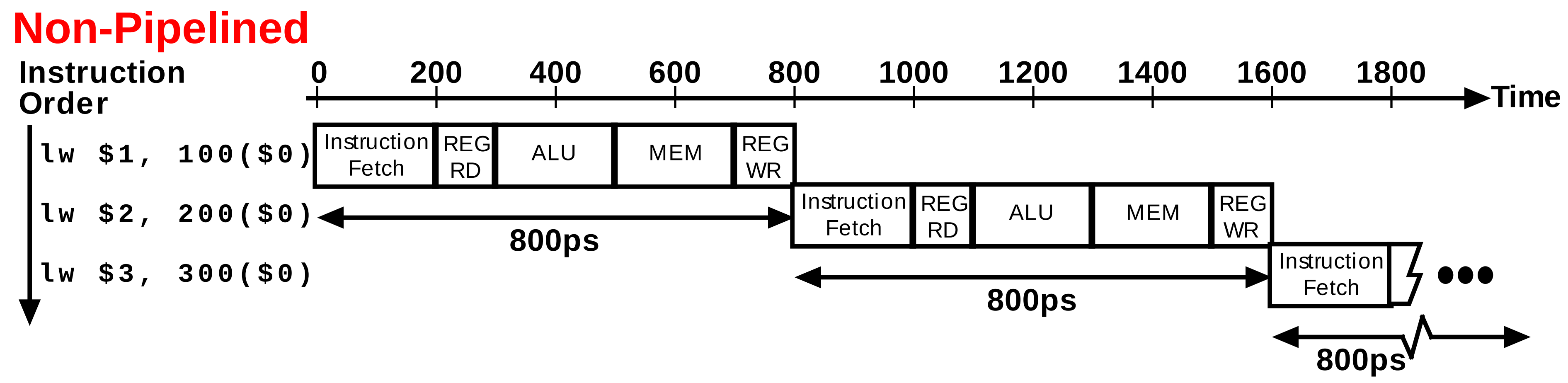
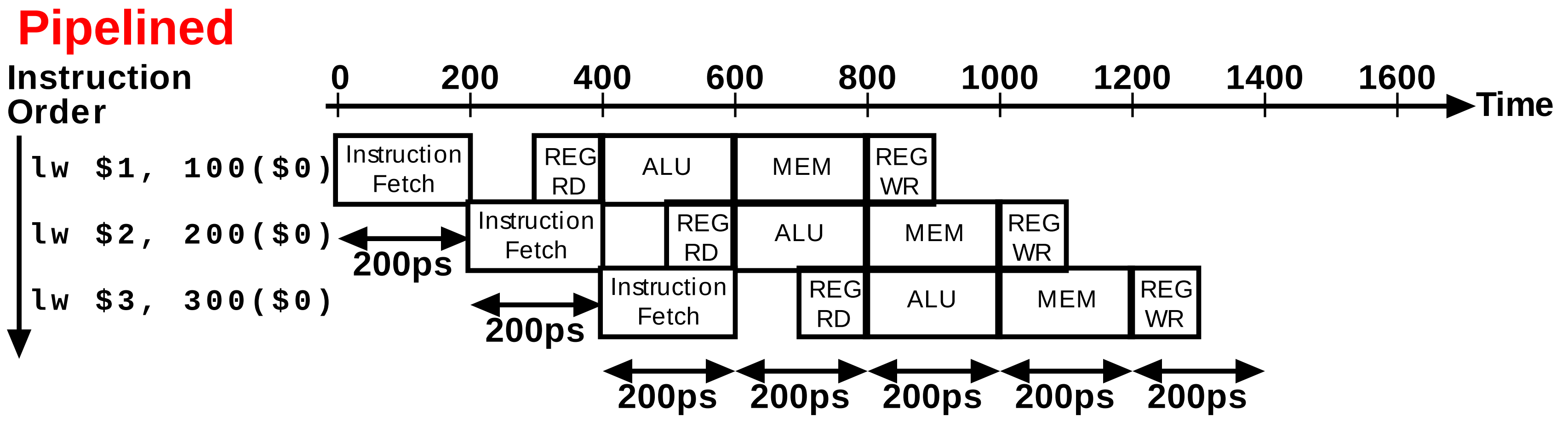
Pipeline: Hazards

Fall, 2017

These slides are adapted from notes by Dr. David Patterson (UCB)

Single-Cycle vs. Pipelined Execution





Speedup

- Consider the unpipelined processor introduced previously. Assume that it has a 1 ns clock cycle and it uses 4 cycles for ALU operations and branches, and 5 cycles for memory operations, assume that the relative frequencies of these operations are 40%, 20%, and 40%, respectively. Suppose that due to clock skew and setup, pipelining the processor adds 0.2ns of overhead to the clock. Ignoring any latency impact, how much speedup in the instruction execution rate will we gain from a pipeline?

Average instruction execution time
  = 1 ns * ((40% + 20%)*4 + 40%*5)
  = 4.4ns

Speedup from pipeline
  = Average instruction time unpiplined/Average instruction time pipelined
  = 4.4ns/1.2ns = 3.7

Comments about Pipelining

- The good news
  - Multiple instructions are being processed at same time
  - This works because stages are isolated by registers
  - Best case speedup of N
- The bad news
  - Instructions interfere with each other - hazards
    - Example: different instructions may need the same piece of hardware (e.g., memory) in same clock cycle
    - Example: instruction may require a result produced by an earlier instruction that is not yet complete

Pipeline Hazards

- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - Structural hazards: two different instructions use same h/w in same cycle
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline
  - Control hazards: Pipelining of branches & other instructions that change the PC

Structural Hazards

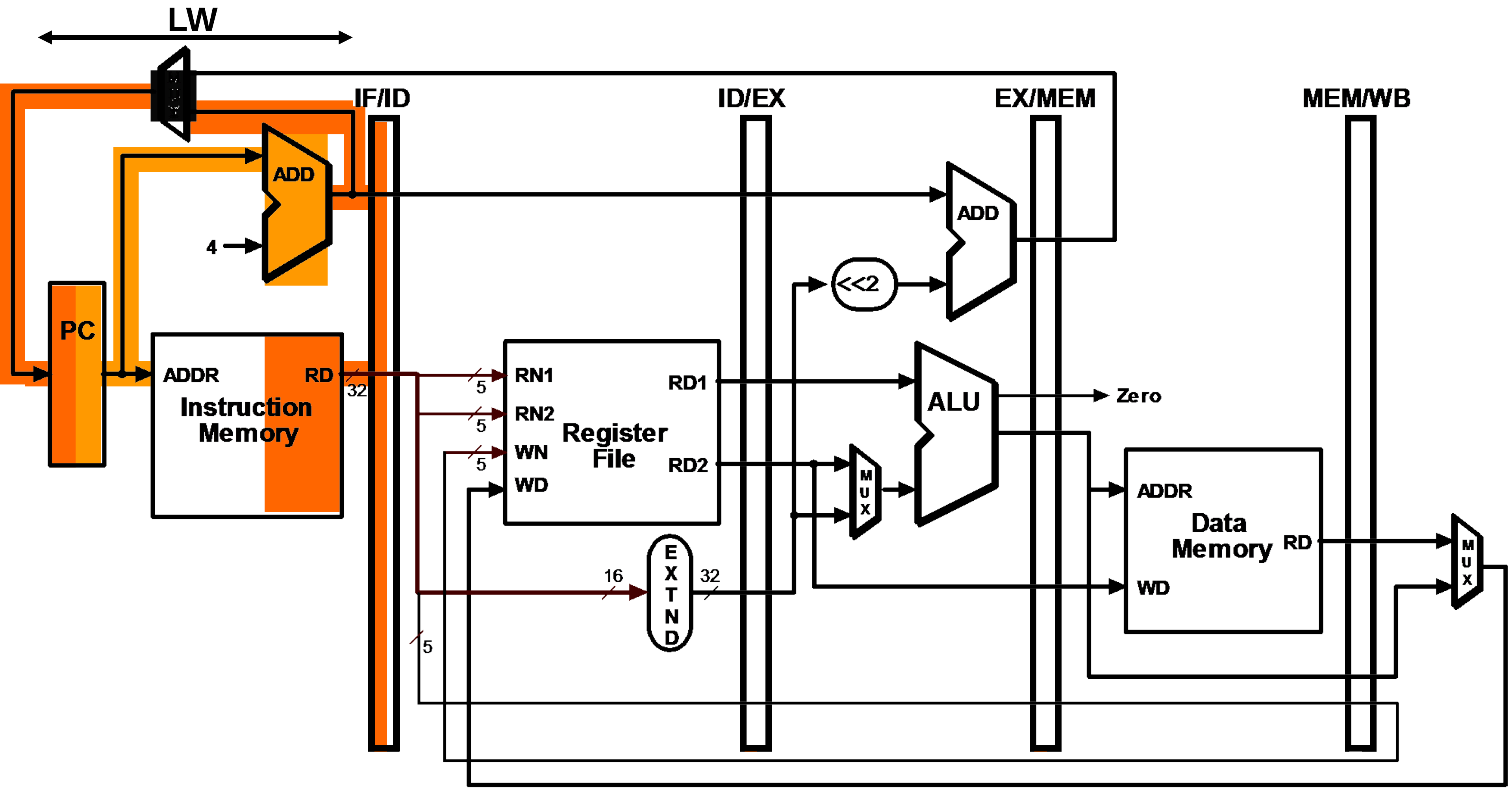
- Attempt to use same resource twice at same time
- Example: Single Memory for instructions, data
  - Accessed by IF stage
  - Accessed at same time by MEM stage
- Solutions ?
  - Delay second access by one clock cycle
  - Provide separate memories for instructions, data
    - This is what the book does
    - This is called a "Harvard Architecture"
    - Real pipelined processors have separate caches

Pipelined Example - Executing Multiple Instructions

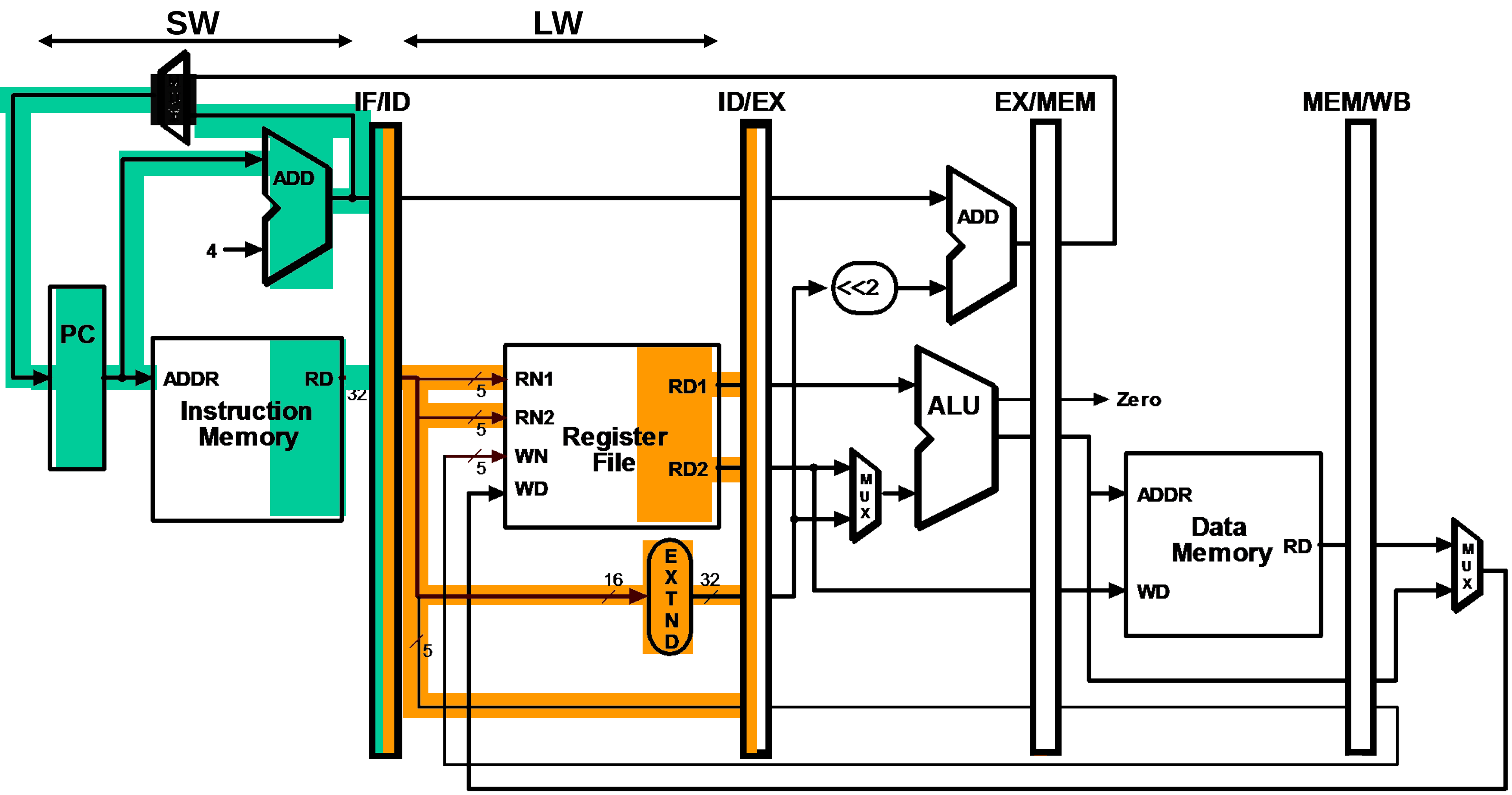
- Consider the following instruction sequence:

```
lw $r0, 10($r1)
sw $sr3, 20($r4)
add $r5, $r6, $r7
sub $r8, $r9, $r10
```

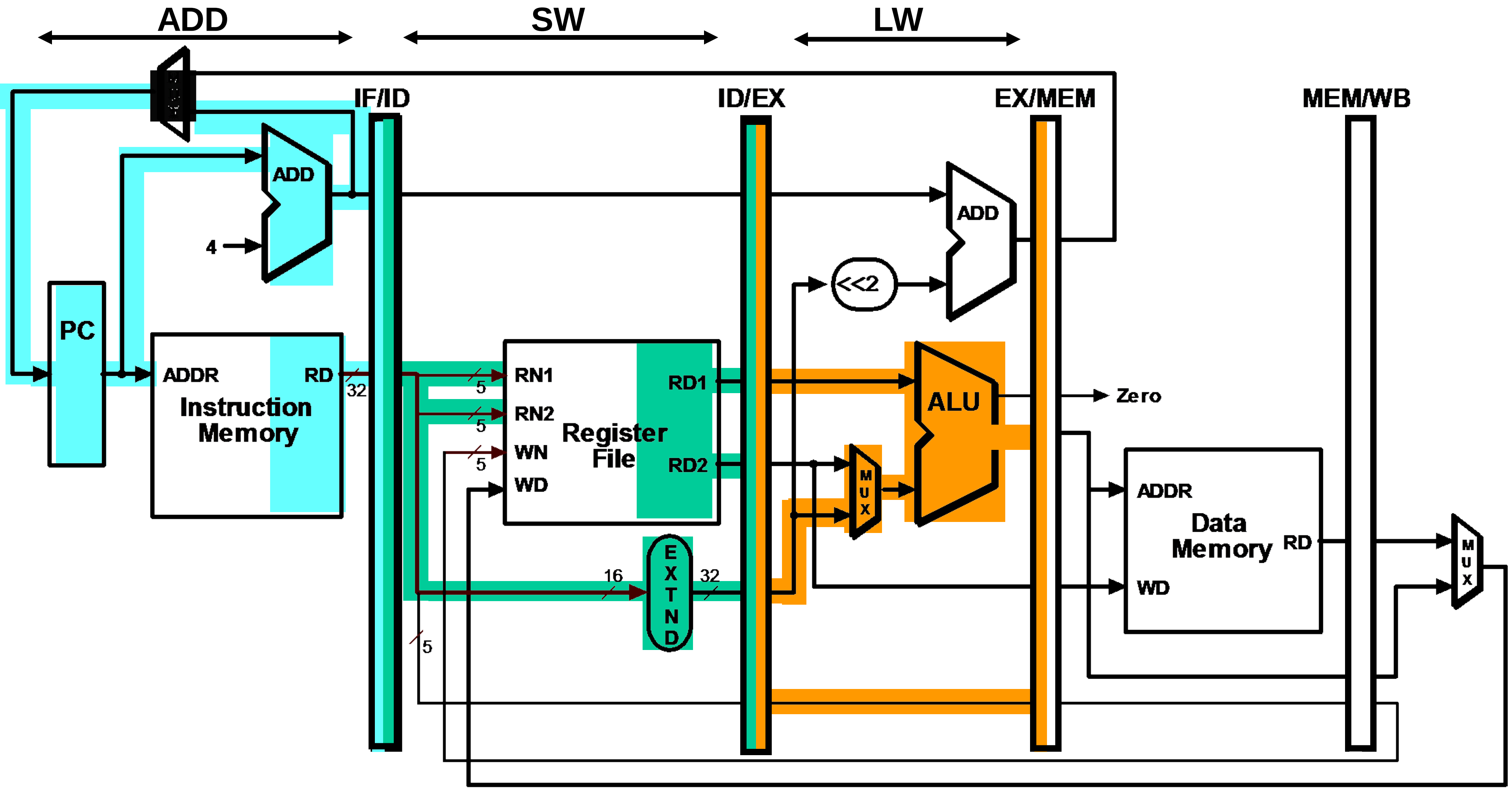
Executing Multiple Instructions
Clock Cycle 1



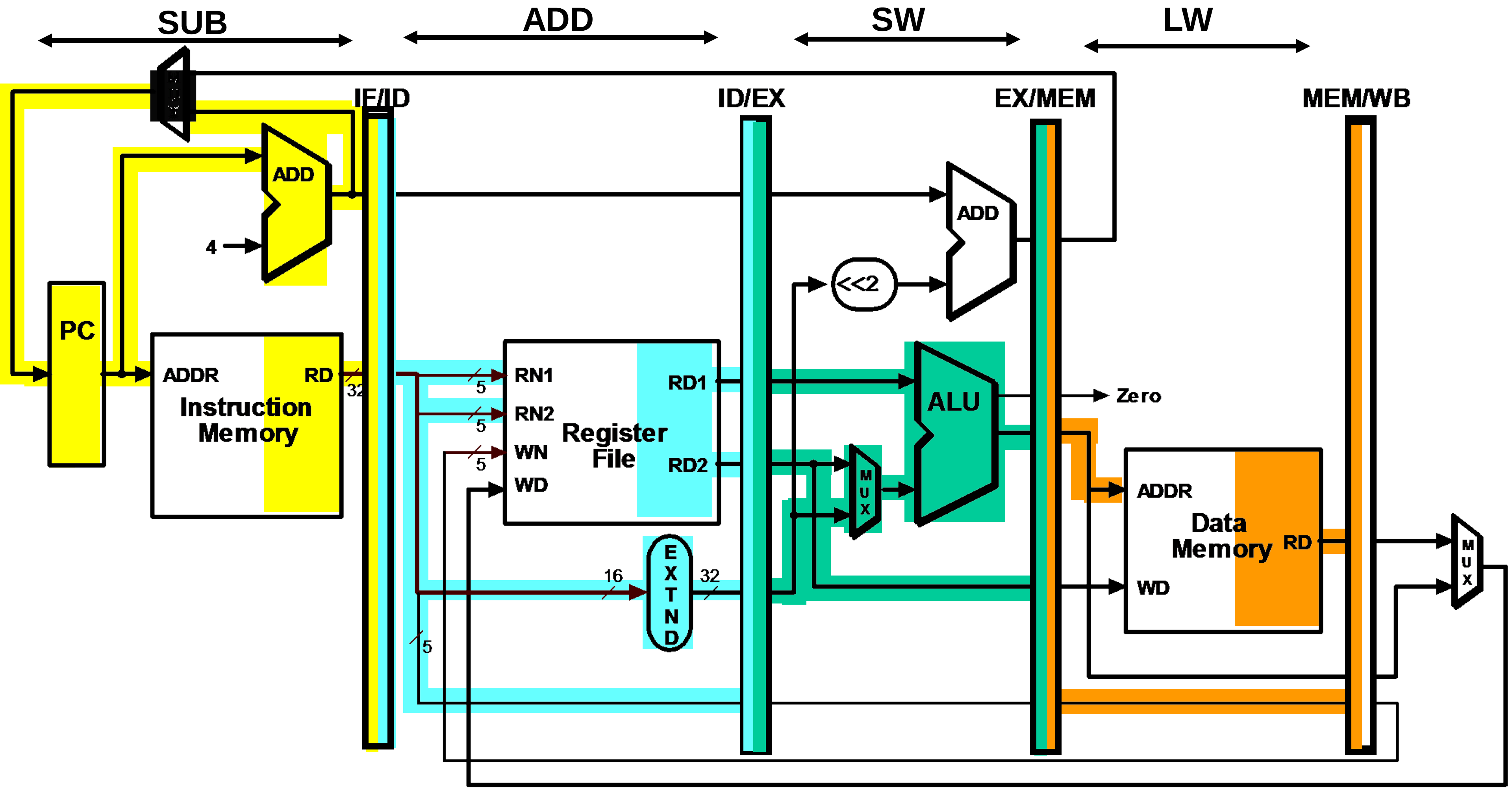
Executing Multiple Instructions Clock Cycle 2



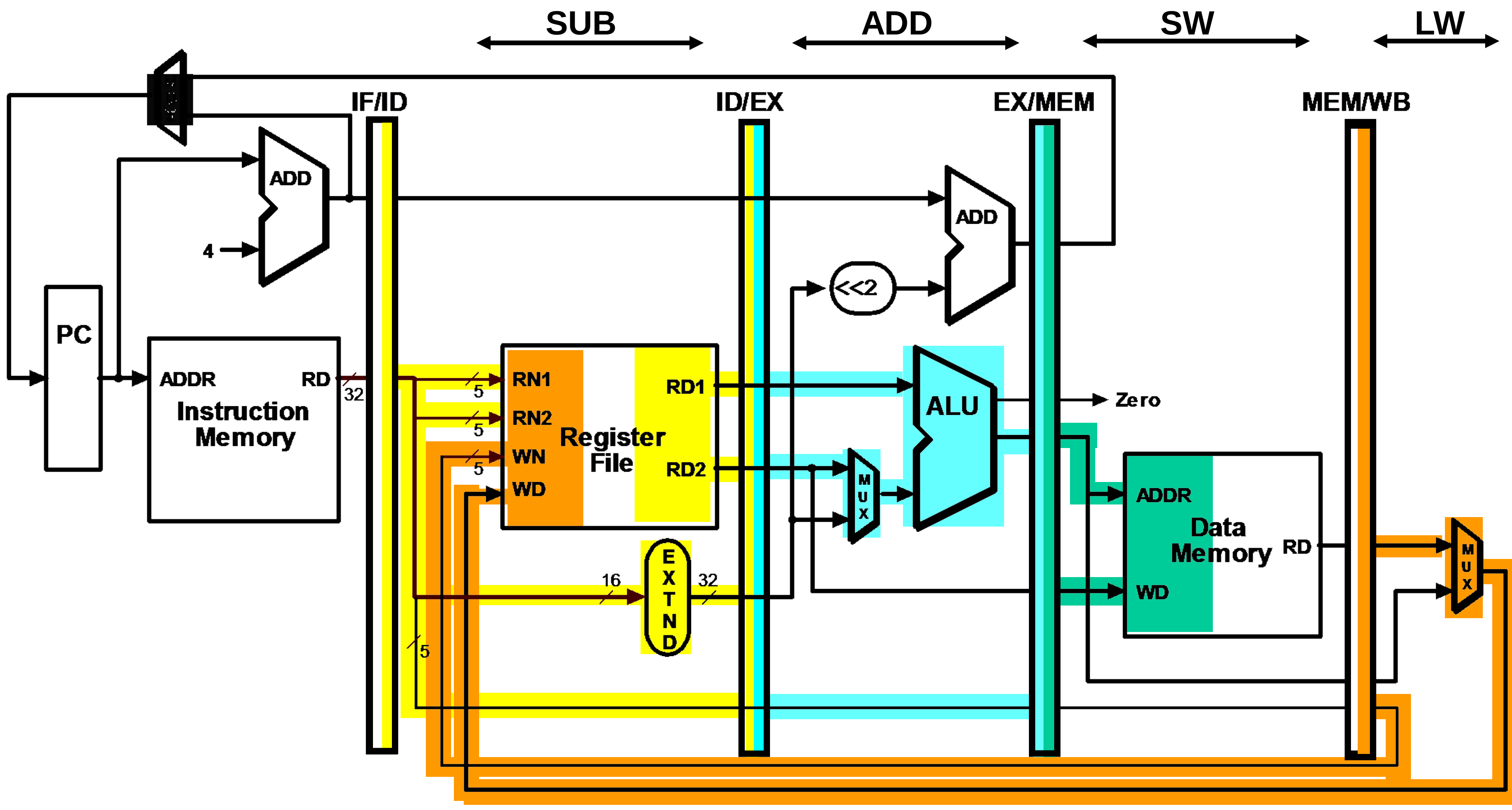
Executing Multiple Instructions Clock Cycle 3



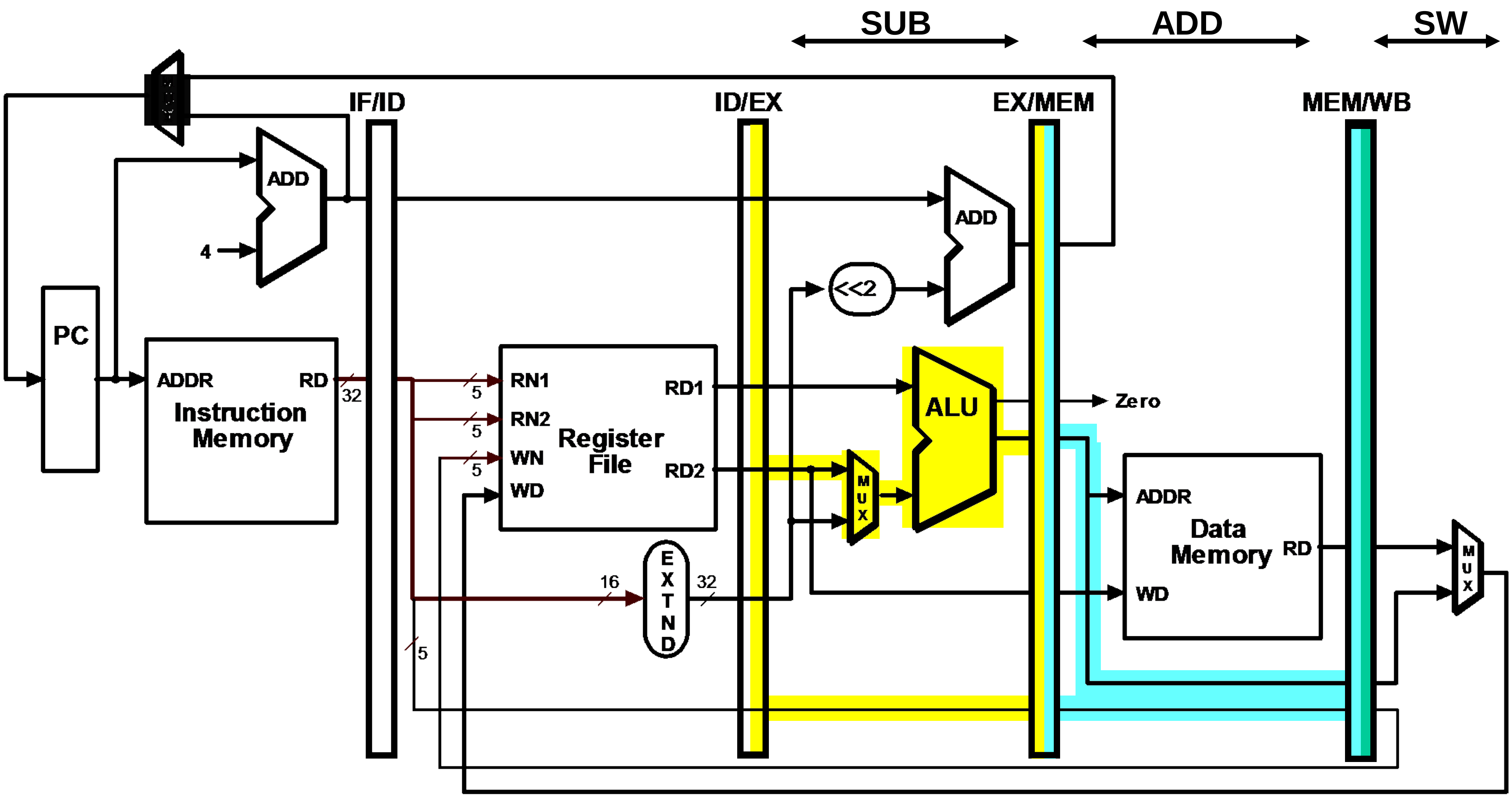
Executing Multiple Instructions Clock Cycle 4



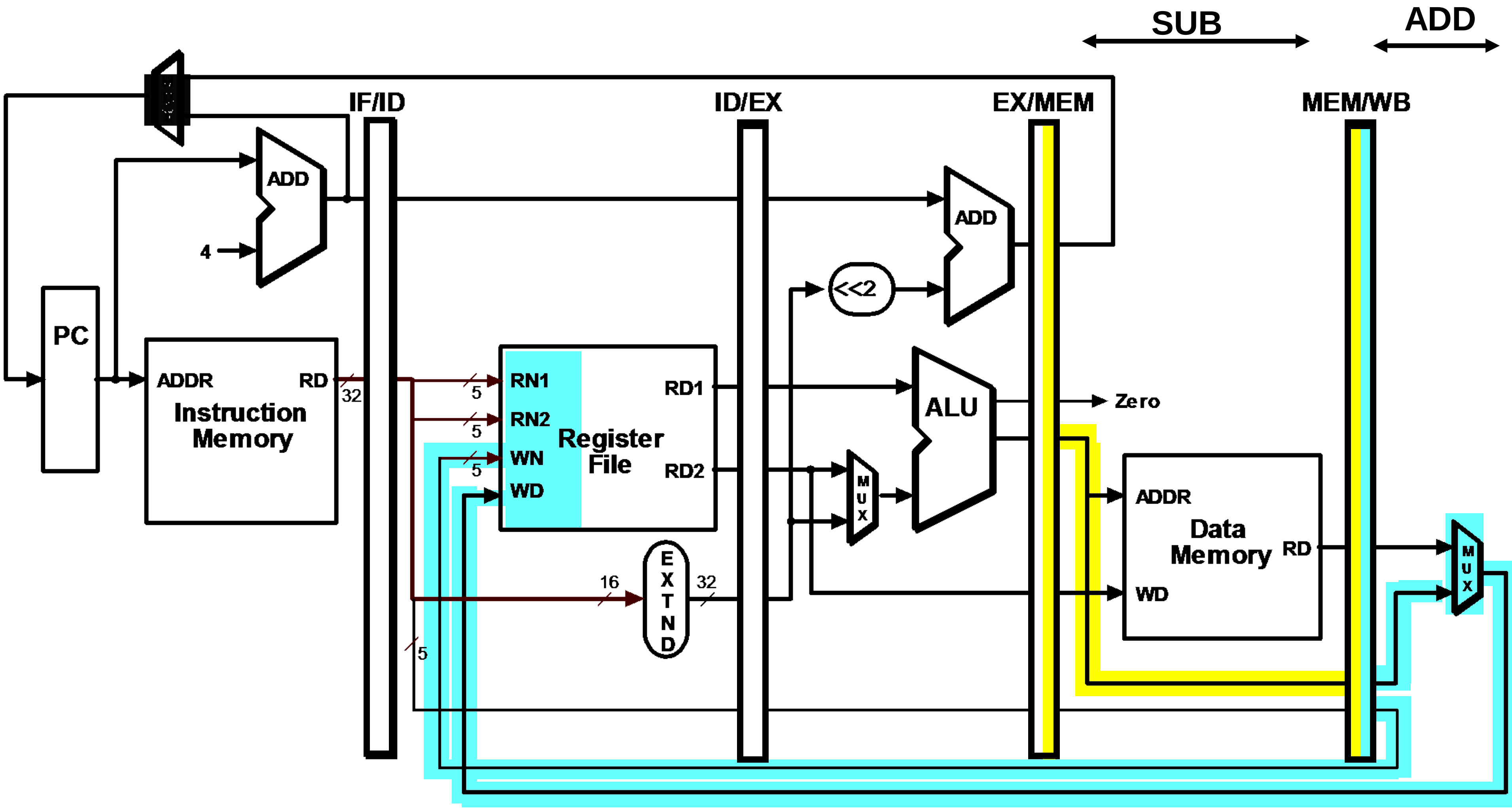
Executing Multiple Instructions
Clock Cycle 5



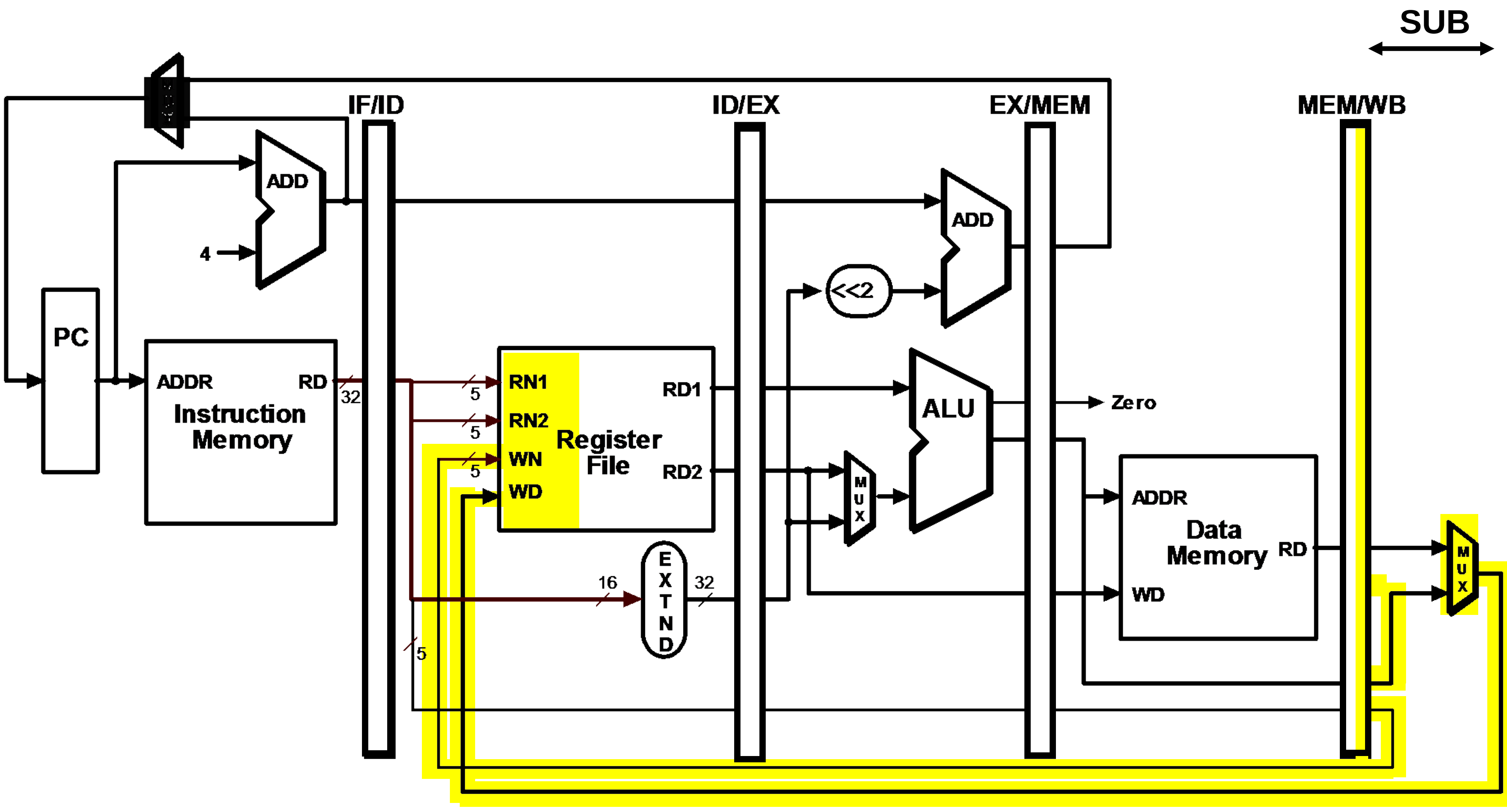
Executing Multiple Instructions
Clock Cycle 6



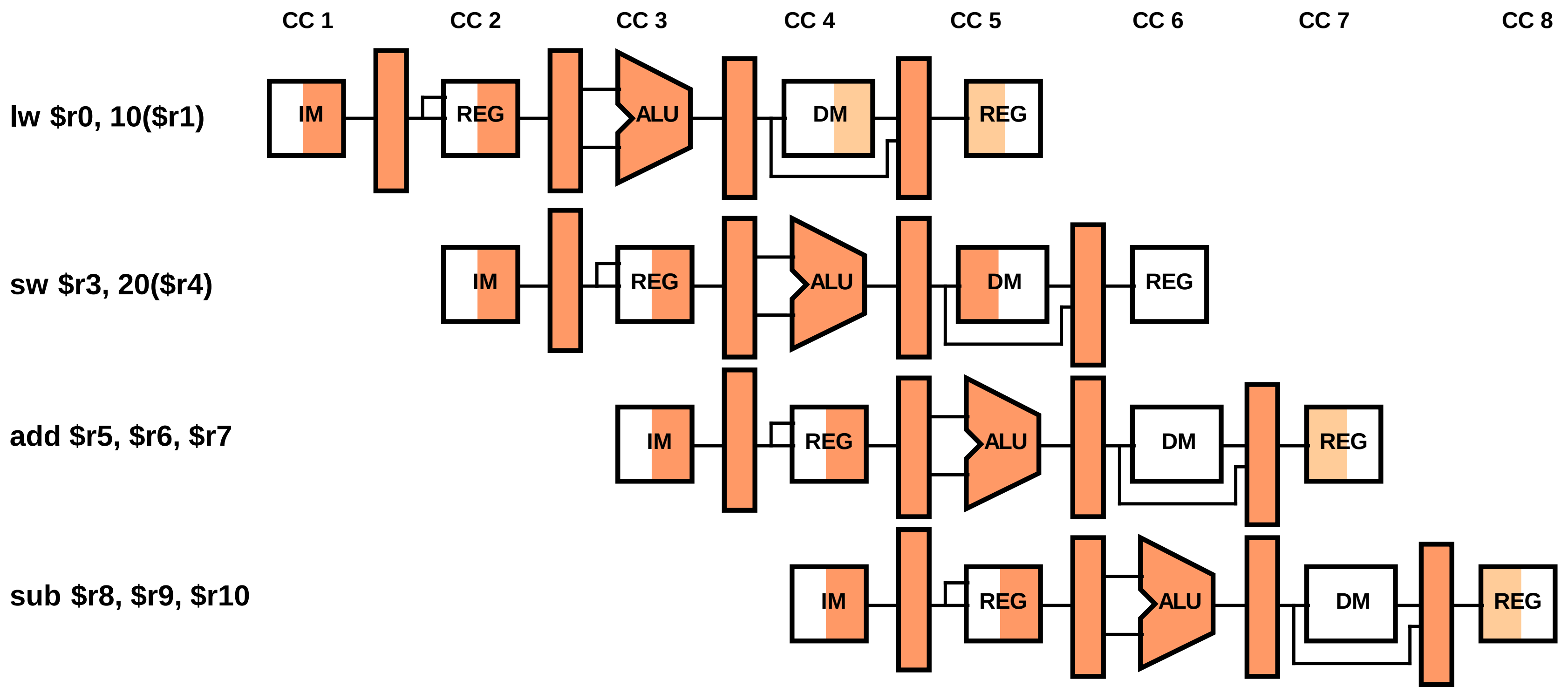
Executing Multiple Instructions Clock Cycle 7



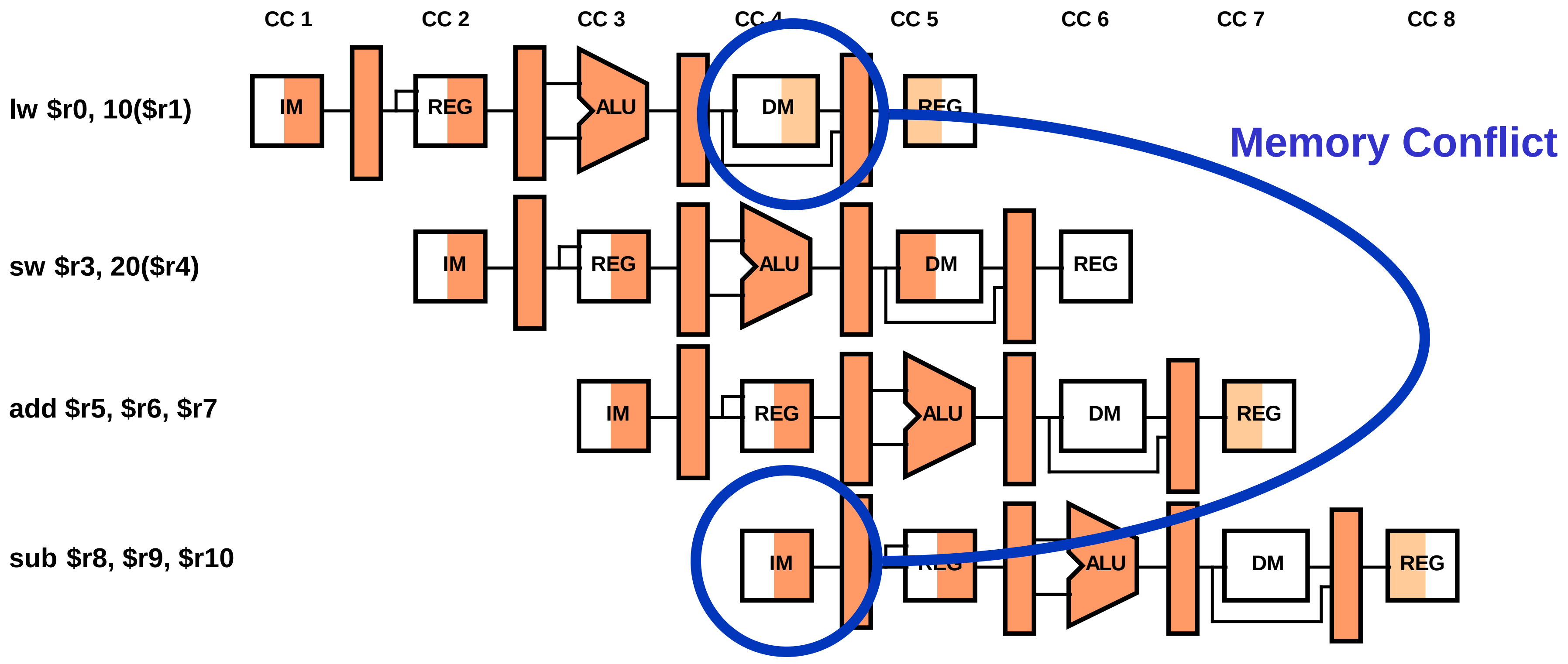
Executing Multiple Instructions Clock Cycle 8



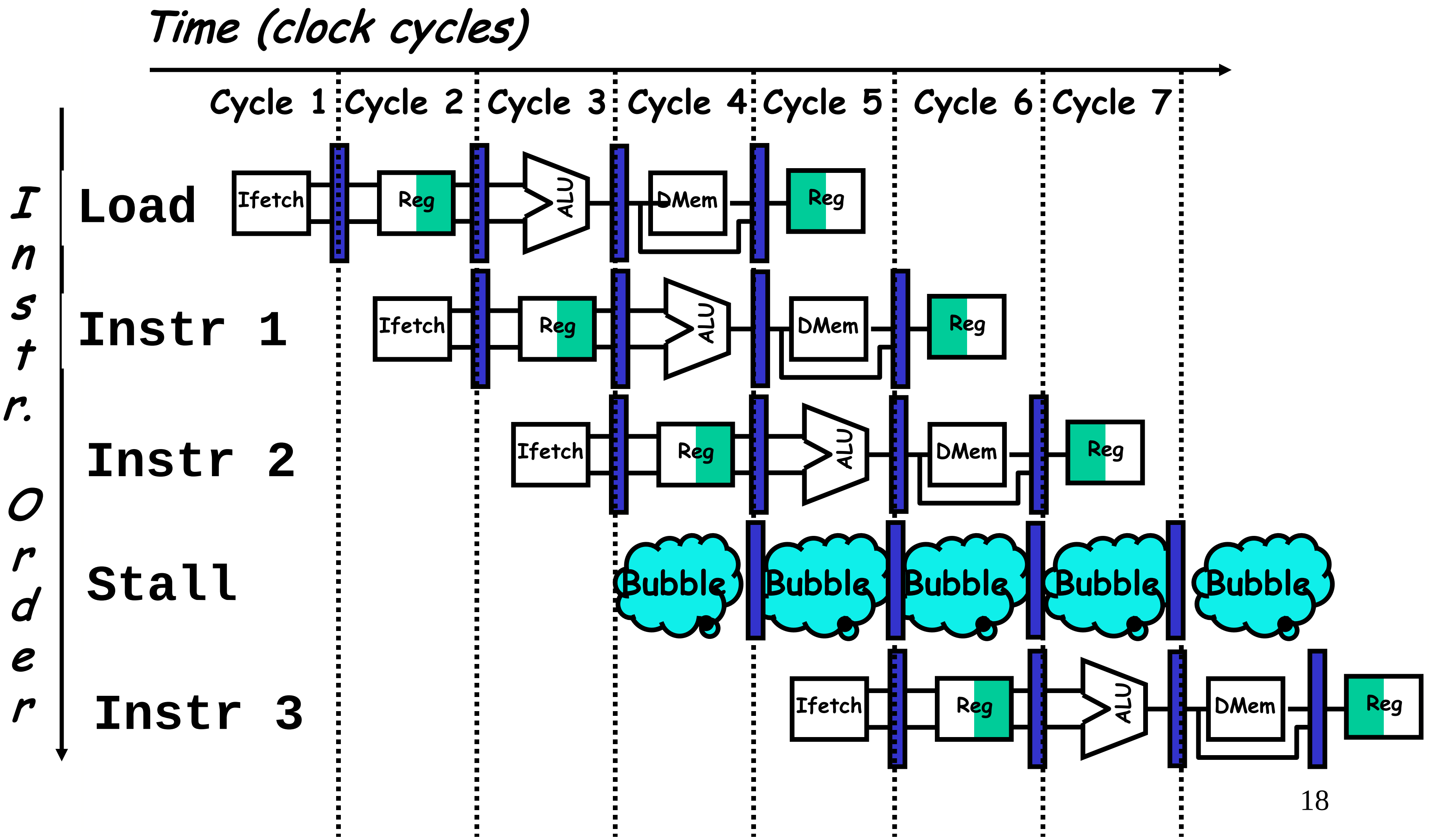
Alternative View - Multicycle Diagram



Alternative View - Multicycle Diagram



One Memory Port Structural Hazards



Structural Hazards

Some Common Structural Hazards:

- Memory:
  - we’ve already mentioned this one.
- Floating point:
  - Since many floating point instructions require many cycles, it’s easy for them to interfere with each other.
- Starting up more of one type of instruction than there are resources.
  - For instance, the PA-8600 can support two ALU + two load/store instructions per cycle - that’s how much hardware it has available.

Dealing with Structural Hazards

Stall
- low cost, simple
- Increases CPI
- use for rare case since stalling has performance effect

Pipeline hardware resource
- useful for multi-cycle resources
- good performance
- sometimes complex e.g., RAM

Replicate resource
- good performance
- increases cost (+ maybe interconnect delay)
- useful for cheap or divisible resources

Structural Hazards

- Structural hazards are reduced with these rules:
  - Each instruction uses a resource at most once
  - Always use the resource in the same pipeline stage
  - Use the resource for one cycle only
- Many RISC ISA’s designed with this in mind
- Sometimes very complex to do this.
  - For example, memory of necessity is used in the IF and MEM stages.

Structural Hazards

We want to compare the performance of two machines. Which machine is faster?

- Machine A: Dual ported memory - so there are no memory stalls
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate

Assume:

- Ideal CPI = 1 for both
- Loads are 40% of instructions executed

Speedup from Pipelining

$$\text{Speedup from pipelining} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}$$

$$= \frac{\text{CPI}_{\text{unpipelined}} \times \text{Clock cycle}_{\text{unpipelined}}}{\text{CPI}_{\text{pipelined}} \times \text{Clock cycle}_{\text{pipelined}}}$$

$\text{CPI}_{\text{pipelined}} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction}$

$\text{CPI}_{\text{unpipelined}} = \text{Ideal CPI} \times \text{Pipeline depth}$

Speed Up Equations for Pipelining

$$\text{CPI}_{\text{pipelined}} = \text{Ideal CPI} + \text{Average Stall cycles per Inst}$$

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$

For simple RISC pipeline, the Ideal CPI on a pipelined processor = 1:

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$

Structural Hazards

We want to compare the performance of two machines. Which machine is faster?

- Machine A: Dual ported memory - so there are no memory stalls
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate

Assume:

- Ideal CPI = 1 for both
- Loads are 40% of instructions executed

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$

$$\text{SpeedUp}_A = \text{Pipeline Depth}/(1 + 0) \text{ x}(\text{clock}_{\text{unpipe}}/\text{clock}_{\text{pipe}})$$
$$= \text{Pipeline Depth}$$

$$\text{SpeedUp}_B = \text{Pipeline Depth}/(1 + 0.4 \text{ x } 1)$$
$$\text{x } (\text{clock}_{\text{unpipe}}/(\text{clock}_{\text{unpipe}} / 1.05)$$
$$= (\text{Pipeline Depth}/1.4) \text{ x } 1.05$$
$$= 0.75 \text{ x Pipeline Depth}$$

$$\text{SpeedUp}_A / \text{SpeedUp}_B = \text{Pipeline Depth} / (0.75 \text{ x Pipeline Depth}) = 1.33$$

- Machine A is 1.33 times faster

Summary - Structural Hazards

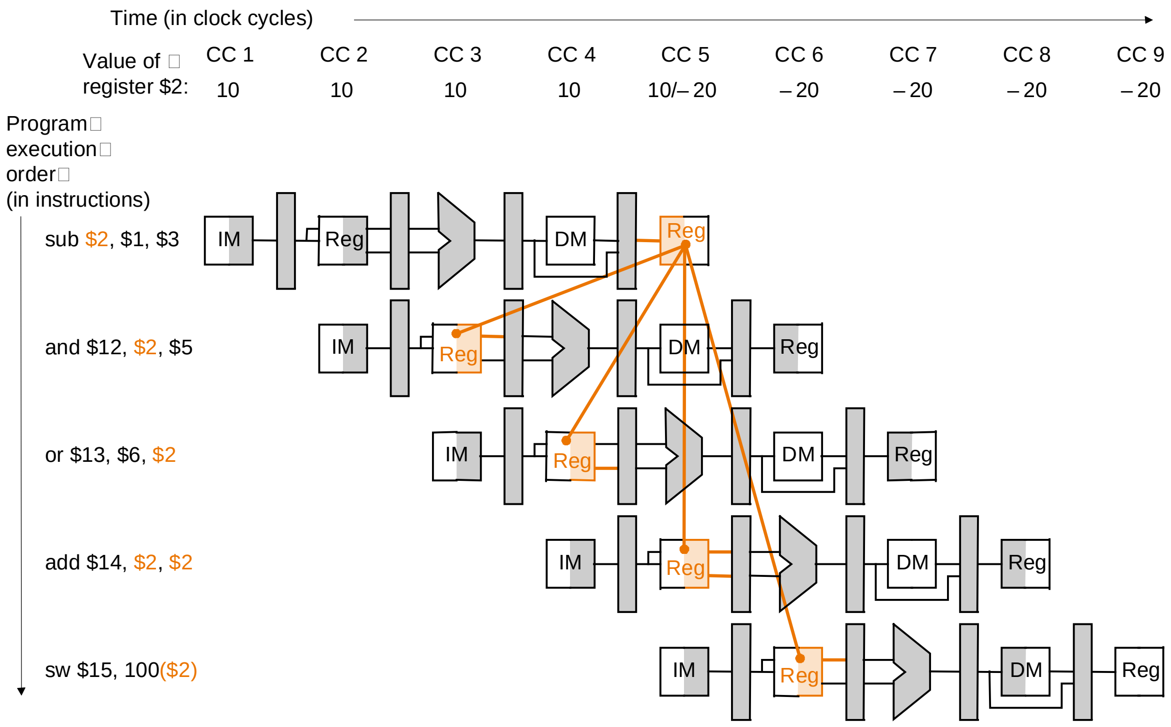
- Speed Up <= Pipeline Depth; if ideal CPI is 1, then:

$$\text{Speedup} = \frac{\text{Pipeline Depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Clock Cycle Unpipelined}}{\text{Clock Cycle Pipelined}}$$

- Hazards limit performance on computers:
  - Structural: need more HW resources
  - Data (RAW,WAR,WAW):
  - Control

Data Hazards

- Data hazards occur when data is used before it is stored



The use of the result of the SUB instruction in the next three instructions causes a data hazard, since the register is not written until after those instructions read it.

Data Hazards

Execution Order is:
Instr$_I$
Instr$_J$

Read After Write (RAW)

Instr$_J$ tries to read operand before Instr$_I$ writes it

```
I: add r1,r2,r3
J: sub r4,r1,r3
```

- Caused by a "Dependence" (in compiler nomenclature). This hazard results from an actual need for communication.

Data Hazards

Execution Order is:
Instr_I
Instr_J

Write After Read (WAR)

Instr_J tries to write operand *before* Instr_I reads i

- Gets wrong operand

```
I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- Called an “anti-dependence” by compiler writers. This results from reuse of the name “r1”.

- Can’t happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Reads are always in stage 2, and
  - Writes are always in stage 5

Data Hazards

Execution Order is:
Instr_I
Instr_J

Write After Write (WAW)

Instr_J tries to write operand *before* Instr_I writes it

– Leaves wrong result (Instr_I not Instr_J)

```
I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- Called an “output dependence” by compiler writers
  This also results from the reuse of name “r1”.
- Can’t happen in MIPS 5 stage pipeline because:
  –All instructions take 5 stages, and
  – Writes are always in stage 5

•Will see WAR and WAW in later more complicated pipes