CMPE110 Lecture 10
Pipelining II

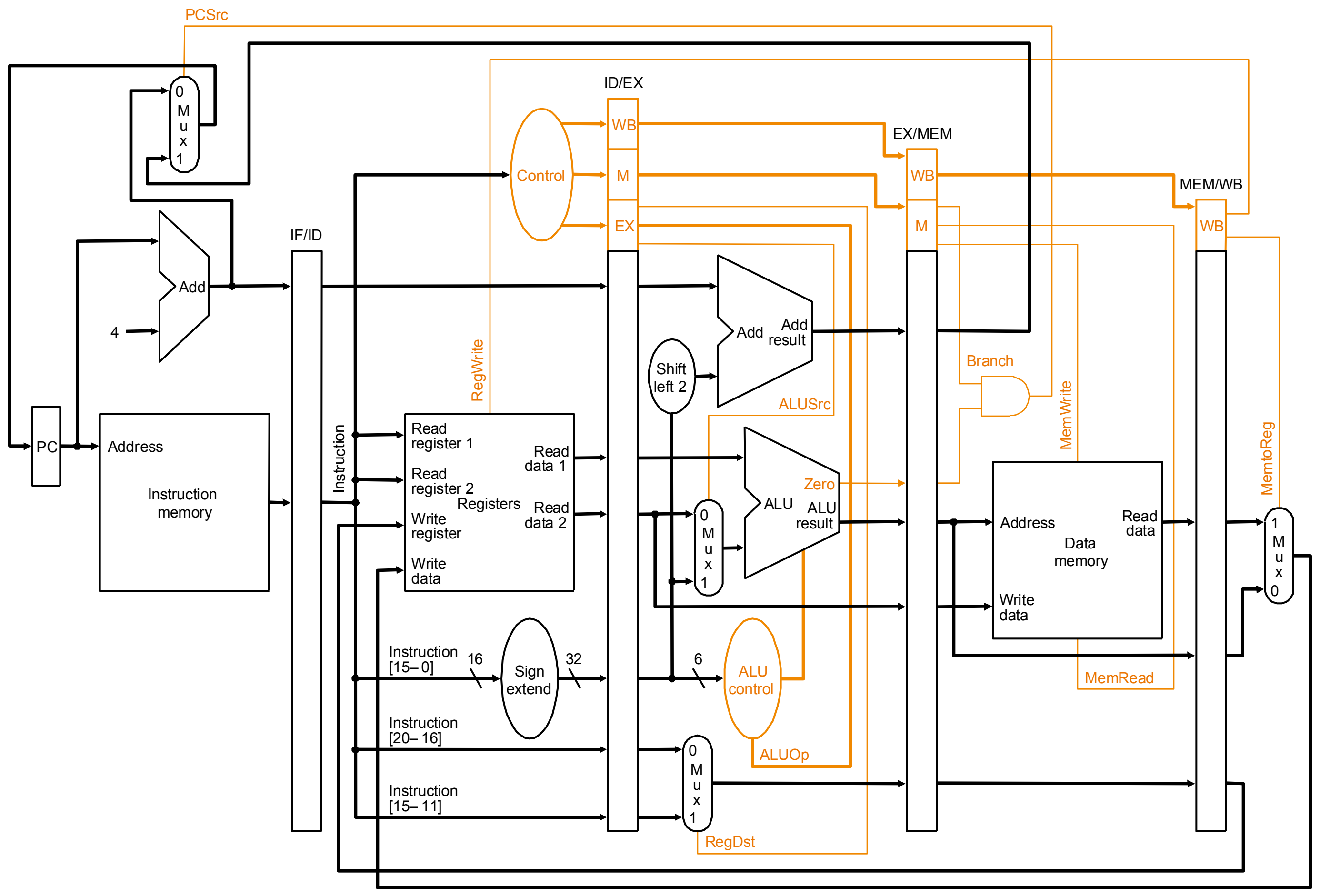
Heiner Litz

https://canvas.ucsc.edu/courses/12652

Announcements

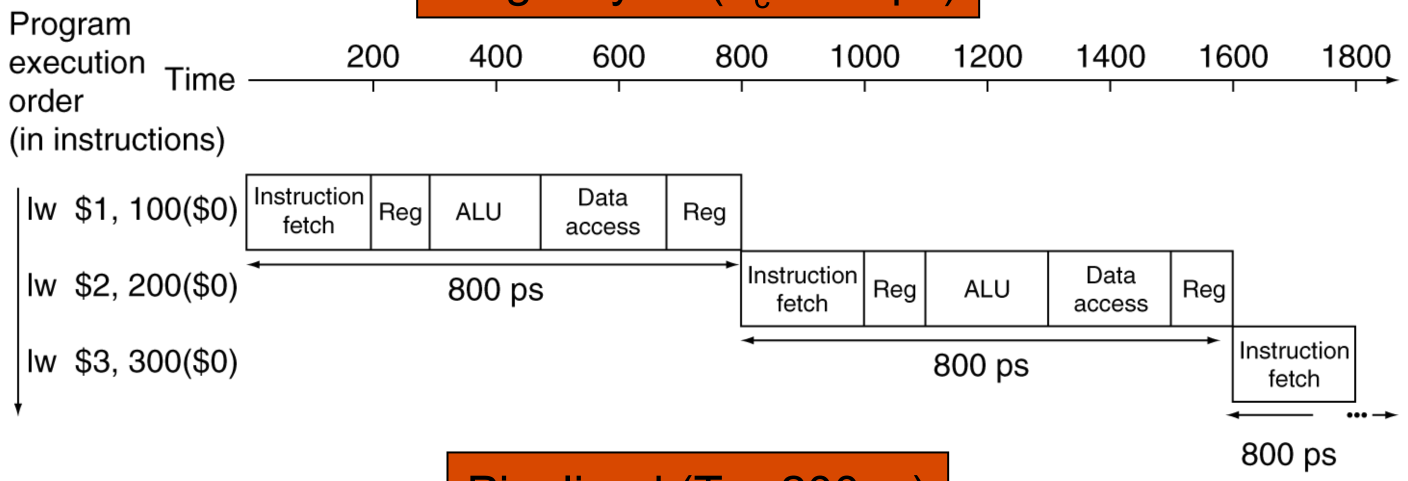
Review

Putting it All Together: Pipelined Processor

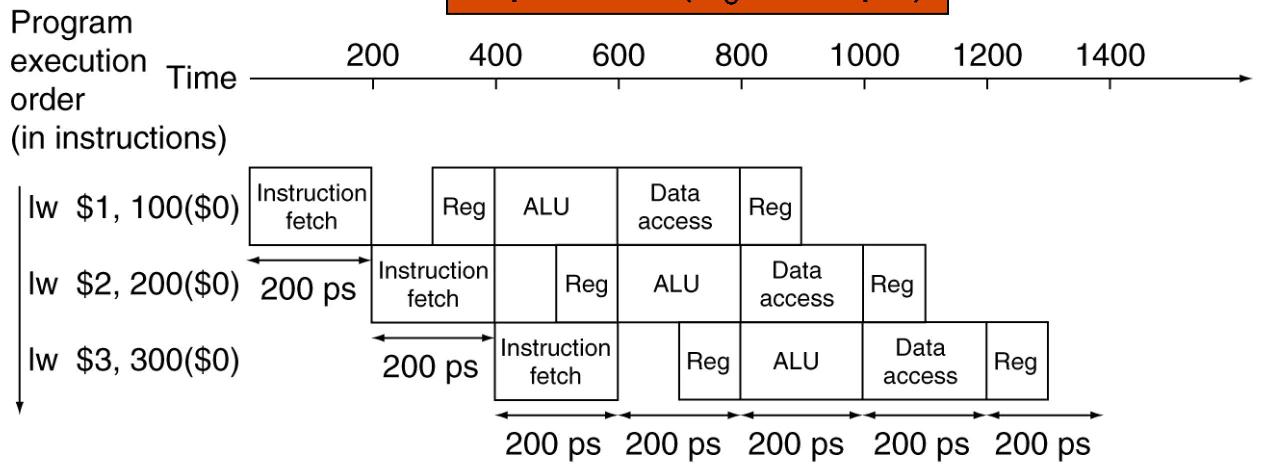


Pipeline Performance

Single-cycle (T_c= 800ps)



Pipelined (T_c= 200ps)



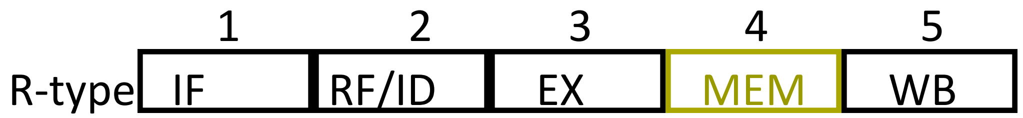
Pipeline Hazards

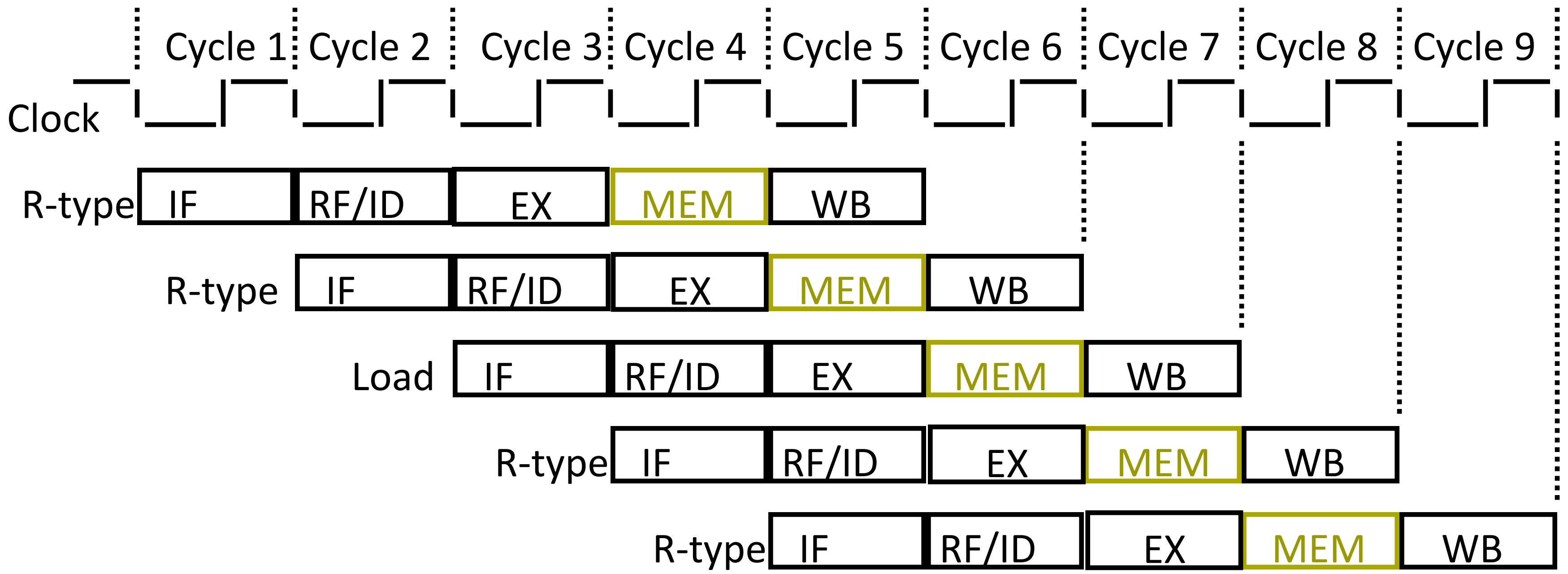
- Situations that prevent completing an instruction every cycle
  - Lead to CPI > 1

- Structure hazards
  - A required resource is busy
- Data hazard
  - Must wait previous instructions to produce/consume data
- Control hazard
  - Next PC depends on previous instruction

Delayed Write-back in 5-stage Pipeline

- Delay R-type register write by one cycle
  - Does this increase the CPI of instruction?
  - What is the cost?





