CS23402 - COMPUTER ARCHITECTURE

UNIT – 3 – PROCESSOR DESIGN

Presented By
Dr. S. Muthurajkumar
Associate Professor
Department of Computer Technology
Anna University – MIT Campus
Chennai

INTRODUCTION

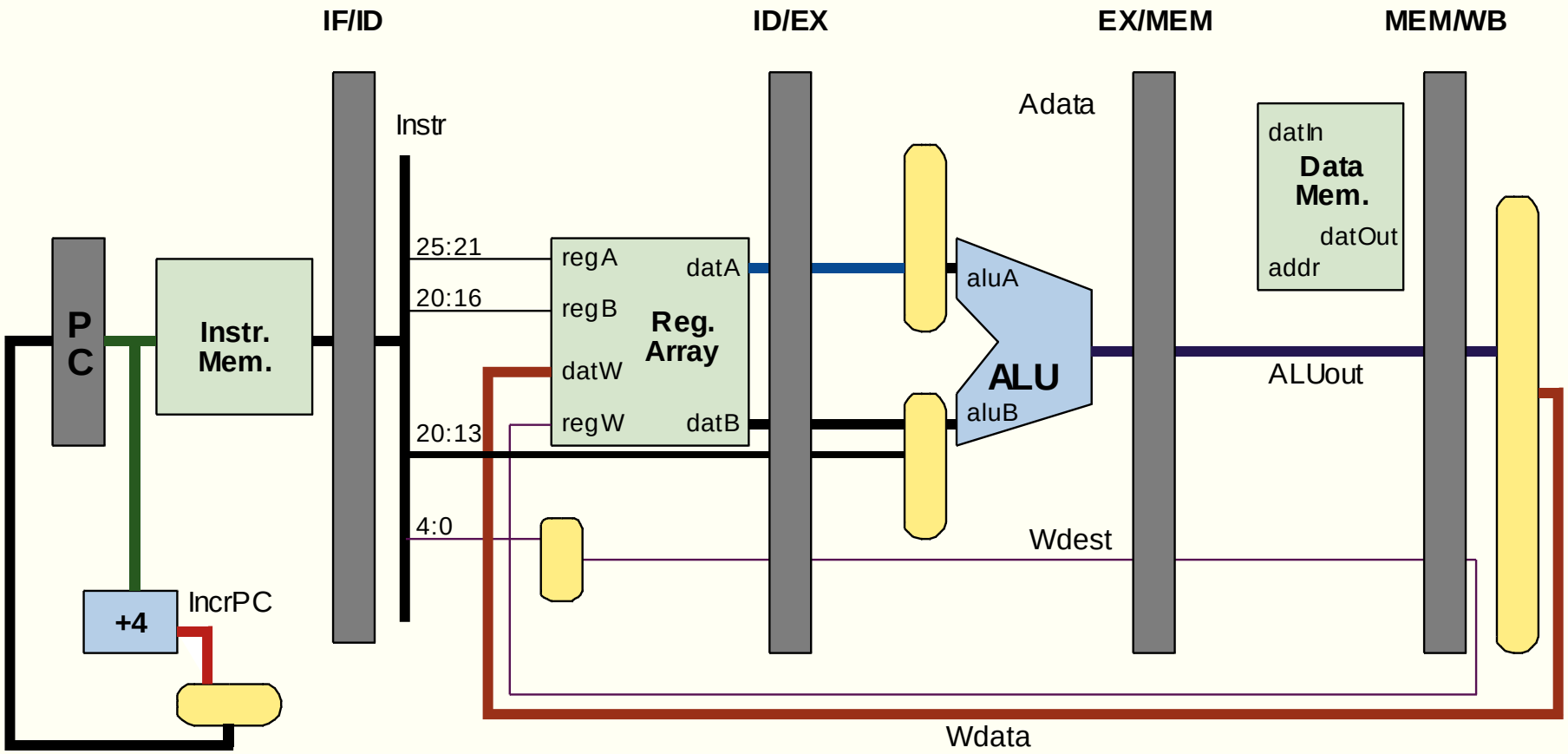
- Datapath design
- Implementation of the basic MIPS ISA
- Building the datapath
- A simple implementation scheme
- Drawbacks
- Instruction Level Parallelism
- Pipelining
- Performance
- Pipeline hazards
- Pipelined datapath and control
- Handling data hazards and control hazards
- Exceptions.

DATA PATH DESIGN

- A datapath is a collection of functional units such as arithmetic logic units or multipliers that perform data processing operations, registers, and buses. Along with the control unit it composes the Central Processing Unit (CPU).
- IF – Instruction Fetch
- ID – Instruction Decode / Register Fetch
- EX – Execute
- MEM – Memory Access
- WB – Write Back

DATA PATH DESIGN

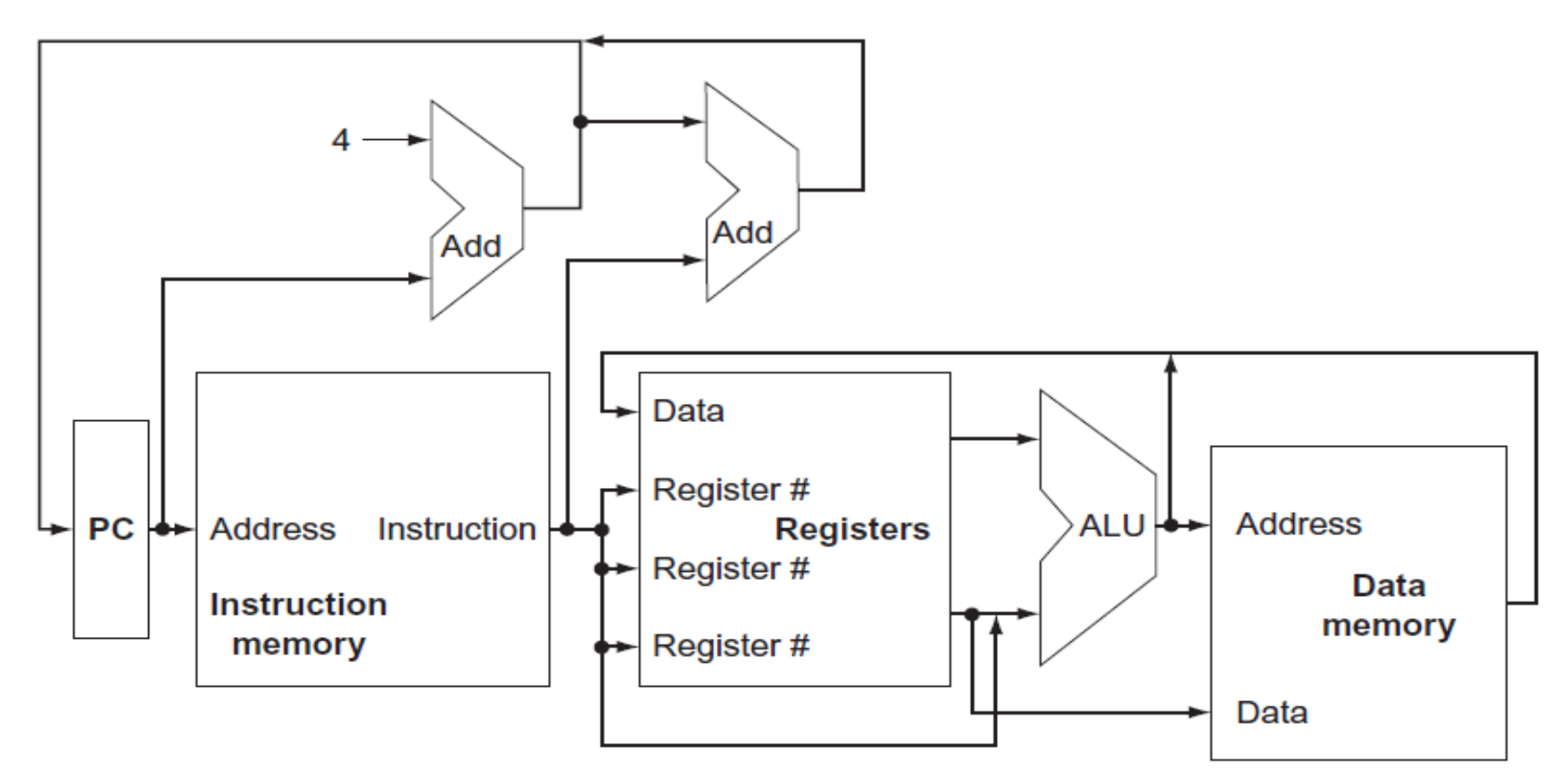
IF	ID	EX	MEM	WB
instruction fetch	instruction decode/ register fetch	execute	memory access	write back



IMPLEMENTATION OF THE BASIC MIPS

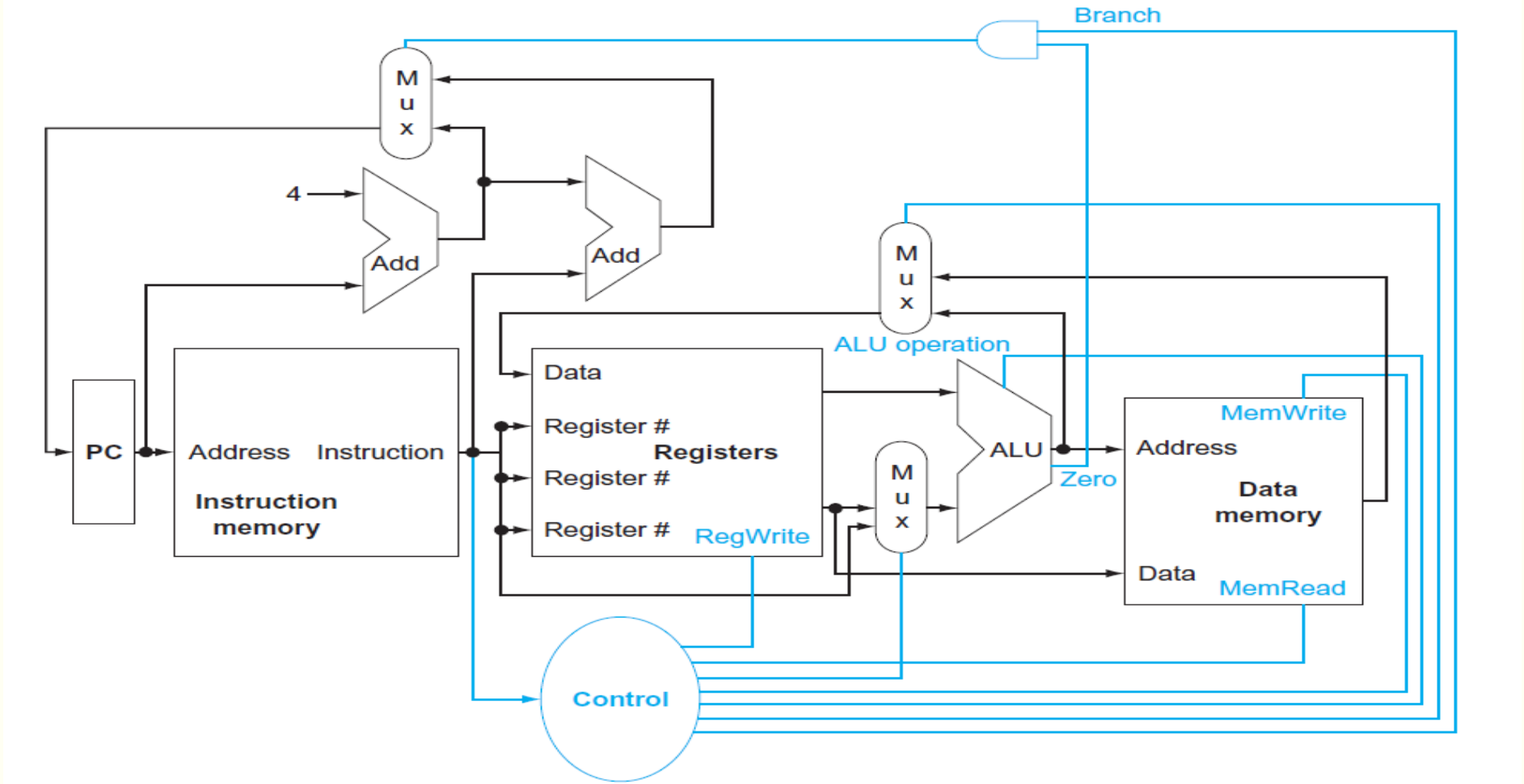
- IF – Instruction Fetch
- ID – Instruction Decode / Register Fetch
- EX – Execute
- MEM – Memory Access
- WB – Write Back

IMPLEMENTATION OF THE BASIC MIPS



An abstract view of the implementation of the MIPS subset showing the major functional units and the major connections between them.

IMPLEMENTATION OF THE BASIC MIPS



The basic implementation of the MIPS subset, including the necessary multiplexors and control lines.

IMPLEMENTATION OF THE BASIC MIPS

Instruction Fetch (IF)

- IR = Mem[PC];
- NPC = PC + 4;

Operation:

- Send out the PC and fetch the instruction from memory into the instruction register (IR). Increment the PC by 4 to address the next sequential instruction.
- IR - holds instruction that will be needed on subsequent clock cycles
- Register NPC - holds next sequential PC.

IMPLEMENTATION OF THE BASIC MIPS

Instruction Decode / Register Fetch (ID)

- A = Regs[rs];
- B = Regs[rt];
- Imm = sign-extended immediate field of IR;

Operation:

- Decode instruction and access register file to read the registers (rs and rt - register specifiers). Outputs of general purpose registers are read into 2 temporary registers (A and B) for use in later clock cycles.
- Lower 16 bits of IR are sign extended and stored into the temporary register Imm, for use in the next cycle.

IMPLEMENTATION OF THE BASIC MIPS

Execute (EX)

- ALU operates on the operands prepared in the prior cycle, performing one of four functions depending on the MIPS instruction type.

i) Memory reference:

- ALUOutput = A + Imm;

ii) Register-Register ALU instruction:

- ALUOutput = A func B;

Operation:

a) ALU performs the operation specified by the function code on the value in register A and in register B.

b) Result is placed in temporary register ALUOutput

IMPLEMENTATION OF THE BASIC MIPS

Execute (EX)

iii) Register-Immediate ALU instruction:

- ALUOutput = A op Imm;

Operation:

a) ALU performs operation specified by the opcode on the value in register A and register Imm.

b) Result is placed in temporary register ALUOutput.

IMPLEMENTATION OF THE BASIC MIPS

Execute (EX)

iv)Branch:

- ALUOutput = NPC + (Imm << 2);
- Cond = (A == 0)

Operation:

a) ALU adds NPC to sign-extended immediate value in Imm, which is shifted left by 2 bits to create a word offset, to compute address of branch target.

b) Register A, which has been read in the prior cycle, is checked to determine whether branch is taken.

c) Considering only one form of branch (BEQZ), the comparison is against 0.

IMPLEMENTATION OF THE BASIC MIPS

Memory Access (MEM)

- PC is updated for all instructions: PC = NPC;

i. Memory reference:

- LMD = Mem[ALUOutput] or Mem[ALUOutput] = B;

Operation:

a) Access memory if needed.

b) Instruction is load-data returns from memory and is placed in LMD (Load Memory Data)

c) Instruction is store-data from the B register is written into memory

IMPLEMENTATION OF THE BASIC MIPS

Memory Access (MEM)

ii. Branch:

- if (cond) PC = ALUOutput

Operation:

- If the instruction branches, PC is replaced with the branch destination address in register ALUOutput

IMPLEMENTATION OF THE BASIC MIPS

Write Back (WB)

- Register-Register ALU instruction: Regs[rd] = ALUOutput;
- Register-Immediate ALU instruction: Regs[rt] = ALUOutput;
- Load instruction:
- Regs[rt] = LMD;

Operation:

- Write the result into register file, depending on the effective opcode.

BUILDING DATA PATH

Data path

- Components of the processor that perform arithmetic operations and holds data

Control

- Components of the processor that commands the datapath, memory, I/O devices according to the instructions of the memory.

Building a Data path

- Elements that process data and addresses in the CPU - Memories, registers, ALUs.
- MIPS datapath can be built incrementally by considering only a subset of instructions.

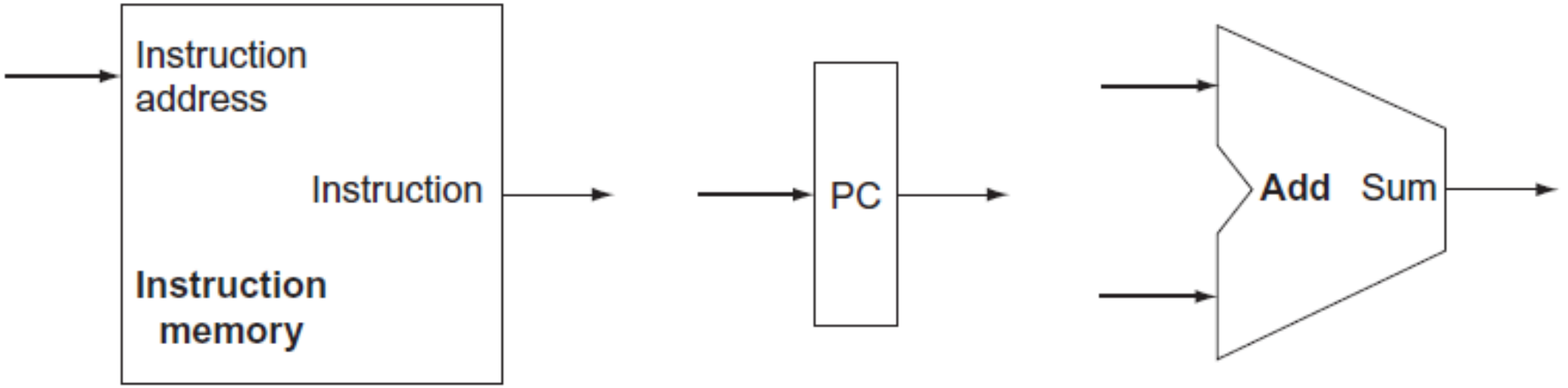
BUILDING DATA PATH

3 main elements are instruction memory, program counter and adder.

- A memory unit to store instructions of a program and supply instructions given an address. Needs to provide only read access (once the program is loaded).- No control signal is needed
- PC (Program Counter or Instruction address register) is a register that holds the address of the current instruction
- A new value is written to it every clock cycle. No control signal is required to enable write
- Adder to increment the PC to the address of the next instruction
- An ALU permanently wired to do only addition. No extra control signal required

BUILDING DATA PATH

Two state elements are needed to store and access instructions, and an adder is needed to compute the next instruction address.



a. Instruction memory b. Program counter c. Adder

BUILD DATA PATH – ELEMENTS TYPES

State element:

- A memory element, i.e., it contains a state
- E.g., program counter,

Instruction memory Combinational element:

- Elements that operate on values
- Eg. adder ALU

Elements required by the different classes of instructions

- Arithmetic and logical instructions
- Data transfer instructions
- Branch instructions

BUILD DATA PATH – ELEMENTS TYPES

R-Format ALU Instructions

- E.g., add $t1, $t2, $t3

Perform arithmetic/logical operation

- Read two register operands and write register result

Register file:

- A collection of the registers
- Any register can be read or written by specifying the number of the register contains the register state of the computer

BUILD DATA PATH – ELEMENTS TYPES

Read from register

- 2 inputs to the register file specifying the numbers
- 5 bit wide inputs for the 32 registers
- 2 outputs from the register file with the read values
- 32 bit wide
- For all instructions. No control required.

BUILD DATA PATH – ELEMENTS TYPES

Write to register file

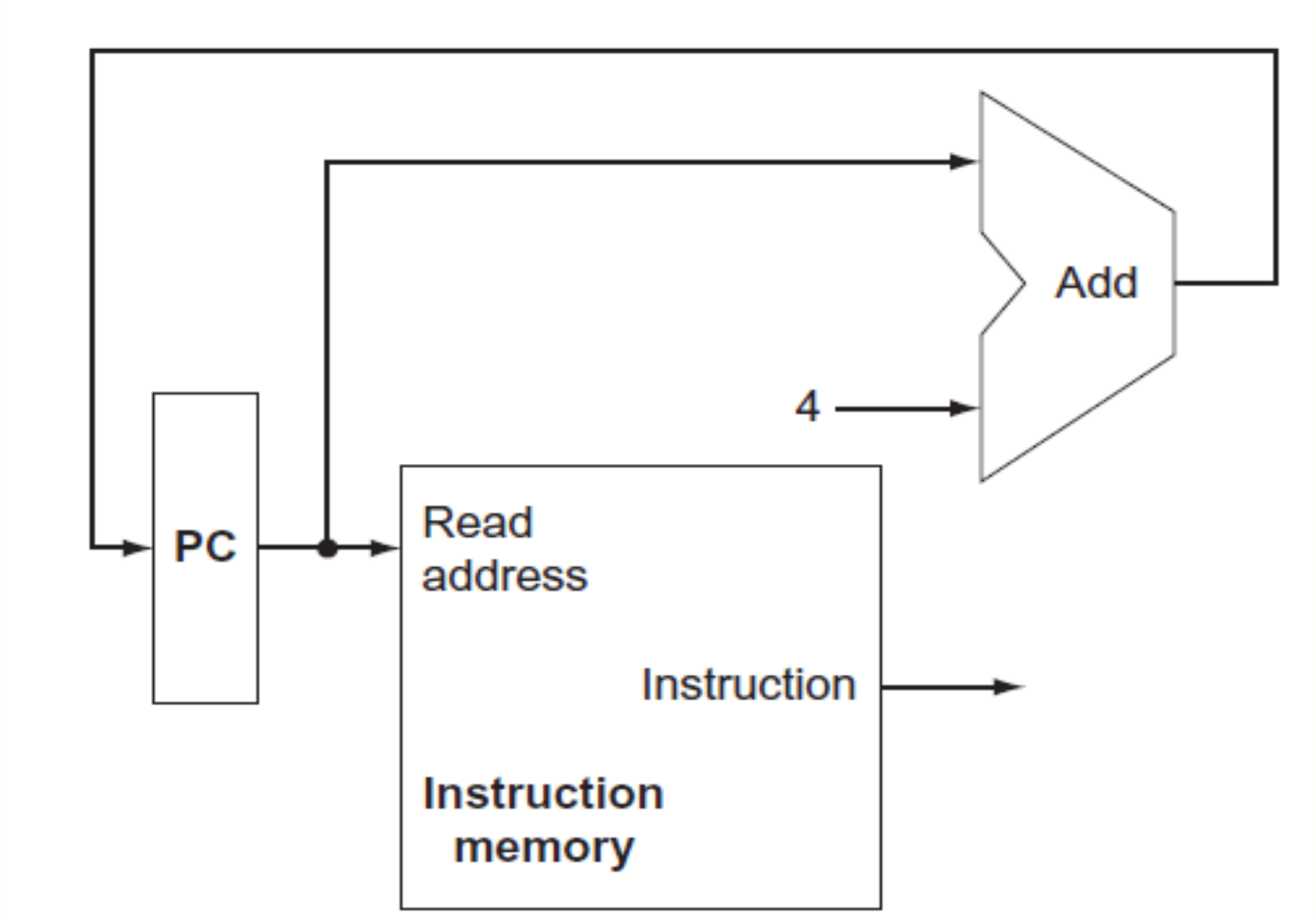
- 1 input to the register file specifying the number 5 bit wide inputs for the 32 registers
- 1 input to the register file with the value to be written 32 bit wide
- Only for some instructions. RegWrite control signal.

BUILD DATA PATH – ELEMENTS TYPES

ALU

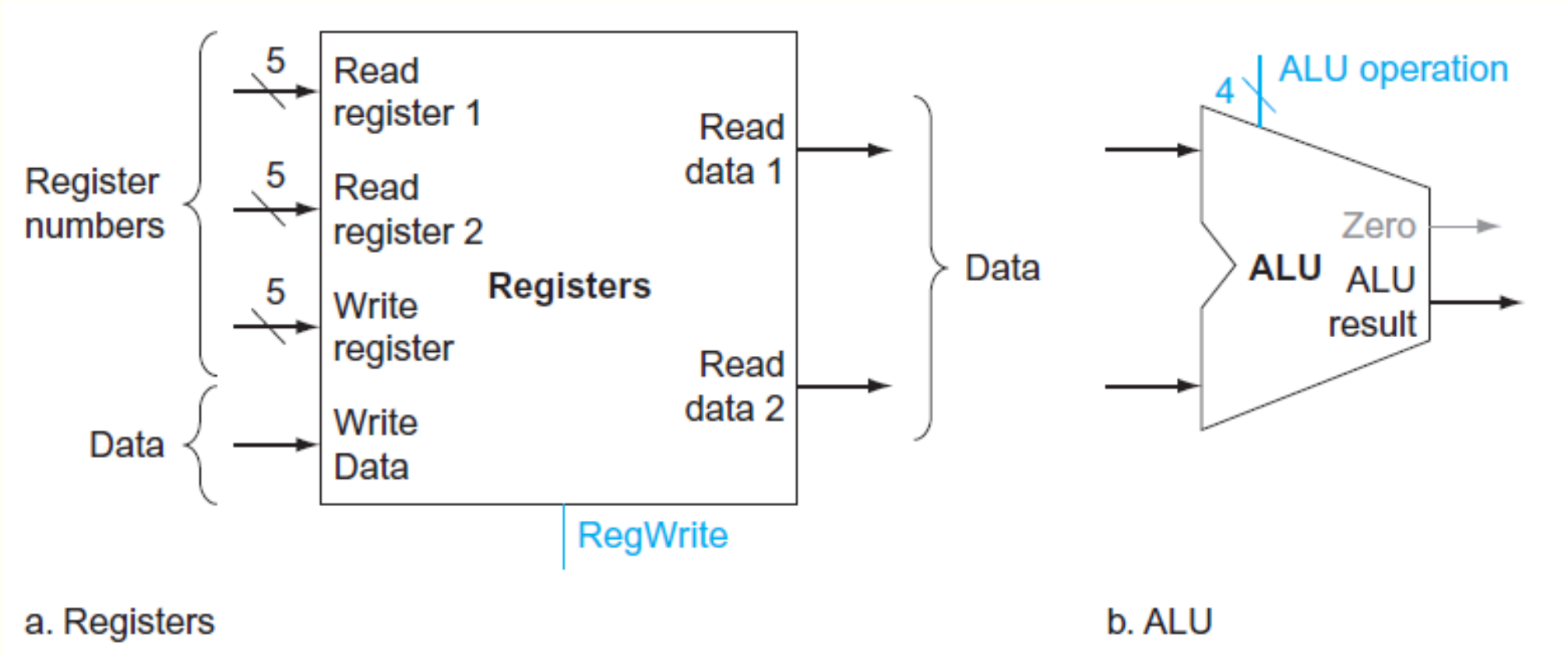
- Takes two 32 bit input and produces a 32 bit output
- Also, sets one-bit signal if the results is 0
- The operation done by ALU is controlled by a 4 bit control signal input.
- This is set according to the instruction.

BUILD DATA PATH – ELEMENTS TYPES



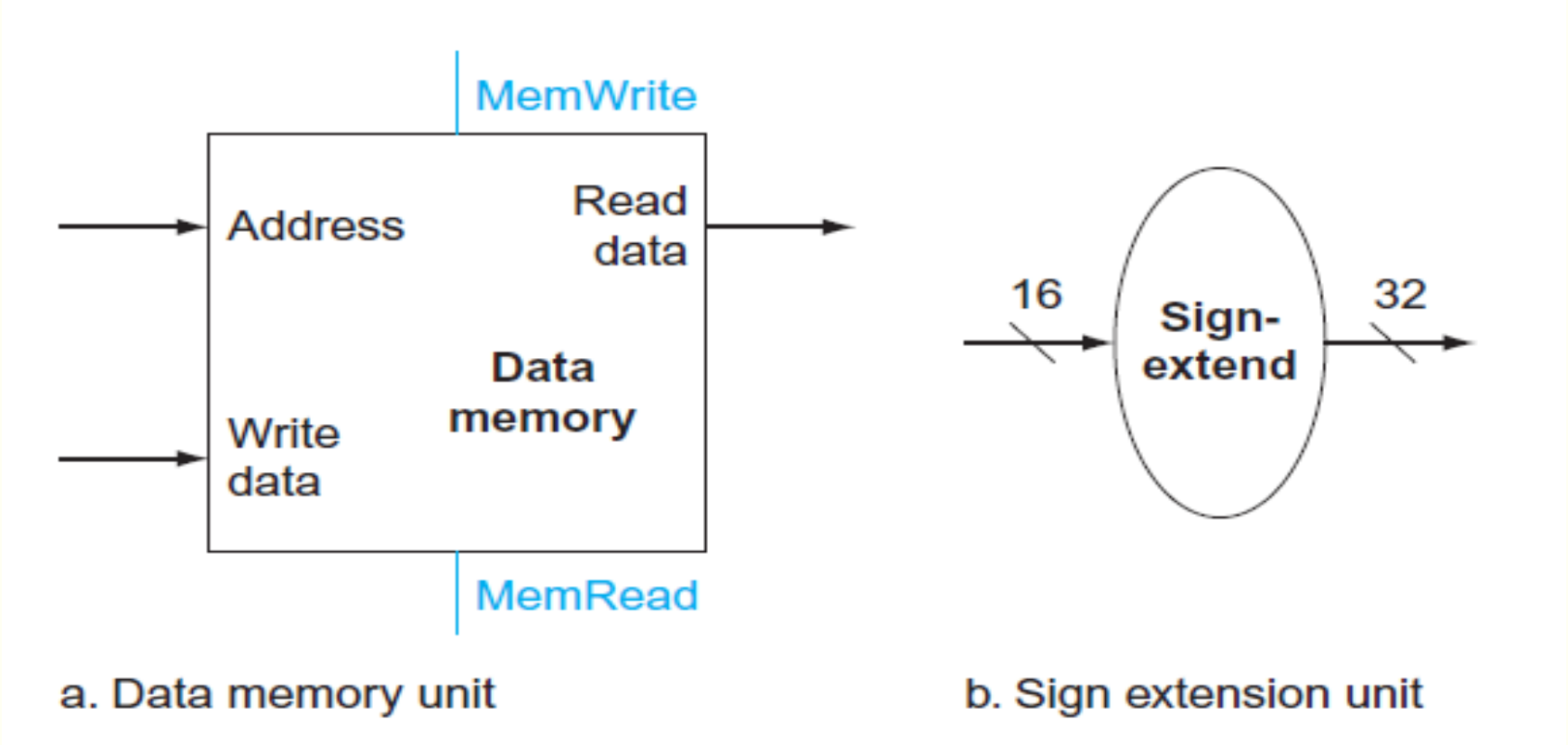
A portion of the datapath used for fetching instructions and incrementing the program counter.

BUILD DATA PATH – ELEMENTS TYPES



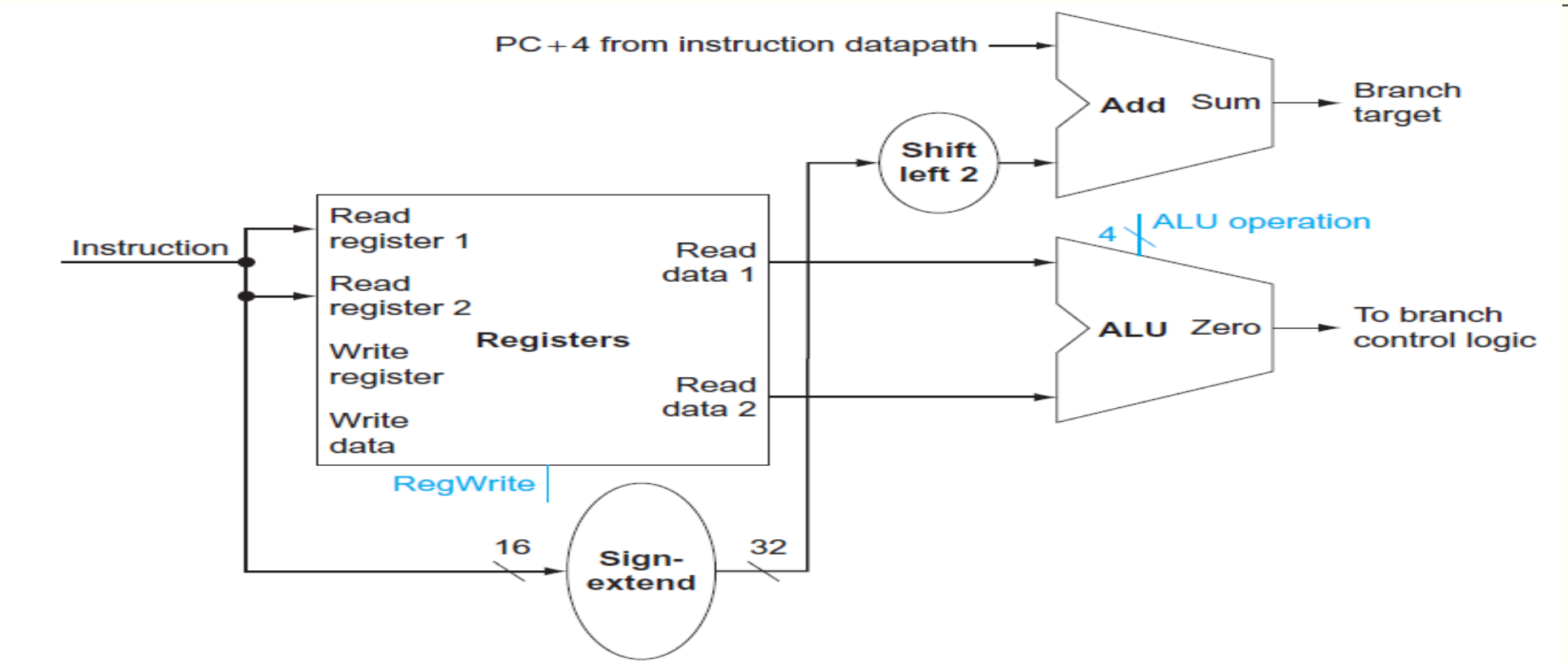
The two elements needed to implement R-format ALU operations are the register file and the ALU.

BUILD DATA PATH – ELEMENTS TYPES



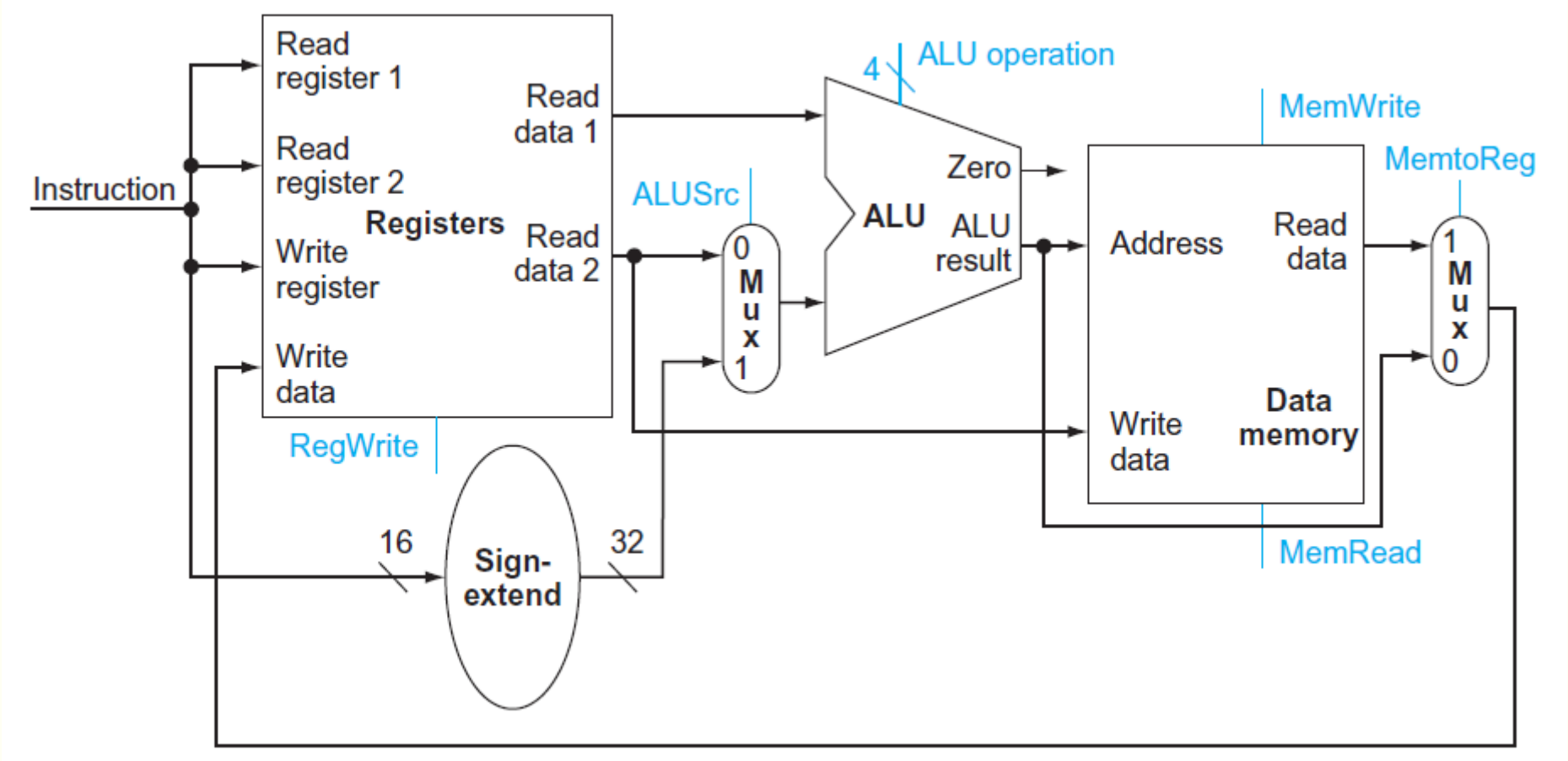
The two units needed to implement loads and stores, in addition to the register file and ALU of Slide no. 25, are the data memory unit and the sign extension unit.

BUILD DATA PATH – ELEMENTS TYPES



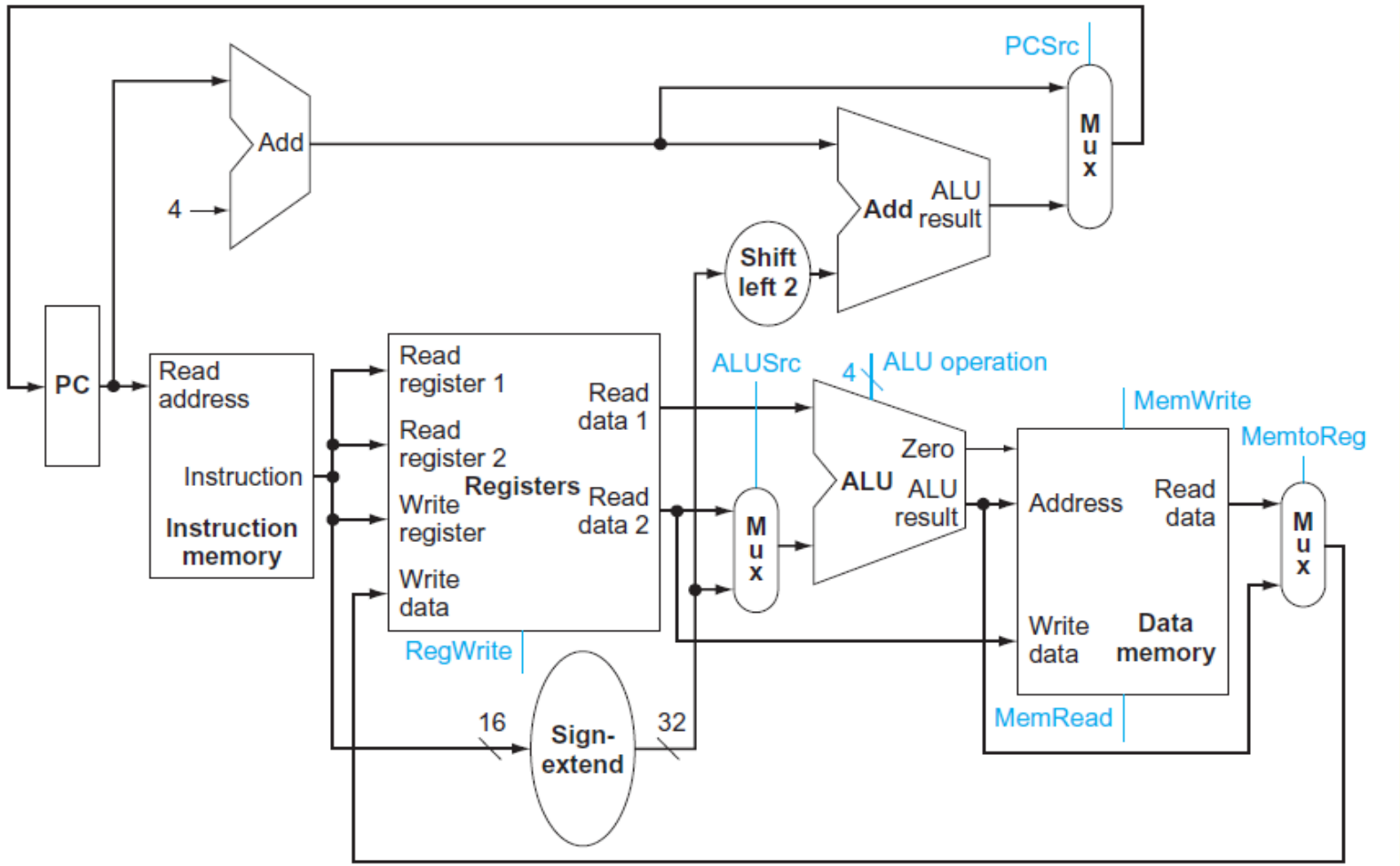
The datapath for a branch uses the ALU to evaluate the branch condition and a separate adder to compute the branch target as the sum of the incremented PC and the sign-extended, lower 16 bits of the instruction (the branch isplacement), shifted left 2 bits.

CREATING A SIMPLE DATAPATH



The datapath for the memory instructions and the R-type instructions.

CREATING A SIMPLE DATAPATH



The simple datapath for the core MIPS architecture combines the elements required by different instruction classes.

A SIMPLE IMPLEMENTATION SCHEME

- The ALU Control
- Designing the Main Control Unit
- Operation of the Datapath
- Finalizing Control

A SIMPLE IMPLEMENTATION SCHEME

The ALU Control

The MIPS ALU defines the 6 following combinations of four control inputs:

ALU Control Lines	Functions
0000	AND
0001	OR
0010	Add
0110	Subtract
0111	Set on less than
1100	NOR

A SIMPLE IMPLEMENTATION SCHEME

- How the ALU control bits are set depends on the ALUOp control bits and the different function codes for the R-type instruction.

Instruction Opcode	ALUOp	Instruction Operation	Function Field	Desired ALU Action	ALU Control Input
LW	00	load word	xxxxxx	add	0010
SW	00	store word	xxxxxx	add	0010
Branch equal	01	Branch equal	xxxxxx	subtract	0110
R-type	10	add	100000 (32)	add	0010
R-type	10	subtract	100010 (34)	Subtract	0110
R-type	10	AND	100100 (36)	AND	0000
R-type	10	OR	100101 (37)	OR	0001
R-type	10	Set on less than	101010 (42)	Set on less than	0111

A SIMPLE IMPLEMENTATION SCHEME

The truth table for the 4 ALU control bits (called Operation).

ALUOp		Function Field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
0	1	X	X	X	X	X	X	0110
1	0	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	0	X	X	0	1	0	0	0000
1	0	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

A SIMPLE IMPLEMENTATION SCHEME

Designing the Main Control Unit

Field	0	rs	rt	rd	shamt	funct
Bit Position	31:26	25:21	20:16	15:11	10:6	5:0

R-type instruction

Field	35 or 43	rs	rt	address
Bit Position	31:26	25:21	20:16	15:0

Load or Store Instruction

Field	4	rs	rt	address
Bit Position	31:26	25:21	20:16	15:0

Branch Instruction

The three instruction classes (R-type, load and store, and branch) use two different instruction formats.

A SIMPLE IMPLEMENTATION SCHEME

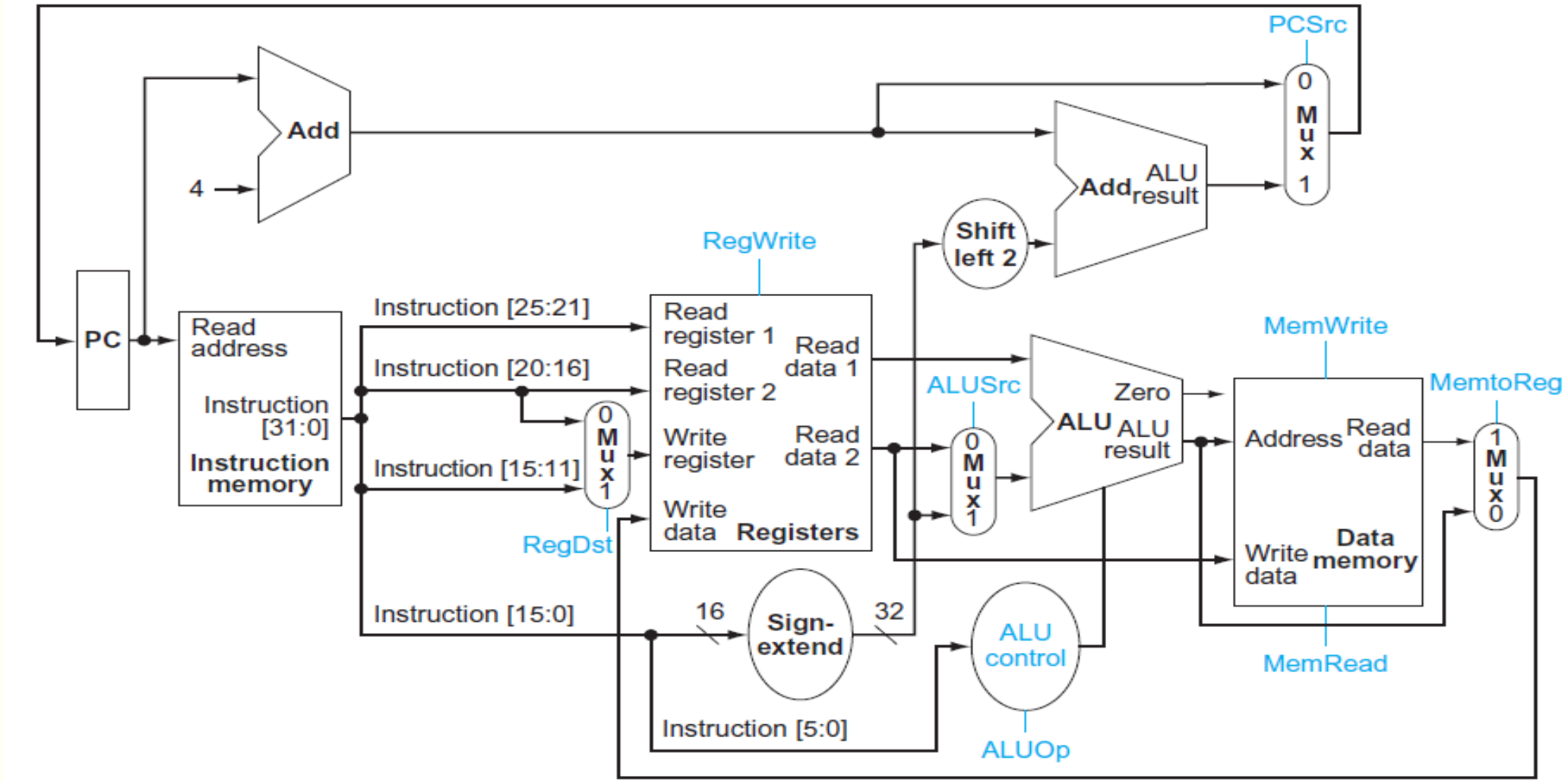
Designing the Main Control Unit

- Opcode – The field that denotes the operation and format of an instruction.
- There are several major observations about this instruction format that we will rely on:
- The op field, also called the opcode, is always contained in bits 31:26. We will refer to this field as Op[5:0].
- The two registers to be read are always specified by the rs and rt fields, at positions 25:21 and 20:16. This is true for the R-type instructions, branch equal, and store.

A SIMPLE IMPLEMENTATION SCHEME

- The base register for load and store instructions is always in bit positions 25:21 (rs).
- The 16-bit off set for branch equal, load, and store is always in positions 15:0.
- The destination register is in one of two places. For a load it is in bit positions 20:16 (rt), while for an R-type instruction it is in bit positions 15:11 (rd).
- Thus, we will need to add a multiplexor to select which field of the instruction is used to indicate the register number to be written.

A SIMPLE IMPLEMENTATION SCHEME



The datapath with all necessary multiplexors and all control lines identified.

A SIMPLE IMPLEMENTATION SCHEME

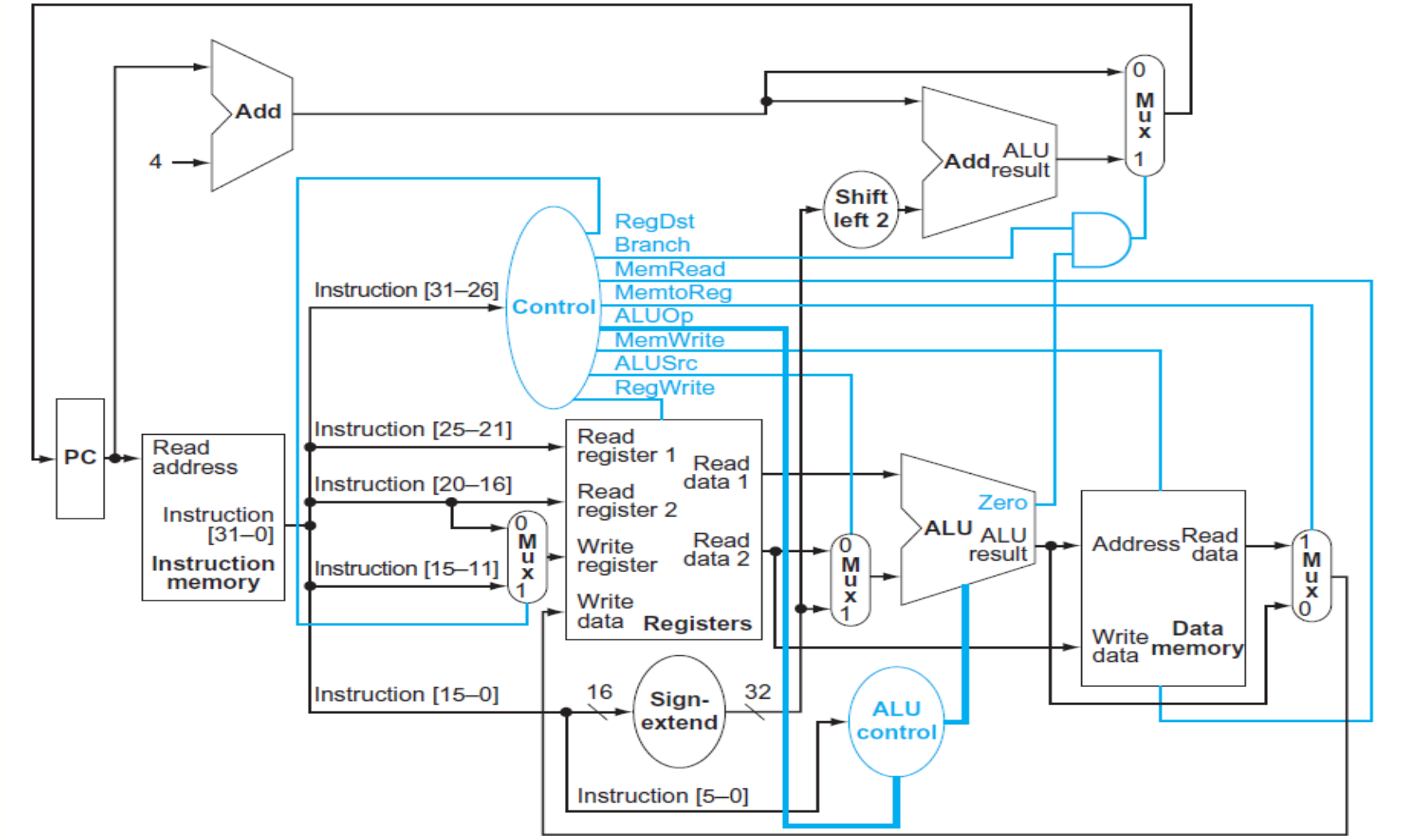
Signal Name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

A SIMPLE IMPLEMENTATION SCHEME

Operation of the Datapath

- The asserted control signals and active datapath elements are highlighted in each of these.
- Note that a multiplexor whose control is 0 has a definite action, even if its control line is not highlighted.
- Multiple-bit control signals are highlighted if any constituent signal is asserted.

A SIMPLE IMPLEMENTATION SCHEME



The simple datapath with the control unit.

A SIMPLE IMPLEMENTATION SCHEME

- The asserted control signals and active datapath elements are highlighted in The datapath for an R-type instruction, such as add $t1,$t2,$t3. Although everything occurs in one clock cycle, we can think of four steps to execute the instruction; these steps are ordered by the flow of information:

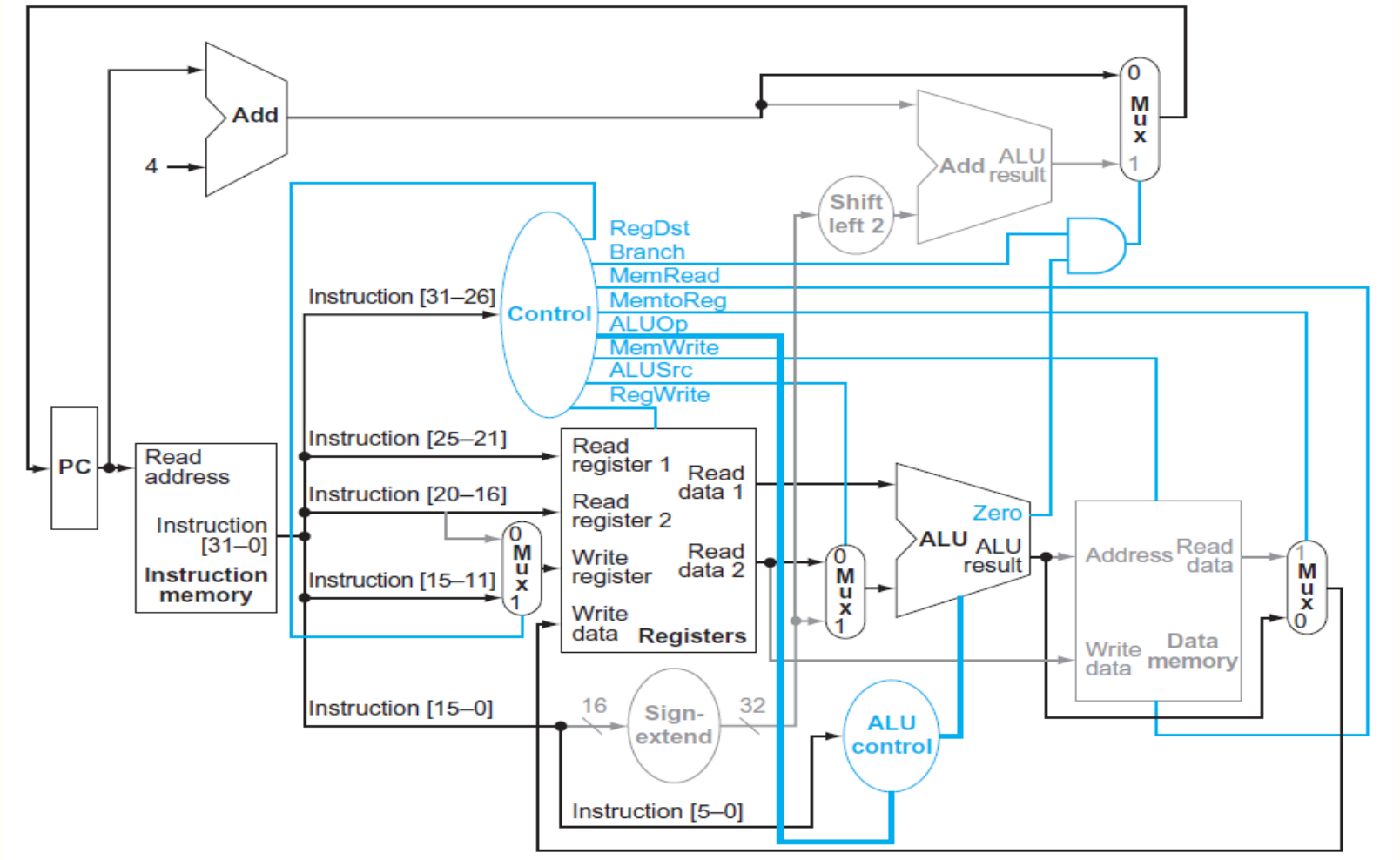
1. The instruction is fetched, and the PC is incremented.
2. Two registers, $t2 and $t3, are read from the register file; also, the main control unit computes the setting of the control lines during this step.
3. The ALU operates on the data read from the register file, using the function code (bits 5:0, which is the functfield, of the instruction) to generate the ALU function.
4. The result from the ALU is written into the register file using bits 15:11 of the instruction to select the destination register ($t1).

A SIMPLE IMPLEMENTATION SCHEME

The setting of the control lines is completely determined by the opcode fields of the instruction.

Instruction	Reg Dst	ALU Src	Mem to-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALU Op1	ALU Op0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	x	1	x	0	0	1	0	0	0
beq	x	0	x	0	0	0	1	0	1

A SIMPLE IMPLEMENTATION SCHEME



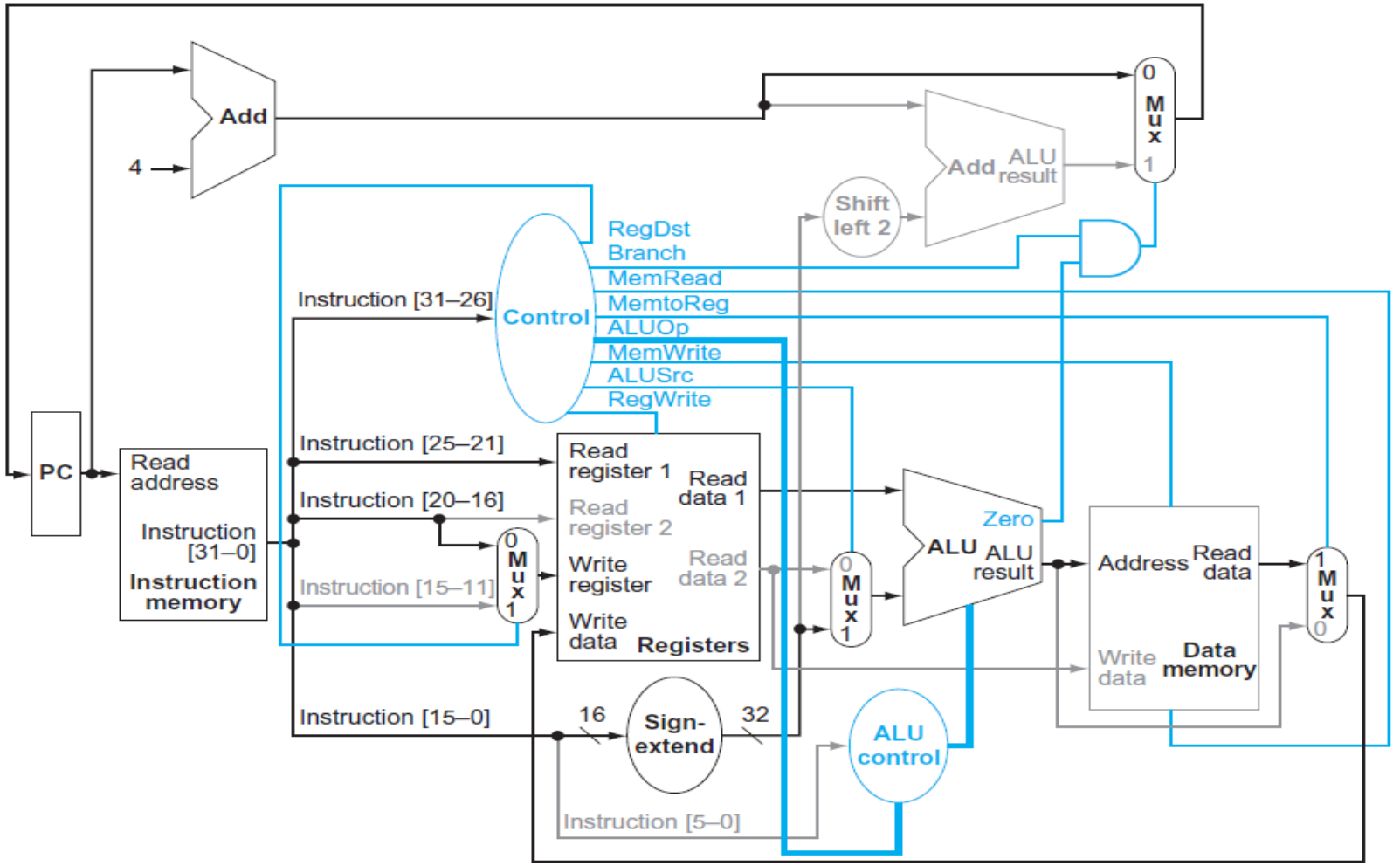
The datapath in operation for an R-type instruction, such as add $t1,$t2,$t3.

A SIMPLE IMPLEMENTATION SCHEME

- lw $t1, offset($t2)
- The active functional units and asserted control lines for a load. We can think of a load instruction as operating in five steps:

1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. A register ($t2) value is read from the register file.
3. The ALU computes the sum of the value read from the register file and the sign-extended, lower 16 bits of the instruction (offset).
4. The sum from the ALU is used as the address for the data memory.
5. The data from the memory unit is written into the register file; the register destination is given by bits 20:16 of the instruction ($t1).

A SIMPLE IMPLEMENTATION SCHEME



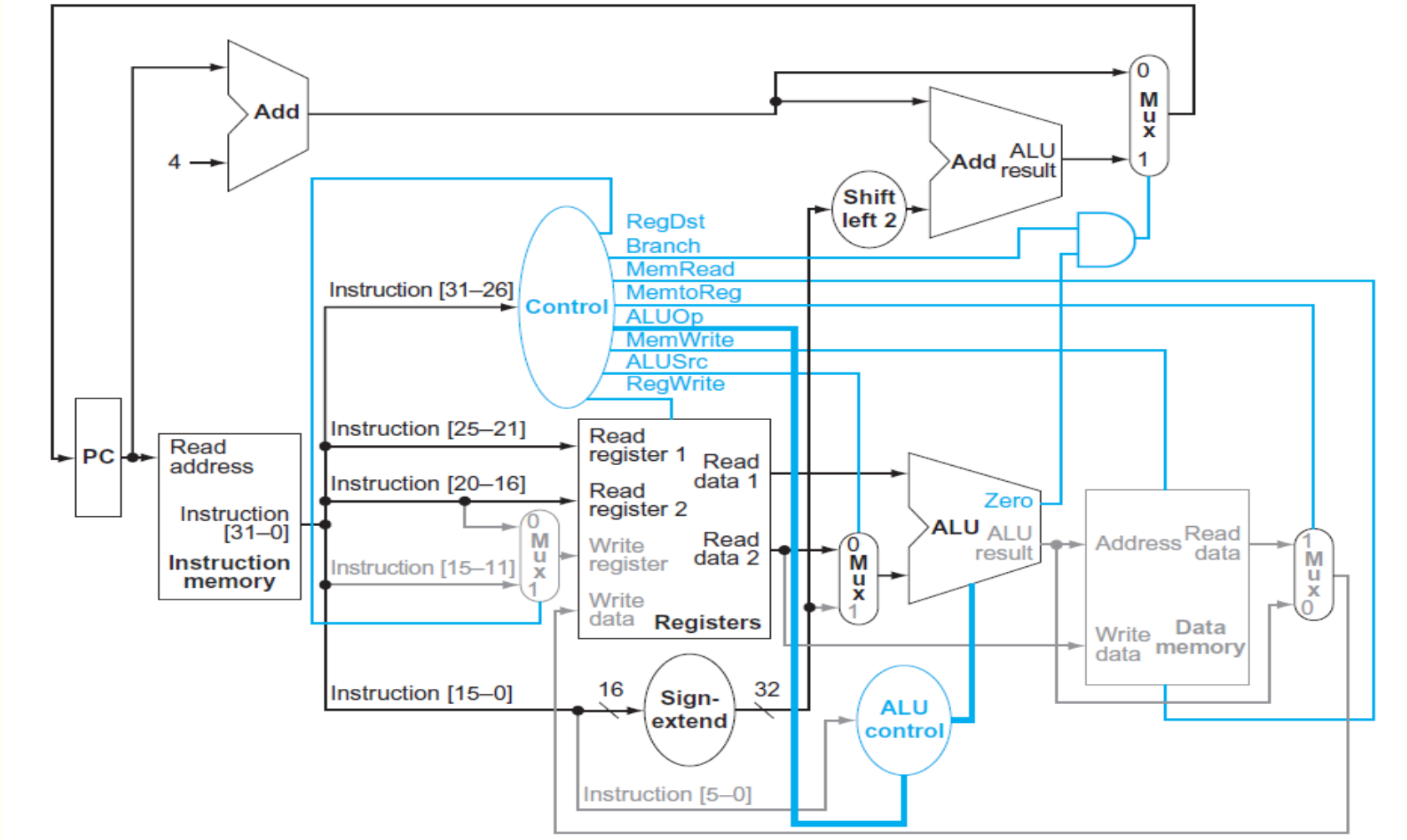
The datapath in operation for a load instruction.

A SIMPLE IMPLEMENTATION SCHEME

- Finally, we can show the operation of the branch-on-equal instruction, such as beq $t1, $t2, offset, in the same fashion. It operates much like an R-format instruction, but the ALU output is used to determine whether the PC is written with PC + 4 or the branch target address. The four steps in execution:

1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. Two registers, $t1 and $t2, are read from the register file.
3. The ALU performs a subtract on the data values read from the register file. The value of PC + 4 is added to the sign-extended, lower 16 bits of the instruction (offset) shifted left by two; the result is the branch target address.
4. The Zero result from the ALU is used to decide which adder result to store into the PC.

A SIMPLE IMPLEMENTATION SCHEME



The datapath in operation for a branch-on-equal instruction.

A SIMPLE IMPLEMENTATION SCHEME

Finalizing Control

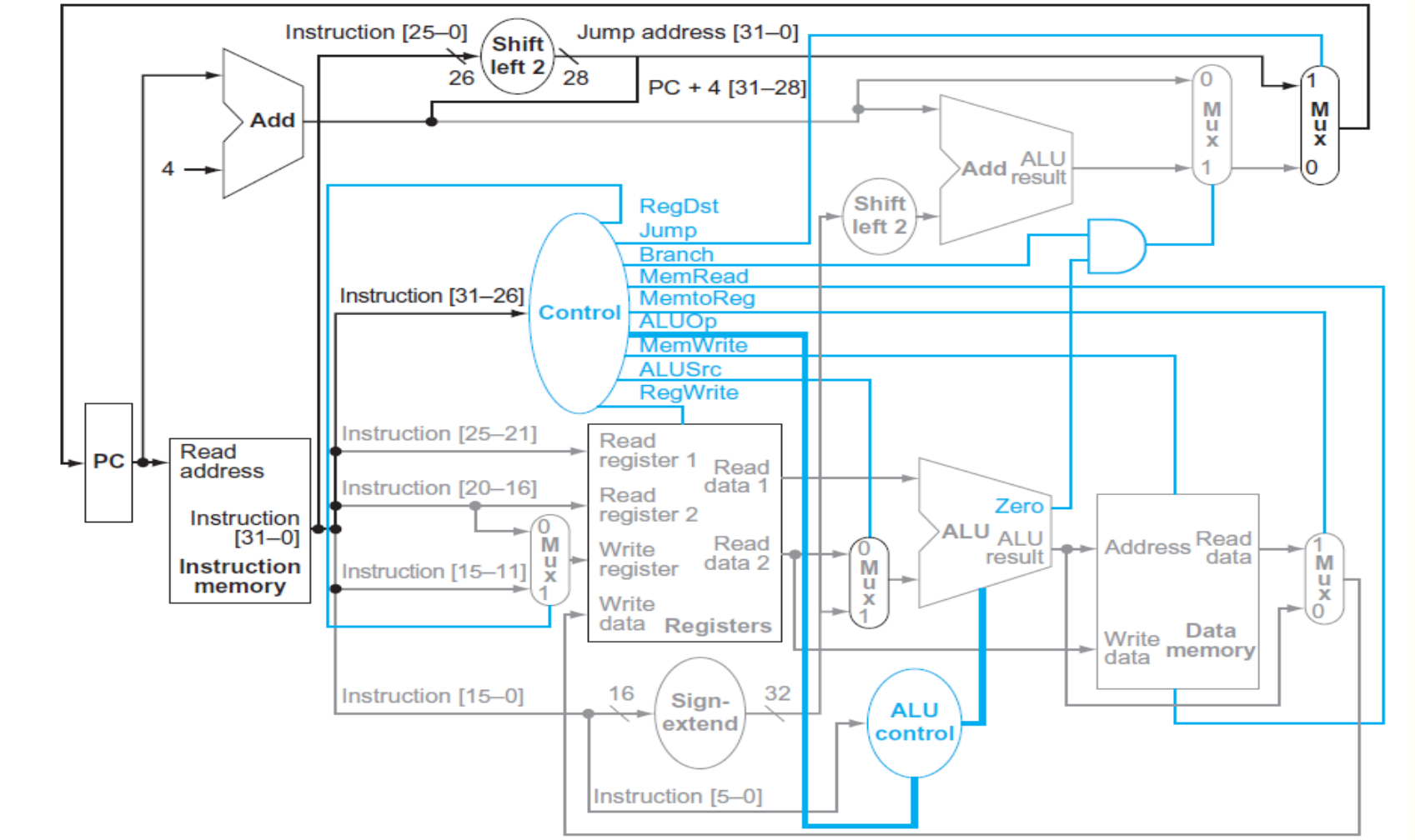
- The control function can be precisely defined using the contents of Slide no. 42.
- The outputs are the control lines, and the input is the 6-bit opcode field, Op [5:0].
- Thus, we can create a truth table for each of the outputs based on the binary encoding of the opcodes.

A SIMPLE IMPLEMENTATION SCHEME

Input or Output	Signal Name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	x	x
	ALUSrc	0	1	1	0
	MemtoReg	0	1	x	x
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

The control function for the simple single-cycle implementation is completely specified by this truth table.

A SIMPLE IMPLEMENTATION SCHEME



The simple control and datapath are extended to handle the jump instruction.

DRAWBACKS

- The single-cycle datapath is not used in modern processors, because it is inefficient
- Long cycle time
- Cycle time is much longer than needed for all other instructions
- All instructions take the same time although
- Some combinational units must be replicated since used in the same cycle

INTRODUCTION

- Instruction Level Parallelism
- Pipelining
- Overview of pipelining
- Performance
- Pipeline Hazards
- Pipelined datapath and control
- Handling data hazards and control hazards
- Exceptions

INSTRUCTION LEVEL PARALLELISM

- Instruction-level Parallelism (ILP) is a family of processor and compiler design techniques that speed up execution by causing individual machine operations to execute in parallel.
- Instruction Level Parallelism (ILP) is used to refer to the architecture in which multiple operations can be performed parallelly in a particular process, with its own set of resources – address space, registers, identifiers, state, program counters.
- It refers to the compiler design techniques and processors designed to execute operations, like memory load and store, integer addition, float multiplication, in parallel to improve the performance of the processors.
- Examples of architectures that exploit ILP are VLIWs, Superscalar Architecture.

INSTRUCTION LEVEL PARALLELISM

- Pipelining can overlap the execution of instructions when they are independent of one another.
- This potential overlap among instructions is called instruction-level parallelism (ILP) since the instructions can be evaluated in parallel.
- ILP is a measure of how many of the operations in a computer program can be performed simultaneously.
- Consider the following program:
- e = a + b
- f = c + d
- g =e * f

INSTRUCTION LEVEL PARALLELISM

- Here, Operation 3 depends on the results of operations 1 and 2, so it cannot be calculated until both of them are completed.
- As, operations 1 and 2 do not depend on any other operation, so they can be calculated simultaneously.
- If each operation is completed in one unit of time then three instructions can be completed in two units of time, giving an ILP of 3/2.
- Ordinary programs are written and executed sequentially.
- ILP allows the compiler and the processor to overlap the execution of multiple instructions or even to change the order in which instructions are executed.

PIPELINING

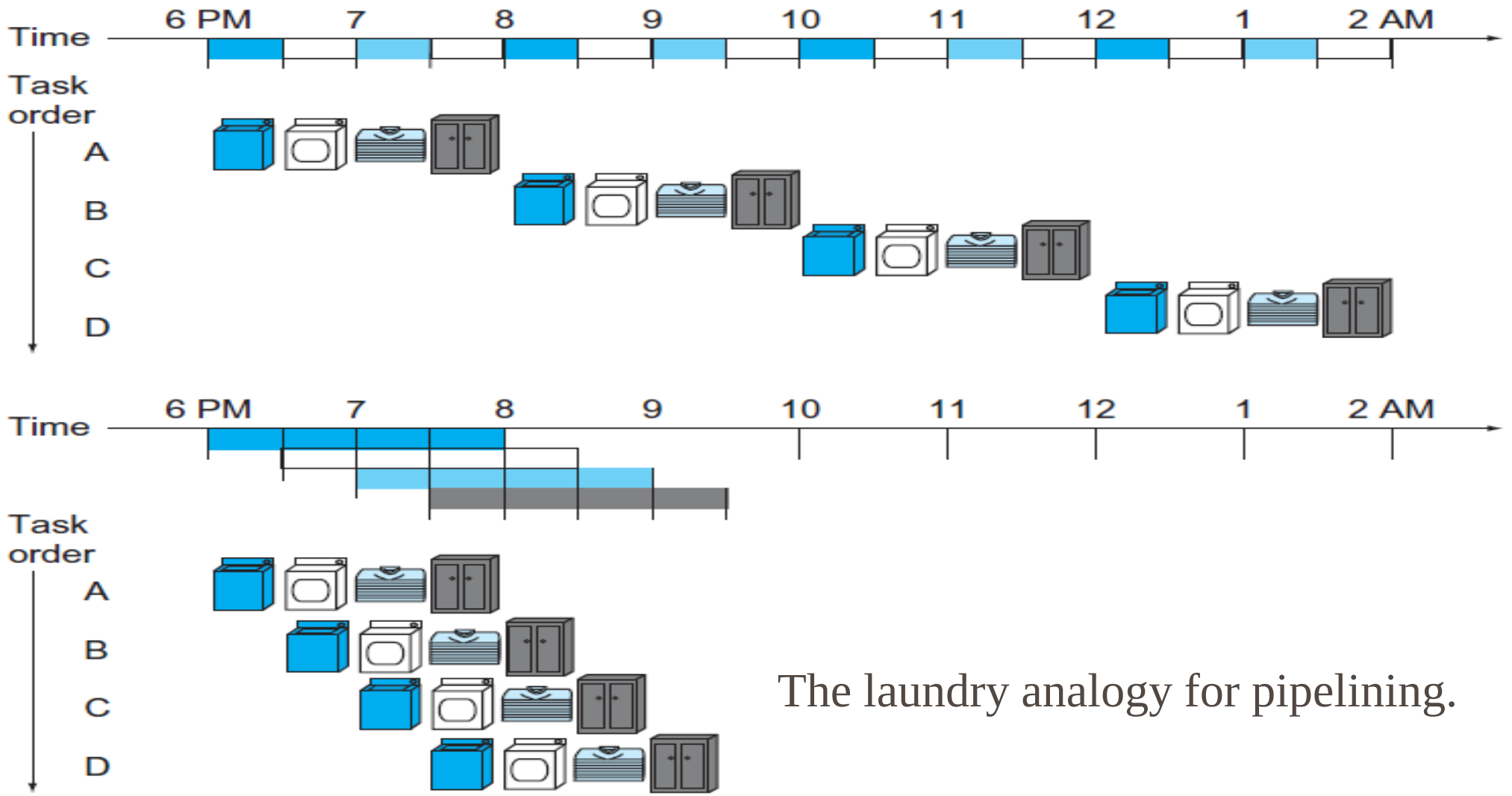
- Pipelining is a process of arrangement of hardware elements of the CPU such that its overall performance is increased.
- Simultaneous execution of more than one instruction takes place in a pipelined processor.
- Pipelining is an implementation technique where multiple instructions are overlapped in execution. The computer pipeline is divided in stages.

PIPELINING

- The non-pipelined approach to laundry would be as follows:

1. Place one dirty load of clothes in the washer.
2. When the washer is finished, place the wet load in the dryer.
3. When the dryer is finished, place the dry load on a table and fold.
4. When folding is finished, ask your roommate to put the clothes away.

PIPELINING



The laundry analogy for pipelining.

PIPELINING INSTRUCTION EXECUTION

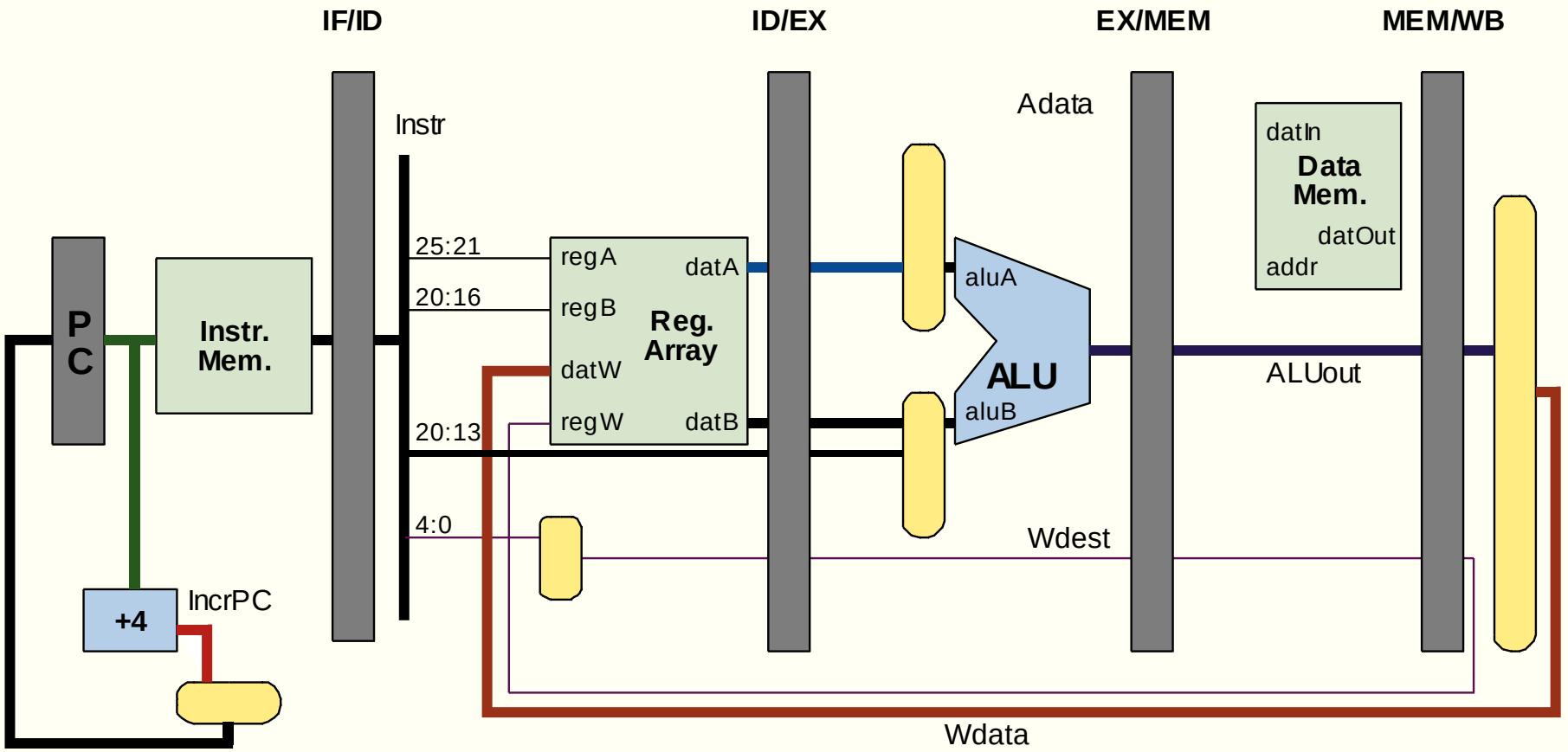
1. IF – Instruction Fetch
2. ID – Instruction Decode / Register Fetch
3. EX – Execute
4. MEM – Memory Access
5. WB – Write Back

PIPELINING INSTRUCTION EXECUTION

1. IF – Instruction Fetch
   - Fetch instruction from memory.
2. ID – Instruction Decode / Register Fetch
   - Read registers while decoding the instruction. The regular format of MIPS instructions allows reading and decoding to occur simultaneously.
3. EX – Execute
   - Execute the operation or calculate an address.
4. MEM – Memory Access
   - Access an operand in data memory.
5. WB – Write Back
   - Write the result into a register.

PIPELINING

IF	ID	EX	MEM	WB
instruction fetch	instruction decode/ register fetch	execute	memory access	write back



OVERVIEW PIPELINING

- Types of Pipelining
- Performance
- Pipeline Hazards

TYPES OF PIPELINING

Types of Pipelining

- **Arithmetic Pipelining**
  - It is designed to perform high-speed floating-point addition, multiplication and division.
  - Here, the multiple arithmetic logic units are built in the system to perform the parallel arithmetic computation in various data format.
  - Examples of the arithmetic pipelined processor are Star-100, TI-ASC, Cray-1, Cyber-205.
- **Instruction Pipelining**
  - Here, the number of instruction are pipelined and the execution of current instruction is overlapped by the execution of the subsequent instruction.
  - It is also called instruction look-ahead.

PIPELINING - PERFORMANCE

Performance

- The potential increase in performance resulting from pipelining is proportional to the number of pipeline stages.
- However, this increase would be achieved only if all pipeline stages require the same time to complete, and there is no interruption throughout program execution.
- Unfortunately, this is not true.
- The previous pipeline is said to have been stalled for two clock cycles.
- Any condition that causes a pipeline to stall is called a hazard.

PIPELINING - PERFORMANCE

Performance

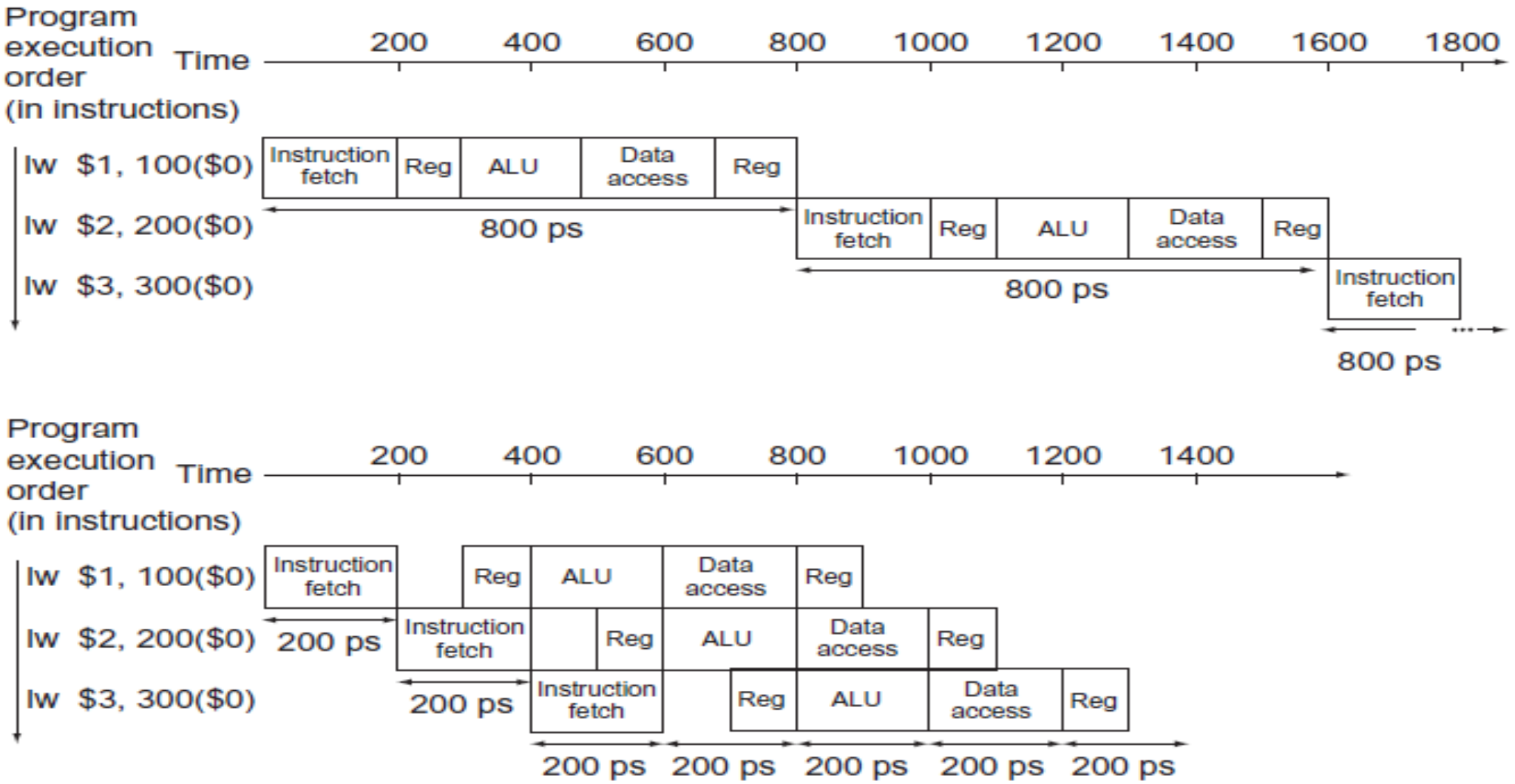
- Again, pipelining does not result in individual instructions being executed faster; rather, it is the throughput that increases.
- Throughput is measured by the rate at which instruction execution is completed.
- Pipeline stall causes degradation in pipeline performance.
- We need to identify all hazards that may cause the pipeline to stall and to find ways to minimize their impact.

PIPELINING - PERFORMANCE

Total time for each instruction calculated from the time for each component.

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

PIPELINING - PERFORMANCE



PIPELINING HAZARDS

- The previous pipeline is said to have been stalled for two clock cycles.
- Any condition that causes a pipeline to stall is called a hazard.

Pipeline Hazards

- Data Hazards
- Instruction (Control) Hazards
- Structural Hazards

PIPELINING HAZARDS

Data Hazards

- Any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline.
- So some operation has to be delayed, and the pipeline stalls.

Instruction (Control) Hazards

- A delay in the availability of an instruction causes the pipeline to stall.

Structural Hazards

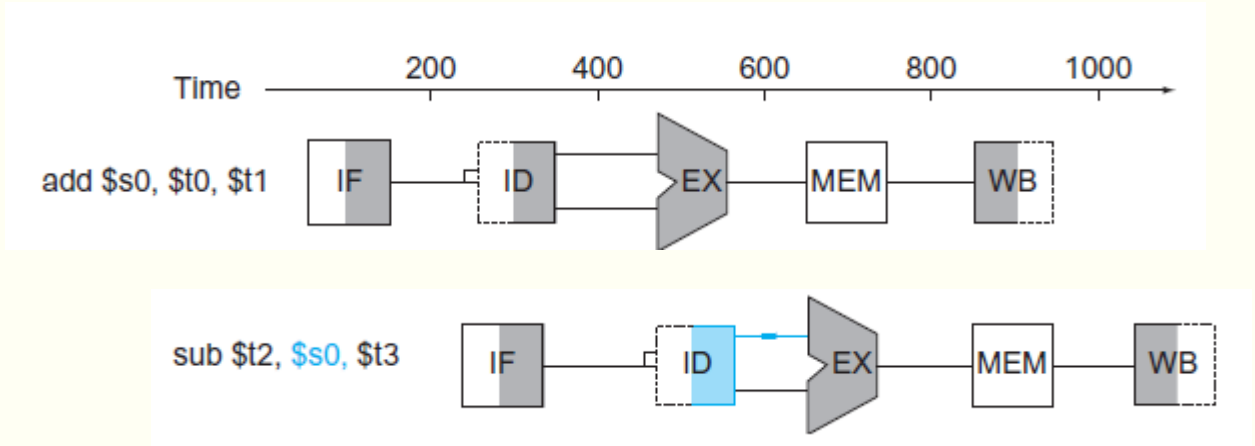
- The situation when two instructions require the use of a given hardware resource at the same time.

PIPELINING – DATA HAZARDS

- When a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available.
- Example:

  add $s0, $t0, $t1

  sub $t2, $s0, $t3

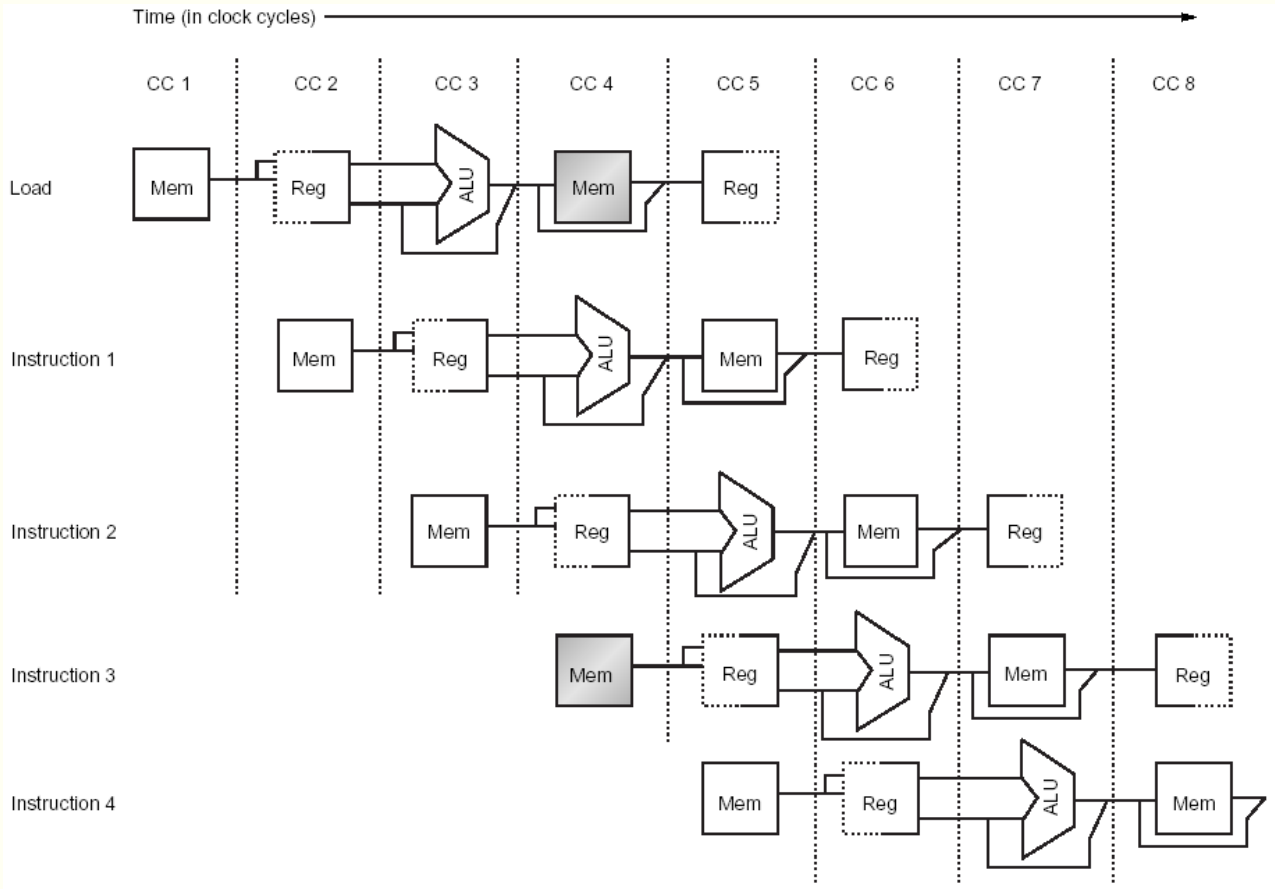


Graphical representation of the instruction pipeline

PIPELINING – STRUCTURAL HAZARDS

- When a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute.
- Example:

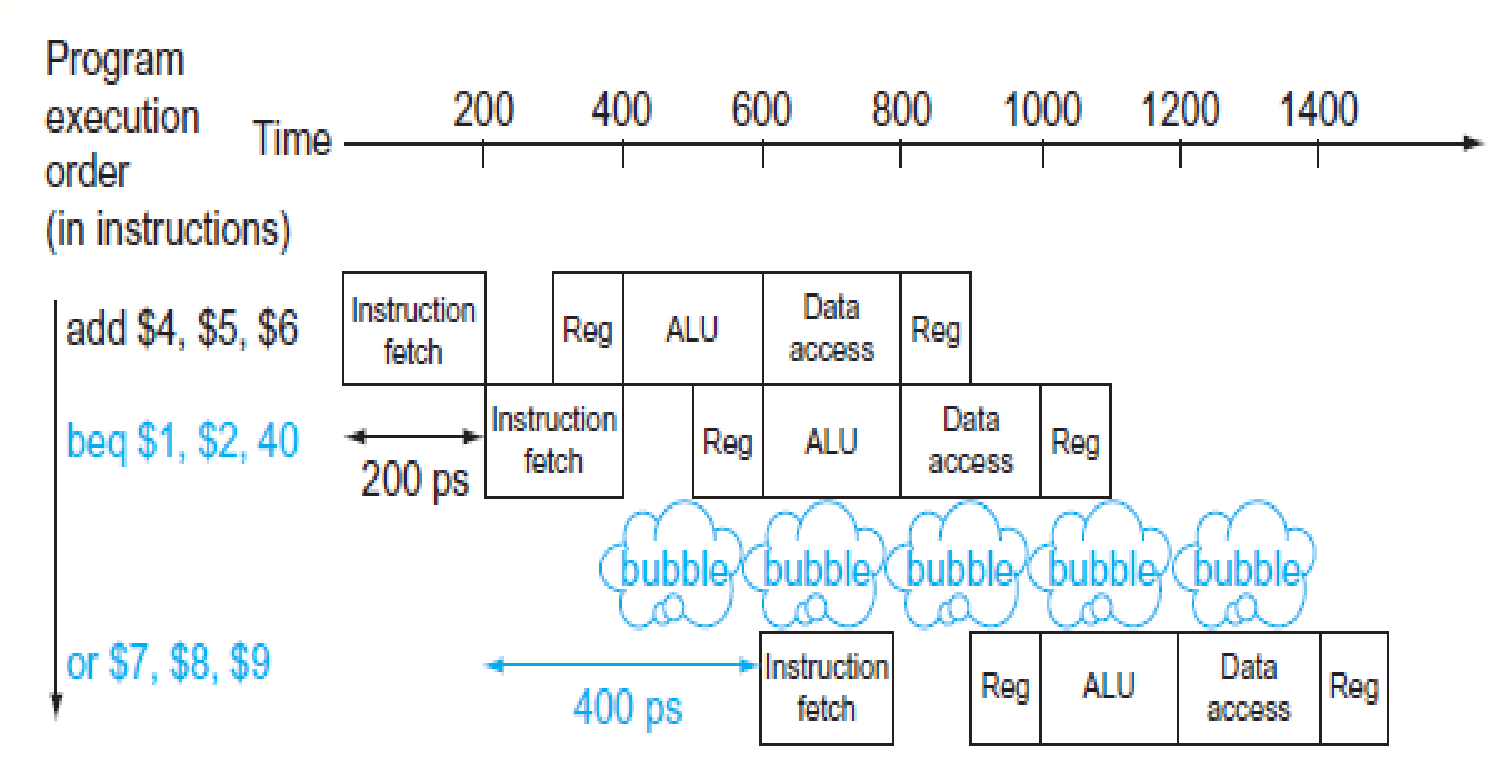
  Same memory for instructions and data



Graphical representation of the structural pipeline

INSTRUCTION (Control) HAZARDS

- Example:
  - add $4, $5, $6
  - beq $1, $2, 40
  - or $7, $8, $9



- This example assumes the conditional branch is taken, and the instruction at the destination of the branch is the OR instruction. There is a one-stage pipeline stall, or bubble, aft er the branch.

INSTRUCTION (Control) HAZARDS

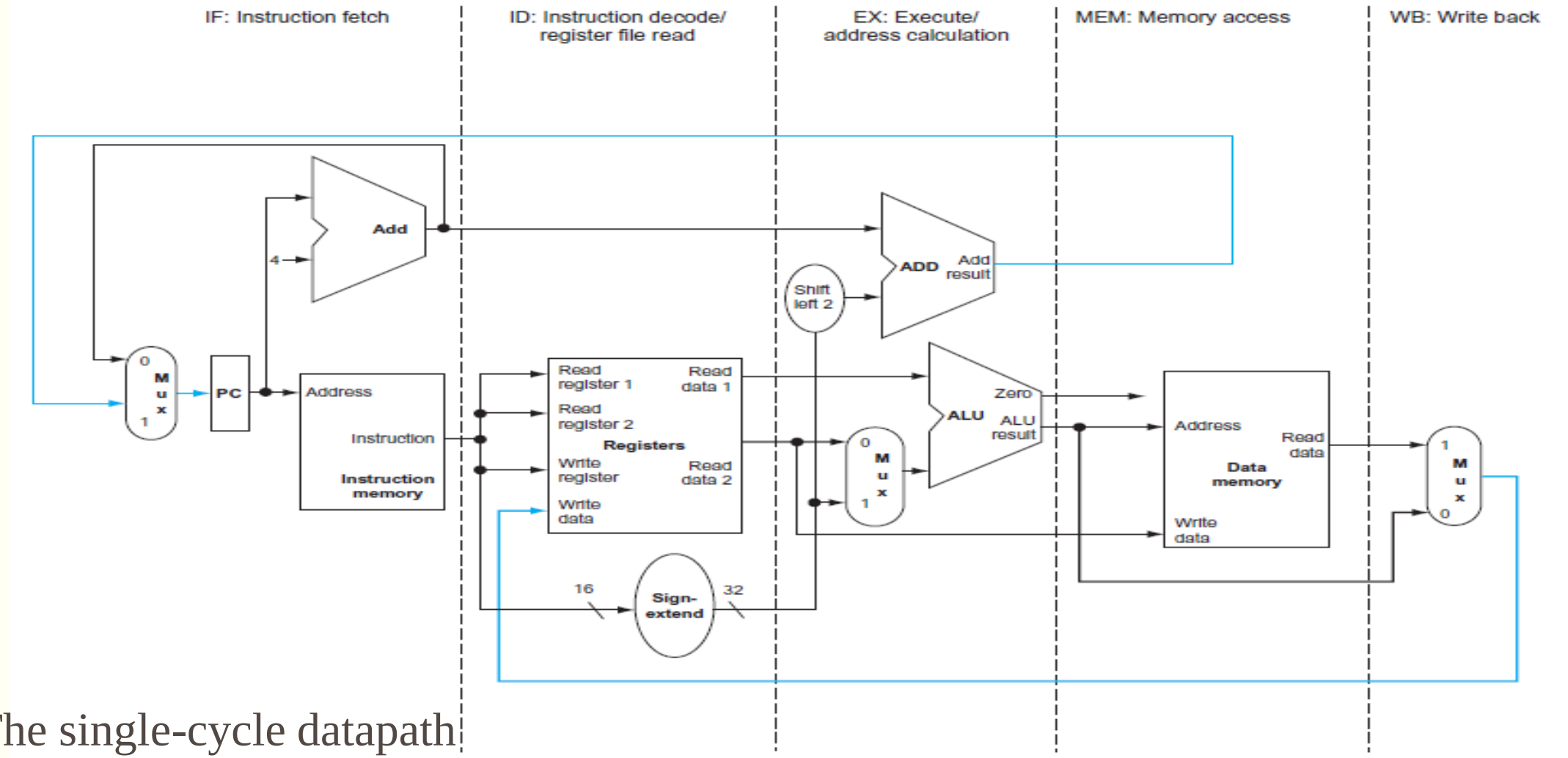
- When the proper instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched is not the one that is needed; that is, the flow of instruction addresses is not what the pipeline expected.
- Example:
  - add $4, $5, $6
  - beq $1, $2, 40
  - or $7, $8, $9

PIPELINED DATAPATH AND CONTROL

- The datapath into five pieces, with each piece named corresponding to a stage of instruction execution:

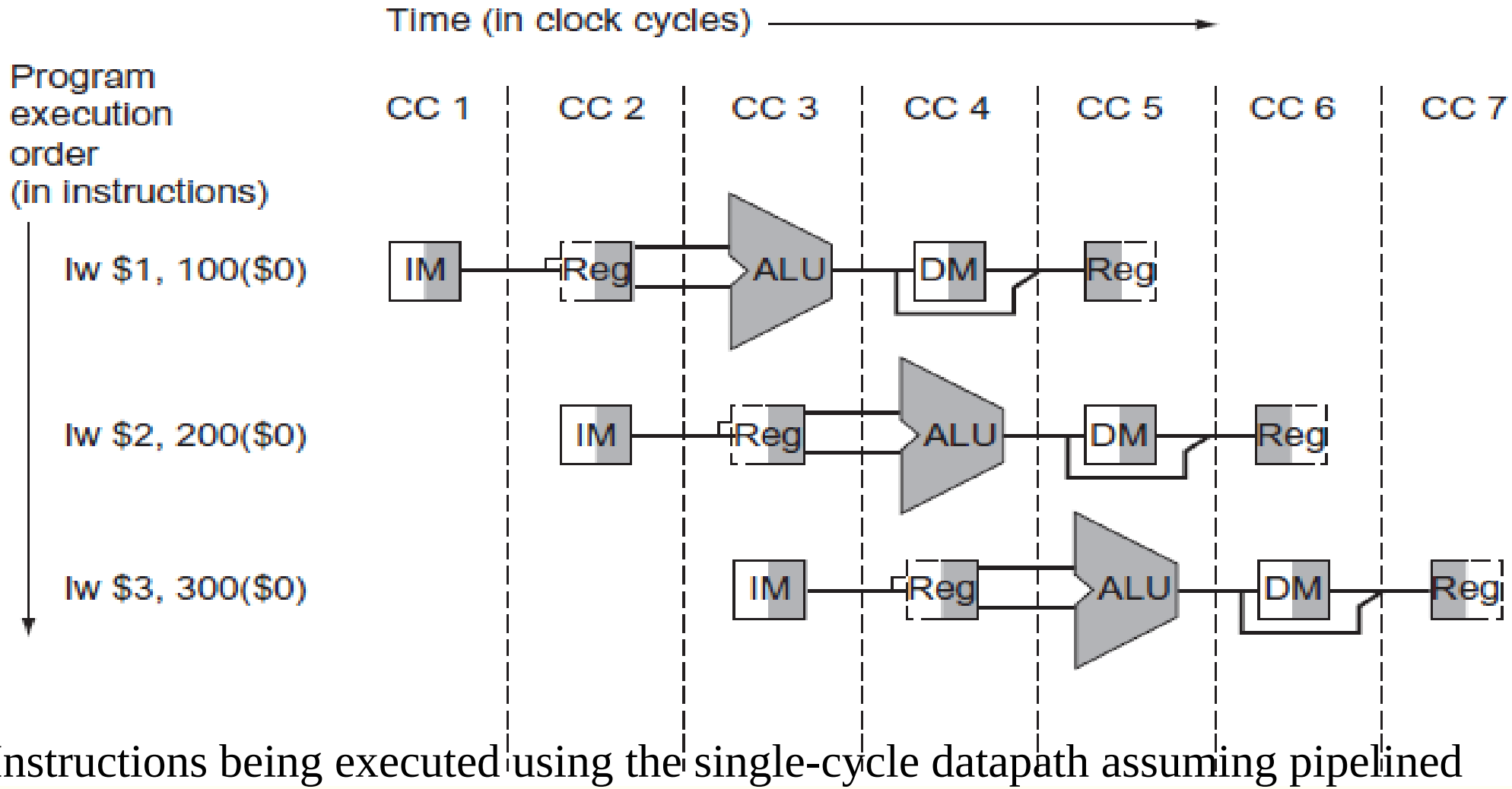
1. IF – Instruction Fetch
2. ID – Instruction Decode / Register Fetch
3. EX – Execute
4. MEM – Memory Access
5. WB – Write Back

PIPELINED DATAPATH AND CONTROL



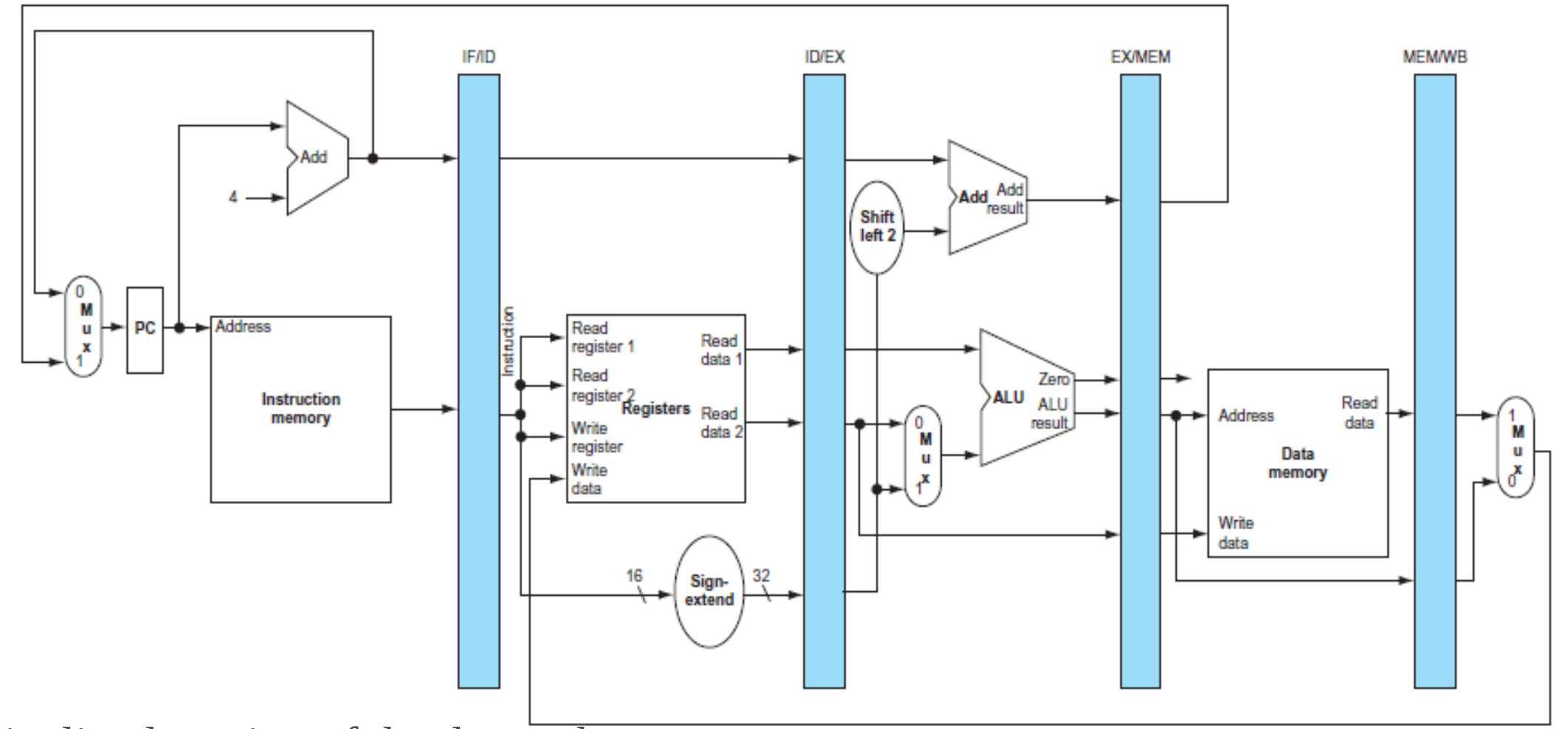
The single-cycle datapath

PIPELINED DATAPATH AND CONTROL



Instructions being executed using the single-cycle datapath assuming pipelined execution

PIPELINED DATAPATH AND CONTROL



The pipelined version of the datapath

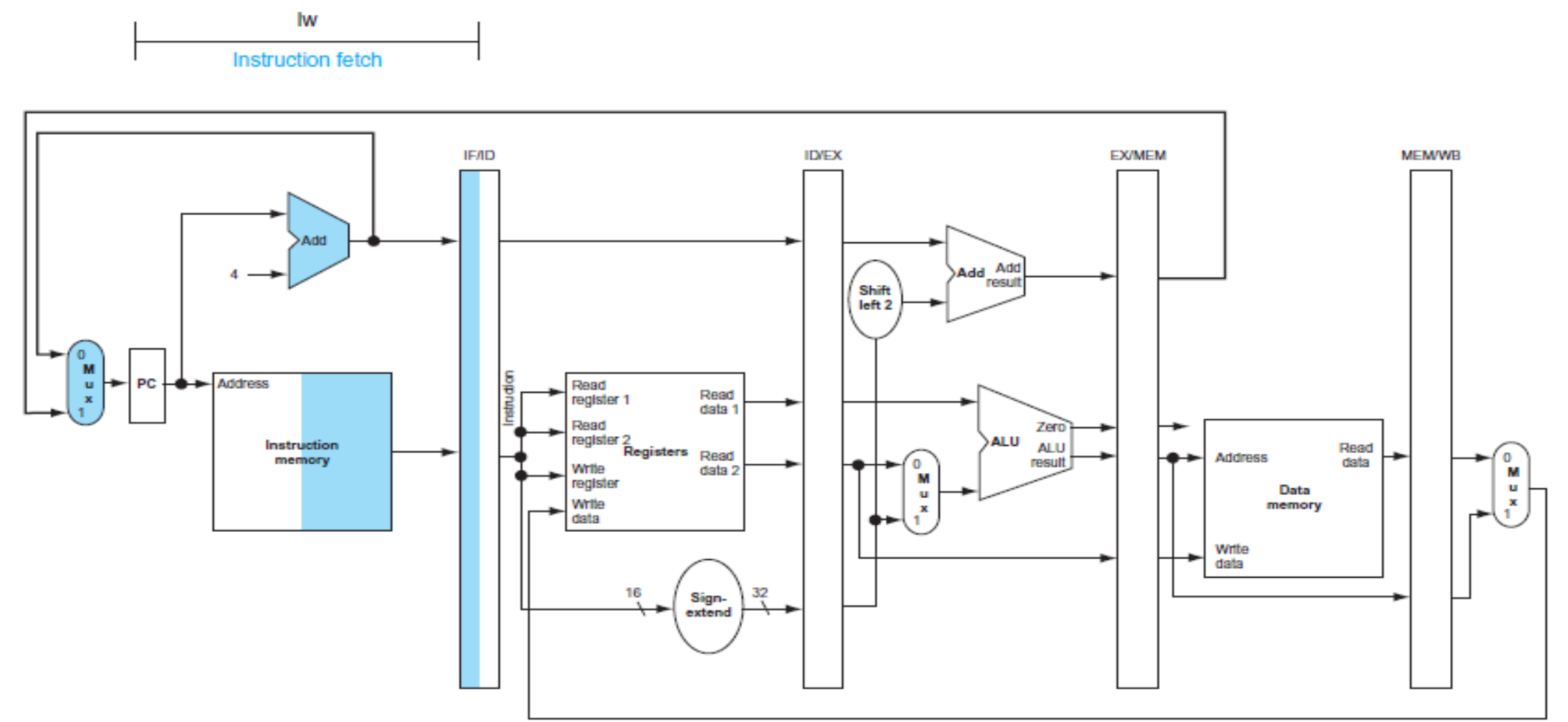
PIPELINED DATAPATH AND CONTROL

- **The five stages are the following:**

1. INSTRUCTION FETCH:

- The top portion of 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.
- The computer cannot know which type of instruction is being fetched, so it must prepare for any instruction, passing potentially needed information down the pipeline.

PIPELINED DATAPATH AND CONTROL



INSTRUCTION FETCH

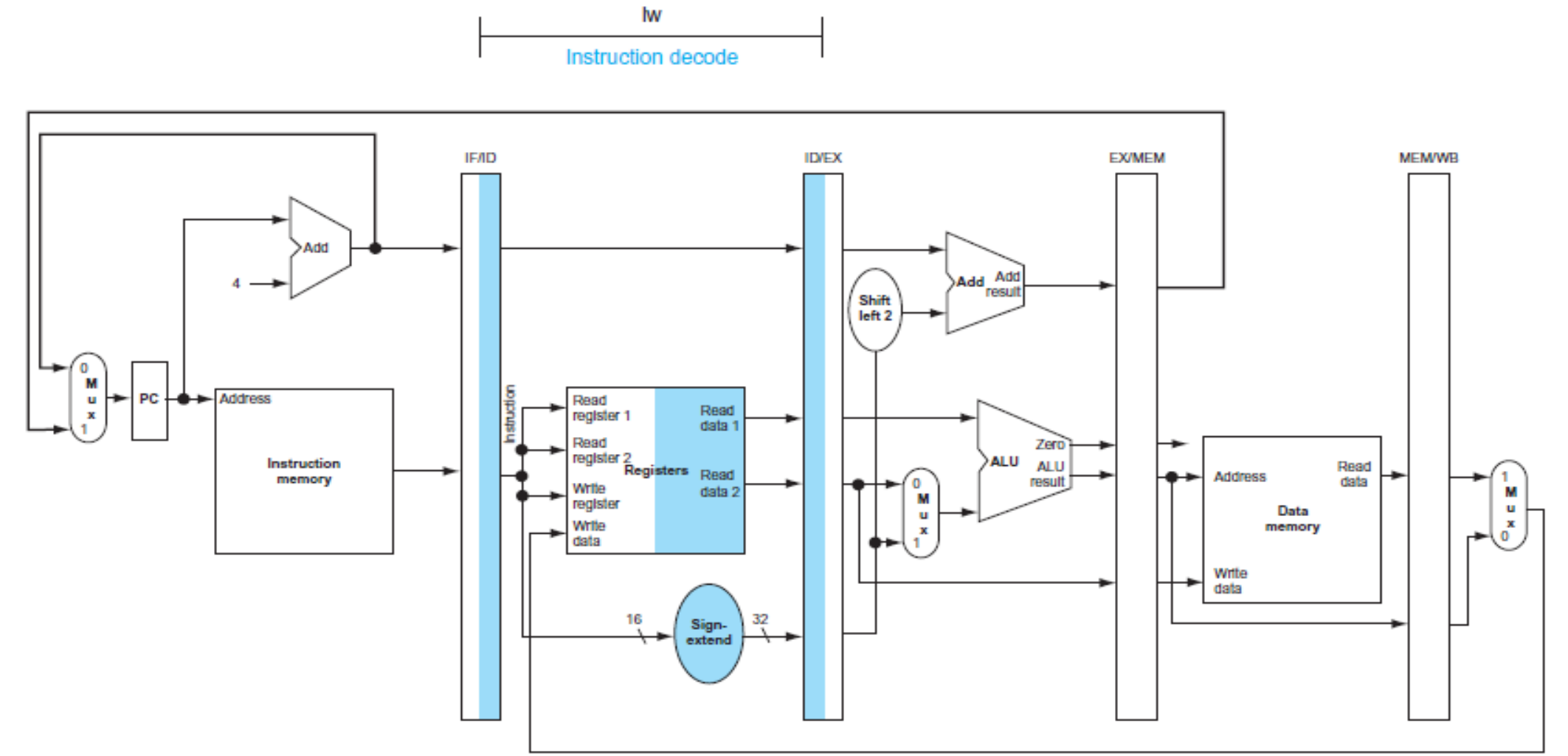
PIPELINED DATAPATH AND CONTROL

- **The five stages are the following:**

2. INSTRUCTION DECODE AND REGISTER FILE READ:

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

PIPELINED DATAPATH AND CONTROL



INSTRUCTION DECODE AND REGISTER FILE READ

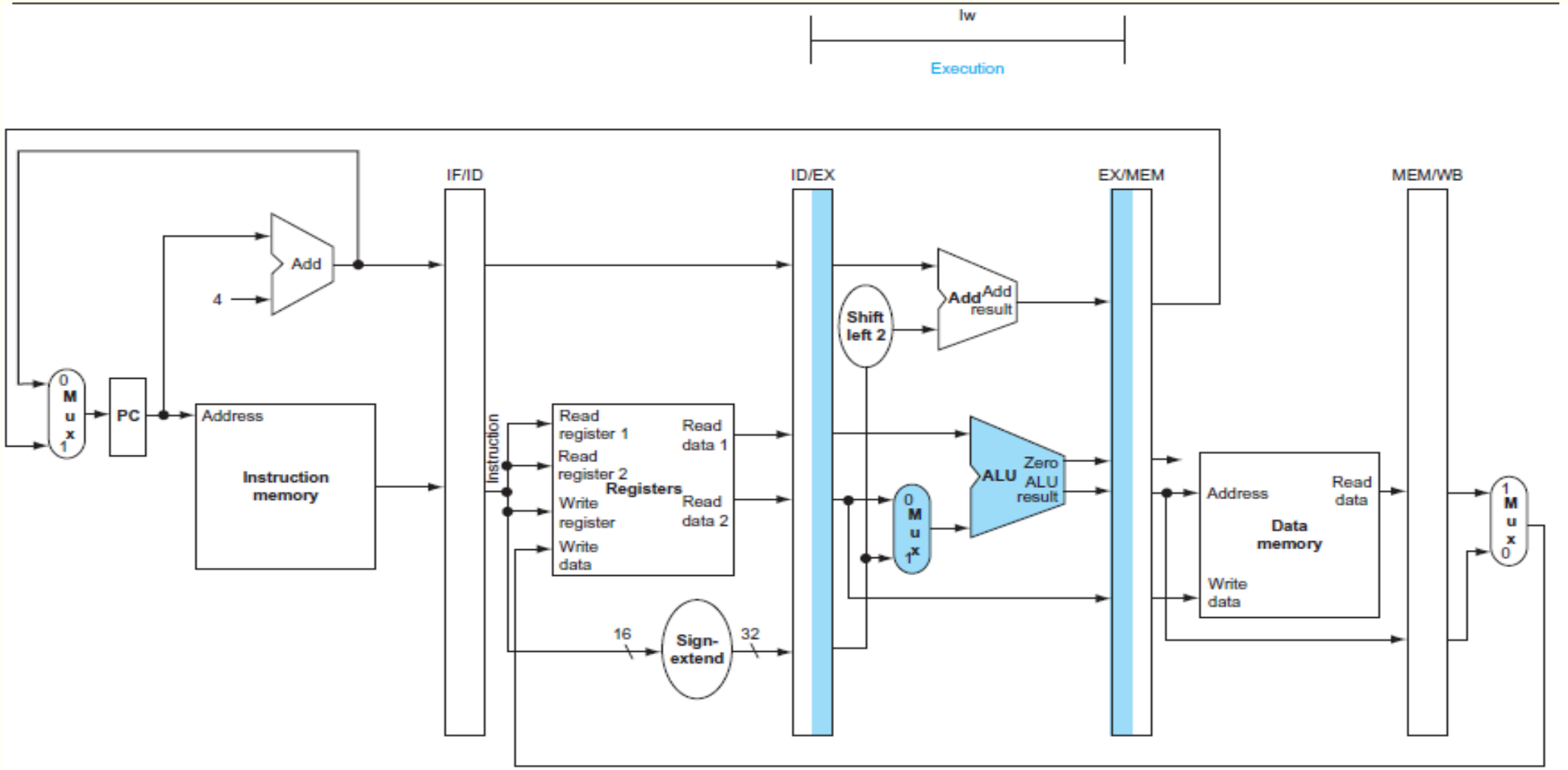
PIPELINED DATAPATH AND CONTROL

- **The five stages are the following:**

3. EXECUTE OR ADDRESS CALCULATION:

- The load instruction reads the contents of register 1 and the sign-extended immediate from the ID/EX pipeline register and adds them using the ALU.
- That sum is placed in the EX/MEM pipeline register.

PIPELINED DATAPATH AND CONTROL



EXECUTE OR ADDRESS CALCULATION

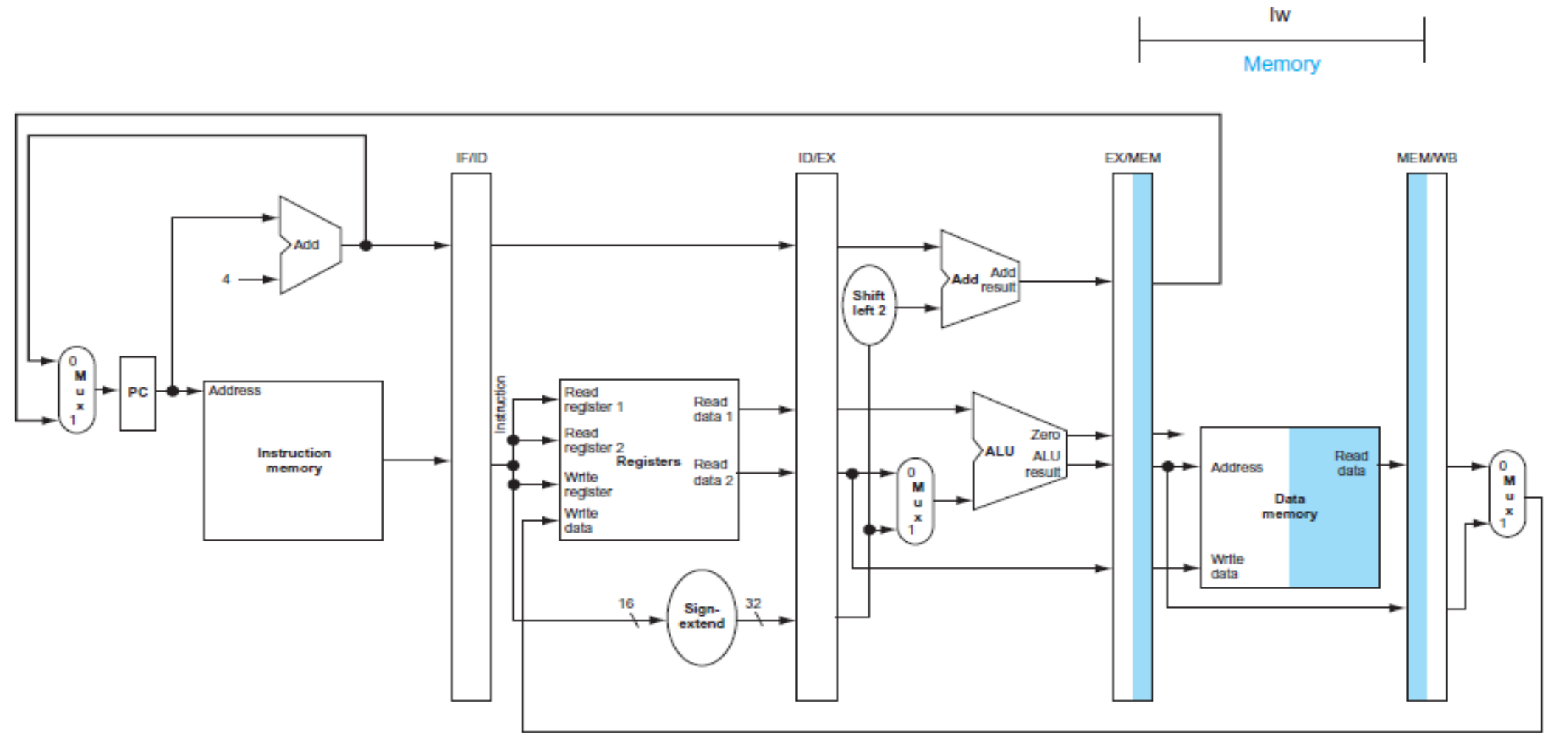
PIPELINED DATAPATH AND CONTROL

- **The five stages are the following:**

4. MEMORY ACCESS:

- The top portion of the load instruction reading the data memory using the address from the EX/MEM pipeline register and loading the data into the MEM/WB pipeline register.

PIPELINED DATAPATH AND CONTROL



MEMORY ACCESS

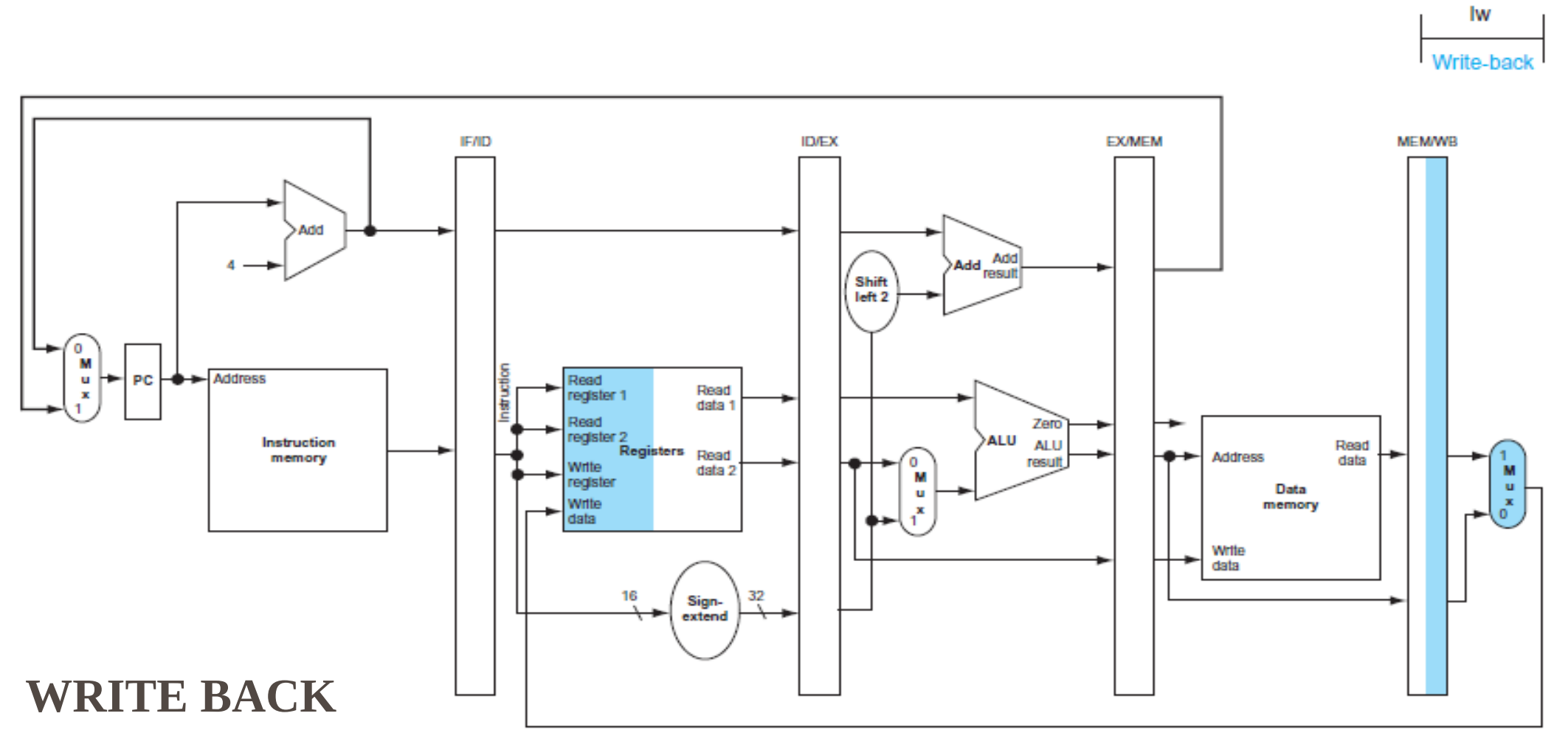
PIPELINED DATAPATH AND CONTROL

- **The five stages are the following:**

5. WRITE BACK:

- The bottom portion of the final step: reading the data from the MEM/WB pipeline register and writing it into the register file in the middle of the figure.

PIPELINED DATAPATH AND CONTROL



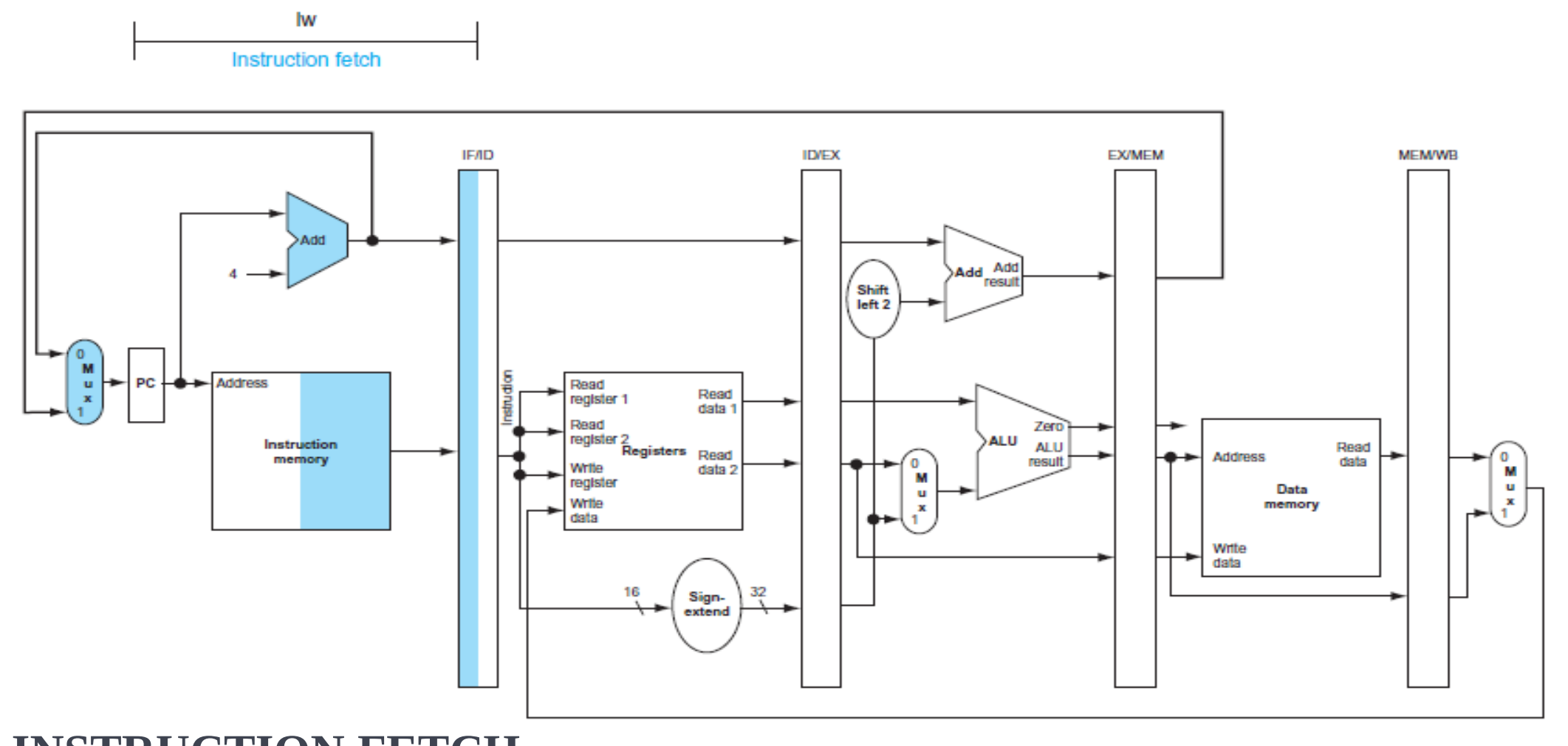
WRITE BACK

DATAPATH AND CONTROL – SW

- The datapath into five pieces, with each piece named corresponding to a stage of instruction execution:

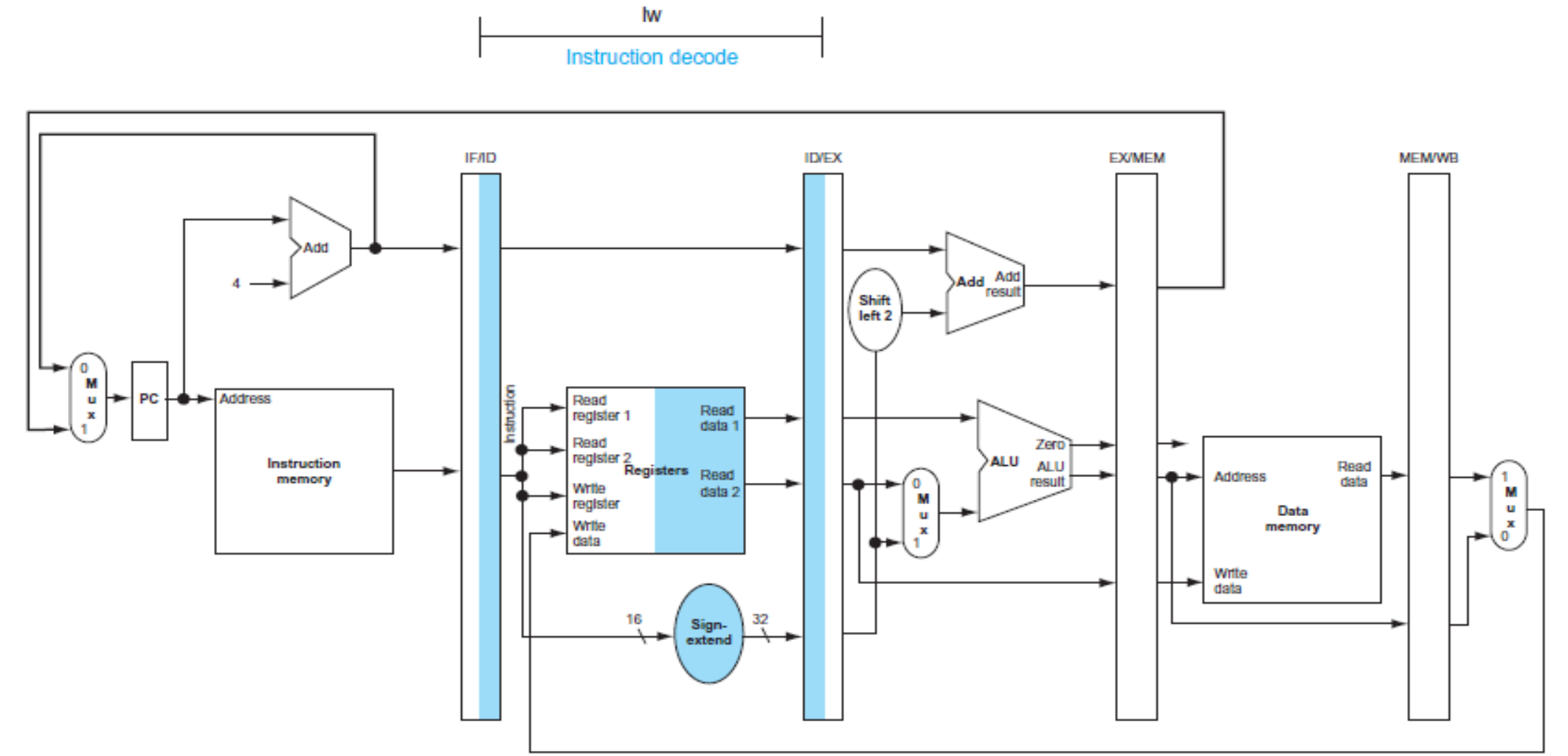
1. IF – Instruction Fetch
2. ID – Instruction Decode / Register Fetch
3. EX – Execute
4. MEM – Memory Access
5. WB – Write Back

DATAPATH AND CONTROL – SW



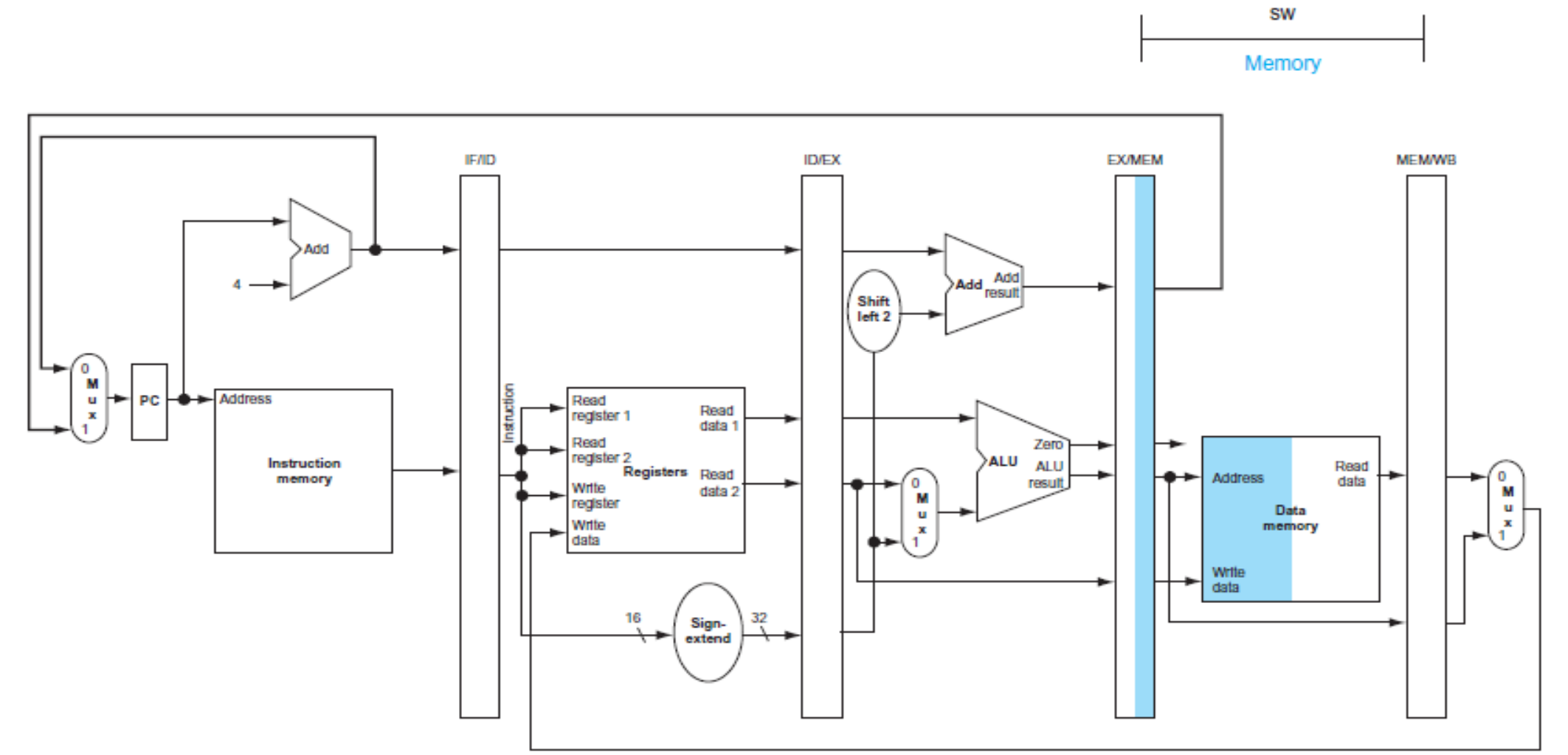
INSTRUCTION FETCH

DATAPATH AND CONTROL – SW



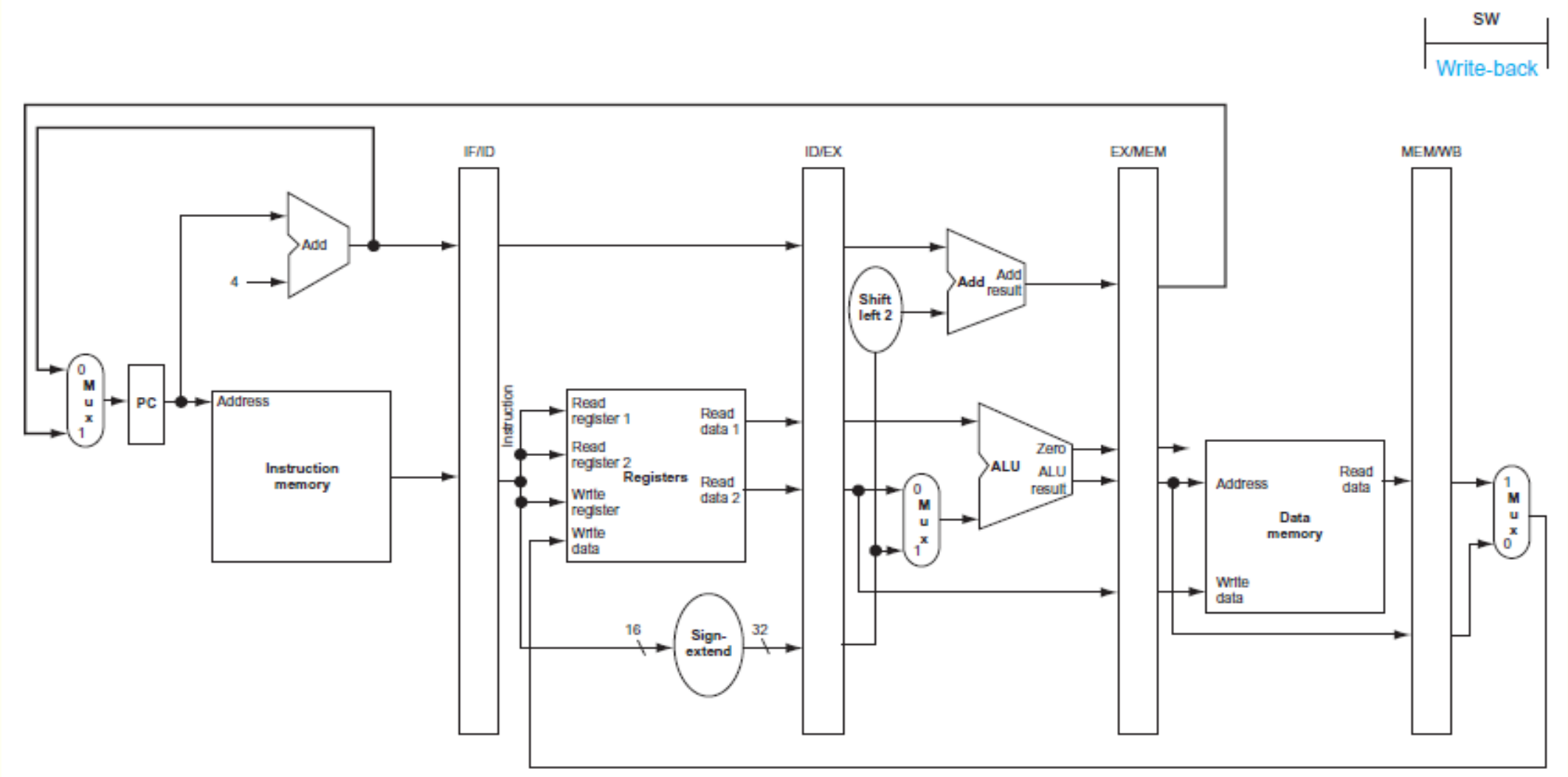
INSTRUCTION DECODE AND REGISTER FILE READ

DATAPATH AND CONTROL – SW



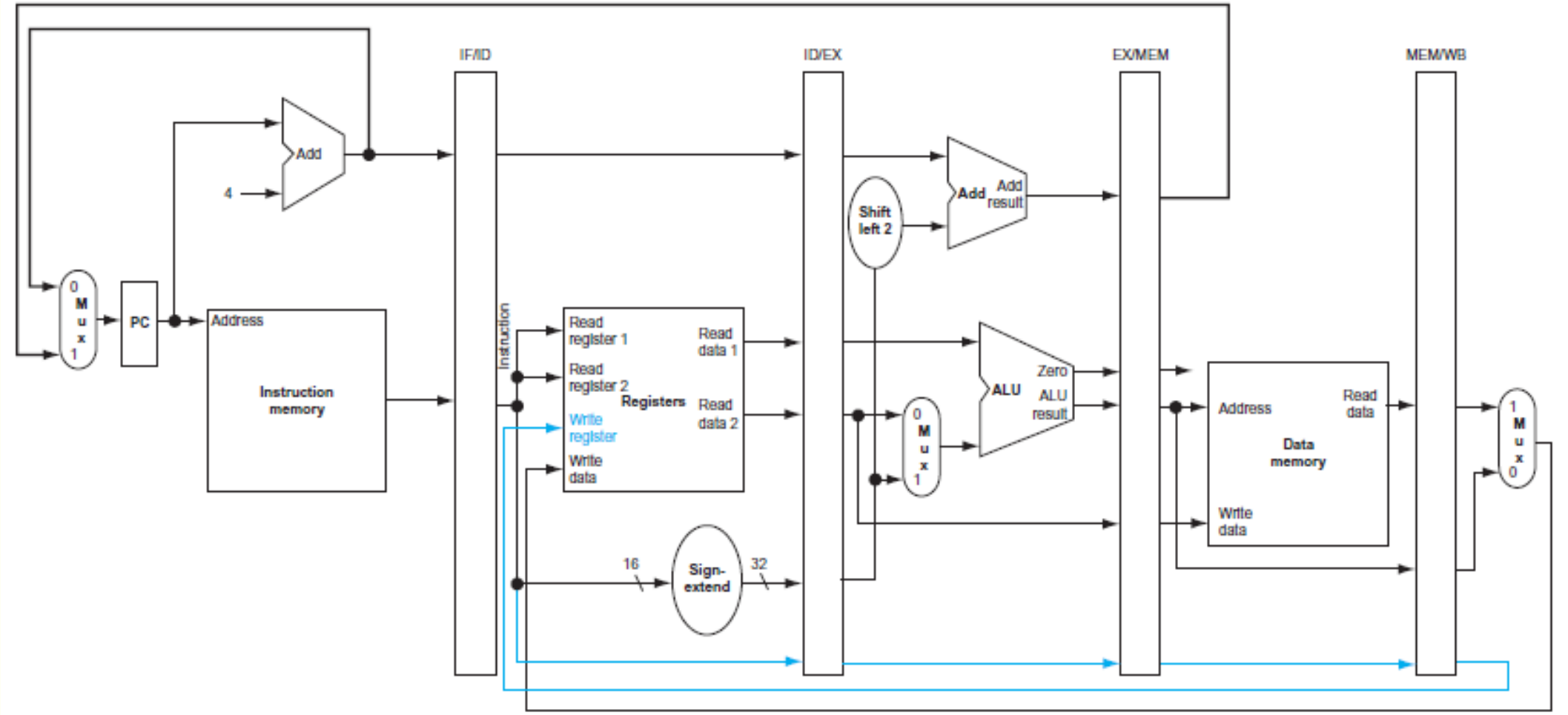
MEMORY ACCESS

DATAPATH AND CONTROL – SW



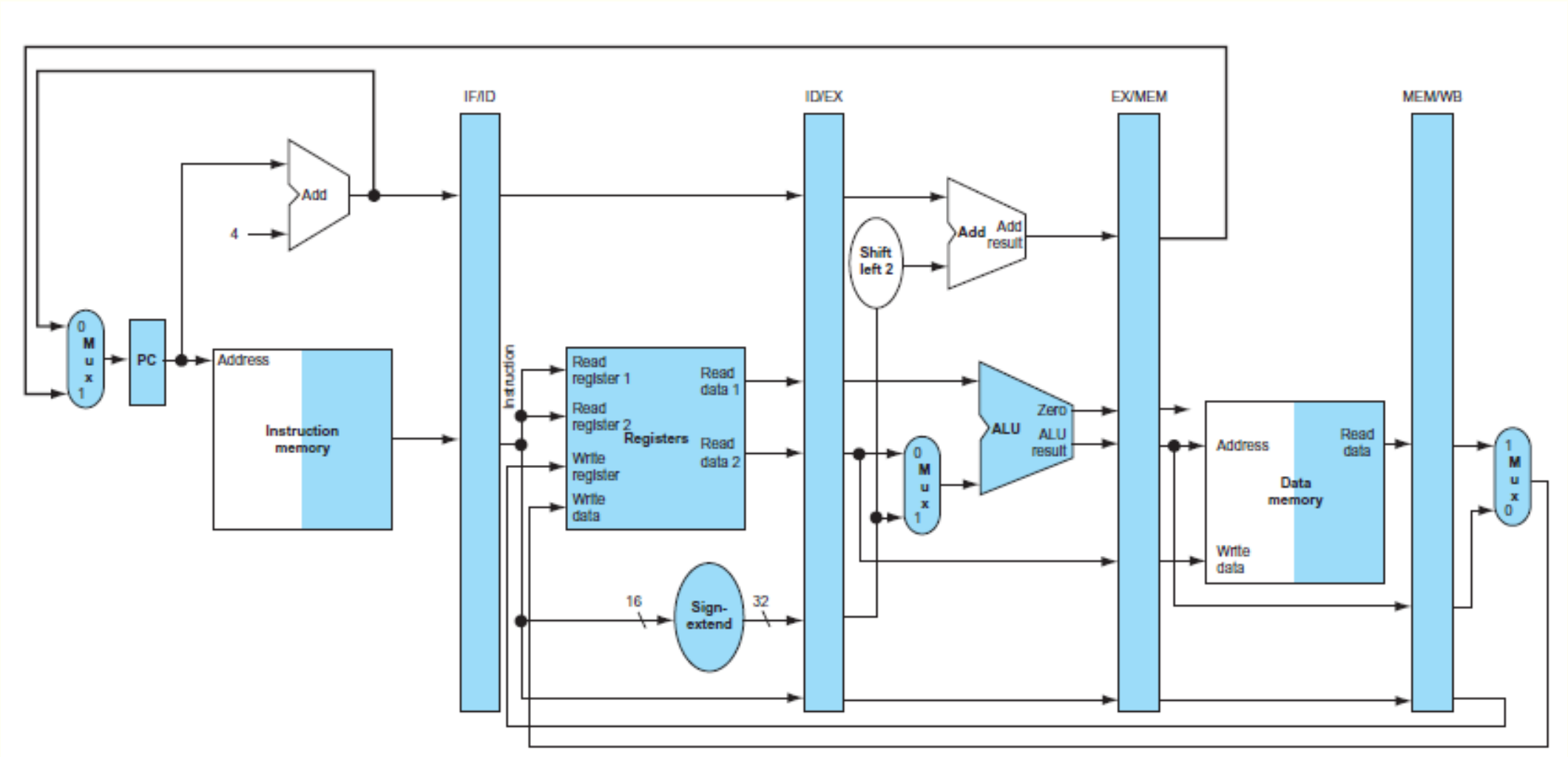
WRITE BACK

DATAPATH AND CONTROL – LOAD



CONTROL LINES

DATAPATH AND CONTROL – LOAD



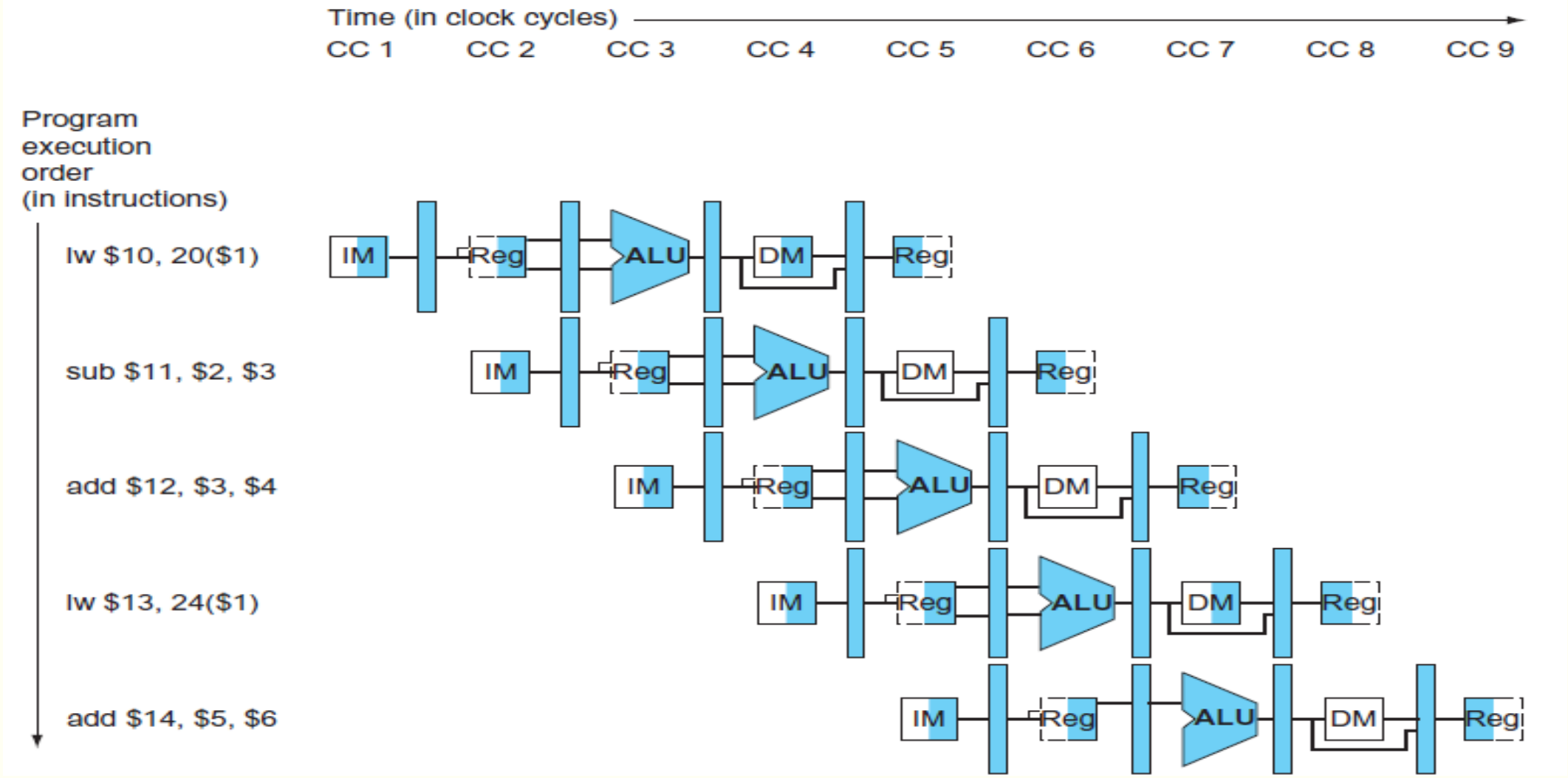
ALL FIVE STAGES

GRAPHICALLY REPRESENTING PIPELINES

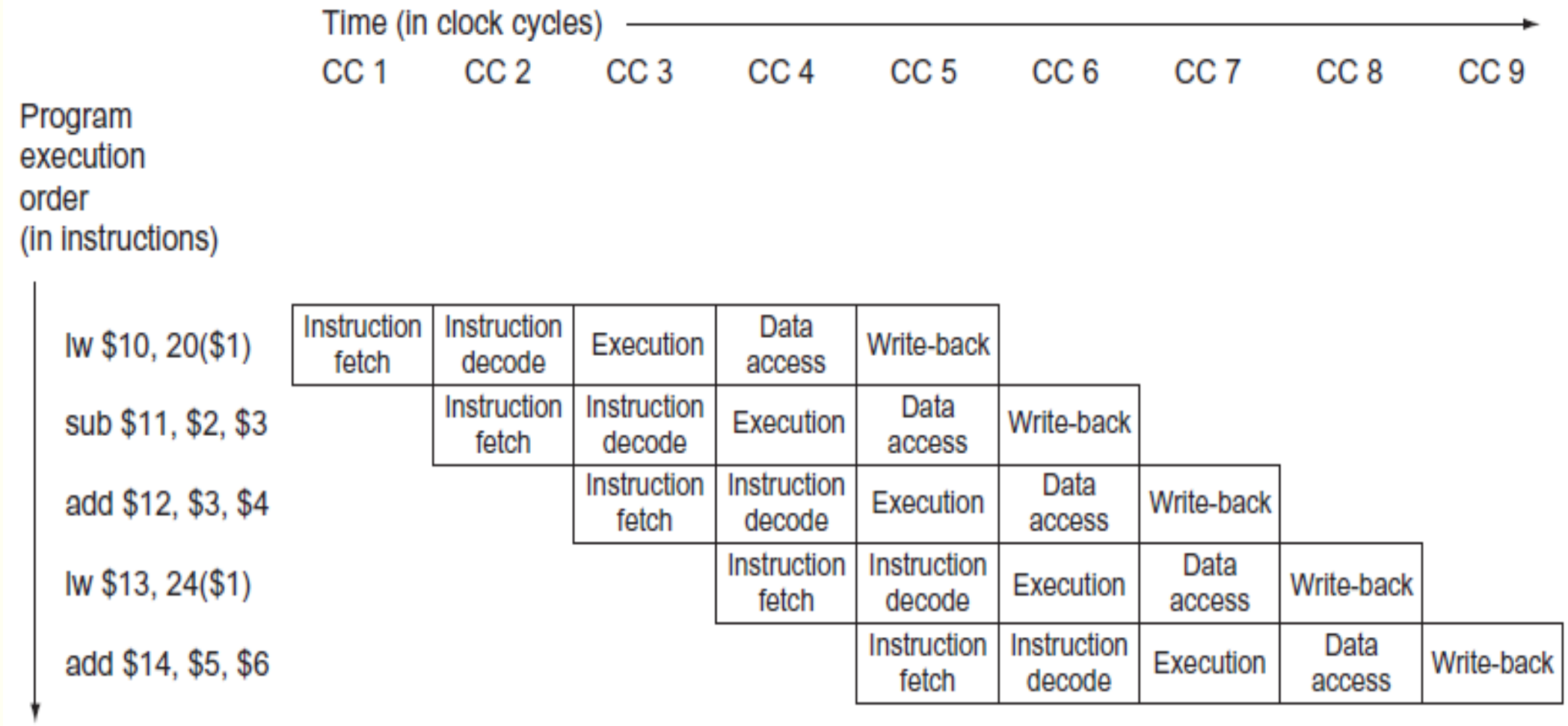
- Example:

```
lw $10, 20($1)
sub $11, $2, $3
add $12, $3, $4
lw $13, 24($1)
add $14, $5, $6
```

GRAPHICALLY REPRESENTING PIPELINES



GRAPHICALLY REPRESENTING PIPELINES



HANDLING DATA HAZARDS

Data Hazards

- Arise when an instruction depends on the results of a previous instruction in a way that is exposed by overlapping of instruction in pipeline.

Handling Data Hazards

- Stalling and Forwarding
- Systematic testing of hazard-handling logic

HANDLING DATA HAZARDS

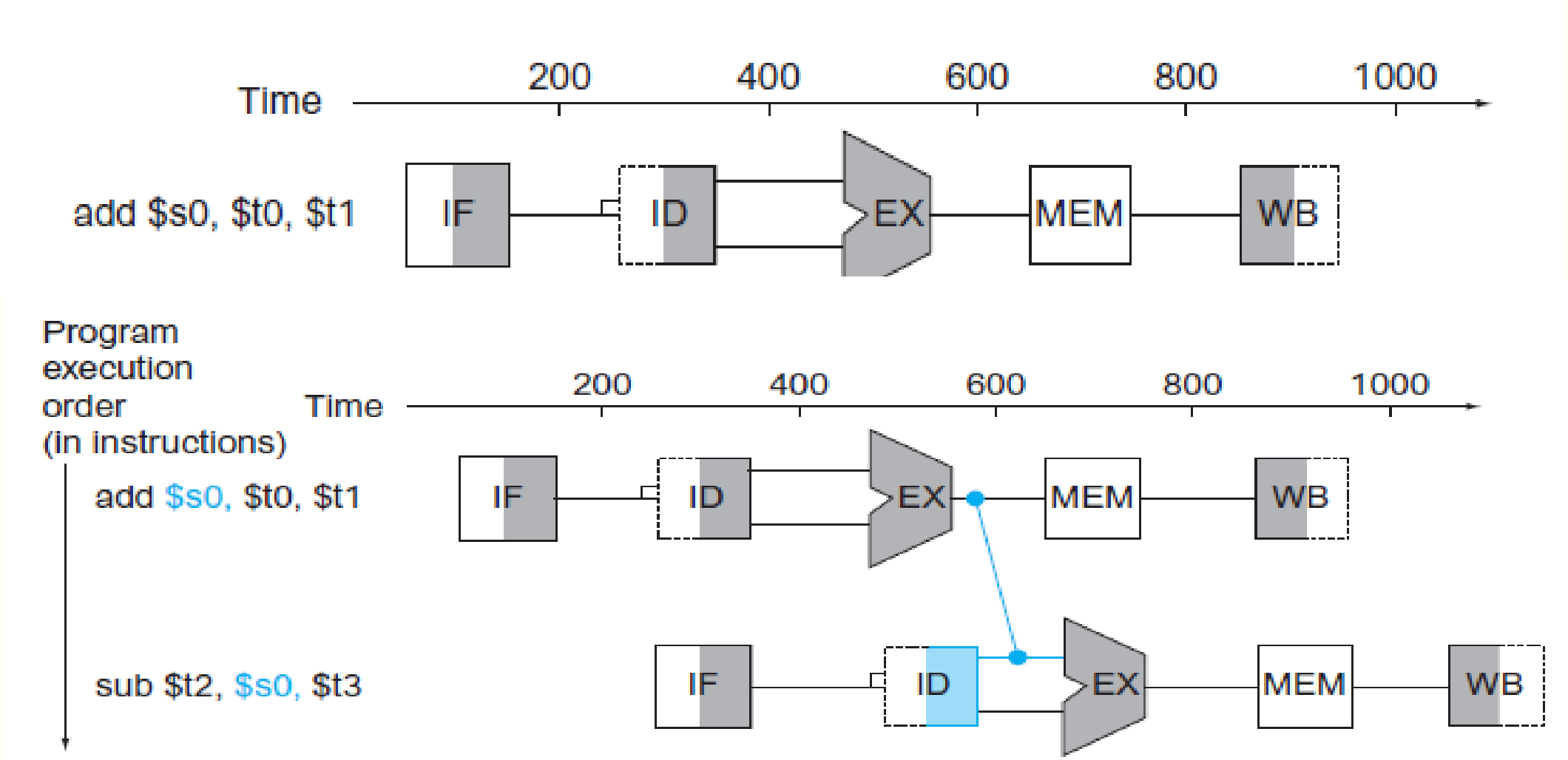
Handling Data Hazards

- Stalling (bubble)

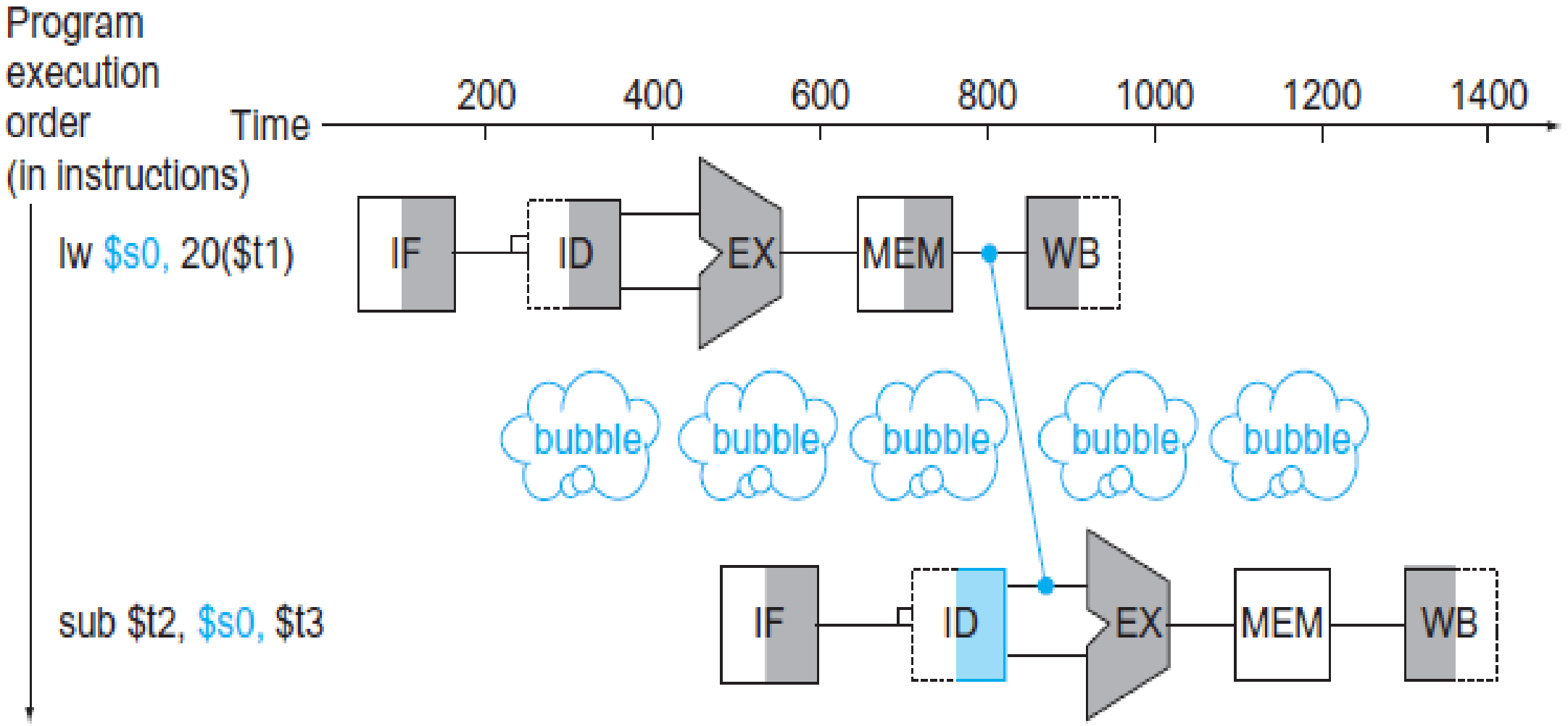
Example:

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

HANDLING DATA HAZARDS



HANDLING DATA HAZARDS



HANDLING DATA HAZARDS

Handling Data Hazards

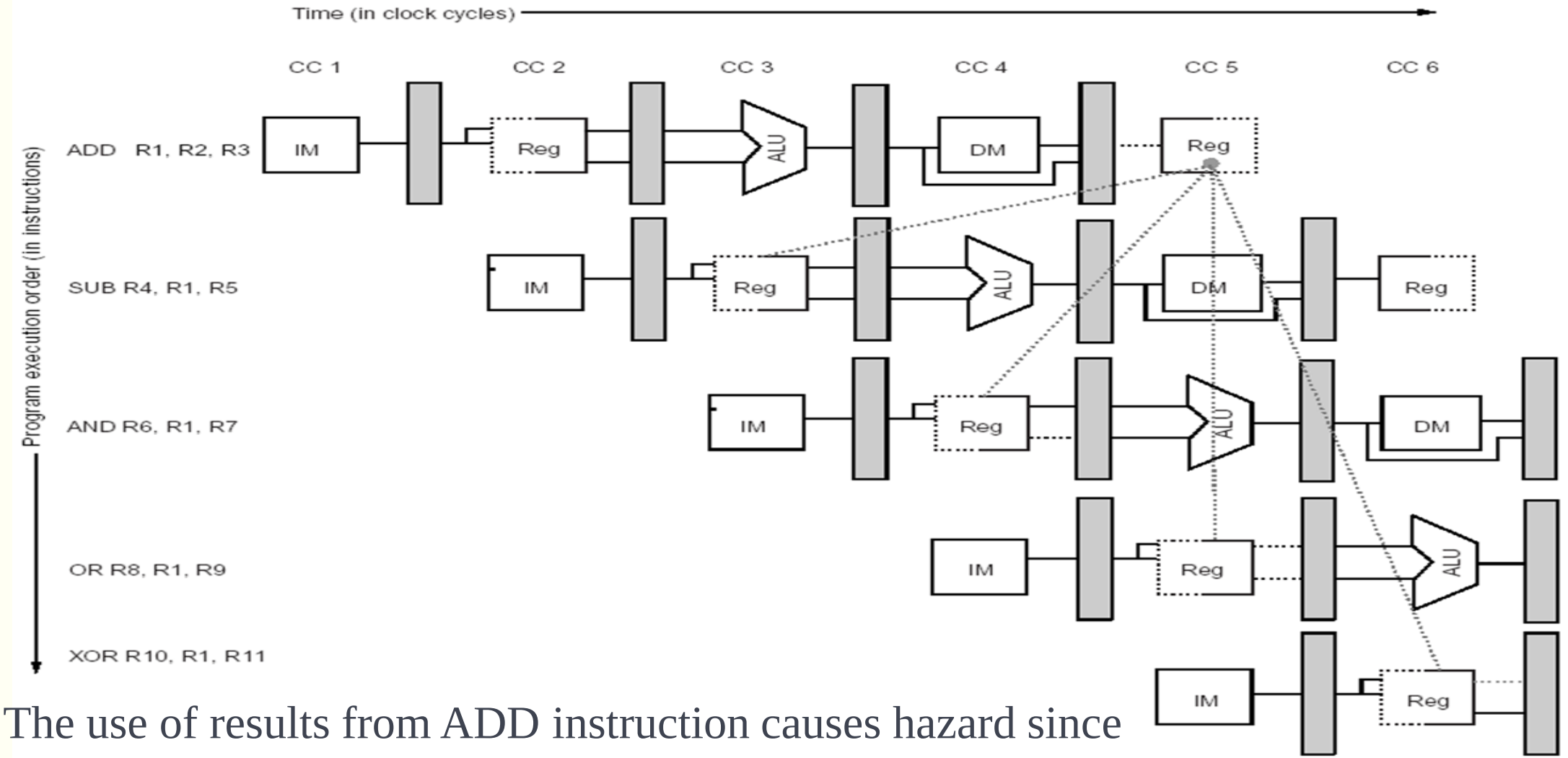
- Forwarding

Example:

```
ADD R1, R2, R3
SUB R4, R1, R5
AND R6, R1, R7
OR R8, R1, R9
XOR R10, R1, R11
```

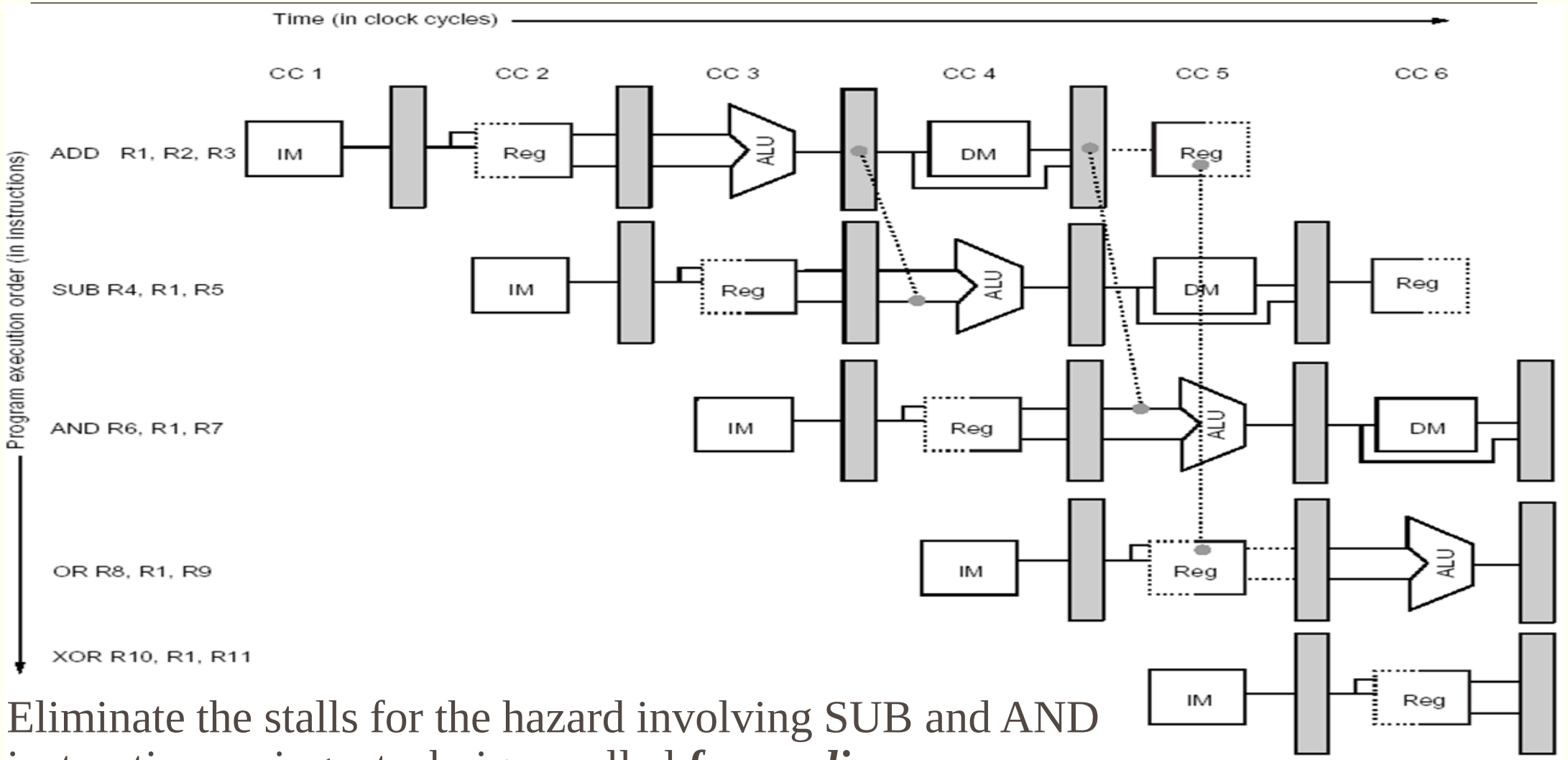
- The use of results from ADD instruction causes hazard since the register is not written until after those instructions read it.

HANDLING DATA HAZARDS



The use of results from ADD instruction causes hazard since the register is not written until after those instructions read it.

HANDLING DATA HAZARDS



Eliminate the stalls for the hazard involving SUB and AND instructions using a technique called ***forwarding***

HANDLING DATA HAZARDS

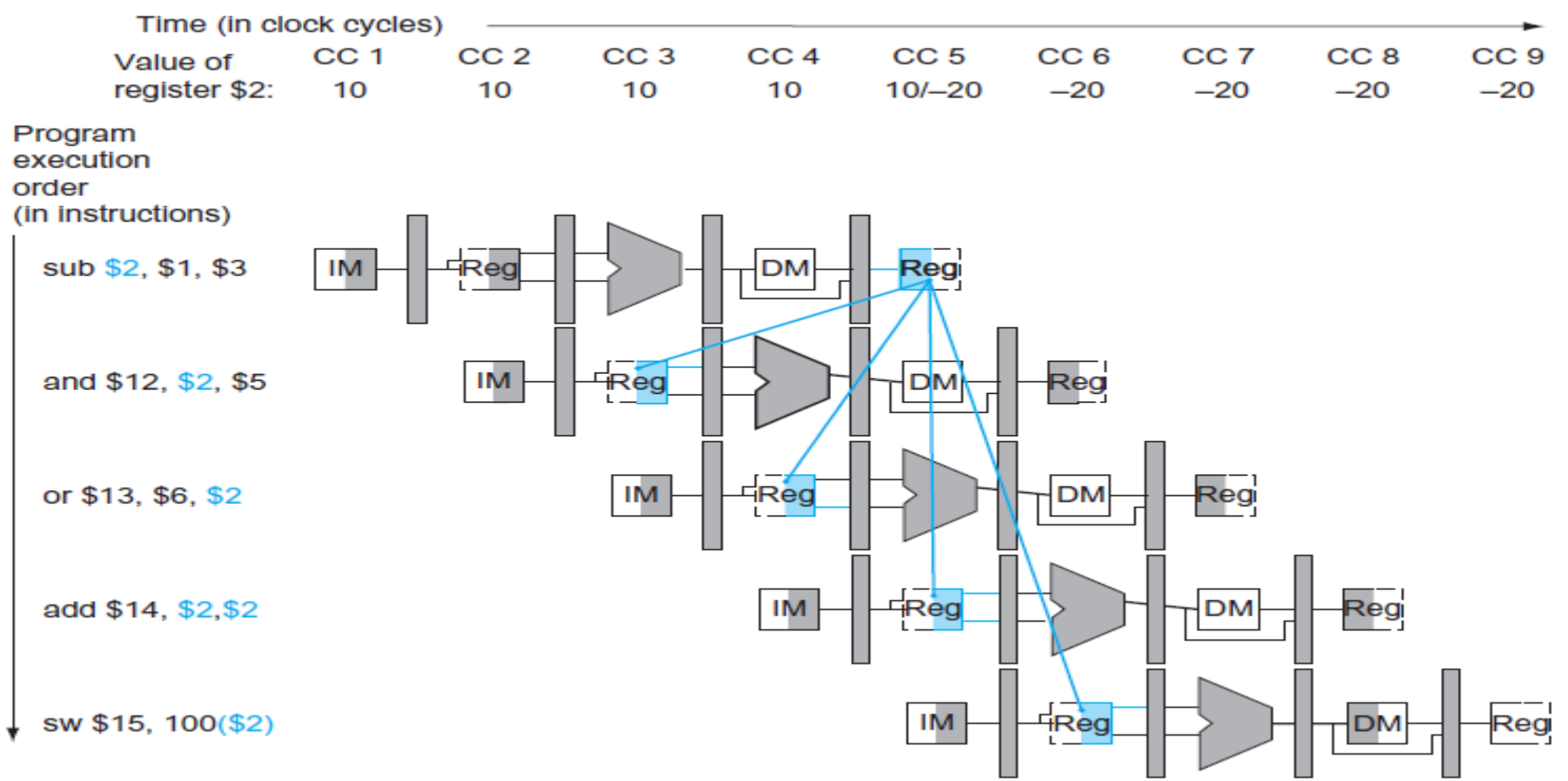
Handling Data Hazards

- Forwarding

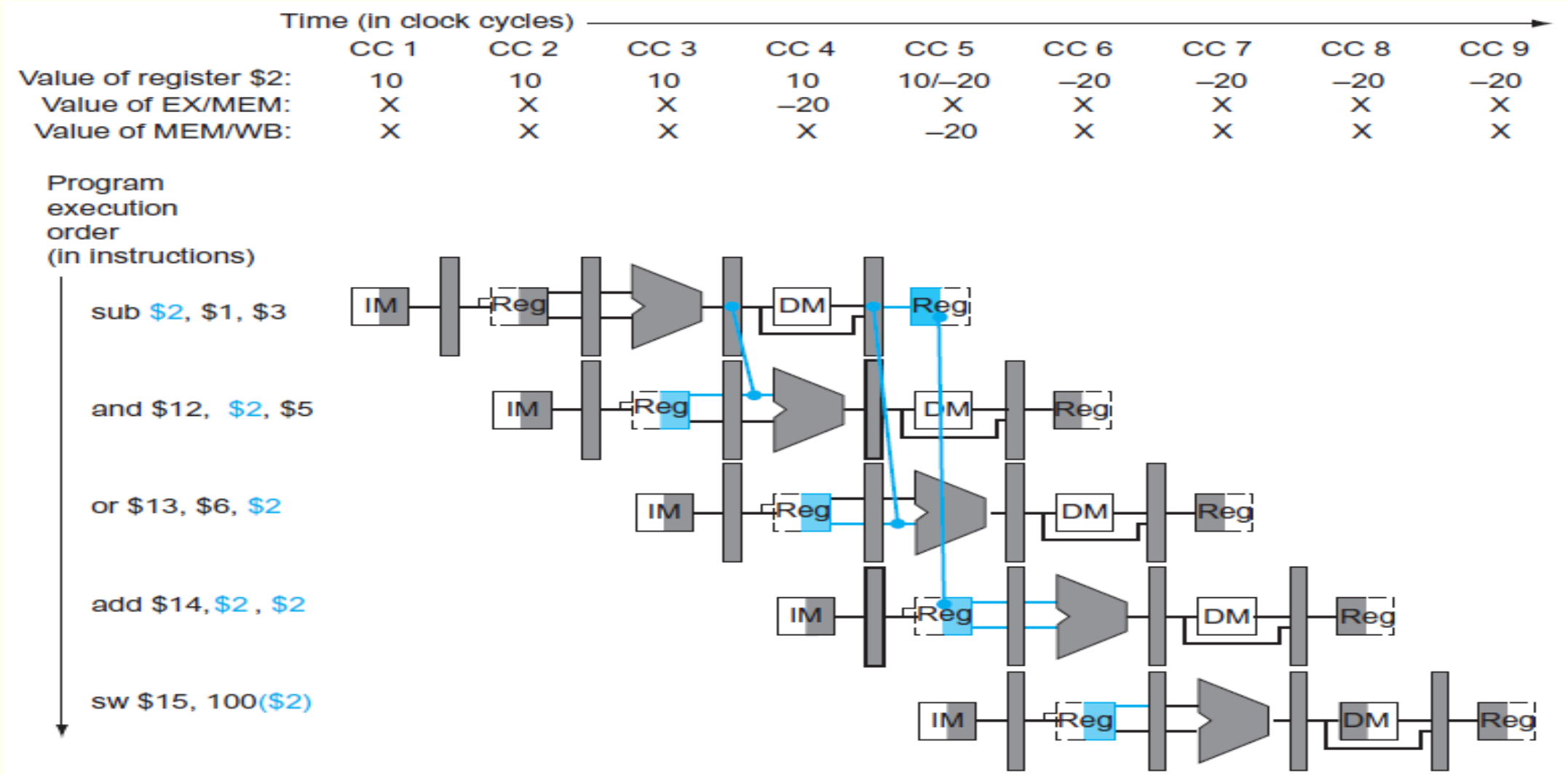
Example:

```
sub $2, $1,$3          # Register $2 written by sub
and $12,$2,$5          # 1st operand($2) depends on sub
or $13,$6,$2           # 2nd operand($2) depends on sub
add $14,$2,$2          # 1st($2) & 2nd($2) depend on sub
sw $15,100($2)         # Base ($2) depends on sub
```

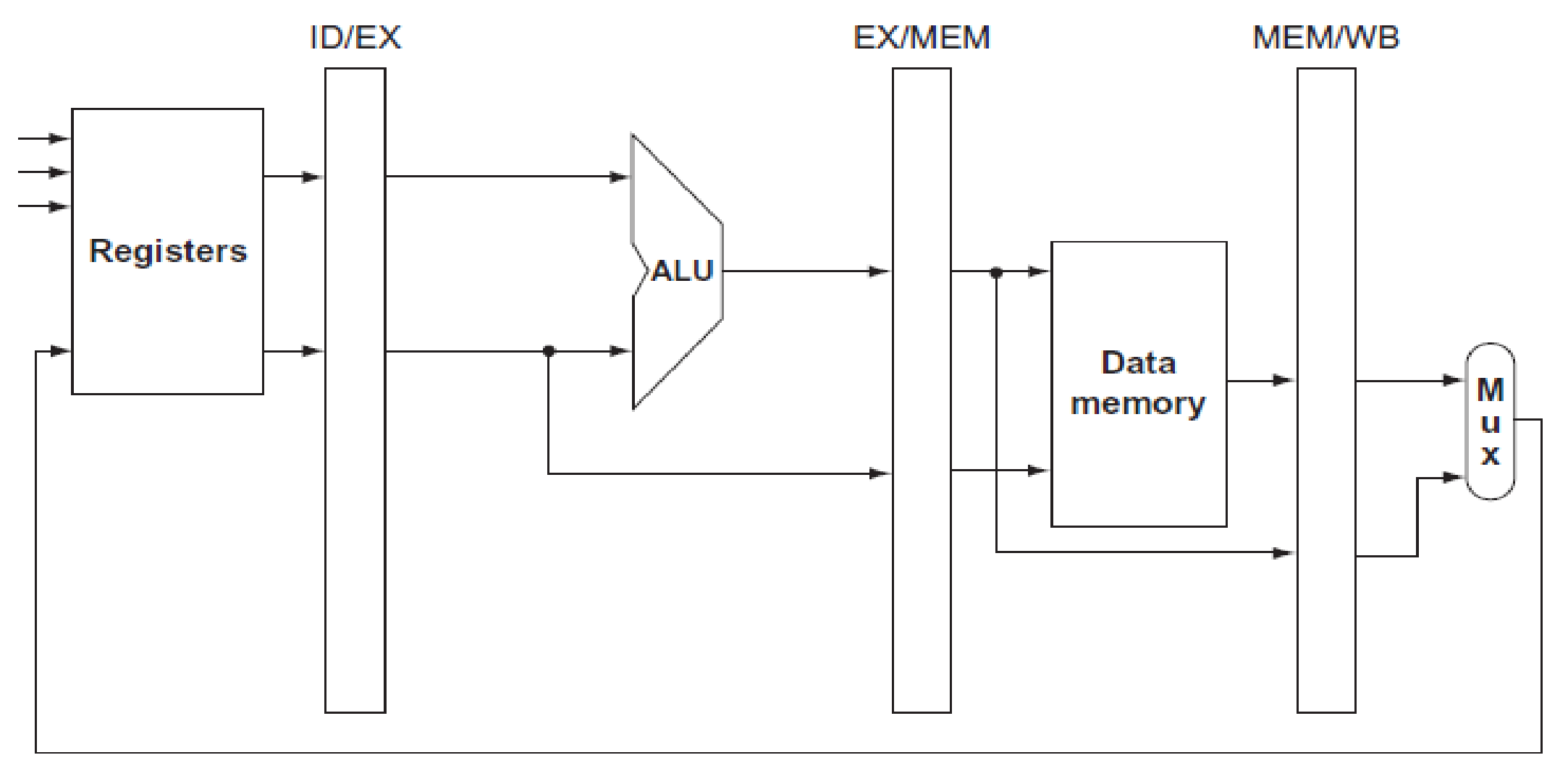
HANDLING DATA HAZARDS



HANDLING DATA HAZARDS

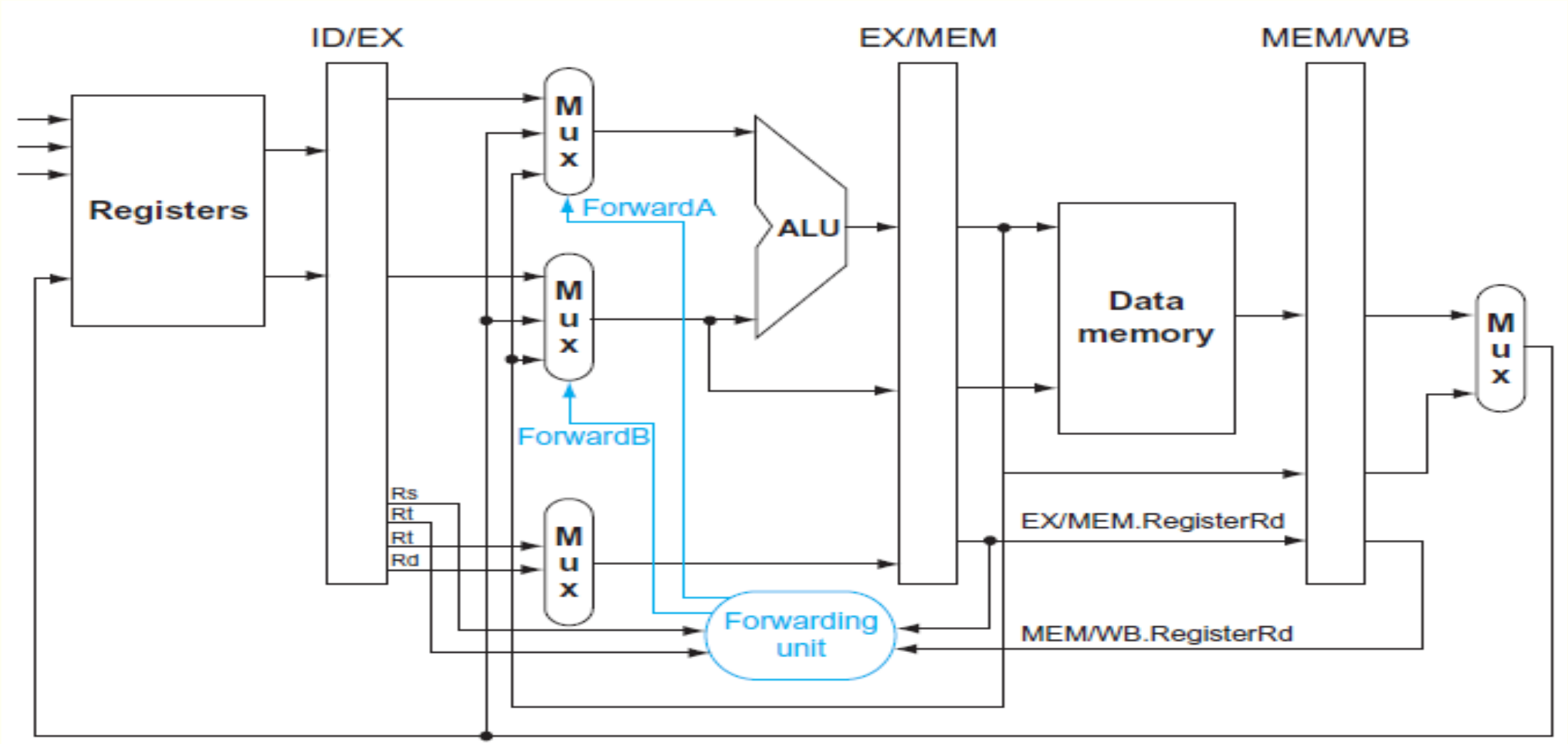


HANDLING DATA HAZARDS



No forwarding

HANDLING DATA HAZARDS



With forwarding

HANDLING DATA HAZARDS

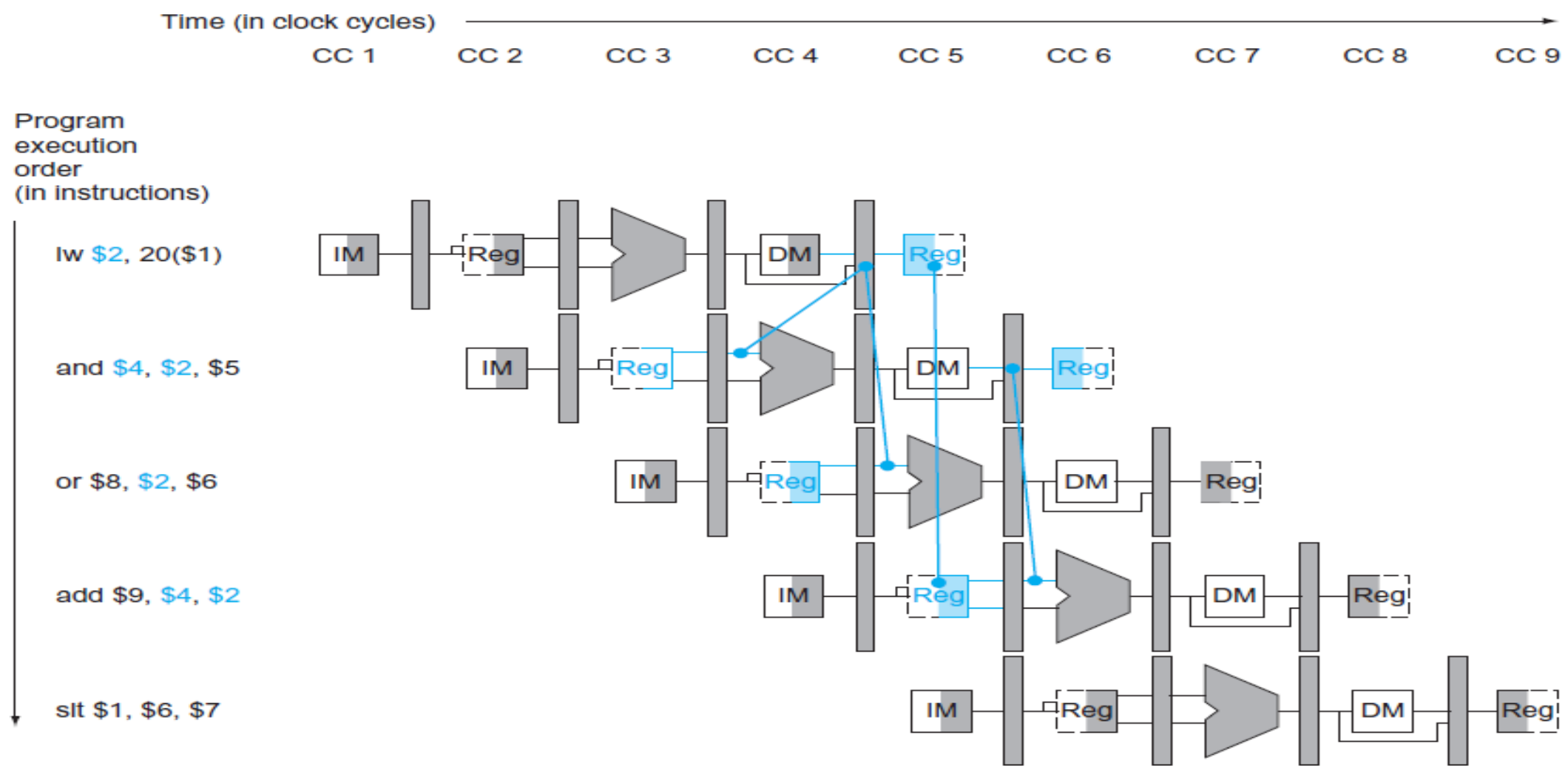
Handling Data Hazards

- Stalling

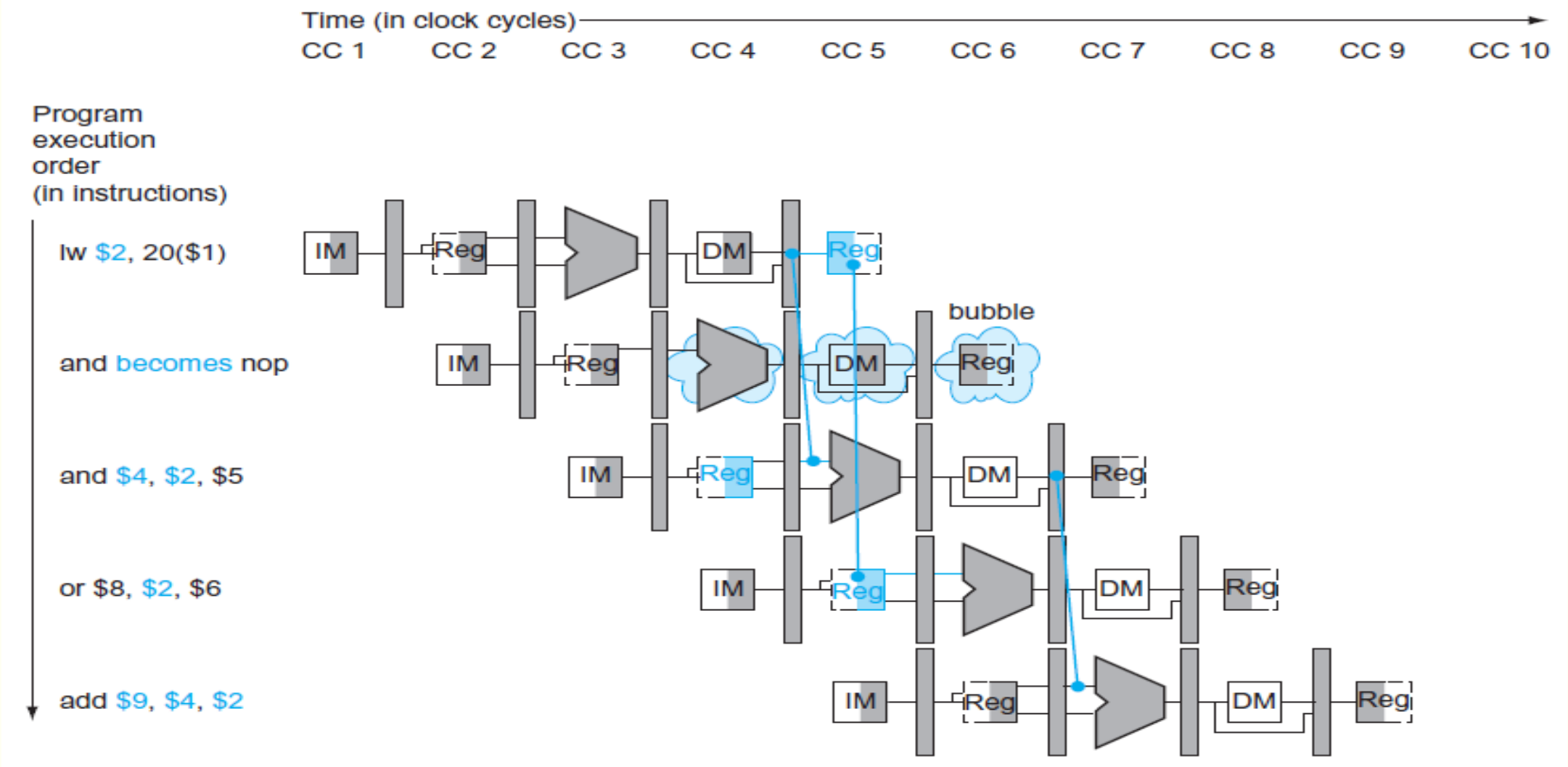
Example:

```
lw $s0, 20($1)
and $4, $2, $5
or $8, $2, $6
add $9, $4, $2
slt $1, $6, $7
```

HANDLING DATA HAZARDS



HANDLING DATA HAZARDS



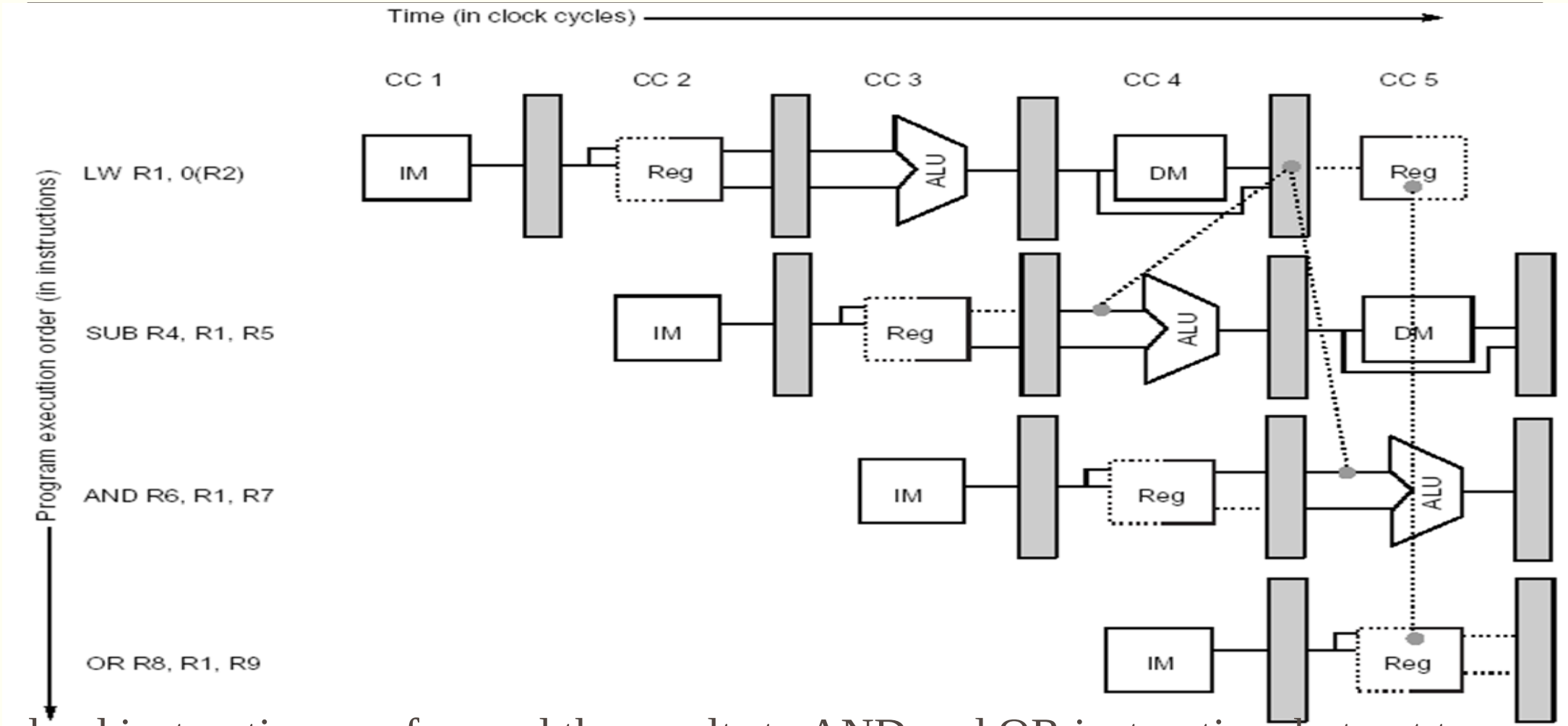
HANDLING DATA HAZARDS

- Consider two instructions i and j, with i occurring before j. Possible data hazards:
  - RAW (Read After Write)
    - j tries to read a source before i writes to it , so j incorrectly gets the old value;
    - most common type of hazard, that is what we tried to explain so far.
  - WAW (Write After Write)
    - j tries to write an operand before is written by i. The write ends up being performed in wrong order, having i overwrite the operand written by j, the destination containing the operand written by i rather than the one written by j
    - Present in pipelines that write in more than one pipe stage
  - WAR (Write After Read)
    - j tries to write a destination before it is read by i, so the instruction i incorrectly gets the new value
    - This doesn’t happen in our example, since all reads are early and writes late

HANDLING DATA HAZARDS

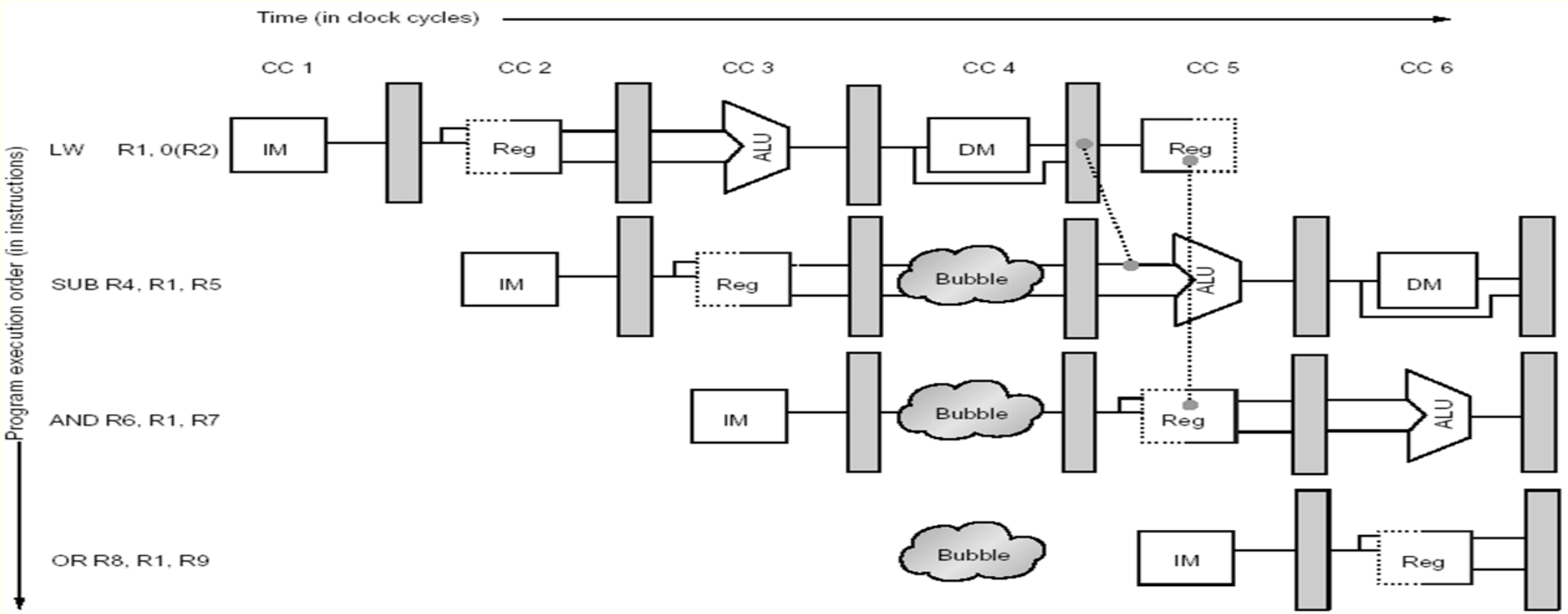
- Consider the following sequence:
  - LW R1, 0(R2)
  - SUB R4, R1, R5
  - AND R6, R1, R7
  - OR R8, R1, R9
- The problem with this sequence is that the Load operation will not have data until the end of MEM stage.

HANDLING DATA HAZARDS



The load instruction can forward the results to AND and OR instruction, but not to the SUB instruction since that would mean forwarding results in “negative” time

HANDLING DATA HAZARDS



The load interlock causes a stall to be inserted at clock cycle 4, delaying the SUB instruction and those that follow by one cycle.

This delay allows the value to be successfully forwarded onto the next clock cycle

HANDLING DATA HAZARDS

LW R1, 0(R2)	IF	ID	EX	MEM	WB					
SUB R4, R1, R5		IF	ID	EX	MEM	WB				
AND R6, R1, R7			IF	ID	EX	MEM	WB			
OR R8, R1, R9				IF	ID	EX	MEM	WB		

- Before Stall Instruction

LW R1, 0(R2)	IF	ID	EX	MEM	WB					
SUB R4, R1, R5		IF	ID	stall	EX	MEM	WB			
AND R6, R1, R7			IF	stall	ID	EX	MEM	WB		
OR R8, R1, R9				stall	IF	ID	EX	MEM	WB	

- After Stall Instruction

HANDLING CONTROL HAZARDS

Control Hazards

- Arise from the pipelining of branches and other instructions that change the PC (Program Counter).

Handling Control Hazards

- Stalling and Forwarding

HANDLING CONTROL HAZARDS

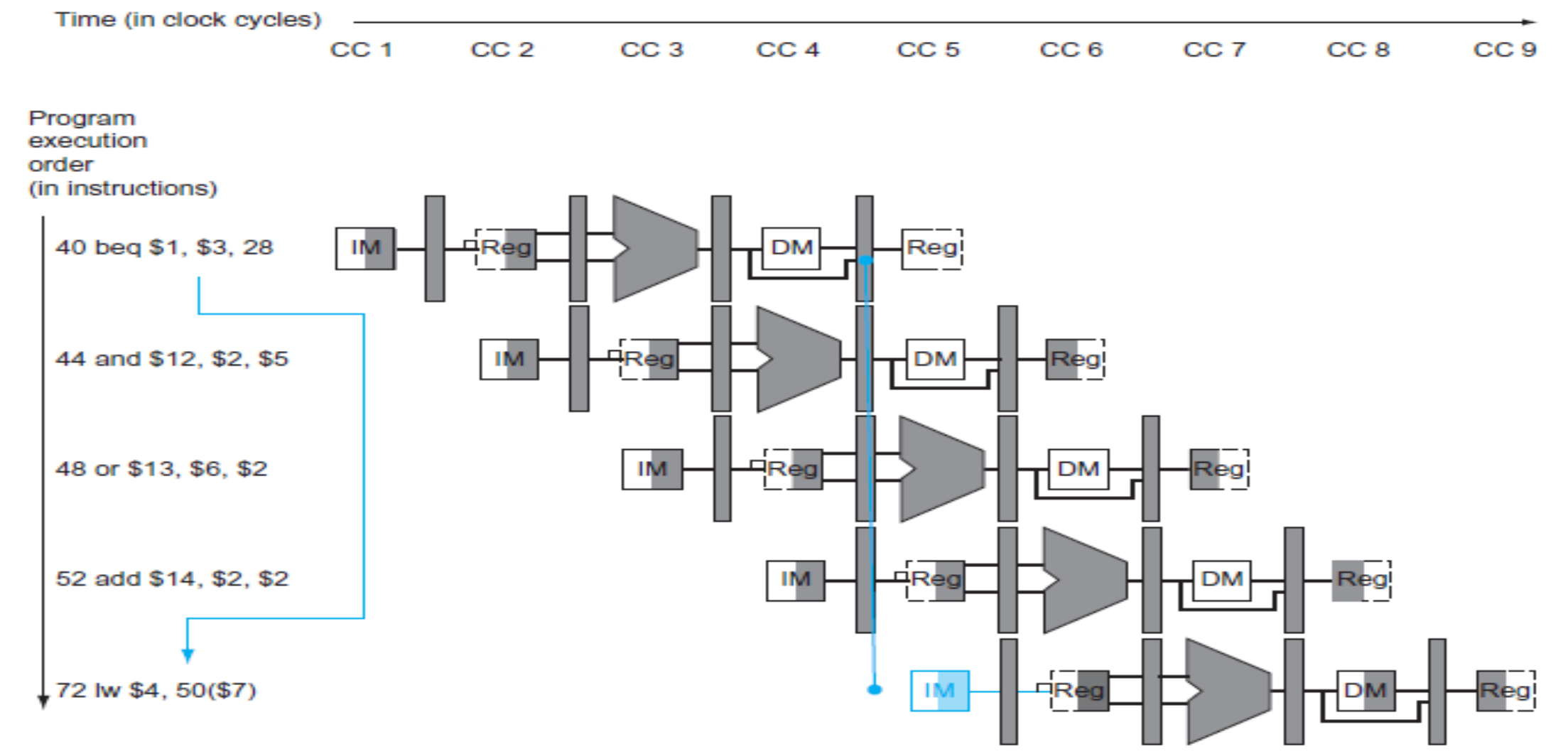
- Can cause a greater performance loss than the data hazards
- When a branch is executed it may or it may not change the PC (to other value than its value + 4)
  - If a branch is changing the PC to its target address, than it is a ***taken*** branch
  - If a branch doesn’t change the PC to its target address, than it is a ***not taken*** branch
- If instruction i is a taken branch, than the value of PC will not change until the end MEM stage of the instruction execution in the pipeline
  - A simple method to deal with branches is to stall the pipe as soon as we detect a branch until we know the result of the branch

HANDLING CONTROL HAZARDS

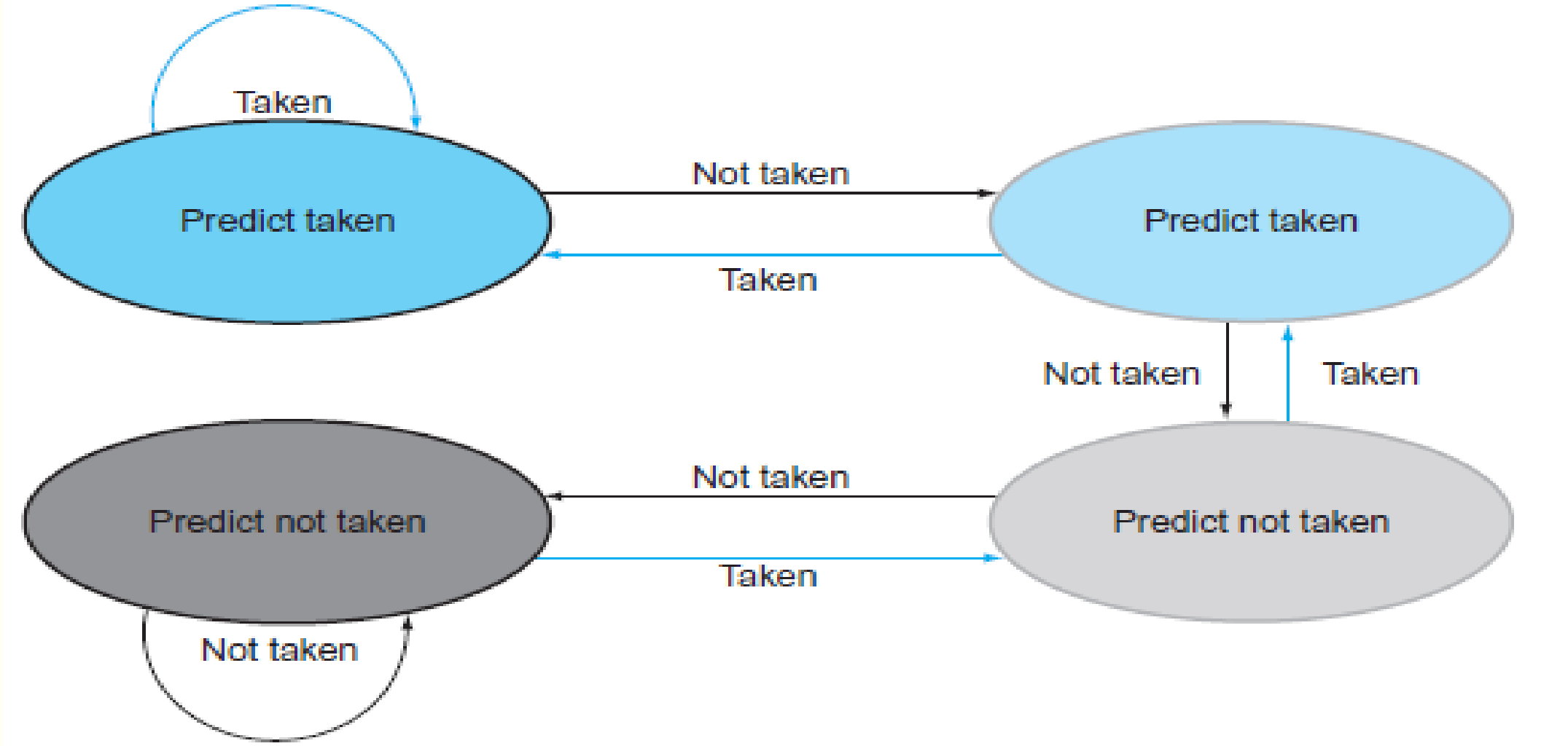
Example:

```
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)
```

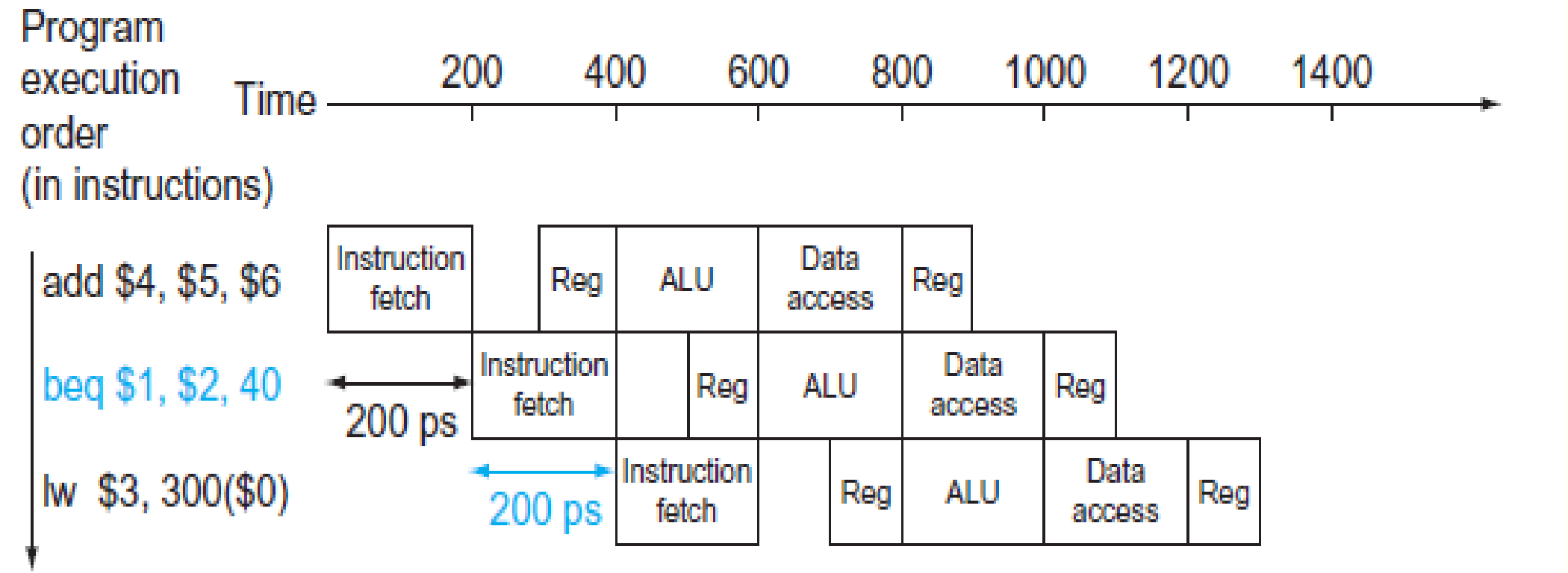
HANDLING CONTROL HAZARDS



HANDLING CONTROL HAZARDS

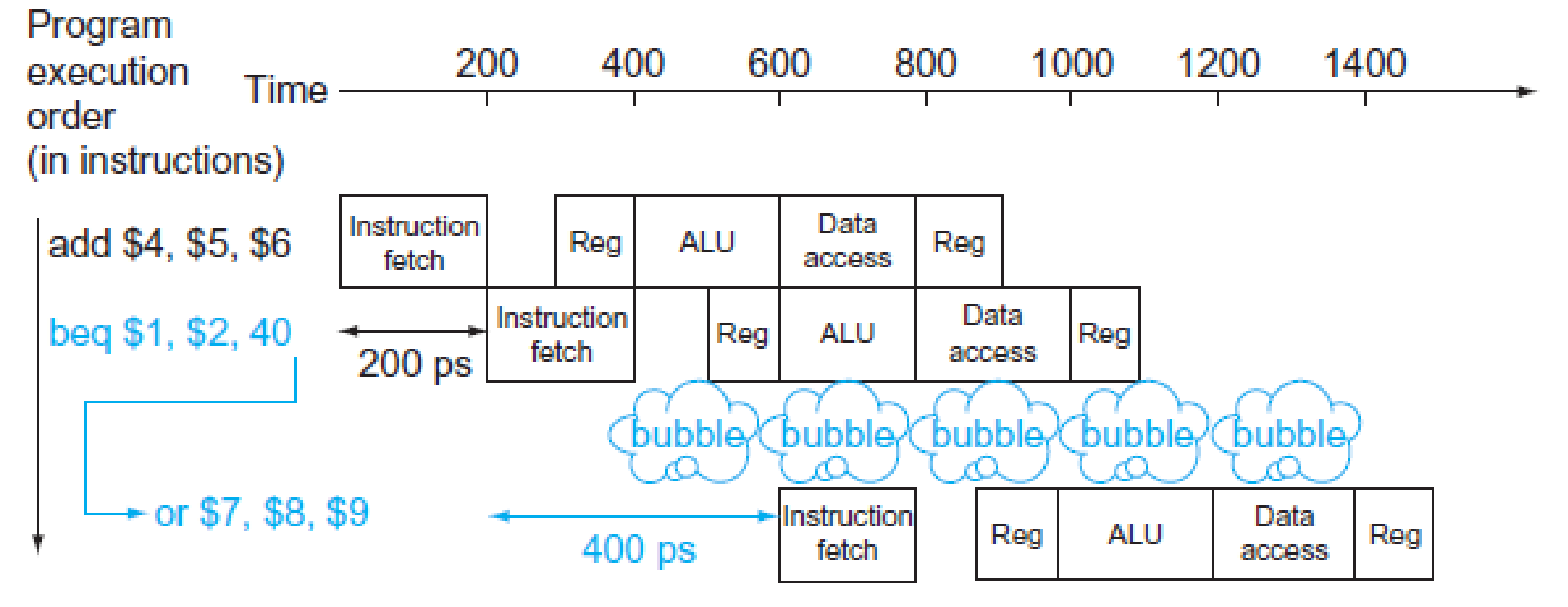


HANDLING CONTROL HAZARDS



The pipeline when the branch is not taken

HANDLING CONTROL HAZARDS



The pipeline when the branch is taken

HANDLING STRUCTURAL HAZARDS

Structural Hazards

- Arise from resource conflicts when the hardware can’t support all possible combinations of overlapping instructions

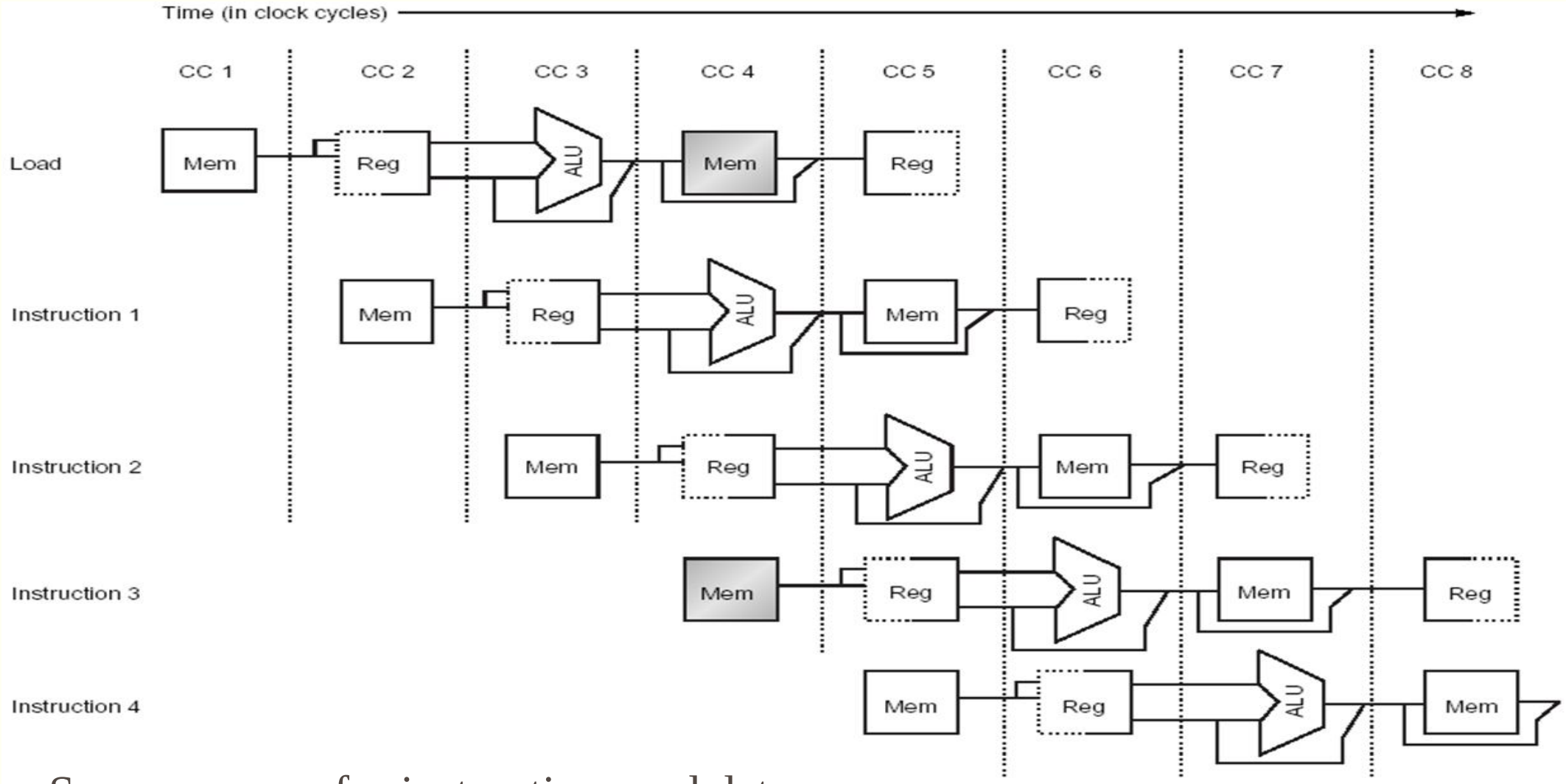
Handling Structural Hazards

- Stall

HANDLING STRUCTURAL HAZARDS

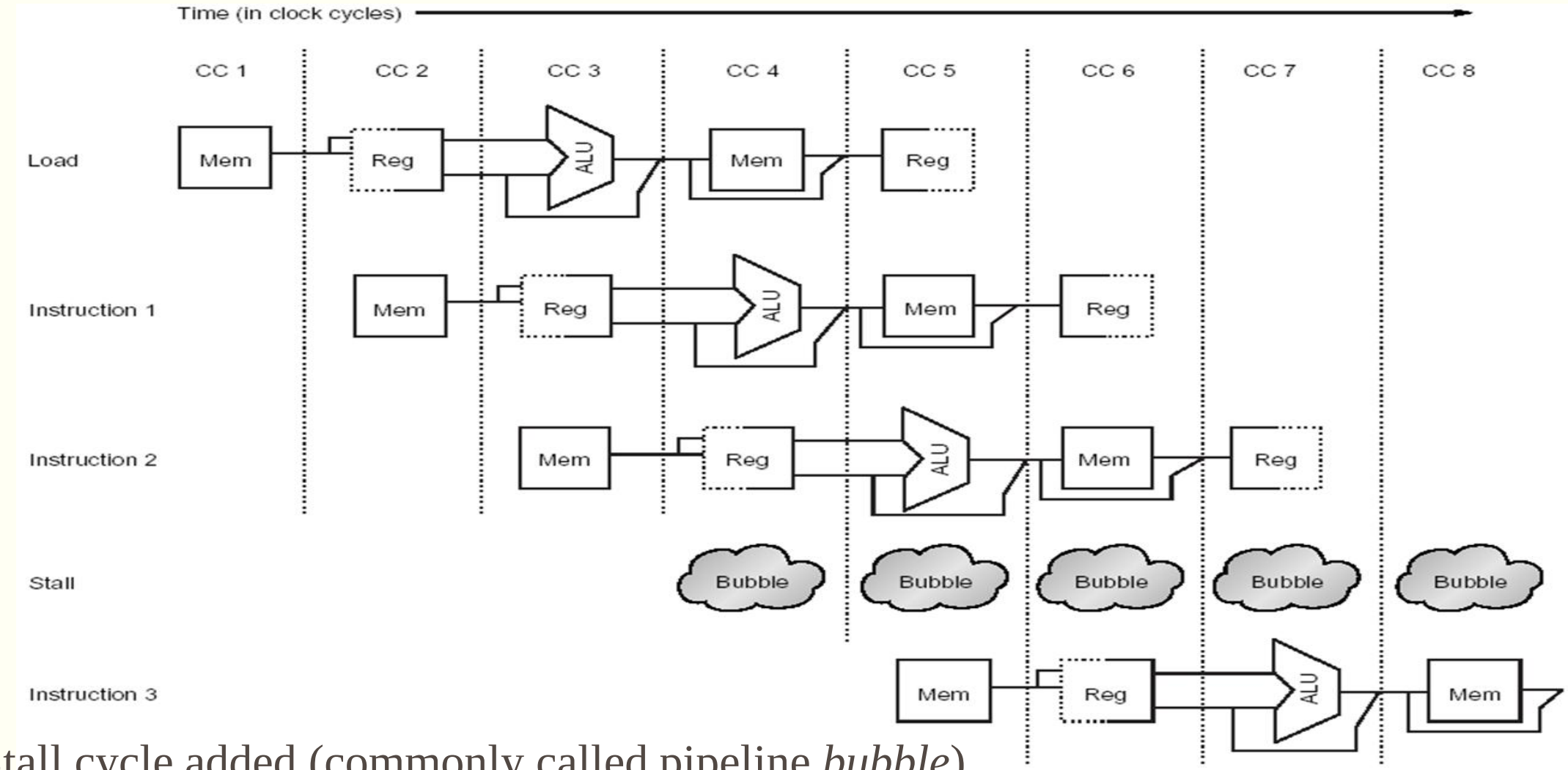
- If certain combination of instructions can't be accommodated because of resource conflicts, the machine is said to have a *structural hazard*
- It can be generated by:
  - Some functional unit is not fully pipelined
  - Some resources has not been duplicated enough to allow all the combinations in the pipeline to execute
  - For example: a machine may have only one register file write port, but under certain conditions, the pipeline might want to perform two writes in one clock cycle – this will generate structural hazard
    - When a sequence of instructions encounter this hazard, the pipeline will stall one of the instructions until the required unit is available
    - Such stalls will increase the Clock cycle Per Instruction from its ideal 1 for pipelined machines

HANDLING STRUCTURAL HAZARDS



Same memory for instructions and data

HANDLING STRUCTURAL HAZARDS



Stall cycle added (commonly called pipeline *bubble*)

HANDLING STRUCTURAL HAZARDS

Instruction Number	Clock number									
	1	2	3	4	5	6	7	8	9	10
load	IF	ID	EX	MEM	WB					
Instruction i+1		IF	ID	EX	MEM	WB				
Instruction i+2			IF	ID	EX	MEM	WB			
Instruction i+3				stall	IF	ID	EX	MEM	WB	
Instruction i+4						IF	ID	EX	MEM	WB
Instruction i+5							IF	ID	EX	MEM

Another way to represent the stall – no instruction is initiated in clock cycle 4

EXCEPTIONS

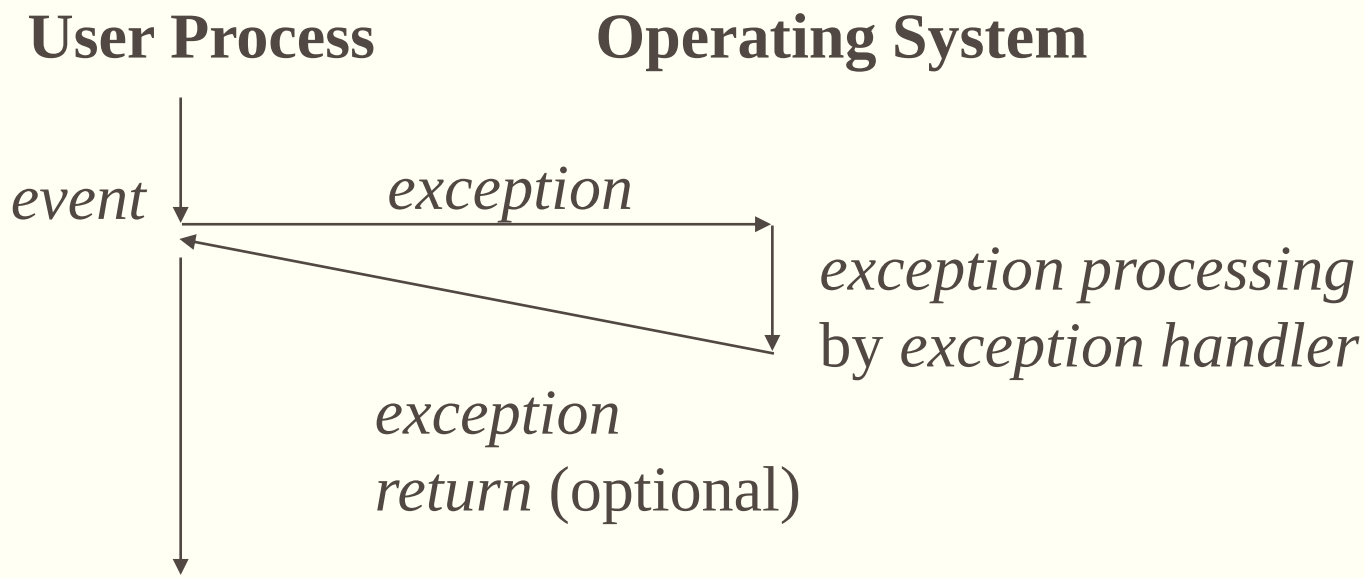
- One of the hardest parts of control is implementing exceptions and interrupts events other than branches or jumps that change the normal flow of instruction execution.
- They were initially created to handle unexpected events from within the processor, like arithmetic overflow.
- Exception also called interrupt. An unscheduled event that disrupts program execution; used to detect overflow.

Interrupt:

- An exception that comes from outside of the processor. (Some architectures use the term interrupt for all exceptions.)

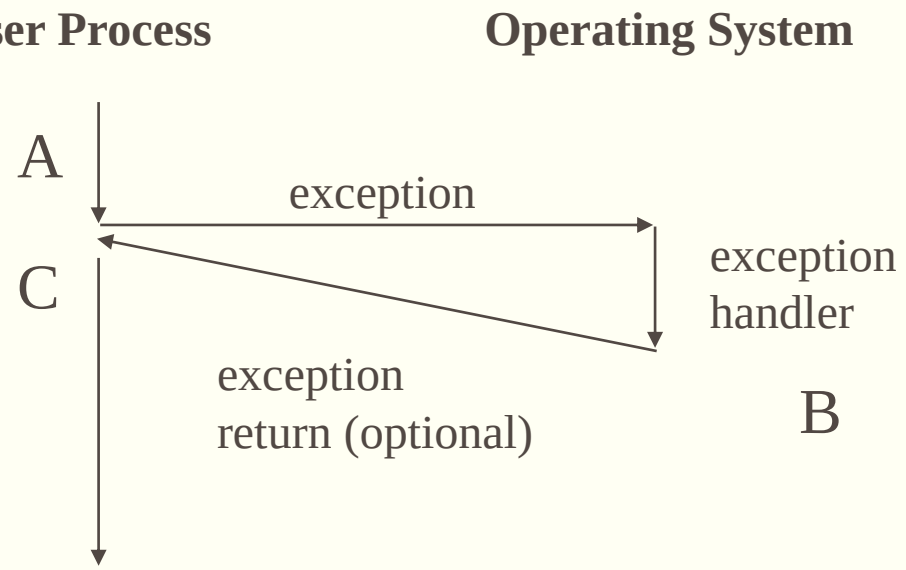
EXCEPTIONS

- An exception is a transfer of control to the OS in response to some event (i.e. change in processor state)



ISSUES WITH EXCEPTIONS

- A1: What kinds of events can cause an exception?
- A2: When does the exception occur?
- B1: How does the handler determine the location and cause of the exception?
- B2: Are exceptions allowed within exception handlers?
- C1: Can the user process restart?
- C2: If so, where?



EXCEPTIONS

Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or Interrupt

INTERNAL (CPU) EXCEPTIONS

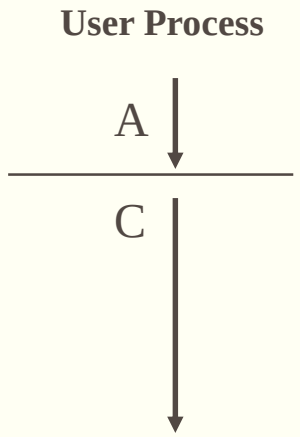
- Internal exceptions occur as a result of events generated by executing instructions.
- Execution of a CALL_PAL instruction.
  - allows a program to transfer control to the OS
- Errors during instruction execution
  - arithmetic overflow, address error, parity error, undefined instruction
- Events that require OS intervention
  - virtual memory page fault

EXTERNAL (I/O) EXCEPTIONS

- External exceptions occur as a result of events generated by devices external to the processor.
- I/O interrupts
  - hitting ^C at the keyboard
  - arrival of a packet
  - arrival of a disk sector
- Hard reset interrupt
  - hitting the reset button
- Soft reset interrupt
  - hitting ctl-alt-delete on a PC

EXCEPTIONS HANDLING (Hardware Tasks)

- Recognize event(s)
- Associate one event with one instruction.
  - external event: pick any instruction
  - multiple internal events: typically choose the earliest instruction.
  - multiple external events: prioritize
  - multiple internal and external events: prioritize
- Create Clean Break in Instruction Stream
  - Complete all instructions before excepting instruction
  - Abort excepting and all following instructions
    - this clean break is called a *"precise exception"*



EXCEPTIONS HANDLING (Hardware Tasks)

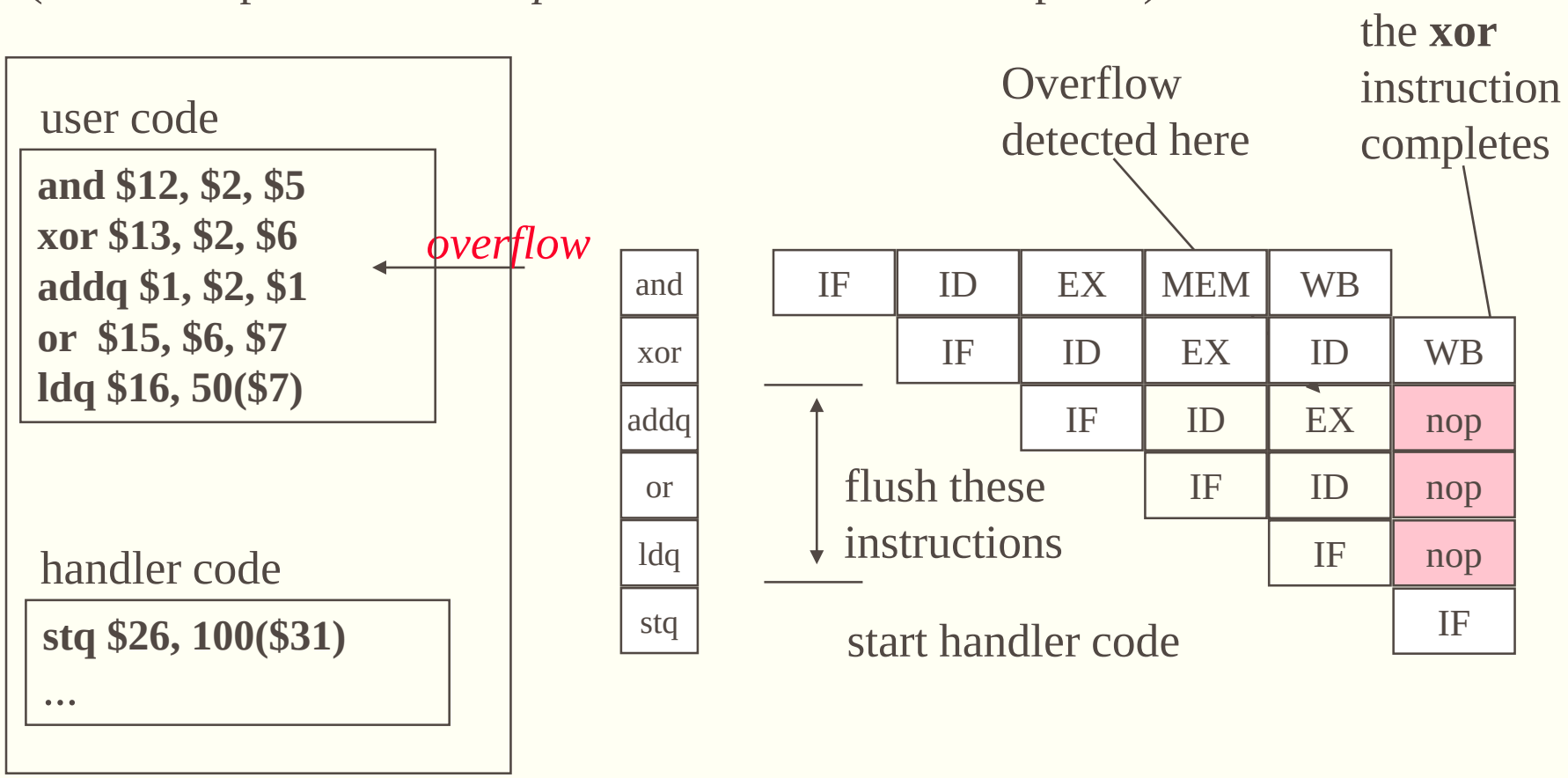
- Set status registers
  - Exception Address: the EXC_ADDR register
    - external exception: address of instruction about to be executed
    - internal exception: address of instruction causing the exception
      - except for arithmetic exceptions, where it is the following instruction
  - Cause of the Exception: the EXC_SUM and FPCR registers
    - was the exception due to division by zero, integer overflow, etc.
  - Others
    - which ones get set depends on CPU and exception type
- Disable interrupts and switch to kernel mode
- Jump to common exception handler location

EXCEPTIONS HANDLING (Software Tasks)

- Deal with event
- (Optionally) resume execution
  - using special REI (return from exception or interrupt) instruction
  - similar to a procedure return, but restores processor to user mode as a side effect.
- Where to resume execution?
  - usually re-execute the instruction causing exception

EXAMPLE: INTEGER OVERFLOW

(This example illustrates a *precise* version of the exception.)



REFERENCES

1. David A. Patterson and John L. Hennessy, “Computer Organization and Design: The Hardware/Software Interface”, Fifth Edition, Morgan Kaufmann / Elsevier, 2013.
2. Carl Hamacher, Zvonko Vranesic, Safwat Zaky and Naraig Manjikian, “Computer Organization and Embedded Systems”, Sixth Edition, Tata McGraw Hill, 2012.
3. John P. Hayes, “Computer Architecture and Organization”, Third Edition, Tata McGraw Hill, 1998.
4. William Stallings, “Computer Organization and Architecture – Designing for Performance”, Sixth Edition, Pearson Education, 2003.
5. John P. Hayes, “Computer Architecture and Organization”, Third Edition, Tata McGraw Hill, 2017.
6. V.P. Heuring, H.F. Jordan, “Computer Systems Design and Architecture”, Second Edition, Pearson Education, 2004.

Thank You…