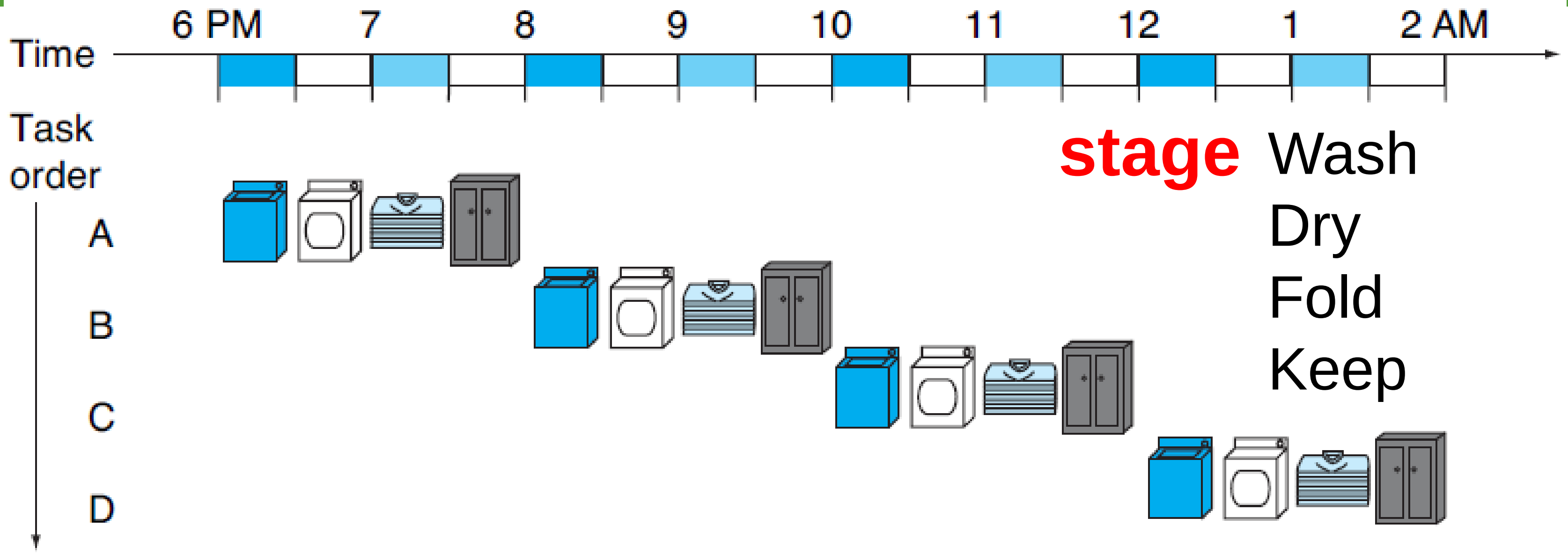
CHAPTER 5-3

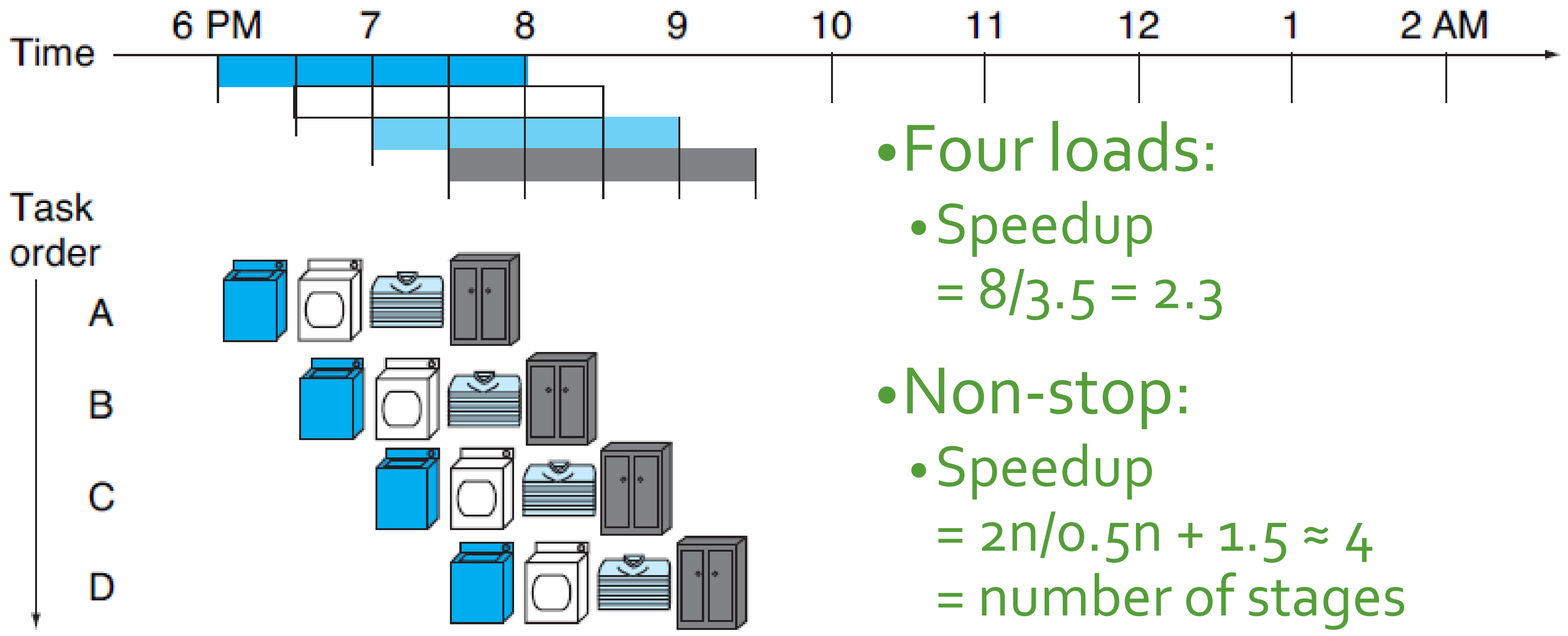
The Processor

By Pattama Longani
Collage of arts, media and Technology

IMPLEMENTATION TECHNIQUE

- **single-cycle**
  - the clock cycle must have the same length for every instruction in this single-cycle design
  - the clock cycle is determined by the longest possible path in the processor
- **Pipelining**
  - multiple instructions are overlapped in execution
  - everything is working in parallel, so more loads are finished per hour (improve throughput)



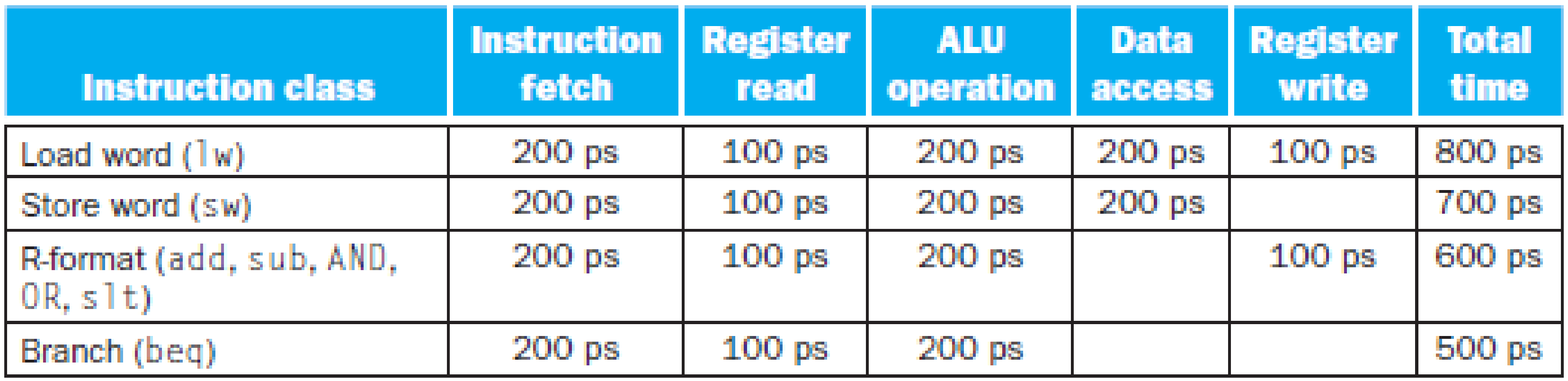


- Four loads:
  - Speedup = 8/3.5 = 2.3
- Non-stop:
  - Speedup = 2n/0.5n + 1.5 ≈ 4 = number of stages

SINGLE-CYCLE VS PIPELINE PERFORMANCE

- If all the stages take about the same amount of time and there is enough work to do, then the **speed-up** due to pipelining is equal to the **number of stages** in the pipeline
- Pipelining improves performance by **increasing instruction throughput**, as opposed to decreasing the execution time of an individual instruction.

EXAMPLE

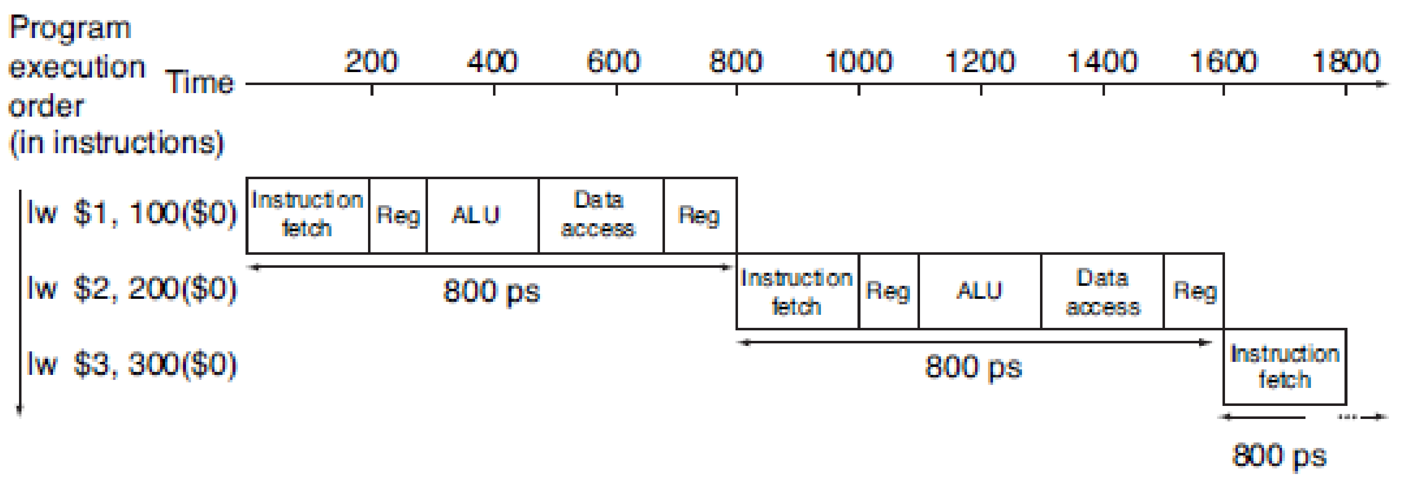


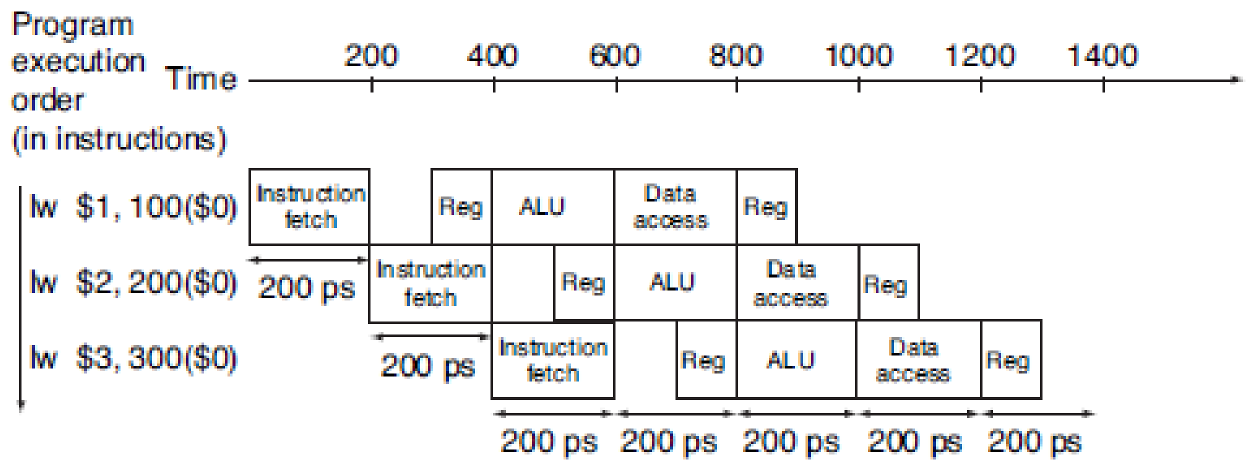
Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, AND, OR, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

- **the single-cycle model**, **every instruction takes exactly one clock cycle**, so the clock cycle must be stretched to accommodate **the slowest instruction** = **800 ps**
- **Pipeline**, **all stages take a single clock cycle**, so the clock cycle must be long enough to accommodate the slowest operation the pipelined execution clock cycle must have **the worst-case clock cycle of 200 ps**

EXAMPLE

- The formula suggests that a five-stage pipeline should offer nearly a fivefold improvement over the 800 ps non-pipelined time, or a 160 ps clock cycle
  - however, that the stages may be imperfectly balanced.

