Santa Clara University

COEN 210

Chapter 4, Part 3 – Pipelined Processor

Yan Cui

ycui4@scu.edu

Where are we?

- Chapter 1 – Introduction
- Chapter 2 – RISC-V
- Chapter 3 - Arithmetic design
  - Chapter 3.1 - logic gates
  - Chapter 3.2 – Arithmetic
- Chapter 4 – Processor design
  - Chapter 4.1 - Sequential Logic & Memory
  - Chapter 4.2 - Single Cycle Processor
  - Chapter 4.3 - Pipelined Processor
- chapter 5 - memory hierarchy
  - Chapter 5.1 - Caches
  - Chapter 5.2 - Virtual Memory

Why a Single-Cycle Implementation Is Not Used Today

- Although the single cycle design will work correctly
- Inefficient
  - why?
    - The clock cycle must have the same length for every instruction.
    - The longest possible path in the processor determines the clock cycle
  - Although the CPI is 1, the overall performance is poor, since the clock cycle is too long.
- Early computers with very simple instruction sets did use single-cycle design, however, not suitable for floating-point unit, or a complex instruction set.

Example: Determine Clock Cycle Time for Single Cycle CPU

- Let us assume that the **only** delays introduced are by the following tasks:
  - Memory access (read or write time = 200ps)
    - Instruction memory, data memory
  - Register file access (read or write time = 100ps)
  - ALU to perform function (= 200ps)
- Instruction execution times:

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
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

- Clock cycle time needs to be greater than or equal to the longest datapath (800ps) ➔ clock frequency <= 1.25 GHz

Drawbacks of Single Cycle CPU

- Longest datapath determines the clock cycle time (or clock period)
  - Critical path: load instruction
    - Instruction memory → register file → ALU → data memory → register file
- This results in a lot of waste
  - both in terms of time and space since we need multiple of the same kinds of datapath elements to execute a single instruction
- One solution: Pipelining
  - All modern processors use pipelining to improve performance
  - Pipelining involves not only executing an instruction over multiple cycles, but also executing multiple instructions per cycle.
  - In other words, we're going to overlap the execution of several instructions.
- Pipelining is nearly universal

A Classic, Intuitive Analogy: Laundry

- Let's say we have a couple of loads of laundry to do.
- Each load of laundry involves the following steps: washing, drying, folding, and putting away.

Place one dirty load of clothes in the washer

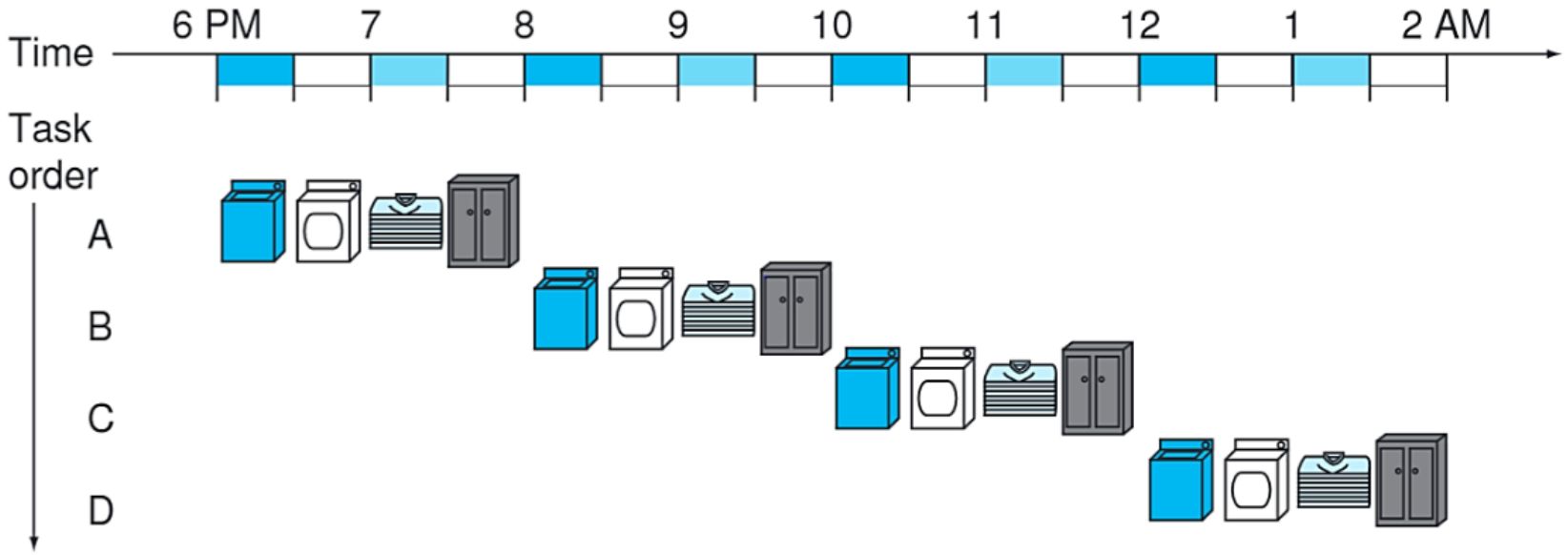
When the washer is finished, place the wet load in the dryer

When the dryer is finished, Place the dry load on a table and fold

Ask your roommate to put the clothes away

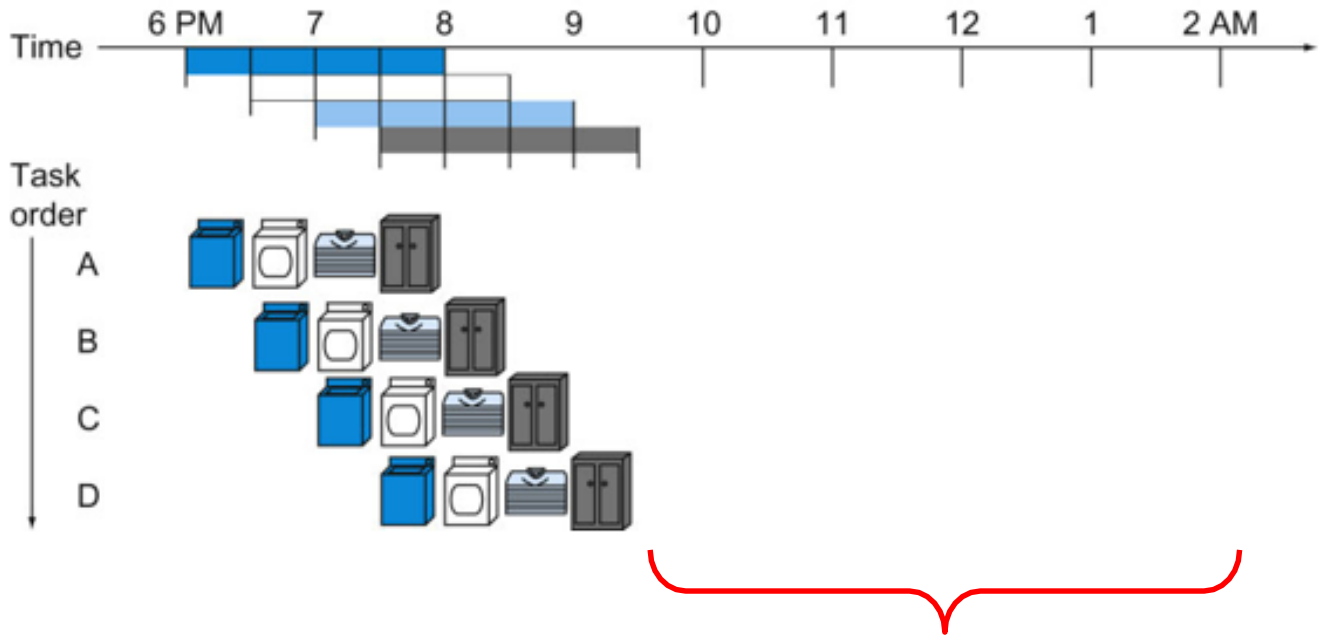
Laundry Analogy-Do It Sequentially

- each step takes 30 minutes
- Assume we have 4 loads
- Do it sequentially
  - Starting the next load after the previous load has finished
  - It will take 8 hours



Do It In Pipeline

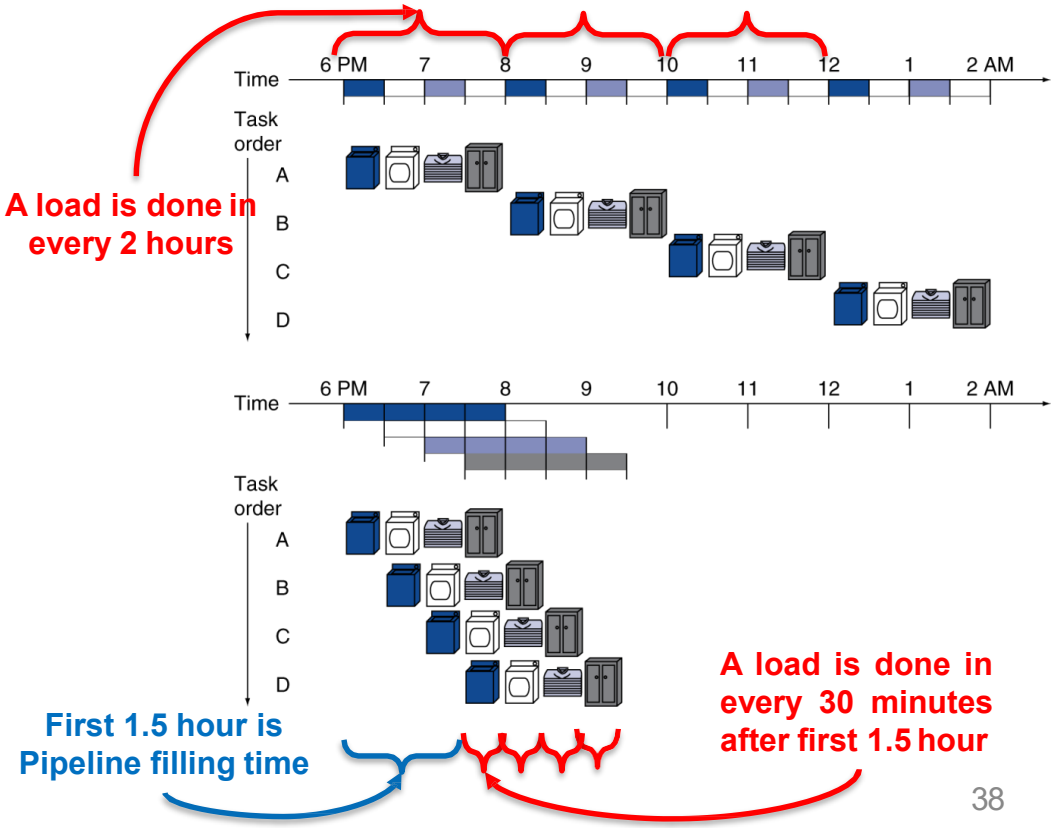
- Start the next load right after the first load is out of the washer. The washer is available now, after all.
- This is analogous to pipelining.



- Pipelined laundry takes 3.5 hours

Pipeline is One Form of Parallelism

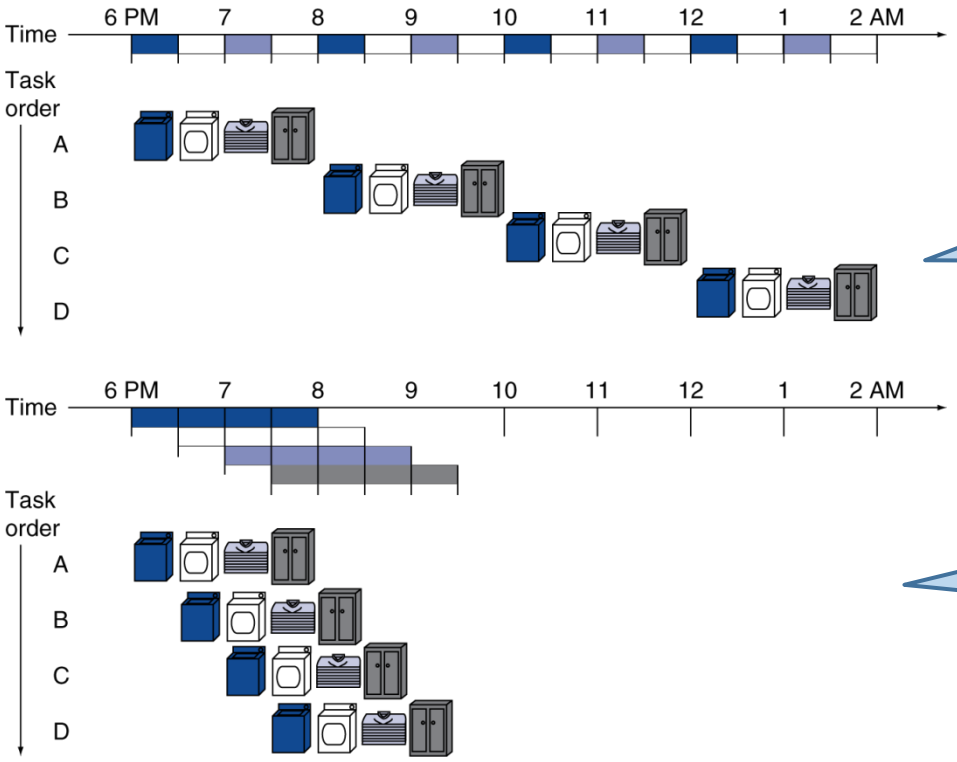
- Pipelined laundry: overlapping execution



- Four workloads
  - Speedup = 8 hrs /3.5 hrs = 2.3
- Infinite laundry loads (n loads):
  - Time for sequential : 2*n
  - Time for pipeline : 0.5*n + 1.5
  - Speedup = 2n/(0.5n+1.5) ≈ 4
  - Which is number of stages (washer, dryer, fold, stash)
  - *All steps are called stages*

Pipelined CPU

- Similarly, we can parallelize the single-cycle CPU tasks

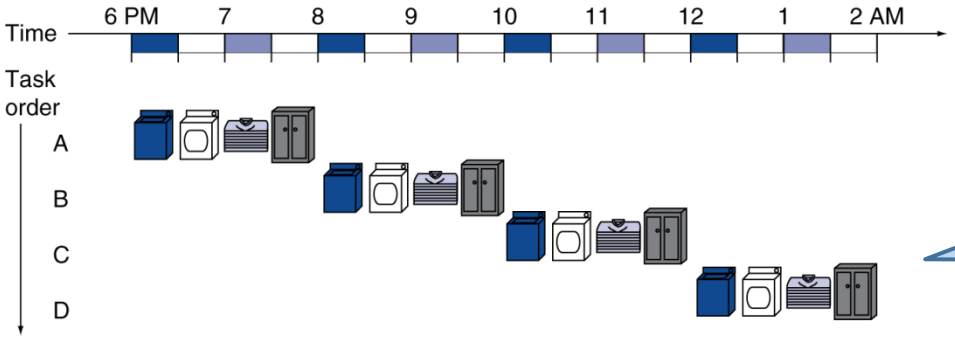


Single-cycle CPU
Clock period = 2 hrs
CPI = 1

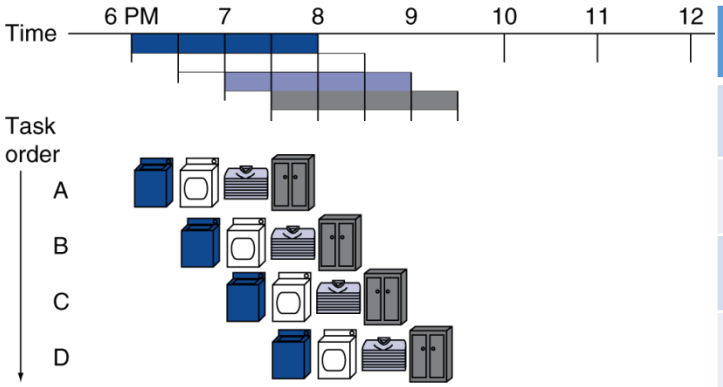
Pipelined CPU
Clock period = 0.5 hrs
CPI = ??

Pipelined CPU

- Similarly, we can parallelize the single-cycle CPU tasks



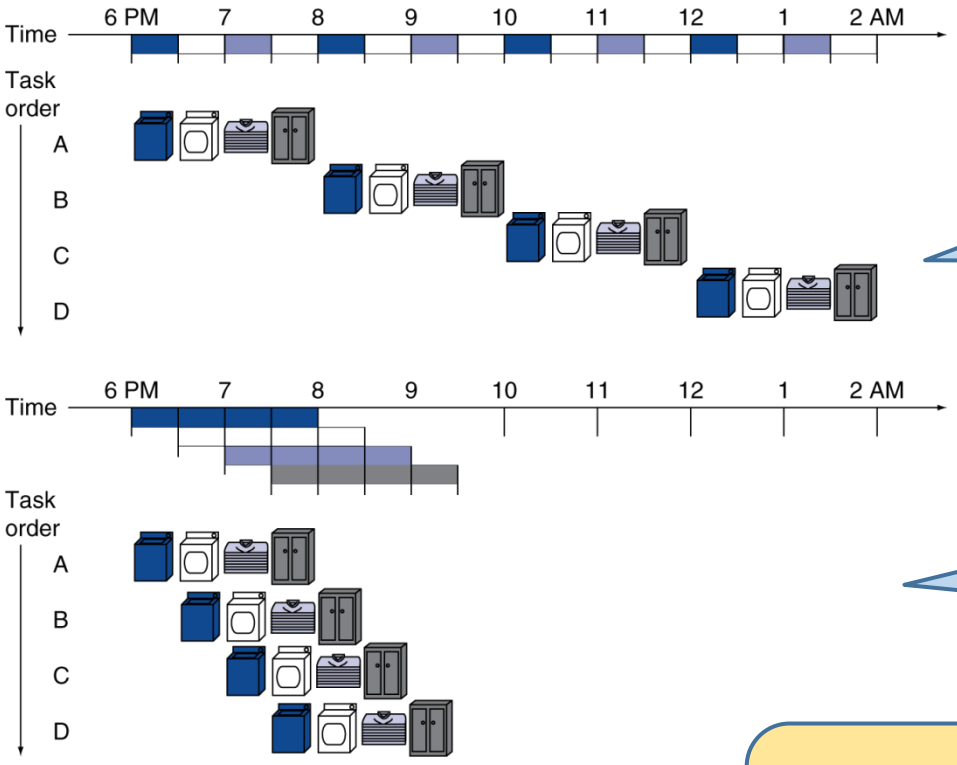
Single-cycle CPU
Clock period = 2 hrs
CPI = 1



Load	Cycles	CPI
1	4	4
2	5	2.5
3	6	2
4	7	1.75
…	…	…
N	N+3	≈ 1 (when n -> ∞)

Pipelined CPU

- Similarly, we can parallelize the single-cycle CPU tasks



Single-cycle CPU
Clock period = 2 hrs
CPI = 1

Pipelined CPU
Clock period = 0.5 hrs
CPI = 1

Pipelined CPU can offer up to m times speedup where m is the number of pipelined stages

RISC-V Pipeline

PIPELINING

- Pipelining essentially involves creating an assembly line for instruction execution.
- Each step in the pipeline is called a stage.
- Multiple instructions can be processed in parallel as long as they are at different stages.

Stages

- Executing a RISC-V instruction can take up to five stages.

Step	Name	Description
Instruction Fetch	IF	Read an instruction from memory.
Instruction Decode	ID	Read source registers and generate control signals.
Execute	EX	Compute an R-type result or a branch outcome.
Memory	MEM	Read or write the data memory.
Writeback	WB	Store a result in the destination register.

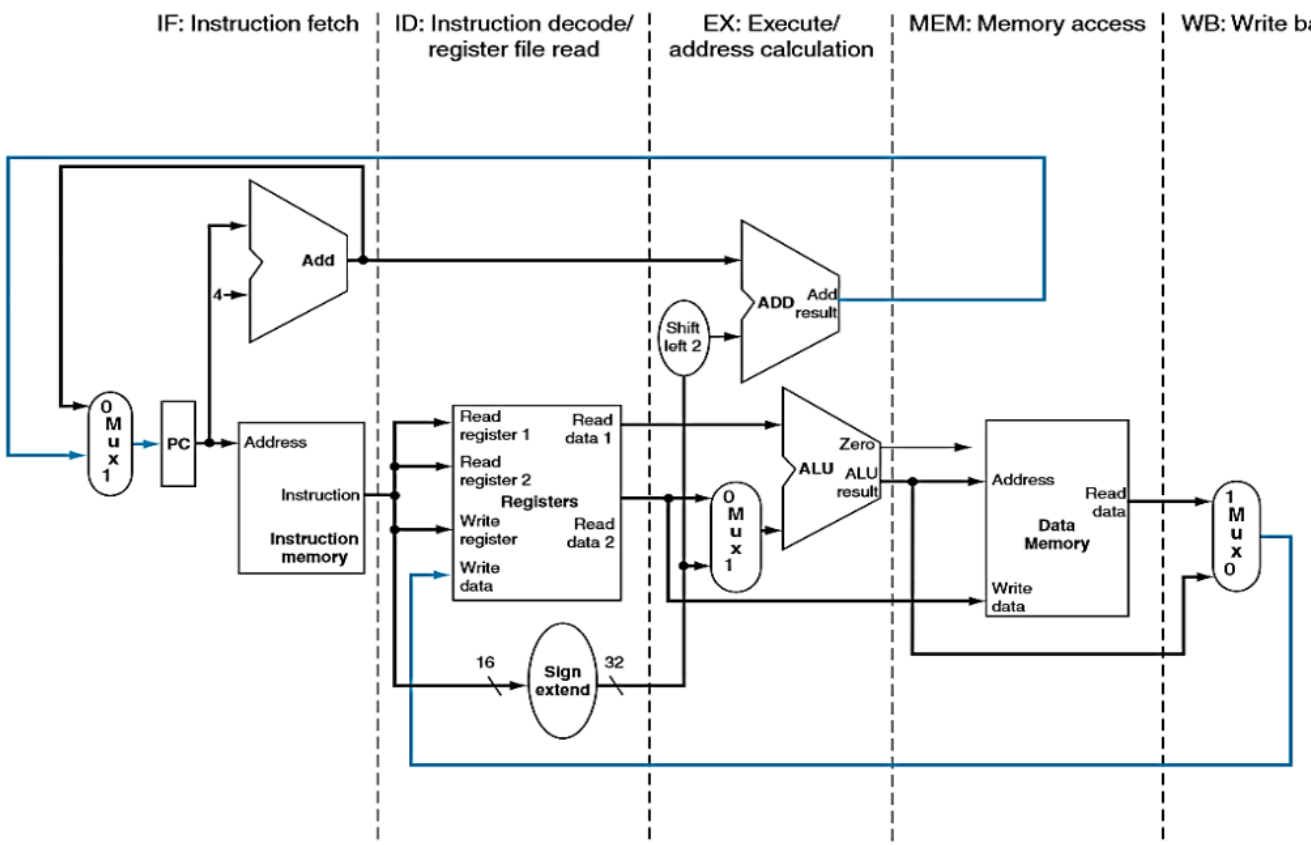
- Not all instructions execute all five stages

Instruction	Steps required				
beq	IF	ID	EX		
R-type	IF	ID	EX		WB
sw	IF	ID	EX	MEM	
lw	IF	ID	EX	MEM	WB

Pipelining Stages

We start by taking the single-cycle datapath and dividing it into 5 stages.

A 5-stage pipeline allows 5 instructions to be executing at once, as long as they are in different stages.

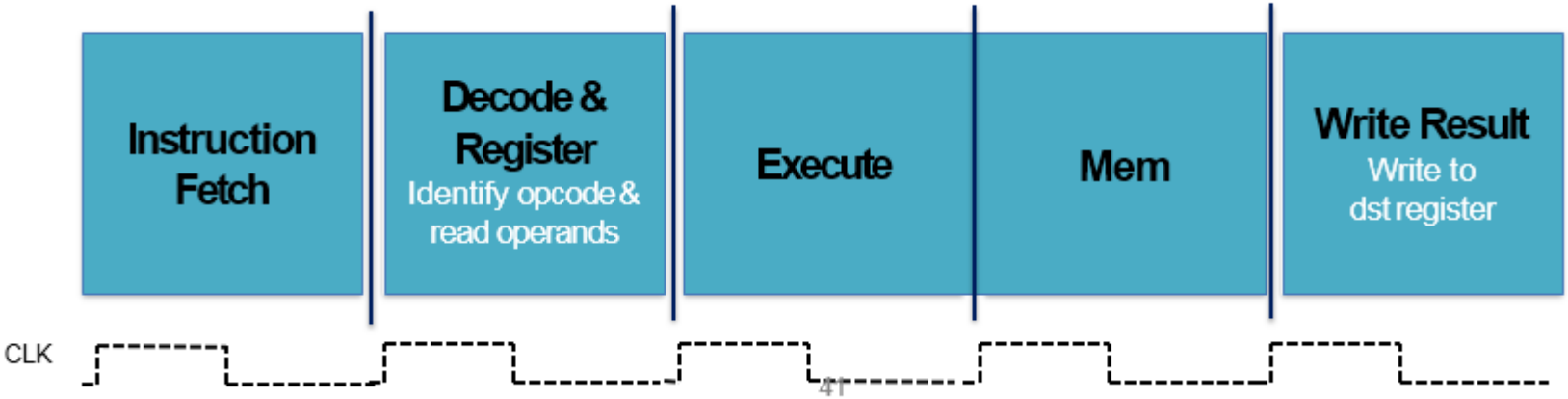


The single cycle datapath with the pipeline stages identified

5-Stage RISC-V Pipeline

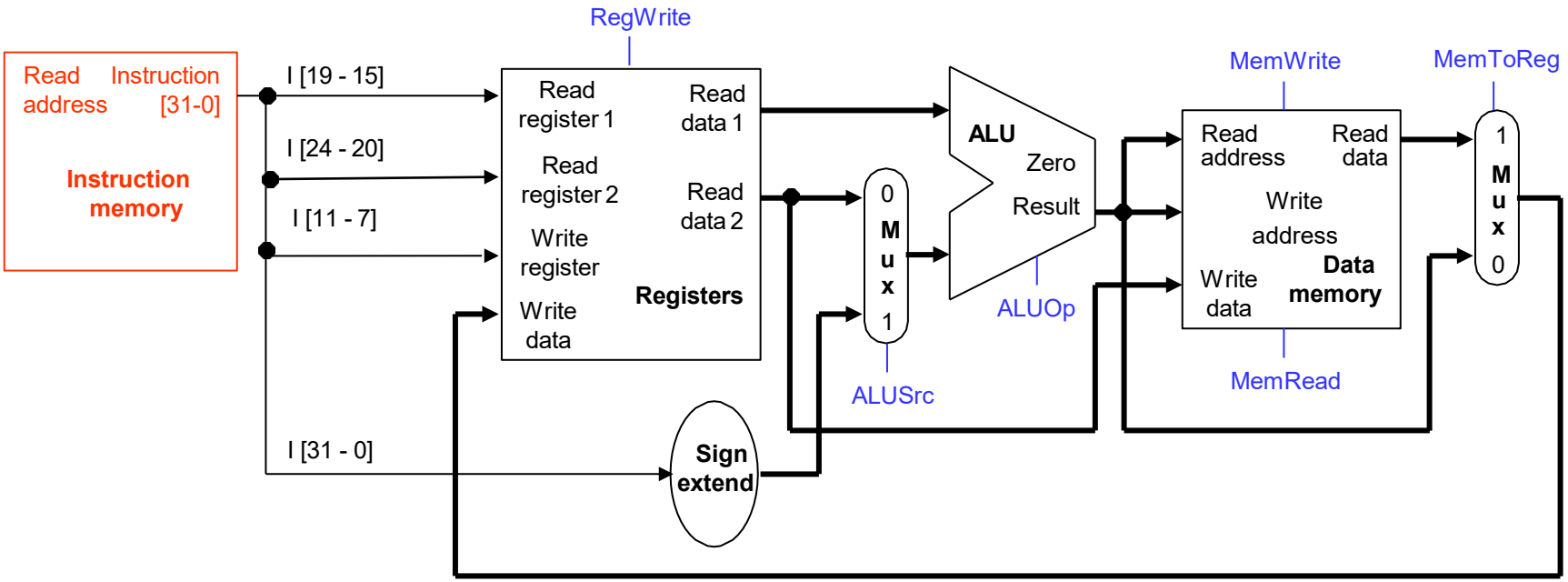
- Entire datapath can be broken into these five stages
  - With each step using separate resources
  - Clock cycle time is changed to the time taken for a stage
    - A single instruction requires 5 cycles to complete
    - However, much shorter clock cycle time

- *Q: What is the CPI now?*
  - CPI is 1 as workload goes infinite.



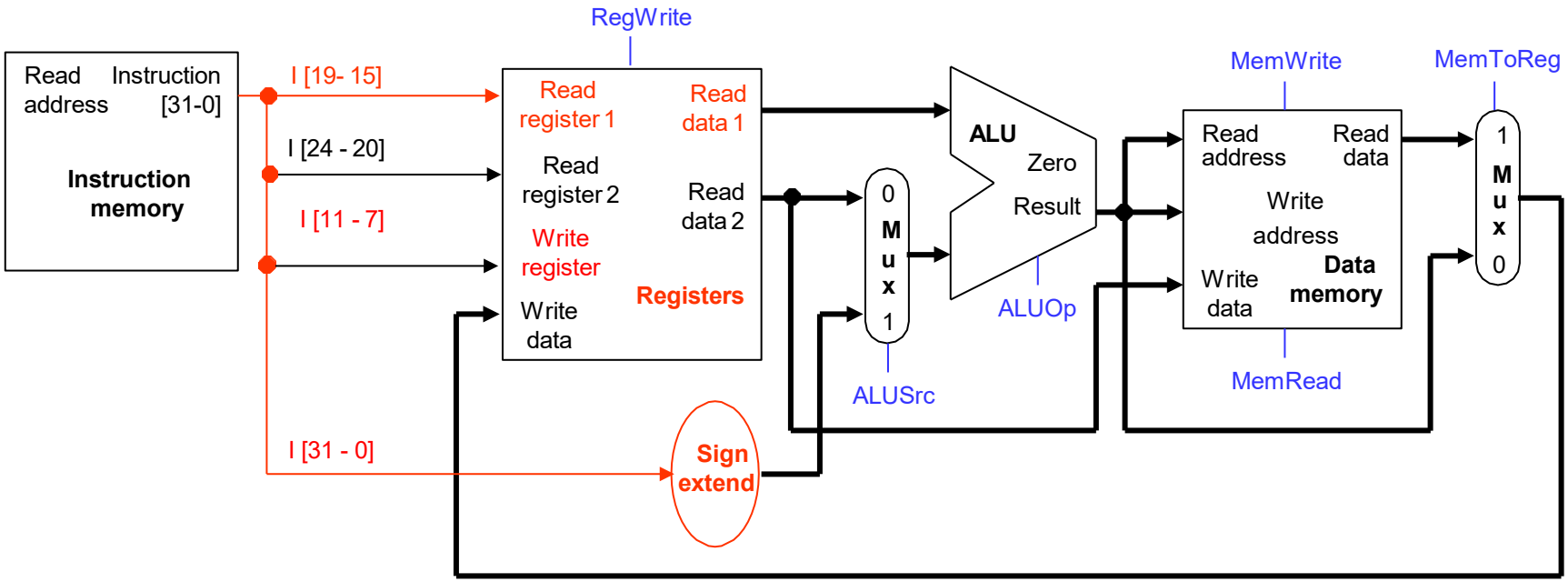
Review: Instruction Fetch (IF)

- Let's quickly review how lw is executed in the single-cycle datapath.
  - We'll ignore PC incrementing and branching for now.
- In the Instruction Fetch (IF) step, we read the instruction memory.



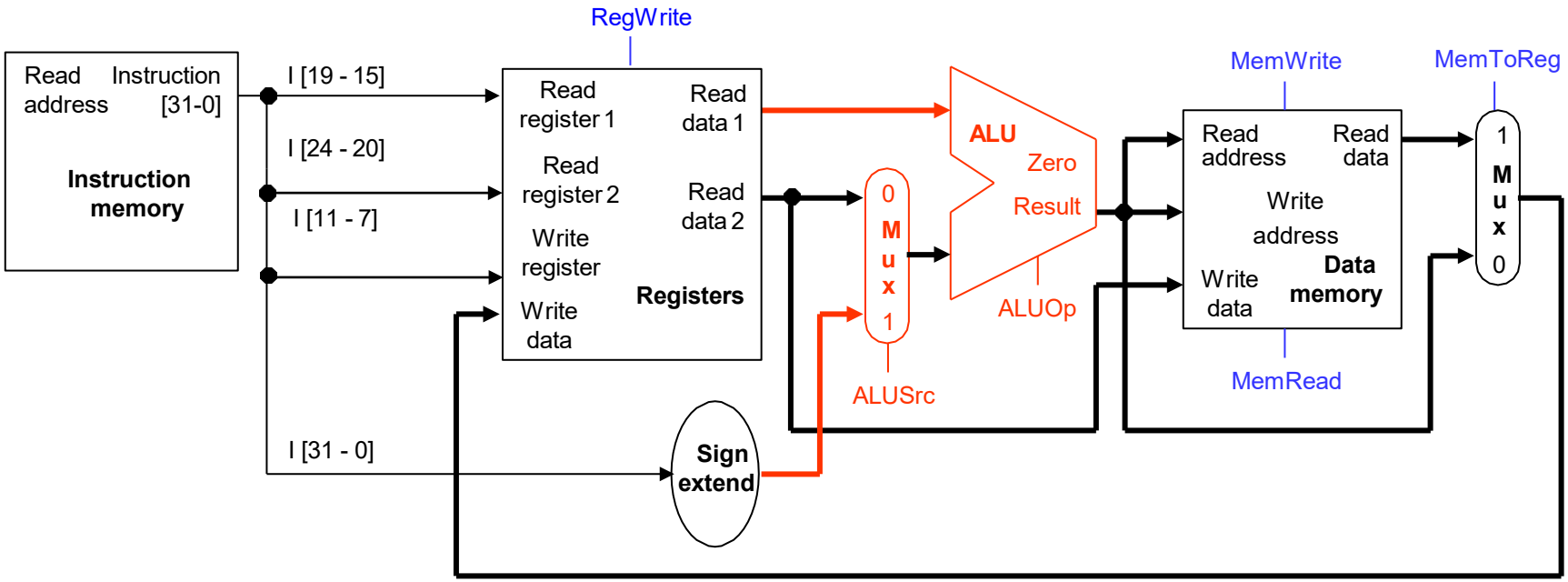
Instruction Decode (ID)

- **The Instruction Decode (ID) step reads the source register from the register file.**



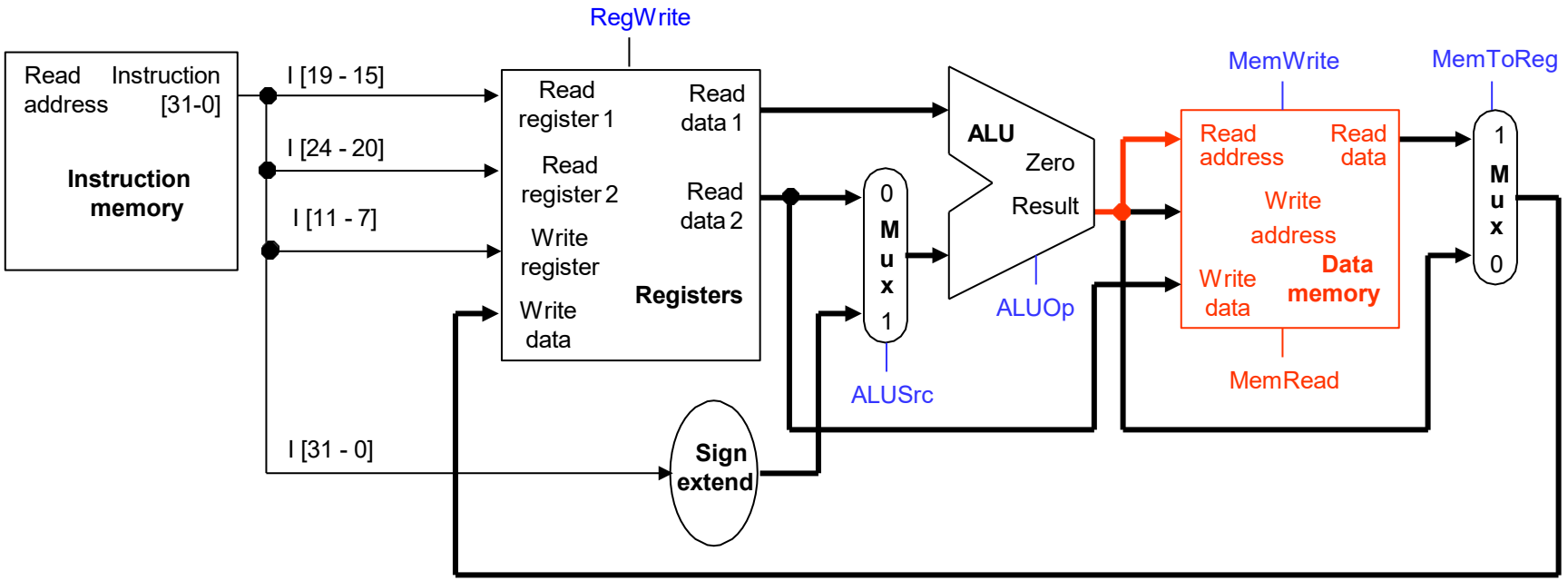
Execute (EX)

- **The third step, Execute (EX), computes the effective memory address from the source register and the instruction's constant field.**



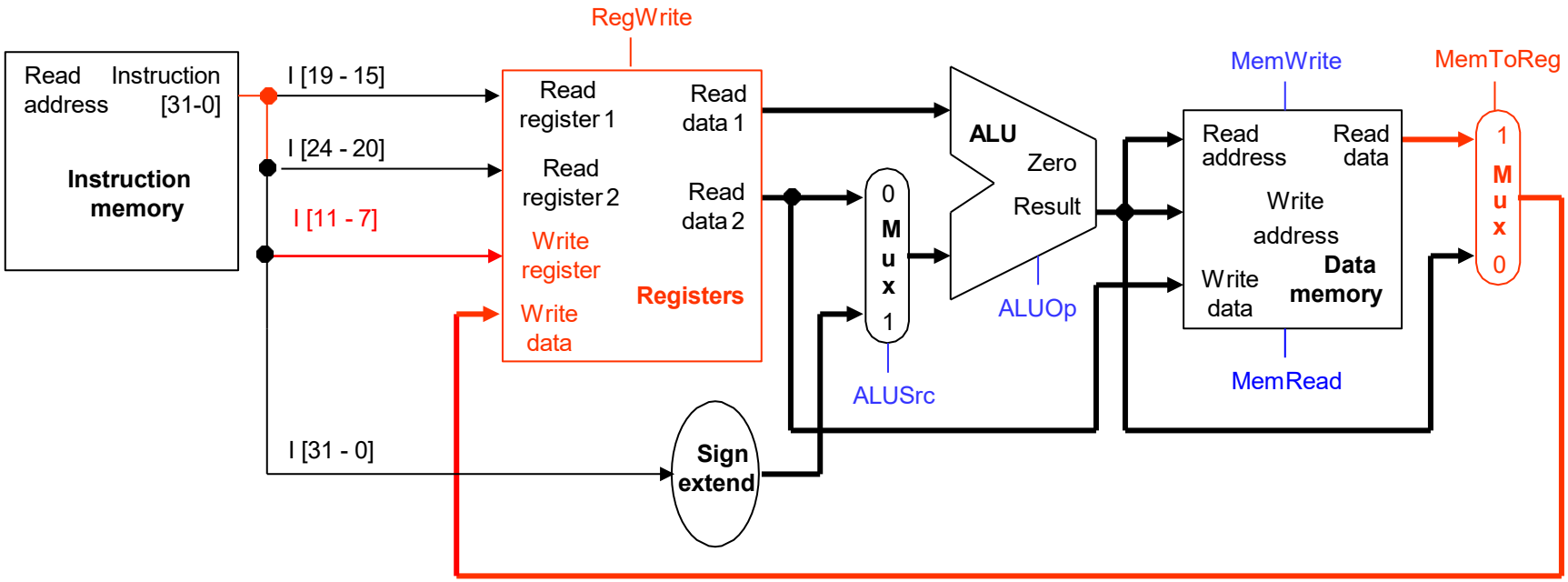
Memory (MEM)

- **The Memory (MEM) step involves reading the data memory, from the address computed by the ALU.**



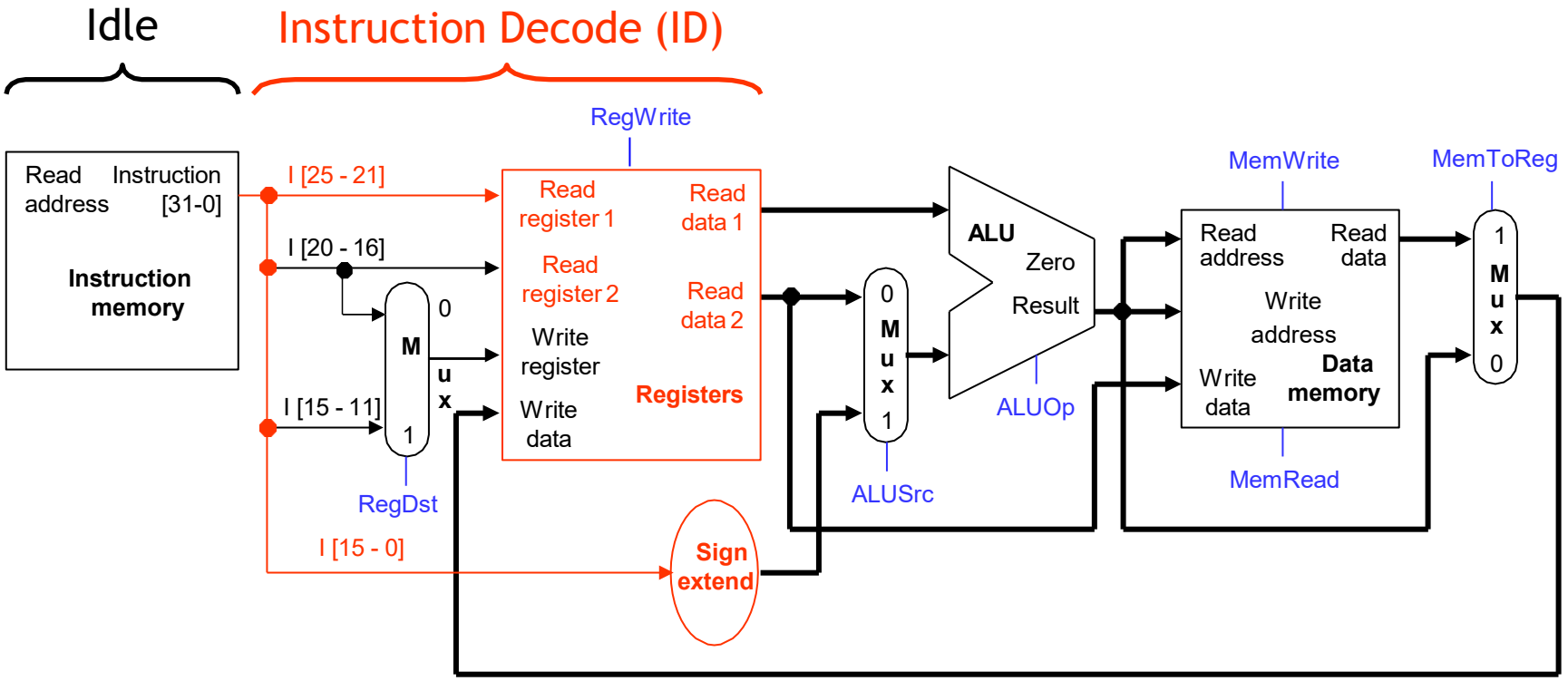
Writeback (WB)

- **Finally, in the Writeback (WB) step, the memory value is stored into the destination register.**



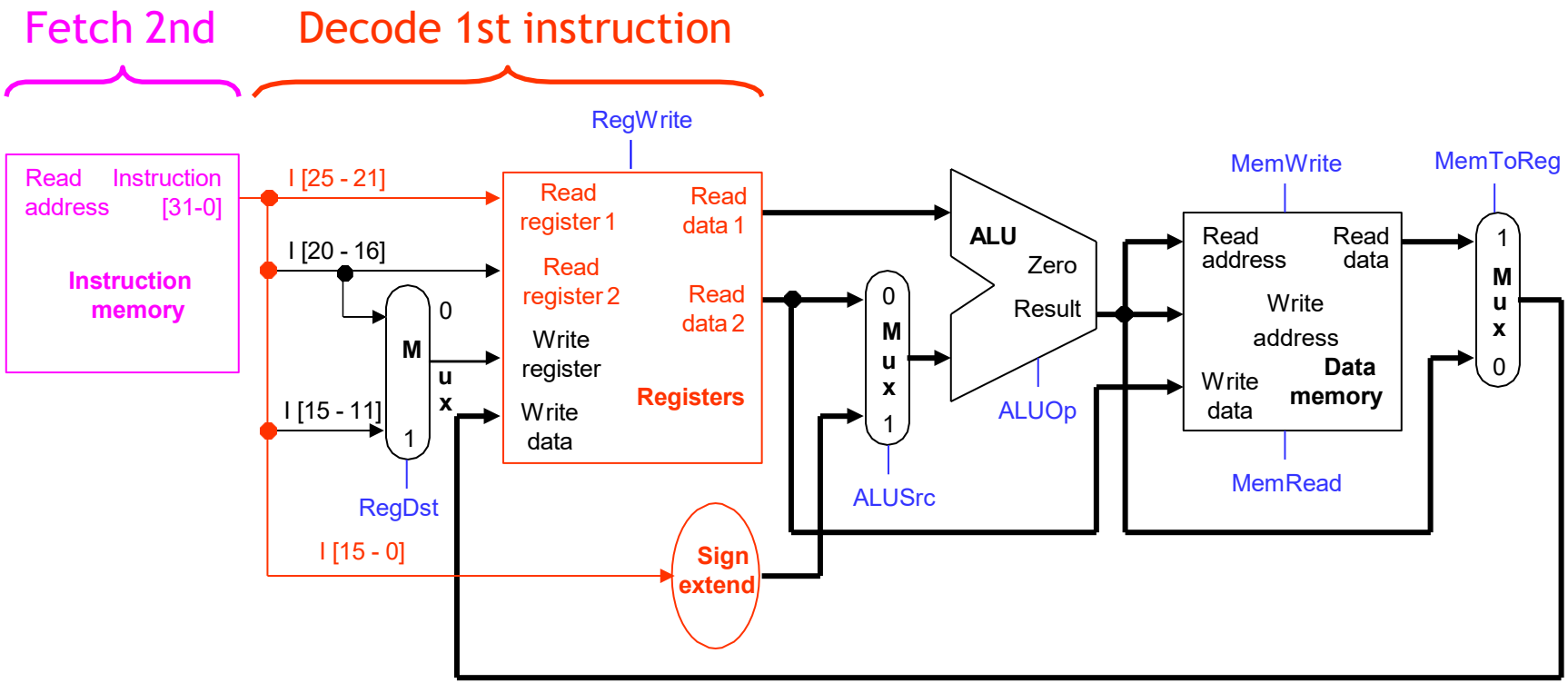
Putting those slackers to work

- **We shouldn't have to wait for the entire instruction to complete before we can re-use the functional units.**
- **For example, the instruction memory is free in the Instruction Decode step as shown below, so...**



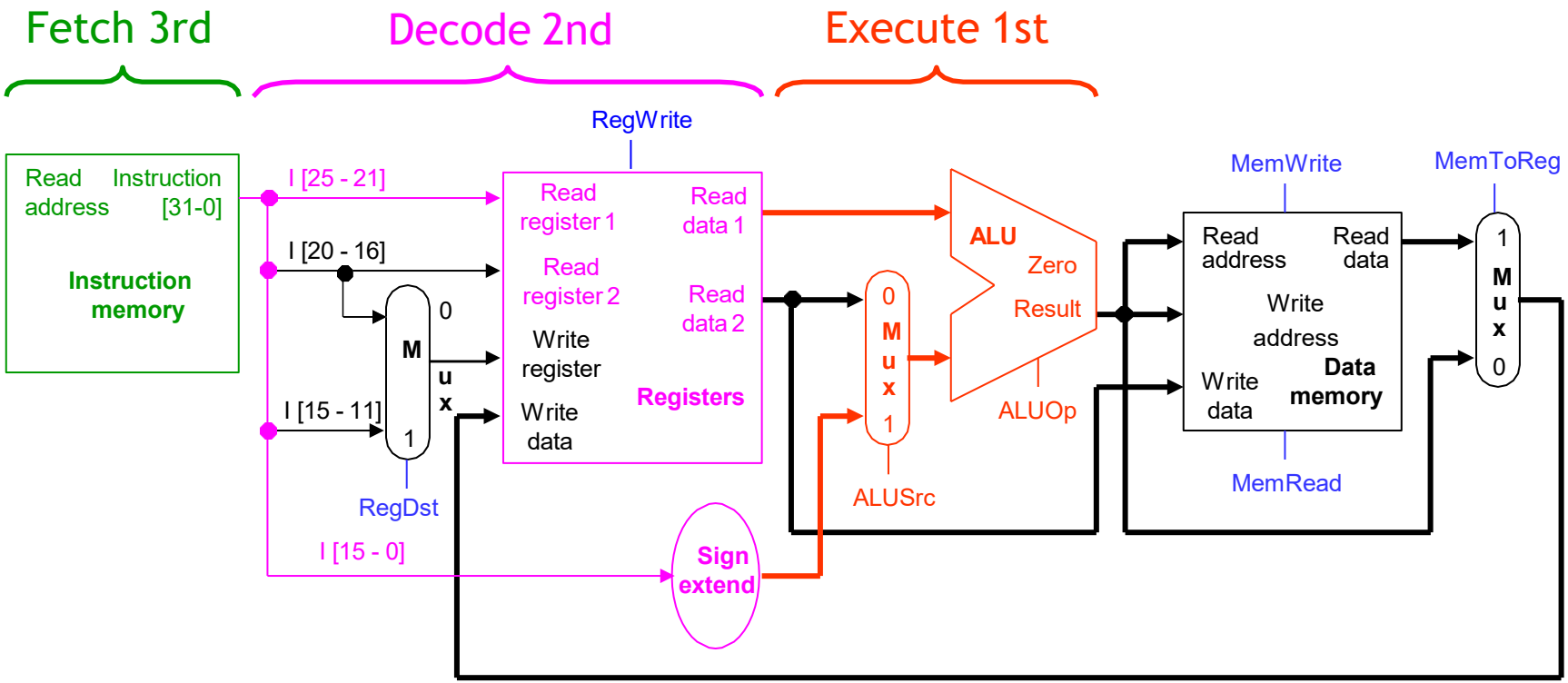
Decoding and fetching together

- **Why don't we go ahead and fetch the *next* instruction while we're decoding the first one?**



Executing, decoding and fetching

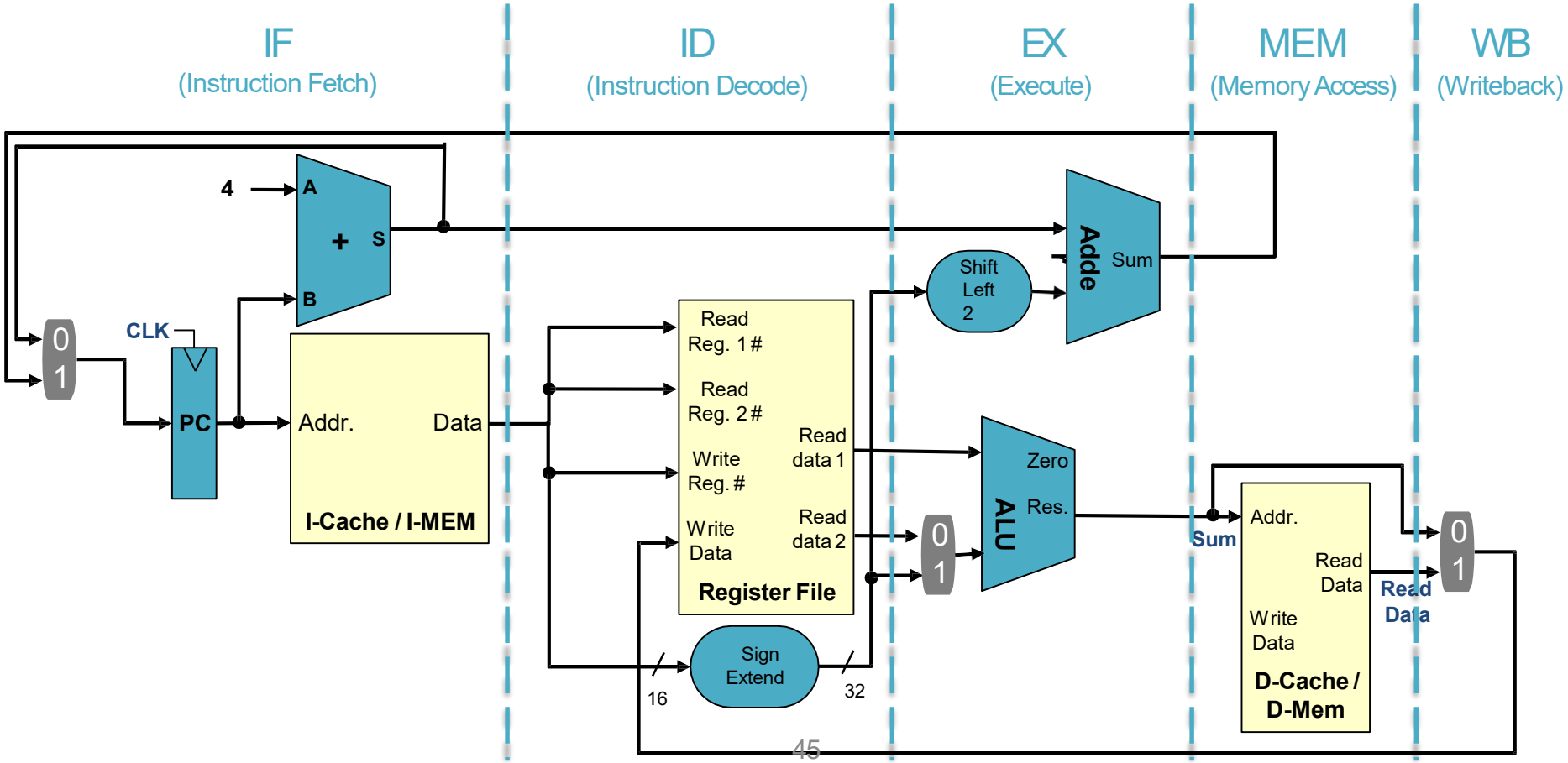
- **Similarly, once the first instruction enters its Execute stage, we can go ahead and decode the second instruction.**
- **But now the instruction memory is free again, so we can fetch the third instruction!**



Design Pipeline & Walk-Through

Basic 5 Stage Pipeline

- Same structure as single cycle but now broken into 5 stages



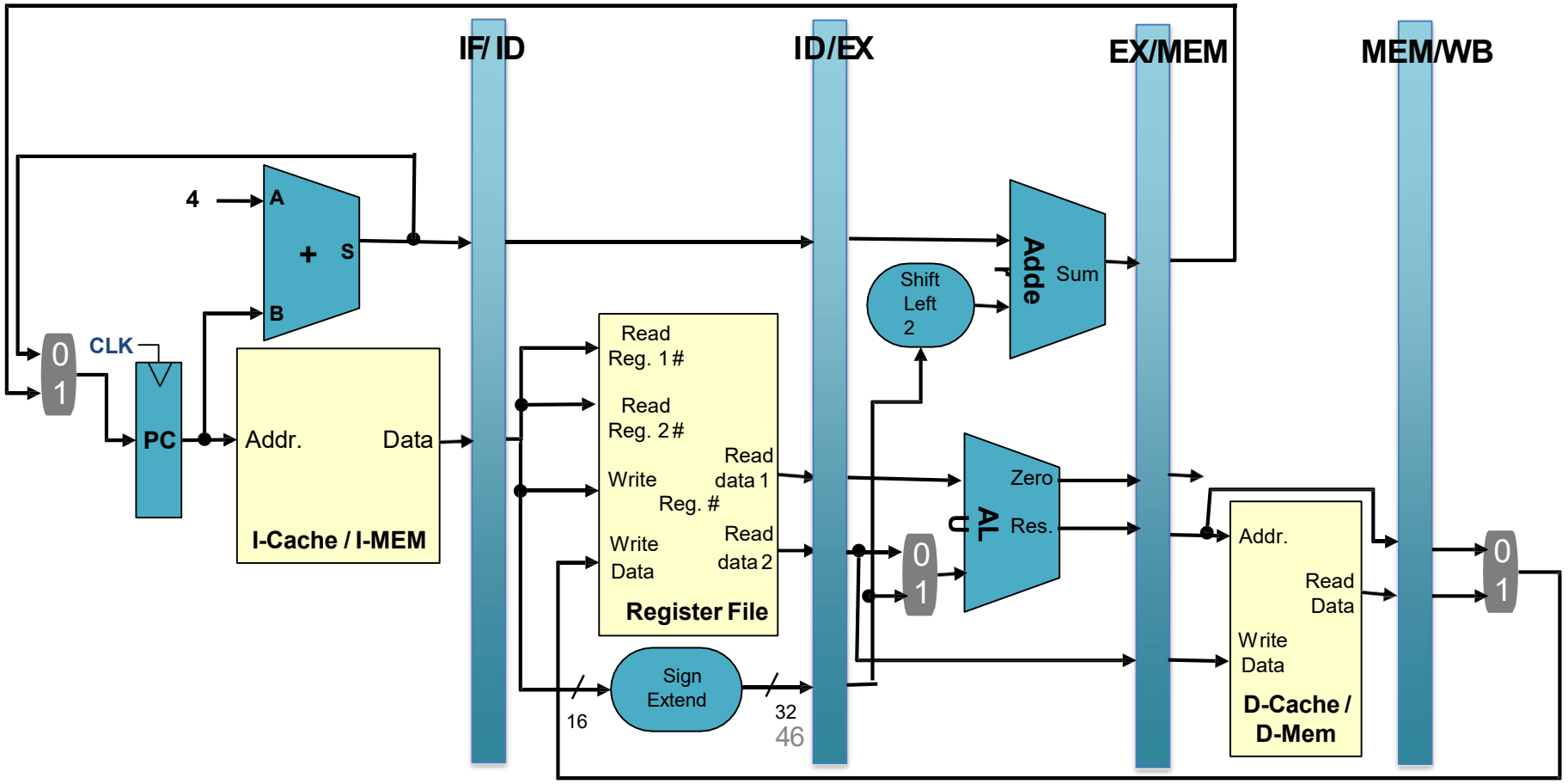
Pipeline Registers

- There's a lot of information to save, however. We'll simplify our diagrams by drawing just one big pipeline register between each stage.
  - The registers are named for the stages they connect.

