CMPE110 Lecture 13
Pipelining 3

Heiner Litz

https://canvas.ucsc.edu/courses/19290

Announcements

- **Evaluation of the Tensor Processing Unit: A Deep Neural Network Accelerator for the Datacenter**
- David Patterson, UC Berkeley & Google
- Friday, February 8, 2:40 PM, E2-180

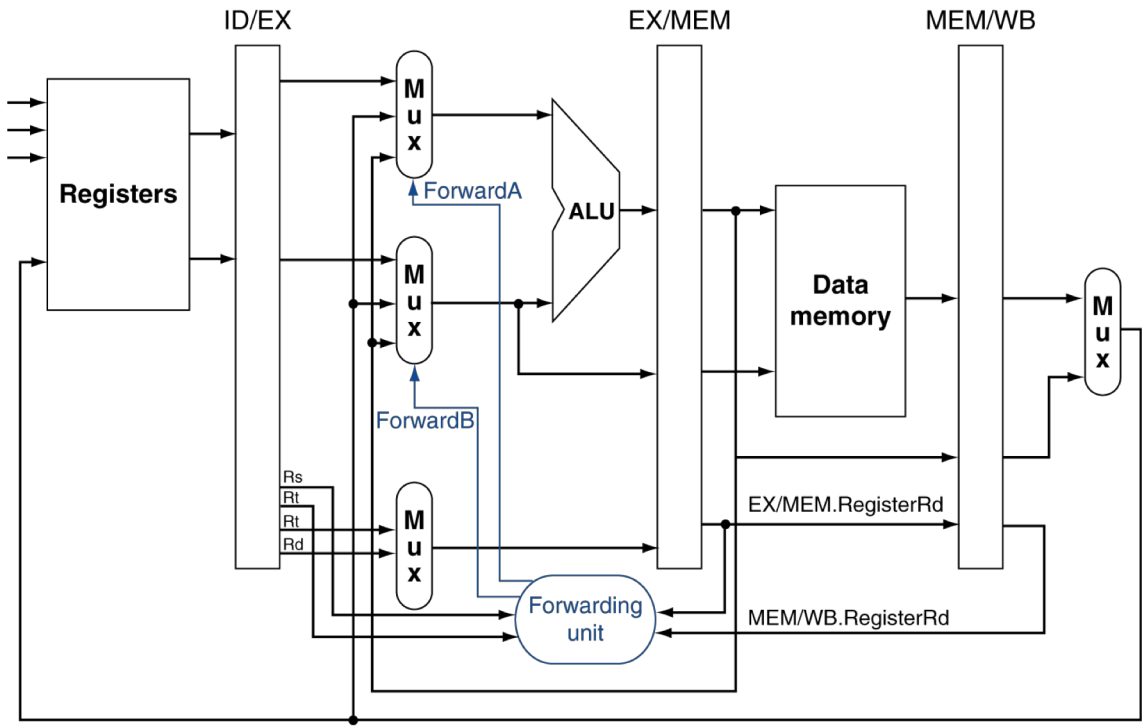
Announcements

- No Quiz on Friday
- Midterm: Wed 02/13
  - 1 double sided page of notes
  - RISC-V Green card/cheat sheet part of the exam

