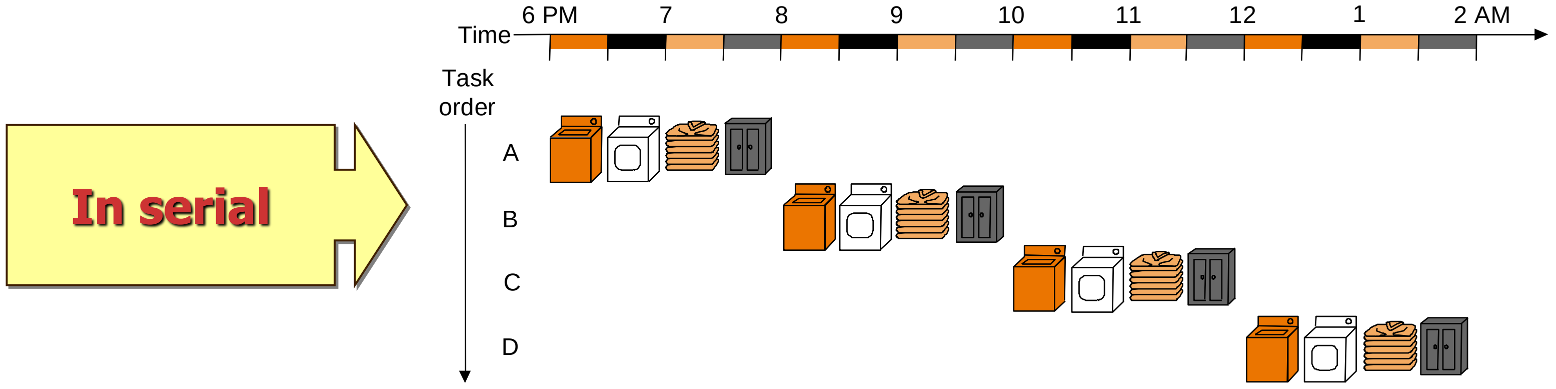
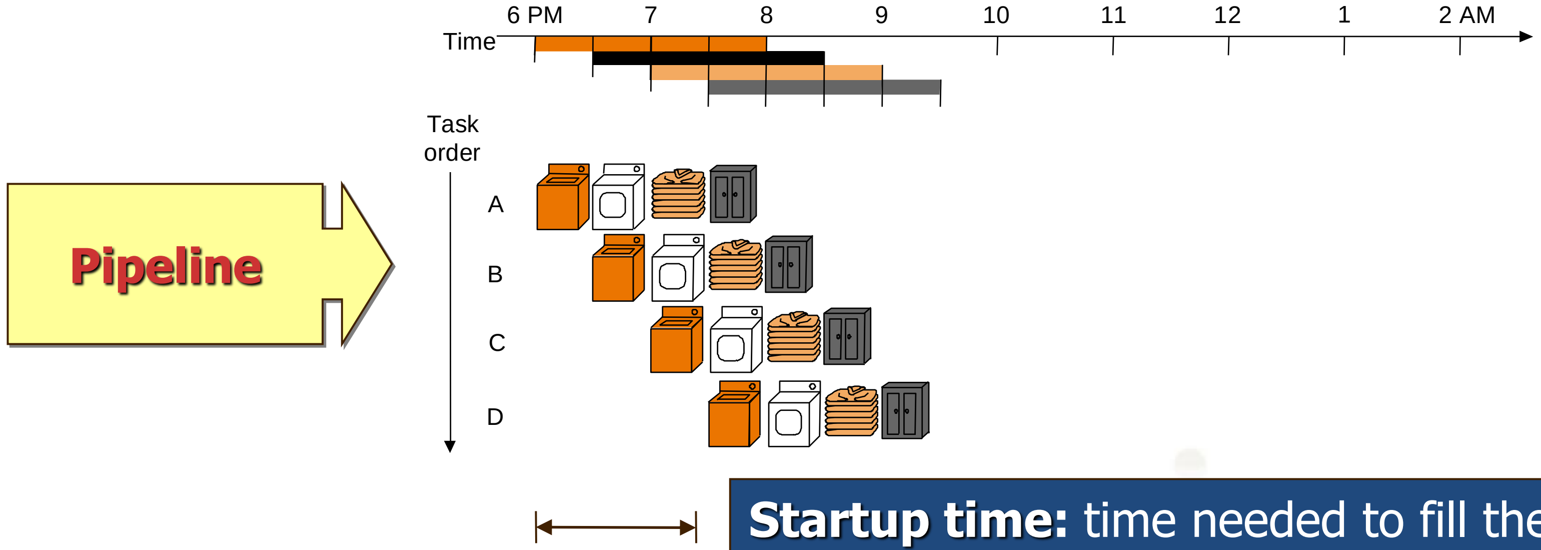
COMP2611 COMPUTER ORGANIZATION
PIPELINED PROCESSOR

Pipelining Analogy

- Pipelined laundry: overlapping execution with parallelism





Startup time: time needed to fill the pipeline

Pipelining Principles

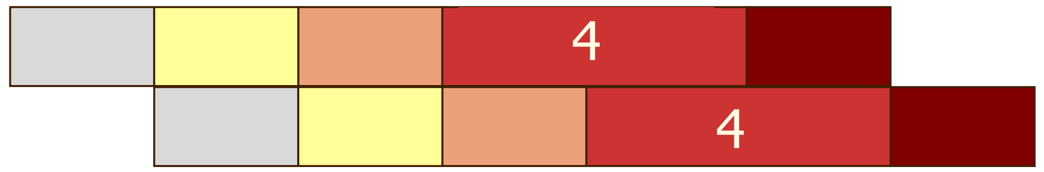
Key characteristics:

- **Multiple tasks** are processed simultaneously
- Ideally, these tasks should be **independent** of each other otherwise we need to make this the case
- Pipelining **does not help the latency** of a single task
- It **helps the throughput** of the entire workload
- Completion order in pipelined execution **=** that in sequential execution

How much can a pipeline improve?

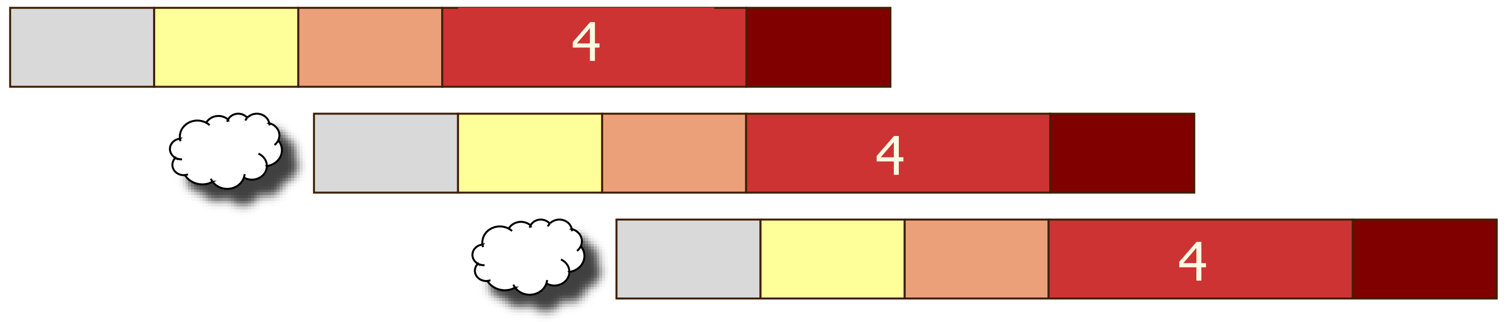
- **Potential speedup** = **number of pipeline stages**
- The pipeline rate is limited by the slowest pipeline stage
- ➢ Unbalanced lengths of pipeline stages can reduce speedup. Why?

Example: Bottleneck in Pipelining



- Can I align the pipeline stages as above?
- Answer: NO, because the tasks executing in parallel are not independent (task 4 overlaps task 4)

- The condition to align is to make sure NO OVERLAP of any stages?



THE MIPS PIPELINE

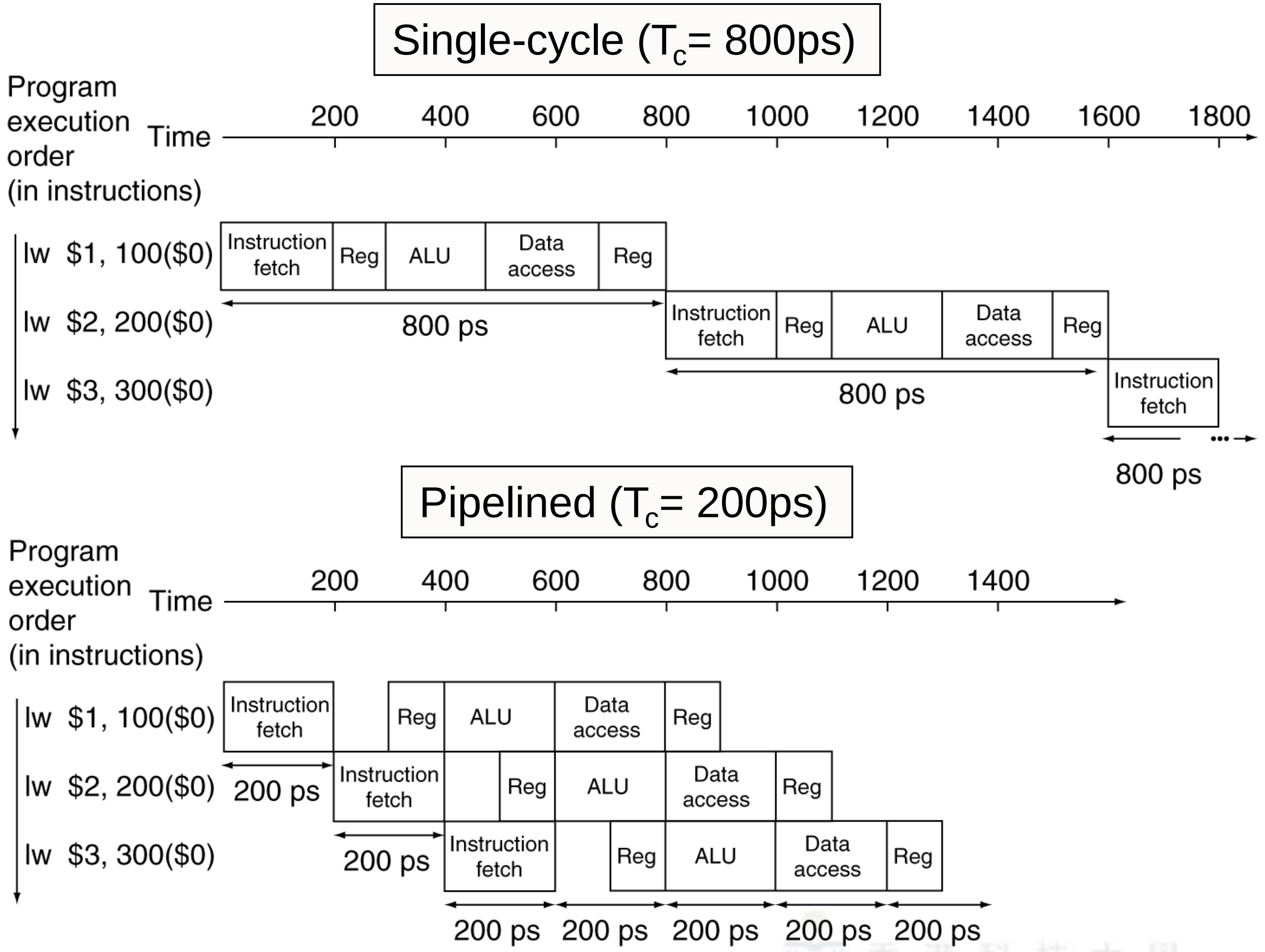
Pipeline Performance Example

- Assume time for stages is
  - 100 picoseconds for register read or write
  - 200 picoseconds for all other stages

- Compare pipelined datapath with single-cycle datapath

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

Pipeline Performance



Pipeline Speedup

- Pipeline speed is limited by the most time consuming pipeline stage, as this stage determines the duration of a clock cycle (why?)

