#7 : MIPS Implementation

Computer Architecture 2019/2020

Ricardo Rocha

Computer Science Department, Faculty of Sciences, University of Porto

Slides based on the book

'Computer Organization and Design, The Hardware/Software Interface, 5th Edition

David Patterson and John Hennessy, Morgan Kaufmann'

Sections 4.1 – 4.6 and 4.9

Instruction Set Architecture

An **instruction set architecture (ISA)** is an abstract model of a computer that defines the interface between software and hardware. It is also referred to as **architecture** or **computer architecture**.

An ISA **defines everything a machine language programmer needs to know** in order to program a computer:

- Instruction set
- Supported data types
- What state there is, such as the memory hierarchy and registers
- State semantics, such as the memory consistency and addressing modes
- Input/output model

Instruction Set Architecture

An ISA is different from a microarchitecture, which is the set of processor design techniques used, in a particular processor, to implement the ISA. A **realization of an ISA is called an implementation**.

An ISA permits multiple implementations that may vary in performance, physical size, cost, among other things. Processors with different microarchitectures can implement a common ISA and software that has been written for an ISA can run on different implementations of the same ISA. This has enabled compatibility between different generations of computers to be easily achieved, and the development of computer families. For these reasons, the ISA is one of the **most important abstractions** in computing today.

Single-Register Architectures

Single-register (or accumulator) architectures have a **single register, called accumulator**, for all arithmetic instructions and use a memory-based operand-addressing mode.

For example, the add instruction would look like this:

```
add 200
```

meaning add the accumulator to the word in memory at address 200 and place the sum back into the accumulator. No registers are specified because the accumulator is known to be both the source and the destination of the operation.

Special-Purpose Register Architectures

Special-purpose register (or dedicated-register or extended accumulator) architectures have the **addition of registers dedicated to specific operations**. Registers might be included to act as indices for array references in data transfer instructions, to act as separate accumulators for multiply or divide instructions, and to serve as stack pointer.

Like the single-register accumulator architectures, one operand may be in memory for arithmetic instructions. However, there are also instructions where all the operands are registers.

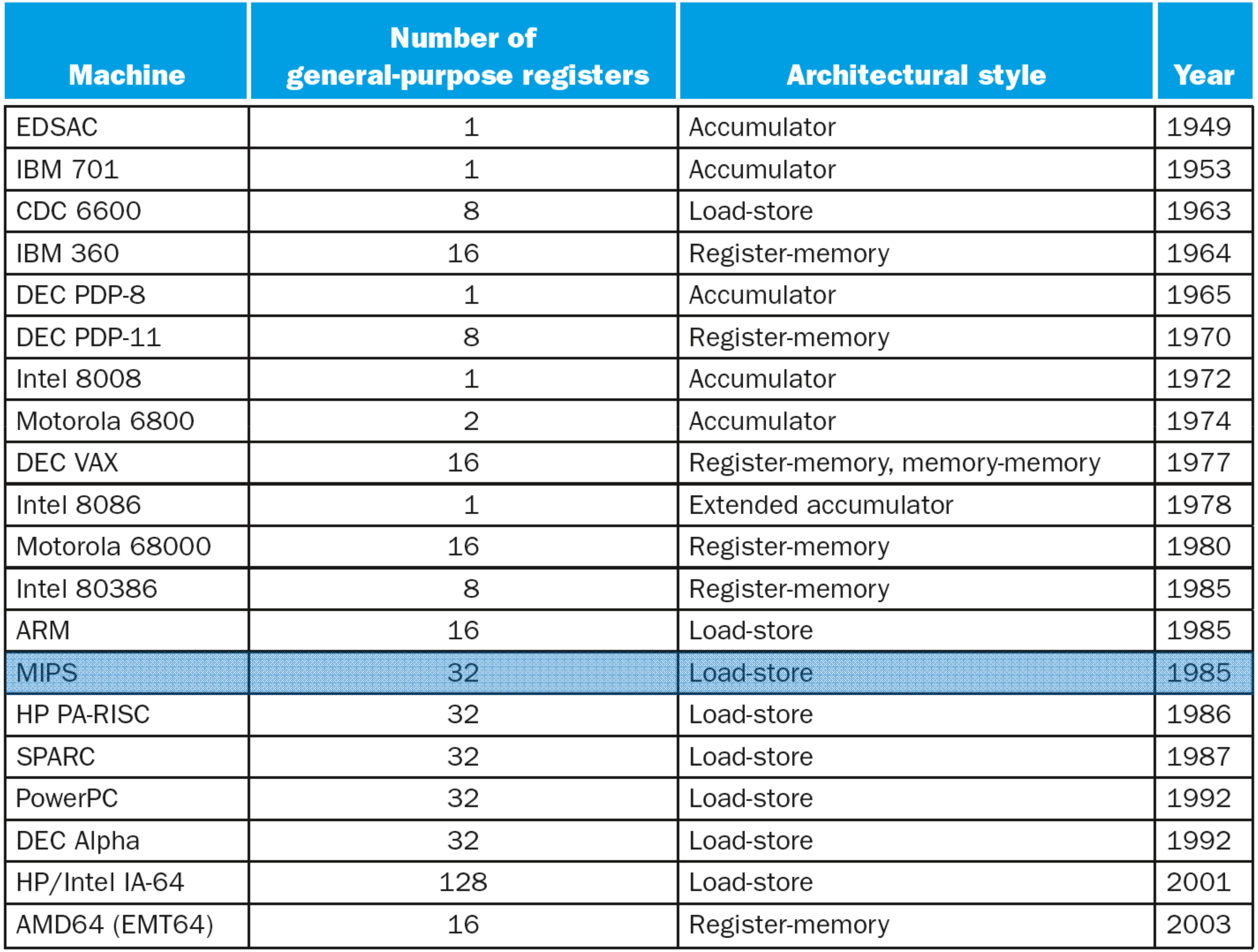
General-Purpose Register Architectures

General-purpose register architectures allow **all the registers to be used for any purpose**. This style of architecture may be further divided into:

- **Register-memory architectures** allow one operand to be in memory (as found in accumulator architectures)
- **Load-store or register-register architectures** demand that operands always be in registers

In addition, a style of architecture in which all operands can also be in memory is called **memory-memory architecture**.

Architectures Over Years



Machine	Number of general-purpose registers	Architectural style	Year
EDSAC	1	Accumulator	1949
IBM 701	1	Accumulator	1953
CDC 6600	8	Load-store	1963
IBM 360	16	Register-memory	1964
DEC PDP-8	1	Accumulator	1965
DEC PDP-11	8	Register-memory	1970
Intel 8008	1	Accumulator	1972
Motorola 6800	2	Accumulator	1974
DEC VAX	16	Register-memory, memory-memory	1977
Intel 8086	1	Extended accumulator	1978
Motorola 68000	16	Register-memory	1980
Intel 80386	8	Register-memory	1985
ARM	16	Load-store	1985
MIPS	32	Load-store	1985
HP PA-RISC	32	Load-store	1986
SPARC	32	Load-store	1987
PowerPC	32	Load-store	1992
DEC Alpha	32	Load-store	1992
HP/Intel IA-64	128	Load-store	2001
AMD64 (EMT64)	16	Register-memory	2003

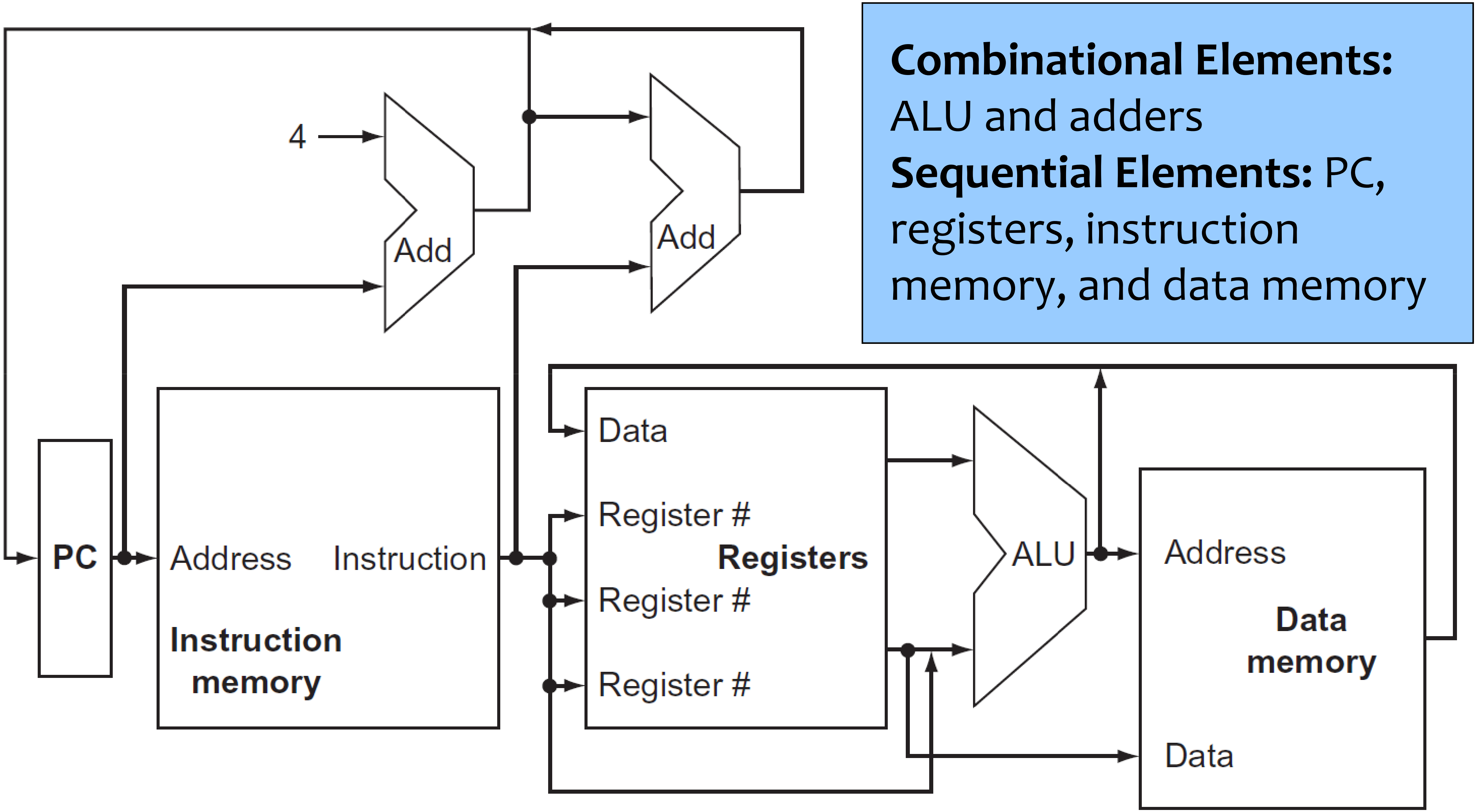
MIPS Simplified Implementation

We will consider a simplified implementation of the MIPS instruction set architecture as a way to illustrate how it determines many aspects of the implementation, and how the choice of different implementation strategies affects the clock rate and CPI for the computer.

We will consider an implementation that includes a subset of the core MIPS instruction set:

- Memory reference instructions : **lw** (load word) and **sw** (store word)
- Arithmetic-logical instructions: **add**, **sub**, **and**, **or**, and **slt**
- Control transfer instructions: **beq** (branch equal) and **j** (jump unconditionally)

Data Path – Overview



Combinational Elements: ALU and adders
Sequential Elements: PC, registers, instruction memory, and data memory

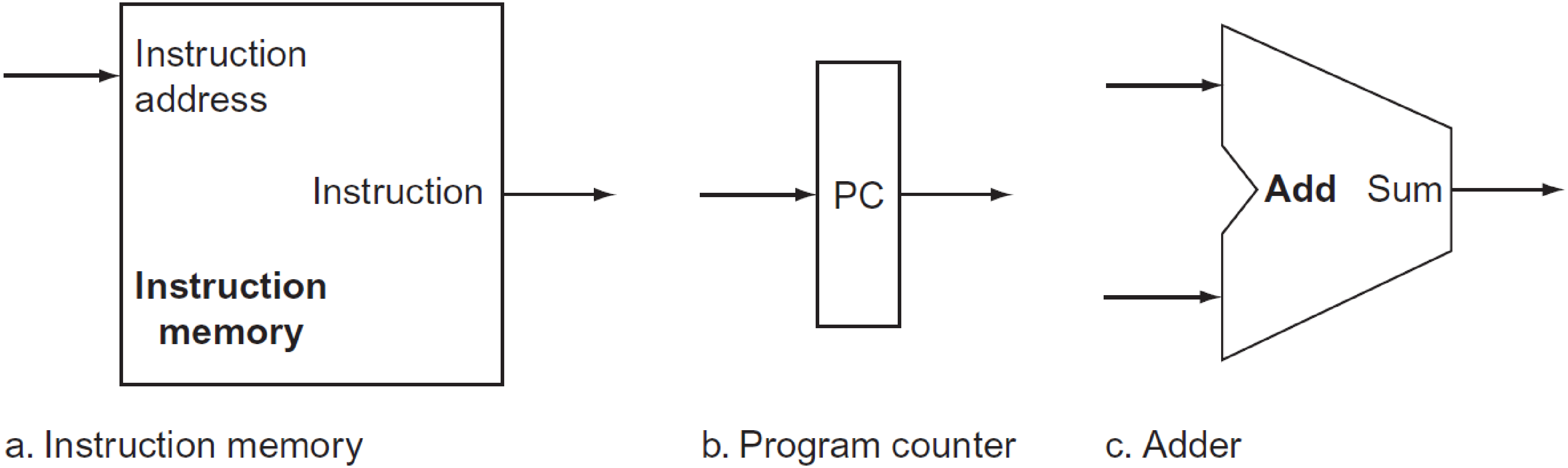
Data Path – Instruction Fetch

To execute any instruction, we must start by fetching the instruction from memory. To prepare for executing the next instruction, we must also increment the program counter so that it points at the next instruction, 4 bytes later.

Fetching instructions thus requires:

- **Memory unit** where instructions are stored
- **Program counter (PC)** to hold the address of the current/next instruction
- **Adder** to increment the PC to the address of the next instruction

Data Path – Instruction Fetch



a. Instruction memory

b. Program counter

c. Adder

FIGURE 4.5 Two state elements are needed to store and access instructions, and an adder is needed to compute the next instruction address. The state elements are the instruction memory and the program counter. The instruction memory need only provide read access because the datapath does not write instructions. Since the instruction memory only reads, we treat it as combinational logic: the output at any time reflects the contents of the location specified by the address input, and no read control signal is needed. (We will need to write the instruction memory when we load the program; this is not hard to add, and we ignore it for simplicity.) The program counter is a 32-bit register that is written at the end of every clock cycle and thus does not need a write control signal. The adder is an ALU wired to always add its two 32-bit inputs and place the sum on its output.

Data Path – Instruction Fetch

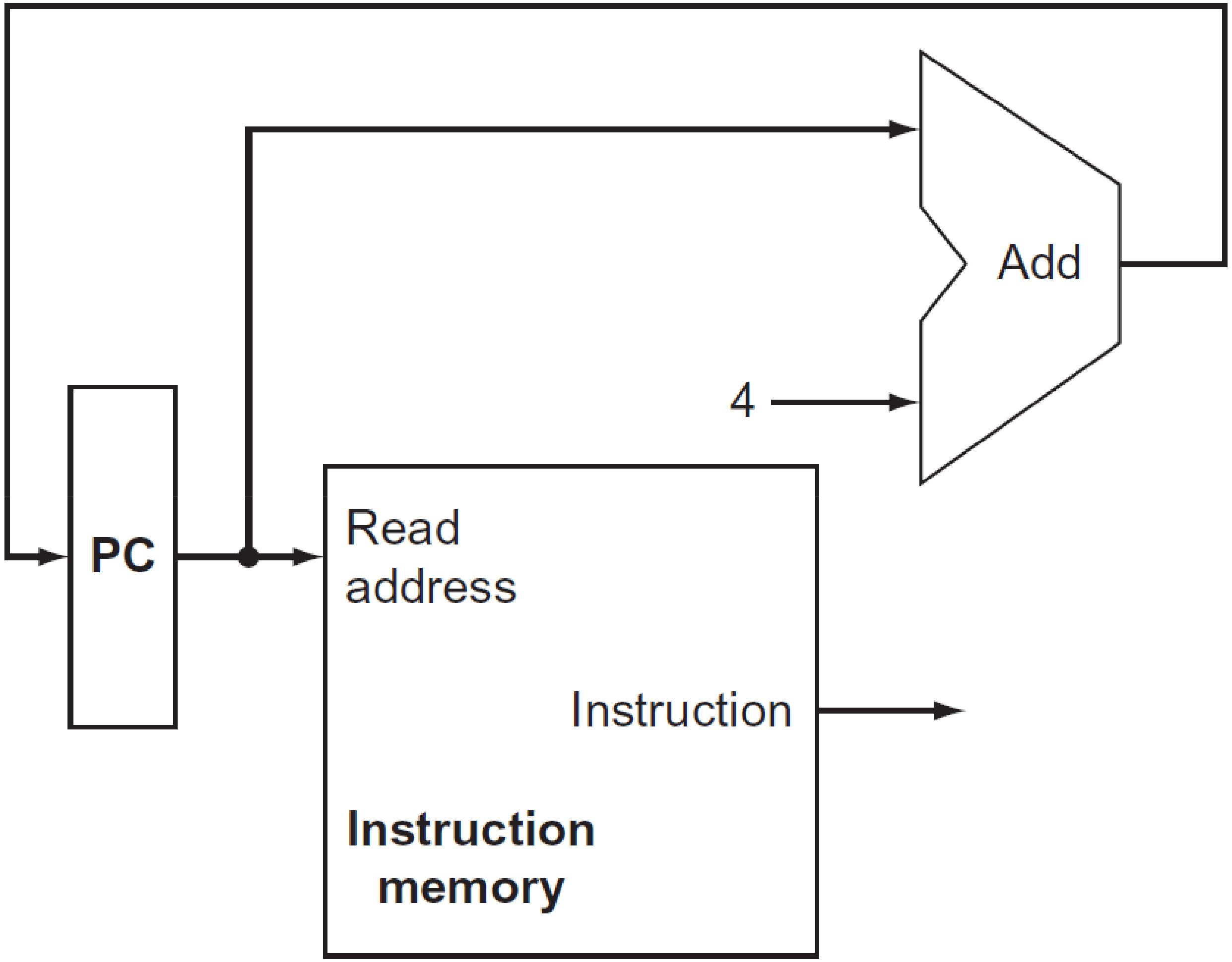


FIGURE 4.6 A portion of the datapath used for fetching instructions and incrementing the program counter. The fetched instruction is used by other parts of the datapath.

Data Path – Arithmetic-Logical Operations

The R-type instructions (e.g., **add $t1, $t2, $t3**) perform arithmetic-logical operations. They all read two registers, perform an ALU operation on the contents of the registers, and write the result to a register.

R-type instructions thus require:

- **Register file** where the register's contents are stored
- **ALU** to operate on the values read from the registers

Data Path – Arithmetic-Logical Operations

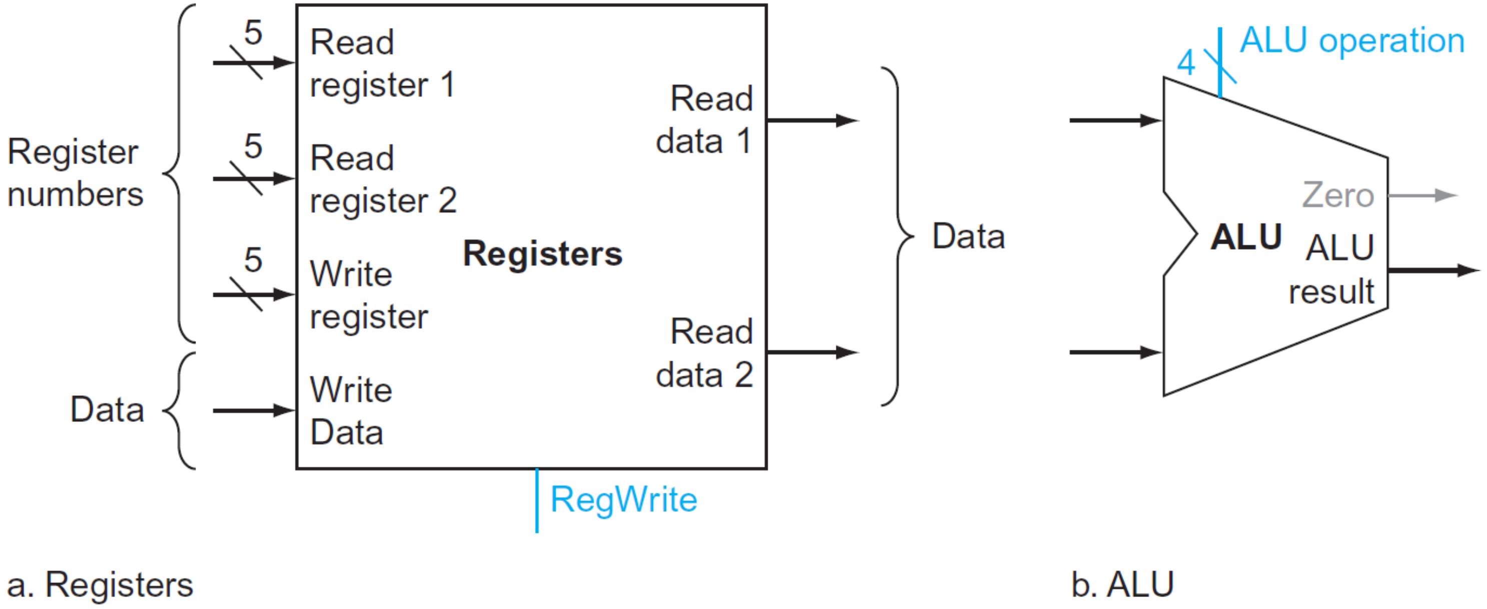


FIGURE 4.7 The two elements needed to implement R-format ALU operations are the register file and the ALU. The register file contains all the registers and has two read ports and one write port. The design of multiported register files is discussed in Section B.8 of 🌐 Appendix B. The register file always outputs the contents of the registers corresponding to the Read register inputs on the outputs; no other control inputs are needed. In contrast, a register write must be explicitly indicated by asserting the write control signal. Remember that writes are edge-triggered, so that all the write inputs (i.e., the value to be written, the register number, and the write control signal) must be valid at the clock edge. Since writes to the register file are edge-triggered, our design can legally read and write the same register within a clock cycle: the read will get the value written in an earlier clock cycle, while the value written will be available to a read in a subsequent clock cycle. The inputs carrying the register number to the register file are all 5 bits wide, whereas the lines carrying data values are 32 bits wide. The operation to be performed by the ALU is controlled with the ALU operation signal, which will be 4 bits wide, using the ALU designed in 🌐 Appendix B. We will use the Zero detection output of the ALU shortly to implement branches. The overflow output will not be needed until Section 4.9, when we discuss exceptions; we omit it until then.

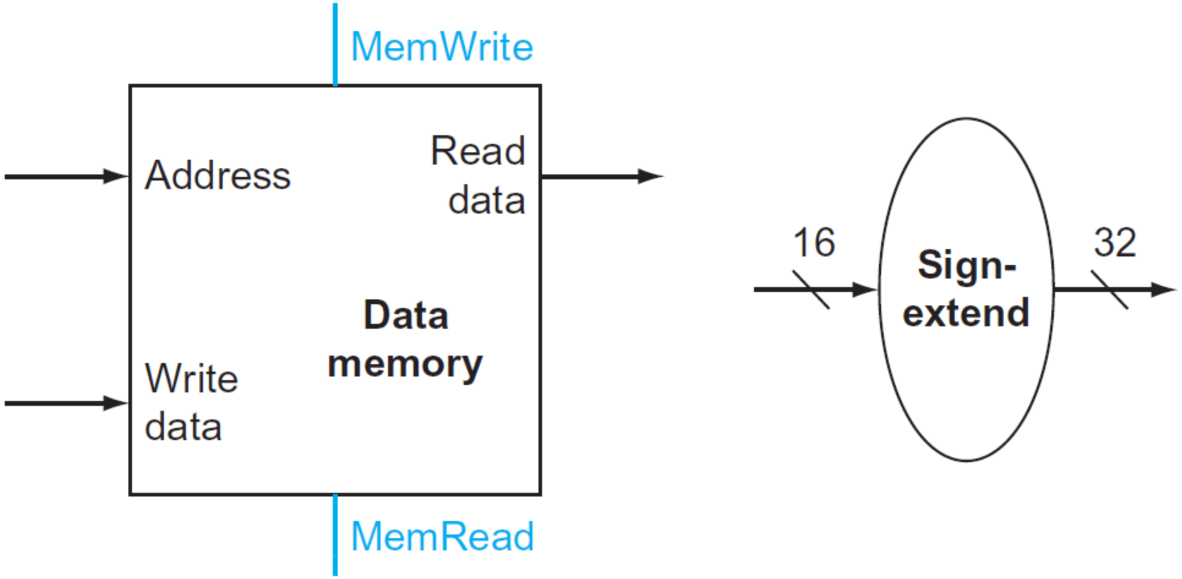
Data Path – Loads and Stores

Memory reference instructions (e.g., **lw $t1, offset($t2)**) perform load word and store word operations. They compute a memory address by adding a base register to a 16-bit signed offset field contained in the instruction. Moreover, if the instruction is a store, the value to be stored must also be read from a specified register. If the instruction is a load, the value read from memory must be written into the specified register.

Memory reference instructions thus require:

- **Register file** where the register's contents are loaded/stored
- **ALU** to operate on the value read from the base register and the offset
- **Memory unit** where data is stored
- **Sign extension unit** to sign-extend the 16-bit offset field to a 32-bit signed value

Data Path – Loads and Stores



a. Data memory unit

b. Sign extension unit

FIGURE 4.8 The two units needed to implement loads and stores, in addition to the register file and ALU of Figure 4.7, are the data memory unit and the sign extension unit. The memory unit is a state element with inputs for the address and the write data, and a single output for the read result. There are separate read and write controls, although only one of these may be asserted on any given clock. The memory unit needs a read signal, since, unlike the register file, reading the value of an invalid address can cause problems, as we will see in Chapter 5. The sign extension unit has a 16-bit input that is sign-extended into a 32-bit result appearing on the output (see Chapter 2). We assume the data memory is edge-triggered for writes. Standard memory chips actually have a write enable signal that is used for writes. Although the write enable is not edge-triggered, our edge-triggered design could easily be adapted to work with real memory chips. See Section B.8 of Appendix B for further discussion of how real memory chips work.

