COMPUTER ORGANIZATION AND ARCHITECTURE

Fundamentals of Computer Architecture

Dr. Bore Gowda S B
Additional Professor
Dept. of ECE
MIT, Manipal

Introduction

- Why Computer Organization and Architecture?
  - Because you use it everyday
  - Because you will likely use it for the rest of your life
  - It focuses on the interface between hardware and software
  - It emphasizes the structure and behavior of the system
  - You can build your own computing device as per your requirement
  - Knowing how the hardware operates makes you a better coder
  - For a hardware engineer, designing the parts is a basic requirement

Introduction

- What is Computer Architecture?
- It refers to system attributes that are visible to the software programmer and have a direct influence on a program's logical execution, such as
  - the number of bits needed to represent distinct types of data,
  - the computer's instruction set,
  - the technique for addressing memory,
  - the method used for input and output, and so on.
- Computer architecture is a group of rules, orders, and processes that describe the functionality and performance of computer systems.
- Basically, it deals with the operational behaviour of computer systems.
- It involves decisions about the organization of the hardware, such as:
  - the instruction set architecture
  - the data path design
  - the control unit design.
- Computer architecture is concerned with optimizing the performance of a computer system and ensuring that it can execute instructions quickly and efficiently.

Introduction

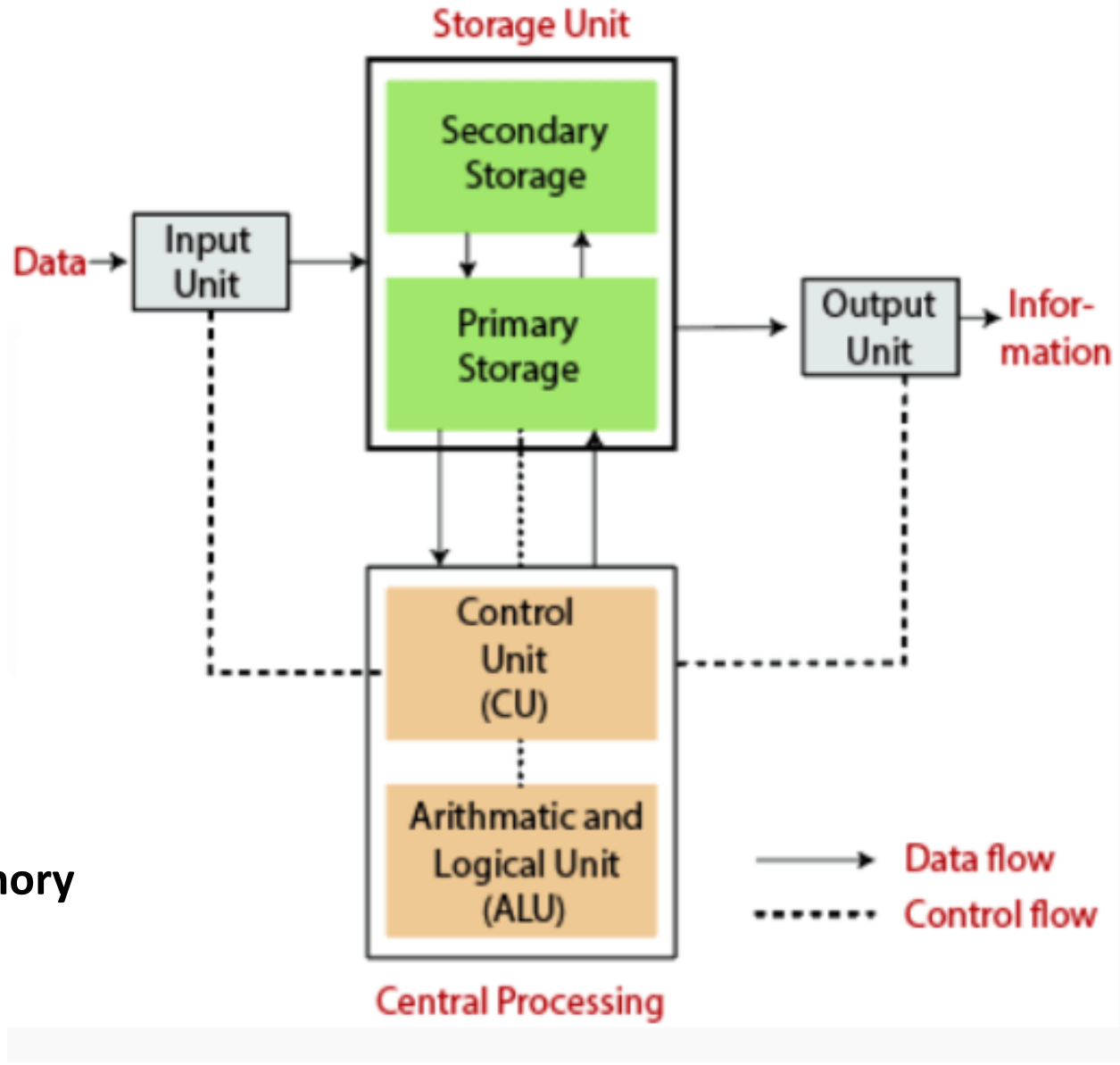
- What is Computer Organisation?
- Computer organisation is also known as Microarchitecture.
- What does microarchitecture include?
  - **Instruction implementation**: How instructions are carried out
  - **Data management**: How data is handled
  - **Control flow management**: How control flow is managed
  - **Techniques**: Techniques such as single-cycle, multicycle, pipelined, superscalar, and out-of-order processing
- It provides deep knowledge of ***functionality***, ***structuring***, ***internal working***, and ***implementation*** of a computer system.
- It refers to the operational units and their interconnections that realize the architectural specifications
- Examples of organizational attributes include those hardware details transparent to the programmer, such as
  - ✓ Control signals,
  - ✓ Interfaces between the computer and peripherals
  - ✓ Memory technology

Introduction

S.No	Computer Architecture	Computer Organization
1.	Explain what a computer does.	Explain how a computer actually does it.
2.	Majorly focus on the functional behaviour of computer systems.	Majorly focus on the structural relationship and deep knowledge of the internal working of a system.
3.	Deal with high-level design matters.	Deal with low-level design matters.
4.	It comes before computer organisation.	It comes after the architecture part.
5.	It is also called instruction set architecture.	It is also called microarchitecture.
6.	It covers logical functions, such as registers, data types, instruction sets, and addressing modes.	It covers physical units like peripherals, circuit designs, and adders.
7.	They coordinate between the hardware and software of the system.	They manage the portion of the network in a system.

Basic components of a computer

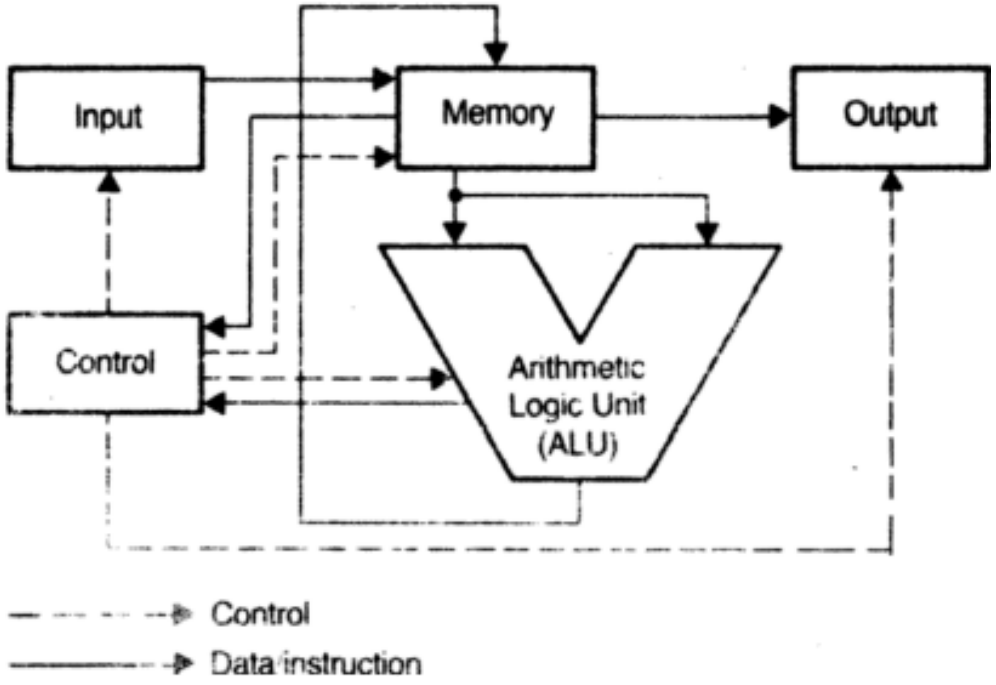
- To execute instructions, the computer performs the following steps:
  - the control unit reads or fetches an instruction from memory and decodes are translates it
  - for arthritic or logic type instructions, the control unit generates enables signals for the ALU to perform the required operations
  - for input and output(I/O) instructions, the control unit generates enable signals for the IO either to input data from or output data to external devices



- **CPU = program control unit + ALU + registers**
- **Memory = RAM + ROM + ERPROM etc+ secondary memory**

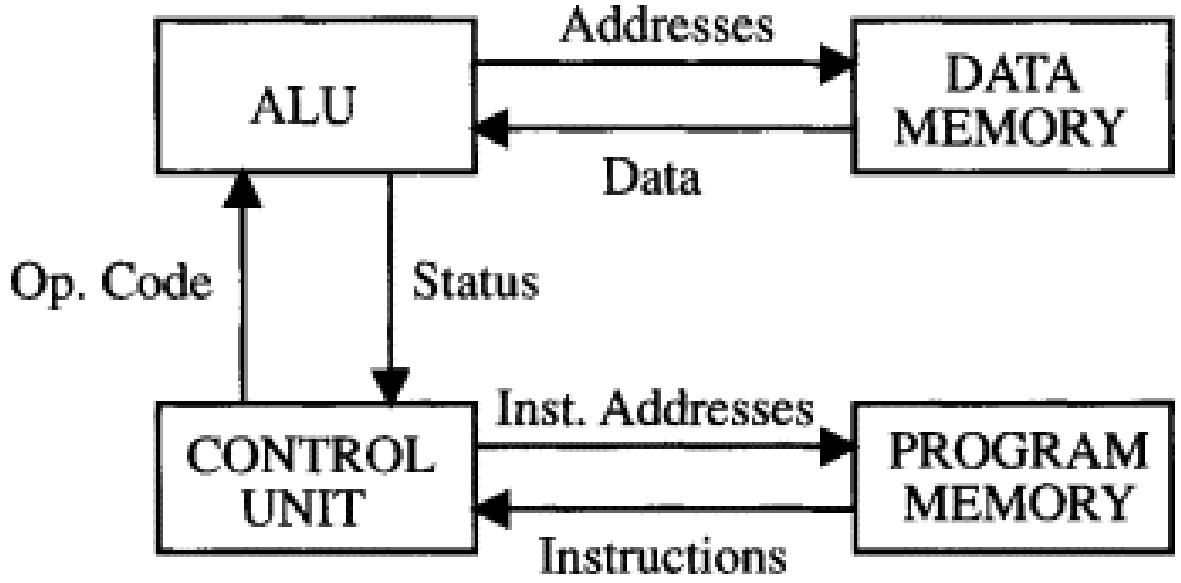
Von-Neumann computer architecture

- Von-Neumann proposed his computer architecture design in 1945 which was later known as Von-Neumann Architecture.
- The major components of it are:
  - ✓ a Control Unit
  - ✓ Arithmetic and Logical Memory Unit (ALU),
  - ✓ Registers and Inputs/Outputs.
- It is based on the ***stored-program computer concept***, where ***instruction data and program data*** are stored in the ***same memory***.
- This design is still used in most computers produced today.
- A Von Neumann-based computer:
  - ✓ Uses a single processor
  - ✓ Uses one memory for both instructions and data.
  - ✓ Executes programs following the fetch-decode-execute cycle



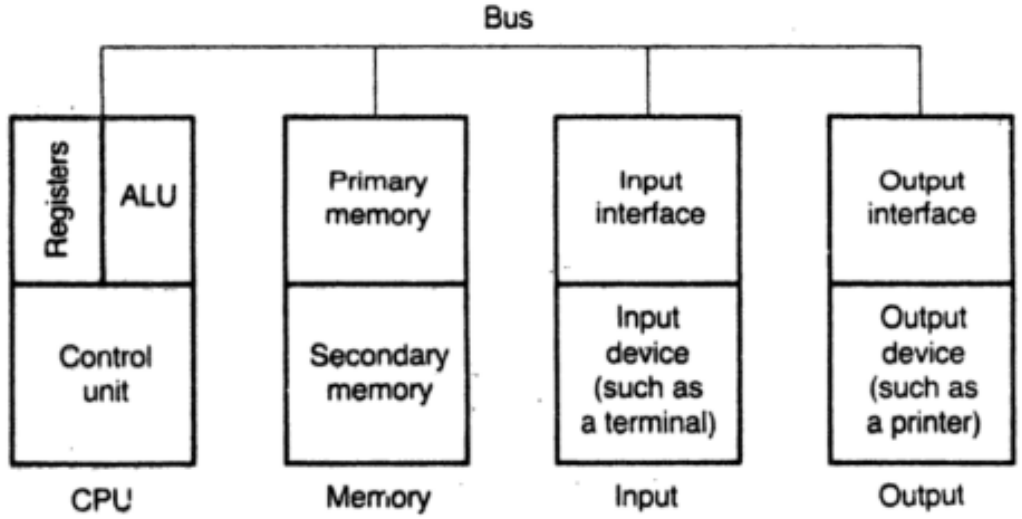
Harvard computer Architecture

- Harvard Architecture is the computer architecture that contains separate storage and separate buses (signal path) for instruction and data.
- It was basically developed to overcome the bottleneck of Von Neumann's Architecture.
- The main advantage of having separate buses for instruction and data is that the CPU can access instructions and read/write data at the same time.



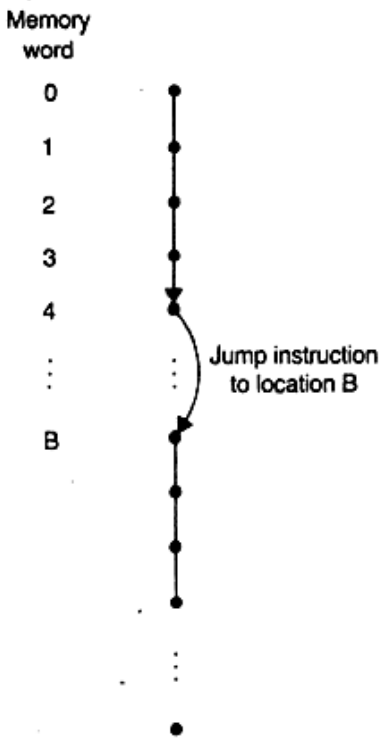
Organization of Stored Program Computer System

- Computer: device capable of processing information by executing a program consisting of several sequence of instructions stored in memory
- This kind is called a stored program computer, because the instructions and data are held in the same memory
- Instruction execution takes place within the CPU and is controlled by the control unit
- **Control Unit**
- Control unit has to perform the following task to carry out the instruction execution process:
  - ✓ instruction interpretation
    - Reads an instruction
    - recognizes the instruction operation
    - retrieves the required data
    - Perform desired operation by activating the ALU
  - ✓ instruction sequencing
    - Determines the address of the next instruction
    - programs are executed in a strictly sequential fashion
    - the next instruction to be executed stored in the consecutive memory locations

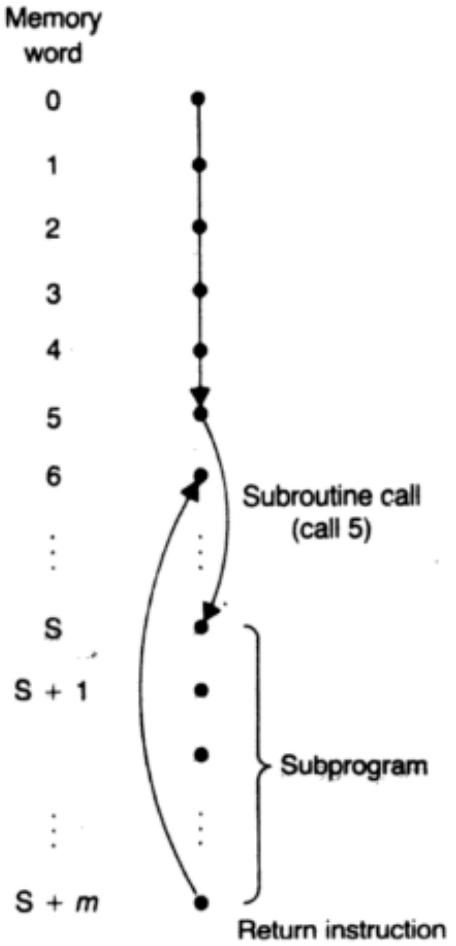


Organization of Stored Program Computer System

- **Typical control Flow sequences -**
  - Branch instruction
  - Subroutine call



a. Branch Instruction



b. Subroutine Call

Organization of Stored Program Computer System

- **CPU**
  - includes set of high speed registers
  - registers may hold data, temporary results or memory address when a computation is in progress
- Common CPU registers
  - **Program counter(PC):** holds address of the instruction to be executed
  - **Instruction register(IR):** holds the instruction currently being executed
  - **Effective address register(EAR):** holds the address of the data to be retrieved from the memory
- Refer to as dedicated registers because they are available for the exclusive use of the control unit and cannot be accessed by a user program
- **Memory Unit**
- Divided into two sections: Primary memory and Secondary memory
- Primary memory
- designed using semiconductor technology
- it is of two types: read write memory and read only memory

Organization of Stored Program Computer System

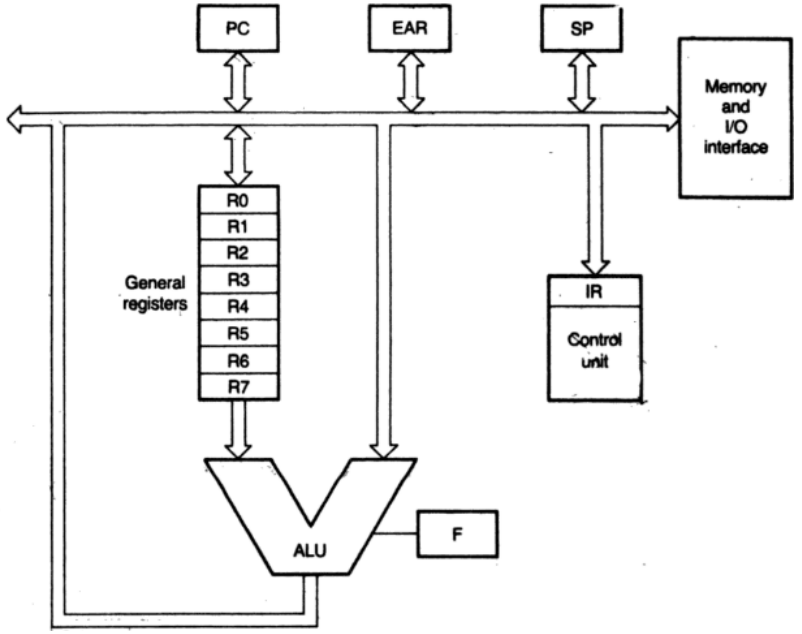
- **Memory Unit**
- Divided into two sections: Primary memory and Secondary memory
- **Primary memory**
  - designed using semiconductor technology
  - It is of two types: read write memory(RWM) and read only memory(ROM)
  - RWM AND ROM are random access memory
- Secondary memory
  - Used to store large amount of data
  - **Examples:** magnetic disks and tapes etc.
- **I/O device :** allow user to communicate with the computer
  - To facilitate communication between the CPU and an IO device we need interface circuitry between them
- System bus
  - The components of a stored program computer systems are connected to each other by a bus
  - three types of buses: ***address bus, data bus and control bus***

Computer structure(Organization)

- Classified into following three groups
  - General register-based processor
  - Accumulator based processor
  - Stack based processor

Computer structure(Organization)

- **General register-based processor**
  - 8 general registers : R0 through R7 can hold
    - Data
    - Memory addresses
    - Result of arithmetic or logic operation
  - **Program counter:** Holds address of the next instruction executed
  - **Effective address register**: Holds address of data to be retrieved from memory
  - **Instruction register:** Holds instruction currently being executed
  - **Stack Pointer:** Address of the top element of stack
  - **F flag register:** holds carry flag or zero flag
  - **Support two- and three-address instructions**



two-address instructions

INSTRUCTION	OPERATION
MOV x, y	y ← (x)
ADD x, y	y ← (x) + (y)
SUB y, x	x ← (x) − (y)
MUL y, x	x ← (x) * (y)
DIV y, x	x ← (x) / (y)

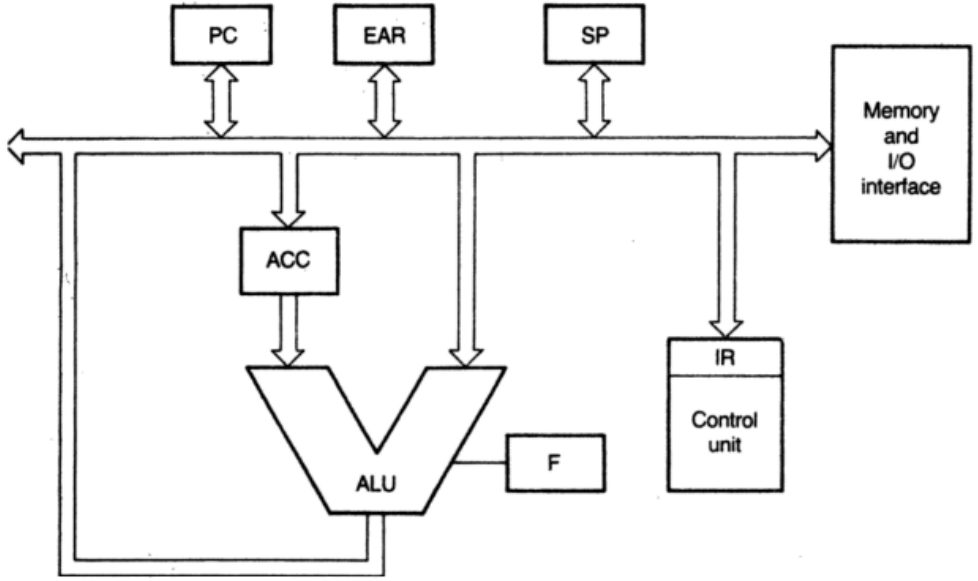
three-address instructions

INSTRUCTION	OPERATION
ADD x, y, z	z ← (x) + (y)
SUB y, x, z	z ← (x) − (y)
MUL x, y, z	z ← (x) * (y)
DIV y, x, z	z ← (x) / (y)

Computer structure(Organization)

Accumulator based processor

- One of the operands is assumed to be held in the accumulator register (ACC) for arithmetic or logic operation
- results of all arithmetic and logic operations are routed to the ACC
- one address instructions are very predominant in this organization



INSTRUCTION		OPERATION
LDA	x	Acc ← (x)
STA	x	x ← (Acc)
ADD	x	Acc ← (Acc) + (x)
SUB	x	Acc ← (Acc) − (x)
MUL	x	Acc ← (Acc) * (x)
DIV	x	Acc ← (Acc) / (x)

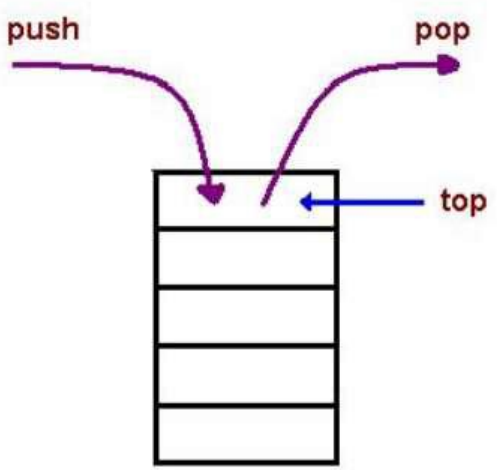
Implementation of assignment statement D = A + B*C using different addressing instruction format

THREE ADDRESS	TWO ADDRESS	ONE ADDRESS
MUL B,C,D ; D←(B)*(C)	MOV B,D ; D←(B)	LDA B ; Acc←(B)
ADD A,D,D ; D←(A)+(D)	MUL C,D ; D←(C)*(D)	MUL C ; Acc←(Acc)*(C)
	ADD A,D ; D←(A)+(D)	ADD A ; Acc←(Acc)+(A)
		STA D ; D←(Acc)

Stack based processor

Stack

- A stack is an ordered linear list in which all insertions and deletions are made at one end, called top.
- It uses Last In First Out (LIFO) principle
- A register is used to store the address of the topmost element of the stack which is known as Stack Pointer(SP)
- The main two operations that are performed on the operands of the stack are: **Push and Pop.**
- These two operations are performed from one end only.
- **Push Operation:** The operation of inserting an item onto a stack
- **Pop Operation:** The operation of deleting an item onto a stack



Introduction

Stack based processor

Stack

- Applications:
  - ✓ Evaluation of mathematical expressions using Reverse Polish Notation.
  - ✓ To implement subroutine calls and returns
  - ✓ to pass parameters from a main program to a subroutine
  - ✓ to handle interrupts
  - ✓ To reverse a word. A given word is pushed to stack-letter by letter-and then popped out letters from the stack.
  - ✓ An undo mechanism in text editors; this operation is accomplished by keeping all text changes in a stack.
  - ✓ **Backtracking:** this is a process when it is required to access the most recent data element in a series of elements.
  - ✓ Language processing
  - ✓ space for parameters and local variables is created internally using a stack.
  - ✓ The compiler's syntax check for matching braces is implemented by using stack.

Stack based processor

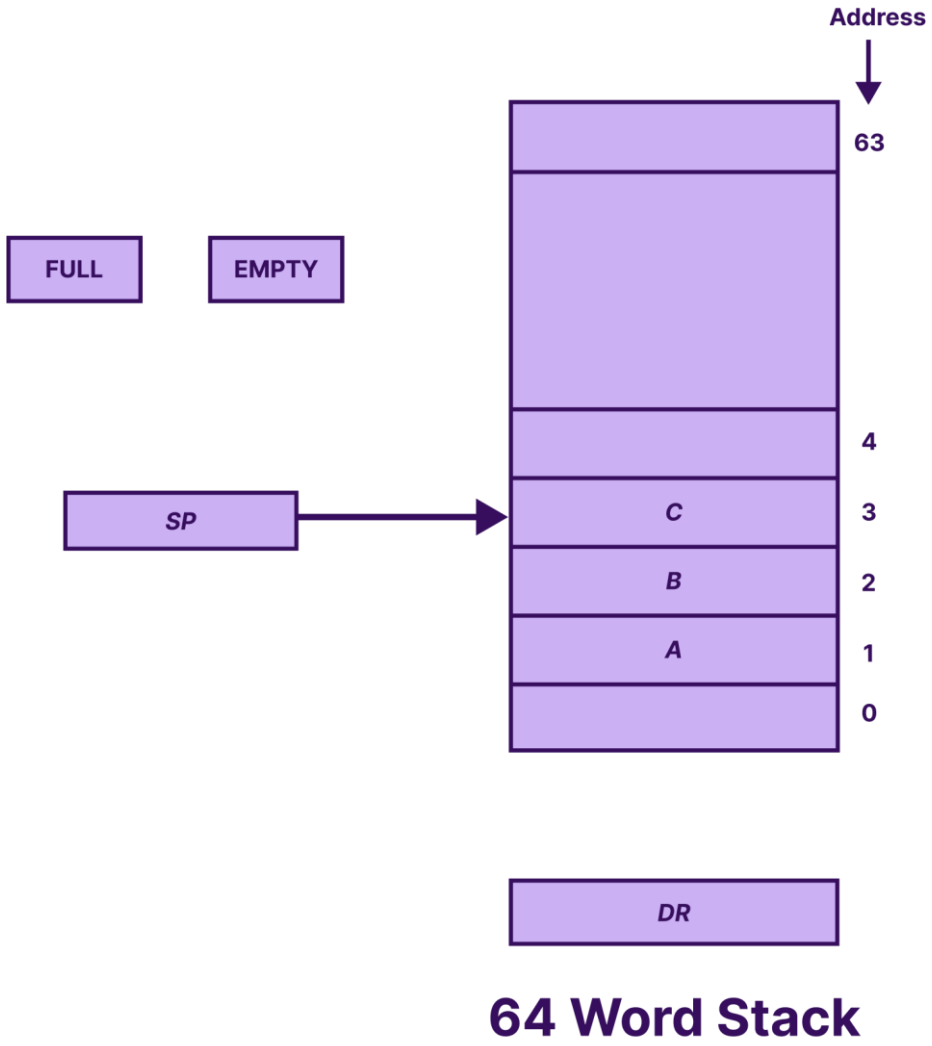
Stack

- There are two types of stack organization which are used in the computer hardware:
- **Register stack:** It is built using register
- **Memory stack:** It is logical part of memory allocated as stack. The logically partitioned part of RAM is used to implement stack.

Stack based processor

Register stack

- The stack can be arranged as a set of memory words or registers.
- Each time an element is added (pushed), the stack pointer is incremented.
- When an element is removed (popped), the pointer is decremented.
- Consider a 64-word register stack arranged as displayed in the figure.
- The stack pointer register includes a binary number, which is the address of the element present at the top of the stack.
- **Stack Pointer (SP):** Points to the top of the stack.
- **Data Register (DR):** Holds data being transferred.
- **Full:** The stack is at maximum capacity and cannot hold more data (stack overflow).
- **Empty:** The stack contains no data, and no items can be popped (stack underflow).

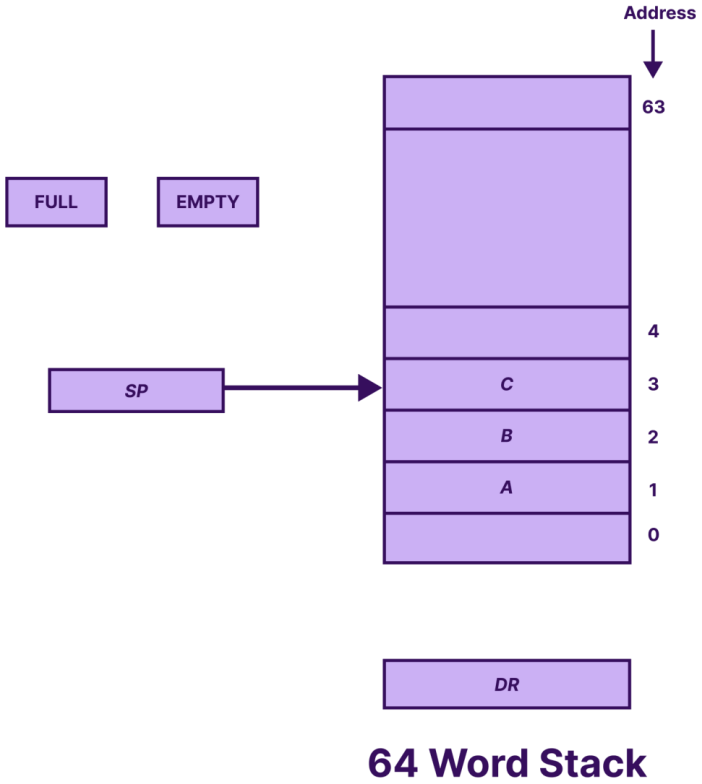


Stack based processor

Register stack

The PUSH operation is implemented with the following sequence of microoperations:

SP ← SP+1	Increment stack pointer
M[SP] ← DR	Write item on top of the stack
If (SP=0) then (FULL←1)	Check if stack is full
EMTY←0	Mark the stack not empty

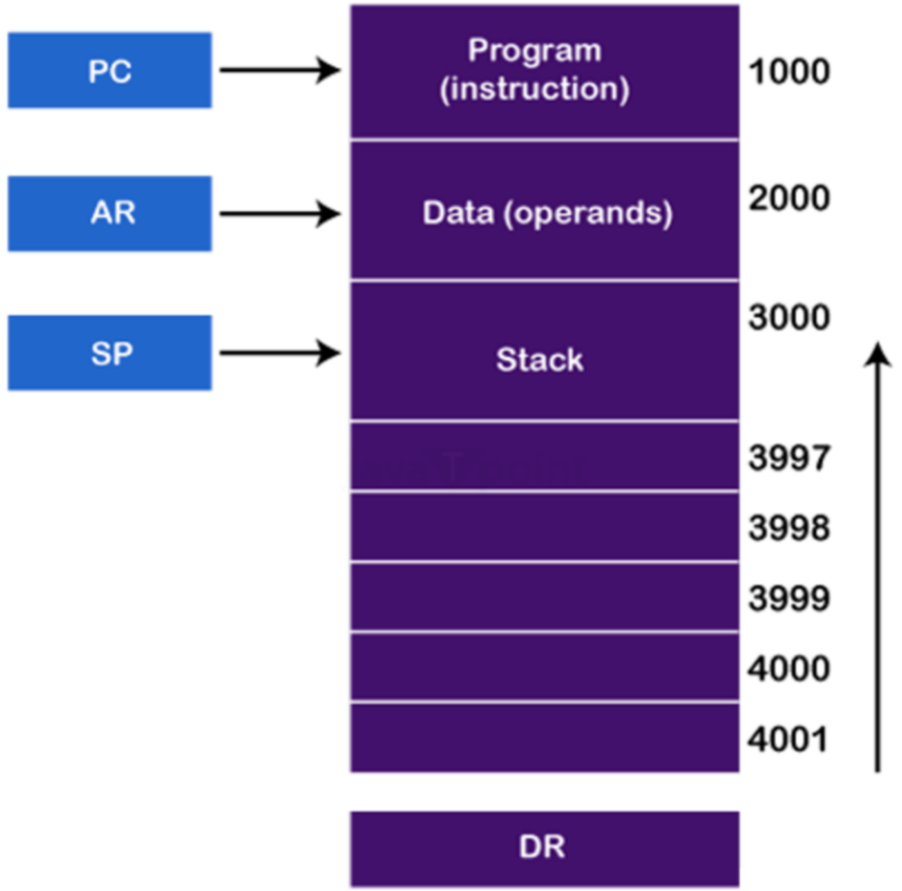


A new item is deleted from the stack if the stack is not empty (if EMTY=0). The POP operation consists of following sequence of microoperations:

DR←M[SP]	Read item from the top of the stack
SP←SP-1	Decrement stack pointer
If(SP=0) then (EMTY←1)	Check if stack is empty
FULL ←0	Mark the stack not full

Computer structure(Organization)

- **Stack based processor**
  - **Memory stack**
    - can be implemented by configuring some portion of the read write memory
    - one of the CPU registers called the ***stack pointer (SP)*** is used to hold the address of the most recently entered item onto the stack
    - The stack pointer register keeps track of the top of the stack, and elements are inserted or removed using this pointer.
    - The stack typically grows downwards, with new elements being pushed at lower memory addresses.
    - **Program Counter (PC):** Holds the address of the next instruction.
    - **Address Register (AR):** Stores memory addresses.
    - **Stack Pointer (SP):** Points to the top of the stack.
    - **Data Register (DR):** Holds data being transferred.