Cycle 1
Cycle 2
Cycle 3
Cycle 4
Cycle 5
Cycle 6
Cycle 7
Cycle 8
Cycle 9
Clock
R-type IF RF/ID EX MEM WB
R-type IF RF/ID EX MEM WB
Load IF RF/ID EX MEM WB
R-type IF RF/ID EX MEM WB
R-type IF RF/ID EX MEM WB

Data Dependencies

- Dependencies for instruction *j* following instruction *i*
  - Read after Write (RAW or true dependence)
    - Instruction *j* tries to read before instruction *i* tries to write it
  - Write after Write (WAW or output dependence)
    - Instruction *j* tries to write an operand before *i* writes its value
  - Write after Read (WAR or (anti dependence)
    - Instruction *j* tries to write a destination before it is read by *i*
- Dependencies through registers or through memory

- Dependencies are a property of your program (always there)
- Dependencies may lead to hazards on a specific pipeline

Dependency Examples

- True dependency => RAW hazard

```
addu    $t0, $t1, $t2
subu    $t3, $t4, $t0
```

- Output dependency => WAW hazard

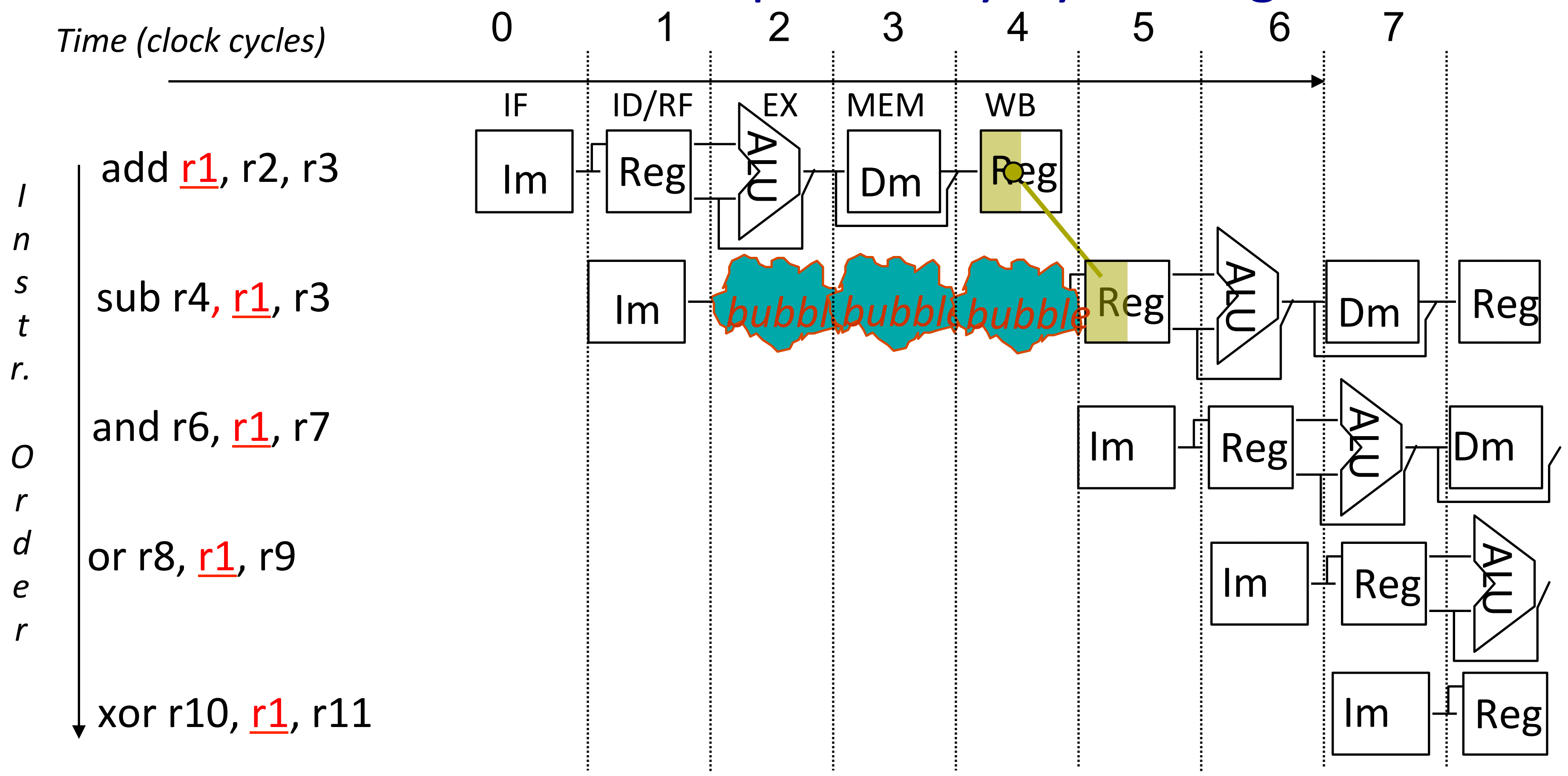
```
addu    $t0, $t1, $t2
subu    $t0, $t4, $t5
```

- Anti dependency => WAR hazard

```
addu    $t0, $t1, $t2
subu    $t1, $t4, $t5
```

Data Hazard - Stalls

- Eliminate reverse time dependency by stalling



How to Stall the Pipeline

- Discover need to stall when 2nd instruction is in ID stage
  - Repeat its ID stage until hazard resolved
  - Let all instructions ahead of it move forward
  - Stall all instructions behind it

1. Force control values in ID/EX register a NOP instruction
   - As if you fetched or x0, x0, x0
   - When it propagates to EX, MEM and WB, nothing will happen
2. Prevent update of PC and IF/ID register
   - Using instruction is decoded again
   - Following instruction is fetched again

Performance Effect

- Stalls can have a significant effect on performance
- Consider the following case
  - The ideal CPI of the machine is 1
  - A RAW hazard causes a 3 cycle stall
- If 40% of the instructions cause a stall?
  - The new effective CPI is $1 + 3 \times 0.4 = 2.2$
  - And the real % is probably higher than 40%
- You get less than ½ the desired performance!

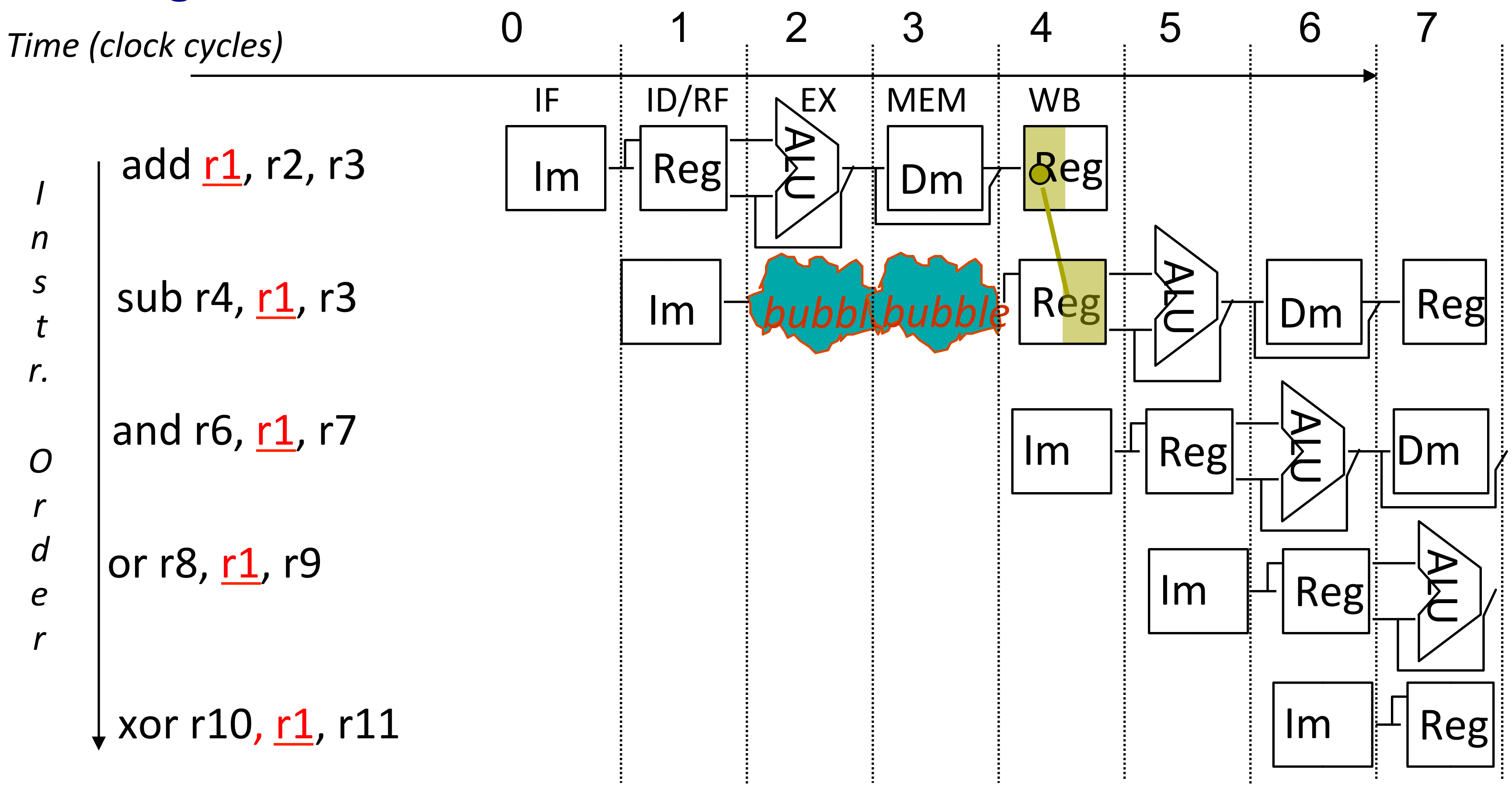
Reducing Stalls

- Key: when you say new data is actually available?

- In the 5-stage pipeline
  - After WB stage?
  - During WB stage?
    - Register file is typically fast
    - Write in the first half, read in the second half
  - After EX stage?

