# PRATHYUSHA
## ENGINEERING COLLEGE

**Poonamallee – Tiruvallur Road, Chennai – 602025.**

# CS8491
# Computer Architecture

**(Anna University – 2017 Regulation)**

**Prepared by**
**Ms.R.Kannamma,**
**AP in CSE, PEC**

# UNIT – I

**UNIT I BASIC STRUCTURE OF A COMPUTER SYSTEM 9**

Functional Units – Basic Operational Concepts – Performance – Instructions: Language of the Computer – Operations, Operands – Instruction representation – Logical operations –decision making – MIPS Addressing.

## COMPONENTS OF A COMPUTER SYSTEM

## BASIC STRUCTURE OF COMPUTER

### COMPUTER

A Computer is a machine which accepts input information in the digitized form, processes the input according to a set of stored instructions and gives an output in a form understandable by user.

### PROGRAM AND DATA

The set of stored instructions according which input are processed by computer are called programs and input and output information is called data.

### TYPES OF COMPUTER

Computers are classified according to size, cost, power of the processor, and type of usage. Personal computers are most common computers and Super computers are most powerful computers.

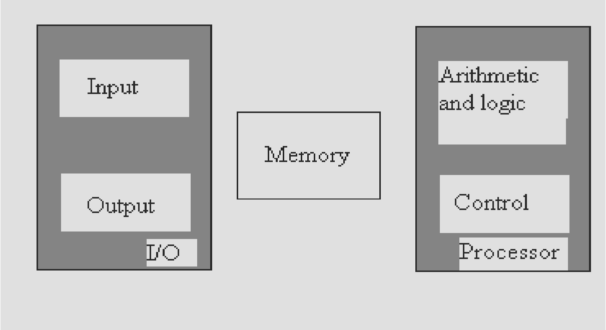
Some of types of computer are

- Personal computers
- Notebook computers
- Workstations
- Enterprise systems and Servers
- Mainframes
- Supercomputers

### I. HARDWARE PARTS OF THE COMPUTER SYSTEM:

A computer consists of five functionally independent main parts

1. Input
2. Output
3. ALU
4. Control Unit
5. Memory



**The operation of a computer can be summarized as follows**

- ✓ The computer accepts programs and the data through an input and stores them in the memory.
- ✓ The stored data are processed by the arithmetic and logic unit under program control.
- ✓ The processed data is delivered through the output unit.
- ✓ All above activities are directed by control unit

**Input unit**

- ➢ **The computer accepts coded information through input unit.** The input can be from human operators, electromechanical devices such as keyboards or from other computer over communication lines.
- ➢ Examples of input devices are
    - ✓ Keyboard, joysticks, trackballs and mouse are used as graphic input devices in conjunction with display.
    - ✓ Microphones can be used to capture audio input.

Keyboard

- It is a common input device.
- Whenever a key is pressed, the corresponding letter or digit is automatically translated into its corresponding binary code and transmitted over cable to the memory of the computer.

**Output unit**

- The function of output unit is to produce processed result to the outside world in human understandable form.
- Examples of output devices are Printer, Graphical display.

**Arithmetic and logic unit**

Arithmetic and logic unit (ALU) and control unit together form a processor.

Actual execution of most computer operations takes place in arithmetic and logic unit of the processor.

Example:

Suppose two numbers located in the memory are to be added. They are brought into the processor, and the actual addition is carried out by the ALU.

**Registers:**

- ➢ Registers are high speed storage elements available in the processor.
- ➢ Each register can store one word of data.
- ➢ When operands are brought into the processor for any operation, they are stored in the registers.
- ➢ Accessing data from register is faster than that of the memory.

**Control unit** Control unit coordinates the operation of memory, arithmetic and logic unit, input unit, and output unit. Control unit sends control signals to other units and senses their states.

**Central processor unit(CPU) :** Also called processor. The active part of the computer, which contains the datapath and control and which adds numbers, tests numbers, signals I/O devices to activate, and so on.

**Datapath:** The component of the processor that performs arithmetic operations.

**Memory unit**

The storage area in which programs are kept when they are running and that contains the data needed by the running programs. Memory is classified into primary and secondary storage.

**Primary storage**

- ➢ It also called main memory.
- ➢ It operates at high speed and it is expensive.
- ➢ It is Volatile memory (which loses its data when the power is turned off)
- ➢ It is directly accessed by CPU to store and retrieve information.
- ➢ It is made up of large number of semiconductor storage cells, each capable of storing one bit of information.
- ➢ Programs must reside in the primary memory during execution.
- ➢ RAM:
    - ✓ It stands for random access memory. Memory in which any location can be reached in a short and fixed amount of time by specifying its address is called random-access memory.
- ➢ Two types are static DRAM (SRAM) and Dynamic RAM(DRAM).
- ➢ The **memory** is where the programs are kept when they are running; it also contains the data needed by the running programs. The memory is built from DRAM chips. *DRAM* stands for **dynamic random access memory**.
- ➢ **static random access memory (SRAM)** Also memory built as an integrated circuit, but faster and less dense than DRAM
- ➢ **Cache Memory**

- ✓ They are small and fast RAM units.
- ✓ They are tightly coupled with the processor.
- ✓ They are often contained on the same integrated circuits(IC) chip to achieve high performance.

**Secondary memory**

- ➢ It is slow in speed.
- ➢ It is cheaper than primary memory.
- ➢ Its capacity is high.
- ➢ It is used to store information that is not accessed frequently.
- ➢ Various secondary devices are magnetic tapes and disks, optical disks (CD-ROMs), floppy etc.

## MEMORY ORGANIZATION

The memory is organized as collection of words. Each word is referred by an address 0 to M

**BIT (BINARY DIGIT)**:The memory contain large number of storage cells, each capable of storing 0 or 1 is called a **BIT**

**WORD:** cells are grouped together in a fixed size called word. This facilitates reading and writing the content of one word (n bits) in single basic operation instead of reading and writing one bit for each operation. Each word is associated with a distinct address that identifies word location. A given word is accessed by specifying its address.

**Wordlength** :The number of bits in each word is called as Wordlength

**Addresses:**These are the numbers that identified each location / word in memory

**Byte:** 8 Bits of memory is called **Byte**

**Byte Addressable:**

The memory is said to be byte addressable if every byte has separate address.Modern Processors use Byte addressing

Byte addressing within a word is represented in two ways

i. **Big Endian Addressing**

ii. **Little Endian Addressing**

**Big-endian Arrangement**

| Word address | Byte address | | | |
|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 |
| 4 | 4 | 5 | 6 | 7 |
| | | .<br>.<br>. | | |
| | | | | |
| $2^k$-4 | $2^k$-4 | $2^k$-3 | $2^k$-2 | $2^k$-1 |

**Little-endian Arrangement**

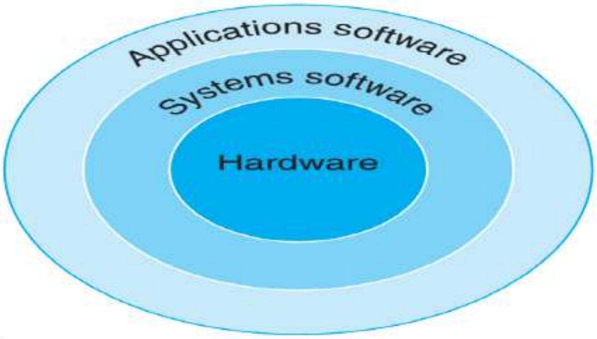
| Word address | Byte address | | | |
|---|---|---|---|---|
| 0 | 3 | 2 | 1 | 0 |
| 4 | 7 | 6 | 5 | 4 |
| | | .<br>.<br>. | | |
| $2^k$-4 | $2^k$-1 | $2^k$-2 | $2^k$-3 | $2^k$-4 |

II. **Software:**

Software can be divided into two categories. 1. System software and 2. Application software.

**System software:** System software are the program that runs and control the hardware units of the system. System software gets installed when the operating system is installed on the computer. It includes programs for compiler, linker and loader etc.

**Application software:** Application software runs over the system software. These software are installed according to the requirements of the user. Example: C, C++, Java, Ms-word etc.



**Compiler:** A Program that translates high-level language statements into assembly language statements. It takes entire program as input. The program need not be compiled everytime. The errors are displayed after the entire program is checked.

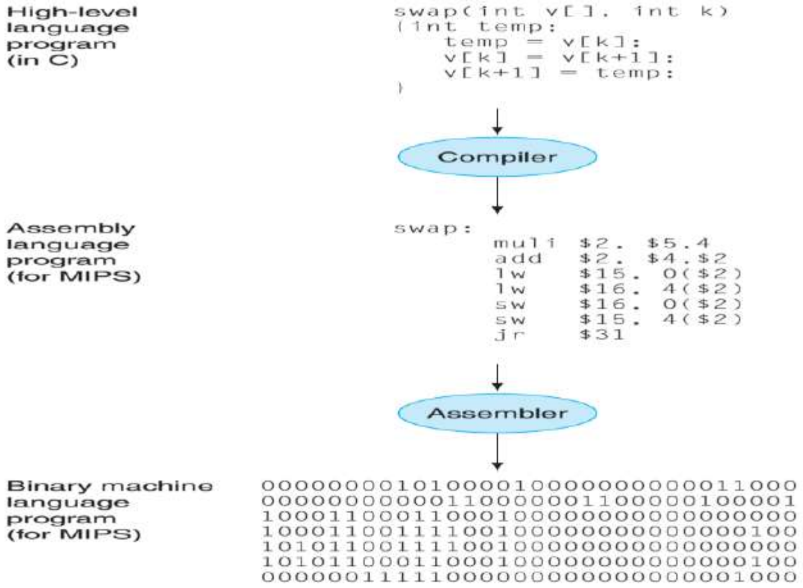
**Interpreter:** It takes single instruction as input. Everytime high level languages are converted in to Low level language. The errors are displayed after every instruction interpreted.

**Assembler:** A program that translates a symbolic version of instructions(Assembly language program) into the binary version.

**Assembly language:** A symbolic representation of machine instructions. Instructions arewritten using Mnemonics.

M**achine language:** A binary representation of machine instructions.

**High level Programming Language:** A portable language such as C, C++, Java or Visual Basicthat is composed of words and algebraic notation that can be translated by a compiler into assembly language.

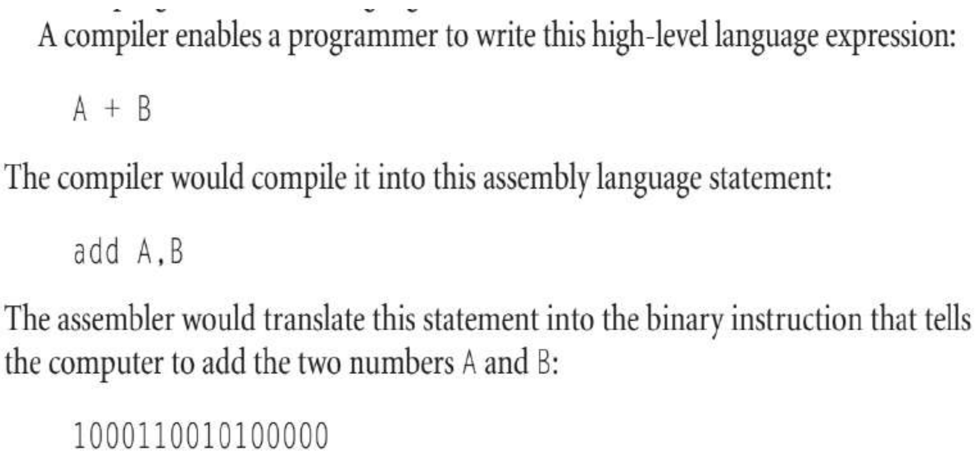

High-level language program (in C)
Compiler
Assembly language program (for MIPS)
Assembler
Binary machine language program (for MIPS)

```
swap(int v[], int k)
{int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

```
swap:
        muli $2, $5,4
        add  $2, $4,$2
        lw   $15, 0($2)
        lw   $16, 4($2)
        sw   $16, 0($2)
        sw   $15, 4($2)
        jr   $31
```

```
00000000101000010000000000011000
00000000000110000001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

For example:



A compiler enables a programmer to write this high-level language expression:

```
A + B
```

The compiler would compile it into this assembly language statement:

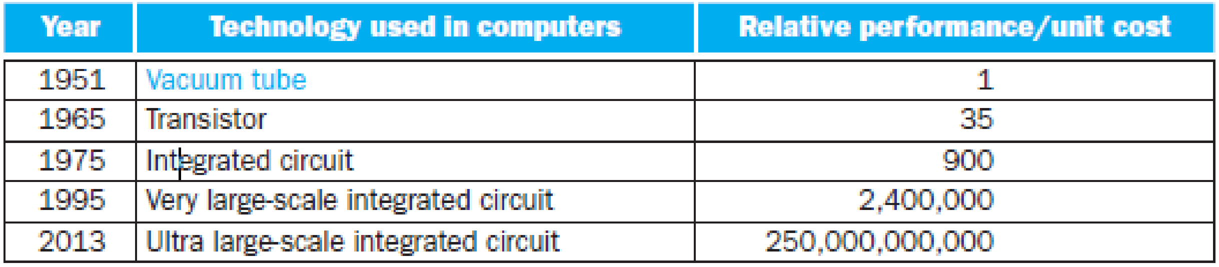
```
add A,B
```

The assembler would translate this statement into the binary instruction that tells the computer to add the two numbers A and B:

```
1000110010100000
```

## TECHNOLOGY USED IN COMPUTER

The technology that have been used in computers are changing constantly over time and it is given below.



| Year | Technology used in computers | Relative performance/unit cost |
|---|---|---|
| 1951 | Vacuum tube | 1 |
| 1965 | Transistor | 35 |
| 1975 | Integrated circuit | 900 |
| 1995 | Very large-scale integrated circuit | 2,400,000 |
| 2013 | Ultra large-scale integrated circuit | 250,000,000,000 |

- **Vacuum Tube**is the electronic component used in I$^{st}$ generation computer. An electronic component that consists of a hollow glass tube about 5 to 10 cm long from which as much air hasbeen removed as possible and which uses an electron beam to transfer data.
- **Transistor** is the semiconductor device used in II$^{nd}$ generation computer. It is an on/off switch controlled by an electric signal. This will reduce the physical size and power dissipation of the computer and increase the performance.
- **Integrated Circuit(IC)**combined dozens to hundreds of transistors into a single chip. This will further reduce the physical size, power dissipation and increase the performance.
- **Very large scale integrated (VLSI) circuit:**A device containing hundreds of thousands to millions of transistors.
- **Ultra large-scale integration (ULSI)** is the process of integrating or embedding millions of transistors on a single silicon semiconductor microchip.

  **Moore's Law: The number of transistors is a chip is doubled in every 18 to 24 months.**

- **Growth of DRAM capacity**

  Moore's law resulted from a prediction of such growth in IC capacity made by Gordon Moore, one of the founders of Intel during the 1960s.

  The growth of Dram capacity quadrupled for every 3 years from 1977 to 1995 and in recent years the rate of increase has slowdown and approximately doubling every 2 or 3 years.

| Year | - | Chip Size |
|---|---|---|
| 1977 | - | 16 K |
| 1980 | - | 64 K |
| 1983 | - | 256 K |
| 1986 | - | 1 M |

| | |
|---|---|
| 1989 | 4M |
| 1992 | 16 M |
| 1996 | 64 M |
| 1998 | 128 M |
| 2000 | 256 M |
| 2004 | 512 M |
| 2007 | 1 Gbit |

- **Growth of Transistor in processors.**

| Year | Processor | No of Transistor |
|---|---|---|
| 1978 | 8086 | 20,000 |
| 1982 | 80286 | 100,000 |
| 1985 | 80386 | 200,000 |
| 1989 | 80486 | 1000,000 |
| 1993 | Pentium | 5 x $10^6$ |
| 2001 | Pentium IV | 20 x $10^6$ |
| 2005 | Core 2 dual | 200 x $10^6$ |

**POWER WALL**

Both clock rate and power increased rapidly fordecades, and then flattened off recently.

For CMOS, the primary source of energy consumption is so-called dynamic energy—that is, energy that is consumed when transistors switch states from 0 to 1 and vice versa. The dynamic energy depends on the capacitive loading of each transistor and the voltage applied:

**The increase in clock rate and power of eight generations of Intel microprocessors over 30 years.**

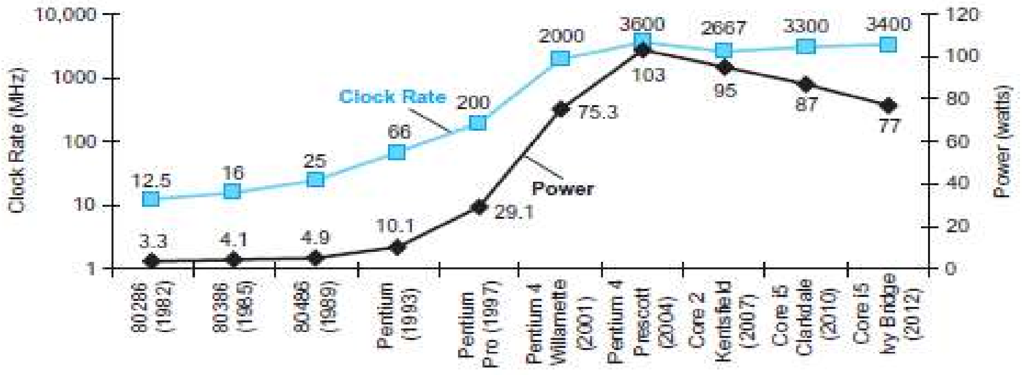


**FIGURE 1.16 Clock rate and Power for Intel x86 microprocessors over eight generations and 25 years.** The Pentium 4 made a dramatic jump in clock rate and power but less so in performance. The Prescott thermal problems led to the abandonment of the Pentium 4 line. The Core 2 line reverts to a simpler pipeline with lower clock rates and multiple processors per chip. The Core i5 pipelines follow in its footsteps.

The dynamic energy depends on the capacitive loading of each transistor and the voltage applied

$$\text{Energy} \propto \text{Capacitive load} \times \text{Voltage}^2$$

The energy of a single transition is then

$$\text{Energy} \propto 1/2 \times \text{Capacitive load} \times \text{Voltage}^2$$

The dynamic power dissipation depends on the capacitive loading of each transistor, the voltage applied and the frequency that the transistor is switched.

Power = Capacitive Load x $\text{Voltage}^2$ x Frequency switched.

Frequency switched is a function of the clock rate. The capacitive load per transistor is a function of both the number of transistors connected to an output (called the ***fanout*)** and the technology, which determines the capacitance of both wires and transistors.

**The above figure shows how could clock rates grow by a factor of 1000 while power grew by only a factor of 30?**

Energy and thus power can be reduced by lowering the voltage, which occurred with each new generation of technology, and power is a function of the voltage squared. Typically, the voltage was reduced about15% per generation. In 20 years, voltages have gone from 5 V to 1 V, which is whythe increase in power is only 30 times.

## PERFORMANCE MEASURE

Performance of a computer can be measured by speed with which it can execute the program. Response time and Throughput are the two factors to measure the performance.

**Response time** Also called **execution time**. The total time required for the computer to complete a task, including disk accesses, memory accesses, I/O activities, operating system overhead, CPU execution time, and so on.

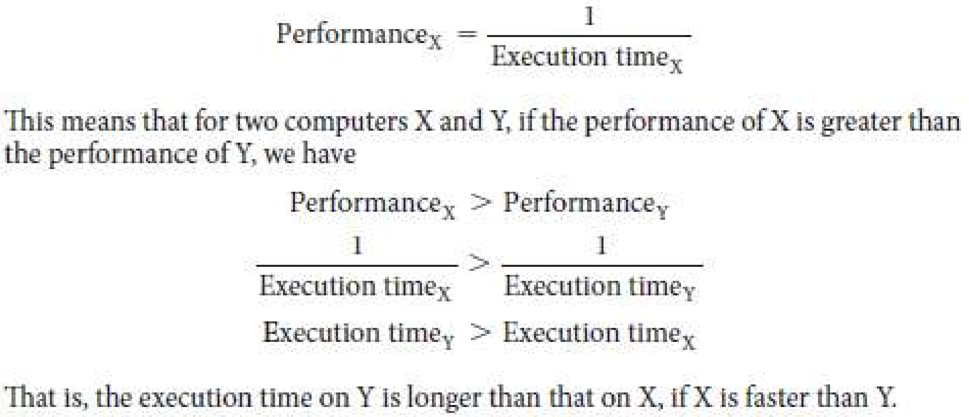
**Throughput** Also called **bandwidth**. Another measure of performance, it is the number of tasks completed per unit time.

Speed of the computer is affected by

- Hardware design
- Machine language instruction of the computer.
- Compiler, which translates high-level language into machine language.

For best performance, it is necessary to design hardware, machine instruction set and compiler in a coordinated way.

## I. RELATIVE PERFORMANCE



$$\text{Performance}_X = \frac{1}{\text{Execution time}_X}$$

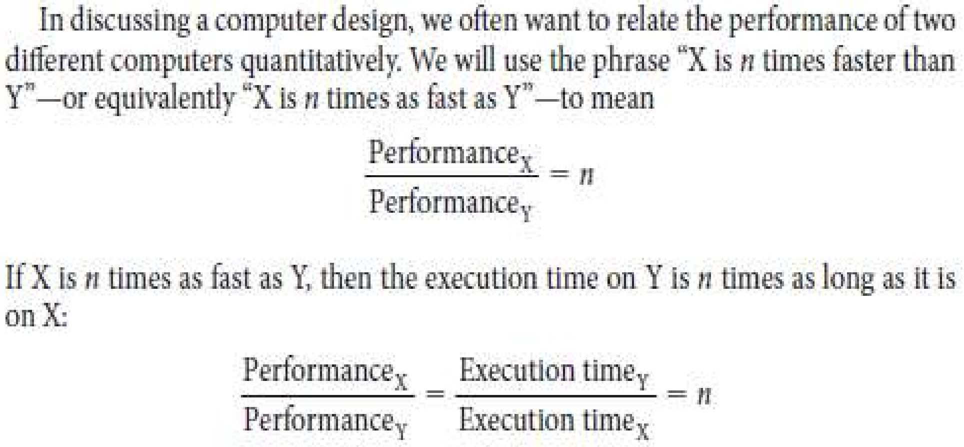
This means that for two computers X and Y, if the performance of X is greater than the performance of Y, we have

$$\text{Performance}_X > \text{Performance}_Y$$

$$\frac{1}{\text{Execution time}_X} > \frac{1}{\text{Execution time}_Y}$$

$$\text{Execution time}_Y > \text{Execution time}_X$$

That is, the execution time on Y is longer than that on X, if X is faster than Y.

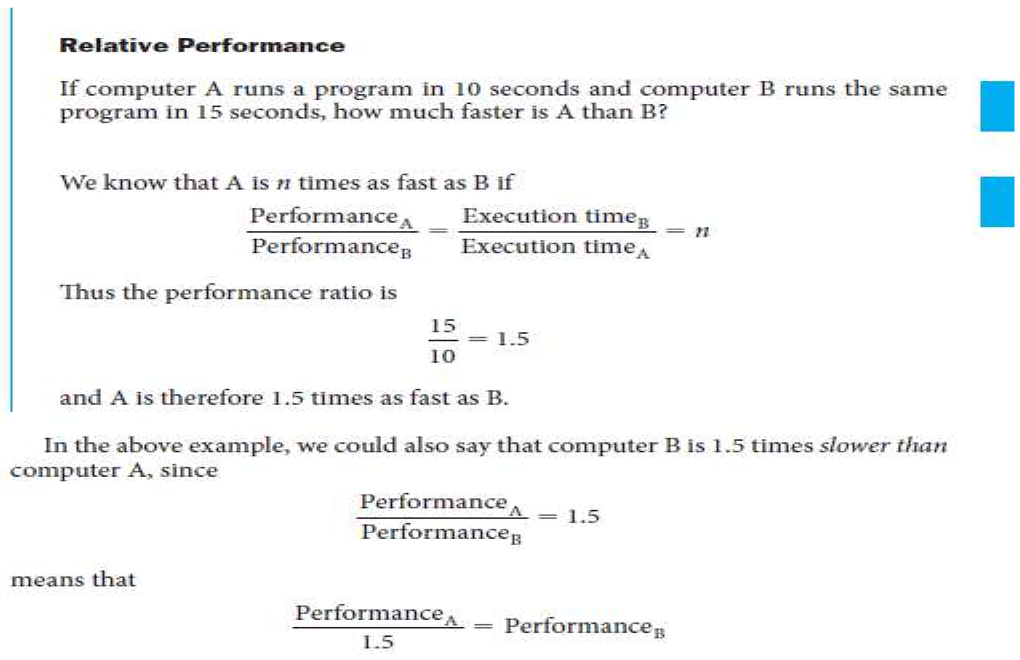


In discussing a computer design, we often want to relate the performance of two different computers quantitatively. We will use the phrase "X is *n* times faster than Y"—or equivalently "X is *n* times as fast as Y"—to mean

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = n$$

If X is *n* times as fast as Y, then the execution time on Y is *n* times as long as it is on X:

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{Execution time}_Y}{\text{Execution time}_X} = n$$



**Relative Performance**

If computer A runs a program in 10 seconds and computer B runs the same program in 15 seconds, how much faster is A than B?

We know that A is $n$ times as fast as B if

$$\frac{\text{Performance}_A}{\text{Performance}_B} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = n$$

Thus the performance ratio is

$$\frac{15}{10} = 1.5$$

and A is therefore 1.5 times as fast as B.

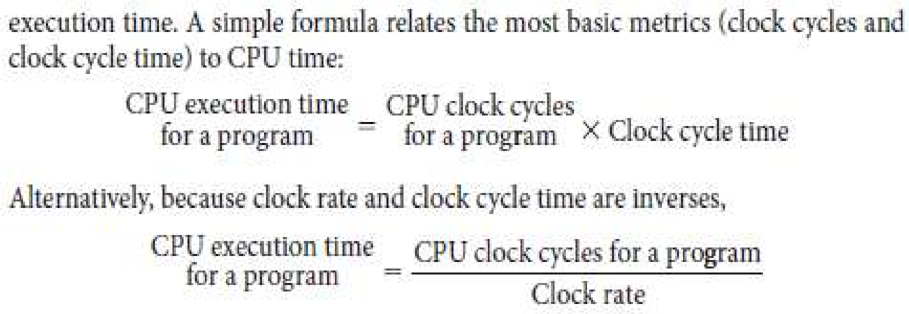
In the above example, we could also say that computer B is 1.5 times *slower than* computer A, since

$$\frac{\text{Performance}_A}{\text{Performance}_B} = 1.5$$

means that

$$\frac{\text{Performance}_A}{1.5} = \text{Performance}_B$$

## II. CPU PERFORMANCE AND ITS FACTOR

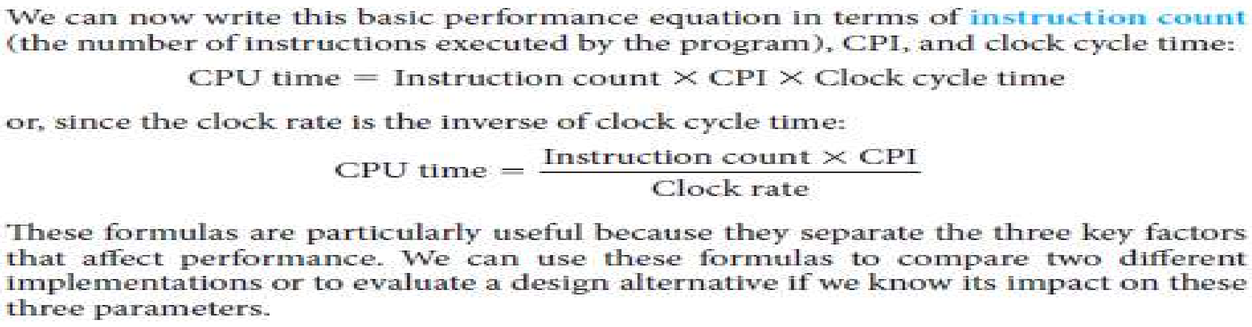


execution time. A simple formula relates the most basic metrics (clock cycles and clock cycle time) to CPU time:

$$\begin{matrix}\text{CPU execution time}\\ \text{for a program}\end{matrix} = \begin{matrix}\text{CPU clock cycles}\\ \text{for a program}\end{matrix} \times \text{Clock cycle time}$$

Alternatively, because clock rate and clock cycle time are inverses,

$$\begin{matrix}\text{CPU execution time}\\ \text{for a program}\end{matrix} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$



We can now write this basic performance equation in terms of instruction count (the number of instructions executed by the program), CPI, and clock cycle time:

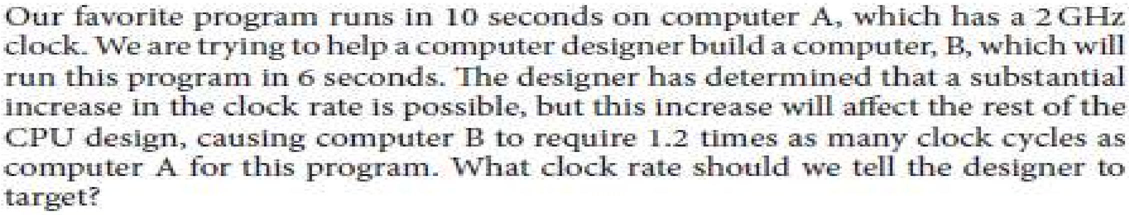
$$\text{CPU time} = \text{Instruction count} \times \text{CPI} \times \text{Clock cycle time}$$

or, since the clock rate is the inverse of clock cycle time:

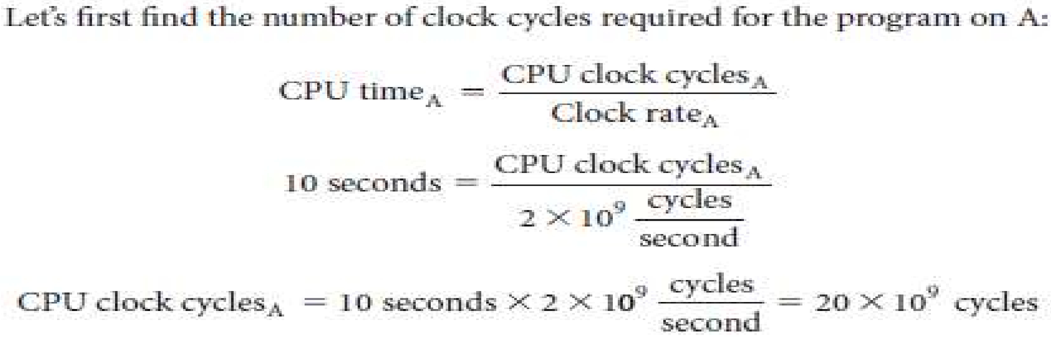
$$\text{CPU time} = \frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}}$$

These formulas are particularly useful because they separate the three key factors that affect performance. We can use these formulas to compare two different implementations or to evaluate a design alternative if we know its impact on these three parameters.

**EXAMPLE**
**IMPROVING PERFORMANCE**



Our favorite program runs in 10 seconds on computer A, which has a 2 GHz clock. We are trying to help a computer designer build a computer, B, which will run this program in 6 seconds. The designer has determined that a substantial increase in the clock rate is possible, but this increase will affect the rest of the CPU design, causing computer B to require 1.2 times as many clock cycles as computer A for this program. What clock rate should we tell the designer to target?

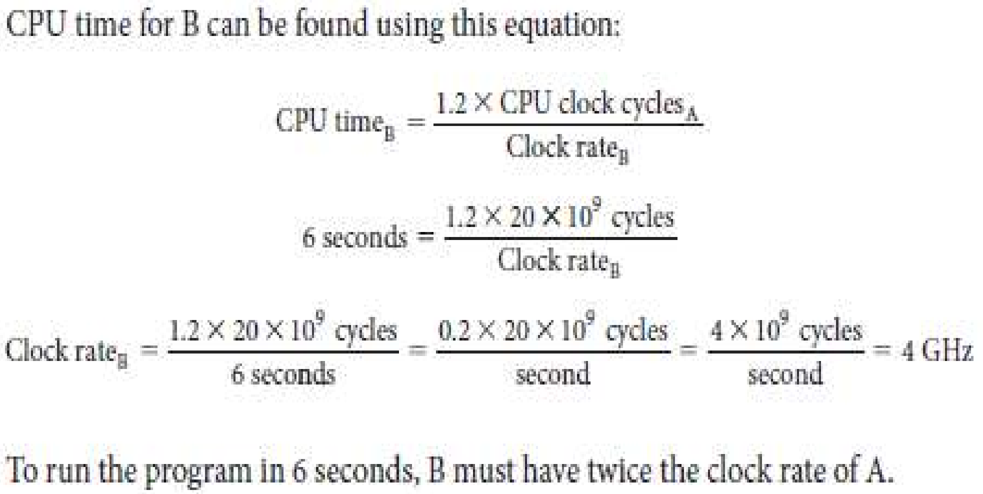


Let's first find the number of clock cycles required for the program on A:

$$\text{CPU time}_A = \frac{\text{CPU clock cycles}_A}{\text{Clock rate}_A}$$

$$10 \text{ seconds} = \frac{\text{CPU clock cycles}_A}{2 \times 10^9 \ \frac{\text{cycles}}{\text{second}}}$$

$$\text{CPU clock cycles}_A = 10 \text{ seconds} \times 2 \times 10^9 \ \frac{\text{cycles}}{\text{second}} = 20 \times 10^9 \text{ cycles}$$



CPU time for B can be found using this equation:

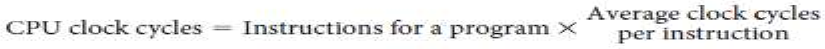
$$\text{CPU time}_B = \frac{1.2 \times \text{CPU clock cycles}_A}{\text{Clock rate}_B}$$

$$6 \text{ seconds} = \frac{1.2 \times 20 \times 10^9 \text{ cycles}}{\text{Clock rate}_B}$$

$$\text{Clock rate}_B = \frac{1.2 \times 20 \times 10^9 \text{ cycles}}{6 \text{ seconds}} = \frac{0.2 \times 20 \times 10^9 \text{ cycles}}{\text{second}} = \frac{4 \times 10^9 \text{ cycles}}{\text{second}} = 4 \text{ GHz}$$

To run the program in 6 seconds, B must have twice the clock rate of A.

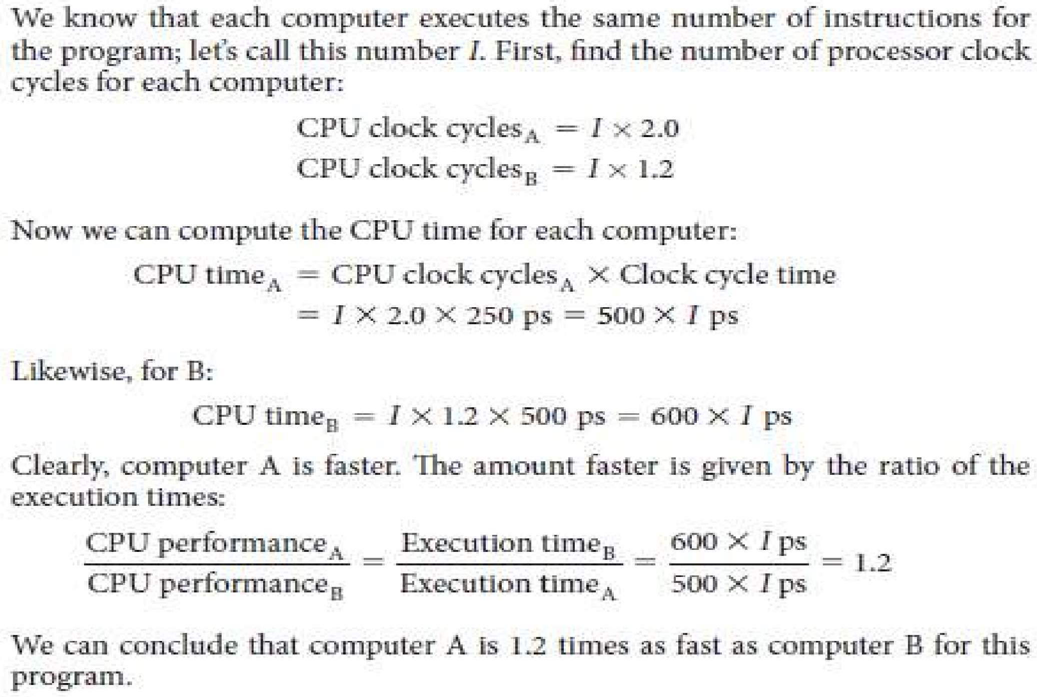
## III. INSTRUCTION PERFORMANCE



$$\text{CPU clock cycles} = \text{Instructions for a program} \times \begin{array}{c}\text{Average clock cycles}\\ \text{per instruction}\end{array}$$

**EXAMPLE:**

Suppose we have two implementations of the same instruction set architecture. Computer A has a clock cycle time of 250 ps and a CPI of 2.0 for some program, and computer B has a clock cycle time of 500 ps and a CPI of 1.2 for the same program. Which computer is faster for this program and by how much?



We know that each computer executes the same number of instructions for the program; let's call this number *I*. First, find the number of processor clock cycles for each computer:

$$\text{CPU clock cycles}_A = I \times 2.0$$
$$\text{CPU clock cycles}_B = I \times 1.2$$

Now we can compute the CPU time for each computer:

$$\text{CPU time}_A = \text{CPU clock cycles}_A \times \text{Clock cycle time}$$
$$= I \times 2.0 \times 250 \text{ ps} = 500 \times I \text{ ps}$$

Likewise, for B:

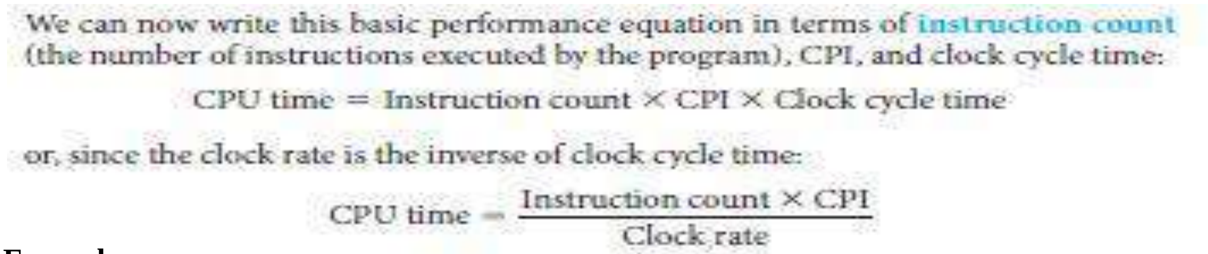
$$\text{CPU time}_B = I \times 1.2 \times 500 \text{ ps} = 600 \times I \text{ ps}$$

Clearly, computer A is faster. The amount faster is given by the ratio of the execution times:

$$\frac{\text{CPU performance}_A}{\text{CPU performance}_B} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = \frac{600 \times I \text{ ps}}{500 \times I \text{ ps}} = 1.2$$

We can conclude that computer A is 1.2 times as fast as computer B for this program.

**The Classic CPU Performance Equation:**



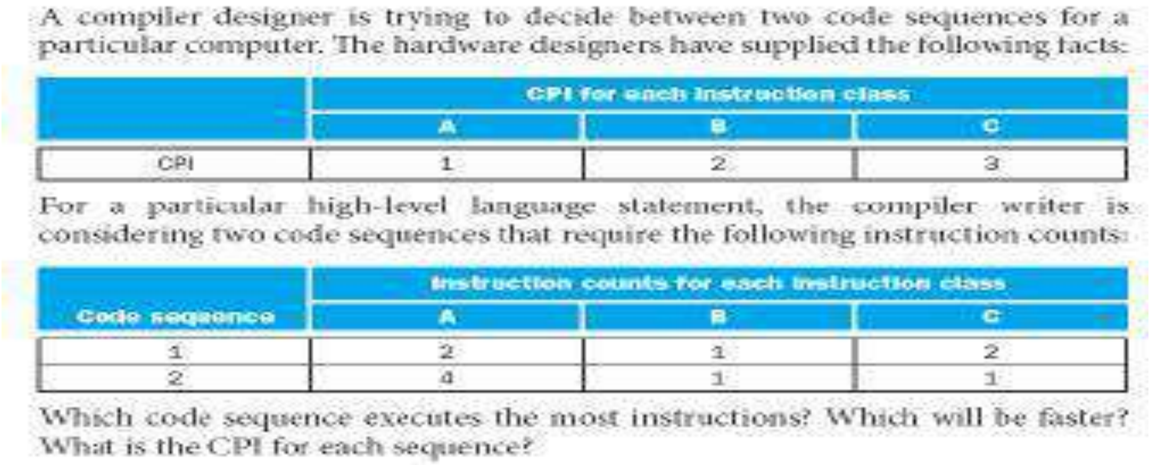
We can now write this basic performance equation in terms of instruction count (the number of instructions executed by the program), CPI, and clock cycle time:

$$\text{CPU time} = \text{Instruction count} \times \text{CPI} \times \text{Clock cycle time}$$

or, since the clock rate is the inverse of clock cycle time:

$$\text{CPU time} = \frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}}$$

**Example:**



A compiler designer is trying to decide between two code sequences for a particular computer. The hardware designers have supplied the following facts:

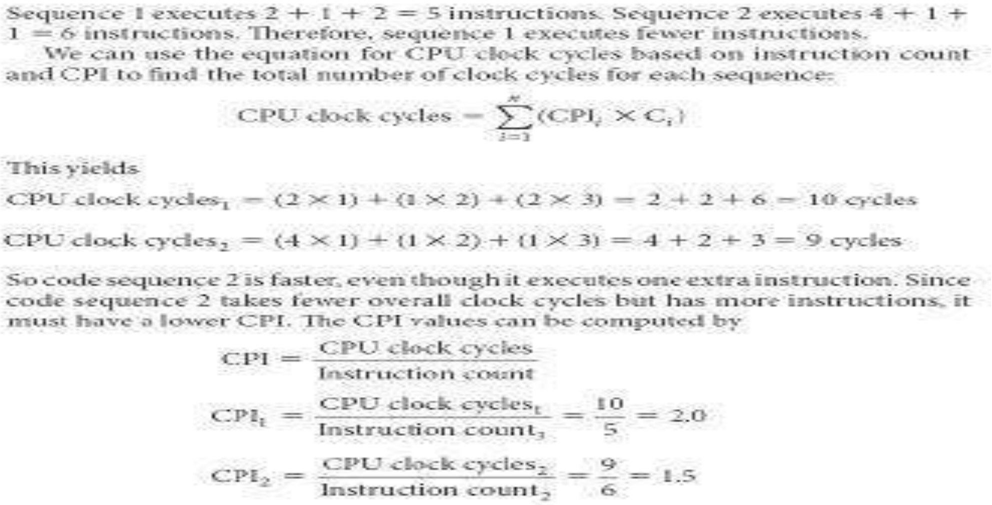
| | CPI for each instruction class | | |
|---|---|---|---|
| | A | B | C |
| CPI | 1 | 2 | 3 |

For a particular high-level language statement, the compiler writer is considering two code sequences that require the following instruction counts:

| | Instruction counts for each instruction class | | |
|---|---|---|---|
| Code sequence | A | B | C |
| 1 | 2 | 1 | 2 |
| 2 | 4 | 1 | 1 |

Which code sequence executes the most instructions? Which will be faster? What is the CPI for each sequence?

**Solution:**



Sequence 1 executes 2 + 1 + 2 = 5 instructions. Sequence 2 executes 4 + 1 + 1 = 6 instructions. Therefore, sequence 1 executes fewer instructions.

We can use the equation for CPU clock cycles based on instruction count and CPI to find the total number of clock cycles for each sequence:

$$\text{CPU clock cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times C_i)$$

This yields

$$\text{CPU clock cycles}_1 = (2 \times 1) + (1 \times 2) + (2 \times 3) = 2 + 2 + 6 = 10 \text{ cycles}$$

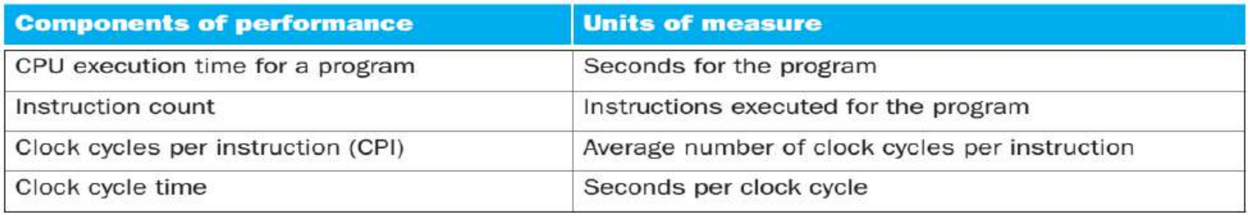
$$\text{CPU clock cycles}_2 = (4 \times 1) + (1 \times 2) + (1 \times 3) = 4 + 2 + 3 = 9 \text{ cycles}$$

So code sequence 2 is faster, even though it executes one extra instruction. Since code sequence 2 takes fewer overall clock cycles but has more instructions, it must have a lower CPI. The CPI values can be computed by

$$\text{CPI} = \frac{\text{CPU clock cycles}}{\text{Instruction count}}$$

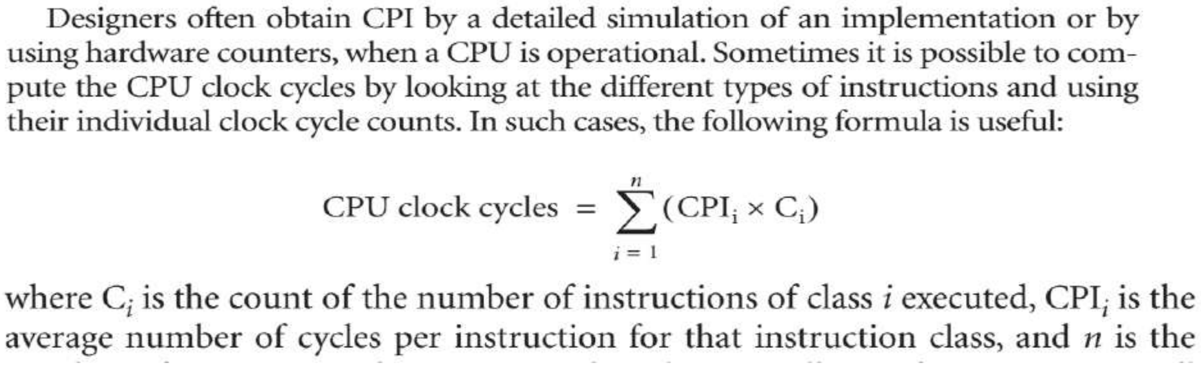
$$\text{CPI}_1 = \frac{\text{CPU clock cycles}_1}{\text{Instruction count}_1} = \frac{10}{5} = 2.0$$

$$\text{CPI}_2 = \frac{\text{CPU clock cycles}_2}{\text{Instruction count}_2} = \frac{9}{6} = 1.5$$



| Components of performance | Units of measure |
|---|---|
| CPU execution time for a program | Seconds for the program |
| Instruction count | Instructions executed for the program |
| Clock cycles per instruction (CPI) | Average number of clock cycles per instruction |
| Clock cycle time | Seconds per clock cycle |

**FIGURE 4.2 The basic components of performance and how each is measured.**



Designers often obtain CPI by a detailed simulation of an implementation or by using hardware counters, when a CPU is operational. Sometimes it is possible to compute the CPU clock cycles by looking at the different types of instructions and using their individual clock cycle counts. In such cases, the following formula is useful:

$$\text{CPU clock cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times C_i)$$

where $C_i$ is the count of the number of instructions of class *i* executed, $\text{CPI}_i$ is the average number of cycles per instruction for that instruction class, and *n* is the number of instruction classes.

## PERFORMANCE MEASUREMENT

The computer community adopted the idea of measuring computer performance using benchmark programs. To make comparisons possible, standardized programs must be used. The performance measure is the time it takes a computer to execute a given benchmark.

A nonprofit organization called System Performance Evaluation Corporation (SPEC) measures the performance of the system.

The SPEC rating is computed as follows

Running time on the reference computer
SPEC rating = ------------------------------------------------

Running time on the computer under test

Let SPECi be the rating for program i in the suite. The overall SPEC rating for the computer is given by

$$\text{SPEC rating} = \left(\prod_{i=1}^{n} \text{SPEC}_i\right)^{1/n}$$

where n is the number of programs in the suite

## INSTRUCTION IN MIPS

## OPERATION AND OPERANDS IN MIPS

**Based on operations the MIPS instructions** are classified into five groups/Instructions types

1. Arithmetic instruction
2. Logical Instructions
3. Data transfer Instructions
4. Conditional Branch Instructions
5. Unconditional Jump Instructions

### 1. Arithmetic instructions(R-Type)

These Instructions will perform arithmetic operations with register only.

| Examples: | | |
|---|---|---|
| | add | for addition of two registers |
| | Sub | for subtraction |
| | addi | for adding a constant |

**Arithmetic Instructions**

| Instruction | Example | Meaning | Comments |
|---|---|---|---|
| add | add $s1,$s2,$s3 | $s1=$s2+$s3 | Three register operands |
| subtract | sub $s1,$s2,$s3 | $s1=$s2 - $s3 | Three register operands |
| add immediate | addi $s1,$s2,20 | $s1=$s2 + 20 | Used to add constants |

**Example 1:**

C code for the following instruction:

**f = (g + h) - (i + j);**

- f, …, j in $s0, …, $s4

- Compiled MIPS code:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

### 2. Logical Instructions ( LOGICAL OPERATIONS)

These Instruction will perform logic operations and shift operations with register only.

Shift Instruction moves all the bits in a word to the left or right, filling the emptied bits with 0s. The *shamt* field in the R-format. It stands for *shift amount* and is used in shift instructions.

