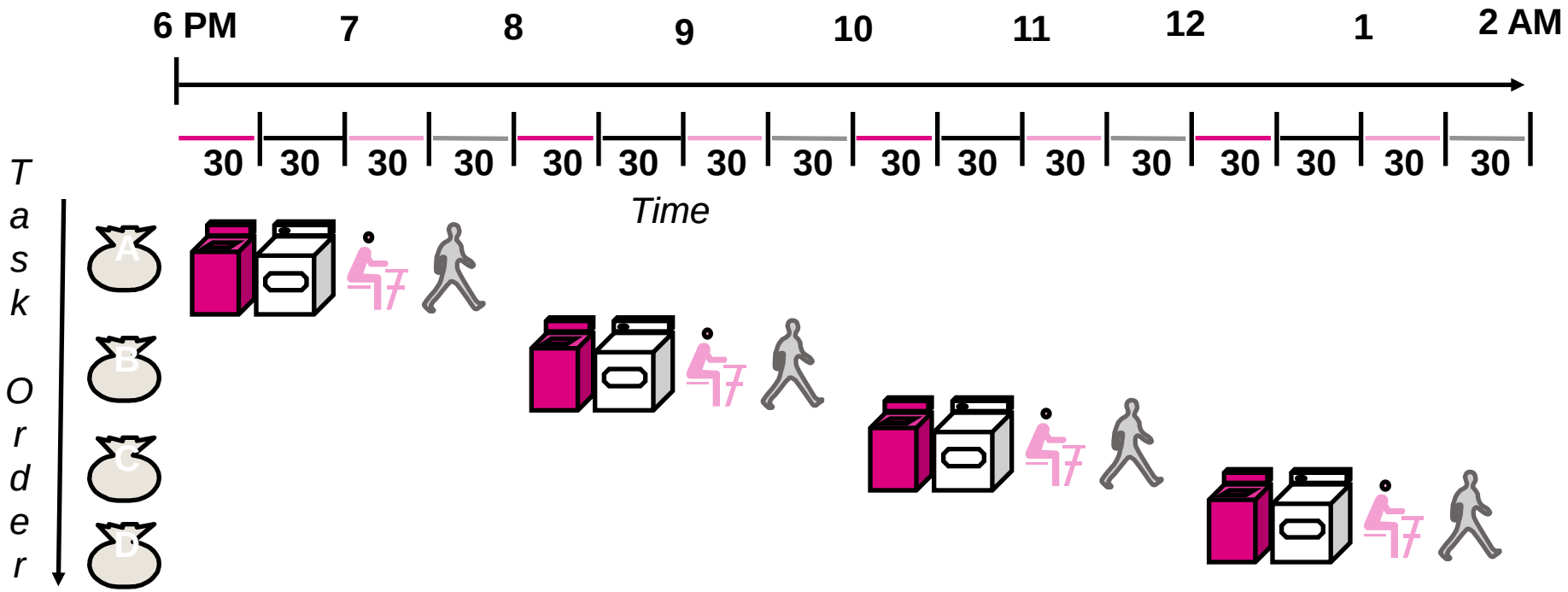
Chapter 4
(Part II)

The Processor

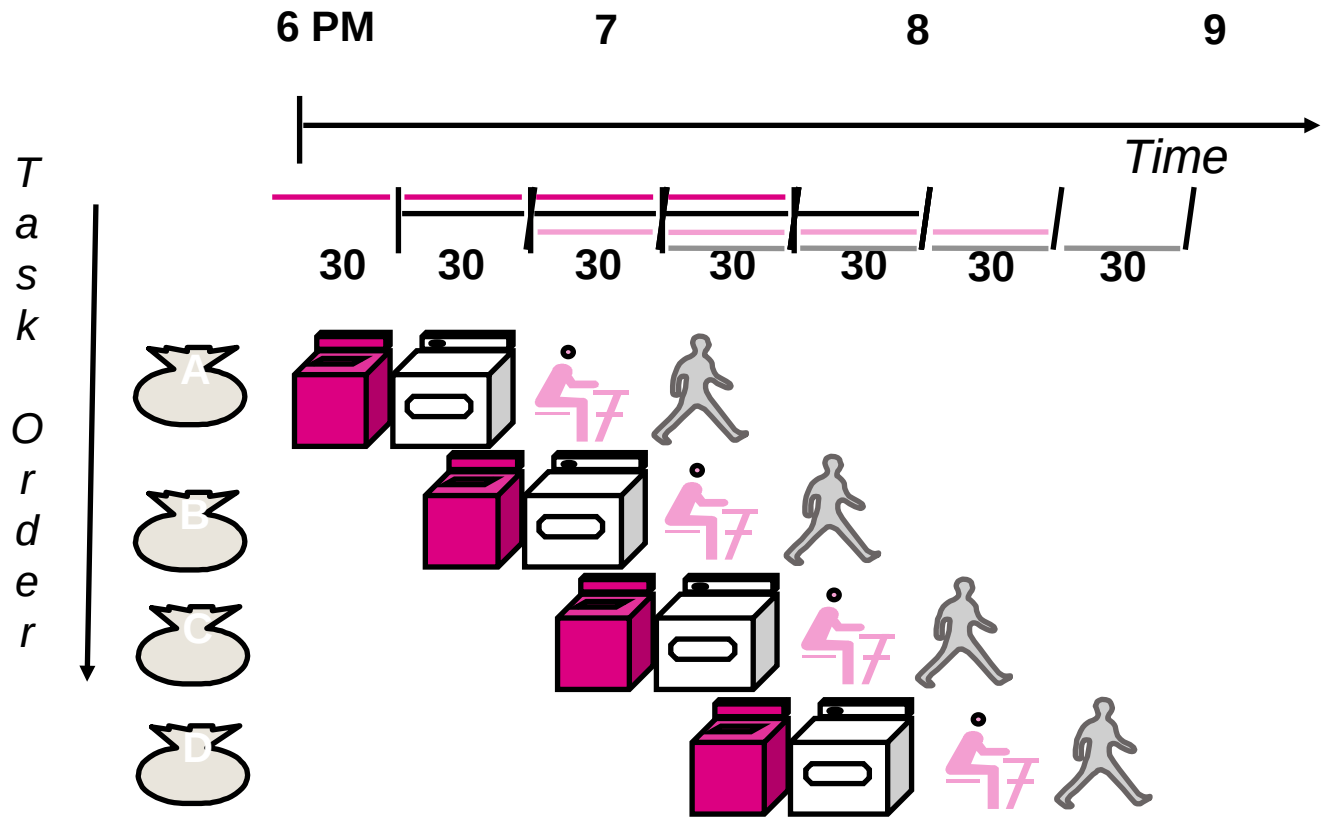
Baback Izadi
ECE Department
bai@engr.newpaltz.edu

Sequential Laundry



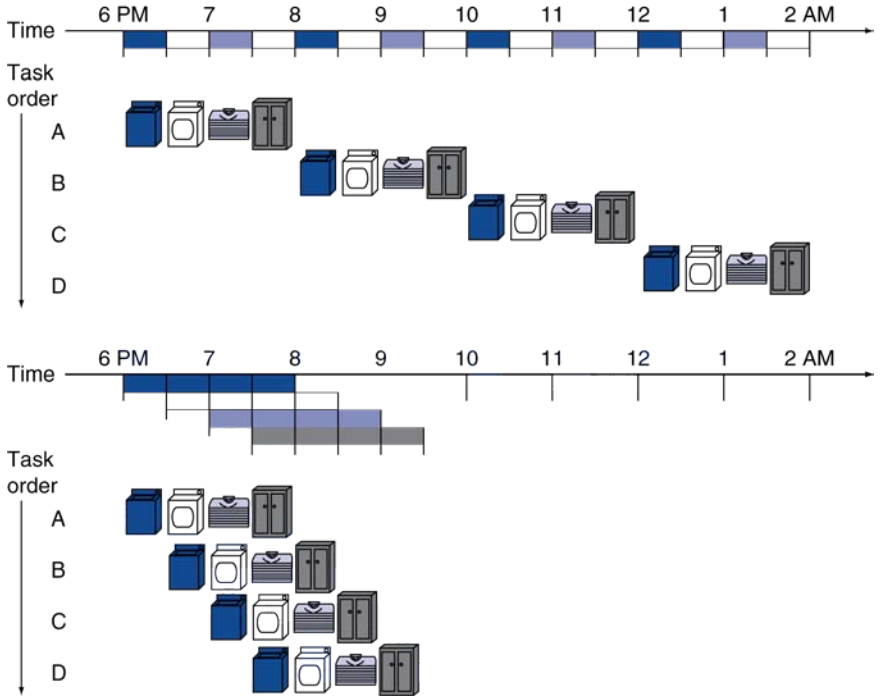
- Sequential laundry takes 8 hours for 4 loads

Pipelined Laundry



- Pipelined laundry: overlapping execution
  - Parallelism improves performance

Pipelining Analogy

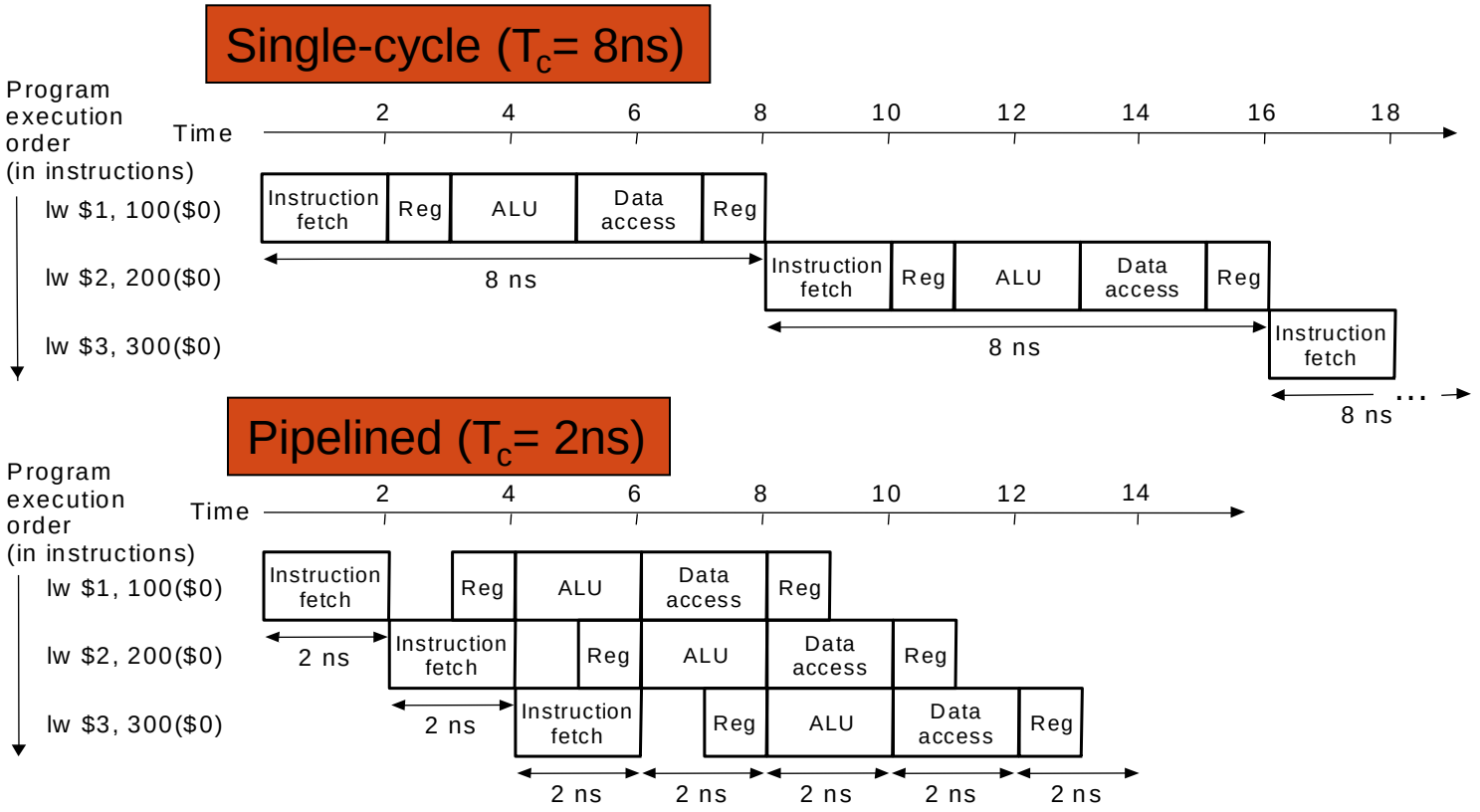


- Four loads:
  - Speedup = 8/3.5 = 2.3
- Non-stop:
  - Speedup = 2n/0.5n + 1.5 ≈ 4 = number of stages

- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously using different resources
- 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
- Stall for dependencies

Single Stage VS. Pipeline Performance

Instruction	Instr. Memory	Register Read	ALU Op.	Data Memory	Reg. Write	Total
R-format	2	1	2	0	1	6 ns
lw	2	1	2	2	1	8 ns
sw	2	1	2	2		7 ns
beq	2	1	2			5 ns



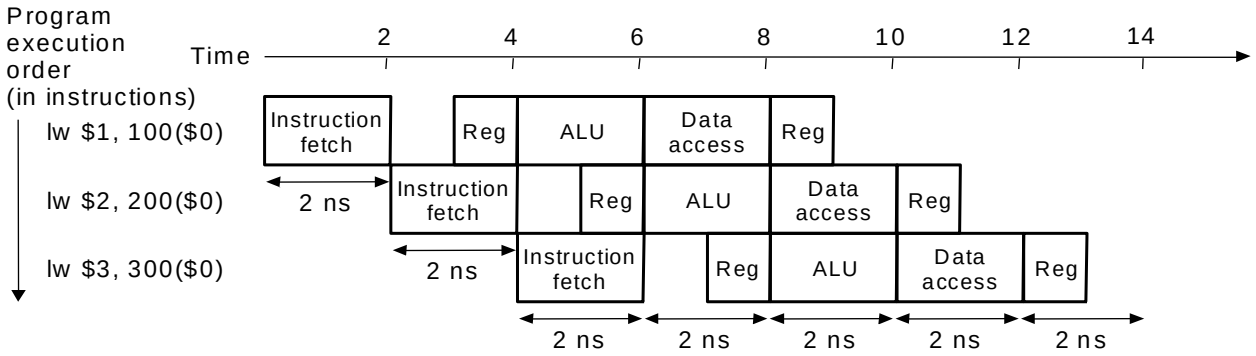
MIPS Pipeline

- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register

Pipelining

- What makes it easy in MIPS
  - All instructions are the same length
  - Just a few instruction formats
  - Memory operands appear only in loads and stores
- What makes it hard?
  - Structural hazards: suppose we had only one memory
  - Control hazards: need to worry about branch instructions
  - Data hazards: an instruction depends on a previous instruction
- We'll build a simple pipeline and look at these issues
- what makes it really hard for modern processors
  - Exception handling
  - Trying to improve performance with out-of-order execution

Pipeline Speedup



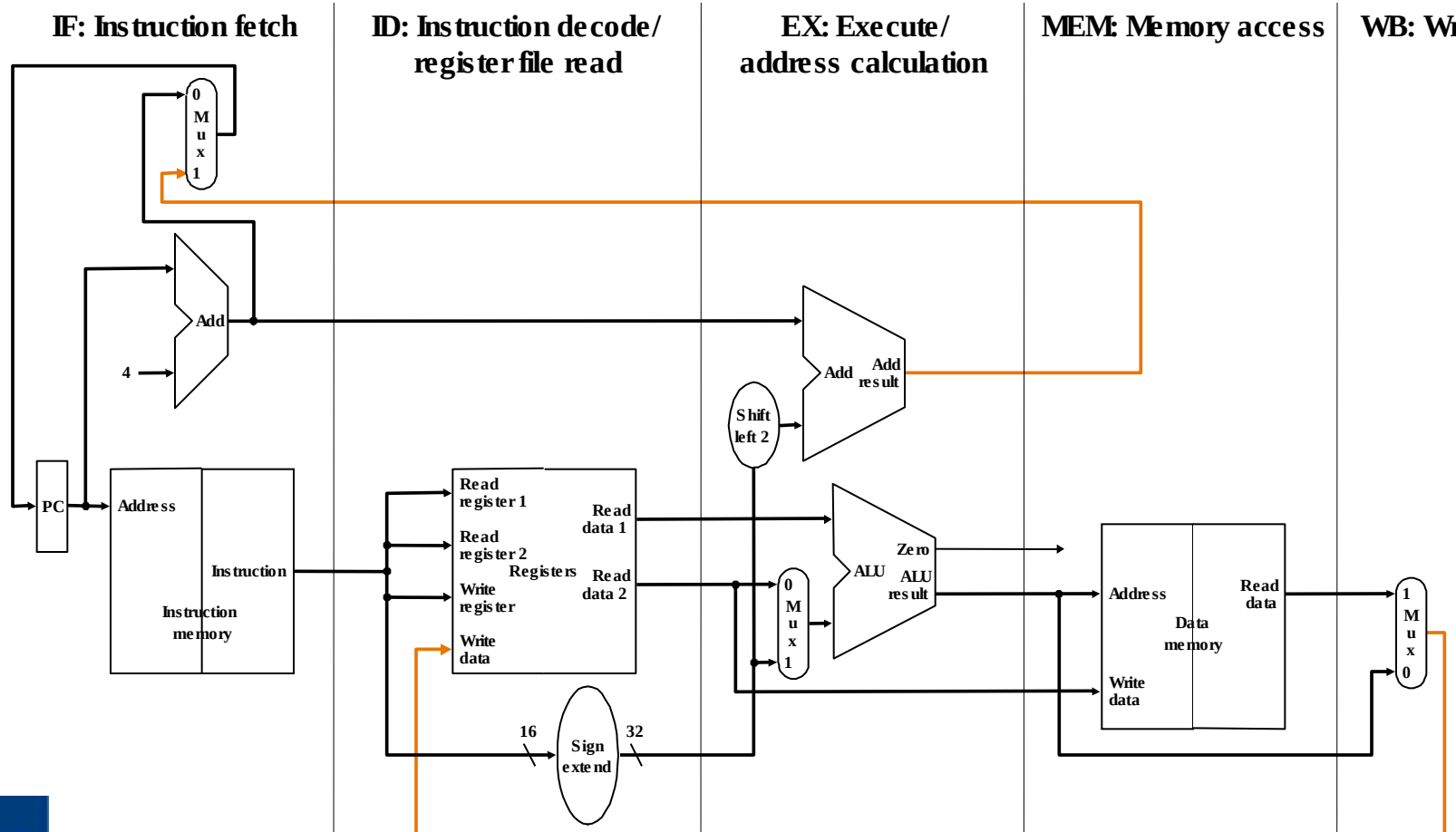
- If all stages are balanced
  - i.e., all take the same time
  - $$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$$
  - If not, speedup is less
- Speedup is due to increased throughput
  - Latency (time for each instruction) does not decrease

The Five Stages of Load

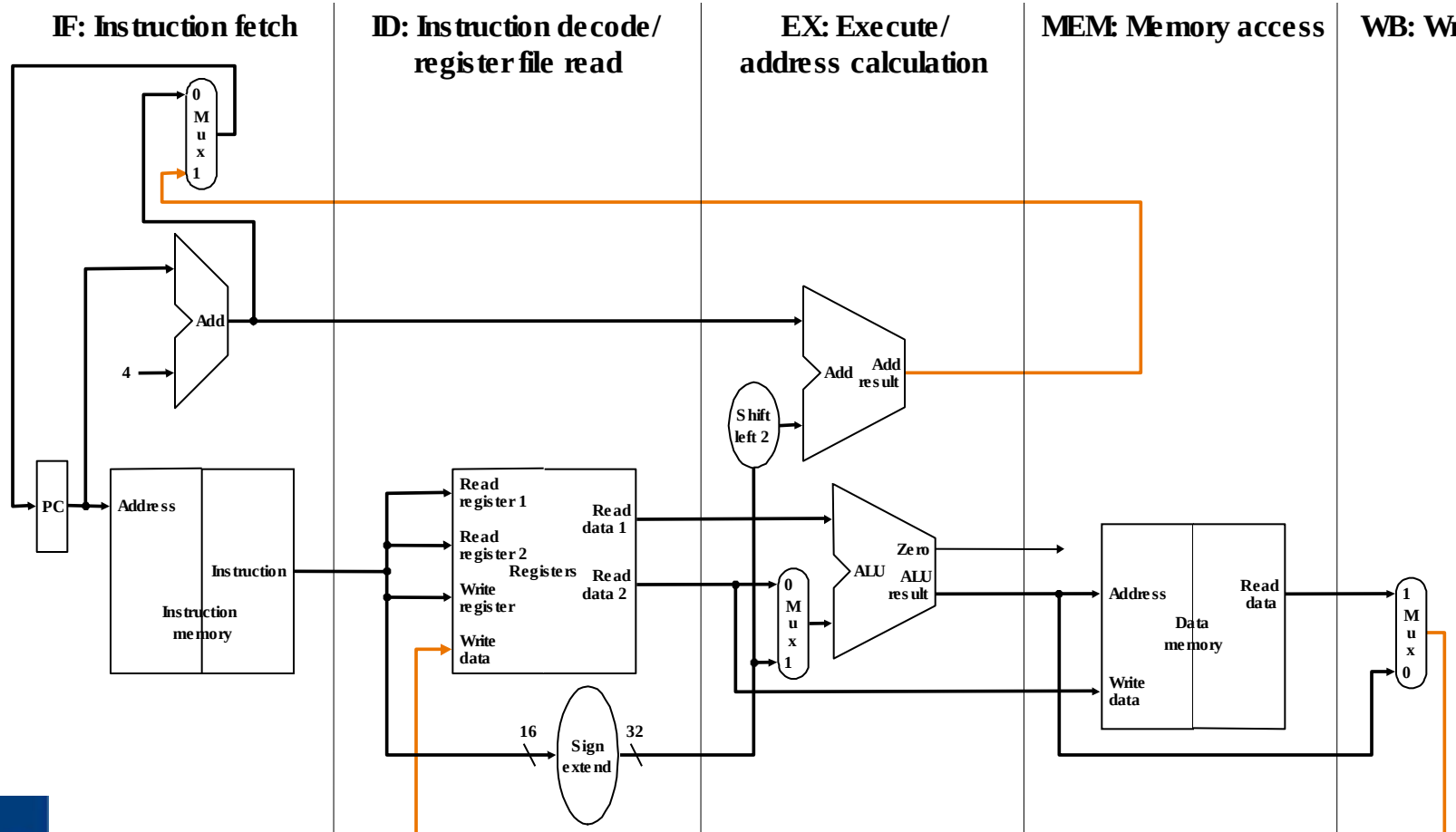
- *What do we need to add to actually split the datapath into stages?*

- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Read the data from the Data Memory
- Wr: Write the data back to the register file

Basic Idea


IF: Instruction fetch
ID: Instruction decode/ register file read
EX: Execute/ address calculation
MEM: Memory access
WB: Write back
0 M u x 1
Add
4
PC
Address
Instruction
Instruction memory
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
0 M u x 1
ALU
Zero
ALU result
Address
Data memory
Read data
Write data
1 M u x 0
16
Sign extend
32

Basic Idea

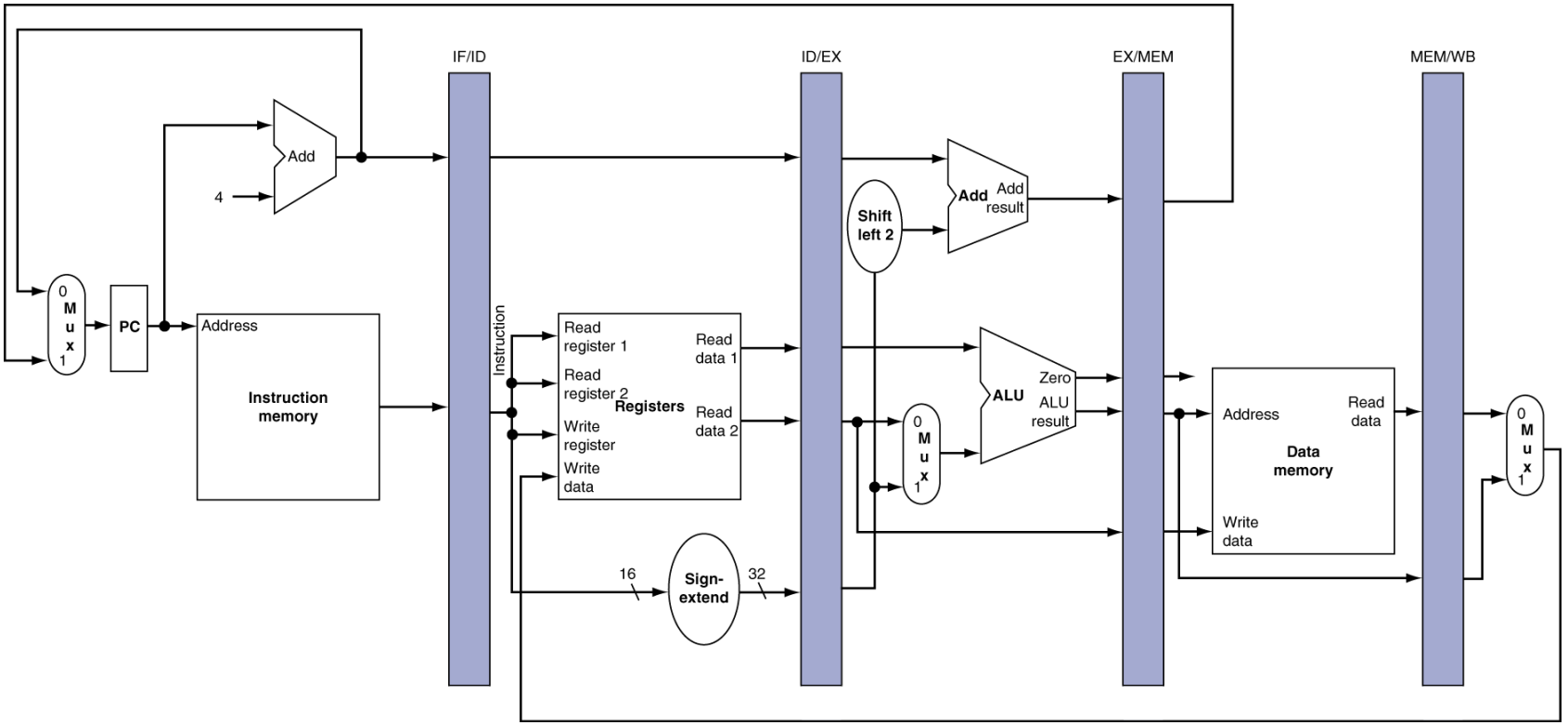

IF: Instruction fetch
ID: Instruction decode/ register file read
EX: Execute/ address calculation
MEM: Memory access
WB: Write back
0 M u x 1
Add
4
PC
Address
Instruction
Instruction memory
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
0 M u x 1
ALU
Zero
ALU result
Address
Data memory
Read data
Write data
1 M u x 0
16
Sign extend
32

Pipelining and ISA Design

- MIPS 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
  - Alignment of memory operands
    - Memory access takes only one cycle

Pipeline registers

- Need registers between stages
  - To hold information produced in previous cycle

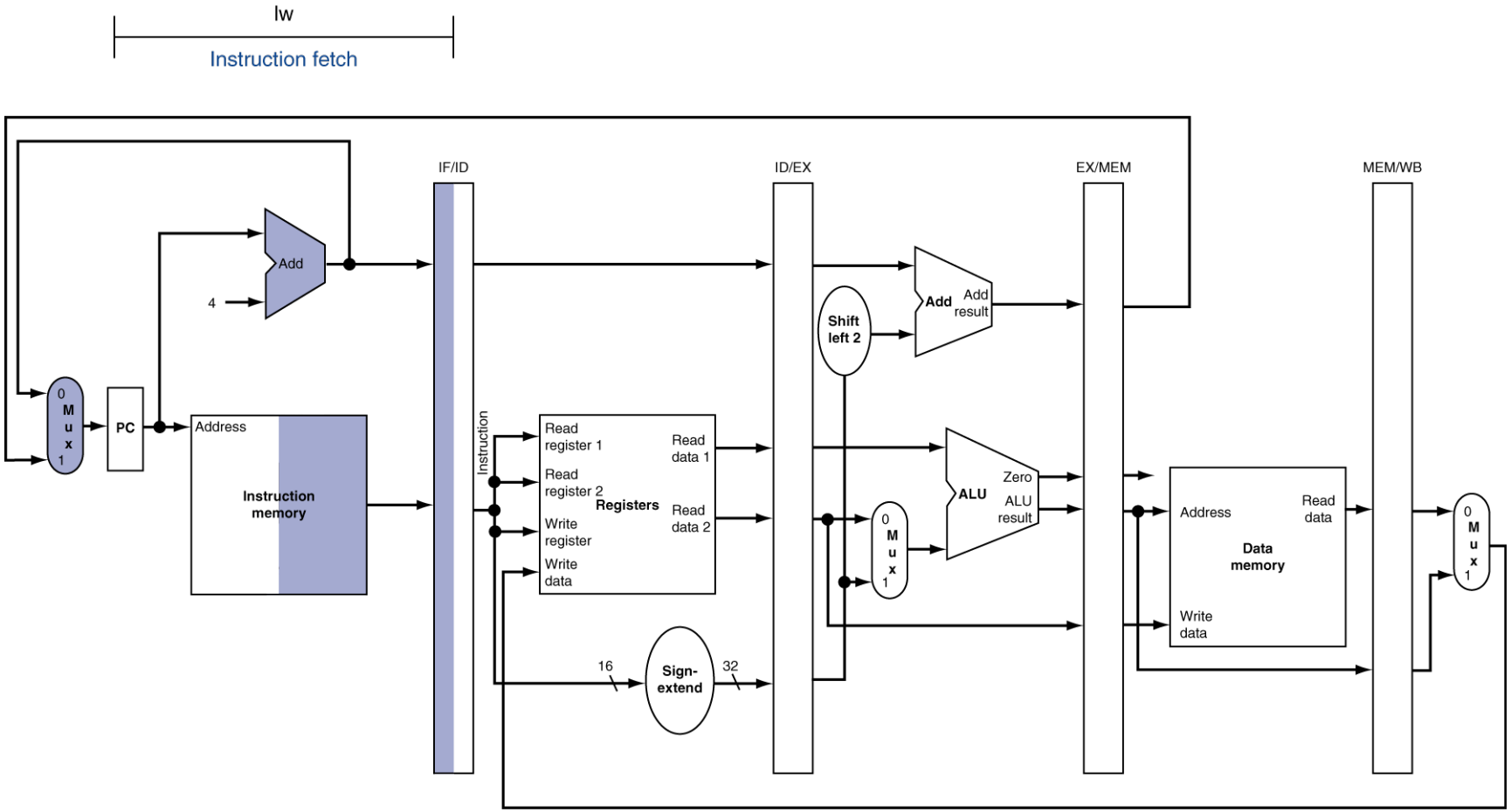


- *Can you find a problem?*
- *What instructions can we execute to manifest the problem?*

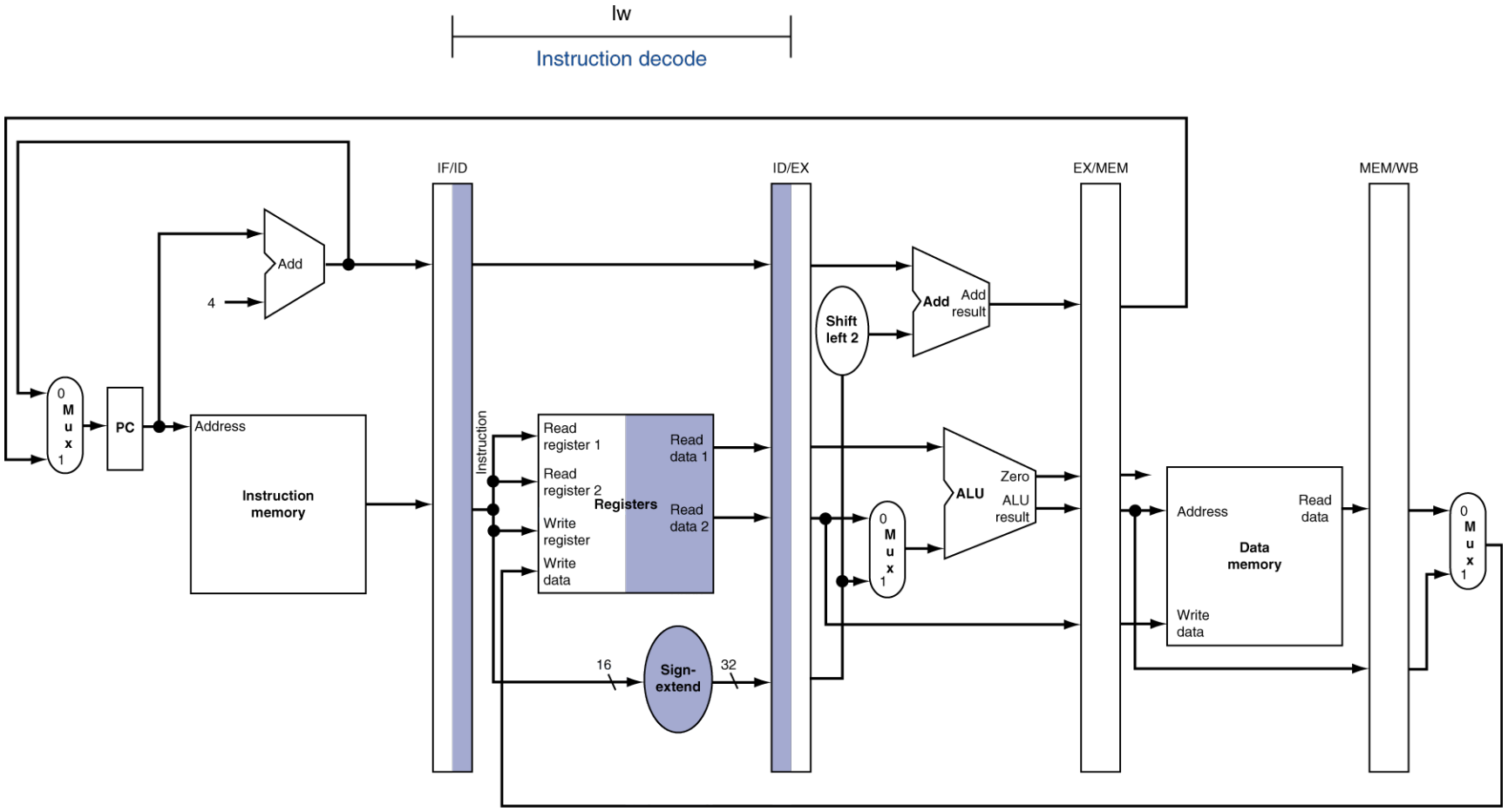
Pipeline Operation

- Cycle-by-cycle flow of instructions through the pipelined datapath
  - "Single-clock-cycle" pipeline diagram
    - Shows pipeline usage in a single cycle
    - Highlight resources used
  - c.f. "multi-clock-cycle" diagram
    - Graph of operation over time
- We'll look at "single-clock-cycle" diagrams for load & store

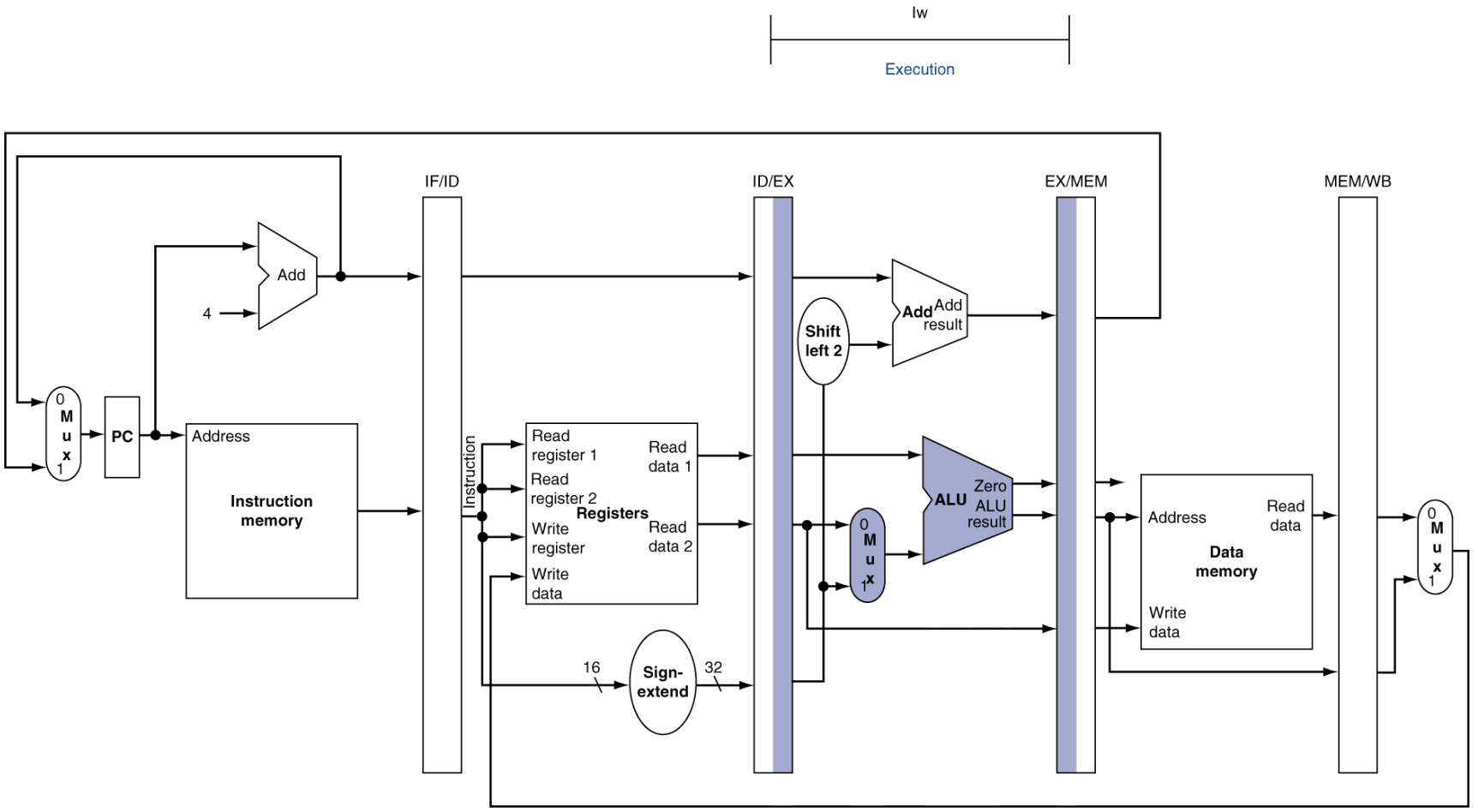
IF for Load, Store, ...



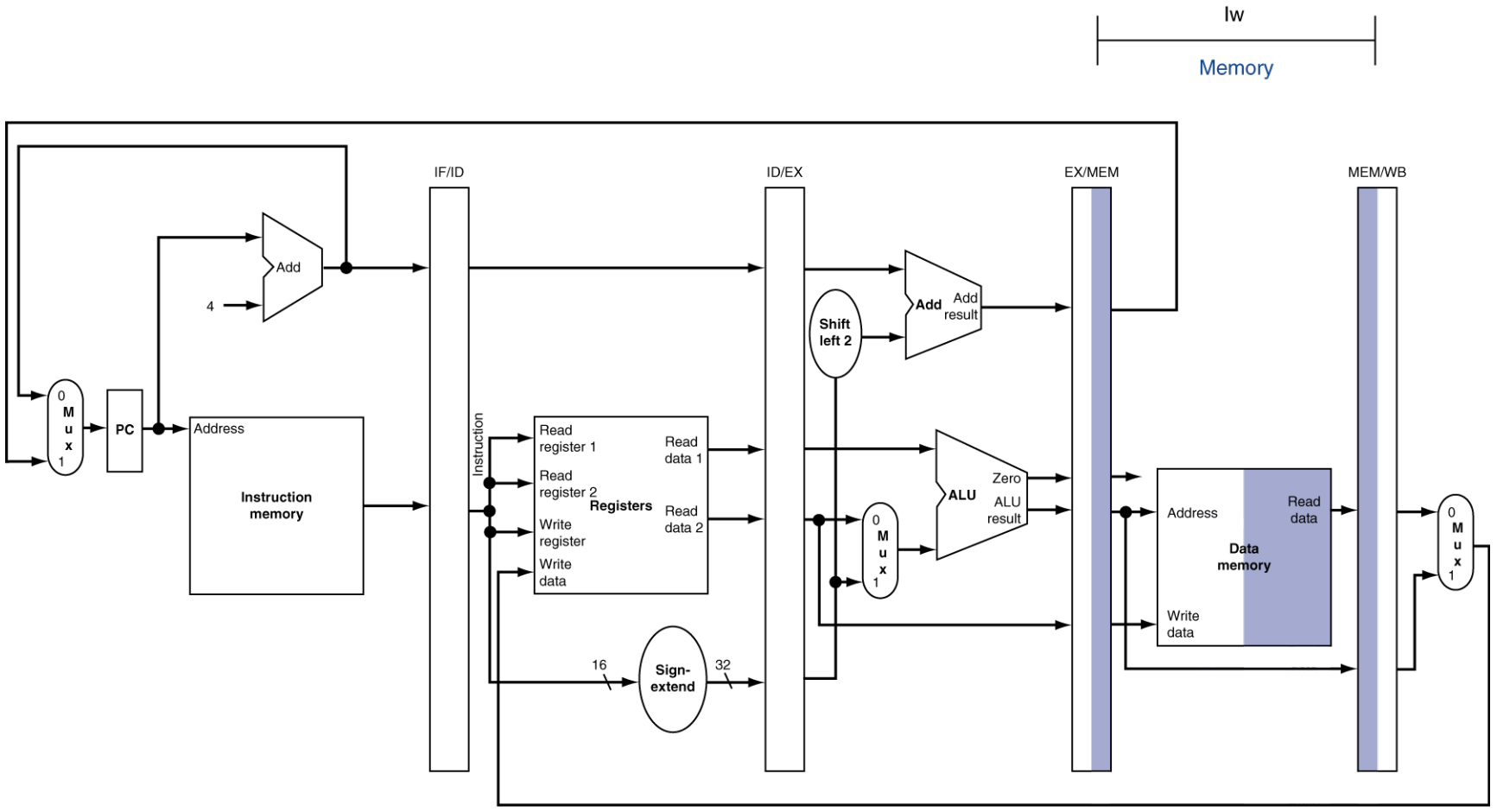
ID for Load, Store, ...



