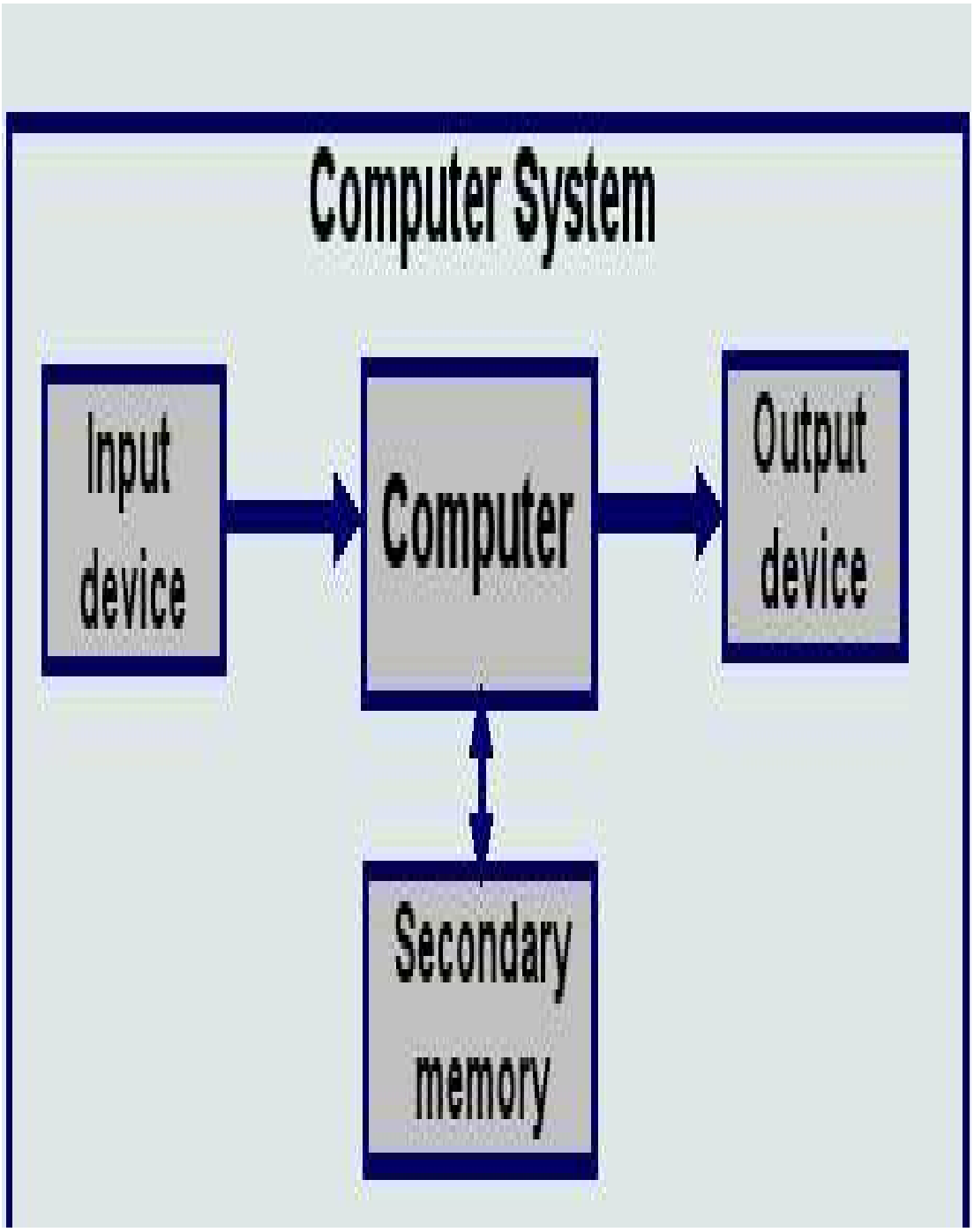
LECTURE 1: Overview Computer Organization
OUTLINE
- Overview of computer organization. Computer architecture vs organization.
-Computers and their instruction set architectures.
- Processor Design: The design process, A 1-Bus micro architecture , Data path implementation logic design for the 1-Bus , the control unit, the 1 and 3 Bus processor designs.
- RISC Vs CISC. Formal simple RISC computer.
- Pipelining, pipeline hazards.
- Superscalar and super pipeline
- VLIW
-Introduction to multiprocessors

Reference:

- William Stallings: Computer Organization and Architecture, 9th edition, Peason, 2013.
- Hennessy and patterson : : "Computer Architecture: A Quantitative Approach."

A Computer System
A computer system consists of a computer and its peripherals. Computer peripherals include input devices, output devices, and secondary memories.



Basic Principles of Computers

Virtually all modern computer designs are based on the von Neumann architecture principles:

- Data and instructions are stored in a single read/write memory.
- The contents of this memory are addressable by location, without regard to what are stored there.
- Instructions are executed sequentially (from one instruction to the next) unless the order is explicitly modified.

Why von-Neumann Architecture?

General-purpose, programmable:

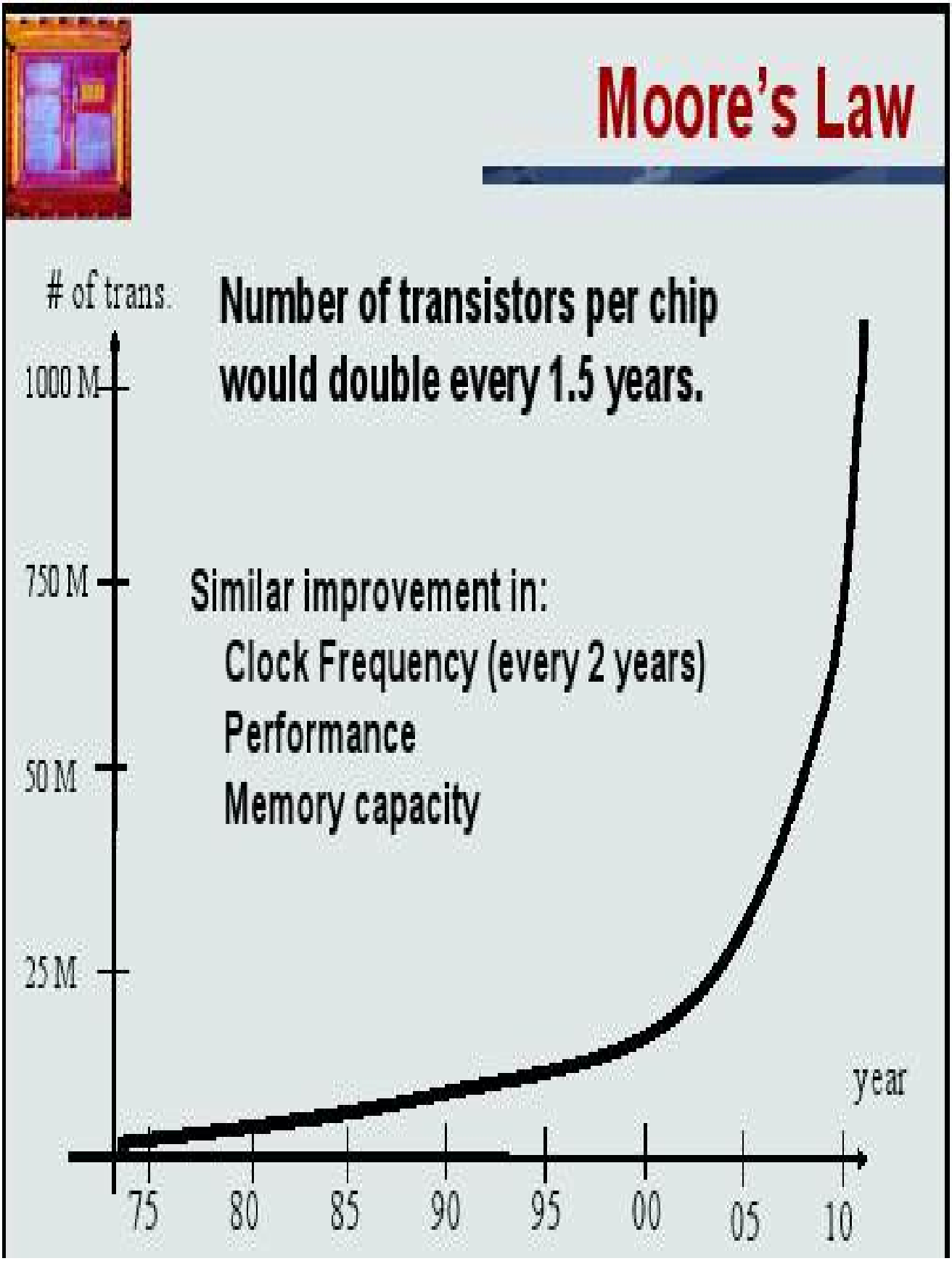
- They can solve very different problems by executing different programs.
- Instruction execution is done automatically.
- It can be built with very simple electronics components:

Data processing function is performed by electronic gates:

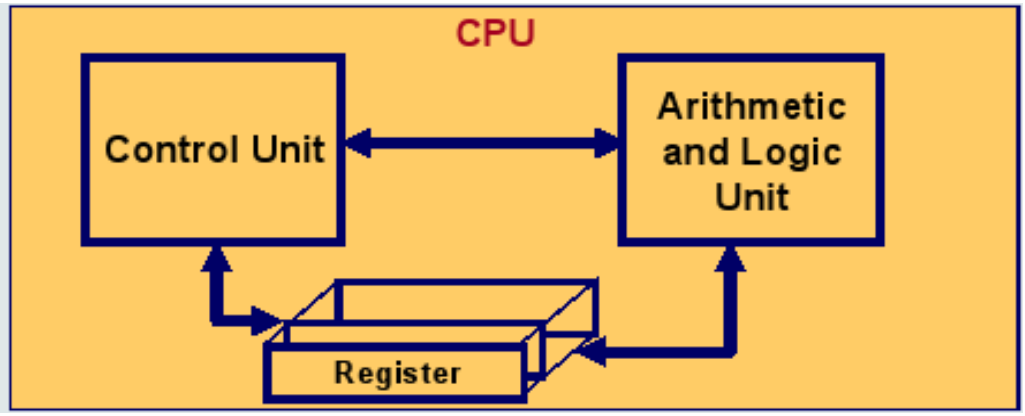
- Data storage function is provided by memory cells.
- Data communication is achieved by electrical wires.

Technology development: Moore's Law

It states that Number of transistors per chip would double every 1.5 years.


Moore's Law
# of trans.
Number of transistors per chip would double every 1.5 years.
1000 M
750 M
50 M
25 M
Similar improvement in:
Clock Frequency (every 2 years)
Performance
Memory capacity
year
75
80
85
90
95
00
05
10

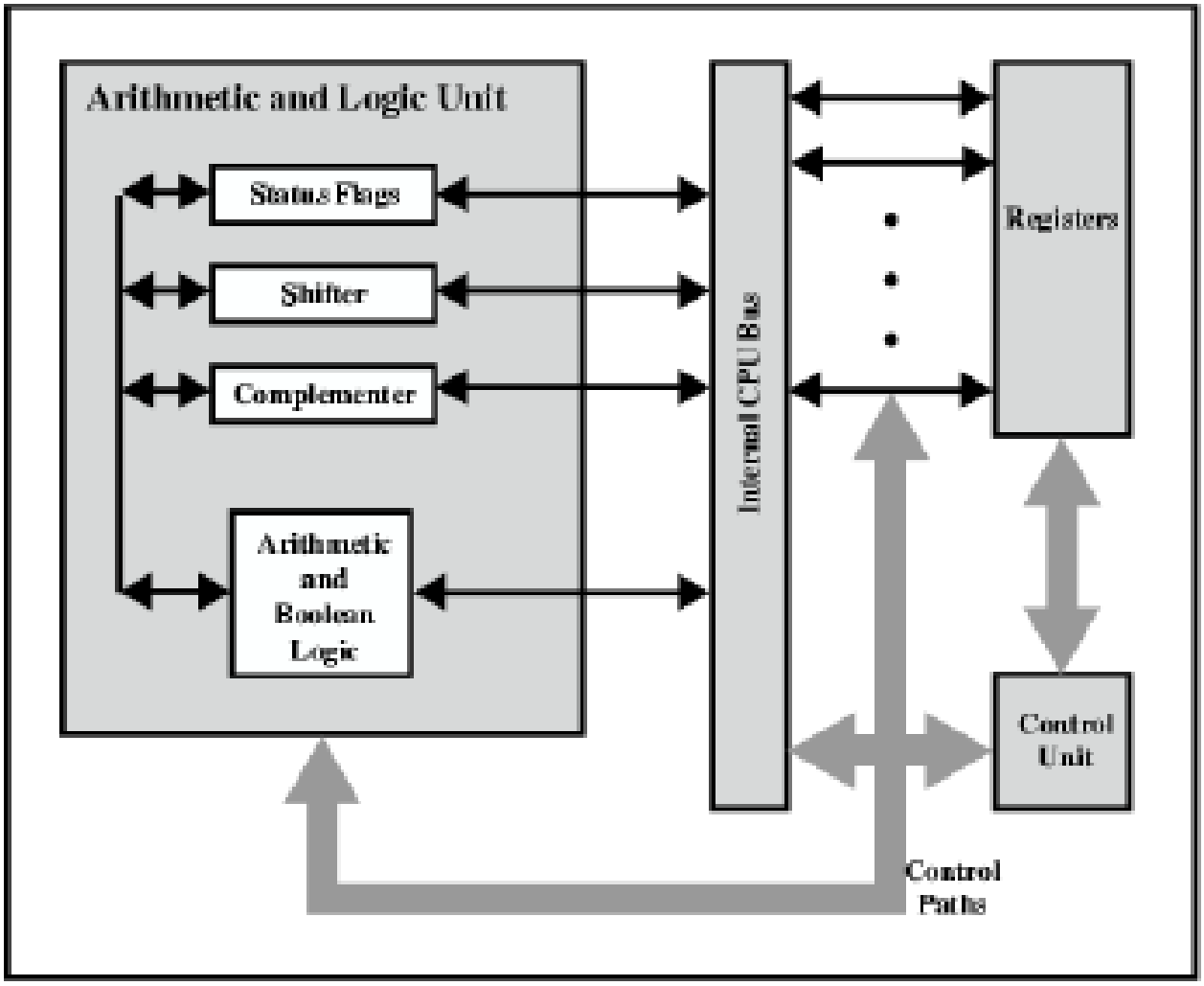
Central Processing Unit (CPU)
The Central Processing Unit (CPU), also called processor, includes three main units: A program control unit, Register and Arithmetic and Logic Unit (ALU).



The primary function of a CPU is to execute the instructions stored in the main memory.
-An instruction tells the CPU to perform one of its basic operations.

- The CPU includes also a set of registers, which are temporary storage devices used to hold control information, key data, and intermediate results.
- It includes also an internal bus infrastructure, which provides data movement paths among the control unit, ALU, and registers.
-The CU is the one which interprets (decodes) the instruction to be executed and "tells" the other components what to do.

CPU Internal Structure



Registers
CPU must have some working space (temporary storage) and these storage units are called registers. They are the top level component in the memory hierarchy.
Number and function of the registers vary between different computers and it is one of the major design decisions.

Register Organization
The registers serve two main functions:

- **User-Visible Registers**: used by machine or assembly language programmers to minimize memory access. This includes
  - General-purpose registers
  - Data registers
  - Address registers
  - Condition code registers

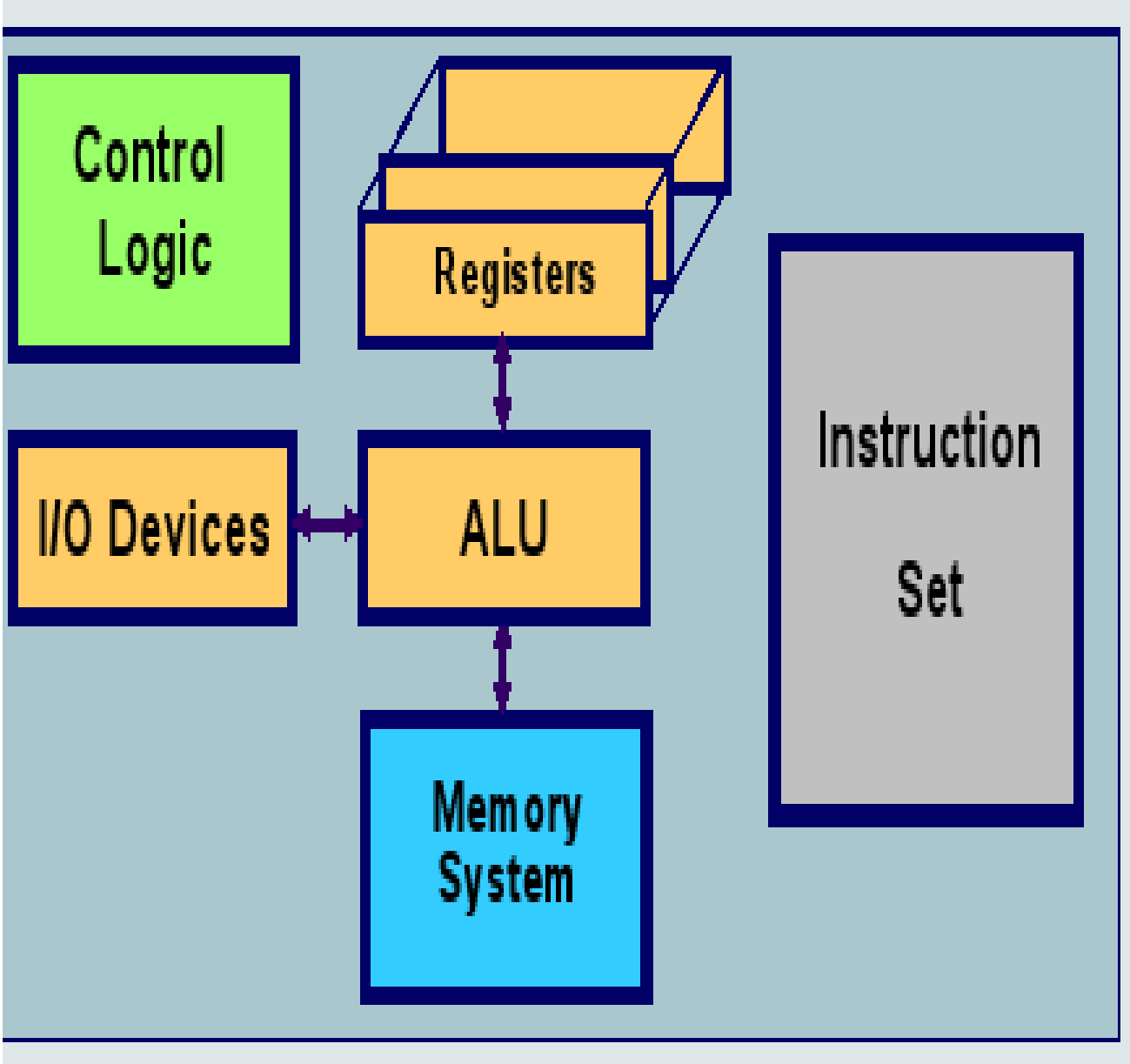
- **Control and Status Registers**: used by the control unit to control the operation of the CPU, and by the operating system to control the execution of programs. This includes
    - The Program Counter
    - Status registers
    - Instruction Registers

Difference btw architecture and Organization

Computer Architecture is the field of study of selecting and interconnecting hardware components to create computers that satisfy functional performance and cost goals. It refers to those attributes of the computer system that are visible to a programmer and have a direct effect on the execution of a program.

Computer architecture concerns Machine Organization, interfaces, application, technology, measurement & simulation that Includes:

- Instruction set
- Data formats
- Principle of Operation (formal description of every operation)
- Features (organization of programmable storage, registers used, interrupts mechanism, etc.)



In short, it is the combination of Instruction Set Architecture, Machine Organization and the related features.

Typical Architecture Attributes

- The instruction set.
- Data representation methods.
- The basic hardware units in the CPU.
- Functions of the main components.
- Instruction execution.
- Memory organization.
- I/O mechanisms.
- The ways in which the main components are interconnected.

Computer organization refers to the operational units and their interconnections that realize the architectural specifications

Example. Multiplication function:

- Architectural issue: having a multiply instruction or not.
- Organization issue: a special multiply unit or repeated use of the add unit to perform multiplication.

Many computer manufacturers offer a family of computer models all with the same architecture.E.g

- All Intel x86 family share the same basic architecture.
- The IBM System/370 family share the same basic architecture.
- This gives compatibility for new models.
- Organization differs between different versions with changing technology.

Evolution of Instruction Sets

Instruction Set Architecture (ISA) is an **a**bstract interface between the Hardware and lowest-level Software

- 1950: Single Accumulator: EDSAC
- 1953: Accumulator plus Index Registers: Manchester Mark I, IBM 700 series

Separation of programming Model from implementation:

- 1963: High-level language Based: B5000
- 1964: Concept of a Family: IBM 360

General Purpose Register Machines:

- 1977-1980: **CISC** - Complex Instruction Sets computer: Vax, Intel 432
- 1963-1976: **Load/Store Architecture**: CDC 6600, Cray 1
- 1987: **RISC**: Reduced Instruction Set Computer: Mips, Sparc, HP-PA, IBM RS6000

Typical features of RISC:

- Simple, no complex addressing
- Constant length instruction, 32-bit fixed format
- Large register file
- Hard wired control unit, no need for micro programming
- Just about every opposites of CISC

Major advances in computer architecture are typically associated with landmark instruction set designs. Computer architecture's definition itself has been through bit changes.

The following are the main concern for computer architecture through different times:

1930-1950: Computer arithmetic

- Microprogramming
- Pipelining
- Cache
- Timeshared multiprocessor

1960: Operating system support, especially memory management

- Virtual memory

1970-1980: **Instruction Set Design**, especially for compilers; **Vector processing** and **shared memory multiprocessors**

- RISC

1990s: Design of CPU, memory system, I/O system, multi-processors, networks

- CC-UMA multiprocessor
- CC-NUMA multiprocessor
- Not-CC-NUMA multiprocessor
- Message-passing multiprocessor

2000s: Special purpose architecture, functionally reconfigurable, special considerations for low power/mobile processing, chip multiprocessors, memory systems

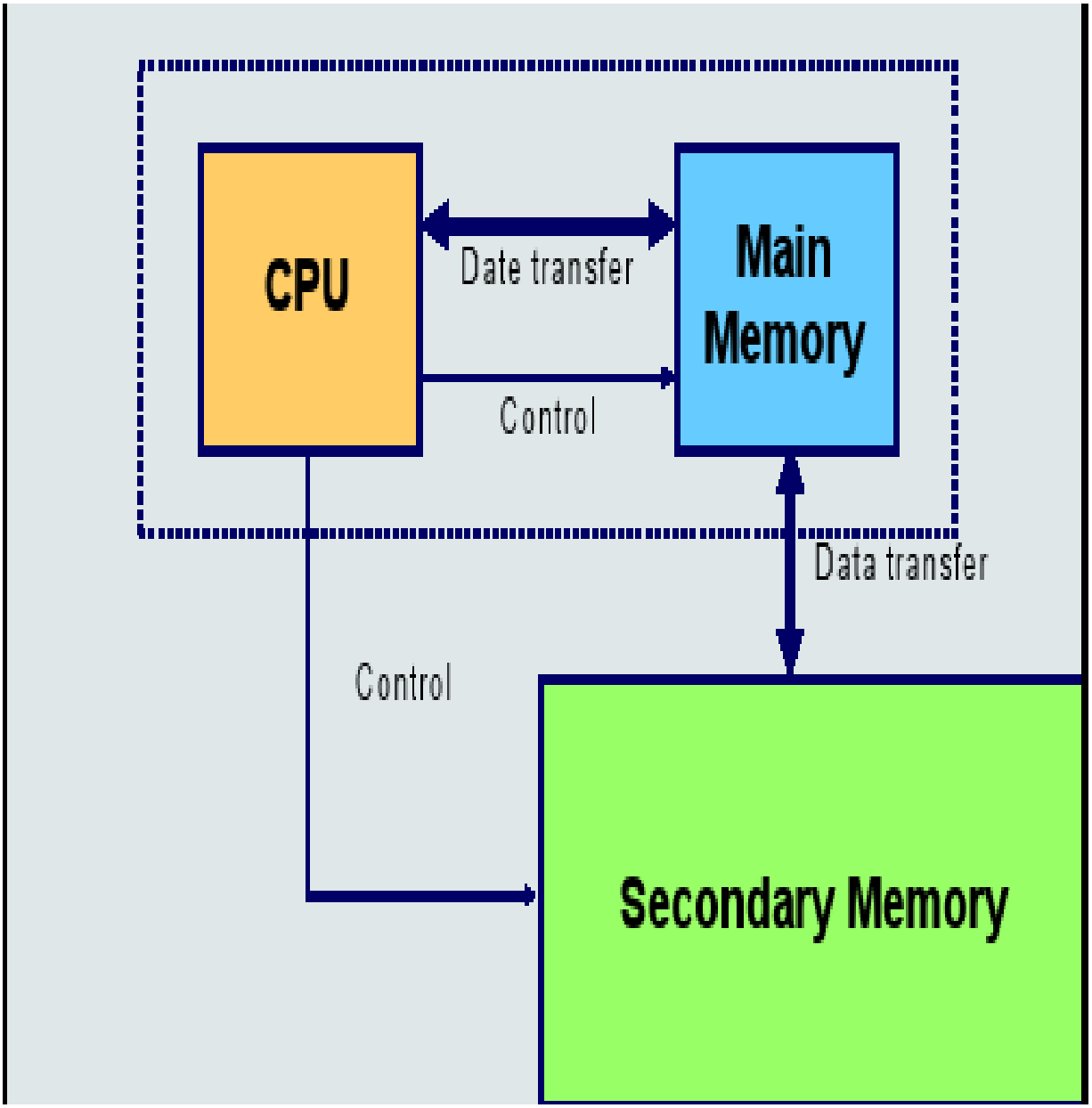
- Massive SIMD
- Parallel processing multiprocessor

Under a rapidly changing set of forces, computer technology keeps at dramatic change, for example:

- **Processor clock rate** at about 20% increase a year
- **Logic capacity** at about 30% increase a year
- **Memory speed** at about 10% increase a year
- **Memory capacity** at about 60% increase a year
- **Cost per bit** improves about 25% a year
- **The disk capacity** increase at 60% a year.

Memory Systems

Memories are group into internal and external memory



Memory Characteristics

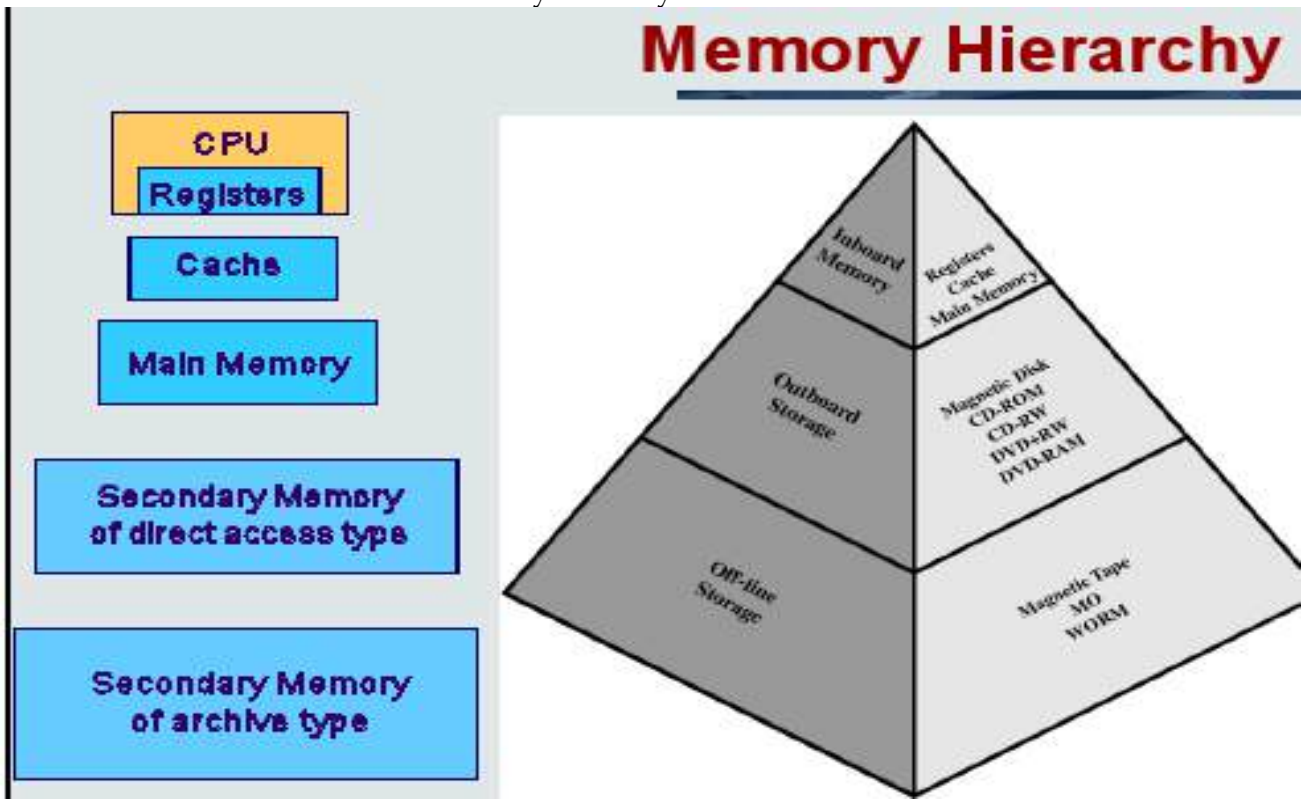
The most important characteristics of a memory are

Speed — as fast as possible;
Size — as large as possible;
Cost — reasonable price.

They are determined by the technology used for implementation.
We require a memory to store very large programs and to work at a speed comparable to that of the CPU but the reality is:
- The larger a memory, the slower it will be;
- The faster a memory, the greater the cost per bit.
- The larger a memory, the smaller the cost per bit.