**Logical Instructions**

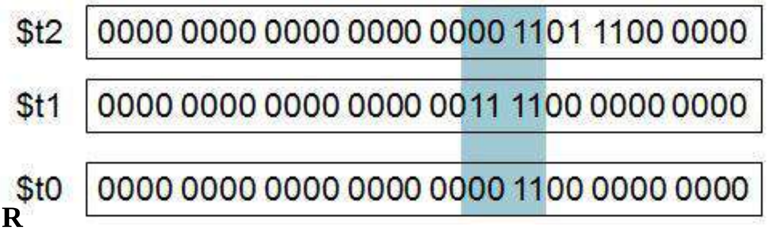
| Instruction | Example | Meaning | Comments |
|---|---|---|---|
| and | and | $s1 =$s2 & $s3 | Three register operands; bit-by-bit |

|  |  |  |  |
|---|---|---|---|
|  | $s1,$s2,$s3 |  | AND |
| or | or    $s1,$s2,$s3 | $s1 =$s2 \| $s3 | Three register operands; bit-by-bit OR |
| not | nor  $s1,$s2,$s3 | $s1 = ~($s2 \| $s3) | Three register,operands; bit-by-bit NOR |
| and immediate | andi $s1,$s2,20 | $s1 =$s2 & $20 | Bit-by-bit AND register with constant |
| or immediate | ori  $s1,$s2,20 | $s1 =$s2 \| $20 | Bit-by-bit OR register with constant |
| shift left logical | sll  $s1,$s2,10 | $s1 =$s2 << 10 | Shift left by constant |
| shift right logical | sr l $s1,$s2,10 | $s1 =$s2 >> 10 | Shift right lo by constant |

Example

**AND**

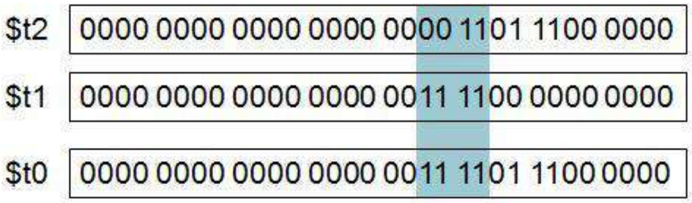
- Useful to mask bits in a word
  - Select some bits, clear others to 0 **and $t0, $t1, $t2**



```
$t2  0000 0000 0000 0000 0000 1101 1100 0000
$t1  0000 0000 0000 0000 0011 1100 0000 0000
$t0  0000 0000 0000 0000 0000 1100 0000 0000
```

**OR**

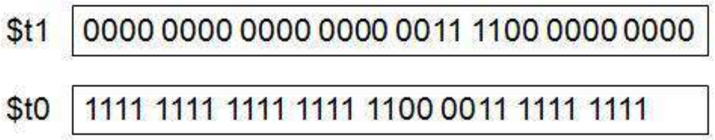
- Used to include bit in a word
  - Set some bits to 1, leave others unchanged **or $t0, $t1, $t2**



```
$t2  0000 0000 0000 0000 0000 1101 1100 0000
$t1  0000 0000 0000 0000 0011 1100 0000 0000
$t0  0000 0000 0000 0000 0011 1101 1100 0000
```

**NOT**

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0



```
$t1  0000 0000 0000 0000 0011 1100 0000 0000
$t0  1111 1111 1111 1111 1100 0011 1111 1111
```

**SHIFT LEFT LOGICAL**
**sll $s1,$s2,4 # reg $t2 = reg $s0 << 4 bits**
**Before Execution**
For example, if register $s0=9
0000 0000 0000 00000 000 0000 0000 0000 1001two= 9ten
The instruction to shift left by 4 was executed, the new value would look like this:
**After Execution**
0000 0000 0000 0000 0000 0000 0000 1001 0000two= 144ten

**SHIFT RIGHT LOGICAL**
**Srl $s1,$s2,1 # reg $t2 = reg $s0 » 1 bits**
**Before Execution**
For example, if register $s0 = 9
0000 0000 0000 00000 000 0000 0000 0000 1001two= 9ten
The instruction to shift right by 1 was executed, the new value would look like this:
**After Execution**
0000 0000 0000 0000 0000 0000 0000 0000 0100two= 4two

## 3. Data transfer Instructions

These instruction will transfer data between registers and main memory.

| | | |
|---|---|---|
| Examples: | lw | load word i.e. transfer of data from memory to register |
| | sw | store word i.e. transfer of data from register to memory |

Arithmetic operations occur only on registers in MIPS instructions; thus, MIPS must include instructions that transfer data between memory and registers. Such instructions are called data transfer instructions. To access a word in memory, the instruction must supply the memory address.Memory is just a large, single-dimensional array, with the address acting as the index to that array, starting at 0.

### EXAMPLE

**load word lw $s1,100($s2) $s1 = Memory[$s2 + 100] Data from memory to register**

The data transfer instruction that copies data from memory to a register is traditionally called *load*. The format of the load instruction is the name of the operation followed by the register to be loaded, then a constant and register used to access memory. The sum of the constant portion of the instruction and the contents of the second register forms the memory address.

**store word sw $s1,100($s2) Memory[$s2 + 100] = $s1 Data from register to memory**

The data transfer instruction that copies data from Register to a Memory is traditionally called store.

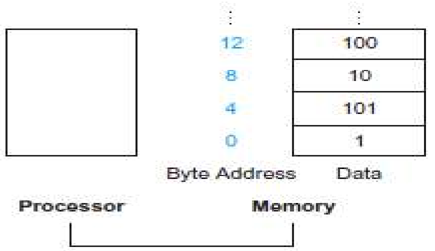


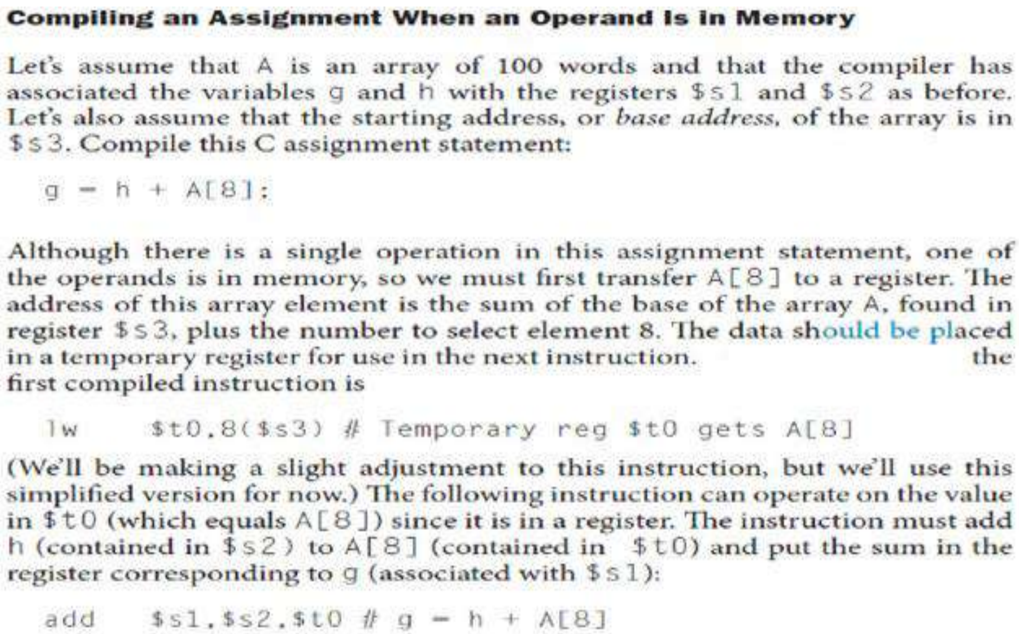
**FIGURE 2.3 Actual MIPS memory addresses and contents of memory for those words.** The changed addresses are highlighted to contrast with Figure 2.2. Since MIPS addresses each byte, word addresses are multiples of 4: there are 4 bytes in a word.

.

| Instruction | Example | Meaning | Comments |
|---|---|---|---|
| load word | lw $s1,20($s2) | $s1 = Memory ($s2 +20) | Word from memory to register |
| store word | sw $s1,20($s2) | Memory ($s2 +20) = $s1 | Word from register to memory |
| load half | lh $s1,20($s2) | $s1 = Memory ($s2 +20) | Half word memory to register |
| load half unsigned | lhu $s1,20($s2) | $s1 = Memory ($s2 +20) | Half word memory to register |
| store half | sh $s1,20($s2) | Memory ($s2 +20) = $s1 | Half word register to memory |
| load byte | lb $s1,20($s2) | $s1 = Memory ($s2 +20) | Byte from memory to register |
| load byte unsigned | lbu $s1,20($s2) | $s1 = Memory ($s2 +20) | Byte from memory to register |
| store byte | sb $s1,20($s2) | Memory ($s2 +20) = $s1 | Byte from register to memory |
| load linked word | ll $s1,20($s2) | $s1 = Memory ($s2 +20) | Load word as 1st half of atomic swap |
| store condition | sc $s1,20($s2) | Memory ($s2 +20) = | store word as 2nd half of |

|  |  |  |  |
|---|---|---|---|
| word |  | $s1; $s1= 0 or 1 | atomic swap |
| load upper immed | lui  $s1,20 | $s1=20 * $2^{16}$ | Loads constant in upper 16 bits |

```
Example:      add i  $s1, $s2, 20          # $s1= $s2+20
              lw  $s1, 20($s2)             #$s1=memory($s2+20)
              sw  $s1, 20($s2)             #memory($s2+20) = $s1
```



**Compiling an Assignment When an Operand Is in Memory**

Let's assume that A is an array of 100 words and that the compiler has associated the variables g and h with the registers $s1 and $s2 as before. Let's also assume that the starting address, or *base address*, of the array is in $s3. Compile this C assignment statement:

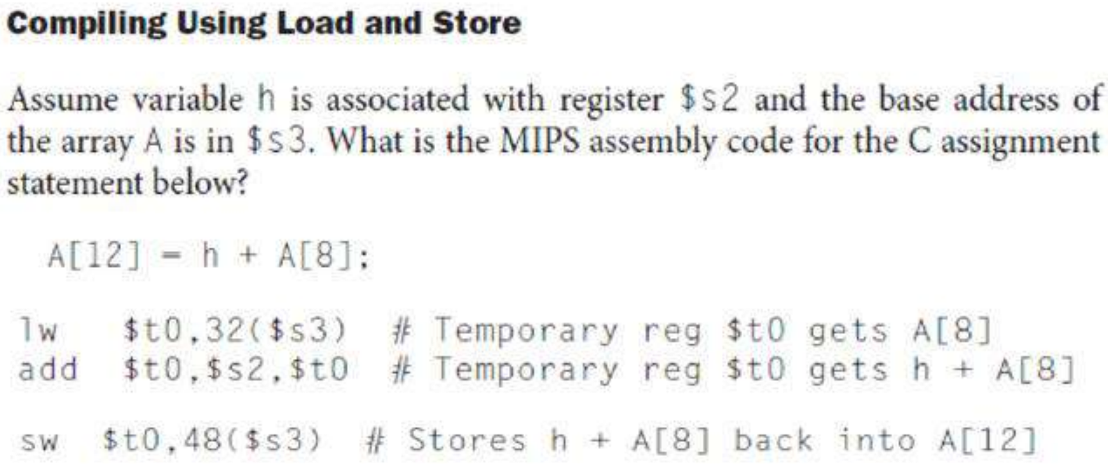
```
g = h + A[8];
```

Although there is a single operation in this assignment statement, one of the operands is in memory, so we must first transfer A[8] to a register. The address of this array element is the sum of the base of the array A, found in register $s3, plus the number to select element 8. The data should be placed in a temporary register for use in the next instruction. the first compiled instruction is

```
lw    $t0,8($s3) # Temporary reg $t0 gets A[8]
```

(We'll be making a slight adjustment to this instruction, but we'll use this simplified version for now.) The following instruction can operate on the value in $t0 (which equals A[8]) since it is in a register. The instruction must add h (contained in $s2) to A[8] (contained in $t0) and put the sum in the register corresponding to g (associated with $s1):

```
add   $s1,$s2,$t0 # g = h + A[8]
```



**Compiling Using Load and Store**

Assume variable h is associated with register $s2 and the base address of the array A is in $s3. What is the MIPS assembly code for the C assignment statement below?

```
A[12] = h + A[8];
```

```
lw    $t0,32($s3)   # Temporary reg $t0 gets A[8]
add   $t0,$s2,$t0   # Temporary reg $t0 gets h + A[8]

sw   $t0,48($s3)   # Stores h + A[8] back into A[12]
```

## 4. Conditional Branch Instructions (DECISION MAKING)

These instruction will perform transfer the control from current location to target address when the specified condition is satisfied otherwise control will go to next instructions.

| | | |
|---|---|---|
| Examples: | beq | branch on equal |
| | bne | branch not equal |

Branch instruction loads a new value into the program counter. As a result, the processor fetches and executes the instruction at this new address, called branch target, instead of executing in sequential order.

A Conditional branch instruction repeats the loop only if a specified condition is satisfied. If the condition is not satisfied it comes out of the loop.

Decision making is commonly represented in programming languages using the *if* statement, sometimes combined with *go to* statements and labels. MIPS assembly language includes two decision-making instructions, similar to an *if* statement with a *go to*.

**EXAMPLE**

**beq register1, register2, L1**

This instruction means go to the statement labeled L1 if the value in register1 equals the value in register2. The mnemonic beq stands for *branch if equal*. The second instruction is

**bne register1, register2, L1**

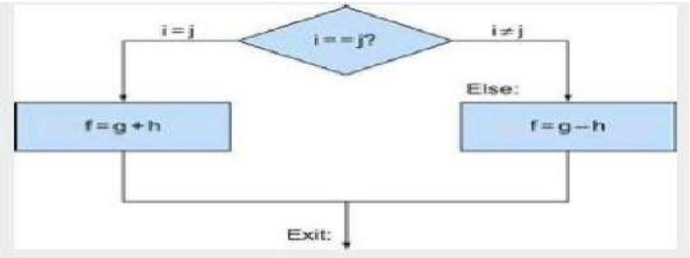
It means go to the statement labeled L1 if the value in register1 does *not* equal the value in register2. The mnemonic bne stands for *branch if not equal*. These two instructions are traditionally called conditional branches.

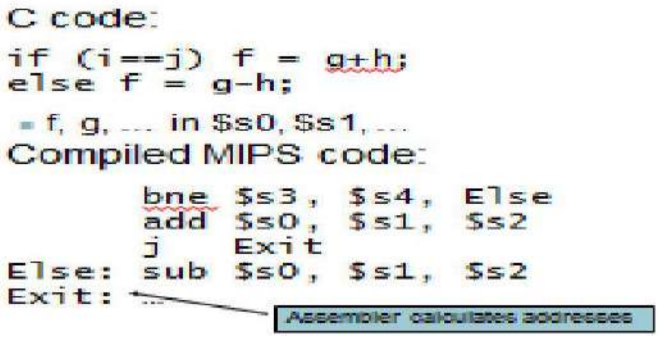
**Conditional Branch Instructions**

| **Instruction** | **Example** | **Meaning** | **Comments** |
|---|---|---|---|
| branch on equal | beq $s1,$s2,25 | if($s1 == $s2) go t o PC + 4 +100 | Equal test ; PC- relative branch |
| branch on not equal | bne $s1,$s2,25 | if($s1!= $s2) go t o PC + 4 +100 | Not equal test ; PC- relative |
| set on less than | slt $s1,$s2,$s3 | if($s2 < $s3) $s1=1;<br>else $s1 = 0 | Compare less than for beq, bne |
| set on less than unsigned | sltu $s1,$s2,$s3 | if($s2 < $s3) $s1=1;<br>else $s1 = 0 | Compare less than unsigned |
| set on less than | slti $s1,$s2,20 | if($s2 < 20) $s1=1; | Compare less than constant |

| | | | |
|---|---|---|---|
| immediate | | else $s1 = 0 | |
| set on less than immediate unsigned | sltiu  $s1,$s2,20 | if($s2 < 20) $s1=1;<br>else $s1 = 0 | Compare less than constant unsigned |

**Example:**
**Compiling IF statements:**




C code:

```
if (i==j) f = g+h;
else f = g-h;
```

▪ f, g, … in $s0, $s1, …

Compiled MIPS code:

```
      bne  $s3, $s4, Else
      add  $s0, $s1, $s2
      j    Exit
Else: sub  $s0, $s1, $s2
Exit: …
```

Assembler calculates addresses

## 5. Unconditional Jump Instructions

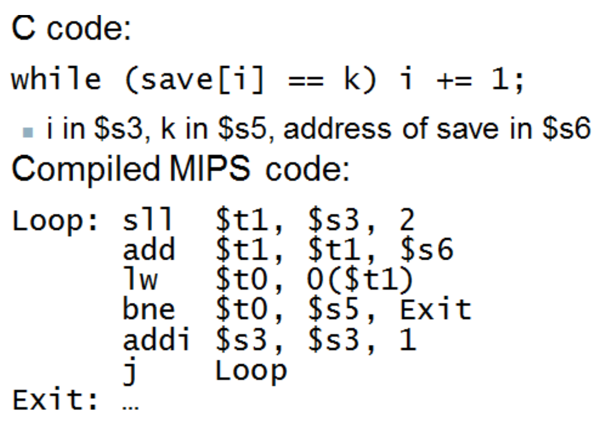
These instructions will transfer control from current location to target address.
Examples:   J: jump to transfer address
jr: jump to target address available in register $ra used for procedure return.

jal: load the return address (PC+4 )into register $ra and jump to target address, used for procedure call.

Note: MIPS processor has no I/O Instructions.

**Unconditional Jump Instructions**

| Instruction | Example | Meaning | Comments |
|---|---|---|---|
| jump | j 2500 | go to 10000 | Jump to target address |
| jump register | jr $ra | go to $ra | For switch, procedure return |
| jump and link | jal 2500 | $ra = PC + 4; go to 10000 | For procedure call |



C code:

```
while (save[i] == k) i += 1;
```

- i in $s3, k in $s5, address of save in $s6

Compiled MIPS code:

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

## OPERANDS AVAILABLE IN MIPS

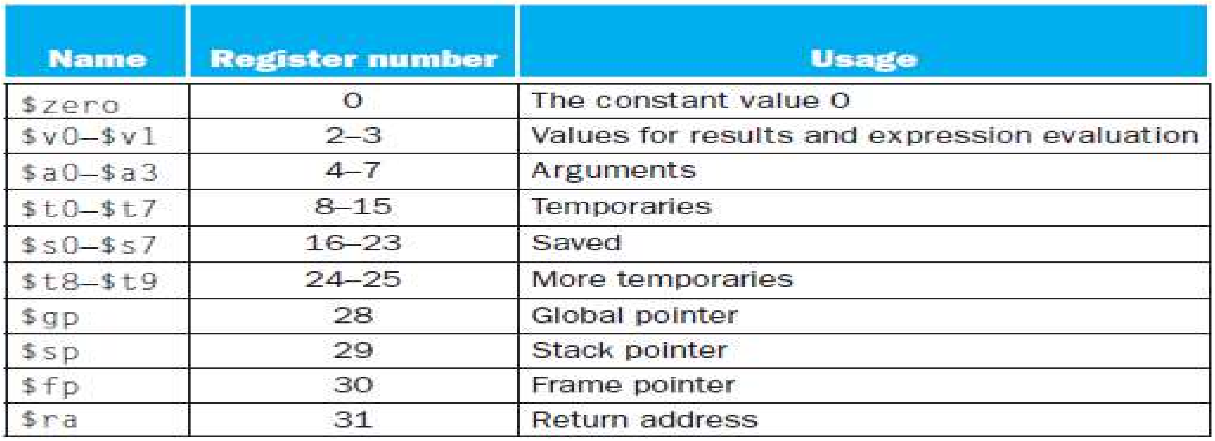
MIPS instruction uses three type of operands

1. Register 2. Memory 3. Immediate

1. **Register:** MIPS processor has 32 registers.

The size of the register is 32 bits. These registers are represented by 5 bit binary number in the MIPS instruction and they are referred by 2 or 3 character name in MIPS assembly code.

Example: add $t1, $s1, $s2 #$t1=$s1+$s2



| Name | Register number | Usage |
|---|---|---|
| $zero | 0 | The constant value 0 |
| $v0–$v1 | 2–3 | Values for results and expression evaluation |
| $a0–$a3 | 4–7 | Arguments |
| $t0–$t7 | 8–15 | Temporaries |
| $s0–$s7 | 16–23 | Saved |
| $t8–$t9 | 24–25 | More temporaries |
| $gp | 28 | Global pointer |
| $sp | 29 | Stack pointer |
| $fp | 30 | Frame pointer |
| $ra | 31 | Return address |

2. **Memory**

Since arithmetic and logic instructions use only register operands, MIPS need instruction to transfer data between memory & register. Such instruction are called Data Transfer instructions.

The instruction which transfer the data from memory to register is called Load.

Example: lw $t0, 8($s1)

| | |
|---|---|
| Ist operand | Register operand |
| II nd operand | Memory operand |
| 8 | is called offset |

$s1 is called base register

- To get the proper memory address, the offset to be added to base registers $s1.
- Normally base register $s1 will have address of data array
- The offset 8 permits to get second element of data array.

3. **Immediate**

Many times, a program will use a constant in an operation for example incrementing an index to point to the next element of an array. This constant field size is 16 bit that permits $\pm 2^{15}$(32768).

Example: add i $s3,$s3,4 # $s3 =$s3+4

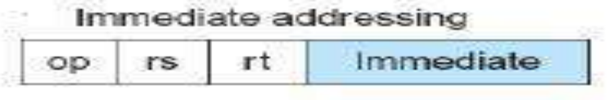
add i $s3,$s3,1 #$s3 =$s3+1

## ADDRESSING MODES OF MIPS

The different ways in which the address of the operand is specified in an instruction is called addressing mode. The addressing modes vary from one computer to another MIPS processor supports the following five addressing modes.

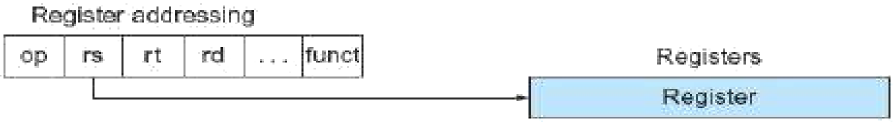
1. Immediate addressing, where the operand is a constant within the instruction itself.
2. Register addressing, where the operand is a register.
3. Base or displacement addressing, where the operand is at the memory location whose address is the sum of a register and a constant in the instruction.
4. PC-relative addressing, where the branch address is the sum of the PC and a constant in the instruction.
5. Pseudodirect addressing, where the jump address is the 26 bits of the instruction concatenated with the upper Four bits of the PC.

***1. Immediate addressing:*** *The operand is a constant within the* instructionitself. i.e. The operand is specified in the instruction itself.



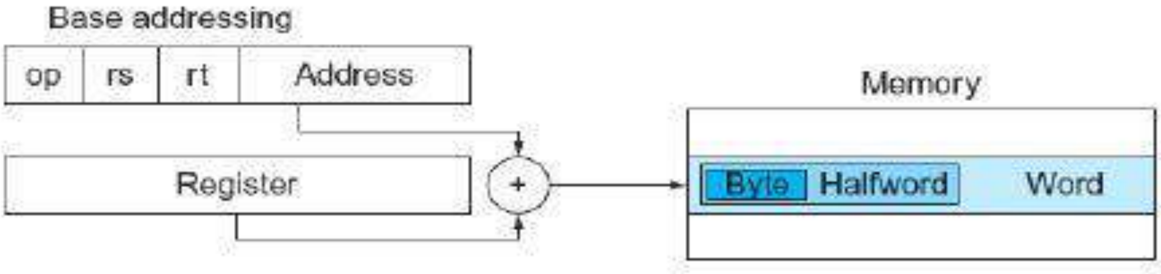
*Example: Add i $t1, $s1,d*

***2. Register addressing:*** *The operand is in a CPU register. The register is specified in the instruction.*



Example: add $t1, $S1, $S2

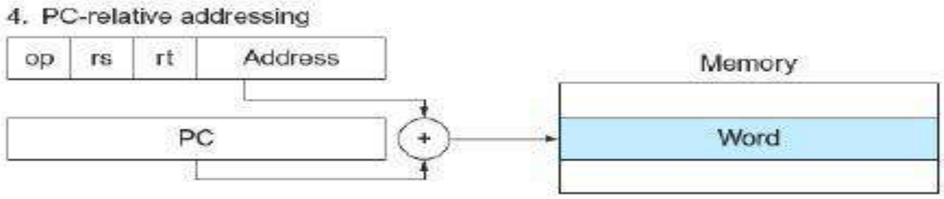
3. **Base or displacement addressing:** The operand is at the memory location whose address is the sum of a register and a constant in the instruction.



Here the operand is a memory location whose address is the sum of register (base register) and a constant (offset/displacement)

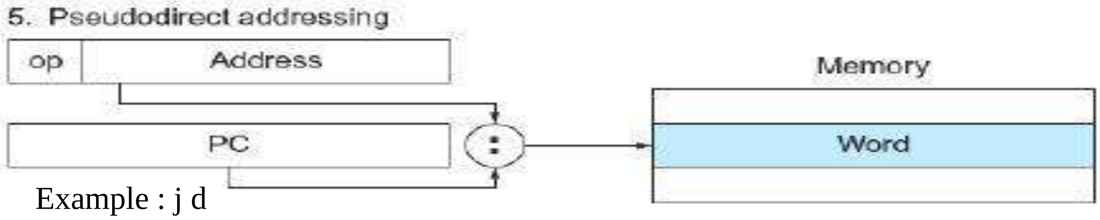
Example: lw $s1, 20($s2)

4. ***PC-relative addressing:*** *The branch address is the sum of the PC and a* constant in the instruction.



Example :beq $rs, $rt, d

5. *Pseudo-direct addressing: The jump address is the 26 bits of the* instruction concatenated withthe upper Four bits of the PC.



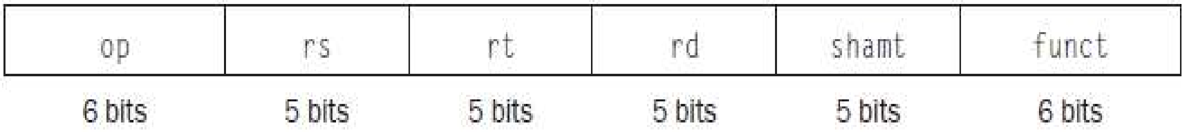
Example : j d

**<u>REPRESENTATION OF INSTRUCTION (INSTRUCTION FORMAT)</u>**

1. **<u>R-TYPE : Used in arithmetic Instruction</u>**
2. **<u>I-type : used in data transfer instruction and conditional branch instruction</u>**
3. **<u>J-type: used in unconditional branch instructions.</u>**

**<u>R-TYPE:</u>**

This instruction format is called R type and used for arithmetic and logic instruction.



| op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

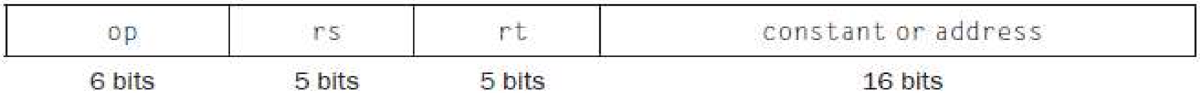
The meaning of each field is given below

- op : Basic operation of the instruction (op code)
- rs : The first register source field
- rt : The second register source field
- rd : The register destination field get the result
- shamt: shift amount for shift instruction and zero for other instruction
- funct : called function code specific variant of the operation in the op field

All register field are 5 bit to refer any one of the 32 MIPS registers.

Example : add $t0, $s1, $s2 # $t0=$s1+$s2

**I-TYPE:** This is used for immediate as well as data transfer instruction.

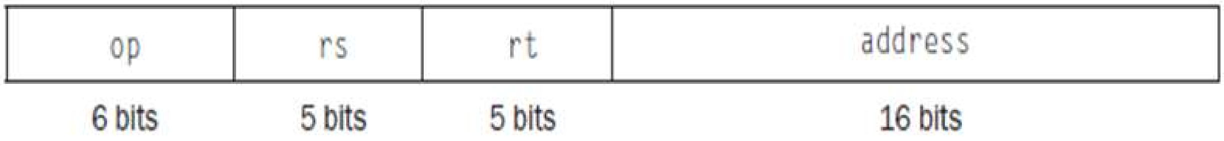


| op | rs | rt | constant or address |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

The meaning of each field is given below

- op : op code
- rs : The first register source field
- rt : register destination for load & source for store
- constant or address : constant is used in immediate instruction and address is used in data transfer instruction

This format is used for branch instruction and it is similar to I type.



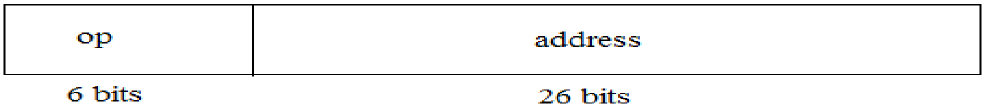
| op | rs | rt | address |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

The meaning of each field is given below

- op : op code
- rs : The first register source field
- rt : Second register source field
- address : address is used in branch instruction

Example :beq $s1, $s2,d #if($s1=$s2) goto (PC+4+(4*d))

iii) J-TYPE: This instruction format is used for jump instruction called J type.



| op | address |
|---|---|
| 6 bits | 26 bits |

The meaning of each field is given below

WHERE op : op code and Address : used in jump

Example : j d # go to 4*d

# UNIT II
# ARITHMETIC OPERATIONS

**ALU - Addition and subtraction – Multiplication – Division – Floating Point operations  Subword parallelism**.

## ARITHEMETIC AND LOGICAL UNIT

The **arithmetic logic unit (ALU)** performs

- The arithmetic operations like addition and subtraction.
- Logical operations like AND and OR.

Because the MIPS word is 32 bits wide, we need a 32-bit-wide ALU. Let's assume that we will connect 32 1-bit ALUs to create the desired ALU.

## ALU-ADDITION

### 1-Bit MIPS ALU

### 1.1 HALF ADDER

**Truth table :**CarryOut = (b · CarryIn) + (a · CarryIn) + (a · b) + (a · b · CarryIn)

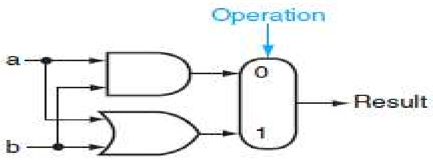


**FIGURE C.5.1 The 1-bit logical unit for AND and OR.**

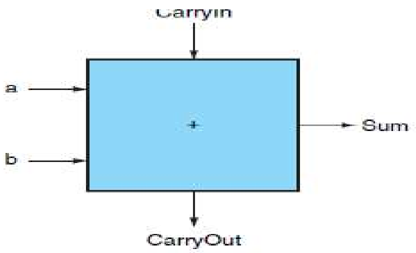


**FIGURE C.5.2 A 1-bit adder.** This adder is called a full adder; it is also called a (3,2) adder because it has 3 inputs and 2 outputs. An adder with only the a and b inputs is called a (2,2) adder or half-adder.

**FULL ADDER**

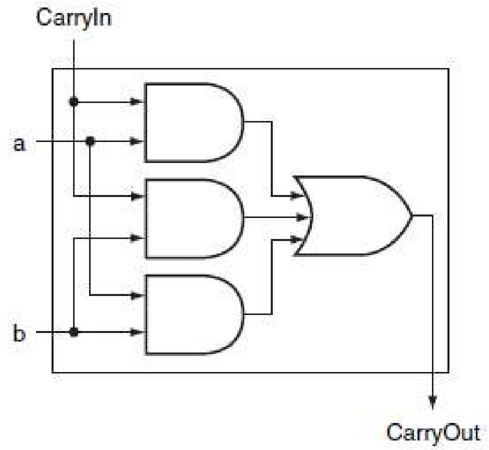
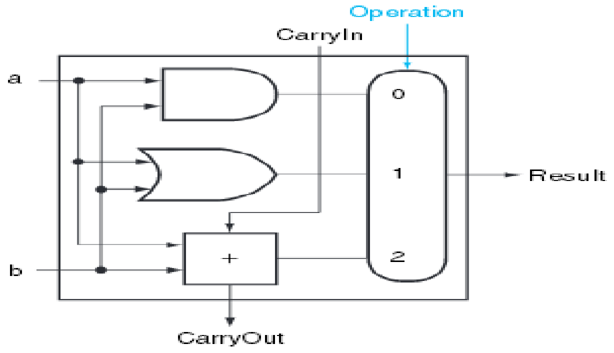


**FIGURE C.5.5 Adder hardware for the CarryOut signal.** The rest of the adder hardware is the logic for the Sum output given in the equation on this page.

The Sum bit is set when exactly one input is 1 or when all three inputs are 1. The Sum results in a complex Boolean equation (recall that $\bar{a}$ means NOT a):

$$\text{Sum} = (a \cdot \bar{b} \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot b \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot \bar{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})$$



The multiplexer selects AND or OR or adder output based on the value of operation 0(00) or 1 (01) or 2(10).The adder takes three input (two digits & one carry from previous stage) and produce sum and carry outputs, and it is called full adder. The truth table is shown below.

| Inputs | | | Outputs | | |
|---|---|---|---|---|---|
| a | b | CarryIn | CarryOut | Sum | Comments |
| 0 | 0 | 0 | 0 | 0 | $0 + 0 + 0 = 00_{two}$ |
| 0 | 0 | 1 | 0 | 1 | $0 + 0 + 1 = 01_{two}$ |
| 0 | 1 | 0 | 0 | 1 | $0 + 1 + 0 = 01_{two}$ |
| 0 | 1 | 1 | 1 | 0 | $0 + 1 + 1 = 10_{two}$ |
| 1 | 0 | 0 | 0 | 1 | $1 + 0 + 0 = 01_{two}$ |
| 1 | 0 | 1 | 1 | 0 | $1 + 0 + 1 = 10_{two}$ |
| 1 | 1 | 0 | 1 | 0 | $1 + 1 + 0 = 10_{two}$ |
| 1 | 1 | 1 | 1 | 1 | $1 + 1 + 1 = 11_{two}$ |

The logic expression for sum is

$$\text{Sum} = (a \cdot \bar{b} \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot b \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot \bar{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})$$

The logical expression for carryout

$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b)$$

is

**32 BITS ADDER**

32 bit ALU can be constructed by connecting 32 1-bit ALU in cascade. 1-bit ALU with logical unit for AND

and OR and adder is shown below.

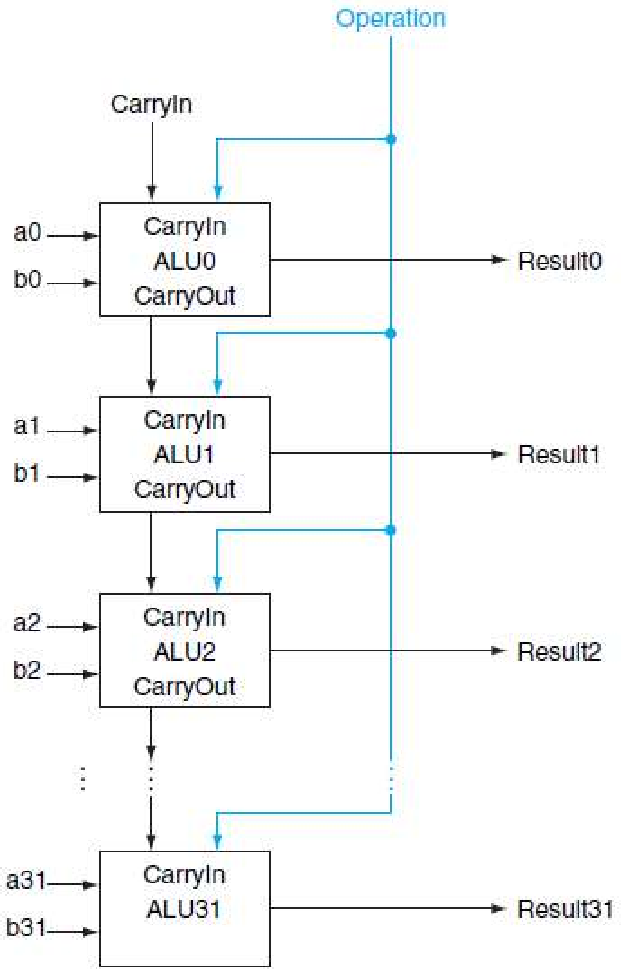


**FIGURE C.5.7 A 32-bit ALU constructed from 32 1-bit ALUs.** CarryOut of the less significant bit is connected to the CarryIn of the more significant bit. This organization is called ripple carry.

## ALU MIPS SUBTRACTION

Subtraction is the same as adding the negative version of an operand and this is how adder performs subtraction. For negating a 2's complement number is to invert each bit and add 1. To invert each bit, 2:1 multiplexer chooses between b and b as in **Fig .3**

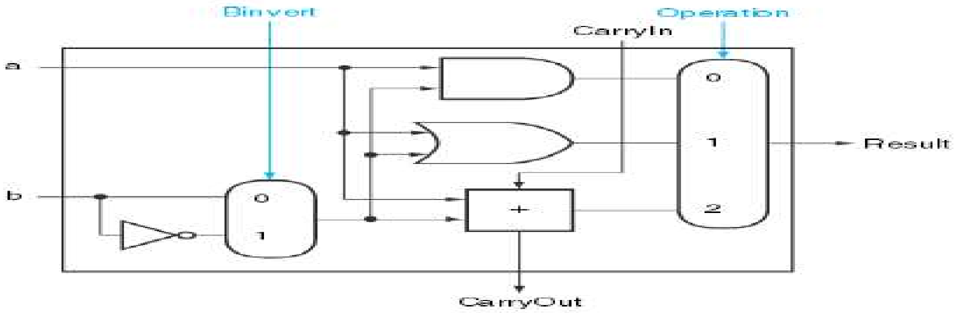
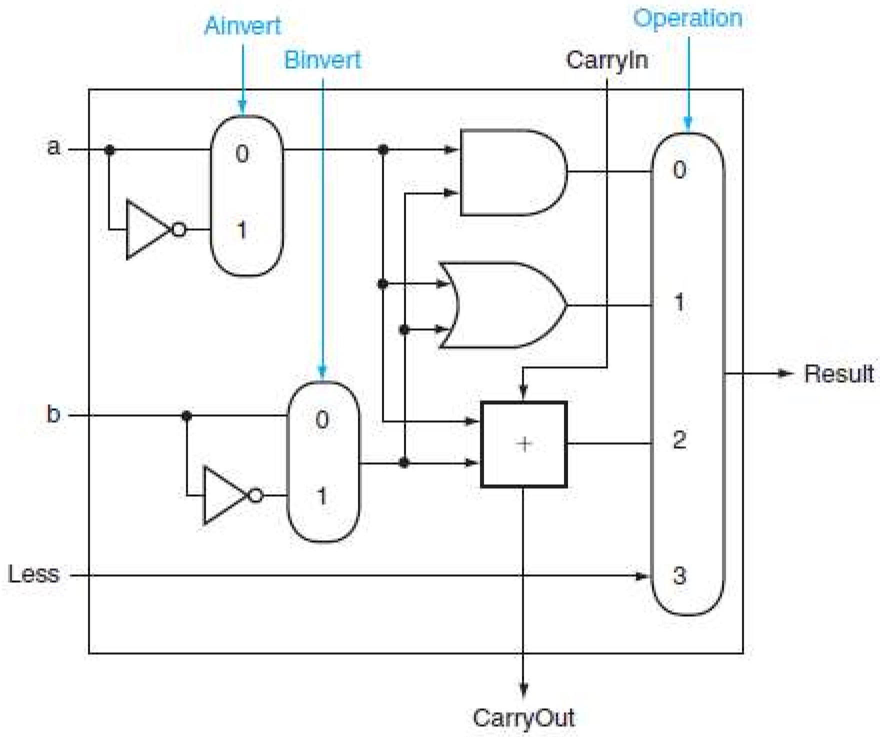


**Fig 3 : 1-Bit ALU with AND, OR, ADD and SUBTRACT**



These four operations—add, subtract, AND, OR—are found in the ALU of almost every computer, and the operations of most MIPS instructions can be performed by this ALU. But the design of the ALU is incomplete. One instruction that still needs support is the set on less than instruction (slt). Recall that the operation produces 1 if rs < rt, and 0 otherwise. Consequently, slt will set all but the least significant bit to 0, with the least significant bit set according to the comparison. For the ALU to perform slt, we first need to expand the three-input multiplexor in above figure  to add an input for the slt result. We call that new input Less and use it only for slt.

From the description of slt above, we must connect 0 to the Less input for the upper 31 bits of the ALU, since those bits are always set to 0. What remains to consider is how to compare and set the least significant bit for set on less than instructions.

What happens if we subtract b from a?

If the difference is negative, then a < b since $(a - b) < 0 \Rightarrow ((a - b) + b) < (0 + b) \Rightarrow a < b$

We want the least significant bit of a set on less than operation to be a 1 if a < b;

that is, a 1 if a − b is negative and a 0 if it’s positive.

This desired result corresponds exactly to the sign bit values:

1 means negative and 0 means positive.

Following this line of argument, we need only connect the sign bit from the adder output to the least significant bit to get set on less than.

Unfortunately, the Result output from the most significant ALU bit in the top of Figure C.5.10 for the slt operation is not the output of the adder; the ALU output for the slt operation is obviously the input value Less.

Thus, we need a new 1-bit ALU for the most significant bit that has an extra output bit: the adder output. The bottom drawing of Figure C.5.10 shows the design, with this new adder output line called Set, and used only for slt.

As long as we need a special ALU for the most significant bit, we added the overflow detection logic since it is also associated with that bit.

**32 bits**

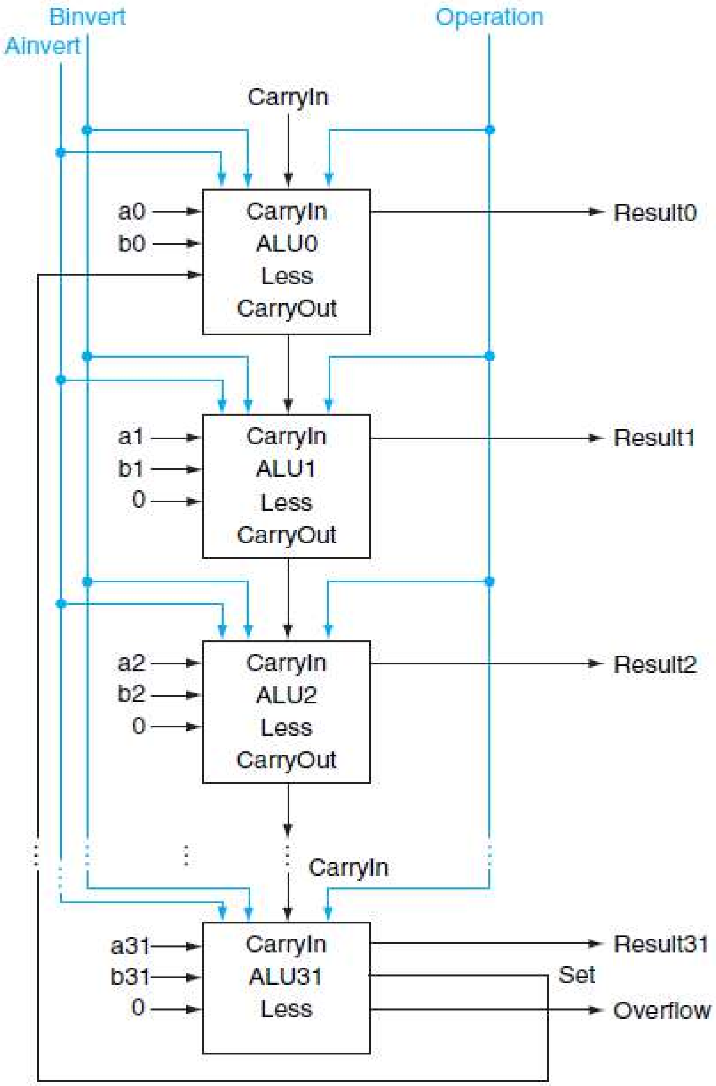
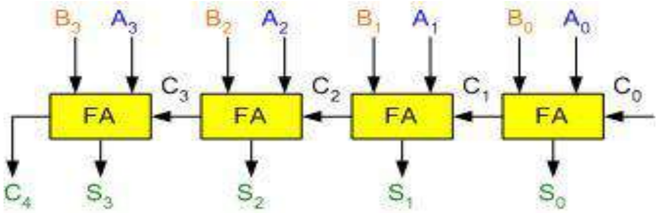


**FIGURE C.5.11 A 32-bit ALU constructed from the 31 copies of the 1-bit ALU in the top of Figure C.5.10 and one 1-bit ALU in the bottom of that figure.** The Less inputs are connected to 0 except for the least significant bit, which is connected to the Set output of the most significant bit. If the ALU performs a – b and we select the input 3 in the multiplexor in Figure C.5.10, then Result = 0 . . . 001 if a < b, and Result = 0 . . . 000 otherwise.

**Subtraction**

## Carry Look Ahead Adder:

In ***ripple carry adders***, the carry propagation time is the major speed limiting factor as seen in the previous lesson.



Most other arithmetic operations, e.g. multiplication and division are implemented using several add/subtract steps. Thus, improving the speed of addition will improve the speed of all other arithmetic operations.

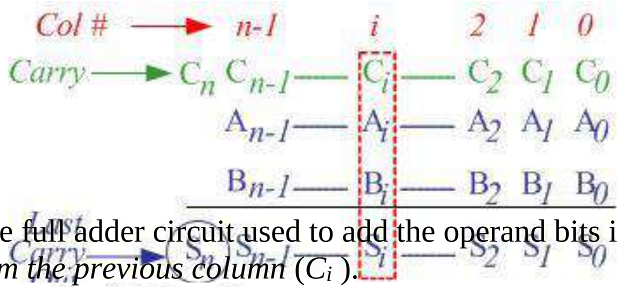
Accordingly, reducing the carry propagation delay of adders is of great importance. Different logic design approaches have been employed to overcome the carry propagation problem.One widely used approach employs the principle of ***carry look-ahead*** solves this problem by calculating the carry signals in advance, based on the input signals.

This type of adder circuit is called as ***carry look-ahead adder (CLA adder).*** It is based on the fact that a carry signal will be generated in two cases:

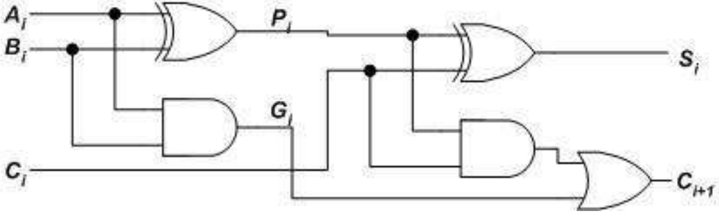
**(1)** when both bits $A_i$ and $B_i$ are 1, or

**(2)** when one of the two bits is 1 and the carry-in (carry of the previous stage) is 1.

**To understand the carry propagation problem, let's consider the case of adding two *n-bit* numbers *A* and *B*.**



The Figure shows the full adder circuit used to add the operand bits in the $i^{th}$ column*; namely $A_i$ & $B_i$ and the carry bit coming from the previous column* ($C_i$ ).



**In this circuit, the 2 internal signals $P_i$ and $G_i$ are given by:**

$Pi = Ai \oplus Bi$ **……………………..(1)**

$G_i = A_i B_i$ **……………………(2)**

**The output sum and carry can be defined as :**

$S_i = P_i \oplus C_i$ **……………………(3)**

$C_{i+1} = G_i + P_i C_i$ **…………(4)**

**$G_i$** is known as the **carry Generate** signal since a carry ($C_{i+1}$) is generated whenever $G_i$ =1, regardless of the input carry ($C_i$).**$P_i$** is known as the **carry propagate** signal since whenever $P_i$ =1, the input carry is propagated to the output carry, i.e., $C_{i+1}$. = $C_i$ (note that whenever $P_i$ =1, $G_i$ =0).Computing the values of $P_i$ and $G_i$ only depend on the input operand bits ($A_i$ & $B_i$) as clear from the Figure and equations. Thus, these signals settle to their **steady-state value** after the propagation through their respective gates. Computed values of **all** the $P_i$'s are valid one XOR-gate delay after the operands A and B are made valid. Computed values of **all** the $G_i$'s are valid one AND-gate delay after the operands A and B are made valid.

**The Boolean expression of the carry outputs of various stages can be written as follows**:

$\mathbf{C_1 = G_0 + P_0C_0}$

$\mathbf{C_2 = G_1 + P_1C_1 = G_1 + P_1 (G_0 + {}_0C_0) = G_1 + P_1G_0 + P_1P_0C_0}$

$\mathbf{C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0}$

$\mathbf{C_4 = G_3 + P_3C_3 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0}$

In general, the $i^{th.}$ carry output is expressed in the form $\mathbf{C}_{\boldsymbol{i}}$ = $F_i$ (P's, G's , $\mathbf{C_0}$).

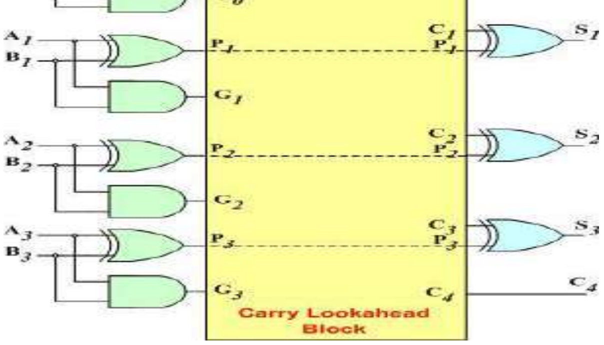
In other words, each carry signal is expressed as a direct SOP function of $\mathbf{C_0}$ rather than its preceding carry signal.Since the Boolean expression for each output carry is expressed in SOP form, it can be implemented in two-level circuits.The 2-level implementation of the carry signals has a propagation delay of 2 gates, i.e., 2τ.

**The 4-bit carry look-ahead (CLA) adder consists of 3 levels of logic:**

**First level:** Generates all the P & G signals. Four sets of P & G logic (each consists of an XOR gate and an AND gate). Output signals of this level (P's & G's) will be valid after 1τ.

**Second level:** The Carry Look-Ahead (CLA) logic block which consists of four 2-level implementation logic circuits. It generates the carry signals ($C_1$, $C_2$, $C_3$, and $C_4$) as defined by the above expressions. Output signals of this level ($C_1$, $C_2$, $C_3$, and $C_4$) will be valid after 3τ.

**Third level:** Four XOR gates which generate the sum signals ($S_i$) ($S_i = P_i \oplus C_i$). Output signals of this level ($S_0$, $S_1$, $S_2$, and $S_3$) will be valid after 4τ.



# MULTIPLICATION

## The basics multiplication (The traditional Approach)

Multiplying $1000_{ten}$ by $1001_{ten}$

Multiplicand $1000_{ten}$

Multiplier x $\underline{1001_{ten}}$

1000

0000

0000

$\underline{1000}$

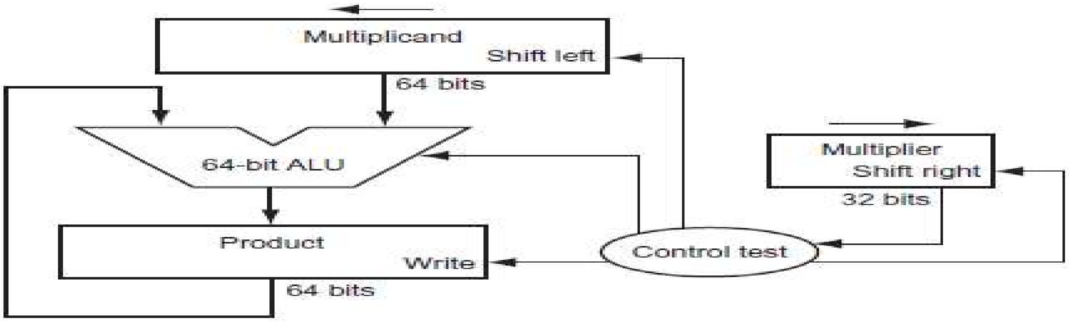
Product $1001000_{ten}$

- The first operand is called the **multiplicand** and the second the **multiplier**. The final result is called the **product**.
- The length of the multiplication of an *n*-bit multiplicand and an *m*-bit multiplier is a product that is *n* + *m* bits long. That is, *n* + *m* bits are required to represent all possible products. Hence, like add, multiply must cope with overflow because we frequently want a 32-bit product as the result of multiplying two 32-bit numbers.

## MULTIPLICATION HARDWARE

### The following fig shows hardware for 32 bit multiplication

In the Hardware Diagram, Let's assume that the multiplier is in the 32-bit Multiplier register and that the 64-bit Product register is initialized to 0. It's clear that we will need to move the multiplicand left one digit each step, as it may be added to the intermediate products. Over 32 steps, a 32-bit multiplicand would move 32 bits to the left. Hence, we need a 64-bit Multiplicand register, initialized with the 32-bit multiplicand in the right half and zero in the left half. This register is then shifted left 1 bit each step to align the multiplicand with the sum being accumulated in the 64-bit Product register.



**Multiplication Hardware Diagram**

## MULTIPLICATION ALGORITHM

**Three basic steps are required for each bit of multiplier.**

Step1. The least significant bit of multiplier is examined ,if 1 add multiplicand with product otherwise no add

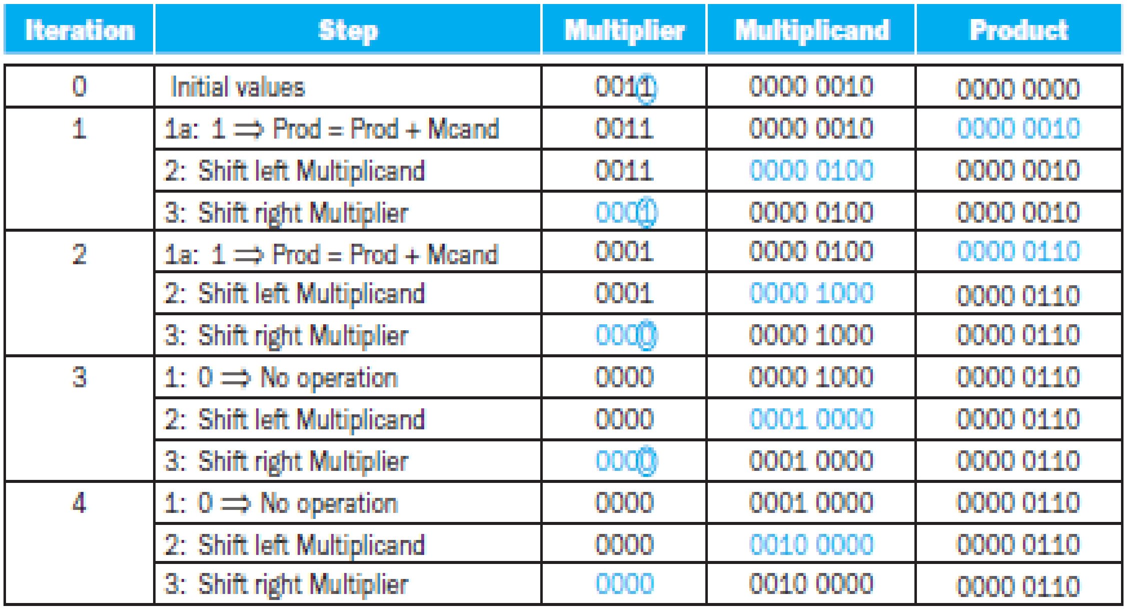
Step2. Multiplicand register is shifted left by one bit

Step3. Multiplier register is shifted right by one bit

These 3 steps are repeated 32 times and at the end product is available in product register.
The hardware is further optimized to adder with 32 bit and multiplicand register with 32 bit,no multiplier register and product register of 64 bit.

**Example**

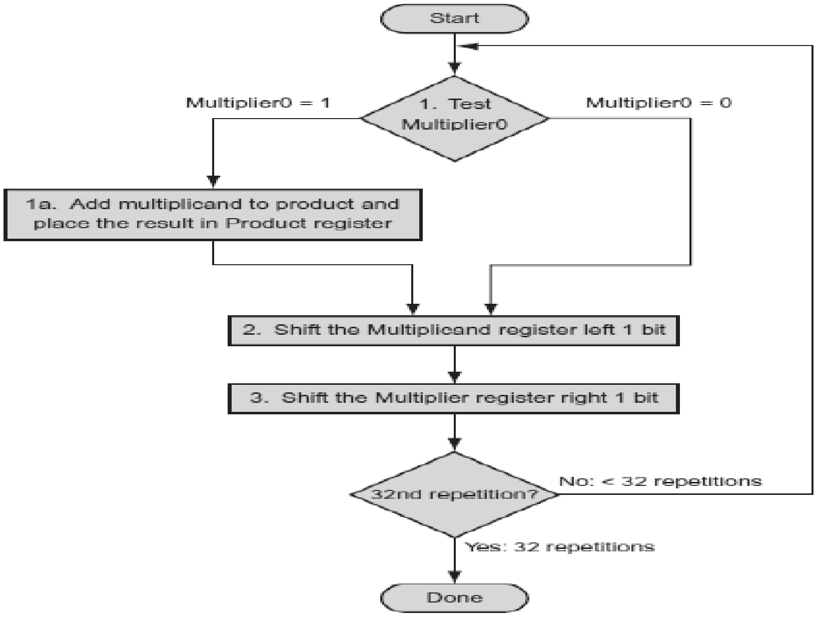
Using 4-bit numbers to save space, multiply **2ten X 3ten, or 0010two X 0011two**. Figure 3.6 shows the value of each register for each of the steps labeled according to Figure 3.4, with the final value of 0000 0110two or 6ten.



| Iteration | Step | Multiplier | Multiplicand | Product |
|---|---|---|---|---|
| 0 | Initial values | 0011 | 0000 0010 | 0000 0000 |
| 1 | 1a: 1 ⇒ Prod = Prod + Mcand | 0011 | 0000 0010 | 0000 0010 |
| | 2: Shift left Multiplicand | 0011 | 0000 0100 | 0000 0010 |
| | 3: Shift right Multiplier | 0001 | 0000 0100 | 0000 0010 |
| 2 | 1a: 1 ⇒ Prod = Prod + Mcand | 0001 | 0000 0100 | 0000 0110 |
| | 2: Shift left Multiplicand | 0001 | 0000 1000 | 0000 0110 |
| | 3: Shift right Multiplier | 0000 | 0000 1000 | 0000 0110 |
| 3 | 1: 0 ⇒ No operation | 0000 | 0000 1000 | 0000 0110 |
| | 2: Shift left Multiplicand | 0000 | 0001 0000 | 0000 0110 |
| | 3: Shift right Multiplier | 0000 | 0001 0000 | 0000 0110 |
| 4 | 1: 0 ⇒ No operation | 0000 | 0001 0000 | 0000 0110 |
| | 2: Shift left Multiplicand | 0000 | 0010 0000 | 0000 0110 |
| | 3: Shift right Multiplier | 0000 | 0010 0000 | 0000 0110 |

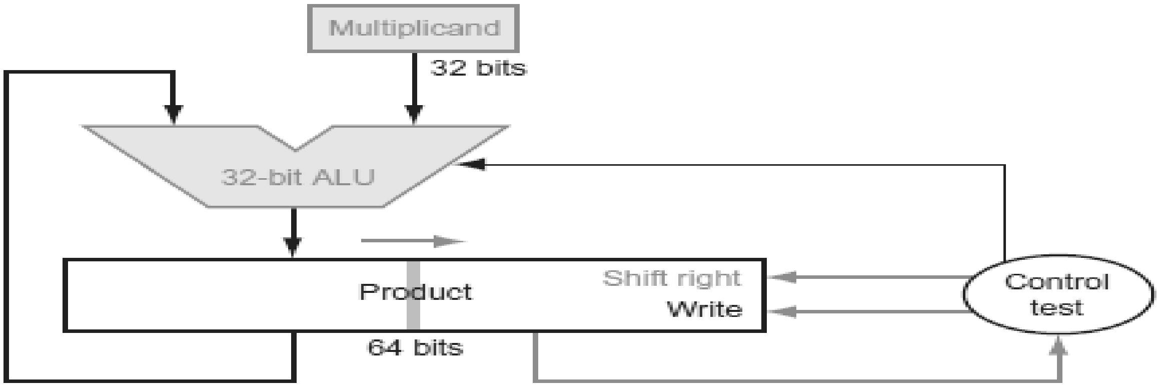
**FIGURE 3.6 Multiply example using algorithm in Figure 3.4.** The bit examined to determine the next step is circled in color.

**MULTIPLICATION FLOWCHART**


Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

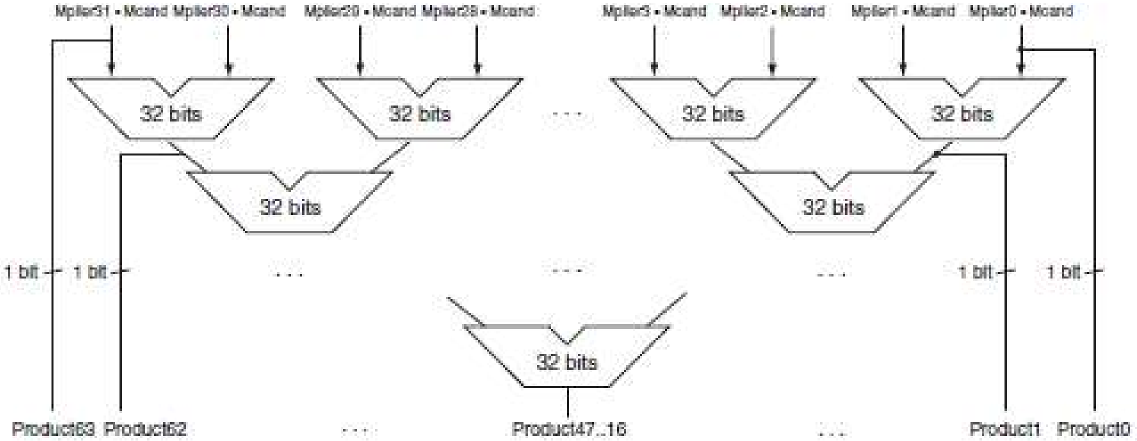
## REFI NED VERSION OF THE MULTIPLICATION HARDWARE

Compare with the first version. The Multiplicand register, ALU, and Multiplier register are all 32 bits wide, with only the Product register left at 64 bits. Now the product is shifted right. The separate Multiplier register also disappeared. The multiplier is placed instead in the right half of the Product register.



## FASTER MULTIPLICATION:

- **Moore's Law** has provided so much more in resources that hardware designers can now build much faster multiplication hardware. Whether the multiplicand is to be added or not is known at the beginning of the multiplication by looking at each of the 32 multiplier bits.
- Faster multiplications are possible by essentially providing one 32-bit adder for each bit of the multiplier: one input is the multiplicand ANDed with a multiplier bit, and the other is the output of a prior adder.
- A straightforward approach would be to connect the outputs of adders on the right to the inputs of adders on the left, making a stack of adders 32 high.
- An alternative way to organize these 32 additions is in a parallel tree shows. Instead of waiting for 32 add times, we wait just the log2 (32) or five 32-bitadd times.



## DIVISION

### Basic Division Operation

The hardware division is identical to school division algorithm.

**Example : 7 ÷ 2; Quotient = 3; Remainder = 1**

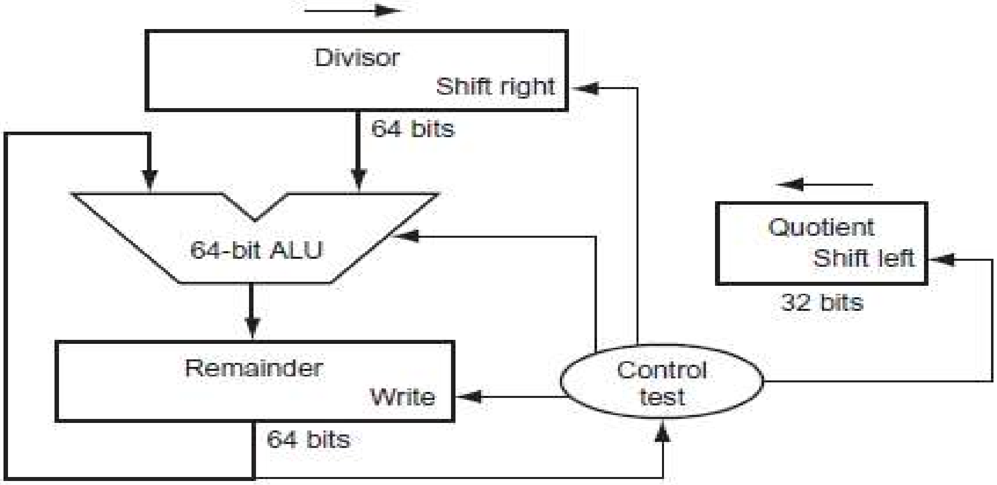
```
                      1001ten      Quotient
Divisor 1000ten | 1001010ten       Dividend
                 -1000
                    10
                    101
                    1010
                   -1000
                      10ten        Remainder
```

Divide's two operands, called the **dividend** and **divisor**, and the result, called the **quotient**, are accompanied by a second result, called the **remainder**. Here is another way to express the relationship between the components:

Dividend _ Quotient _ Divisor _ Remainder

where the remainder is smaller than the divisor.

### DIVISION HARDWARE



To start with 32 bit quotient register set to 0. Since each iteration divisor need to be shifted right by one digit, start with the divisor is placed in the left half of 64 bit DIVISIOR register and shift right by 1 bit each step to align with the dividend. The remainder register is initialized with dividend

### DIVISION ALGORITHM

Three basic steps are.

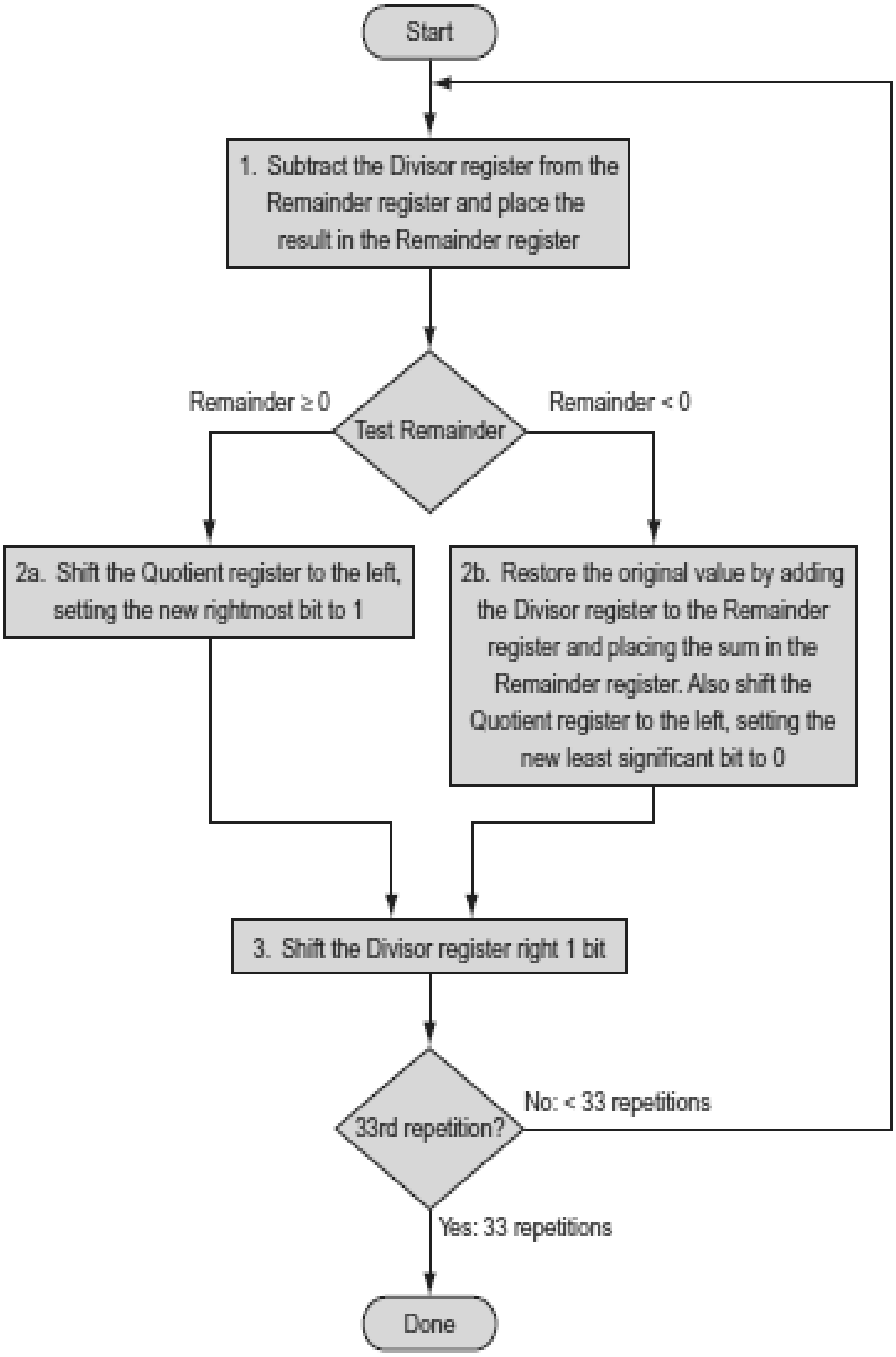
**Step1:** Subtract divisor register from remainder register place the result in remainder register.

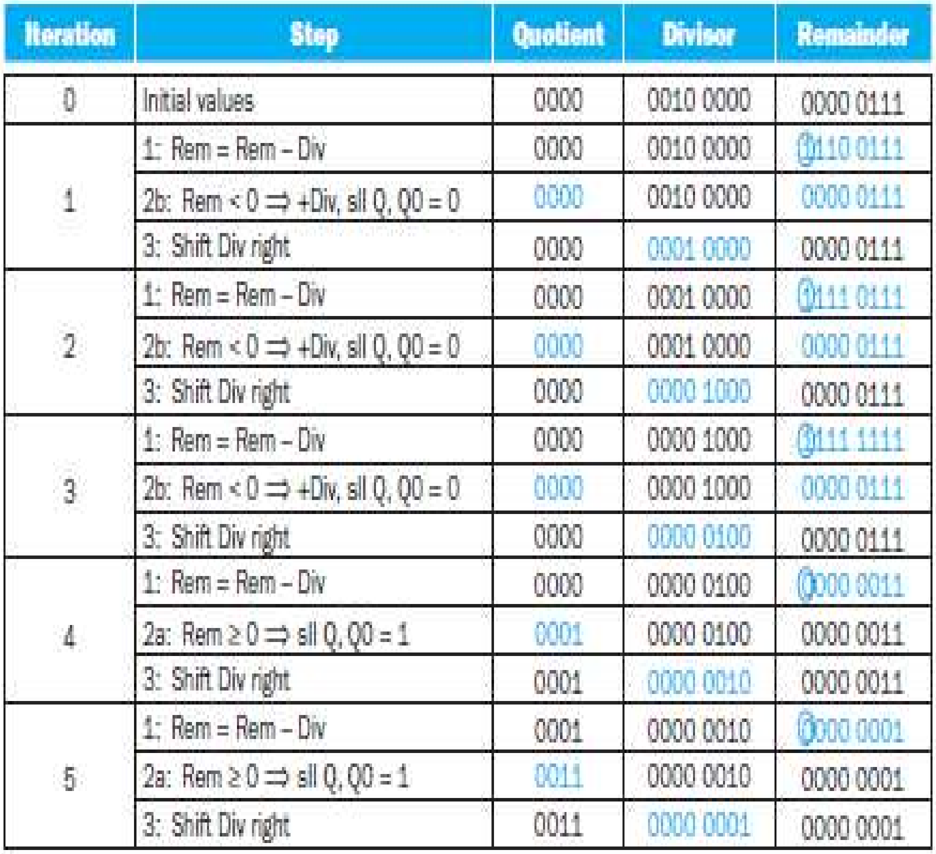
**Step1 A**: If the reminder is +ve. Shift the quotient register to the left and set Qo=1

**Step2B**: If the remainder is –ve. Restore the original value by adding the DIVISOR register to the remainder register and place the result in remainder register and also shift the quotient register to the left and set Qo = 0.

**Step 3:** Shift the divisor right by 1 bit. Repeat these three steps 33 times and at the end remainder and quotient are available in the corresponding registers. This is called **Restoring Division**.

## DIVISION FLOW CHART


Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
33rd repetition?
No: < 33 repetitions
Yes: 33 repetitions
Done



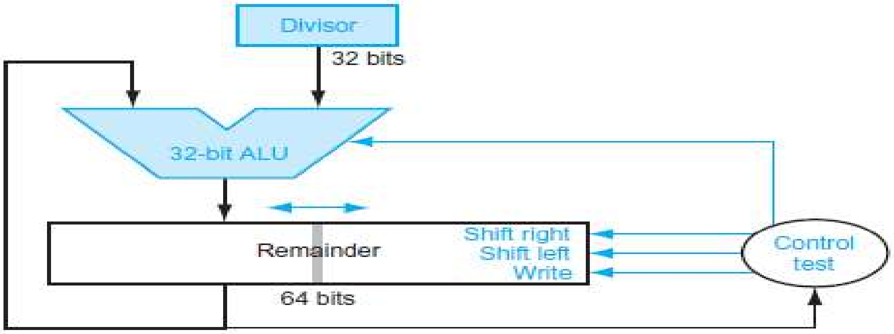
| Iteration | Step | Quotient | Divisor | Remainder |
|---|---|---|---|---|
| 0 | Initial values | 0000 | 0010 0000 | 0000 0111 |
| 1 | 1: Rem = Rem – Div | 0000 | 0010 0000 | 1110 0111 |
| | 2b: Rem < 0 ⇒ +Div, sll Q, Q0 = 0 | 0000 | 0010 0000 | 0000 0111 |
| | 3: Shift Div right | 0000 | 0001 0000 | 0000 0111 |
| 2 | 1: Rem = Rem – Div | 0000 | 0001 0000 | 1111 0111 |
| | 2b: Rem < 0 ⇒ +Div, sll Q, Q0 = 0 | 0000 | 0001 0000 | 0000 0111 |
| | 3: Shift Div right | 0000 | 0000 1000 | 0000 0111 |
| 3 | 1: Rem = Rem – Div | 0000 | 0000 1000 | 1111 1111 |
| | 2b: Rem < 0 ⇒ +Div, sll Q, Q0 = 0 | 0000 | 0000 1000 | 0000 0111 |
| | 3: Shift Div right | 0000 | 0000 0100 | 0000 0111 |
| 4 | 1: Rem = Rem – Div | 0000 | 0000 0100 | 0000 0011 |
| | 2a: Rem ≥ 0 ⇒ sll Q, Q0 = 1 | 0001 | 0000 0100 | 0000 0011 |
| | 3: Shift Div right | 0001 | 0000 0010 | 0000 0011 |
| 5 | 1: Rem = Rem – Div | 0001 | 0000 0010 | 0000 0001 |
| | 2a: Rem ≥ 0 ⇒ sll Q, Q0 = 1 | 0011 | 0000 0010 | 0000 0001 |
| | 3: Shift Div right | 0011 | 0000 0001 | 0000 0001 |

**FIGURE 3.10 Division example using the algorithm in Figure 3.9.** The bit examined to determine the next step is circled in color.

## AN IMPROVED VERSION OF THE DIVISION HARDWARE.

The Divisor register, ALU, and Quotient register are all 32 bits wide, with only the Remainder register left at 64 bits the ALU and Divisor registers are halved and the remainder is shifted left.

The hardware is further optimized with divisor register and Adder 32-bit wide with only reminder register left at 64 bits. This version also combines the Quotient register with the right half of the Remainder register.



# FLOATING POINT OPERATIONS

## IEEE standards for Binary Arithmetic operations for floating point numbers

Includes Fraction and Exponent

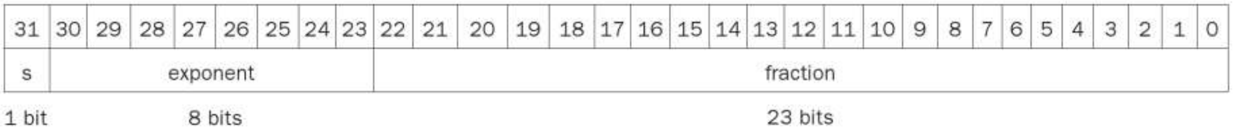
**Representation of ARM(Advanced RISC machine) Floating-point Number**

S →Sign of the floating point number

E→ Value of the 8-bit exponent field

F→ 23-bit number

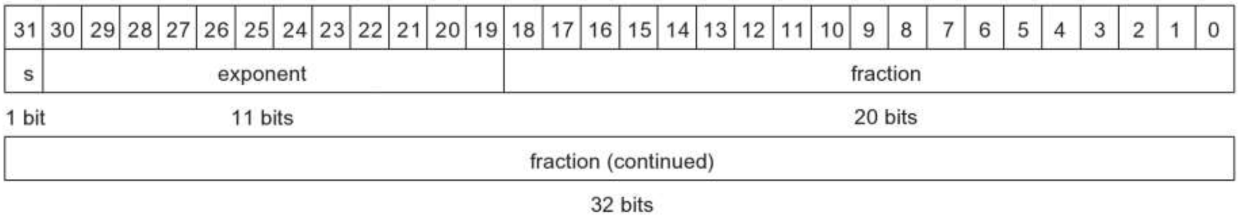
- **Single Precision Format**



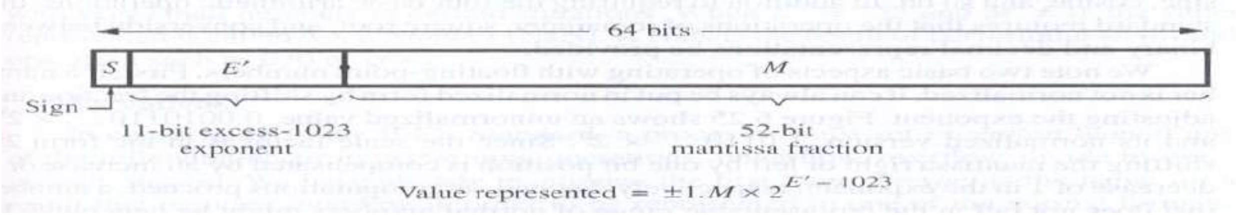
**Format of Floating point number**

$$(-1)^{S} \times F \times 2^{E}$$

- Overflow Interrupt occurs in Floating-point Arithmetic
- Overflow→ Positive Exponent is too large to be represented in the Exponent field
- Underflow→ Negative Exponent is too large to be represented in the Exponent field
- To reduce the chances of Underflow and overflow
- Another format is used has a larger exponent
- Double Precision floating-point arithmetic
- Both called IEEE 754 Floating-point Standard invented in 1980
- **Double Precision floating-point**
  - S →Sign of the floating point number
  - E→ Value of the 11-bit exponent field
  - F→ 32-bit number



**(Or)**


64 bits
S
E′
M
Sign
11-bit excess-1023 exponent
52-bit mantissa fraction
Value represented = ±1.M × 2^(E′ − 1023)

Exponent bias is 127 for single-precision

1023 for double-precision

Representation

- sign, exponent, significand

$$(-1)^{\text{sign}} \times \text{significand} \times 2^{\text{exponent}}$$

- more bits for significand gives more accuracy
- more bits for exponent increases range

*Exponent => range, significand => precision*

**Example**

IEEE 754 binary Representation of the number $-0.75_{10}$ in Single and Double Precision

Number $-0.75_{10}$ is also $-3/4_{\text{ten}}$ or $-3/2^2_{\text{ten}}$

Represented in binary fraction $-11/2^2_{\text{ten}}$

Scientific Notation $-0.11_{\text{two}} \times 2^0$

Normalized into $-1.1_{\text{two}} \times 2^{-1}$

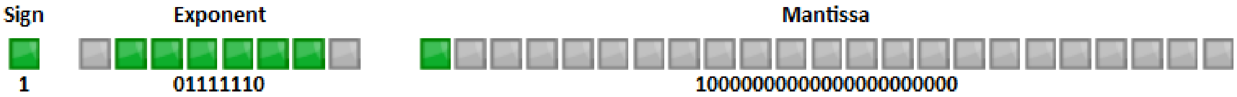
$-1.1_{\text{two}} \times 2^{-1}$

**Single precision representation**

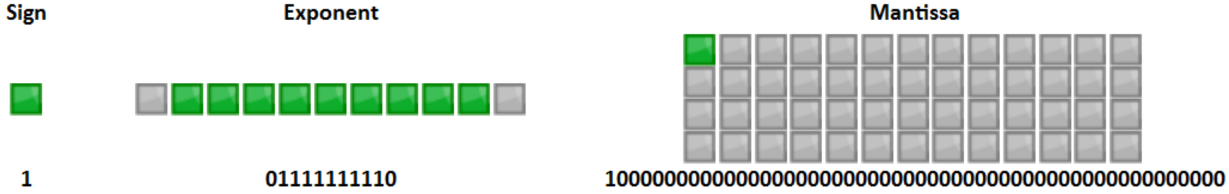
$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent}-\text{Bias})}$$

Subtracting the bias 127 from the exponent

**Single:** $-1 + 127 = 126 = 01111110_2$



**Double:** $-1+1023=1022= 01111111110_2$



**Underflow and Overflow in single precision**

- In single precision floating point representation exponent is represented in excess 127
- Total exponent field range is -127 to +128
- Since +128 and -127 are used for special case, real exponent range is -126 to +127
- If the exponent exceeds +127 it is called overflow
- If the exponent is < -126 it is called underflow

**Underflow and Overflow in double precision**

- In double precision floating point representation exponent is represented in excess 1023
- Total exponent field range is -1023 to +1024
- Since -1023 and +1024 are used for special case, real exponent range is -1022 to +1023
- If the exponent exceeds +1023 it is called overflow
- If the exponent is < -1022 it is called underflow

**NORMALIZATION**

**Normalization of Floating Point Number**

In floating point the number is always put in normalized form. By convention when the decimal point is placed to the right of first significant digit, the number is said to be normalized.

**Example of unnormalized number**

$.0010110 \times 2^{9}$

Normalized Form $= 1.0110 \text{ X } (2^{9} \text{ X } 2^{-3})$

$= 1.0110 \text{ X } 2^{6}$

**<u>FLOATING POINT ADDITION</u>**

**<u>FLOATING POINT ADDITION ALGORITHM</u>**

**<u>The following steps gives the algorithm for IEEE floating point addition(also indicated in the flow chart..</u>**

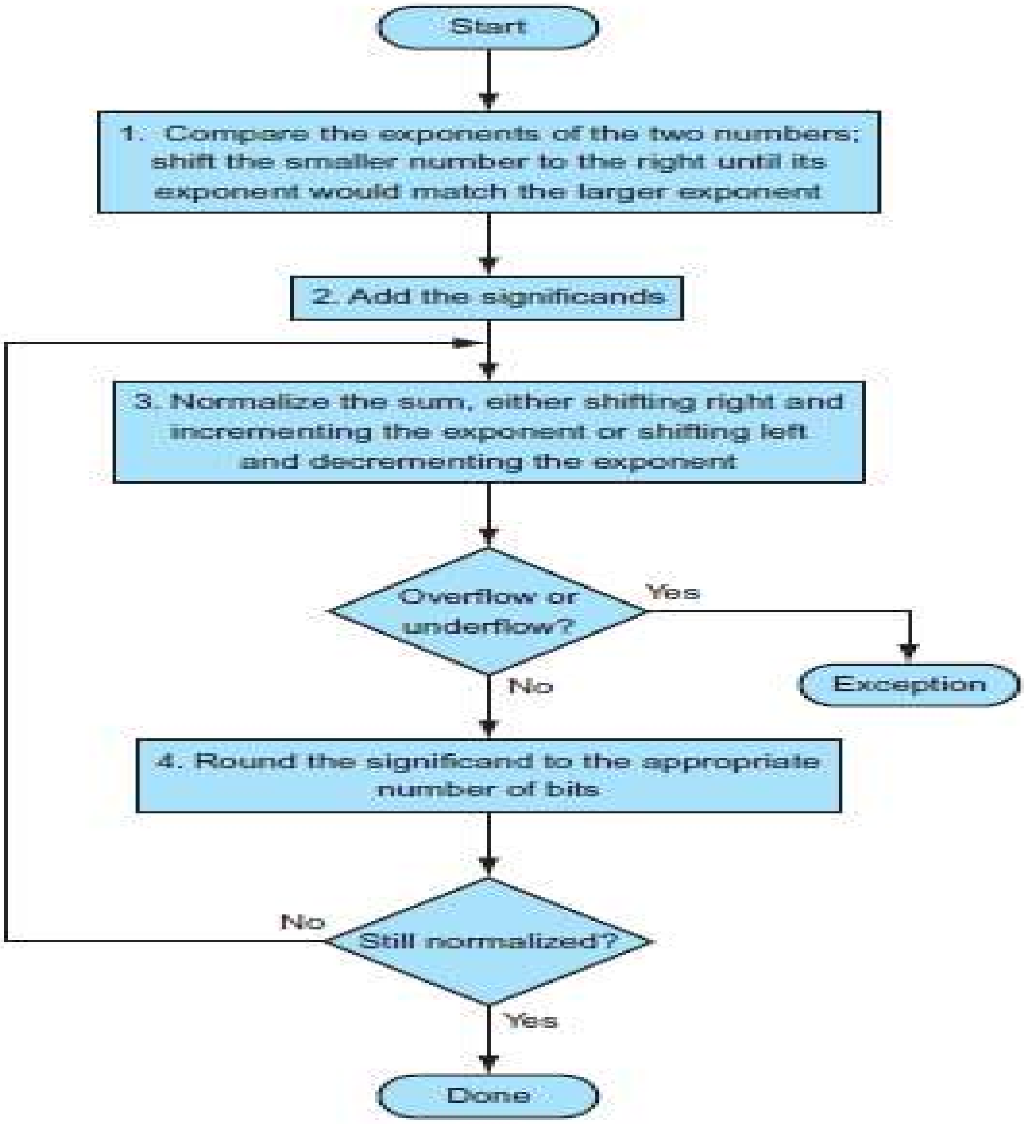
**Step 1:**Compare the exponents of two numbers and shift the number with smaller exponent, until the exponent matches the larger exponent.

**Step 2**:Add the significands.

Step 3:Normalize the result if required either shift right and increment exponent or shift left and decrement the exponent and check for overflow / under flow.

Step 4:Round the significant to appropriate number of bits normalize if required and check for overflow / underflow.

FLOWCHART:


Start
1. Compare the exponents of the two numbers; shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
Done

**Explain how floating point addition is carried out in a computer system. Give an example for a binary floating point addition.**

### Floating-Point Addition

Let's add numbers in scientific notation by hand to illustrate the problems in floating-point addition: $9.999_{\text{ten}} \times 10^{1} + 1.610_{\text{ten}} \times 10^{-1}$. Assume that we can store only four decimal digits of the significand and two decimal digits of the exponent.

Step 1. To be able to add these numbers properly, we must align the decimal point of the number that has the smaller exponent. Hence, we need a form of the smaller number, $1.610_{\text{ten}} \times 10^{-1}$, that matches the larger exponent. We obtain this by observing that there are multiple representations of an unnormalized floating-point number in scientific notation:

$$1.610_{\text{ten}} \times 10^{-1} = 0.1610_{\text{ten}} \times 10^{0} = 0.01610_{\text{ten}} \times 10^{1}$$

The number on the right is the version we desire, since its exponent matches the exponent of the larger number, $9.999_{\text{ten}} \times 10^{1}$. Thus, the first step shifts the significand of the smaller number to the right until its corrected exponent matches that of the larger number. But we can represent only four decimal digits so, after shifting, the number is really

$$0.016 \times 10^{1}$$

Step 2. Next comes the addition of the significands:

$$\begin{array}{r} 9.999_{\text{ten}} \\ +\ \underline{0.016_{\text{ten}}} \\ 10.015_{\text{ten}} \end{array}$$

The sum is $10.015_{\text{ten}} \times 10^{1}$.

Step 3. This sum is not in normalized scientific notation, so we need to adjust it:

$$10.015_{\text{ten}} \times 10^{1} = 1.0015_{\text{ten}} \times 10^{2}$$

Thus, after the addition we may have to shift the sum to put it into normalized form, adjusting the exponent appropriately. This example shows shifting to the right, but if one number were positive and the other were negative, it would be possible for the sum to have many leading 0s, requiring left shifts. Whenever the exponent is increased or decreased, we must check for overflow or underflow—that is, we must make sure that the exponent still fits in its field.

Step 4. Since we assumed that the significand can be only four digits long (excluding the sign), we must round the number. In our grammar school algorithm, the rules truncate the number if the digit to the right of the desired point is between 0 and 4 and add 1 to the digit if the number to the right is between 5 and 9. The number

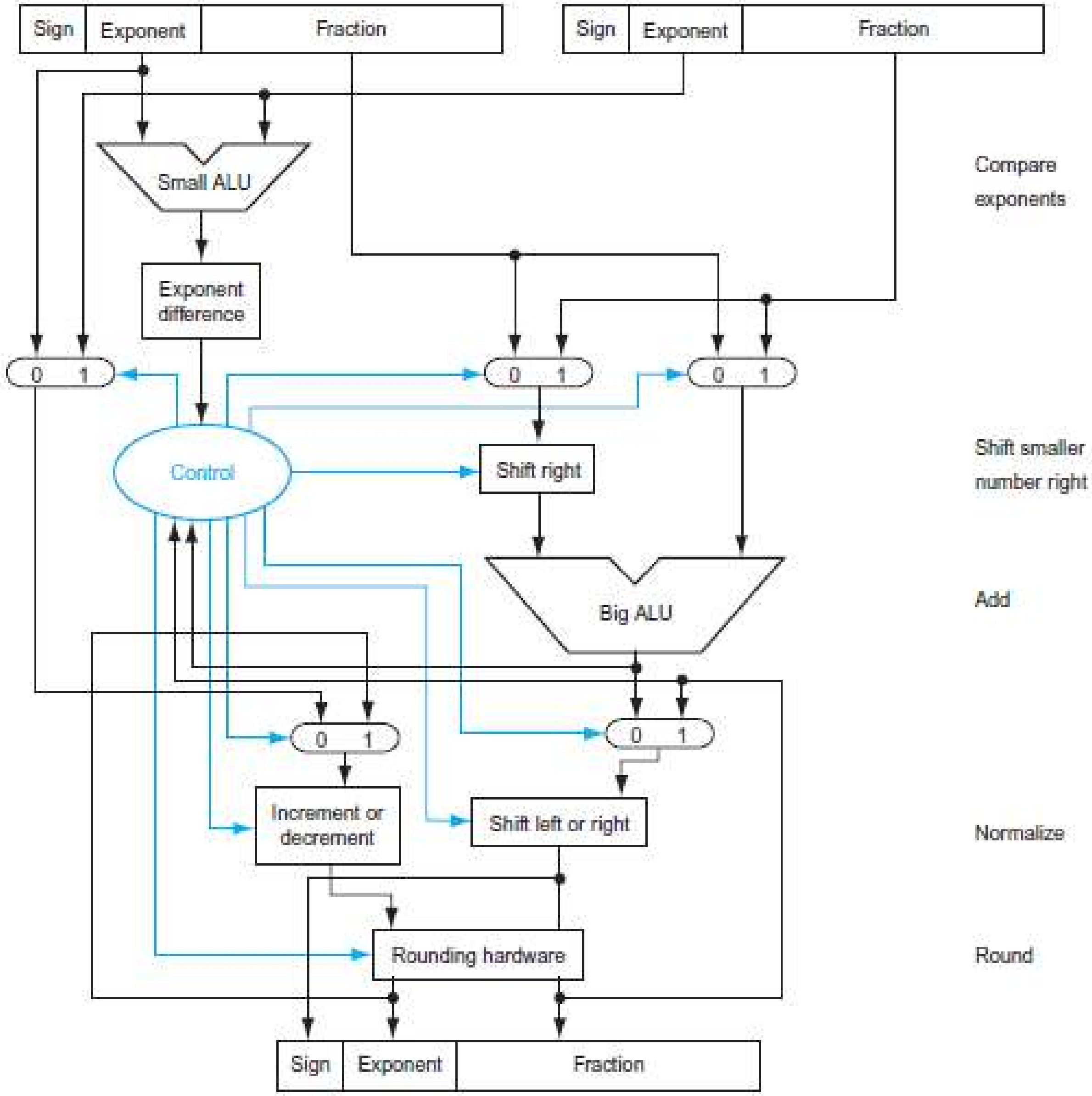
$$1.0015_{\text{ten}} \times 10^{2}$$

is rounded to four digits in the significand to

$$1.002_{\text{ten}} \times 10^{2}$$

since the fourth digit to the right of the decimal point was between 5 and 9. Notice that if we have bad luck on rounding, such as adding 1 to a string of 9s, the sum may no longer be normalized and we would need to perform step 3 again.

## BLOCK DIAGRAM OF ARITHEMETIC UNIT TO FLOATING POINT ADDITION



## FLOATING POINT MULTIPLICATION

### FLOATING POINT MULTIPLICATION ALGORITHM

**Step1:** Add the biased exponent and subtract the bias (127 for Sp and 1023 for Dp).

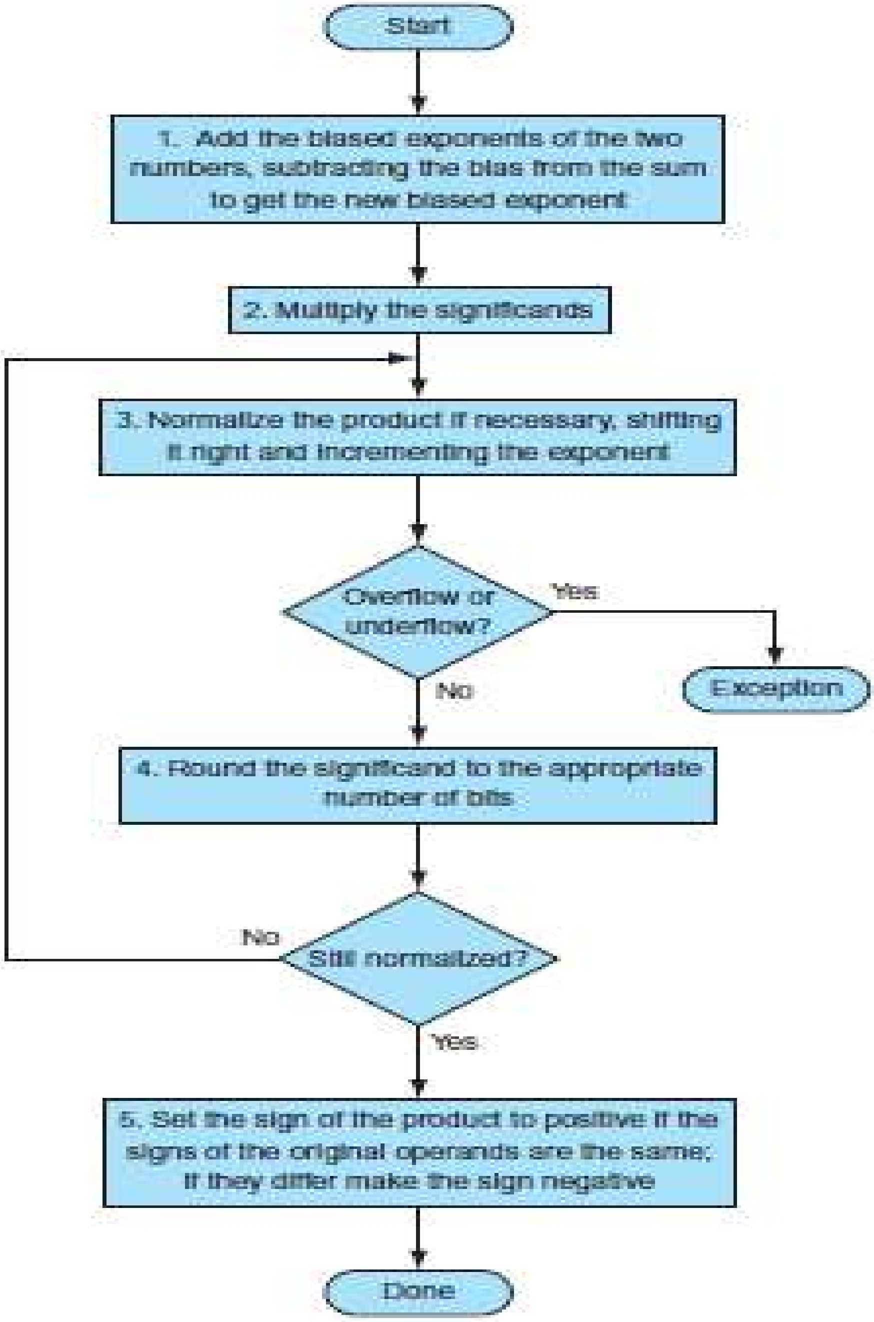
**Step 2:** Multiply the significands.

**Step3**: Normalize the result if required shift right and incrementing the exponent and check for overflow / under flow.

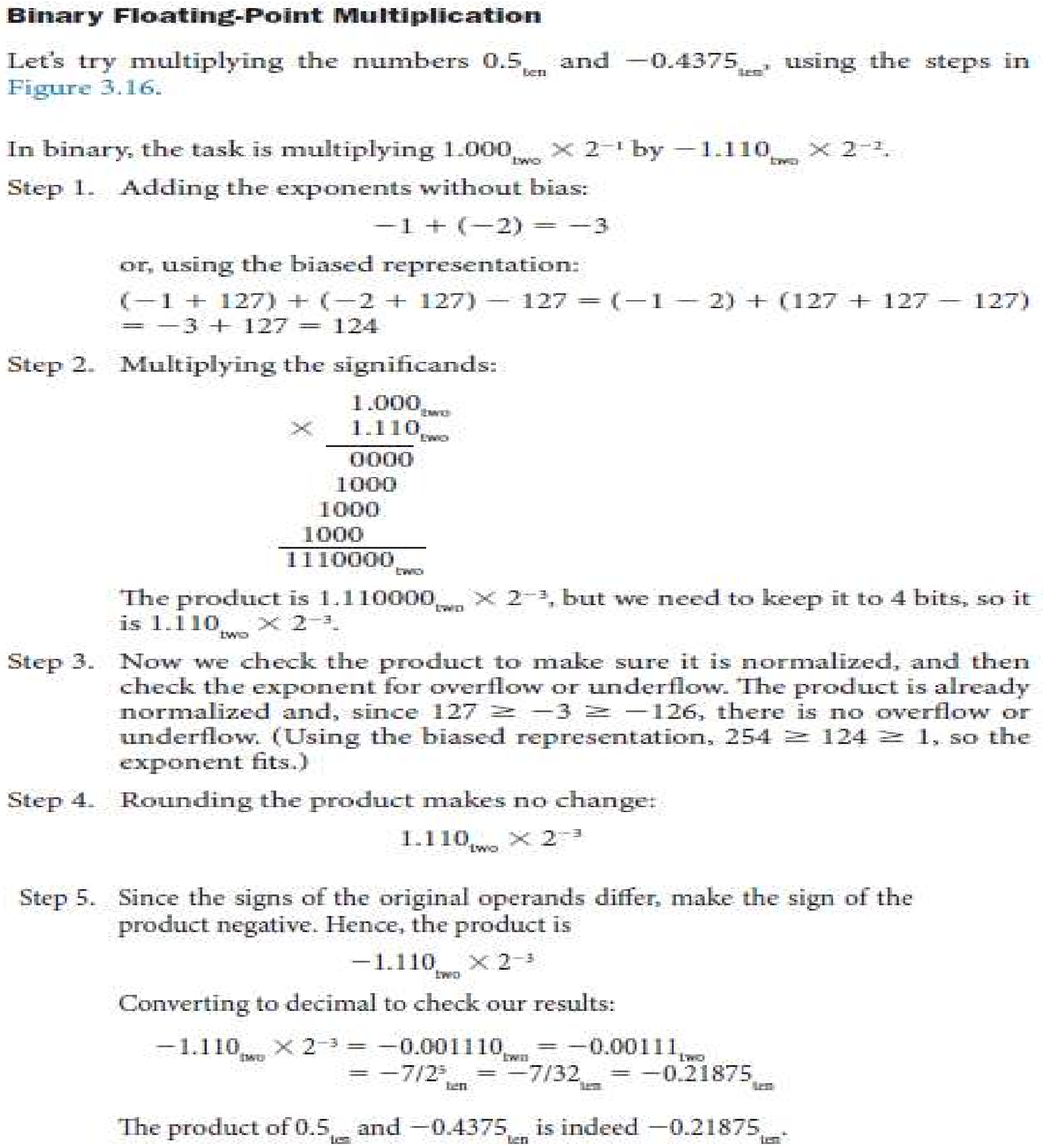
**Step4**: Round the significand to appropriate number of bits and normalize if required.

**Step 5:** Set the sign of the product.

**FLOWCHART**


Start
1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent
2. Multiply the significands
3. Normalize the product if necessary, shifting it right and incrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
5. Set the sign of the product to positive if the signs of the original operands are the same; if they differ make the sign negative
Done

## EXAMPLE



**Binary Floating-Point Multiplication**

Let's try multiplying the numbers $0.5_{ten}$ and $-0.4375_{ten}$, using the steps in Figure 3.16.

In binary, the task is multiplying $1.000_{two} \times 2^{-1}$ by $-1.110_{two} \times 2^{-2}$.

Step 1. Adding the exponents without bias:

$$-1 + (-2) = -3$$

or, using the biased representation:

$$(-1 + 127) + (-2 + 127) - 127 = (-1 - 2) + (127 + 127 - 127)$$
$$= -3 + 127 = 124$$

Step 2. Multiplying the significands:

$$\begin{array}{r} 1.000_{two} \\ \times \quad 1.110_{two} \\ \hline 0000 \\ 1000\phantom{0} \\ 1000\phantom{00} \\ 1000\phantom{000} \\ \hline 1110000_{two} \end{array}$$

The product is $1.110000_{two} \times 2^{-3}$, but we need to keep it to 4 bits, so it is $1.110_{two} \times 2^{-3}$.

Step 3. Now we check the product to make sure it is normalized, and then check the exponent for overflow or underflow. The product is already normalized and, since $127 \geq -3 \geq -126$, there is no overflow or underflow. (Using the biased representation, $254 \geq 124 \geq 1$, so the exponent fits.)

Step 4. Rounding the product makes no change:

$$1.110_{two} \times 2^{-3}$$

Step 5. Since the signs of the original operands differ, make the sign of the product negative. Hence, the product is

$$-1.110_{two} \times 2^{-3}$$

Converting to decimal to check our results:

$$-1.110_{two} \times 2^{-3} = -0.001110_{two} = -0.00111_{two}$$
$$= -7/2^5{}_{ten} = -7/32_{ten} = -0.21875_{ten}$$

The product of $0.5_{ten}$ and $-0.4375_{ten}$ is indeed $-0.21875_{ten}$.

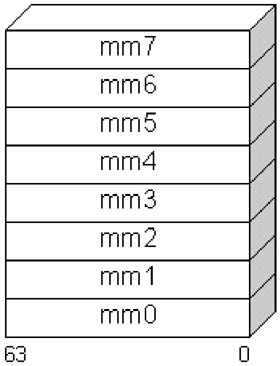
## SUBWORD PARALLELISM

By partitioning the existing 64 bit ALU, the processor can perform simultaneous operation on short vectors of eight 8 bit operands, four 16 bit operands or two 32 bit operands. This type of parallelism with in a wide word is called as subword parallelism..

This is also called vector or SIMD(Single Instruction stream multiple data stream)

**MMX for Intel 80 x 86**

- MMX (multi media Extension) was the first set of SIMD extensions applied to Intel 80x86 processor( Pentium MMX)
- This was introduced in 1997
- MMX provides 8 registers of 64 bit.
- These registers are called MM0 –MM7



MMX REGISTER SET

- These register can be used as
  - One 64 bit Quad word
  - Two 32 bits double word
  - Four 16 bits word
  - Eight 8 bits byte

**MMX has 57 instructions and a few MMX instructions are listed**

**MMX - Data Movement**

| | | |
|---|---|---|
| movd | - | Move double word |
| movq | - | Move Quad word |

**MMX - Boolean Logic**

| | | |
|---|---|---|
| por | - | bitwise OR |
| pand | - | AND |

**MMX - Arithmetic**

| | | |
|---|---|---|
| padd b | - | adds mmx registers as unsigned 8 bit byte |
| padd sb | - | adds as signed 8 bit byte |
| padd w | - | adds as unsigned 16 bit word |
| padd sw | - | adds as signed 16 bit word |

**MMX - Compare**

| | | |
|---|---|---|
| pcmpeqb | - | Compare for 8 bit equality |
| pcmeqw | - | compare for 16 bit equality |

The later SIMD extensions for 80x86 are called SSE, SSE2, SSE3, SSE4 and AVX.

**Neon Multimedia Extension for ARM Processor**

Here 128 bit registers are used as

**Integer Data Type:** 16 no's of 8 bit.
8 no's of 16 bit.
4 no's of 32 bit.
2 no's of 64 bit.

**Float data type**

4 no's of 32 bit.

- Neon has more than 100 instruction to support subword parallelism.

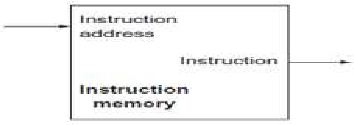
# UNIT III
# PROCESSOR AND CONTROL UNIT

Basic MIPS implementation – Building datapath – Control Implementation scheme – Pipelining –Pipelined datapath and control – Handling Data hazards & Control hazards – Exceptions

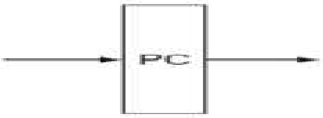
## Basic MIPS implementation

## I. INSTRUCTION FETCH

**Instruction Memory :** Memory to store the instruction of a program and supply instruction for given address.



**Program counter:** Program counter is a 32 bit register that hold the address of current instruction to be fetched.



Datapath Used For Fetching Instructios And Incrementing The Program Counter

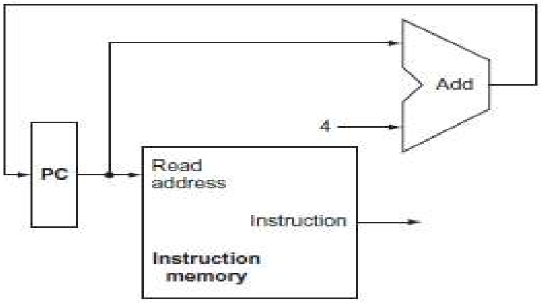
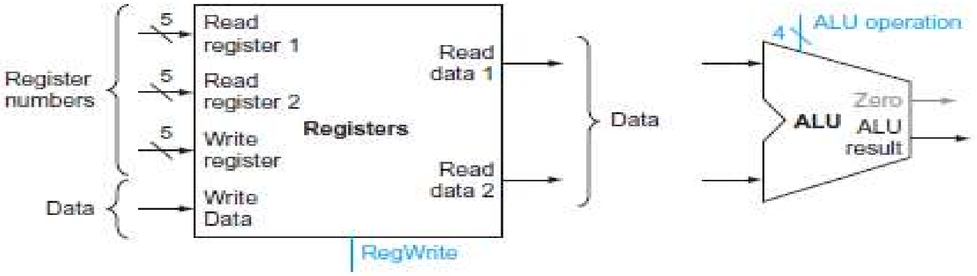


**FIGURE 4.6 A portion of the datapath used for fetching instructions and incrementing the program counter.** The fetched instruction is used by other parts of the datapath.

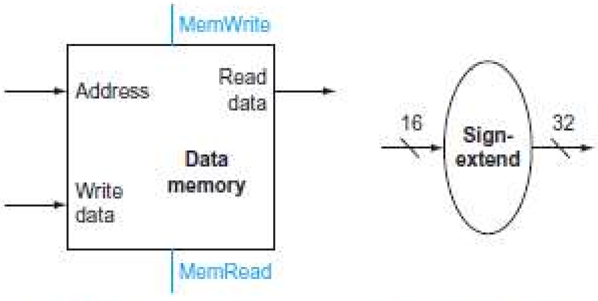
**Register File:**

The processor's 32 general purpose register are stored in a structure called register file. The register file is a collection of registers in which any register can be read or written by specifying the address of the register.



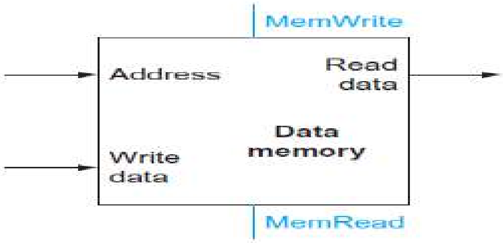
a. Registers b. ALU

**Sign extension of 16 bit offset into 32 bit :**This unit extend 16 bit offset in the instruction to 32 bit



a. Data memory unit b. Sign extension unit

**Data Memory:**The data memory has to be read on load instruction and written on store instruction. Hence the data memory has both read and write control signals, an address input , data output and input for the data to be written.



**<u>BUILDING A DATAPATH:</u>**

The operations of arithmetic-logical (or R-type) instructions and the memory instructions datapath are quite similar. The key differences are the following:

▪ The arithmetic-logical instructions use the ALU, with the inputs coming from the two registers. The memory instructions can also use the ALU to do the address calculation, although the second input is the sign extended 16-bit off set fi eld from the instruction.

▪ The value stored into a destination register comes from the ALU (for an R-type instruction) or the memory (for a load).

**OPERATION OF THE DATAPATH :**

**R type - add $t1,$t2,$t3**

1. The instruction is fetched, and the PC is incremented.

2. Two registers, $t2 and $t3, are read from the register file; also, the main control unit computes the setting of the control lines during this step.

3. The ALU operates on the data read from the register file, using the function code (bits 5:0, which is the funct field, of the instruction) to generate the ALU function.

4. The result from the ALU is written into the register file using bits 15:11 of the instruction to select the destination register ($t1).

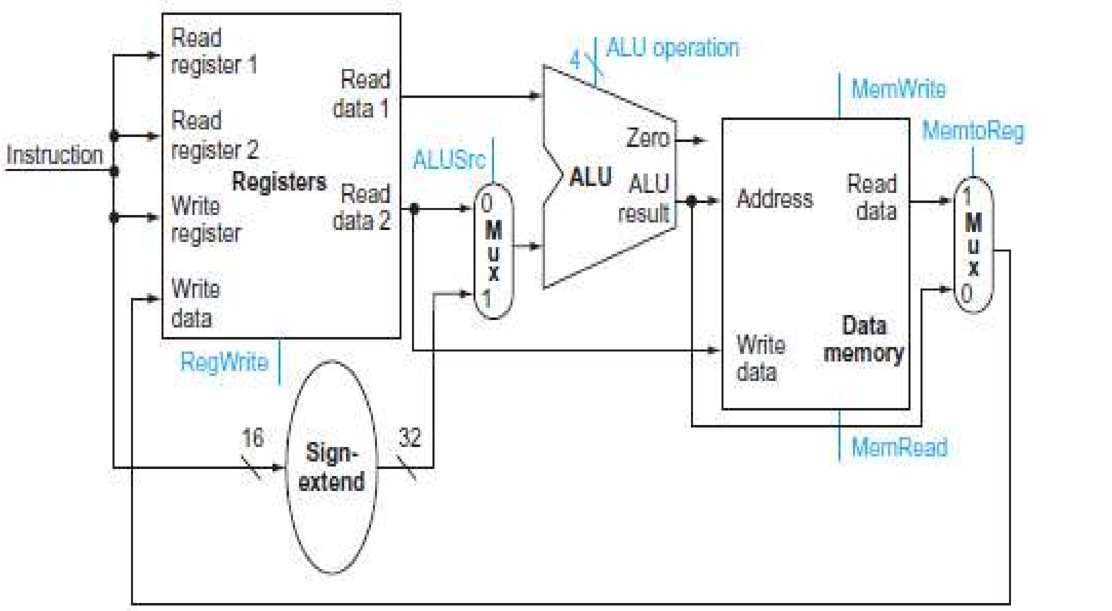


**FIGURE 4.10 The datapath for the memory instructions and the R-type instructions.** This example shows how a single datapath can be assembled from the pieces in Figures 4.7 and 4.8 by adding multiplexors. Two multiplexors are needed, as described in the example.

**lw $t1, offset($t2)**

1. An instruction is fetched from the instruction memory, and the PC is incremented.

2. A register ($t2) value is read from the register file.

3. The ALU computes the sum of the value read from the register fi le and the sign-extended, lower 16 bits of the instruction (offset).

4. The sum from the ALU is used as the address for the data memory.

5. The data from the memory unit is written into the register fi le; the register destination is given by bits 20:16 of the instruction ($t1).

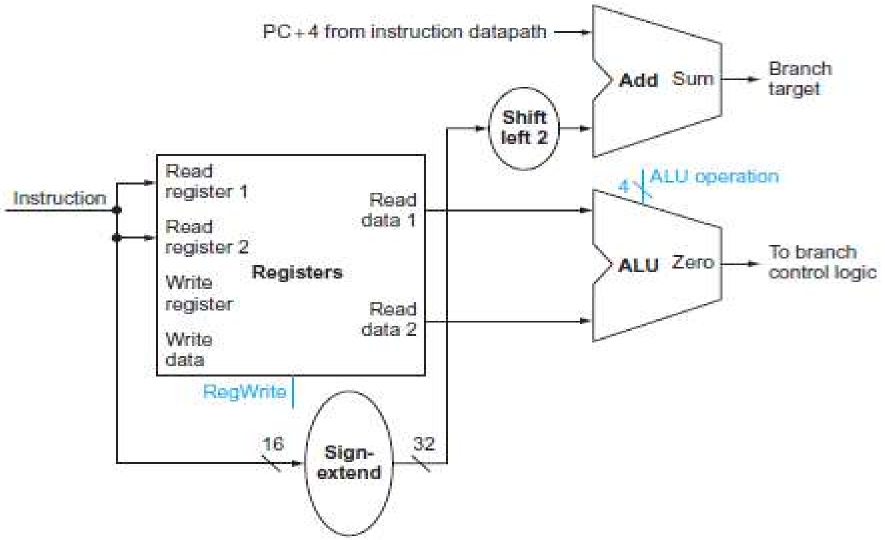


**FIGURE 4.9 The datapath for a branch uses the ALU to evaluate the branch condition and a separate adder to compute the branch target as the sum of the incremented PC and the sign-extended, lower 16 bits of the instruction (the branch displacement), shifted left 2 bits.** The unit labeled *Shift left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

**beq $t1, $t2, offset**

1. An instruction is fetched from the instruction memory, and the PC is incremented.

2. Two registers, $t1 and $t2, are read from the register file.

3. The ALU performs a subtract on the data values read from the register file. The value of PC + is added to the sign-extended, lower 16 bits of the instruction (offset) shifted left by two; the result is the branch target address.

4. The Zero result from the ALU is used to decide which adder result to store into the PC.

## SIMPLE DATAPATH FOR MIPS ARCHITECTURE

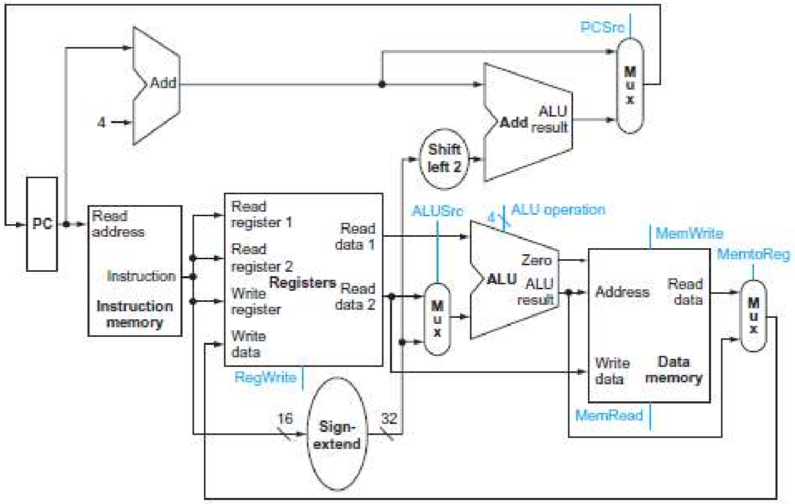


**FIGURE 4.11 The simple datapath for the core MIPS architecture combines the elements required by different instruction classes.** The components come from Figures 4.6, 4.9, and 4.10. This datapath can execute the basic instructions (load-store word, ALU operations, and branches) in a single clock cycle.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

# CONTROL IMPLEMENTATION SCHEME

This control unit consists of two components.

Main control unit

ALU control Unit

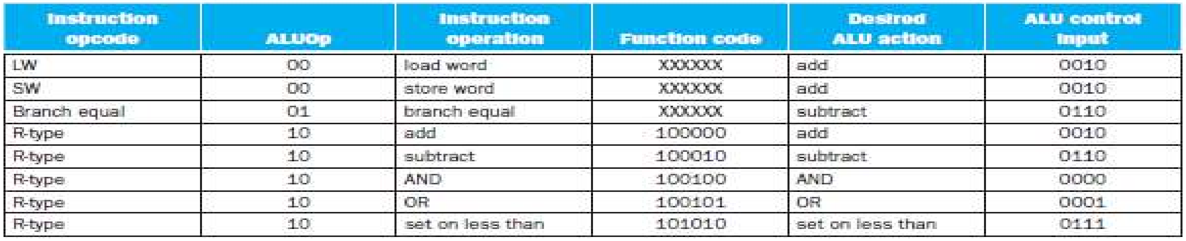
**Main control unit :**Main control unit takes instruction opcode (6 bit) as input and generate 7 control signals and two ALU op

**ALU CONTROL SIGNAL:**The ALU control unit takes two inputs. Function field of the instruction 6 (bits) and 2 bit control field called ALU op and generate 4 ALU control signals.

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

| | |
|---|---|
| **0000** | **AND** |
| **0001** | **OR** |
| **0010** | **Add** |
| **0110** | **Subtract** |
| **0111** | **Set on less than** |
| **1100** | **NOR** |

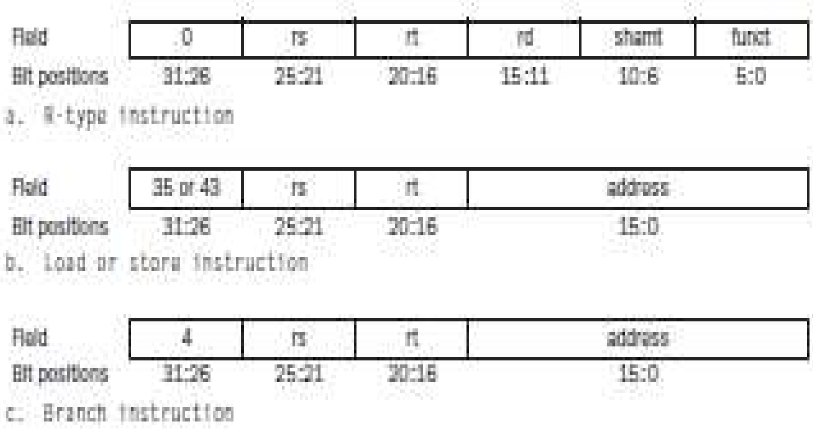
- Depending on the instruction class, the ALU will need to perform one of these first five functions.
- For load word and store word instructions, we use the ALU to compute the memory address by addition
- For the R-type instructions, the ALU needs to perform one of the five actions (AND, OR, subtract, add, or set on less than), depending on the value of the 6-bit funct (or function) field
- in the low-order bits of the instruction .
- For branch equal, the ALU must perform a subtraction.



| Instruction opcode | ALUOp | Instruction operation | Function code | Desired ALU action | ALU control input |
|---|---|---|---|---|---|
| LW | 00 | load word | XXXXXX | add | 0010 |
| SW | 00 | store word | XXXXXX | add | 0010 |
| Branch equal | 01 | branch equal | XXXXXX | subtract | 0110 |
| R-type | 10 | add | 100000 | add | 0010 |
| R-type | 10 | subtract | 100010 | subtract | 0110 |
| R-type | 10 | AND | 100100 | AND | 0000 |
| R-type | 10 | OR | 100101 | OR | 0001 |
| R-type | 10 | set on less than | 101010 | set on less than | 0111 |

**FIGURE 4.47 A copy of Figure 4.12.** This figure shows how the ALU control bits are set depending on the ALUOp control bits and the different function codes for the R-type instruction.

Instruction Classes / Instruction Format:



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

a. R-type instruction

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

b. Load or store instruction

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

c. Branch instruction

There are several major observations about this instruction format that we will rely on:

■ The op field, also called the Opcode, is always contained in bits 31:26. We will refer to this field as Op[5:0].

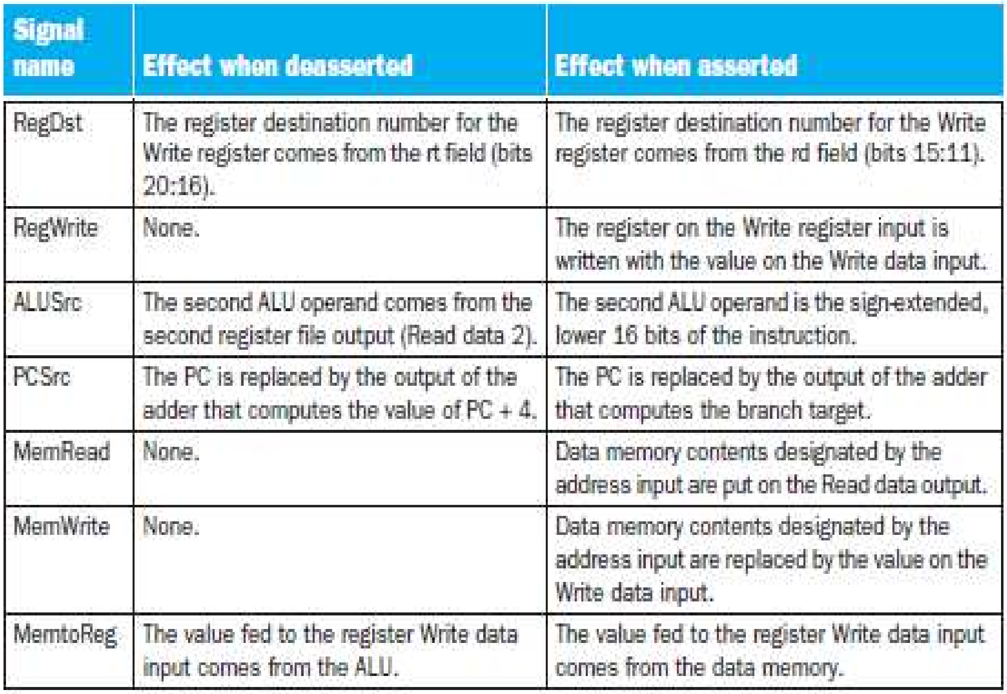
■ The two registers to be read are always specified by the rs and rt fields, at positions 25:21 and 20:16. This is true for the R-type instructions, branch equal, and for store.

■ The base register for load and store instructions is always in bit positions 25:21 (rs).

The 16-bit offset for branch equal, load, and store is always in positions 15:0.

■ The destination register is in one of two places. For a load it is in bit positions 20:16 (rt), while for an R-type instruction it is in bit positions 15:11 (rd). Thus we will need to add a multiplexor to select which field of the instruction is used to indicate the register number to be written.

**Fuction of seven control signals**

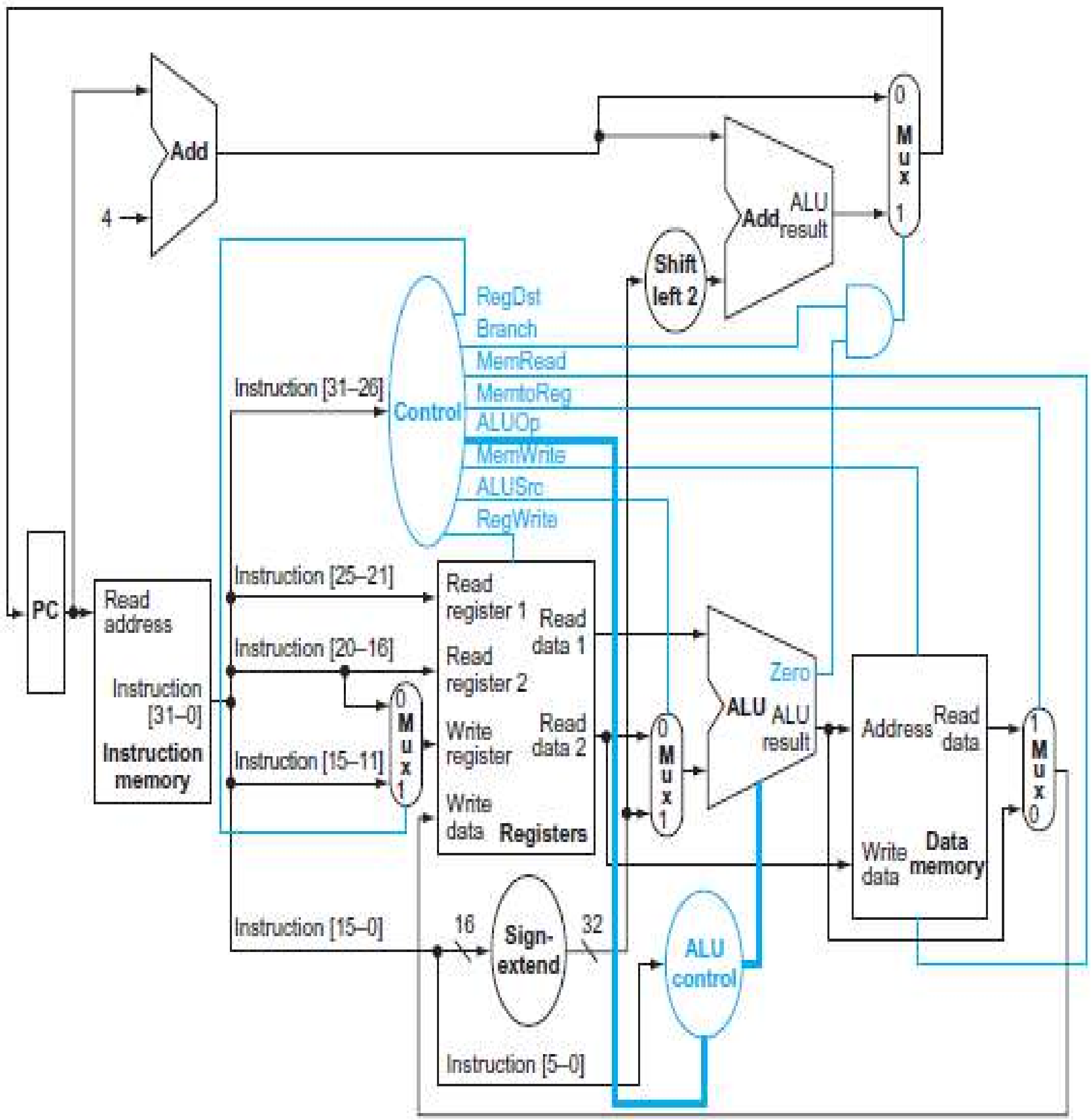


| Signal name | Effect when deasserted | Effect when asserted |
|---|---|---|
| RegDst | The register destination number for the Write register comes from the rt field (bits 20:16). | The register destination number for the Write register comes from the rd field (bits 15:11). |
| RegWrite | None. | The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

Explanation of the above mentioned tabular column

| | | |
|---|---|---|
| Regdst | 0 | Register destination rt field (bits 20:16) |
| | 1 | Register destination rd field(bits 15:11) |
| Regwrite | 0 | none |
| | 1 | Data is written into register |
| ALUSrc | 0 | Second register file out load data 2 |
| | 1 | Sign extended 16 bit of instruction |
| PCSrc | 0 | normal Pc is replaced by PC+4 |
| | 1 | the PC is replaced by PC+4 + (4 * offset) |
| MemRead | 0 | None |
| | 1 | Data memory Read |
| memWrite | 0 | None |
| | 1 | Data memory Write |
| MemReg | 0 | Write data for reg comes from ALU |
| | 1 | Comes from memory |

A SIMPLE DATAPATH WITH THE CONTROL UNIT



**R type - add $t1,$t2,$t3**

1. The instruction is fetched, and the PC is incremented.

2. Two registers, $t2 and $t3, are read from the register file; also, the main control unit computes the setting of the control lines during this step.

3. The ALU operates on the data read from the register file, using the function code (bits 5:0, which is the funct field, of the instruction) to generate the ALU function.

4. The result from the ALU is written into the register file using bits 15:11 of the instruction to select the destination register ($t1).

**lw $t1, offset($t2)**

1. An instruction is fetched from the instruction memory, and the PC is incremented.

2. A register ($t2) value is read from the register file.

3. The ALU computes the sum of the value read from the register file and the sign-extended, lower 16 bits of the instruction (offset).

4. The sum from the ALU is used as the address for the data memory.

5. The data from the memory unit is written into the register file; the register destination is given by bits 20:16 of the instruction ($t1).

**beq $t1, $t2, offset**

1. An instruction is fetched from the instruction memory, and the PC is incremented.

2. Two registers, $t1 and $t2, are read from the register file.

3. The ALU performs a subtract on the data values read from the register file. The value of PC + is added to the sign-extended, lower 16 bits of the instruction (offset) shifted left by two; the result is the branch target address.

4. The Zero result from the ALU is used to decide which adder result to store into the PC.

# PIPELINE

## Pipeline

Pipelining is used to overlap the execution of instructions and improve performance. This potential overlap among instructions is called instruction level parallelism (ILP) since the instruction can be evaluated in parallel.

The value of the CPI (cycles per instruction) for a pipelined processor is the sum of the base CPI and all contributions from stalls:

**Pipeline CPI = Ideal pipeline CPI + Structural stalls + Data hazard stalls + Control stalls**

The *ideal pipeline CPI* is a measure of the maximum performance attainable by the implementation.

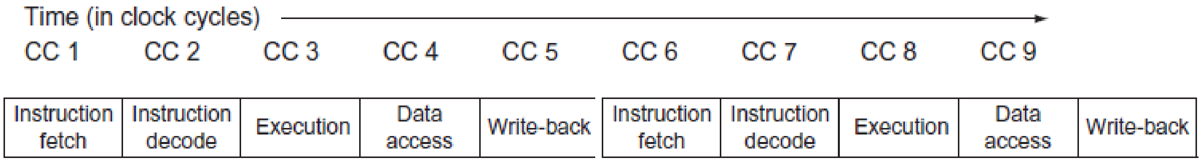
**Pipeline Instruction-Execution**

- Fetch instruction from memory (IF)
- Decoding the instruction & read registers (ID)

- Execute the operation or calculate address( E)
- Access the operand in memory (Mem)
- Write result into a register (WB)

**Non pipelined execution of instructions.**

The sequential execution of instructions is shown in diagram below.

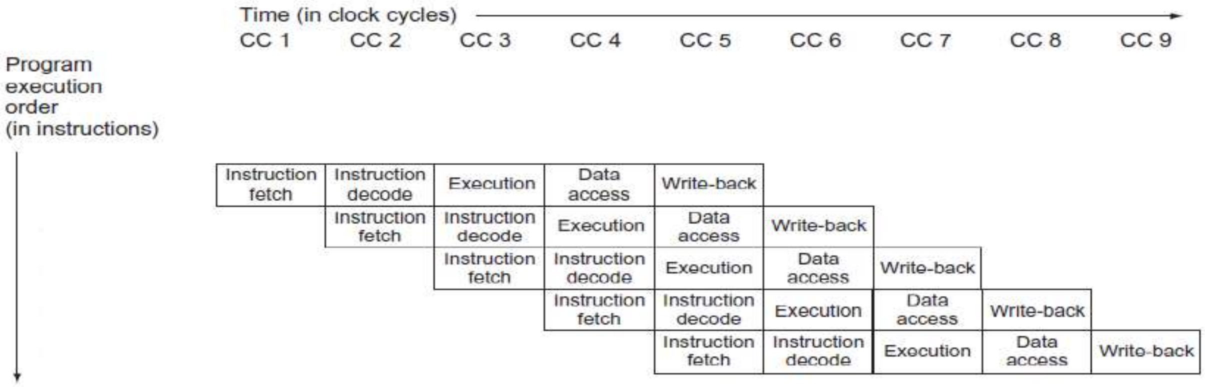


Assuming each step takes one clock cycle, each instruction execution takes

**5 clock cycles**

**Pipelined execution of instructions.**

The sequence of events in the five stage pipeline system is shown in diagram below.



Here 1st instruction gets completed in clock cycle 5

Then for each clock cycle one instruction is getting completed.

# PIPELINE PERFORMANCE

- In sequential operation, each instruction needs five clock cycles
- In pipelined processor, the processing of one instruction is completed in each clock cycle
- Under ideal condition with a large number of instructions, the speedup of pipelining is approximately equal to number of pipeline stages. i.e. the speed up of five stage pipeline is nearly five.

## PIPELINE DATAPATH

The goal of pipelining is to allow multiple instructions execute at the same time. It may need to perform several operations in a clock cycle such as increment the PC and add registers at the same time, fetch one instruction while another one reads or writes data, etc,.

In figure 3.1, these five components correspond roughly to the way the datapath is drawn; instructions and data move generally from left to right through the five stages as they complete execution.

There are however, two exceptions to this left-to-right flow of instructions:

- The write-back stage, which places the result back into the register file in the middle of the datapath.
- The selection of the next value of the PC, choosing between the incremented PC and the branch address from the MEM stage.

Data flowing from left does not affect the current instruction; only later instructions in the pipeline are influenced by these reverse data movements. Note that the first right-to-left flow of data can lead to data hazards and the second leads to control hazards.

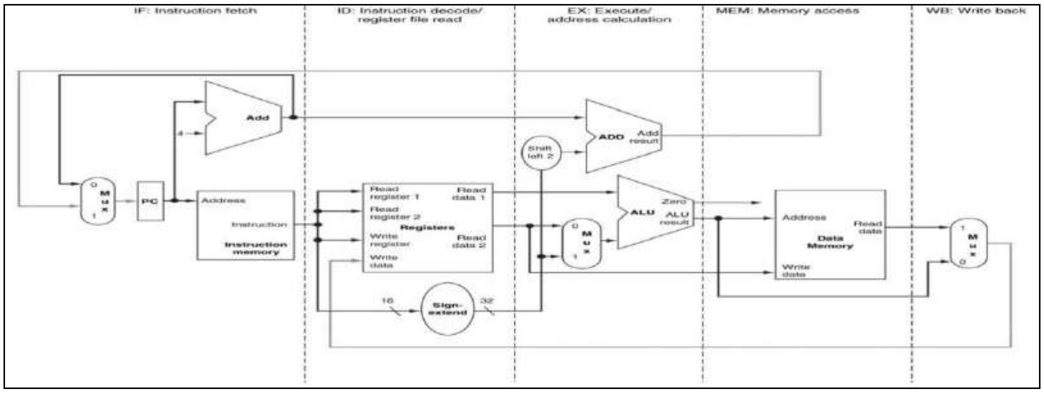


Figure 3.1 Pipeline Process for Single-Cycle Datapath

One way to show what happens in pipelined execution is to pretend that each instruction has its own datapath, and then to place these data paths on a timeline to show their relationship. Figure 3.2 shows the execution of the instructions by displaying their private data paths on a common timeline.

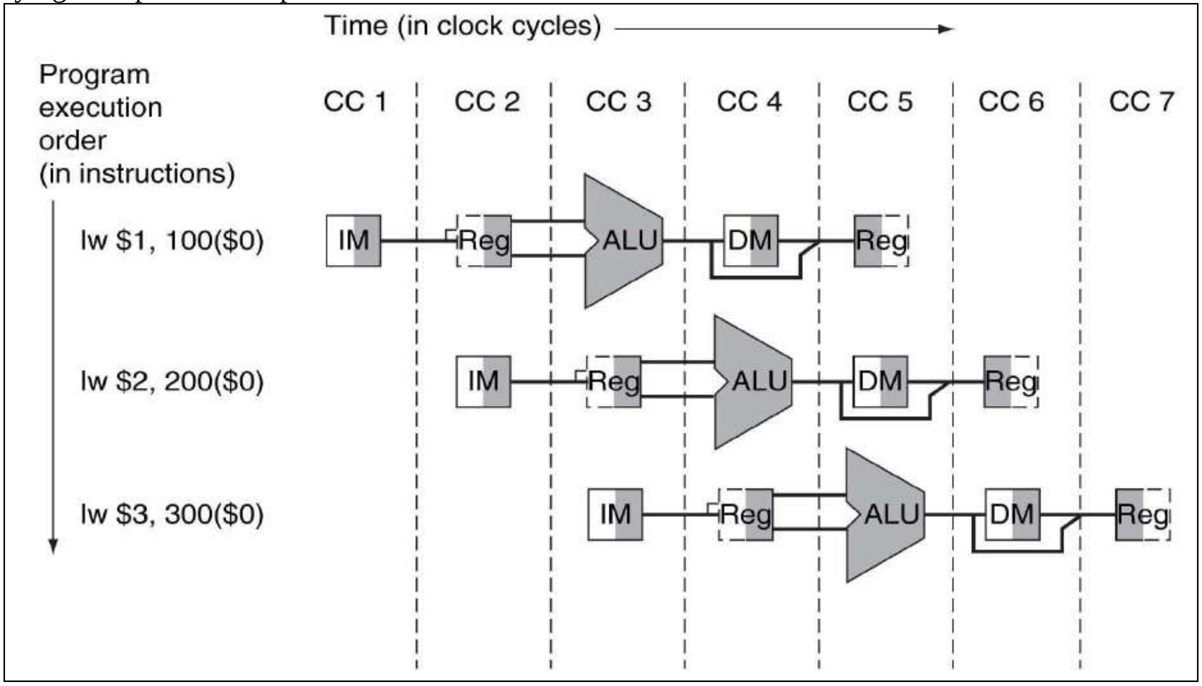


Figure 3.2 Instructions being executed using the single-cycle datapath

This figure seems to suggest that three instructions need three datapaths. Instead, add registers to hold data so that portions of a single datapath can be shared during instruction execution. The instruction memory is used during only one of the five stages of an instruction, allowing it to be shared by following instructions during the other four stages. To retain the value of an individual instruction for its other four stages, the value read from instruction memory must be saved in a register. Similar arguments apply to every pipeline stage must be placed in registers.

Figure 3.3 shows the pipelined datapath with the pipeline registers highlighted. All instructions advance during each clock cycle from one pipeline register to the next. The registers are named for the two stages separated by that register. For example, the pipeline register between the IF and ID stages is called IF/ID.

**PIPELINED DATAPATH**

The division of an instruction into five stages means a five-stage pipeline, which in turn means that up to five instructions will be in execution during any single clock cycle. Thus, we must separate the datapath into five pieces, with each piece named corresponding to a stage of instruction execution:

1. IF: Instruction fetch
2. ID: Instruction decode and register file read
3. EX: Execution or address calculation
4. MEM: Data memory access
5. WB: Write back

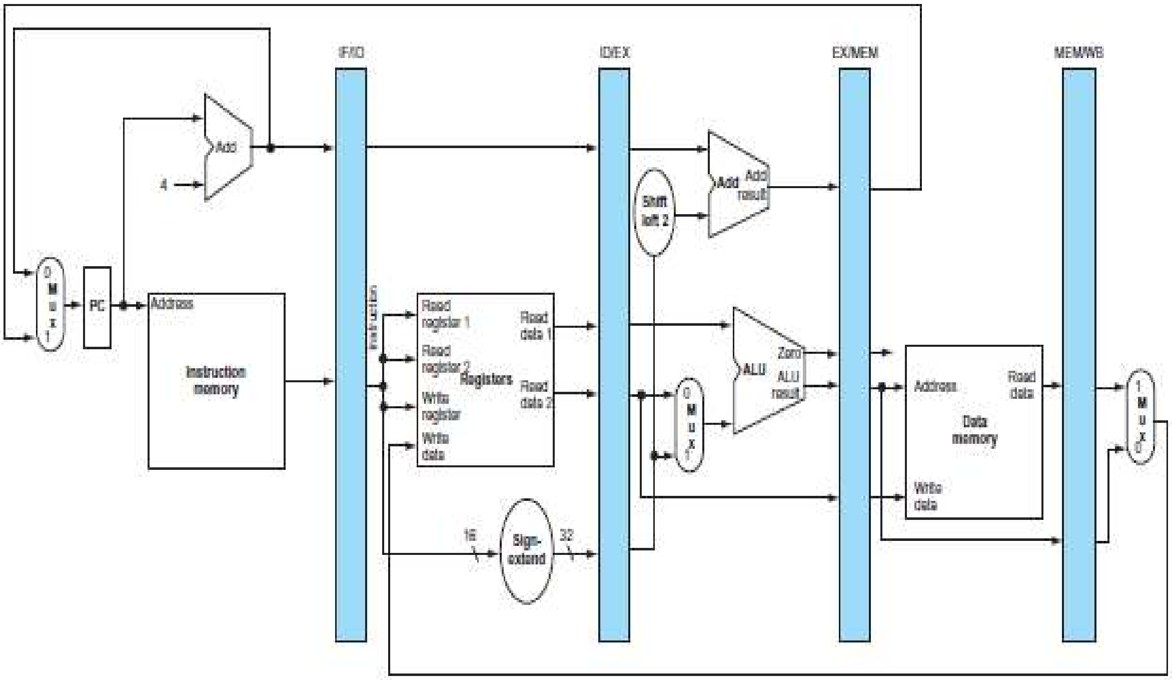


**FIGURE 4.35 The pipelined version of the datapath in Figure 4.33.** The pipeline registers, in color, separate each pipeline stage. They are labeled by the stages that they separate; for example, the first is labeled *IF/ID* because it separates the instruction fetch and instruction decode stages.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

**The five stages are the following:**

1. **Instruction fetch:** In this stage instruction is being read from memory using the address in the PC and then being placed in the IF/ID pipeline register. The PC address is incremented by 4 and then written back into the PC to be ready for the next clock cycle. This incremented address is also saved in the IF/ID pipeline register in case it is needed later for an instruction, such as beq.

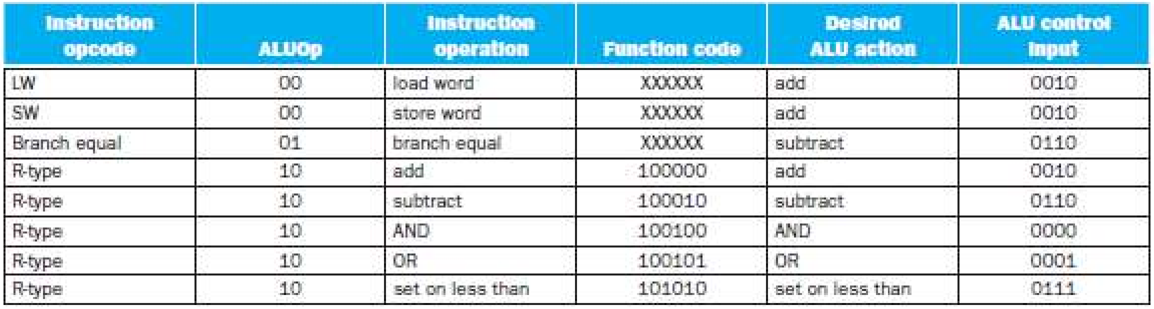
2. **Instruction decode and register file read:** The bottom portion of shows the instruction portion of the IF/ID pipeline register supplying the 16-bit immediate fi eld, which is sign-extended to 32 bits, and the register numbers to read the two registers. All three values are stored in the ID/EX pipeline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

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

4. **Memory access:** The top portion of the diagram shows the load instruction reading the data memory using the address from the EX/MEM pipeline register and loading the data into the MEM/WB pipeline register.

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

**<u>PIPELINE CONTROL LINES</u>**



| Instruction opcode | ALUOp | Instruction operation | Function code | Desired ALU action | ALU control input |
|---|---|---|---|---|---|
| LW | 00 | load word | XXXXXX | add | 0010 |
| SW | 00 | store word | XXXXXX | add | 0010 |
| Branch equal | 01 | branch equal | XXXXXX | subtract | 0110 |
| R-type | 10 | add | 100000 | add | 0010 |
| R-type | 10 | subtract | 100010 | subtract | 0110 |
| R-type | 10 | AND | 100100 | AND | 0000 |
| R-type | 10 | OR | 100101 | OR | 0001 |
| R-type | 10 | set on less than | 101010 | set on less than | 0111 |

**FIGURE 4.47** **A copy of Figure 4.12.** This figure shows how the ALU control bits are set depending on the ALUOp control bits and the different function codes for the R-type instruction.

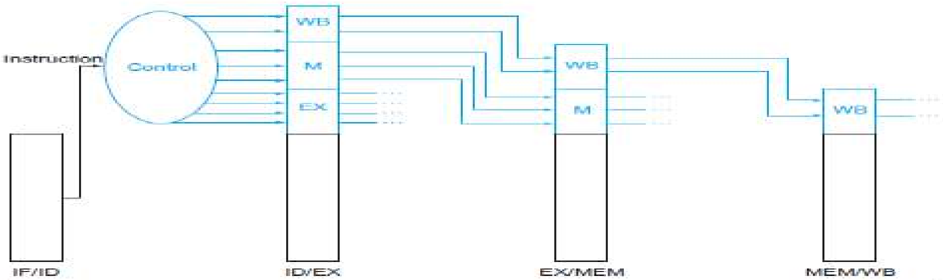


**FIGURE 4.50** **The control lines for the final three stages.** Note that four of the nine control lines are used in the EX phase, with the remaining five control lines passed on to the EX/MEM pipeline register extended to hold the control lines; three are used during the MEM stage, and the last two are passed to MEM/WB for use in the WB stage.

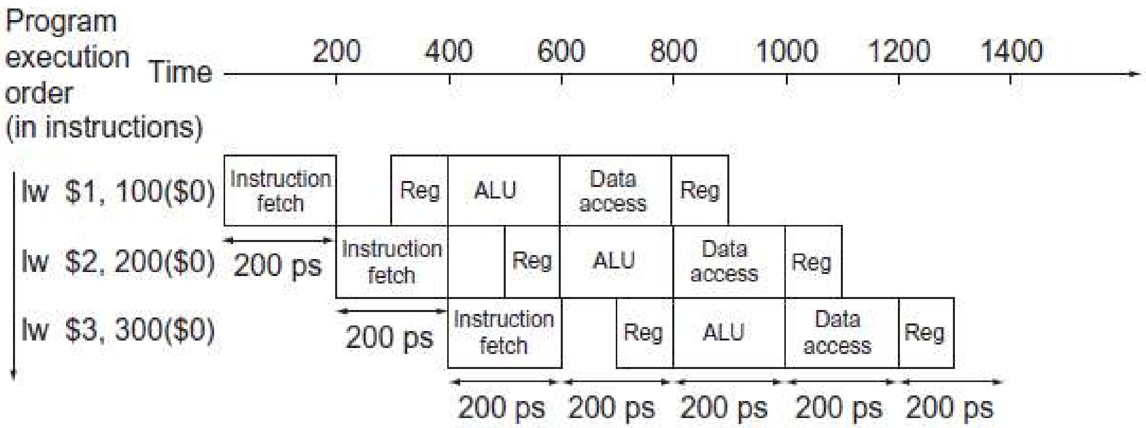
**PIPELINE HAZARDS**

Due to non availability of the data or instruction, the next instruction cannot be executed in the following clock cycle, these events are called hazards. There are three different types of hazards.

1. Structural hazard
2. Data hazard
3. Control Hazard

## Structural hazard

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



This will happen when the system has **single unified cache memory** for both instruction and data. The pipelined executions of three instructions are shown in the following figure.

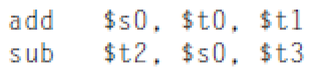
When the fourth instruction is fetched, in the same clock cycle the first instruction is accessing data from the memory while the fourth instruction is fetching an instruction from the same memory. Without two memories, the pipeline had a structural hazard.

**To overcome structural Hazards**

Hence the most of the current processors have two caches one for **instruction and one for data (split cache) which will reduce the structural hazard.**

**DATA HAZARD**

Data Hazards occur when pipeline must be stalled because one Instruction must wait for another instruction to complete.



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

Suppose we have an add instruction followed immediately by a subtract instruction that uses the sum ($S0).The second instruction has to wait until the add instruction has written its result. add intruction writes its result only during fifth clock cycle, there clock cycle is wasted in pipeline, this stall condition is known as Data Hazard. Data hazard is caused due the data dependency between instructions.

**Solution for Data hazards(Various techniques to reduce Data Hazards)**

**1.Forwarding or Bypassing**

The primary solution is based on the observation that we don't need to wait for the instruction to complete before trying to resolve the data hazard. For the code sequence above, as soon as the ALU creates the sum for the add, we can supply it as an input for the subtract. Adding extra hardware to retrieve the missing item early from the internal resources is called forwarding or bypassing.

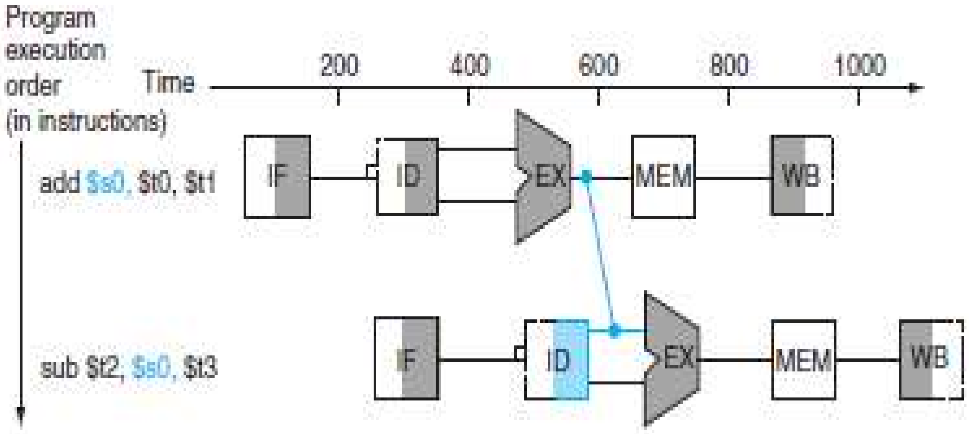


**FIGURE 6.5 Graphical representation of forwarding.** The connection shows the forwarding path from the output of the EX stage of add to the input of the EX stage for sub, replacing the value from register $s0 read in the second stage of sub.