- If all stages take the same time, the pipeline is well balanced

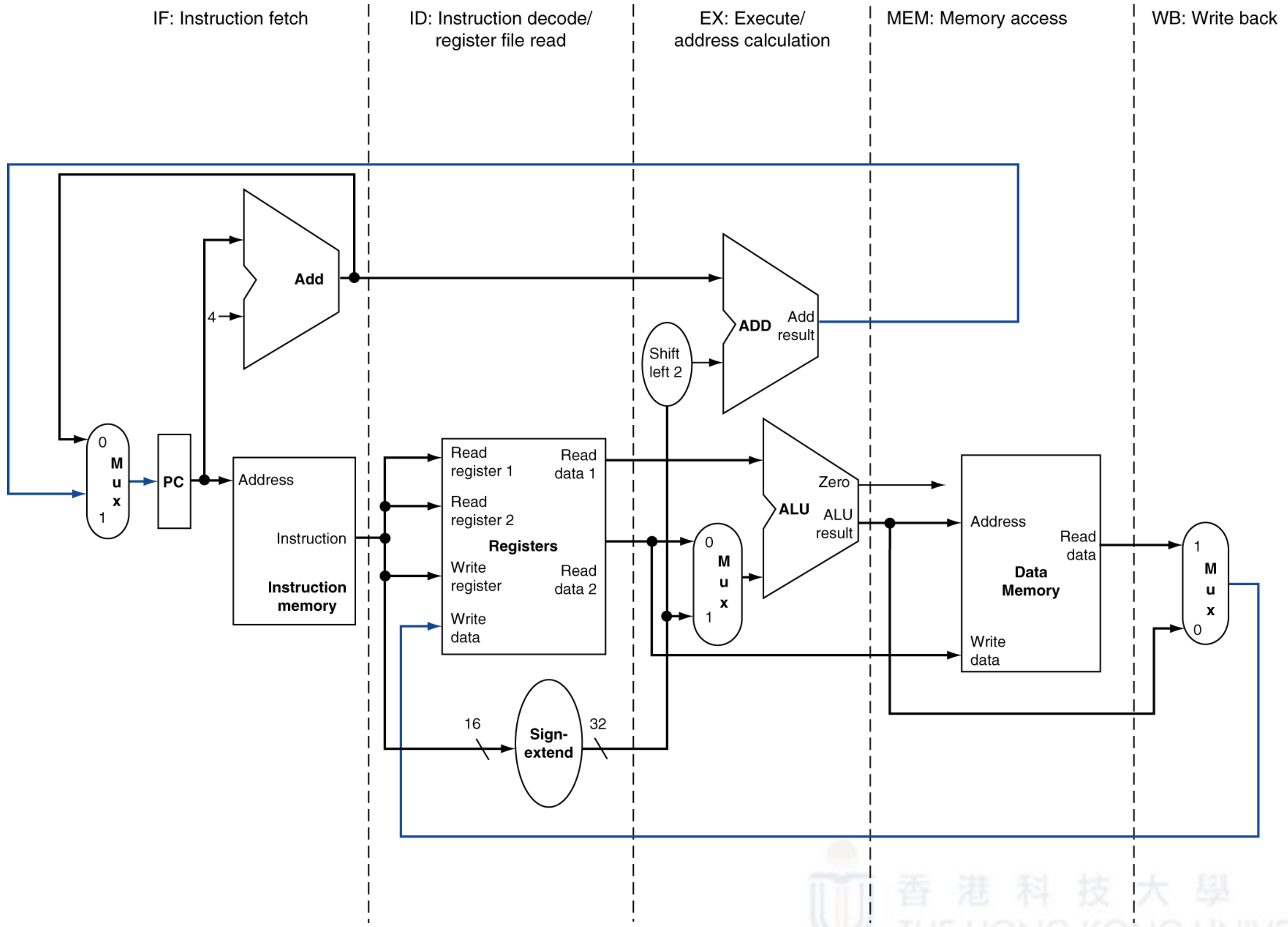
$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$$

- The speedup equals to Number of pipeline stages (a.k.a depth of the pipeline)
- If not balanced, speedup is less

- Pipelining does not improve the latency of a single instruction, it improves the throughput of the system (i.e., the datapath)

MIPS Pipelined Datapath

- Basic idea: take a single-cycle datapath and separate it into 5 pieces. Each piece responsible for a single instruction execution stage.



MIPS ISA for Pipelining

ISA design affects the complexity of pipeline implementation.

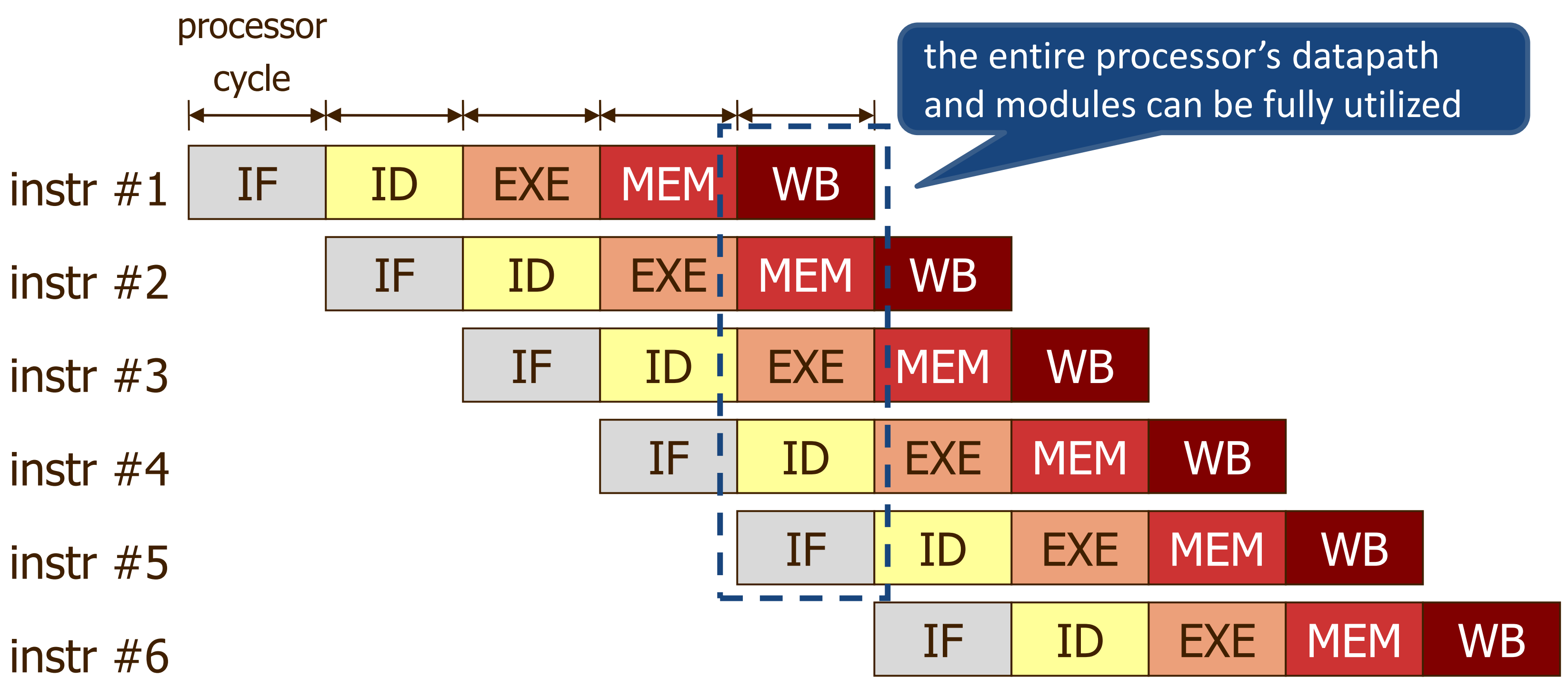
MIPS ISA is designed for pipelining

- **All instruction are of the same length (32-bit)**
  - Easy to fetch one instruction in first stage of the pipeline and decode it in the second
- **It has just a few similar instruction formats**
  - With the source register fields being located in the same place in all instructions, 2nd stage can read the register file while decoding the type of instruction just fetched
- **Memory operands only appear in loads and stores**
  - We can use the execute stage to calculate the memory address and then access memory in the following stage
- **Alignment of memory operands on word boundaries**
  - We need not worry about a single data transfer instruction requiring two memory accesses; the data can be transferred between processor and memory in a single pipeline stage

Pipelining Instructions

In The Processor: Datapath & Control,

- **Each instruction takes multiple steps**
- **Each step is independent of each other and takes different datapath**



- **At each cycle, one instruction is fetched and sent to the processor**
- **Ideally, after pipeline is fully filled, one instruction completes each cycle**

MIPS Pipeline Stages

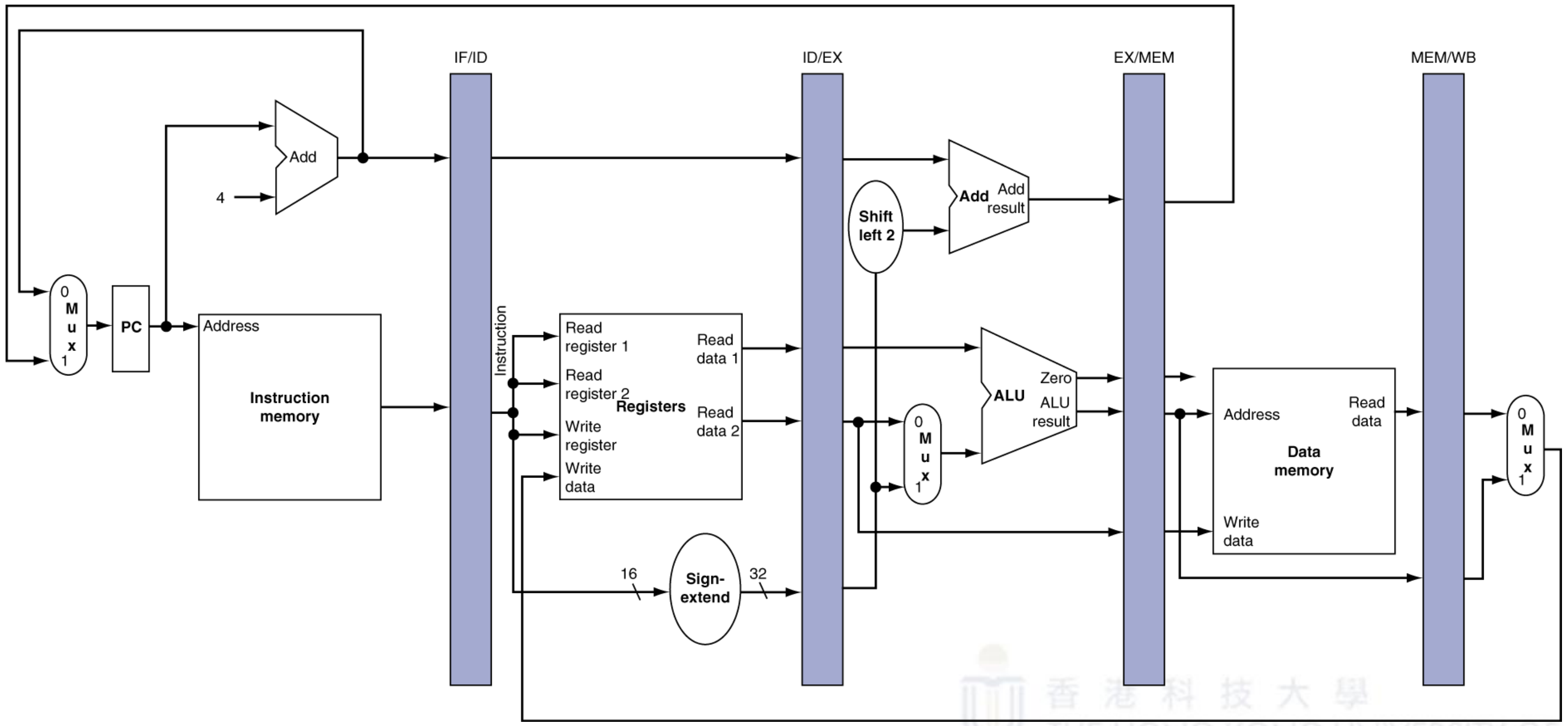
- Execution of each instruction is broken into 5 stages: (in the order of execution)

