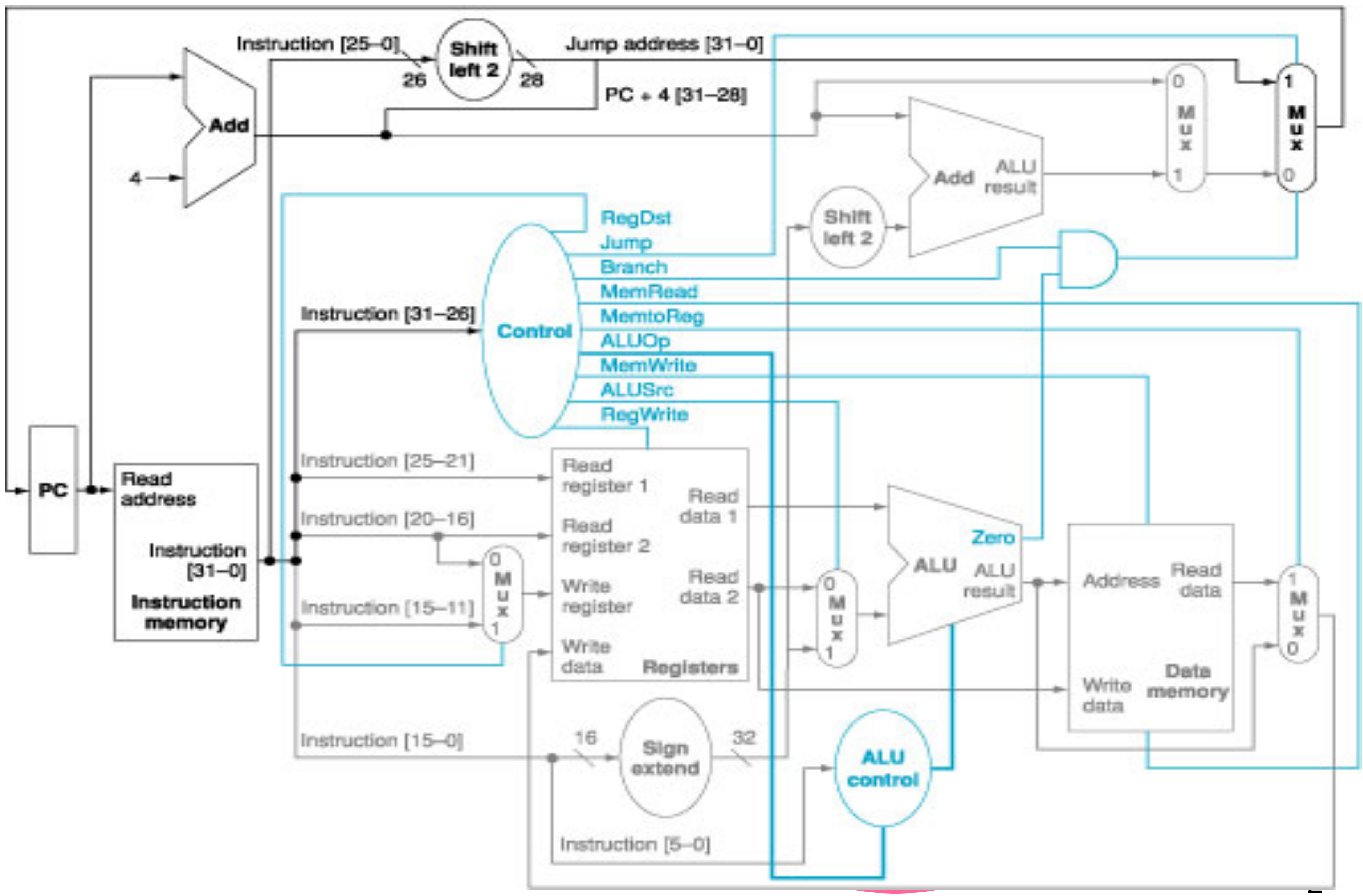
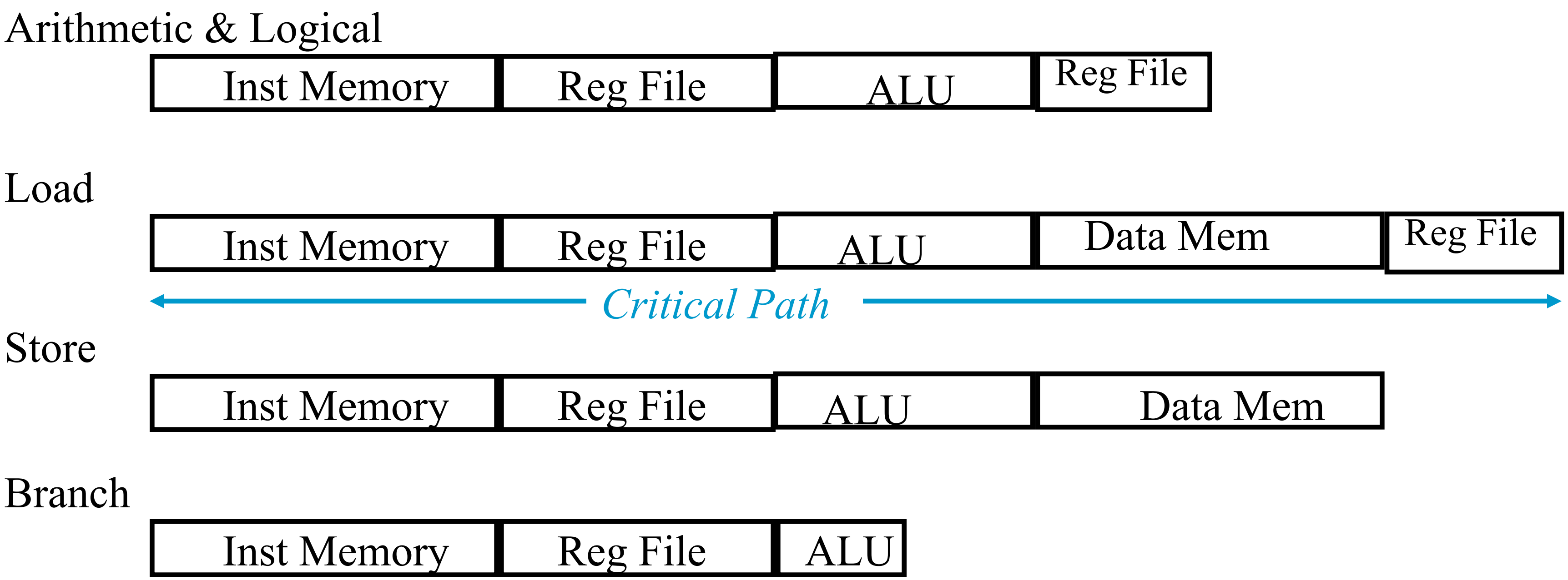
Lecture 5

Pipelining

Single Cycle Implementations

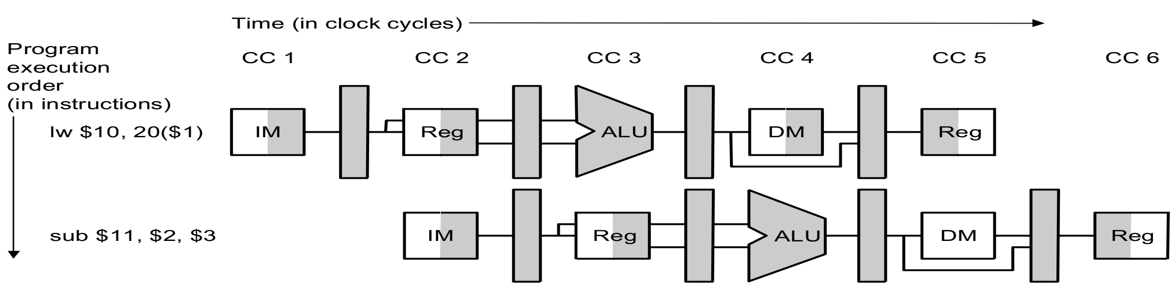


What’s wrong with our CPI=1 processor?



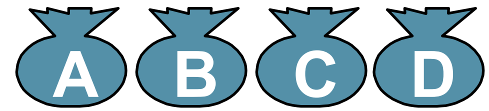
- Long Cycle Time
- All instructions take as much time as the slowest
- Real memory is not so nice as our idealized memory
  - cannot always get the job done in one (short) cycle

Pipelining

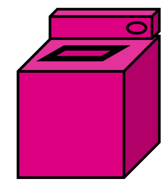

Time (in clock cycles)
Program execution order (in instructions)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
lw $10, 20($1)
sub $11, $2, $3
IM
Reg
ALU
DM
Reg
IM
Reg
ALU
DM
Reg

Pipelining is Natural!

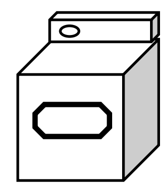
- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold



- Washer takes 30 minutes



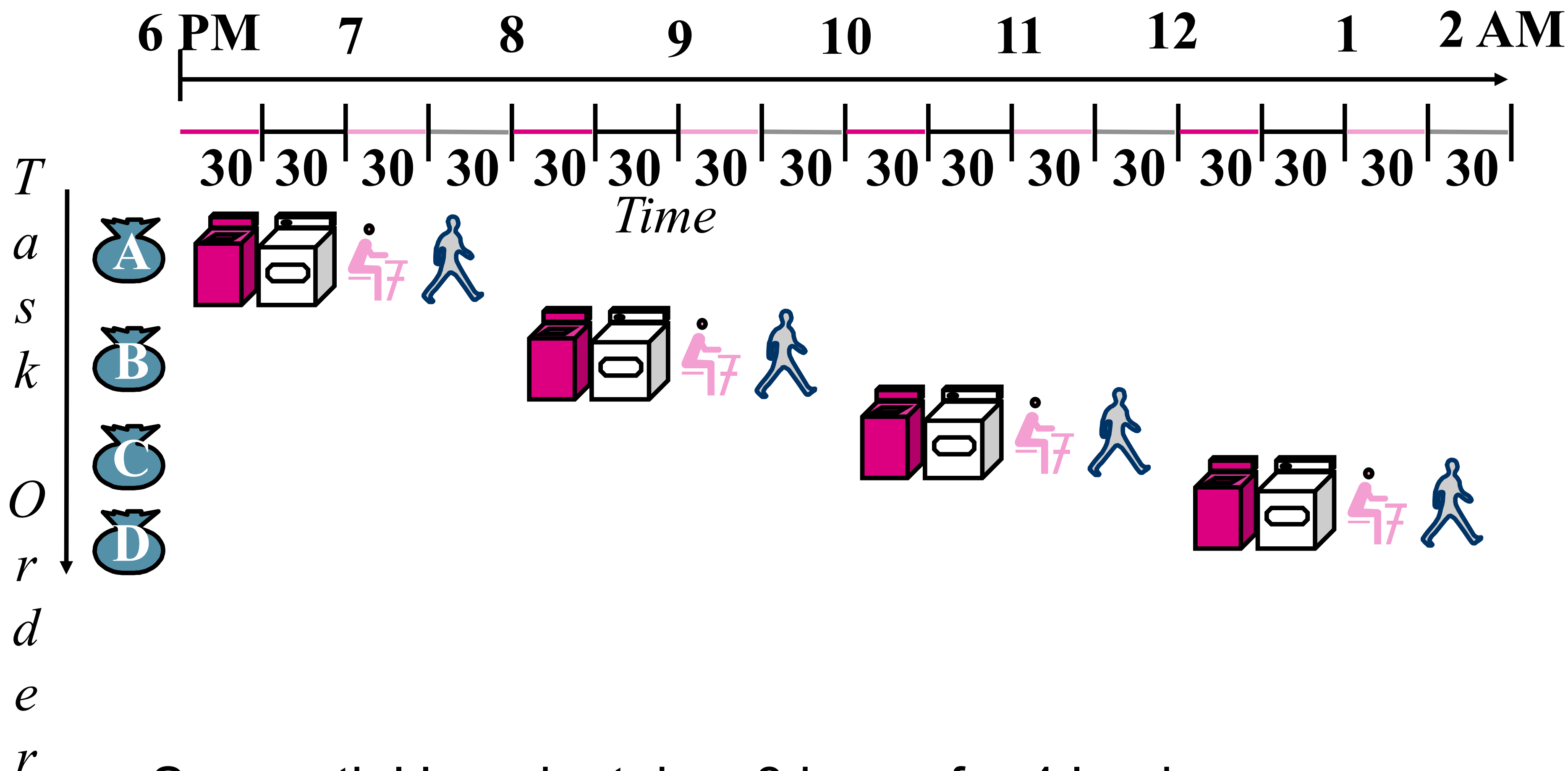
- Dryer takes 30 minutes



- “Folder” takes 30 minutes

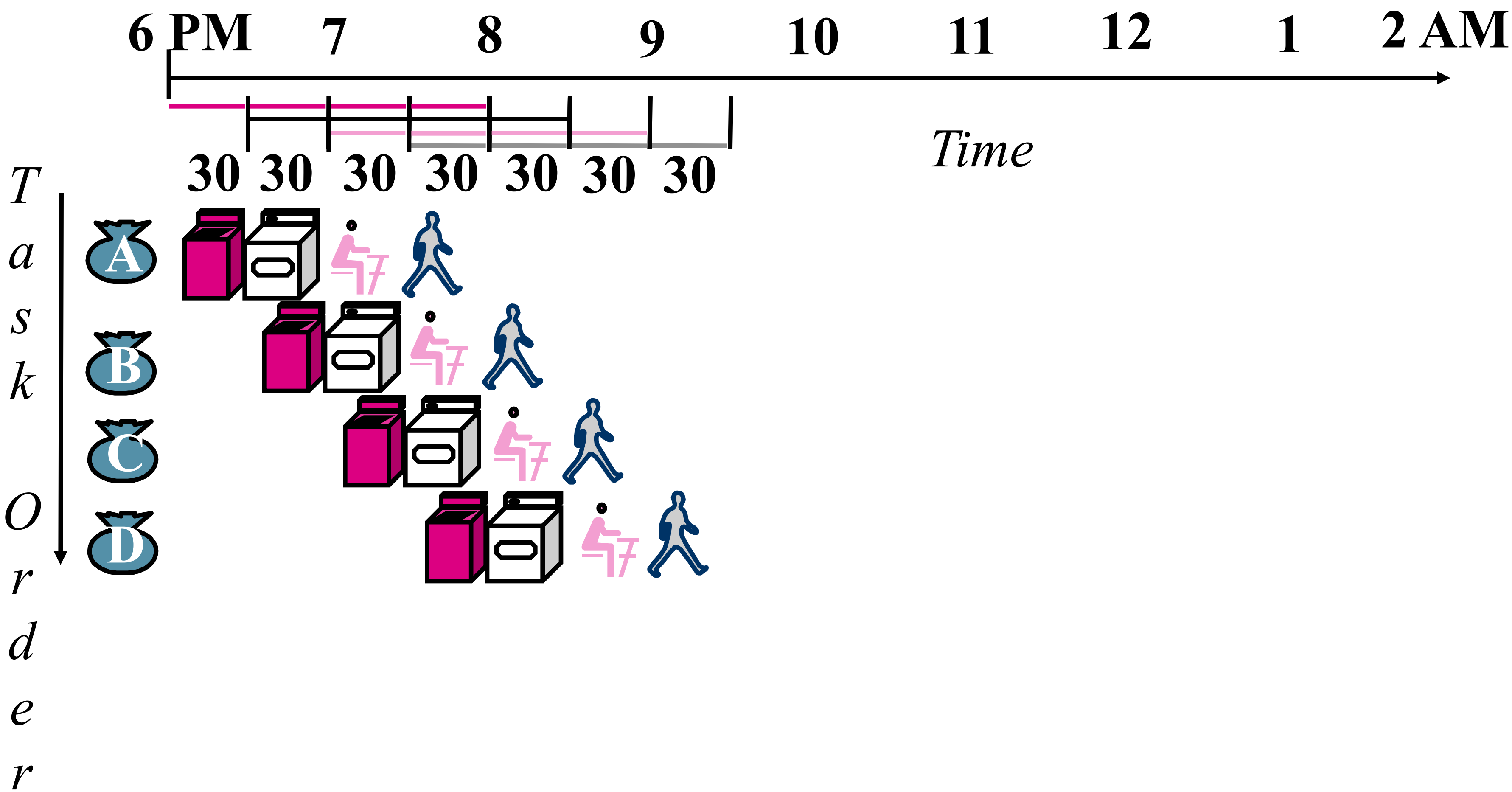
- Putting clothes into drawers takes 30 minutes

Sequential Laundry



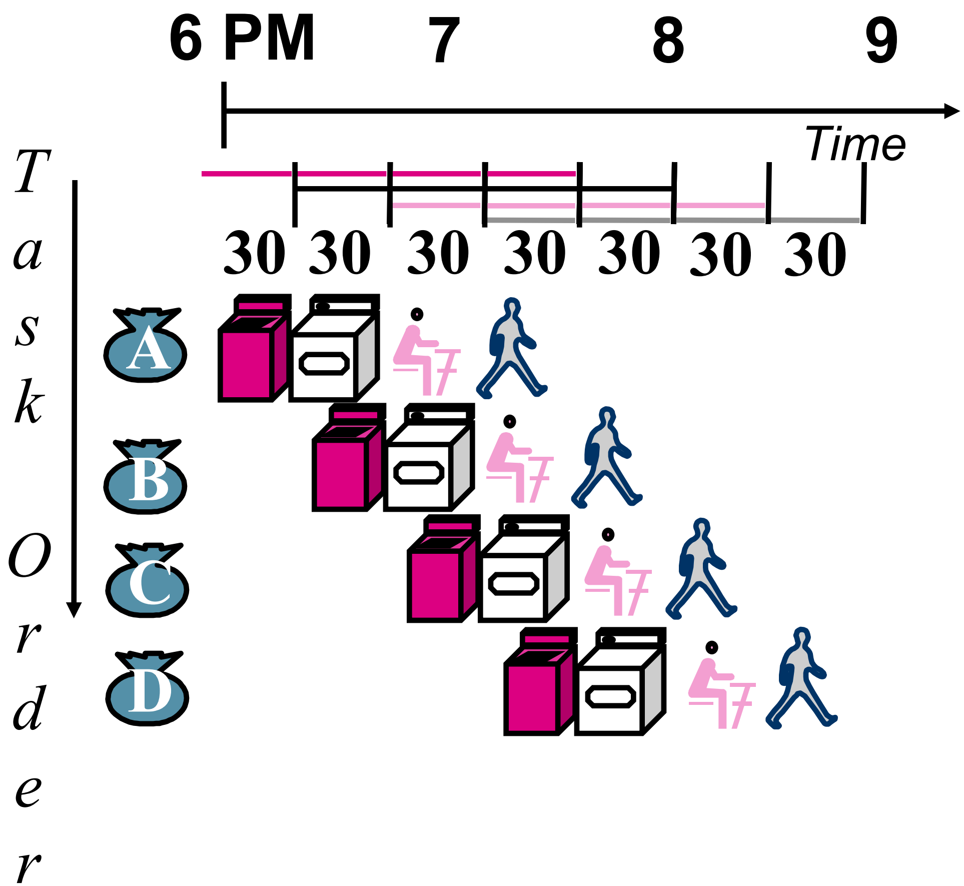
- Sequential laundry takes 8 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Pipelined Laundry: Start work ASAP



- Pipelined laundry takes 3.5 hours for 4 loads!

Pipelining Lessons



- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously using different resources
- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup

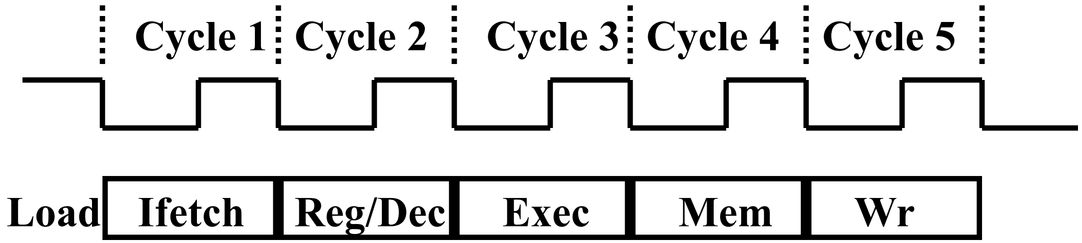
Pipelining in CPU

- Break the instruction into smaller steps
- Execute each step (instead of the entire instruction) in 1 clock cycle
  - Cycle time: time it takes to execute the longest step
  - Try to make all the steps have similar length

5 steps in MIPS

- Instruction Fetch
- Instruction Decode and Register Fetch
- Execution, Memory Address Computation, or Branch Completion
- Memory Access or R-type instruction completion
- Write-back

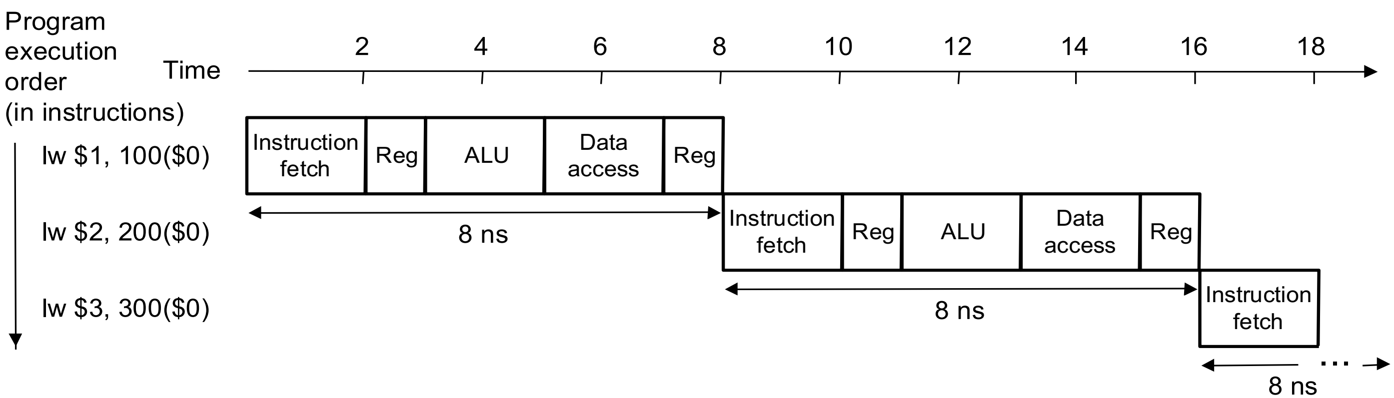
The Five Stages of Load

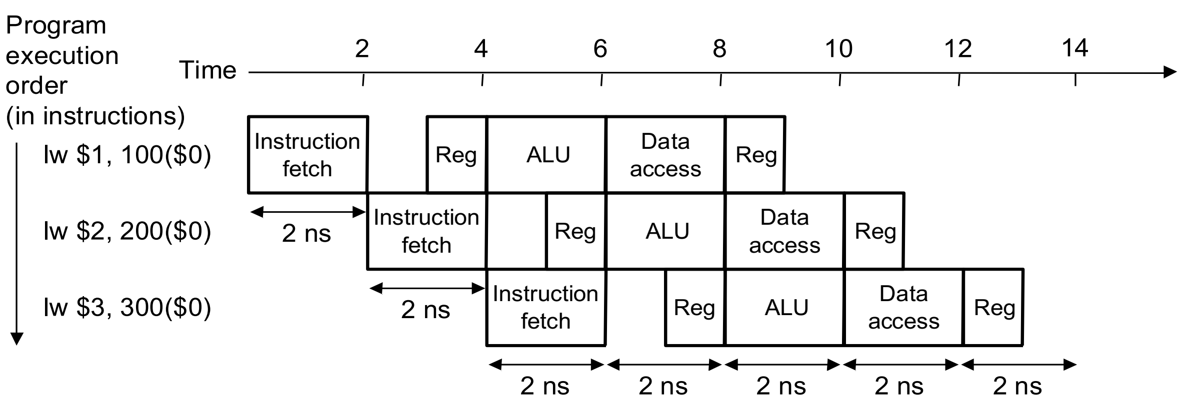


- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Read the data from the Data Memory
- Wr: Write the data back to the register file

Pipelining

- Improve performance by increasing instruction throughput





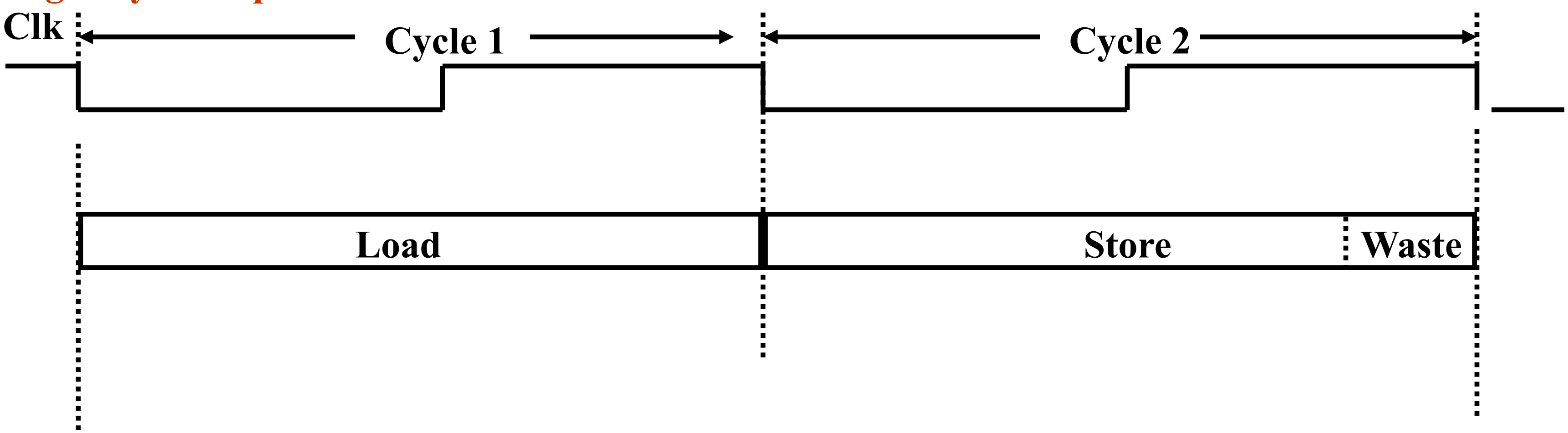
Time to execute 4 load instructions:

