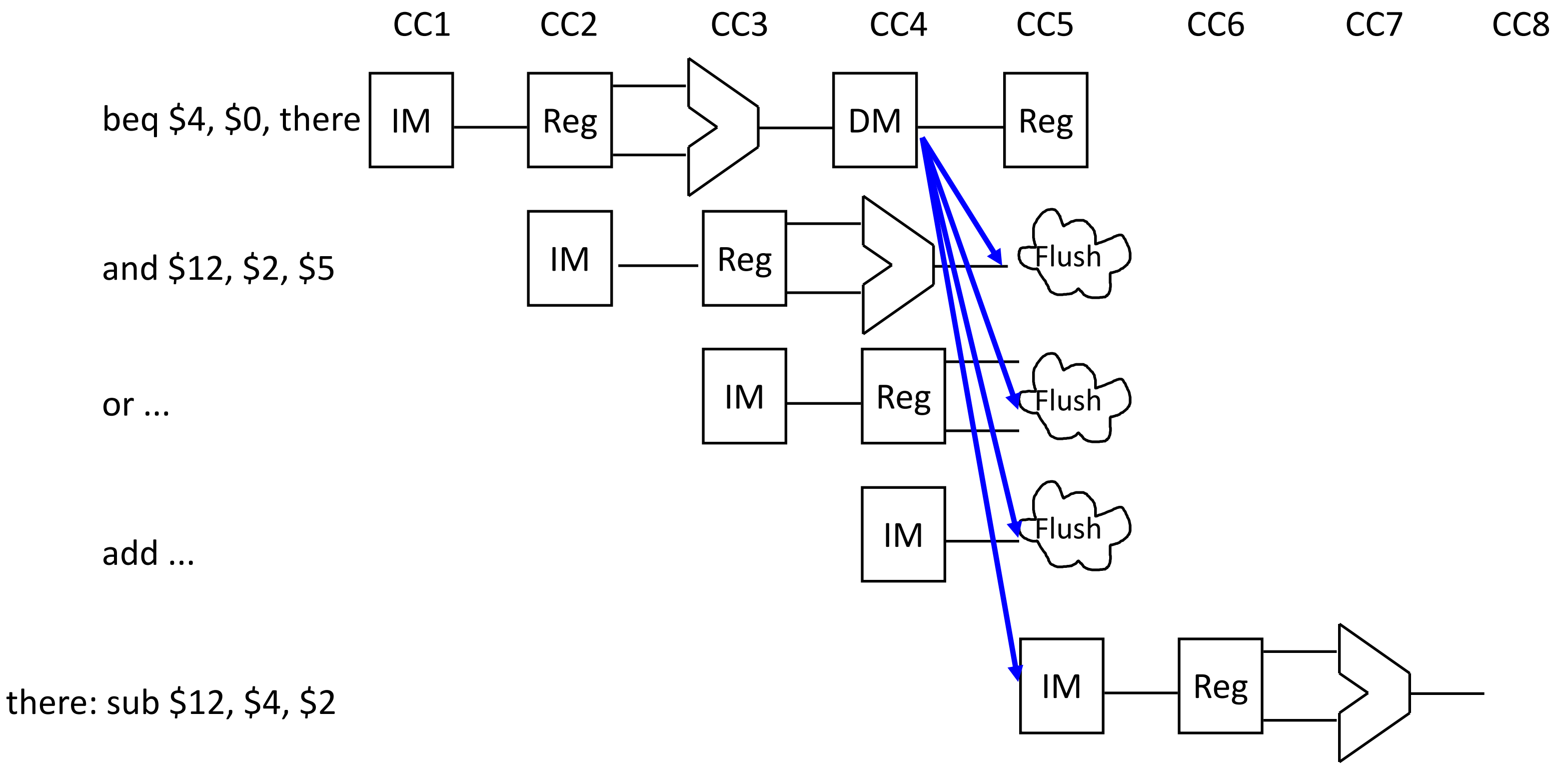
Assume Branch Not Taken

works pretty well when you're right



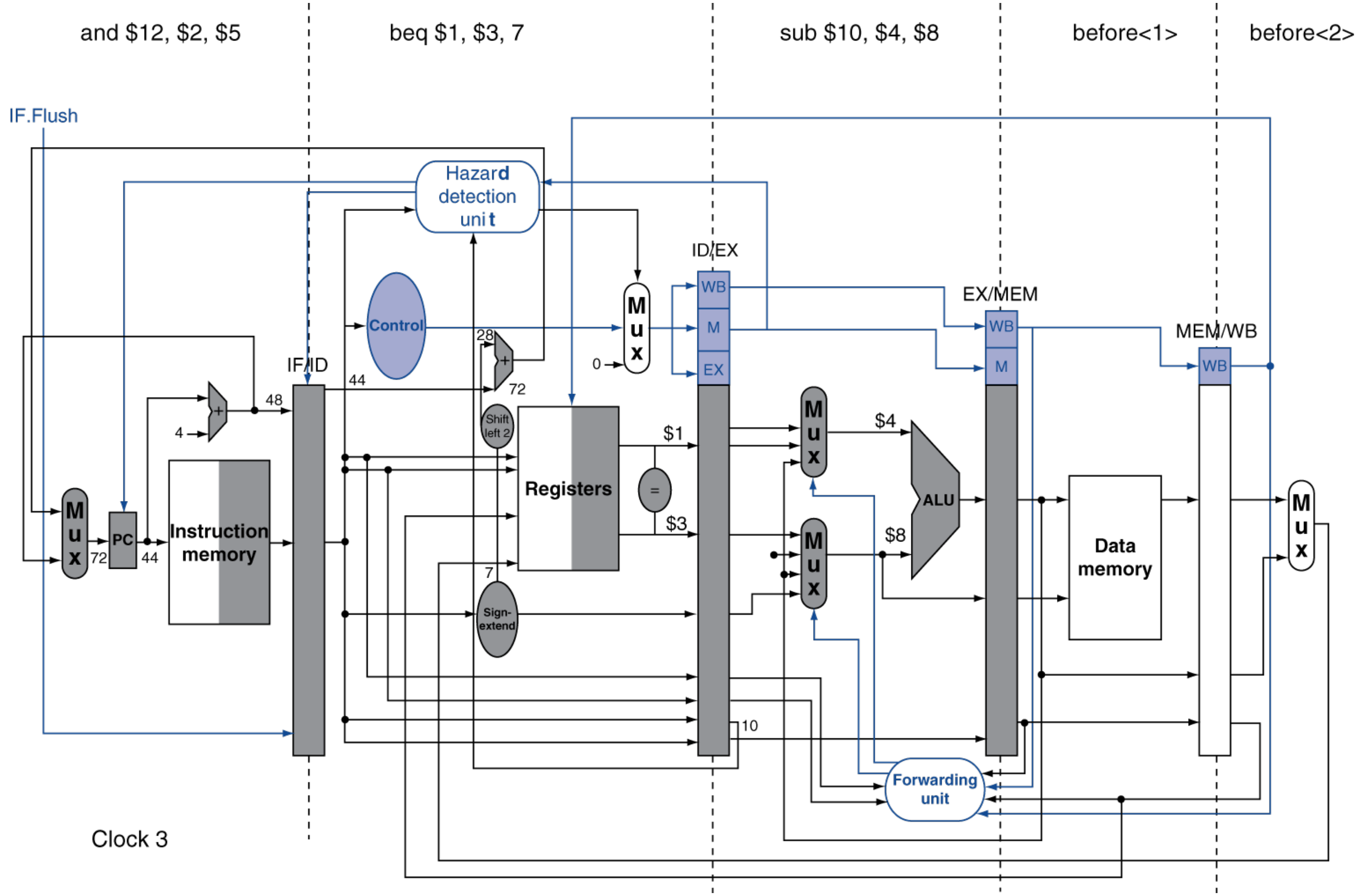