–	**IF**	: Fetch the instruction from memory
–	**ID**	: Instruction decode & register read
–	**EX**	: Perform ALU operation
–	**MEM**	: Memory access (if necessary)
–	**WB**	: Write result back to register

- **Each stage uses a different hardware unit and takes one clock cycle to complete.**
- **Instructions can co-exist in the datapath if all of them are in different stages of execution from one another**

Pipeline Registers

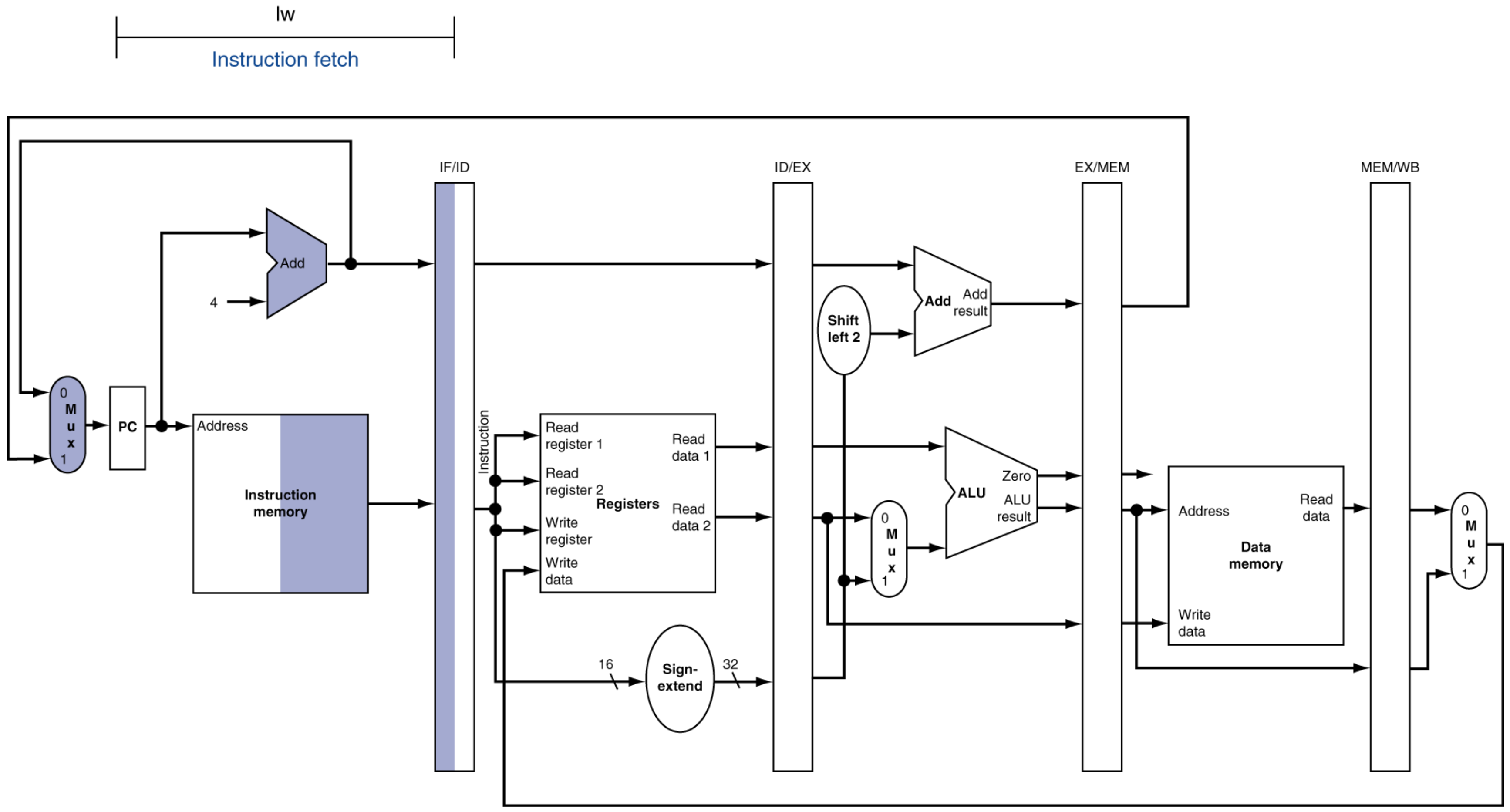
- Additional **pipeline registers** are needed
- Located **between the stages, i.e. IF/ID, ID/EX, EX/MEM, MEM/WB**
- Hold information produced in the previous cycle



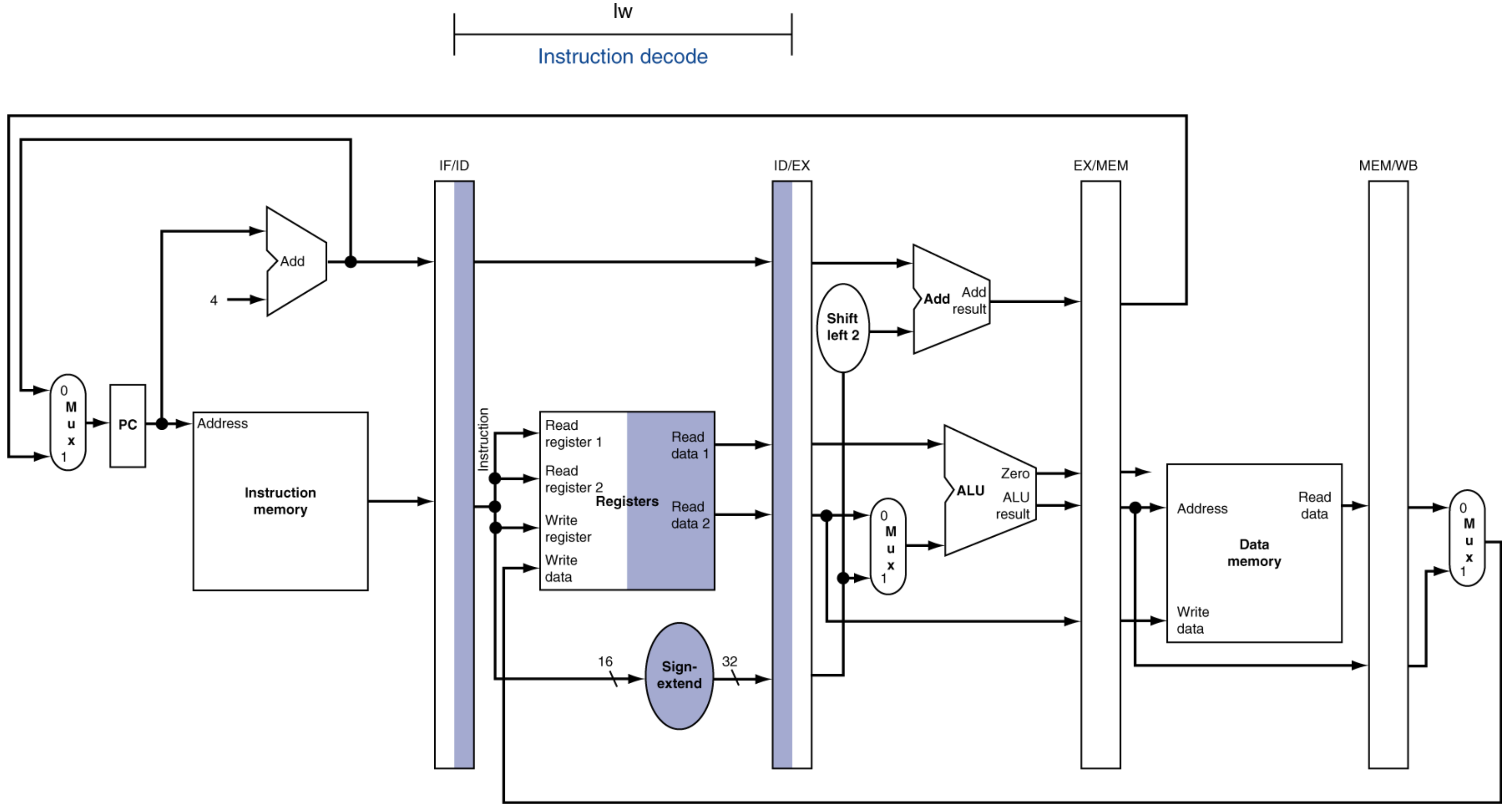
Pipeline Operation

- Every clock cycle, many instructions are simultaneously executing in a single datapath
- Cycle-by-cycle flow of instructions through the pipelined datapath
- Single-clock-cycle pipeline diagram
  - Shows pipeline usage in a single cycle
  - Highlight resources used
- Multi-clock-cycle pipeline diagram
  - Graph of operation over time