Single cycle: 8 x 4 = 32 ns

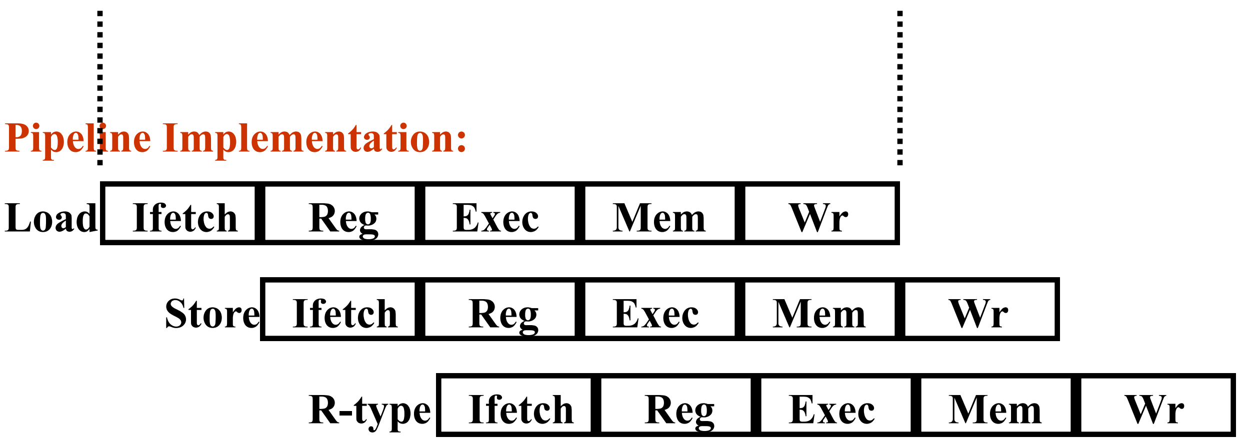
Pipeline: 4 x 1 + 4 (time to drain the pipeline) = 8 Cycle = 16 ns

Single Cycle vs. Pipeline

Single Cycle Implementation:

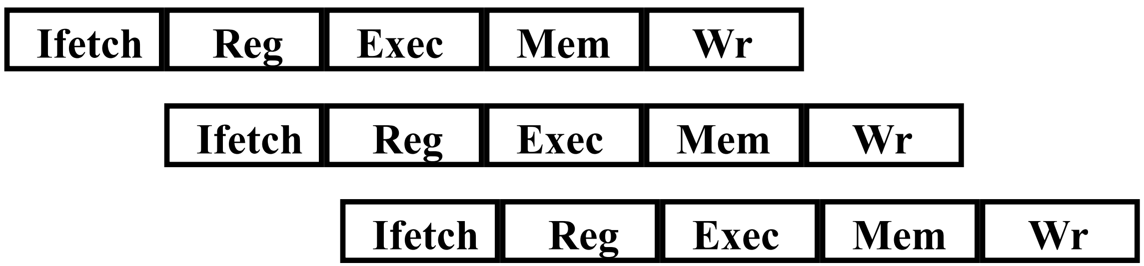


Pipeline Implementation:



Ideal Pipeline Performance

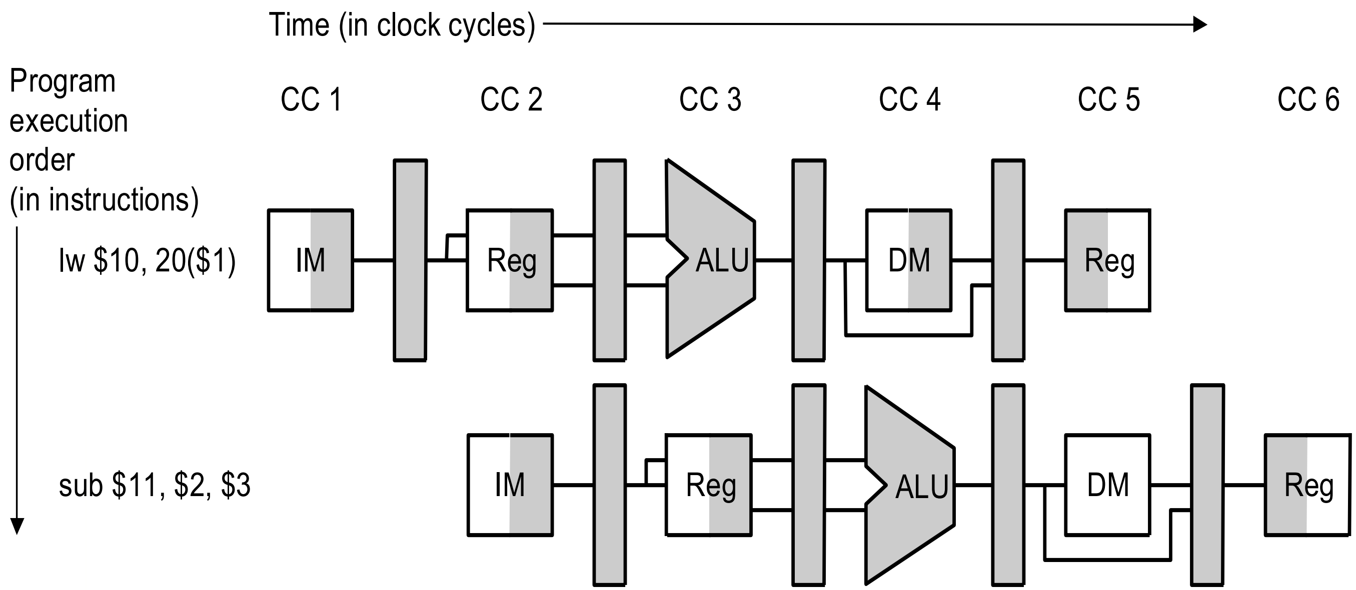
- Ideal CPI = 1
- Time to execute n instruction
  - 1 x n + time to drain the pipeline



- Ideal speedup from pipelining == # of pipeline stages
  - If the stages are perfectly balanced, and a large number of instructions

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of pipe stages}}$$

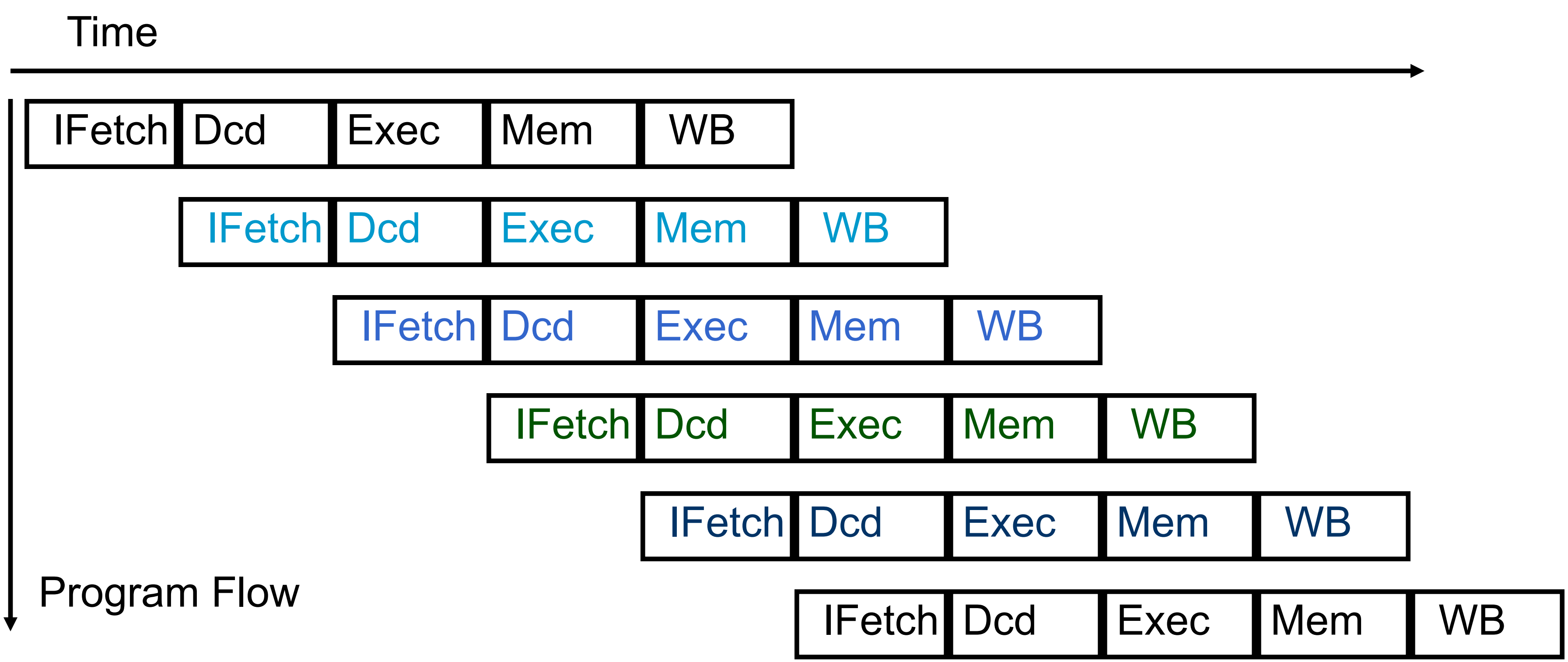
Graphically Representing Pipelines



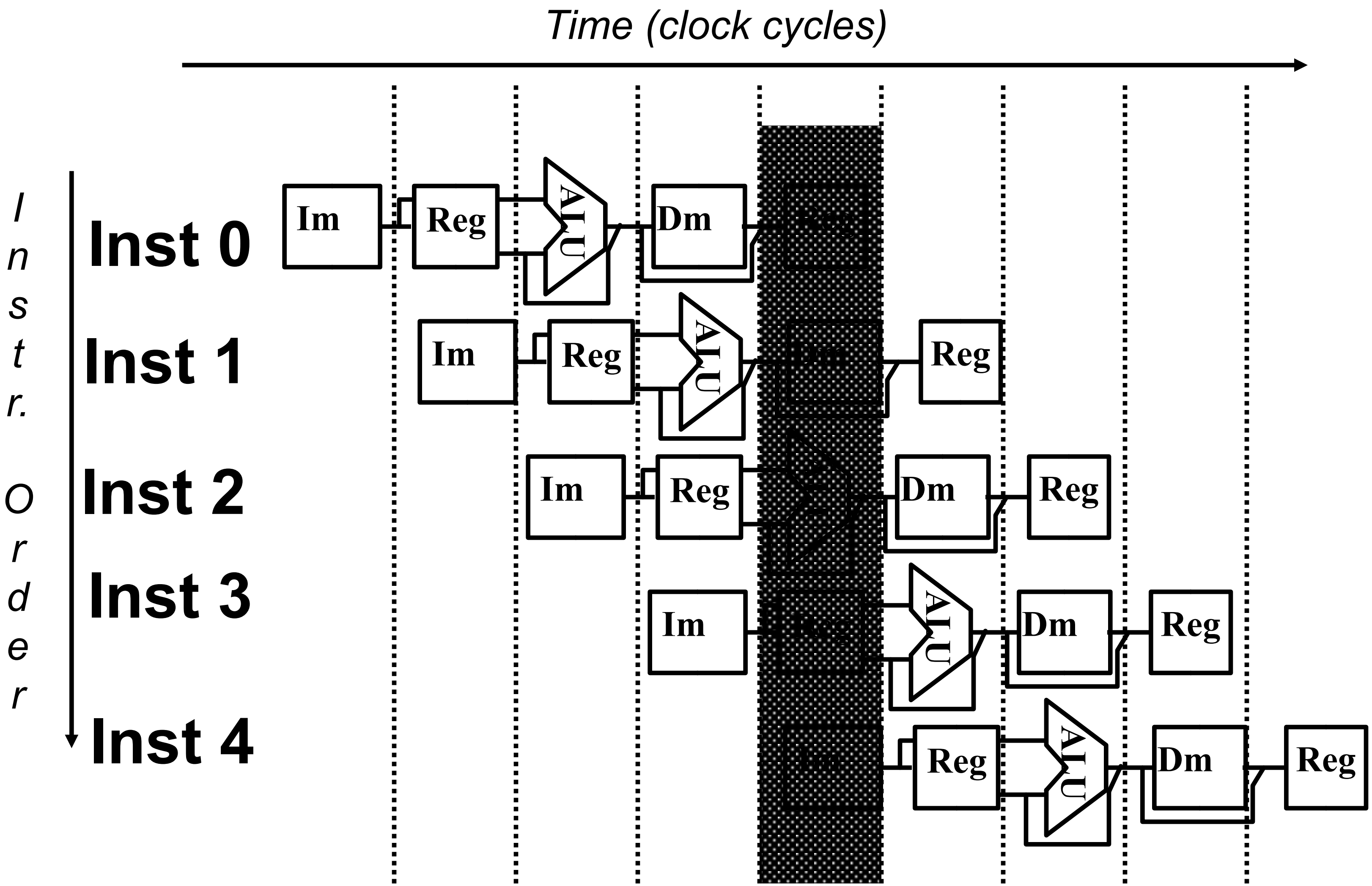
(right half highlight means read, left half write)

- Can help with answering questions like:
  - how many cycles does it take to execute this code?
  - what is the ALU doing during cycle 4?
  - use this representation to help understand datapaths

Conventional Pipelined Execution Representation



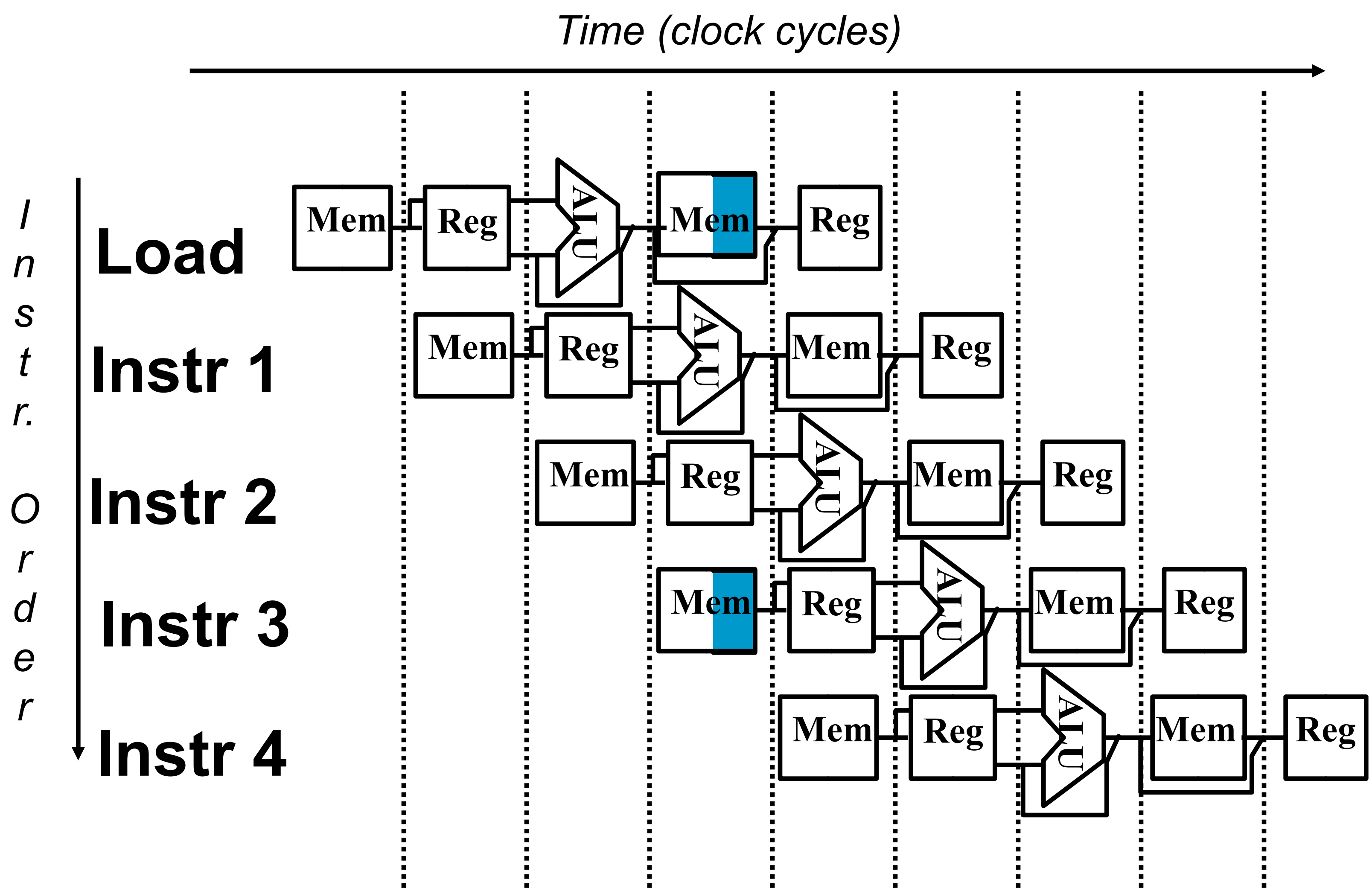
Instruction Pipelining



Can pipelining get us into trouble?

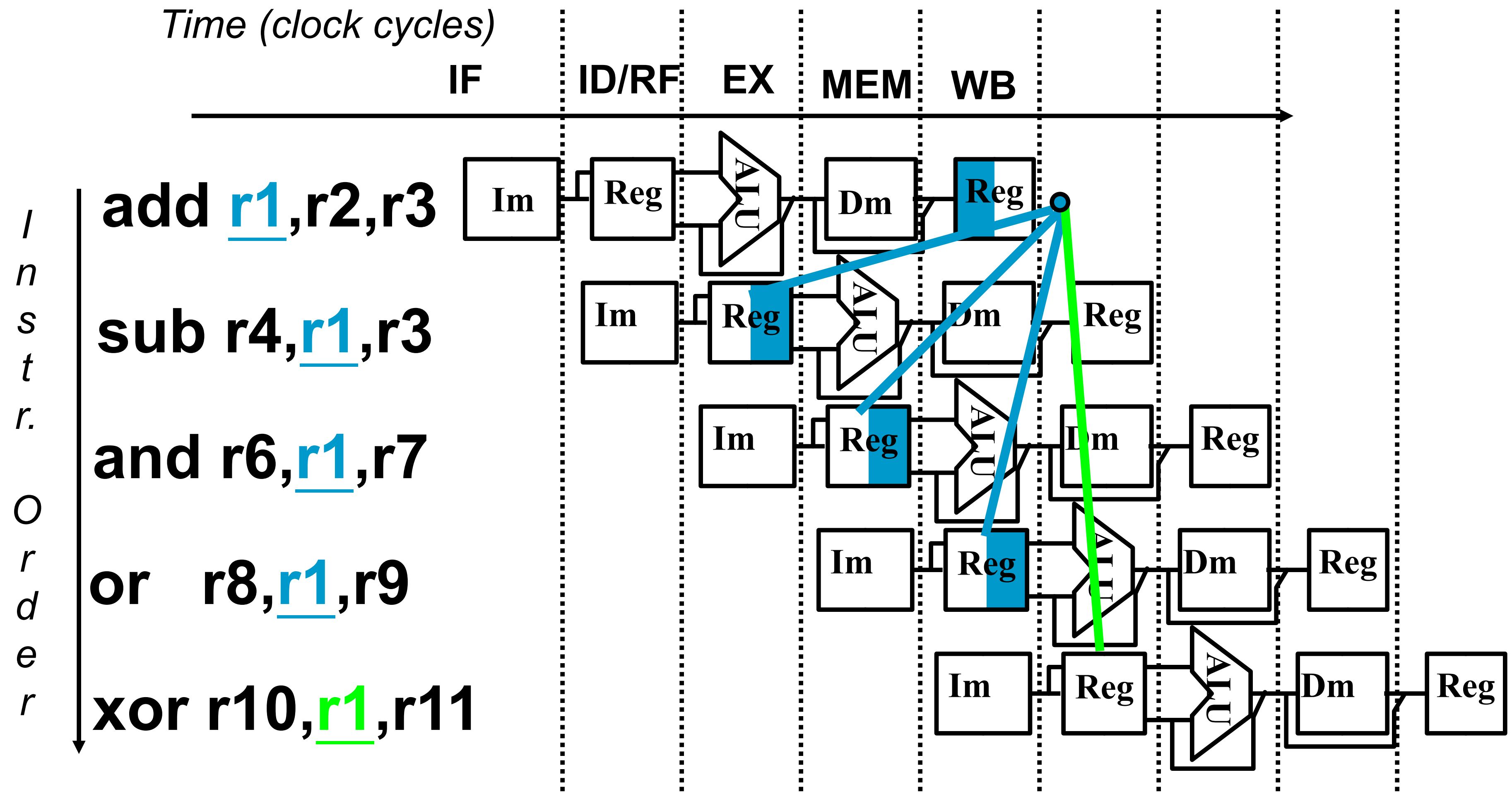
- Yes: Pipeline Hazards
  - structural hazards: attempt to use the same resource two different ways at the same time
    - E.g., combined washer/dryer would be a structural hazard or folder busy doing something else (watching TV)
  - data hazards: attempt to use item before it is ready
    - E.g., one sock of pair in dryer and one in washer; can't fold until get sock from washer through dryer
    - instruction depends on result of prior instruction still in the pipeline
  - control hazards: attempt to make a decision before condition is evaluated
    - E.g., washing football uniforms and need to get proper detergent level ; need to see after dryer before next load in
    - branch instructions
- Can always resolve hazards by waiting
  - pipeline control must detect the hazard
  - take action (or delay action) to resolve hazards

Single Memory is a Structural Hazard

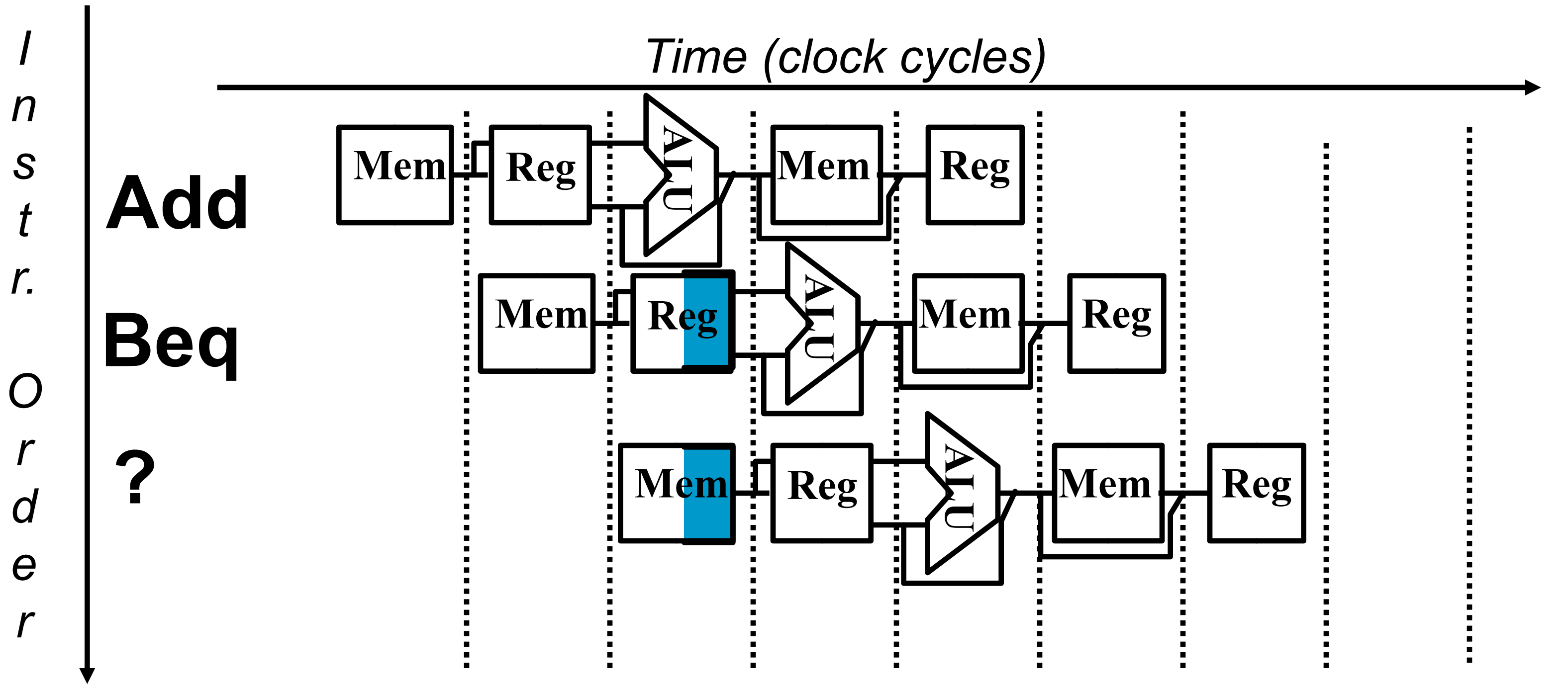


Data Hazard on r1:

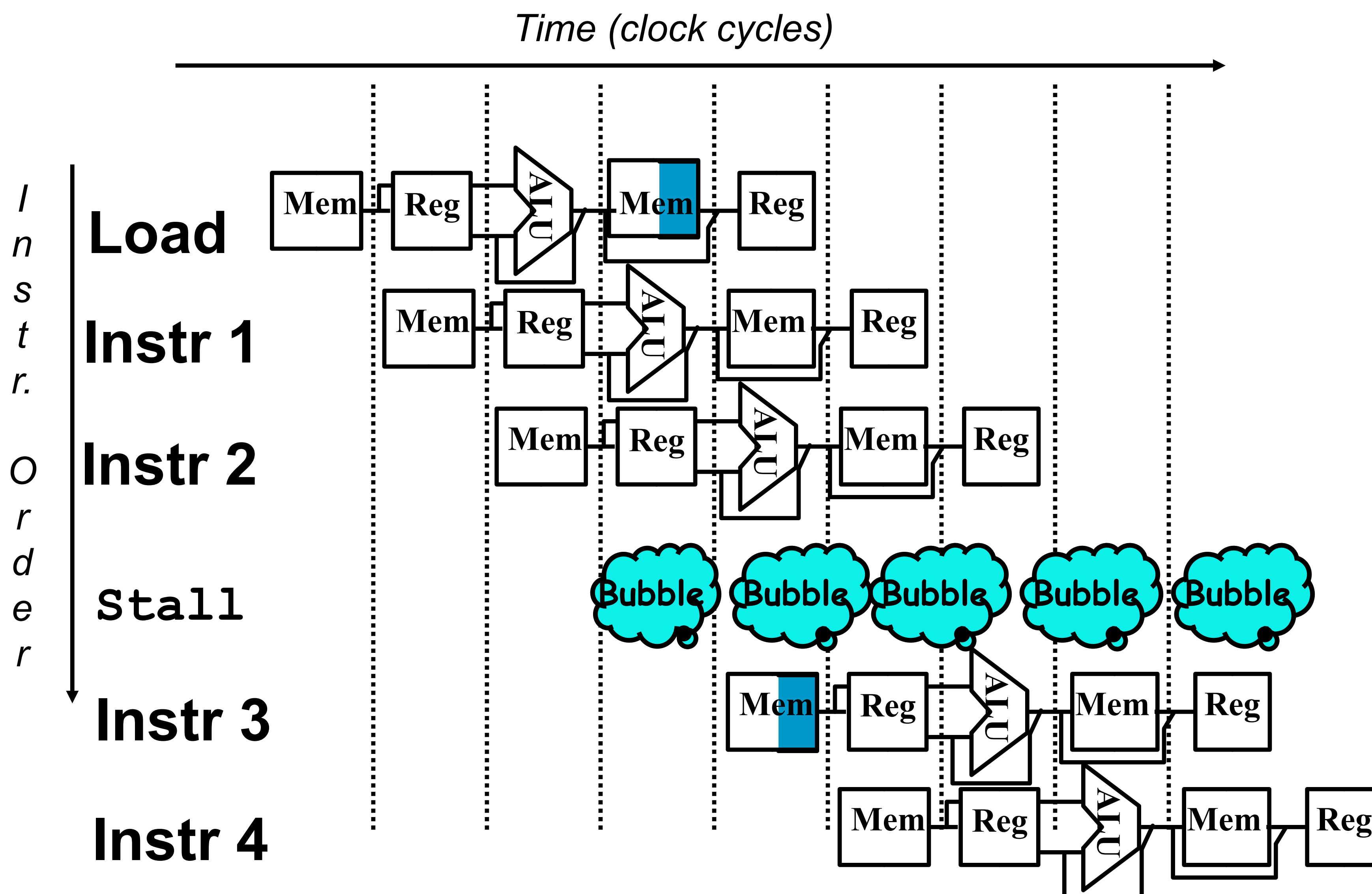
- **Dependencies backwards in time are hazards**



