CMPE110 Lecture 09
Pipelining

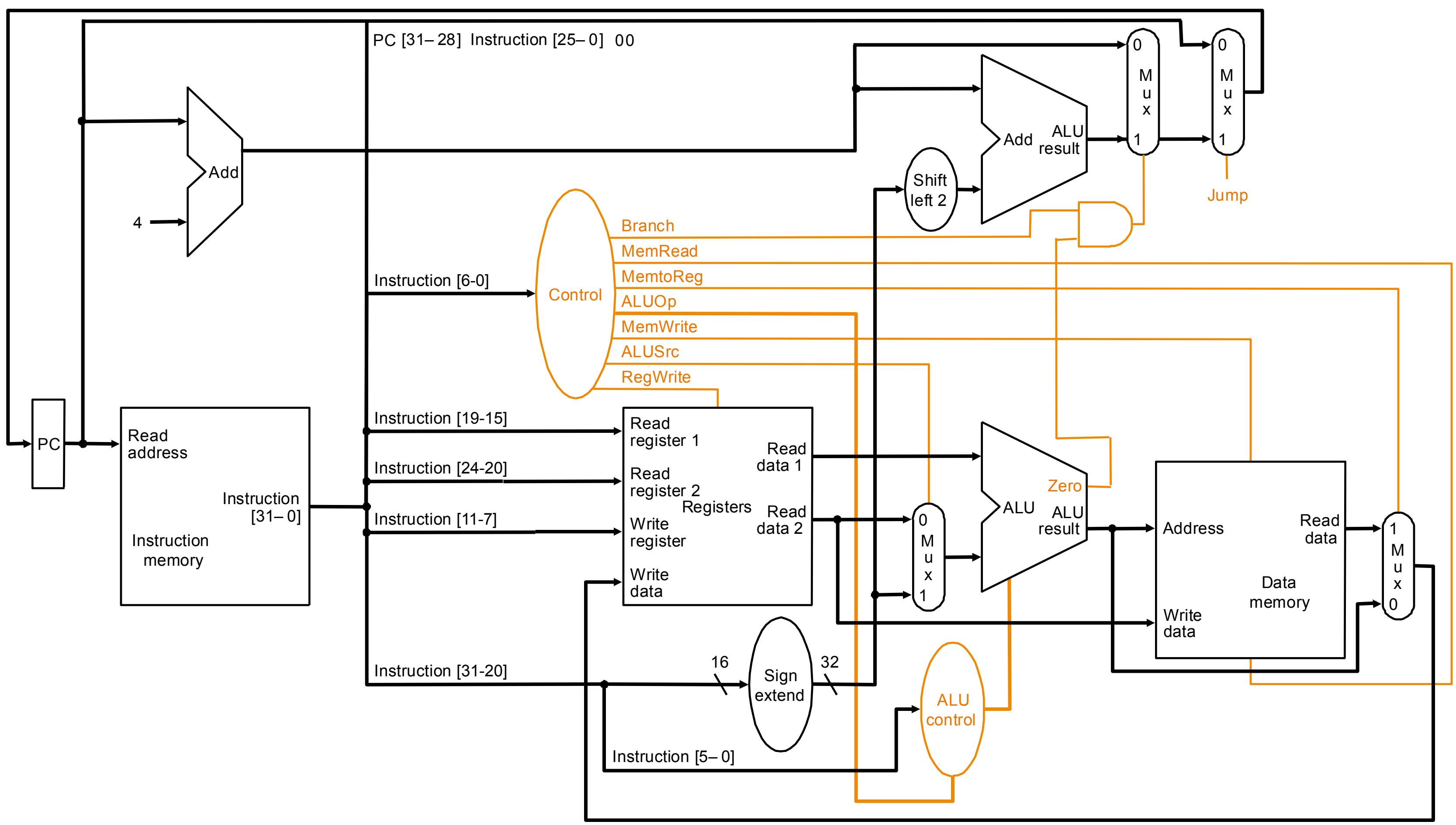
Heiner Litz

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

Announcements

Review

Putting It All Together: Our First Processor



Single Cycle Processor Performance

- Functional unit delay
  - Memory: 200ps
  - ALU and adders: 200ps
  - Register file: 100 ps

Instruction Class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
R-type	200	100	200		100	600
load	200	100	200	200	100	800
store	200	100	200	200		700
branch	200	100	200			500
jump	200					200

- CPU clock cycle = 800 ps = 0.8ns (1.25GHz)

Single Cycle RISC-V Processor

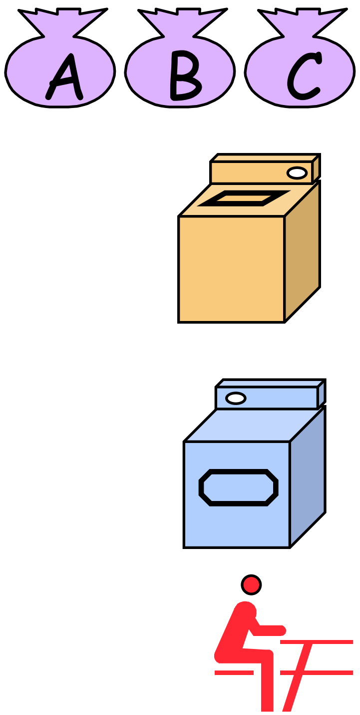
- Pros
  - Single cycle per instruction makes logic simple
- Cons
  - Cycle time is the worst case path → long cycle times
    - Worst case = load
  - Hardware is underutilized
    - ALU and memory used only for a fraction of clock cycle
    - Not well amortized!
  - Best possible CPI is 1

Key Tools for System Architects

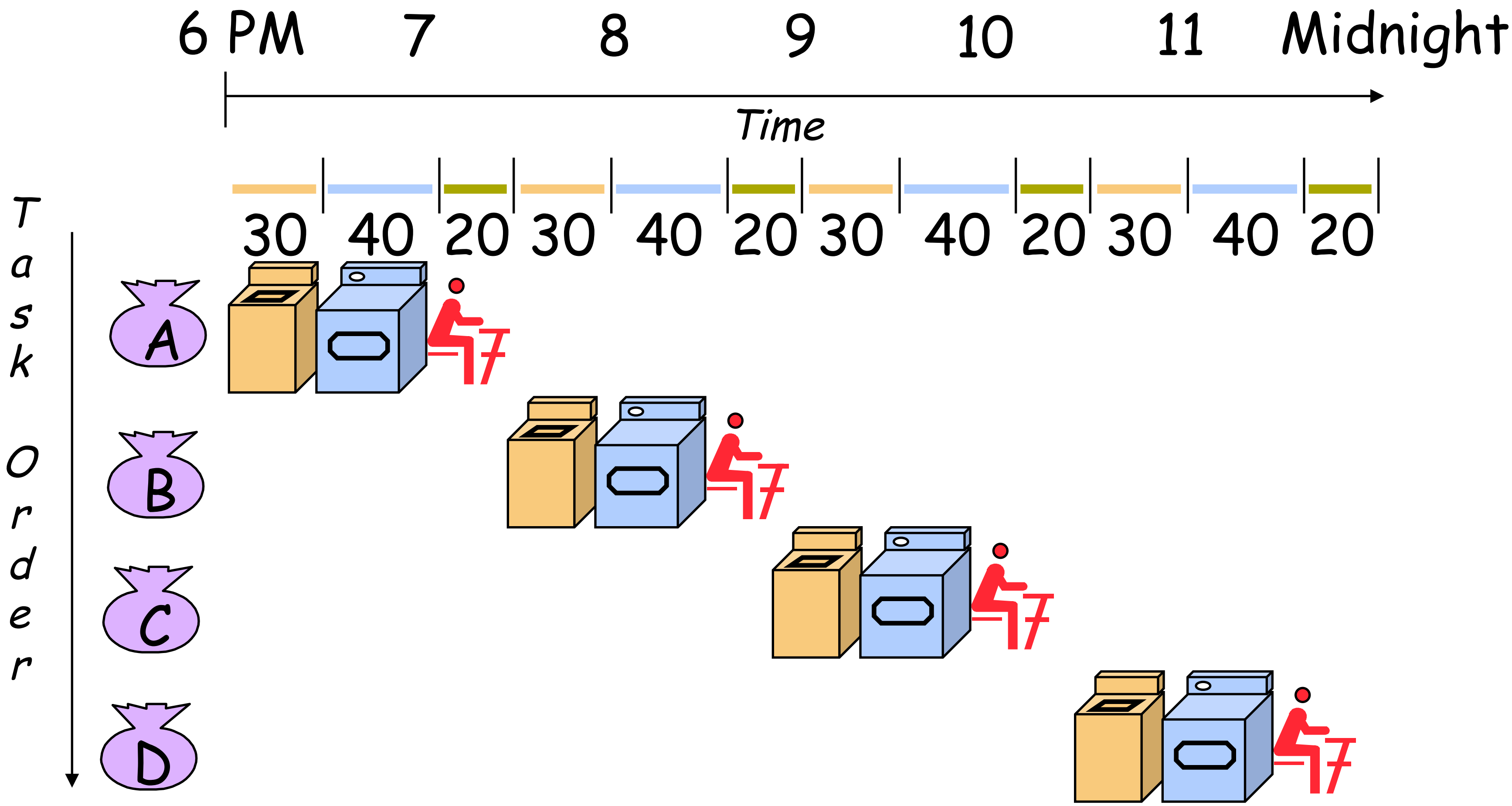
1. **Pipelining**
2. Parallelism
3. Out-of-order execution
4. Prediction
5. Caching
6. Indirection
7. **Amortization**
8. Redundancy
9. Specialization
10. Focus on the common case

Pipelining: The Laundry Analogy

- Ann, Brian, Cathy, Dave doing laundry
- Washer takes 30 minutes
- Dryer takes 40 minutes
- “Folding bench” takes 20 minutes

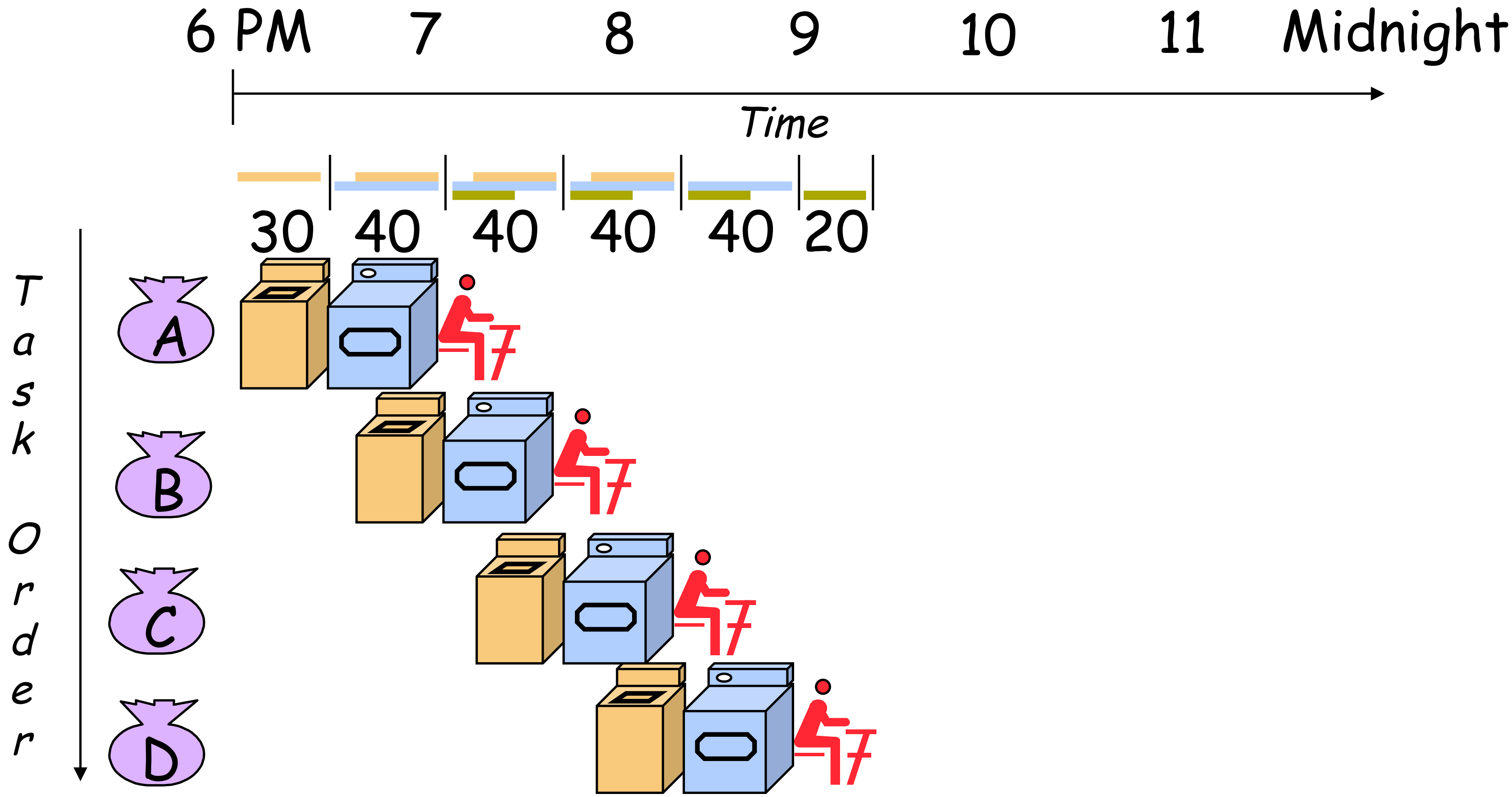


Single-cycle Laundry



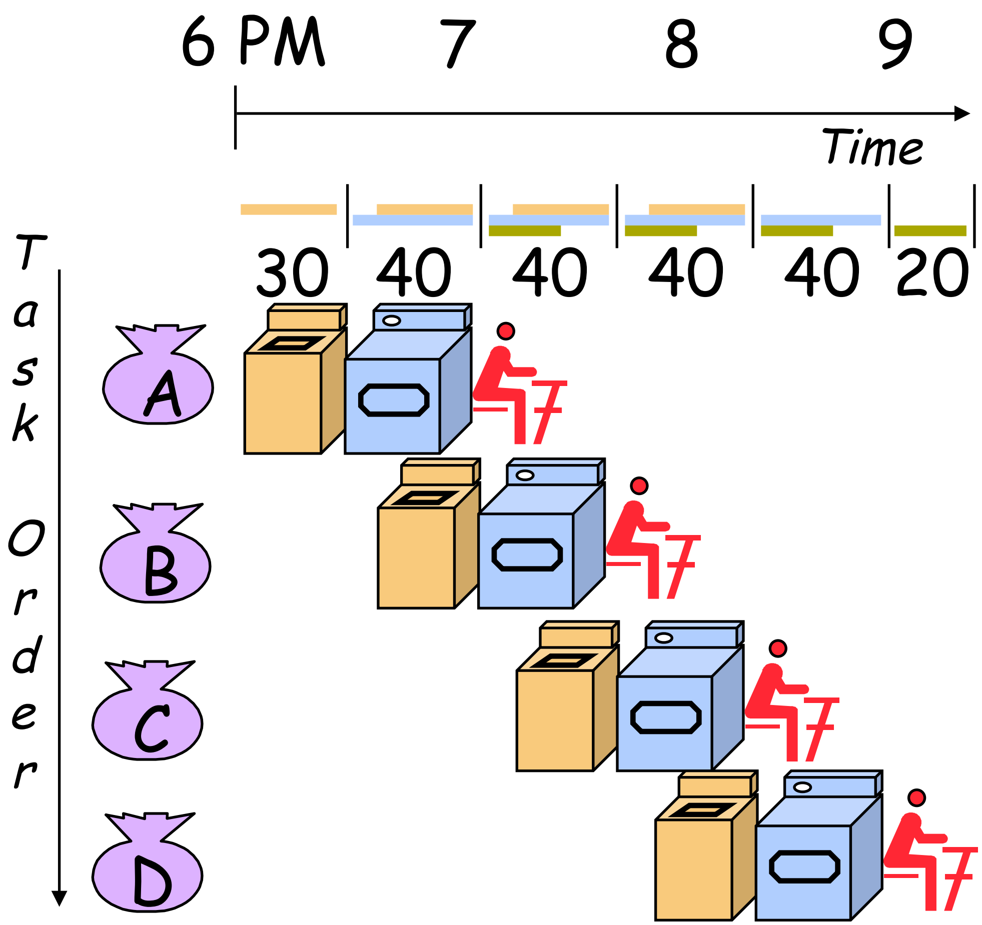
Single-cycle laundry takes 6 hours for 4 loads