Review

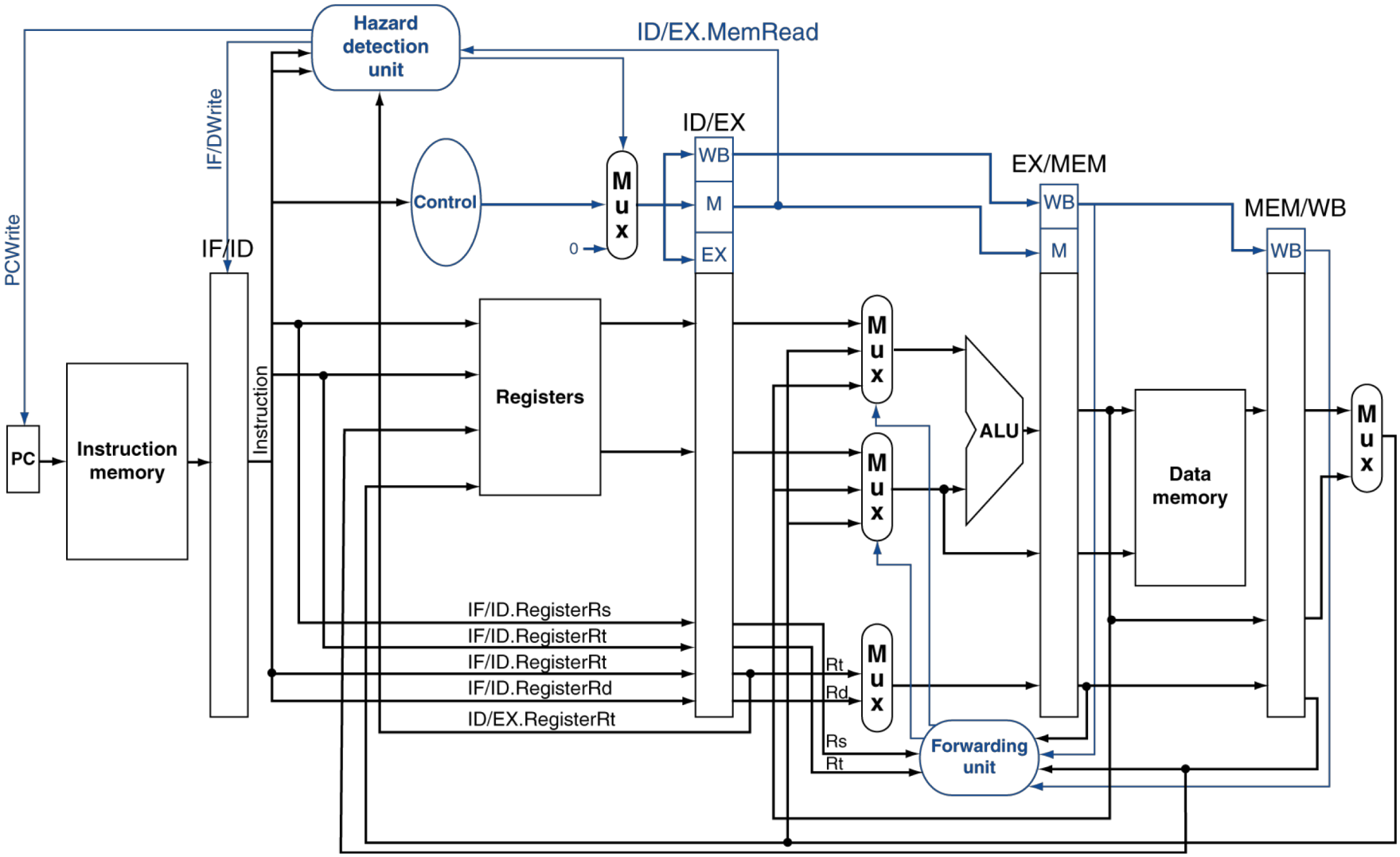
Forwarding Paths: Example

```
Example 1:
add $1,$1,$2
or  $1,$1,$4

Example 2:
add $1,$1,$2
sub $4,$5,$3
or  $1,$1,$4

Example 3:
Ld  $1,0($2)
sub $4,$5,$3
or  $1,$1,$4
```



b. With forwarding

Datapath with Hazard Detection



Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for `A = B + E; C = B + F;`

stall

stall

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
lw   $t4, 8($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

9 cycles

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
lw   $t4, 8($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