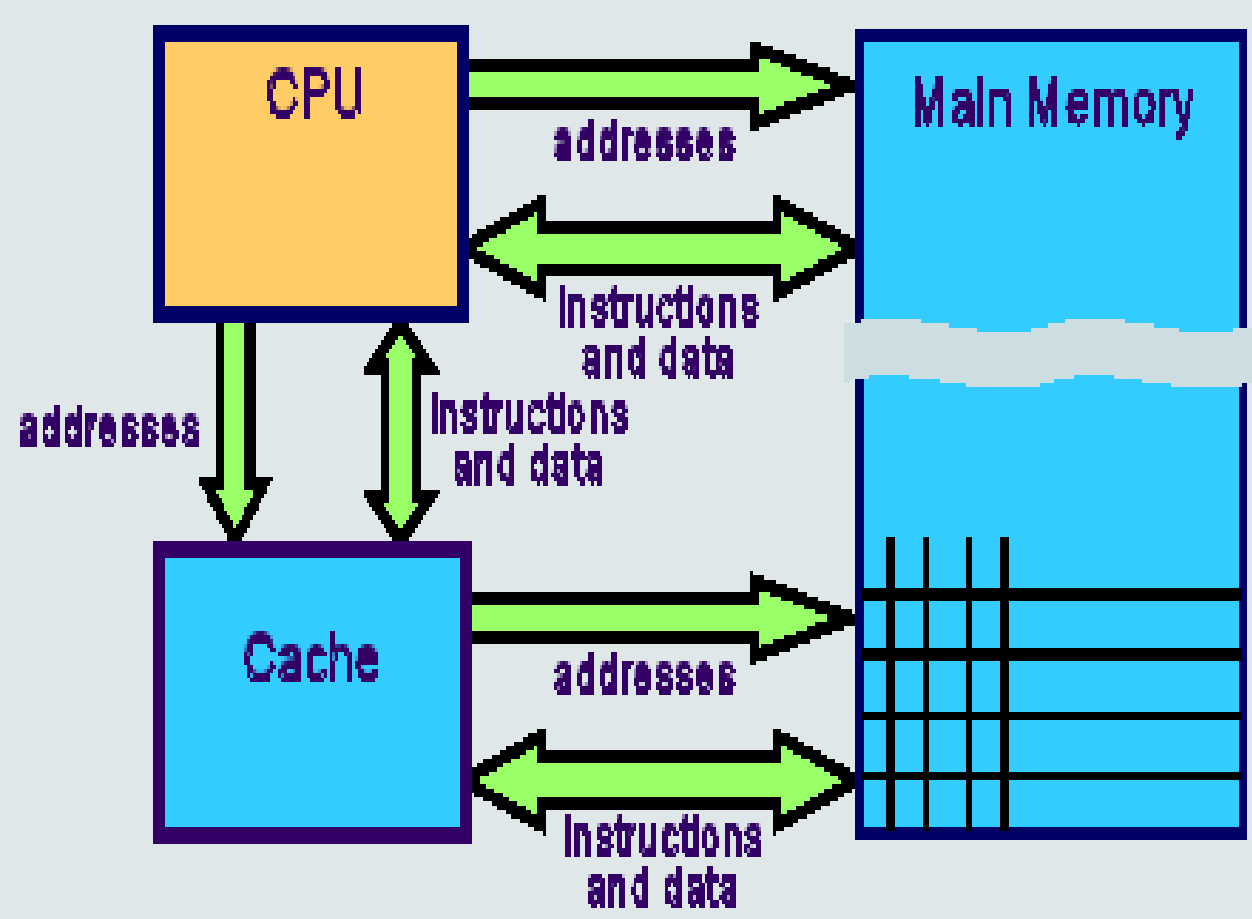
The solution it to build a composite memory system which combines a small and fast memory with a large and slow memory, and behaves, most of the time, like a large and fast memory.

These can be extended into a hierarchy of many levels.



1) Cache Memory:

A cache is a very fast memory which is put between the main memory and the CPU, and used to hold segments of program and data of the main memory.



Cache Memory Features

- It is transparent to the programmers.
- It is only a small part of the program/data in the main memory which has its copy in the cache (e.g., 8KB cache with 8MB memory).
- If the CPU wants to access program/data not in the cache (called a cache miss), the relevant block of the main memory will be copied into the cache.
- The intermediate-future memory access will usually refer to the same word or words in the neighborhood, and will not have to involve the main memory. This property of program executions is denoted as locality of reference.

Locality of Reference

Temporal locality: If an item is referenced, it will tend to be referenced again soon.

Spatial locality: If an item is referenced, items whose addresses are close by will tend to be referenced soon.

This access pattern is referred as locality of reference principle, which is an intrinsic features of the von Neumann architecture: It is applied in Sequential instruction storage, Loops and iterations (e.g., subroutine calls) and Sequential data storage (e.g., array).

Cache Design

The size and nature of the copied block must be care-fully designed, as well as the algorithm to decide which block to be removed from the cache when it is full: The design issues includes:

- Cache block size (line size).
- Total cache size.
- Mapping function.
- Replacement method.
- Write policy.
- Numbers of caches: Single, two-level, or three-level cache.
- Unified or split(data/instruction separate)

Mapping function

i) Direct Mapping: Each block of the main memory is mapped into a fixed cache slot.e.g A 10,000 word MM and a 100 word cache.10 Memory cells are grouped into a block.

Pros and cons

- Simple to implement and therefore inexpensive since it is Fixed location for blocks.
- If a program accesses 2 blocks that map to the same cache slot repeatedly, cache miss rate is very high.

ii) Associative Mapping: A main memory block can be loaded into any slot of the cache.

To determine if a block is in the cache, a mechanism is needed to simultaneously examine every slot's tag.

iii) Set Associative Organization: The cache is divided into a number of sets (K). Each set contains a number of slots (W). A given block maps to any slot in a given set. e.g. block i can be in any slot of set j.

Replacement Algorithms

With direct mapping, it is no need.

With associative mapping, a replacement algorithm is needed in order to determine which block to replace: This can be :

-First-in-first-out (FIFO).

- Least-recently used (LRU) - replace the block that has been in the cache longest with not reference to it.

-Least-frequently used (LFU) - replace the block that has experienced the fewest references.

- Random.

Write Policy

How to keep cache content and main memory content consistent without losing too much performance?

i) Write through:

All write operations are passed to main memory: If the addressed location is currently in the cache, the cache is updated so that it is coherent with the main memory. For writes, the processor always slows down to main memory speed.

Since the percentage of writes is small (ca. 15%), this scheme doesn't lead to large performance reduction

ii) Write through with buffered write:

The same as write-through, but instead of slowing the processor down by writing directly to main memory, the write address and data are stored in a high-speed write buffer; the write buffer transfers data to main memory while the processor continues its task.

Higher speed, but more complex hardware.

iii) Write back:

Write operations update only the cache memory which is not kept coherent with main memory. When the slot is replaced from the cache, its content has to be copied back to memory.

Good performance (usually several writes are performed on a cache block before it is replaced), but more complex hardware is needed.

Cache coherence problems are very complex and difficult to solve in multiprocessor systems

Cache Architecture Examples

Intel Pentium (introduced 1993)

- Has two on-chip caches, one for data and one for instructions each cache: 8 Kbytes
- line size: 32 bytes
-2-way set associative organization

IBM PowerPC 620 (introduced 1995)

- Has two on-chip caches, one for data and one for instructions each cache: 32 Kbytes
- line size: 64 bytes
- 8-way set associative organization

Machine Instructions

The CPU can only execute machine code in binary format, called machine instructions.

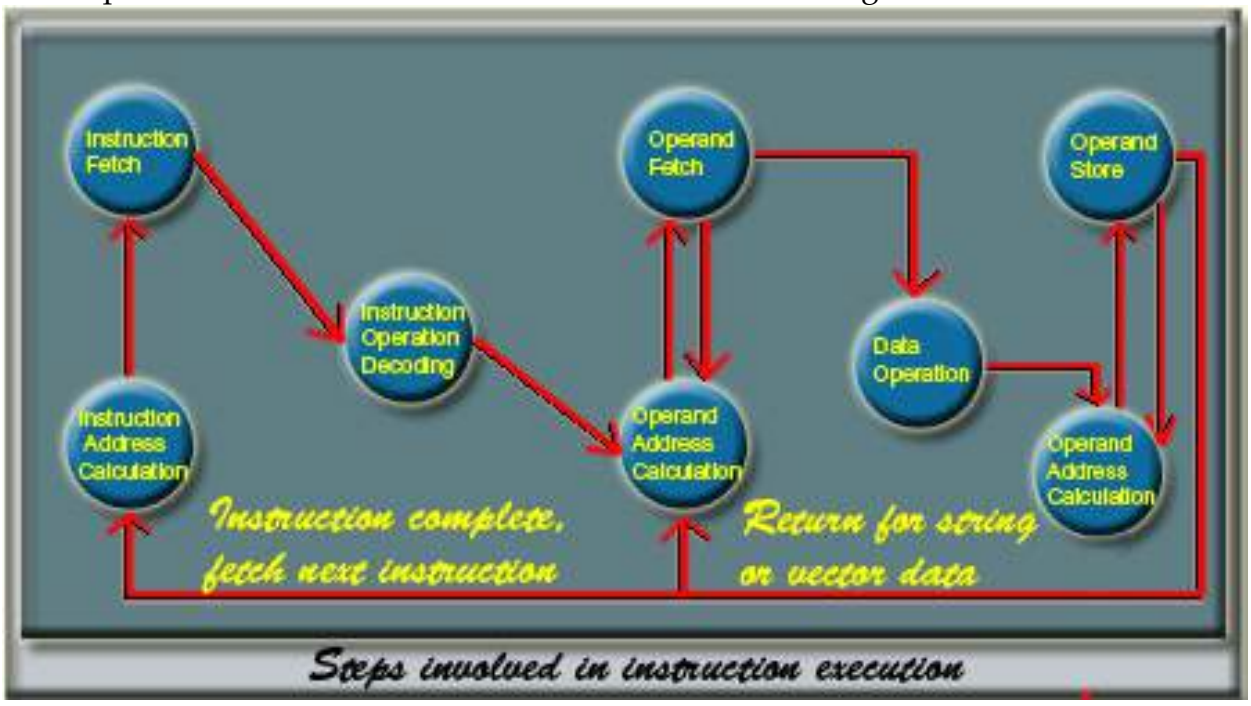
A machine instruction specifies the following information:

- What has to be done (the operation code)
- To whom the operation applies (source operands)
- Where does the result go (destination/Result operand)
- How to continue after the operation is finished (next instruction address).

The next instruction to be fetched is located in main memory. But in case of virtual memory system, it may be either in main memory or secondary memory (disk). In most cases, the next instruction to be fetched immediately follow the current instruction. In those cases, there is no explicit reference to the next instruction. When an explicit reference is needed, then the main memory or virtual memory address must be given. Source and result operands can be in one of the three areas:

- Main or virtual memory,
- CPU register or
- I/O device.

The steps involved in instruction execution is shown in the figure-



Instruction Representation

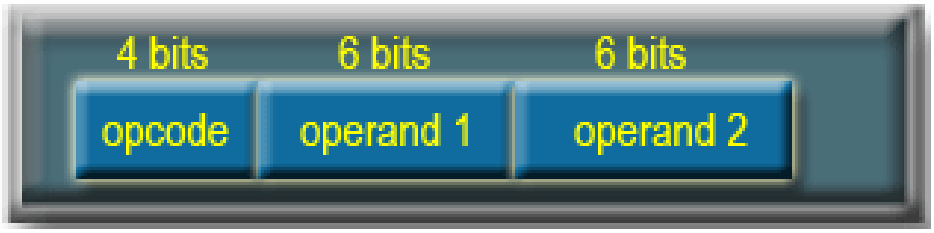
Within the computer, each instruction is represented by a sequence of bits. The instruction is divided into fields, corresponding to the constituent elements of the instruction. The instruction format is highly machine specific and it mainly depends on the machine architecture.

It is difficult to deal with binary representation of machine instructions. Thus, it has become common practice to use a symbolic representation of machine instructions. Opcodes are represented by abbreviations, called *mnemonics*,that indicate the operations.

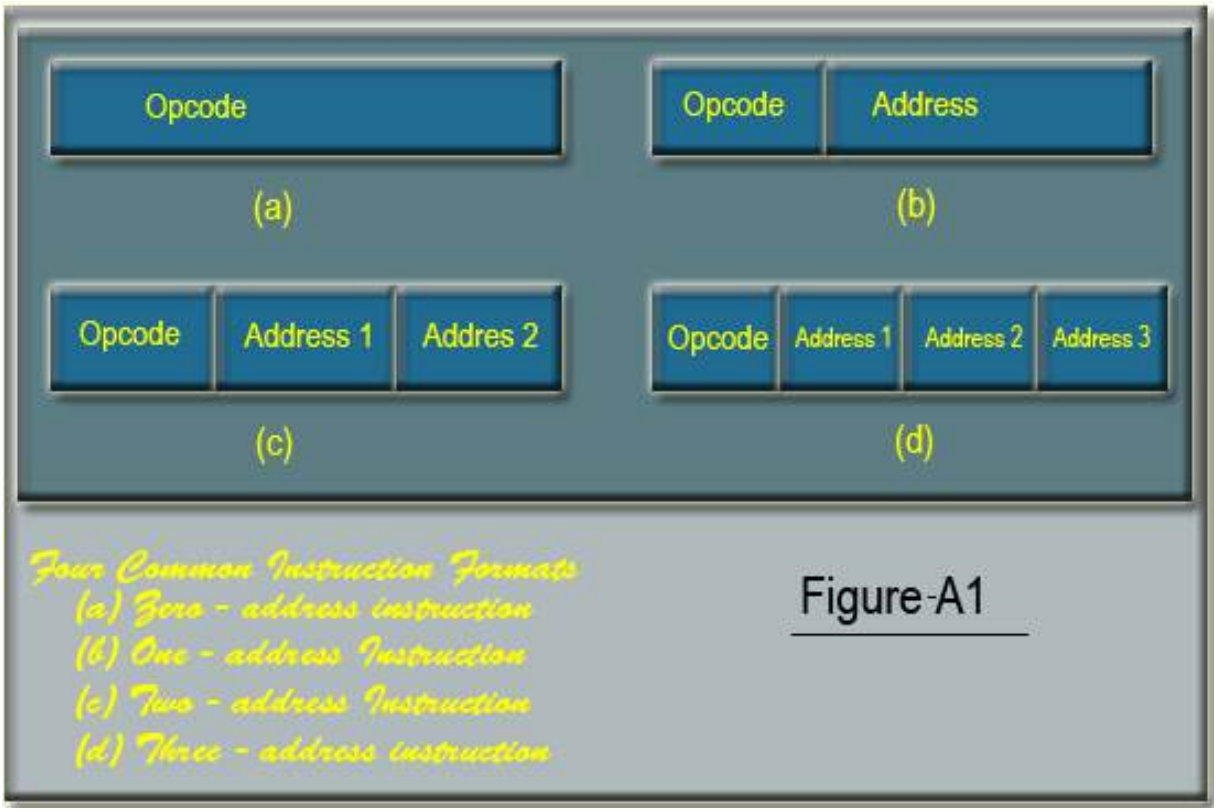
Common examples include:
ADD Add
SUB Subtract
MULT Multiply
DIV Division
LOAD Load data from memory to CPU
STORE Store data to memory from CPU.

Operands are also represented symbolically. For example, the instruction
MULT R, X : R ---------→ R x X
Mean multiply the value contained in the data location X by the contents of register R and put the result in register R
In this example, X refers to the address of a location in memory and R refers to a particular register.

Instruction Format:
An instruction format defines the layout of the bits of an instruction, in terms of its constituents parts.
A simple example of an instruction format is shown in the figure.



It is assumed that it is a 16-bit CPU. 4 bits are used to provide the operation code. So, we may have 16 ($2^4 = 16$) different set of instructions. With each instruction, there are two operands. To specify each operands, 6 bits are used.
An instruction format must include an opcode and, implicitly or explicitly, zero or more operands. Each explicit operand is referenced using one of the addressing mode that is available for that machine. The format must, implicitly or explicitly, indicate the addressing mode of each operand. For most instruction sets, more than one instruction format is used. Four common instruction format are shown below



Instruction Types

The instruction set of a CPU can be categorized as follows:

i) **Data Processing:**

Arithmetic and Logic instructions Arithmetic instructions provide computational capabilities for processing numeric data.

Logic (Boolean) instructions operate on the bits of a word as bits rather than as numbers. Logic instructions thus provide capabilities for processing any other type of data. There operations are performed primarily on data in CPU registers.

ii) **Data Storage:**

Memory instructions are used for moving data between memory and CPU registers.

iii) **Data Movement:**

I/O instructions I/O instructions are needed to transfer program and data into memory from storage device or input device and the results of computation back to the user.

iv) **Control:**

Test and branch instructions. Test instructions are used to test the value of a data word or the status of a computation. Branch instructions are then used to branch to a different set of instructions depending on the decision made.

Number of Addresses

What is the maximum number of addresses one might need in an instruction? Most of the arithmetic and logic operations are either *unary* (one operand) or *binary* (two

operands). Thus we need a maximum of two addresses to reference operands. The result of an operation must be stored, suggesting a third address. Finally after completion of an instruction, the next instruction must be fetched, and its address is needed. This reasoning suggests that an instruction may require to contain *four address references:* two operands, one result, and the address of the next instruction. In practice, four address instructions are rare. Most instructions have one, two or three operands addresses, with the address of the next instruction being implicit (obtained from the program counter).

Instruction Set Design

One of the most interesting, and most analyzed, aspects of computer design is instruction set design. The instruction set defines the functions performed by the CPU. The instruction set is the programmer's means of controlling the CPU. Thus programmer requirements must be considered in designing the instruction set.
Most important and fundamental design issues:

- **Operation repertoire :** How many and which operations to provide, and how complex operations should be.
- **Data Types :** The various type of data upon which operations are performed.
- **Instruction format :** Instruction length (in bits), number of addresses, size of various fields and so on.
- **Registers :** Number of CPU registers that can be referenced by instructions and their use.
- **Addressing :** The mode or modes by which the address of an operand is specified.

Types of Operands

Machine instructions operate on data. Data can be categorised as follows :

- **Addresses:** It basically indicates the address of a memory location. Addresses are nothing but the unsigned integer,but treated in a special way to indicate the address of a memory location. Address arithmatic is somewhat different from normal arithmatic and it is related to machine architecture.
- **Numbers:** All machine languages include numeric data types. Numeric data are classified into two broad categories: integer or fixed point and floating point.
- **Characters:** A common form of data is text or character strings. Since computer works with bits, so characters are represented by a sequence of bits. The most commonly used coding scheme is ASCII (American Standard Code for Information Interchange) code.
- **Logical Data:** Normally each word or other addressable unit (byte, halfword, and so on) is treated as a single unit of data. It is sometime useful to consider an n-bit unit as consisting of n 1-bit items of data, each item having the value 0 or 1. When data are viewed this way, they are considered to be logical data. Generally 1 is treated as true and 0 is treated as false.

Types of Operations

The number of different *opcodes* and their types varies widely from machine to machine. However, some general type of operations are found in most of the machine architecture. Those operations can be categorized as follows:

- Data Transfer
- Arithmetic
- Logical
- Conversion
- Input Output [I/O]
- System Control
- Transfer Control

Addressing Modes

The most common addressing techniques are:

- Immediate
- Direct
- Indirect
- Register
- Register Indirect
- Displacement
- Stack

All computer architectures provide more than one of these addressing modes. The question arises as to how the control unit can determine which addressing mode is being used in a particular instruction. Several approaches are used. Often, different opcodes will use different addressing modes. Also, one or more bits in the instruction format can be used as a mode field. The value of the mode field determines which addressing mode is to be used.

In a system without virtual memory, the effective address will be either a main memory address or a register. In a virtual memory system, the effective address is a virtual address or a register. The actual mapping to a physical address is a function of the paging mechanism and is invisible to the programmer.

To explain the addressing modes, we use the following notation:

A = contents of an address field in the instruction that refers to a memory
R = contents of an address field in the instruction that refers to a register
EA = actual (effective) address of the location containing the referenced operand
(X) = contents of location X

i) Immediate Addressing:

The simplest form of addressing is immediate addressing, in which the operand is actually present in the instruction:

OPERAND = A

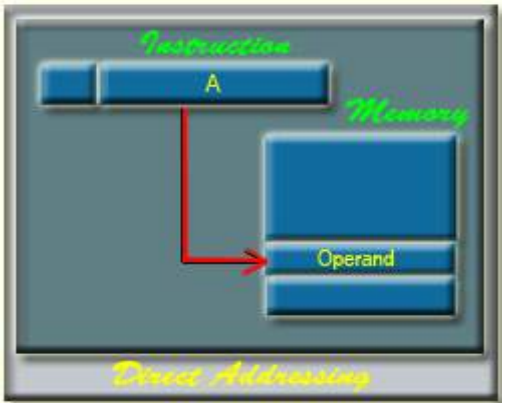
This mode can be used to define and use constants or set initial values of variables. The advantage of immediate addressing is that no memory reference other than the instruction fetch is required to obtain the operand. The disadvantage is that the size of the number is restricted to the size of the address field, which, in most instruction sets, is small compared with the world length.

ii) Direct Addressing:

A very simple form of addressing is direct addressing, in which the address field contains the effective address of the operand:

EA = A

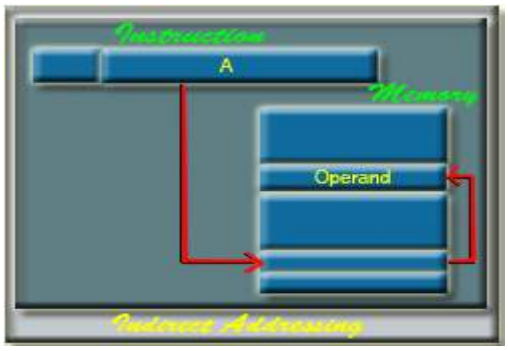
It requires only one memory reference and no special calculation.



iii) Indirect Addressing:

With direct addressing, the length of the address field is usually less than the word length, thus limiting the address range. One solution is to have the address field refer to the address of a word in memory, which in turn contains a full-length address of the operand. This is know as indirect addressing:

EA = (A)

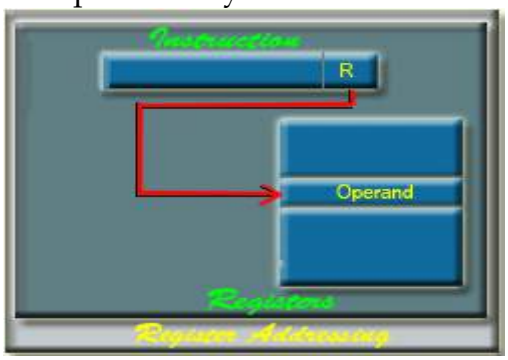


iv) Register Addressing:

Register addressing is similar to direct addressing. The only difference is that the address field referes to a register rather than a main memory address:

EA = R

The advantages of register addressing are that only a small address field is needed in the instruction and no memory reference is required. The disadvantage of register addressing is that the address space is very limited.



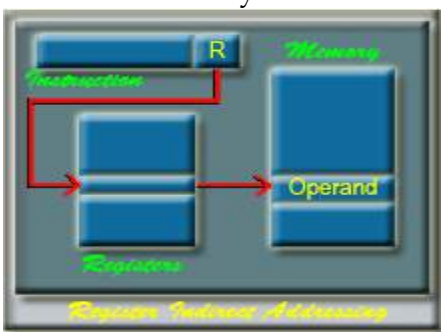
v) Register Indirect Addressing:

Register indirect addressing is similar to indirect addressing, except that the address field refers to a register instead of a memory location.

It requires only one memory reference and no special calculation.

EA = (R)

Register indirect addressing uses one less memory reference than indirect addressing. Because, the first information is available in a register which is nothing but a memory address. From that memory location, we use to get the data or information. In general, register access is much faster than the memory access.

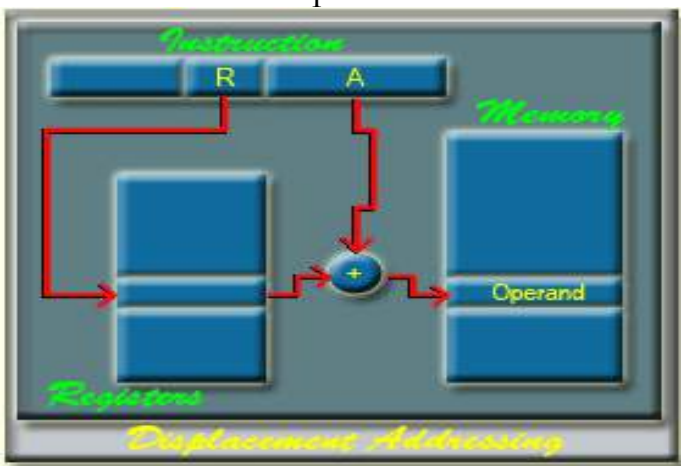


vi) Displacement Addressing:

A very powerful mode of addressing combines the capabilities of direct addressing and register indirect addressing, which is broadly categorized as displacement addressing:

EA = A + (R)

Displacement addressing requires that the instruction have two address fields, at least one of which is explicit. The value contained in one address field (value = A) is used directly. The other address field, or an implicit reference based on opcode, refers to a register whose contents are added to A to produce the effective address.



Three of the most common use of displacement addressing are:

i) Relative Addressing:

For relative addressing, the implicitly referenced register is the *program counter* (PC). That is, the current instruction address is added to the address field to produce the EA. Thus, the effective address is a displacement relative to the address of the instruction.

ii) Base-Register Addressing:

The reference register contains a memory address, and the address field contains a displacement from that address. The register reference may be *explicit* or *implicit*.

In some implementation, a single segment/base register is employed and is used implicitly. In others, the programmer may choose a register to hold the base address of a segment, and the instruction must reference it explicitly.

iii) Indexing:

The address field references a main memory address, and the reference register contains a positive displacement from that address. In this case also the register reference is sometimes explicit and sometimes implicit.

Generally index register are used for iterative tasks, it is typical that there is a need to increment or decrement the index register after each reference to it.

Because this is such a common operation, some system will automatically do this as part of the same instruction cycle.

This is known as *auto-indexing*. If general purpose register are used, the autoindex operation may need to be signaled by a bit in the instruction.

Auto-indexing using *increment* can be depicted as follows:

EA = A + (R)

R = (R) + 1

Stack Addressing:

A stack is a linear array or list of locations. It is sometimes referred to as a *pushdown list* or *last-in-firstout queue*. A stack is a reserved block of locations. Items are appended to the top of the stack so that, at any given time, the block is partially filled. Associated with the stack is a pointer whose value is the address of the top of the stack. The stack pointer is maintained in a register. References to stack locations in memory are in fact register indirect addresses.

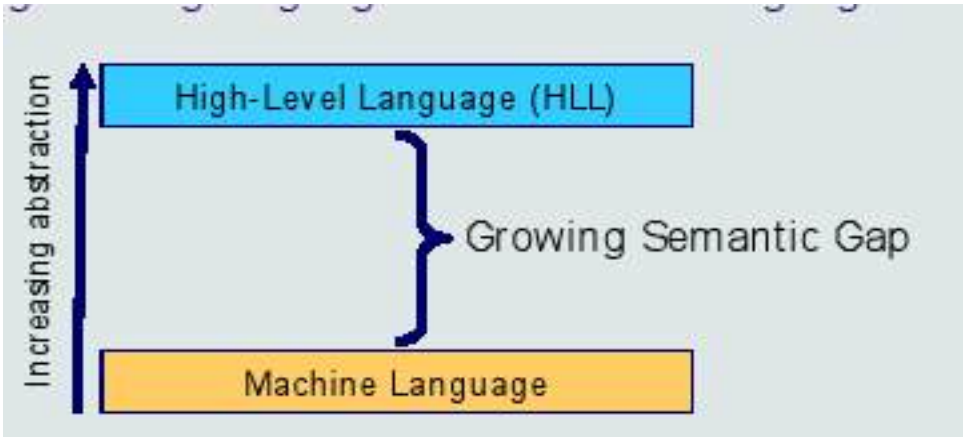
Lecture 3: CISC VS RISC Computers

Reduced Instruction Set Computer (RISC) represents an important innovation in computer architecture. It is an attempt to produce more CPU power by simplifying the instruction set of the CPU. The opposed trend to RISC is that of **Complex Instruction Set Computer** (CISC), or the "regular computers."

Both RISC and CISC architectures have been developed to address problems caused by the **semantic gap**, and consequently to reduce the ever growing software costs.

In order to improve efficiency of software development, powerful high-level programming languages have been developed (e.g., Ada, C++, Java) which support higher levels of abstraction.

This evolution has increased the semantic gap between programming languages and machine languages.



Main features of CISC

- CISC attempts to make ML instructions similar to HLL statements. For example, CASE (switch) on VAX.
- A large number of instructions (> 200) with complex instructions and data types.
- Many and complex addressing modes (e.g., indirect addressing is often used).
- Microprogramming techniques are used to implement the complicated instructions.

Memory bottleneck is a major problem, due to complex addressing modes and multiple memory accesses instruction.

Arguments for CISC

- A rich instruction set should simplify the compiler by having instructions which match HLL statements. This works fine if the number of HL languages is very small.
- Since the programs are smaller in size, they have better performance: They take up less memory space and need fewer instruction fetch cycles.
- Fewer number of instructions are executed, which may lead to smaller execution time.
- Program execution efficiency is improved by implementing complex operations in microcode rather than machine code.

Problems with CISC

A large instruction set requires complex and time consuming hardware steps to decode and execute instructions. Complex machine instructions may not match HLL statements exactly, in which case they may be of little use. This will be a major problem if the number of languages is getting bigger. Instruction sets designed with specialized instructions for several HL languages will not be efficient when executing program of a given language. It will lead also to a complex design tasks, thus large time-to-market.

Main Characteristics of RISC

- A small number of simple instructions (desirably < 100).

  Simple and small decode and execution hardware is required. A hard-wired controller is needed, rather than microprogramming. CPU takes less silicon area to implement, and runs also faster.

- Execution takes one instruction per clock cycle:- The instruction pipeline performs more efficiently due to simple instructions and similar execution patterns. Complex operations are executed as a sequence of simple instructions. In the case of CISC they are executed as one single or a few complex instructions.

- Load-and-store architecture

  Only LOAD and STORE instructions reference data in memory. All other instructions operate only with registers (are register-to-register instructions).

- Only a few simple addressing modes are used. Ex. register, direct, register indirect, and displacement.

-Instructions are of fixed length and uniform format.

  Loading and decoding of instructions are simple and fast. It is not needed to wait until the length of an instruction is known in order to start decoding it.Decoding is simplified because the opcode and address fields are located in the same position for all instructions.

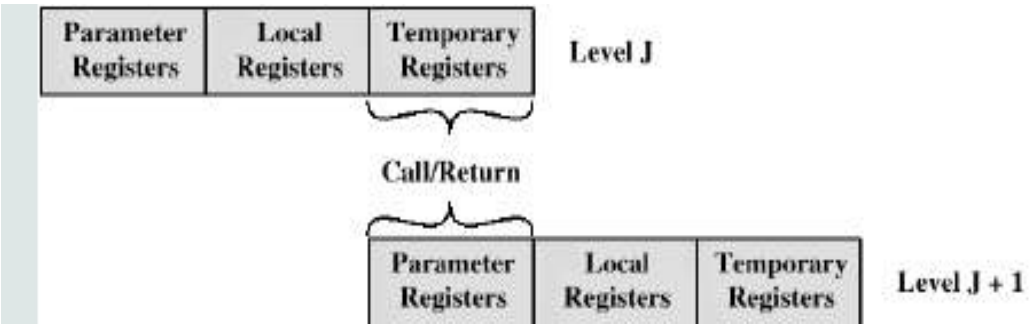
-A large number of registers is available.

  Variables and intermediate results can be stored in registers and do not require repeated loads and stores from/to memory. All local variables of procedures and the passed parameters can be stored in registers. The large number of registers is due to that the reduced complexity of the processor leaves silicon space on the chip to implement them which is usually not the case with CISC machines.

Register Windows

A large number of registers is usually very useful. However, if contents of all registers must be saved at every procedure call, more registers mean longer delay. A solution to this problem is to divide the register file into a set of fixed-size windows.

Each window is assigned to a procedure. Windows for adjacent procedures are overlapped to allow parameter passing.



Main Advantages of RISC

Best support is given by optimizing most used and most time consuming architecture aspects.

- Frequently executed instructions.
- Simple memory reference.
- Procedure call/return.
- Pipeline design.

Consequently, we have:

- High performance for many applications,
- Less design complexity;
- reducing design cost;
- reducing time-to-market.

Criticism of RISC

- An operation might need two, three, or more instructions to accomplish therefore more memory access might be needed hence execution speed may be reduced for certain applications.
- It usually leads to longer programs, which larger memory needs space to store.
- It makes it more difficult to program machine codes and assembly programs and more time consuming.

RISC vs. CISC

Studies have shown that benchmark programs run often faster on RISC processors than on CISC machines. However, it is difficult to identify which RISC feature really produces the higher performance. Some "CISC fans" argue that the higher speed is not produced by the typical RISC features but because of technology, better compilers, etc.