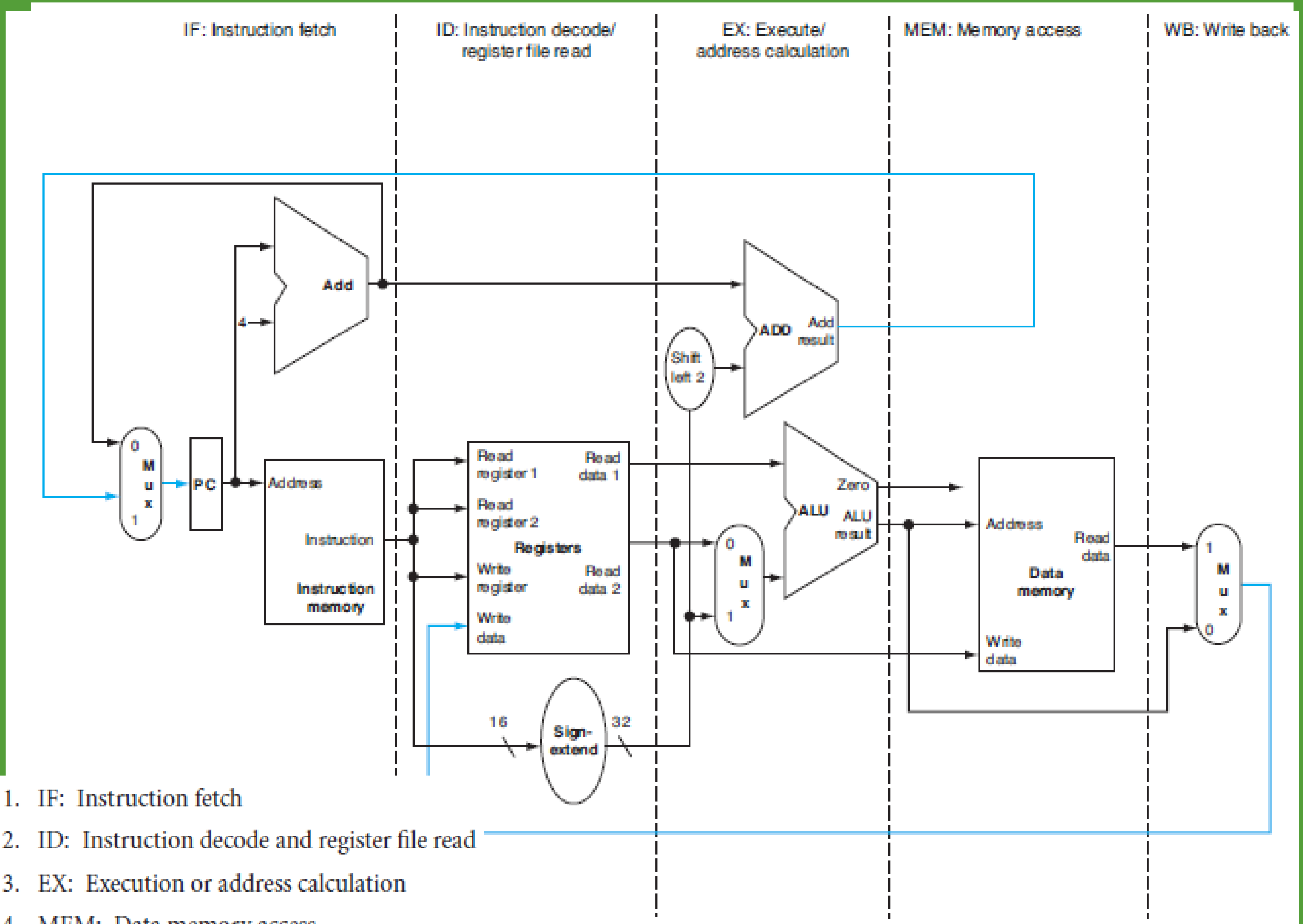




- it's 2,400 ps versus 1400 ps for 3 instructions
- It's 800*1,000,000+2,400 versus ~1,000,000*200+1,400 for 1,000,003 instructions
- The real pipeline speedup = $\frac{800{,}002{,}400 \text{ ps}}{200{,}001{,}400 \text{ ps}} \approx \frac{800 \text{ ps}}{200 \text{ ps}} \approx 4.00$

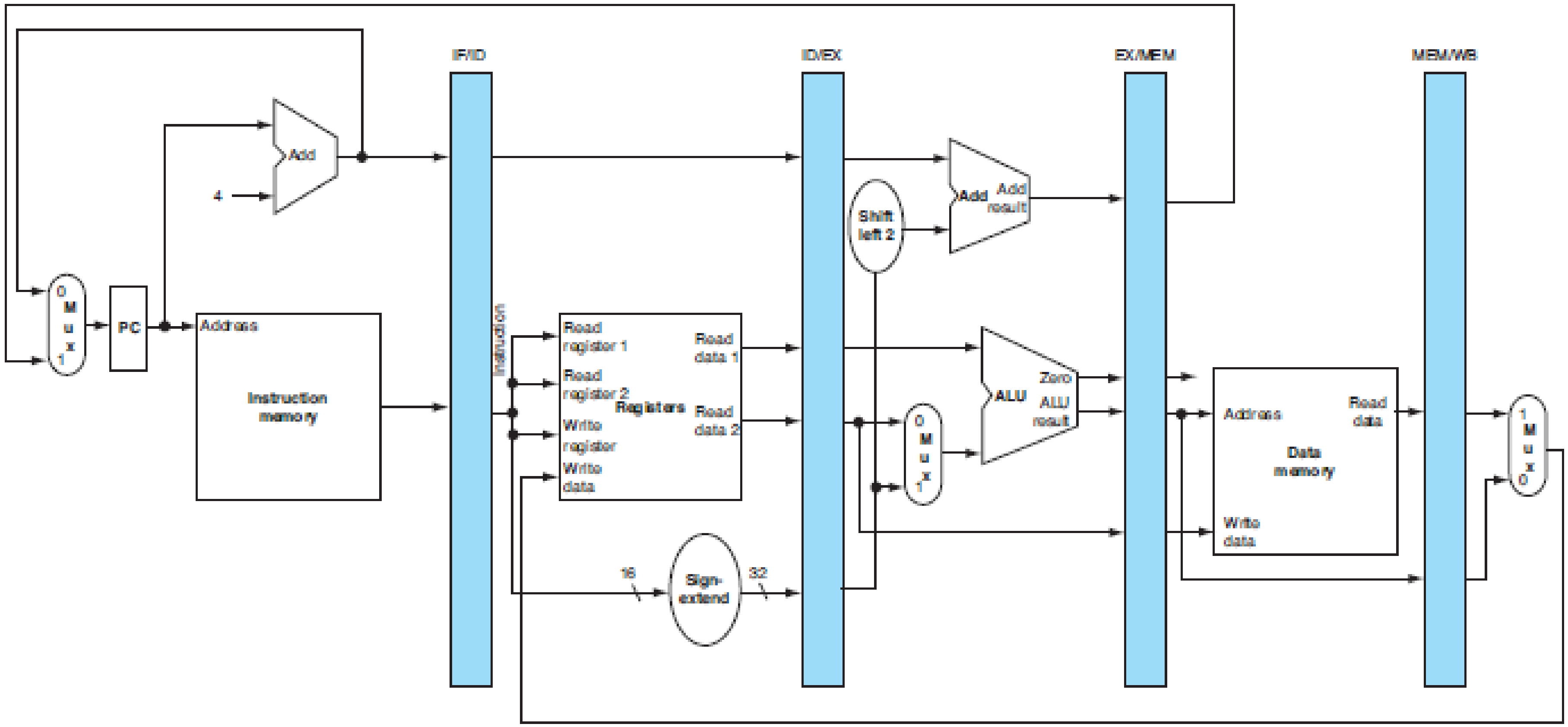
PRINCIPLES APPLY TO PROCESSORS

- MIPS instructions take five steps:
  1. Fetch instruction from memory.
  2. Read registers while decoding the instruction. The regular format of MIPS instructions allows reading and decoding to occur simultaneously.
  3. Execute the operation or calculate an address.
  4. Access an operand in data memory.
  5. Write the result into a register.



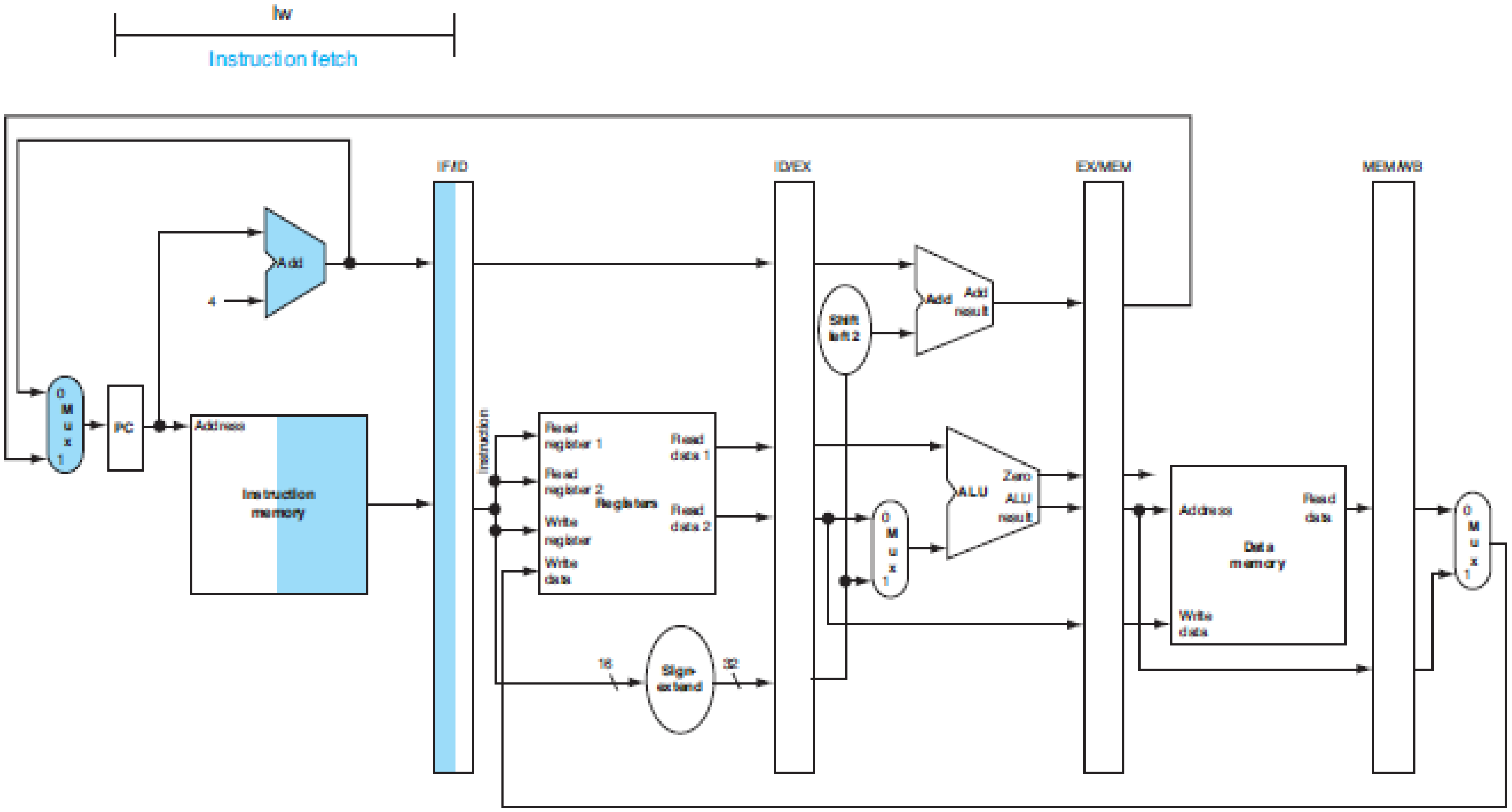
1. IF: Instruction fetch
2. ID: Instruction decode and register file read
3. EX: Execution or address calculation
4. MEM: Data memory access
5. WB: Write back

The blue stick represent “pipeline register”