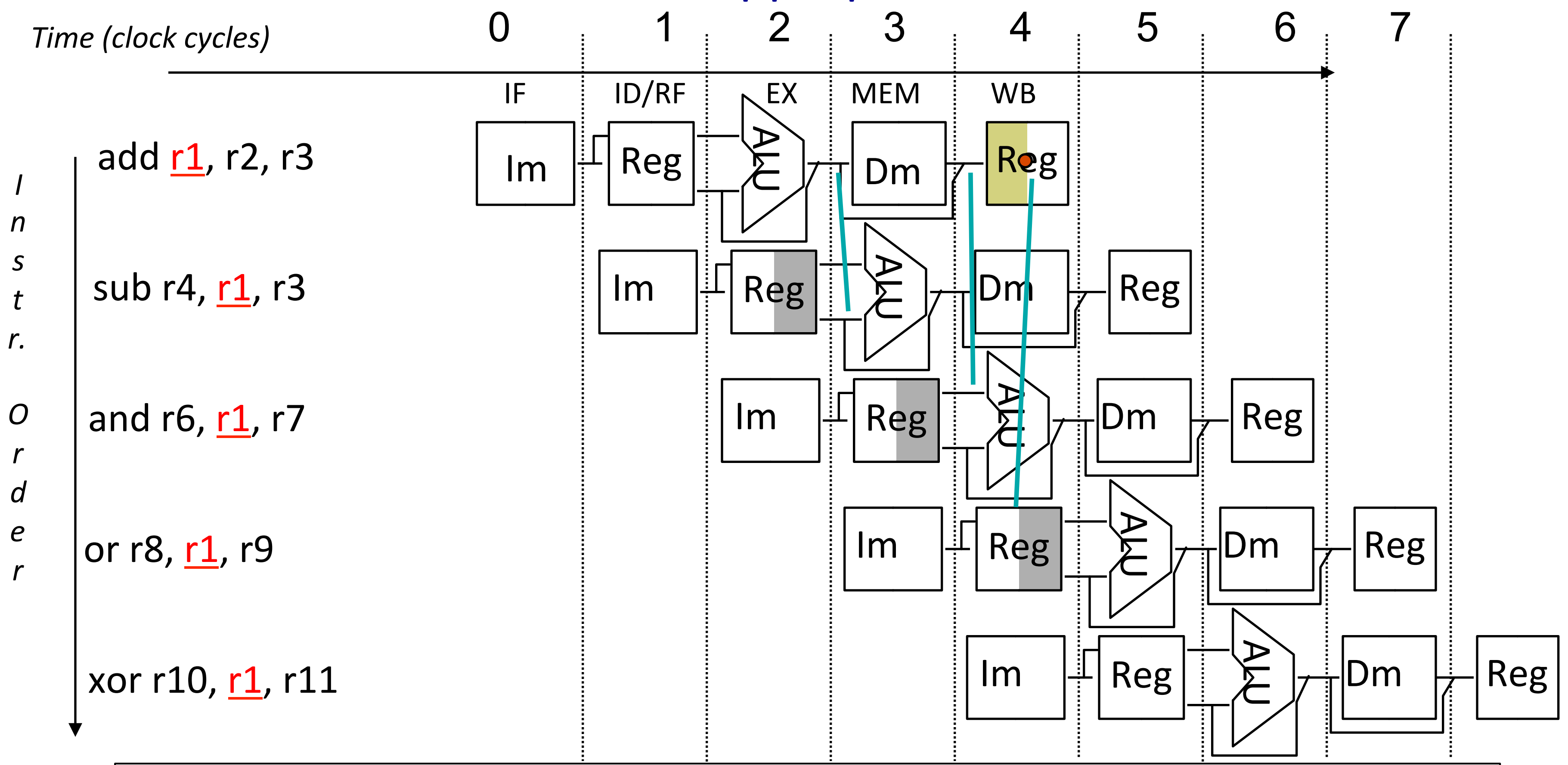
Decreasing Stalls: Fast RF

- Register file writes on first half and reads on second half



Decreasing Stalls: Forwarding

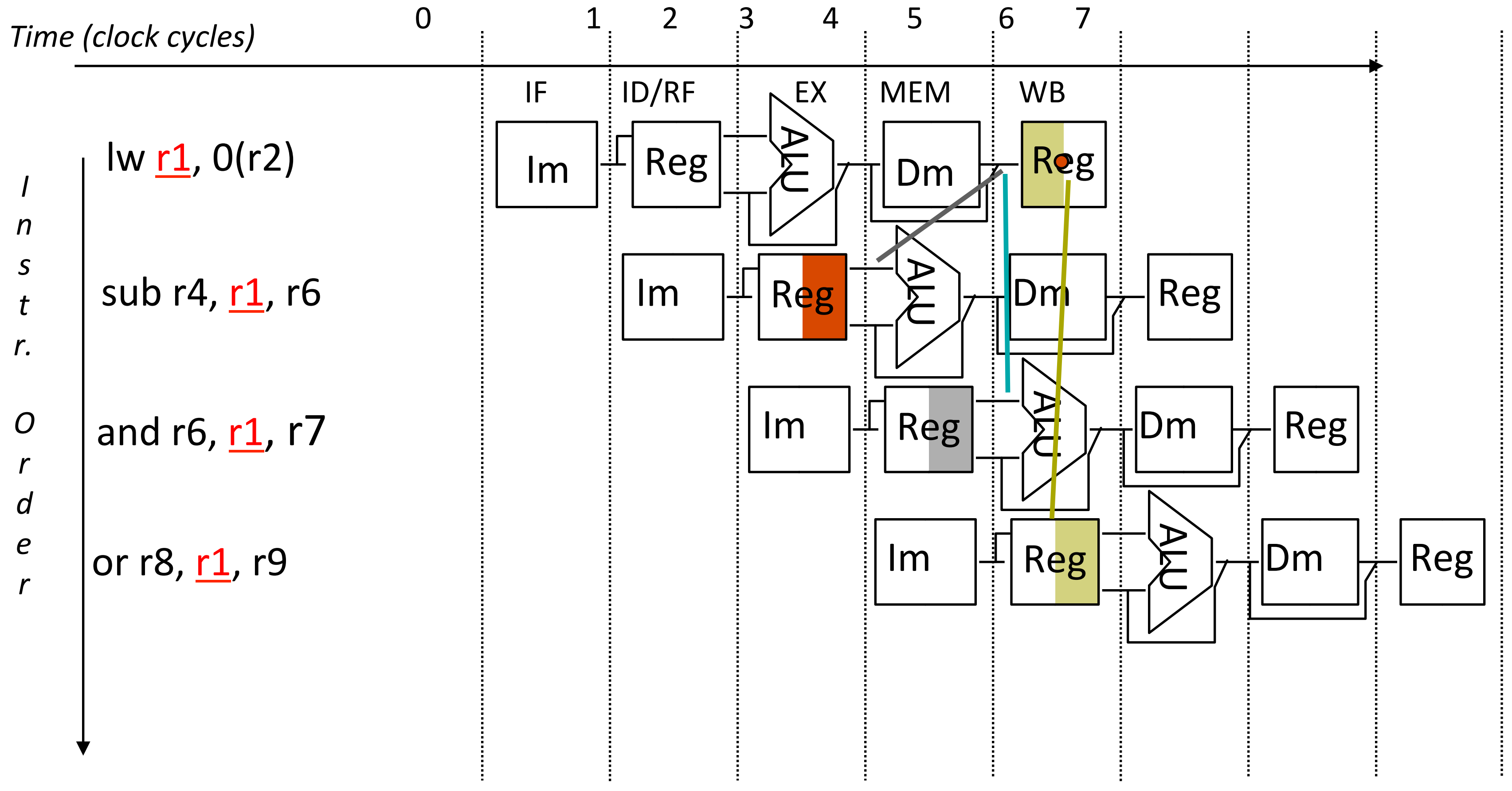
- "Forward" the data to the appropriate unit



Eliminates stalls for dependencies between ALU instrs.

