CSCI 210: Computer Architecture
Lecture 31: Control Hazards

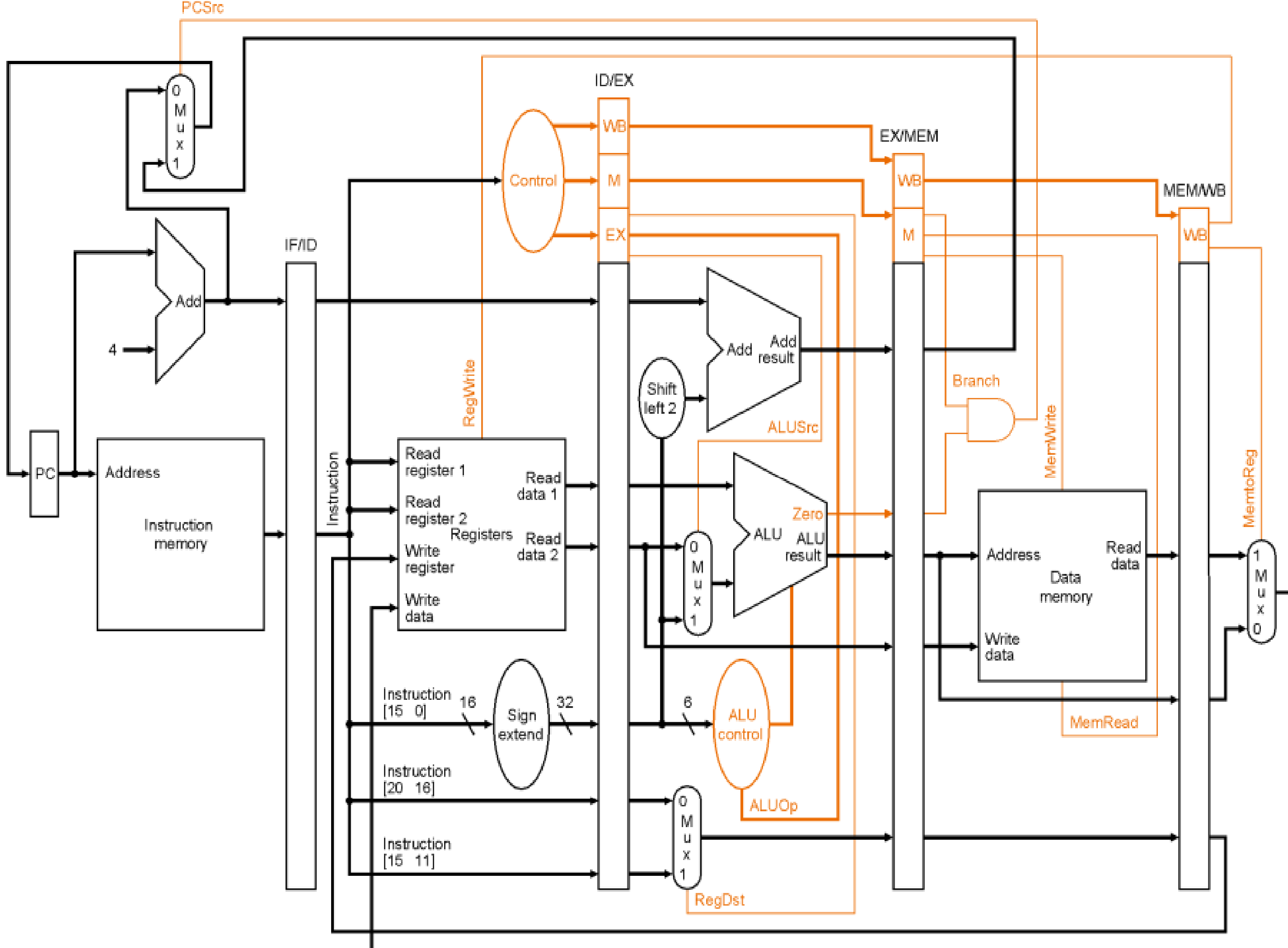
Stephen Checkoway
Slides from Cynthia Taylor

CS History: Branch Prediction

- The IBM Stretch implemented “predict taken” branch prediction in the 1950s
- Two-bit branch prediction was developed at Livermore Labs in 1977, and independently at the CDC in 1979
- MIPS R2000 was introduced in January 1986, and did ”not-taken” branch prediction
  - This was not a big performance hit because of the use of the branch-delay slot and the short pipeline
- In the 1990s, the introduction of super-scalar computers made branch prediction more important, and the Intel Pentium, DEC Alpha, MIPS R8000, and the IBM POWER all had 1 and 2-bit branch predictors

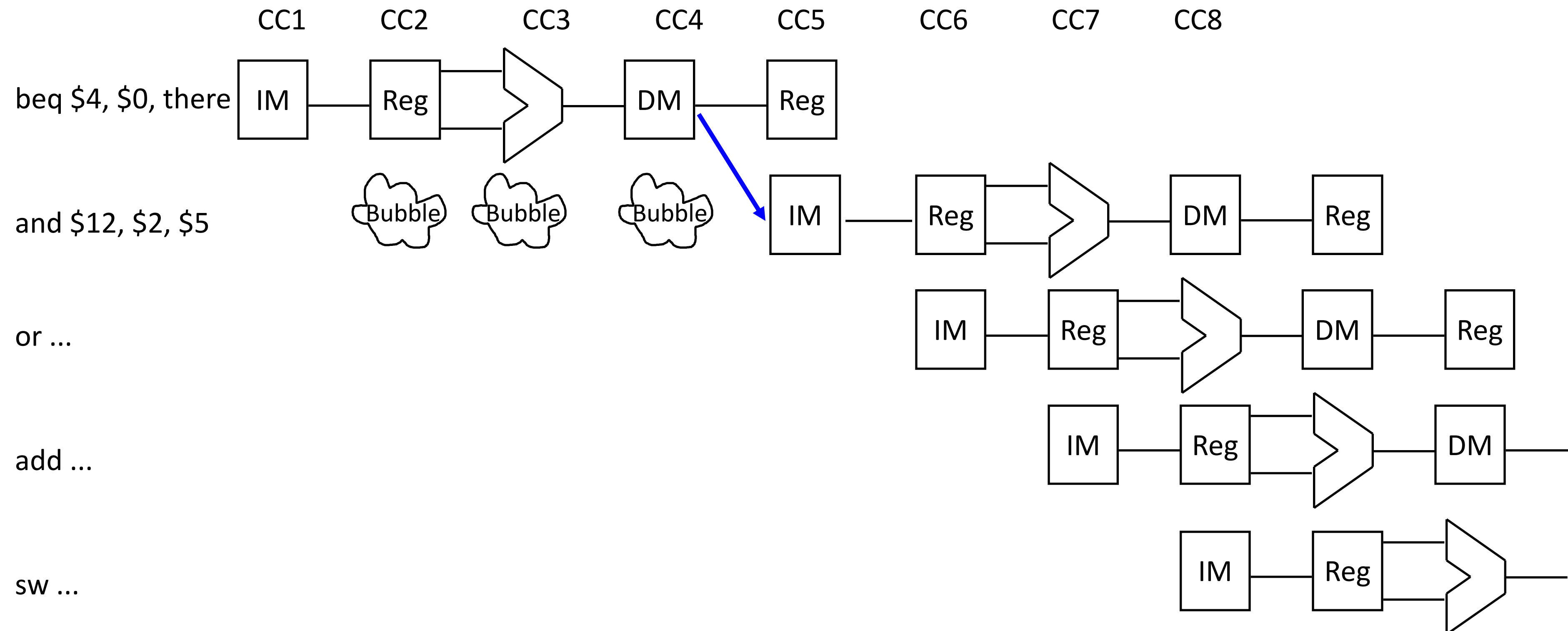
Stalling the pipeline

Given this pipeline where branches are resolved by the ALU and PC is updated in the MEM stage – 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