Assume Branch Not Taken

same performance as stalling when you're wrong



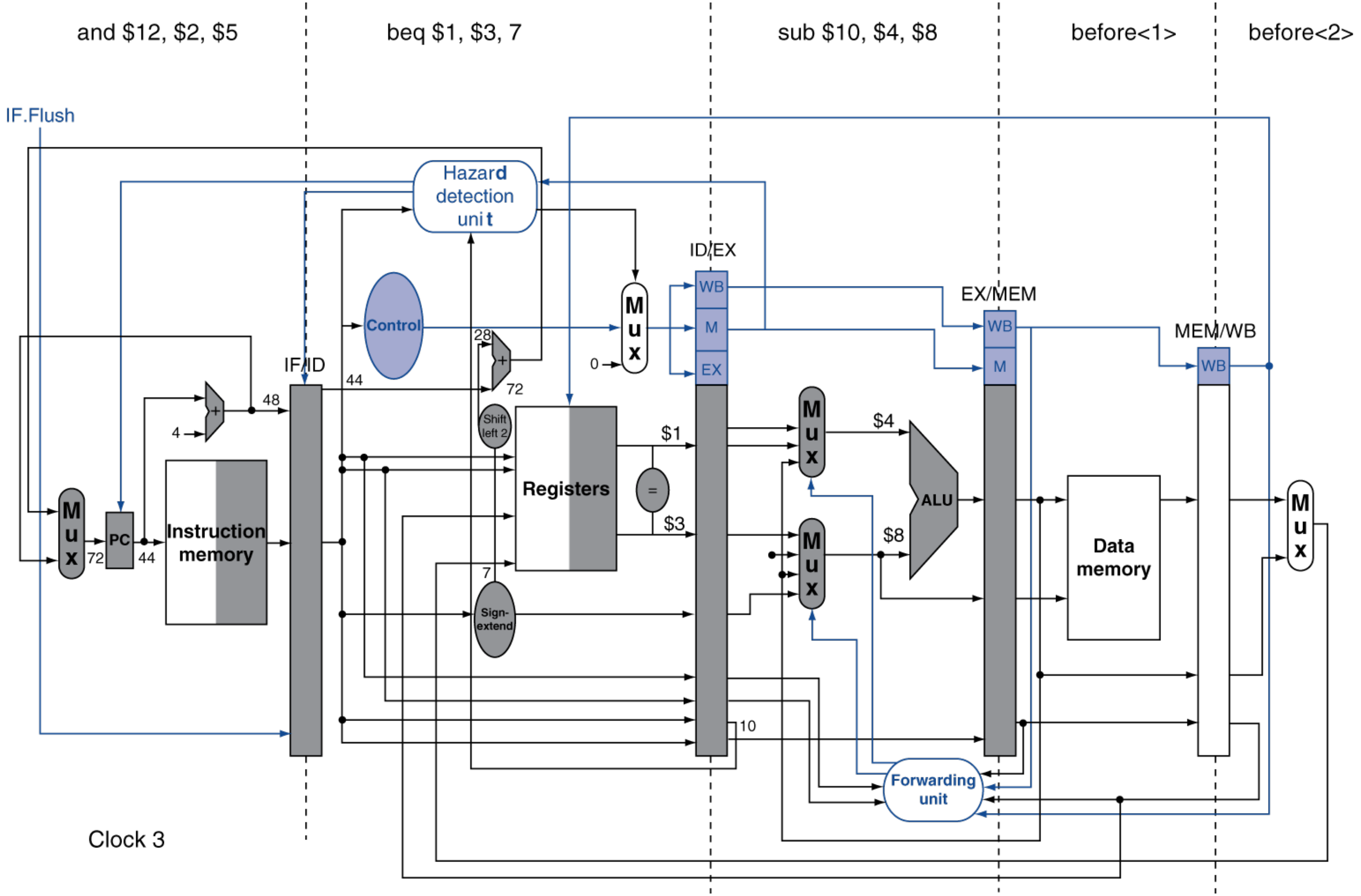
Stalling the pipeline

Let's improve the pipeline so we move branch resolution to Decode + assume branches are not taken. How many cycles would we lose then on a taken branch?