7 cycles

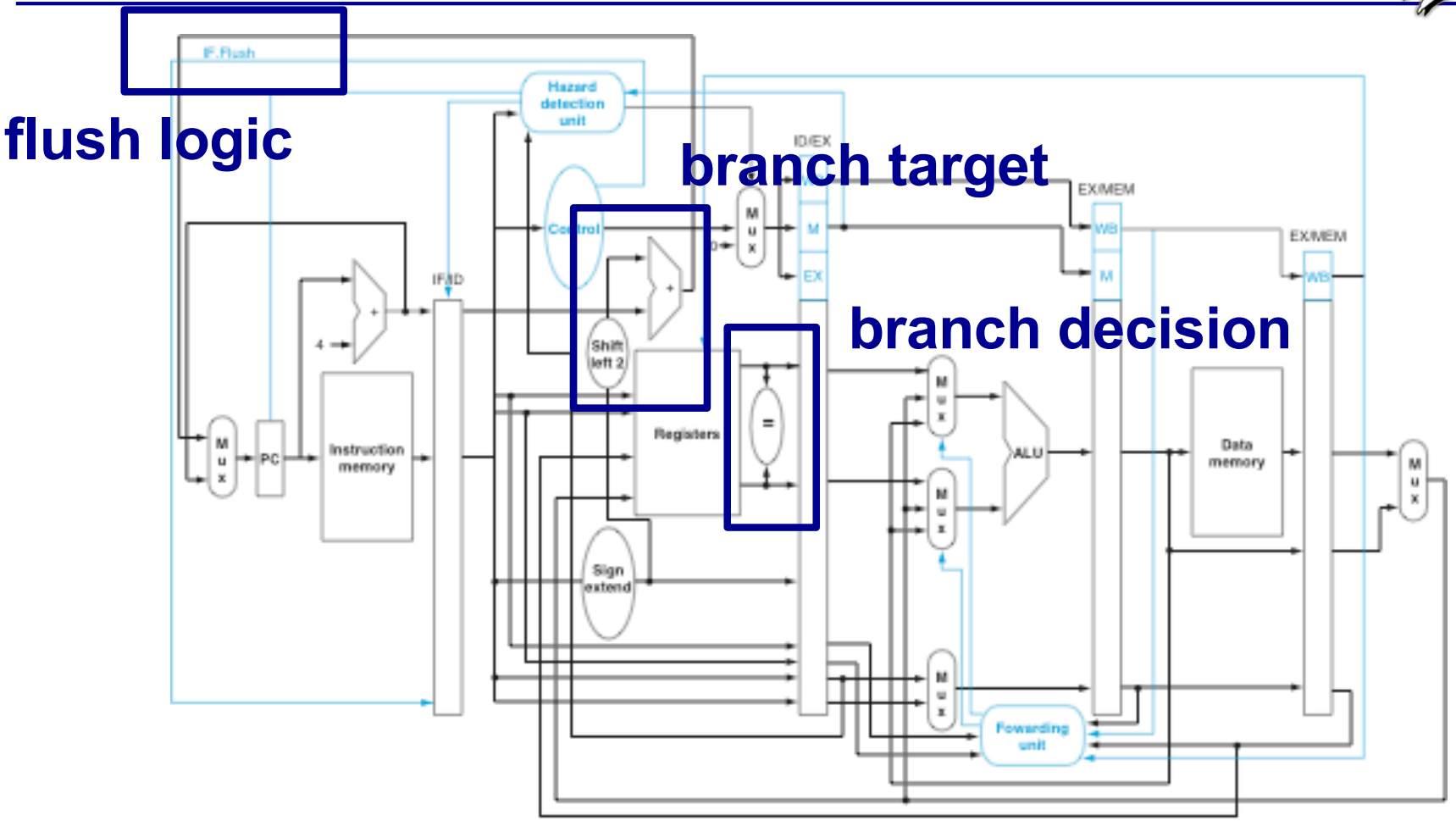
Control Hazards

- Branches get computed in the EX stage
- Requires 2 NOPs after every branch ☹
- Can we do better?