An argument in favor of the CISC: the simpler RISC instruction set esults in a larger memory requirement compared with the CISC case.

Most recent processors are not typical RISC or CISC, but combine advantages of both approaches. e.g. PowerPC and Pentium II.

A CISC Example — Intel i486.

32-bit integer processor:

Registers 8 general
6 address (16-bits)
2 status/control
1 instruction-pointer (PC)

On-chip floating-point unit

Microprogrammed control

Instruction set:

Number of instructions: 235

- e.g., “FYL2XP1 x y” performs y•log2(x+1) in 313 clock cycles.

Instruction size: 1-12 bytes (3.2 on average).

Addressing modes: 11

Memory organization:

Address length: 32 bits (4 GB space)

Memory is segmented for protection purpose

Support virtual memory by paging

Cache-memory: 8 KB internal (on-chip) (96% hit rate for DOS applications)

Instruction execution:

Execution models: reg-reg, reg-mem, & mem-mem.

Five-stage instruction pipeline: Fetch, Decode1, Decode2, Execute, Write-Back.

A RISC Example — SPARC

32-bit processor:

An Integer Unit (IU) — to execute all instructions.

A Floating-Point Unit (FPU) — co-processor working concurrently with IU for arithmetic operations on floating point numbers.

Instruction set: Has 69 basic instructions.

Has a linear, 32-bit virtual-address space (4G).

Registers: 40 to 520 general purpose 32-bit registers. Grouped into 2 to 32 overlapping register windows + 8 global registers. Each group has 16 registers.

Lecture 3: Instruction Pipelining

It is observed that organization enhancements to the CPU can improve performance. We have already seen that use of multiple registers rather than a single a accumulator, and use of cache memory improves the performance considerably. Another organizational approach, which is quite common, is instruction pipelining.

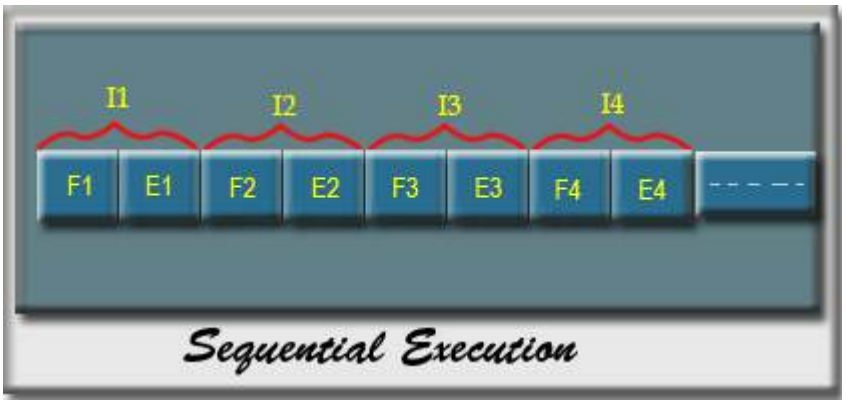
Pipelining is a particularly effective way of organizing parallel activity in a computer system. The basic idea is very simple. It is frequently encountered in manufacturing plants, where pipelining is commonly known as an assembly line operation.

By laying the production process out in an assembly line, product at various stages can be worked on simultaneously. This process is also referred to as pipelining, because, as in a pipeline, new inputs are accepted at one end before previously accepted inputs appear as outputs at the other end.

To apply the concept of instruction execution in pipeline, it is required to break the instruction in different task. Each task will be executed in different processing elements of the CPU.

As we know that there are two distinct phases of instruction execution: one is instruction fetch and the other one is instruction execution. Therefore, the processor executes a program by fetching and executing instructions, one after another.

Let F_i and E_i refer to the fetch and execute steps for instruction. Execution of a program consists of a sequence of fetch and execute steps



Now consider a CPU that has two separate hardware units, one for fetching instructions and another for executing them.

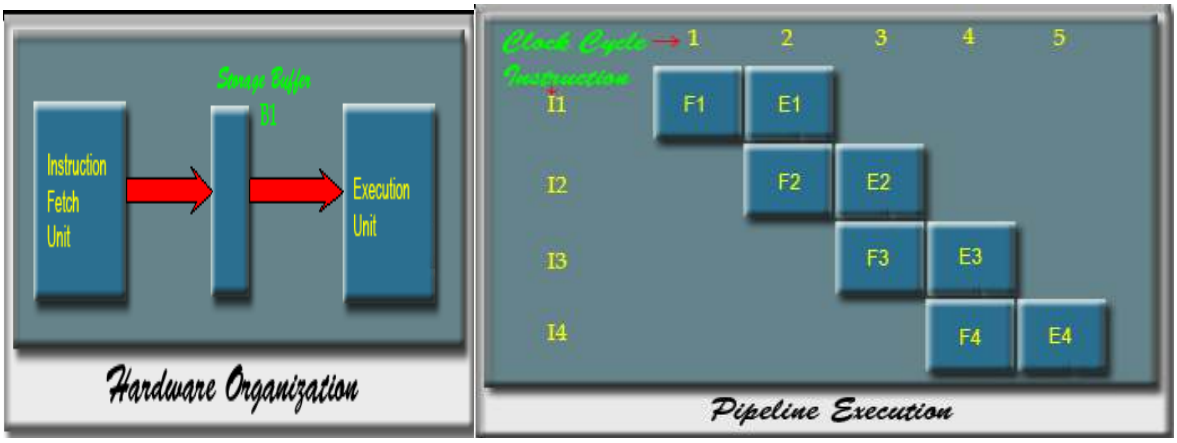
The instruction fetch by the fetch unit is stored in an intermediate storage buffer B_i
The results of execution are stored in the destination location specified by the instruction.

For simplicity it is assumed that fetch and execute steps of any instruction can be completed in one clock cycle.

The operation of the computer proceeds as follows:

- In the first clock cycle, the fetch unit fetches an instruction (instruction I_i , step F_i) and stored it in buffer B_1 at the end of the clock cycle.

- In the second clock cycle, the instruction fetch unit proceeds with the fetch operation for instruction I_2 (step F_2).
- Meanwhile, the execution unit performs the operation specified by instruction I_1which is already fetched and available in the buffer B_1 (step E_1).
- By the end of the second clock cycle, the execution of the instruction I_1 is completed and instruction I_2 is available.
- Instruction I_2 is stored in buffer B_1 replacing I_1 , which is no longer needed.
- Step E_2 is performed by the execution unit during the third clock cycle, while instruction is being fetched by the fetch unit.
- Both the fetch and execute units are kept busy all the time and one instruction is completed after each clock cycle except the first clock cycle.
- If a long sequence of instructions is executed, the completion rate of instruction execution will be twice that achievable by the sequential operation with only one unit that performs both fetch and execute.
-



The processing of an instruction need not be divided into only two steps. To gain further speed up, the pipeline must have more stages.
Let us consider the following decomposition of the instruction execution:

- Fetch Instruction (FI): Read the next expected instruction into a buffer.
- Decode Instruction ((DI): Determine the opcode and the operand specifiers.
- Calculate Operand (CO): calculate the effective address of each source operand.
- Fetch Operands(FO): Fetch each operand from memory.
- Execute Instruction (EI): Perform the indicated operation.
- Write Operand(WO): Store the result in memory.

There will be six different stages for these six subtasks. For the sake of simplicity, let us assume the equal duration to perform all the subtasks.

From this timing diagram it is clear that the total execution time of 8 instructions in this 6 stages pipeline is 13-time unit. The first instruction gets completed after 6 time unit, and thereafter in each time unit it completes one instruction.
Without pipeline, the total time required to complete 8 instructions would have been 48 (6 X 8) time unit. Therefore, there is a speed up in pipeline processing and the speed up is related to the number of stages.

Number of Pipeline Stages

In general, a larger number of stages gives better performance. However:

- A larger number of stages increases the overhead in moving information between stages and synchronization between stages.
- The complexity of the CPU grows with the number of stages.
- It is difficult to keep a large pipeline at maximum rate because of pipeline hazards.

Examples:

Intel 80486 and Pentium:

Five-stage pipeline for integer instructions.
Eight-stage pipeline for FP (floating points) instructions.

IBM PowerPC:

Have different numbers of stages (3-9) for different machines. PowerPC 440, introduced in 1999, has 7 stages.

Pipeline Hazards (Conflicts)

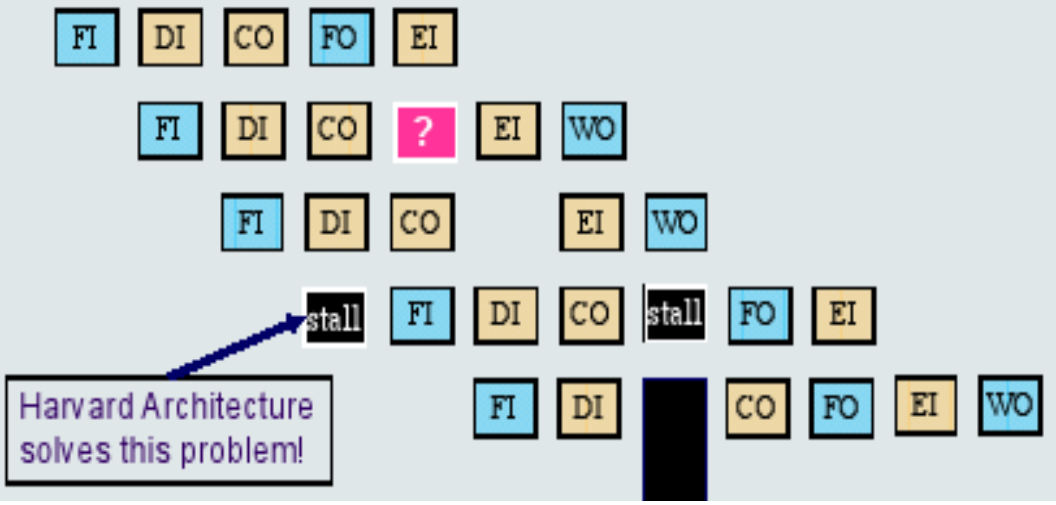
Keeping a pipeline at its maximal rate is prevented by pipeline hazards.
These are situations that prevent the next instruction in the instruction stream from executing during its designated clock cycle. Therefore the instruction is said to be stalled.
When an instruction is stalled, all instructions later in the pipeline than it are also stalled and no new instructions are fetched during the stall
Instructions earlier than the stalled one continue as usual.

Types of hazards:

a. Structural hazards
b. Data hazards
c. Control hazards

Structural (Resource) Hazards

Hardware conflicts is caused by the use of the same hardware resource at the same time (e.g., memory conflicts).



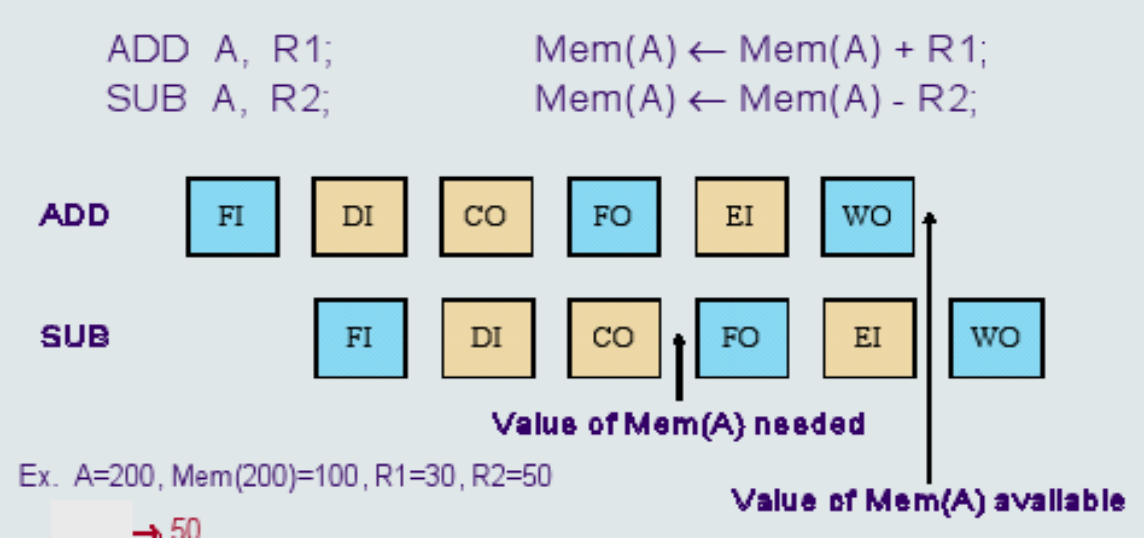
Penalty: 1 cycle (NOTE: the performance lost is multiplied by the number of stages).

Structural Hazard Solutions

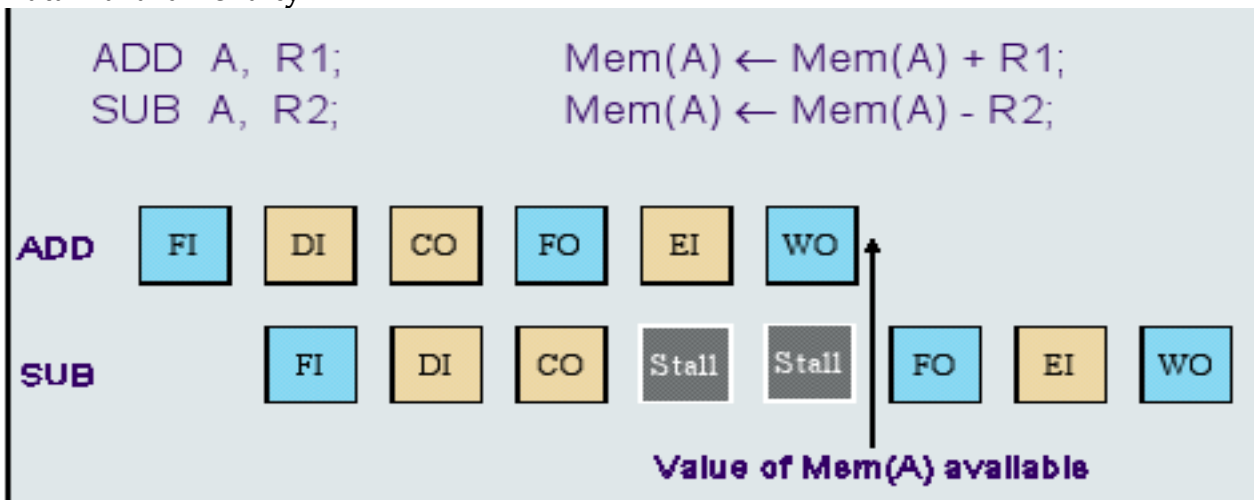
- In general, the hardware resources in conflict are duplicated in order to avoid structural hazards. -Functional units (ALU, FP unit) can also be pipelined themselves to support several instructions at the same time.
- Memory conflicts can be solved by having two separate caches, one for instructions and the other for operands (Harvard architecture);
- Using multiple banks of the main memory; or keeping as many intermediate results as possible in the registers (!).

Data Hazards

It is caused by reversing the order of data-dependent operations due to the pipeline (e.g., WRITE/READ conflicts).



Data Hazard Penalty

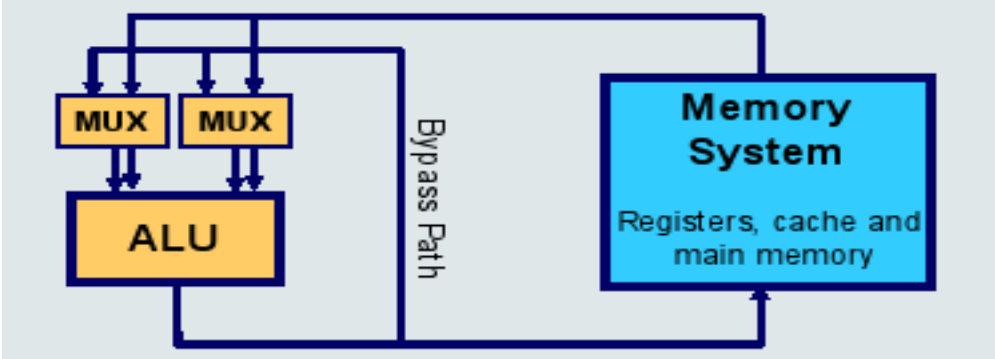


Data hazard is an important issue:

- Penalty: 2 cycles.
- It happens very often, since we have many data dependencies.

Data Hazard Solutions

The penalty due to data hazards can be reduced by a technique called forwarding (bypassing).

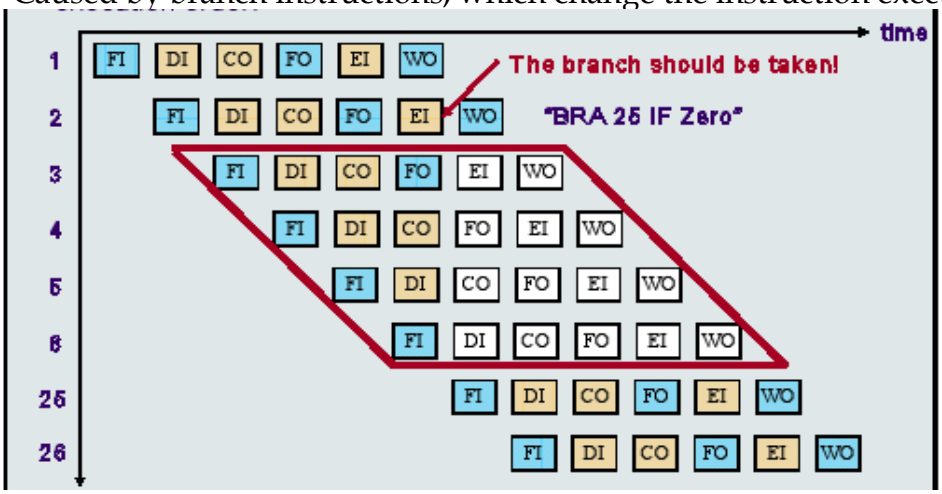


The ALU result is fed back to the ALU input.

If it detects that the value needed for an operation is the one produced by the previous one, and has not yet been written back.
ALU selects the forwarded result, instead of the value from the memory system.

Control Hazards

Caused by branch instructions, which change the instruction execution order.



Branch Handling

- Stop the pipeline until the branch instruction reaches the last stage. This leads to very large lost of performance, in particular, since 20%-35% of the instructions executed are branches.
- Multiple streams — implement hardware resources to deal with different branch alternatives. This is an expensive solution and you need special memory techniques to fully utilize it.
- Pre-fetch branch target — when a conditional branch is recognized, the following instruction is fetched, and the branch target is also pre-fetched.
- Loop buffer — use a small, high-speed memory to keep the most recently fetched instructions. If a branch is to be taken, the buffer is first checked to see if the branch target is in it. Special cache for branch target instructions is required.
- Delayed branch — re-arrange the instructions so that branching occur later than originally specified. E.g The compiler or the programmer has to find an instruction which

can be moved from its original place to the branch delay slot (it will be executed regardless of the branch outcome). It is 60% to 85% success rate but it leads, however, to un-readable code.

Branch Prediction

When a branch is encountered, a prediction is made and the predicted path is followed. The instructions on the predicted path are fetched. The fetched instruction can also be executed and this is call **Speculative Execution**.

Results produced of these executions should be marked as tentative.

When the branch outcome is decided, if the prediction is correct, the special tags on tentative results are removed. If not, the tentative results are removed, and the execution goes to the other path.

Branch prediction is based on static or dynamic information.

Static Branch Prediction: Predict is always taken. It assumes that jump will happen and always fetch target instruction.

Dynamic Branch Prediction: It is based on branch history. It store information regarding branches in a branch-history table so as to more accurately predict the branch outcome. It assume that the branch will do what it did last time.

Lecture 4: Superscalar Processors

Superscalar Architecture

Superscalar is a computer designed to improve the performance of the execution of scalar instructions.
A scalar is a variable that can hold only one atomic value at a time, e.g., an integer or a real. A scalar architecture processes one data item at a time the computers we discussed up till now.
Examples of non-scalar variables:

- Arrays
- Matrices
- Records

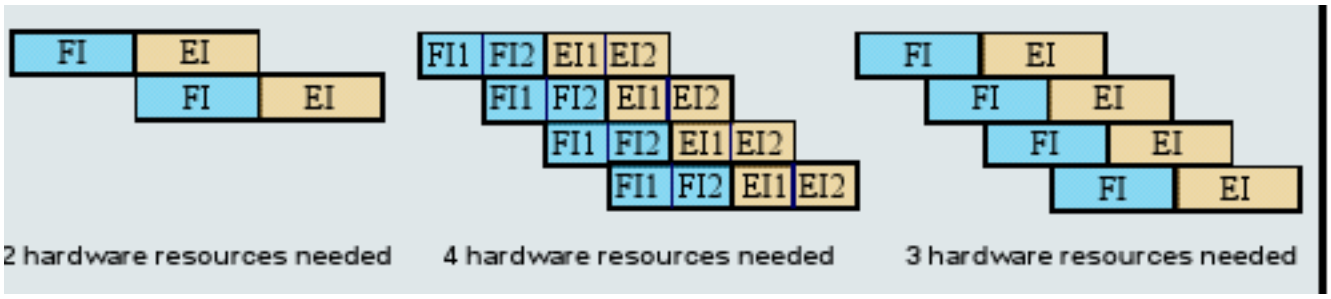
In a superscalar architecture (SSA), several scalar instructions can be initiated simultaneously and executed independently.
Pipelining allows also several instructions to be executed at the same time, but they have to be in different pipeline stages at any given moment.
SSA includes all features of pipelining but, in addition, there can be several instructions executing simultaneously in the same pipeline stage.
SSA introduces therefore a new level of parallelism, called **instruction-level parallelism.**
Most operations are on scalar quantities (about 80%). Speedup these operations will lead to large performance improvement.

How to implement the idea?

- A SSA processor fetches multiple instructions at a time, and attempts to find nearby instructions that are independent of each other and therefore can be executed in parallel.
- Based on the dependency analysis, the processor may issue and execute instructions in an order that differs from that of the original machine code.
- The processor may eliminate some unnecessary dependencies by the use of additional registers and renaming of register references.

Super pipelining

Super pipelining is based on dividing the stages of a pipeline into several sub-stages, and thus increasing the number of instructions which are handled by the pipeline at the same time.



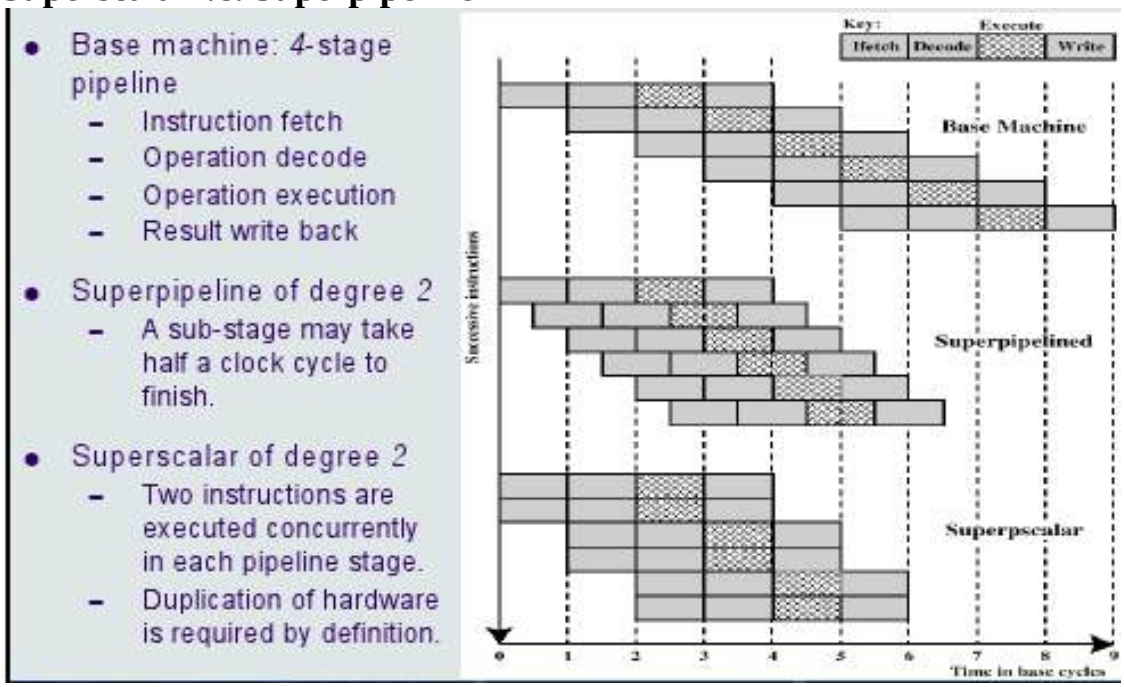
For example, by dividing each stage into two sub-stages, a pipeline can perform at twice the speed in the ideal situation.
Many pipeline stages may perform tasks that require less than half a clock cycle. No

duplication of hardware is needed for these stages.

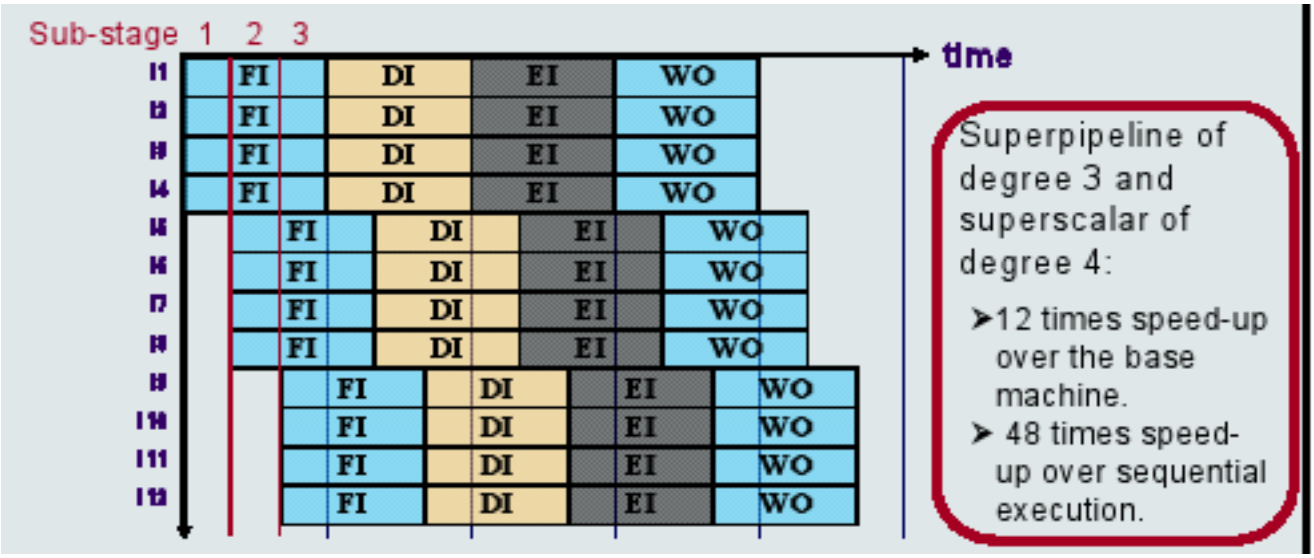
For a given architecture and the corresponding instruction set there is an optimal number of pipeline stages/sub-stages. Increasing the number of stages/sub-stages over this limit reduces the overall performance. i.e

- Not all stages can be divided into (equal-length) sub-stages
- Overhead of data buffering between the stages.
- The hazards will be more difficult to resolve.
- More complex hardware required.
- Interrupt handling and testing will be more complicated.

Superscalar vs. Superpipeline



Super pipelined Superscalar Design



This is a new trend of architecture design:

Ex. Pentium Pro(P6): 3-degree superscalar, 12-stage "super pipeline".

Superscalar Design

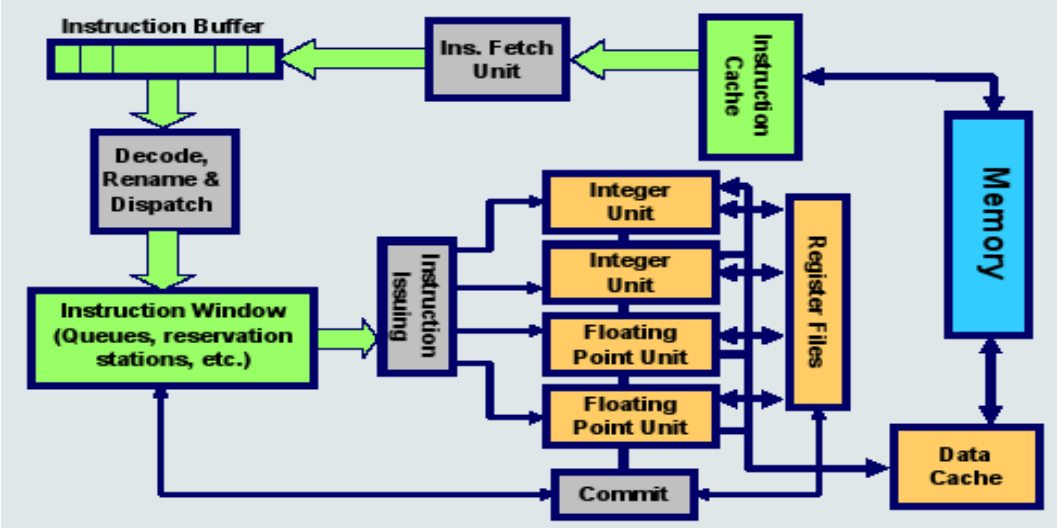
SSA allows several instructions to be issued and completed per clock cycle.

It consists of a number of pipelines that are working in parallel.

Depending on the number and kind of parallel units available, a certain number of instructions can be executed in parallel.

Example, two floating point and two integer operations can be issued and executed simultaneously.

Each unit is also pipelined and can execute several operations in different pipeline stages.



Parallel Execution Limitation

The situations which prevent instructions to be executed in parallel by SSA are mainly due to:

- Resource conflicts.
- Control (procedural) dependency.
- Data dependencies.

They are very similar to those which prevent efficient execution on a pipelined architecture (pipeline hazards):

Their consequences on SSA are more severe than those on simple pipelines, because the potential of parallelism in SSA is greater and, thus, a larger amount of performance will be lost.

Resource Conflicts

Several instructions compete for the same hardware resource at the same time.

e.g., two arithmetic instructions need the same floating-point unit for execution. similar to structural hazards in pipeline.

They can be solved partly by introducing several hardware units for the same functions. e.g., have two floating-point units.

The hardware units can also be pipelined to support several operations at the same time. However, memory units can't be duplicated.

Procedural Dependency

- The presence of branches creates major problems in assuring the optimal parallelism. It cannot execute instructions after a branch in parallel with instructions before a branch. It is similar to control hazards in pipeline.
- If instructions are of variable length, they cannot be fetched and issued in parallel, since an instruction has to be decoded in order to identify the following one therefore, superscalar techniques are more efficiently applicable to RISCs, with fixed instruction length and format.

Data Conflicts

- It is caused by data dependencies between instructions in the program which is similar to date hazards in pipeline.
- To address the problem and to increase the degree of parallel execution, SSA provides a great liberty in the order in which instructions are issued and executed.
- Therefore, data dependencies have to be considered and dealt with much more carefully.