Stack based processor

Memory stack

- **PUSH – writing onto the stack**
- Item can be inserted into the stack by executing PUSH instruction

PUSH ⟨Mem adr⟩

The semantics of this instruction can be formally described as follows:

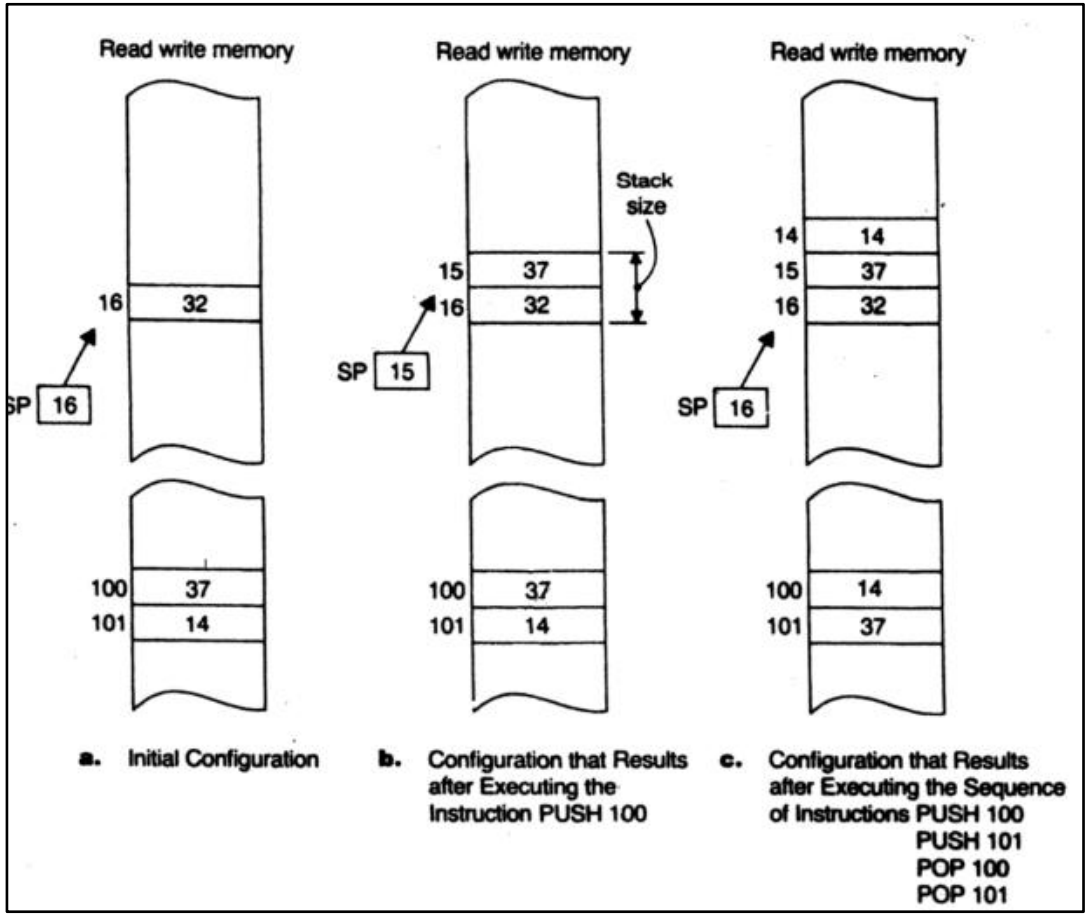
SP ← SP − 1	;	decrement the SP by 1
(SP) ←(⟨Mem adr⟩)	;	copy the contents of the specified memory address into the location whose address is the current contents of the SP

- **POP – reading from the stack**
- Item can be removed from the stack by executing PUSH instruction

POP ⟨Mem Adr⟩

The semantics of this instruction can be formally expressed as follows:

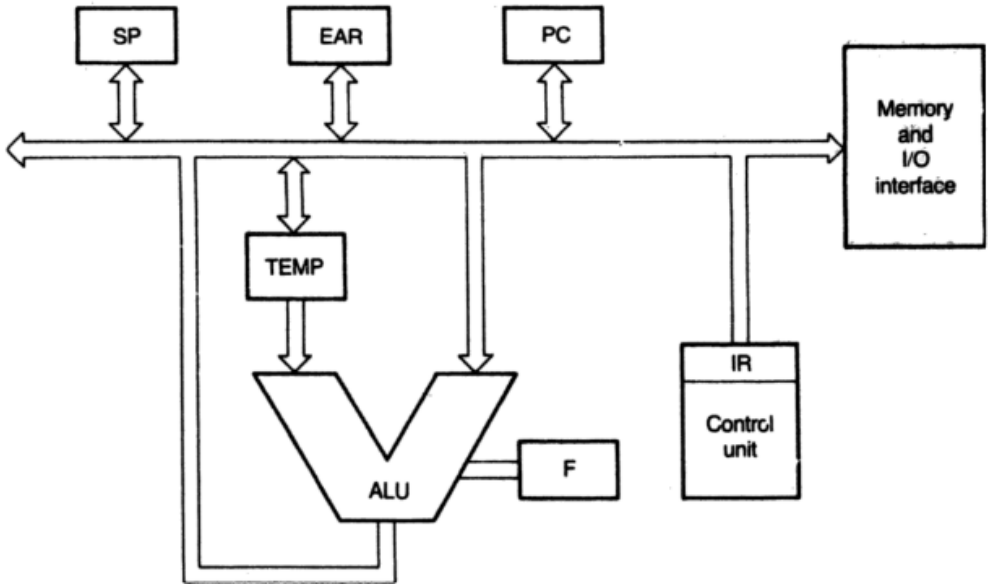
⟨Mem Adr⟩ ← (SP)	:	Copy the contents of the memory location pointed to by the SP into the specified memory address
SP ← SP + 1;		Increment the Sp by 1



Stack based processor/Stack machine

Stack

- In the event of binary arithmetic or logic operation, the first operand will be removed from the stack and held in register TEMP
- the second operand will be directly routed from the stack to the right input of the ALU
- the result of the operation is stored in the top of the stack



Some Typical Stack Machine Instructions

INSTRUCTION	*OPERATION*
PUSH X	(TOS) ← (X)
POP Z	Z ← ((TOS))
ADD	(NOS) ← ((TOS)) + ((NOS))
SUB	(NOS) ← ((TOS)) − ((NOS))
MUL	(NOS) ← ((TOS)) * ((NOS))
DIV	(NOS) ← ((TOS)) / ((NOS))

Note: TOS = top of stack
NOS = next on the stack

Implementation of assignment statement D = A + B*C using stack machine instructions

```
PUSH A
PUSH B
PUSH C
MUL     ; calculate B*C
ADD     ; Add A to B*C
POP D   ; save result.
```

Arithmetic Expression Evaluation

- Expression notation is a way to write mathematical expressions using operators and operands.
- The three most common types of expression notation are infix, prefix, and postfix.
- **Infix notation**
  - ✓ The most common way to write mathematical expressions
  - ✓ Operators are written between operands
  - ✓ **For example:** A + B
  - ✓ Parentheses are used to specify the order of operations
  - ✓ This is the most natural way to write math, but it can be ambiguous
- **Prefix notation**
  - ✓ Also known as **Polish notation (PN)**
  - ✓ Operators are written before operands
  - ✓ **For example:** + A B
  - ✓ Parentheses are not needed because the order of operations is clear
  - ✓ This notation is easy for computers to read, but it can be unintuitive for humans
- **Postfix notation**
  - ✓ Also known as **Reverse Polish notation (RPN)**
  - ✓ Operators are written after operands
  - ✓ **For example:** A B +
  - ✓ Parentheses are not needed because the order of operations is clear
  - ✓ This notation is useful for analyzing and manipulating mathematical formulas

Infix	Prefix	Postfix
A + B * C + D	+ + A * B C D	A B C * + D +
(A + B) * (C + D)	* + A B + C D	A B + C D + *
A * B + C * D	+ * A B * C D	A B * C D * +
A + B + C + D	+ + + A B C D	A B + C + D +

Arithmetic Expression Evaluation

- **Convert the following expressions into RPN:**
  1. X = (A + B + C) / (D –E * F)
  2. X = A * B + C * D + E * F
  3. X = (8 + 2 * 5) / (1 + 3 * 2 –4)
  4. A = (B + C) * D –E
  5. [(A + B) * C + D] / (E + F + G)

Evaluation of Postfix expression using stack

❑**The evaluation rule/algorithm for the postfix expression is stated as:**

1. Read the expression from left to right.
2. If it is an operand then push the element into the stack.
3. If the element is an operator except NOT operator, pop the two operands from the stack and evaluate them with the read operator and push back the result of the evaluation into the stack.
4. If it is the NOT operator then pop one operand from the stack and then evaluate it and push back the result of the evaluation into the stack.
5. Repeat it till the end of stack.

Introduction

- Evaluate the postfix expression: **10 8 4 + * 6 3 / -**

Sr. no.	Expression	Stack	Operations
0	10	10	Push onto stack
1	8	10 8	Push onto stack
2	4	10 8 4	Push onto stack
3	+	10 12	Pop top two elements i.e. 8 and 4 from the stack, add them, then push the result onto stack
4	*	120	Pop top two elements i.e. 10 and 12 from the stack, multiply them, then push the result onto stack
5	6	120 6	Push onto stack
6	3	120 6 3	Push onto stack
7	/	120 2	Pop top two elements i.e. 6 and 3 from the stack, divide them, then push the result onto stack
8	-	118	Pop top two elements i.e. 120 and 2 from the stack, substract them, then push the result onto stack

COMPUTER ORGANIZATION AND ARCHITECTURE

Computer Instruction Set

Dr. Bore Gowda S B
Additional Professor
Dept. of ECE
MIT, Manipal

Introduction

- **Instruction Set Architecture (ISA)**
  - An instruction set architecture (ISA) defines the set of basic operations a computer must support
  - ISA is the part of the processor architecture related to programming, including:
    - ✓ the native data types
    - ✓ Instructions
    - ✓ Registers
    - ✓ addressing modes
    - ✓ memory architecture
    - ✓ interrupt and exception handling
    - ✓ external I/O
  - Essentially, an ISA provides a blueprint that describes how software controls the hardware to perform computations.

Instruction

- A programmable system uses a sequence of instructions to control its operation
- A typical instruction specifies:
  - Operation to be performed
  - Operands to use, and
  - Where to place the result, or
  - Which instruction to execute next
- Instructions are stored in RAM or ROM as a program
- The addresses for instructions in a computer are provided by a program counter (PC) that can
  - Count up
  - Load a new address based on an instruction and, optionally, status information
- The PC and associated control logic are part of the Control Unit
- Executing an instruction - activating the necessary sequence of operations specified by the instruction
- Execution is controlled by the control unit and performed:
  - In the datapath
  - In the control unit
  - In external hardware such as memory or input/output

Instruction Formats

- Computer instructions are a set of machine language instructions that a particular processor understands and executes.
- Instruction is represented as a sequence of bits.
- A computer performs tasks on the basis of the instruction provided.
- An instruction comprises of groups called fields. These fields include:
  - **Operation code (Opcode) field:** which specifies the operation to be performed.
  - **Address field:** contains the location of the operand, i.e., register or memory location.
  - **Mode field:** specifies how the operand will be located.
- Other special fields are sometimes employed under certain circumstances
  - For example a field that gives the number of shifts in a shift-type instruction.

Instruction Formats

- The ***operation code field*** of an instruction is a group of bits that define various processor operations: *add, subtract, complement, and shift*.
- The bits that define the ***mode field*** of an instruction code specify a variety of alternatives for choosing the operands from the given address.
- The various ***addressing modes that*** have been formulated for digital computers
- Operations specified by computer instructions are executed on some data stored in memory or processor registers.
- Operands residing in memory are specified by their memory address.
- Operands residing in processor registers are specified with a register address.
- A register address is a binary number of **k bits** that defines one of $\mathbf{2^k}$ registers in the CPU.
- For example: CPU with 16 processor registers RO through R15 will have a register address field of four bits.
- The binary number 0101, for example, will designate register R5.

Instruction Formats

- Computers may have instructions of several different lengths containing varying number of addresses.
- The number of address fields in the instruction format of a computer depends on the internal organization of its registers.
- Most computers fall into one of three types of CPU organizations:
  1. Single accumulator organization.
  2. General register organization.
  3. Stack organization.
- Depending on number of addresses, the instructions are classified into following categories:
  - Three Address Instructions
  - Two Address Instructions
  - One Address Instruction
  - Zero Address Instructions

Instruction Formats

Three Address Instructions

- Computers with three-address instruction formats can use each address field to specify either a processor register or a memory operand.

- Evaluation of expression using three address instructions: **X = (A + B) • (C + D)**

```
ADD    R1, A, B     R1 ← M[A] + M[B]
ADD    R2, C, D     R2 ← M[C] + M[D]
MUL    X, R1, R2    M[X] ← R1 * R2
```

- It is assumed that the computer has two processor registers, R 1 and R2.
- The symbol M [A] denotes the operand at memory address symbolized by A.
- **Advantage:** it results in short programs when evaluating arithmetic expressions
- **Disadvantage:** the binary-coded instructions require too many bits to specify three addresses

Instruction Formats

Two Address Instructions

- Two-address instructions are the most common in commercial computers.
- Here again each address field can specify either a processor register or a memory word.
- The program to evaluate **X = (A + B) • (C + D)** is as follows:

```
MOV   R1, A     R1 ← M[A]
ADD   R1, B     R1 ← R1 + M[B]
MOV   R2, C     R2 ← M[C]
ADD   R2, D     R2 ← R2 + M[D]
MUL   R1,R2     R1 ← R1*R2
MOV   X, R1     M[X] ← R1
```

- The MOV instruction moves o r transfers the operands t o and from memory and processor registers.
- The first symbol listed in an instruction is assumed to be both a source and the destination where the result of the operation is transferred.

Instruction Formats

One Address Instruction

- One-address instructions use an implied accumulator (AC) register for all data manipulation.
- For multiplication and division there is a need for a second register.
- However, here we will neglect the second register and assume that the AC contains the result of all operations.
- The program to evaluate **X = (A + B) • (C + D)** is

```
LOAD    A    AC ← M[A]
ADD     B    AC ← AC + M[B]
STORE   T    M[T] ← AC
LOAD    C    AC ← M[C]
ADD     D    AC ← AC + M[D]
MUL     T    AC ← AC * M[T]
STORE   X    M[X] ← AC
```

- All operations are done between the AC register and a memory operand.
- T is the address of a temporary memory location required for storing the intermediate result.

Instruction Formats

Zero Address Instructions

- A stack-organized computer does not use an address field for the instructions ADD and MUL.
- The PUSH and POP instructions, however, need an address field to specify the operand that communicates with the stack.
- The following program shows how **X = (A + B) • (C + D)** will be written for a stack organized computer. (TOS stands for top of stack.)

```
PUSH    A    TOS←A
PUSH    B    TOS←B
ADD          TOS←(A + B)
PUSH    C    TOS←C
PUSH    D    TOS←D
ADD          TOS←(C + D)
MUL          TOS←(C + D)*(A + B)
POP     X    M[X]←TOS
```

- The name "***zero-address***" is given to this type of computer because of the absence of an address field in the computational instructions.

Addressing Modes

- **Operation field:** specifies the operation to be performed.
- **Operation** must be executed on some data stored in computer registers or memory words.
- The way the operands are chosen during program execution is dependent on the addressing mode of the instruction.
- The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced.
- Computers use addressing mode techniques for the purpose of accommodating one or both of the following provisions:
  1. To give programming versatility to the user by providing such facilities as pointers to memory, counters for loop control, indexing of data, and program relocation.
  2. To reduce the number of bits in the addressing field of the instruction.

Addressing Modes

- Different types of addressing modes are:
  1. Implied / Implicit Addressing Mode
  2. Immediate Addressing Mode
  3. Register Addressing Mode
  4. Register Indirect Addressing Mode
  5. Auto-Increment or Auto-Decrement Addressing Mode
  6. Direct Addressing Mode
  7. Indirect Addressing Mode
  8. Relative Addressing Mode
  9. Indexed Addressing Mode
  10. Base Register Addressing Mode

Addressing Modes

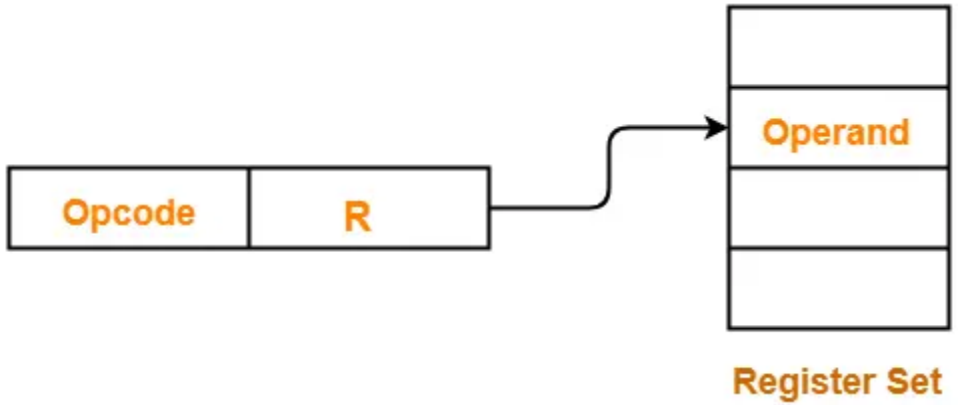
1. Implied / Implicit Addressing Mode

- In this mode the operands are specified implicitly in the definition of the instruction.
- For example, the instruction "complement accumulator“ is an implied-mode instruction because the operand in the accumulator register is implied in the definition of the instruction.
- **Example: CMA**
- In fact, all register reference instructions that use an accumulator are implied-mode instructions
- Zero-address instructions in a stack-organized computer are implied-mode instructions
- **Example: MUL** *Pop top two items from stack and add*

Addressing Modes

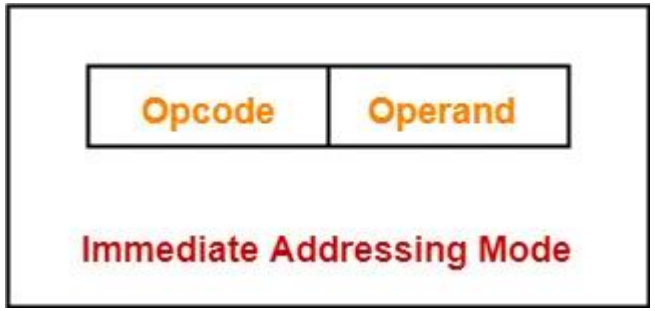
2. Register Addressing Mode

- In this mode the operands are in registers that reside within the CPU.
- The particular register is selected from a register field in the instruction.
- **Example: ADD R0, R1**



3. Immediate Addressing Mode

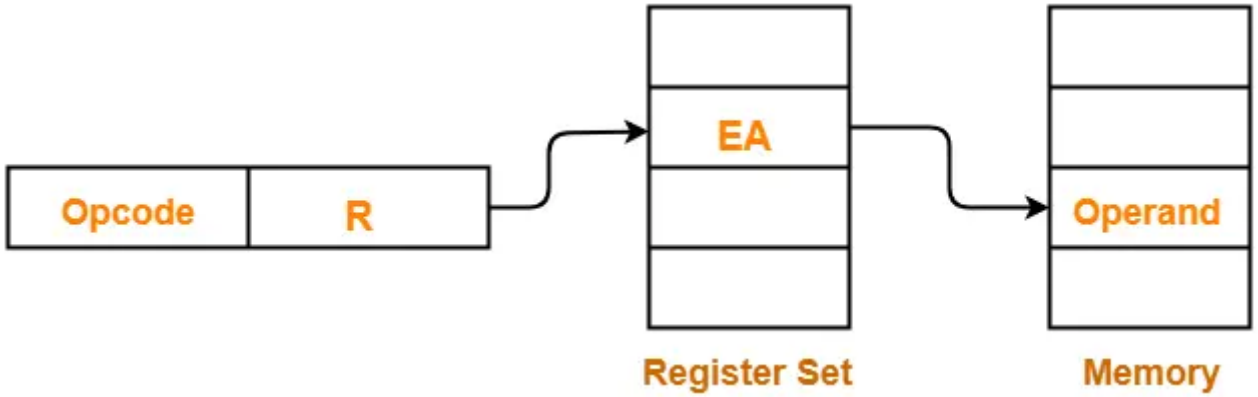
- In this mode the operand is specified in the instruction itself.
- In other words, an immediate-mode instruction has an operand field rather than an address field.
- **Example: ADD A, #0x31**



Addressing Modes

4. Register Indirect Addressing Mode

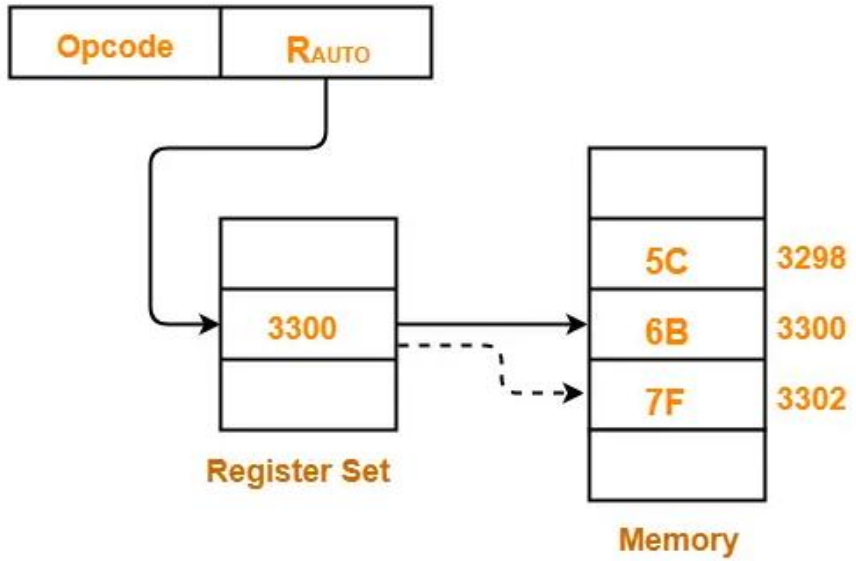
- In this mode the instruction specifies a register in the CPU whose contents give the address of the operand in memory
- This method allows for the execution of the same set of instructions for different memory locations by incrementing the register content and pointing to the new location each time.
- The indirect mode is denoted by enclosing the given register in parentheses.
- In this scenario, the effective address (EA) is the content of the memory location found in the register.
- EA=(R)
- **Example: LOAD R1, (R2)**



Addressing Modes

5. Auto-Increment or Auto-Decrement Addressing Mode

- This is similar to the register indirect mode except that the register is incremented or decremented after (or before) its value is used to access memory.
- When the address stored in the register refers to a table of data in memory, it is necessary to increment or decrement the register after every access to the table.
- This can be achieved by using the increment or decrement instruction.
- The effective address is defined to be the memory address obtained from the computation dictated by the given addressing mode.
- **Example of ARM processor instructions**
- --- Auto pre increment/decrement addressing mode
- LDR r0, [r1, #+/-4]!
- Auto post increment/decrement addressing mode
- LDR r0, [r1], #+/-4

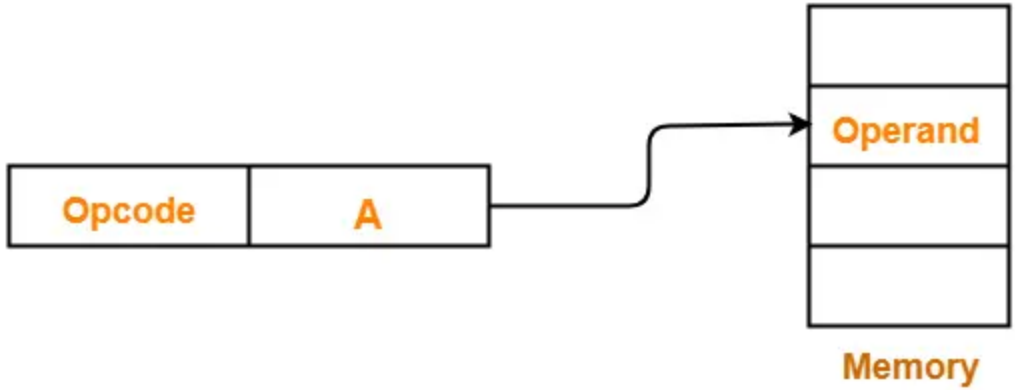


Addressing Modes

6. Direct Addressing Mode

- In this mode the effective address is equal to the address part of the instruction.
- The operand resides in memory and its address is given directly by the address field of the instruction.
- The Direct addressing mode is one that contains the actual address of the data.
- In a branch-type instruction the address field specifies the actual branch address
- The direct addressing mode is also known as the ***absolute addressing mode***.

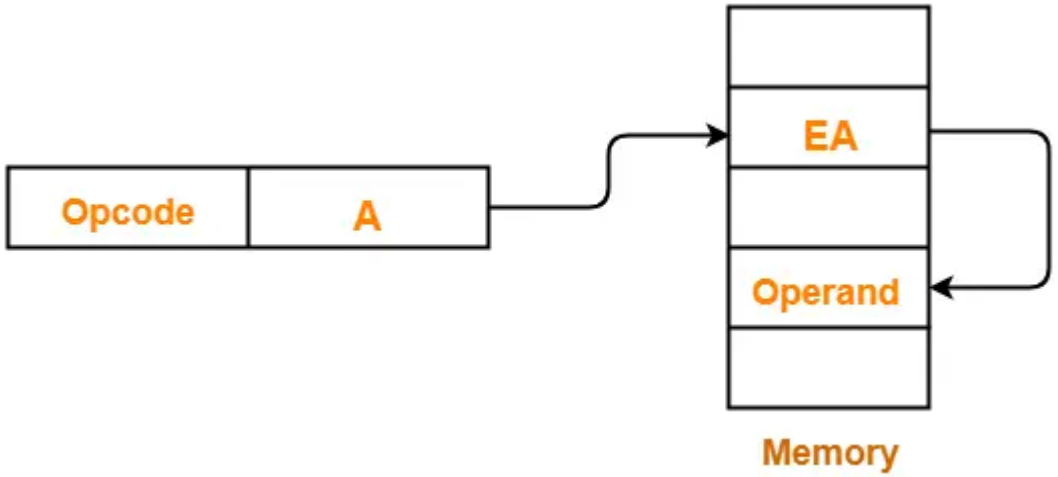
- **Example:**
- MOV A, 81H ; 8051 microcontroller instruction
- JMP 5000H
- MOV R1, @6000H



Addressing Modes

7. Indirect Addressing Mode

- In this mode the address field of the instruction gives the address where the effective address is stored in memory.
- Control fetches the instruction from memory and uses its address part to access memory again to read the effective address.
- Two references to memory are required to fetch the operand.
- **Example:**
- ADD X will increment the value stored in the accumulator by the value stored at memory location specified by X.
- AC ← AC + ((X))



Addressing Modes

- A few addressing modes require that the address field of the instruction be added to the content of a specific register in the CPU.
- The effective address in these modes is obtained from the following computation:

  effective address(EA) = address part of instruction + content of CPU register

- The CPU register used in the computation may be the:
  - ✓program counter
  - ✓An index register, or
  - ✓a base register.
- In either case we have a different addressing mode which is used for a different application.

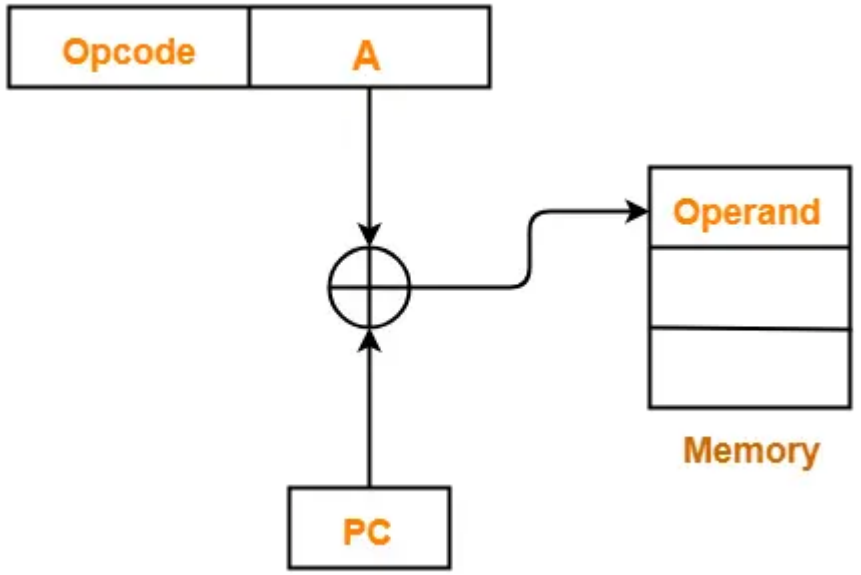
Addressing Modes

8. Relative Addressing Mode

- In this mode the content of the program counter/specified register is added to the address part of the instruction in order to obtain the effective address.
- The address part of the instruction is usually a signed number (in 2' s complement representation) which can be either positive or negative.
- When this number is added to the content of the program counter, the result produces an effective address whose position in memory is relative to the address of the next instruction.

- **Example:**

- JMP 20H ; 8051 instruction

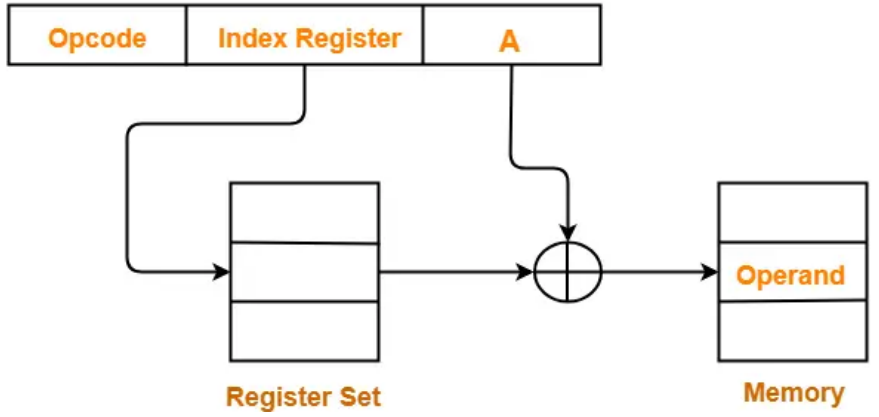


Addressing Modes

9. Indexed Addressing Mode

- In this mode the content of an index register is added to the address part of the instruction to obtain the effective address.
- The index register is a special CPU register that contains an index value.
- The address field of the instruction defines the beginning address of a data array in memory.
- Each operand in the array is stored in memory relative to the beginning address.
- The distance between the beginning address and the address of the operand is the index value stored in the index register.
- **Effective Address = Content of Index Register + Address part of the instruction**

- **Example:**
- MOV AX, [SI+2000H] ; 8086 instructions



Instruction types

- Depending on operation they perform, all instructions are divided in several groups
  - Data transfer instructions
  - Data manipulation instructions
    - ✓ Arithmetic instructions
    - ✓ Logical and bit manipulation instructions
    - ✓ Shift instructions
  - Program control instructions
  - Input/Output Instruction

Instruction types

Data transfer Instructions

- Data transfer instructions transfer the data between
  - ✓ memory and processor registers, processor registers, I/O devices, and from one processor register to another.

Name	Mnemonics	Operations
Load	LD	transfer data from the memory to a processor register, which is usually an accumulator
Store	ST	transfers data from processor registers to memory
Move	MOV	transfers data from processor register to memory or memory to processor register or between processor registers itself
Exchange	XCH	swaps information either between two registers or between a register and a memory word
Input	IN	transfers data between the processor register and the input terminal
Output	OUT	transfers data between the processor register and the output terminal
Push	PUSH	transfer data between a processor register and memory stack
Pop	POP	transfer data between a processor register and memory stack

Instruction types

Arithmetic Instructions

- The four basic arithmetic operations are addition, subtraction, multiplication, and division.
- Most computers provide instructions for all four operations.
- Some small computers have only addition and possibly subtraction instructions.
- The multiplication and division must then be generated by means of software subroutines.

Name	Mnemonics	Operations
Increment	INC	adds 1 to the value stored in the register or memory word
Decrement	DEC	subtracts 1 from the contents stored in the register or memory word
Add	ADD	adds register/memory with accumulator
Subtract	SUB	subtract register/memory with accumulator
Multiply	MUL	multiply register/memory with accumulator
Divide	DIV	Divide register/memory with accumulator
Add with carry	ADDC	adds register/memory with accumulator with carry
Subtract with borrow	SUBB	subtract register/memory with accumulator with borrow
Negate (2's comp)	NEG	2's complement of accumulator

Instruction types

Logical and bit manipulation Instructions

- Logical instructions carry out binary operations on the bits stored in the registers.
- In logical operations, each bit of the operand is treated as a Boolean variable.
- Logical instructions can change bit value, clear a group of bits, or can even insert new bit value into operands that are stored in registers or memory words.

Name	Mnemonics	Operations
Clear	CLR	Clear register or memory
Complement	COM	Complement register or memory
AND	AND	Logical Bitwise AND operations with register or memory
OR	OR	Logical Bitwise OR operations with register or memory
Exclusive-OR	XOR	Logical Bitwise Exclusive OR operations with register or memory
Clear carry	CLRC	Carry ← 0
Set carry	SETC	Carry ← 1
Complement carry	COMC	Carry ← (complement of Carry)
Enable interrupt	EI	Enable maskable interrupt
Disable interrupt	DI	Disable maskable interrupt

Instruction types

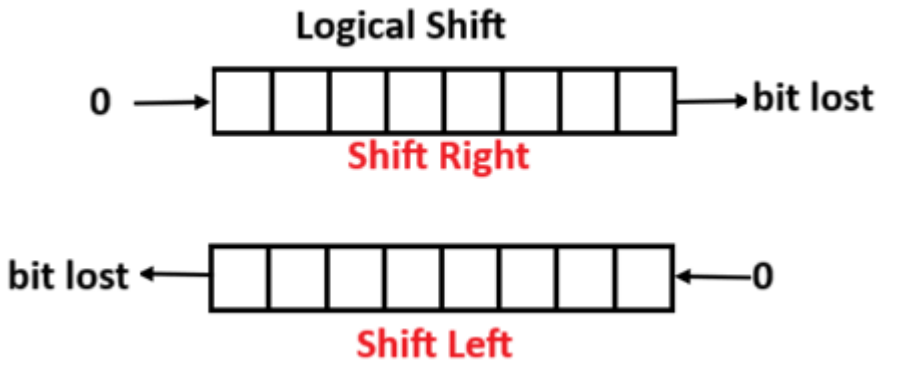
Shift instructions

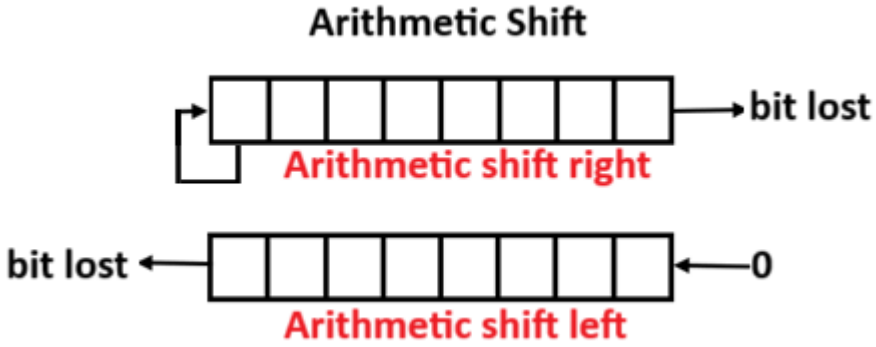
- Shifts are operations in which the bits of a word are moved to the left or right.
- The bit shifted in at the end of the word determines the type of shift used.
- Shift instructions may specify either logical shifts, arithmetic shifts, or rotate-type operations.
- In either case the shift may be to the right or to the left.

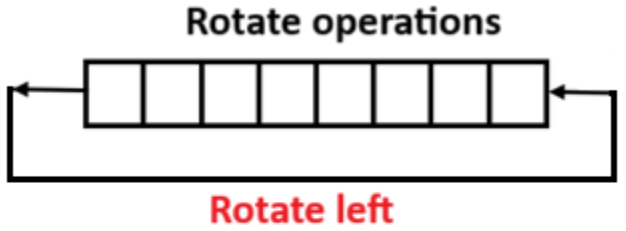
Name	Mnemonics	Operations
Logical shift left	SHL	Shift one bit position to left
Logical shift right	SHR	Shift one bit position to right
Arithmetic shift left	SHLA	Arithmetic shift one bit position to left
Arithmetic shift right	SHRA	Arithmetic shift one bit position to right
Rotate right	ROR	Rotate one bit position to right
Rotate left	ROL	Rotate one bit position to left
Rotate right through carry	RORC	Rotate one bit position to right through carry
Rotate left through carry	ROLC	Rotate one bit position to left through carry

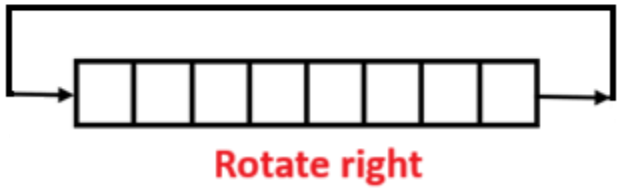
Instruction types

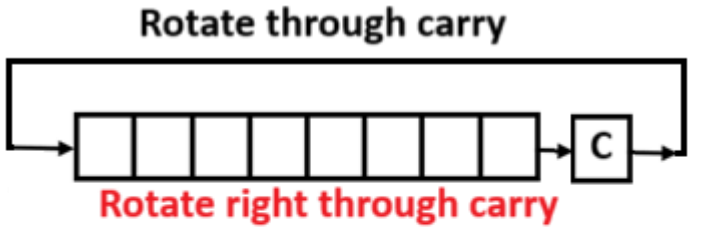
Shift operations

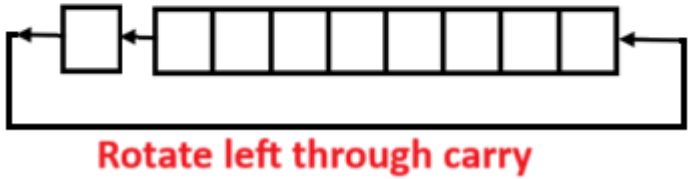












Instruction types

Program control instructions

- Instructions are always stored in successive memory locations.
- When processed in the CPU, the instructions are fetched from consecutive memory locations and executed.
- Each time an instruction is fetched from memory, the program counter is incremented so that it contains the address of the next instruction in sequence.
- A ***program control type of instruction***, when executed, may change the address value in the program counter and cause the flow of control to be altered
- The change in value of ***the program counter*** as a result of the execution of a program control instruction ***causes a break*** in the sequence of instruction execution.
- This is an important feature in digital computers, as it provides control over the flow of program execution and a capability for branching to different program segments.