Data Path – Branches

Branch instructions (e.g., **beq $t1, $t2, offset**) compare two registers and compute a branch target address by adding the PC+4 (the base for computing the branch target address) to a 16-bit signed offset field contained in the instruction. The offset is shifted left 2 bits so that it is a word offset (this shift increases the offset range by a factor of 4).

Branch instructions thus require:

- **Register file** where the register's contents are stored
- **ALU** to compare the values read from the registers
- **Adder** to compute the branch target address
- **Sign extension and shift left units** to sign-extend and shift left the 16-bit offset field to a 32-bit signed value

Data Path – Branches

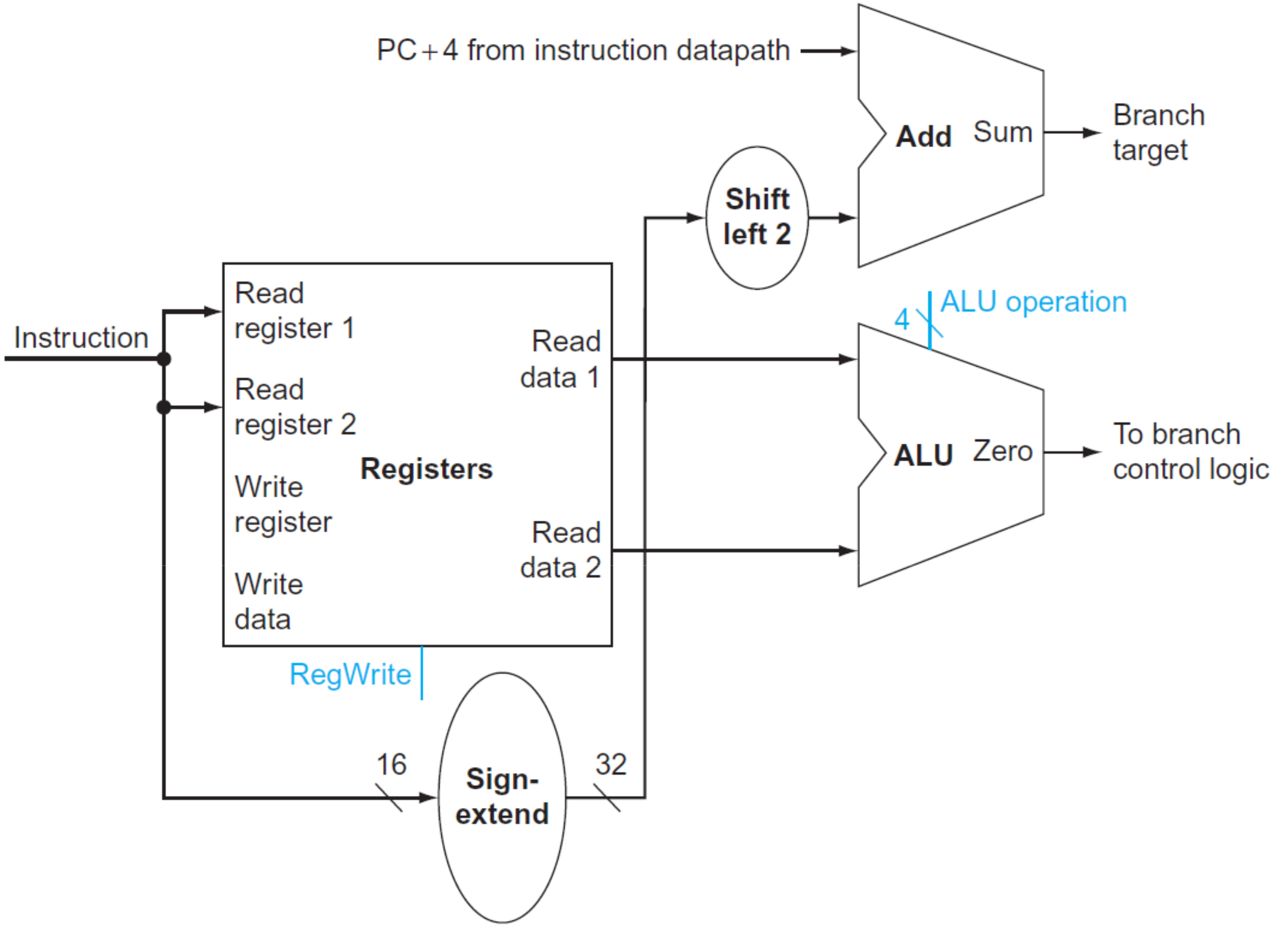


FIGURE 4.9 The datapath for a branch uses the ALU to evaluate the branch condition and a separate adder to compute the branch target as the sum of the incremented PC and the sign-extended, lower 16 bits of the instruction (the branch displacement), shifted left 2 bits. The unit labeled *Shift left 2* is simply a routing of the signals between input and output that adds 00_{two} to the low-order end of the sign-extended offset field; no actual shift hardware is needed, since the amount of the "shift" is constant. Since we know that the offset was sign-extended from 16 bits, the shift will throw away only "sign bits." Control logic is used to decide whether the incremented PC or branch target should replace the PC, based on the Zero output of the ALU.

Data Path – Jumps

Jump instructions (e.g., **j label**) operate by replacing the lower 28 bits of the PC with the lower 26 bits of label field contained in the instruction shifted left by 2 bits (as before, this shift increases the label range by a factor of 4).

Jump instructions thus require:

- **Shift left unit** to shift left the 26-bit label field to a 32-bit value

Data Path – Full Picture

The simplest data path will **execute all instructions in one clock cycle**.

This means that **no element can be used more than once per instruction** – any element needed more than once must be duplicated – and that **most elements can be shared by different instruction flows** – to share an element we may need to allow multiple input connections, using a multiplexor and control signal to select among the multiple inputs.

Data Path – Full Picture

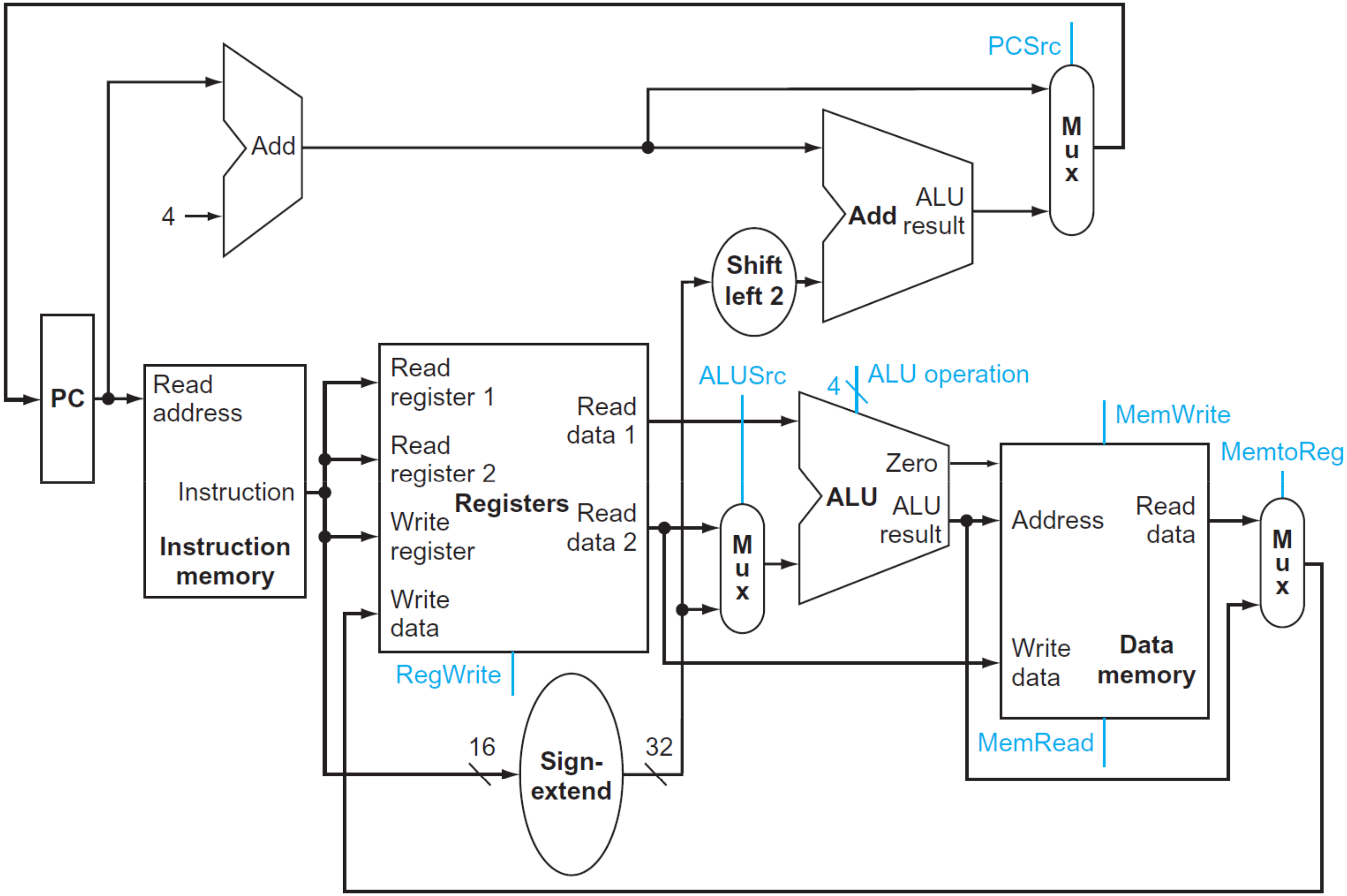
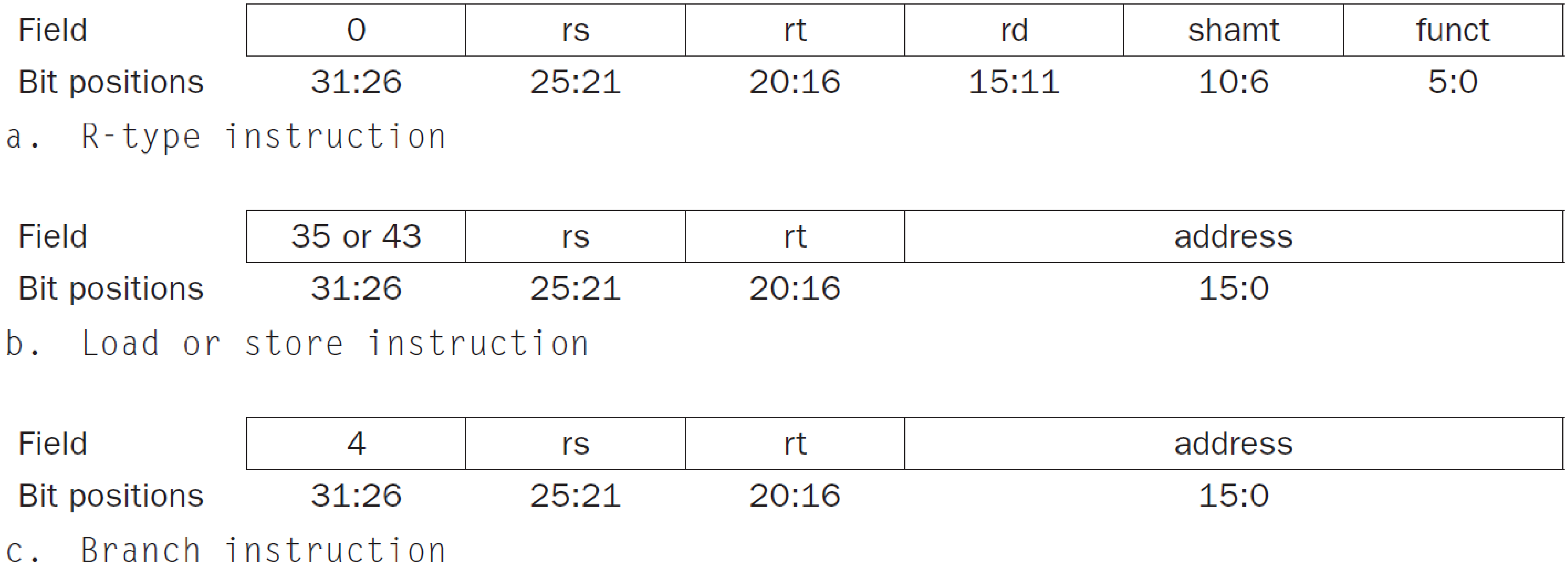


FIGURE 4.11 The simple datapath for the core MIPS architecture combines the elements required by different instruction classes. The components come from Figures 4.6, 4.9, and 4.10. This datapath can execute the basic instructions (load-store word, ALU operations, and branches) in a single clock cycle. Just one additional multiplexor is needed to integrate branches. The support for jumps will be added later.

Control – Overview

To understand how to connect the fields of an instruction to the data path and identify all control lines necessary, let's remember the different instruction formats (the jump instruction is discussed later).



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

a. R-type instruction

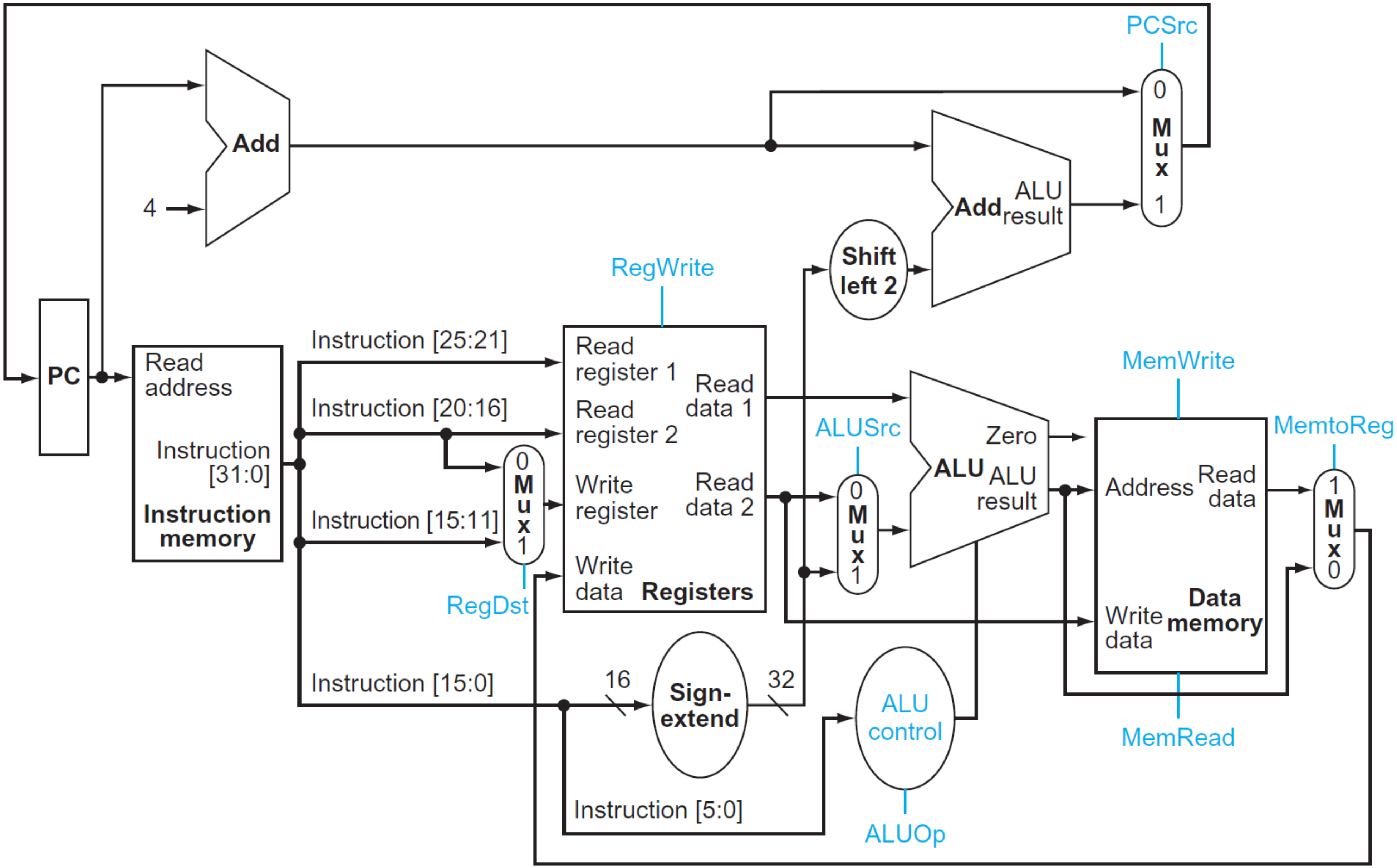
Field	35 or 43	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

b. Load or store instruction

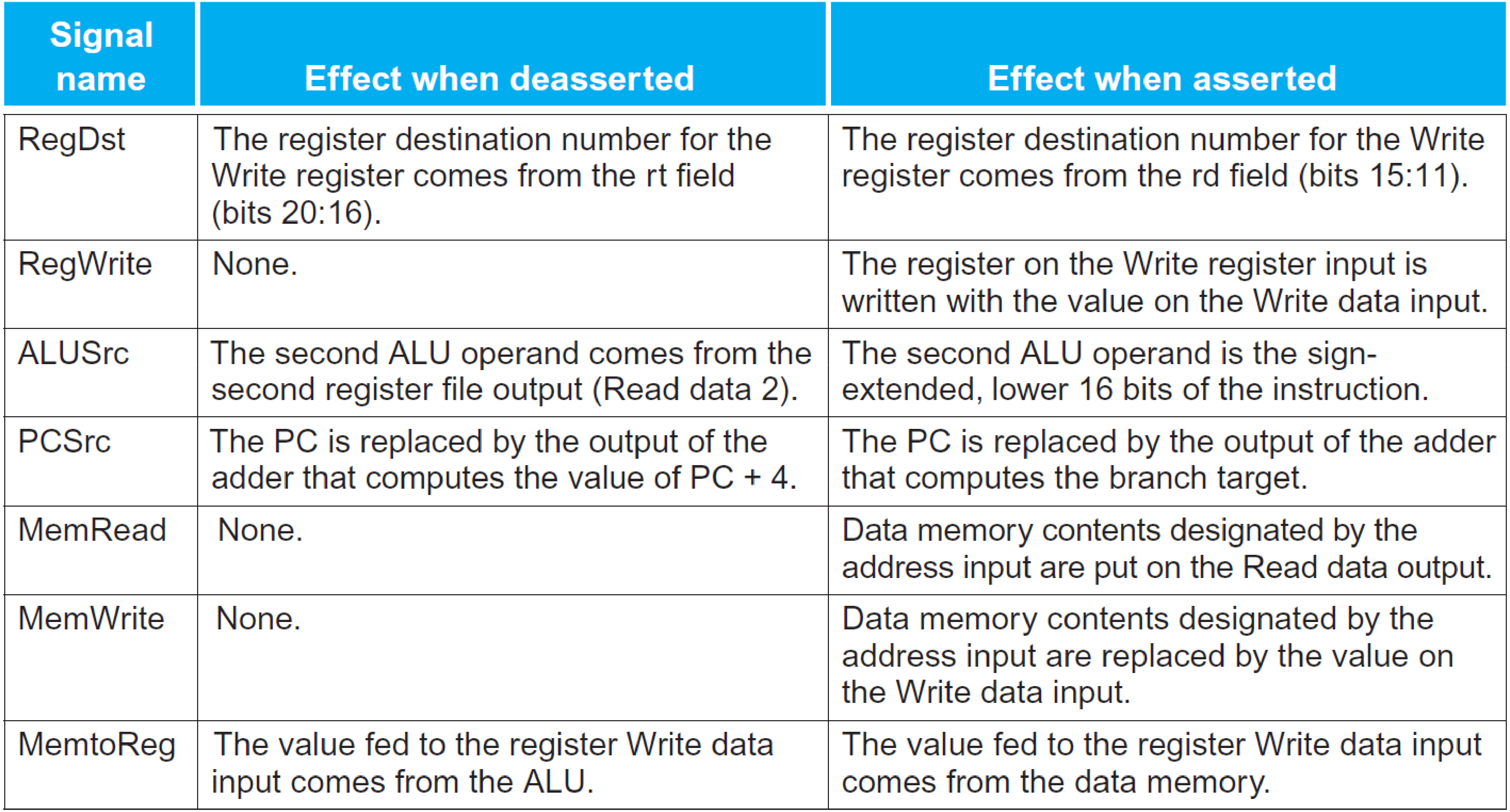
Field	4	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

c. Branch instruction

Control – Overview

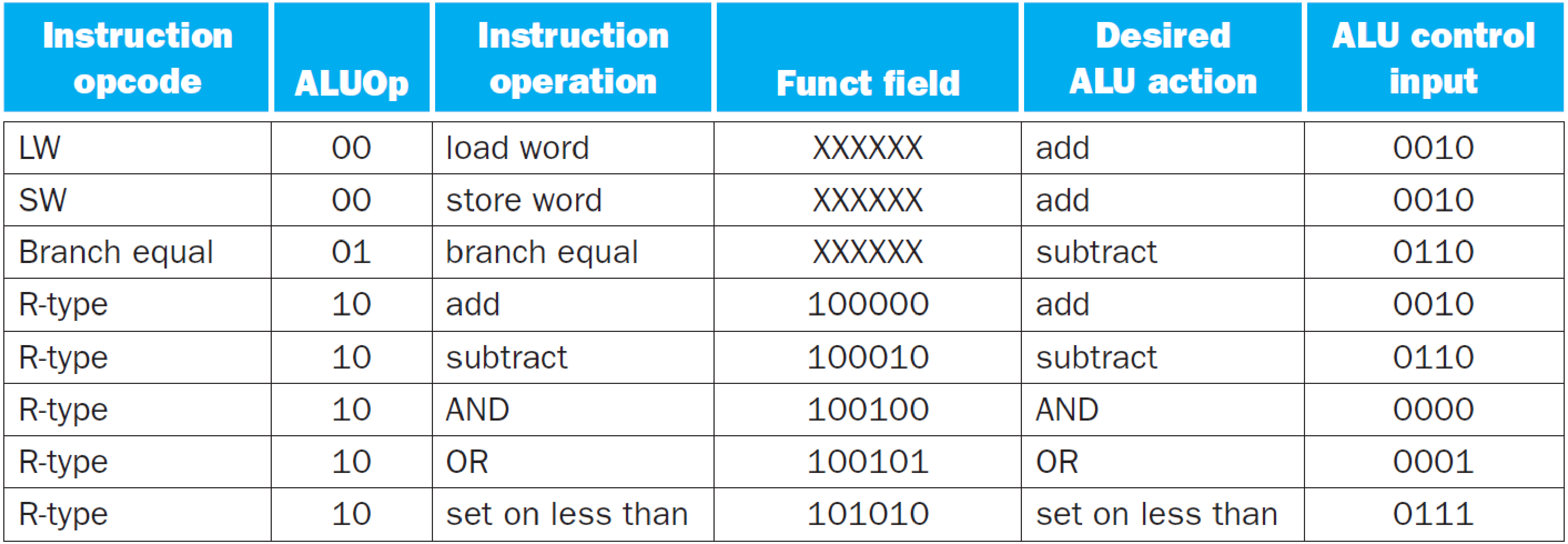


Single-Bit Control Lines



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.

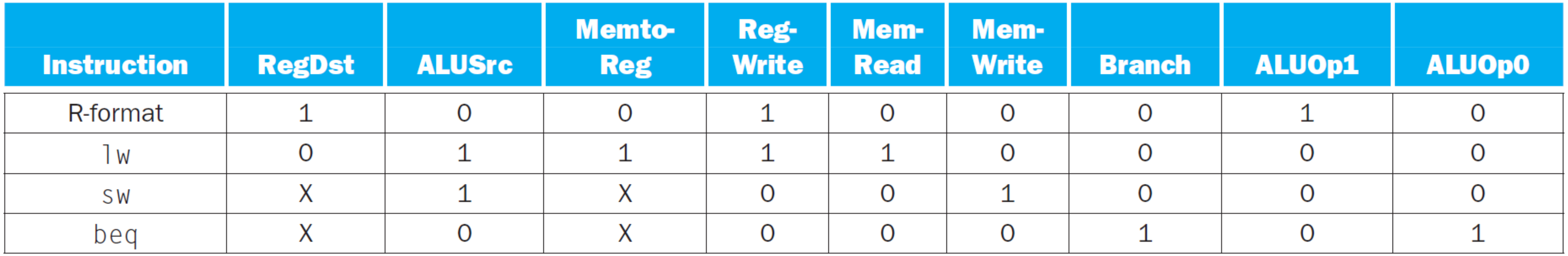
ALUOp 2-Bit Control Line



Instruction opcode	ALUOp	Instruction operation	Funct 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	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

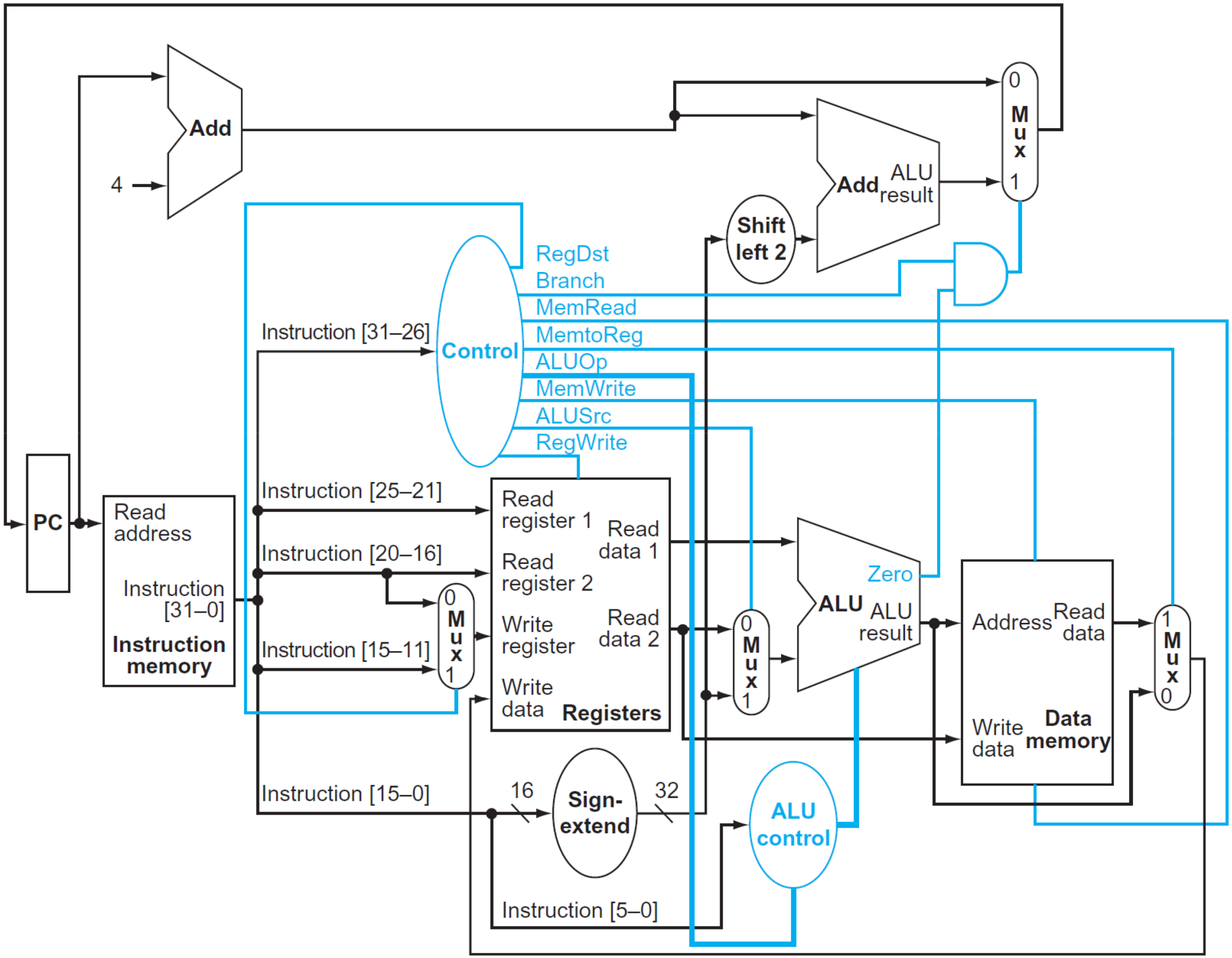
Control Lines Setting

The setting of the control lines is completely determined by the opcode fields of the instruction. Each control signal can be 0, 1, or don't care (X) for each of the opcode values.



Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
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

Control – Full Picture

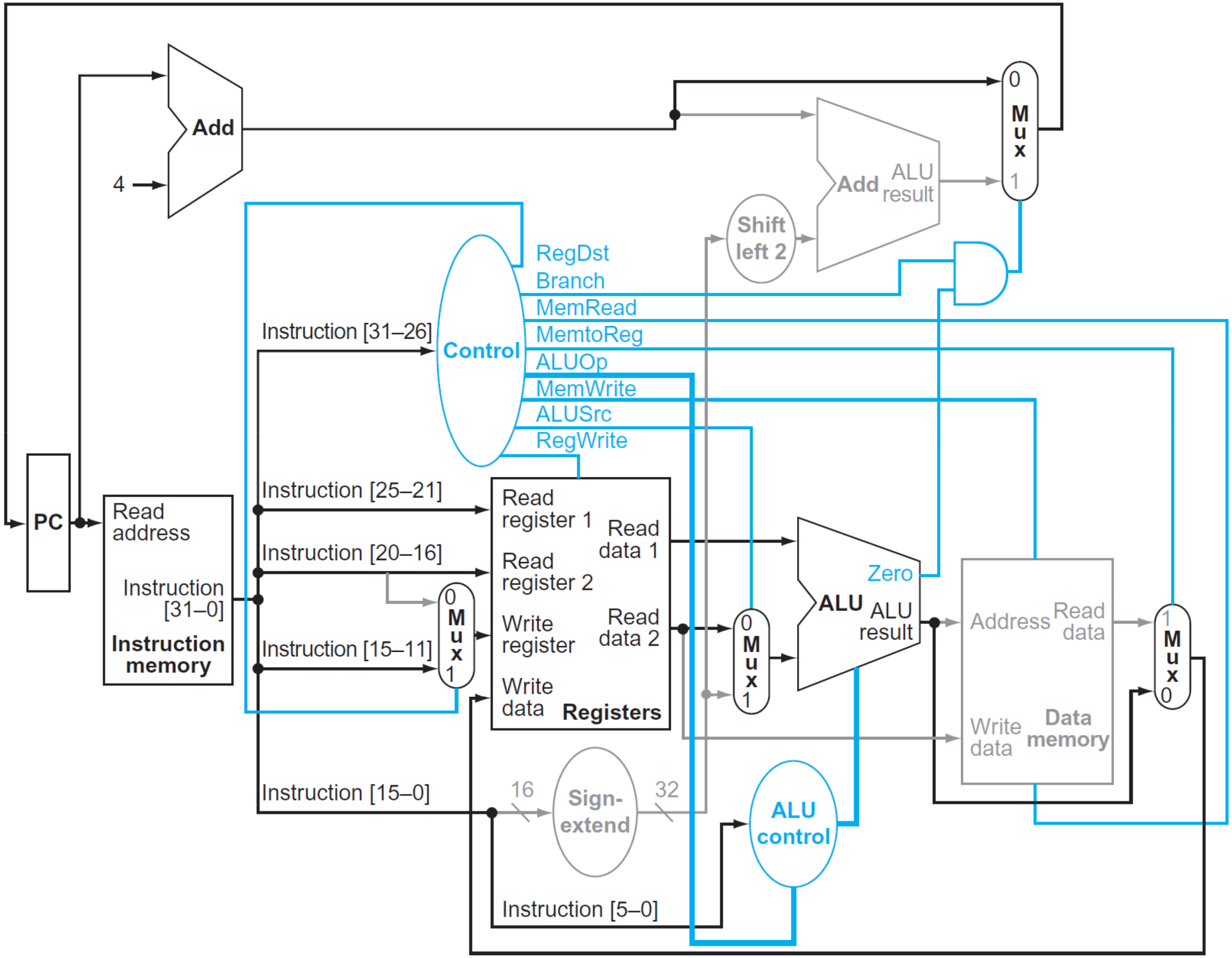


Control – R-Type Instructions

We can think of a R-type instruction (e.g., **add $t1, $t2, $t3**) as operating in four steps:

- The instruction is fetched and the PC is incremented
- The main control unit computes the setting of the control lines and two registers ($t2 and $t3) are read from the register file
- The ALU operates on the data read from the register file using the function code (bits 5:0 of the instruction)
- The result from the ALU is written into the register file using bits 15:11 of the instruction to select the destination register ($t1)

Control – R-Type Instructions

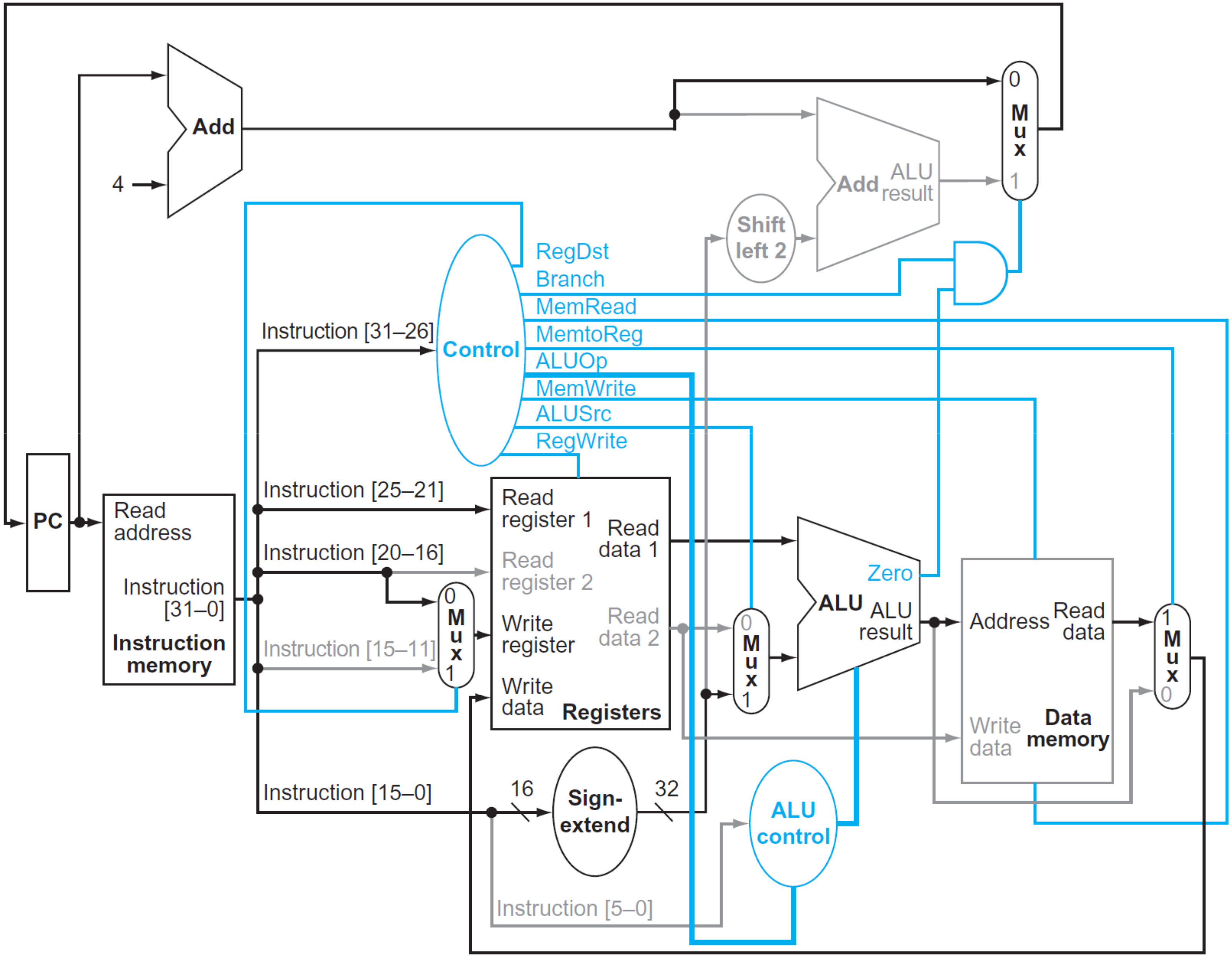


Control – Load Instructions

We can think of a load instruction (e.g., **lw $t1, offset($t2)**) as operating in five steps:

- The instruction is fetched and the PC is incremented
- The main control unit computes the setting of the control lines and a register ($t2) is read from the register file
- The ALU computes the sum of the value read from the register file and the sign-extended lower 16 bits of the instruction (offset)
- The result from the ALU is used as the address for the data memory
- The data from the memory unit is written into the register file using bits 20:16 of the instruction to select the destination register ($t1)

Control – Load Instructions

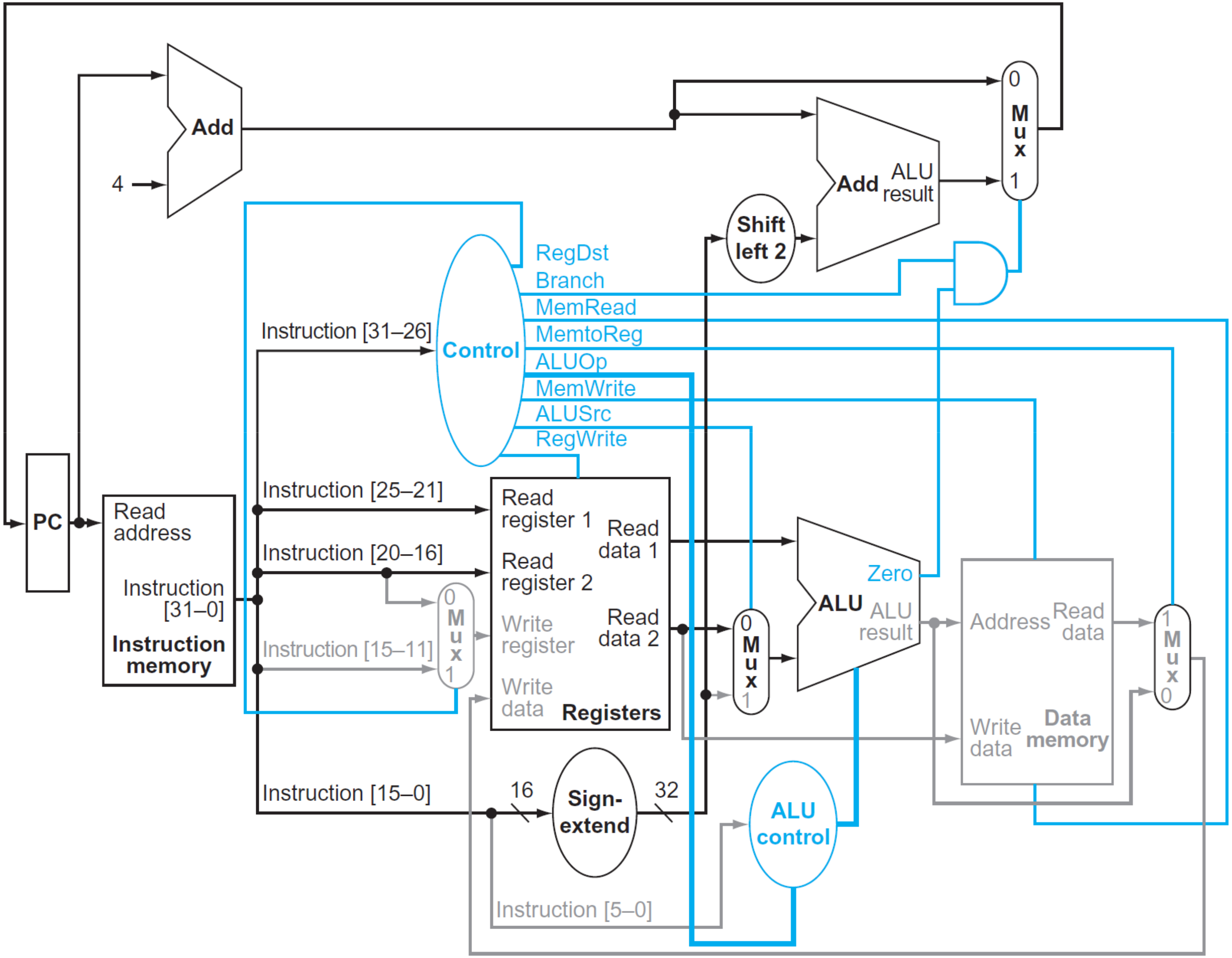


Control – Branch Instructions

We can think of a branch instruction (e.g., **beq $t1, $t2, offset**) as operating in four steps:

- The instruction is fetched and the PC is incremented
- The main control unit computes the setting of the control lines and two registers ($t1 and $t2) are read from the register file
- The ALU performs a subtract (beq) on the values read from the register file and the adder computes the sum (branch target address) of the PC+4 and the sign-extended lower 16 bits of the instruction (offset) shifted left by two
- The Zero result from the ALU is used to decide which result (PC+4 or branch target address) to store into the PC

Control – Branch Instructions

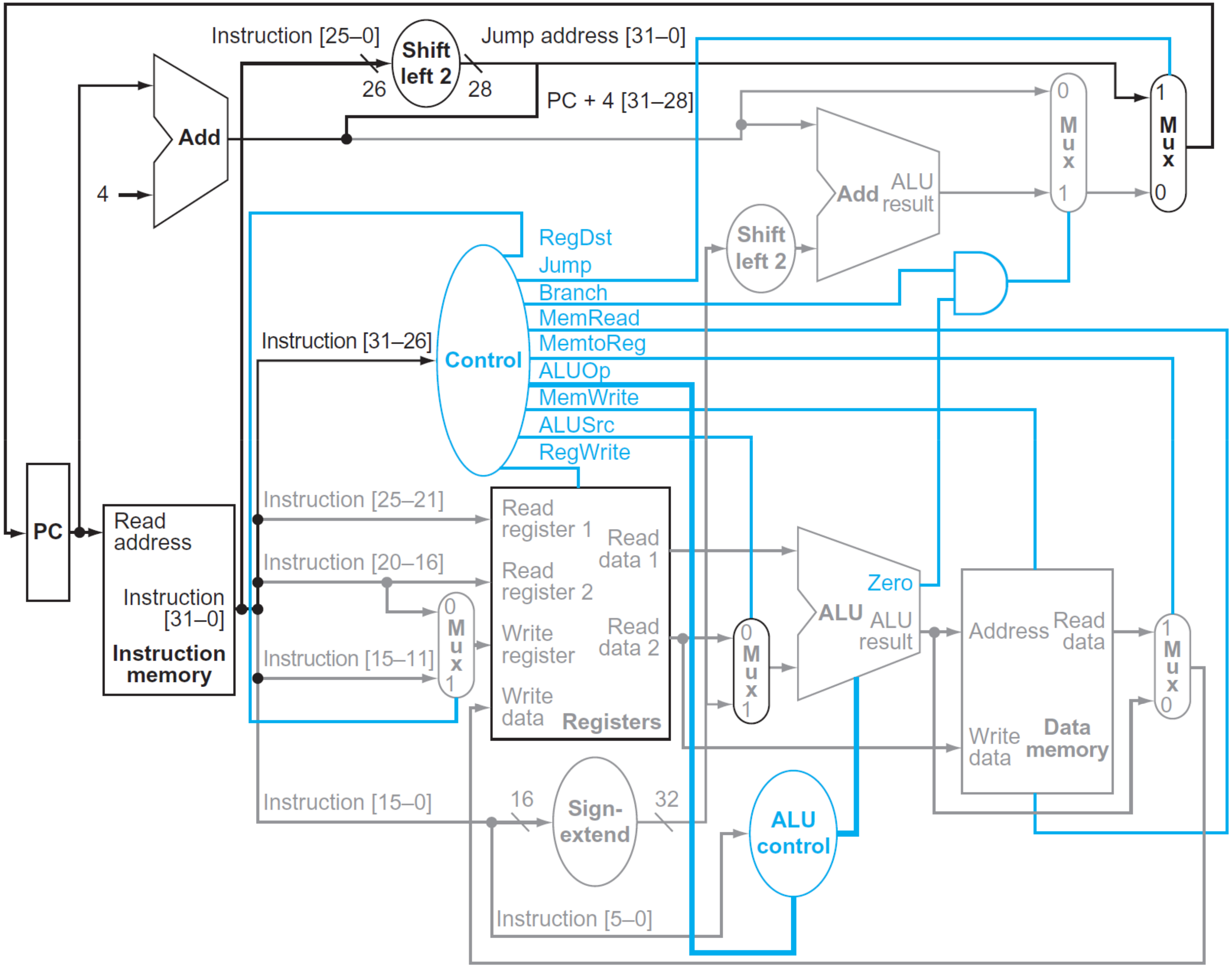


Control – Jump Instructions

We can think of a jump instruction (e.g., **j label**) as operating in three steps:

- The instruction is fetched and the PC is incremented
- The main control unit computes the setting of the control lines and the top 4-bits of the PC+4 are concatenated with the lower 26 bits of label field shifted left by 2 bits
- The new control line Jump is used to update the PC accordingly

Control – Jump Instructions

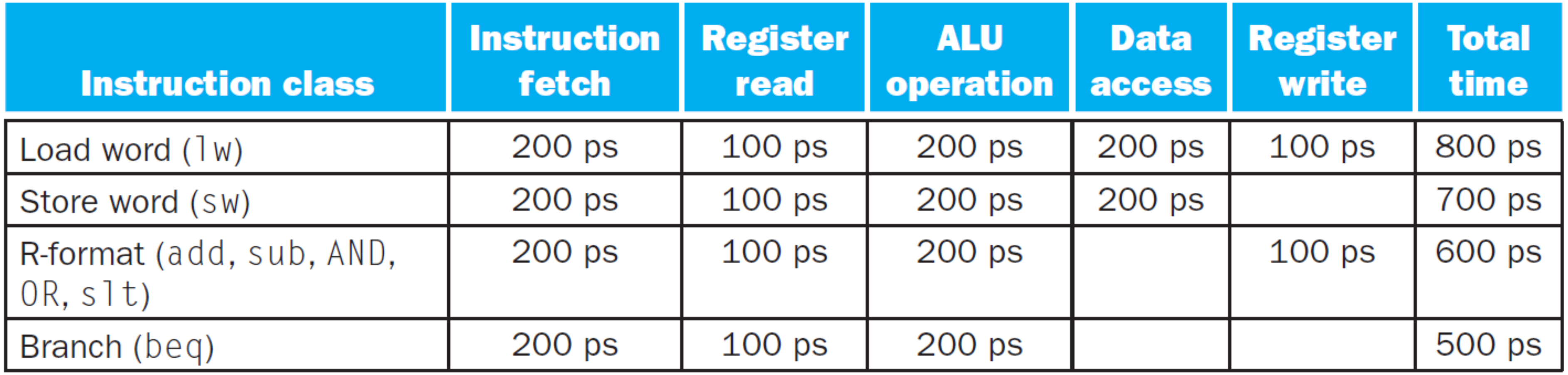


Single-Cycle Implementation

In a single-cycle implementation, **all instructions are executed in one clock cycle and take the same time to execute**. This makes the implementation easy to understand, but too inefficient to be practical since the **slowest instruction to execute determines the clock cycle**.

Early computers, with very simple instruction sets, did use single-cycle implementations. However, if we **support a floating-point unit or an instruction set with more complex instructions, the single-cycle implementation would be too slow**. Moreover, it is useless to design techniques that reduce the delay of the common case since they do not improve the worst-case cycle time.

Single-Cycle Implementation



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

Assuming the execution times above, the clock cycle for a single-cycle implementation is 800ps, which is the execution time of the slowest instruction (lw). Although all the other instructions (sw, add, sub, and, or, slt, beq) take less time, they still execute in 800ps.

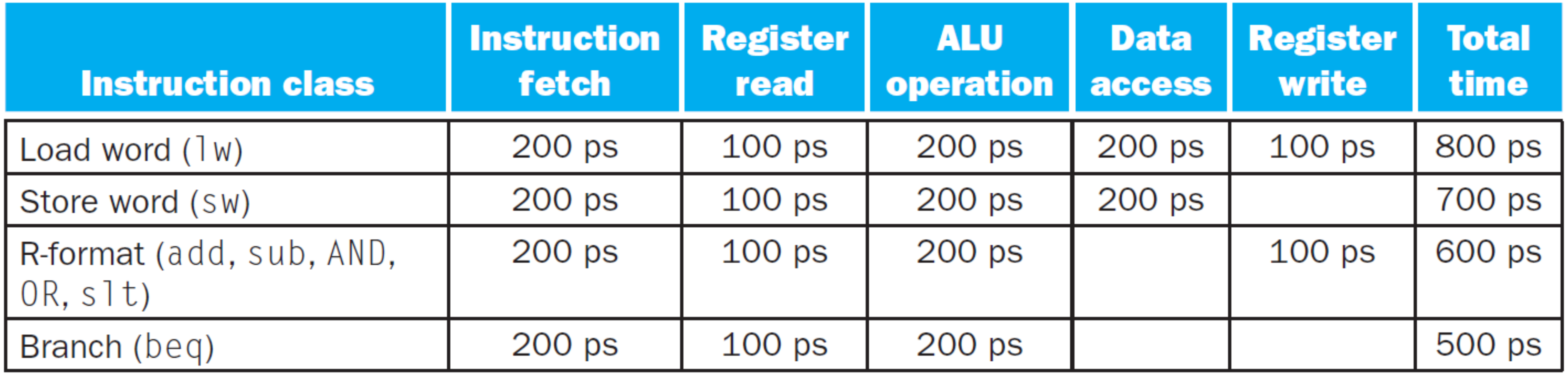
Multi-Cycle Implementation

In a multi-cycle implementation, an **instruction is executed in multiple clock cycles but only takes the clock cycles it actually needs**.

Each instruction is split into a series of steps corresponding to the functional unit operations it traverses (e.g., instruction fetch, register read, ALU operation) and uses the clock cycle to move between steps. **Each step in the execution will take one clock cycle** and different instructions can take a different number of clock cycles to execute.

The **slowest functional unit (step) to execute determines the clock cycle**. It is thus fundamental to balance the amount of work done in each step since we want to minimize the clock time cycle.

Multi-Cycle Implementation



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

Assuming the five steps above, the clock cycle for a multi-cycle implementation is 200ps, which is the execution time of the slowest steps (instruction fetch, ALU operation, and data access).