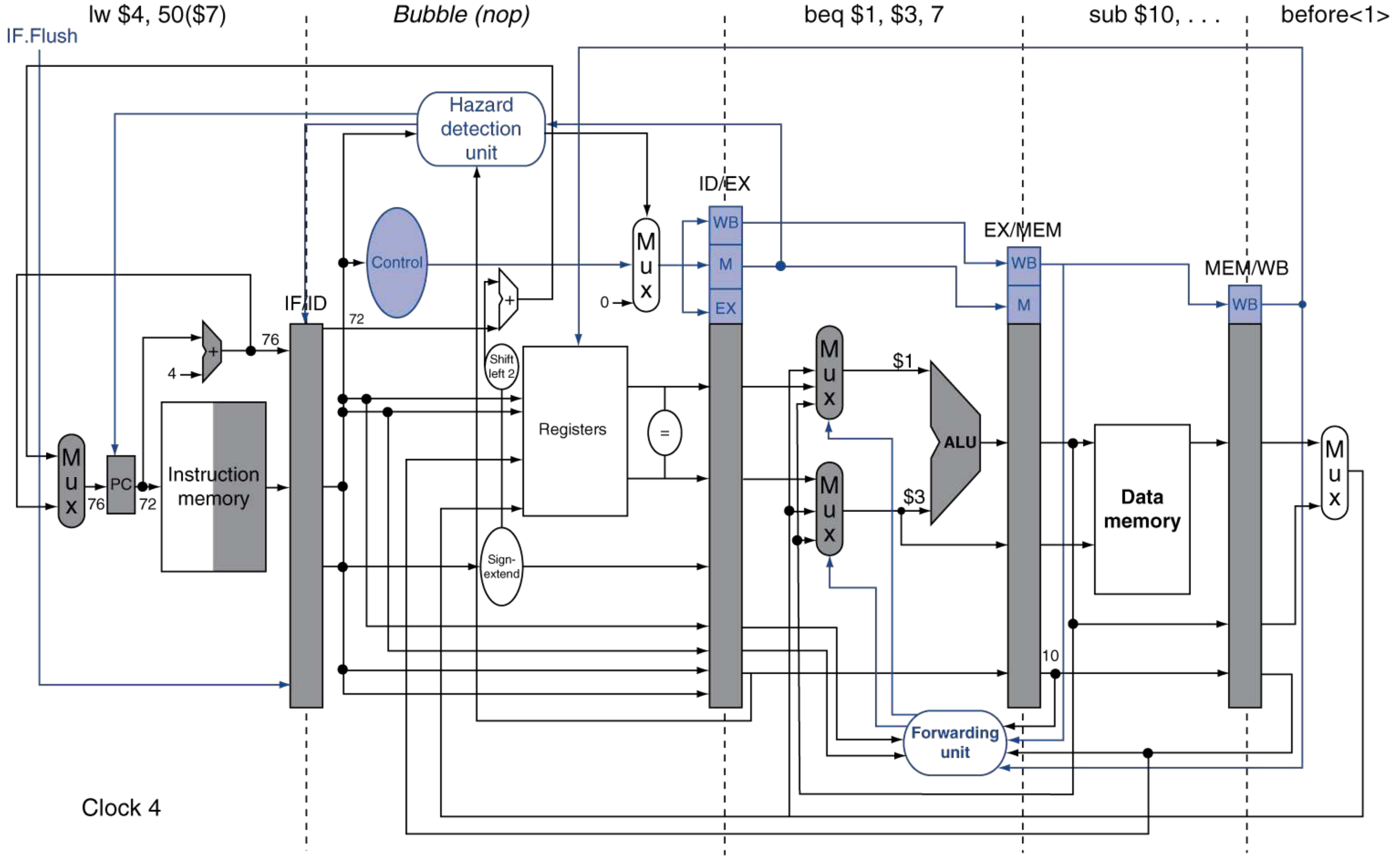


Selection	cycles
A	0
B	1
C	2
D	3
E	4

Example: Branch Taken



Example: Branch Taken

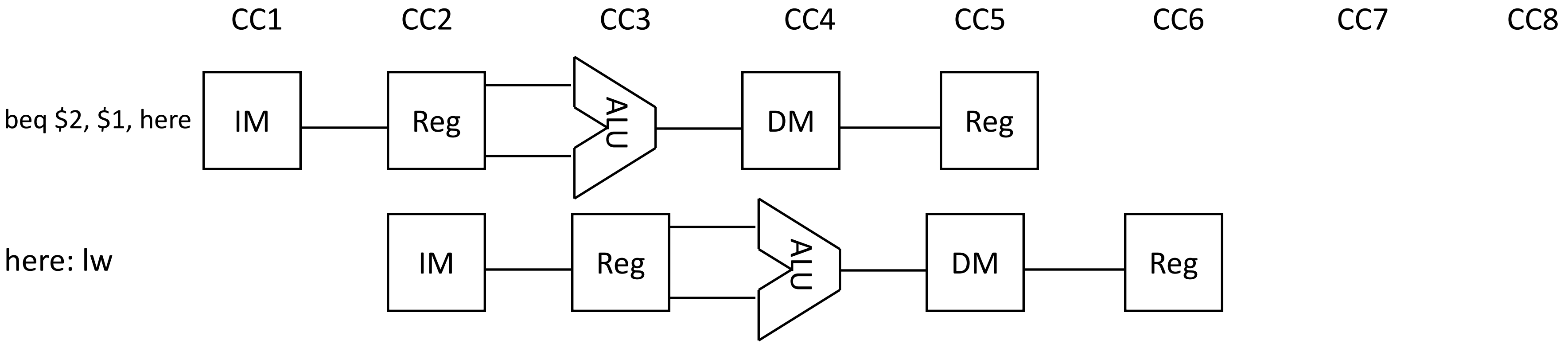


Branch Hazards – Assume Not Taken

- Great if most of your branches aren’t taken.

- What about loops which are taken 95% of the time?
  - We would like the option of assuming not taken for some branches, and taken for others, depending on what they usually do

Branch Hazards – Predicting Taken



Required information to predict branch outcomes:

1. An instruction is a branch before decode
2. The target of the branch (where it branches to)
3. Values in the registers the branch will compare