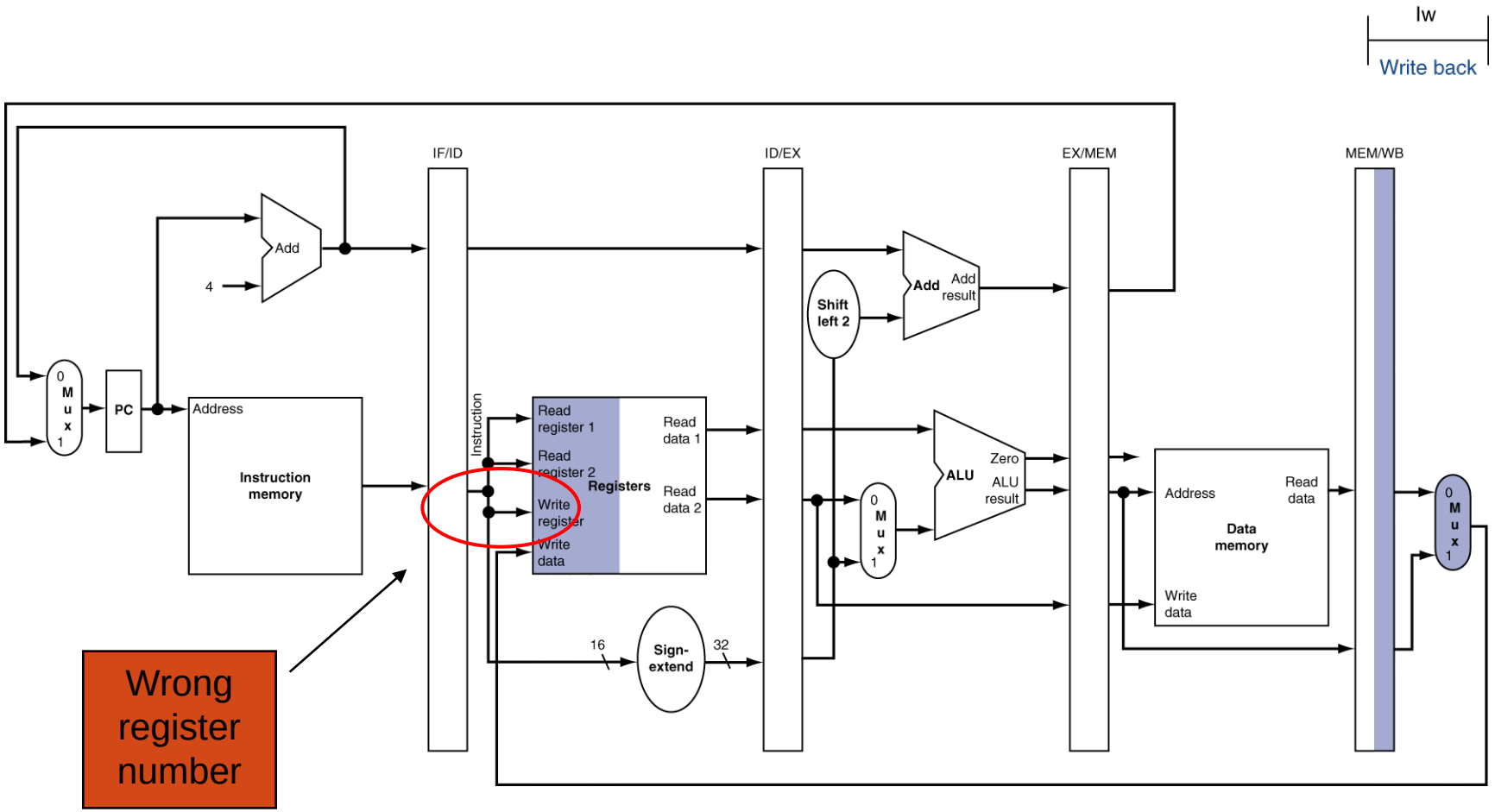
EX for Load


lw
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M
u
x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
16
Sign-extend
32

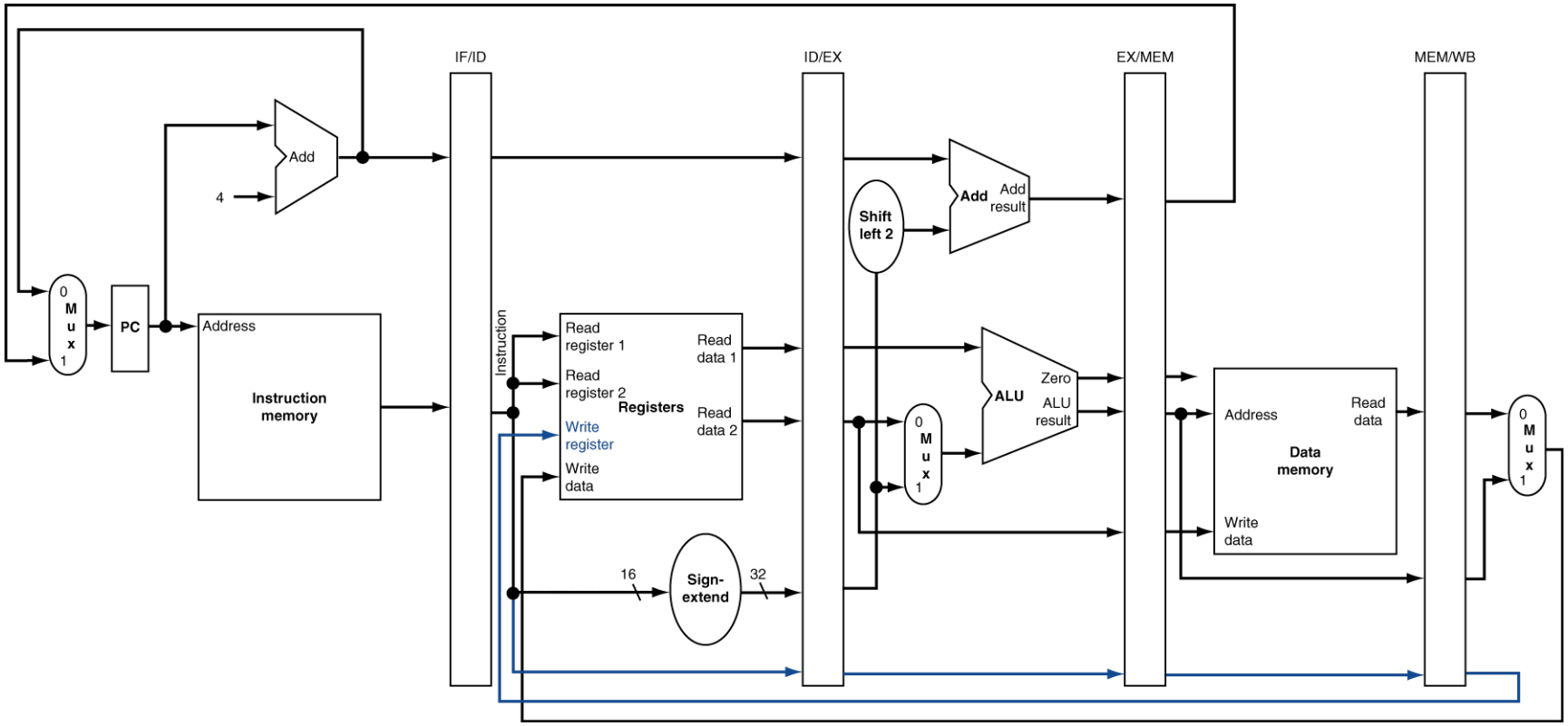
MEM for Load


lw
Memory
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M u x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Data memory
Read data
Write data

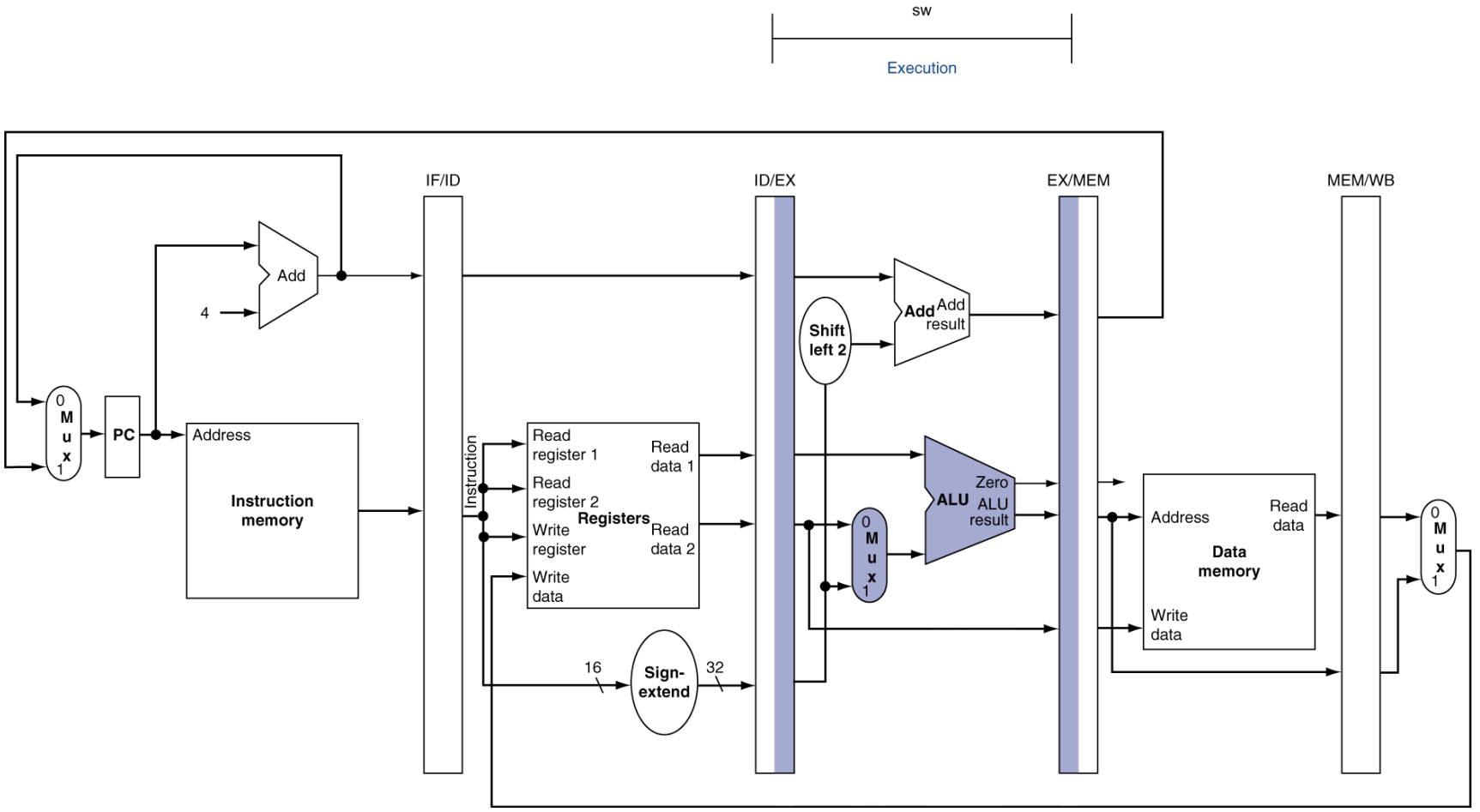
WB for Load


lw
Write back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M u x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
16
Sign-extend
32
Wrong register number

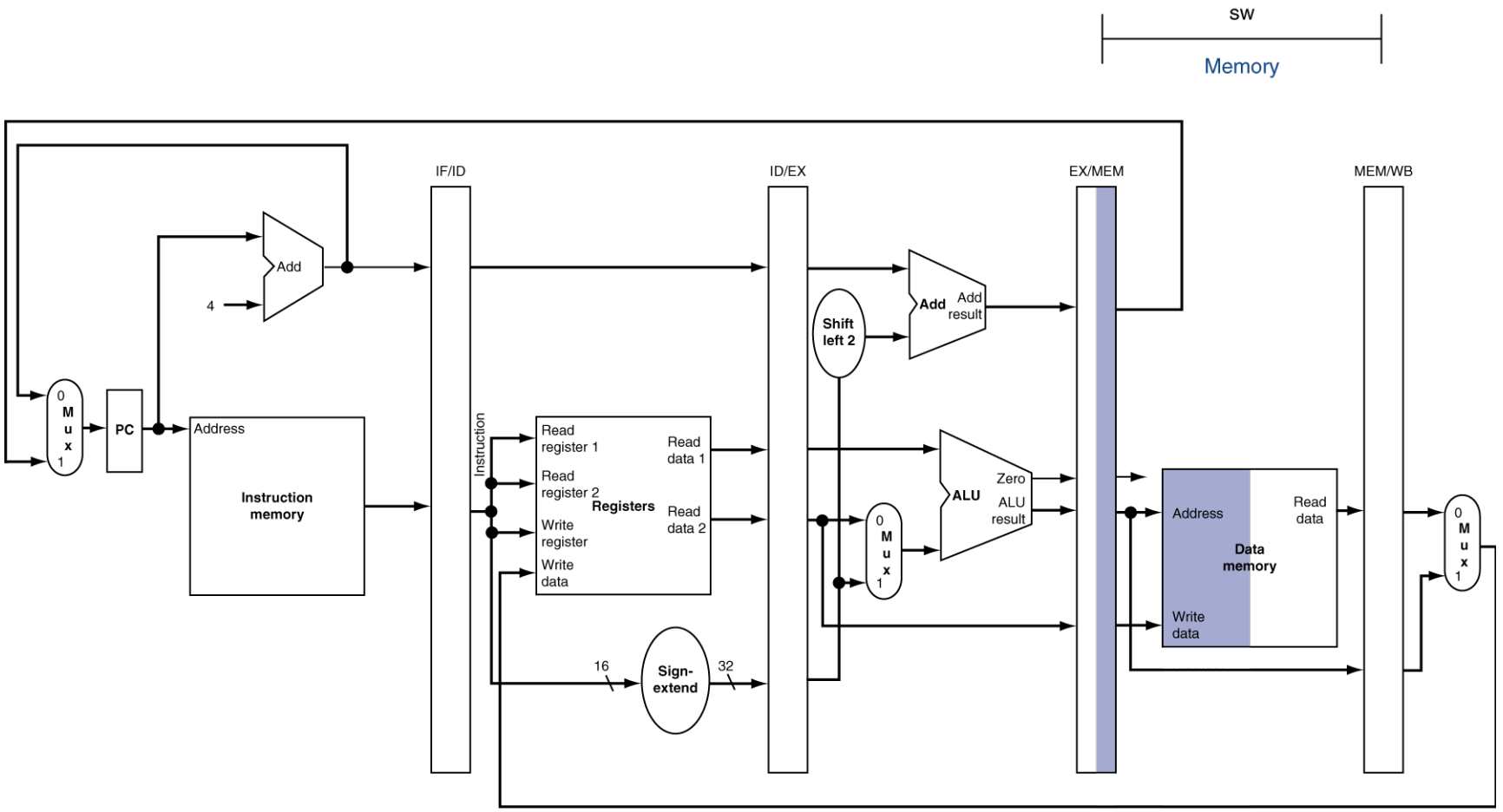
Corrected Datapath for Load



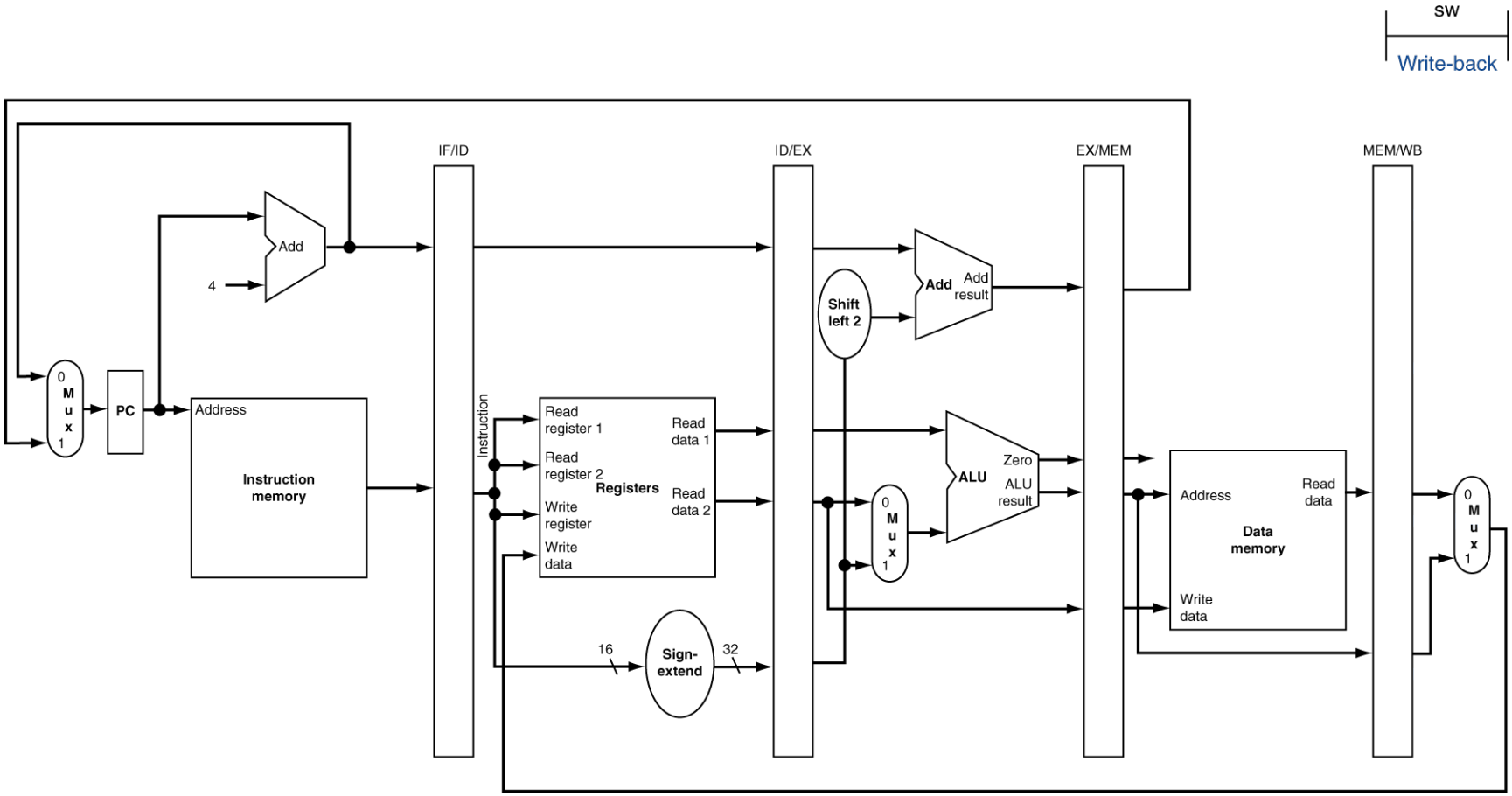
EX for Store


sw
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M
u
x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
16
Sign-extend
32

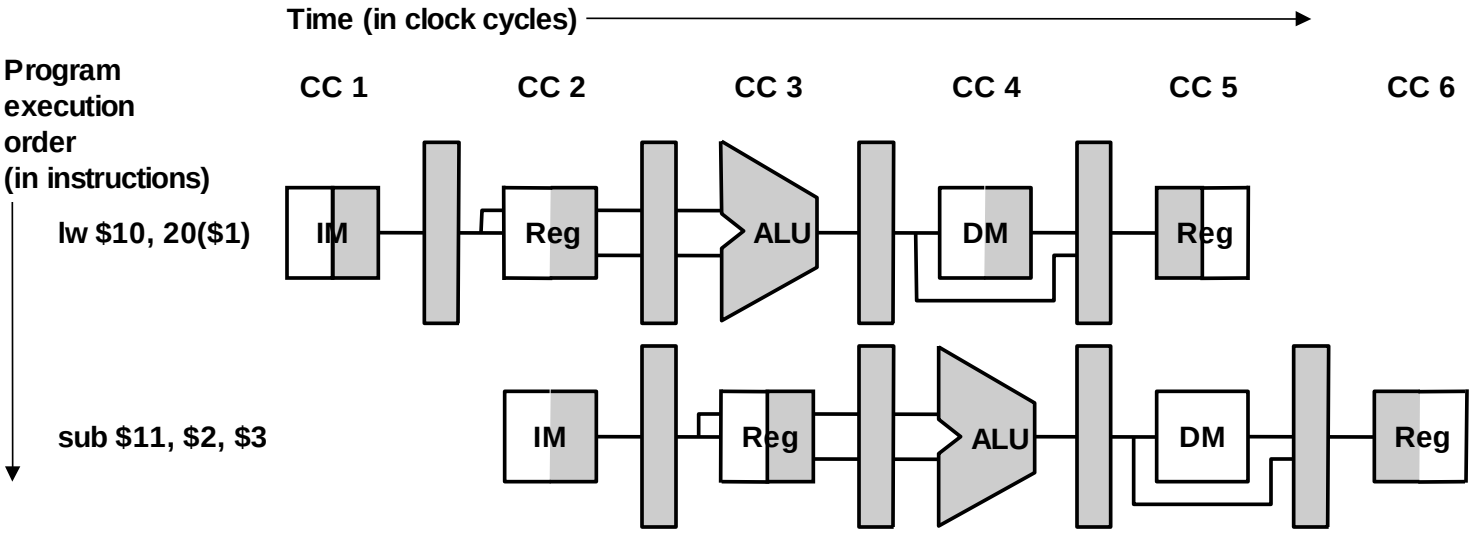
MEM for Store



WB for Store


SW
Write-back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M
u
x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data

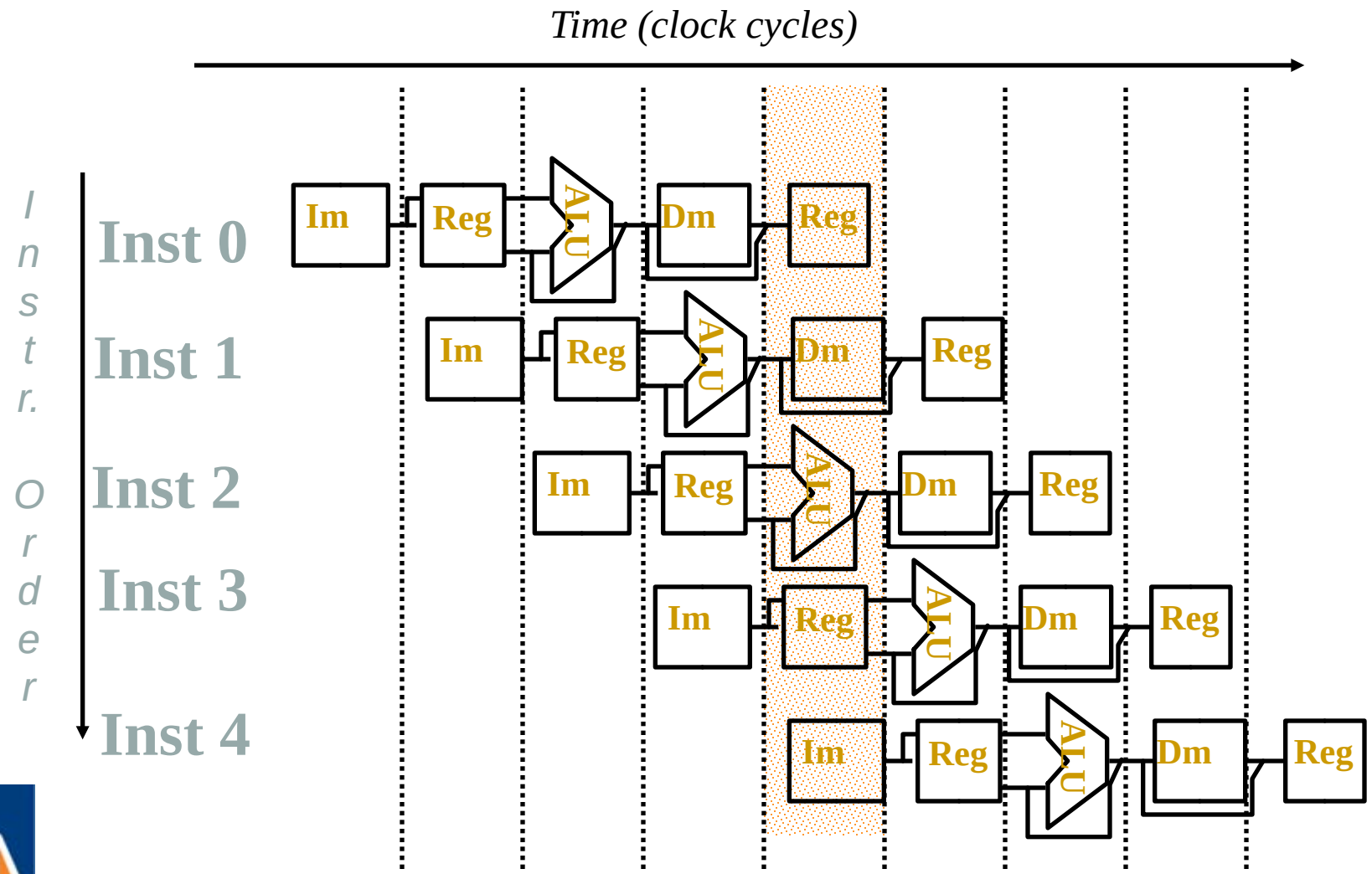
Graphically Representing Pipelines



Can help with answering questions like:

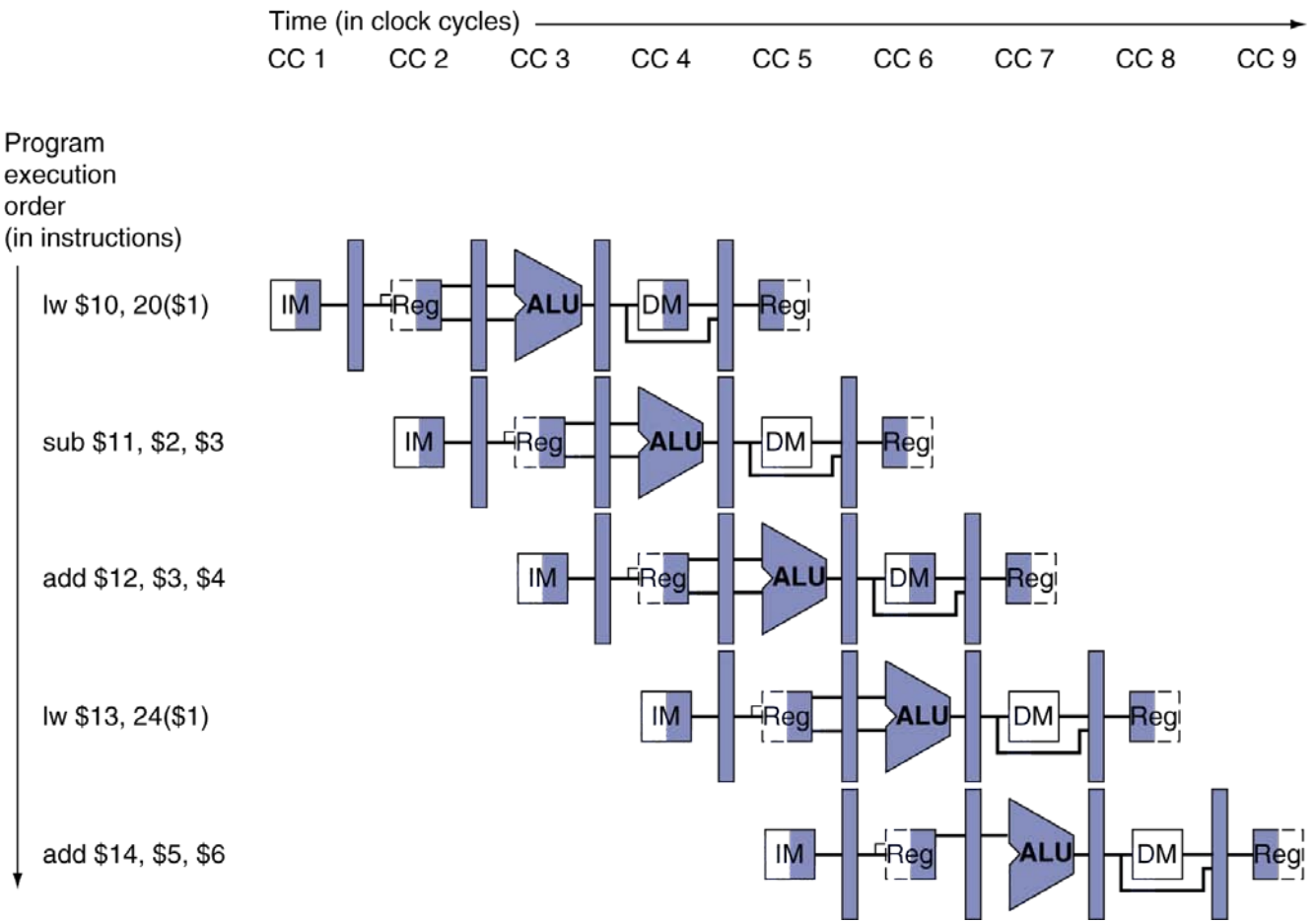
- How many cycles does it take to execute this code?
- What is the ALU doing during cycle 4?
- Use this representation to help understand datapaths

Why Pipeline?



Multi-Cycle Pipeline Diagram

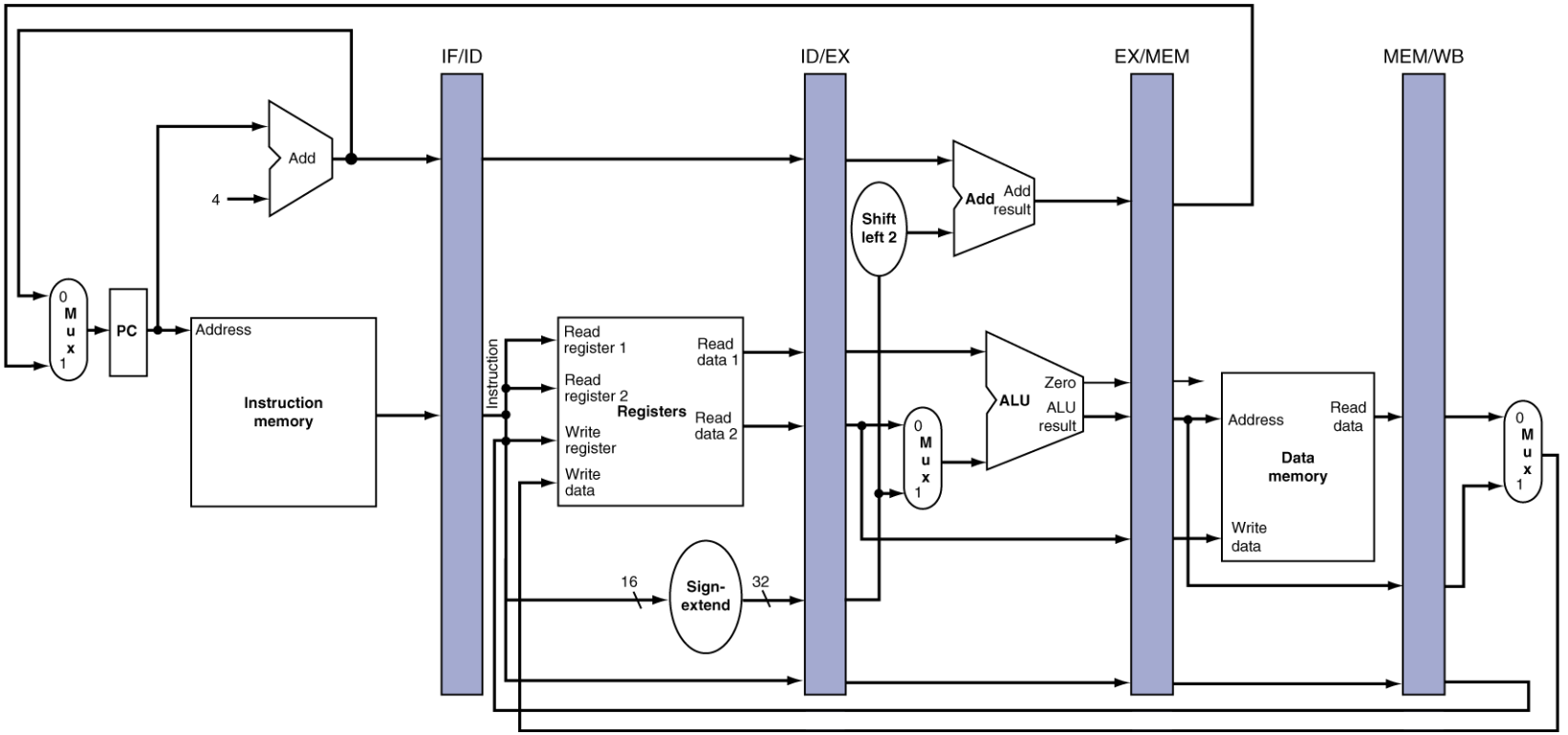
- Form showing resource usage



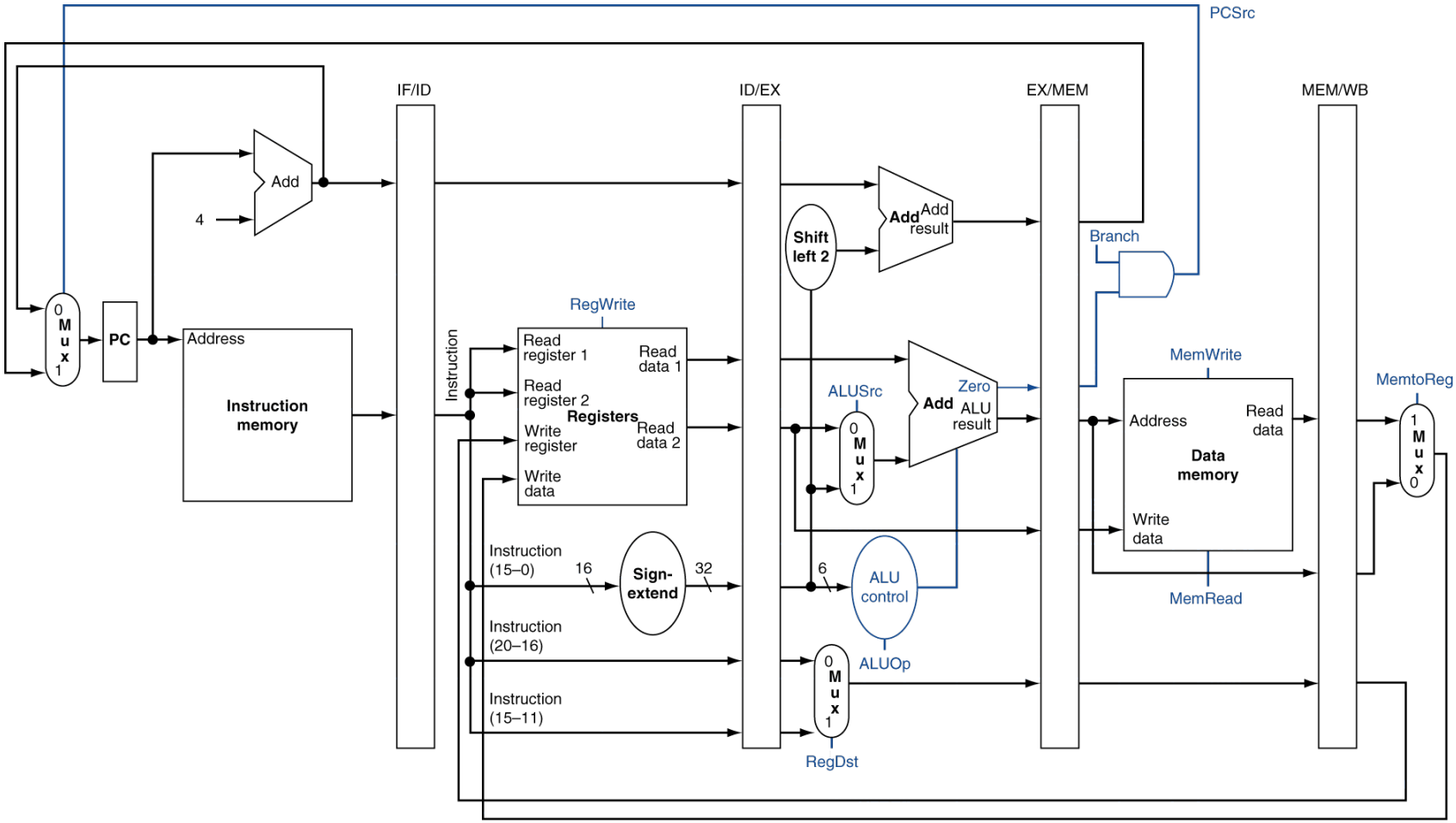
Single-Cycle Pipeline Diagram

- State of pipeline in a given cycle

add $14, $5, $6	lw $13, 24 ($1)	add $12, $3, $4	sub $11, $2, $3	lw $10, 20($1)
Instruction fetch	Instruction decode	Execution	Memory	Write-back



Pipelined Control (Simplified)



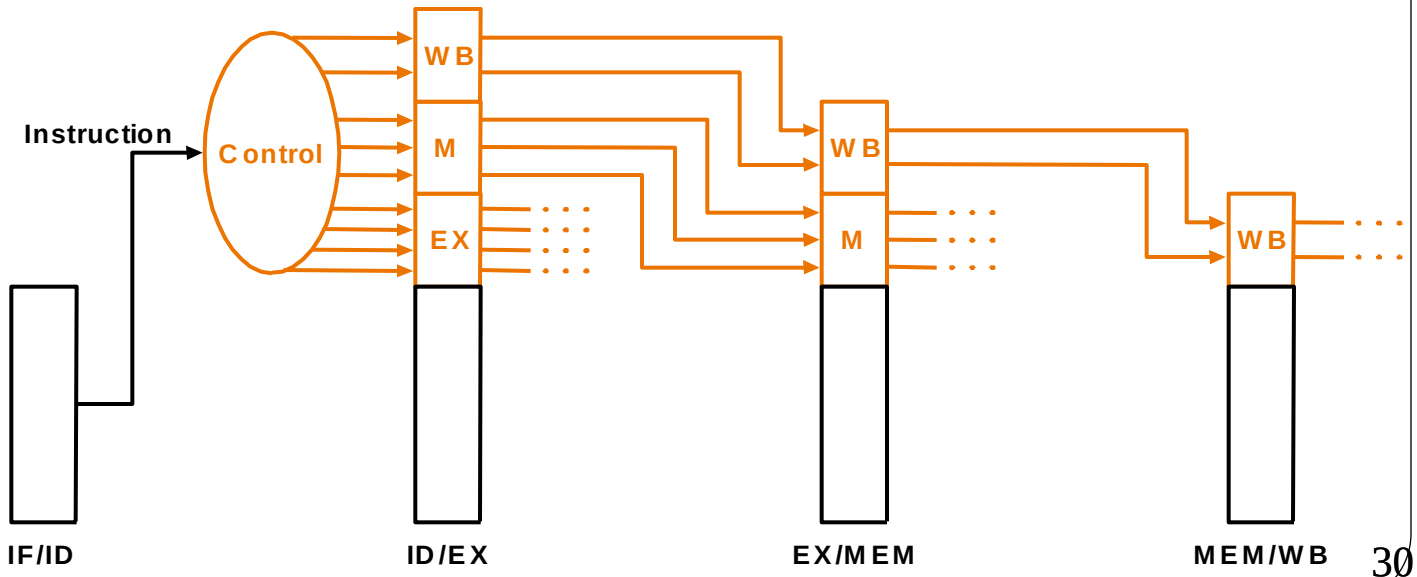
Pipeline Control

- We have 5 stages.
- What needs to be controlled in each stage?
  - Instruction Fetch and PC Increment
  - Instruction Decode / Register Fetch
  - Execution
  - Memory Stage
  - Write Back
- How would control be handled in an automobile plant?
  - A fancy control center telling everyone what to do?
  - Should we use a finite state machine?

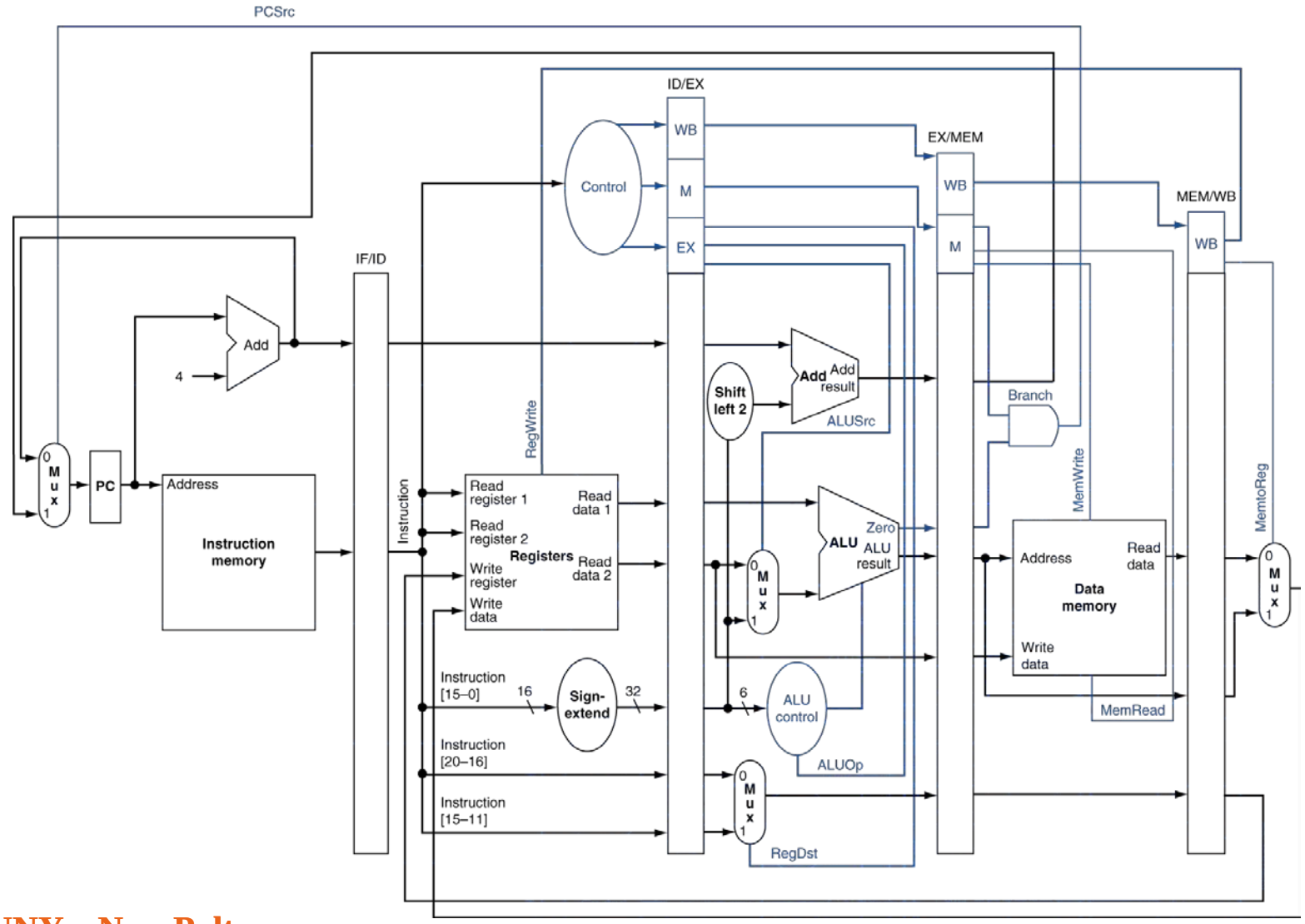
Pipeline Control

- Control signals derived from instruction
  - As in single-cycle implementation
- Pass control signals along just like the data

Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X



Pipelined Control



Designing a Pipelined Processor

- Go back and examine your datapath and control diagram
- Associated resources with states
- Ensure that flows do not conflict, or figure out how to resolve
- Assert control in appropriate stage

Pipelining Troubles?

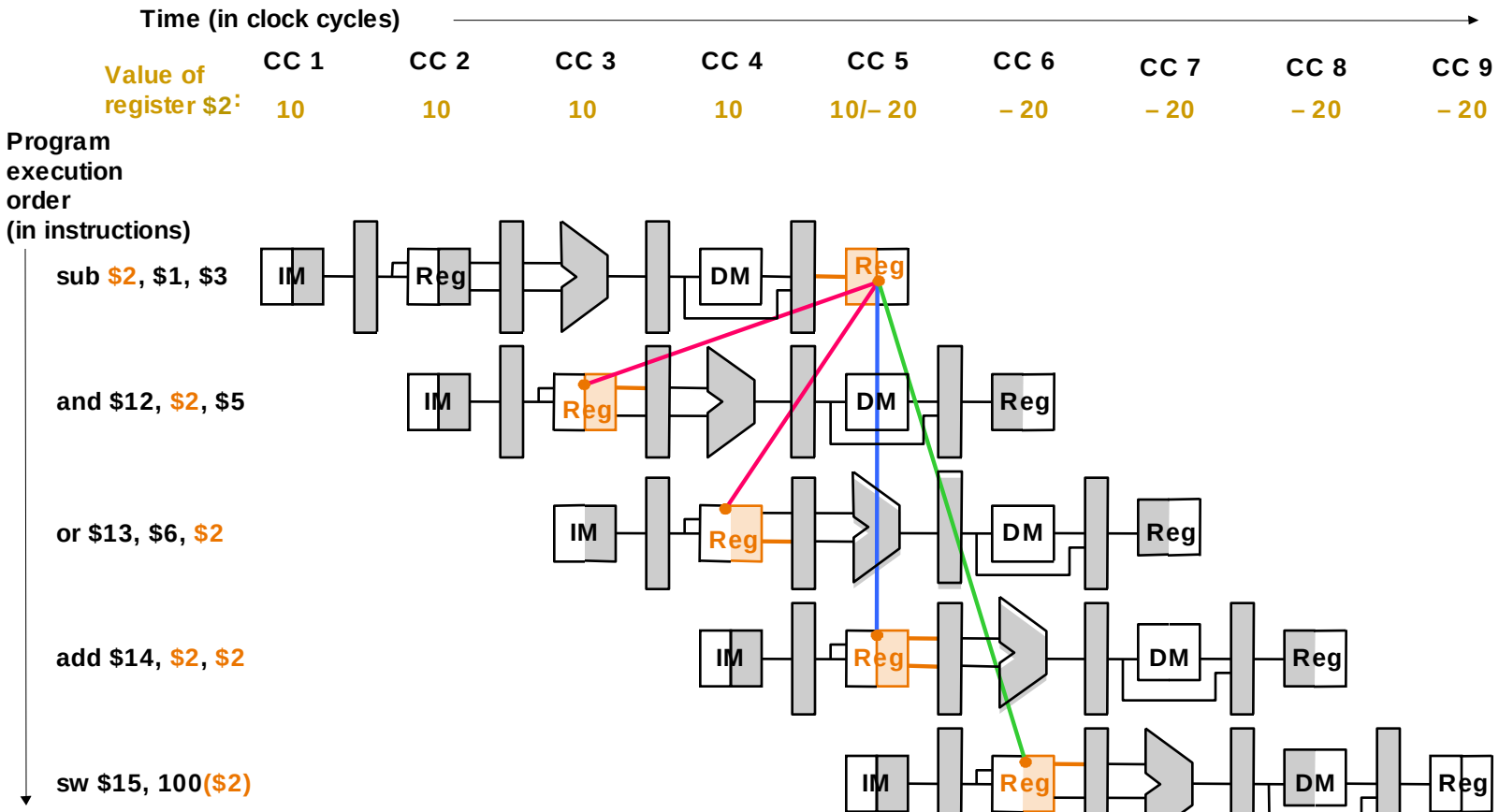
- Pipeline Hazards
- Situations that prevent starting the next instruction in the next cycle
  - Structural hazards: attempt to use the same resource two different ways at the same time.
  - Data hazards: attempt to use item before it is ready.
    - Instruction depends on result of prior instruction still in the pipeline.
  - Control hazards: attempt to make a decision before condition is evaluated.
    - Branch instructions
- Can always resolve hazards by waiting.
  - Pipeline control must detect the hazard.
  - Take action (or delay action) to resolve hazards.