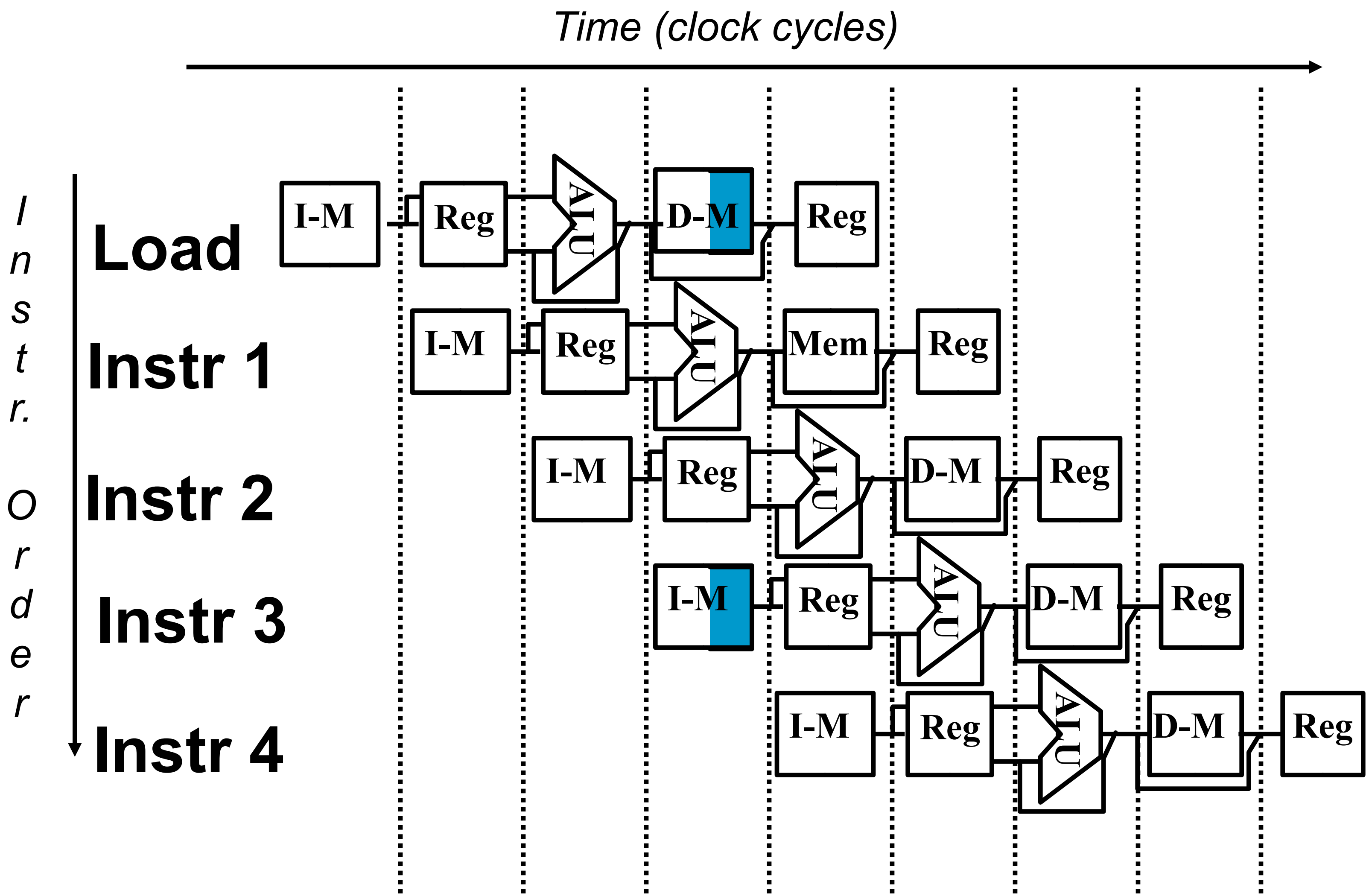
Control Hazard



How to solve structure hazards: (1) Stall

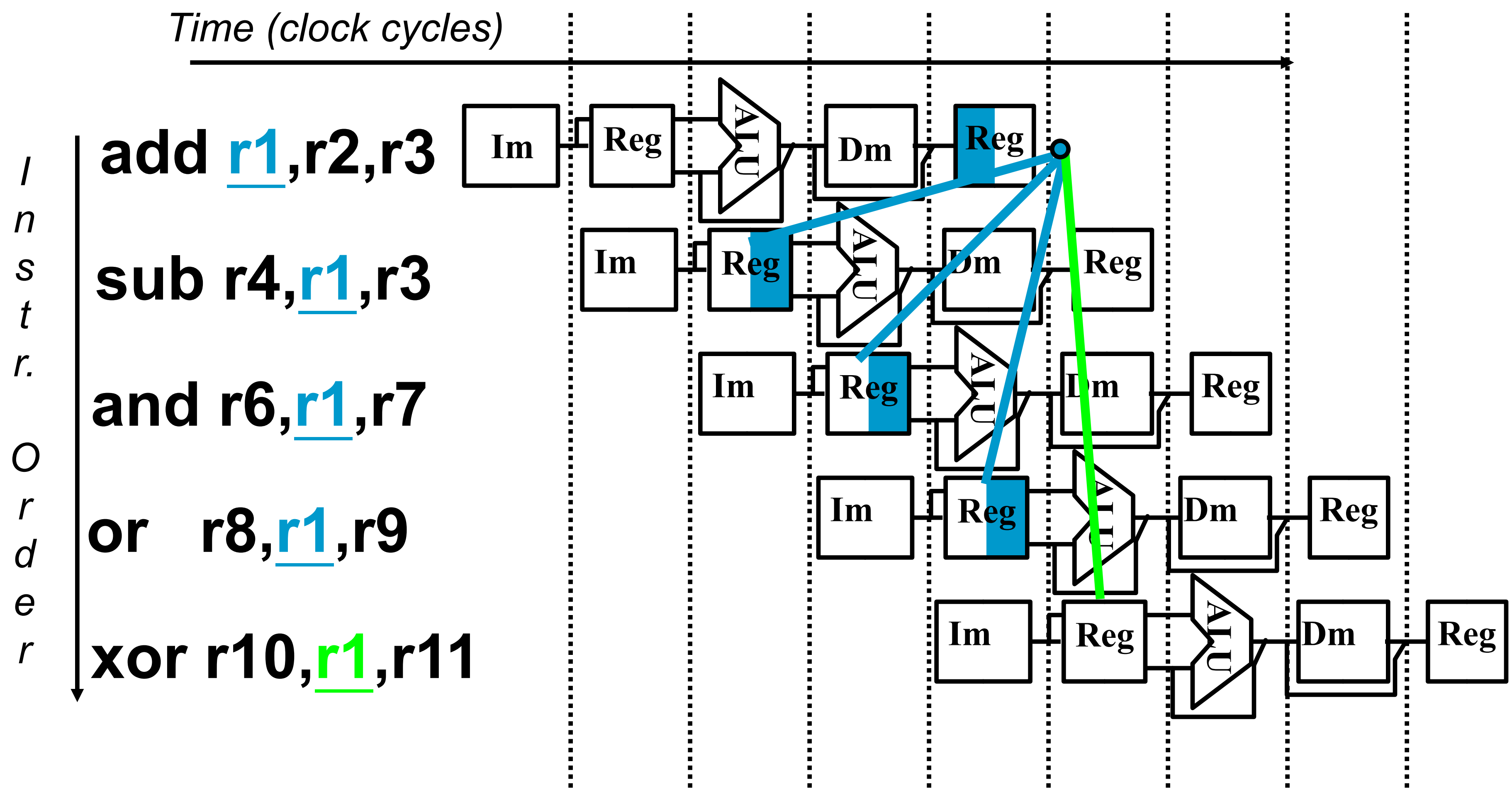


How to solve structure hazards: (2) Split instruction and data memory



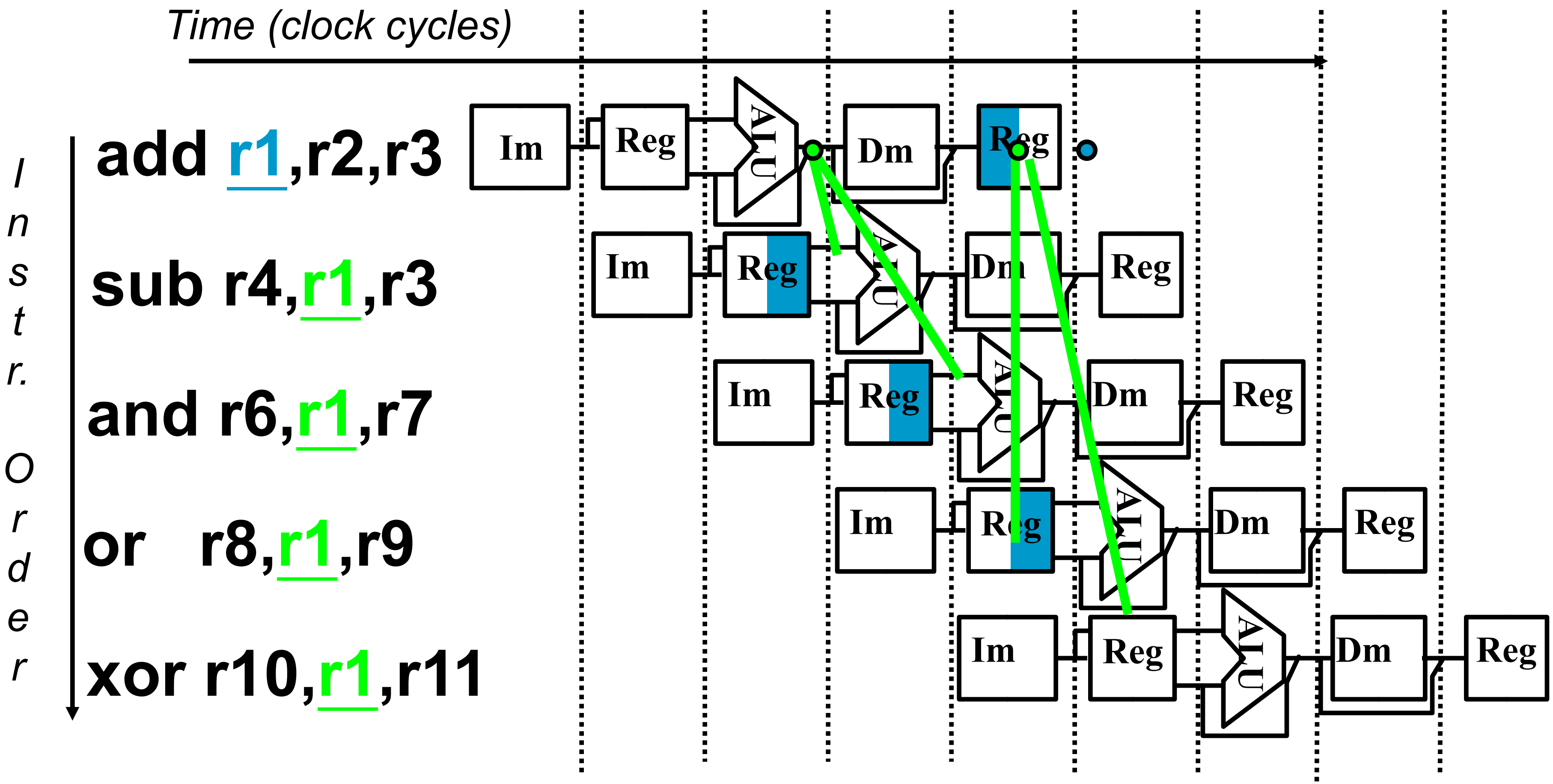
Data Hazard on r1:

- **Dependencies backwards in time are hazards**



Data Hazard Solution:

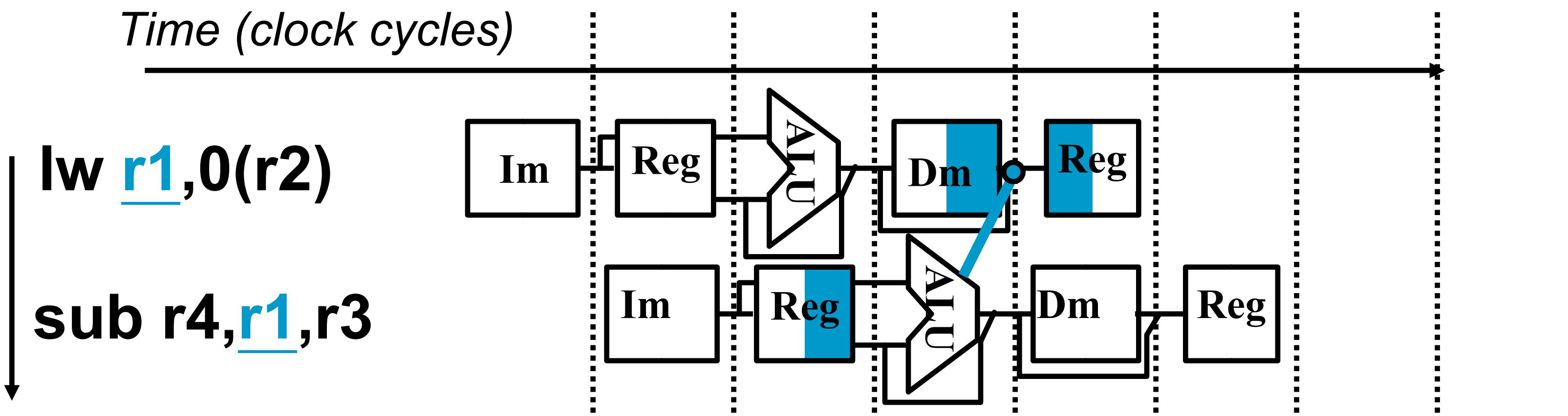
- “Forward” result from one stage to another



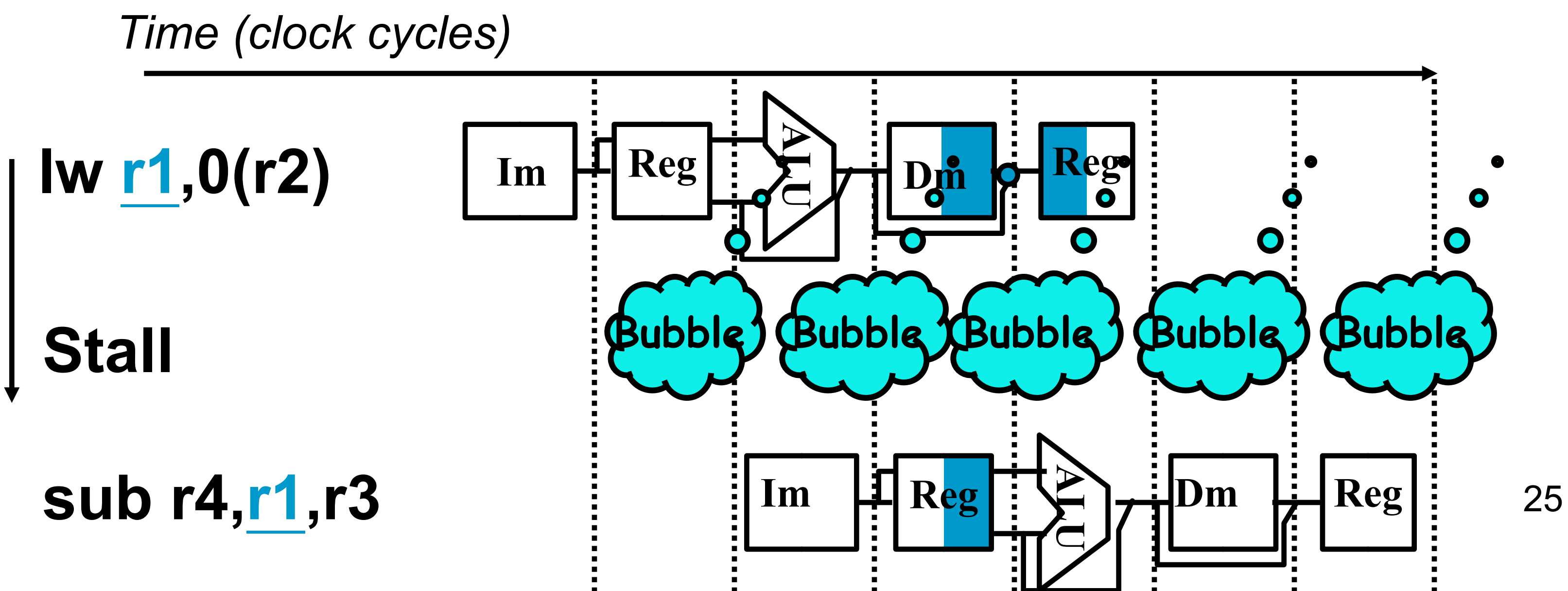
- “or” OK if register writes occur in the first half of the clock cycle, and register reads in the last half of the clock cycle

Forwarding (or Bypassing): What about Loads

- **Load-use data hazard: cannot solve with forwarding**



- **Must delay/stall instruction dependent on loads**



Reordering Code to Avoid Pipeline Stalls

A = B + E;
C = B + F;

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

Reordering Code to Avoid Pipeline Stalls

A = B + E;
C = B + F;

On a pipelined processor with forwarding, the reordered sequence will complete in two fewer cycles than left-hand side sequence.

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

Load-use
Forwarding
Load-use
Forwarding

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

Forwarding
Forwarding

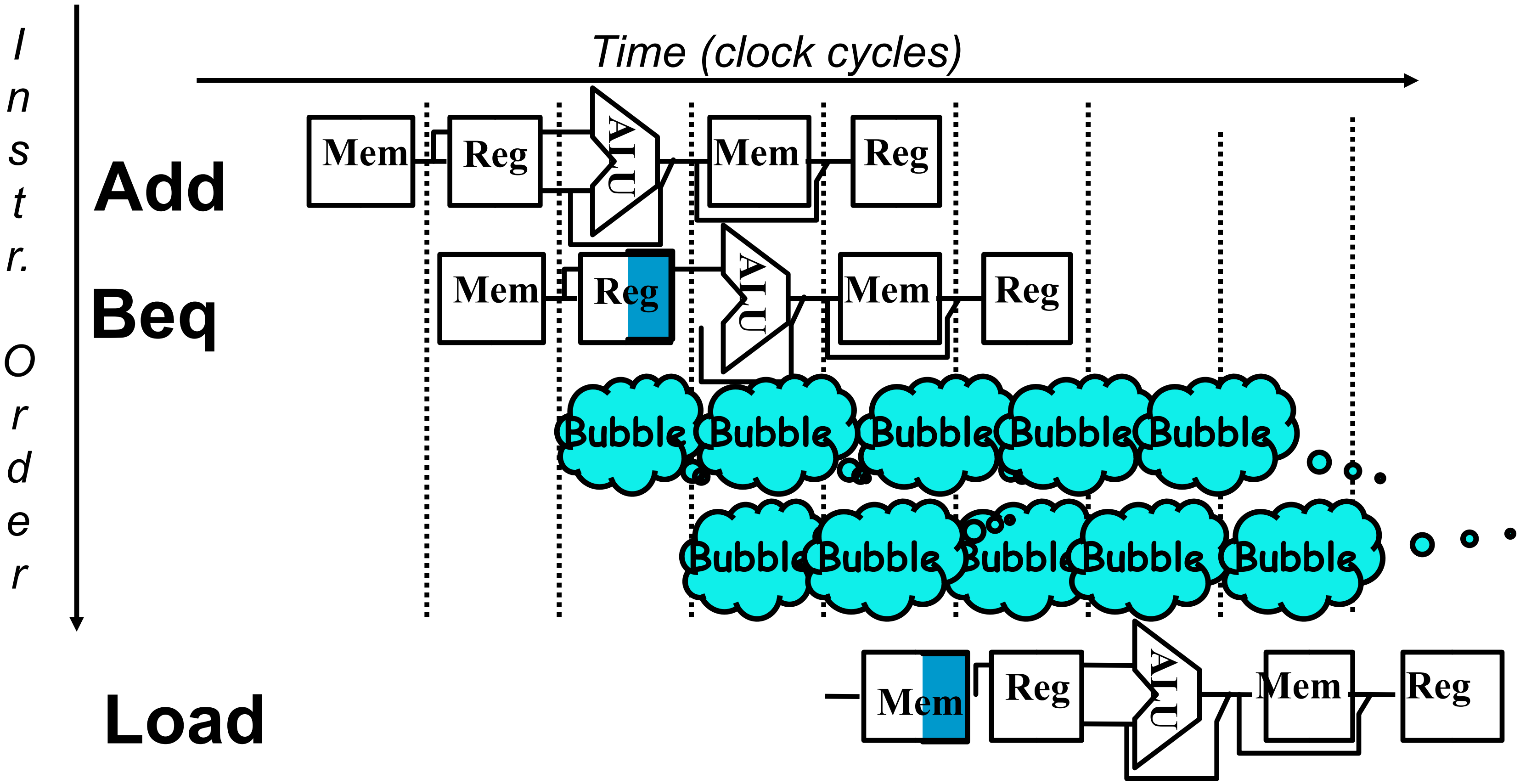
out of order execution

Control Hazard

- It arises from the need to make a decision based on the results of one instruction while others are executing.
- Two solutions to control hazards
  - Stall
    - After fetching a branch instruction, pipeline is to stall immediately, waiting until the outcome of the branch is known.
    - Slow down the pipeline.
  - Predict
    - Assumes a given outcome for the branch and proceeds from the assumption rather than waiting to ascertain the actual outcome.
    - Should rollback if the assumption is wrong,

Control Hazard Solutions (1) Stall

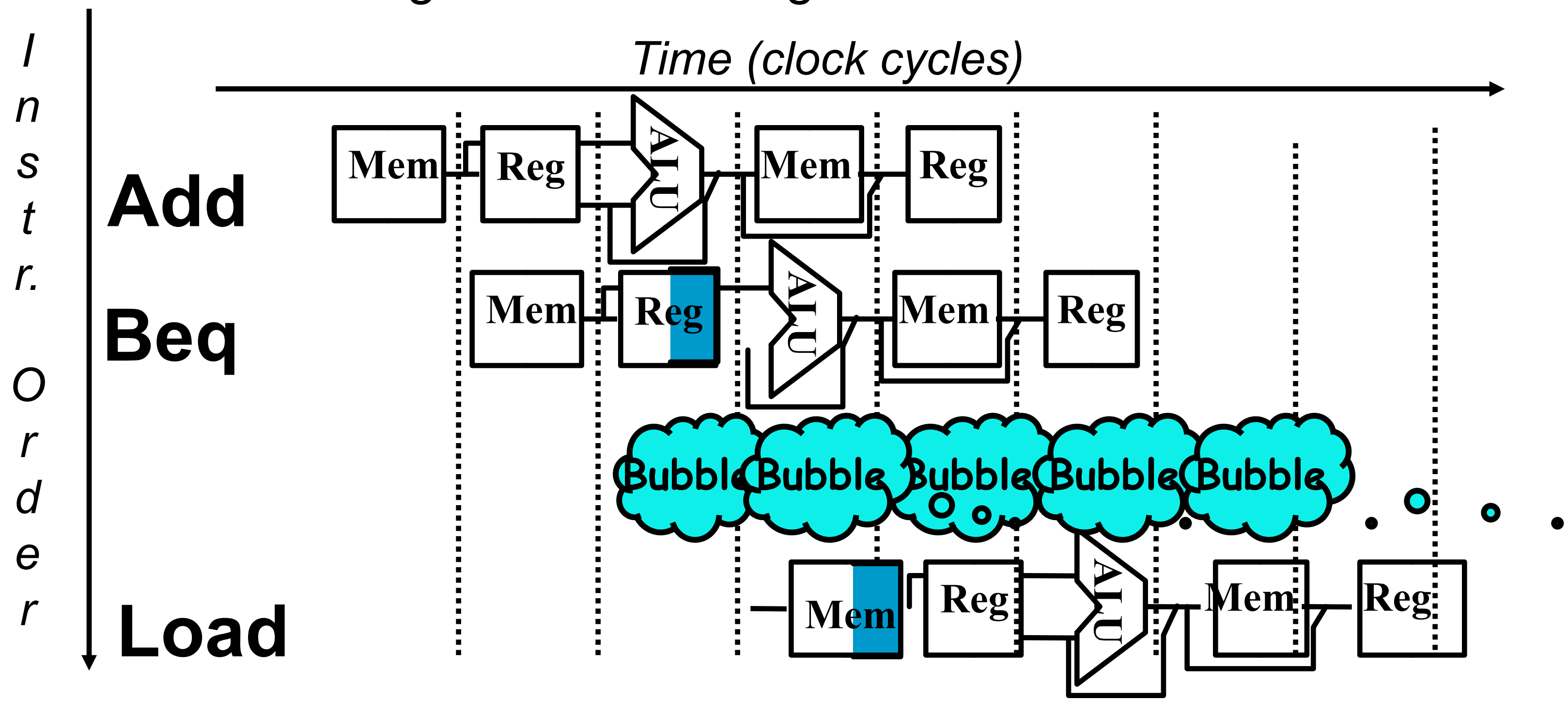
- When is a branch resolved?