Selection	Required knowledge
A	2, 3
B	1, 2, 3
C	1, 2
D	2
E	None of the above

Branch Target Buffer

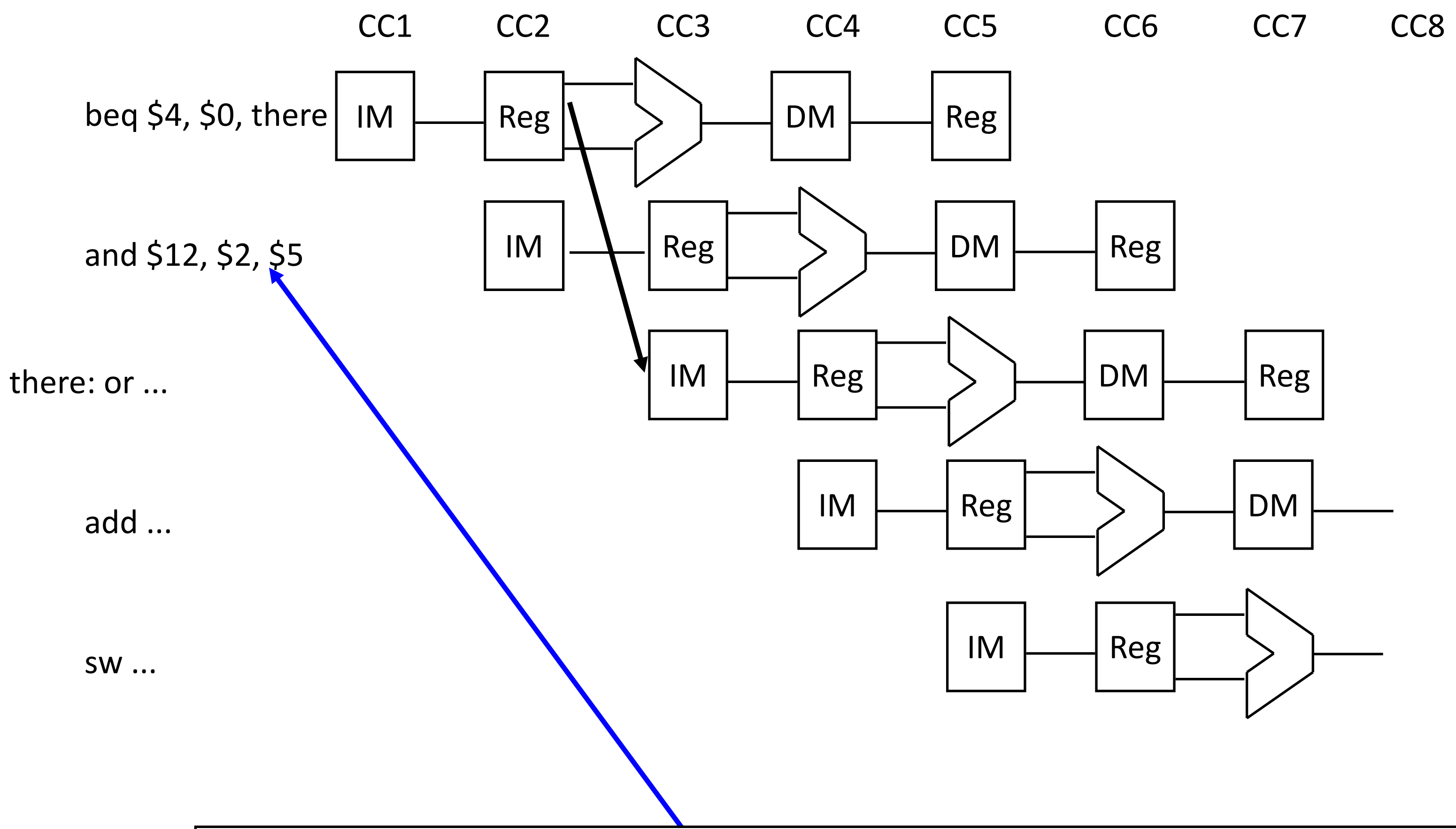
- Keeps track of the PCs of recently seen branches and their targets.
- Consult during Fetch (in parallel with Instruction Memory read) to determine:
  - Is this a branch?
  - If so, what is the target

PC	Target
0x40024	0x4018C
0x40188	0x40028
⋮	⋮

Branch Hazards – Predict Taken

- Static policy:
  - Forward branches (if statements) predict not taken
  - Backward branches (loops) predict taken
- Dynamic prediction
- Branch Delay Slots

Branch Delay Slot



Branch delay slot instruction (next instruction after a branch) is executed even if the branch is taken.