Instruction types

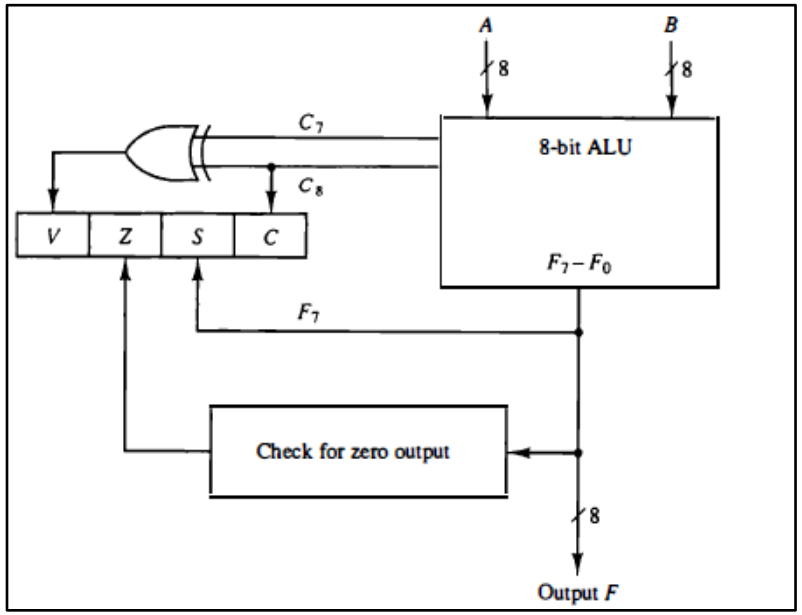
Program control instructions

Name	Mnemonics	Operations
Branch	BR	Jump to a specified memory locations
Jump	JMP	Jump to a specified memory location
Skip	SKP	Skip the next instruction
Call	CALL	Call subroutine
Return	RET	Return from subroutine
Compare (by subtraction)	CMP	Compare two operands
Test (by ANDing)	TST	Logically AND bits in the operands

Instruction types

Status register

- Stores status of the result produced by the ALU used for further analysis.
- Status bits are also called ***condition-code bits or flag bits***.
- The four status bits: C. S, Z, and V.
- The bits are set or cleared as a result of an operation performed in the ALU.



- ✓ ***Bit C (carry):*** set to 1 if the end carry C8 is 1 otherwise it is cleared to 0
- ✓ ***Bit S (sign):*** set to 1 if MSB of the result is 1 otherwise it is cleared to 0
- ✓ ***Bit Z (zero):*** set to 1 if the result contains all O's, otherwise it is cleared to 0
- ✓ ***Bit V (overflow):*** set to 1 if the result exceds the given range otherwise it is cleared to 0

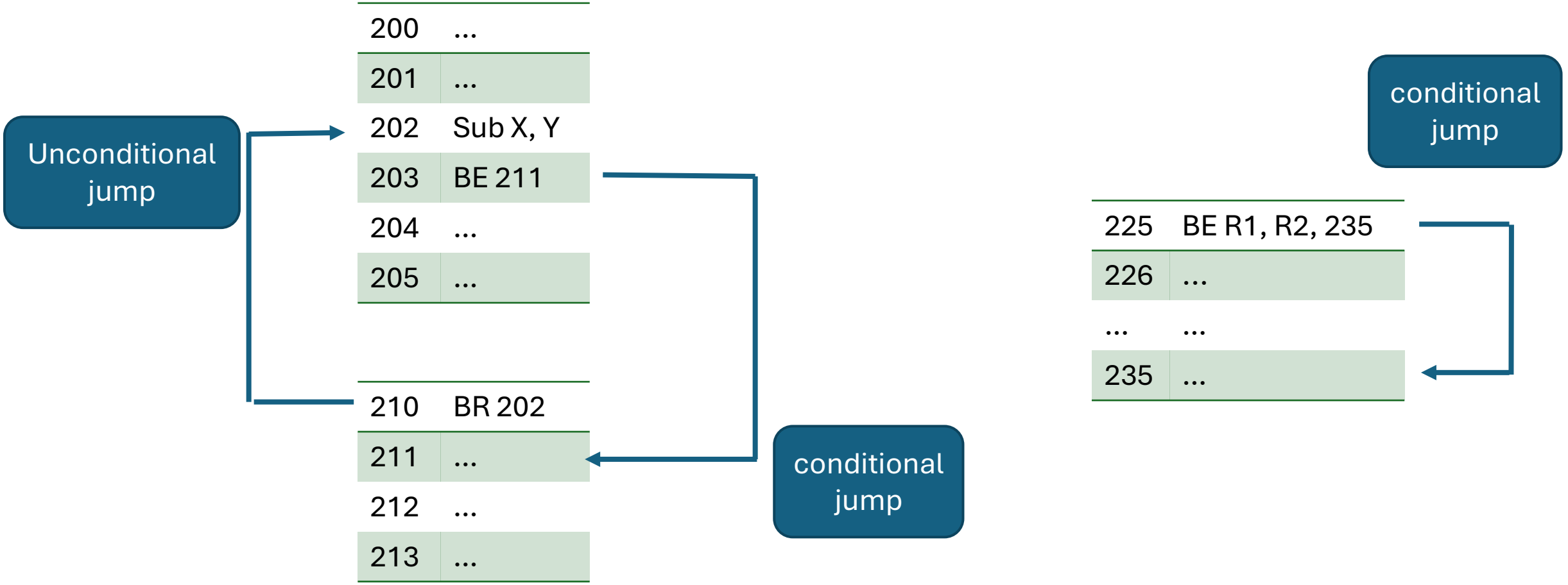
Instruction types

Conditional branch instructions

Mnemonic	Branch condition	Tested condition
BZ	Branch if zero	$Z = 1$
BNZ	Branch if not zero	$Z = 0$
BC	Branch if carry	$C = 1$
BNC	Branch if no carry	$C = 0$
BP	Branch if plus	$S = 0$
BM	Branch if minus	$S = 1$
BV	Branch if overflow	$V = 1$
BNV	Branch if no overflow	$V = 0$
	Unsigned compare conditions ($A - B$)	
BHI	Branch if higher	$A > B$
BHE	Branch if higher or equal	$A \geq B$
BLO	Branch if lower	$A < B$
BLOE	Branch if lower or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$
	Signed compare conditions ($A - B$)	
BGT	Branch if greater than	$A > B$
BGE	Branch if greater or equal	$A \geq B$
BLT	Branch if less than	$A < B$
BLE	Branch if less or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$

Instruction types

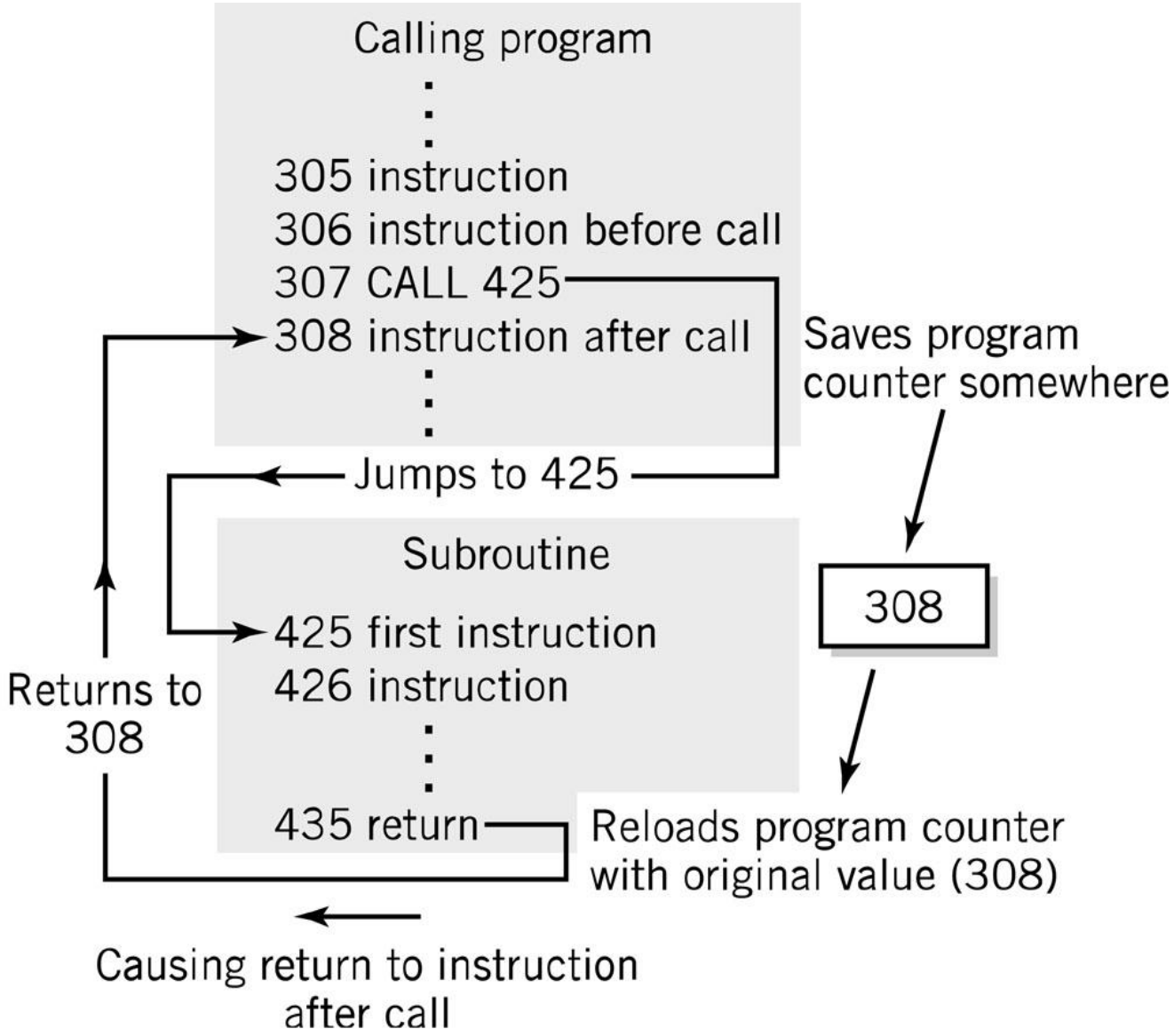
- **unconditional and conditional jumps**



Instruction types

CALL and RET illustration

- Call => Push PC followed by a jump to
- RET => Pop PC



Instruction types

Nested Procedure Calls

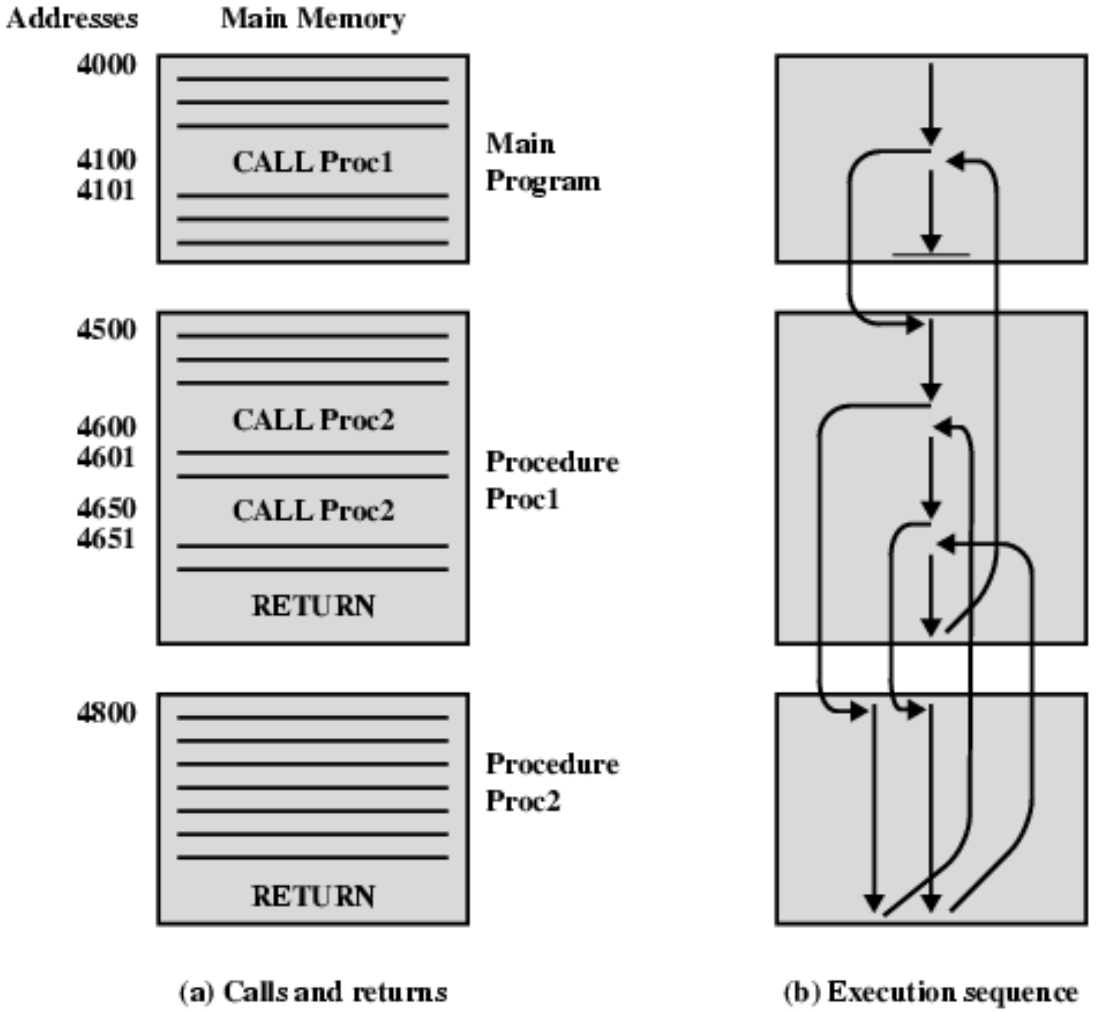
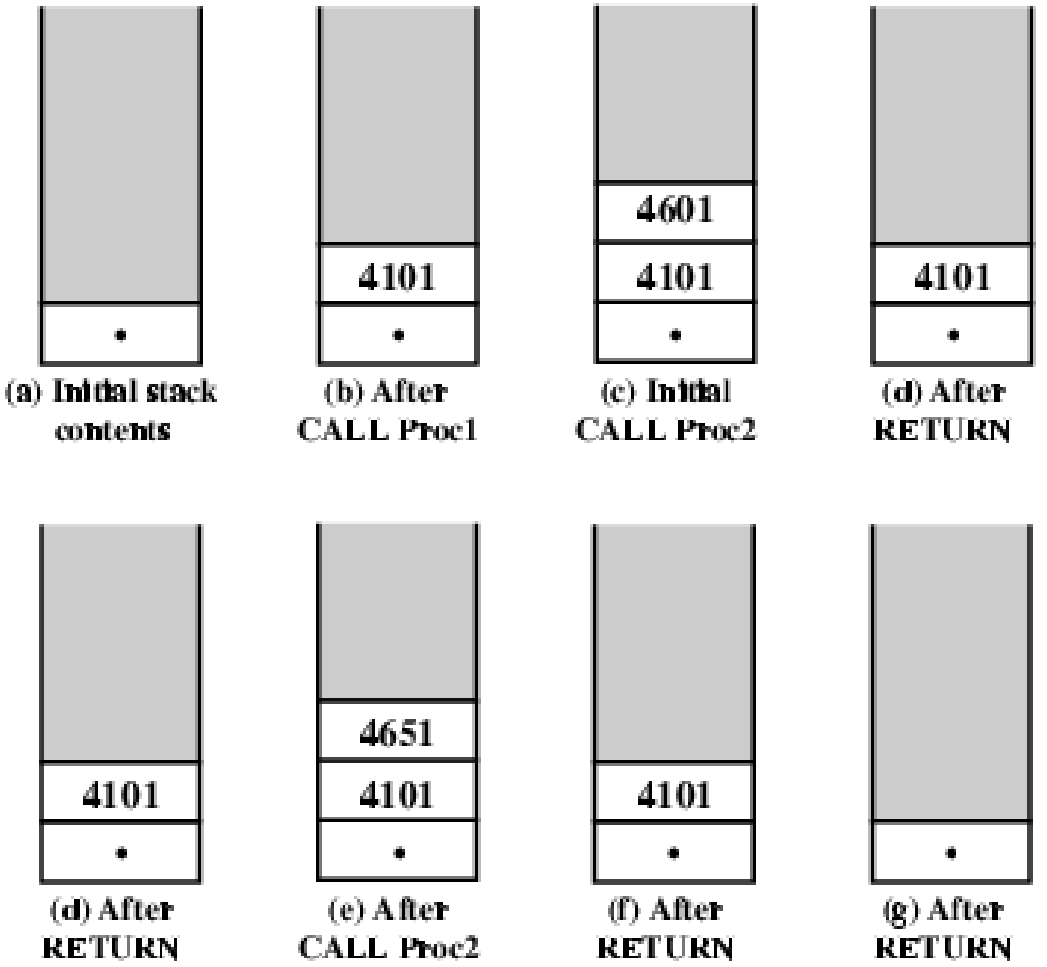


Illustration of function call's effect on stack



Instruction types

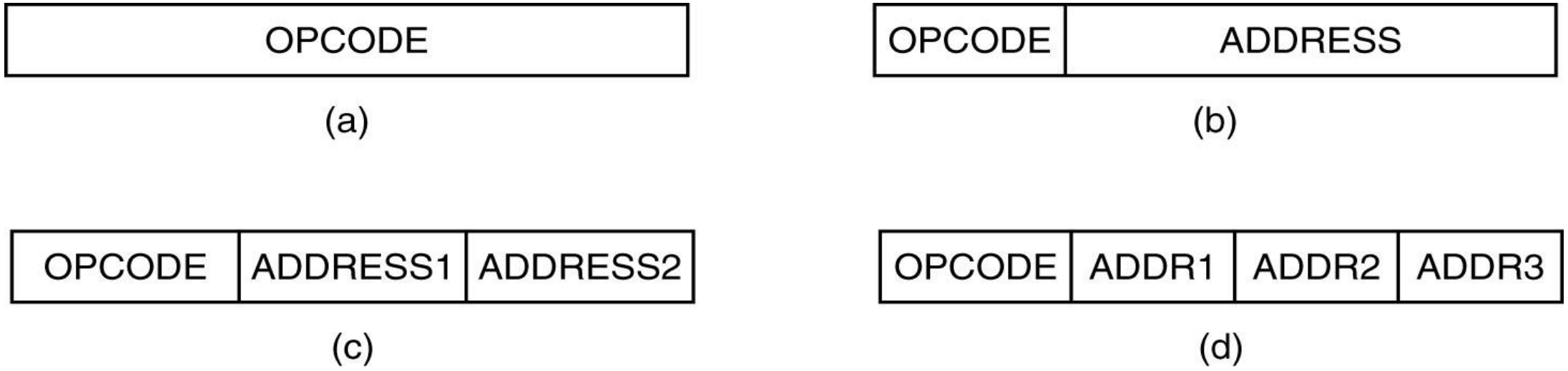
Input/Output Instruction

- The I/O devices are given specific addresses.
- The processor similarly views the I/O operations as memory operations.
- It include the address for the device.
- transfer data to or from the AC register.

Symbol	Description
INP	The INP instruction address the information from the INPR to AC which has 8 low order bits. It also clears the input flag to 0.
OUT	It can send the 8 low order bits from AC into output register OUTPR. It also clears the output flag to 0.

Opcode/Instruction Encoding

- A processor can execute an instruction only if it is represented as a binary sequence
- A unique binary pattern must be assigned to each opcode
- Process is known as opcode encoding
- Some popular upcoding techniques are:
  - Block code technique
  - Expanding opcode technique
  - Huffmann encoding

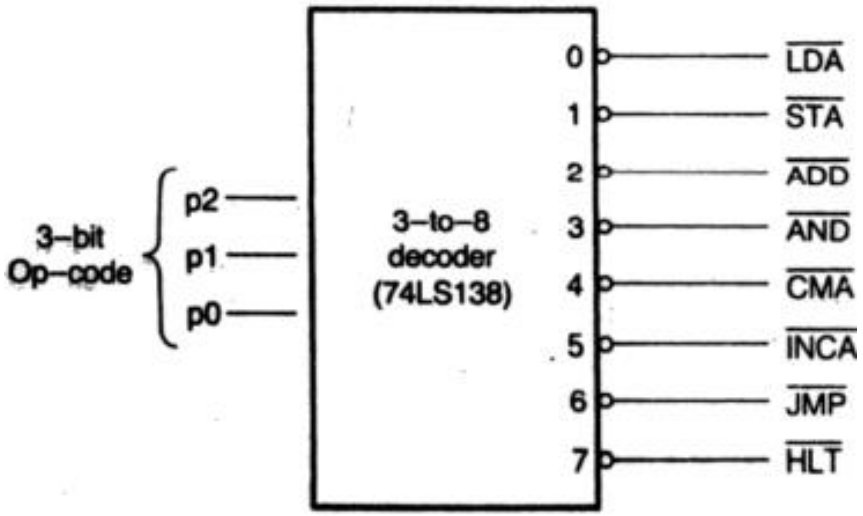


Opcode/Instruction Encoding

Block code technique

- Simplest way to carry out output encoding
- Assign a ***fixed length*** of binary pattern to each opcode
- For example: a k-bit binary pattern can represent 2^k distinct opcodes
- This method is known as a block code encoding
- **Example:** Hypothetical instruction set shown in table has 8 different instructions encoded using 3-bit binary pattern

Op-Code	Binary Pattern
	$p_2p_1p_0$
LDA	000
STA	001
ADD	010
AND	011
CMA	100
INCA	101
JMP	110
HLT	111



Opcode/Instruction Encoding

Expanding opcode technique

- In any instruction set, not all instructions require the same number of operands.
- Operations that require no operands, such as ***HALT***, necessarily waste some space when fixed length instructions are used.
- One way to recover some of this space is to use expanding opcodes.
- The idea of expanding opcodes is to make some opcodes short, but have a means to provide longer ones when needed.
- When the opcode is short, a lot of bits are left to hold operands
- So, we could have two or three operands per instruction
- If an instruction has no operands (such as Halt), all the bits can be used for the opcode
- Many unique instructions are hence available
- In between, there are longer opcodes with fewer operands as well as shorter opcodes with more operands.

Opcode/Instruction Encoding

Expanding opcode technique

- **Example:** consider an instruction set of length 16-bit
- **Option 1**

4-bit	12-bit
Opcode	Address

- Possible operations = $2^4=16$
- Memory locations accessible: $2^{12}=4096$
- **Option 2**

3-bit	13-bit
Opcode	Address

- Possible operations = $2^3=8$
- memory locations accessible: $2^{13}=8192$

Expanding opcode technique

- **Option 3**

5-bit	11-bit
Opcode	Address

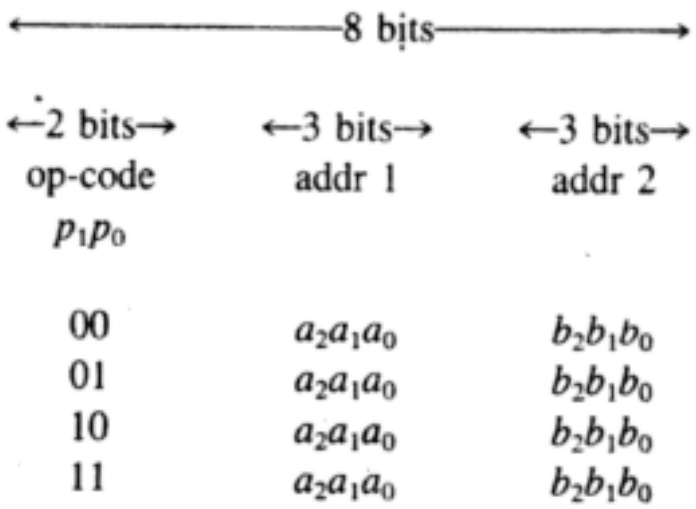
- Possible operations = $2^5=32$
- Memory locations accessible: $2^{11}=2048$
- **Option 2**
- Reduces the no. of possible operations by 50% compared to option 1
- Increase the memory accessibility by 100%
- **Option 3**
- Increases the no. of possible operations by 100% compared to option 1
- Decrease the memory accessibility by 50%

Opcode/Instruction Encoding

Expanding opcode technique

- Instruction format with an instruction length 8-bits and address field is 3-bit.
- **Two address instruction format**

8-bits		
3-bits	3-bits	3-bits
Opcode	**Address 1**	**Address 2**

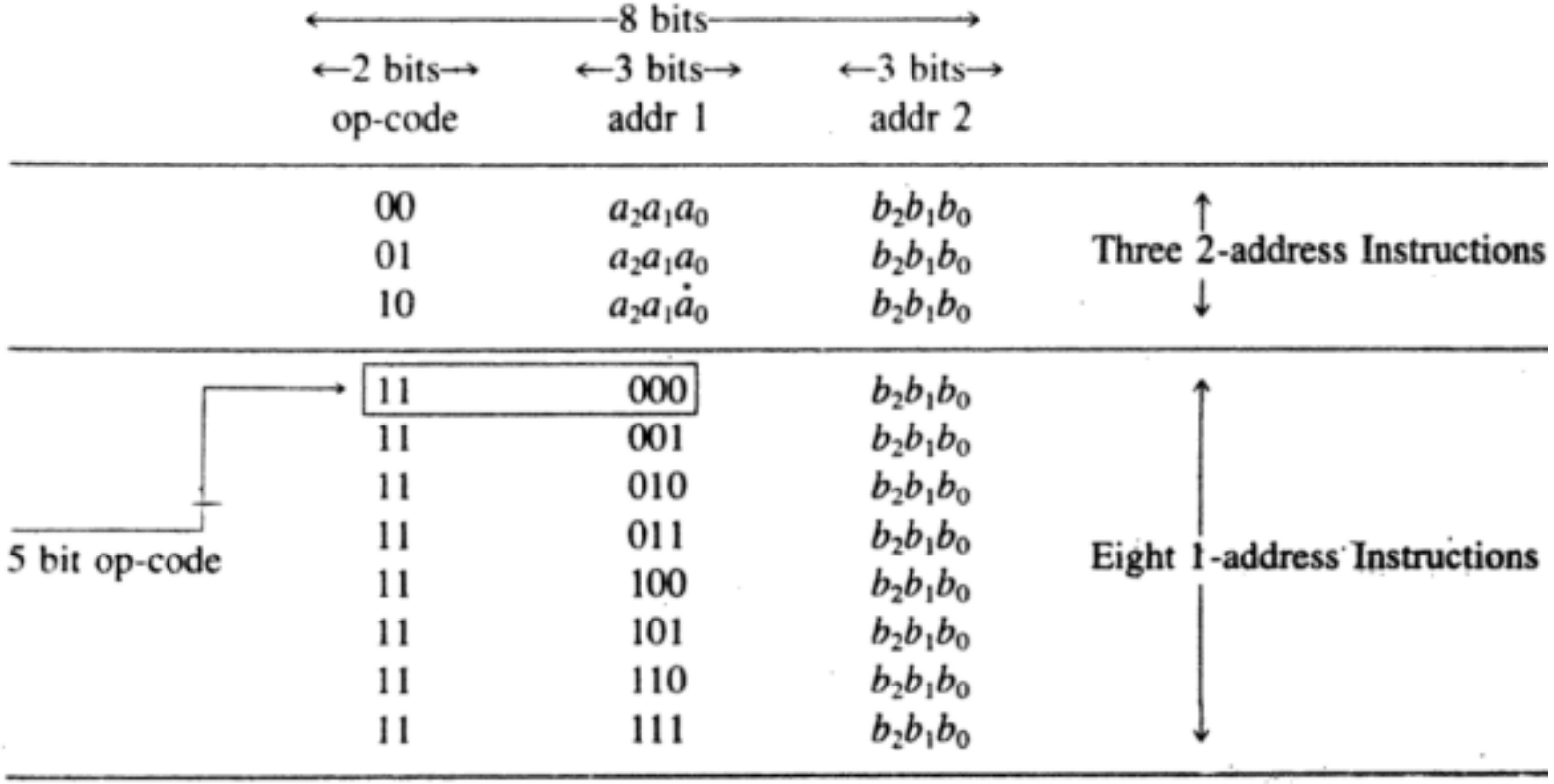


- ***4 two-address instructions can be obtained***

Opcode/Instruction Encoding

Expanding opcode technique

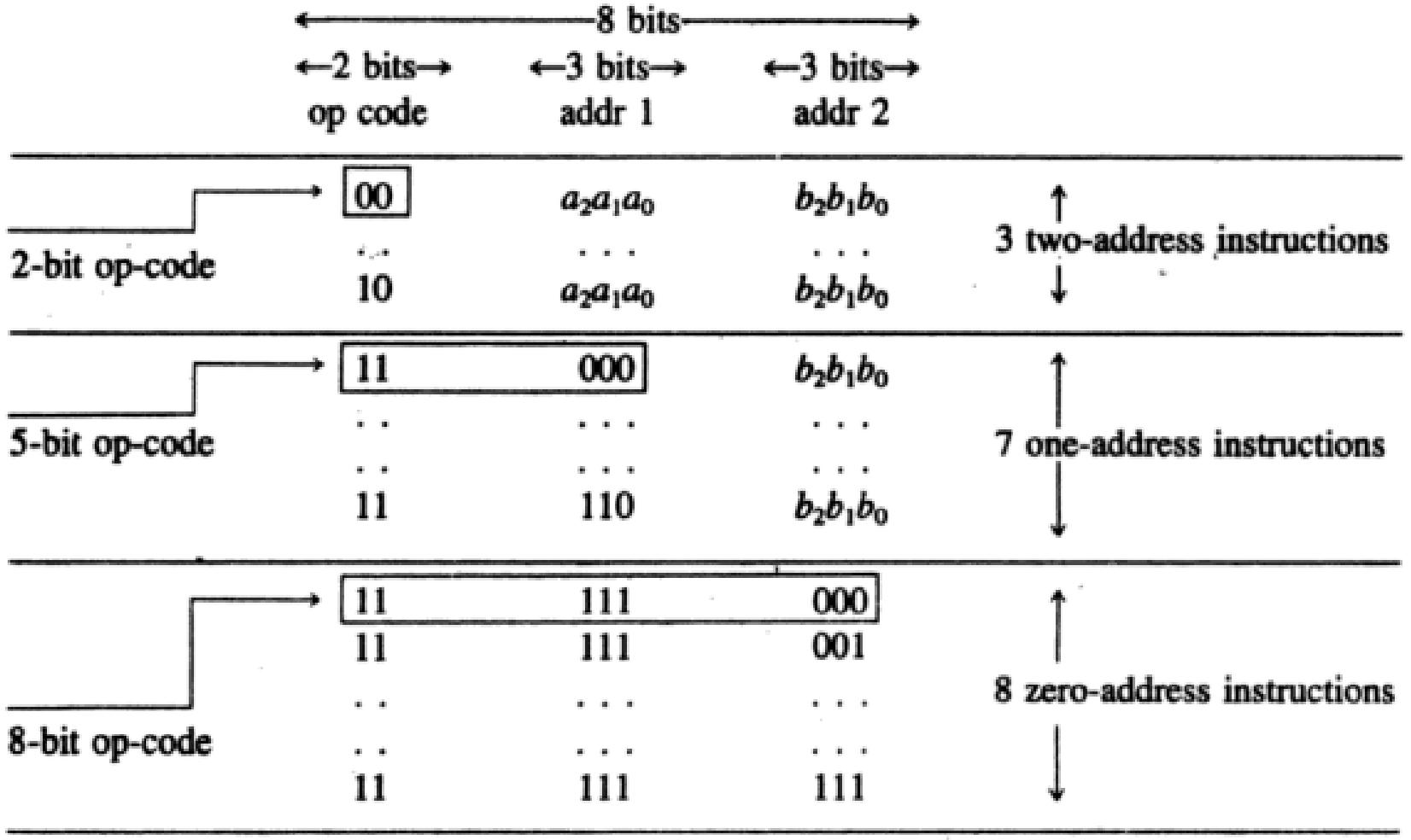
- **Example: 3 two-address instructions and 8 one-address instructions**
- Each one-address instruction requires only one address field, so opcode bits + one address bits are used to form 8 opcodes i.e. opcode for each one-address instruction is 5-bits
- ***This technique is referred to as expanding opcode technique***



Opcode/Instruction Encoding

Expanding opcode technique

- **Example: 3 two-address, 7 one-address and 8 zero address instructions**



Opcode/Instruction Encoding

- **Expanding opcode technique**
  - *How do we know if the instruction set we want is possible when using expanding opcodes?*

Ans: We must determine if we have enough bits to create the desired number of bits patterns

Opcode/Instruction Encoding

Expanding opcode technique

- Consider a machine with 16-bit instructions and 16 registers. We wish to encode the following instructions:
  - ✓ 15 instructions with 3 addresses
  - ✓ 14 instructions with 2 addresses
  - ✓ 31 instructions with 1 address
  - ✓ 16 instructions with 0 addresses

Can we encode this instruction set in 16 bits?