Pipelined Laundry



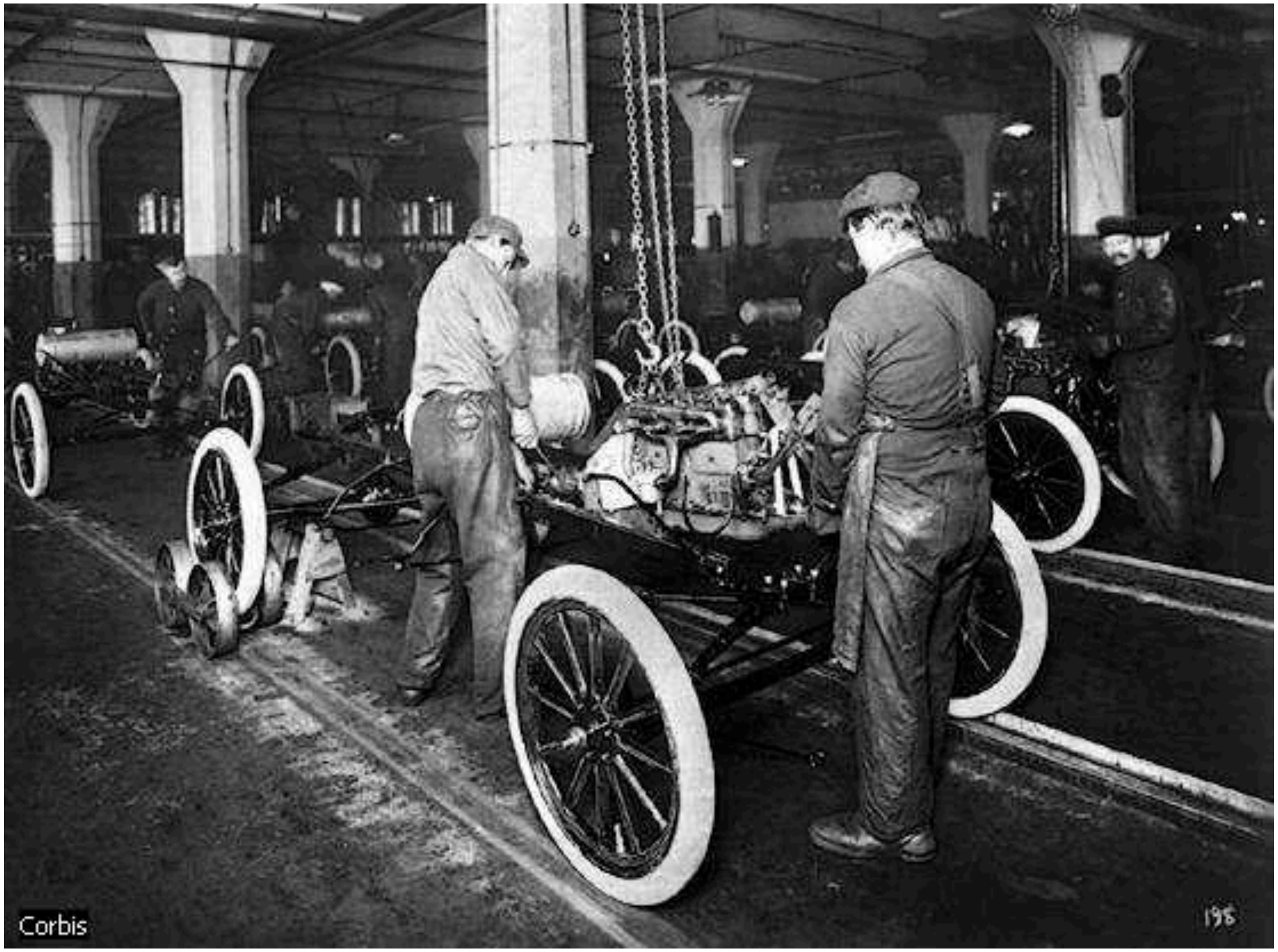
Pipelined laundry takes 3.5 hours for 4 loads

Lessons from Laundry Analogy



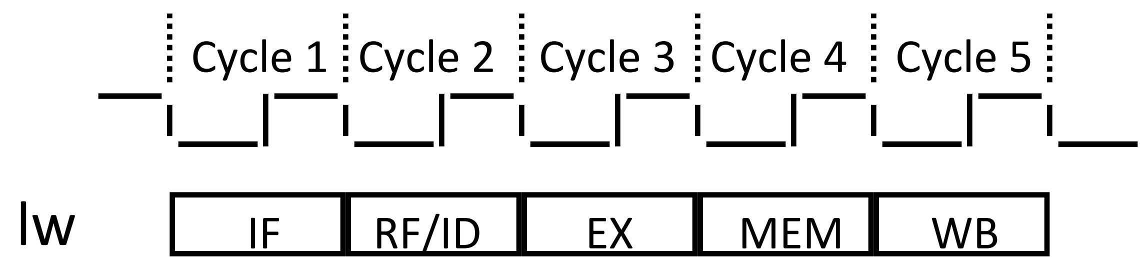
- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously
- Potential speedup = Number pipe stages
- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup

Another Analogy: Model T Assembly Line



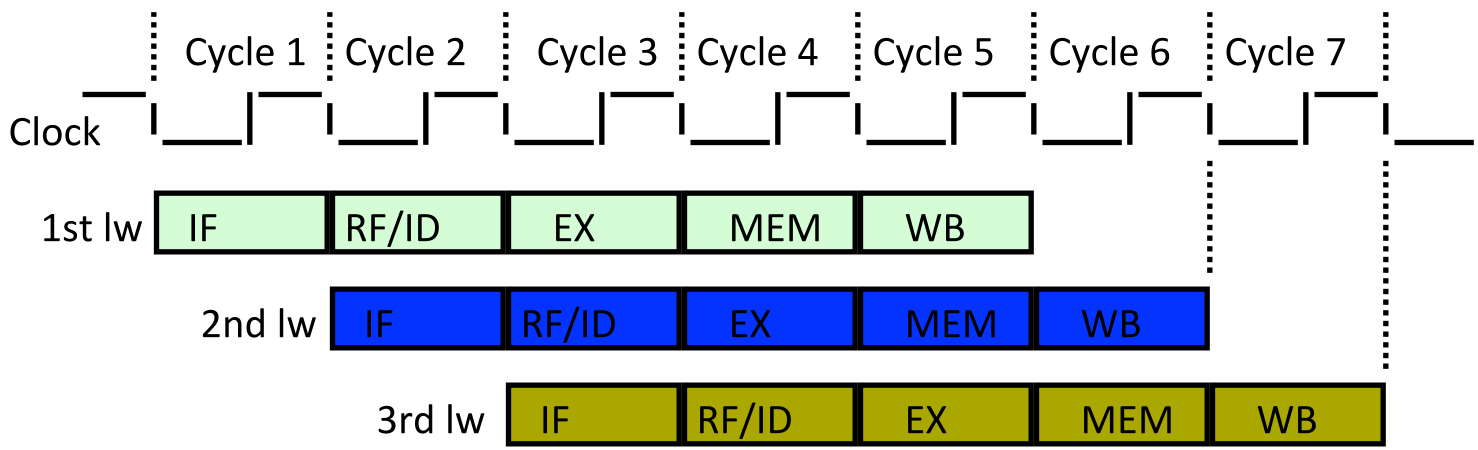
Pipelining the Processor

- 5 stages, one clock cycle per stage
  - IF: instruction fetch from memory
  - ID: instruction decode & register read
  - EX: execute operation or calculate address
  - MEM: access memory operand
  - WB: write result back to register

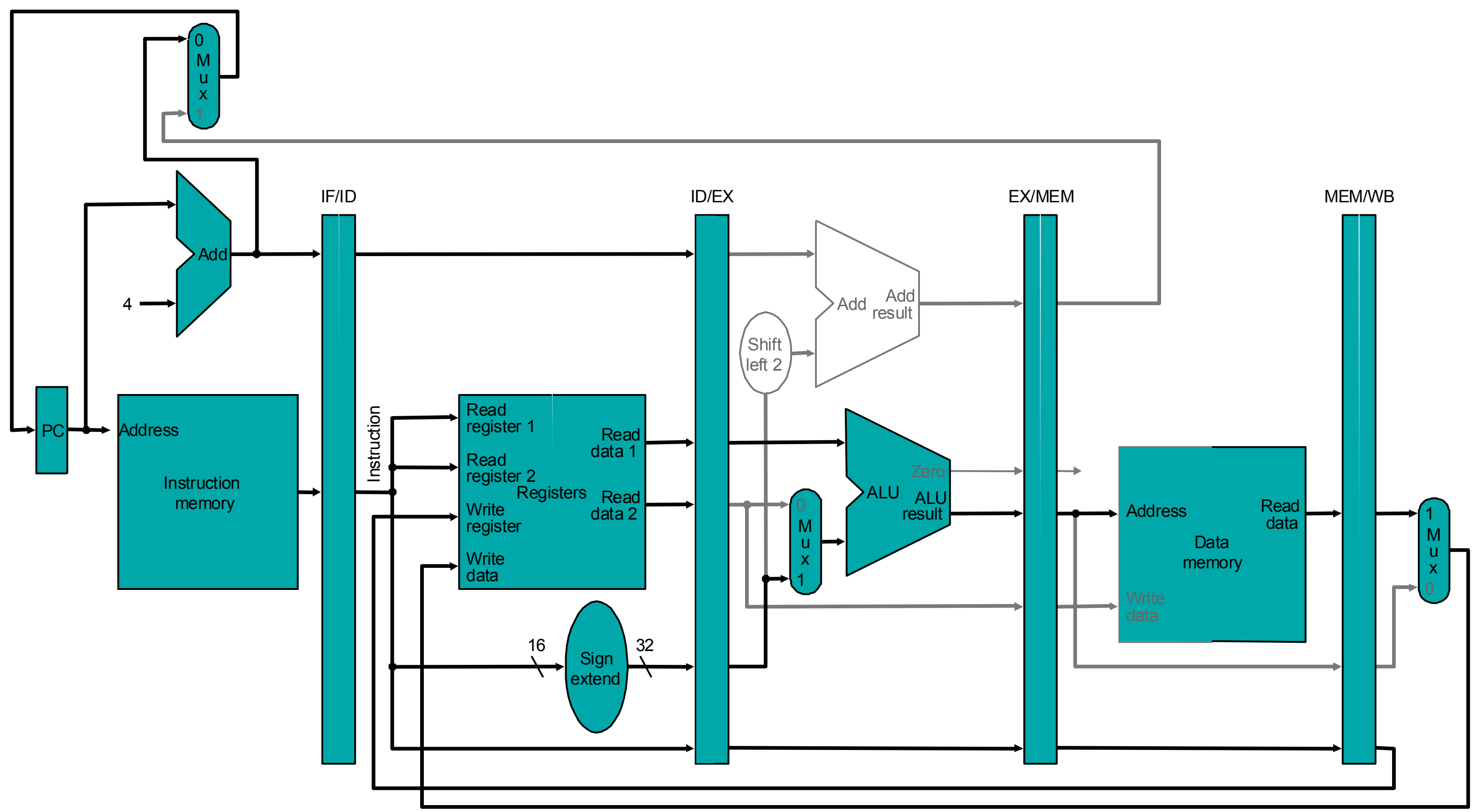


Pipelining the Processor

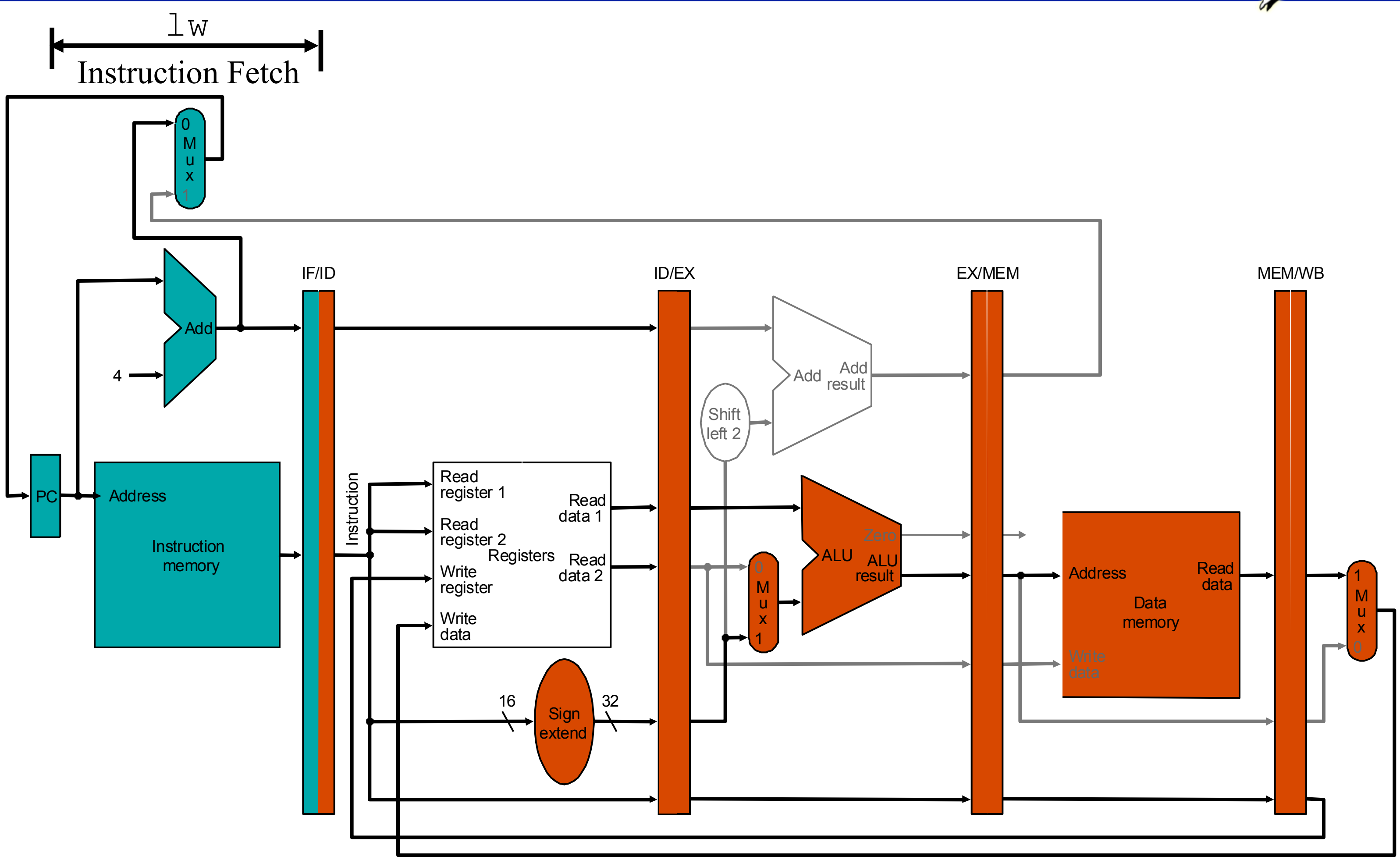
- Overlap instructions in different stages
  - All hardware used all the time
  - Clock cycle is fast
  - CPI is still 1



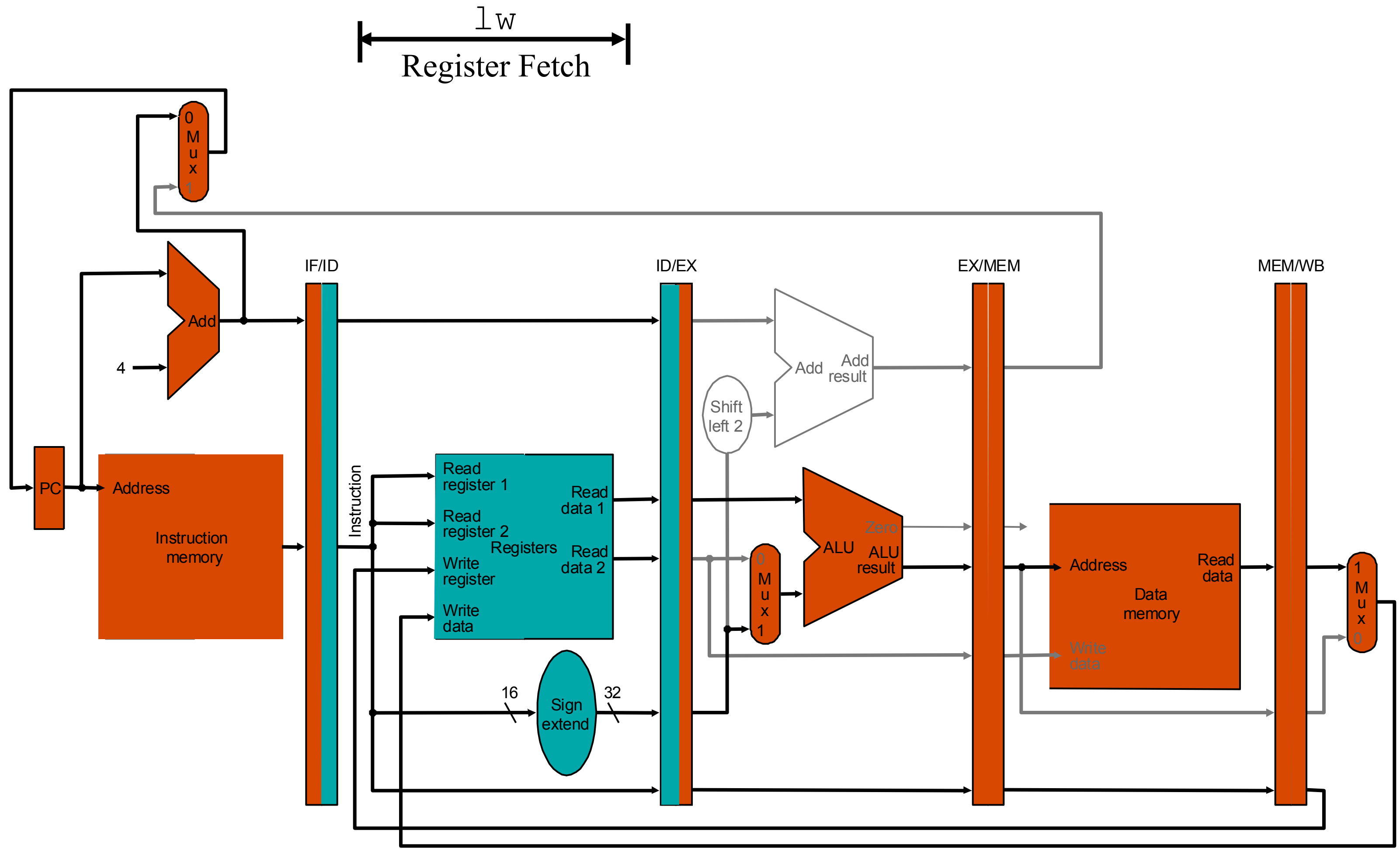
Pipeline Datapath



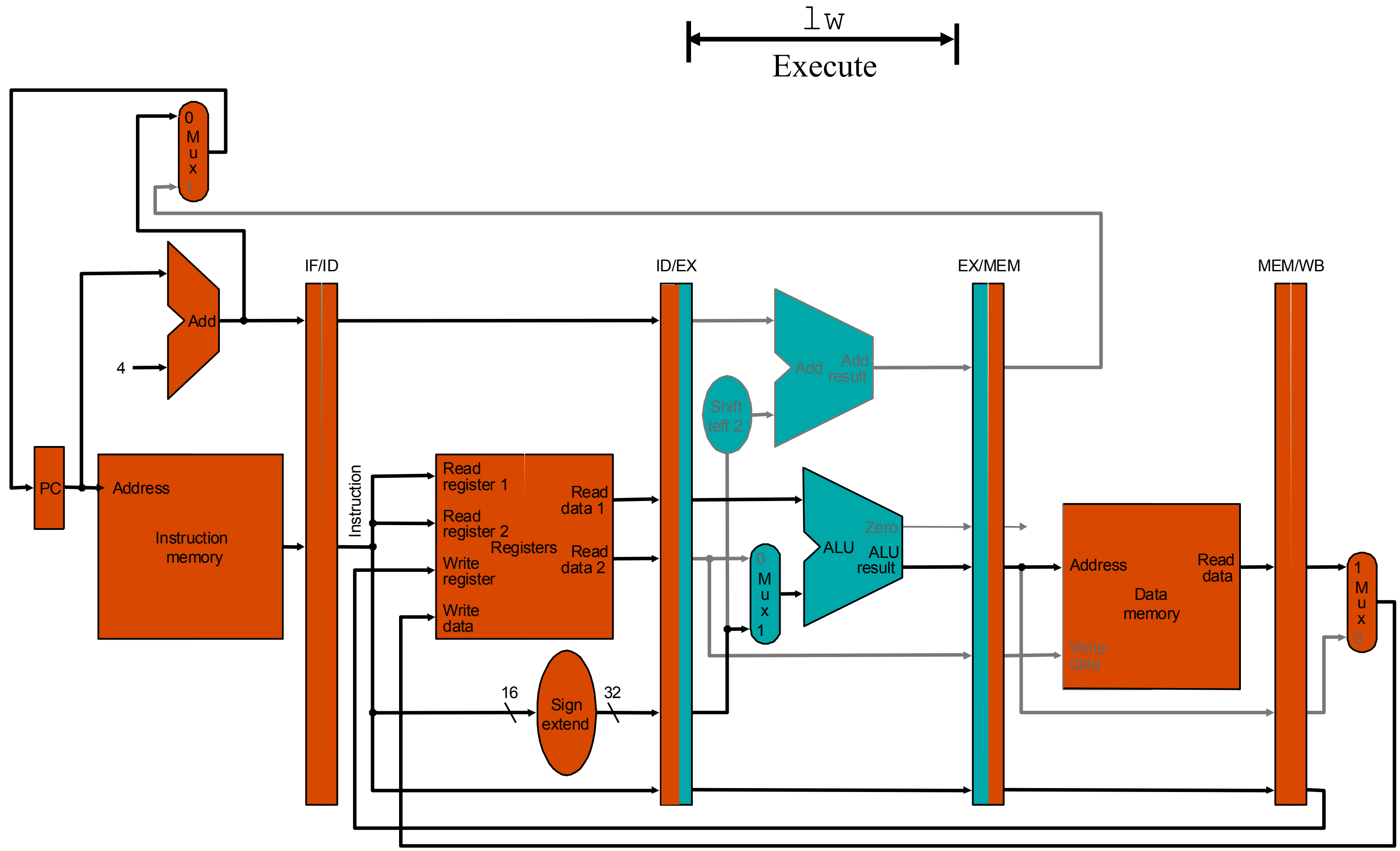
Load: Stage 1 (IF)



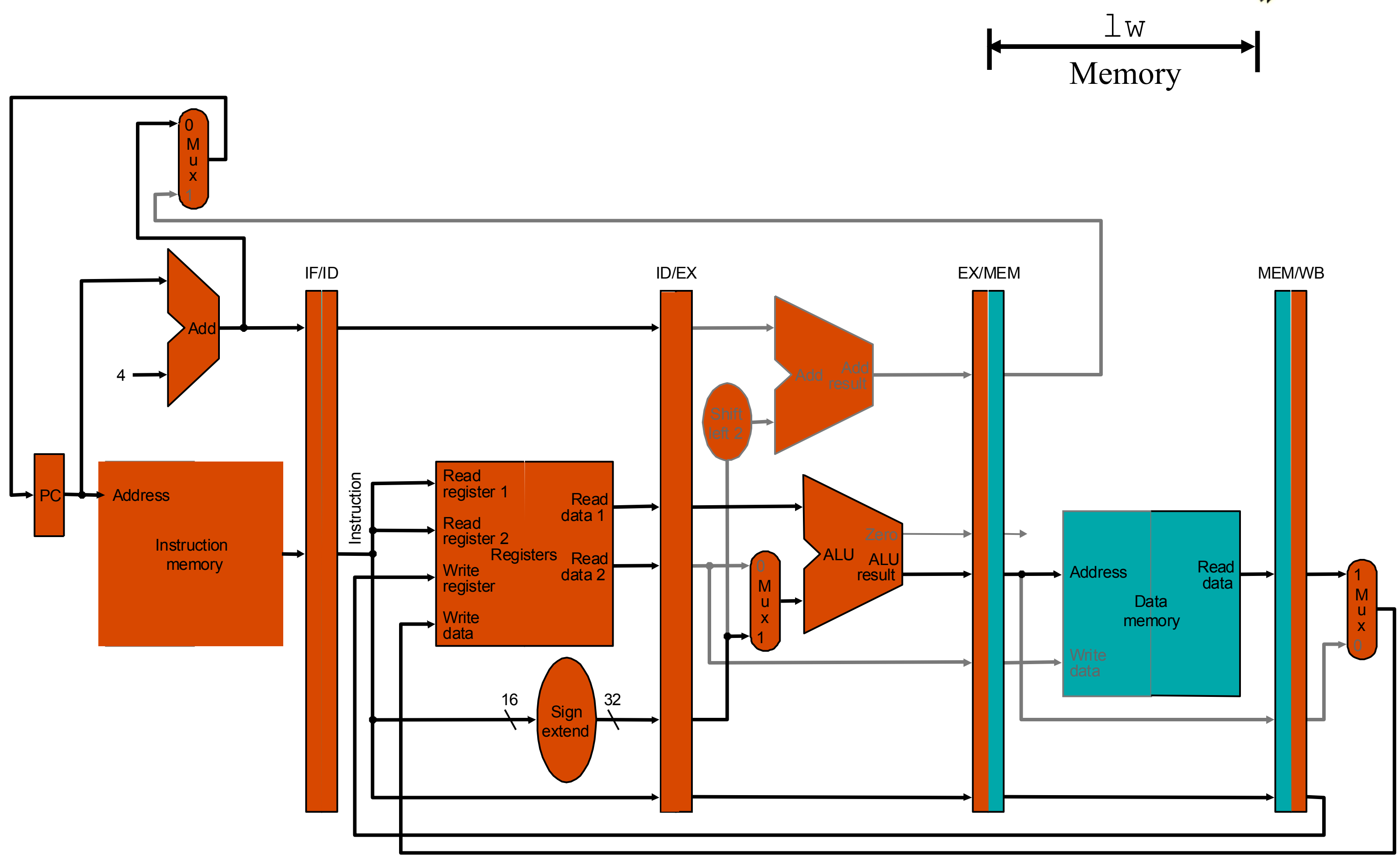
Load: Stage 2 (ID)



Load: Stage 3 (EX)



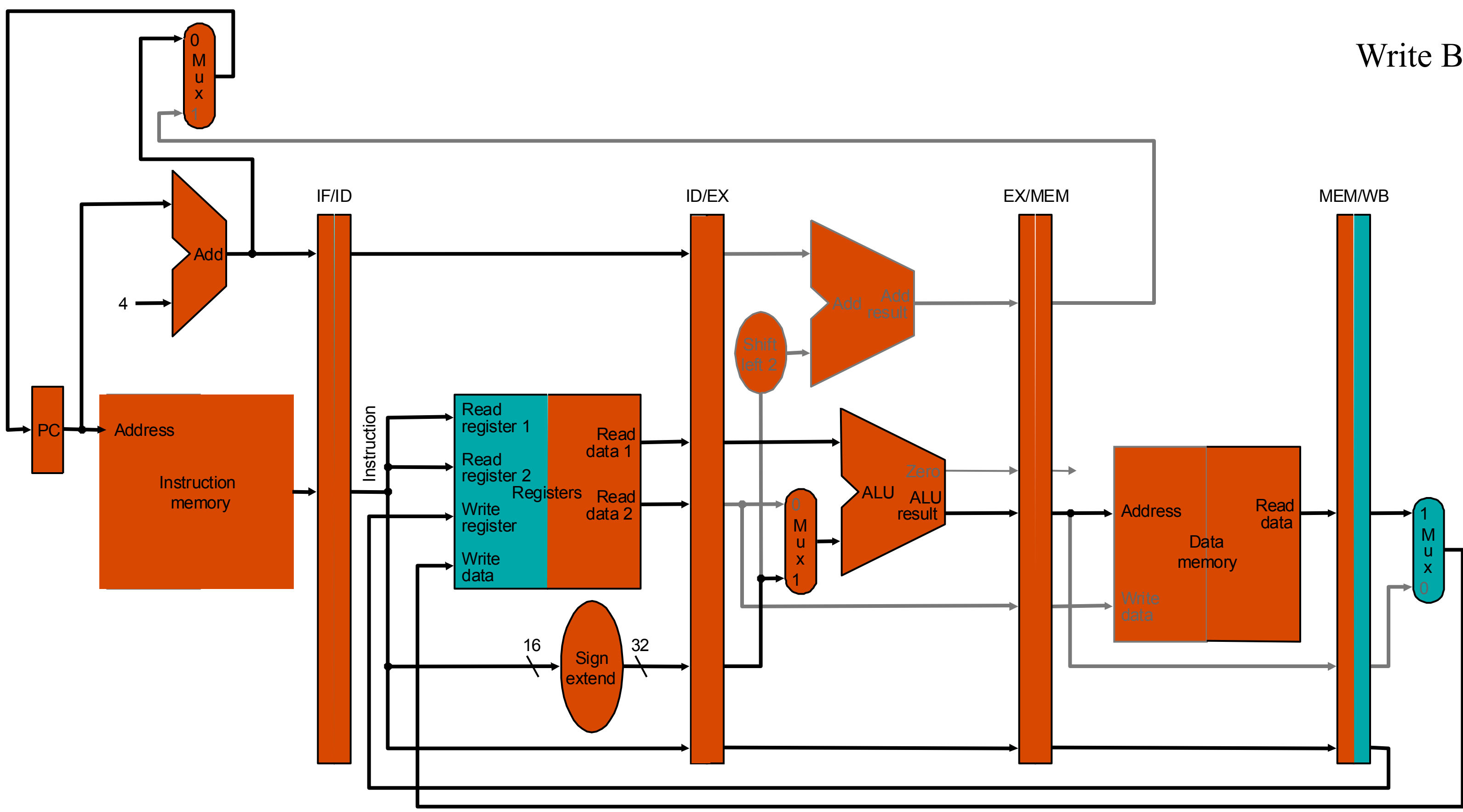
Load: Stage 4 (MEM)



Load: Stage 5 (WB)

lw

Write Back



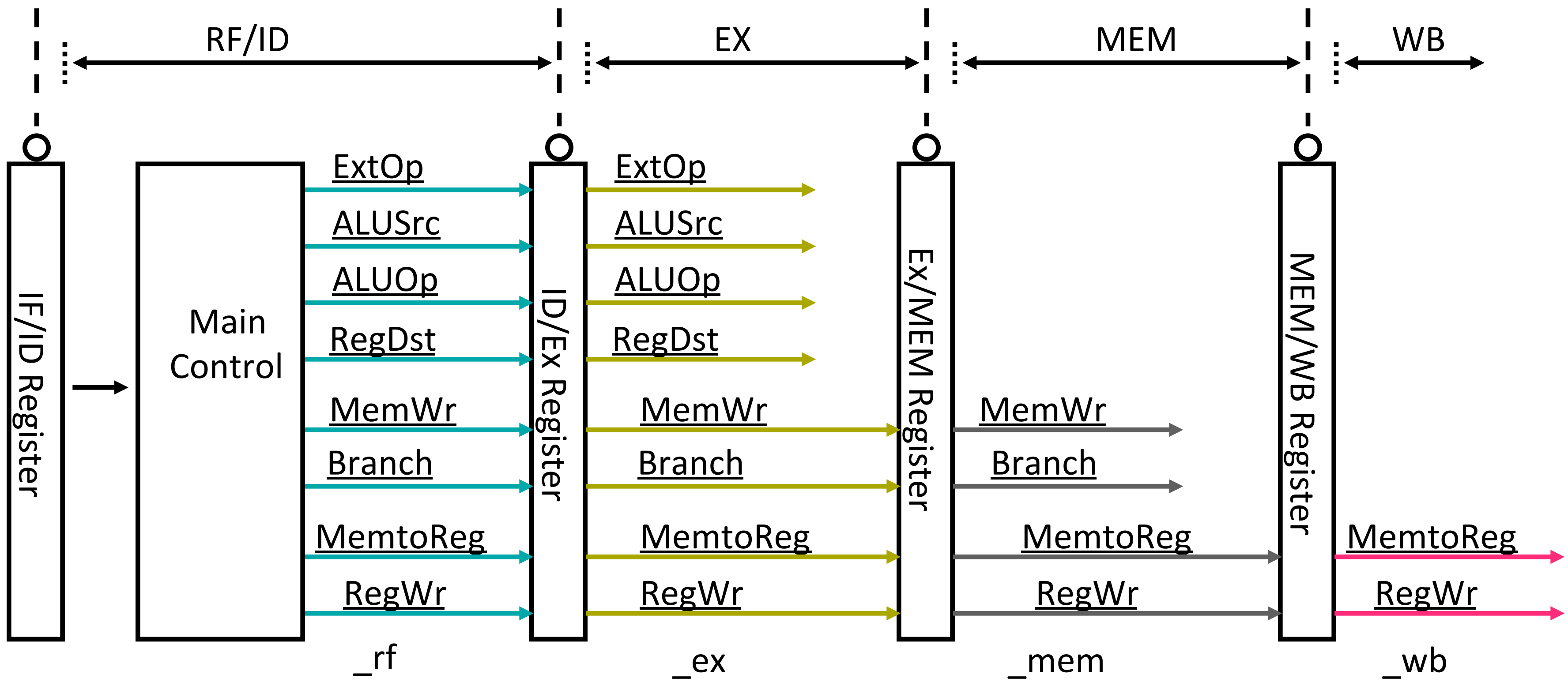
Pipeline Control

- Need to control functional units
  - But they are working on different instructions!
- Not a problem
  - Just pipeline the control signals along with data
  - Make sure they line up
- Using labeling conventions often helps
  - Instruction_rf – means this instruction is in RF
  - Every time it gets flopped, changes pipestage
    - Make sure right signals go to the right places

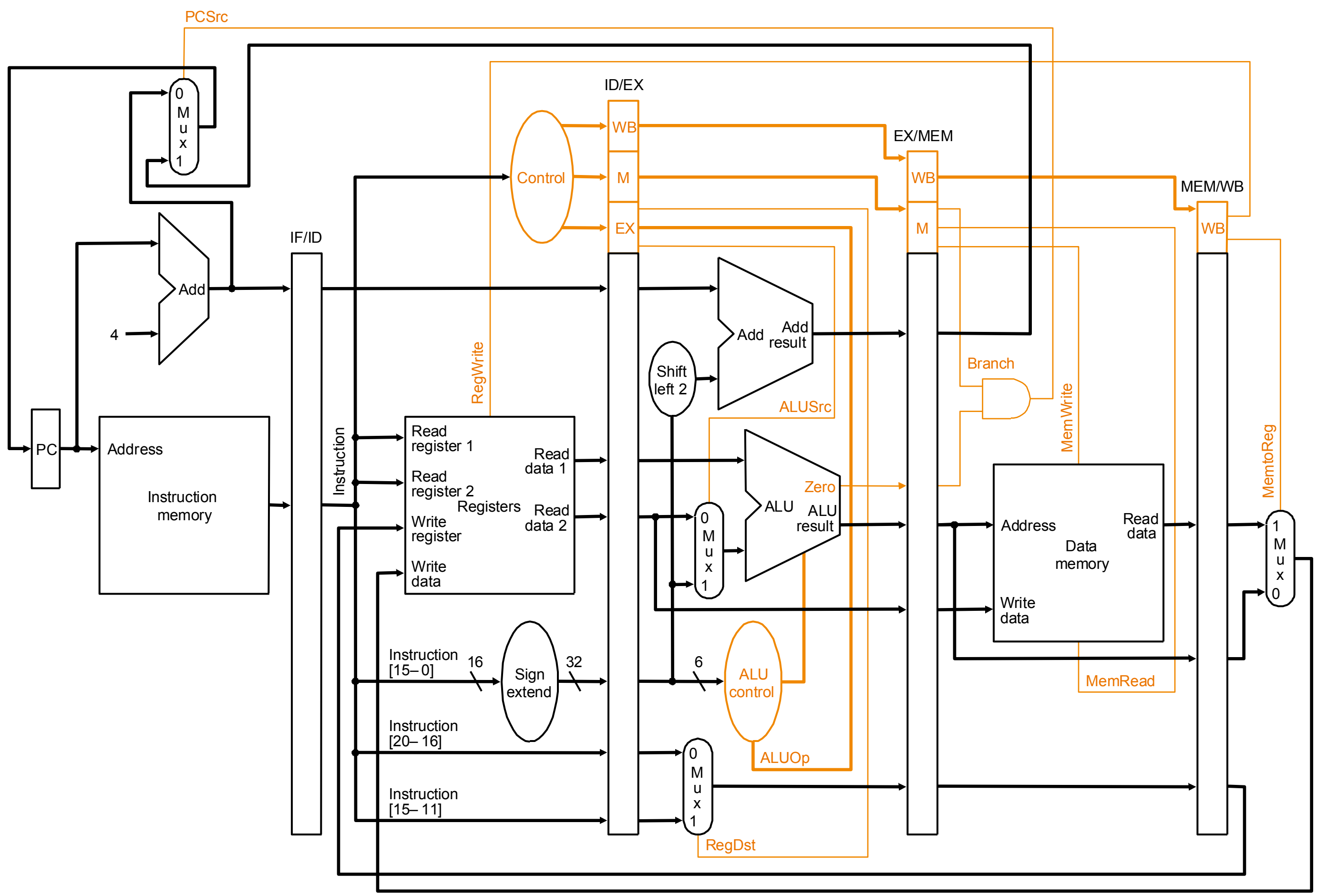
Control Signals

- Same control unit generates signals in ID stage
  - Control signals for EX
    - (ExtOp, ALUSrc, ...) used 1 cycle later
  - Control signals for Mem
    - (MemWr, Branch) used 2 cycles later
  - Control signals for WB
    - (MemtoReg, MemWr) used 3 cycles later

Pipelined Control



Putting it All Together: Pipelined Processor



RISC-V ISA designed for pipelining

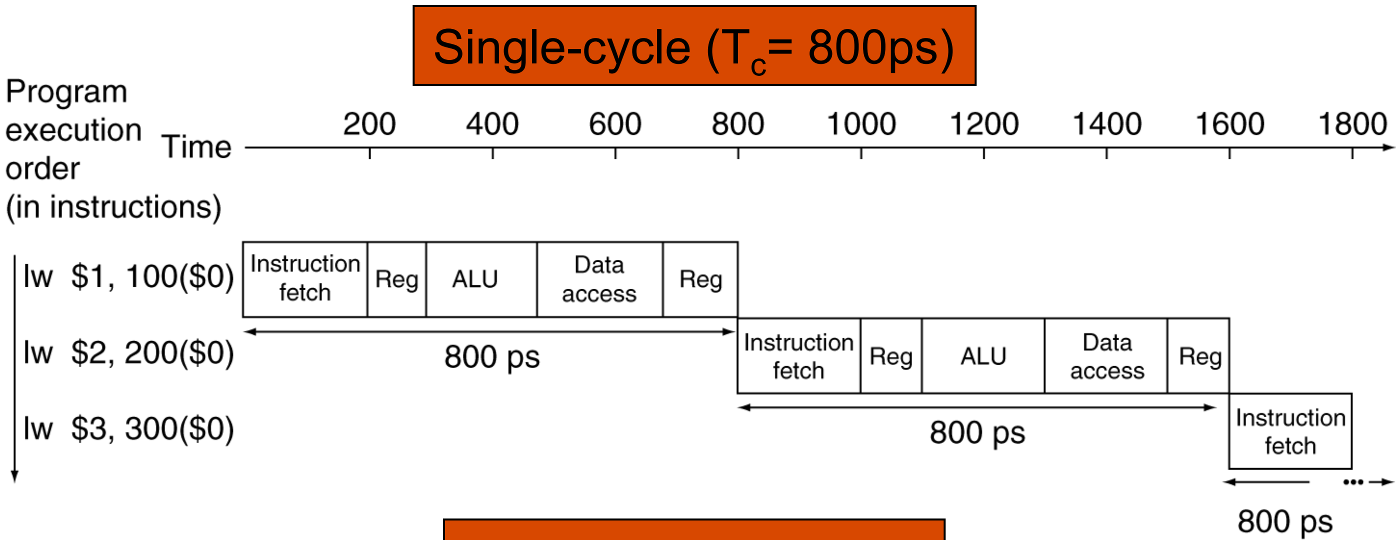
- All instructions are 32-bits
  - Easier to fetch and decode in one cycle
  - c.f. x86: 1- to 17-byte instructions
- Few and regular instruction formats
  - Can decode and read registers in one step
- Load/store addressing
  - Can calculate address in 3rd stage, access memory in 4th stage

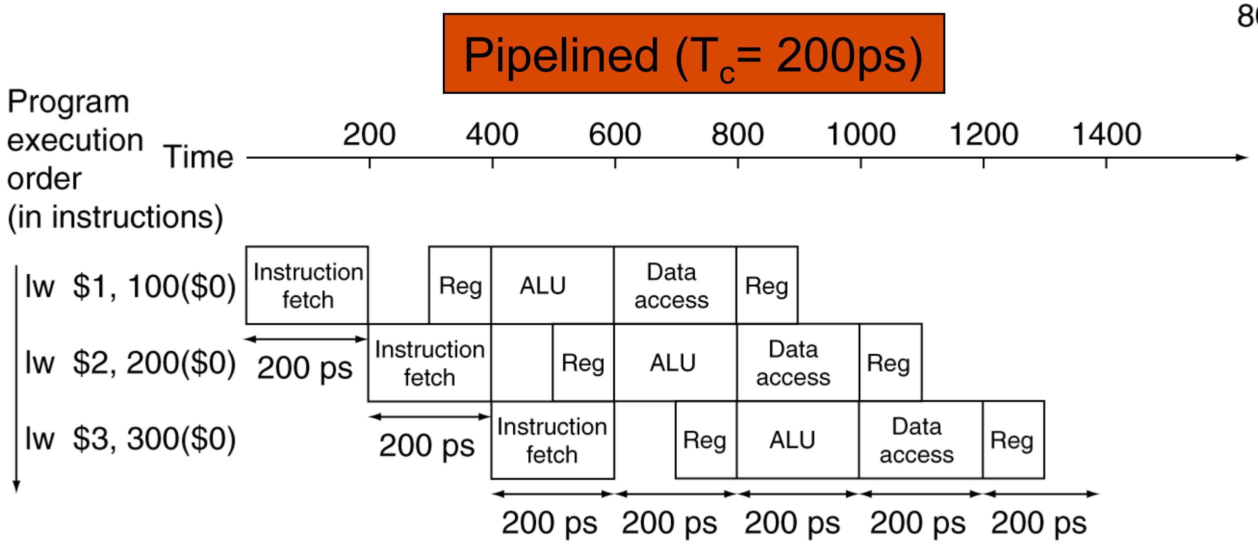
Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined with single-cycle processor

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
ALU ops	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Pipeline Performance





But Something Feels Wrong

- Why stop at 5 pipeline stages?
  - If pipelining improves Tclock & CPI=1
  - We should keep subdividing the cycle

- Three issues
  - Some things have to complete in a cycle
  - CPI is not really one
  - Cost (area and power)

Quiz

- Ignoring all other issues, what is the highest clock frequency you can achieve with pipelining?
  - Lowest clock cycle time?
- What are the limiting factors?

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

Structural Hazards

- Resource conflict
  - Two instructions use same hardware in the same cycle
- Example: pipeline with a single unified memory
  - No separate instruction & data memories
  - Load/store requires data access
  - One instruction would have to stall for that cycle
    - Which one?
    - Would cause a pipeline "bubble"
- Other examples
  - Functional units that are not fully pipelined (mult, div)

Avoiding Structural Hazards

1. Do nothing (performance hit)
2. Replicate resources
   - Separate instruction/data memories, multiported memories, …
3. Design away the structural stall
   - Use resource once per instruction, always in the same stage
   - Example of bad pipeline arrangement
     - Load uses Register File’s Write port during its 5th stage

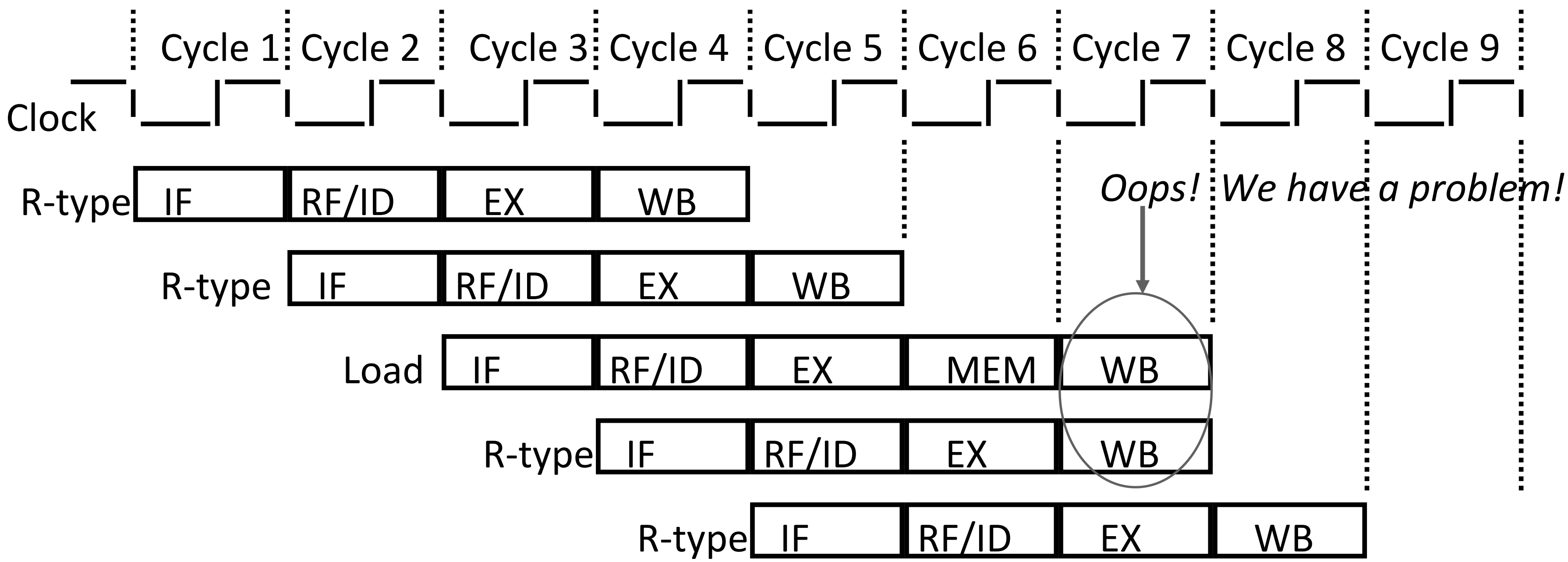
	1	2	3	4	5
Load	IF	RF/ID	EX	MEM	WB

     - R-type uses Register File’s Write port during the 4th stage

	1	2	3	4
R-type	IF	RF/ID	EX	WB

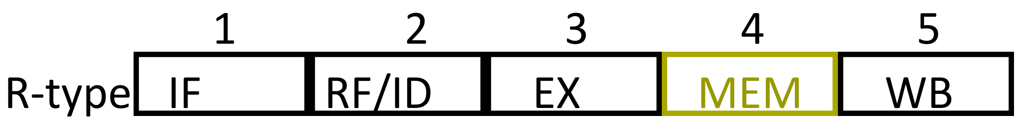
Structural Hazard Example

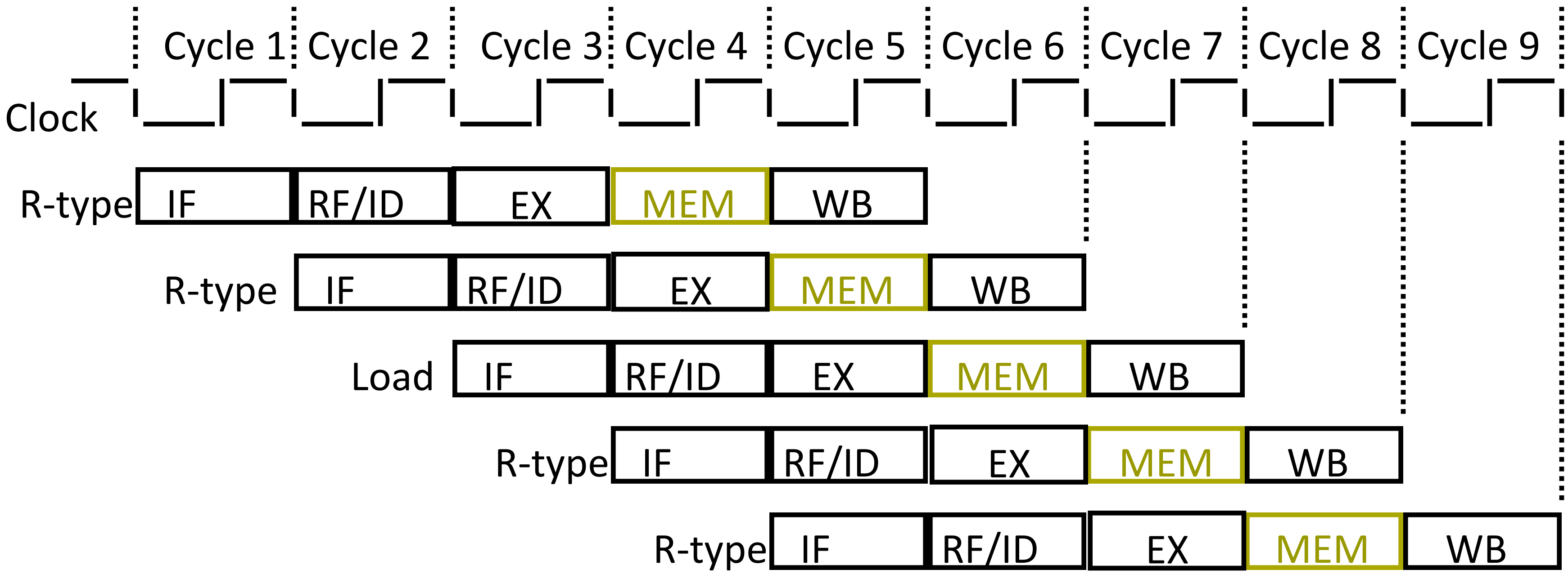
- Consider a load followed immediately by an ALU operation
  - Register file only has a single write port
  - But need to write the results of the ALU and the memory back



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?





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
```

Analyzing the Problem

- Can an output dependency cause a WAW hazard in 5-stage pipeline?
- Can an anti-dependency cause a WAR hazard in 5-stage pipeline?
- Are these answers universally true?

Dealing with RAW Hazards

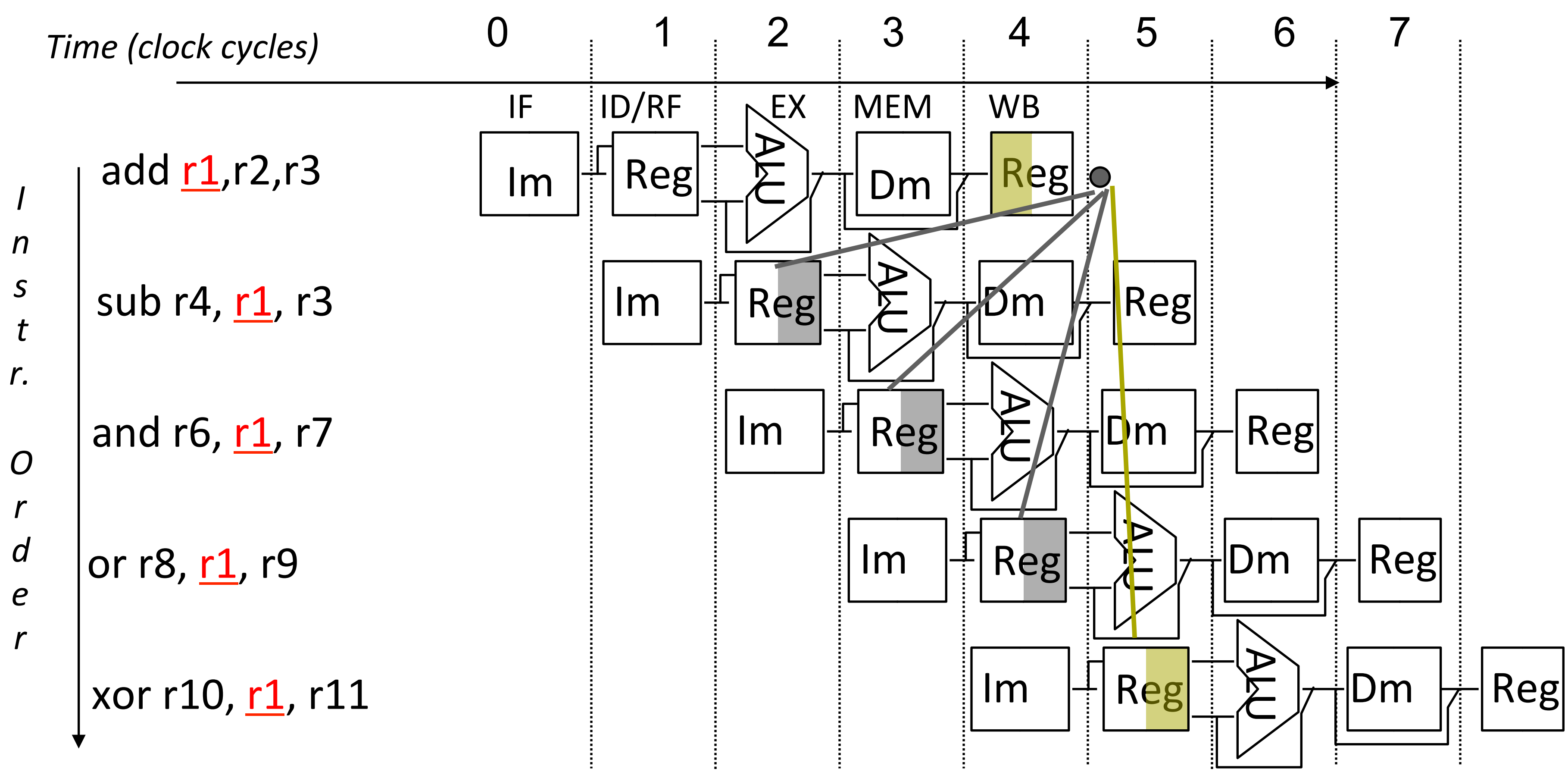
- Must keep our "promise" in the instruction set
  - Each instruction fully completes before next on starts
  - All RAW dependencies are respected

- Pipelining may break this promise
  - Overlapping i and j
  - i writes late in the pipeline (WB); j reads early (ID)

- Must ensure that programmers cannot observe this behavior
  - Without necessarily reverting to single-cycle design…

RAW Hazard Example

- Dependencies backwards in time are hazards



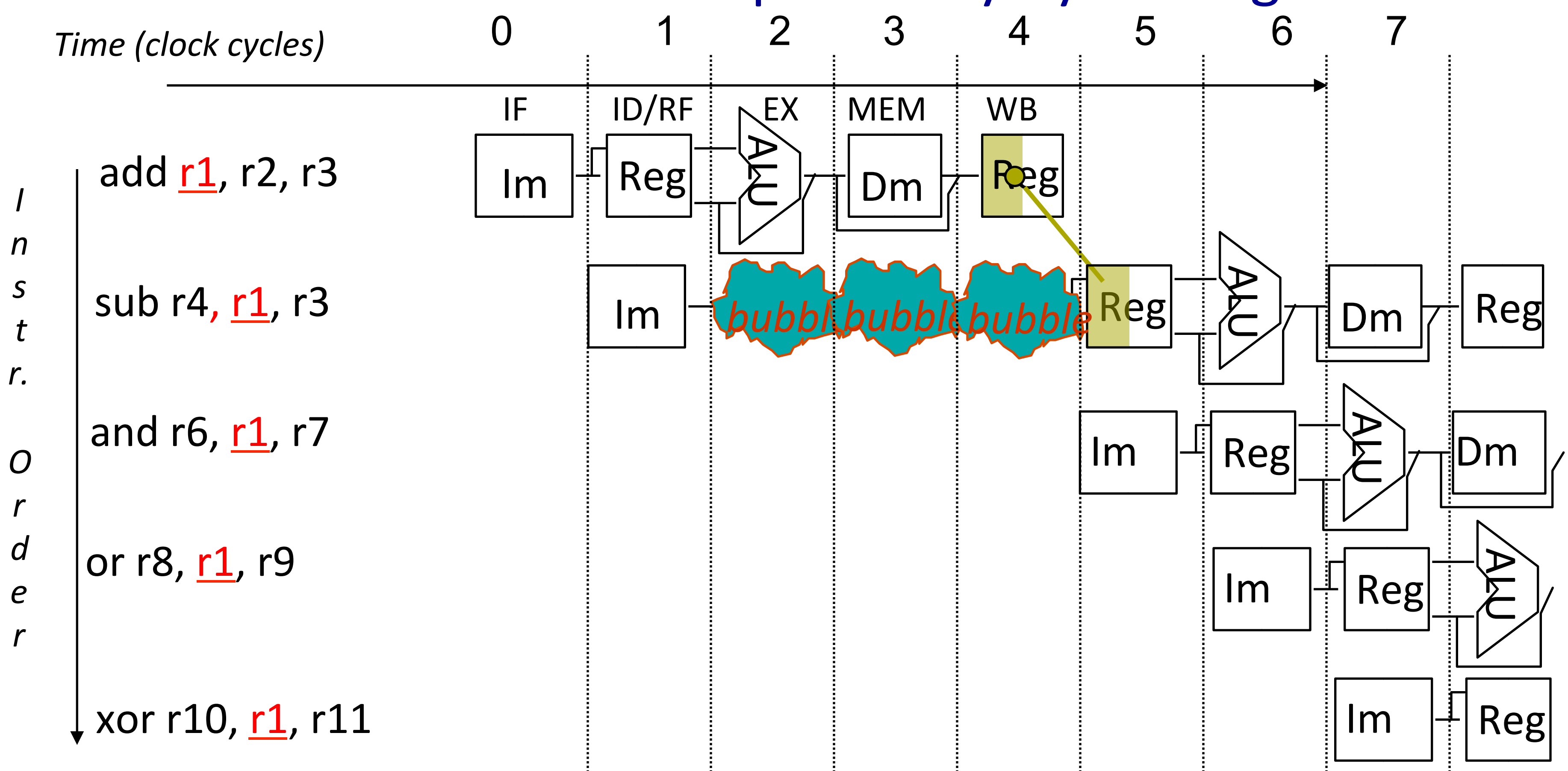
Solutions for RAW Hazards

- Delay the reading instruction until data is available
  - Also called stalling or inserting pipeline bubbles

- How can we delay the younger instruction?
  - Compiler insert independent work or NOPS ahead of it
    - NOP example: or x0, x0, x0
    - Disadvantage: pipeline-specific binary program
  - Hardware inserts NOPs as needed (interlocks)
    - Advantage: correct operation for all programs/pipelines
    - Disadvantage: may miss some optimization opportunities
  - Most modern machines
    - Hardware inserts NOPs but compiler may try to minimize need

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!

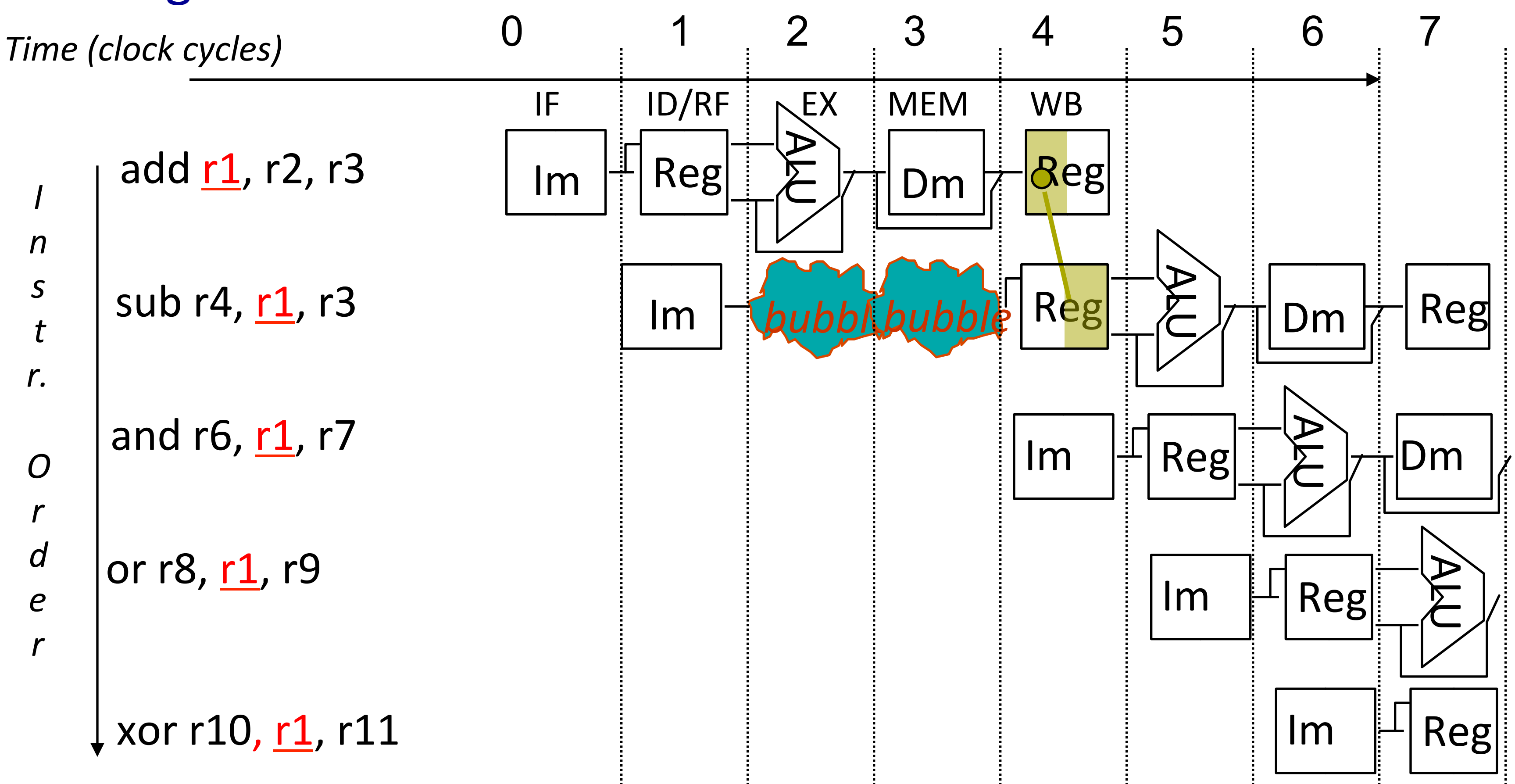
Reducing Stalls

- 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

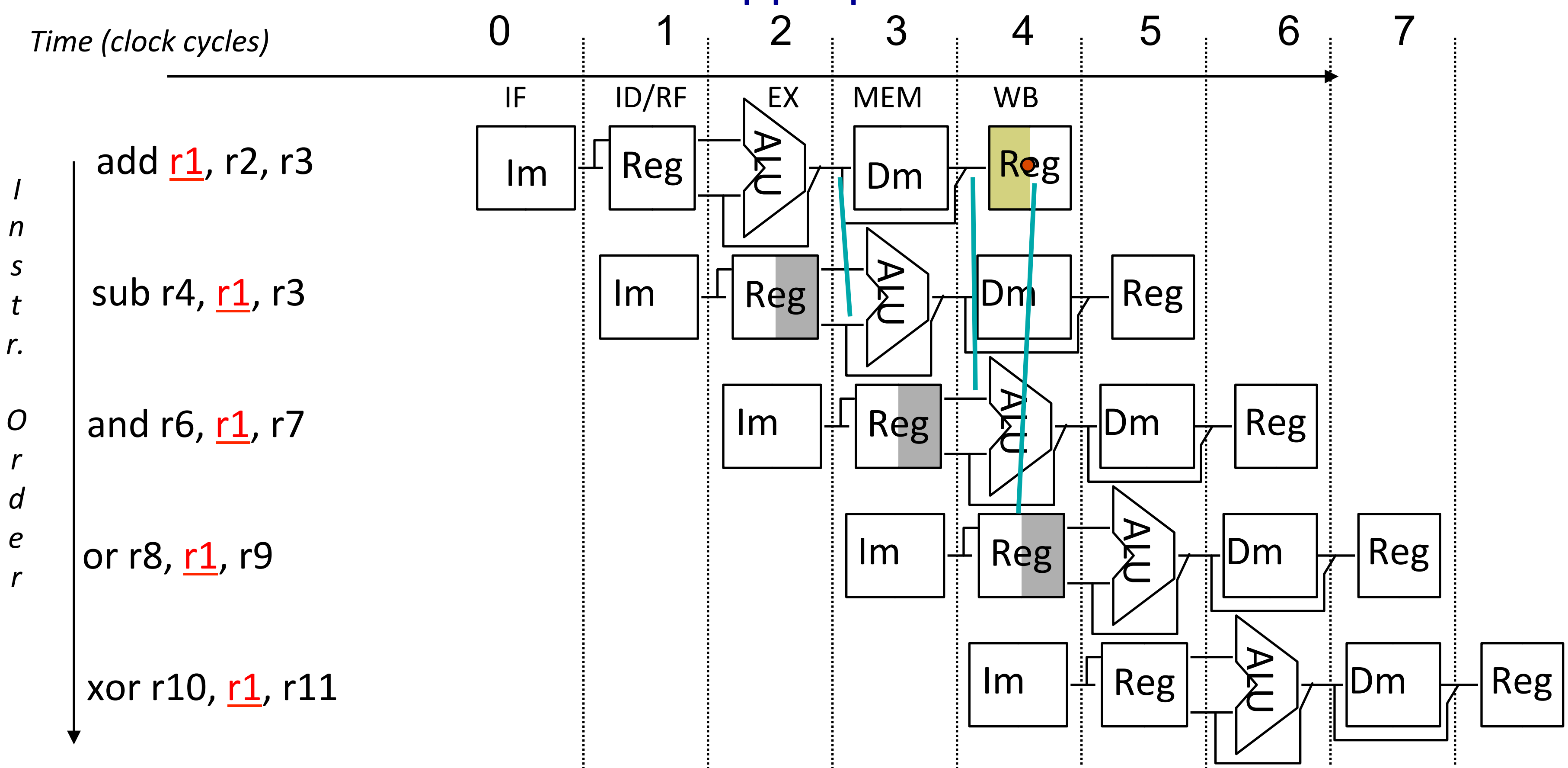


Performance Effect

- 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 2 cycle stall
- If 40% of the instructions cause a stall?
  - The new effective CPI is $1 + 2 \times 0.4 = 1.8$
  - And the real % is probably higher than 40%
- You get a little more than ½ the desired performance!

Decreasing Stalls: Forwarding

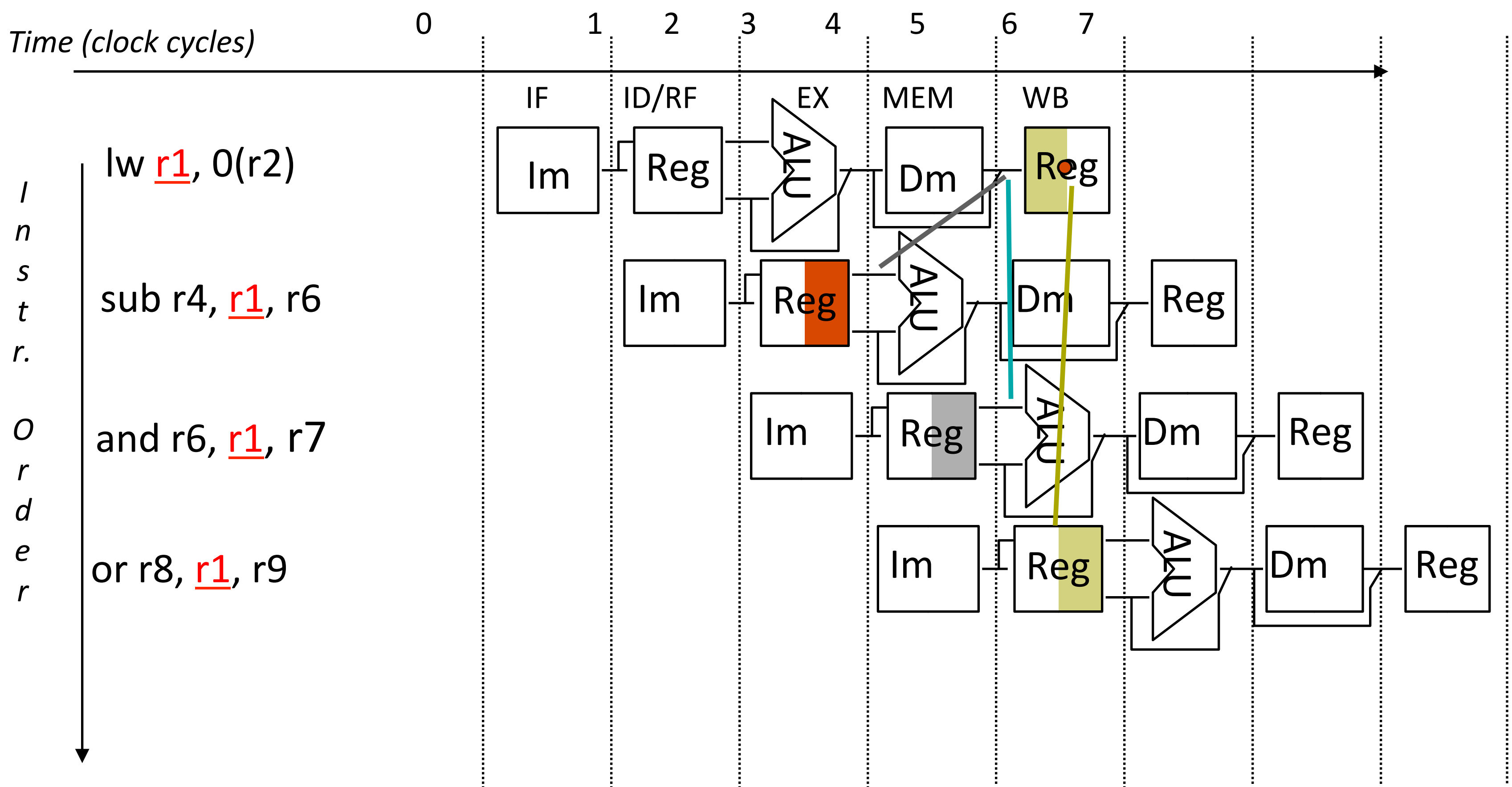
- "Forward" the data to the appropriate unit



Eliminates stalls for dependencies between ALU instrs.

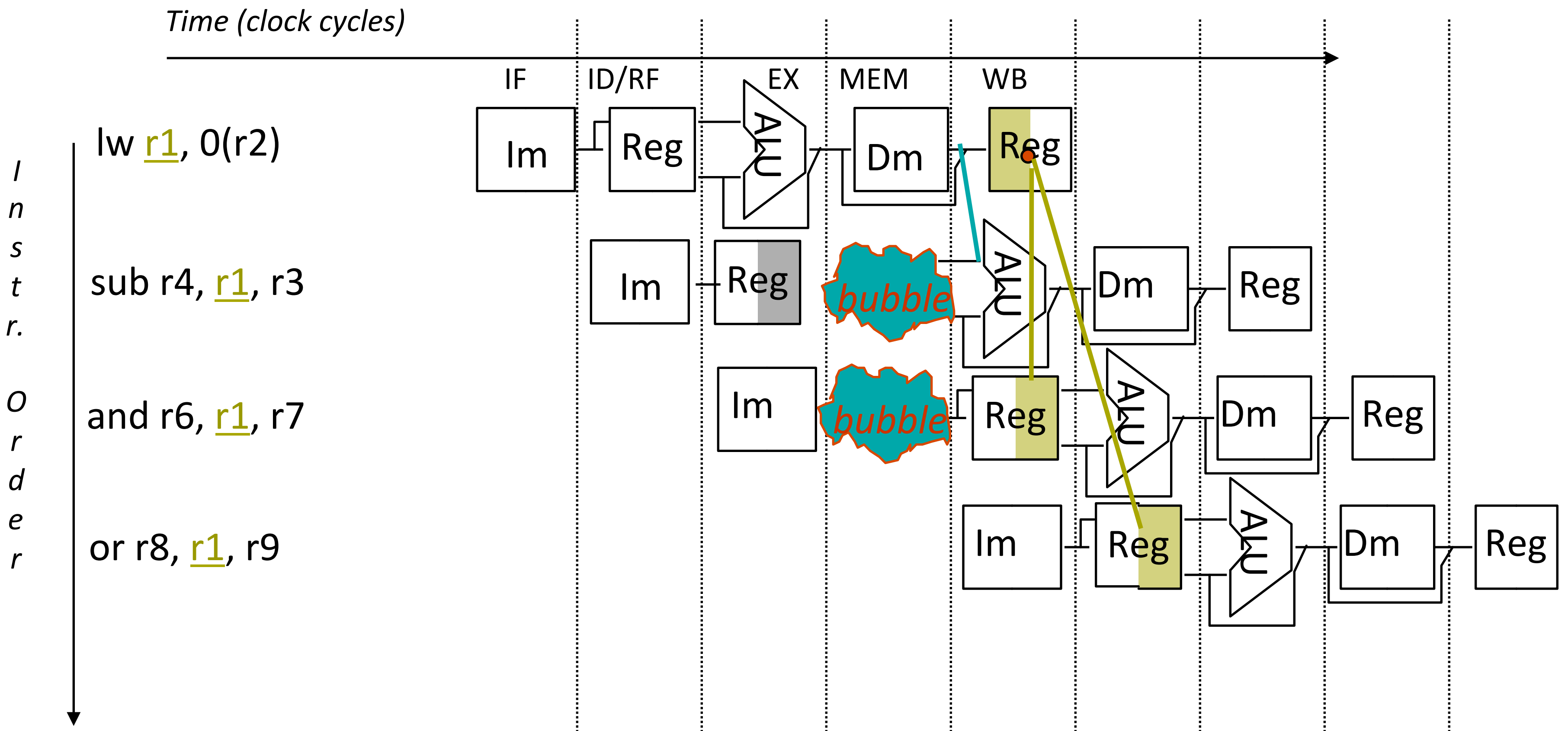
Forwarding Limitation: Load-Use Case

- Data is not available yet to be forwarded



Load-Use Case: Hardware Stall

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



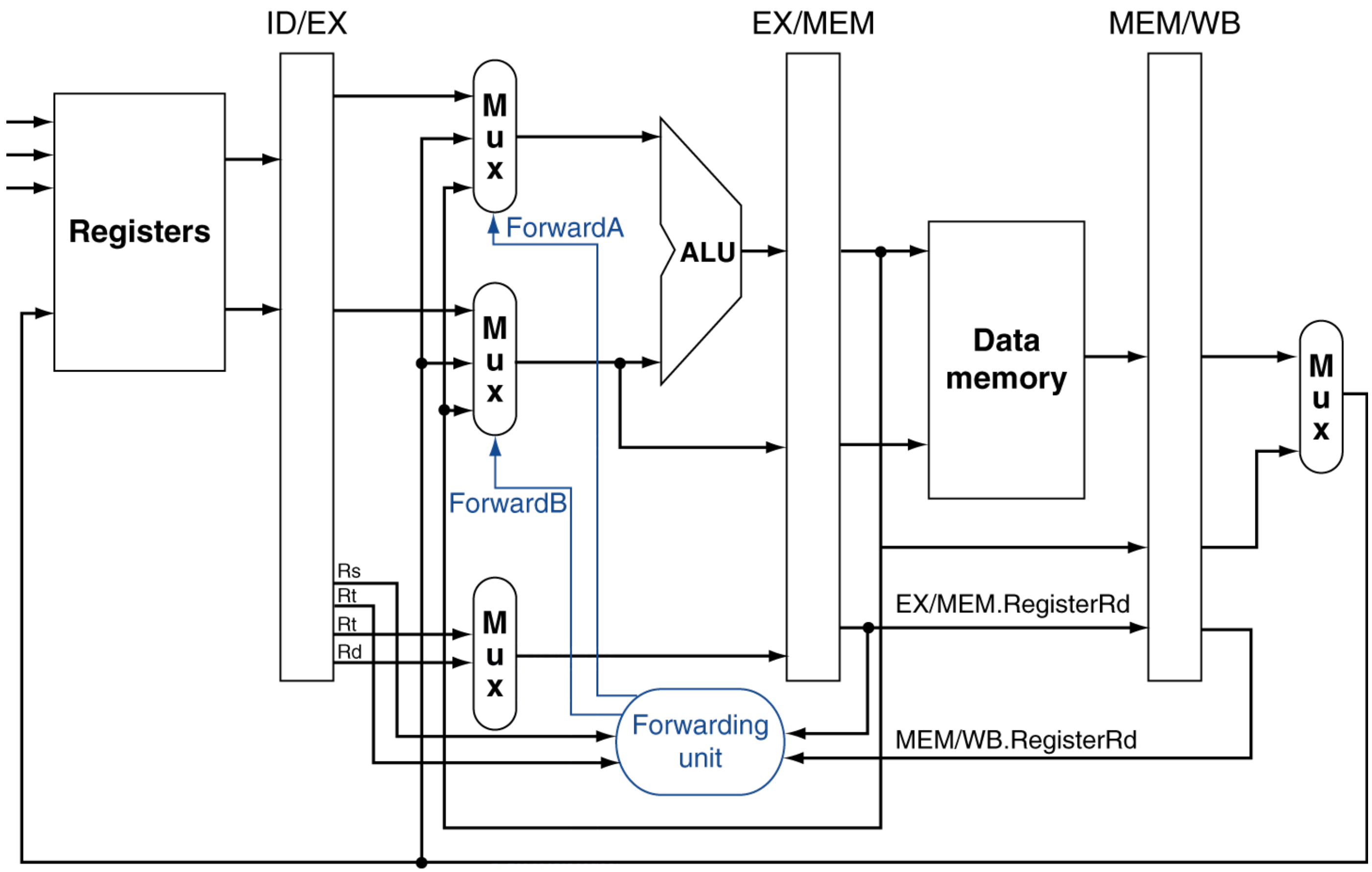
Identifying the Forwarding Datapaths

- 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