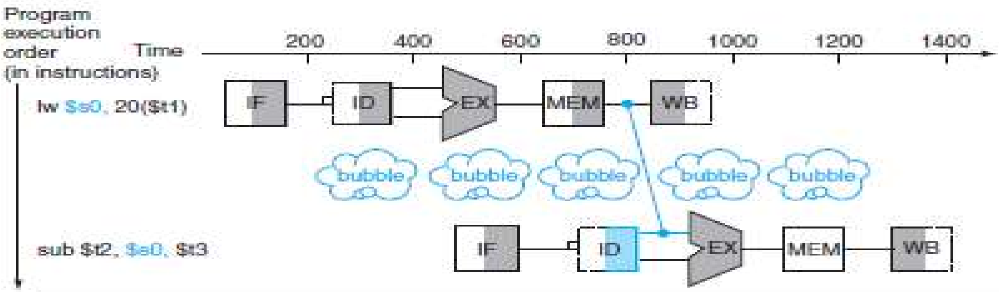
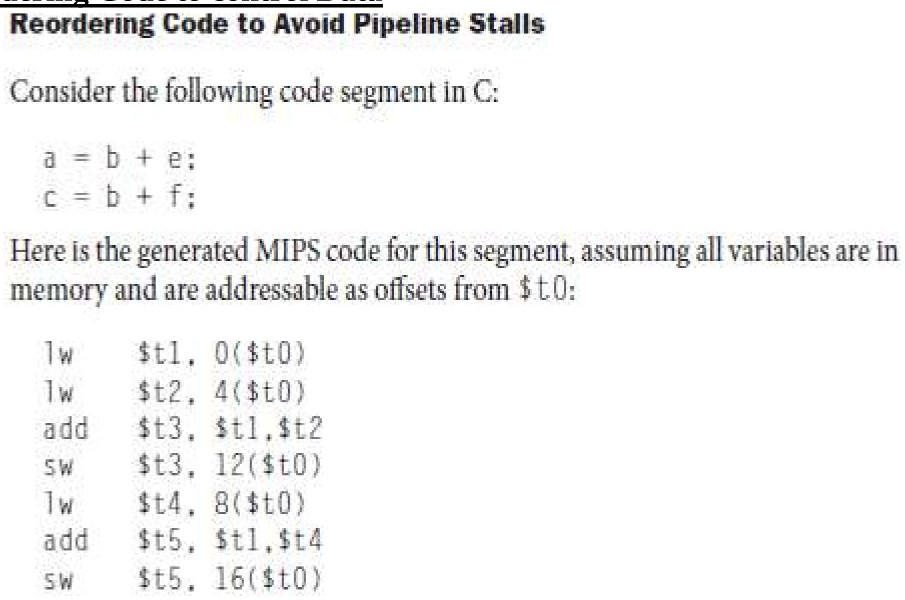


**FIGURE 6.6 We need a stall even with forwarding when an R-format instruction following a load tries to use the data.** Without the stall, the path from memory access stage output to execution stage input would be going backwards in time, which is impossible. This figure is actually a simplification, since we cannot know until after the subtract instruction is fetched and decoded whether or not a stall will be necessary. Section 6.5 shows the details of what really happens in the case of a hazard.

## 2. Reordering Code to control Data



### Reordering Code to Avoid Pipeline Stalls

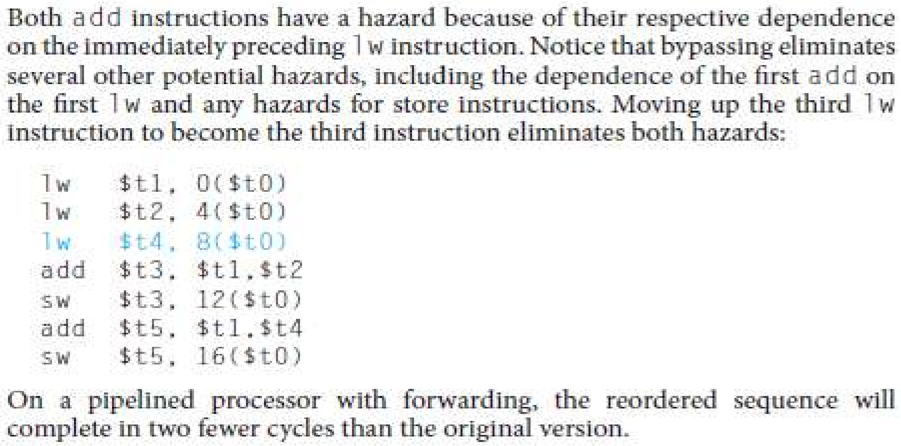
Consider the following code segment in C:

```
a = b + e;
c = b + f;
```

Here is the generated MIPS code for this segment, assuming all variables are in memory and are addressable as offsets from $t0:

```
lw    $t1, 0($t0)
lw    $t2, 4($t0)
add   $t3, $t1,$t2
sw    $t3, 12($t0)
lw    $t4, 8($t0)
add   $t5, $t1,$t4
sw    $t5, 16($t0)
```

## Hazard



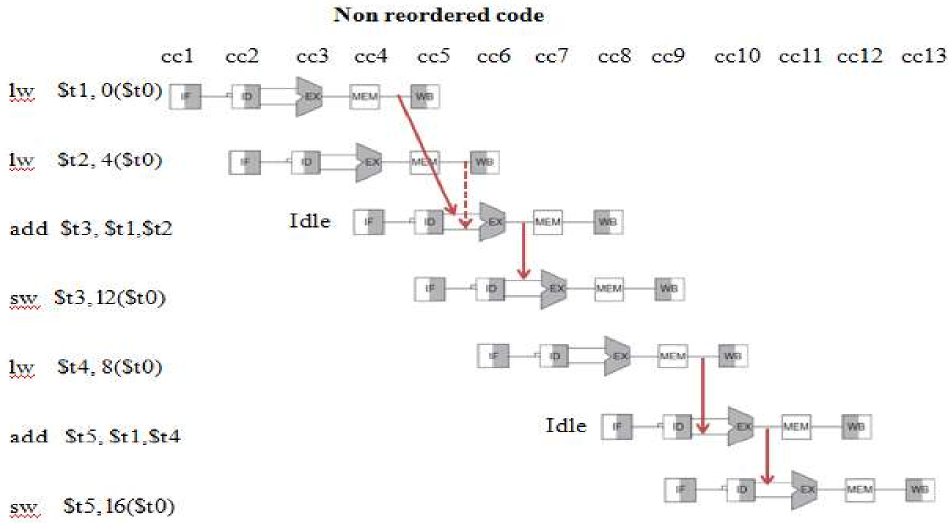
Both add instructions have a hazard because of their respective dependence on the immediately preceding lw instruction. Notice that bypassing eliminates several other potential hazards, including the dependence of the first add on the first lw and any hazards for store instructions. Moving up the third lw instruction to become the third instruction eliminates both hazards:

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
lw   $t4, 8($t0)
add  $t3, $t1,$t2
sw   $t3, 12($t0)
add  $t5, $t1,$t4
sw   $t5, 16($t0)
```

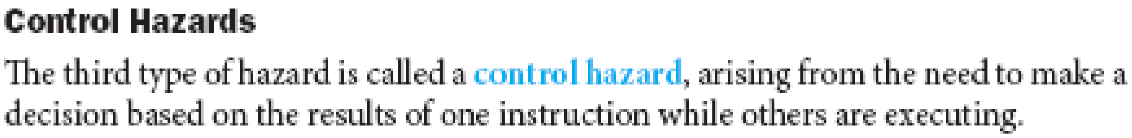
On a pipelined processor with forwarding, the reordered sequence will complete in two fewer cycles than the original version.

**Nop: An instruction that does no operation to change state.**

**Diagrammatic representation:**

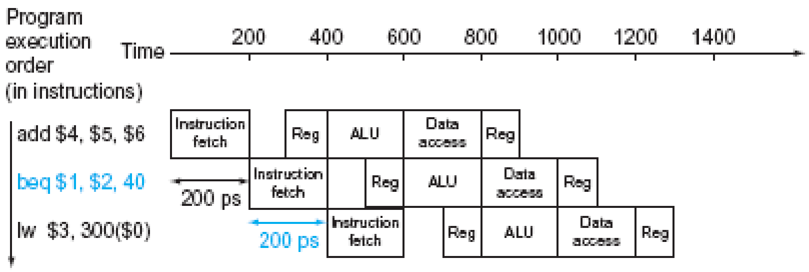


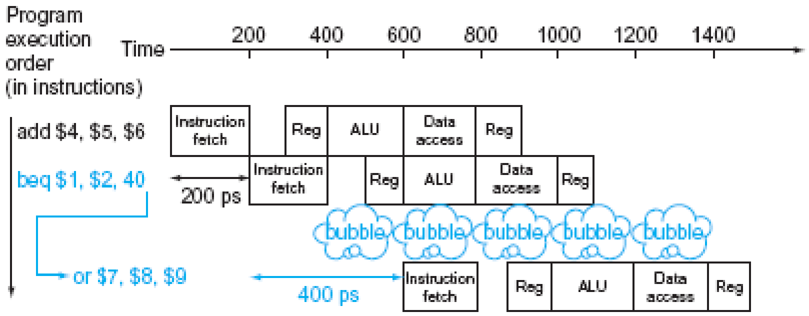
CONTROL  HAZARDS



**Control Hazards**

The third type of hazard is called a control hazard, arising from the need to make a decision based on the results of one instruction while others are executing.





## Various techniques to reduce control Hazards

### 1. Branch prediction:

#### Static Branch Prediction:

To reorder code around branches so that it runs faster, we need to predict the branch statically when we compile the program. There are several different methods to statically predict branch behavior. The simplest scheme is to predict a branch as taken.

This scheme has an average misprediction rate that is equal to the untaken branch frequency, which for the SPEC programs is 34%. Unfortunately, the misprediction rate for the SPEC programs ranges from not very accurate (59%) to highly accurate (9%).A more accurate technique is to predict branches on the basis of profile information collected from earlier runs. The effectiveness of any branch prediction scheme depends both on the accuracy of the scheme and the frequency of conditional branches, which vary in SPEC from 3% to 24%. The fact that the misprediction rate for the integer programs is higher and that such programs typically have a higher branch frequency is a major limitation for static branch prediction.

**2.Dynamic Branch Prediction:** Is the ability of the hardware to make an educated guess about which way a branch will go - will the branch be taken or not. The hardware can look for clues based on the instructions, or it can use past history .

The simplest dynamic branch-prediction scheme is a ***branch-prediction buffer*** **or** ***branch history table.*** A branch-prediction buffer is a small memory indexed by the lower portion of the address of the branch instruction. The memory contains a bit that says whether the branch was recently taken or not. The prediction is a hint that is assumed to be correct, and fetching begins in the predicted direction. If the hint turns out to be wrong, the prediction bit is inverted and stored back. This simple 1-bit prediction scheme has a performance shortcoming: Even if a branch is almost always taken, we will likely predict incorrectly twice, rather than once, when it is not taken, since the misprediction causes the prediction bit to be flipped.

`To remedy this weakness, 2-bit prediction schemes are often used. In a 2-bit scheme, a prediction must miss twice before it is changed. Figure 2.4 shows the finite-state processor for a 2-bit prediction scheme. A branch-prediction buffer can be implemented as a small, special "cache" accessed with the instruction address during the IF pipe stage, or as a pair of bits attached to each block in the instruction cache and fetched with the instruction. If the instruction is decoded as a branch and if the branch is predicted as taken, fetching begins from the target as soon as the PC is known. Otherwise, sequential fetching and executing continue. As Figure 2.4 shows, if the prediction turns out to be wrong, the prediction bits are changed.

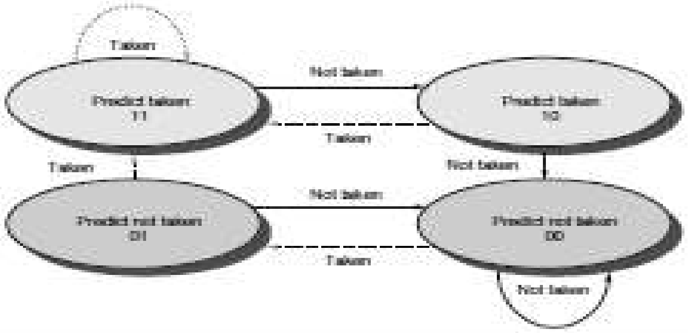


**Figure 2.4 The states in a 2-bit prediction scheme.** By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted less often than with a 1-bit predictor. The 2 bits are used to encode the four states in the system. The 2-bit scheme is actually a specialization of a more general scheme that has an $n$-bit saturating counter for each entry in the prediction buffer. With an $n$-bit counter, the counter can take on values between 0 and $2^n - 1$: When the counter is greater than or equal to one-half of its maximum value ($2^n - 1$), the branch is predicted as taken; otherwise, it is predicted untaken. Studies of $n$-bit predictors have shown that the 2-bit predictors do almost as well, and thus most systems rely on 2-bit branch predictors rather than the more general $n$-bit predictors.

### *3.Correlating Branch Predictors*

The 2-bit predictor schemes use only the recent behavior of a single branch to predict the future behavior of that branch. It may be possible to improve the prediction accuracy if we also look at the recent behavior of *other* branches rather than just the branch we are trying to predict. Consider a small code fragment

```
if (aa==2)
aa=0;
if (bb==2)
bb=0;
```

```
                    if (aa!=bb) {}
```

Let’s label these branches b1, b2, and b3. The key observation is that the behavior of branch b3 is correlated with the behavior of branches b1 and b2. Clearly, if branches b1 and b2 are both not taken (i.e., if the conditions both evaluate to true and aa and bb are both assigned 0), then b3 will be taken, since aa and bb are clearly equal. A predictor that uses only the behavior of a single branch to predict.

Branch predictors that use the behavior of other branches to make a prediction are called *correlating predictors* or *two-level predictors*.

### 4. Tournament Predictors: Adaptively Combining Local and Global Predictors

*Tournament predictors* take this insight to the next level, by using multiple predictors, usually one based on global information and one based on local information, and combining them with a selector.The advantage of a tournament predictor is its ability to select the right predictor for a particular branch.

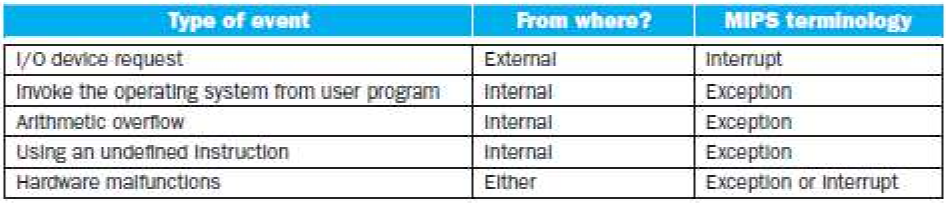
## <u>EXCEPTION HANDING IN MIPS ARCHITECTURE</u>

One of the hardest parts of control is implementing **EXCEPTIONS** AND **INTERRUPTS**

events other than branches or jumps that change the normal flow of instruction execution. They were initially created to handle unexpected events from within the processor, like arithmetic overflow.

**EXCEPTION** to refer to *any* unexpected change in control flow without distinguishing whether the cause is internal or external;

**INTERRUPT** refer to when the event is externally caused. Here are five examples showing whether the situation is internally generated by the processor or externally generated:



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

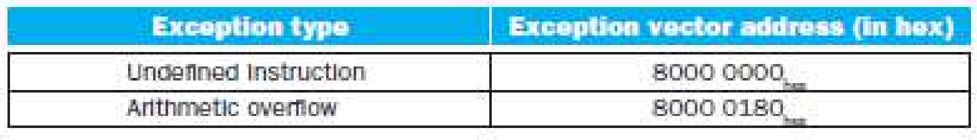
- The **two types of exceptions** that our current implementation can generate are execution of an **undefined instruction and an arithmetic overflow**.
- The basic action that the processor must perform when an exception occurs is to save the address of the offending instruction in the *exception program counter* (EPC) and then transfer control to the operating system at some specified address.
- The operating system can then take the appropriate action, which may involve providing some service to the user program, taking some predefined action in response to an overflow, or stopping the execution of the program and reporting an error.

- After performing whatever action is required because of the exception, the operating system can terminate the program or may continue its execution, using the EPC to determine where to restart the execution of the program.
- For the operating system to handle the exception, it must know the reason for the exception, in addition to the instruction that caused it.
- There are two main methods used to communicate the reason for an exception.

  The First method used in the MIPS architecture is to include **a status register (called the *Cause* *register*)**, which holds a field that indicates the reason for the exception.

A second method is to use **vectored interrupts**. In a vectored interrupt, the address to which control is transferred is determined by the cause of the exception.

For example, to accommodate the two exception types listed above, we might define the following two exception vector addresses:



| Exception type | Exception vector address (in hex) |
|---|---|
| Undefined Instruction | 8000 0000$_{hex}$ |
| Arithmetic overflow | 8000 0180$_{hex}$ |

The addresses are separated by 32 bytes or eight instructions, and the operating system must record the reason for the exception and may perform some limited processing in this sequence. When the exception is not vectored, a single entry point for all exceptions can be used, and the operating system decodes the status register to find the cause.

Let’s assume that we are implementing the exception system used in the MIPS architecture, with the single entry point being the address 8000 0180hex.(Implementing vectored exceptions is no more difficult.) We will need to add two additional registers to our current MIPS implementation:

■ ***EPC:*** A 32-bit register used to hold the address of the affected instruction. (Such a register is needed even when exceptions are vectored.)

■ ***Cause****:* A register used to record the cause of the exception. In the MIPS architecture, this register is 32 bits, although some bits are currently unused.

### Exceptions in a Pipelined Implementation

For example, suppose there is an arithmetic overflow in an add instruction. we must flush the instructions that follow the add instruction from the pipeline and begin fetching instructions from the new address.

- To flush the instruction in the IF stage by turning it into a nop.
- To flush instructions in the ID stage, we use the multiplexor already in the ID stage that zeros control signals for stalls. A new control signal, called ID.Flush, is ORed with the stall signal from the hazard detection unit to flush during ID.
- To flush the instruction in the EX phase, we use a new signal called EX.Flush to cause new multiplexors to zero the control lines.

To start fetching instructions from location 8000 0180hex, which is the MIPS exception address, we simply add an additional input to the PC multiplexor that sends 8000 0180hex to the PC.

Example

**Exception in a Pipelined Computer**

Given this instruction sequence,

```
40hex         sub $11, $2, $4
44hex         and $12, $2, $5
48hex         or $13, $2, $6
4Chex         add $1, $2, $1
50hex         slt $15, $6, $7
54hex         lw $16, 50($7)
```

assume the instructions to be invoked on an exception begin like this:

```
80000180hex  sw $26, 1000($0)
80000184hex   sw $27, 1004($0)
```

Show what happens in the pipeline if an overflow exception occurs in the add instruction.

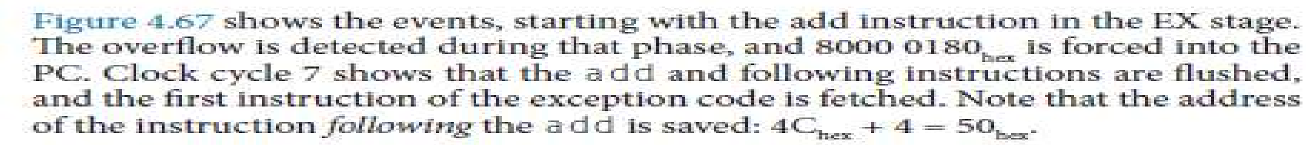


Figure 4.67 shows the events, starting with the add instruction in the EX stage. The overflow is detected during that phase, and $8000\ 0180_{hex}$ is forced into the PC. Clock cycle 7 shows that the `add` and following instructions are flushed, and the first instruction of the exception code is fetched. Note that the address of the instruction *following* the `add` is saved: $4C_{hex} + 4 = 50_{hex}$.

# UNIT IV PARALLELISM

Parallel processing challenges – Flynn’s classification – SISD, MIMD, SIMD, SPMD, and Vector Architectures - Hardware multithreading – Multi-core processors and other Shared Memory Multiprocessors - Introduction to Graphics Processing Units, Clusters, Warehouse Scale Computers and other Message-Passing Multiprocessors.

## Instruction Level Parallelism

The parallelism among instructions is called Instruction Level Parallelism.

**<u>There are two primary methods for increasing the potential amount of instruction-level parallelism</u>**

- ➢ **Increasing the depths of pipeline**
- **Multiple Issue:**Replicate the internal components of the computer so that it can issue multiple instructions in a clock cycle. The general name of this technique is multiple issues. **SuperScalar – Issue varying number of instructions per clock**

- ➢ **There are two types of multiple issue processor**
- ➢ **static multiple issue** An approach to implementing a multiple-issue processor where many decisions are made by the compiler before execution.
- ➢ D**ynamic multiple issue** An approach to implementing a multiple issue processor where many decisions are made during execution by the processor.
- ➢ **Issue slots** The positions from which instructions could issue in a given clock cycle.

## Static Multiple Issue(Example VLIW-Very Long Instruction Word)

Static multiple-issue processors all use the compiler to assist with packaging instructions and handling hazards.

## Very Long Instruction Word (VLIW)

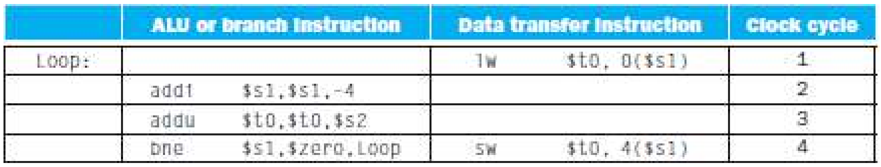
A style of instruction set architecture that launches many operations that are defined to be independent in a single wide instruction, typically with many separate opcode fields.

## Simple Multiple-Issue Code Scheduling (CPI of 0.8 )

How would this loop be scheduled on a static two-issue pipeline for MIPS?

```
Loop:  lw $t0, 0($s1)          # $t0=array element
       addu $t0,$t0,$s2        # add scalar in $s2
       sw $t0, 0($s1)          # store result
       addi $s1,$s1,–4         # decrement pointer
       bne $s1,$zero,Loop      # branch $s1!=0
```

Reorder the instructions to avoid as many pipeline stalls as possible.



| | ALU or branch instruction | Data transfer instruction | Clock cycle |
|---|---|---|---|
| Loop: | | lw $t0, 0($s1) | 1 |
| | addi $s1,$s1,-4 | | 2 |
| | addu $t0,$t0,$s2 | | 3 |
| | bne $s1,$zero,Loop | sw $t0, 4($s1) | 4 |

**FIGURE 4.70 The scheduled code as it would look on a two-issue MIPS pipeline.** The empty slots are no-ops.

# Dynamic Multiple-Issue Processors(Hardware Approach)Superscalar Processor.

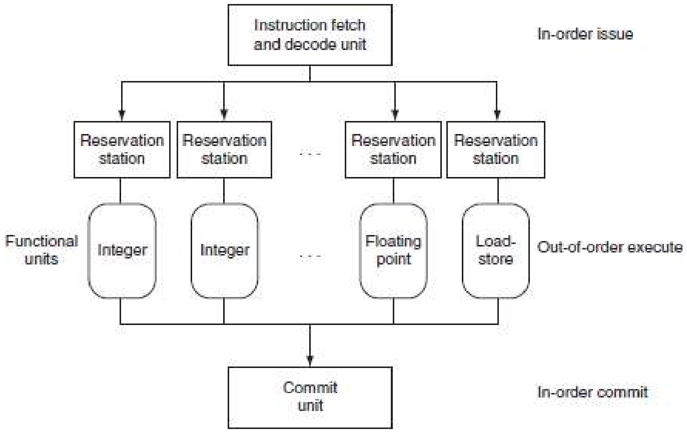


**FIGURE 4.72 The three primary units of a dynamically scheduled pipeline.** The final step of updating the state is also called retirement or graduation.

**Dynamic Pipeline Scheduling**

Dynamic pipeline scheduling chooses which instructions to execute next, possibly reordering them to avoid stalls.

In such processors, the pipeline is divided into three major units:

- instruction fetch
- issue unit
- multiple functional units
- a commit unit

The first unit fetches instructions, decodes them, and sends each instruction to a corresponding functional unit for execution. Each functional unit has buffers, called **reservation stations**, which hold the operands and the operation. As soon as the buffer contains all its operands and the functional unit is ready to execute, the result is calculated. When the result is completed, it is sent to any reservation stations waiting for this particular result as well as to the commit unit, which buffers the result until it is safe to put the result into the register file or, for a store, into memory. The buffer in the commit unit, often called the **reorder buffer**, is also used to supply operands, in much the same way as forwarding logic does in a statically scheduled pipeline. Once a result is committed to the register file, it can be fetched directly from there, just as in a normal pipeline.

## Consider the following steps:

1. When an instruction issues, it is copied to a reservation station for the appropriate functional unit. Any operands that are available in the register file or reorder buffer are also immediately copied into the reservation station. The instruction is buffered in the reservation station until all the operands and the functional unit is available. For the issuing instruction, the register copy of the operand is no longer required, and if a write to that register occurred, the value could be overwritten.

2. If an operand is not in the register file or reorder buffer, it must be waiting to be produced by a functional unit. The name of the functional unit that will produce the result is tracked. When that unit eventually produces the result, it is copied directly into the waiting reservation station from the functional unit bypassing the registers. These steps effectively use the reorder buffer and the reservation stations to implement register renaming.

## PARALLEL PROCESSING CHALLENGES : (Developing parallel processing program in harder/difficult than sequential program. Explain)

**Parallel processing program:**

A single program which runs on multiple processors simultaneously is called parallel processing program. It is difficult to develop S/W that uses multiple processors to complete one task faster.

The difficulties are:

1. Sequential portion restricts speedup.
2. Speedup is function of problem size
3. Speedup is function of load balancing.
4. Communication & Synchronization.

These problems gets worse or stiffer when the number of processors increases.

## Sequential portion restricts speedup.

Suppose one want to achieve a speedup of 90 times faster with 100 processors. What percentage of the original code can be sequential?

$$\text{Execution Time after improvement} = \frac{Executiontimeaffectedbyimprovement}{Amountof\ improvement} + ExecutionTimeunaffected$$

$$Speedup = \frac{\text{Execution Time before}}{Execution\ \text{Time after Improvemen t}}$$

$$= \frac{Execution\text{Time before}}{\text{ExecutionTime Unaffecte d}+\frac{\text{ExecutionTime affected by improvement}}{\text{Amount of improvemnt}}}$$

$$\frac{Execution\ \text{time before}}{(\text{Execution time before - Execution Time affected by improvement}) + \frac{\text{Execution time affected by improvement}}{\text{improvement factor}}}$$

$$= \frac{1}{1 - k + \frac{k}{n}}$$

Where k - portion of parallisable code

n - improvement factor or no of processor

$$speedup = 90 = \frac{1}{1 - k + \frac{k}{100}}$$

90 - 90k + .9k = 1

90-1 = 90k - .9k = 89.1k

$$\therefore \text{k} = \frac{89}{89.1} = .999$$

Sequential code is =0.1%

## Speedup is function of problem size.

Initial problem=10addition+10 x10 matrix addition

10 addition is sequential and matrix addition is parallelizable

Execution Time for Uniprocessor = 10t+100t =110t where t is addition time

Execution Time in 10 processor = $10t + \frac{100}{10} t = 20t$

$$speedup = \frac{110t}{20t} = 5.5$$

$$ExecutionTime \text{ in 40 processor} = 10t + \frac{100t}{40t} = 12.5t$$

$$speedup = \frac{110t}{12.5t} = 8.8$$

We get potential speedup 55% for 10 processor & 22% for 40 processor.

Now we change the matrix size to 20 x 20

Execution time in Uniprocessor = 10t+400t = 410t

$$ExecutionTime \text{ in 10 processor} = 10t + \frac{400}{10} t = 50t$$

$$speedup = \frac{410t}{50t} = 8.2$$

Execution Time for 40 processor = 10t+10t = 20t

$$speedup = \frac{410t}{20t} = 20.5$$

Potential speedup 82% for 10 processor & 51% for 40 processors.

| Problem Size | Speedup(10 processor) | Speed up(10 processor) |
|---|---|---|
| 10 x 10 | 55 % | 22% |
| 20 x 20 | 82% | 51 % |

Hence the speedup increases with problem size.
**Strong scaling:** Strong scaling means measuring speed-up while keeping the problem size fixed.
**Weak scaling:** Weak scaling means that the problem size grows proportionally to the increase in the number of processors.

## Speedup is function of load balancing

The problem has 400 addition and this was carried 40 processors and each processor has 2.5% load & speedup obtained is 20.5. Impact on Speedup can be seen of one processor

a) 5% load & b) 12.5% load. How well other processors are utilized.

Sequential Execution Time = 410t

5% load = 400/100 X 5 =20

Balance load = 400-20=380

$$ExecutionTime = \max\{\frac{380}{39}, \frac{20}{1}\}t + 10t = 30t$$

$$speedup = \frac{410t}{30t} = 13.66$$

$$12.5\%\ load = \frac{400}{100} X 12.5 = 50$$

Balance load = 350

$$ExecutionTime = \max\{\frac{350}{39}, \frac{50}{1}\}t + 10t = 60t$$

$$speedup = \frac{410t}{60t} = 6.83$$

Hence speedup drops from 20.5 to 14 for 5% load and 7 for 12.5% load on heavily loaded processor .This example shows the dropping of speedup with unbalance in load.

## FLYNN CLASSIFICATION OF COMPUTER ARCHITECTURE

- In 1966, Michael Flynn proposed a classification for computer architectures based on the number of instruction steams and data streams (Flynn's Taxonomy).

- Flynn uses the stream concept for describing a machine's structure
- A stream simply means a sequence of items (data or instructions).
- The classification of computer architectures based on the number of instruction steams and data streams (Flynn's Taxonomy).

## Flynn's Taxonomy

SISD: Single instruction single data
SIMD: Single instruction multiple data
MISD: Multiple instructions single data
MIMD: Multiple instructions multiple data

### 1)Single Instruction And Single Data Stream

A sequential computer which exploits no parallelism in either the instruction or data streams. Single control unit (CU) fetches single Instruction Stream (IS) from memory. The CU then generates appropriate control signals to direct single processing element (PE) to operate on single Data Stream (DS) i.e. one operation at a time.

Examples of SISD architecture are the traditional uniprocessor machines like a PC (currently manufactured PCs have multiple processors) or old mainframes.

### 2)Single Instruction And Multiple Data Stream

A computer which exploits multiple data streams against a single instruction stream to perform operations which may be naturally parallelized. For example, an array processor.

- Each instruction is executed on a different set of data by different processors i.e multiple processing units of the same type process on multiple-data streams.
- This group is dedicated to array processing machines.
- Sometimes, vector processors can also be seen as a part of this group.

3)**Multiple Instruction And Single Data Stream**

- Multiple instructions operate on a single data stream. Uncommon architecture which is generally used for fault tolerance. Heterogeneous systems operate on the same data stream and must agree on the result. Examples include the Space Shuttle flight control computer.
- MISD (Multiple-Instruction streams, Singe-Data stream)
- Each processor executes a different sequence of instructions.
- In case of MISD computers, multiple processing units operate on one single-data stream .
- In practice, this kind of organization has never been used

4)**Multiple Instruction And Multiple Data Stream**

Multiple autonomous processors simultaneously executing different instructions on different data. Distributed systems are generally recognized to be MIMD architectures; either exploiting a single shared memory space or a distributed memory space. A multi-core superscalar processor is an MIMD processor.

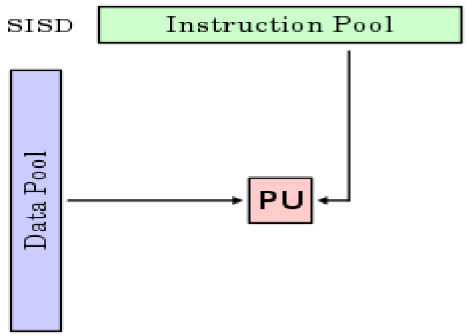
- **MIMD** (**M**ultiple-**I**nstruction streams, **M**ultiple-**D**ata streams)
- Each processor has a separate program.
- An instruction stream is generated from each program.
- Each instruction operates on different data.
- This last machine type builds the group for the traditional multi-processors. Several

processing units operate on multiple-data streams.

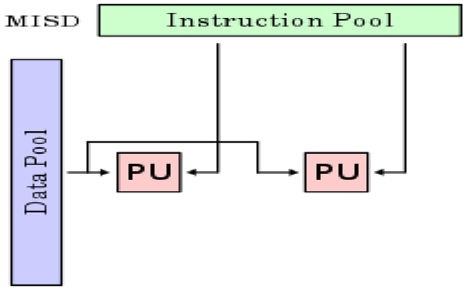
Diagram comparing classifications

Visually, these four architectures are shown below where each "PU" is a central processing unit:

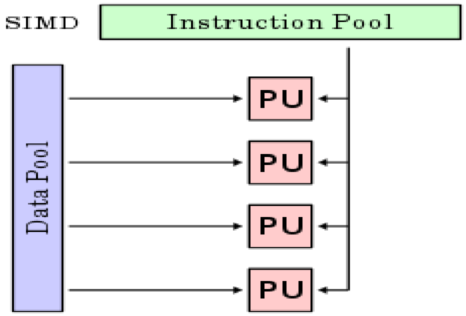
**SISD**



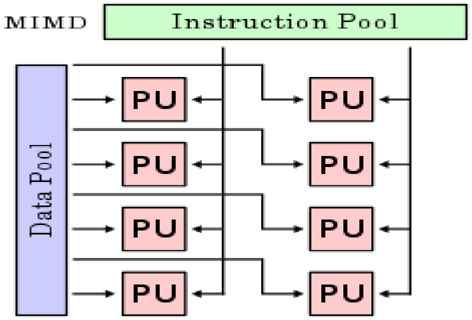
**MISD**



**SIMD**



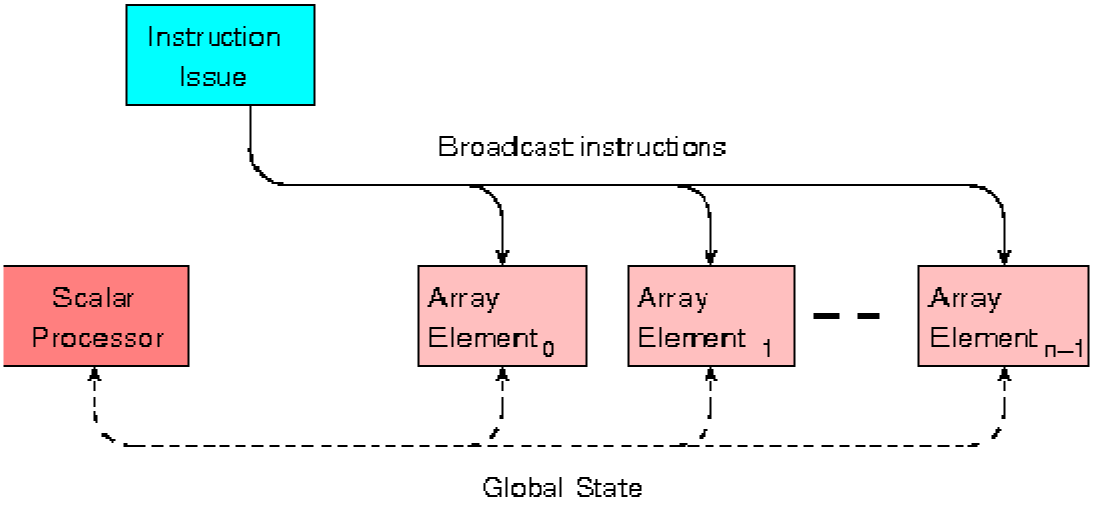
**MIMD**



SIMD stands for single instruction stream and multiple data streams. Early implementation of SIMD concepts is as array processor, later as vector processor, and recently as MMX & SSE instructions of every micro processor.

# Array Processor

A single SIMD instruction might add 64 numbers by sending 64 data streams to 64 ALU to form 64 sum with in single clock cycle. Such system is called array processor and now it is almost disappeared.



## Vector processor:

Vector processor is another implementation of SIMD architecture. This was implemented by SeyMour Cray in Cray super computers. Instead of using 64 ALU performing 64 additions simultaneously like old array processors, this vector architecture pipelined the ALU to get good performance. Here the data elements are transferred from memory and loaded into large set of register and operate them sequentially using pipelined execution units and then write back the results into memory. A key feature of vector architecture is the set of vector registers.

Vector processor may have 32 nos vector registers each with 64 nos of 64-bit registers. All modern vector computers have vector functional unit with multiple parallel pipelines called vector lanes. Generally vector architecture are very efficient to execute instructions with data vectors.

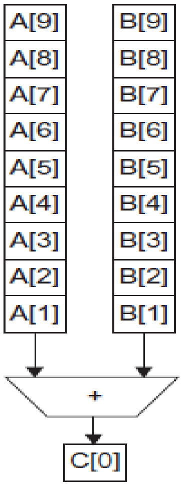


Fig: Single add pipeline can complete one add/clock

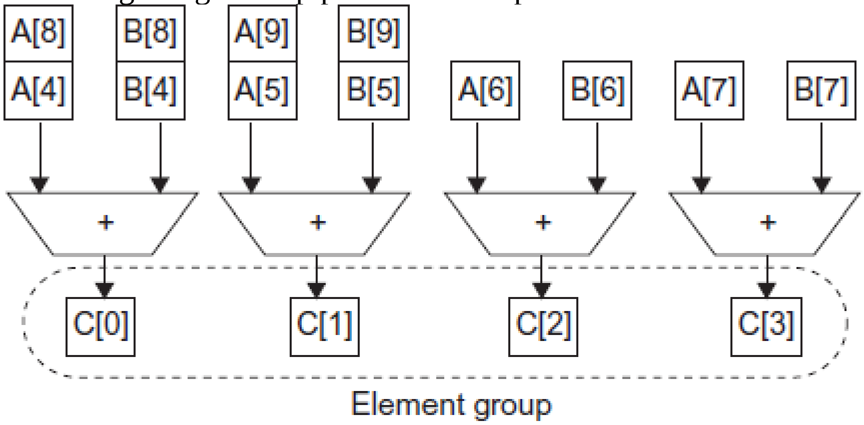


Fig: Four add pipeline can complete 4 add/clock

**Difference between Vector architecture and Multimedia extension:**

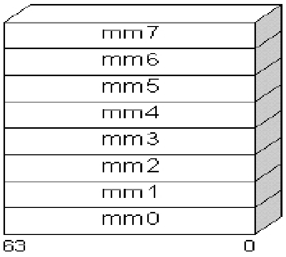
| s.no | Vector architecture | Multimedia Extension |
|---|---|---|
| 1 | It specifies dozens of operations. | It specifies a few operations. |
| 2 | Number of elements in a vector operation is not in the opcode. | Number of elements in a multimedia extension. Operation is not in the opcode. |
| 3 | Vector has data transfers need not be contiguous. | Multimedia extension has data transfers need to be contiguous. |
| 4 | It specifies multiple operations. | It alsospecifies multiple operations. |

**MMX & SSE of Intel 80x86 processor.**

This is mostly widely used implementation of SIMD and this is found in almost every microprocessor. The implementation of Intel 80x86 processor is given below.

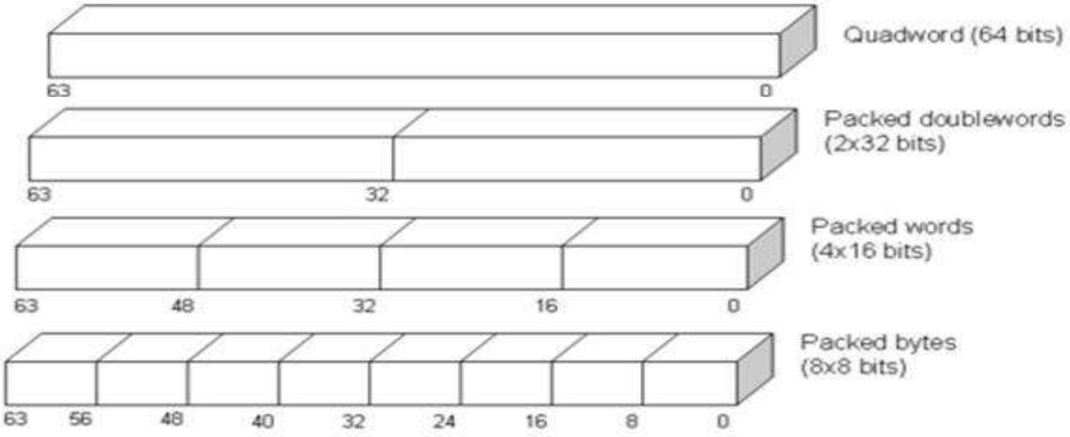
## MMX for Intel 80 x 86

- MMX (multi media Extension) was the first set of SIMD extensions applied to Intel 80x86 processor( Pentium MMX)
- This was introduced in 1997
- MMX provides 8 registers of 64 bit.
- These registers are called MM0 –MM7



MMX REGISTER SET

- These register can be used as
  - One 64 bit Quad word
  - Two 32 bits double word
  - Four 16 bits word
  - Eight 8 bits byte



# MMX DATA TYPE

- MMX has 57 instructions and a few MMX instructions are listed

### MMX - Data Movement

| | | |
|---|---|---|
| movd | - | Move double word |
| movq | - | Move Quad word |

### MMX - Boolean Logic

| | | |
|---|---|---|
| por | - | bitwise OR |
| pand | - | AND |

### MMX - Arithmetic

| | | |
|---|---|---|
| padd b | - | adds mmx registers as unsigned 8 bit byte |
| padd sb | - | adds as signed 8 bit byte |
| padd w | - | adds as unsigned 16 bit word |
| padd sw | - | adds as signed 16 bit word |

### MMX - Compare

| | | |
|---|---|---|
| pcmpeqb | - | Compare for 8 bit equality |
| pcmeqw | - | compare for 16 bit equality |

- The later SIMD extensions for 80x86 are called SSE, SSE2, SSE3, SSE4 and AVX.

## HARDWARE MULTITHREADING

**Threads:** multiple processes that share code and data (and much of their address space) recently, the term has come to include processes that may run on different processors and even have disjoint address spaces, as long as they share the code.

**Hardware multithreading** allows multiple threads to share the functional units of a single processor in an overlapping fashion. To permit this sharing, the processor must duplicate the independent state of each thread.

There are two main approaches to hardware multithreading.

- Fine-grained Multithreading
- Coarse-grained multithreading
- Simultaneous multithreading

**Fine-grained Multithreading** switches between threads on each instruction, resulting in interleaved execution of multiple threads. This interleaving is often done in a round-robin fashion, skipping any threads that are stalled at that clock cycle. To make fine-grained multithreading practical, the processor must be able to switch threads on every clock cycle.

Advantage

It can hide the throughput losses that arise from both short and long stalls, since instructions from other threads can be executed when one thread stalls.

Disadvantage

It slows down the execution of the individual threads, since a thread that is ready to execute without stalls will be delayed by instructions from other threads.

## Coarse-grained multithreading

- Coarse grained multithreading is a version of hardware multithreading that implies switching between thread only after significant events such as last level cache miss.

- Switches threads only on costly stalls, such as second-level cache misses.
- This change relieves the need to have thread switching be extremely fast and is much less likely to slow down the execution of an individual thread, since instructions from other threads will only be issued when a thread encounters a costly stall.
- Multithreading issues instructions from a single thread, when a stall occurs, the pipeline must be emptied or frozen.
- The new thread that begins executing after the stall must fill the pipeline before instructions will be able to complete. Due to this start-up overhead, coarse-grained multithreading is much more useful for reducing the penalty of high-cost stalls, where pipeline refill is negligible compared to the stall time.

## Disadvantage

It is limited in its ability to overcome throughput losses, especially from shorter stalls.

**Simultaneous multithreading (SMT)** is a variation on hardware multithreading that uses the resources of a multiple-issue, dynamically scheduled **pipelined** processor to exploit thread-level parallelism at the same time it exploits instruction level parallelism. Since SMT relies on the existing dynamic mechanisms, it does not switch resources every cycle. Instead, SMT is *always* executing instructions from multiple threads, leaving it up to the hardware to associate instruction slots and renamed registers with their proper threads.
Shows how the four threads could be combined to execute on the processor more efficiently using three multithreading options:

- A superscalar coarse-grained multithreading
- A superscalar with fine-grained multithreading
- A superscalar with simultaneous multithreading

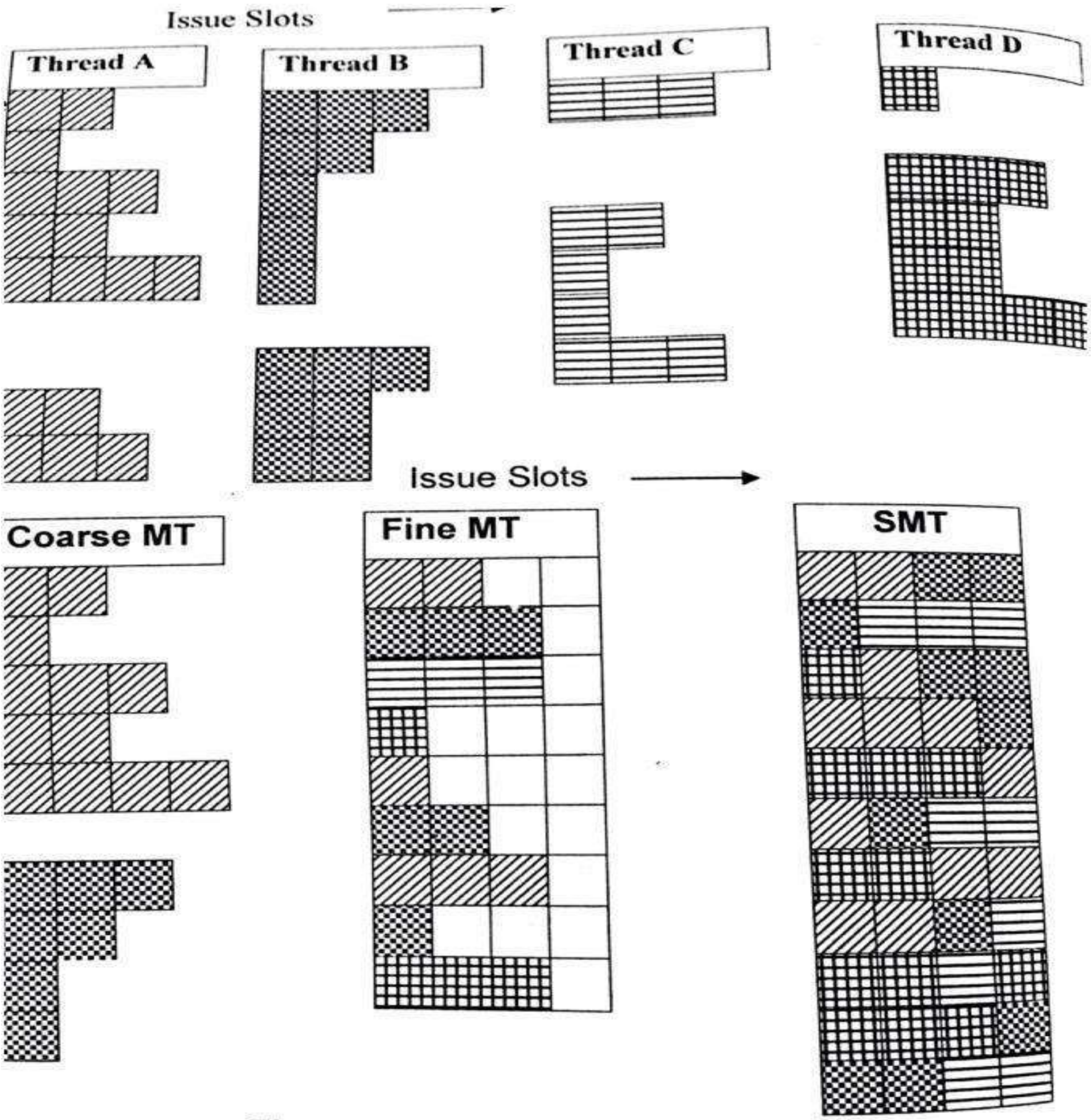


Figure 5.10 Three types of multithreading

# Multi-core processors and other Shared Memory Multiprocessors

## Multicore Processor :

A **multi-core processor** is a single computing component with two or more independent actual central processing units (called "cores"), which are the units that read and execute program instructions. The instructions are ordinary CPU instructions such as add, move data, and branch, but the multiple cores can run multiple instructions at the same time, increasing overall speed for programs agreeable to parallel computing. Manufacturers typically integrate the cores onto a single integrated circuit die (known as a chip multiprocessor or CMP), or onto multiple dies in a single chip package.

***Homogeneous* multi-core systems** include only identical cores, **Heterogeneous multi-core** systems have cores that are not identical.

Just as with single-processor systems, cores in multi-core systems may implement architectures such as superscalar, VLIW, vector processing, SIMD, or multithreading.

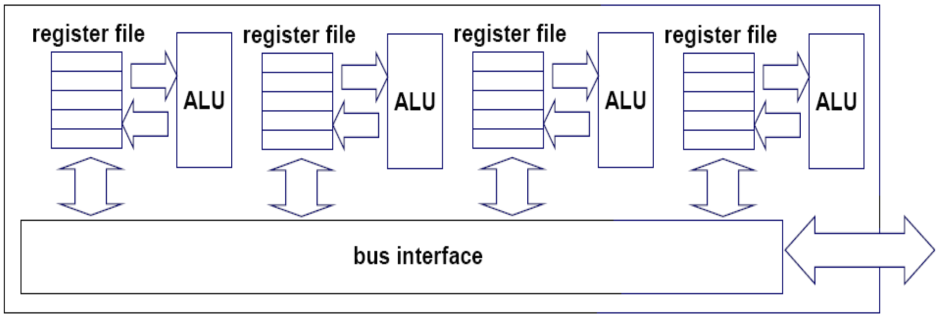
Multi-core processors are widely used across many application domains including general-purpose, embedded, network, digital signal processing (DSP), and graphics.

A Microprocessor containing multiple processors (called cores) in a single Integrated Circuit is called Multicore microprocessor. Dual core means two core and Quad core means four core. A multicore processor with four core is shown below.

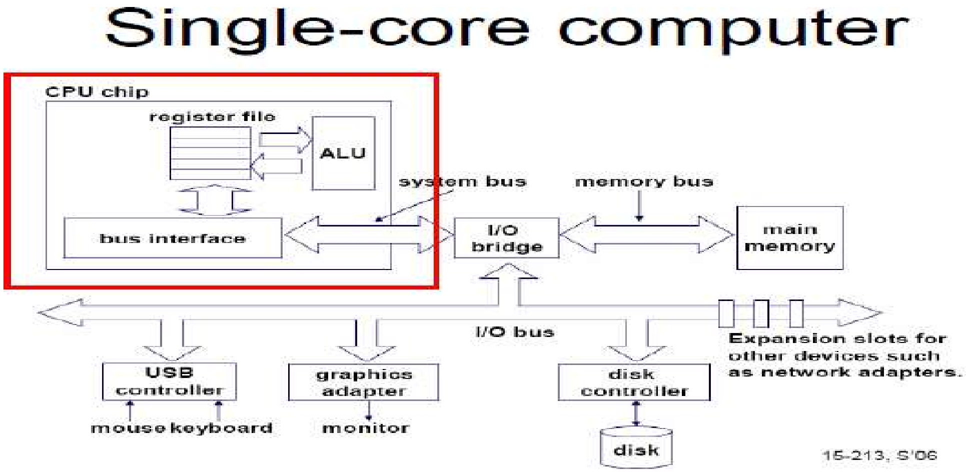
Examples of multicore processors

- ❖ Intel Core i3 - Dual cores
- ❖ Intel Corei7 - Quad cores
- ❖ Intel xenon phi - More than 50 cores

## Multi-core architecture



- ▪ Replicate multiple processor cores on a single die.
- ▪ The cores fit on a single processor socket.
- ▪ The cores run in parallel(like on a uniprocessor)



## Advantages:

1. Cache coherency circuitry can operate at a much higher clock rate than is possible if the signals have to travel off-chip.
2. Signals between different CPUs travel shorter distances, those signals degrade less.
3. These higher quality signals allow more data to be sent in a given time period since individual signals can be shorter and do not need to be repeated as often.
4. A dual-core processor uses slightly less power than two coupled single-core processors.
5. Multi-core chips also allow higher performance at lower energy. This can be a big factor in mobile devices that operate on batteries.

## Disadvantages

- **Maximizing the utilization of the computing resources** provided by multi-core processors requires adjustments both to the operating system (OS) support and to existing application software.
- **Integration of a multi-core chip** drives chip production yields down and they are **more difficult to manage thermally than lower-density single-chip designs.**
- Intel has partially countered this first problem by creating its quad-core designs by combining two dual-core on a single die with a unified cache, hence any two working dual-core dies can be used, as opposed to producing four cores on a single die and requiring all four to work to produce a quad-core.
- From an architectural point of view, ultimately, single CPU designs may make better use of the silicon surface area than multiprocessing cores, so a development commitment to this architecture may carry the risk of obsolescence.
- Finally, **raw processing power is not the only constraint on system performance**. Two processing cores sharing the **same system bus and memory bandwidth** limits the real-world performance advantage.

### Shared Memory Multiprocessors

A computer system with two or more processor is called multiprocessor. This can be classified into types

1. Shared Memory Multiprocessor(SMP).
2. Message passing Multi processor or cluster. The organization of shared memory multi processor is shown below

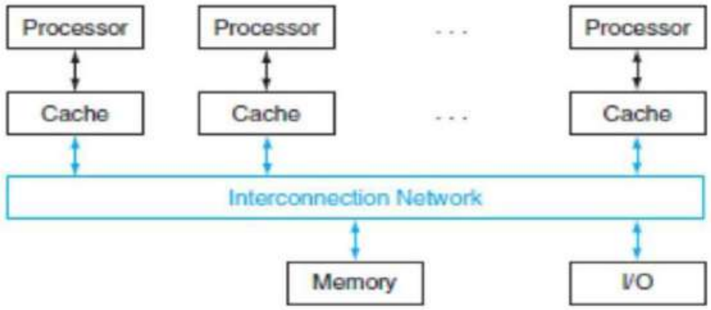


**FIGURE 6.7 Classic organization of a shared memory multiprocessor.**

Here all the processors share a common single address space as well as single common physical memory. Such system will have one as and each system can run independent job in their own virtual address space. Processor communicates through shared variable in memory. As processors operating in parallel will normally share data otherwise, one processor could start working on data before another is finished with it.

### SMP come in two styles.

1. Uniform memory access multi processor(UMA).
2. Non uniform memory access multiprocessor(NUMA).

## Uniform Memory Access (UMA) Multiprocessor

A multiprocessor in which accesses to main memory take about the **same amount** of time no matter which processor request the access and no matter which word is asked.

## Non Uniform Memory Access (NUMA)Multiprocessor

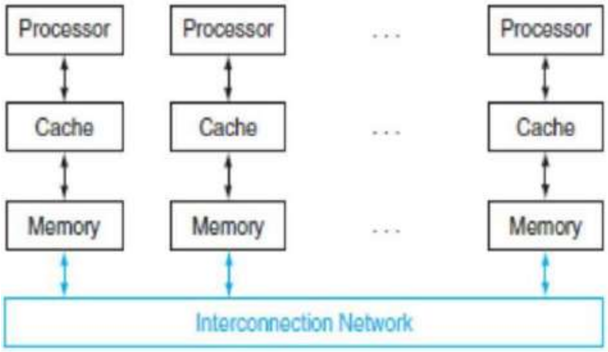
A multiprocessor in which some memory accesses are much faster than others depending on which processor asks for which word.
Here each processor has some dedicated local memory and also in addition they have common memory. Hence local memory can be accessed by the local processor at a faster rate and common memory can be accessed at slow rate. Such machines are called Non Uniform Memory Access multiprocessor.

## Advantages:

- ❖ NUMA has lower latency for local memory
- ❖ This can scale to large size

**Synchronization:** The process of coordinating the behavior of two or more processes, which may be running on different processors.
**Lock:** A synchronization device that allows access to data to only one processor at a time.

The organizing of a multiprocessor with **message passing** is shown below



Here the each processor has its own physical memory address space and the processor are connected by inter connection network. The processor communicate via explicit message passing such system is also called cluster. Application with little communication like web search, mail server and file server do not require shared memory multiprocessor.

**Advantage:** Message passing multiprocessor or cluster is scalable to thousands of nodes.

## Draw back of cluster compared shared memory multiprocessor:

1)Cluster need N operating systems compare to single copy of OS in SMP.
2)Processor in cluster are usually connected using I/O interconnect. whereas the processors in SMP are usually connected on the memory interconnect.

3)Cluster administering is equal to administering of N machines. Where as the administering SMP is same as administering single machine.

In the symmetric memory architecture single main memory has a relationship to all processors and a uniform access time from any processor, these multiprocessor are most often called as symmetric memory architecture sometimes called **uniform memory access.**

The use of large multilevel caches can substantially reduce the memory bandwidth demands of a processor. If the main memory bandwidth demands of a single processor are reduced, multiple processors may be able to share the same memory

Symmetric memory supports both **shared and private data. Private data are used by a single processor, while shared data is used by multiple processors**;

This symmetric shared memory contains 2 problem.

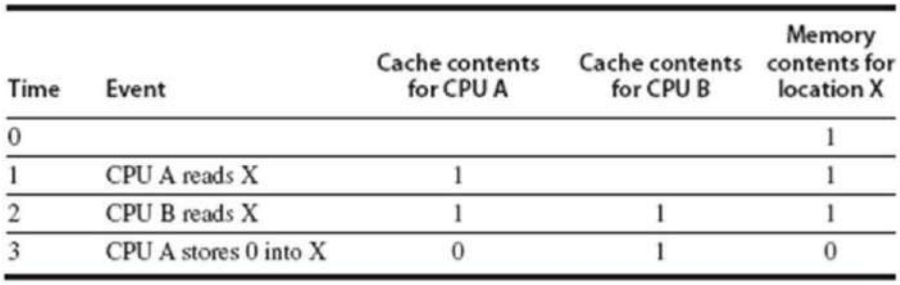
## PROBLEM 1:

Caching of a shared data introduces new problem. Which is called as a cache coherence.

Two different processor can have two different values for the same location. This is called as **Cache Coherence.** It defines what values can be returned by a read instruction or operation. PROBLEM 2: **CONSISTENCY:**Consistency determines when a written value will be returned by a read operation.

A cache coherence problem for a single memory location X, read and write by 2-processor A,B.

EXAMPLE: The cache coherence problem for a single Memory location(x), read and written by 2 processor( A and B).



| Time | Event | Cache contents for CPU A | Cache contents for CPU B | Memory contents for location X |
|---|---|---|---|---|
| 0 | | | | 1 |
| 1 | CPU A reads X | 1 | | 1 |
| 2 | CPU B reads X | 1 | 1 | 1 |
| 3 | CPU A stores 0 into X | 0 | 1 | 0 |

THE SNOOPING PROTOCOL IS USED in the symmetric shared memory based protocol. Which is the hardware approach to overcome the cache coherence? It has two types.

invalidation protocol or write invalidate protocol

write update or broadcast protocol

## Distributed Shared Memory Architecture

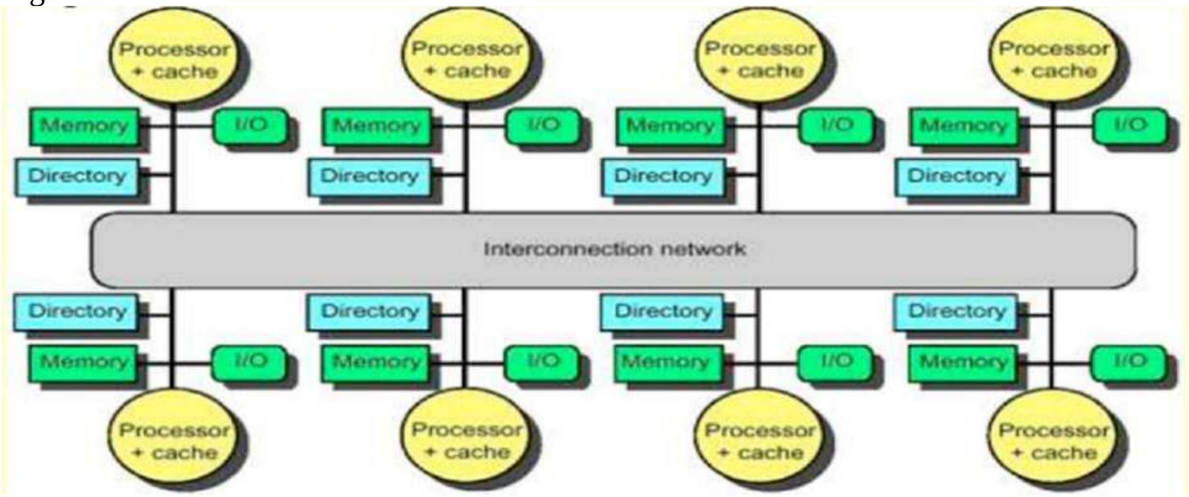
- Each processor has its own memory system, which it can access directly
- To obtain data that is stored in some other processor‘s memory, a processor must communicate with that to request the data

### Advantages

Each processor has its own local memory system.More total bandwidth in the memory system than in a centralized memory system The latency to complete a memory request is lower─ each processor‘s memory is located physically close to it

### Disadvantages

Only some of the data in the memory is directly accessible by each processor, since a processor can only read and write its local memory system. Leads to different processors having different values for the same variable



## INTRODUCTION TO GRAPHICS PROCESSING UNITS

**Introduction:**
**Graphics processing unit (GPU)** A processor optimized for 2D and 3D graphics, video, visual computing, and display.
**Visual computing:** A mix of graphics processing and computing that lets you visually interact with computed objects via graphics, images, and video.
**Heterogeneous system**: A system combining diff erent processor types. A PC is a heterogeneous CPU–GPU system.

## GPU Evolution

Graphics on a PC were performed by a **video graphics array (VGA) controller**.
By 1997, **VGA controllers** were beginning to incorporate some three-dimensional (3D) acceleration functions, including hardware for triangle setup and rasterization (dicing triangles into individual pixels) and texture mapping and shading (applying “decals” or patterns to pixels and blending colors).
**In 2000**, the single chip graphics processor incorporated almost every detail of the traditional high-end workstation graphics pipeline and, therefore, deserved a new name beyond VGA controller. The term **GPU was coined to denote that the graphics device had become a processor.**
**GPUs have become massively parallel programmable processors with hundreds of cores and thousands of threads**. Recently, processor instructions and memory hardware were added to support general purpose programming languages, and a programming environment was created to allow GPUs to be programmed using familiar languages, including C and C. This innovation makes a GPU a fully general-purpose, programmable, many core processor, albeit still with some special benefits and limitations.

## GPU Graphics Trends

GPUs and their associated drivers implement the OpenGL and DirectX models of graphics processing. OpenGL is an open standard for 3D graphics programming available for most computers. DirectX is a series of Microsoft multimedia programming interfaces, including Direct3D for 3D graphics.

**Application programming interface (API)**:A set of function and data structure definitions providing an interface to a library of functions

## Why CUDA and GPU Computing?

GPU computing Using a GPU for computing via a parallel programming language and API. (General Purpose computation on GPU)GPGPU: Using a GPU for general-purpose computation via a traditional graphics API and graphics pipeline.

(Compute Unifed Device Architecture)CUDA: A scalable parallel programming model and language based on C/C. It is a parallel programming platform for GPUs and multicore CPUs

## Basic Unifed GPU Architecture

Unified GPU architectures are based on a parallel array of many programmable processors. They unify vertex, geometry, and pixel shader processing and parallel computing on the same processors, unlike earlier GPUs which had separate processors dedicated to each processing type. The programmable processor array is tightly integrated with fixed function processors for texture filtering, rasterization, raster operations, anti-aliasing, compression, decompression, display, video decoding, and high-definition video processing. Although the fixed-function processors significantly outperform more general programmable processors in terms of absolute performance constrained by an area, cost, or power budget, we will focus on the programmable processors here. Compared with multicore CPUs, manycore GPUs have a different architectural design point, one focused on executing many parallel threads efficiently on many processor cores. By using many simpler cores and optimizing for data-parallel behavior among groups of threads, more of the per- chip transistor budget is devoted to computation, and less to on-chip caches and overhead.

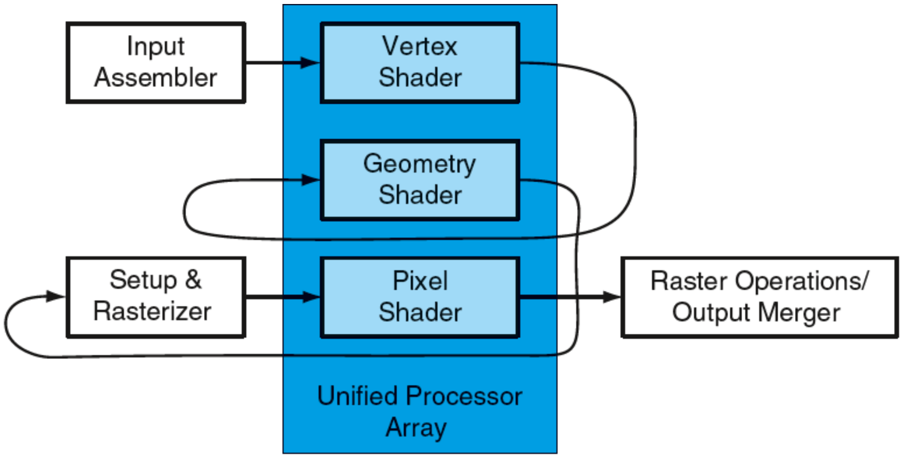


**FIGURE C.2.4 Logical pipeline mapped to physical processors.** The programmable shader stages execute on the array of unified processors, and the logical graphics pipeline dataflow recirculates through the processors.

**Shader** A program that operates on graphics data such as a vertex or a pixel fragment.
**Shading language** A graphics rendering language, usually having a dataflow or streaming programming model.

## The CUDA Paradigm

CUDA is a minimal extension of the C and C programming languages. The programmer writes a serial program that calls parallel kernels, which may be simple functions or full programs
**kernel** A program or function for one thread, designed to be executed by many threads.
**Thread block** A set of concurrent threads that execute the same thread program and may cooperate to compute a result.

**Grid** A set of thread blocks that execute the same kernel program.
Parallel execution and thread management is automatic.Threads may access data from multiple memory spaces during their execution. Each thread has a private local memory.
**local memory** Perthread local memory private to the thread.
**shared memory** memory shared by all threads of the block.
**global memory** Perapplication memory shared by all threads.

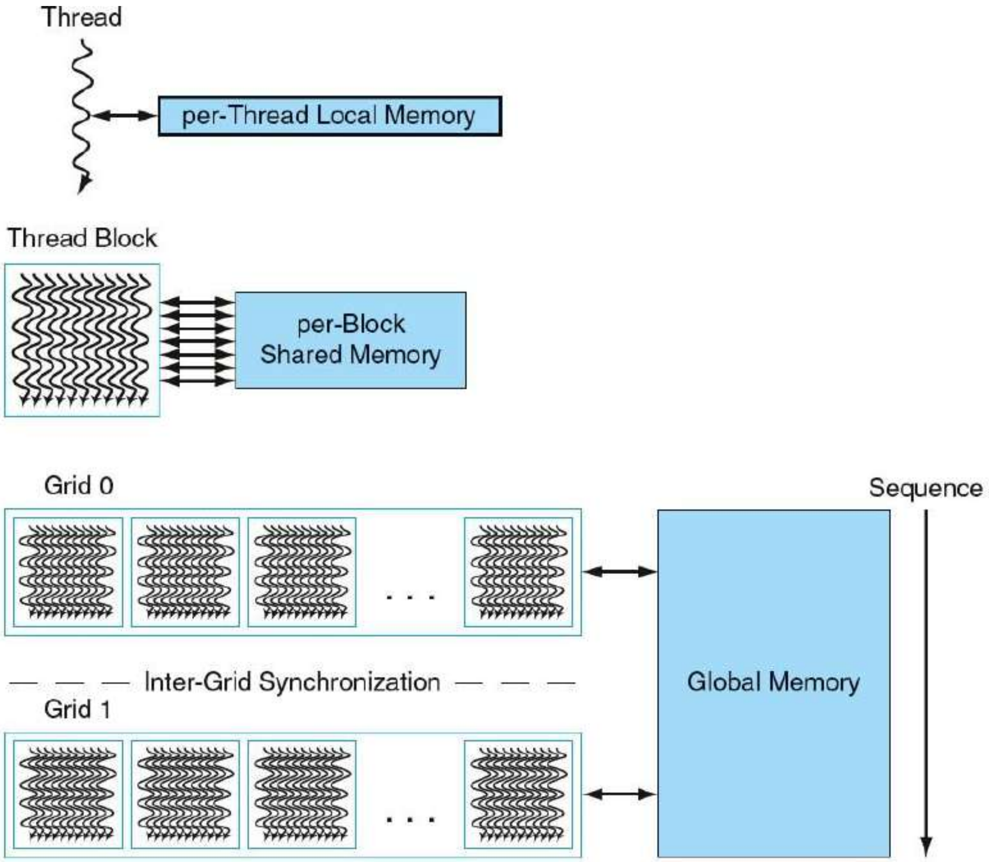


**FIGURE C.3.5 Nested granularity levels—thread, thread block, and grid—have corresponding memory sharing levels—local, shared, and global.** Per-thread local memory is private to the thread. Per-block shared memory is shared by all threads of the block. Per-application global memory is shared by all threads.

## Implications for Architecture

The parallel programming models for graphics and computing have driven GPU architecture to be different than CPU architecture.

The key aspects of GPU programs driving GPU processor architecture are:

■ Extensive use of fine-grained data parallelism: Shader programs describe how to process a single pixel or vertex, and CUDA programs describe how to compute an individual result.

■ Highly threaded programming model: A shader thread program processes a single pixel or vertex, and a CUDA thread program may generate a single result. A GPU must create and execute millions of such thread programs per frame, at 60 frames per second.

■ Scalability: A program must automatically increase its performance when provided with additional processors, without recompiling.

■ Intensive floating-point (or integer) computation.

■ Support of high throughput computations.

## COMPUTER ARCHITECTURE OF WAREHOUSE-SCALE COMPUTERS

**Ware House Scale Computers:**

Warehouse-scale computers that basically exploit request level parallelism and data level parallelism.

A Warehouse-scale computer(WSC) is a cluster comprised of tens of thousands of servers.

Warehouse-scale computers (WSCs) form the foundation of internet services that people use for search, social networking, online maps, video sharing, online shopping, email, cloud computing, etc.

**WSCs share many goals and requirements with servers.** They are:

**Cost-performance**: Because of the scale involved, the cost-performance becomes very critical. Even small savings can amount to a large amount of money.

**Energy efficiency:** There is a lot of money invested in power distribution for consumption and also to remove the heat generated through cooling systems). Hence, work done per joule is critical for both WSCs and servers because of the high cost of building the power and mechanical infrastructure for a warehouse of computers and for the monthly utility bills to power servers. If servers are not energy-efficient they will increase (1) cost of electricity, (2) cost of infrastructure to provide electricity, (3) cost of infrastructure to cool the servers. Therefore, we need to optimize the computation per joule.

**Dependability via redundancy:** The hardware and software in a WSC must collectively provide at least 99.99% availability, while individual servers are much less reliable. Redundancy is the key to dependability for both WSCs and servers. While servers use more hardware at higher costs to provide high availability, WSC architects rely on multiple cost-effective servers connected by a low cost network and redundancy managed by software. Multiple WSCs may be needed to handle faults in whole WSCs. Multiple WSCs also reduce latency for services that are widely deployed.

**Network I/O:** Networking is needed to interface to the public as well as to keep data consistent between multiple WSCs.

**Both interactive and batch-processing workloads:** Search and social networks are interactive and require fast response times. At the same time, indexing, big data analytics etc. create a lot of batch processing workloads also.

WSCs also have characteristics that are not shared with servers. They are:

**Ample parallelism:** In the case of servers, we need to worry about the parallelism available in applications to justify the amount of parallel hardware. This is not the case with WSCs. Most jobs are totally independent and exploit "Request-level parallelism". Interactive internet service applications, known as Software as a Service (SaaS) workload consists of independent requests of millions of users. Secondly, data of many batch applications can be processed in independent chunks, exploiting data-level parallelism.

**Operational costs count:** Server architects normally design systems for peak performance within a cost budget. Power concerns are not too much as long as the cooling requirements are maintained. The operational costs are ignored. WSCs, however, have a longer life times and the building, electrical and cooling costs are very high. So, the operational costs cannot be ignored.

**Scale and its opportunities and problems:** Since WSCs are so massive internally, you get volume discounts and economy of scale, even if there are not too many WSCs. On the other hand,

custom hardware can be very expensive, particularly if only small numbers are manufactured. The economies of scale lead to cloud computing, since the lower per-unit costs of WSCs lead to lower rental rates. The flip side of the scale is failures. Even if a server had a Mean Time To Failure (MTTF) of twenty five years, the WSC architect should design for five server failures per day. Similarly, if there were four disks per server and their annual failure rate was 4%, there would be one disk failure per hour, considering 50,000 servers.

**Computer Architecture of WSCs:** Networks are the connecting tissue that binds all the tens of thousands of servers in a WSC. WSCs often use a hierarchy of networks for interconnection.

The most popular switch for a rack is a 48-port Ethernet switch. The bandwidth within the rack is the same for each server.

As far as the storage is concerned, the simplest solution is to use disks inside the servers and use the Ethernet connectivity for accessing information on the disks of remote servers. The other option is to use Network Attached Storage (NAS) through a storage network like Infiniband. This is more expensive, but supports multiple features including RAID for improving dependability. WSCs generally rely on local disks and provide storage software that handle connectivity and dependability. For example, the Google File System (GFS) uses local disks and maintains at least three replicas to provide dependability.

- Servers are typically placed in 19-inch (48.3-cm) *racks*
- Servers are measured in the number of *rack units* (U) that they occupy in the ra
  - One U is 1.75 inches (4.45 cm) high
- A typical rack offers between 42 and 48 U

21 Cloud Computing | **Warehouse-Scale Computing** | Academic year 2013/2014

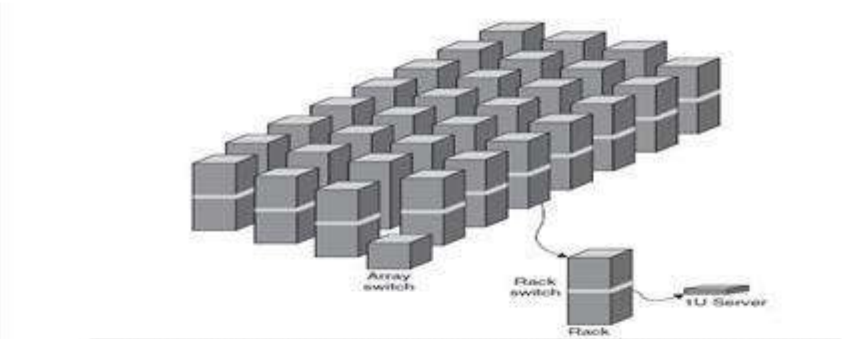


Figure 6.5 Hierarchy of switches in a WSC. (Based on Figure 1.2 of Barroso and Hölzle [2009].)

**Measuring Efficiency of a WSC:** A widely used measure to evaluate the efficiency of a datacenter or WSC is the Power Utilization Effectiveness (PEU).

PUE = Total facility power / IT equipment power

## UNIT V MEMORY AND I/O SYSTEMS

Memory Hierarchy - memory technologies – cache memory – measuring and improving cache performance – virtual memory, TLBs – Accessing I/O Devices – Interrupts – Direct Memory Access – Bus structure – Bus operation – Arbitration – Interface circuits - USB.

## MEMORY HIERARCHY

### BASIC CONCEPTS OF MEMORY SYSTEMS:

- Programs and data that operate in a system are stored in the memory of computer.
- The execution speed of the program is highly dependent on the speed with which instructions and data can be transferred between the processor and the memory.
- The large memory helps to execute programs that are large and deal with huge amount of data.
- For good system, the memory should be fast, large and inexpensive. But it is difficult to meet all these requirements. **Because increased speed and size are achieved at increased cost.**
- By improving the apparent speed and size of the memory, will help to keep the cost reasonable.
- The apparent speed can be increased by using **cache memory** and the apparent size can be achieved by **Virtual memory**.

**The speed of the memory unit is measured in terms of**

- ✓ **Memory access time**
- ✓ **Memory cycle time**

**Memory access time:**It is the time that elapses between the initiation of an operation and the completion of that operation.

**Memory cycle time:** It is the minimum time delay required between two successive memory operations. Example the time between two successive read operation

### Memory hierarchy

Memory hierarchy is a structure that uses multiple levels of memories, as the distance from the processor increases, the size of the memories and the access time both increase. A memory hierarchy consists of multiple levels of memory with different speeds and sizes. The faster memories are more expensive per bit than the slower memories and thus are smaller..

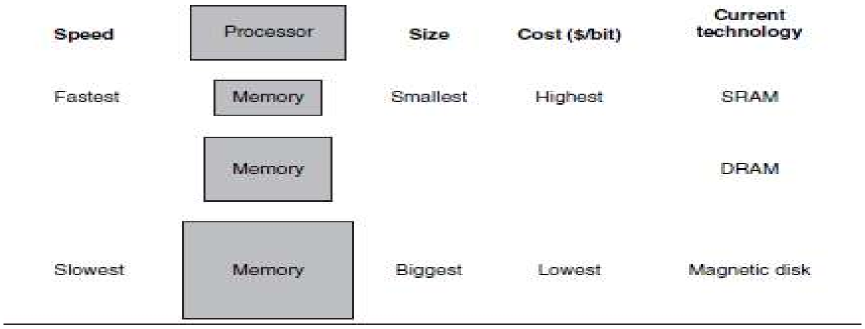


**Fig:1 The basic structure of memory hierarchy**

- **The principle of locality** states that programs access a relatively small portion of their address space at any instant of time

There are two different types of locality

- Temporal locality
- Spatial locality

**Temporal locality(locality in time):** The tendency to reuse recently accessed data items.

**Spatial locality(locality in space):** The tendency to reference data items that are close to other recently accessed items.

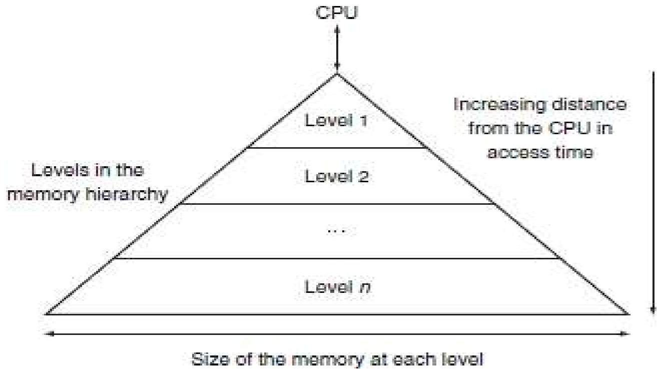
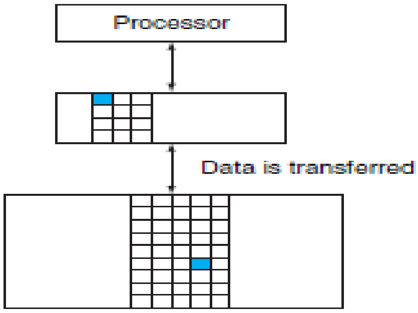

CPU
Level 1
Level 2
...
Level n
Levels in the memory hierarchy
Increasing distance from the CPU in access time
Size of the memory at each level

**Fig.2 This diagram shows the structure of a memory hierarchy: as the distance from the processor increases, so does the size.**

- The above fig shows that a memory hierarchy uses smaller and faster memory technologies close to the processor. Thus, accesses that hit in the highest level of the hierarchy can be processed quickly. Accesses that miss go to lower levels of the hierarchy, which are larger but slower.

**Cache memory:**It 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.



Upper level

Lower level

**Fig 3: The pair of levels in the memory hierarchy**

Upper level memories are closer to the processor and it is smaller and faster than the lower level memory because the technology used in the upper level is more expensive.

**Block:** The **minimum unit of information** that can be either present or not present in the two-level hierarchy.
**Hit**: If the **data requested by the processor appears** in some block in the upper level,this is called a *hit*..
**Miss**: If the **data is not found in the upper level**, the request is called a *miss*.

The **hit rate or *hit ratio***, is the fraction of memory accesses found in the upper level; it is often used as a measure of the performance of the memory hierarchy.
**Miss rate:** The fraction of memory accesses not found in a level of the memory hierarchy.
**Hit time:** The time required to access a level of the memory hierarchy, including the time needed to determine whether the access is a hit or a miss.
**Miss penalty:** is the time to replace a block in the upper level with the corresponding block from the lower level, plus the time to deliver this block to the processor.

## MEMORY TECHNOLOGIES

There are four primary technologies used today in memory hierarchies.

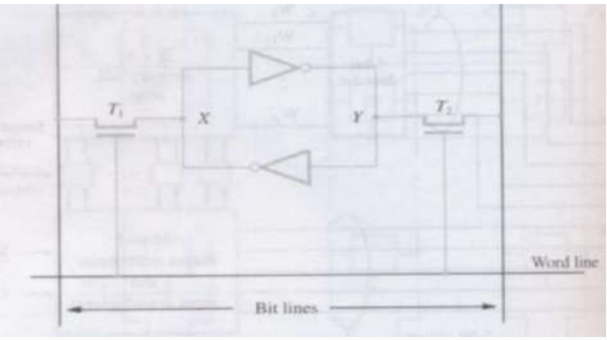
- Main memory is implemented from **DRAM** (dynamic random access memory),
- levels closer to the processor (caches) use **SRAM** (static random access memory).
- The third technology is **Flash memory**.
- The fourth technology, used to implementthe largest and slowest level in the hierarchy in servers, is **Magnetic disk**.

### a) SRAM Technology:

SRAM is based on F/F cell as shown in fig and array of cells is implemented as a chip.

- Here two inverters are cross coupled to form Flip Flop
- The cells are connected to bit line by switches T1 & T2
- The switches are activated by word line.
- When the word line is 1, the switches are closed and F/F output are connected to bit line.

STATIC RAM (SRAM) CELL

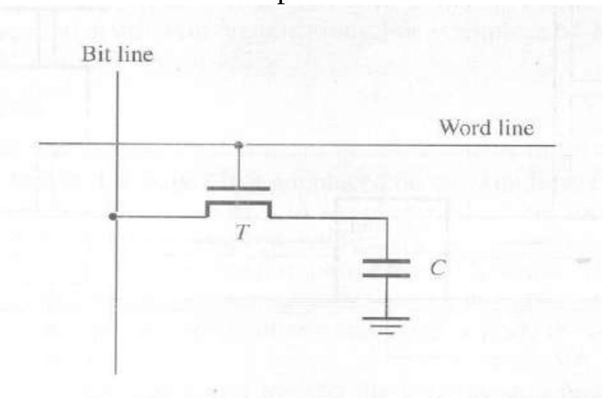


- Here, as long as power is applied, the content is kept indefinitely. Hence it is called Static RAM.
- SRAM have a fixed access time to any data.
- No need to refresh.
- Uses six to eight transistors per bit.

- In the past, most PC and server systems uses separate SRAMS chips for caches. Today caches are integrated on to the processor chip so the market for separate SRAM chips has reduced.

## b) DYNAMIC RAM

- Static Memory is fast, but they are costly since the cell require more transistors and hence more semi conductor area.
- Hence simple RAM cell is implemented to reduce the cost.
- Sample DRAM cell consists of a capacitor and a Transistor shown below.



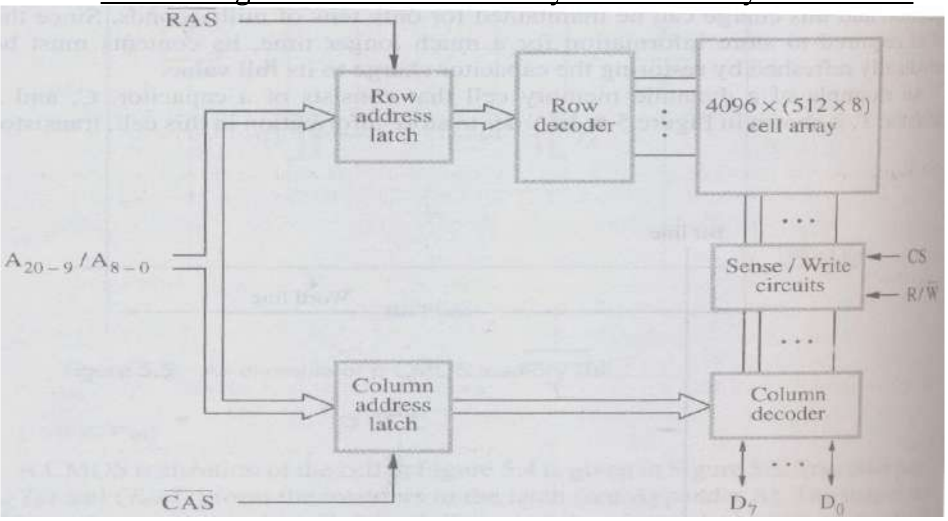
**Dynamic RAM (DRAM)**

- Information is stored in the form of charge in a capacitor.
- Since the charge can be maintained for only Tens of millisecond, the contents must be periodically refreshed by restoring the charge in the capacitor to keep the information for a much longer time.
- Hence they are called Dynamic RAM

**<u>ASYNCHRONOUS DYNAMIC RAM</u>**

- The initial DRAM was asynchronous type.
- Here the processor must take into account the delay in the response of the memory. Such memories are referred as ASYNCHRONOUS DRAM

<u>Internal organization of a 2M x 8 dynamic memory cell</u>



- The cells are organized in the form of a **4K x 4K array**. The **4096 cells** in each row are divided into **512 groups of 8**, so that a row can store 512 bytes of data. 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.

- **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 is selected.
- If the R/Wcontrol signal indicates a Read operation, the output values of the selected circuits are transferred to the data lines, D7-D0.
- For a Write operation, the information on the D7-D0 lines is transferred to the selected circuits.
- 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. **Here, the dynamic nature of these memory chips is almost invisible to the user.**

A specialized memory controller circuit provides the necessary control signals, RAS and 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.

Advantage of DRAMs :

- **High density and low cost**, DRAMs are widely used in the memory units of computers.
- 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.

## FAST PAGE MODE:

When the DRAM in above Figure is accessed, the contents of all 4096 cells in the **selected row are sensed**, but only 8 bits are placed on the data lines D7-D0. This byte is selected by the column address bits A8-A0.

A simple modification can make it possible to **access the other bytes in the same row without having to reselect the row**. A latch can be added at the output of the sense amplifier in each column. **The application of a row address will load the latches corresponding to all bits in the selected row.** Then, it is only necessary to **apply different column addresses** to place the different bytes on the data lines.

## Advantage of Fast page mode:

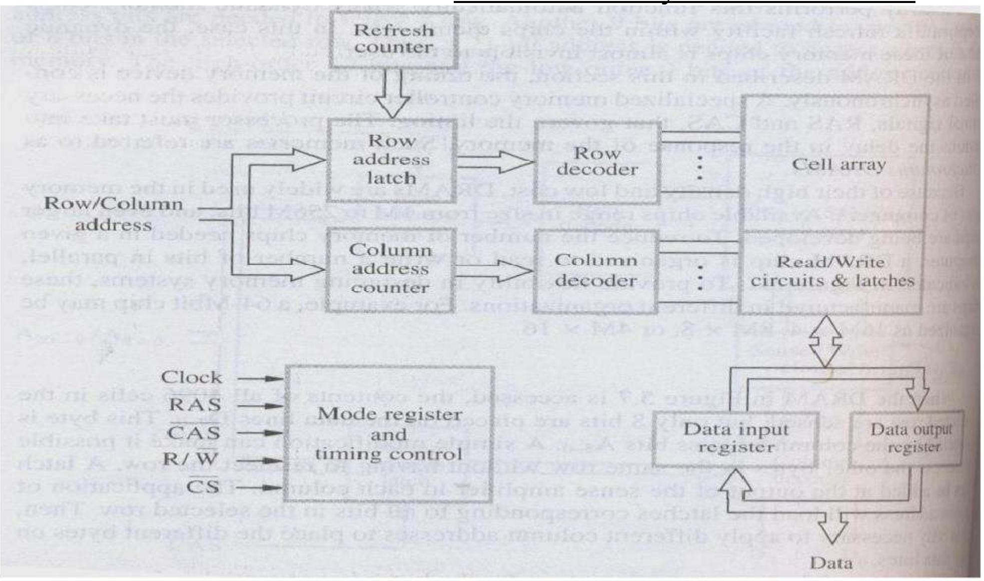
The faster rate attainable in block transfers can be exploited in applications in which memory accesses follow regular patterns, such as in graphics terminals.

This feature is also beneficial in general-purpose computers for transferring data blocks between the main memory and a cache.

### 2.Synchronous DRAM

**DRAMs operation is directly synchronized with a clock signal. Such memories are known as synchronous DRAMs (SDRAMs).**

Structure of a Synchronous DRAM



- The cell array is the same as in asynchronous DRAMs. The address and data connections are buffered by means of registers.
- 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.
- All actions are triggered by the rising edge of the clock. The burst operation is used for the block transfer of data as fast page mode feature. It is specified by setting the mode register.
- First, the row address is latched under control of the RAS signal. The memory typically takes 2 or 3 clock cycles 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.**

### Double – data rate SDRAM (DDR SDRAM)

**The standard SDRAM performs all actions on the rising edge of the clock signal**.The SDRAM,which accesses the cell array for transfers of data on both edges of the clock signal is called **double – data rate SDRAMs**. The latency of these devices is same as for standard SDRAMs. But, they transfer data on both edges of the clock. Such devices is called **double – data- rate SDRAMs**. Their bandwidth is doubled for long burst transfers.

## Application of DDR SDRAM and Standard SDRAMs:

Mainly used in the application where block transfers are established. Some examples are,
Used in general purpose computers in which transfer is mainly between main memory and processor caches.Used in high - quality video displays.

## FLASH MEMORY

Flash memory is a type of EPROM. Similar to EPROM, a flash cell is based on a single transistor controlled by trapped charge.

## Advantages:

Flash memory devices have greater density than EPROM.
Higher capacity and lower cost per bit.
They require a single power supply voltage.
Consumes less power in their operation.
More suitable for portable equipments that are battery driven like hand-held computers, cell phones, digital cameras and MP3 music players.

## Disadvantages:

It writes can wear out flash memory bits. Flash controllers spread the writes by remapping blocks that have been written many times to less used blocks. This techniques is called wear Leveling.

## Disk Memory:

A magenetic hard disk consists of a collection of platters, which rotate on a spindle at 5400 to 15000 revolutions per minutes.
Each disk surface is divided into concentric circles called Tracks. There are typically tens of thousands of tracks per surface. Each track is in turn dividd into sector that contain the information. Each track may have thousands of sectors.

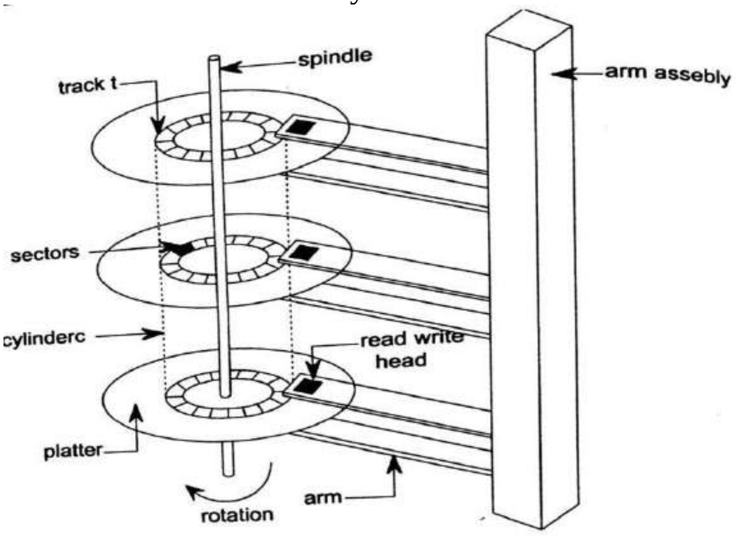


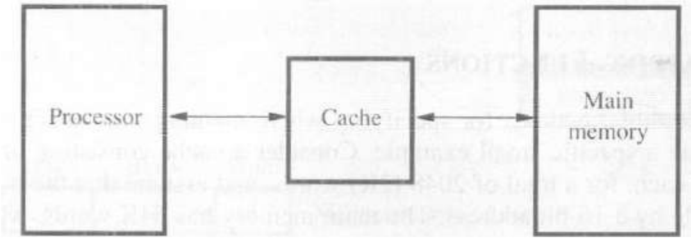
*Figure 6.7 Structure of platters in magnetic disk memory*

To access data, the operating system must direct the disk through a three stage process.

1. The first step is to position the head over the proper track. The operation is called a seek and the time to move the head to the desired track is called the seek time.
2. Once the head has reached the correct track, we must wait for the desired sector to rotate under the read/write head. This time is called the Rotational Latency/ Delay.
3. The last component of a disk access, Transfer Time, is the time to transfer a block of bits. The transfer time is a function of the sector size, the rotation speed, and the recording density of a track.

# CACHE :

A small, fast memory introduced between the processor and the main memory is called cache memory which improves the effective speed of the memory.



## Use of a Cache Memory

### I) <u>BLOCK IDENTIFICATION (How to find the Cache block):</u>

- Caches have an **address tag on each block frame that gives the block address**. The most common procedure is to add a ***valid bit*** **t**o the tag to say whether or not this entry **contains a valid addres**s. If the bit is not set, there cannot be a match on this address.
- Figure C.3 shows how an address is divided. The first division is between the **block address and the block offset.** The block frame address can be further divided into the tag field and the index field. The block offset field selects the desired data from the block, the index field selects the set, and the tag field is compared against it for a hit.

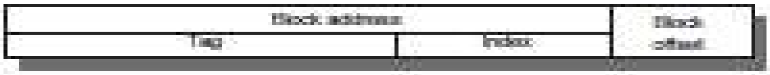


**Figure C.3** The three portions of an address in a set-associative or direct-mapped cache. The tag is used to check all the blocks in the set, and the index is used to select the set. The block offset is the address of the desired data within the block. Fully associative caches have no index field.

### II) <u>CACHE REPLACEMENT STRATEGIES:</u>

There are three primary strategies employed for selecting which block to replace:

***<u>Random</u>* :**To spread allocation uniformly, candidate blocks are randomly selected.

**<u>Least-recently used(LRU):</u>** Relying on the past to predict the future, the block replaced is the one that has been unused for the longest time. If recently used blocks are likely to be used again, then a good candidate for disposal is the least-recently used block.

**<u>First in, first out (FIFO)</u>**<u>:</u>Because LRU can be complicated to calculate, this approximates LRU by determining the *oldest* block rather than the LRU.

### III) <u>WHAT HAPPENS ON A WRITE? OR WRITE STRATEGY</u>

**Write through**—The information is written to both the block in the cache *and* to the block in the lower-level memory.

**Write back**—The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.

### IV) <u>BLOCK PLACEMENT OR MAPPING FUNCTIONS</u>:

**Explain Mapping function in cache memory to determine how memory blocks are placed in Cache.**

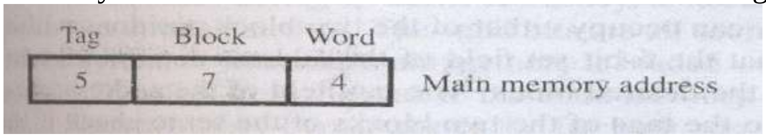
**Mapping functions are used as a way to decide which main memory block occupies which line of cache.**

Let us discuss the possible methods for specifying where memory blocks are placed in the cache. Consider a cache consisting of 128 blocks of 16 words for a total of 2048 (2K) words and Assume that the main memory is 64K words, which we will view as 4K blocks of 16 words each.

For ease of understanding, assume that consecutive addresses refer to consecutive words.

## Direct Mapping

- **The simplest way to determine cache locations in which to store memory blocks is the direct-mapping technique.**
- Both main memory and cache memory is divided into blocks of size equal to 16 words.
- In this technique, (block j of the main memory) mod (Number of cache blocks in the cache)
    - Here block j of main memory maps to (block j mod 128) of cache.
    - Main memory blocks 0, 128, 256 … loaded into cache block 0
    - Main memory blocks 1, 129, 257 … loaded into cache block 1 and so on
- Placement of a block in the cache is determined from memory address.
    - Memory address is divided into **three fields** as shown in figure



- - **Lower order 4 bit** select one of 16 words within a block.
    - Next **7 bits** select cache block (determines the cache position in which the memory block must be stored. )
    - Next **5 bits** decides block from which segment is mapped to cache.
    - During processing higher order 5 bits are compared with tag bits of cache block pointed by 7 bit cache block field
- If match – desired word is in the block – **Cache Hit**
- If not – desired word is brought from main memory and loaded to cache – **Cache Miss**

### *Advantages:*

It is easy to implement.

### *Disadvantages:*

It is not very flexible. Contention problem may arise.

- 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 aprogram 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.

## ASSOCIATIVE MAPPING

**Associative mapping is much more flexible mapping method, in which a main memory block can be placed into any cache block position.**

### Associative mapped cache

- Here 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.

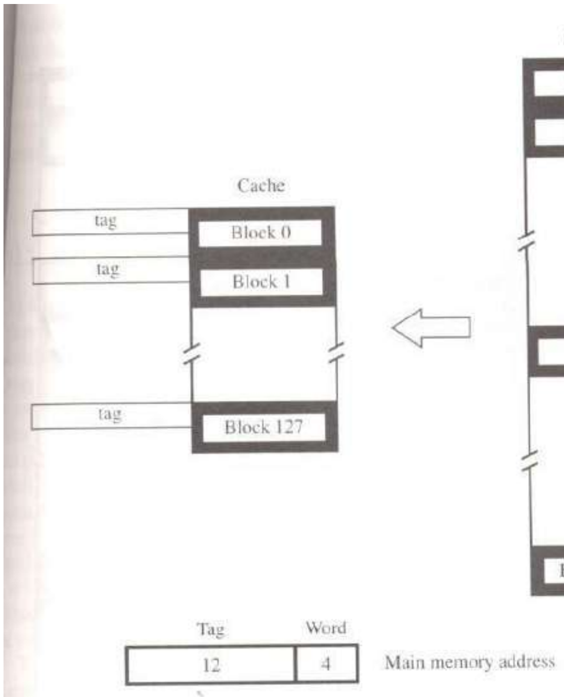
- **A new block that has to be brought into the cache has to replace an existing block only if the cache is full. In this case, we need an algorithm to select the block to be replaced. (By using replacement algorithm – Random, LRU, FIFO)**
- The cost of an associative cache is higher than the cost of 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 better performance, the tags must be searched in parallel.

**Merit:**

It is more flexible than direct mapping technique.

## Demerit:

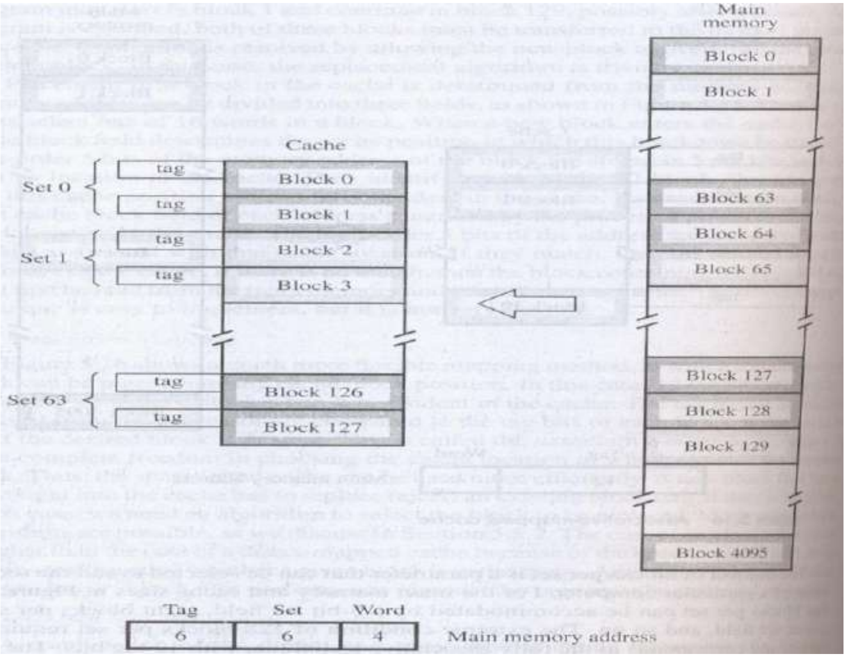
- Its cost is high.



## 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 below for a cache with two blocks per set.
- Block j of main block is mapped into (j mod 64) of cache block
  memory blocks 0, 64, 128 … mapped to cache set 0 and they can occupy either of two blocks position within the set
  memory blocks 1, 65, 129 … mapped to cache set 1 and so on

**Set-associative-mapped cache with two blocks per set**



- During operation 6 bit set field give the set which contain the data and 6 bit Tag of address is associatively compared to the tags of the two blocks of the set to check the desired data is in cache.
- 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.
- A cache that has K blocks per set is referred as a K way set associative cache.
- 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.

**Measuring and Improving Cache Performance**

CPU time can be divided into the clock cycles that the CPU spends executing the program and the clock cycles that the CPU spends waiting for the memory system. Thus,

$$\text{CPU time} = (\text{CPU execution clock cycles} + \text{Memory-stall clock cycles}) \\ \times \text{Clock cycle time}$$

The memory-stall clock cycles come primarily from cache misses, and we make that assumption here. We also restrict the discussion to a simplified model of the memory system. In real processors, the stalls generated by reads and writes can be quite complex, and accurate performance prediction usually requires very detailed simulations of the processor and memory system.

Memory-stall clock cycles can be defined as the sum of the stall cycles coming from reads plus those coming from writes:

$$\text{Memory-stall clock cycles} = (\text{Read-stall cycles} + \text{Write-stall cycles})$$

The read-stall cycles can be defined in terms of the number of read accesses per program, the miss penalty in clock cycles for a read, and the read miss rate:

$$\text{Read-stall cycles} = \frac{\text{Reads}}{\text{Program}} \times \text{Read miss rate} \times \text{Read miss penalty}$$

For a write-through scheme, we have two sources of stalls: write misses, which usually require that we fetch the block before continuing the write and write buffer stalls, which occur when the write buffer is full when a write occurs. Th us, the cycles stalled for writes equals the sum of these two:

$$\text{Write-stall cycles} = \left(\frac{\text{Writes}}{\text{Program}} \times \text{Write miss rate} \times \text{Write miss penalty}\right) + \text{Write buffer stalls}$$

In most write-through cache organizations, the read and write miss penalties are the same (the time to fetch the block from memory). If we assume that the write buffer stalls are negligible, we can combine the reads and writes by using a single miss rate and the miss penalty:

$$\text{Memory-stall clock cycles} = \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

We can also factor this as

$$\text{Memory-stall clock cycles} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

Average memory access time (AMAT) is the average time to access memory considering both hits and misses and the frequency of different accesses;

$$\text{AMAT} = \text{Time for a hit} + \text{Miss rate} \times \text{Miss penalty}$$

**Advanced Cache Optimizations Technique:**

The average memory access time formula above gives us three metrics for cache optimizations: hit time, miss rate, and miss penalty.

- Reducing the hit time: small and simple caches, way prediction, and trace caches
- Increasing cache bandwidth: pipelined caches, multibanked caches, and nonblocking caches
- Reducing the miss penalty: critical word first and merging write buffers
- Reducing the miss rate: compiler optimizations
- Reducing the miss penalty or miss rate via parallelism: hardware prefetching and compiler prefetching

## Small and Simple Caches to Reduce Hit Time:

- A time consuming portion of a cache hit is using the index portion of the address to read the tag memory and then compare it to the address.
- Smaller hardware can be faster, so a small cache can help the hit time.
- keep the cache simple, such as using direct mapping. One benefit of direct-mapped caches is that the designer can overlap the tag check with the transmission of the data.

## Way Prediction to Reduce Hit Time

In *way prediction,* extra bits are kept in the cache to predict the way, or block within the set of the *next* cache access.

- This prediction means the multiplexor is set early to select the desired block, and only a single tag comparison is performed that clock cycle in parallel with reading the cache data.
- Way prediction is a good match to speculative processors, since they must already undo actions when speculation is unsuccessful

## Trace Caches to Reduce Hit Time

- To provide instruction-level parallelism find enough instructions every cycle without use dependencies.
- To address this challenge, blocks in a *trace cache* contain dynamic traces of the executed instructions rather than static sequences of instructions.
- Trace caches have much more complicated address-mapping mechanisms.

## Pipelined Cache Access to Increase Cache Bandwidth

- This optimization is simply to pipeline cache access so that the effective latency of a first-level cache hit can be multiple clock cycles, giving fast clock cycle time and high bandwidth but slow hits.

## Nonblocking Caches to Increase Cache Bandwidth

- For pipelined computers that allow out-of-order completion, the processor need not stall on a data cache miss.
- The processor could continue fetching instructions from the instruction cache while waiting for the data cache to return the missing data.
- A *nonblocking cache* or *lockup-free cache* escalates the potential benefits of such a scheme by allowing the data cache to continue to supply cache hits during a miss.
- This "hit under miss" optimization reduces the effective miss penalty by being helpful during a miss instead of ignoring the requests of the processor.

## Multibanked Caches to Increase CacheBandwidth

- Rather than treat the cache as a single monolithic block, we can divide it into independent banks that can support simultaneous accesses.

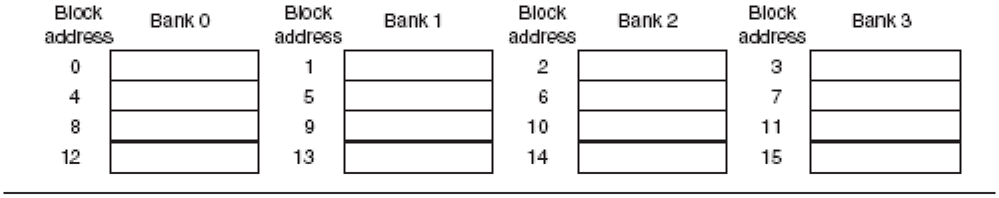


**Figure 5.6 Four-way interleaved cache banks using block addressing.** Assuming 64 bytes per blocks, each of these addresses would be multiplied by 64 to get byte addressing.

- Banking works best when the accesses naturally spread themselves across the banks, so the mapping of addresses to banks affects the behavior of the memory system.
- A simple mapping that works well is to spread the addresses of the block sequentially across the banks, called *sequential interleaving*.
- For example, if there are four banks, bank 0 has all blocks whose address modulo 4 is 0; bank 1 has all blocks whose address modulo 4 is 1; and so on. Figure 5.6 shows this interleaving.

## Critical Word First and Early Restart to Reduce Miss Penalty

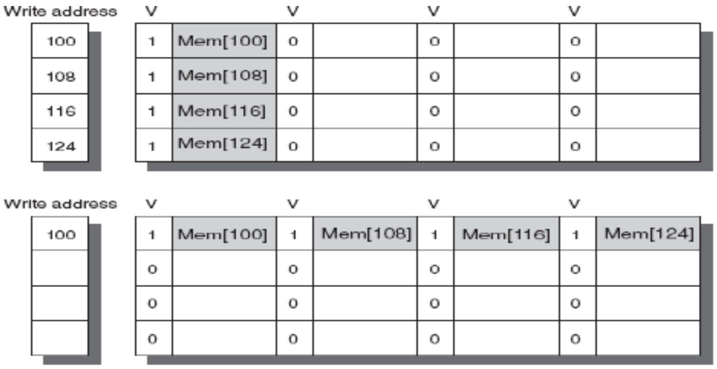
- This strategy is impatience: Don't wait for the full block to be loaded before sending the requested word and restarting the processor. Here are two specific strategies:

  ***Critical word first***—Request the missed word first from memory and send it to the processor as soon as it arrives; let the processor continue execution while filling the rest of the words in the block.

  ***Early restart***—Fetch the words in normal order, but as soon as the requested word of the block arrives, send it to the processor and let the processor continue execution.
- These techniques only benefit designs with large cache blocks, since the benefit is low unless blocks are large.

## Merging Write Buffer to Reduce Miss Penalty

- write-back caches use a simple buffer when a block is replaced. If the write buffer is empty, the data and the full address are written in the buffer, and the write is finished from the processor's perspective; the processor continues working while the write buffer prepares to write the word to memory.
- If the buffer contains other modified blocks, the addresses can be checked to see if the address of this new data matches the address of a valid write buffer entry. If so, the new data are combined with that entry. *Write merging* is the name of this optimization.
- This optimization uses the memory more efficiently since multiword writes are usually faster than writes performed one word at a time.
- In a write-back cache, the block that is replaced is sometimes called the *victim*.Hence, the AMD Opteron calls its write buffer a *victim buffer*. The write victim buffer or victim buffer contains the dirty blocks that are discarded from a cache because of a miss. Rather than stall on a subsequent cache miss, the contents of the buffer are checked on a miss to see if they have the desired data before going to the next lower-level memory.
- It sounds like another optimization called a *victim cache*. In contrast, the victim cache can include any blocks discarded from the cache on a miss, whether they are dirty or not.



| Write address | V | | V | | V | | V | |
|---|---|---|---|---|---|---|---|---|
| 100 | 1 | Mem[100] | 0 | | 0 | | 0 | |
| 108 | 1 | Mem[108] | 0 | | 0 | | 0 | |
| 116 | 1 | Mem[116] | 0 | | 0 | | 0 | |
| 124 | 1 | Mem[124] | 0 | | 0 | | 0 | |

| Write address | V | | V | | V | | V | |
|---|---|---|---|---|---|---|---|---|
| 100 | 1 | Mem[100] | 1 | Mem[108] | 1 | Mem[116] | 1 | Mem[124] |
| | 0 | | 0 | | 0 | | 0 | |
| | 0 | | 0 | | 0 | | 0 | |
| | 0 | | 0 | | 0 | | 0 | |

**Figure 5.7 To illustrate write merging, the write buffer on top does not use it while the write buffer on the bottom does.** The four writes are merged into a single buffer entry with write merging; without it, the buffer is full even though three-fourths of each entry is wasted. The buffer has four entries, and each entry holds four 64-bit words. The address for each entry is on the left, with a valid bit (V) indicating whether the next sequential 8 bytes in this entry are occupied. (Without write merging, the words to the right in the upper part of the figure would only be used for instructions that wrote multiple words at the same time.)

## Compiler Optimizations to Reduce Miss Rate

### *Code and Data Rearrangement*

- Code can easily be rearranged without affecting correctness; for example, reordering the procedures of a program might reduce instruction miss rates by reducing conflict misses.
- Reoodering the instruction reduced misses by 50% for a 2kb direct mapped instruction cache with 4byte blocks.75% in an 8Kb cache.
- Aligning basic blocks so that the entry point is at the beginning of a cache block decreases the chance of a cache miss for sequential code.
- Data have even fewer restrictions on location than code. The goal of such transformations is to try to improve the spatial and temporal locality of the data.

### *Loop Interchange*

- Some programs have nested loops that access data in memory in non sequential order. Simply exchanging the nesting of the loops can make the code access the data in the order they are stored. Assuming the arrays do not fit in the cache, this technique reduces misses by improving spatial locality; reordering maximizes use of data in a cache block before they are discarded.

```
        /* Before */
         for (j = 0; j < 100; j = j+1)
              for (i = 0; i < 5000; i = i+1)
                  x[i][j] = 2 * x[i][j];
                   /* After */
                  for (i = 0; i < 5000; i = i+1)
                     for (j = 0; j < 100; j = j+1)
                            x[i][j] = 2 * x[i][j];
```

### *Blocking*

- This optimization tries to reduce cache misses via improved temporal locality.
- Consider multiple arrays,with some array accessed by rows and some by columns.
- Storing the array row by row or column by column does not solve the problem because both the rows and columns are used in every iteration by the loop.Such orthogonal access meanse that transformation such as loop interchange are not helpful.
- Instead of operating on entire rows or columns of an array, blocked algorithms operate on submatrices or *blocks*. The goal is to maximize accesses to the data loaded into the cache before the data are replaced. The code example below, which performs matrix multiplication, helps motivate the optimization:

```
/* Before */
                for (i = 0; i < N; i = i+1)
                for (j = 0; j < N; j = j+1)
                {r = 0;
                for (k = 0; k < N; k = k + 1)
                r = r + y[i][k]*z[k][j];
                          x[i][j] = r; };
```

- The two inner loops read all N-by-N elements of z, read the same N elements in a row of y repeatedly, and write one row of N elements of x.

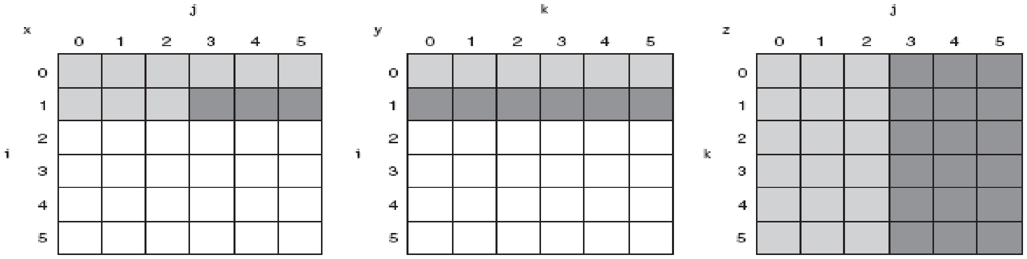


**Figure 5.8** A snapshot of the three arrays x, y, and z when $N$ = 6 and i = 1. The age of accesses to the array elements is indicated by shade: white means not yet touched, light means older accesses, and dark means newer accesses. Compared to Figure 5.9, elements of y and z are read repeatedly to calculate new elements of x. The variables i, j, and k are shown along the rows or columns used to access the arrays.

- The number of capacity misses clearly depends on N and the size of the cache.
- To ensure that the elements being accessed can fit in the cache, the original code is changed to compute on a submatrix of size B by B.
- Two inner loops now compute in steps of size B rather than the full length of x and z. B is called the ***blocking factor***. (Assume x is initialized to zero.)

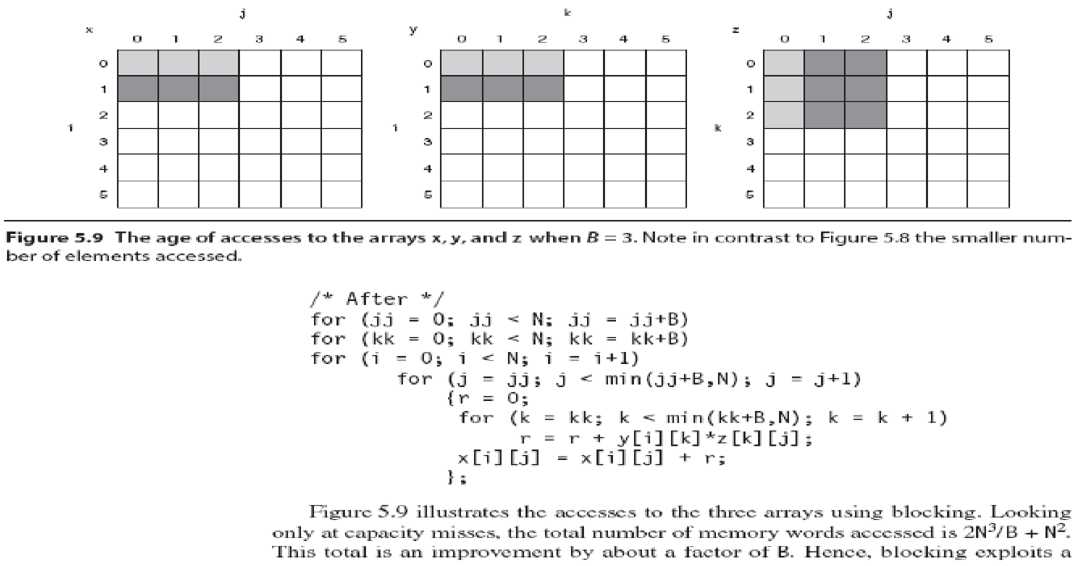

x
j
0 1 2 3 4 5
0 1 2 3 4 5
i
y
k
0 1 2 3 4 5
0 1 2 3 4 5
i
z
j
0 1 2 3 4 5
0 1 2 3 4 5
k

**Figure 5.9** The age of accesses to the arrays x, y, and z when $B = 3$. Note in contrast to Figure 5.8 the smaller number of elements accessed.

```
/* After */
for (jj = 0; jj < N; jj = jj+B)
for (kk = 0; kk < N; kk = kk+B)
for (i = 0; i < N; i = i+1)
      for (j = jj; j < min(jj+B,N); j = j+1)
          {r = 0;
           for (k = kk; k < min(kk+B,N); k = k + 1)
                r = r + y[i][k]*z[k][j];
           x[i][j] = x[i][j] + r;
          };
```

Figure 5.9 illustrates the accesses to the three arrays using blocking. Looking only at capacity misses, the total number of memory words accessed is $2N^3/B + N^2$. This total is an improvement by about a factor of B. Hence, blocking exploits a

## Hardware Prefetching of Instructions and Data to Reduce Miss Penalty or MissRate

- Instruction prefetch is frequently done in hardware outside of the cache Both instructions and data can be prefetched, either directly into the caches or into an external buffer that can be more quickly accessed than main memory.
- Typically, the processor fetches two blocks on a miss: the requested block and the next consecutive block. The requested block is placed in the instruction cache when it returns, and the prefetched block is placed into the instruction stream buffer.
- If the requested block is present in the instruction stream buffer, the original cache request is canceled, the block is read from the stream buffer, and the next prefetch request is issued.
- Prefetching relies on utilizing memory bandwidth that otherwise would be unused, but if it interferes with demand misses, it can actually lower performance.

## Compiler-Controlled Prefetching to Reduce Miss Penalty or Miss Rate

- An alternative to hardware prefetching is for the compiler to insert prefetch instructions to request data before the processor needs it. There are two flavors of prefetch:

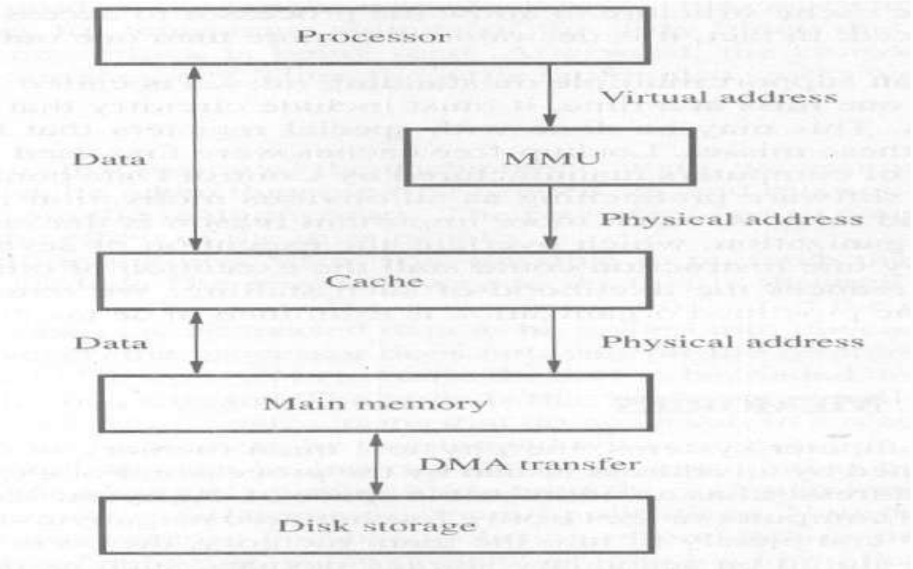
1. *Register prefetch* will load the value into a register.
2. *Cache prefetch* loads data only into the cache and not the register.

Either of these can be *faulting* or *nonfaulting;*“faulting register prefetch instruction.” Nonfaulting prefetches simply turn into no-ops if they would normally result in an exception, which is what we want.

## Virtual Memory (What is virtual memory? Explain the steps involved in virtual memory address translation.)

The main memory act as a “cache” for the secondary storage, usually implemented with magnetic disks. This technique is called **virtual memory.**

A virtual memory block is called a *page*, and a virtual memory miss is called a **page fault**. With virtual memory, the processor produces a **virtual address**, which is translated by a combination of hardware and software to a *physical address*, which in turn can be used to access main memory. Figure 5.25 shows the virtually addressed memory with pages mapped to main memory. This process is **address mapping or address translation**.



In virtual memory, the address is broken into a *virtual page number* and a *pageoff set*. Figure 5.26 shows the translation of the virtual page number to a *physical page number*. The physical page number constitutes the upper portion of the physical address, while the page off set, which is not changed, constitutes the lower portion.

The number of bits in the page off set field determines the page size. The number of pages addressable with the virtual address need not match the number of pages addressable with the physical address.

In virtual memory systems, we locate pages by using atable that indexes the memory; this structure is called a **page table**, and it resides in memory. A page table is indexed with the page number from the virtual address to discover the corresponding physical page number. Each program has its own page table, which maps the virtual address space of that program to main memory.

**To indicate the location of the page table in memory, the hardware includes a register that points to the start of the page table; we call this the *page table register*.**

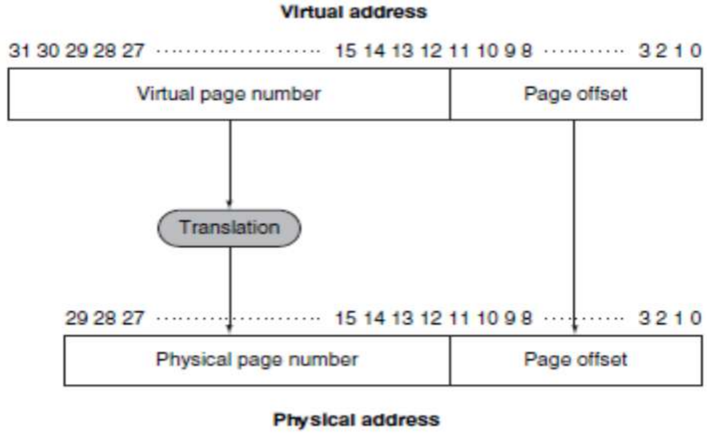


**FIGURE 5.26 Mapping from a virtual to a physical address.** The page size is $2^{12}$ = 4 KiB. The number of physical pages allowed in memory is $2^{18}$, since the physical page number has 18 bits in it. Thus, main memory can have at most 1 GiB, while the virtual address space is 4 GiB.

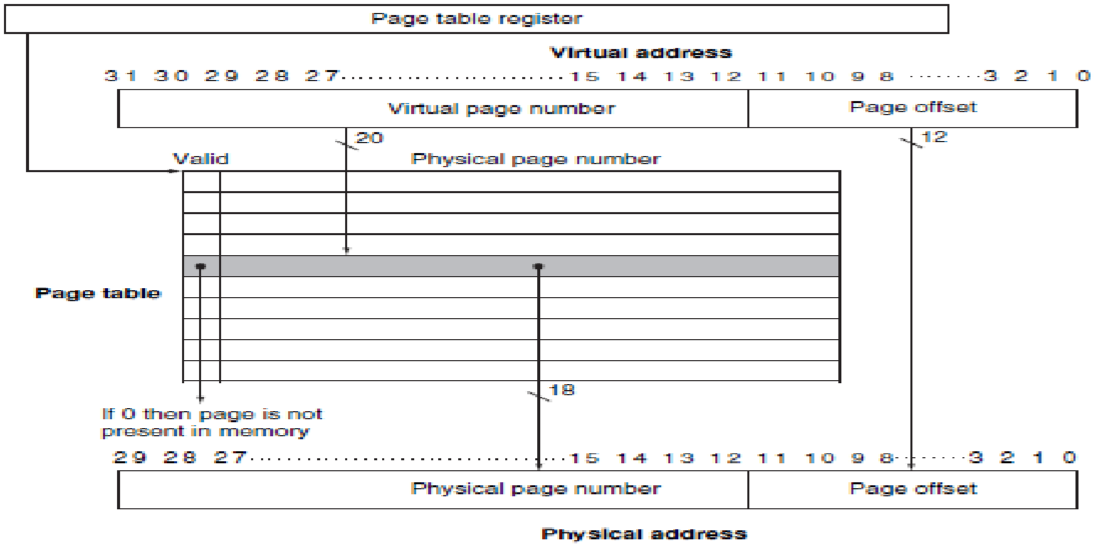


**FIGURE 5.27 The page table is indexed with the virtual page number to obtain the corresponding portion of the physical address.** We assume a 32-bit address. The page table pointer gives the starting address of the page table. In this figure, the page size is $2^{12}$ bytes, or 4 KiB. The virtual address space is $2^{32}$ bytes, or 4 GiB, and the physical address space is $2^{30}$ bytes, which allows main memory of up to 1 GiB. The number of entries in the page table is $2^{20}$, or 1 million entries. The valid bit for each entry indicates whether the mapping is legal. If it is off, then the page is not present in memory. Although the page table entry shown here need only be 19 bits wide, it would typically be rounded up to 32 bits for ease of indexing. The extra bits would be used to store additional information that needs to be kept on a per-page basis, such as protection.

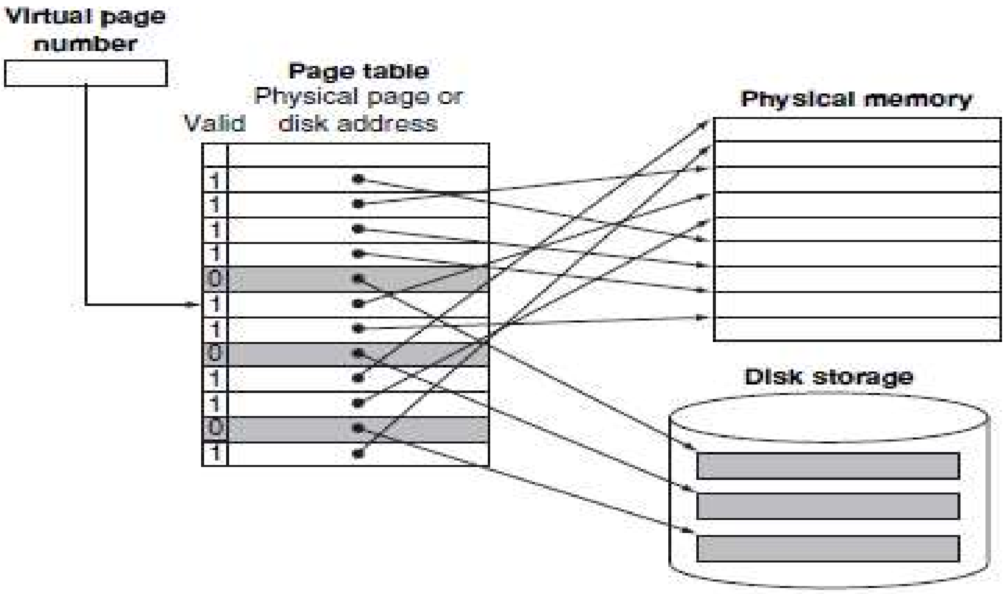


**FIGURE 5.28 The page table maps each page in virtual memory to either a page in main memory or a page stored on disk, which is the next level in the hierarchy.** The virtual page number is used to index the page table. If the valid bit is on, the page table supplies the physical page number (i.e., the starting address of the page in memory) corresponding to the virtual page. If the valid bit is off, the page currently resides only on disk, at a specified disk address. In many systems, the table of physical page addresses and disk page addresses, while logically one table, is stored in two separate data structures. Dual tables are justified in part because we must keep the disk addresses of all the pages, even if they are currently in main memory. Remember that the pages in main memory and the pages on disk are the same size.

## Making Address Translation Fast: the TLB

The key to improving access performance is to rely on locality of reference to the page table. When a translation for a virtual page number is used, it will probably be needed again in the near future, because the references to the words on that page have both temporal and spatial locality. Accordingly, **modern processors include a special cache that keeps track of recently used translations. This special address translation cache is traditionally referred to as a translation-lookaside buffer (TLB)**

Figure 5.29 shows that each tag entry in the TLB holds a portion of the virtual page number, and each data entry of the TLB holds a physical page number. Because we access the TLB instead of the page table on every reference, the TLB will need to include other status bits, such as the **dirty and the reference bits**.

On every reference, we look up the virtual page number in the TLB.

If we get a hit, **the physical page number is used to form the address**, and the corresponding **reference bit is turned on**. If the processor is **performing a write, the dirty bit is also turned on**.

If a miss in the TLB occurs, we must determine whether it is a **page fault or merely a TLB miss.** If the **page exists in memory**, then the TLB miss indicates only that the translation is missing. In such cases, the processor can handle the TLB miss by loading the translation from the page table into the TLB and then trying to the reference again.

If the **page is not present in memory**, then the TLB miss indicates a true page fault. In this case, the processor invokes the operating system using an exception.

Because the TLB has many fewer entries than the number of pages in main memory,

**TLB misses will be much more frequent than true page faults.**

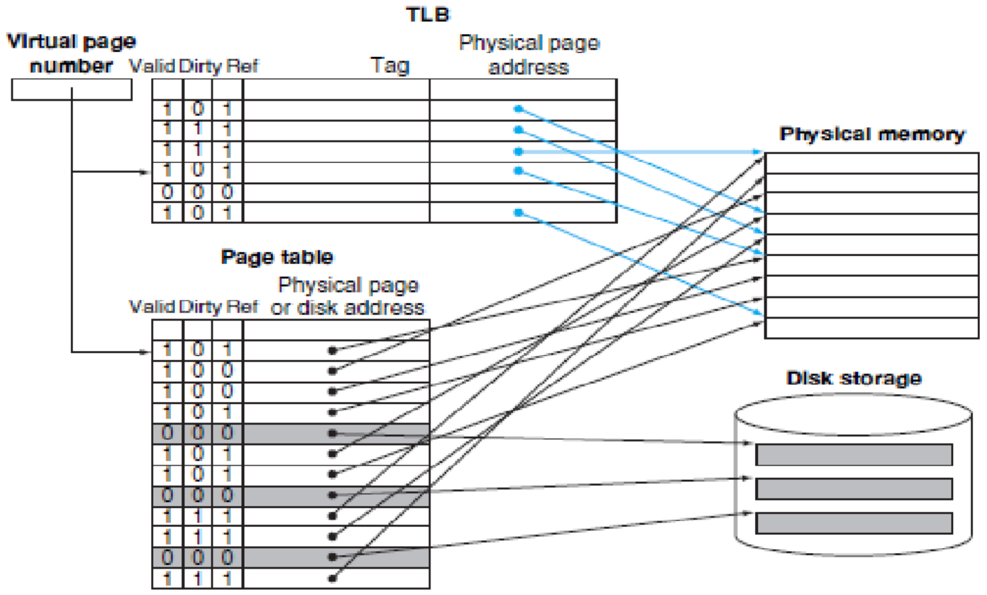


**FIGURE 5.29 The TLB acts as a cache of the page table for the entries that map to physical pages only.** The TLB contains a subset of the virtual-to-physical page mappings that are in the page table. The TLB mappings are shown in color. Because the TLB is a cache, it must have a tag field. If there is no matching entry in the TLB for a page, the page table must be examined. The page table either supplies a physical page number for the page (which can then be used to build a TLB entry) or indicates that the page resides on disk, in which case a page fault occurs. Since the page table has an entry for every virtual page, no tag field is needed; in other words, unlike a TLB, a page table is *not* a cache.

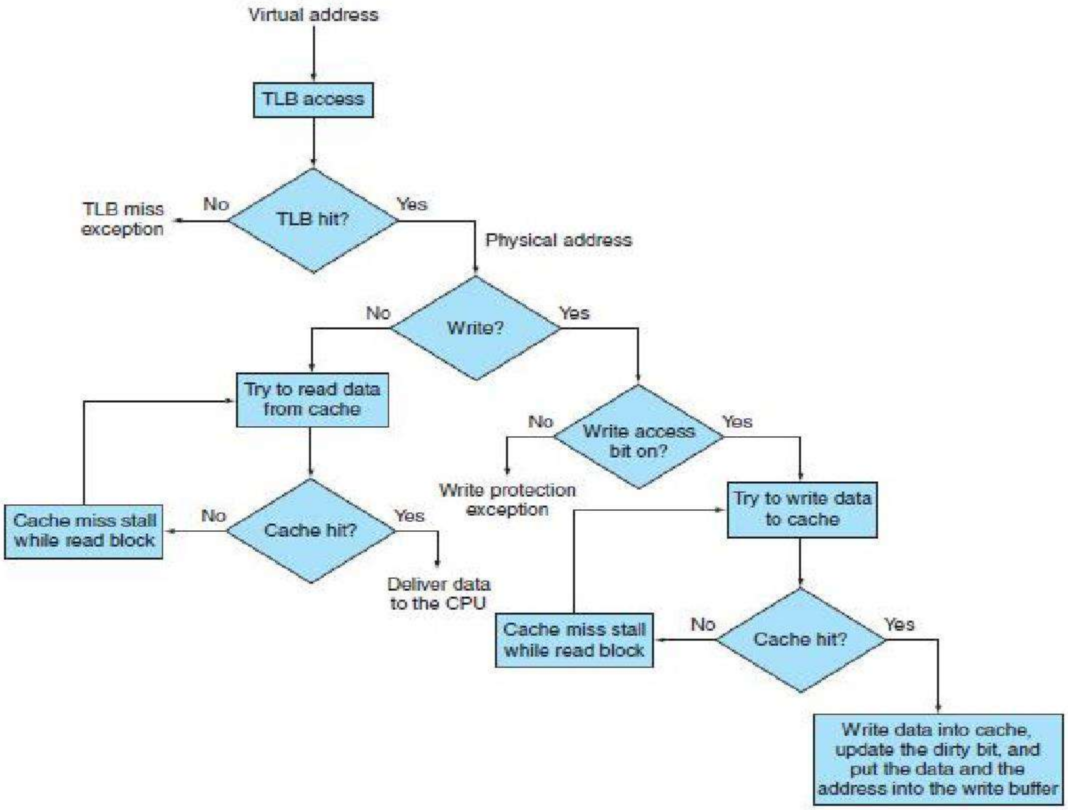
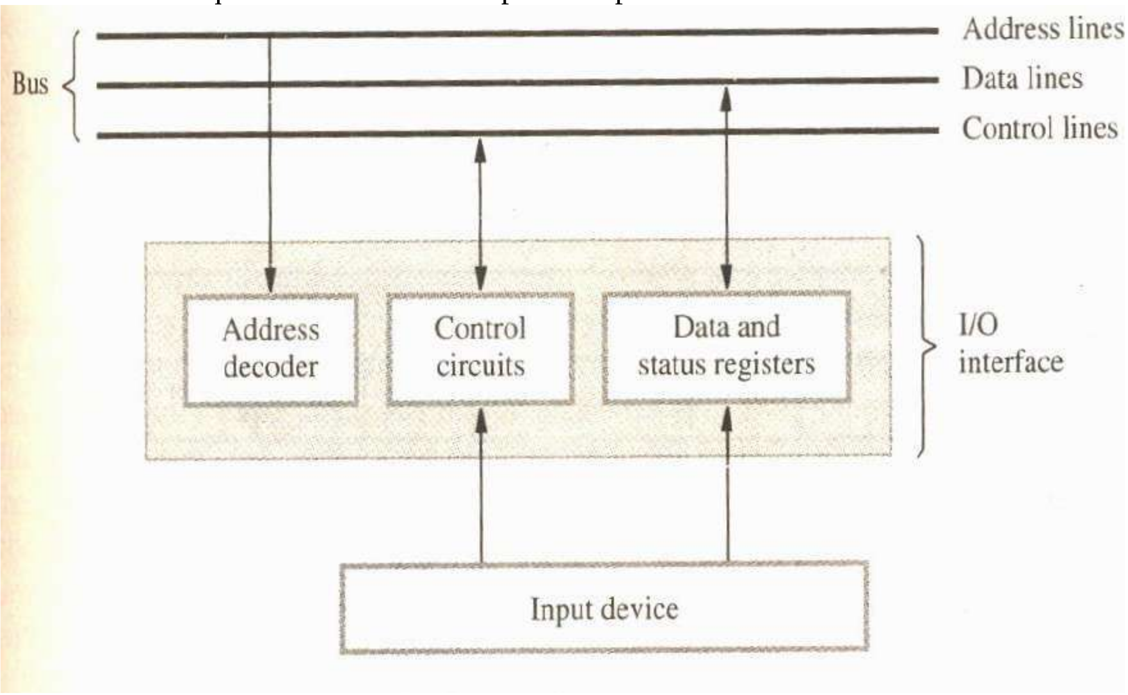

Virtual address
TLB access
TLB hit?
No
TLB miss exception
Yes
Physical address
Write?
No
Yes
Try to read data from cache
Write access bit on?
No
Yes
Write protection exception
Try to write data to cache
Cache miss stall while read block
No
Cache hit?
Yes
Deliver data to the CPU
Cache miss stall while read block
No
Cache hit?
Yes
Write data into cache, update the dirty bit, and put the data and the address into the write buffer

**FIGURE 5.31 Processing a read or a write-through in the Intrinsity FastMATH TLB and cache.** If the TLB generates a hit, the cache can be accessed with the resulting physical address. For a read, the cache generates a hit or miss and supplies the data or causes a stall while the data is brought from memory. If the operation is a write, a portion of the cache entry is overwritten for a hit and the data is sent to the write buffer if we assume write-through. A write miss is just like a read miss except that the block is modified after it is read from memory. Write-back requires writes to set a dirty bit for the cache block, and a write buffer is loaded with the whole block only on a read miss or write miss if the block to be replaced is dirty. Notice that a TLB hit and a cache hit are independent events, but a cache hit can only occur after a TLB hit occurs, which means that the data must be present in memory. The relationship between TLB misses and cache misses is examined further in the following example and the exercises at the end of this chapter.

---

## **ACCESSING I/O DEVICES (I/O INTERFACE AND ITS COMPONENTS)**

The hardware required to connect an Input / Output device to the bus is called I/O interface.



The major components of I/O interface are

i. Address decoder
ii. Control circuit
iii. Data & status register

***Address Decoder:*** Enables the device to recognize its address when this address appears on the address lines.

***Data Register:*** This holds the data being transferred to or appears on the data lines.

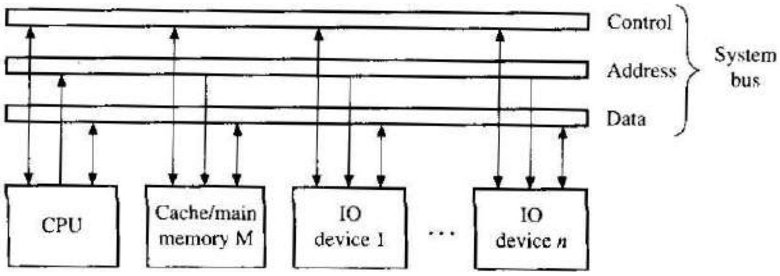
***Status Register:*** Contains information relevant to the operation of the I/O device.

**I/O and memory mapped I/O:**

The components of a computer system CPU, Memory and I/O Units are connected by a group of wires called Bus

The system bus consists of 3 group of lines

a. Address
b. Data and
c. Control



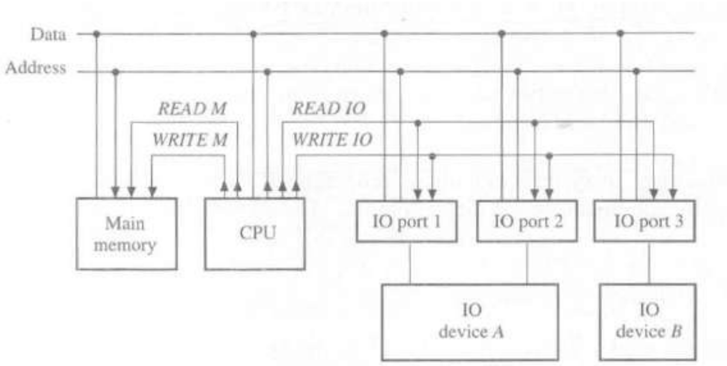
## Methods of I/O

## Addressing There are two methods of I/O addressing

a. **I/O mapped I/O**
b. **Memory mapped I/O**

**a. I/O mapped I/O**

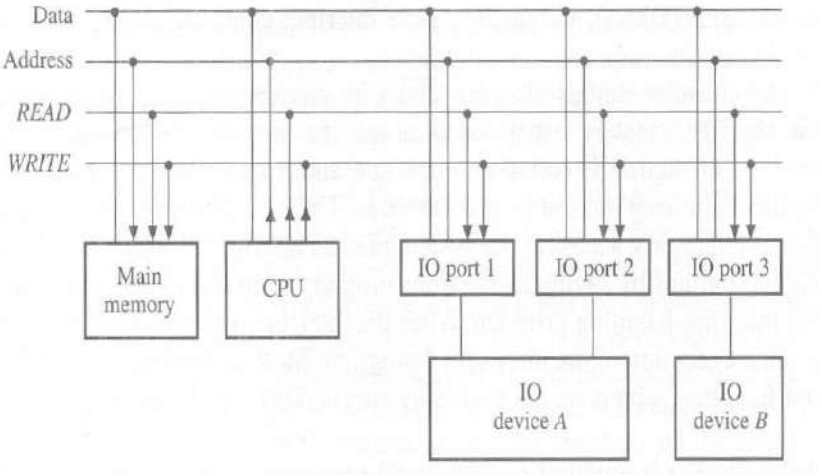
- Here the processor has separate I/O and memory space
- The memory reference instruction activates Read M or Write M control line which will not affect I/O device
- The CPU has separate Input / Output instruction which activates Read I/O or Write I/O
- This method is used in intel 8085



## I/O Mapped I/O

**b. Memory mapped I/O**

- Here there is no separate I/O address and no I/O instructions
- Here some memory address is reserved for I/O register by the designer
- This method is used in Motorola - 6800



## Memory Mapped I/O

Comparison of I/O Mapped I/O and Memory Mapped I/O

| | Memory Mapped I/O | I/O Mapped I/O |
|---|---|---|
| 1. | Part of memory space is used as I/O address Motorola – 6800 | Separate I/O Address space Intel – 8085 |
| 2. | No Separate I/O instruction | Separate I/O Instruction (In / Out) |
| 3. | Extra decoding Logic | No extra Logic |

## INTERRUPTS (Explain Interrupts and Interrupts Hardware.)

➢ There are many situations where other tasks can be performed while waiting for an I/O device to become ready. A hardware signal called an Interrupt will alert the processor when an I/O device becomes ready. Interrupt-request line is usually dedicated for this purpose.

**Example:** Consider a task that requires some computations to be performed and the results to be printed on a line printer. Let the program consist of two routines, COMPUTE and PRINT. Assume that COMPUTE produces a set of n lines of output, to be printed by the PRINT routine.

➢ This example illustrates the concept of interrupts. The routine executed in response to an interrupt request is called the interrupt-service routine, which is the PRINT routine in our example.

➢ Assume that an interrupt request arrives during execution of instruction I in fig 4.5

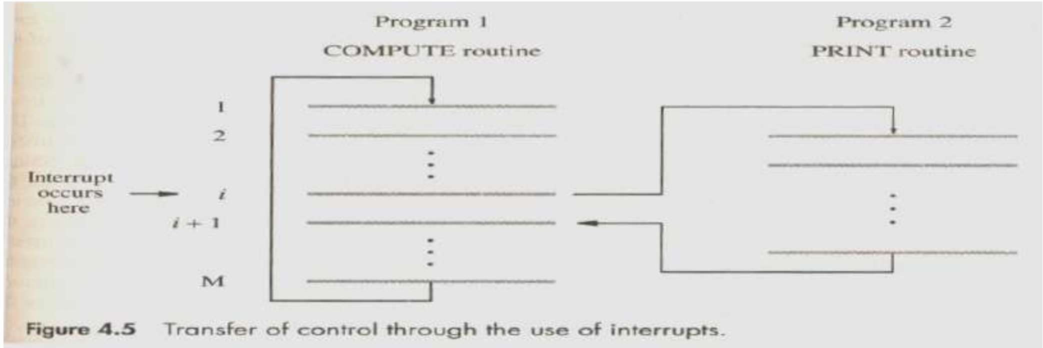


Figure 4.5 Transfer of control through the use of interrupts.

- 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.
- 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 that temporary storage location, causing execution to resume at instruction *i* + 1. In many processors, the return address is saved on the processor stack.
- The execution of an instruction in the interrupt-service routine that accesses a status or data register in the device interface implicitly informs the device that its interrupt request has been recognized.
- 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.
- Saving registers also increases 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***.

## INTERRUPT HARDWARE

An I/O device requests can interrupt by activating a bus line called interrupt-request.

Most computers are likely to have several I/O devices that can request an interrupt.

A single interrupt-request line may be used to serve *n* devices as depicted in Figure 4.6.

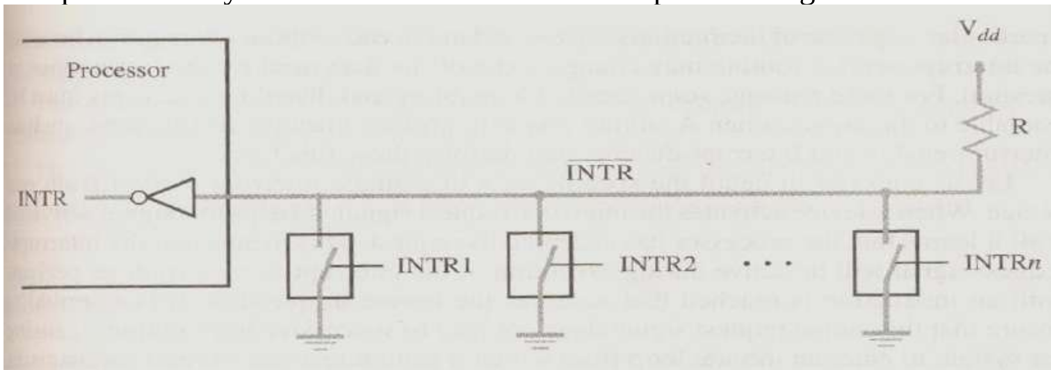


**Figure 4.6** An equivalent circuit for an open-drain bus used to implement a common interrupt-request line.

- All devices are connected to the line via switches to ground. To request an interrupt, a device closes its associated switch. Thus, if all interrupt-request signals INTR1 to INTRn are inactive, that is, if all switches are open, the voltage on the interrupt-request line will be equal to $V_{dd}$. This is the inactive state of the line.
- When a device requests an interrupt by closing its switch, the voltage on the line drops to 0, causing the interrupt-request signal, INTR, received by the processor to go to 1.
- Since the closing of one or more switches will cause the line voltage to drop to 0, the valueof INTR is the logical OR of the requests from individual devices, that is,

$$INTR = INTR1 + ... + INTRn$$

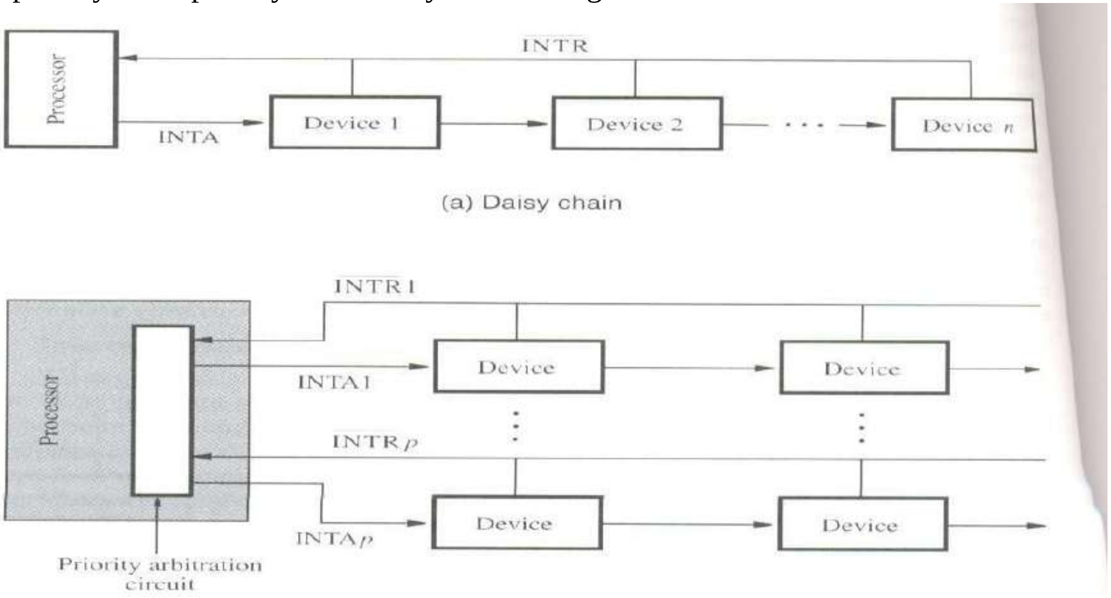
Assuming that interrupts are enabled, the following is a typical scenario:

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 edge-triggered 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.

# HANDLING MULTIPLE DEVICES

**Daisy Chaining:**

. The interrupt request line INTR is common to all the devices. The interrupt acknowledgement line INTA is connected to devices in a DAISY CHAIN way. INTA propagates serially through the devices. Device that is electrically closest to the processor gets high priority. Low priority device may have a danger of STARVATION.



(a) Daisy chain

(b) Arrangement of priority groups

Combining Daisy chaining and Interrupt nesting to form priority group. Each group has different priority levels and within each group devices are connected in daisy chain way.

## *Vectored Interrupts*

- The term ***vectored interrupts*** refers to all interrupt-handling schemes.
- To reduce the time involved in the polling process, a device requesting an interrupt may identify itself directly to the processor.
- 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 bythe 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 interrupt-vector code immediately. For example, it must first  complete the execution of the current instruction, which may require the use of the bus.
- A multiple-priority scheme can be implemented easily by using separate interrupt-request and interrupt-ack lines for each device as shown in fig.4.7

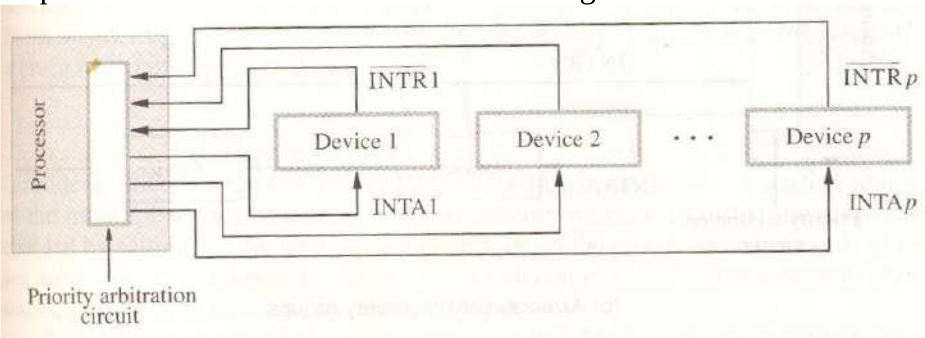

Processor
$\overline{\text{INTR}}1$
$\overline{\text{INTR}}p$
Device 1
Device 2
. . .
Device *p*
INTA1
INTA*p*
Priority arbitration circuit

Figure 4.7 Implementation of interrupt priority using individual interrupt-request and acknowledge lines.

- Each of the interrupt-request lines is assigned a different priority level. Interrupt request receive 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.

## DIRECT MEMORY ACCESS

**A special control unit may be provided to allow transfer of a block of data directly between an external device and the main memory, without continuous intervention by the processor. This approach is called direct memory access or DMA.**

- DMA transfers are performed by a control circuit that is part of the I/O device interface. We refer to this circuit as a ***DMA controller.***

Figure 4.18 shows DMA controller registers that are accessed by the processor to initiate transfer operations.

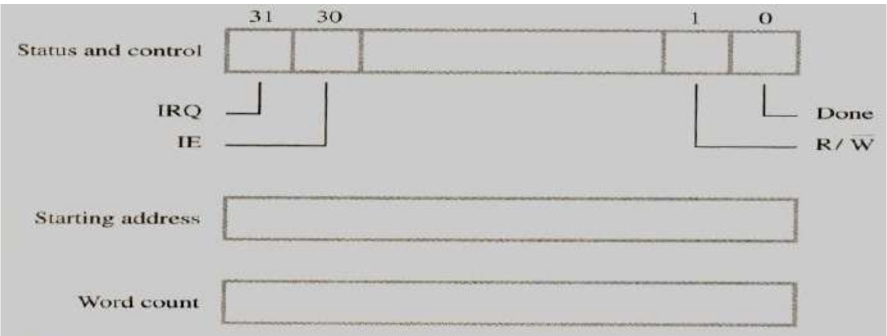


Figure 4.18 Registers in a DMA interface.

- Two registers are used for storing the starting address and the word count. The third register contains status and control flags.
- The R/W bit determines the direction of the transfer. When this bit is set to 1 by a program instruction, the controller performs a read operation, that is, it transfers data

from the memory to the I/O device. Otherwise, it performs a write operation.

- When the controller has completed transferring a block of data and is ready to receive another command, it sets the Done flag to 1.
- Bit 30 is the Interrupt-enable flag, IE. When this flag is set to 1, it causes the controller to raise an interrupt after it has completed transferring a block of data. Finally, the controller sets the IRQ bit to 1 when it has requested an interrupt.

Figure 4.19 showing how DMA controllers may be used.

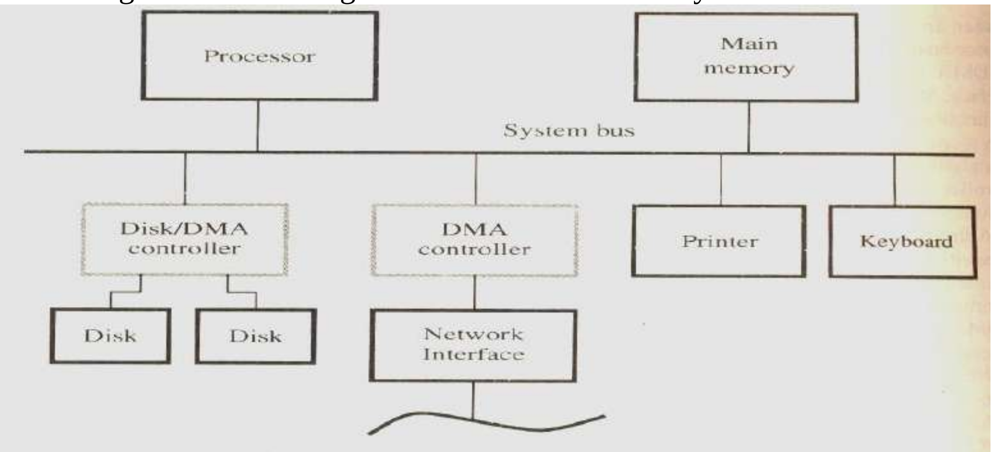
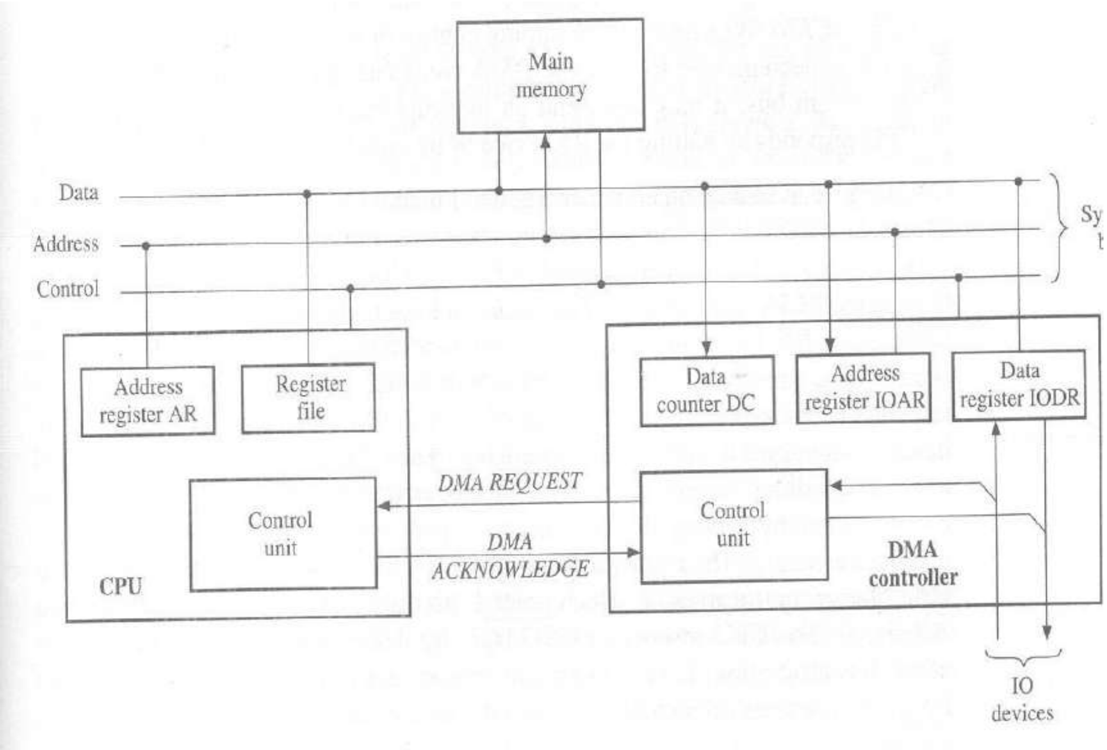


**Figure 4.19** Use of DMA controllers in a computer system.

- A DMA controller connects a high-speed network to the computer bus. The disk controller, which controls two disks, also has DMA capability and provides two DMA channels.
- It can perform two independent DMA operations, as if each disk had its own DMA controller. The registers needed to store the memory address, the word count, and so on are duplicated, so that one set can be used with each device.
- To start a DMA transfer of a block of data from the main memory to one of the disks, a program writes the address and word count information into the registers of the corresponding channel of the disk controller.
- Among different DMA devices, top priority is given to high-speed peripherals such as a disk, a high-speed network interface, or a graphics display device.
- Since the processor originates most memory access cycles, the DMA controller can be said to "steal" memory cycles from the processor. Hence, this interweaving technique is usually called **cycle stealing**. Alternatively, the DMA controller may be given exclusive access to the main memory to transfer a block of data without interruption. This is known as ***block*** **or** ***burst*** **mode.**
- **A conflict may arise if both the processor and a DMA controller or two DMA controllers try to use the bus at the same time to access the main memory.** To resolve these conflicts, an arbitration procedure is implemented on the bus to coordinate the activities of all devices requesting memory transfers.

However in DMA, **the block of data is directly transferred between external device and memory without involving CPU. This type of I/O is called DMA.** The circuit which performs DMA is called DMA controller.



# Circuit required for DMA

**DMA Data Transfer**

1. CPU Load the counter and address Reg with initial values Counter – contain the number of words to be transferred Address Reg – contain the starting address of the block to be transferred
2. When DMA Controller is ready to transmit or receive
   - It activates DMA request to CPU
   - CPU waits for the next DMA Breakpoint and activate DMA acknowledge

   DMA request & DMA acknowledge are essentially bus request and bus grant lines
3. The DMA controller now transfers data directly to or from memory. After each word is transferred Address Register, Word Counter is updated.
4. The Step 3 is repeated until word counter reaches Zero. DMA controller now give back the controls of the system bus it may also send the interrupt signal to CPU. The CPU responds by halting the I/O device or by initiating another DMA transfer.

- Interrupt can take place – between two instruction
- DMA interrupt can take place every bus cycle of the

instruction Types of data Transfers in DMA

a) DMA Block Transfer:

Block of words is transferred in a single burst, while DMA controller is the master of Bus

b) Cycle Stealing:

This allows the DMA controller to use the system bus to transfer one data word and return the control to CPU. Hence long block of I/O data are transferred by a sequence of DMA transactions interspersed with CPU transactions.

## BUS ARBITRATION

- The device that is allowed to initiate data transfers on the bus at any given time is called the ***bus master***.
- ***Bus arbitration*** is the process by which the next device to become the bus master is selected and bus mastership is transferred to it. There are two approaches to bus arbitration:
    - ***Centralized***: In centralized arbitration, a single *bus arbiter* performs the required arbitration
    - ***Distributed:***In distributed arbitration, all devices participate in the selection of the next bus master.

Centralized Arbitration

- The bus arbiter may be the processor or a separate unit connected to the bus.
- Figure 4.20 illustrates a basic arrangement in which the processor contains the bus arbitration circuitry.

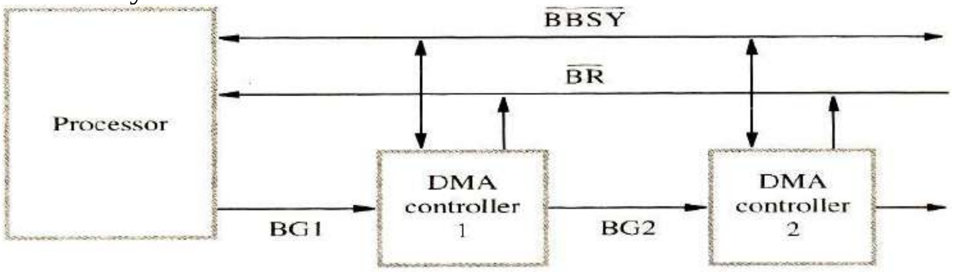


**Figure 4.20** A simple arrangement for bus arbitration using a daisy chain.

- The processor is normally the bus master unless it grants bus mastership to one of the DMA controllers.
- A DMA controller indicates that it needs to become the bus master by activating the Bus- Request line, BR.
- The signal on the Bus-Request line is the logical OR of the bus requests from all the devices connected to it. When Bus-Request is activated, the processor activates the Bus- Grant signal, BG1, indicating to the DMA controllers that they may use the bus when it becomes free. This signal is connected to all DMA controllers using a daisy-chain arrangement.
- Thus, if DMA controller 1 is requesting the bus, it blocks the propagation of the grant signal to other devices. Otherwise, it passes the grant downstream by asserting BG2.
- The current bus master indicates to all devices that it is using the bus by activating another open-collector line called Bus-Busy, BBSY.
- Hence, after receiving the Bus-Grant signal, a DMA controller waits for Bus-Busy to become inactive, and then assumes mastership of the bus. At this time,

it activates Bus- Busy to prevent other devices from using the bus at the same time.

- **Polling Method**

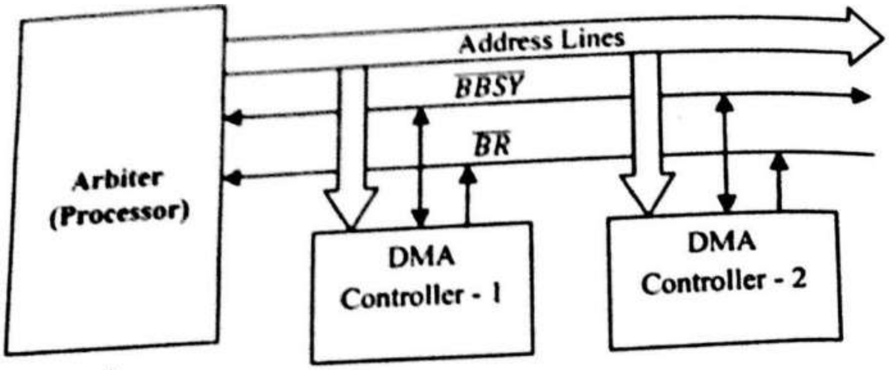


***Figure 7.5 Bus arbitration using polling method***

In Polling Method, all masters use the same line for the bus request.The Controller places the address of the bus master on address bus. The requesting master recognizes its address, activates the busy line and gets the control of the bus.

Advantages: The priority of the masters can be changed by altering the polling sequences.Failure of one device does not affects the others.

- **Independent Priority:**

Each master has a separate pair of bus request and bus grant lines. Each master has different priority . The priority decoder in the controller selects the highest priority master which makes the request ad enables the corresponding grant signal.

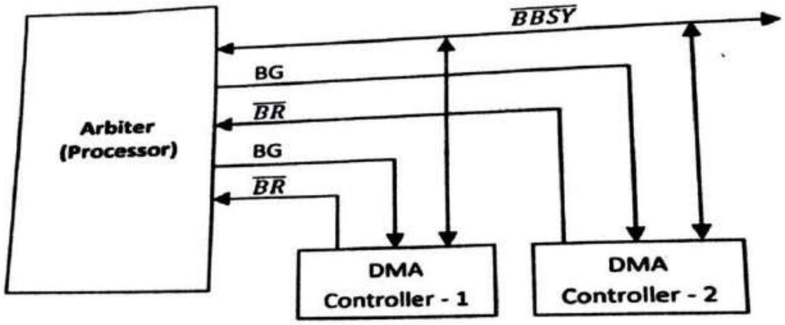


***Figure 7.6 Bus arbitration using independent priority scheme***

## Distributed Arbitration

***Distributed arbitration*** means that all devices waiting to use the bus have equalresponsibility in carrying out the arbitration process, without using a central arbiter

**BUS STRUCTURE**

A ***bus*** is a collection of wires that connect several devices within a computer system. When a word of data is transferred between units, all its bits are transferred in parallel.

A computer must have some lines for addressing and control purposes.
Three main groupings of lines:
1. ***Data Bus***. This is for the transmission of data.
2. ***Address Bus***. This specifies the location of data in MM.
3. ***Control Bus***. This indicates the direction of data transfer and coordinates the timing of events during the transfer.

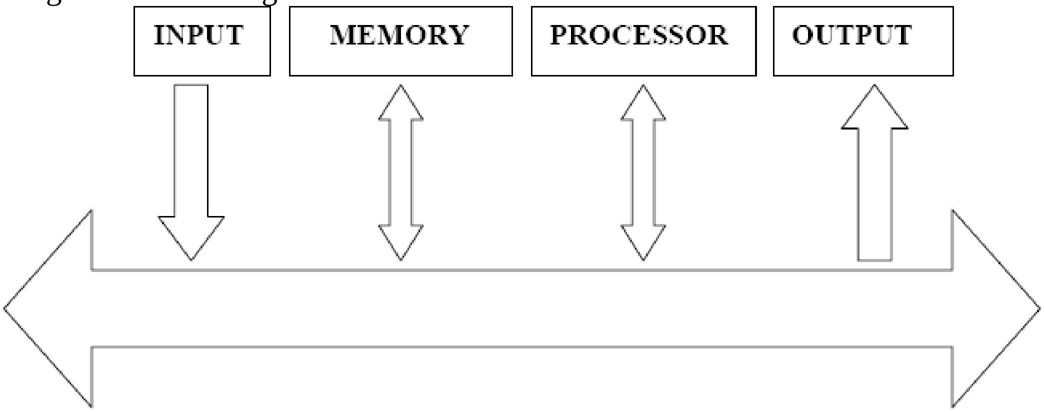


Fig:Single bus Structure

## BUS OPERATION

**Synchronous Bus**: All devices derive timing information from a control line called the *bus clock*. The timing diagram shows an **idealized representation** of the actions that take place on the bus lines.

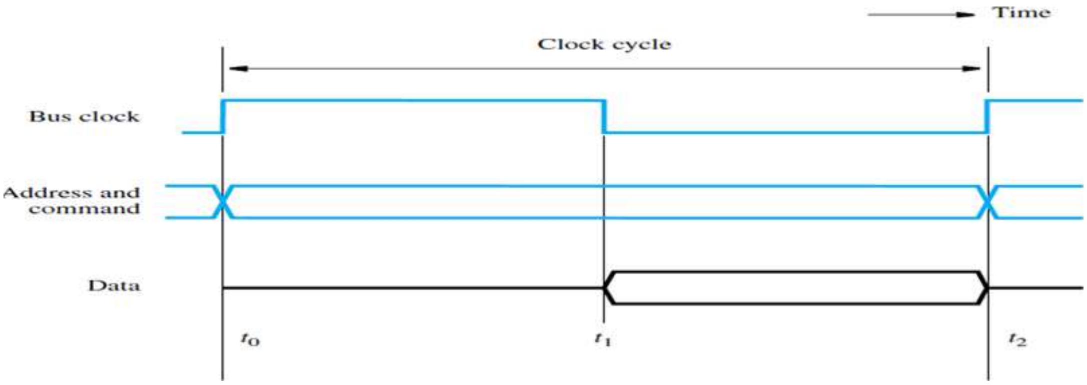


**Figure 5.28: Timing of an input transfer on a synchronous bus**

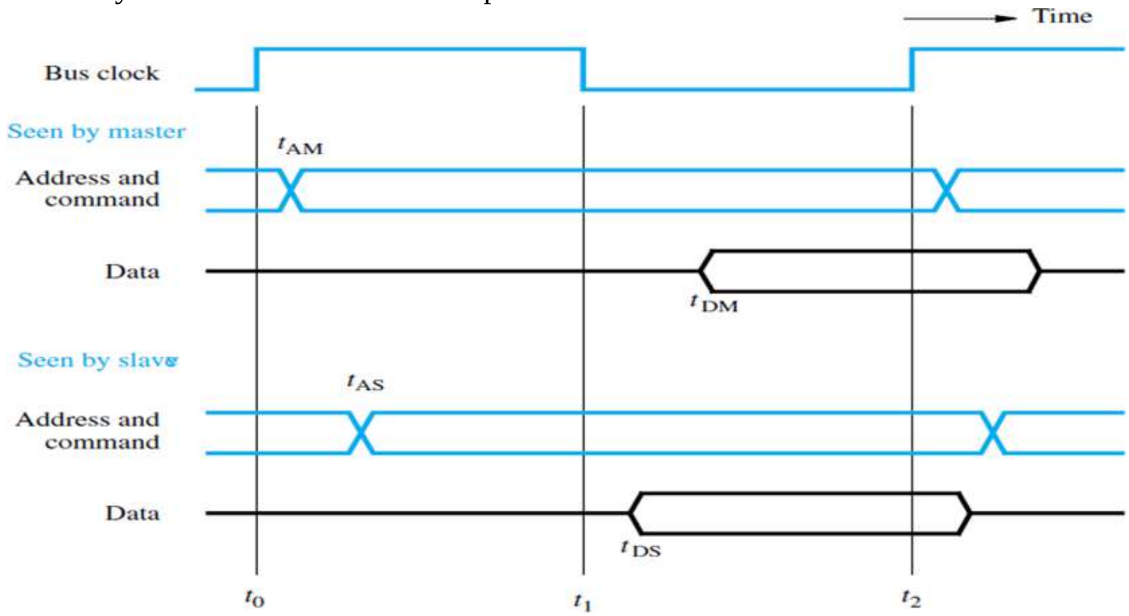
The sequence of signal events during an input (Read) operation.

- At time ***t*0**, the master places the device address on the address lines and sends a command on the control lines indicating a Read operation.
- Information travels over the bus.
- The clock pulse width, **$t1 - t0$**, must be longer than the maximum propagation delay over the bus. Also, it must be long enough to allow all devices to decode the address and control signals, so that the addressed device (the slave) can respond at time ***t*1** by placing the requested input data on the data lines.
- At the end of the clock cycle, at time ***t*2**, the master loads the data on the data lines into one of the setup time of the register. Hence, the period **$t2 - t1$** must be greater than the maximum

propagation time on the bus plus the setup time of the master‘s register.The **exact times at** which signals change state are somewhat different from **idealized representation**, because of

□ propagation delays on bus wires and in the circuits of the devicesThe master sends the address and command signals on the rising edge of the clock at the beginning of the clock cycle (**at *t*0**).

□ However, these signals do not actually appear on the bus until ***t*AM**, largely due to the delay in the electronic circuit output from the master to the bus lines.A short while later, at ***t*AS**, the signals reach the slave. The slave decodes the address, and at ***t*1**sends the requested data.Here again, the data signals do not appear on the bus until ***t*DS**. They travel toward the master and arrive at ***t*DM**. At ***t*2**, the master loads the data into its register.



□ Hence the period ***t*2 − *t*DM** must be greater than the setup time of that register. The data must continue to be valid after ***t*2** for a period equal to the hold time requirement of the register

**Asynchronous Bus**: An alternative scheme for controlling data transfers on a bus is based on the use of a *handshake* protocol between the master and the slave. A handshake is an exchange of command and response signals between the master and the slave.

□ The master places the address and command information on the bus. Then it indicates to all devices that it has done so by activating the Master-ready line.

□ This causes all devices to decode the address. The selected slave performs the required operation and informs the processor that it has done so by activating the Slave-ready line.

□ The master waits for Slave-ready to become asserted before it removes its signals from the bus.

□ In the case of a Read operation, it also loads the data into one of its registers.

□ *t*0—The master places the address and command information on the bus, and all devices on the bus decode this information.

□ *t*1—The master sets the Master-ready line to 1 to inform the devices that the address and command information is ready. Sufficient time should be allowed for the device interface circuitry to decode the address. The delay needed can be included in the period *t*1 − *t*0.

□ *t*2—The selected slave, having decoded the address and command information, performs the required input operation by placing its data on the data lines. At the same time, it sets the Slave- ready signal to 1. If extra delays are introduced by the interface circuitry before it places the data on the bus, the slave must delay the Slave-ready signal accordingly. The period *t*2 − *t*1 depends on the distance between the master and the slave and on the delays introduced by the slave‘s circuitry.

□ *t*3—The Slave-ready signal arrives at the master, indicating that the input data are available on the bus. After a delay to the master loads the data into its register. Then, it drops the Master- ready signal, indicating that it has received the data.

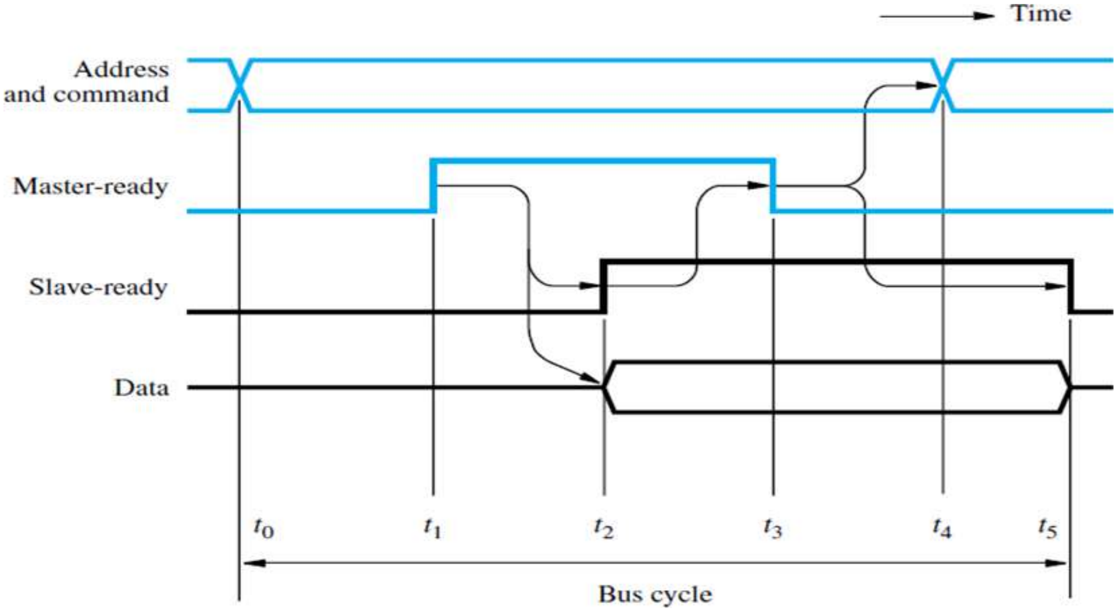


**Figure: Handshake control of data transfer during an input operation** *t*4—The master removes the address and command information from the bus. The delay between *t*3 and *t*4 is again intended to allow for bus skew. Erroneous addressing may take place if the address, as seen by some device on the bus, starts to change while the Master-ready signal is still equal to 1.

□ *t*5—when the device interface receives the 1-to-0 transition of the Master-ready signal, it removes the data and the Slave-ready signal from the bus. This completes the input transfer.

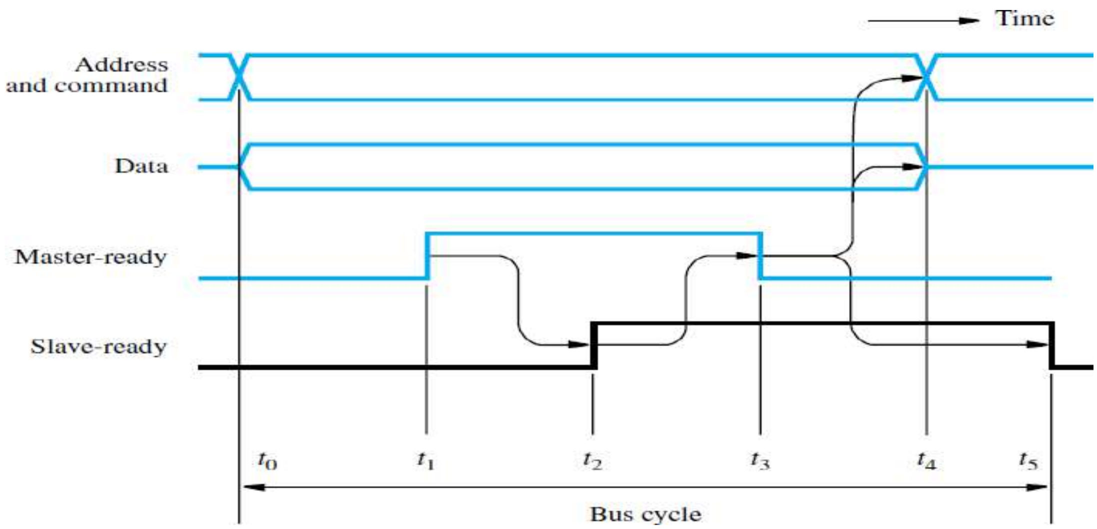


**Figure**: **Handshake control of data transfer during an output**

**INTERFACE CIRCUITS**

Datapath and associated controls to transfer data between the interface and the I/O device. This side is called as a port.

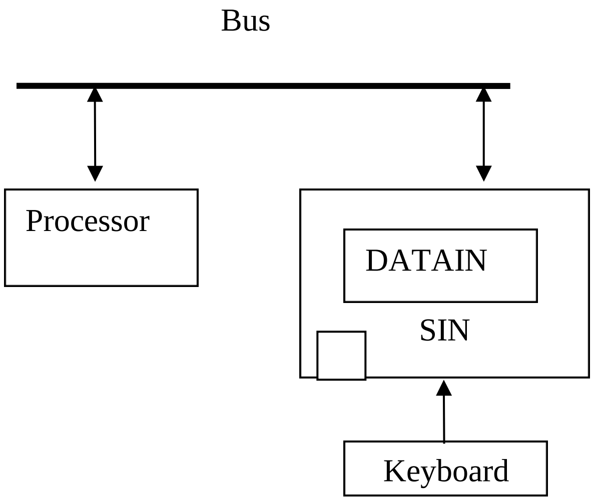
Ports can be

□ classified into

□ two: Parallel port
Serial port.

Parallel port transfers data in the form of a number of bits, normally 8 or 16 to or from the device. Serial port transfers and receives data one bit at a time**.**

## Bus connections between processor and keyboard



- Striking a character stores the corresponding character in Data in (8 bit Buffer Register)
- The availability of character is indicated by a Flag SIN (Status Control Flag)
- The processor checks this flag and if the flag is set, the processor will read the character from data in

