Module: 1	**BASIC STRUCTURE OF COMPUTERS -** Functional units – Basic operational concepts – Bus structures. Memory locations and addresses – memory operations – instructions and instruction sequencing – addressing modes. **Basic processing unit -**Fundamental concepts –instruction cycle - execution of a complete instruction –single bus and multiple bus organization.

WHY COMPUTER ORGANIZATION AND ARCHITECTURE?

Computer architecture is a key component of computer engineering and it is concerned with all aspects of the design and organization of the central processing unit and the integration of the CPU into the computer system itself.

Architecture extends upward into computer software because a processor's architecture must cooperate with the operating system and system software. It is difficult to design an operating system well without knowledge of the underlying architecture.

Moreover, the computer designer must have an understanding of software in order to implement the optimum architecture.

INTRODUCTION

Computer: A device that accepts input, processes data, stores data, and produces output, all according to a series of stored instructions.

Software: A computer program that tells the computer how to perform particular tasks.

Hardware: Includes the electronic and mechanical devices that process the data; refers to the computer as well as peripheral devices.

Peripheral devices: Used to expand the computer's input, output and storage capabilities.

Network: Two or more computers and other devices that are connected, for the purpose of sharing data and programs.

Computer Types:Computers are classified based on the parameters likeSpeed of operation, Cost, Computational power and Type of application

Difference between computer organization and computer architecture

Architecture describes what the computer does and organization describes how it does it.

Computer organization:

Computer organization is concerned with the way the hardware components operate and the way they are connected together to form computer system. It includes Hardware details

transparent to the programmer such as control signal and peripheral. It describes how the computer performs. Example: circuit design, control signals, memory types this all are under computer organization.

Computer Architecture:

Computer architecture is concerned with the structure and behavior of computer system as seen by the user. It includes information, formats, instruction set and techniques for addressing memory. It describes what the computer does.

FUNCTIONAL UNITS:

The computer system is divided into five separate units for its operation.

- Input Unit.
- ALU.
- Control Unit.
- Memory Unit.
- Output Unit.

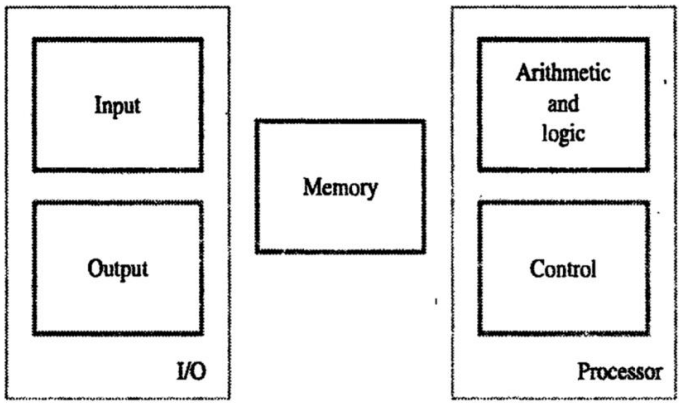


Figure 1

Input & Output unit

The method of feeding data and programs to a computer is accomplished by an input device. Computer input devices read data from a source, such as magnetic disks, and translate that data into electronic impulses [ADC] for transfer into the CPU. Some typical input devices are a keyboard, a mouse, scanner, etc.

Computer output devices converts the electronic impulses [DAC] into human readable form. Output unit sends processed results to the outside world. Examples: Display screens, Printers, plotters, microfilms, synthesizers, high-tech blackboards, film recorders, etc.

Memory Unit (MU)

A Memory Unit is a collection of storage cells together with associated circuits needed to transfer information in and out of storage. Data storage is a common term for archiving data or information in a storage medium for use by a computer. It's one of the basic yet fundamental functions performed by a computer. It's like a hierarchy of comprehensive storage solution for fast access to computer resources.

A computer stores data or information using several methods, which leads to different levels of data storage. Primary storage is the most common form of data storage which typically refers to the random access memory (RAM). It refers to the main storage of the computer because it holds data and applications that are currently in use by the computer. Then, there is

secondary storage which refers to the external storage devices and other external media such as hard drive and optical media.

Arithmetic Logical Unit (ALU)

After you enter data through the input device it is stored in the primary storage unit. Arithmetic Logical Unit performs the actual processing of data and instruction. The major operations performed by the ALU are addition, subtraction, multiplication, division, logic and comparison.

Data is transferred to ALU from storage unit when required. After processing, the output is returned back to storage unit for further processing or getting stored.

Control Unit

The next component of computer is the control unit, which acts like the supervisor seeing whether things are done in proper fashion. Control unit controls and coordinates the entire operations of the computer system.

The control unit determines the sequence in which computer programs and instructions are executed. Things like processing of programs stored in the main memory, interpretation of the instructions and issuing of signals for other units of the computer to execute them.

It also acts as a switch board operator when several users access the computer simultaneously. Thereby it coordinates the activities of computer's peripheral equipment as they perform the input and output. Therefore it is the manager of all operations.

Central Processing Unit (CPU)

The Arithmetic Logical Unit (ALU), Control Unit (CU) and Memory Unit (MU) of a computer system are jointly known as the central processing unit. We may call CPU as the brain of any computer system. It is just like a human brain that takes all major decisions, makes all sorts of calculations and directs different part of the computer by activating and controlling the operations.

BASIC OPERATIONAL CONCEPTS

To perform a given task, an appropriate program consisting of a list of instructions is stored in the memory. Individual instructions are brought from the memory into the processor, which executes the specified operations. [**Load** – Transfers data to register. **Store** – Transfers data to memory.] A typical instruction might be

Load LOC,R2	The operand at LOC is fetched from the memory into the processor. The operand is stored in register R2.
Add R1, R2, R3	Adds the contents of registers R1 and R2, then places their sum into register R3.
Store R4, LOC	This instruction copies the operand in register R4 to memory location
Add R1, R0	Add contents of R1, R0 and place the sum to R0.

The figure 2 shows how the memory and the processor can be connected. In addition to the ALU and the control circuitry, the processor contains a number of registers used for several different purposes.

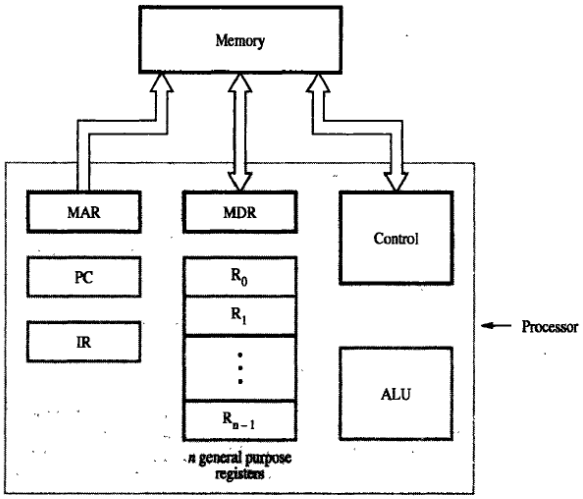


Figure 1.2 Connections between the processor and the memory.

The *instruction register (IR)* holds the instruction that is currently being executed. The *program counter (PC)* contains the memory address of the next instruction to be fetched and executed. In addition to the IR and PC, general-purpose registers R_0 through R_{n-1}, often called *processor registers*. They serve a variety of functions, including holding operands that have been loaded from the memory for processing.

Operating Steps

- Programs reside in the memory through input devices.
- PC is set to point to the first instruction.
- The contents of PC are transferred to MAR. A read signal is sent to the memory.
- The first instruction is read out and loaded into MDR.
- The contents of MDR are transferred to IR.
- Decode and execute the instruction. Get operands for ALU (Address to MAR – Read – MDR to ALU).
- Perform operation in ALU and Store the result back to general-purpose register.
- Transfer the result to memory (address to MAR, result to MDR – Write).
- During the execution, PC is incremented to the next instruction.

In addition to transferring data between the memory and the processor, the computer accepts data from input devices and sends data to output devices.

In order to respond immediately to some instruction, execution of the current program must be suspended. To cause this, the device raises an *interrupt signal*, which is a request for service by the processor. The processor provides the requested service by executing a program called an *interrupt-service routine.*

BUS STRUCTURES

The bus shown in Figure 3 is a simple structure that implements the interconnection network. Only one source/destination pair of units can use this bus to transfer data at any one time.

The bus consists of three sets of lines used to carry address, data, and control signals. I/O device interfaces are connected to these lines, as shown in Figure 4 for an input device. Each I/O

device is assigned a unique set of addresses for the registers in its interface. When the processor places a particular address on the address lines, it is examined by the address decoders of all devices on the bus. The device that recognizes this address responds to the commands issued on the control lines.

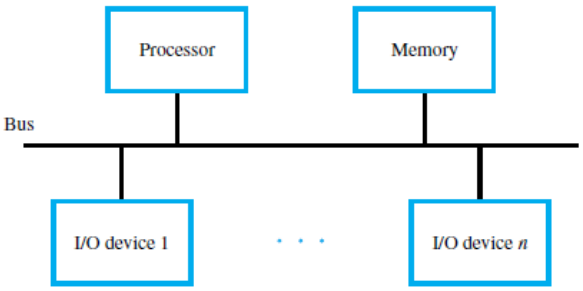


Figure 2

The processor uses the control lines to request either a Read or a Write operation, and the requested data are transferred over the data lines. When I/O devices and the memory share the same address space, the arrangement is called ***memory-mapped I/O.*** Any machine instruction that can access memory can be used to transfer data to or from an I/O device.

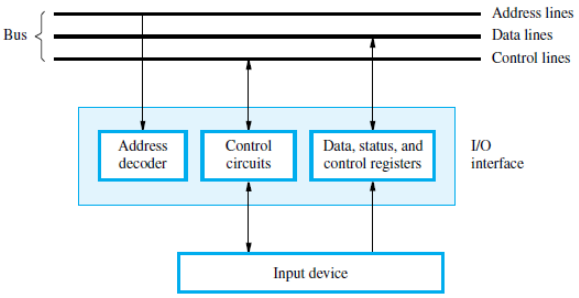


Figure 3

For example, if the input device is a keyboard and if DATAIN is its data register and DATAOUT may be the data register of a display device interface.

Load R2, DATAIN	reads the data from DATAIN and stores them into processor register R2
Store R2, DATAOUT	Sends the contents of register R2 to location DATAOUT.

The address decoder, the data and status registers, and the control circuitry required to coordinate I/O transfers constitute the device's interface circuit.

MEMORY LOCATIONS AND ADDRESSES

The memory consists of many millions of storage cells, each of which can store a bit of information having the value 0 or 1. The memory is organized so that a group of n bits can be stored or retrieved in a single, basic operation. Each group of n bits is referred to as a ***word*** of information, and n is called the ***word length***.

The memory of a computer can be schematically represented as a collection of words, as shown in Figure 5.Modern computers have word lengths that typically range from 16 to 64 bits.

Aunit of 8 bits iscalled a **byte**. Machine instructions may require one or more words for their representation. Accessing the memory to store or retrieve a single item of information, either a wordor a byte, requires distinct names or addresses for each location.

The memory can have up to 2k addressable locations.The2k addresses constitute the address space of the computer.

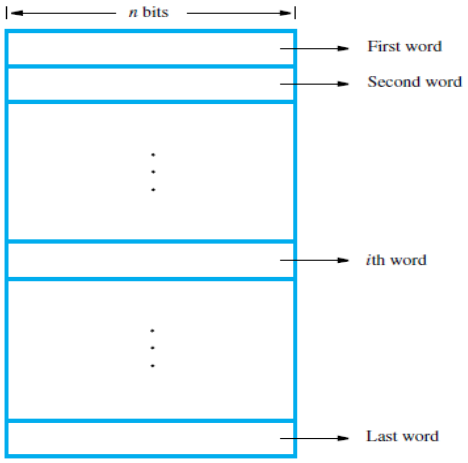


Figure 4

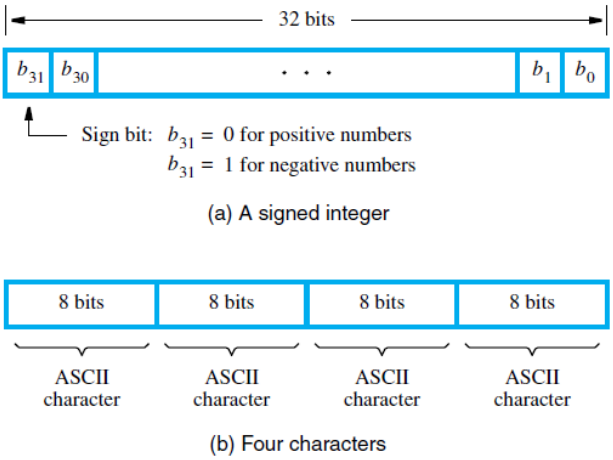


Figure 5

Byte Addressability

We now have three basic information quantities to deal with: **bit, byte, and word**. A byte is always 8 bits, but the word length typically ranges from 16 to 64 bits. It is impractical to assign distinct addresses to individual bit locations in the memory. The most practical assignment is to have successive addresses refer to successive byte locations in the memory.

The term byte-addressable memory is used for this assignment. Byte locations have addresses 0, 1, 2 . . . Thus, if the word length of the machine is 32 bits, successive words are located at addresses 0, 4, 8… with each word consisting of four bytes.