Structure Hazards

- Conflict for use of a resource
- In MIPS pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline "bubble"
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches

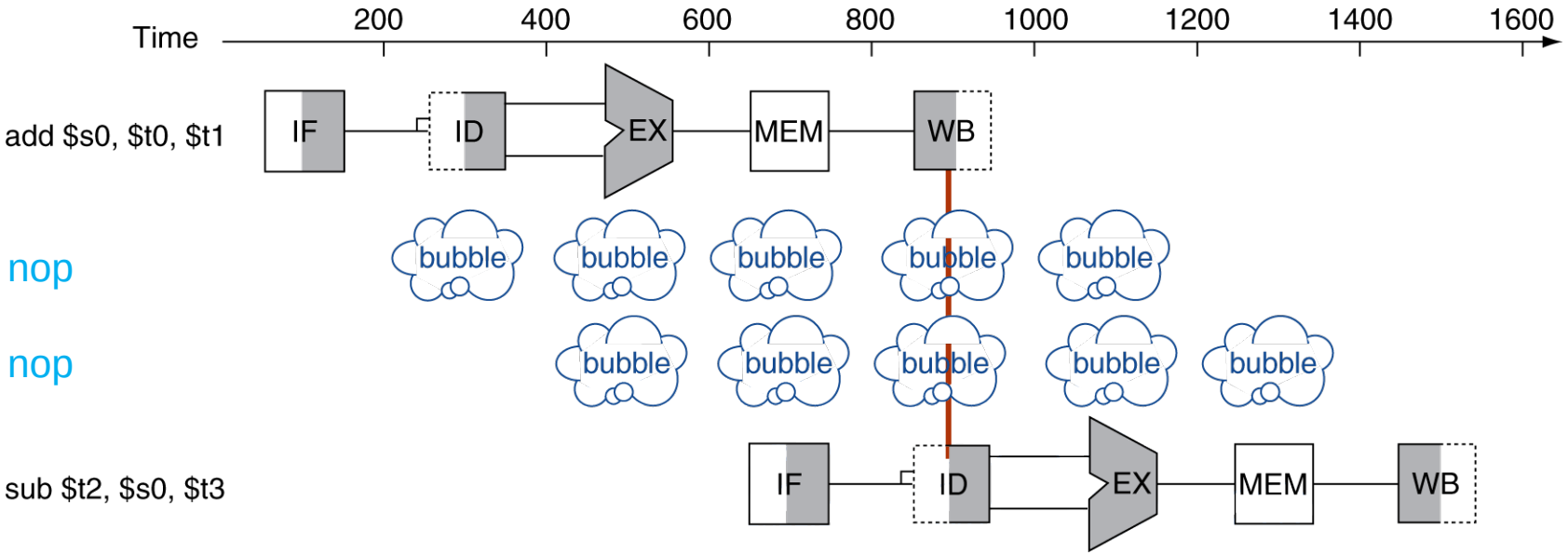
Data Hazards

- Problem with starting next instruction before first is finished
- Dependencies that "go backward in time" are data hazards



Data Hazard Solution

- An instruction depends on completion of data access by a previous instruction
  - `add  $s0, $t0, $t1`
    `sub  $t2, $s0, $t3`



Software Solution

- Have compiler guarantee no hazards
- Where should compiler insert "*nop*" instructions?

```
sub   $2, $1, $3
and   $12, $2, $5
or    $13, $6, $2
add   $14, $2, $2
sw    $15, 100($2)
```

- Problem:
  - It happens too often to rely on compiler
  - It really slows us down!

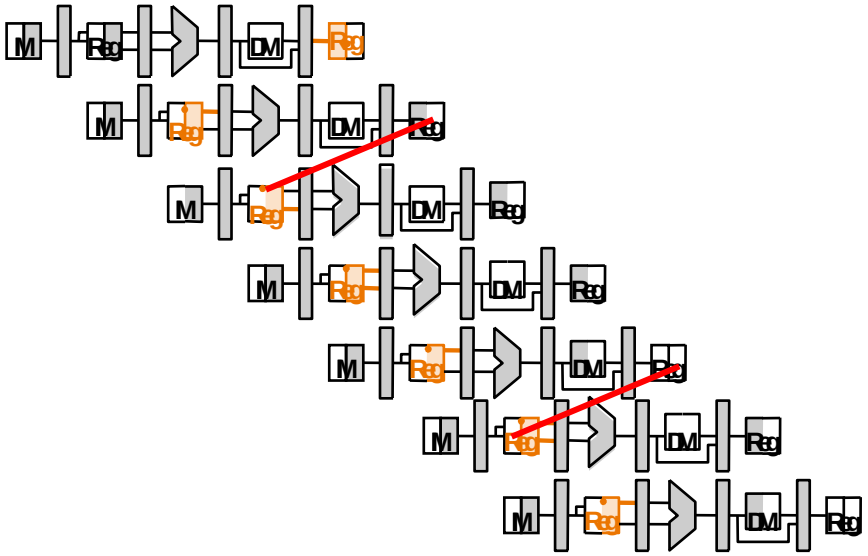
Code Scheduling to Avoid Stalls

- C code for `A = B + E; C = B + F;`

stall

stall

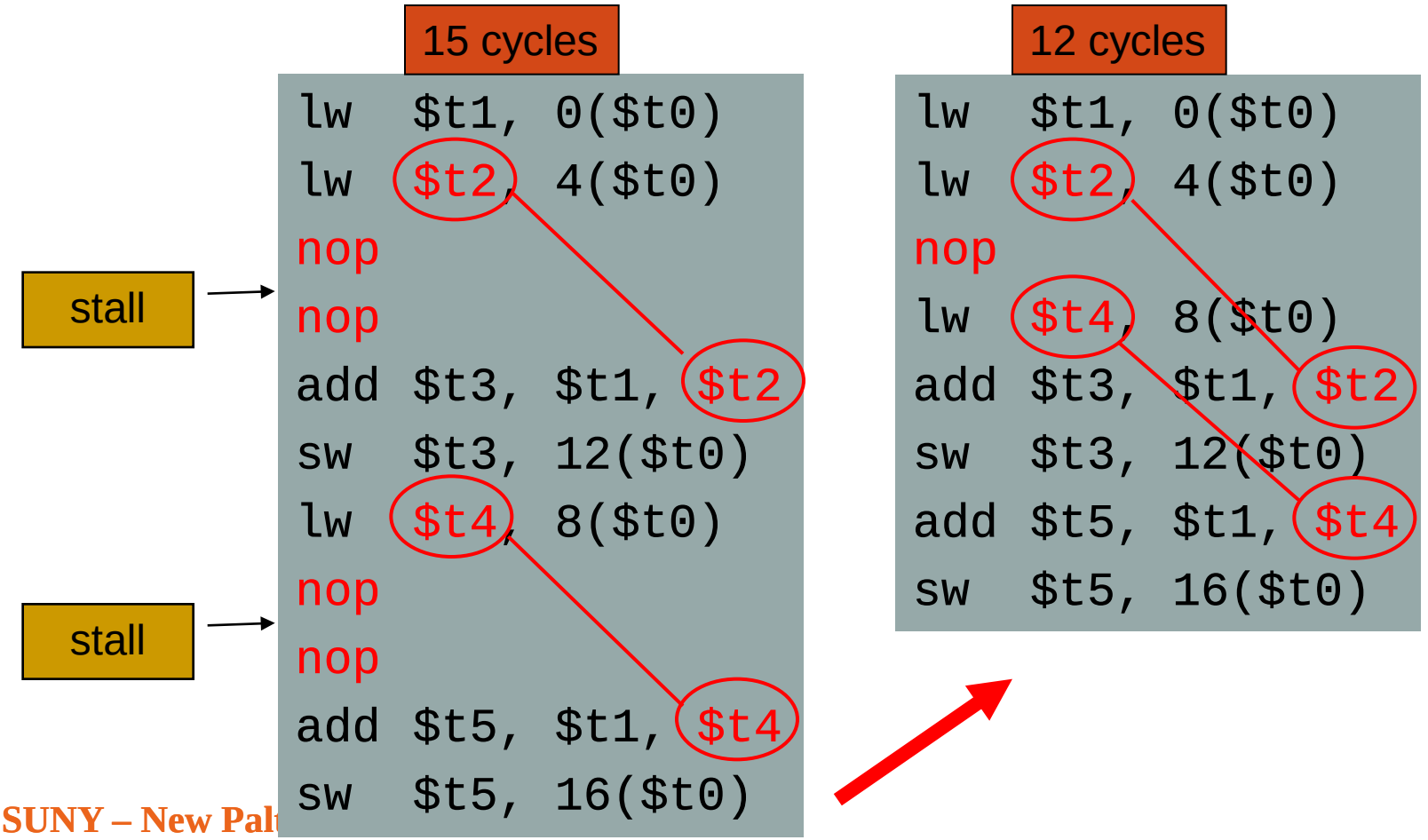
```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
add $t3, $t1, $t2
sw  $t3, 12($t0)
lw  $t4, 8($t0)
add $t5, $t1, $t4
sw  $t5, 16($t0)
```



11 cycles not counting the dependencies

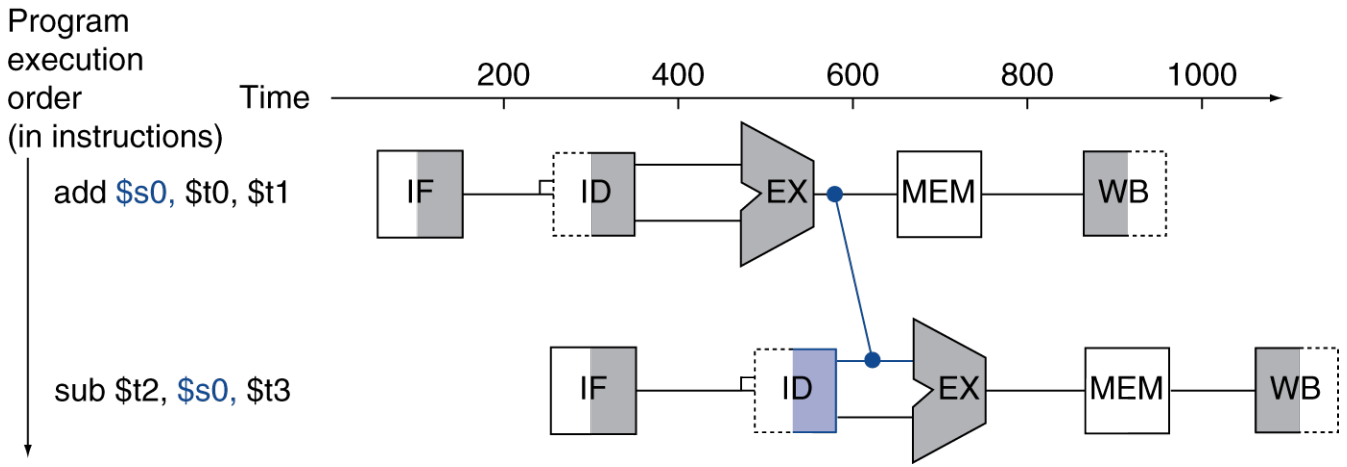
Code Scheduling to Avoid Stalls

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



Forwarding (aka Bypassing)

- Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath



Data Hazards in ALU Instructions

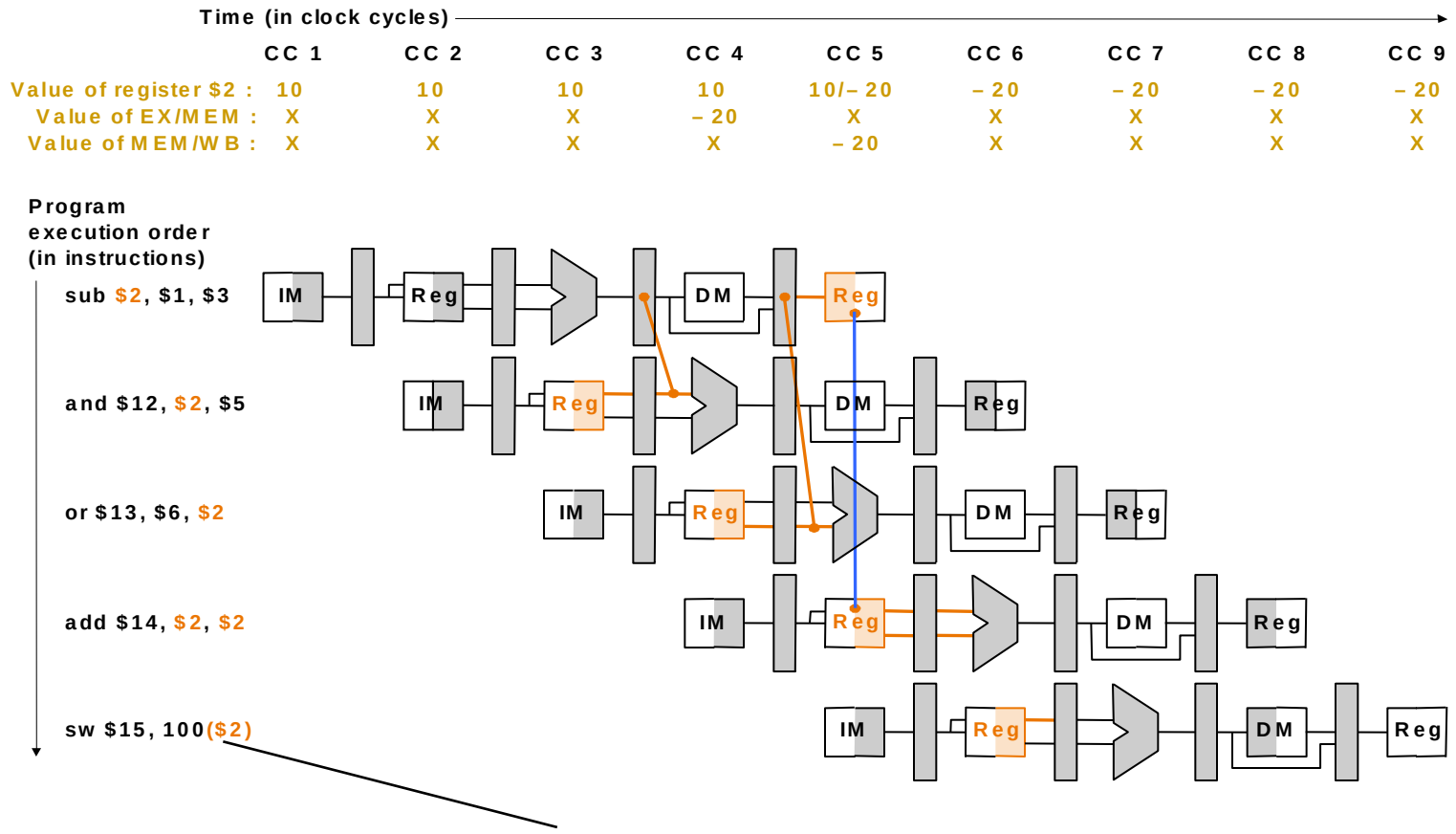
- Consider this sequence:

```
sub $2, $1,$3
and $12,$2,$5
or  $13,$6,$2
add $14,$2,$2
sw  $15,100($2)
```

- We can resolve hazards with forwarding
  - How do we detect when to forward?

Data Hazard Solution: Forwarding

- Use temporary results (ALU forwarding), don't wait for them to be written
- Also, write register file during 1st half of clock and read during 2nd half

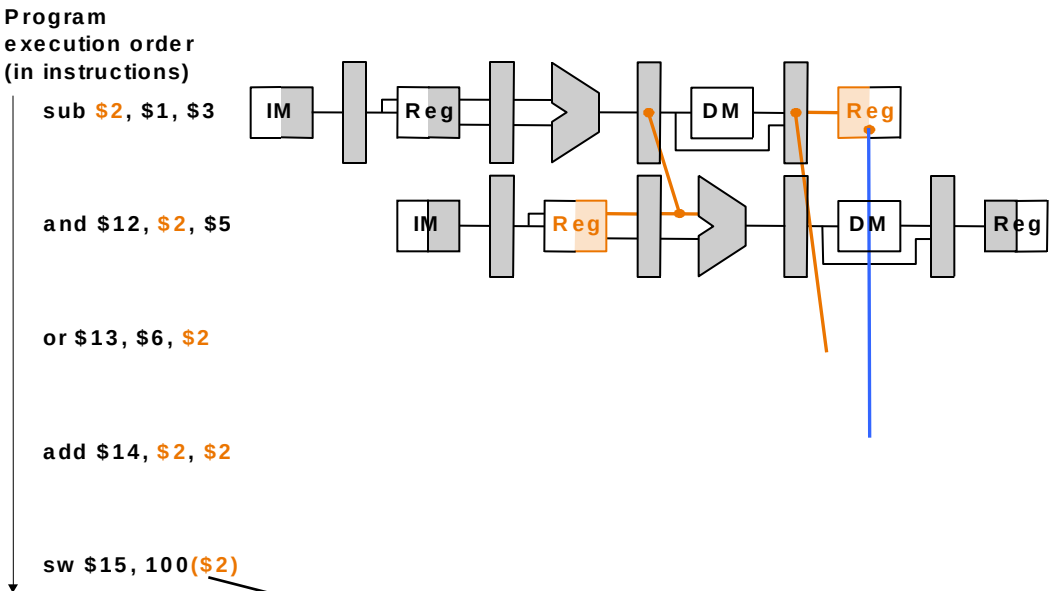


Data Hazard Solution: Forwarding

Time (in clock cycles) →

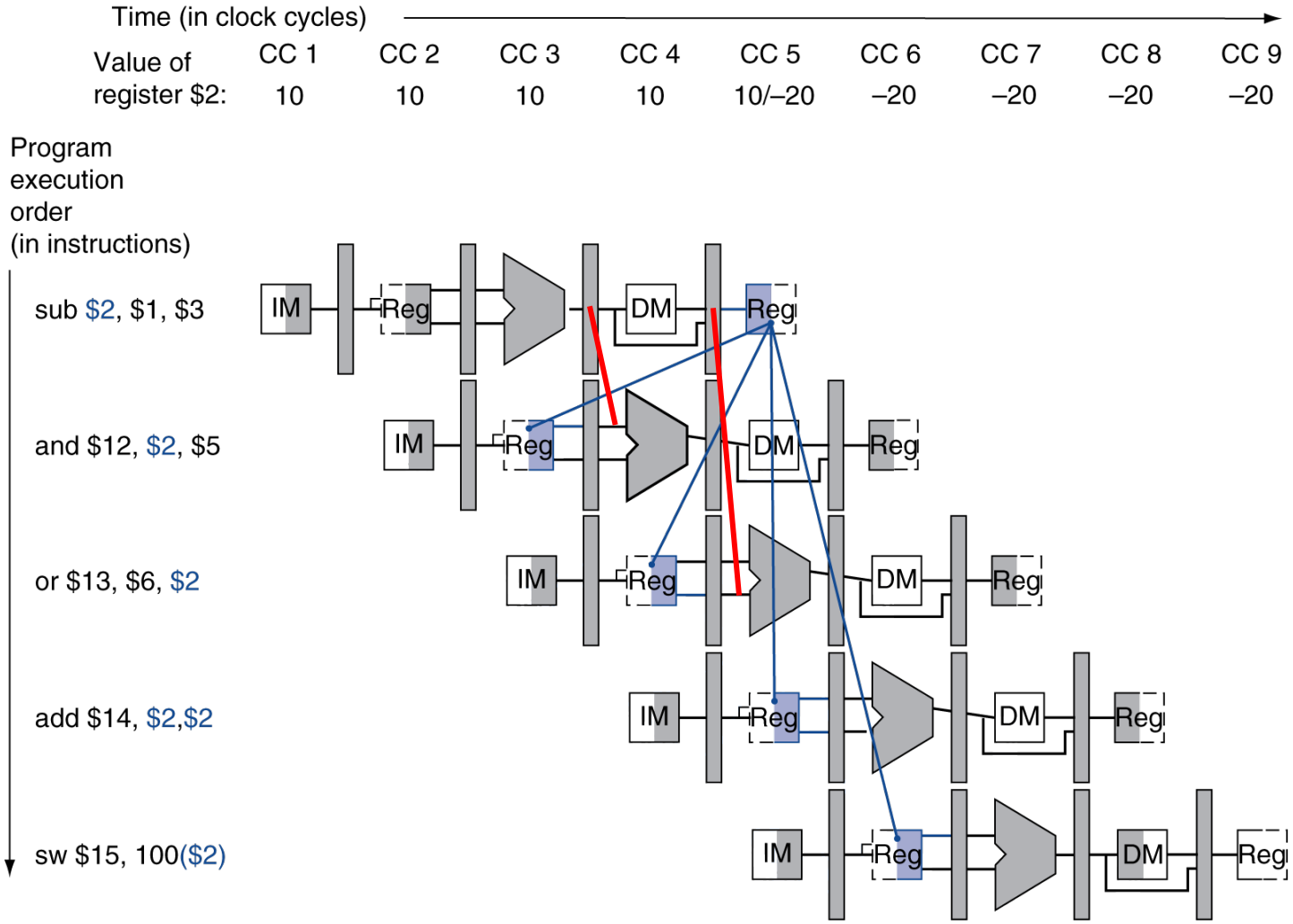
	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register $2 :	10	10	10	10	10/– 20	– 20	– 20	– 20	– 20
Value of EX/MEM :	X	X	X	– 20	X	X	X	X	X
Value of MEM/WB :	X	X	X	X	– 20	X	X	X	X