```
Readwait:    BR    Readwait if SIN = 0
             IN    Datain, R1
```

- Reading the character from datain to register, will clear sin flag
- This procedure repeats for each character

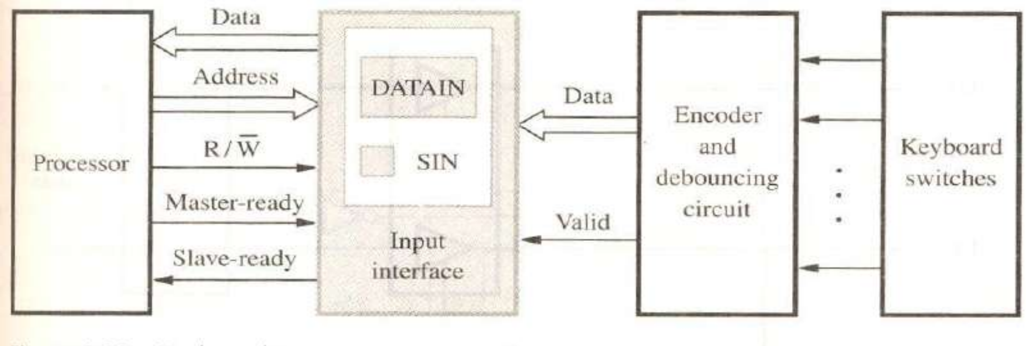


Figure 4.28 Keyboard to processor connection.

- The fig 4.28 shows the hardware components needed for connecting a keyboard to a processor. A typical keyboard consists of mechanical switches that are normally open. When a key is pressed, its switch closes and establishes a path for an electrical signal. This signal is detected by an encoder circuit that generates the ASCII code for the corresponding character. A difficulty with such push-button switches it that the contacts bounce when a key is pressed.

## The effect of bouncing must be eliminated by two ways.

- A simple debouncing circuit can be included or a software approach can be used.
- When debouncing is implemented in software, the I/O routine that reads a character from the keyboard waits long enough to ensure that bouncing has subsided.

Fig 4. 28 Illustrates the hardware approach; debouncing circuits are included as a part of the encoder block.

- The output of the encoder consists of the bits that represent the encoded character and one control signal called valid, which indicates that a key is being pressed. This information is sent to the interface circuit, which contains a data register, DATAIN, and a status flag, SIN.
- When a key is pressed, the valid signal changes from 0 to 1, causing the ASCII code to be loaded into DATAIN and SIN to be set to 1. The status flag SIN is cleared to 0 when the processor reads the contents of the DATAIN register.
- The interface circuit is connected to an asynchronous bus on which transfers are controlled using handshake signals Master-ready and Salve-ready as in the fig 4.26
- The third control line, R/W distinguishes read and write transfers.

Let us now consider an output interface that can be used to connect an output device, such as a printer, to a processor, as shown in Fig 4.31

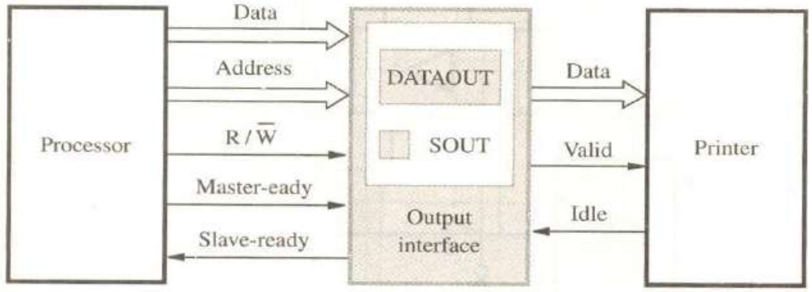


**Figure 4.31** Printer to processor connection.

- The printer operates under control of the handshake signals valid and Idle in a manner similar to the handshake used on the bus with the Master-ready and Slave-ready signals. When it is ready to accept a character, the printer asserts its Idle signal. The interface circuit can then place a new character on the data lines and activate the valid signal.
- In response, the printer starts printing the new character and negates the Idle signal, which in turn causes the interface to deactivate the valid signal.
- The interface contains a data register, DATAOUT, and a status flag, SOUT. The SOUT flag is set to 1 when the printer is ready to accept another character and it is cleared to 0 when a new character is loaded into DATAOUT by the processor.

Serial port

- Serial port is used to connect the processor to I/O devices that require transmission of data one bit at a time.
- Serial port communicates in a bit-serial fashion on the device side and bit parallel fashion on the bus side.
- Transformation between the parallel and serial formats is achieved with shift registers that have parallel access capability.

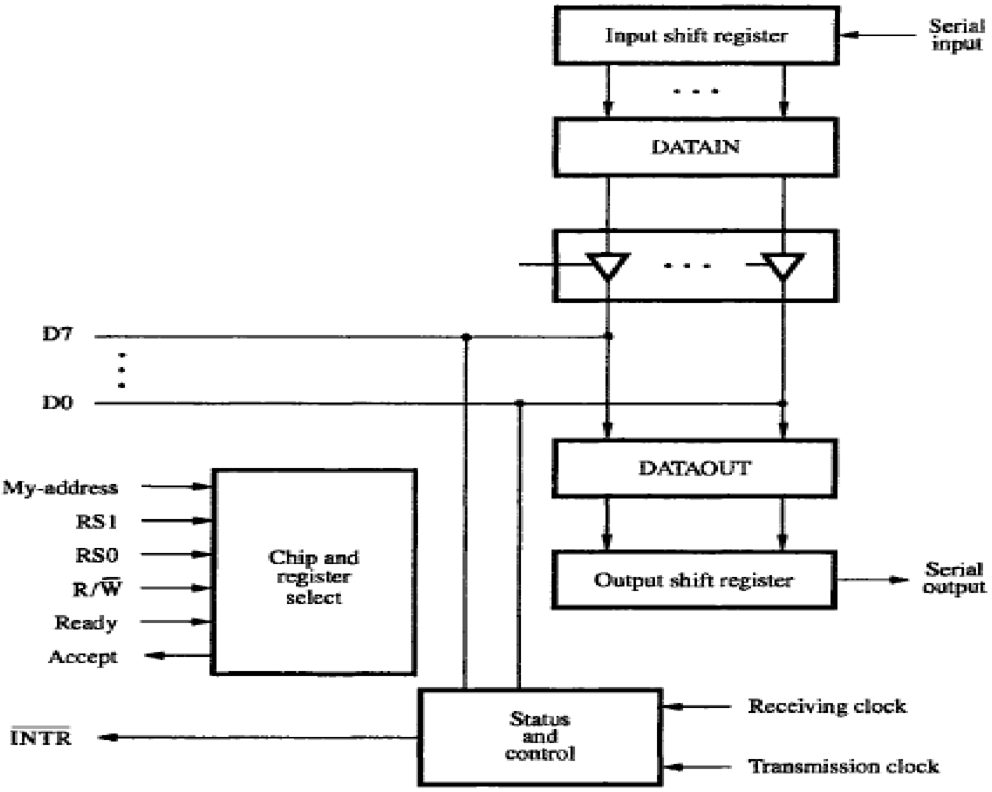


Figure : A serial Interface

Input shift register accepts input one bit at a time from the I/O device.
• Once all the 8 bits are received, the contents of the input shift register are loaded in parallel into DATAIN register.
• Output data in the DATAOUT register are loaded into the output shift register.
Bits are shifted out of the output shift register and sent out to the I/O device one bit at a time.
• As soon as data from the input shift registers are loaded into DATAIN, it can start accepting another 8 bits of data.
• Input shift register and DATAIN registers are both used at input so that the input shift register can start receiving another set of 8 bits from the input device after loading the contents to DATAIN, before the processor reads the contents of DATAIN. This is called as double- buffering.

 Serial interfaces require fewer wires, and hence serial transmission is convenient for connecting devices that are physically distant from the computer.
 Speed of transmission of the data over a serial interface is known as the
 —bit rate. Bit rate depends on the nature of the devices connected.
 In order to accommodate devices with a range of speeds, a serial interface must be able to use a range of clock speeds.
 Several standard serial interfaces have been developed:
 Universal Asynchronous Receiver Transmitter (UART) for low-speed
 serial devices. RS-232-C for connection to communication links

**Input/output system(INTERFACE)**

A different interface may have to be designed for every combination of I/O device and computer, resulting in many different interfaces.

There are three widely used bus standards,

- ✓ PCI(Peripheral Component Interconnect),
- ✓ SCSI(Small Computer System Interface)
- ✓ USB(Universal Serial Bus).

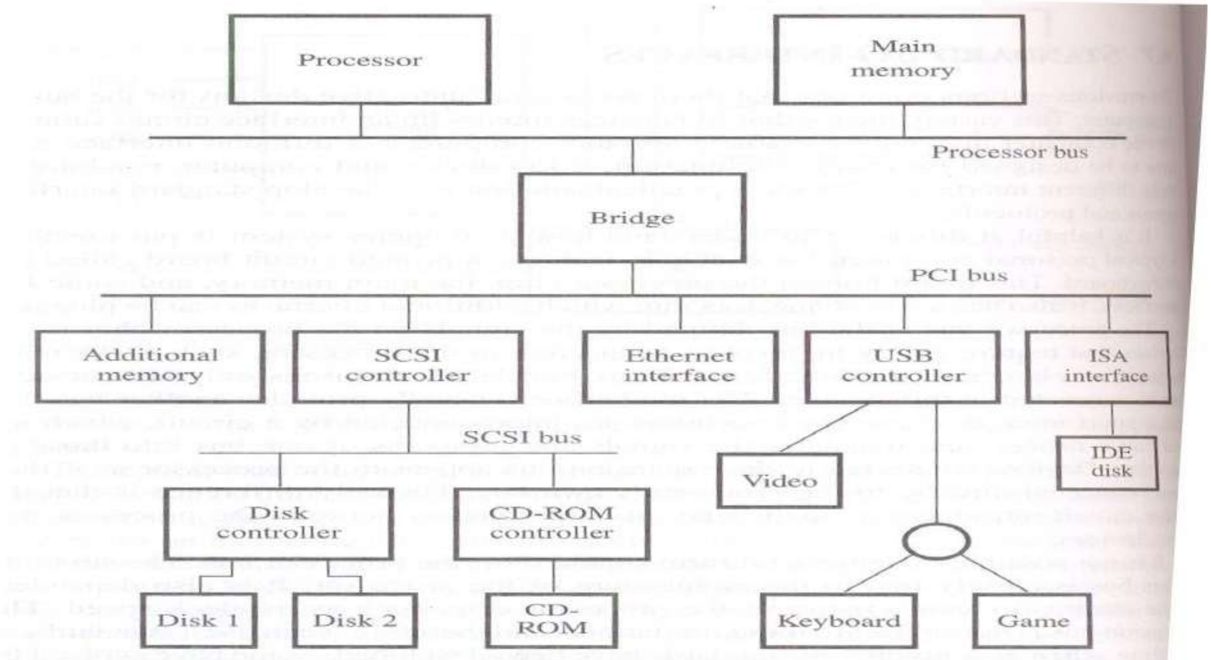


**Figure 4.38** An example of a computer system using different interface standards.

PCI → Defines an expansion bus on the motherboard.
SCSI → Used for connecting additional devices, both inside and outside the computer box
→ It is a high speed parallel bus intended for devices such as disk and
video Displays USB → Used for connecting additional devices, both inside and
outside the computer box
→ Uses serial transmission to suit the needs of equipment ranging
fromkeyboard to game controls to internet connections

## Peripheral Component Interconnect

- 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.
- The PCI follows a sequence of bus standards that were used primarily in IBM PCs. Early PCs used the 8-bit XT bus, whose signals closely mimicked those of Intel's *80x* 86 processors.

- 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.
- 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

- Data are transferred between the cache and the main memory in bursts of several words each.
- The words involved in such a transfer are stored at successive memory locations. When the processor specifies an address and requests a read operation from the main memory.
- The memory responds by sending a sequence of data words starting at that address. Similarly, during a write operation, the processor sends a memory address followed by a sequence of data words, to be written in successive memory locations starting at that address.

## SCSI BUS

- The acronym SCSI stands for **Small Computer System Interface**. It refers to a standard bus defined by the American National Standards Institute(ANSI).
- In the original specification 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 upto 5 megabytes.
- 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. Alternatively, a wide SCSI bus has 16 data lines and transfers data 16 bits at a time.
- There are also several options for the electrical signaling scheme used. The bus may use single-ended transmission (SE), where each signal uses one wire, with a common ground return for signals. In another option, differential signaling is used, where a separate return wire is provided for each signal.
- In this case, two voltage levels are possible. Earlier versions tile 5 V (TTL levels) and are known as High Voltage Differential (HVD). More recently, 3.3 V versions have been introduced and are known as Low Voltage Differential (LVD).
- 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.
- The maximum transfer rate on a given bus is often a function of thelength of the cable and the number of devices connected, with higher rates for a shorter cable and fewer devices.
- To achieve the top data transfer rate, the bus length is typically limited to 1.6 m for SE signaling and 12 m for LVD signaling. However, manufacturers

often provide special bus expanders to connect devices that are farther away.

- The maximum capacity of the bus is 8 devices for a narrow bus and 16 devices for a wide bus.
- 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. 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 *information about the device.*

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

- Speed
  - Low-speed(1.5 Mb/s)
  - Full-speed(12 Mb/s)
  - High-speed(480 Mb/s)
- Port Limitation: the parallel and serial ports provide a general purpose point of connection through which a variety of low to medium speed devices can be connected to a computer.Device Characteristics: the kinds of devices that 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.
- Plug-and-play: Feature means that a new device, such as an additional speaker can be connected at any time while system is operating.

## Universal Serial Bus tree structure

- To accommodate a large number of devices that can be added or removed at any time, the USB has the tree structure as shown in the figure.
- 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, Internet connection, speaker, or digital TV)
- 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**.**

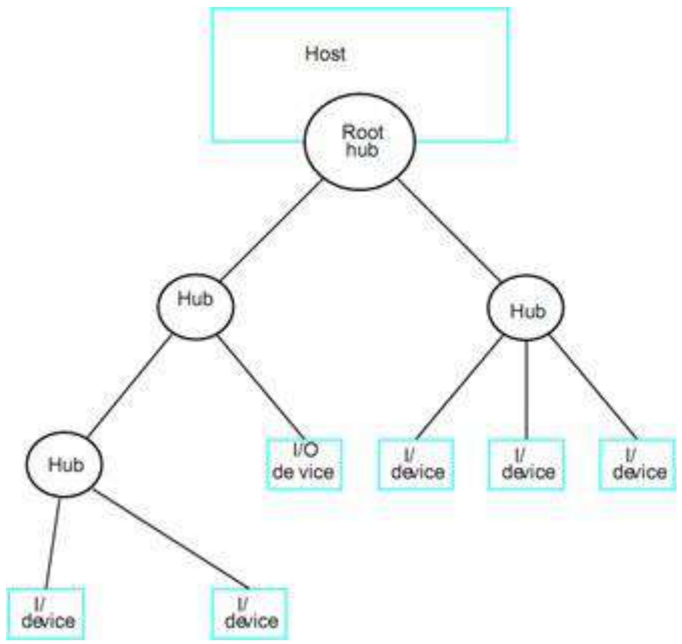


**Fig:Universal serial bus tree structure**

**Addressing**

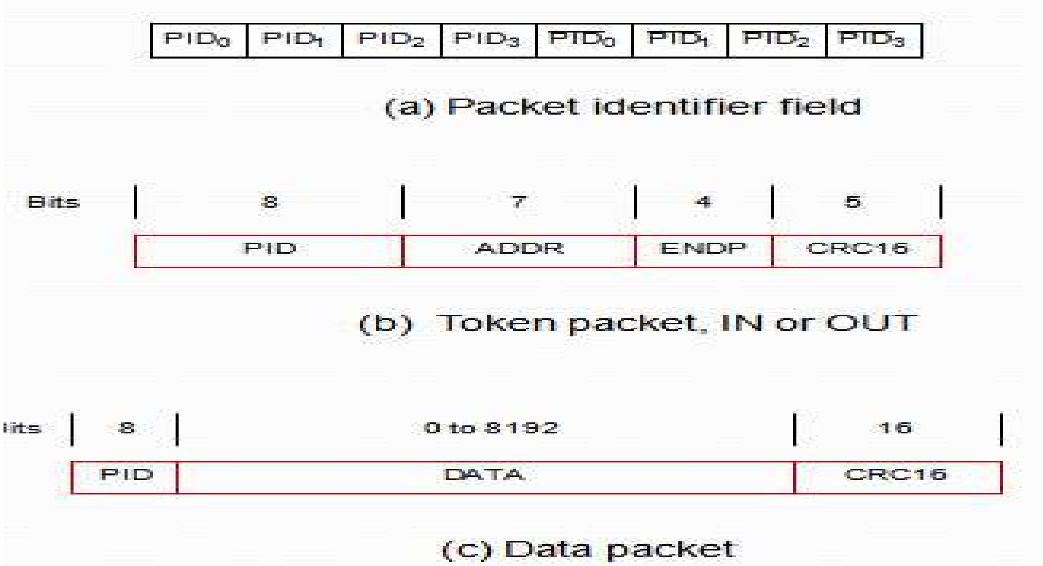
- When a USB is connected to a host computer, its root hub is attached to the processor bus, where it appears as a single device. The host software communicates with individual devices attached to the USB by sending packets of information, which the root hub forwards to the appropriate device in the 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.
- When the host is informed that a new device has been connected, it uses a sequence of commands to send a reset signal on the corresponding hub port, read information from the device about its capabilities, send configuration information to the device, and assign the device a unique USB address. Once this sequence is completed the device begins normal operation and responds only to the new address.

## USB Protocols

- 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.
- 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.
- They are transmitted twice. The first time they are sent with their true values, and the second time with each bit complemented
- The four PID bits identify one of 16 different packet types. Some control packets, such as ACK (Acknowledge), consist only of the PID byte.



(a) Packet identifier field

(b) Token packet, IN or OUT

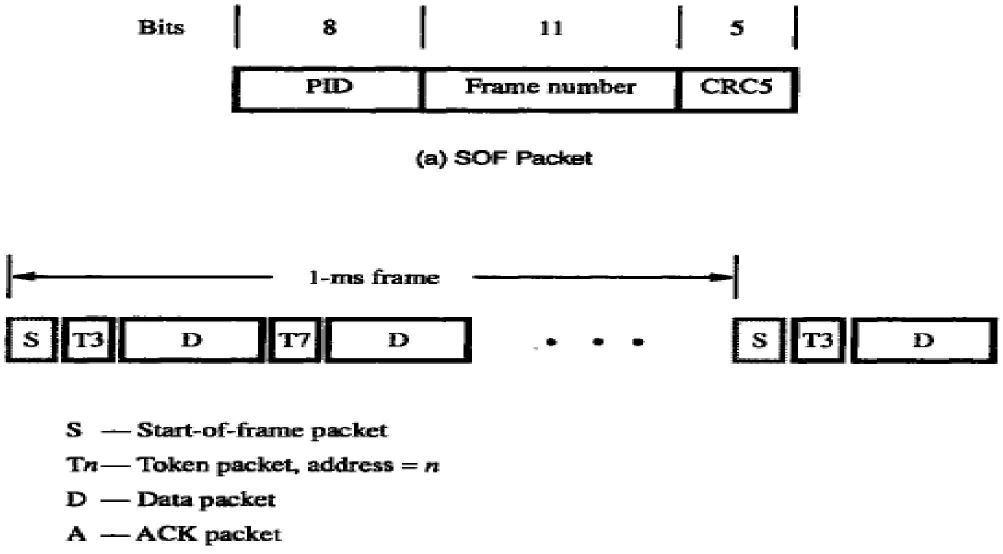
(c) Data packet

**Figure: USB packet format.**

**Isochronous Traffic on USB**

- One of the key objectives of the USB is to support the transfer of isochronous data.
- Devices that generates or receives isochronous data require a time reference to control the sampling process.
- To provide this reference. Transmission over the USB is divided into frames of
- equal length. A frame is 1ms long for low-and full-speed data.
- The root hub generates a Start of Frame control packet (SOF) precisely once every 1 ms to mark the beginning of a new frame.
- The arrival of an SOF packet at any device constitutes a regular clock signal that the device can use for its own purposes.
- To assist devices that may need longer periods of time, the SOF packet carries an 11-bit frame number.
- Following each SOF packet, the host carries out input and output transfers for isochronous devices.

□ This means that each device will have an opportunity for an input or output transfer once every 1 ms.



(a) SOF Packet

(b) Frame example

## Electrical Characteristics

- The cables used for USB Connections consist of four wires.
- Two are used to carry power, +5V and ground.
- Thus, a hub or an I/O device may be powered directly from the bus, or it may have its own. Different signaling schemes are used for different speeds of transmission.
- At low speed, 1s and 0s are transmitted by sending a high voltage state (5V) on one or the other of the two signal wires.
- □ For high-speed links, differential transmission is used. Control packets used for controlling data transfer operations are called token packets.