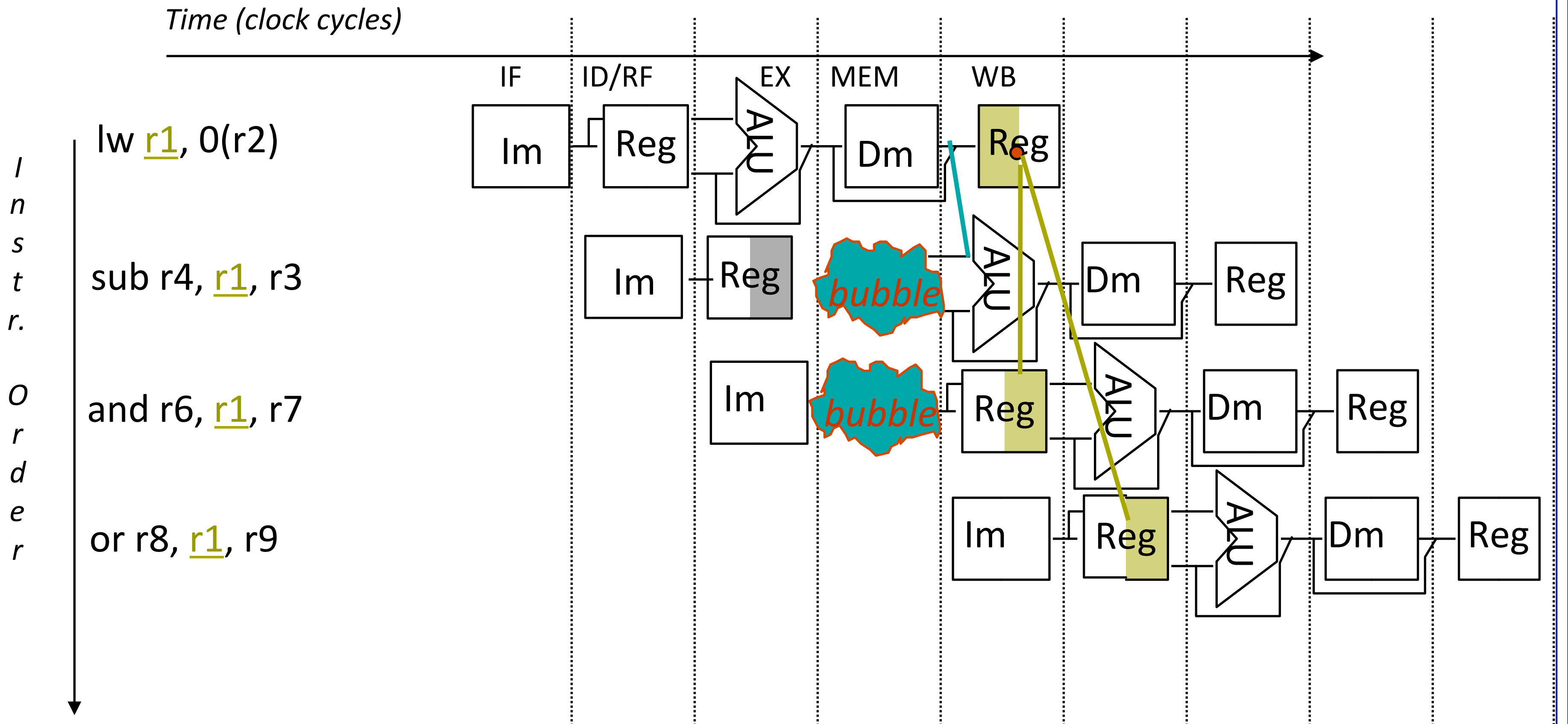
Forwarding Limitation: Load-Use Case

- Data is not available yet to be forwarded



Load-Use Case: Hardware Stall

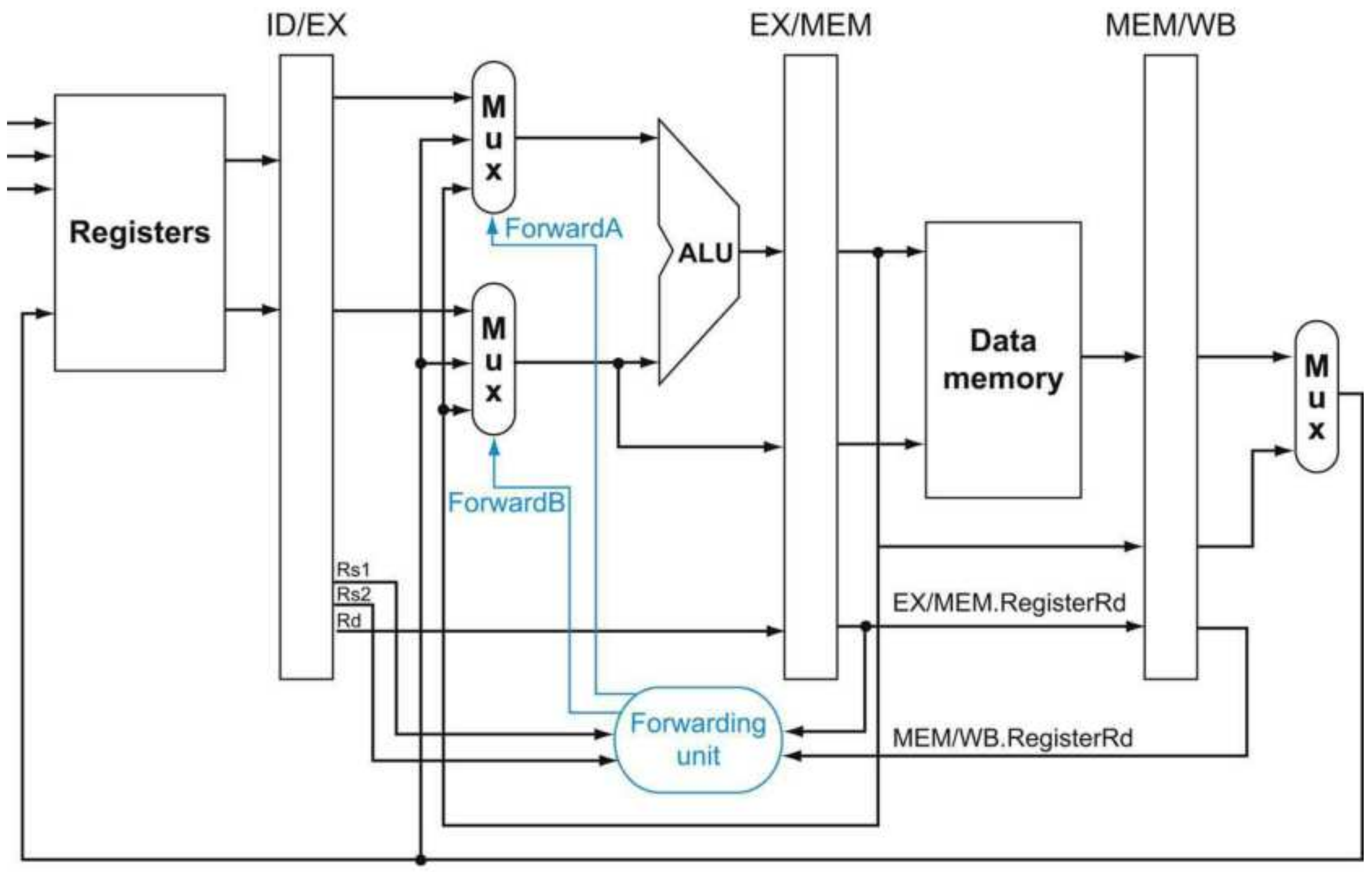
- A *pipeline interlock* checks and stops the *instruction issue*



Identifying the Forwarding Datapaths

- Identify all stages that <u>produce</u> new values
  - EX and MEM
- All stages after first producer are sources of forwarding data
  - MEM, WB
- Identify all stages that really <u>consume</u> values
  - EX and MEM
- These stages are the destinations of a forwarding data
- Add multiplexor for each pair of source/destination stages
  - Consider both possible instruction operands

Forwarding Paths: Partial



Forwarding Control

- Pass register numbers along pipeline
  - e.g., ID/EX.RegisterRs = register number for Rs in ID/EX pipeline register
- ALU operand register numbers in EX stage are given by
  - ID/EX.RegisterRs, ID/EX.RegisterRt
- Data hazards possible when
  - 1a. EX/MEM.RegisterRd == ID/EX.RegisterRs
  - 1b. EX/MEM.RegisterRd == ID/EX.RegisterRt
  - 2a. MEM/WB.RegisterRd == ID/EX.RegisterRs
  - 2b. MEM/WB.RegisterRd == ID/EX.RegisterRt

Fwd from EX/MEM pipeline reg

Fwd from MEM/WB pipeline reg

Forwarding Control

- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite, MEM/WB.RegWrite
- And if Rd for that instruction is not x0 (zero register)
  - EX/MEM.RegisterRd ≠ 0,
    MEM/WB.RegisterRd ≠ 0
- And if forwarding instruction is not a load in MEM stage
  - EX/MEM.MemToReg==0
  - This is a case we have to stall…

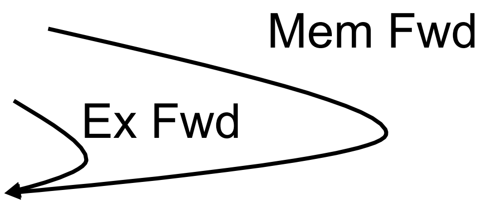
Forwarding Control (Stall Case not Shown)

- EX hazard
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
      and (EX/MEM.RegisterRd == ID/EX.RegisterRs))
    ForwardA = 10
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
      and (EX/MEM.RegisterRd == ID/EX.RegisterRt))
    ForwardB = 10
- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
      and (MEM/WB.RegisterRd == ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
      and (MEM/WB.RegisterRd == ID/EX.RegisterRt))
    ForwardB = 01

Double Data Hazard

- Consider the sequence:

```
add $1,$1,$2
sub $1,$1,$3
or  $1,$1,$4
```



- Both hazards occur
  - Want to use the most recent result from the sub
- Revise MEM hazard condition
  - Only fwd if EX hazard condition isn't true

Forwarding Control (Revised)

- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)

        and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)

                and (EX/MEM.RegisterRd == ID/EX.RegisterRs))

        and (MEM/WB.RegisterRd = ID/EX.RegisterRs))

    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)

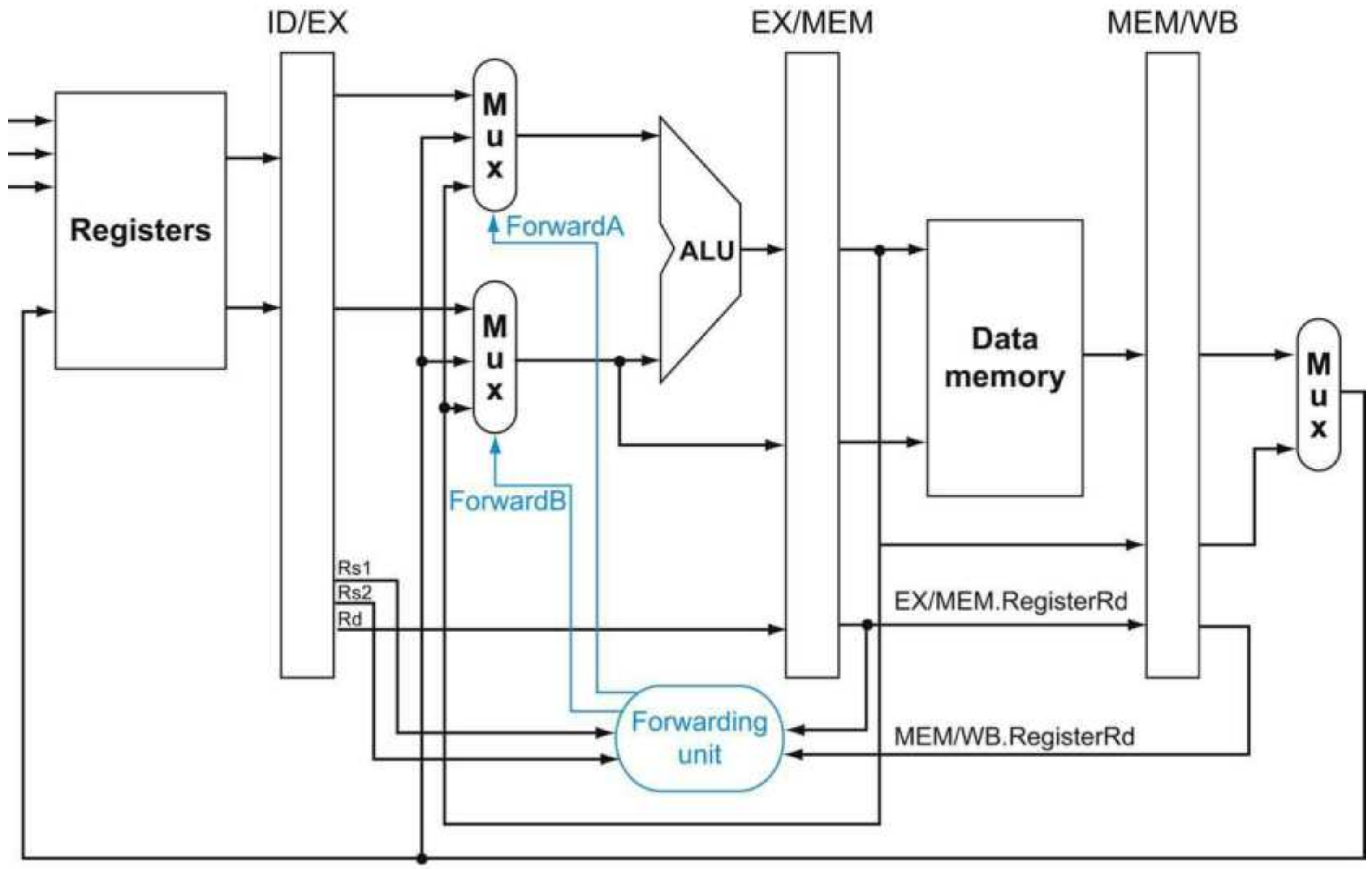
        and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)

                and (EX/MEM.RegisterRd == ID/EX.RegisterRt))

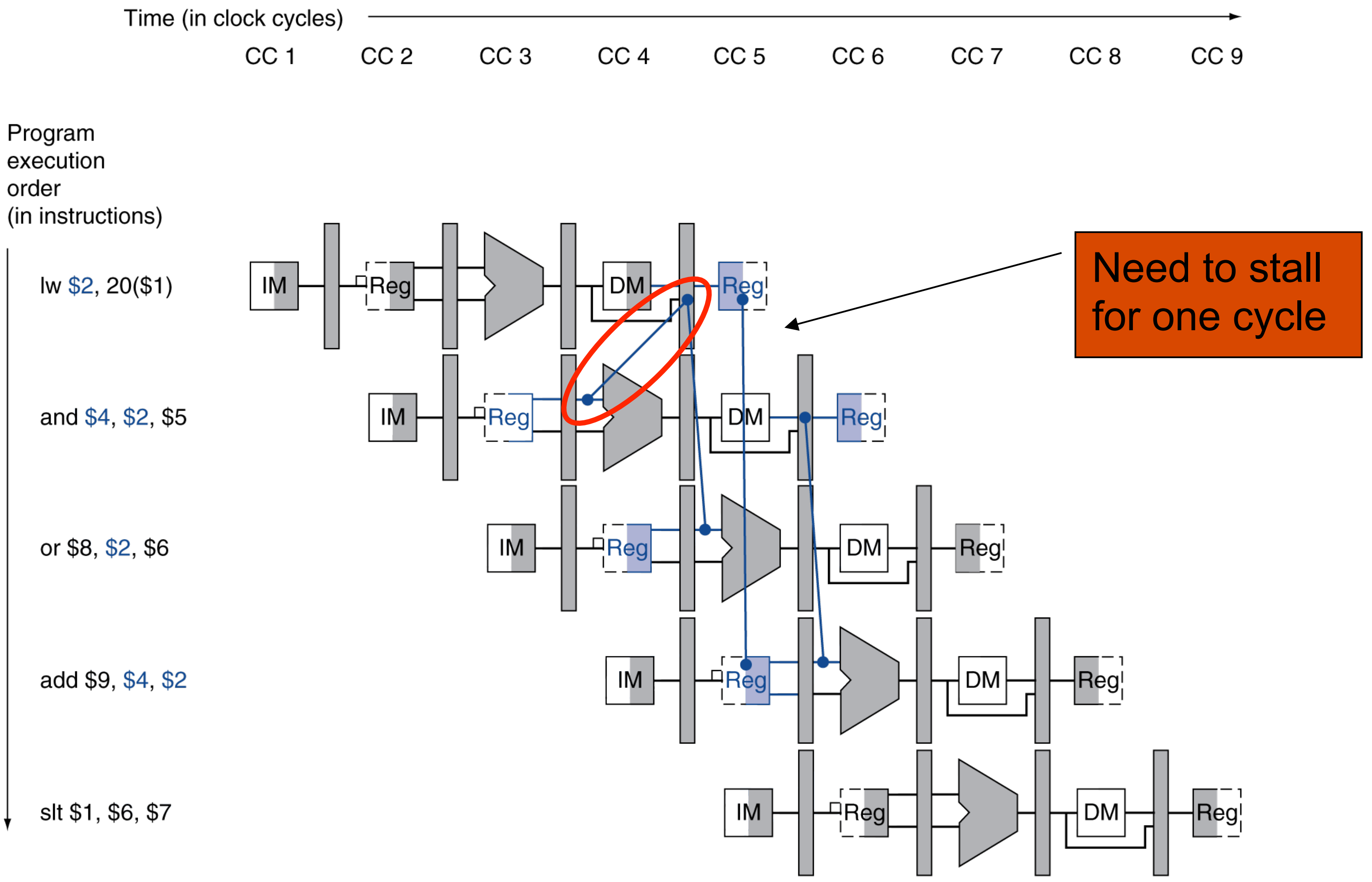
        and (MEM/WB.RegisterRd = ID/EX.RegisterRt))