Big-Endian and Little-Endian Assignments

There are two ways that byte addresses can be assigned across words. The name **big-endian** is used when lower byte addresses are used for the more significant bytes (the leftmost bytes) of the word. The name **little-endian** is used for the opposite ordering, where the lower byte addresses are used for the less significant bytes (the rightmost bytes) of the word.

In both cases, byte addresses 0, 4, and 8… are taken as the addresses of successive words in the memory of a computer with a 32-bit word length. These are the addresses used when accessing the memory to store or retrieve a word.

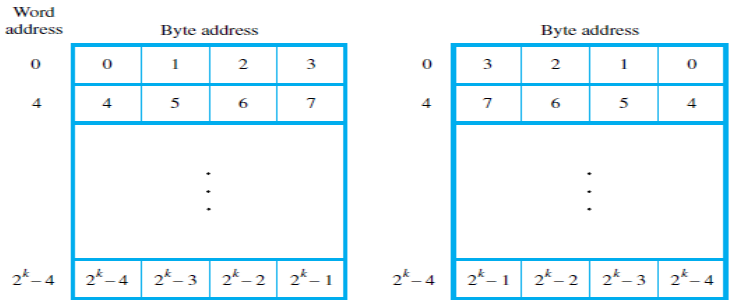


Figure 6

Word Alignment

In the case of a 32-bit word length, natural word boundaries occur at addresses 0, 4, 8… We say that the word locations have aligned addresses if they begin at a byte address that is a multiple of the number of bytes in a word. For practical reasons associated with manipulating binary-coded addresses, the number of bytes in a word is a power of 2. Hence, if the word length is 16 (2 bytes), aligned words begin at byte addresses 0, 2, 4... and for a word length of 64 (23 bytes), aligned words begin at byte addresses 0, 8, 16…

Accessing Numbers and Characters

A **number** usually occupies one word, and can be accessed in the memory by specifying its word address. Similarly, individual **characters** can be accessed by their byte address. For programming convenience it is useful to have different ways of specifying addresses in program instructions.

MEMORY OPERATIONS

Both program instructions and data operands are stored in the memory. Two basic operations involvingthe memory are needed, namely, Read and Write.The Read operation transfers a copy of the contents of a specific memory location tothe processor. The memory contents remain unchanged.

To start a Read operation, theprocessor sends the address of the desired location to the memory and requests that itscontents be read. The memory reads the data stored at that address and sends them to theprocessor.

The Write operation transfers an item of information from the processor to a specificmemory location, overwriting the former contents of that location. To initiate a Writeoperation, the processor sends the address of the desired location to the memory, togetherwith the data to be written into that location. The memory then uses the address and datato perform the write.

INSTRUCTIONS AND INSTRUCTION SEQUENCING

A computer must have instructions capable of performing four types of operations:

- Data transfers between the memory and the processor registers
- Arithmetic and logic operations on data
- Program sequencing and control
- I/O transfers

Begin by discussing instructions for the first two types of operations. To facilitate the discussion, we first need some notation.

Register Transfer Notation

To describe the transfer of information, the contents of any location are denoted by placing square brackets around its name.

$$R1 \leftarrow [LOC]$$

Thus, this expression means that the contents of memory location LOC are transferred into processor register R1.

As another example, consider the operation that adds the contents of registers R1 and R2, and places their sum into register R3. This action is indicated as

$$R3 \leftarrow [R1] + [R2]$$

This type of notation is known as **Register Transfer Notation (RTN).** Note that the righthandside of an RTN expression always denotes a value, and the left-hand side is the name of a location where the value is to be placed, overwriting the old contents of that location.

Assembly-Language Notation

We need another type of notation to represent machine instructions and programs. For this, we use **assembly language**. For example, a generic instruction that causes the transfer described above, from memory location LOC to processor register R1, is specified by the statement

Move LOC, R1

The contents of LOC are unchanged by the execution of this instruction, but the old contents of register R1 are overwritten.

The second example of adding two numbers contained in processor registers R1 and R2 and placing their sum in R3 can be specified by the assembly-language statement

Add R1, R2, R3

In this case, registers R1 and R2 hold the **source** operands, while R3 is the **destination**.

Basic Instruction Types

(1)Three Address Instruction:

Operation Source1, Source 2, Destination

Add A, B, C

Operand A and B are source operands, C is the destination operand. Add is the operation to be performed on the operands.

(2)Two Address Instruction:

Operation Source, Destination

Add A, B

Performs the operation B ← [A] + [B]. When the sum is calculated, the result is sent to memory and stored in location B, replacing the original contents of this location. This means operand B is both source and destination.

The **problem** of adding the contents of location A and B without destroying either of them. and to place the sum in location C is solved using the **Move** instruction. [Move works same as Copy]. **Move Source, Destination**.

Move B,C

Add A,C

(3)One Address Instruction:

A **processor register** called **Accumulator** is used.

Add A - Add the contents of memory location A to the contents of accumulator register and place the sum back to the **accumulator**.

Load A - Load instruction copies the content of memory location A into **accumulator**

Store A -Store instruction copies the content of accumulator into **memory location A**

Instruction Execution and Straight-Line Sequencing

Let's consider task C = A + B, implemented as C←[A] + [B]. Figure 8 shows a possible program segment for this task as it appears in the memory of a computer. We assume that the word length is 32 bits and the memory is byte-addressable. The three instructions of the program are in successive word locations, starting at location i.

Since each instruction is 4 bytes long, the second and third instructions are at addresses i + 4 and i + 8. Let us consider how this program is executed.

- To begin executing a program, the address of its first instruction (i in our example) must be placed into the PC.
- Then, the processor control circuits use the information in the PC to fetch and execute instructions, one at a time, in the order of increasing addresses. This is called **straight-line sequencing.**
- During the execution of each instruction, the PC is incremented by 4 to point to the next instruction. Thus, after the Move instruction at location i + 8 is executed, the PC contains the value i + 12, which is the address of the first instruction of the next program segment.

Executing a given instruction is a two-phase procedure. In the **first phase**, called **instruction fetch**, the instruction is fetched from the memory location whose address is in the PC. This instruction is placed in the instruction register (IR) in the processor.

At the start of the **second phase**, called **instruction execute**, the instruction in IR is examined to determine which operation is to be performed. The specified operation is then performed by the processor. This involves a small number of steps such as fetching operands from the memory or from processor registers, performing an arithmetic or logic operation, and storing the result in the destination location.

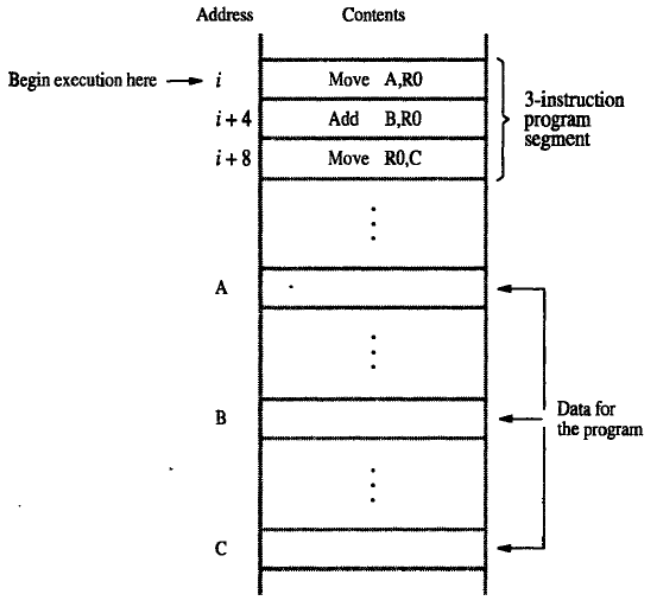


Figure 8 A program for C ← [A] + [B].

At some point during this two-phase procedure, the contents of the PC are advanced to point to the next instruction. When the execute phase of an instruction is completed, the PC contains the address of the next instruction, and a new instruction fetch phase can begin.

Branching

Consider the task of adding a list of n numbers. LOOP is a straight line sequence of instructions executed as many times as needed. Assume that the number of entries in the list, n, is stored in memory location N. Register R1 is used as a counter to determine the number of times the loop is executed. Hence, the contents of location N are loaded into register R1 at the beginning of the program. Then, within the body of the loop, the instruction **Decrement R1** reduces the contents of R1 by 1 each time through the loop. Execution of the loop is repeated as long as the content of R1 is greater than zero.

We now introduce branch instructions. This type of instruction loads a new address into the program counter. As a result, the processor fetches and executes the instruction at this new address, called the **branch target**, instead of the instruction at the location that follows the branch instruction in sequential address order.

A **conditional branch** instruction causes a branch only if a specified condition is satisfied. If the condition is not satisfied, the PC is incremented in the normal way, and the next instruction in sequential address order is fetched and executed.

In the program in Figure 2.10, the instruction

Branch>0 LOOP is same as

Branch_if_[R1]>0 LOOP

is a conditional branch instruction that causes a branch to location LOOP if the contents of register R1 are greater than zero. This means that the loop is executed as long as there are entries in the list that are yet to be added to R0. At the end of the n^{th} pass through the loop, the Decrement instruction produces a zero and hence branching does not occur.

Move instruction is fetched and executed. It moves the final result from R0 into memory location SUM.

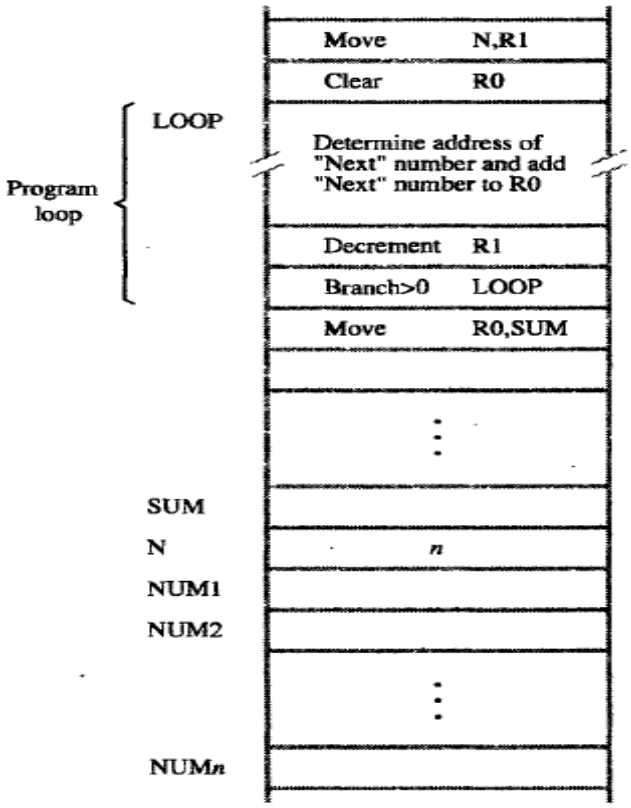


Figure 2.10 Using a loop to add *n* numbers.

Condition Codes

The processor keeps track of instruction about the results of various operations for use by subsequent conditional branch instructions. This is accomplished by recording the required information in individual bits often called as **conditional code flags**. These flags are usually grouped together in a special processor register called the condition **code register or status register**. Individual condition codes are set to 1 or cleared to 0, depending upon the outcome of the operation performed.

Four commonly used flags are

N (negative)	Set to 1 if the result is negative; otherwise, cleared to 0
Z (zero)	Set to 1 if the result is 0; otherwise, cleared to 0
V (overflow)	Set to 1 if arithmetic overflow occurs; otherwise, cleared to 0
C (carry)	Set to 1 if a carry-out results from the operation; otherwise, cleared to 0

ADDRESSING MODES

The different ways for specifying the locations of instruction operands are known as **addressing modes.**

1. Implementation of Variables and Constants

Variables are found in almost every computer program. In assembly language, a variable is represented by allocating a register or a memory location to hold its value. This value can be changed as needed using appropriate instructions.

Register mode: The operand is the contents of a processor register; the name (address) of the register is given in the instruction.

Example: The instruction ***Add R1, R2, R3*** uses the Register mode for all three operands. Registers R1 and R2 hold the two source operands, while R3 is the destination.

Absolute/Direct mode: The operand is in a memory location; the address of this location is given explicitly in the instruction.

Example: The Absolute mode is used in the instruction ***Move LOC, R1*** which copies the value in the memory location LOC into register R1.

Immediate mode: The operand is given explicitly in the instruction.

Example: The instruction ***Add #200, R1, R2*** adds the value 200 to the contents of register R1, and places the result into register R2. A common convention is to use the **number sign (#)** in front of the value to indicate that this value is to be used as an **immediate** operand.

2. Indirection and Pointers

In the addressing modes that follow, the instruction does not give the operand or its address explicitly. Instead, it provides information from which the memory address of the operand can be determined. This address is known as **Effective Address (EA)** of the operand.

Indirect mode: The effective address of the operand is the contents of a register or memory location whose address appears in the instruction. We denote indirection by placing the name of the register given in the instruction in **parentheses ()**.

To execute the Add instruction in Figure 2.11(a), the processor uses the value B, which is in register R1, as the effective address of the operand. It requests a read operation from the memory to read the contents of location B. The value read is the desired operand, which the processor adds to the contents of register R0. Indirect addressing through a memory location is also possible as shown in Figure 2.11(b). In this case the processor first reads the contents of memory location A, then request a second read operation using the value B as an address to obtain the operand.

The register or memory location that contains the address of an operand is called a **pointer.**

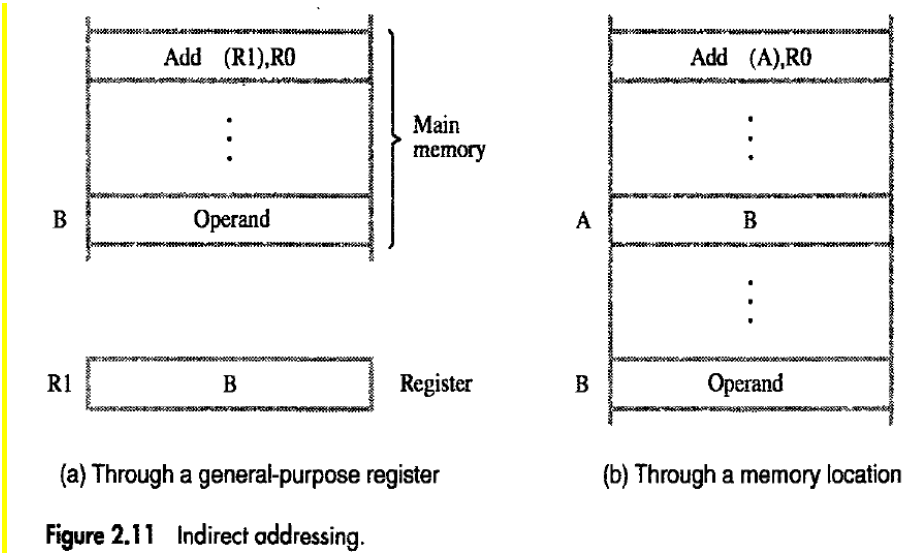


Figure 2.11 Indirect addressing.

3. Indexing and Arrays

The next addressing mode we discuss provides a different kind of flexibility for accessing operands. It is useful in dealing with lists and arrays.

Index mode: The effective address of the operand is generated by adding a constant value to the contents of a register. The register used in this mode is referred as the **index register**.

We indicate the Index mode symbolically as $\mathbf{X(R_i)}$ where X denotes a constant signed integer value contained in the instruction and R_i is the name of the register involved. The effective address of the operand is given by $\mathbf{EA = X + [R_i]}$. The contents of index register are not changed in the process of generating the effective address.

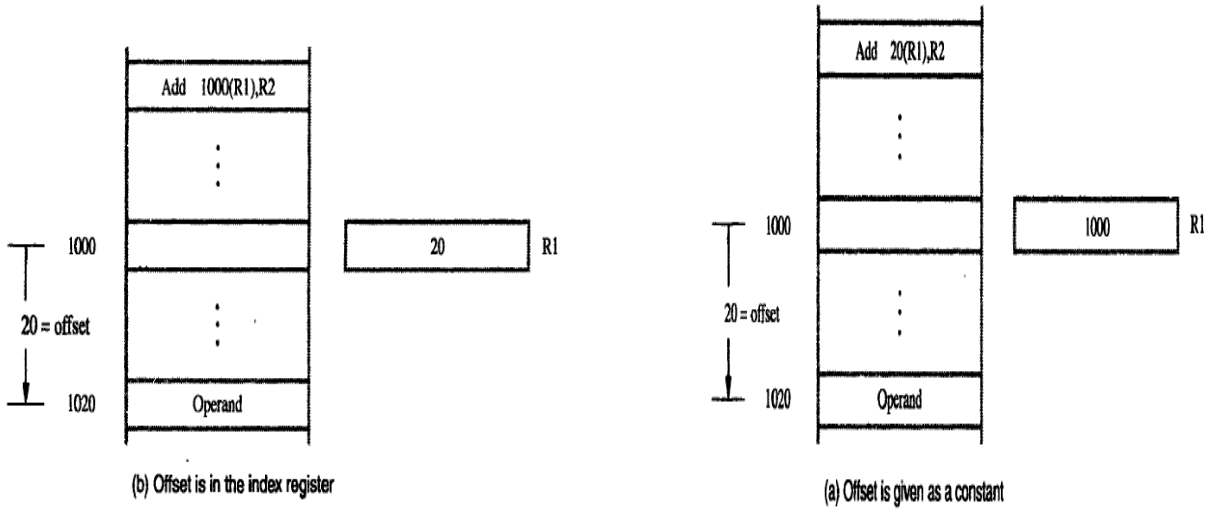


Figure 2.13 Indexed addressing.

Figure 2.13 illustrates two ways of using the index mode. In Figure 2.13(a), the index register R1 contains the address of memory location and the value X defines an offset (displacement) from this address to the location where the operand is found. Figure 2.13(b), Here the constant X corresponds to a memory address and the contents of the index register define the offset of the operand. in either case, the effective address is the sum of two values; one is given explicitly in the instruction and the other is stored in a register.

Base with index: A second register may be used to contain the offset X, in which case the index mode is written as, **(R_i,R_j).** The effective address is the sum of the contents of register R_i and R_j.The second register is called as **base register**. This form of addressing provides more **flexibility** in accessing operands, because both components of the effective address can be changed.

Base with index and offset: Uses two registers plus a constant denoted as **$X(R_i,R_j)$.** The effective address is the sum of the constant X and the contents of register R_i and R_j. This added flexibility is useful in accessing multiple components inside each item in a record, where the beginning of an item is specified by **(R_i,R_j) part of the addeessing mode.**

4. Relative Addressing

In index addressing, if the program counter PC, is used instead of a general-purpose register then X(PC) can be used to address a memory location that is X bytes away from the location presently pointed to by the program counter. Since the addressed location is identified "relative" to the program counter, which always identifies the current execution point in a program, the name Relative mode is associated with this type of addressing.

Relative mode: The effective address is determined by the Index mode using the program counter in place of the general-purpose register R_i.

This mode can be used to access data operands. But, its most common use is to specify the target address in branch instructions. An instruction such as **Branch>0 LOOP** causes program execution to go to the branch target location identified by the name LOOP if the branch condition is satisfied.

5. Additional Modes

Many computers provide additional modes intended to aid certain programming tasks. The two modes described next are useful for accessing data items in successive locations in the memory.

Auto-increment mode: The effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are automatically incremented to point to the next item in a list.

We denote the Auto-increment mode by putting the specified register in parentheses, to show that the contents of the register are used as the effective address, followed by a plus sign to

indicate that these contents are to be incremented after the operand is accessed. Thus, the Auto-increment mode is written as **$(R_i)+$**

Auto-decrement mode: The contents of a register specified in the instruction are first automatically decremented and are then used as the effective address of the operand.

The Auto-increment mode is written as **$-(R_i)$**

Table 2.1 Generic addressing modes

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	*Ri*	EA = *Ri*
Absolute (Direct)	LOC	EA = LOC
Indirect	(*Ri*) (LOC)	EA = [*Ri*] EA = [LOC]
Index	X(*Ri*)	EA = [*Ri*] + X
Base with index	(*Ri*,*Rj*)	EA = [*Ri*] + [*Rj*]
Base with index and offset	X(*Ri*,*Rj*)	EA = [*Ri*] + [*Rj*] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(*Ri*)+	EA = [*Ri*]; Increment *Ri*
Autodecrement	−(*Ri*)	Decrement *Ri*; EA = [*Ri*]

EA = effective address
Value = a signed number

BASIC PROCESSING UNIT - SOME FUNDAMENTAL CONCEPTS

To execute a program, the processor fetches one instruction at a time and performs the operations specified. Instructions are fetched from successive memory locations until a branch or a jump instruction is encountered.

The processor keeps track of the address of the memory location containing the next instruction to be fetched using the program counter, PC. After fetching an instruction, the contents of the PC are updated to point to the next instruction in the sequence. A branch instruction may load a different value into the PC. Another key register in the processor is the instruction register, IR.

Suppose that each instruction comprises 4 bytes, and that it is stored in one memory word. To execute an instruction, the processor has to perform the following three *steps:

1. Fetch the contents of the memory location pointed to by the PC. The contents of this location are the instruction to be executed; hence they are loaded into the IR. In register transfer notation, the required action is

   IR←[[PC]]

2. Increment the PC to point to the next instruction. Assuming that the memory is byte addressable, the PC is incremented by 4; that is
   PC←[PC] + 4
3. Carry out the operation specified by the instruction in the IR.

Fetching an instruction and loading it into the IR is usually referred to as the ***instruction fetch phase***. Performing the operation specified in the instruction constitutes the ***instruction execution phase.***

Single Bus organization of Processor

Figure shows the organization in which the arithmetic and logic unit (ALU) and all the registers are interconnected via a single common bus. This bus is internal to the processor and should not be confused with the external bus that connects the processor to the memory and I/O devices.

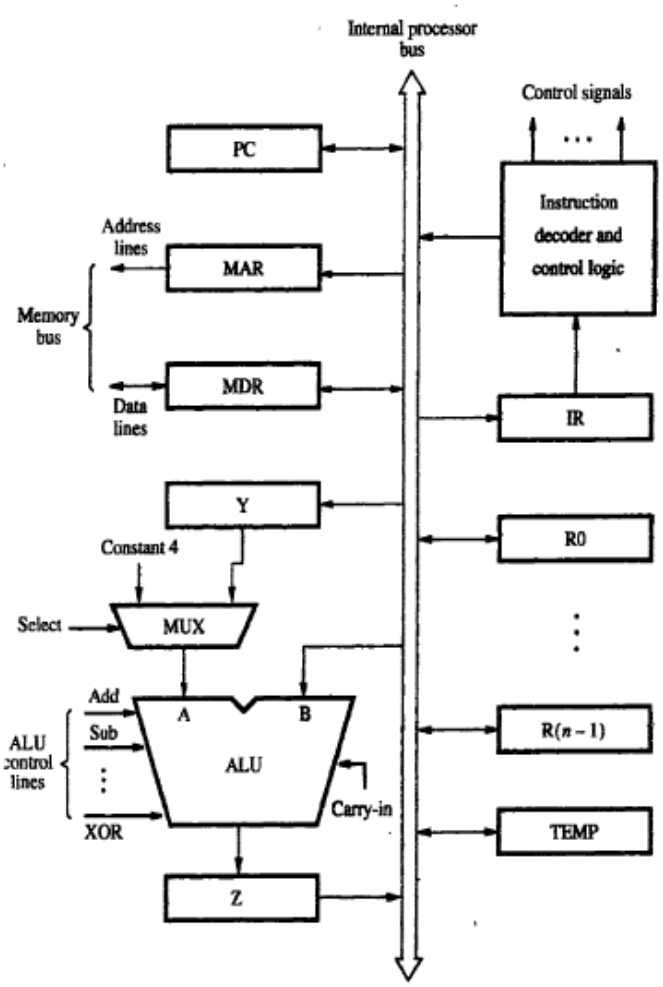


Figure 7: Single Bus Organization

The data and address lines of the external memory bus are connected to the internal processor bus via the memory data register, MDR, and the memory address register, MAR,

respectively. Register MDR has two inputs and two outputs. Data may be loaded into MDR either from the memory bus or from the internal processor bus.

The data stored in MDR may be placed on either bus. The input of MAR is connected to the internal bus, and its output is connected to the external bus. The control lines of the memory bus are connected to the instruction decoder and control logic block.

Three registers Y, Z, and TEMP registers are used by the processor for temporary storage during execution of some instructions. The multiplexer MUX selects either the output of register Y or a constant value 4 to be provided as input A of the ALU. The constant 4 is used to increment the contents of the program counter.

With few exceptions, an instruction can be executed by performing one or more of the following operations in some specified sequence:

- Transfer a word of data from one processor register to another or to the ALU
- Perform an arithmetic or a logic operation and store the result in a processor register
- Fetch the contents of a given memory location and load them into a processor register
- Store a word of data from a processor register into a given memory location

Register Transfers

Instruction execution involves a sequence of steps in which data are transferred from one register to another. For each register, two control signals are used to place the contents of that register on the bus or to load the data on the bus into the register.

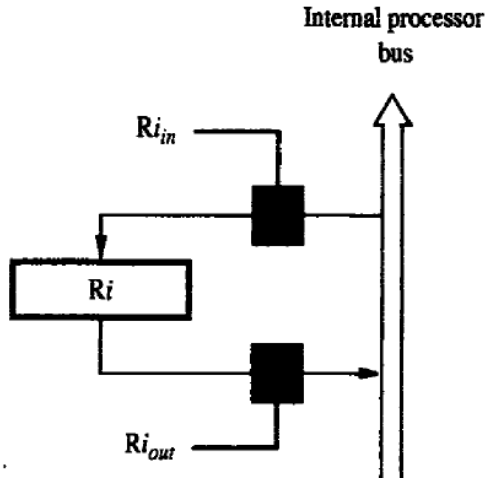


Figure 8

The input and output of register Ri are connected to the bus via switches controlled by the signals Ri_{in} and Ri_{out}, respectively. When Ri_{in} is set to 1, the data on the bus are loaded into Ri. Similarly, when Ri_{out}, is set to 1, the contents of register Ri are placed on the bus. While Ri_{out} is equal to 0, the bus can be used for transferring data from other registers.

Suppose that we wish to transfer the contents of register R1 to register R4. This can be accomplished as follows:

- Enable the output of register R1 by setting $R1_{out}$ to 1. This places the contents of R1 on the processor bus.
- Enable the input of register R4 by setting $R4_{in}$, to 1. This loads data from the processor bus into register R4.

7All operations and data transfers within the processor take place within time periods defined by the ***processor clock.***

Performing an Arithmetic or Logic Operation

The ALU is a combinational circuit that has no internal storage. It performs arithmetic and logic operations on the two operands applied to its A and B inputs, one of the operands is the output of the multiplexer MUX and the other operand is obtained directly from the bus. The result produced by the ALU is stored temporarily in register Z.

Therefore, a sequence of operations to add the contents of register R1 to those of register R2 and store the result in register R3 is

1. $R1_{out}$, Y_{in}
2. $R2_{out}$, Select Y, Add, Z_{in}
3. Z_{out} $R3_{in}$

Step 1:The output of register R1 and the input of register Y are enabled, causing the contents of R1 to be transferred over the bus to Y.

Step 2:The multiplexer's Select signal is set to SelectY, causing the multiplexer to gate the contents of register Y to input A of the ALU. At the same time, the contents of register R2 are gated onto the bus and, hence, to input B. The function performed by the ALU depends on the signals applied to its control lines. In this case, the Add line is set to 1, causing the output of the ALU to be the sum of the two numbers at inputs A and B. This sum is loaded into register Z because its input control signal is activated.

Step 3:The contents of register Z are transferred to the destination register, R3. This last transfer cannot be carried out during step 2, because only one register output can be connected to the bus during any clock cycle.

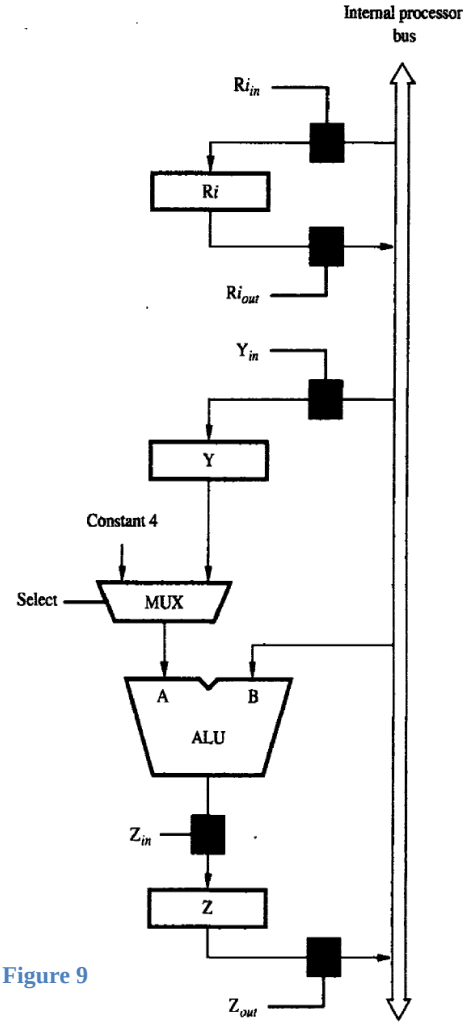


Figure 9

Fetching a Word from Memory

To fetch a word of information from memory, the processor has to specify the address of the memory location where this information is stored and request a Read operation. The connections for register MDR are illustrated in Figure 4.

It has four control signals: MDR_{in} and MDR_{out}, control the connection to the internal bus, and MDR_{inE} and MDR_{outE} control the connection to the external bus.

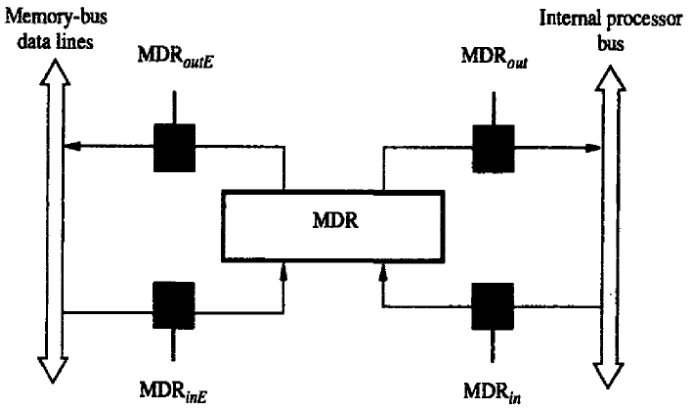


Figure 10

As an example of a read operation, consider the instruction Move (R1),R2. The actions needed to execute this instruction are:

1. MAR ← [RI]
2. Start a Read operation on the memory bus
3. Wait for the MFC(Memory Function Completed) response from the memory
4. Load MDR from the memory bus
5. R2 ← [MDR]

These actions may be carried out as separate steps, but some can be combined into a single step. Each action can be completed in one clock cycle, except action 3 which requires one or more clock cycles, depending on the speed of the addressed device.

The memory read operation requires three steps, which can be described by the signals being activated as follows:

1. Rl_{out}, MAR_{in}, Read
2. MDR_{inE}, WMFC
3. MDR_{out}, $R2_{in}$

where WMFC is the control signal that causes the processor's control circuitry to wait for the arrival of the MFC signal.

Storing a word in Memory

Writing a word into a memory location follows a similar procedure. The desired address is loaded into MAR. Then, the data to be written are loaded into MDR, and a Write command is issued. Hence, executing the instruction Move R2,(R1) requires the following sequence: 1.

1. $R1_{out}$, MAR_{in}
2. $R2_{out}$, MDR_{in}, Write
3. MDR_{outE}, WMFC

As in the case of the read operation, the Write control signal causes the memory bus interface hardware to issue a Write command on the memory bus. The processor remains in step 3 until the memory operation is completed and an MFC response is received.

EXECUTION OF A COMPLETE INSTRUCTION

Consider the instruction **Add (R3),R1** which adds the contents of a memory location pointed to by R3 to register R 1.

Executing this instruction requires the following actions:

1. Fetch the instruction.
2. Fetch the first operand (the contents of the memory location pointed to by R3).
3. Perform the addition.
4. Load the result into RI.

Instruction execution proceeds as follows.

Step 1: The instruction fetch operation is initiated by loading the contents of the PC into the MAR and sending a Read request to the memory. The Select signal is set to Select4, which causes the multiplexer MUX to select the constant 4. This value is added to the operand at input B, which is the contents of the PC, and the result is stored in register Z.

Step 2: The updated value is moved from register Z back into the PC, while waiting for the memory to respond.

Step	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	$R3_{out}$, MAR_{in}, Read
5	$R1_{out}$, Y_{in}, WMFC
6	MDR_{out}, SelectY, Add, Z_{in}
7	Z_{out}, $R1_{in}$, End

Figure 11

Step 3:The word fetched from the memory is loaded into the IR.

*(Steps 1 through 3 constitute the instruction **fetch phase**, which is the same for all instructions.)*

Step 4: The instruction decoding circuit interprets the contents of the IR. This enables the control circuitry to activate the control signals for *steps 4 through 7, which constitute the **execution phase***. The contents of register R3 are transferred to the MAR in step 4, and a memory read operation is initiated.

Step 5: the contents of R1 are transferred to register Y, to prepare for the addition operation.

Step 6: When the Read operation is completed, the memory operand is available in register MDR, and the addition operation is performed. The contents of MDR are gated to the bus, and thus also to the B input of the ALU, and register Y is selected as the second input to the ALU by choosing SelectY.

Step 7: The sum is stored in register Z, and then transferred to R1. The End signal causes a new instruction fetch cycle to begin by returning to step 1.

This discussion accounts for all control signals in Figure 11 except Yin in step 2. There is no need to copy the updated contents of PC into register Y when executing the Add instruction.

But, in Branch instructions the updated value of the PC is needed to compute the Branch target address.

To speed up the execution of Branch instructions, this value is copied into register Y in step 2. Since step 2 is part of the fetch phase, the same action will be performed for all instructions. This does not cause any harm because register Y is not used for any other purpose at that time.

Branch Instruction

A branch instruction replaces the contents of the PC with the branch target address. This address is usually obtained by adding an offset X, which is given in the branch instruction, to the updated value of the PC. Figure 12 gives a control sequence that implements an unconditional branch instruction. Processing starts, as usual, with the fetch phase. This phase ends when the instruction is loaded into the IR in step 3.

The offset value is extracted from the IR by the instruction decoding circuit, which will also perform sign extension if required. Since the value of the updated PC is already available in register Y, the offset X is gated onto the bus in step 4, and an addition operation is performed. The result, which is the branch target address, is loaded into the PC in step 5.

Step	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Offset-field-of-IR_{out}, Add, Z_{in}
5	Z_{out}, PC_{in}, End

Figure 12

The offset X used in a branch instruction is usually the difference between the branch target address and the address immediately following the branch instruction.

For example: if the branch instruction is at location 2000 and if the branch target address is 2050, the value of X must be 46. The PC is incremented during the fetch phase before knowing the type of the instruction being executed. Thus, when the branch address is computed in step 4, the PC value uses the updated value, which points to the instruction following the branch instruction in the memory.

MULTIPLE BUS ORGANIZATION

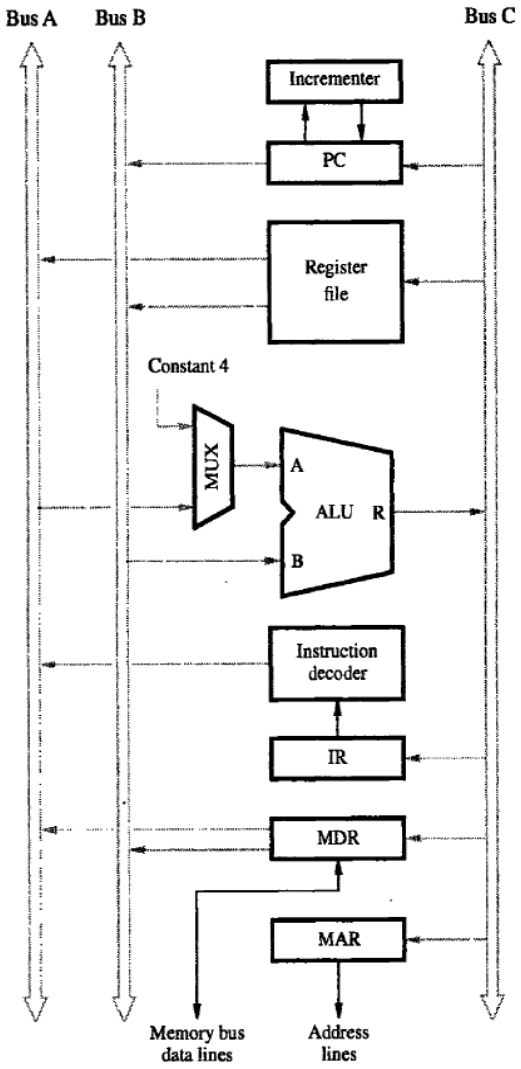
We used the simple single-bus structure to illustrate the basic ideas. The resulting control sequences are quite long because only one data item can be transferred over the bus in a clock cycle.

To reduce the number of steps needed, most commercial processors provide multiple internal paths that enable several transfers to take place in parallel. Figure depicts a three-bus structure used to connect the registers and the ALU of a processor.

The register file is said to have three ports. There are two outputs, allowing the contents of two different registers to be accessed simultaneously and have their contents placed on buses A and B. The third port allows the data on bus C to be loaded into a third register during the same clock cycle.

Buses A and B are used to transfer the source operands to the A and B inputs of the ALU, where an arithmetic or logic operation may be performed. The result is transferred to the destination over bus C. If needed, the ALU may simply pass one of its two input operands unmodified to bus C. We will call the ALU control signals for such an operation R=A or R=B. The three-bus arrangement obviates the need for registers Y and Z.

A second feature is the introduction of the Incrementer unit, which is used to increment the PC by 4. Using the Incrementer eliminates the need to add 4 to the PC using the main ALU. The source for the constant 4 at the ALU input multiplexer is still useful. It can be used to increment other addresses, such as the memory addresses in LoadMultiple and StoreMultiple instructions.



Consider the three-operand instruction

Add R4,R5,R6

The control sequence for executing this instruction is given as below

Step	Action
1	PC_{out}, R=B, MAR_{in}, Read, IncPC
2	WMFC
3	MDR_{outB}, R=B, IR_{in}
4	$R4_{outA}$, $R5_{outB}$, SelectA, Add, $R6_{in}$, End

Step 1: the contents of the PC are passed through the ALU, using the R=B control signal, and loaded into the MAR to start a memory read operation. At the same time the PC is incremented by 4. Note that the value loaded into MAR is the original contents of the PC. The incremented value is loaded into the PC at the end of the clock cycle and will not affect the contents of MAR.

Step 2: the processor waits for MFC and loads the data received into MDR.

Step 3: Transfers the data received in MDR to IR.

Step 4: The execution phase of the instruction requires only one control step to complete.

By providing more paths for data transfer a significant reduction in the number of clock cycles needed to execute an instruction is achieved.