IF/ID **ID/EX** **EX/MEM** **MEM/WB**

- No register is needed after the WB stage, because after WB the instruction is done.

Pipeline Registers



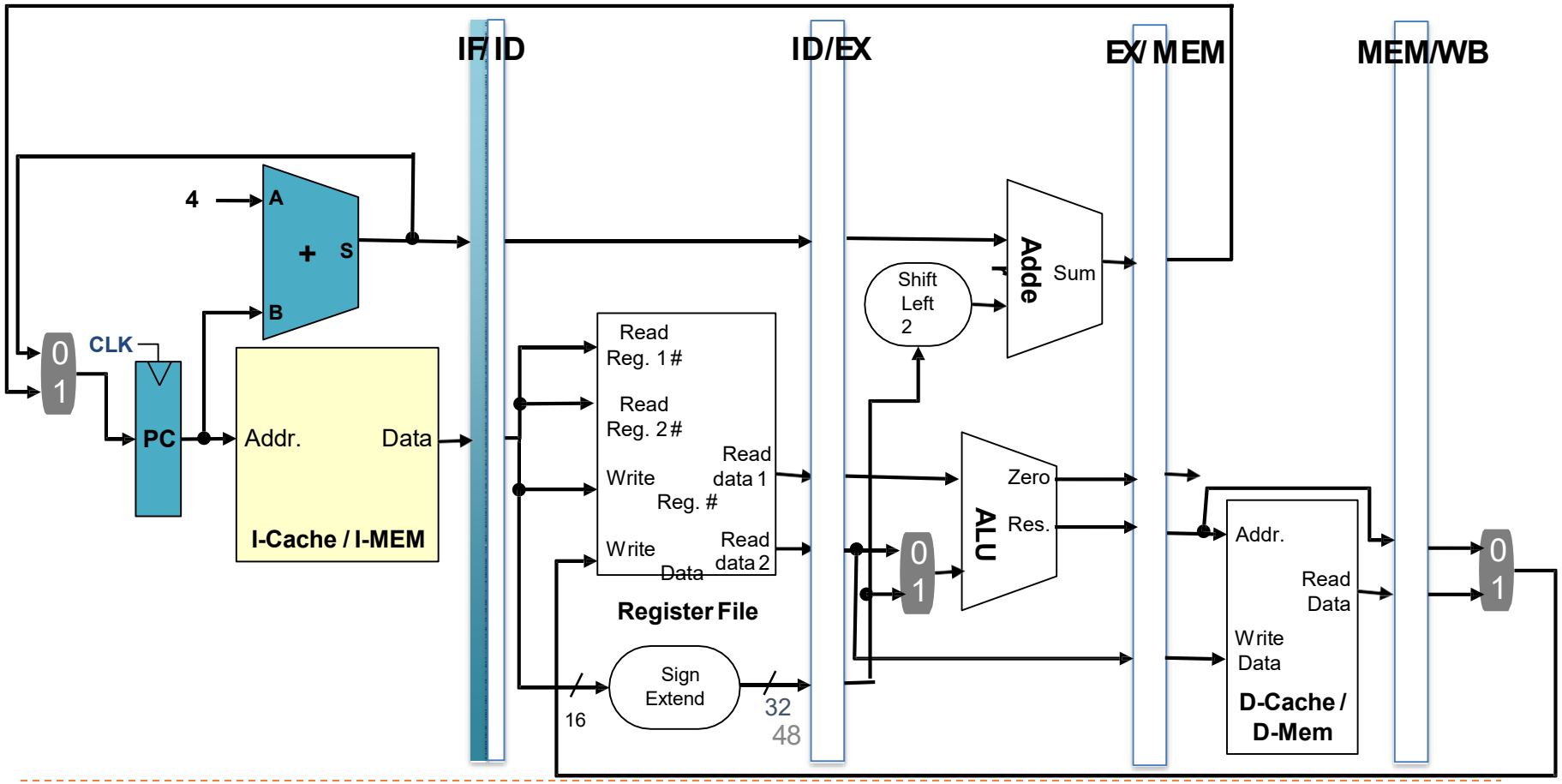
The pipelined version of the datapath. The pipeline registers, in color.

Walk-through Examples

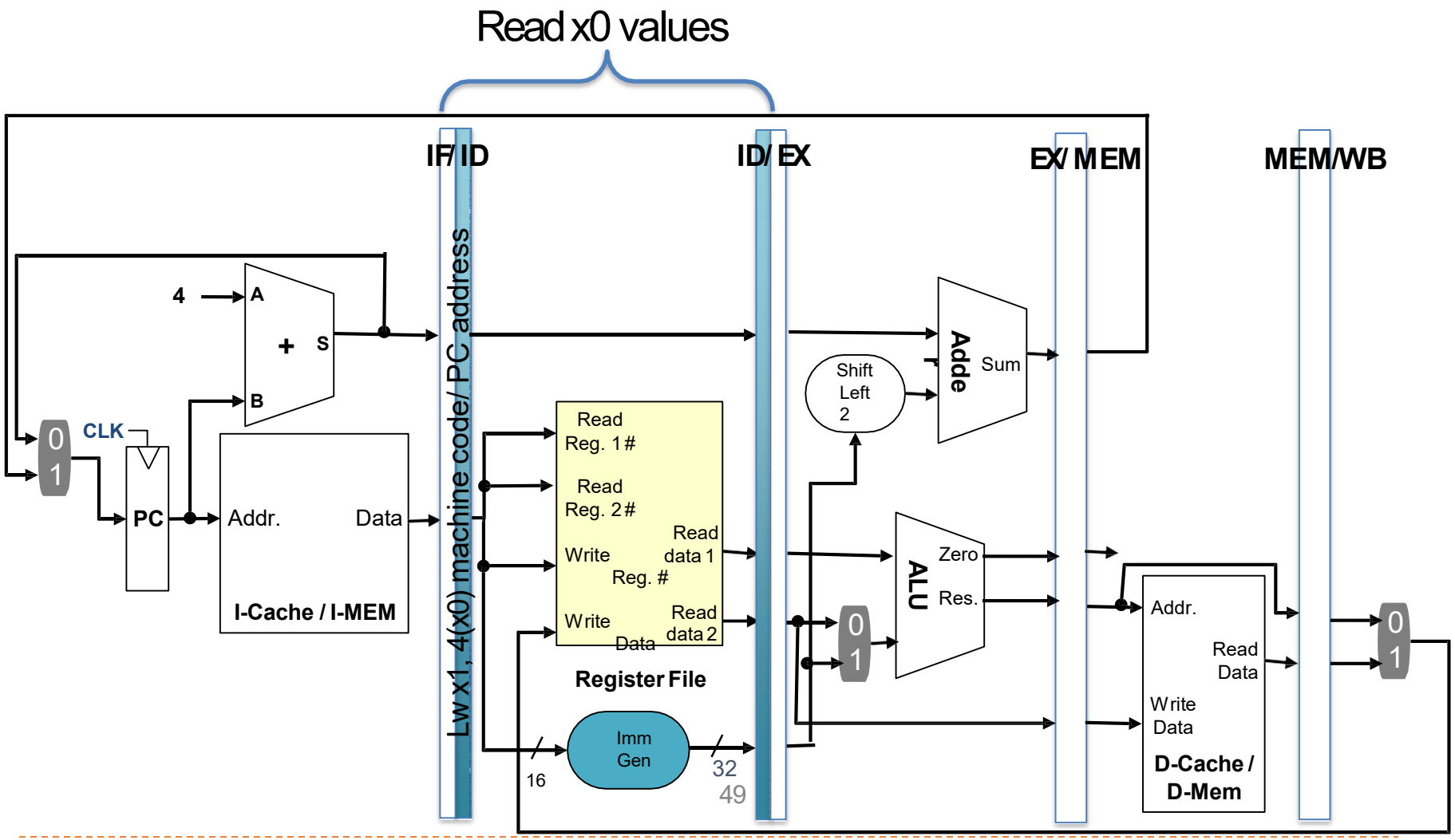
- lw x1, 4(x0)
- add x4,x5,x6
- beq x0,x1,LOOP

lw x1, 4(x0)

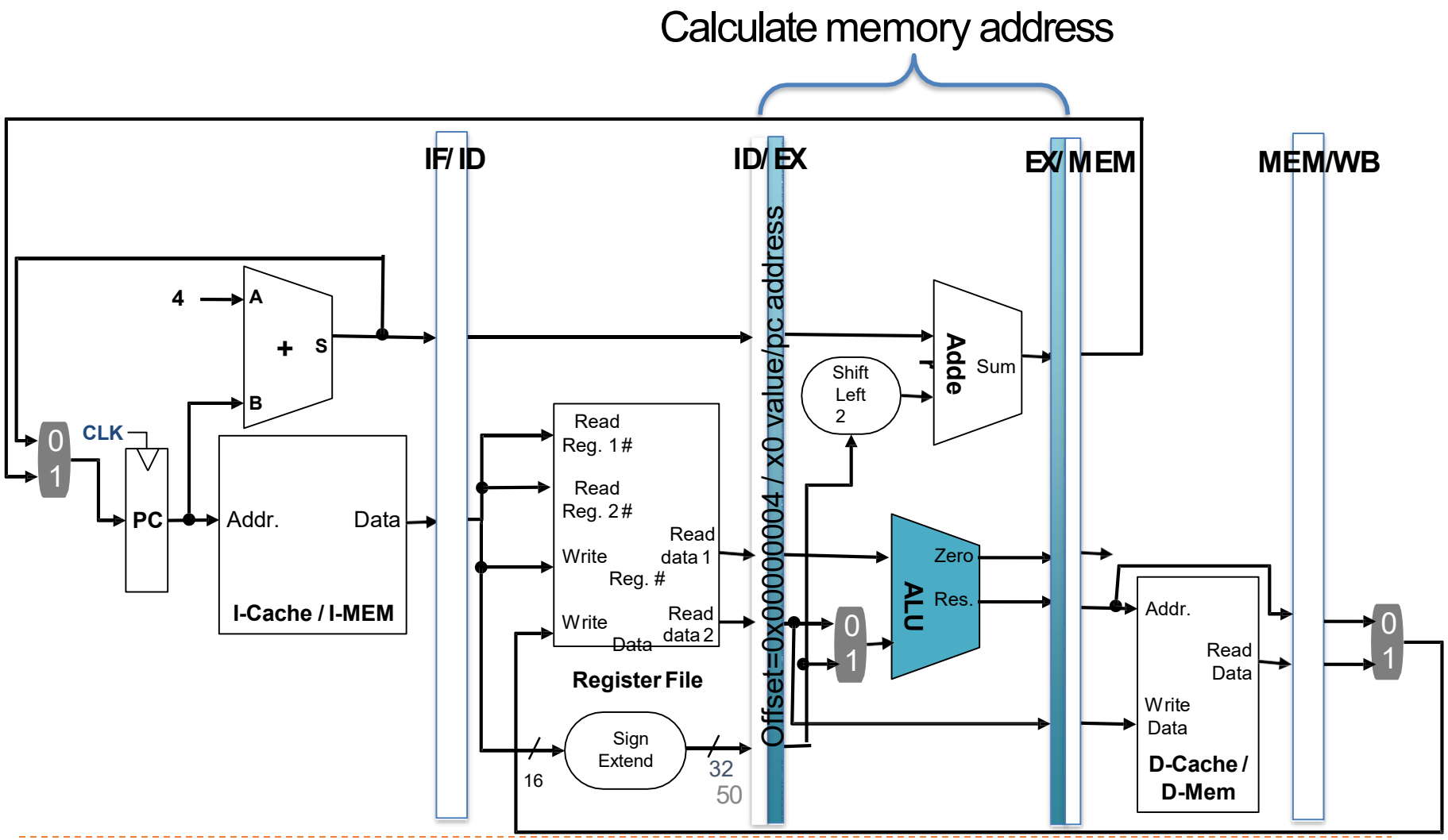
Fetch inst and increment PC



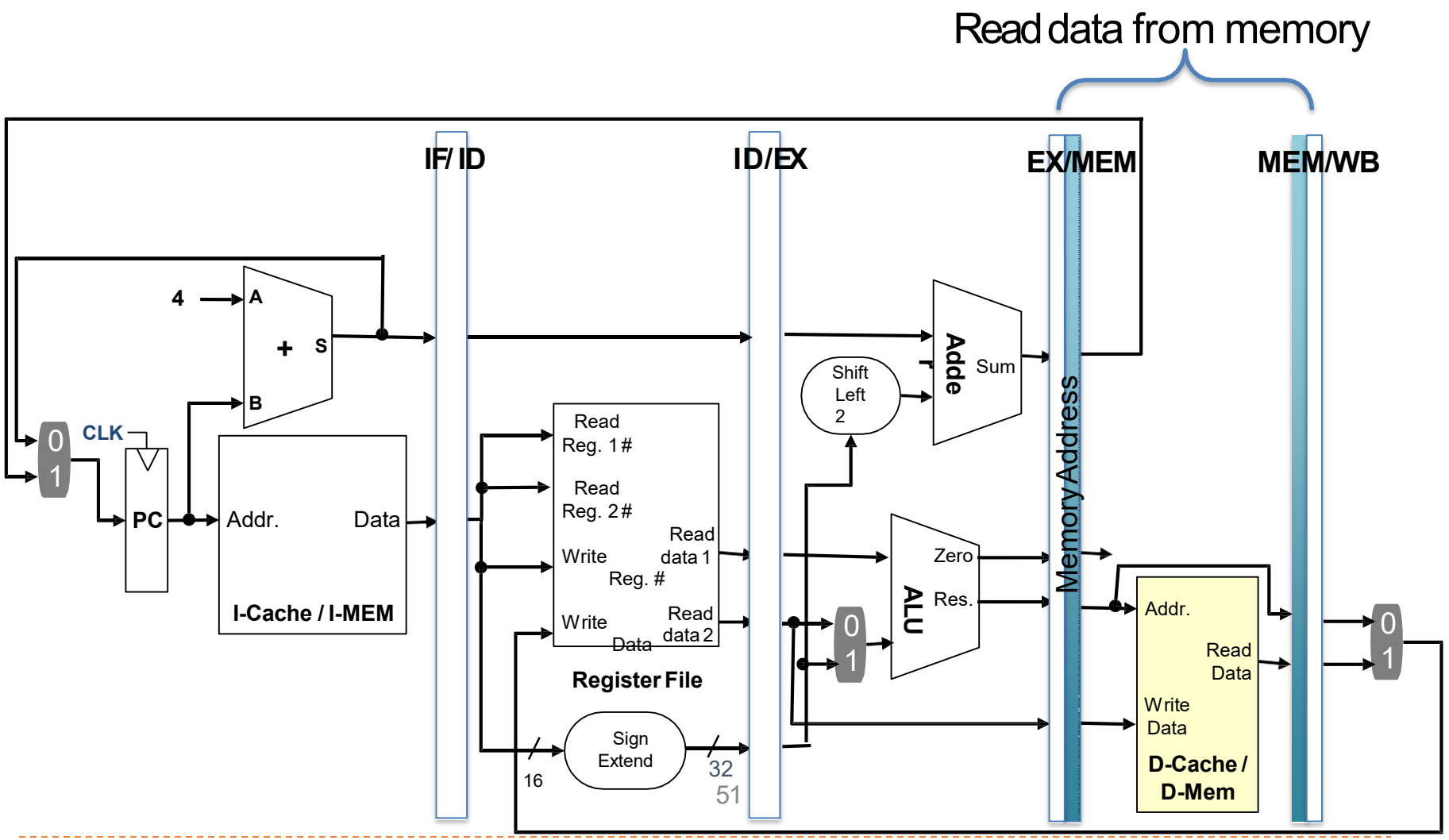
lw x1, 4(x0)


Read x0 values
IF/ID
ID/ EX
EX/ MEM
MEM/WB
Lw x1, 4(x0) machine code/ PC address
4
A
+
S
B
CLK
0
1
PC
Addr.
Data
I-Cache / I-MEM
Read
Reg. 1#
Read
Reg. 2#
Write
Reg. #
Write
Data
Read
data 1
Read
data 2
Register File
Imm
Gen
16
32
Shift
Left
2
Adde
r
Sum
ALU
Zero
Res.
0
1
Addr.
Read
Data
Write
Data
D-Cache /
D-Mem
0
1

lw x1, 4(x0)

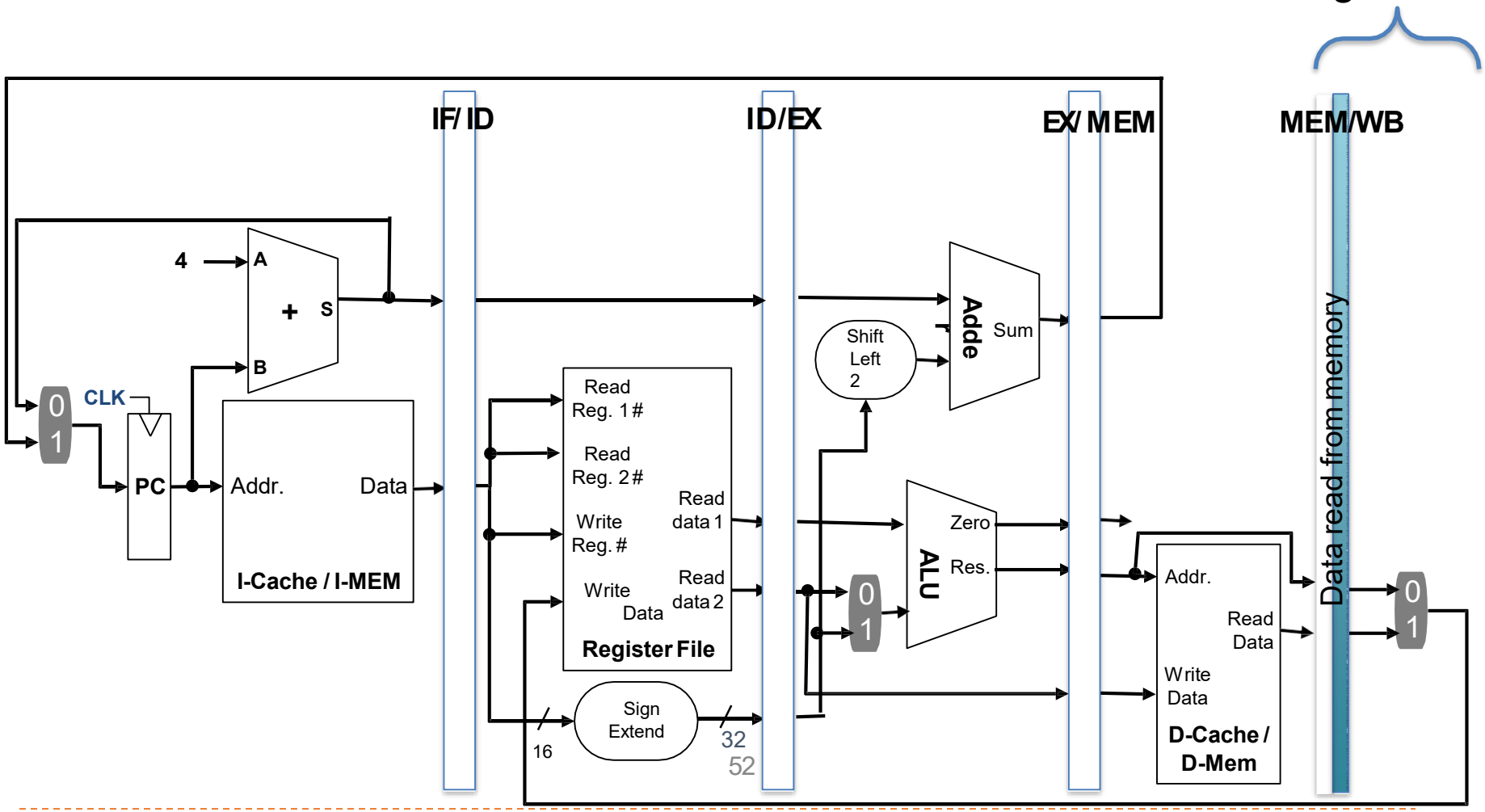

Calculate memory address
IF/ID
ID/EX
EX/MEM
MEM/WB
4
A
+
S
B
CLK
0
1
PC
Addr.
Data
I-Cache / I-MEM
Read Reg. 1#
Read Reg. 2#
Write Reg. #
Write Data
Read data 1
Read data 2
Register File
Sign Extend
16
32
Offset=0x00000004 / x0 value/pc address
Shift Left 2
Adde
Sum
Zero
ALU
Res.
Addr.
Read Data
Write Data
D-Cache / D-Mem

lw x1, 4(x0)


Read data from memory
IF/ID
ID/EX
EX/MEM
MEM/WB
4
A
+
S
B
CLK
0
1
PC
Addr.
Data
I-Cache / I-MEM
Read Reg. 1#
Read Reg. 2#
Write Reg. #
Write Data
Read data 1
Read data 2
Register File
Sign Extend
16
32
Shift Left 2
Adde
Sum
Zero
ALU
Res.
Memory Address
Addr.
Read Data
Write Data
D-Cache / D-Mem

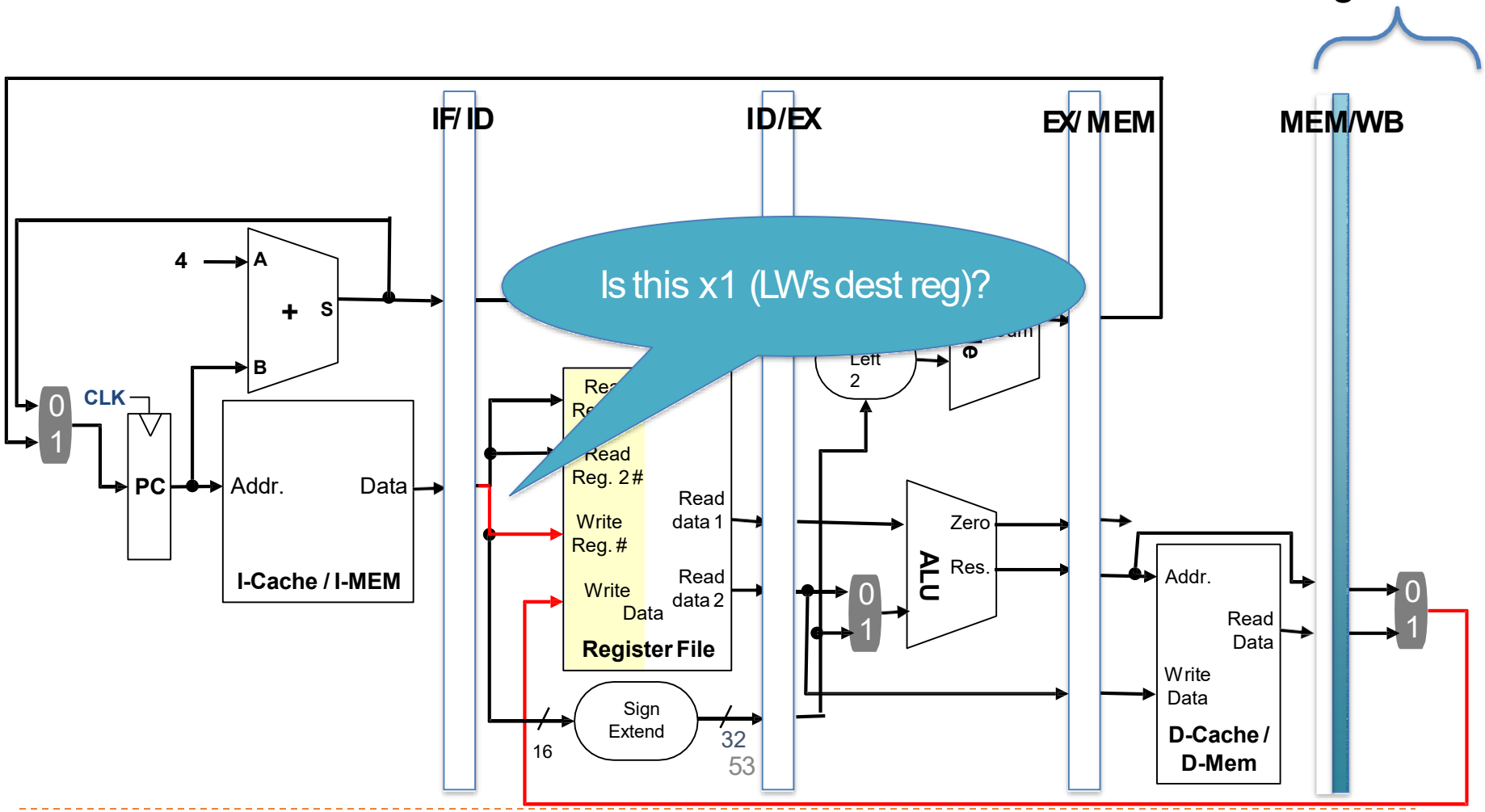
lw x1, 4(x0)

Store the read data to registerfile



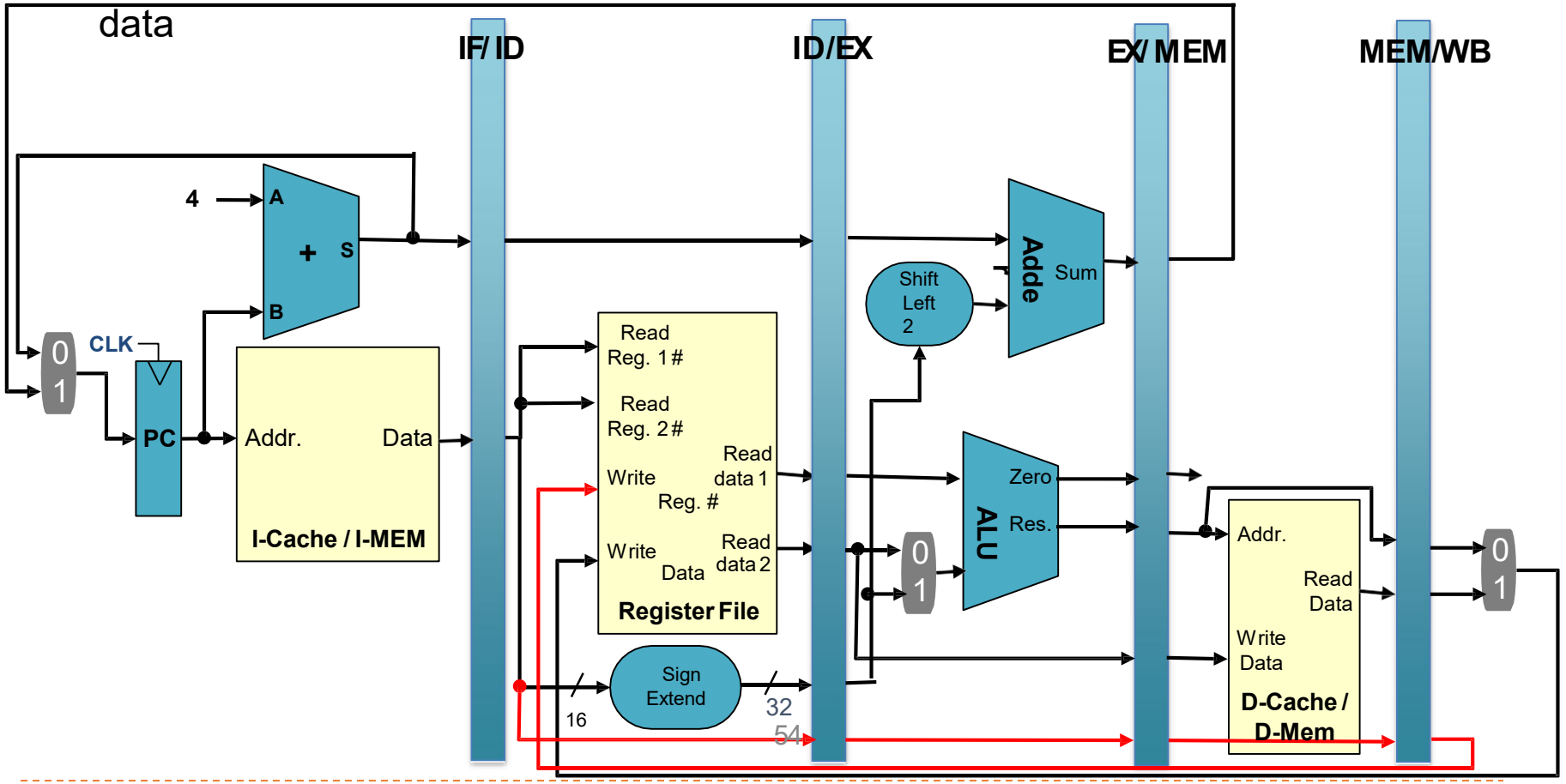
lw x1, 4(x0)

Store the read data to registerfile



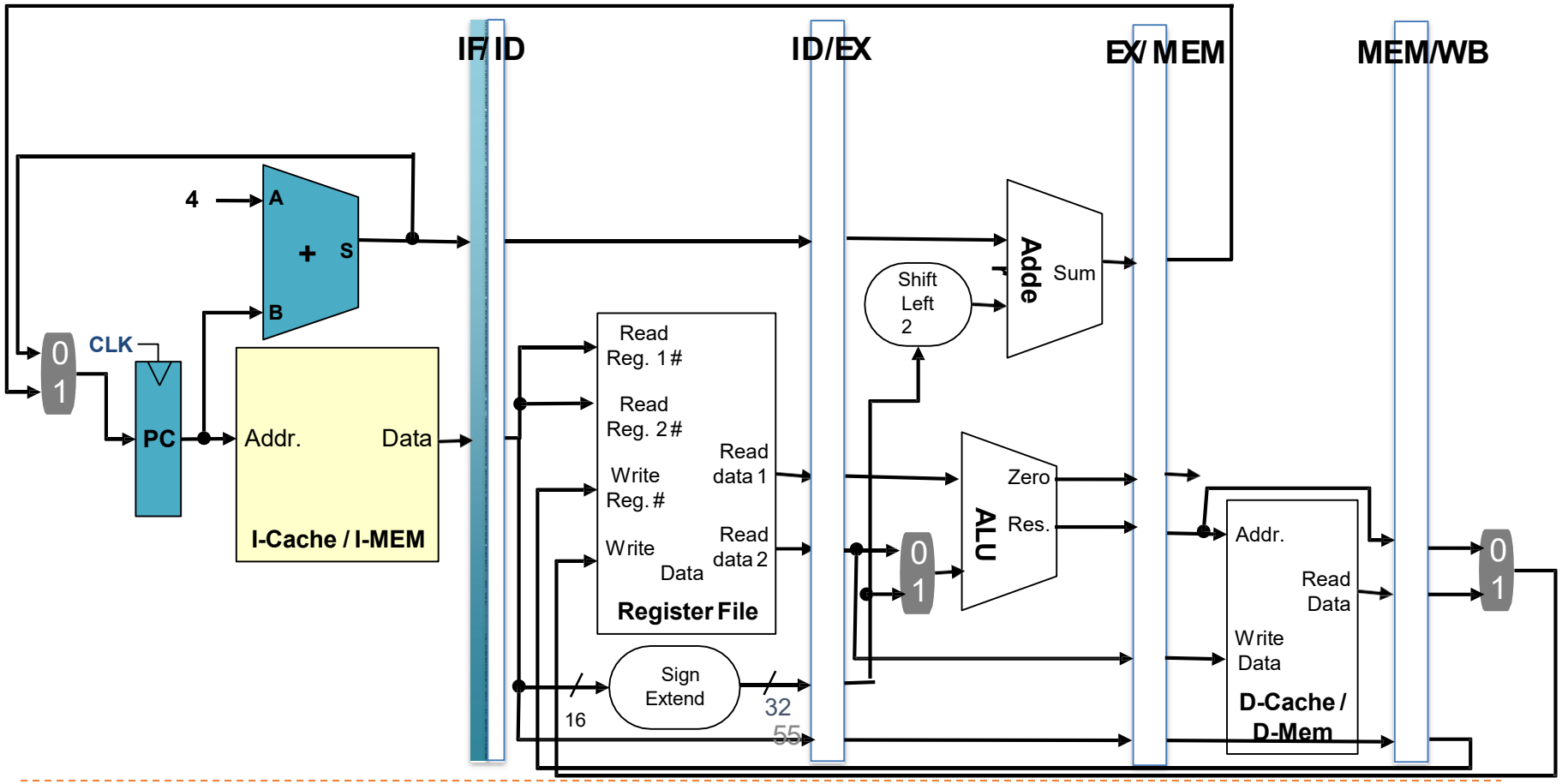
Revised Datapath

- Dest. register number x1 should be stored in the pipeline registers
- The dest. register number x1 is passed to register file with writeback data

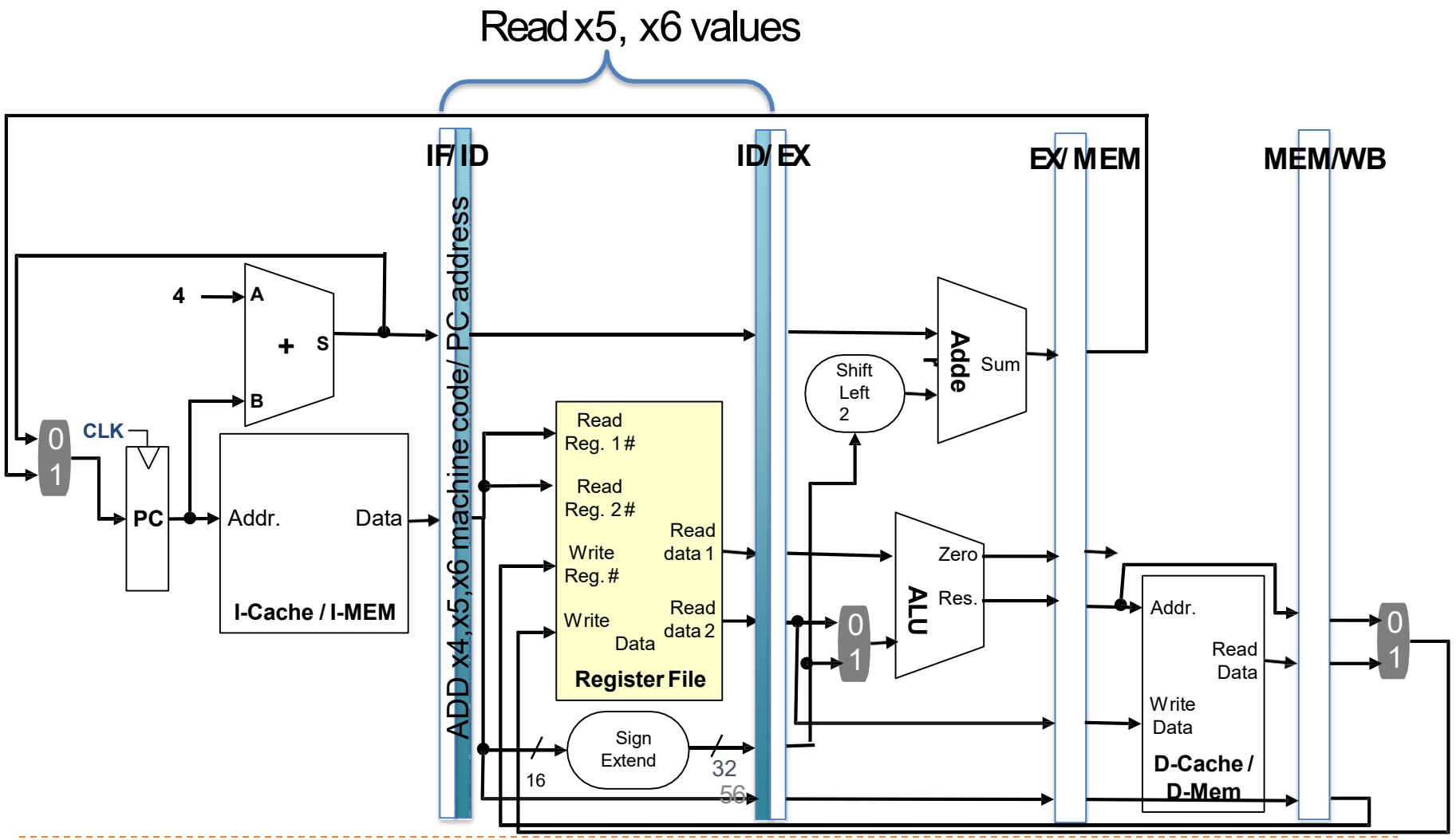


ADD x4,x5,x6

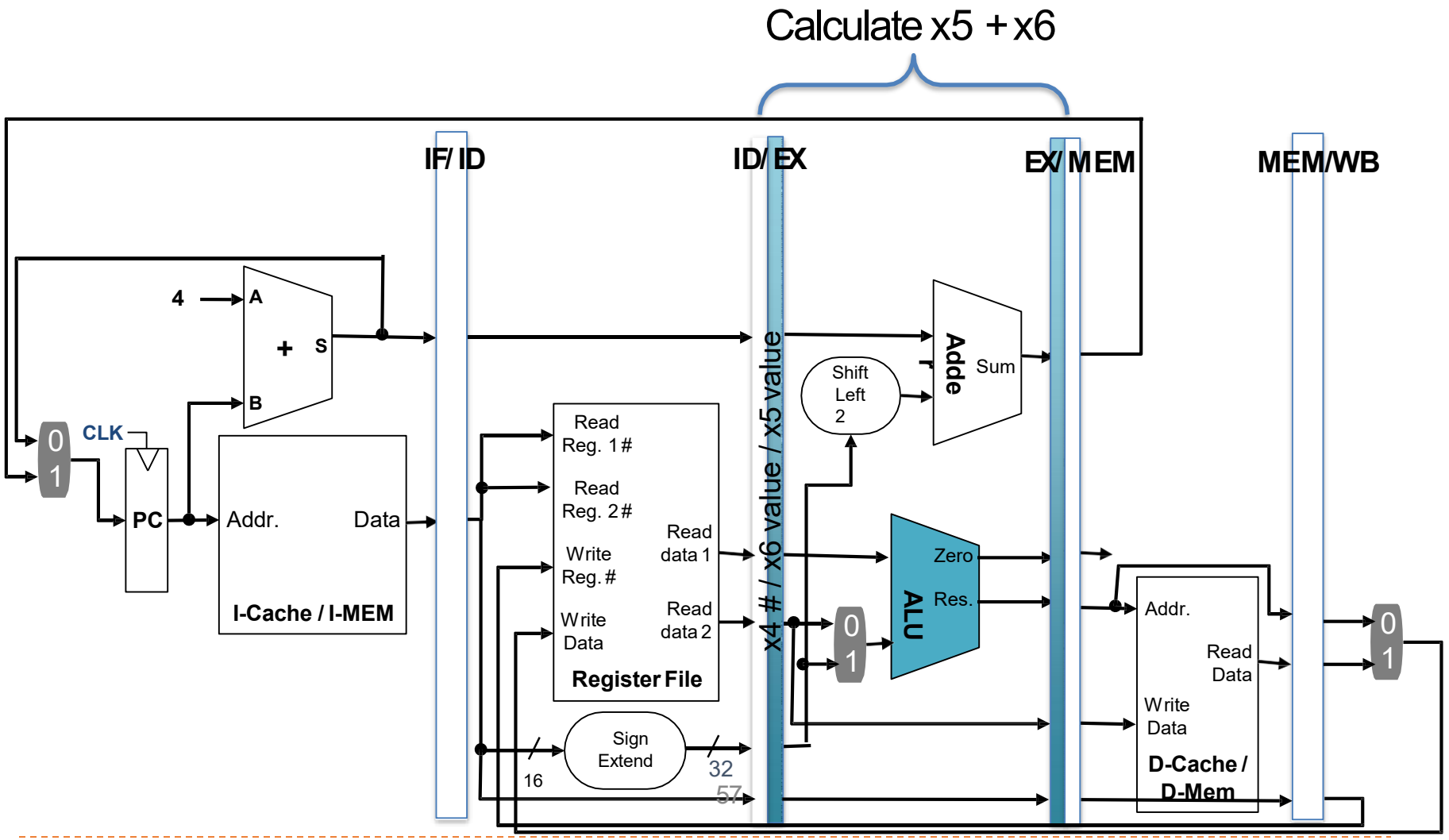
Fetch inst and increment PC



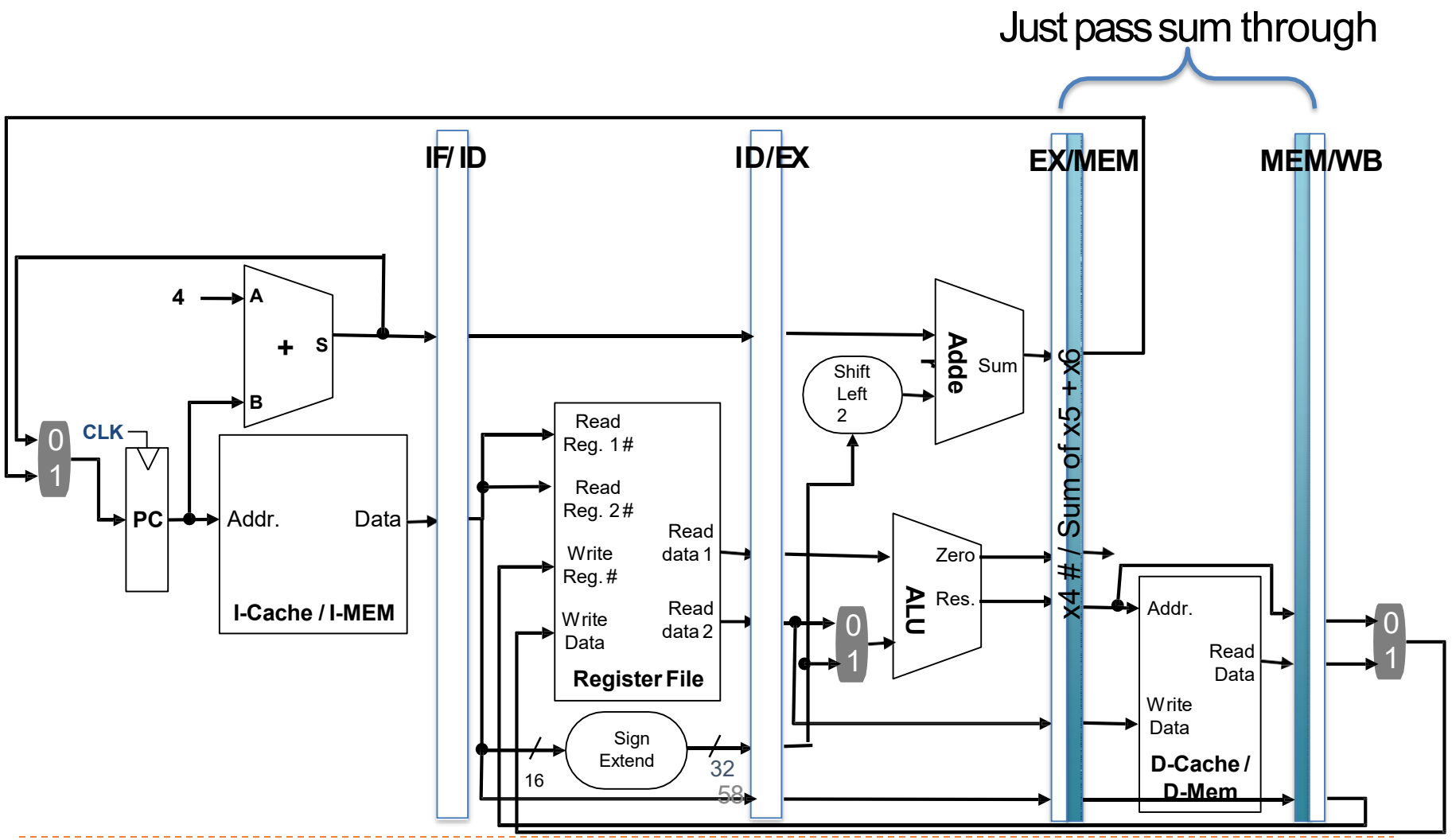
ADD x4,x5,x6


Read x5, x6 values
IF/ID
ID/EX
EX/MEM
MEM/WB
ADD x4,x5,x6 machine code/ PC address
4
A
+
S
B
CLK
0
1
PC
Addr.
Data
I-Cache / I-MEM
Read Reg. 1#
Read Reg. 2#
Write Reg.#
Write Data
Read data 1
Read data 2
Register File
Sign Extend
16
32
Shift Left 2
Adder
Sum
Zero
ALU
Res.
Addr.
Read Data
Write Data
D-Cache / D-Mem

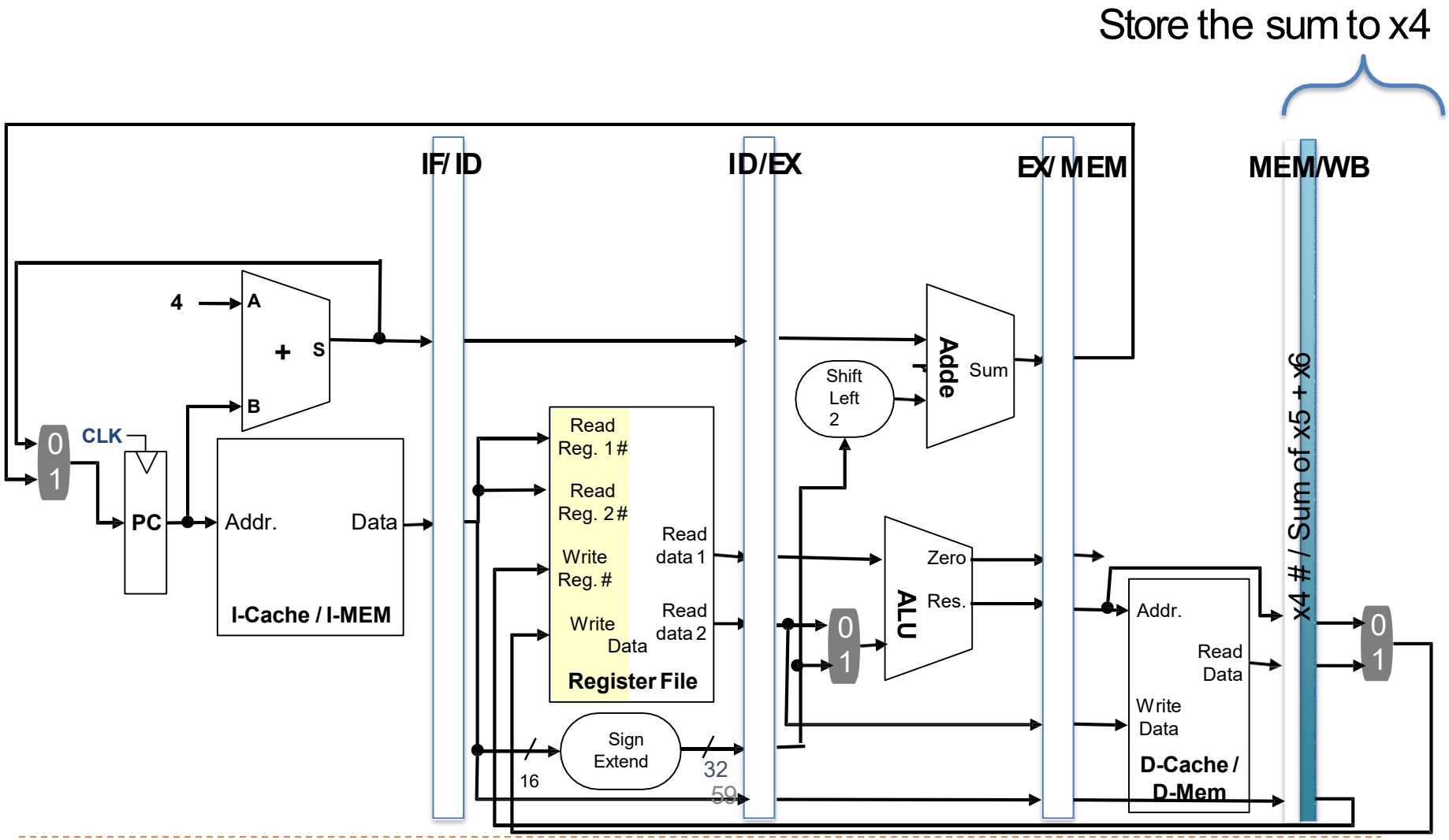
ADD x4,x5,x6


Calculate x5 + x6
IF/ ID
ID/ EX
EX/ MEM
MEM/WB
4
A
+
S
B
CLK
0
1
PC
Addr.
Data
I-Cache / I-MEM
Read Reg. 1#
Read Reg. 2#
Write Reg.#
Write Data
Read data 1
Read data 2
Register File
Sign Extend
16
32
x4 # / x6 value / x5 value
Shift Left 2
Adde
Sum
Zero
Res.
ALU
Addr.
Read Data
Write Data
D-Cache / D-Mem

ADD x4,x5,x6

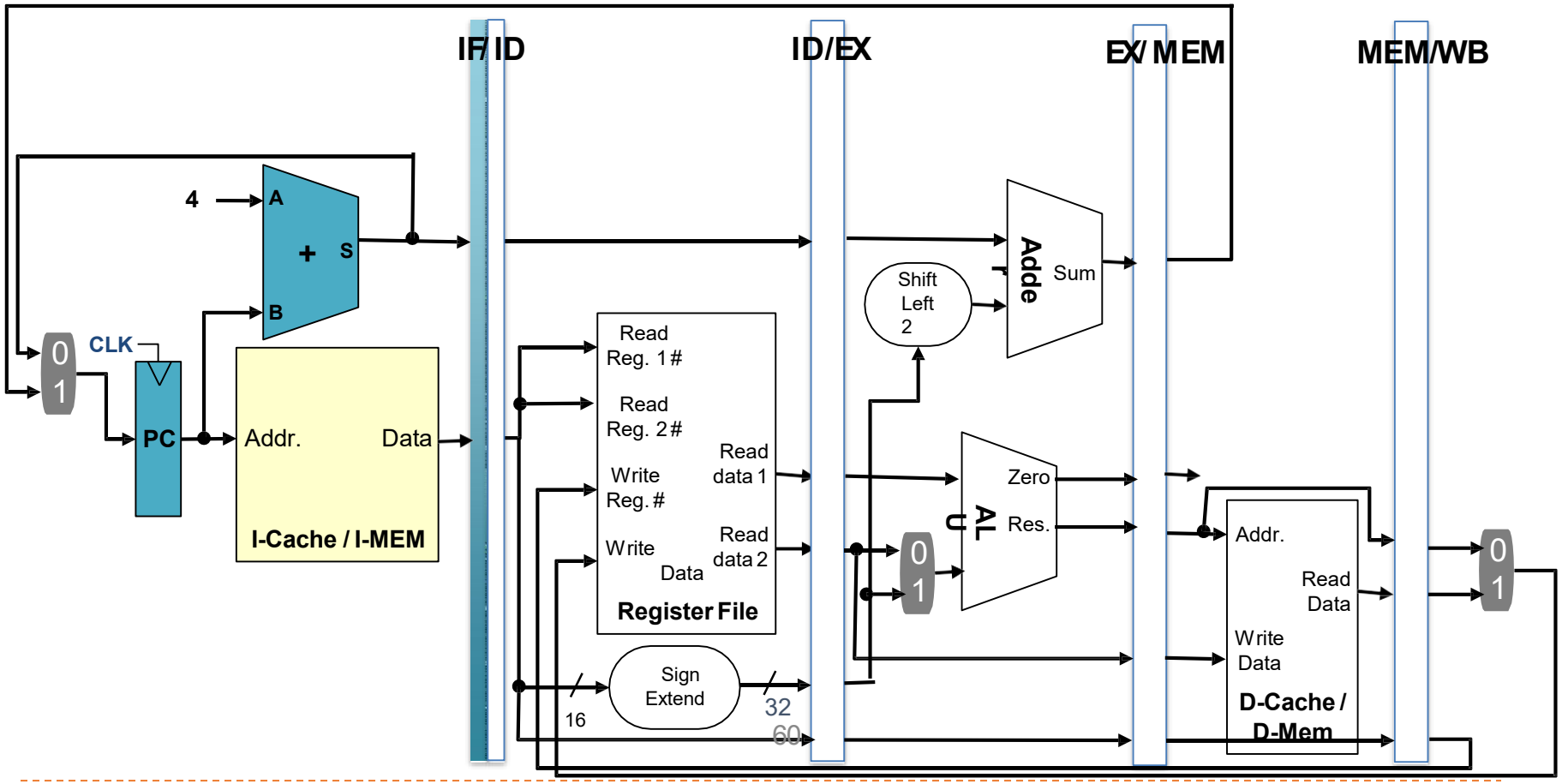

Just pass sum through
IF/ID
ID/EX
EX/MEM
MEM/WB
4
A
+
S
B
CLK
0
1
PC
Addr.
Data
I-Cache / I-MEM
Read Reg. 1#
Read Reg. 2#
Write Reg.#
Write Data
Read data 1
Read data 2
Register File
Sign Extend
16
32
Shift Left 2
Adde
Sum
Zero
ALU
Res.
x4 # / Sum of x5 + x6
Addr.
Read Data
Write Data
D-Cache / D-Mem

ADD x4,x5,x6

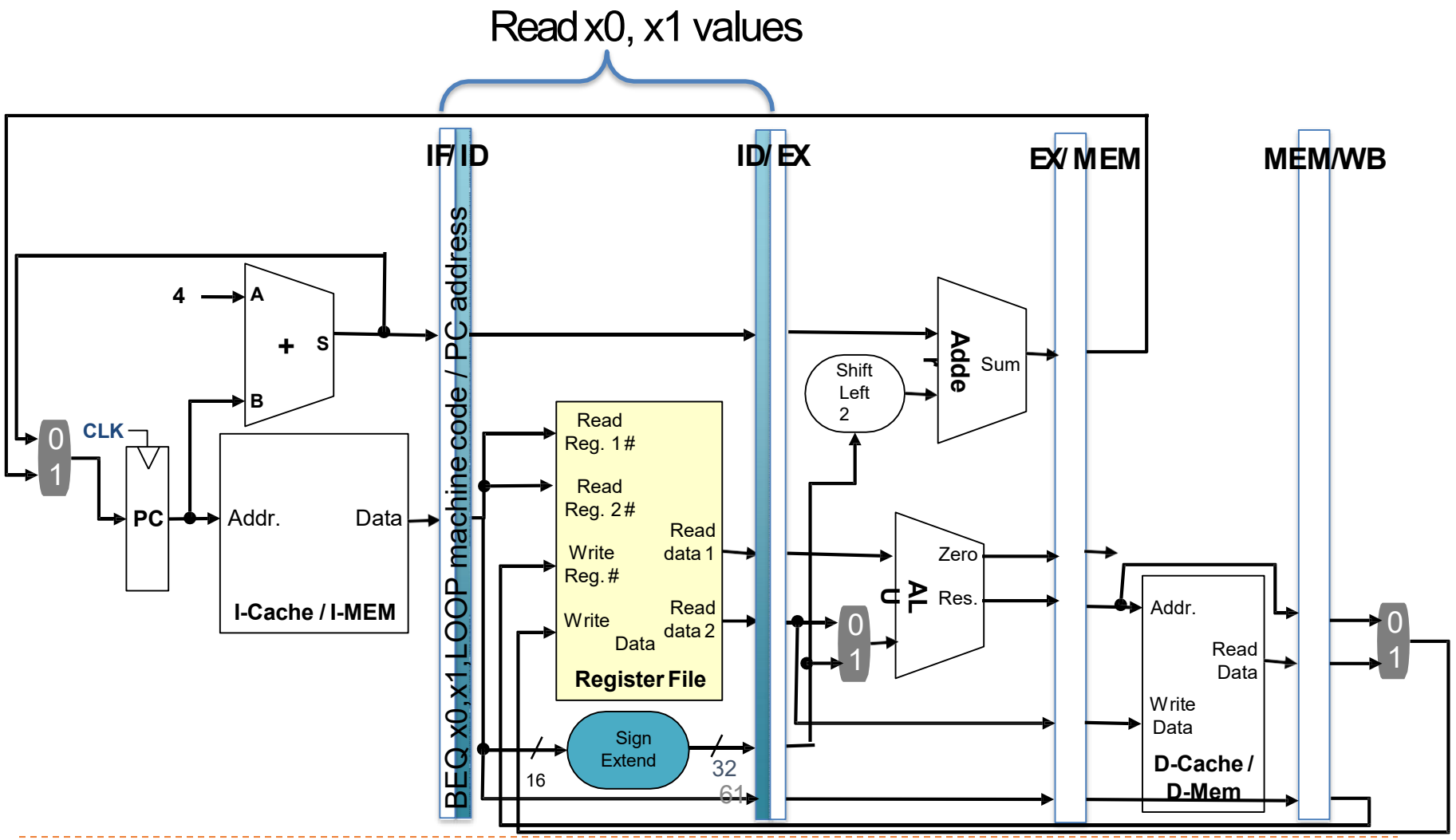

Store the sum to x4
IF/ID
ID/EX
EX/MEM
MEM/WB
4
A
+
S
B
CLK
0
1
PC
Addr.
Data
I-Cache / I-MEM
Read Reg. 1#
Read Reg. 2#
Write Reg. #
Write Data
Read data 1
Read data 2
Register File
Sign Extend
16
32
Shift Left 2
Adde
Sum
Zero
ALU
Res.
Addr.
Read Data
Write Data
D-Cache / D-Mem
x4 # / Sum of x5 + x6