    ForwardB = 01

Datapath with Forwarding



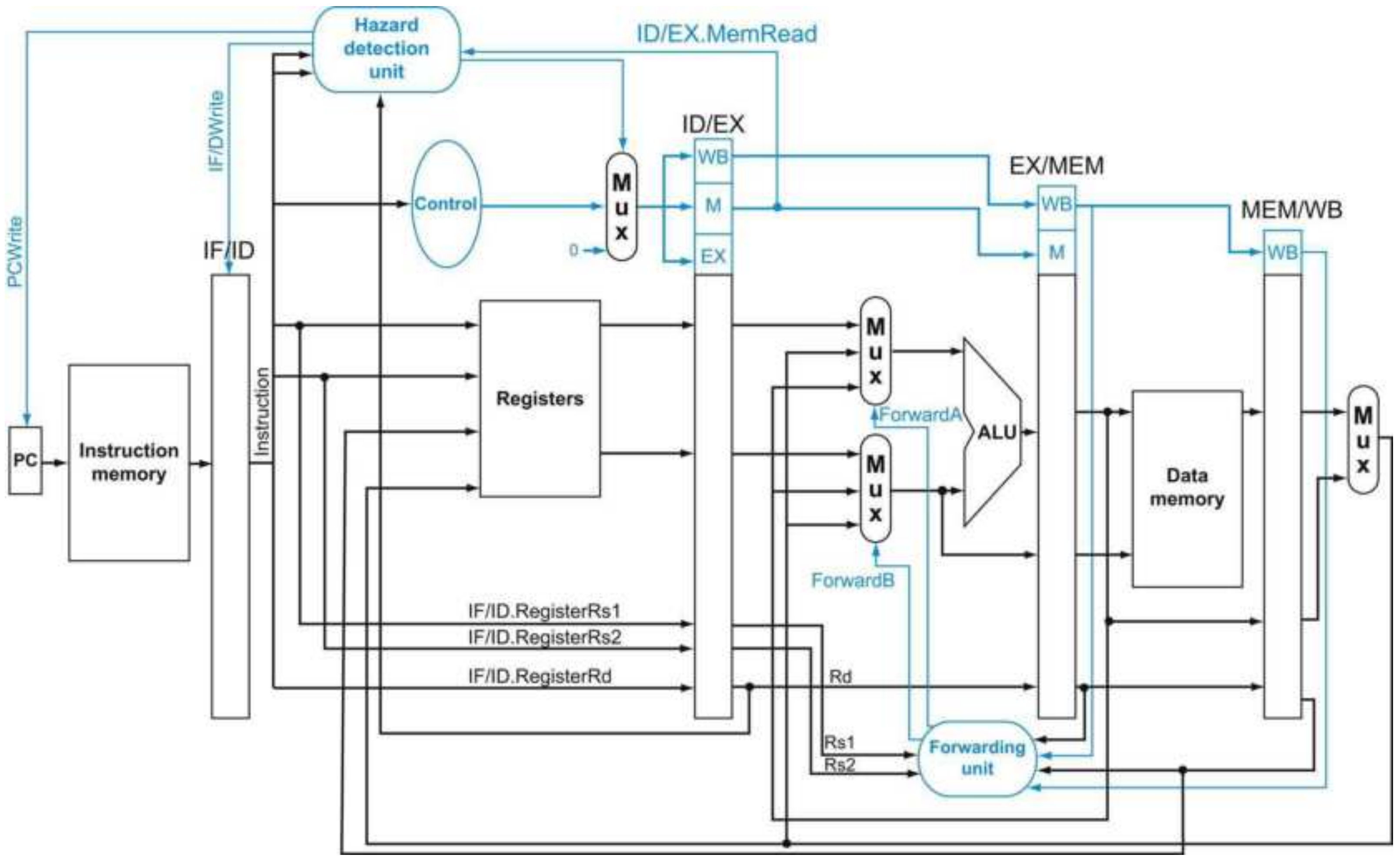
Load-Use Data Hazard



Load-Use Hazard Detection

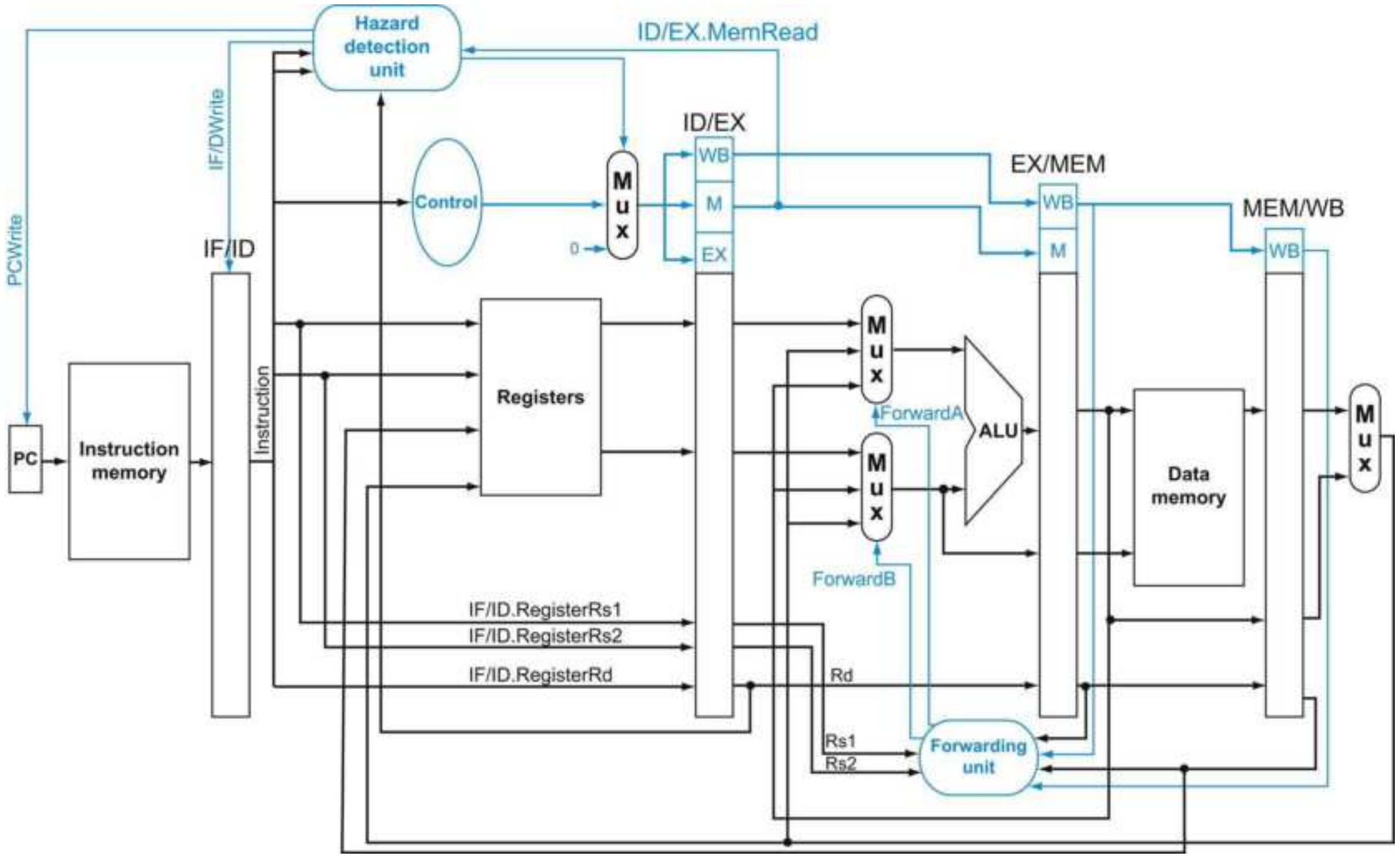
- Check when use instruction is decoded in ID stage
- ALU register numbers in ID stage are given by
  - IF/ID.RegisterRs, IF/ID.RegisterRt
- Load-use hazard when
  - ID/EX.MemRead and
    ((ID/EX.RegisterRd = IF/ID.RegisterRs) or
    (ID/EX.RegisterRd = IF/ID.RegisterRt))
- If detected, stall and insert bubble

Datapath with Hazard Detection



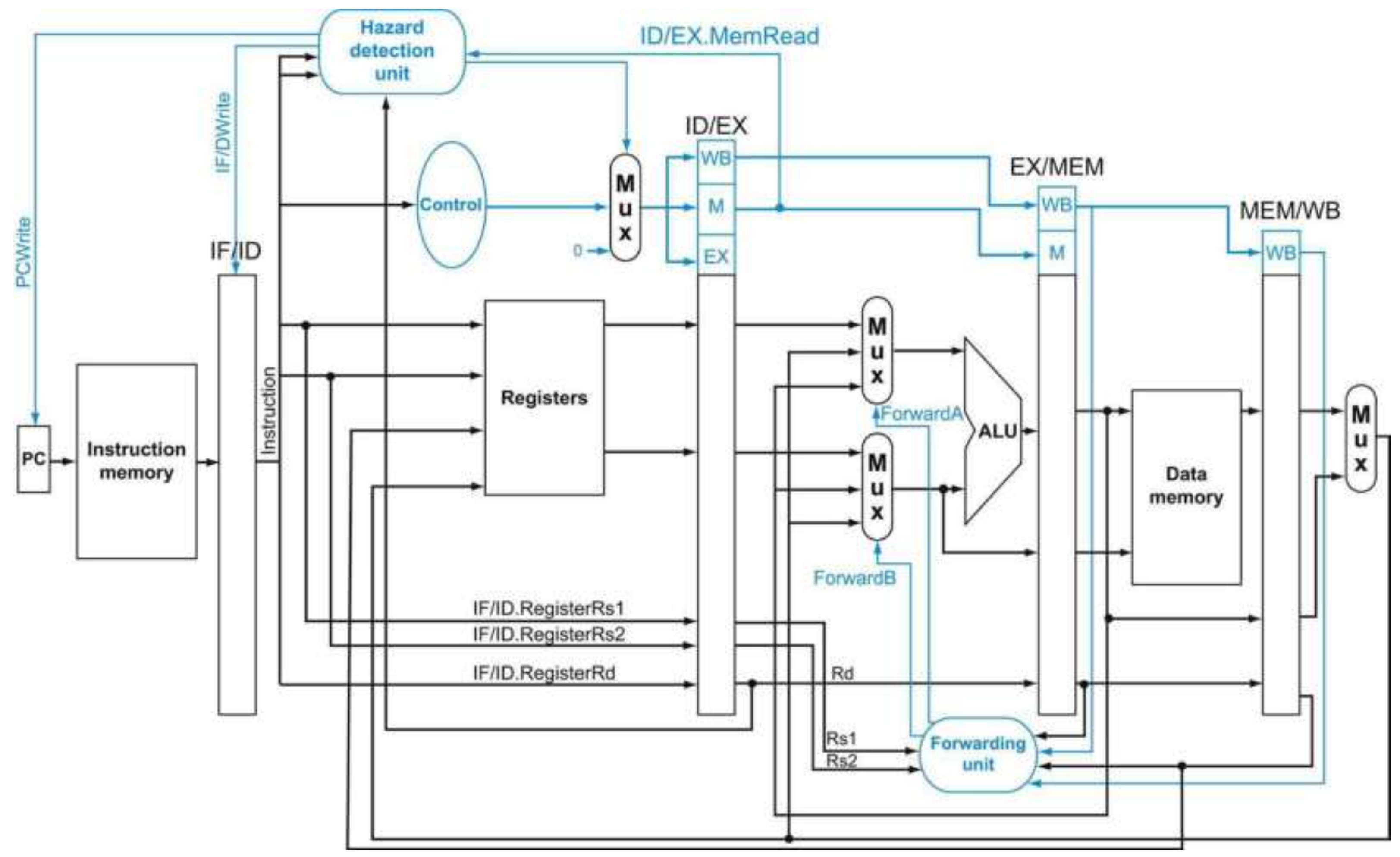
Example: Load-Use Stall

`sub r4, r1, r3   lw r1, 0(r2)`



Example: Load-Use Stall
1 cycle later

sub r4, r1, r3 nop lw r1, 0(r2)