Opcode/Instruction Encoding

Expanding opcode technique

- Having a total of 16-bits we can create 2^{16} = 65536 bit patterns

- 15 instructions with 3 addresses
  - $15x2^4x2^4x2^4 = 15 \times 212 = 61440$ bit patterns
- 14 instructions with 2 addresses
  - $14 \times 2^4 \times 2^4 = 14 \times 2^8 = 3584$ bit patterns
- 31 instructions with 1 address
  - $31 \times 2^4 = 496$ bit patterns
- 16 instructions with 0 addresses
  - The last 16 instructions account for 16-bit patterns

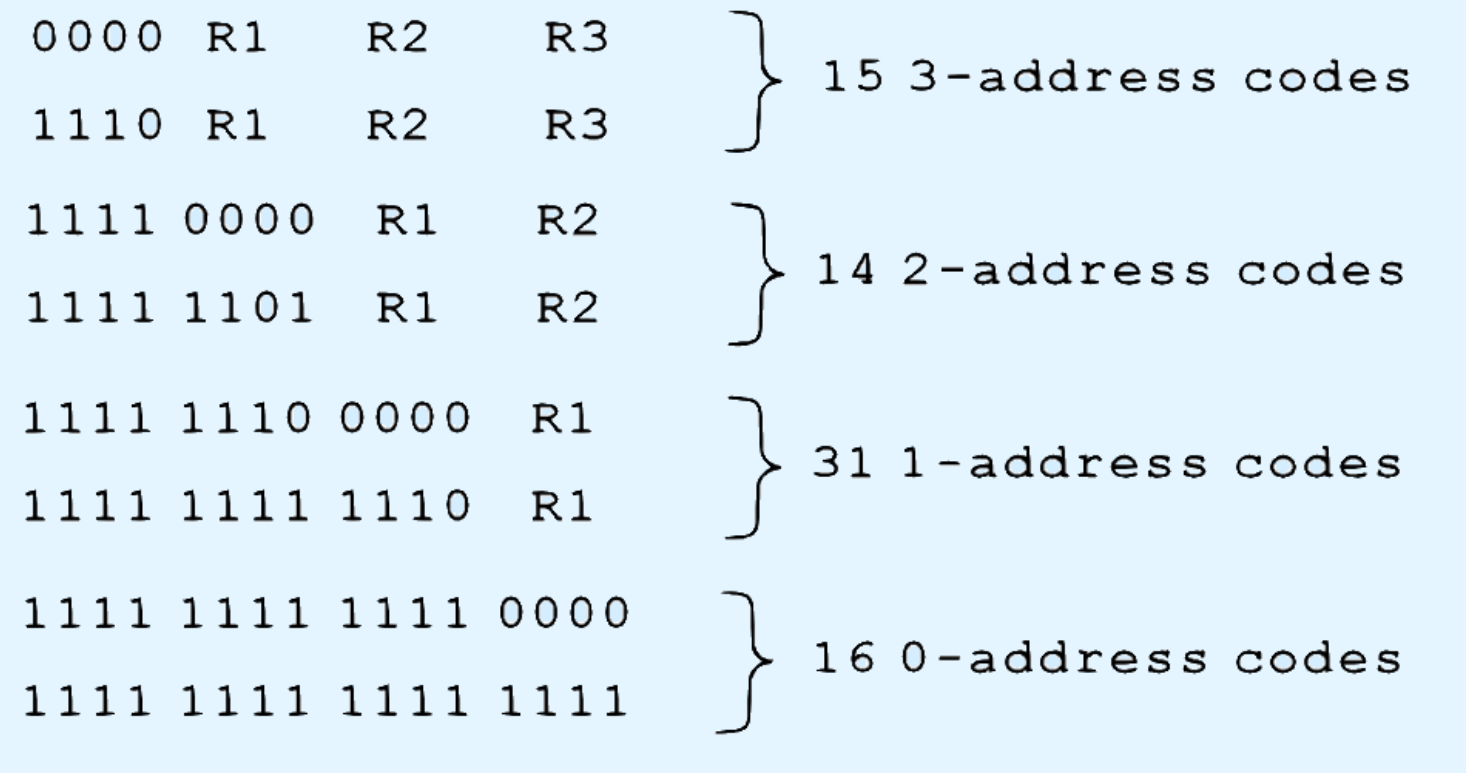
- In total we need 61440 + 3584 + 496 + 16 = 65536 different bit patterns

We have an exact match with no wasted patterns. So our instruction set is possible.

Opcode/Instruction Encoding

Expanding opcode technique

- Consider a machine with 16-bit instructions and 16 registers. We wish to encode the following instructions:
  - ✓ 15 instructions with 3 addresses; 14 instructions with 2 addresses
  - ✓ 31 instructions with 1 address; 16 instructions with 0 addresses



Opcode/Instruction Encoding

Expanding opcode technique

- Is it possible to design an expanding opcode to allow the following to be encoded with a 12-bit instruction? Assume a register operand requires 3 bits.
  - ✓ 4 instructions with 3 registers
  - ✓ 255 instructions with 1 register
  - ✓ 16 instructions with 0 register

- **Solution:**
  - The first 4 instructions account for: 4x23x23x23= 4 x 29= 2048 bit patterns
  - The next 255 instructions account for: 255 x 23= 2040 bit patterns
  - The last 16 instructions account for: 16 bit patterns

  - In total we need 2048 + 2040 + 16 = 4104 bit patterns
  - With 12-bit instruction, we can only have 2^{12} = 4096 bit patterns

Required bit patterns (4104) is more than what we have (4096), so this instruction set is not possible with only 12 bits.

Opcode/Instruction Encoding

Huffman encoding technique

- The block code technique assumes all instructions are used with equal probability
- In practice, not all instructions are used with the same relative frequency count
- On an average, 40% of the instructions used in a program are load and store instructions
- It is required an encoding scheme that will encode the opcode of the ***most frequently used instructions with fewer bits*** and the ***least frequently used instructions with more bits***
- This allows the average number of bits required to encode a typical program to be optimal
- **Huffman coding techniques is used to encode instructions with variable bit lengths**
- It uses variable length encoding.
- It assigns variable length code to all the instructions.
- The code length of an instruction depends on how frequently it occurs in the given text.
- The instruction which occurs most frequently encoded with fewer bits.
- The instruction which occurs least frequently encoded with more bits

Opcode/Instruction Encoding

Building Huffman tree

1. Sort the instructions in ascending order of frequency.
2. Create a leaf node for every unique instruction and its frequency from the given table of instruction set.
3. Extract the two minimum frequency nodes and the sum of these frequencies is made the new root of the tree.
4. Make the first extracted node its left child and the other extracted node as its right child.
5. Repeat **steps 3 and 4** until we left with only one node. The remaining node is the root node and the tree is complete.

Opcode/Instruction Encoding

Building Huffman tree

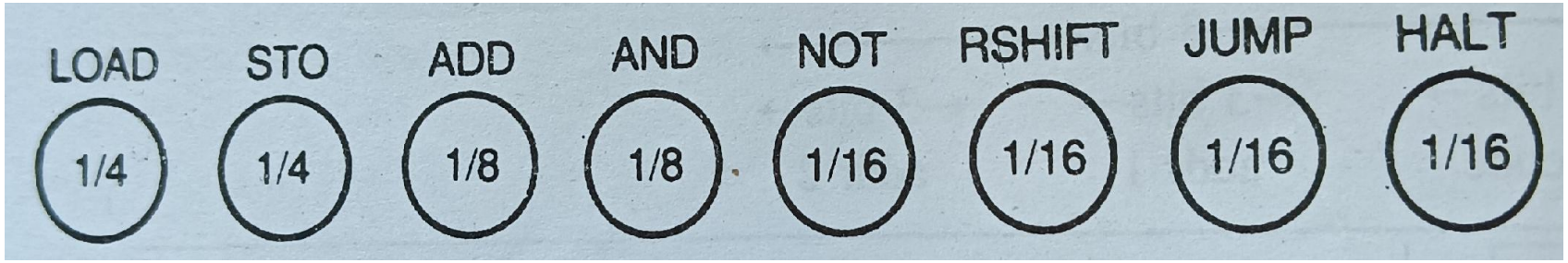
- **Encode the following hypothetical instructions using huffman coding technique**

Instruction	Probability
LOAD	1/4
STO	1/4
ADD	1/8
AND	1/8
NOT	1/16
RSHIFT	1/16
JUMP	1/16
HALT	1/16

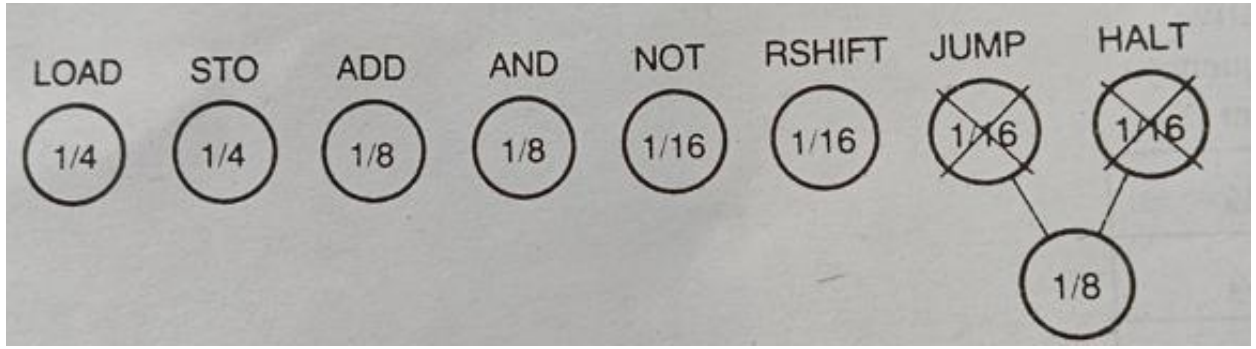
Opcode/Instruction Encoding

Building Huffman tree

- **Arrange the instructions in the ascending order**



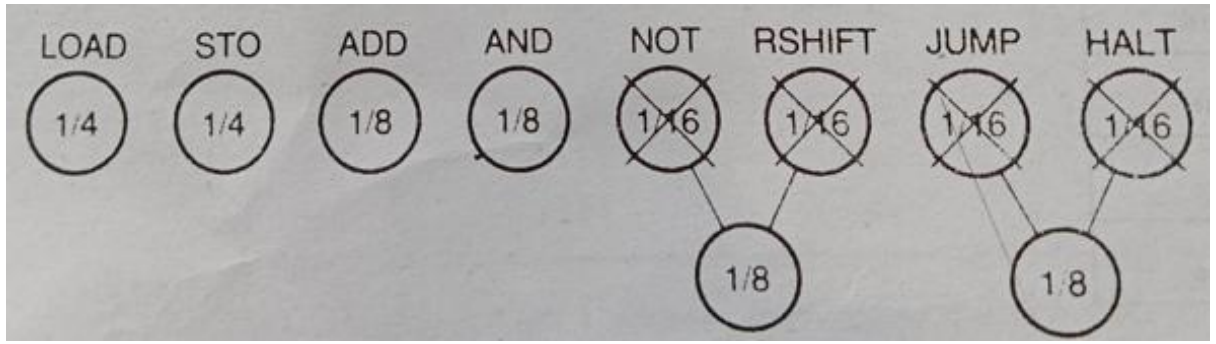
- Select the leaf nodes with least probability and create a new node and assign the weight



Opcode/Instruction Encoding

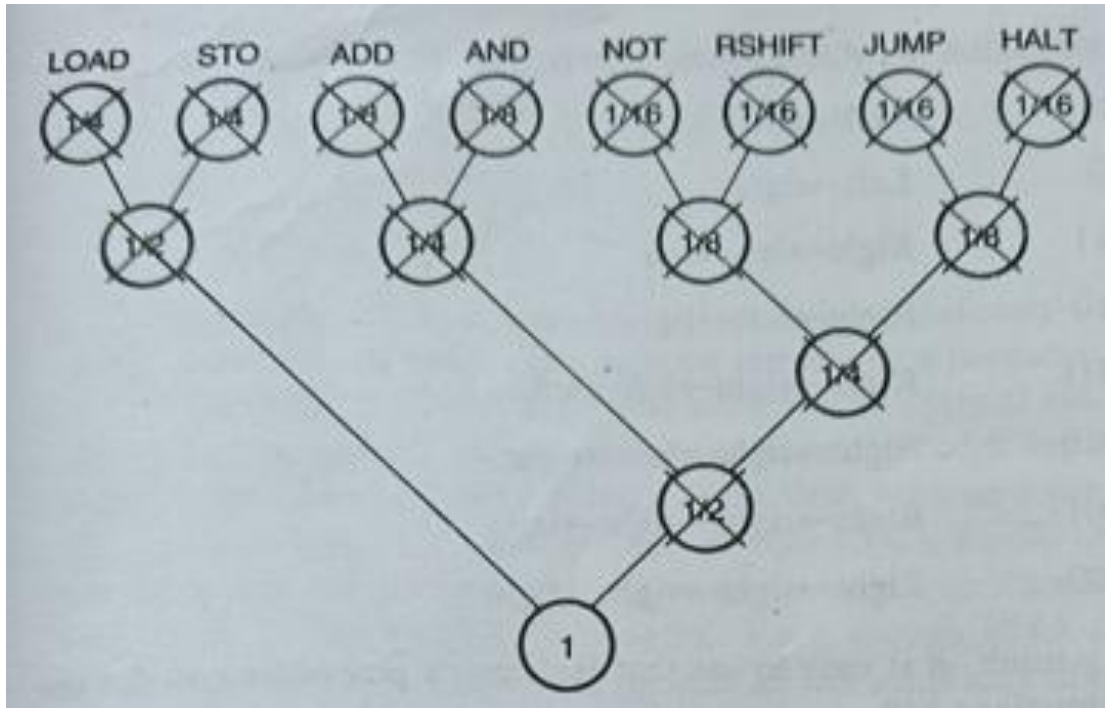
Building Huffman tree

- Select the next two leaf nodes with least probability and create a new node and assign the weight



Building Huffman tree

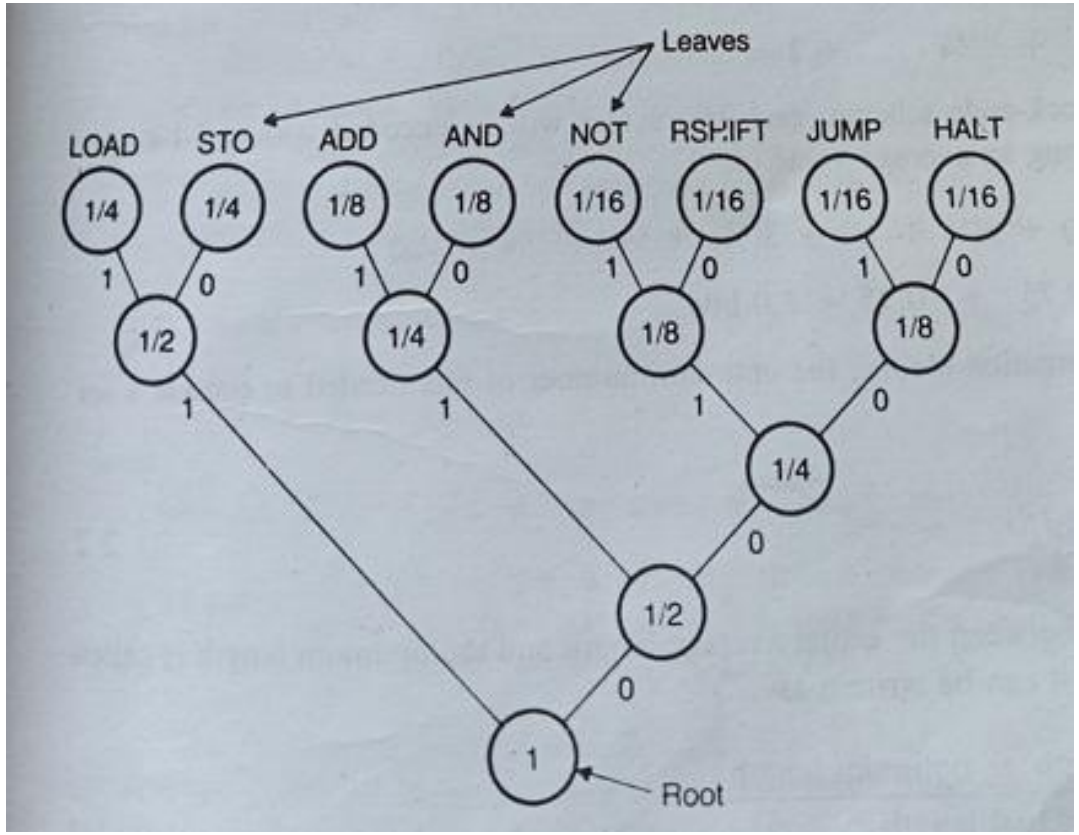
- Continue till only one node left



Opcode/Instruction Encoding

Building Huffman tree

- Label the left and right branches of the tree
- **Assign '1' to the left branches**
- **Assign '0' to the right branches**



Building Huffman tree

- Determining the opcode
- **Path from the root node to the desired leaf node will give you the opcode for the instruction**

MNEMONIC	OP-CODE	PATH FROM THE ROOT
LOAD	11	Left→left
STO	10	Left→right
ADD	011	Right→left→left
AND	010	Right→left→right
NOT	0011	Right→right→left→left
RSHIFT	0010	Right→right→left→right
JUMP	0001	Right→right→right→left
HALT	0000	Right→right→right→right

Opcode/Instruction Encoding

Huffman encoding

- Average number of bits per instructions using Huffman technique

$$Average\ no.of\ bits_{Huffman} = \sum_{i=1}^{n} l_i f_i = 2.75 \text{ bits}$$

- Where l_i is opcode length and f_i relative frequency of the ith instruction

- Average number of bits per instructions using Block code technique

$$Average\ no.of\ bits_{Block\ code} = \sum_{i=1}^{n} l_i f_i = 3.0 \text{ bits}$$

- Optimum number of bits needed to encode

$$Optimum\ length = -\sum_{i=1}^{n} f_i log_2(f_i) = 2.75 \text{ bits}$$

- Redundancy bits calculated as

$$R = \frac{actual\ length - optimum\ length}{actual\ length}$$

$$R_{Huffman} = 0 \quad \text{and} \quad R_{Block\ code} = \frac{1}{12} or\ 8.33\%$$

Opcode/Instruction Encoding

Exercise:

- The hypothetical instruction set of a computer consists of eight instructions, I_0, I_1, I_2,,,,,,I_7. The relative frequency of these instructions are as follows:

Instruction	Relative frequency
I_0	1/4
I_1	1/8
I_2	1/8
I_3	1/8
I_4	1/8
I_5	1/8
I_6	1/16
I_7	1/16

- Encode these instructions using Huffman's method
- Calculate the redundancy introduced by Huffman's and Block code scheme

COMPUTER ORGANIZATION AND ARCHITECTURE

Data Representation

Dr. Bore Gowda S B
Additional Professor
Dept. of ECE
MIT, Manipal

Introduction

- A *bit* is the most basic unit of information in a computer.
  - It is a state of “on” or “off” in a digital circuit.
  - Sometimes these states are “high” or “low” voltage instead of “on” or “off..”
- A *byte* is a group of eight bits.
  - A byte is the smallest possible *addressable* unit of computer storage.
  - The term, “addressable,” means that a particular byte can be retrieved according to its location in memory.

Positional Numbering Systems

- Bytes store numbers using the position of each bit to represent a power of 2.
  - The binary system is also called the base-2 system.
  - Our decimal system is the base-10 system. It uses powers of 10 for each position in a number.
  - Any integer quantity can be represented exactly using any **base (or *radix*).**

Positional Numbering Systems

- The decimal number 947 in powers of 10 is:

$$9 \times 10^{2} + 4 \times 10^{1} + 7 \times 10^{0}$$

- The decimal number 5836.47 in powers of 10 is:

$$5 \times 10^{3} + 8 \times 10^{2} + 3 \times 10^{1} + 6 \times 10^{0} + 4 \times 10^{-1} + 7 \times 10^{-2}$$

Positional Numbering Systems

- The binary number 11001 in powers of 2 is:

$$1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

$$= 16 + 8 + 0 + 0 + 1 = 25$$

- When the radix of a number is something other than 10, the base is denoted by a subscript.
  - Sometimes, the subscript 10 is added for emphasis:

$$11001_2 = 25_{10}$$

Converting Between Bases

- **Converting 190 to base 2**

$190_{10} = 10111110_2$

2	190	
2	95	0
2	47	1
2	23	1
2	11	1
2	5	1
2	2	1
2	1	0
	0	1

Converting Between Bases

- **Converting 0.8125 to binary . . .**
  - You are finished when the product is zero, or until you have reached the desired number of binary places.
  - Our result, reading from top to bottom is:

    $0.8125_{10} = 0.1101_2$
  - This method also works with any base. Just use the target radix as the multiplier.

```
  .8125
 ×   2
------
1.6250

  .6250
 ×   2
------
1.2500

  .2500
 ×   2
------
0.5000

  .5000
 ×   2
------
1.0000
```

Positional Numbering Systems

Decimal	Binary	Hexadecimal	Octal
0	00000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17

Numbering Systems

Numbers we can represent using binary representations

- **Positive numbers**
  - Unsigned binary
- **Negative numbers**
  - Two's complement
  - Sign/magnitude numbers

What about **fractions**?

Numbering Systems

Two common notations:

- **Fixed-point:** binary point fixed
- **Floating-point:** binary point floats to the right of the most significant 1

Numbering Systems

- 6.75 using 4 integer bits and 4 fraction bits:

  01101100

  0110.1100

  $2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75$

  - Binary point is implied
  - The number of integer and fraction bits must be agreed upon beforehand

- Represent 7.5_{10} using 4 integer bits and 4 fraction bits.

01111000

Numbering Systems

Signed Integer Representation

- To represent signed integers, computer systems allocate the high-order bit to indicate the sign of a number.
  - The high-order bit is the leftmost bit. It is also called the most significant bit.
  - 0 is used to indicate a positive number;
  - 1 indicates a negative number.
- The remaining bits contain the value of the number (but this can be interpreted different ways)

Signed Integer Representation

- There are three ways in which signed binary integers may be expressed:
  - Signed magnitude
  - One's complement
  - Two's complement
- In an 8-bit word, *signed magnitude* representation places the absolute value of the number in the 7 bits to the right of the sign bit.

Signed Integer Representation

- For example, in 8-bit signed magnitude representation:

  +3 is: `00000011`

  - 3 is: `10000011`

- Computers perform arithmetic operations on signed magnitude numbers in much the same way as humans carry out pencil and paper arithmetic.
  - Humans often ignore the signs of the operands while performing a calculation, applying the appropriate sign after the calculation is complete.

Signed Integer Representation

- Binary addition is as easy as it gets. You need to know only four rules:

```
0 + 0 =  0      0 + 1 =  1
1 + 0 =  1      1 + 1 = 10
```

- The simplicity of this system makes it possible for digital circuits to carry out arithmetic operations.

- **Example:**
  - Using signed magnitude binary arithmetic, find the sum of 75 and 46.
- First, convert 75 and 46 to binary, and arrange as a sum, but separate the (positive) sign bits from the magnitude bits.

```
0   1001011
0 + 0101110
```

Signed Integer Representation

Disadvantage of signed magnitude representation

- Signed magnitude representation is easy for people to understand, but it requires complicated computer hardware.
- Another disadvantage of signed magnitude is that it allows two different representations for zero: positive zero and negative zero.
- For these reasons (among others) computers systems employ *complement systems* for numeric value representation

Signed Integer Representation

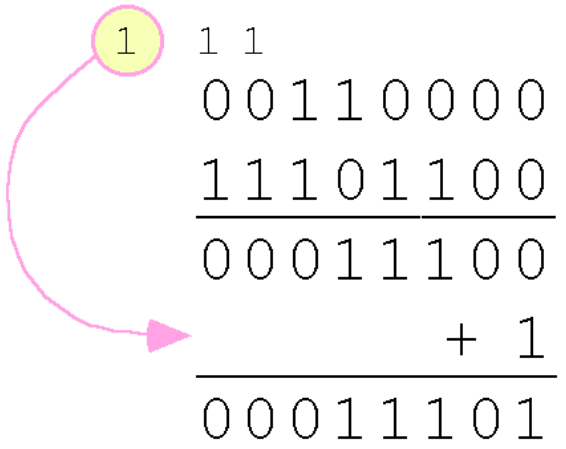
- In complement systems, negative values are represented by some difference between a number and its base.
- The *diminished radix complement* of a non-zero number *N* in base *r* with *d* digits is $(r^d - 1) - N$
- In the binary system, this gives us *one's complement*. It amounts to little more than flipping the bits of a binary number.

Signed Integer Representation

- For example, using 8-bit one's complement representation:
  + 3 is: `00000011`
  - 3 is: `11111100`
- In one's complement representation, as with signed magnitude, negative values are indicated by a 1 in the high order bit.
- Complement systems are useful because they eliminate the need for subtraction. The difference of two values is found by adding the minuend to the complement of the subtrahend.

Signed Integer Representation

- With one's complement addition, the carry bit is "carried around" and added to the sum.
  - Example: Using one's complement binary arithmetic, find the sum of 48 and - 19



We note that 19 in binary is `00010011`,
so -19 in one's complement is: `11101100`.

Signed Integer Representation

- Although the “end carry around” adds some complexity, one’s complement is simpler to implement than signed magnitude.
- But it still has the disadvantage of having two different representations for zero: positive zero and negative zero.
- Two’s complement solves this problem.
- Two’s complement is the radix complement of the binary numbering system; the *radix complement* of a non-zero number N in base r with d digits is $r^d - N$.

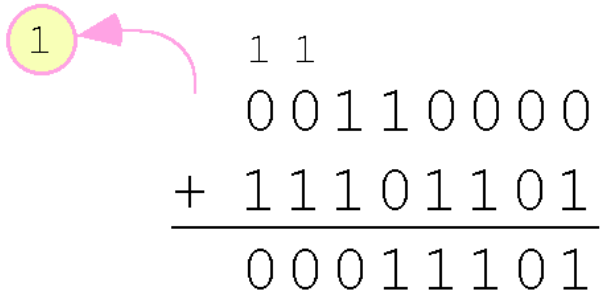
Signed Integer Representation

- To express a value in two's complement representation:
  - If the number is positive, just convert it to binary and you're done.
  - If the number is negative, find the one's complement of the number and then add 1.
- Example:
  - In 8-bit binary, 3 is: `00000011`
  - -3 using one's complement representation is: `11111100`
  - Adding 1 gives us -3 in two's complement form: `11111101`.

Signed Integer Representation

- With two's complement arithmetic, all we do is add our two binary numbers. Just discard any carries emitting from the high order bit.

  – Example: Using two's complement binary arithmetic, find the sum of 48 and - 19.



We note that 19 in binary is: 00010011
so -19 using one's complement is: 11101100
and -19 using two's complement is: 11101101

Signed Integer Representation

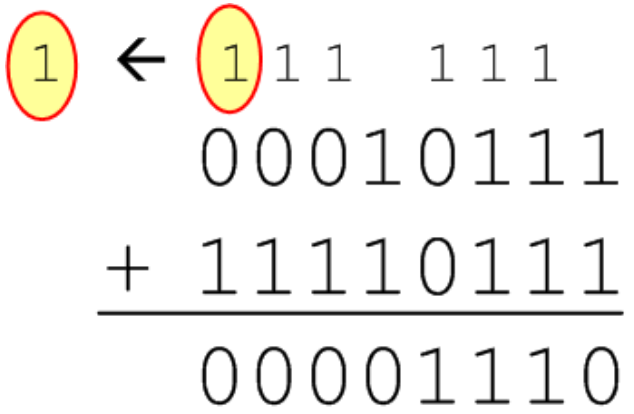
- When we use any finite number of bits to represent a number, we always run the risk of the result of our calculations becoming too large or too small to be stored in the computer.
- While we can't always prevent overflow, we can always *detect* overflow.
- In complement arithmetic, an overflow condition is easy to detect.
- **Example:**
  - Using two's complement binary arithmetic, find the sum of 107 and 46.
  - We see that the nonzero carry from the seventh bit *overflows* into the sign bit, giving us the erroneous result: 107 + 46 = -103.

```
  1 1   1 1 1
  01101011
+ 00101110
----------
  10011001
```

But overflow into the sign bit does not always mean that we have an error.

Signed Integer Representation

- Example:
  - Using two's complement binary arithmetic, find the sum of 23 and -9.
  - We see that there is carry into the sign bit and carry out. The final result is correct: 23 + (-9) = 14.



Rule for detecting signed two's complement overflow: When the "carry in" and the "carry out" of the sign bit differ, overflow has occurred. If the carry into the sign bit equals the carry out of the sign bit, no overflow has occurred.

Floating-Point Representation

- The signed magnitude, one's complement, and two's complement representation that we have just presented deal with signed integer values only.
- **Without modification**, these formats are not useful in scientific or business applications that deal with real number values.
- Floating-point representation solves this problem.
- The Floating point representation is a way to the encode numbers in a format that can handle very large and very small values.
- It is based on scientific notation where numbers are represented as a fraction and an exponent.
- In computing, this representation allows for trade-off between *range* and *precision*.

Floating-Point Representation

- **Need for Floating Point Representation**
- The Floating point representation is crucial because:

  - ✓ **Range:** It can represent a wide range of values from the very large to very small numbers.
  - ✓ **Precision:** It provides a good balance between the precision and range, making it suitable for the scientific computations, graphics and other applications where exact values and wide ranges are necessary.
  - ✓ **Flexibility:** It adapts to different scales of numbers allowing for the efficient storage and computation of real numbers in the computer systems.

Floating-Point Representation

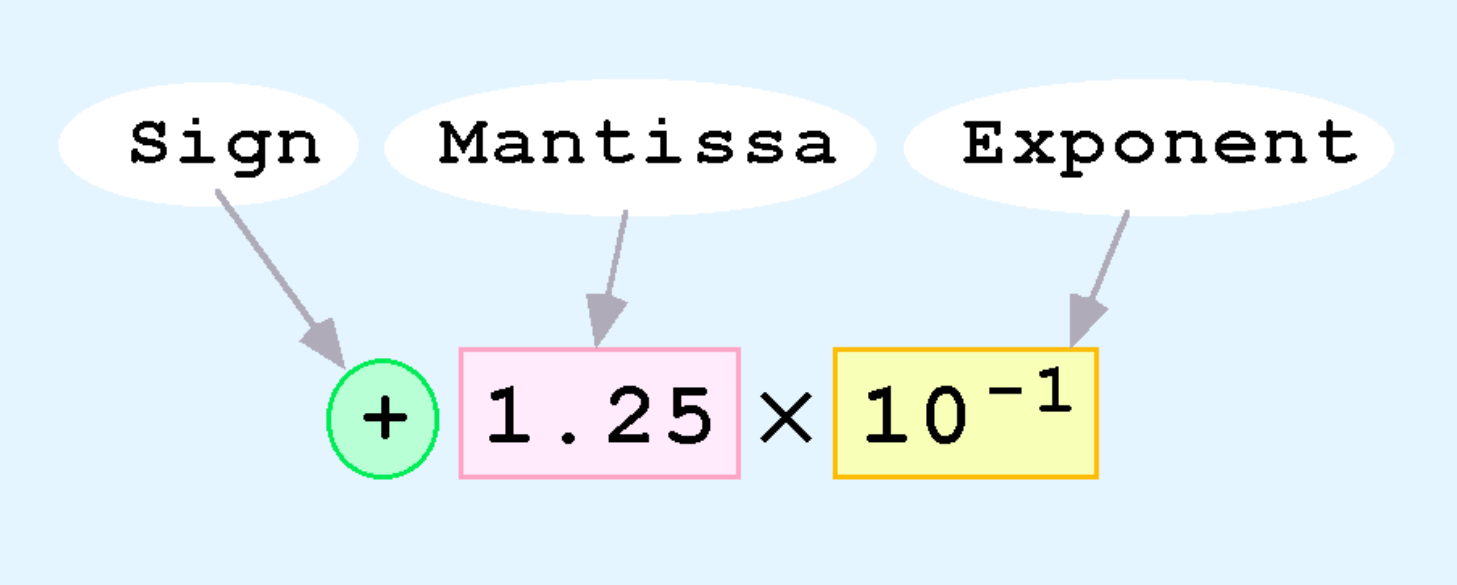
- Floating-point numbers allow an arbitrary number of decimal places to the right of the decimal point.
  - For example: $0.5 \times 0.25 = 0.125$
- They are often expressed in scientific notation.
  - For example:

    $0.125 = 1.25 \times 10^{-1}$

    $5{,}000{,}000 = 5.0 \times 10^{6}$

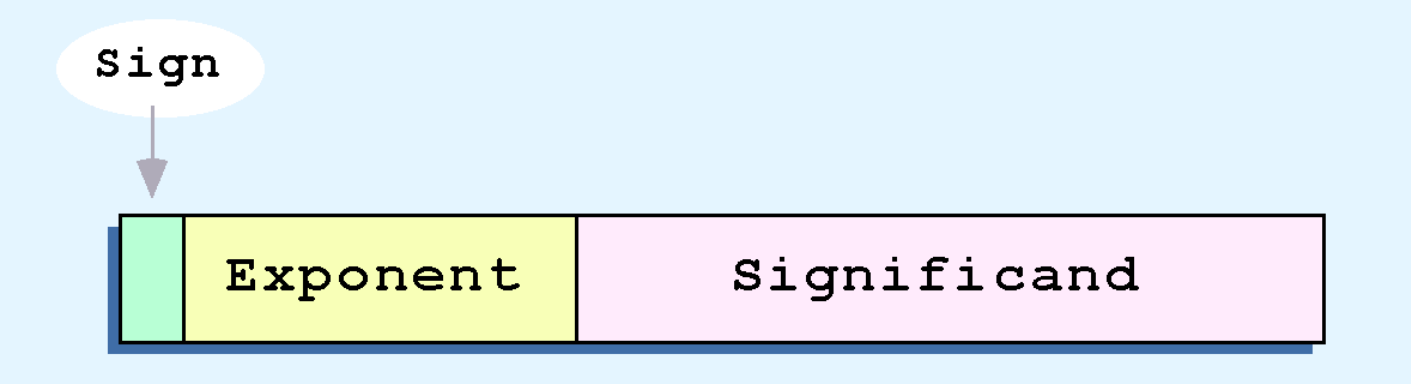
Floating-Point Representation

- Computers use a form of scientific notation for floating-point representation
- Numbers written in scientific notation have three components:



Floating-Point Representation

- Computer representation of a floating-point number consists of three fixed-size fields:

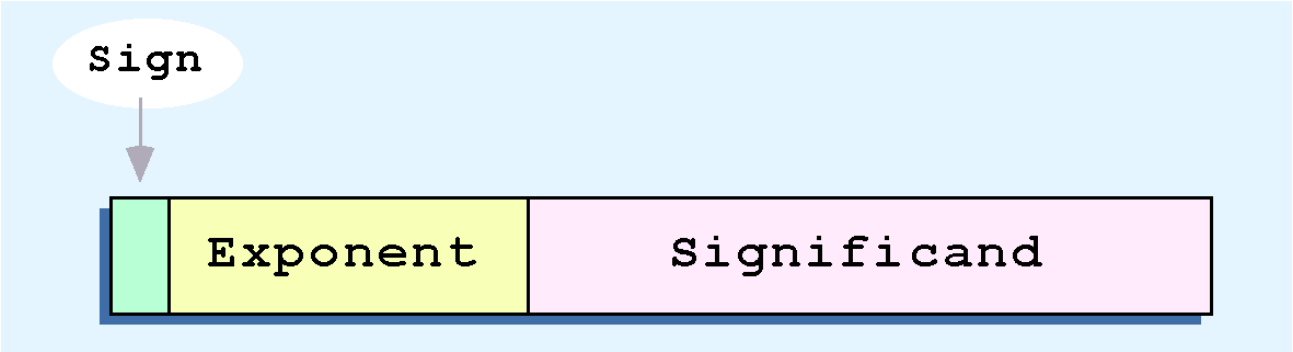


  - This is the standard arrangement of these fields

Note: Many people use "significand" and "mantissa" terms interchangeably. We use the term "significand" to refer to the fractional part of a floating point number.

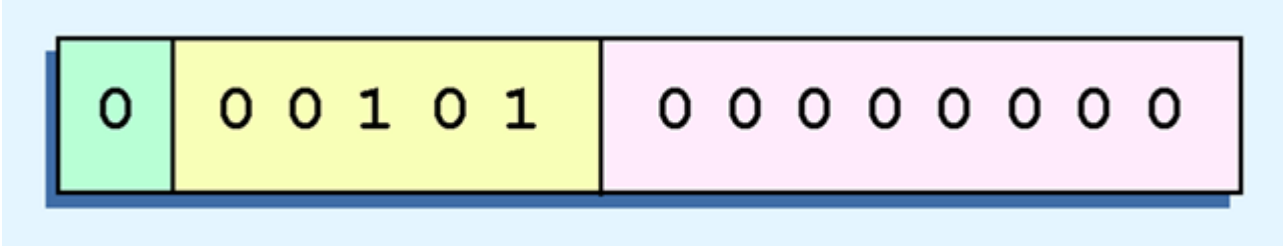
Floating-Point Representation

- a hypothetical "Simple Model" to explain the concepts
- In this model:
  - A floating-point number is 14 bits in length
  - The exponent field is 5 bits
  - The significand field is 8 bits



Floating-Point Representation

- **Example:**
  - Express 32_{10} in the simplified 14-bit floating-point model.
- We know that 32 is 2^5. So in (binary) scientific notation $32 = 1.0 \times 2^5$
- Using this information, we put 101 (= 5_{10}) in the exponent field and 0 in the significand as shown.



IEEE Standard 754 Floating Point Numbers

- The IEEE Standard for Floating-Point Arithmetic (IEEE 754) is a technical standard for floating-point computation which was established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE).
- The standard addressed many problems found in the diverse floating point implementations that made them difficult to use reliably and reduced their portability.
- IEEE Standard 754 floating point is the most common representation today for real numbers on computers, including Intel-based PC's, Macs, and most Unix platforms.

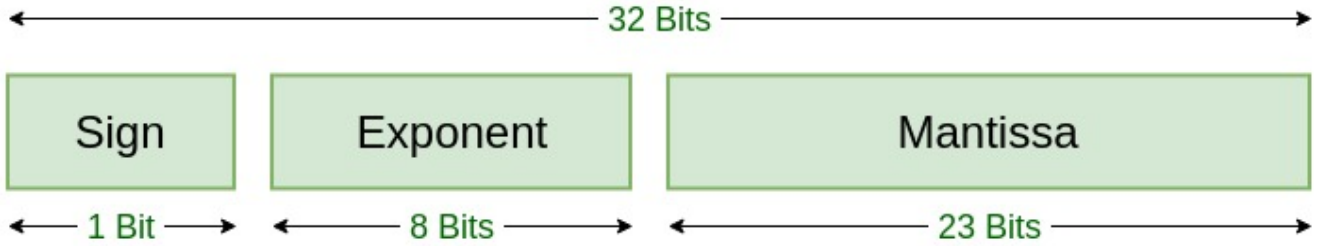
IEEE Standard 754 Floating Point Numbers

- **IEEE 754 has 3 basic components:**
- **The Sign of Mantissa –**
  - ✓ 0 represents a positive number while 1 represents a negative number.
- The Biased exponent –
  - ✓ The exponent field needs to represent both positive and negative exponents.
  - ✓ A bias is added to the actual exponent in order to get the stored exponent.
- The Normalised Mantissa –
  - ✓ The mantissa is part of a number in scientific notation or a floating-point number, consisting of its significant digits.
  - ✓ Here we have only 2 digits, i.e. O and 1. So a normalised mantissa is one with only one 1 to the left of the decimal..

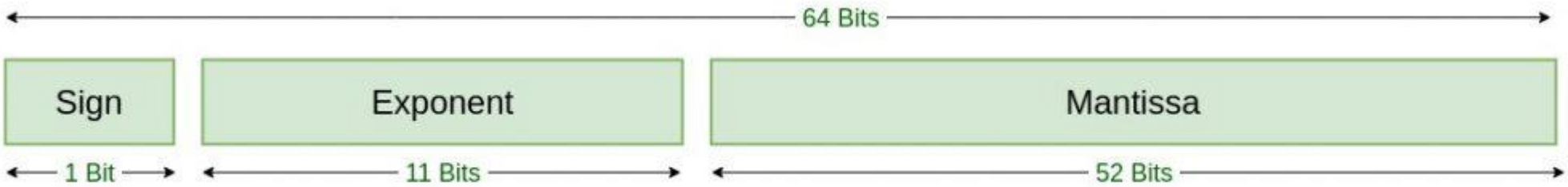
IEEE Standard 754 Floating Point Numbers

- **Types**

❑ 32-bit Single-Precision Floating-Point Numbers



❑ 64-bit Double-Precision Floating-Point Numbers



TYPES	SIGN	BIASED EXPONENT	NORMALISED MANTISA	BIAS
Single precision	1(31st bit)	8(30-23)	23(22-0)	127
Double precision	1(63rd bit)	11(62-52)	52(51-0)	1023

IEEE Standard 754 Floating Point Numbers

- Express the number $(85.125)_{10}$ in IEEE 754 single and double precision methods
  - 85 = 1010101
  - 0.125 = 001
  - 85.125 = 1010101.001
  - Normalized number =1.010101001 x 2^6
  - sign = 0

- **Single precision:**
  - biased exponent 127+6=133
  - 133 = 10000101
  - Normalised mantisa = 010101001
  - we will add 0's to complete the 23 bits

  - The IEEE 754 Single precision is: = **0 10000101 01010100100000000000000**
  - This can be written in hexadecimal form **42AA4000**

IEEE Standard 754 Floating Point Numbers

- **Express the number $(85.125)_{10}$ in IEEE 754 single and double precision methods**
  - 85 = 1010101
  - 0.125 = 001
  - 85.125 = 1010101.001
  - Normalized number =1.010101001×2^6
  - sign = 0

- **Double precision**
  - biased exponent 1023+6=1029
  - 1029 = 10000000101
  - Normalised mantisa = 010101001
  - we will add 0's to complete the 52 bits

  - The IEEE 754 Double precision is:
  - = **0 10000000101 0101010010000000000000000000000000000000000000000000**
  - This can be written in hexadecimal form 405548000000000

IEEE Standard 754 Floating Point Numbers

- **Example 1: Suppose that IEEE-754 32-bit floating-point representation pattern is**

0 10000000 110 0000 0000 0000 0000 0000.

Determine the number

```
Sign bit S = 0 ⇒ positive number
E = 1000 0000B = 128D (in normalized form)
Fraction is 1.11B (with an implicit leading 1) = 1 + 1×2^-1 + 1×2^-2 = 1.75D
The number is +1.75 × 2^(128-127) = +3.5D
```

- **Example 2: Suppose that IEEE-754 32-bit floating-point representation pattern is**

1 01111110 100 0000 0000 0000 0000 0000.

```
Sign bit S = 1 ⇒ negative number
E = 0111 1110B = 126D (in normalized form)
Fraction is 1.1B  (with an implicit leading 1) = 1 + 2^-1 = 1.5D
The number is -1.5 × 2^(126-127) = -0.75D
```

Numbering Systems

COMPUTER ORGANIZATION AND ARCHITECTURE

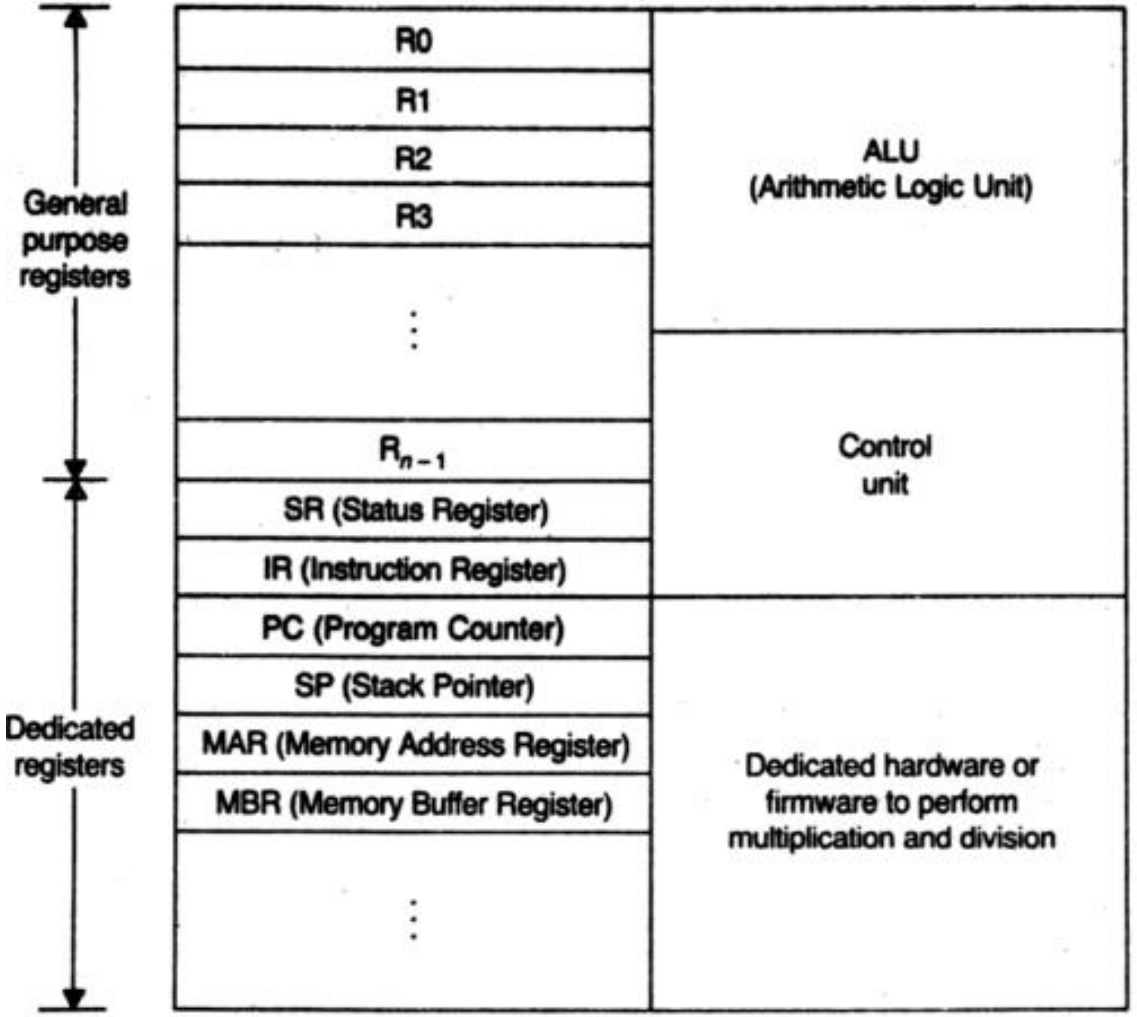
Execution Unit

Dr. Bore Gowda S B
Additional Professor
Dept. of ECE
MIT, Manipal

Introduction

A typical CPU model

- A typical CPU model is shown in figure
- A conventional CPU consists of the following:
  - ✓ General purpose registers
  - ✓ Dedicated registers
  - ✓ An ALU
  - ✓ Dedicated hardware or firmware elements that perform a special operations such as multiplication or division
  - ✓ A control unit



Register Section

- CPU with many register reduces the number of memory access
- ***General purpose register:*** may be configured as an accumulator, address pointer or a data pointer
- ***Dedicated register:*** *used for some specific tasks*

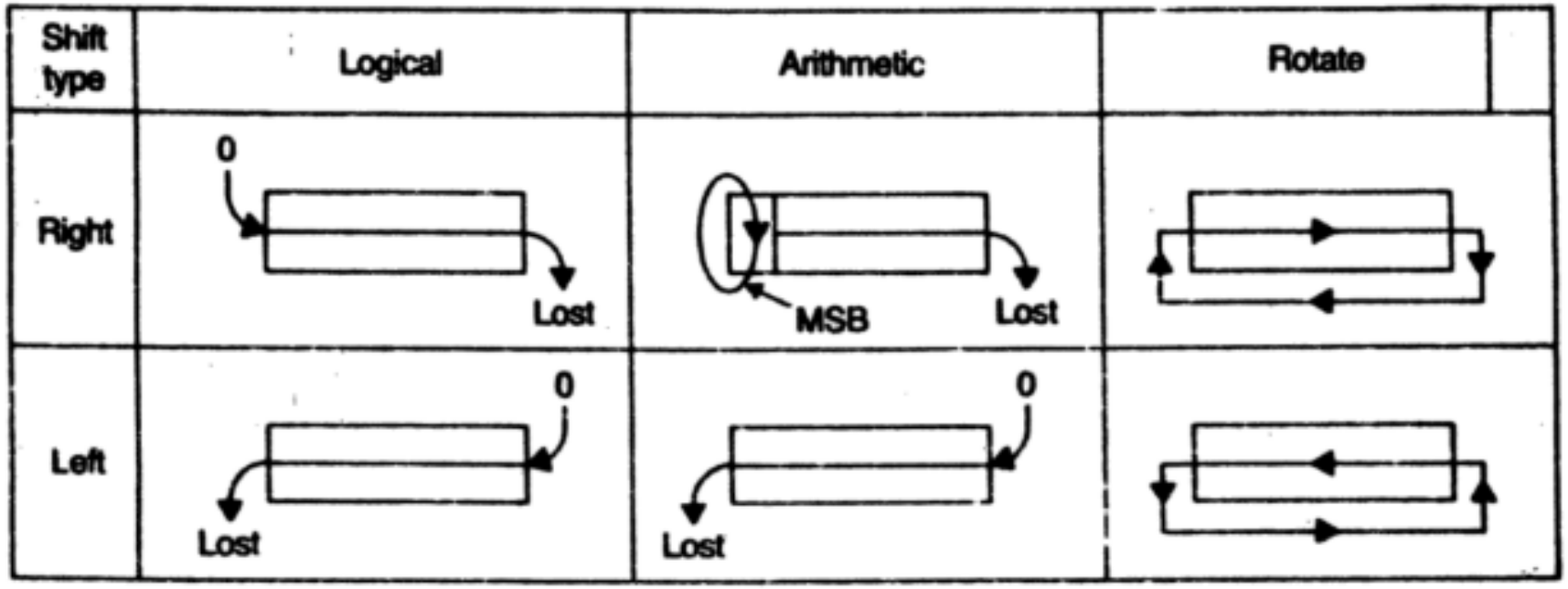
- Commonly known special purpose registers and their tasks rre:

REGISTER	TASK
PC	Usually holds the address of the next instruction to be executed.
SP	Usually holds the address of the top element of the stack.
IR	Holds the instruction code currently being executed.
MAR	Holds the address of the data item to be retrieved from the main memory.
MBR	Holds the data item retrieved from the main memory.
SR or PSW	Holds the condition code flags and other information that describe the status of the running program.

General Register Design

Typical shift operations:

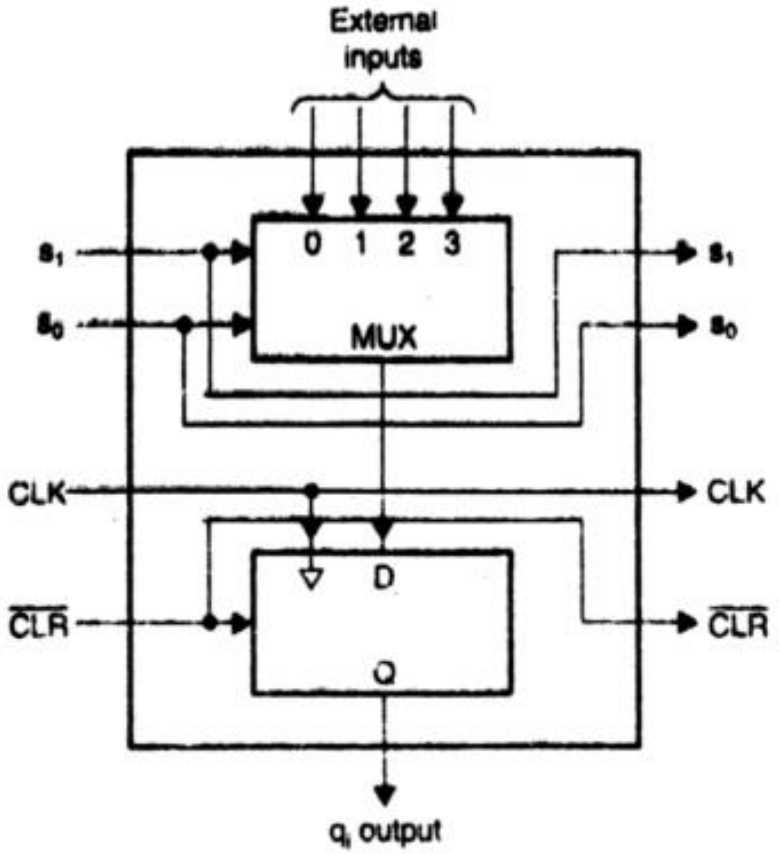
- GPR is to store address or data, then to be able to retrieve the data when needed
- GPR is also capable of manipulating the stored data by shift left or shift right operation
- Logical shift operations
- Arithmetic shift operations



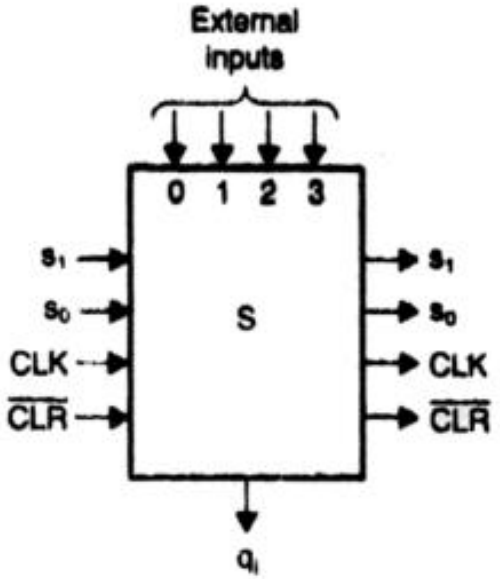
General Register Design

Basic cell for designing GPR

- The hypothetical instruction set of a computer consists of eight instructions, I_0, I_1, I_2,,,,,,I_7. The relative frequency of these instructions are as follows:



a. Internal Organization of the Basic Cell S

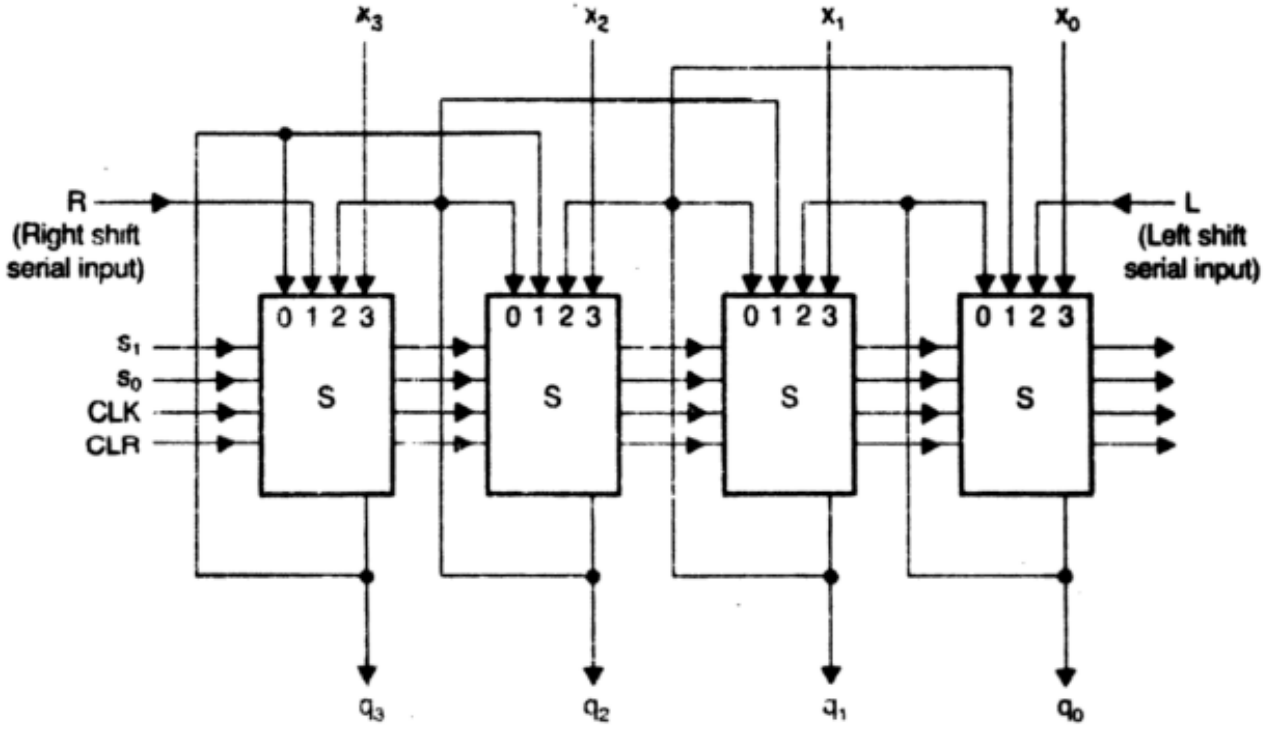


b. Block Diagram of the Basic Cell S

General Register Design

4-bit GPR:

- Parallel load, shift left, shift right, serial loading of data



Selection Inputs		Clock Input	Clear Input	Operation
s_1	s_0	CLK	$\overline{CLR}$	
X	X	X	0	Clear
0	0	↑	1	No operation
0	1	↑	1	Shift right
1	0	↑	1	Shift left
1	1	↑	1	Parallel load

Note: X – don't care; ↑ : rising edge of the clock

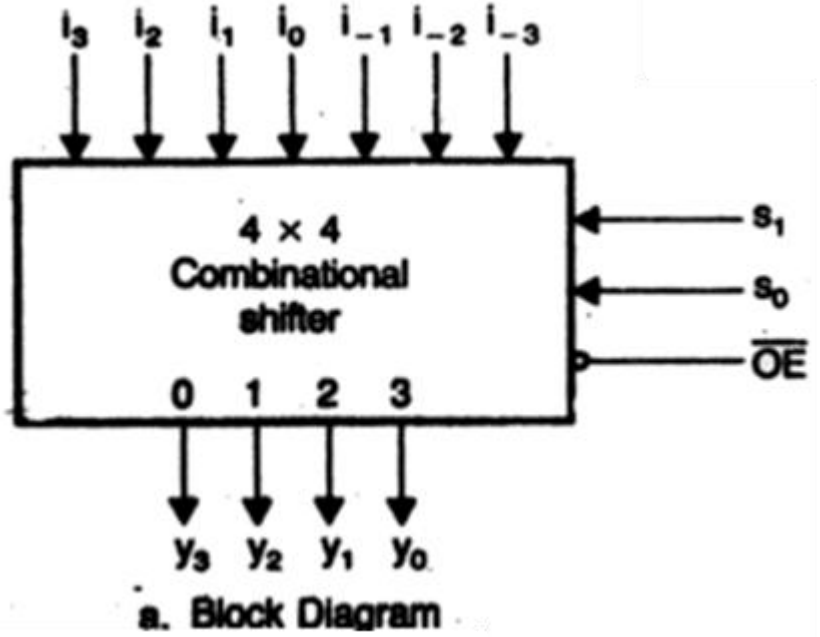
Three Variations of the Right Shift Operations

s_1s_0 = 01	
Value of R	Shift realized
0	Logical right shift
q_3	Arithmetic right shift
q_0	Rotate right

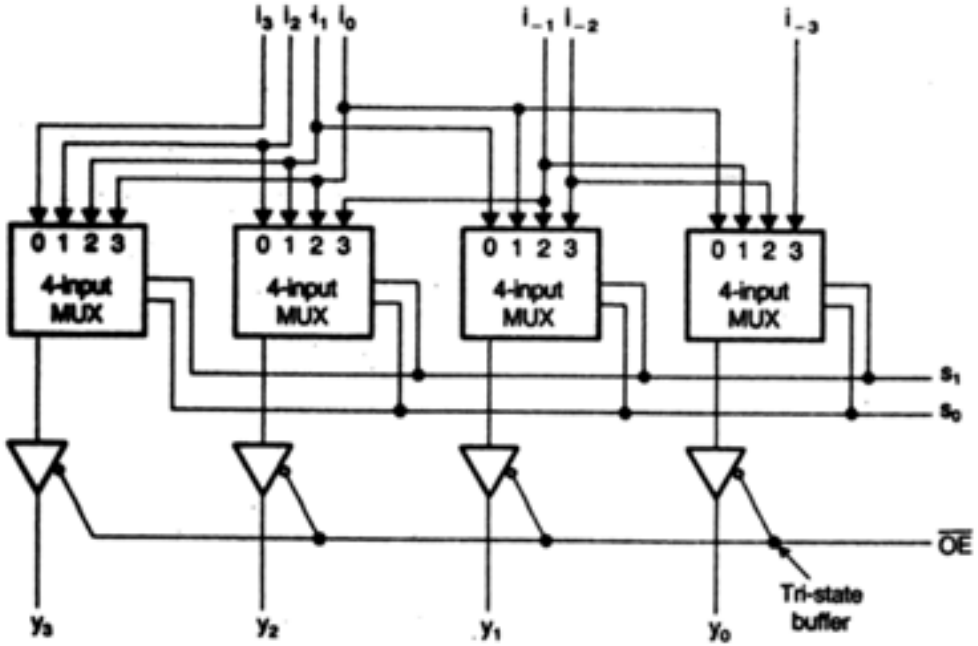
General Register Design

4x4 Combinational shifter design:

- The speed of a flip flop-based shifter is a function of the clock frequency
- A high-speed shifter can be designed using combinational circuit components such as a multiplexer



a. Block Diagram



b. Internal Schematic

General Register Design

4x4 Combinational shifter design:

	Shift Count		Output				
$\overline{OE}$	s_1	s_0	y_3	y_2	y_1	y_0	Comment
1	X	X	Z	Z	Z	Z	Output lines float
0	0	0	i_3	i_2	i_1	i_0	Pass (no shift)
0	0	1	i_2	i_1	i_0	i_{-1}	Left shift once
0	1	0	i_1	i_0	i_{-1}	i_{-2}	Left shift twice
0	1	1	i_0	i_{-1}	i_{-2}	i_{-3}	Left shift three times

Note: Z—High-impedance state.
X—Don't care.

c. Truth Table

$$Y_3 = s_1's_0'i_3 + s_1's_0i_2 + s_1s_0'i_1 + s_1s_0i_0$$

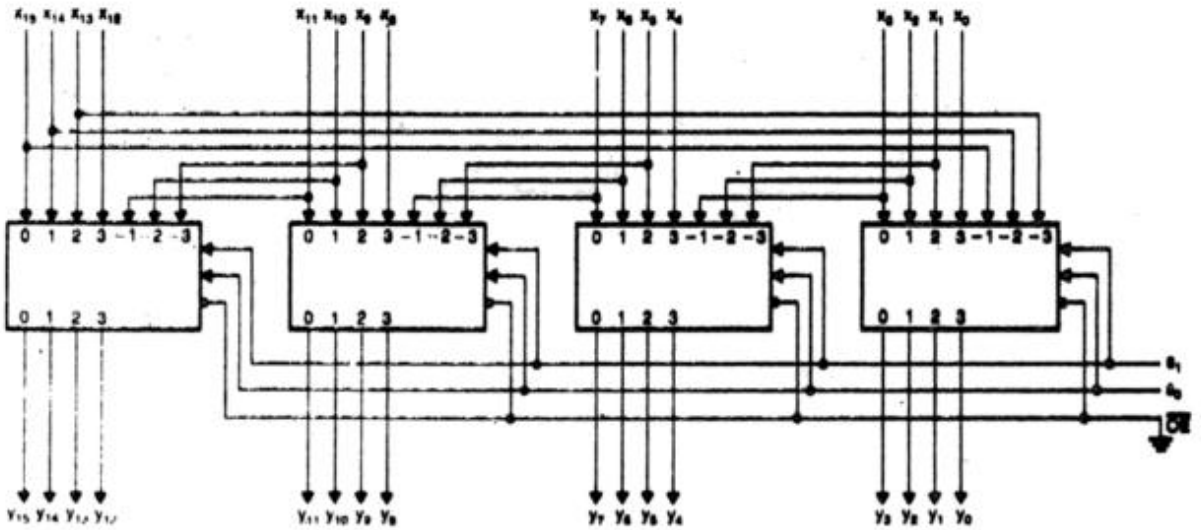
$$Y_2 = s_1's_0'i_2 + s_1's_0i_1 + s_1s_0'i_0 + s_1s_0i_{-1}$$

$$Y_1 = s_1's_0'i_1 + s_1's_0i_0 + s_1s_0'i_{-1} + s_1s_0i_{-2}$$

$$Y_0 = s_1's_0'i_0 + s_1's_0i_{-1} + s_1s_0'i_{-2} + s_1s_0i_{-3}$$

General Register Design

- **Combinational shifter capable of rotating 16-bit data to the left by 0, 1, 2 and 3 positions:**



a. Logic Diagram

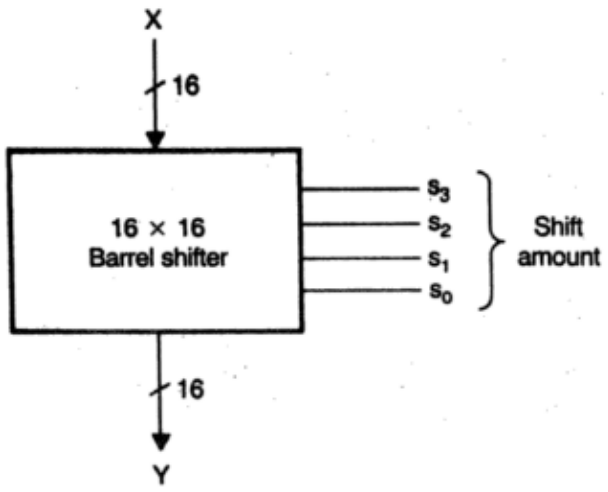
Shift Count	Output															
s_1 s_0	y_{15}	y_{14}	y_{13}	y_{12}	y_{11}	y_{10}	y_9	y_8	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0
0 0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
0 1	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}
1 0	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}
1 1	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}

b. Truth Table

General Register Design

16x16 Barrel shifter:

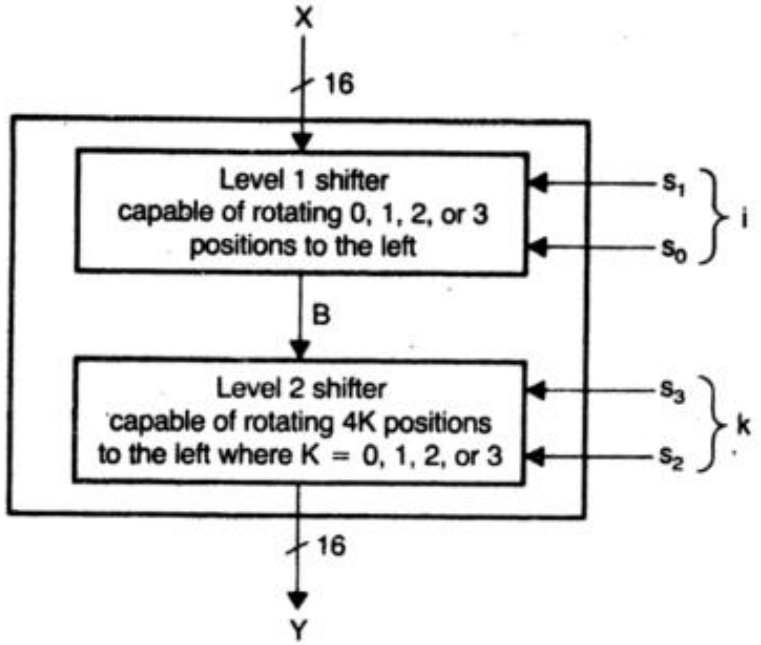
- Capable of rotating the given 16-bit data to the left n position where $0 \leq n \leq 15$



a. Block Diagram of a 16 × 16 Block Shifter

Shift Amount				Output													
s_3	s_2	s_1	s_0	y_{15}	y_{14}	y_{13}	y_{12}	y_{11}	y_{10}	y_9	y_8	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0
0	0	0	0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
0	0	0	1	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}
0	0	1	0	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}
0	0	1	1	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}
0	1	0	0	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}
0	1	0	1	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}
0	1	1	0	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}
0	1	1	1	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9
1	0	0	0	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8
1	0	0	1	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7
1	0	1	0	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6
1	0	1	1	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5
1	1	0	0	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4
1	1	0	1	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3
1	1	1	0	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2
1	1	1	1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1

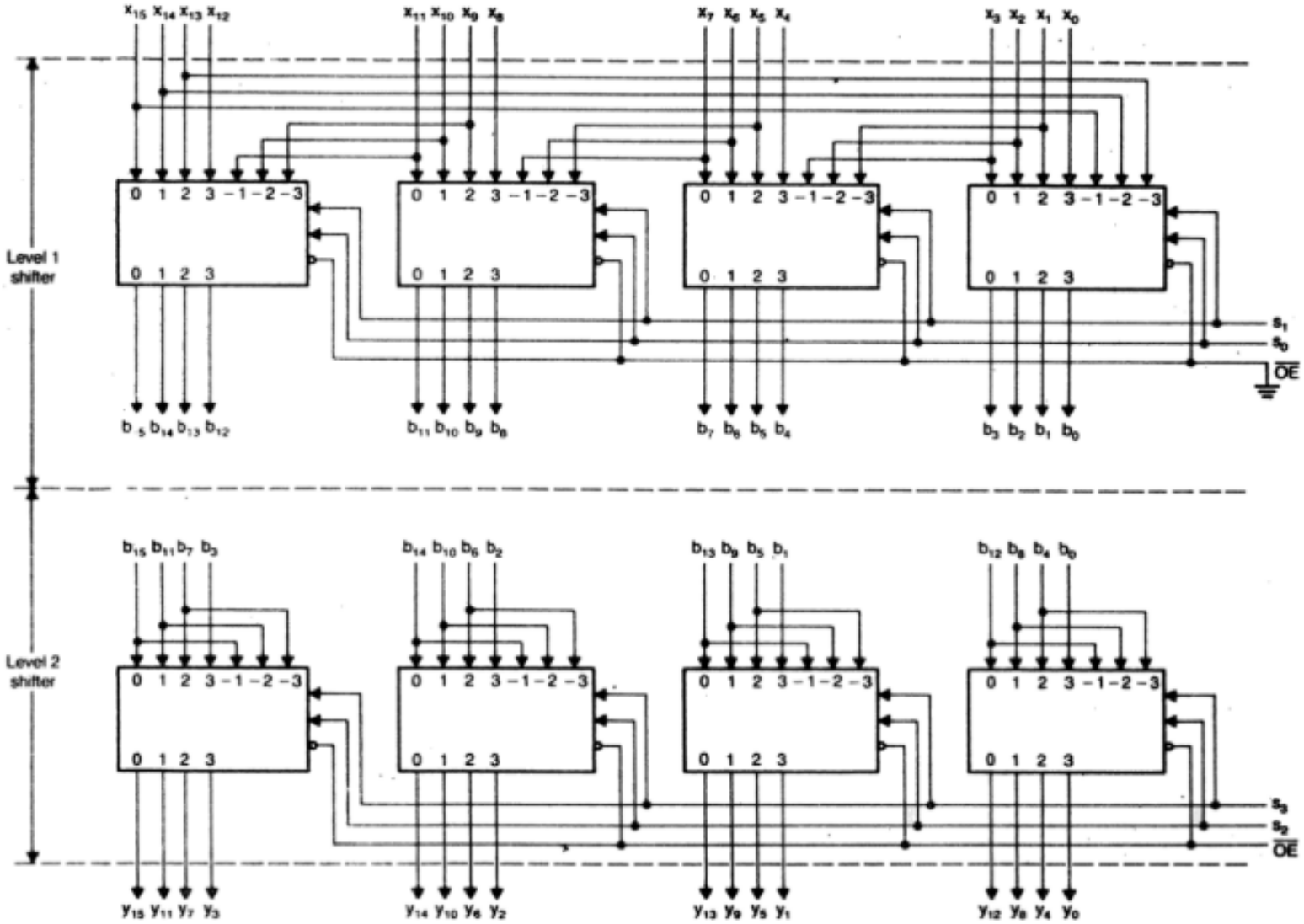
b. Truth Table of the 16 × 16 Barrel Shifter



c. Functional Block Diagram of a 16 × 16 Barrel Shifter

General Register Design

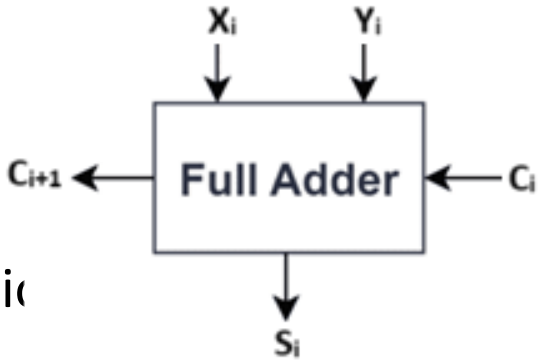
- **Complete Logic diagram of full 16-bit Barrel shifter:**



Adder Design

Full adder:

- Addition is the basic operation performed by an ALU
- Operation is versatile
- A- B = A + 2's complement of B
- B*C may be obtained by adding A to itself for C - 1 times
- The speed of the hardware unit is essential to the efficient operati of execution unit



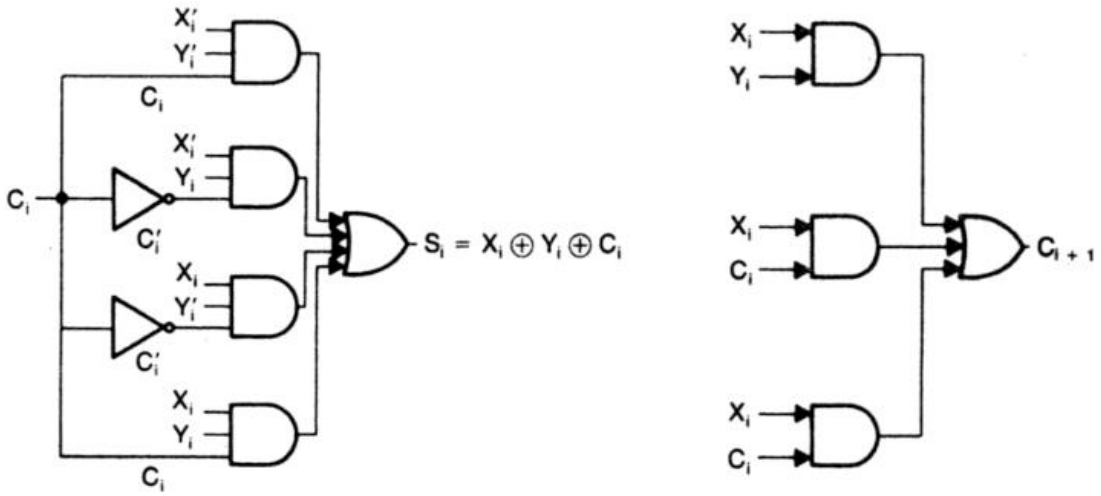
X_i	Y_i	C_i	S_i	C_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$s_i = \overline{x_i}.\overline{y_i}.c_i + \overline{x_i}.y_i.\overline{c_i} + x_i.\overline{y_i}.\overline{c_i} + x_i.y_i.c_i$$

$$c_{i+1} = x_i.y_i + x_i.c_i + y_i.c_i$$

Adder Design

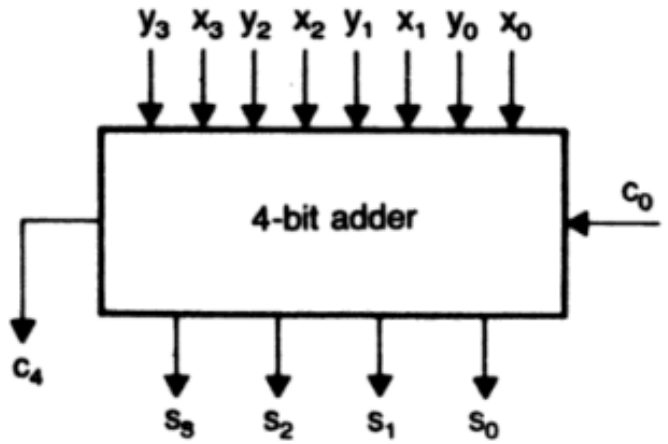
Full adder:



- To generate ***C_{i+1} from C_i, 2 gate delays*** are required
- To generate ***sum S_i from C_i, 3 gate delays*** are required
- Assume that the gate delay is Δ time units and the actual value of it is decided by the technology used
- For example, TTL logic circuits have a Δ will be 10 ns

Adder Design

4-bit ripple carry adder/Carry Propagate Adder(CPA)

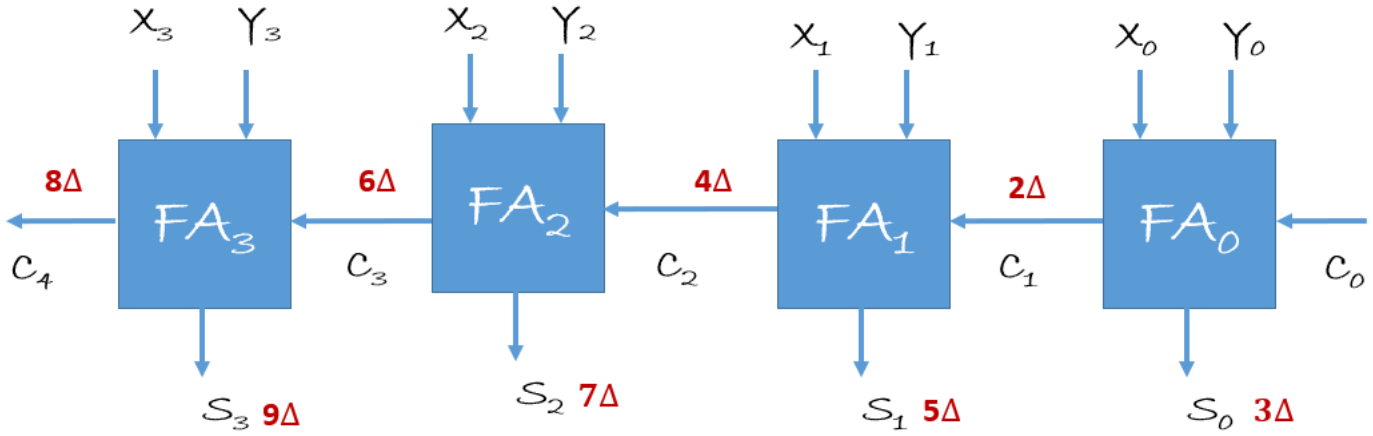


Total delay = (n-1)* delay between each block + last block delay

Wnere n is number of Full Adders

For n = 4 and if gate delay is 0.5ns, we get

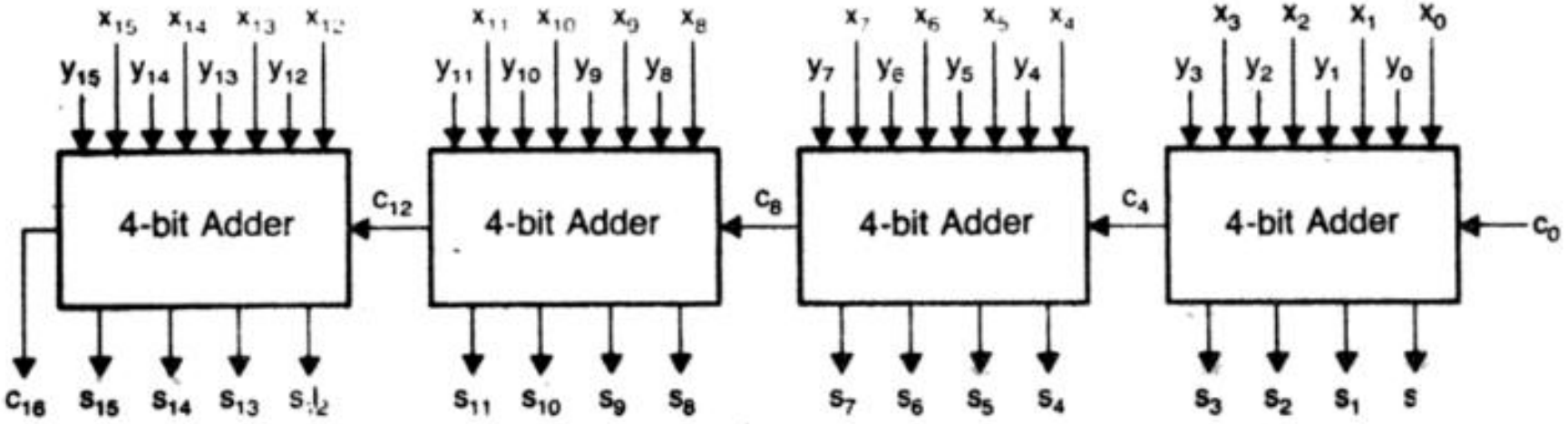
Delay = (4 – 1) * 1 + 1.5 = 4.5 ns



Adder Design

16-bit adder using 4-bit ripple carry adder as basic block

- Design of an n-bit CPA is straightforward the carry propagation time limits the speed of operation
- For example: 16-bit CPA when the addition operation is completed only when the sum bits S_0 through are S_{15} available
- In the worst case carry propagate through 15 full adders
- **Total Time delay**= Time taken for carry to propagate through 15 full adder + Time taken to generate S_{15} from C_{14} = $15 * 2\Delta + 3\Delta = 33\Delta$



- *Circuit is simple and easy to build*
- *Ripple of carry is causing delay and it increases as **n** increases.*

Fast Adder Design

Carry Look-Ahead Adder (CLA)

- It is a type of adder used in digital logic that improves the speed of arithmetic operations by reducing the time it takes to calculate carry bits
- Key Concepts:
  1. **Carry Propagation:** In a traditional ripple carry adder, each bit must wait for the carry bit from the previous bit to be calculated, which can be slow.
  2. **Carry Look-Ahead**: The CLA adder speeds up this process by calculating the carry bits in advance, using the concepts of generate and propagate.

- How It Works:
  1. **Carry Generate (G):** A bit generates a carry if both of its input bits are 1.
  2. **Carry Propagate (P):** A bit propagates a carry if at least one of its input bits is 1.

Adder Design

Carry Look-Ahead Adder (CLA)

- For Full adder, we know that:

$$c_{i+1} = x_i.y_i + x_i.c_i + y_i.c_i$$

- **Carry generate function**

$$G_i = x_i.y_i$$

- **Carry propagate function**

$$P_i = x_i + y_i$$

- **Carry function**

$$c_{i+1} = G_i + P_i.C_i$$

- These equations show that a carry signal will be generated in two cases:

1. if both bits x_i and y_i are 1
2. if either x_i or y_i is 1 and the carry-in C_i is 1.

Adder Design

4-bit Carry Look-Ahead Adder (CLA)

- Apply these equations for a 4-bit adder:

$$C_1 = G_0 + P_0C_0$$
$$C_2 = G_1 + P_1C_1 = G_1 + P_1(G_0 + P_0C_0) = G_1 + P_1G_0 + P_1P_0C_0$$
$$C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$$
$$C_4 = G_3 + P_3C_3 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$$

- These expressions show that C_2, C_3 and C_4 do not depend on its previous carry-in.
- Therefore C_4 does not need to wait for C_3 to propagate.
- As soon as C_0 is computed, C_4 can reach steady state.
- The same is also true for C_2 and C_3

Adder Design

4-bit Carry Look-Ahead Adder (CLA)

- Apply these equations for a 4-bit adder:

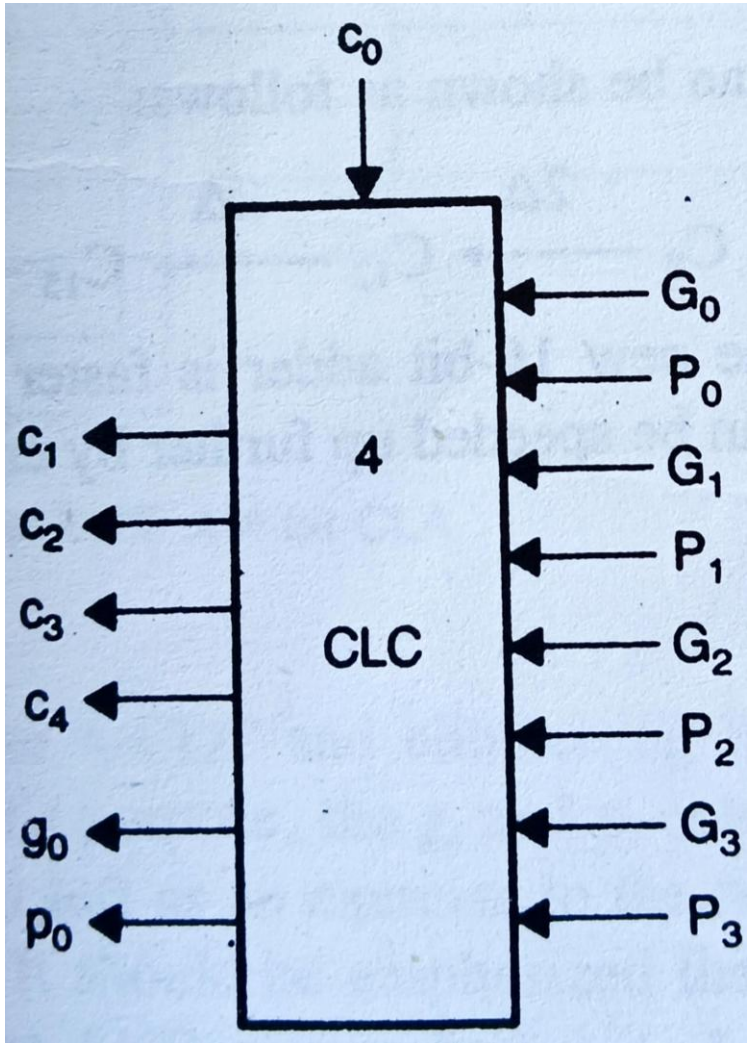
$$C_1 = G_0 + P_0C_0$$
$$C_2 = G_1 + P_1C_1 = G_1 + P_1(G_0 + P_0C_0) = G_1 + P_1G_0 + P_1P_0C_0$$
$$C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$$
$$C_4 = G_3 + P_3C_3 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$$

- These expressions show that C_2, C_3 and C_4 do not depend on its previous carry-in.
- Therefore C_4 does not need to wait for C_3 to propagate.
- As soon as C_0 is computed, C_4 can reach steady state.
- The same is also true for C_2 and C_3
- For these reason, the equations are called *carry look ahead equations*
- The hardware that implements these equations is called *4-stage carry-look ahead circuit (4 CLC)*

General Register Design

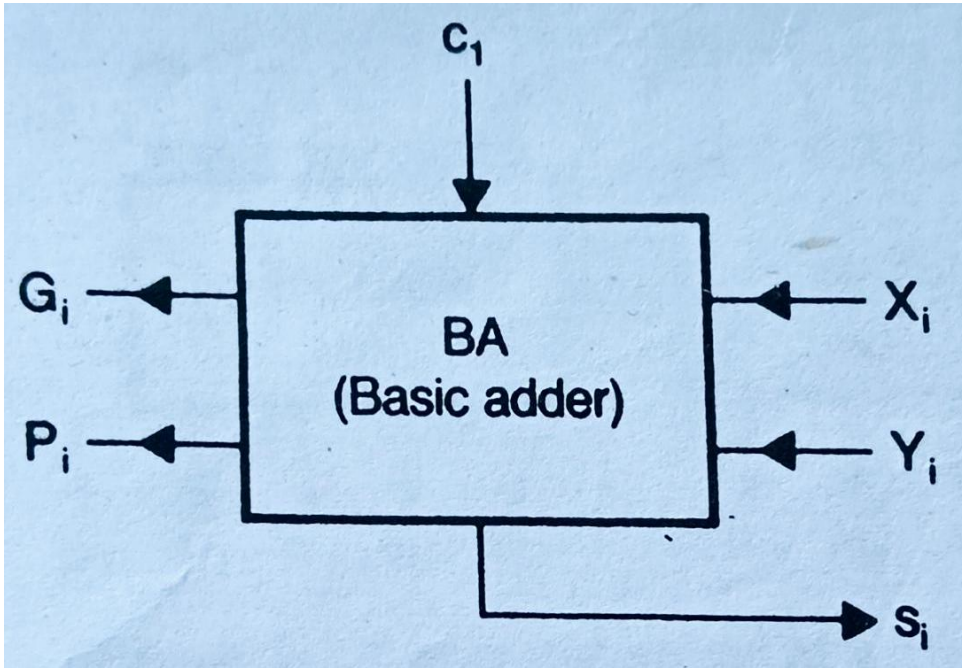
4-stage Carry Look-Ahead circuit(4-CLC)

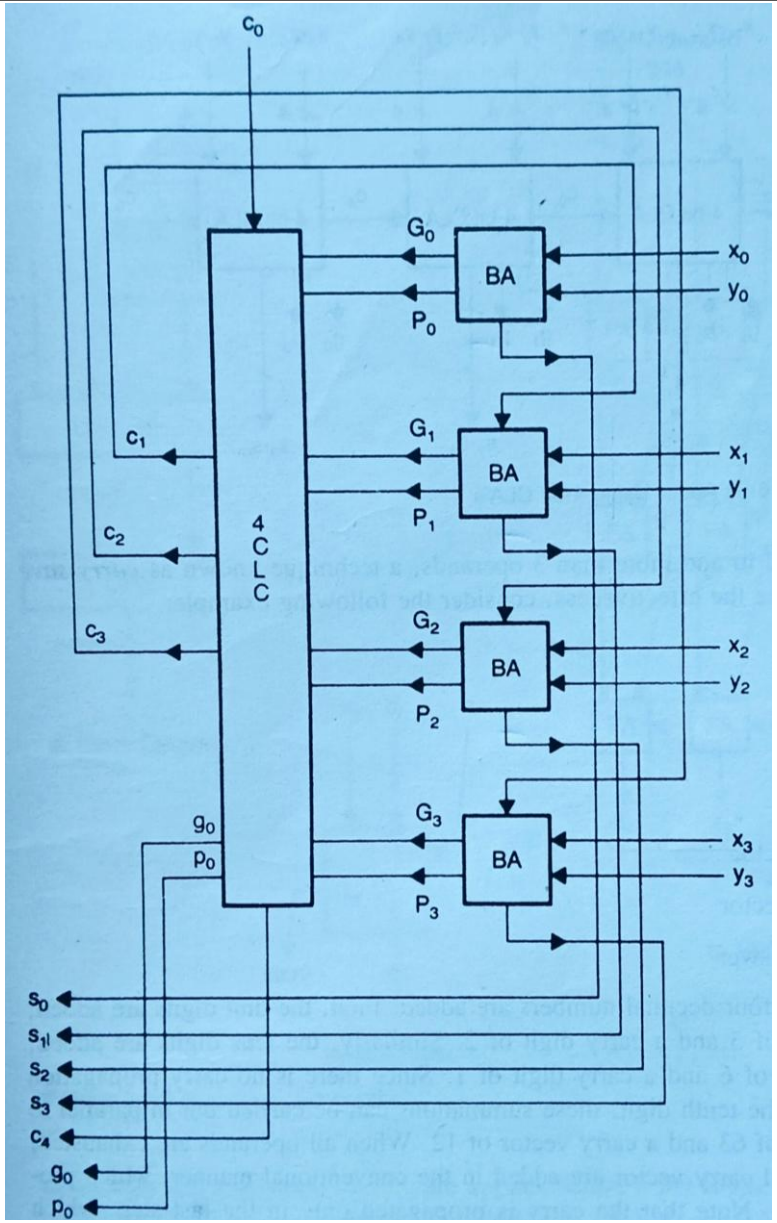
- A 4-CLC can be implemented as a teo-level AND/OR logic circuit
- To generate C4, 5-input **AND** gate and 4-input **OR** gate required
- If $G_i \leq (0 \leq i \leq 3)$ is available, then all C_i's with $(1 \leq i \leq 4)$ can be generated in 2 gate delays
- The output G_0 and P_0 are useful to obtain a higher order look ahead system



General Register Design

- 4-bit CLA using basic adder

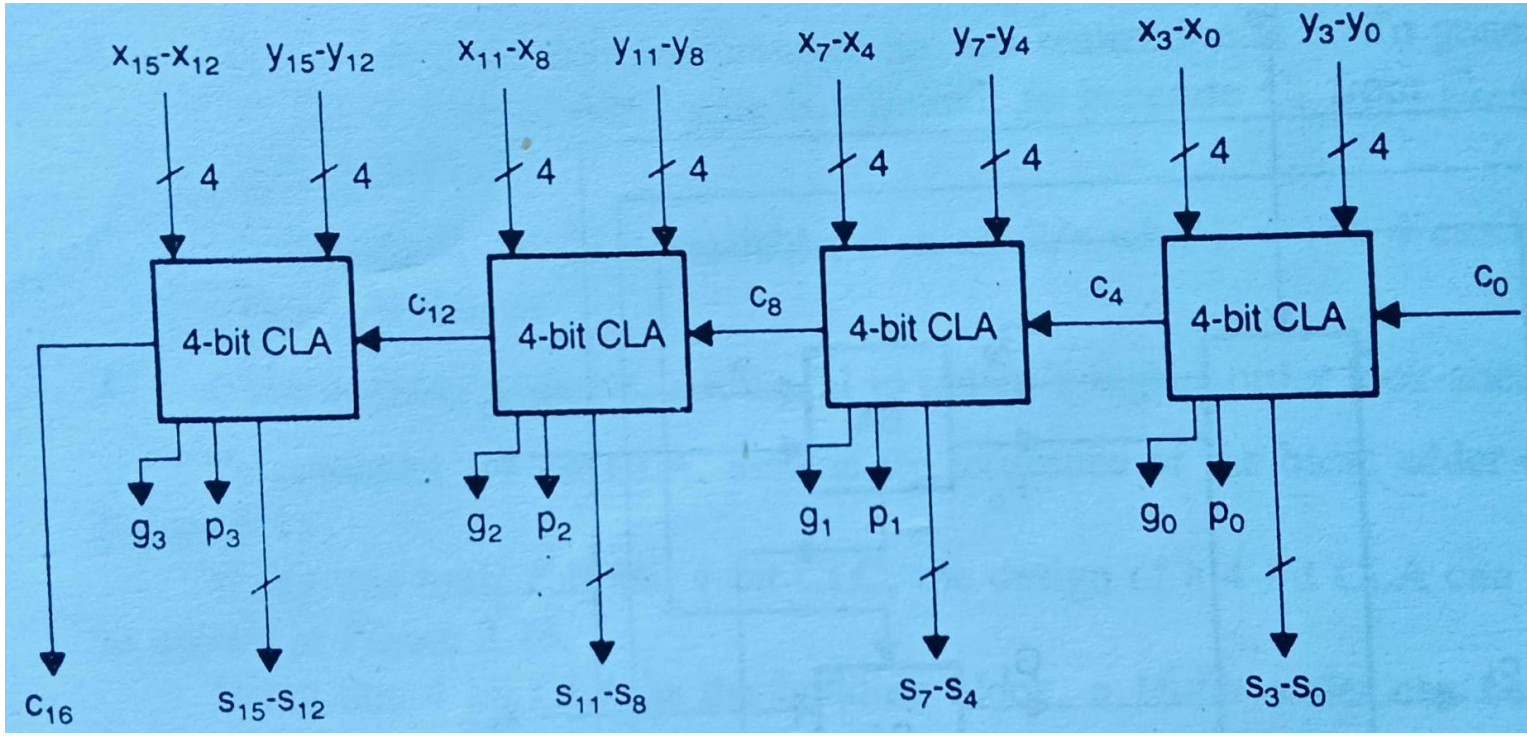




General Register Design

16-bit CLA using 4-bit CLA

- A 16-bit CLA is typically divided into four 4-bit CLAs.
- Each 4-bit CLA generates its own carry-out signals, which are then used to compute the carry-out for the next higher level.



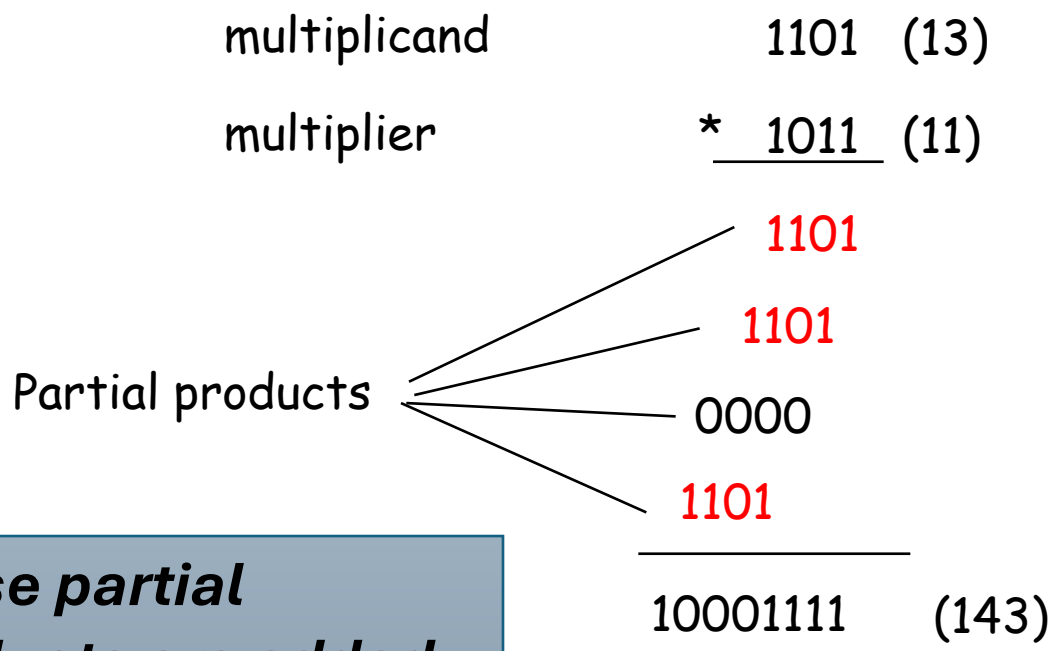
	Delay
For G_i P_i generation from X_iY_i $(0 \leq i \leq 15)$	Δ
To generate C_4 from C_0	2Δ
To generate C_8 from C_4	2Δ
To generate C_{12} from C_8	2Δ
To generate C_{15} from C_{12}	2Δ
To generate S_{15} from C_{15}	3Δ
Total delay	12Δ

A graphical illustration of this calculation can be shown as follows:

Data available $\xrightarrow{\Delta}$ G_iP_i $\xrightarrow{2\Delta}$ C_4 $\xrightarrow{2\Delta}$ C_8 $\xrightarrow{2\Delta}$ C_{12} $\xrightarrow{2\Delta}$ C_{15} $\xrightarrow{3\Delta}$ S_{15}

Multiplication of Binary numbers

Basic Concept



These partial products are added to get the final product.

product of two 4-bit numbers is an 8-bit number

Binary multiplication is easy

0 × multiplicand = 0

1 × multiplicand = multiplicand

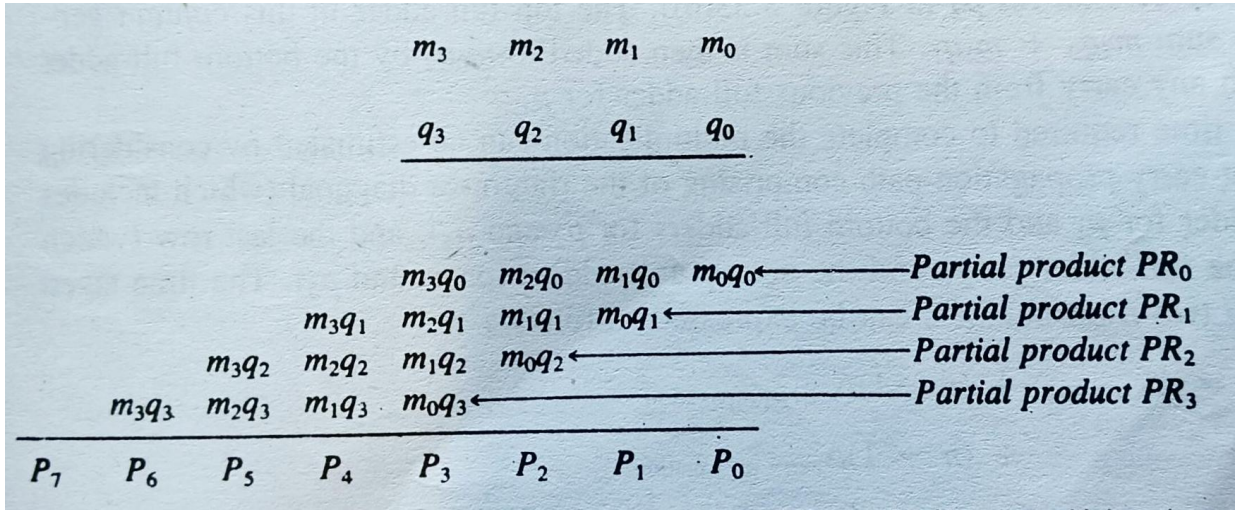
It's interesting to note that binary multiplication is a sequence of shifts and adds of one number (depending on the bits in the second number).

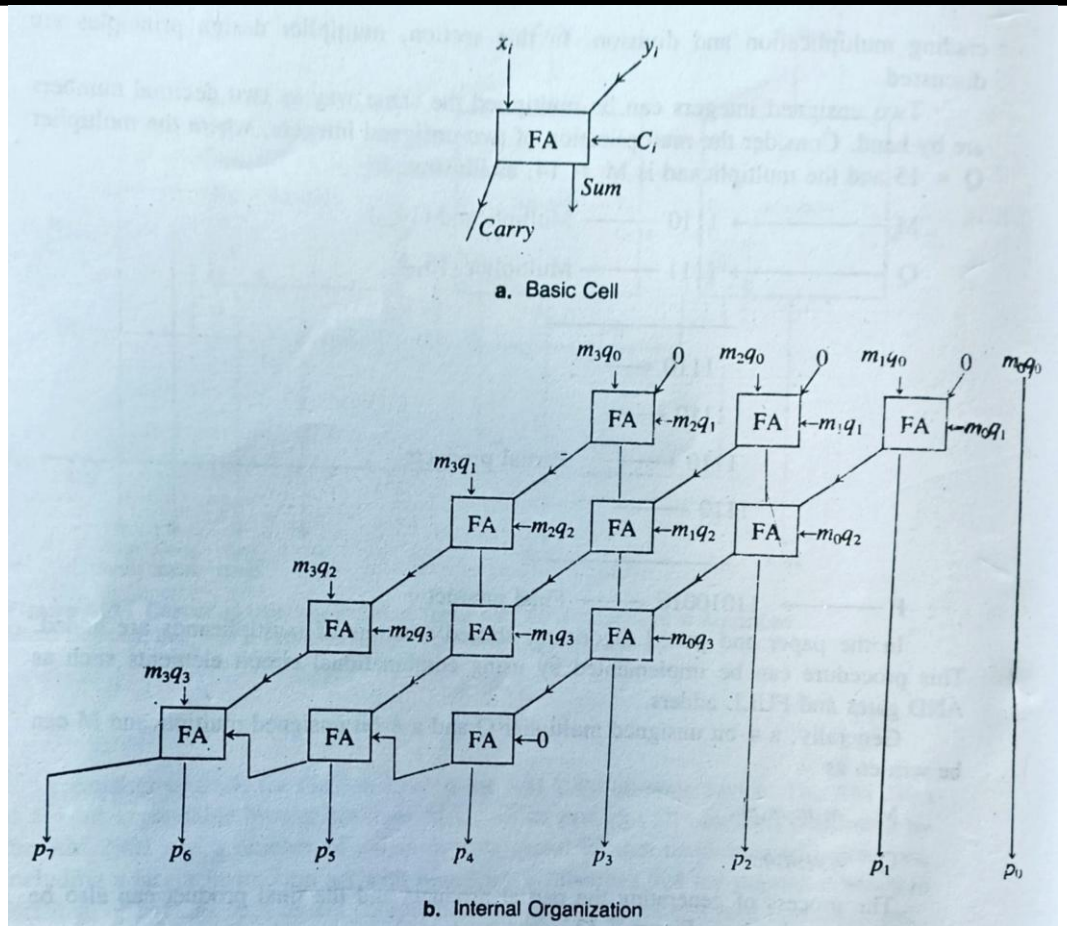
Multiplication of Binary numbers

- **4x4 array multiplication**

M: $m_3 m_2 m_1 m_0$

Q: $q_3 q_2 q_1 q_0$





- Each cross-product term in this can be generated using an AND gate
- This requires 16 AND gate to generate all cross-product terms that are summed by full adder arrays
- It is nonadditive multiplier(NM), since it does not include any additive inputs

$$T(n) = \Delta_{\text{AND gate}} + (n-1)\Delta_{\text{carry propagation}} + (n-1)\Delta_{\text{carry propagation}}$$

Multiplication of Binary numbers

4x4 array multiplication

- An additive multiplier(AM) includes an extra input, it computes products of the form

$$P = M*Q + R$$

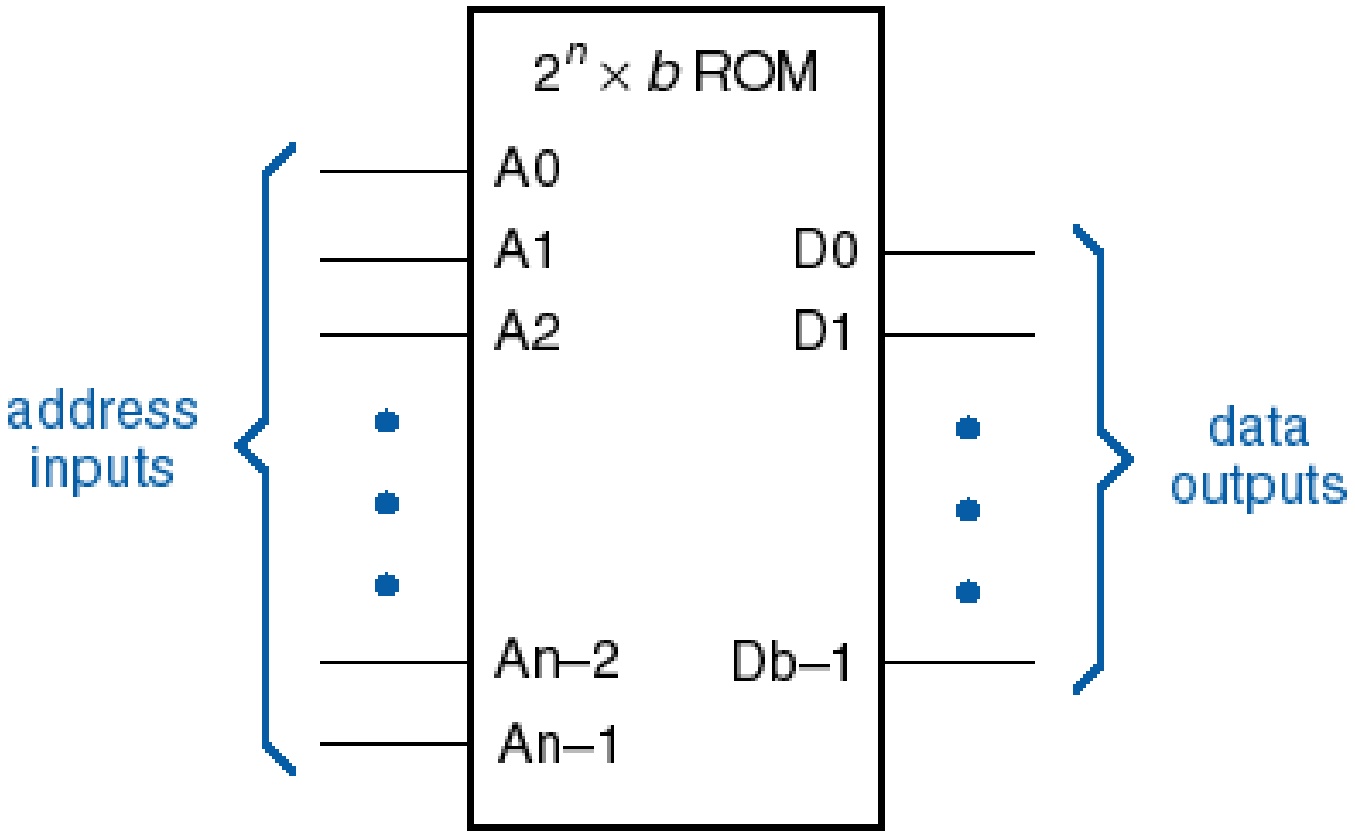
- Both NM and an AM are available as standard IC blocks

- Simple and repetitive structure
- Can multiply only unsigned numbers
- Need additional logic to multiply signed numbers
- Too big circuit and utilization factor is too less
- Delay grows as the number of bits to be multiplied increases

Multiplication of Binary numbers

- **Read Only Memory**

ROM is a combinational circuit; it's a truth-table

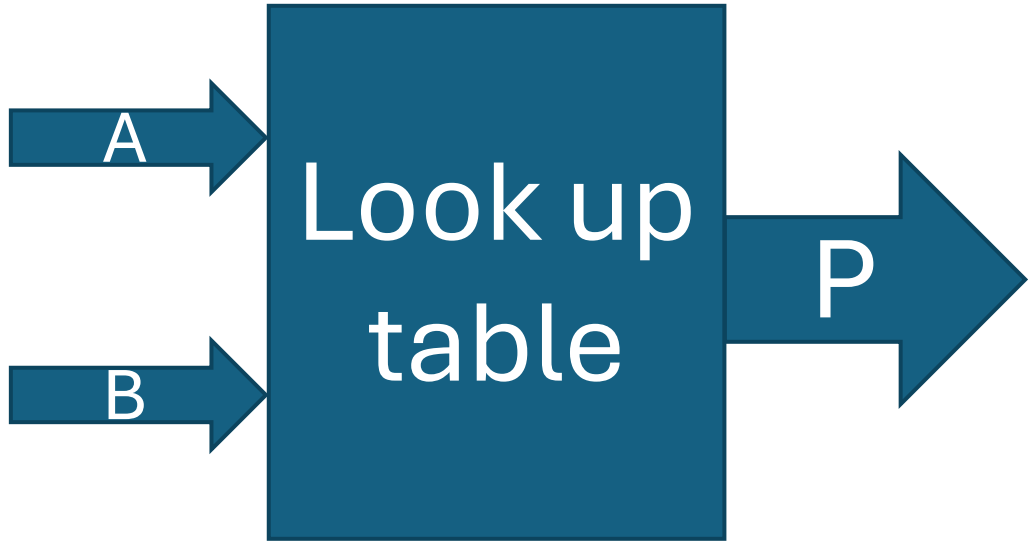


- Can perform any combinational logic function
  - Address inputs = function inputs
  - Data outputs = function outputs

you give input numbers to be multiplied and it will give you the product!!

Multiplication of Binary numbers

- **Read Only Memory**



Address input

Data output

A	B	Product
00	00	0000
00	01	0000
00	10	0000
00	11	0000
01	00	0000
01	01	0001
01	10	0010
01	11	0011
11	00	0000
11	01	0011
11	10	0110
11	11	1001

Consider A and B to be 2-bit numbers, How many bits will be in product, P?
What is the memory size of this ROM?

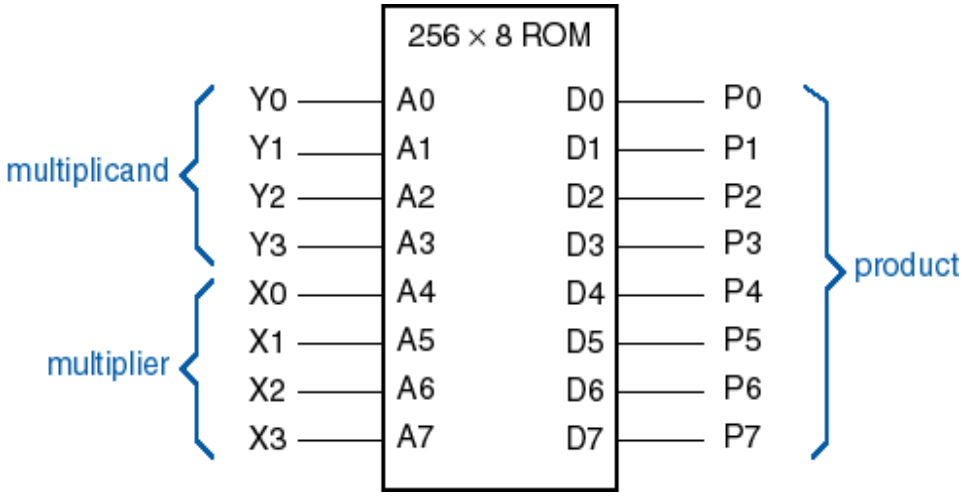
8 bytes!!

16 locations

Multiplication of Binary numbers

4x4 ROM based multiplier example

How many address and data lines?



	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
10:	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
20:	00	02	04	06	08	0A	0C	0E	10	12	14	16	18	1A	1C	1E
30:	00	03	06	09	0C	0F	12	15	18	1B	1E	21	24	27	2A	2D
40:	00	04	08	0C	10	14	18	1C	20	24	28	2C	30	34	38	3C
50:	00	05	0A	0F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
60:	00	06	0C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A
70:	00	07	0E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
80:	00	08	10	18	20	28	30	38	40	48	50	58	60	68	70	78
90:	00	09	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A0:	00	0A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96
B0:	00	0B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C0:	00	0C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D0:	00	0D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E0:	00	0E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2
F0:	00	0F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

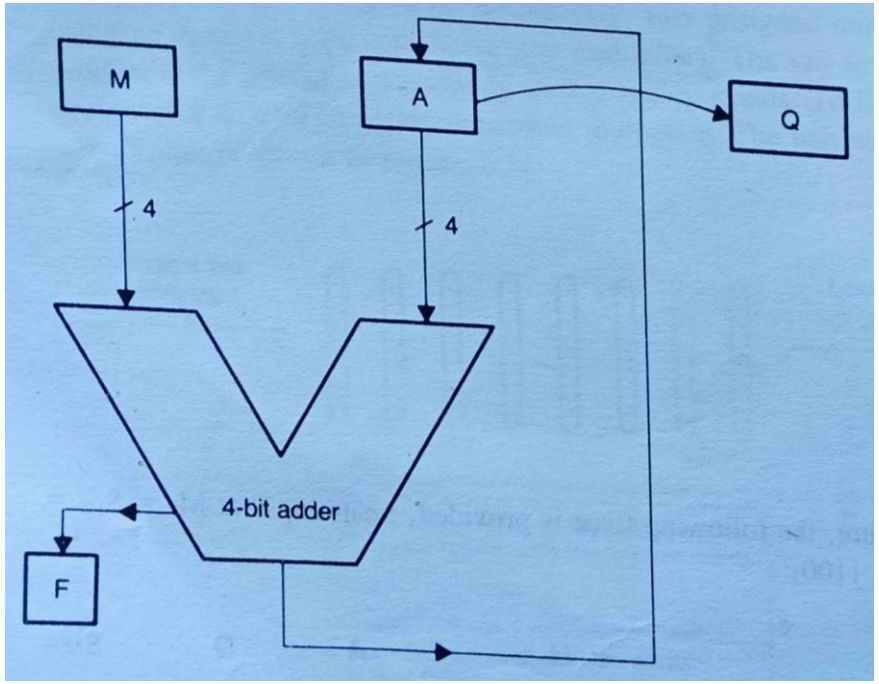
What is the memory size of this ROM?

265 bytes

Multiplication of Binary numbers

Sequential multiplier

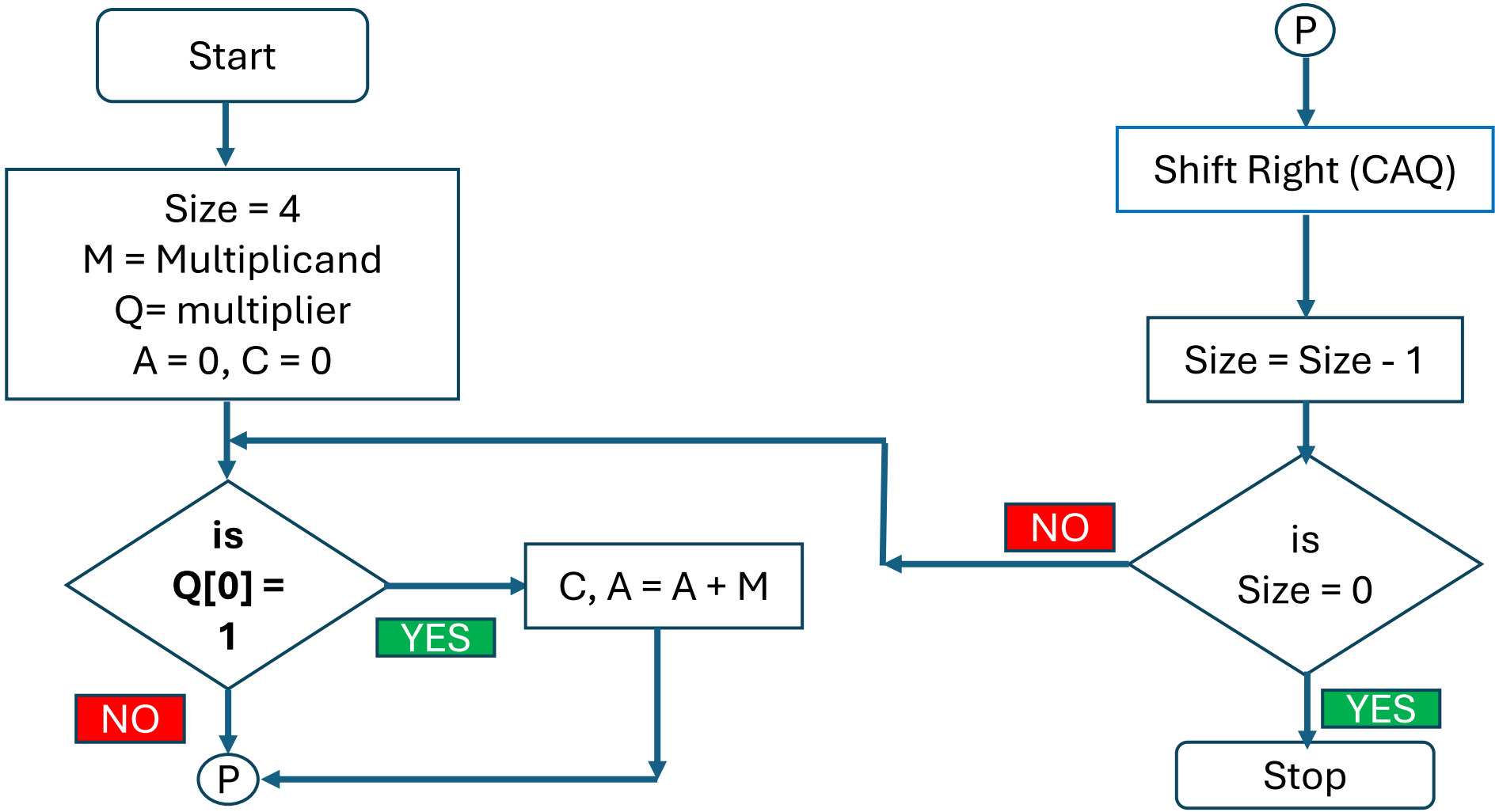
- Sequential multiplier use the fact that multiplication can be achieved using repeated addition
- It can multiply unsigned integers
- Next slides has a flowchart that can multiply two 4-bit unsigned numbers and a block diagram for implementing it
- **4x4 unsigned sequential multiplier**



Multiplication of Binary numbers

Sequential multiplier

Algorithm to multiply two unsigned 4-bit numbers



Multiplication of Binary numbers

Sequential multiplier

- **Example:** M = 1100, Q = 1001

Description	C	A	Q	Q[0]
Initialization	0	0000	1001	1
Ite-1, A = A+M	0	1100	1001	1
SHR CAQ	0	0110	0100	0
Ite-2, No add	0	0110	0100	0
SHR CAQ	0	0011	0010	0
Ite-3, No add	0	0011	0010	0
SHR CAQ	0	0001	1001	1
Ite-4, A=A+M	0	1101	1001	1
SHR CAQ	0	0110	1100	0

0xC*0x9 = 0x6C

Multiplication of Binary numbers

- **Sequential multiplier**

M = 01110, Q = 10100

Description	C	A	Q	Q[0]
Initialization	0	00000	10100	0
Ite-1, SHR CAQ	0	00000	01010	0
Ite-2, SHR CAQ	0	00000	00101	1
Ite-3 A=A+M	0	01110	00101	1
SHR CAQ	0	00111	00010	0
Ite-4, SHR CAQ	0	00011	10000	1
Ite- 5 A=A+M	0	10001	10000	1
SHR CAQ	0	01000	11000	0

Worst case: n additions and n right shift operations are needed
Speed increased

0x0E*0x14 = 0x118

Multiplication of Binary numbers

Signed multiplication

- There are three possible cases:
  1. M&Q are in signed magnitude form
  2. M and Q are in one's complement form
  3. M and Q are in Two's complement form
- **Case 1:**
  - Multiply the magnitudes either combinationally or sequentially
  - Sign of the product = sign of Multiplier XOR sig of Multiplicand
- **Case 2:**
  - Step 1: if multiplicand is negative, then compute the 1's complement of M
  - Step 2: if multiplier is negative, then compute the 1's complement of Q
  - Step 3: multiply the (n – 1) bits of the multiplier and the multiplicand either combinnationally or sequentially
  - Step 4: sign of the product = sign of multiplier XOR sign of multiplicand
  - Step 5: if sign of the product is negative, then compute the 1's of the result obtained in Step 3

Multiplication of Binary numbers

Signed multiplication

- **Case 3:**
  - When M & Q are in 2's complement form, the same procedure as in **Case 2** is repeated with the following exception:
  - Product must be 2's complemented when
    - both multiplicand and multiplier are negative or
    - sign of Multiplicand XOR Sign of Multiplier = 1
    - ***Example: M=1100 and Q=0111, take 2'c complement of M, then multiply with Q. since sign of product is –ve, take 2's complement of product.***
    - ***Result: 11100100***
  - **Demerits:**
  - extra processing when they are represented in 1's or 2's complement form
  - Overhead is maximum when multiplier and multiplicand are in 2's complement form, since incrementation is required besides the complementation.
  - Overhead can be eliminated by ***recoded multiplication approach***

Multiplication of Binary numbers

Booth's Recoded multiplication approach

- **String property:**
- *In a binary sequence a block of consecutive K ones may be replaced with a block of k - 1 consecutive 0's surrounded by the digits **1** and $\overline{1}$*

5 ones

0 0 $\overbrace{1\,1\,1\,1\,1}$ 0

By the string property it may be rewritten as follows;

0 0 11111 0

$\Downarrow$

0 1 0000$\overline{1}$ 0

- String property increases the density of zeros in a given multiplier
- number of additions operations to be performed is reduced
- allows for quick multiplication

- ***If a weight of -1 is assigned to the digit $\overline{1}$(over bar), assuming that the original sequence is in 2's complement form, both represent same number***

$$-2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0$$

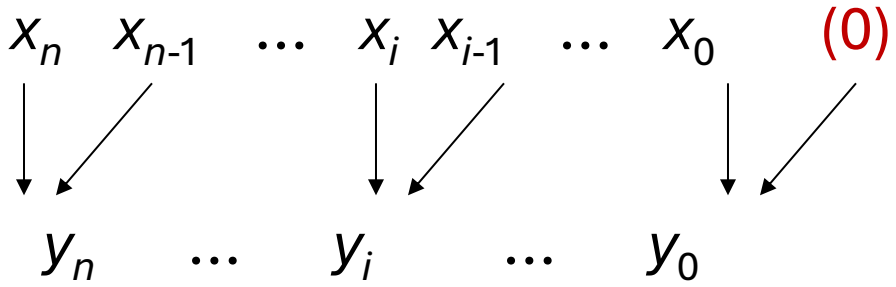
Original pattern: $0\,0\,1\,1\,1\,1\,1\,0 = 2^5 + 2^4 + 2^3 + 2^2 + 2^1 = 62_{10}$

Recoded pattern: $0\,1\,0\,0\,0\,0\,\overline{1}\,0 = 1.2^6 + (-1)2^1 = 62_{10}$

Multiplication of Binary numbers

Booth's Recoded multiplication approach

- Booth Observed that a String of 1's May be Replaced as: $2^j + 2^{j-1} + \cdots + 2^{i+1} + 2^i = 2^{j+1} - 2^i$



$y_i = x_{i-1} - x_i$

x_i	x_{i-1}	Comments	y_i
0	0	string of zeros	0
1	1	string of ones	0
1	0	beg. string of ones	-1
0	1	end string of ones	1

EXAMPLE: 0 0 1 1 1 1 0 0 1 1 (0)

0 1 0 0 0 $\bar{1}$ 0 1 0 $\bar{1}$

Booth's Recoding Drawbacks

- case can come up where number of 1s originally will be less than the recoded 1's

001010101 (0)

01$\bar{1}$1$\bar{1}$1$\bar{1}$1$\bar{1}$

Recode 15

01111 10001

Multiplication of Binary numbers

Booth's Algorithm

- Booth's Algorithm is a technique used in binary multiplication to optimize the process, especially for signed numbers.
- It reduces the number of partial products, which can simplify and speed up the multiplication.

- ***Key Concepts***
- **Recoding:** Booth's Algorithm recodes the multiplier to reduce the number of partial products. This is done by examining pairs of bits in the multiplier.
- **Handling Signed Numbers:** It efficiently manages both positive and negative multipliers by using 2's complement representation.
- **Efficiency with Consecutive Ones:** The algorithm is particularly effective when the multiplier has consecutive ones, converting them into fewer operations.

Multiplication of Binary numbers

Booth's Algorithm

1. Initialization:

- Set up the multiplicand (A) and the multiplier (B).
- Compute the negative of the multiplicand (-A) using 2's complement.

2. Recoding:

- Examine pairs of bits in the multiplier (B) along with an extra bit (initially 0).
- Depending on the pair of bits, decide whether to add, subtract, or do nothing with the multiplicand.

3. Partial Product Generation:

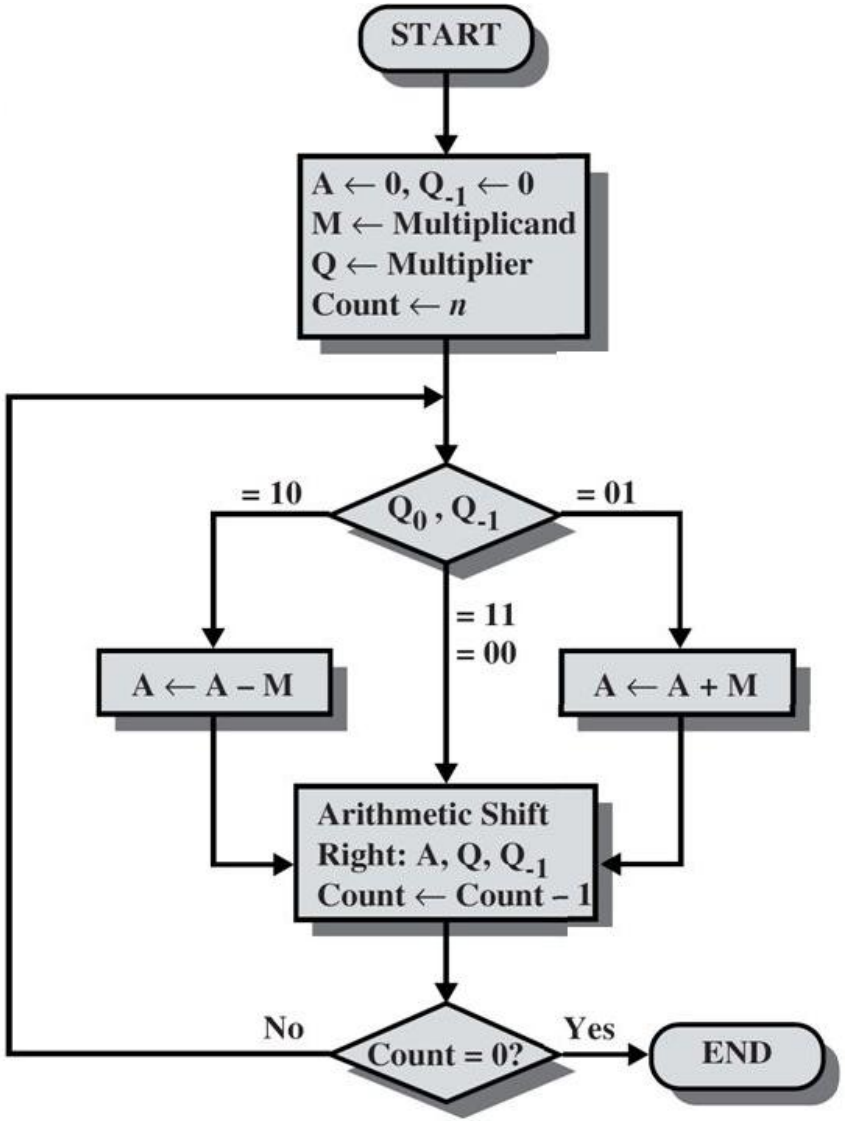
- Generate partial products based on the recoding step.
- Shift the partial products appropriately.

4. Summation:

- Sum the partial products to get the final result.

Multiplication of Binary numbers

Booth's Algorithm



- Starting from right to left, look at *two adjacent bits* of the multiplier
  - Before starting, place a zero at the right of the LSB
- If bits = 00, **do nothing**
- If bits = 10 [read from LSB side], **subtract** the multiplicand from the product
  - Beginning of a string of 1's
- If bits = 11, **do nothing**
  - Middle of a string of 1's
- If bits = 01 [read from LSB side], **add** the multiplicand to the product
  - End of a string of 1's
- apply **arithmetic right shift** by one bit on product register

Multiplication of Binary numbers

Booth's Algorithm

- Example: Multiply the numbers -- M = 0101, Q = 0111

Description	A	Q	q_{-1}
Initialization	0000	0111	0
1: A = A - M (subtract)	1011	0111	0
ASR AQ	1101	1011	1
2: ASR AQ	1110	1101	1
3: ASR AQ	1111	0110	1
4: A = A + M (Addition)	0100	0110	1
ASR AQ	0010	0011	0

5*7 = 23H

Multiplication of Binary numbers

Booth's Algorithm

- **Example:** *Multiply M=-9 and Q=-13*
- **2's complement of -13 = 10011 and 2's complement of -9 = 10111**

A	Q	Q_{-1}	M	Operation
00000	10011	0	10111	initial stage
01001	10011	0	10111	A<-A-M
00100	11001	1	10111	ASR (1st iteration)
00010	01100	1	10111	ASR (2nd iteration)
11001	01100	1	10111	A<-A+M
11100	10110	0	10111	ASR (3rd iteration)
11110	01011	0	10111	ASR (4th iteration)
00111	01011	0	10111	A<-A-M
00011	**10101**	1	10111	ASR (5th iteration)

- **Product : 0001110101 = $(117)_{10}$**

Multiplication of Binary numbers

Booth's Algorithm

- **Example: Multiply -8 and 12**
- **2's complement of -8 = 11000 and 12 = 01100**

A	Q	Q_{-1}	M	operation
00000	01100	0	11000	initial stage
00000	00110	0	11000	ASR (1^{st} iteration)
00000	00011	0	11000	ASR (2^{nd} iteration)
01000	00011	0	11000	A<-A-M
00100	00001	1	11000	ASR (3^{rd} iteration)
00010	00000	1	11000	ASR (4^{th} iteration)
11010	00000	1	11000	A<-A+M
11101	**00000**	0	11000	ASR (5^{th} iteration)

- **Product : 2's complement of $(1110100000)_2 = (-96)_{10}$**

Multiplication of Binary numbers

Drawbacks to Booth's Algorithm

- Variable number of add/subtract operations and of shift operations between two consecutive add/subtract operations
  - Inconvenient when designing a synchronous multiplier
- Algorithm inefficient with isolated 1's
- **Example:**
- $00101010\,1(0)$ recoded as $01\bar{1}1\bar{1}1\bar{1}1\bar{1}$, requiring 8 instead of 4 operations
- Situation can be improved by examining 3 bits at a time rather than 2

Adder Design

- **Carry Look-Ahead Adder (CLA)**

COMPUTER ORGANIZATION AND ARCHITECTURE

Memory Organization

Dr. Bore Gowda S B
Additional Professor
Dept. of ECE
MIT, Manipal

Topics to be covered

- Memory Hierarchy
- Main Memory
- Auxiliary Memory
- Associative Memory
- Cache Memory,
- Virtual Memory
- Memory Management

- **Reference book:** M Morris Mano, “Computer System Architecture”, 3rd Edition

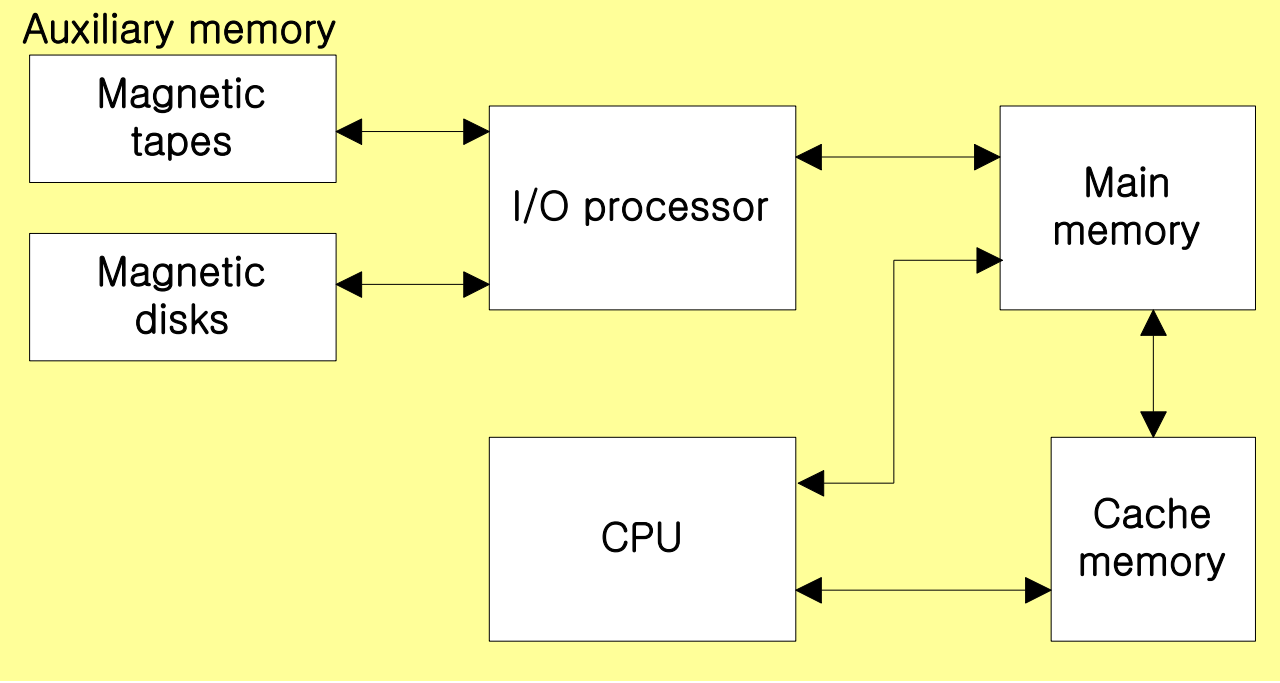
Chapter 12 - Memory Organization

Memory Hierarchy

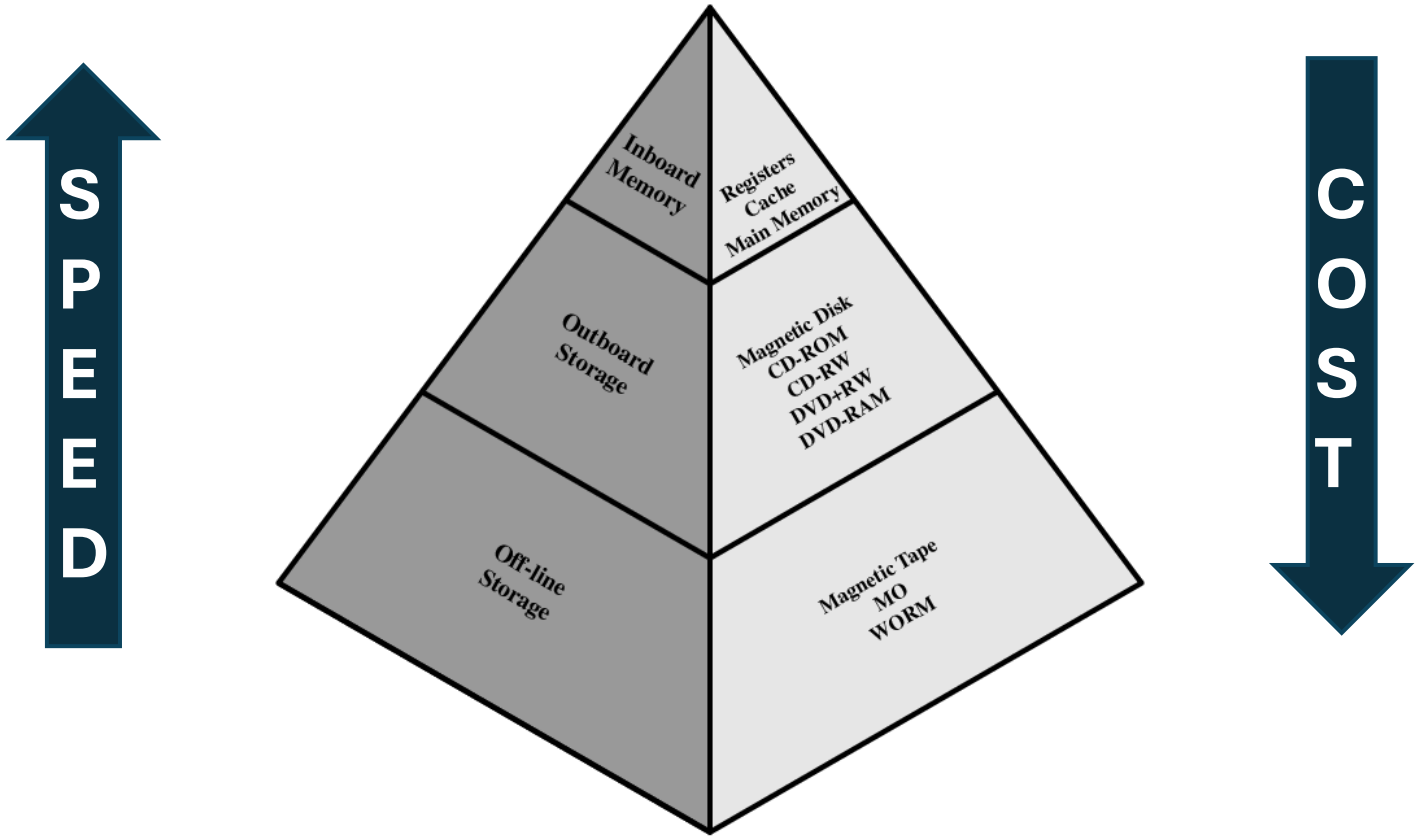
- Memory hierarchy in a computer system
- **Main memory:** The memory unit that communicates directly with the CPU
- **Auxiliary memory:** Devices that provide backup storage
  - The most common auxiliary memory devices used in computer systems are magnetic disks and tapes.
- **Cache:** It is high speed memory used to increase the speed of processing by making current programs and data available to CPU at a rapid rate.
- The total memory capacity of a computer can be visualized as being a hierarchy of components.
- The memory hierarchy system consists of all storage devices employed in a computer system from the slow but high capacity auxiliary memory to a relatively faster main memory to an even smaller Cache memory.

Memory Hierarchy

- **Typical memory hierarchy**
- At the bottom of the hierarchy are the relatively slow magnetic tapes used to store removable files.
- Next are the magnetic disks used as backup storage.
- The main memory able to communicate directly with the CPU and with auxiliary memory devices through an IO processor.
- When programs not residing in main memory are needed by the CPU, they are brought in from auxiliary memory.
- Programs not currently needed in main memory are transferred into auxiliary memory to provide space for currently used programs and data.
- A special very-high speed memory called a Cache is sometimes used to increase the speed of processing by making current programs and data available to the CPU at a rapid rate.



Memory Hierarchy



Memory Hierarchy

- Registers – semiconductor memory on the CPU; fastest
- L1 Cache – Static RAM placed very close to CPU
- L2 Cache – static RAM placed close to CPU
- Main memory – dynamic RAM placed on board
- Disk – for bulk data
- Optical – for bulk data
- Tape – for bulk data

Main Memory

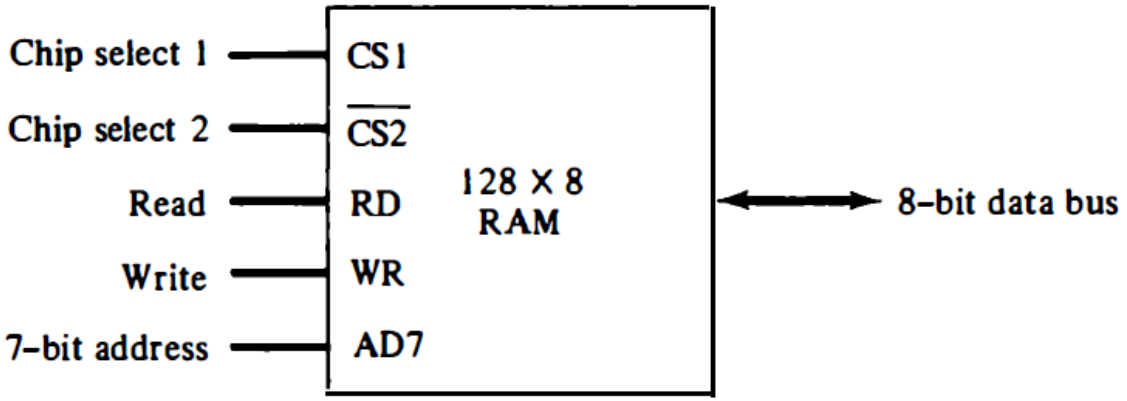
- It is the central storage unit in a computer system.
- Relatively large and fast memory used to store programs and data during the computer operation.
- Main memory is based on semiconductor integrated circuits.
- ICs RAM chips are available in two possible operating modes,
  - Static RAM and dynamic RAM.
- **Static RAM**
  - consists essentially of internal flip-flops
  - stored information remains valid as long as power is applied to the unit.
- **Dynamic RAM**
  - stores the binary information in the form of electric charges that are applied to capacitors.
  - The periodically capacitors are recharged by refreshing the dynamic memory.
  - Refreshing is done by cycling through the words every few milliseconds to restore the decaying charge.
- The dynamic RAM offers reduced power consumption and larger storage capacity in a single memory chip.
- The static RAM is easier to use and has shorter read and write cycles.

Main Memory

- A portion of the memory may be constructed with ROM chips.
- ROM is used for storing programs that are permanently resident in the computer
- ROM portion of main memory is needed for storing an initial program called a **bootstrap loader.**
- The bootstrap loader is a program whose function is to start the computer software operating when power is turned on.
- The bootstrap program loads a portion of the operating system from disk to main memory and control is then transferred to the operating system
- RAM is volatile, but ROM is non-volatile

Main Memory

Typical RAM Chips

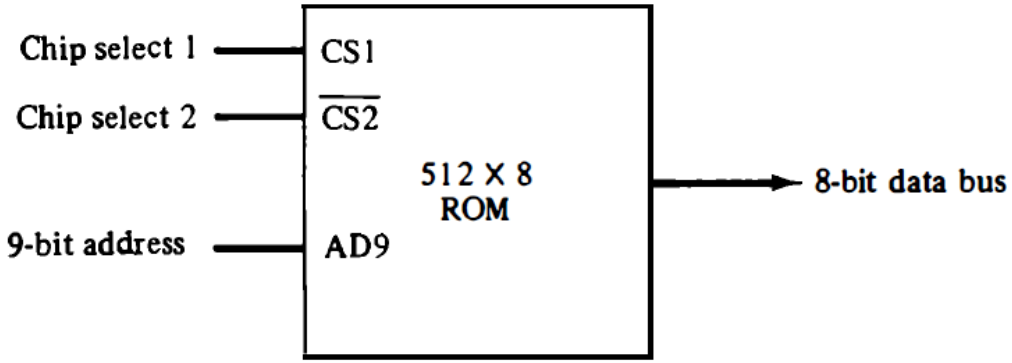


(a) Block diagram

CS1	$\overline{CS2}$	RD	WR	Memory function	State of data bus
0	0	×	×	Inhibit	High-impedance
0	1	×	×	Inhibit	High-impedance
1	0	0	0	Inhibit	High-impedance
1	0	0	1	Write	Input data to RAM
1	0	1	×	Read	Output data from RAM
1	1	×	×	Inhibit	High-impedance

(b) Function table

Typical ROM Chips



Main Memory

- **Memory Address Map**
- The designer of a computer system must calculate the amount of memory required for the particular application and assign it to either RAM or ROM.
- The interconnection between memory and processor is then established from knowledge of the size of memory needed and the type of RAM and ROM chips available.
- The addressing of memory can be established by means of a table that specifies the memory address assigned to each chip.
- The table, called a memory address map, is a pictorial representation of assigned address space for each chip in the system.