Program execution order (in instructions)

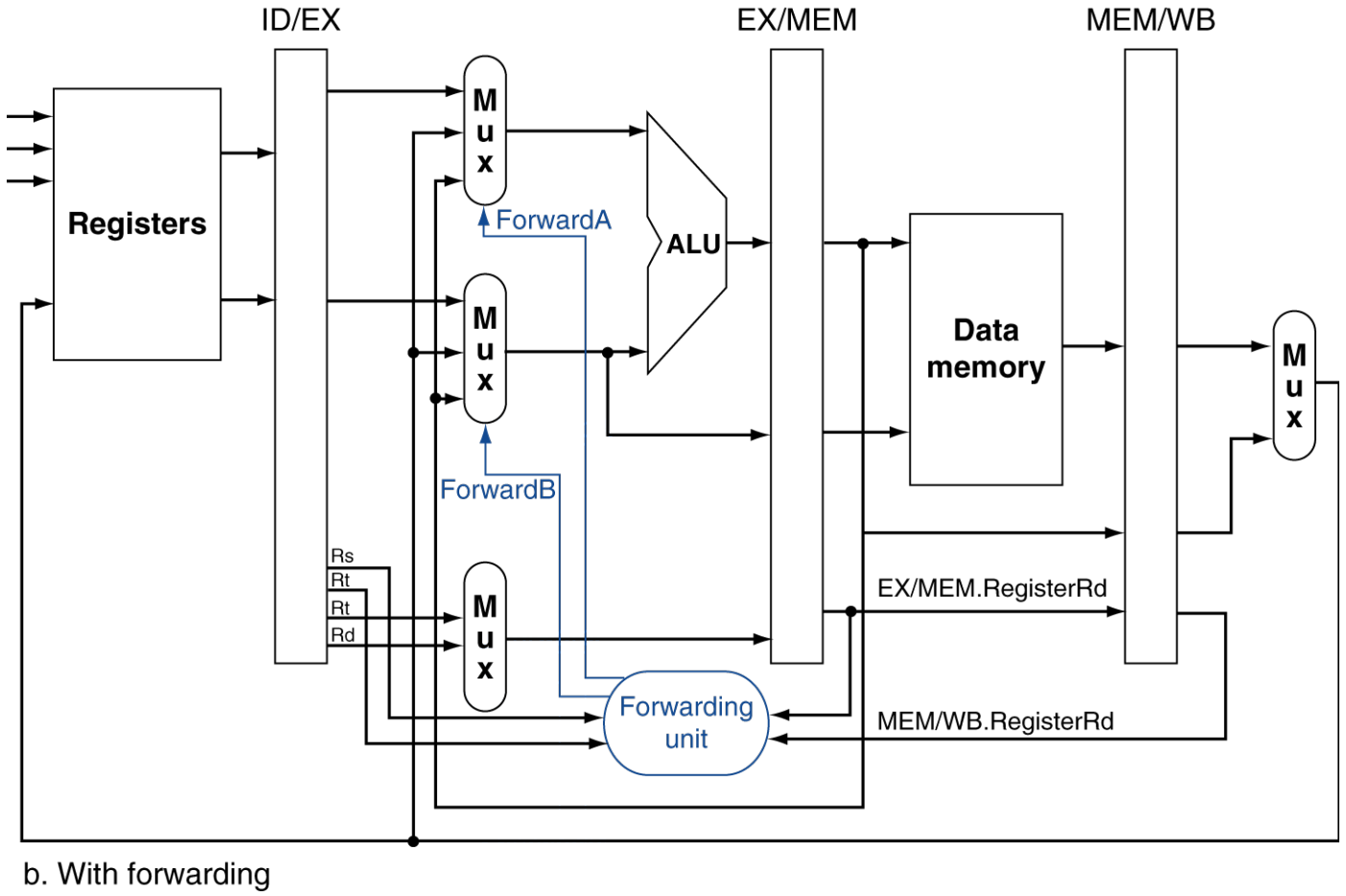


what if this $2 was $13?

Dependencies & Forwarding



Forwarding Paths



b. With forwarding

Detecting the Need to Forward

- Pass register numbers along pipeline
  - e.g., ID/EX.RegisterRs = register number for Rs sitting in ID/EX pipeline register
- ALU operand register numbers in EX stage are given by
  - ID/EX.RegisterRs, ID/EX.RegisterRt
- Data hazards 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

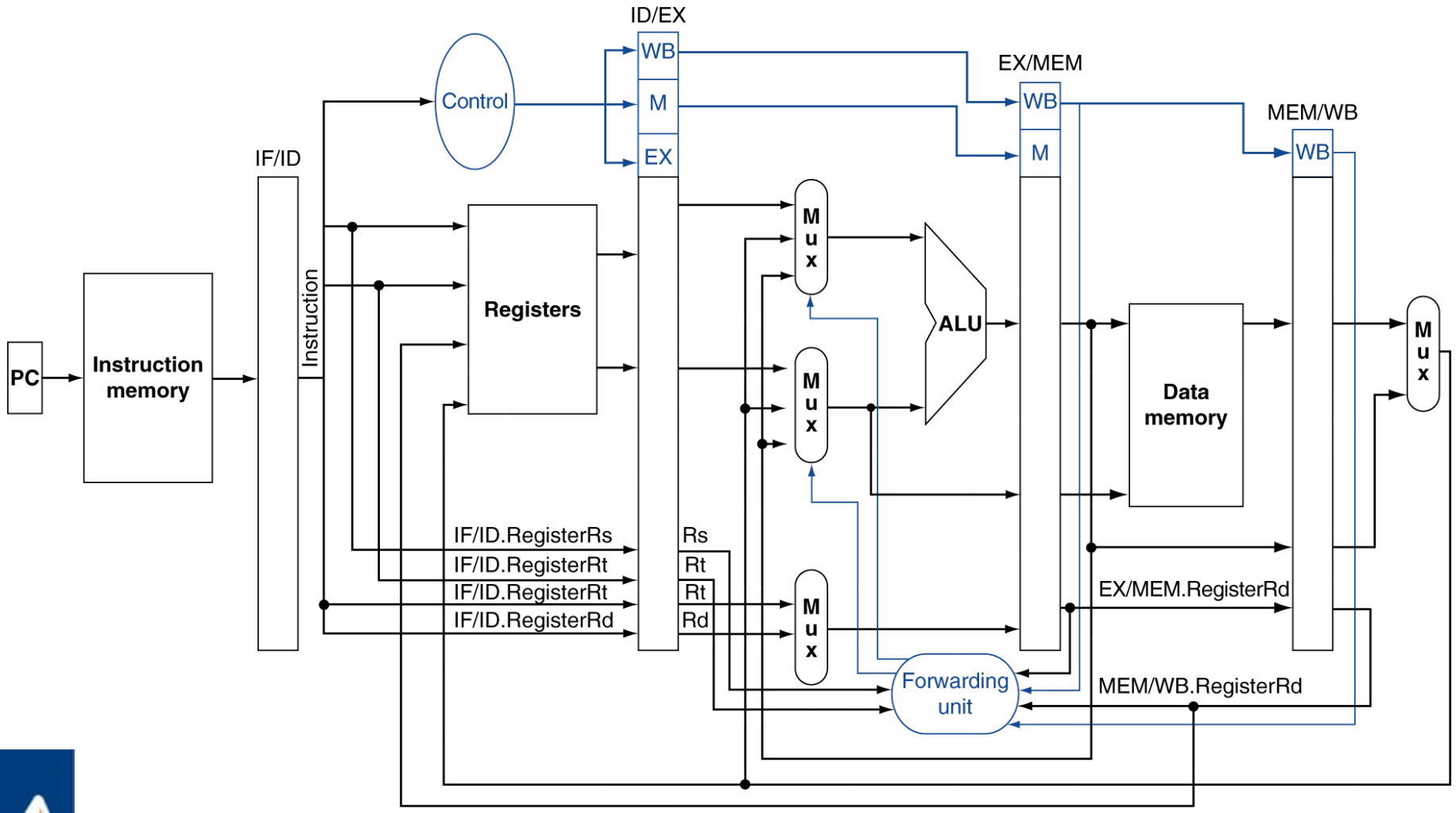
Fwd from MEM/WB pipeline reg

Detecting the Need to Forward

- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite, MEM/WB.RegWrite
- And only if Rd for that instruction is not $zero
  - EX/MEM.RegisterRd ≠ 0,
    MEM/WB.RegisterRd ≠ 0

Datapath with Forwarding

00	Register file
01	Mem. or earlier ALU
10	Prior ALU



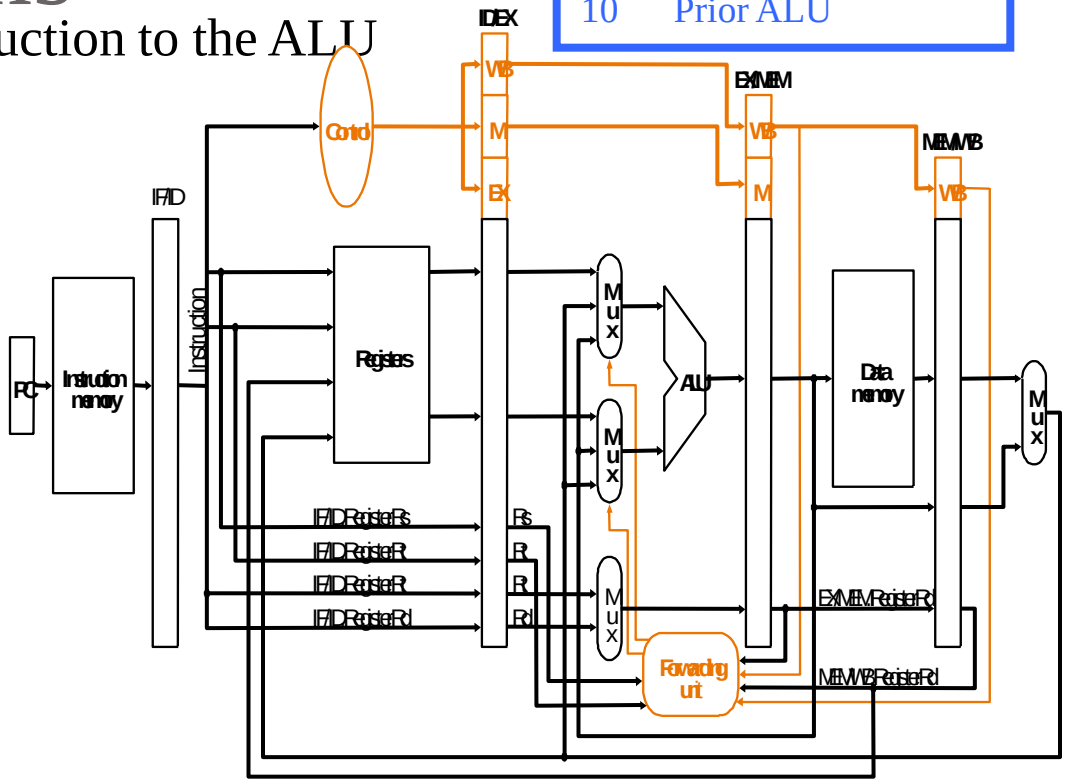
Forwarding Conditions

- 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

Hazard Conditions

- Steer the result from previous instruction to the ALU

00	Register file
01	Mem. or earlier ALU
10	Prior ALU



- EX hazard

  if (EX/MEM.RegWrite

  and (EX/MEM.RegisterRd ≠0)

  and (EX /MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10

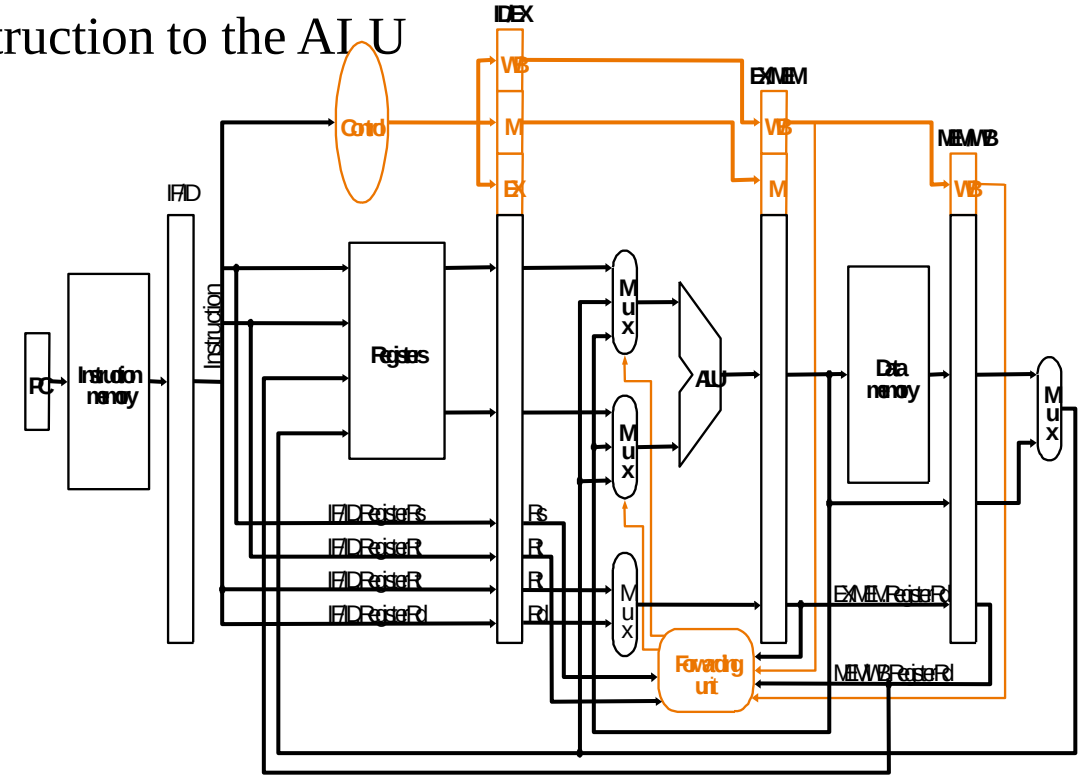
  if (EX/MEM.RegWrite

  and (EX/MEM.RegisterRd ≠0)

  and (EX /MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10

Hazard Conditions

- Steer the result from precious instruction to the ALU



- MEM hazard

  if (MEM/WB.RegWrite

  and (MEM/WB.RegisterRd ≠0)

  and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

  if (MEM/WB.RegWrite

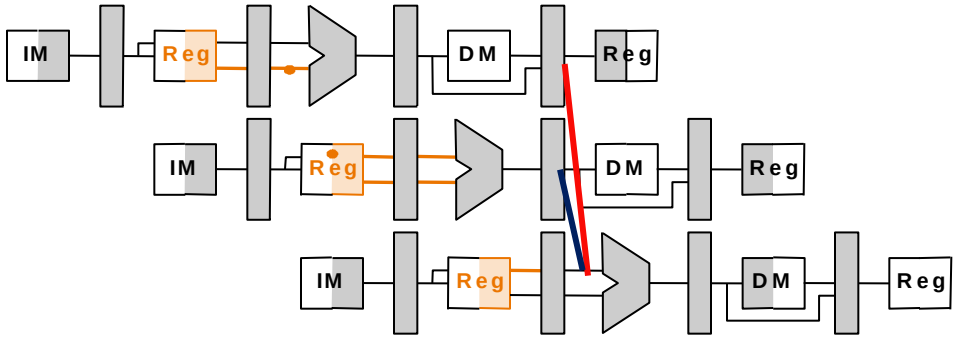
  and (MEM/WB.RegisterRd ≠0)

  and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01

Double Data Hazard

- Consider the sequence:

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



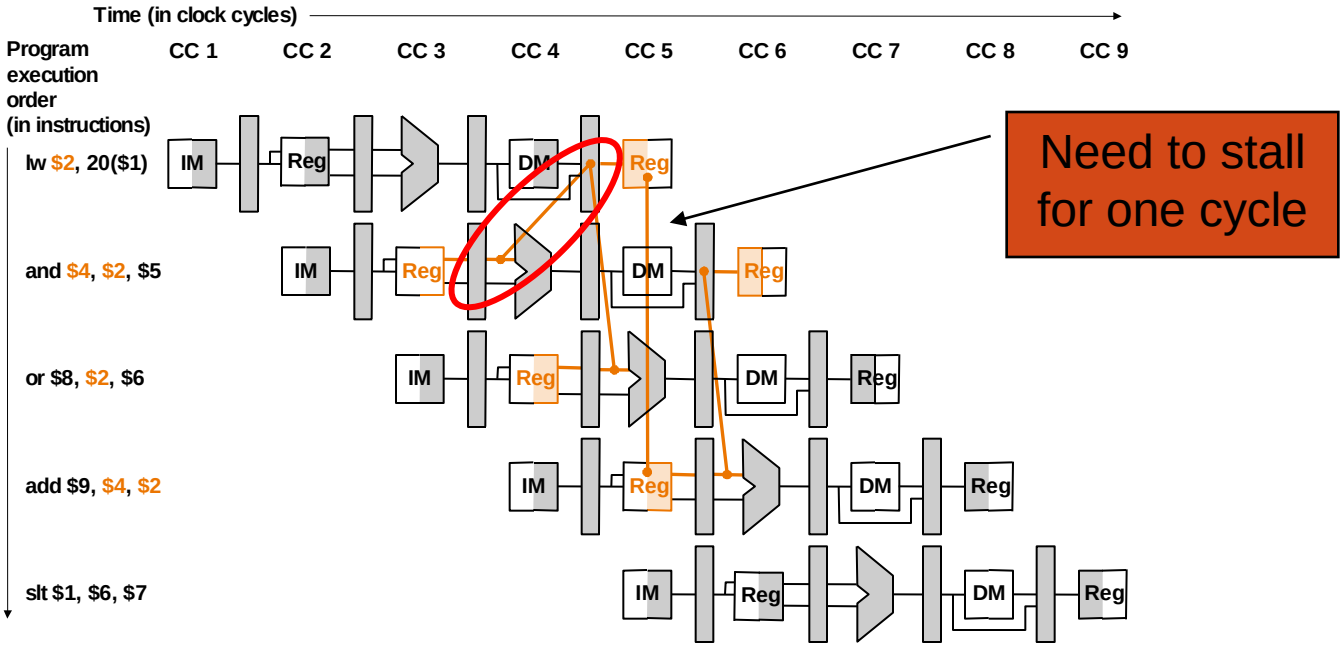
- Both hazards occur
  - Want to use the most recent
- Revise MEM hazard condition
  - Only fwd if EX hazard condition isn't true

Revised Forwarding Condition

- 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

Can't always forward

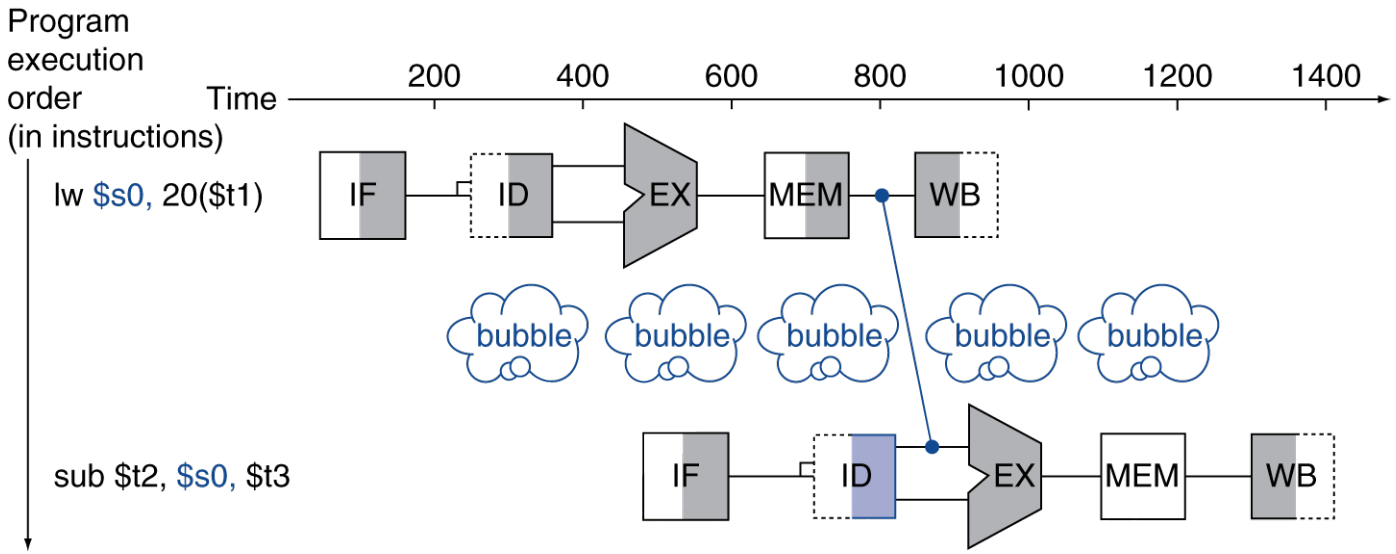
- *lw* can still cause a hazard:
  - An instruction tries to read a register following a load instruction that writes to the same register.



- Thus, we need a hazard detection unit to "stall" the load instruction

Load-Use Data Hazard

- Can't always avoid stalls by forwarding
  - If value not computed when needed
  - Can't forward backward in time!