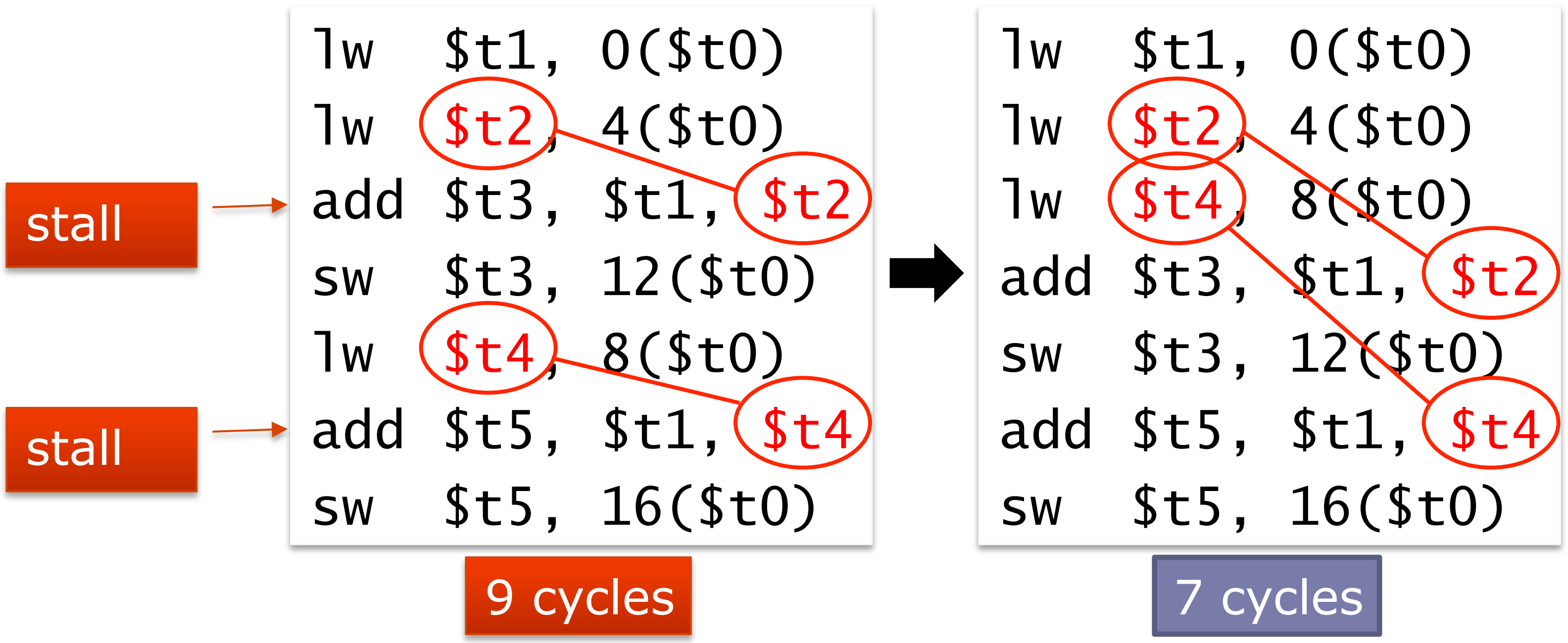


Compiler Optimizations & Data Hazards

- Compilers rearrange code to try to avoid load-use stalls
  - Also known as: filling the load delay slot
- Idea: find an independent instruction to space apart load & use
  - When successful, no stall is introduced dynamically
- Independent instruction from before or after the load-use
  - No data dependency to the load-use instructions
  - No control dependency to the load-use instructions
  - No exception issues
- When can't fill the slot
  - Leave it empty, hardware will introduce the stall

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;`



```
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)
```

```
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)
```

Data Hazards: Last Thoughts

- Different pipelines have different forwarding/stall cases
- Carefully examine each pipeline
  - Use the methodology above to identify forwarding cases
    - Search for stages where data is produced/consumed
  - Address remaining cases with stalls

- How about dependencies through memory?
  - WAR, WAW, RAW through loads and stores
  - Can they cause problems in our 5-stage pipeline?
  - Can they cause problems in general?

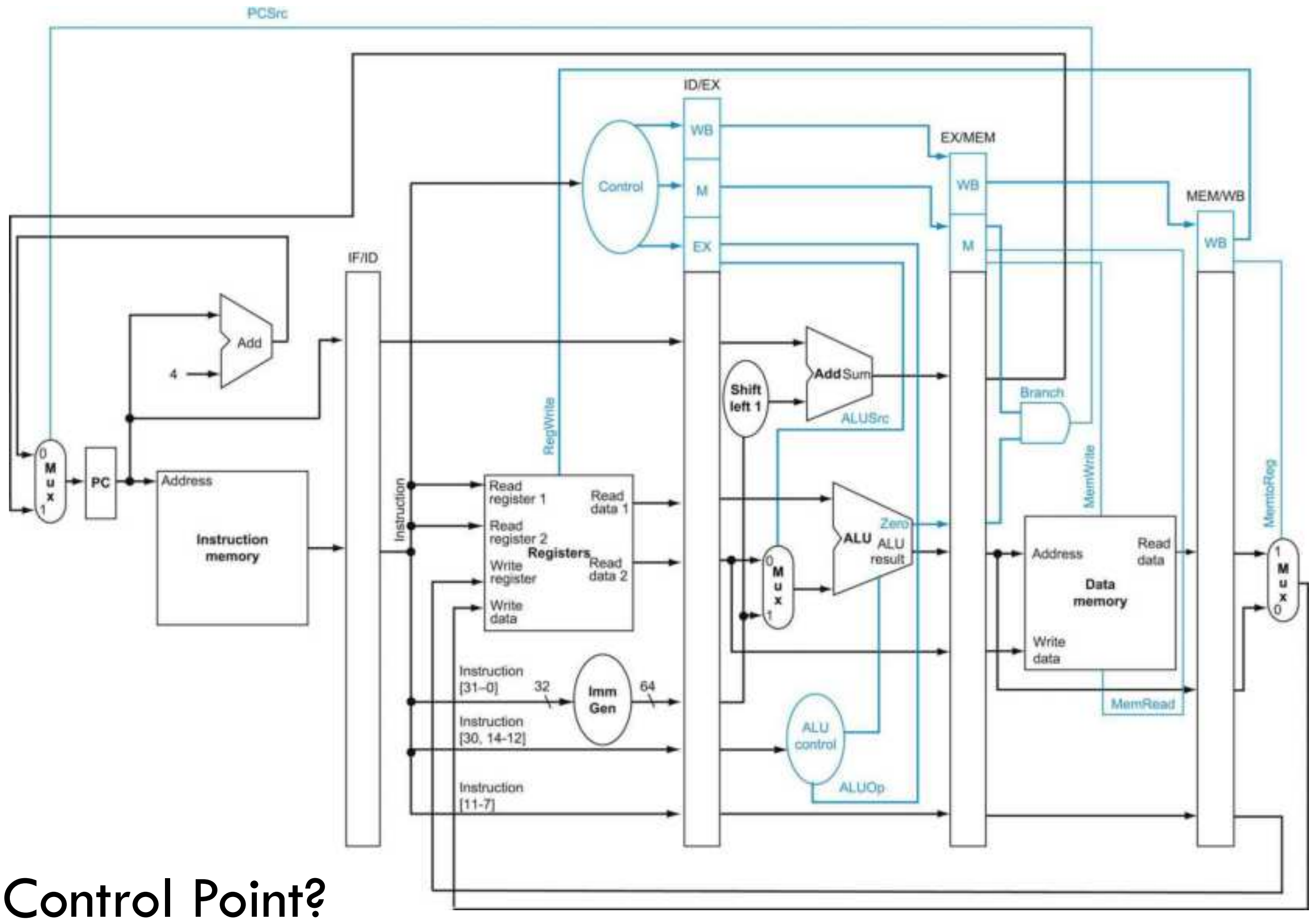
Control Hazards

Control Hazards

- Just like a data hazard on the program counter (PC)
  - Cannot fetch the next instruction if I don’t know its address (PC)
  - Only worse: we use the PC at the beginning of the pipeline
    - Makes forwarding very difficult

- Calculating the next PC for RISC-V:
  - Most instructions: PC+4
    - Can be calculated in IF stage and forward to next instructions
  - Jump
    - Must calculate new PC based on offset
    - Must fetch and decode jump
  - Branches
    - Must first compare registers and calculate PC+4+offset
    - Must fetch, decode, and execute branch

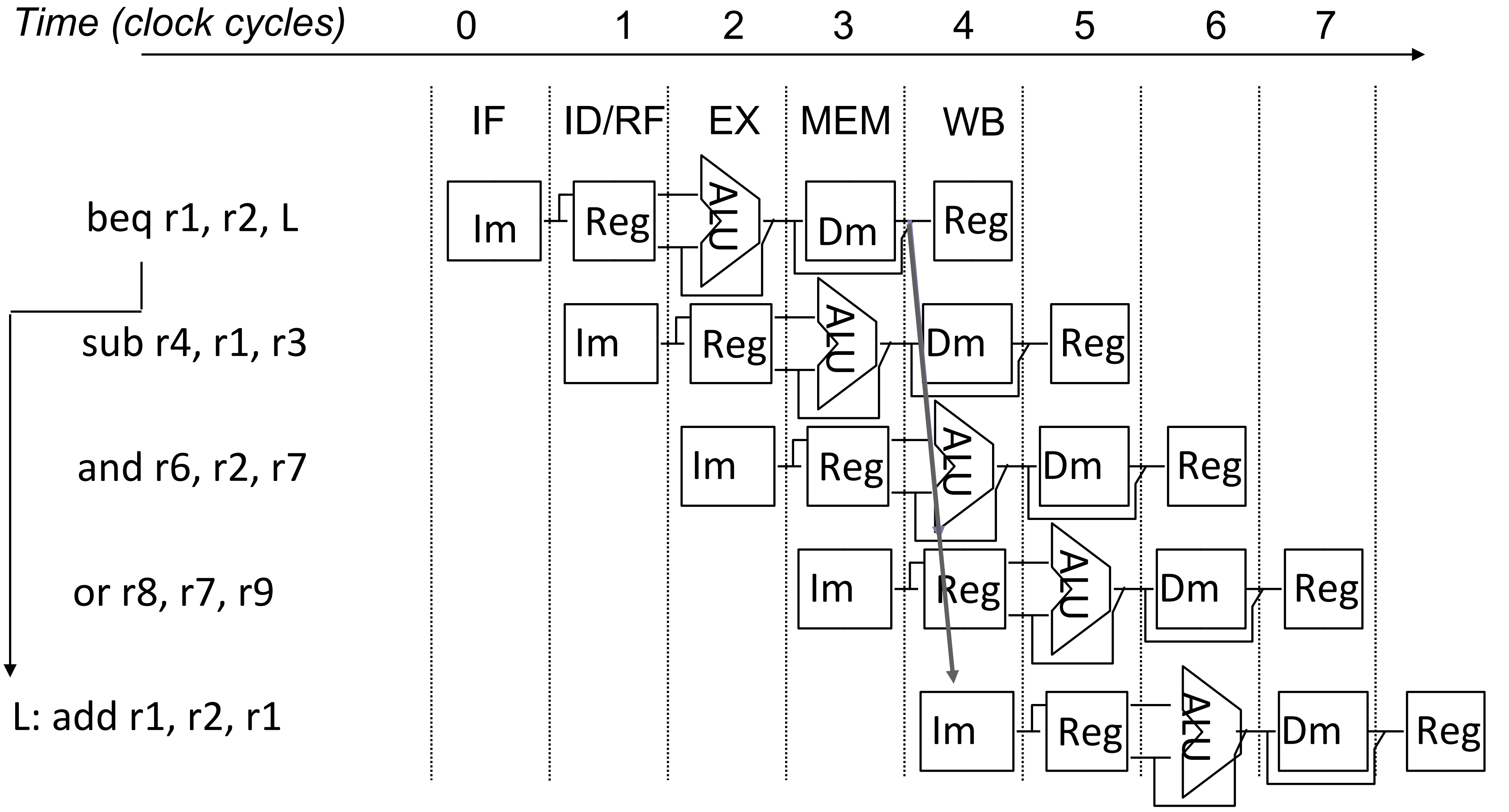
Our Pipeline So Far
(without forwarding and hazard detection)