- Impact: 3 clock cycles per branch instruction
  => slow

Control Hazard Solutions (2) Resolve branch earlier

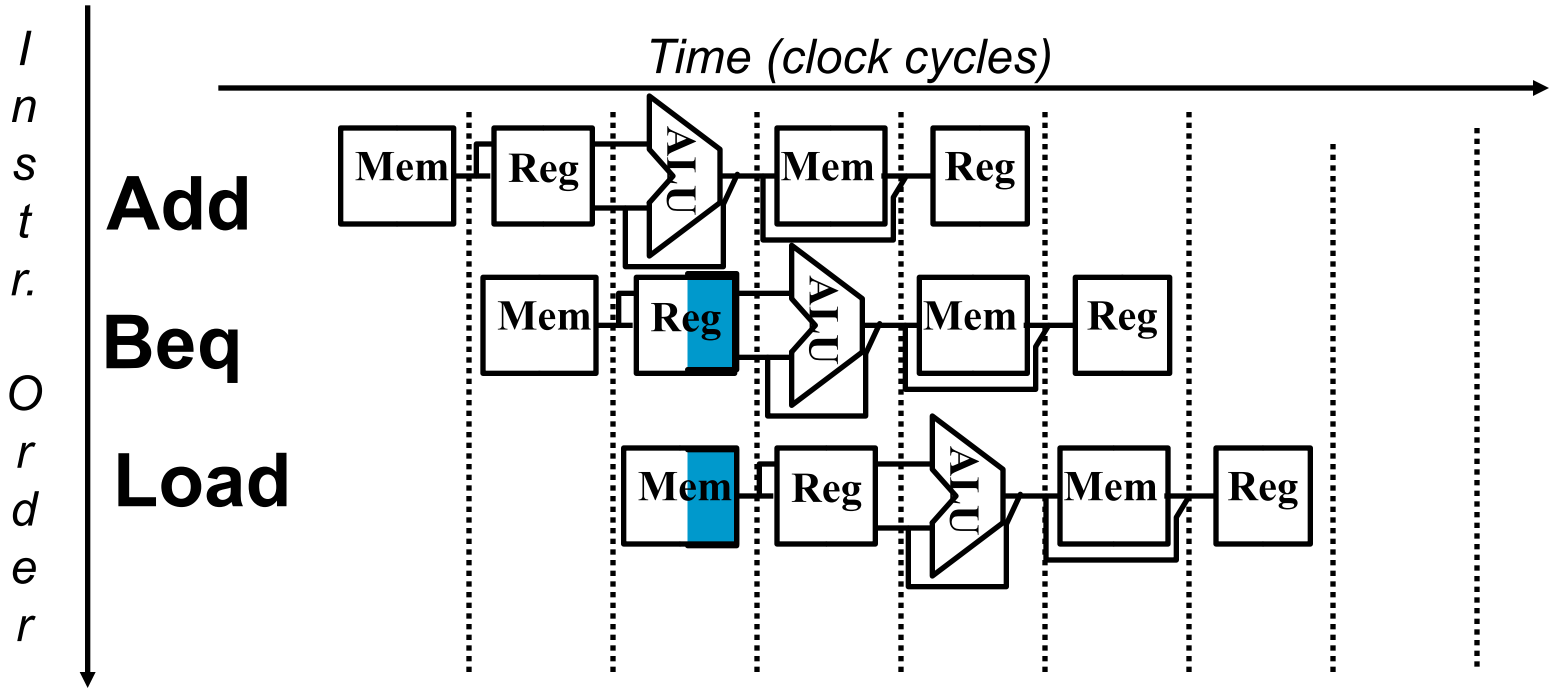
- What if a branch is resolved in the second stage (ID)?
  - Assume that we can put in enough extra hardware so that we can test registers, calculate the branch address, and update the PC during the second stage



- Impact: 2 clock cycles per branch instruction
- As the branch are 13% of the instructions executed in SPECint2000, the CPI is slowdown of 1.13 versus the ideal case.

Control Hazard Solutions (3) Predicted Untaken

- Predict: guess one direction then back up if wrong
  - Predict not taken



- Impact: 1 clock cycles per branch instruction if right, 2 if wrong (right 50% of time)
- More dynamic scheme: history of 1 branch (90%)

Predict branch not taken

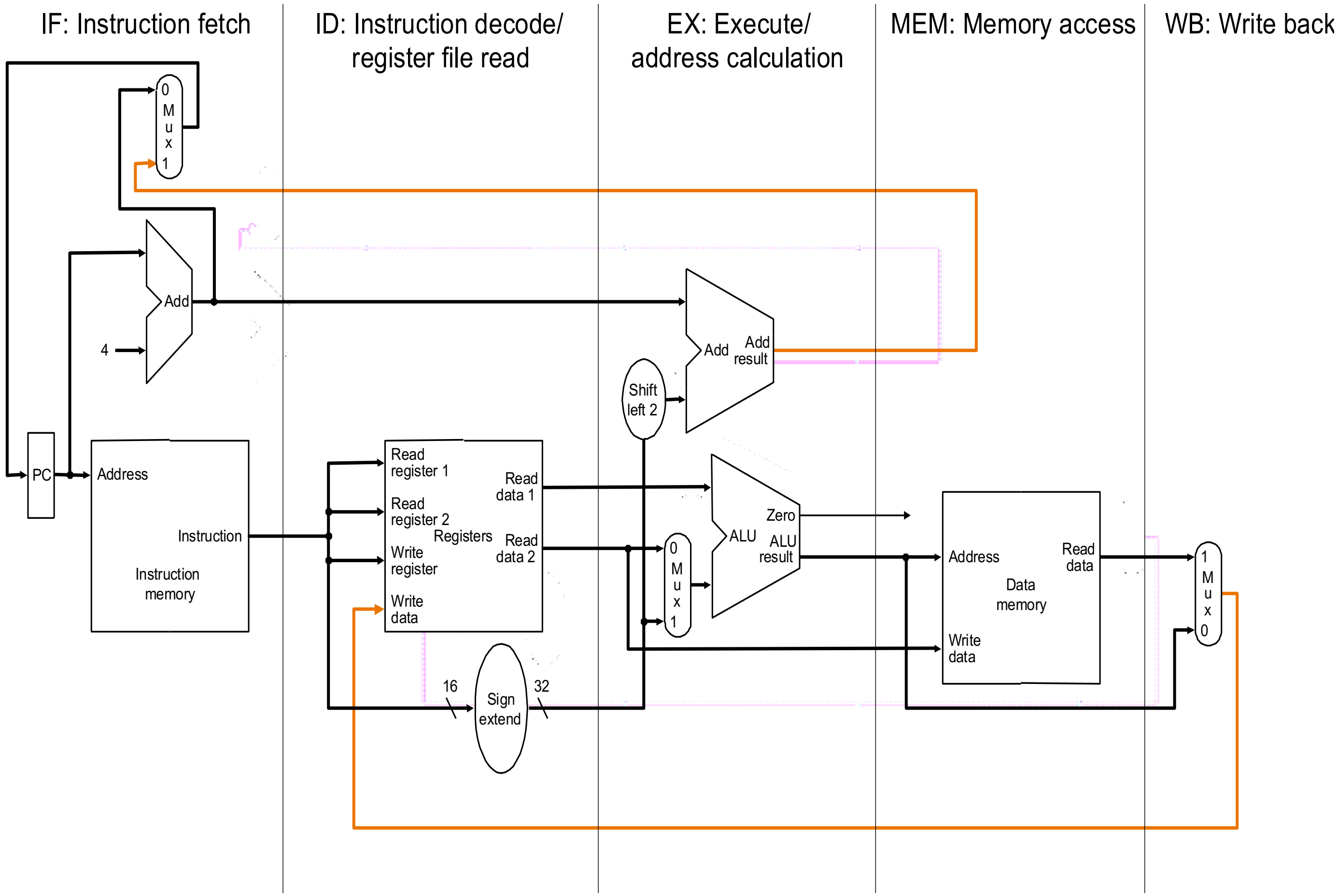
Branch Inst (i)	**IF**	**ID**	**EX**	**MEM**	**WB**			
Inst i+1		**IF**	**ID**	**EX**	**MEM**	**WB**		
Inst i+2			**IF**	**ID**	**EX**	**MEM**	**WB**	
Inst i+3				**IF**	**ID**	**EX**	**MEM**	**WB**
Inst i+4					**IF**	**ID**	**EX**	**MEM**

Correct Prediction : Zero Cycle Branch Penalty!

Branch Inst (i)	**IF**	**ID**	**EX**	**MEM**	**WB**		
Inst i+1		**IF**	**nop**	**nop**	**nop**	**nop**	
Branch target			**IF**	**ID**	**EX**	**MEM**	**WB**

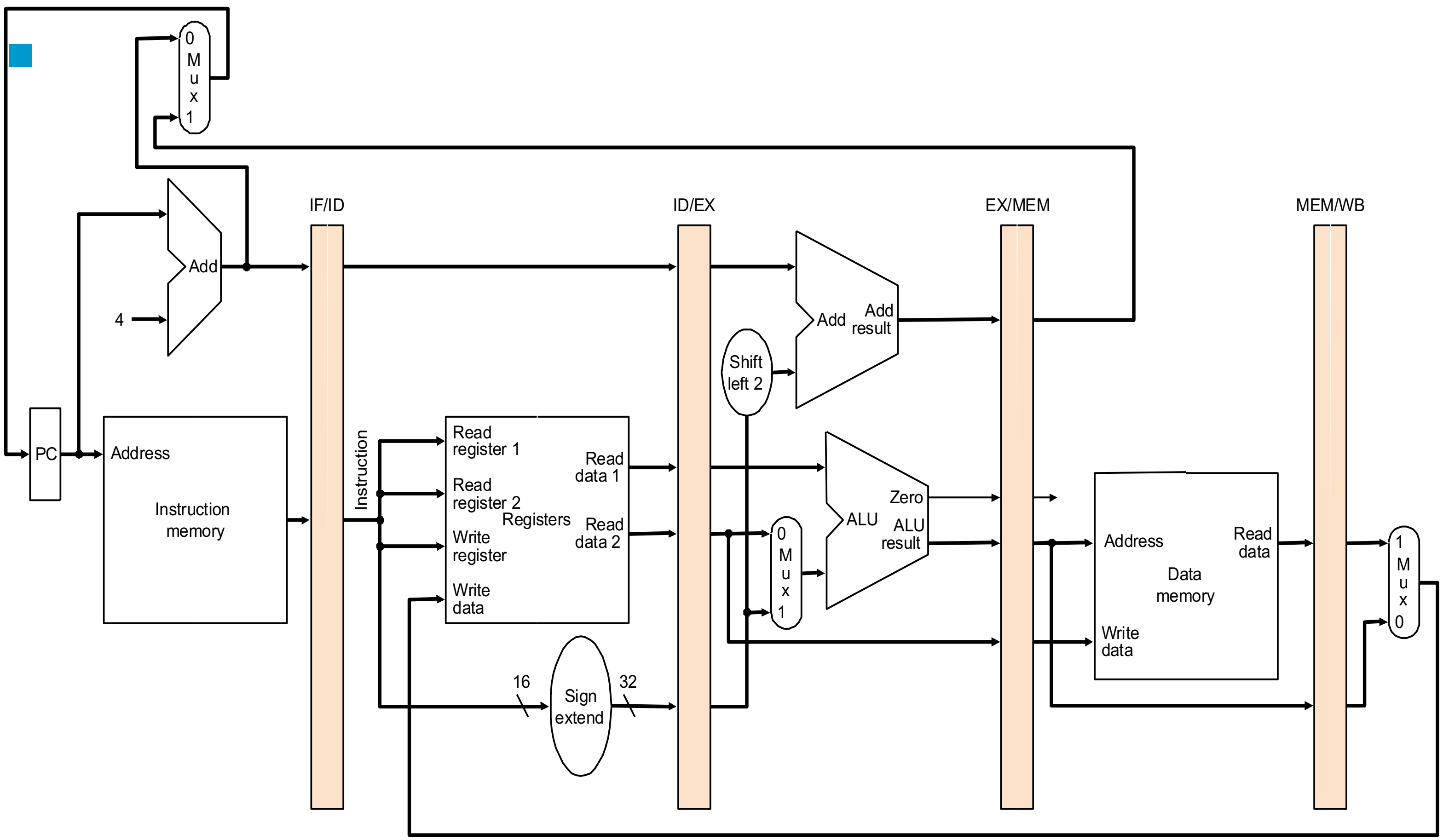
Incorrect Prediction - waste one cycle
How to flush pipeline?

A Pipelined Datapath: Basic Idea



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

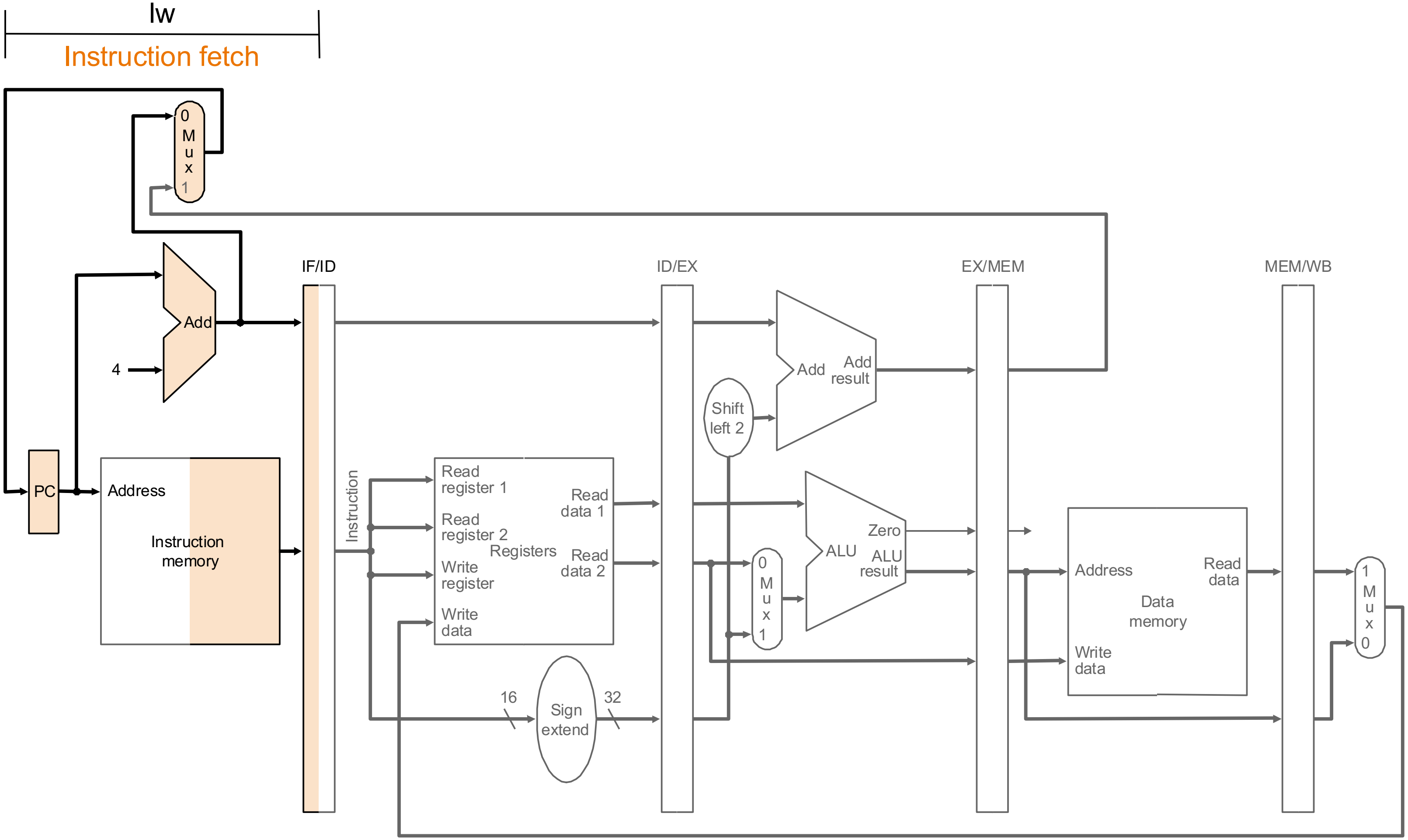
Pipelined Datapath



- Pipeline registers:
  - hold data needed for later pipeline stage
  - The registers are named for the two stages separated by that register.
- Why do we need to have separate I/D data memory? Adder?

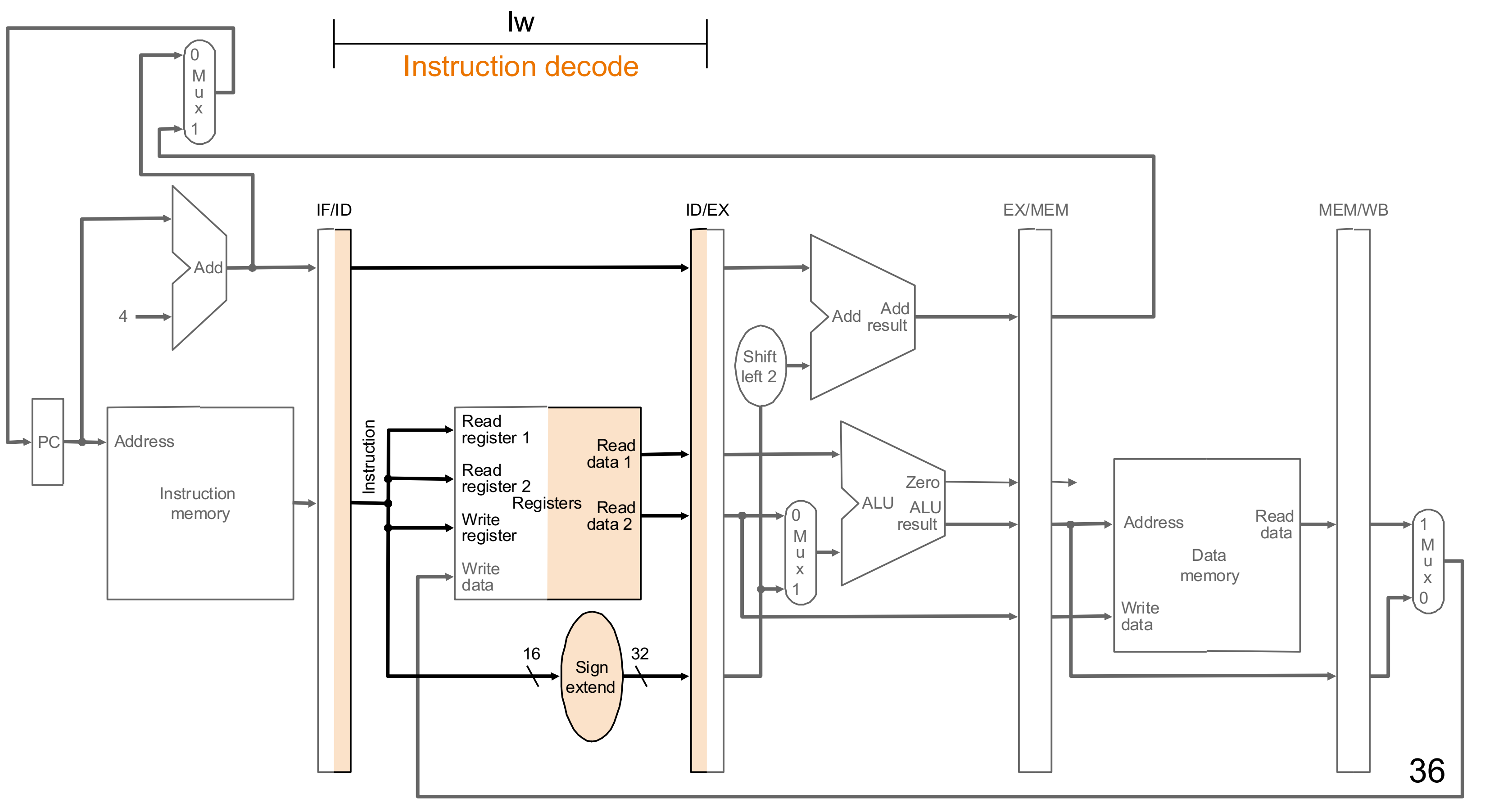
IF Stage of load

- `IF/ID.IR = mem[PC]; IF/ID.PC = PC+4 ; PC = PC +4;`



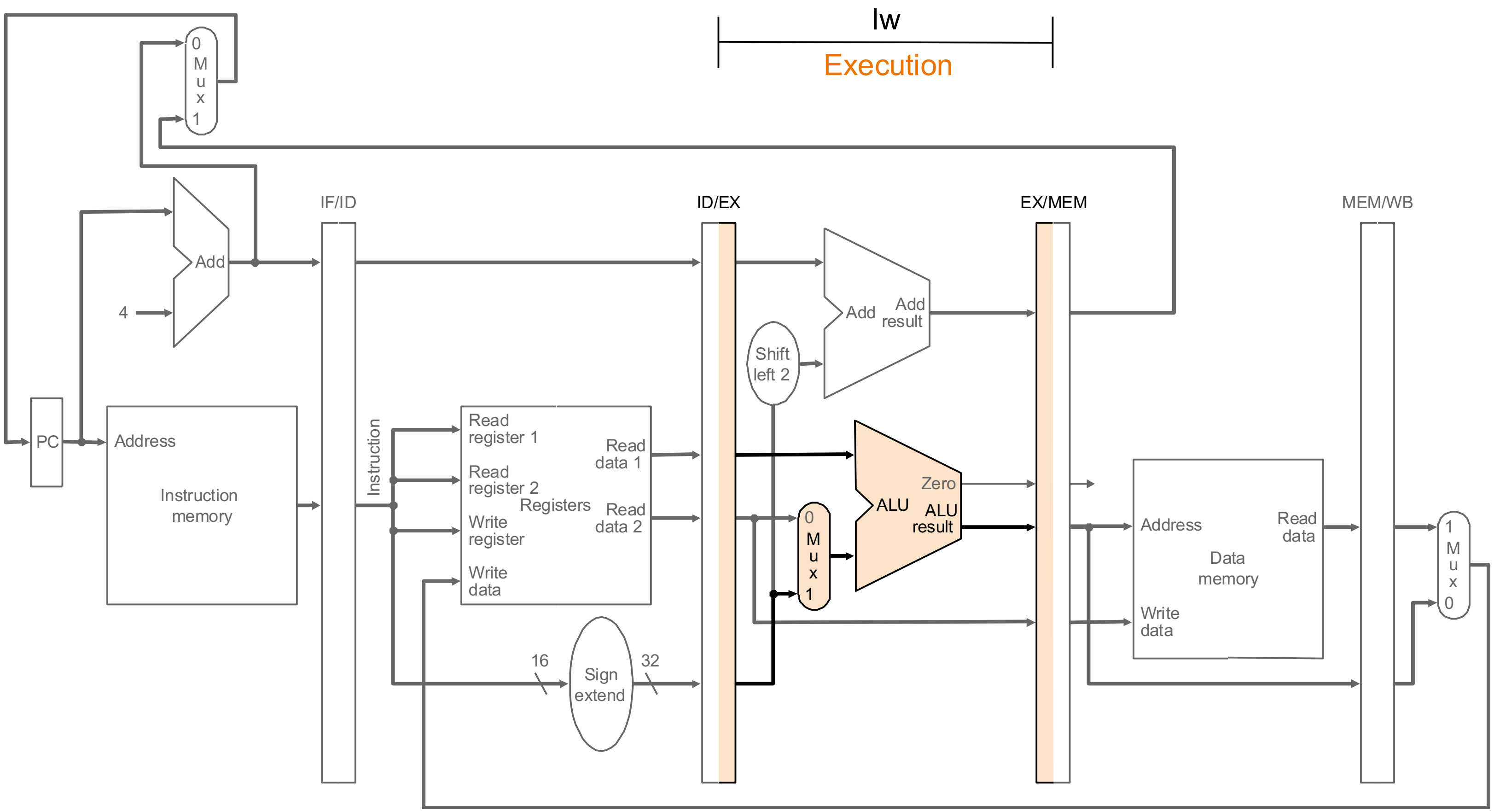
ID Stage of load

- `ID/EX.A = Reg[IF/ID.IR[25-21]]`
- `ID/EX.B = Reg[IF/ID.IR[20-16]]`
- `ID/EX.Immediate = (sign-ext(IF/ID.IR[15-0])`
- `ID/EX.PC = IF/ID.PC`