BEQ x0,x1,LOOP

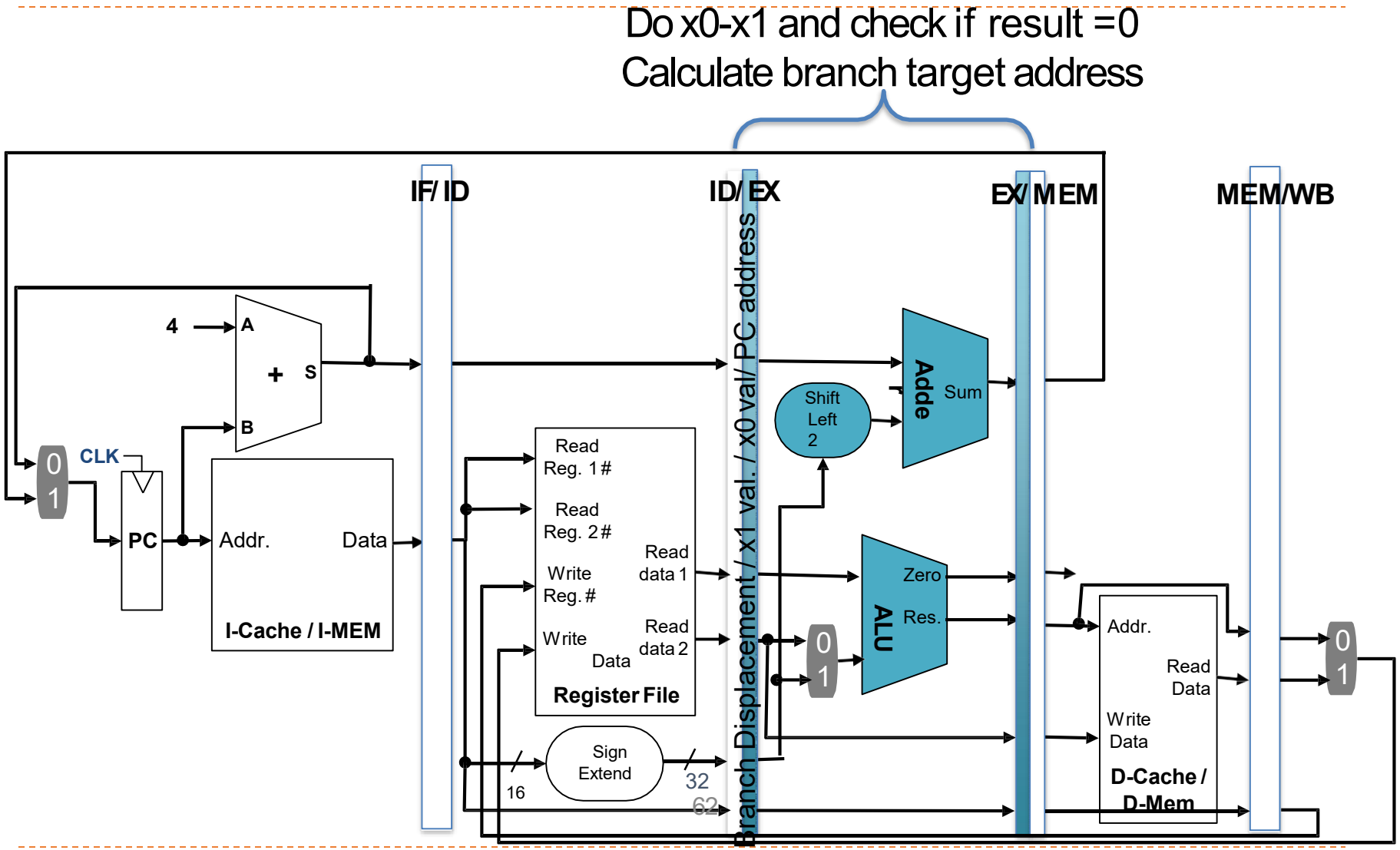
Fetch inst and increment PC



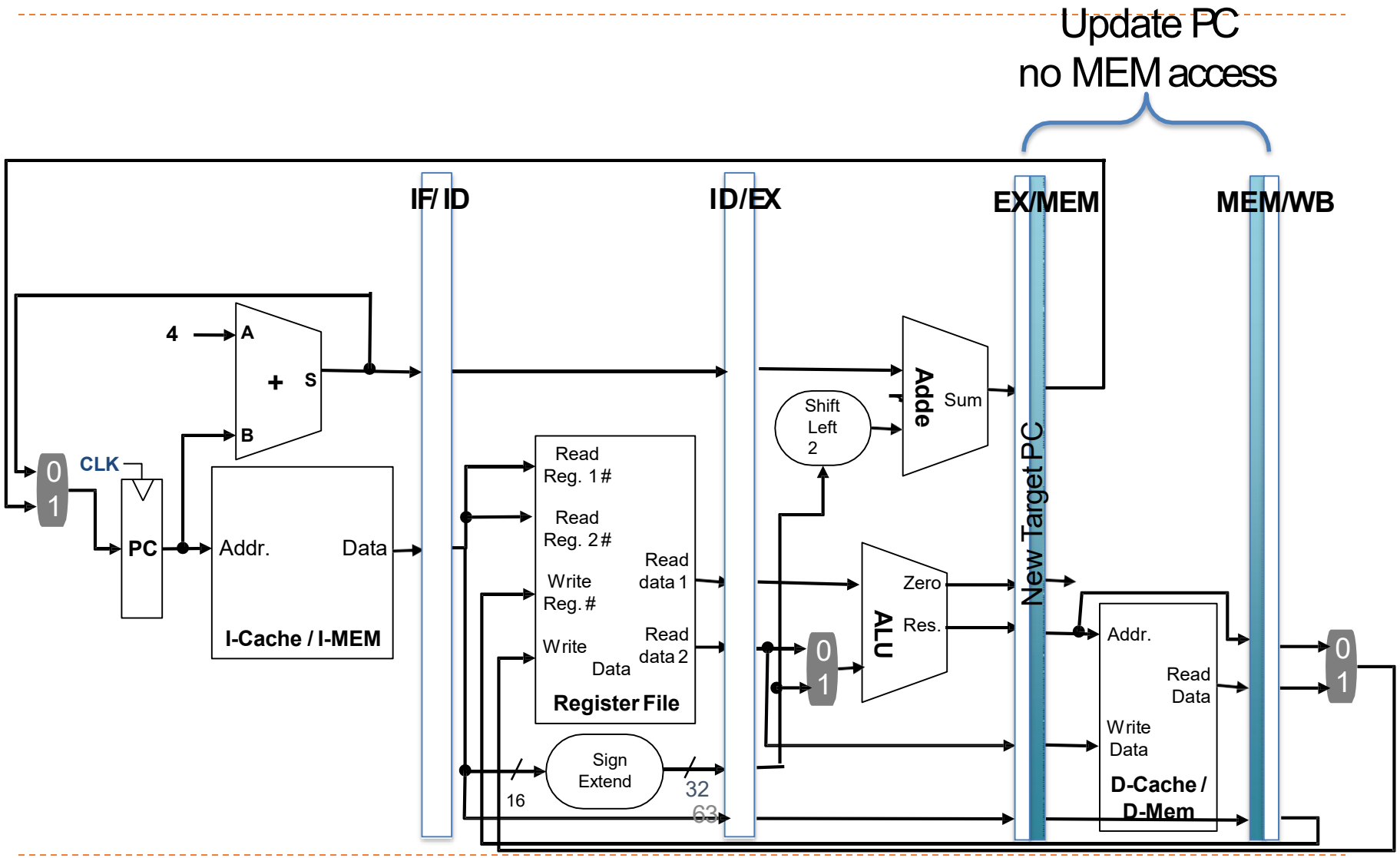
BEQ x0,x1,LOOP



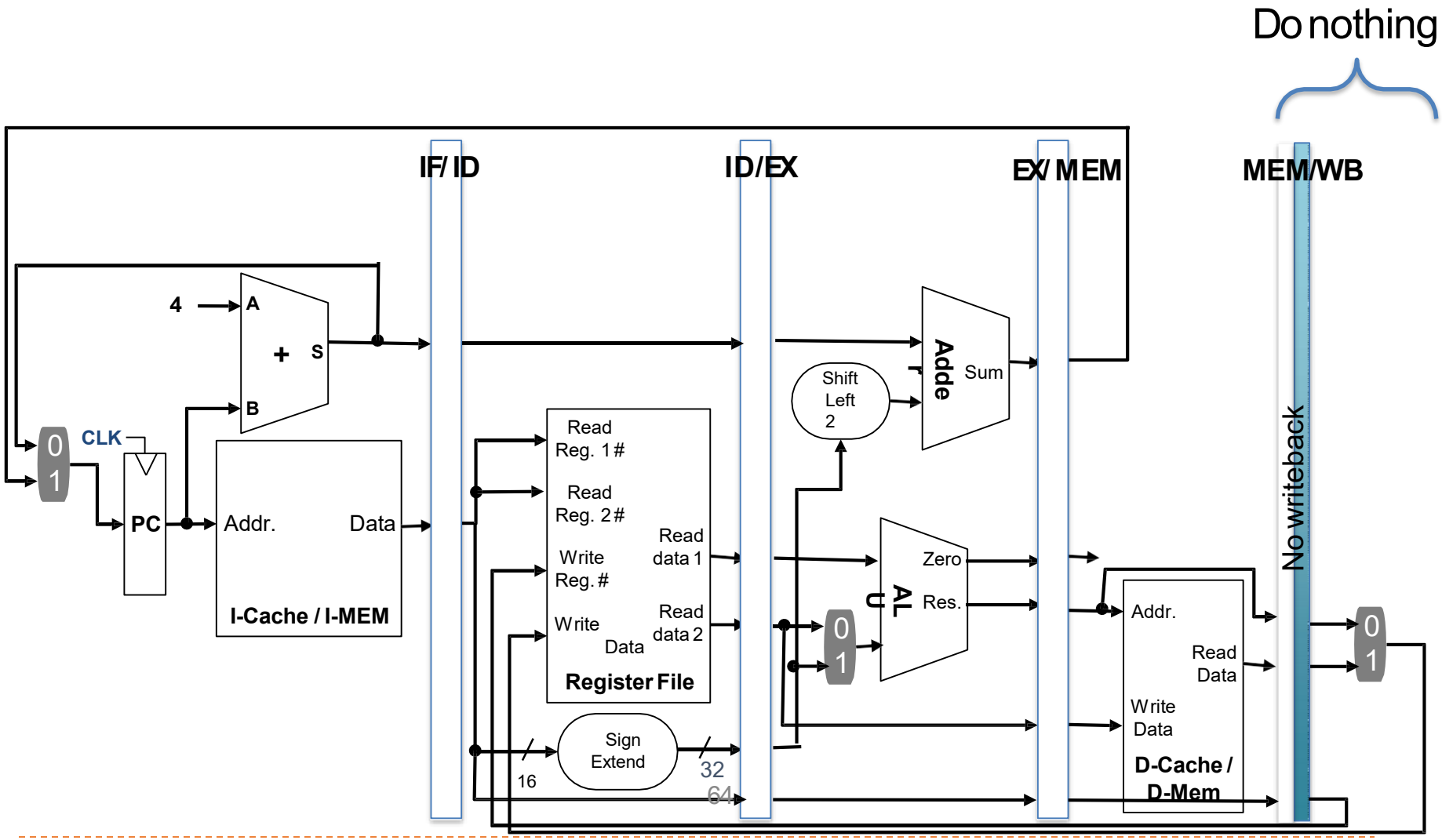
BEQ x0,x1,LOOP



BEQ x0,x1,LOOP

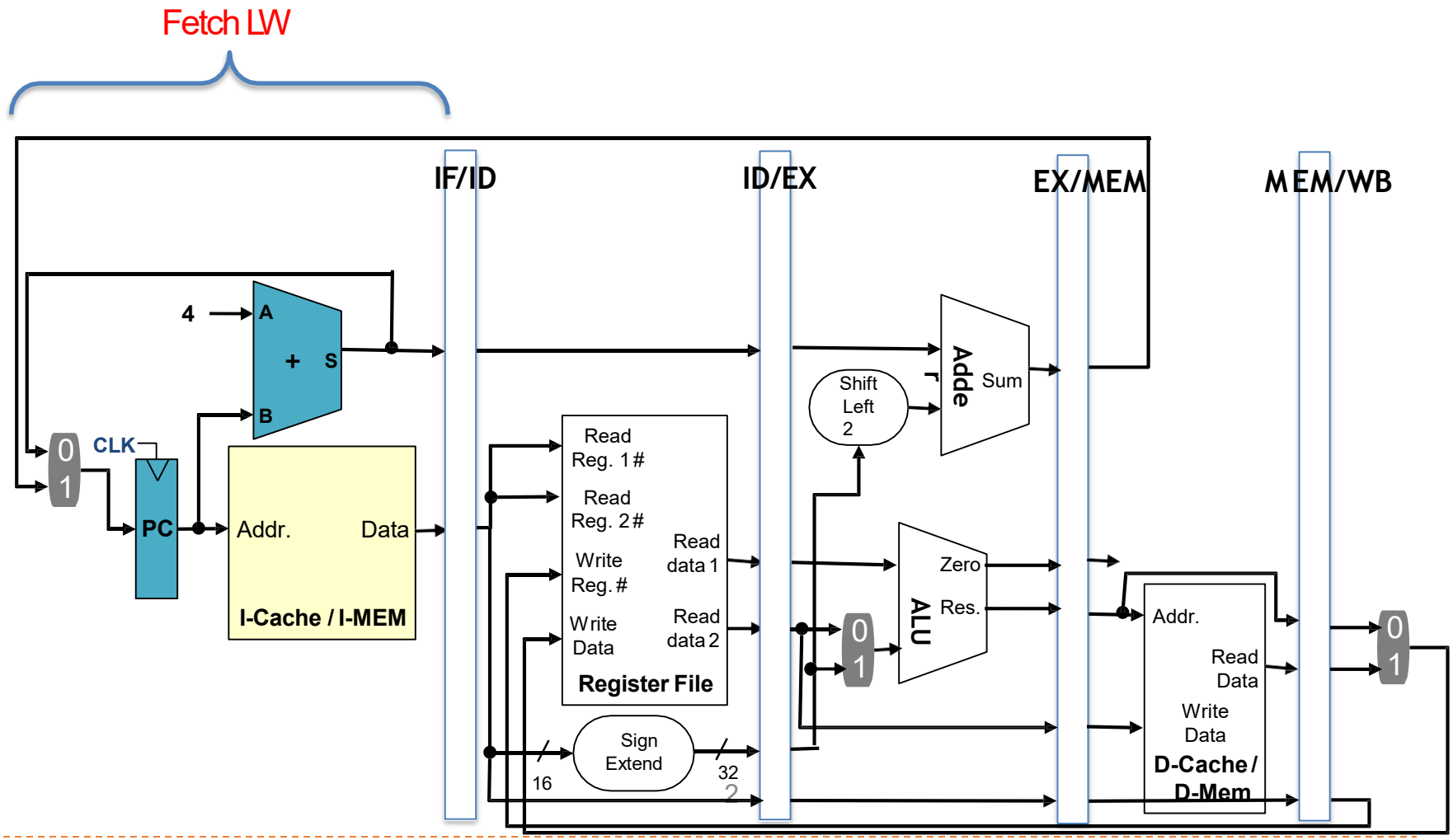


BEQ x0,x1,LOOP

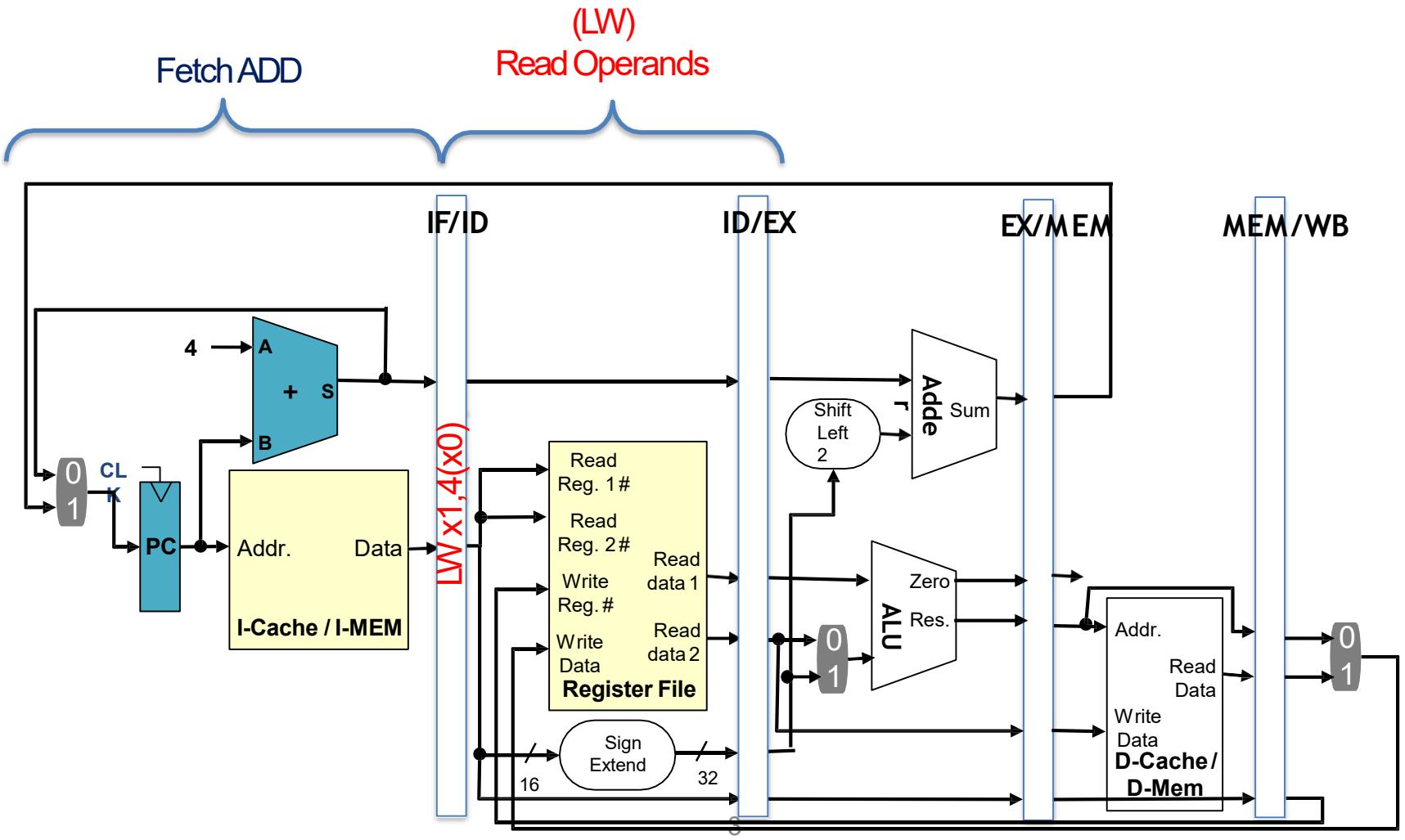


Run Them Together

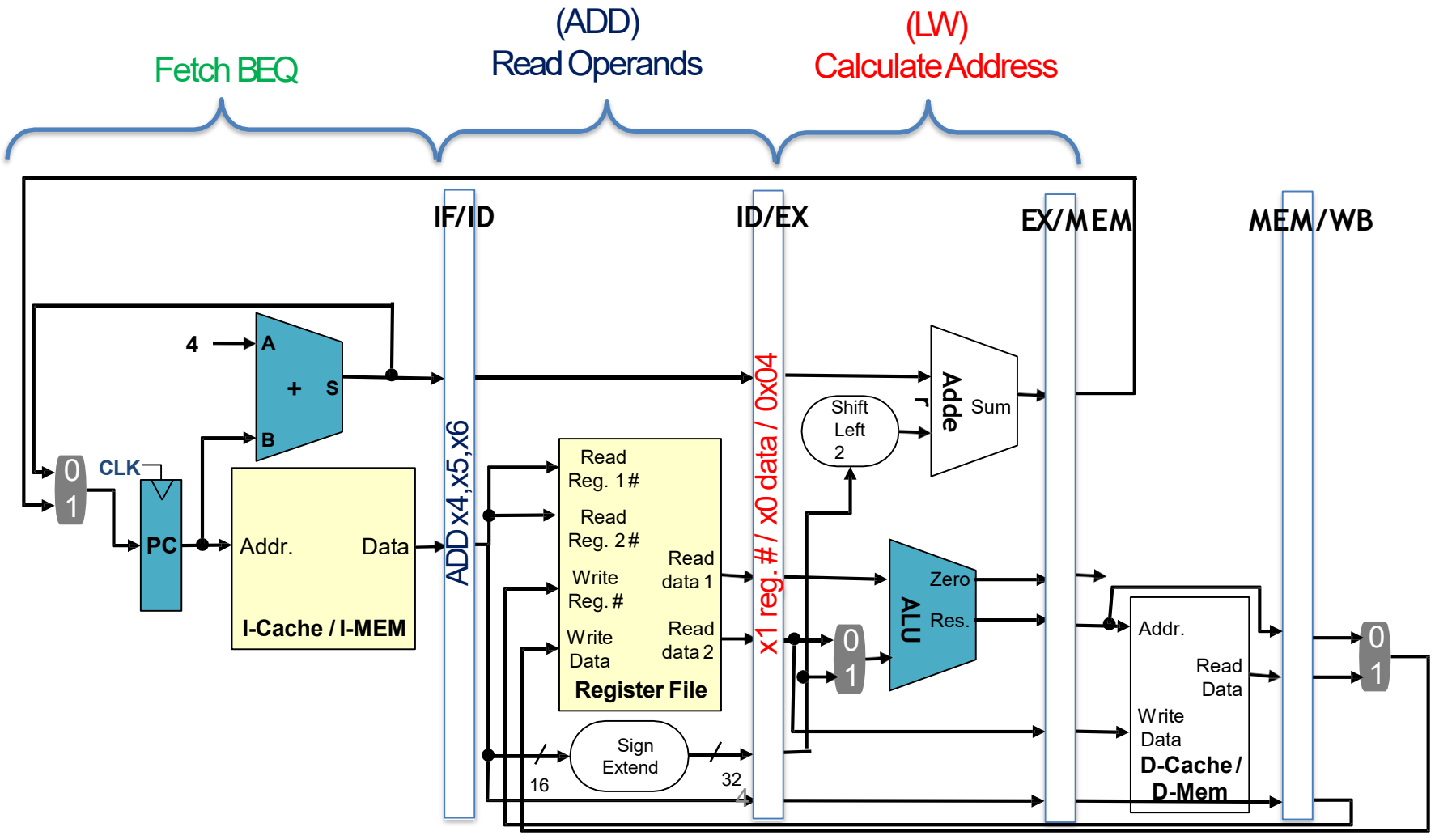
Executing Instructions altogether in 5-Stage Pipeline – Cycle 1



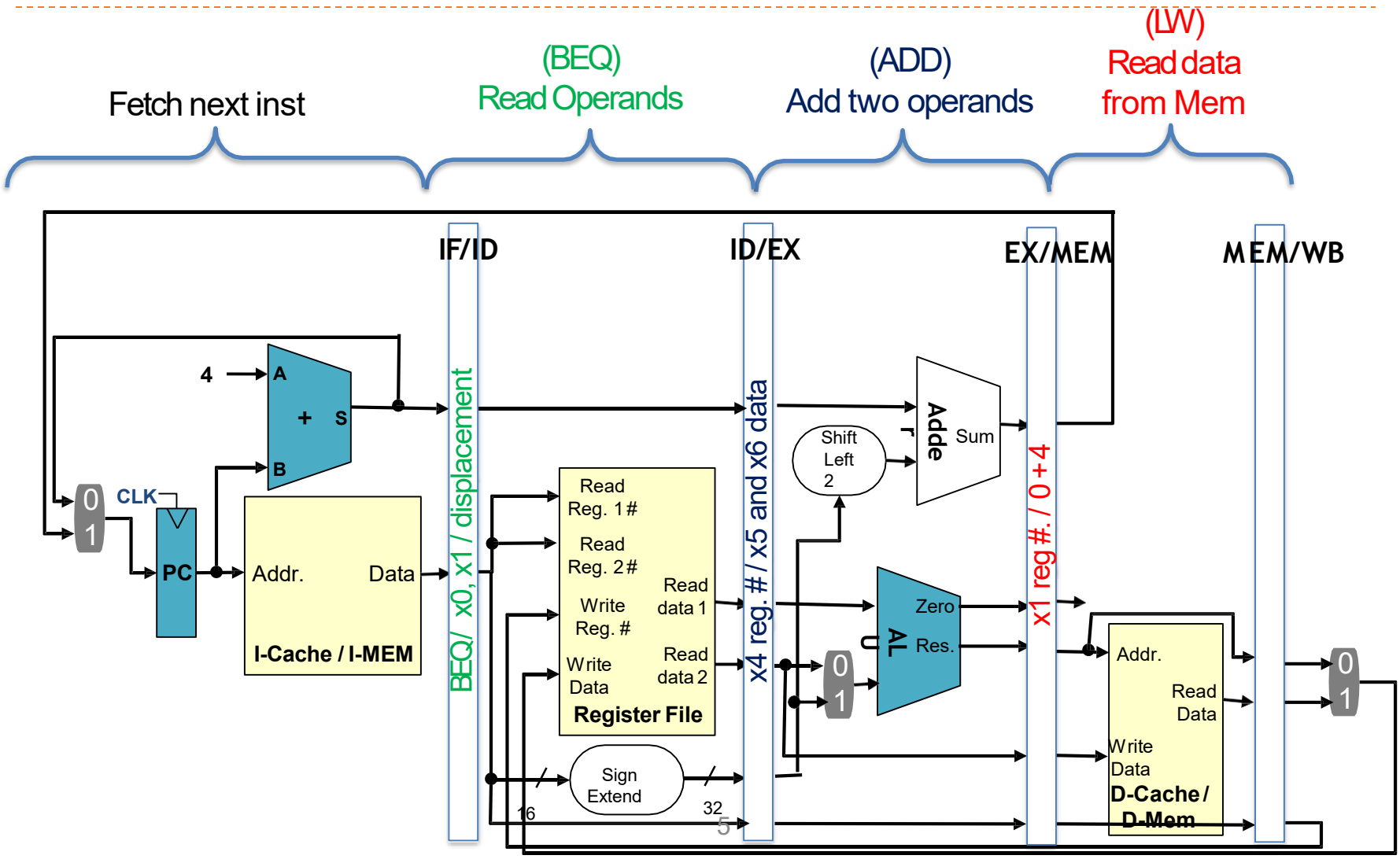
Executing Instructions altogether in 5-Stage Pipeline – Cycle 2



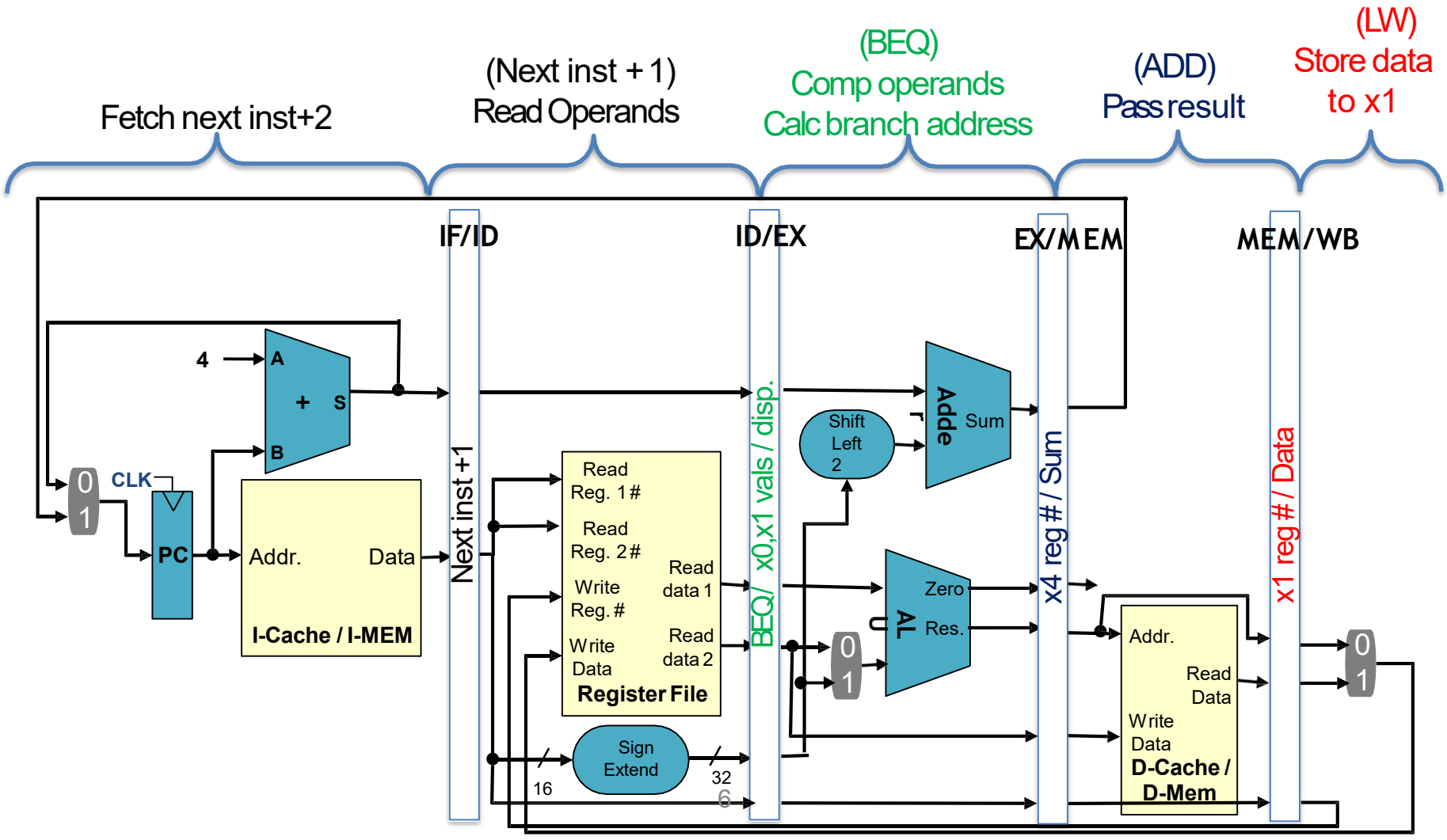
Executing Instructions altogether in 5-Stage Pipeline – Cycle 3



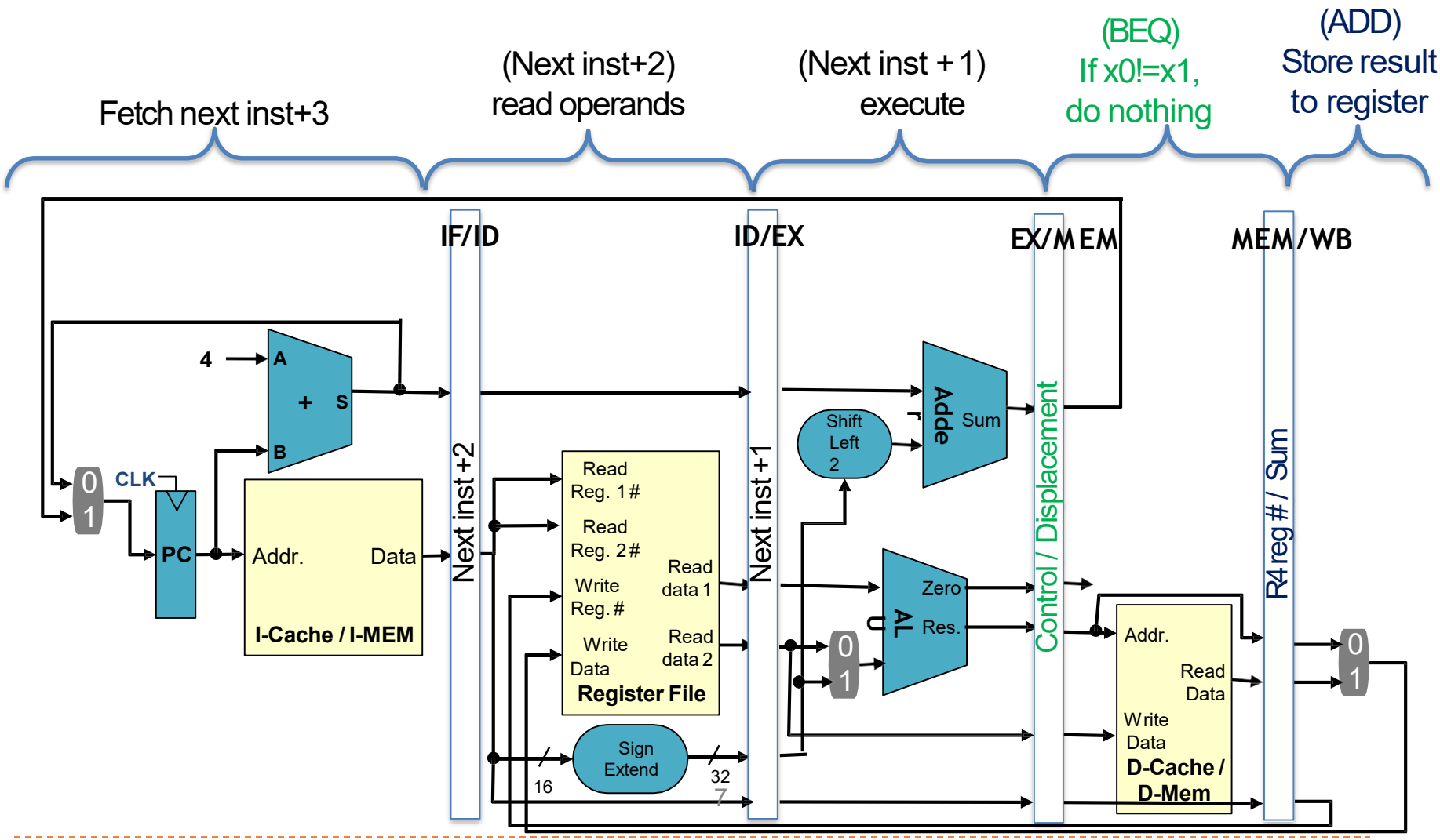
Executing Instructions altogether in 5-Stage Pipeline – Cycle 4



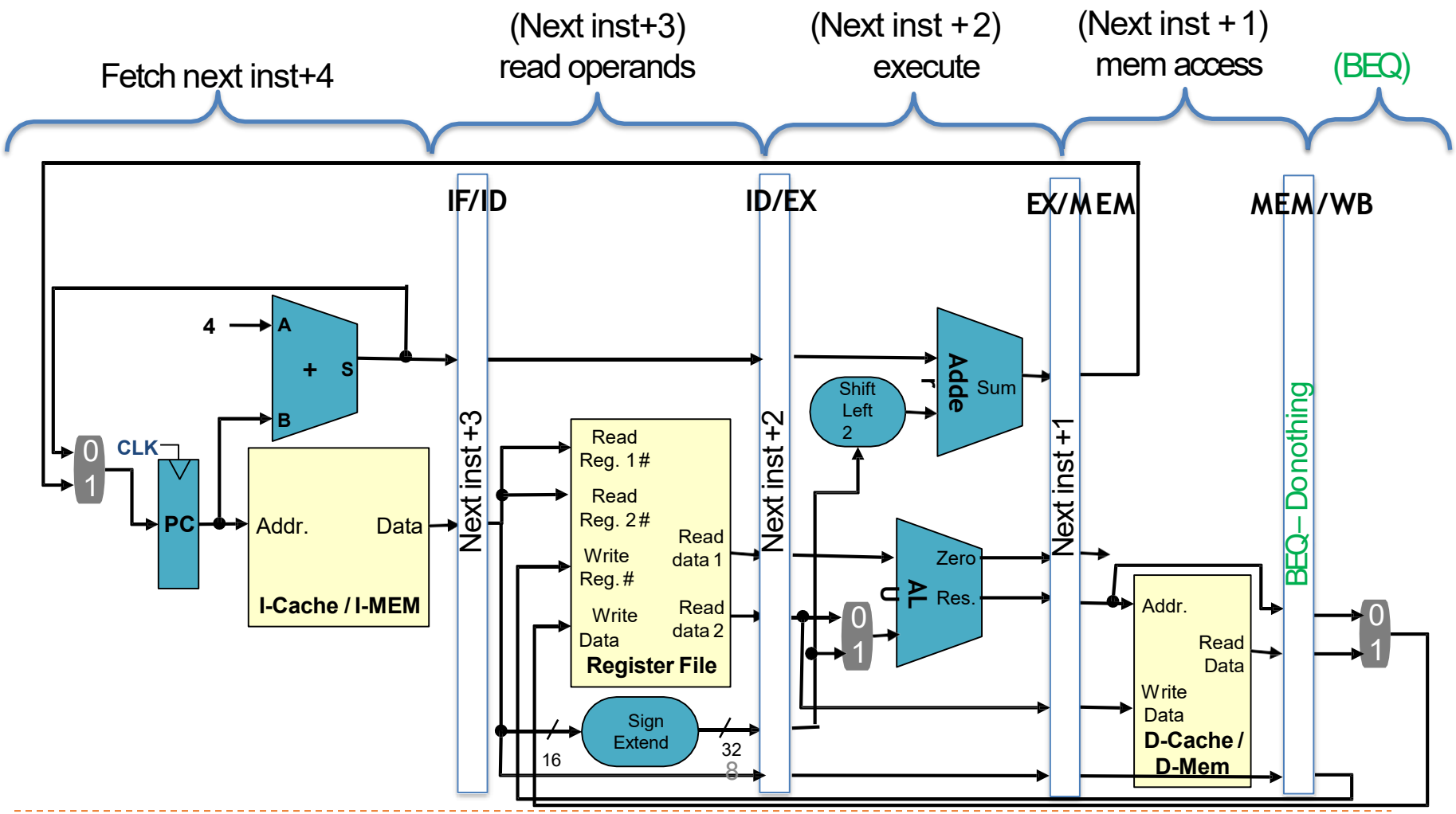
Executing Instructions altogether in 5-Stage Pipeline – Cycle 5



Executing Instructions altogether in 5-Stage Pipeline – Cycle 6



Executing Instructions altogether in 5-Stage Pipeline – Cycle 7



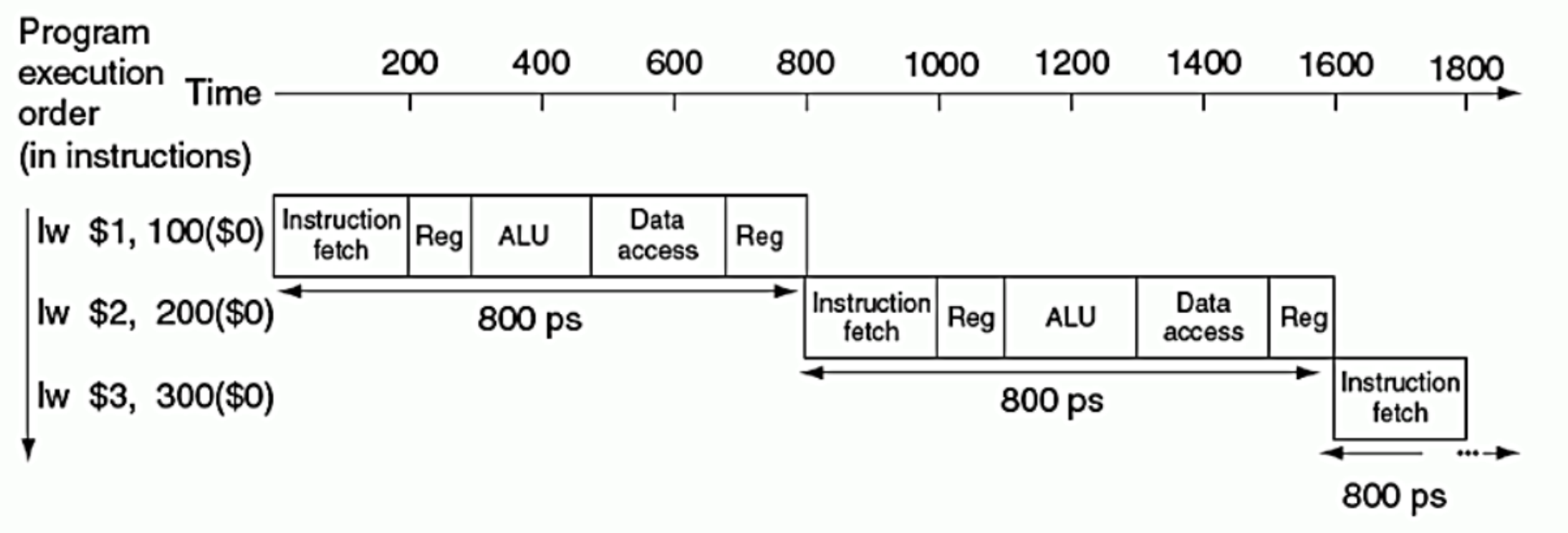
Pipeline Speedup for 5-Stage RISC-V CPU

- Assume operation time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- *Q: What are clock periods for single-cycle datapath and 5-stage pipeline datapath?*
  - Single cycle datapath: longest instruction execution (800ps)
  - Pipeline datapath: longest single stage execution (200ps)

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
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

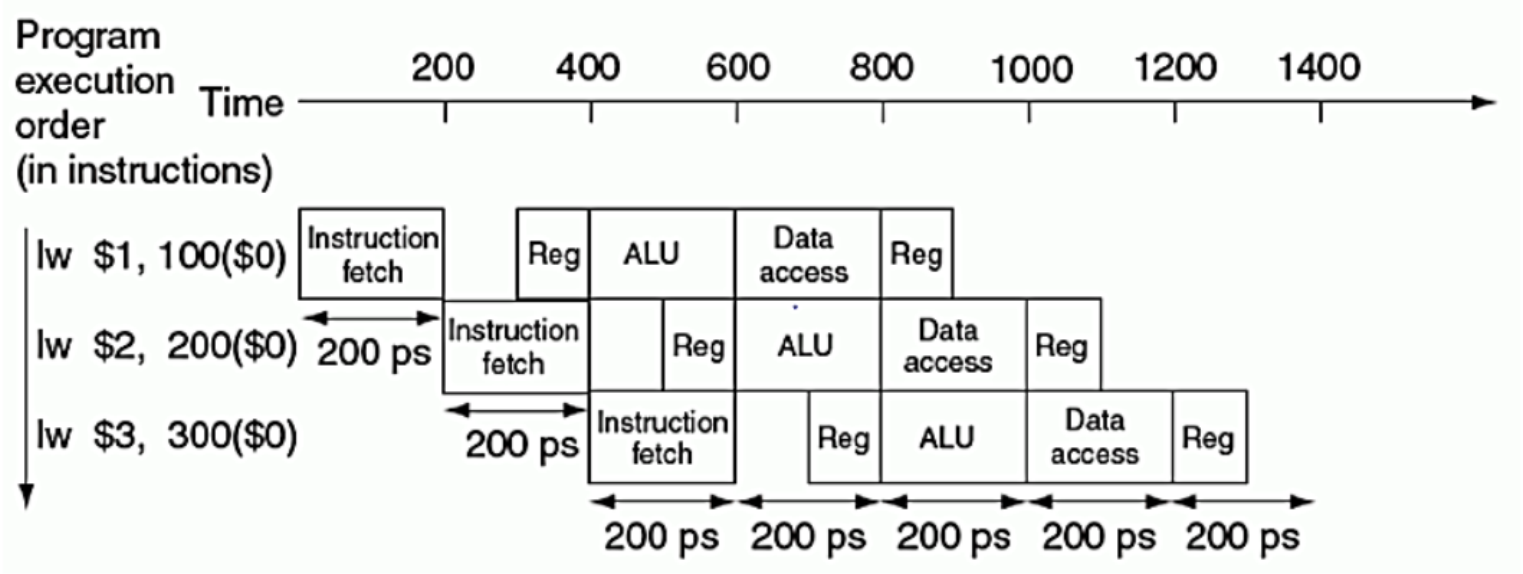
Single-cycle, non pipelined execution

- In the single-cycle implementation, lw takes 1 cycle totaling 800 ps. We cannot start the next instruction until the last cycle ends so the time between the first and fourth instruction is *2400 ps*



Pipeline Execution

- The pipeline stage times of a computer are limited by the slowest resource.



PIPELINING SPEEDUP

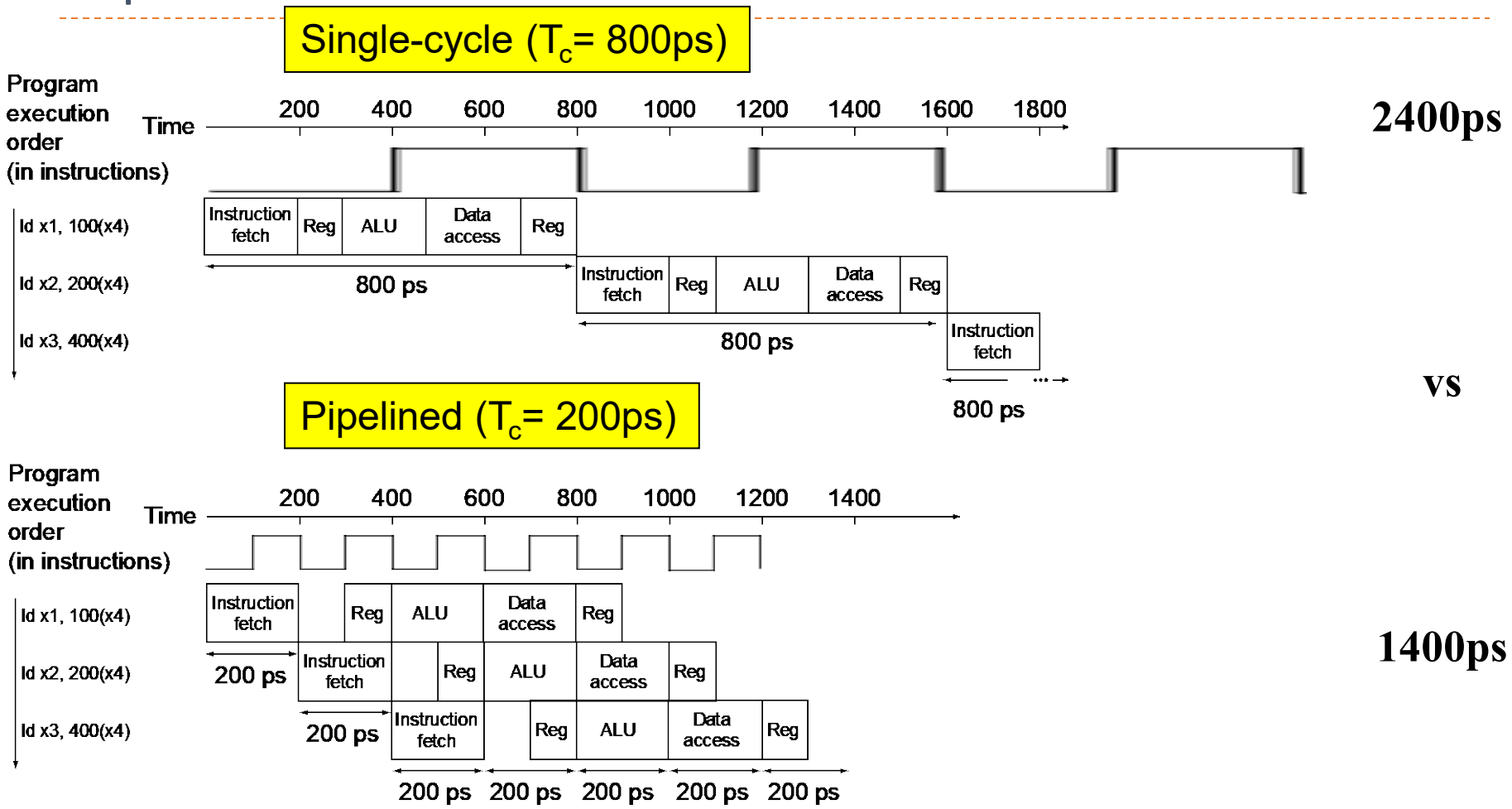
As you may have already noticed, our `lw` example does *not* exhibit 5-fold speedup even though there are 5 stages. We have an overall completion time of 2400 ps for single-cycle and an overall completion time of 1400 ps for pipelining. This is merely a 1.7 times speedup.

Imagine instead that we are executing 1,000,000 lw instructions. For single-cycle, this means 800,000,000 ps since each instruction requires 800 ps. But for pipelining, this only means 200,000,800 ps since each additional instruction only adds 200 ps.

$$\frac{8{,}000{,}000\ ps}{2{,}000{,}800\ ps} \cong \frac{8}{2} = 4$$

When we increase the number of instructions, we get roughly 4 times speedup.

Pipeline Performance



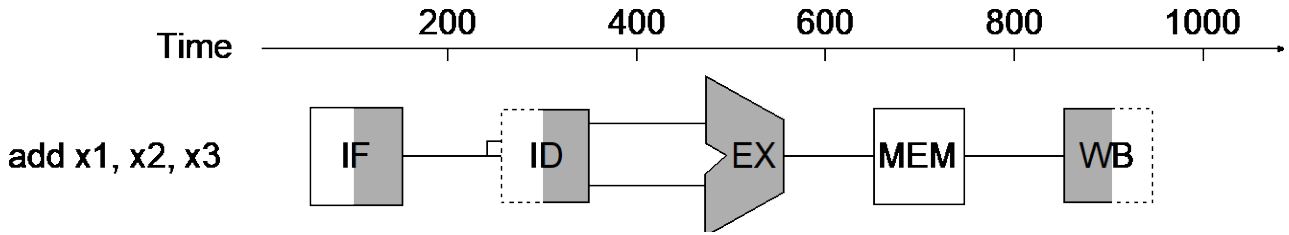
- For large number of instructions, say 1M, the speedup will be
  - ~= 800ps/200ps = 4

Summary so far

- The pipelined datapath uses multiple memories and ALUs.
  - Instruction execution is split into several stages.
- Pipeline registers propagate data and control values to later stages.
- The RISC-V instruction set architecture supports pipelining with uniform instruction formats and simple addressing modes.

Hazards

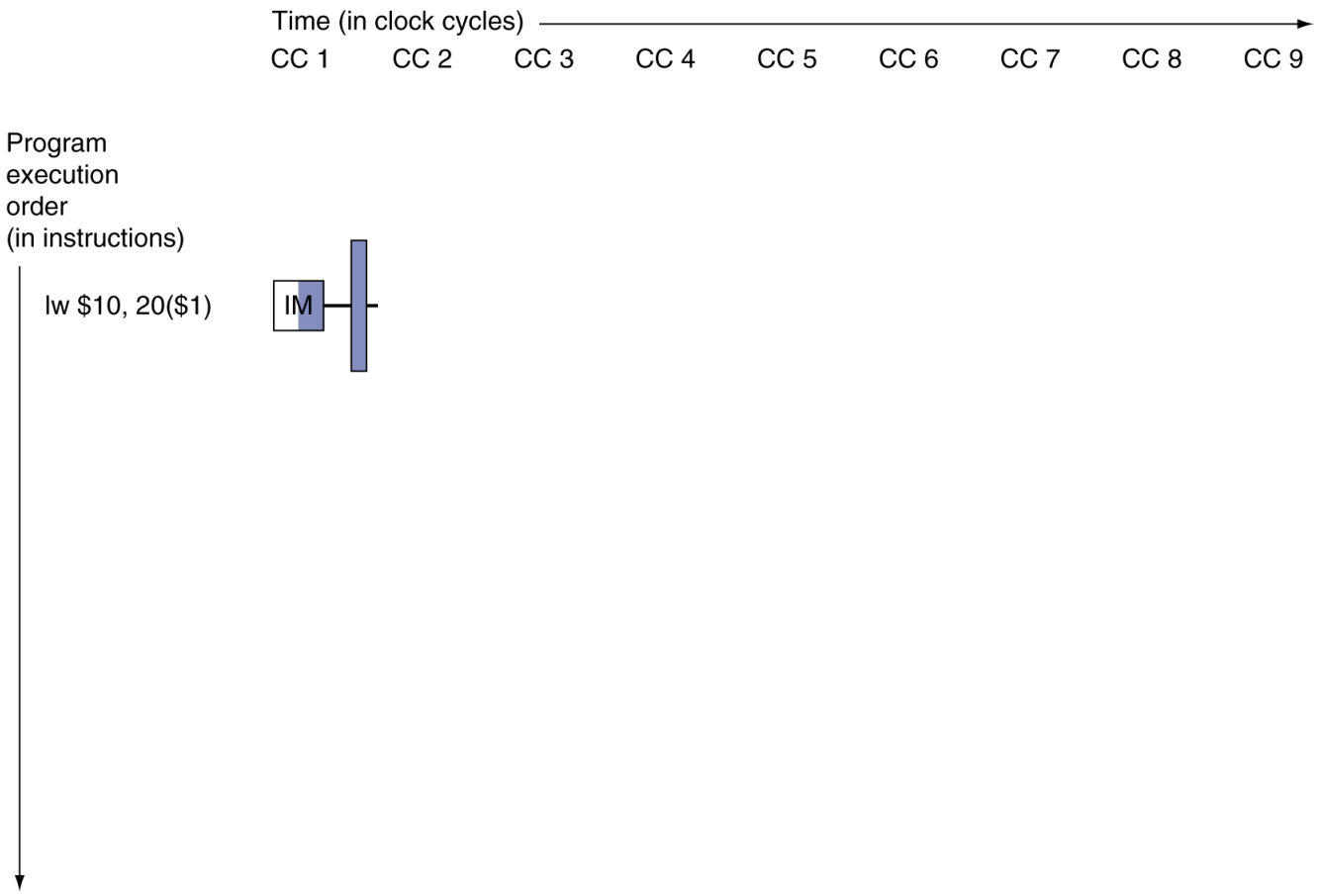
Graphic Representation of Pipeline



- IF: Instruction Fetch from memory
  - Box representing instruction memory
  - Right-half shade representing usage of IF at the second half of the cycle
- ID: Instruction Decode & register read
  - Box representing register
  - Right-half shade representing usage (read) of Register at the second half of the cycle
- EX: Execute operation or calculate address
  - Shade representing usage
- MEM: Access memory operand (only for load/store)
  - White background representing NOT used by add instruction in this example
- WB: Write result Back to register (only for load and AL instructions)
  - Box representing register
  - Left-half shade representing write to register at the first half of the cycle

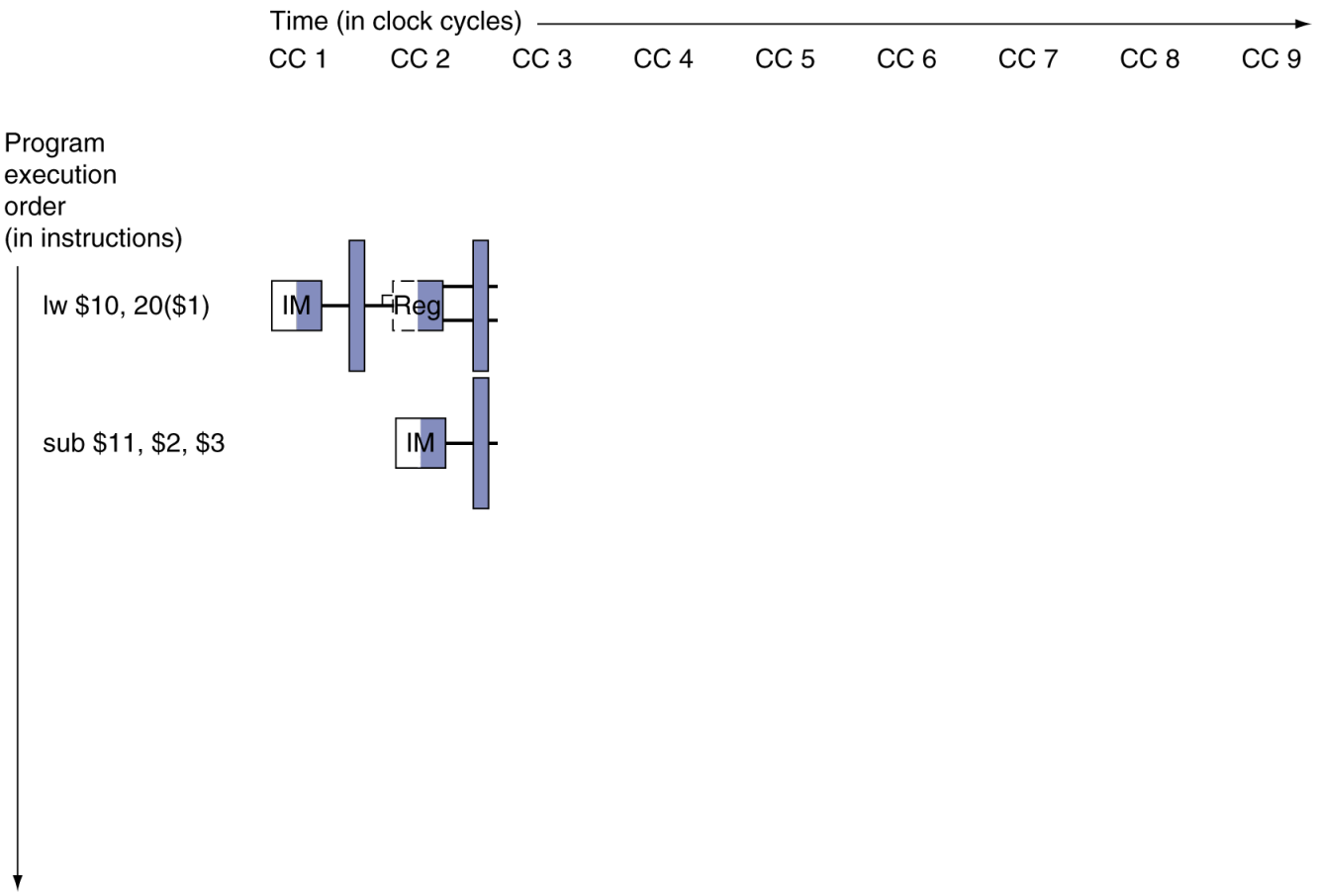
Multi-Cycle Pipeline Diagram

- Form showing resource usage



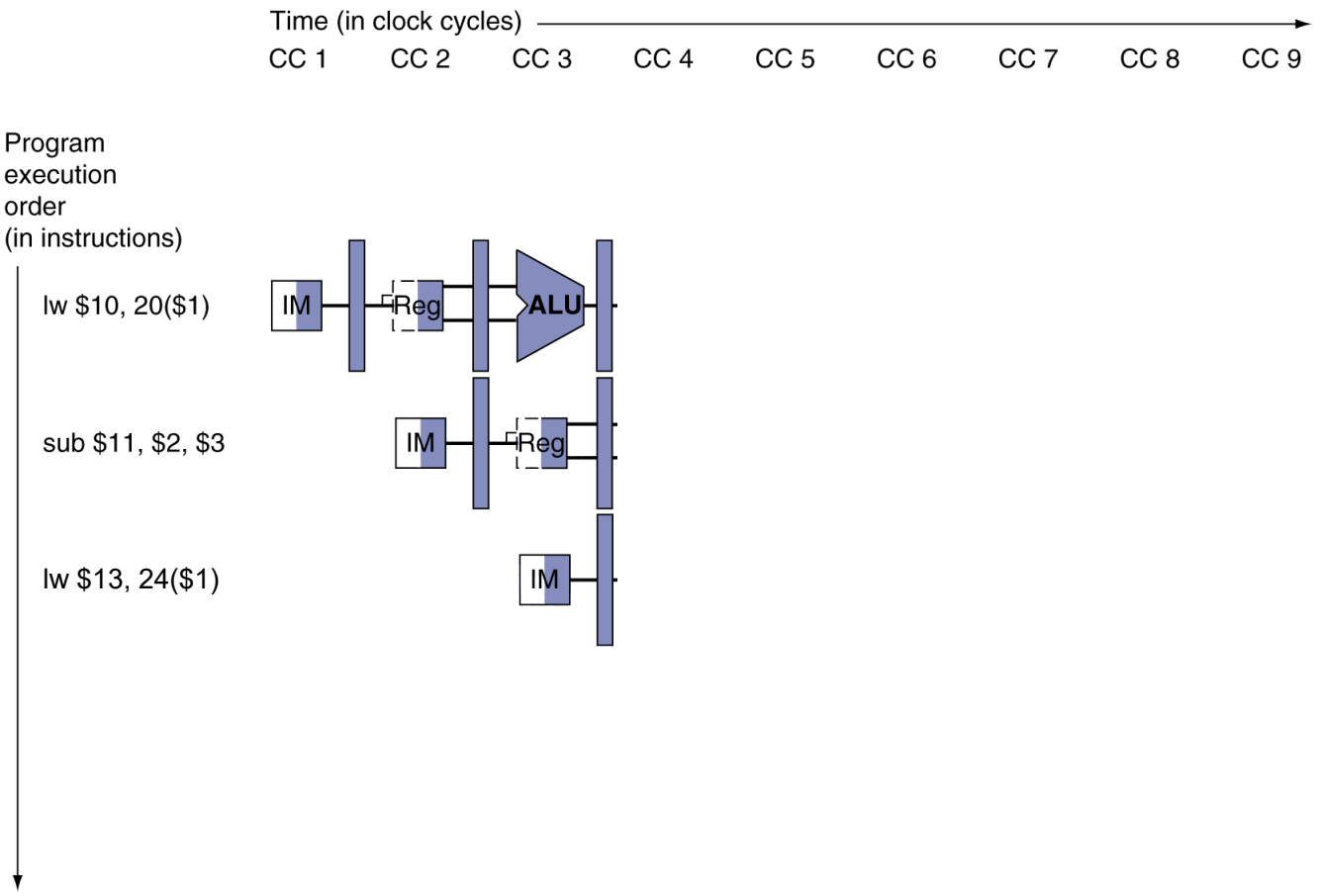
Multi-Cycle Pipeline Diagram

- Form showing resource usage



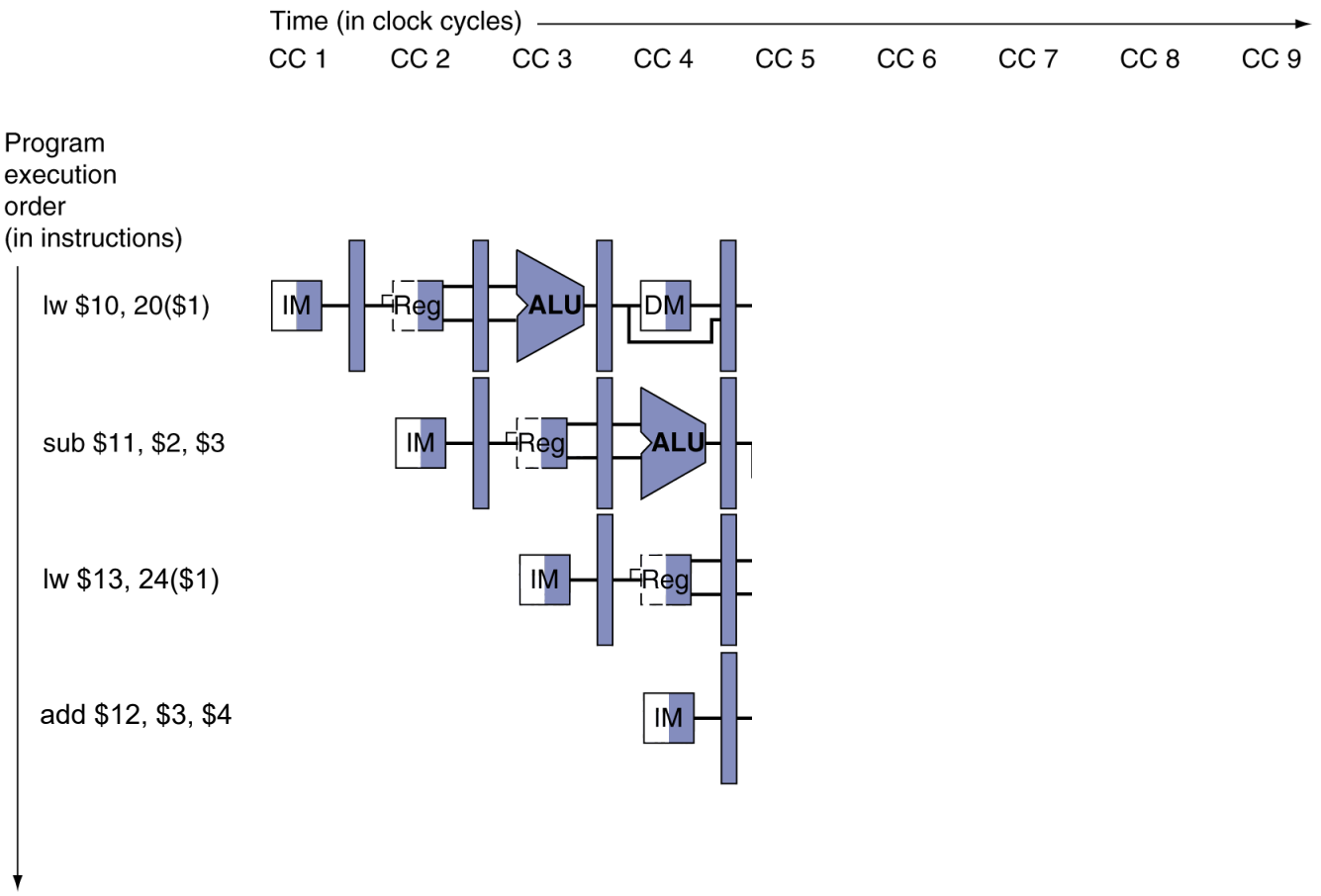
Multi-Cycle Pipeline Diagram

- Form showing resource usage



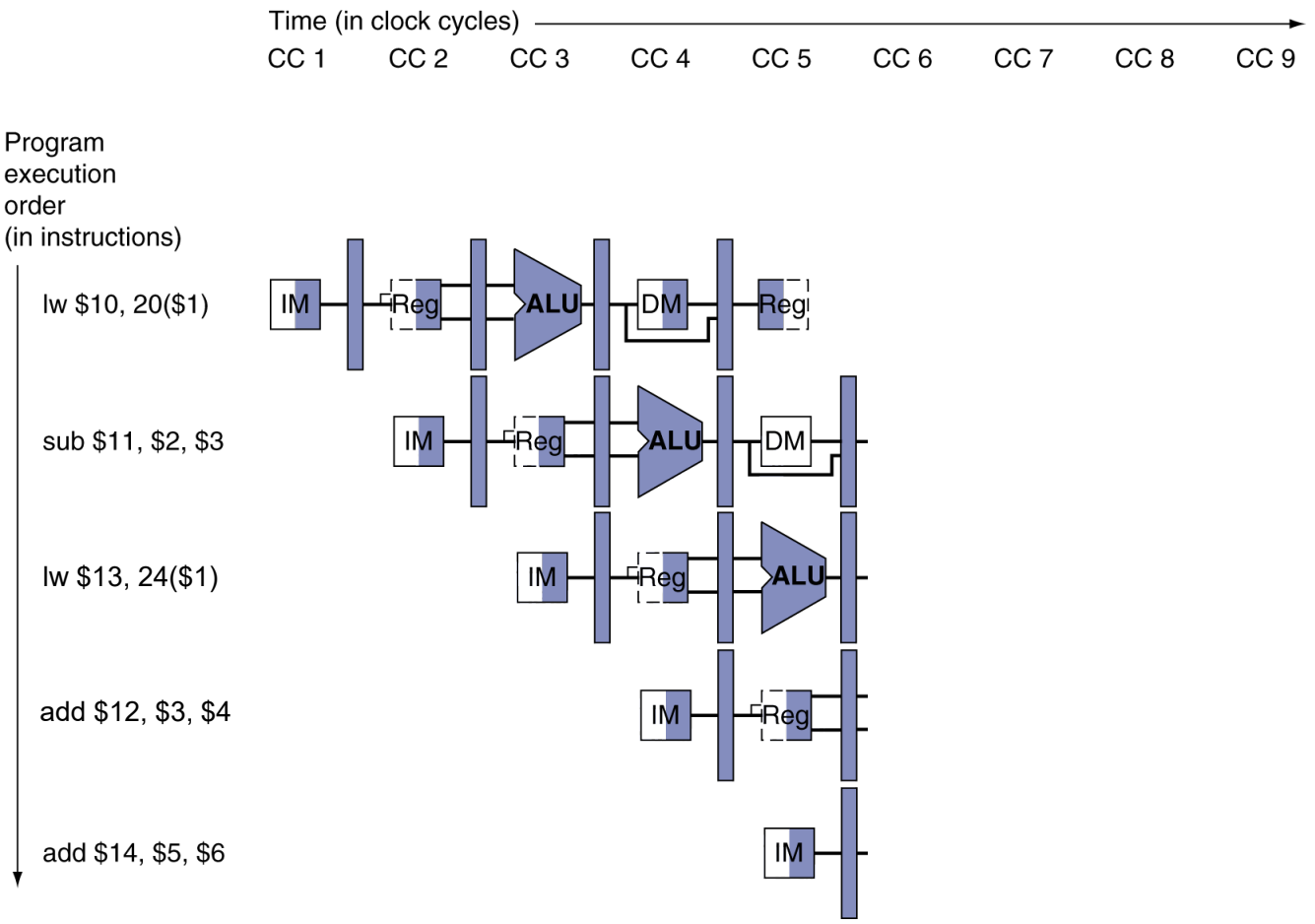
Multi-Cycle Pipeline Diagram

- Form showing resource usage



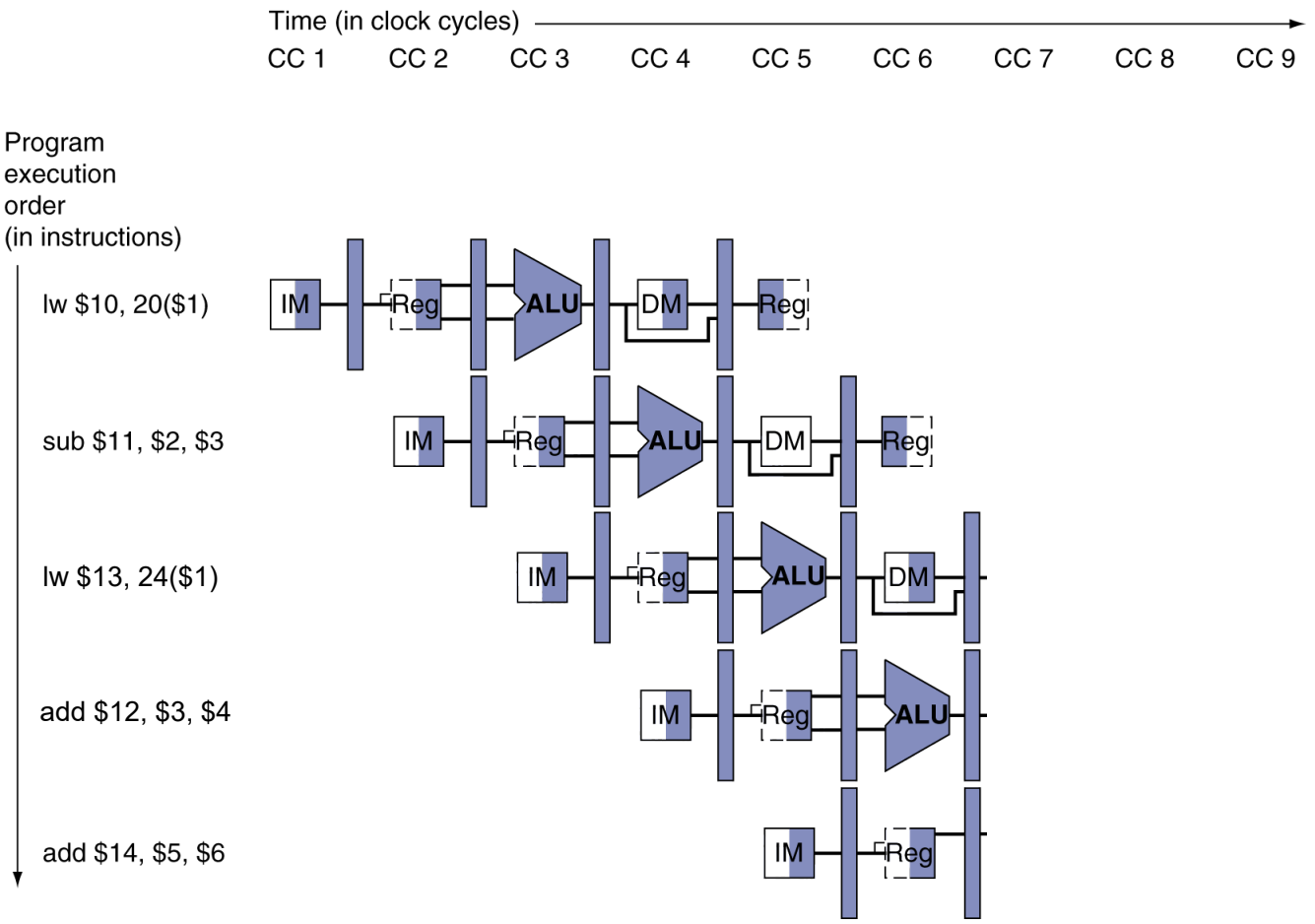
Multi-Cycle Pipeline Diagram

- Form showing resource usage



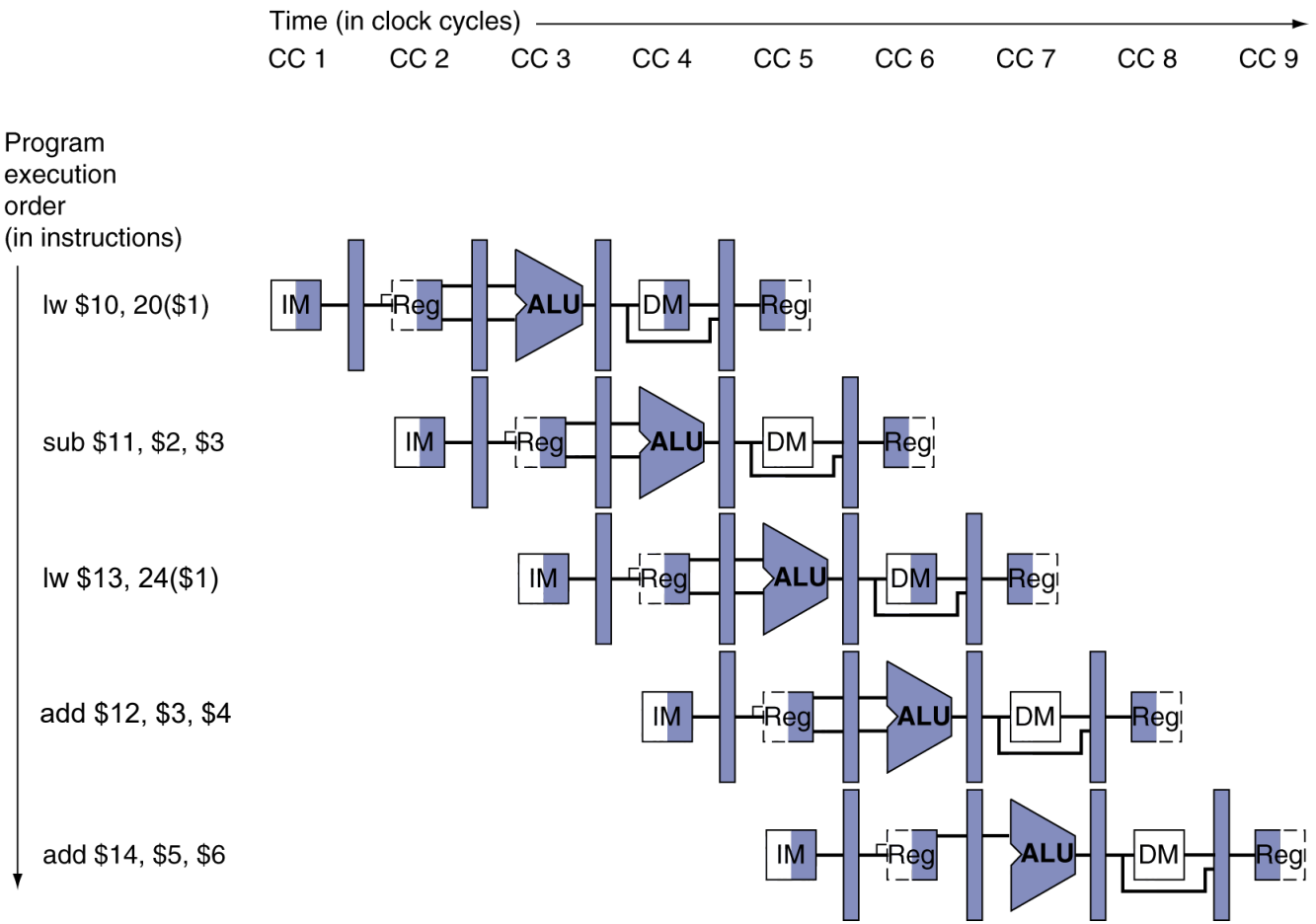
Multi-Cycle Pipeline Diagram

- Form showing resource usage



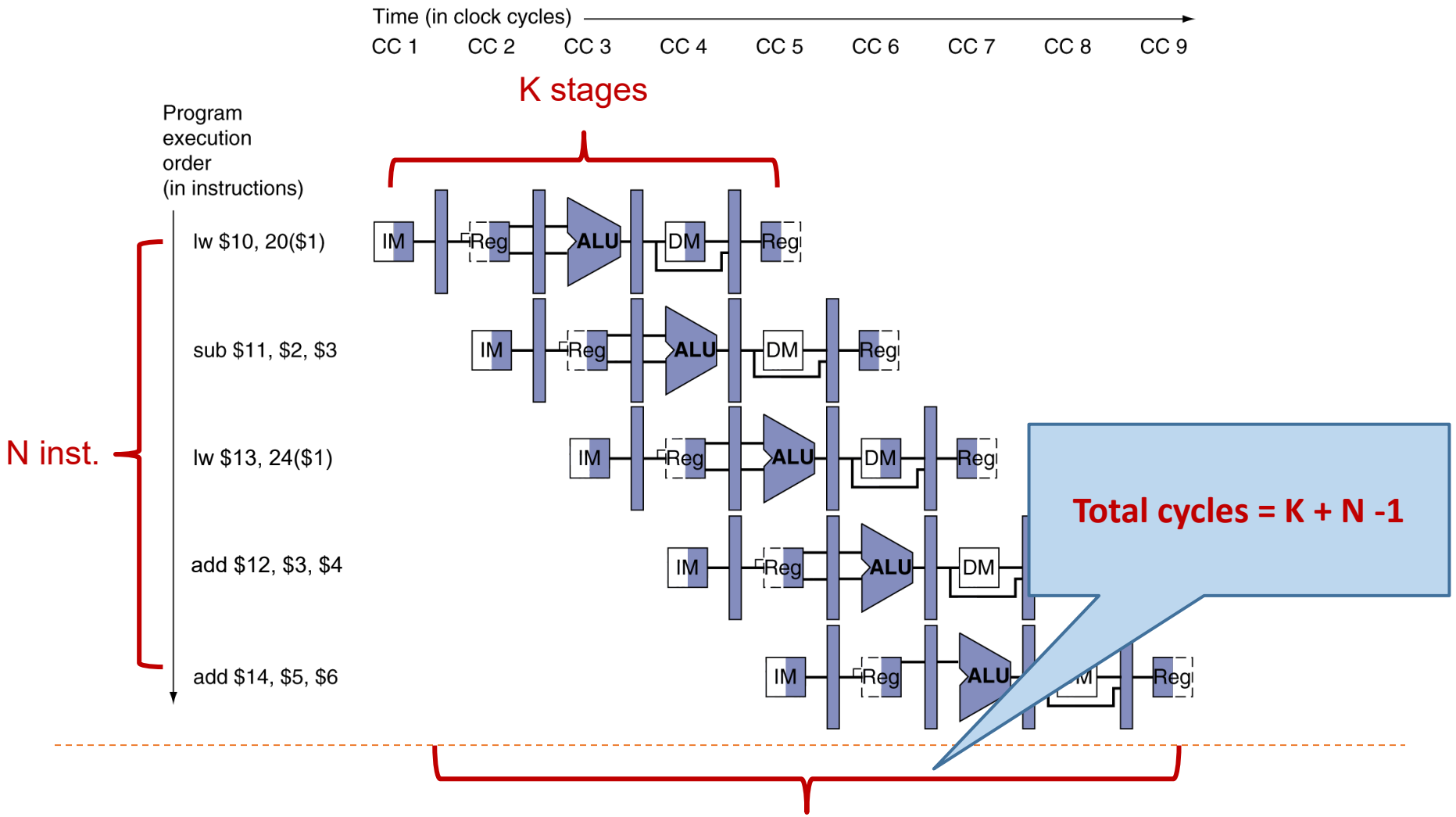
Multi-Cycle Pipeline Diagram

- Form showing resource usage



Pipeline Timing

- Form showing resource usage



CPI

$$CPI = \frac{Total\_cycles}{Total\_instructions} = \frac{K + N - 1}{N} \approx 1 \ (N \rightarrow \infty)$$

Recall that CPI is also 1 for single sycle datapath.
Are we getting any faster after all these troubles?

We assume that pipeline is full all the time
.... Which is true unless there are "hazards"

Pipelining Hazards

- **Dependencies**: relationships between instructions that prevent one instruction from being moved past another.

- **Hazards**: situation where next instruction cannot execute in the following cycle.
  - **Structure hazards**
    - A required resource is busy
  - **Data hazard**
    - Need to wait for previous instruction to complete its data read/write
  - **Control hazard because of branch or jump**
    - Deciding on control action depends on previous instruction

- **Stalls**: technique of stalling an instruction until a pipeline hazard no longer exists.

Structure Hazards

- occurs when a planned instruction cannot execute in the proper clock cycle because the hardware cannot support the combination of instructions that are set to execute in the given clock cycle.
  - In laundry analogy, single person to fold and put clothes away or if we used a combo washer/dryer instead of separate washer and dryer machines.

Structure Hazard 1– Single Memory

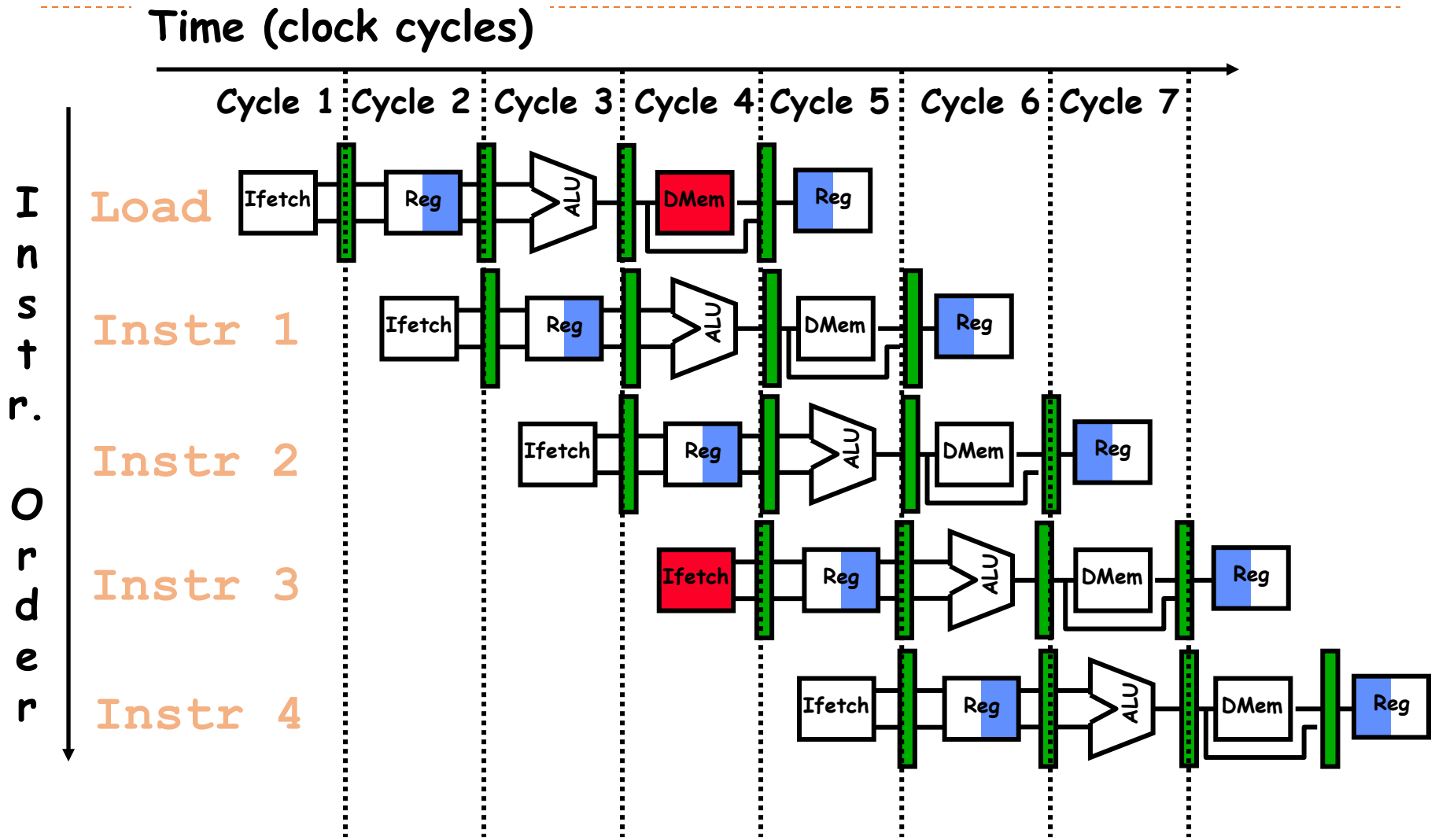
- Imagine the following instructions are executed over 8 clock cycles.
- Notice in cycle 4, we have a MEM and IF phase executing. If there is only one single memory unit, we will have a structural hazard.

cycle	1	2	3	4	5	6	7	8
Inst 1	IF	ID	EX	MEM	WB			
Inst 2		IF	ID	EX	MEM	WB		
Inst 3			IF	ID	EX	MEM	WB	
Inst 4				IF	ID	EX	MEM	WB

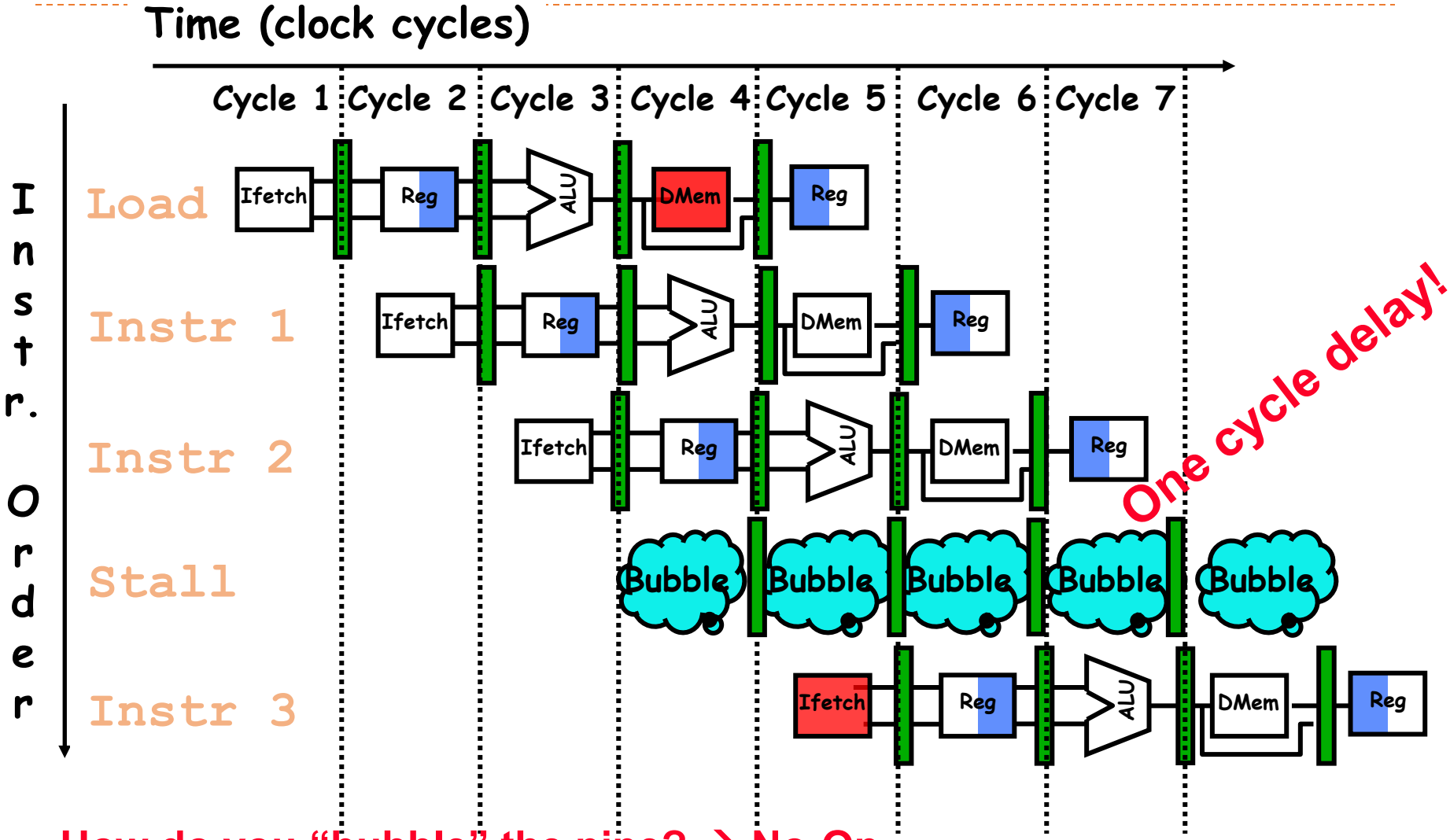
Solution for Structure Hazards 1

- In RISC-V pipeline with a single memory model
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline “bubble”

One Memory Port→Structural Hazards



One Memory Port/Structural Hazards



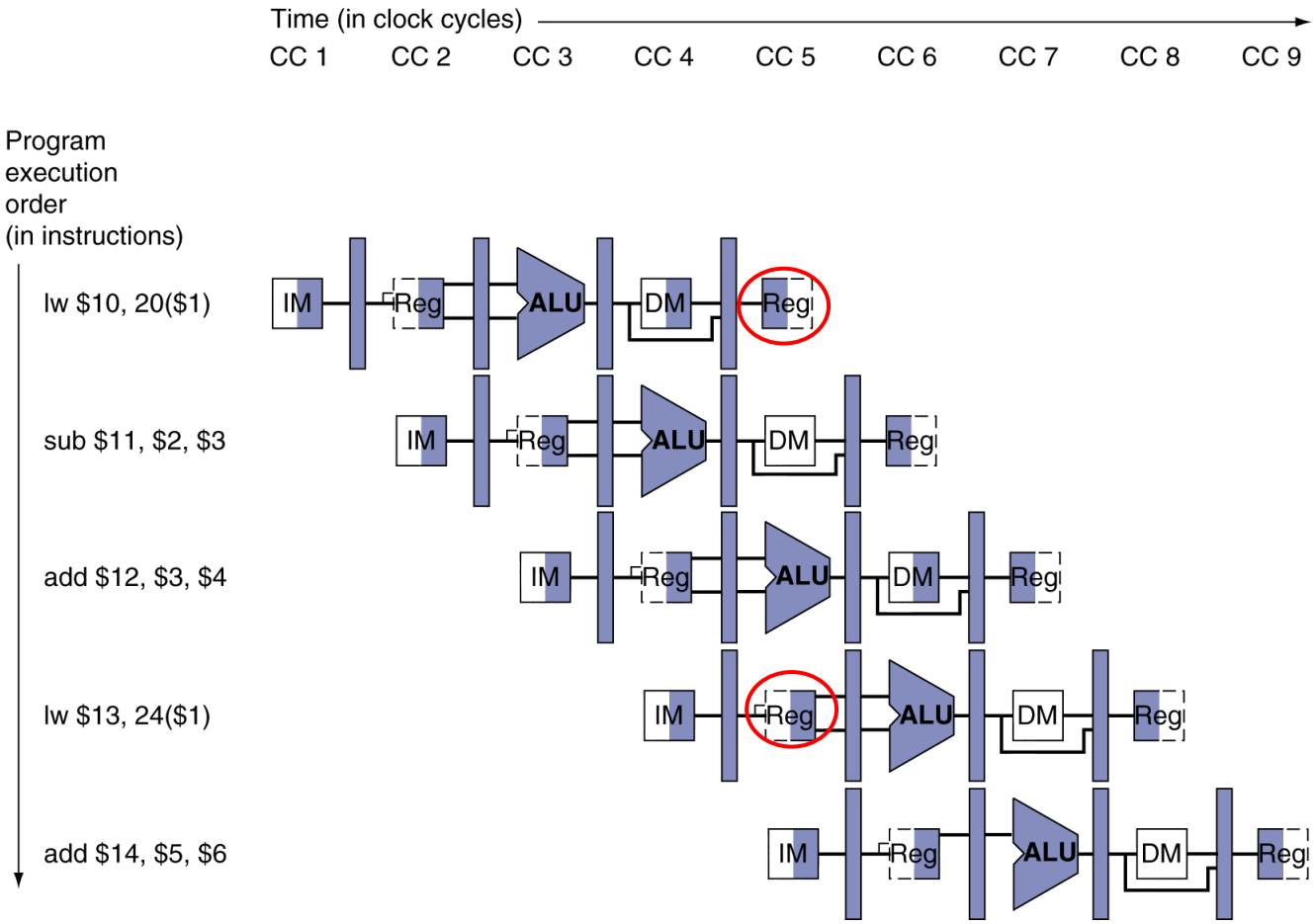
How do you "bubble" the pipe? → No-Op

Summary of Structure Hazard 1

- To address structure hazard, have separate memories for instructions and data
- However, it will increase cost
  - E.g.: pipelining function units or duplicated resources is a high cost;

† If the structure hazard is rare, it may not be worth the cost to avoid it.

Structural Hazard 2: Register File



In clock cycle 5, try read and write to registers simultaneously

Solution for Structural Hazards 2:

- Structural hazard 2: register file
  - Register access takes less that ½ ALU stage time

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
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

=> Use the following convention:

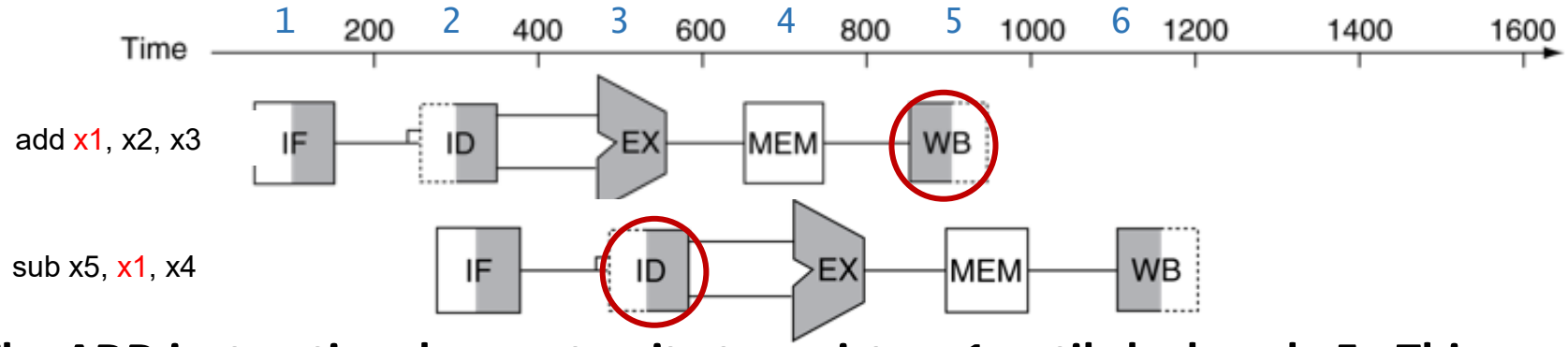
- Always Write during first half of each cycle
- Always Read during second half of each cycle

- Both, Read and Write can be performed during the same clock cycle (a small delay between)

Data Hazards in Pipeline Diagram

- **data hazard** occurs when a planned instruction cannot execute in the proper clock cycle because the data that is needed to execute this instruction is not yet available.

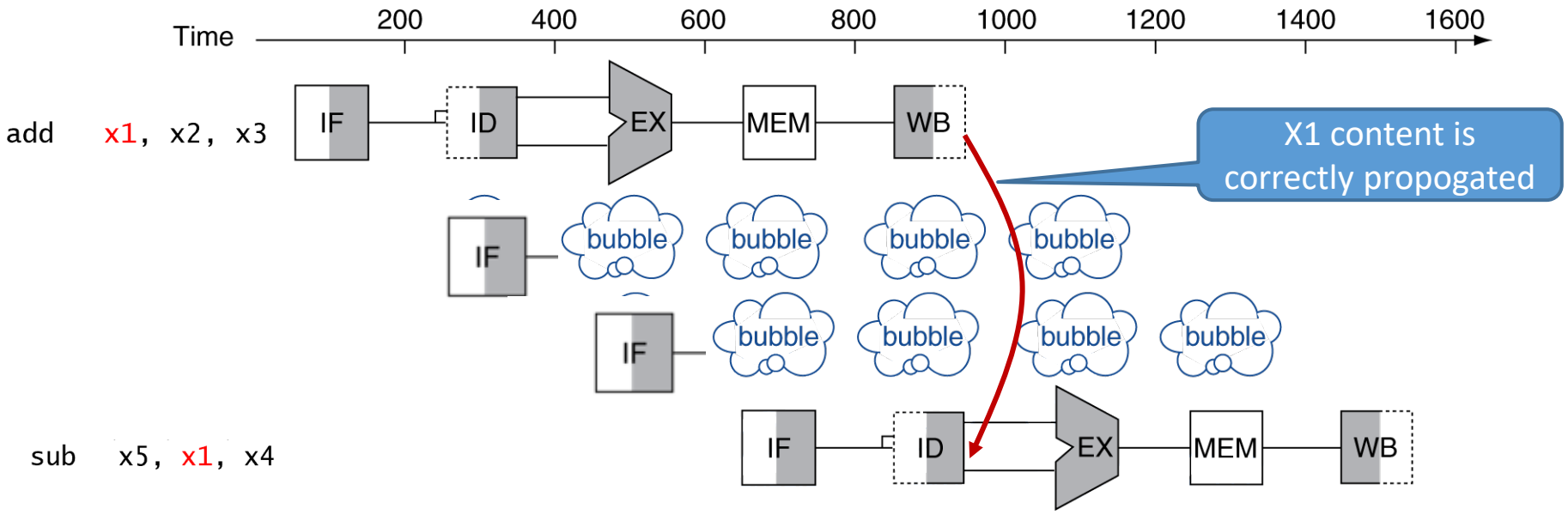
```
add   x1, x2, x3
sub   x5, x1, x4
```



- **The ADD instruction does not write to register x1 until clock cycle 5. This causes a data hazard in our current pipelined datapath.**
  - The SUB reads register x1 in cycle 3. Since ADD hasn't modified the register yet, this will be the *old* value of x1, not the new one.

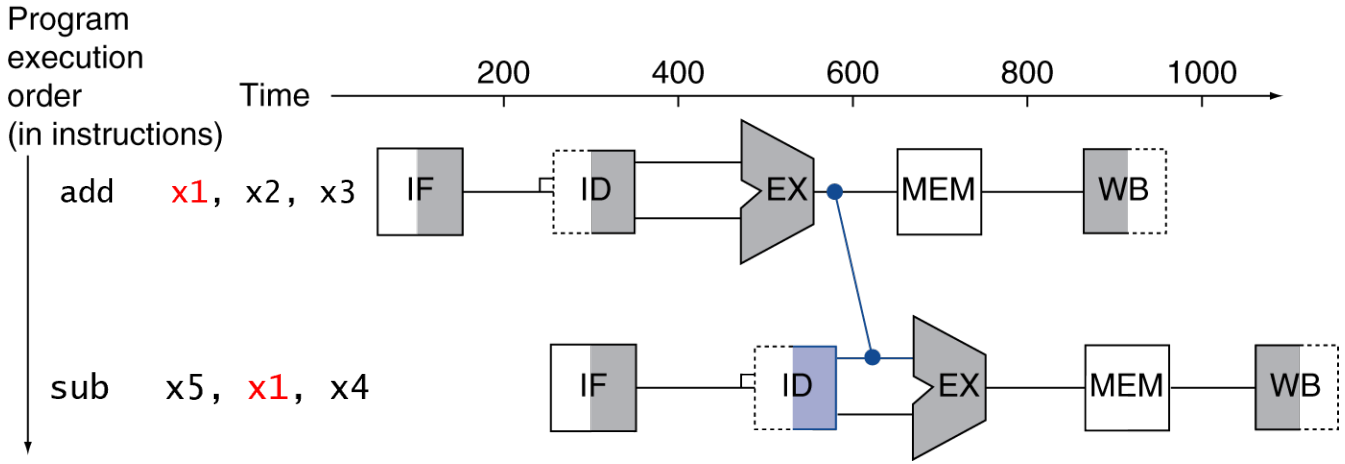
Solution #1 - Stall Pipeline (Or Add Bubbles)

- stalling the sub instruction for 2 cycles.
  - Insert no-op instructions, e.g. “add x0,x0,x0”, as bubble
  - **Waste 400: two instructions in between since sub needs to wait for two stages for add to write the result x1 to register**



Solution #2 - Forwarding (aka Bypassing)

- The method involves retrieving the data from internal buffers rather than waiting for the data to be updated in the register file or data memory.



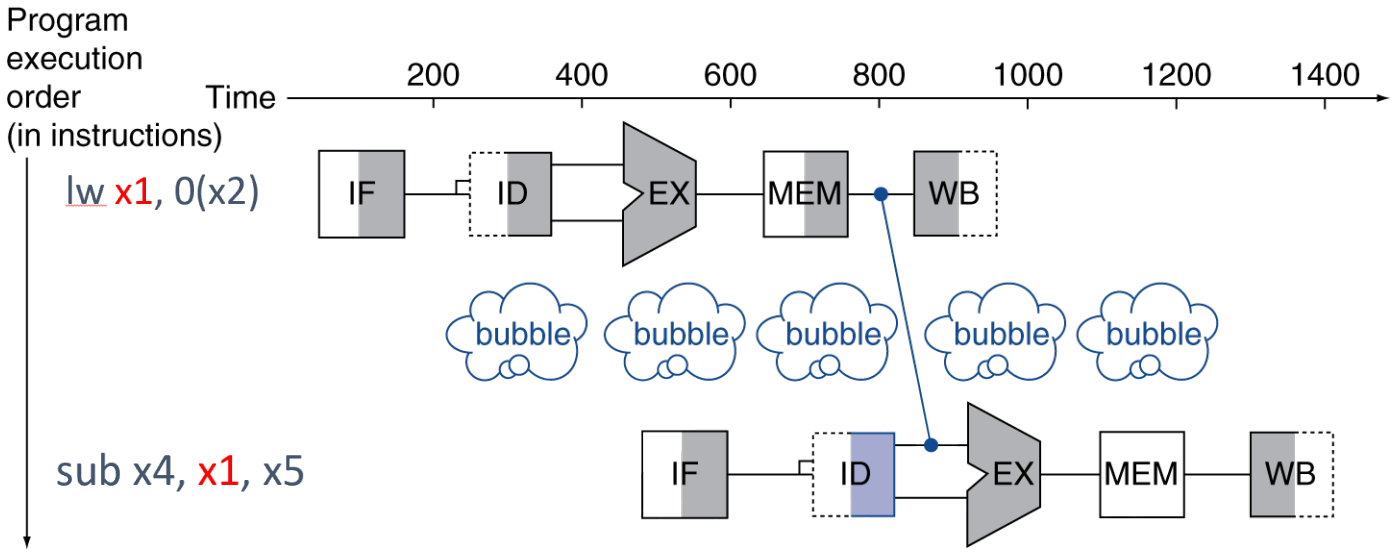
- Because the result of the add operation is available at the end of the EX stage, we can grab its value for use in sub instruction.

Forwarding Data Hazard

- We can't always avoid stalls by forwarding, considering the following instructions

  lw x1, 0(x2)

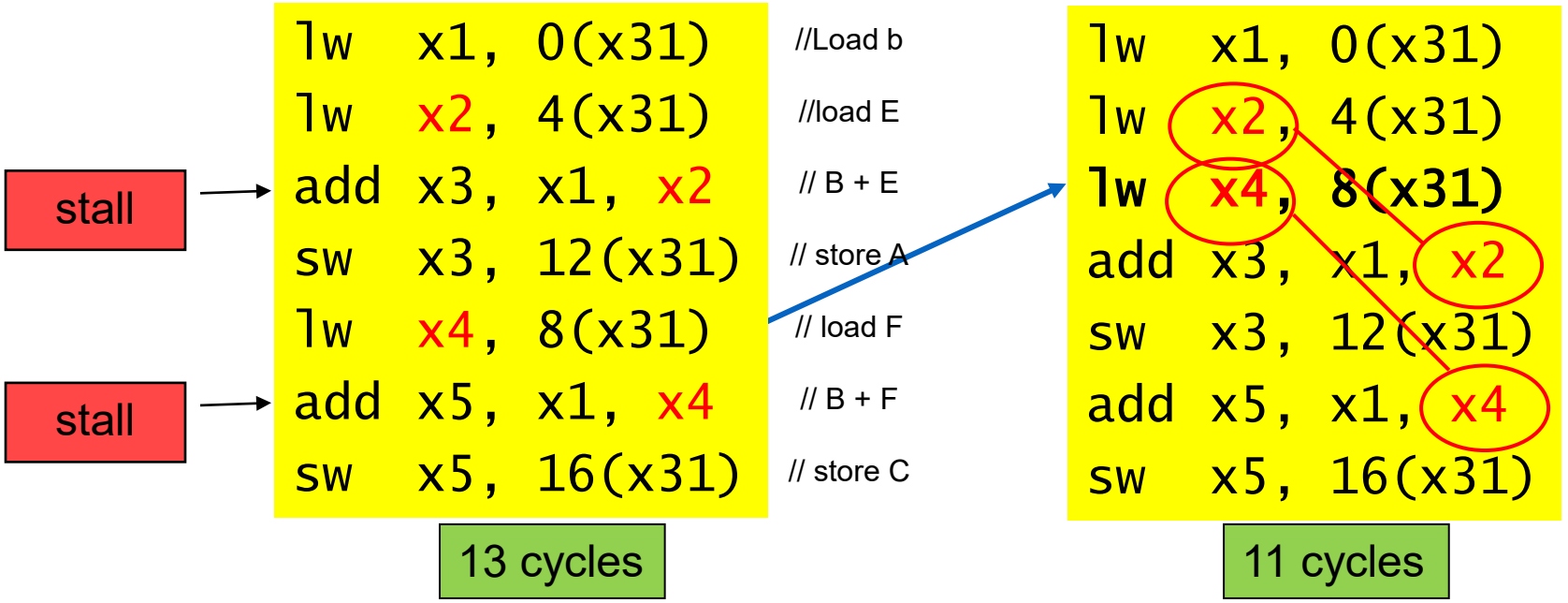
  sub x4, x1, x5



- Even if we use forwarding, the new contents of x1 are only available after load word's MEM stage. So we'll have to stall the sub instruction one cycle.

Solution #3 – Reorder code

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



Reorder Code

- If we are using a pipelined processor with forwarding, we have the following stages executing in each cycle:

cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
lw $t1,0($t0)	IF	ID	EX	MEM	WB								
lw $t2,4($t0)		IF	ID	EX	MEM	WB							
add $t3,$t1,$t2				IF	ID	EX	MEM	WB					
sw $t3,12($t0)					IF	ID	EX	MEM	WB				
lw $t4,8($t0)						IF	ID	EX	MEM	WB			
add $t5,$t1,$t4								IF	ID	EX	MEM	WB	
sw $t5,16($t0)									IF	ID	EX	MEM	WB

Reorder Code

- The reordering allows us to execute the program in two fewer cycles than before.

cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
lw $t1,0($t0)	IF	ID	EX	MEM	WB								
lw $t2,4($t0)		IF	ID	EX	MEM	WB							
lw $t4,8($t0)			IF	ID	EX	MEM	WB						
add $t3,$t1,$t2				IF	ID	EX	MEM	WB					
sw $t3,12($t0)					IF	ID	EX	MEM	WB				
add $t5,$t1,$t4						IF	ID	EX	MEM	WB			
sw $t5,16($t0)							IF	ID	EX	MEM	WB		

Control Hazard (Branch Hazard)

- occurs when the flow of instruction addresses is not known at the time that the next instruction must be loaded.
- Assume the following instructions

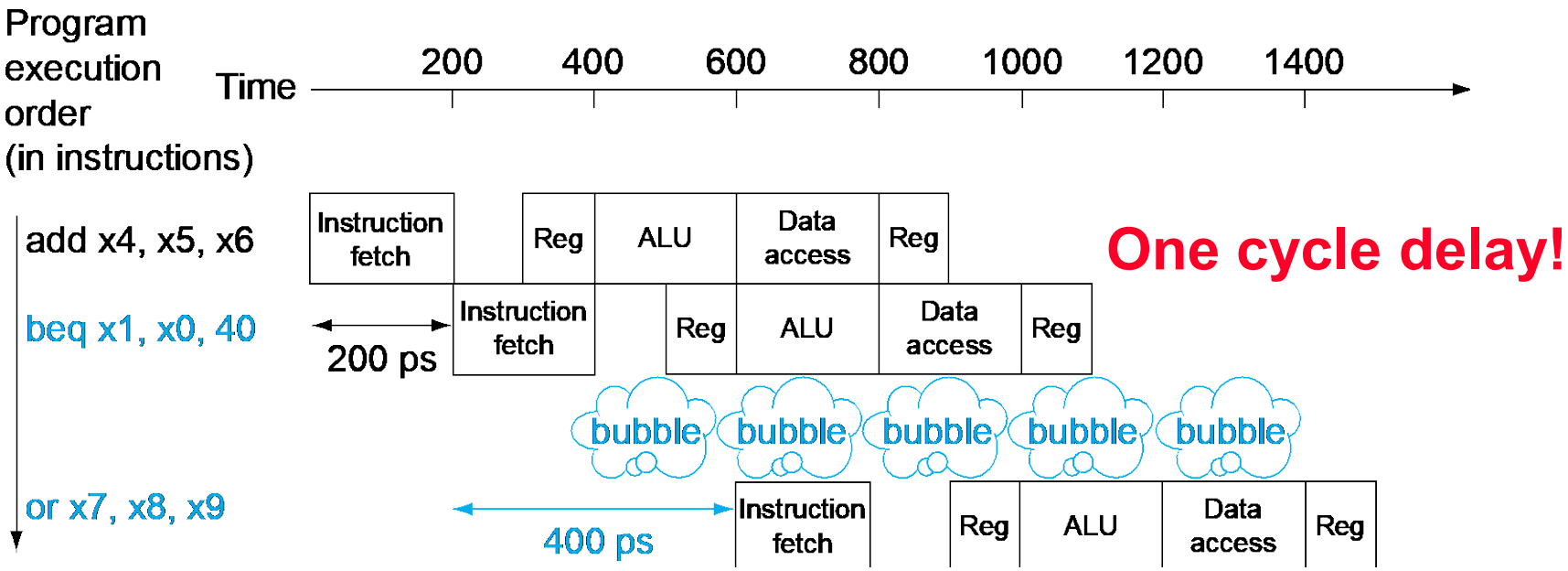
```
beq x3, x2, L1
sub x2, x3, x4
```

- We have a problem: we do not know what the next instruction should be until the end of the third cycle. But we're automatically fetching the next instruction in the second cycle. We will run into a similar problem with jumps as well.

cycle	1	2	3	4	5	6
beq	IF	ID	EX	MEM	WB	
sub		IF	ID	EX	MEM	WB

Solution #1 - Stall on Branch

- stall until branch outcome becomes available.
  - One cycle stall (bubble) if branch condition is determined at ID stage
    - Even with extra hardware to compute PC at ID stage, 1 extra stall is needed
  - Two cycles stall if branch condition is determined at EXE stage



Solution #2 - Branch Prediction

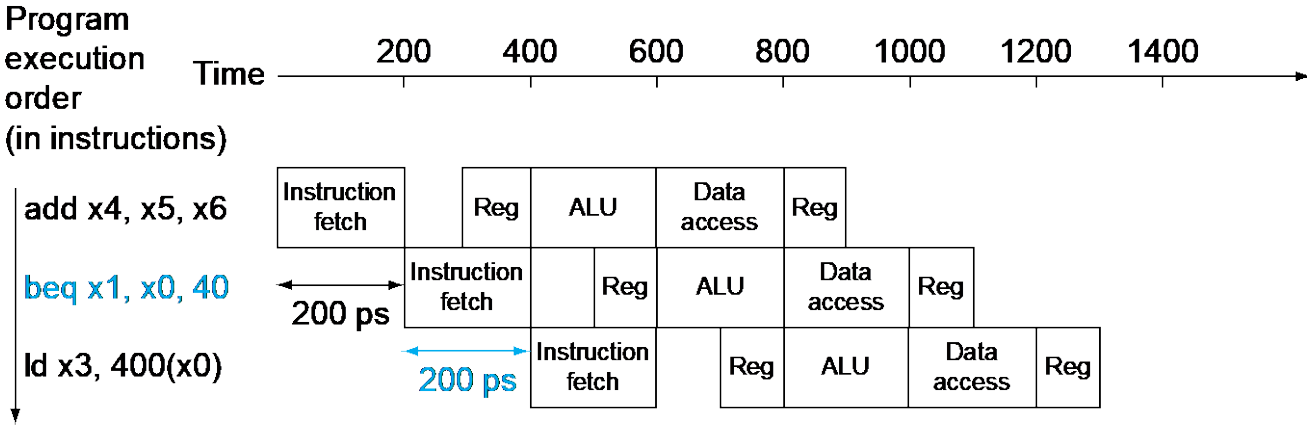
- If we stall until the branch target is known, we will always incur a penalty for stalling
  - Stall penalty becomes unacceptable
- **Predict** the outcome and act accordingly
  - Only stall if prediction is wrong
  - we will only incur a penalty when the branch actually is taken

- In RISC-V pipeline
  - Predict branches are not taken
  - Fetch instruction after branch, with no delay

RISC-V with Predict Not Taken

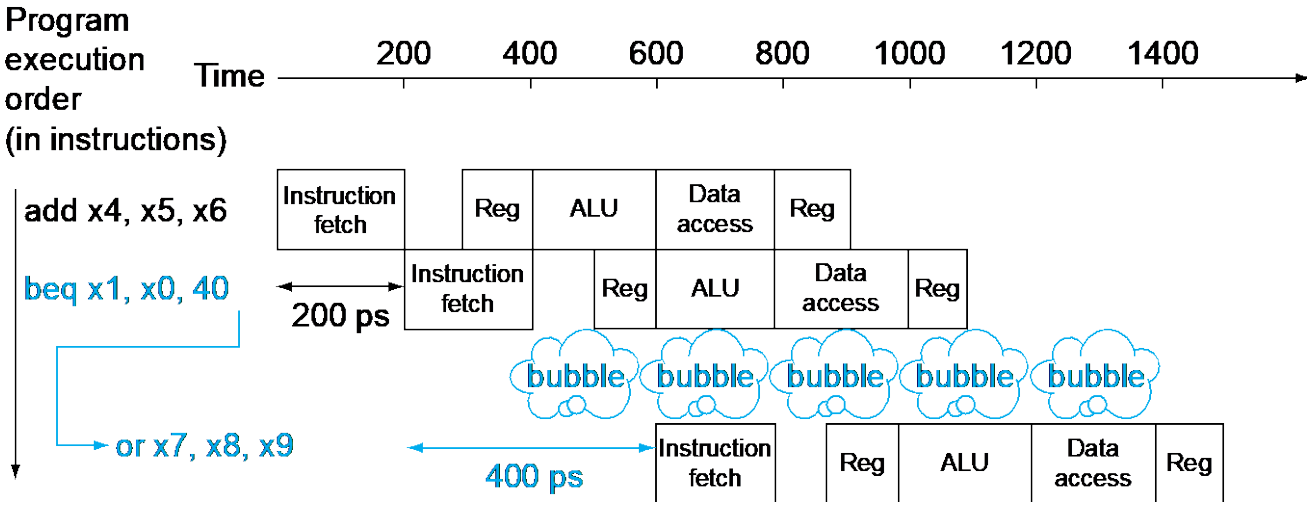
When predicting that a branch is not taken, we proceed as normal.
If the branch is not taken there is no issue

Prediction correct



If the branch is taken, however, we incur a stalling penalty.

Prediction incorrect



Exercises

- For the code sequence below, state whether it must stall, can avoid stalls using only forwarding, or can execute without stalling or forwarding.

```
lw x2, 0(x1)
add x3, x2, x2
```

Answer: We will have to stall while waiting for the result of lw.

Exercises

- For the code sequence below, state whether it must stall, can avoid stalls using only forwarding, or can execute without stalling or forwarding.

```
add x1, x0, x0
addi x1, x0, 5
addi x4 ,x1, 5
```

Answer: We can forward the results of add to the second addi instruction.

More-Realistic Branch Prediction

- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch as taken or untaken
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history

Pipeline Summary

- Pipelining improves performance by increasing instruction throughput rather than individual instruction execution time
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
    - Not reduce the time it takes to complete an individual instruction.
- Subject to hazards
  - Structure, data, control
- Instruction set design affects complexity of pipeline implementation

Pipeline Execution Diagram: Steps

1. Identify RAW dependencies between two instructions **that are one after the other or there is one instruction in between**
   - AL-Use: 2-cycle delay without forwarding, no delay with forwarding
   - Load-Use: 2-cycle delay without forwarding, 1 cycle delay with forwarding
     - With forwarding, we can reschedule load to eliminate the 1 cycle delay even with forwarding
   - No need to looking for RAW dependency between instructions that are far from each other (>=1 instructions in between)
     - Thus only check for the two instructions that could be executed **one after another or has one other instruction in between**
2. Identify branch instruction
   - 1 cycle delay (or two cycles delay) depending on the implementation (question)
3. Pipeline diagrams (4 situations)
   - a) No pipeline at all, one cycle per stage, no overlap
   - b) Pipeline with no forwarding, 2 cycle delay for AL-USE, Load-USE, beq (EXE outcome)
   - c) Pipeline with forwarding, 1 cycle delay for Load-use, and 2 cycle-delay for beq
   - d) Pipeline with forwarding and load-use rescheduling: reschedule the instruction to eliminate the 1-cycle delay for load-use

- No any two instructions can be in the same stage in the same cycle
  - Structural hazard

```
for (i=1; i<M-1; i++) B2[i] = B[i-1] + B[i] + B[i+1];
```

- Base address B and B2 are in register x22 and x23. i is stored in register x5, M is stored in x4.

Using beq (==) for (<) to exit

```
                add x5, x0, 1       // i=0
                add x22, x4, -1     // loop bound x22 has M-1
LOOP: beq x5, x22, Exit
                slliw x6, x5, 2     // x6 now store i*4, slliw is i<<2 (shift left logic)
                add x7, x22, x6     // x7 now stores address of B[i].
                lw x9, 0(x7)        // load B[i] from memory location (x7+0) to x9
                lw x10, -4(x7)      // load B[i-1] to x10
                add x9, x10, x9     // x9 = B[i] + B[i-1]
                lw x10, 4(x7)       //load B[i+1] to x10
                add x9, x10, x9     // x9 = B[i-1] + B[i] + B[i+1]
                add x8, x23, x6     // x8 now stores the address of B2[i]
                sw x9, 0(x8)        // store value for B2[i] from register x9 to
memory (x8+0)
                addi x5, x5, 1      // i++
                beq x0, x0, LOOP
```

Exit:

```
for (i=1; i<M-1; i++) B2[i] = B[i-1] + B[i] + B[i+1];
```

- Base address B and B2 are in register x22 and x23. i is stored in register x5, M is stored in x4.

Using beq (==) for (<) to exit

```
1.                  add x5, x0, 1
2.                  add x22, x4, -1
3.      LOOP: beq x5, x22, Exit
4.                  slliw x6, x5, 2
5.                  add x7, x22, x6
6.                  lw x9, 0(x7)
7.                  lw x10, -4(x7)
8.                  add x9, x10, x
9.                  lw x10, 4(x7)
10.                 add x9, x10, x
11.                 add x8, x23, x
12.                 sw x9, 0(x8)
13.                 addi x5, x5, 1
14.                 beq x0, x0, LOO
15.     Exit:
```

RAW Dependencies

Instruction that writes the register	Instruction that reads the register	The register	# instructions in between	Load-use
add x5, x0, 1	beq x5, x22, Exit	x5	1	
add x22, x4, -1	beq x5, x22, Exit	x22	0	
slliw x6, x5, 2	add x7, x22, x6	x6	0	
add x7, x22, x6	lw x9, 0(x7)	x7	0	
add x7, x22, x6	lw x10, -4(x7)	x7	1	
lw x9, 0(x7)	add x9, x10, x9	x9	1	Y
lw x10, -4(x7)	add x9, x10, x9	x10	0	Y
lw x10, 4(x7)	add x9, x10, x9	x10	0	Y
add x9, x10, x9	sw x9, 0(x8)	x9	1	
add x8, x23, x6	sw x9, 0(x8)	x8	0	
addi x5, x5, 1	beq x5, x22, Exit	x5	1	

Instruction that writes the register	Instruction that reads the register	The register	In instructions in between	Load-use
add x5, x0, 1	beq x5, x22, Exit	x5	1	
add x22, x4, -1	beq x5, x22, Exit	x22	0	
lw x10, -4(x7)	add x9, x10, x9	x10	0	Y
lw x10, 4(x7)	add x9, x10, x9	x10	0	Y

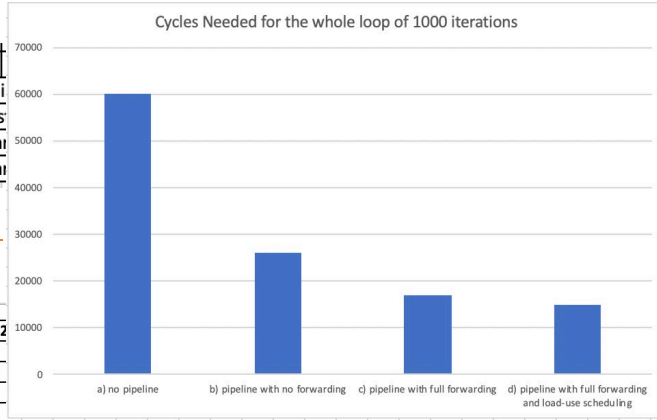
	Cycles Needed for the whole loop of 1000 iterations	
a) no	60000	Each iteration has 12 instructions, 5 cycles to finish each instruction. Thus each iteration needs 12*5 cycles, for total 1000 iterations, it needs 60000 plus the 10 cycles for the very first two instructions.
b) pip	26000	Each iteration needs 26 cycles, thus 26000 cycles for 1000 iterations. 26006 is the actual number of cycles if we count the starting 2 cycles and the ending 4 cycles.
c) pip	17000	First iteration takes 16 cycles, other iterations each takes 17 cycles. 17005 is the actual number of cycles if we count the starting 2 cycles and the ending 4 cycles.
d) pip	15000	First iteration takes 14 cycles, other iterations each takes 15 cycles. 15005 is the actual number of cycles if we count the starting 2 cycles and the ending

			Cycles																													
	a)	no pipeline	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
		add x5, x0, 1	IF	ID	EX	ME	WB																											
		add x22, x4, -1						IF	ID	EX	ME	WB																					
	LOOP	beq x5, x22, Exit											IF	ID	EX	ME	WB																
		slliw x6, x5, 2																IF	ID	EX	ME	WB												
		add x7, x22, x6																					IF	ID	EX	ME	WB							
		lw x9, 0(x7)																										IF	ID	EX	ME	WB			
		lw x10, -4(x7)																															IF	ID	EX
		add x9, x10, x9																																	
		lw x10, 4(x7)																																	
		add x9, x10, x9																																	
	b)	pipeline no forwarding	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
		add x5, x0, 1	IF	ID	EX	ME	WB																											
		add x22, x4, -1		IF	ID	EX	ME	WB																											
Two cycles delay because RAW hazard from add to beq for x22	LOOP	beq x5, x22, Exit					IF	ID	EX	ME	WB																							
Two cycles delay because of beq		slliw x6, x5, 2						IF	ID	EX	ME	WB																						
Two cycles deplay because of RAW hazards from slliw to add for x6		add x7, x22, x6								IF	ID	EX	ME	WB																			
Two cycles delay because of RAW hazard from add to lw for x7		lw x9, 0(x7)									IF	ID	EX	ME	WB																		
		lw x10, -4(x7)										IF	ID	EX	ME	WB																		
Two cycles delay because Load-Use RAW hazard from lw to add for x10		add x9, x10, x9											IF	ID	EX	ME	WB																
		lw x10, 4(x7)												IF	ID	EX	ME	WB																
Two cycles delay because Load-Use RAW hazard from lw to add for x10		add x9, x10, x9													IF	ID	EX	ME	WB														
		add x8, x23, x6														IF	ID	EX	ME	WB														
Two cycles delay because of RAW hazard from add to sw for x8		sw x9, 0(x8)															IF	ID	EX	ME	WB												
		addi x5, x5, 1																IF	ID	EX	ME	WB												
		beq x0, x0, LOOP																	IF	ID	EX	ME	WB											
Two cycles delay because of beq	second iteration	beq x5, x22, Exit																		IF	ID	EX	ME	WB										
		slliw x6, x5, 2																			IF	ID	EX	ME	WB									
		add x7, x22, x6																				IF	ID	EX	ME	WB								
		lw x9, 0(x7)																					IF	ID	EX	ME	WB							
		lw x10, -4(x7)																						IF	ID	EX	ME	WB						
		add x9, x10, x9																							IF	ID	EX	ME	WB					
		lw x10, 4(x7)																								IF	ID	EX	ME	WB				
		add x9, x10, x9																									IF	ID	EX	ME	WB			
		add x8, x23, x6																										IF	ID	EX	ME	WB		
		sw x9, 0(x8)																											IF	ID	EX	ME	WB	
		addi x5, x5, 1																												IF	ID	EX	ME	WB
		beq x0, x0, LOOP																													IF	ID	EX	ME	WB

start

The first iteration, 26 cycles

	Cycles Needed for the whole loop of 1000 iterations	
a) no	60000	Each iteration has 12 instructions, 5 cycles to finish each instruction. Thus each iteration needs 12*5 cycles, for total 1000 i
b) pip	26000	Each iteration needs 26 cycles, thus 26000 cycles for 1000 iterations. 26006 is the actual number of cycles if we count the s
c) pip	17000	First iteration takes 16 cycles, other iterations each takes 17 cycles. 17005 is the actual number of cycles if we count the star
d) pip	15000	First iteration takes 14 cycles, other iterations each takes 15 cycles. 15005 is the ac Vertical (Value) Axis e count the star



				1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42
		c)	pipeline with forwar																																										
			add x5, x0, 1	IF	ID	EX	ME	WB																																					
			add x22, x4, -1		IF	ID	EX	ME	WB																																				
		LOOP	beq x5, x22, Exit			IF	ID	EX	ME	WB																																			
	Two cycles delay because of beq		sllіw x6, x5, 2						IF	ID	EX	ME	WB																																
			add x7, x22, x6							IF	ID	EX	ME	WB																															
			lw x9, 0(x7)								IF	ID	EX	ME	WB																													
	load-use hazard, 1 cycle delay		**lw x10, -4(x7)**									IF	ID	EX	ME	WB																											
			add x9, x10, x9											IF	ID	EX	ME	WB																									
	load-use hazard, 1 cycle delay		**lw x10, 4(x7)**												IF	ID	EX	ME	WB																								
			add x9, x10, x9													IF	ID	EX	ME	WB																							
			add x8, x23, x6														IF	ID	EX	ME	WB																						
			sw x9, 0(x8)															IF	ID	EX	ME	WB																					
			addi x5, x5, 1																IF	ID	EX	WB																					
			beq x0, x0, LOOP																	IF	ID	EX	WB																				
	2 cycles delay because of beq		beq x5, x22, Exit																				IF	ID	EX	ME	WB																
	2 cycles delay because of beq		sllіw x6, x5, 2																					IF	ID	EX	ME	WB															
			add x7, x22, x6																						IF	ID	EX	ME	WB														
			lw x9, 0(x7)																							IF	ID	EX	ME	WB													
			lw x10, -4(x7)																									IF	ID	EX	ME	WB											
			add x9, x10, x9																											IF	ID	EX	ME	WB									
			lw x10, 4(x7)																													IF	ID	EX	ME	WB							
			add x9, x10, x9																															IF	ID	EX	ME	WB					
			add x8, x23, x6																																	IF	ID	EX	ME	WB			
			sw x9, 0(x8)																																			IF	ID	EX	ME	WB		
			addi x5, x5, 1																																				IF	ID	EX	ME	WB	
			beq x0, x0, LOOP																																					IF	ID	EX	ME	WB
		d)	pipeline with forwar	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42
			add x5, x0, 1	IF	ID	EX	ME	WB																																					
			add x22, x4, -1		IF	ID	EX	ME	WB																																				
		LOOP	beq x5, x22, Exit			IF	ID	EX	ME	WB																																			
			sllіw x6, x5, 2						IF	ID	EX	ME	WB																																
			add x7, x22, x6							IF	ID	EX	ME	WB																															
			lw x9, 0(x7)								IF	ID	EX	ME	WB																													
The four instructions are rescheduled so we have two loads first and then two add, we will need to use one register x11 to help on this			**lw x10, -4(x7)**									IF	ID	EX	ME	WB																											
			lw x11, 4(x7)										IF	ID	EX	ME	WB																											
			add x9, x10, x9											IF	ID	EX	ME	WB																									
			add x9, x11, x9												IF	ID	EX	ME	WB																								
			add x8, x23, x6													IF	ID	EX	ME																								
			sw x9, 0(x8)														IF	ID	EX	WB																							
			addi x5, x5, 1															IF	ID	EX	WB																						
			beq x0, x0, LOOP																IF	ID	EX	WB																					
	2 cycles delay because of beq		beq x5, x22, Exit																			IF	ID	EX	ME	WB																
	2 cycles delay because of beq		sllіw x6, x5, 2																				IF	ID	EX	ME	WB															
			add x7, x22, x6																					IF	ID	EX	ME	WB														
			lw x9, 0(x7)																						IF	ID	EX	ME	WB													
			lw x10, -4(x7)																							IF	ID	EX	ME	WB												
			lw x11, 4(x7)																								IF	ID	EX	ME	WB											
			add x9, x10, x9																									IF	ID	EX	ME	WB										
			add x9, x11, x9																										IF	ID	EX	ME	WB									
			add x8, x23, x6																											IF	ID	EX	ME	WB								
			sw x9, 0(x8)																												IF	ID	EX	ME	WB							
			addi x5, x5, 1																													IF	ID	EX	ME	WB						
			beq x0, x0, LOOP																														IF	ID	EX	ME	WB					

The first iteration, 16 cycles

The second iteration, and the rest ..., 17 cycles

The first iteration, 14 cycles

The second iteration, and the rest ..., 15 cycles

MeltDown & Spectre

- https://www.youtube.com/watch?v=bs0xswK0eZk

Exceptions and Interrupts

- "Unexpected" events requiring change in flow of control
  - Different ISAs use the terms differently
    - E.g., Intel x86 uses interrupt
- Exception
  - Arises within the CPU
    - e.g., undefined opcode, overflow, syscall, …
- Interrupt
  - From an external I/O controller
- Dealing with them without sacrificing performance is hard

Handling Exceptions

- Save PC of offending (or interrupted) instruction
  - In MIPS: Exception Program Counter (EPC)
- Save indication of the problem
  - In MIPS: Cause register
    - We'll assume 1-bit: 0 for undefined opcode, 1 for overflow
  - In RISC-V: Supervisor Exception Cause Register
    - There is a field that encodes the two possible exception:
      2 for an undefined instruction, 12 for hardware malfunction

An Alternate Mechanism

- To use Vectored Interrupts
  - Handler address determined by the cause
- Example:
  - Undefined opcode: C000 0000
  - Overflow: C000 0020
  - …: C000 0040
- Instructions either
  - Deal with the interrupt, or
  - Jump to real handler

Handler Actions

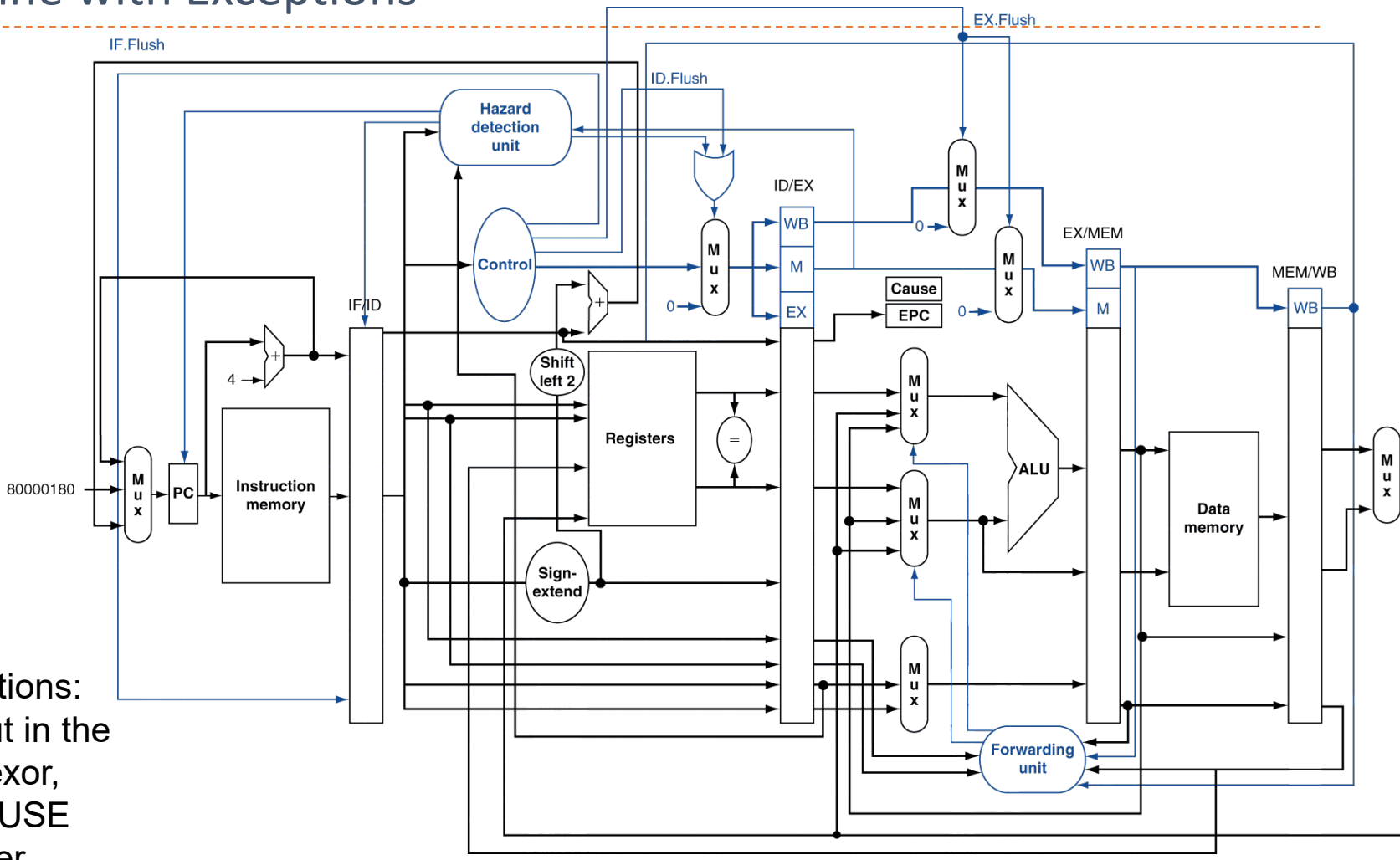
- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
  - Take corrective action
  - use EPC to return to program
- Otherwise
  - Terminate program
  - Report error using EPC, cause, …

Exceptions in a Pipelined Implementation

- Treat it as another form of control hazard
- Consider overflow on add in EX stage

  `add $1, $2, $1`
  - Prevent $1 from being clobbered
  - Complete previous instructions
  - Flush `add` and subsequent instructions
  - Set SCause and SEPC register values
  - Transfer control to handler
- Similar to mispredicted branch
  - Use much of the same hardware

Pipeline with Exceptions



Key additions:
A new input in the multiplexor,
An SCAUSE register,
An SEPC register

Exception Properties

- Restartable exceptions
  - Pipeline can flush the instruction
  - Handler executes, then returns to the instruction
    - Refetched and executed from scratch
- PC saved in EPC register
  - Identifies causing instruction
  - Actually PC + 4 is saved
    - Handler must adjust

Exception Example

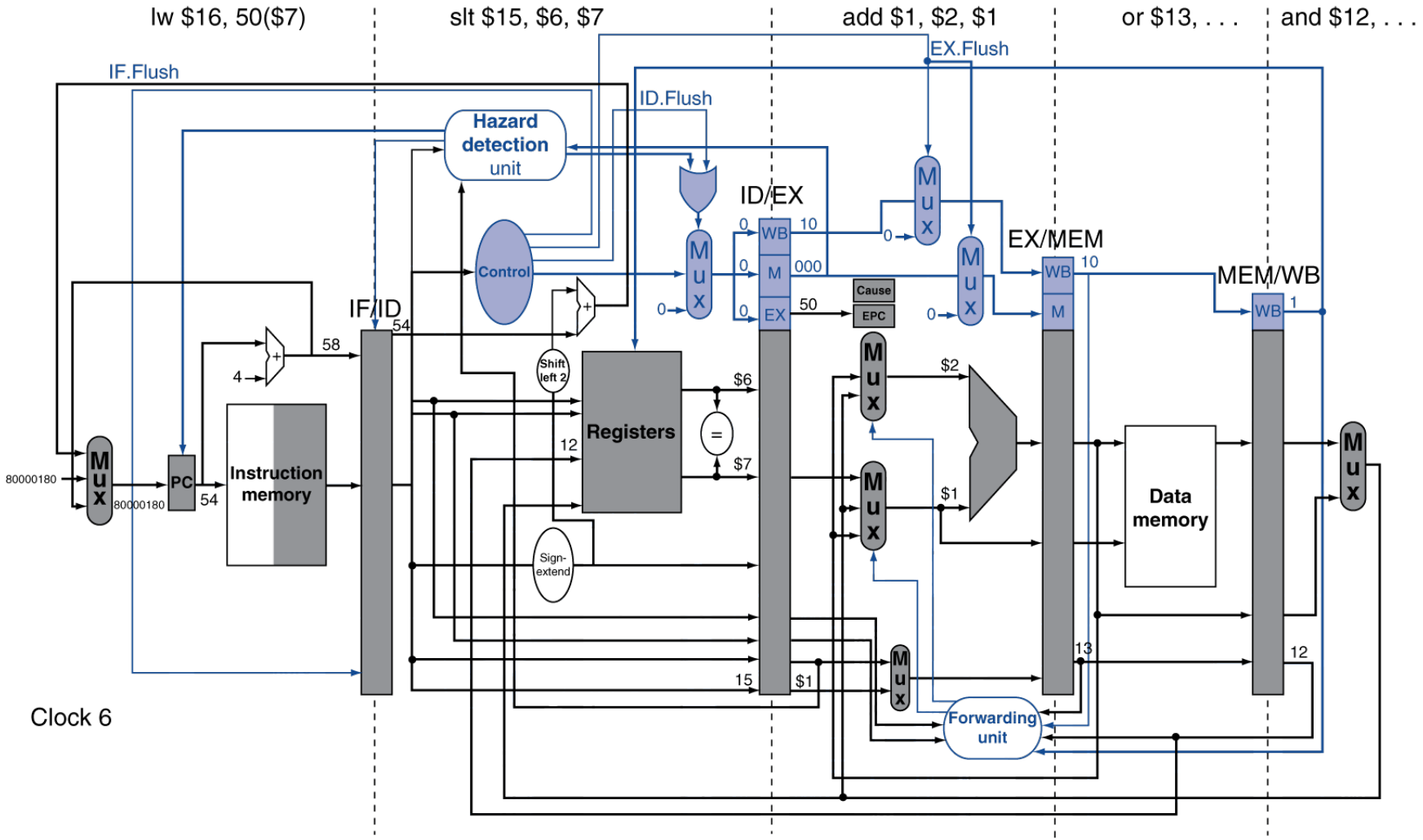
- Show in the pipeline if a hardware malfunction Exception occurs in the `add` instruction in

```
40 sub  $11, $2, $4
44 and  $12, $2, $5
48 or   $13, $2, $6
4C add  $1,  $2, $1
50 slt  $15, $6, $7
54 lw   $16, 50($7)
…
```

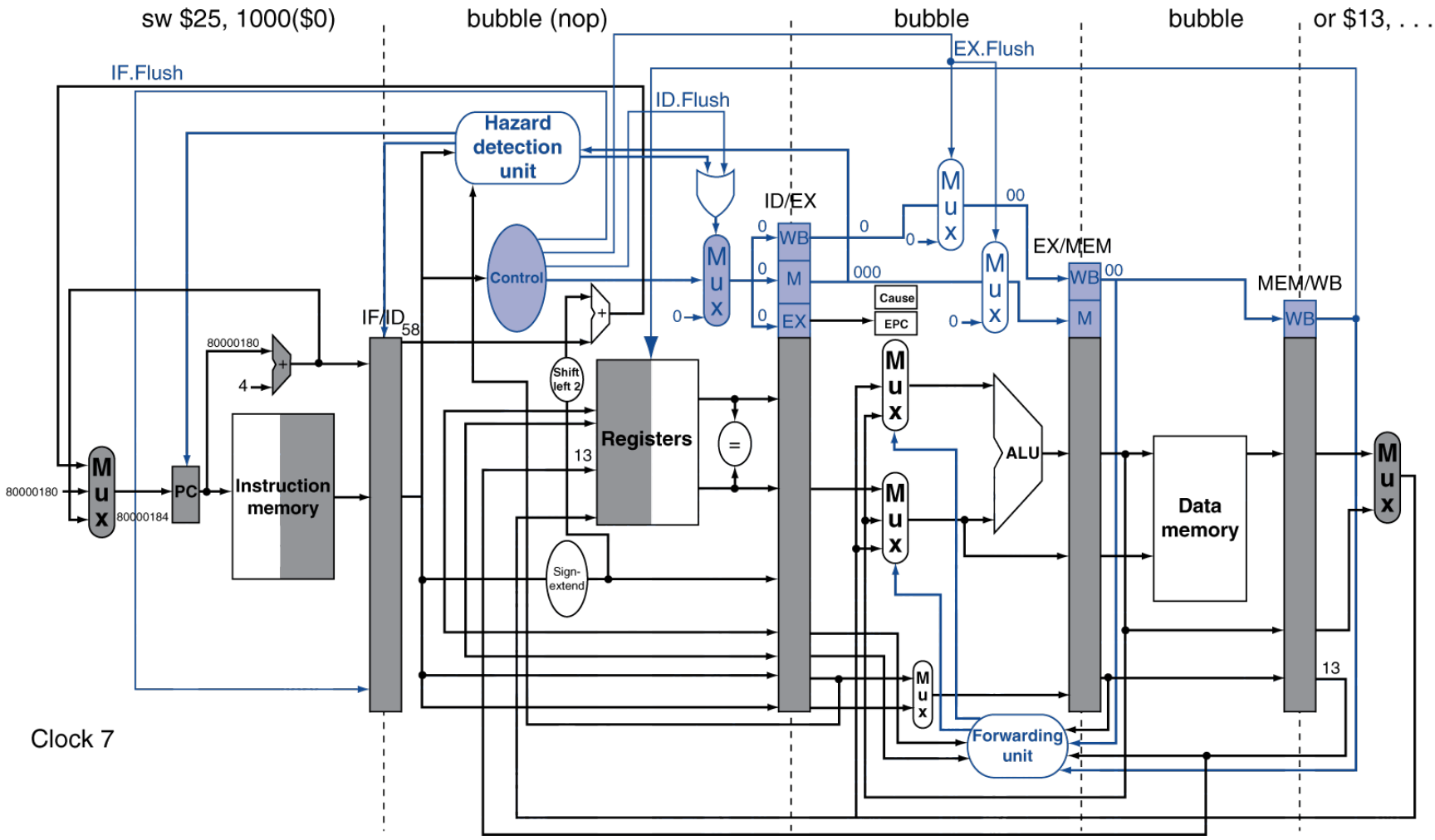
- Handler

```
80000180     sw   $25, 1000($0)
80000184     sw   $26, 1004($0)
…
```

Exception Example



Exception Example



Multiple Exceptions

- Pipelining overlaps multiple instructions
  - Problem: Could have multiple exceptions at once
  - Solution: to prioritize the exceptions to determine which is serviced first.
    - The SEPC capture the address of the interrupted instructions
    - The SCAUSE records the highest priority exception if more than one exception occurs.
- In complex pipelines
  - Multiple instructions issued per cycle
  - Out-of-order completion
  - Maintaining precise exceptions is difficult!

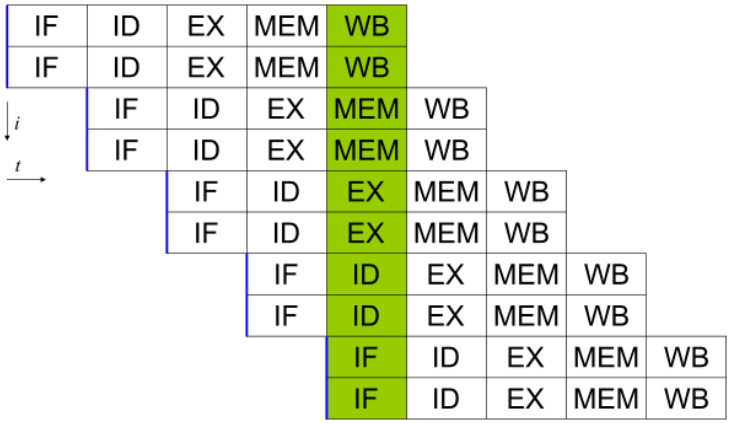
Imprecise Exceptions

- Just stop pipeline and save state
  - Including exception cause(s)
- Let the handler work out
  - Which instruction(s) had exceptions
  - Which to complete or flush
    - May require “manual” completion
- Simplifies hardware, but more complex handler software
- Not feasible for complex multiple-issue out-of-order pipelines

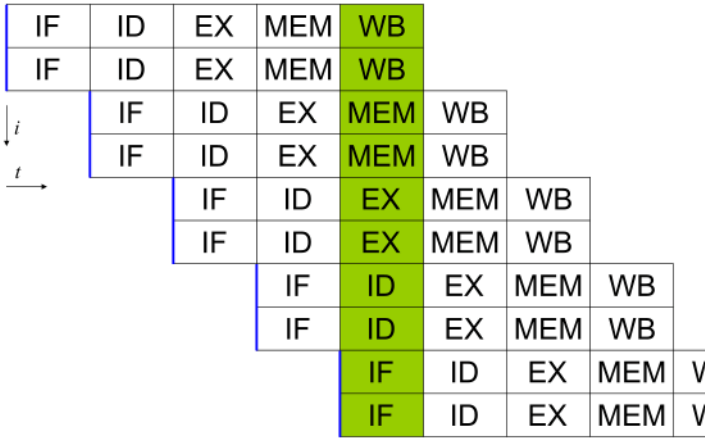
Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel
  - CPI ~= 1
- To increase ILP
  - Deeper pipeline by having more stages
    - Less work per stage ⇒ shorter clock cycle
  - Multiple issue
    - Replicate pipeline stages ⇒ multiple pipelines
    - Start multiple instructions per clock cycle
- Performance of Multiple issue
  - E.g., 4GHz 2-way multiple-issue
  - IPC (Instruction Per Cycle): 2
    - peak CPI = 0.5
    - Instr/Second: $4*10^9 * 2 = 8*10^9$
  - But dependencies reduce this in practice
    - Pipeline hazards for single issue happen
    - Not always have two instruction to be issued per cycle

	Clock number								
Instruction number	1	2	3	4	5	6	7	8	9
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i*+1		IF	ID	EX	MEM	WB			
Instruction *i*+2			IF	ID	EX	MEM	WB		
Instruction *i*+3				IF	ID	EX	MEM	WB	
Instruction *i*+4					IF	ID	EX	MEM	WB



Multiple Issue

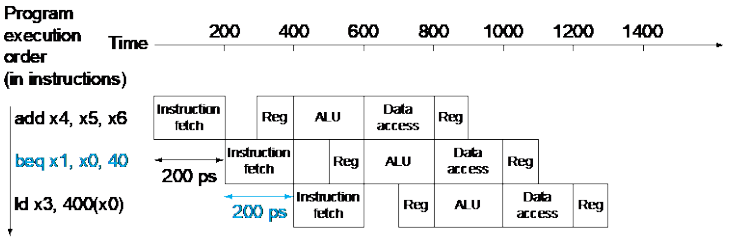


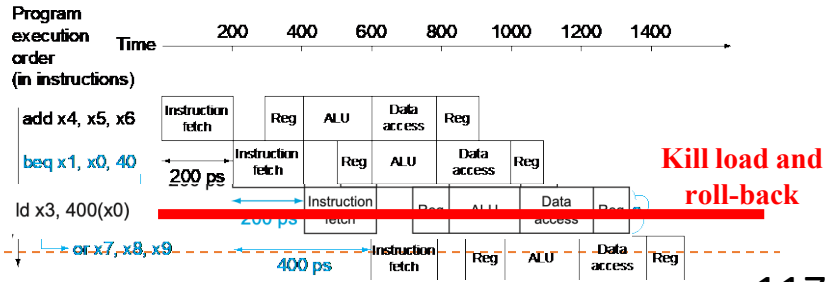
- Static multiple issue
  - Compiler groups instructions to be issuec together
  - Packages them into “issue slots”
  - Compiler detects and avoids hazards
- **Dynamic multiple issue**
  - **CPU examines instruction stream and chooses instructions to issue each cycle**
  - **Compiler can help by reordering instructions**
  - **CPU resolves hazards using advanced techniques at runtime**

High-end processors (desktop, server) use dynamic multiple issue

Speculation

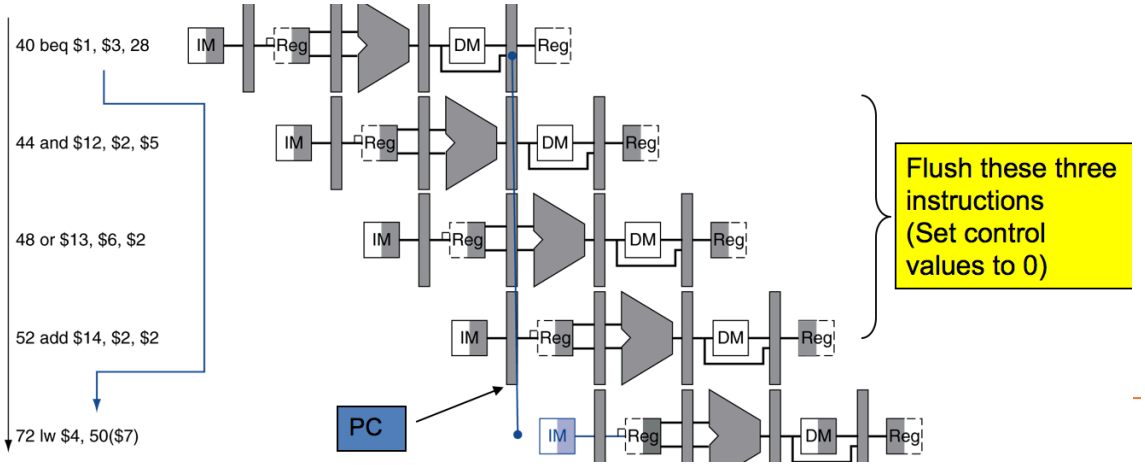
- "Guess" what to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right
    - If so, complete the operation
    - If not, roll-back and do the right thing
- Examples
  - Speculate on branch outcome
    - The instruction after branch executed earlier
    - Roll back if path taken is different
  - Speculate on load
    - a store that precedes a load does not Refer to the same address.
    - Roll back if location is updated





Compiler/Hardware Speculation

- Compiler can reorder instructions
  - e.g., move load before branch
  - Can include "fix-up" instructions to recover from incorrect guess
    - Move lw to remove load-use cycle delay in RAW hazards
    - Schedule delayed slot
- Hardware can look ahead for instructions to execute
  - Buffer results until it determines they are actually needed
  - Flush buffers on incorrect speculation



Static Multiple Issue

- Compiler groups instructions into “issue packets”
  - Group of instructions that can be issued on a single cycle
    - **2 IPC: ALU/BEQ + LW/SW**
  - Determined by pipeline resources required
- Think of an issue packet as a very long instruction
  - Specifies multiple concurrent operations
  - ⇒ Very Long Instruction Word (VLIW)

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

Static two-issue pipline in operation

Scheduling Static Multiple Issue

- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - No dependencies with a packet
  - Possibly some dependencies between packets
    - Varies between ISAs; compiler must know!
  - Pad with nop if necessary
    - If one instruction of the pair cannot be used

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

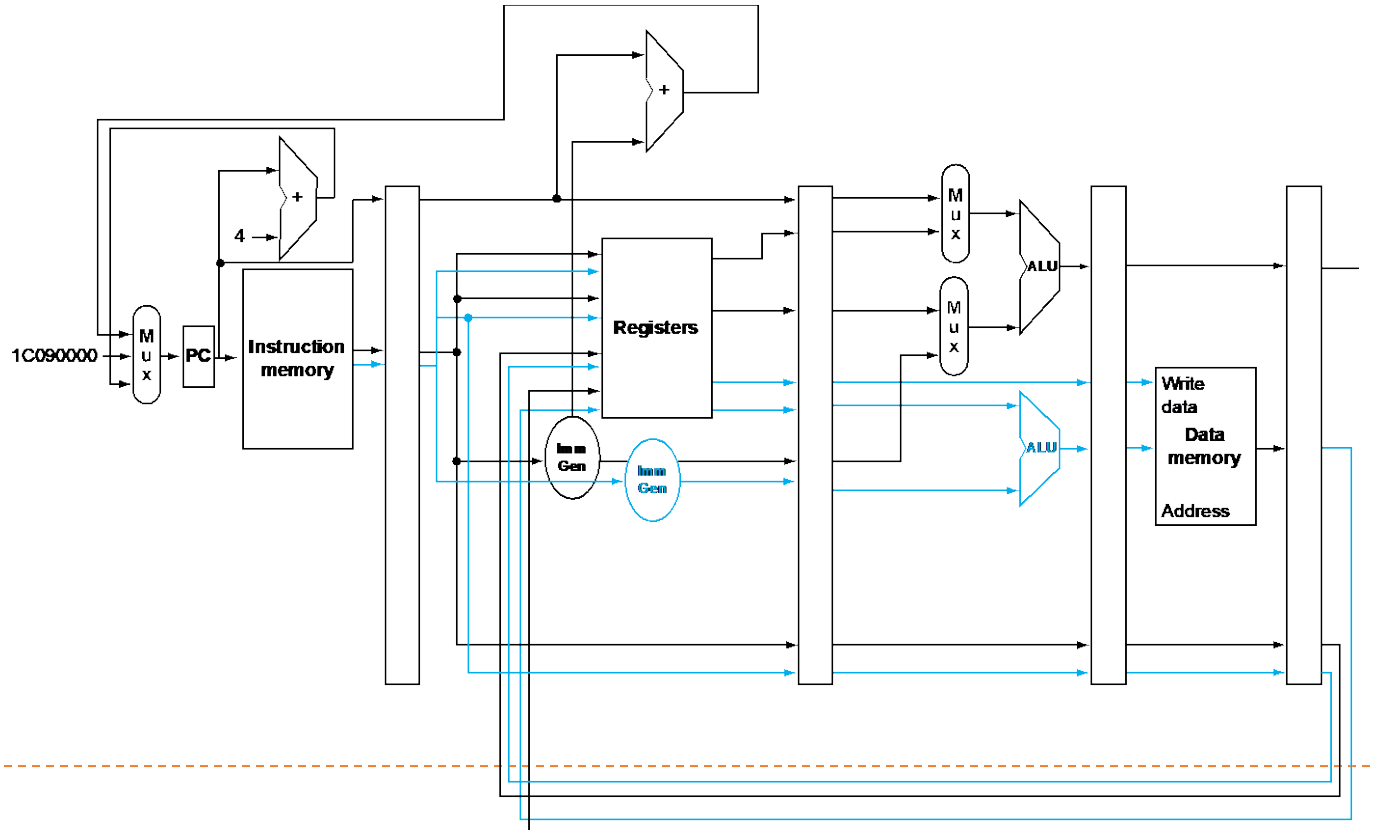
RISC-V with Static Dual Issue

- Two-issue packets
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction with nop

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

RISC-V with Static Dual Issue

- Double resources
  - 2 set of register R/W ports
  - 2 ALUs
    - Bottom handles address calculation for Load/store



Hazards in the Dual-Issue RISC-V

- More instructions executing in parallel
- EX data hazard
  - Forwarding avoided stalls with single-issue
  - Now can't use ALU result in load/store in same packet
    - ```
      add  x10, $s0, $s1
      load $s2, 0(x10)
      ```
    - Split into two packets, effectively a stall
- Load-use hazard
  - Still one cycle use latency, but now two instructions
- More aggressive scheduling required
      ```

```
for (i=1000;i!=0;i--)
  A[i] += a;
```
Each element is 8 bytes

Scheduling Example

- Schedule this for dual-issue RISC-V

```
Loop: ld   x31, 0(x20)      # x31=array element
      add  x31, x31, x21    # add scalar in x21
      sd   x31, 0(x20)      # store result
      addi $20, x20, -8     # decrement pointer
      blt  x22, x20, Loop # compare to loop limit
                            # branch if x20 > x22
```

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:		ld x31,0(x20)	1
	addi x20,x20,-8		2
	add x31,x31,x21		3
	blt x22,x20,Loop	sd x31,8(x20)	4

A[i] += a;

- **addi and ld CANNOT be in one cycle**
  - **Load-use (ld-add) hazard: 1 cycle delay**
- **IPC = 5/4 = 1.25 (c.f. peak IPC = 2)**

Loop Unrolling

```
for (i=1000;i!=0;i--)
  A[i] += a;
```

Unrolling with factor 2

```
for (i=1000;i!=0;i-=2) {
  A[i] += a;;
  A[i-1] += a;
}
```

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead
  - For two calculations, e.g. A[i]+=a
    - 2 beq vs 1 beq; 2 i-1 vs 1 i-2
- Use different registers per replication
  - Called “register renaming”
  - Avoid loop-carried “anti-dependencies”
    - Store followed by a load of the same register
    - Aka “name dependence”
      - Reuse of a register name

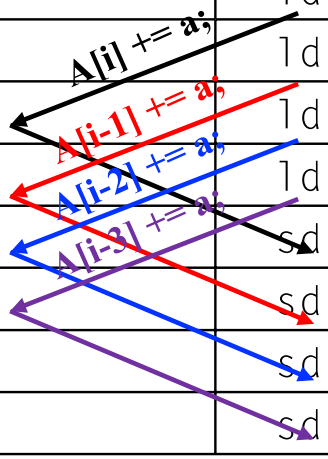
Loop Unrolling Example

Unrolling with factor 4

```
for (i=1000;i!=0;i-=4){
  A[i] += a;
  A[i-1] += a;
  A[i-2] += a;
  A[i-3] += a;
}
```

- Load-use hazard
  - 1 cycle use delay

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:	addi x20,x20,-32	ld x28,0(x20)	1
		ld x29,24(x20)	2
	add x28,x28,x21	ld x30,16(x20)	3
	add x29,x29,x21	ld x31,8(x20)	4
	add x30,x30,x21	sd x28,32(x20)	5
	add x31,x31,x21	sd x29,24(x20)	6
		sd x30,16(x20)	7
	blt x22,x20,Loop	sd x31,8(x20)	8



- IPC = 14/8 = 1.75
  - Closer to 2, but at cost of registers and code size

Summary for Four Iterations of the Loop

```
for (i=1000;i!=0;i--)
   A[i] += a;
```

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:		ld x31,0(x20)	1
	addi x20,x20,-8		2
	add x31,x31,x21		3
	blt x22,x20,Loop	sd x31,8(x20)	4

Unrolling with factor 4

```
for (i=1000;i!=0;i-=4){
   A[i] += a;;
   A[i-1] += a;
   A[i-2] += a;
   A[i-3] += a;
}
```

- Original version + single issue
  - Total 20 instructions → ~5 cycles/calculation
- Original version + multi-issue
  - About 4 cycles/calculation
- Unrolling by 4 + single issue
  - 14 instructions → 3.5 cycles/calculation (14/4)
- Unrolling + multi-issue
  - About 2 clocks/calculation (8/4)

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:	addi x20,x20,-32	ld x28,0(x20)	1
		ld x29,24(x20)	2
	add x28,x28,x21	ld x30,16(x20)	3
	add x29,x29,x21	ld x31,8(x20)	4
	add x30,x30,x21	sd x28,32(x20)	5
	add x31,x31,x21	sd x29,24(x20)	6
		sd x30,16(x20)	7
	blt x22,x20,Loop	sd x31,8(x20)	8

Static Multiple Issue

- Compiler groups instructions into "issue packets"
  - Group of instructions that can be issued on a single cycle
    - 2 IPC: ALU/BEQ + LW/SW
  - Determined by pipeline resources required
- Think of an issue packet as a very long instruction
  - Specifies multiple concurrent operations
  - ⇒ Very Long Instruction Word (VLIW)

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	

Dynamic Multiple Issue Processors

- “Superscalar” processors
- CPU decides whether to issue 0, 1, 2, … each cycle
  - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
  - Though it may still help
  - Code semantics ensured by the CPU

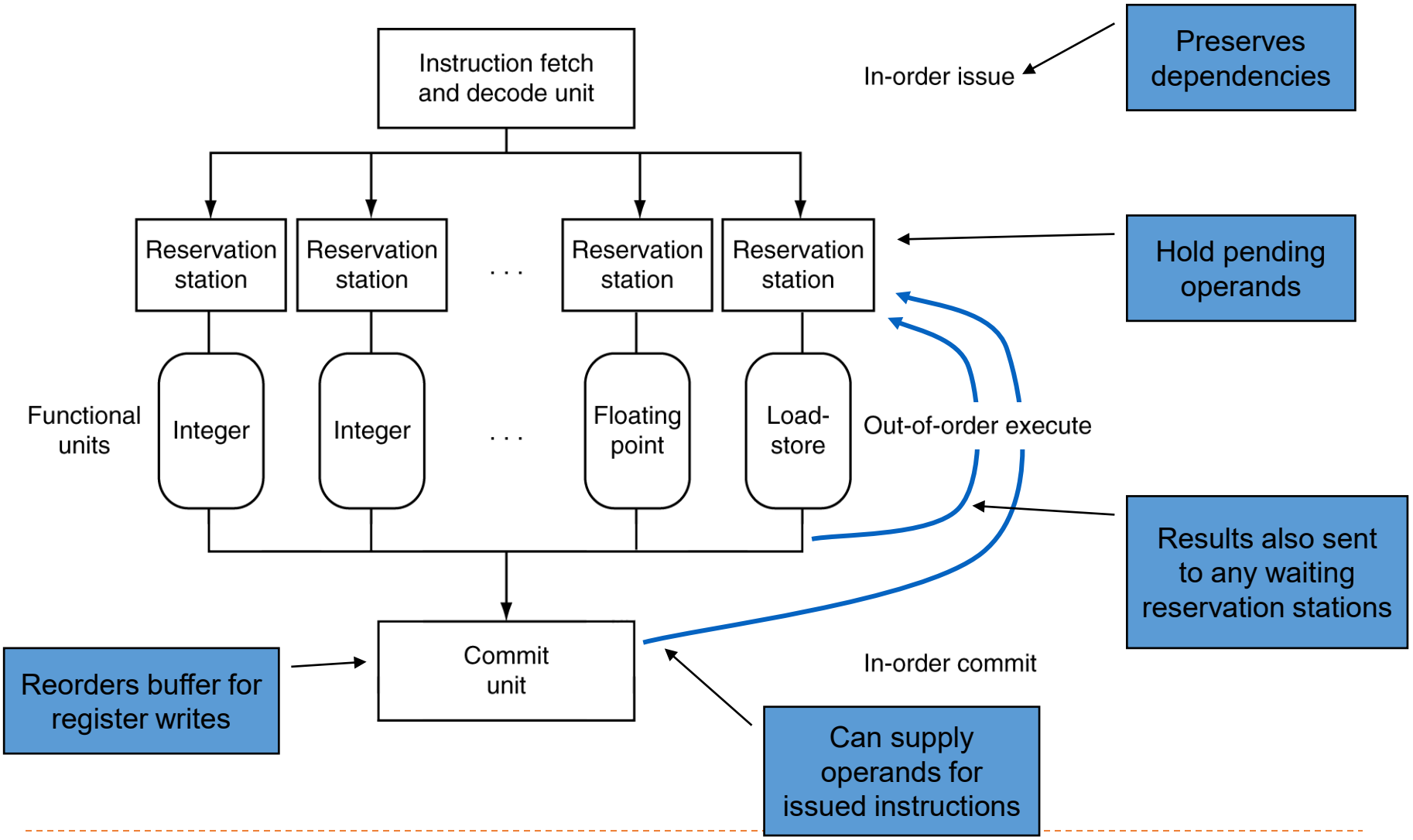
Dynamic Pipeline Scheduling

- Allow the CPU to execute instructions out of order to avoid stalls
  - But commit result to registers in order
- Example

```
ld    x31, 0(x21)
add   x1, x31, x2
sub   x23,x23, x3
andi  x5, x23, 20
```

  - Can start `sub` while `add` is waiting for lw

Dynamically Scheduled CPU



Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
- Not all stalls are predicable
  - e.g., cache misses
- Can't always schedule around branches
  - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards

Does Multiple Issue Work?