Branch Control Point?

When do we know branch direction and target

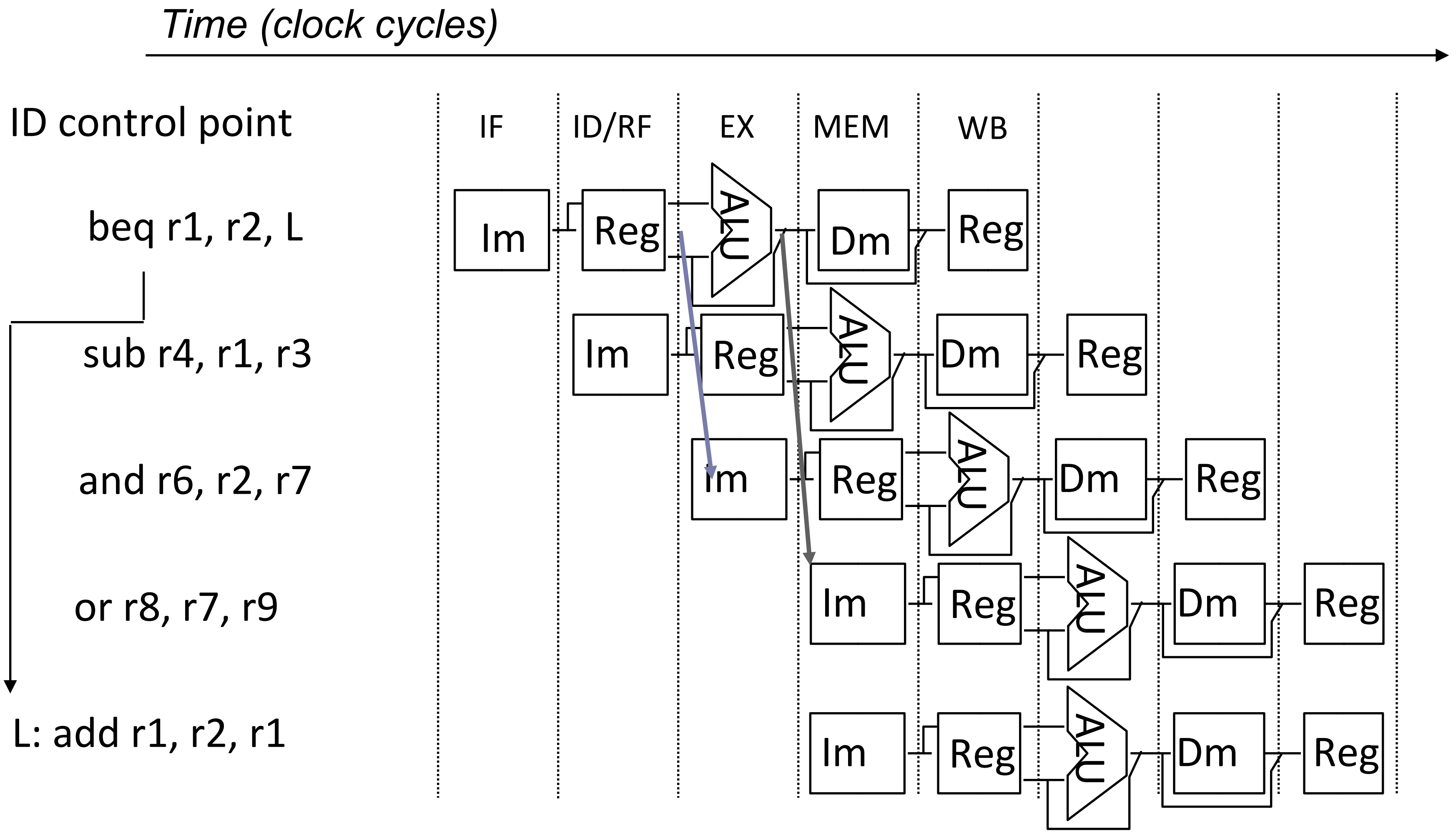
Branch Control Hazard



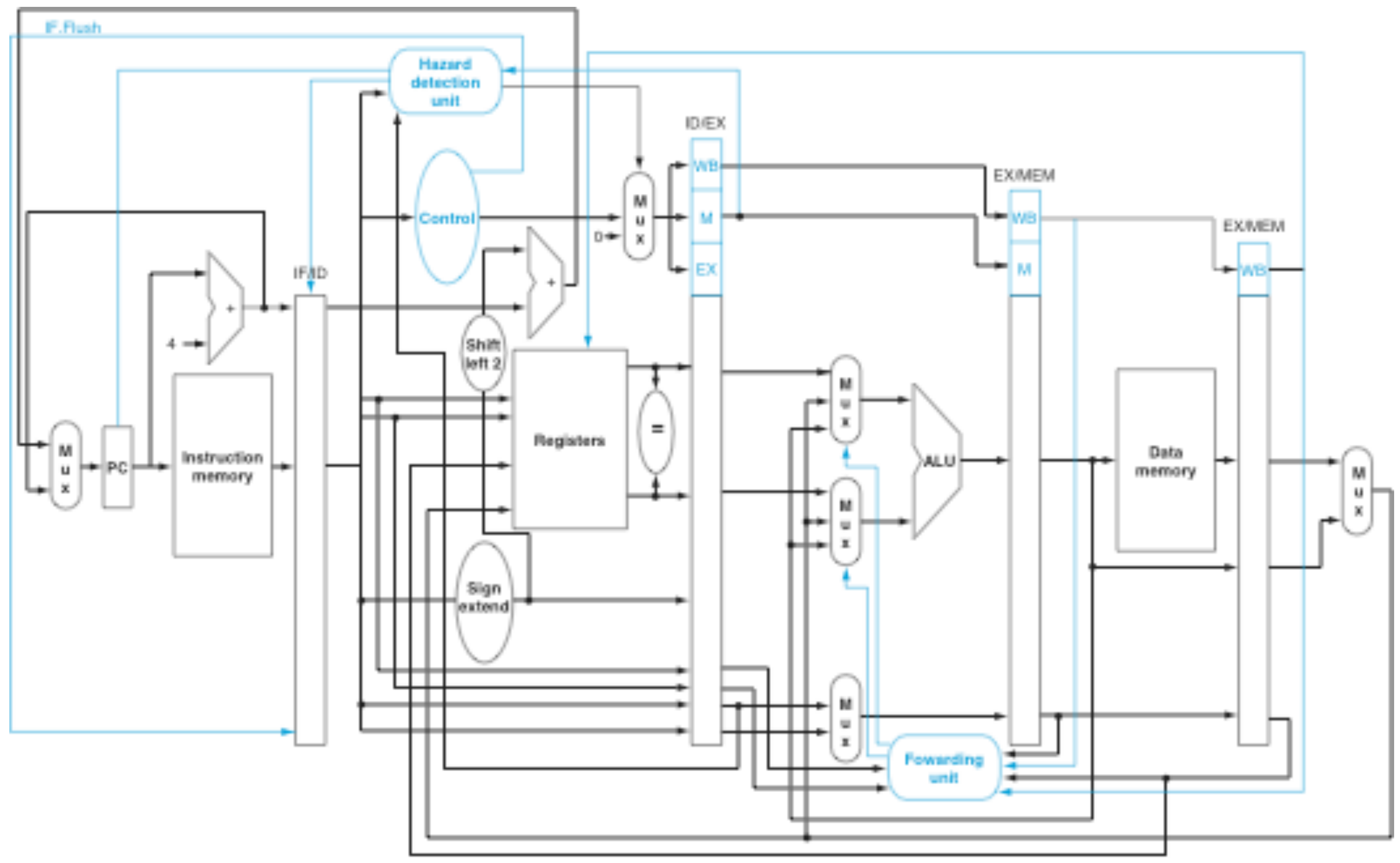
Control Hazard Solutions

- Stall on branches
  - Wait until you know the answer (branch CPI becomes 4)
  - Check for stall in ID stage
- Predict not-taken
  - Assume branch is not taken and continue with PC+4
  - If branch is actually not taken, all is good
  - Otherwise, nullify misfetched instructions and restart from PC+4+offset
- Predict taken
  - Assume branch is taken
  - More complex, cause you also need to predict PC+4+offset or delay until ID complete
  - Still need to nullify instructions if assumption is wrong

Optimizing Control Hazards: Execute Branch Earlier?