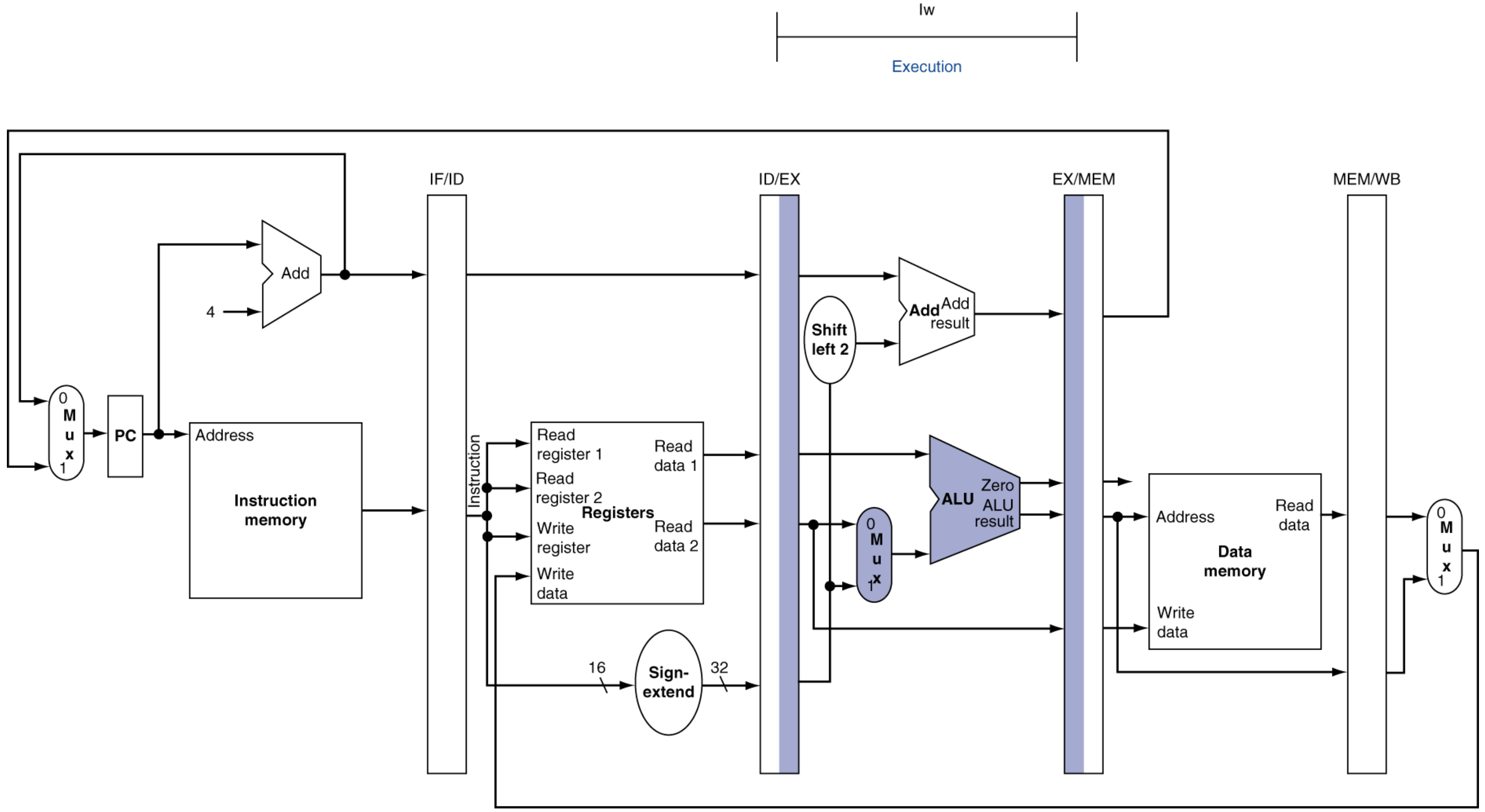
Single clock cycle diagram: IF stage of lw



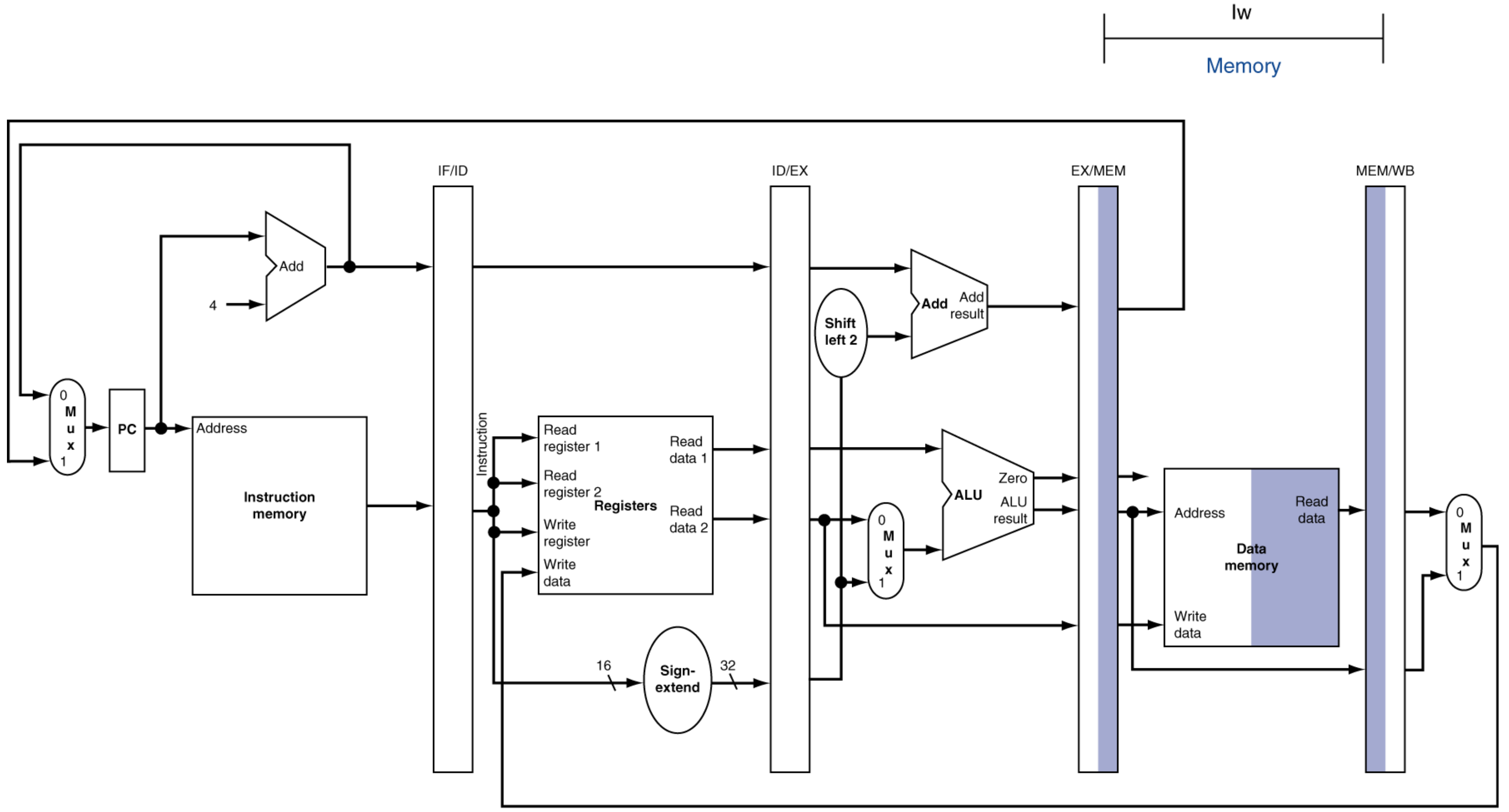
Single clock cycle diagram: ID stage of lw



Single clock cycle diagram: EXE stage of lw

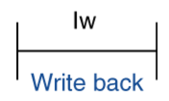


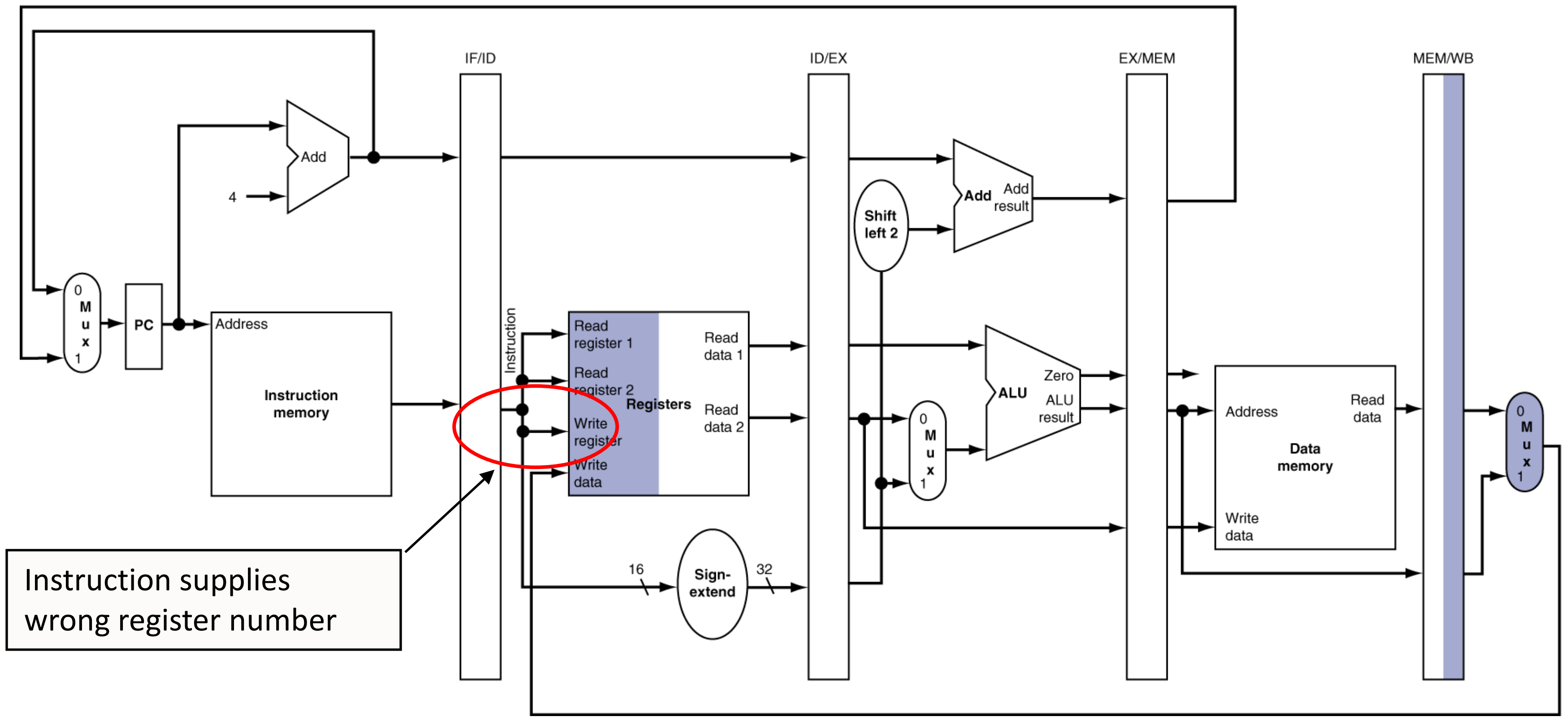
Single clock cycle diagram: MEM stage of lw