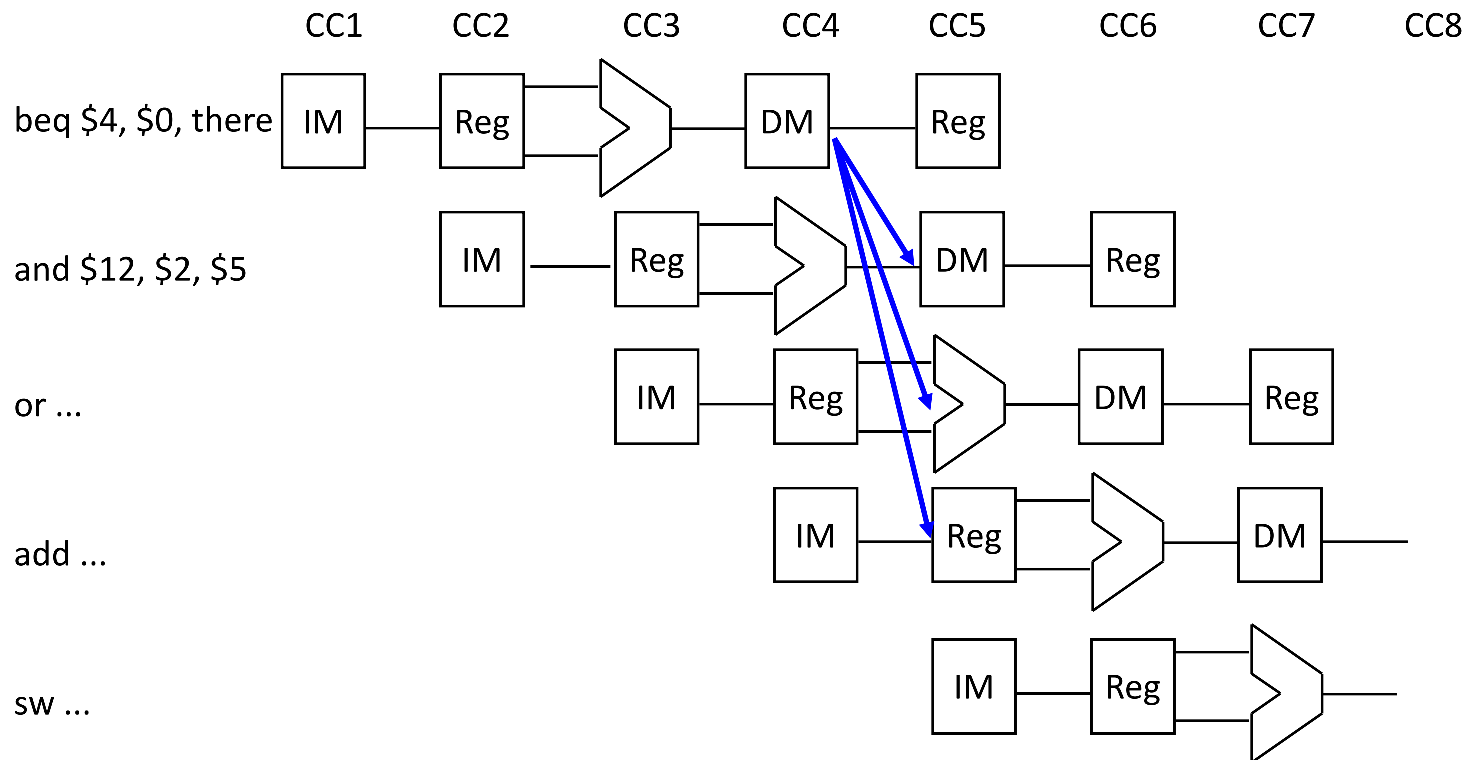


Stalling for Branch Hazards

- 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?

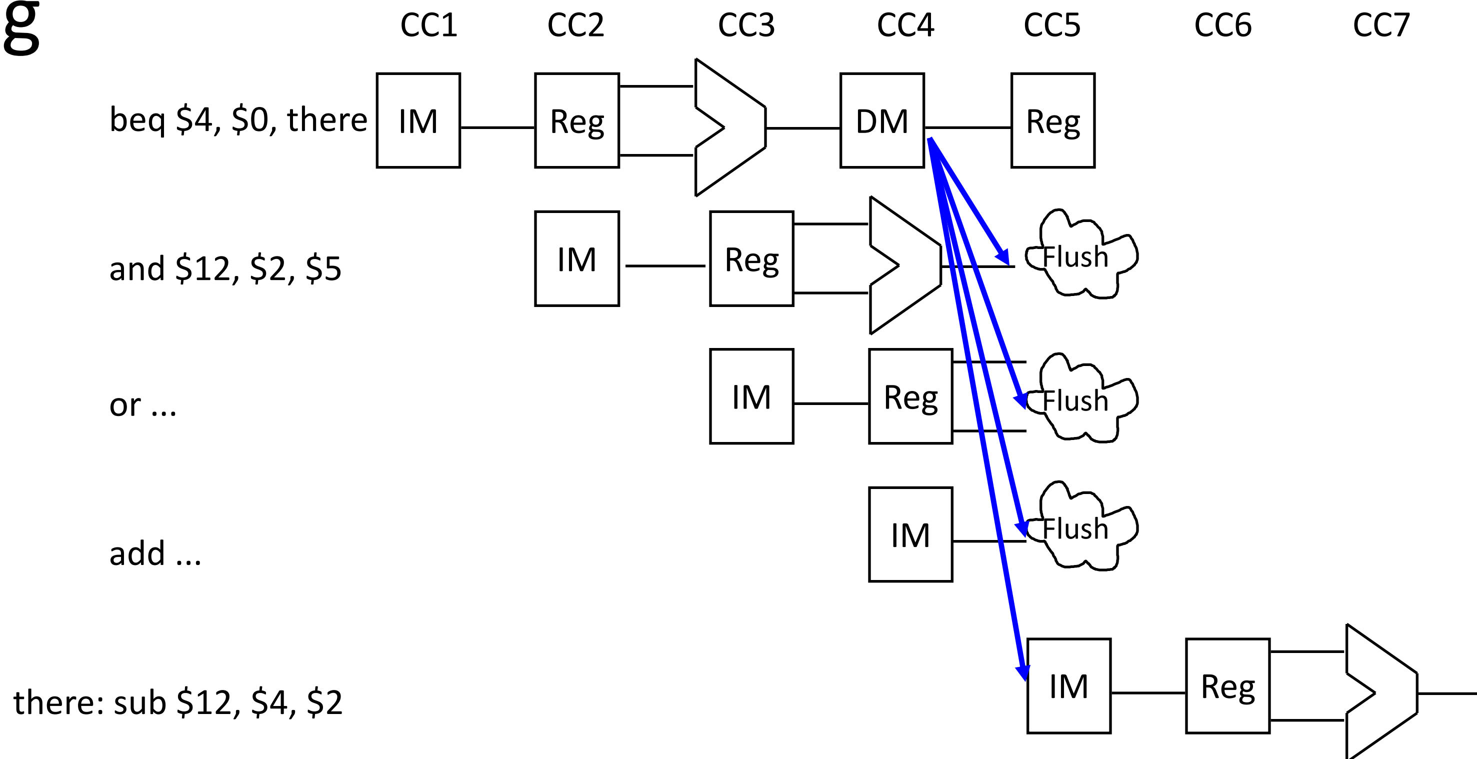
Assume Branch Not Taken

works pretty well when you're right



Assume Branch Not Taken

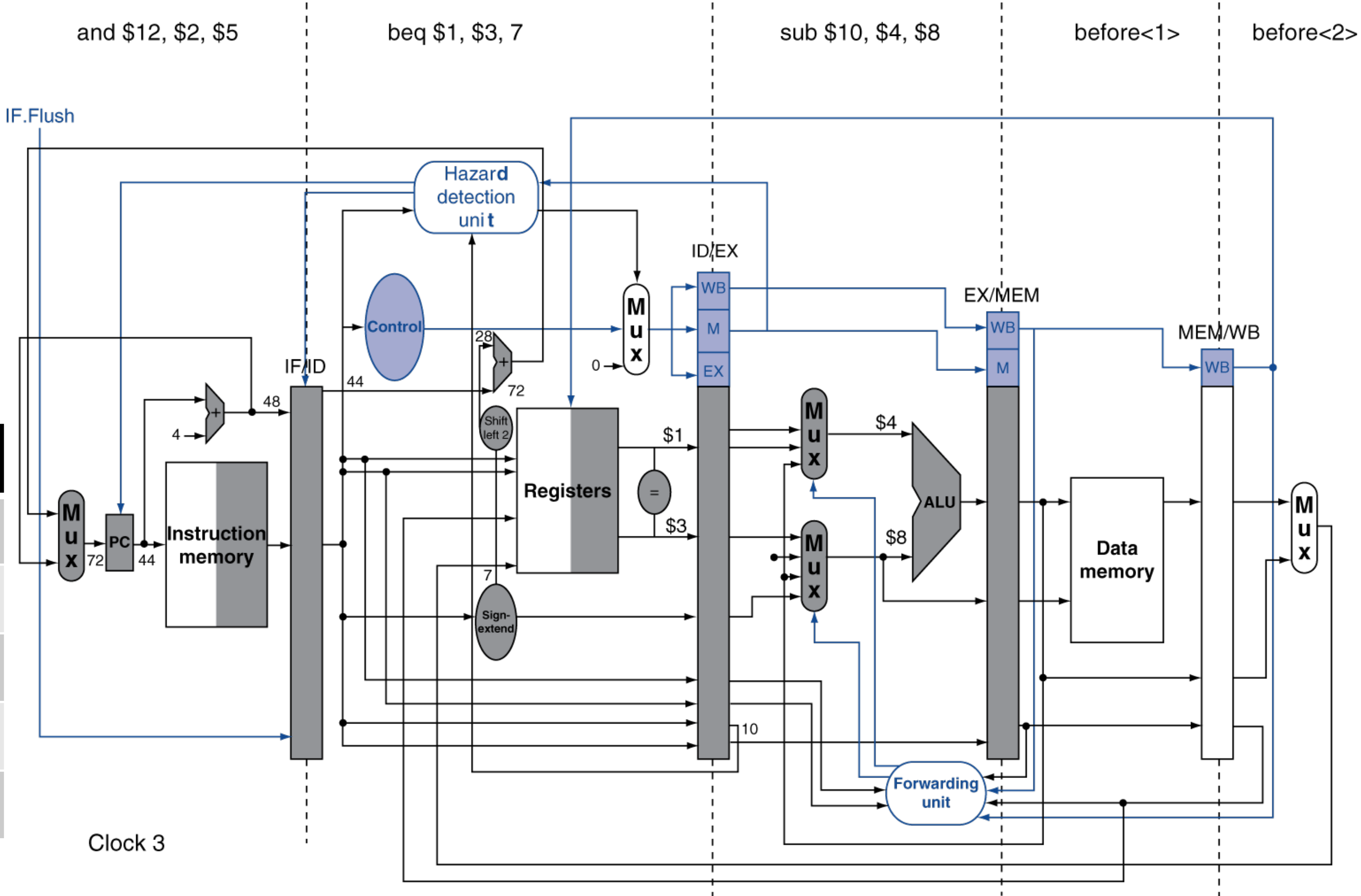
Flush the pipeline when you're wrong; same performance as stalling



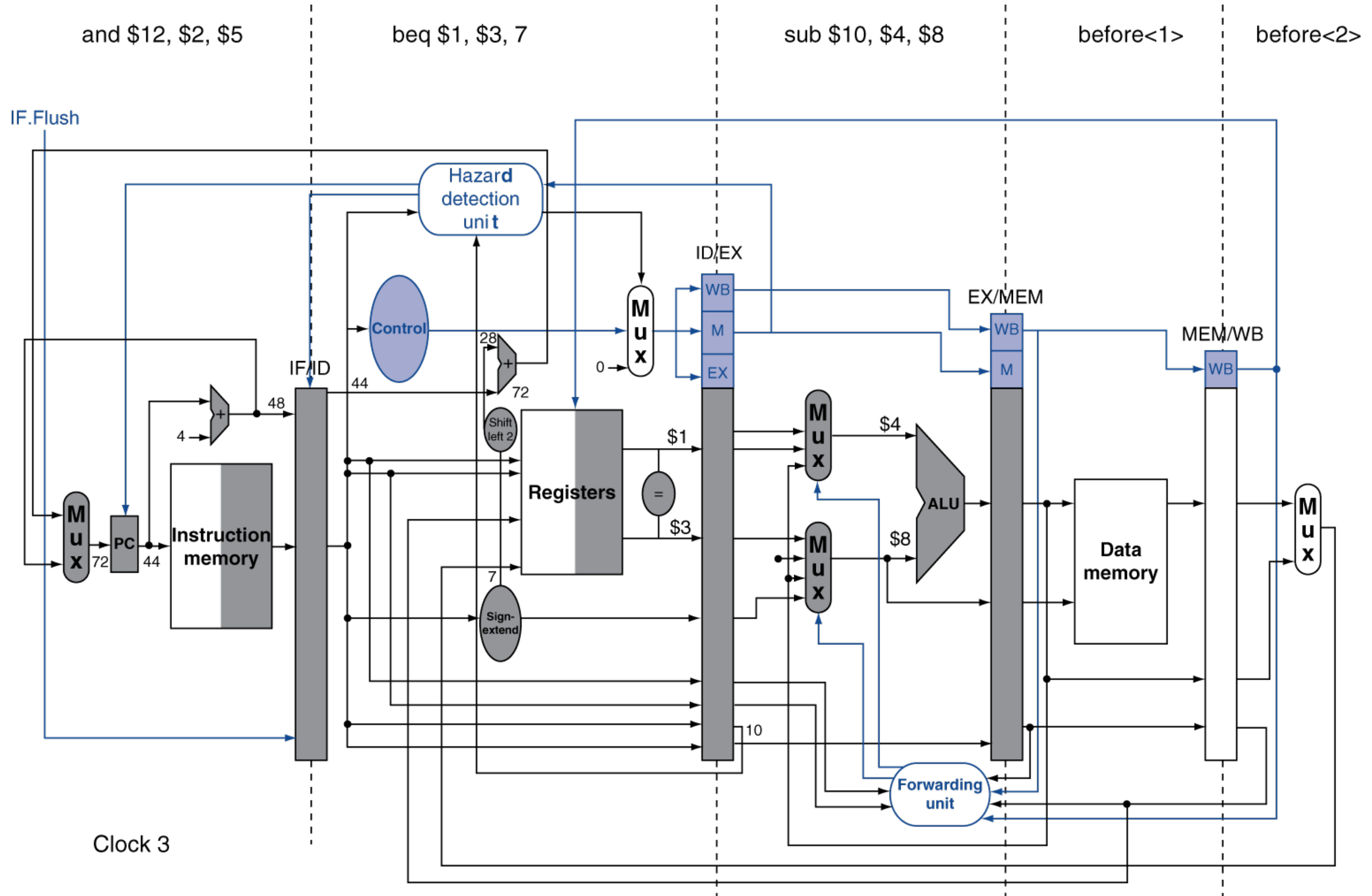
Stalling the pipeline

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?

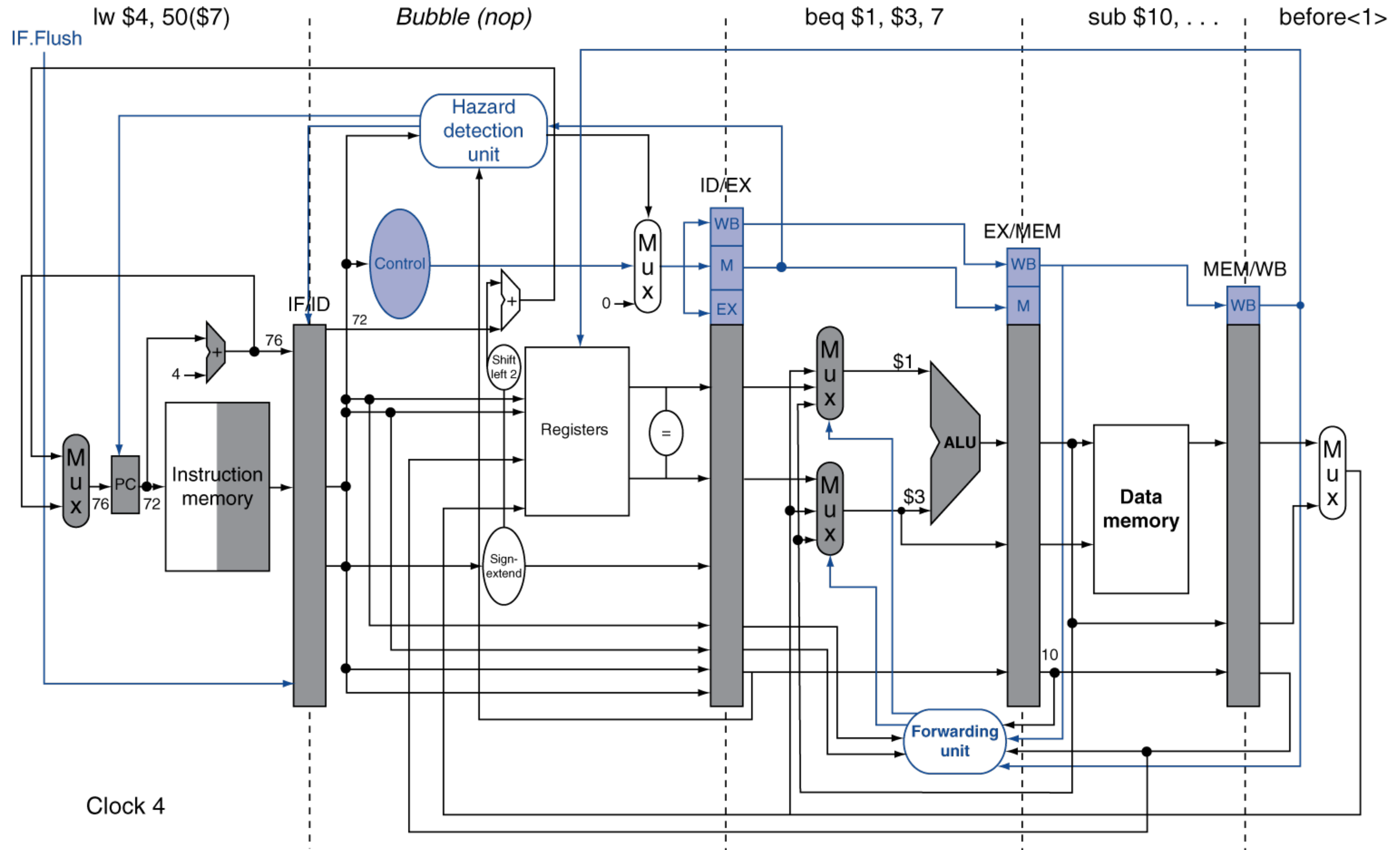
Selection	cycles
A	0
B	1
C	2
D	3
E	4



Example: Branch Taken



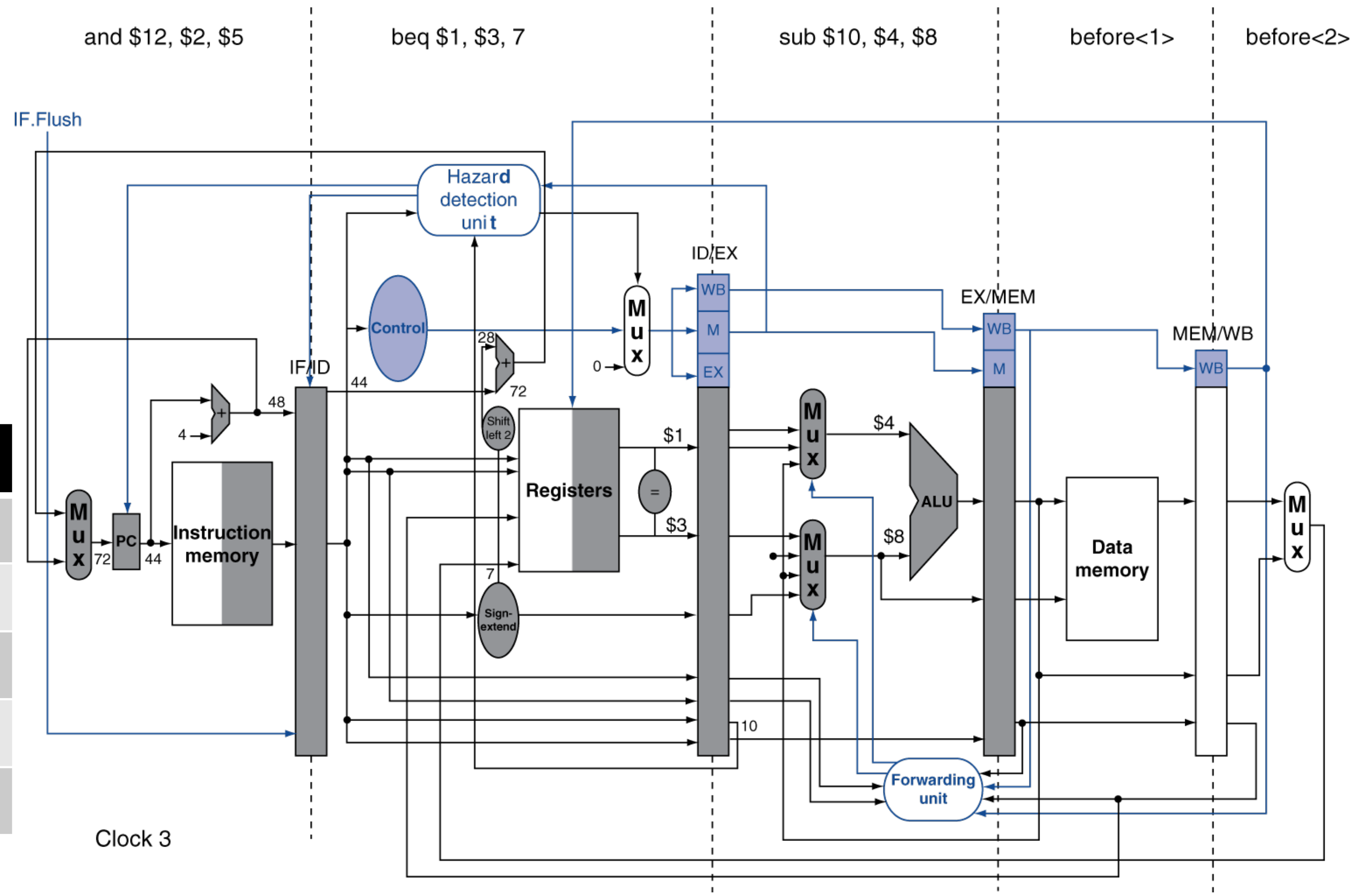
Example: Branch Taken



Stalling when branch not taken

How many cycles will we lose if the branch is not taken?

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



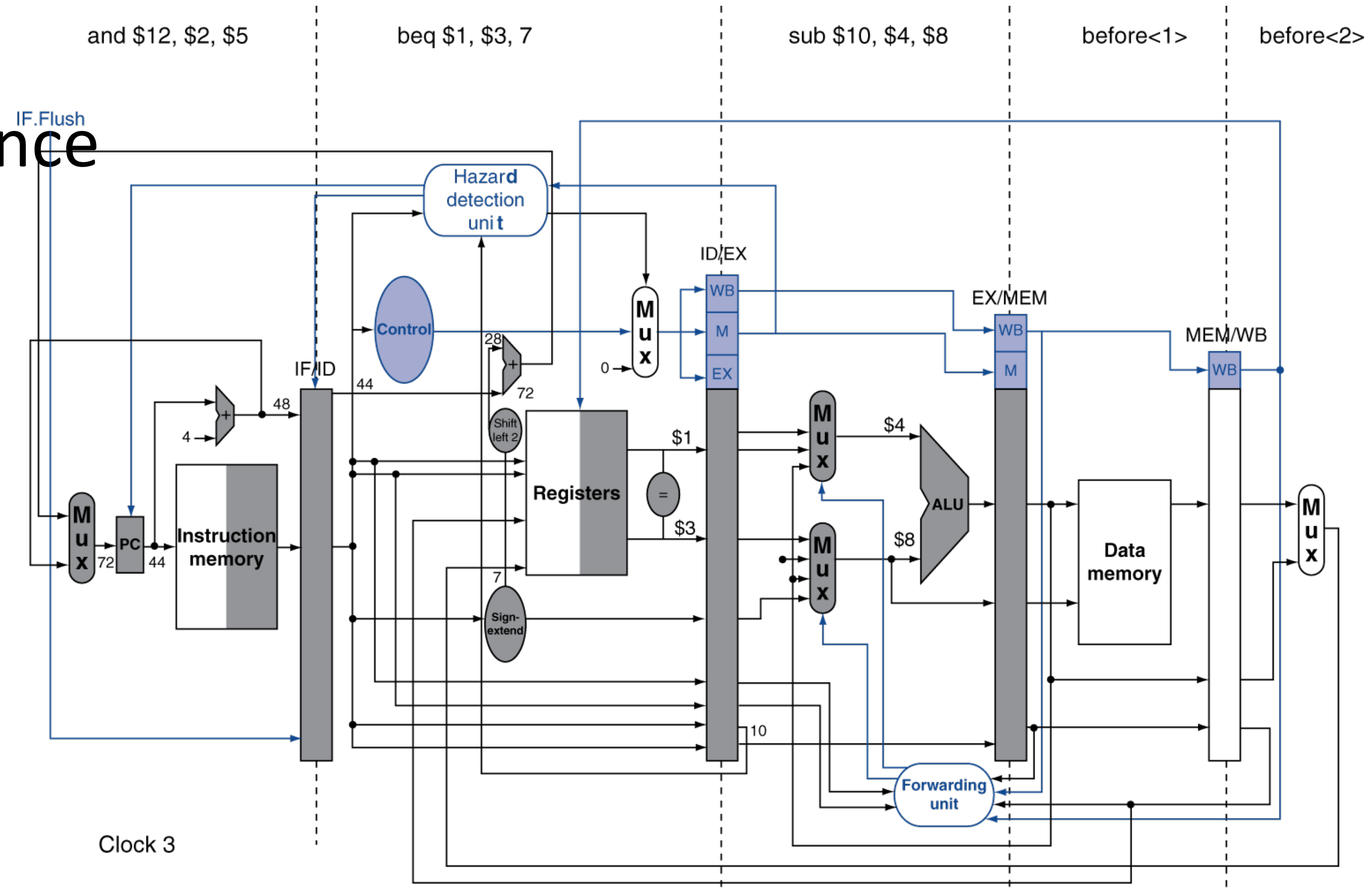
Branch depends on previous instruction

Consider the following instructions instead of the ones in the image

sub $1, $4, $8

beq $1, $3, target

Does this dependence cause a hazard?



A	No hazard
B	Structure hazard
C	Data hazard
D	Control hazard

Branch depends on previous instruction

Consider the following instructions instead of the ones in the image

sub $1, $4, $8

beq $1, $3, target

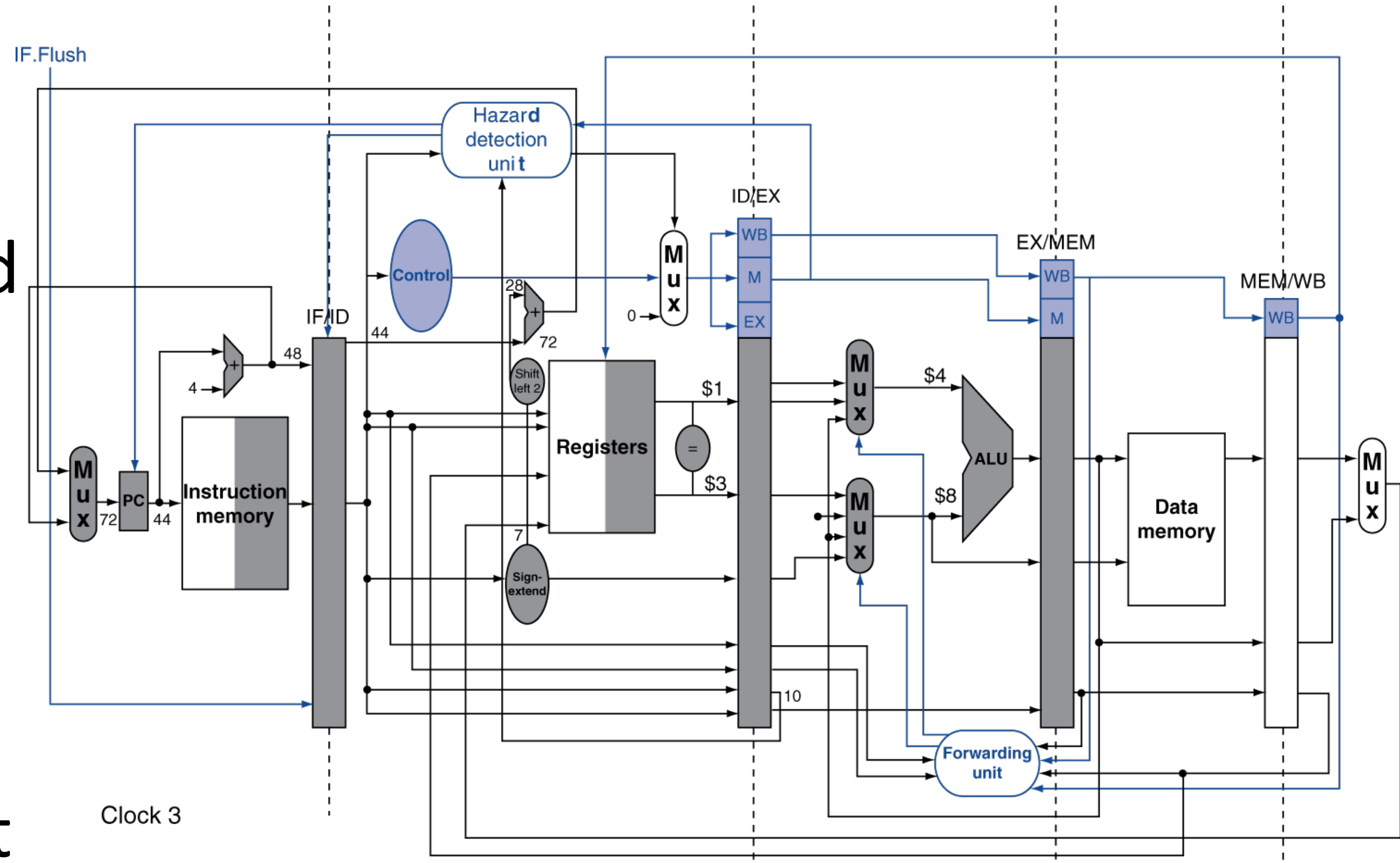
How do we deal with this hazard?

A	Stall until sub's WB
B	Forward from sub to beq
C	Stall 1 cycle, then forward
D	Stall 2 cycles, then forward

Changes to the pipeline

We've moved the branch detection to the ID stage but the values being compared might be computed later in the pipeline

We need new data path connections and forwarding and hazard detection logic



add $t0, __, __
sub $t1, __, __
beq $t0, $t1, target

Worst case for branching in the 5-stage pipeline

lw $t0, 0($s0)

beq $t0, $zero, target

and the branch is taken

We need to

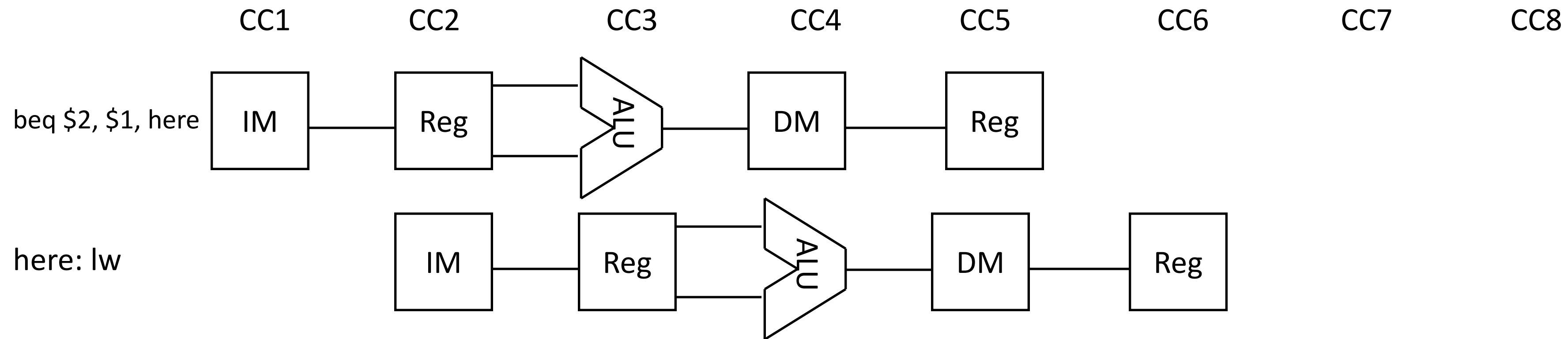
- stall for 2 cycles for the lw;
- bypass the register file to compare the just-loaded value; and
- flush 1 instruction from the pipeline (insert a bubble) for the taken branch

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 without stalls:

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.

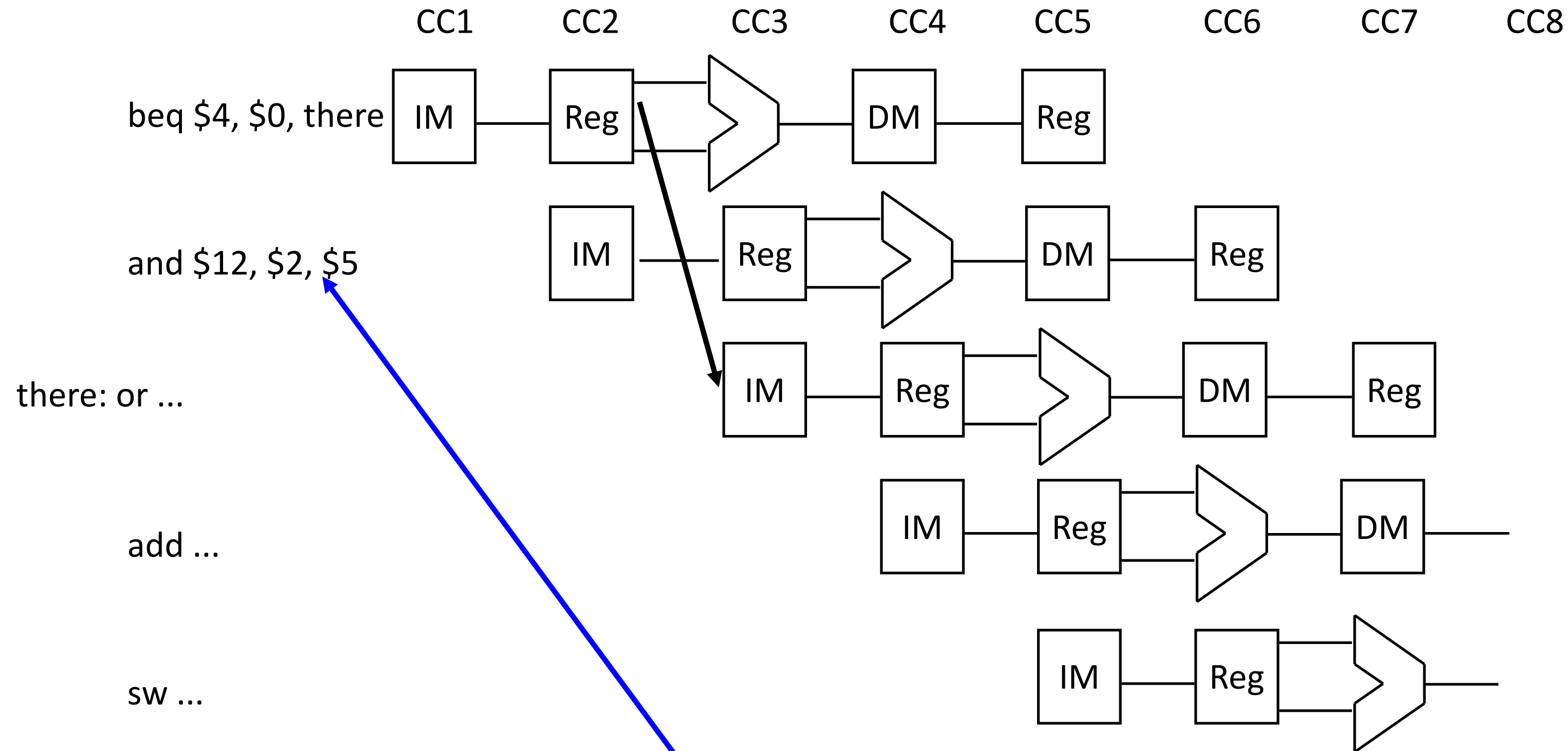
- Consult during Fetch (in parallel with Instruction Memory read) to determine:
  - Is this a branch?
  - If so, what is the target

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

Branch Hazards – Three Approaches

- 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

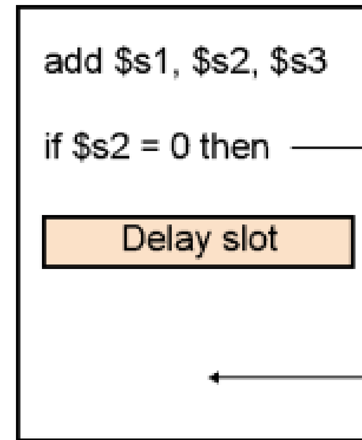
Filling the branch delay slot

#	Code	Comment
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 on not taken path)
	…	

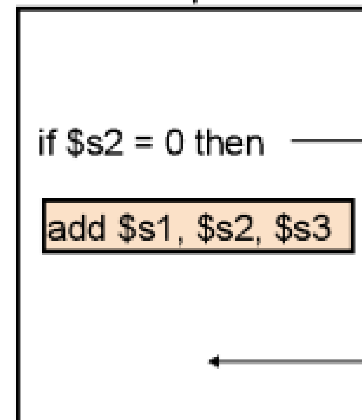
Filling the branch delay slot

- The branch delay slot is only useful if we can find something to put there.
- If the we can’t find anything, we must put a nop to ensure correctness.