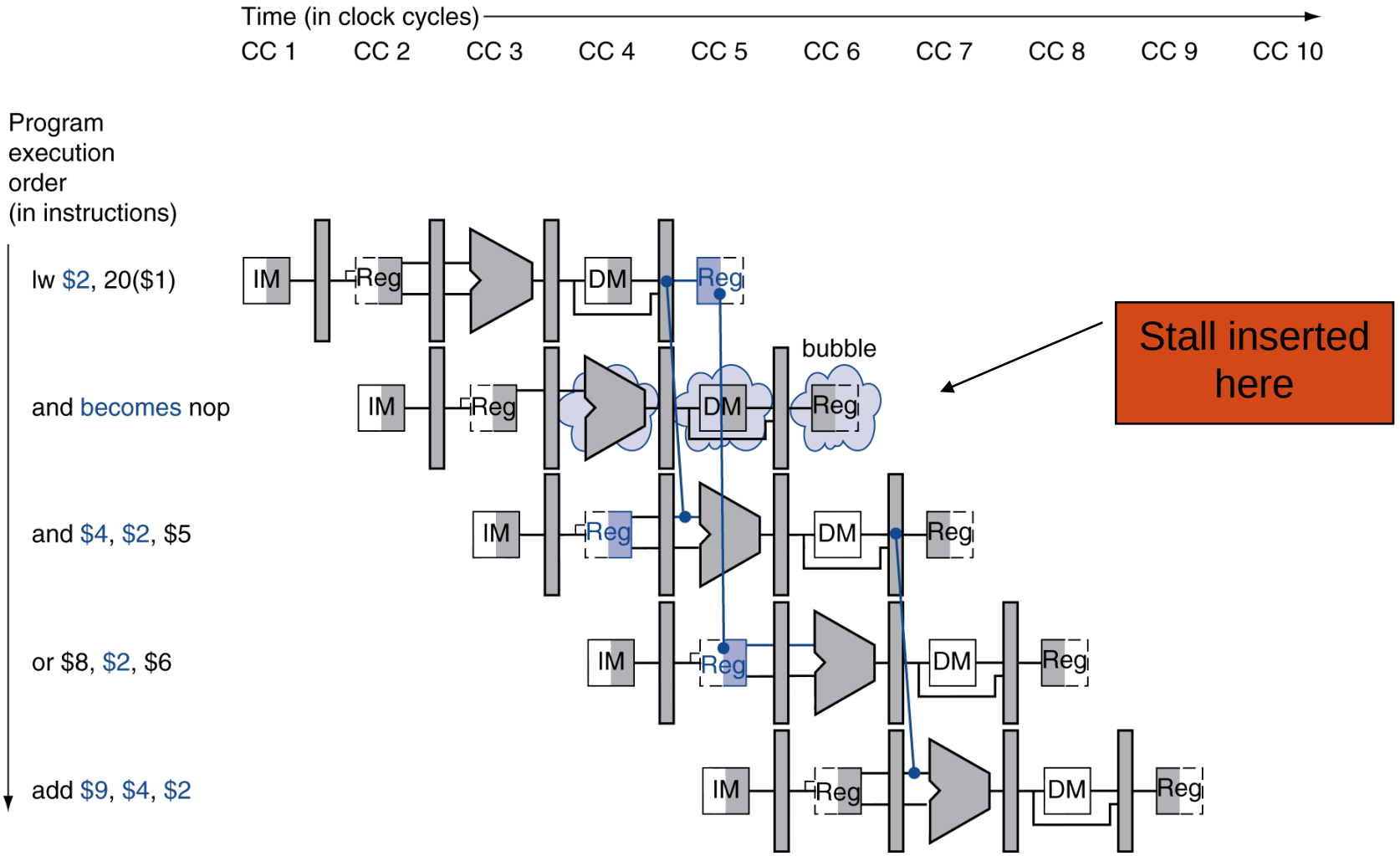
Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand 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

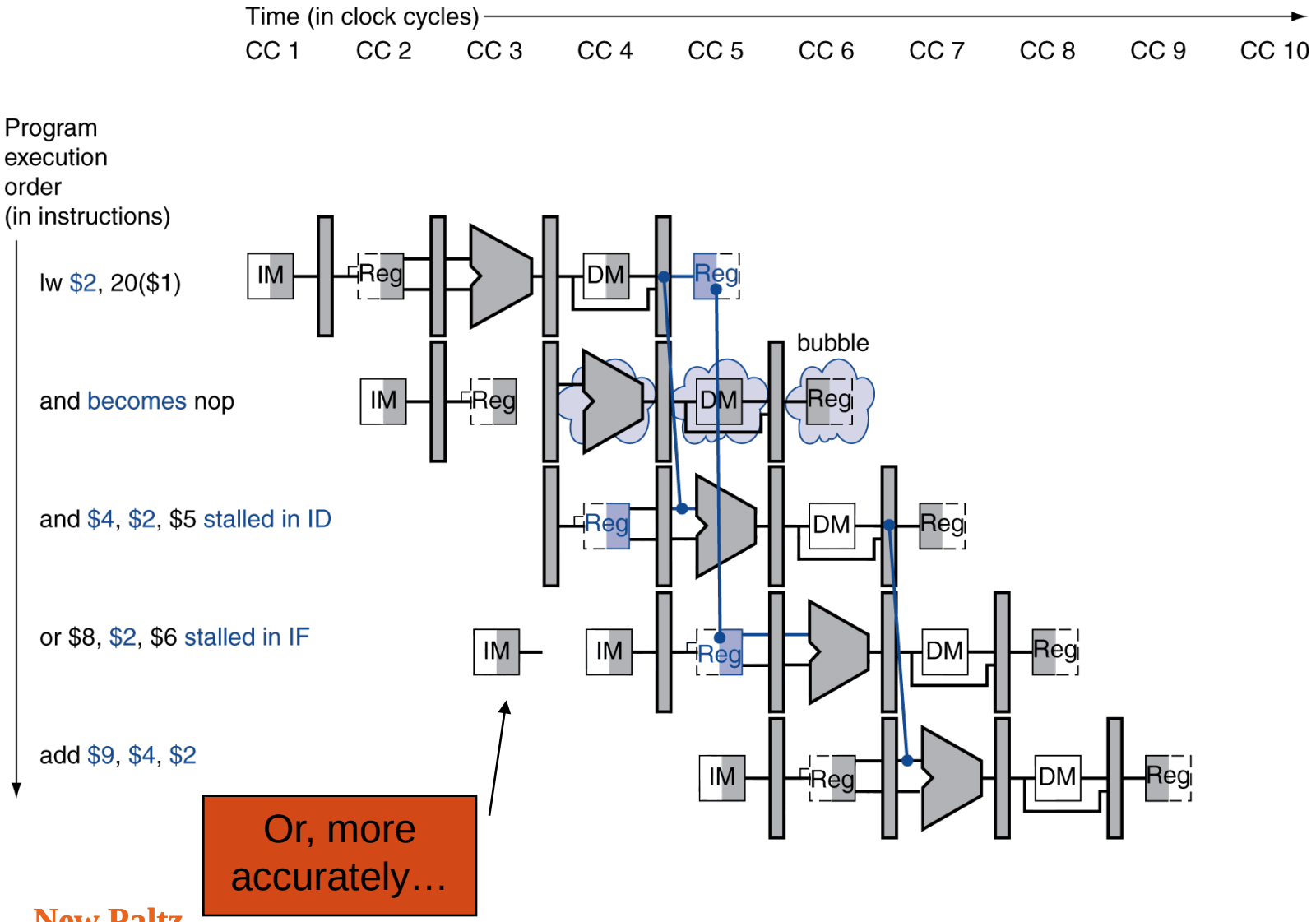
How to Stall the Pipeline

- Force control values in ID/EX register to 0
  - EX, MEM and WB do `nop` (no-operation)
- Prevent update of PC and IF/ID register
  - Using instruction is decoded again
  - Following instruction is fetched again
  - 1-cycle stall allows MEM to read data for `lw`
    - Can subsequently forward to EX stage

Stall/Bubble in the Pipeline

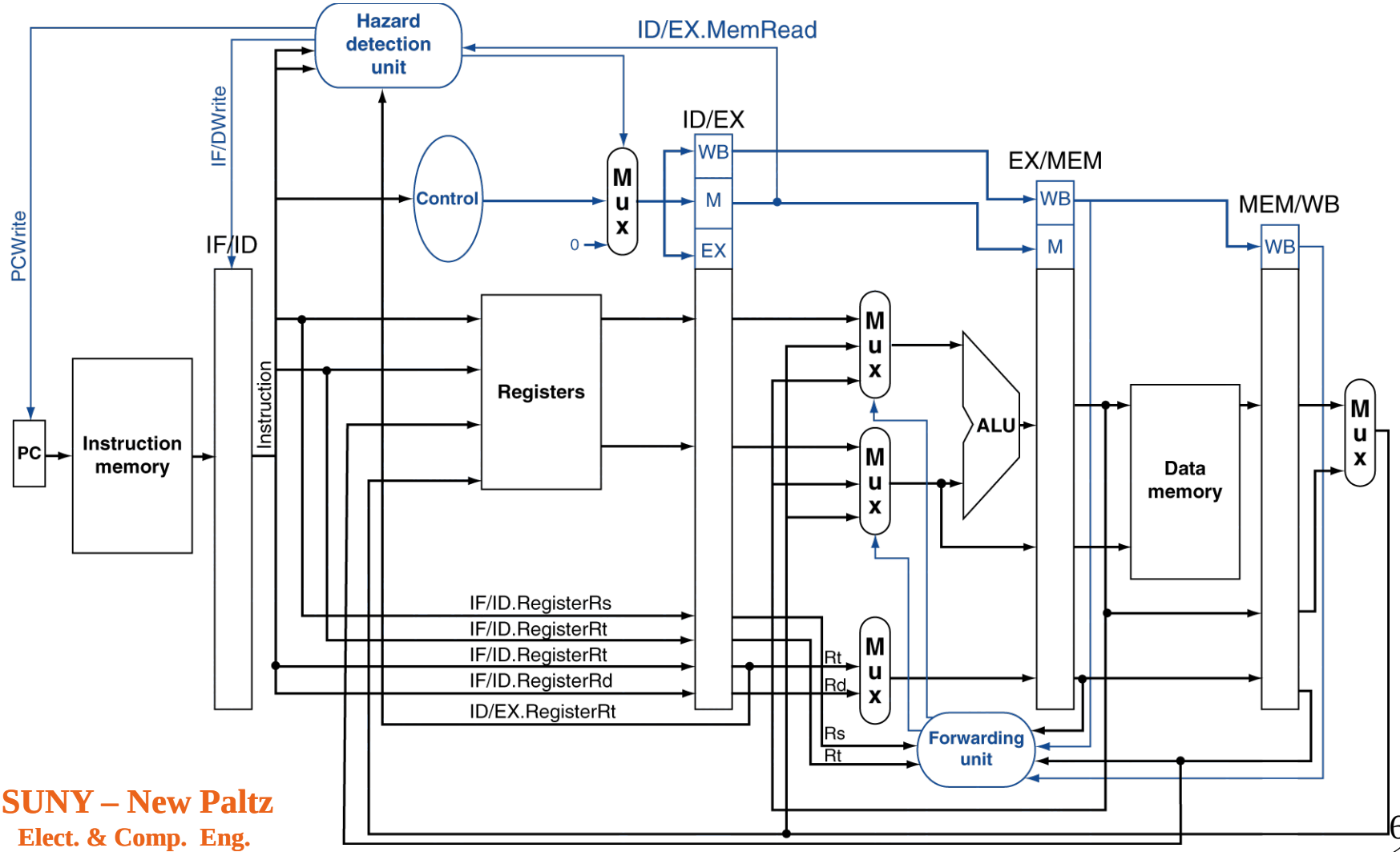


Stall/Bubble in the Pipeline



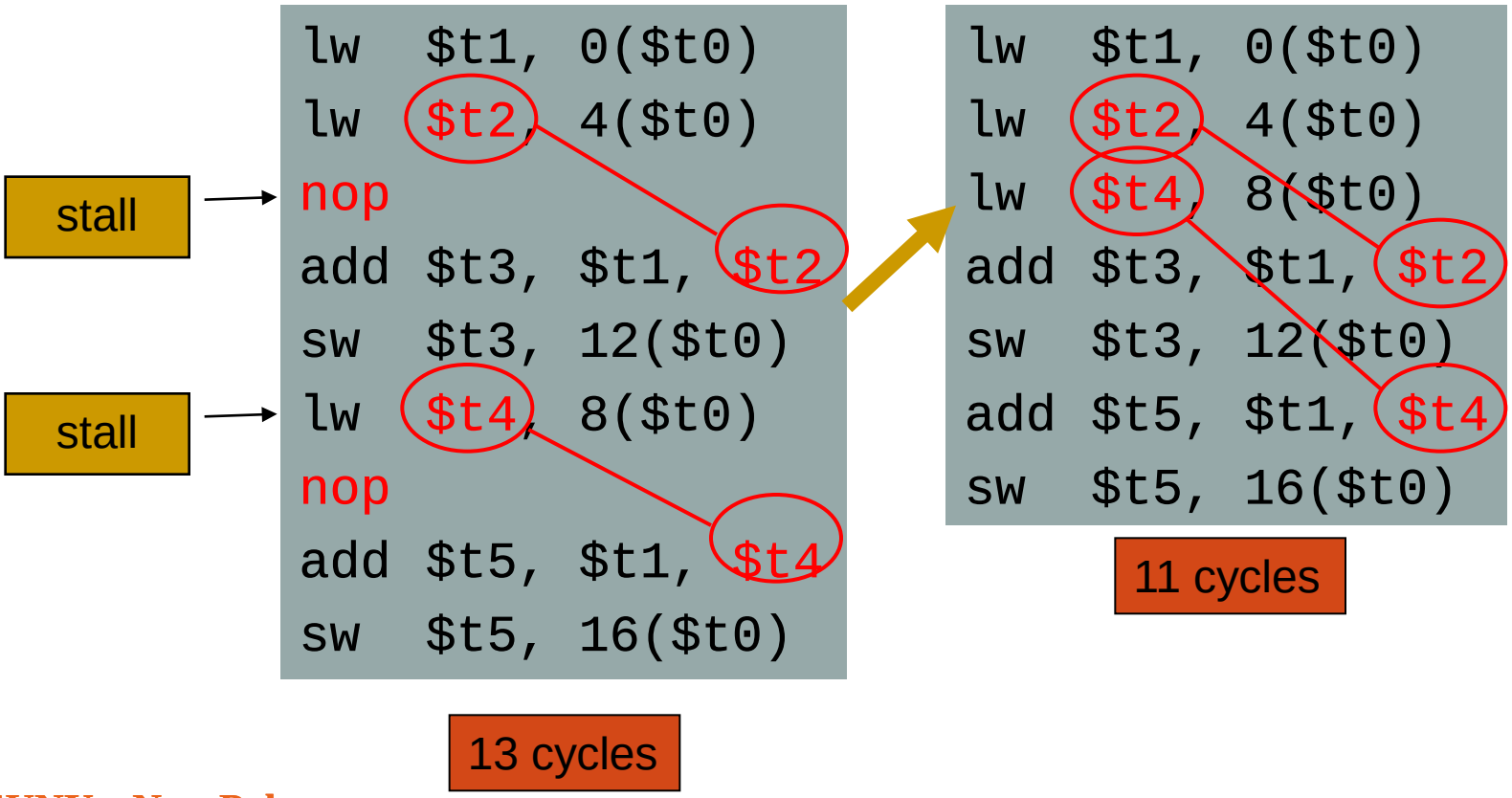
Datapath with Hazard Detection

- Stall by letting an instruction that won't write anything go forward
- Controls writing of the PC and IF/ID plus MUX