Window of Execution

- Due to data dependencies, only some part of the instructions is potential subjects for parallel execution. In order to find instructions to be issued in parallel, the processor has to select from a sufficiently large instruction sequence.
- There are usually a lot of data dependencies in a short instruction sequence.
- Window of execution is defined as the set of instructions that is considered for execution at a certain moment. The number of instructions in the window should be as large as possible. However, it is limited by:
    - Capacity to fetch instructions at a high rate. The problem of branches.
    - The cost of hardware needed to analyze data dependencies.
- The window of execution can be extended over basic block borders by branch prediction called Speculative execution.
- With speculative execution, instructions of the predicted path are entered into the window of execution.

- Instructions from the predicted path are executed tentatively. If the prediction turns out to be correct the state change produced by these instructions will become permanent and visible (the instructions commit), otherwise, all effects are removed.

Data Dependencies

- All instructions in the window of execution may begin execution, subject to data dependence and resource constraints.

True Data Dependency

- True data dependencies exist when the output of one instruction is required as an input to a subsequent instruction: can fetch and decode second instruction in parallel with first. can NOT execute second instruction until first is finished.

```
MUL  R4,R3,R1    (R4  := R3  * R1)
. . .
ADD  R2,R4,R5    (R2  := R4  + R5)
```

- They are intrinsic features of a program, and cannot be eliminated by compiler or hardware techniques. They have to be detected and handled by hardware.
- In the above example, the simplest solution is to stall the adder until the multiplier has completed. In order to avoid the adder to be idle, the hardware can find other instructions which can be executed by the adder.

Output Dependency

- An output dependency exists if two instructions are writing into the same location. If the second instruction writes before the first one, an error occurs:

```
MUL  R4,R3,R1    (R4 := R3 * R1)
. . .
ADD  R4,R2,R5    (R4 := R2 + R5)
        L2 move   r3,r7
           load   r8,(r3)
           add    r3,r3,#4
           load   r9,(r3)
           ble    r8,r9,L3
```

Anti-dependency

- Anti-dependency exists if an instruction uses a location as an operand while a following one is writing into that location.
- If the first one is still using the location when the second one writes into it, an error occurs:

```
MUL  R4,R3,R1     (R4 := R3 * R1)
. . .
ADD  R3,R2,R5     (R3 := R2 + R5)

        L2 move   r3,r7
           load   r8,(r3)
           add    r3,r3,#4
           load   r9,(r3)
           ble    r8,r9,L3
```

Output and Anti- Dependencies

- Output dependencies and anti-dependencies are not intrinsic features of the executed program. They are not real data dependencies but storage conflicts.
- They are due to the competition of several instructions for the same register and sometimes they are only the consequence of the manner in which the programmer or the compiler is using registers (or memory locations).
- Output dependencies and anti-dependencies can usually be eliminated by using additional registers. This technique is called **register renaming.**

Instruction vs Machine Parallelism

- **Instruction-level parallelism (ILP)** is the average number of instructions in a program that a processor might be able to execute at the same time. It is determined by the number of true dependencies and procedural (control) dependencies in relation to the number of other instructions.
- **Machine parallelism** of a processor is the ability of the processor to take advantage of the ILP of the program. It is determined by the number of instructions that can be fetched and executed at the same time, i.e., the capacity of the hardware.
- To achieve high performance, we need both ILP and machine parallelism. Ideally, we have the same ILP and machine parallelism. However, this is impossible, since the same computer is used for programs with different ILPs.

Division and Decoupling

- To increase ILP, we should divide the instruction execution into smaller tasks and decouple them. In particular, we have three important activities:

    - Instruction issue an instruction is initiated and starts execution.
    - Instruction completion an instruction has competed its specified operations.
    - Instructions commit the operation results are written back to the register files or cache.

SSA Instruction Execution Policies

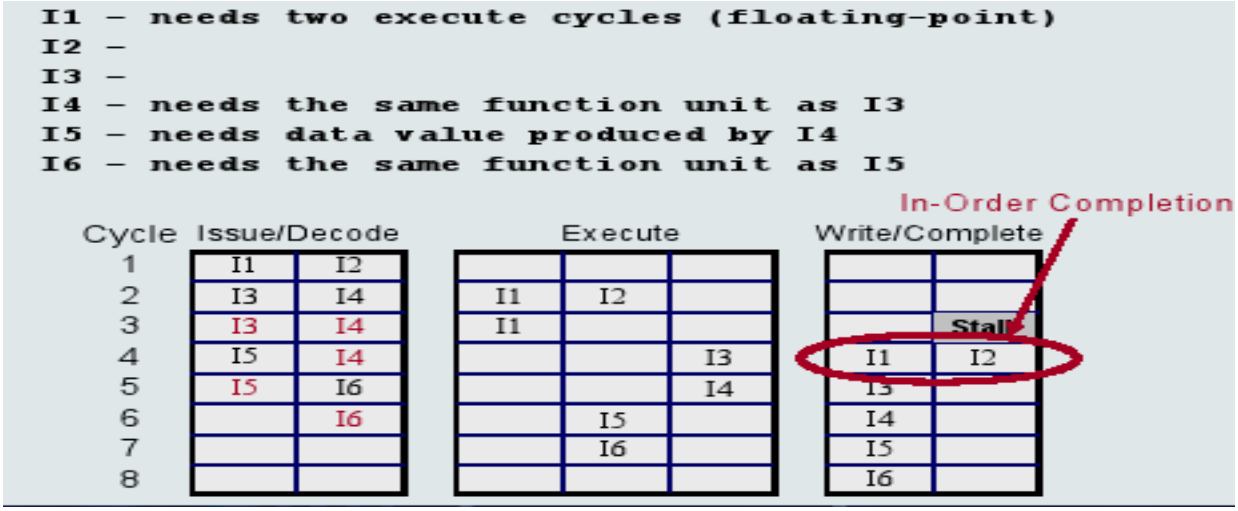
- Instructions can be executed in an order different from the strictly sequential one, with the requirement that the results must be the same.
- Execution policies usually used are :
    - In-order issue with in-order completion.
    - In-order issue with out-of-order completion.
    - Out-of-order issue with out-of-order completion

In-Order Issue with In-Order Completion

- Instructions are issued in exact program order, and completed in the same order (with parallel issue and completion) .An instruction cannot be issued before the previous one has been issued; An instruction cannot be completed before the previous one has been completed.
- To guarantee in-order completion, an instruction will stall when there is a conflict and when a unit requires more than one cycle to execute.

An example:

- A processor can issue and decode 2 instructions per cycle, has 3 functional units (2 single- cycle integer units, and 1 two-cycle floating-point unit), and can write back 2 results per cycle.



- The processor detects and handles true data dependencies and resource conflicts by stalling.SSA does not rely on compiler-based technique
- SSA allows hardware alone to detect instructions which can be executed in parallel and to do so accordingly. IOI with IOC is not very efficient, but it simplifies the hardware.

Out of Order Issue with Out of Order Completion

- With in-order issue, no new instruction can be issued when the processor has detected a conflict, and is stalled until after the conflict has been resolved. The

processor is not allowed to look ahead for further instructions, which could be executed in parallel with the current ones.

- Out-of-order issue takes a set of decoded instructions, issues any instruction, in any order, as long as the program execution is correct. Decouple decode pipeline from execution pipeline, by introducing an instruction window.
- When a functional unit becomes available an instruction can be executed. Since instructions have been decoded, processor can look ahead.

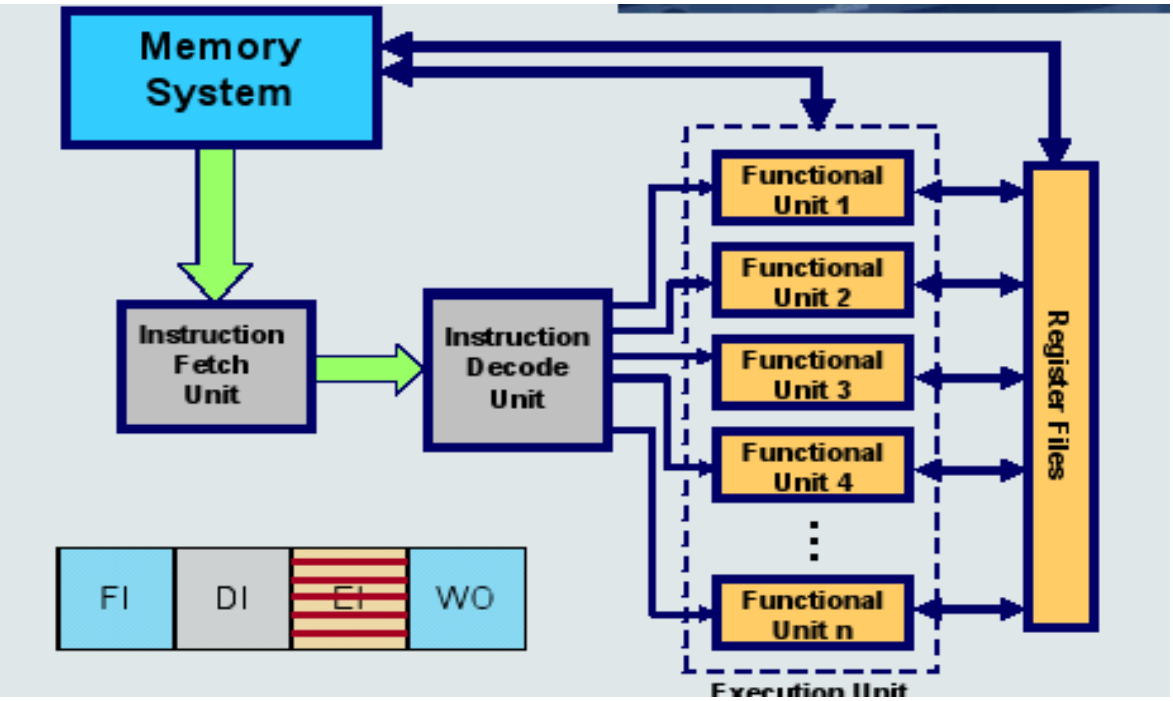
Lecture 5: VLIW Processors

In a VLIW (also called Very Large Instruction Word) processor, several operations that can be executed in parallel are placed on a single instruction word.

Instruction 1	op1	op2	op3	op4
Instruction 2	op1	Ø	op3	op4
Instruction 3	Ø	op2	op3	Ø

VLIW architectures rely on compile-time detection of parallelism. The compiler analyzes the program and detects operations to be executed in parallel.
After one instruction has been fetched all the corresponding operations are issued in parallel.
No hardware is needed for run-time detection of parallelism.
The instruction window problem disappears: the compiler can potentially analyze the whole program to detect parallel operations.

Typical Organization

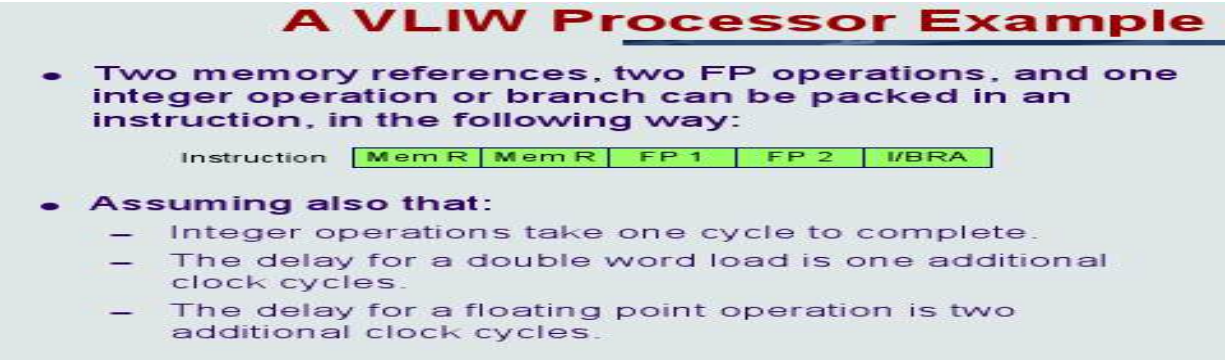


Explicit Parallelism
It is Instruction parallelism scheduled at compile time. Included within the machine instructions explicitly.
Processor uses this information to perform parallel execution. The hardware is less complex and the controller is similar to a simple scalar computer.
The number of FUs can be increased without needing additional sophisticated hardware to detect parallelism, as in SSA and the compiler has much more time to determine possible parallel operations.
The analysis is only done once off-line, while run-time detection is carried out by SSA hardware for each execution of the code. Good compilers can detect parallelism based on global analysis of the whole program.

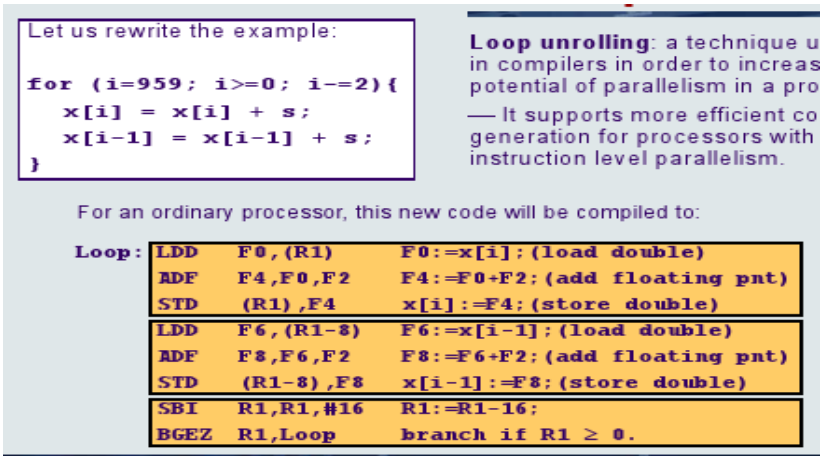
Main Issues

- A large number of registers is needed in order to keep all FUs active (to store operands and results).
- Large data transport capacity is needed between FUs and the register files and between register files and memory.
- High bandwidth between instruction cache and fetch unit is also needed due to long instructions.
- Ex: each instruction with 8 operations, each 24 bits = 192 bits/instruction.
- If a new version of the processor introduces additional FUs, the number of operations to execute in parallel is increased. Therefore, the instruction word changes, and old binary code cannot be run on the new processor.
- Insufficient parallelism in the program to utilize the large degree of machine parallelism causes the hardware resource wastage. A technique to address this problem is loop unrolling, which can be performed by a compiler.

Loop unrolling: It is a technique used in compilers in order to increase the potential of parallelism in a program. It supports more efficient code generation for processors with instruction level parallelism.



For an ordinary processor , the code will look like this:



- Loop Unrolling iteration

1	LDD F0,(R1)	LDD F6,(R1-8)			
2					
3			ADF F4,F0,F2	ADF F8,F6,F2	
4					
5					SBI R1,R1,#16
6	STD(R1+16),F4	STD(R1+8),F8			BGEZ R1,Loop

- Given a certain set of resources (processor architecture) and a given loop, there is a limit on how many iterations should be unrolled. Beyond that limit there is no gain in performance any more.
- Loop unrolling increases the memory space needed to store the machine code.

The IA-64 Architecture

IA-64 is not a pure VLIW architecture, but many of its features are typical for VLIW processors. Which includes:

- Instruction-level parallelism fixed at compile-time. (Very) long instruction word (128 bits).
- Speculative loading.
- Itanium is an EPIC (Explicitly Parallel Instruction Computing) processor, because the parallelism of operations is explicitly specified in the instruction word.
- Registers (both integer and floating point) are 64-bit.
- Predicate registers are 1-bit.
- 8 or more functional units.

Superscalar vs IA-64

Superscalar	IA-64 (improved VLIW)
Multiple parallel execution units	Multiple parallel execution units
RISC-line instructions, one per word	RISC-line instructions, bundled into group of three
Reorder and optimize instruction stream at run time	Reorder and optimize instruction stream at compiler time
Branch prediction with speculative execution of one path	Speculative execution along both paths of a branch
Load data from memory (caches) only when needed.	Speculatively load data before it is needed (caches or memory).

Parallel Processing

Performance Improvement

Reduction of instruction execution time:

Increased clock frequency by fast circuit technology. Simplify instructions (RISC).

Parallelism within processor:

Pipelining.

Parallel execution of instructions (ILP):

- Superscalar processors.
- VLIW architectures.

Parallel processing: Huge degree of parallelism possible.

Why Parallel Processing?

1. Traditional computers are not able to meet high-performance requirements for many applications: e.g
   - Simulation of large complex systems in physics, economy, biology...
   - Distributed data base with search function. Computer-aided design.
   - Visualization and multimedia.
   - Multi-tasking and multi-user systems (e.g., super computers).

Such applications are characterized by a very large amount of numerical computations and/or a high quantity of input data. In order to deliver sufficient performance for such applications, we can have many processors in a single computer.

2. Technology development: Hardware and silicon technology makes it possible to build machines with huge degree of parallelism cost effectively. It started with mainframes and super computers. Now even file servers and regular PCs are implemented often as parallel machines.

3. PP has also the potential of being more reliable: If one processor fails, the system continues to work at a slightly lower performance.

4. PP provides also a platform to build scalable systems with different performances and capabilities.

Parallel Computer

Parallel computers refer to architectures in which many CPUs are running in parallel to implement a certain application or a set of applications.

Such computers can be organized in different ways, depending on several key parameters:

- Number and complexity of individual CPUs;
- Availability of common (shared) memory;
- Interconnection technology and topology;
- Performance of interconnection network;

Parallel Program

In order to fully utilize a parallel computer, one must decompose a problem into sub-problems that can be solved in parallel. The results of sub-problems may have to be combined to get the final result of the main problem.

- Due to data dependency among the sub-problems, it is not easy to decompose some problem to get a large degree of parallelism.

- Due to data dependency, the processors may also have to communicate among each other.

The time taken for communication is usually very high when compared

with the processing time. The communication mechanism must therefore be very well designed in order to get a good performance.

Parallel Program Example

- Matrix computations:

$$C = A + B = \begin{pmatrix} A_{11}+B_{11} & A_{12}+B_{12} & A_{13}+B_{13} & \cdots & A_{1M}+B_{1M} \\ A_{21}+B_{21} & A_{22}+B_{22} & A_{23}+B_{23} & \cdots & A_{2M}+B_{2M} \\ A_{31}+B_{31} & A_{32}+B_{32} & A_{33}+B_{33} & \cdots & A_{3M}+B_{3M} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A_{N1}+B_{N1} & A_{N2}+B_{N2} & A_{N3}+B_{N3} & \cdots & A_{NM}+B_{NM} \end{pmatrix}$$

Vector computation with vector of *m* elements:

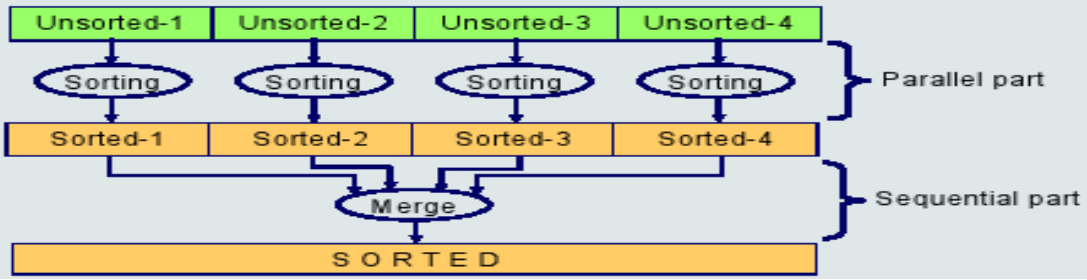
```
for i:=1 to n do
   C[i,1:m]:=A[i,1:m] + B[i,1:m];
end for;
```

Example 2

- A vector dot product is common in filtering:

$$Y = \sum_{i=1}^{N} a(i) \cdot x(i)$$

- Parallel sorting:



Flynn's Classification of Architectures

Based on the nature of the instruction flow executed by the computer and the data flow on which the instructions operate.

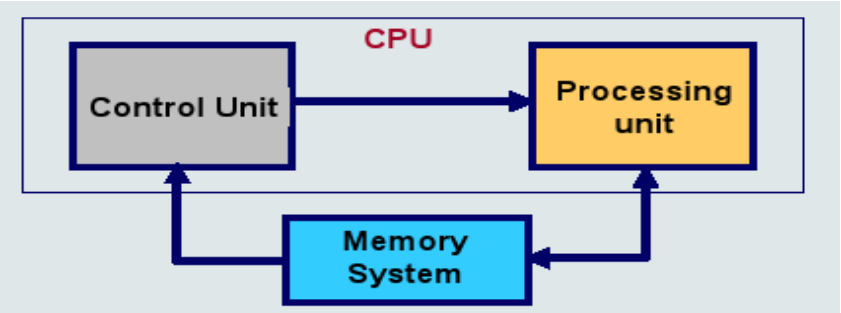
The multiplicity of instruction stream and data stream gives us four different classes:

- Single instruction, single data stream - SISD Single
- Single instruction, multiple data stream - SIMD
- Multiple instruction, single data stream - MISD Multiple
- Multiple instruction, multiple data stream- MIMD

Single Instruction, Single Data - SISD

The regular computers we have discussed up till now:

- A single processor;
- A single instruction stream; and Data stored in a single memory.



Single Instruction, Multiple Data - SIMD

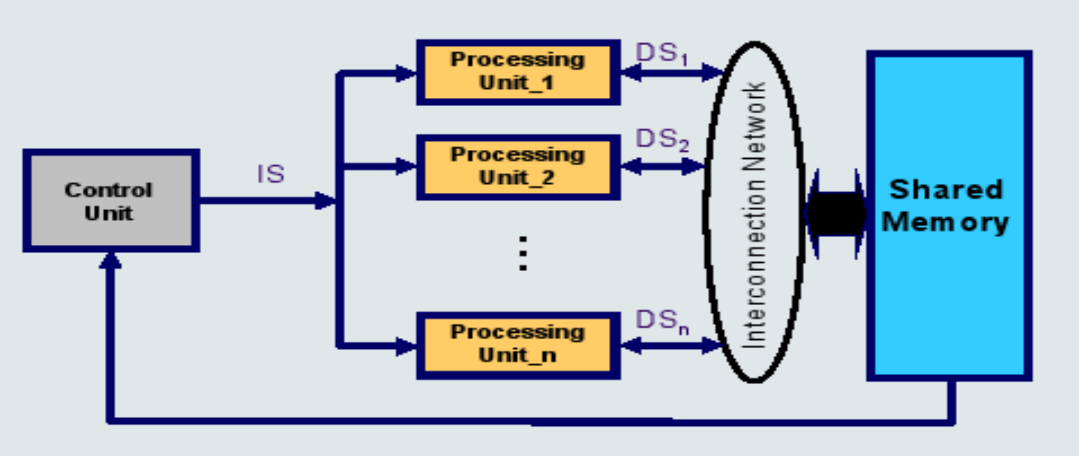
A single machine instruction stream and Simultaneous execution on different sets of data. A large number of processing elements with Lockstep synchronization among the process elements.

The processing elements can:

- have their respective private data memory; or share a common memory via an interconnection network.

Array and vector processors are the most common examples of SIMD machines.

SIMD with shared memory



Multiple Instruction, Single Data - MISD

A single sequence of data. Transmitted to a set of processors.

Processors executes different instruction sequences.

*Not been commercially implemented up till now!

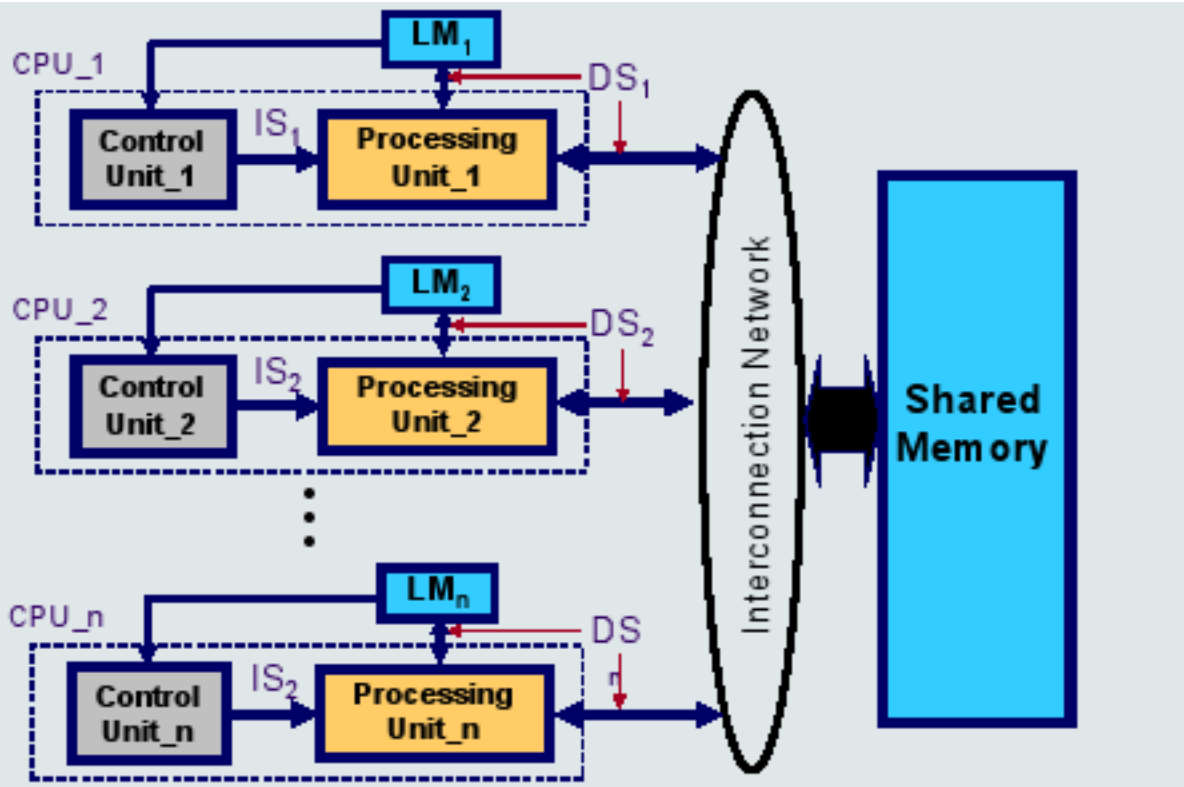
Multiple Instruction, Multiple Data - MIMD

It consists of a set of processors which Simultaneously execute different instruction sequences on different sets of data

The MIMD class can be further divided:

- Shared memory (tightly coupled):
- Symmetric multiprocessor (SMP)
- Non-uniform memory access (NUMA)
- Distributed memory (loosely coupled) = Clusters

MIMD with Shared Memory



Performance Metrics

How fast does a parallel computer run at its maximal potential?

Peak rate: the maximal computation rate that can be theoretically achieved when all processors are fully utilized. Ex. The fastest supercomputer in the world has a peak rate of 55 PFlop/s. use by vendors

How fast execution can we expect from a parallel computer for a given application or a given set of applications? Note the increase of multi-tasking and multi-thread computing.

Speedup: measures the gain we get by using a parallel computer, over a sequential one, to run a given application.

$$S = \frac{T_s}{T_p}$$

T_s: execution time needed with the sequential computer;
T_p : execution time needed with the parallel computer.

Efficiency: to relate speedup to the number of processors used; it provides therefore a measure of the efficiency with which the processors are used.

$$E = \frac{S}{P}$$

S: speedup;
P: number of processors.

For the ideal situation, *in theory*:
$S = P$; which means $E = 1$.

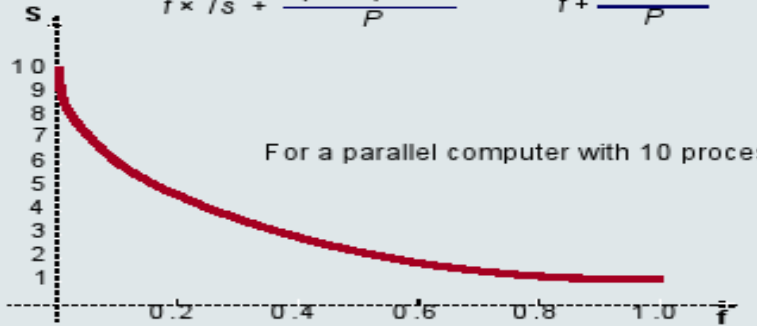
Practically the ideal efficiency of 1 cannot be achieved!

Amdahl's Law
Let f be the ratio of computations that, according to the algorithm, have to be executed sequentially (0 f 1); and P the number of processors.

- Let f be the ratio of computations that, according to the algorithm, have to be executed sequentially ($0 \le f \le 1$); and P the number of processors.

$$T_p = f \times T_s + \frac{(1-f) \times T_s}{P}$$

$$S = \frac{T_s}{f \times T_s + \frac{(1-f) \times T_s}{P}} = \frac{1}{f + \frac{(1-f)}{P}}$$



- Amdahl's law says that even a little ratio of sequential computation imposes a limit on the speedup.
  - A higher speedup than 1/*f* can't be achieved, regardless of the number of processors, since

$$S = \frac{1}{f + \frac{(1-f)}{P}} \leq \frac{1}{f}$$

If there is 20% sequential computation, the speedup will maximally be 5, even If you have 1 million processors.

- To efficiently exploit a high number of processors, *f* must be small (the algorithm has to be highly parallel), since

$$E = \frac{S}{P} = \frac{1}{f \times (P-1) + 1}$$

Other Factors that Limit Speedup

Beside the intrinsic sequentiality of parts of an algorithm, there are also other factors that limit the achievable speedup:

- communication cost;
- load balancing of the processors;
- costs of creating and scheduling processes; and I/O operations (mostly sequential in nature).

There are many algorithms with a high degree of parallelism.The value of f is very small and can be ignored, and they are suited for massively parallel systems but other limiting factors, such as the cost of communications, become critical, in such algorithms.

Interconnection Network

Interconnection network (IN) is a key component of a parallel architecture. It has a decisive influence on:

- the overall performance; and
- the total cost of the architecture.

The traffic in an IN consists of:

- data transfer; and
- transfer of commands and requests (control information).

The key parameters of an IN are

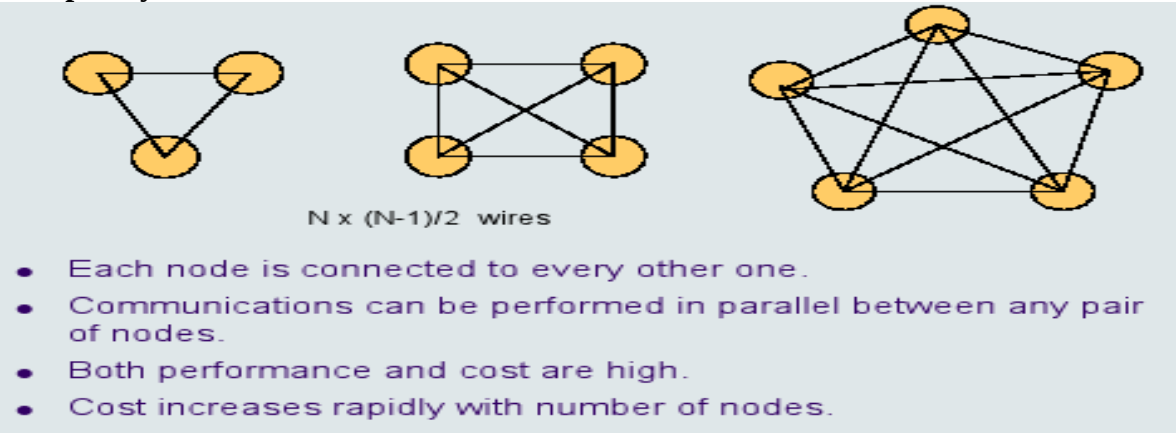
- total bandwidth: transferred bits/second;
- implementation cost.

Single Bus

Single bus networks are simple, cheap and relatively flexible; and you have a broadcast mechanism. One single communication is allowed at a time; the bandwidth is shared by all nodes.

Performance is relatively poor. In order to have good performance, the number of nodes is limited (to around 16 - 20). Multiple buses can be used instead.

Completely Connected Network

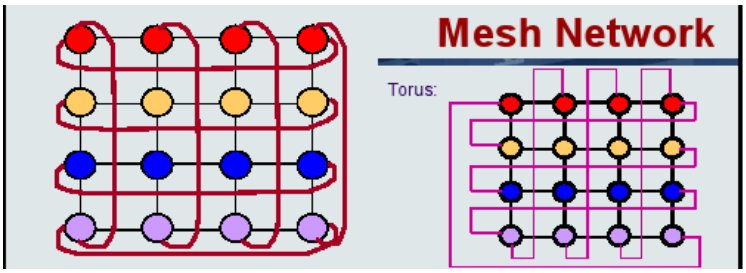


Crossbar Network

- A dynamic network: the interconnection topology can be modified by configurating the switches.
- It is completely connected: any node can be directly connected to any other.
- Fewer interconnections are needed than for the static completely connected network; however, a large number of switches is needed.
- A large number of communications can be performed in parallel (even though one node can receive or send only one data at a time).

Mesh Network

- Cheaper than completely connected networks, while giving relatively good performance.
- In order to transmit data between two nodes, routing through intermediate nodes is needed (maximum 2×(n-1) intermediates for an n×n mesh).
- It is possible to provide wrap-around connections:



Hypercube Network

- 2n nodes are arranged in an n-dimensional cube. Each node is connected to n neighbors.
- In order to transmit data between two nodes, routing through intermediate nodes is needed (maximum n intermediates).

LECTURE 6: MULTIPROCESSORS

INTRO:

Originally, the computer has been viewed as a sequential machine. Most computer programming languages require the programmer to specify algorithms as sequence of instruction.

Processor executes programs by executing machine instructions in a sequence and one at a time.

Each instruction is executed in a sequence of operations (fetch instruction, fetch operands, perform operation store result.)

It is observed that, at the micro operation level, multiple control signals are generated at the same time.

Instruction pipelining, at least to the extent of overlapping fetch and execute operations, has been around for long time.

By looking into these phenomenon, researcher has look into the matter whether some operations can be performed in parallel or not.

As computer technology has evolved, and as the cost of computer hardware has dropped, computer designers have sought more and more opportunities for parallelism, usual to enhance performance and, in some cases, to increase availability.

The taxonomy first introduced by Flynn is still the most common way of categorizing systems with parallel processing capability. Flynn proposed the following categories of computer system:

Single Instruction, Multiple Data (SIMD) system: A single machine instruction controls the simultaneous execution of a number of processing elements on a lockstep basis. Each processing element has an associated data memory, so that each instruction is executed on a different set of data by the different processors. Vector and array processors fall into this category

Multiple Instruction, Single Data (MISD) system : A sequence of data is transmitted to a set of processors, each of which executes a different instruction sequence. This structure has never been implemented.

Multiple Instruction, Multiple Data (MIMD) system: A set of processors simultaneously execute different instruction sequences on different data sets. SMPs, clusters, and NUMA systems fits into this category. With the MIMD organization, the processors are general purpose; each is able to process all of the instructions necessary to perform the appropriate data transformation.

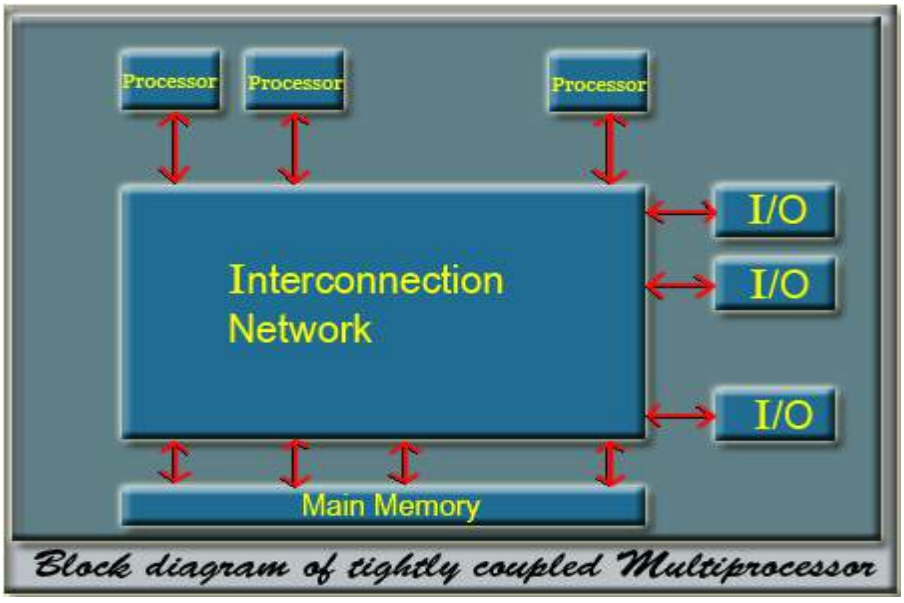
Further MIMD can be subdivided into two main categories:

- Symmetric multiprocessor (SMP): In an SMP, multiple processors share a single memory or a pool of memory by means of a shared bus or other interconnection mechanism. A distinguish feature is that the memory access time to any region of memory is approximately the same for each processor.

- Non uniform memory access (NUMA): The memory access time to different regions of memory may differ for a NUMA processor.

Symmetric Multiprocessors:

A symmetric multiprocessor (SMP) can be defined as a standalone computer system with the following characteristic:
- There are two or more similar processor of comparable capability.
- These processors share the same main memory and I/O facilities and are interconnected by a bus or other internal connection scheme.
- All processors share access to I/O devices, either through the same channels or through different channels that provide paths to the same device.
- All processors can perform the same functions.
- The system is controlled by an integrated operating system that provides interaction between processors and their programs at the job, task, file and data element levels.The operating system of a SMP schedules processors or thread across all of the processors.
SMP has a potential advantages over uniprocessor architecture:
- Performance: A system with multiple processors will perform in a better way than one with a single processor of the same type if the task can be organized in such a manner that some portion of the work done can be done in parallel.
- Availability: Since all the processors can perform the same function in a symmetric multiprocessor, the failure of a single processor does not stop the machine. Instead, the system can continue to function at reduce performance level.
- Incremental growth: A user can enhance the performance of a system by adding an additional processor.
- Scaling: Vendors can offer a range of product with different price and performance characteristics based on number of processors configured in the system.



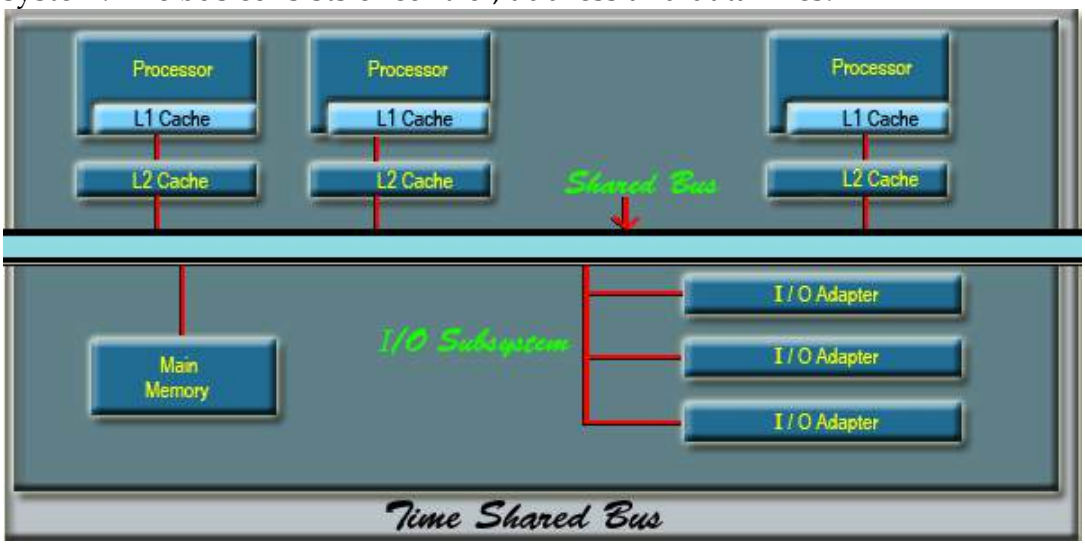
- There are two or more processors. Each processor is self sufficient, including a control unit, ALU, registers and cache.
-Each processor has access to a shared main memory and the I/O devices through an interconnection network.
-The processor can communicate with each other through memory (messages and status information left in common data areas).
- It may also be possible for processors to exchange signal directly.
-The memory is often organized so that multiple simultaneous accesses to separate blocks of memory are possible.
- In some configurations each processor may also have its own private main memory and I/O channels in addition to the shared resources.

The organization of multiprocessor system can be classified as follows:

- Time shared or common bus
- Multiport memory
- Central control unit.

Time shared Bus:

- Time shared bus is the simplest mechanism for constructing a multiprocessor system. The bus consists of control, address and data lines.



The following features are provided in time-shared bus organization:

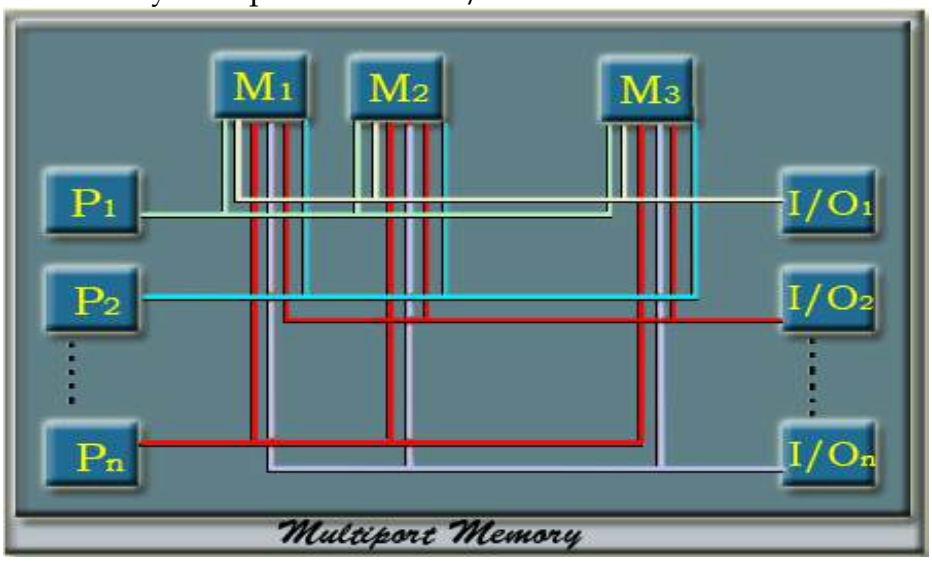
- Addressing: It must be possible to distinguish modules on the bus to determine the source and destination of data
- Arbitration: Any I/O module can temporarily function as "master". A mechanism is provided to arbitrate competing request for bus control, using some sort of priority scheme.
- Time sharing: when one module is controlling the bus, other modules are locked out and if necessary suspend operation until bus access in achieved.

The bus organization has several advantages compared with other approaches:

- Simplicity: This is the simplest approach to multiprocessor organization. The physical interface and the addressing, arbitration and time sharing logic of each processor remain the same as in a single processor system.
- Flexibility: It is generally easy to expand the system by attaching more processor to the bus.
- Reliability: The bus is essentially a passive medium and the failure of any attached device should not cause failure of the whole system.

Multiport Memory:

The multiport memory approach allows the direct, independent access of main memory modules by each processor and I/O module.



The multiport memory approach is more complex than the bus approach, requiring a fair amount of logic to be added to the memory system. Logic associated with memory is required for resolving conflict. The method often used to resolve conflicts is to assign permanently designated priorities to each memory port.

Example is Non-uniform Memory Access (NUMA)

In NUMA architecture, all processors have access to all parts of main memory using loads and stores. The memory access time of a processor differs depending on which region of main memory is accessed.

A NUMA system in which cache coherence is maintained among the cache of the various processors is known as cache-coherence NUMA (CC-NUMA)

LOGIC GATES

70.1. Definition of a Logic Gate

A logic gate is an electronic circuit which makes logic decisions. It has one output and one or more inputs. The output signal appears only for certain combinations of input signals. Logic gates are the basic building blocks from which most of the digital systems are built up. They implement the hardware logic function based on the logical algebra developed by George Boole which is called Boolean algebra in his honour. A unique characteristic of the Boolean algebra is that variables used in it can assume only one of the two values *i.e.* either 0 or 1. Hence, every variable is either a 0 or a 1.

These gates are available today in the form of various IC families. The most popular families are: transistor-transistor logic (*TTL*), emitter-coupled logic (*ECL*), metal-oxide-semiconductor (*MOS*) and complementary metal-oxide-semiconductor (*CMOS*).

In this chapter, we will consider the *OR*, *AND*, *NOT*, *NOR*, *NAND*, exclusive *OR* (*XOR*) and exclusive *NOR* (*XNOR*) gates along with their truth tables.

70.2. Positive and Negative Logic

In computing systems, the number symbols 0 and 1 represent two possible states of a circuit or device. It makes no difference if these two states are referred to as *ON* and *OFF*, *CLOSED* and *OPEN*, *HIGH* and *LOW PLUS* and *MINUS* or *TRUE* and *FALSE* depending on the circumstances. Main point is that they must be symbolized by two opposite conditions.

In positive logic, a 1 represents

1. an *ON* circuit
2. a *CLOSED* switch
3. a *HIGH* voltage
4. a *PLUS* sign
5. a *TRUE* statement

Consequently, a 0 represents

1. an *OFF* circuit 2. an *OPEN* switch 3. a *LOW* voltage 4. a *MINUS* sign 5. a *FALSE* statement.

In negative logic, just opposite conditions prevail.

Suppose, a digital system has two voltage levels of 0V and 5V. If we say that symbol 1 stands for

5V and symbol 0 for 0V, then we have positive logic system. If, on other hand, we decide that a 1 should represent 0 V and 0 should represent 5V, then we will get negative logic system.

Main point is that in *positive logic*, the more positive of the two voltage levels represents the 1 while in negative logic, the more negative voltage represents the 1. Moreover, it is not essential that a 0 has to be represented by 0V although in some cases the two may coincide. Suppose, in a circuit, the two voltage levels are 2V and 10V. Then for positive logic, the 1 represents 10V and the 0 represents 2V (*i.e.* lesser of the two voltages). If the voltage levels are – 2V and – 8V, then, in positive logic, the 1 represents – 2V and the 0 represents – 8V (*i.e.* lesser of the two voltages).

Unless stated otherwise, we will be using only *positive logic* in this chapter.

70.3. The OR Gate

The electronic symbol for a two-input *OR* gate is shown in Fig. 70.1 (*a*) and its equivalent switching circuit in Fig. 70.1 (*b*). The two inputs have been marked as *A* and *B* and the output as *X*. It is worth reminding the reader that as per Boolean algebra, the three variables *A*, *B* and *X* can have only one of the two values *i.e.* either 0 or 1.

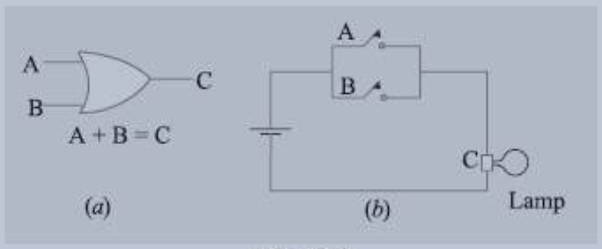


Fig. 70.1

Logic Operation

The *OR* gate has an output of 1 when either A or B or both are 1.

In other words, it is an any-or-all gate because an output occurs when any or all the inputs are present.

As seen from Fig. 70.1 (*b*), the lamp will light up (logic 1) when either switch *A* or *B* or both are closed.

Obviously, the output would be 0 if and only if both its inputs are 0. In terms of the switching conditions, it means that lamp would be *OFF* (logic 0) only when both switches *A* and *B* are *OFF*.

The *OR* gate represents the Boolean equation $A + B = X$

The meaning of this equation is that *X* is true when either *A* is true or *B* is true or both are true. Alternatively, it means that output *X* is 1 when either *A* or *B* or both are 1.

Table 70.1

A	B	X
0	0	1
0	1	0
1	0	0
1	1	1

Fig. 70.2

The above logic operation of the *OR* gate can be summarised with the help of the truth table given in Fig. 70.2. A truth table may be defined as a table which *gives the output state for all possible input combinations*. The *OR* Table 70.1 gives outputs for all possible *A B* inputs of 00, 01, 10 and 11.

We may interpret the truth table as follows:

When both inputs are 0 (switches are *OPEN*), output *X* is 0 (lamp is *OFF*). When *A* is in logic state 0 (switch *A* is *OPEN*) but *B* is in logic state 1 (switch *B* is *CLOSED*), the output *X* is logic state 1 (lamp is *ON*). Lamp would be also *ON* when *A* is *CLOSED* and *B* is *OPEN*. Of course, lamp would be *ON* when both switched are *CLOSED*. It is so because an *OR* gate is equivalent to a parallel *circuit in its logic function*.

Another point worth remembering is that the above *OR* gate is called inclusive *OR* gate because it includes the case when both inputs are true.

70.4. Equivalent Relay Circuit of an OR Gate

In Fig. 70.3, the relay contacts have been wired *in parallel*. When +5V is applied to A, relay K_1 is energised and pulls M down thereby closing the contact. Hence, supply voltage of +5V appears at the output X.

Similarly, when +5V are applied to input B, K_2 is energised and pulls N down thereby bringing X in contact with S. Of course, when both A and B are at +5V, X is at +5V. Incidentally, when inputs at A and B are 0, X is also 0.

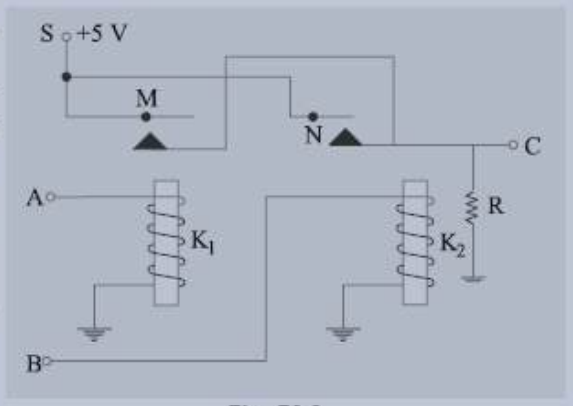


Fig. 70.3

70.5. Diode OR Gate

Fig. 70.4. shows the diode *OR* gate consisting of two ideal diodes D_1 and D_2 connected in parallel across the output X.

1. When A is at +5V, D_1 is forward-biased and hence conducts. The circuit current flows *via* R dropping 5V across it. In this way, point X achieves potential of +5 V.
2. When + 5V is applied to B, D_2 conducts causing point X to go to +5 V.
3. When both A and B are +5V, the drop across R is 5V because voltages of A and B *are in parallel*.

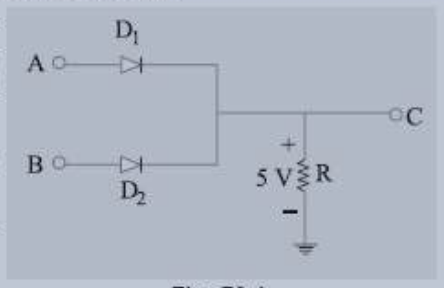


Fig. 70.4

Again, point X is driven to +5 V.

4. Obviously, when there is no voltage either at A or B, output X remains 0.

70.6. Transistor OR Gate

Fig. 70.5 illustrates a possible transistor *OR* gate consisting of three interconnected transistors Q_1, Q_2, and Q_3 supplied from a common supply V_{cc} = +5 V.

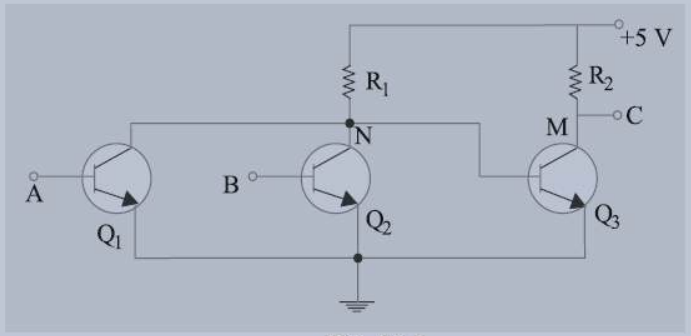


Fig. 70.5

1. When +5 V is applied to A, Q_1 is forward-biased and so it conducts. Assuming that Q_1 is saturated, entire V_{cc} = 5 V drops across R_1 thus causing N to go to ground. This, in turn, cuts off Q_3 thereby causing X to go to V_{cc} *i.e.* + 5V.

2. When +5 V is applied to B, Q_2 conducts thereby driving N to ground *i.e.* 0V. With no forward bias on its base, Q_3 is cut-off thus driving X again to V_{cc} *i.e.* +5 V.

3. If both inputs A and B are grounded, Q_1 and Q_2 are cut-off driving N to +5 V. As a result, Q_3 becomes forward-biased and conducts fully. In that case, entire V_{cc} drops across R_2 driving M and hence X to ground.

70.7. OR Gate Symbolizes Logic Addition

According to Boolean algebra, *OR* gate performs **logical addition**. Its truth table can be written as given below:

$0 + 0 = 0$
$0 + 1 = 1$
$1 + 0 = 1$
$1 + 1 = 1$

It must be clearly understood that '+' sign in Boolean algebra **does not stand for the addition** as understood in the ordinary or numerical algebra. In symbolic logic, the '+' sign indicates **OR** operation whose rules are given above. In logic algebra, $A + B = X$ means that if A is true *OR* B is true, then X will be true. ***It does* not *mean here that sum of A and B equals X.*** The other symbols used for '+' sign are U and V. Hence, the above equation could also be written as $A \, U \, B = X$ or $A \, V \, B = X$.

The meaning of the last three logic additions is that output is 1 when either input A or B or both are 1. The first addition implies that output is 0 ***only when both inputs are 0.***

The meaning of the '+' sign often becomes clear from the context as shown below:

$1 + 1 = 2$ — decimal addition
$1 + 1 = 10$ — binary addition
$1 + 1 = 1$ — *OR* addition

We can put the above *OR* laws in more general terms

$A + 1 = 1$
$A + 0 = A$

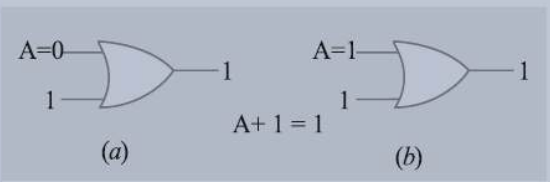


Fig. 70.6

(*i*) A + 1 = 1

As we know, A can have two values: 0 or 1. When A is 0, then we have 0 + 1 = 1 as shown in Fig. 70.6 (*a*).

When $A = 1$, then the above expression becomes : 1 + 1 = 1 as shown in Fig. 70.6 (*b*), Hence, we find that **irrespective of the value of A.**

$A + 1 = 1$

(*ii*) A + 0 = A

If $A = 0$, then $0 + 0 = 0$ *i.e.* output is 0 which is correct and is shown in Fig. 70. 7 (*a*). The output is what the value of A is.

As shown in Fig. 70.7 (*b*), when $A = 1$, output is 1 because 1 + 0 = 1. Again, output is what the value of A is.

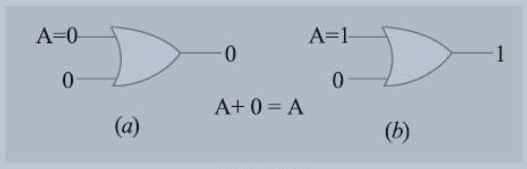


Fig. 70.7

(*iii*) A + A = A

With A set to 0, the output is 0 because $0 + 0 = 0$ as shown in Fig. 70.8 (*a*).

With A set to 1, the output is 1 because 1 + 1 = 1 as shown in Fig. 70.8 (*b*). Obviously, the output in both cases is A.

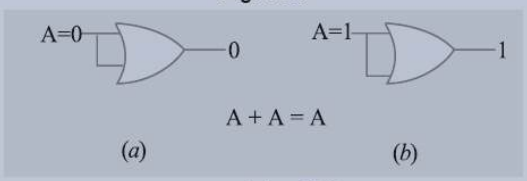


Fig. 70.8

70.8. Three Input OR Gate

The electronic symbol for a 3-input (fan-in of 3) ***inclusive OR*** gate is shown in Fig. 70.9. As is usual in logic algebra, the inputs A, B, C as well as the output X can have only one of the two values *i.e.* 0 or 1.

Truth Table

It is shown in Table 70.2. Following points are worth noting:

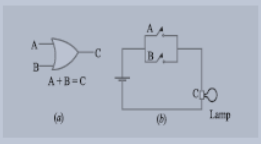


Fig. 70.9

1. The number of rows in the table is $2^3 = 8$ *i.e.* there are eight ways of combining the three inputs. In general, the number of horizontal rows is 2^n where n is the number of inputs.
2. In first column A, logic values alternate between 0 and 1 every four rows twice.
3. The second input column B alternates between 0 and 1 every two rows four times.
4. The third input column C alternates between 0 and 1 every other row eight times.

The truth table symbolizes the Boolean equation $A + B + C = X$ which means that output X is 1 when ***either A or B or C is 1 or all are 1.*** Alternatively, X is true when either A or B or C is true or all are true.

Table No. 70.2

A	B	C	X
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

vvhh

Its electronic symbol is shown in Fig. 70.10 (*a*) and its equivalent switching circuit in Fig. 70.10 (*b*).

In this gate, output is 1 if its either input **but not both**, is 1. In other words, it has an output 1 **when its inputs are different**. The output is 0 only when inputs **are the same**.

To put it a bit differently, this logic gate has output 0 **when inputs are either all 0 or all 1.**

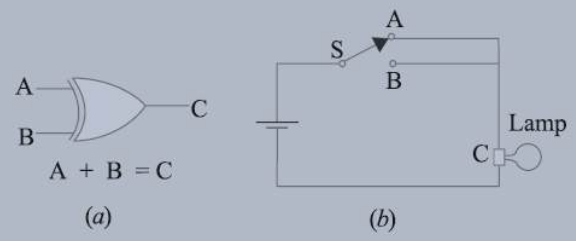


Fig. 70.10

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

Fig. 70.11

This gate works on the Boolean equation $A \oplus B = X$

The circle around plus (+) sign is worth noting.

The circuit is also called an ***inequality comparator*** or detector because it produces an output only **when the two inputs are different**.

Explanation

The ***inclusive OR*** gate exemplifies the everyday usage of the word *OR* which stands for one or the other or both. Take the following statement:

To qualify for a competition, you might have to subscribe to a magaine OR belong to a club.
Obviously, there is no bar on your doing both. But now take **exclusive** statement:

You can be rich OR you can be poor.

Obviously, you cannot be both at the same time.

The change-over switching ***circuit*** of Fig. 70.10 (*b*) simulates the ***exclusive OR* (*XOR*)** gate. Switch positions *A* and *B* will individually light up the lamp but a combination of *A* and *B* is not possible.

The truth table for a 2-input *XOR* gate is given in Table No.70.3. It is instructive to compare it with that for an *inclusive OR* gate (Table 70.1).

70.10. The AND Gate

The electronic (or logic) symbol for a 2-input *AND* gate is shown in Fig. 70.12 (*a*) and its equivalent switching circuit in Fig. 70.12 (*b*). It is worth reminding the readers once again that the three variables *A*, *B*, *C* can have a **value of either 0 or 1**.

Logic Operation

1. The *AND* gate gives an output only **when all its inputs are present**.
2. The *AND* gate has a 1 output when both *A* **and** *B* are 1. Hence, this gate is an **all-or-nothing** gate whose output occurs only when all its inputs are present.
3. In True/False terminology, the output of an *AND* gate will be **true** only if **all its inputs are true**. Its output would be false if **any** ***of its inputs is false.***

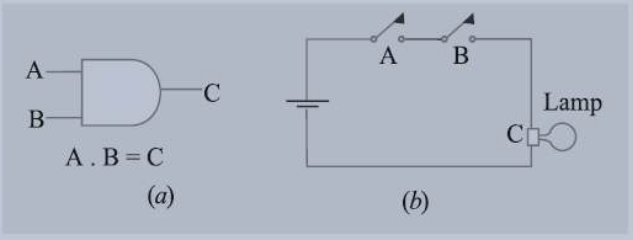


Fig. 70.12

The *AND* gate works on the Boolean algebra

$$A \times B = X \quad \text{or} \quad A \,.\, B = X \quad \text{or} \quad AB = X$$

It is a **logical** multiplication and is different from the **arithmetic** multiplication. Often the sign '×' is replaced by a dot which itself is generally omitted as shown above. The logical meaning of the above equation is that

1. output *X* is 1 only when both *A and B* are 1.
2. output *X* is true only when both *A and B* are true.

Table 70.4

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

Fig. 70.13

Table 70.5

A	B	C	X
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

ABC = X

Fig. 70.14

As seen from Fig. 70.12 (*b*), the lamp would be *ON* when both switches *A* and *B* are closed. Even when one switch is open, the lamp would be *OFF*. Obviously, an *AND* gate is equivalent to a **series switching circuit.**

Truth Table Fig. 70.13 shows truth table for a 2-input *AND* gate and Fig. 70.14 gives the same for a 3-input *AND* gate.

As seen, *X* is at logic 1 only when *all* inputs are at logic 1, not otherwise. The procedure for writing down the first three columns is the same as explained in Art. 70.8 earlier.

70.11. Equivalent Relay Circuit of an AND Gate

The *AND* gate can be physically realized with the help of relay circuit shown in Fig. 70.15. Here the two relay contacts have been **wired in series**.

When +5 V is applied to both input circuits, relays K_1 and K_2 are energized thereby pulling M and N downwards which brings C in contact with the supply point S. Hence, X goes to +5 V.

It is obvious that energizing only ***one relay*** will not make X go to + 5 V.

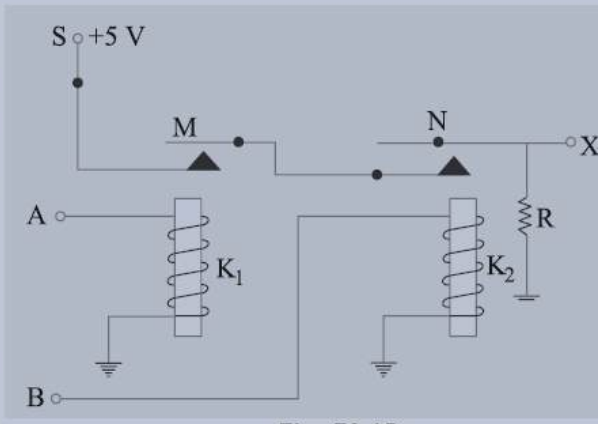


Fig. 70.15

70.12. Diode AND Gate

It is shown in Fig. 70.16. Its logical operation is as under :

1. When A is at 0 V, diode D_1 conducts and the supply voltage of +5 V drops across R. Consequently, point N and hence point X are driven to V. Therefore, the output C is 0.

2. Similarly, when B is at 0 V, D_2 conducts thereby driving N and hence X to ground.

3. Obviously, when both A and B are at 0 V, both diodes conduct and, again, the output X is 0.

4. There is no supply current and hence no drop across R ***only when both A and B are at +5 V.*** Only in that case, the output X goes to supply voltage of +5 V.

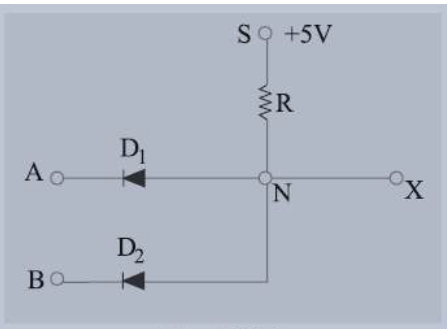


Fig. 70.16

70.13. Transistor AND Circuit

It is shown in Fig. 70.17. When both A and B are at +5 V, the two transistors Q_1 and Q_2 conduct. The current so produced drops the supply voltage of +5 V across R_1 thereby driving point N and hence base of Q_3 to ground or 0V. This cuts off Q_3 so that X goes to supply voltage of +5 V.

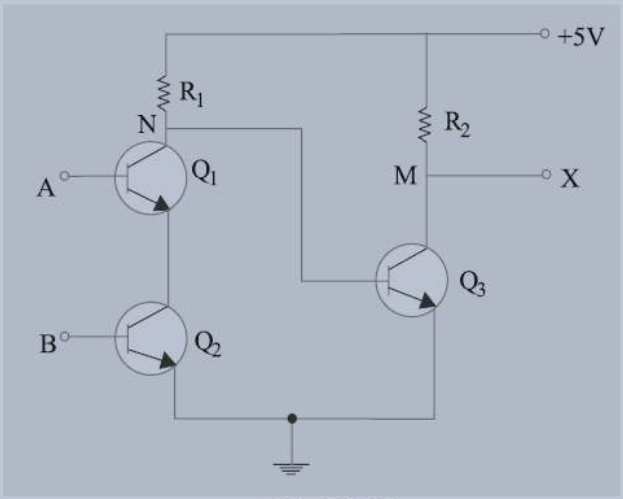


Fig. 70.17

Obivously, there is an output at X only when there is an input at A *and* B.

If either A or B is at 0 V, then Q_1 or Q_2 will be cut off and no drop will take place across R_1. Hence, point N will go to supply voltage of +5V. Consequently, Q_3 will conduct and whole of supply voltage will be dropped across R_2. As a result, point M and hence output X will go to 0 V.

70.14. AND Gate Symbolizes Logic Multiplication

According to Boolean algebra, the *AND* gate performs logical multiplication on its inputs as given below:

$0.0 = 0$
$0.1 = 0$
$1.0 = 0$
$1.1 = 1$

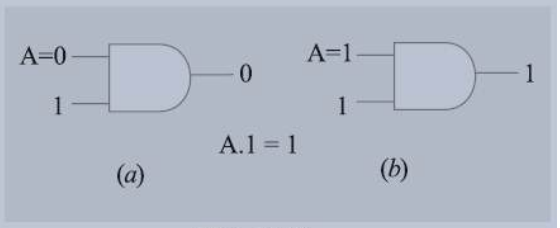


Fig. 70.18

In general, we can put the laws of Boolean multiplication in the following form:

$A.1 = A$ $\qquad A.0 = 0$

$A.A = A$ $\qquad$ — not A^2

The above indentities can be verified by giving values of 0 and 1 to A.

1. **A.1 = A** When $A = 0$

then 0.1 = 0 —Fig. 70.18 (*a*)

When A = 1

then 1.1 = 1 —Fig. 70.18 (*b*)

It is seen that in each case, output has the same value as that of A.

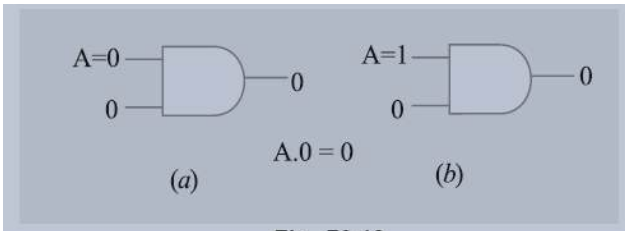


Fig. 70.19

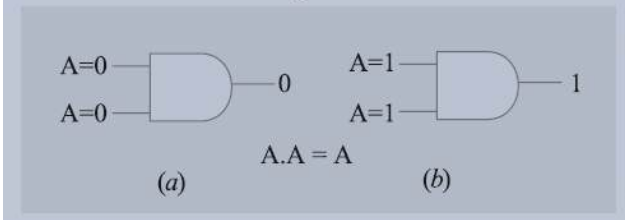


Fig. 70.20

2. **A.0 = 0**

When $A = 0$

then $0.0 = 0$ —Fig. 70.19 (*a*)

When $A = 1$

then $1.0 = 0$ —Fig. 70.19 (*b*)

It is seen that output is always 0 ***whatever the value of A.***

3. **A.A = A**

When $A = 0$, then $0.0 = 0$ — Fig 70.20 (*a*)

When $A = 1$, then $1.1 = 1$ — Fig. 70.20 (*b*)

It is seen that output always ***takes on the value of A.***

70.15. The NOT Gate

It is so called because *its output is NOT the* ***same as its input.*** It is also called an ***inverter*** because it inverts the input signal. It has **one** input and **one** output as shown in Fig. 70.21 (*a*). All it does is to invert (or complement) the input as seen from its truth table of Fig. 70.21 (*b*).

The schematic symbol for inversion is a small circle as shown in Fig. 70.21 (*a*). The logical symbol for inversion or negation or complementation is a bar over the function to indicate the opposite state.

Sometimes, a prime is also used as A'. For ex-

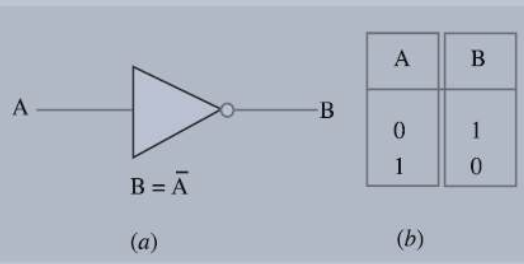


Fig. 70.21

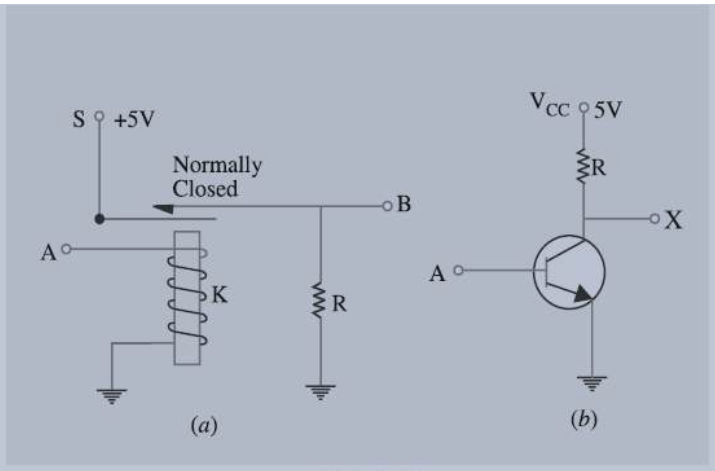


Fig. 70.22

ample, $\overline{A}$ means not-A. Similarly, $\overline{(A + B)}$ means the complement of $(A + B)$.

70.16. Equivalent Circuits for a NOT Gate

The relay circuit of Fig. 70.22 (*a*) is a physical realization of the complementation operation of the Boolean algebra. When +5 V is applied to input A, K is energised and opens the ***normally-closed*** contact thereby driving output X to 0 V. Of course, when A is at 0V, X has the supply voltage of +5 V applied to it because the relay contact is normally closed.

In the equivalent transistor circuit of Fig. 70.22 (*b*) when +5V is applied to A, the transistor will be fully turned *ON*, drawing maximum ***collector*** current. Hence, whole of V_{CC} = 5 V will drop across R thereby sending X to 0 V. With 0 V applied at A, the transistor will be cut *OFF* and the output X, therefore, will go to V_{CC} *i.e.* + 5 V. Obviously, in each case, **output is the opposite of input**.

70.17. The NOT Operation

It is a ***complementation*** operation and its symbol is an **overbar**. It can be defined as under:

As stated earlier, $\overline{0}$ means taking the negation or complement of 0 which is 1.

$$\overline{0} = 1$$

$$\overline{1} = 0$$

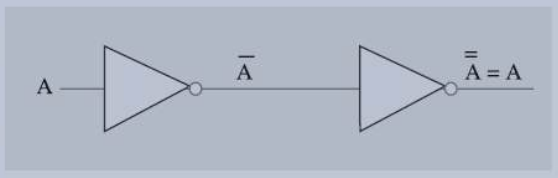


Fig. 70.23

It should also be noted that complement of a value can be taken repeatedly. For example,

$$\overline{\overline{1}} = \overline{0} = 1 \quad \text{or} \quad \overline{\overline{0}} = \overline{1} = 0$$

As seen double complementation gives the original value as shown in Fig. 70.23.

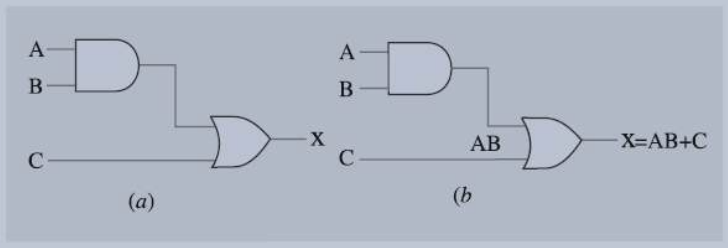
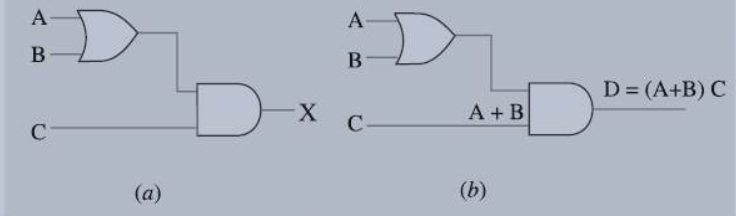


Fig. 70.24

Example 70.1. *Find the Boolean equation for the output X of Fig. 70.24 (a). Evaluate X when (i) A = 0, B = 1, C = 1 (ii) A = 1, B = 1, C = 1.* **[Computer Engg., Pune Univ. 1991]**



(a) (b)

Fig. 70.25

Solution. The output of the *AND* gate is *AB*. It then becomes one of the inputs for the 2 input *OR* gate. When *AB* is ORed with *C*, we get (*AB*+C).

$\therefore \quad X = AB + C$ —Fig. 70.24 (*b*)

(*i*) $X = 0.1 + 1 = 0 + 1 = 1$ —Art 70.7

(*ii*) $X = 1.1 + 1 = 1 + 1 = \mathbf{1}$

Example 70.2. *Find the Boolean expression for the output of Fig. 70.25 (a) and evaluate it when (i) A = 0, B = 1, C = 1, (ii) A = 1, B = 1, C = 0.*

Solution. The output of the *OR* gate is (*A* + *B*). Afterwards, it becomes the input of the *AND* gate. When *AND*ed with *C*, it becomes (*A* + *B*).*C*.

$\therefore \quad X = (A+B).C$ —Fig. 70.25 (*b*)

(*i*) $X = (0+1).1 = 1.1 = 1$

(*ii*) $X = (1+1).0 = 1.0 = 0$

Example 70.3. *Find the Boolean expression for the output of Fig. 70.26 and compute its value when A = B = C = 1 and X = 0.*

(Digital Computations, Punjab Univ. 1990)

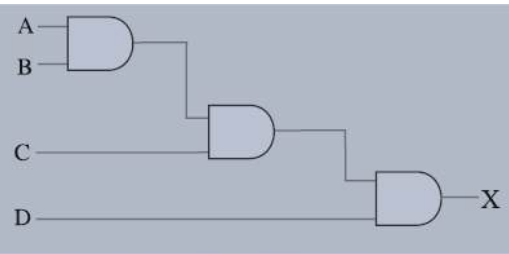


Fig. 70.26

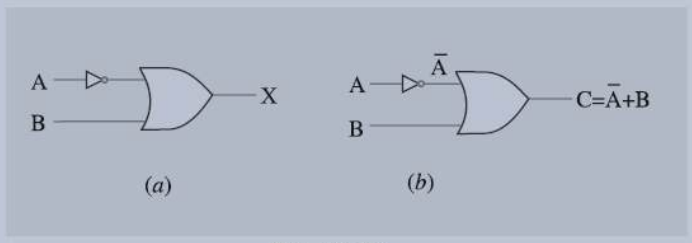


Fig. 70.27

Solution. The circuit is made up of three *AND* gates. Obviously, it is equivalent to a single 4-input *AND* gate *i.e.* an *AND* gate with a fan-in of four.

Output of the first gate is AB, that of the second is ABC and that of the third is $ABCD$. Hence, final output is $X = ABCD$.

Substituting the given values, we get

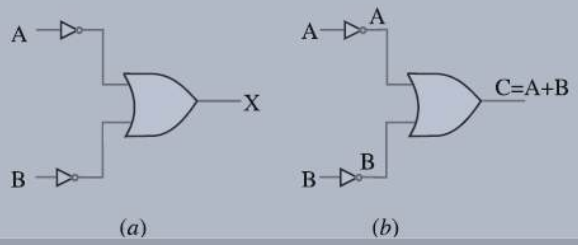
$X = 1.1.1.0 = 1.1.0 = 1.0 = 0$

Example 70.4. *Find the Boolean expression for the output X of Fig. 70.27 (a) and compute its value when*

(*i*) $A = 0, B = 1$

(*ii*) $A = 1, B = 0$

Solution. As seen, one of the inputs to the *OR* gate is inverted *i.e.* A becomes $\bar{A}$ as shown in Fig. 70.27 (*b*). Hence, output is given by $X = \bar{A} + B$



(*i*) $X = \bar{0} + 1 = 1 + 1 = \mathbf{1}$ (*ii*) $X = \bar{1} + 0 = 0 + 0 = \mathbf{0}$

Example 70.5. *What is the Boolean expression for the output X of Fig. 70.28 (a) ? Compute the value of X when*

(i) A = 0, B = 0 (ii) A = 1, B = 1

Solution. As seen, in this case, both inputs to the OR gate have been inverted. Hence as shown in Fig. 70. 28 (*b*), the inputs become $\bar{A}$ and $\bar{B}$. Therefore, Boolean expression for the output becomes $X = \bar{A} + \bar{B}$.

(*i*) $X = \bar{0} + \bar{0} = 1 + 1 = \mathbf{1}$

(*ii*) $X = \bar{1} + \bar{1} = 0 + 0 = \mathbf{0}$

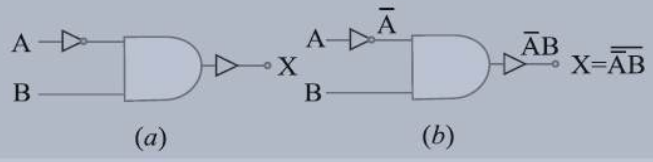
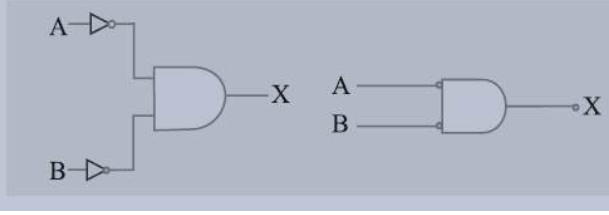


Fig. 70.29



Example 70.6. *Write down the Boolean equation for the output X of Fig. 70.29 (a). Compute its value when*

(*i*) $A = 0, B = 0$

(*ii*) $A = 0, B = 1$

(*iii*) $A = 1, B = 0$

(*iv*) $A = 1, B = 1$

Fig. 70.30

Solution. As seen, inputs to the *AND* gate are A and B [Fig. 70.29 (*a*)]. The output of the *AND* gate is $A.B$. However, this output is inverted by the second inverter connected in the output. Hence, final output equation is

$$X = \overline{A}\,B$$

(*i*) $X = \overline{\overline{0}.0} = \overline{1.0} = \overline{0} = \mathbf{1}$

(*ii*) $X = \overline{\overline{0}.1} = \overline{1.1} = \overline{1} = \mathbf{0}$

(*iii*) $X = \overline{\overline{1}.0} = \overline{0.0} = \overline{0} = \mathbf{1}$

(*iv*) $X = \overline{\overline{1}.1} = \overline{0.1} = \overline{0} = \mathbf{1}$

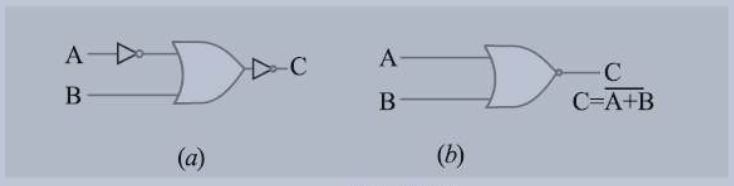


Fig. 70.31

While evaluating expressions of the above type, you must remember the following two points :

1. take the *NOT i.e.* inversion of the individual term first.
2. When a *NOT* or ***inversion*** **is** applied to more than one term (like 1.0), you should work out the *OR* (or *AND*) operation first and then take the *NOT* of the result so obtained.

70.18. Bubbled Gates

A bubbled gate is one whose inputs are *NOT*ed or inverted *i.e.* it is a negated gate. Fig. 70.30 (*a*) shows an *AND* gate whose both input are inverted.

In practice, instead of this logic symbol, the one shown in Fig. 70.30 (*b*) is widely used. As seen, the inverter triangles have been eliminated and the two small circles or bubbles have been moved to the inputs of the gate. Such a gate is called a ***bubbled AND*** gate, the bubbles acting as a reminder of the inversion or complementation that takes place before *AND*ing the inputs.

It would be shown later that a bubbled *AND* gate is equivalent to a *NOR* gate.

Table 70.6

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

Fig. 70.32

Similarly, a bubbled *OR* gate is equivalent to a *AND* gate.

70.19. The NOR Gate

In fact, it is a *NOT-OR* gate. It can be made out of an *OR* gate by connecting an inverter in its output as shown in Fig. 70.31 (*a*).

The output equation is given by

$$X = \overline{A + B}$$

A *NOR* function is just the reverse of the *OR* function.

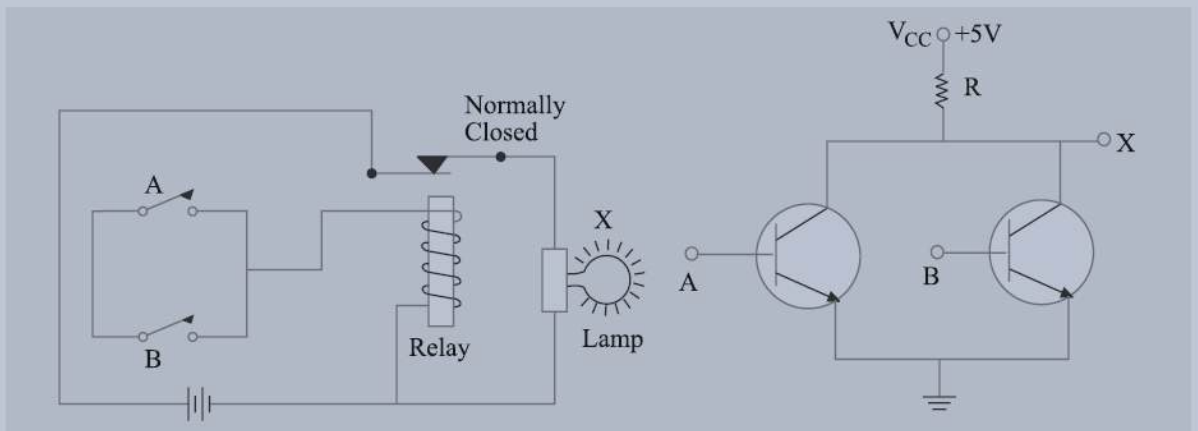


Fig. 70.33 Fig. 70.34

Logic Operation

A *NOR* gate will have an output of 1 **only when all its inputs are 0**. Obviously, if any input is **1**, the output will be 0. Alternatively, in a *NOR* gate, output is **true** only **when all inputs are false**.

The truth table for a 2-input *NOR* gate is shown in Fig. 70.32. It will be observed that the output X is just the reverse of that shown in Fig. 70.2.

The equivalent relay circuit for a *NOR* gate is shown in Fig. 70.33.

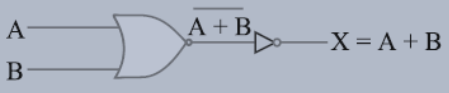


Fig. 70.35

It is seen that the lamp glows under 00 input condition only but not under 01, 10, 11 input conditions.

The transistor equivalent of the *NOR* gate is shown in Fig. 70.34. As seen, output X is 1 only when both transistors are cut-off *i.e.* when $A = 0$ and $B = 0$. For any other input condition like 01, 10 and 11, one or both transistors saturate forcing point X to go to ground.

70.20. NOR Gate is a Universal Gate

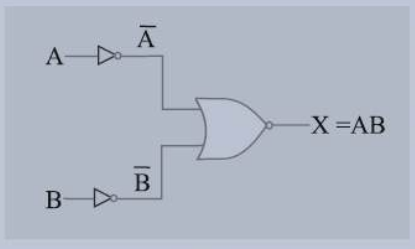


Fig. 70.36

It is interesting to note that a *NOR* gate can be used to realize the basic logic functions : *OR*, *AND* and *NOT*. That is why it is often referred to as a ***universal*** gate. Examples are given below:

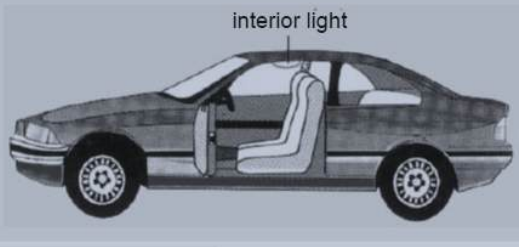
1. As OR Gate

As shown in Fig. 70.35, the output from *NOR* gate is $A + B$. By using another inverter in the output, the final output is inverted and is given by $X = A + B$ which is the logic function of a normal *OR* gate.

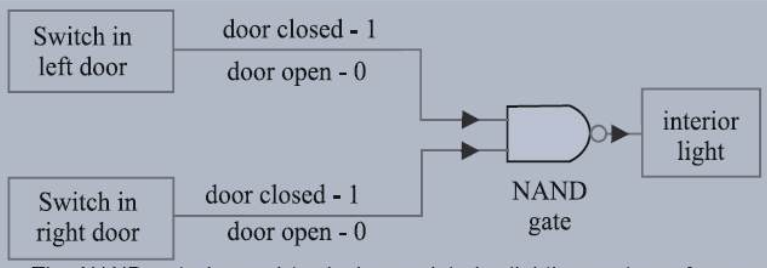
2. As AND Gate

Here, two inverters have been used, one for each input (Fig. 70.36). The inputs have, thus, been **inverted before they are applied** to the *NOR* gate. The output is $\overline{A} + \overline{B}$ which can be proved (with the help of De Morgan's theorem) to be equal to AB.

Incidentally, we could have used a bubbled *NOR* gate for the above purpose.



3. As NOT Gate



The NAND gate is used to design an interior lighting system of a car such that the light is switched off only when both doors are shut

The two inputs have been tied together as shown in Fig. 70.37 (*a*). The output is $\overline{A} + \overline{A}$ which can be proved to be equal to A with the help of De Morgan's theorem. Instead of the first symbol, the second symbol shown in Fig. 70.37 (*b*) is widely used where only single input has been used.

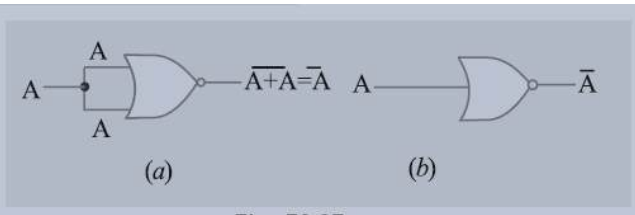


Fig. 70.37

Here is a different way of making *OR* and *AND* gates. Fig. 70.38 (*a*) shows how we can use *NOR* gates to produce an *OR* gate. Similarly, Fig. 70.38 (*b*) shows the formation of an *AND* gate from three *NOR* gates. Knowledge of De Morgan's theorem is needed to understand their logic operation.

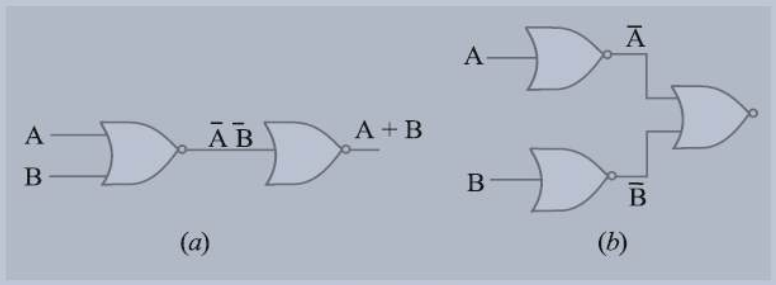


Fig. 70.38

70.21. The NAND Gate

It is, in fact, a *NOT-AND* gate. It can be obtained by connecting a NOT gate in the output of an AND gate as shown in Fig. 70.39. Its output is given by the Boolean equation.

This gate gives an output of 1 if its ***both inputs are*** **not 1**. In other words, it gives an output 1 if **either A or B or both are 0**.

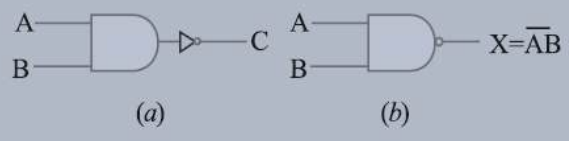


Fig. 70.39

The truth table for a 2-input *NAND* gate is given in Fig. 70.40. It is just the opposite of the truth for *AND* gate. It is so because *NAND* gate performs reverse function of an *AND* gate.

The equivalent relay circuit of a *NAND* gate is shown in Fig. 70.41.

It is seen that lamp glows under input conditions of 00, 01, 10 but not under 11 input condition

A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

Fig. 70.40

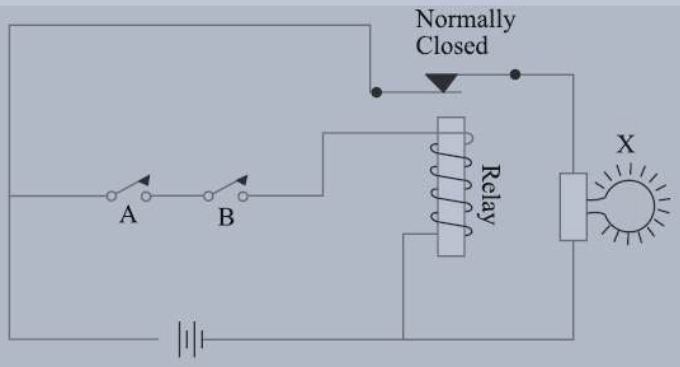


Fig. 70.41

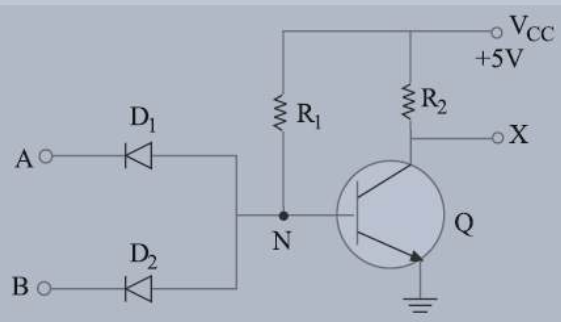


Fig. 70.42

when both switches *A* and *B* are *ON*. The diode-transistor equivalent of a *NAND* gate is shown in Fig. 70.42.

It is seen that point *N* would be driven to ground when either D_1 or D_2 or both D_1 and D_2 conduct. It represents input conditions of 10, 01 and 11*. Under such conditions, *Q* is cut off and hence *X* goes to V_{CC} meaning logic 1 state. Only time *X* is 0 is when $A = 1$ and $B = 1$ (*i.e.* input voltages at *A* and *B* are at +5V) so that *N* is +5 V and *Q* is saturated.

70.22. NAND Gate is a Universal Gate

NAND gate is also called universal gate because it can perform all the three logic functions of an *OR* gate, *AND* gate and inverter as shown below.

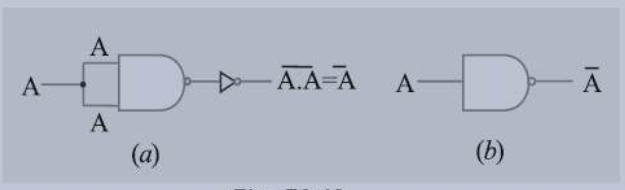


Fig. 70.43

As shown in Fig. 70.43 (*a*), a *NOT* gate can be made out of a *NAND* gate by connecting its two inputs together. When a *NAND* gate is used as a *NOT* gate, the logic symbol of Fig. 70.43 (*b*) is employed instead.

The use of two *NAND* gates to produce an *AND* gate is shown in Fig. 70.44 (*a*).

Similarly, Fig. 70.44 (*b*) shows how *OR* gate can be made out of three *NAND* gates. The *OR* function may not be clear from the figure because we need De Morgan's theorem to prove that $\overline{A}\,\overline{B} = A + B$.

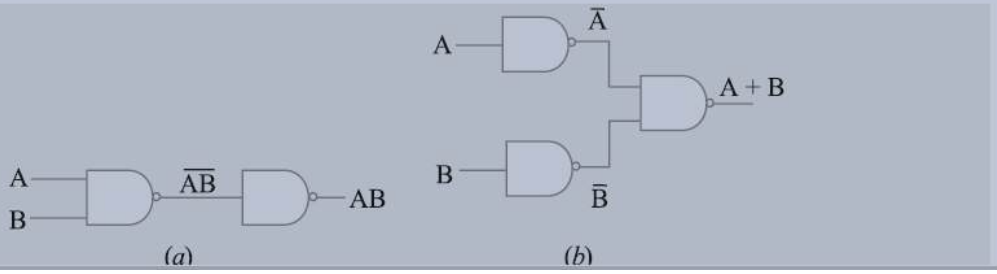
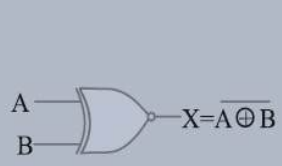


Fig. 70.44

* In this case, $V_A = V_B = 0$ V

70.23. The XNOR Gate

It is known as a not-*XOR* gate *i.e.* $\overline{XOR}$ gate. Its logic symbol and truth table are shown in Fig. 70.45.



A	B	X
0	0	1
0	1	0
1	0	0
1	1	1

Fig. 70.45

Its logic function and truth table are ***just the reverse of those for XOR gate*** (Art 70.9).

This gate has an output 1 if **its both inputs are either 0 or 1**. In other words, for getting an output, its both inputs should be ***at the* same** *logic level* of either 0 or 1. Obviously, it produces **no** output if its two inputs are at the **opposite** logic level.

70.24. Logic Gates at a Glance

In Fig. 70.46 is shown the summary of all the 2-output logic gates considered so far along with their truth tables.

Following points should prove helpful when writing these truth tables:

1. In first column A, logic values alternate between 0 and 1 every two rows
2. In second column B, logic values alternate every other row
3. Column X is filled up as per the logic function it performs

$A + B = X$

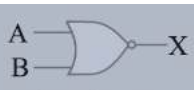
A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

$AB = X$

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

$A \oplus B = X$

A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

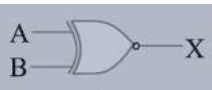


$\overline{A + B} = X$

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

$\overline{AB} = X$

A	B	X
0	0	1
0	1	1
1	0	1
1	1	0



$\overline{A \oplus B} = X$

A	B	X
0	0	1
0	1	0
1	0	0
1	1	1

Fig. 70.46

4. Truth tables for *NOR*, *NAND* and *XNOR* (or $\overline{XOR}$) gates are ***just the opposite*** of those for *OR*, *AND* and *XOR* gates.