Which instructions could we put in the branch delay slot?

```
1 add  $5, $3, $7
2 add  $9, $1, $3
3 sub  $6, $1, $4
4 and  $7, $8, $2
5 beq $6, $7, there
  nop   /* branch delay slot */
6 add  $9, $1, $2
7 sub  $2, $9, $5
  …
  there:
8 mult $2, $10, $9
  …
```

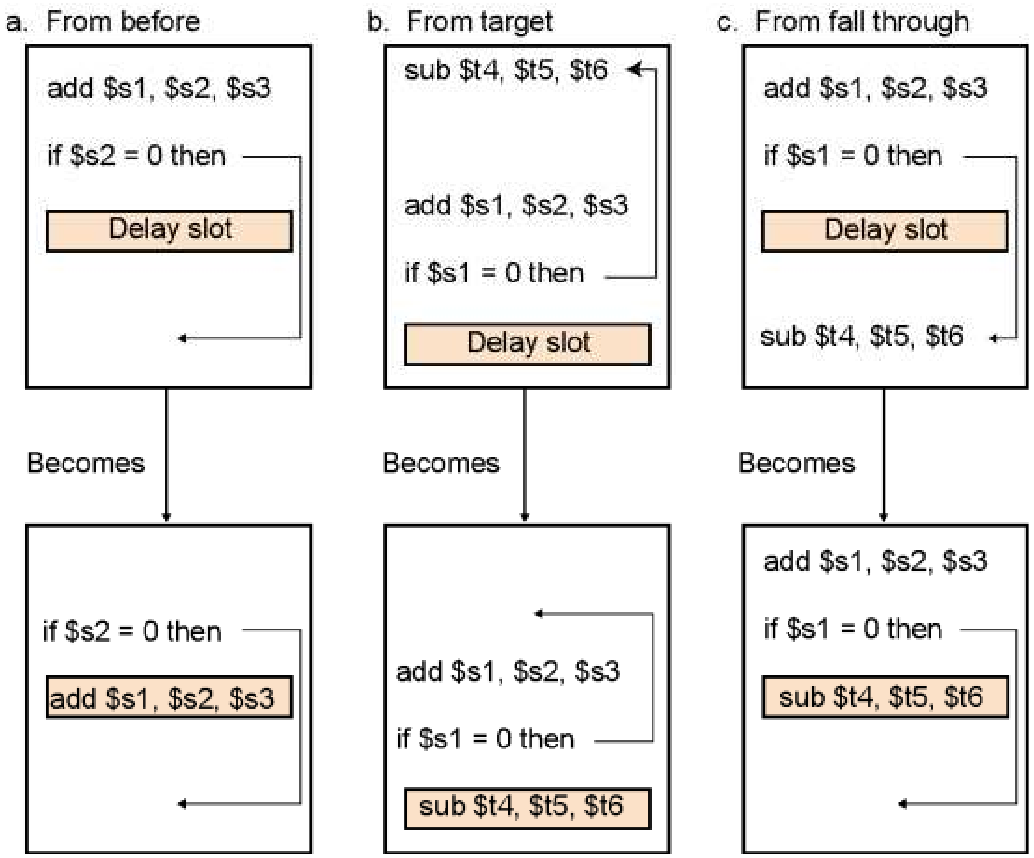
Selection	Safe instructions
A	2
B	1,2
C	2,6
D	1,2,7,8
E	None of the above

Filling the branch delay slot

```
1  add  $5, $3, $7        No-$7 overwritten
2  add  $9, $1, $3        Safe, $1 and $3 are fine
3  sub  $6, $1, $4        No-$6
4  and  $7, $8, $2        No-$7
5  beq $6, $7, there
    nop   # branch delay slot
6  add  $9, $1, $2        Not safe ($9 on taken path)
7  sub  $2, $9, $5        Not safe (needs $9 not yet produced)
    ...
    there:
8  mult $2, $10, $9       Not safe ($2 is used before overwritten)
    …
```

Filling the branch delay slot

- The branch delay slot is only useful if you can find something to put there.
- If you can't find anything, you must put a nop to insure correctness.



Which MIPS instruction is the best nop?

A. `addi  $t0, $t0, 0`
B. `sll   $zero, $zero, 0`
C. `or    $v0, $v0, $zero`
D. `and   $s0, $s0, $zero`
E. `add   $zero, $t0, $t0`

Unnecessary load-use penalty

```
lw    $t0, 0($s0)
add   $zero, $t0, $t0
```

This doesn't matter for filling the branch delay slot because even `lw ; beq; add` can't cause a load-use stall