Single clock cycle diagram: WB stage of lw

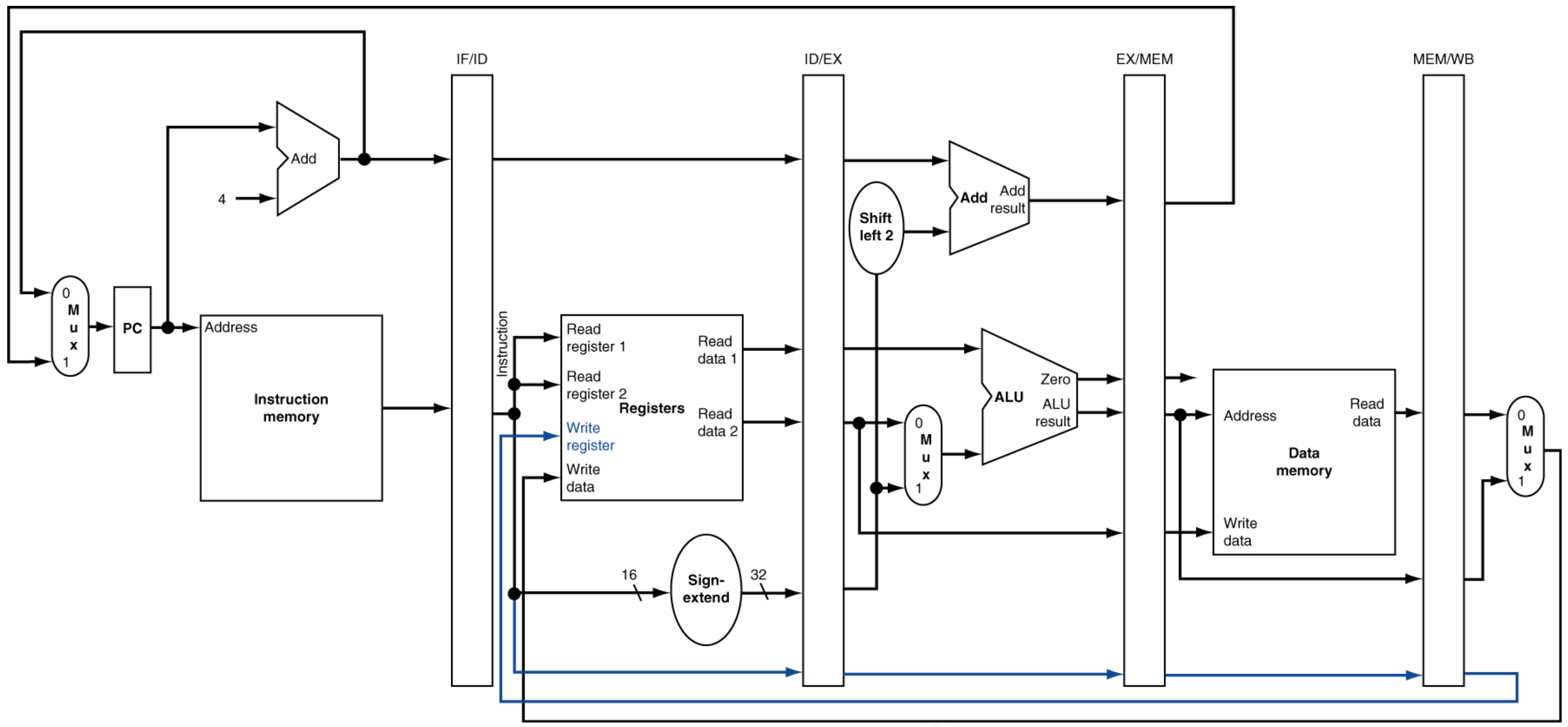
- There is a problem with the WB stage of lw!





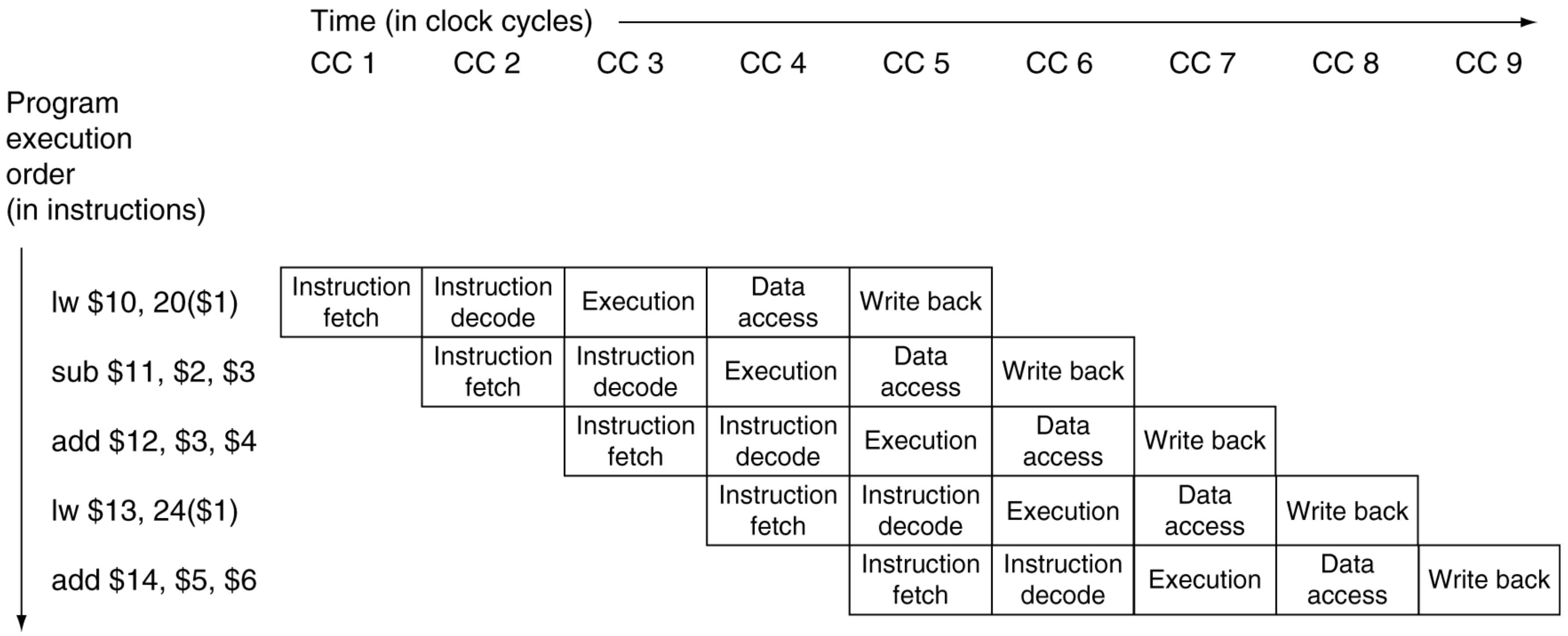
The Corrected Datapath for lw

- To solve this problem: the "write register" information is forwarded from the MEM/WB pipeline registers.



Multi-clock-cycle pipeline diagram : traditional view

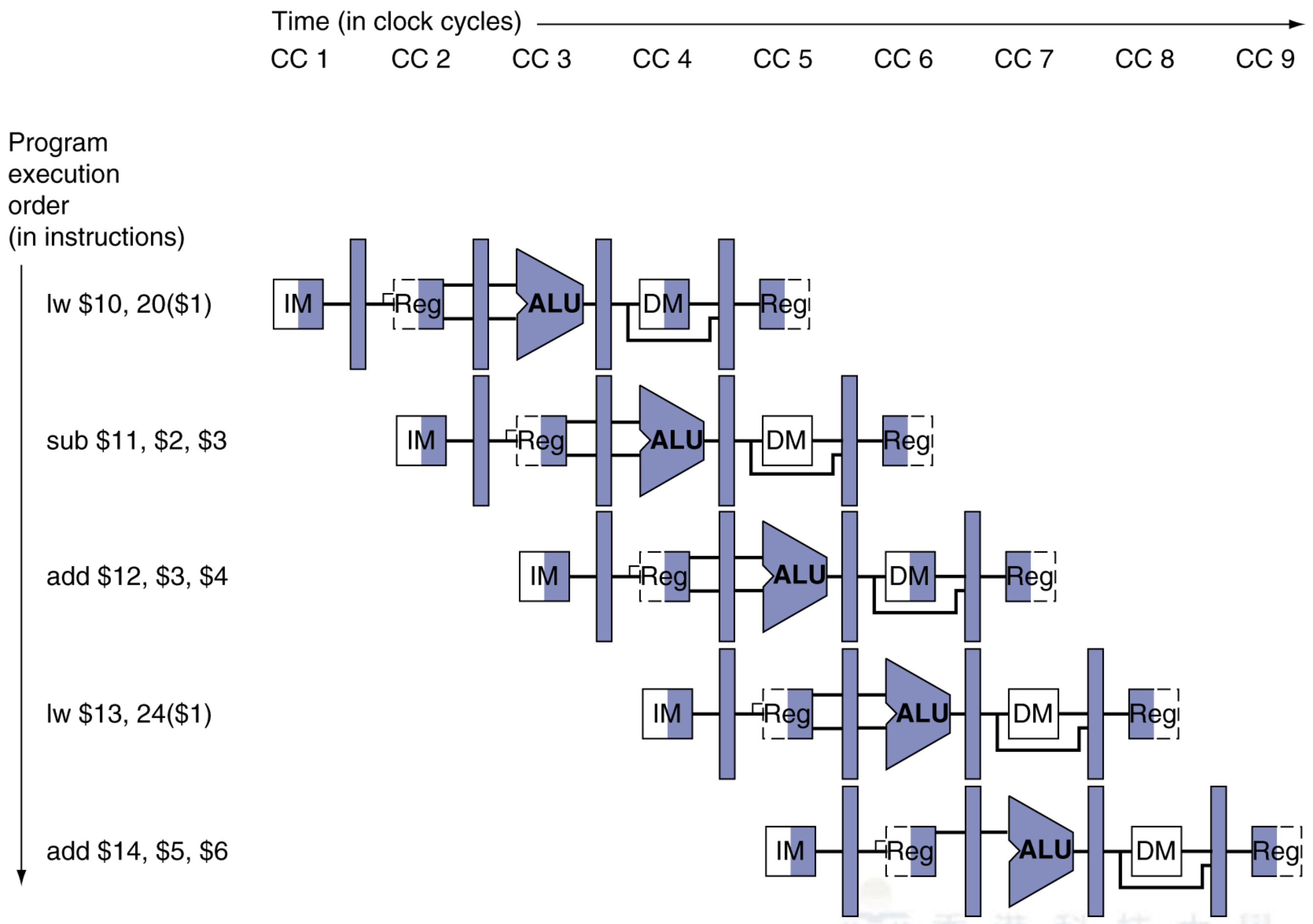
- The following diagram shows the execution of **a series of instructions.**



	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
lw $10, 20($1)	Instruction fetch	Instruction decode	Execution	Data access	Write back				
sub $11, $2, $3		Instruction fetch	Instruction decode	Execution	Data access	Write back			
add $12, $3, $4			Instruction fetch	Instruction decode	Execution	Data access	Write back		
lw $13, 24($1)				Instruction fetch	Instruction decode	Execution	Data access	Write back	
add $14, $5, $6					Instruction fetch	Instruction decode	Execution	Data access	Write back