Example 70.7. *An electrical signal is expressed as 101011. Explain its meaning. If this signal is applied to a NOT gate, what would be the output signal ?*

Fig. 70.47

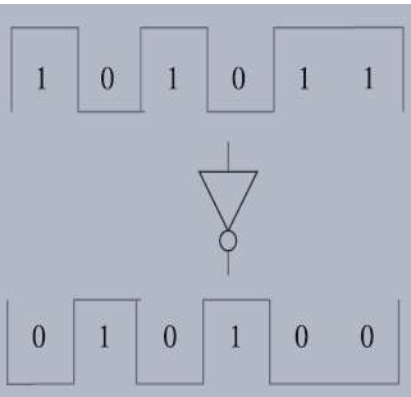


Fig. 70.48

Solution. The signal represents binary number 101011_2. It is electrically represented as a train of pulses. Taking positive logic, 1 will represent high voltage and 0 will represent low (or zero) voltage as shown in Fig. 70.47.

When such a signal is applied to a *NOT* gate, it would be inverted or complemented as shown in Fig. 70.48.

The *NOT* output will represent the binary number 010100_2.

Example 70.8. *Two electrical signals represented by A = 101101 and B = 110101 are applied to a 2-input AND gate. Sketch the output signal and the binary number it represents.*

Solution. The pulse trains corresponding to A and B are shown in Fig. 70.49.

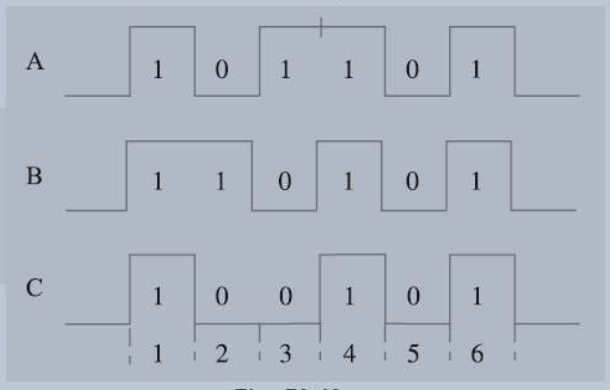


Fig. 70.49

Remember that in an *AND* gate, C is 1 only when both A and B are 1. It is an ***all-or-nothing gate***. The output can be found in different time intervals as under :

1. 1st interval : $1+1=1$
2. 2nd interval : $0+1=0$
3. 3rd interval : $1+0=0$
4. 4th interval : $1+1=1$
5. 5th interval : $0+0=0$
6. 6th interval : $1+1=1$

Hence, ouput of the *AND* gate is 100101_2. It is sketched in Fig. 70.49.

Example 70.9. *Convert the Boolean expression (AB + C) into a logic circuit using different logic gates.* **(Computer Engg. Pune Univ. 1992)**

Solution. In such cases, it is best to start with the ***output and work towards the input***. As seen, C has been *OR*ed with $A\,B$. Hence, the output gate must be a 2-input *OR* gate as shown in Fig. 70.50 (*a*).

Now, term $A\,B$ is an *AND* function. Hence, we need an *AND* gate with inputs A and B. The complete logic circuit is shown in Fig. 70.50 (*b*).

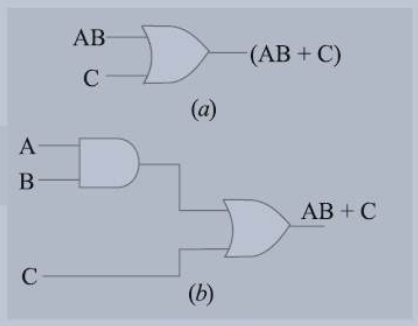


Fig. 70.50

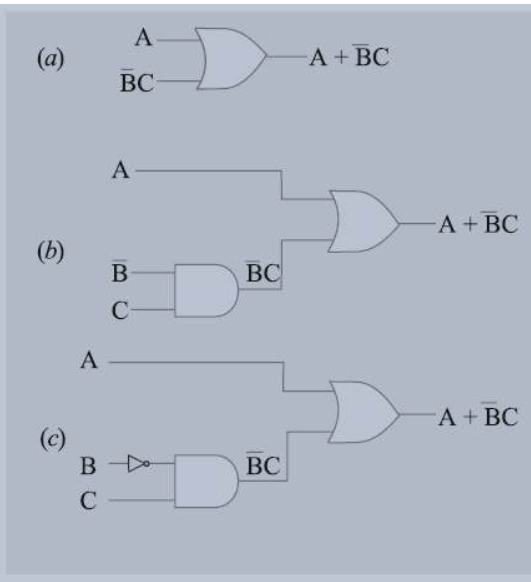
Example 70.10. *Design logic hardware based on the Boolean expression* $(A + \overline{B}\,C)$.

Solution. We will work from ***output to input***. It is seen that the last gate is a 2-input *OR* gate with inputs of A and $\overline{B}\,C$. It is shown in Fig. 70.51 (*a*).

Since $\overline{B}$ has been *AND*ed with C, it requires an *AND* gate as shown in Fig. 70.51 (*b*). For inversion of B, a *NOT* gate has been used as shown in Fig. 70.51 (*c*).

Example 70.11. *Design a logic circuit whose output is given by the Boolean expression* $(A + B).\overline{AB}$. **(Computer Science, Allahabad Univ. 1992)**

Solution. Working from output to input, we find that the output gate has to be a 2-input *AND* gate with inputs of $(A + B)$ and $\overline{AB}$. The first step of the circuit design is shown in Fig. 70.52 (*a*). It is also seen that the input to the entire circuit consists of A and B only.



Fi.g 70.51

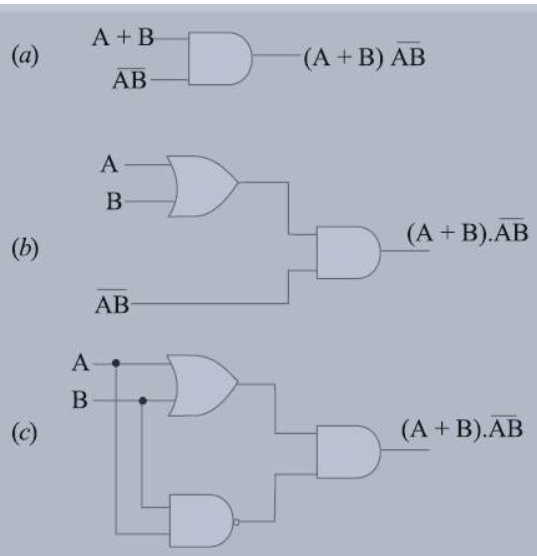
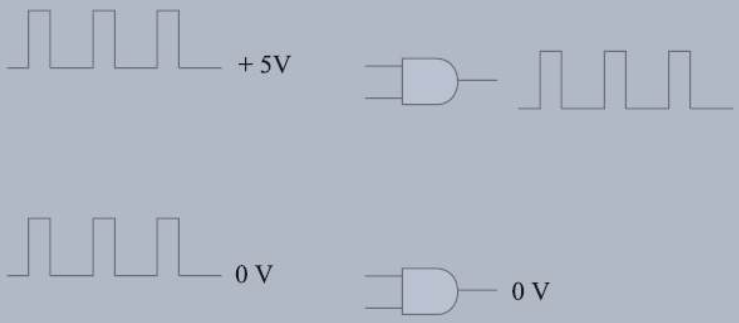


Fig. 70.52

The input of $(A + B)$ has been obtained with the help of an *OR* gate as shown in Fig. 70.52 (*b*).

Finally, a *NAND* gate is connected **in parallel** with the *OR* gate for getting its inputs of A and B and thereafter for supplying an output of $\overline{AB}$. The complete circuit is shown in Fig. 70.52 (*c*).

70.25. Digital Signals Applied to Logic Gates



A binary digital signal applied to a logic gate is nothing else but the application of a time sequence of 1's and 0's. The response of *AND* and *OR* gates to various periodic digital signals is shown in Fig. 70.53.

Fig. 70.54 shows how two waveforms can be *OR*ed by using two input gates.

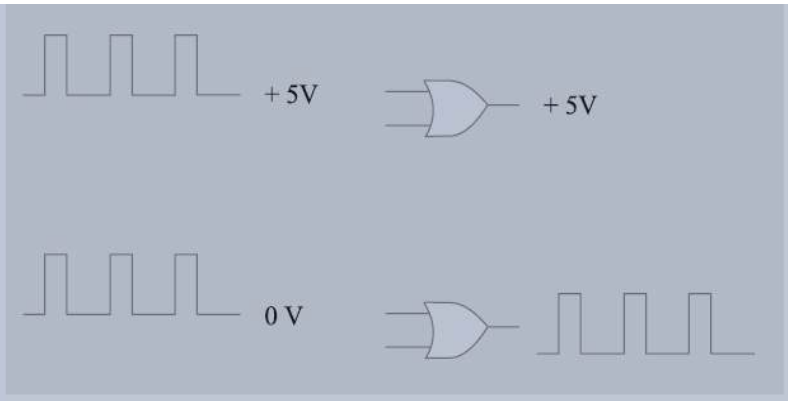


Fig. 70.53

70.26. Applications of Logic Gates

Range of application of the logic gates is very wide but the main headings would include.

1. to build more complex devices like binary counters etc.,
2. for decision making in automatic control of machines and various industrial processes,
3. in calculators and computers,
4. in digital measuring techniques,

Chapter 10

Number Systems and Arithmetic Operations

10.1 The Decimal Number System:

The Decimal number system is a number system of base or radix equal to 10, which means that there are 10, called Arabic numerals, symbols used to represent number : 0, 1, 2, 3,……..,9 , which are used for counting.

To represent more than nine units, we must either develop additional symbols or use those we have in combination. When used in combination, the value of the symbol depends on its position in the position in the combination of symbols. We refer to this as positional notation and refer to the position as having a weight designated as units, tens, hundreds, thousands, and so on.

The units symbol occupies the first position to the left of the decimal point is represented as 10^0. The second position is represented as 10^1, and so forth. To determine what the actual number is in each position, take the number that appears in the position, and multiply it by 10^X, where x is the power representation.

This is expressed mathematically of the first five positions as

10^4	10^3	10^2	10^1	10^0
Ten thousands	thousands	hundreds	tens	units

For example the value of the combination of symbols 435 is determined by adding the weight of each position as

$$4 \times 10^2 + 3 \times 10^1 + 5 \times 10^o$$

Which can be written as

$$4 \times 100 + 3 \times 10 + 5 \times 1$$

Or $400 + 30 + 5 = 435$

The position to the right of the decimal point carry a positional notation and corresponding weight as well. The exponents to the right of the decimal point are negative and increase in integer steps starting with-1. This is expressed mathematically for each of the first four positions as;

weight 10^{-1} $1\ 0^{-2}$ 10^{-3} 10^{-4}

tenths hundredths thousandths ten thousandths

For example the value of the combination of symbols, 249.34 determined by adding the weight of each position as

$$2 \times 10^2 + 4 \times 10^1 + 9 \times 10° + 3 \times 10^{-1} + 4 \times 10^{-2}$$

Or $200 + 40 + 9 + \frac{3}{10} + \frac{4}{100}$

Or $200 + 40 + 9 + 0.3 + 0.04$

$= 249.34$

10.2 The Binary Number System:

The binary number system is a number system of base or radix equal to 2, which means that there are two symbols used to represent number : 0 and 1.

A seventeenth-century German Mathematician, Gottfriend Wilhelm Von Leibniz, was a strong advocate of the binary number system. The binary number system has become extremely important in the computer age.

The symbols of the binary number system are used to represent number in the same way as in the decimal system symbol is used individually; then the symbols are use combination. Since there are only two symbols, we can represent two numbers , 0 and 1, with individual symbols. The position of the 1 or 0 in a binary number system indicates its weight or value within the number. We then combine the 1 with 0 and with itself to obtain additional numbers.

10.3 Binary and Decimal Number Correspondence:

Here are first 15 equivalence decimal and binary numbers:

Decimal Number	**Binary Numbers**
0	0000
1	0001
2	0010
3	0011

4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

An easy way to remember that how to write a binary sequence such as in the above table for a four-bits example is as follows:

1. The right most column in the binary number begins with a 0 and alternate each bit.

2. The next column begin with two O's and alternate every two bits.

3. The next column begin with four 0's and alternate every four bits.

4. The next column begin with eight-O's and alternate every eight bits.

It is seen that it takes at least four bits from 0 to 15. The formula to Count the decimal number with n bits, beginning with zero is:

$$\text{Highest decimal number} = 2^n - 1$$

for example, with two bits we can count the decimal number from 0 to 3 as,

$$2^2 - 1 = 3$$

For three bits, the decimal number is from 0 to 7, as,

$2^3 - 1 = 7$

The same type of positional weighted system is used with binary numbers as in the decimal system, The base 2 is raised to power equal to the number of positions away from the binary point The weight and designation of the several positions are as follows:

Power equal to position Base

weight	4	3	2	1	0	-1	-2	-3
positional notation	2	2	2	2	2	2	2	2
(decimal value)	16	8	4	2	1	0.5	0.25	0.125

when the symbols 0 and 1 are used to represent binary number, each symbol is called a binary digit or a bit. Thus the binary number 1010 is a four-digit binary number or a 4-bit binary number,

10.4 Binary-to-Decimal Conversion:

Since we are programmed to count in the decimal number system, it is only natural that we think in terms of the decimal equivalent value when we see a binary number. The conversion process is straight forward and is done as follows: Multiply binary digit (1 or 0) in each position by the weight of the position and add the results. The following examples explain the process.

Example 1: Convert the following binary number to their decimal equivalent. (a) 1101 (b) 1001

Solution:

(a) $1101 = (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^o)$

$= 8 + 4 + 0 + 1 = 13$

(b) $1001 = (1 \times 2^3) + (0 \times 2^2) + (0 \times 2^1)(1 \times 2^0)$

$= 8 + 0 + 0 + 1 = 9$

Example 2: Convert the following binary numbers to their decimal equivalent. (a) 0.011 (b) 0.111

Solution:

(a) $0.011 = (0 \times 2^{-1}) + (1 \times 2^{-2}) + (1 \times 2^{-3})$

$$= 0 + \frac{1}{4} + \frac{1}{8}$$

$$= 0.25 + 0.125 = 0.375$$

(b) $0.111 = (1 \times 2^{-1}) + (1 \times 2^{-2}) + (1 \times 2^{-3})$

$$= \frac{1}{2} + \frac{1}{4} + \frac{1}{8}$$

$$= 0.5 + 0.25 + 0.125 = 0.875$$

Example 3: Convert the binary number 110.011 to its decimal equivalent.

Solution:

$$110.011 = (1 \times 2^{2}) + (1 \times 2^{1}) + (0 \times 2^{\circ}) + (0 \times 2^{-1}) + (1 \times 2^{-2}) + (1 \times 2^{-3})$$

$$= 4 + 2 + 0 + 0 + \frac{1}{4} + \frac{1}{8}$$

$$= 4 + 2 + 0.25 + 0.125 = 6.375$$

10.5 Decimal-to-Binary Conversion:

It is frequently necessary to convert decimal numbers to equivalent binary numbers. The two most frequently used methods for making the conversion are the

- Repeated division-by-2 or multiplication-by-2 method.

Which is discussed below:

10.6 Repeated Division-by-2 Or Multiplication-by-2 Method:

To convert a decimal whole number to an equivalent number in a new base, the decimal number is repeatedly divided by the new base. For the case of interest here, the new base is 2, hence the repeated division by 2. Repeated division by 2 means that the original number is divided by 2, the resulting quotient is divides by 2, and each resulting quotient thereafter is divided by 2 until the quotient is 0. The remainder resulting from each division forms binary number. The first remainder to be produced is called the least significant bit (LSB) and the last remainder is called most significant bit (MSB).

When converting decimal fraction to binary, multiply repeatedly by 2 any fractional part. The equivalent binary number is formed from the

1 or 0 in the units position. The following examples illustrate the procedure.

Example 4: Convert the decimal number 17 to binary.

Solution:

2	17	
2	8 – 1	L.S.B.
2	4 – 0	
2	2 – 0	
2	1 – 0	
	0 – 1	M.S.B.

Therefore, 17 =10001

Example 5: Convert the decimal number 0.625 to binary.

Solution:

$0.625 \times 2 = 1.250$	1 (M S B)
$0.250 \times 2 = 0.500$	0
$0.25 \times 2 = 1.00$	1 (L S B)

Therefore, 0.625 = 0.101

Note: Any further multiplication by 2 in example 5 will equal to 0; therefore the multiplication can be terminated. However, this, is not so. Often it will be necessary to terminate the multiplication when an acceptable degree of accuracy is obtained. The binary number obtained will then be an approximation.

Example 6: Convert the number 0.6 to binary:

Solution: Carry

	$0.6 \times 2 = 1.2$	1 (M S B)
	$0.2 \times 2 = 0.4$	0
	$0.4 \times 2 = 0.8$	0
	$0.8 \times 2 = 1.6$	1
	$0.6 \times 2 = 1.2$	1 (LSB)
Therefore,	$0.6 = 0.10011$	

10.7 Double-Dibble Technique:

To convert a binary integer to a decimal integer we make use of double-dibble technique. The verb dibble is a neologism (i.e., a made-up-word) which has found wide spread acceptance among programmer's and other computer-oriented persons. To dibble a number is to double it and then add 1. The double-dibble technique for converting a binary integr (whole-number) goes a follows:

Begin by setting the first 'results equal to 1. If the second digit of the binary number is a zero then double this 1 (= 2) and if the second binary digit is a 1, then dibble this 1 (= 3) to obtain the second result; continue to double or dibble the successive results according to whether the successive binary digits are 0 or 1; the result corresponding to the last binary digit is the decimal equivalent of the binary integer.

Example 7: Convert 110101101 to a decimal number.

Solution:

Binary digits		1	1	0	1	0	1	1	0	1
Results	double	1		6		26			214	
	double		3		13		53	107		429

Thus 110101101 = 429

10.8 The Octal Number System:

The octal number system is used extensively in digital work because it is easy to convert from octal to binary, vice versa. The octal system has a base; or radix, of 8, which means that there are eight symbols which are used to form octal numbers. Therefore, the single-digit numbers of the octal, number system are

0 1, 2, 3, 4, 5, 6, 7

To count beyond 7, a 1 is carried to the next higher-order column and combined with each of the other symbols, as in the decimal system. The weight of the different positions far the octal system is the base raised to the appropriate power, as shown below

weight	3	2	1	0	-1	-2
positional notation	8	8	8	8	8	8
(decimal value)	512	64	8	1	$\frac{1}{8}$	$\frac{1}{64}$

Octal numbers look just like decimal numbers except that the symbols 8 and 9 are not used. To distinguish between octal and decimal numbers, we must subscript the numbers with their bas. for example, $20_8 = 16_{10}$.

The following table shows octal numbers 0 through 37 and their decimal equivalent.

Octal numbers and their Decimal equivalent

Octal	Decimal	Octal	Decimal	Octal	Decimal	Octal	Decimal
0	0	10	8	20	16	30	24
1	1	11	9	21	17	31	25
2	2	12	10	22	18	32	26
3	3	13	11	23	19	33	27
4	4	14	12	24	20	34	28
5	5	15	13	25	21	35	29
6	6	16	14	26	22	36	30
7	7	17	15	27	23	37	31

10.9 Octal-to-Decimal Conversion:

Octal numbers are converted to their decimal equivalent by multiplying the weight of each position by the digit in that position and adding the products. This is illustrated in the following examples.

Example 8: Convert the following octal numbers to their decimal, equivalent.

Solution: (a) 35_8 (b) 100_8 (c) 0.24_8

(a) $35_8 = (3 \times 8^1) + (5 \times 8^o)$

$= 24 + 5 = 29$

(a) $100_8 = (1 \times 8^2) + (0 \times 8^1) + (0 \times 8^o)$

$= 64 + 0 + 0 = 64_{10}$

(b) $0.24_8 = (2 \times 8^{-1}) + (4 \times 8^{-2})$

$= \frac{2}{8} + \frac{4}{64}$

$= 0.3125_{10}$

10.10 Decimal-to-Octal Conversion:

To convert decimal numbers to their octal equivalent, the following procedures are employed:

- Whole-number conversion: Repeated division-by-8.
- Fractional number conversion: Repeated multipli-cation-by-8.

10.11 Repeated Division-by-8 Method:

The repeated-division by 8 method of converting decimal to octal applies only to whole numbers. The procedure is illustrated in the following example.

Example 8: Convert the following decimal numbers to their octal equivalent:

(a) 245 (b) 175

Solution:

8	245		8	175	
8	30 – 5	(LSD)	8	21 – 7	
8	3 – 6		8	2 – 5	
	0 – 6 (MSD)			0 – 2	

Therefore, $245_{10} = 365_8$, therefore , $175_{10} = 257_8$

10.12 Repeated Multiplication-by-8 Method:

To convert decimal fractions to their Octal equivalent requires repeated Multiplication by 8, as shown in the following example.

Example 9: Convert the decimal fraction 0.432 to octal equivalent .

Solution:

		Carry
0.432×8	$= 3.456$	3(MSD)
0.456×8	$= 3.648$	3
0.648×8	5.184	5
0.184×8	$= 1.472$	1 (LSD)

The first carry is nearest the octal point, therefore,

$$0.432_{10} = 0.3351_8$$

The conversion to octal is not precise, since there is a remainder. If greater accuracy is required, we simply continue multiplying by 8 to obtained more octal digits.

10.13 Octal-to-Binary-Conversion:

The primary reason for our interest in octal numbers lies in entering and outputting computer data and because of the ease of octal-to-binary conversion. Computers recognize only binary information and may be programmed using only, 1's and O's.

Converting numbers from octal to binary can be done essentially by inspection. Since there are eight symbols used for counting in the octal system and eight combinations of three binary digits that corresponds to these single-digit octal numbers, we can assign a binary three-digit combination to each single-digit octal number as shown in the Table given below:

10.14 Octal and Binary Number Correspondence.

Octal	Binary
0	0 0 0
1	1 0 1
2	0 1 0
3	0 1 1
4	1 0 0
5	1 0 1
6	1 1 0
7	1 1 1

To convert from octal to binary, simply replace each octal digit with the corresponding thee-digit binary number, as illustrated in the following example.

Example 10: Convert the following octal numbers to their binary equivalent.

(a) 247_8 (b) 501_8

Solution:

(a)	2	4	7	Octal
	010	100	111	binary

Thus, $247_8 = 010100111_2$

(b)	5	0	1	octal
	101	000	001	binary

Thus, $501_8 = 101000001_2$

10.15 Binary-to-Octal Conversion:

In printing out octal numbers, the modern electronic digital computer performs a binary-to-octal conversion. This is a simple procedure. The binary number is divided into groups to three bits, counting to the right and to the left from the binary point and then each group Of three is interpreted as an octal digit; as shown in above table.

Example 11: Convert 11010101 . 01101 to an octal-number.

Solution:	011	010	101	.	011	10
	011	010	101		011	010
	3	2	5		3	2

Therefore $11010101.0110_1 = 325.32_8$

10.16 Binary Arithmetic:

Binary arithmetic includes the basic arithmetic operations of addition, subtraction, multiplication and division. The following sections present the rules that apply to these operations when they are performed on binary numbers.

Binary Addition:

Binary addition is performed in the same way as addition in the decimal-system and is, in fact, much easier to master. Binary addition obeys the following four basic rules:

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

The results of the last rule may seem some what strange, remember that these are binary numbers. Put into words, the last rule states that

binary one + binary one = binary two = binary "one zero"

When adding more than single-digit binary number, carry into, higher order columns as is done when adding decimal numbers. For example 11 and 10 are added as follows:

$$\begin{array}{r} 11 \\ +\,10 \\ \hline 101 \end{array}$$

In the first column (L S C or 2°) '1 plus 0 equal 1. In the second column (2^1) 1 plus 1 equals 0 with a carry of 1 into the third column (2^2).

When we add 1 + 1.+ 1 (carry) produces 11, recorded as 1 with a carry to the next column.

Example 12: Add (a) 111 and 101 (b) 1010, 1001 and 1101.

Solution:

(a)
$$\begin{array}{r} {\scriptstyle (1)\,(1)} \\ 1\,1\,1 \\ 1\,0\,1 \\ \hline 1\,1\,0\,0 \end{array}$$

(b)
$$\begin{array}{r} {\scriptstyle (2)\,(1)\,(1)\,(1)} \\ 1\;0\;1\;0 \\ 1\;0\;0\;1 \\ \hline 1\;1\;0\;1 \\ \hline 1\,0\,0\,0\,0 \end{array}$$

Binary Subtraction:

Binary subtraction is just as simple as addition subtraction of one bit from another obey the following four basic rules

$0 - 0 = 0$

$1 - 1 = 0$

$1 - 0 = 1$

$10 - 1 = 1$ with a transfer (borrow) of 1.

When doing subtracting, it is sometimes necessary to borrow from the next higher-order column. The only it will be necessary to borrow is when we try to subtract a 1 from a 0. In this case a 1 is borrowed from the next higher-order column, which leaves a 0 in that column and creates a

10 i.e., 2 in the column being subtracted. The following examples illustrate binary subtraction.

Example 13: Perform the following subtractions.

(a) 11 - 01 , (b) 11-10 (c) 100 - 011

Solution:

$$\text{(a)}\quad \begin{array}{r} 11 \\ -\,01 \\ \hline 10 \end{array} \qquad \text{(b)}\quad \begin{array}{r} 11 \\ -\,10 \\ \hline 01 \end{array} \qquad \text{(c)}\quad \begin{array}{r} 100 \\ -\,011 \\ \hline 001 \end{array}$$

Part (c) involves to borrows, which handled as follows. Since a 1 is to be subtracted from a 0 in the first column, a borrow is required from the next higher-order column. However, it also contains a 0; therefore, the second column must borrow the 1 in the third column. This leaves a 0 in the third column and place a 10 in the second column. Borrowing a 1 from 10 leaves a 1 in the second column and places a 10 i.e, 2 in the first column:

When subtracting a larger number from a smaller number, the results will be negative. To perform this subtraction, one must subtract the smaller number from the larger and prefix the results with the sign of the larger number.

Example 14: Perform the following subtraction 101 – 111.

Solution:

Subtract the smaller number from the larger.

$$\begin{array}{r} 111 \\ -\,101 \\ \hline 010 \end{array}$$

Thus 1 0 1 – 1 1 1 = - 010 = - 10

Binary multiplication:

Binary multiplication is performed in the same manner as decimal multiplication. It is much easier, since there are only two possible results of multiplying two bits. The Binary multiplication obeys the four basic rules.

$$0 \times 0 = 0$$

$$0 \times 1 = 0$$

$$1 \times 0 = 0$$

$1 \times 1 = 1$

Example 15: Multiply the following binary numbers.

(a) $101 \times 1\,1$ **(b) 1101×10**

(c) $1010 \times l\,0\,1$ **(d) $1\,0\,1\,l \times 1\,0\,1\,0$**

Solution:

(a)
$$\begin{array}{r} 101 \\ 1\,1\times \\ \hline 101 \\ 101 \\ \hline 1111 \end{array}$$

(b)
$$\begin{array}{r} 11101 \\ 1\,0\ \times \\ \hline 0000 \\ 1101 \\ \hline 11010 \end{array}$$

(c)
$$\begin{array}{r} 1010 \\ 101\times \\ \hline 1010 \\ 0000 \\ 1010 \\ \hline 110010 \end{array}$$

(d)
$$\begin{array}{r} 1011 \\ 1010\times \\ \hline 0000 \\ 1011\times \\ 0000\times \\ 1011\times \\ \hline 1101110 \end{array}$$

Multiplication of fractional number is performed in the same way as with fractional numbers in the decimal numbers.

Example 16: Perform the binary multiplication 0.01×11.
Solution:

$$\begin{array}{r} 0.01 \\ 1\,1\times \\ \hline 01 \\ 01\times \\ \hline 0.11 \end{array}$$

Binary Division:

Division in the binary number system employees the same procedure as division in the decimal system, as will be seen in the following examples.

Example 17: Perform the following binary division.

(a) $110 \div 11$ (b) $1100 \div 11$

Solution:

(a)

```
      10
   ------
11 | 110
     11
     ---
      0 0
      0 0
      ---
      0 0
      ---
```

(b)

```
      100
   ------
11 | 11000
     11
     ---
      0 0
      0 0
      ---
        0 0
        0 0
        ---
          0 0
```

Binary division problems with remainders are also treated the same as in the decimal system, as illustrates the following example.

Example 18: Perform the following binary division:

(a) $1111 \div 110$ (b) $1100 \div 101$

Solution:

(a)

```
      10. 1
   ---------
110 | 1 11 1.00
      1 10
      ----
        1 10
        1 10
        ----
          0 0 0
```

(b)

```
      10. 011
   ----------
110 | 1 100.00
      1 01
      ----
        100
        000
        ---
         1000
          101
          ---
           110
           101
           ---
             1
        (reminder)
```

EXERCISE 10

Q.1: Convert the following binary numbers to decimal equivalent.

(a) 100 (b) 11010 (c) 10110010 (d) 1.001
(e) 110100.010011 (f) 11010.10110 (g) 1000001.111

Q.2: Convert the following decimal numbers to binary equivalent.

(a) 16 (b) 247 (c) 962.84 (d) 0.0132
(e) 6.74

Q.3: Convert the following octal numbers to their decimal equivalent:

(a) 14_8 (b) 236_8 (c) 1432_8 (d) 0.43_8
(e) 0.254_8 (f) $(16742.3)_8$ (g) $(206.104)_8$

Q.4: Convert the following decimals numbers to their octal equivalent:

(a) 29 (b) 68 (c) 125 (d) 243.67
(e) 419.95 (f) 645.7 (g) 39.4475 (h) 49.21875

Q.5: Convert the following octal numbers to their binary equivalent:

(a) 13_8 (b) 27_8 (c) 65_8 (d) 124.375_8
(e) 217.436_8

Q.6: Convert the following binary numbers to their octal equivalent:

(a) 10110010 (b) 10101101
(c) 1110101 (d) 10111.101
(e) 111010.001

Q.7: Convert 18 × 24 to binary form and then perform binary multiplication.

Q.8: Convert 58.75 ÷ 23.5 to binary form and then perform operation.

Q.9: Add the following binary numbers:

(a)
$$\begin{array}{r} 1\ 1 \\ +\ 1\ 1 \\ \hline \end{array}$$
…. …..

(b)
$$\begin{array}{r} 1011 \\ +\ 1101 \\ \hline \end{array}$$
…….

(c)
$$\begin{array}{r} 101011 \\ +\ 110101 \\ \hline \end{array}$$
……...

(d)
$$\begin{array}{r} 1110101 \\ +\ 1011111 \\ \hline \end{array}$$
……

Q.10: Add the following binary numbers:

(a)
$$\begin{array}{r} 1011 \\ +\ 1101 \\ \hline 1011 \\ \hline \ldots\ldots \end{array}$$

(b)
$$\begin{array}{r} 1010110 \\ 1110101 \\ +\ 1001010 \\ \hline \ldots\ldots\ldots\ldots \end{array}$$

(c)
$$\begin{array}{r} 1010110 \\ 1111011 \\ +\ 1011111 \\ \hline \ldots\ldots\ldots \end{array}$$

(d)
$$\begin{array}{r} 10110110 \\ 11010101 \\ +\ 11010110 \\ \hline \ldots\ldots\ldots\ldots \end{array}$$

Q.11: Subtract the following binary numbers:

(a)
$$\begin{array}{r} 1\ 1 \\ -\ 1\ 0 \\ \hline \ldots\ldots\ldots \end{array}$$

(b)
$$\begin{array}{r} 111 \\ -\ 101 \\ \hline \ldots\ldots\ldots \end{array}$$

(c)
$$\begin{array}{r} 101011 \\ -\ 100101 \\ \hline \ldots\ldots\ldots\ldots\ldots \end{array}$$

(d)
$$\begin{array}{r} 11100001 \\ -\ 10011110 \\ \hline \ldots\ldots\ldots\ldots \end{array}$$

Q.12: Multiply the following binary numbers:

(a) 11×11 (b) 101×10 (c) 110×101

(d) 1011×1101 (e) 1010×101

Q.13: Divide the following binary numbers:

(a) $110 \div 10$ (b) $10110 \div 10$

(d) $101101 \div 1.1$ (e) $11001.11 \div 1101$

Answers 10

Q.1: (a) 4 (b) 26 (c) 178 (d) 1.125

(e) 52.2969 (f) 26.6875 (g) 65.875

Q.2: (a) 10000_2 (b) 11110111_2 (c) 1111000010.11010111_2

(d) 0.000000110110_2 (e) 110.1011110_2

Q.3: (a) 12 (b) 158 (c) 794 (d) 0.5469

(e) 0.3359 (f) 7650.375 (g) 134.1328

Q.4: (a) 35_8 (b) 104_8 (c) 175_8 (d) 363.527_8

(e) 643.2631_8 (f) $(1205.043)_8$ (g) 47.345_8

(h) 61.61_8

Q.5: (a) 001011 (b) 010111 (c) 110101

(d) 001010100.011111101 (e) 010001111.100011110

Q.6: (a) 262_8 (b) 255_8 (c) 165_8 (d) 27.5_8 (e) 72.1_8

Q.7: 110110000_2

Q.8: 10.1_2

Q.9: (a) 110 (b) 11000 (c) 1100000 (d) 11010100

Q.10: (a) 100011 (b) 100010101 (c) 100110000

(d) 1001100001

Q.11: (a) 01 (b) 10 (c) 110 (d) 1000011

Q.12: (a) 1001 (b) 1100 (c) 11110 (d) 110010

(e) 10001111

Q.13: (a) 11 (b) 1011 (c) 01.111111 (d) 11110

Short Questions

Write the short answers of the following:

Q.1: Define Binary Number.

Q.2: Define Octal numbers.

Q.3: Define Decimal number.

Q.4: Convert binary number 10101_2 to decimal numbers.

Q.5: Convert binary numbers 11111_2 to decimal numbers.

Q.6: Convert 110011.11_2 to decimal numbers.

Q.7: Add the binary number $110_2 + 1011_2$

Q.8: Add binary numbers $10011.1_2 + 11011.01_2$

Q.9: Multiply the binary numbers $111_2 \times 101_2$

Q.10: Subtract the binary numbers $1100_2 - 1001_2$

Q.11: Divide the binary numbers $1000_2 \div 100$

Q.12: Convert binary 101101_2 to octal nos.

Q.13: Convert binary numbers 10110111_2 to octal numbers.

Q.14: Convert binary numbers 11 0 11 0 . 0 11 to octal numbers.

Q.15: Convert octal numbers $103 . 45_8$ to binary number.

Q.16: Convert octal number 107_8 binary nos.

Q.17: Find the octal equivalent to $(359.325)_{10.}$

Q.18: Convert the decimal numbers 932_{10} to octal numbers.

Note: LB means least bit and GB means greatest bit

Answers

Q4 21 **Q5.** 31 **Q6.** 51.75

Q7. 11000_2 **Q8.** 101110.11_2 **Q9.** 100011_2

Q10 0011_2 **Q11**. 11_2 **Q12.** 55_8

Q13. 267_8 **Q14.** 66.3_8 **Q15.** 001000 0 11.100 101_2

Q16. 001 000 111_2 **Q17.** $(547 . 246)_8$ **Q18.** 1644_8

Objective Type Questions

Q.1 Each questions has four possible answers. Choose the correct answer and encircle it.

__1. $(11)_2$ to decimal is:

(a) 3 (b) 5 (c) 4 (d) None of these

__2. $(11)_2$ to decimal is:

(a) $\frac{3}{2}$ (b) 2.2 (c) 3 (d) None of these

__3. 25 when converted to octal is:

(a) $(31)_8$ (b) $(2.5)_8$ (c) $(13)_8$ (d) None of these

__4. $(11)_2 + (101)_2$ is equal to:

(a) $(1100)_2$ (b) $(121)_2$ (c) $(112)_2$ (d) None of these

__5. Addition of $(45)_8$ and $(73)_8$ is:

(a) $(140)_8$ (b) $(118)_8$ (c) $(110)_8$

__6. $(11)_8 \times (7)_8$ is equal to:

(a) $(105)_8$ (b) $(77)_8$ (c) $(43)_8$ (d) None of these

__7. Number of digits in a binary system are:

(a) 2 (b) 7 (c) 10 (d) None of these

__8. $(11)_2 \times (11)_2$ is equal to:

(a) $(1101)_2$ (b) $(1001)_2$ (c) $(121)_2$ (d) None of these

__9. $(11)_8$ to decimal is equal to:

(a) 9 (b) 88 (c) 110 (d) None of these

__10. Conversion of 9 to binary system is:

(a) $(1001)_2$ (b) $(101)_2$ (c) $(11)_2$ (d) None of these

ANSWERS:

1.	a	2.	a	3.	d	4.	a	5.	a
6.	b	7.	a	8.	a	9.	a	10.	a

Number Systems, Base Conversions, and Computer Data Representation

Decimal and Binary Numbers

When we write decimal (base 10) numbers, we use a positional notation system. Each digit is multiplied by an appropriate power of 10 depending on its position in the number:

For example:

$$843 = 8 \times 10^2 + 4 \times 10^1 + 3 \times 10^0$$
$$= 8 \times 100 + 4 \times 10 + 3 \times 1$$
$$= 800 + 40 + 3$$

For whole numbers, the rightmost digit position is the one's position ($10^0 = 1$). The numeral in that position indicates how many ones are present in the number. The next position to the left is ten's, then hundred's, thousand's, and so on. Each digit position has a weight that is ten times the weight of the position to its right.

In the decimal number system, there are ten possible values that can appear in each digit position, and so there are ten numerals required to represent the quantity in each digit position. The decimal numerals are the familiar zero through nine (0, 1, 2, 3, 4, 5, 6, 7, 8, 9).

In a positional notation system, the number base is called the radix. Thus, the base ten system that we normally use has a radix of 10. The term radix and base can be used interchangeably. When writing numbers in a radix other than ten, or where the radix isn't clear from the context, it is customary to specify the radix using a subscript. Thus, in a case where the radix isn't understood, decimal numbers would be written like this:

$$127_{10} \quad 11_{10} \quad 5673_{10}$$

Generally, the radix will be understood from the context and the radix specification is left off.

The binary number system is also a positional notation numbering system, but in this case, the base is not ten, but is instead two. Each digit position in a binary number represents a power of two. So, when we write a binary number, each binary digit is multiplied by an appropriate power of 2 based on the position in the number:

For example:

$$101101 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$
$$= 1 \times 32 + 0 \times 16 + 1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1$$
$$= 32 + 8 + 4 + 1$$

In the binary number system, there are only two possible values that can appear in each digit position rather than the ten that can appear in a decimal number. Only the numerals 0 and 1 are used in binary numbers. The term 'bit' is a contraction of the words 'binary' and 'digit', and when talking about binary numbers the terms bit and digit can be used interchangeably. When talking about binary numbers, it is often necessary to talk of the number of bits used to store or represent the number. This merely describes the number of binary digits that would be required to write the number. The number in the above example is a 6 bit number.

The following are some additional examples of binary numbers:

$$101101_2 \quad 11_2 \quad 10110_2$$

Conversion between Decimal and Binary

Converting a number from binary to decimal is quite easy. All that is required is to find the decimal value of each binary digit position containing a 1 and add them up.

For example: convert 10110_2 to decimal.

```
1 0 1 1 0
 \   \ \_____________1 x 2^1 =  2
  \   \_____________1 x 2^2 =  4
   \_________________1 x 2^4 = 16
                               22
```

Another example: convert 11011_2 to decimal

```
1 1 0 1 1
 \ \   \ \_________1 x 2^0 =  1
  \ \   \__________1 x 2^1 =  2
   \ \_____________1 x 2^3 =  8
    \_______________1 x 2^4 = 16
                              27
```

The method for converting a decimal number to binary is one that can be used to convert from decimal to any number base. It involves using successive division by the radix until the dividend reaches 0. At each division, the remainder provides a digit of the converted number, starting with the least significant digit.

An example of the process: convert 37_{10} to binary

```
37 / 2 = 18       remainder 1       (least significant digit)
18 / 2 = 9        remainder 0
9 / 2 = 4         remainder 1
4 / 2 = 2         remainder 0
2 / 2 = 1         remainder 0
1 / 2 = 0         remainder 1       (most significant digit)
```

The resulting binary number is: 100101

Another example: convert 93_{10} to binary

```
93 / 2 = 46       remainder 1       (least significant digit)
46 / 2 = 23       remainder 0
23 / 2 = 11       remainder 1
11 / 2 = 5        remainder 1
 5 / 2 = 2        remainder 1
 2 / 2 = 1        remainder 0
 1 / 2 = 0        remainder 1       (most significant digit)
```

The resulting binary number is: 1011101

Hexadecimal Numbers

In addition to binary, another number base that is commonly used in digital systems is base 16. This number system is called hexadecimal, and each digit position represents a power of 16. For any number base greater than ten, a problem occurs because there are more than ten symbols needed to represent the numerals for that number base. It is customary in these cases to use the

ten decimal numerals followed by the letters of the alphabet beginning with A to provide the needed numerals. Since the hexadecimal system is base 16, there are sixteen numerals required. The following are the hexadecimal numerals:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

The following are some examples of hexadecimal numbers:

10_{16} 47_{16} $3FA_{16}$ $A03F_{16}$

The reason for the common use of hexadecimal numbers is the relationship between the numbers 2 and 16. Sixteen is a power of 2 ($16 = 2^4$). Because of this relationship, four digits in a binary number can be represented with a single hexadecimal digit. This makes conversion between binary and hexadecimal numbers very easy, and hexadecimal can be used to write large binary numbers with much fewer digits. When working with large digital systems, such as computers, it is common to find binary numbers with 8, 16 and even 32 digits. Writing a 16 or 32 bit binary number would be quite tedious and error prone. By using hexadecimal, the numbers can be written with fewer digits and much less likelihood of error.

To convert a binary number to hexadecimal, divide it into groups of four digits starting with the rightmost digit. If the number of digits isn't a multiple of 4, prefix the number with 0's so that each group contains 4 digits. For each four digit group, convert the 4 bit binary number into an equivalent hexadecimal digit. (See the Binary, BCD, and Hexadecimal Number Tables at the end of this document for the correspondence between 4 bit binary patterns and hexadecimal digits)

For example: Convert the binary number 10110101 to a hexadecimal number

Divide into groups for 4 digits	1011 0101
Convert each group to hex digit	B 5
	$B5_{16}$

Another example: Convert the binary number 0110101110001100 to hexadecimal

Divide into groups of 4 digits	0110 1011 1000 1100
Convert each group to hex digit	6 B 8 C
	$6B8C_{16}$

To convert a hexadecimal number to a binary number, convert each hexadecimal digit into a group of 4 binary digits.

Example: Convert the hex number 374F into binary

	3 7 4 F
Convert the hex digits to binary	0011 0111 0100 1111
	0011011101001111_2

There are several ways in common use to specify that a given number is in hexadecimal representation rather than some other radix. In cases where the context makes it absolutely clear that numbers are represented in hexadecimal, no indicator is used. In much written material where the context doesn't make it clear what the radix is, the numeric subscript 16 following the hexadecimal number is used. In most programming languages, this method isn't really feasible, so there are several conventions used depending on the language. In the C and C++ languages, hexadecimal constants are represented with a '0x' preceding the number, as in: 0x317F, or 0x1234, or 0xAF. In assembler programming languages that follow the Intel style, a hexadecimal constant begins with a numeric character (so that the assembler can distinguish it from a variable

name), a leading '0' being used if necessary. The letter 'h' is then suffixed onto the number to inform the assembler that it is a hexadecimal constant. In Intel style assembler format: 371Fh and 0FABCh are valid hexadecimal constants. Note that: A37h isn't a valid hexadecimal constant. It doesn't begin with a numeric character, and so will be taken by the assembler as a variable name. In assembler programming languages that follow the Motorola style, hexadecimal constants begin with a '$' character. So in this case: $371F or $FABC or $01 are valid hexadecimal constants.

Binary Coded Decimal Numbers

Another number system that is encountered occasionally is Binary Coded Decimal. In this system, numbers are represented in a decimal form, however each decimal digit is encoded using a four bit binary number.

For example: The decimal number 136 would be represented in BCD as follows:

```
136 = 0001 0011 0110
        1    3    6
```

Conversion of numbers between decimal and BCD is quite simple. To convert from decimal to BCD, simply write down the four bit binary pattern for each decimal digit. To convert from BCD to decimal, divide the number into groups of 4 bits and write down the corresponding decimal digit for each 4 bit group.

There are a couple of variations on the BCD representation, namely packed and unpacked. An unpacked BCD number has only a single decimal digit stored in each data byte. In this case, the decimal digit will be in the low four bits and the upper 4 bits of the byte will be 0. In the packed BCD representation, two decimal digits are placed in each byte. Generally, the high order bits of the data byte contain the more significant decimal digit.

An example: The following is a 16 bit number encoded in packed BCD format:

```
01010110 10010011
```

This is converted to a decimal number as follows:

```
0101 0110 1001 0011
  5    6    9    3
```

The value is 5693 decimal

Another example: The same number in unpacked BCD (requires 32 bits)

```
00000101 00000110 00001001 00000011
     5        6        9        3
```

The use of BCD to represent numbers isn't as common as binary in most computer systems, as it is not as space efficient. In packed BCD, only 10 of the 16 possible bit patterns in each 4 bit unit are used. In unpacked BCD, only 10 of the 256 possible bit patterns in each byte are used. A 16 bit quantity can represent the range 0-65535 in binary, 0-9999 in packed BCD and only 0-99 in unpacked BCD.

Fixed Precision and Overflow.

So far, in talking about binary numbers, we haven't considered the maximum size of the number. We have assumed that as many bits are available as needed to represent the number. In most computer systems, this isn't the case. Numbers in computers are typically represented using a fixed number of bits. These sizes are typically 8 bits, 16 bits, 32 bits, 64 bits and 80 bits. These sizes are generally a multiple of 8, as most computer memories are organized on an 8 bit byte basis. Numbers in which a specific number of bits are used to represent the value are called fixed precision numbers. When a specific number of bits are used to represent a number, that determines the range of possible values that can be represented. For example, there are 256

possible combinations of 8 bits, therefore an 8 bit number can represent 256 distinct numeric values and the range is typically considered to be 0-255. Any number larger than 255 can't be represented using 8 bits. Similarly, 16 bits allows a range of 0-65535.

When fixed precision numbers are used, (as they are in virtually all computer calculations) the concept of overflow must be considered. An overflow occurs when the result of a calculation can't be represented with the number of bits available. For example when adding the two eight bit quantities: 150 + 170, the result is 320. This is outside the range 0-255, and so the result can't be represented using 8 bits. The result has overflowed the available range. When overflow occurs, the low order bits of the result will remain valid, but the high order bits will be lost. This results in a value that is significantly smaller than the correct result.

When doing fixed precision arithmetic (which all computer arithmetic involves) it is necessary to be conscious of the possibility of overflow in the calculations.

Signed and Unsigned Numbers.

So far, we have only considered positive values for binary numbers. When a fixed precision binary number is used to hold only positive values, it is said to be *unsigned*. In this case, the range of positive values that can be represented is 0 -- 2^n-1, where n is the number of bits used. It is also possible to represent signed (negative as well as positive) numbers in binary. In this case, part of the total range of values is used to represent positive values, and the rest of the range is used to represent negative values.

There are several ways that signed numbers can be represented in binary, but the most common representation used today is called two's complement. The term two's complement is somewhat ambiguous, in that it is used in two different ways. First, as a representation, two's complement is a way of interpreting and assigning meaning to a bit pattern contained in a fixed precision binary quantity. Second, the term two's complement is also used to refer to an operation that can be performed on the bits of a binary quantity. As an operation, the two's complement of a number is formed by inverting all of the bits and adding 1. In a binary number being interpreted using the two's complement representation, the high order bit of the number indicates the sign. If the sign bit is 0, the number is positive, and if the sign bit is 1, the number is negative. For positive numbers, the rest of the bits hold the true magnitude of the number. For negative numbers, the lower order bits hold the complement (or bitwise inverse) of the magnitude of the number. It is important to note that two's complement representation can only be applied to fixed precision quantities, that is, quantities where there are a set number of bits.

Two's complement representation is used because it reduces the complexity of the hardware in the arithmetic-logic unit of a computer's CPU. Using a two's complement representation, all of the arithmetic operations can be performed by the same hardware whether the numbers are considered to be unsigned or signed. The bit operations performed are identical, the difference comes from the interpretation of the bits. The interpretation of the value will be different depending on whether the value is considered to be unsigned or signed.

For example: Find the 2's complement of the following 8 bit number

```
   00101001

   11010110        First, invert the bits
 + 00000001        Then, add 1
 = 11010111
```

The 2's complement of 00101001 is 11010111

Another example: Find the 2's complement of the following 8 bit number
10110101

```
  01001010        Invert the bits
+ 00000001        then add 1
= 01001011
```

The 2's complement of 10110101 is 01001011

The counting sequence for an eight bit binary value using 2's complement representation appears as follows:

01111111	7Fh	127	largest magnitude positive number
01111110	7Eh	126	
01111101	7Dh	125	
…			
00000011	03h		
00000010	02h		
00000001	01h		
00000000	00h		
11111111	0FFh	-1	
11111110	0FEh	-2	
11111101	0FDh	-3	
…			
10000010	82h	-126	
10000001	81h	-127	
10000000	80h	-128	largest magnitude negative number

Notice in the above sequence that counting up from 0, when 127 is reached, the next binary pattern in the sequence corresponds to -128. The values jump from the greatest positive number to the greatest negative number, but that the sequence is as expected after that. (i.e. adding 1 to –128 yields –127, and so on.). When the count has progressed to 0FFh (or the largest unsigned magnitude possible) the count wraps around to 0. (i.e. adding 1 to –1 yields 0).

ASCII Character Encoding

The name ASCII is an acronym for: American Standard Code for Information Interchange. It is a character encoding standard developed several decades ago to provide a standard way for digital machines to encode characters. The ASCII code provides a mechanism for encoding alphabetic characters, numeric digits, and punctuation marks for use in representing text and numbers written using the Roman alphabet. As originally designed, it was a seven bit code. The seven bits allow the representation of 128 unique characters. All of the alphabet, numeric digits and standard English punctuation marks are encoded. The ASCII standard was later extended to an eight bit code (which allows 256 unique code patterns) and various additional symbols were added, including characters with diacritical marks (such as accents) used in European languages, which don't appear in English. There are also numerous non-standard extensions to ASCII giving different encoding for the upper 128 character codes than the standard. For example, The character set encoded into the display card for the original IBM PC had a non-standard encoding for the upper character set. This is a non-standard extension that is in very wide spread use, and could be considered a standard in itself.

Some important things to note about ASCII code:

1) The numeric digits, 0-9, are encoded in sequence starting at 30h
2) The upper case alphabetic characters are sequential beginning at 41h
3) The lower case alphabetic characters are sequential beginning at 61h

4) The first 32 characters (codes 0-1Fh) and 7Fh are control characters. They do not have a standard symbol (glyph) associated with them. They are used for carriage control, and protocol purposes. They include 0Dh (CR or carriage return), 0Ah (LF or line feed), 0Ch (FF or form feed), 08h (BS or backspace).
5) Most keyboards generate the control characters by holding down a control key (CTRL) and simultaneously pressing an alphabetic character key. The control code will have the same value as the lower five bits of the alphabetic key pressed. So, for example, the control character 0Dh is carriage return. It can be generated by pressing CTRL-M. To get the full 32 control characters a few at the upper end of the range are generated by pressing CTRL and a punctuation key in combination. For example, the ESC (escape) character is generated by pressing CTRL-[(left square bracket).

Conversions Between Upper and Lower Case ASCII Letters.

Notice on the ASCII code chart that the uppercase letters start at 41h and that the lower case letters begin at 61h. In each case, the rest of the letters are consecutive and in alphabetic order. The difference between 41h and 61h is 20h. Therefore the conversion between upper and lower case involves either adding or subtracting 20h to the character code. To convert a lower case letter to upper case, subtract 20h, and conversely to convert upper case to lower case, add 20h. It is important to note that you need to first ensure that you do in fact have an alphabetic character before performing the addition or subtraction. Ordinarily, a check should be made that the character is in the range 41h–5Ah for upper case or 61h-7Ah for lower case.

Conversion Between ASCII and BCD.

Notice also on the ASCII code chart that the numeric characters are in the range 30h-39h. Conversion between an ASCII encoded digit and an unpacked BCD digit can be accomplished by adding or subtracting 30h. Subtract 30h from an ASCII digit to get BCD, or add 30h to a BCD digit to get ASCII. Again, as with upper and lower case conversion for alphabetic characters, it is necessary to ensure that the character is in fact a numeric digit before performing the subtraction. The digit characters are in the range 30h-39h.

Binary, BCD and Hexadecimal Number Tables

Powers of 2:

2^0 1
2^1 2
2^2 4
2^3 8
2^4 16
2^5 32
2^6 64
2^7 128
2^8 256
2^9 512
2^{10} 1024
2^{11} 2048
2^{12} 4096
2^{13} 8192
2^{14} 16384
2^{15} 32768
2^{16} 65536

Hexadecimal Digits

0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

BCD Digits

0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Equivalent Numbers in Decimal, Binary and Hexadecimal Notation:

Decimal	Binary	Hexadecimal
0	00000000	00
1	00000001	01
2	00000010	02
3	00000011	03
4	00000100	04
5	00000101	05
6	00000110	06
7	00000111	07
8	00001000	08
9	00001001	09
10	00001010	0A
11	00001011	0B
12	00001100	0C
13	00001101	0D
14	00001110	0E
15	00001111	0F
16	00010000	10
17	00010001	11
31	00011111	1F
32	00100000	20
63	00111111	3F
64	01000000	40
65	01000001	41
127	01111111	7F
128	10000000	80
129	10000001	81
255	11111111	FF
256	0000000100000000	0100
32767	0111111111111111	7FFF
32768	1000000000000000	8000
65535	1111111111111111	FFFF

ASCII Character Set

Low Order Bits		High Order Bits 0000 **0**	0001 **1**	0010 **2**	0011 **3**	0100 **4**	0101 **5**	0110 **6**	0111 **7**
0000	**0**	NUL	DLE	Space	0	@	P	`	p
0001	**1**	SOH	DC1	!	1	A	Q	a	q
0010	**2**	STX	DC2	“	2	B	R	b	r
0011	**3**	ETX	DC3	#	3	C	S	c	s
0100	**4**	EOT	DC4	$	4	D	T	d	t
0101	**5**	ENQ	NAK	%	5	E	U	e	u
0110	**6**	ACK	SYN	&	6	F	V	f	v
0111	**7**	BEL	ETB	‘	7	G	W	g	w
1000	**8**	BS	CAN	(	8	H	X	h	x
1001	**9**	HT	EM	)	9	I	Y	i	y
1010	**A**	LF	SUB	*	:	J	Z	j	z
1011	**B**	VT	ESC	+	;	K	[	k	{
1100	**C**	FF	FS	,	<	L	\	l	|
1101	**D**	CR	GS	-	=	M	]	m	}
1110	**E**	SO	RS	.	>	N	^	n	~
1111	**F**	SI	US	/	?	O	_	o	DEL

BINARY CODES

In the coding, when numbers, letters or words are represented by a specific group of symbols, it is said that the number, letter or word is being encoded. The group of symbols is called as a code. The digital data is represented, stored and transmitted as group of binary bits. This group is also called as **binary code**. The binary code is represented by the number as well as alphanumeric letter.

Advantages of Binary Code

Following is the list of advantages that binary code offers.

- Binary codes are suitable for the computer applications.
- Binary codes are suitable for the digital communications.
- Binary codes make the analysis and designing of digital circuits if we use the binary codes.
- Since only 0 & 1 are being used, implementation becomes easy.

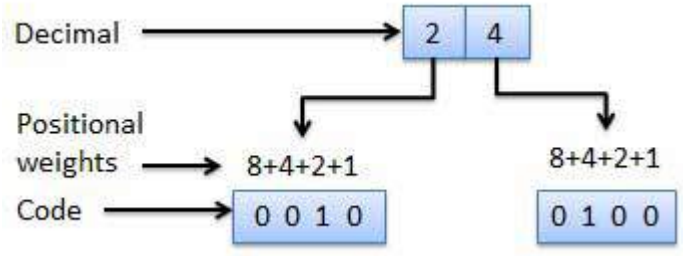
Classification of binary codes

The codes are broadly categorized into following four categories.

- Weighted Codes
- Non-Weighted Codes
- Binary Coded Decimal Code
- Alphanumeric Codes
- Error Detecting Codes
- Error Correcting Codes

Weighted Codes

Weighted binary codes are those binary codes which obey the positional weight principle. Each position of the number represents a specific weight. Several systems of the codes are used to express the decimal digits 0 through 9. In these codes each decimal digit is represented by a group of four bits.

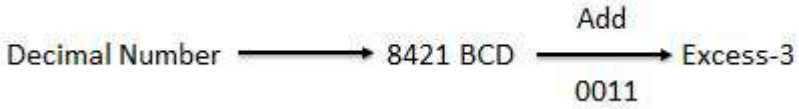


Non-Weighted Codes

In this type of binary codes, the positional weights are not assigned. The examples of non-weighted codes are Excess-3 code and Gray code.

Excess-3 code

The Excess-3 code is also called as XS-3 code. It is non-weighted code used to express decimal numbers. The Excess-3 code words are derived from the 8421 BCD code words adding 0011_2 or 3_{10} to each code word in 8421. The excess-3 codes are obtained as follows −



Example

Decimal	BCD 8 4 2 1	Excess-3 BCD + 0011
0	0 0 0 0	0 0 1 1
1	0 0 0 1	0 1 0 0
2	0 0 1 0	0 1 0 1
3	0 0 1 1	0 1 1 0
4	0 1 0 0	0 1 1 1
5	0 1 0 1	1 0 0 0
6	0 1 1 0	1 0 0 1
7	0 1 1 1	1 0 1 0
8	1 0 0 0	1 0 1 1
9	1 0 0 1	1 1 0 0

Gray Code

It is the non-weighted code and it is not arithmetic codes. That means there are no specific weights assigned to the bit position. It has a very special feature that, only one bit will change each time the decimal number is incremented as shown in fig. As only one bit changes at a time, the gray code is called as a unit distance code. The gray code is a cyclic code. Gray code cannot be used for arithmetic operation.

Decimal	BCD	Gray
0	0 0 0 0	0 0 0 0
1	0 0 0 1	0 0 0 1
2	0 0 1 0	0 0 1 1
3	0 0 1 1	0 0 1 0
4	0 1 0 0	0 1 1 0
5	0 1 0 1	0 1 1 1
6	0 1 1 0	0 1 0 1
7	0 1 1 1	0 1 0 0
8	1 0 0 0	1 1 0 0
9	1 0 0 1	1 1 0 1

Application of Gray code

- Gray code is popularly used in the shaft position encoders.
- A shaft position encoder produces a code word which represents the angular position of the shaft.

Binary Coded Decimal *BCD* code

In this code each decimal digit is represented by a 4-bit binary number. BCD is a way to express each of the decimal digits with a binary code. In the BCD, with four bits we can represent sixteen numbers $0000 to 1111$. But in BCD code only first ten of these are used $0000 to 1001$. The remaining six

code combinations i.e. 1010 to 1111 are invalid in BCD.

Decimal	0	1	2	3	4	5	6	7	8	9
BCD	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

Advantages of BCD Codes

- It is very similar to decimal system.
- We need to remember binary equivalent of decimal numbers 0 to 9 only.

Disadvantages of BCD Codes

- The addition and subtraction of BCD have different rules.
- The BCD arithmetic is little more complicated.
- BCD needs more number of bits than binary to represent the decimal number. So BCD is less efficient than binary.

Alphanumeric codes

A binary digit or bit can represent only two symbols as it has only two states '0' or '1'. But this is not enough for communication between two computers because there we need many more symbols for communication. These symbols are required to represent 26 alphabets with capital and small letters, numbers from 0 to 9, punctuation marks and other symbols.

The alphanumeric codes are the codes that represent numbers and alphabetic characters. Mostly such codes also represent other characters such as symbol and various instructions necessary for conveying information. An alphanumeric code should at least represent 10 digits and 26 letters of alphabet i.e. total 36 items. The following three alphanumeric codes are very commonly used for the data representation.

- American Standard Code for Information Interchange *ASCII*.
- Extended Binary Coded Decimal Interchange Code *EBCDIC*.
- Five bit Baudot Code.

ASCII code is a 7-bit code whereas EBCDIC is an 8-bit code. ASCII code is more commonly used worldwide while EBCDIC is used primarily in large IBM computers.

Error Codes

There are binary code techniques available to detect and correct data during data transmission.

Error Code	Description
Error Detection and Correction	Error detection and correction code techniques

Subtraction by 2's Complement

With the help of subtraction by 2's complement method we can easily subtract two binary numbers.

The operation is carried out by means of the following steps:

(i) At first, 2's complement of the subtrahend is found.

(ii) Then it is added to the minuend.

(iii) If the final carry over of the sum is 1, it is dropped and the result is positive.

(iv) If there is no carry over, the two's complement of the sum will be the result and it is negative.

The following examples on subtraction by 2's complement will make the procedure clear:

Evaluate:

(i) 110110 - 10110

Solution:

The numbers of bits in the subtrahend is 5 while that of minuend is 6. We make the number of bits in the subtrahend equal to that of minuend by taking a `0' in the sixth place of the subtrahend.

Now, 2's complement of 010110 is (101101 + 1) i.e.101010. Adding this with the minuend.

	1 1 0 1 1 0	Minuend
	1 0 1 0 1 0	2's complement of subtrahend
Carry over 1	1 0 0 0 0 0	Result of addition

After dropping the carry over we get the result of subtraction to be 100000.

(ii) 10110 – 11010

Solution:

2's complement of 11010 is (00101 + 1) i.e. 00110. Hence

Minued -	1 0 1 1 0
2's complement of subtrahend -	0 0 1 1 0
Result of addition -	1 1 1 0 0

As there is no carry over, the result of subtraction is negative and is obtained by writing the 2's complement of 11100 i.e.(00011 + 1) or 00100.

Hence the difference is – 100.

(iii) 1010.11 – 1001.01

Solution:

2's complement of 1001.01 is 0110.11. Hence

Minued -		1 0 1 0 . 1 1
2's complement of subtrahend -		0 1 1 0 . 1 1
Carry over	1	0 0 0 1 . 1 0

After dropping the carry over we get the result of subtraction as 1.10.

(iv) 10100.01 – 11011.10

Solution:

2's complement of 11011.10 is 00100.10. Hence

Minued -	1 0 1 0 0 . 0 1
2's complement of subtrahend -	0 1 1 0 0 . 1 0
Result of addition -	1 1 0 0 0 . 1 1

As there is no carry over the result of subtraction is negative and is obtained by writing the 2's complement of 11000.11.

Hence the required result is – 00111.01.