MIPS uses the all-zero instruction as `nop`:

```
sll   $zero, $zero, 0
```

Branch Delay Slots

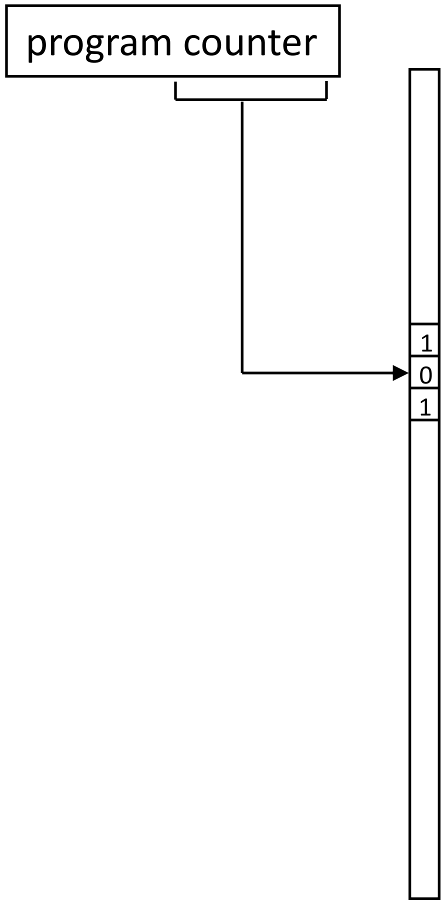
- This works great for this implementation of the architecture.

- What about the MIPS R10000, which has a *5*-cycle *branch penalty*, and executes 4 instructions per cycle???

Dynamic Branch Prediction

- Can we guess the outcome of branches?
- What should we base that guess on?

1-bit Branch Predictor




```
for (i=0;i<10;i++) {
...
...
}
```

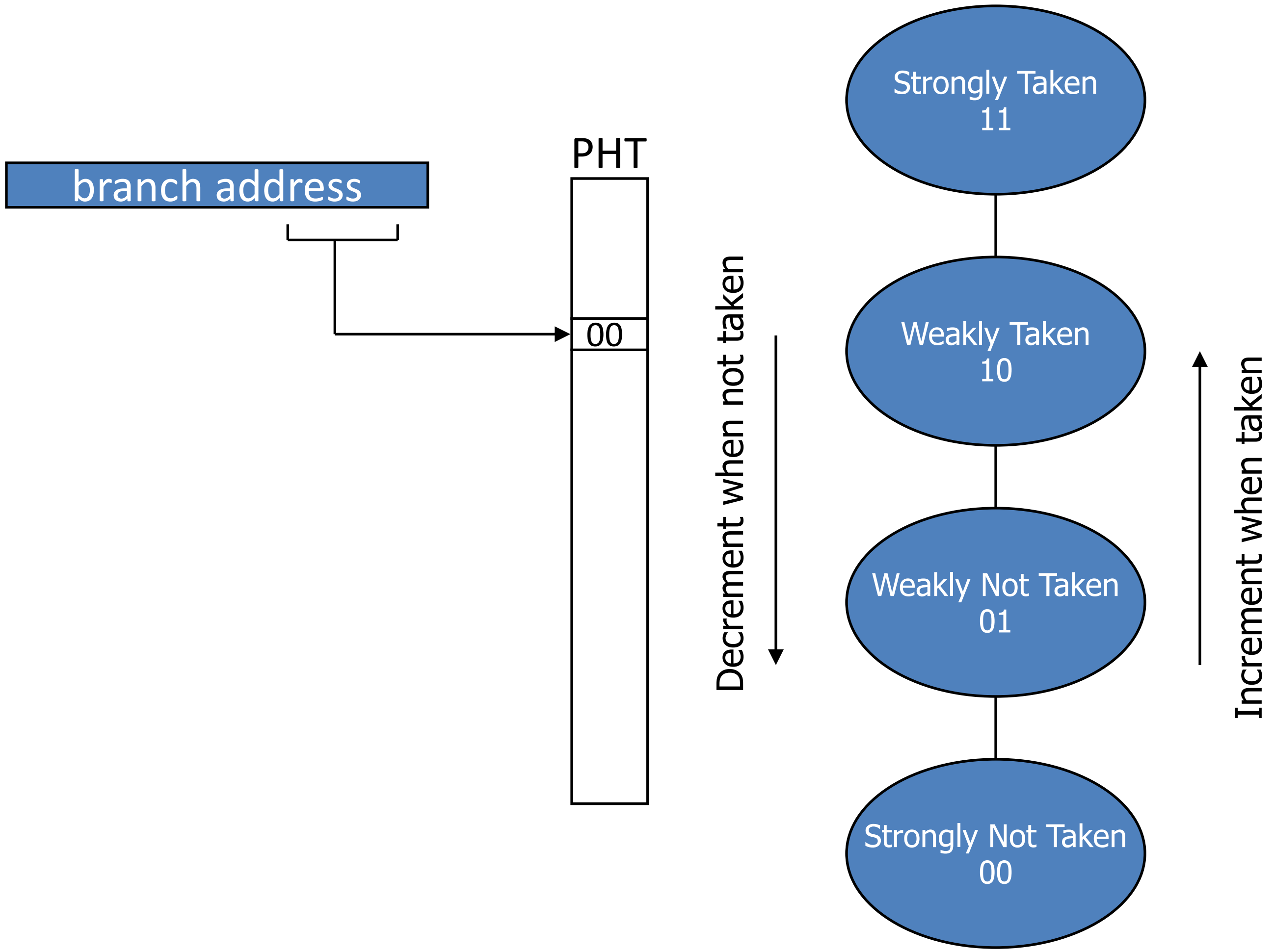
```
...
...
add  $i, $i, #1
beq $i, #10, loop
```

Every time branch is taken, set bit to 1, untaken, 0.