- Yes, but not as much as we'd like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
  - e.g., pointer aliasing
- Some parallelism is hard to expose
  - Limited window size during instruction issue
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well

Summary: Extracting More Performance

- To achieve high performance, need both machine parallelism and instruction level parallelism (ILP) by
  - Superpipelining
  - Static multiple-issue (VLIW)
  - Dynamic multiple-issue (superscalar)
- A processor's instruction issue and execution policies impact the available ILP
- Register renaming can solve these storage dependencies

CISC vs RISC vs SS vs VLIW

	CISC	RISC	Superscalar	VLIW
Instr size	variable size	fixed size	fixed size	fixed size (but large)
Instr format	variable format	fixed format	fixed format	fixed format
Registers	few, some special Limited # of ports	Many GP Limited # of ports	GP and rename (RUU) Many ports	many, many GP Many ports
Memory reference	embedded in many instr's	load/store	load/store	load/store
Key Issues	decode complexity	data forwarding, hazards	hardware dependency resolution	(compiler) code scheduling

Power Efficiency

- Complexity of dynamic scheduling and speculations requires power
- Multiple simpler cores may be better

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue Width	Out-of-Order/ Speculation	Cores/ Chip	Power	
Intel 486	1989	25 MHz	5	1	No	1	5	W
Intel Pentium	1993	66 MHz	5	2	No	1	10	W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29	W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75	W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103	W
Intel Core	2006	2930 MHz	14	4	Yes	2	75	W
Intel Core i5 Nehalem	2010	3300 MHz	14	4	Yes	2–4	87	W
Intel Core i5 Ivy Bridge	2012	3400 MHz	14	4	Yes	8	77	W

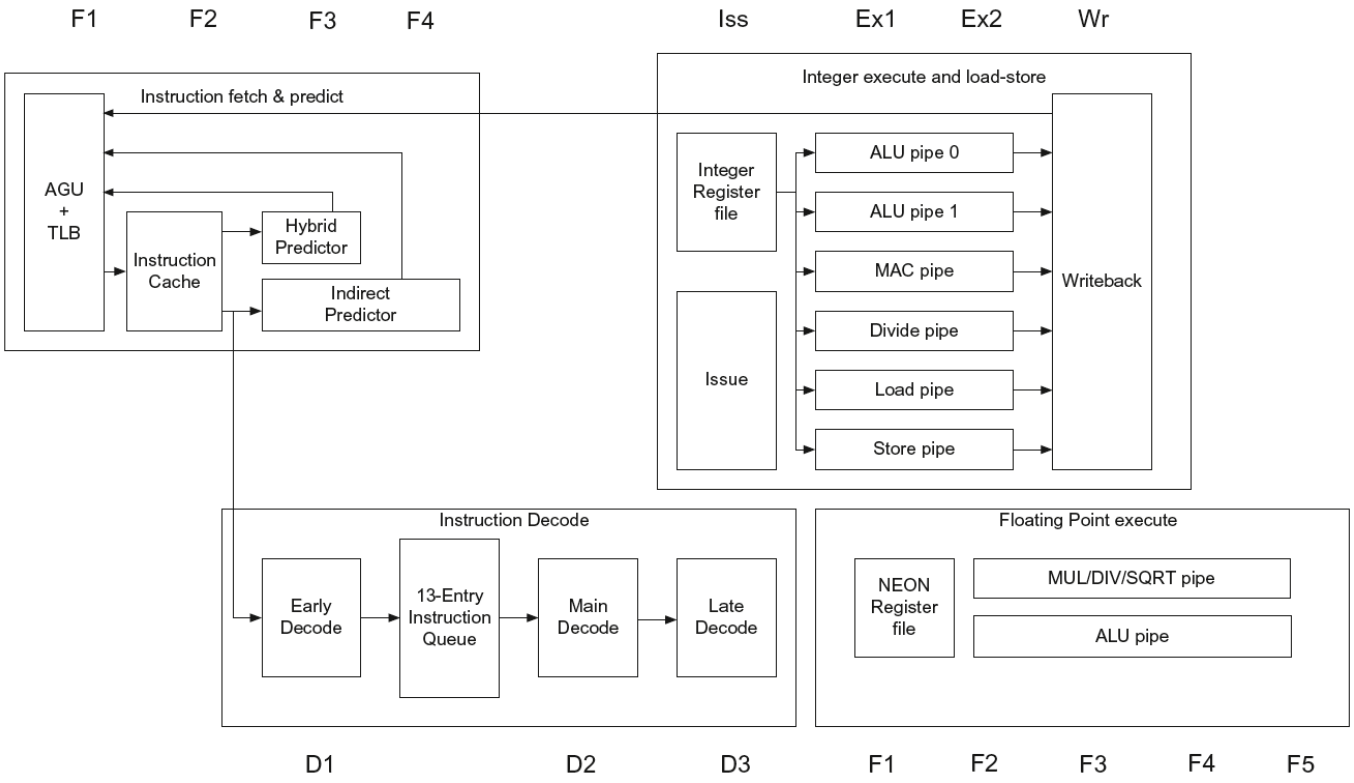
Note: the drop in pipeline stages and power because companies switch to multicore designs

Cortex A53 and Intel i7

Processor	ARM A53	Intel Core i7 920
Market	Personal Mobile Device	Server, cloud
Thermal design power	100 milliWatts (1 core @ 1 GHz)	130 Watts
Clock rate	1.5 GHz	2.66 GHz
Cores/Chip	4 (configurable)	4
Floating point?	Yes	Yes
Multiple issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline stages	8	14
Pipeline schedule	Static in-order	Dynamic out-of-order with speculation
Branch prediction	Hybrid	2-level
1st level caches/core	16-64 KiB I, 16-64 KiB D	32 KiB I, 32 KiB D
2nd level caches/core	128-2048 KiB	256 KiB (per core)
3rd level caches (shared)	(platform dependent)	2-8 MB

ARM Cortex-A53 Pipeline

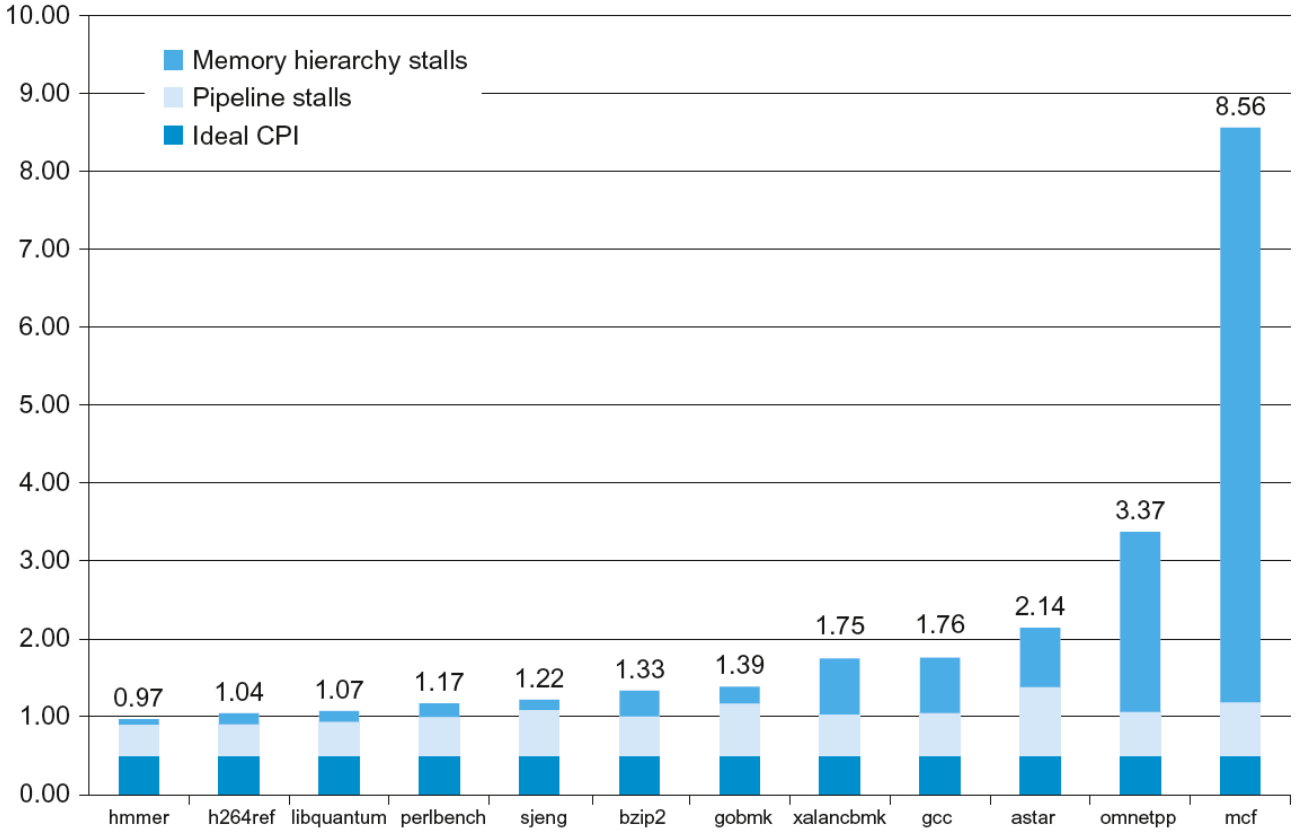
- Used as the basis for several tablets and cell phones
  - Dual-issue, statically scheduled superscalar with dynamic issue detection → 0.5 CPI ideally



The basis structure of the pipeline

ARM Cortex-A53 Performance using SPEC2006

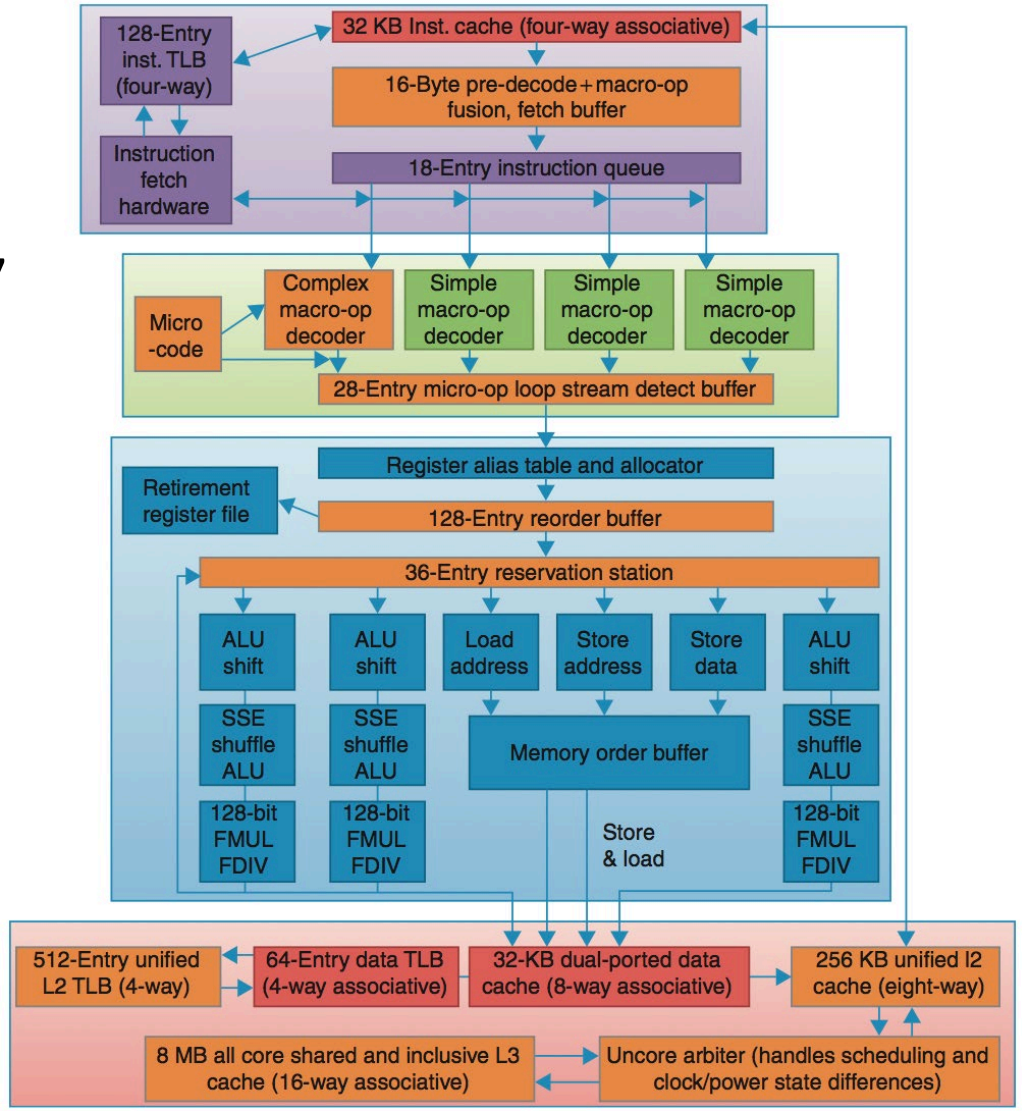
- Ideal CPI: 0.5 since it is 2-way multi-issue (IPC=2)
  - Best case 1.0, median case 1.3, worst 8.6
  - 60% stalls due to pipelining hazards
  - 40% stalls due to the memory hierarchy



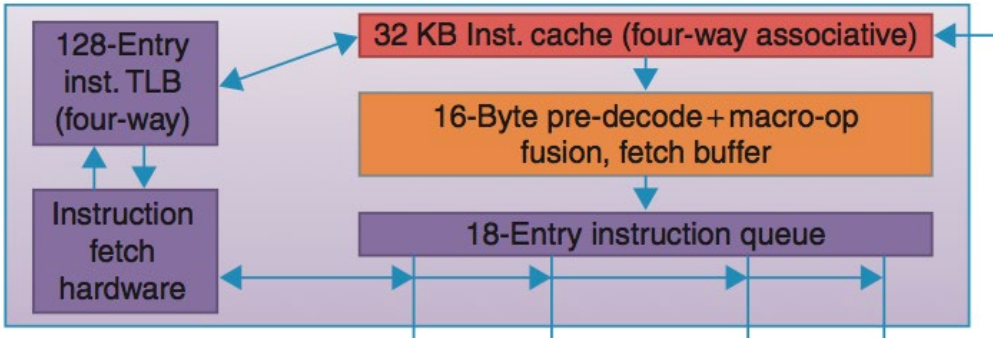
Intel Core i7

- Aggressive out-of-order execution and speculation
- 14 stages pipeline,
- Branch mispredictions costing 17 cycles.
- 48 load and 32 store buffers.
- Six independent functional units
  - 6-wide superscalar

Processor	Intel Core i7 920
Market	Server, Cloud
Thermal design power	130 Watts
Clock rate	2.66 GHz
Cores/Chip	4
Floating point?	Yes
Multiple Issue?	Dynamic
Peak instructions/clock cycle	4
Pipeline Stages	14
Pipeline schedule	Dynamic Out-of-order with Speculation
Branch prediction	2-level
1st level caches/core	32 KiB I, 32 KiB D
2nd level cache/core	256 KiB (per core)
3rd level cache (shared)	2–8 MiB

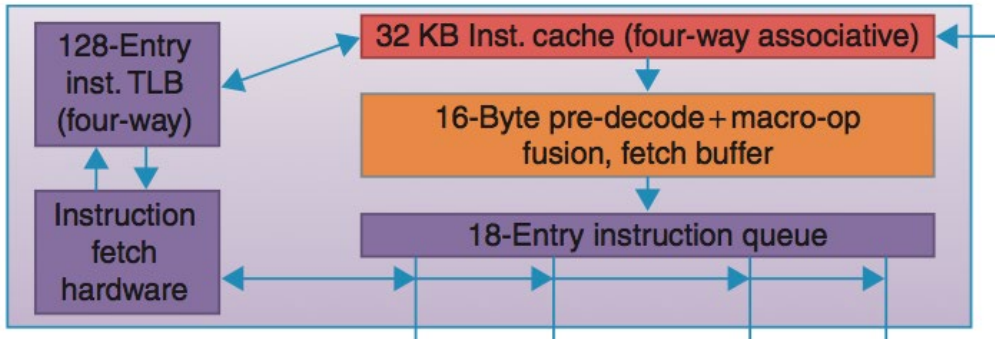


Core i7 Pipeline: IF



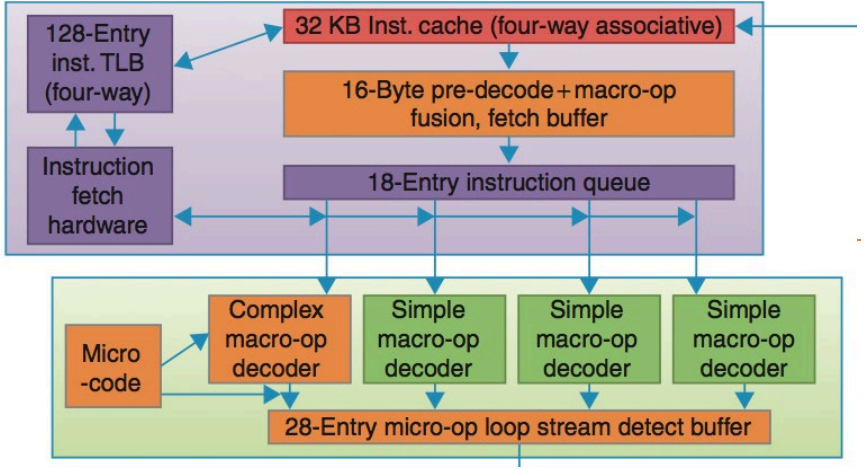
- Instruction fetch – Fetch 16 bytes from the I cache
  - A multilevel branch target buffer to achieve a balance between speed and prediction accuracy.
  - A return address stack to speed up function return.
  - Mispredictions cause a penalty of about 15 cycles.

Core i7 Pipeline: Predecode



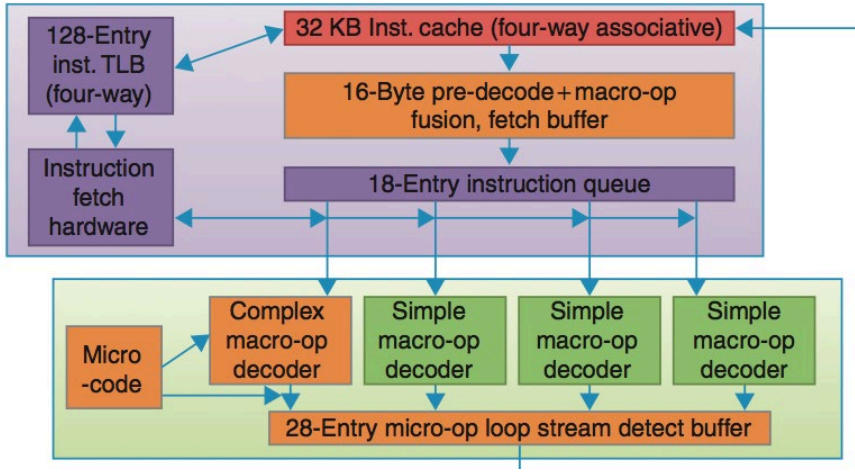
- Predecode –16 bytes instr in the predecode I buffer
  - *Macro-op fusion:* Fuse instr combinations such as compare followed by a branch into a single operation.
  - Instr break down: breaks the 16 bytes into individual x86 instructions.
    - nontrivial since the length of an x86 instruction can be from 1 to 17 bytes and the predecoder must look through a number of bytes before it knows the instruction length.
  - Individual x86 instructions (including some fused instructions) are placed into the 18-entry instruction queue.

Core i7 Pipeline: Micro-op decode



- Micro-op decode – Translate Individual x86 instructions into micro-ops.
  - Micro-ops are simple MIPS-like instructions that can be executed directly by the pipeline (RISC style)
    - introduced in the Pentium Pro in 1997 and has been used since.
  - Three simple micro-op decoders handle x86 instructions that translate directly into one micro-op.
  - One complex micro-op decoder produce the micro-op sequence of complex x86 instr;
    - produce up to four micro-ops every cycle
  - The micro-ops are placed according to the order of the x86 instructions in the 28- entry micro-op loop stream detect buffer.

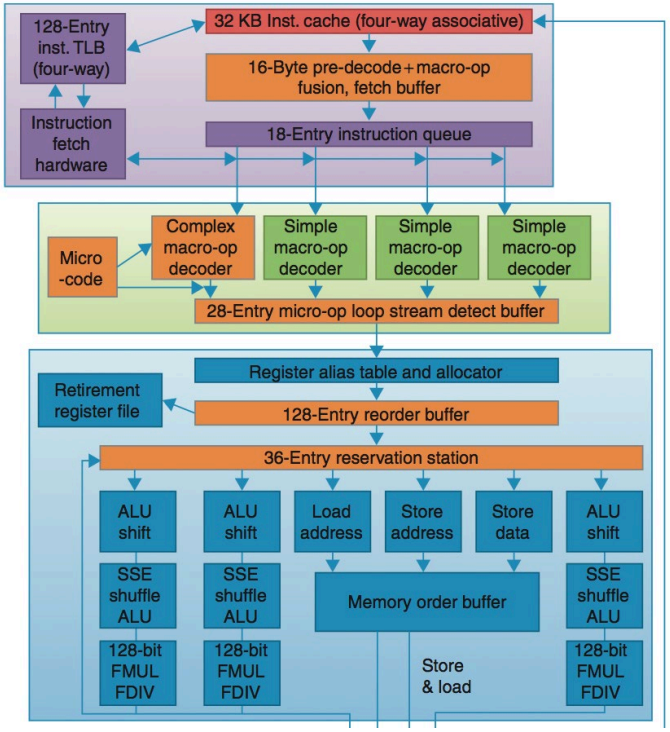
Core i7 Pipeline: loop stream detection and microfusion



- *loop stream detection* and *microfusion by* the micro-op buffer preforms
  - If there is a sequence of instructions (less than 28 instrs in length) that comprises a loop, the loop stream detector will find the loop and directly issue the micro-ops from the buffer
    - eliminating the need for the instruction fetch and instruction decode stages to be activated.
  - Microfusion combines instr pairs such as load/ALU operation and ALU operation/store and issues them to a single reservation station, thus increasing the usage of the buffer.
    - Study comparing the microfusion and macrofusion by Bird et al. [2007] discovered that microfusion had little impact on per- formance, while macrofusion appears to have a modest positive impact on integer performance and little impact on FP.

Core i7 Pipeline: Issue

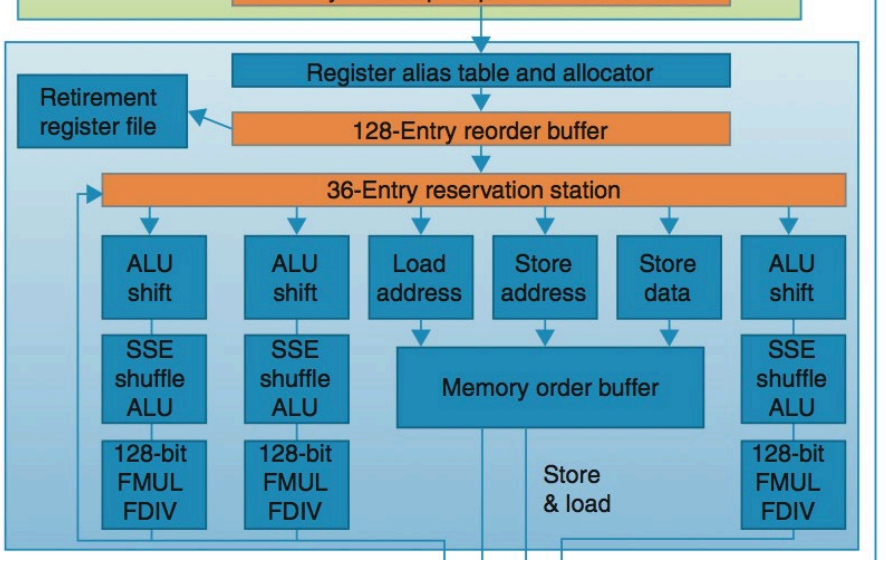
- Basic instruction issue
  - Looking up the register location in the register tables
  - renaming the registers
  - allocating a reorder buffer entry
  - fetching any results from the registers or reorder buffer before sending the micro-ops to the reservation stations.



- 36-entry centralized reservation station shared by six functional units

  Up to six micro-ops may be dispatched to the functional units every clock cycle.

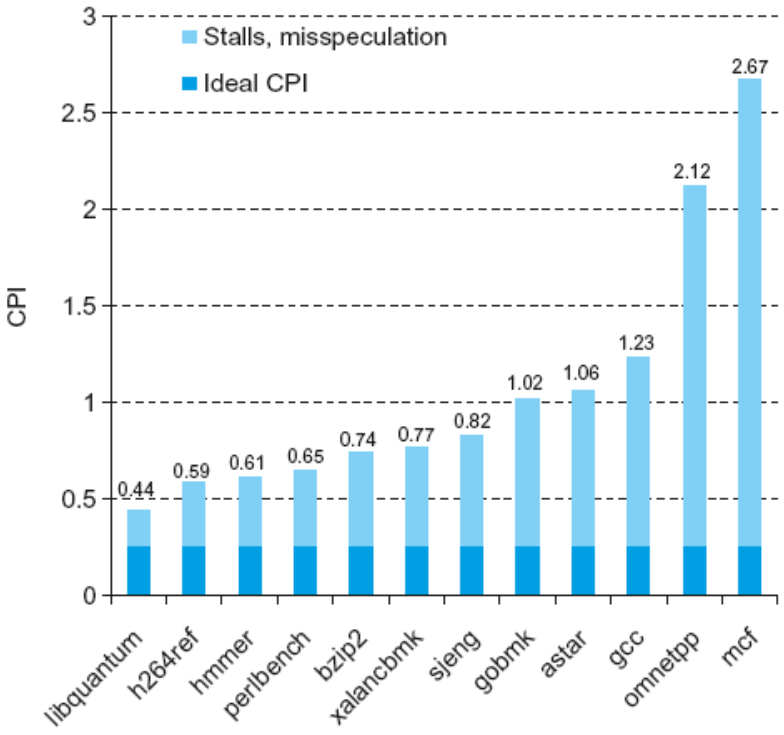
Core i7 Pipeline: EXE and Retirement

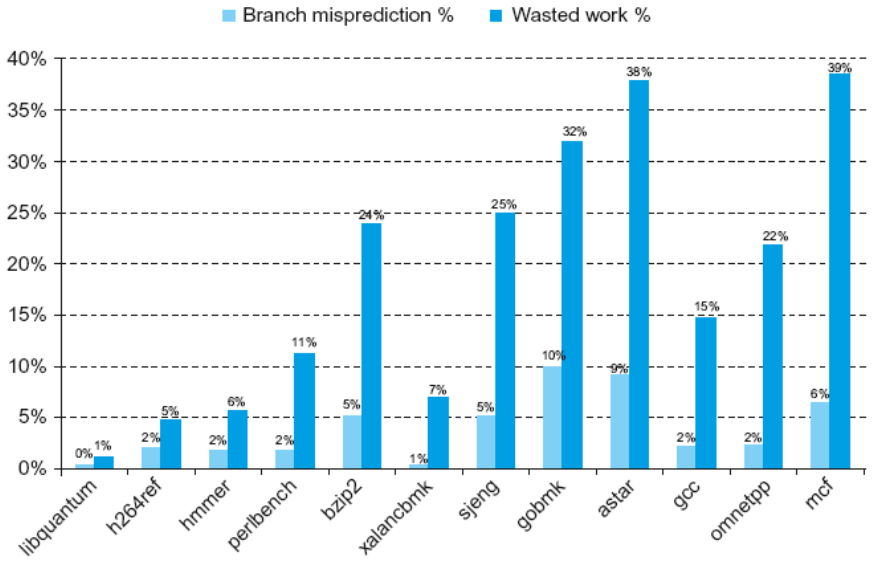


- Micro-ops are executed by the individual function units
  - results are sent back to any waiting reservation station as well as to the register retirement unit, where they will update the register state. The entry corresponding to the instruction in the reorder buffer is marked as complete.
- Retirement
  - When one or more instructions at the head of the reorder buffer have been marked as complete, the pending writes in the register retirement unit are executed, and the instructions are removed from the reorder buffer.

Core i7 Performance running SPEC2006 INT

- Ideal CPI: 0.25
- Best 0.44, median 0.79, worst 2.67;





Concluding Remarks

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control
- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall

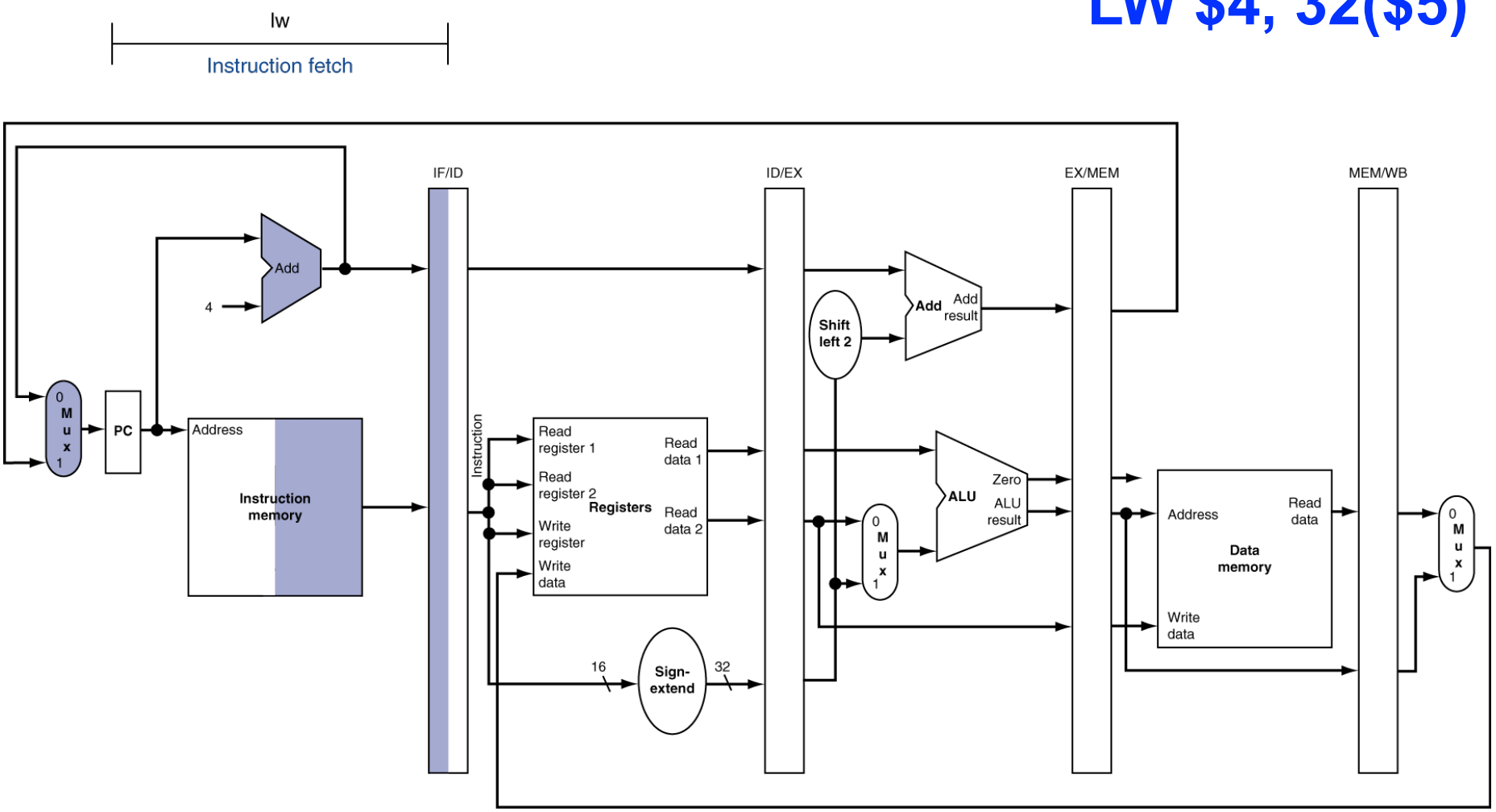
Reference Reading

- 4.6-4.13 4.15 4.16

Following slides will not be presented, but as reference

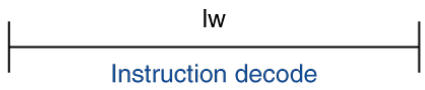
IF for Load, Store, …

LW $4, 32($5)

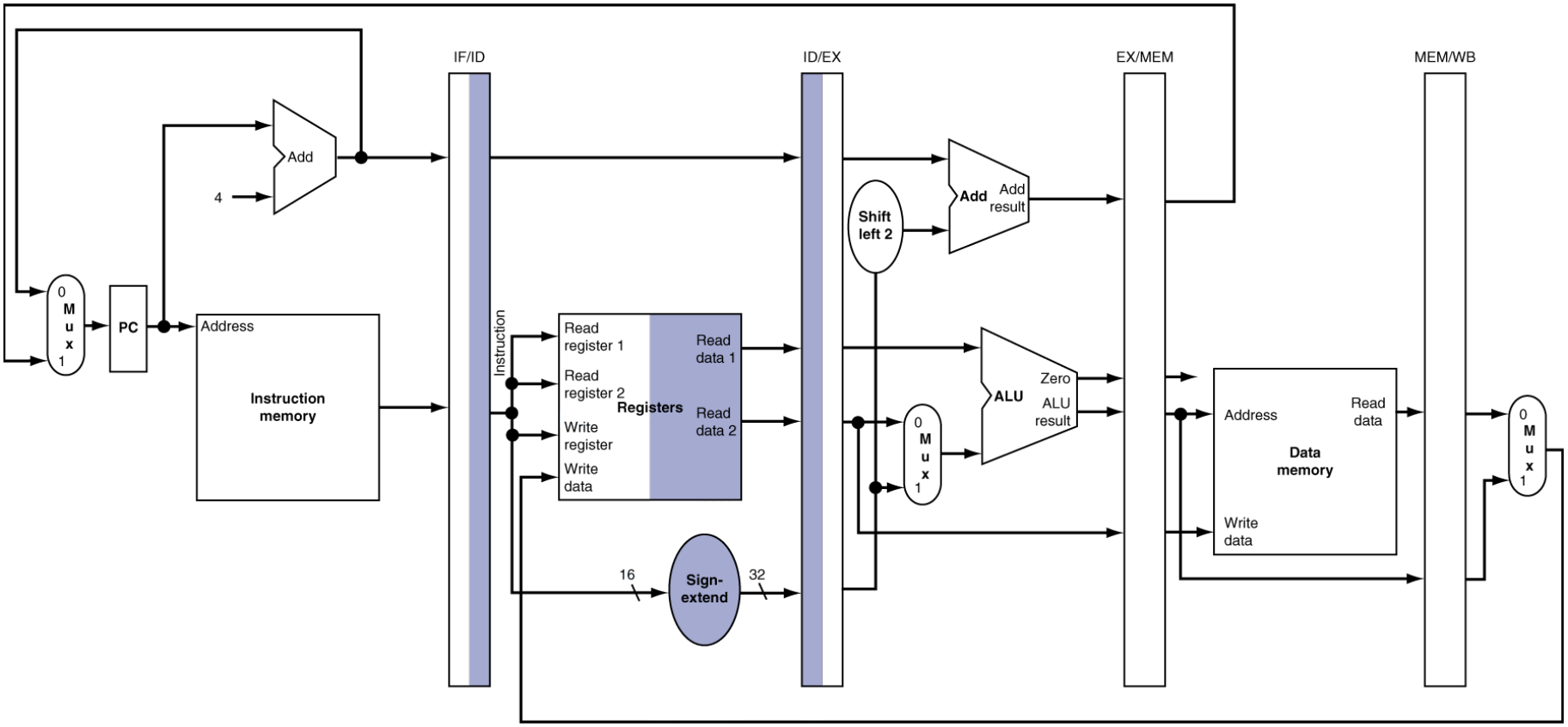


Instruction word and PC+4 are in the IF/ID pipeline register

ID for Load, Store, …



LW $4, 32($5)

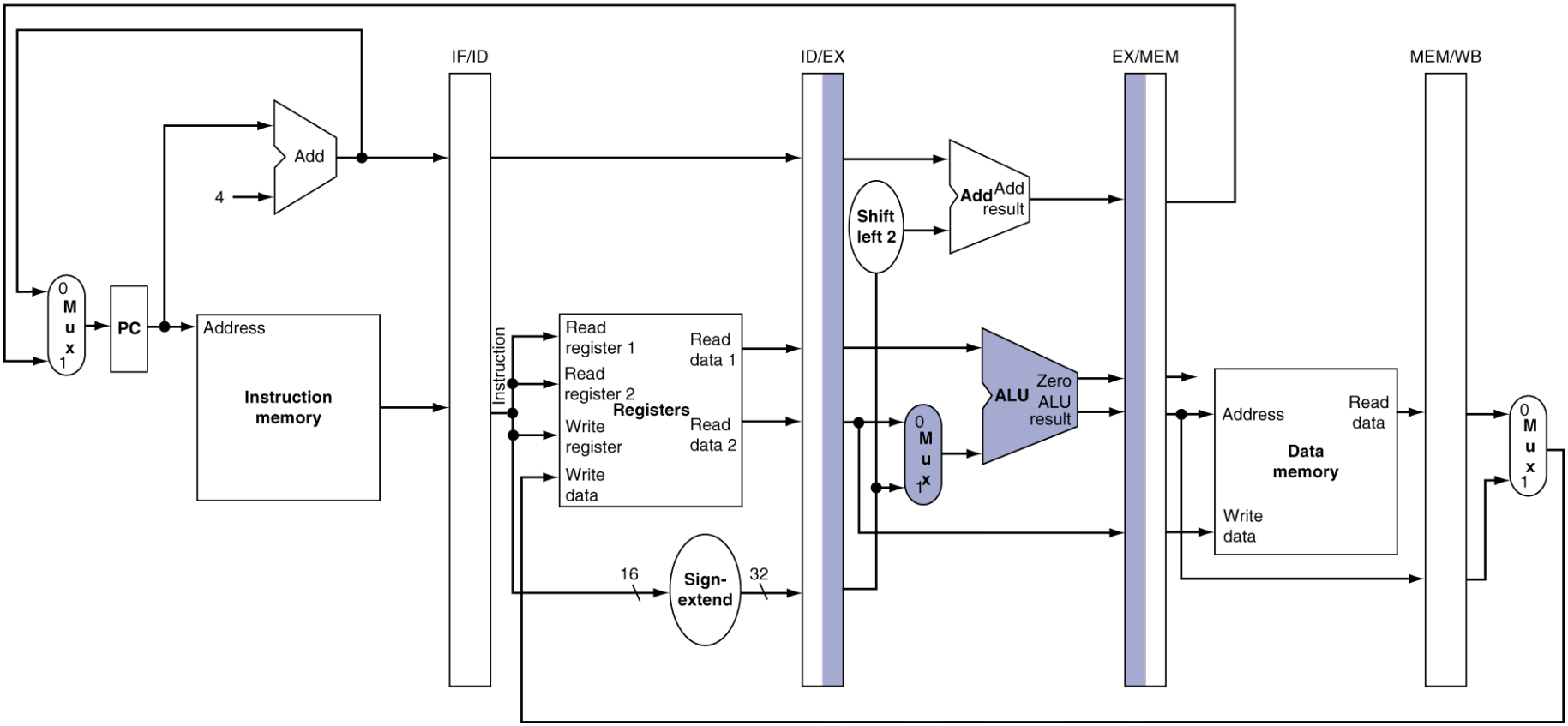


Value of $5, 32, and others are in ID|EX pipeline register

Similar info of the following instruction are now in IF|ID register

EX for Load

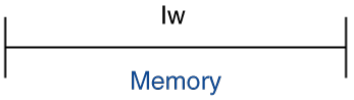
LW $4, 32($5)

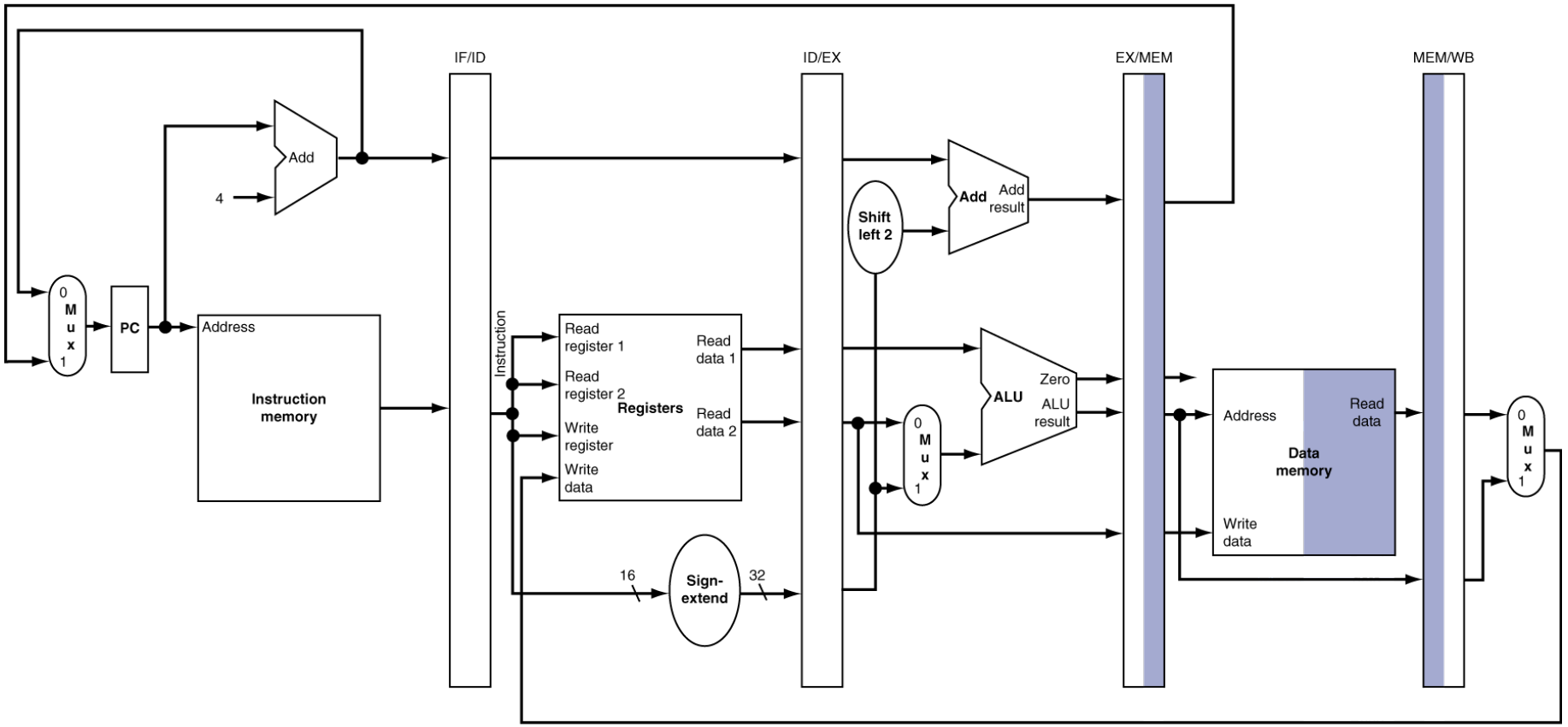


Value of $5+32, and others are in EX|MEM pipeline register

MEM for Load

LW $4, 32($5)

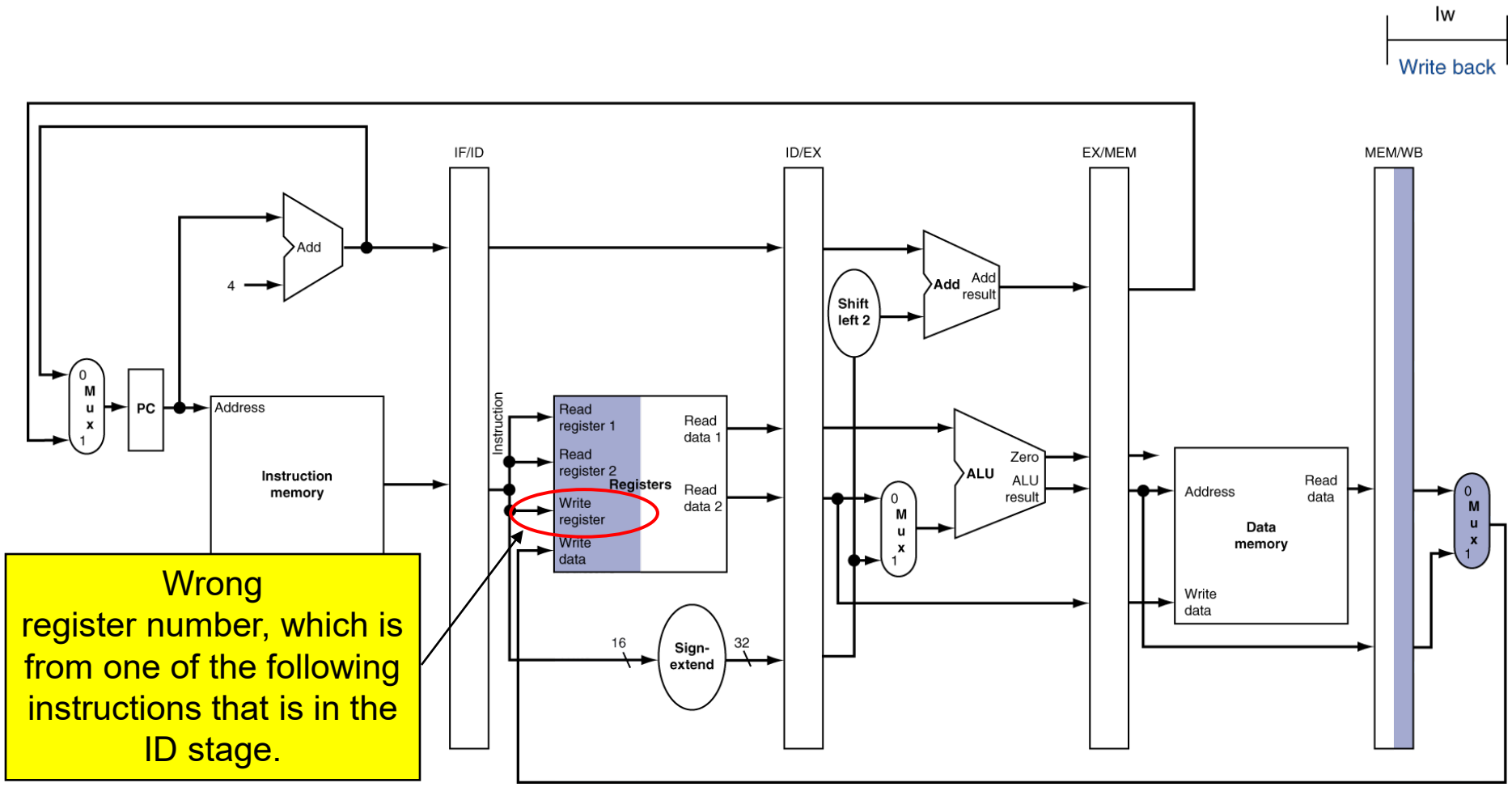




Value of MEM[$5+32], and others are in MEM|WB pipeline register for WB

WB for Load

LW $4, 32($5)

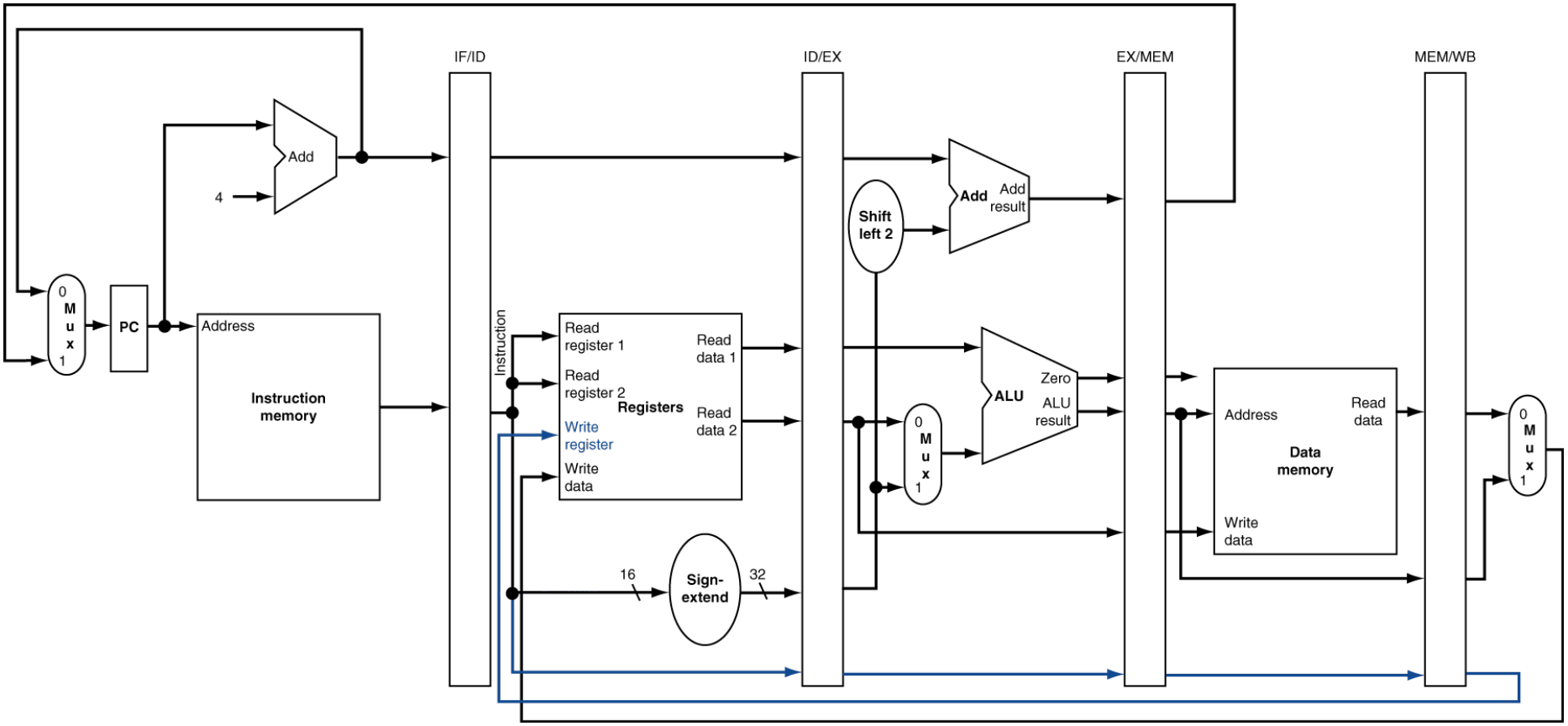


Wrong register number, which is from one of the following instructions that is in the ID stage.

Value of MEM[$5+32], and others are in MEM|WB pipeline register for WB

Corrected Datapath for Load

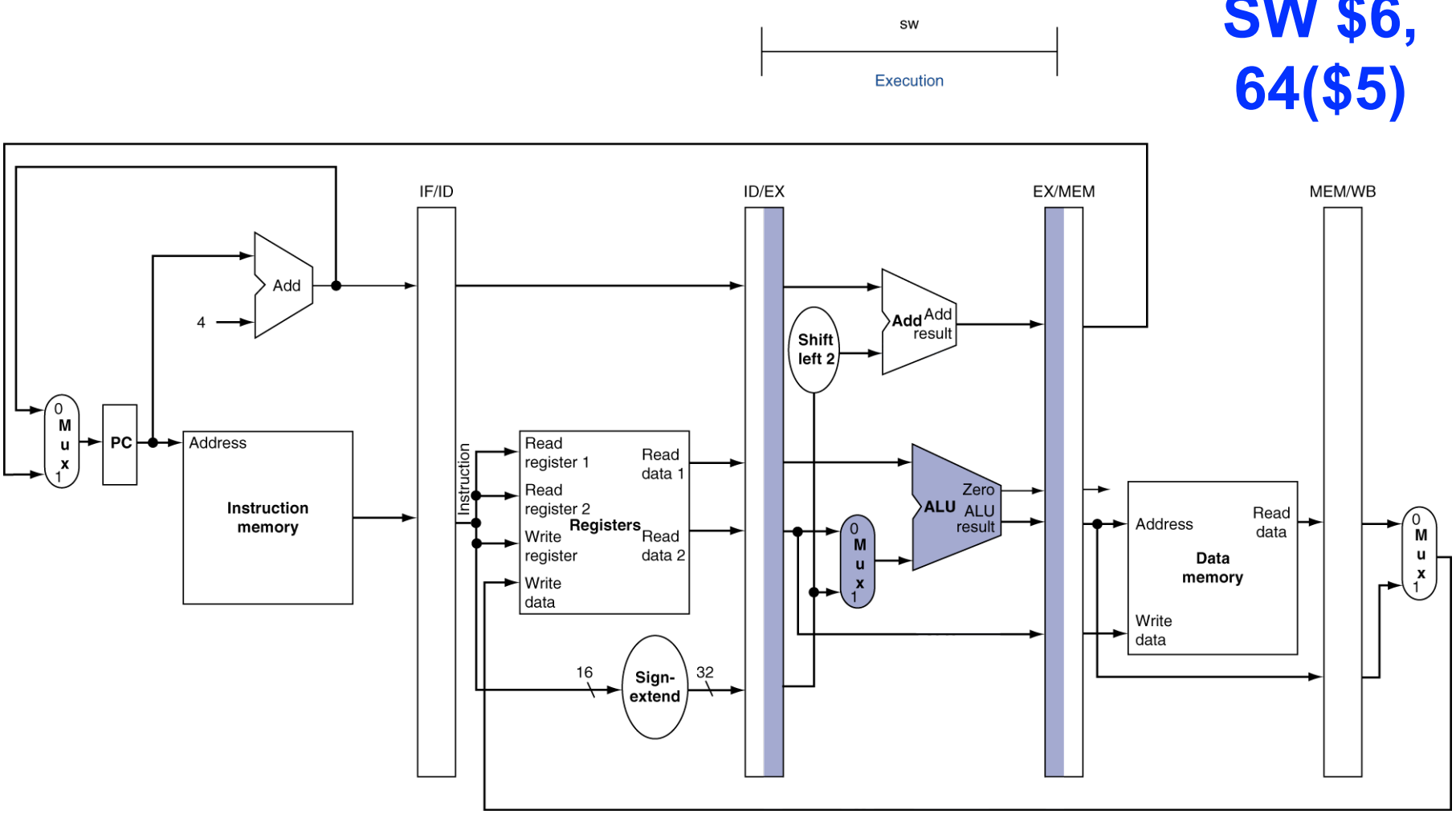
LW $4, 32($5)



LW completes and exits from the pipeline.