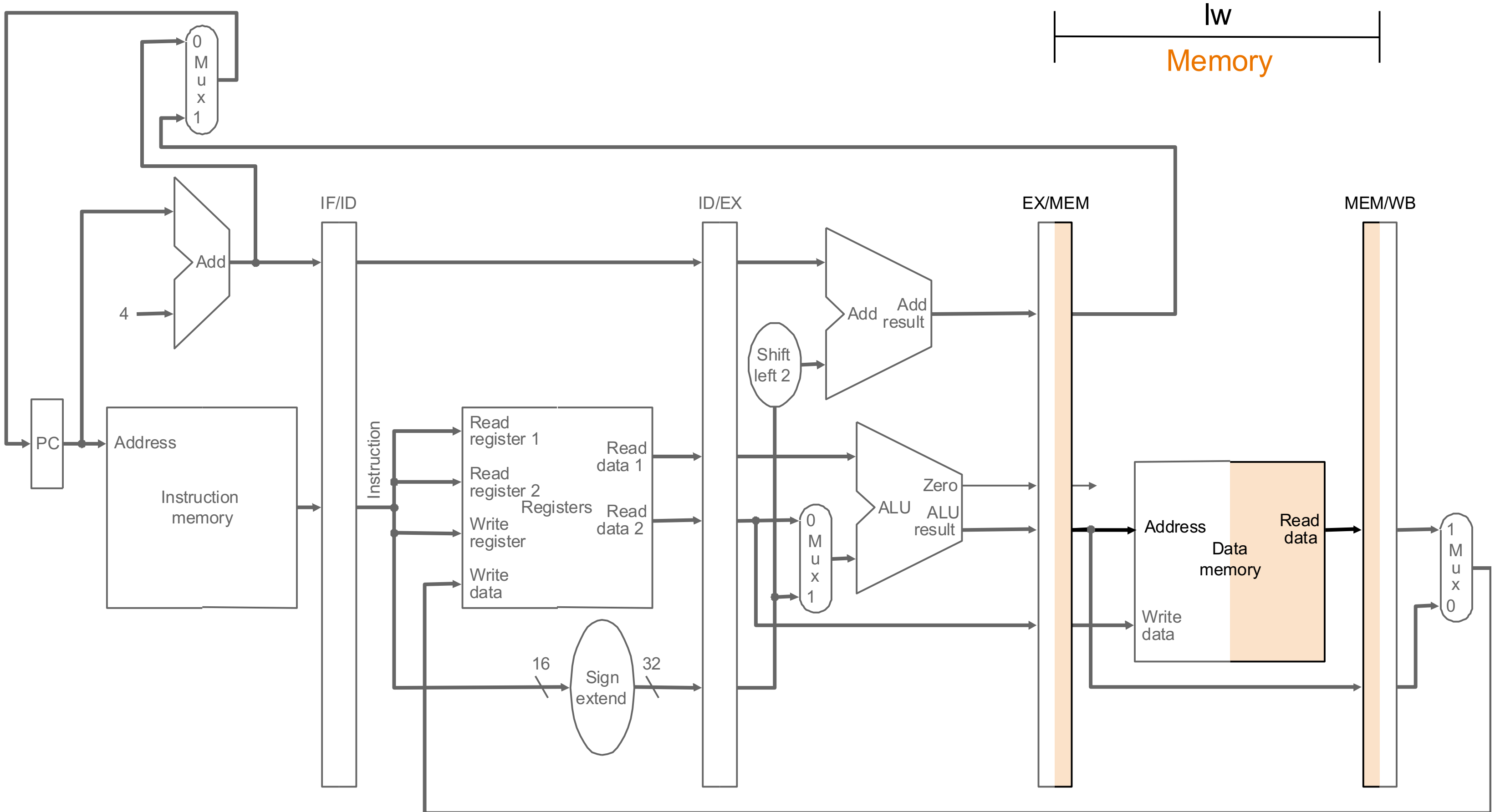
EX Stage of load

- **`EX/MEM.ALUout = ID/EX.A + ID/EX.immediate`**



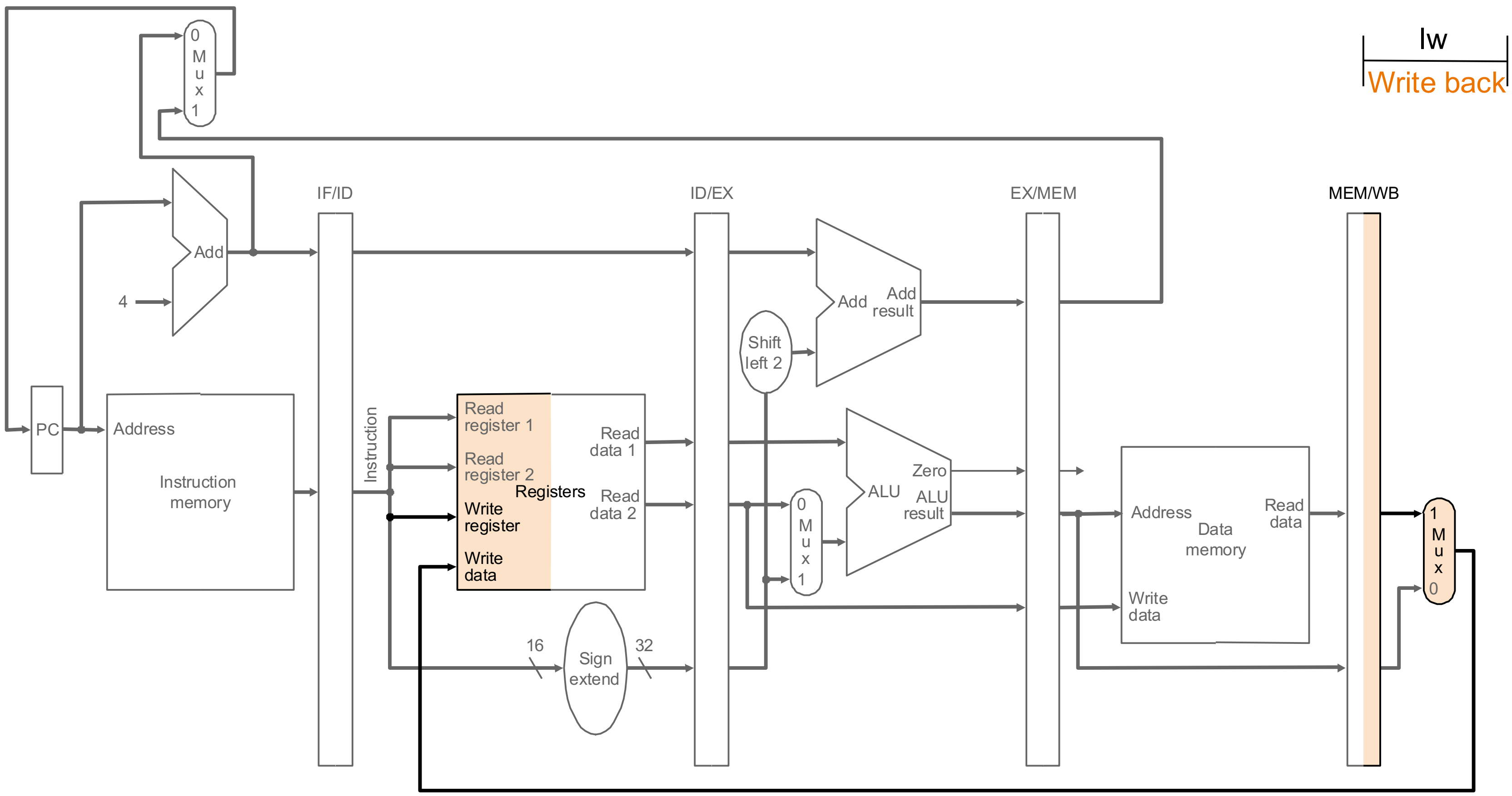
MEM Stage of load

- `MEM/WB.MDR = mem[EX/MEM.ALUout]`



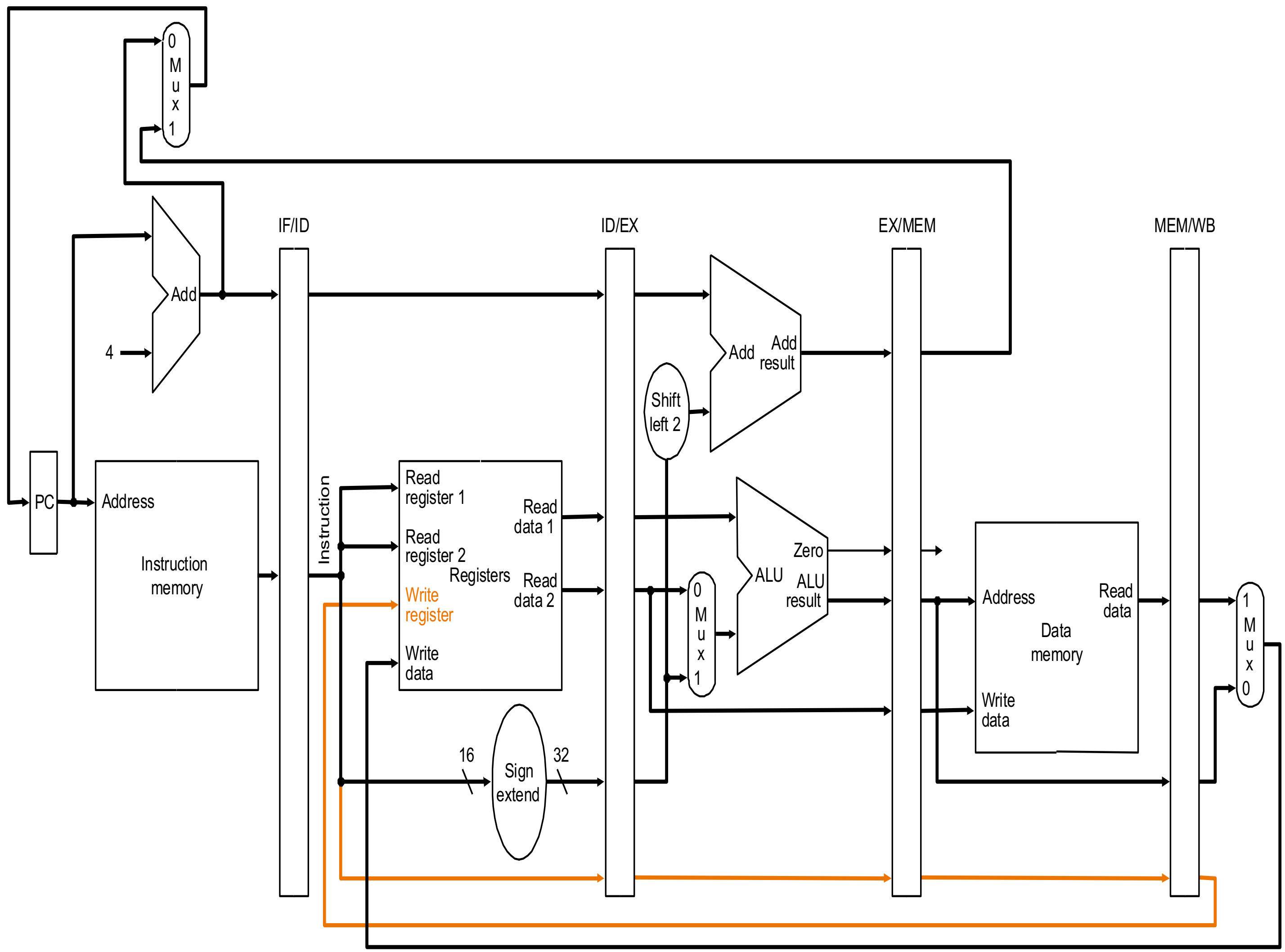
WB Stage of load

- `Reg[Rt] = MEM/WB.MDR`
  - `Where does Rt come from?`



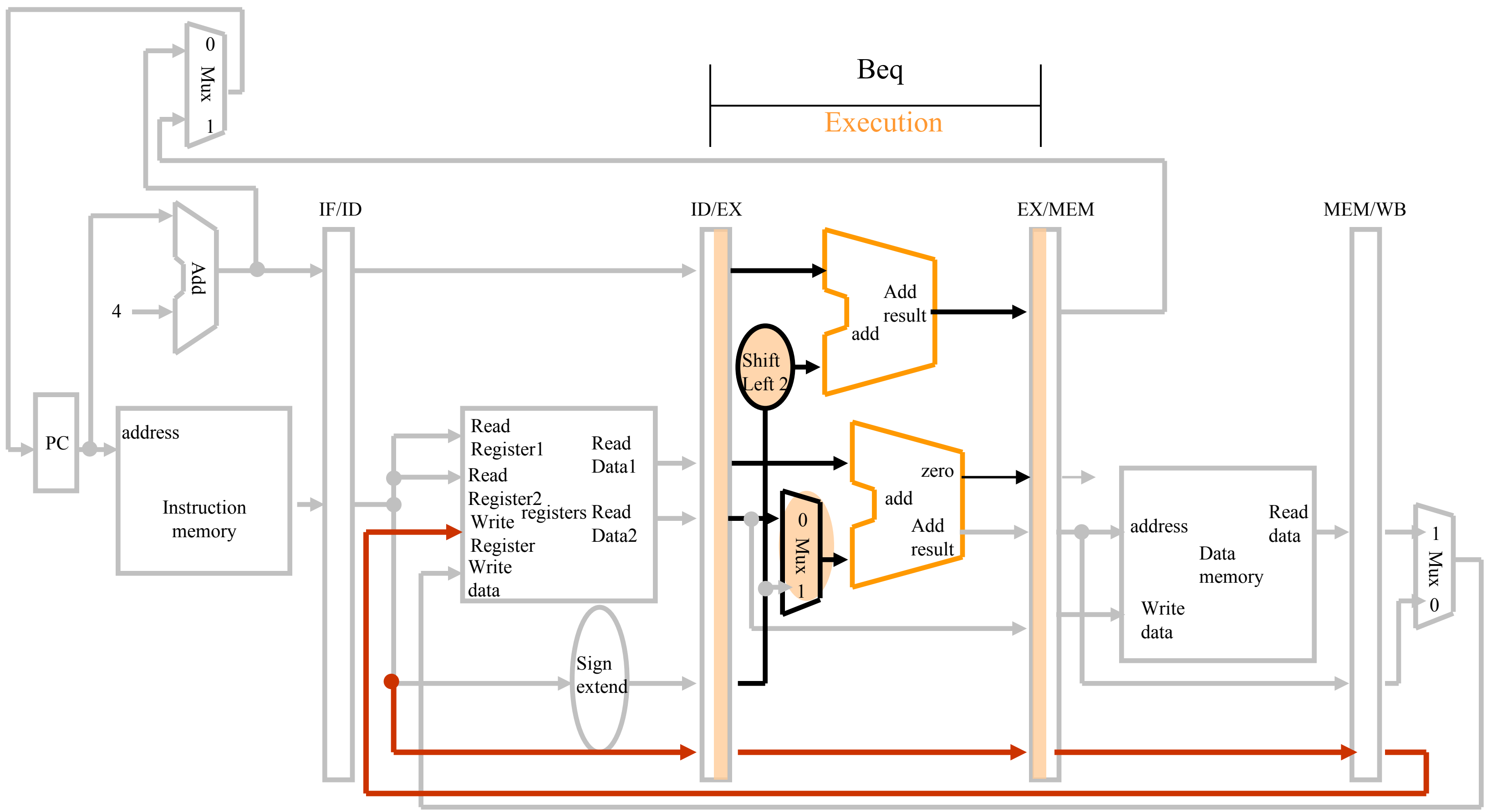
- Load must pass the register number from ID/EX through EX/MEM to the MEM/WB pipeline register for use in the WB stage

Corrected Datapath



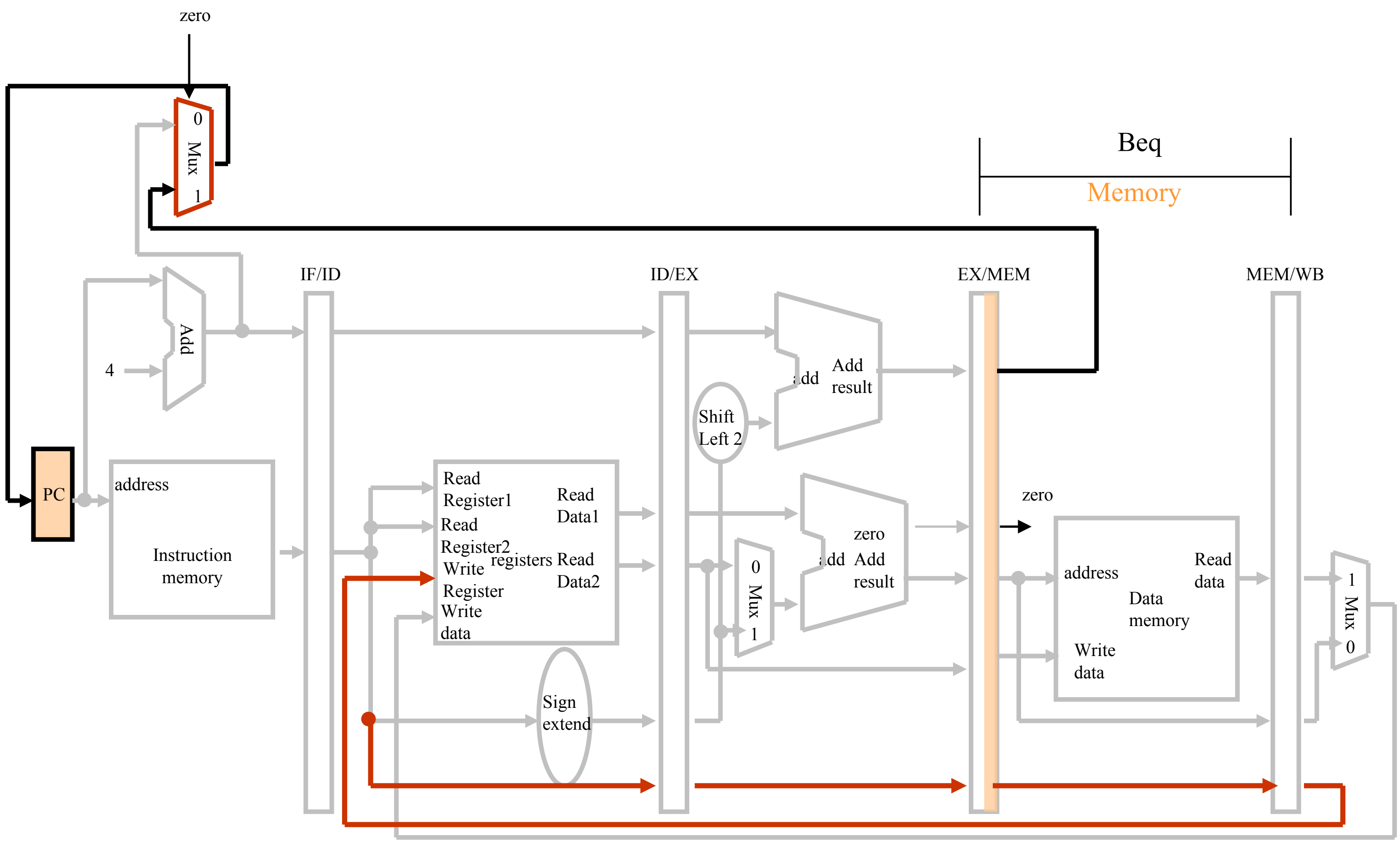
EX Stage of beq

- `EX/MEM.PC = ID/EX.PC + (ID/EX.immediate << 2);`
- `EX/MEM.Zero = (ID/EX.A – ID/EX.B) ? 0 : 1`

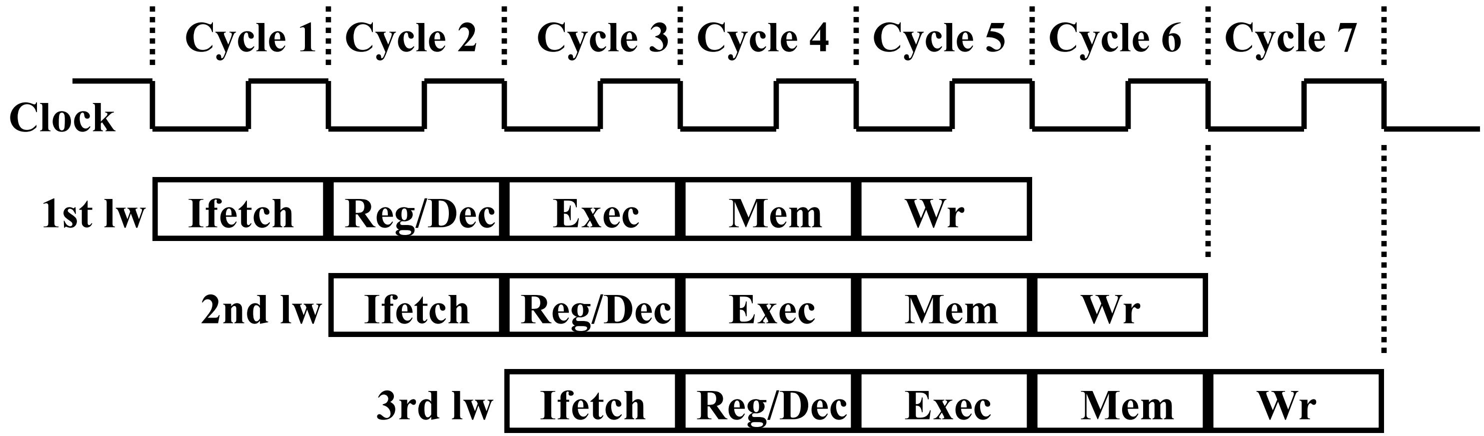


MEM Stage of beq

- **`If (EX/MEM.zero) PC = EX/MEM.PC`**

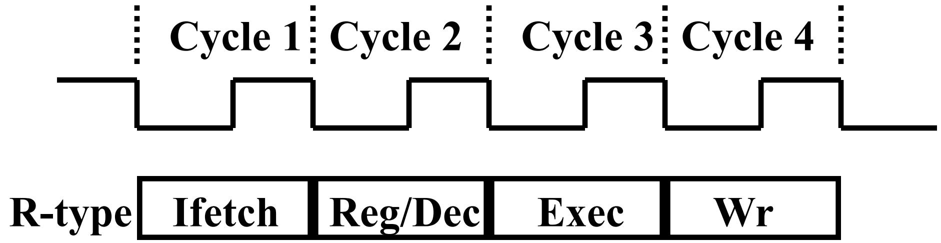


Pipelining the Load Instruction



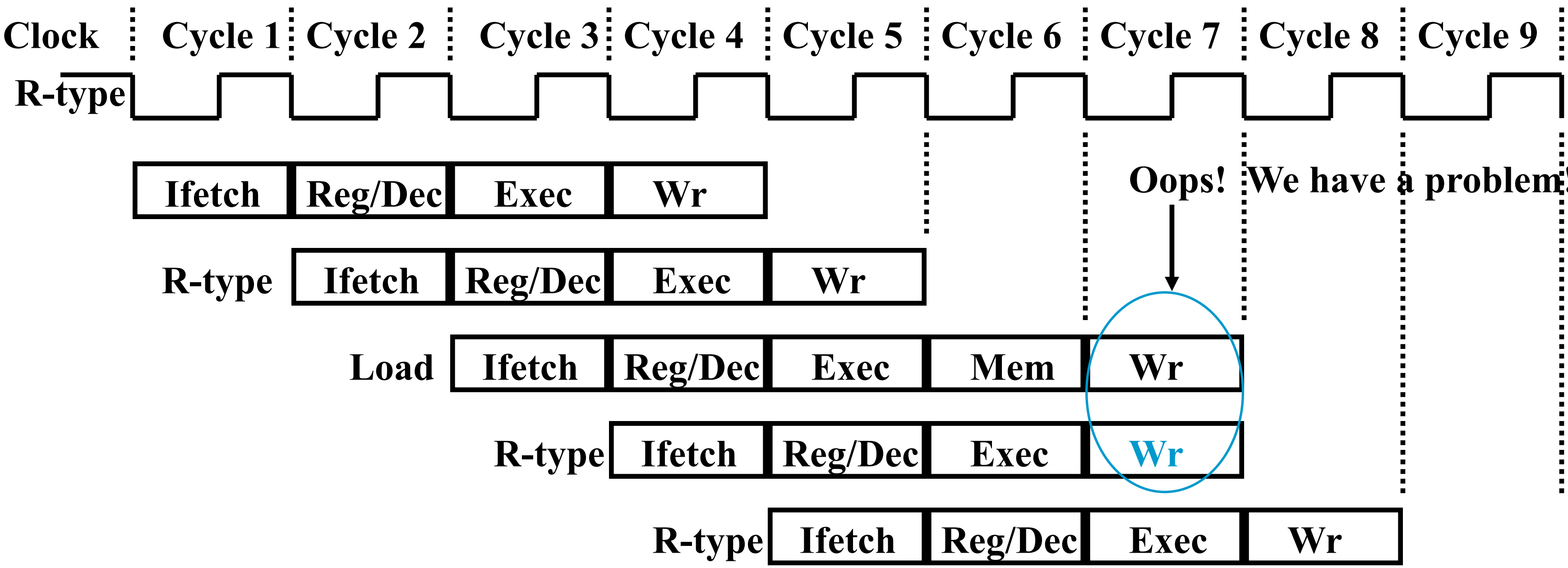
- The five independent functional units in the pipeline datapath are:
  - Instruction Memory for the Ifetch stage
  - Register File's Read ports (bus A and busB) for the Reg/Dec stage
  - ALU for the Exec stage
  - Data Memory for the Mem stage
  - Register File's Write port (bus W) for the Wr stage

The Four Stages of R-type



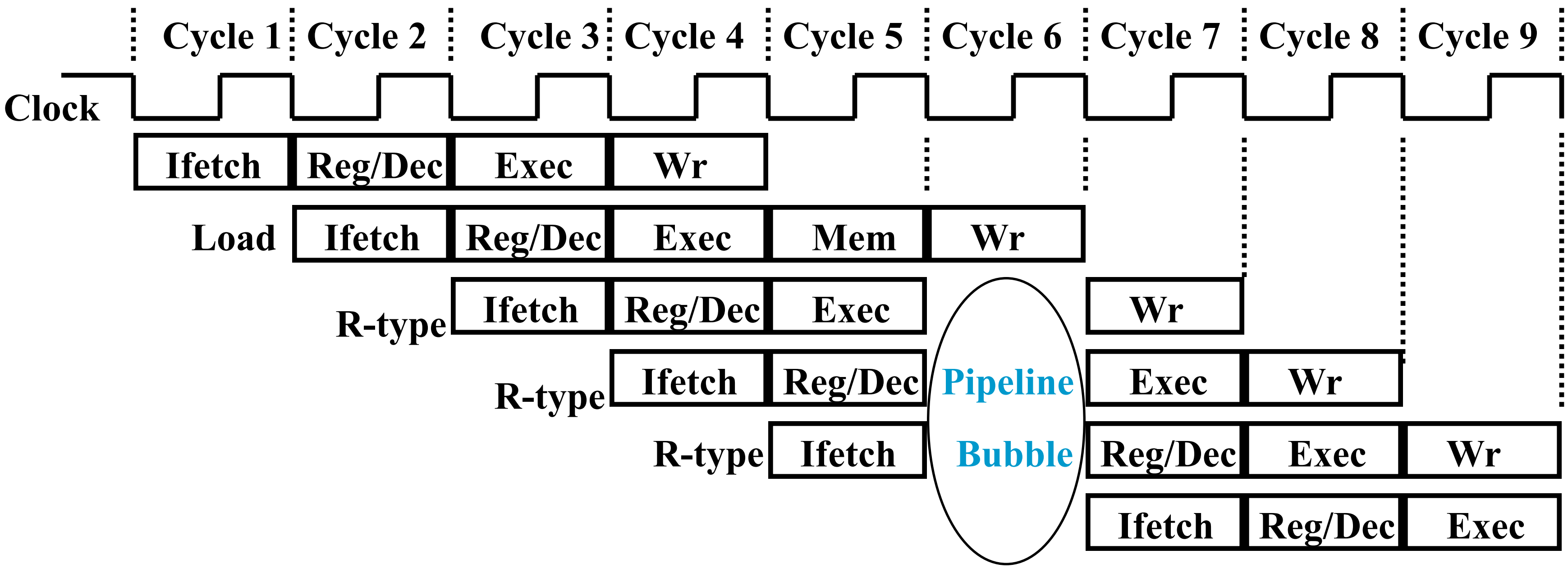
- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec:
  - ALU operates on the two register operands
- Wr: Write the ALU output back to the register file

Pipelining the R-type and Load Instruction



- We have pipelined conflict or structural hazard:
  - Two instructions try to write to the register file at the same time!
  - Only one write port

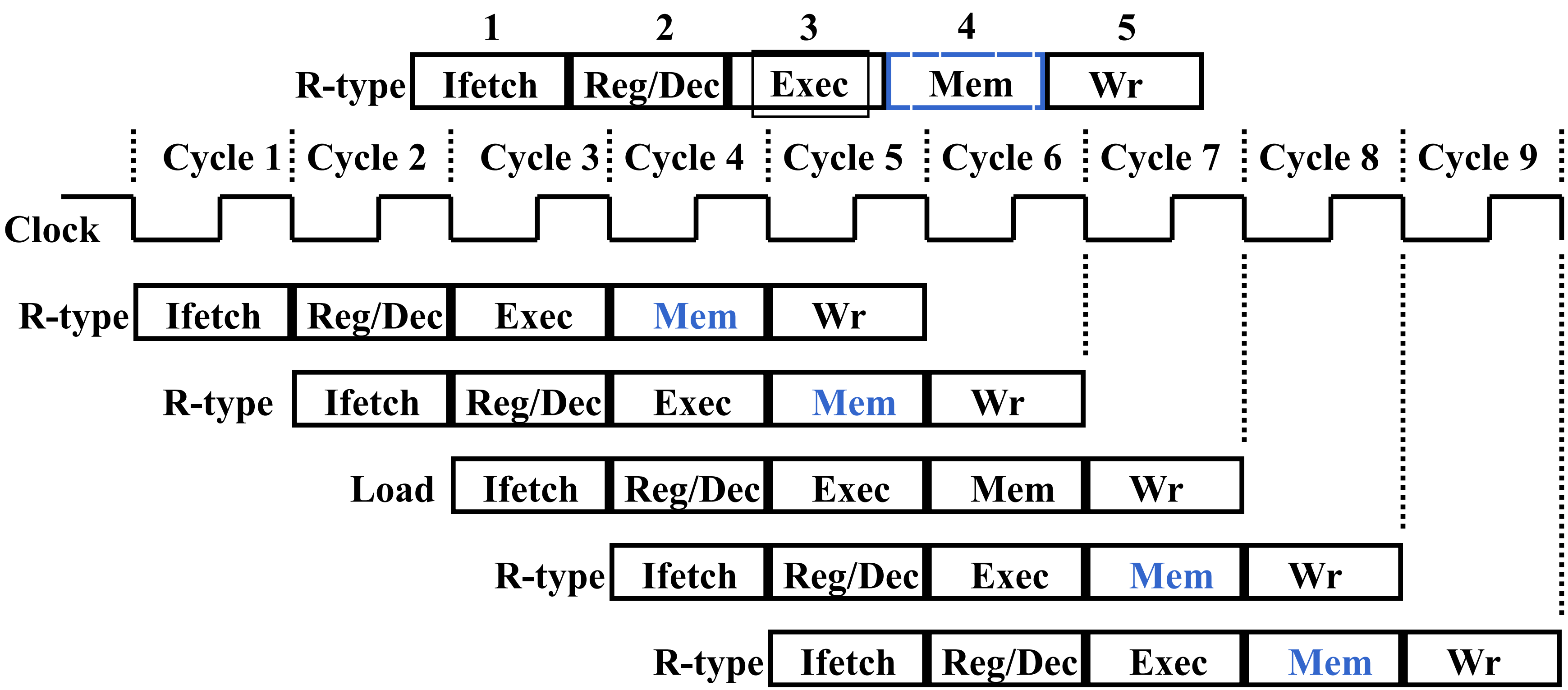
Solution 1: Insert "Bubble" into the Pipeline



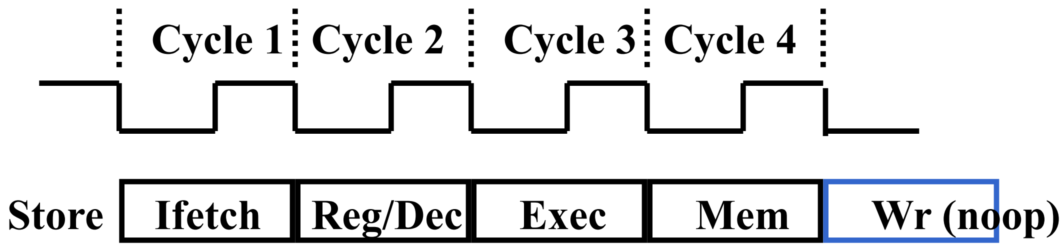
- Insert a "bubble" into the pipeline to prevent 2 writes at the same cycle
  - The control logic can be complex.
  - Lose instruction fetch and issue opportunity.
    - No instruction is started in Cycle 6!