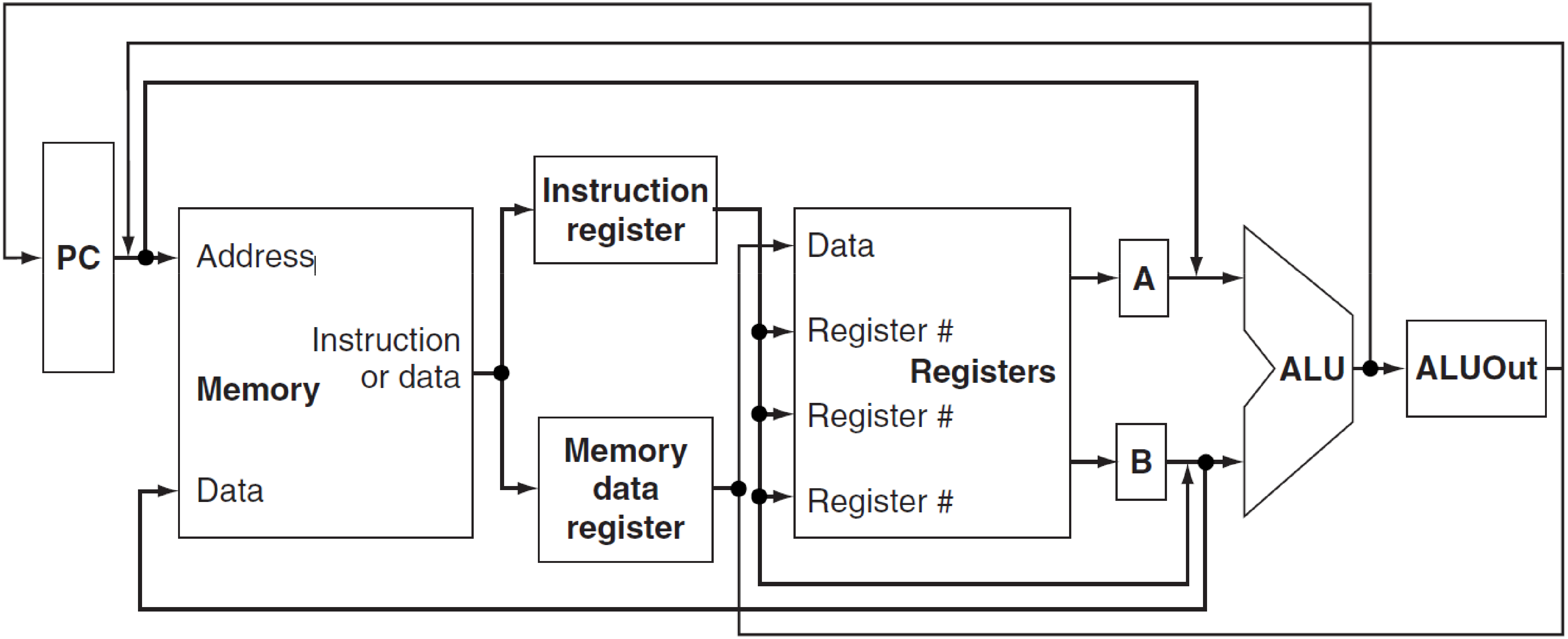
The load (lw) and store (sw) instructions execute in 1000ps (1ns), the R-type instructions in 800ps and the branch (beq) instructions in 600ps.

Multi-Cycle Implementation

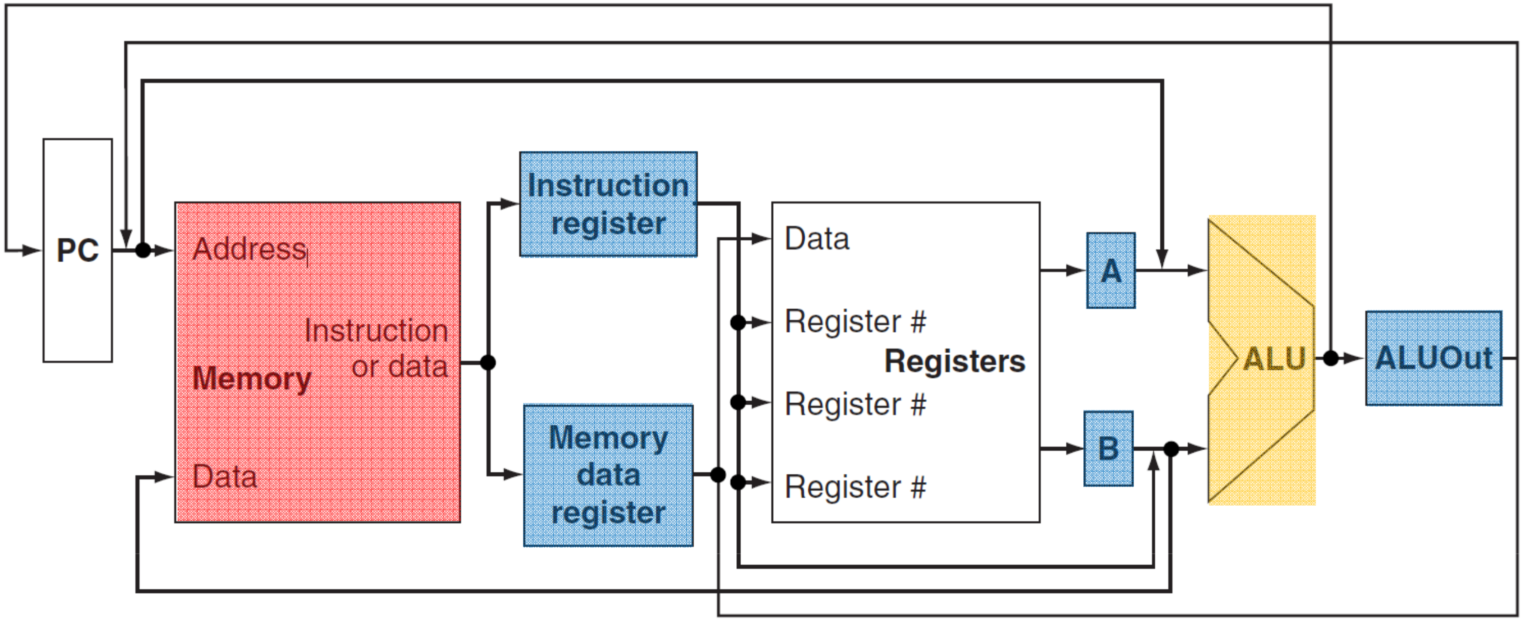
The multi-cycle implementation **allows a functional unit to be used more than once per instruction**, as long as it is used on different clock cycles. This sharing can help **reduce the amount of hardware required**. On the other hand, **extra registers are required** since at the end of a clock cycle the data must be saved.

The ability to allow instructions to take different numbers of clock cycles and the ability to share functional units within the execution of a single instruction are the major advantages of a multi-cycle implementation.

Multi-Cycle Data Path – Overview



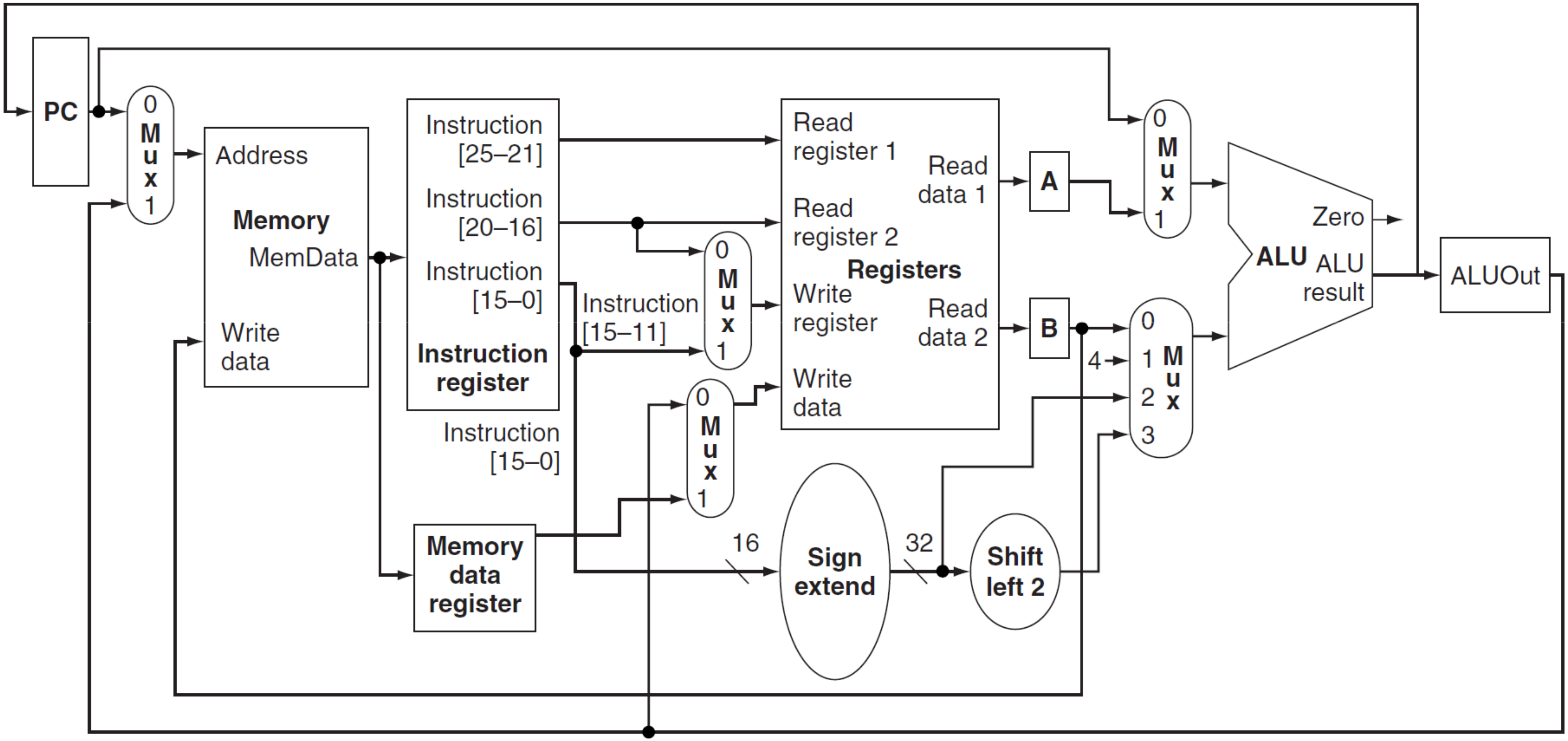
Multi-Cycle Data Path – Overview



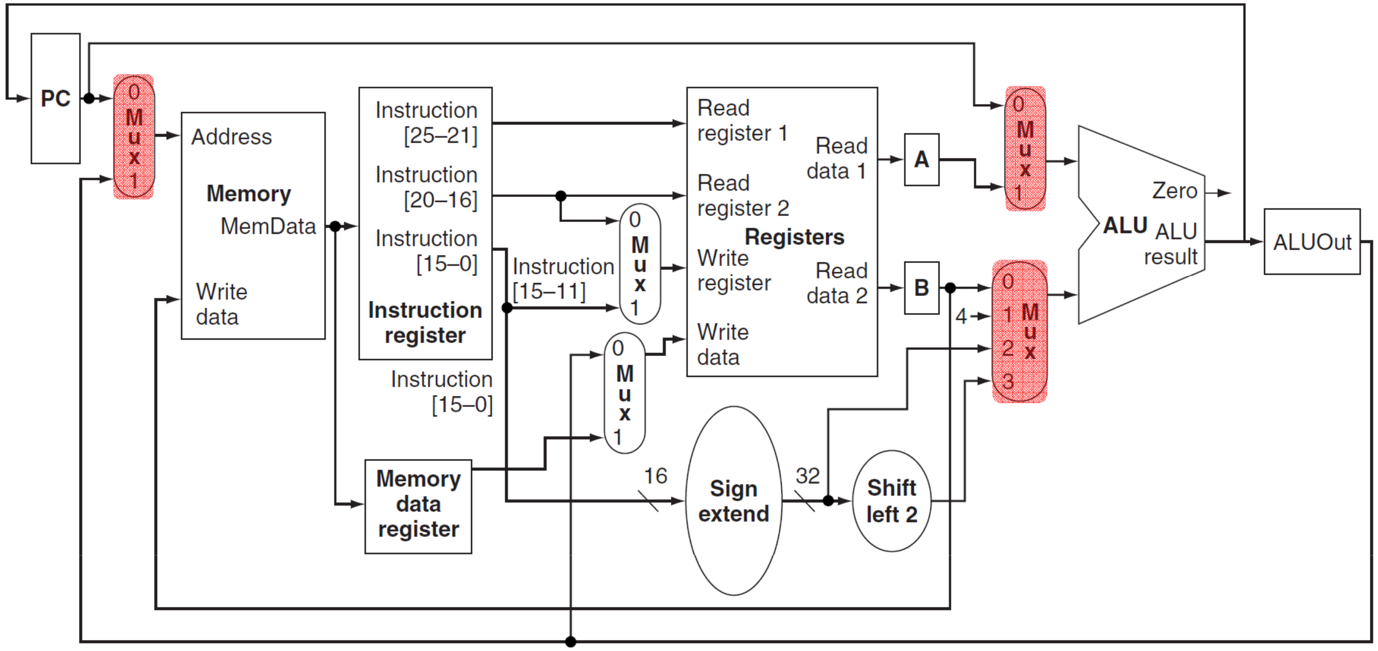
Differences for the single-cycle data path:

- A single memory unit used for both instructions and data
- A single ALU rather than an ALU and two adders
- Extra registers between functional units to save data from previous cycle

Multi-Cycle Data Path – Full Picture



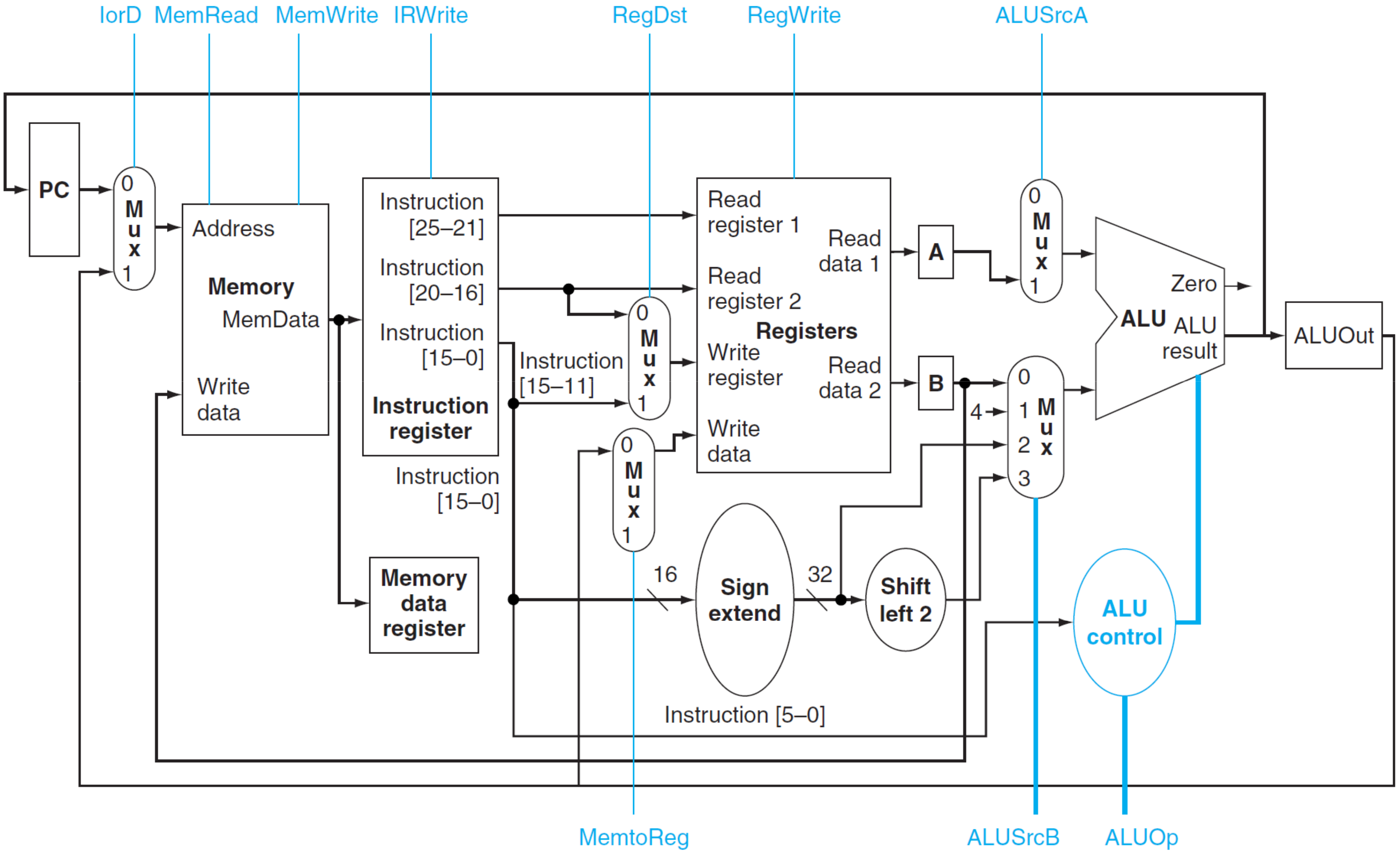
Multi-Cycle Data Path – Full Picture



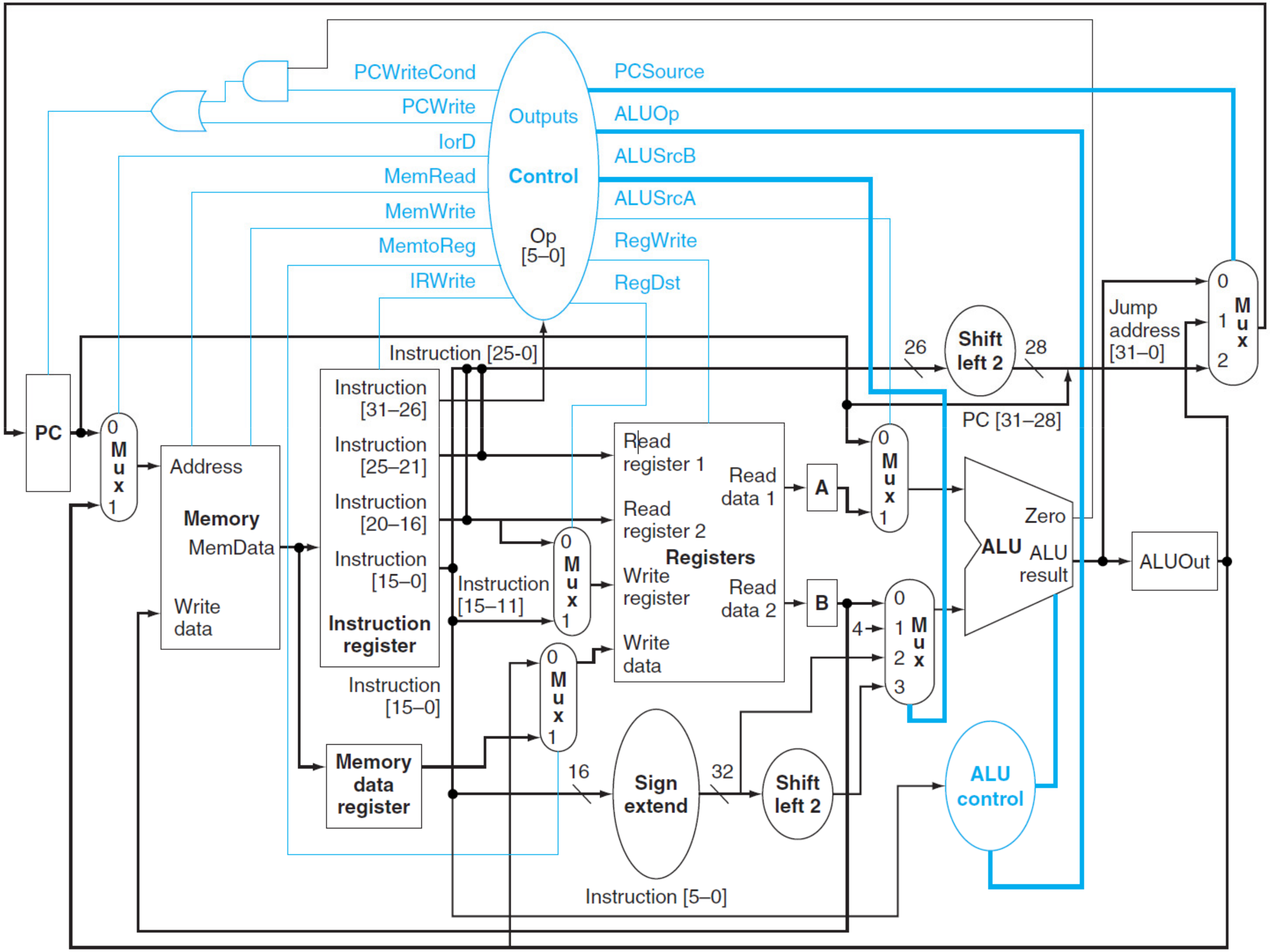
Because several functional units are shared for different purposes, we need both to add new multiplexors and to expand existing ones:

- A new multiplexor for the memory address
- A new multiplexor for the top ALU input
- Expanding the multiplexor on the bottom ALU input into a four-way selector

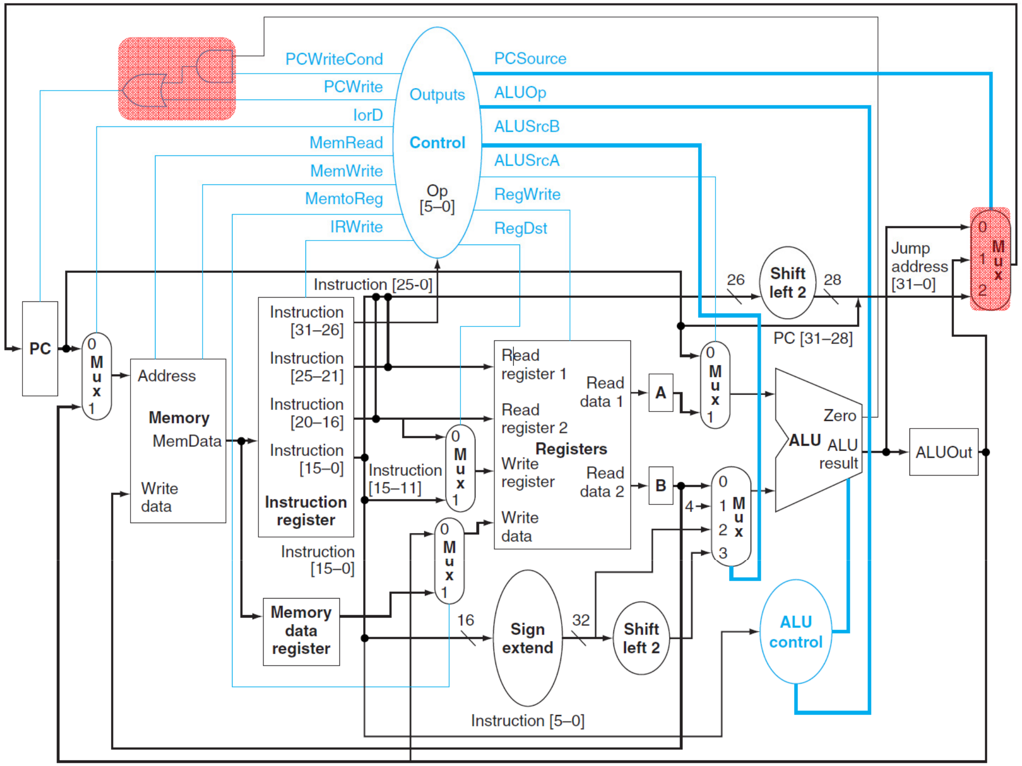
Multi-Cycle Control – Overview



Multi-Cycle Control – Full Picture



Multi-Cycle Control – Full Picture



Control for the multi-cycle implementation (including jump support):

- ALUOp, ALUSrcB and PCSource are 2-bit control lines, all other are 1-bit
- A new multiplexor to select the source of a new PC value
- Extra gates and extra signals PCSource, PCWrite and PCWriteCond, to combine the PC write signals and decide whether a conditional branch should be taken

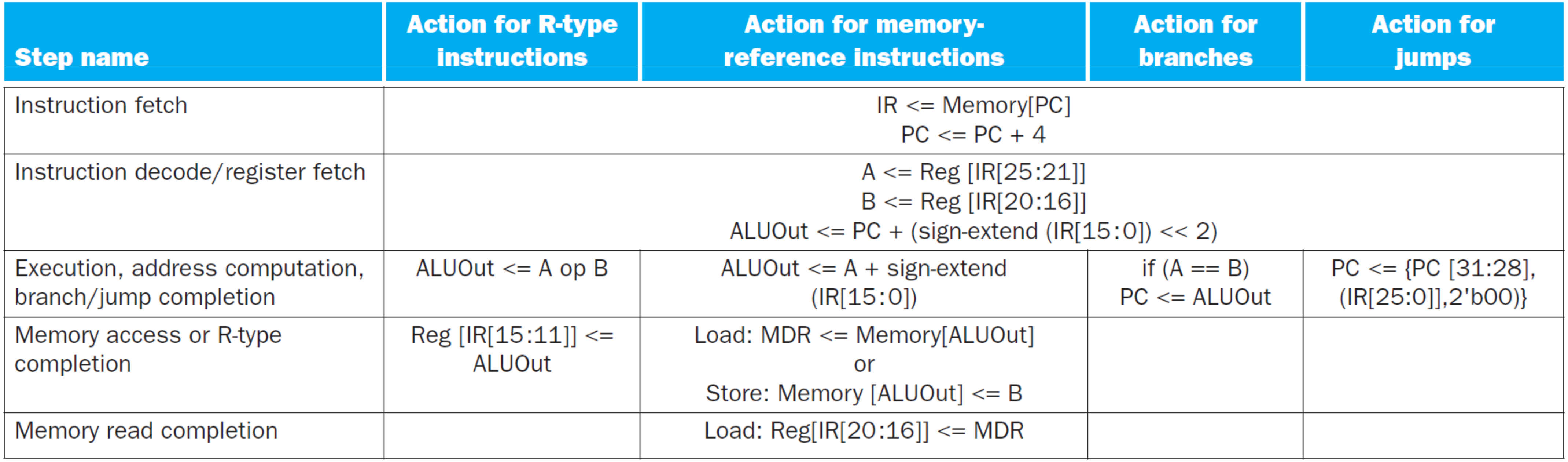
Multi-Cycle Steps

MIPS instructions classically include five steps:

- **IF** – instruction fetch
- **ID** – instruction decode and register fetch
- **EX** – execute operation , address calculation or branch/jump completion
- **MEM** – data memory access or R-type completion
- **WB** – write result back to register

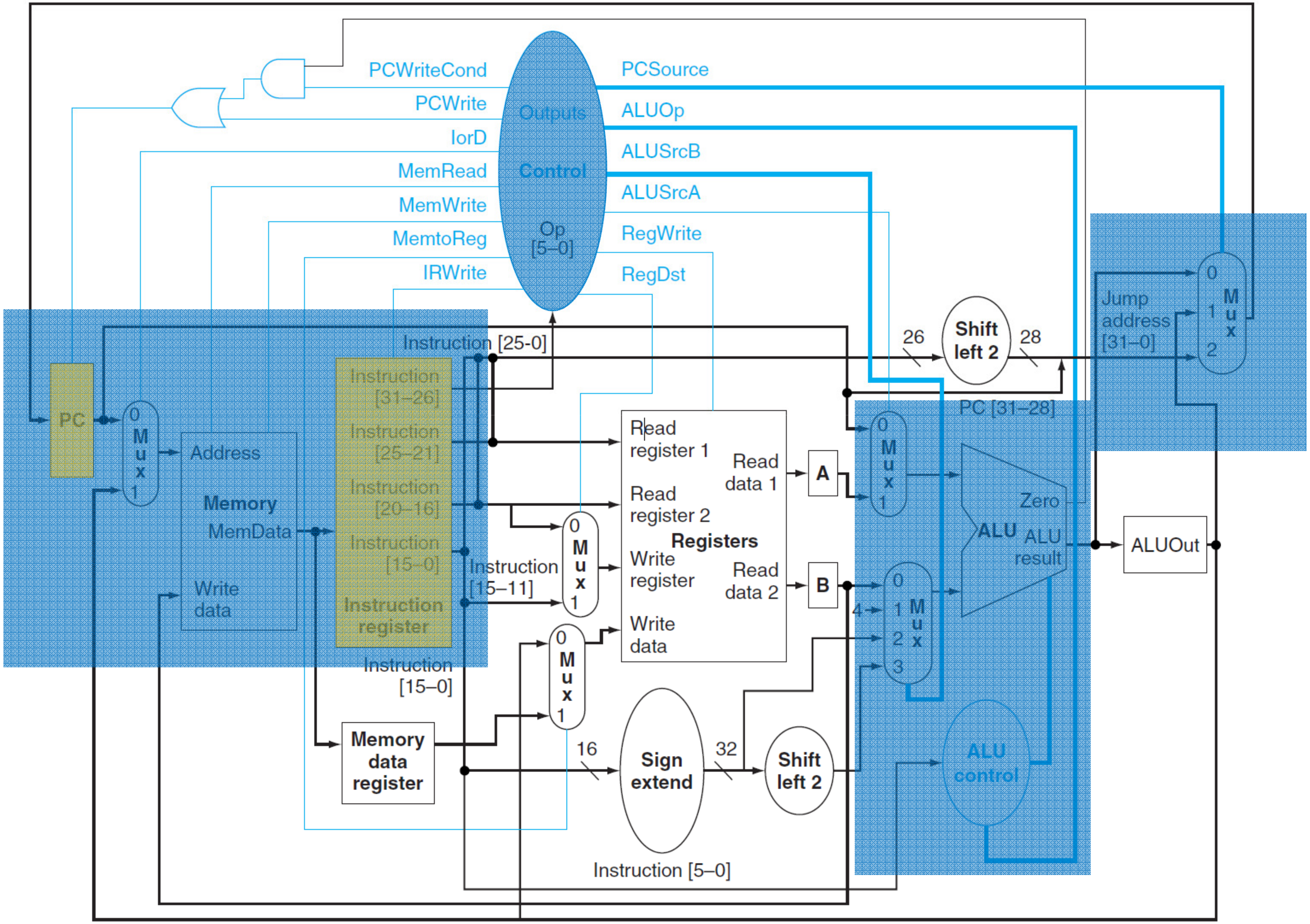
Multi-Cycle Steps

Instructions take from three to five execution steps. The first two steps are independent of the instruction type. After these steps, an instruction takes from one to three more cycles to complete, depending on the instruction type.