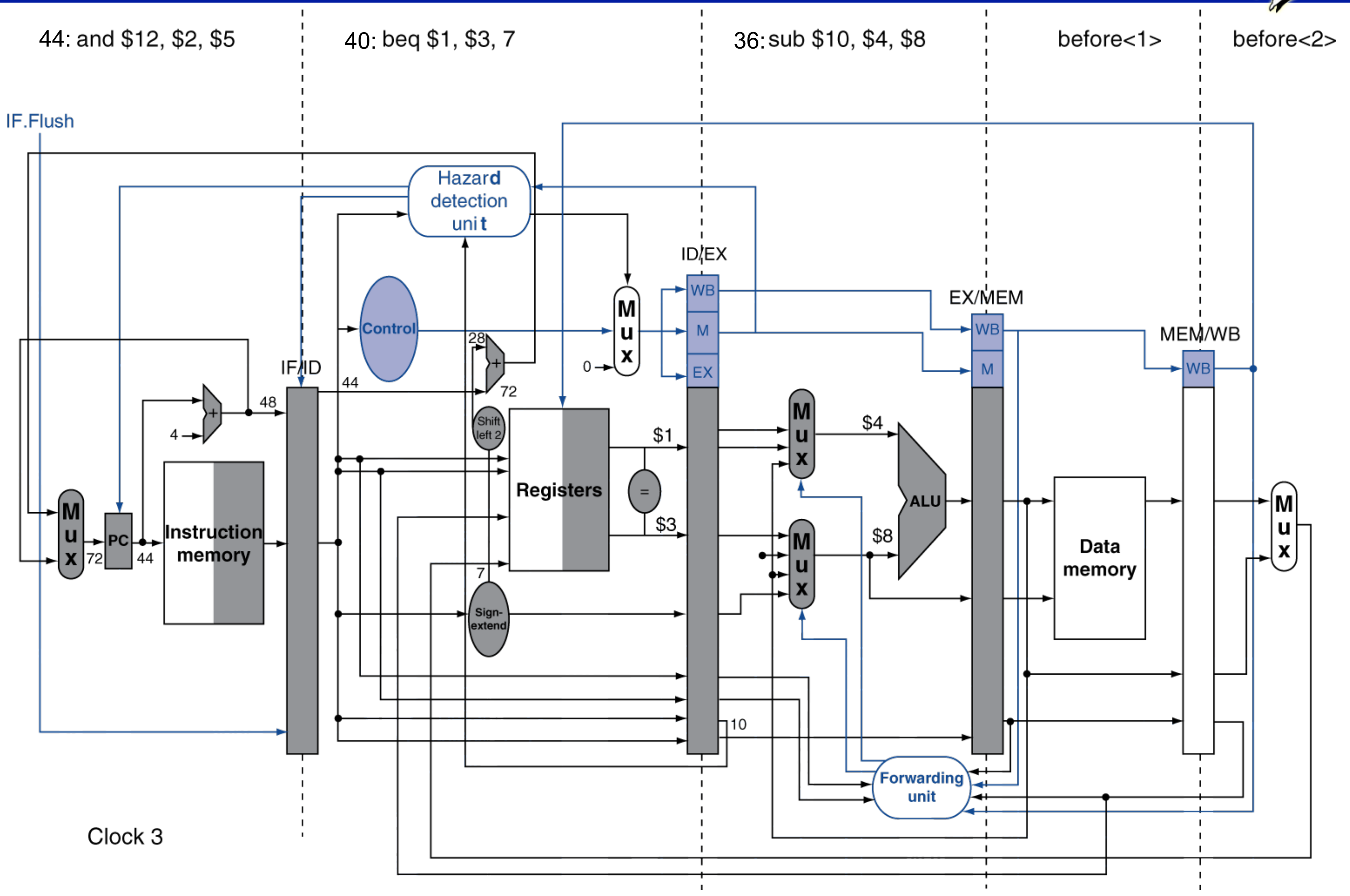
Reducing Branch Delay



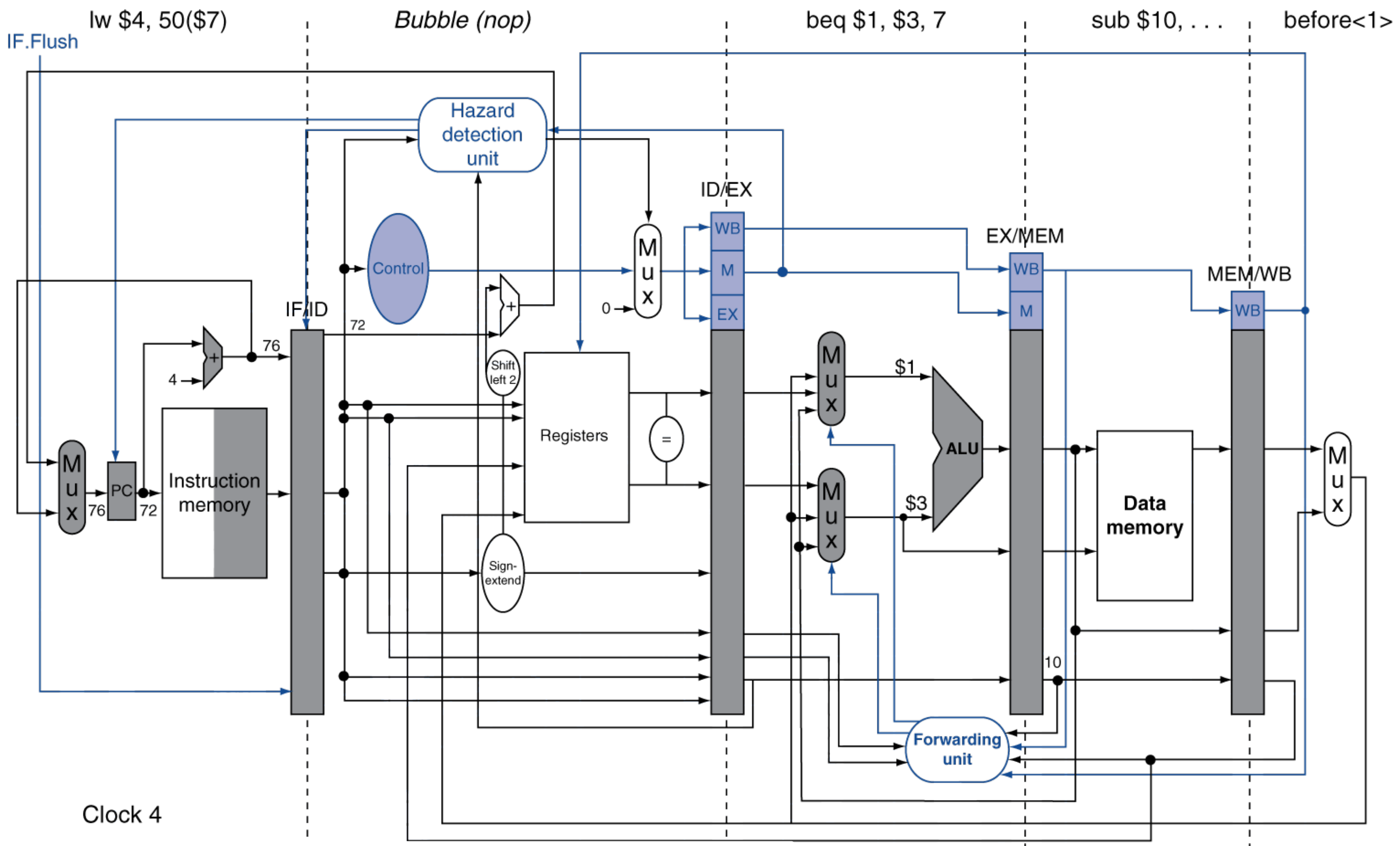
ID control point

Reason for simple branches in RISC-V

Example: Branch Taken



Example: Branch Taken



nop inserted using IF.Flush

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
- What does this mean for forwarding/stalls?
  - Need to check!!

Data Hazards for Branches

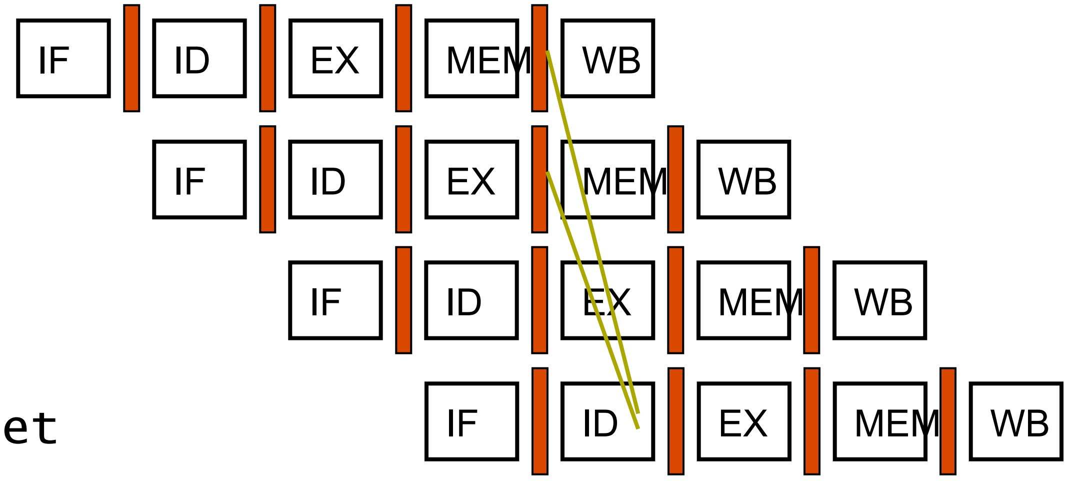
- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction

```
add $1, $2, $3

add $4, $5, $6

…

beq $1, $4, target
```

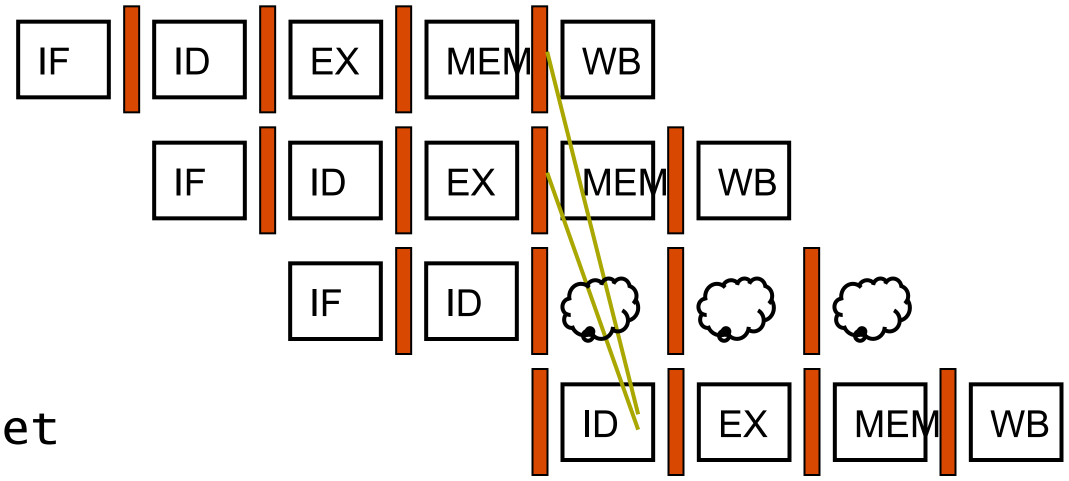


- Can resolve using forwarding
  - Additional datapaths and control

Data Hazards for Branches

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

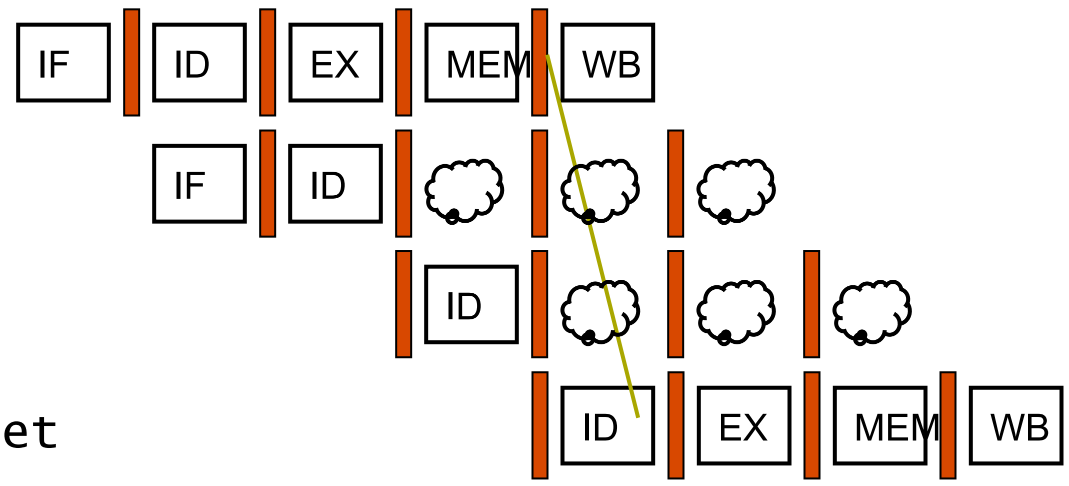
```
lw  $1, addr
add $4, $5, $6
beq stalled
beq $1, $4, target
```



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