EX for Store

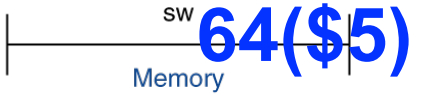
SW $6, 64($5)

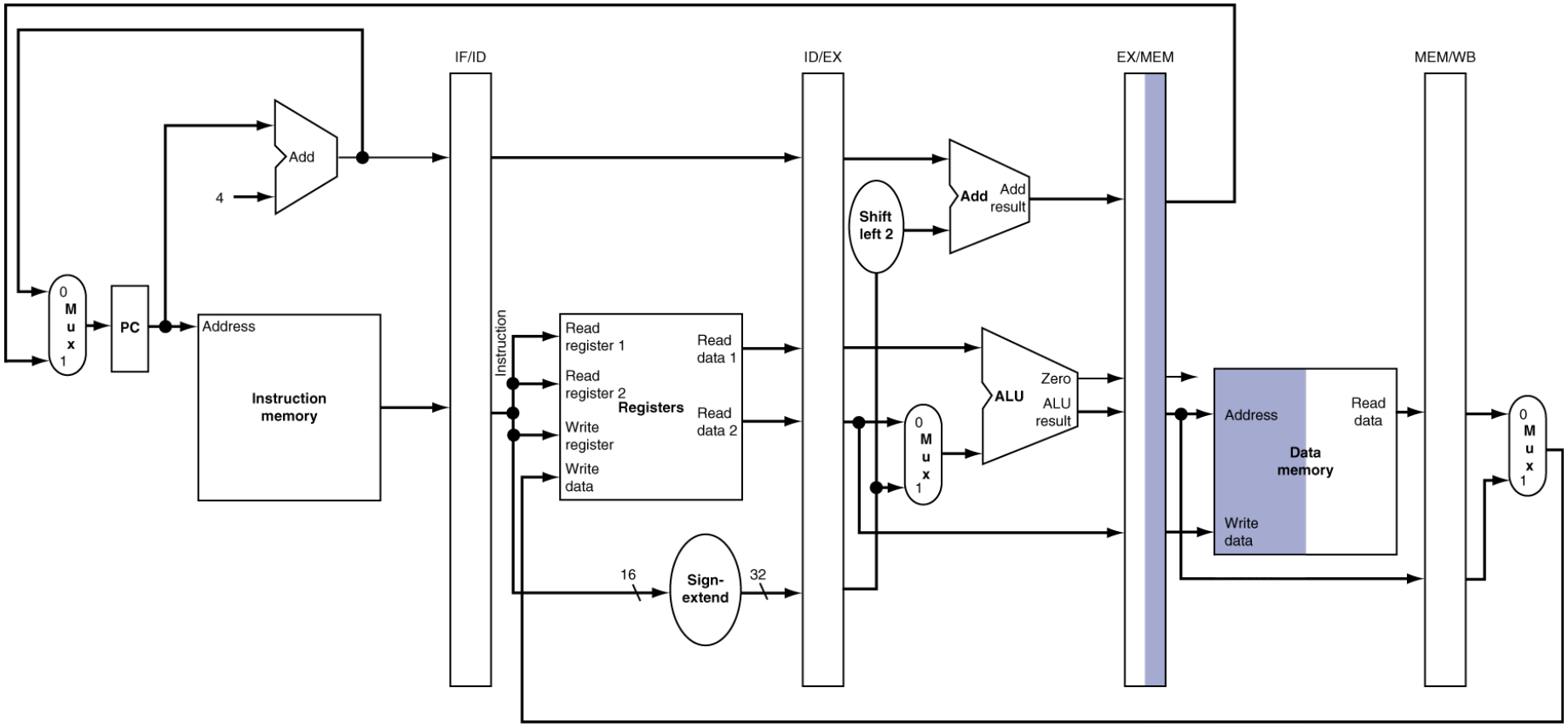


Value of $5+64 and $6 are in EX|MEM pipeline register

MEM for Store

SW $6, 64($5)

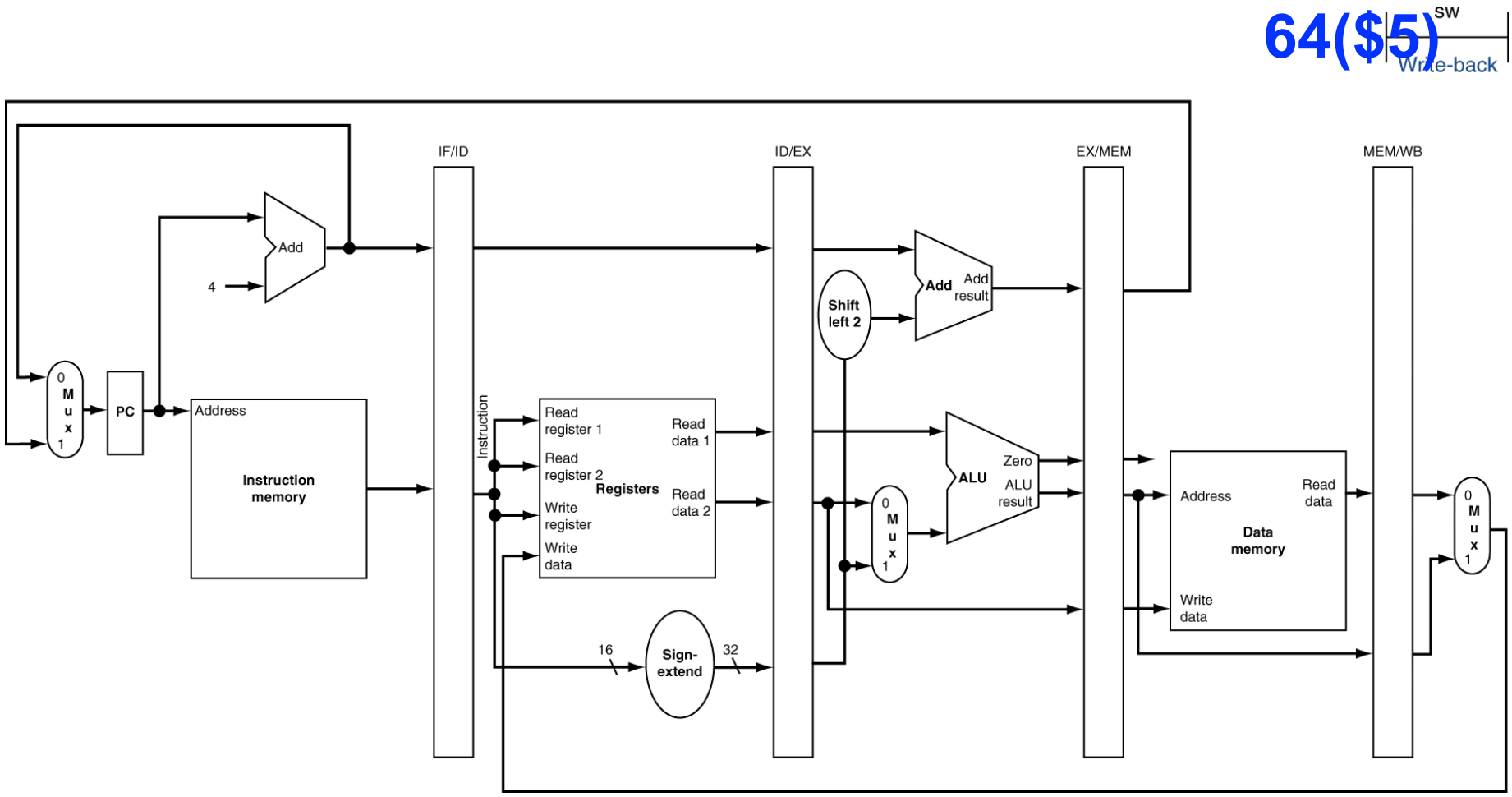




$6 is written to MEM[$5+64]

WB for Store

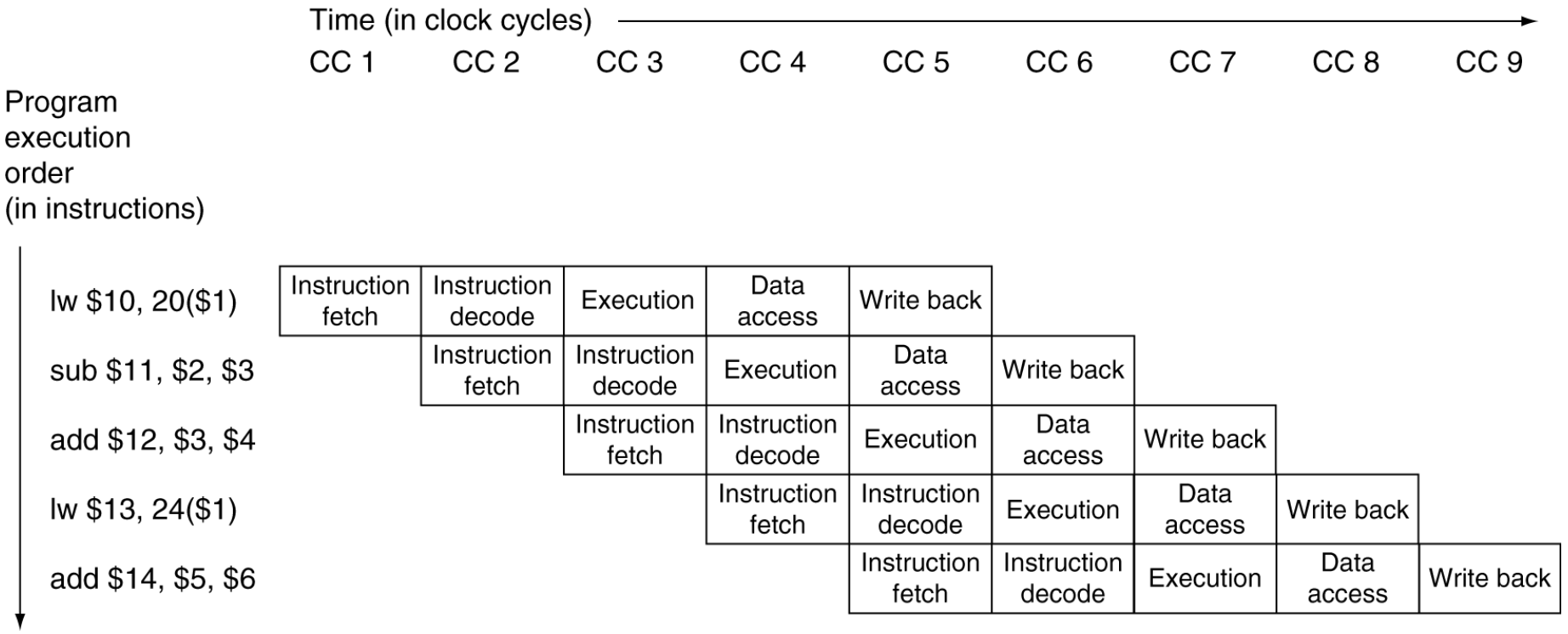
SW $6, 64($5)



Nothing to do for SW in WB stage and SW completes

Multi-Cycle Pipeline Diagram

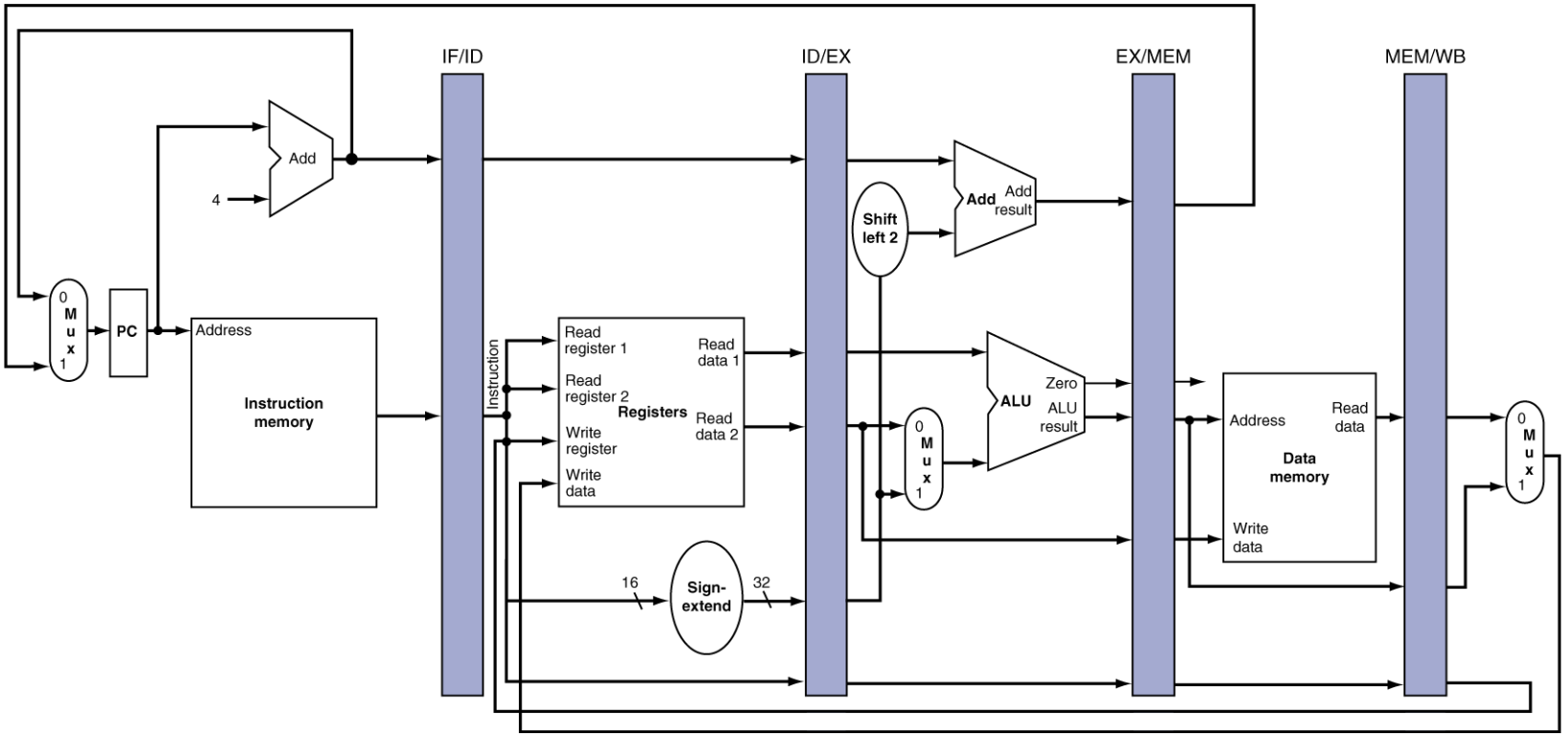
- Traditional form



Single-Cycle Pipeline Diagram

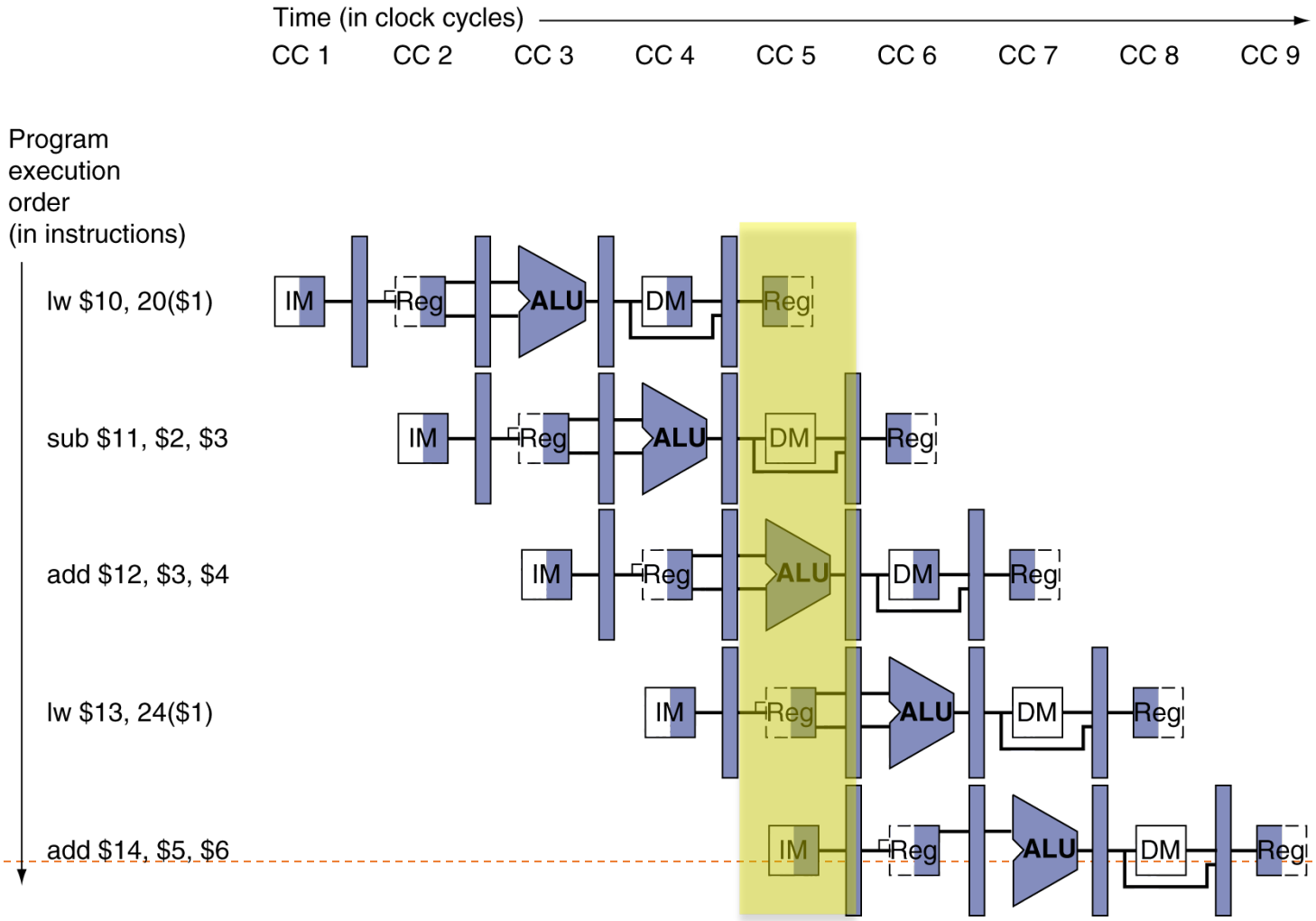
- State of pipeline in a given cycle

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

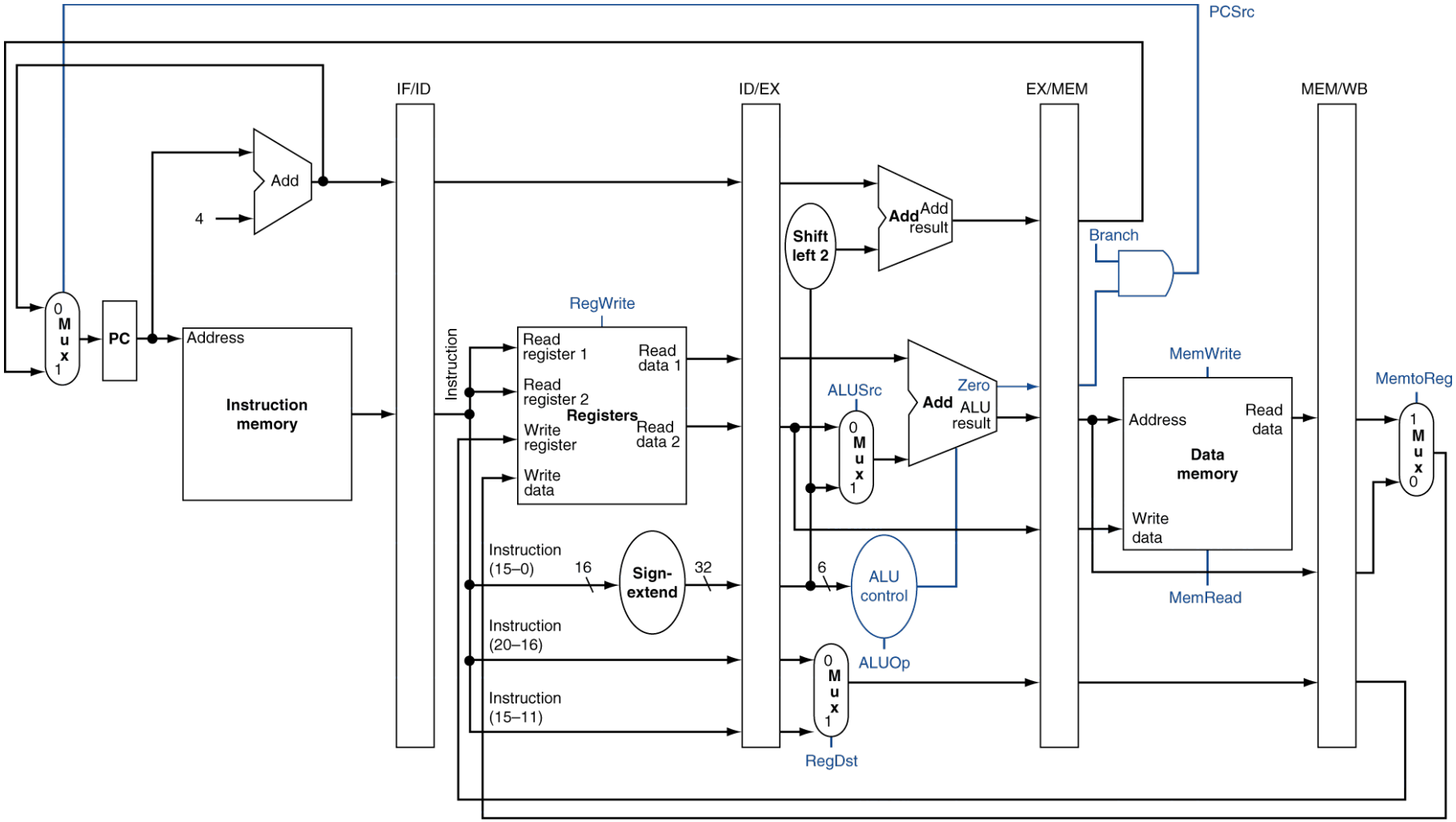


Multi-Cycle Pipeline Diagram

- Form showing resource usage

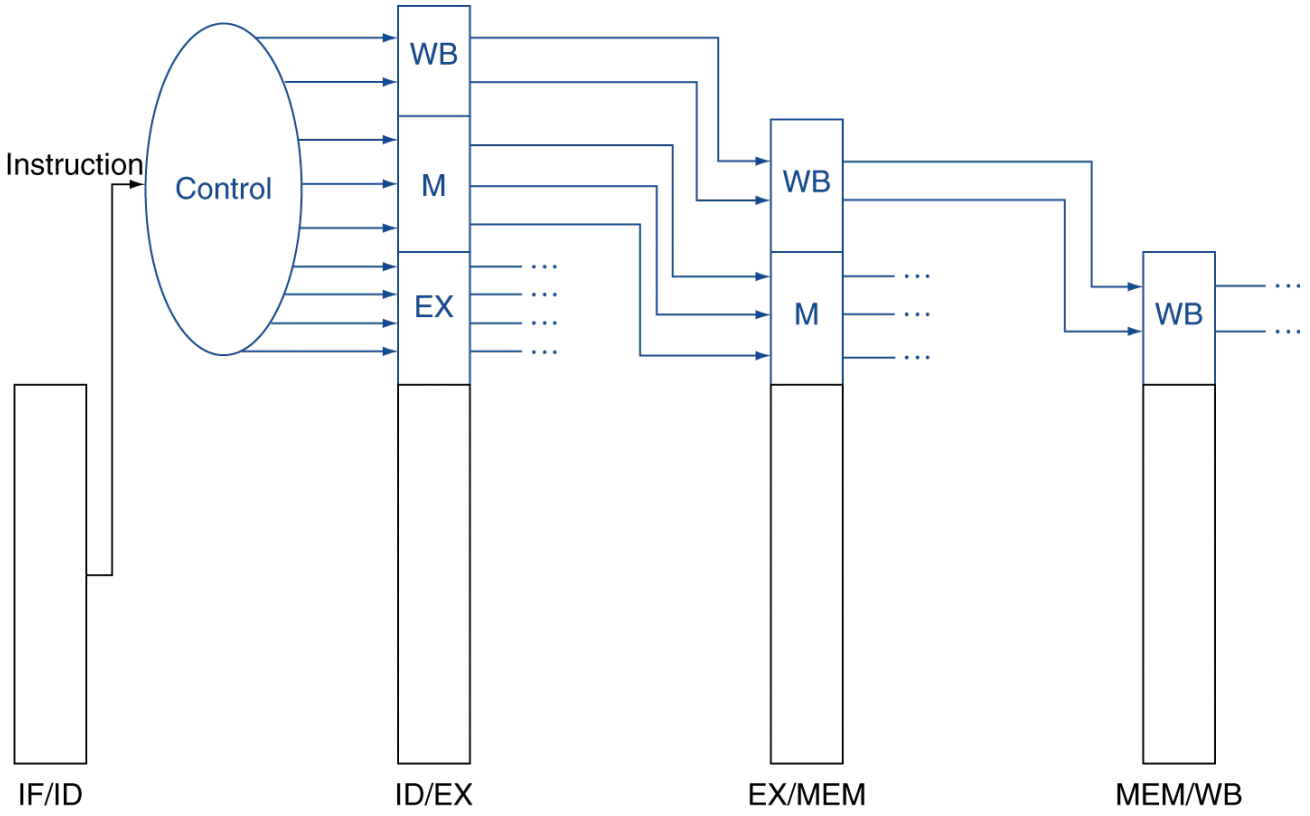


Pipelined Control (Simplified)

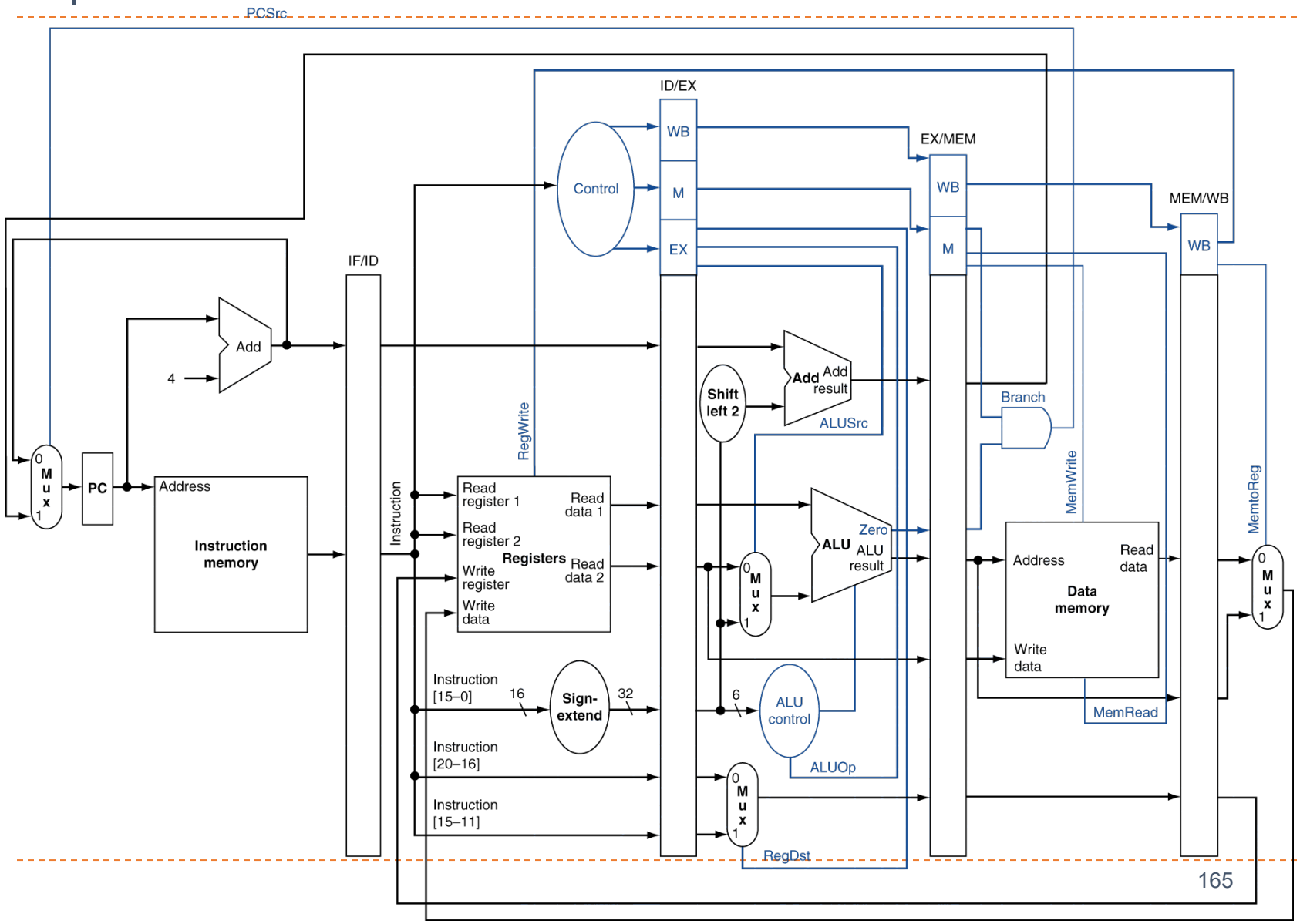


Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation



Pipelined Control



Data Hazards in ALU Instructions

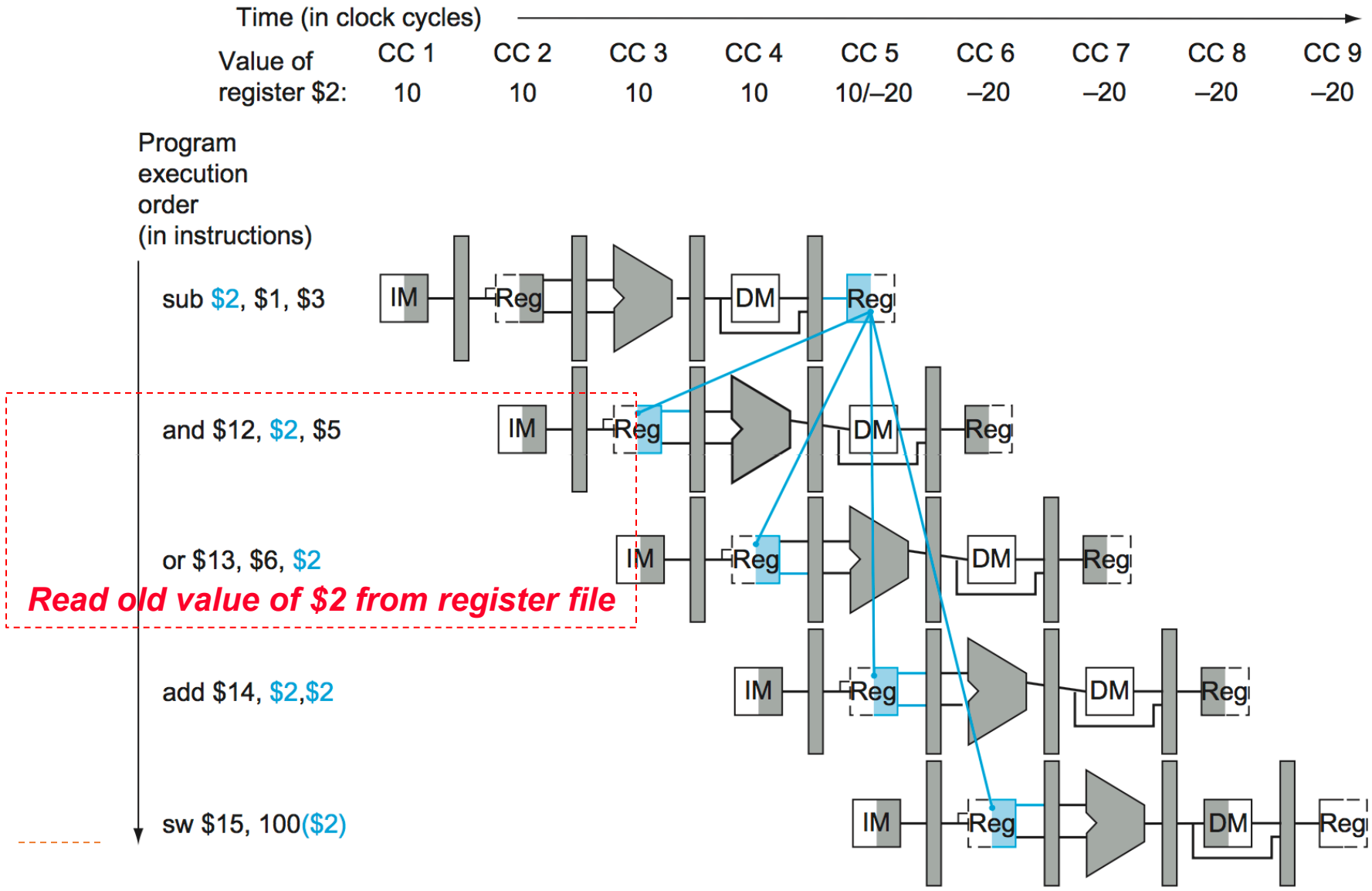
- Consider this sequence:

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

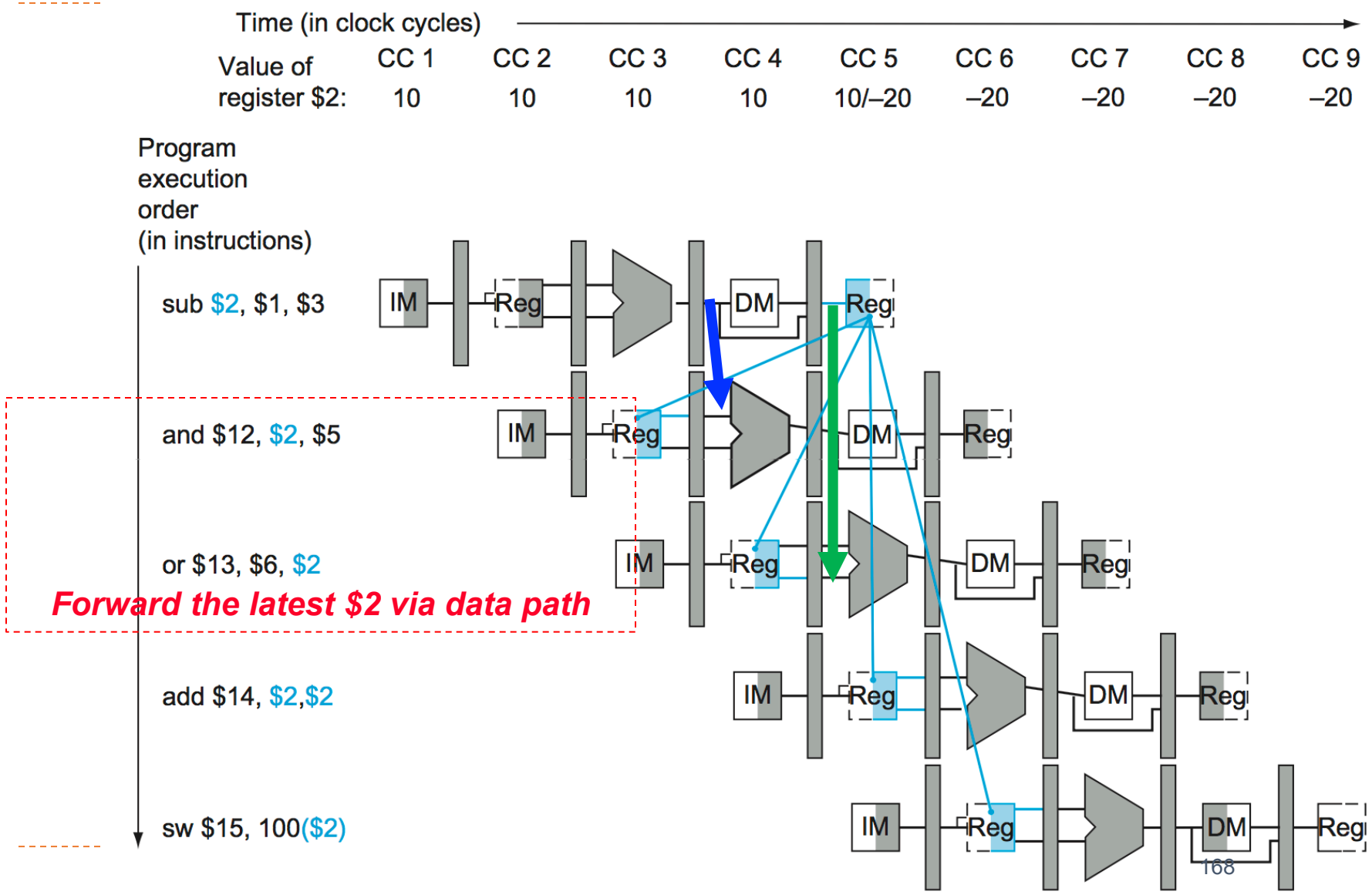
- Called Read After Write (RAW) hazards

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

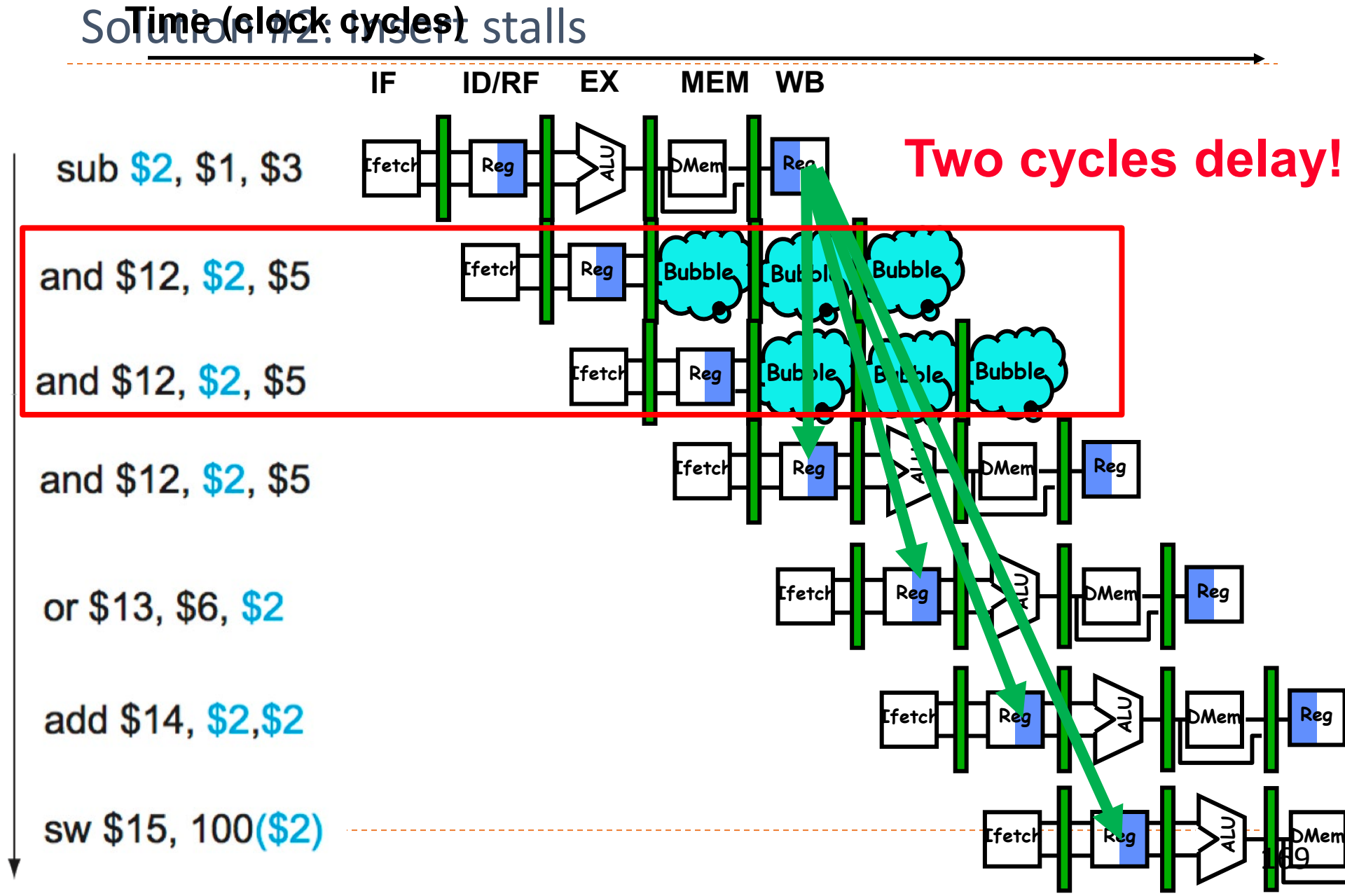
Data Dependency → Data Hazards



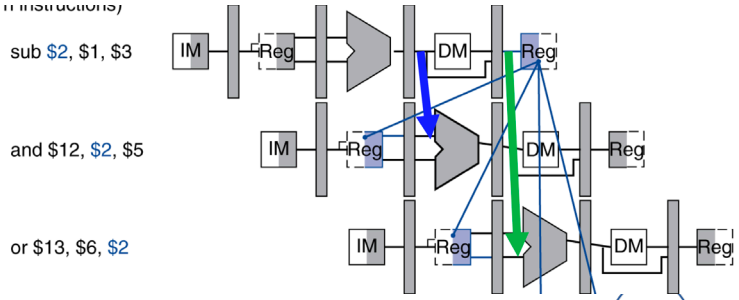
Solution #1: Handling RAW Hazards by Forwarding



Solution #2: Insert stalls



Datapath for Forwarding

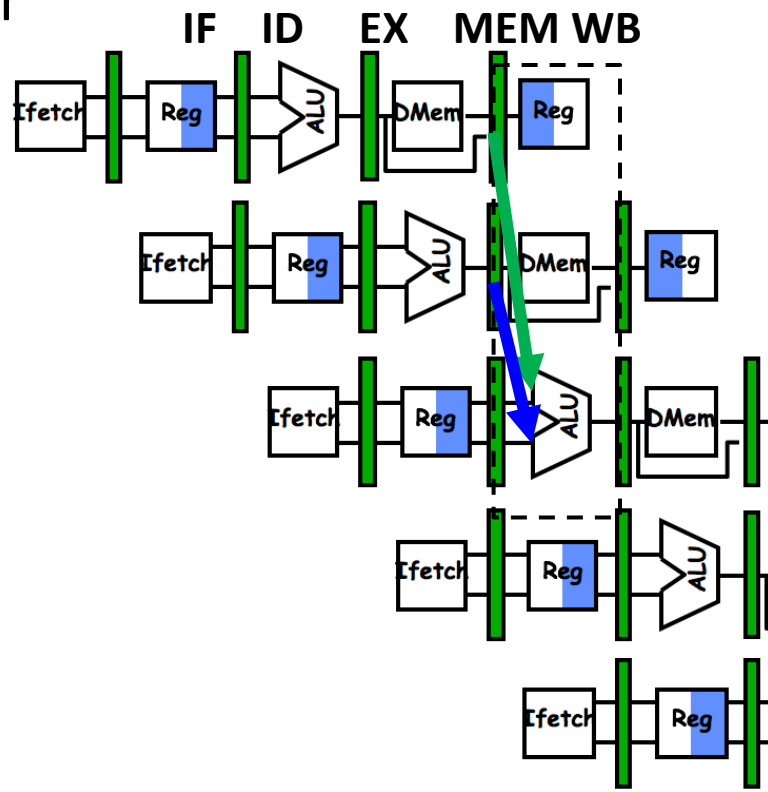


- Forwarding happens in two consecutive cycles

Detecting RAW Hazards

- Current instruction being executed in ID/EX register
- Previous instruction is in the EX/MEM register
- 2nd Previous is in the MEM/WB register

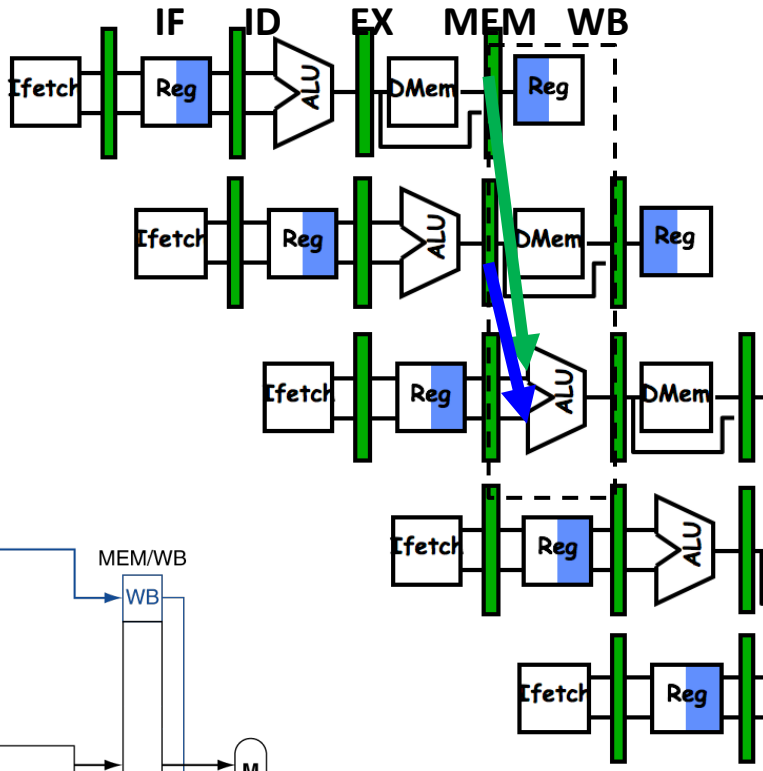
```
ADD R1, R2, R3  #2nd Previous in MEM/WB
SUB R6, R4, R5  #Previous in EX/MEM
AND R7, R1, R6  #Current in ID/EX
```

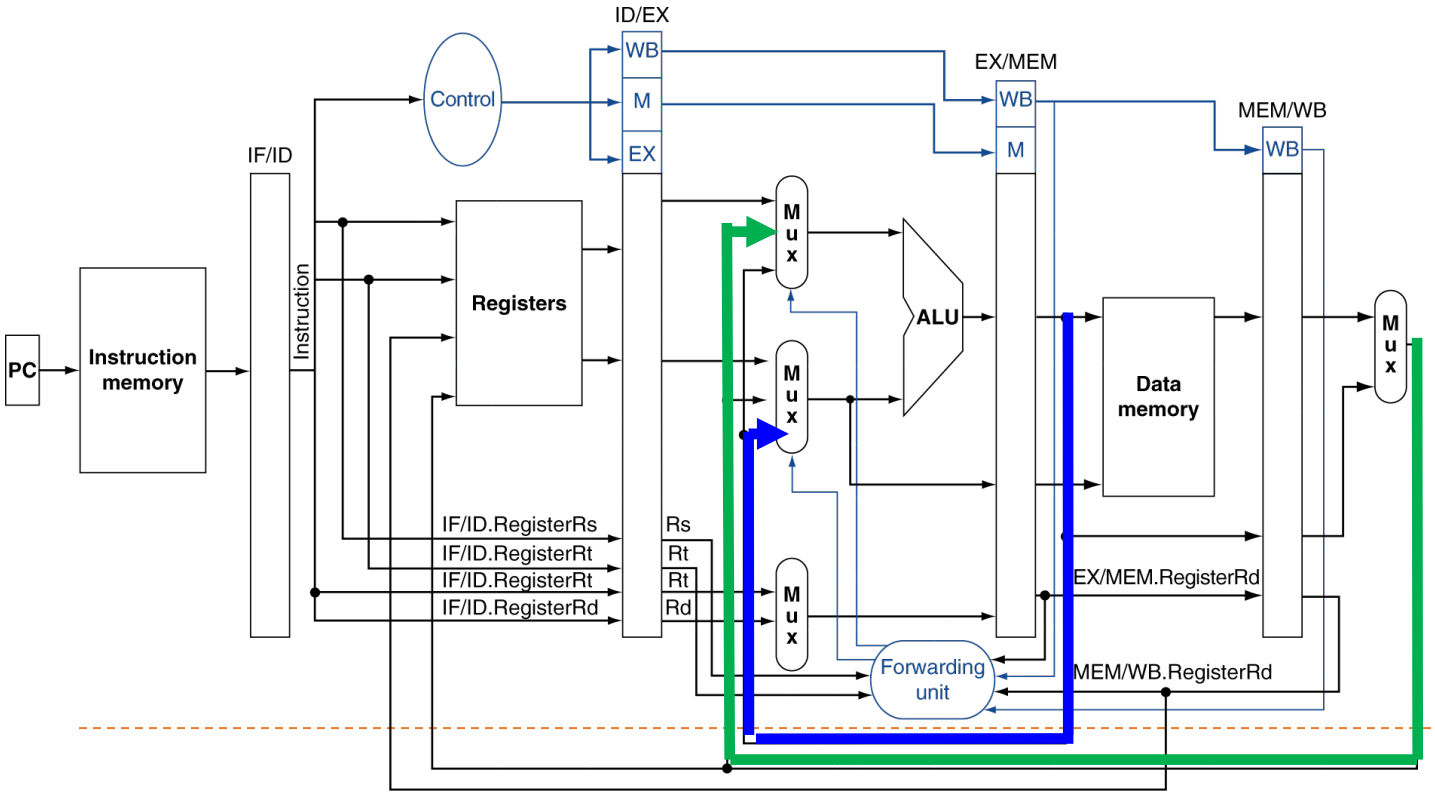


- Forwarding happens in the same cycle

Detecting RAW Hazards

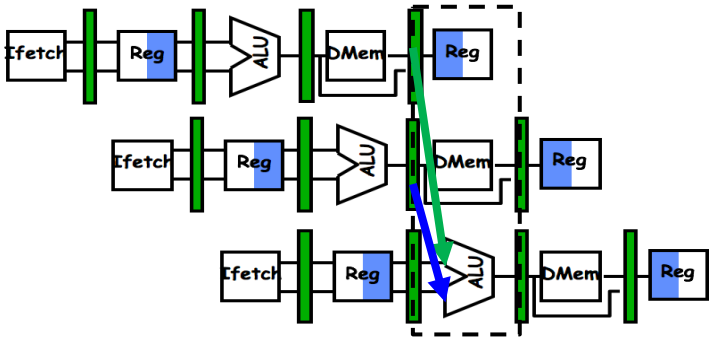
```
ADD R1, R2, R3  #2nd Previous in MEM/WB
SUB R6, R4, R5  #Previous in EX/MEM
AND R7, R1, R6  #Current in ID/EX
```





Detecting RAW Hazards

```
ADD R1, R2, R3  #2nd Previous in MEM/WB
  SUB R6, R4, R5  #Previous in EX/MEM
    AND R7, R1, R6  #Current in ID/EX
```



- Pass register numbers along pipeline
  - ID/EX.RegisterRs = register number for Rs in ID/EX (Rs1)
  - ID/EX.RegisterRt = register number for Rt in ID/EX (Rs2)
  - ID/EX.RegisterRd = register number for Rd in ID/EX

- RAW 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**

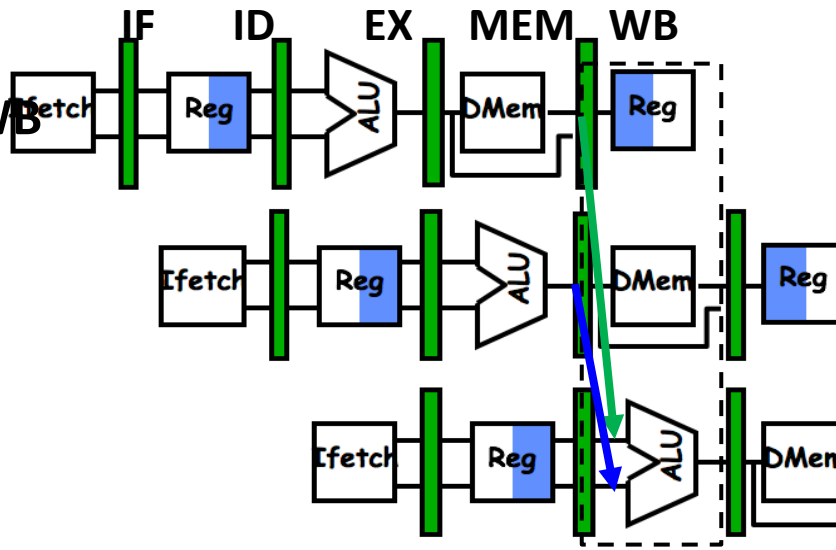
Fwd from EX/MEM pipeline reg

Fwd from MEM/WB pipeline reg

Detecting the Need to Forward

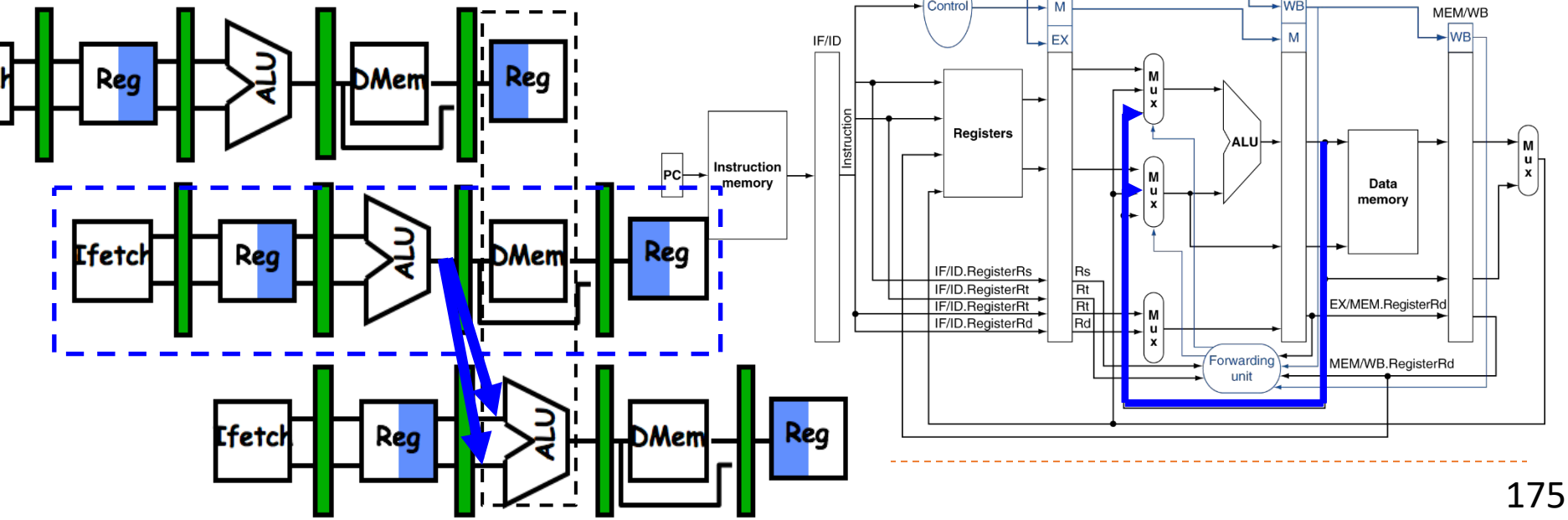
- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite, MEM/WB.RegWrite
- And only if Rd for that instruction is not R0
  - EX/MEM.RegisterRd ≠ 0
  - MEM/WB.RegisterRd ≠ 0

```
ADD R1, R2, R3  #2nd Previous in MEM/WB
SUB R6, R4, R5  #Previous in EX/MEM
AND R7, R1, R6  #Current in ID/EX
```



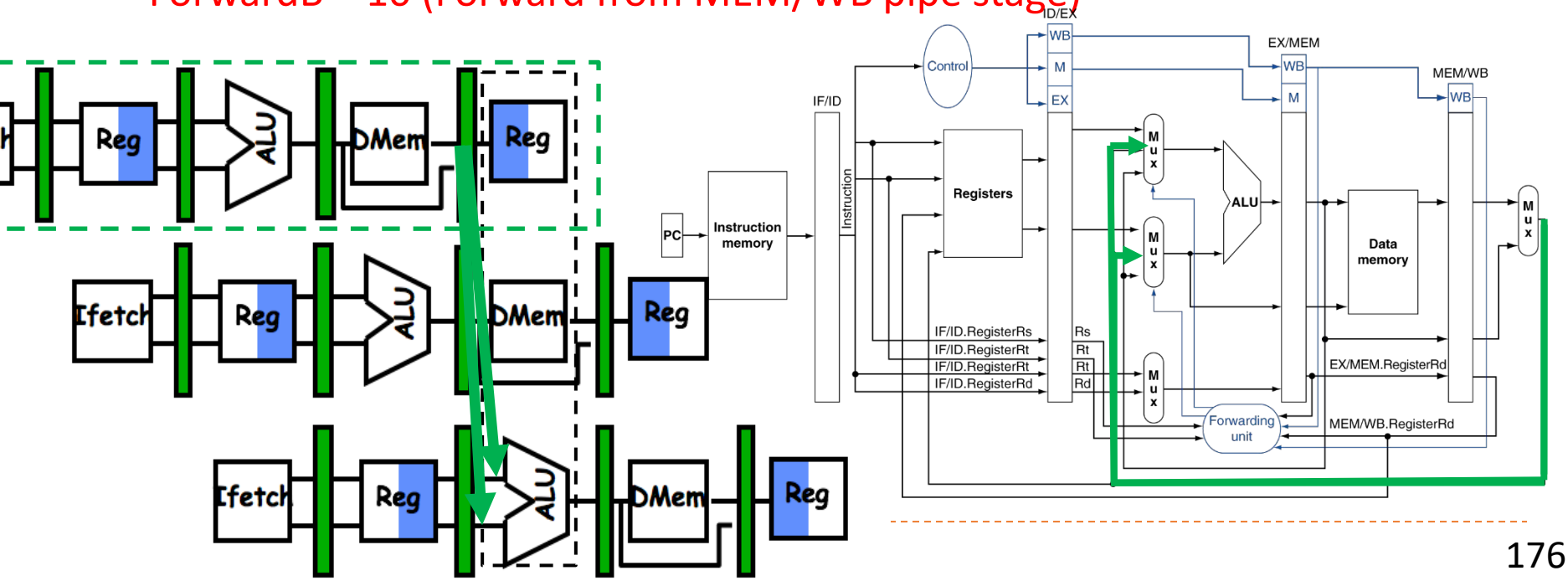
Forwarding Conditions

- Detecting RAW hazard with Previous Instruction
  - **if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)**
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01 (Forward from EX/MEM pipe stage)
  - **if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)**
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01 (Forward from EX/MEM pipe stage)



Forwarding Conditions

- Detecting RAW hazard with Second Previous
  - **if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)**
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 10 (Forward from MEM/WB pipe stage)
  - **if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)**
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 10 (Forward from MEM/WB pipe stage)

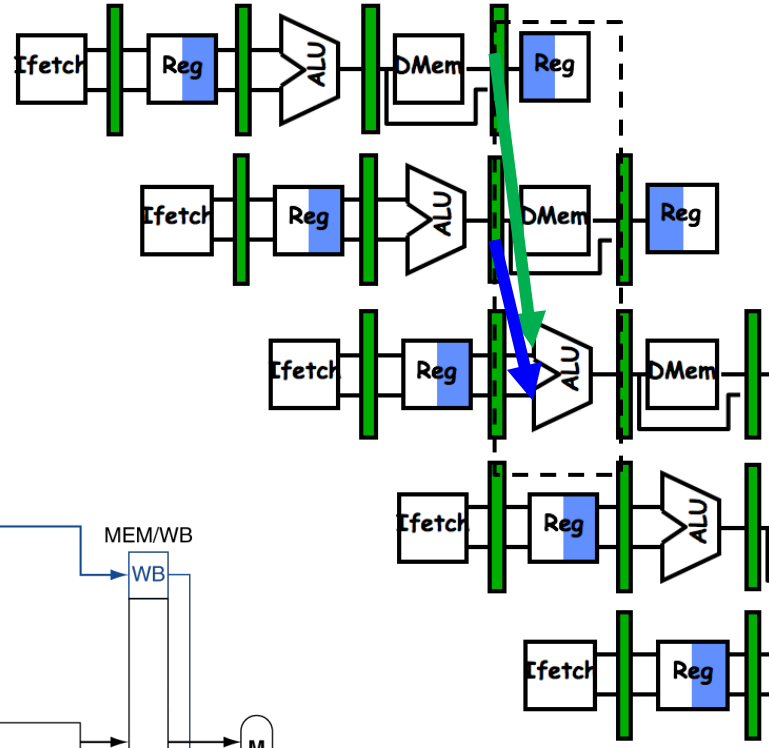


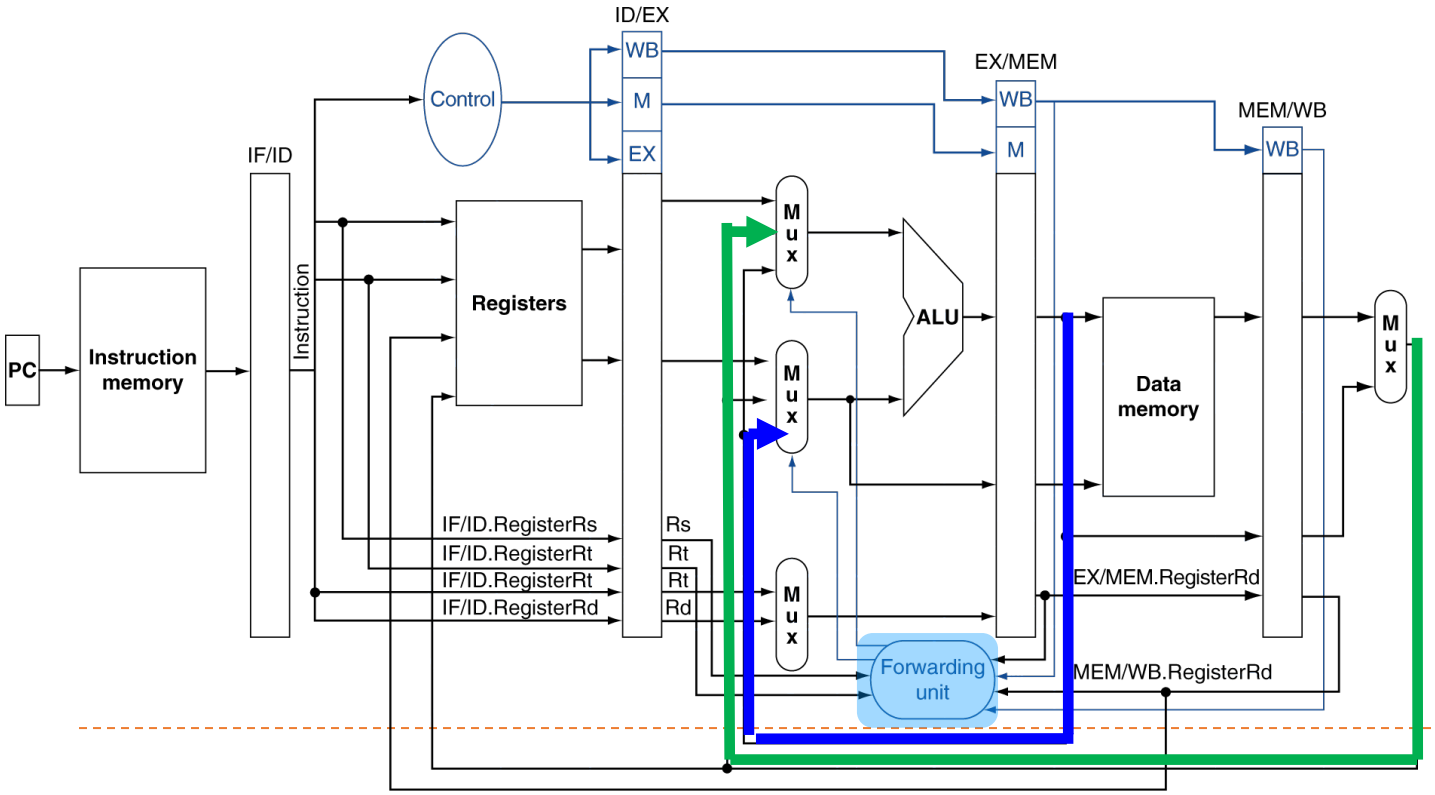
Control Signals During Forwarding:
Those Light Blue lines

ADD R1, R2, R3 #2nd Previous in MEM/WB

SUB R6, R4, R5 #Previous in EX/MEM

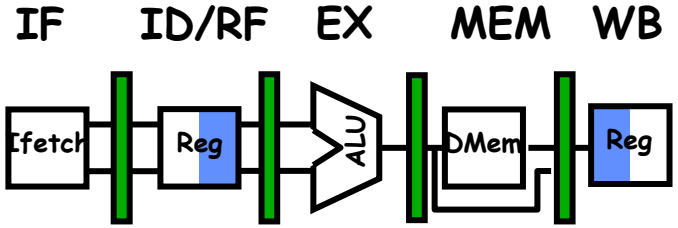
AND R7, R1, R6 #Current in ID/EX





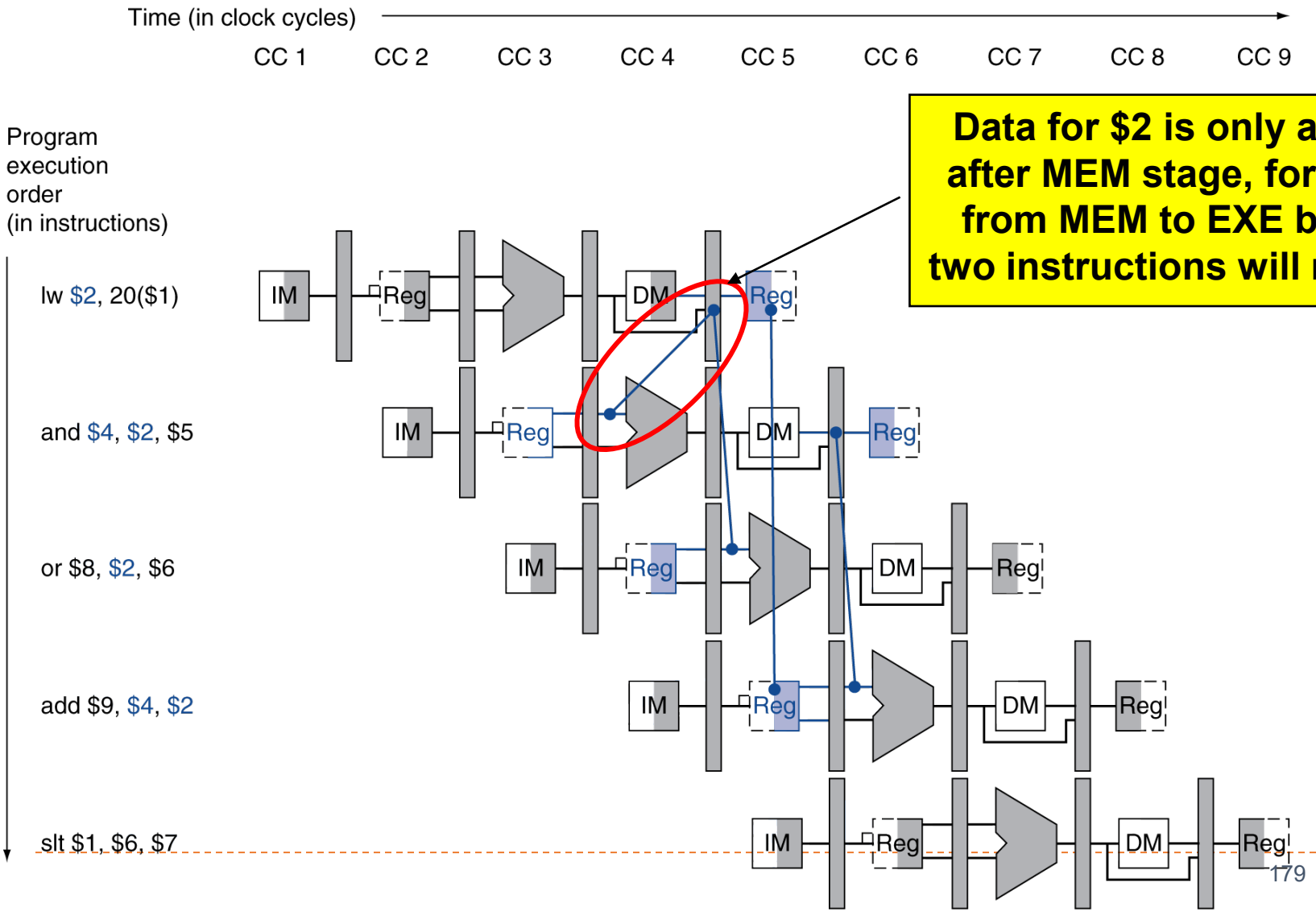
RAW Hazards with Load/Store

- **LW Rt, 20(Rs): load a word from memory @ [Rs]+20 into Rt**
  - ID/RF: Read register Rs: [Rs] (rs select)
  - EX: Calculate effective address: [Rs] + 20
  - **MEM: Memory read from [Rs]+20**
    - **Data is available in MEM|WB**
    - **Unlike ALU: data is available in EX|MEM**
  - WB: data write back to Rt (rt select)

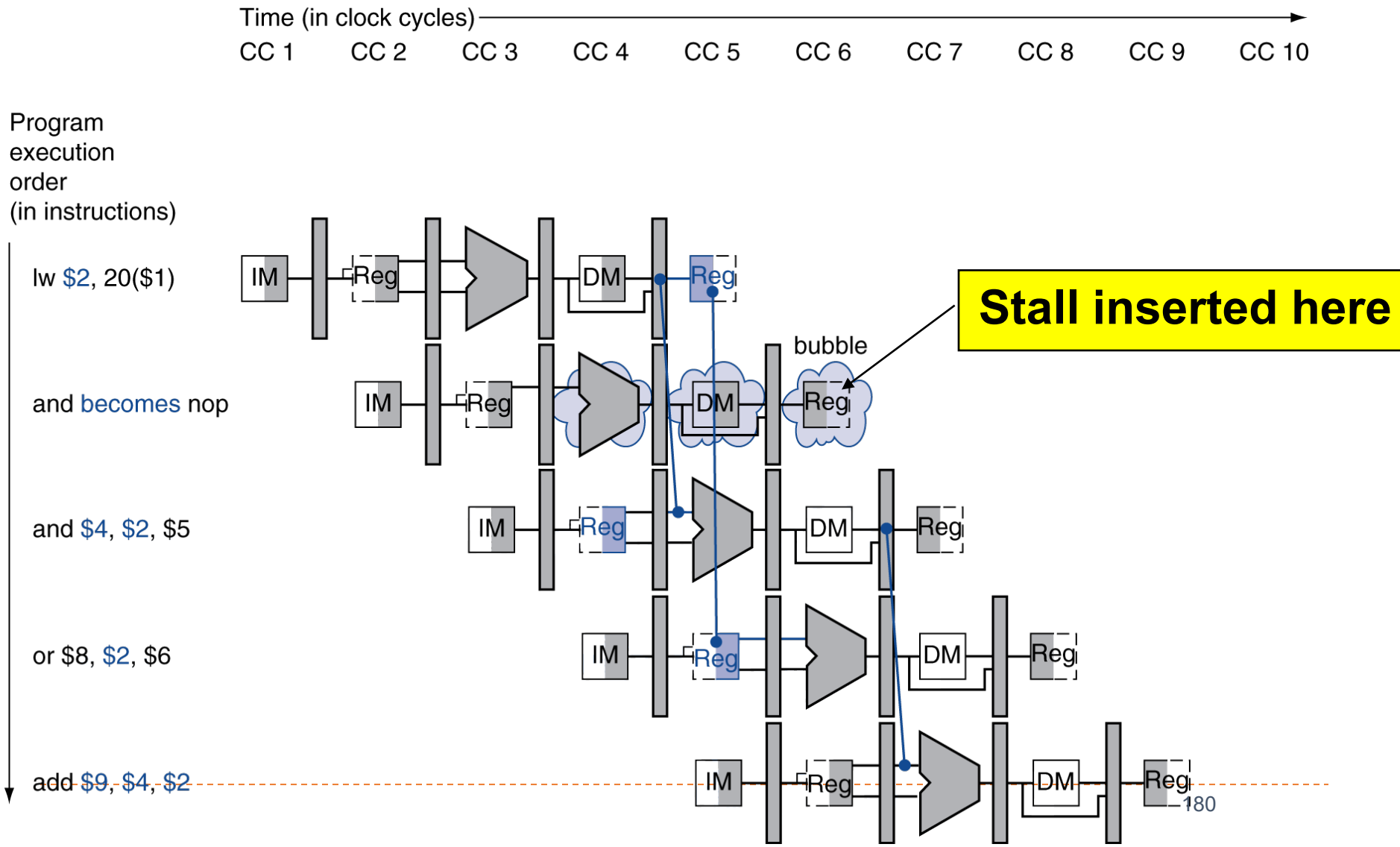


- **SW Rt,12(Rs): store a word in Rt in the memory @ [Rs]+12**
  - ID/RF: Read register Rs and Rt (rs and rt select, no rd)
    - Rs is needed in EX, and **Rt is needed in MEM**
  - EX: Calculate effective address: [Rs] + 12
  - **MEM: Memory write to [Rs]+12**
    - **Need Rt to be available**
    - **Unlike ALU, data needs to be available in ID|EX**
  - No need WB

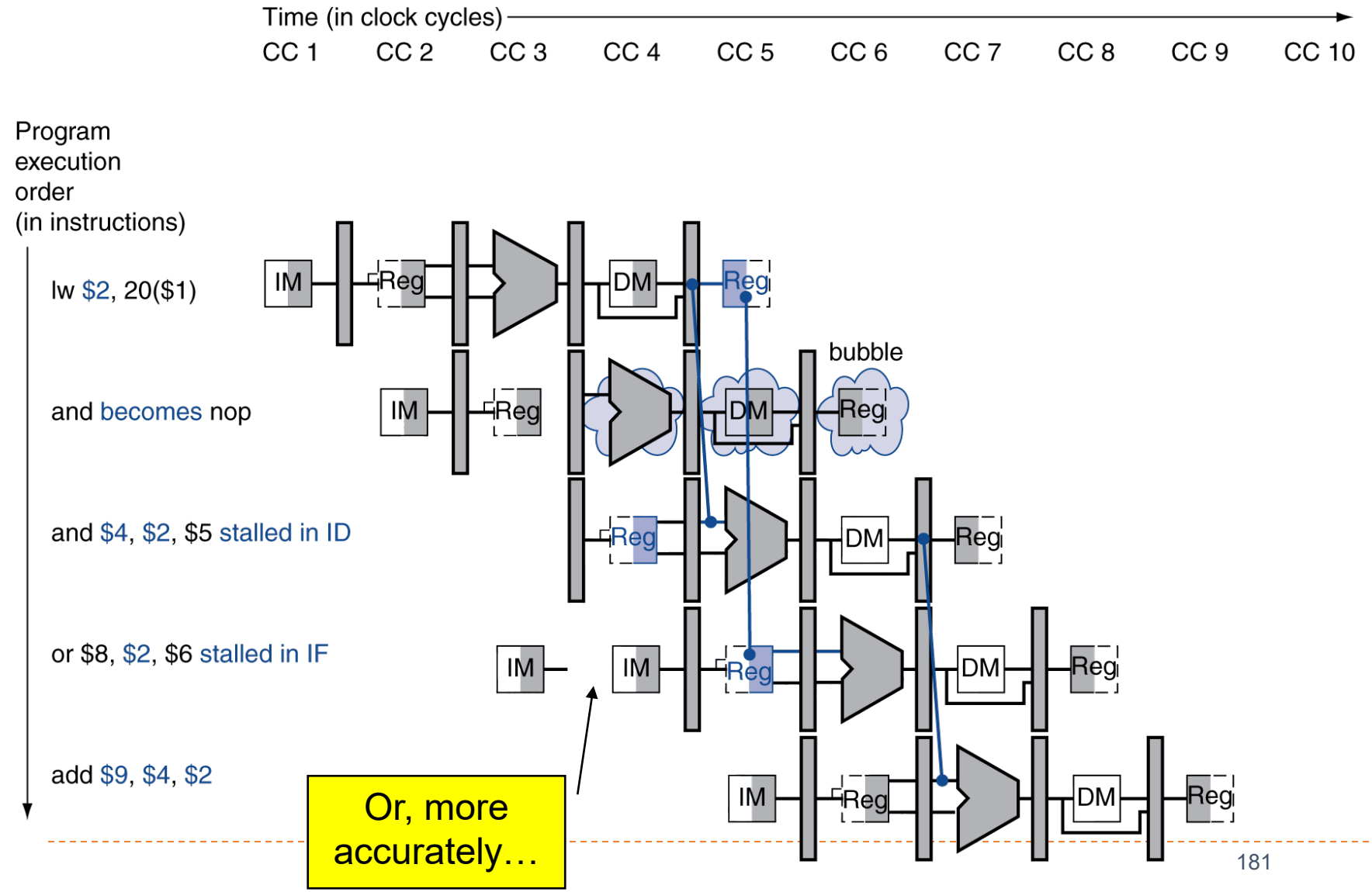
Load-Use RAW Data Hazard



Stall/Bubble in the Pipeline

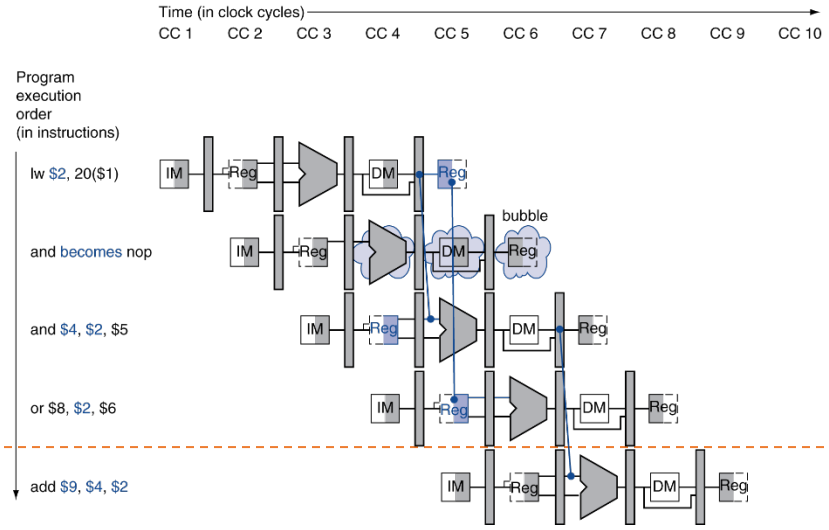


Stall/Bubble in the Pipeline



Load-Use Hazard Detection

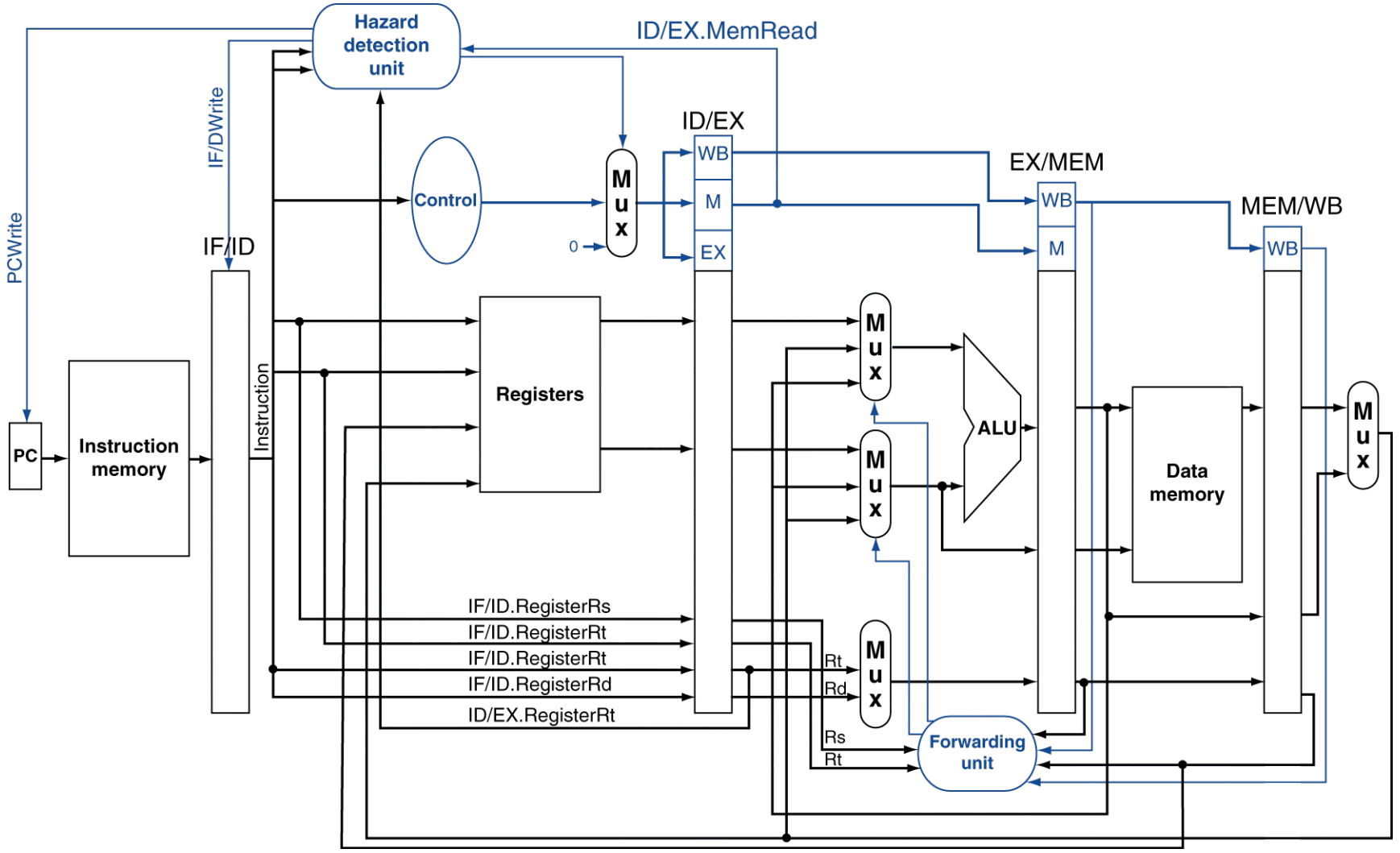
- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs, IF/ID.RegisterRt
- Load-use hazard when
  - ID/EX.MemRead and
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
    (ID/EX.RegisterRt = IF/ID.RegisterRt))
- If detected, stall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

Datapath with Hazard Detection



Compiler Scheduling for Removing Load-Use Stall

- Compilers can schedule code in a way to avoid load → ALU-use stalls

  a = b + c; d = e – f;

- Slow code: 2 stall cycles

```
lw    r10, (r1)          # r1 = addr b
lw    r11, (r2)          # r2 = addr c
                         # stall
add   r12, r10, r11      # b + c
sw    r12, (r3)          # r3 = addr a
lw    r13, (r4)          # r4 = addr e
lw    r14, (r5)          # r5 = addr f
                         # stall
sub   r15, r13, r14      # e - f
sw    r15, (r6)          # r6 = addr d
```

Fast code: No Stalls

```
lw    r10, 0(r1)
lw    r11, 0(r2)
lw    r13, 0(r4)
lw    r14, 0(r5)
add   r12, r10, r11
sw    r12, 0(r3)
sub   r15, r13, r14
sw    r15, 0(r6)
```

Stalls and Performance

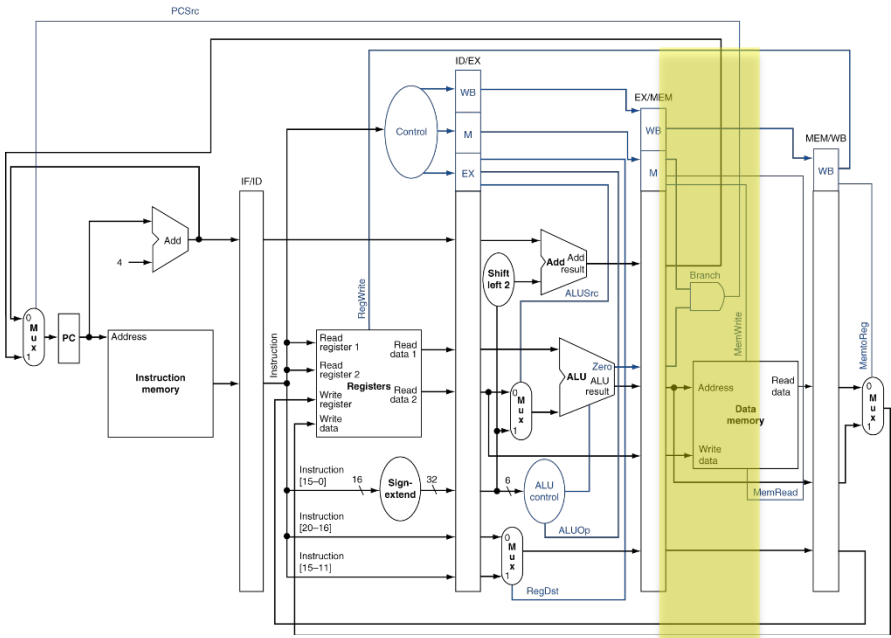
The BIG Picture

- 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

Control Hazards Because of Branches

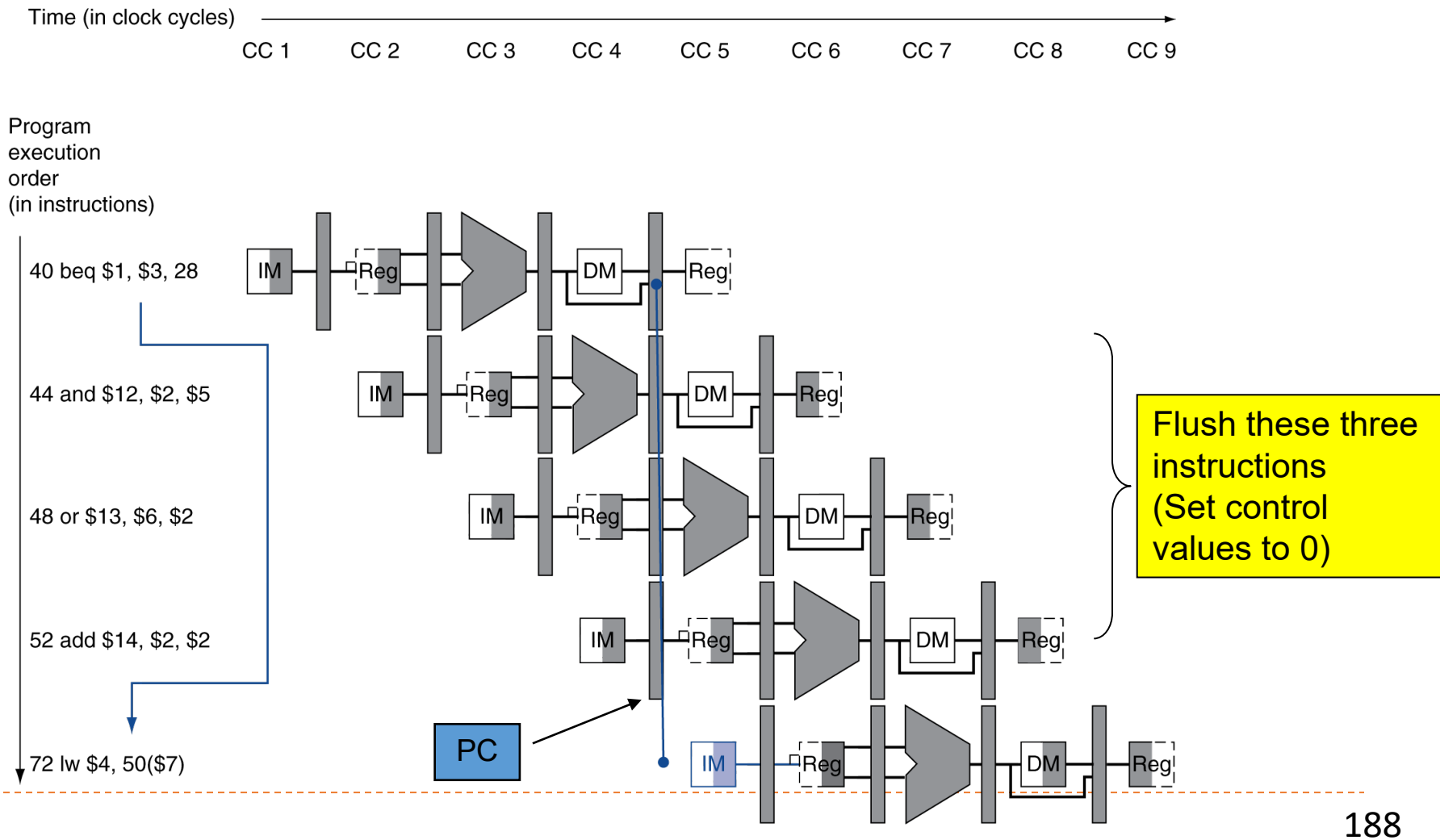
- Branch outcome determined in MEM

```
40      beq   $1,    $3, 28
44      and   $12, $2, $5
48            or    $13, $6,  $2
52            add   $14, $2, $2
…
72            lw    $4,  50($7)
```



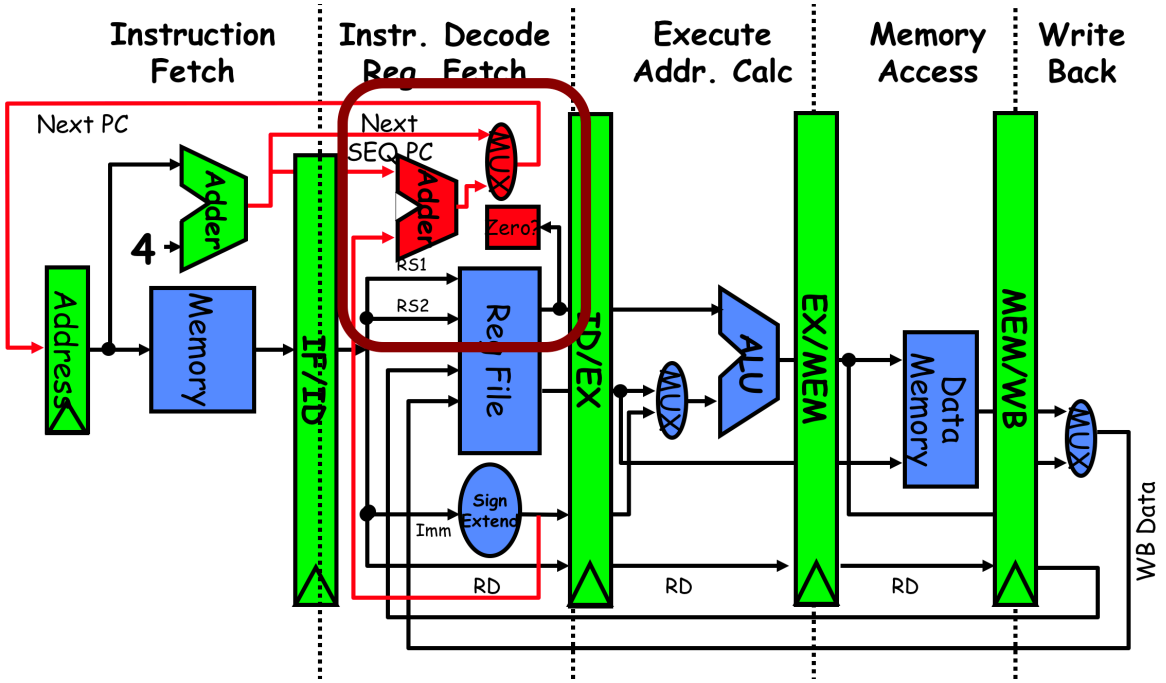
Control Hazards

- Branch outcome determined in MEM



Reducing Branch Delay

- In general, branch could cause 3 cycle delay
  - Since branch outcome is determined at MEM stage
- Move hardware to determine outcome at ID stage → 1 cycle delay
  - BEQZ instruction



- For BEQ: add target address adder and Register comparator in the ID stage

Reducing Branch Delay

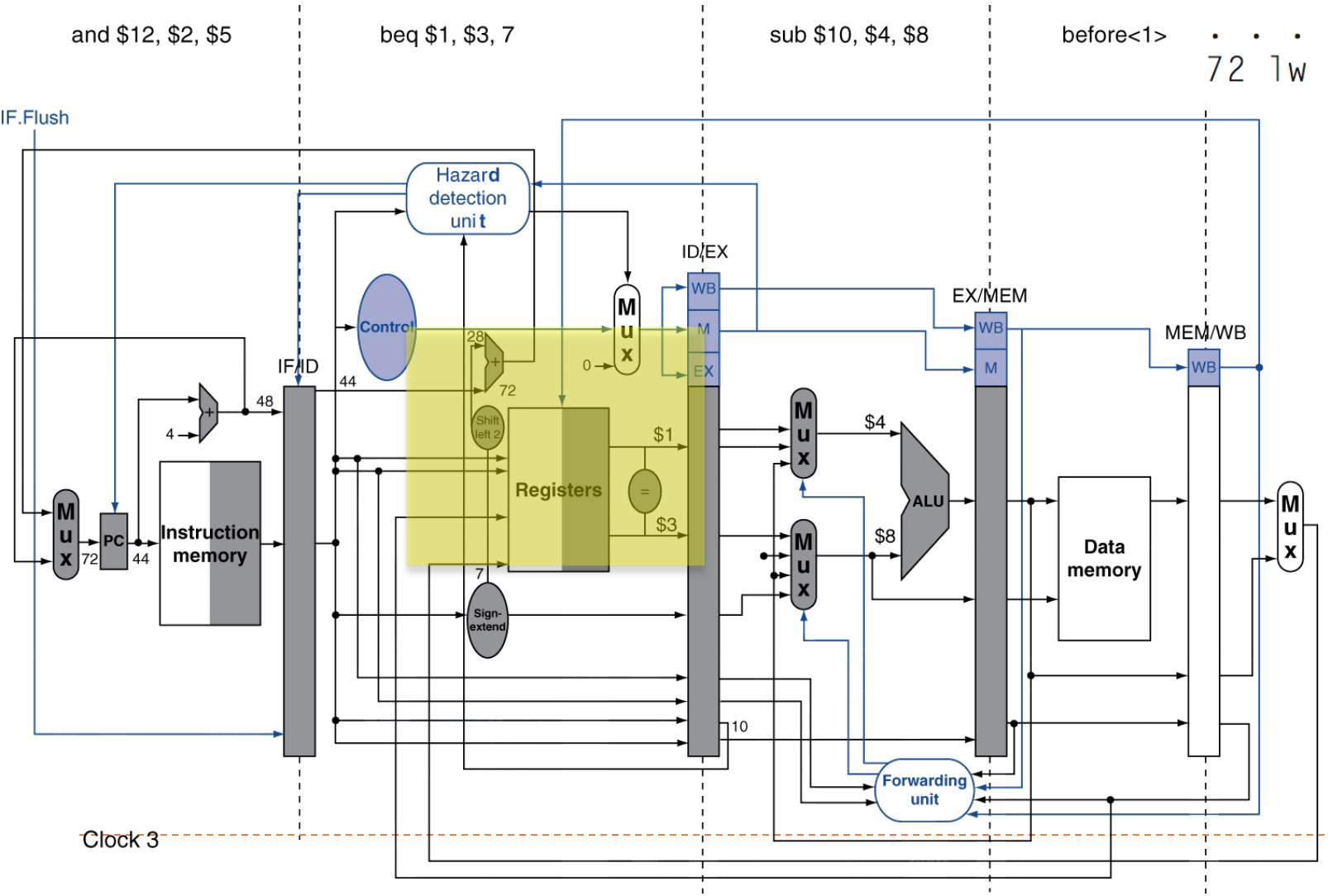
- Move hardware to determine outcome to ID stage → 1 cycle delay
  - Add Target address adder and Register comparator
- Example: branch taken

```
36 sub $10, $4, $8
40 beq $1, $3, 7 # PC-relative branch to 40+4+7*4=72
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

- Add is already fetched when beq outcome is determined

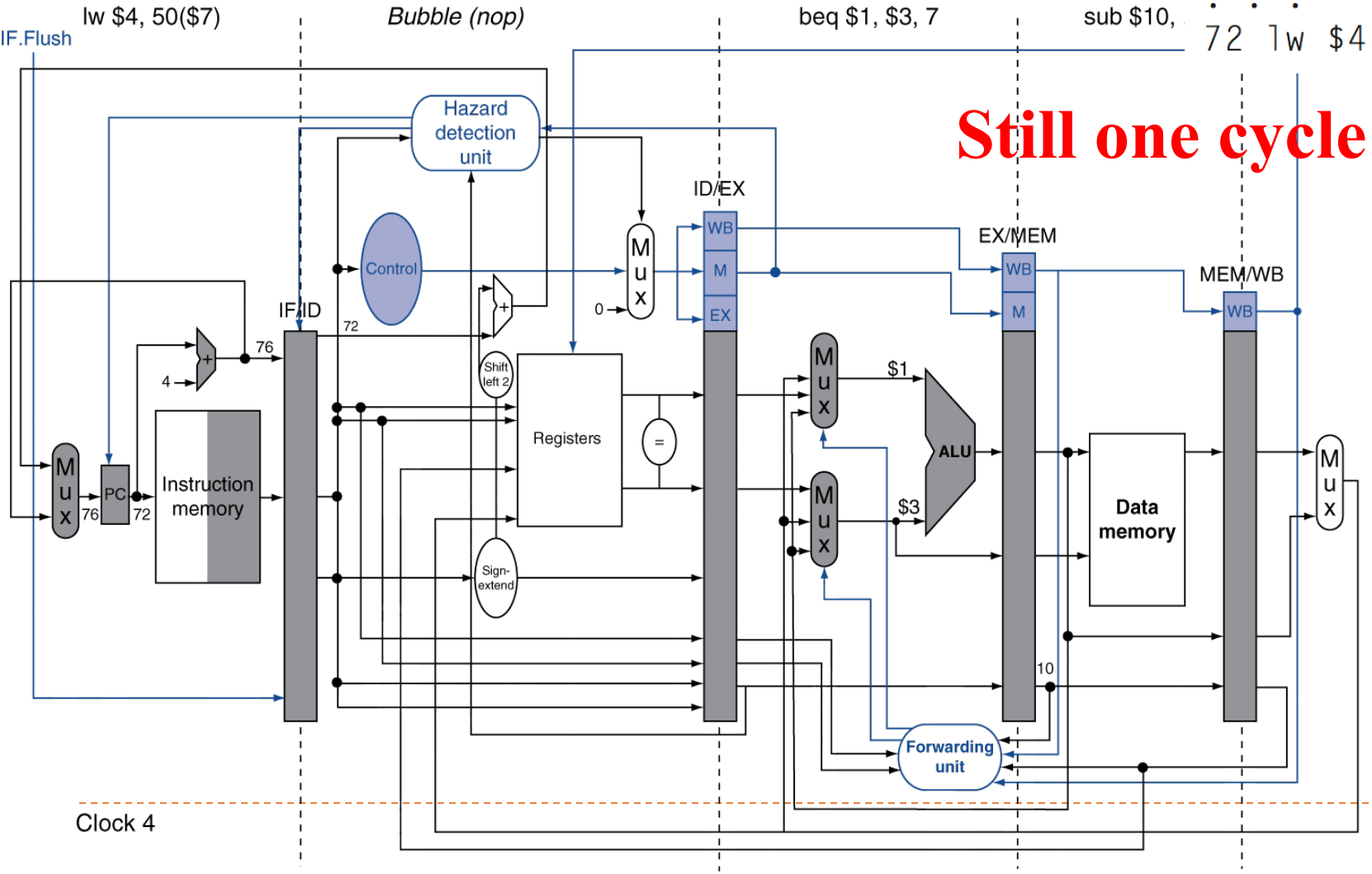
```
36 sub $10, $4, $8
40 beq $1, $3, 7 #
44 and $12, $2, $5
48 or  $13, $2, $6
52 add $14, $4, $2
56 slt $15, $6, $7
. . .
72 lw $4, 50($7)
```



Example: Branch Taken

- Add won't enter ID stage and branch target (lw) is fetched

```
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)
```



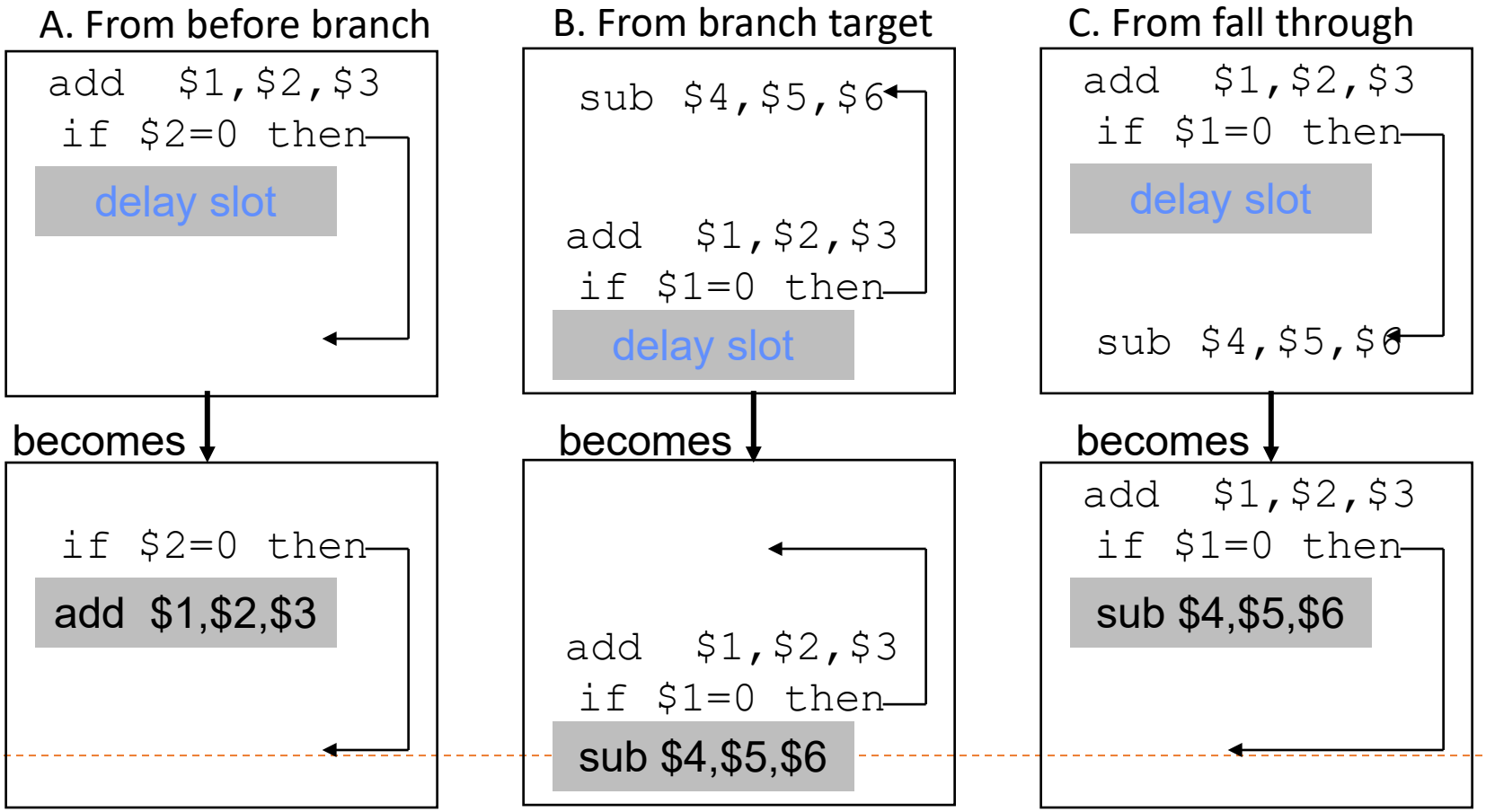
Still one cycle delay!

Four Branch Hazard Alternatives

- #1: Stall until branch direction is clear
- #2: Predict Branch Not Taken
  - Execute successor instructions in sequence
  - "Squash" instructions in pipeline if branch actually taken
  - Advantage of late pipeline state update
  - 47% MIPS branches not taken on average
  - PC+4 already calculated, so use it to get next instruction
- #3: Predict Branch Taken
  - 53% MIPS branches taken on average
  - But haven't calculated branch target address in MIPS
    - MIPS still incurs 1 cycle branch penalty
    - Other machines: branch target known before outcome

Four Branch Hazard Alternatives

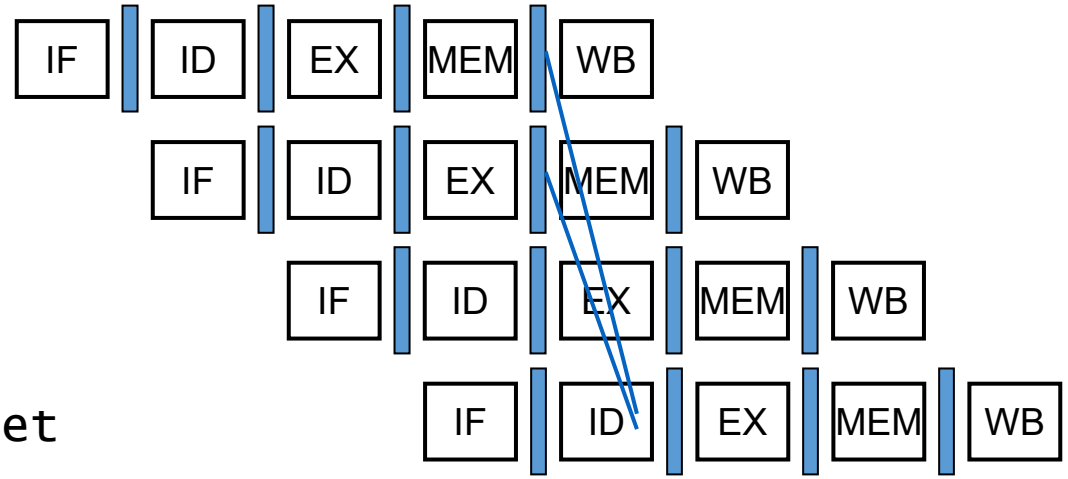
- #4: Schedule Branch Delay Slots
  - Exec an instruction in that delay slot regardless whether branch will be taken or not



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
```

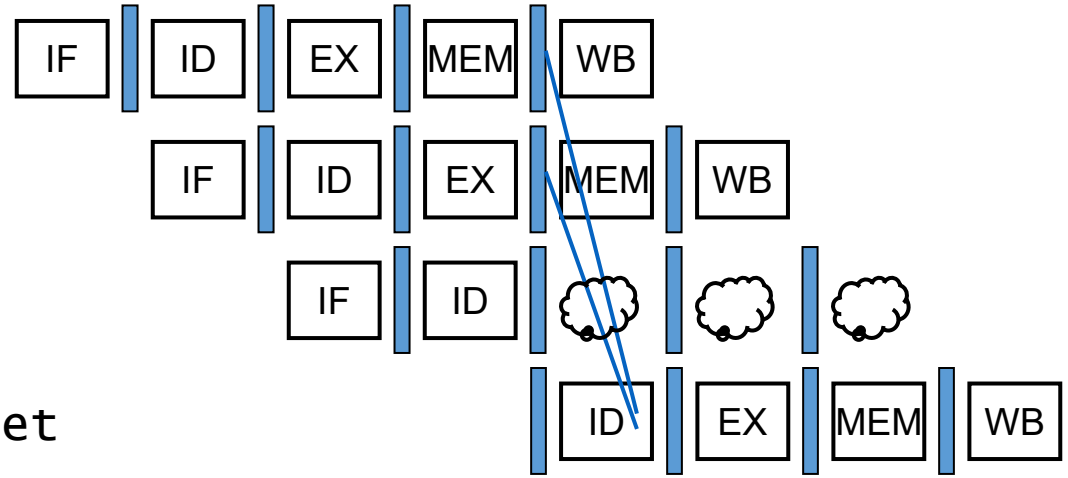


- Can resolve using forwarding

Data Hazards for Branches

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

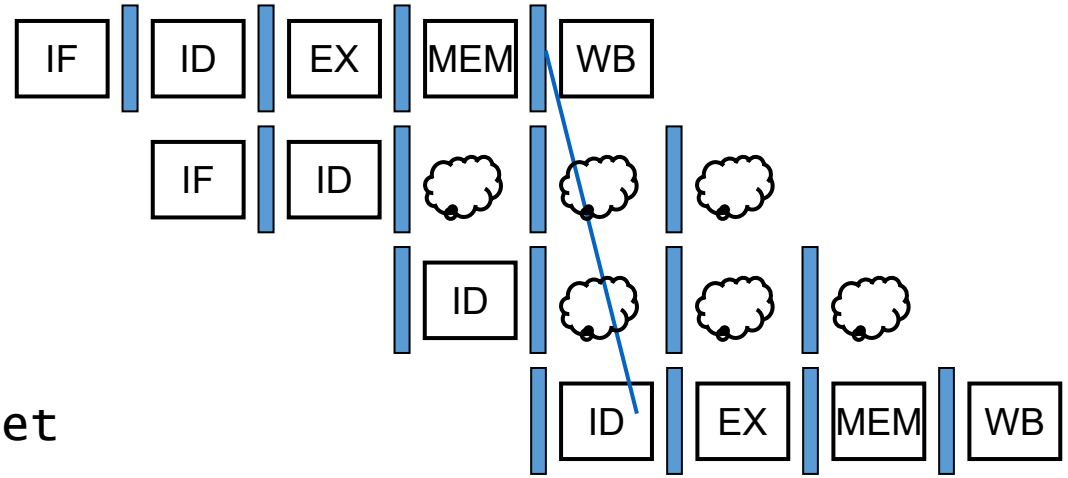
```
lw  $1, addr
add $4, $5, $6
beq stalled
beq $1, $4, target
```



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
```



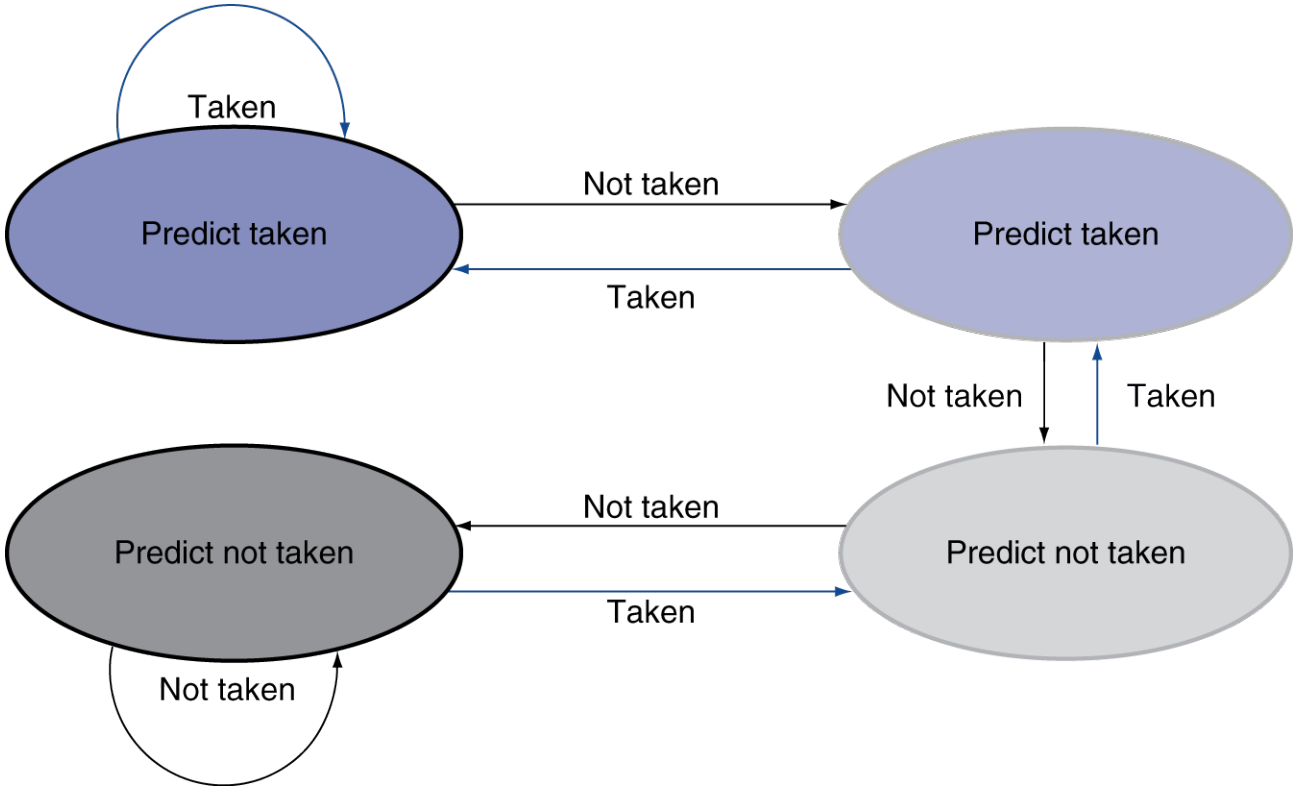
1-Bit Predictor: Shortcoming

- Inner loop branches mispredicted twice!

- Mispredict as taken on last iteration of inner loop
- Then mispredict as not taken on first iteration of inner loop next time around

2-Bit Predictor

- Only change prediction on two successive mispredictions



Calculating the Branch Target

- Even with predictor, still need to calculate the target address
  - 1-cycle penalty for a taken branch
- Branch target buffer
  - Cache of target addresses
  - Indexed by PC when instruction fetched
    - If hit and instruction is branch predicted taken, can fetch target immediately

Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction
  - Branch prediction buffer (aka branch history table)
  - Indexed by recent branch instruction addresses
  - Stores outcome (taken/not taken)
  - To execute a branch
    - Check table, expect the same outcome
    - Start fetching from fall-through or target
    - If wrong, flush pipeline and flip prediction

Speculation and Exceptions

- What if exception occurs on a speculatively executed instruction?
  - e.g., speculative load before null-pointer check
- Static speculation
  - Can add ISA support for deferring exceptions
- Dynamic speculation
  - Can buffer exceptions until instruction completion (which may not occur)

Matrix Multiply

- Unrolled C code

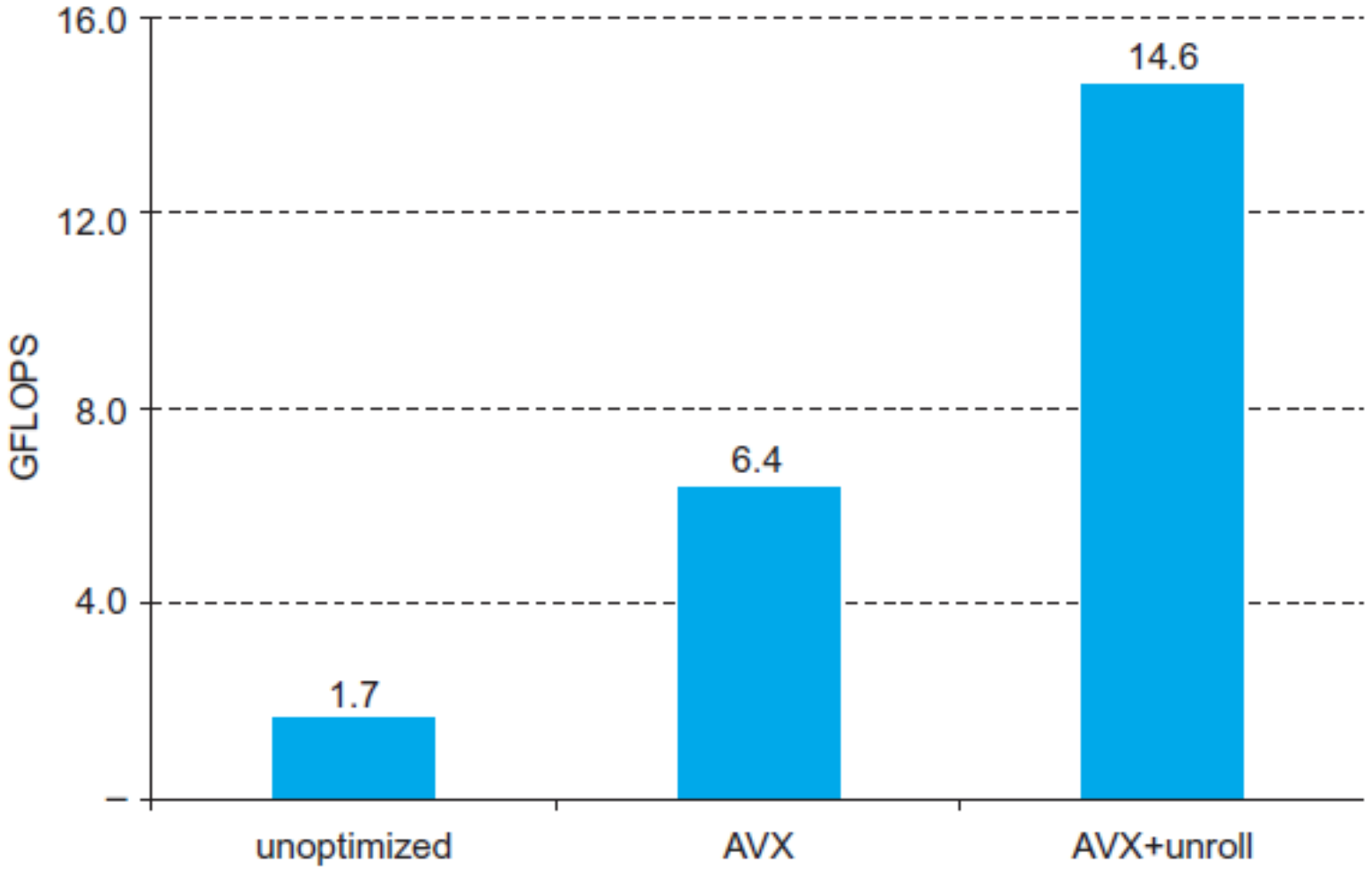
```
1 #include <x86intrin.h>
2 #define UNROLL (4)
3
4 void dgemm (int n, double* A, double* B, double* C)
5 {
6  for ( int i = 0; i < n; i+=UNROLL*4 )
7   for ( int j = 0; j < n; j++ ) {
8    __m256d c[4];
9    for ( int x = 0; x < UNROLL; x++ )
10    c[x] = _mm256_load_pd(C+i+x*4+j*n);
11
12   for( int k = 0; k < n; k++ )
13   {
14    __m256d b = _mm256_broadcast_sd(B+k+j*n);
15    for (int x = 0; x < UNROLL; x++)
16    c[x] = _mm256_add_pd(c[x],
17                         _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
18   }
19
20    for ( int x = 0; x < UNROLL; x++ )
21     _mm256_store_pd(C+i+x*4+j*n, c[x]);
22  }
23 }
```

Matrix Multiply

- Assembly code:

```
1 vmovapd (%r11),%ymm4                  # Load 4 elements of C into %ymm4
2 mov %rbx,%rax                         # register %rax = %rbx
3 xor %ecx,%ecx                         # register %ecx = 0
4 vmovapd 0x20(%r11),%ymm3              # Load 4 elements of C into %ymm3
5 vmovapd 0x40(%r11),%ymm2              # Load 4 elements of C into %ymm2
6 vmovapd 0x60(%r11),%ymm1              # Load 4 elements of C into %ymm1
7 vbroadcastsd (%rcx,%r9,1),%ymm0       # Make 4 copies of B element
8 add $0x8,%rcx # register %rcx = %rcx + 8
9 vmulpd (%rax),%ymm0,%ymm5             # Parallel mul %ymm1,4 A elements
10 vaddpd %ymm5,%ymm4,%ymm4             # Parallel add %ymm5, %ymm4
11 vmulpd 0x20(%rax),%ymm0,%ymm5        # Parallel mul %ymm1,4 A elements
12 vaddpd %ymm5,%ymm3,%ymm3             # Parallel add %ymm5, %ymm3
13 vmulpd 0x40(%rax),%ymm0,%ymm5        # Parallel mul %ymm1,4 A elements
14 vmulpd 0x60(%rax),%ymm0,%ymm0        # Parallel mul %ymm1,4 A elements
15 add %r8,%rax                         # register %rax = %rax + %r8
16 cmp %r10,%rcx                        # compare %r8 to %rax
17 vaddpd %ymm5,%ymm2,%ymm2             # Parallel add %ymm5, %ymm2
18 vaddpd %ymm0,%ymm1,%ymm1             # Parallel add %ymm0, %ymm1
19 jne 68 <dgemm+0x68>                  # jump if not %r8 != %rax
20 add $0x1,%esi                        # register % esi = % esi + 1
21 vmovapd %ymm4,(%r11)                 # Store %ymm4 into 4 C elements
22 vmovapd %ymm3,0x20(%r11)             # Store %ymm3 into 4 C elements
23 vmovapd %ymm2,0x40(%r11)             # Store %ymm2 into 4 C elements
24 vmovapd %ymm1,0x60(%r11)             # Store %ymm1 into 4 C elements
```

Performance Impact



Concluding Remarks

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control
- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall

Fallacies

- Pipelining is easy (!)
  - The basic idea is easy
  - The devil is in the details
    - e.g., detecting data hazards
- Pipelining is independent of technology
  - So why haven't we always done pipelining?
  - More transistors make more advanced techniques feasible
  - Pipeline-related ISA design needs to take account of technology trends
    - e.g., predicated instructions

Pitfalls

- Poor ISA design can make pipelining harder
  - e.g., complex instruction sets (VAX, IA-32)
    - Significant overhead to make pipelining work
    - IA-32 micro-op approach
  - e.g., complex addressing modes
    - Register update side effects, memory indirection
  - e.g., delayed branches
    - Advanced pipelines have long delay slots

MeltDown & Spectre

- https://www.youtube.com/watch?v=bs0xswK0eZk

Reference Readings

- Patterson, "Computer Organization And Design"
  - Sec 4.5, 4.6,4.7, 4.8