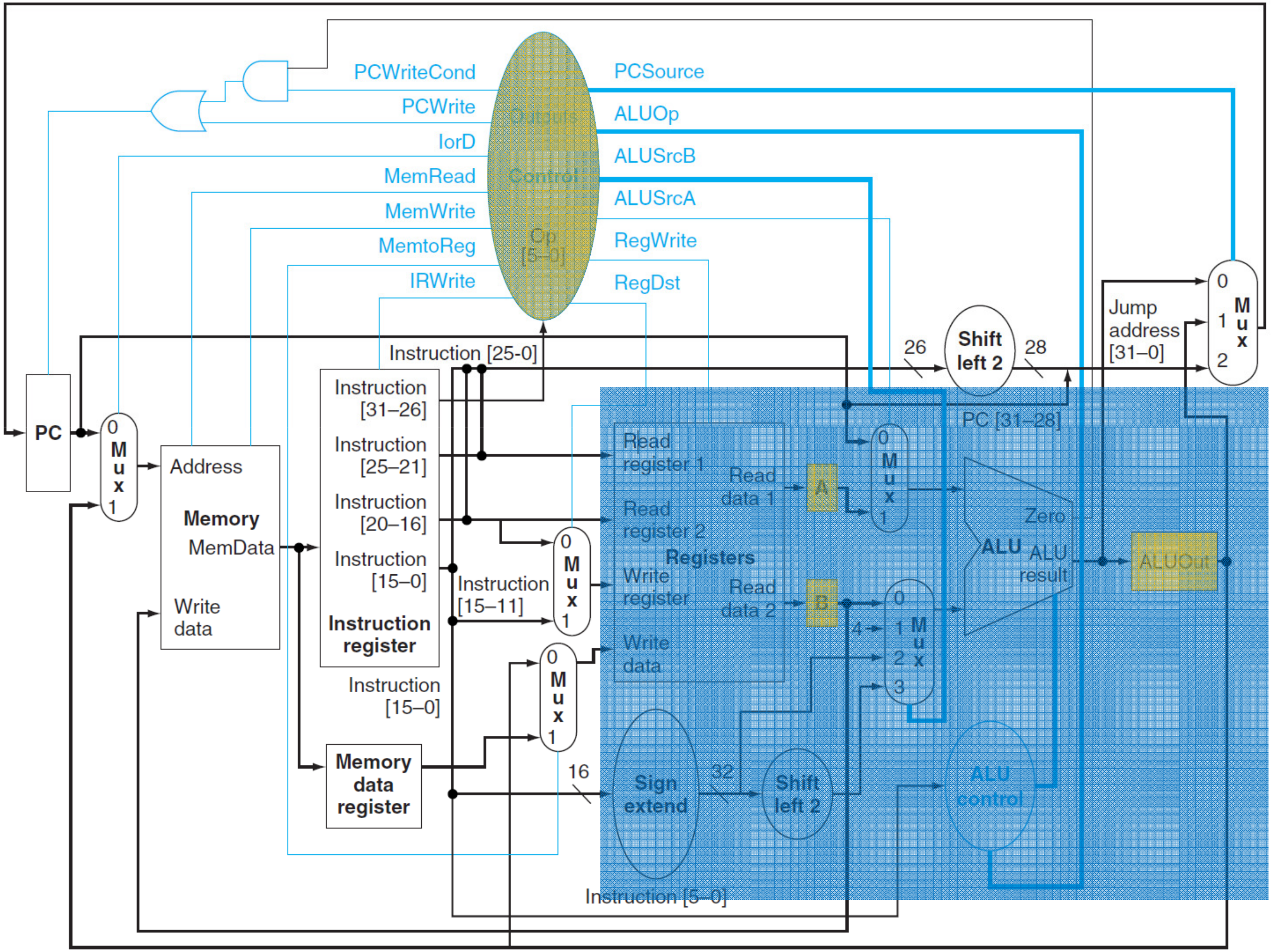


Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR <= Memory[PC] PC <= PC + 4			
Instruction decode/register fetch	A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC + (sign-extend (IR[15:0]) << 2)			
Execution, address computation, branch/jump completion	ALUOut <= A op B	ALUOut <= A + sign-extend (IR[15:0])	if (A == B) PC <= ALUOut	PC <= {PC [31:28], (IR[25:0]],2'b00)}
Memory access or R-type completion	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B		
Memory read completion		Load: Reg[IR[20:16]] <= MDR		

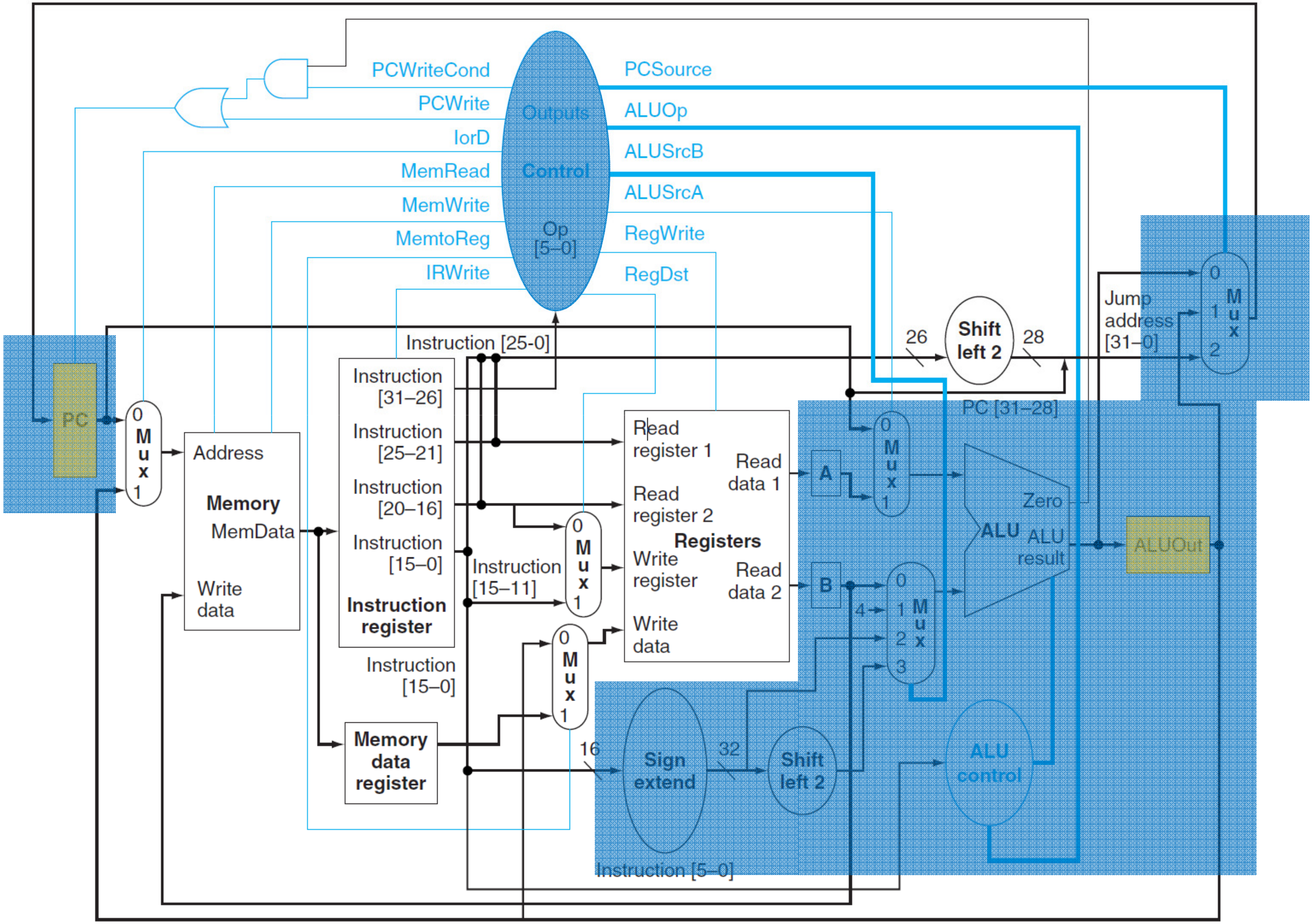
Multi-Cycle Steps – IF



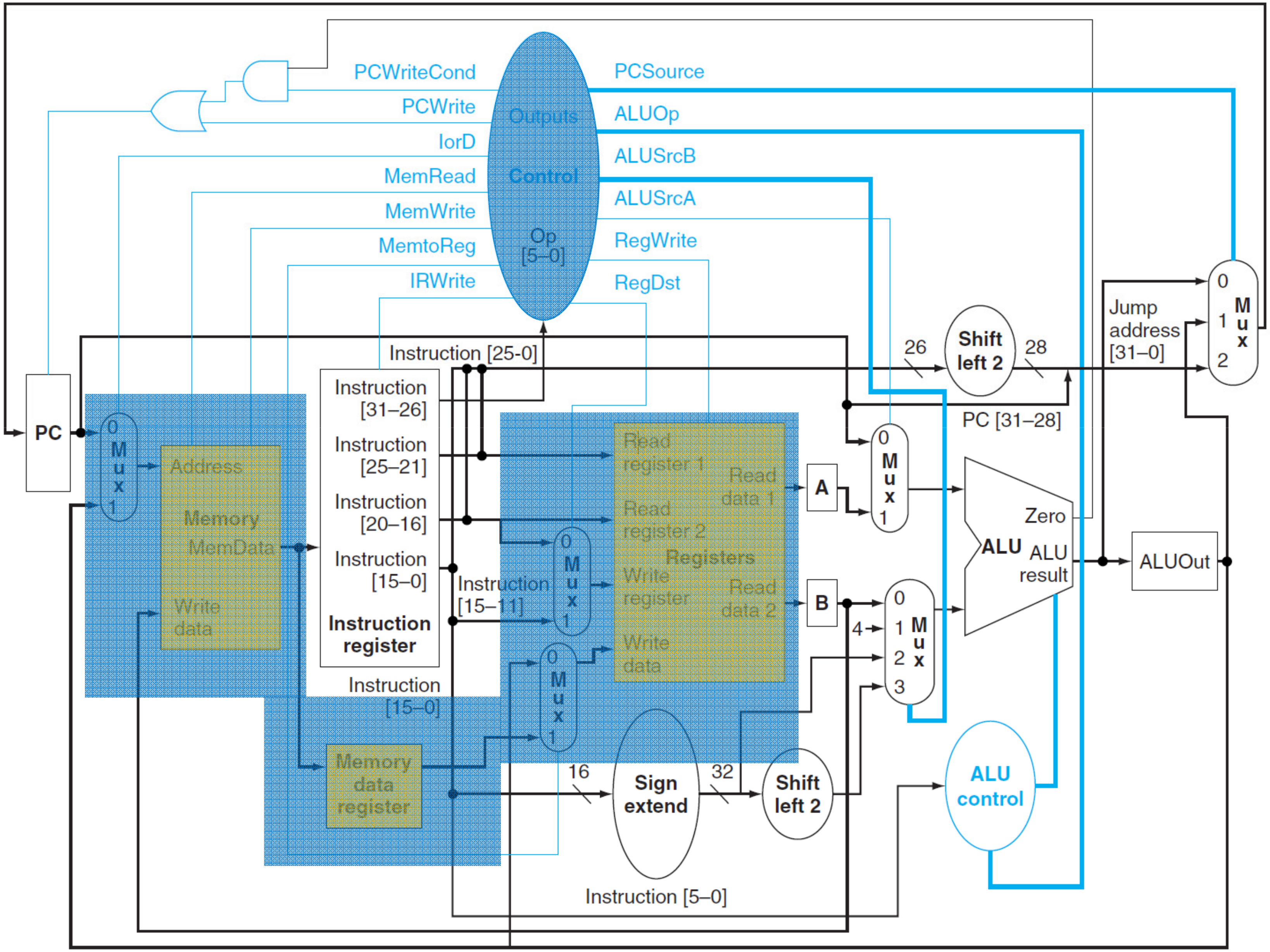
Multi-Cycle Steps – ID



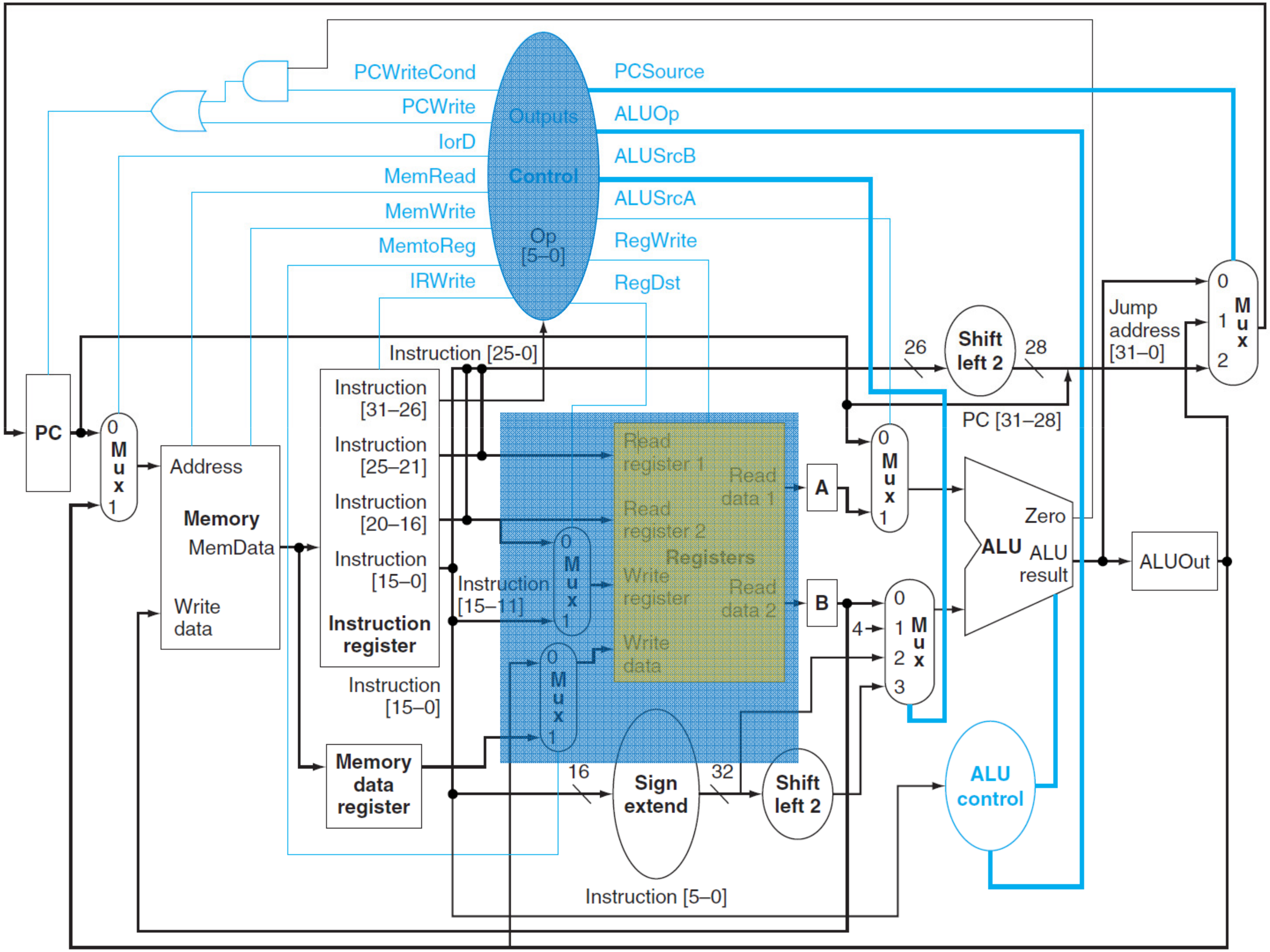
Multi-Cycle Steps – EX



Multi-Cycle Steps – MEM



Multi-Cycle Steps – WB

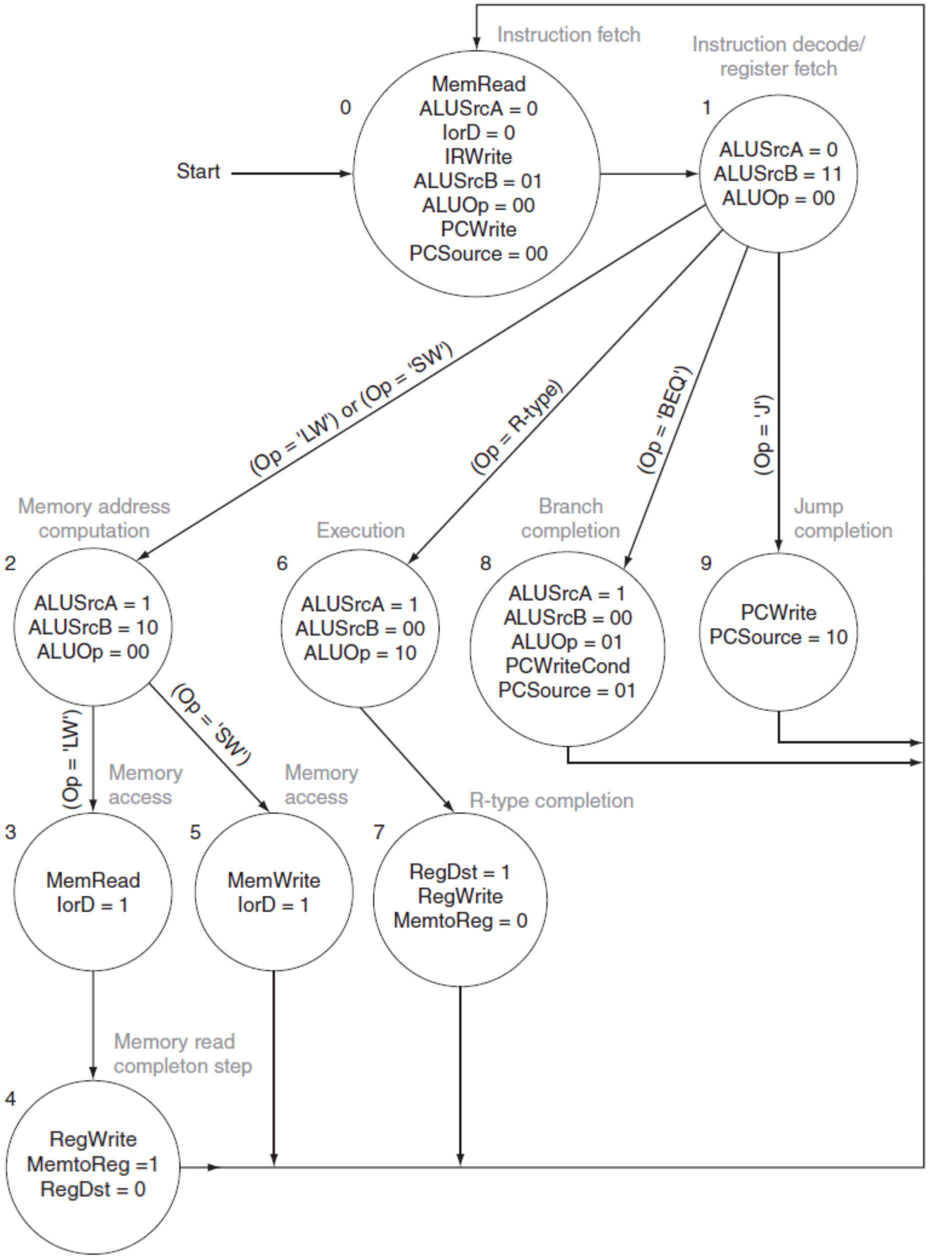


Multi-Cycle Control – Finite State Machine

For the multi-cycle data path, the control is more complex because the instructions are executed in a series of steps. A common technique to specify the control is to use a **finite state machine**.

A finite state machine consists of a set of states and directions on how to change states. The directions are defined by a **next-state function**, which maps the current state and the inputs to a new state. Each state also specifies a **set of output signals that are asserted** when the machine is in that state.

Multi-Cycle Control – Finite State Machine



Multi-Cycle Control – Finite State Machine

Example: **lw $t2, 100($t1)** (I-format: op rs rt address --> **35 $t1 $t2 100**)

Step IF – machine state 0

- Signals ALUSrcA=0 / ALUSrcB=01 / ALUOp=00 / PCSource=00 / PCWrite
  PC <= PC + 4
- Signals IorD=0 / MemRead / IRWrite
  IR <= Memory[PC] = **35 $t1 $t2 100**

Step ID – machine state 1

- Signals ALUSrcA=0 / ALUSrcB=11 / ALUOp=00
  Ignored for load instructions
- A <= Reg[IR[25:21]] = **Reg[$t1]**
  B <= Reg[IR[20:16]] = **Reg[$t2]**

Multi-Cycle Control – Finite State Machine

Example: **lw $t2, 100($t1)** (I-format: op rs rt address --> **35 $t1 $t2 100**)

Step EX – machine state 2

- Signals ALUSrcA=1 / ALUSrcB=10 / ALUOp=00
  ALUOut <= A + sign-extend(IR[15:0]) = **Reg[$t1] + 100**

Step MEM – machine state 3

- Signals IorD=1 / MemRead
  MDR <= Memory[ALUOut] = **Memory[Reg[$t1] + 100]**

Step WB – machine state 4

- Signals RegDst=0 / RegWrite / MemtoReg=1
  Reg[IR[20:16]] = **Reg[$t2]** <= MDR = **Memory[Reg[$t1] + 100]**

Pipelining

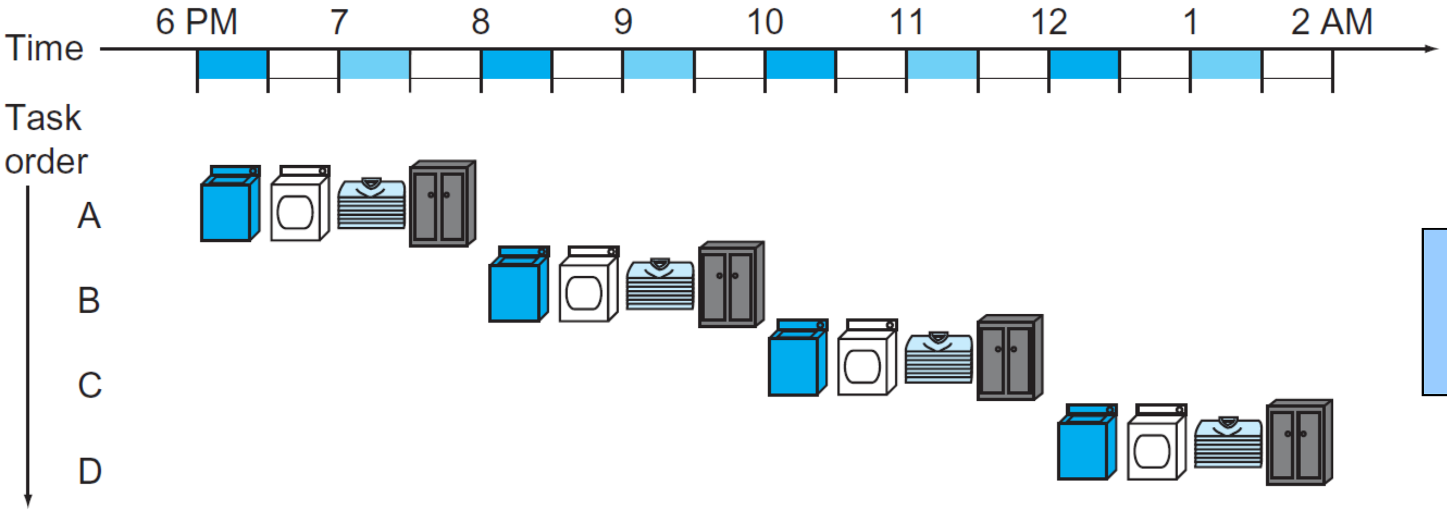
Pipelining is an implementation technique in which **multiple instructions are overlapped in execution**.