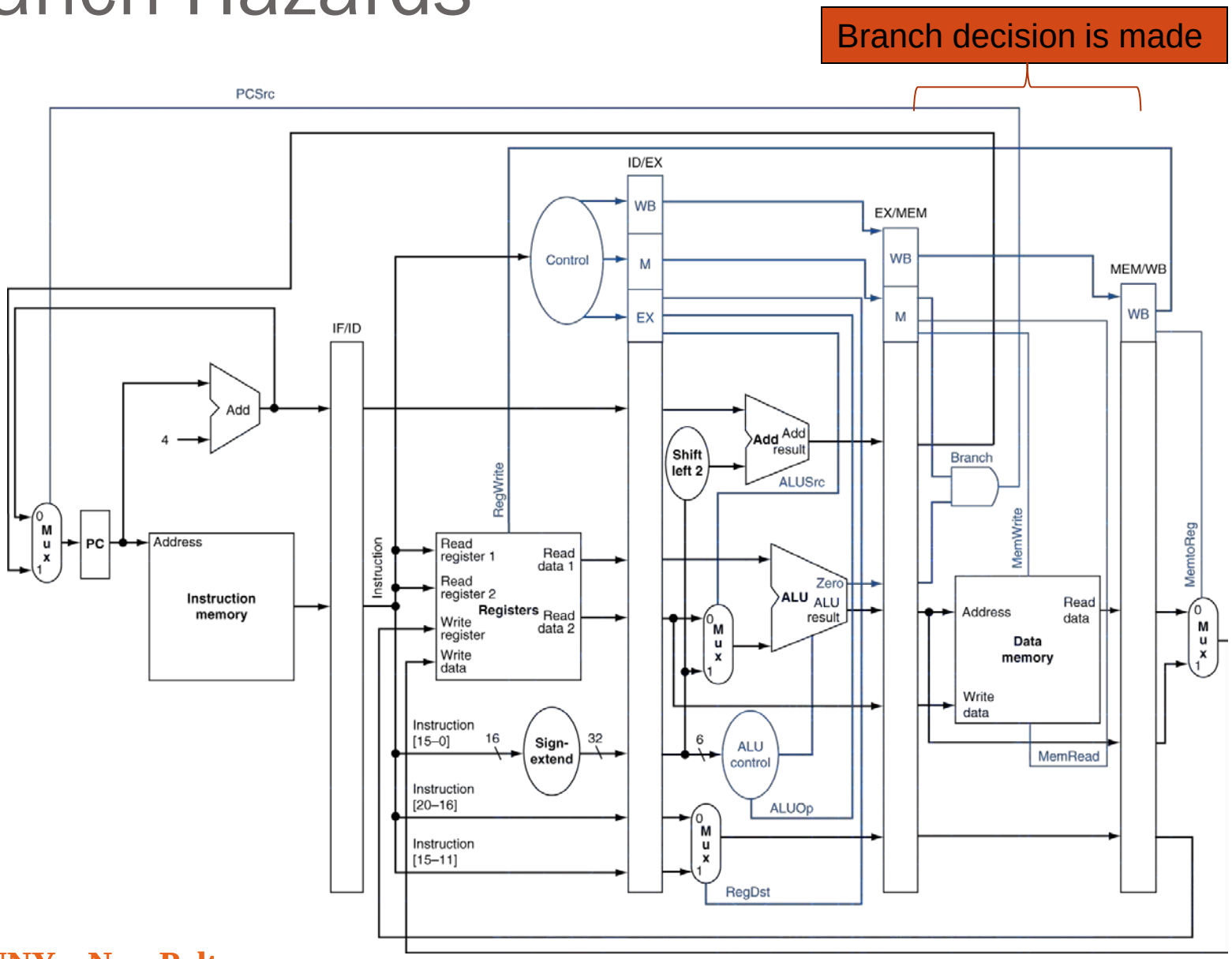


Code Scheduling to Avoid Stalls

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

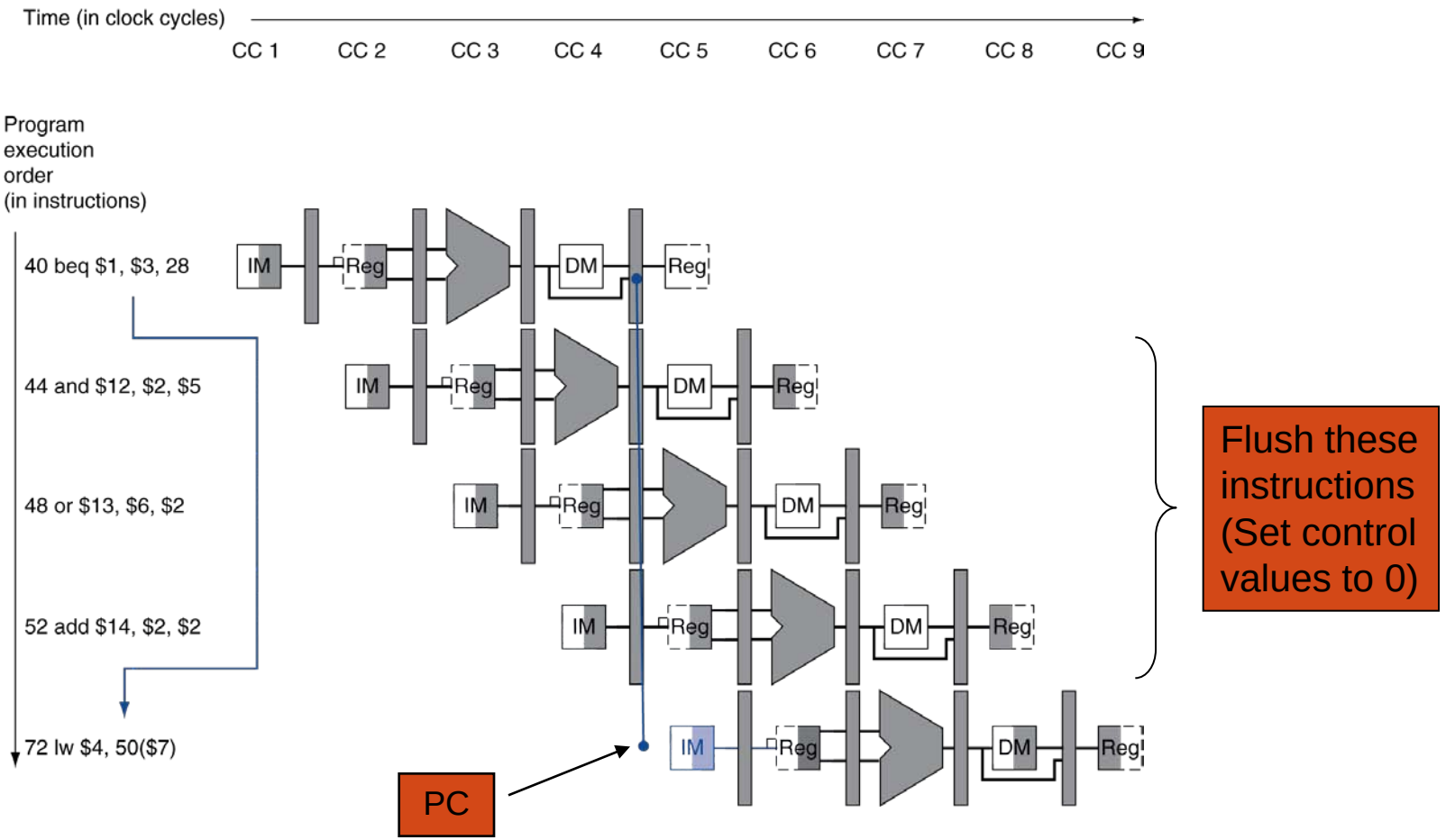


Branch Hazards



Branch Hazards

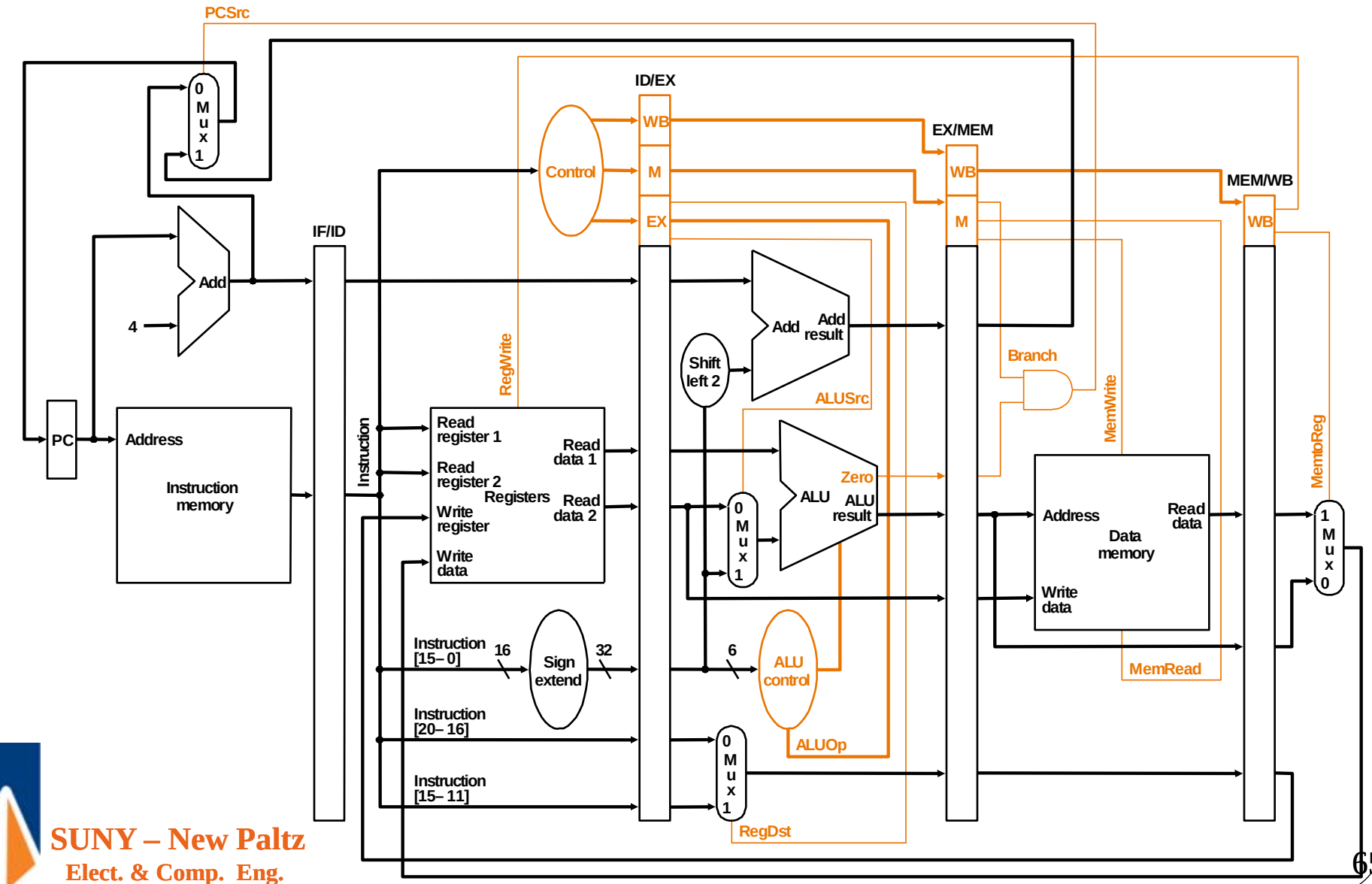
- When decide to branch, other instructions may be in the pipeline!
- If branch outcome determined in MEM



Control Hazards

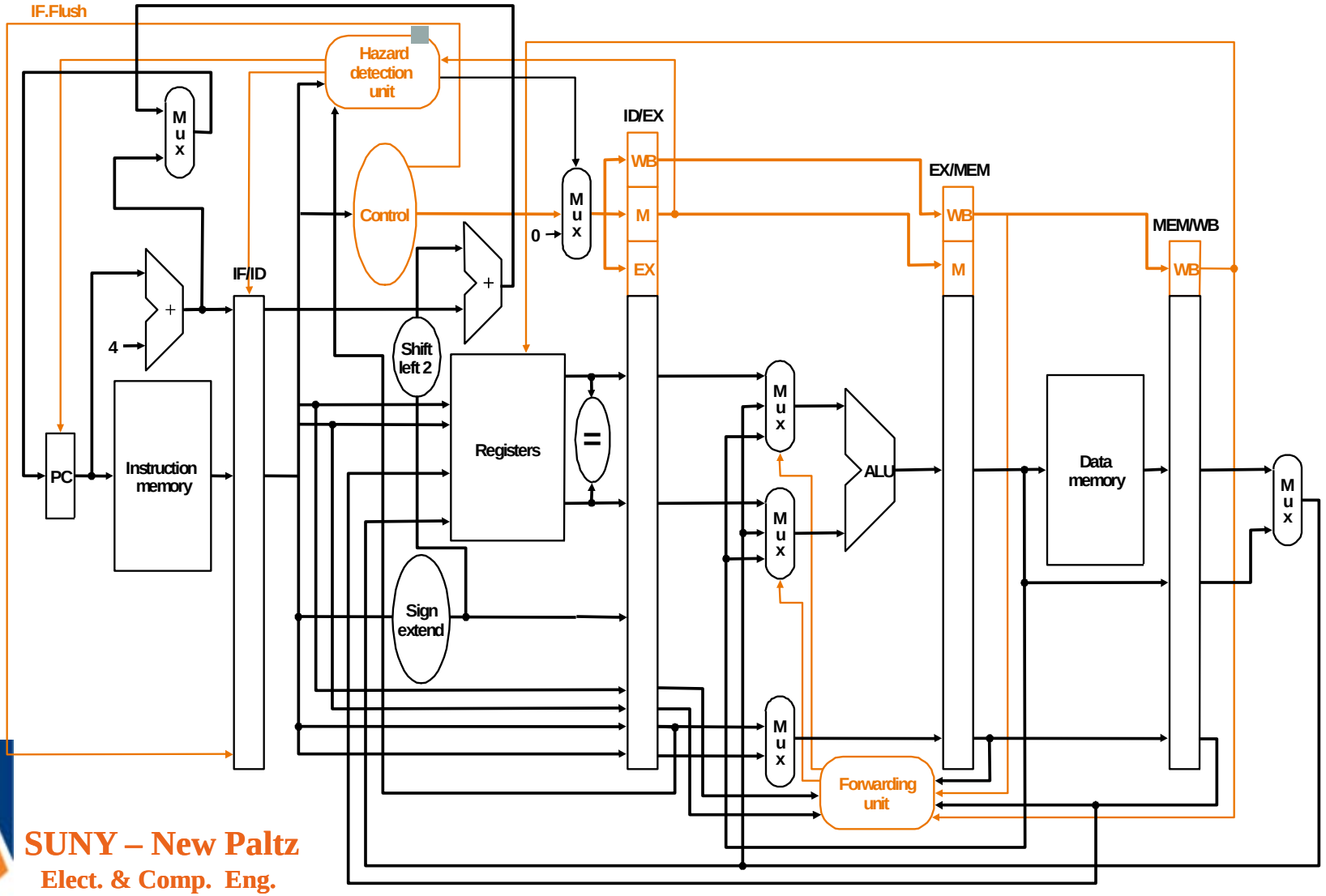
- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction
    - Still working on ID stage of branch
- In MIPS pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

Our Original Datapath



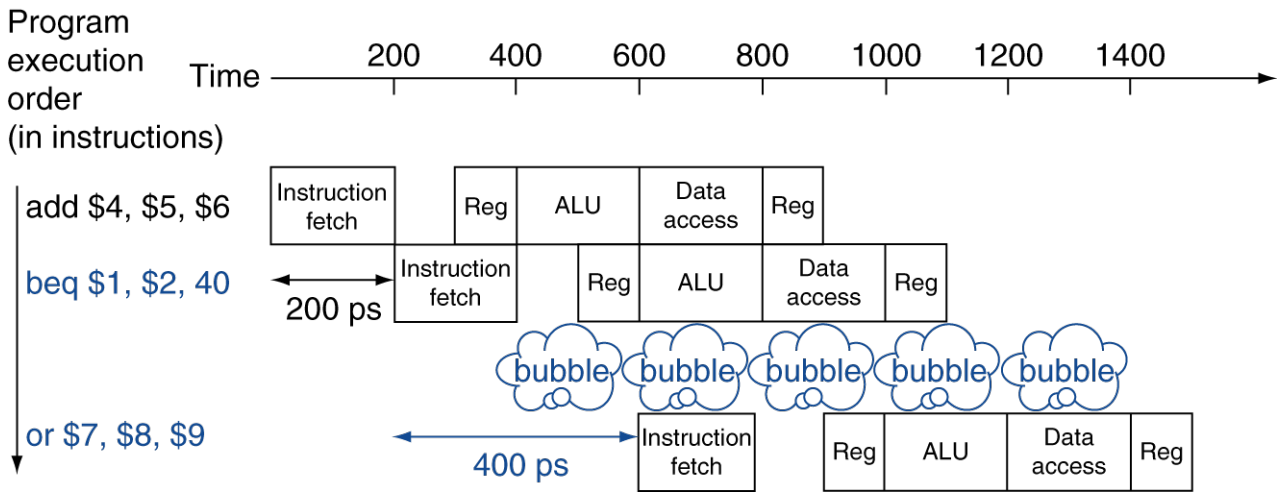
SUNY – New Paltz
Elect. & Comp. Eng.

Reduce Branch Delay



Stall on Branch

- One solution: Wait until branch outcome determined before fetching next instruction
- Another solution: flush the pipe if branch is taken – only one delay penalty.

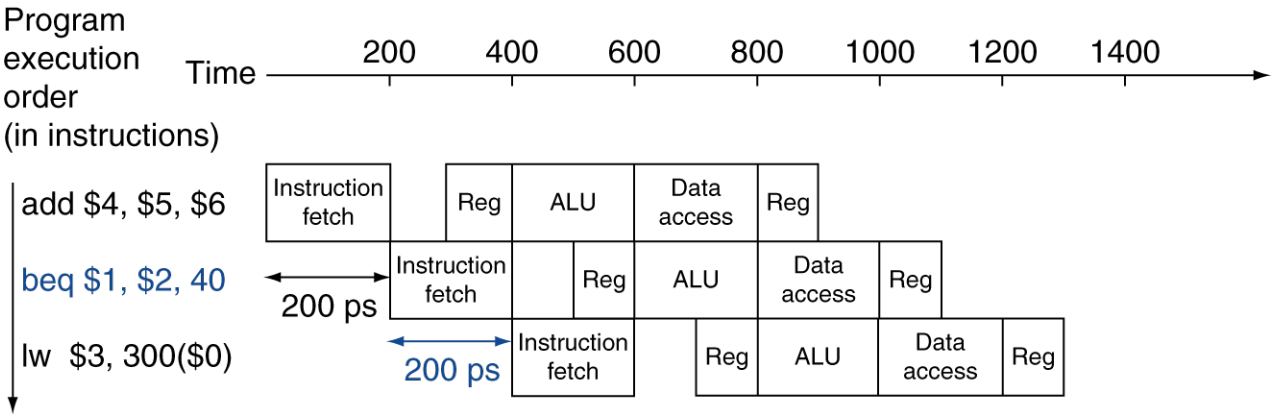


Branch Prediction

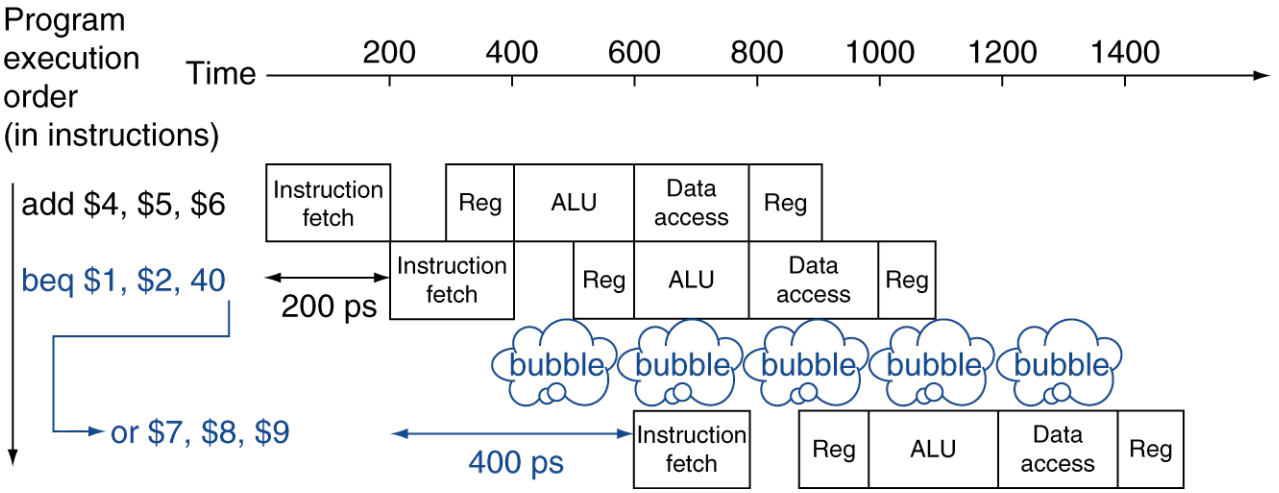
- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In MIPS pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay
    - Need to add hardware for flushing instructions if we are wrong

MIPS with Predict Not Taken

Prediction correct



Prediction incorrect

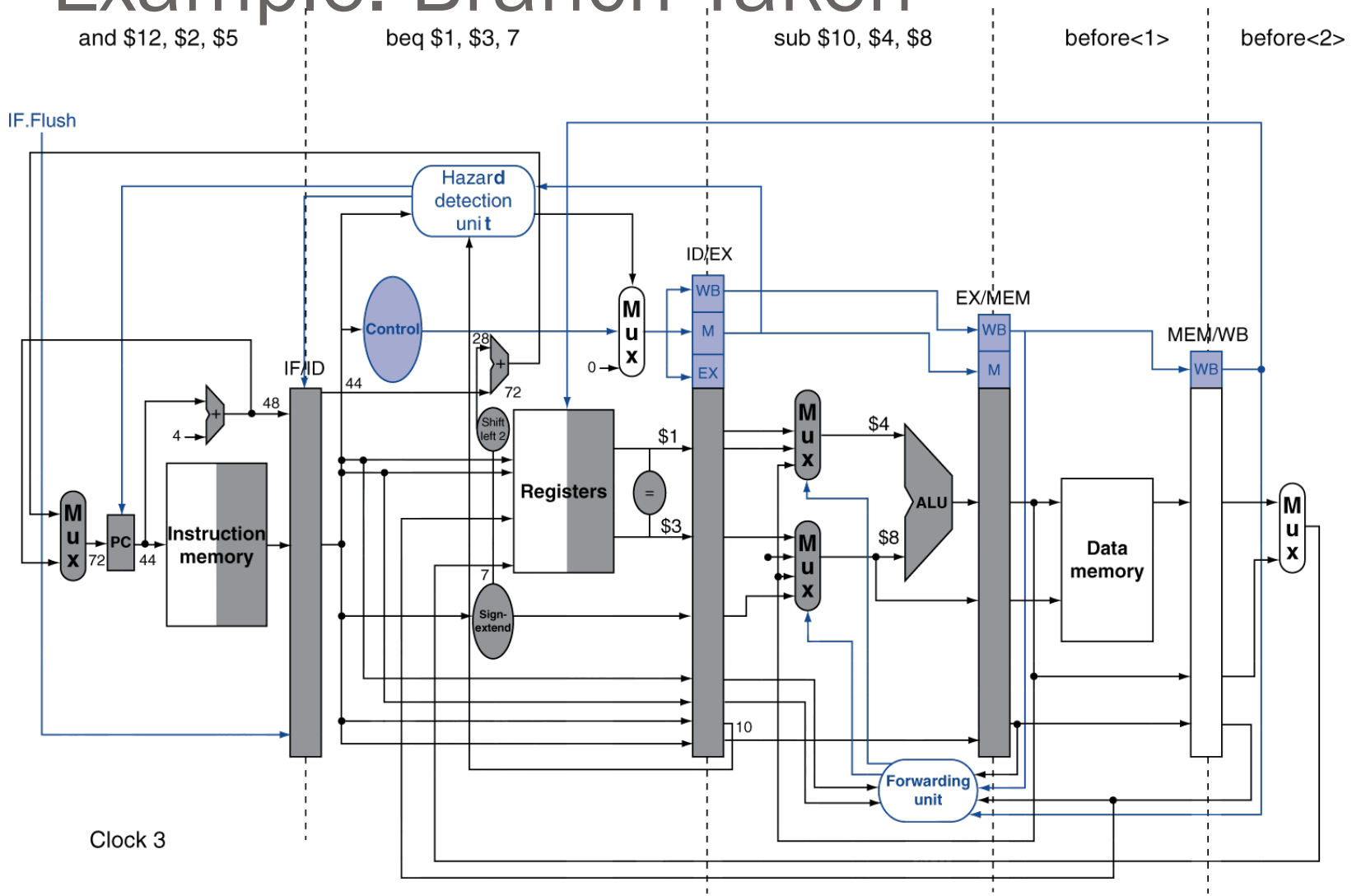


Reducing Branch Delay

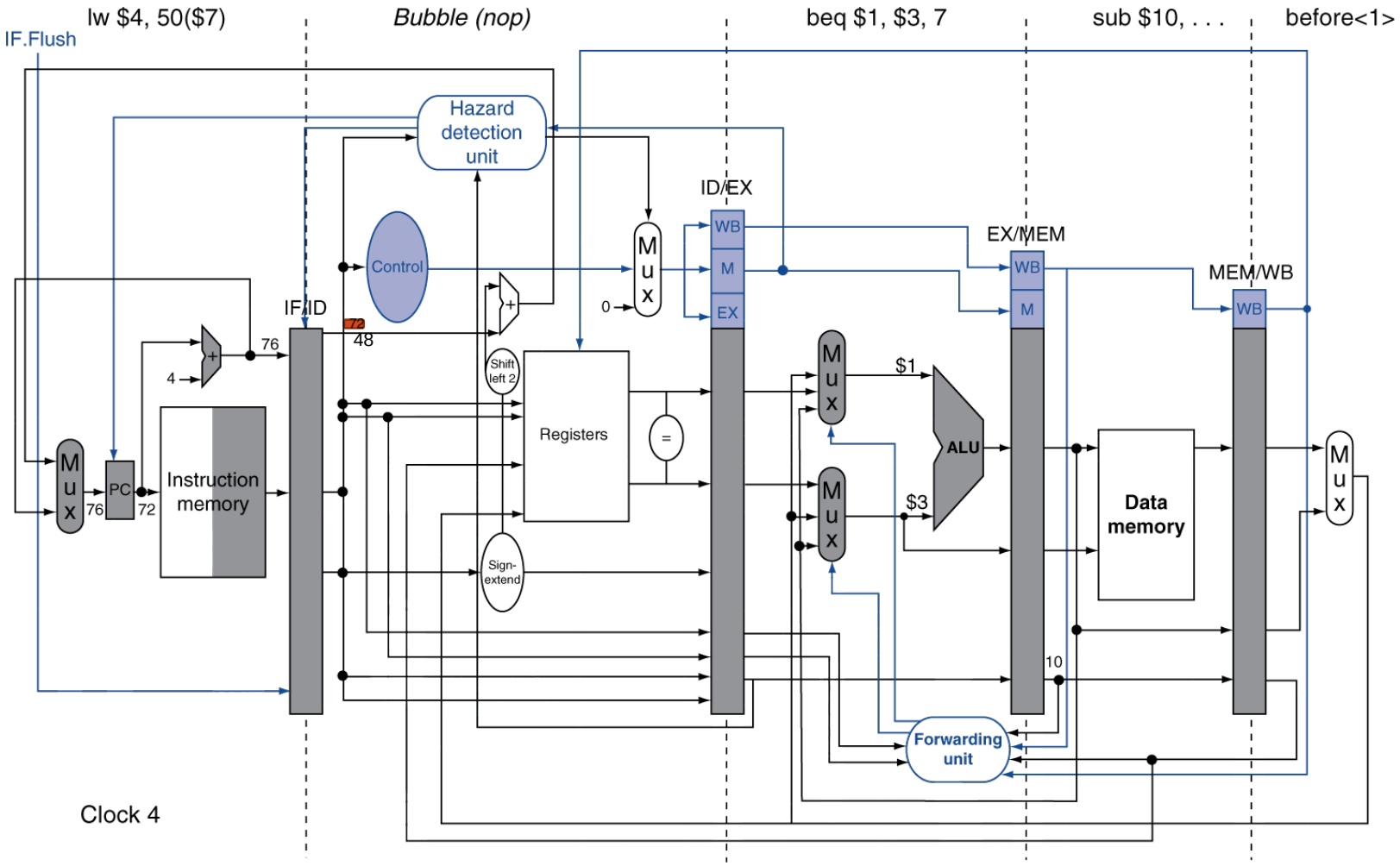
- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator
- Example: branch taken

```
36:   sub   $10, $4, $8
40:   beq   $1,  $3, 7
44:   and   $12, $2, $5
48:   or    $13, $2, $6
52:   add   $14, $4, $2
56:   slt   $15, $6, $7
      ...
72:   lw    $4, 50($7)
```

Example: Branch Taken



Example: Branch Taken



Pipeline Summary

The BIG Picture

- Pipelining improves performance by increasing instruction throughput
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
- Instruction set design affects complexity of pipeline implementation

Stalls and Performance

The BIG Picture

- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure