

MCA103:

UNIT-I Flip-flops- Registers and registers- counters- decoders
Multiplexers- PLDs- sequential circuits. Structure of Computers
ITS: Basic computer structures-
Multiprocessors and Multi computers-Historical Perspective.

UNIT-II Addressing Methods and Machine Program Sequencing: Basic Concepts:-
Memory locations and address, Main Memory operations, Instructions and
Instruction Sequencing-Addressing Modes.

UNIT-III Input Output
Direct Memory Access- Hardware-

UNIT-IV Memory System Concepts:- Semiconductor RAM Memories-Read only
memories- Cache Memories- Performance Considerations-
Virtual Memories: Memory Management Requirements, Arithmetic:- Addition
and subtraction of sign members- Design of fast adders- Multiplication of
positive members- Signed operand multiplication- Fast multiplication-
Integer division- Floating point numbers and operations.

UNIT-V
Multiple- Hardware control-
Concepts: Data path - Instruction hazards-
Instruction sets-data path and control constructions.

Text book:

1) Hamacher C, Vranesic Z, and Zaky S. Computer Organization, 5th edition, McGraw-Hill, 2002.

2) Stallings W, Computer Organization and Architecture, 6th edition. Pearson Education, 20

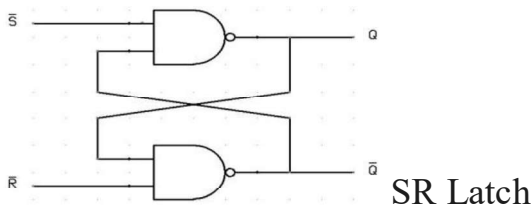
UNIT-I

➤ Flip-Flops

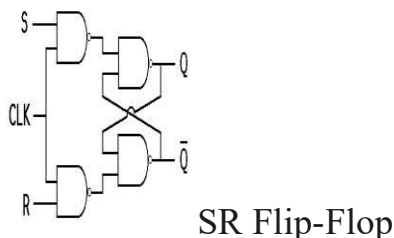
Flip flop is an electronic circuit with two stable states that can be used to store binary data. The stored data can be changed by applying varying inputs. Flip-flops and latches are fundamental building blocks of digital electronics systems used in computers, communications, and many other types of systems. Flip-flops and latches are used as data storage elements. It is the basic storage element in sequential logic. But first, let's clarify the difference between a latch and a flip-flop.

Flip flop v/s Latch

The basic difference between a latch and a flip-flop is a gating or clocking mechanism. For example, let us talk about SR latch and SR flip-flops. In this circuit when you Set S as active the output Q would be high and Q' will be low. This is irrespective of anything else. (This is an active-low circuit so active here means low, but for an active high circuit active would mean high)



A flip flop, on the other hand, is synchronous and is also known as gated or clocked SR latch.

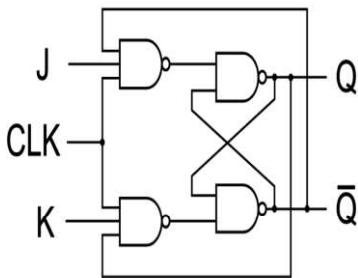


In this circuit diagram, the output is changed (i.e. the stored data is changed) only when you give an active clock signal. Otherwise, even if the S or R is active the data will not change. Let's look at the types of flip-flops to understand better. **SR Flip Flop** There are majorly 4 types of flip flops, with the most common one being SR flip flop. This simple flip flop circuit has a set input (S) and a reset input (R). In this circuit when you Set "S" as active the output "Q" would be high and "Q'" will be low. Once the outputs are established, the wiring of the circuit is maintained until "S" or "R" go high, or power is turned off. As shown above, it is the simplest and easiest to

understand. The two outputs, as shown above, are the inverse of each other. The truth table of SR Flip Flop is highlighted below.

JK Flip-flop

Due to the undefined state in the SR flip flop, another flip flop is required in electronics. The JK flip flop is an improvement on the SR flip flop where S=R=1 is not a problem.



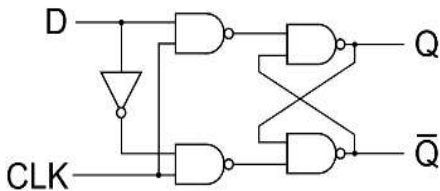
J	K	Q	Q'
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	1
0	0	1	1
0	1	1	0
1	0	1	1
1	1	1	0

JK Flip-Flop

The input condition of J=K=1, gives an output state. However, the outputs are the same when practically.

In simple words, If J and K data input are and low) then the output Q takes the value of J edge. If J and K are both low then no change occurs. If J and K are both high at the clock edge then the output will toggle from one state to the other. JK Flip Flop can function as Set or Reset Flip flop

D Flip Flop



Clock	D	Q	Q'
↓ » 0	0	0	1
↑ » 1	0	0	1
↓ » 0	1	0	1
↑ » 1	1	1	0

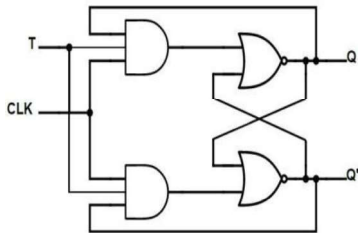
D flip flop is a better alternative that is very popular with digital electronics. They are commonly used for counters and shift-registers and input synchronisation.

In a D flip flop, the output can be only changed at the clock edge, and if the input changes at other times, the output will be unaffected. The change of state of the output is dependent on the rising edge of the clock. The output (Q) is same as the input and can only change at the rising edge of the clock.

T Flip Flop

A T flip flop is like JK flip-flop. These are basically a single input version of JK flip flop. This modified form of JK flip-flop is obtained by connecting both inputs J and K together. This flip-flop has only one input along with the clock input.

T	Q	Q (t+1)
0	0	0
1	0	1
0	1	1
1	1	0



These flip-flops are called T flip-flops because of their ability to complement its state (i.e.) Toggle, hence the name Toggle flip-flop.

Applications of Flip-Flops

These are the various types of flip-flops being used in digital electronic circuits and the applications of Flip-flops are as specified below.

- Counters, Frequency Dividers, Shift Registers, Storage Registers

Registers

A **Register** is a collection of flip flops. A flip flop is used to store single bit digital data. For storing a large number of bits, the storage capacity is increased by grouping more than one flip flops. If we want to store an n-bit word, we have to use an n-bit register containing n number of flip flops.

The register is used to perform different types of operations. For performing the operations, the CPU use these registers. The faded inputs to the system will store into the registers. The result returned by the system will store in the registers. There are the following operations which are performed by the registers:

Fetch:It is used

- To take the instructions given by the users.
- To fetch the instruction stored into the main memory.

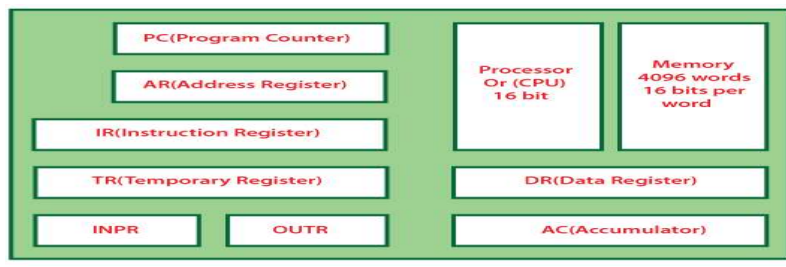
Decode:

The decode operation is used to interpret the instructions. In decode, the operation performed on the instructions is identified by the **CPU**. In simple words, the decode operation is used to decode the instructions.

Execute:

The execution operation is used to store the result produced by the **CPU** into the memory. After storing this result, it is displayed on the user screen.

Types of Register There are various types of registers which are as follows:



MAR or Memory Address Register

The MAR is a special type of register that contains the memory address of the data and instruction. The main task of the MAR is to access instruction and data from memory in the execution phase. The MAR stores the address of the memory location where the data is to be read or to be stored by the CPU.

Program Counter

The program counter is also called an instruction address register or instruction pointer. The next memory address of the instruction, which is going to be executed after completing the

execution of current instruction is contained in the program counter. In simple words, the program counter contains the memory address of the location of the next instruction.

Accumulator Register

The CPU mostly uses an accumulator register. The accumulator register is used to store the system result. All the results will be stored in the accumulator register when the CPU produces some results after processing.

MDR or Memory Data Register

Memory Data Register is a part of the computer's control unit. It contains the data that we want to store in the computer storage or the data fetched from the computer storage. The MDR works as a buffer that contains anything for which the processor is ready to use it. The MDR contains the copied data of the memory for the processor. Firstly the MDR holds the information, and then it goes to the decoder. The data which is to be read out or written into the address location is contained in the **Memory Data Register**.

The data is written in one direction when it is fetched from memory and placed into the MDR. In write instruction, the data place into the MDR from another CPU register. This CPU register writes the data into the memory. Half of the minimal interface between the computer storage and the microprogram is the memory data address register, and the other half is the memory data register.

Index Register

The **Index Register** is the hardware element that holds the number. The number adds to the computer instruction's address to create an effective address. In CPU, the index register is a processor register used to modify the operand address during the running program.

Memory Buffer Register

Memory Buffer Register is mostly called MBR. The MBR contains the Metadata of the data and instruction written in or read from memory. In simple words, it adds is used to store the upcoming data/instruction from the memory and going to memory.

Data Register

The data register is used to temporarily store the data. This data transmits to or from a peripheral device.

➤ Shift registers:

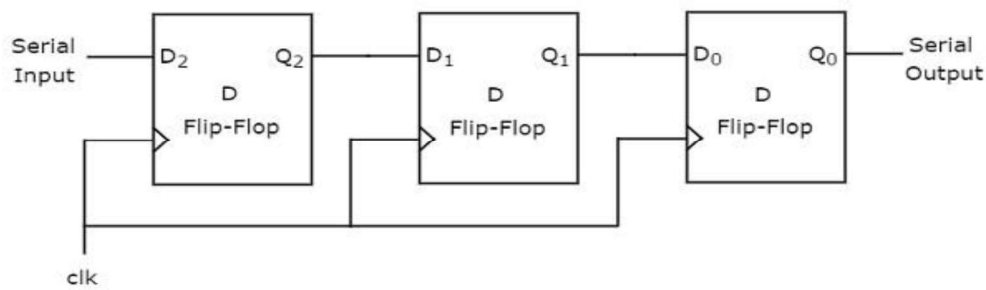
We know that one flip-flop can store one-bit of information. In order to store multiple bits of information, we require multiple flip-flops. The group of flip-flops, which are used to hold storestore the binary data is known as **register**.

If the register is capable of shifting bits either towards right hand side or towards left hand side is known as **shift register**. An 'N' bit shift register contains 'N' flip-flops. Following are the four types of shift registers based on applying inputs and accessing of outputs.

- Serial In – Serial Out shift register
- Serial In – Parallel Out shift register
- Parallel In – Serial Out shift register
- Parallel In – Parallel Out shift register

Serial In – Serial Out SISOSISO Shift Register

The shift register, which allows serial input and produces serial output is known as Serial In – Serial Out SISOSISO shift register. The **block diagram** of 3-bit SISO shift register is shown in the following figure.

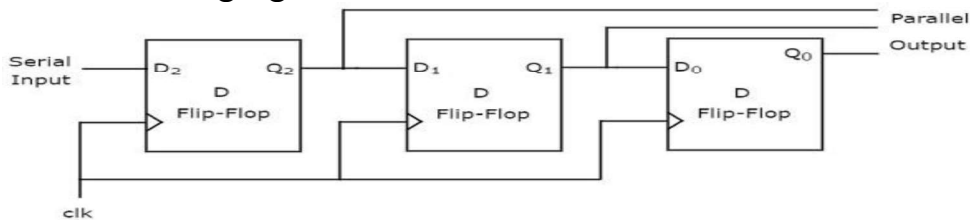


This block diagram consists of three D flip-flops, which are **cascaded**. That means, output of one D flip-flop is connected as the input of next D flip-flop. All these flip-flops are synchronous with each other since, the same clock signal is applied to each one.

In this shift register, we can send the bits serially from the input of left most D flip-flop. Hence, this input is also called as **serial input**. For every positive edge triggering of clock signal, the data shifts from one stage to the next. So, we can receive the bits serially from the output of right most D flip-flop. Hence, this output is also called as **serial output**.

Serial In - Parallel Out SIPO SIPO Shift Register

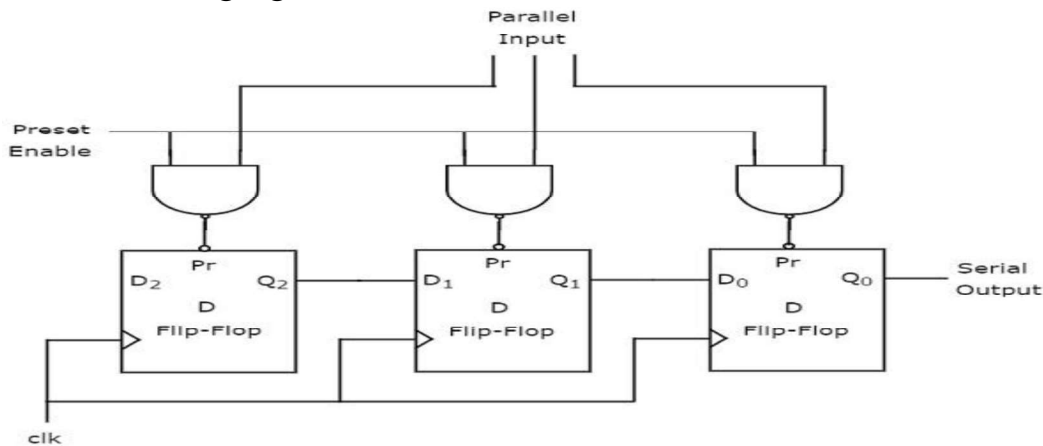
The shift register, which allows serial input and produces parallel output is known as Serial In – Parallel Out SIPOSIPO shift register. The **block diagram** of 3-bit SIPO shift register is shown in the following figure.



This circuit consists of three D flip-flops, which are cascaded. That means, output of one D flip-flop is connected as the input of next D flip-flop. All these flip-flops are synchronous with each other since, the same clock signal is applied to each one.

Parallel In – Serial Out PISO PISO Shift Register

The shift register, which allows parallel input and produces serial output is known as Parallel In – Serial Out PISO PISO shift register. The **block diagram** of 3-bit PISO shift register is shown in the following figure.

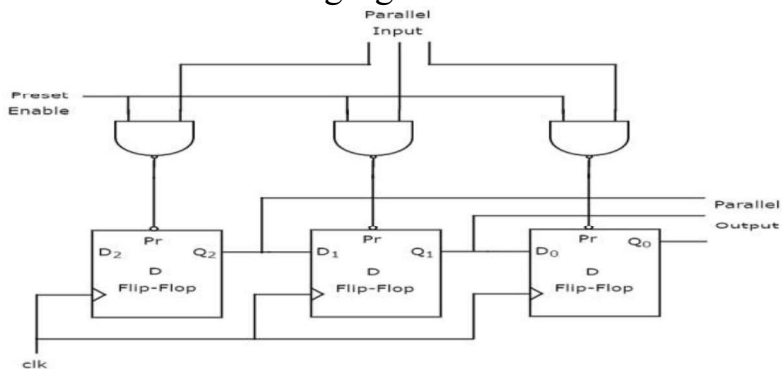


This circuit consists of three D flip-flops, which are cascaded. That means, output of one D flip-flop is connected as the input of next D flip-flop. All these flip-flops are synchronous with each other since, the same clock signal is applied to each one.

In this shift register, we can apply the **parallel inputs** to each D flip-flop by making Preset Enable to 1. For every positive edge triggering of clock signal, the data shifts from one stage to the next. So, we will get the **serial output** from the right most D flip-flop.

Parallel In - Parallel Out PIPOPIPO Shift Register

The shift register, which allows parallel input and produces parallel output is known as Parallel In – Parallel Out PIPOPIPO shift register. The **block diagram** of 3-bit PIPO shift register is shown in the following figure.



This circuit consists of three D flip-flops, which are cascaded. That means, output of one D flip-flop is connected as the input of next D flip-flop. All these flip-flops are synchronous with each other since, the same clock signal is applied to each one.

In this shift register, we can apply the **parallel inputs** to each D flip-flop by making Preset Enable to 1. We can apply the parallel inputs through preset or clear. These two are asynchronous inputs. That means, the flip-flops produce the corresponding outputs, based on the values of asynchronous inputs. In this case, the effect of outputs is independent of clock transition. So, we will get the **parallel outputs** from each D flip-flop.

➤ Counters

A special type of sequential circuit used to count the pulse is known as a counter, or a collection of flip flops where the clock signal is applied is known as counters.

The counter is one of the widest applications of the flip flop. Based on the clock pulse, the output of the counter contains a predefined state. The number of the pulse can be counted using the output of the counter.

Truth tables:

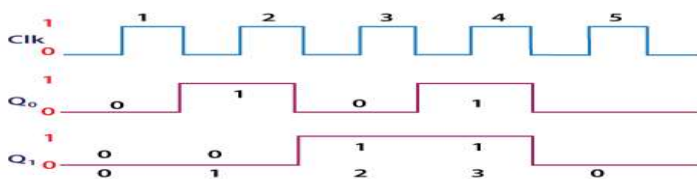
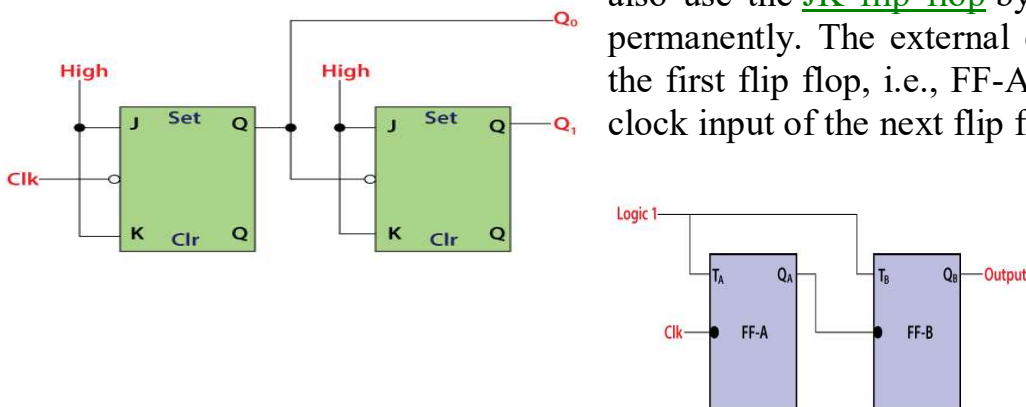
Clock	Counter output		State number	Decimal counter output
	Q _B	Q _A		
Initially	0	0	-	0
1 st	0	1	1	1
2 nd	1	0	2	2
3 rd	1	1	3	3
4 th	0	0	4	0

There are the following types of counters:

- Asynchronous Counters
- Synchronous Counters

Asynchronous or ripple counters

The **Asynchronous counter** is also known as the **ripple counter**. Below is a diagram of the 2-bit **Asynchronous counter** in which we used two T flip-flops. Apart from the T flip flop, we can also use the [JK flip flop](#) by setting both of the inputs to 1 permanently. The external clock pass to the clock input of the first flip flop, i.e., FF-A and its output, i.e., is passed to clock input of the next flip flop, i.e., FF-B. Block diagram:



signal Diagram

Operation

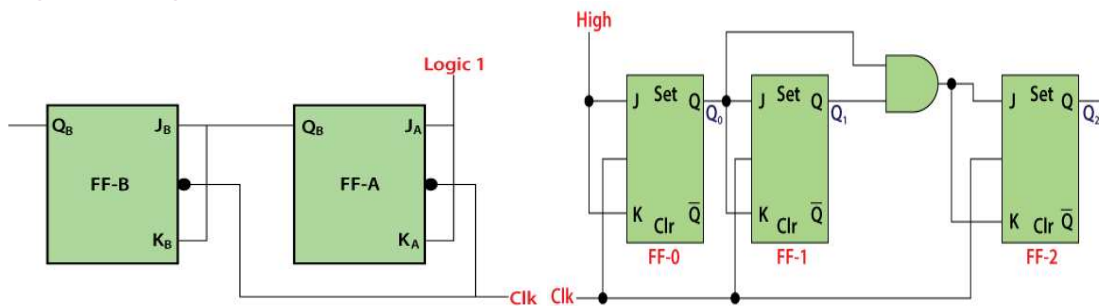
- Condition 1:** When both the flip flops are in reset condition.
Operation: The outputs of both flip flops, i.e., Q_A Q_B , will be 0.
- Condition 2:** When the first negative clock edge passes.
- Condition 3:** When the second negative clock edge is applied.
- Condition 4:** When the third negative clock edge is applied.
- Condition 5:** When the fourth negative clock edge is applied.

Synchronous counters

In the **Asynchronous counter**, the present counter's output passes to the input of the next counter. So, the counters are connected like a chain. The drawback of this system is that it creates the counting delay, and the propagation delay also occurs during the counting stage. The **synchronous counter** is designed to remove this drawback.

In the **synchronous counter**, the same clock pulse is passed to the clock input of all the flip flops. The clock signals produced by all the flip flops are the same as each other. Below is the diagram of a 2-bit synchronous counter in which the inputs of the first flip flop, i.e., FF-A, are set to 1. So, the first flip flop will work as a toggle flip-flop. The output of the first flip flop is passed to both the inputs of the next JK flip flop.

Logical Diagram



Ripple Counter

Ripple counter is a special type of **Asynchronous counter** in which the clock pulse ripples through the circuit. The n-MOD ripple counter forms by combining n number of flip-flops. The n-MOD ripple counter can count 2^n states, and then the counter resets to its initial value.

Features of the Ripple Counter:

- Different types of flip flops with different clock pulse are used.
- It is an example of an asynchronous counter.
- The flip flops are used in toggle mode.

- The external clock pulse is applied to only one flip flop. The output of this flip flop is treated as a clock pulse for the next flip flop.
- In counting sequence, the flip flop in which external clock pulse is passed, act as LSB.

Based on their circuitry design, the counters are classified into the following types:

Up Counter The up-counter counts the states in ascending order.

Down Counter The down counter counts the states in descending order.

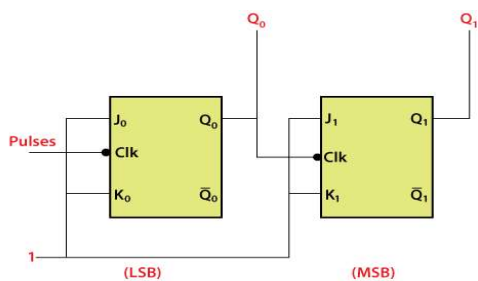
Up-Down Counter

The up and down counter is a special type of bi-directional counter which counts the states either in the forward direction or reverse direction. It also refers to a reversible counter.

Binary Ripple Counter

A **Binary counter** is a **2-Mod counter** which counts up to 2-bit state values, i.e., $2^2 = 4$ values. The flip flops having similar conditions for toggling like T and JK are used to construct the **Ripple counter**. Below is a circuit diagram of a **binary ripple counter**.

In the circuit design of the binary ripple counter, two JK flip flops are used. The high voltage signal is passed to the inputs of both flip flops. This high voltage input maintains the flip flops at a state 1. In JK flip flops, the negative triggered clock pulse use

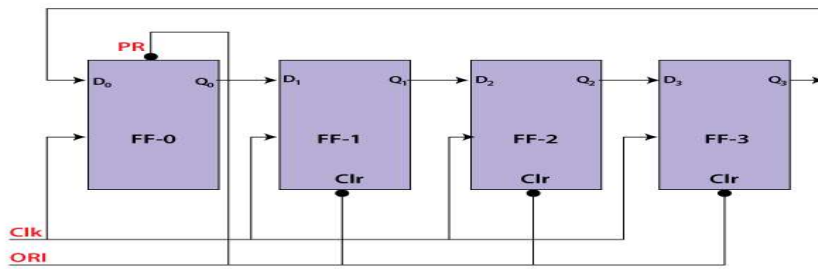


Ring Counter

A **ring counter** is a special type of application of the **Serial IN Serial OUT** Shift register. The only difference between the shift register and the ring counter is that the last flip flop outcome is taken as the output in the shift register. But in the ring counter, this outcome is passed to the first flip flop as an input. All of the remaining things in the ring counter are the same as the shift register.

In the **Ring counter** No. of states in Ring counter = No. of flip-flop used. Below is the block diagram of the 4-bit ring counter. Here, we use 4 D flip flops. The same clock pulse is passed to the clock input of all the flip flops as a synchronous counter. The **Overriding input(ORI)** is used to design this circuit.

The Overriding input is used as **clear** and **pre-set**



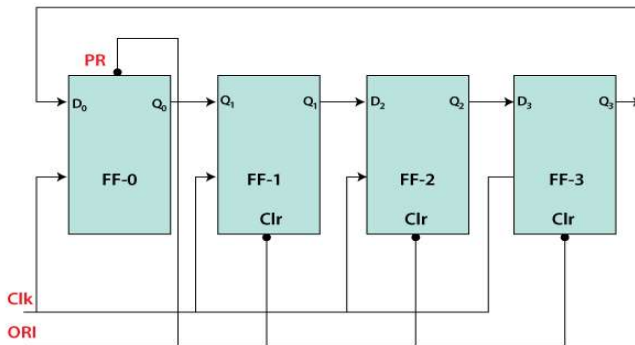
Types of Ring Counter

The ring counter is classified into two parts which are as follows:

Straight Ring Counter

The **Straight Ring Counter** refers to as **One hot Counter**. The outcome of the last flip-flop is passed to the first flip-flop as an input. In the ring counter, the ORI input is passed to the PR input for the first flip flop and to the clear input of the remaining flip flops.

Note: The straight ring counter circulates the single 1 (or 0) bit around the ring.

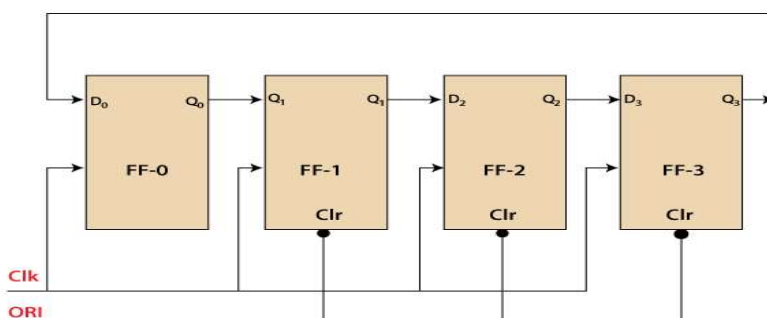


Twisted Ring Counter

The **Twisted Ring Counter** refers to as a **switch-tail ring Counter**. Like the **straight ring counter**, the outcome of the last flip-flop is passed to the first flip-flop as an input. In the twisted ring counter, the ORI input is passed to all the flip flops as **clear** input.

Note: The twisted ring counter circulates a stream of 1's followed by 0 around the ring.

Logic Diagram



The sequential circuit is a special type of circuit that has a series of inputs and outputs. The outputs of the sequential circuits depend on both the combination of present inputs and previous outputs. The previous output is treated as the present state. So, the sequential circuit contains the combinational circuit and its memory storage elements. A sequential circuit doesn't need to always contain a combinational circuit. So, the sequential circuit can contain only the memory element.

Difference between the combinational circuits and sequential circuits are given below:

	Combinational Circuits	Sequential Circuits
1)	The outputs of the combinational circuit depend only on the present inputs.	The outputs of the sequential circuits depend on both present inputs and present state(previous output).
2)	The feedback path is not present in the combinational circuit.	The feedback path is present in the sequential circuits.
3)	In combinational circuits, memory elements are not required.	In the sequential circuit, memory elements play an important role and require.
4)	The clock signal is not required for combinational circuits.	The clock signal is required for sequential circuits.
5)	The combinational circuit is simple to design.	It is not simple to design a sequential circuit.

Types of Sequential Circuits

Asynchronous sequential circuits

The clock signals are not used by the **Asynchronous sequential circuits**. The asynchronous circuit is operated through the pulses. So, the changes in the input can change the state of the circuit. The asynchronous circuits do not use clock pulses. The internal state is changed when the input variable is changed. The un-clocked flip-flops or time-delayed are the memory elements of asynchronous sequential circuits. The asynchronous sequential circuit is similar to the combinational circuits with feedback.

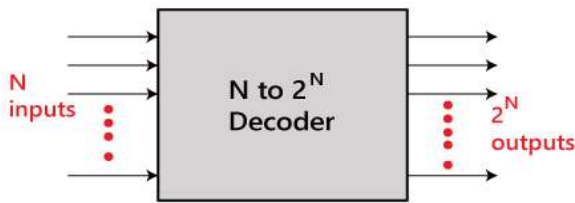
Synchronous sequential circuits

In synchronous sequential circuits, synchronization of the memory element's state is done by the clock signal. The output is stored in either flip-flops or latches(memory devices). The synchronization of the outputs is done with either only negative edges of the clock signal or only positive edges

➤ Decoders

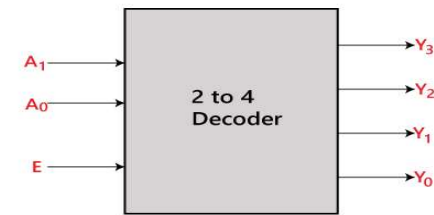
The combinational circuit that change the binary information into 2^N output lines is known as **Decoders**. The binary information is passed in the form of N input lines. The output lines

define the 2^N -bit code for the binary information. In simple words, the **Decoder** performs the reverse operation of the **Encoder**. At a time, only one input line is activated for simplicity. The produced 2^N -bit output code is equivalent to the binary information.



2 to 4 line decoder:

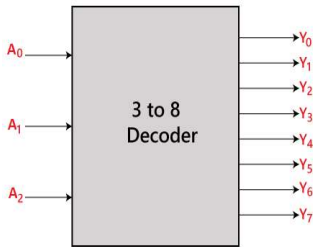
In the 2 to 4 line decoder, there is a total of three inputs, i.e., A_0 , and A_1 and E and four outputs, i.e., Y_0 , Y_1 , Y_2 , and Y_3 . For each combination of inputs, when the enable 'E' is set to 1, one of these four outputs will be 1. The block diagram and the truth table of the



2 to 4 line decoder are given below.

Block Diagram:

Enable	INPUTS		OUTPUTS			
E	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	X	X	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0



3 to 8 line decoder:

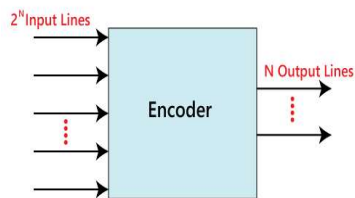
The 3 to 8 line decoder is also known as **Binary to Octal Decoder**. In a 3 to 8 line decoder, there is a total of eight outputs, i.e., Y_0 , Y_1 , Y_2 , Y_3 , Y_4 , Y_5 , Y_6 , and Y_7 and three outputs, i.e., A_0 , A_1 , and A_2 . This circuit has an enable input 'E'. Just like 2 to 4 line decoder, when enable 'E' is set to 1, one of these four outputs will be 1. The block diagram and the truth table of the 3 to 8 line encoder are given below.

Block Diagram

Encoders

The combinational circuits that change the binary information into N output lines are known as **Encoders**. The binary information is passed in the form of 2^N input lines. The output lines define the N-bit code for the binary information. In simple words, the **Encoder** performs the

reverse operation of the **Decoder**. At a time, only one input line is activated for simplicity. The produced N-bit output code is equivalent to the binary information

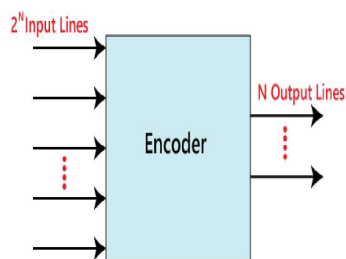


4 to 2 line Encoder:

In 4 to 2 line encoder, there are total of four inputs, i.e., Y_0 , Y_1 , Y_2 , and Y_3 , and two outputs, i.e., A_0 and A_1 . In 4-input lines, one input-line is set to true at a time to get the respective binary code in the output side. Below are the block diagram and the truth table of the 4 to 2 line encoder.

Block Diagram:

TRUTH TABLE:



INPUTS				OUTPUTS	
Y_3	Y_2	Y_1	Y_0	A_1	A_0
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1

➤ Multiplexers

A multiplexer is a combinational circuit that has 2^n input lines and a single output line. Simply, the multiplexer is a multi-input and single-output combinational circuit. The binary information is received from the input lines and directed to the output line. On the basis of the values of the selection lines, one of these data inputs will be connected to the output.

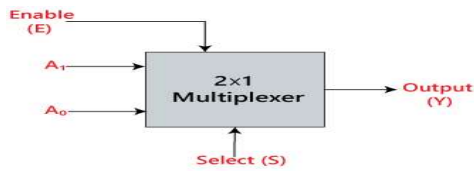
Unlike encoder and decoder, there are n selection lines and 2^n input lines. So, there is a total of 2^N possible combinations of inputs. A multiplexer is also treated as **Mux**.

There are various types of the multiplexer which are as follows:

2×1 Multiplexer:

In 2×1 multiplexer, there are only two inputs, i.e., A_0 and A_1 , 1 selection line, i.e., S_0 and single outputs, i.e., Y. On the basis of the combination of inputs which are present at the selection line S_0 , one of these 2 inputs will be connected to the output. The block diagram and the truth table of the 2×1 multiplexer are given below

Block Diagram

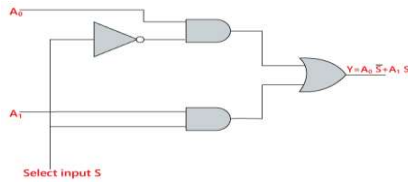


Truth table

INPUTS		Output
S_0		Y
0		A_0
1		A_1

$$Y = S_0' \cdot A_0 + S_0 \cdot A_1$$

Logical circuit of the above expression is given below:

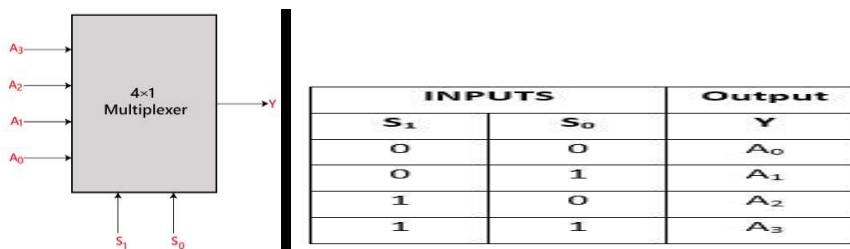


4×1 Multiplexer:

In the 4×1 multiplexer, there is a total of four inputs, i.e., A_0 , A_1 , A_2 , and A_3 , 2 selection lines, i.e., S_0 and S_1 and single output, i.e., Y. On the basis of the combination of inputs that are present at the selection lines S_0 and S_1 , one of these 4 inputs are connected to the output. The block diagram and the truth table of the 4×1 multiplexer are given below

Block Diagram:

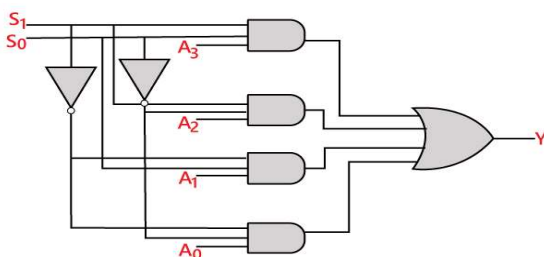
Truth table



The logical expression of the term Y is as follows:

$$Y = S_1' S_0' A_0 + S_1' S_0 A_1 + S_1 S_0' A_2 + S_1 S_0 A_3$$

Logical circuit of the above expression is given below

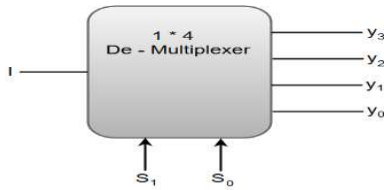


De-Multiplexers

A De-multiplexer (De-Mux) can be described as a combinational circuit that performs the reverse operation of a Multiplexer.

A De-multiplexer has a single input, 'n' selection lines and a maximum of 2^n outputs.

The following image shows the block diagram of a 1×4 De-multiplexer.



➤ Programmable Logic Devices PLDs

PLDs are the integrated circuits. They contain an array of AND gates & another array of OR gates. There are three kinds of PLDs based on the type of arrays, which has programmable feature.

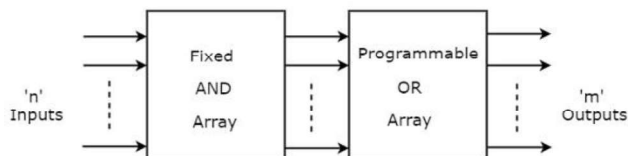
- Programmable Read Only Memory
- Programmable Array Logic
- Programmable Logic Array

The process of entering the information into these devices is known as **programming**. Basically, users can program these devices or ICs electrically in order to implement the Boolean functions based on the requirement. Here, the term programming refers to hardware programming but not software programming.

Programmable Read Only Memory PROM

Read Only Memory ROM is a memory device, which stores the binary information permanently. That means, we can't change that stored information by any means later. If the ROM has programmable feature, then it is called as **Programmable ROM** PROM. The user has the flexibility to program the binary information electrically once by using PROM programmer.

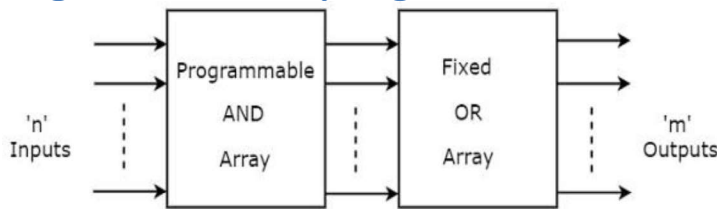
PROM is a programmable logic device that has fixed AND array & Programmable OR array. The **block diagram** of PROM is shown in the following figure.



Here, the inputs of AND gates are not of programmable type. So, we have to generate 2^n product terms by using 2^n AND gates having n inputs each. We can implement these product terms by using $n \times 2^n$ decoder. So, this decoder generates 'n' **min terms**.

Here, the inputs of OR gates are programmable. That means, we can program any number of required product terms, since all the outputs of AND gates are applied as inputs to each OR gate. Therefore, the outputs of PROM will be in the form of **sum of min terms**.

Programmable Array Logic PALPAL



PAL is a programmable logic device that has Programmable AND array & fixed OR array. The advantage of PAL is that we can generate only the required product terms of Boolean function instead of generating all the min

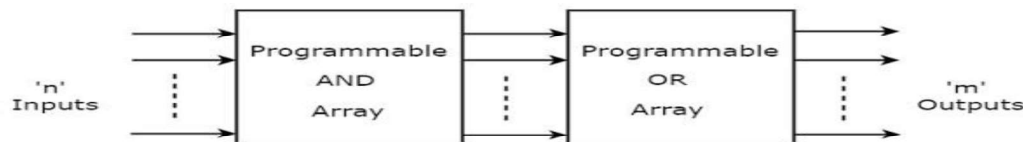
terms by using programmable AND gates. The **block diagram** of PAL is shown in the following figure.

Here, the inputs of AND gates are programmable. That means each AND gate has both normal and complemented inputs of variables. So, based on the requirement, we can program any of those inputs. So, we can generate only the required **product terms** by using these AND gates.

Here, the inputs of OR gates are not of programmable type. So, the number of inputs to each OR gate will be of fixed type. Hence, apply those required product terms to each OR gate as inputs. Therefore, the outputs of PAL will be in the form of **sum of products form**.

Programmable Logic Array PLAPLA

PLA is a programmable logic device that has both Programmable AND array & Programmable OR array. Hence, it is the most flexible PLD. The **block diagram** of PLA is shown in the following figure.



Here, the inputs of AND gates are programmable. That means each AND gate has both normal and complemented inputs of variables. So, based on the requirement, we can program any of those inputs. So, we can generate only the required **product terms** by using these AND gates.

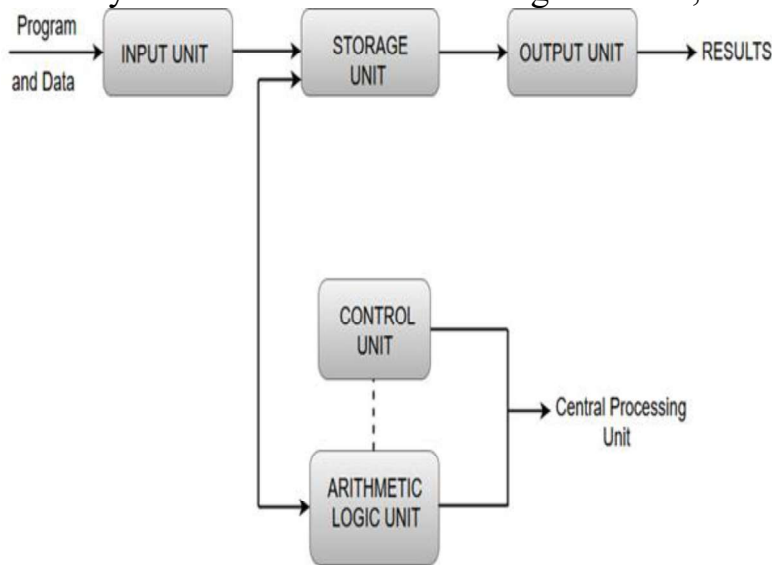
Here, the inputs of OR gates are also programmable. So, we can program any number of required product terms, since all the outputs of AND gates are applied as inputs to each OR gate. Therefore, the outputs of PAL will be in the form of **sum of products form**.

➤ BASIC STRUCTURE OF COMPUTERS

➤ Functional Units

- A computer organization describes the functions and design of the various units of a digital system.

- A general-purpose computer system is the best-known example of a digital system. Other examples include telephone switching exchanges, digital voltmeters, digital counters, electronic calculators and digital displays.
- Computer architecture deals with the specification of the instruction set and the hardware units that implement the instructions.
- Computer hardware consists of electronic circuits, displays, magnetic and optic storage media and also the communication facilities.
- Functional units are a part of a CPU that performs the operations and calculations called for by the computer program.
- Functional units of a computer system are parts of the CPU (Central Processing Unit) that performs the operations and calculations called for by the computer program. A computer consists of five main components namely, Input unit, Central Processing Unit, Memory unit Arithmetic & logical unit, Control unit and an Output unit.



Input unit

- Input units are used by the computer to read the data. The most commonly used input devices are keyboards, mouse, joysticks, trackballs, microphones, etc.
- However, the most well-known input device is a keyboard. Whenever a key is pressed, the corresponding letter or digit is automatically translated into its corresponding binary code and transmitted over a cable to either the memory or the processor.

Central processing unit

- Central processing unit commonly known as CPU can be referred as an electronic circuitry within a computer that carries out the instructions given by a computer program by performing the basic arithmetic, logical, control and input/output (I/O) operations specified by the instructions.

Memory unit

- The Memory unit can be referred to as the storage area in which programs are kept which are running, and that contains data needed by the running programs.

- The Memory unit can be categorized in two ways namely, primary memory and secondary memory.
- It enables a processor to access running execution applications and services that are temporarily stored in a specific memory location.
- Primary storage is the fastest memory that operates at electronic speeds. Primary memory contains a large number of semiconductor storage cells, capable of storing a bit of information. The word length of a computer is between 16-64 bits.
- It is also known as the volatile form of memory, means when the computer is shut down, anything contained in RAM is lost.
- Cache memory is also a kind of memory which is used to fetch the data very soon. They are highly coupled with the processor.
- The most common examples of primary memory are RAM and ROM.
- Secondary memory is used when a large amount of data and programs have to be stored for a long-term basis.
- It is also known as the Non-volatile memory form of memory, means the data is stored permanently irrespective of shut down.
- The most common examples of secondary memory are magnetic disks, magnetic tapes, and optical disks.

Arithmetic & logical unit

- Most of all the arithmetic and logical operations of a computer are executed in the ALU (Arithmetic and Logical Unit) of the processor. It performs arithmetic operations like addition, subtraction, multiplication, division and also the logical operations like AND, OR, NOT operations.

Control unit

- The control unit is a component of a computer's central processing unit that coordinates the operation of the processor. It tells the computer's memory, arithmetic/logic unit and input and output devices how to respond to a program's instructions.
- The control unit is also known as the nerve center of a computer system.
- Let's us consider an example of addition of two operands by the instruction given as Add LOCA, RO. This instruction adds the memory location LOCA to the operand in the register RO and places the sum in the register RO. This instruction internally performs several steps.

