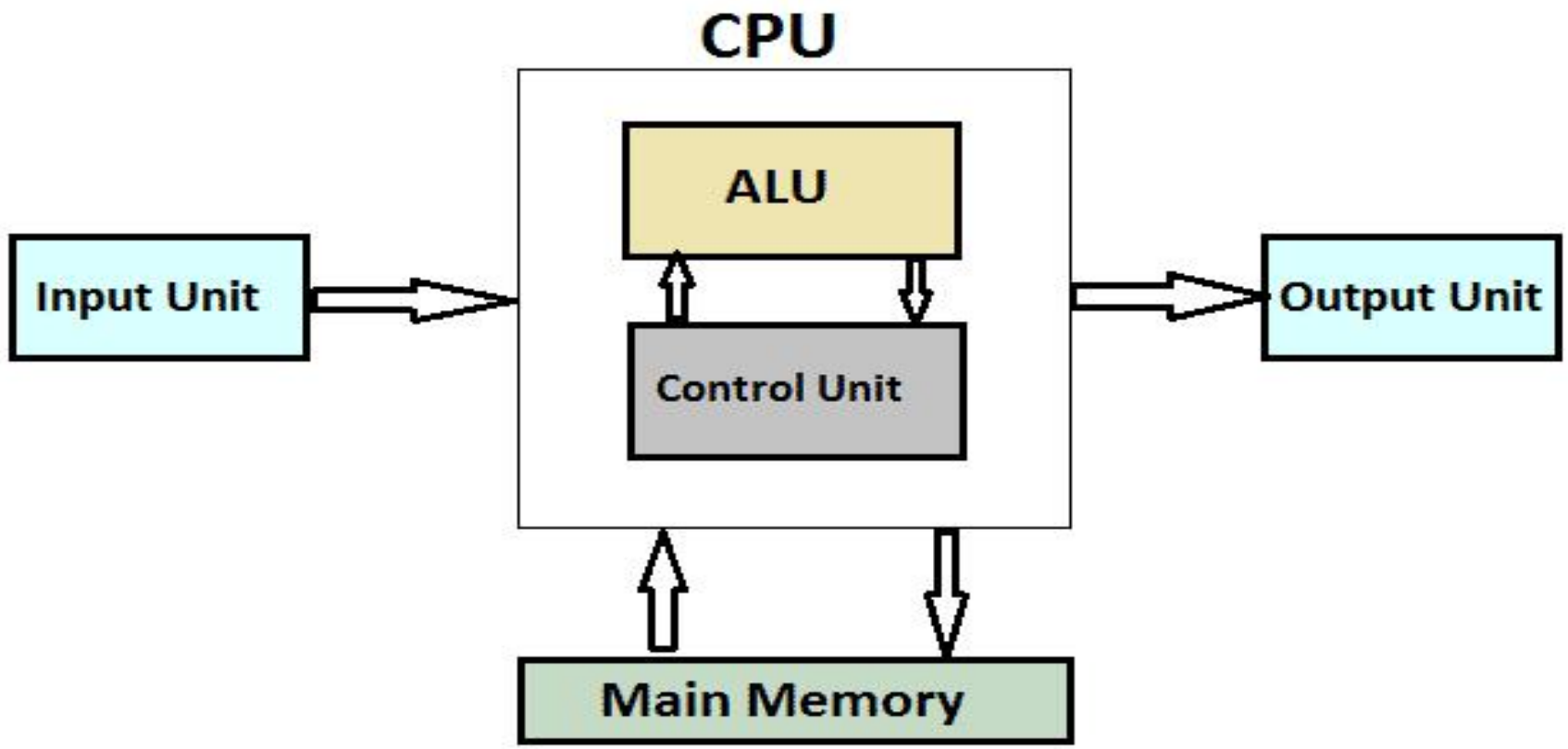
# Module IV

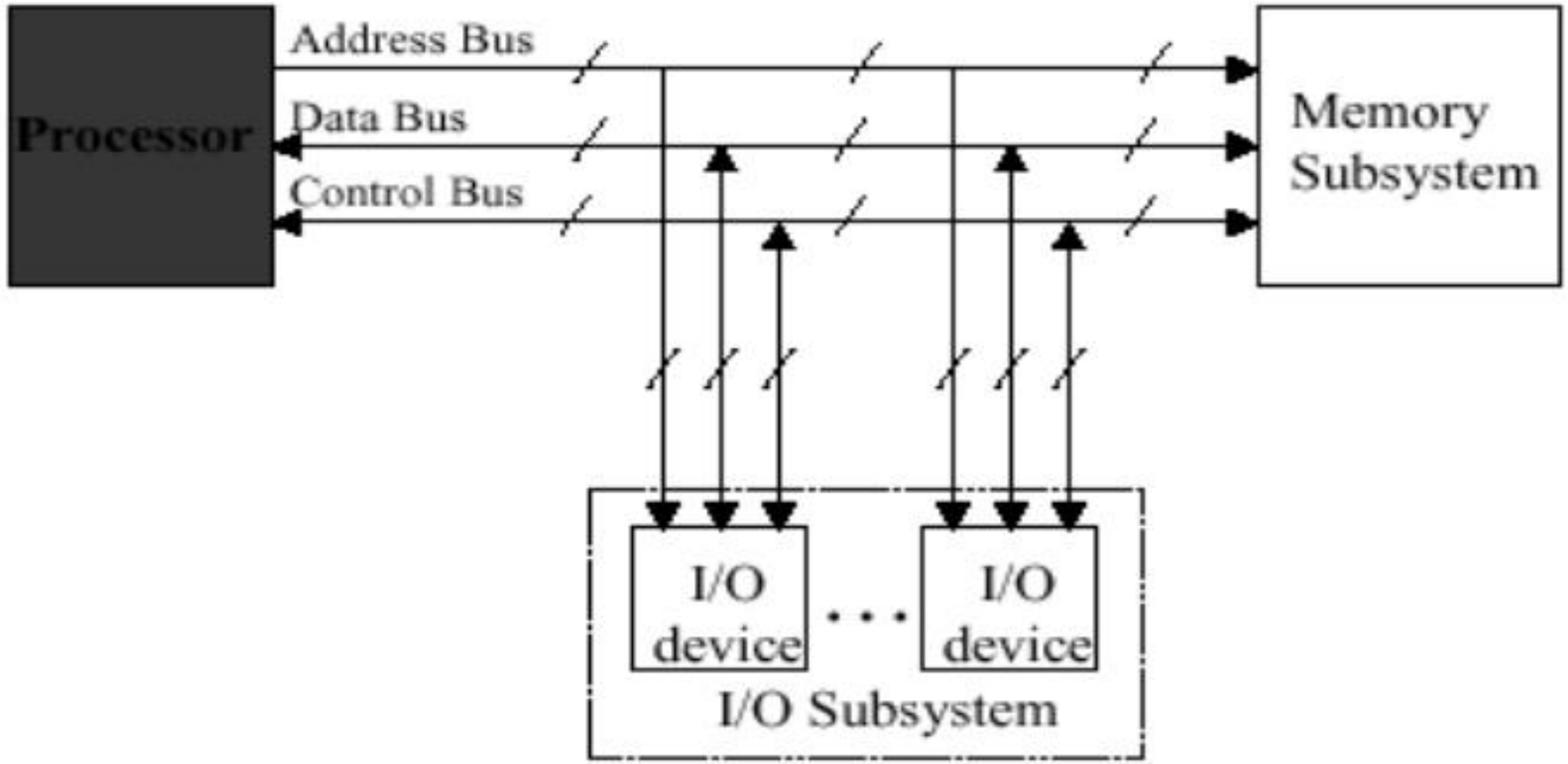
DIGITAL FUNDAMENTALS & COMPUTER ARCHITECTURE

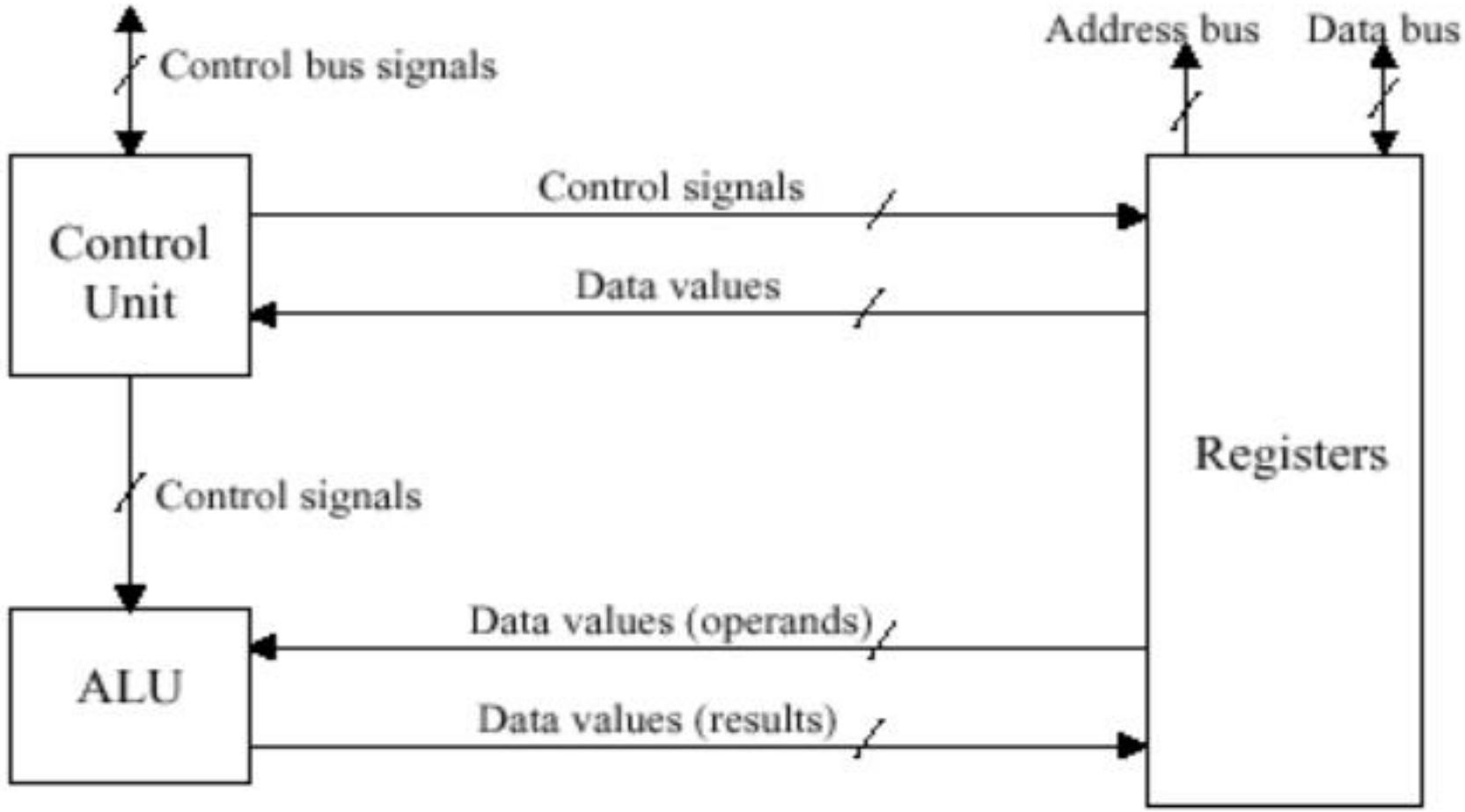
# Module IV

- The Processor - Introduction, Logic design conventions, Building a datapath, A simple implementation scheme, An overview of pipelining - Pipelined datapath and control - Structural hazards - Data hazards - Control hazards, I/O organization - Accessing I/O devices, interrupts - handling multiple devices, Direct memory access

# The Processor




Processor
Address Bus
Data Bus
Control Bus
Memory Subsystem
I/O device
I/O device
I/O Subsystem


Control bus signals
Address bus
Data bus
Control Unit
Control signals
Data values
Registers
Control signals
ALU
Data values (operands)
Data values (results)

- *Processor (CPU)* is the active part of the computer, which does all the work of data manipulation and decision making.
- *Datapath* is the hardware that performs all the required operations, for example, ALU, registers, and internal buses.
- *Control* is the hardware that tells the datapath what to do, in terms of switching, operation selection, data movement between ALU components, etc.

# Introduction

- MIPS (Microprocessor without Interlocked Pipelined Stages) is a reduced instruction set computer (RISC) instruction set architecture developed by MIPS Computer Systems.
- For every instruction, the first two steps are identical:
- 1. Send the *program counter* (PC) to the memory that contains the code and fetch the instruction from that memory.
- 2. Read one or two registers, using fi elds of the instruction to select the registers to read. For the load word instruction, we need to read only one register, but most other instructions require reading two registers.
- After these two steps, the actions required to complete the instruction depend on the instruction class. Fortunately, for each of the three instruction classes (memory-reference, arithmetic-logical, and branches), the actions are largely the same, independent of the exact instruction.
- The simplicity and regularity of the MIPS instruction set simplifies the implementation by making the execution of many of the instruction classes similar.

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

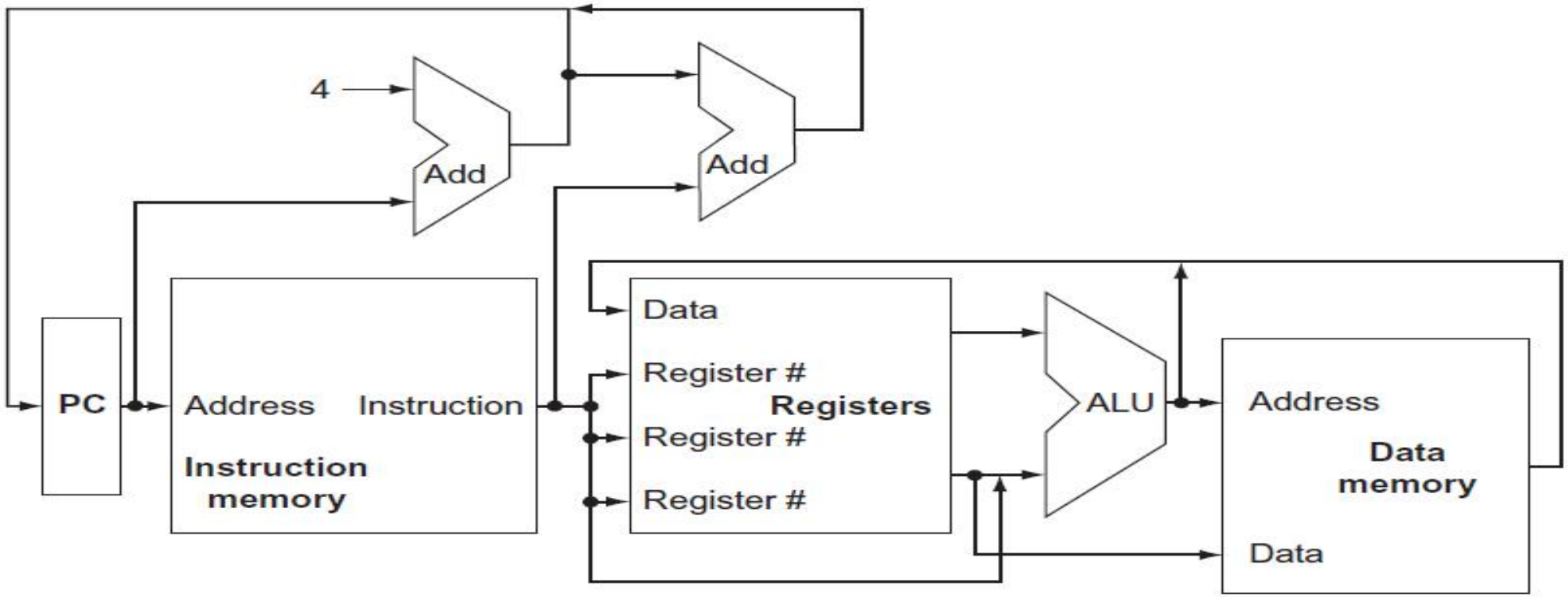
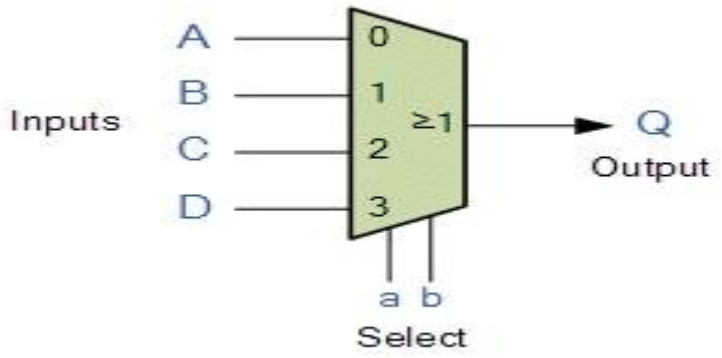



**FIGURE 4.1 An abstract view of the implementation of the MIPS subset showing the major functional units and the major connections between them.** All instructions start by using

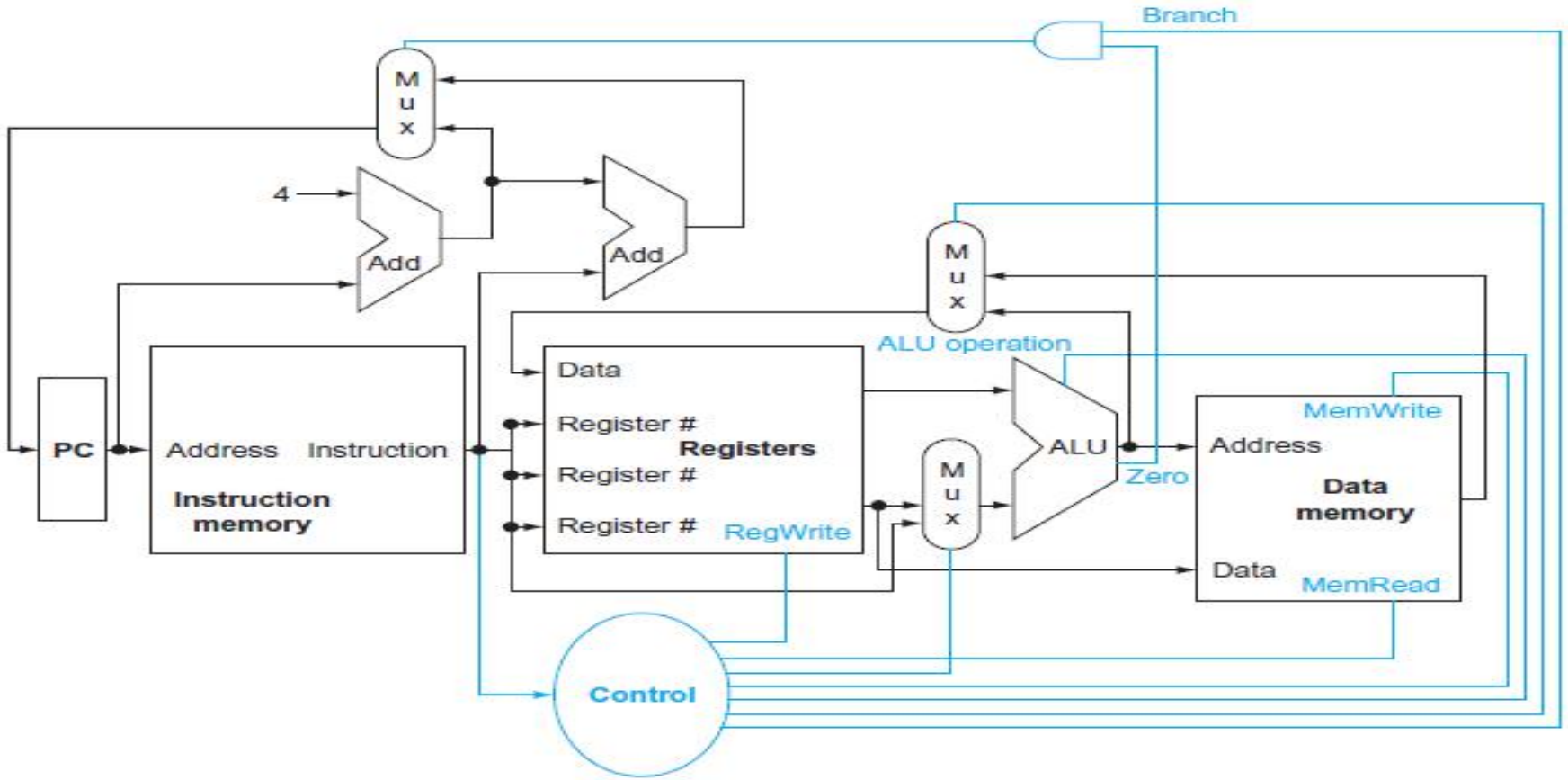
- the value written into the PC can come from one of two adders,
- the data written into the register file can come from either the ALU or the data memory, and the second input to the ALU can come from a register or the immediate field of the instruction.
- In practice, these data lines cannot simply be wired together; we must add a logic element that chooses from among the multiple sources and steers one of those sources to its destination. This selection is commonly done with a device called a **multiplexor**, although this device might better be called a data selector.



- All instructions start by using the program counter to supply the instruction address to the instruction memory.
- After the instruction is fetched, the register operands used by an instruction are specified by fields of that instruction. Once the register operands have been fetched, they can be operated on to compute a memory address (for a load or store), to compute an arithmetic result (for an integer arithmetic-logical instruction), or a compare (for a branch).
- If the instruction is an arithmetic-logical instruction, the result from the ALU must be written to a register.
- If the operation is a load or store, the ALU result is used as an address to either store a value from the registers or load a value from memory into the registers. The result from the ALU or memory is written back into the register file.
- Branches require the use of the ALU output to determine the next instruction address, which comes either from the ALU (where the PC and branch off set are summed) or from an adder that increments the current PC by 4.

- Generally, LDR is used to **load** something from memory into a register, and STR is used to **store** something from a register to a memory address.

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



- several of the units must be controlled depending on the type of instruction.
- the data memory must read on a load and written on a store.
- The register file must be written only on a load or an arithmetic-logical instruction.
- **three required multiplexors ad**ded, as well as **control lines** for the major functional units.
- A ***control unit***, which has the instruction as an input, is used to determine how to set the control lines for the functional units and **two of the multiplexors.**
- The **third multiplexor**, which determines whether PC + 4 or the branch destination address is written into the PC, is set based on the Zero output of the ALU, which is used to perform the comparison of a BEQ instruction.
- BEQ (short for 'Branch if EQual') is the mnemonic for a machine language instruction which branches, or 'jumps', to the address

- **LDA NumA** ***Read the value*** **"NumA"**
- **CMP NumB** ***Compare against*** **"NumB"**
- **BEQ Equal** ***Go to label "Equal" if*** **"NumA" = "NumB"**
- ***... Execution continues here if*** **"Number" <>** ***0***

- **The basic implementation of the MIPS subset, including the necessary multiplexors and control lines.**
- The **top multiplexor** (“Mux”) controls what value replaces the PC (PC + 4 or the branch destination address); the multiplexor is controlled by the gate that “ANDs” together the Zero output of the ALU and a control signal that indicates that the instruction is a branch.
- The middle multiplexor, whose output returns to the register file, is used to steer the output of the ALU (in the case of an arithmetic-logical instruction) or the output of the data memory (in the case of a load) for writing into the register file.
- Finally, the bottommost multiplexor is used to determine whether the second ALU input is from the registers (for an arithmetic-logical instruction or a branch) or from the off set field of the instruction (for a load or store).
- The added control lines are straightforward and determine the operation performed at the ALU, whether the data memory should read or write, and whether the registers should perform a write operation.

# Logic design conventions

- The Datapath elements in the MIPS implementation consist of two different types of logic elements: elements that operate on **data values** and elements that contain **state**.
- The elements that operate on data values are all **combinational**, which means that their outputs depend only on the current inputs.
- An element contains **state** if it has some **internal storage**.
- A **state** element has at least **two inputs** and **one output**.
- The required inputs are the data value to be written into the element and the **clock**, which determines when the data value is written.
- The **output** from a state element provides the value that was written in an earlier clock cycle.
- Logic components that contain state are also called ***sequential***, because their outputs depend on both their inputs and the contents of the internal state.

- the output from the functional unit representing the registers depends both on the register numbers supplied and on what was written into the registers previously

# Clocking Methodology

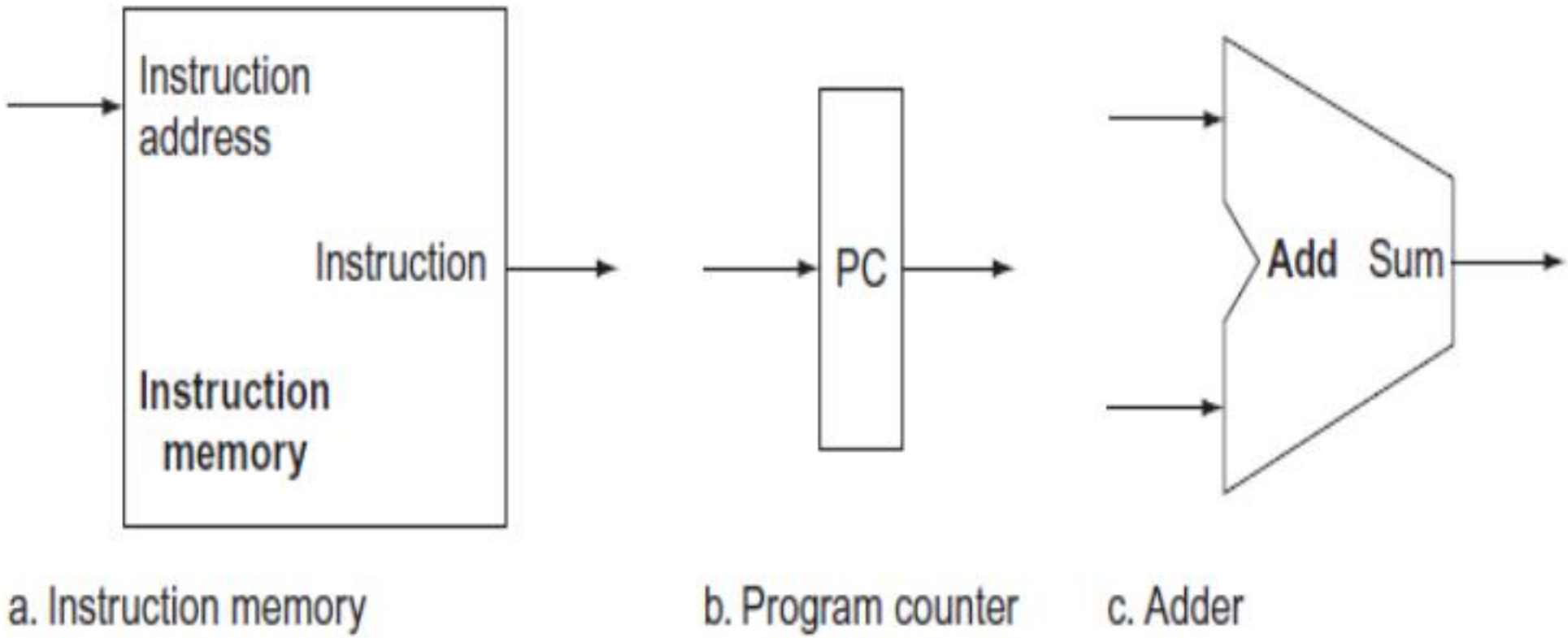
- A **clocking methodology** defines when signals can be read and when they can be written.
- It is important to specify the timing of reads and writes, because if a signal is written at the same time it is read, the value of the read could correspond to the old value, the newly written value, or even some mix of the two! Computer designs cannot tolerate such unpredictability. A clocking methodology is designed to make hardware predictable.
- For simplicity, we will assume an **edge-triggered clocking** methodology. An edge-triggered clocking methodology means that any values stored in a sequential logic element are updated only on a clock edge, which is a quick transition from low to high or *vice versa.*
- Because only state elements can store a data value, any collection of combinational logic must have its inputs come from a set of state elements and its outputs written into a set of state elements. The inputs are values that were written in a previous clock cycle, while the outputs are values that can be used in a following clock cycle.

# Building a datapath

- Major components required to execute each class of MIPS instructions.
- the first element we need: **a memory unit** to store the instructions of a program and supply instructions given an address.
- the **program counter** (**PC**) is a register that holds the address of the current instruction.
- an **adder** to increment the PC to the address of the next instruction.
- 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.

- consider the R-format instructions
- They all read two registers, perform an ALU operation on the contents of the registers, and write the result to a register.
- call these instructions either ***R-type instructions*** or *arithmetic-logical instructions .* This instruction class includes add, sub, AND, OR.
- The processor's 32 general-purpose registers are stored in a structure called a **register file**.
- A **register file** is a collection of registers in which any register can be read or written by specifying the number of the register in the file.
- The **register file** contains the register state of the computer. In addition, we will need an ALU to operate on the values read from the registers.
- **R-format instructions have three register** operands, so we will need to read two data words from the register file and write one

- For each data word **to be read from the registers**, we need an **input to the register file** that specifies the ***register number*** to be read and an output from the register file that will **carry the value** that has been read from the registers.
- **To write a data word**, we will need two inputs: one to specify the register number to be written and one to supply the *data* to be written into the register.
- The register file always outputs the contents of whatever register numbers are on the Read register inputs. Writes, however, are controlled by the write control signal, which must be asserted for a write to occur at the clock edge.




a. Instruction memory b. Program counter c. Adder

- **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.
- 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.

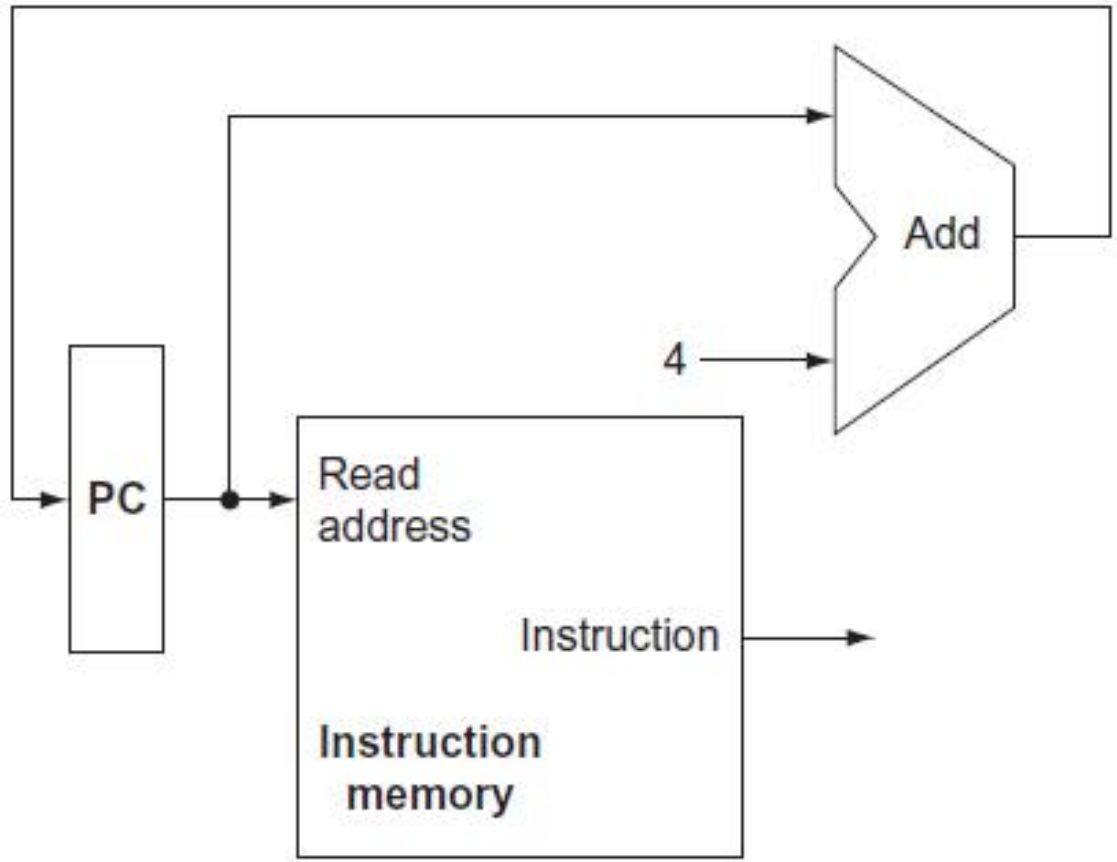



**FIGURE 4.6 A portion of the datapath used for fetching instructions and incrementing the program counter.** The fetched instruction is used by other parts of the datapath.

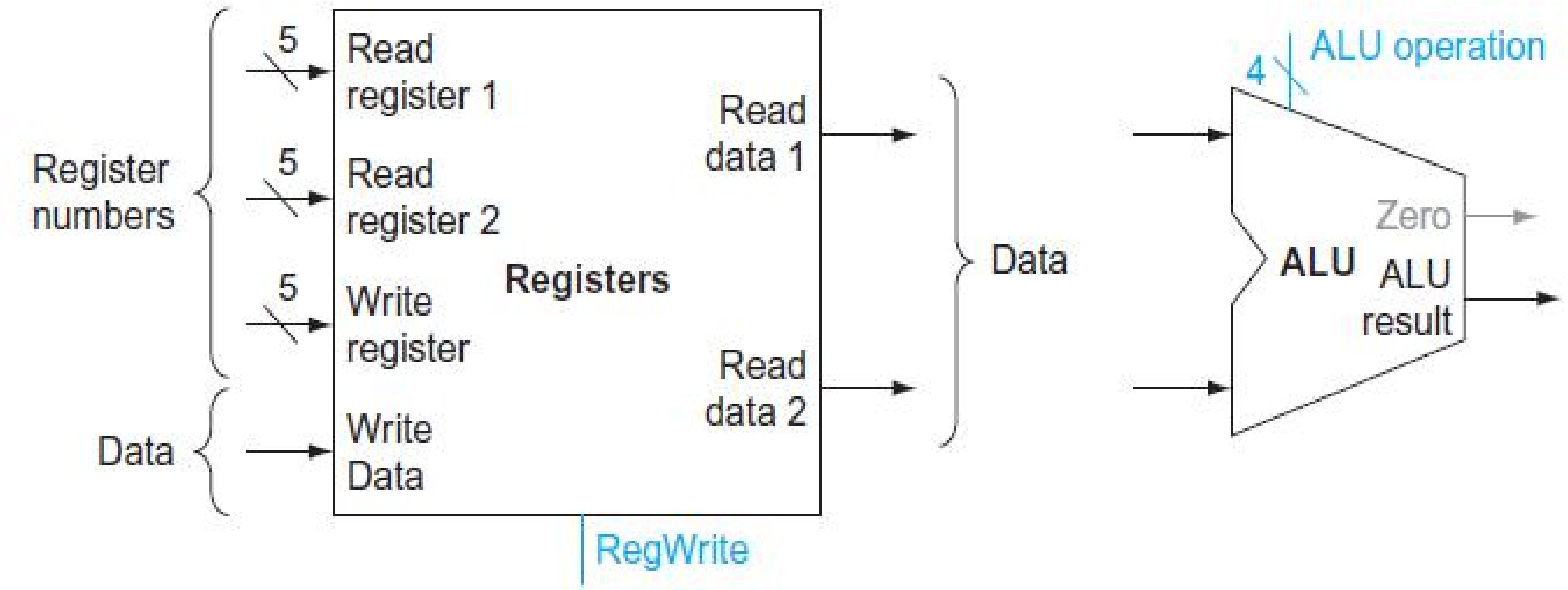



**FIGURE 4.7 The two elements needed to implement R-format ALU operations are the register file and the ALU.** The register file contains all the registers and has two read ports and one write

**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 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.

# MIPS load word and store word instructions

- General form lw t1,offset_value( t2) or sw t1,offset_value ( t2). These instructions compute a memory address by adding the base register, which is $t2, to the 16-bit signed offset field contained in the instruction.
- If the instruction is a store, the value to be stored must also be read from the register file where it resides in t1.
- If the instruction is a load, the value read from memory must be written into the register file in the specified register, which is t1.
- Thus, we will need both the register file and the ALU.
- In addition, we will need a unit to **sign-extend** the 16-bit off set field in the instruction to a 32-bit signed value,
- and a **data memory** unit to read from or write to.

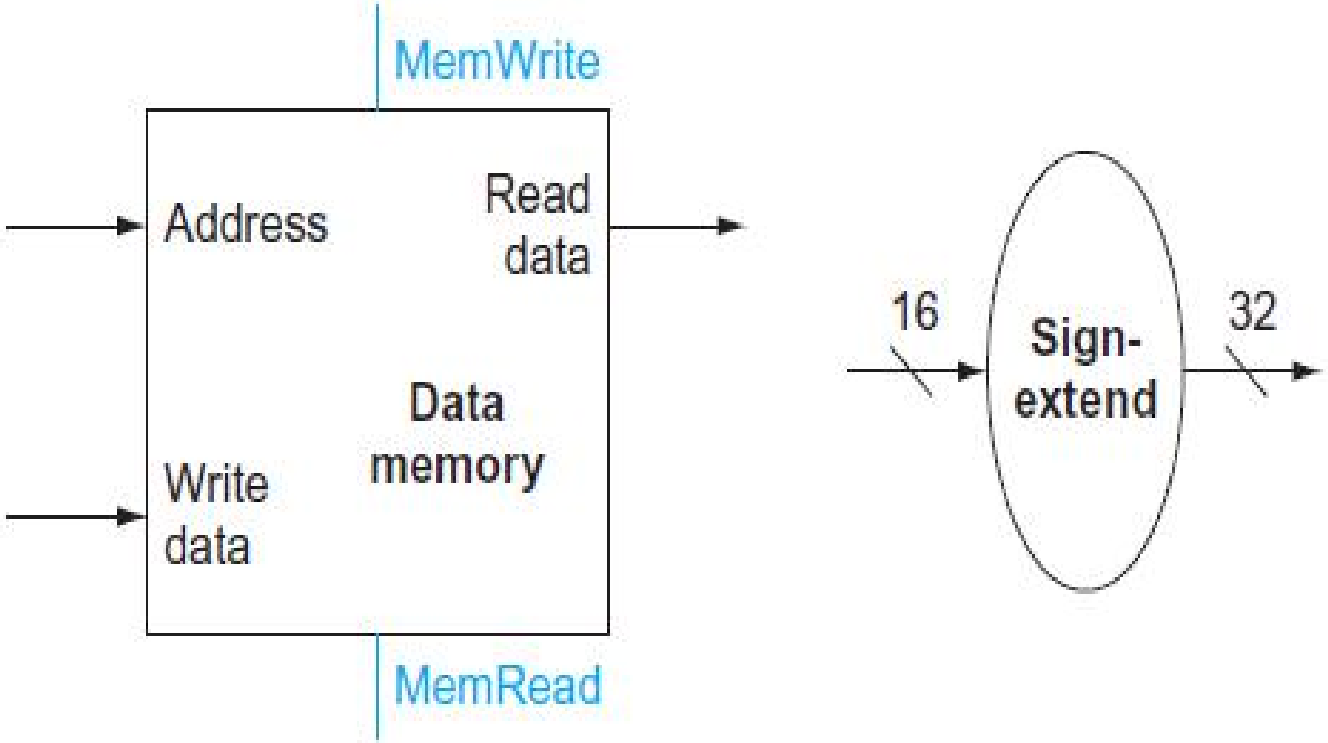



**FIGURE 4.8 The two units needed to implement loads and stores, in addition to the register file and ALU of Figure 4.7, are the data memory unit and the sign extension unit.**

- The **data memory** must be written on store instructions; hence, data memory has read and write control signals, an address input, and an input for the data to be written into memory.
- The **beq** instruction has three operands, **two registers** that are compared for equality, and a **16-bit off set** used to compute the **branch target address** relative to the branch instruction address.
- Its form is **beq t1, t2,offset**.
- To implement this instruction, we must compute the **branch target address** by adding the **sign-extended off set field of the instruction to the PC**.
- The instruction set architecture specifies that the base for the branch address calculation is the address of the instruction following the branch.
- Since we compute PC + 4 (the address of the next instruction) in the instruction fetch datapath, it is easy to use this value as the base for

- The architecture also states that the **off set** field is **shifted left** 2 **bits** so that it is a **word off set**; this shift increases the effective range of the off set field by a factor of 4.
- As well as computing the branch target address, we must also determine whether the next instruction is the instruction that follows sequentially or the instruction at the branch target address.
- When the condition is true (i.e., the operands are equal), the branch target address becomes the new PC, and we say that the **branch** is **taken**. If the operands are not equal, the incremented PC should replace the current PC (just as for any other normal instruction); in this case, we say that the **branch** is **not taken**.
- Thus, the branch datapath must **do two operations**: compute the branch target address and compare the register contents.

- To compute the **branch target address**, the branch datapath includes a **sign extension unit** and an **adder.**
- To perform the compare, we need to use the register file to supply the two register operands (although we will not need to write into the register file).
- In addition, the comparison can be done using the ALU , Since that ALU provides an output signal that indicates whether the result was 0, we can send the two register operands to the ALU with the control set to do a subtract. If the Zero signal out of the ALU unit is asserted, we know that the two values are equal. Although the Zero output always signals if the result is 0, we will be using it only to implement the equal test of branches.

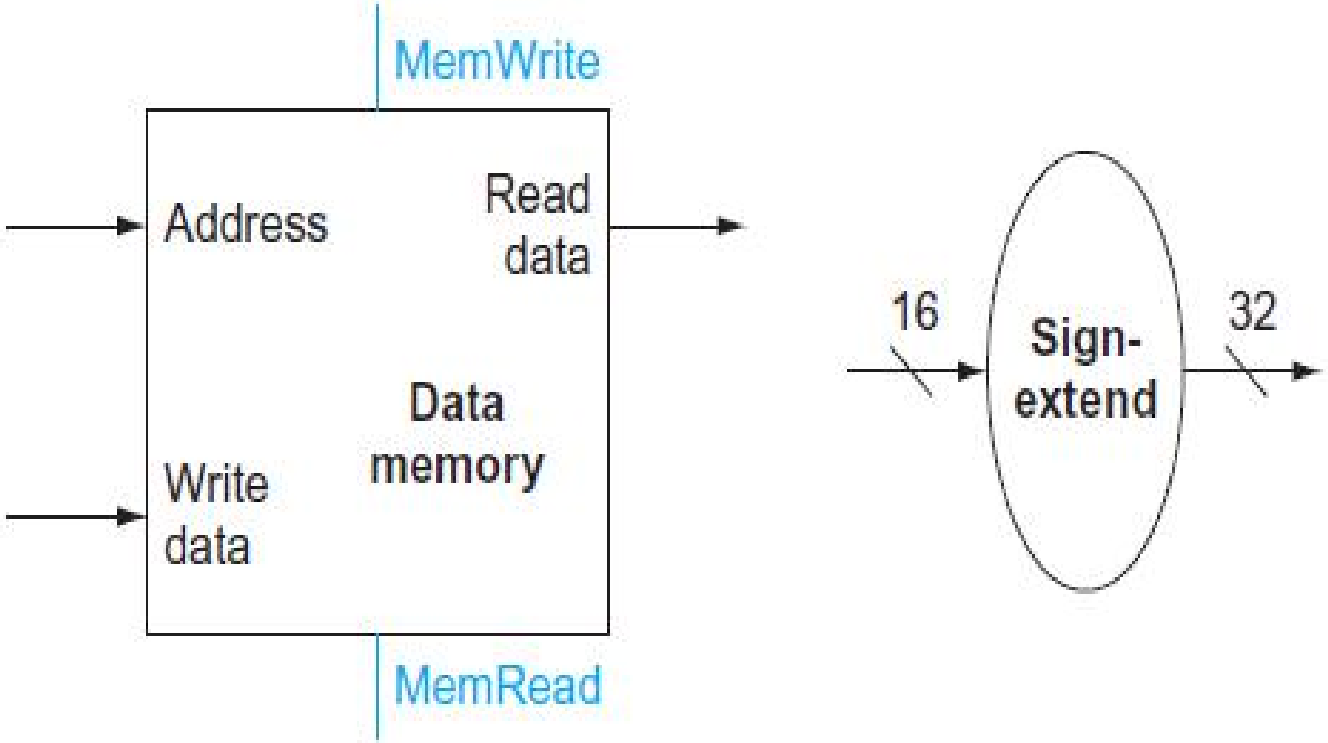
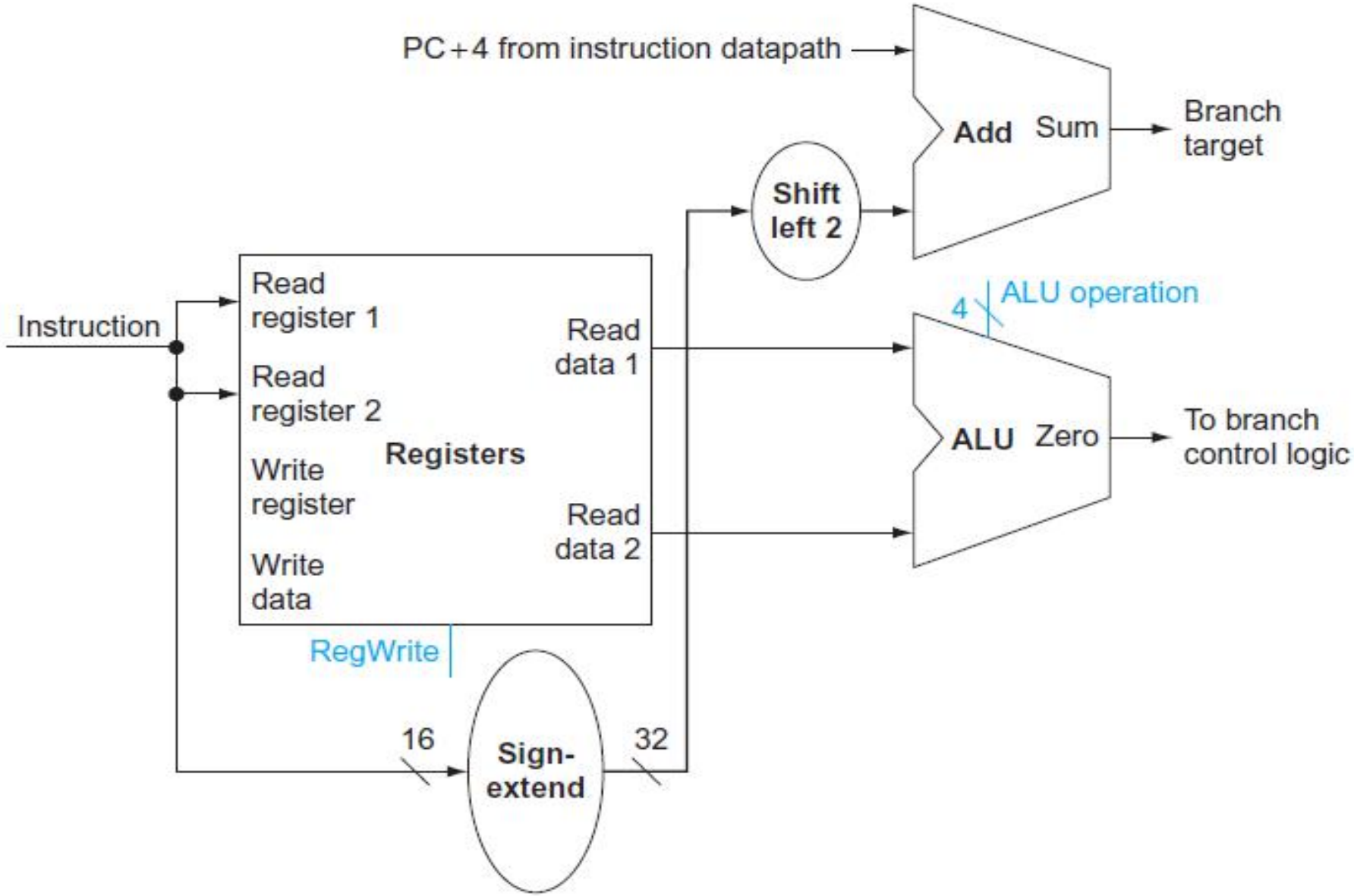



**FIGURE 4.8 The two units needed to implement loads and stores, in addition to the register file and ALU of Figure 4.7, are the data memory unit and the sign extension unit.**

- 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 sign extension unit has a 16-bit input that is sign-extended into a 32-bit result appearing on the output .
- **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 to the low-order end of the sign-extended off set field**; no actual shift hardware is needed, since the amount of the "shift " is constant.
- 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.


PC+4 from instruction datapath
Add
Sum
Branch target
Shift left 2
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
RegWrite
4
ALU operation
ALU
Zero
To branch control logic
16
Sign-extend
32

# Creating a Single Datapath

- This simplest datapath will attempt to execute all instructions in one clock cycle.
- To share a datapath element between two different instruction classes, we may need to allow **multiple connections to the input** of an element, using a **multiplexor** and **control signal** to select among the multiple inputs.
- To create a datapath with only a **single register file** and a **single ALU**, we must support two different sources for the **second ALU input**, as well as two different sources for the data stored into the register file. Thus, **one multiplexor is placed at the ALU input** and **another at the data input to the register file.**

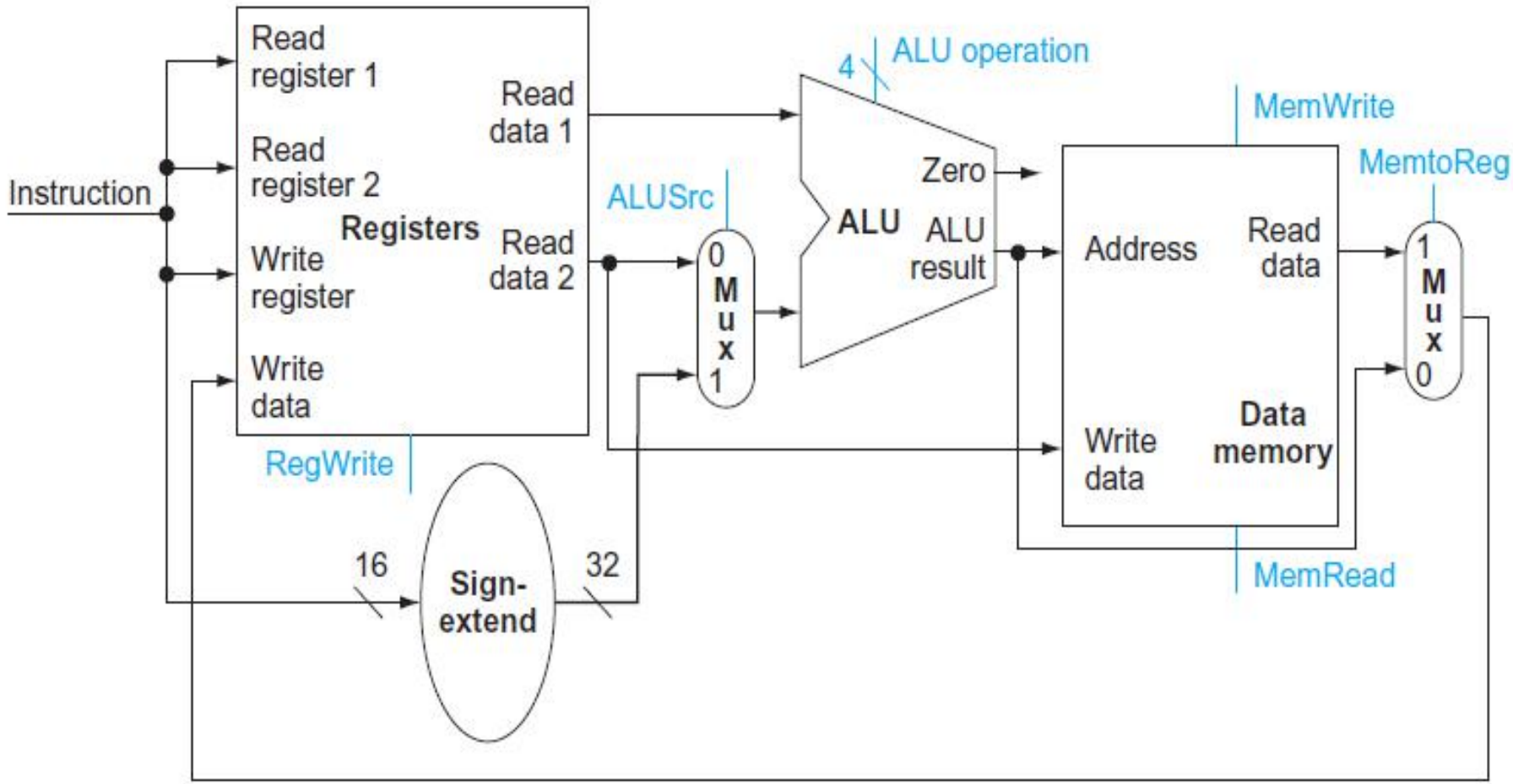



**FIGURE 4.10 The datapath for the memory instructions and the R-type instructions.** This example shows how a single datapath can be assembled from the pieces in Figures 4.7 and 4.8 by adding multiplexors. Two multiplexors are needed, as described in the example.

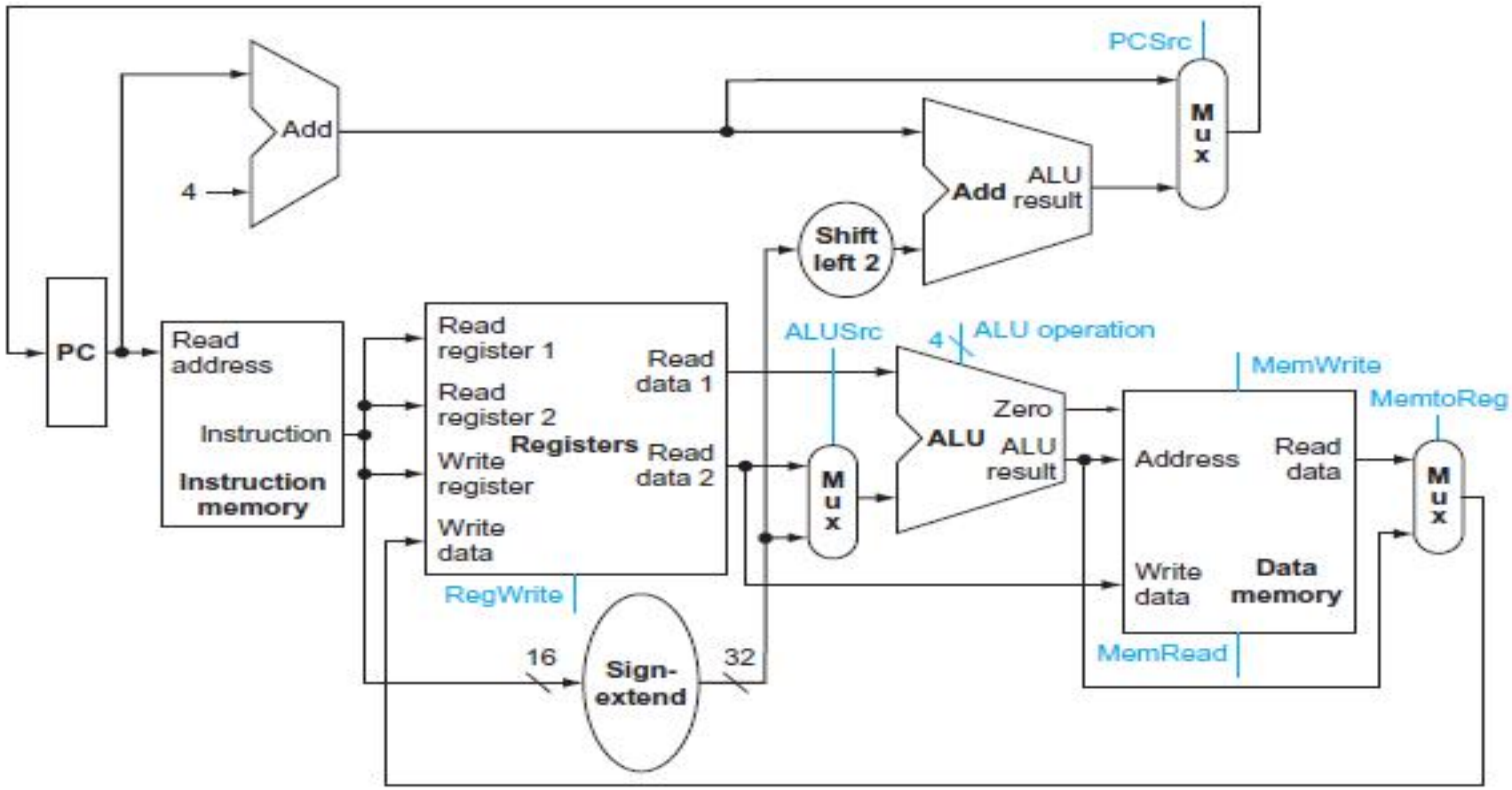



**FIGURE 4.11 The simple datapath for the core MIPS architecture combines the elements required by different instruction classes.** The components come from Figures 4.6, 4.9, and 4.10. This datapath can execute the basic instructions (load-store word, ALU operations, and branches) in a single clock cycle.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

# A simple implementation scheme

- Th is simple implementation covers *load word* (lw), *store word* (sw), *branch equal* (beq), and the arithmetic-logical instructions add, sub, AND, OR, and set on less than.
- **The ALU Control** : Th e MIPS ALU defines the 6 following combinations of four control inputs:

| ALU control lines | Function |
|---|---|
| 0000 | AND |
| 0001 | OR |
| 0010 | add |
| 0110 | subtract |
| 0111 | set on less than |
| 1100 | NOR |

- For **load word and store word instructions**, we use the ALU to compute the memory address by addition.
- For **the R-type instructions**, the ALU needs to perform one of the five actions (AND, OR, subtract, add, or set on less than), depending on the value of the 6-bit function field in the low-order bits of the instruction.
- **For branch equal**, the ALU must perform a subtraction.
- The ALU control block has also been added. The PC does not require a write control, since it is written once at the end of every clock cycle; the **branch control logic** determines whether it is written with the incremented PC or the branch target address.

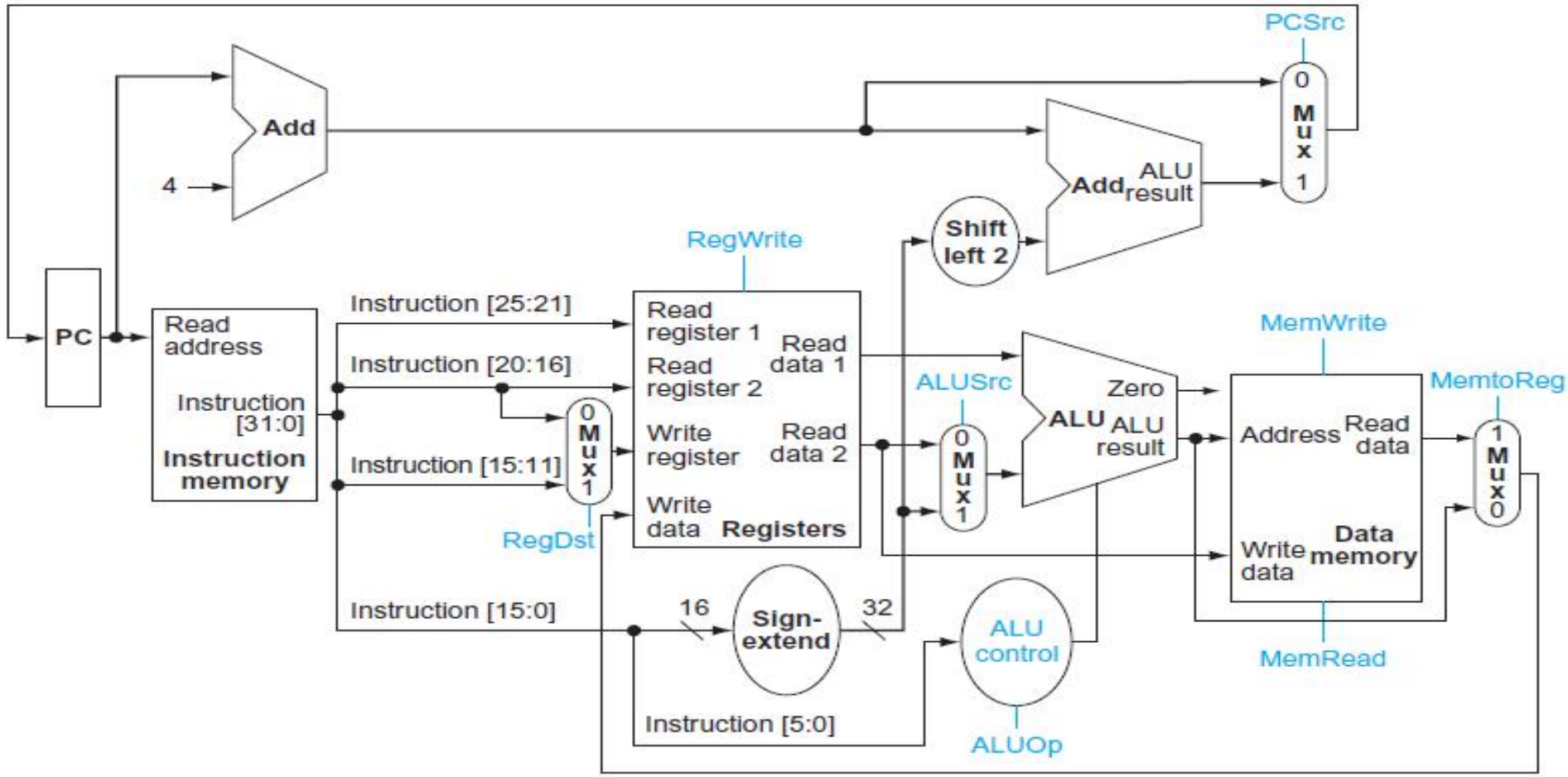



**FIGURE 4.15 The datapath of Figure 4.11 with all necessary multiplexors and all control lines identified.** The control lines are shown in color. The ALU control block has also been added. The PC does not require a write control, since it is written once at the end of every clock cycle; the branch control logic determines whether it is written with the incremented PC or the branch target address.

- We will just put the pieces together and then figure out the control lines that are needed and how to set them. We are not now worried about speed.
- We are assuming that the instruction memory and data memory are separate. So we are not permitting self modifying code. We are not showing how either memory is connected to the outside world (i.e. we are ignoring I/O).
- We have to use the same register file with all the pieces since when a load changes a register a subsequent R-type instruction must see the change, when an R-type instruction makes a change the lw/sw must see it (for loading or calculating the effective address, etc.
- We could use separate ALUs for each type but it is easy not to so we will use the same ALU for all. We do have a separate adder for incrementing the PC.

# An overview of pipelining

- **Pipelining** is an implementation technique in which multiple instructions are overlapped in execution.
- advanced pipelining techniques used in recent processors such as the Intel Core i7 and ARM Cortex-A8..
- Pipelining is a technique where multiple instructions are overlapped during execution. Pipeline is divided into stages and these stages are connected with one another to form a pipe like structure. Instructions enter from one end and exit from another end.
- Pipelining increases the overall instruction throughput.

The *nonpipelined* 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.
- When your roommate is done, start over with the next dirty load.

# The *pipelined* approach takes much less time

- As soon as the washer is finished with the first load and placed in the dryer, you load the washer with the second dirty load. When the first load is dry, you place it on the table to start folding, move the wet load to the dryer, and put the next dirty load into the washer. Next you have your roommate put the first load away, you start folding the second load, the dryer has the third load, and you put the fourth load into the washer. At this point all steps—called *stages* in pipelining—are operating concurrently.
- the reason pipelining is faster for many loads is that everything is working in parallel, so more loads are finished per hour.
- Pipelining improves throughput of our laundry system.
- Hence, pipelining would not decrease the time to complete one load of laundry,
- but when we have many loads of laundry to do, the improvement in throughput decreases the total time to complete the work.

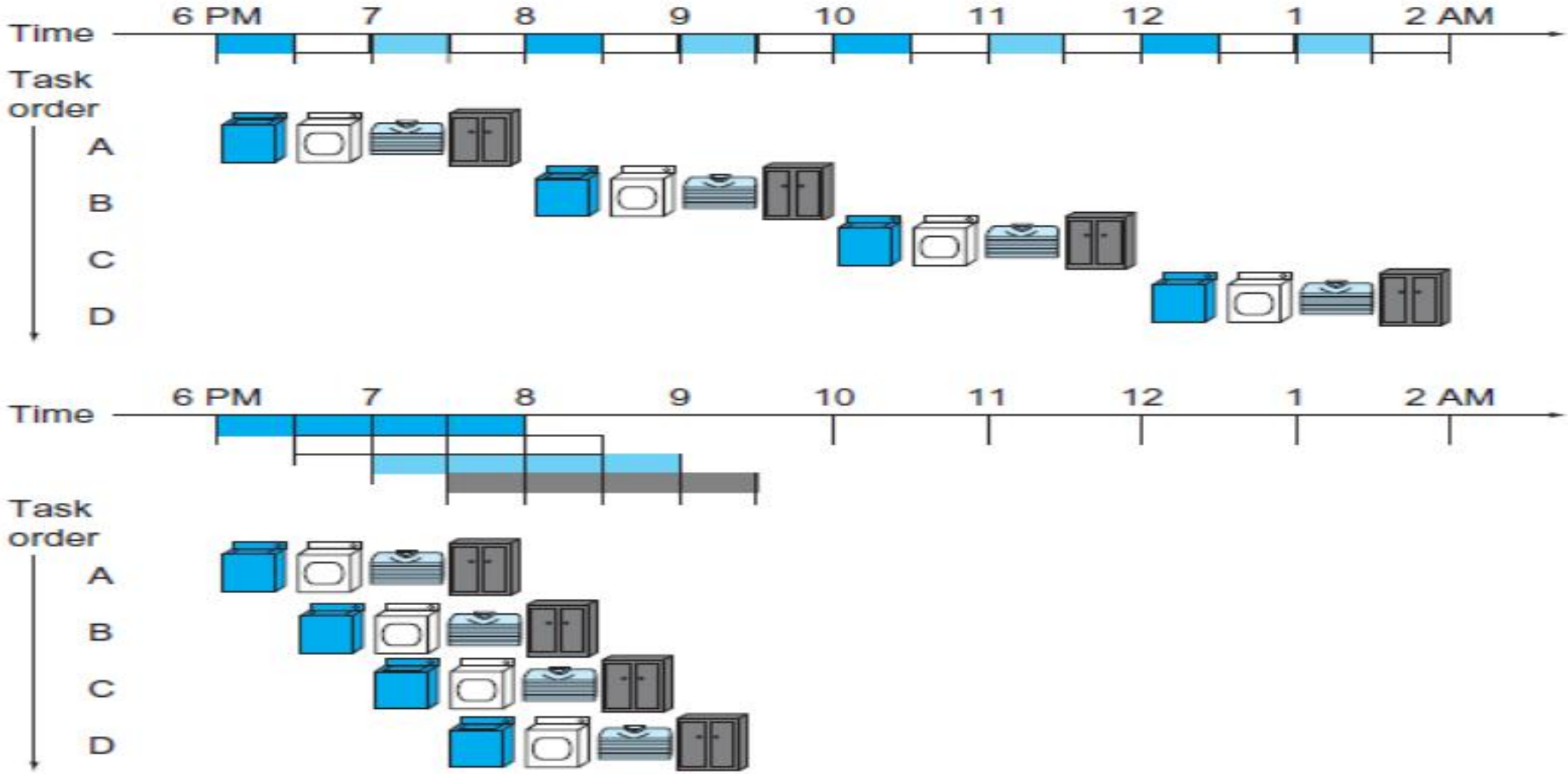



**FIGURE 4.25 The laundry analogy for pipelining.** Ann, Brian, Cathy, and Don each have dirty clothes to be washed, dried, folded, and put away. The washer, dryer, "folder," and "storer" each take 30 minutes for their task. Sequential laundry takes 8 hours for 4 loads of wash, while pipelined laundry takes just 3.5 hours. We show the pipeline stage of different loads over time by showing copies of the four resources on this two-dimensional time line, but we really have just one of each resource.

# pipeline instruction-execution.

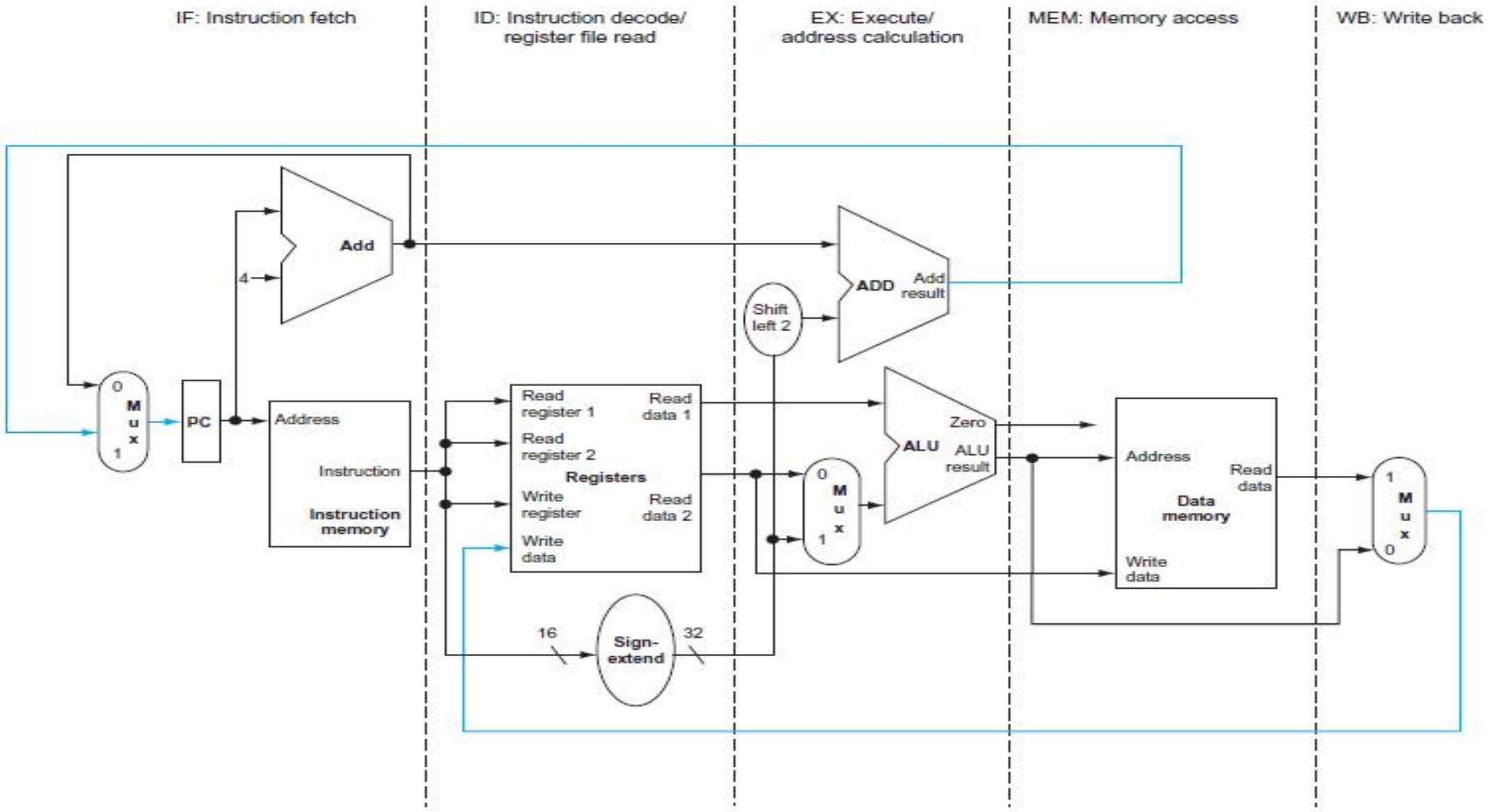
- 1. Fetch instruction from memory.
- 2. Read registers while decoding the instruction. The regular format of MIPS, instructions allows reading and decoding to occur simultaneously.
- 3. Execute the operation or calculate an address.
- 4. Access an operand in data memory.
- 5. Write the result into a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

# Pipelined datapath and control

- As with the single-cycle and multi-cycle implementations, we will start by looking at the datapath for pipelining. pipelining involves breaking up instructions into five stages:
- 1. IF: Instruction fetch
- 2. ID: Instruction decode and register file read
- 3. EX: Execution or address calculation
- 4. MEM: Data memory access
- 5. WB: Write back

- **Pipelining** increases the number of simultaneously executing instructions and the rate at which instructions are started and completed.
- Pipelining does not reduce the time it takes to complete an individual instruction, also called the **latency**. For example, the five-stage pipeline still takes 5 clock cycles for the instruction to complete.
- pipelining improves instruction *throughput* rather than individual instruction *execution time* or *latency*.

# single-cycle datapath, divided into stages.



- Each step of the instruction can be mapped onto the datapath from left to right.
- The only exceptions are the update of the PC and the write-back step  which sends either the ALU result or the data from memory to the left to be written into the register file.
- The two exceptions are:
- The WB stage places the result back into the register file in the middle of the datapath - > leads to data hazards.
- The selection of the next value of the PC – either the incremented PC or the branch address - > leads to control hazards.

# MIPS Instructions and Pipelining

- In order to implement MIPS instructions effectively on a pipeline processor, we must ensure that the instructions are the same length (*simplicity favors regularity*) for easy IF and ID, similar to the multicycle datapath. We also need to have few but consistent instruction formats, to avoid deciphering variable formats during IF and ID, which would prohibitively increase pipeline segment complexity for those tasks. Thus, the register indices should be in the same place in each instruction.

# Datapath Partitioning for Pipelining

- Recall the single-cycle datapath, which can be partitioned (subdivided) into functional unit. The single-cycle datapath contains separate Instruction Memory and Data Memory units, this allows us to directly implement in hardware the IF-ID-EX-MEM-WB representation of the MIPS instruction sequence. Observe that several control lines have been added, for example, to route data from the ALU output (or memory output) to the register file for writing. Also, there are again three ALUs, one for ALUop, another for JTA (Jump Target Address) computation, and a third for adding PC+4 to compute the address of the next instruction.

- In summary, pipelining improves efficiency by first regularizing the instruction format, for simplicity. We then divide the instructions into a fixed number of steps, and implement each step as a pipeline segment. During the pipeline design phase, we ensure that each segment takes about the same amount of time to execute as other segments in the pipeline. Also, we want to keep the pipeline full wherever possible, in order to maximize utilization and throughput, while minimizing set-up time.

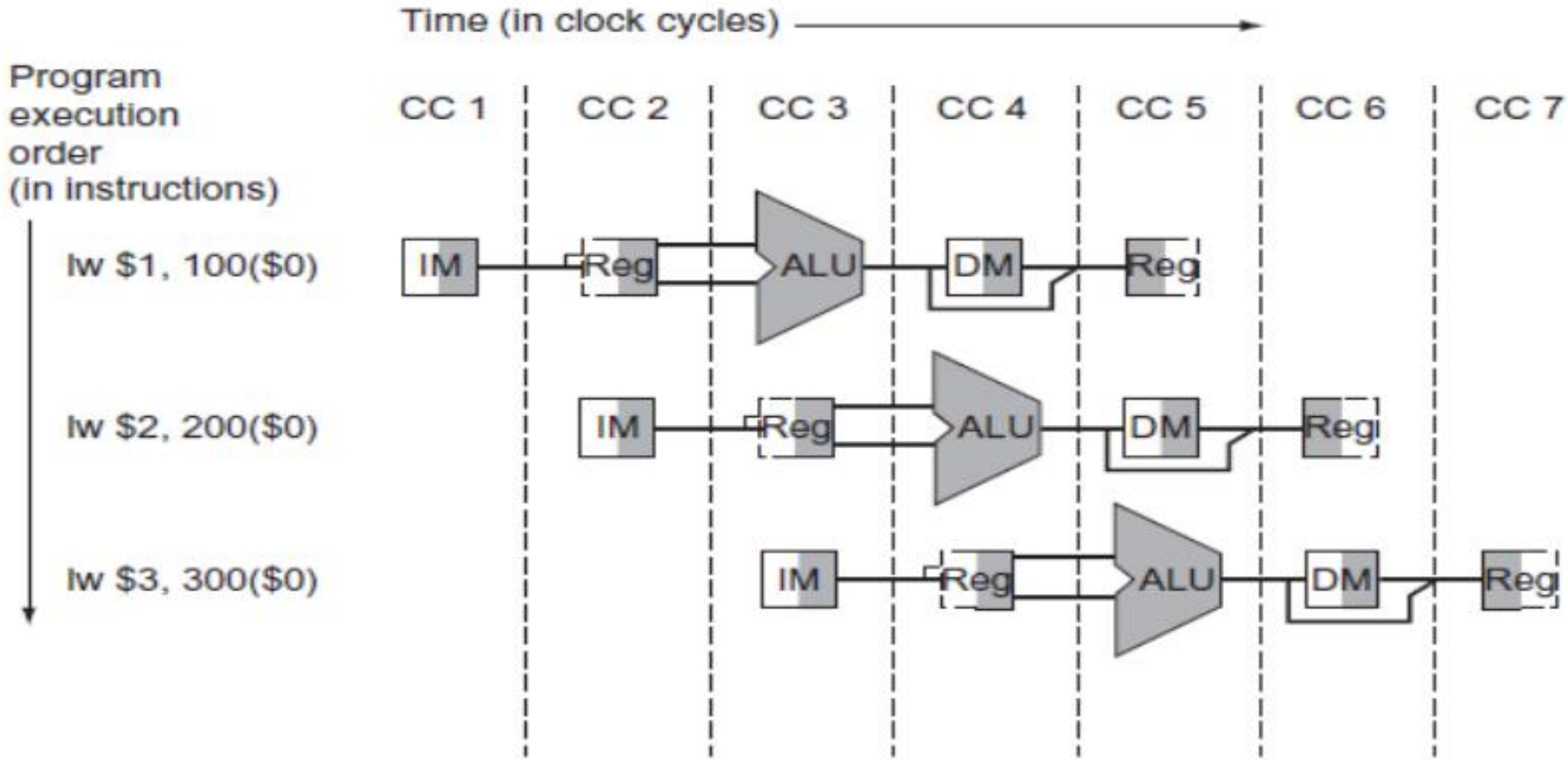



**FIGURE 4.34 Instructions being executed using the single-cycle datapath in Figure 4.33,**

Aactive portions of the datapath highlighted as a load instruction goes through the five stages of pipelined execution.

- 1. *Instruction fetch*
- *2. Instruction decode and register file read:*
- 3. *Execute or address calculation:*
- 4. *Memory access:*
- 5. *Write-back:*

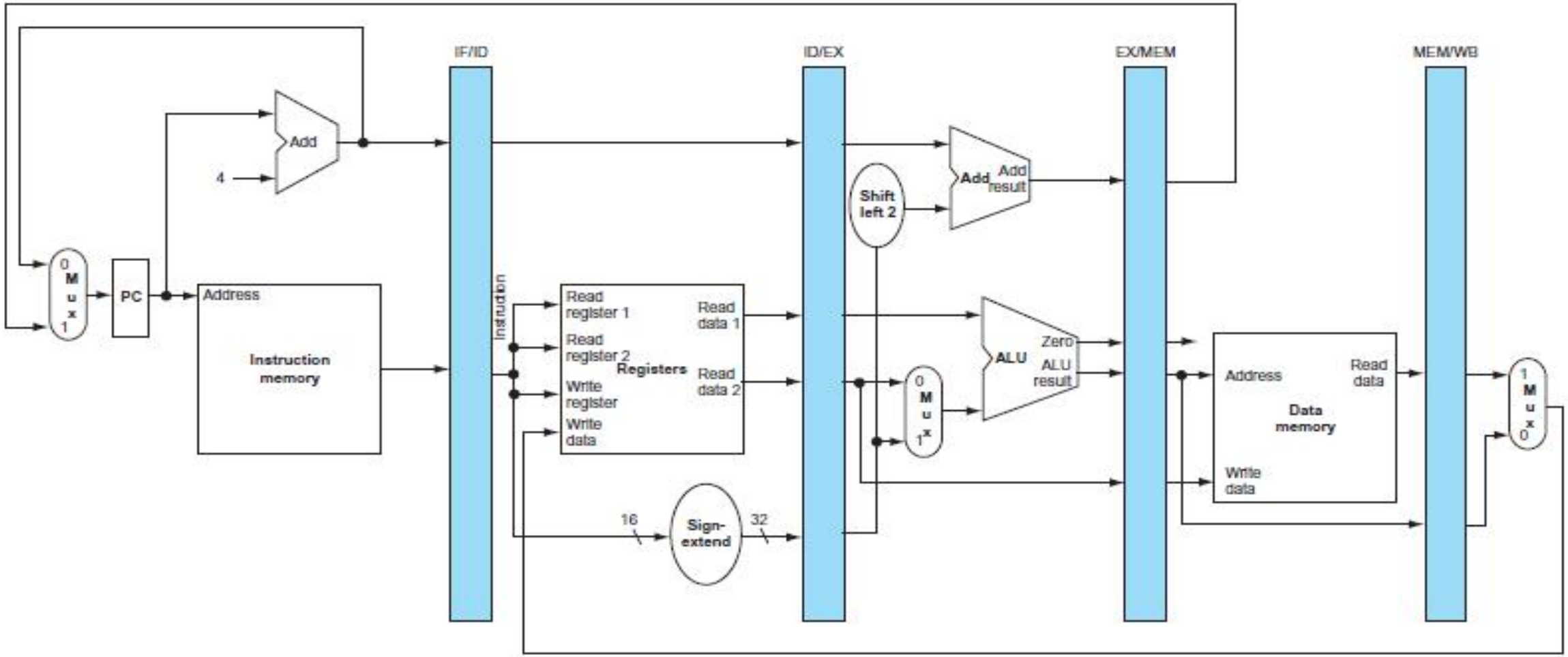



**FIGURE 4.35 The pipelined version of the datapath in Figure 4.33.** The pipeline registers, in color, separate each pipeline stage. They are labeled by the stages that they separate; for example, the first is labeled *IF/ID* because it separates the instruction fetch and instruction decode stages. The registers must be wide enough to store all the data corresponding to the lines that go through them. For example, the IF/ID register must be 64 bits wide, because it must hold both the 32-bit instruction fetched from memory and the incremented 32-bit PC address. We will expand these registers over the course of this chapter, but for now the other three pipeline registers contain 128, 97, and 64 bits, respectively.

# *Instruction fetch*

- The top portion of 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.

# *Instruction decode and register file read:*

- The bottom portion of 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.

# *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. Th at sum is placed in the EX/MEM pipeline register.

# *Memory access:*

- The top portion o shows 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.

# *Write-back:*

- The bottom portion o shows the final step: reading the data from the MEM/WB pipeline register and writing it into the register fi e in the middle.

# *Instruction decode and register file read:*

# Pipeline Hazards

- There are situations in pipelining when the next instruction cannot execute in the following clock cycle. These events are called *hazards*, and there are three different types.
- Structural hazards
- Data hazards
- Control hazards

# Structural hazards

- The first hazard is called a **structural hazard**. It means that the hardware cannot support the combination of instructions that we want to execute in the same clock cycle.
- Eg: A structural hazard in the laundry room would occur if we used a washer dryer combination instead of a separate washer and dryer, or if our roommate was busy doing something else and wouldn't put clothes away. Our carefully scheduled pipeline plans would then be foiled.

# Data hazards

- *Data Hazards* occur when an instruction depends on the result of a previous instruction still in the pipeline, which result has not yet been computed. The simplest remedy inserts stalls in the execution sequence, which reduces the pipeline's efficiency.
- In a computer pipeline, data hazards arise from the dependence of one instruction on an earlier one that is still in the pipeline.
- For example, suppose we have an add instruction followed immediately by a subtract instruction that uses the sum ( s0):
- add s0, t0, t1
- sub t2, s0, t3

- Without intervention, a data hazard could severely stall the pipeline. The add instruction doesn't write its result until the fifth stage, meaning that we would have to waste three clock cycles in the pipeline.
- The primary solution is based on the observation that we don't need to wait for the instruction to complete before trying to resolve the data hazard. For the code sequence above, as soon as the ALU creates the sum for the add, we can supply it as an input for the subtract

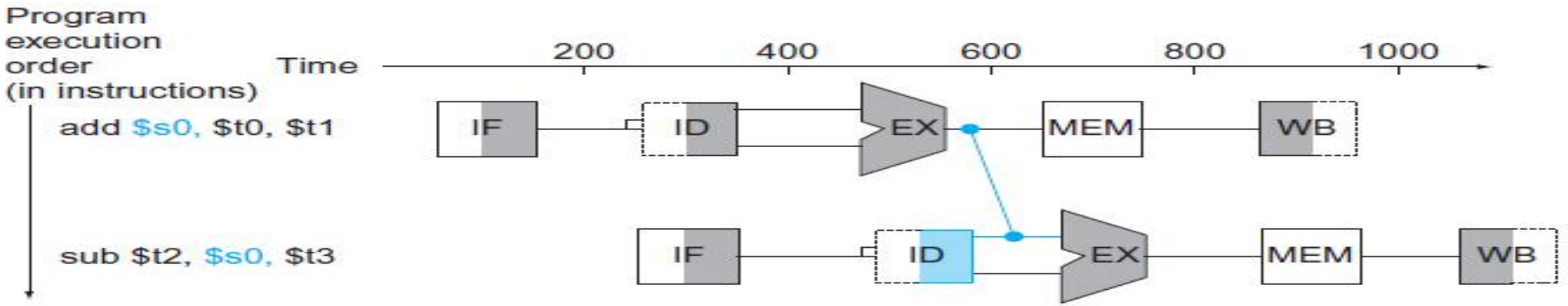



**FIGURE 4.29 Graphical representation of forwarding.** The connection shows the forwarding path from the output of the EX stage of `add` to the input of the EX stage for `sub`, replacing the value from register `$s0` read in the second stage of `sub`.

# Control hazards

- *Control Hazards* can result from branch instructions. Here, the branch target address might not be ready in time for the branch to be taken, which results in *stalls* (dead segments) in the pipeline that have to be inserted as local wait events, until processing can resume after the branch target is executed. Control hazards can be mitigated through accurate branch prediction (which is difficult), and by *delayed branch* strategies.

# I/O organization

One of the basic features of a computer is its ability to exchange data with other devices. This communication capability enables a human operator, for example, to use a keyboard and a display screen to process text and graphics. We make extensive use of computers to communicate with other computers over the Internet and access information around the globe. In other applications, computers are less visible but equally important. They are an integral part of home appliances, manufacturing equipment, transportation systems, banking and point-of-sale terminals. In such applications, input to a computer may come from a sensor switch, a digital camera, a microphone, or a fire alarm. Output may be a sound signal to be sent to a speaker or a digitally coded command to change the speed of a motor, open a valve, or cause a robot to move in a specified manner. In short, a general-purpose computer should have the ability to exchange information with a wide range of devices in varying environments.

# Accessing I/O devices

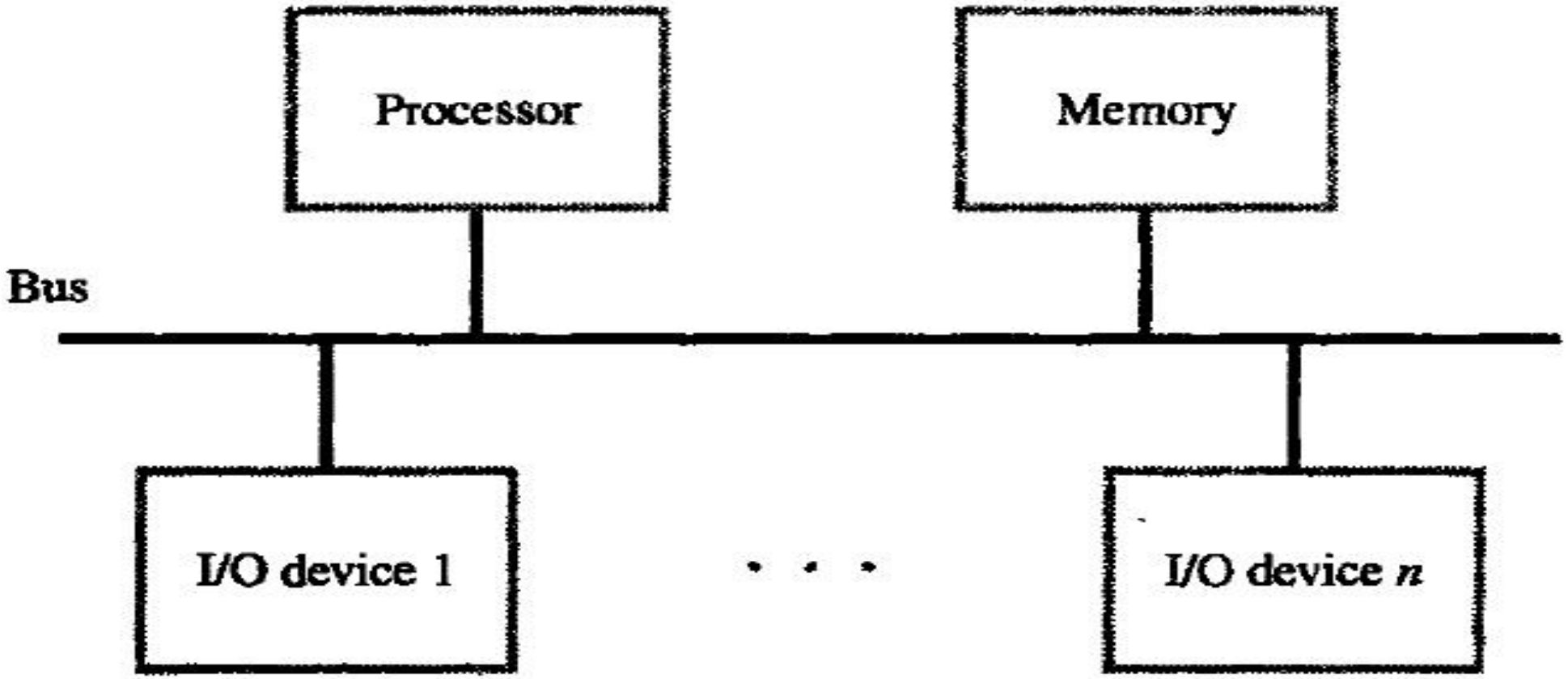



**Figure 4.1** A single-bus structure.

# Accessing I/O devices

A simple arrangement to connect I/O devices to a computer is to use a single bus arrangement, as shown in Figure 4.1. The bus enables all the devices connected to it to exchange information. Typically, it consists of three sets of lines used to carry address, data, and control signals. Each I/O device is assigned a unique set of addresses. When the processor places a particular address on the address lines, the device that recognizes this address responds to the commands issued on the control lines. The processor requests either a read or a write operation, and the requested data are transferred over the data lines. As mentioned in Section 2.7, when I/O devices and the memory share the same address space, the arrangement is called *memory-mapped I/O*.

With memory-mapped I/O, any machine instruction that can access memory can be used to transfer data to or from an I/O device. For example, if DATAIN is the address

of the input buffer associated with the keyboard, the instruction

```
Move    DATAIN,R0
```

reads the data from DATAIN and stores them into processor register R0. Similarly, the instruction

```
Move    R0,DATAOUT
```

sends the contents of register R0 to location DATAOUT, which may be the output data buffer of a display unit or a printer.

Most computer systems use memory-mapped I/O. Some processors have special In and Out instructions to perform I/O transfers. For example, processors in the Intel family described in Chapter 3 have special I/O instructions and a separate 16-bit address space for I/O devices. When building a computer system based on these processors, the designer has the option of connecting I/O devices to use the special I/O address space or simply incorporating them as part of the memory address space. The latter approach is by far the most common as it leads to simpler software. One advantage of a separate I/O address space is that I/O devices deal with fewer address lines. Note that a separate I/O address space does not necessarily mean that the I/O address lines are physically separate from the memory address lines. A special signal on the bus indicates that the requested read or write transfer is an I/O operation. When this signal is asserted, the memory unit ignores the requested transfer. The I/O devices examine the low-order bits of the address bus to determine whether they should respond.

- 1. **Memory-Mapped I/O Interfacing** :
  In this kind of interfacing, we assign a memory address that can be used in the same manner as we use a normal memory location.
- 2. **I/O Mapped I/O Interfacing** :
  A kind of interfacing in which we assign an 8-bit address value to the input/output devices which can be accessed using IN and OUT instruction is called I/O Mapped I/O Interfacing.

# Hardware required to connect an I/O device to the bus

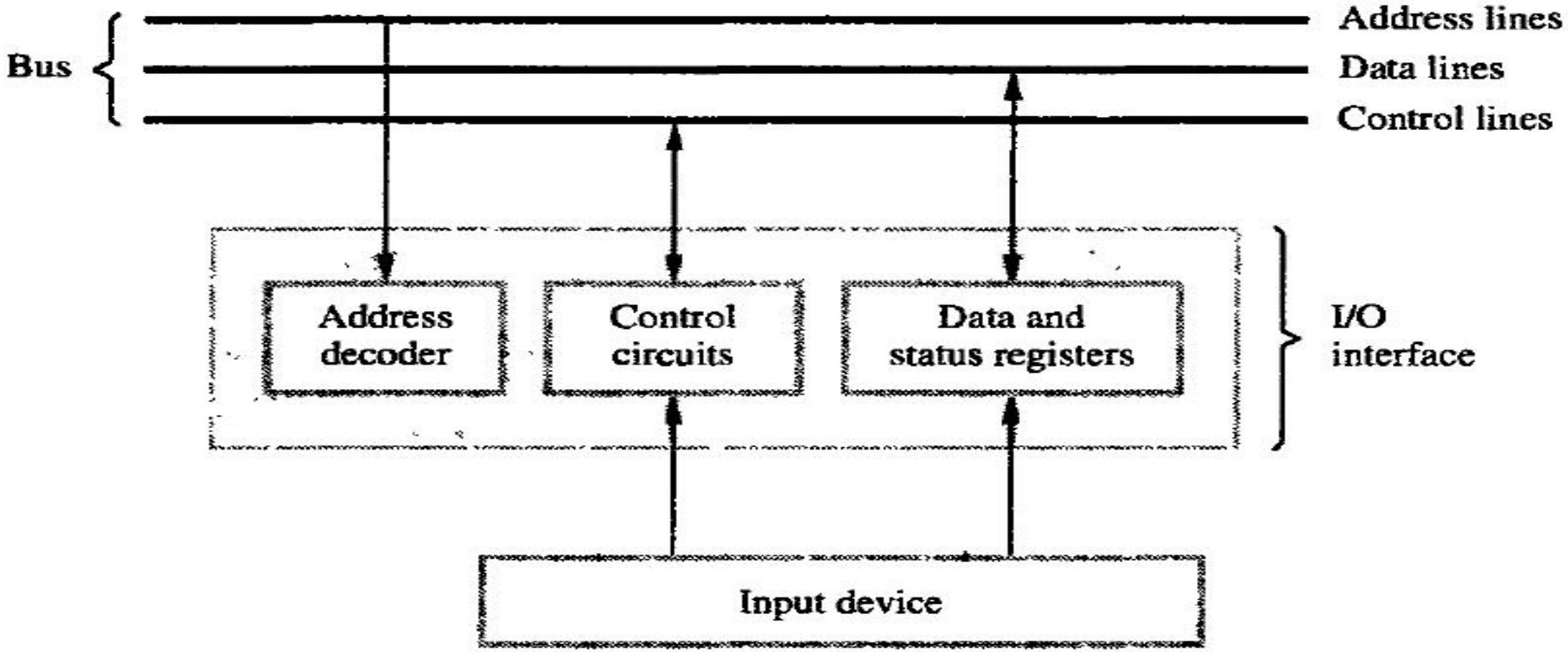



**Figure 4.2** I/O interface for an input device.

The address decoder enables the device to recognize its address when this address appears on the address lines. The data register holds the data being transferred to or from the processor. The status register contains information relevant to the operation of the I/O device. Both the data and status registers are connected to the data bus and assigned unique addresses. The address decoder, the data and status registers, and the control circuitry required to coordinate I/O transfers constitute the device's *interface circuit*.

I/O devices operate at speeds that are vastly different from that of the processor. When a human operator is entering characters at a keyboard, the processor is capable of executing millions of instructions between successive character entries. An instruction that reads a character from the keyboard should be executed only when a character is available in the input buffer of the keyboard interface. Also, we must make sure that an input character is read only once.

The basic ideas used for performing input and output operations were introduced in Section 2.7. For an input device such as a keyboard, a status flag, SIN, is included in the interface circuit as part of the status register. This flag is set to 1 when a character is entered at the keyboard and cleared to 0 once this character is read by the processor. Hence, by checking the SIN flag, the software can ensure that it is always reading valid data. This is often accomplished in a program loop that repeatedly reads the status register and checks the state of SIN. When SIN becomes equal to 1, the program reads the input data register. A similar procedure can be used to control output operations using an output status flag, SOUT.

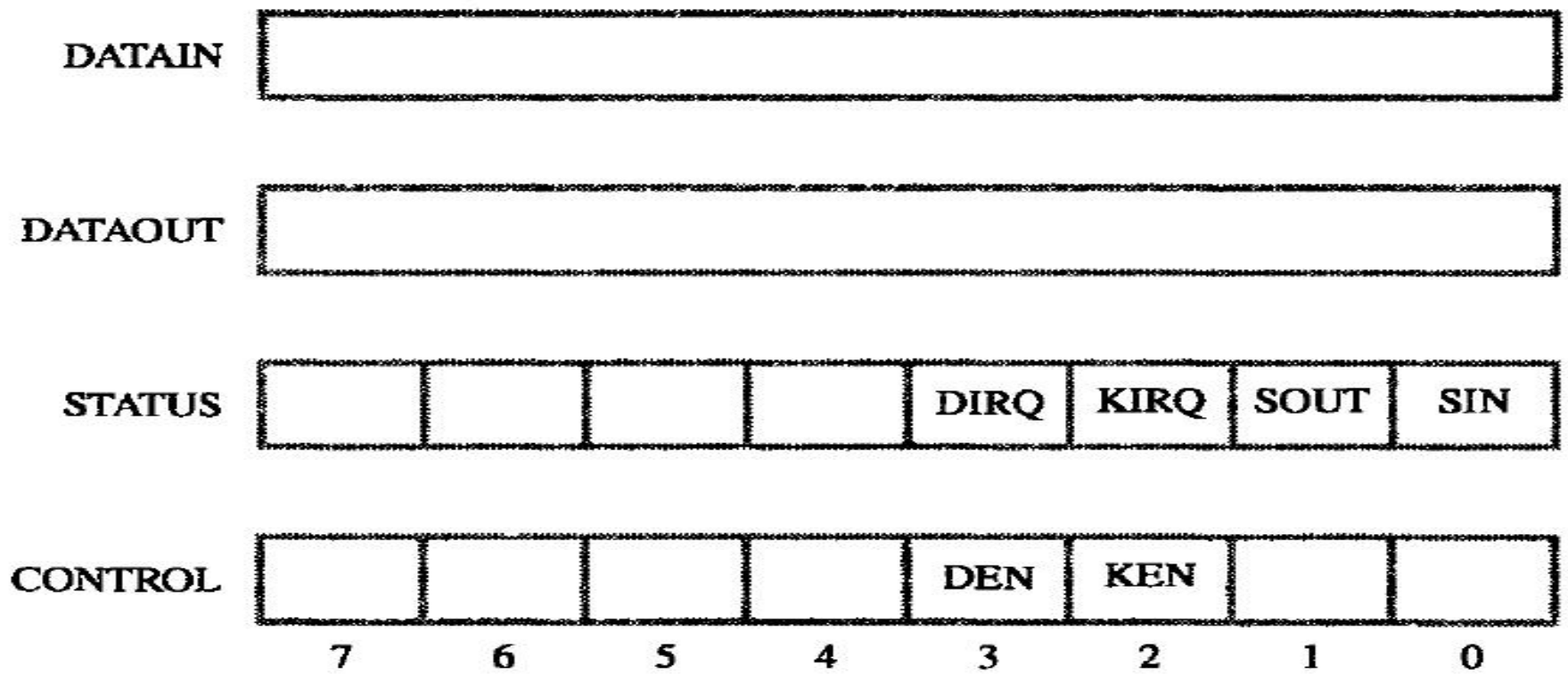



**Figure 4.3** Registers in keyboard and display interfaces.

```
        Move        #LINE,R0          Initialize memory pointer.
WAITK   TestBit     #0,STATUS         Test SIN.
        Branch=0    WAITK             Wait for character to be entered.
        Move        DATAIN,R1         Read character.
WAITD   TestBit     #1,STATUS         Test SOUT.
        Branch=0    WAITD             Wait for display to become ready.
        Move        R1,DATAOUT        Send character to display.
        Move        R1,(R0)+          Store charater and advance pointer.
        Compare     #$0D,R1           Check if Carriage Return.
        Branch≠0    WAITK             If not, get another character.
        Move        #$0A,DATAOUT      Otherwise, send Line Feed.
        Call        PROCESS           Call a subroutine to process the
                                        the input line.
```

**Figure 4.4** A program that reads one line from the keyboard, stores it in memory buffer, and echoes it back to the display.

The program in Figure 4.4 is similar to that in Figure 2.20. This program reads a line of characters from the keyboard and stores it in a memory buffer starting at location LINE. Then, it calls a subroutine PROCESS to process the input line. As each character is read, it is *echoed back* to the display. Register R0 is used as a pointer to the memory buffer area. The contents of R0 are updated using the Autoincrement addressing mode so that successive characters are stored in successive memory locations.

Each character is checked to see if it is the Carriage Return (CR) character, which has the ASCII code 0D (hex). If it is, a Line Feed character (ASCII code 0A) is sent to move the cursor one line down on the display and subroutine PROCESS is called. Otherwise, the program loops back to wait for another character from the keyboard.

This example illustrates *program-controlled I/O*, in which the processor repeatedly checks a status flag to achieve the required synchronization between the processor and an input or output device. We say that the processor *polls* the device. There are two other commonly used mechanisms for implementing I/O operations: interrupts and direct memory access. In the case of interrupts, synchronization is achieved by having the I/O device send a special signal over the bus whenever it is ready for a data transfer operation. Direct memory access is a technique used for high-speed I/O devices. It involves having the device interface transfer data directly to or from the memory, without continuous involvement by the processor. We will discuss these mechanisms

# Interrupts

- In **computer architecture**, an **interrupt** is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention.
- Our processor repeatedly keeps on checking for the SIN and SOUT bits for the synchronization in the program. Hence, the processor does not do any things useful other than running the infinite loop. To avoid this situation input/output devices can have the concept of interrupts . When the input/output device is ready it could signal the processor on a separate line called interrupt request line. On receiving the interrupt the processor reads the input output device and hence removing the infinite loop waiting mechanism.

## *For example let us take a task that involves two activities :*

1. **Perform some computation**
2. **Print the result**

Repeat the above two steps several times in the program, let the program contain 2 routines **COMPUTE** and **PRINT** routine.

**Method #1 :**

The COMPUTE routine passes N lines to the PRINT routine and the PRINT routine then prints the N lines one by one on a printer. All this time the COMPUTE routine keeps on waiting and does not do anything useful.

# Method #2 :

- The COMPUTE routine passes N lines to the PRINT routine. The PRINT routine then sends one line to the printer and instead of printing that line it execute itself and passes the control to the COMPUTE routine . The COMPUTE routine continuous it activity, once the line has been printed the printers sends an interrupt to the processor of the computer. At this point the COMPUTE routine is suspended and the PRINT routine is activated and the PRINT routine send second line to the printer so that the printer can keep on printing the lines and the process continues.

# TYPES OF INTERRUPTS

- **Hardware Interrupts**
- **Software Interrupts:**

# Hardware Interrupts

- If the signal for the processor is from external device or hardware is called hardware interrupts.
- **Example**: from keyboard we will press the key to do some action this pressing of key in keyboard will generate a signal which is given to the processor to do action, such interrupts are called hardware interrupts.

**Hardware interrupts can be classified into two types –।**

- **Maskable Interrupt**: The hardware interrupts which can be delayed when a much highest priority interrupt has occurred to the processor.
- **Non Maskable Interrupt**: The hardware which cannot be delayed and should process by the processor immediately.

# Software Interrupts

Software interrupt can also divided in to two types they are –

- **Normal Interrupts**: the interrupts which are caused by the software instructions are called **Normal Interrupts**.
- **Exception**: unplanned interrupts while executing a program is called Exception. For example: while executing a program if we got a value which should be divided by zero is called a exception.

# NEED FOR INTERRUPTS

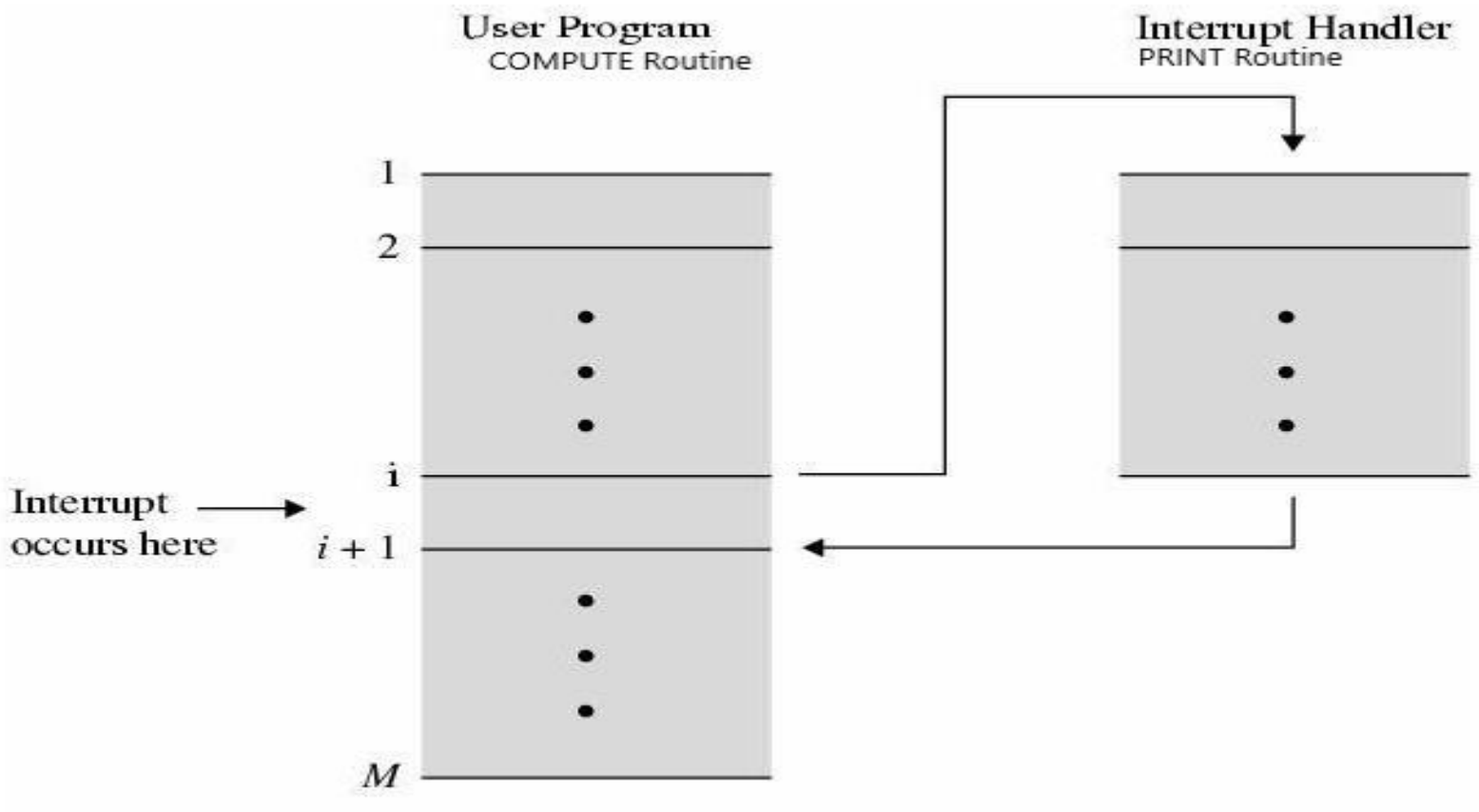
- The operating system is a reactive program

1. When you give some input it will perform computations and produces output but meanwhile you can interact with the system by interrupting the running process or you can stop and start another process

- This reactiveness is due to the interrupts
- Modern operating systems are interrupt driven

# INTERRUPT SERVICE ROUTINE AND IT'S WORKING

The routine that gets executed when an interrupt request is made is called as interrupt service routine.

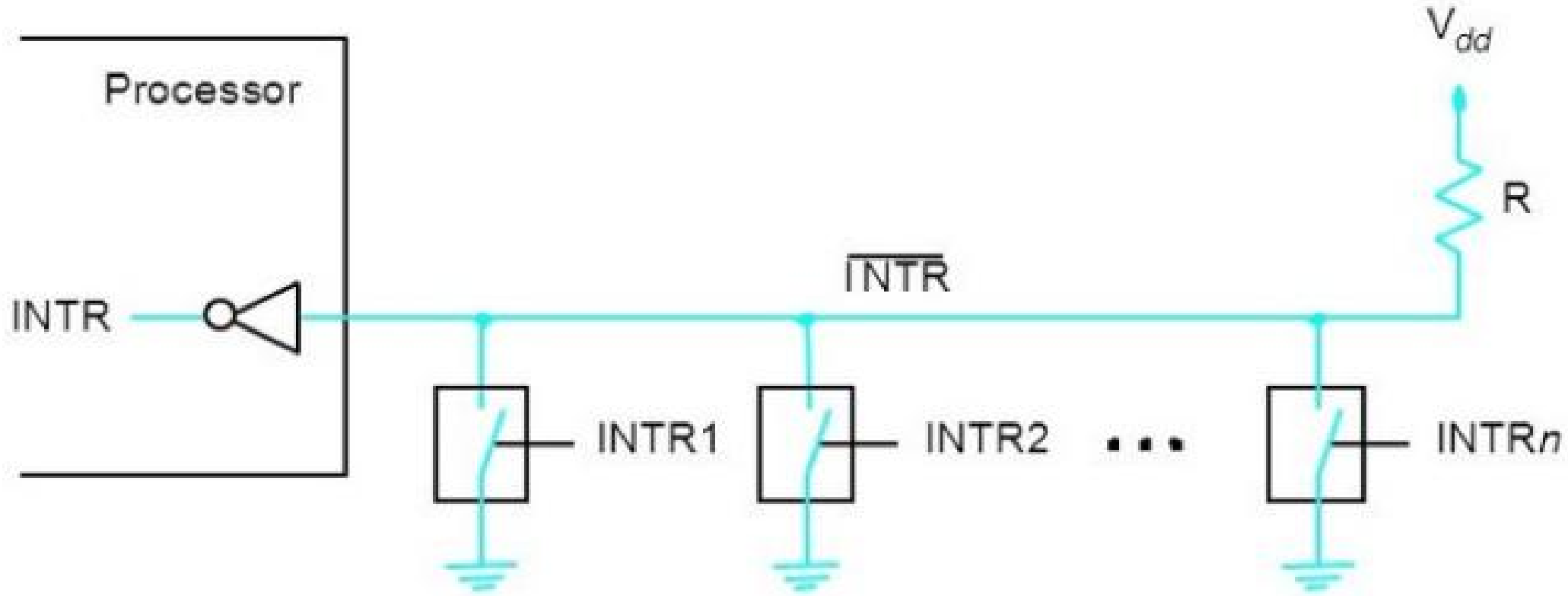


- **Step** 1: When the interrupt occurs the processor is currently executing **i'th** instruction and the program counter will be currently pointing to (**i** + 1)**th** instruction.
- **Step** 2: When the interrupt occurs the program counter value is stored on the processes stack.
- **Step** 3: The program counter is now loaded with the address of interrupt service routine.
- **Step** 4: Once the interrupt service routine is completed the address on the processes stack is pop and place back in the program counter.
- **Step** 5: Execution resumes from (**i** + 1)**th** line of COMPUTE routine.

# INTERRUPT HARDWARE

- Many computers have facility to connect two or more input and output devices.
- A single interrupt request line may be used to serve n devices.
- All devices are connected to the line via switches to ground.
- To request an interrupt a device closes its associated switch.
- Thus if all interrupt request signals INTR1 to INTRn are inactive, that is, if all switches are open, the voltage on the interrupt request line will be equal to Vdd. This is the inactive state of the line.
- When a device request an interrupt by closing its switch, the voltage on the line drops to 0, causing the interrupt request signal, INTR, received by the processor to go to 1.
- Since the closing of one or more switches will cause the line voltage to drop to 0, the value of INTR is the logical OR of the requests from individual devices, that is,
- INTR = INTR1 + …….+ INTRn

there is a common interrupt line for all N input/output devices



- The **resistor R** is called as a pull up resistor because it pulls the line voltage to high voltage state when all switches are open( no interrupt state).

# ENABLING AND DISABLING INTERRUPTS

- An interrupt can stop the currently executed program temporarily and branch to an interrupt service routine.
- An interrupt can occur at any time during the execution of a program.
- Because of the above reasons it may be some time necessary to disable the interrupt and enable it later on in the program. For this reason some processor may provide special machine instructions such as **interrupt enable** an **interrupt disable** that performs these tasks.

- The **first possibility** is to have the processor hardware ignore the interrupt request line until the execution of the first instruction of the interrupt service routine has been completed. Also note that the program written for the interrupt service routine has to enable and disable the interrupts in this case. This is done explicitly **by the programmer.**
- **The second option**, which is suitable for a **simple processor** with only one interrupt request line, is to have the processor automatically disable interrupts before starting the execution of the interrupt service routine.
- After saving the contents of the PC and the processor status register (PS) on the stack, the processor performs the equivalent of executing an **Interrupt disable instruction.**
- It is often the case that **one bit** in **the PS register** called **Interrupt enable**.

- After saving the contents of the PS on the stack, with the Interrupt enable bit equal to 1, the processor **clears the Interrupt-enable** bit in its PS register, thus disabling further interrupts.
- When a Return from interrupt is executed, the contents of the PS are **restored** from the stack, setting the Interrupt enable bit back to 1. Hence, interrupts are **again enabled**.
- In the **third option**, the processor has a special interrupt-request line for which the interrupt handling circuit responds only to **the leading edge** of the signal. Such a line is said to be **edge triggered**. In this case the processor will receive only one request.

# Handling an interrupt requests form a single device

- The device raise an interrupt request.
- The processor interrupts the program currently being executed.
- Interrupts are disabled by changing the control bits in the PS.
- The device is informed that its request has been recognized, and in response, it deactivates the interrupt request signal.
- The action requested by the interrupt is performed by the interrupt service routine.
- Interrupts are enabled and execution of the interrupted program is resumed.

# HANDLING MULTIPLE DEVICES

There could be scenarios when there are multiple input/output devices connected to the CPU that could be raising interrupts, since these interrupts are raised at a random time there can be several issues like-।

- How will the processor identify the device using the interrupt
- How will the processor handle two simultaneous interrupts
- Should the device be allowed to raise an interrupt while another interrupt service routine is already being executed

# How to identify the device raising the interrupts ?

- The status register can be used to identify the device using an interrupt. When a device raises an interrupt it will set a specific bit to one . That bit is called **IRQ**(**Interrupt ReQuest**) **.**
- **IRQs** are hardware lines over which devices can send interrupt signal to the microprocessor.
- When you add a new device to a PC you sometime need to set its **IRQ.**
- The simplest way to identify the interrupting device is to have the interrupt service routine **poll all the I/O devices connected to the bus.**
- The **first device encountered with its IRQ bit set** is the device that should be serviced.

- **DISADVANTAGES** -। A lot of time spent in checking for the IRQ bits of all the devices, considering that most of devices are generating interrupts at a random time.

# Vectored Interrupts

- To reduce the time involved in the polling process, a device requesting an interrupt may identify itself directly to the processor. This approach is called vectored interrupts.
- A device requesting an interrupt can identify itself by sending **a special code to the processor over the bus**. This enables the processor to identify individual devices even if they share a single interrupt request line.
- The code supplied by the device may represent the **starting address of the interrupt service routine for that device**. The code length is typically in the range of 4 to 8 bits. **The remainder of the address** is supplied by the processor based on the area in the memory where the address for interrupt service routines are located.

- The location pointed to by the interrupting device is used to store the **starting address of the interrupt service routine.** The processor reads this address called the **interrupt vector** and **loads it into the PC**. The interrupt vector may also include a new value for the processor status register.
- In most computers, I/O devices send the **interrupt-vector code** over the data bus, using the bus control signals to ensure that devices do not interfere with each other. When a device sends an interrupt request, the processor may not be ready to receive the interrupt-vector code immediately. For example, it must first complete the execution of the current instruction, which may require the use of the bus. There may be further delays if interrupts happen to be disabled at the time the request is raised. The interrupting device must wait to put data on the bus only when the processor is ready to receive it. When the processor is ready to receive the interrupt-vector code, it activates the interrupt-acknowledge line**, INTA**. The I/O device responds by sending its interrupt- vector code and turning off the INTR signal.

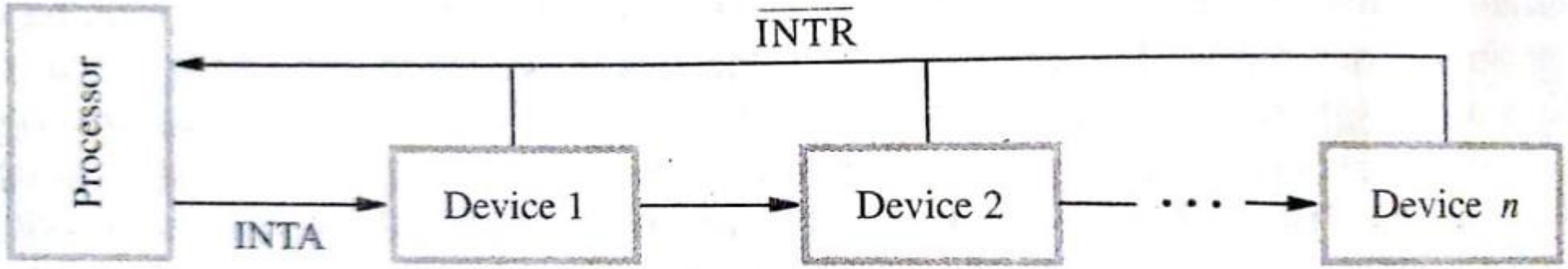
# Interrupt Nesting

- I/O devices should be organized in a **priority structure**. An interrupt request from a **highpriority** device should be accepted while the processor is servicing another request from a **lowerpriority** device. A multiple-level priority organization means that during execution of an interruptservice routine, interrupt requests will be accepted from some devices but not from others, **depending upon the device's priority.** To implement this scheme, priority level can be assigned to the processor that can be changed by the program.
  The processor's priority is usually encoded in a few bits of the **processor status word.** It **can be changed** by program **instructions that write into the PS**. These are **privileged instructions**, which can be executed only while the **processor is running** in the **supervisor mode**. The processor is in the supervisor mode only when executing **operating system routines**. It switches to the **user mode** before beginning to execute application programs. Thus, a user program cannot accidentally, or intentionally, change the priority of the processor and disrupt the system's operation. An attempt to execute a privileged instruction while in the user mode leads to

- **A multiple-priority scheme** can be implemented easily by using **separate interrupt- request and interrupt-acknowledge lines for each device**, Each of the interrupt request lines is assigned a **different priority level**. Interrupt requests received over these lines are sent to a priority arbitration circuit in the processor. A request is accepted only if it has a higher priority level than that currently assigned to the processor.

# DAISY CHAIN MECHANISM

- Suppose when there are multiple input/output devices raising an interrupt simultaneously then it is straight forward for us to select which interrupt to handle depending on the interrupt having the highest priority.




(a) Daisy chain

- **Step** 1: Multiple devices try to raise an interrupt by trying to pull down the interrupt request line(**INTR**).
- **Step** 2 : The processes realises that there are devices trying to raise an interrupt ,so it makes the **INTA** line goes high, is that it is set to 1.
- **Step** 3 : The **INTA** Line is connected to a device, device one in this case.

1. If this device one had raised an interrupt then it goes ahead and passes the identifying code to the data line.
2. If device one had not raise an interrupt then it passes the INTA signal to device two and so on.

So **priority is given to device nearest to the processor**. This method ensures that multiple interrupt request are handled properly, even when all the devices are connected to a common interrupt request line.

**Simultaneous request**: When simultaneous interrupt requests are arrived **from two or more I/O devices to the processor,** the processor must have some means of deciding which request to service first.

•The processor simply accepts the request having the **highest priority**. If several devices share one interrupt-request line,, some other mechanism is needed**.** **Polling** the status registers of the I/O devices is the **simplest such mechanism**. In this case, **priority** is determined by the **order in which the devices are polled**. When **vectored interrupts** are used, we must ensure that only **one device is selected to send its interrupt vector code.** A widely used scheme is to connect the devices to form a **daisy chain**,

- The interrupt request line INTR is common to all devices. The interrupt-acknowledge line, INTA, is connected in a daisy-chain fashion, such that the INTA signal propagates serially through the devices. When several devices raise an interrupt request and the INTR line is activated, the processor responds by setting the INTA line to 1. This signal is received by device 1. Device 1 passes the signal on to device 2 only if it does not require any service. If device 1 has a pending request for interrupt, it blocks the INTA signal and proceeds to put its identifying code on the data lines. Therefore, in the daisy-chain arrangement, the device that is electrically closest to the processor has the highest priority. The second device along the chain has second highest priority, and so on.

# Interrupt Latency

- When an interrupt occur, the service of the interrupt by executing the **ISR** may not start immediately by context switching. The time interval between the occurrence of interrupt and start of execution of the ISR is called interrupt latency.
- **Tswitch** = Time taken for context switch
- **ΣTexec** = The sum of time interval for executing the ISR
- **Interrupt Latency** = Tswitch + ΣTexec

- https://waliamrinal.medium.com/what-are-interrupts-in-computer-organisation-e23a223b3f75

# Direct memory access