Reducing Branch Delay

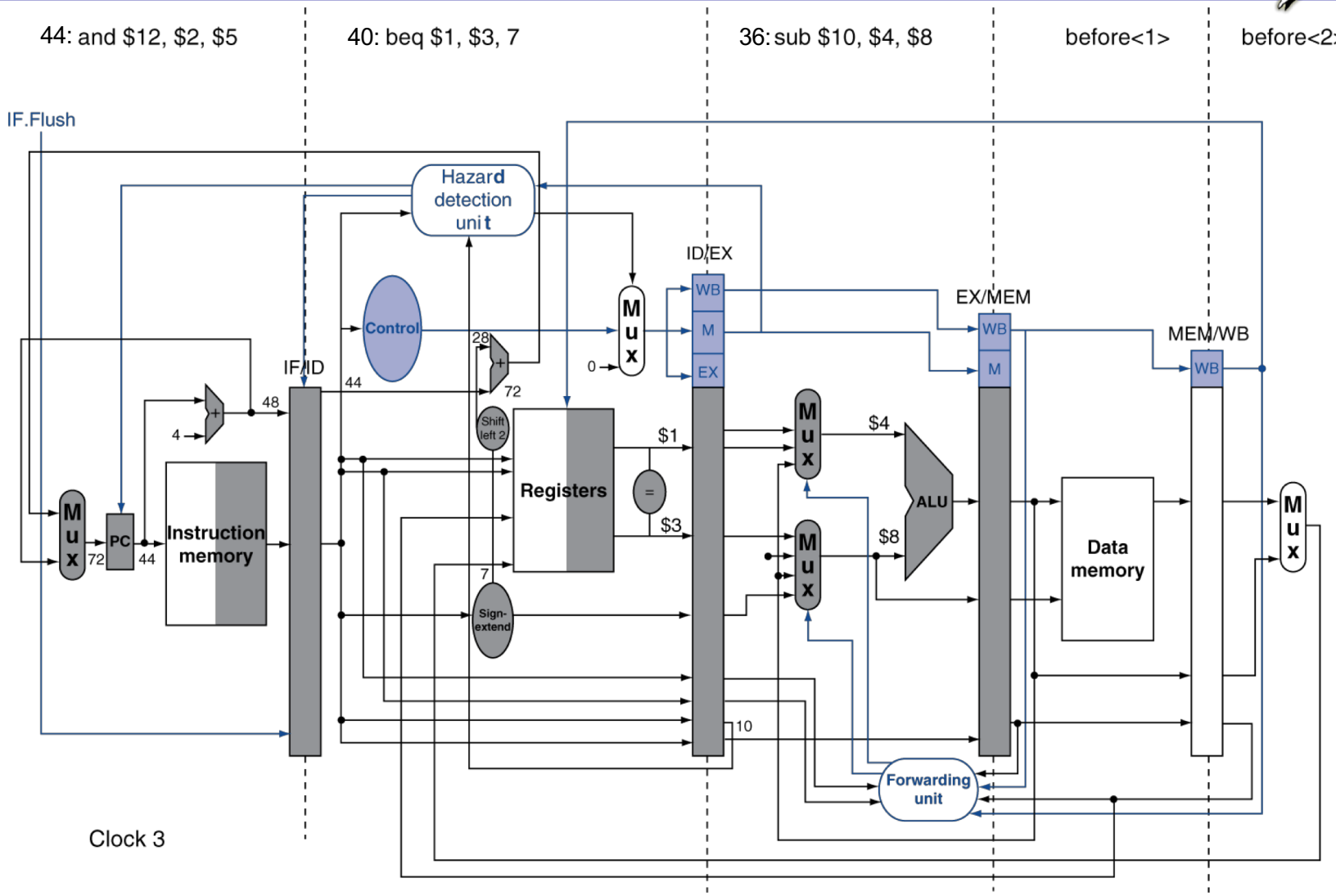
- Minimize “bubble” slots
  - Move branch computation earlier in the pipeline
  - branch outcome: add comparator to ID stage
  - branch target: add adder to ID stage
- Predict branch not taken
  - if correct, no bubbles inserted
  - if wrong, flush pipe, inserting one bubble