Assume we start with our 1-bit predictor at 1, for Taken, and change it to 0 whenever the branch is not taken. How accurate will it be for the branch pattern TTNTTNTT

A. 3/8
B. 4/8
C. 5/8
D. 8/8
E. None of the above

Two-bit predictors give better loop prediction




```
for (i=0;i<10;i++) {
…
…
}
```

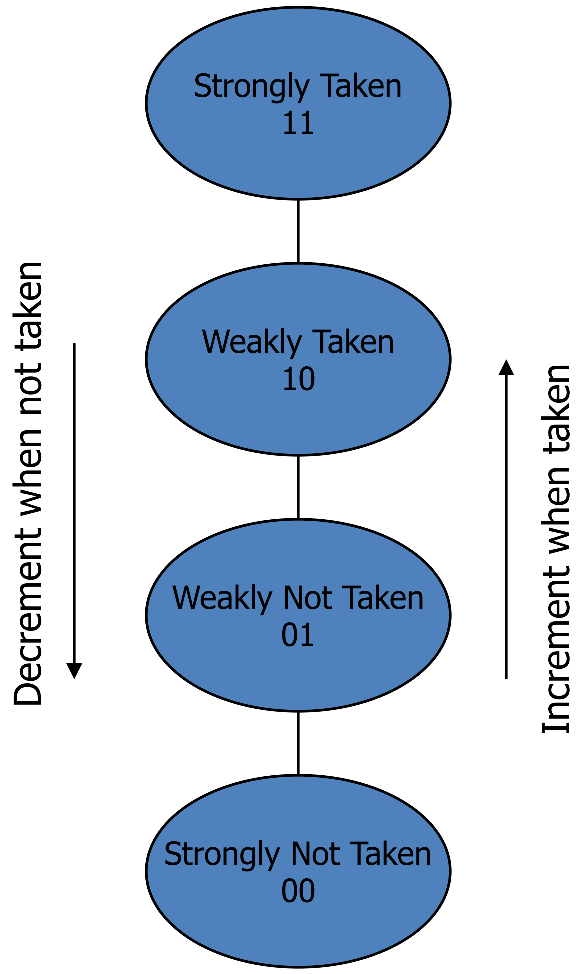
```
…
…
add  $i, $i, #1
beq $i, #10, loop
```

Suppose we have the following branch pattern. What is the accuracy of a 1-bit and 2-bit branch predictors. Assume initial values of 1 (1-bit) and (10) 2-bit.

T T N T N

	1 bit	2 bit
A	2/5	2/5
B	3/5	2/5
C	2/5	3/5
D	1/5	4/5

E. None of the above



Branch Prediction

- Latest branch predictors are significantly more sophisticated, using more advanced correlating techniques, larger structures, and even AI techniques

- Use patterns of branches (local history) and recent other branch history (global history) to make predictions

Putting it all together.

For a given program on our 5-stage MIPS pipeline processor:

- 20% of instructions are loads, 50% of instructions following a load are arithmetic instructions depending on the load. Recall load hazards are a 1 cycle stall.
- 20% of instructions are branches. Using dynamic branch prediction, we achieve 80% prediction accuracy. Mispredicted branches are a 1 cycle stall.

What is the CPI of your program?

Assume a base CPI of 1.

Selection	CPI
A	0.76
B	0.9
C	1.0
D	1.14
E	None of the above

Control Hazards — Key Points

- Control (or branch) hazards arise because we must fetch the next instruction before we know if we are branching or where we are branching.
- Control hazards are detected in hardware.
- We can reduce the impact of control hazards through:
  - early detection of branch address and condition
  - branch prediction
  - branch delay slots

Pipelining — Key Points

- Pipelining focuses on improving instruction throughput, not individual instruction latency.
- Data hazards can be handled by hardware or software – but most modern processors have hardware support for stalling and forwarding.
- Control hazards can be handled by hardware or software – but most modern processors use Branch Target Buffers and advanced dynamic branch prediction to reduce the hazard.
- ET = IC*CPI*CT

Reading

- Reading for today’s lecture since we’re ahead: 5.9
- Next lecture: Caches
  - Section 6.2