Output Unit

- The primary function of the output unit is to send the processed results to the user. Output devices display information in a way that the user can understand.
 - Output devices are pieces of equipment that are used to generate information or any other response processed by the computer. These devices display information that has been held or generated within a computer.
 - The most common example of an output device is a monitor.
-

➤ Basic Operational Concepts

- The primary function of a computer system is to execute a program, sequence of instructions. These instructions are stored in computer memory.
- These instructions are executed to process data which are already loaded in the computer memory through some input devices.
- After processing the data, the result is either stored in the memory for further reference, or it is sent to the outside world through some output port.
- To perform the execution of an instruction, in addition to the arithmetic logic unit, and control unit, the processor contains a number of registers used for temporary storage of data and some special function registers.
- The special function registers include program counters (PC), instruction registers (IR), memory address registers (MAR) and memory and memory data registers (MDR).
- The Program counter is one of the most critical registers in CPU.
- The Program counter monitors the execution of instructions. It keeps track on which instruction is being executed and what the next instruction will be.
- The instruction register IR is used to hold the instruction that is currently being executed.
- The contents of IR are available to the control unit, which generate the timing signals that control, the various processing elements involved in executing the instruction.
- The two registers MAR and MDR are used to handle the data transfer between the main memory and the processor.
- The MAR holds the address of the main memory to or from which data is to be transferred.
- The MDR contains the data to be written into or read from the addressed word of the main memory.
- Whenever the processor is asked to communicate with devices, we say that the processor is servicing the devices. The processor can service these devices in one of the two ways.
- One way is to use the polling routine, and the other way is to use an interrupt.
- Polling enables the processor software to check each of the input and output devices frequently. During this check, the processor tests to see if any devices need servicing or not.
- Interrupt method provides an external asynchronous input that informs the processor that it should complete whatever instruction that is currently being executed and fetch a new routine that will service the requesting device

➤ Bus structure

Single bus structure: In computer architecture, a bus is a subsystem that transfers data between components inside a computer, or between computers. Early computer buses were literally parallel electrical wires with multiple connections, but Modern computer buses can use both parallel and bit serial connections.

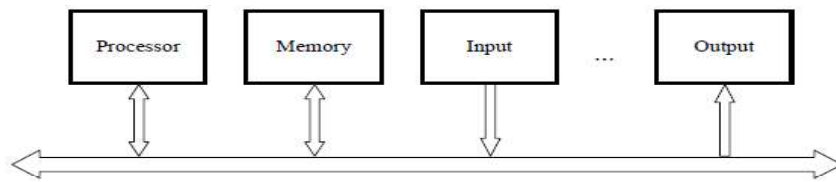


Figure 1.3.1 Single bus structure

To achieve a reasonable speed of operation, a computer must be organized so that all its units can handle one full word of data at a given time. When a word of data is transferred between units, all its bits are transferred in parallel, that is, the bits are transferred simultaneously over many wires, or lines, one bit per line. A group of lines that serves as a connecting path for several devices is called a bus. In addition to the lines that carry the data, the bus must have lines for address and control purposes. The simplest way to interconnect functional units is to use a single bus, as shown in Figure 1.3.1. All units are connected to this bus.

Parts of a System bus: Processor, memory, Input and output devices are connected by system bus, which consists of separate busses as shown in figure 1.3.2. They are:

- (i)Address bus:** Address bus is used to carry the address. It is unidirectional bus. The address is sent to from CPU to memory and I/O port and hence unidirectional. It consists of 16, 20, 24 or more parallel signal lines.
- (ii)Data bus:** Data bus is used to carry or transfer data to and from memory and I/O ports. They are bidirectional. The processor can read on data lines from memory and I/O port and as well as it can write data to memory. It consists of 8, 16, 32 or more parallel signal lines.
- (iii)Control bus:** Control bus is used to carry control signals in order to regulate the control activities. They are bidirectional. The CPU sends control signals on the control bus to enable the outputs of addressed memory devices or port devices. Some of the control signals are: MEMR (memory read), MEMW (memory write), IOR (I/O read), IOW (I/O write), BR (bus request), BG (bus grant), INTR (interrupt request), INTA (interrupt acknowledge), RST (reset), RDY (ready), HLD (hold), HLDA (hold acknowledge),

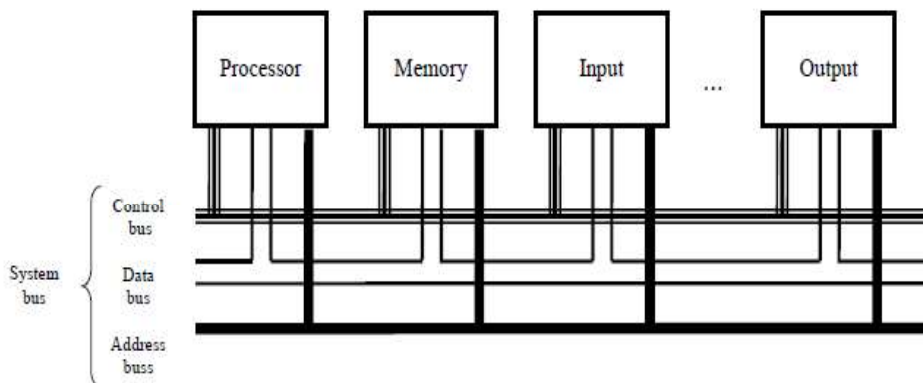


Fig:Businterconnectionscheme

The devices connected to a bus vary widely in their speed of operation. Some electromechanical devices, such as keyboards and printers are relatively slow. Other devices like magnetic or optical disks, are considerably faster. Memory and processor units operate at electronic speeds, making them the fastest parts of a computer. Because all these devices must communicate with each other over a bus, an efficient transfer mechanism that is not constrained by the slow devices

and that can be used to smooth out the differences in timing among processors, memories, and external devices is necessary.

A common approach is to include buffer registers with the devices to hold the information during transfers. To illustrate this technique, consider the transfer of an encoded character from a processor to a character printer. The processor sends the character over the bus to the printer buffer. Since the buffer is an electronic register, this transfer requires relatively little time. Once the buffer is loaded, the printer can start printing without further intervention by the processor. The printer continues printing the character in its buffer and is not available for further transfers until this process is completed. Thus, buffer registers smooth out timing differences among processors, memories, and I/O devices. They prevent a high-speed processor from being locked to a slow I/O device during a sequence of data transfers. This allows the processor to switch rapidly from one device to another, interweaving its processing activity with data transfers involving several I/O devices.

The Figure 1.3.3 shows traditional bus configurations and the Figure 1.3.4 shows high speed bus configurations. The traditional bus connection uses three buses: local bus, system bus and expanded bus. The high speed bus configuration uses high-speed bus along with the three buses used in the traditional bus connection. Here, cache controller is connected to highspeed bus. This bus supports connection to high-speed LANs, such as Fiber Distributed Data Interface (FDDI), video and graphics workstation controllers, as well as interface controllers to local peripheral including SCSI.

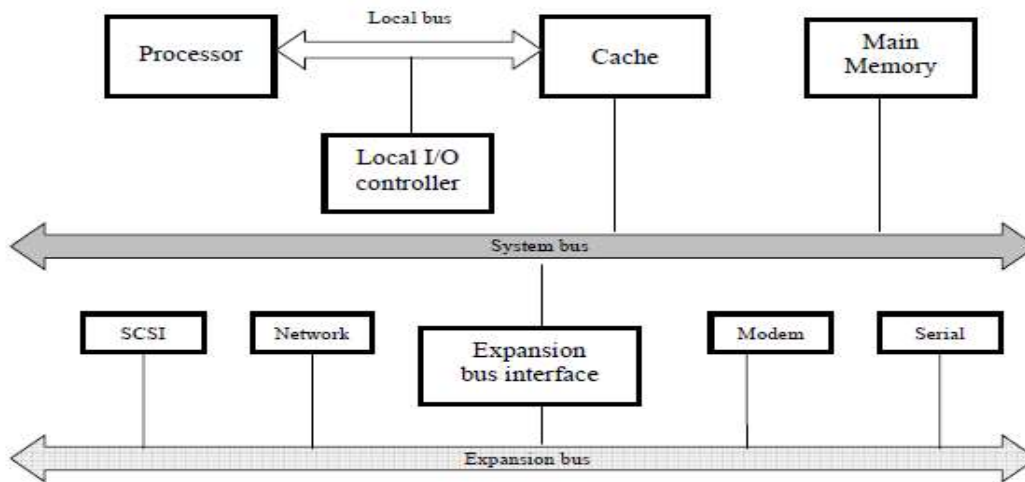


Figure 1.3.3 Traditional bus configuration

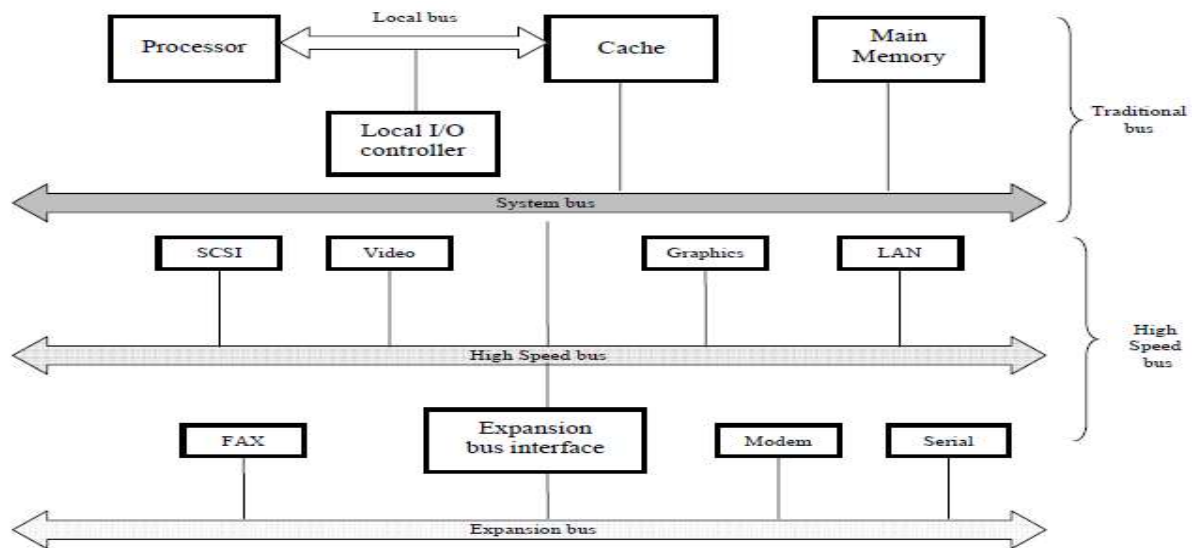
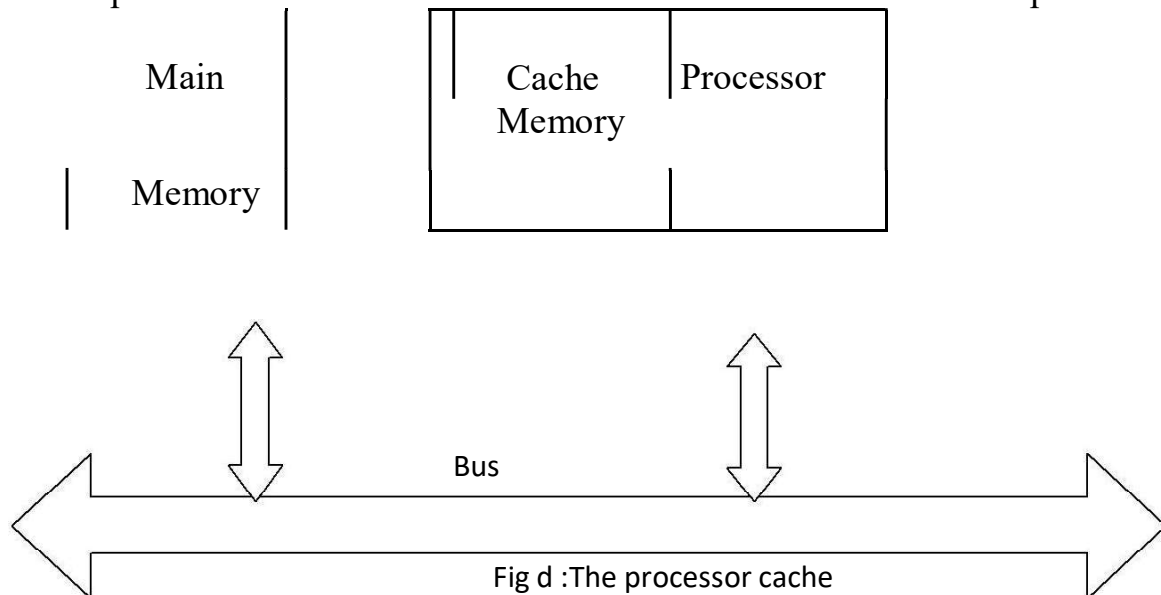


Figure 1.3.4 High speed bus configuration

➤ Performance

The most important measure of the performance of a computer is how quickly it can execute programs. The speed with which a computer executes program is affected by the design of its hardware. For best performance, it is necessary to design the compiles, the machine instruction set, and the hardware in a coordinated way.

The total time required to execute the program is elapsed time is a measure of the performance of the entire computer system. It is affected by the speed of the processor, the disk and the printer. The time needed to execute a instruction is called the processor time



The pertinent parts of the fig. c is repeated in fig. d which includes the cache memory as part of the processor unit.

Let us examine the flow of program instructions and data between the memory and the processor. At the start of execution, all program instructions and the required data are stored in the main memory. As the execution proceeds, instructions are fetched one by one over the

bus into the processor, and a copy is placed in the cache later if the same instruction or data item is needed a second time, it is read directly from the cache.

The processor and relatively small cache memory can be fabricated on a single IC chip. The internal speed of performing the basic steps of instruction processing on chip is very high and is considerably faster than the speed at which the instruction and data can be fetched from the main memory. A program will be executed faster if the movement of instructions and data between the main memory and the processor is minimized, which is achieved by using the cache.

For example:- Suppose a number of instructions are executed repeatedly over a short period of time as happens in a program loop. If these instructions are available in the cache, they can be fetched quickly during the period of repeated use. The same applies to the data that are used repeatedly.

Processor clock

Processor circuits are controlled by a timing signal called clock. The clock designer the regular time intervals called clock cycles. To execute a machine instruction the processor divides the action to be performed into a sequence of basic steps that each step can be completed in one clock cycle. The length P of one clock cycle is an important parameter that affects the processor performance.

Pipelining and super scalar operation

We assume that instructions are executed one after the other. Hence the value of S is the total number of basic steps, or clock cycles, required to execute one instruction.

A substantial improvement in performance can be achieved by overlapping the execution of successive instructions using a technique called pipelining.

Consider Add $R_1 R_2 R_3$

This adds the contents of R_1 & R_2 and places the sum into R_3 .

The contents of R_1 & R_2 are first transferred to the inputs of ALU. After the addition operation is performed, the sum is transferred to R_3 . The processor can read the next instruction from the memory, while the addition operation is being performed. Then of that instruction also uses, the ALU, its operand can be transferred to the ALU inputs at the same time that the add instructions is being transferred to R_3 .

In the ideal case if all instructions are overlapped to the maximum degree possible the execution proceeds at the rate of one instruction completed in each clock cycle. Individual instructions still require several clock cycles to complete. But for the purpose of computing T , effective value of S is 1.

Clock rate:- These are two possibilities for increasing the clock rate 'R'.

1. Improving the IC technology makes logical circuit faster, which reduces the time of execution of basic steps. This allows the clock period P, to be reduced and the clock rate R to be increased.
2. Reducing the amount of processing done in one basic step also makes it possible to reduce the clock period P. however if the actions that have to be performed by an instructions remain the same, the number of basic steps needed may increase.

Increase in the value 'R' that are entirely caused by improvements in IC technology affects all aspects of the processor's operation equally with the exception of the time it takes to access the main memory. In the presence of cache the percentage of accesses to the main memory is small. Hence much of the performance gain excepted from the use of faster technology can be realized.

Instruction set CISC & RISC

Simple instructions require a small number of basic steps to execute. Complex instructions involve a large number of steps. For a processor that has only simple instruction a large number of instructions may be needed to perform a given programming task. This could lead to a large value of 'N' and a small value of 'S' on the other hand if individual instructions perform more complex operations, a fewer instructions will be needed, leading to a lower value of N and a larger value of S. It is not obvious if one choice is better than the other. But complex instructions combined with pipelining (effective value of S 1) would achieve one best performance. However, it is much easier to implement efficient pipelining in processors with simple instruction sets.

Performance measurements

It is very important to be able to access the performance of a computer, comp designers use performance estimates to evaluate the effectiveness of new features.

Inspite of the performance equation being so simple, the evaluation of 'T' is highly complex. Moreover the parameters like the clock speed and various architectural features are not reliable indicators of the expected performance.

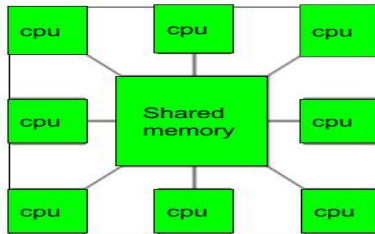
The performance measure is the time taken by the computer to execute a given bench mark. Initially some attempts were made to create artificial programs that could be used as bench mark programs. But synthetic programs do not properly predict the performance obtained when real application programs are run.

The program selected range from game playing, compiler, and data base applications to numerically intensive programs in astrophysics and quantum chemistry. In each case, the program is compiled under test, and the running time on a real computer is measured. The same program is also compiled and run on one computer selected as reference.

➤ Multiprocessor and Multicomputer

1. Multiprocessor:

A Multiprocessor is a computer system with two or more central processing units (CPUs) share full access to a common RAM. The main objective of using a multiprocessor is to boost the system's execution speed, with other objectives being fault tolerance and application matching. There are two types of multiprocessors, one is called shared memory multiprocessor and another is distributed memory multiprocessor. In shared memory multiprocessors, all the CPUs shares the common memory but in a distributed memory multiprocessor, every CPU has its own private memory.



Applications of Multiprocessor –

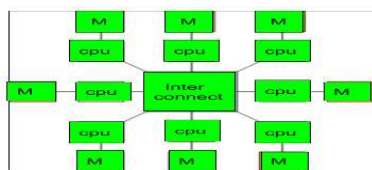
1. As a uniprocessor, such as single instruction, single data stream (SISD).
2. As a multiprocessor, such as single instruction, multiple data stream (SIMD), which is usually used for vector processing.
3. Multiple series of instructions in a single perspective, such as multiple instruction, single data stream (MISD), which is used for describing hyper-threading or pipelined processors.
4. Inside a single system for executing multiple, individual series of instructions in multiple perspectives, such as multiple instruction, multiple data stream (MIMD).

Benefits of using a Multiprocessor –

- Enhanced performance.
- Multiple applications.
- Multi-tasking inside an application.
- High throughput and responsiveness.
- Hardware sharing among CPUs.

3. Multicomputer:

A [multicomputer system](#) is a computer system with multiple processors that are connected together to solve a problem. Each processor has its own memory and it is accessible by that particular processor and those processors can communicate with each other via an interconnection network.



As the multicomputer is capable of messages passing between the processors, it is possible to divide the task between the processors to complete the task. Hence, a multicomputer can be used for distributed computing. It is cost effective and easier to build a multicomputer than a multiprocessor.

Difference between multiprocessor and Multicomputer:

1. Multiprocessor is a system with two or more central processing units (CPUs) that is capable of performing multiple tasks where as a multicomputer is a system with multiple processors that are attached via an interconnection network to perform a computation task.
2. A multiprocessor system is a single computer that operates with multiple CPUs where as a multicomputer system is a cluster of computers that operate as a singular computer.
3. Construction of multicomputer is easier and cost effective than a multiprocessor.
4. In multiprocessor system, program tends to be easier where as in multicomputer system, program tends to be more difficult.
5. Multiprocessor supports parallel computing, Multicomputer supports distributed computing.

➤ Historical Perspective

Introduction:

A computer is an electronic device that manipulates information or data. It has the ability to store, retrieve, and process data. Nowadays, a computer can be used to type documents, send email, play games, and browse the Web. It can also be used to edit or create spreadsheets, presentations, and even videos. But the evolution of this complex system started around 1940 with the first Generation of Computer and evolving ever since.

There are five generations of computers.

1. FIRST GENERATION

• Introduction:

1. 1946-1959 is the period of first generation computer.
2. J.P.Eckert and J.W.Mauchy invented the first successful electronic computer called ENIAC, ENIAC stands for “Electronic Numeric Integrated And Calculator”.

• Few Examples are:

1. ENIAC ,EDVAC,UNIVAC,IBM-701,IBM-650

• Advantages:

1. It made use of vacuum tubes which are the only electronic component available during those days.
2. These computers could calculate in milliseconds.

• Disadvantages:

1. These were very big in size, weight was about 30 tones.
 2. These computers were based on vacuum tubes.
 3. These computers were very costly.
 4. It could store only a small amount of information due to the presence of magnetic drums.
 5. As the invention of first generation computers involves vacuum tubes, so another disadvantage of these computers was, vacuum tubes require a large cooling system.
 6. Very less work efficiency.
 7. Limited programming capabilities and punch cards were used to take inputs.
 8. Large amount of energy consumption.
 9. Not reliable and constant maintenance is required.
-

2. SECOND GENERATION

- **Introduction**

1. 1959-1965 is the period of second-generation computer.
2. 3. Second generation computers were based on Transistor instead of vacuum tubes.

- **Few Examples are:**

Honeywell 400, IBM 7094, CDC 1604, CDC 3600, UNIVAC 1108

- **Advantages:**

1. Due to the presence of transistors instead of vacuum tubes, the size of electron component decreased. This resulted in reducing the size of a computer as compared to first generation computers.
2. Less energy and not produce as much heat as the first generation.
3. Assembly language and punch cards were used for input.
4. Low cost than first generation computers.
5. Better speed, calculate data in microseconds.
6. Better portability as compared to first generation

- **Disadvantages:**

1. A cooling system was required.
2. Constant maintenance was required.
3. Only used for specific purposes.

3. THIRD GENERATION

- **Introduction:**

1. 1965-1971 is the period of third generation computer.
2. These computers were based on Integrated circuits.
3. IC was invented by Robert Noyce and Jack Kilby In 1958-1959.
4. IC was a single component containing number of transistors.

- **Few Examples are:**

1. PDP-8, PDP-11, ICL 2900, IBM 360, IBM 370

- **Advantages:**

1. These computers were cheaper as compared to second-generation computers.
2. They were fast and reliable.
3. Use of IC in the computer provides the small size of the computer.
4. IC not only reduce the size of the computer but it also improves the performance of the computer as compared to previous computers.
5. This generation of computers has big storage capacity.
6. Instead of punch cards, mouse and keyboard are used for input.
7. They used an operating system for better resource management and used the concept of time-sharing and multiple programming.
8. These computers reduce the computational time from microseconds to nanoseconds.

- **Disadvantages:**

1. IC chips are difficult to maintain.
2. The highly sophisticated technology required for the manufacturing of IC chips.
3. Air conditioning is required.

4. FOURTH GENERATION

- **Introduction:**

1. 1971-1980 is the period of fourth generation computer.
-

2. This technology is based on Microprocessor.
 3. A microprocessor is used in a computer for any logical and arithmetic function to be performed in any program.
 4. Graphics User Interface (GUI) technology was exploited to offer more comfort to users.
- **Few Examples are:**
IBM 4341, DEC 10, STAR 1000, PUP 11
 - **Advantages:**
 1. Fastest in computation and size get reduced as compared to the previous generation of computer.
 2. Heat generated is negligible.
 3. Small in size as compared to previous generation computers.
 4. Less maintenance is required.
 5. All types of high-level language can be used in this type of computers.
 - **Disadvantages:**
 1. The Microprocessor design and fabrication are very complex.
 2. Air conditioning is required in many cases due to the presence of ICs.
 3. Advance technology is required to make the ICs.

5. FIFTH GENERATION

- **Introduction:**
 1. The period of the fifth generation in 1980-onwards.
 2. This generation is based on artificial intelligence.
 3. The aim of the fifth generation is to make a device which could respond to natural language input and are capable of learning and self-organization.
 4. This generation is based on ULSI(Ultra Large Scale Integration) technology resulting in the production of microprocessor chips having ten million electronic component.
 - **Few Examples are:**
 1. Desktop, Laptop, NoteBook, UltraBook, Chromebook
 - **Advantages:**
 1. It is more reliable and works faster.
 2. It is available in different sizes and unique features.
 3. It provides computers with more user-friendly interfaces with multimedia features.
 - **Disadvantages:**
 1. They need very low-level languages.
 2. They may make the human brains dull and doomed
-

UNIT-II

Addressing Methods and Machine Program Sequencing:

➤ Basic Concepts

Number Representation : 3 systems are used for representing numbers

1. Sign-and –magnitude
2. 1 s complement
3. 2 s complement

In all three systems ,the leftmost bit is 0 for positive numbers and 1 for negative numbers. Positive values represents have same representations in all systems ,but negative values are different ,in sign-and-magnitude system ,negative values are represented by changing the most significant bit from 0 to 1 in the B vector of the corresponding positive value,negative values are obtained by complementing each bit of the corresponding positive number.

Addition of positive numbers:

$$\begin{array}{r}
 \begin{array}{cccc}
 0 & 1 & 0 & 1 \\
 + & 0 & +0 & +1 \\
 \hline
 0 & 1 & 1 & 10
 \end{array}
 \end{array}$$

Carry out

Addition and subtraction of signed numbers:

For representing positive and negative numbers,or simply, signed numbers are same but negative values representation is differ

$$\begin{array}{r}
 \text{(a)} \quad \begin{array}{ccc}
 0010 & (+2) \\
 0011 & (+3) \\
 \hline
 0101 & (+5)
 \end{array}
 \end{array}$$

$$\begin{array}{r}
 \text{(b)} \quad \begin{array}{ccc}
 0100 & (+4) \\
 +1010 & (-6) \\
 \hline
 1110 & (-2)
 \end{array}
 \end{array}$$

$$\begin{array}{r}
 \text{(c)} \quad \begin{array}{ccc}
 1011 & (-5) \\
 1110 & (-2) \\
 \hline
 \end{array}
 \end{array}$$

$$\begin{array}{r}
 \text{(b)} \quad \begin{array}{ccc}
 0111 & (+7) \\
 +1101 & (-3) \\
 \hline
 \end{array}
 \end{array}$$

1 0 0 1 (+5)

0 1 0 0 (+4)

2's complement add and subtract operations .

➤ Memory locations and address:

Memory locations and addresses:

Number and character operands, as well as instructions, are stored in the memory of a computer. The memory consists of many millions of storage cells, each of which can store a bit of information having the value 0 or 1. Because a single bit represents a very small amount of information, bits are seldom handled individually. The usual approach is to deal with them in groups of fixed size. For this purpose, 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 (a).

Modern computers have word lengths that typically range from 16 to 64 bits. If the word length of a computer is 32 bits, a single word can store a 32-bit 2's complement number or four ASCII characters, each occupying 8 bits. A unit of 8 bits is called a byte.

Accessing the memory to store or retrieve a single item of information, either a word or a byte, requires distinct names or addresses for each item location. It is customary to use numbers from 0 through 2^k-1 , for some suitable values of k , as the addresses of successive locations in the memory. The 2^k addresses constitute the address space of the computer, and the memory can have up to 2^k addressable locations. 24-bit address generates an address space of 2^{24} (16,777,216) locations. A 32-bit address creates an address space of 2^{32} or 4G (4 giga) locations.

BYTE ADDRESSABILITY: -

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

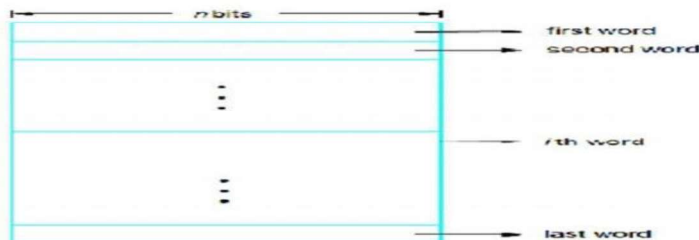
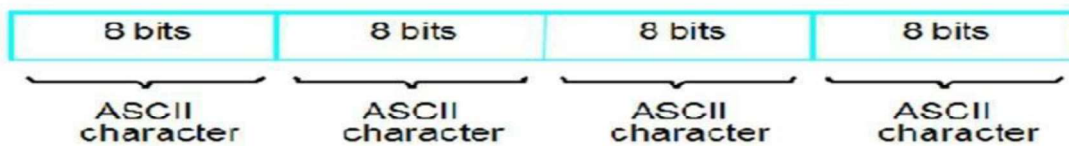
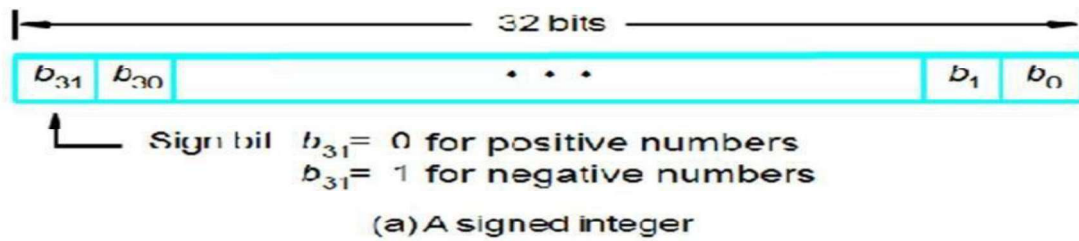
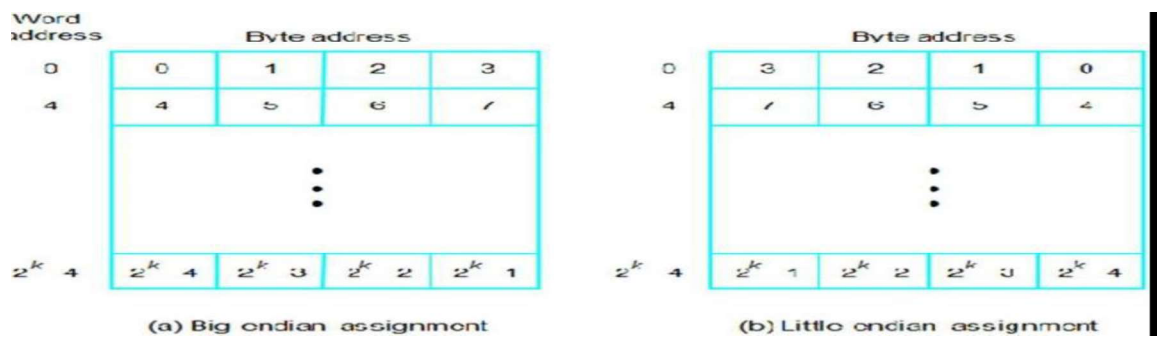


Fig a Memory words



(b) Four characters

Locations in the memory. This is the assignment used in most modern computers, and is the one we will normally use in this book. 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, as shown in fig b. 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 addition to specifying the address ordering of bytes within a word, it is also necessary to specify the labeling of bits within a byte or a word. The same ordering is also used for labeling bits within a byte, that is, b_7 , b_6 , ..., b_0 , from left to right.

Fig. byte and word processing.

WORD ALIGNMENT:-

In the case of a 32-bit word length, natural word boundaries occur at addresses 0,4, 8, ..., as shown in above fig. We say that the word locations have aligned addresses. In general, words are said to be aligned in memory if they begin at a byte address that is

a multiple of the number of bytes in a word. The memory 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 bytes addresses 0,8,16 There is no fundamental reason why words cannot begin at an arbitrary byte address. In that case, words are said to have unaligned addresses. While the most common case is to use aligned addresses, some computers allow the use of unaligned word addresses.

ACCESSING NUMBERS, CHARACTERS, AND CHARACTER STRINGS:-

A number usually occupies one word. It can be accessed in the memory by specifying its word address. Similarly, individual characters can be accessed by their byte address.

In many applications, it is necessary to handle character strings of variable length. The beginning of the string is indicated by giving the address of the byte containing its first character. Successive byte locations contain successive characters of the string. There are two ways to indicate the length of the string.

A special control character with the meaning “end of string” can be used as the last character in the string, or a separate memory word location or processor register can contain a number indicating the length of the string in bytes.

➤ **Main Memory Operations**

➤ **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

REGISTER TRANSFER NOTATION

Transfer of information from one location in the computer to another. Possible locations that may be involved in such transfers are memory locations that may be involved in such transfers are memory locations, processor registers, or registers in the I/O subsystem. Most of the time, we identify a location by a symbolic name standing for its hardware binary address. Example, names for the addresses of memory locations may be LOC, PLACE, A, VAR2; processor registers names may be R0, R5; and I/O register names may be DATAIN, OUTSTATUS, and so on. The contents of a location are denoted by placing square brackets around the name of the location. Thus, the expression $R1 \leftarrow [LOC]$ 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 then 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 right-hand side 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

Another type of notation to represent machine instructions and programs. For this, we use an assembly language format. For example, an 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

BASIC INSTRUCTIONS

The operation of adding two numbers is a fundamental capability in any computer. The statement $C = A + B$ In a high-level language program is a command to the computer to add the current values of the two variables called A and B, and to assign the sum to a third variable, C. When the program containing this statement is compiled, the three variables, A, B, and C, are assigned to distinct locations in the memory. We will use the variable names to refer to the corresponding memory location addresses. The contents of these locations represent the values of the three variables. Hence, the above highlevel language statement requires the action. $C \leftarrow [A] + [B]$ To carry out this action, the contents of memory locations A and B are fetched from the memory and transferred into the processor where their sum is computed. This result is then sent back to the memory and stored in location C. Let us first assume that this action is to be accomplished by a single machine instruction. Furthermore, assume that this instruction contains the memory addresses of the three operands – A, B, and C. This three-address instruction can be represented symbolically as Add A, B, C Operands A and B are called the source operands, C is called the destination operand, and Add is the operation to be performed on the operands. A general instruction of this type has the format. Operation Source1, Source 2, Destination 23 If k bits are needed for specify the memory address of each operand, the encoded form of the above instruction must contain 3k bits for addressing purposes in addition to the bits needed to denote the Add operation. An alternative approach is to use a sequence of simpler instructions to perform the same task, with each instruction having only one or two operands Suppose that two- address instructions of the form Operation Source, Destination Are available. An Add instruction of this type is Add A, B Which performs operation $B \leftarrow [A] + [B]$. A single two-address instruction cannot be used to solve our original problem, which is to add the the contents of locations A and B, without destroying either of them, and to place the sum in location C. The problem can be solved by using another two-address instruction that copies the contents of one memory location into another. Such an instruction is Move B, C Which performs the operations $C \leftarrow [B]$, leaving the contents of location B unchanged. Using only one-address instructions, the operation $C \leftarrow [A] + [B]$ can be performed by executing the sequence of instructions Load A Add B Store C Some early computers were designed around a single accumulator structure. Most modern computers have a number of general-purpose processor registers – typically 8 to 32, and even considerably more in some cases. Access to data in these registers is much faster than to data stored in memory locations because the registers are inside the processor. Let R_i represent a general-purpose register. The instructions Load A, R_i Store R_i , A and Add A, R_i Are generalizations of the Load, Store, and Add instructions for the single accumulator case, in which register R_i performs the

function of the accumulator. When a processor has several generalpurpose registers, many instructions involve only operands that are in the register. In fact, in many modern processors, computations can be performed directly only on data held in processor registers. Instructions such as 24 Add Ri, Rj Or Add Ri, Rj, Rk In both of these instructions, the source operands are the contents of registers Ri and Rj. In the first instruction, Rj also serves as the destination register, whereas in the second instruction, a third register, Rk, is used as the destination. It is often necessary to transfer data between different locations. This is achieved with the instruction Move Source, Destination When data are moved to or from a processor register, the Move instruction can be used rather than the Load or Store instructions because the order of the source and destination operands determines which operation is intended. In processors where arithmetic operations are allowed only on operands that are processor registers, the $C = A + B$ task can be performed by the instruction sequence Move A, Ri Move B, R Add Ri, Rj Move Rj, C In processors where one operand may be in the memory but the other must be in register, an instruction sequence for the required task would be Move A, Ri Add B, Ri Move Ri, C The speed with which a given task is carried out depends on the time it takes to transfer instructions from memory into the processor and to access the operands referenced by these instructions. Transfers that involve the memory are much slower than transfers within the processor. We have discussed three-, two-, and one-address instructions. It is also possible to use instructions in which the locations of all operands are defined implicitly. Such instructions are found in machines that store operands in a structure called a pushdown stack. In this case, the instructions are called zero-address instructions.

INSTRUCTION EXECUTION AND STRAIGHT-LINE SEQUENCING

In the preceding discussion of instruction formats, we used to task $C \leftarrow [A] + [B]$ 25 the be shows a possible program segment for this task as it appears in the memory of a computer. We have assumed that the computer allows one memory operand fi g : APr o g r a m f o r $C \leftarrow A + B$ per instruction and has a number of processor registers. 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 start at addresses $i + 4$ and $i8$. Let us consider how this program is executed. The processor contains a register called the program counter (PC), which holds the address of the instruction to be executed next. To begin executing a program, the address of its first instruction (I in our example) m u s t 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. 26 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. The instruction in IR is examined to determine which operation is to be

performed. The specified operation is then performed by the processor. This often involves fetching operands from the memory or from processor registers, performing an arithmetic or logic operation, and storing the result in the destination location.

BRANCHING

Consider the task of adding a list of n numbers. Instead of using a long list of add instructions, it is possible to place a single add instruction in a program loop, as shown in fig b. The loop is a straight-line sequence of instructions executed as many times as needed. It starts at location LOOP and ends at the instruction Branch > 0 . During each pass through this loop, the address of the next list entry is determined, and that entry is fetched and added to Figure: A straight line programme of n numbers. 27 Fig Addition of n numbers using a loop. Assume that the number of entries in the list, n , is stored in memory location N, as shown. Register R1 is used as a counter to determine the number of time 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. This type of instruction loads a new value 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 28 executed. Branch > 0 LOOP (branch if greater than 0) is a conditional branch instruction that causes a branch to location LOOP if the result of the immediately preceding instruction, which is the decremented value in register R1, is greater than zero. This means that the loop is repeated, 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 value of zero, and hence, branching does not occur.

CONDITION CODES:

The processor keeps track of information 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 condition code flags. These flags are usually grouped together in a special processor register called the condition code register or status register. Individual condition code flags are set to 1 or cleared to 0, depending on 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 The instruction Branch > 0 , discussed in the previous section, is an example of a branch instruction that tests one or more of the condition flags. It causes a branch if the value tested is neither negative nor equal to zero. That is, the branch is taken if neither N nor Z is 1. The conditions are given as logic expressions involving the condition code flags In some computers, the condition code flags are affected automatically by instructions that

perform arithmetic or logic operations. However, this is not always the case. A number of computers have two versions of an Add instruction.

GENERATING MEMORY ADDRESSES

The purpose of the instruction block at LOOP is to add a different number from the list during each pass through the loop. Hence, the Add instruction in the block must refer to a different address during each pass. How are the addresses to be specified ? The memory operand address cannot be given directly 29 in a single Add instruction in the loop. Otherwise, it would need to be modified on each pass through the loop. The instruction set of a computer typically provides a number of such methods, called addressing modes. While the details differ from one computer to another, the underlying concepts are the same

➤ **Addressing modes**

In general, a program operates on data that reside in the computer’s memory. These data can be organized in a variety of ways. If we want to keep track of students’ names, we can write them in a list. These include lists, linked lists, arrays, queues, and so on.

Programs are normally written in a high-level language, which enables the programmer to use constants, local and global variables, pointers, and arrays. The different ways in which the location of an operand is specified in an instruction are referred to as addressing modes.

Name	Assembler syntax	Addressing		
functionImmediate		#	Value	
	Operand = Value			
Register	Ri	EA = Ri		
Absolute (Direct)	LO C	EA	=	LOC

Indirect	(R _i)	EA = [R _i]
	(LOC)	EA = [LOC]
Index	X(R _i)	EA = [R _i] + X
Base with index	(R _i , R _j)	EA = [R _i] + [R _j]
Base with index and offset	(R _i , R _j)	EA = [R _i] + [R _j] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(R _i) +	EA = [R _i]; Increment R _i
Autodecrement	- (R _i)	EA = [R _i]; decrement R _i

IMPLEMENTATION OF VARIABLE AND CONSTANTS

Variables and constants are the simplest data types and are found in almost every computer program. In assembly language, a variable is represented by allocating a register or memory location to hold its value. Thus, the 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.

Absolute mode – The operand is in a memory location; the address of this location is given explicitly in the instruction. (In some assembly languages, this mode is called Direct addressing mode)

The instruction

Move LOC, R2

Processor registers are used as temporary storage locations where the data in a register are accessed using the Register mode. The Absolute mode can represent global variables in a program. A declaration such as

Integer A, B;

Immediate mode – The operand is given explicitly in the instruction. For example, the instruction

Move 200, R0

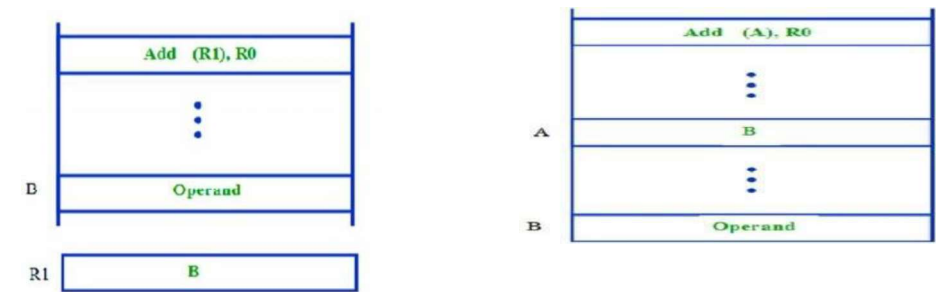
Places the value 200 in register R0. Clearly, the Immediate mode is only used to specify the value of a source operand. Using a subscript to denote the Immediate mode is not appropriate in assembly languages. A common convention is to use the sharp sign (#) in front of the value to indicate that this value is to be used as an immediate operand. Hence, we write the instruction above in the form

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. We refer to this address as the 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.

To execute the Add instruction in fig (a), the processor uses the value 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 fig (b). In this case, the processor first reads the contents of memory location A, then requests a second read operation using the value B



as an address to obtain the operand

Fig (a) Through a general-purpose register (b) Through a memory location

The memory that contains address of an called a pointer. and the use of are important powerful in programming.	Move	N,R1	register or location the operand is Indirection pointers and concepts
	Move	#NUM, R2	
	Clear	R0	
	LOOP ADD	(R2), R0	
	ADD	#4, R2	
	DECREMENT	R1	
	Branch > 0	LOOP	
	Move	R0, SUM	

In the program shown Register R2 is used as a pointer to the numbers in the list, and the operands are accessed indirectly through R2. The initialization section of the program loads the counter value n from memory location N into R1 and uses the immediate addressing mode to place the address value NUM1, which is the address of the first number in the list, into R2. Then it clears R0 to 0. The first two instructions in the loop implement the

loop, the instruction **Add (R2), R0** fetches the operand at location NUM1 and adds it to R0. The second Add instruction adds 4 to the contents of the pointer R2, so that it will contain the address value NUM2 when the above instruction is executed in the second pass through the loop.

Where B is a pointer variable. This statement may be compiled into

```
Move B,  
R1    Move  
(R1), A
```

Using indirect addressing through memory, the same action can be achieved with

```
Move (B), A
```

Indirect addressing through registers is used extensively. The above program shows the flexibility it provides. Also, when absolute addressing is not available, indirect addressing through registers makes it possible to access global variables by first loading the operand's address in a register.

INDEXING AND ARRAYS:-

A different kind of flexibility for accessing operands 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 use may be either a special register provided for this purpose, or, more commonly, it may be any one of a set of general-purpose registers in the processor. In either case, it is referred to as index register. We indicate the Index mode symbolically as

$X(R_i)$

Where X denotes the constant value contained in the instruction and R_i is the name of the register involved. The effective address of the operand is given by

$$EA = X + [R_j]$$

The contents of the index register are not changed in the process of generating the effective address. In an assembly language program, the constant X may be given either as an explicit number or as a symbolic name representing a numerical value.

Fig a illustrates two ways of using the Index mode. In fig a, the index register, R_1 , contains the address of a memory location, and the value X defines an offset (also called a displacement) from this address to the location where the operand is found. An alternative use is illustrated in fig b. Here, the constant X corresponds to a memory address, and the contents of the index register define the offset to 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.

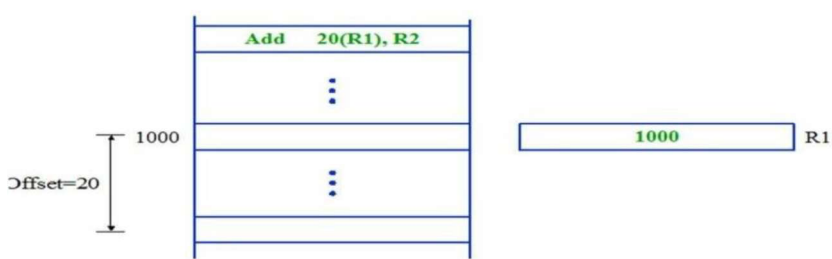
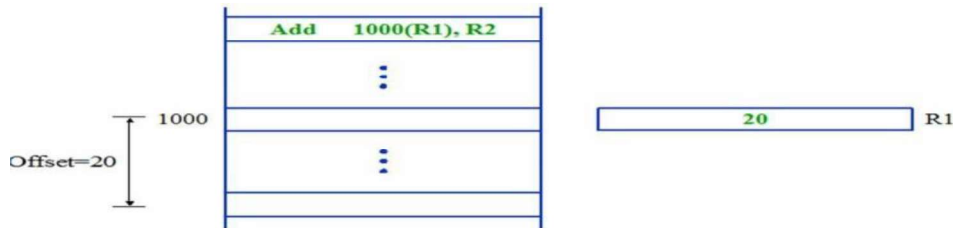


Fig (a) Offset is given as a constant



(b) Offset is in the index register

In the most basic form of indexed addressing several variations of this basic form provide a very efficient access to memory operands in practical programming situations. For example, a second register may be used to contain the offset X , in which case we can write the Index mode as

$$(R_i, R_j)$$

The effective address is the sum of the contents of registers R_i and R_j . The second register is usually called the base register. This form of indexed addressing provides more flexibility in accessing operands, because both components of the effective address can be changed.

Another version of the Index mode uses two registers plus a constant, which can be denoted as

$$X(R_i, R_j)$$

In this case, the effective address is the sum of the constant X and the contents of registers 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 the (R_i, R_j) part of the addressing mode. In other words, this mode implements a three-dimensional array.

RELATIVE ADDRESSING:-

We have defined the Index mode using general-purpose processor registers. A useful version of this mode is obtained 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.

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

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. This location can be computed by specifying it as an offset from the current value of the program counter. Since the branch target may be either before or after the branch instruction, the offset is given as a signed number.

Autoincrement 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 to point to the next item in a list.

(Ri)+

Autodecrement 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.

-(Ri)

```

        Move  N,R1
        Move  #NUM1,R2
    ▶    Clear  R0
        Add   (R2)+R0
        Decrement R1
        Branch>0  Loop
        Move    R0,SUM
```

Fig c The Autoincrement addressing mode used in the program

UNIT-3

Input/Output Organization:

➤ Accessing I/O Devices

A simple arrangement to connect I/O devices to a computer is to use a single bus arrangement. 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 line, 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, when I/O devices and

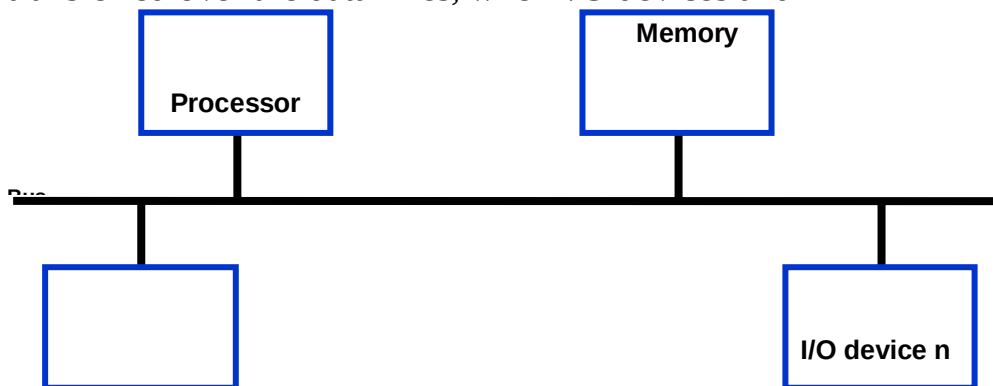


Fig Single Bus Structure

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. When building a computer system based on these processors, the designer had 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 I/O devices examine the low-order bits of the address bus to determine whether they should respond.

The hardware required to connect an I/O device to the bus. 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.

➤ **Interrupts**

To avoid the processor being not performing any useful computation, a hardware signal called an *interrupt* to the processor can do it. At least one of the bus control lines, called an interrupt-request line, is usually dedicated for this purpose.

An interrupt-service routine usually is needed and is executed when an interrupt request is issued. On the other hand, the processor must inform the device that its request has been recognized so that it may remove its interrupt-request signal. An interrupt-acknowledge signal serves this function.

The routine executed in response to an interrupt request is called the interrupt-service routine, which is the PRINT routine in our example. Interrupts bear considerable resemblance to subroutine calls. Assume that an interrupt request arrives during execution of instruction *i* in figure 1.

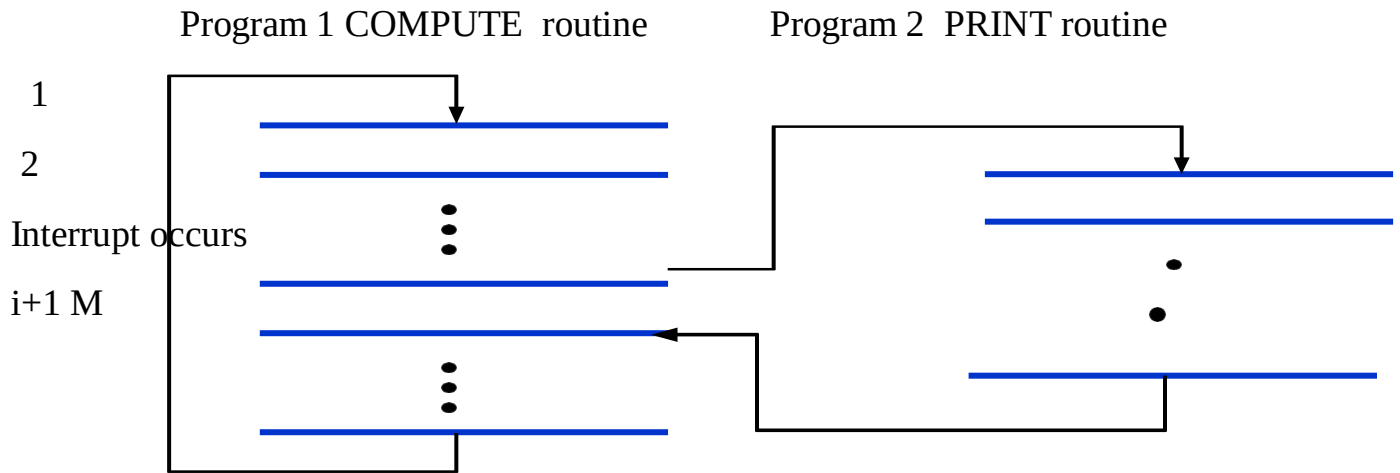


Figure 1. Transfer of control through the use of interrupts

The processor first completes execution of instruction i . Then, it loads the program counter with the address of the first instruction of the interrupt-service routine. For the time being, let us assume that this address is hardwired in the processor. After execution of the interrupt-service routine, the processor has to come back to instruction $i + 1$. Therefore, when an interrupt occurs, the current contents of the PC, which point to instruction $i + 1$, must be put in temporary storage in a known location. A Return-from-interrupt instruction at the end of the interrupt-service routine reloads the PC from the temporary storage location, causing execution to resume at instruction $i + 1$. In many processors, the return address is saved on the processor stack.

We should note that as part of handling interrupts, the processor must inform the device that its request has been recognized so that it may remove its interrupt-request

signal. This may be accomplished by means of a special control signal on the bus. An interrupt-acknowledge signal. The execution of an instruction in the interrupt-service routine that accesses a status or data register in the device interface implicitly informs that device that its interrupt request has been recognized.

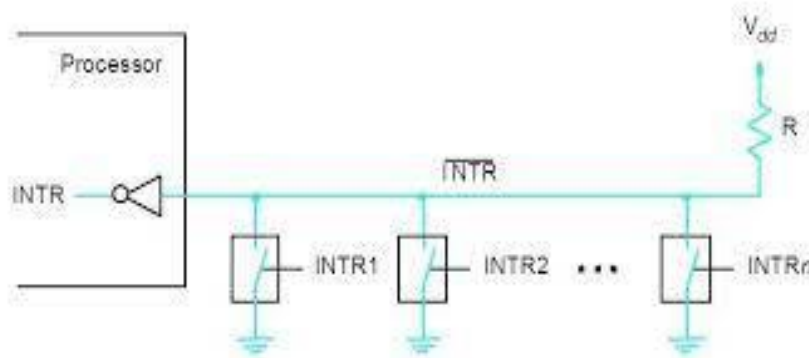
So far, treatment of an interrupt-service routine is very similar to that of a subroutine. An important departure from this similarity should be noted. A subroutine performs a function required by the program from which it is called.

However, the interrupt-service routine may not have anything in common with the program being executed at the time the interrupt request is received. In fact, the two programs often belong to different users. Therefore, before starting execution of the interrupt-service routine, any information that may be altered during the execution of that routine must be saved. This information must be restored before execution of the interrupt program is resumed. In this way, the original program can continue execution without being affected in any way by the interruption, except for the time delay. The information that needs to be saved and restored typically includes the condition code flags and the contents of any registers used by both the interrupted program and the interrupt-service routine.

The task of saving and restoring information can be done automatically by the processor or by program instructions. Most modern processors save only the minimum amount of information needed to maintain the registers involves memory transfers that increase the total execution time, and hence represent execution overhead. Saving registers also increase the delay between the time an interrupt request is received and the start of execution of the interrupt-service routine. This delay is called interrupt latency.

1)Interrupt Hardware

An equivalent circuit for an open-drain bus used to implement a common interrupt-request line



In a hardware interrupt, all the devices are connected to the Interrupt Request Line. A single request line is used for all the n devices. To request an interrupt, a device closes its associated switch. When a device requests an interrupt, the value of INTR is the logical OR of the requests from individual devices.

2)Handling Multiple Devices Let us now consider the situation where a number of devices capable of initiating interrupts are connected to the processor. Because these devices are operationally independent, there is no definite order in which they will generate interrupts. For example, device X may request an interrupt while an interrupt caused by device Y is being serviced, or several devices may

request interrupts at exactly the same time. This gives rise to a number of questions:

1. How can the processor recognize the device requesting an interrupt?
 2. Given that different devices are likely to require different interrupt -service routines, how can the processor obtain the starting address of the appropriate routine in each case?
 3. Should a device be allowed to interrupt the processor while another interrupt is being serviced?
 4. How should two or more simultaneous interrupt requests be handled?
- If two devices have activated the line at the same time, it must be possible to break the tie and elect one of the two requests for service. When the interrupt-service routine for the selected device has been completed, the second request can be serviced.

Vectored interrupts: To reduce the time involved in the polling process, a device requesting an interrupt may identify itself directly to the processor. Then, the processor can immediately start executing the corresponding interrupt-service routine. The term vectored interrupts refers to all interrupt - handling schemes based on this approach.

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 its memory where the addresses for interrupt-service routines are located.

This arrangement implies that the interrupt-service routine for a given device must always start at the same location. The programmer can gain some flexibility by storing in this location an instruction that causes a branch to the appropriate routine. In many computers, this is done automatically by the interrupt-handling mechanism. 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 special type of interrupt called a privilege exception.

A multiple-priority scheme can be implemented easily by using separate interrupt- request and interrupt-acknowledge lines for each device, as shown in Figure 5.7. Each of the interruptrequest 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.

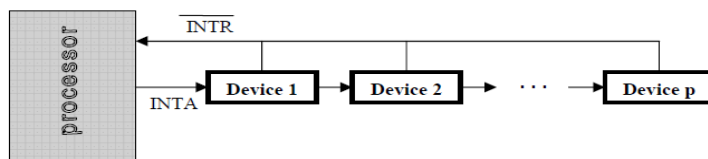


Figure Implementation of interrupt priority using individual interrupt-request and acknowledge lines.

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, as in Figure 5.6, 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, as shown in Figure 5.8. 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.

3)Enabling and Disabling interrupts

There are many situations in which the processor should ignore interrupt requests. For example, in the case of the Compute-Print program of Figure 5.5, an interrupt request from the printer should be accepted only if there are output lines to be printed. After printing the last line of a set of n lines, interrupts should be disabled until another set becomes available for printing.

A simple way is to provide machine instructions, such as Interrupt-enable and Interruptdisable. The processor hardware ignores the interrupt-request line until the execution of the first instruction of the interrupt-service routine has been completed. Then, by using an Interrupt-disable instruction as the first instruction in the interrupt-service routine, the programmer can ensure that no further interruptions will occur until an Interrupt-enable instruction is executed. Typically, the Interrupt- enable instruction will be the last instruction in the interrupt-service routine before the Return-from-interrupt instruction. The processor must guarantee that execution of the Returnfrom- interrupt instruction is completed before –further interruption can occur.

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, indicates whether interrupts are enabled. An interrupt request received while this bit is equal to 1 will be accepted. 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-frominterrupt instruction 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 interrupthandling circuit responds only to the leading edge of the signal. Such a line is said to be edgetriggered. In this case, the processor will receive only one request, regardless of how long the line is activated. Hence, there is no danger of multiple interruptions and no need to explicitly disable interrupt requests from this line. Before proceeding to study more complex aspects of interrupts, let us summarize the sequence of events involved in handling an interrupt request from a single device.

Assuming that interrupts are enabled, the following is a typical scenario:

1. The device raises an interrupt request.
2. The processor interrupts the program currently being executed.
3. Interrupts are disabled by changing the control bits in the PS (except in the case of edgetriggered interrupts).
4. The device is informed that its request has been recognized, and in response, it deactivates the interrupt-request signal.
5. The action requested by the interrupt is performed by the interrupt-service routine.
6. Interrupts are enabled and execution of the interrupted program is resumed.

4)Controlling Devices Requests

The control needed is usually provided in the form of an interrupt-enable bit in the device's interface circuit. The keyboard interrupt-enable, KEN, and display interrupt-enable, DEN, flags in register CONTROL in Figure 5.3 perform this function. If either of these flags is set, the interface circuit generates an interrupt request whenever the corresponding status flag in register STATUS is set. At the same time, the interface circuit sets bit KIRQ or DIRQ to indicate that the keyboard or display

unit, respectively, is requesting an interrupt. If an interrupt-enable bit is equal to 0, the interface circuit will not generate an interrupt request, regardless of the state of the status flag.

There are two independent mechanisms for controlling interrupt requests. At the device end, an interrupt-enable bit in a control register determines whether the device is allowed to generate an interrupt request. At the processor end, either an interrupt enable bit in the PS register or a priority structure determines whether a given interrupt request will be accepted.

5)Exceptions

The term exception is often used to refer to any event that causes an interruption. Hence, I/O interrupts are one example of an exception.

Recovery from errors: Computers use a variety of techniques to ensure that all hardware components are operating properly. For example, many computers include an error-checking code in the main memory, which allows detection of errors in the stored data. If an error occurs, the control hardware detects it and informs the processor by raising an interrupt. The processor may also interrupt a program if it detects an error or an unusual condition while executing the instructions of this program. For example, the OP-code field of an instruction may not correspond to any legal instruction, or an arithmetic instruction may attempt a division by zero. When exception processing is initiated as a result of such errors, the processor proceeds in exactly the same manner as in the case of an I/O interrupt request. It suspends the program being executed and starts an exception-service routine. This routine takes appropriate action to recover from the error, if possible, or to inform the user about it. Recall that in the case of an I/O interrupt, the processor completes execution of the instruction in progress before accepting the interrupt. However, when an interrupt is caused by an error, execution of the interrupted instruction cannot usually be completed, and the processor begins exception processing immediately.

Debugging: Another important type of exception is used as an aid in debugging programs. System software usually includes a program called a debugger, which helps the programmer find errors in a program. The debugger uses exceptions to provide two important facilities called trace and breakpoints. When a processor is operating in the trace mode, an exception occurs after execution of every instruction, using the debugging program as the exception-service routine. The debugging program enables the user to examine the contents of registers, memory locations, and so on. On return from the debugging program, the next

instruction in the program being debugged is executed, then the debugging program is activated again. The trace exception is disabled during the execution of the debugging program. Breakpoints provide a similar facility, except that the program being debugged is interrupted only at specific points selected by the user. An instruction called Trap or Software interrupt is usually provided for this purpose. Execution of this instruction results in exactly the same actions as when a hardware interrupt request is received. While debugging a program, the user may wish to interrupt program execution after instruction i . The debugging routine saves instruction i and replaces it with a software interrupt instruction. When the program is executed and reaches that point, it is interrupted and the debugging routine is activated. This gives the user a chance to examine memory and register contents. When the user is ready to continue executing the program being debugged, the debugging routine restores the saved instruction that was at location i and executes a Return-from-interrupt instruction.

Privilege exception: To protect the operating system of a computer from being corrupted by user programs, certain instructions can be executed only while the processor is in the supervisor mode. These are called privileged instructions. For example, when the processor is running in the user mode, it will not execute an instruction that changes the priority level of the processor or that enables a user program to access areas in the computer memory that have been allocated to other users. An attempt to execute such an instruction will produce a privilege exception, causing the processor to switch to the supervisor mode and begin executing an appropriate routine in the operating system.

6) Use of Interrupts in operating systems The operating system (OS) is responsible for coordinating all activities within a computer. It makes extensive use of interrupts to perform I/O operations and communicate with and control the execution of user programs. The operating system incorporates the interrupt-service routines (ISR) for all devices connected to a computer. Application programs do not perform I/O operations themselves. An operating system provides a variety of services to application programs. To facilitate the implementation of these services, most processors have several different software interrupt instructions, each with its own interrupt vector. They can be used to call different parts of the OS.

- In a computer that has both a supervisor and a user mode, the processor switches its operation to supervisor mode at the time it accepts an interrupt request. It does so by setting a bit in the processor status register after saving the old contents of that register on the stack. Thus, when an application program calls the as by a software interrupt instruction, the processor automatically switches to supervisor

mode, giving the as complete access to the computer's resources. When the as executes a Return-from-interrupt instruction, the processor status word belonging to the application program is restored from the stack. Multitasking is a mode of operation in which a processor executes several user programs at the same time. A common as technique that makes this possible is called time slicing. when operating system is started, an initialization routine OSINIT called for initialization. OSINIT loads the starting address of a routine called SCHEDULER in the interrupt vector corresponding to the timer interrupt. Hence, at the end of each time slice, the timer interrupt causes this routine to be executed.

- A program, together with any information that describes its current state of execution, is regarded by the as an entity called a process. A process can be in one of three states: Running, Runnable, or Blocked. The Running state means that the program is currently being executed. Assume that program A is in the Running state during a given time slice. At the end of that time slice, the timer interrupts the execution of this program and starts the execution of SCHEDULER. This is an operating system routine whose function is to determine which user program should run in the next time slice. It starts by saving all the information that will be needed later when execution of program A is resumed. The information saved, which is called the program state, includes register contents, the program counter, and the processor status word.
- SCHEDULER selects for execution some other program, B, that was suspended earlier and is in the Runnable state. It restores all information saved at the time program R was suspended, including the contents of PS and PC, and executes a Return-from-interrupt instruction. As a result, program B resumes execution for T seconds, at the end of which the timer clock raises an interrupt again, and a context switch to another runnable process takes place. Suppose that program A needs to read an input line from the keyboard. Instead of performing the operation itself, it requests I/O service from the operating system. It uses the stack or the processor registers to pass information to the OS describing the required operation, the I/O device, and the address of a buffer in the program data area where the line should be placed. Then it executes a software interrupt instruction. The interrupt vector for s instruction points to the OS SERVICES. This routine examines the information on the stack and initiates the requested operation by calling an appropriate OS routine. In our example, it calls, which is a routine responsible for starting I/O operations.
- While an I/O operation is in progress, the program that requested it cannot continue execution. Hence, the IOINIT routine sets the process associated with program A into the Blocked state, indicating to the scheduler that the program cannot resume execution at this time. The IOINIT routine carries out any

preparations needed for the I/O operation, such as initializing address pointers and byte count, then calls a routine that performs the I/O transfers. It is common practice in operating system design to encapsulate all software pertaining to a particular device into a self-contained module called the device driver. Such a module can be easily added to or deleted from the OS. We have assumed that the device driver for the keyboard consists of two routines, KBDINIT and KBDDATA, as shown in Figure 4.1. The IOINIT routine calls KBDINIT, which performs any initialization operations needed by the device or its interface circuit. KBDINIT also enables interrupts in the interface circuit by setting the appropriate bit in its control register, and then it returns to IOINIT, which returns to OSSERVICES. The keyboard is now ready to participate in a data transfer operation. It will generate an interrupt request whenever a key is pressed.

- Following the return to OSSERVICES, the SCHEDULER routine selects another user program to run. Of course, the scheduler will not select program A, because that program is now in the Blocked state. The Return-from-interrupt instruction that causes the selected user program to begin execution will also enable interrupts in the processor by loading new contents into the processor status register. Thus, an interrupt request generated by the keyboard's interface will be accepted. The interrupt vector for this interrupt points to an OS routine called IODATA. Because there could be several devices connected to the same interrupt request line, IODATA begins by polling these devices to determine the one requesting service. Then, it calls the appropriate device driver to service the request. In our example, the driver called will be KBDDATA, which will transfer one character of data. If the character is a Carriage Return, it will also set to 1 a flag called END, to inform IODATA that the requested I/O operation has been completed. At this point, the IODATA routine changes the state of process A from Blocked to Runnable, so that the scheduler may select it for execution in some future time slice.

➤ I/O Hardware

- In [computing](#), **input/output (I/O)**, or informally **io** or **IO** is the communication between an information processing system, such as a [computer](#), and the outside world, possibly a human or another information processing system. [Inputs](#) are the signals or data received by the system and outputs are the signals or [data](#) sent from it. The term can also be used as part of an action; to "perform I/O" is to perform an [input or output operation](#).
- **I/O devices** are the pieces of [hardware](#) used by a human (or other system) to communicate with a computer. For instance, a [keyboard](#) or [computer mouse](#) is an [input device](#) for a computer, while [monitors](#) and [printers](#) are [output devices](#).

Devices for communication between computers, such as [modems](#) and [network cards](#), typically perform both input and output operations.

- The designation of a device as either input or output depends on perspective. Mice and keyboards take physical movements that the human user outputs and convert them into input signals that a computer can understand; the output from these devices is the computer's input. Similarly, printers and monitors take signals that computers output as input, and they convert these signals into a representation that human users can understand. From the human [user's](#) perspective, the process of reading or seeing these representations is receiving output; this type of interaction between computers and humans is studied in the field of [human-computer interaction](#). A further complication is that a device traditionally considered an input device, e.g., card reader, keyboard, may accept control commands to, e.g., select stacker, display keyboard lights, while a device traditionally considered as an output device may provide status data, e.g., low toner, out of paper, paper jam.

The computer system's I/O architecture is its interface to the outside world. This architecture is designed to provide a systematic means of controlling interaction with the outside world and to provide the operating system with the information it needs to manage I/O activity effectively.

There are three principal I/O techniques: **programmed I/O**, in which I/O occurs under the direct and continuous control of the program requesting the I/O operation; **interrupt-driven I/O**, in which a program issues an I/O command and then continues to execute, until it is interrupted by the I/O hardware to signal the end of the I/O operations; and **direct memory access (DMA)**, in which a specialized I/O processor takes over control of an I/O operation to move a large block of data.

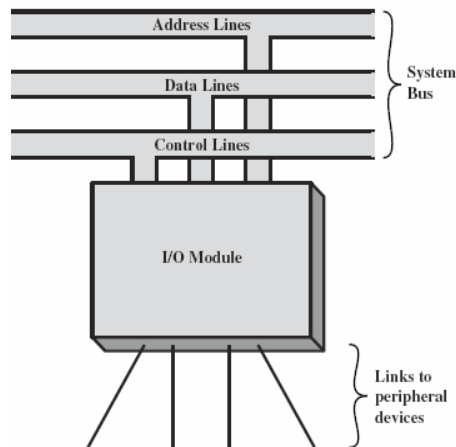
Two important examples of external I/O interfaces are FireWire and Infiniband.

Peripherals and the System Bus

- There are a wide variety of peripherals each with varying methods of operation
 - Impractical for the processor to accommodate all
- Data transfer rates are often slower than the processor and/or memory
 - Impractical to use the high-speed system bus to communicate directly
- Data transfer rates may be faster than that of the processor and/or memory
 - This mismatch may lead to inefficiencies if improperly managed
- Peripherals often use different data formats and word lengths

Purpose of I/O Modules

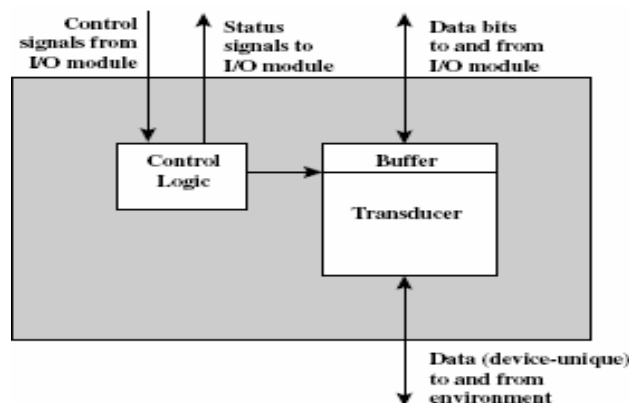
- Interface to the processor and memory via the system bus or control switch
- Interface to one or more peripheral devices



External Devices

External device categories

- **Human readable:** communicate with the computer user – CRT
- **Machine readable:** communicate with equipment – disk drive or tape drive
- **Communication:** communicate with remote devices – may be human readable or machine readable



The External Device – I/O Module

- **Control signals:** determine the function that will be performed
- **Data:** set of bits to be sent or received
- **Status signals:** indicate the state of the device
- **Control logic:** controls the device's operations
- **Transducer:** converts data from electrical to other forms of energy

- **Buffer:** temporarily holds data being transferred

Keyboard/Monitor

- Most common means of computer/user interaction
- Keyboard provides input that is transmitted to the computer
- Monitor displays data provided by the computer
- The character is the basic unit of exchange
- Each character is associated with a 7 or 8 bit code

Disk Drive

- Contains electronics for exchanging data, control, and status signals with an I/O module
- Contains electronics for controlling the disk read/write mechanism
- Fixed-head disk – transducer converts between magnetic patterns on the disk surface and bits in the buffer
- Moving-head disk – must move the disk arm rapidly across the surface

I/O Modules

Module Function

- Control and timing
- Processor communication
- Device communication
- Data buffering
- Error detection

I/O control steps

- Processor checks I/O module for external device status
- I/O module returns status
- If device ready, processor gives I/O module command to request data transfer
- I/O module gets a unit of data from device
- Data transferred from the I/O module to the processor

Processor communication

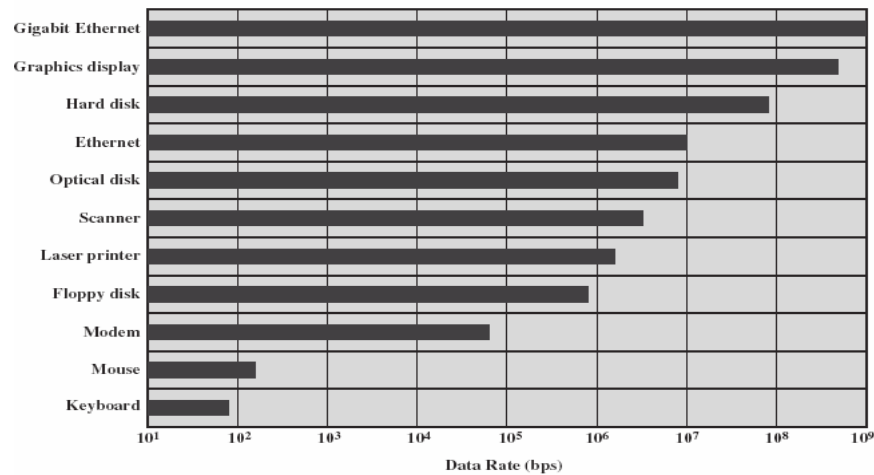
- **Command decoding:** I/O module accepts commands from the processor sent as signals on the control bus
- **Data:** data exchanged between the processor and I/O module over the data bus
- **Status reporting:** common status signals BUSY and READY are used because peripherals are slow

- **Address recognition:** I/O module must recognize a unique address for each peripheral that it controls

I/O module communication

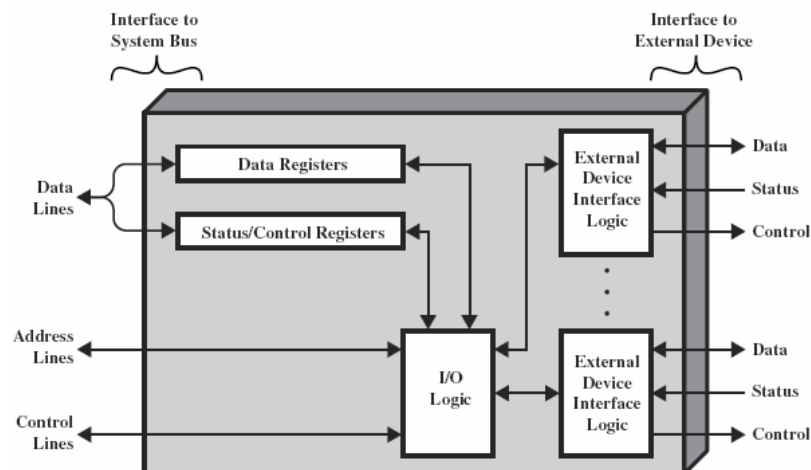
- **Device communication:** commands, status information, and data
- **Data buffering:** data comes from main memory in rapid burst and must be buffered by the I/O module and then sent to the device at the device's rate
- **Error detection:** responsible for reporting errors to the processor

Typical I/O Device Data Rates



I/O Module Structure

Block Diagram of an I/O Module



- Module connects to the computer through a set of signal lines – system bus
 - Data transferred to and from the module are buffered with data registers
 - Status provided through status registers – may also act as control registers
 - Module logic interacts with processor via a set of control signal lines
 - Processor uses control signal lines to issue commands to the I/O module
 - Module must recognize and generate addresses for devices it controls
 - Module contains logic for device interfaces to the devices it controls
-
- I/O module functions allow the processor to view devices as a simple-minded way
 - I/O module may hide device details from the processor so the processor only functions in terms of simple read and write operations – timing, formats, etc...
 - I/O module may leave much of the work of controlling a device visible to the processor – rewind a tape, etc...

I/O channel or I/O processor

- I/O module that takes on most of the detailed processing burden
- Used on mainframe computers

I/O controller or device controller

- Primitive I/O module that requires detailed control
- Used on microcomputers

Programmed I/O

Overview of Programmed I/O

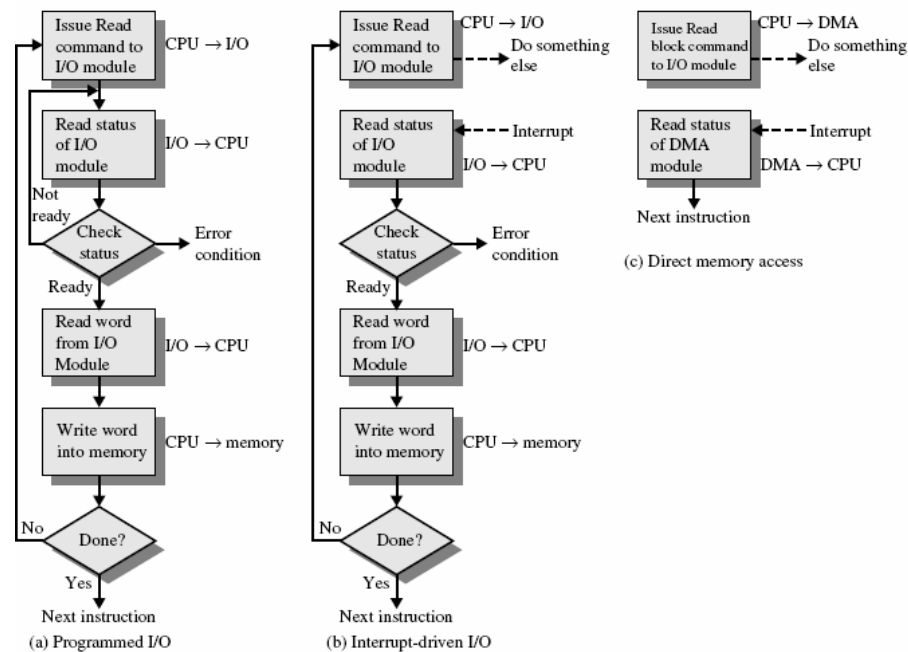
- Processor executes an I/O instruction by issuing command to appropriate I/O module
- I/O module performs the requested action and then sets the appropriate bits in the I/O status register – I/O module takes no further action to alert the processor – it does not interrupt the processor
- The processor periodically checks the status of the I/O module until it determines that the operation is complete

I/O Commands

The processor issues an address, specifying I/O module and device, and an

I/O command. The commands are:

- **Control:** activate a peripheral and tell it what to do
- **Test:** test various status conditions associated with an I/O module and its peripherals
- **Read:** causes the I/O module to obtain an item of data from the peripheral and place it into an internal register
- **Write:** causes the I/O module to take a unit of data from the data bus and transmit it to the peripheral



Three Techniques for Input of a Block of Data I/O Instructions :Processor views I/O operations in a similar manner as memory operations Each device is given a unique identifier or address

Processor issues commands containing device address – I/O module must check address lines to see if the command is for itself

I/O mapping

- Memory-mapped I/O
 - Single address space for both memory and I/O devices
 - disadvantage – uses up valuable memory address space
 - I/O module registers treated as memory addresses
 - Same machine instructions used to access both memory and I/O devices
 - advantage – allows for more efficient programming
 - Single read line and single write lines needed
 - Commonly used
- Isolated I/O
 - Separate address space for both memory and I/O devices
 - Separate memory and I/O select lines needed
 - Small number of I/O instructions
 - Commonly used

Interrupt-Driven I/O

- Overcomes the processor having to wait long periods of time for I/O modules
- The processor does not have to repeatedly check the I/O module status

I/O module view point

- I/O module receives a READ command from the processor
- I/O module reads data from desired peripheral into data register
- I/O module interrupts the processor
- I/O module waits until data is requested by the processor
- I/O module places data on the data bus when requested

Processor view point

- The processor issues a READ command
- The processor performs some other useful work
- The processor checks for interrupts at the end of the instruction cycle
- The processor saves the current context when interrupted by the I/O module
- The processor reads the data from the I/O module and stores it in memory
- The processor restores the saved context and resumes execution

- In computer architecture, the combination of the [CPU](#) and [main memory](#), to which the CPU can read or write directly using individual [instructions](#), is considered the brain of a computer. Any transfer of information to or from the CPU/memory combo, for example by reading data from a [disk drive](#), is considered I/O.^[1] The CPU and its supporting circuitry may provide [memory-mapped I/O](#) that is used in low-level [computer programming](#), such as in the implementation of [device drivers](#), or may provide access to [I/O channels](#). An [I/O algorithm](#) is one designed to exploit locality and perform efficiently when exchanging data with a secondary storage device, such as a disk drive

➤ Standard I/O Interface

Introduction: A number of standards have been developed for I/O Interface. IBM developed a they called ISA (Industry Standard Architecture) for their personal computer known at the time as PC AT. The popularity of that computer led to other manufacturers producing ISA -compatible interfaces for their 110 devices, thus making ISA into a de facto standard. Some standards have been developed through industrial cooperative efforts, even among competing companies driven by their common self-interest in having compatible products. In some cases, organizations such as the IEEE (Institute of Electrical and Electronics Engineers), ANSI (American National Standards Institute), or international bodies such as ISO (International Standards Organization) have blessed these standards and given them an official status. There are three widely used bus standards, PCI (Peripheral Component Interconnect), SCSI (Small Computer System Interface), and USB (Universal Serial Bus). The way these standards are used in a typical computer system is illustrated in Figure 5.28. The PCI standard defines an expansion bus on the motherboard. SCSI and USB are used for connecting additional devices, both inside and outside the computer box. The SCSI bus is a high-speed parallel bus intended for devices such as disks and video displays. The USB bus uses serial transmission to suit the needs of equipment ranging from keyboards to game controls to internet connections. The figure shows an interface circuit that enables devices compatible with the earlier ISA standard, such as the popular IDE (Integrated Device Electronics) disk, to be connected. It also shows a connection to an Ethernet. The Ethernet is a widely used local area network, providing a high-speed connection among computers in a building or a university campus. A given computer may use more than one bus standard. A typical Pentium computer has

both a PCI bus and an ISA bus, thus providing the user with a wide range of devices to choose from.

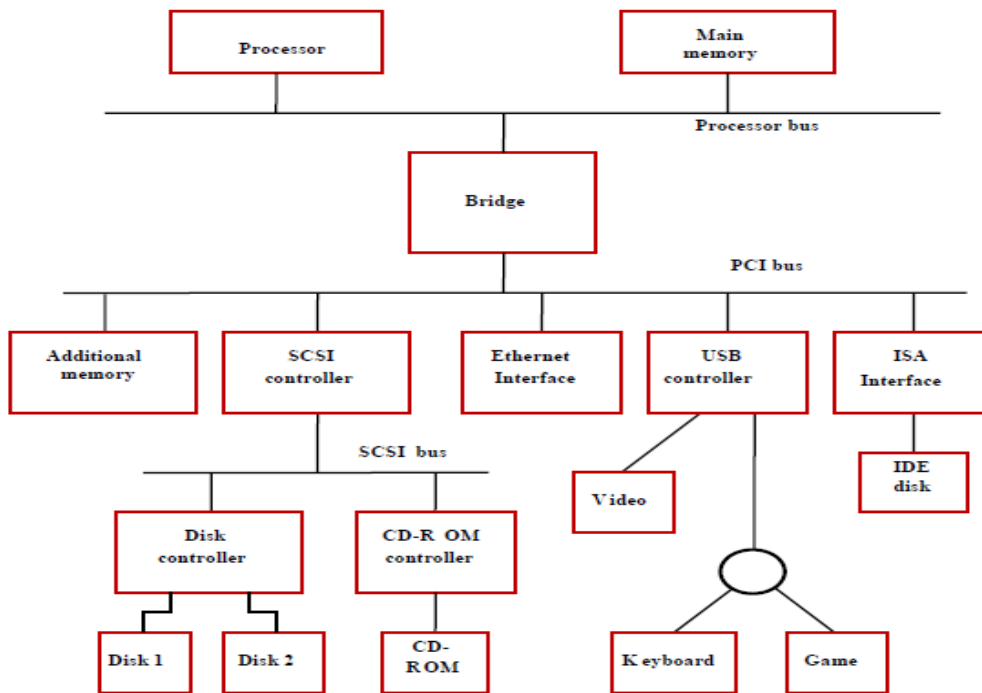


Figure 5.28 An example of a computer system using different interface standards.

5.7.1 PCI (Peripheral Component Interconnect)

1)Peripheral Component Interconnect(PCI) bus

The PCI bus is a good example of a system bus. It supports the functions found on a processor bus but in a standardized format that is independent of any particular processor. Devices connected to the PCI bus appear to the processor as if they were connected directly to the processor bus. They are assigned addresses in the memory address space of the processor. Early PCs used the 8-bit XT bus, whose signals closely mimicked those of Intel's 80x86 processors. Later, the 16-bit bus used on the PC AT computers became known as the ISA bus. Its extended 32-bit version is known as the EISA bus.

The PCI was developed as a low-cost bus that is truly processor independent. Its design anticipated a rapidly growing demand for bus bandwidth to support high-speed disks and graphic and video devices, as well as the specialized needs of multiprocessor systems. As a result, the PCI is still popular as an industry standard almost a decade after it was first introduced in 1992. An important feature that the PCI pioneered is a plug-and-play capability for connecting I/O devices. To

connect a new device, the user simply connects the device interface board to the bus. The software takes care of the rest.

Data Transfer: Most memory transfers involve a burst of data rather than just one word. The reason is that modern processors include a cache memory. The PCI is designed primarily to support this mode of operation. A read or a write operation involving a single word is simply treated as a burst of length one. The bus supports three independent address spaces: memory, I/O, and configuration. The first two are self-explanatory. The I/O address space is intended for use with processors, such as Pentium, that have a separate I/O address space. Figure 5.28 shows the main memory of the computer connected directly to the processor bus. An alternative arrangement that is used often with the PCI bus is shown in Figure 5.29. The PCI Bridge provides a separate physical connection for the main memory. For electrical reasons, the bus may be further divided into segments connected via bridges. However, regardless of which bus segment a device is connected to, it may still be mapped into the processor's memory address space.

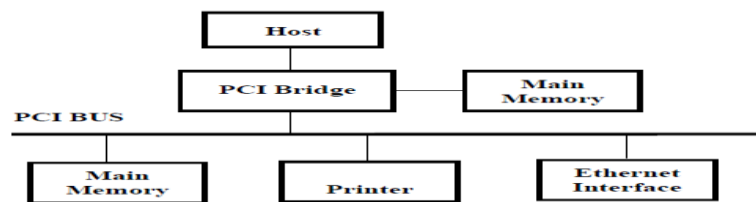


Figure Use of a PCI bus in a computer system.: At any given time, one device is the bus master. It has the right to initiate data transfers by issuing read and write commands. A master is called an initiator in PCI terminology. This is either a processor or a DMA controller. The addressed device that responds to read and write commands is called a target. The main bus signals used for transferring data are listed in Table 5.1. Signals whose name ends with the symbol # are asserted when in the low- voltage state. The main difference between the PCI protocol with others is that in addition to a Target-ready signal, PCI also uses an Initiatorready signal, IRDY #. The latter is needed to support burst transfers.

Table 5.1 Data transfer signals on the PCI bus.

Name	Function
CLK	A 33-MHz or 66-MHz clock.
FRAME#	Sent by the initiator to indicate the duration of a transaction.
AD	32 address/data lines, which may be optionally increased to 64.
C/BE#	4 command/byte-enable lines (8 for a 64-bit bus).
IRDY# , TRDY#	Initiator-ready and Target-ready signals.
DEVSEL#	A response from the device indicating that it has recognized its Address and is ready for a data transfer transaction.
IDSEL#	Initialization Device Select.

Consider a bus transaction in which the processor reads four 32-bit words from the memory. In this case, the initiator is the processor and the target is the memory. A complete transfer operation on the bus, involving an address and a burst of data, is called a transaction. Individual word transfers within a transaction are called phases. The sequence of events on the bus is shown in Figure 5.30. A clock signal provides the timing reference used to coordinate different phases of a transaction. All signal transitions are triggered by the rising edge of the clock. The signals changing later in the clock cycle to indicate the delays they encounter

2)SCSI (Small Computer Interface)System

SCSI (Small Computer System Interface): The acronym SCSI stands for Small Computer System Interface. It refers to a standard bus defined by the American National Standards Institute (ANSI) under the designation X3.131. In the original specifications of the standard, devices such as disks are connected to a computer via a 50-wire cable, which can be up to 25 meters in length and can transfer data at rates up to 5 megabytes/s. The SCSI bus standard has undergone many revisions, and its data transfer capability has increased very rapidly, almost doubling every two years. SCSI-2 and SCSI-3 have been defined, and each has several options. A SCSI bus may have eight data lines, in which case it is called a narrow bus and transfers data one byte at a time. Because of these various options, the SCSI connector may have 50, 68, or 80 pins. The maximum transfer rate in commercial devices that are currently available varies from 5 megabytes/s to 160 megabytes/s. The most recent version of the standard is intended to support transfer rates up to 320 megabytes/s, and 640 megabytes/s is anticipated a little later. Devices connected to the SCSI bus are not part of the address space of the processor in the same way as devices connected to the processor bus. The SCSI bus is connected to the processor bus through a SCSI controller, as shown in Figure 5.28. This controller uses DMA to transfer data packets from the main memory to the device, or vice versa. A packet may contain a block of data, commands from the processor to the device, or status information about the device. A controller connected to a SCSI bus is one of two types - an initiator or a target. An initiator has the ability to select a particular target and to send commands specifying the operations to be performed. The disk controller operates as a target. It carries out the commands it receives from the initiator. The initiator establishes a logical connection with the intended target. Once this connection has been established, it can be suspended and restored as needed to transfer commands and bursts of data. This ability to overlap data transfer requests is one of the key features of the SCSI bus that leads to its high performance.

Data transfers on the SCSI bus are always controlled by the target controller. To send a command to a target, an initiator requests control of the bus and, after winning arbitration, selects the controller it wants to communicate with and hands control of the bus over to it. Then the controller starts a data transfer operation to receive a command from the initiator. Assume that the processor wishes to read a block of data from a disk drive and that these data are stored in two disk sectors that are not contiguous. The processor sends a command to the SCSI controller, which causes the following sequence of events to take place:

1. The SCSI controller, acting as an initiator, contends for control of the bus.
2. When the initiator wins the arbitration process, it selects the target controller and hands over control of the bus to it.
3. The target starts an output operation (from initiator to target); in response to this, the initiator sends a command specifying the required read operation.
4. The target, realizing that it first needs to perform a disk seek operation, sends a message to the initiator indicating that it will temporarily suspend the connection between them. Then it releases the bus.
5. The target controller sends a command to the disk drive to move the read head to the first sector involved in the requested read operation. Then, it reads the data stored in that sector and stores them in a data buffer. When it is ready to begin transferring data to the initiator, the target requests control of the bus. After it wins arbitration, it reselects the initiator controller, thus restoring the suspended connection.
6. The target transfers the contents of the data buffer to the initiator and then suspends the connection again. Data are transferred either 8 or 16 bits in parallel, depending on the width of the bus.
7. The target controller sends a command to the disk drive to perform another seek operation. Then, it transfers the contents of the second disk sector to the initiator, as before. At the end of this transfer, the logical connection between the two controllers is terminated.
8. As the initiator controller receives the data, it stores them into the main memory using the DMA approach.
9. The SCSI controller sends an interrupt to the processor to inform it that the requested operation has been completed.

This scenario shows that the messages exchanged over the SCSI bus are at a higher level than those exchanged over the processor bus. The SCSI bus standard defines a wide range of control messages that can be exchanged between the controllers

to handle different types of I/O devices. Messages are also defined to deal with various error or failure conditions that might arise during device operation or data transfer.

Bus Signal: The bus signals are summarized in Table 5.2. For simplicity we show the signals for a narrow bus (8 data lines). Note that all signal names are preceded by a minus sign. This indicates that the signals are active, or that a data line is equal to 1, when they are in the low-voltage state. The bus has no address lines. Instead, the data lines are used to identify the bus controllers involved during the selection or reselection process and during bus arbitration. For a narrow bus, there are eight possible controllers, numbered 0 through 7, and each is associated with the data line that has the same number. A wide bus accommodates up to 16 controllers. A controller places its own address or the address of another controller on the bus by activating the corresponding data line. Thus, it is possible to have more than one address on the bus at the same time, as in the arbitration process we describe next. Once a connection is established between two controllers, there is no further need for addressing, and the data lines are used to carry data.

Table 5.2 The SCSI bus signal

Category	Name	Function
Data	-DB(O) to -DB(7)	Data lines: Carry one byte of information during the information transfer phase and identify device during arbitration, selection and reselection phases
	-DB(P)	Parity bit for the data bus
Phase	-BSY	Busy: Asserted when the bus is not free
	-SEL	Selection: Asserted during selection and reselection
Information type	-C/D	Control/Data: Asserted during transfer of control information (command, status or message)
	-MSG	Message: indicates that the information being transferred is a message
Handshake	-REQ	Request: Asserted by a target to request a data transfer cycle
	-ACK	Acknowledge: Asserted by the initiator when it has completed a data transfer operation
Direction of transfer	-I/O	Input/Output. Asserted to indicate an input operation
Other	-ATN	Attention: Asserted by an initiator when it wishes to send a message to a target
	-RST	Reset: Causes all Device controls to disconnect from the bus and their start-up state

The main phases involved in the operation of the SCSI bus are arbitration, selection, information transfer, and reselection.

Arbitration: The bus is free when the -BSY signal is in the inactive (high-voltage) state. Any controller can request the use of the bus while it is in this state. Since two or more controllers may generate such a request at the same time, an arbitration scheme must be implemented. A controller requests the bus by asserting the -BSY signal and by asserting its associated data line to identify itself. The

SCSI bus uses a simple distributed arbitration scheme. It is illustrated by the example in Figure 5.32, in which controllers 2 and 6 request the use of the bus simultaneously. Each controller on the bus is assigned a fixed priority, with controller 7 having the highest priority. When $\overline{\text{BSY}}$ becomes active, all controllers that are requesting the bus examine the data lines and determine whether a higher-priority device is requesting the bus at the same time. The controller using the highest-numbered line realizes that it has won the arbitration process. All other controllers disconnect from the bus and wait for $\overline{\text{BSY}}$ to become inactive again.

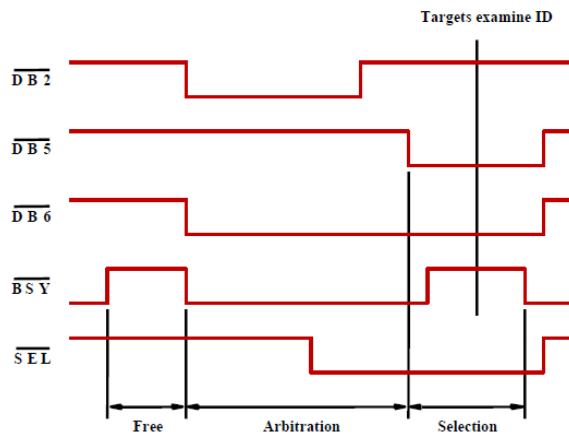


Figure 5.32 Arbitration and selection on the SCSI bus. Device6 wins arbitration and selects device2.

In Figure 5.32, we have assumed that controller 6 is an initiator that wishes to establish a connection to controller 5. After winning arbitration, controller 6 proceeds to the selection phase, in which it identifies the target.

Selection: Having won arbitration, controller 6 continues to assert $\overline{\text{BSY}}$ and $\overline{\text{DB6}}$ (its address). It indicates that it wishes to select controller 5 by asserting the $\overline{\text{SEL}}$ and then the $\overline{\text{DB5}}$ lines. Any other controller that may have been involved in the arbitration phase, such as controller 2 in the figure, must stop driving the data lines once the $\overline{\text{SEL}}$ line becomes active, if it has not already done so. After placing the address of the target controller on the bus, the initiator releases the $\overline{\text{BSY}}$ line. The selected target controller responds by asserting $\overline{\text{BSY}}$. This informs the initiator that the connection it is requesting has been established, so that it may remove the address information from the data lines. The selection process is now complete, and the target controller (controller 5) is asserting $\overline{\text{BSY}}$. From this point on, controller 5 has control of the bus, as required for the information transfer phase.

Information Transfer: The information transferred between two controllers may consist of commands from the initiator to the target, status responses from the target to the initiator, or data being transferred to or from the I/O device.

Handshake signaling is used to control information transfers. The $\overline{\text{REQ}}$ and $\overline{\text{ACK}}$ signals replace the Master-ready and Slave-ready signals. The target asserts $\overline{\text{I/O}}$ during an input operation (target to initiator), and it asserts $\overline{\text{CID}}$ to indicate that the information being transferred is a command or a status response rather than data.

High-speed versions of the SCSI bus use a technique known as double-edge clocking or Double Transitions (DT). Each data transfer requires a high-to-low transition followed by a low-to-high transition on the two handshake signals. Double-edge clocking means that data are transferred on both the rising and falling edges of these signals, thus doubling the transfer rate. At the end of the transfer, the target controller releases the $\overline{\text{BSY}}$ signal, thus freeing the bus for use by other devices.

Reselection: When a logical connection is suspended and the target is ready to restore it, the target must first gain control of the bus. It starts an arbitration cycle, and after winning arbitration, it selects the initiator controller in exactly the same manner as described above. But with the roles of the target and initiator reversed, the initiator is now asserting $\overline{\text{BSY}}$. Before data transfer can begin, the initiator must hand control over to the target. This is achieved by having the target controller assert $\overline{\text{BSY}}$ and selecting the initiator. Meanwhile, the initiator waits for a short period after being selected to make sure that the target has asserted $\overline{\text{BS}}$ and then it releases the $\overline{\text{BSY}}$ line. The connection between the two controllers has now been reestablished, with the target in control of the bus as required for data transfer to proceed.

The bus signaling scheme described above provides the mechanisms needed for two controllers to establish a logical connection and exchange messages. The connection may be suspended and reestablished at any time. The SCSI standard defines the structure and contents of various types of packets that the controllers exchange to handle different situations. The initiator uses these packets to send the commands it receives from the processor to the target. The target responds with status information and data transfer operations. The latter are controlled by the target, because it is the target that knows when data are available, when to suspend and reestablish connections, etc.

3)USB (Universal Serial Bus)

Universal Serial Bus (USB) is an industry standard developed through a collaborative effort of several computer and communications companies, including Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, Nortel Networks,

and Philips. USB is a simple and low cost mechanism to connect the devices such as keyboards, mouse, cameras, speakers, printer and display devices to the computer.

The USB supports two speeds of operation, called low-speed (1.5 megabits/s) and fullspeed (12 megabits/s). The most recent revision of the bus specification (USB 2.0) introduced a third speed of operation, called high-speed (480 megabits/s). The USB is quickly gaining acceptance in the market place, and with the addition of the high-speed capability it may well become the interconnection method of choice for most computer devices. The USB has been designed to meet several key objectives:

- Provide a simple, low-cost, and easy to use interconnection system that overcomes the difficulties due to the limited number of I/O ports available on a computer
- Accommodate a wide range of data transfer characteristics for I/O devices, including telephone and Internet connections
- Enhance user convenience through a "plug-and-play" mode of operation

Port limitation: Only a few ports are provided in a typical computer. To add new ports, a user must open the computer box to gain access to the internal expansion bus and install a new interface card. The user may also need to know how to configure the device and the software. An objective of the USB is to make it possible to add many devices to a computer system at any time, without opening the computer box.

Device Characteristics: The different kinds of devices may be connected to a computer cover a wide range of functionality. The speed, volume, and timing constraints associated with data transfers to and from such devices vary significantly. In the case of a keyboard, one byte of data is generated every time a key is pressed, which may happen at any time. These data should be transferred to the computer promptly. Since the event of pressing a key is not synchronized to any other event in a computer system, the data generated by the keyboard are called asynchronous. Furthermore, the rate at which the data are generated is quite low. It is limited by the speed of the human operator to about 100 bytes per second, which is less than 1000 bits per second.

Let us consider a different source of data. Many computers have a microphone either externally attached or built in. The sound picked up by the microphone produces an analog electrical signal, which must be converted into a digital form

before it can be handled by the computer. This is accomplished by sampling the analog signal periodically. For each sample, an analog-to-digital (A/D) converter generates an n-bit number representing the magnitude of the sample. The number of bits, n, is selected based on, the desired precision with which to represent each sample. Later, when these data are sent to a speaker, a digital-to-analog (D/A) converter is used to restore the original analog signal from the digital format. The sampling process yields a continuous stream of digitized samples that arrive at regular intervals, synchronized with the sampling clock. Such a data stream is called isochronous, meaning that successive events are separated by equal periods of time.

Plug-and-play: The plug-and-play feature means that a new device, such as an additional speaker, can be connected at any time while the system is operating. The system should detect the existence of this new device automatically, identify the appropriate device-driver software and any other facilities needed to service that device, and establish the appropriate addresses and logical connections to enable them to communicate. The plug-and-play requirement has many implications at all levels in the system, from the hardware to the operating system and the applications software. One of the primary objectives of the design of the USB has been to provide a plug-and-play capability.

USB Architecture: A serial transmission format has been chosen for the USB because a serial bus satisfies the low-cost and flexibility requirements. Clock and data information are encoded together and transmitted as a single signal. Hence, there are no limitations on clock frequency or distance arising from data skew. Therefore, it is possible to provide a high data transfer bandwidth by using a high clock frequency. As pointed out earlier, the USB offers three bit rates, ranging from 1.5 to 480 megabits/s, to suit the needs of different I/O devices.

USB has the tree structure shown in Figure 5.33. Each node of the tree has a device called a hub, which acts as an intermediate control point between the host and the I/O devices. At the root of the tree, a root hub connects the entire tree to the host computer. The leaves of the tree are the I/O devices being served (for example, keyboard, speaker, or digital TV), which are called functions in USB terminology.

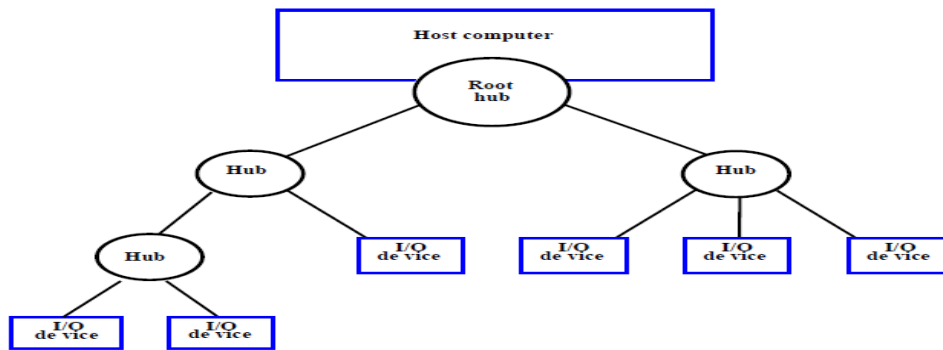


Figure 5.33 Universal Serial Bus tree structure.

The tree structure enables many devices to be connected while using only simple point-to-point serial links. Each hub has a number of ports where devices may be connected, including other hubs. In normal operation, a hub copies a message that it receives from its upstream connection to all its downstream ports. As a result, a message sent by the host computer is broadcast to all I/O devices, but only the addressed device will respond to that message. A message from an I/O device is sent only upstream towards the root of the tree and is not seen by other devices. Hence, the USB enables the host to communicate with the I/O devices, but it does not enable these devices to communicate with each other.

The USB operates strictly on the basis of polling. A device may send a message only in response to a poll message from the host. Hence, upstream messages do not encounter conflicts or interfere with each other, as no two devices can send messages at the same time. This restriction allows hubs to be simple, Low-cost devices.

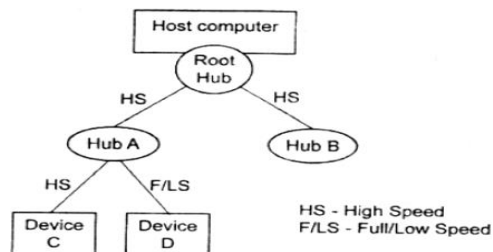


Fig Universal serial bus tree structure

The mode of operation described above is observed for all devices operating at either low speed or full speed. However, one exception has been necessitated by the introduction of high-speed operation in USB version 2.0. Consider the situation in Figure 5.34. Hub A is connected to the root hub by a high-speed link. This hub serves one high-speed device, C, and one low-speed device, D. Normally, a message to device D would be sent at low speed from the root hub. At 1.5

megabits/s, even a short message takes several tens of microseconds. For the duration of this message, no other data transfers can take place, thus reducing the effectiveness of the high-speed links and introducing unacceptable delays for high-speed devices. To mitigate this problem, the USB protocol requires that a message transmitted on a high-speed link is always transmitted at high speed, even when the ultimate receiver is a low-speed device. Hence, a message intended for device D is sent at high speed from the root hub to hub A, then forwarded at low speed to device D. The message to device D is preceded and followed by special commands to hub A to start

The USB standard specifies the hardware details of USB interconnections as well as the organization and requirements of the host software. The purpose of the USB software is to provide bidirectional communication links between application software and I/O devices. These links are called pipes. Any data entering at one end of a pipe is delivered at the other end. Issues such as addressing, timing, or error detection and recovery are handled by the USB protocols. The software that transfers data to or from a given I/O device is called the device driver for that device. The device drivers depend on the characteristics of the devices they support. Hence, a more precise description of the USB pipe is that it connects an I/O device to its device driver. It is established when a device is connected and assigned a unique address by the USB software. Once established, data may flow through the pipe at any time.

Addressing: I/O devices are normally identified by assigning them a unique memory address. In fact, a device usually has several addressable locations to enable the software to send and receive control and status information and to transfer data. When a USB is connected to a host computer, its root hub is attached to the processor bus, where it appears as a single device. The host software communicates with individual devices attached to the USB by sending packets of information, which the root hub forwards to the appropriate device in the USB tree. Each device on the USB, whether it is a hub or an I/O device, is assigned a 7-bit address. This address is local to the USB tree and is not related in any way to the addresses used on the processor bus. A hub may have any number of devices or other hubs connected to it, and addresses are assigned arbitrarily. When a device is first connected to a hub, or when it is powered on, it has the address 0. The hardware of the hub to which this device is connected is capable of detecting that the device has been connected, and it records this fact as part of its own status information. Periodically, the host polls each hub to collect status information and learn about new devices that may have been added or disconnected.

USB Protocol: All information transferred over the USB is organized in packets, where a packet consists of one or more bytes of information. There are many types of packets that perform a variety of control functions. The information transferred on the USB can be divided into two broad categories: control and data. Control packets perform such tasks as addressing a device to initiate data transfer, acknowledging that data have been received correctly, or indicating an error. Data packets carry information that is delivered to a device. For example, put and output data are transferred inside data packets.

A packet consists of one or more fields containing different kinds of information. The first field of any packet is called the packet identifier, PID, which identifies the type of that packet. There are four bits of information in this field, but they are transmitted twice. The first time they are sent with their true values, and the second time with each bit complemented, as shown in Figure 5.35(a). This enables the receiving device to verify that the PID byte has been received correctly.

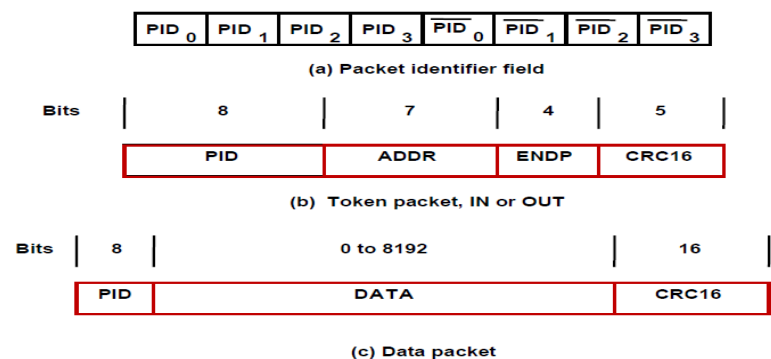


Fig **USB** **packet** **format**

The four PID bits identify one of 16 different packet types. Some control packets, such as ACK (Acknowledge), consist only of the PID byte. Control packets used for controlling data transfer operations are called token packets. A token packet starts with the PID field, using one of two PID values to distinguish between an IN packet and an OUT packet, which control input and output transfers, respectively. The PID field is followed by the 7 -bit address of a device and the 4-bit endpoint number within that device. The packet ends with 5 bits for error checking, using a method called cyclic redundancy check (CRC). The CRC bits are computed based on the contents of the address and endpoint fields. By performing an inverse computation, the receiving device can determine whether the packet has been received correctly.

Data packets, which carry input and output data.The packet identifier field is followed by up to 8192 bits of data, then 16 error-checking bits. Three different

PID patterns are used to identify data packets, so that data packets may be numbered 0, 1, or 2. Note that data packets do not carry a device address or an endpoint number. This information is included in the IN or OUT token packet that initiates the transfer. Consider an output device connected to a USB hub, which in turn is connected to a host computer. An example of an output operation is shown in Figure 5.36. The host computer sends a token packet of type OUT to the hub, followed by a data packet containing the output data. The PID field of the data packet identifies it as data packet number 0. The hub verifies that the transmission has been error free by checking the error control bits, and then sends an acknowledgment packet (ACK) back to the host. The hub forwards the token and data packets downstream. All I/O devices receive this sequence of packets, but only the device that recognizes its address in the token packet accepts the data in the packet that follows. After verifying that transmission has been error free, it sends an ACK packet to the hub.

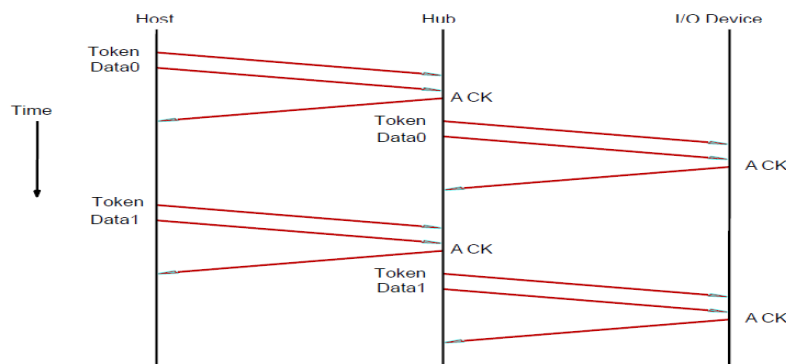


Figure 5.36 An output transfer

Successive data packets on a full-speed or low-speed pipe carry the numbers 0 and 1, alternately. This simplifies recovery from transmission errors. If a token, data, or acknowledgment packet is lost as a result of a transmission error, the sender resends the entire sequence. By checking the data packet number in the PID field, the receiver can detect and discard duplicate packets. High-speed data packets are sequentially numbered 0, 1, 2, 0, and so on. Input operations follow a similar procedure. The host sends a token packet of type IN containing the device address. In effect, this packet is a poll asking the device to send any input data it may have. The device responds by sending a data packet followed by an ACK. If it has no data ready, it responds by sending a negative acknowledgment (NAK) instead.

Electrical characteristics: The cables used for USB connections consist of four wires. Two are used to carry power, 5 V and Ground. Thus, a hub or an I/O device may be powered directly from the bus, or it may have its own external power connection. The other two wires are used to carry data. Different signaling

schemes are used for different speeds of transmission. At low speed, 1s and 0s are transmitted by sending a high voltage state (5 V) on one or the other of the two signal wires. For high-speed links, differential transmission is used.

➤ Direct Memory Access(DMA)

The main idea of direct memory access (DMA) is to enable peripheral devices to cut out the “middle man” role of the CPU in data transfer. It allows peripheral devices to transfer data directly from and to memory without the intervention of the CPU. Having peripheral devices access memory directly would allow the CPU to do other work, which would lead to improved performance, especially in the cases of large transfers. The DMA controller is a piece of hardware that controls one or more peripheral devices. It allows devices to transfer data to or from the system’s memory without the help of the processor. In a typical DMA transfer, some event notifies the DMA controller that data needs to be transferred to or from memory. Both the DMA and CPU use memory bus and only one or the other can use the memory at the same time. The DMA controller then sends a request to the CPU asking its permission to use the bus. The CPU returns an acknowledgment to the DMA controller granting it bus access. The DMA can now take control of the bus to independently conduct memory transfer. When the transfer is complete the DMA relinquishes its control of the bus to the CPU. Processors that support DMA provide one or more input signals that the bus requester can assert to gain control of the bus and one or more output signals that the CPU asserts to indicate it has relinquished the bus. Figure 8.10 shows how the

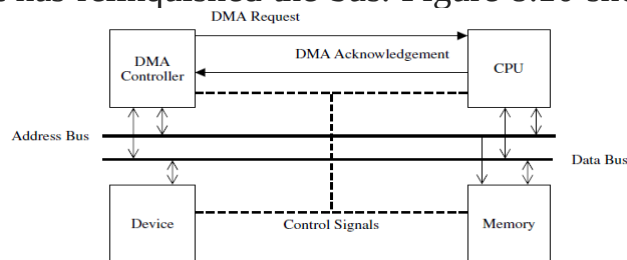


Figure 8.10 DMA controller shares the CPU's memory bus

DMA controller shares the CPU's memory bus.

he CPU's

Direct memory access controllers require initialization by the CPU. Typical setup parameters include the address of the source area, the address of the destination area, the length of the block, and whether the DMA controller should generate a processor interrupt once the block transfer is complete. A DMA controller has an address register, a word count register, and a control register. The address register contains an address that specifies the memory location of the data to be transferred. It is typically possible to have the DMA controller automatically increment the

address register after each word transfer, so that the next transfer will be from the next memory location. The word count register holds the number of words to be transferred. The word count is decremented by one after each word transfer. The control register specifies the transfer mode.

Direct memory access data transfer can be performed in burst mode or singlecycle mode. In burst mode, the DMA controller keeps control of the bus until all the data has been transferred to (from) memory from (to) the peripheral device. This mode of transfer is needed for fast devices where data transfer cannot be stopped until the entire transfer is done. In single-cycle mode (cycle stealing), the DMA controller relinquishes the bus after each transfer of one data word. This minimizes the amount of time that the DMA controller keeps the CPU from controlling the bus, but it requires that the bus request/acknowledge sequence be performed for every single transfer. This overhead can result in a degradation of the performance. The single-cycle mode is preferred if the system cannot tolerate more than a few cycles of added interrupt latency or if the peripheral devices can buffer very large amounts of data, causing the DMA controller to tie up the bus for an excessive amount of time.

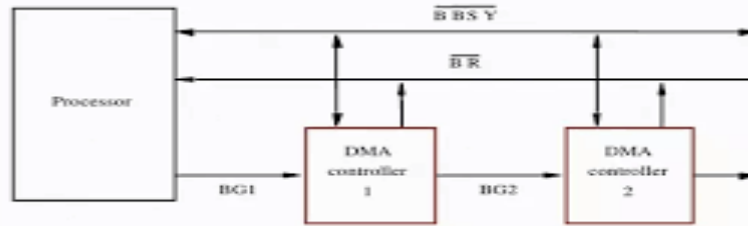
The following steps summarize the DMA operations:

1. DMA controller initiates data transfer.
2. Data is moved (increasing the address in memory, and reducing the count of words to be moved).
3. When word count reaches zero, the DMA informs the CPU of the termination by means of an interrupt.
4. The CPU regains access to the memory bus.

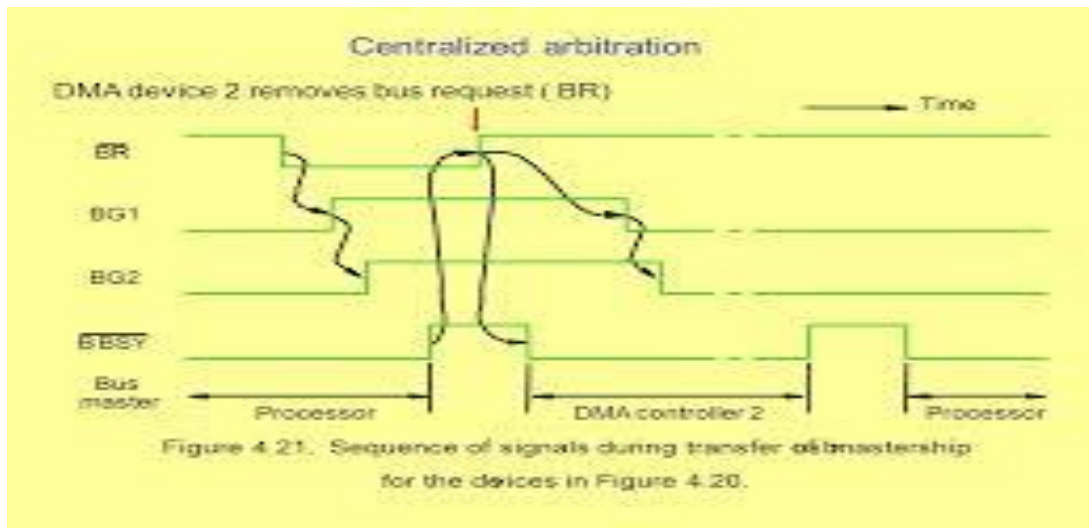
A DMA controller may have multiple channels. Each channel has associated with it an address register and a count register. To initiate a data transfer the device driver sets up the DMA channel's address and count registers together with the direction of the data transfer, read or write. While the transfer is taking place, the CPU is free to do other things. When the transfer is complete, the CPU is interrupted. Direct memory access channels cannot be shared between device drivers. A device driver must be able to determine which DMA channel to use. Some devices have a fixed DMA channel, while others are more flexible, where the device driver can simply pick a free DMA channel to use.

1)BUS ARBITRATION

Centralized Bus Arbitration



Centralized arbitration scheme with one Bus-Request (BR) line and one Bus-Grant (BG) line forming a daisy chain. Several pairs of BR and BG lines are possible, perhaps one per device as in the case of interrupts. Bus arbiter has to ensure that only one request is granted at any given time. It may do so according to a fixed priority scheme, or a rotating priority scheme. Rotating priority scheme: There are four devices, and initial priority is 1,2,3,4. After the request from device 1 is granted, the priority changes to 2,3,4,1.



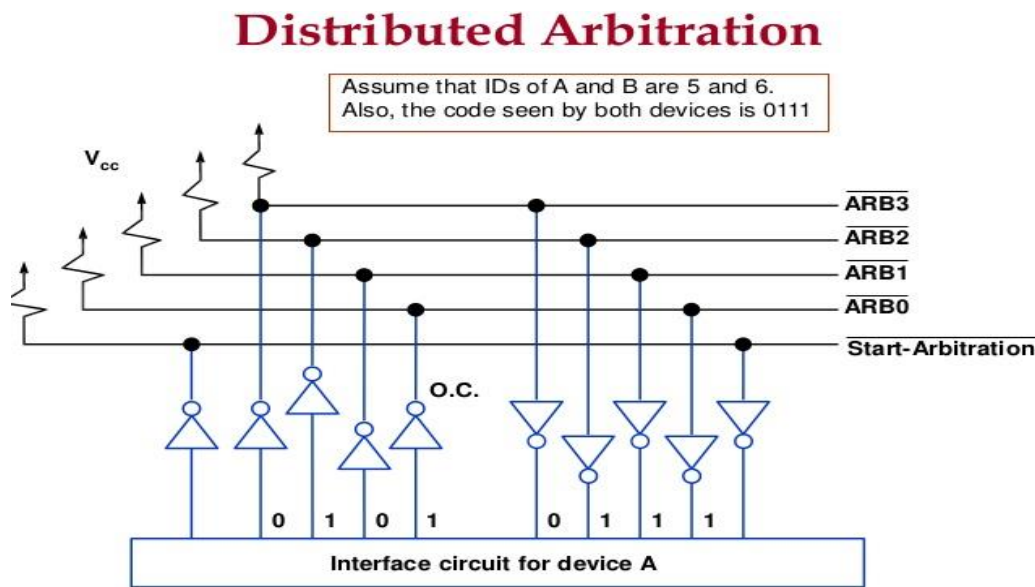
Distributed

arbitration

All devices waiting to use the bus share the responsibility of carrying out the arbitration process. Arbitration process does not depend on a central arbiter and hence distributed arbitration has higher reliability. Each device is assigned a 4-bit ID number. All the devices are connected using 5 lines, 4 arbitration lines to transmit the ID, and one line for the Start-Arbitration signal. To request the bus a device: Asserts the Start-Arbitration signal. Places its 4-bit ID number on the arbitration lines. The pattern that appears on the arbitration lines is the logical-OR of all the 4-bit device IDs placed on the arbitration lines.

Device A has the ID 5 and wants to request the bus:- Transmits the pattern 0101 on the arbitration lines. Device B has the ID 6 and wants to request the bus:- Transmits

the pattern 0110 on the arbitration lines. Pattern that appears on the arbitration lines is the logical OR of the patterns:- Pattern 0111 appears on the arbitration lines. Arbitration process: Each device compares the pattern that appears on the arbitration lines to its own ID, starting with MSB. If it detects a difference, it transmits 0s on the arbitration lines for that and all lower bit positions. Device A compares its ID 5 with a pattern 0101 to pattern 0111. It detects a difference at bit position 0, as a result, it transmits a pattern 0100 on the arbitration lines. The pattern that appears on the arbitration lines is the logical-OR of 0100 and 0110, which is 0110. This pattern is the same as the device ID of B, and hence B has won the arbitration



UNIT-4

➤ Memory System Concepts

Introduction: Computer should have a large memory to facilitate execution of programs that are large and deal with huge amounts of data. The memory should be fast, large, and inexpensive. Unfortunately, it is impossible to meet all three of these requirements simultaneously. Increased speed and size are achieved at increased cost. To solve this problem, much work has gone into developing clever structures that improve the apparent speed and size of the memory, yet keep the cost reasonable.

The maximum size of the memory that can be used in any computer is determined by the addressing scheme. For example, a 16-bit computer that generates 16-bit addresses is capable of addressing up to $2^{16} = 64K$ (65536) memory locations. Similarly, machines whose instructions generate 32-bit addresses can utilize a memory that contains up to $2^{32} = 4G$ (giga) memory locations, whereas machines with 40-bit addresses can access up to $2^{40} = 1T$ (tera) locations. The number of locations represents the size of the address space of the computer.

From the system standpoint, we can view the memory unit as a black box. Data transfer between the memory and the processor takes place through the use of two processor registers, usually called MAR (memory address register) and MDR (memory data register). If MAR is k bits long and MDR is n bits long, then the memory unit may contain up to 2^k addressable locations. During a memory cycle, n bits of data are transferred between the memory and the processor. This transfer takes place over the processor bus, which has k address lines and n data lines. The bus also includes the control lines Read / Write (R / $\bar{W}$) and Memory Function Completed (MFC) for coordinating data transfers.

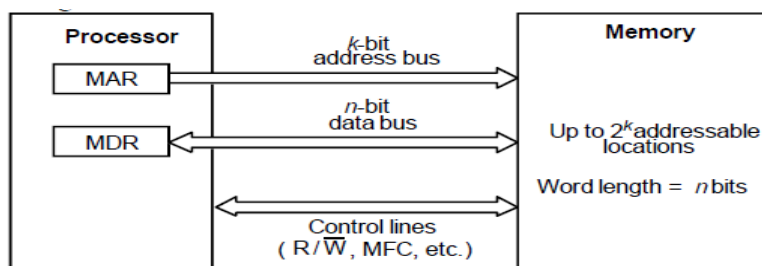


Figure Connection of the memory to the processor

- The processor reads data from the memory by loading the address of the required memory location into the MAR register and setting the R / W line to 1. The memory responds by placing the data from the addressed location onto the data lines, and confirms this action by asserting the MFC signal. Upon receipt of the MFC signal, the processor loads the data on the data lines into the MDR register. The processor writes data into a memory location by loading the address of this location into MAR and loading the data into MDR. It indicates that a write operation is involved by setting the R/ W line to 0. If read or write operations involve consecutive address locations in the main memory, then a "block transfer" operation can be performed in which the only address sent to the memory is the one that identifies the first location.
- The time between the Read and the MFC signals is referred to as the memory access time. The memory cycle time is the minimum time delay required between the initiations of two successive memory operations, If any location can be accessed for a Read or Write operation some fixed amount of time that is independent of the location's address in a memory unit is called random-access memory (RAM). One way to reduce the memory access time is to use a cache memory. This is a small, fast memory that is inserted between the larger, slower main memory and the processor. It holds the currently active segments of a program and their data.
- Virtual memory is used to increase the apparent size of the physical memory. Data are addressed in a virtual address space that can be as large as the addressing capability of the processor. But at any given time, only the active portion of this space is mapped onto locations in the physical memory. The remaining virtual addresses are mapped onto the bulk: storage devices used, which are usually magnetic disks. The virtual address space is mapped onto the physical memory where data are actually stored. The mapping function is implemented by a special memory control circuit, often called the memory management unit.

➤ **Semiconductor RAM Memories**

Discuss Semiconductor RAM Memories: Semiconductor memories are available in a wide range of speeds. Their cycle times range from 100ns to less than 10 ns.

- 1) **INTERNAL ORGANIZATION OF MEMORY CHIPS:** Memory cells are usually organized in the form of an array, in which each cell is capable of storing one bit of information.

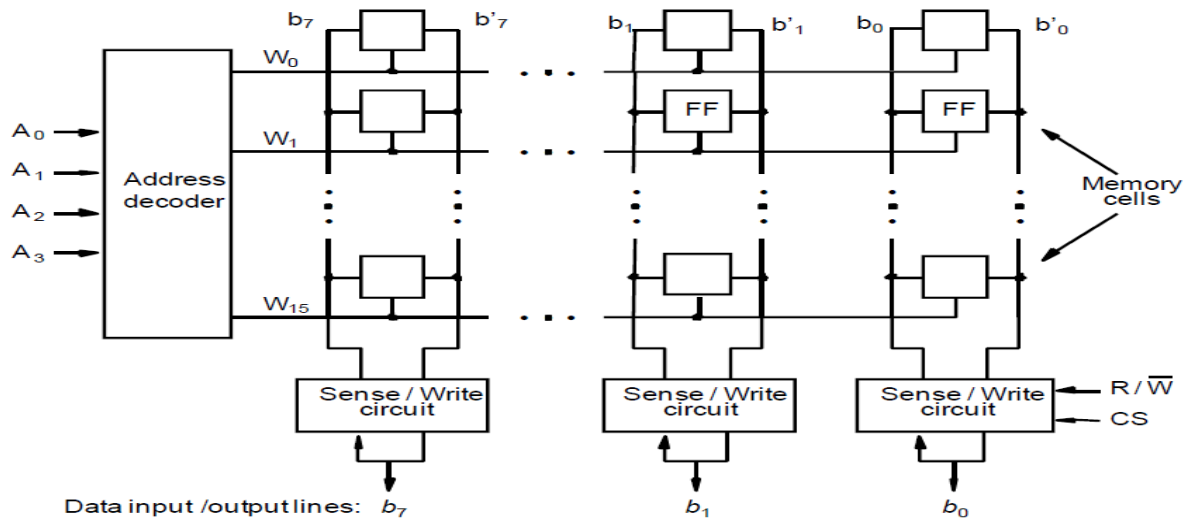
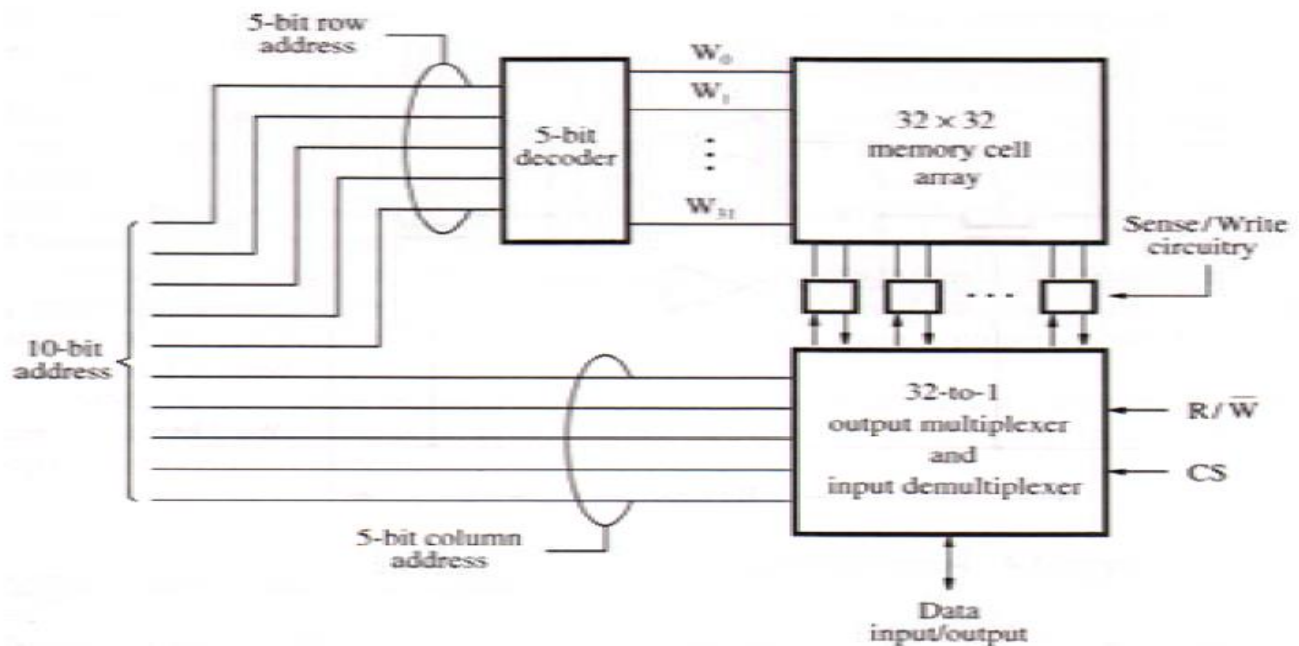


Figure Organization of bit cell in a memory chip

Each row of cells constitutes a memory word, and all cells of a row are connected to a common line referred to as the word line, which is driven by the address decoder on the chip. The cells in each column are connected to a Sense/Write circuit by two bit lines. The Sense/Write circuits are connected to the data input/output lines of the chip. During a Read operation, these circuits sense, or read, the information stored in the cells selected by a word line and transmit this information to the output data lines. During a Write operation, the Sense/Write circuits receive input information and store it in the cells of the selected word.

Figure 5.2 is an example of a very small memory chip consisting of 16 words of 8 bits each. This is referred to as a 16 x 8 organization. The data input and the data output of each Sense/Write circuit are connected to a single bidirectional data line that, can be connected to the data bus of a computer. Two control lines, $R/\bar{W}$ and CS , are provided in addition to address and data lines. The $R/\bar{W}$ (Read/Write) input specifies the required operation, and the CS (Chip Select) input selects a given chip in a multichip memory system.



- The memory circuit in Figure 4.2 stores 128 bits and requires 14 external connections for address, data, and control lines. Of course, it also needs two lines for power supply and ground connections. Consider now a slightly larger memory circuit, one that has 1 K (1024) memory cells.
- This circuit can be organized as a 128 x 8 memory, requiring a total of 19 external connections. Alternatively, the same number of cells can be organized into a 1 K x 1 format. In this case, a 100bit address is needed, but there is only one data line, resulting in 15 external connections. Figure 4.3 shows such an organization.
- The required 100bit address is divided into two groups of 5 bits each to form the row and column addresses for the cell array. A row address selects a row of 32 cells, all of which are accessed in parallel. However, according to the column address, only one of these cells is connected to the external data line by the output multiplexer and input demultiplexer. For an example, a 4M-bit chip may have a 512K x 8 organization, in which case 19 address and 8 data input/output pins are needed.

2) Static Memories

Static memories are the Memories that consist of circuits capable of retaining their state as long as power is applied are known as static memories. Figure 4.4 illustrates how a static RAM (SRAM) cell may be implemented. Two inverters are cross-connected to form a latch. The latch is connected to two bit lines by transistors T1 and T2. These transistors act as

switches that can be opened or closed under control of the word line. When the word line is at ground level, the transistors are turned off and the latch retains its state. For example, let us assume that the cell is in state 1 if the logic value at point X is 1 and at point Y is 0. This state is maintained as long as the signal on the word line is at ground level.

Read operation: In order to read the state of the SRAM cell, the word line is activated to close switches T1 and T2. If the cell is in state 1, the signal on bit line b is high and the signal on bit line b' is low. The opposite is true if the cell is in state 0. Thus, b and b' are complements of each other. Sense/Write circuits at the end of the bit lines monitor the state of b and b' and set the output accordingly.

Write operation: The state of the cell is set by placing the appropriate value on bit line b and its complement on b', and then activating the word line. This forces the cell into the corresponding state. The required signals on the bit lines are generated by the Sense/Write circuit. If T1 and T2 are turned on (closed), it lines b and b' will have high and low signals, respectively.

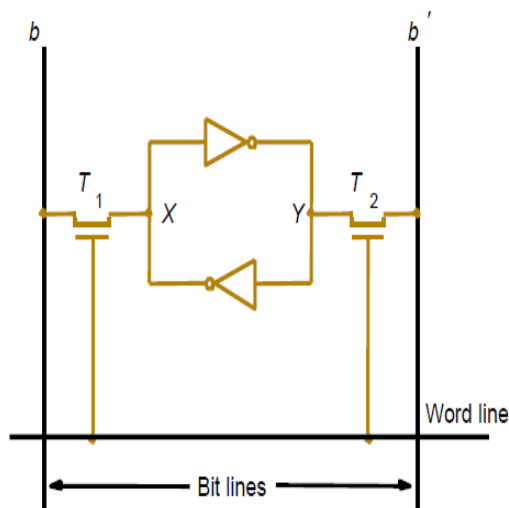
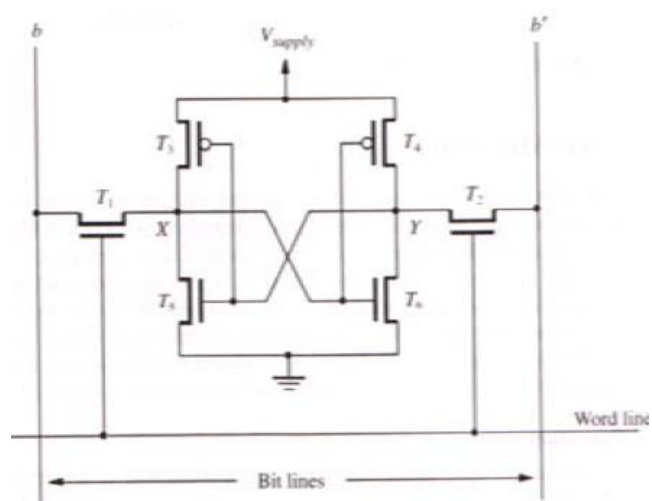


Figure A static RAM cell



An example of a CMOS memory cell

The power supply voltage, V_{supply} , is 5 V in older CMOS SRAMs or 3.3 V in new lowvoltage versions. Note that "continuous power is needed for the cell to retain its state. If power is interrupted, the cell's contents will be lost. When power is restored, the latch will settle into a stable state, but it will not necessarily be the same state the cell was in before the interruption. Hence, SRAMs are said to be volatile memories because their contents are lost when power is interrupted.

A major advantage of CMOS SRAMs is their very low power consumption because current flows in the cell only when the cell is being accessed. Otherwise, T₁, T₂ and one transistor in each inverter are turned off, ensuring that there is no active path between V_{supply} and ground. Static RAMs can be accessed very quickly. Access times of just a few nanoseconds are found in commercially available chips. SRAMs are used in applications where speed is of critical concern.

3) Asynchronous DRAMs

Static RAMs are fast, but they come at a high cost because their cells require several transistors. Less expensive RAMs can be implemented if simpler cells are used. However, such cells do not retain their state indefinitely; hence, they are called dynamic RAMs (DRAMs). Information is stored in a dynamic memory cell in the form of a charge on a capacitor, and this charge can be maintained for only tens of milliseconds. Since the cell is required to store information for a much longer time, its contents must be periodically refreshed by restoring the capacitor charge to its full value. An example of a dynamic memory cell that consists of a capacitor, C, and a transistor, T, is shown in Figure 4.6. In order to store information in this cell, transistor T is turned on and an appropriate voltage is applied to the bit line. This causes a known amount of charge to be stored in the capacitor.

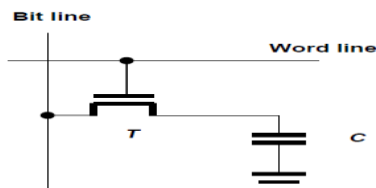


Figure 4.6 A single transistor dynamic memory cell

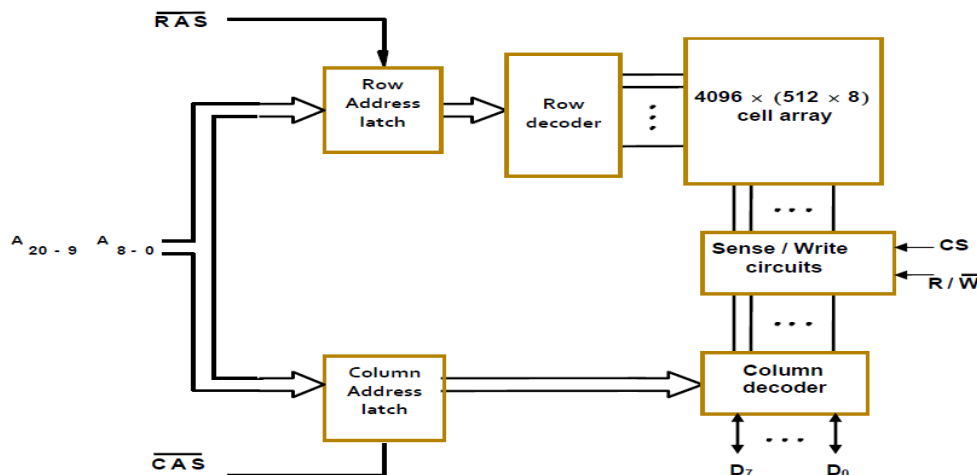


Figure 4.7 Internal Organization of a 2M x 8 dynamic memory chip

A 16-megabit DRAM chip, configured as $2\text{M} \times 8$, is shown in Figure 4.7. The cells are organized in the form of a $4\text{K} \times 4\text{K}$ array. The 4096 cells in each row are divided into 512 groups of 8, so that a row can store 512 bytes of data. Therefore, 12 address bits are needed to select a row. Another 9 bits are needed to specify a group of 8 bits in the selected row. Thus, a 21-bit address is needed to access a byte in this memory. The high-order 12 bits and the low-order 9 bits of the address constitute the row and column addresses of a byte, respectively. To reduce the number of pins needed for external connections, the row and column addresses are multiplexed on 12 pins. During a Read or a Write operation, the row address is applied first. It is loaded into the row address latch in response to a signal pulse on the Row Address Strobe (RAS) input of the chip.

Then a Read operation is initiated, in which all cells on the selected row are read and refreshed. Shortly after the row address is loaded, the column address is applied to the address pins and loaded into the column address latch under control of the Column Address Strobe (CAS) signal. The information in this latch is decoded and the appropriate group of 8 Sense/Write circuits are selected. If the R/W control signal indicates a Read operation, the output values of the selected circuits are transferred to the data lines, D7-0. For a Write operation, the information on the D7-0 lines is transferred to the selected circuits. This information is then used to overwrite the contents of the selected cells in the corresponding 8 columns. We should note that in commercial DRAM chips, the RAS and CAS control signals are active low so that they cause the latching of addresses when they change from high to low. To indicate this fact, these signals are shown on diagrams as $\overline{\text{RAS}}$ and $\overline{\text{CAS}}$.

Applying a row address causes all cells on the corresponding row to be read and refreshed during both Read and Write operations. To ensure that the contents of a DRAM are maintained, each row of cells must be accessed periodically. A refresh circuit usually performs this function automatically. Many dynamic memory chips incorporate a refresh facility within the chips themselves. In this case, the dynamic nature of these memory chips is almost invisible to the user. In the DRAM described in this section, the timing of the memory device is controlled asynchronously. A specialized memory controller circuit provides the necessary control signals, $\overline{\text{RAS}}$ and $\overline{\text{CAS}}$, that govern the timing. The processor must take into account the delay in the response of the memory. Such memories are referred to as asynchronous DRAMs.

Because of their high density and low cost, DRAMs are widely used in the memory units of computers. Available chips range in size from 1M to 256M bits,

and even larger chips are being developed. To reduce the number of memory chips needed in a given computer, a DRAM chip is organized to read or write a number of bits in parallel, as indicated in Figure 4.7. To provide flexibility in designing memory systems, these chips are manufactured in different organizations. For example, a 64-Mbit chip may be organized as 16M x 4, 8M x 8, or 4M x 16.

4) Synchronous DRAMS

DRAMs whose operation is directly synchronized with a clock signal are known as synchronous DRAMs (SDRAMs). Figure 4.8 indicates the structure of an SDRAM. The cell array is the same as in asynchronous DRAMs. The address and data connections are buffered by means of registers. We should particularly note that the output of each sense amplifier is connected to a latch. A Read operation causes the contents of all cells in the selected row to be loaded into these latches. But, if an access is made for refreshing purposes only, it will not change the contents of these latches; it will merely refresh the contents of the cells. Data held in the latches that correspond to the selected column(s) are transferred into the data output register, thus becoming available on the data output pins. SDRAMs have several different modes of operation, which can be selected by writing control information into a mode register. In SDRAMs, it is not necessary to provide externally generated pulses on the CAS line to select successive columns.

Figure 4.9 shows a timing diagram for a typical burst read of length 4. First, the row address is latched under control of the RAS signal. The memory typically takes 2 or 3 clock cycles (we use 2 in the figure) to activate the selected row. Then, the column address is latched under control of the CAS signal. After a delay of one clock cycle, the first set of data bits is placed on the data lines. The SDRAM automatically increments the column address to access the next three sets of bits in the selected row, which are placed on the data lines in the next 3 clock cycles.

SDRAMs have built-in refresh circuitry. A part of this circuitry is a refresh counter, which provides the addresses of the rows that are selected for refreshing. In a typical SDRAM, each row must be refreshed at least every 64 ms. Commercial SDRAMs can be used with clock speeds above 100 MHz. These chips are designed to meet the requirements of commercially available processors that are used in large volume. For example, Intel has defined PCI00 and PC133 bus specifications in which the system bus (to which the main memory is connected) is controlled by a 100 or 133 MHz clock, respectively. Therefore, major manufacturers of memory chips produce 100 and 133 MHz SDRAM chips.

Transfers between the memory and the processor involve single words of data or small blocks of words. Large blocks, constituting a page of data, are transferred between the memory and the disks. The speed and efficiency of these transfers have a large impact on the performance of a computer system.

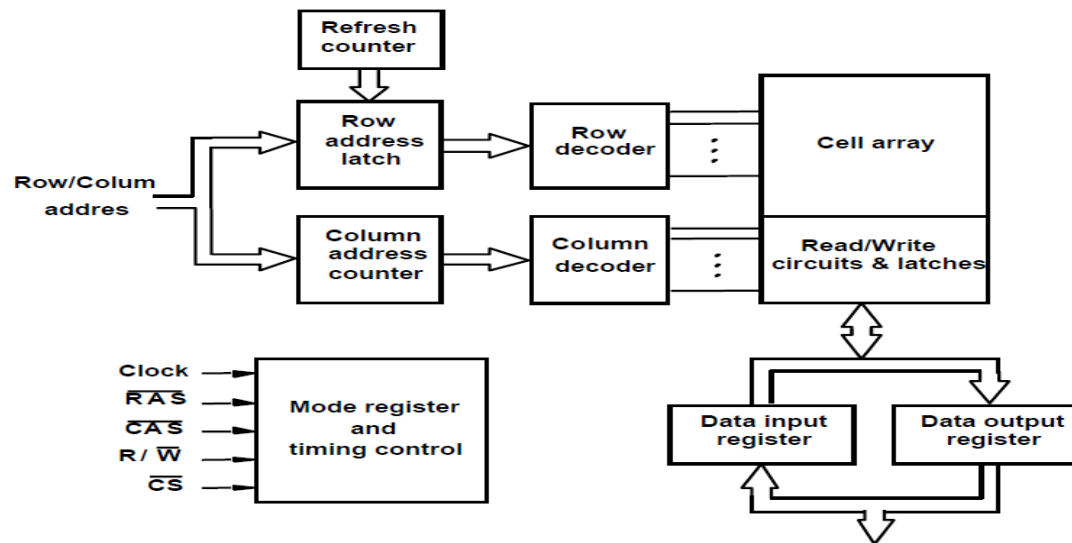


Figure Synchronous DRAM

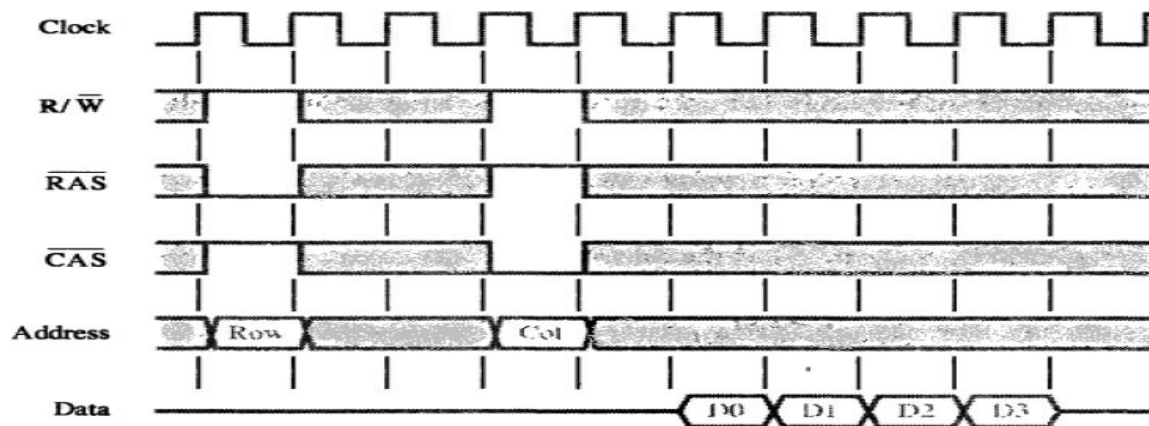


Figure 4.9 Burst read of length 4 in SDRAM

A good indication of the performance is given by two parameters: latency and bandwidth. The term memory latency is used to refer to the amount of time it takes to transfer a word of data to or from the memory. In the case of reading or writing a single word of data, the latency provides a complete indication of memory performance. But, in the case of burst operations that transfer a block of data, the time needed to complete the operation depends also on the rate at which successive

words can be transferred and on the size of the block. In block transfers, the term latency is used to denote the time it takes to transfer the first word of data.

When transferring blocks of data, it is of interest to know how much time is needed to transfer an entire block. Since blocks can be variable in size, it is useful to define a performance measure in terms of the number of bits or bytes that can be transferred in one second. This measure is often referred to as the memory bandwidth. The bandwidth of a memory unit (consisting of one or more memory chips) depends on the speed of access to the stored data and on the number of bits that can be accessed in parallel.

5) STRUCTURE OF LARGER MEMORIES: Many memory chips are connected to form a much larger memory.

Static Memory System: Consider a memory consisting of $2M$ (2,097,152) words of 32 bits each. Figure 4.10 shows how we can implement this memory using $512K \times 8$ static memory chips. Each column in the figure consists of four chips, which implement one byte position. Four of these sets provide the required $2M \times 32$ memory. Each chip has a control input called Chip Select. When this input is set to 1, it enables the chip to accept data from or to place data on its data lines. The data output for each chip is of the three-state type. Only the selected chip places data on the data output line, while all other outputs are in the high-impedance state. Twenty one address bits are needed to select a 32-bit word in this memory. The high-order 2 bits of the address are decoded to determine which of the four Chip Select control signals should be activated and the remaining 19 address bits are used to access specific byte locations inside each chip of the selected row. The R/W inputs of all chips are tied together to provide a common Read/Write control

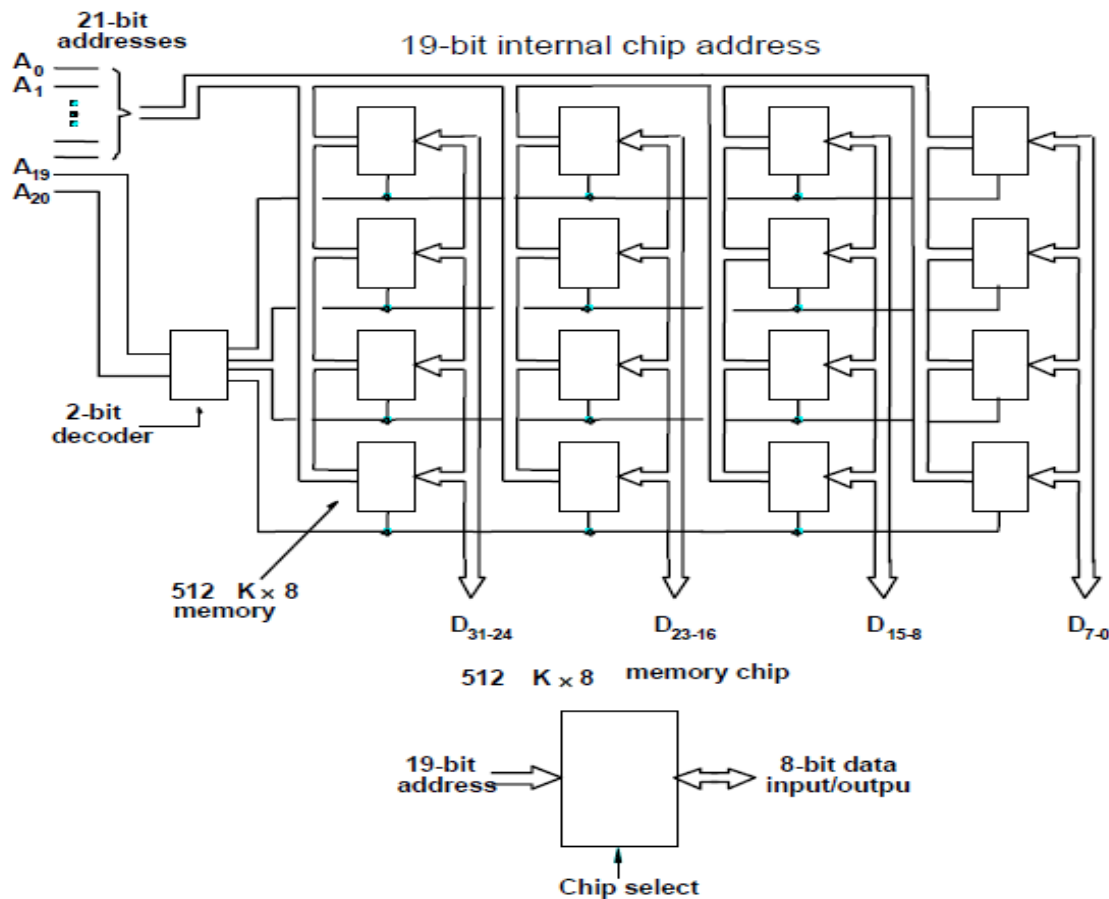


Figure 4.10 Organization of a $2M \times 32$ memory module using $512K \times 8$ static memory chips.

Dynamic Memory System: The organization of large dynamic memory systems is essentially the same as the memory shown in Figure 4.10. However, physical implementation is often done more conveniently in the form of memory modules. Modern computers use very large memories; even a small personal computer is likely to have at least 32M bytes of memory. Typical workstations have at least 128M bytes of memory. A large memory leads to better performance because more of the programs and data used in processing can be held in the memory, thus reducing the frequency of accessing the information in secondary storage. However, if a large memory is built by placing DRAM chips directly on the main system printed-circuit board that contains the processor, often referred to as a motherboard, it will occupy an unacceptably large amount of space on the board. Also, it is awkward to provide for future expansion of the memory, because space must be allocated and wiring provided for the maximum expected size. These packaging considerations have led to the development of larger memory units

known as SIMMs (Single In-line Memory Modules) and DIMMs (Dual In-line Memory Modules). Such a module is an assembly of several Memory chips on a separate small board that plugs vertically into a single socket on the motherboard. SIMMs and DIMMs of different sizes are designed .

6) Memory System Consideration

There are several factors that determine the choice of a RAM chip for a given application. These factors are the cost, speed, power dissipation, and size of the chip. Static RAMs are generally used only when very fast operation is the primary requirement. Their cost and size are adversely affected by the complexity of the circuit that realizes the basic cell. They are used mostly in cache memories. Dynamic RAMs are the predominant choice for implementing computer main memories. The high densities achievable in these chips make large memories economically feasible.

Memory' Controller: To reduce the number of pins, the dynamic memory chips use multiplexed address inputs. The address is divided into two parts. The high-order address bits, which select a row in the cell array, are provided first and latched into the memory chip under control of the RAS signal. Then, the low-order address bits, which select a column, are provided on the same address pins and latched using the CAS signal.

A typical processor issues all bits of an address at the same time. The required multiplexing of address bits is usually performed by a memory controller circuit,

which is interposed between the processor and the dynamic memory as shown in Figure 4.11. The controller accepts a complete address and the R/W signal from the processor, under control of a Request signal which indicates that a memory access operation is needed. The controller then forwards the row and column portions of the address to the memory and generates the RAS and CAS signals. Thus, the controller provides the RAS-CAS timing, in addition to its address multiplexing function. It also sends the R/W and CS signals to the memory. The CS signal is usually active low, hence it is shown as $\overline{CS}$ in Figure 4.11. Data lines are connected directly between the processor and the memory. Note that the clock signal is needed in SDRAM chips.

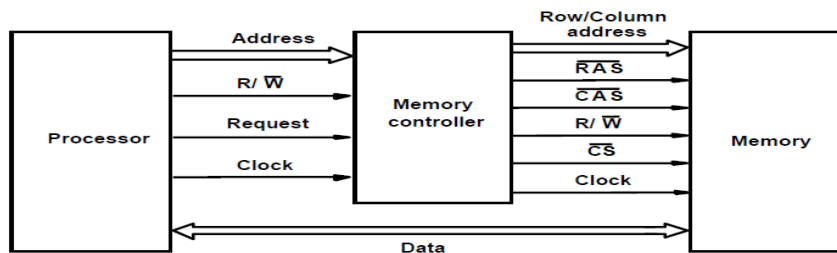


Figure 4.11 Use of memory controller

When used with DRAM chips, which do not have self-refreshing capability, the memory controller has to provide all the information needed to control the refreshing process. It contains a refresh counter that provides successive row addresses. Its function is to cause the refreshing of all rows to be done within the period specified for a particular device.

Refresh overhead: All dynamic memories have to be refreshed. In older DRAMs, a typical period for refreshing all rows was 16ms. In typical SDRAMs, a typical period is 64 ms

7) Rambus Memory

Introduction: A very wide bus is expensive and requires a lot of space on a motherboard. An alternative approach is to implement a narrow bus that is much faster. This approach was used by Rambus Inc. to develop a proprietary design known as Rambus. The key feature of Rambus technology is a fast signaling method used to transfer information between chips. Instead of using signals that have voltage levels of either 0 or V_{supply} to represent the logic values, the signals consist of much smaller voltage swings around a reference voltage, V_{ref} . The reference voltage is about 2 V, and the two logic values are represented by 0.3 V swings above and below V_{ref} . This type of signaling is generally known as

differential signaling. Small voltage swings make it possible to have short transition times, which allows for a high speed of transmission.

Rambus requires specially designed memory chips. These chips use cell arrays based on the standard DRAM technology. Multiple banks of cell arrays are used to access more than one word at a time. Circuitry needed to interface to the Rambus channel is included on the chip. Such chips are known as Rambus DRAMs (RDRAMs).

The original specification of Rambus provided for a channel consisting of 9 data lines and a number of control and power supply lines. Eight of the data lines are intended for transferring a byte of data. The ninth data line can be used for purposes such as parity checking. Subsequent specifications allow for additional channels. A two-channel Rambus, also known as Direct RDRAM, has 18 data lines intended to transfer two bytes of data at a time. There are no separate address lines.

Communication between the processor, or some other device that can serve as a master, and RDRAM modules, which serve as slaves, is carried out by means of packets transmitted on the data lines. There are three types of packets: request, acknowledge, and data. A request packet issued by the master indicates the type of operation that is to be performed. It contains the address of the desired memory location and includes an 8-bit count that specifies the number of bytes involved in the transfer. The operation types include memory reads and writes, as well as reading and writing of various control registers in the RDRAM chips. When the master issues a request packet, the addressed slave responds by returning a positive acknowledgement packet if it can immediately satisfy the request. Otherwise, the slave indicates that it is "busy" by returning a negative acknowledgement packet, in which case the master will try again.

The number of bits in a request packet exceeds the number of data lines, which means that several clock cycles are needed to transmit the entire packet. Using a narrow communication link is compensated by the very high rate of transmission. RDRAM chips can be assembled into larger modules, similar to SIMMs and DIMMs. One such module, called RIMM, can hold up to 16 RDRAMs. Rambus technology competes directly with the DDR SDRAM technology. Each has certain advantages and disadvantages. A nontechnical consideration is that the specification of DDR SDRAM is an open standard, while RDRAM is a proprietary design of Rambus Inc. for which the manufacturers of chips have to pay a royalty.

➤ Read only Memories

Read Only Memories (ROM): RAM memories are volatile memories, which mean that they lose the stored information if power is turned off. There are many applications that need memory devices which retain the stored retained even if power is turned off. A practical solution is to provide a small amount of nonvolatile memory that holds the instructions whose execution results in loading the boot program from the disk. The contents of such memory can be read as if they were SRAM or DRAM memories. But, a special writing process is needed to place the information into this memory. Since its normal operation involves only reading of stored data, a memory of this type is called read-only memory (ROM).

1)ROM Figure 4.12 shows a possible configuration for a ROM cell. A logic value 0 is stored in the cell if the transistor is connected to ground at point P; otherwise, a 1 is stored. The bit line is connected through a resistor to the power supply. To read the state of the cell, the word line is activated. Thus, the transistor switch is closed and the voltage on the bit line drops to near zero if there is a connection between the transistor and ground. If there is no connection to ground, the bit line remains at the high voltage, indicating a 1. A sense circuit at the end of the bit line generates the proper output value. Data are written into a ROM when it is manufactured.

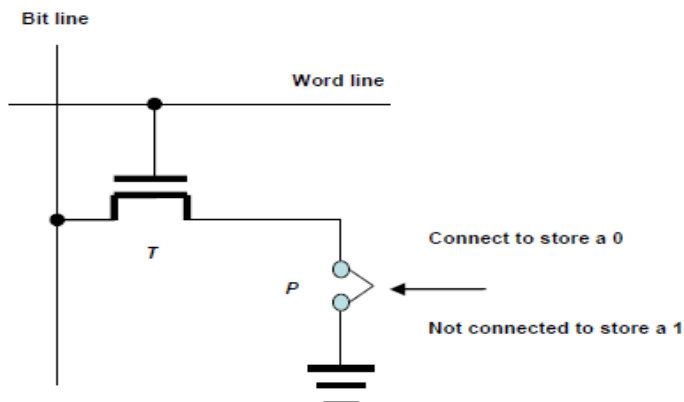


Figure 4.12 A ROM cell

2)PROM Some ROM designs allow the data to be loaded by the user, thus providing a programmable ROM (PROM). Programmability is achieved by inserting a fuse at point P in Figure 4.12. Before it is programmed, the memory contains all 0s. The user can insert 1s at the required locations by burning out the fuses at these locations using high-current pulses. Of course, this process is irreversible. PROMs provide flexibility and convenience not available with ROMs. The latter are economically attractive for storing fixed programs and data when

high volumes of ROMs are produced. However, the cost of preparing the masks needed for storing a particular information pattern in ROMs makes them very expensive when only a small number are required. In this case, PROMs provide a faster and considerably less expensive approach because they can be programmed directly by the user.

3)EPROM: Another type of ROM chip allows the stored data to be erased and new data to be loaded. Such an erasable, reprogrammable ROM is usually called an EPROM. It provides considerable flexibility during the development phase of digital systems. Since EPROMs are capable of retaining stored information for a long time, they can be used in place of ROMs while software is being developed. In this way, memory changes and updates can be easily made. An EPROM cell has a structure similar to the ROM cell in Figure 4.12. In an EPROM cell, however, the connection to ground is always made at point P and a special transistor is used, which has the ability to function either as a normal transistor or as a disabled transistor that is always turned off. This transistor can be programmed to behave as a permanently open switch, by injecting charge into it that becomes trapped inside.

The important advantage of EPROM chips is that their contents can be erased and reprogrammed. Erasure requires dissipating the charges trapped in the transistors of memory cells; this can be done by exposing the chip to ultraviolet light. For this reason, EPROM chips are mounted in packages that have transparent windows.

4)EEPROM: A significant disadvantage of EPROMs is that a chip must be physically removed from the circuit for reprogramming and that its entire contents are erased by the ultraviolet light. It is possible to implement another version of erasable PROMs that can be both programmed and erased electrically. Such chips, called EEPROMs, do not have to be removed for erasure. Moreover, it is possible to erase the cell contents selectively. The only disadvantage of EEPROMs is that different voltages are needed for erasing, writing, and reading the stored data.

5)FLASH MEMORIES: An approach similar to EEPROM technology has more recently given rise to flash memory devices. A flash cell is based on a single transistor controlled by trapped charge, just like an EEPROM cell. While similar in some respects, there are also substantial differences between flash and EEPROM devices. In EEPROM it is possible to read and write the contents of a single cell. In a flash device it is possible to read the contents of a single cell, but it is only possible to write an entire block of cells. Prior to writing, the previous contents of the block are erased. Flash devices have greater density, which leads to higher capacity and a lower cost per bit. They require a single power supply voltage, and

consume less power in their operation. The low power consumption of flash memory makes it attractive for use in portable equipment that is battery driven. Typical applications include hand-held computers, cell phones, digital cameras, and MP3 music players.

Single flash chips do not provide sufficient storage capacity for the applications mentioned above. Larger memory modules consisting of a number of chips are needed. There are two popular choices for the implementation of such modules: flash cards and flash drives.

FlashCards

One way of constructing a larger module is to mount flash chips on a small card. Such flash cards have a standard interface that makes them usable in a variety of products. A card is simply plugged into a conveniently accessible slot. Flash cards come in a variety of memory sizes. Typical sizes are 8, 32, 64 MB and so on.

Flashdrive

Larger flash memory modules have been developed to replace, hard disk drives. These flash drives are designed to fully emulate the hard disks, to the point that they can be fitted into standard disk drive bays. The fact that flash drives are solid state electronic devices that have no movable parts provides some important advantages. They have shorter seek and access times, which results in faster response. They have lower power consumption, which makes them attractive for battery driven applications, and they are also insensitive to vibration. Another type of ROM chip allows the stored data to be erased and new data to be loaded. The disadvantages of flash drives are less storage capacity, higher cost per bit and it will become weak after it has been written several times.

➤ Cache Memories

Cache memories: In most of the computer system, the speed of the main memory is very low than the speed of modern processors. For good performance, it is important to devise a scheme that reduces the time needed to access the necessary information. Since the speed of the main memory unit is limited by electronic and packaging constraints, the solution must be sought in a different architectural arrangement an efficient solution is to use a fast cache memory which essentially makes the main memory appear to the processor to be faster.

The effectiveness of the cache mechanism is based on a property of computer programs called locality of reference. Analysis of programs shows that most of their execution time is spent on routines in which many instructions are

executed repeatedly. These instructions may constitute a simple loop, nested loops, or a few procedures that repeatedly call each other. Many instructions in localized areas of the program are executed repeatedly during some time period, and the remainder of the program is accessed relatively infrequently. This is referred to as locality of reference. It manifests itself in two ways: temporal and spatial. The temporal means that a recently executed instruction is likely to be executed again very soon. The spatial aspect means that instructions in close proximity to a recently executed instruction (with respect to the instructions' addresses) are also likely to be executed soon. Block refers to a set of contiguous address locations of some size. Another term that is often used to refer to a cache block is cache line.

Consider the simple arrangement in Figure 4.14. When a Read request is received from the processor, the contents of a block of memory words containing the location specified are transferred into the cache one word at a time. Subsequently, when the program references any of the locations in this block, the desired contents are read directly from the cache. Usually, the cache memory can store a reasonable number of blocks at any given time, but this number is small compared to the total number of blocks in the main memory. The correspondence between the main memory blocks and those in the cache is specified by a mapping function. When the cache is full and a memory word (instruction or data) that is not in the cache is referenced, the cache control hardware must decide which block should be removed to create space for the new block that contains the referenced word. The collection of rules for making this decision constitutes the replacement algorithm.

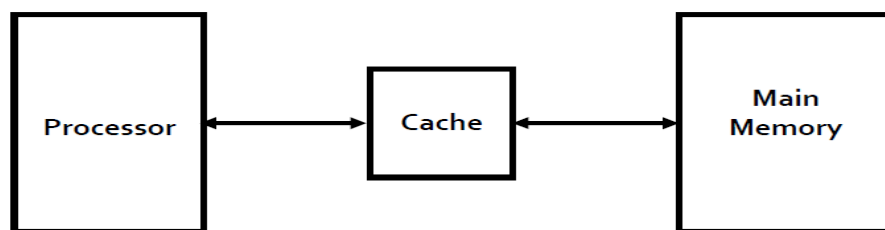


Figure Use of cache memory

The processor does not need to know explicitly about the existence of the cache. It simply issues Read and Write requests using addresses that refer to locations in the memory. The cache control circuitry determines whether the requested word currently exists in the cache. If it does, the Read or Write operation is performed on the appropriate cache location. In this case, a read or write hit is said to have occurred. In a Read operation, the main memory is not involved. For a Write operation, the system can proceed in two ways. In the first technique, called the

writethrough protocol, the cache location and the main memory location are updated simultaneously. The second technique is to update only the cache location and to mark it as updated with an associated flag bit, often called the dirty or modified bit. The main memory location of the word is updated later, when the block containing this marked word is to be removed from the cache to make room for a new block. This technique is known as the write-back, or copy-back, protocol. The write-through protocol is simpler, but it results in unnecessary Write operations in the main memory when a given cache word is updated several times during its cache residency. Note that the write-back protocol may also result in unnecessary Write operations because when a cache block is written back to the memory all words of the block are written back, even if only a single word has been changed while the block was in the cache.

When the addressed word in a Read operation is not in the cache, a read miss occurs. The block of words that contains the requested word is copied from the main memory into the cache. After the entire block is loaded into the cache, the particular word requested is forwarded to the processor. Alternatively, this word may be sent to the processor as soon as it is read from the main memory.

1)MAPPING FUNCTIONS: The process of keeping information of moved data blocks from main memory to cache memory is known as mapping. For example: Consider a cache consisting of 128 blocks of 16 words each, for a total of 2048 (2K) words, and assumes that the main memory is addressable by a 16-bit address. The main memory has 64K words, which we will view as 4K (4096) blocks of 16 words each. These mappings can be done in three ways. They are Direct mapping, Associative mapping and Set associative mapping.

Direct mapping: The simplest way to determine cache locations in which to store memory blocks is the direct-mapping technique. In this technique, block j of the main memory maps onto block j modulo 128 of the cache, as depicted in Figure 4.15. Thus, whenever one of the main memory blocks 0, 128, 256,... is loaded in the cache, it is stored in cache block 0. Blocks 1, 129, 257,... are stored in cache block 1, and so on. Since more than one memory block is mapped onto a given cache block position, contention may arise for that position even when the cache is not full. For example, instructions of a program may start in block 1 and continue in block 129, possibly after a branch. As this program is executed, both of these blocks must be transferred to the block-1 position in the cache. Contention is resolved by allowing the new block to overwrite the currently resident block. In this case, the replacement algorithm is trivial.

Placement of a block in the cache is determined from the memory address. The memory address can be divided into three fields, as shown in Figure 4.15. The low-order 4 bits select one of 16 words in a block. When a new block enters the cache, the 7-bit cache block field determines the cache position in which this block must be stored. The high-order 5 bits of the memory address of the block are stored in 5 tag bits associated with its location in the cache. They identify which of the 32 blocks that are mapped into this cache position are currently resident in the cache. As execution proceeds, the 7-bit cache block field of each address generated by the processor points to a particular block location in the cache. The high-order 5 bits of the address are compared with the tag bits associated with that cache location. If they match, then the desired word is in that block of the cache. If there is no match, then the block containing the required word must first be read from the main memory and loaded into the cache. The direct-mapping technique is easy to implement, but it is not very flexible.

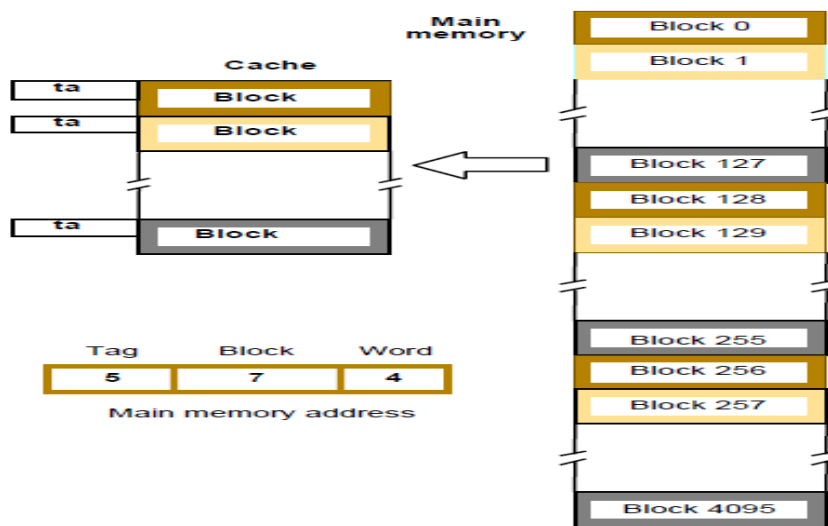


Figure 4.15 Direct-mapped cache.

Associative mapping: Figure 4.16 shows a much more flexible mapping method, in which a main memory block can be placed into any cache block position. In this case, 12 tag bits are required to identify a memory block when it is resident in the cache. The tag bits of an address received from the processor are compared to the tag bits of each block of the cache to see if the desired block is present. This is called the associative-mapping technique. It gives complete freedom in choosing the cache location in which to place the memory block. Thus, the space in the cache can be used more efficiently. A new block that has to be brought into the cache has to replace (eject) an existing block only if the cache is full. In this case, we need an algorithm to select the block to be replaced. Many replacement

algorithms are possible; the cost of an associative cache is higher than the cost of a direct-mapped cache because of the need to search all 128 tag patterns to determine whether a given block is in the cache. A search of this kind is called an associative search. For performance reasons, the tags must be searched in parallel.

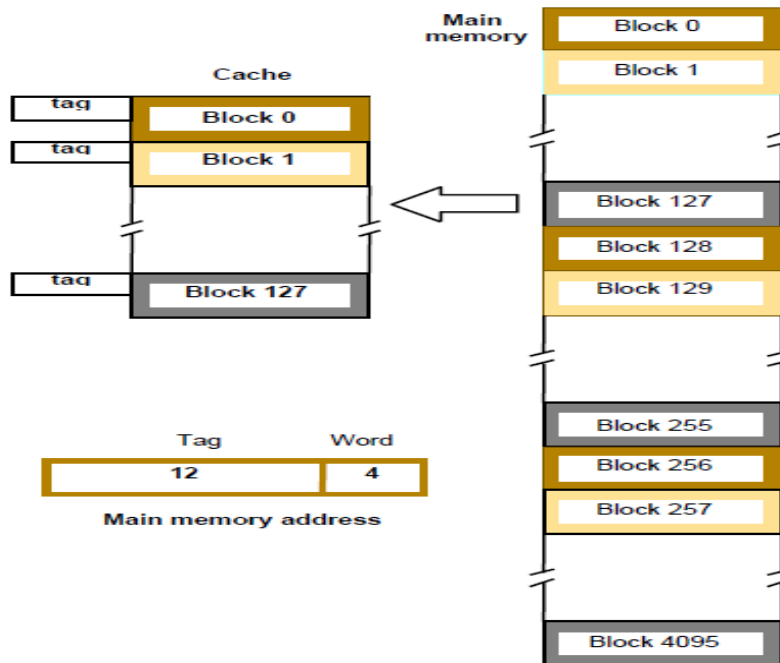


Figure 4.16 Associative-mapped cache.

Set-Associative

mapping

A combination of the direct- and associative mapping techniques can be used. Blocks of the cache are grouped into sets, and the mapping allows a block of the main memory to reside in any block of a specific set. Hence, the contention problem of the direct method is eased by having a few choices for block placement. At the same time, the hardware cost is reduced by decreasing the size of the associative search. An example of this set-associative-mapping technique is shown in Figure 4.17 for a cache with two blocks per set. In this case, memory blocks 0, 64, 128, ..., 4032 map into cache set 0, and they can occupy either of the two block positions within this set. Having 64 sets means that the 6-bit set field of the address determines which set of the cache might contain the desired block. The tag field of the address must then be associatively compared to the tags of the two blocks of the set to check if the desired block is present. This two-way associative search is simple to implement.

The number of blocks per set is a parameter that can be selected to suit the requirements of a particular computer. For the main memory and cache

sizes in Figure 4.17, four blocks per set can be accommodated by a 5-bit set field, eight blocks per set by a 4-bit set field, and so on. The extreme condition of 128 blocks per set requires no set bits and corresponds to the fully associative technique, with 12 tag bits. The other extreme of one block per set is the direct-mapping method. A cache that has k blocks per set is referred to as a k -way set-associative cache.

improving cache performance: Two key factors in the commercial success of a computer are performance and cost; the best possible performance at the lowest cost is the objective. The challenge in considering design alternatives is to improve the performance without increasing the cost. A common measure of success is the price/performance ratio. Performance depends on how fast machine instructions can be brought into the processor for execution and how fast they can be executed. The memory hierarchy is used for the best price/performance ratio. The main purpose of this hierarchy is to create a memory that the processor sees as having a short access time and a large capacity. Each level of the hierarchy plays an important role. The speed and efficiency of data transfer between various levels of the hierarchy are also of great significance. It is beneficial if transfers to and from the faster units can be done at a rate equal to that of the faster unit. This is not possible if both the slow and the fast units are accessed in the same manner, but it can be achieved when parallelism is used in the organization of the slower unit. An effective way to introduce parallelism is to use an interleaved organization.

➤ **Performance Considerations**

1)INTERLEAVING: If the main memory of a computer is structured as a collection of physically separate modules, each with its own address buffer register (ABR) and data buffer register (DBR), memory access operations may proceed in more than one module at the same time. Thus, the aggregate rate of transmission of words to and from the main memory system can be increased.

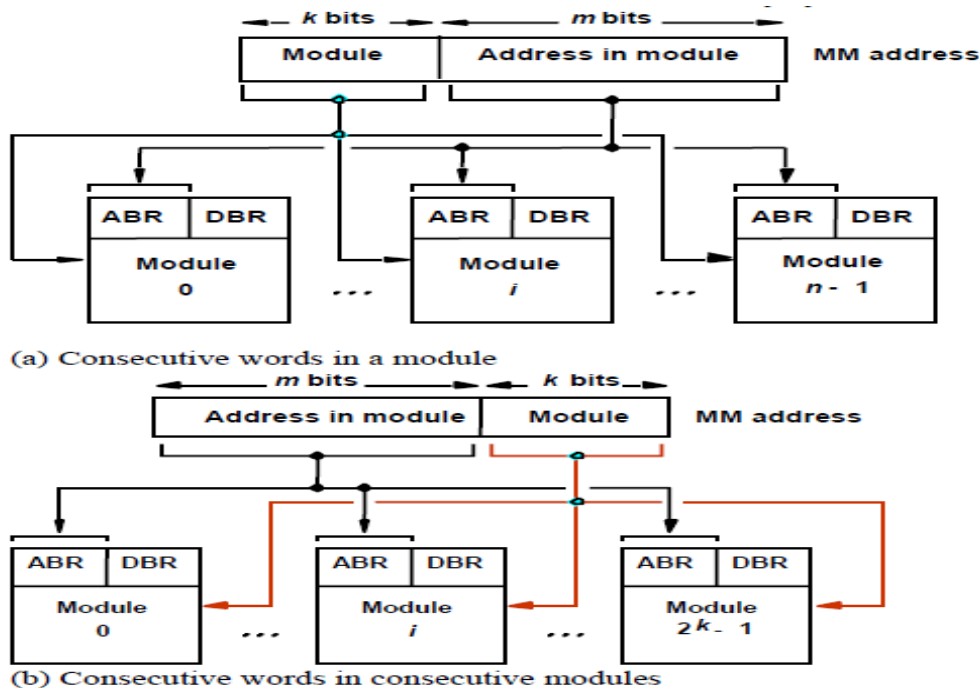


Figure 4.18 Addressing multiple-module memory systems. How individual addresses are distributed over the modules is critical in determining the average number of modules that can be kept busy as computations proceed. Two methods of address layout are indicated in Figure 4.18. In the first case, the memory address generated by the processor is decoded as shown in Figure 4.18a. The high-order k bits name one of n modules, and the low-order m bits name a particular word in that module. When consecutive locations are accessed, as happens when a block of data is transferred to a cache, only one module is involved. At the same time, however, devices with direct memory access (DMA) ability may be accessing information in other memory modules.

The second and more effective way to address the modules is shown in Figure 4.18b. It is called memory interleaving. The low-order k bits of the memory address select a module, and the high-order m bits name a location within that module. In this way, consecutive addresses are located in successive modules. Thus, any component of the system that generates requests for access to consecutive memory locations can keep several modules busy at anyone time. This results in both faster accesses to a block of data and higher average utilization of the memory system as a whole. To implement the interleaved structure, there must be 2^k modules; otherwise, there will be gaps of nonexistent locations in the memory address space

2) HIT RATE AND MISS PENALTY: An excellent indicator of the effectiveness of a particular implementation of the memory hierarchy is the success rate in accessing information at various levels of the hierarchy. Recall that a successful access to data in a cache is called a hit. The number of hits stated as a fraction of all attempted accesses is called the hit rate, and the miss rate is the number of misses stated as a fraction of attempted accesses. Ideally, the entire memory hierarchy would appear to the processor as a single memory unit that has the access time of a cache on the processor chip and the size of a magnetic disk. How close we get to this ideal depends largely on the hit rate at different levels of the hierarchy. High hit rates, well over 0.9, are essential for high-performance computers.

Performance is adversely affected by the actions that must be taken after a miss. The extra time needed to bring the desired information into the cache is called the miss penalty. This penalty is ultimately reflected in the time that the processor is stalled because the required instructions or data are not available for execution. In general, the miss penalty is the time needed to bring a block of data from a slower unit in the memory hierarchy to a faster unit. The miss penalty is reduced if efficient mechanisms for transferring data between the various units of the hierarchy are implemented. The previous section shows how an interleaved memory can reduce the miss penalty substantially.

1) CACHES ON THE PROCESSOR CHIP: From the speed point of view, the optimal place for a cache is on the processor chip. Unfortunately, space on the processor chip is needed for many other functions; this limits the size of the cache that can be accommodated. All high-performance processor chips include some form of a cache. Some manufacturers have chosen to implement two separate caches, one for instructions and another for data. A combined cache for instructions and data is likely to have a somewhat better hit rate because it offers greater flexibility in mapping new information into the cache.

In high-performance processors two levels of caches are normally used. The L1 cache(s) is on the processor chip. The L2 cache, which is much larger, may be implemented externally using SRAM chips. But, a somewhat smaller L2 cache may also be implemented on the processor chip. If both L1 and L2 caches are used, the L1 cache should be designed to allow very fast access by the processor because its access time will have a large effect on the clock rate of the processor. A cache cannot be accessed at the same speed as a register file because the cache is much bigger and, hence, more complex. A practical way to speed up access to the cache

is to access more than one word simultaneously and then let the processor use them one at a time.

The L2 cache can be slower, but it should be much larger to ensure a high hit rate. Its speed is less critical because it only affects the miss penalty of the L1 cache. A workstation computer may include an L1 cache with the capacity of tens of kilobytes and an L2 cache of several megabytes. Including an L2 cache further reduces the impact of the main memory speed on the performance of a computer. The average access time experienced by the processor in a system with two levels of caches is

$$T_{ave} = h_1 C_1 + (1 - h_1) h_2 C_2 + (1 - h_1)(1 - h_2) M$$
 where

h_1 is the hit rate in the L1 cache.
 h_2 is the hit rate in the L2 cache.
 C_1 is the time to access information in the L1 cache.
 C_2 is the time to access information in the L2 cache.
 M is the time to access information in the main memory.
 The number of misses in the L2 cache, given by the term $(1 - h_1)(1 - h_2)$, should be low. If both h_1 and h_2 are in the 90 percent range, then the number of misses will be less than 1 percent of the processor's memory accesses.

4) OTHER ENHANCEMENTS: In addition to the main design issues just discussed, several other possibilities exist for enhancing performance. They are

Write Buffer: When the write-through protocol is used, each write operation results in writing a new value into the main memory. If the processor must wait for the memory function to be completed, as we have assumed until now, then the processor is slowed down by all write requests. Yet the processor typically does not immediately depend on the result of a write operation, so it is not necessary for the processor to wait for the write request to be completed. To improve performance, a write buffer can be included for temporary storage of write requests. The processor places each write request into this buffer and continues execution of the next instruction. The write requests stored in the write buffer are sent to the main memory whenever the memory is not responding to read requests. Note that it is important that the read requests be serviced immediately because the processor usually cannot proceed without the data that are to be read from the memory. Hence, these requests are given priority over write requests.

Write-back protocol: A different situation occurs with the write-back protocol. In this case, the write operations are simply performed on the corresponding word in

the cache. But consider what happens when a new block of data is to be brought into the cache as a result of a read miss, which replaces an existing block that has some dirty data. The dirty block has to be written into the main memory. If the required write-back is performed first, then the processor will have to wait longer for the new block to be read into the cache. It is more prudent to read the new block first. This can be arranged by providing a fast write buffer for temporary storage of the dirty block that is ejected from the cache while the new block is being read. Afterward, the contents of the buffer are written into the main memory. Thus, the write buffer also works well for the write-back protocol.

Prefetching: In the cache mechanism, new data are brought into the cache when they are first needed. A read miss occurs, and the desired data are loaded from the main memory. The processor has to pause until the new data arrive, which is the effect of the miss penalty.

To avoid stalling the processor, it is possible to prefetch the data into the cache before they are needed. The simplest way to do this is through software. A special prefetch instruction may be provided in the instruction set of the processor. Executing this instruction causes the addressed data to be loaded into the cache, as in the case of a read miss. However, the processor does not wait for the referenced data. A prefetch instruction is inserted in a program to cause the data to be loaded in the cache by the time they are needed in the program. The hope is that prefetching will take place while the processor is busy executing instructions that do not result in a read miss, thus allowing accesses to the main memory to be overlapped with computation in the processor.

Lockup-Free cache: The software prefetching scheme just discussed does not work well if it interferes significantly with the normal execution of instructions. This is the case if the action of prefetching stops other accesses to the cache until the prefetch is completed. A cache of this type is said to be locked while it services a miss. This problem can be solved by modifying the basic cache structure to allow the processor to access the cache while a miss is being serviced. A cache that can support multiple outstanding misses is called lockup-free. Since it can service only one miss at a time, it must include circuitry that keeps track of all outstanding misses. This may be done with special registers that hold the pertinent information about these misses.

➤ Virtual Memory

Virtual memory: The physical main memory is not as large as the address space spanned by an address issued by the processor. When a program does not completely fit into the main memory, the parts of it not currently being executed are stored on secondary storage devices, such as magnetic disks. Of course, all parts of a program that are eventually executed are first brought into the main memory. When a new segment of a program is to be moved into a full memory, it must replace another segment already in the memory. The operating system moves programs and data automatically between the main memory and secondary storage. This process is known as swapping. Thus, the application programmer does not need to be aware of limitations imposed by the available main memory.

Techniques that automatically move program and data blocks into the physical main memory when they are required for execution are called virtual-memory techniques. Programs, and hence the processor, reference an instruction and data space that is independent of the available physical main memory space. The binary addresses that the processor issues for either instructions or data are called virtual or logical addresses. These addresses are translated into physical addresses by a combination of hardware and software components. If a virtual address refers to a part of the program or data space that is currently in the physical memory, then the contents of the appropriate location in the main memory are accessed immediately.

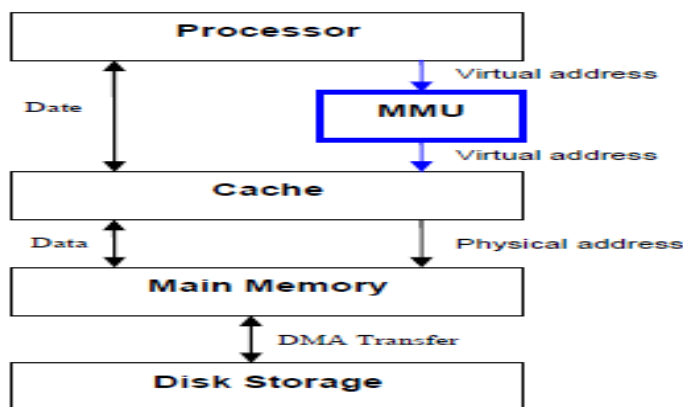


Figure 4.19 Virtual memory organization.

Figure 4.19 shows a typical organization that implements virtual memory. A special hardware unit, called the Memory Management Unit (MMU), translates virtual addresses into physical addresses. When the desired data (or instructions)

are in the main memory, these data are fetched as described in our presentation of the cache mechanism. If the data are not in the main memory, the MMU causes the operating system to bring the data into the memory from the disk. The DMA scheme is used to perform the data Transfer between the disk and the main memory.

1) ADDRESS TRANSLATION: The process of translating a virtual address into physical address is known as address translation. It can be done with the help of MMU. A simple method for translating virtual addresses into physical addresses is to assume that all programs and data are composed of fixed-length units called pages, each of which consists of a block of words that occupy contiguous locations in the main memory. Pages commonly range from 2K to 16K bytes in length. They constitute the basic unit of information that is moved between the main memory and the disk whenever the translation mechanism determines that a move is required. Pages should not be too small, because the access time of a magnetic disk is much longer (several milliseconds) than the access time of the main memory. The reason for this is that it takes a considerable amount of time to locate the data on the disk, but once located, the data can be transferred at a rate of several megabytes per second. On the other hand, if pages are too large it is possible that a substantial portion of a page may not be used, yet this unnecessary data will occupy valuable space in the main memory.

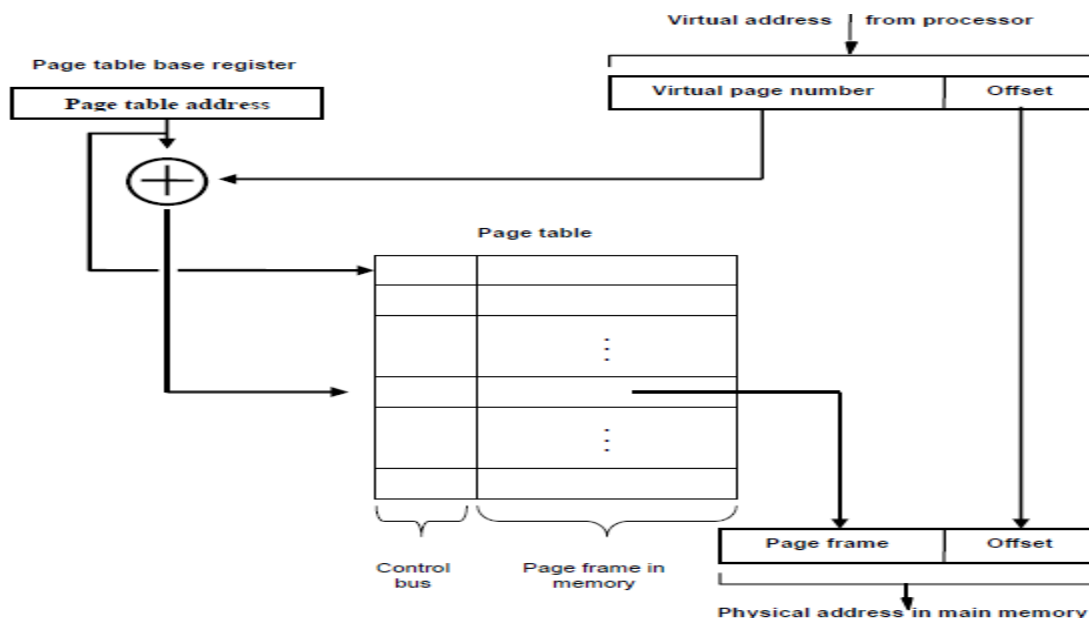


Figure 4.20 Virtual memory address translation.

The cache bridges the speed gap between the processor and the main memory and is implemented in hardware. The virtual-memory mechanism bridges the size and

speed gaps between the main memory and secondary storage and is usually implemented in part by software techniques. Conceptually, cache techniques and virtual-memory techniques are very similar. They differ mainly in the details of their implementation.

A virtual-memory address translation method based on the concept of fixed-length pages is shown schematically in Figure 4.20. Each virtual address generated by the processor, whether it is for an instruction fetch or an operand fetch/store operation, is interpreted as a virtual page number (high-order bits) followed by an offset (low-order bits) that specifies the location of a particular byte (or word) within a page. Information about the main memory location of each page is kept in a page table. This information includes the main memory address where the page is stored and the current status of the page. An area in the main memory that can hold one page is called a page frame. The starting address of the page table is kept in a page table base register. By adding the virtual page number to the contents of this register, the address of the corresponding entry in the page table is obtained. The contents of this location give the starting address of the page if that page currently resides in the main memory.

Each entry in the page table also includes some control bits that describe the status of the page while it is in the main memory. One bit indicates the validity of the page, that is, whether the page is actually loaded in the main memory. This bit allows the operating system to invalidate the page without actually removing it. Another bit indicates whether the page has been modified during its residency in the memory. As in cache memories, this information is needed to determine whether the page should be written back to the disk before it is removed from the main memory to make room for another page. Other control bits indicate various restrictions that may be imposed on accessing the page. For example, a program may be given full read and write permission, or it may be restricted to read accesses only.

Translation Lookaside Buffer(TLB) The MMU must use the page table information for every read and write access; so ideally, the page table should be situated within the MMU. Unfortunately, the page table may be rather large, and since the MMU is normally implemented as part of the processor chip (along with the primary cache), it is impossible to include a complete page table on this chip. Therefore, the page table is kept in the main memory. However, a copy of a small portion of the page table can be accommodated within the MMU.

This portion consists of the page table entries that correspond to the most recently accessed pages. A small cache, usually called the Translation Lookaside Buffer (TLB) is incorporated into the MMU for this purpose. The operation of the TLB with respect to the page table in the main memory is essentially the same as the operation of cache memory; the TLB must also include the virtual address of the entry. Figure 4.21 shows a possible organization of a TLB where the associative-mapping technique is used. Setassociative mapped TLBs are also found in commercial products.

An essential requirement is that the contents of the TLB be coherent with the contents of page tables in the memory. When the operating system changes the contents of page tables, it must simultaneously invalidate the corresponding entries in the TLB. One of the control bits in the TLB is provided for this purpose. When an entry is invalidated, the TLB will acquire the new information as part of the MMU's normal response to access misses. Address translation proceeds as follows. Given a virtual address, the MMU looks in the TLB for the referenced page. If the page table entry for this page is found in the TLB, the physical address is obtained immediately. If there is a miss in the TLB, then the required entry is obtained from the page table in the main memory and the TLB is updated.

When a program generates an access request to a page that is not in the main memory, a page fault is said to have occurred. The whole page must be brought from the disk into the memory before access can proceed. When it detects a page fault, the MMU asks the operating system to intervene by raising an exception (interrupt). Processing of the active task is interrupted, and control is transferred to the operating system. The operating system then copies the requested page from the disk into the main memory and returns control to the interrupted task. Because a long delay occurs while the page transfer takes place, the operating system may suspend execution of the task that caused the page fault and begin execution of another task whose pages are in the main memory.

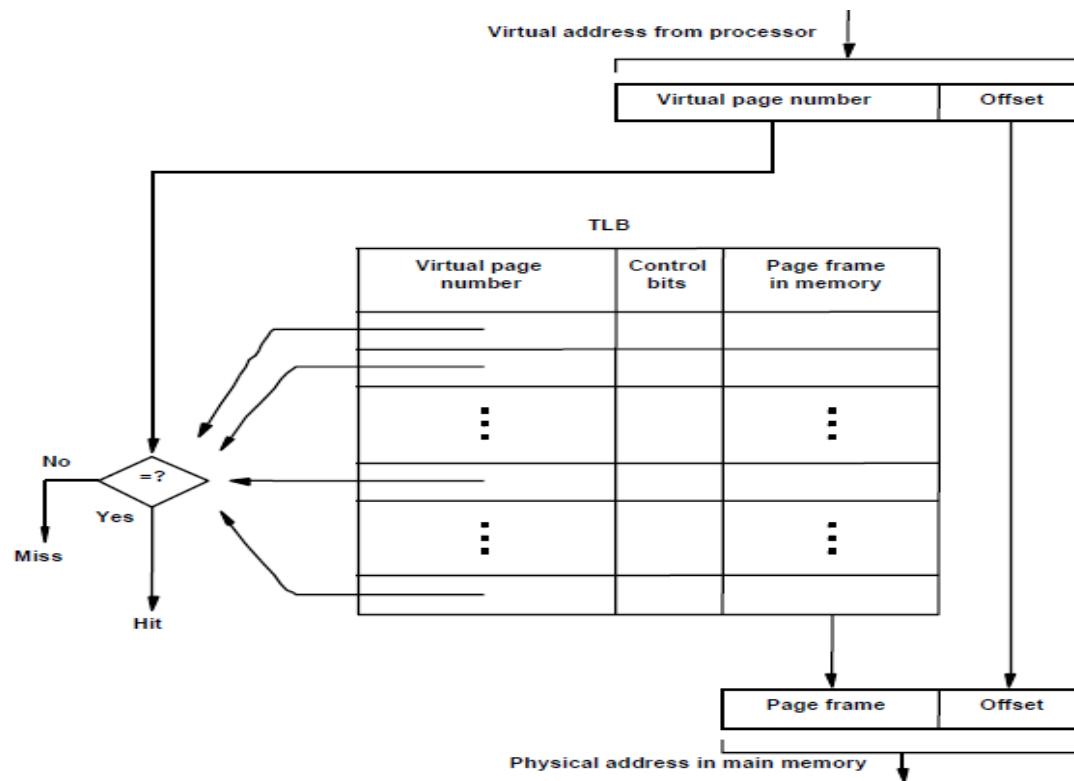


Figure 4.21 Use of an associative-mapped TLB.

If a new page is brought from the disk when the main memory is full, it must replace one of the resident pages. The problem of choosing which page to remove is just as critical here as it is in a cache, and the idea that programs spend most of their time in a few localized areas also applies. Because main memories are considerably larger than cache memories, it should be possible to keep relatively larger portions of a program in the main memory. This will reduce the frequency of transfers to and from the disk. Concepts similar to the FIFO, Optimal and LRU replacement algorithms can be applied to page replacement and the control bits in the page table entries can indicate usage. One simple scheme is based on a control bit that is set to 1 whenever the corresponding page is referenced (accessed). The operating system occasionally clears this bit in all page table entries, thus providing a simple way of determining which pages have not been used recently.

A modified page has to be written back to the disk before it is removed from the main memory. It is important to note that the write-through protocol, which is useful in the framework of cache memories, is not suitable for virtual memory. The access time of the disk is so long that it does not make sense to access it frequently to write small amounts of data. The address translation process in the MMU

requires some time to perform, mostly dependent on the time needed to look up entries in the TLB. Because of locality of reference, it is likely that many successive translations involve addresses on the same page. This is particularly evident in fetching instructions. Thus, we can reduce the average translation time by including one or more special registers that retain the virtual page number and the physical page frame of the most recently performed translations. The information in these registers can be accessed more quickly than the TLB.

➤ **Memory Management Requirements**

Virtual memory fills the gap between the physical memory and secondary memory (disc), when one large program is being executed and it does not fit into the available physical memory, parts of it (pages) are moved from the disk into the main memory when they are to be executed. Some software routines are needed to manage this movement of program segments. Management routines are part of the operating system of the computer. It is convenient to assemble the operating system routines into a virtual address space, called the system space, that is separate from the virtual space in which user application programs reside. The latter space is called the user space.

In fact, there may be a number of user spaces, one for each user. This is arranged by providing a separate page table for each user program. The MMU uses a page table base register to determine the address of the table to be used in the translation process. Hence, by changing the contents of this register, the operating system can switch from one space to another. The physical main memory is thus shared by the active pages of the system space and several user spaces. However, only the pages that belong to one of these spaces are accessible at any given time.

In any computer system in which independent user programs coexist in the main memory, the notion of protection must be addressed. No program should be allowed to destroy either the data or instructions of other programs in the memory. Such protection can be provided in several ways. Let us first consider the most basic form of protection. Recall that in the simplest case, the processor has two states, the supervisor state and the user state. As the names suggest, the processor is usually placed in the supervisor state when operating system routines are being executed and in the user state to execute user programs.

In the user state, some machine instructions (Access I/O devices, Poll for IO, perform DMA, catch hardware interrupt, Manipulate memory management, Set up

page tables, load/flush the TLB and CPU caches, etc.) cannot be executed. These privileged instructions, which include such operations as modifying the page table base register, can only be executed while the processor is in the supervisor state. Hence, a user program is prevented from accessing the page tables of other user spaces or of the system space.

It is sometimes desirable for one application program to have access to certain pages belonging to another program. The operating system can arrange this by causing these pages to appear in both spaces. The shared pages will therefore have entries in two different page tables. The control bits in each table entry can be set to control the access privileges granted to each program. For example, one program may be allowed to read and write a given page, while the other program may be given only read access.

➤ **Arithmetic**

- **Sign Magnitude**

Sign magnitude is a very simple representation of negative numbers. In sign magnitude the first bit is dedicated to represent the sign and hence it is called sign bit.

Sign bit '1' represents negative sign.

Sign bit '0' represents positive sign.

In sign magnitude representation of a n – bit number, the first bit will represent sign and rest $n-1$ bits represent magnitude of number.

For example,

- $+25 = 011001$

Where $11001 = 25$

And 0 for ‘+’

- $-25 = 111001$

Where $11001 = 25$

And 1 for ‘-’.

Range of number represented by sign magnitude method = $-(2^{n-1}-1)$ to $+(2^{n-1}-1)$
(for n bit number)

But there is one problem in sign magnitude and that is we have two representations of 0

+0 = 000000

– 0 = 100000

- **2's complement method**

To represent a negative number in this form, first we need to take the 1's complement of the number represented in simple positive binary form and then add 1 to it.

For example:

$(-8)_{10} = (1000)_2$

1's complement of 1000 = 0111

Adding 1 to it, $0111 + 1 = 1000$

So, $(-8)_{10} = (1000)_2$

Please don't get confused with $(8)_{10} = 1000$ and $(-8)_{10} = 1000$ as with 4 bits, we can't represent a positive number more than 7. So, 1000 is representing -8 only.

Range of number represented by 2's complement = $(-2^{n-1} \text{ to } 2^{n-1} - 1)$

➤ Fast Adders

1. We have done what is called a ripple carry adder.
 - The carry "ripples" from one bit to the next (LOB to HOB).
 - So the time required is proportional to the wordlength
 - Each carry can be computed with two levels of logic (any function can be so computed) hence the number of gate delays for an n bit adder is $2n$.
 - For a 4-bit adder 8 gate delays are required.
 - For an 16-bit adder 32 gate delays are required.
 - For an 32-bit adder 64 gate delays are required.
 - For an 64-bit adder 128 gate delays are required.
2. What about doing the entire 32 (or 64) bit adder with 2 levels of logic?
 - Such a circuit clearly exists. Why?
Ans: A two levels of logic circuit exists for *any* function.
 - But it would be very expensive: many gates and wires.
 - The big problem: When expressed with two levels of logic, the AND and OR gates have high fan-in, i.e., they have a large number of inputs. It is *not* true that a 64-input AND takes the same time as a 2-input AND.
 - Unless you are doing full custom VLSI, you get a toolbox of primitive functions (say 4 input NAND) and must build from that

3. There are faster adders, e.g. carry lookahead and carry save. We will study carry lookahead adders.

Carry Lookahead Adder (CLA)

This adder is much faster than the ripple adder we did before, especially for wide (i.e., many bit) addition.

- For each bit position we have two input bits, a and b (really should say a_i and b_i as I will do below).
- We can, in one gate delay, calculate two other bits called generate g and propagate p, defined as follows:
- The idea for propagate is that p is true if the current bit will propagate a carry from its input to its output.
- It is easy to see that $p = (a \text{ OR } b)$, i.e. if and only if $(a \text{ OR } b)$ then if there is a carry in then there is a carry out
- The idea for generate is that g is true if the current bit will generate a carry out (independent of the carry in).
- It is easy to see that $g = (a \text{ AND } b)$, i.e. if and only if $(a \text{ AND } b)$ then there must be a carry-out independent of the carry-in

To summarize, using a subscript i to represent the bit number,

to generate a carry: $g_i = a_i b_i$

to propagate a carry: $p_i = a_i + b_i$

H&P give a [plumbing analogue](#) for generate and propagate.

Given the generates and propagates, we can calculate all the carries for a 4-bit addition (recall that $c_0 = C_{in}$ is an input) as follows (this is the formula version of the plumbing):

$$c_1 = g_0 + p_0 c_0$$

$$c_2 = g_1 + p_1 c_1 = g_1 + p_1 g_0 + p_1 p_0 c_0$$

$$c_3 = g_2 + p_2 c_2 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$$

$$c_4 = g_3 + p_3 c_3 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0$$

Thus we can calculate $c_1 \dots c_4$ in just two additional gate delays (where we assume one gate can accept upto 5 inputs). Since we get g_i and p_i after one gate delay, the total delay for calculating all the carries is 3 (this includes c_4 =Carry-Out)

Each bit of the sum s_i can be calculated in 2 gate delays given a_i , b_i , and c_i . Thus, for 4-bit addition, 5 gate delays after we are given a , b and Carry-In, we have calculated s and Carry-Out.

So, for 4-bit addition, the faster adder takes time 5 and the slower adder time 8.

Now we want to put four of these together to get a fast 16-bit adder.

As black boxes, both ripple-carry adders and carry-lookahead adders (CLAs) look the same.

We could simply put four CLAs together and let the Carry-Out from one be the Carry-In of the next. That is, we could put these CLAs together in a ripple-carry manner to get a hybrid 16-bit adder.

- Since the Carry-Out is calculated in 3 gate delays, the Carry-In to the high order 4-bit adder is calculated in $3 \times 3 = 9$ delays.
- Hence the overall Carry-Out takes time $9 + 3 = 12$ and the high order four bits of the sum take $9 + 5 = 14$. The other bits take less time.
- So this mixed 16-bit adder takes 14 gate delays compared with $2 \times 16 = 32$ for a straight ripple-carry 16-bit adder.

We want to do better so we will put the 4-bit carry-lookahead adders together in a carry-lookahead manner. Thus the diagram above is **not** what we are going to do.

- We have 33 inputs $a_0, \dots, a_{15}$; $b_0, \dots, b_{15}$; c_0 =Carry-In
- We want 17 outputs $s_0, \dots, s_{15}$; c_{16} =Carry-Out
- Again we are assuming a gate can accept upto 5 inputs.
- It is important that the number of inputs per gate does not grow with the number of bits in each number.
- If the technology available supplies only 4-input gates (instead of the 5-input gates we are assuming), we would use groups of three bits rather than four

We start by determining ``super generate" and ``super propagate" bits.

- The super generate indicates whether the **4-bit adder** constructed above generates a Carry-Out.
- The super propagate indicates whether the **4-bit adder** constructed above propagates a Carry-In to a Carry-Out.

$P0 = p3 \ p2 \ p1 \ p0$ Does the low order 4-bit adder
propagate a carry?

$P1 = p7 \ p6 \ p5 \ p4$

$P2 = p11 \ p10 \ p9 \ p8$

$P3 = p15 \ p14 \ p13 \ p12$ Does the high order 4-bit adder
propagate a carry?

$G0 = g3 + p3 \ g2 + p3 \ p2 \ g1 + p3 \ p2 \ p1 \ g0$ Does low order 4-bit
adder generate a carry

$G1 = g7 + p7 \ g6 + p7 \ p6 \ g5 + p7 \ p6 \ p5 \ g4$

$G2 = g11 + p11 \ g10 + p11 \ p10 \ g9 + p11 \ p10 \ p9 \ g8$

$G3 = g15 + p15 \ g14 + p15 \ p14 \ g13 + p15 \ p14 \ p13 \ g12$

From these super generates and super propagates, we can calculate the
super carries, i.e. the carries for the four 4-bit adders.

- The first super carry $C0$, the Carry-In to the low-order 4-bit adder, is just $c0$ the input Carry-In.
- The second super carry $C1$ is the Carry-Out of the low-order 4-bit adder (which is also the Carry-In to the 2nd 4-bit adder.
- The last super carry $C4$ is the Carry-out of the high-order 4-bit adder (which is also the overall Carry-out of the entire 16-bit adder).

$C1 = G0 + P0 \ c0$

$C2 = G1 + P1 \ C1 = G1 + P1 \ G0 + P1 \ P0 \ c0$

$$C3 = G2 + P2 C2 = G2 + P2 G1 + P2 P1 G0 + P2 P1 P0 c0$$

$$C4 = G3 + P3 C3 = G3 + P3 G2 + P3 P2 G1 + P3 P2 P1 G0 + P3 P2 P1 P0 c0$$

Now these C's (together with the original inputs a and b) are just what the 4-bit CLAs need.

How long does this take, again assuming 5 input gates?

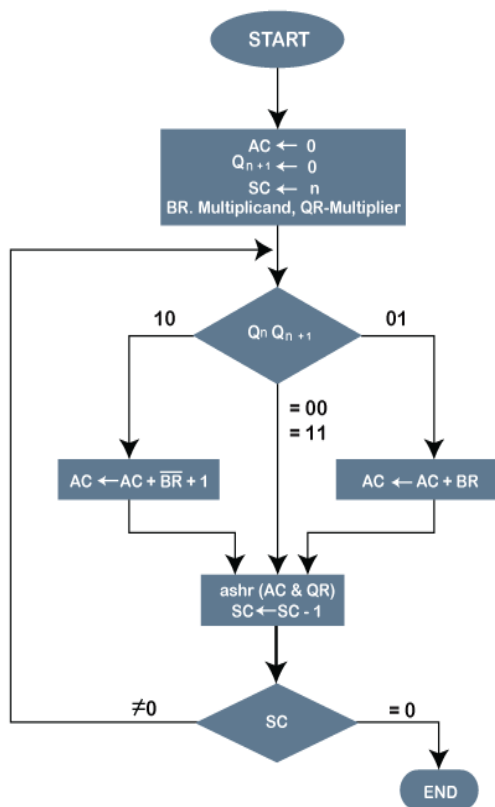
1. We calculate the p's and g's (lower case) in 1 gate delay (as with the 4-bit CLA).
2. We calculate the P's one gate delay after we have the p's or 2 gate delays after we start.
3. The G's are determined 2 gate delays after we have the g's and p's. So the G's are done 3 gate delays after we start.
4. The C's are determined 2 gate delays after the P's and G's. So the C's are done 5 gate delays after we start.
5. Now the C's are sent back to the 4-bit CLAs, which have already calculated the p's and g's. The C's are calculated in 2 more gate delays (7 total) and the s's 2 more after that (9 total).

In summary, a 16-bit CLA takes 9 cycles instead of 32 for a ripple carry adder and 14 for the mixed adder.

➤ Signed-Operand Multiplication

Booth's Multiplication Algorithm

The booth algorithm is a multiplication algorithm that allows us to multiply the two signed binary integers in 2's complement, respectively. It is also used to speed up the performance of the multiplication process. It is very efficient too. It works on the string bits 0's in the multiplier that requires no additional bit only shift the right-most string bits and a string of 1's in a multiplier bit weight 2^k to weight 2^m that can be considered as $2^{k+1} - 2^m$.



Following is the pictorial representation of the Booth's Algorithm:

In the above flowchart, initially, AC and Q_{n+1} bits are set to 0, and the SC is a sequence counter that represents the total bits set n , which is equal to the number of bits in the multiplier. There are BR that represent the multiplicand bits, and QR represents the multiplier bits. After that, we encountered two bits of the multiplier as Q_n and Q_{n+1} , where Q_n represents the last bit of QR, and Q_{n+1} represents the

incremented bit of Q_n by 1. Suppose two bits of the multiplier is equal to 10; it means that we have to subtract the multiplier from the partial product in the accumulator AC and then perform the arithmetic shift operation (ashr). If the two of the multipliers equal to 01, it means we need to perform the addition of the multiplicand to the partial product in accumulator AC and then perform the arithmetic shift operation (ashr), including Q_{n+1} . The arithmetic shift operation is used in Booth's algorithm to shift AC and QR bits to the right by one and remains the sign bit in AC unchanged. And the sequence counter is continuously decremented till the computational loop is repeated, equal to the number of bits (n).

Working on the Booth Algorithm

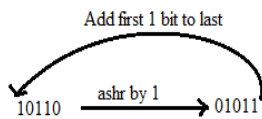
1. Set the Multiplicand and Multiplier binary bits as M and Q, respectively.
2. Initially, we set the AC and Q_{n+1} registers value to 0.
3. SC represents the number of Multiplier bits (Q), and it is a sequence counter that is continuously decremented till equal to the number of bits (n) or reached to 0.
4. A Q_n represents the last bit of the Q, and the Q_{n+1} shows the incremented bit of Q_n by 1.
5. On each cycle of the booth algorithm, Q_n and Q_{n+1} bits will be checked on the following parameters as follows:
 - i. When two bits Q_n and Q_{n+1} are 00 or 11, we simply perform the arithmetic shift right operation (ashr) to the partial product AC. And the bits of Q_n and Q_{n+1} is incremented by 1 bit.
 - ii. If the bits of Q_n and Q_{n+1} is shows to 01, the multiplicand bits (M) will be added to the AC (Accumulator register). After that, we perform the right shift operation to the AC and QR bits by 1.
 - iii. If the bits of Q_n and Q_{n+1} is shows to 10, the multiplicand bits (M) will be subtracted from the AC (Accumulator register). After that, we perform the right shift operation to the AC and QR bits by 1.
6. The operation continuously works till we reached $n - 1$ bit in the booth algorithm.
7. Results of the Multiplication binary bits will be stored in the AC and QR

registers.

There are two methods used in Booth's Algorithm:

ISC (Right Shift Circular)

It shifts the right-most bit of the binary number, and then it is added to the beginning of the binary bits.



ISA (Right Shift Arithmetic)

It adds the two binary bits and then shift the result to the right by 1-bit position.

Example: $0100 + 0110 \Rightarrow 1010$, after adding the binary number shift each bit by 1 to the right and put the first bit of resultant to the beginning of the new bit.

Example: Multiply the two numbers 7 and 3 by using the Booth's multiplication algorithm.

Ans. Here we have two numbers, 7 and 3. First of all, we need to convert 7 and 3 into binary numbers like $7 = (0111)$ and $3 = (0011)$. Now set 7 (in binary 0111) as multiplicand (M) and 3 (in binary 0011) as a multiplier (Q). And SC (Sequence Count) represents the number of bits, and here we have 4 bits, so set the $SC = 4$. Also, it shows the number of iteration cycles of the booth's algorithms and then cycles run $SC = SC - 1$ time.

Q_n	Q_{n+1}	M = (0111) M' + 1 = (1001) & Operation	AC	Q	Q_{n+1}	SC
1	0	Initial	0000	0011	0	4
		Subtract (M' + 1)	1001			

			1001			
		Perform Arithmetic Right Shift operations (ashr)	1100	1001	1	3
1	1	Perform Arithmetic Right Shift operations (ashr)	1110	0100	1	2
0	1	Addition (A + M)	0111			
			0101	0100		
		Perform Arithmetic right shift operation	0010	1010	0	1
0	0	Perform Arithmetic right shift operation	0001	0101	0	0

The numerical example of the Booth's Multiplication Algorithm is $7 \times 3 = 21$ and the binary representation of 21 is 10101. Here, we get the resultant in binary 00010101. Now we convert it into decimal, as $(000010101)_{10} = 2 \cdot 4 + 2 \cdot 3 + 2 \cdot 2 + 2 \cdot 1 + 2 \cdot 0 \Rightarrow 21$.

Example: Multiply the two numbers 23 and -9 by using the Booth's multiplication algorithm.

Here, $M = 23 = (010111)$ and $Q = -9 = (110111)$

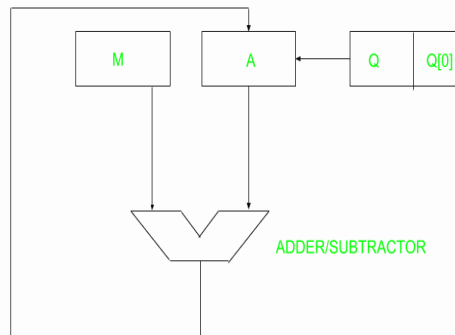
Q_n + 1	Q_n	$M = 0\ 1\ 0\ 1\ 1\ 1$ $M' + 1 = 1\ 0\ 1\ 0\ 0\ 1$	AC	Q	Q_n 1	SC^+
		Initially	000000	110111	0	6
1	0	Subtract M	101001			
			101001			
		Perform Arithmetic right shift operation	110100	111011	1	5
1	1	Perform Arithmetic right shift operation	111010	011101	1	4
1	1	Perform Arithmetic right shift operation	111101	001110	1	3
0	1	Addition (A + M)	010111			
			010100			
		Perform Arithmetic right shift operation	001010	000111	0	2
1	0	Subtract M	101001			
			110011			
		Perform Arithmetic right shift operation	111001	100011	1	1
1	1	Perform Arithmetic	111100	110001	1	0

	right shift operation			
--	--------------------------	--	--	--

➤ Integer Division

A division algorithm provides a quotient and a remainder when we divide two number. They are generally of two type **slow algorithm and fast algorithm**. Slow division algorithm are restoring, non-restoring, non-performing restoring, SRT algorithm and under fast comes Newton–Raphson and Goldschmidt.

In this article, will be performing restoring algorithm for unsigned integer. Restoring term is due to fact that value of register A is restored after each iteration.



Here, register Q contain quotient and register A contain remainder. Here, n-bit dividend is loaded in Q and divisor is loaded in M. Value of Register is initially kept 0 and this is the register whose value is restored during iteration due to which it is named Restoring.

Let's pick the step involved:

- **Step-1:** First the registers are initialized with corresponding values (Q = Dividend, M = Divisor, A = 0, n = number of bits in dividend)
- **Step-2:** Then the content of register A and Q is shifted right as if they are a single unit
- **Step-3:** Then content of register M is subtracted from A and result is stored in A
- **Step-4:** Then the most significant bit of the A is checked if it is 0 the least significant bit of Q is set to 1 otherwise if it is 1 the least significant bit of Q is set to 0 and value of register A is restored i.e the value of A before the subtraction with M
- **Step-5:** The value of counter n is decremented

- **Step-6:** If the value of n becomes zero we get of the loop otherwise we repeat from step 2
- **Step-7:** Finally, the register Q contain the quotient and A contain remainder

Examples:

Perform Division Restoring Algorithm

Dividend = 11

Divisor = 3

n	M	A	Q	Operation
4	00011	00000	1011	initialize
	00011	00001	011_	shift left AQ
	00011	11110	011_	A=A-M
	00011	00001	0110	Q[0]=0 And restore A
3	00011	00010	110_	shift left AQ
	00011	11111	110_	A=A-M
	00011	00010	1100	Q[0]=0
2	00011	00101	100_	shift left AQ
	00011	00010	100_	A=A-M
	00011	00010	1001	Q[0]=1
1	00011	00101	001_	shift left AQ
	00011	00010	001_	A=A-M
	00011	00010	0011	Q[0]=1

Remember to restore the value of A most significant bit of A is 1. As that register Q contain the quotient, i.e. 3 and register A contain remainder 2.

Floating –point Numbers and Operations

➤ Floating-point numbers and operations

Representation

When you have to represent very small or very large numbers, a fixed point representation will not do. The accuracy will be lost. Therefore, you will have to look at floating-point representations, where the binary point is assumed to be floating. When you consider a decimal number 12.34×10^7 , this can also be treated as 0.1234×10^9 , where 0.1234 is the fixed-point mantissa. The other part represents the exponent value, and indicates that the actual position of the binary point is 9 positions to the right (left) of the indicated binary point in the fraction.

Since the binary point can be moved to any position and the exponent value adjusted appropriately, it is called a floating-point representation. By convention, you generally go in for a normalized representation, wherein the floating-point is placed to the right of the first nonzero (significant) digit. The base need not be specified explicitly and the sign, the significant digits and the signed exponent constitute the representation.

The IEEE (Institute of Electrical and Electronics Engineers) has produced a standard for floating point arithmetic. This standard specifies how single precision (32 bit) and double precision (64 bit) floating point numbers are to be represented, as well as how arithmetic should be carried out on them. The IEEE single precision floating point standard representation requires a 32 bit word, which may be represented as numbered from 0 to 31, left to right. The first bit is the sign bit, S, the next eight bits are the exponent bits, 'E', and the final 23 bits are the fraction 'F'. Instead of the signed exponent E, the value stored is an unsigned integer $E' = E + 127$, called the excess-127 format. Therefore, E' is in the range $0 \leq E' \leq 255$.

S E'E'E'E'E'E'E'E' FFFFFFFFFFFFFFFFFFFFFFFF

0 1 8 9 31

The value V represented by the word may be determined as follows:

If $E' = 255$ and F is nonzero, then $V = \text{NaN}$ (“Not a number”)

- If $E' = 255$ and F is zero and S is 1, then $V = -\text{Infinity}$
- If $E' = 255$ and F is zero and S is 0, then $V = \text{Infinity}$
- If $0 < E' < 255$ then $V = (-1)^{S} * 2^{(E'-127)} * (1.F)$ where “1.F” is intended to represent the binary number created by prefixing F with an implicit leading 1 and a binary point.
- If $E' = 0$ and F is nonzero, then $V = (-1)^{S} * 2^{(-126)} * (0.F)$. These are “unnormalized” values.
- If $E' = 0$ and F is zero and S is 1, then $V = -0$
- If $E' = 0$ and F is zero and S is 0, then $V = 0$

For example,

$$0\ 00000000\ 000000000000000000000000 = 0$$
$$1\ 00000000\ 000000000000000000000000 = -0$$

63

The value V represented by the word may be determined as follows:

- If $E' = 2047$ and F is nonzero, then $V = \text{NaN}$ (“Not a number”)
- If $E' = 2047$ and F is zero and S is 1, then $V = -\text{Infinity}$
- If $E' = 2047$ and F is zero and S is 0, then $V = \text{Infinity}$
- If $0 < E' < 2047$ then $V = (-1)^{S} * 2^{(E'-1023)} * (1.F)$ where “1.F” is intended to represent the binary number created by prefixing F with an implicit leading 1 and a binary point.
- If $E' = 0$ and F is nonzero, then $V = (-1)^{S} * 2^{(-1022)} * (0.F)$ These are “unnormalized” values.
- If $E' = 0$ and F is zero and S is 1, then $V = -0$
- If $E' = 0$ and F is zero and S is 0, then $V = 0$

Arithmetic unit

Arithmetic operations on floating point numbers consist of addition, subtraction, multiplication and division. The operations are done with algorithms similar to those used on sign magnitude integers (because of the similarity of representation) — example, only add numbers of the same sign. If the numbers are of opposite sign, must do subtraction.

ADDITION

Example on decimal value given in scientific notation:

$$\begin{array}{r} 3.25 \times 10^{**3} \\ + 2.63 \times 10^{** -1} \\ \hline \end{array}$$

first step: align decimal points

second step: add

$$\begin{array}{r} 3.25 \quad \times 10^{**3} \\ + 0.000263 \times 10^{**3} \\ \hline \end{array}$$

$$3.250263 \times 10^{**3}$$

(presumes use of infinite precision, without regard for accuracy)

third step: normalize the result (already normalized!)

Example on floating pt. value given in binary:

.25 = 0 01111101 000000000000000000000000

100 = 0 10000101 100100000000000000000000

To add these fl. pt. representations,

step 1: align radix points

shifting the mantissa left by 1 bit decreases the exponent by 1

shifting the mantissa right by 1 bit increases the exponent by 1

we want to shift the mantissa right, because the bits that fall off the end should come from the least significant end of the mantissa

-> choose to shift the .25, since we want to increase it's exponent.

-> shift by 10000101

-01111101

00001000 (8) places.

0 01111101 000000000000000000000000 (original value)

0 01111110 100000000000000000000000 (shifted 1 place)

(note that hidden bit is shifted into msb of mantissa)

0 01111111 010000000000000000000000 (shifted 2 places)

0 10000000 001000000000000000000000 (shifted 3 places)

0 10000001 000100000000000000000000 (shifted 4 places)

0 10000010 000010000000000000000000 (shifted 5 places)

0 10000011 000001000000000000000000 (shifted 6 places)

0 10000100 000000100000000000000000 (shifted 7 places)

0 10000101 000000010000000000000000 (shifted 8 places)

step 2: add (don't forget the hidden bit for the 100)

0 10000101 1.100100000000000000000000 (100)

+ 0 10000101 0.000000010000000000000000 (.25)

0 10000101 1.100100010000000000000000

step 3: normalize the result (get the “hidden bit” to be a 1)
It already is for this example.

result is 0 10000101 100100010000000000000000

SUBTRACTION

Same as addition as far as alignment of radix points
Then the algorithm for subtraction of sign mag. numbers takes over.

before subtracting,
compare magnitudes (don’t forget the hidden bit!)
change sign bit if order of operands is changed.

don’t forget to normalize number afterward.

MULTIPLICATION

Example on decimal values given in scientific notation:

3.0 x 10 ** 1
+ 0.5 x 10 ** 2

Algorithm: multiply mantissas
add exponents

3.0 x 10 ** 1
+ 0.5 x 10 ** 2

1.50 x 10 ** 3

Example in binary: Consider a mantissa that is only 4 bits.

0 10000100 0100
x 1 00111100 1100

Add exponents:

always add true exponents (otherwise the bias gets added in twice)

DIVISION

It is similar to multiplication.

do unsigned division on the mantissas (don't forget the hidden bit)
subtract TRUE exponents

The organization of a floating point adder unit and the algorithm is given below.

The floating point multiplication algorithm is given below. A similar algorithm based on the steps discussed before can be used for division.

Rounding

The floating point arithmetic operations discussed above may produce a result with more digits than can be represented in 1.M. In such cases, the result must be *rounded* to fit into the available number of M positions. The extra bits that are used in intermediate calculations to improve the precision of the result are called *guard bits*. It is only a tradeoff of hardware cost (keeping extra bits) and speed versus accumulated rounding error, because finally these extra bits have to be rounded off to conform to the IEEE standard.

Rounding Methods:

- Truncate

Remove all digits beyond those supported
– 1.00100 -> 1.00

- Round up to the next value

– 1.00100 -> 1.01

- Round down to the previous value

- 1.00100 -> 1.00
- Differs from Truncate for negative numbers
- Round-to-nearest-even
 - Rounds to the even value (the one with an LSB of 0)
 - 1.00100 -> 1.00
 - 1.01100 -> 1.10
 - Produces zero average bias
 - Default mode

A product may have twice as many digits as the multiplier and multiplicand

 - $1.11 \times 1.01 = 10.0011$

For round-to-nearest-even, we need to know the value to the right of the LSB (*round bit*) and whether any other digits to the right of the round digit are 1's (the *sticky bit* is the OR of these digits). The IEEE standard requires the use of 3 extra bits of less significance than the 24 bits (of mantissa) implied in the single precision representation – guard bit, round bit and sticky bit. When a mantissa is to be shifted in order to align radix points, the bits that fall off the least significant end of the mantissa go into these extra bits (guard, round, and sticky bits). These bits can also be set by the normalization step in multiplication, and by extra bits of quotient (remainder) in division. The guard and round bits are just 2 extra bits of precision that are used in calculations. The sticky bit is an indication of what is/could be in lesser significant bits that are not kept. If a value of 1 ever is shifted into the sticky bit position, that sticky bit remains a 1 (“sticks” at 1), despite further shifts.

To summarize, in this module we have discussed the need for floating point numbers, the IEEE standard for representing floating point numbers, Floating point addition / subtraction, multiplication, division and the various rounding methods.

UNIT-V

➤ Basic Processing unit

➤ Some Fundamental Concepts

Fundamental concepts

To execute an instruction, processor has to perform following 3 steps:

- 1) Fetch contents of memory-location pointed to by PC. Content of this location is an instruction to be executed. The instructions are loaded into IR, Symbolically, this operation can be written as
$$IR \leftarrow [PC]$$
 - 2) Increment PC by 4
$$PC \leftarrow [PC] + 4$$
 - 3) Carry out the actions specified by instruction (in the IR).
- The first 2 steps are referred to as fetch phase;
Step 3 is referred to as execution phase.

SINGLE BUS ORGANIZATION

MDR has 2 inputs and 2 outputs. Data may be loaded

- into MDR either from memory-bus (external) or
- from processor-bus (internal).

MAR's input is connected to internal-bus, and MAR's output is connected to external-bus.

- Instruction-decoder & control-unit is responsible for
 - issuing the signals that control the operation of all the units inside the processor (and for interacting with memory bus).
 - implementing the actions specified by the instruction (loaded in the IR)
- Registers R0 through R(n-1) are provided for general purpose use by programmer.
- Three registers Y, Z & TEMP are used by processor for temporary storage during execution of some instructions. These are transparent to the programmer i.e. programmer need not be concerned with them because they are never referenced explicitly by any instruction.
- MUX(Multiplexer) selects either
 - output of Y or
 - constant-value 4(is used to increment PC content). This is provided as input A of ALU.
- B input of ALU is obtained directly from processor-bus.
- As instruction execution progresses, data are transferred from one register to another, often passing through ALU to perform arithmetic or logic operation.
- An instruction can be executed by performing one or more of

the following operations:

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

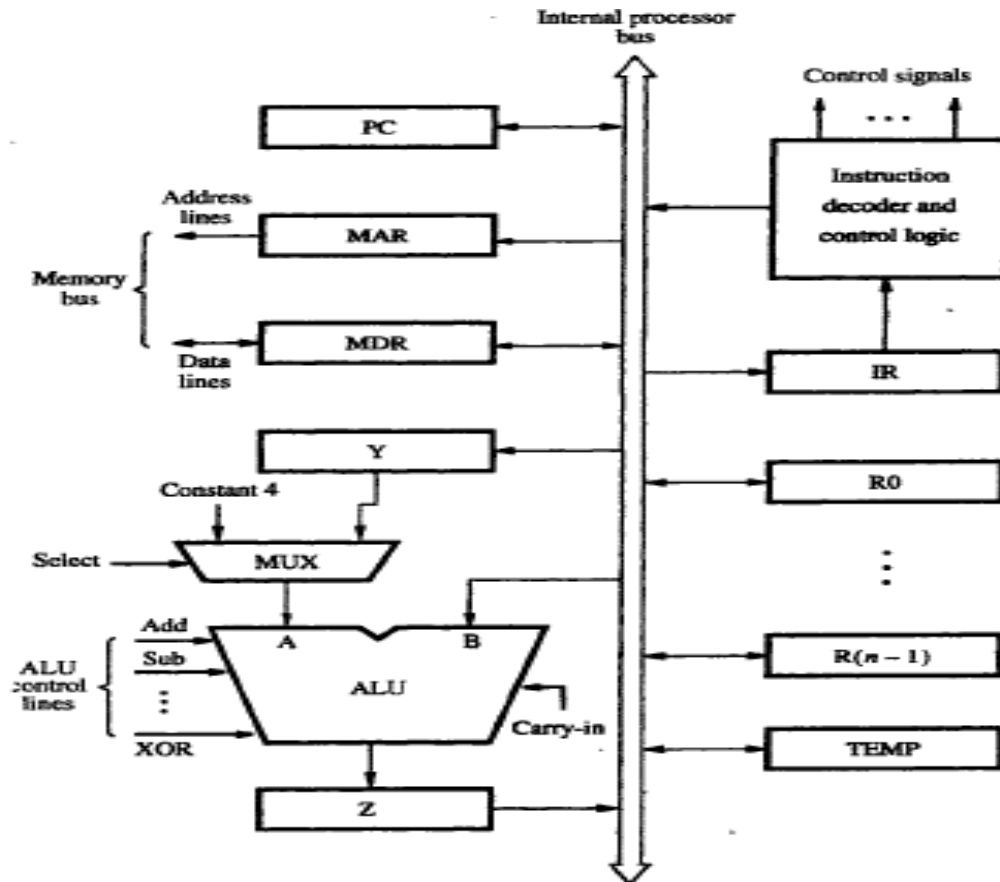


Figure 7.1 Single-bus organization of the datapath inside a processor.

1) Register Transfers

- Instruction execution involves a sequence of steps in which data are transferred from one register to another.
- Input & output of register R_i is connected to bus via switches controlled by 2 control-signals: R_{in} & R_{out} . These are called *gating signals*.
- When $R_{in}=1$, data on bus is loaded into R_i .
Similarly, when $R_{out}=1$, content of R_i is placed on bus.
- When $R_{out}=0$, bus can be used for transferring data from other registers.
- All operations and data transfers within the processor take place within time-periods defined by the processor-clock.
- When edge-triggered flip-flops are not used, 2 or more clock-signals may be needed to guarantee proper transfer of data. This is known as *multiphase clocking*.

Input & Output Gating for one Register Bit

- A 2-input multiplexer is used to select the data applied to the input of an edge-triggered D flip-flop.
- When $R_{i\text{in}}=1$, mux selects data on bus. This data will be loaded into flip-flop at rising-edge of clock. When $R_{i\text{in}}=0$, mux feeds back the value currently stored in flip-flop.
- Q output of flip-flop is connected to bus via a tri-state gate. When $R_{i\text{out}}=0$, gate's output is in the high-impedance state. (This corresponds to the open-circuit state of a switch).

When $R_{i\text{out}}=1$, the gate drives the bus to 0 or 1, depending on the value of

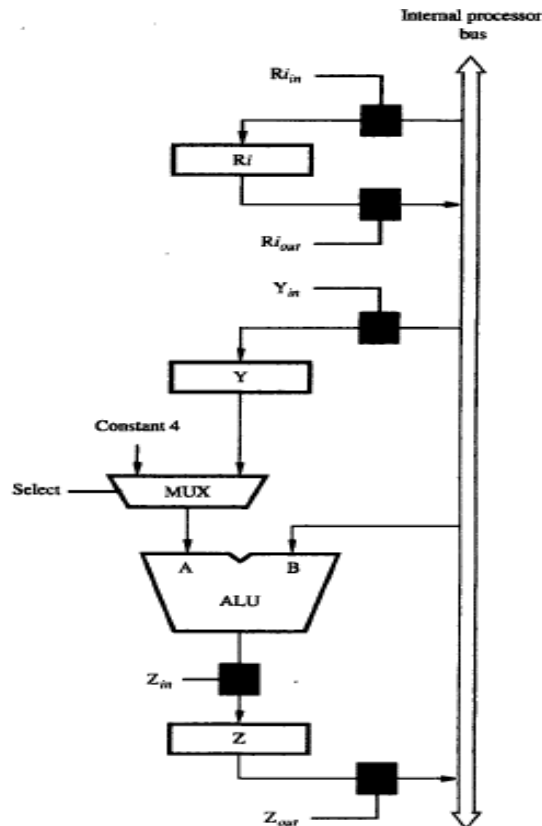


Figure 7.2 Input and output gating for the registers in Figure 7.1.

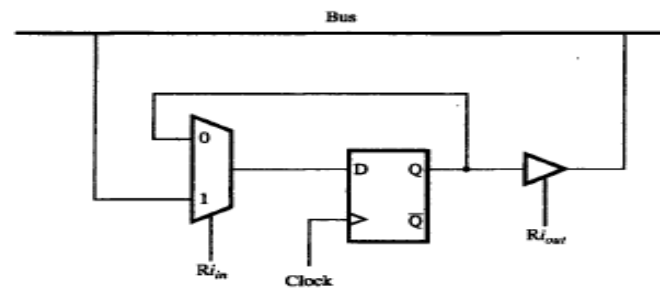


Figure 7.3 Input and output gating for one register bit.

Q.

2) Performing an Arithmetic or Logic Operation

- The ALU performs arithmetic operations on the 2 operands applied to its A and B inputs.
- One of the operands is output of MUX & the other operand is obtained directly from bus.
- The result (produced by the ALU) is stored temporarily in register Z.
- The sequence of operations for $[R3] \leftarrow [R1] + [R2]$ is as follows
 - 1) $R1_{\text{out}}, Y_{\text{in}}$ //transfer the contents of R1 to Y register
 - 2) $R2_{\text{out}}, \text{Select} Y, \text{Add}, Z_{\text{in}}$ //R2 contents are transferred directly to B input of ALU.

// The numbers of added. Sum stored in register Z

- 3) Zout, R3in //sum is transferred to register R3
- The signals are activated for the duration of the clock cycle corresponding to that step. All other signals are inactive.

Write the complete control sequence for the instruction : Move (Rs),Rd

- This instruction copies the contents of memory-location pointed to by Rs into Rd. This is a memory read operation. This requires the following actions
 - fetch the instruction
 - fetch the operand (i.e. the contents of the memory-location pointed by Rs).
 - transfer the data to Rd.
- The control-sequence is written as follows
 - 1) PCout, MARin, Read, Select4, Add, Zin
 - 2) Zout, PCin, Yin, WMFC
 - 3) MDRout, IRin
 - 4) Rs, MARin, Read
 - 5) MDRinE, WMFC
 - 6) MDRout, Rd, End

3) Fetching A Word From Memory

- To fetch instruction/data from memory, processor transfers required address to MAR (whose output is connected to address-lines of memory-bus).
At the same time, processor issues Read signal on control-lines of memory-bus.
- When requested-data are received from memory, they are stored in MDR. From MDR, they are transferred to other registers
- MFC (Memory Function Completed): Addressed-device sets MFC to 1 to indicate that the contents of the specified location
 - have been read &
 - are available on data-lines of memory-bus
- Consider the instruction Move (R1),R2. The sequence of steps is:
 - 1) R1out, MARin, Read ;desired address is loaded into MAR & Read command is issued
 - 2) MDRinE, WMFC ;load MDR from memory bus & Wait for MFC response from memory
 - 3) MDRout, R2in ;load R2 from MDR

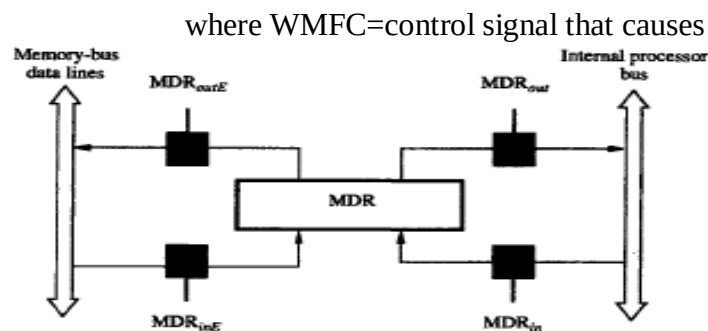


Figure 7.4 Connection and control signals for register MDR.

processor's control circuitry to wait for arrival of MFC signal

4) Storing a Word in Memory

- Consider the instruction *Move R2,(R1)*. This requires the following sequence:
 - 1) R1out, MARin ;desired address is loaded into MAR
 - 2) R2out, MDRin, Write ;data to be written are loaded into MDR & Write command is issued
 - 3) MDRoutE, WMFC ;load data into memory location pointed by R1 from MDR

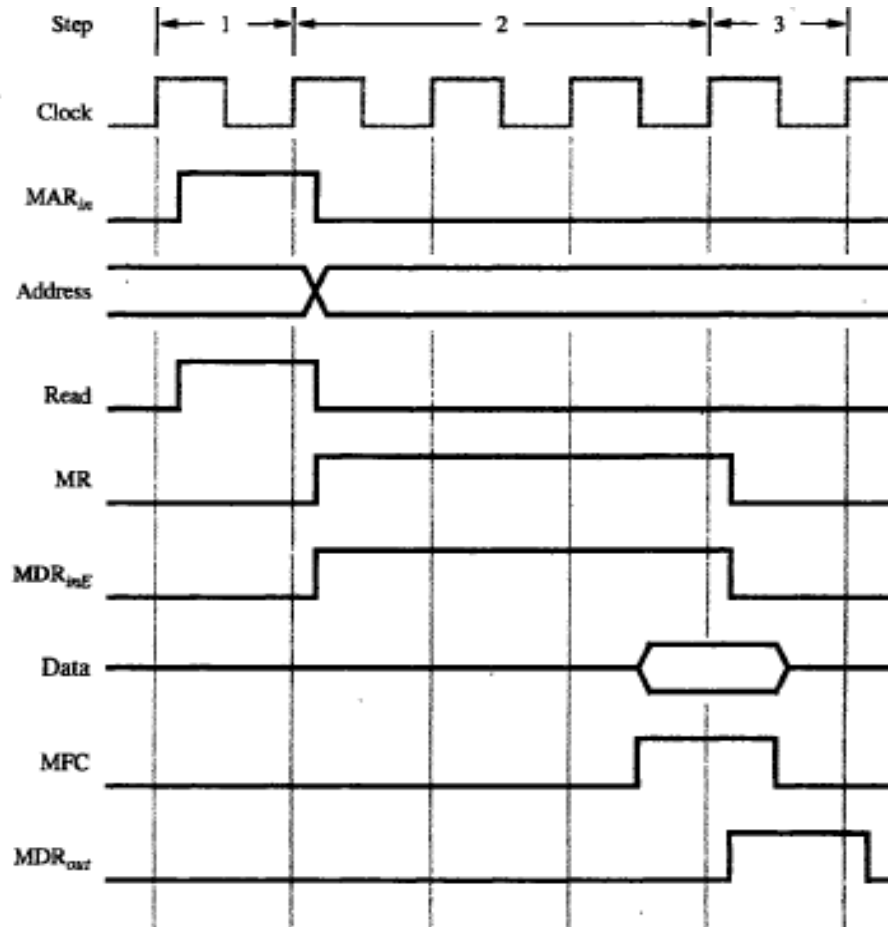


Figure 7.5 Timing of a memory Read operation.

➤ Execution of A Complete Instruction

- Consider the instruction *Add (R3),R1* which adds the contents of a memory-location pointed by R3 to register R1. Executing this instruction requires the following actions:
 - 1) Fetch the instruction.
 - 2) Fetch the first operand.
 - 3) Perform the addition.

- 4) Load the result into R1.
- Control sequence for execution of this instruction is as follows
 - 1) PCout, MARin, Read, Select4, Add, Zin
 - 2) Zout, PCin, Yin, WMFC
 - 3) MDRout, IRin
 - 4) R3out, MARin, Read
 - 5) R1out, Yin, WMFC
 - 6) MDRout, SelectY, Add, Zin
 - 7) Zout, R1in, End
- Instruction execution proceeds as follows:
 - Step1--> The instruction-fetch operation is initiated by loading contents of PC into MAR & sending a Read request to memory. The Select signal is set to Select4, which causes the Mux to select constant 4. This value is added to operand at input B (PC's content), and the result is stored in Z
 - Step2--> Updated value in Z is moved to PC.
 - Step3--> Fetched instruction is moved into MDR and then to IR.
 - Step4--> Contents of R3 are loaded into MAR & a memory read signal is issued. Step5--> Contents of R1 are transferred to Y to prepare for addition.
 - Step6--> When Read operation is completed, memory-operand is available in MDR, and the addition is performed.
 - Step7--> Sum is stored in Z, then transferred to R1. The End signal causes a new instruction fetch cycle to begin by returning to step1.

1) Branch Instructions

- Control sequence for an unconditional branch instruction is as follows:
 - 1) PCout, MARin, Read, Select4, Add, Zin
 - 2) Zout, PCin, Yin, WMFC
 - 3) MDRout, IRin
 - 4) Offset-field-of-IRout, Add, Zin
 - 5) Zout, PCin, End
- The processing starts, as usual, the fetch phase ends in step3.
- In step 4, the offset-value is extracted from IR by instruction-decoding circuit.
- Since the updated value of PC is already available in register Y, the offset X is gated onto the bus, and an addition operation is performed.
- In step 5, the result, which is the branch-address, is loaded into the PC.
- 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 1000 and branch target-address is 1200, then the value of X must be 196, since the PC will be containing the address 1004 after fetching the instruction at location 1000).
- In case of conditional branch, we need to check the status of the condition-codes before loading a new value into the PC.
 - e.g.: Offset-field-of-IRout, Add, Zin, If N=0 then End
 - If N=0, processor returns to step 1 immediately after step 4. If N=1, step 5

is performed to load a new value into PC.

➤ Multiple Bus Organozation

- All general-purpose registers are combined into a single block called the *register file*.
- Register-file has 3 ports. There are 2 outputs allowing the contents of 2 different registers to be simultaneously placed on the buses A and B.
- Register-file has 3 ports.
 - 1) Two output-ports allow the contents of 2 different registers to be simultaneously placed on buses A & B.
 - 2) Third input-port allows data on bus C to be loaded into a third register during the same clock-cycle.
- Buses A and B are used to transfer source-operands to A & B inputs of ALU.
- Result is transferred to destination over bus C.
- Incrementer-unit is used to increment PC by 4.
- Control sequence for the instruction *Add R4,R5,R6* is as follows
 - 1) PCout, R=B, MARin, Read, IncPC
 - 2) WMFC
 - 3) MDRout, R=B, IRin
 - 4) R4outA, R5outB, SelectA, Add, R6in, End
- Instruction execution proceeds as follows:
 - Step 1--> Contents of PC are passed through ALU using R=B control-signal and loaded into MAR to start a memory Read operation. At the same time, PC is incremented by 4.
 - Step2--> Processor waits for MFC signal from memory.
 - Step3--> Processor loads requested-data into MDR, and then transfers them to IR.
 - Step4--> The instruction is decoded and add operation take place in a single step.

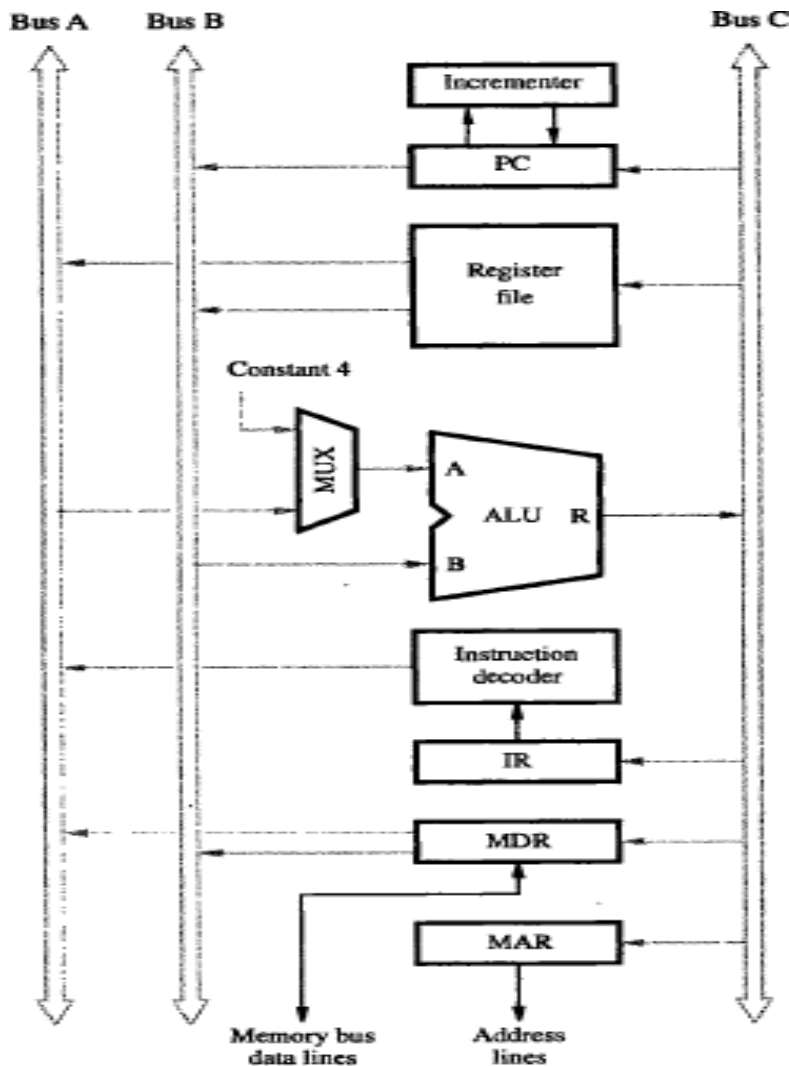


Figure 7.8 Three-bus organization of the datapath.

Note:

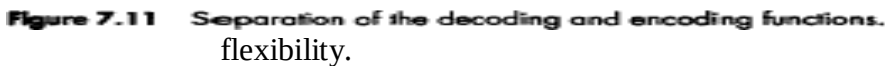
To execute instructions, the processor must have some means of generating the control signals needed in the proper sequence. There are two approaches for this purpose:

1) Hardwired control and 2) Microprogrammed control.

➤ **Hardwired Control**

- Decoder/encoder block is a combinational-circuit that generates required control-outputs depending on state of all its inputs.
- Step-decoder provides a separate signal line for each step in the control sequence.
Similarly, output of instruction-decoder consists of a separate line for each machine instruction.
- For any instruction loaded in IR, one of the output-lines INS_1 through INS_m is set to 1, and all other lines are set to 0.

- during T6 for an Add instruction
during T4 for unconditional branch instruction



1)Complete Processor

- This has separate processing-units to deal with integer data and floating-point data.
- A data-cache is inserted between these processing-units & main-memory.
- Instruction-unit fetches instructions
 - from an instruction-cache or
 - from main-memory when desired instructions are not already in cache
- Processor is connected to system-bus & hence to the rest of the computer by means of a bus interface
- Using separate caches for instructions & data is common practice in many processors today.
- A processor may include several units of each type to increase the potential for concurrent operations.

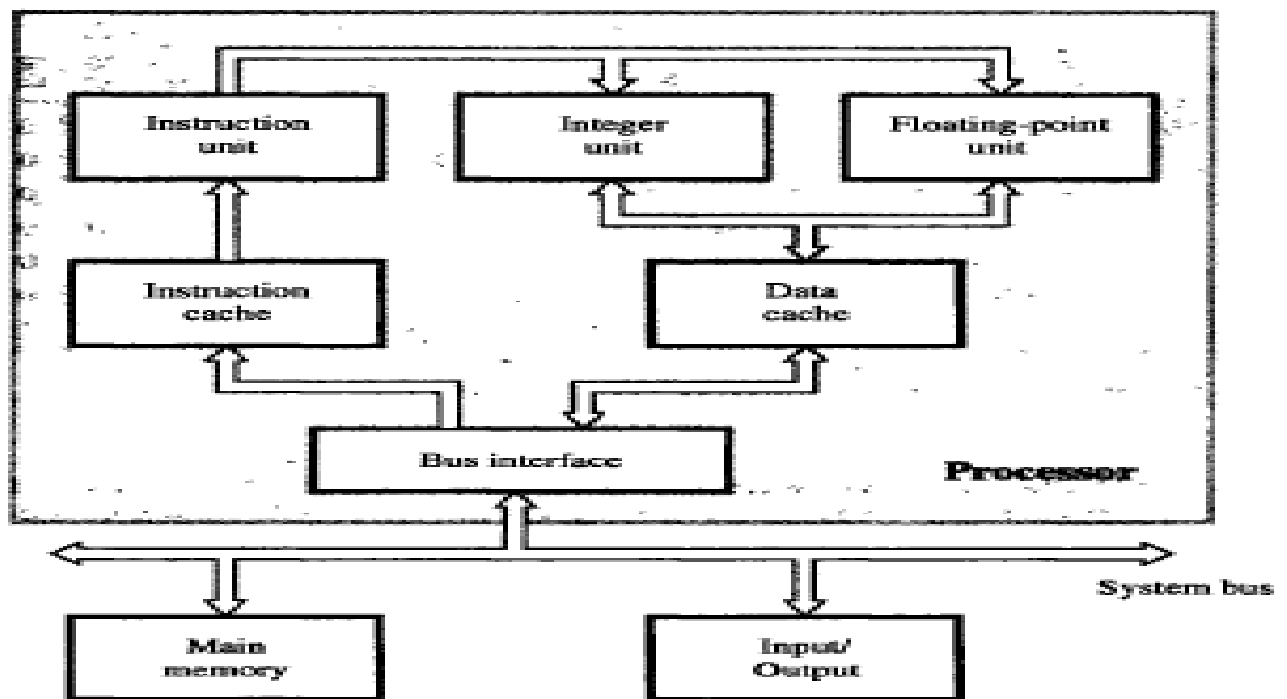


Figure 7.14 Block diagram of a complete processor.

➤ Micro Program Control

- Control-signals are generated by a program similar to machine language programs.
- *Control word(CW)* is a word whose individual bits represent various control-signals(like Add, End, Zin). {Each of the control-steps in control sequence of an instruction defines a unique combination of 1s & 0s in the CW}.
- Individual control-words in microroutine are referred to as *microinstructions*.
- A sequence of CWs corresponding to control-sequence of a machine instruction constitutes the *microroutine*.

- The microroutines for all instructions in the instruction-set of a computer are stored in a special memory called the *control store* (CS).
- Control-unit generates control-signals for any instruction by sequentially reading CWs of corresponding microroutine from CS.
- *Microprogram counter* (μPC) is used to read CWs sequentially from CS.
- Every time a new instruction is loaded into IR, output of "starting address generator" is loaded into μPC .
- Then, μPC is automatically incremented by clock, causing successive microinstructions to be read from CS.

Hence, control-signals are delivered to various parts of processor in correct

Micro-instruction	..	PC _{in}	PC _{out}	MAR _{in}	Read	MDR _{out}	IR _{in}	Y _{in}	Select	Add	Z _{in}	Z _{out}	R1 _{out}	R1 _{in}	R3 _{out}	WMFC	End	..
1		0	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0	
2		1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	
3		0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	
4		0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	
5		0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	
6		0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	
7		0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	

Figure 7.15 An example of microinstructions for Figure 7.6. sequence.

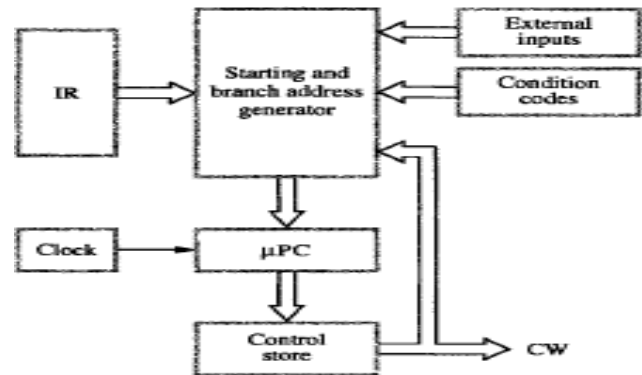


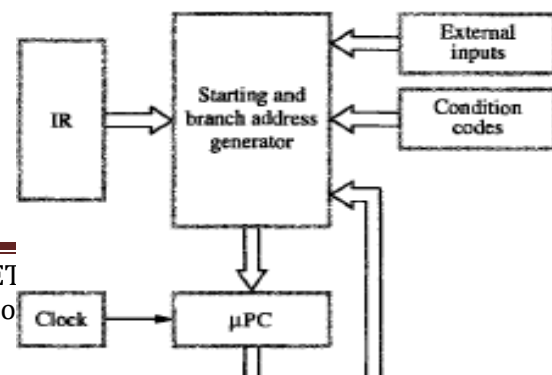
Figure 7.18 Organization of the control unit to allow conditional branching in the microprogram.

ORGANIZATION OF MICROPROGRAMMED CONTROL UNIT (TO SUPPORT CONDITIONAL BRANCHING)

- In case of conditional branching, microinstructions specify which of the external inputs, condition-codes should be checked as a condition for branching to take place.
- The *starting and branch address generator block* loads a new address into μPC when a microinstruction instructs it to do so.
- To allow implementation of a conditional branch, inputs to this block consist of
 - external inputs and condition-codes
 - contents of IR
- μPC is incremented every time a new microinstruction is fetched from microprogram memory except in following situations
 - i) When a new instruction is loaded into IR, μPC is loaded with starting-address of microroutine for that instruction.
 - ii) When a Branch microinstruction is encountered and branch condition is satisfied, μPC is loaded with branch-address.
 - iii) When an End microinstruction is encountered, μPC is loaded with address of

Address	Microinstruction
0	PC _{out} , MAR _{in} , Read, Select4, Add, Z _{in}
1	Z _{out} , PC _{in} , Y _{in} , WMFC
2	MDR _{out} , IR _{in}
3	Branch to starting address of appropriate microroutine
25	If N=0, then branch to microinstruction 0
26	Offset-field-of-IR _{out} , SelectY, Add, Z _{in}
27	Z _{out} , PC _{in} , End

technology {ICEI
epartment of Co



first CW in microroutine for instruction fetch cycle.

1) Micro Instructions

- Drawbacks of microprogrammed control:
 - 1) Assigning individual bits to each control-signal results in long microinstructions because the number of required signals is usually large.
 - 2) Available bit-space is poorly used because only a few bits are set to 1 in any given microinstruction.
- Solution: Signals can be grouped because
 - 1) Most signals are not needed simultaneously.
 - 2) Many signals are mutually exclusive.
- Grouping control-signals into fields requires a little more hardware because decoding-circuits must be used to decode bit patterns of each field into individual control signals.
- Advantage: This method results in a smaller control-store (only 20 bits are needed to store the patterns for the 42 signals).

Vertical organization	Horizontal organization
Highly encoded schemes that use compact codes to specify only a small number of control functions in each microinstruction are referred to as a vertical organization	The minimally encoded scheme in which many resources can be controlled with a single microinstruction is called a horizontal organization
This approach results in considerably slower operating speeds because more microinstructions are needed to perform the desired control functions	This approach is useful when a higher operating speed is desired and when the machine structure allows parallel use of resources

Micro Instruction

F1	F2	F3	F4	F5
F1 (4 bits)	F2 (3 bits)	F3 (3 bits)	F4 (4 bits)	F5 (2 bits)
0000: No transfer 0001: PC _{out} 0010: MDR _{out} 0011: Z _{out} 0100: R0 _{out} 0101: R1 _{out} 0110: R2 _{out} 0111: R3 _{out} 1010: TEMP _{out} 1011: Offset _{out}	000: No transfer 001: PC _{in} 010: IR _{in} 011: Z _{in} 100: R0 _{in} 101: R1 _{in} 110: R2 _{in} 111: R3 _{in}	000: No transfer 001: MAR _{in} 010: MDR _{in} 011: TEMP _{in} 100: Y _{in}	0000: Add 0001: Sub ⋮ 1111: XOR 16 ALU functions	00: No action 01: Read 10: Write
F6	F7	F8	...	
F6 (1 bit)	F7 (1 bit)	F8 (1 bit)		
0: Select Y 1: Select 4	0: No action 1: WMFC	0: Continue 1: End		

Figure 7.19 An example of a partial format for field-encoded microinstructions.

2)Micro Program Sequencing

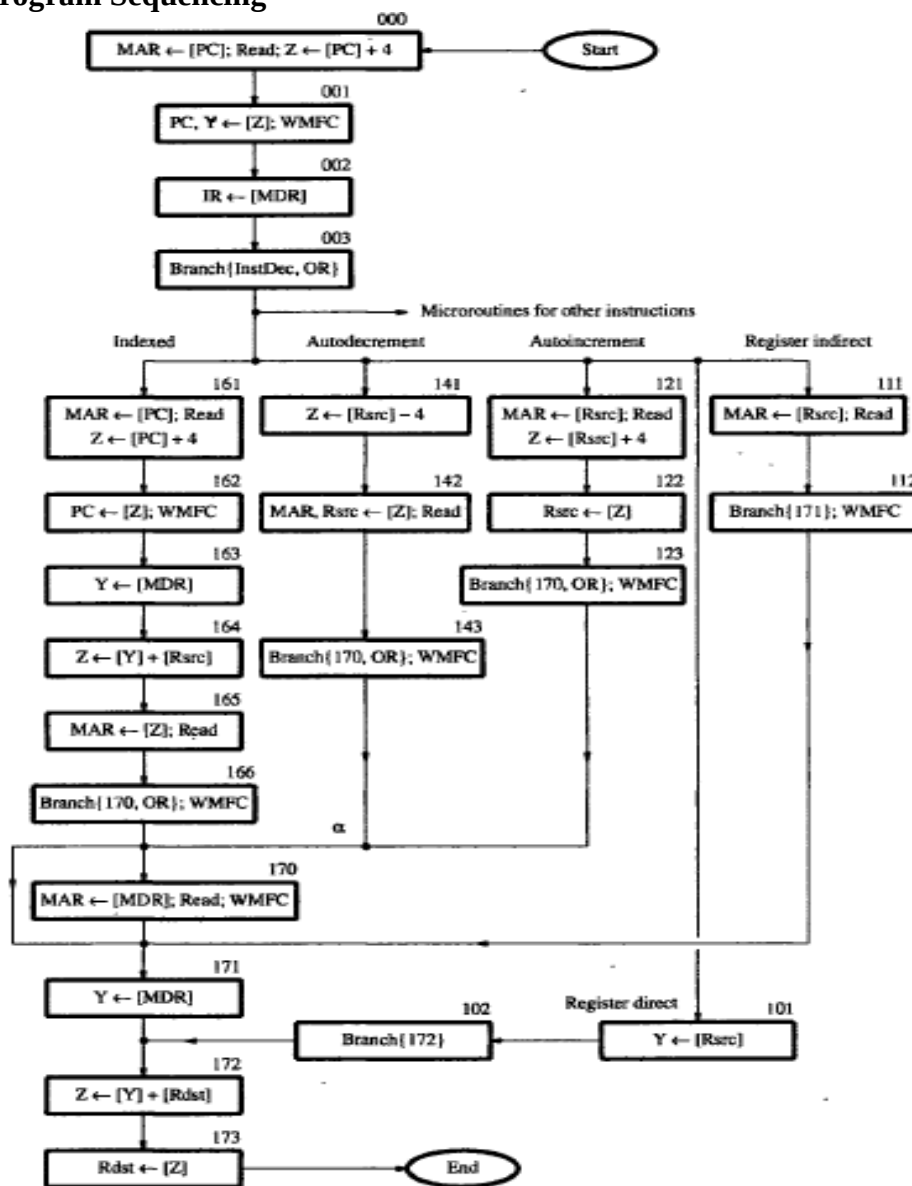


Figure 7.20 Flowchart of a microprogram for the Add src, Rdst instruction.

- Two major disadvantage of microprogrammed control is:
 - 1) Having a separate microroutine for each machine instruction results in a large total number of microinstructions and a large control-store.
 - 2) Execution time is longer because it takes more time to carry out the required branches.
- Consider the instruction *Add src, Rdst* ;which adds the source-operand to the contents of Rdst and places the sum in Rdst.

- Let source-operand can be specified in following addressing modes: register, autoincrement, autodecrement and indexed as well as the indirect forms of these 4 modes.
- Each box in the chart corresponds to a microinstruction that controls the transfers and operations indicated within the box.
- The microinstruction is located at the address indicated by the octal number (001,002).

Branch Address Modification Using Bit- ORing

Consider the point labeled in the figure. At this point, it is necessary to choose between direct and indirect addressing modes.

- If indirect-mode is specified in the instruction, then the microinstruction in location 170 is performed to fetch the operand from the memory.
If direct-mode is specified, this fetch must be bypassed by branching immediately to location 171.
- The most efficient way to bypass microinstruction 170 is to have the preceding branch microinstructions specify the address 170 and then use an OR gate to change the LSB of this address to 1 if the direct addressing mode is involved. This is known as the *bit-ORing* technique.

3) Wide Branch Addressing

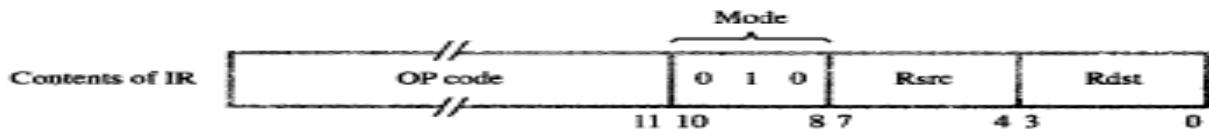
- The instruction-decoder(InstDec) generates the starting-address of the microinstruction that implements the instruction that has just been loaded into the IR.
- Here, register IR contains the Add instruction, for which the instruction decoder generates the microinstruction address 101. (However, this address cannot be loaded as is into the μ PC).
- The source-operand can be specified in any of several addressing-modes. The bit-ORing technique can be used to modify the starting-address generated by the instruction-decoder to reach the appropriate path.

Use of WMFC

- WMFC signal is issued at location 112 which causes a branch to the microinstruction in location 171.
- WMFC signal means that the microinstruction may take several clock cycles to complete. If the branch is allowed to happen in the first clock cycle, the microinstruction at location 171 would be fetched and executed prematurely. To avoid this problem, WMFC signal must inhibit any change in the contents of the μ PC during the waiting-period.

Detailed Examination

- Consider *Add (Rsrc)+, Rdst*; which adds Rsrc content to Rdst content, then stores the sum in Rdst and finally increments Rsrc by 4 (i.e. auto-increment mode).
- In bit 10 and 9, bit-patterns 11, 10, 01 and 00 denote indexed, auto-decrement, auto-increment and register modes respectively. For each of these modes, bit 8 is used to specify the indirect version.
- The processor has 16 registers that can be used for addressing purposes; each specified using a 4-bit-code.
- There are 2 stages of decoding:
 - 1) The microinstruction field must be decoded to determine that an Rsrc or Rdst

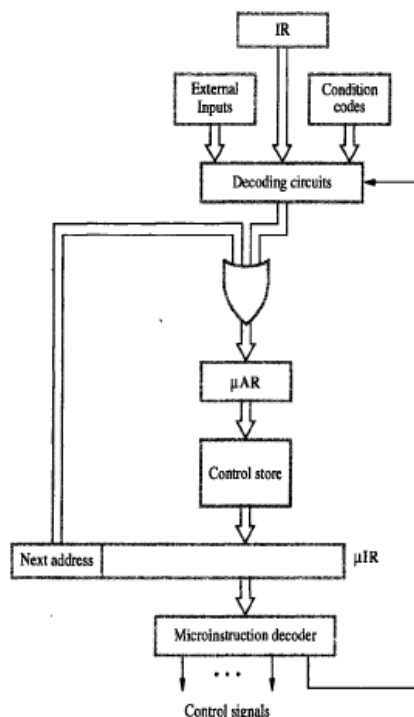


Address (octal)	Microinstruction
000	$PC_{out} \leftarrow MAR_{in}$, Read, Select4, Add, Z_{in}
001	$Z_{out} \leftarrow PC_{in}$, Y_{in} , WMFC
002	$MDR_{out} \leftarrow IR_{in}$
003	μ Branch [$\mu PC \leftarrow 101$ (from Instruction decoder); $\mu PC_{5,4} \leftarrow [IR_{10,9}]$; $\mu PC_3 \leftarrow [\overline{IR_{10}}] \cdot [\overline{IR_9}] \cdot [IR_8]$]
121	$Rsrc_{out} \leftarrow MAR_{in}$, Read, Select4, Add, Z_{in}
122	$Z_{out} \leftarrow Rsrc_{in}$
123	μ Branch [$\mu PC \leftarrow 170$; $\mu PC_0 \leftarrow [\overline{IR_8}]$], WMFC
170	$MDR_{out} \leftarrow MAR_{in}$, Read, WMFC
171	$MDR_{out} \leftarrow Y_{in}$
172	$Rdst_{out} \leftarrow SelectY$, Add, Z_{in}
173	$Z_{out} \leftarrow Rdst_{in}$, End

Figure 7.21 Microinstruction for Add [Rsrc]+,Rdst.
 register is involved.

The decoded output is then used to gate the contents of the Rsrc or Rdst fields in the IR into a second decoder, which produces the gating-signals for the actual registers R0 to R15

4) MICROINSTRUCTIONS WITH NEXT-ADDRESS FIELDS



Microinstruction

F0	F1	F2	F3
----	----	----	----

F0 (8 bits)	F1 (3 bits)	F2 (3 bits)	F3 (3 bits)
Address of next microinstruction	000: No transfer 001: PC_{out} 010: MDR_{out} 011: Z_{out} 100: $Rsrc_{out}$ 101: $Rdst_{out}$ 110: $TEMP_{out}$	000: No transfer 001: PC_{in} 010: IR_{in} 011: Z_{in} 100: $Rsrc_{in}$ 101: $Rdst_{in}$	000: No transfer 001: MAR_{in} 010: MDR_{in} 011: $TEMP_{in}$ 100: Y_{in}

F4	F5	F6	F7
----	----	----	----

F4 (4 bits)	F5 (2 bits)	F6 (1 bit)	F7 (1 bit)
0000: Add 0001: Sub ...	00: No action 01: Read 10: Write	0: SelectY 1: Select4	0: No action 1: WMFC
1111: XOR			

F8	F9	F10
----	----	-----

F8 (1 bit)	F9 (1 bit)	F10 (1 bit)
------------	------------	-------------

Octal address	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
000	000000001	001	011	001	0000	01	1	0	0	0	0
001	000000010	011	001	100	0000	00	0	1	0	0	0
002	00000011	010	010	000	0000	00	0	0	0	0	0
003	00000000	000	000	000	0000	00	0	0	1	1	0
121	01010010	100	011	001	0000	01	1	0	0	0	0
122	01111000	011	100	000	0000	00	0	1	0	0	1
170	01111001	010	000	001	0000	01	0	1	0	0	0
171	01111010	010	000	100	0000	00	0	0	0	0	0
172	01111011	101	011	000	0000	00	0	0	0	0	0
173	00000000	011	101	000	0000	00	0	0	0	0	0

Figure 7.24 Implementation of the microroutine of Figure 7.21 using a next-microinstruction address field. (See Figure 7.23 for encoded signals.)

- The microprogram requires several branch microinstructions which perform no useful operation. Thus, they detract from the operating speed of the computer.
- Solution: Include an address-field as a part of every microinstruction to indicate the location of the next microinstruction to be fetched. (This means every microinstruction becomes a branch microinstruction).
- The flexibility of this approach comes at the expense of additional bits for the address-field.
- Advantage: Separate branch microinstructions are virtually eliminated. There are few limitations in assigning addresses to microinstructions. There is no need for a counter to keep track of sequential addresses. Hence, the μ PC is replaced with a μ AR (Microinstruction Address Register). {which is loaded from the next-address field in each microinstruction}.
- The next-address bits are fed through the OR gate to the μ AR, so that the address can be modified on the basis of the data in the IR, external inputs and condition-codes.
- The decoding circuits generate the starting-address of a given microroutine on the basis of the opcode in the IR.

5) PREFETCHING MICROINSTRUCTIONS

- Drawback of microprogrammed control: Slower operating speed because of the time it takes to fetch microinstructions from the control-store.
- Solution: Faster operation is achieved if the next microinstruction is pre-fetched while the current one is being executed.

6) Emulation

- The main function of microprogrammed control is to provide a means for simple, flexible and relatively inexpensive execution of machine instruction.
- Its flexibility in using a machine's resources allows diverse classes of instructions to be implemented.
- Suppose we add to the instruction-repository of a given computer M1, an entirely new set of instructions that is in fact the instruction-set of a different computer M2.
- Programs written in the machine language of M2 can then be run on computer M1 i.e. M1 emulates M2.
- Emulation allows us to replace obsolete equipment with more up-to-date machines.
- If the replacement computer fully emulates the original one, then no software changes have to be made to run existing programs.
- Emulation is easiest when the machines involved have similar architectures.

➤ **Pipelining**

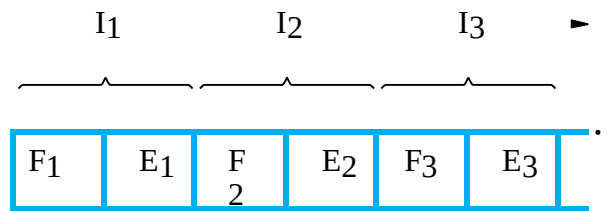
➤ **BASIC CONCEPTS**

The speed of execution of programs is influenced by many factors. One way to improve performance is to use faster circuit technology to build the processor and the main memory. Another possibility is to arrange the hardware so that more than one operation can be performed at the same time. In this way, the number of operations performed per second is increased even though the elapsed time needed to perform any one operation is not changed.

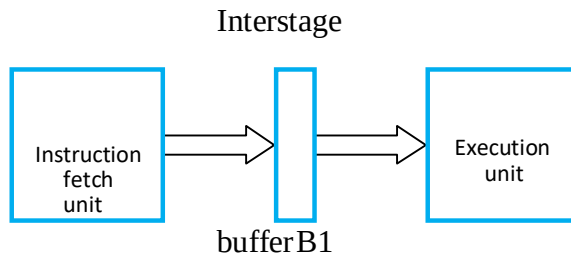
Pipelining is a particularly effective way of organizing concurrent activity in a computer system. The basic idea is very simple. It is frequently encountered in manufacturing plants, where pipelining is commonly known as an assembly-line operation. Readers are undoubtedly familiar with the assembly line used in car manufacturing. The first station in an assembly line may prepare the chassis of a car, the next station adds the body, the next one installs the engine, and so on. While one group of workers is installing the engine on one car, another group is fitting a car body on the chassis of another car, and yet another group is preparing a new chassis for a third car. It may take days to complete work on a given car, but it is possible to have a new car rolling off the end of the assembly line every few minutes.

The processor executes a program by fetching and executing instructions, one after the other. Let F_i and E_i refer to the fetch and execute steps for instruction I_i . Execution of a program consists of a sequence of fetch and execute steps, as shown in Figure 8.1a.

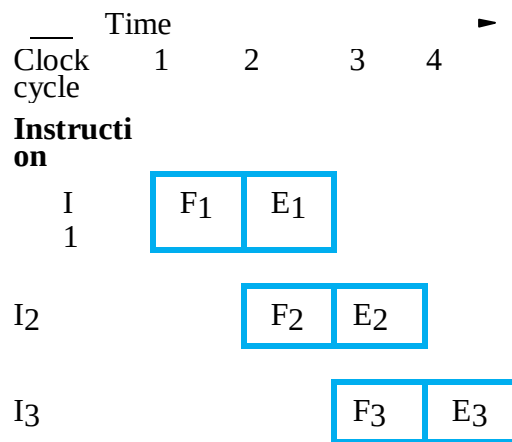
Now consider a computer that has two separate hardware units, one for fetching instructions and another for executing them, as shown in Figure 8.1b. The instruction fetched by the fetch unit is deposited in an intermediate storage buffer, B1. This buffer is needed to enable the execution unit to execute the instruction while the fetch unit is fetching the next instruction. The results of execution are deposited in the destination location specified by the instruction. For the purposes of this discussion, we assume that both the source and the destination of the data operated on by the instructions are inside the block labeled “Execution unit.”



(a) Sequential execution



(b) Hardware organization



(c) Pipelined execution

Figure 8.1 Basic idea of instruction pipelining.

The computer is controlled by a clock whose period is such that the fetch and execute steps of any instruction can each be completed in one clock cycle. Operation of the computer proceeds as in Figure 8.1c. In the first clock cycle, the fetch unit fetches an instruction I_1 (step F_1) and stores it in buffer B1 at the end of the clock cycle. In the second clock cycle, the instruction fetch unit proceeds with the fetch operation for instruction I_2 (step F_2). Meanwhile, the execution unit performs the operation specified by instruction I_1 , which is available to it in buffer B1 (step E_1). By the end

of the

second clock cycle, the execution of instruction I_1 is completed and instruction I_2 is available. Instruction I_2 is stored in B1, replacing I_1 , which is no longer needed. Step E_2 is performed by the execution unit during the third clock cycle, while instruction I_3 is being fetched by the fetch unit. In this manner, both the fetch and execute units are kept busy all the time. If the pattern in Figure 8.1c can be sustained for a long time, the completion rate of instruction execution will be twice that achievable by the sequential operation depicted in Figure 8.1a.

In summary, the fetch and execute units in Figure 8.1b constitute a two-stage pipeline in which each stage performs one step in processing an instruction. An inter-stage storage buffer, B1, is needed to hold the information being passed from one stage to the next. New information is loaded into this buffer at the end of each clock cycle.

The processing of an instruction need not be divided into only two steps. For example, a pipelined processor may process each instruction in four steps, as follows:

- F Fetch: read the instruction from the memory.
- D Decode: decode the instruction and fetch the source operand(s).
- E Execute: perform the operation specified by the instruction.
- W Write: store the result in the destination location.

The sequence of events for this case is shown in Figure 8.2a. Four instructions are in progress at any given time. This means that four distinct hardware units are needed, as shown in Figure 8.2b. These units must be capable of performing their tasks simultaneously and without interfering with one another. Information is passed from one unit to the next through a storage buffer. As an instruction progresses through the pipeline, all the information needed by the stages downstream must be passed along. For example, during clock cycle 4, the information in the buffers is as follows:

- Buffer B1 holds instruction I_3 , which was fetched in cycle 3 and is being decoded by the instruction-decoding unit.
- Buffer B2 holds both the source operands for instruction I_2 and the specification of the operation to be performed. This is the information produced by the decoding hardware in cycle 3. The buffer also holds the information needed for the write step of instruction I_2 (step W_2). Even though it is not needed by stage E, this information must be passed on to stage W in the following clock cycle to enable that stage to perform the required Write operation.
- Buffer B3 holds the results produced by the execution unit and the destination information for instruction I_1 .

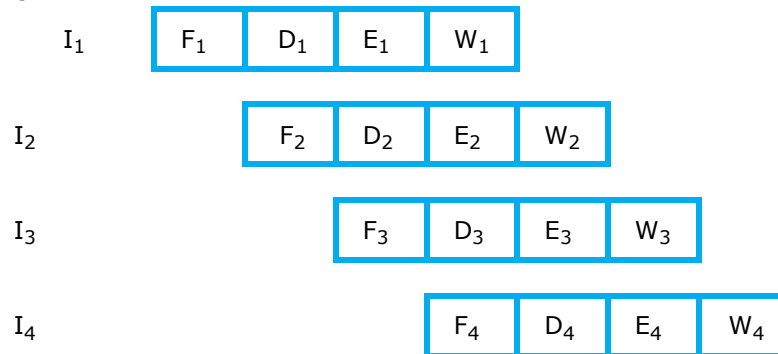
1)ROLE OF CACHE MEMORY

Each stage in a pipeline is expected to complete its operation in one clock

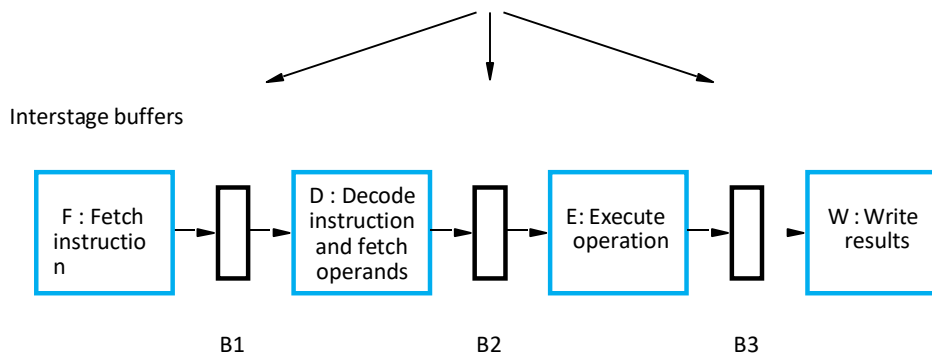
cycle. Hence, the clock period should be sufficiently long to complete the task being performed in any stage. If different units require different amounts of time, the clock period must allow the longest task to be completed. A unit that completes its task early is idle for the remainder of the clock period. Hence, pipelining is most effective in improving



Instruction



(a) Instruction execution divided into four steps



(b) Hardware organization

Figure 8.2 A 4-stage pipeline

performance if the tasks being performed in different stages require about the same amount of time.

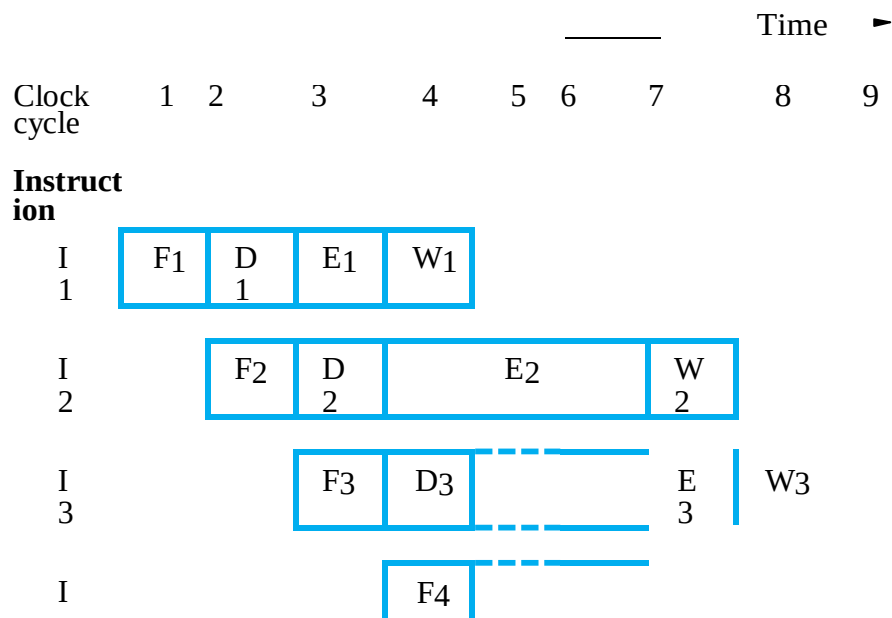
This consideration is particularly important for the instruction fetch step, which is assigned one clock period in Figure 8.2a. The clock cycle has to be equal to or greater than the time needed to complete a fetch operation. However, the access time of the main memory may be as much as ten times greater than the time needed to perform basic pipeline stage operations inside the processor, such as adding two numbers. Thus, if each

instruction fetch required access to the main memory, pipelining would be of little value.

The use of cache memories solves the memory access problem. In particular, when a cache is included on the same chip as the processor, access time to the cache is usually the same as the time needed to perform other basic operations inside the processor.

2) PIPELINE PERFORMANCE

The pipelined processor in Figure 8.2 completes the processing of one instruction in each clock cycle, which means that the rate of instruction processing is four times that of sequential operation. The potential increase in performance resulting from pipelining is proportional to the number of pipeline stages. However, this increase would be achieved only if pipelined operation as depicted in Figure 8.2a could be sustained without interruption throughout program execution. Unfortunately, this is not the case. For a variety of reasons, one of the pipeline stages may not be able to complete its processing task for a given instruction in the time allotted. For example, stage E in the four-stage pipeline of Figure 8.2b is responsible for arithmetic and logic operations, and one clock cycle is assigned for this task. Although this may be sufficient for most operations, some operations, such as divide, may require more time to complete. Figure 8.3 shows an example in which the operation specified in instruction I_2 requires three cycles to complete, from cycle 4 through cycle 6. Thus, in cycles 5 and 6, the Write stage must be told to do nothing, because it has no data to work with. Meanwhile, the information in buffer B2 must remain intact until the Execute stage has completed its operation. This means that stage 2 and, in turn, stage 1 are blocked from accepting new instructions because the information in B1 cannot be overwritten. Thus, steps D_4 and F_5 must be postponed as shown.



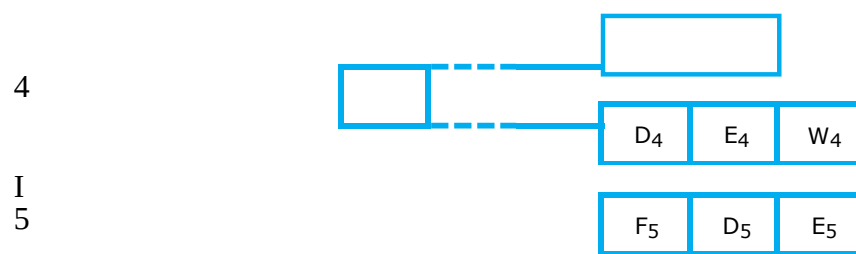
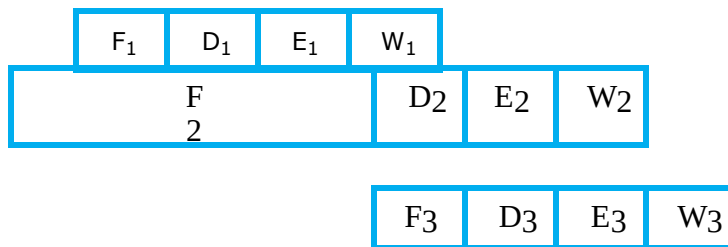


Figure 8.3 Effect of an execution operation taking more than one clock cycle.

Pipelined operation in Figure 8.3 is said to have been *stalled* for two clock cycles. Any condition that causes the pipeline to stall is called a *hazard*. In any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline. As a result some operation has to be delayed, and the pipeline stalls.

The pipeline may also be stalled because of a delay in the availability of an instruction. For example, this may be a result of a miss in the cache, requiring the instruction to be fetched from the main memory. Such hazards are often called *control hazards* or *instruction hazards*. The effect of a cache miss on pipelined operation is illustrated in Figure 8.4. Instruction I₁ is fetched from the cache in cycle 1, and its execution proceeds normally. However, the fetch operation for instruction I₂, which is started in cycle 2, results in a cache miss. The instruction fetch unit must now suspend any further fetch requests and wait for I₂ to arrive. We assume that instruction I₂ is received and loaded into buffer B1 at the end of cycle 5. The pipeline resumes its normal operation at that point.



(a) Instruction execution steps in successive clock cycles

Clock cycle	1	2	3	4	5	6	7	8	9
Stage	F1	F2	F2	F2	F2	F3			
F: Fetch									
D: Decode		D1	idle	idle	idle	D2	D3		
E: Execute			E1	idle	idle	idle	E2	E3	
W: Write				W1	idle	idle	idle	W2	W3

➤ DATA HAZARDS

A data hazard is a situation in which the pipeline is stalled because the data to be operated on are delayed for some reason, as illustrated in Figure 8.3. We will now examine the issue of availability of data in some detail.

Consider a program that contains two instructions, I_1 followed by I_2 . When this program is executed in a pipeline, the execution of I_2 can begin before the execution of I_1 is completed. This means that the results generated by I_1 may not be available for use by I_2 . We must ensure that the results obtained when instructions are executed in a pipelined processor are identical to those obtained when the same instructions are executed sequentially. The potential for obtaining incorrect results when operations are performed concurrently can be demonstrated by a simple example. Assume that $A = 5$, and consider the following two operations:

$$\begin{aligned} A &\leftarrow 3 + \\ AB &\leftarrow 4 \\ &\times A \end{aligned}$$

When these operations are performed in the order given, the result is $B = 32$. But if they are performed concurrently, the value of A used in computing B would be the original value, 5, leading to an incorrect result. If these two operations are performed by instructions in a program, then the instructions must be executed one after the other, because the data used in the second instruction depend on the result of the first instruction. On the other hand, the two operations

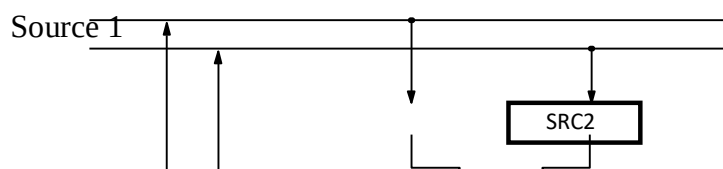
$$\begin{aligned} A &\leftarrow 5 \times C \\ B &\leftarrow 20 + \\ &C \end{aligned}$$

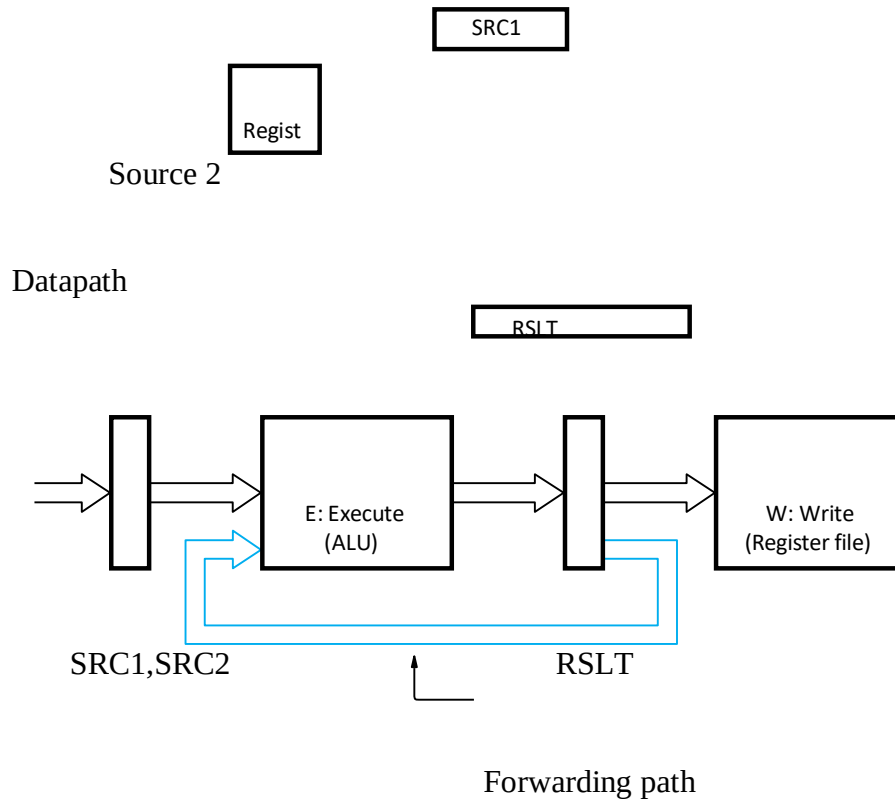
1) OPERAND FORWARDING

The data hazard just described arises because one instruction, instruction I_2 in Figure 8.6, is waiting for data to be written in the register file. However, these data are available at the output of the ALU once the Execute stage completes step E_1 . Hence, the delay can

be reduced, or possibly eliminated, if we arrange for the result of instruction I_1 to be forwarded directly for use in step E_2 .

Figure 8.7a shows a part of the processor datapath involving the ALU and the register file. This arrangement is similar to the three-bus structure in Figure 7.8, except that registers SRC1, SRC2, and RSLT have been added. These registers constitute the





(a) Position of the source and result registers in the processor pipeline

Figure 8.7 Operand forwarding in a pipelined processor.

2) HANDLING DATA HAZARDS IN SOFTWARE

In Figure 8.6, we assumed the data dependency is discovered by the hardware while the instruction is being decoded. The control hardware delays reading register R4 until cycle 5, thus introducing a 2-cycle stall unless operand forwarding is used. An alternative approach is to leave the task of detecting data dependencies and dealing with them to the software. In this case, the compiler can introduce the two-cycle delay needed between instructions I_1 and I_2 by inserting NOP (No-operation) instructions, as follows:

```

I1:  Mul
      R2,R3,R
      4NOP
      NOP
I2:  Add  R5,R4,R6

```

If the responsibility for detecting such dependencies is left entirely to the software, the compiler must insert the NOP instructions to obtain a correct result. This possibility illustrates the close link between the compiler and the hardware. A particular feature can be either implemented in hardware or left to the compiler. Leaving tasks such as inserting NOP instructions to

the compiler leads to simpler hardware. Being aware of the need for a delay, the compiler can attempt to reorder instructions to perform useful tasks in the NOP slots, and thus achieve better performance. On the other hand, the insertion of NOP instructions leads to larger code size. Also, it is often the case that a given processor architecture has several hardware implementations, offering different features. NOP instructions inserted to satisfy the requirements of one implementation may not be needed and, hence, would lead to reduced performance on a different implementation.

3)SIDE EFFECTS

The data dependencies encountered in the preceding examples are explicit and easily detected because the register involved is named as the destination in instruction I_1 and as a source in I_2 . Sometimes an instruction changes the contents of a register other than the one named as the destination. An instruction that uses an autoincrement or autodecrement addressing mode is an example. In addition to storing new data in its destination location, the instruction changes the contents of a source register used to access one of its operands. All the precautions needed to handle data dependencies involving the destination location must also be applied to the registers affected by an autoincrement or autodecrement operation. When a location other than one explicitly named in an instruction as a destination operand is affected, the instruction is said to have a *side effect*. For example, stack instructions, such as push and pop, produce similar side effects because they implicitly use the autoincrement and autodecrement addressing modes.

Another possible side effect involves the condition code flags, which are used by instructions such as conditional branches and add-with-carry. Suppose that registers R1 and R2 hold a double-precision integer number that we wish to add to another double-precision number in registers R3 and R4. This may be accomplished as follows:

Add R1,R3
AddWithCarry R2,R4

An implicit dependency exists between these two instructions through the carry flag. This flag is set by the first instruction and used in the second instruction, which performs the operation

$$R4 \leftarrow [R2] + [R4] + \text{carry}$$

Instructions that have side effects give rise to multiple data dependencies, which lead to a substantial increase in the complexity of the hardware or software needed to resolve them. For this reason, instructions designed for execution on pipelined hardware should have few side effects. Ideally, only the contents of the destination location, either a register or a memory location, should be affected by any given instruction. Side effects, such as setting the condition code flags or updating the contents of an address pointer, should be kept to a minimum. However, Chapter 2 showed that the autoincrement and autodecrement addressing modes are potentially

useful. Condition code flags are also needed for recording such information as the generation of a carry or the occurrence of overflow in an arithmetic operation. In Section 8.4 we show how such functions can be provided by other means that are consistent with a pipelined organization and with the requirements of optimizing compilers.

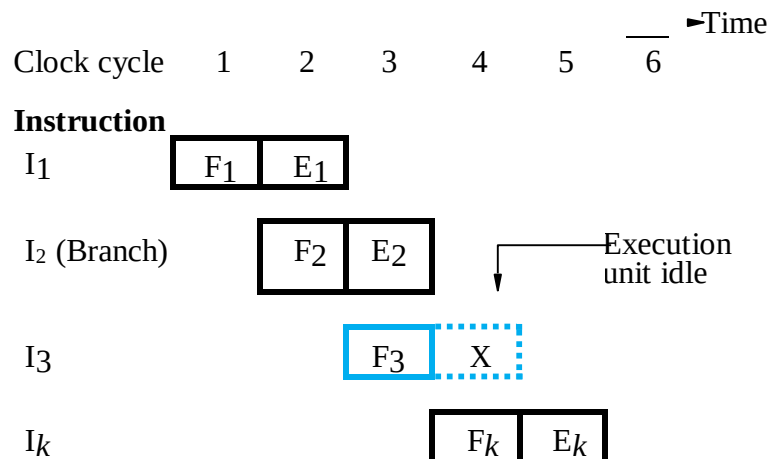
➤ INSTRUCTION HAZARDS

The purpose of the instruction fetch unit is to supply the execution units with a steady stream of instructions. Whenever this stream is interrupted, the pipeline stalls, as Figure 8.4 illustrates for the case of a cache miss. A branch instruction may also cause the pipeline to stall. We will now examine the effect of branch instructions and the techniques that can be used for mitigating their impact. We start with unconditional branches.

1) UNCONDITIONAL BRANCHES

Figure 8.8 shows a sequence of instructions being executed in a two-stage pipeline. Instructions I_1 to I_3 are stored at successive memory addresses, and I_2 is a branch instruction. Let the branch target be instruction I_k . In clock cycle 3, the fetch operation for instruction I_3 is in progress at the same time that the branch instruction is being decoded and the target address computed. In clock cycle 4, the processor must discard I_3 , which has been incorrectly fetched, and fetch instruction I_k . In the meantime, the hardware unit responsible for the Execute (E) step must be told to do nothing during that clock period. Thus, the pipeline is stalled for one clock cycle.

The time lost as a result of a branch instruction is often referred to as the *branch penalty*. In Figure 8.8, the branch penalty is one clock cycle. For a longer pipeline, the branch penalty may be higher. For example, Figure 8.9a shows the effect of a branch instruction on a four-stage pipeline. We have assumed that the branch address is computed in step E_2 . Instructions I_3 and I_4 must be discarded, and the target instruction, I_k , is fetched in clock cycle 5. Thus, the branch penalty is two clock cycles.



I_{k+1}

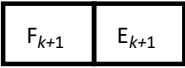


Figure 8.8 An idle cycle caused by a branch instruction.

unit, which we call the *dispatch unit*, takes instructions from the front of the queue and

Instruction fetch unit

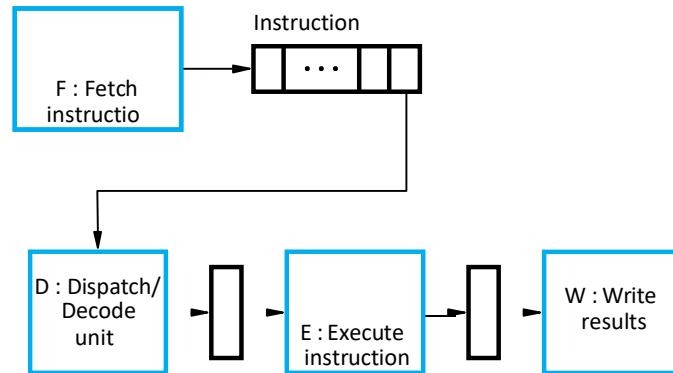


Figure 8.10 Use of an instruction queue in the hardware organization of Figure 8.2b.

To be effective, the fetch unit must have sufficient decoding and processing capability to recognize and execute branch instructions. It attempts to keep the instruction queue filled at all times to reduce the impact of occasional delays when fetching instructions. When the pipeline stalls because of a data hazard, for example, the dispatch unit is not able to issue instructions from the instruction queue. However, the fetch unit continues to fetch instructions and add them to the queue. Conversely, if there is a delay in fetching instructions because of a branch or a cache miss, the dispatch unit continues to issue instructions from the instruction queue.

Instruction I_5 is a branch instruction. Its target instruction, I_k , is fetched in cycle 7, and instruction I_6 is discarded. The branch instruction would normally cause a stall in cycle 7 as a result of discarding instruction I_6 . Instead, instruction I_4 is dispatched from the queue to the decoding stage. After discarding I_6 , the queue length drops to 1 in cycle 8. The queue length will be at this value until another stall is encountered.

Now observe the sequence of instruction completions in Figure 8.11. Instructions I_1 , I_2 , I_3 , I_4 , and I_k complete execution in successive clock cycles. Hence, the branch instruction does not increase the overall execution time. This is because the instruction fetch unit has executed the branch instruction (by computing the branch address) concurrently with the execution of other instructions. This technique is referred to as *branch folding*.

Having an instruction queue is also beneficial in dealing with cache misses. When a cache miss occurs, the dispatch unit continues to send instructions for execution as long as the instruction queue is not empty. Meanwhile, the desired cache block is read from the main memory or from a secondary cache. When fetch operations are resumed, the instruction queue is refilled. If the queue does not become empty, a cache miss will have no effect on the rate of instruction execution.

2)CONDITIONAL BRANCHES AND BRANCH PREDICTION

A conditional branch instruction introduces the added hazard caused by the dependency of the branch condition on the result of a preceding instruction. The decision to branch cannot be made until the execution of that instruction has been completed.

Branch instructions occur frequently. In fact, they represent about 20 percent of the dynamic instruction count of most programs. (The dynamic count is the number of instruction executions, taking into account the fact that some program instructions are executed many times because of loops.) Because of the branch penalty, this large percentage would reduce the gain in performance expected from pipelining. Fortunately, branch instructions can be handled in several ways to reduce their negative impact on the rate of execution of instructions.

Delayed Branch

In Figure 8.8, the processor fetches instruction I_3 before it determines whether the current instruction, I_2 , is a branch instruction. When execution of I_2 is completed and a branch is to be made, the processor must discard I_3 and fetch the instruction at the branch target. The location following a branch instruction is called a *branch delay slot*. There may be more than one branch delay slot, depending on the time it takes to execute a branch instruction. For example, there are two branch delay slots in Figure 8.9a and one delay slot in Figure 8.9b. The instructions in the delay slots are always fetched and at least partially executed before the branch decision is made and the branch target address is computed.

A technique called *delayed branching* can minimize the penalty incurred as a result of conditional branch instructions. The idea is simple. The instructions in the delay slots are always fetched. Therefore, we would like to arrange for them to be fully executed whether or not the branch is taken. The objective is to be able to place useful instructions in these slots. If no useful instructions can be placed in the delay slots, these slots must be filled with NOP instructions. This situation is exactly the same as in the case of data dependency discussed in Section 8.2.

Consider the instruction sequence given in Figure 8.12a. Register R2 is used as a counter to determine the number of times the contents of register R1 are shifted left. For a processor with one delay slot, the

instructions can be reordered as shown in Figure 8.12b. The shift instruction is fetched while the branch instruction is being executed. After evaluating the branch condition, the processor fetches the instruction at LOOP or at NEXT, depending on whether the branch condition is true or false, respectively. In either case, it completes execution of the shift instruction. The sequence of events during the last two passes in the loop is illustrated in Figure 8.13. Pipelined operation is not interrupted at any time, and there are no idle cycles. Logically, the program is executed as if the branch instruction were placed after the shift instruction. That is, branching takes place one instruction later than where the branch instruction appears in the instruction sequence in the memory, hence the name “delayed branch.”

LOOP	Shift_left	R1
	Decrement	R2
	Branch=0	LOOP
NEXT	Add	R1,R3

(a) Original program loop

LOOP	Decrement	R2
	Branch	LOOP
NEXT	=0	OP
	Shift_l	R1
	eftAdd	R1,R3

(b) Reordered instructions

Figure 8.12 Reordering of instructions for a delayed branch.

Branch Prediction

Another technique for reducing the branch penalty associated with conditional branches is to attempt to predict whether or not a particular branch will be taken. The simplest form of branch prediction is to assume that the branch will not take place and to continue to fetch instructions in sequential address order. Until the branch condition is evaluated, instruction execution along the predicted path must be done on a

speculative basis. *Speculative execution* means that instructions are executed before the processor is certain that they are in the correct execution sequence. Hence, care must be taken that no processor registers or memory locations are updated until it is confirmed that these instructions should indeed be executed. If the branch decision indicates otherwise, the instructions and all their associated data in the execution units must be purged, and the correct instructions fetched and executed.

An incorrectly predicted branch is illustrated in Figure 8.14 for a four-stage pipeline. The figure shows a Compare instruction followed by a Branch>0 instruction. Branch

Dynamic Branch Prediction

The objective of branch prediction algorithms is to reduce the probability of making a wrong decision, to avoid fetching instructions that eventually have to be discarded. In dynamic branch prediction schemes, the processor hardware assesses the likelihood of a given branch being taken by keeping track of branch decisions every time that instruction is executed.

In its simplest form, the execution history used in predicting the outcome of a given branch instruction is the result of the most recent execution of that instruction. The processor assumes that the next time the instruction is executed, the result is likely to be the same. Hence, the algorithm may be described by the two-state machine in Figure 8.15a. The two states are:

LT: Branch is likely to be taken

LNT: Branch is likely not to be
taken

Suppose that the algorithm is started in state LNT. When the branch instruction is

➤ INFLUENCE ON INSTRUCTION SETS

We have seen that some instructions are much better suited to pipelined execution than others. For example, instruction side effects can lead to undesirable data dependencies. In this section, we examine the relationship between pipelined execution and machine instruction features. We discuss two key aspects of machine instructions — addressing modes and condition code flags.

1) ADDRESSING MODES

Addressing modes should provide the means for accessing a variety of data structures simply and efficiently. Useful addressing modes include index,

indirect, autoincrement, and autodecrement. Many processors provide various combinations of these modes to increase the flexibility of their instruction sets. Complex addressing modes, such as those involving double indexing, are often encountered.

In choosing the addressing modes to be implemented in a pipelined processor, we must consider the effect of each addressing mode on instruction flow in the pipeline. Two important considerations in this regard are the side effects of modes such as autoincrement and autodecrement and the extent to which complex addressing modes cause the pipeline to stall. Another important factor is whether a given mode is likely to be used by compilers.

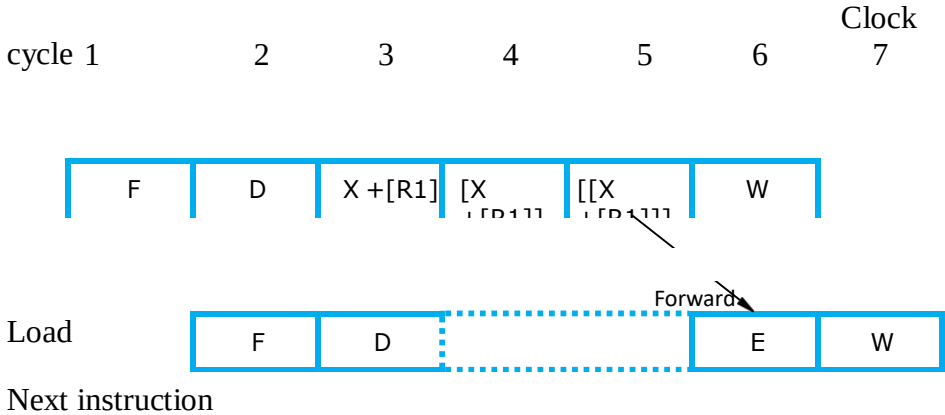
To compare various approaches, we assume a simple model for accessing operands in the memory. The load instruction Load X(R1),R2 takes five cycles to complete execution, as indicated in Figure 8.5. However, the instruction

Load (R1),R2

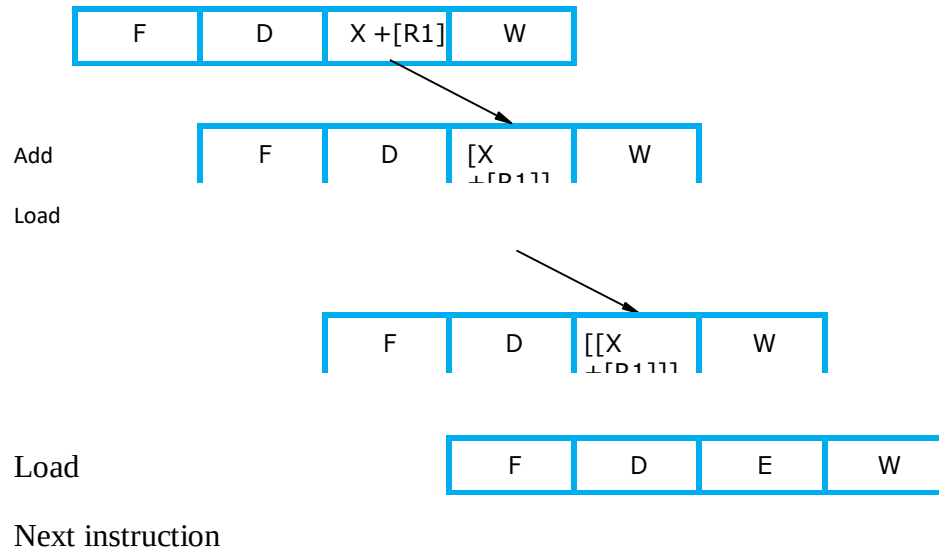
can be organized to fit a four-stage pipeline because no address computation is required. Access to memory can take place in stage E. A more complex addressing mode may require several accesses to the memory to reach the named operand. For example, the instruction

Load (X(R1)),R2

may be executed as shown in Figure 8.16a, assuming that the index offset, X, is given in the instruction word. After computing the address in cycle 3, the processor needs to access memory twice — first to read location X[R1] in clock cycle 4 and then to read location [X[R1]] in cycle 5. If R2 is a source operand in the next instruction, that instruction would be stalled for three cycles, which can be reduced to two cycles with operand forwarding, as shown.



(a) Complex addressing mode



(b) Simple addressing mode

Figure 8.16 Equivalent operations using complex and simple addressing modes.

2) CONDITION CODES

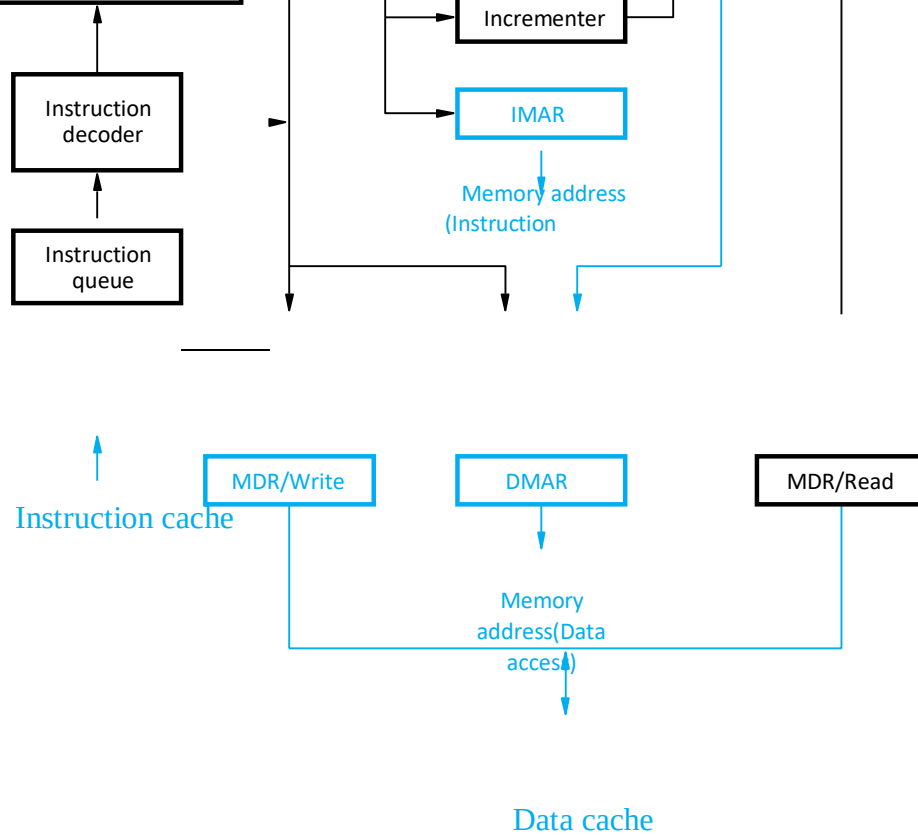
In many processors, such as those described in Chapter 3, the condition code flags are stored in the processor status register. They are either set or cleared by many instructions, so that they can be tested by subsequent conditional branch instructions to change the flow of program execution. An optimizing compiler for a pipelined processor attempts to reorder instructions to avoid stalling the pipeline when branches or data dependencies between successive instructions occur. In doing so, the compiler must ensure that reordering does not cause a change in the outcome of a computation. The dependency introduced by the condition-code flags reduces the flexibility available for the compiler to reorder instructions.

Consider the sequence of instructions in Figure 8.17a, and assume that the execution of the Compare and Branch 0 instructions proceeds as in Figure 8.14. The branch decision takes place in step E_2 rather than D_2 because it must await the result of the Compare instruction. The execution time of the Branch instruction can be reduced

by interchanging the Add and Compare instructions, as shown in Figure 8.17b. This will delay the branch instruction by one cycle relative to the Compare instruction. As a result, at the time the Branch instruction is being decoded the result of the Compare instruction will be available and a correct branch decision will be made. There would be no need for branch prediction. However, interchanging the Add and Compare instructions can be done only if the Add instruction does not affect the condition codes by interchanging the Add and Compare instructions, as shown in Figure 8.17b. This will delay the branch instruction by one cycle relative to the Compare instruction. As a result, at the time the Branch instruction is being decoded the result of the Compare instruction will be available and a correct branch decision will be made. There would be no need for branch prediction. However, interchanging the Add and Compare instructions can be done only if the Add instruction does not affect the condition codes.

➤ DATAPATH AND CONTROL CONSIDERATIONS

Organization of the internal datapath of a processor was introduced in Chapter 7. Consider the three-bus structure presented in Figure 7.8. To make it suitable for pipelined execution, it can be modified as shown in Figure 8.18 to support a 4-stage pipeline. The resources involved in stages F and E are shown in blue and those used in stages D and W in black. Operations in the data cache may happen during stage E or at a later stage, depending on the addressing mode and the implementation details. This section



Datapath modified for pipelined execution with interstage buffers at the input and output of the ALU.

is shown in blue. Several important changes to Figure 7.8 should be noted:

1. There are separate instruction and data caches that use separate address and data connections to the processor. This requires two versions of the MAR register, IMAR for accessing the instruction cache and DMAR for accessing the data cache.
2. The PC is connected directly to the IMAR, so that the contents of the PC can be transferred to IMAR at the same time that an independent ALU operation is taking place.
3. The data address in DMAR can be obtained directly from the register file or from the ALU to support the register indirect and indexed addressing modes.
4. Separate MDR registers are provided for read and write operations. Data can be transferred directly between these registers and the register file during load and store operations without the need to pass through the ALU.
5. Buffer registers have been introduced at the inputs and output of the ALU. These are registers SRC1, SRC2, and RSLT in Figure 8.7. Forwarding connections are not included in Figure 8.18. They may be added if desired.

6. The instruction register has been replaced with an instruction queue, which is loaded from the instruction cache.

7. The output of the instruction decoder is connected to the control signal pipeline. The need for buffering control signals and passing them from one stage to the next along with the instruction is discussed in Section 8.1. This pipeline holds the control signals in buffers B2 and B3 in Figure 8.2a.

The following operations can be performed independently in the processor of Figure 8.18:

- Reading an instruction from the instruction cache
- Incrementing the PC
- Decoding an instruction
- Reading from or writing into the data cache
- Reading the contents of up to two registers from the register file
- Writing into one register in the register file
- Performing an ALU operation

Because these operations do not use any shared resources, they can be performed simultaneously in any combination. The structure provides the flexibility required to implement the four-stage pipeline in Figure 8.2. For example, let I_1 , I_2 , I_3 , and I_4 be a sequence of four instructions. As shown in Figure 8.2a, the following actions all happen during clock cycle 4:

- Write the result of instruction I_1 into the register file
- Read the operands of instruction I_2 from the register file
- Decode instruction I_3
- Fetch instruction I_4 and increment the PC.