b. With forwarding

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:

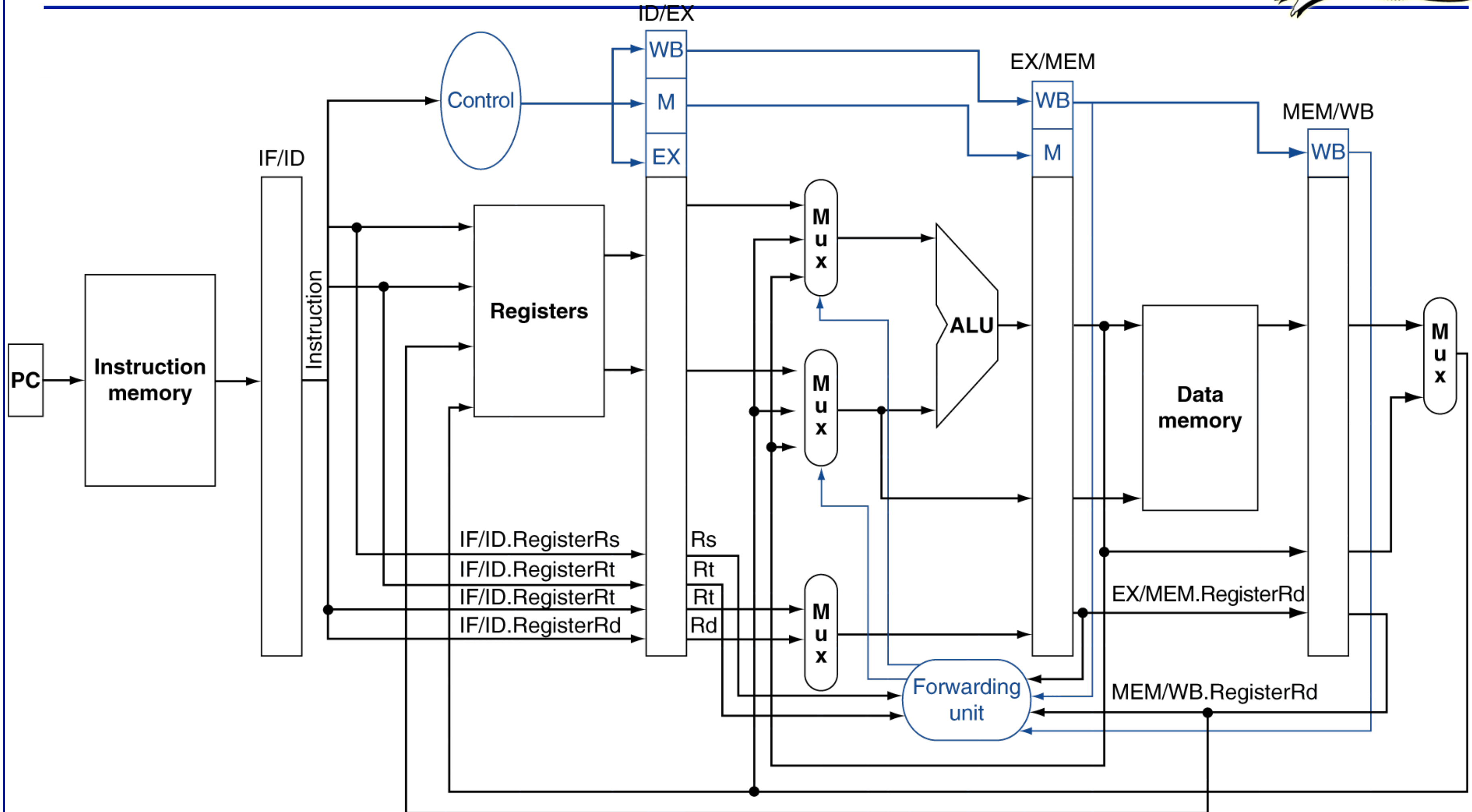
```
add $1,$1,$2
sub $1,$1,$3
or  $1,$1,$4
```

- 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

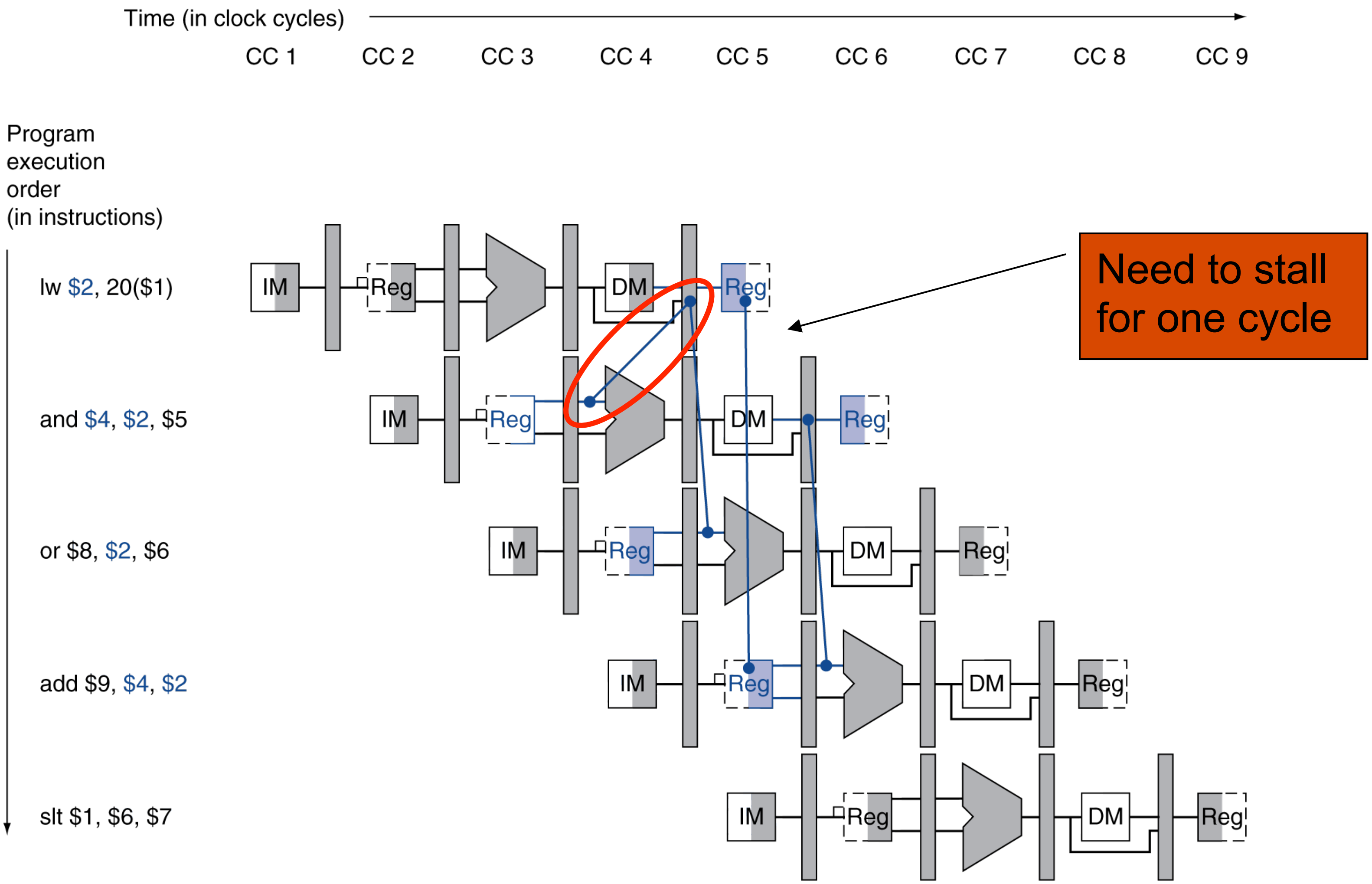
Forwarding Control (Revised)

- 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



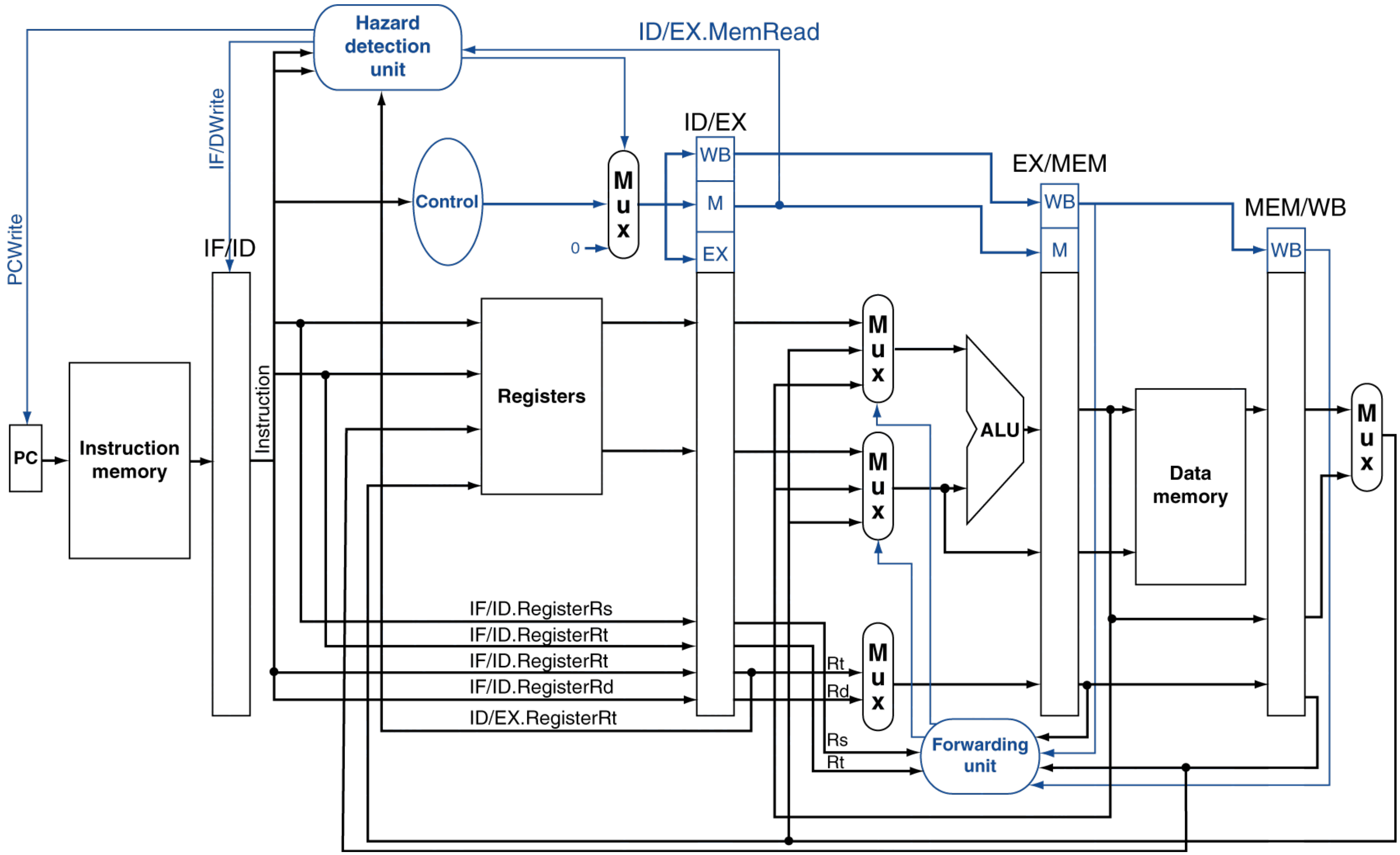
Load-Use Data Hazard



Load-Use Hazard Detection

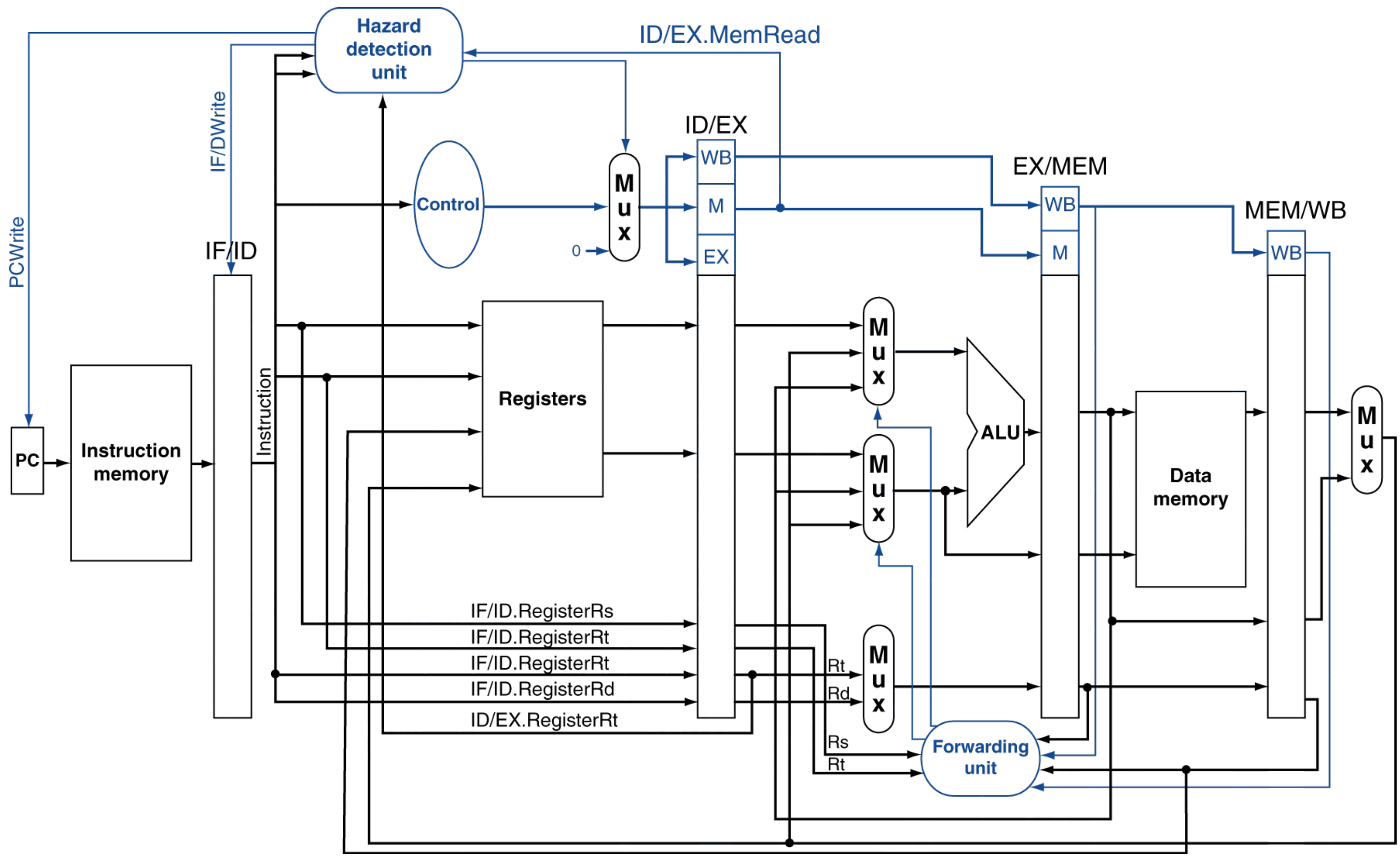
- Check when use instruction is decoded in ID stage
- ALU register numbers in ID stage are given by
  - IF/ID.RegisterRs, IF/ID.RegisterRt
- Load-use hazard when
  - ID/EX.MemRead and
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
    (ID/EX.RegisterRt = IF/ID.RegisterRt))
- If detected, stall and insert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

Datapath with Hazard Detection



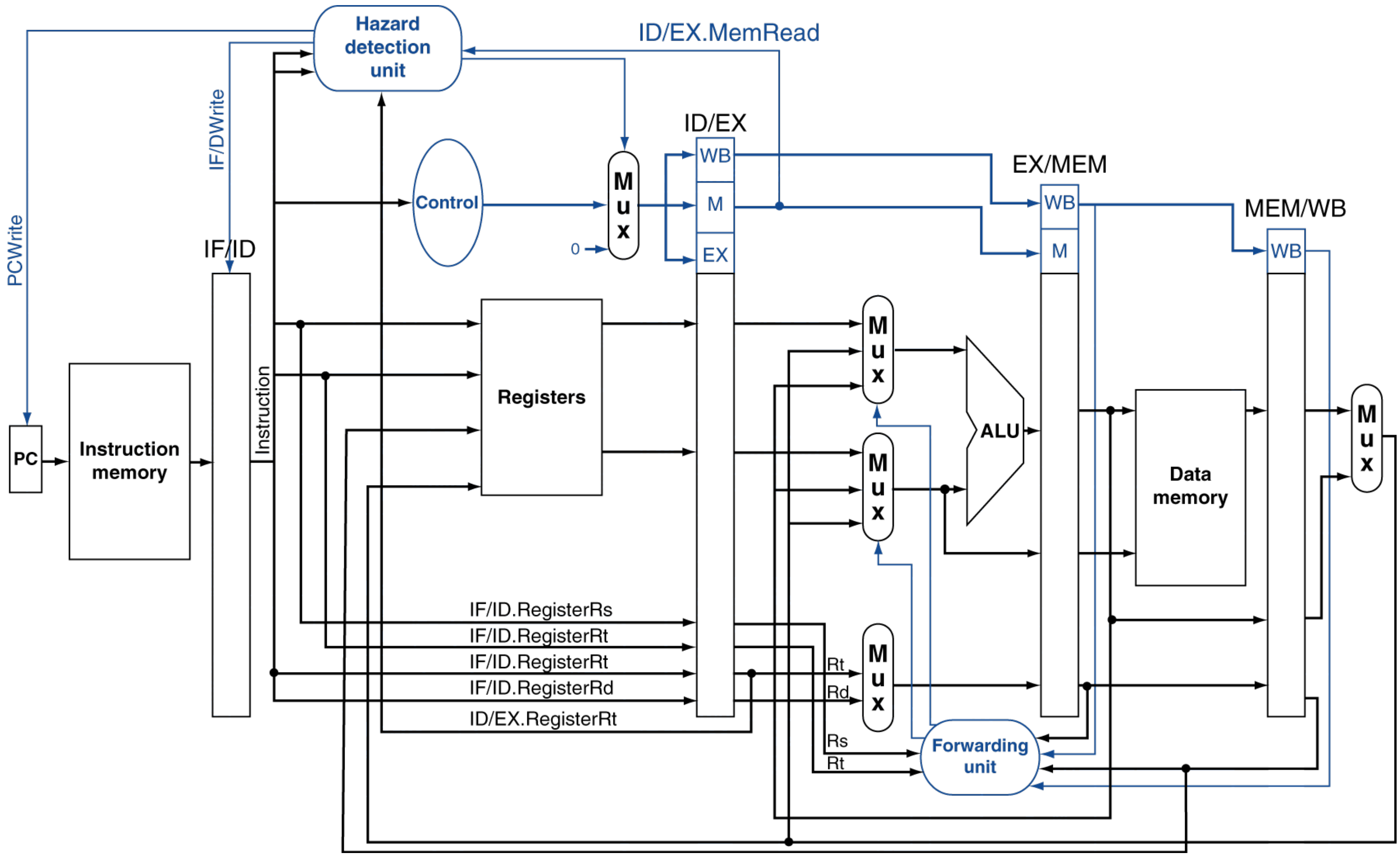
Example: Load-Use Stall

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



Example: Load-Use Stall
1 cycle later

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



Looking Ahead

- Compilers and data hazards
- Control hazards
- Exceptions and interrupts
- Advanced pipelining – (CPI < 1.0)