As in the case of a multi-cycle implementation:

- Instructions are executed in steps and each step takes one clock cycle
- Different instructions may take a different number of steps to execute
- The pipelined clock cycle is determined by the worst-case step

As opposed to decreasing the average instruction execution time, pipelining improves performance by **increasing instruction throughput**.

Pipelining



Execution time: 8h

Pipelining

Pipelining **exploits the potential parallelism among instructions**. This parallelism is often called **instruction-level parallelism (ILP)**.

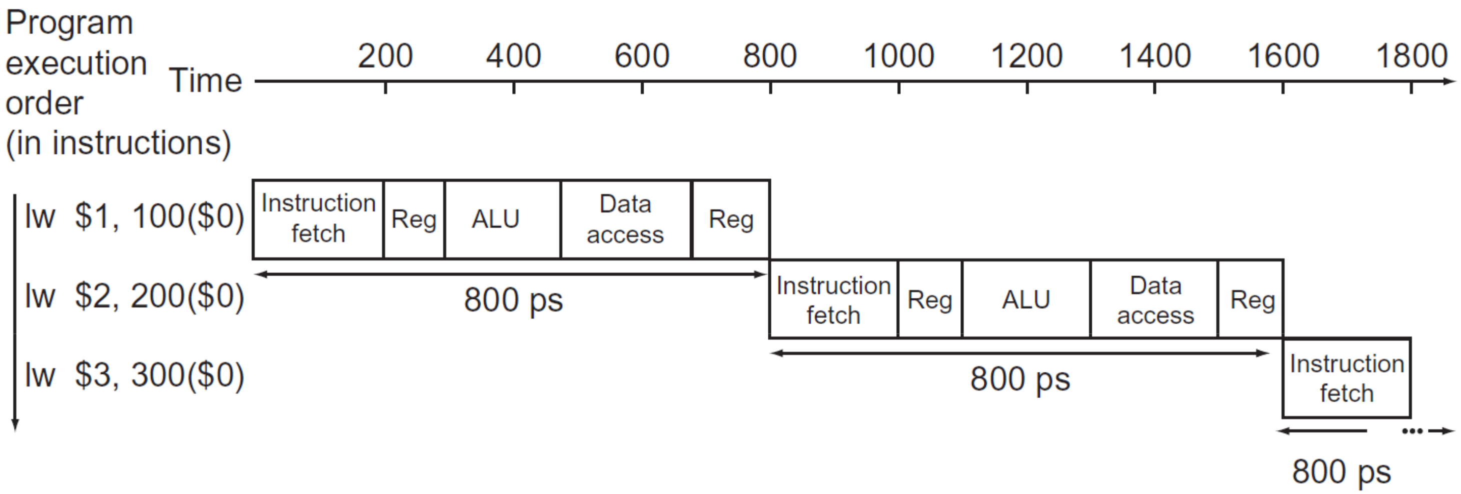
There are two methods for increasing the potential amount of ILP:

- Increase the number of steps (depth) of the pipeline to overlap more instructions (and potentially reduce the clock cycle)
- Replicate the functional units of the computer so that it can launch multiple instructions in every pipeline stage

In what follows, we will not consider the case of using several functional units per stage.

Pipelining

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



Clock cycle: 800ps

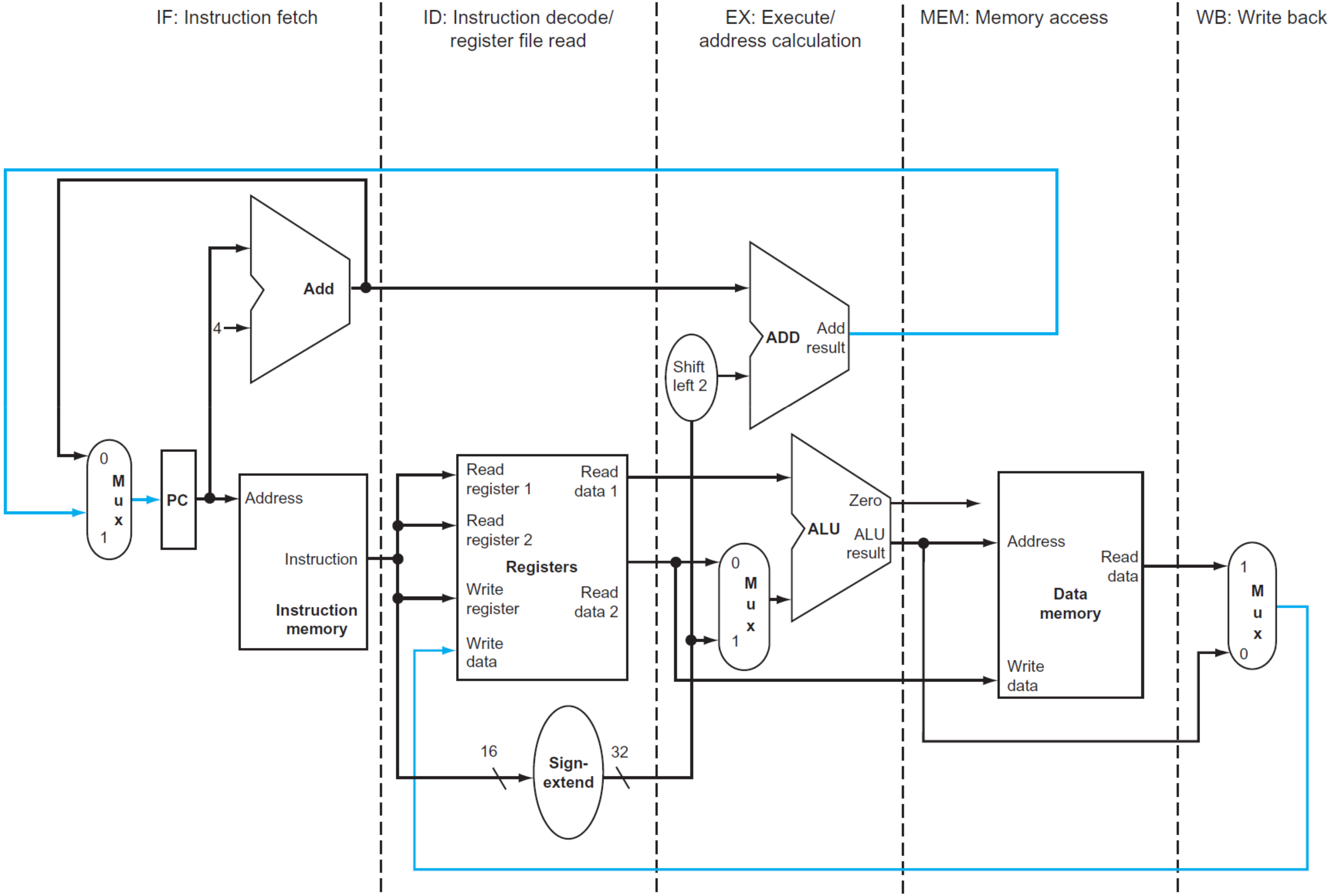
Pipeline Steps

In MIPS, the same five steps are considered for pipelining:

- **IF** – instruction fetch
- **ID** – instruction decode and register fetch
- **EX** – execute operation , address calculation or branch/jump completion
- **MEM** – data memory access or R-type completion
- **WB** – write result back to register

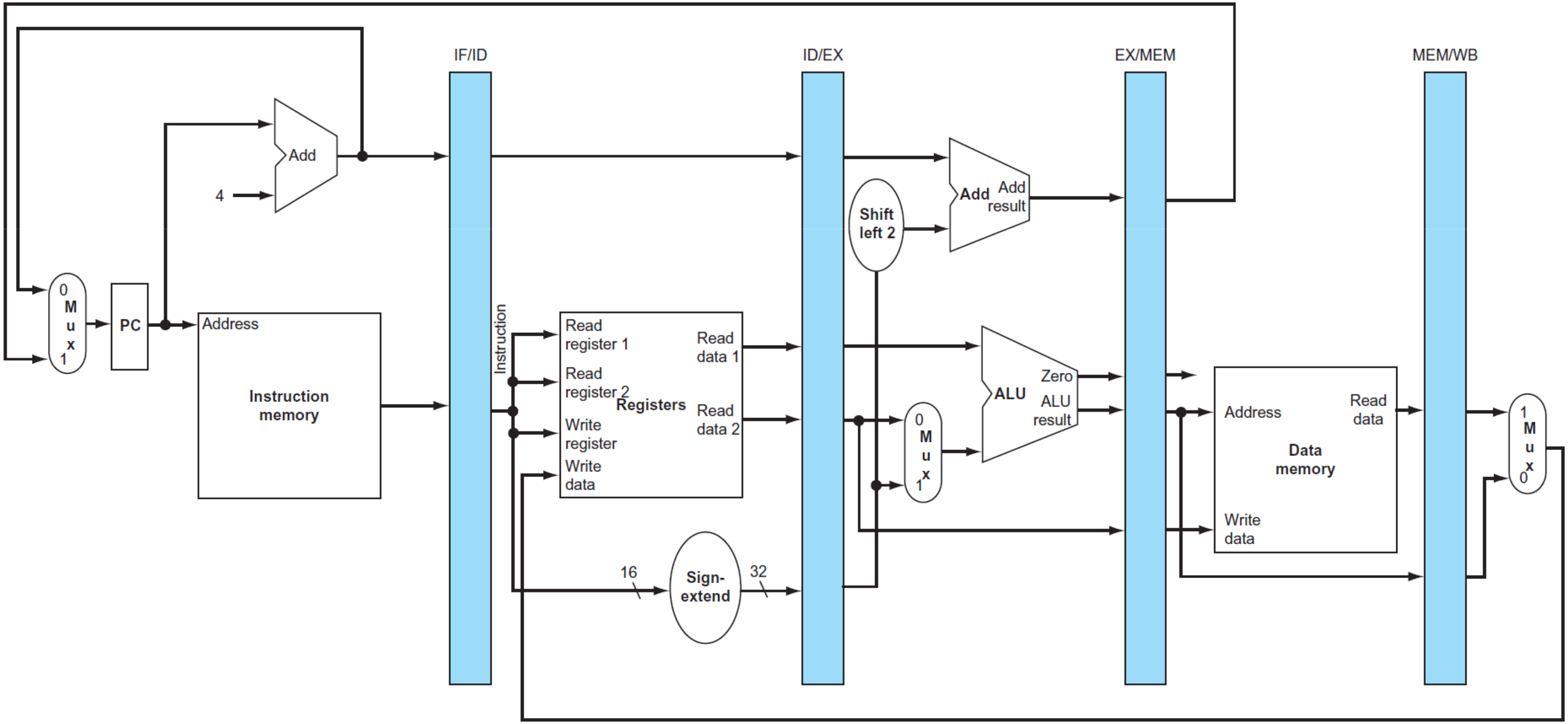
These five steps correspond roughly to the way the data path is drawn – instructions and data move generally from left to right through the five stages as they complete execution.

Pipelined Data Path



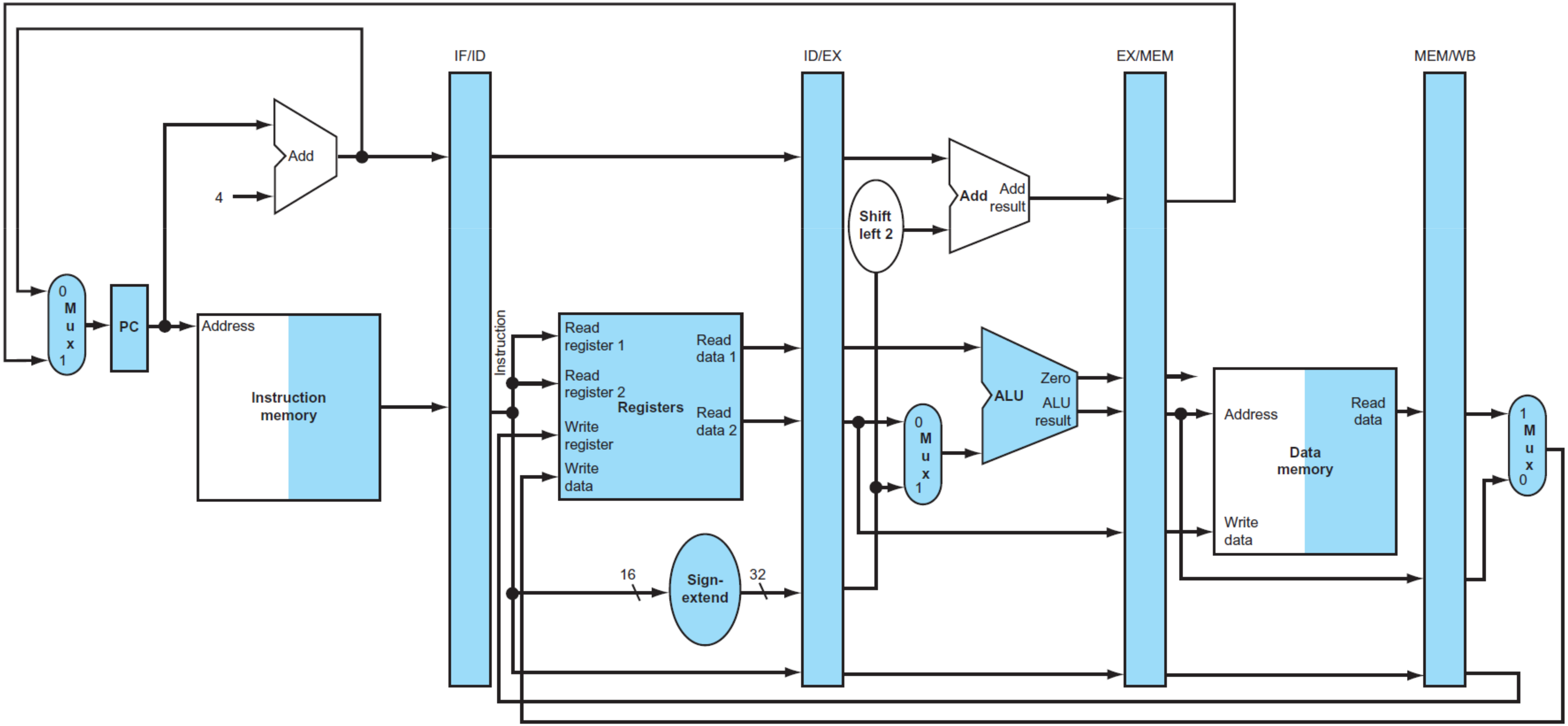
Pipeline Registers

Pipelined data path **allows sharing functional units** but that requires **extra registers between stages to save data from the previous cycle**.

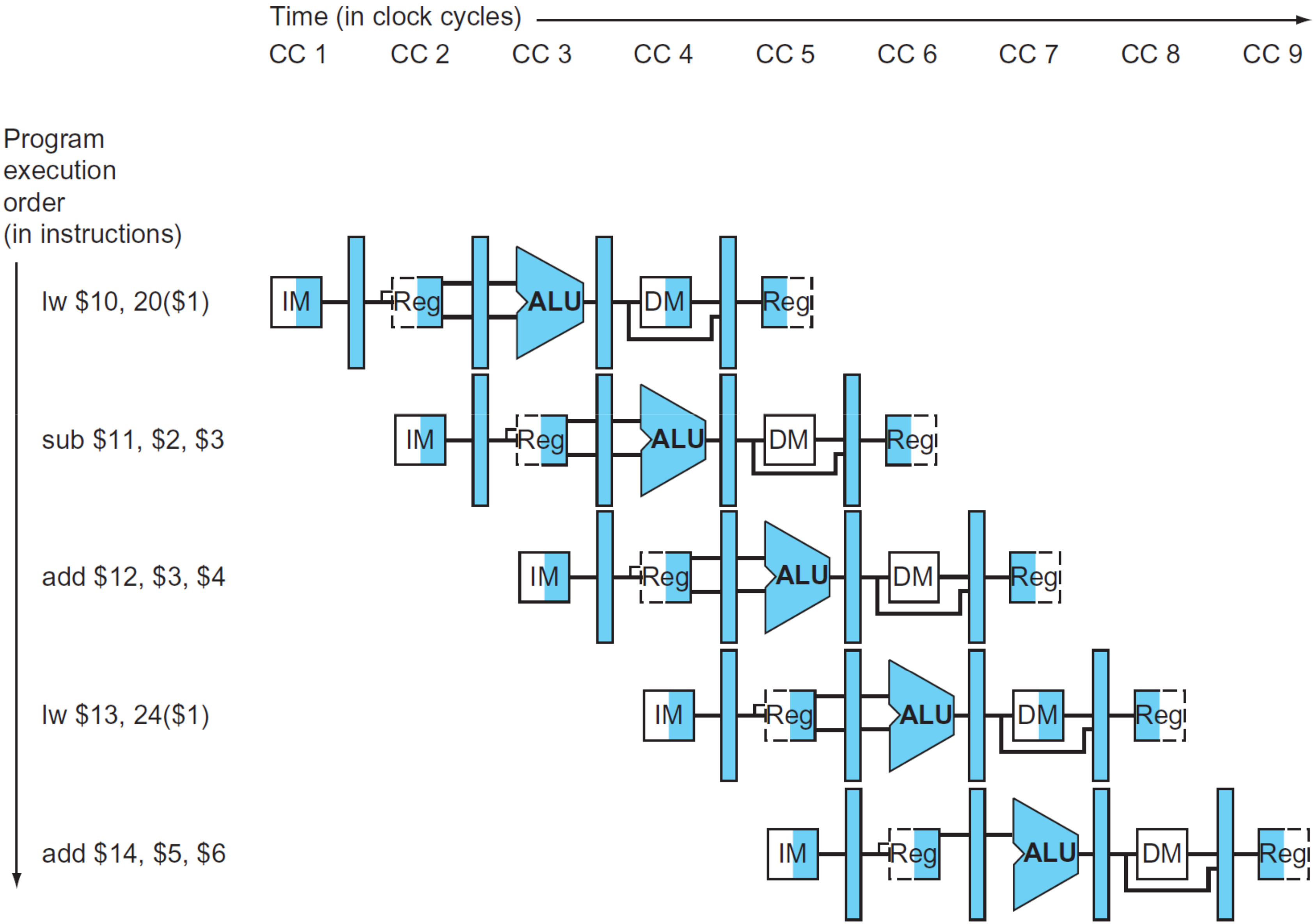


Pipeline Registers – Load Instructions

We need to **preserve the destination register**, which is only used during the WB step, by **passing it from the ID/EX stage to the MEM/WB stage**.



Pipelined Data Path – Multiple Instructions



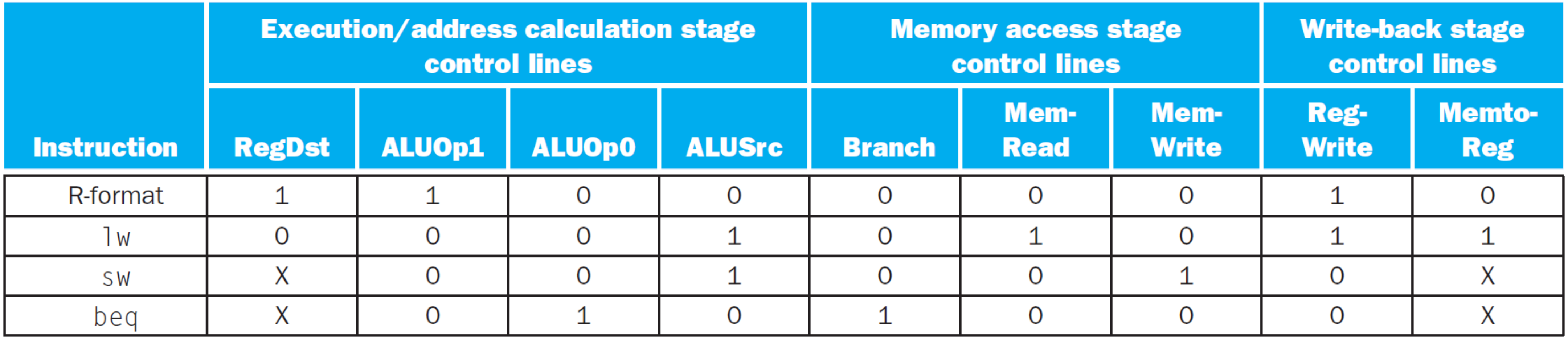
Pipelined Data Path – Instruction Throughput

Trying to allow some instructions to take fewer cycles (i.e., use less pipe stages) does not help, since the **instruction throughput is determined by the clock cycle**. The number of pipe stages per instruction affects latency, not throughput.

Instead of trying to make instructions take fewer cycles, we should **explore making the pipeline longer**, so that instructions take more cycles, but the cycles are shorter. This could improve performance.

Pipeline Control Lines

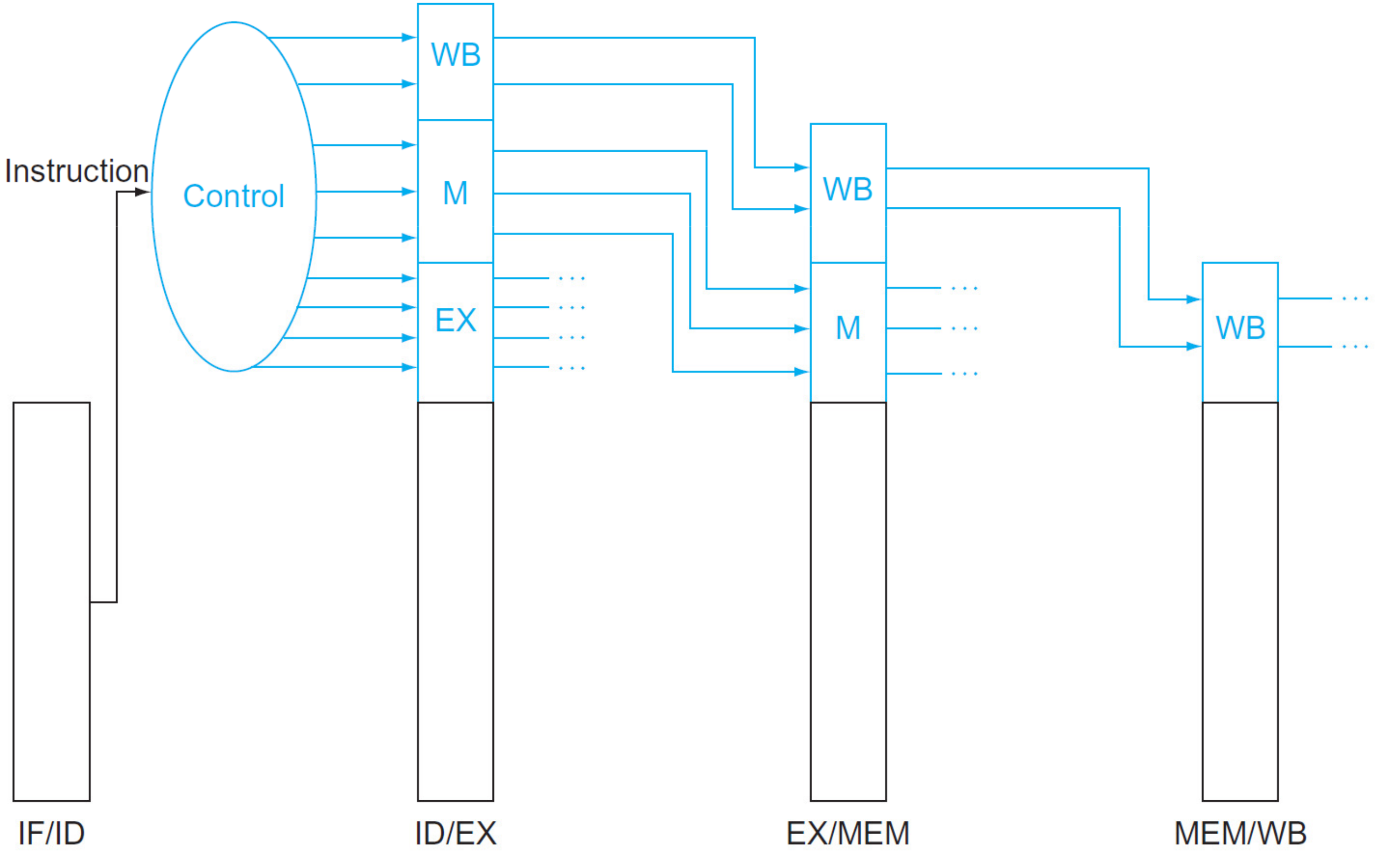
Because each control line is associated with an element active in only a single pipeline stage, we can divide the control lines into three groups according to the pipeline stage (no control is required for the IF and ID stages).