Reducing Branch Delay

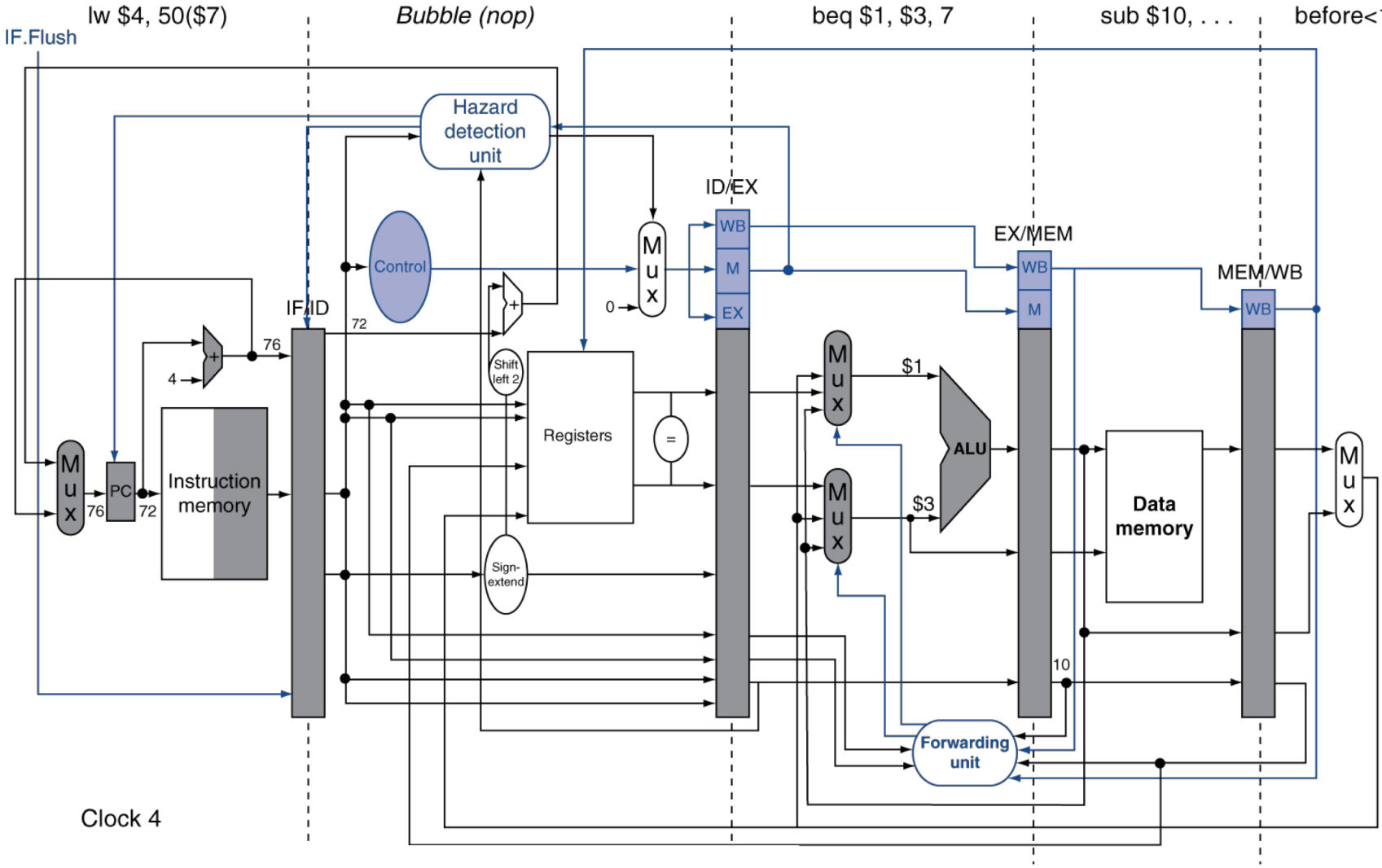


move branch computation earlier in the pipeline
enable flush capability if mispredicted

Example: Branch Taken



Example: Branch Taken



nop inserted using IF.Flush

How to Flush?

- 5-stage Pipeline
  - only one instruction to flush
  - Can do by “flushing” the output register of IF
- Deeper Pipelines
  - Multiple cycles to determine branch outcome
  - Need to know how many instructions in the pipe
  - Flush all state changing actions
    - RegWrite, MemWrite, PCWrite (jmp, beq), overflow..

Performance Impact of Branch Stalls

- Need to stall for one cycle on every branch
- Consider the following case
  - The ideal CPI of the machine is 1
  - The branch causes a stall
- Effective CPI if 15% of the instructions are branches?
  - The new effective CPI is $1 + 1 \times 0.15 = 1.15$
  - The old effective CPI was $1 + 3 \times 0.15 = 1.45$

Delayed Branches

- An ISA-based solution used in early MIPS machines
  - Had a branch delay of one

- Example:

beq r1, r2, L	Branch instruction
sub r4, r1, r3	This operation ALWAYS is executed
and r6, r2, r7	This operation executes if branch fails

- Worked well initially
  - Compiler can fill one slot 50% of the time (50% nops)
- For modern processors, it is a pain
  - Many delay slots needed due to deeper pipelines
  - Processors deal with this using branch prediction
  - Delayed branches complicate exceptions as well