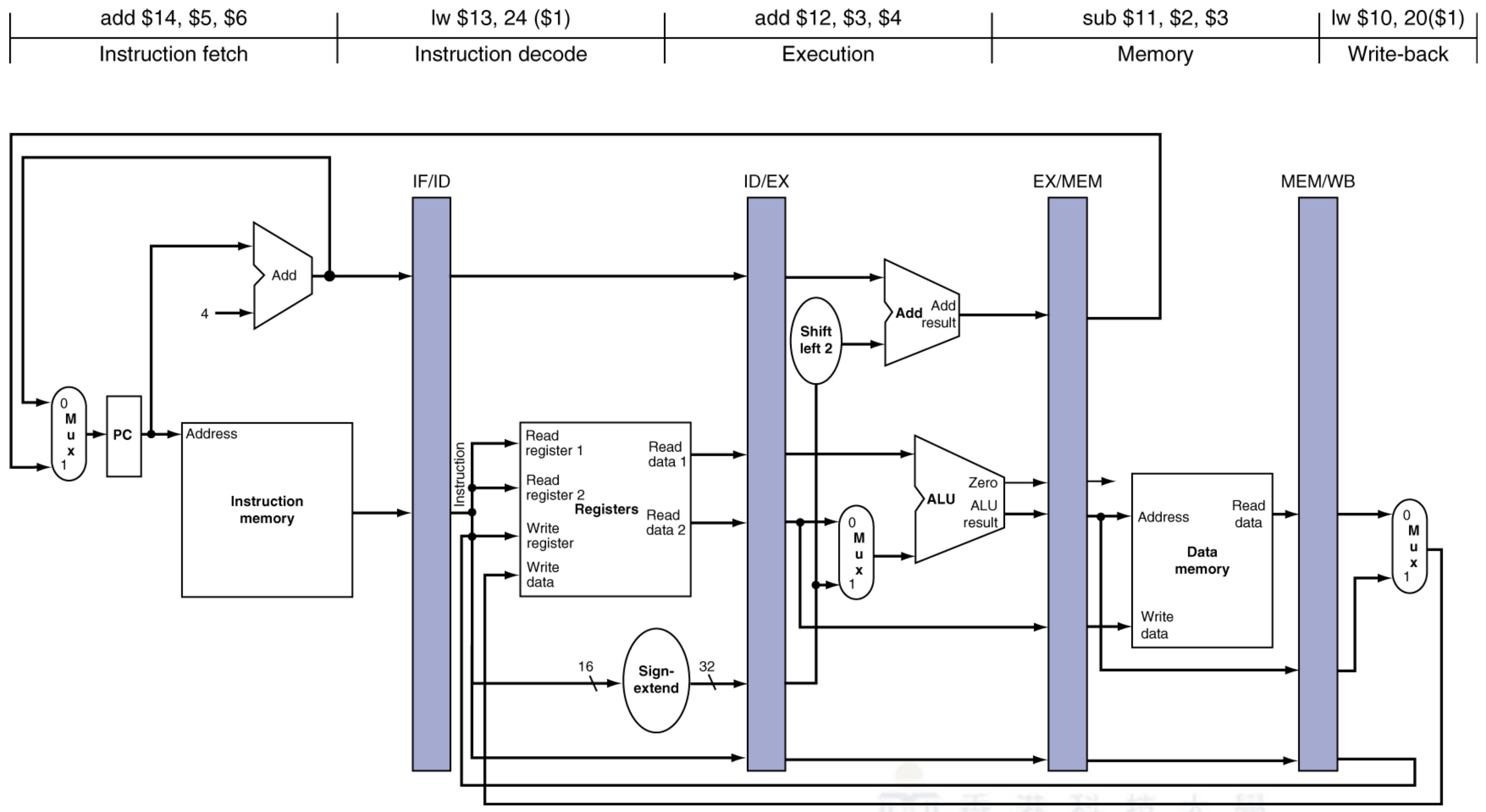
Multi-clock-cycle pipeline diagram: graphical view

- The multi-clock-cycle form showing the resource usage.



Single-clock-cycle diagram in CC5

- State of the pipeline in a given cycle



The Pipeline operation

- **Ideally**
- One stage begins in every cycle.
- One stage completes in each cycle.
- Each instruction takes 5 cycles

- In each clock cycle, several instructions are active.
- Different stages are executing different instructions.

- **Difficulty**
- How to generate the control signals ?
- we need to set the control signals for each pipeline stage for each instruction.

PIPELINED CONTROL

Pipelined Control Simplified

- Let's start with a simple design that views the problem in a greatly simplified way
- Temporarily ignore data dependence related problems (Hazards), and will provide solutions to this problem later.

Control Signals for Each Stage

- The control signals required by each stage are grouped
- Stage 1: Instruction fetch (IF) – no control signals, the instruction is read from the instruction memory and PC is updated to PC+4.
- Stage 2: Instruction decode and register read (ID) – no control signals, instruction is decoded and source operands are read from register file.
- Stage 3: Execute (EX)– RegDst, ALUOp, and ALUSrc.
- Stage 4: Memory Access (MEM)– Branch, MemRead, and MemWrite
- Stage 5: Write Back (WB) –MemToReg and RegWrite

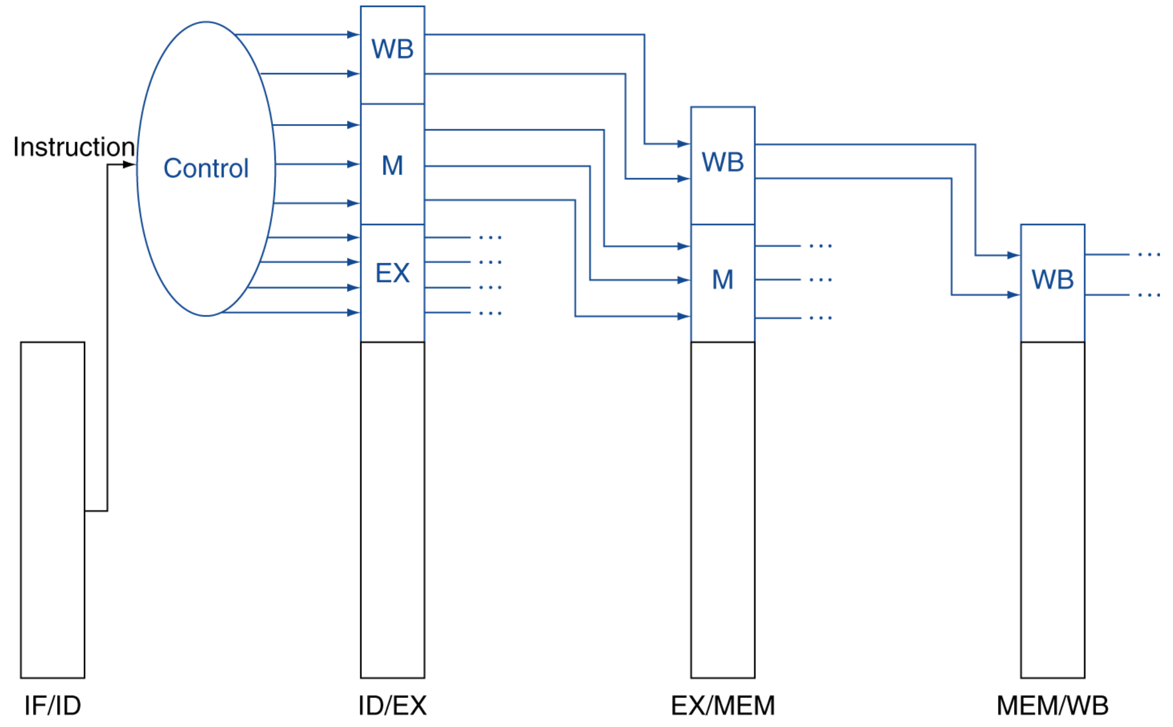
Control Signals for Each Stage (cont.)

- The group of control signals and their values for different classes of instructions

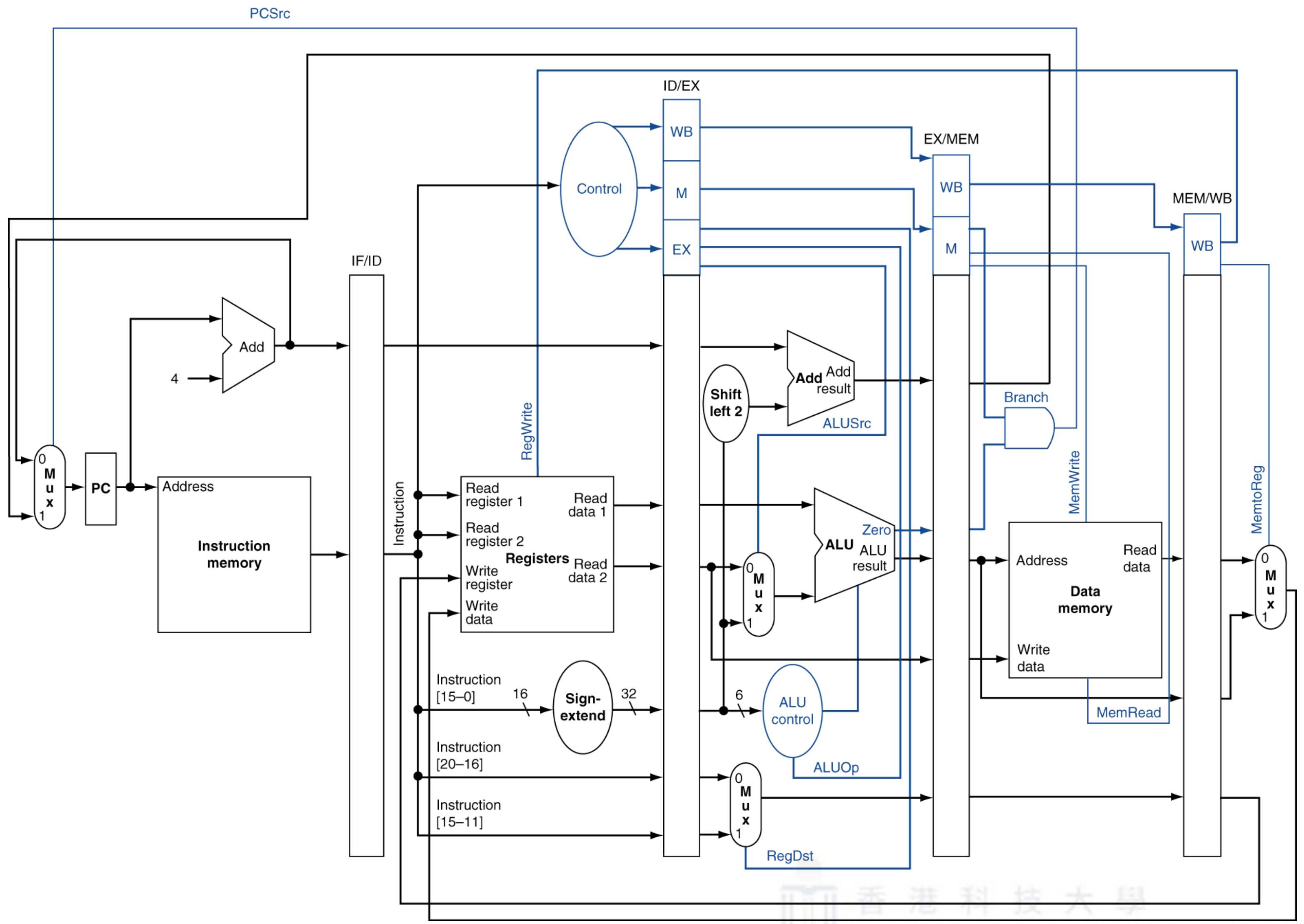
Instructions	EX				MEM			WB	
	RegDst	ALUOp1	ALUOp2	ALUSrc	Branch	MemRead	MemWrite	RegWrite	MemToReg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

Passing the Controls

- Use the same control signals derived from instruction as in single-cycle implementation
- Pipeline control signals too, so that the correct control signals are supplied for each stage of the instruction
- Control signals are passed to the next stage only if they are required



The Pipelined Control: the Complete Datapath



PIPELINE DATAPATH WITH HAZARDS

Dependences in Programs

- **Data dependence**

```
lw $s1, 200($s0)
add $s3, $s4, $s1
```

  add can’t do ID (i.e., read register $s1) until lw updates $s1

- **Control dependence**

```
bne $s1, $s2, target
add $s3, $s4, $s5
```

  next IF can’t start until bne completes the comparison

- **These dependences may cause the pipeline not be fully filled**
- **Execution stops to wait for data or control to be produced**
- **next instruction cannot be executed in next cycle**

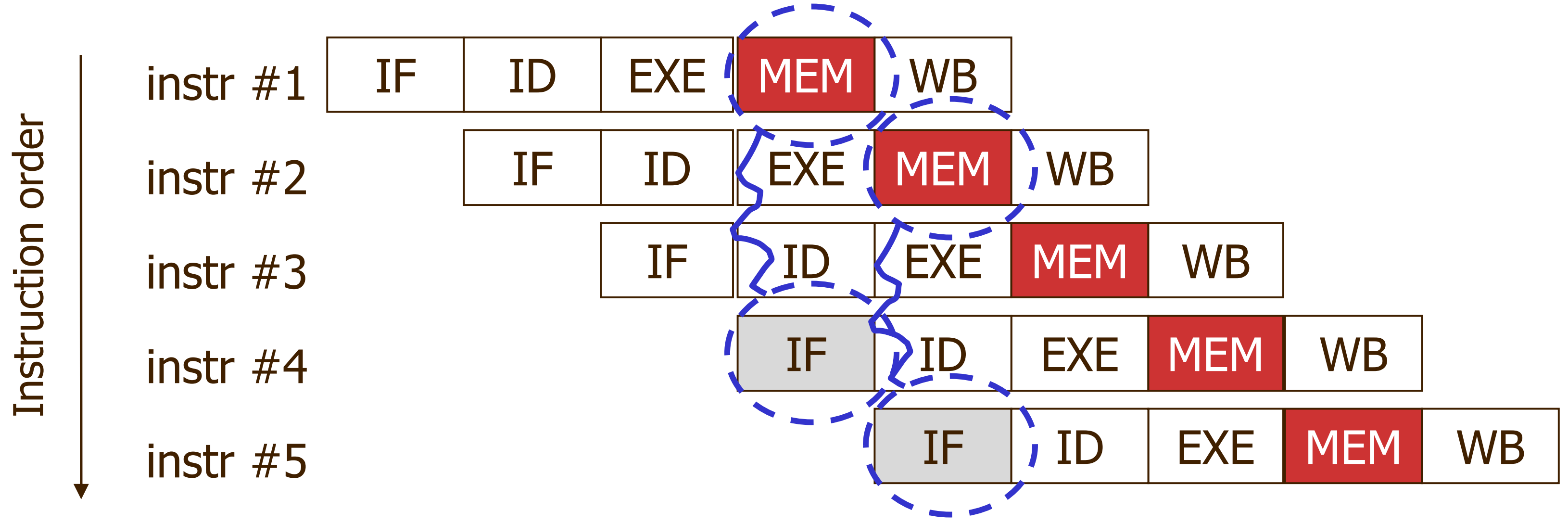
Pipeline Hazards

- **Hazards are situations in pipelining when the next instruction cannot be executed in the following clock cycle.**

- **Three types of pipelined hazards**
  - **Structural hazards:** A required resource is busy
  - **Data hazards:** Need to wait for previous instruction to complete its data read/write
  - **Control hazards:** Deciding on control action depends on previous instruction

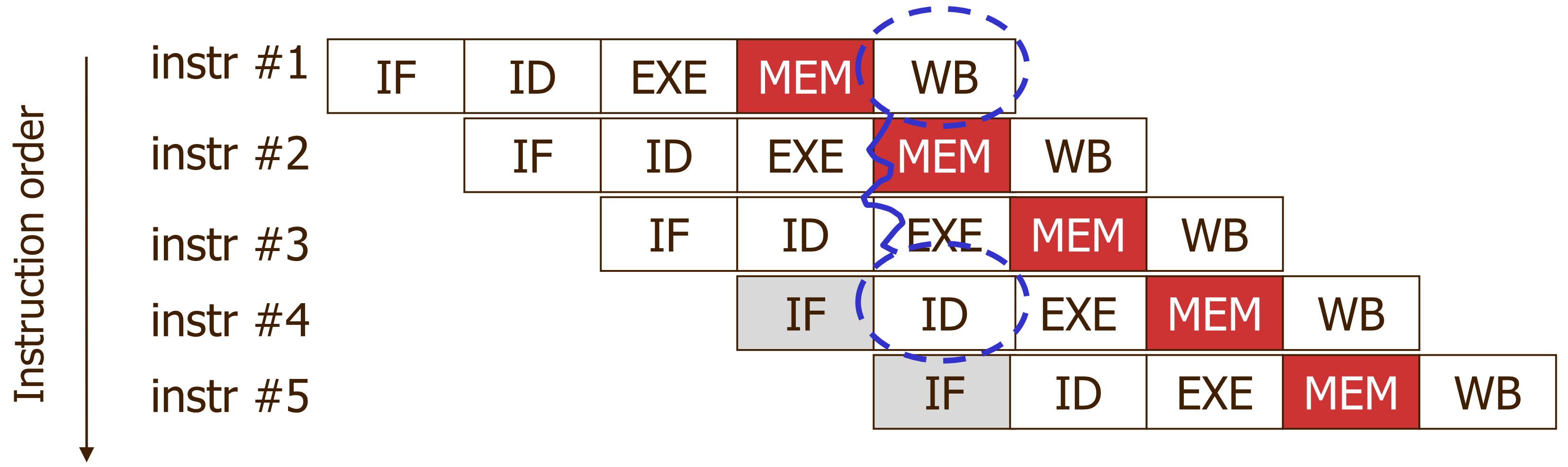
- **Hazards can always be resolved by waiting. But this slows down the pipeline.**

Structural Hazards: Memory



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

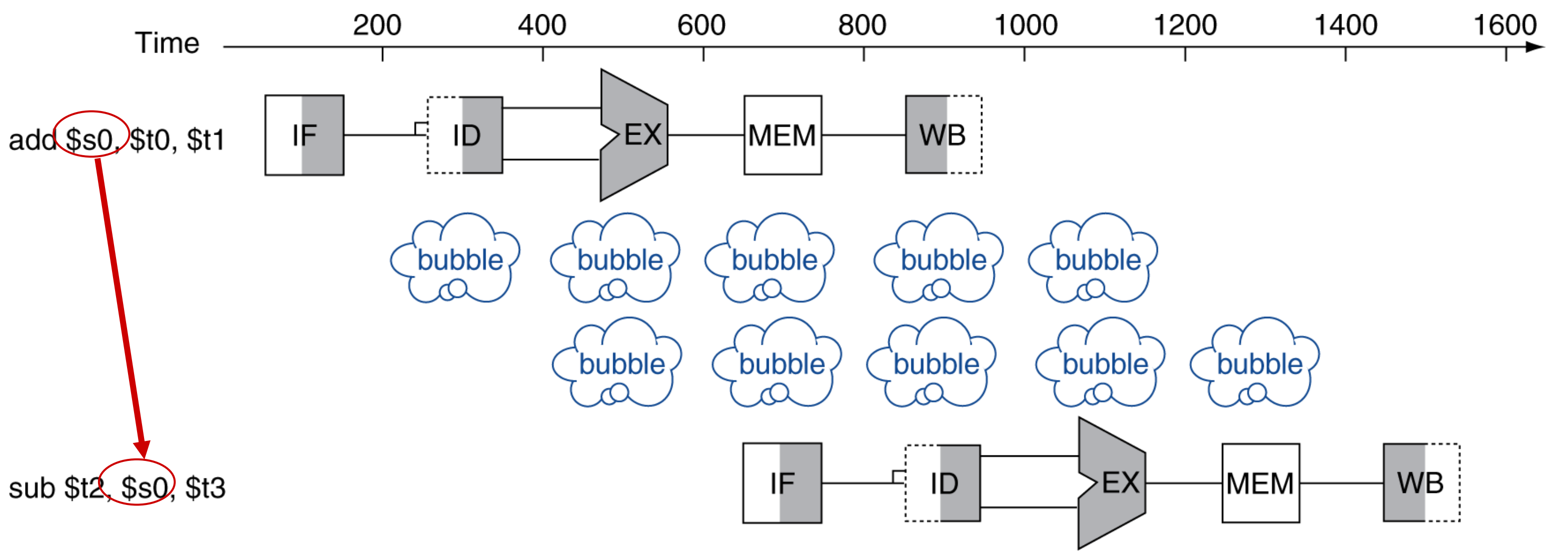
Structural Hazards: Registers



- **Fact: Register access VERY fast. Takes half the time of ALU stage or less**
  - always Write to registers during 1st half of each clock cycle
  - always Read from Registers during 2nd half of each clock cycle
  - Register file supports Write and Read during same clock cycle (in this order)

Data Hazard

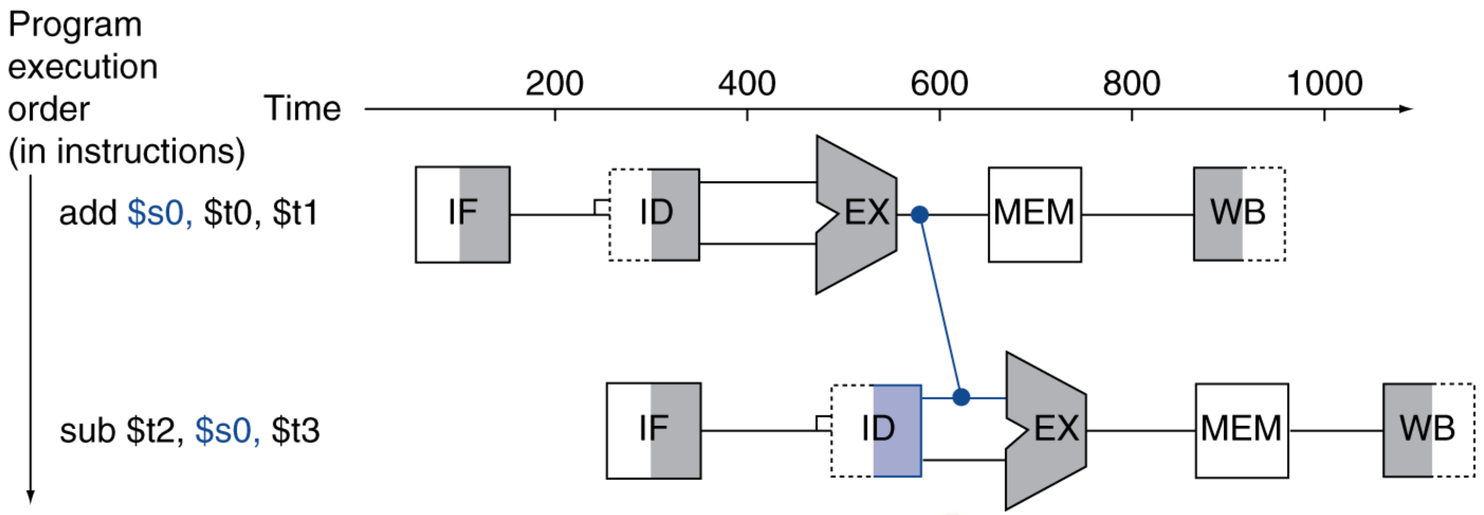
- An instruction depends on completion of data access by a previous instruction



- a bubble or **pipeline stall** is a delay in execution of an instruction in an instruction pipeline in order to resolve a hazard.

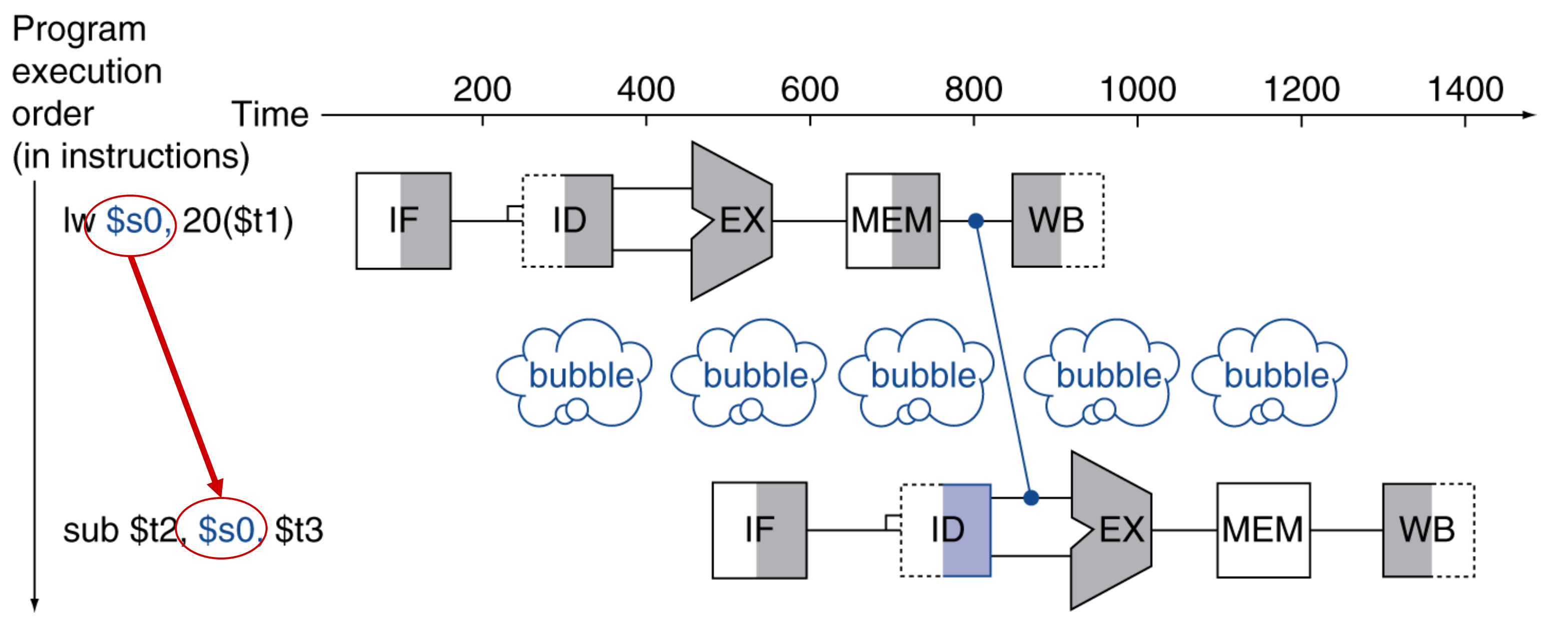
Forwarding(aka Bypassing)

- **Forwarding partially solves the data hazard problem**
  - Use result when it is computed
  - Don’t wait for it to be stored in a register
  - Requires extra connections in the datapath



Load-Use Data Hazard

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



Code Scheduling to Avoid Stalls

- Consider this code sequence

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

Assume a to e are stored in memory address **0($t0), 4($t0), 8($t0), 12($t0)** and **16($t0)** respectively. Assume **forwarding is used**.

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

13 cycles

```
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)
```

11 cycles

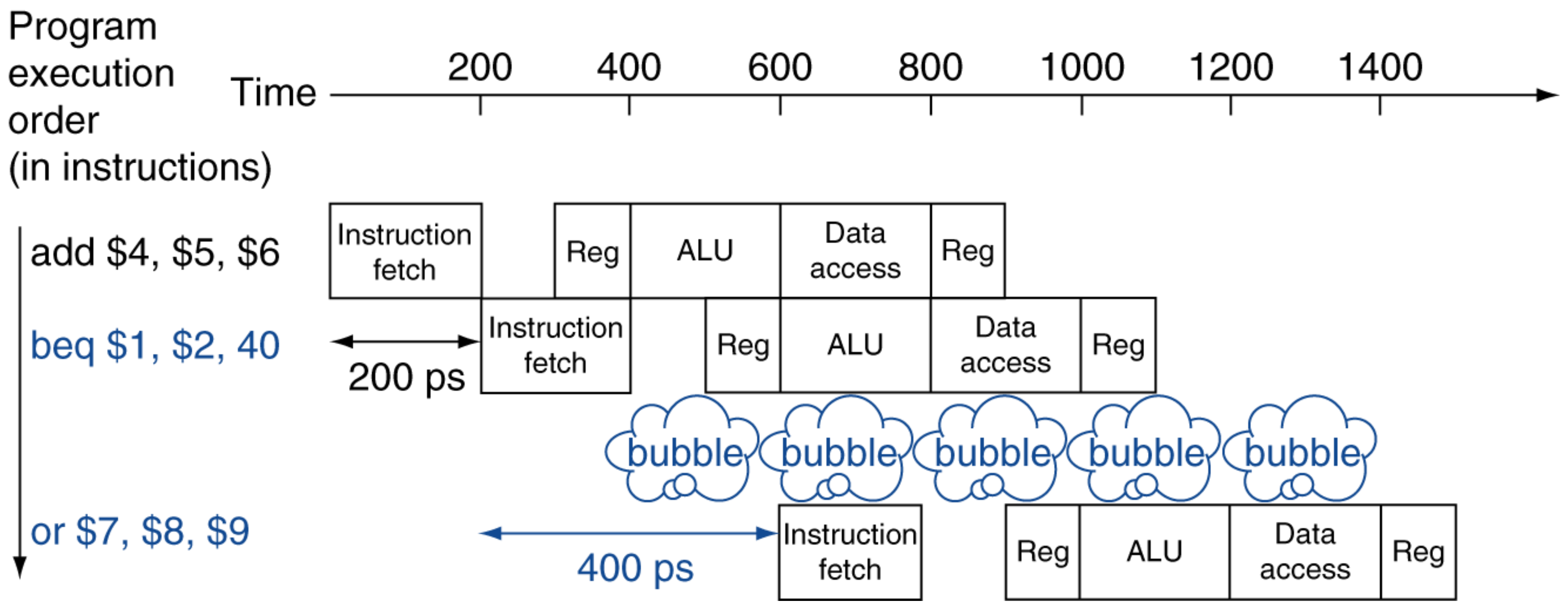
Control Hazards

- **Branch determines flow of control**
  - Fetching next instruction depends on branch outcome
  - Pipeline can’t always fetch correct instruction
    - Still working on ID stage of branch

- **In MIPS pipeline**
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

Stall on Branch

- Wait until branch outcome determined before fetching next instruction

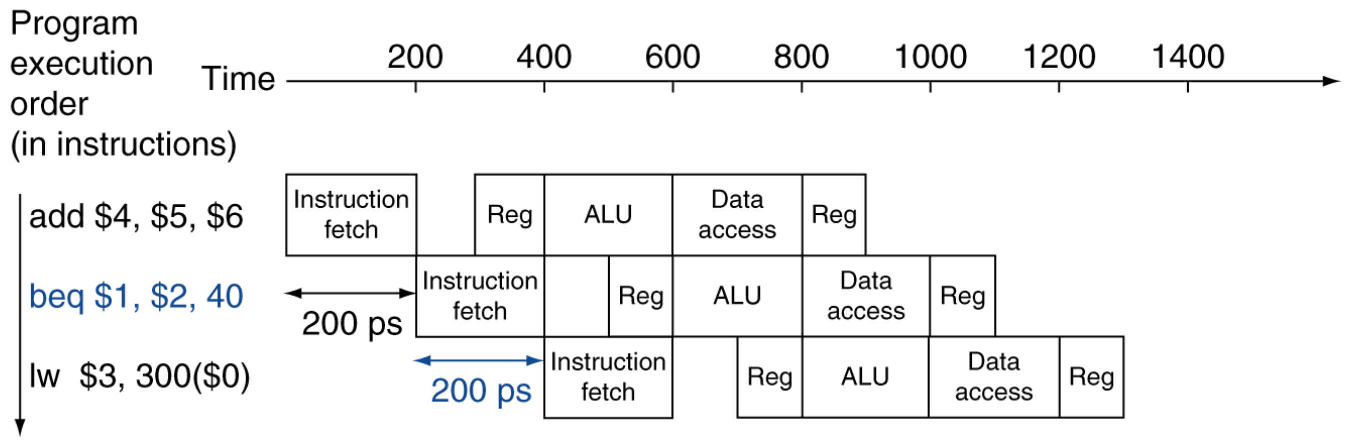


Branch Prediction

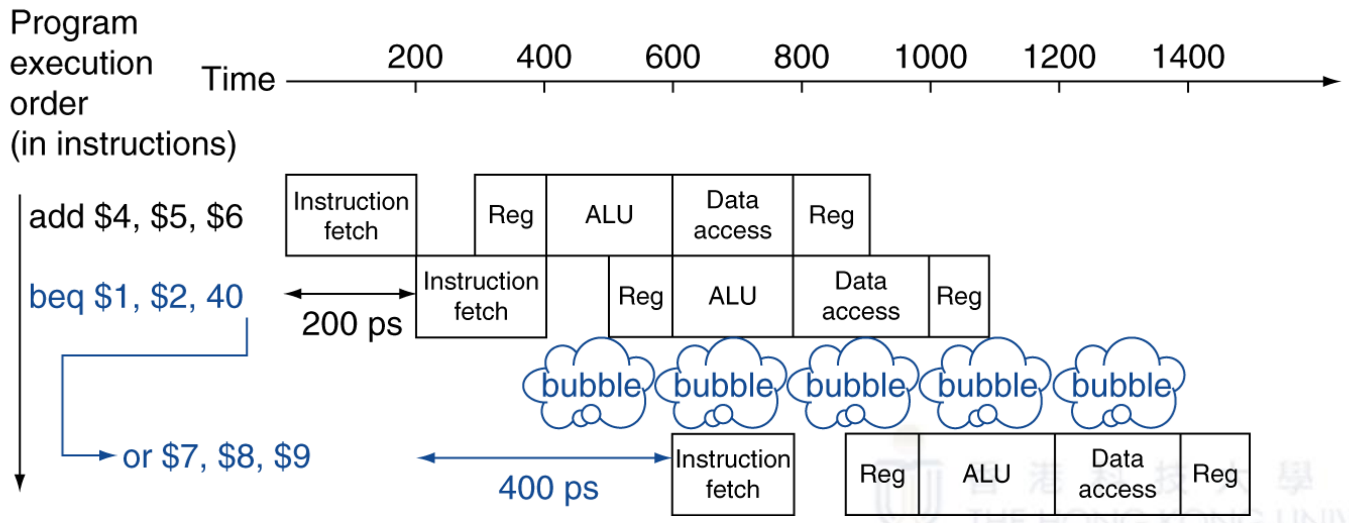
- **Longer pipelines can't readily determine branch outcome early**
  - Stall penalty becomes unacceptable
- **Predict outcome of branch**
  - Only stall if prediction is wrong
- **In MIPS pipeline**
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

MIPS with Static Branch Prediction (Not Taken)

Prediction correct



Prediction incorrect



Concluding Remarks

- **Pipelining improves the throughput by allowing reuse of functional units by different instructions**
- **Pipelining allows an instruction to complete in each clock cycle, but it requires a very careful design and additional registers to store intermediate results between pipeline stages**
- **Pipelined Control is implemented like single cycle control with needed control signals are forwarded down the pipeline**
- **Concurrence between instructions in the pipeline may cause**
  - **Data Hazard:** data is needed by an instruction before it is produced by a previous one
  - **Structural Hazard:** a hardware unit is needed by an instruction while another is still using it
  - **Control Hazard:** the next instruction cannot be determined in the next clock cycle
- **Hazards can always be solved by delaying (inserting bubbles)**

Concluding Remarks (cont.)

- **Structural hazard is solved by:**
  - Separating the instruction memory from the data memory
  - Writing to the register file in the first half of the clock cycle and reading from it in the second half
- **Data hazard is solved by:**
  - **Forwarding/Bypassing**
  - **Inserting bubbles**
- **Control hazards are solved by:**
  - **Hardware:** add comparator to complete the comparison earlier
  - **Speculation:** guess if the branch is taken or not
  - **Delay the branch:** fill the bubbles with useful work that is independent of the branch