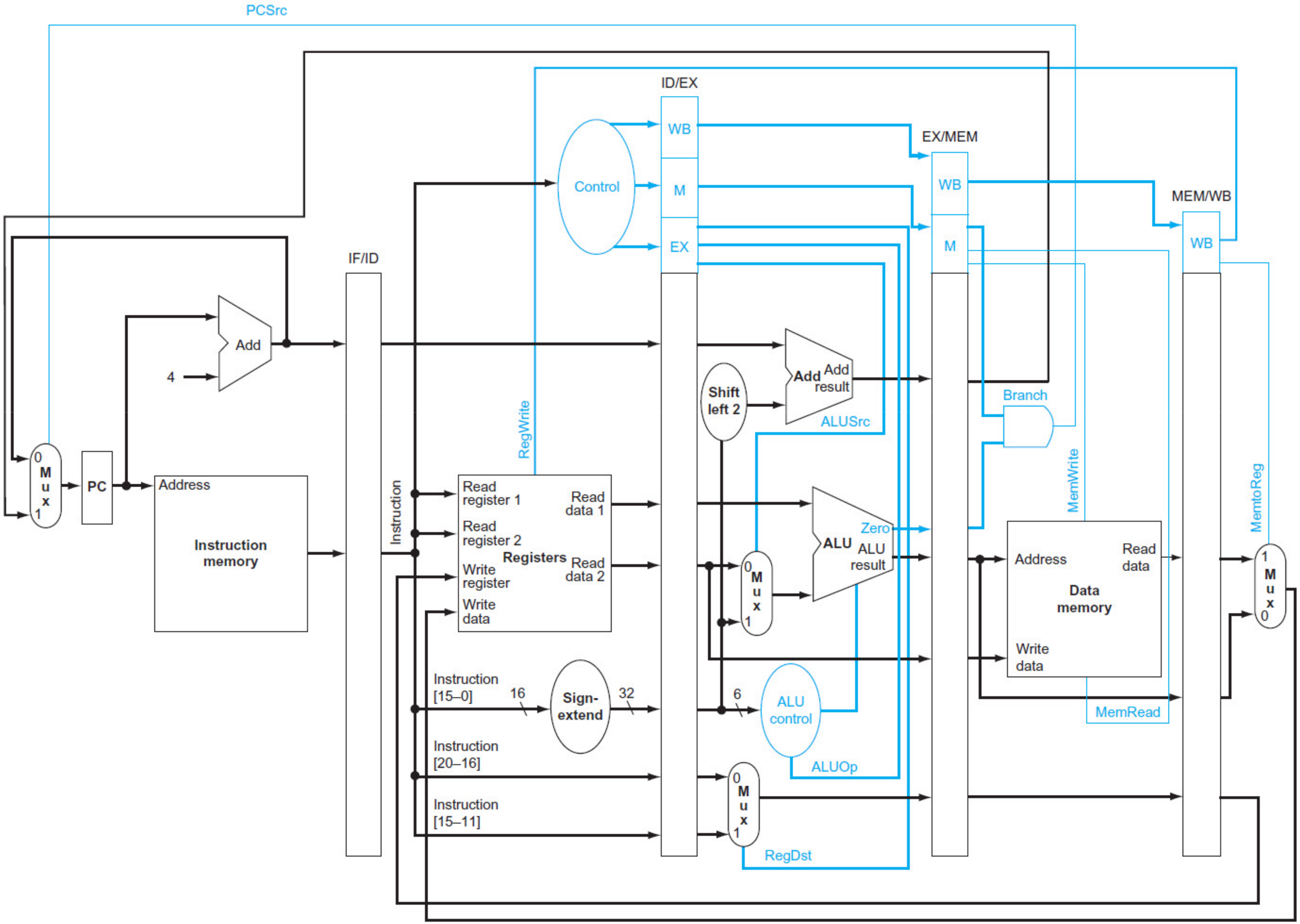
Instruction	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

Pipeline Control Lines

Implementing control means setting the nine control lines to these values in each stage for each instruction. The simplest way to do this is to extend the pipeline registers to include control information.



Pipelined Data Path and Control – Full Picture



Pipeline Hazards

In pipelining, there are situations when the next instruction cannot execute in the following clock cycle:

- **Structural hazard** occurs when the hardware does not support the combination of instructions that are set to execute (e.g., sharing of functional units)
- **Data hazard** occurs when data that is needed to execute the instruction is not yet available (e.g., dependence of instructions)
- **Control hazard** occurs when the instruction that was fetched is not the one that is needed, i.e., the flow of execution is not what the pipeline expected

Without intervention, a pipeline hazard could **severely stall the pipeline**. Although the compiler relies upon the hardware to resolve hazards and ensure correct execution, the compiler must understand the pipeline to achieve the best performance and avoid unexpected stalls.

Data Hazards – Dependence of Instructions

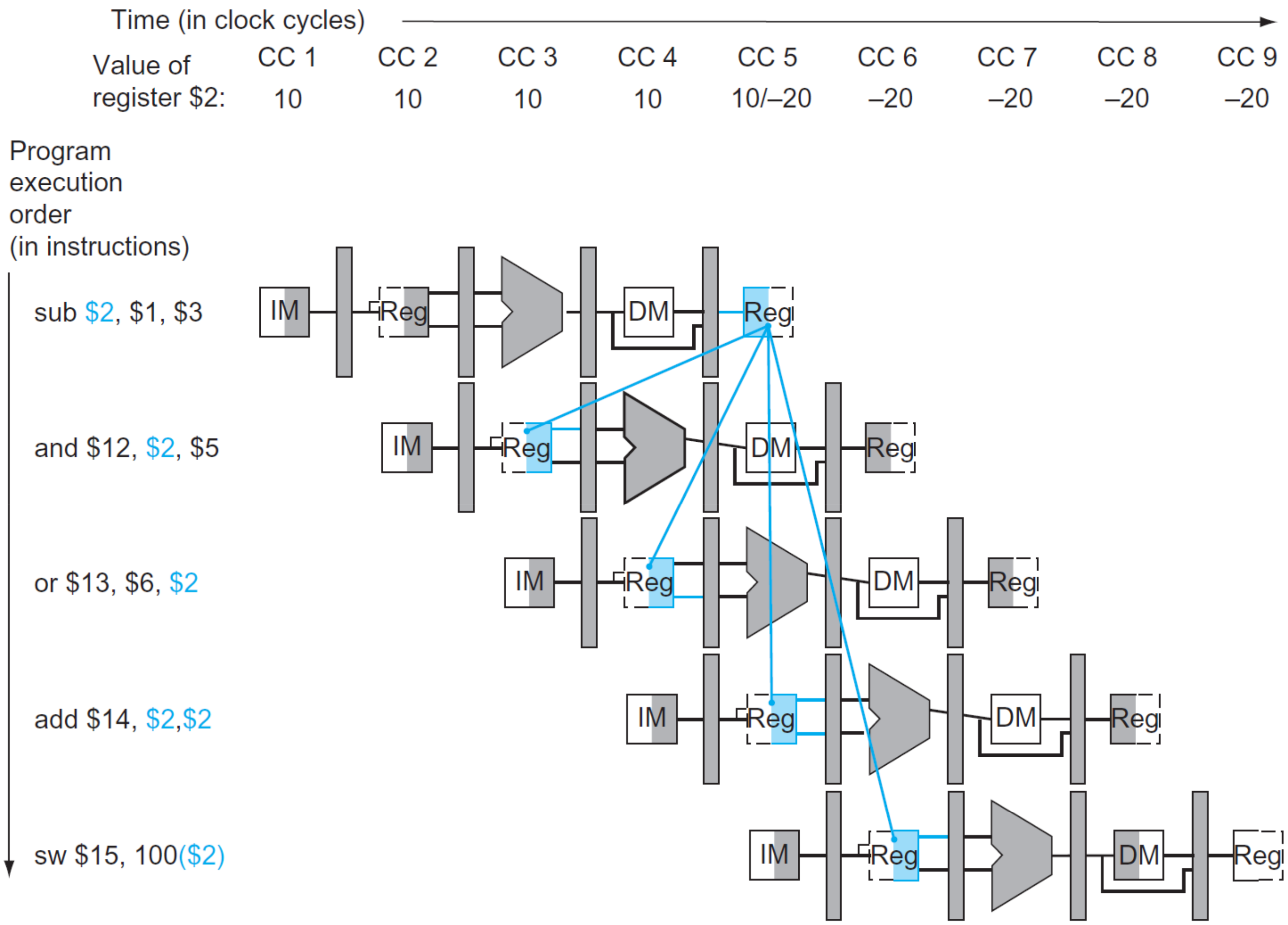


FIGURE 4.52 Pipelined dependences in a five-instruction sequence using simplified datapaths to show the dependences. All the dependent actions are shown in color, and "CC 1" at the top of the figure means clock cycle 1. The first instruction writes into $2, and all the following instructions read $2. This register is written in clock cycle 5, so the proper value is unavailable before clock cycle 5. (A read of a register during a clock cycle returns the value written at the end of the first half of the cycle, when such a write occurs.) The colored lines from the top datapath to the lower ones show the dependences. Those that must go backward in time are *pipeline data hazards*.

Data Hazards – Dependence of Instructions

When an instruction tries to use a register in its EX stage that an earlier instruction intends to write in its WB stage, we actually need the values as inputs to the ALU (**hazard condition**).

The hazard conditions are:

1a) EX/MEM.RegisterRd = ID/EX.RegisterRs

1b) EX/MEM.RegisterRd = ID/EX.RegisterRt

2a) MEM/WB.RegisterRd = ID/EX.RegisterRs

2b) MEM/WB.RegisterRd = ID/EX.RegisterRt

Data Hazards – Dependence of Instructions

Consider again the previous sequence:

```
sub $2, $1, $3          # register $2 set by sub
and $12, $2, $5         # 1st operand ($2) set by sub
or $13, $6, $2          # 2nd operand ($2) set by sub
add $14, $2, $2         # 1st and 2nd operands ($2) set by sub
sw $15, 100($2)         # index ($2) set by sub
```

The dependences are:

- The **sub-and is a type 1a hazard** (EX/MEM.RegisterRd = ID/EX.RegisterRs)
- The **sub-or is a type 2b hazard** (MEM/WB.RegisterRd = ID/EX.RegisterRt)
- The two dependences on **sub-add are not hazards** (the register file supplies the proper data during the ID stage of add) and there is **no data hazard between sub-sw** (sw reads $2 the clock cycle after sub writes $2)

Data Hazards – Dependence of Instructions

Time (in clock cycles)	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register $2:	10	10	10	10	10/–20	–20	–20	–20	–20
Value of EX/MEM:	X	X	X	–20	X	X	X	X	X
Value of MEM/WB:	X	X	X	X	–20	X	X	X	X

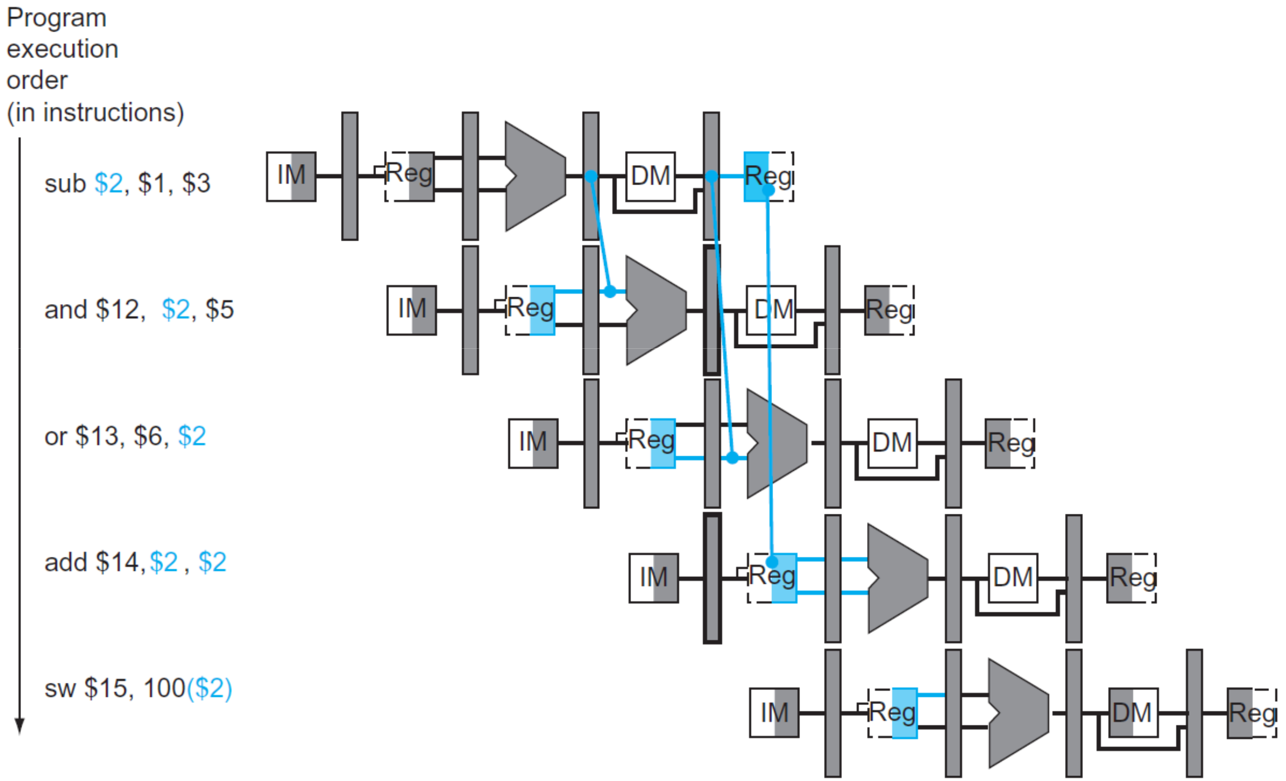


FIGURE 4.53 The dependences between the pipeline registers move forward in time, so it is possible to supply the inputs to the ALU needed by the AND instruction and OR instruction by forwarding the results found in the pipeline registers. The values in the pipeline registers show that the desired value is available before it is written into the register file. We assume that the register file forwards values that are read and written during the same clock cycle, so the add does not stall, but the values come from the register file instead of a pipeline register. Register file "forwarding"—that is, the read gets the value of the write in that clock cycle—is why clock cycle 5 shows register $2 having the value 10 at the beginning and –20 at the end of the clock cycle. As in the rest of this section, we handle all forwarding except for the value to be stored by a store instruction.

Data Hazards – Forwarding

Forwarding (or bypassing) is a method of resolving a data hazard by **retrieving the missing data element from extra connections/buffers** rather than waiting for it to arrive from registers or memory.

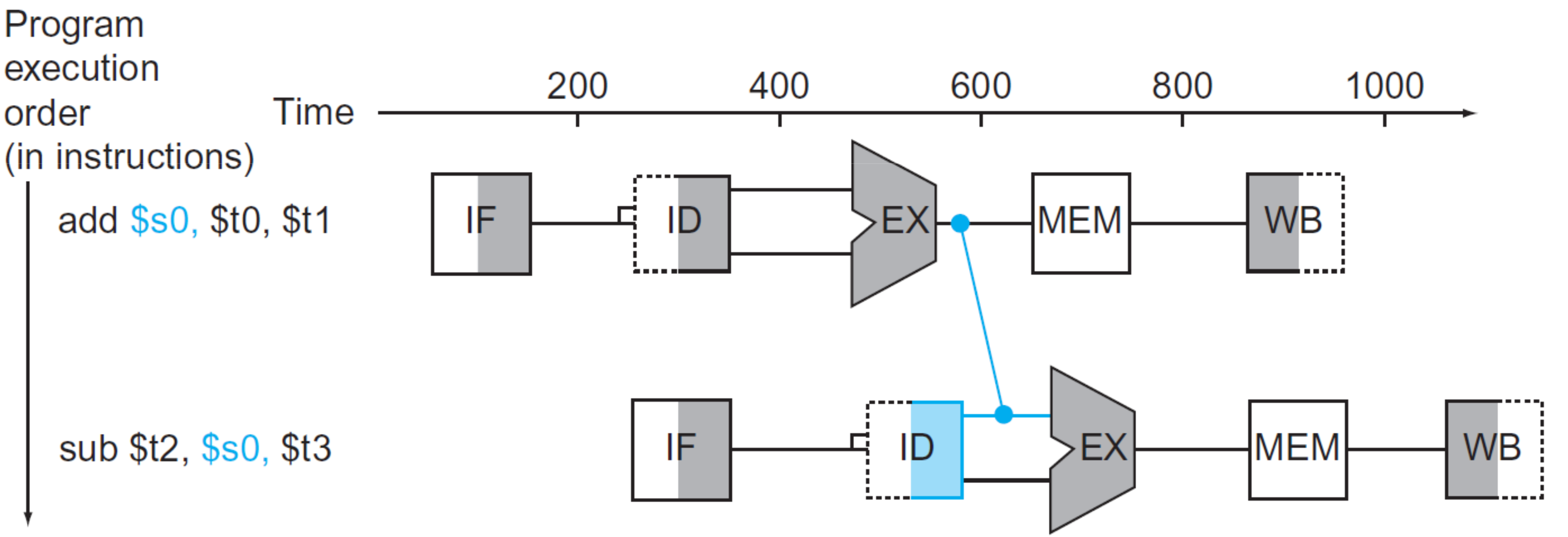
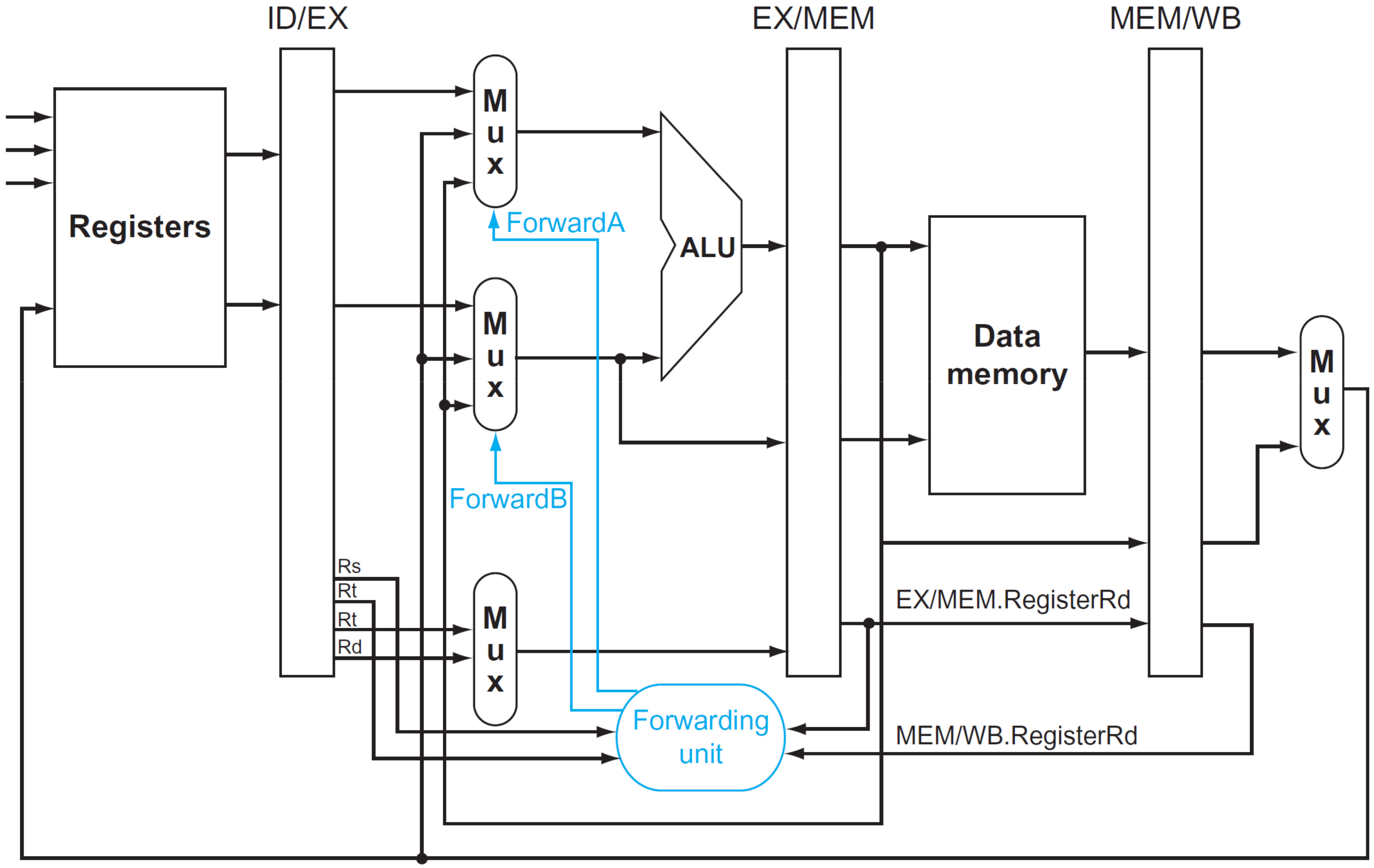


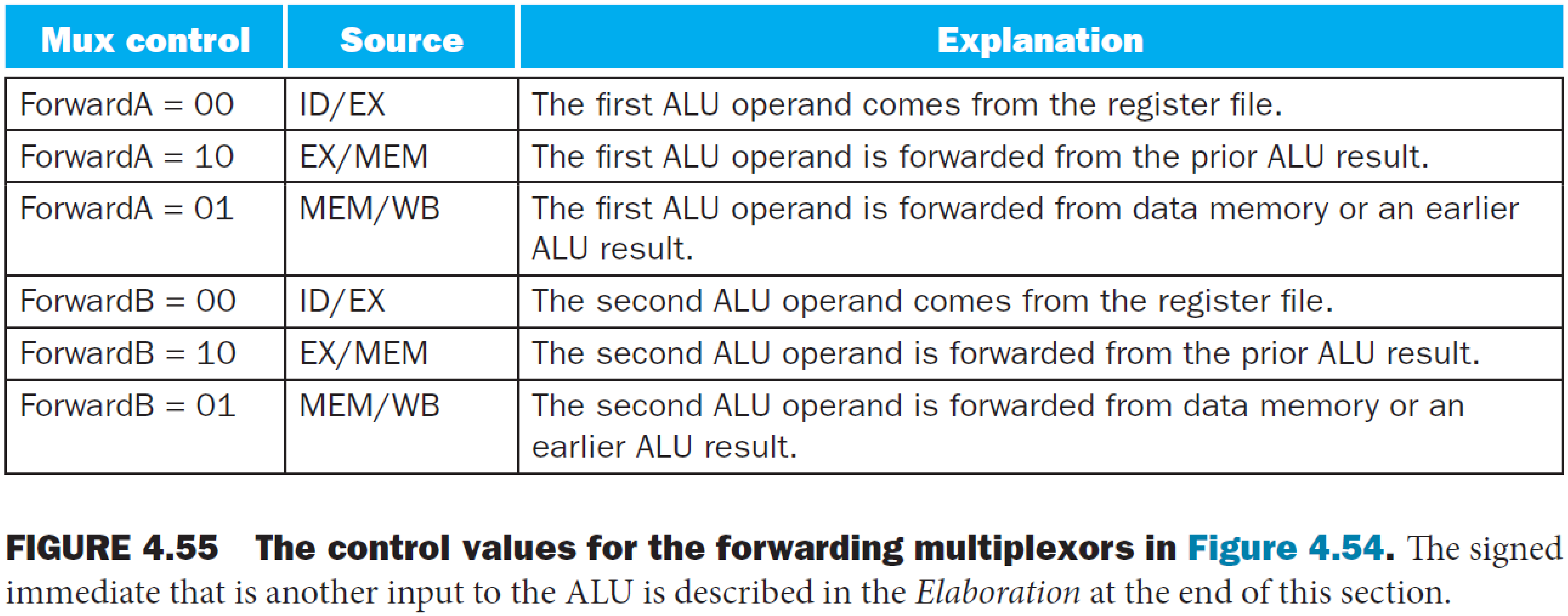
FIGURE 4.29 Graphical representation of forwarding. The connection shows the forwarding path from the output of the EX stage of add to the input of the EX stage for sub, replacing the value from register $s0 read in the second stage of sub.

Data Hazards – Forwarding Unit



Data Hazards – Forwarding Control Lines

The control values for the forwarding control lines (multiplexors) are:

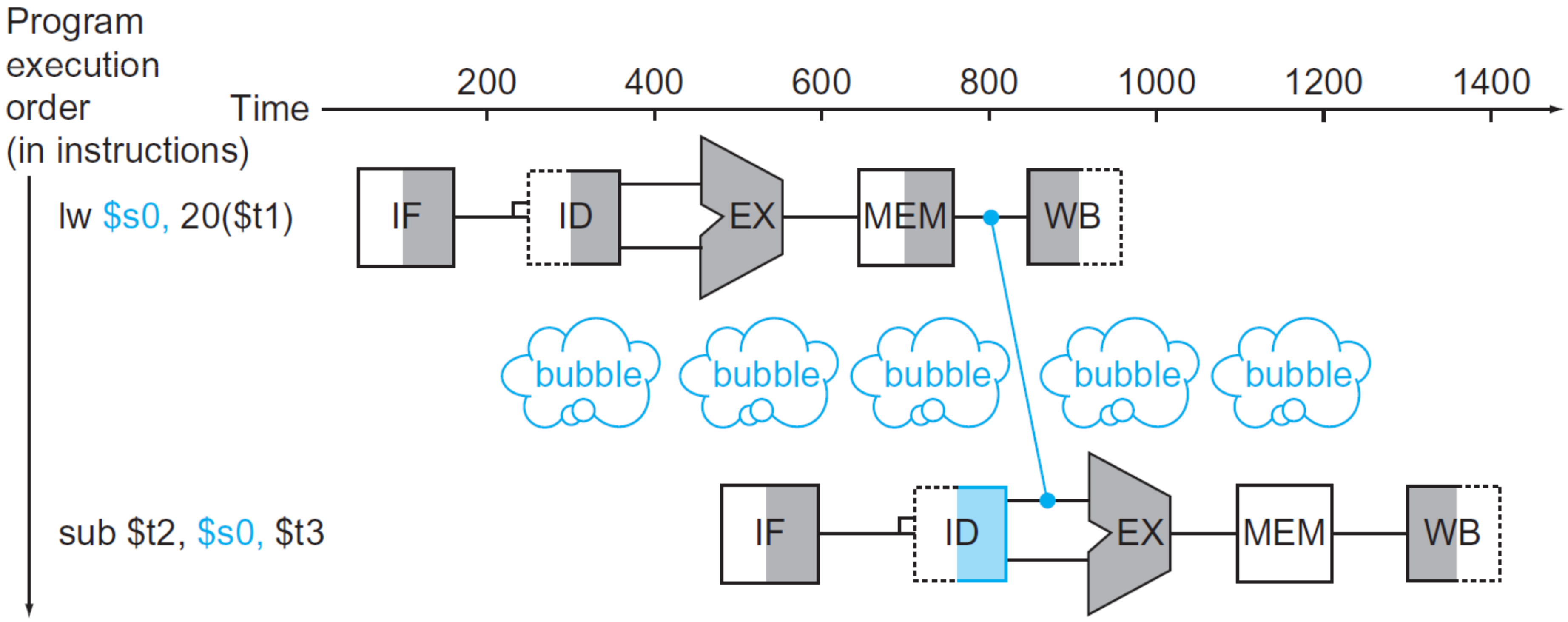


Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

FIGURE 4.55 The control values for the forwarding multiplexors in Figure 4.54. The signed immediate that is another input to the ALU is described in the *Elaboration* at the end of this section.