Solution 2: Delay R-type's Write by One Cycle

- Delay R-type's register write by one cycle:
  - Now R-type instructions also use Reg File's write port at Stage 5
  - Mem stage is a NOOP stage: nothing is being done.

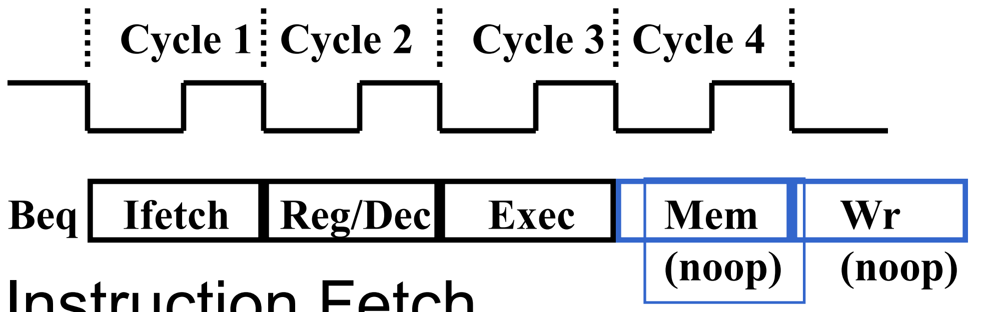


The Four Stages of Store



- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Write the data into the Data Memory

The Three Stages of Beq



- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec:
  - Registers Fetch and Instruction Decode
- Exec:
  - compares the two register operand,
  - select correct branch target address
  - latch into PC

lw $10, 20($1)

Instruction fetch

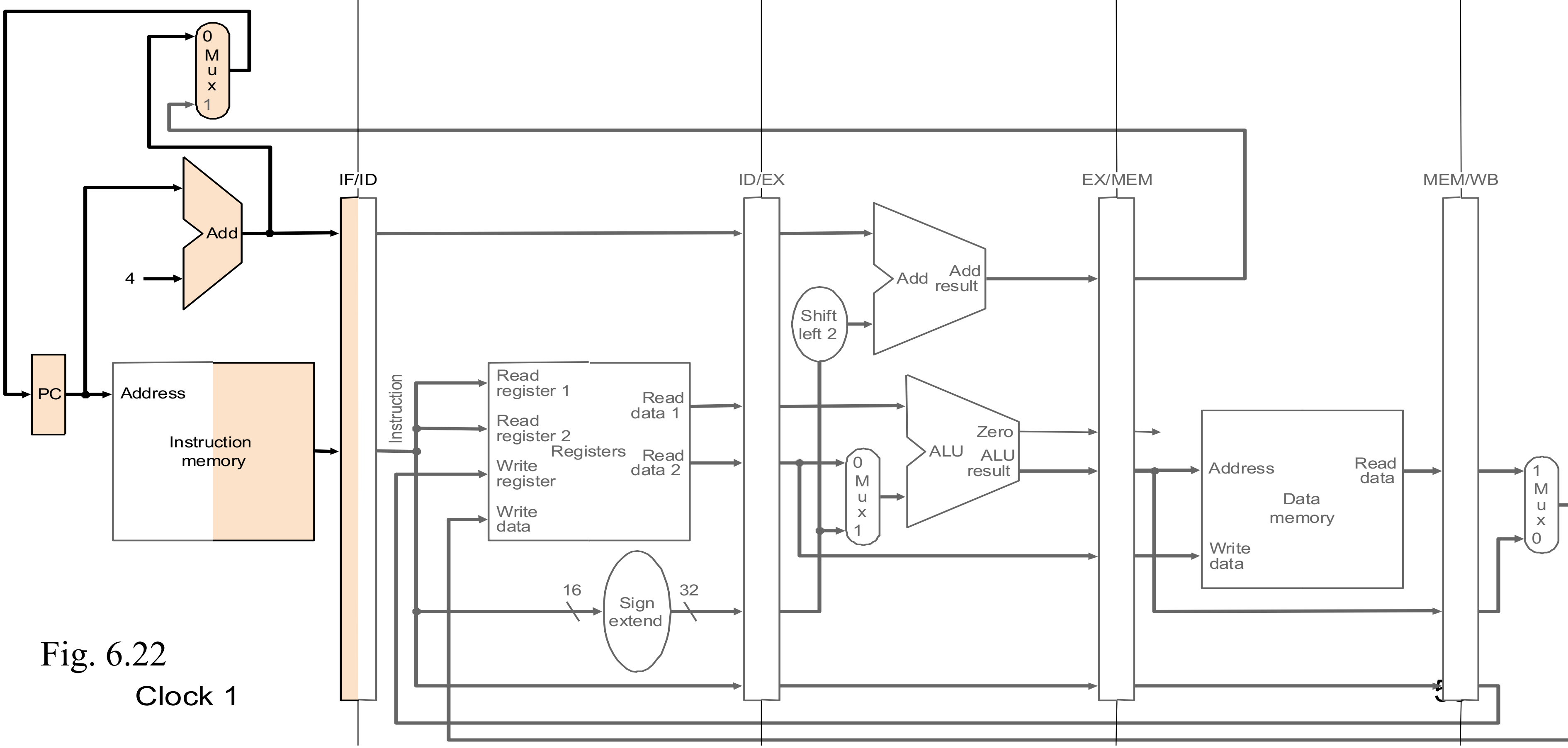


Fig. 6.22

Clock 1

sub $11, $2, $3

Instruction fetch

lw $10, 20($1)

Instruction decode

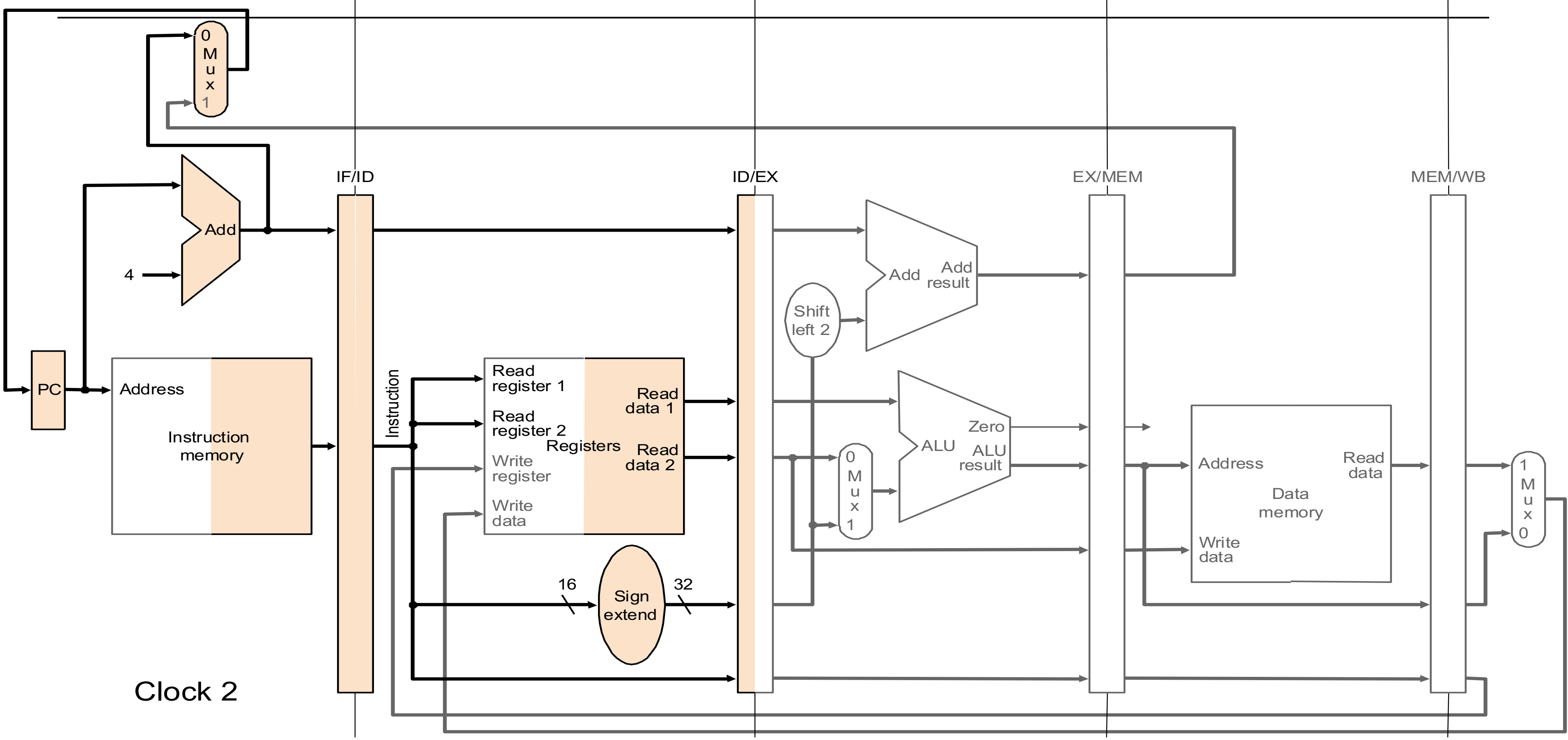


Fig. 6.22

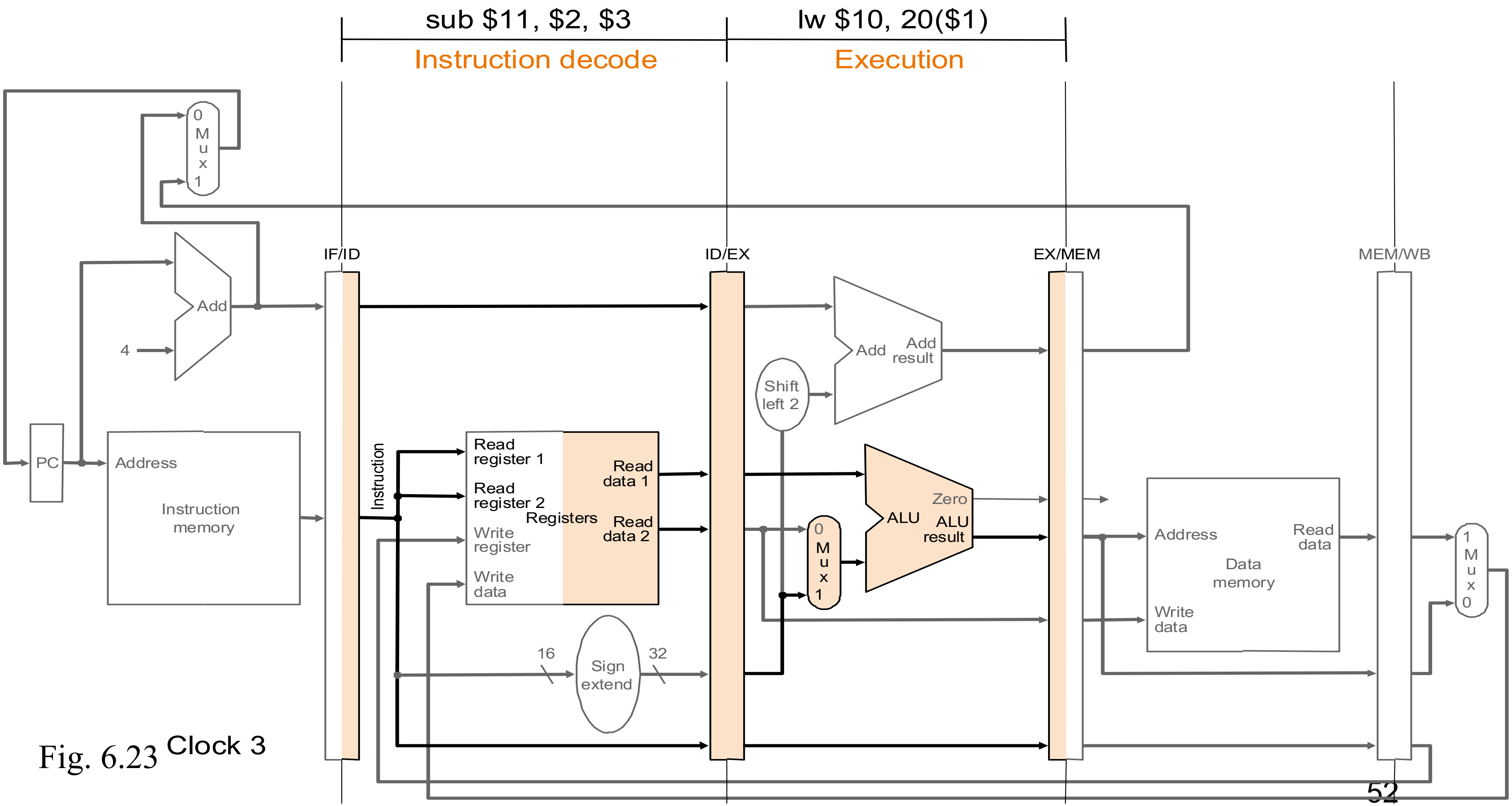
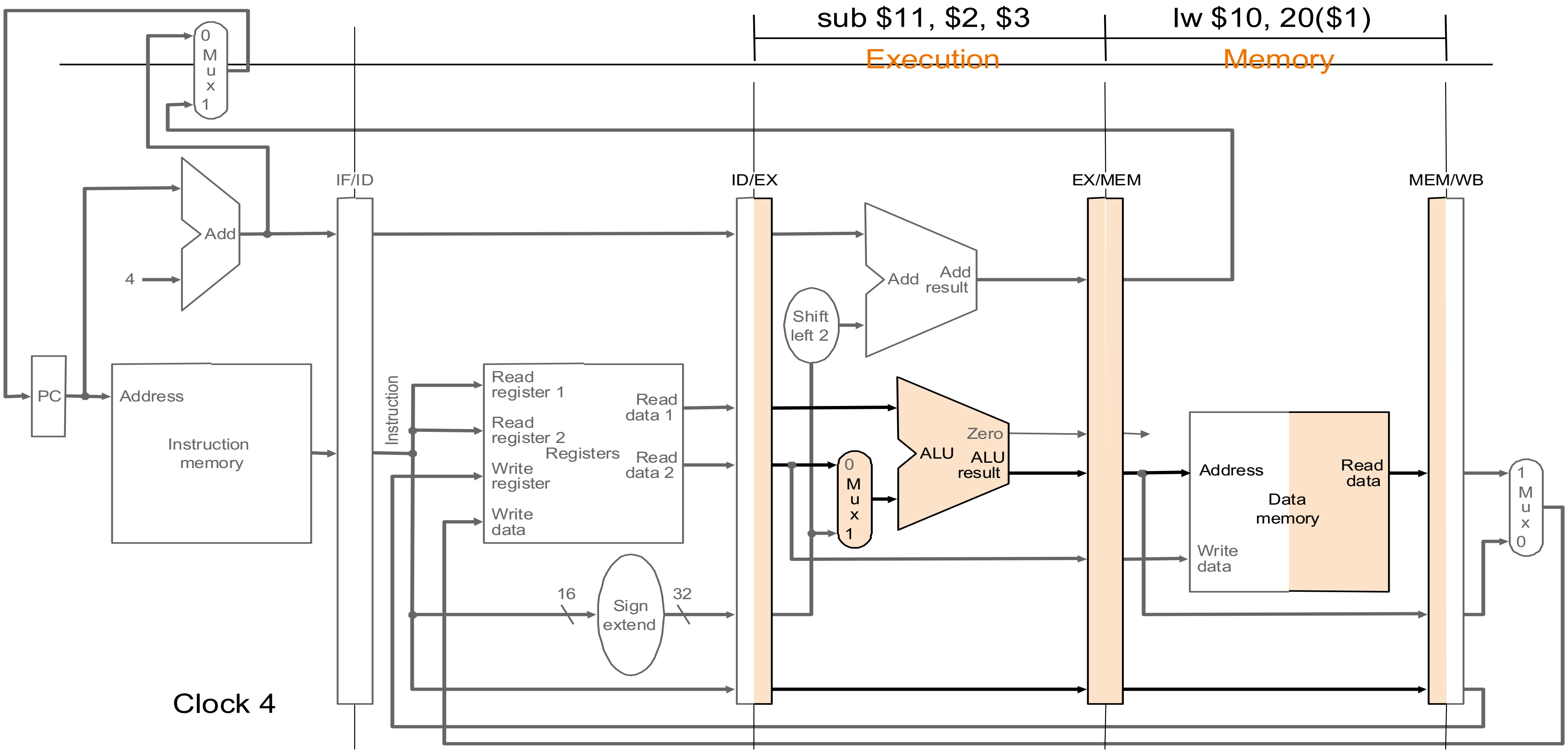


Fig. 6.23 Clock 3



sub $11, $2, $3
lw $10, 20($1)
Execution
Memory
0
M
u
x
1
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
Add
Add result
Shift left 2
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Zero
ALU
ALU result
Address
Data memory
Write data
Read data
16
Sign extend
32
Clock 4

Fig. 6.23

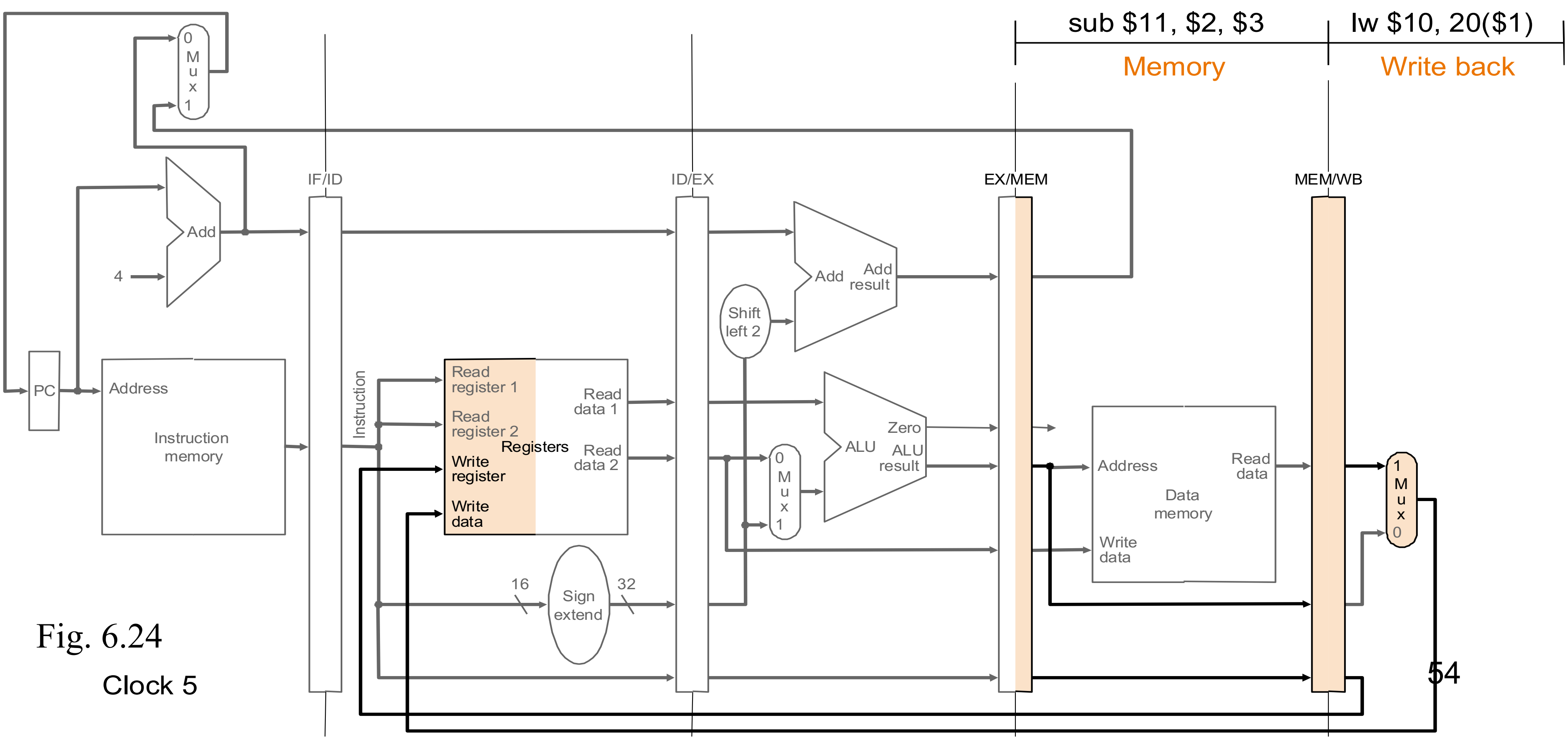
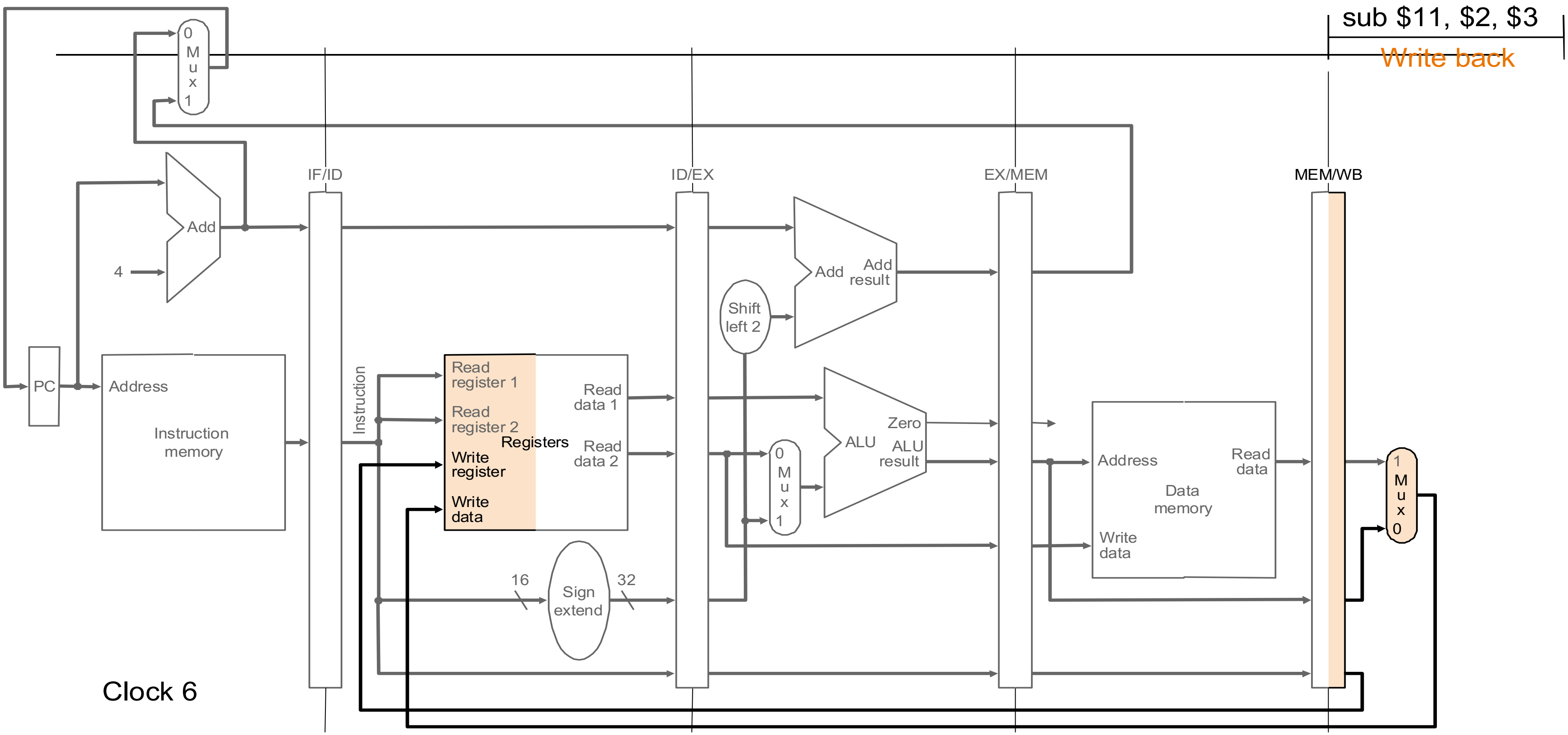