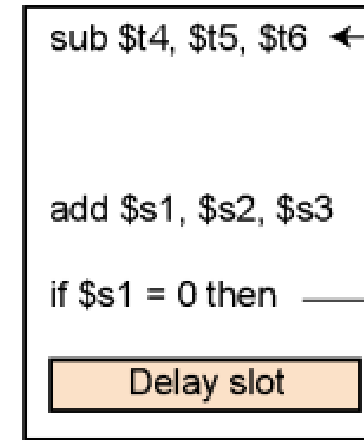
a. From before



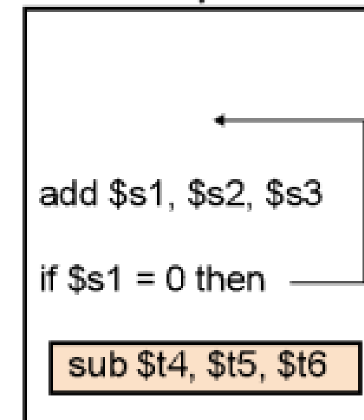
Becomes



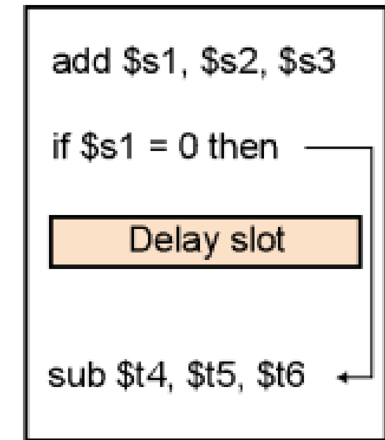
b. From target



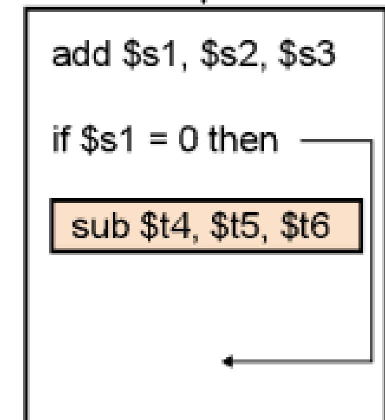
Becomes



c. From fall through



Becomes



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`

Branch Delay Slots

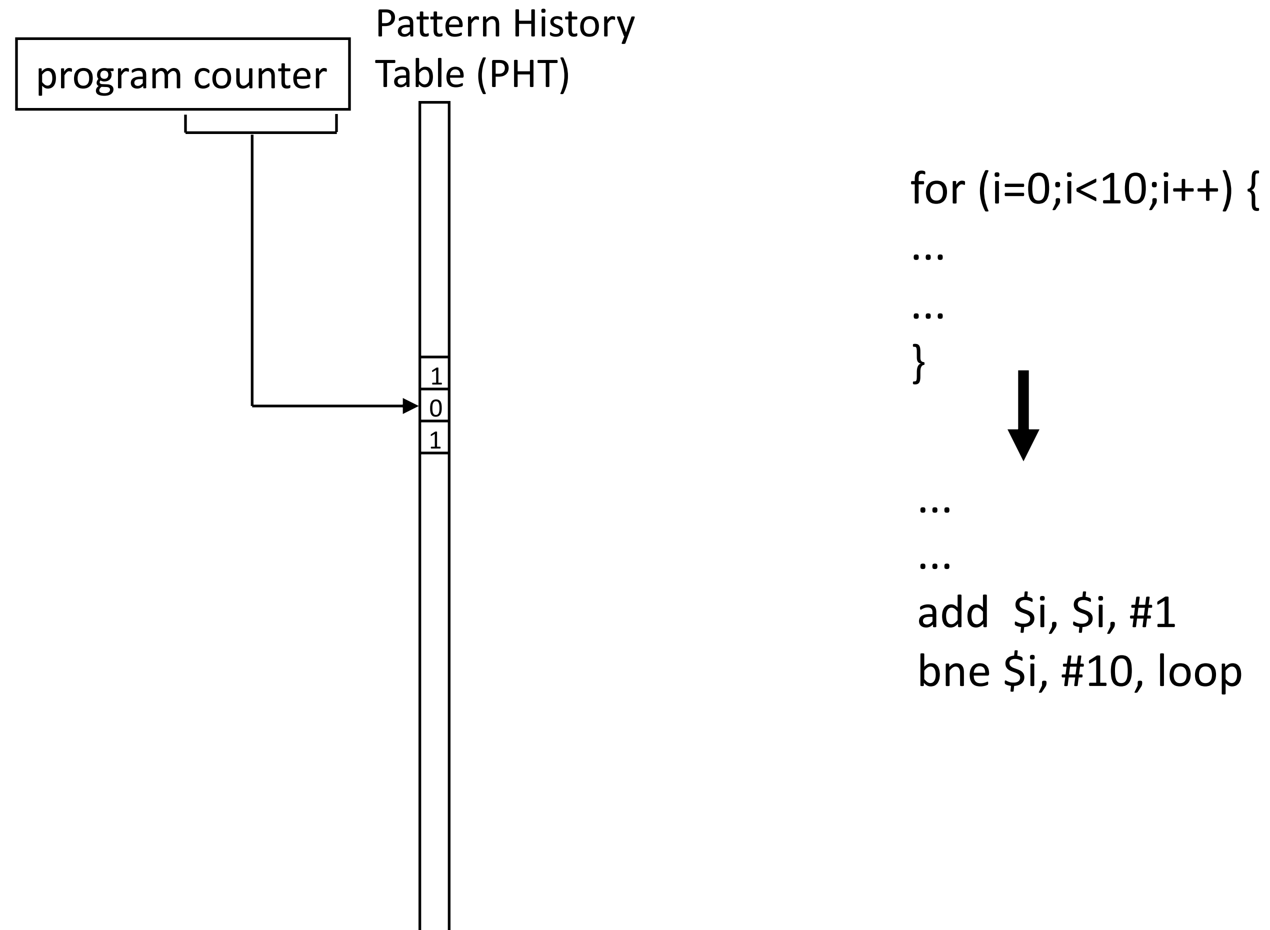
- This works great for this implementation of the architecture.

- 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

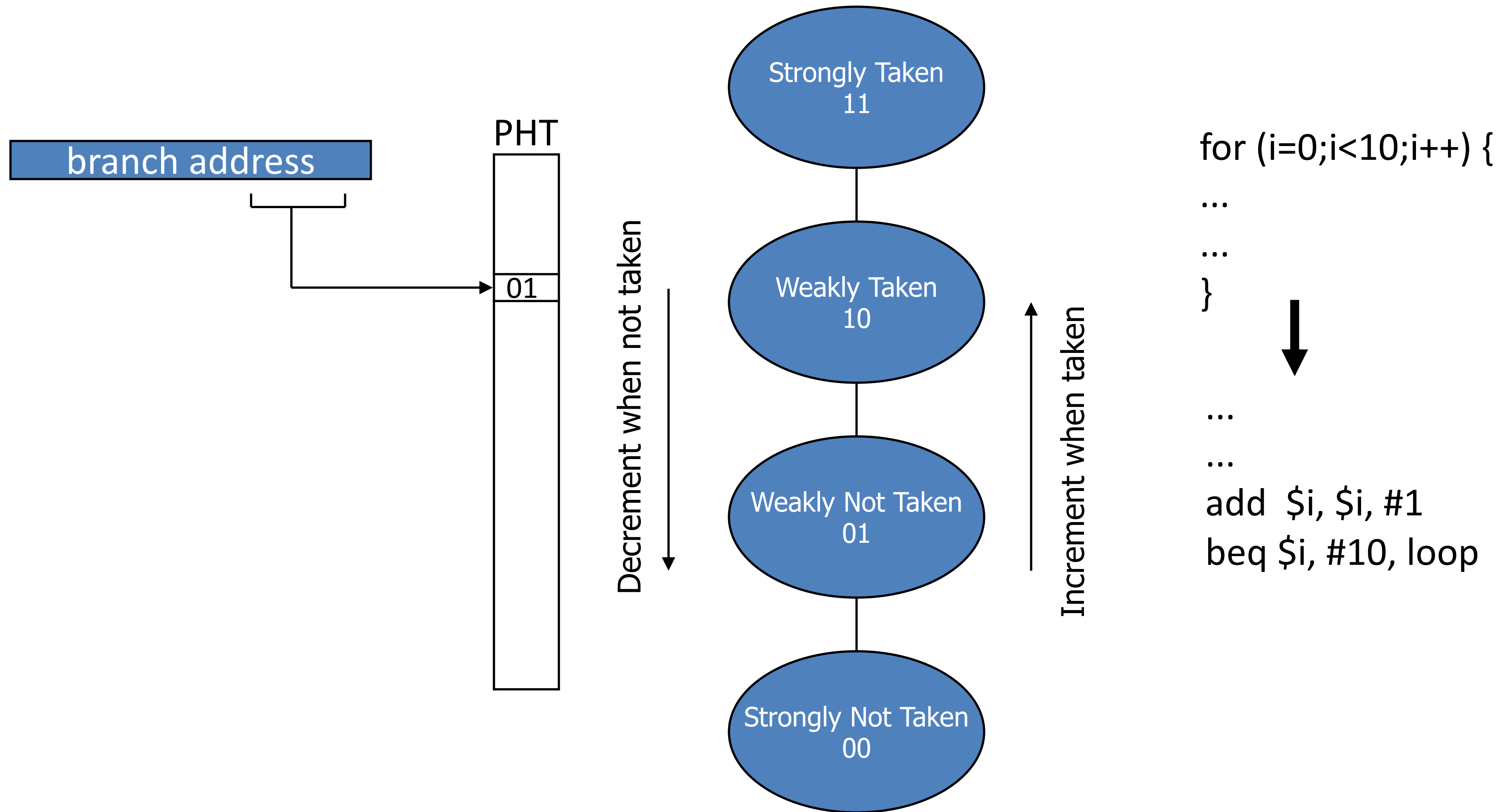


Every time branch is taken, set bit to 1, untaken, 0.

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 T T N T T N T T

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

Two-bit predictors give better loop prediction

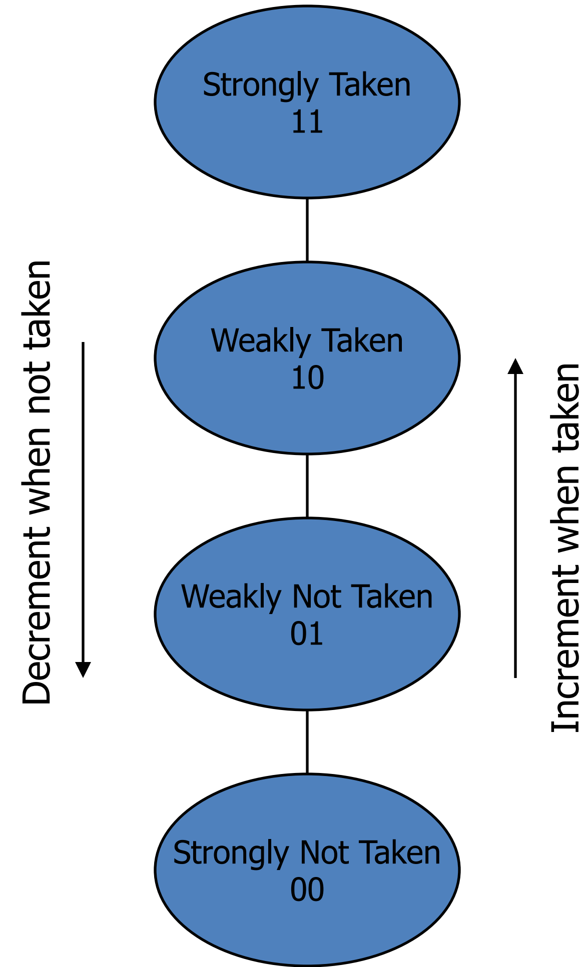


```
while (…) {
    for (int x = 0; x < 4; x += 1) {
        …
    }
}
```

The branch at the end of the for loop will have the following pattern: T T T N T T T N T T T N T T T N T T T N…

What values will a 1-bit and 2-bit branch predictor have after executing the inner for loop (e.g., after the 4 branches in red).

	1 bit	2 bit
A	0	00
B	1	01
C	0	10
D	1	11
E	0	It depends on the value of the predictor at the start

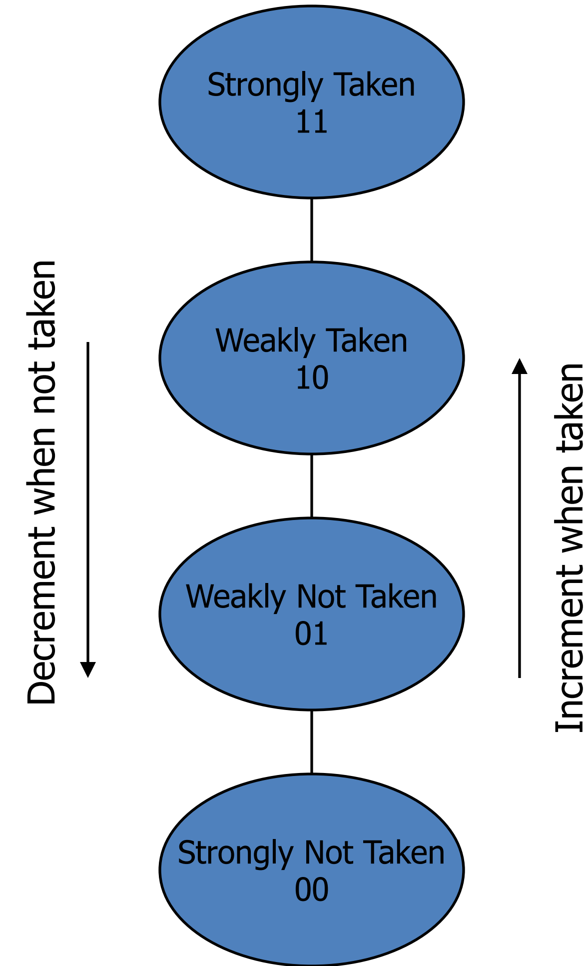


```
while (…) {
    for (int x = 0; x < 4; x += 1) {
        …
    }
}
```

The branch at the end of the for loop will have the following pattern: T T T N T T T N T T T N T T T N T T T N…

After the first time through the for loop, a 1-bit predictor will have the value 0 and a 2-bit predictor will have the value 10. Discounting the first 4 branches what are the accuracies of a 1-bit and a 2-bit predictor on the remaining branches?

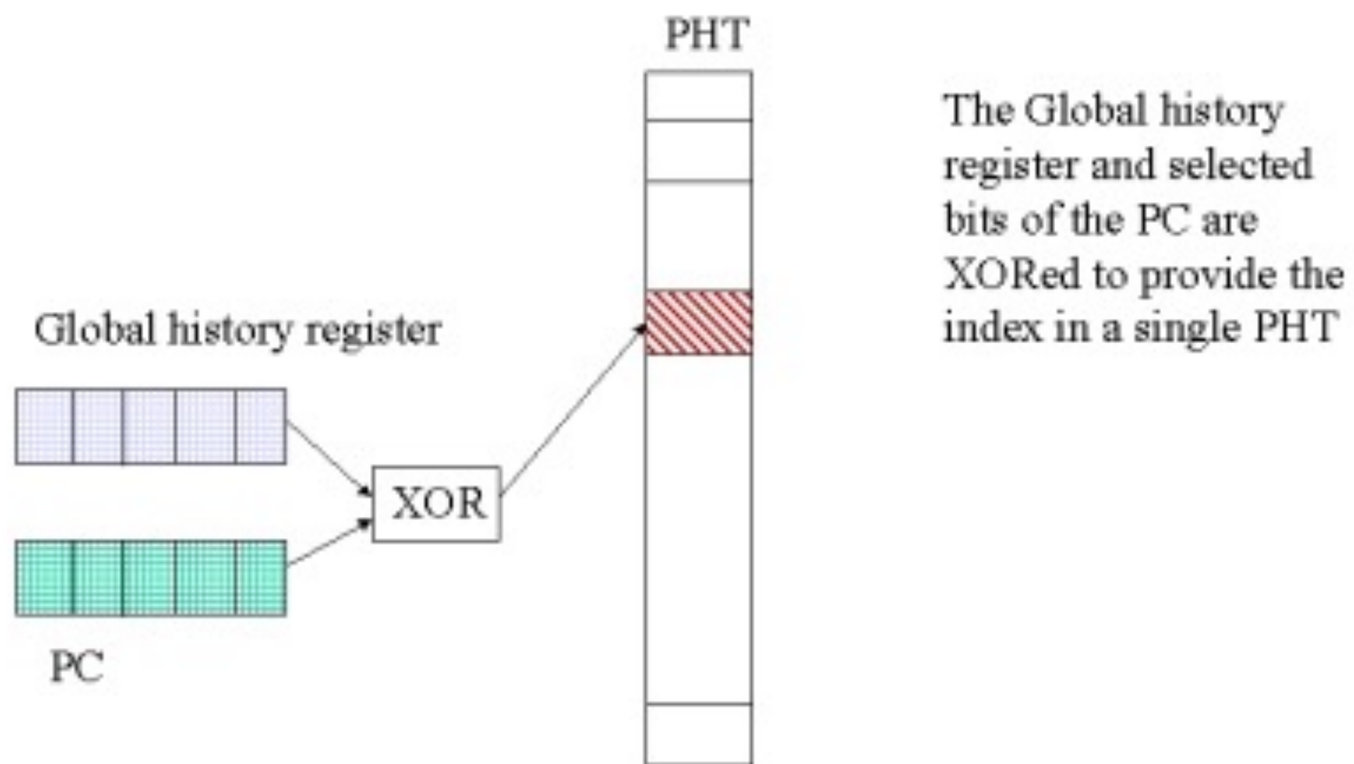
	1 bit	2 bit
A	25%	25%
B	25%	33%
C	33%	50%
D	50%	67%
E	50%	75%



Branch Prediction

- Latest branch predictors are significantly more sophisticated, using more advanced correlating techniques, larger structures, and even AI techniques (not generative AI!)

- Use patterns of branches (local history) and recent other branch history (global history) to make predictions
  - E.g., “gshare” predictor takes a global branch history and XORs that with the PC to look up a 2-bit saturating counter in the PHT (pattern history table). Works shockingly well



Slide shamelessly borrowed from CSE 586 at the University of Washington, spring 2000

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-use 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 (i.e., 1 instruction completes every cycle unless there's a stall)

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

Questions on Branch Prediction/Pipelining?

Control Hazards — Key Points

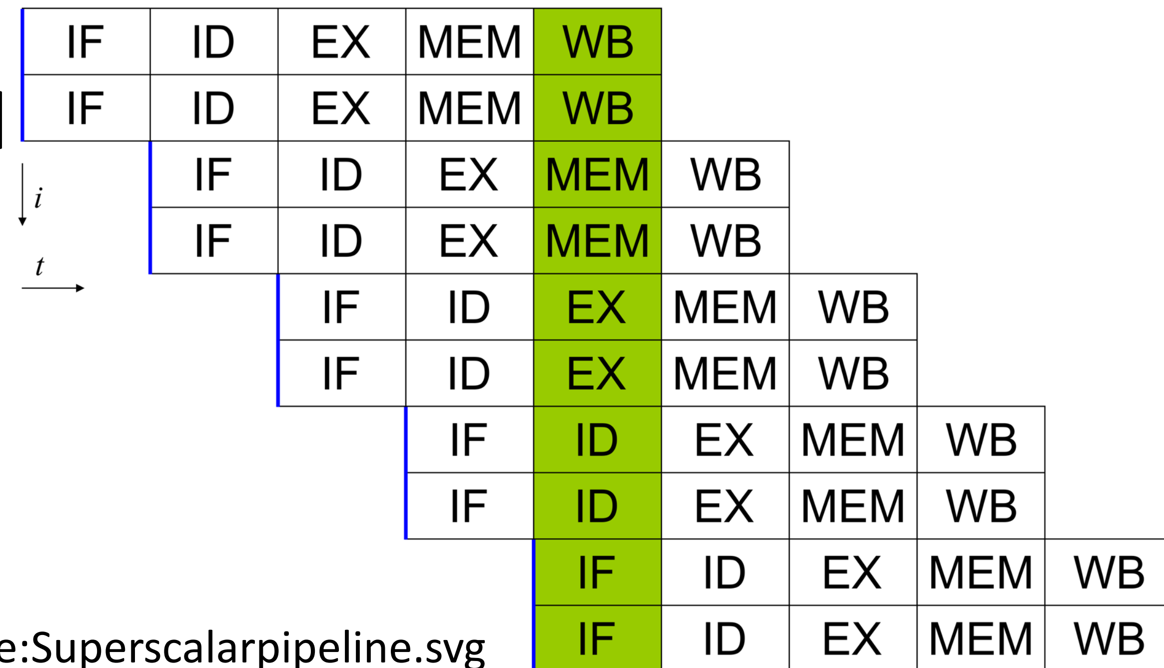
- Control (or branch) hazards arise because we must fetch the next instruction before we know if we are branching or where we are branching.
- Control hazards are detected in hardware.
- We can reduce the impact of control hazards through:
  - early detection of branch address and condition
  - branch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

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

Bonus material: superscalar processor

- Multiple instructions fetched and start executing per cycle
- Multiple **functional units** like ALUs
- During instruction decode, instructions are “issued” to different functional units
- The processor is usually pipelined as well (but need not be)



https://en.wikipedia.org/wiki/Superscalar_processor#/media/File:Superscalarpipeline.svg

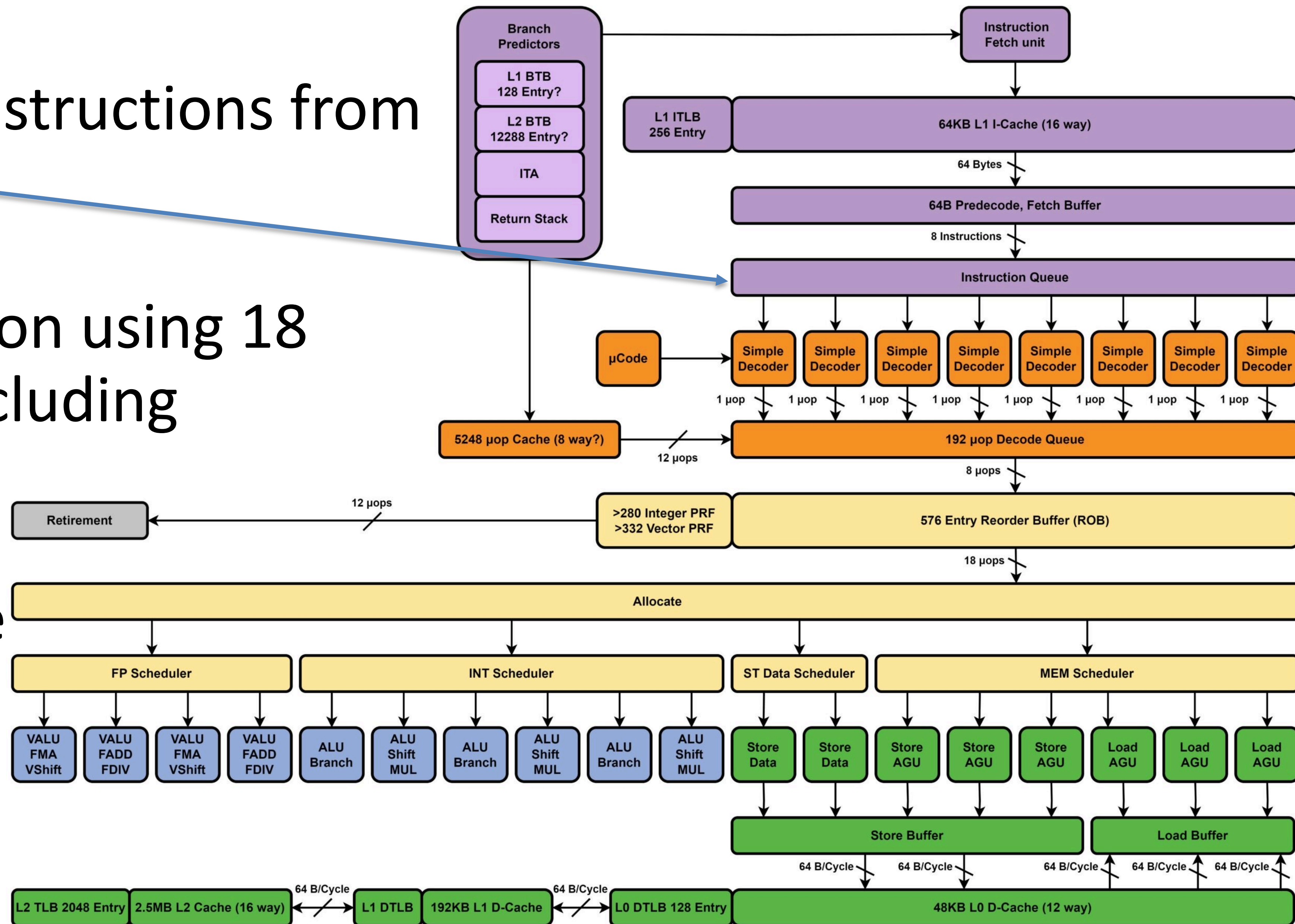
In-order, superscalar example: Intel Pentium

- Two general purpose integer pipelines (called U and V), one pipelined floating point unit
- Two instructions issued per clock cycle (if possible), one per pipeline
- Functional units
  - 2 ALUs (U and V)
  - 1 Multiplier (shared between U and V)
  - 1 Shifter (shared between U and V)
  - 1 Memory access (U only)

Intel Lion Cove microarchitecture

8-way decoding of instructions from instruction queue

Out-of-order execution using 18 "execution ports" including
- 6 ALUs
- 3 Multipliers
- separate load/store units



Bonus material: Simultaneous Multithreading (SMT)

- Multiple (usually 2), independent “threads” of execution make use of the various functional units/execution ports
- Intel calls this hyperthreading
- The different threads compete for resources but can improve overall system performance by exploiting the fact that instructions from independent threads don’t depend on each other

Bonus material: Single-instruction multiple data (SIMD)

- “Vector” units perform the same operation on multiple pieces of data simultaneously
- Intel has SSE, SSE2, SSE3, SSE4, AVX, AVX2, AVX-512, and others
- ARM has Neon
- Details differ but principle is the same: Divide, for example, 128-bit registers into four 32-bit (integer or floating point) values, perform the same operation (e.g., add) on corresponding pairs