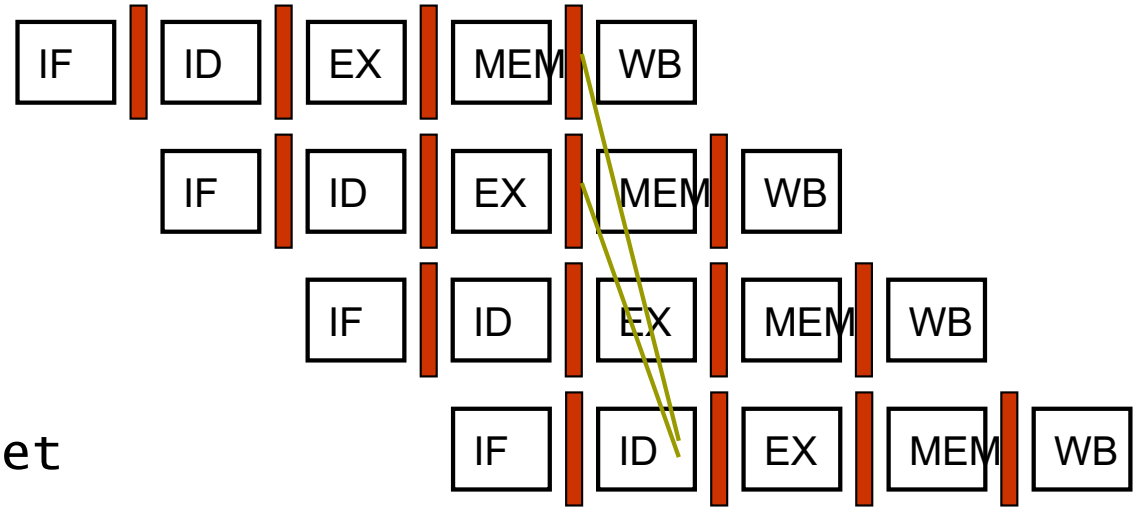
It's All Connected

- We moved the branch control point to ID
- Which means we consume two registers in ID
  - earlier in the pipeline as before (EX)
- What does this mean for forwarding/stalls?
  - Need to check!!

Data Hazards for Branches

- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction

```
add $1, $2, $3
add $4, $5, $6
…
beq $1, $4, target
```