Fig. 6.24

Clock 5



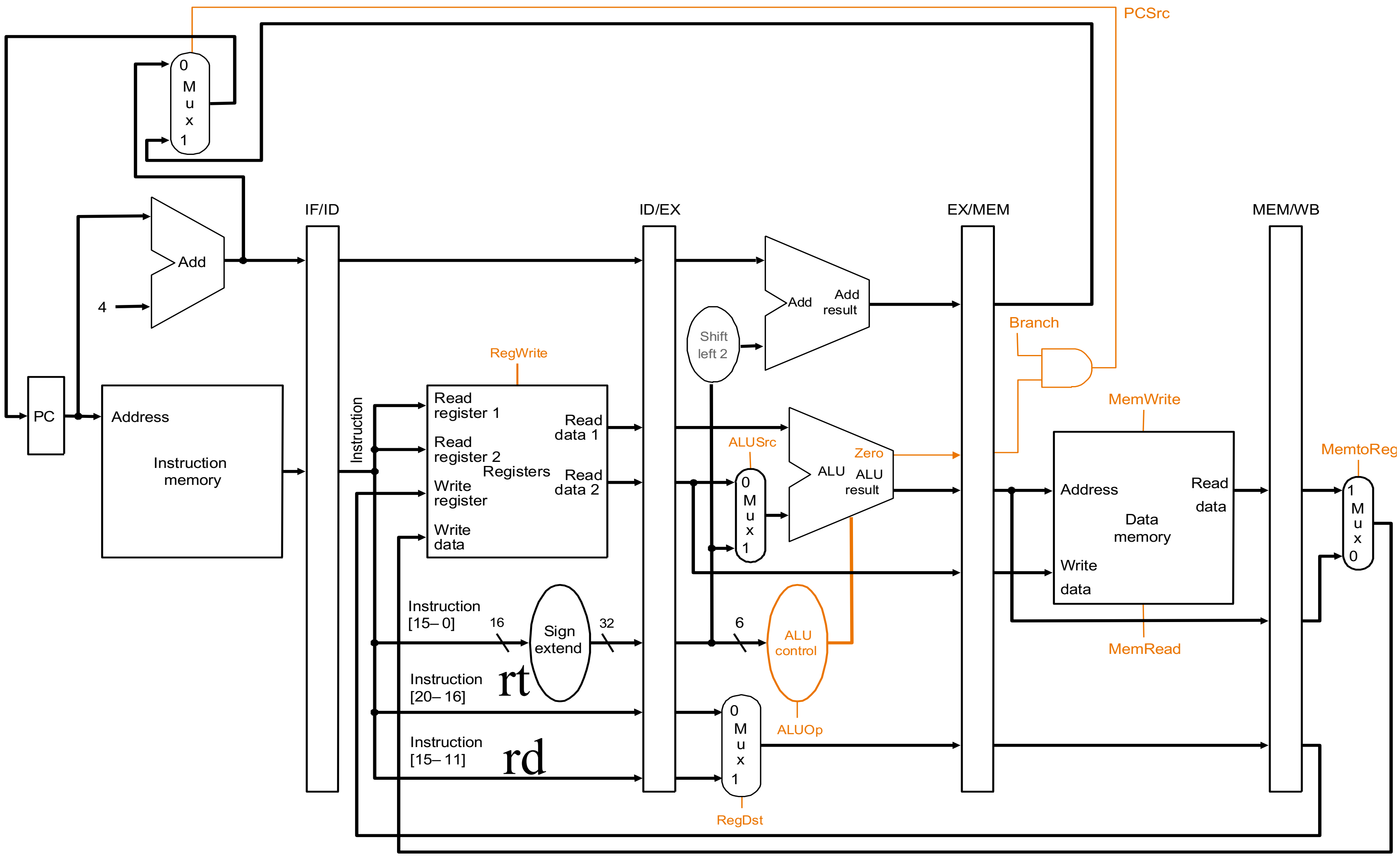
sub $11, $2, $3
Write back
IF/ID
ID/EX
EX/MEM
MEM/WB
0
M
u
x
1
Add
4
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Data memory
Read data
Write data
Clock 6

Fig. 6.24

Pipeline control

- We have 5 stages. What needs to be controlled in each stage?
  - Instruction Fetch and PC Increment
  - Instruction Decode / Register Fetch
  - Execution
  - Memory Stage
  - Write Back

Pipelined Datapath with Control Signals Identified



1. Program counter is updated in the mem stage for beq
2. Do we need PCwrite control signal?

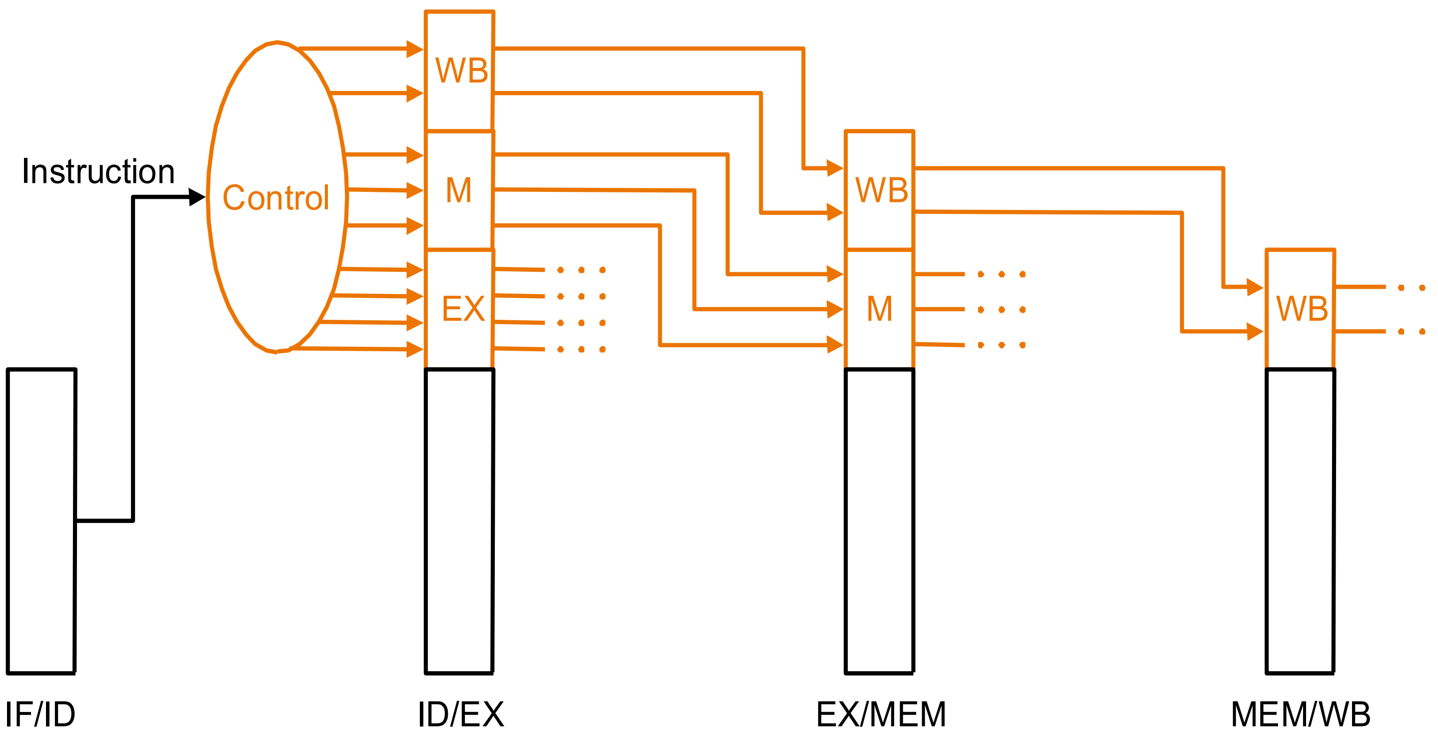
Group Signals According to Stages

- Can use control signals of single-cycle CPU (Fig. 6.23, 6.24, 6.25 <==> 5.12, 5.16, 5.18)

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

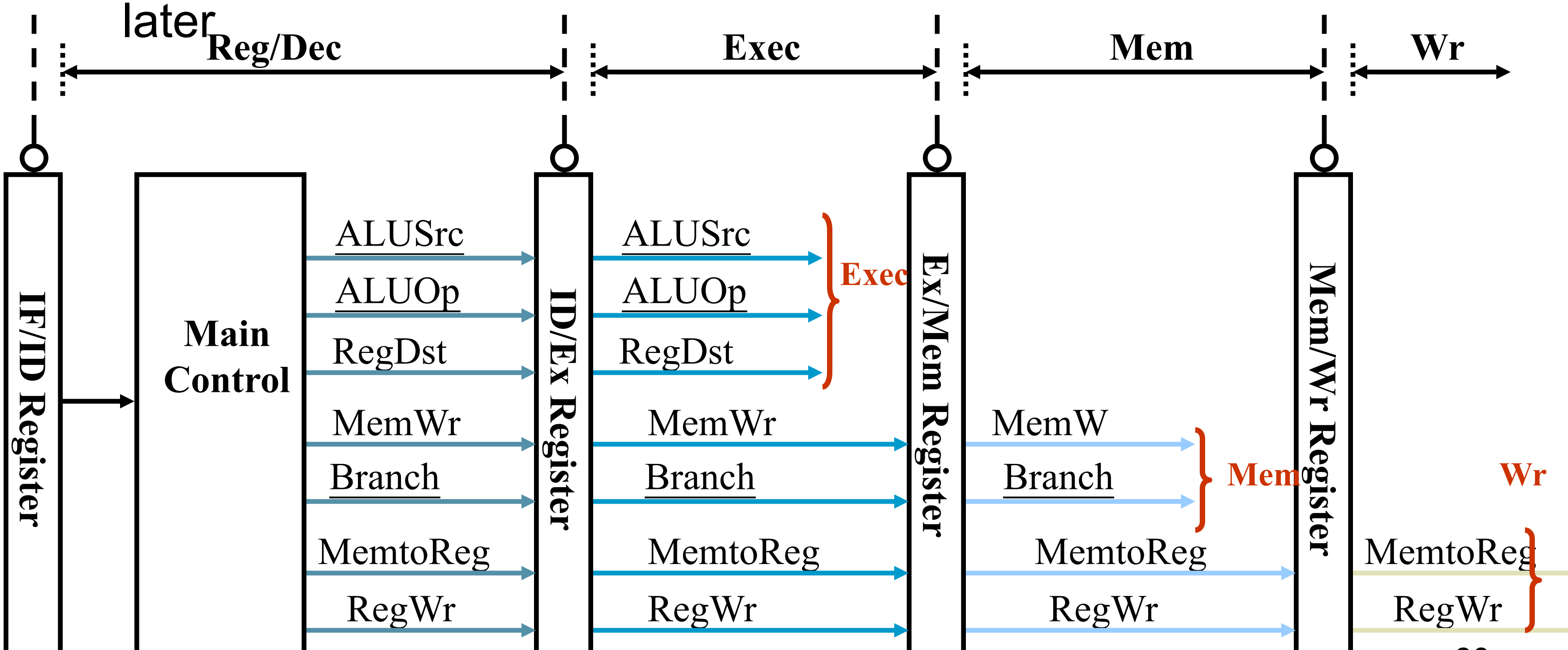
Control Lines for the Final Three Stages

- Pass control signals along just like the data
  - Main control generates control signals during ID

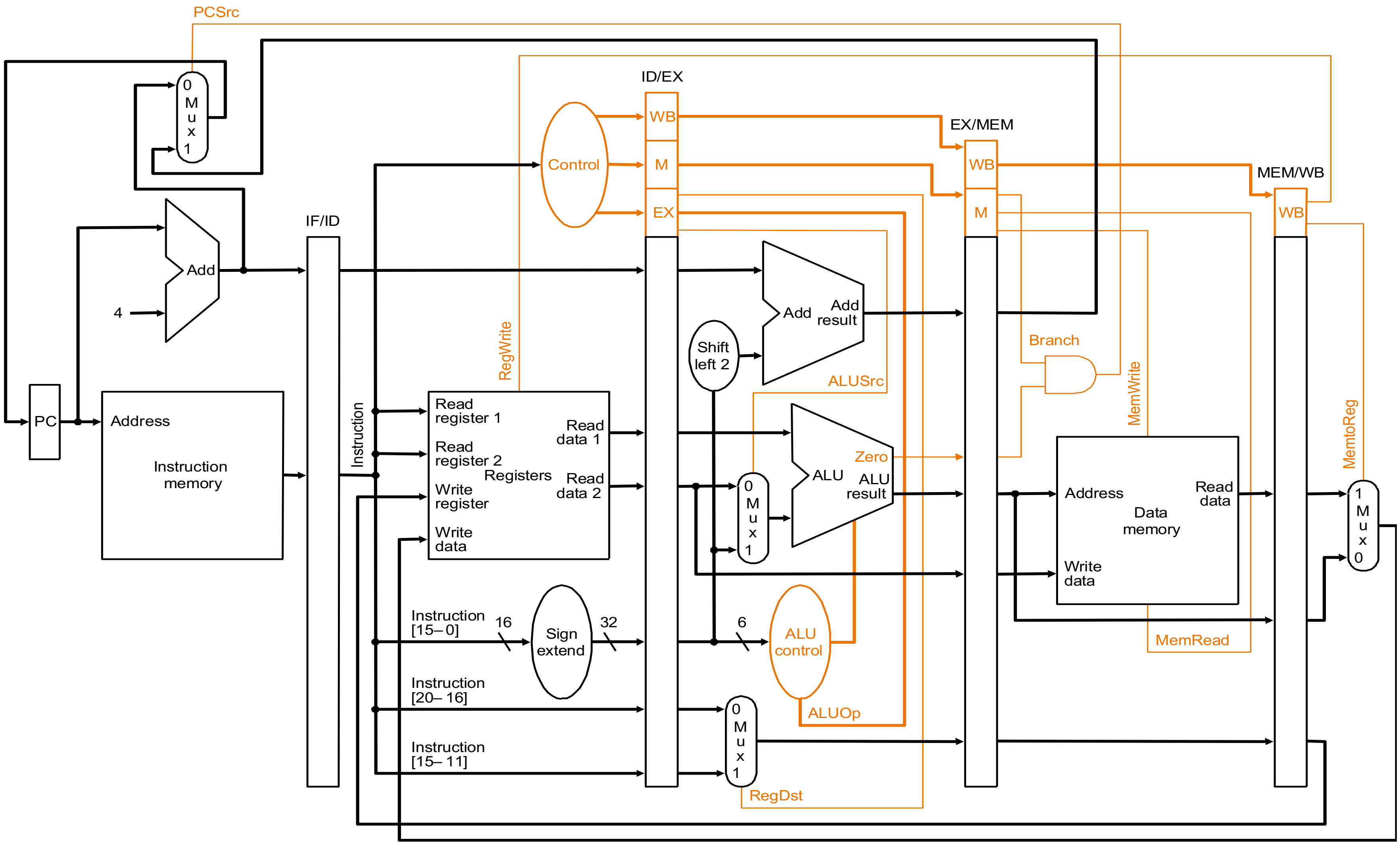


Pipeline Control

- The Main Control generates the control signals during Reg/Dec
  - Control signals for Exec (ALUSrc, ...) are used 1 cycle later
  - Control signals for Mem (MemWr Branch) are used 2 cycles later
  - Control signals for Wr (MemtoReg MemWr) are used 3 cycles later



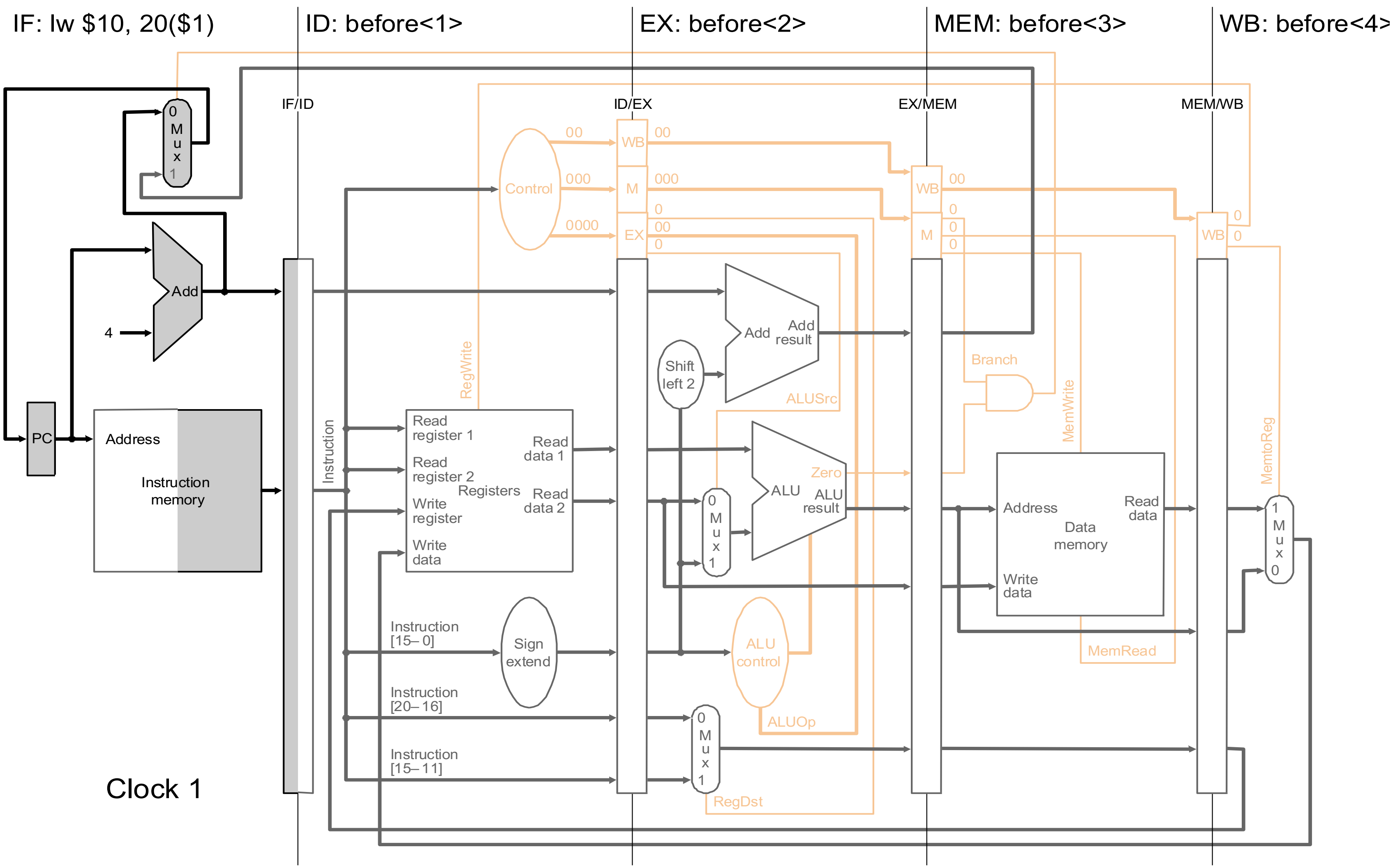
Datapath with Control



Instructions Pipelining Example

```
lw     $10, 20($1)
sub    $11, $2, $3
and    $12, $4, $5
or     $13, $6, $7
add    $14, $8, $9
```

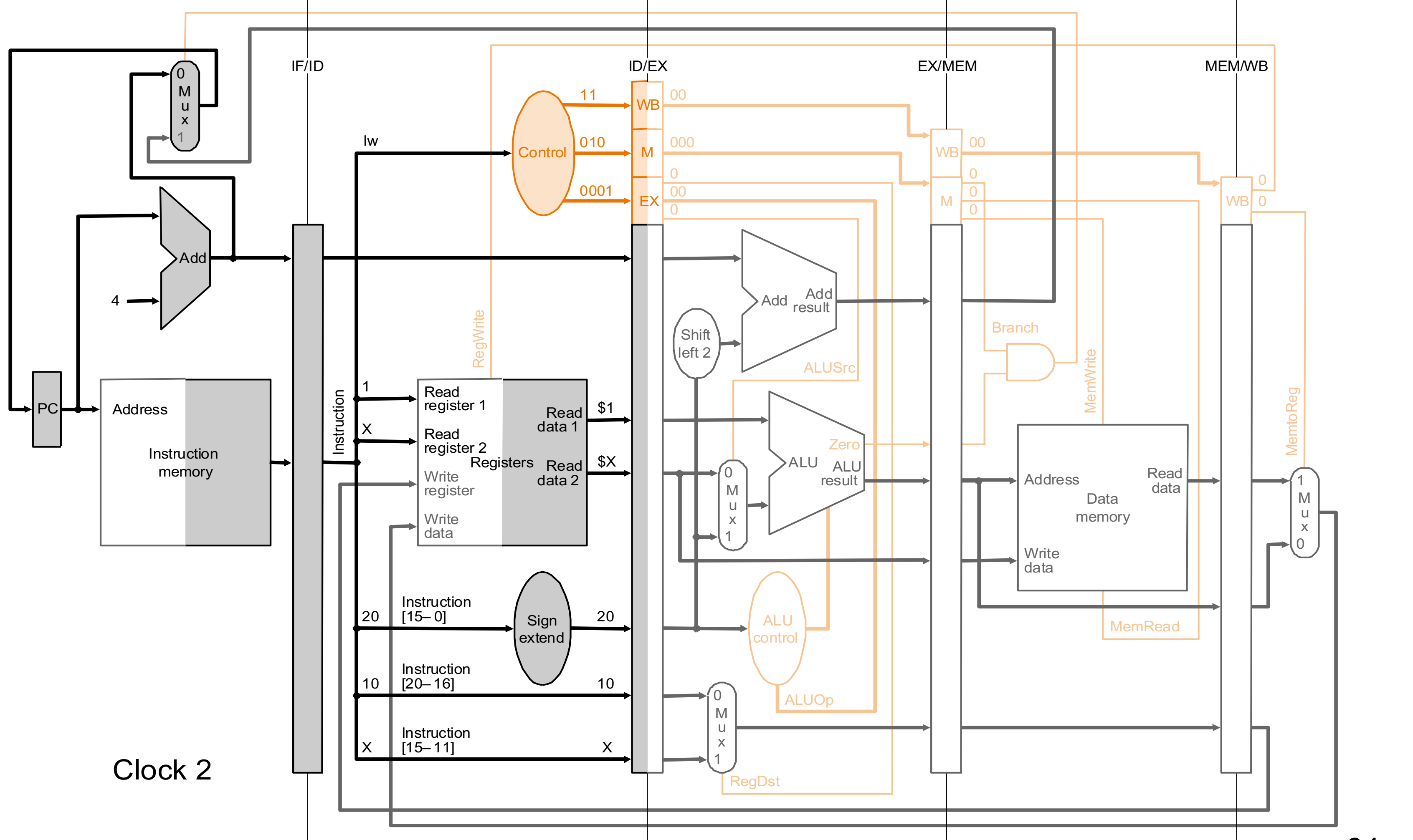
Cycle 1



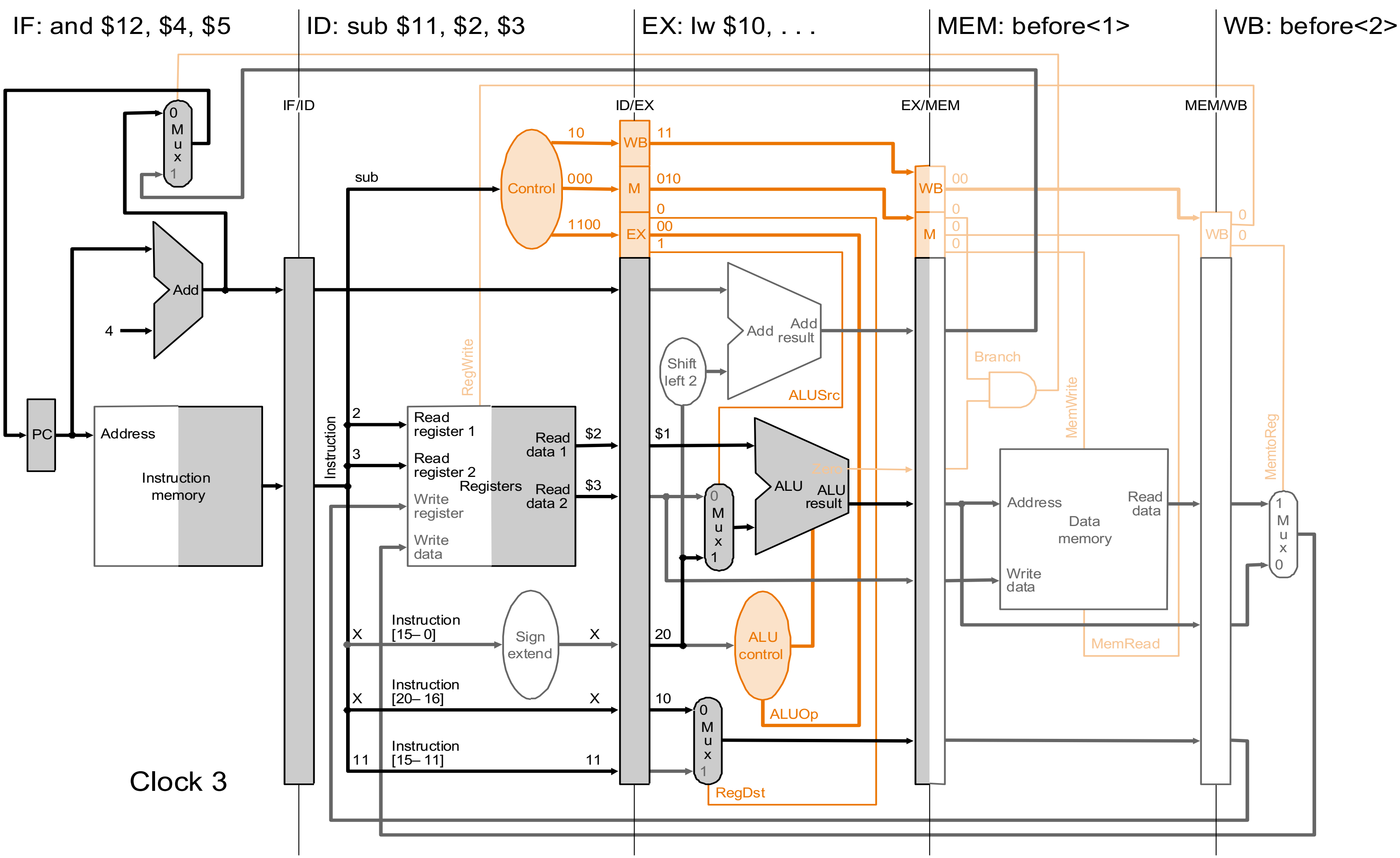
IF: lw $10, 20($1)
ID: before<1>
EX: before<2>
MEM: before<3>
WB: before<4>
IF/ID
ID/EX
EX/MEM
MEM/WB
0 Mux 1
Add
4
PC
Address
Instruction memory
Instruction
Control
WB
M
EX
00
000
0000
RegWrite
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
Instruction [15– 0]
Sign extend
Instruction [20– 16]
Instruction [15– 11]
Shift left 2
Add
Add result
ALUSrc
ALU
Zero
ALU result
ALU control
ALUOp
RegDst
Branch
MemWrite
Address
Data memory
Read data
Write data
MemRead
MemtoReg
1 Mux 0
Clock 1

Cycle 2

IF: sub $11, $2, $3 | ID: lw $10, 20($1) | EX: before<1> | MEM: before<2> | WB: before<3>