Data Hazards – Stalls

Sometimes we must **stall the pipeline** for the **combination of load followed by an instruction that reads its result** and, in addition to a forwarding unit, we need a **hazard detection unit** operating during the ID stage so that it can **insert the stall between the load and its use**.



Data Hazards – Stalls

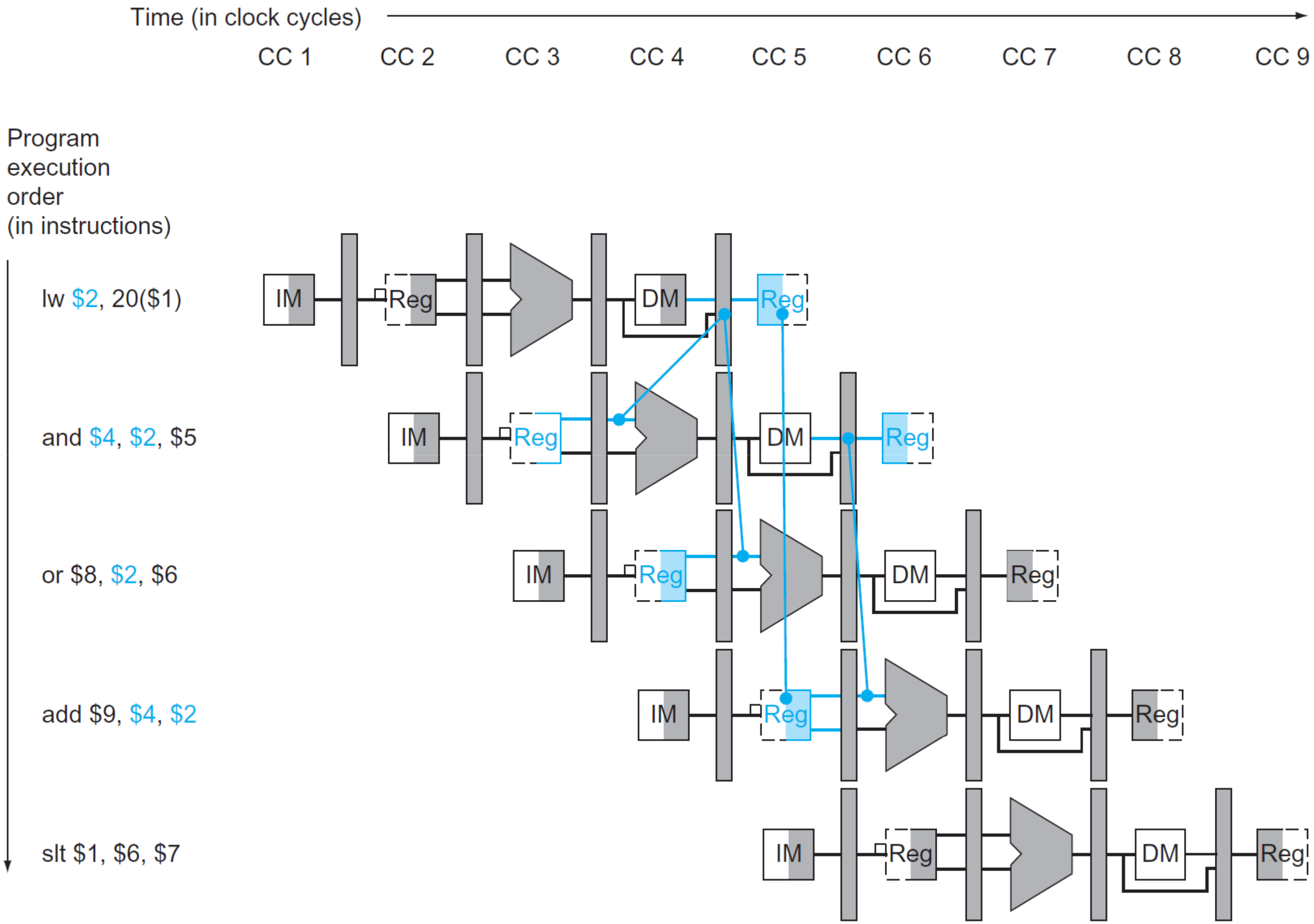


FIGURE 4.58 A pipelined sequence of instructions. Since the dependence between the load and the following instruction (and) goes backward in time, this hazard cannot be solved by forwarding. Hence, this combination must result in a stall by the hazard detection unit.

Data Hazards – Stalls

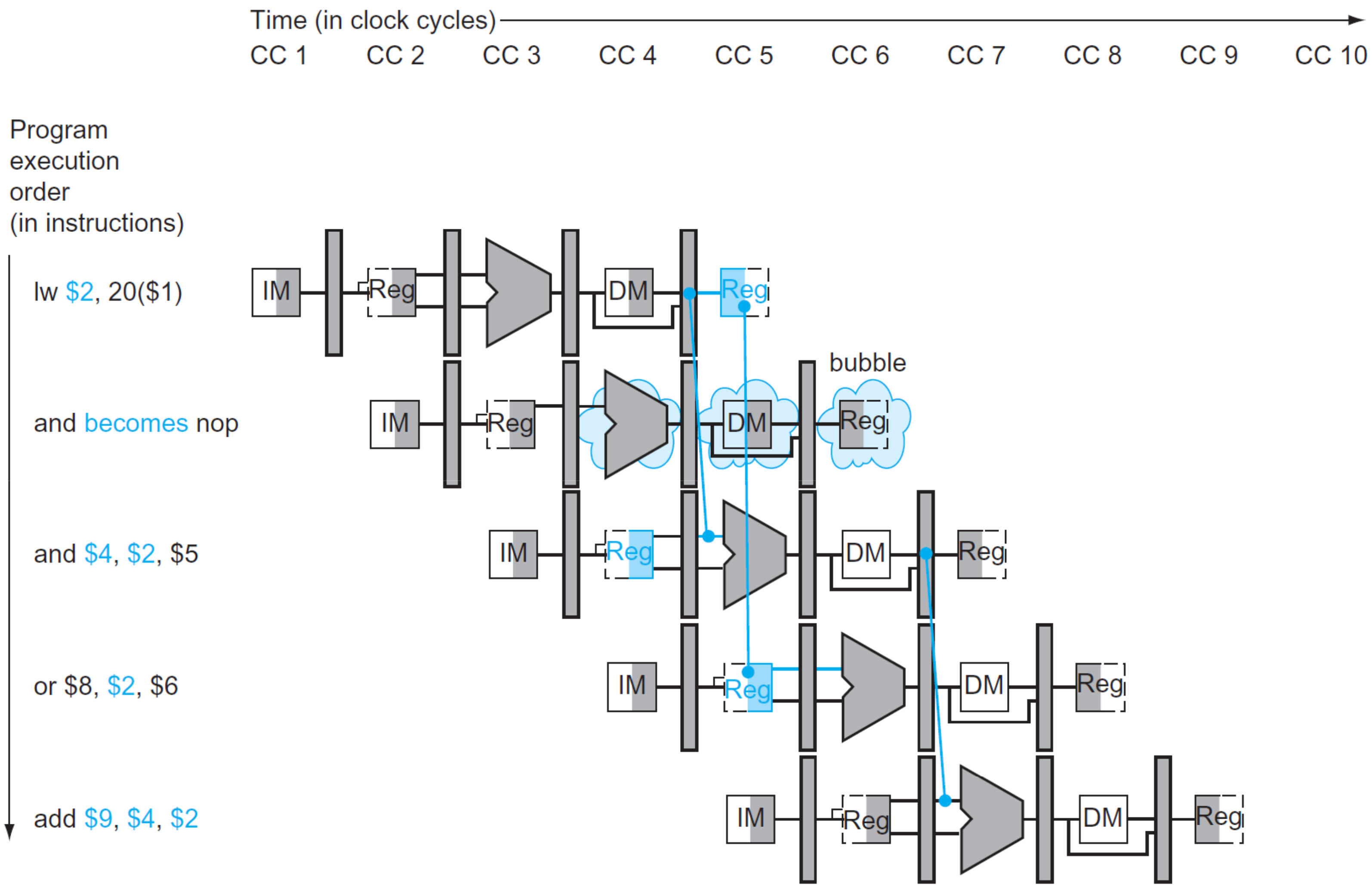


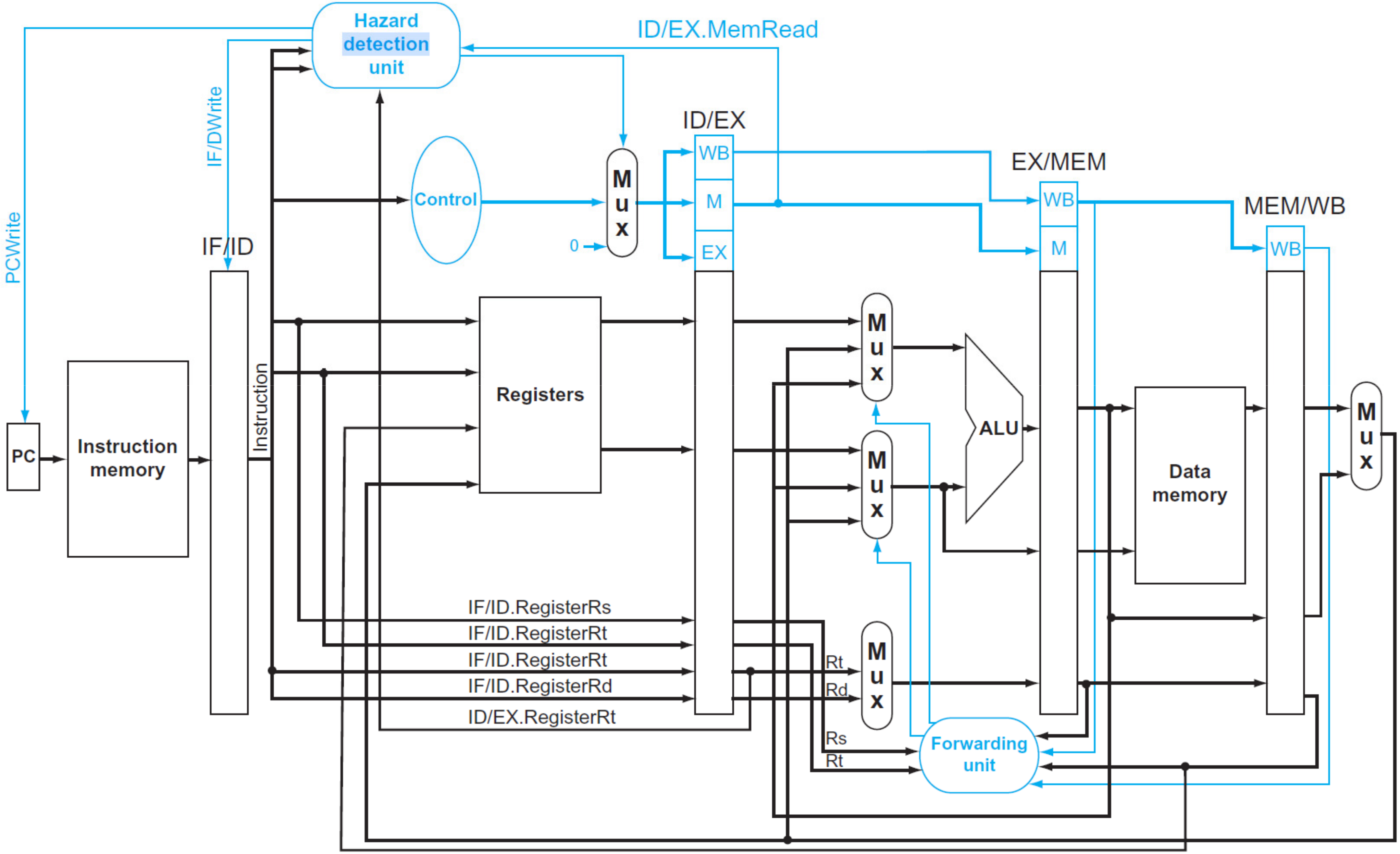
FIGURE 4.59 The way stalls are really inserted into the pipeline. A bubble is inserted beginning in clock cycle 4, by changing the `and` instruction to a nop. Note that the `and` instruction is really fetched and decoded in clock cycles 2 and 3, but its EX stage is delayed until clock cycle 5 (versus the unstalled position in clock cycle 4). Likewise the `OR` instruction is fetched in clock cycle 3, but its ID stage is delayed until clock cycle 5 (versus the unstalled clock cycle 4 position). After insertion of the bubble, all the dependences go forward in time and no further hazards occur.

Data Hazards – Detection Unit

The hazard detection unit must check if a load is in execution and if the destination register of the load in the EX stage matches either source register of the instruction in the ID stage:

```
if (ID/EX.MemRead and
   ((ID/EX.RegisterRt = IF/ID.RegisterRs) or (ID/EX.RegisterRt = IF/ID.RegisterRt)))
      stall the pipeline one clock cycle
```

Data Hazards – Detection Unit



Data Hazards – Code Reordering

Code reordering is a complementary method of resolving data hazards stalls. Consider the following C code and its corresponding MIPS code:

```
a[3] = a[0] + a[1];
a[4] = a[0] + a[2];
```

original code

```
         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)
```

code reordering

```
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

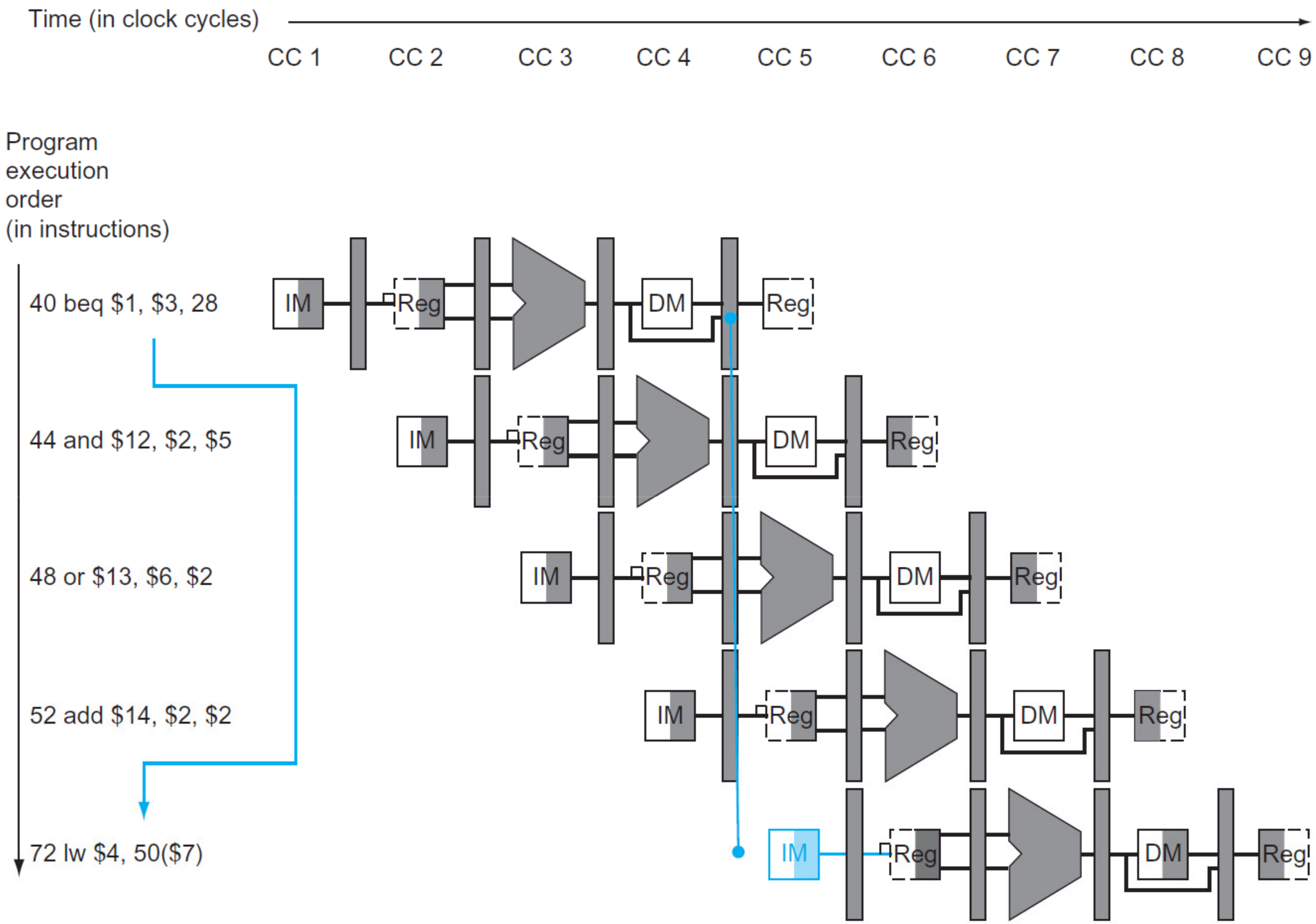
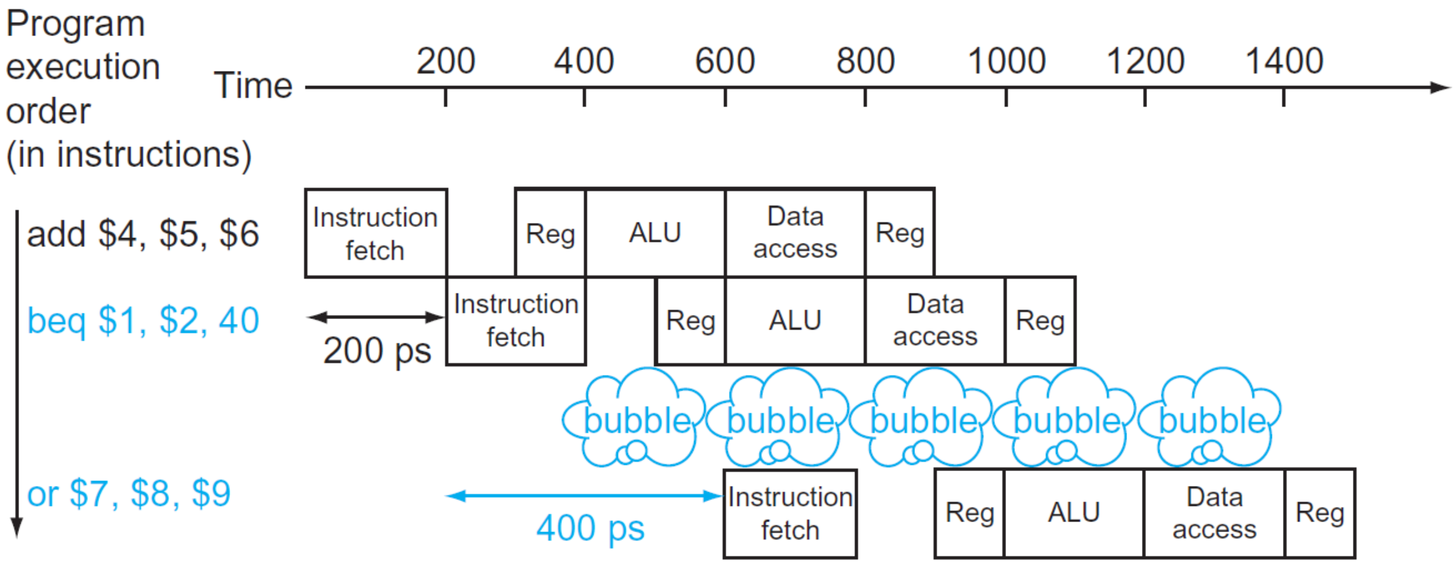


FIGURE 4.61 The impact of the pipeline on the branch instruction. The numbers to the left of the instruction (40, 44, …) are the addresses of the instructions. Since the branch instruction decides whether to branch in the MEM stage—clock cycle 4 for the beq instruction above—the three sequential instructions that follow the branch will be fetched and begin execution. Without intervention, those three following instructions will begin execution before beq branches to lw at location 72. (Figure 4.31 assumed extra hardware to reduce the control hazard to one clock cycle; this figure uses the nonoptimized datapath.)

Control Hazards – Stall

In the case of a branch instruction, we need to make a decision regarding the **next instruction to fetch on the very next clock cycle**. Nevertheless, the pipeline cannot possibly know what the next instruction should be. 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.



Control Hazards – Branch Prediction

Branch prediction is a method of resolving a control hazard that **assumes a given outcome for the branch and proceeds from that assumption** rather than waiting to be certain of the actual outcome. **Only stall when prediction is wrong**.

If branches are untaken half the time, and if it costs little to discard the instructions, this optimization halves the cost of control hazards.

To stall, we follow a similar approach as we did for a load-use data hazard. The difference is that here we must **discard the three instructions in the IF, ID and EX stages when the branch reaches the MEM stage** (for load-use, we just discard the instruction in the ID stage).

Control Hazards – Branch Prediction

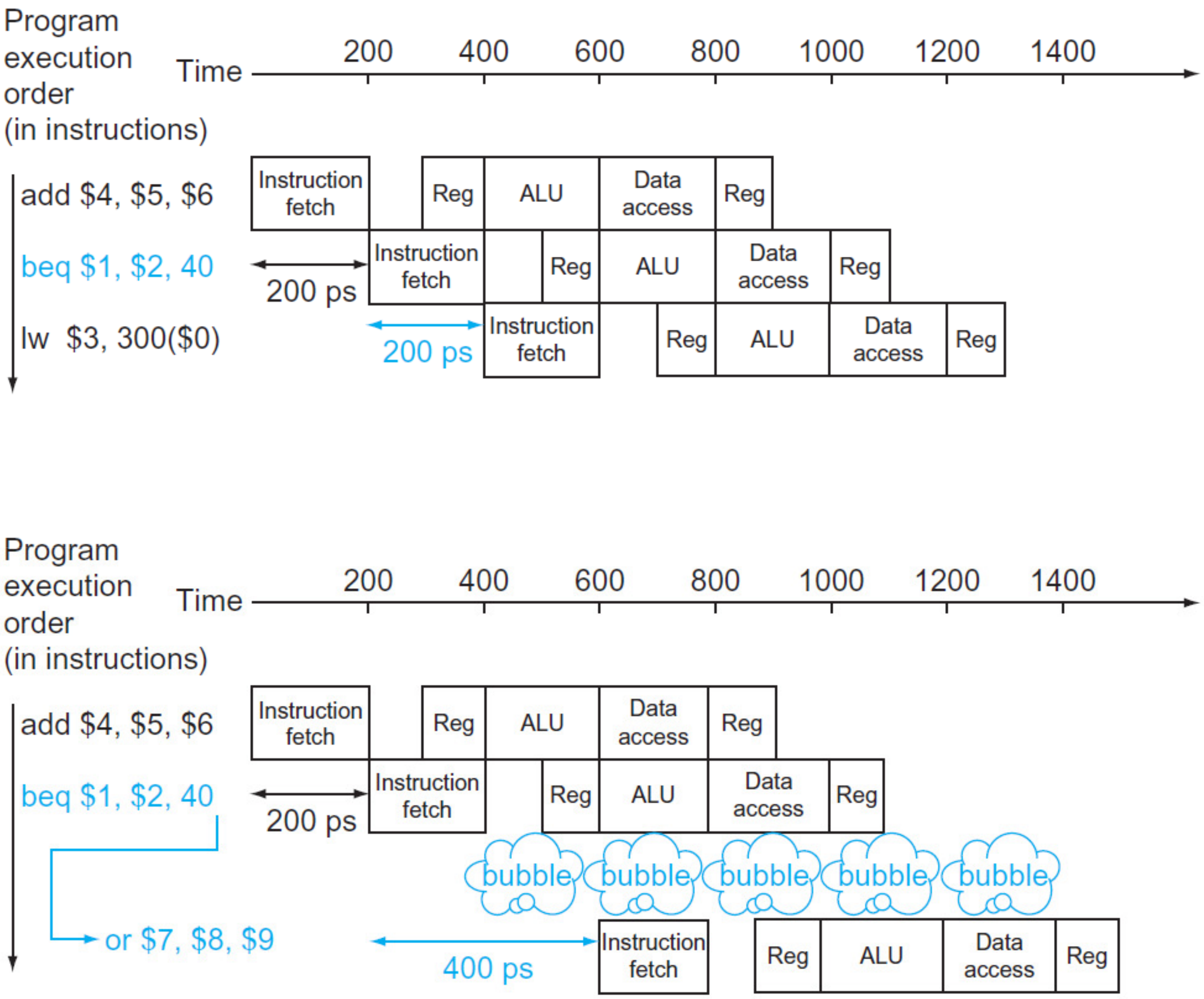


FIGURE 4.32 Predicting that branches are not taken as a solution to control hazard. The top drawing shows the pipeline when the branch is not taken. The bottom drawing shows the pipeline when the branch is taken. As we noted in Figure 4.31, the insertion of a bubble in this fashion simplifies what actually happens, at least during the first clock cycle immediately following the branch. Section 4.8 will reveal the details.

Exceptions

An exception is an **unscheduled event that changes the normal flow of instruction execution**. They were initially created to handle unexpected events from within the processor. The same basic mechanism was extended for I/O devices to communicate with the processor.

An exception that happens **synchronously with respect the clock** (i.e., occurs at the same place every time the program is executed) is called a **trap** (e.g., arithmetic overflows, invalid memory access, system calls, …)

An exception that happens **asynchronously** (i.e., occurs from outside the CPU and independently of the program) is called an **interrupt** (e.g., I/O request, timer, hardware malfunction, …)

Exceptions

The hardware and the operating system must work in conjunction so that exceptions behave as expected.

The hardware contract is to:

- Stop the offending instruction in midstream
- Set a register to show the cause of the exception
- Save the address of the offending instruction
- Jump to a prearranged procedure called **exception handler**

Exceptions

The operating system contract is to look at the cause of the exception and act appropriately.

For an undefined instruction, hardware failure, or arithmetic overflow exception, the operating system normally kills the program and returns an indicator of the reason.

For an I/O device request or system call, the operating system saves the state of the program, performs the desired task, and restores the program to continue execution. In the case of I/O requests, it may often choose to run another task in the meantime, since the requesting task may often not be able to proceed until the I/O is complete.