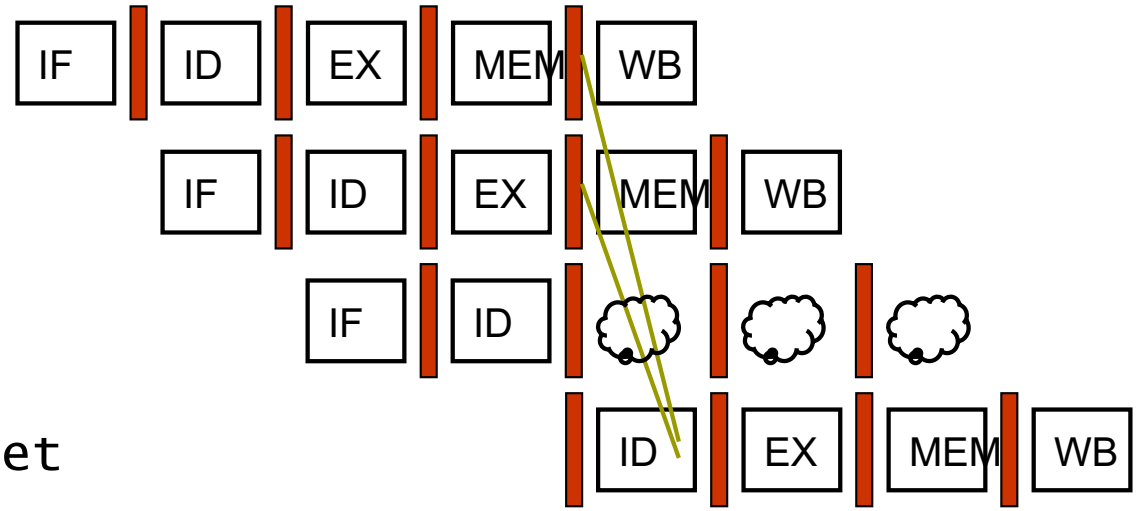


- Can resolve using forwarding
  - Additional datapaths and control

Data Hazards for Branches

- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
  - Need 1 stall cycle

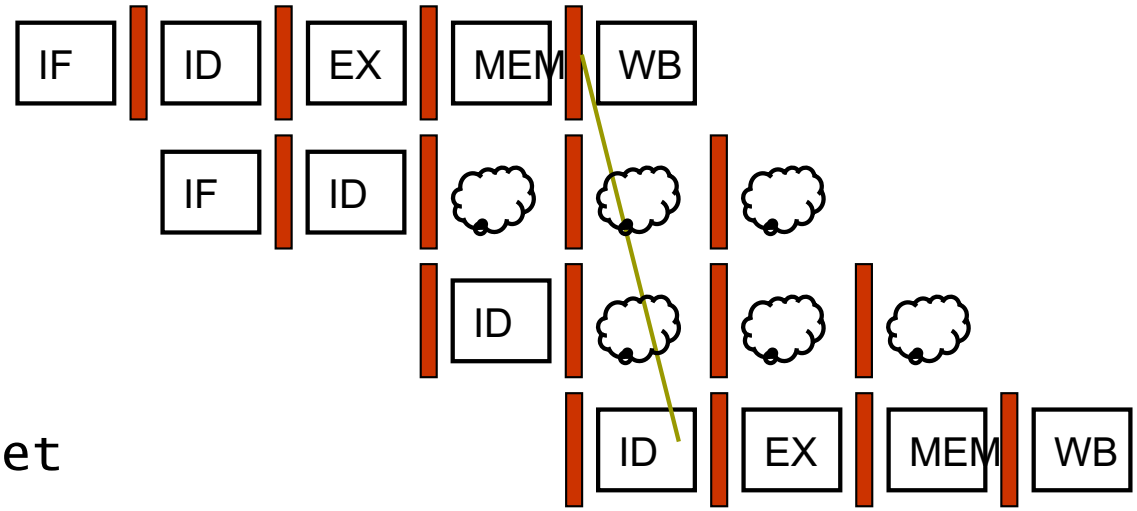
```
lw   $1, addr
add  $4, $5, $6
beq  stalled
beq  $1, $4, target
```



Data Hazards for Branches

- If a comparison register is a destination of immediately preceding load instruction
  - Need 2 stall cycles

```
lw   $1, addr
beq  stalled
beq  stalled
beq  $1, $0, target
```



Branch CPI

15% Branch frequency

65% branches are taken

50% of single delay slots are filled usefully

Control point	Branch strategy	Branch Stall CPI
MEM	stall	
ID	stall	
ID	Predict taken	
ID	Predict Not taken	
ID (target) / MEM (==)	Predict taken	
ID	Delayed branch	

Branch CPI

15% Branch frequency

65% branches are taken

50% of single delay slots are filled usefully

Control point	Branch strategy	Branch Stall CPI
MEM	stall	3 • 0.15 = 0.45
ID	stall	1 • 0.15 = 0.15
ID	Predict taken	1 • 0.15 = 0.15
ID	Predict Not taken	(1 • 0.15)•0.65 + 0•0.35 = 0.098
ID (target) / MEM (==)	Predict taken	(1 • 0.15)•0.65 + (3 • 0.15) •0.35 = 0.255
ID	Delayed branch	(1 • 0.15)•0.50 = 0.075

Branch Prediction

- Static prediction vs. dynamic prediction

- Static prediction schemes:
  - always predict taken
  - always predict not-taken
  - compiler/programmer hint
  - if (target < PC)
    predict taken
    else
    predict not-taken

What's the rationale behind this?

Dynamic Branch Prediction

- Branch History Table (BHT)
  - One entry for each branch PC
  - Taken/Not taken bit
- Branch Target Buffer (BTB)
  - One entry for each branch PC
  - Target address
- Increasingly important for long pipelines (IDx)
  - x86 vs. RISC-V instruction decode

Branch Prediction Example


```
void foo() {
  for(e=0;e<4;e++) {
    stuff …
  }
}

foo:j E
L:  stuff..
    ..
E:  bne $t3,$t4,L
```

How many times will bne at E: be predicted correctly?

Two-bit saturating counter:
strongly taken, weakly taken, weakly n-t, strongly n-t