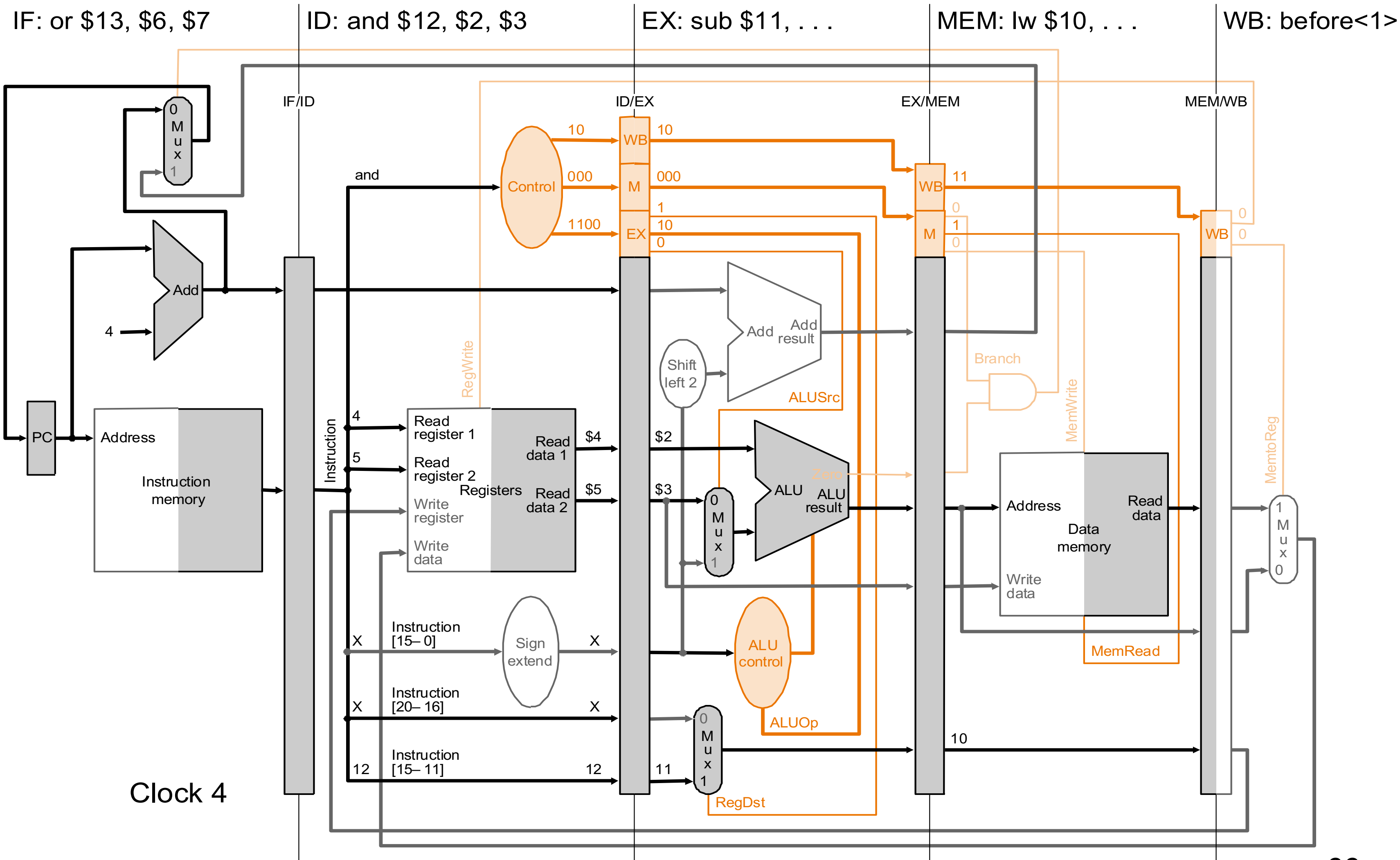


Cycle 3



IF: and $12, $4, $5
ID: sub $11, $2, $3
EX: lw $10, . . .
MEM: before<1>
WB: before<2>
IF/ID
ID/EX
EX/MEM
MEM/WB
sub
Control
10
000
1100
WB
M
EX
11
010
0
00
1
00
0
0
0
0
0
Add
4
PC
Address
Instruction memory
Instruction
2
3
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
$2
$3
$1
RegWrite
Shift left 2
Add
Add result
ALUSrc
ALU
ALU result
Zero
Branch
MemWrite
Address
Data memory
Write data
Read data
MemRead
MemtoReg
Instruction [15– 0]
Sign extend
X
X
20
ALU control
ALUOp
Instruction [20– 16]
X
X
10
Instruction [15– 11]
11
11
RegDst
M u x
0
1
Clock 3

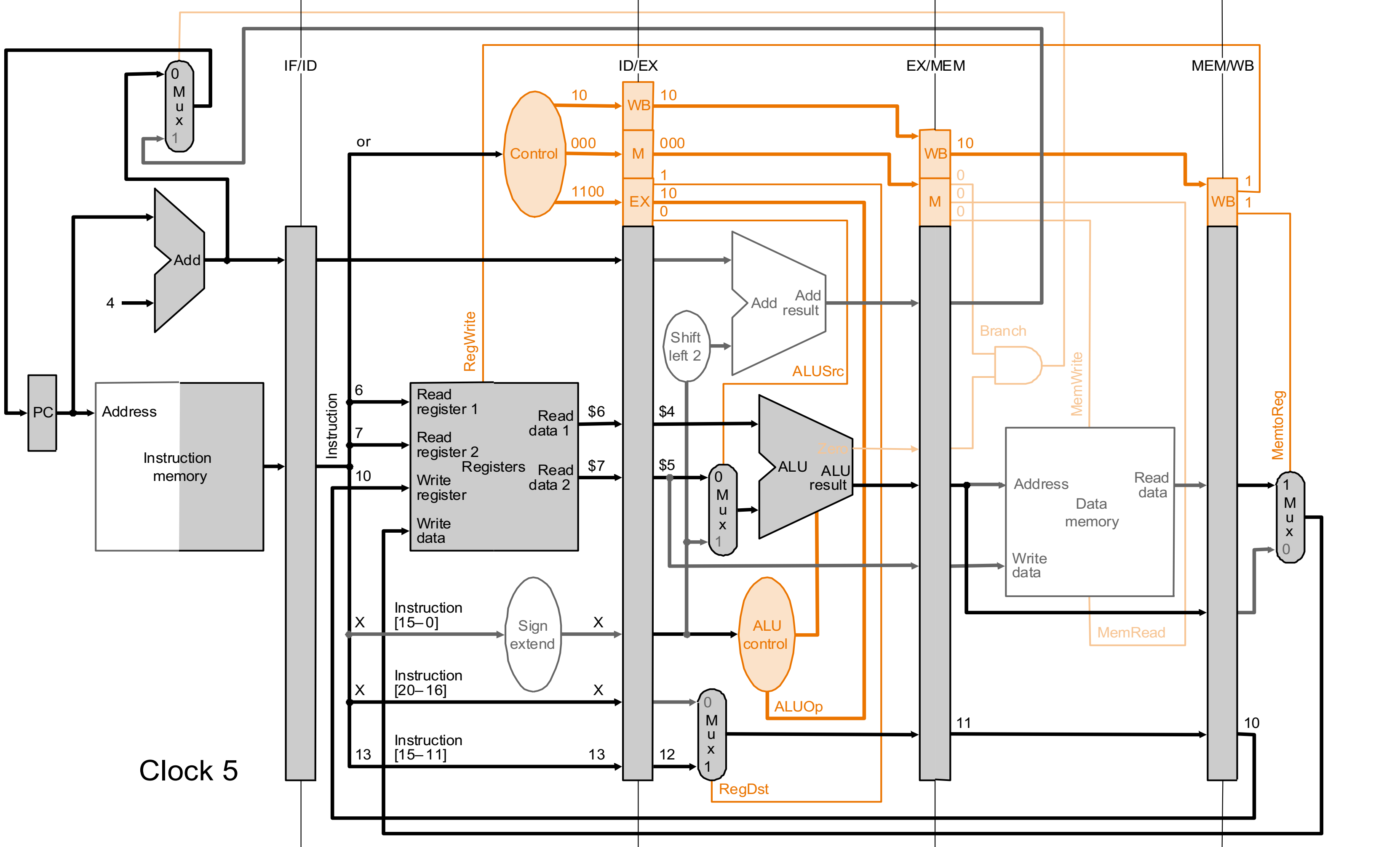
Cycle 4



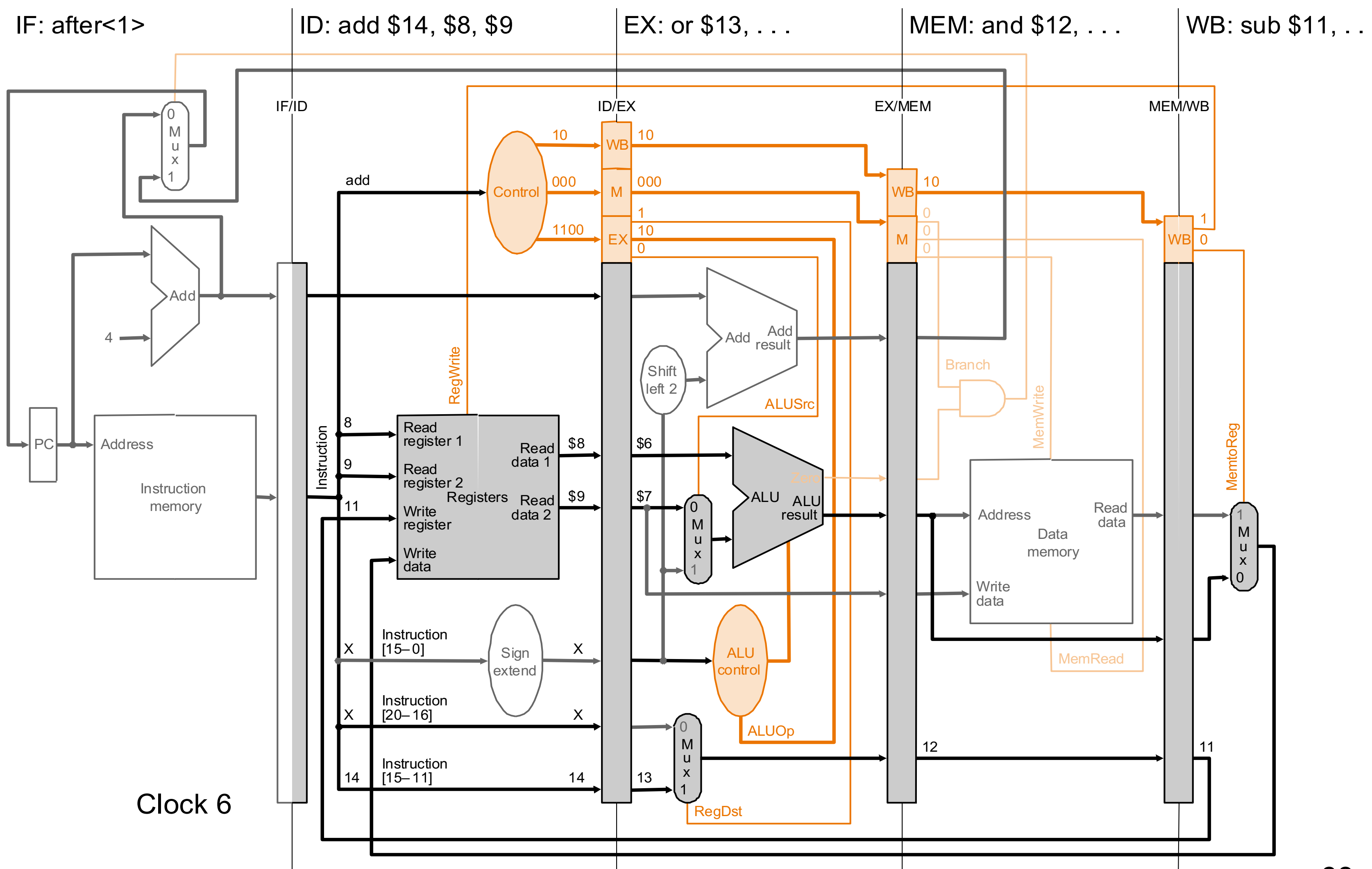
IF: or $13, $6, $7
ID: and $12, $2, $3
EX: sub $11, . . .
MEM: lw $10, . . .
WB: before<1>
IF/ID
ID/EX
EX/MEM
MEM/WB
Control
WB
M
EX
10
000
1100
11
and
PC
Address
Instruction memory
Instruction
Add
4
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
$4
$5
$2
$3
Sign extend
Instruction [15– 0]
Instruction [20– 16]
Instruction [15– 11]
X
12
11
RegWrite
Shift left 2
Add result
ALUSrc
ALU
ALU result
Zero
ALU control
ALUOp
RegDst
Branch
MemWrite
Data memory
Read data
Write data
MemRead
MemtoReg
Mux
Clock 4

Cycle 5

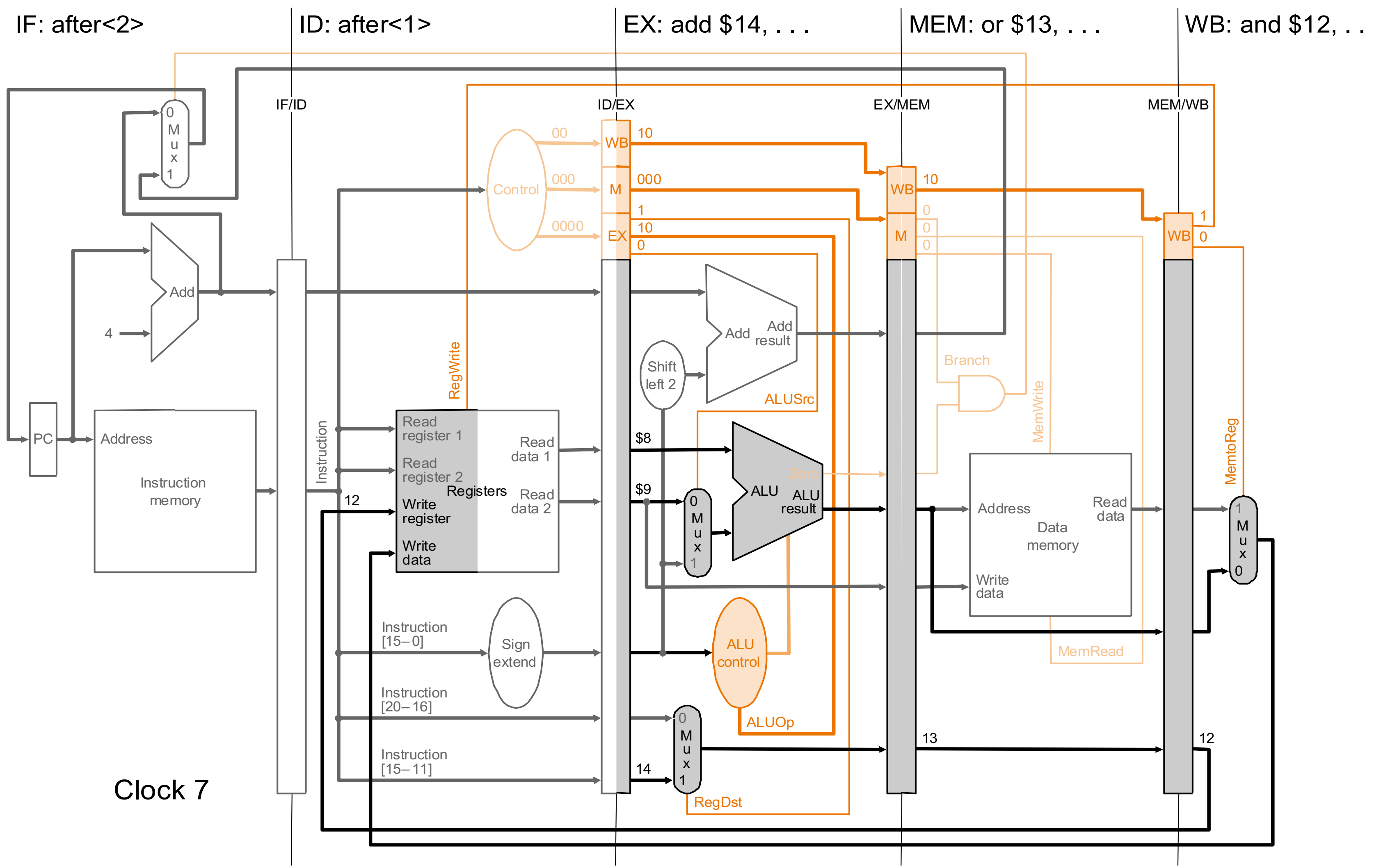
IF: add $14, $8, $9 | ID: or $13, $6, $7 | EX: and $12, . . . | MEM: sub $11, . . . | WB: lw $10, . . .



Cycle 6

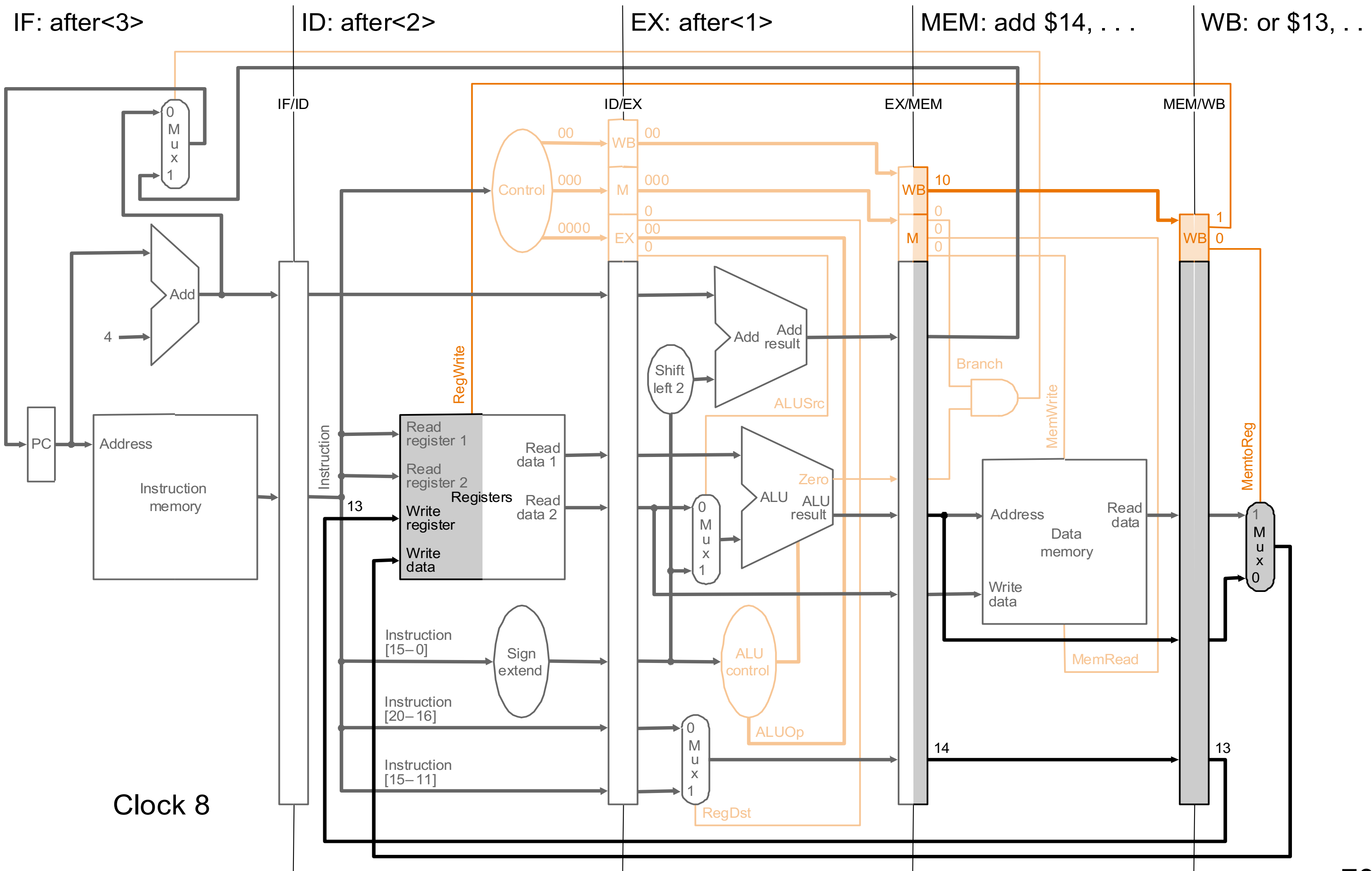

IF: after<1>
ID: add $14, $8, $9
EX: or $13, . . .
MEM: and $12, . . .
WB: sub $11, . . .
IF/ID
ID/EX
EX/MEM
MEM/WB
Control
WB
M
EX
10
000
1100
1
0
PC
Address
Instruction memory
Add
4
Mux
add
Instruction
8
9
11
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
$8
$9
$6
$7
RegWrite
Shift left 2
Add
Add result
ALUSrc
ALU
ALU result
Zero
ALU control
ALUOp
Sign extend
Instruction [15–0]
Instruction [20–16]
Instruction [15–11]
X
14
13
12
11
RegDst
Branch
MemWrite
Address
Data memory
Read data
Write data
MemRead
MemtoReg
Clock 6

Cycle 7



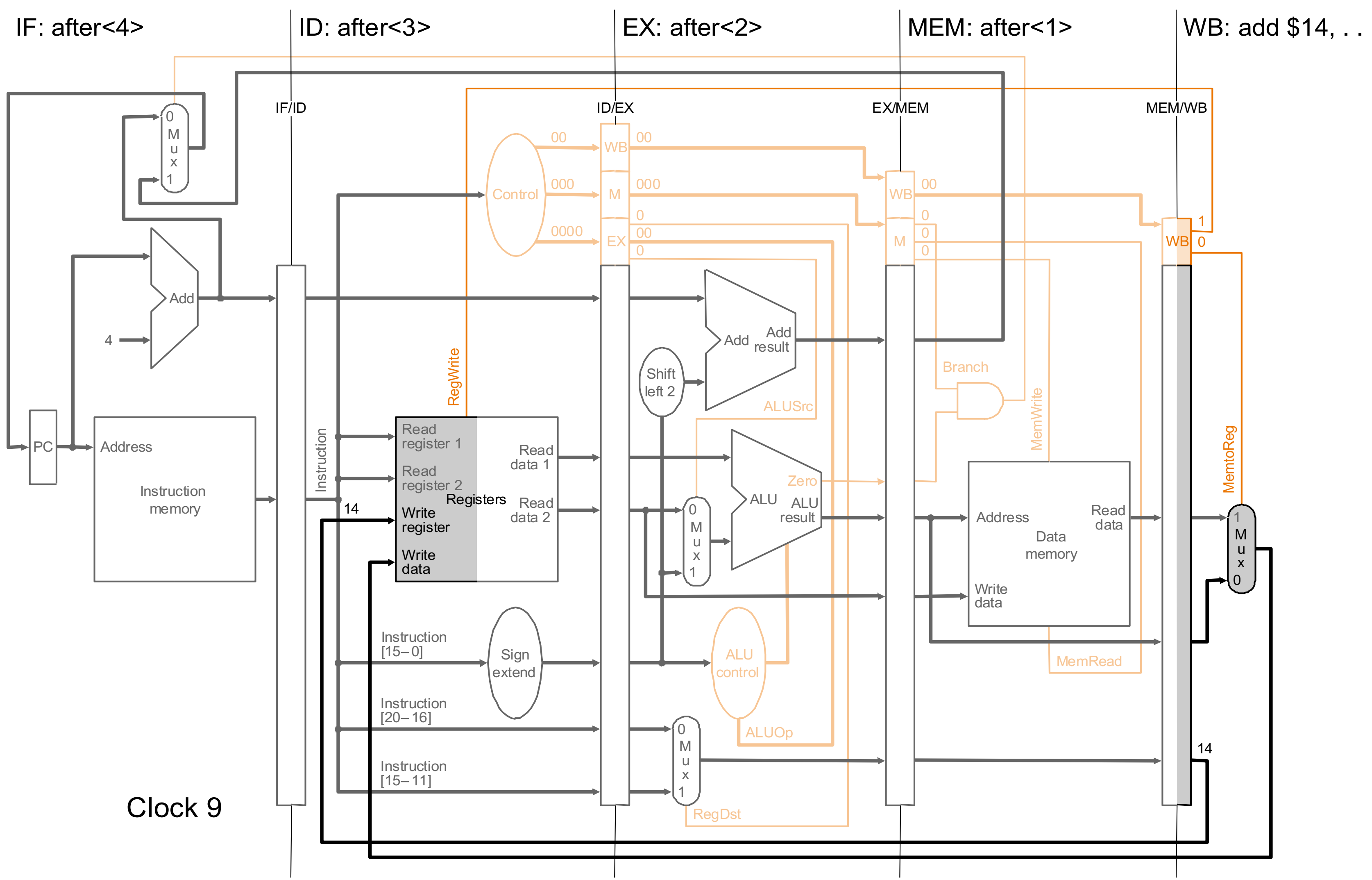
IF: after<2>
ID: after<1>
EX: add $14, . . .
MEM: or $13, . . .
WB: and $12, . . .
IF/ID
ID/EX
EX/MEM
MEM/WB
Control
00
000
0000
WB
M
EX
10
000
1
10
0
10
0
0
0
1
0
Mux
Add
4
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
12
RegWrite
Shift left 2
Add
Add result
ALUSrc
$8
$9
ALU
ALU result
Zero
ALU control
ALUOp
Sign extend
Instruction [15– 0]
Instruction [20– 16]
Instruction [15– 11]
14
RegDst
Branch
MemWrite
Address
Data memory
Read data
Write data
MemRead
MemtoReg
13
12
Clock 7

Cycle 8



IF: after<3>
ID: after<2>
EX: after<1>
MEM: add $14, . . .
WB: or $13, . . .
IF/ID
ID/EX
EX/MEM
MEM/WB
Control
WB
M
EX
00
000
0000
10
1
0
Add
4
PC
Address
Instruction memory
Instruction
RegWrite
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
13
14
Shift left 2
Add
Add result
Branch
ALUSrc
ALU
Zero
ALU result
Mux
Address
Data memory
Read data
Write data
MemWrite
MemRead
MemtoReg
Sign extend
Instruction [15–0]
Instruction [20–16]
Instruction [15–11]
ALU control
ALUOp
RegDst
Clock 8

Cycle 9



IF: after<4>
ID: after<3>
EX: after<2>
MEM: after<1>
WB: add $14, . . .
IF/ID
ID/EX
EX/MEM
MEM/WB
0
M
u
x
1
Add
4
PC
Address
Instruction memory
Instruction
14
Control
00
000
0000
WB
M
EX
RegWrite
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Instruction [15–0]
Sign extend
Instruction [20–16]
Instruction [15–11]
Shift left 2
Add
Add result
ALUSrc
ALU
Zero
ALU result
ALU control
ALUOp
RegDst
Branch
MemWrite
Address
Data memory
Read data
Write data
MemRead
MemtoReg
1
0
14
Clock 9

Summary of Pipeline

- Pipelining is a fundamental concept
  - Multiple steps using distinct resources
  - Utilize capabilities of datapath by pipelined instruction processing
    - Start next instrunction while working on the current one
    - Limited by length of longest stage (plus fill/flush)
    - Need to detect and resolve hazards
- What makes it easy in MIPS?
  - All instructions are of the same length
  - Just a few instruction formats
  - Memory operands only in loads and stores
- What makes pipelining hard? hazards