LOAD1

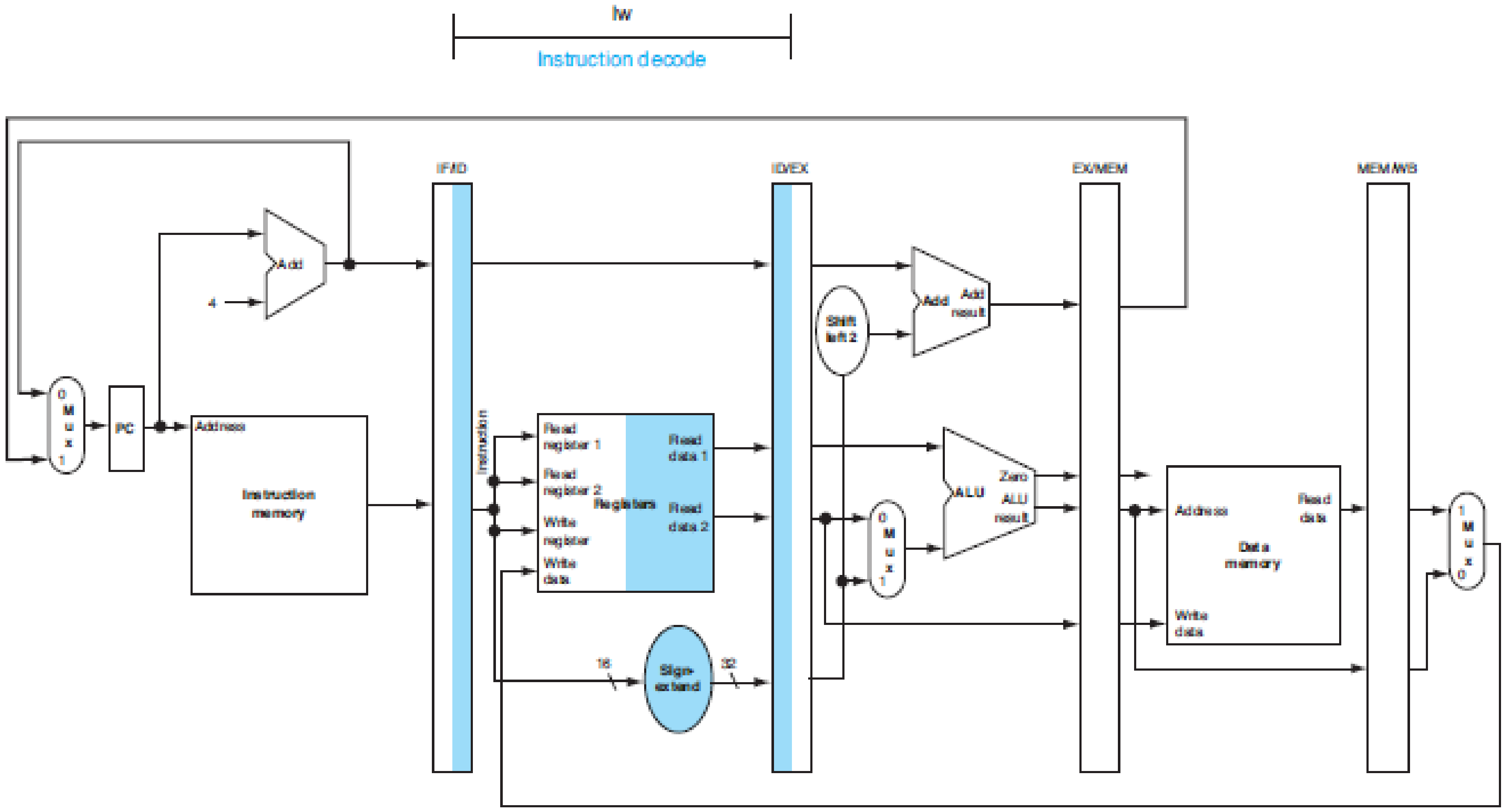
- highlight the *right half = read*
- highlight the *left half =write*



Instruction fetch:

- the instruction being read from memory using the address in the PC and then being placed in the IF/ID pipeline register.
- The PC address is incremented by 4 and then written back into the PC to be ready for the next clock cycle.
  - This incremented address is also saved in the IF/ID pipeline register in case it is needed later for an instruction, such as beq.

LOAD2

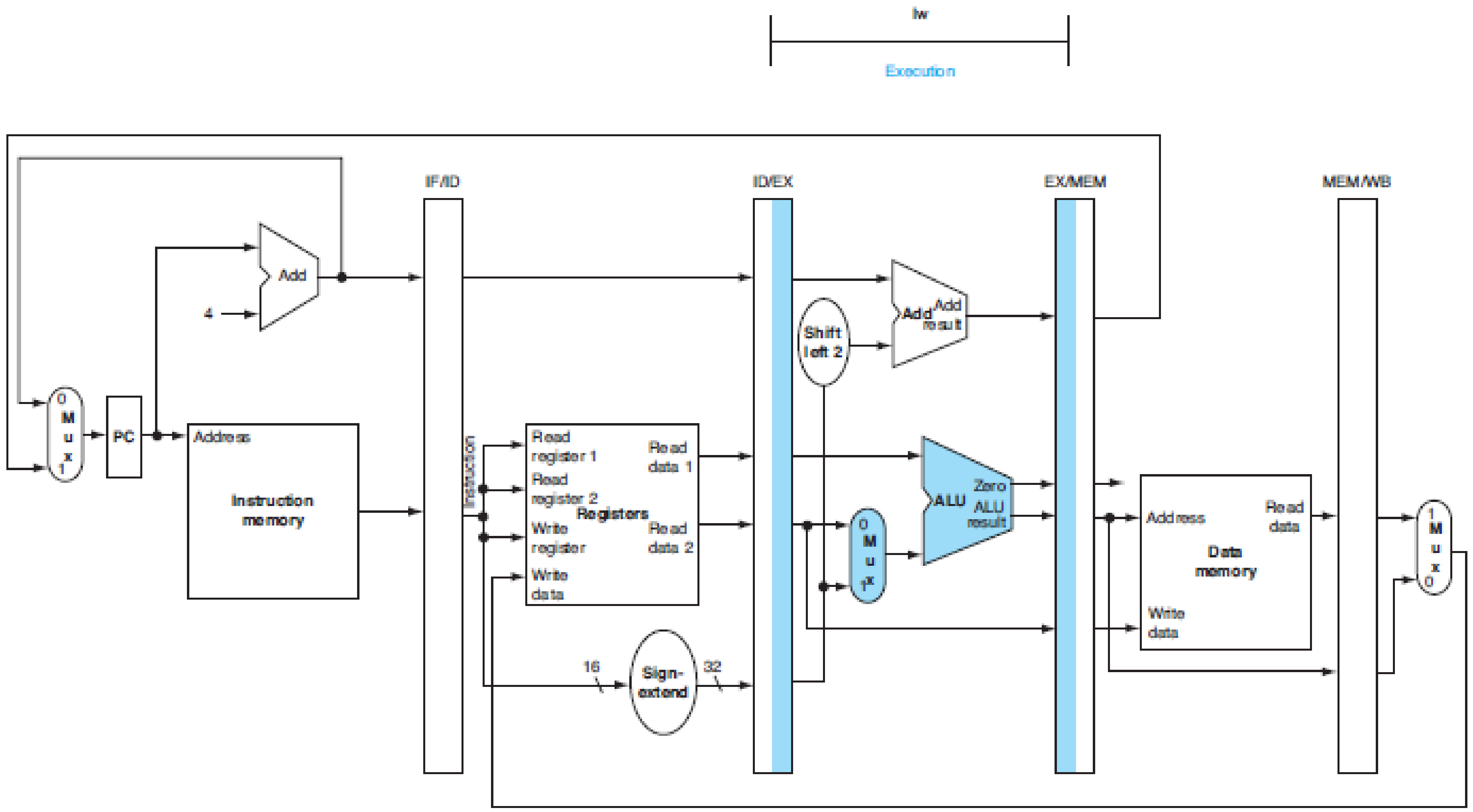


lw
Instruction decode
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data

Instruction decode and register file read:

- The instruction kept in IF/ID pipeline register supplying
  - the 16-bit immediate field, which is sign-extended to 32 bits,
  - the register numbers to read the two registers.
- All three values will be written in the ID/EX pipeline register, along with the incremented PC address.
  - Transfer everything that might be needed by any instruction during a later clock cycle.

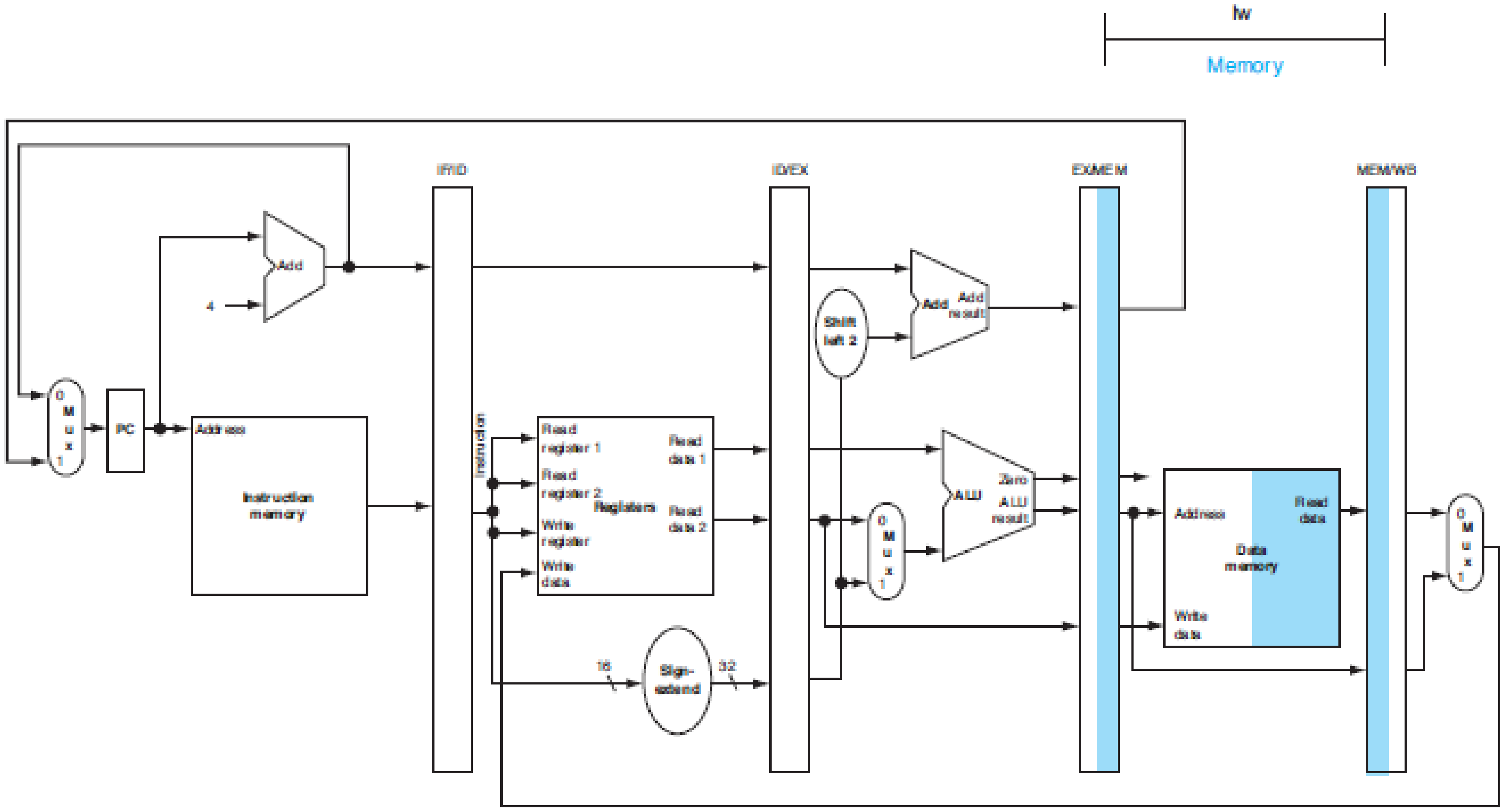
LOAD3


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

Execute or address calculation:

- the load instruction reads the contents of register 1 immediate from the ID/EX pipeline register
- the load instruction reads the sign-extended immediate from the ID/EX pipeline register
- Adds both data using the ALU.
- That sum is placed in the EX/MEM pipeline register.

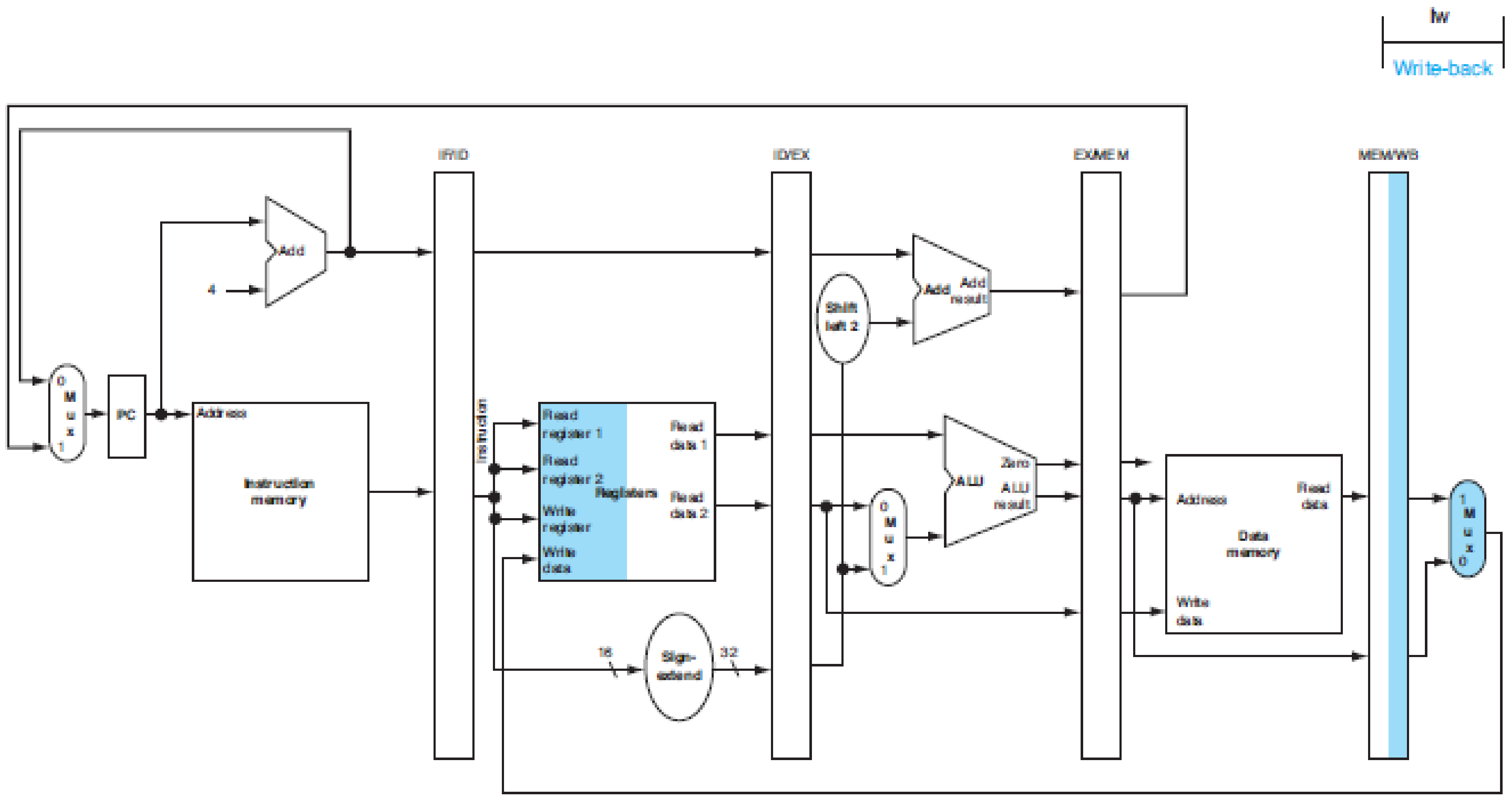
LOAD4


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

Memory access:

- the load instruction reading the data memory using the address from the EX/MEM pipeline register
- loading the data into the MEM/WB pipeline register.

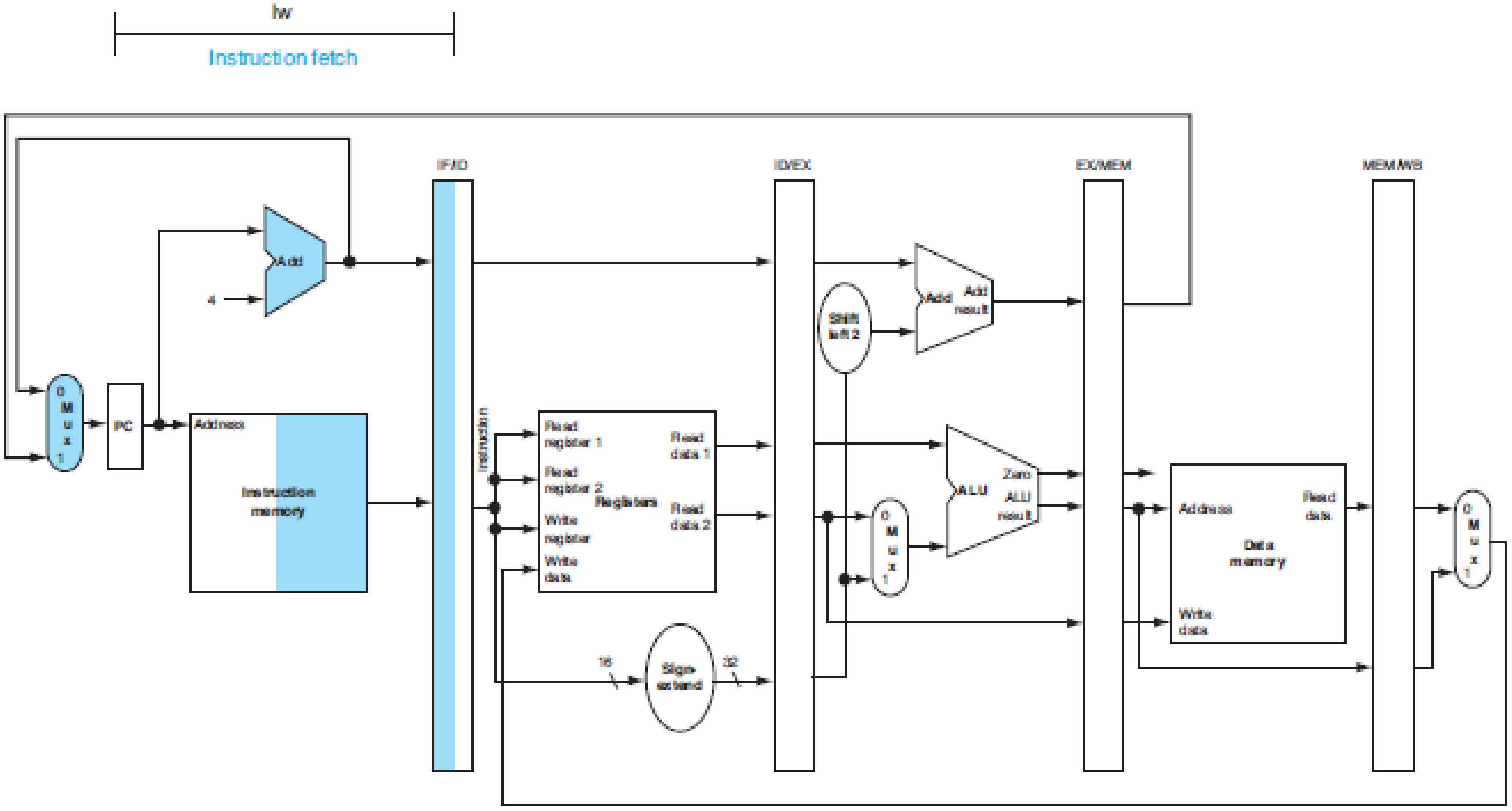
LOAD5


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

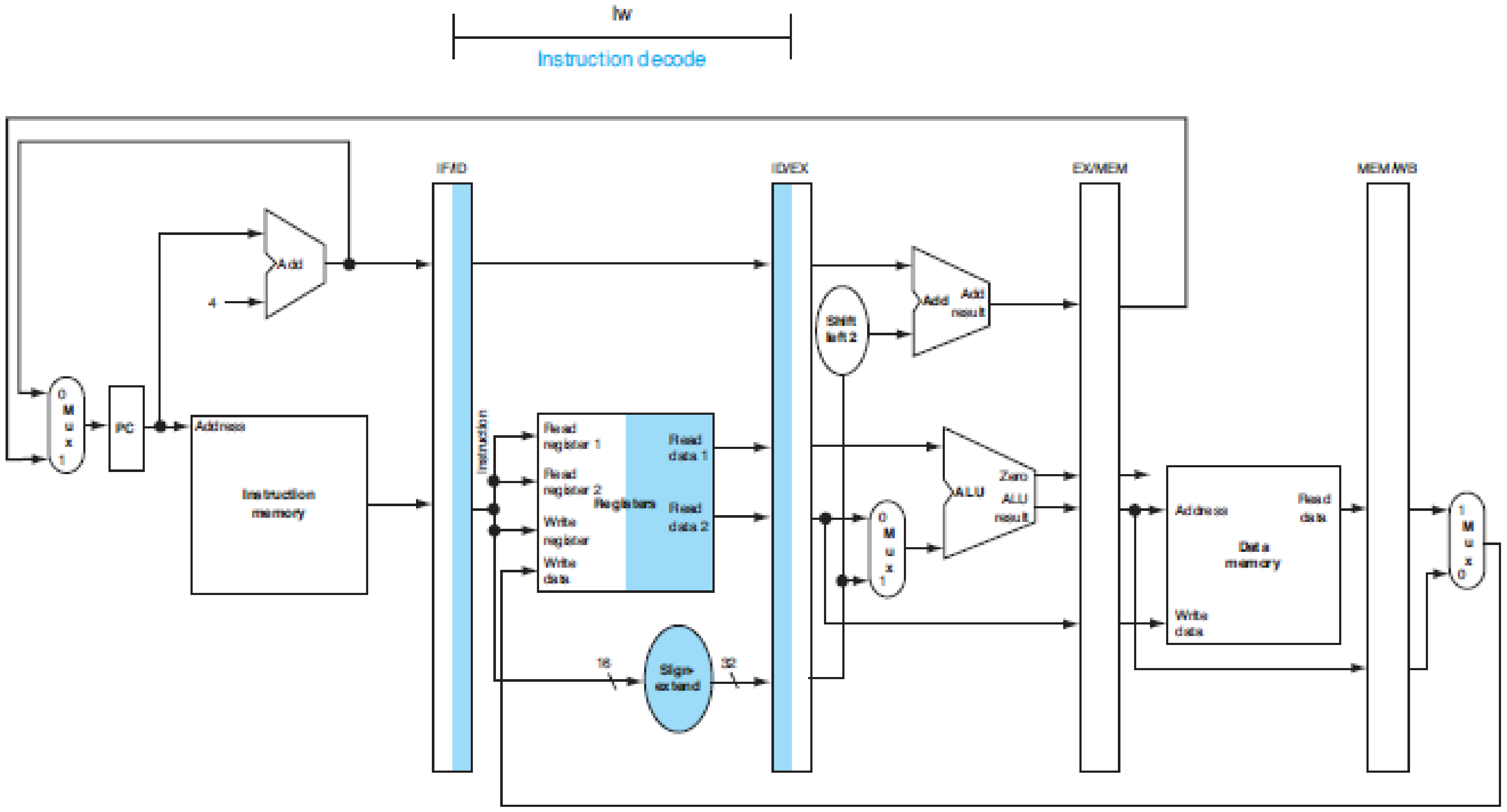
Write-back:

- Reading the data from the MEM/WB pipeline register and writing it into the register file in the middle of the figure.

STORE1

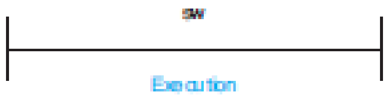

lw
Instruction fetch
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
Add
Add result
Shift left 2
0 M u x 1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
0 M u x 1
ALU
Zero
ALU result
Address
Read data
Data memory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

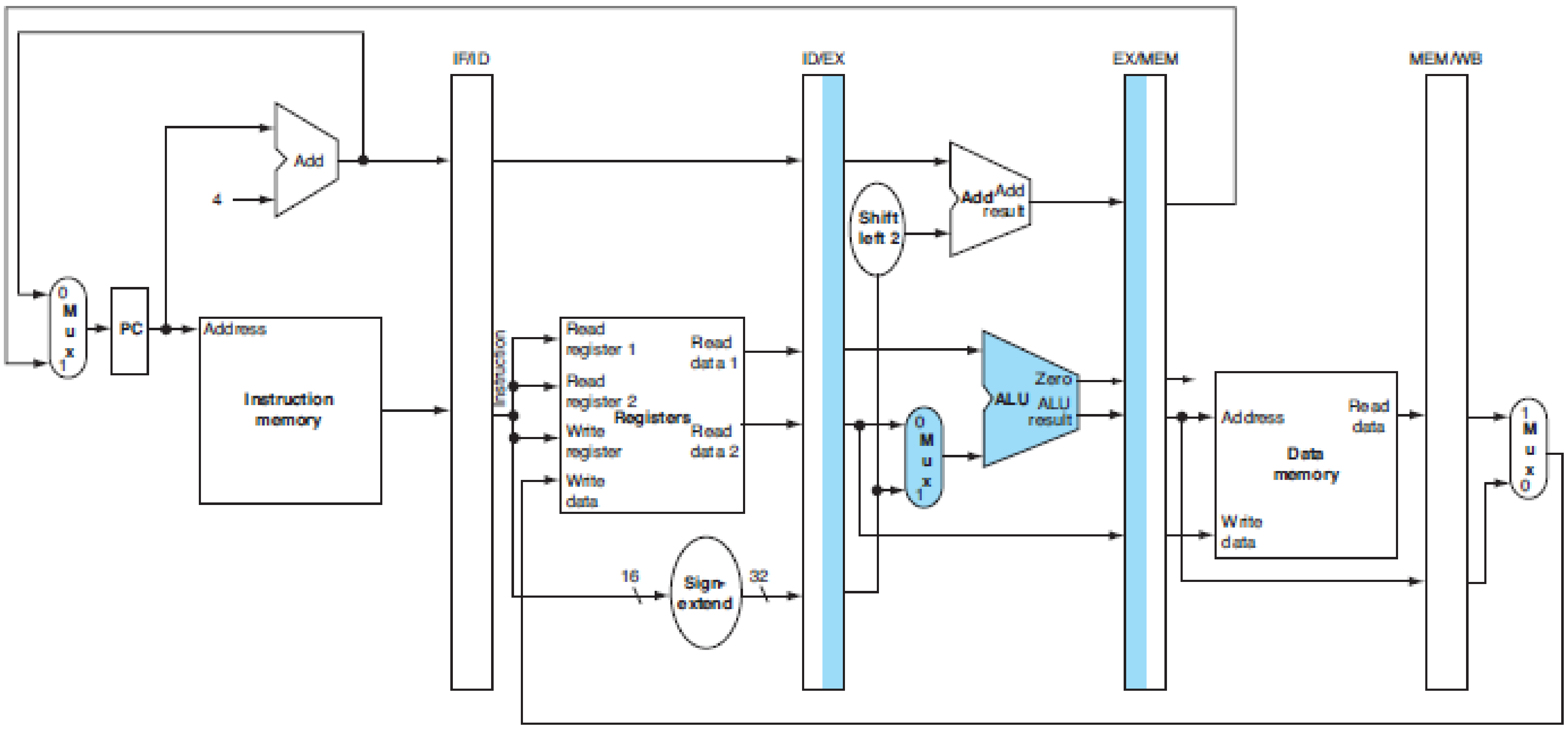
STORE2



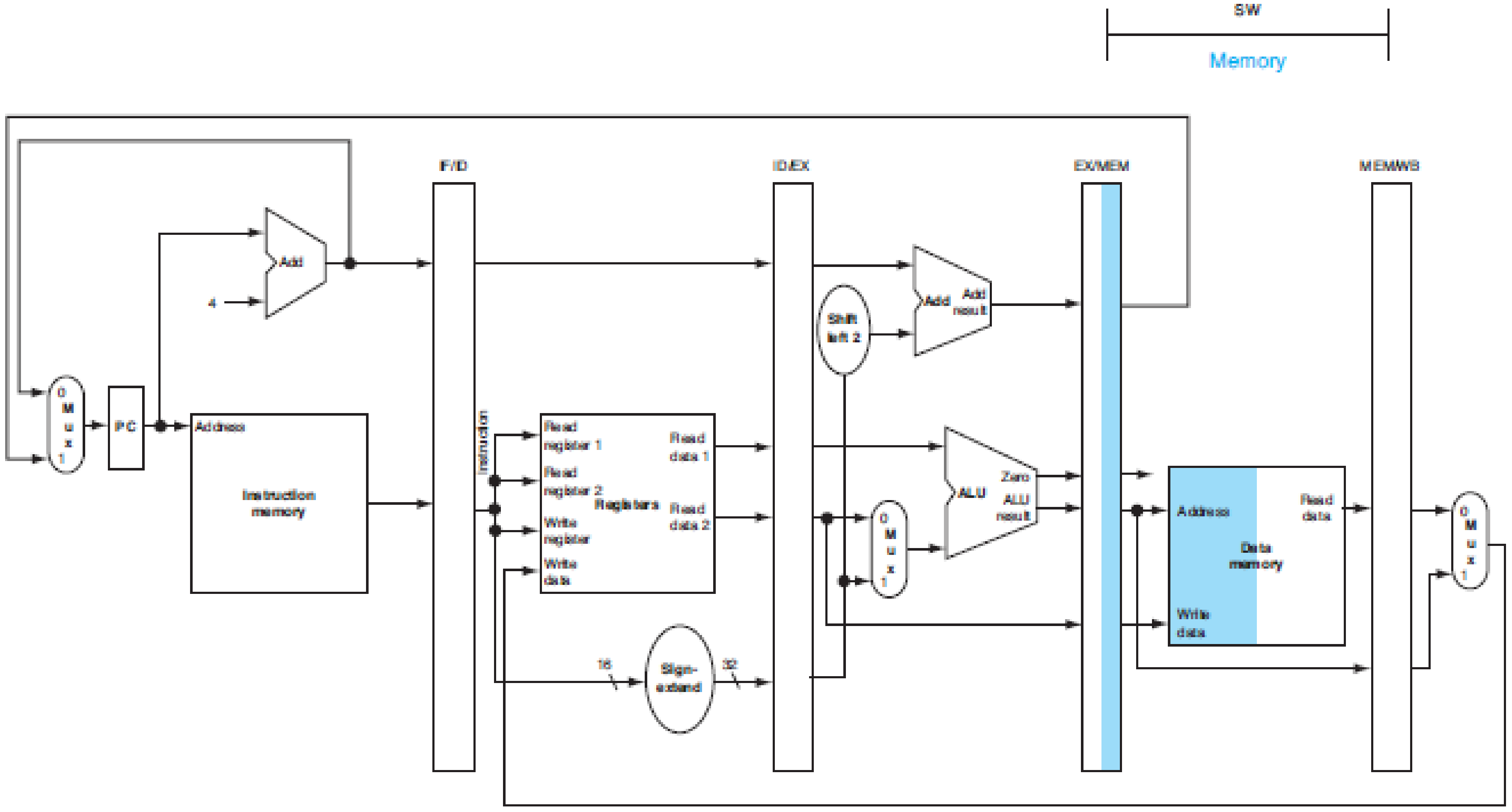
lw
Instruction decode
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
Add
Add result
Shift left 2
M u x
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
ALU
Zero
ALU result
Address
Read data
Data memory
Write data

STORE3


sw
Execution


IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
PC
0
M
u
x
1
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
0
M
u
x
1
ALU
Zero
ALU result
Address
Data memory
Write data
Read data
1
M
u
x
0

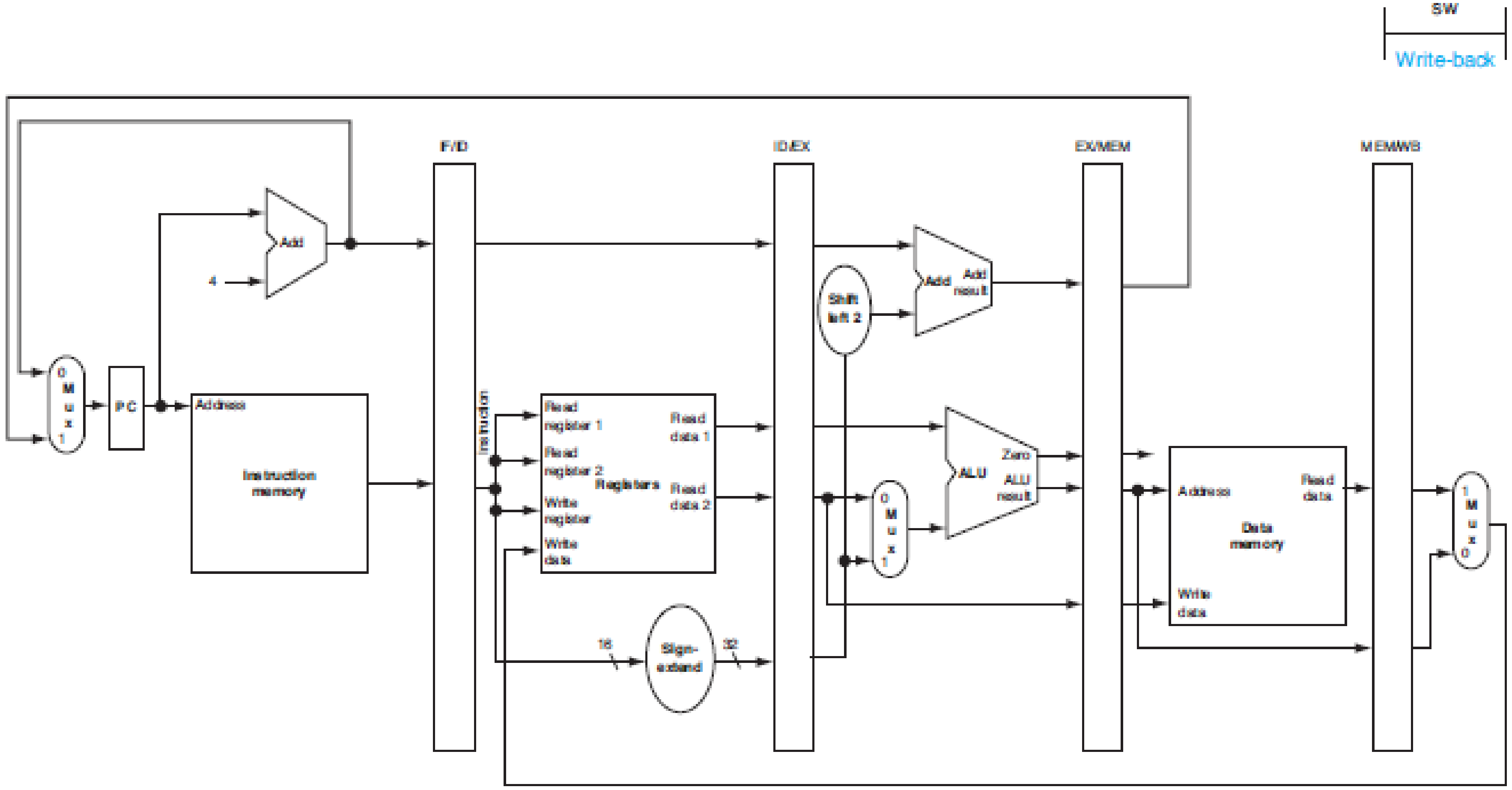
STORE4


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

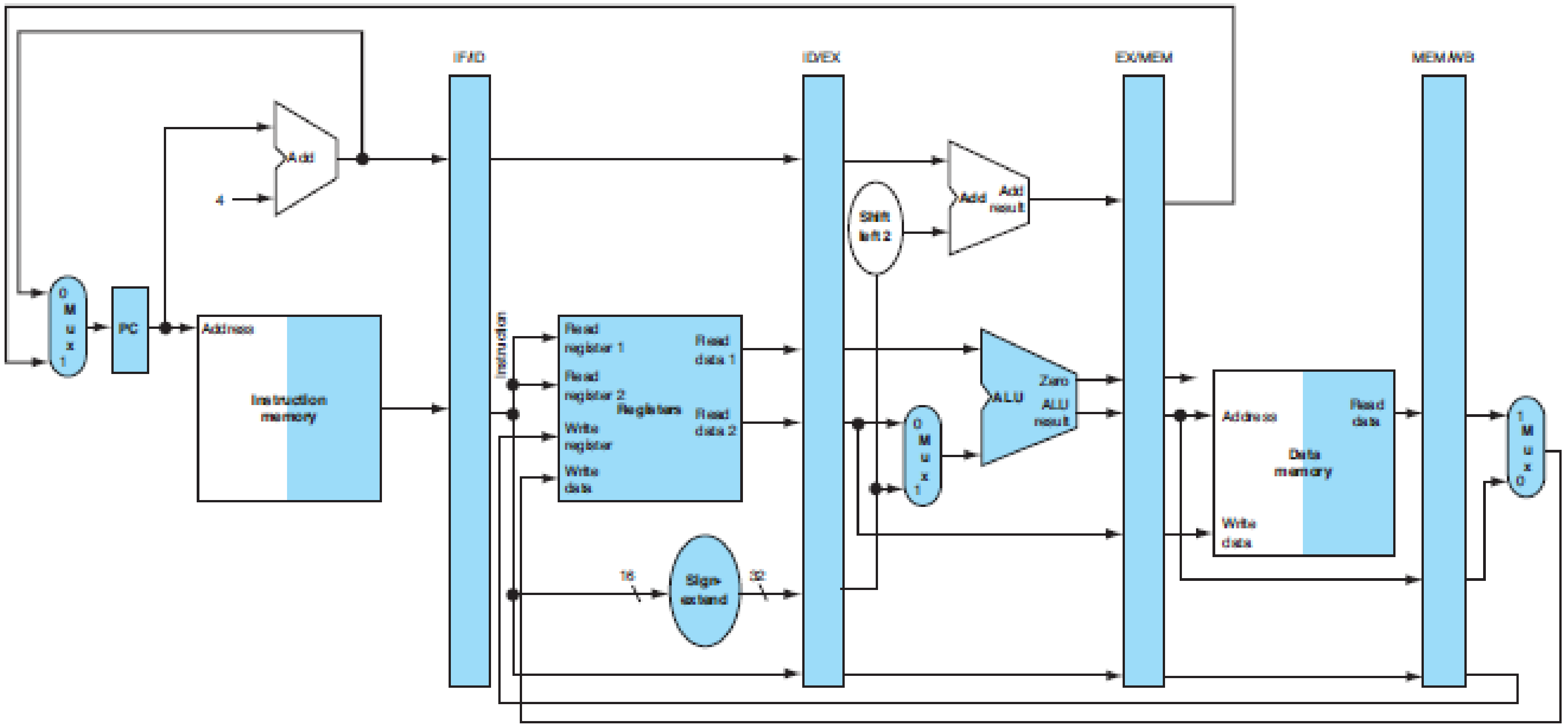
Memory access:

- In an earlier stage, the register ID/EX containing the data to be stored was read
- The only way to make the data available during the MEM stage is to place the data into the EX/MEM pipeline register in the EX stage
- just as we stored the effective address into EX/MEM.

STORE5



SW
Write-back
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
16
Sign-extend
32
M u x
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
M u x
0
1

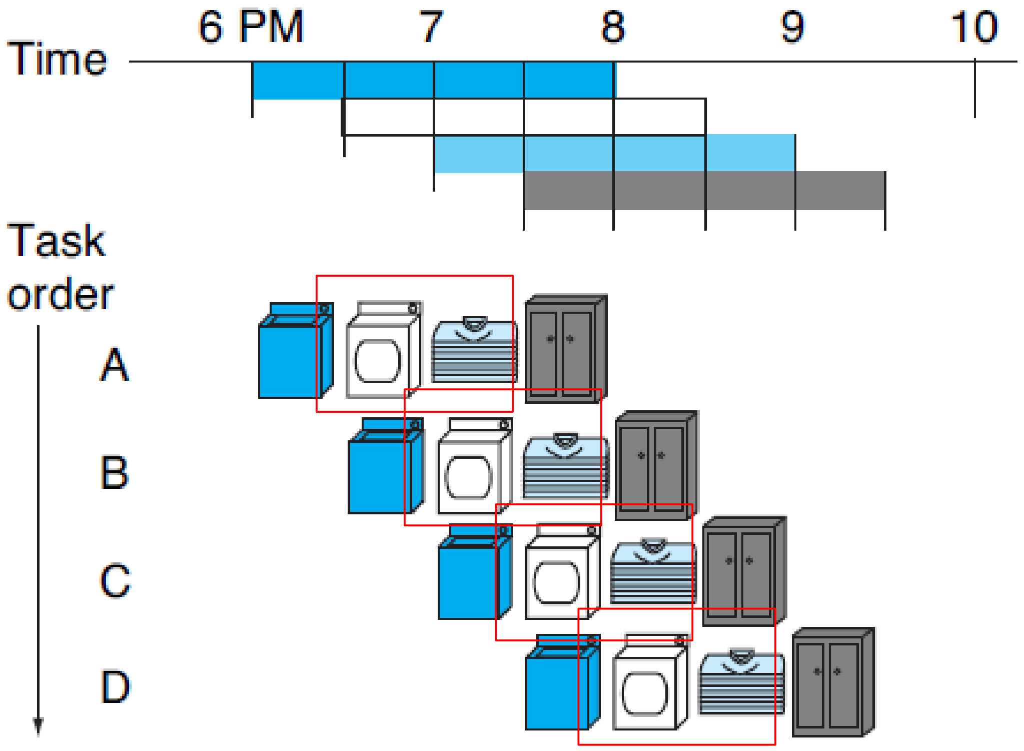

IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0 M u x 1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
0 M u x 1
ALU
Zero
ALU result
Address
Data memory
Write data
Read data
1 M u x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

PIPELINE HAZARDS

- There are situations in pipelining when the ***next instruction cannot execute in the following clock cycle.*** These events are called ***hazards***.
    - Structural Hazards
    - Data Hazards
    - Control Hazards

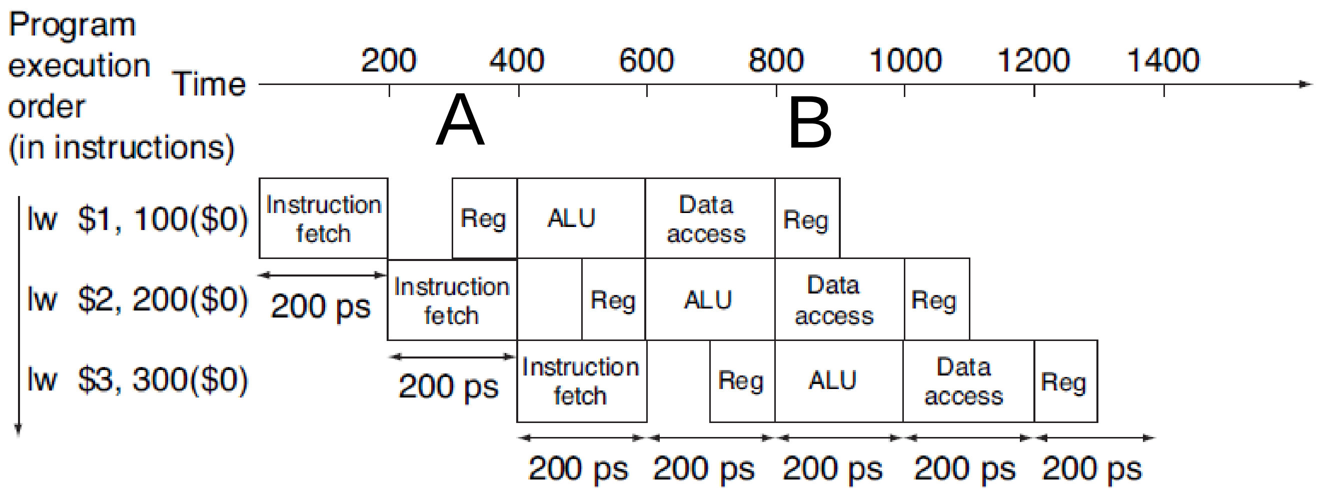
STRUCTURAL HAZARDS

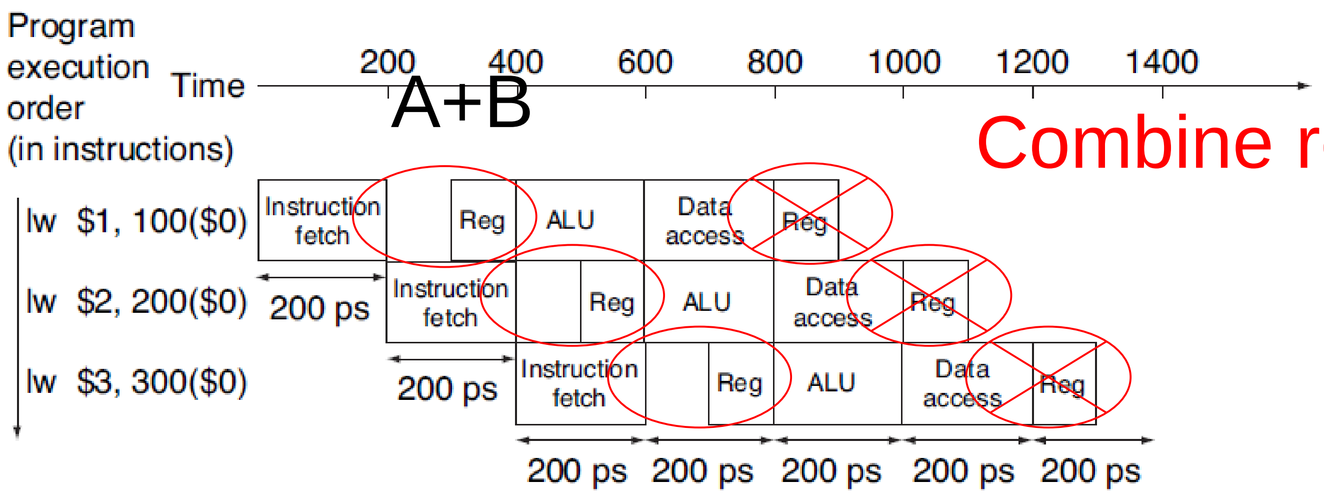
- The hardware cannot support the ***combination of instructions*** that we want to execute in the same clock cycle.



Wash
Dry
Fold
Keep

STRUCTURAL HAZARDS

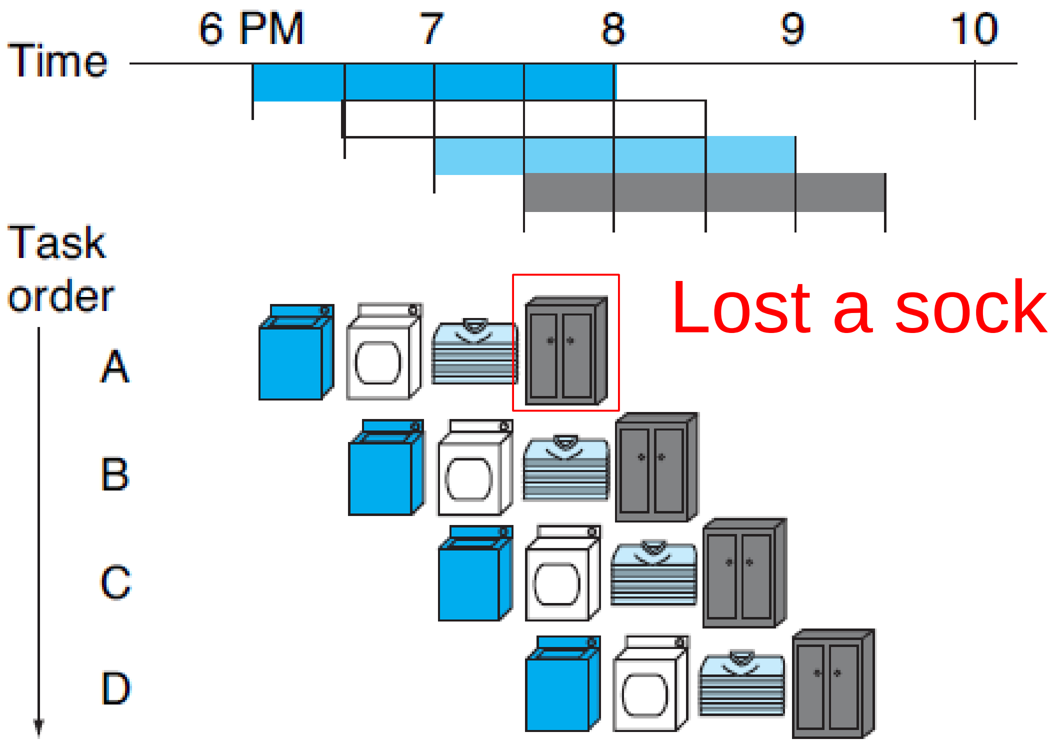




Combine register

DATA HAZARDS

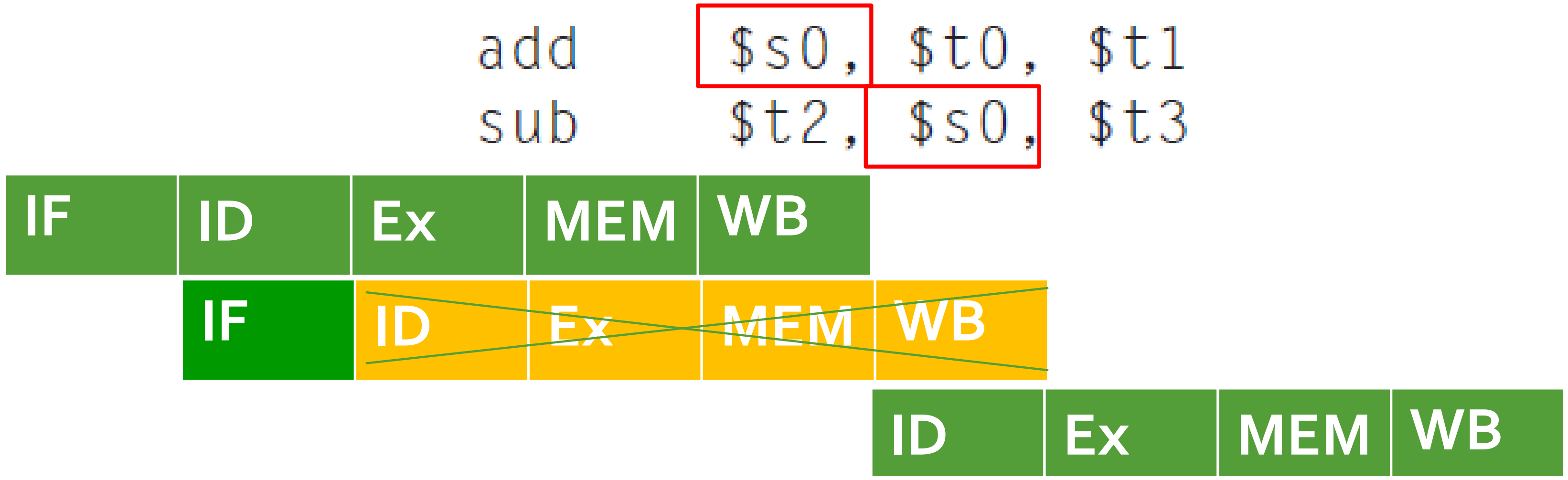
- One step must wait for another to complete.



Wash
Dry
Fold
Keep

DATA HAZARDS

```
add     $s0, $t0, $t1
sub     $t2, $s0, $t3
```



instruction execution

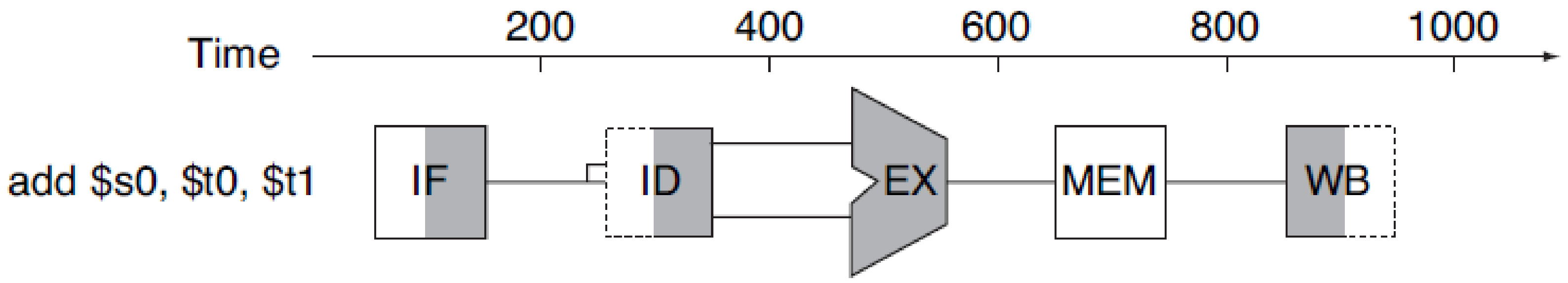
- Instruction Fetch
- Register Read
- Execution
- Memory Access
- Write Back

- The add instruction doesn't write its result until the fifth stage
- we have to waste three clock cycles in the pipeline.

FORWARDING (BYPASSING)

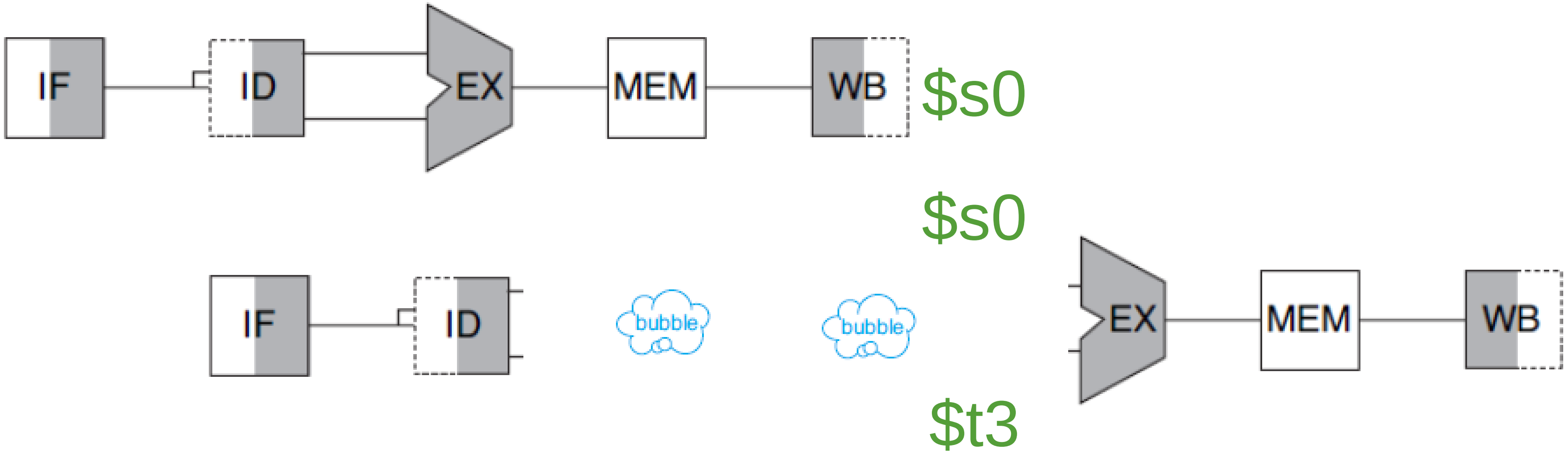
- Although we could try to rely on ***compilers*** to remove all such hazards, the results would not be satisfactory.
- These dependences ***happen just too often*** and the ***delay is just too long*** to expect the compiler to rescue us.
- Therefore, we ***adding extra hardware*** to retrieve the missing item early from the internal resources is called **forwarding** or **bypassing**.

FORWARDING (BYPASSING)

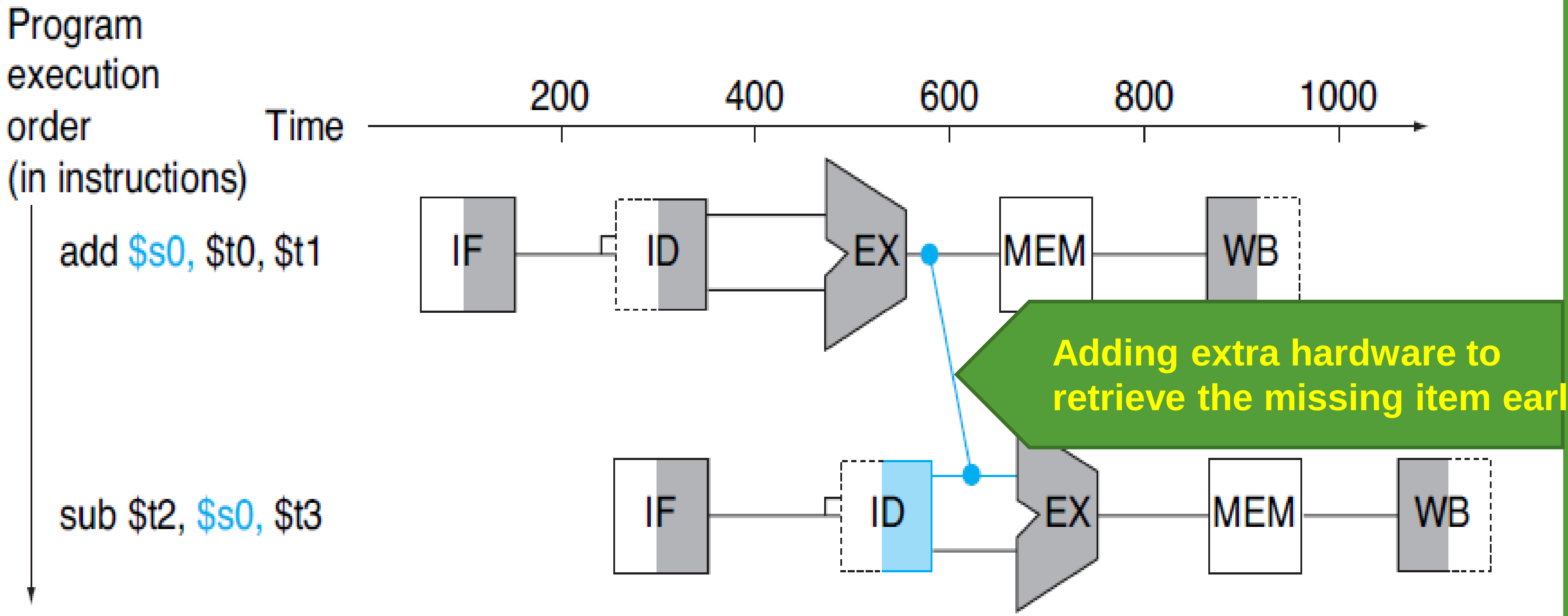


- *IF* for the instruction fetch
- *ID* for the instruction decode/register file read stage
- *EX* for the execution stage
- *MEM* for the memory access stage
- *WB* for the write-back stage

```
add    $s0, $t0, $t1
sub    $t2, $s0, $t3
```



FORWARDING (BYPASSING)

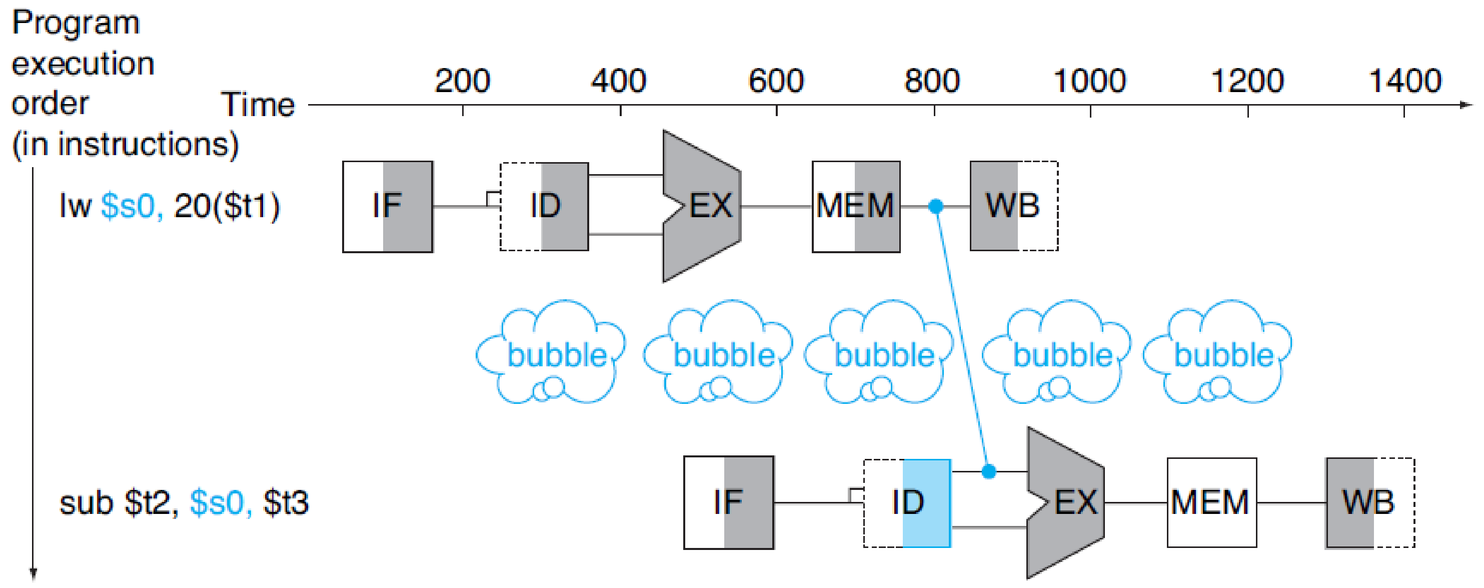


Program execution order (in instructions)
Time
200
400
600
800
1000
add $s0, $t0, $t1
IF
ID
EX
MEM
WB
sub $t2, $s0, $t3
IF
ID
EX
MEM
WB
Adding extra hardware to retrieve the missing item early

- forwarding paths are valid only if the destination stage is later in time than the source stage.

FORWARDING (BYPASSING)

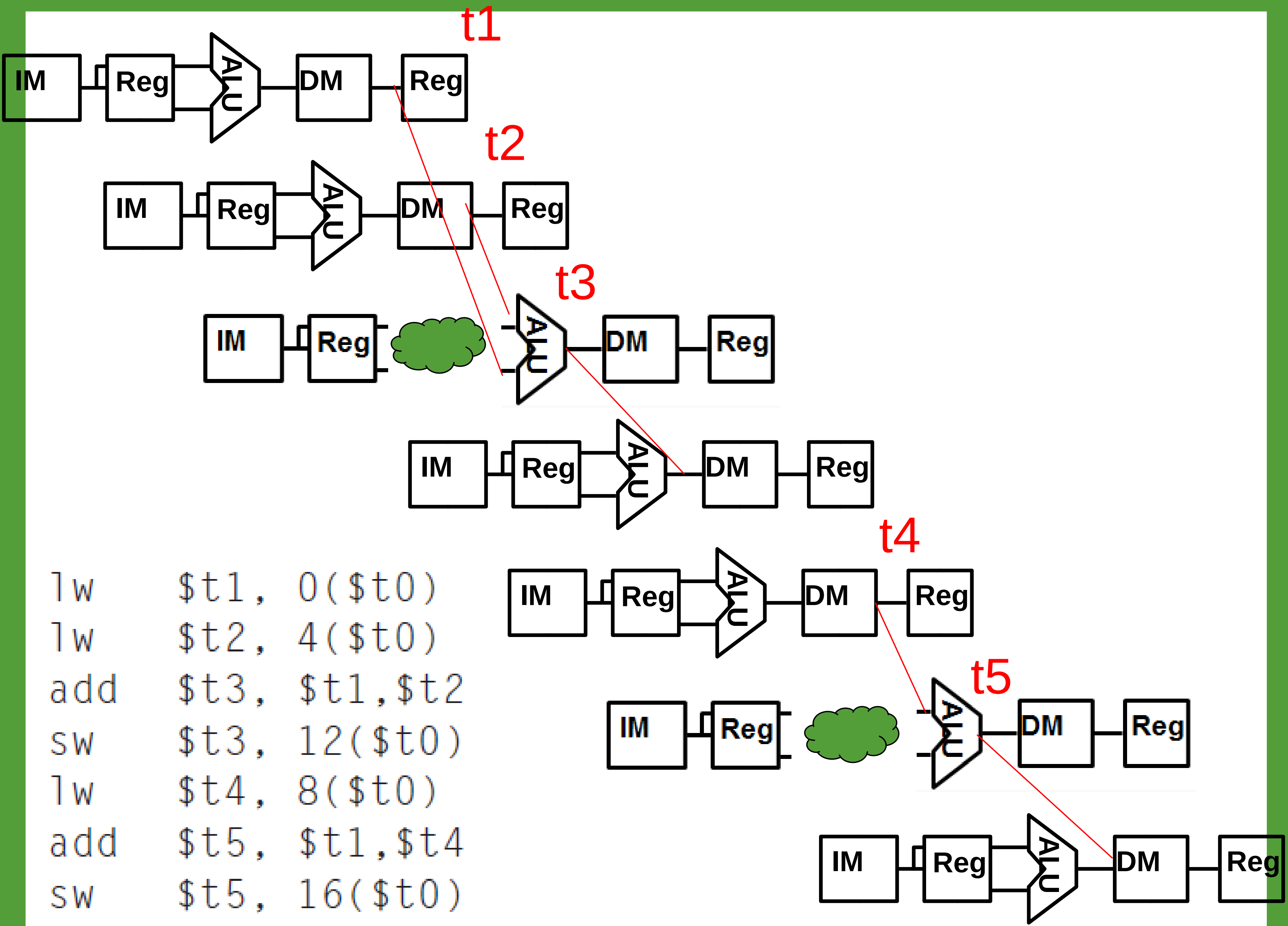
- Forwarding works very well, but it cannot prevent all pipeline stalls.
- **load-use data hazard** = we would have to stall one stage.
- We call **pipeline stall** or **bubble**.



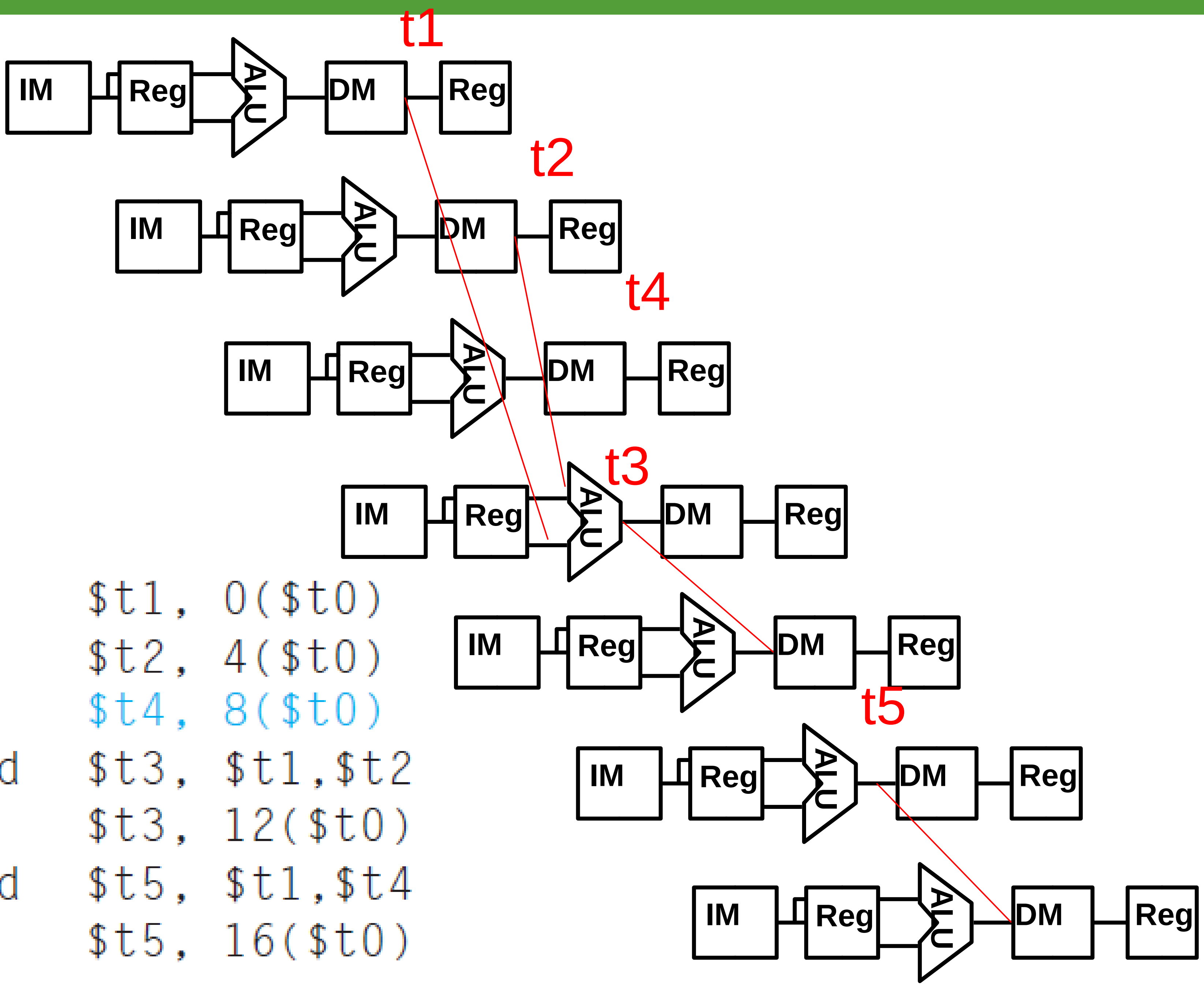
REORDERING CODE

- We can reordering code to avoid pipeline stalls
- Ex : assuming all variables are in memory and are addressable as offsets from $to

```
a = b + e;
c = b + f;
```



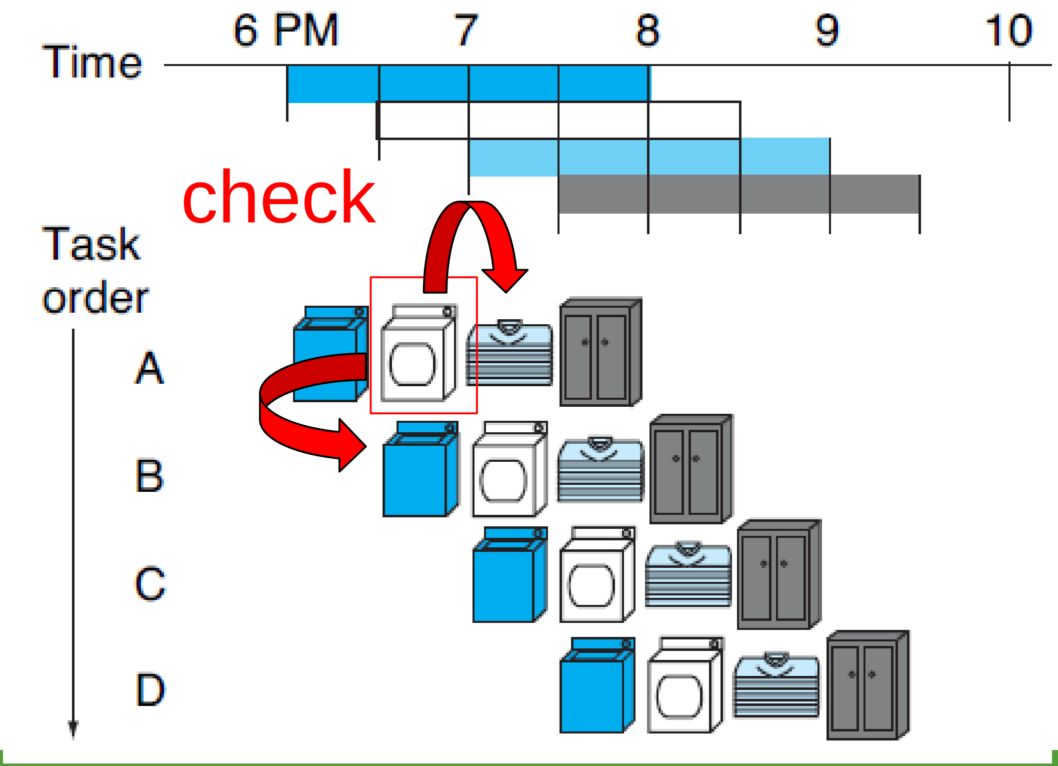
```
lw    $t1, 0($t0)
lw    $t2, 4($t0)
add   $t3, $t1,$t2
sw    $t3, 12($t0)
lw    $t4, 8($t0)
add   $t5, $t1,$t4
sw    $t5, 16($t0)
```



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

CONTROL HAZARDS

- Arising from the need to ***make a decision*** based on the results of one instruction while others are executing.
- **branch instruction**

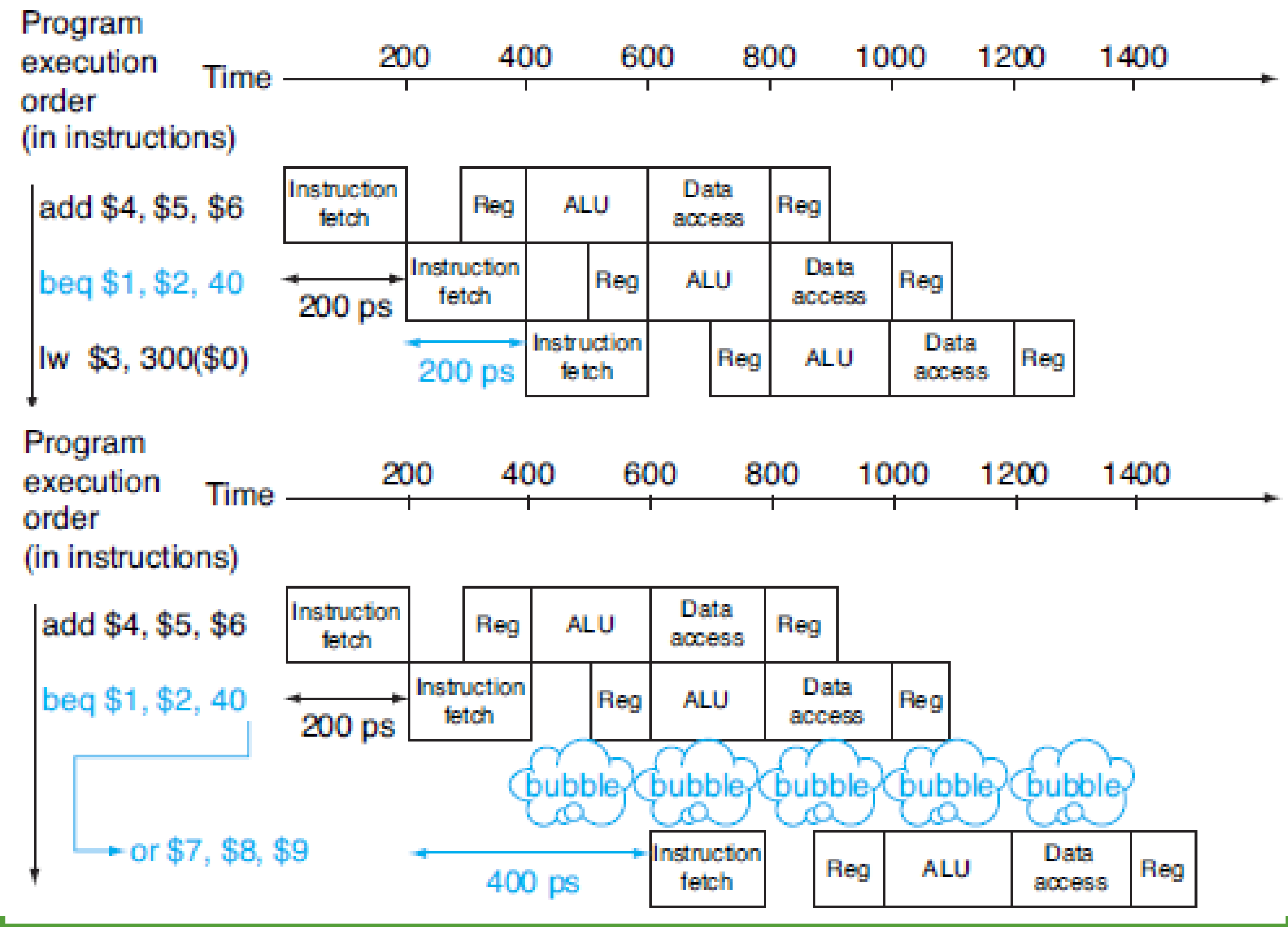


Wash
Dry
Fold
Keep

CONTROL HAZARDS

- Nevertheless, the pipeline cannot possibly know what the next instruction should be, since it *only just received* the branch instruction from memory!
- one possible solution is to stall immediately after we fetch a branch, waiting until the pipeline determines the outcome of the branch and knows what instruction address to fetch from.

- Let’s assume that we put in enough extra hardware so that we can test registers, calculate the branch address, and update the PC during the second stage of the pipeline


Program execution order (in instructions)
Time
200
400
600
800
1000
1200
1400
add $4, $5, $6
Instruction fetch
Reg
ALU
Data access
Reg
beq $1, $2, 40
200 ps
lw $3, 300($0)
200 ps
Program execution order (in instructions)
Time
add $4, $5, $6
beq $1, $2, 40
200 ps
bubble
bubble
bubble
bubble
bubble
or $7, $8, $9
400 ps