Main Memory

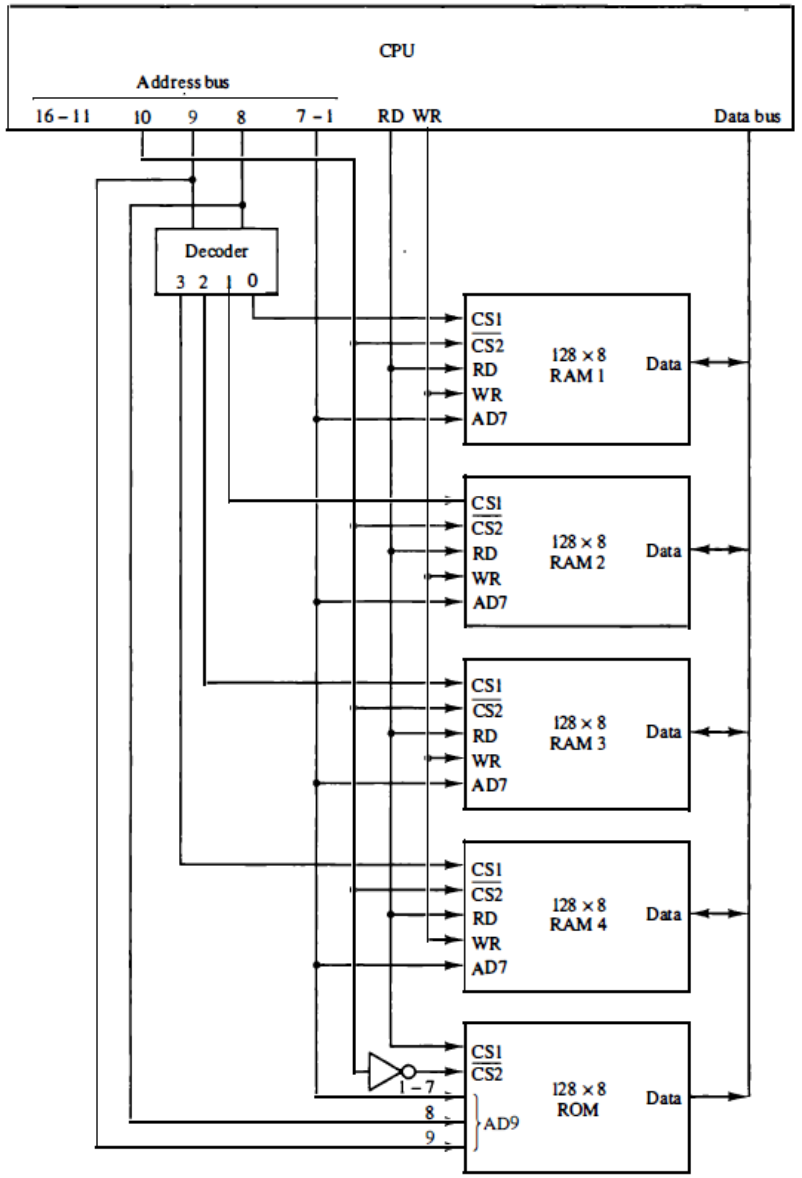
- **Memory Interfacing**
- *The microprocessor in the computer system has 16-bit address lines and 8-bit data lines. Design a computer system with 512 bytes of RAM and 512 bytes of ROM. The memory chips with following capacity are available: 128x8 RAM chip and 512x8 ROM chip*

TABLE 12-1 Memory Address Map for Microprocomputer

Component	Hexadecimal address	Address bus 10	9	8	7	6	5	4	3	2	1
RAM 1	0000–007F	0	0	0	x	x	x	x	x	x	x
RAM 2	0080–00FF	0	0	1	x	x	x	x	x	x	x
RAM 3	0100–017F	0	1	0	x	x	x	x	x	x	x
RAM 4	0180–01FF	0	1	1	x	x	x	x	x	x	x
ROM	0200–03FF	1	x	x	x	x	x	x	x	x	x

Main Memory

Memory Interfacing

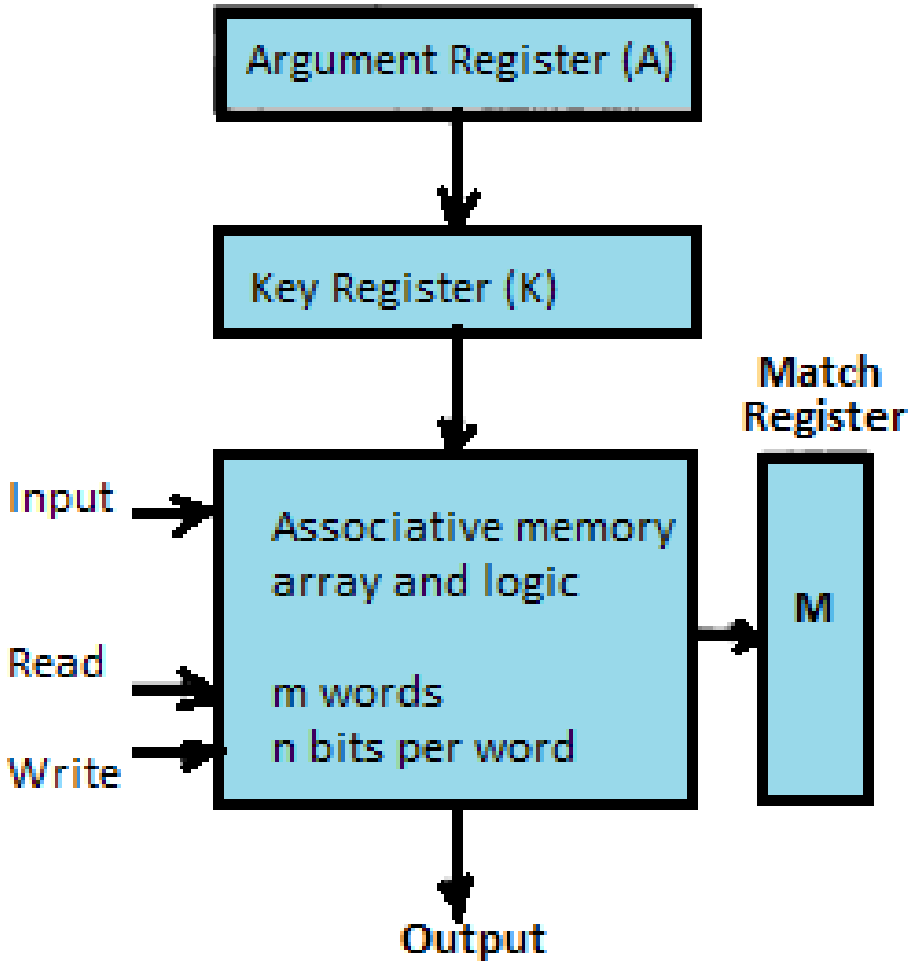


Associative Memory

- Many data-processing applications require the search of items in a table stored in memory.
- The time required to find an item stored in memory can be reduced considerably if *stored data can be identified for access by the content of the data itself rather than by an address*.
- *A memory unit accessed by content is called an associative memory or content addressable memory (CAM).*
- This type of memory is accessed simultaneously and in parallel on the basis of data content rather than by specific address or location.
- When a word is written in an associative memory, no address is given.
- The memory is capable of finding an empty unused location to store the word.
- When a word is to be read from an associative memory, the content of the word, or part of the word, is specified.
- The memory locates all words which match the specified content and marks them for reading.
- Searches can be done on an entire word or on a specific field within a word.
- An associative memory is more expensive than a random access memory because each cell must have storage capability as well as logic circuits for matching its content with an external argument.
- For this reason, associative memories are used in applications where the search time is very critical and must be very short.

Associative Memory

- **Hardware Organization**
- It consists of a memory array and logic for ***m words with n bits per word***.
- A - argument register
- K - key register: key register provides a mask for choosing a particular field or key in the argument word.
- A and K each have n bits, one for each bit of a word.
- The match register M has m bits, one for each memory word.
- **Procedure for searching**
- Each word in memory is compared in parallel with the content of the argument register.
- The words that match the bits of the argument register set a corresponding bit in the match register.
- After the matching process, those bits in the match register that have been set indicate the fact that their corresponding words have been matched.
- Reading is accomplished by a sequential access to memory for those words whose corresponding bits in the match register have been set.



Associative Memory

- **Numerical example.**
- Let A 101 1 1 1 1 0 0 and K 111 0 0 0 0 0 0

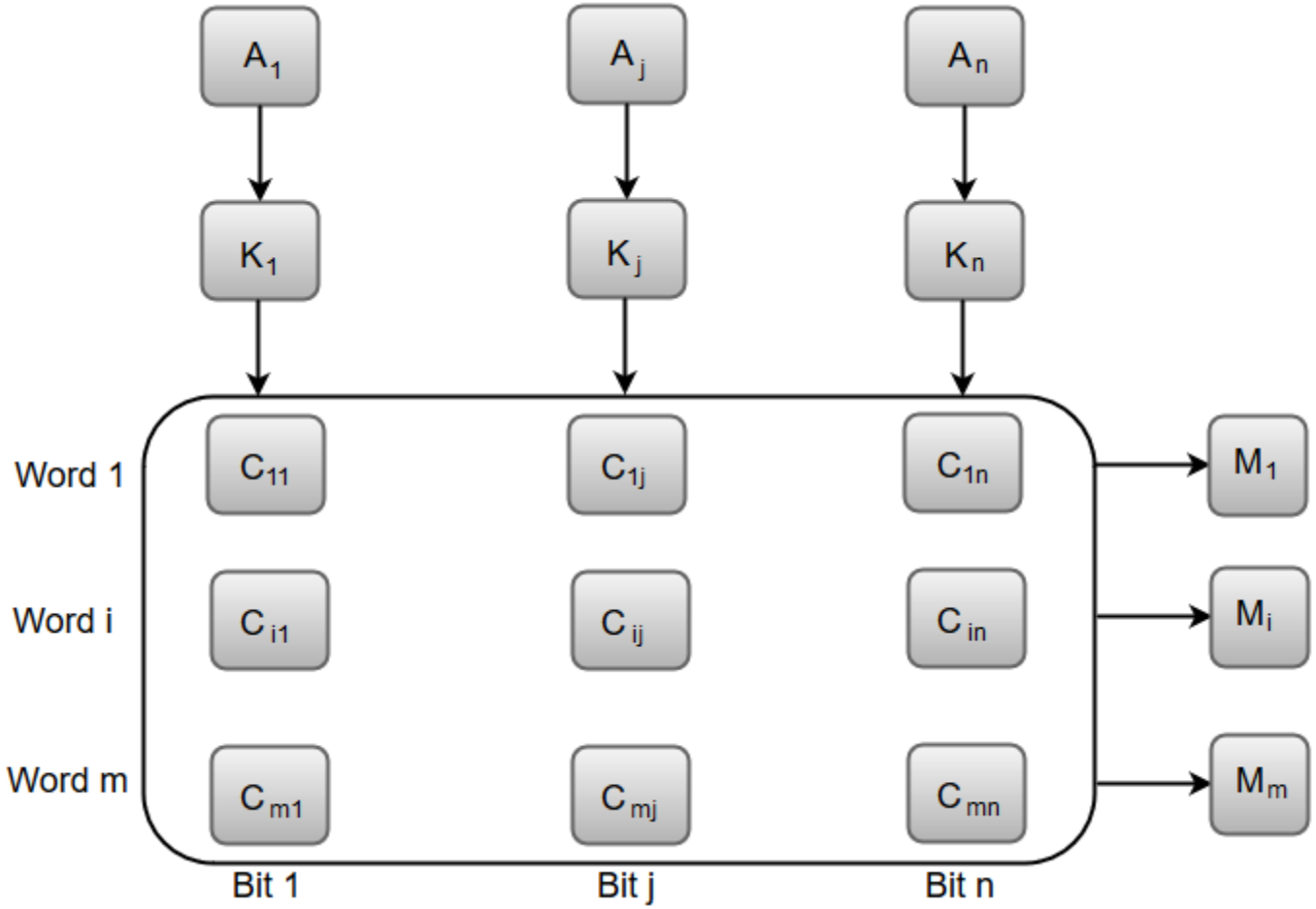
A	101	111100	
K	111	000000	
Word 1	100	111100	no match
Word 2	101	000001	match

- Word 2 matches the unmasked argument field because the three leftmost bits of the argument and the word are equal.

Associative Memory

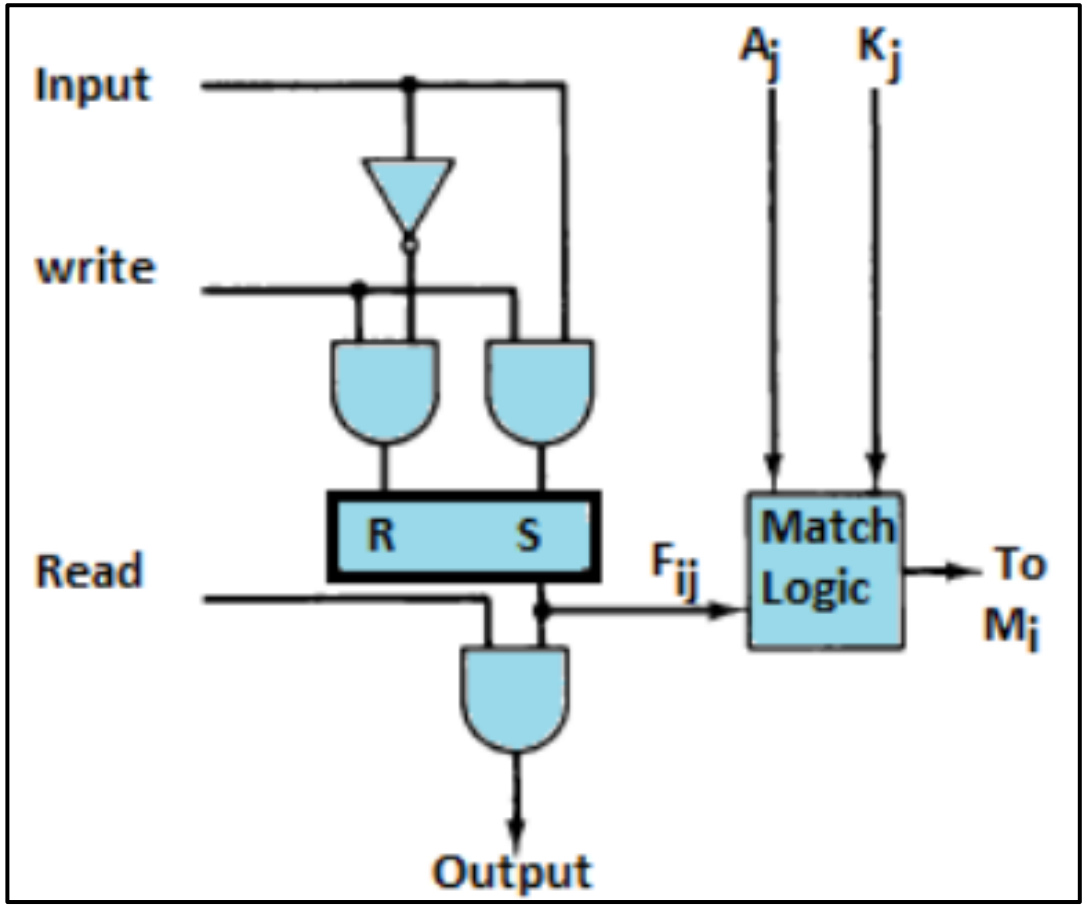
Associative memory of m word, n cells per word

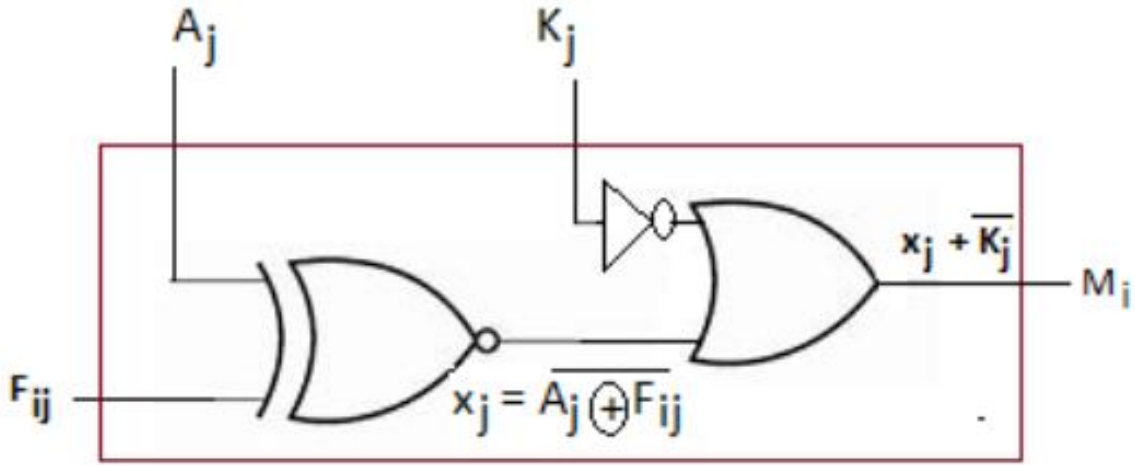
- The cells in the array are marked by the letter C with two subcripts. The first subscript gives the word number and the second specifies the bit position in the word.



Associative Memory

- One cell of associative memory





Associative Memory

- **Match Logic**
- First, we neglect the key bits and compare the argument in A with the bits stored in the cells of the words.
- Two bits are equal if they are both 1 or both 0. The equality of two bits can be expressed logically by the Boolean function:

$$x_j = A_j F_{ij} + A_j' F_{ij}'$$

where $x_j = 1$ if the pair of bits in position j are equal; otherwise, $x_j = 0$.

- For a word i to be equal to the argument in A we must have all x_j variables equal to 1. This is the condition for setting the corresponding match bit M, to 1. The Boolean function for this condition is

$$M_i = x_1 x_2 x_3 \cdots x_n$$

constitutes the AND operation of all pairs of matched bits in a word.

Associative Memory

- **Match Logic**
- We now include the key bit K_j in the comparison logic.
- The requirement is that if:

  $K_j = 0$, the corresponding bits of A_j and F_{ij} need no comparison

  $K_j = 1$, the corresponding bits of A_j and F_{ij} need comparison

- This is achieved by ORing each term:

$$x_j + K_j' = \begin{cases} x_j & \text{if } K_j = 1 \\ 1 & \text{if } K_j = 0 \end{cases}$$

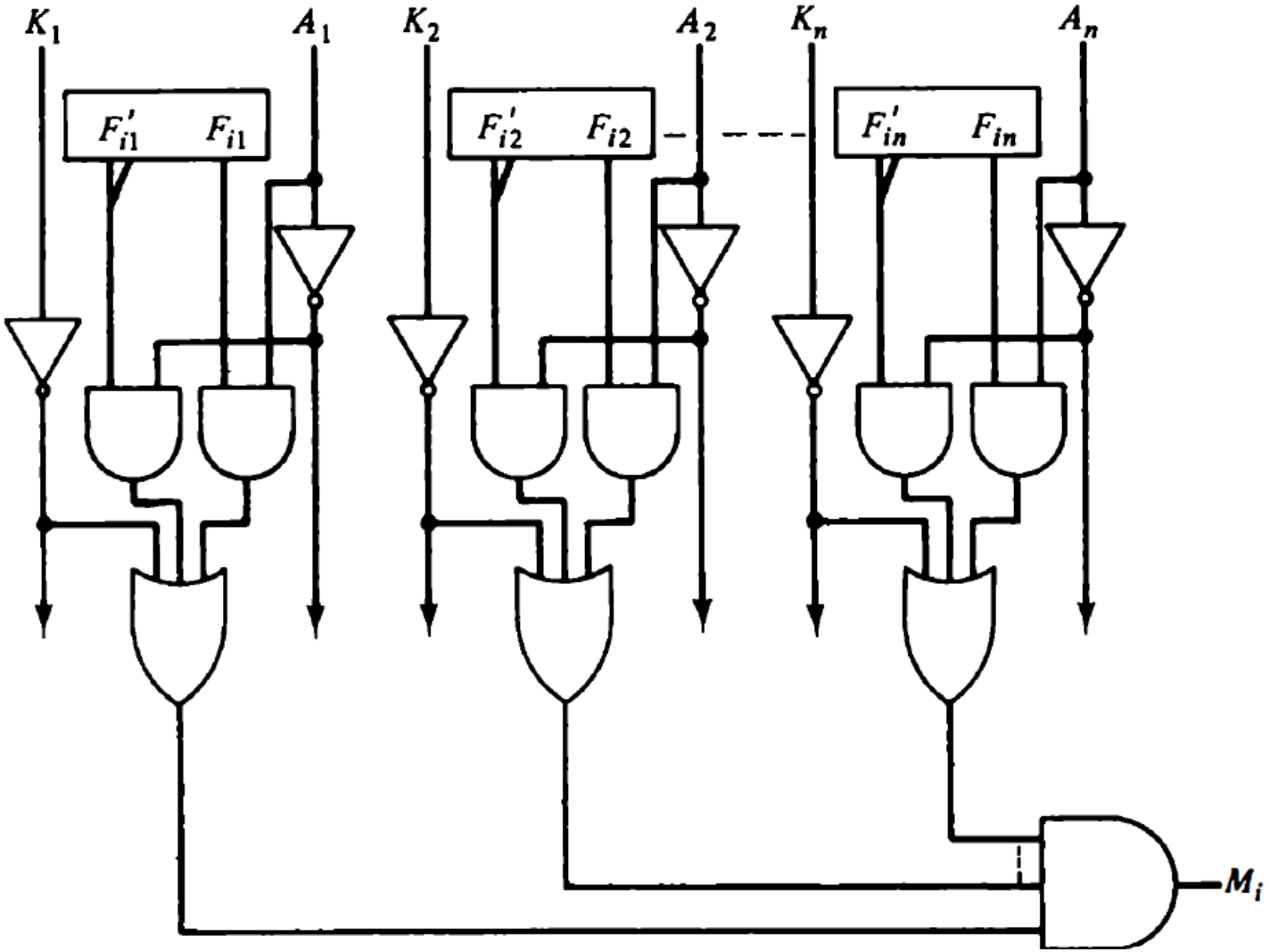
- The match logic for word ***i*** in an associative memory can now be expressed by the following Boolean function:

$$M_i = (x_1 + K_1')(x_2 + K_2')(x_3 + K_3') \cdots (x_n + K_n')$$

- I f we substitute the original definition o f x_j, the Boolean function above can be expressed as follows:

$$M_i \models \prod_{j=1}^{n} (A_j F_{ij} + A_j' F_{ij}' + K_j')$$

- Match logic for one word of associative memory



Auxiliary Memory

- The most common auxiliary memory devices used: ***magnetic disks and tapes***.
- Other components used, but not as frequently: ***magnetic drums, magnetic bubble memory, and optical disks***
- The important characteristics of any device are its ***access mode***, ***access time***, ***transfer rate***, ***capacity***, and ***cost***.
- **Access time:** The average time required to reach a storage location in memory and obtain its contents
- **Transfer rate:** the number of characters or words that the device can transfer per second
- In electromechanical devices with moving parts such as disks and tapes:

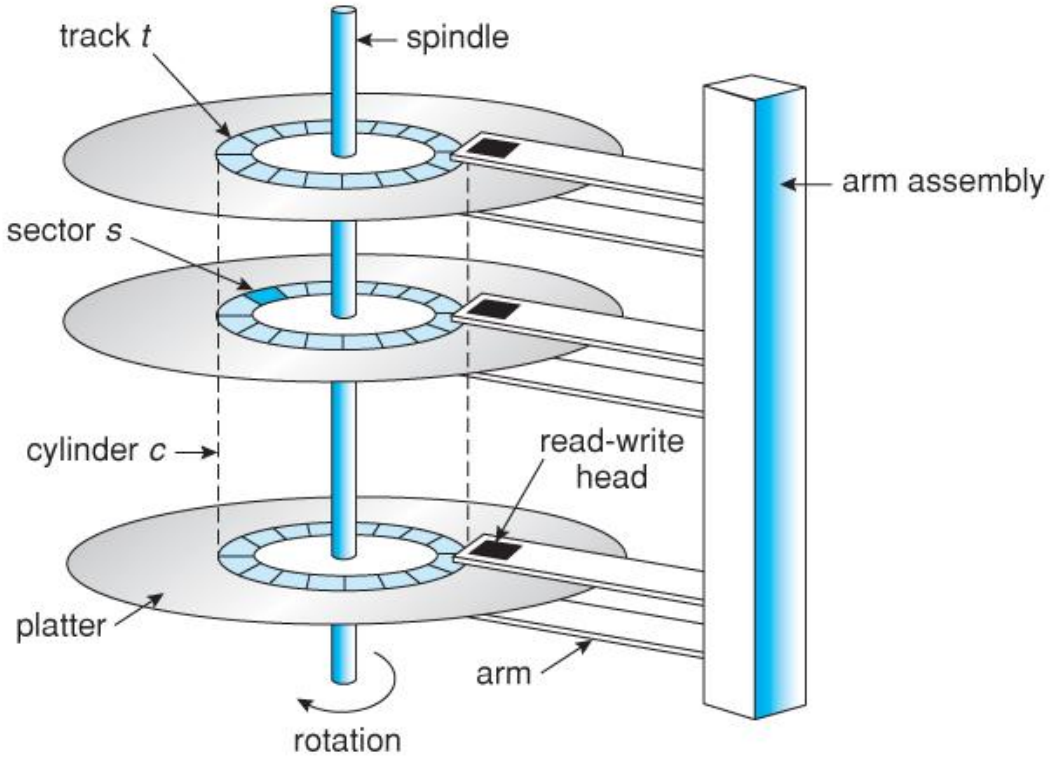
Access time = seek time + transfer time

Auxiliary Memory

- Magnetic drums and disks are quite similar in operation.
- Both consist of high-speed rotating surfaces coated with a magnetic recording medium.
- The rotating surface of the drum is a cylinder and that of the disk, a round flat plate.
- The recording surface rotates at uniform speed and is not started or stopped during access operations.
- Bits are recorded as magnetic spots on the surface as it passes a stationary mechanism called a **write head**.
- Stored bits are detected by a change in magnetic field produced by a recorded spot on the surface as it passes through a **read head**.
- The amount of surface available for recording in a disk is greater than in a drum of equal physical size.
- Disks have replaced drums in more recent computers

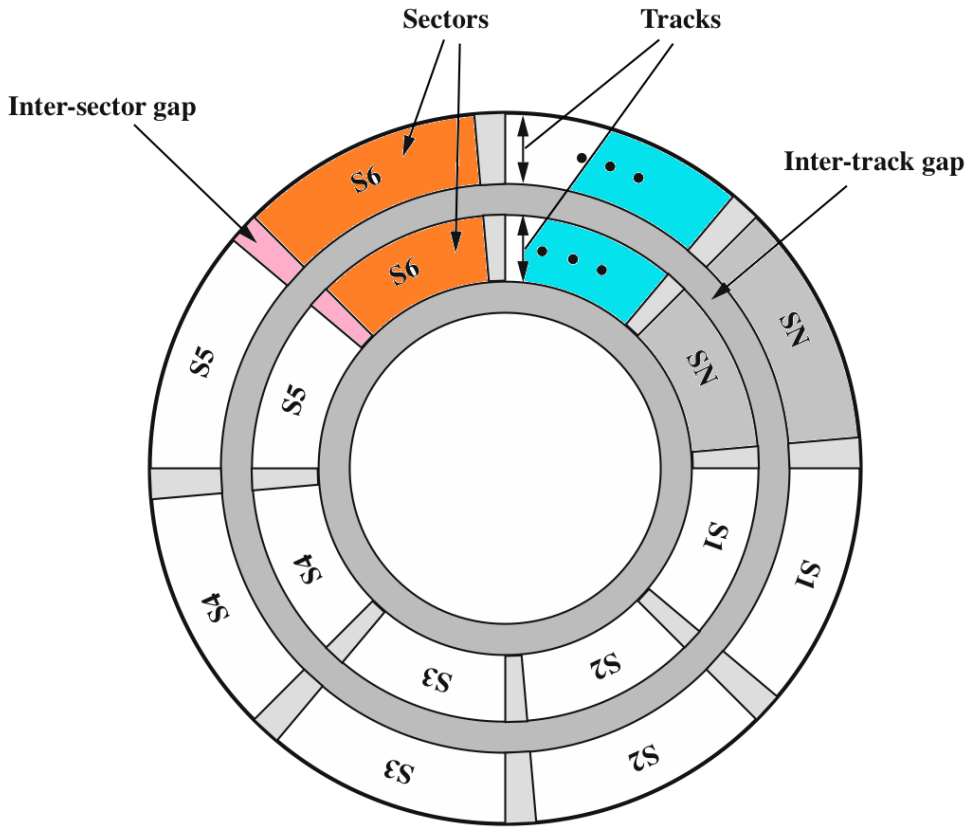
Auxiliary Memory - Magnetic Disks

- It is a circular plate constructed of metal or plastic coated with magnetized material.
- Often both sides of the disk are used
- Several disks may be stacked on one spindle with read/write heads available on each surface.
- All disks rotate together at high speed and are not stopped or started for access purposes.



Auxiliary Memory - Magnetic Disks

- Bits are stored in the magnetized surface in spots along concentric circles called **tracks**.
- The tracks are commonly divided into sections called **sectors**.
- In most systems, the minimum quantity of information which can be transferred is a sector.
- A disk system is addressed by address bits that specify the disk number, the disk surface, the sector number and the track within the sector.
- the track address bits are used by a mechanical assembly to move the head into the specified track position before reading or writing
- The address bits can then select a particular track electronically through a decoder circuit.



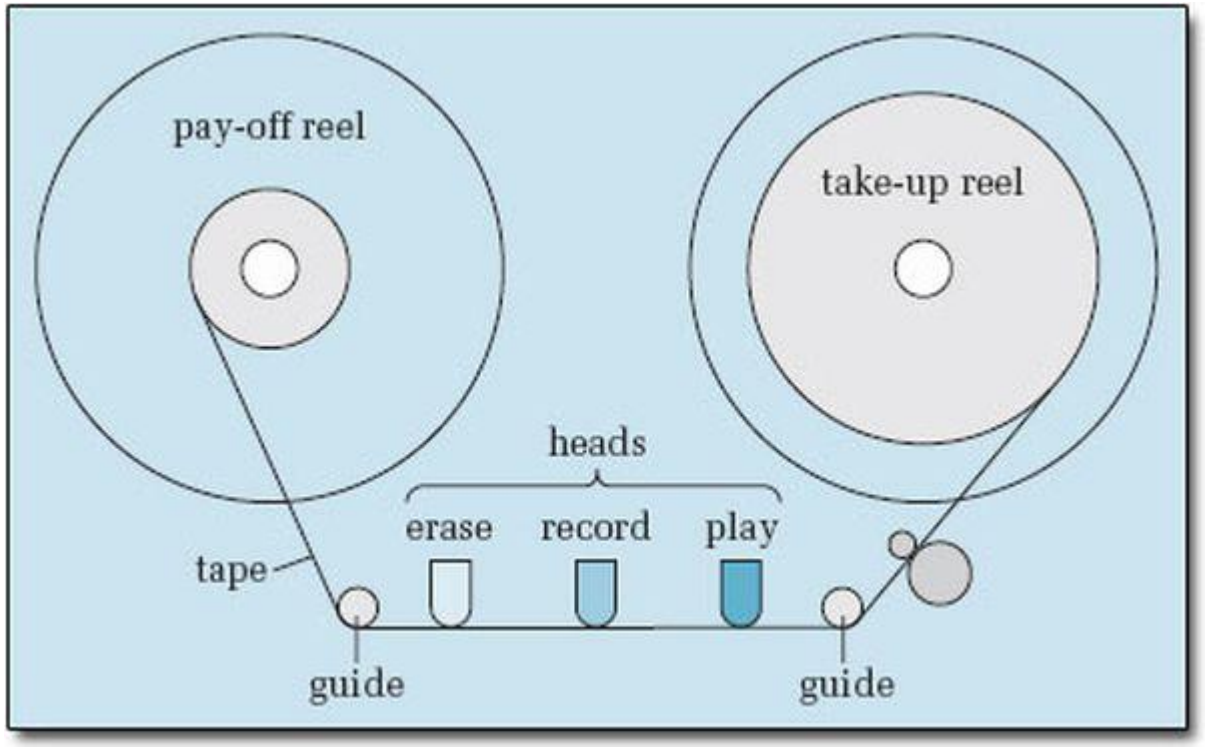
Auxiliary Memory - Magnetic Disks

- A track in a given sector near the circumference is longer than a track near the center of the disk
- If bits are recorded with equal density, some tracks will contain more recorded bits than others.
- To make all the records in a sector of equal length, some disks use a variable recording density with higher density on tracks near the center than on tracks near the circumference.
- ***Hard disks:*** Disks that are permanently attached to the unit assembly and cannot be removed
- ***Floppy disk:*** A disk drive with removable disks
  - Common sizes with diameters of 5.25 and 3. 5 inches.
  - Floppy disks are extensively used in personal computers as a medium for distributing software to computer users.

Auxiliary Memory - Magnetic Tape

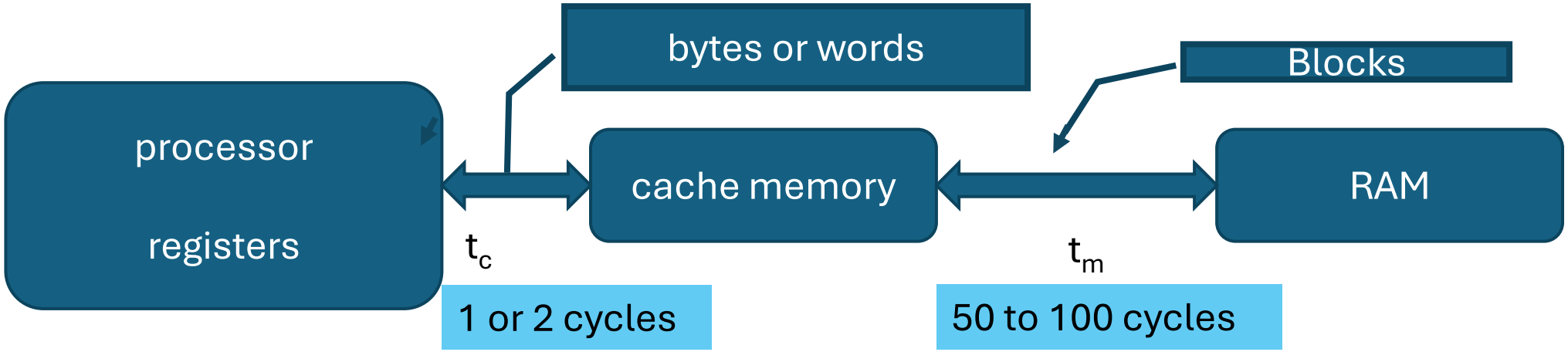
- The tape is a strip of plastic coated with a magnetic recording medium.
- Bits are recorded as magnetic spots on the tape along several tracks.
- Usually, seven or nine bits are recorded simultaneously to form a character together with a parity bit.
- Read/write heads are mounted one in each track so that data can be recorded and read as a sequence of characters
- Magnetic tape units can be stopped, started to move forward or in reverse, or can be rewound.
- They cannot be started or stopped fast enough between individual characters
- Information is recorded in blocks referred to as records.
- Gaps of unrecorded tape are inserted between records where the tape can be stopped.
- The tape starts moving while in a gap and attains its constant speed by the time it reaches the next record.
- Each record on tape has an identification bit pattern at the beginning and end.
- A tape unit is addressed by specifying the record number and the number of characters in the record.
- Records may be of fixed or variable length.

Auxiliary Memory - Magnetic Tape



Cache Memory

- Small amount of fast memory
- Sits between main memory (RAM) and CPU
- May be located on CPU chip or in system
- Objective is to make slower memory system look like fast memory.
- The data or contents that are used frequently by CPU are stored in the cache memory
- Processor can easily access that data in a shorter time.
- Whenever the CPU needs to access memory, it first checks the cache memory.
- If the data is not found in cache memory, then the CPU moves into the main memory.



Cache Memory

Cache operations

- When the CPU needs to access memory, the cache is examined. If the word is found in the cache, it is read from the fast memory.
- If the word addressed by the CPU is not found in the cache, the main memory is accessed to read the word.
- A block of words once just accessed is then transferred from main memory to cache memory. The block size may vary from one word (the one just accessed) to about 16 words adjacent to the one just accessed.
- Performance of the cache memory is measured in terms of a quantity called ***hit ratio***.
- When the CPU refers to memory and finds the word in cache, it is said to produce a ***hit***.
- If the word is not found in the cache, it is in main memory and it counts as a ***miss***.

The ratio of the number of hits divided by the total CPU references to memory (hits plus misses) is the hit ratio.

Cache Memory

Cache operations

Hit ratio (h) = Total no. of Hits / Total no. of attempts

Efficiency is maximum when h = 1

But h = 0.9 is common (next example)

Let, t_C → cache-access time

h → hit ratio

t_m → main memory access time

The avg. access time $(t_{avg}) = h\, t_C + (1 - h)\,(t_C + t_m)$

$$\text{Efficiency} = t_C / t_{avg}$$

$$\text{Efficiency} = 1 / [\,1 + \Upsilon\ (1 - h)\,] \quad \text{where } \Upsilon = t_m / t_C$$

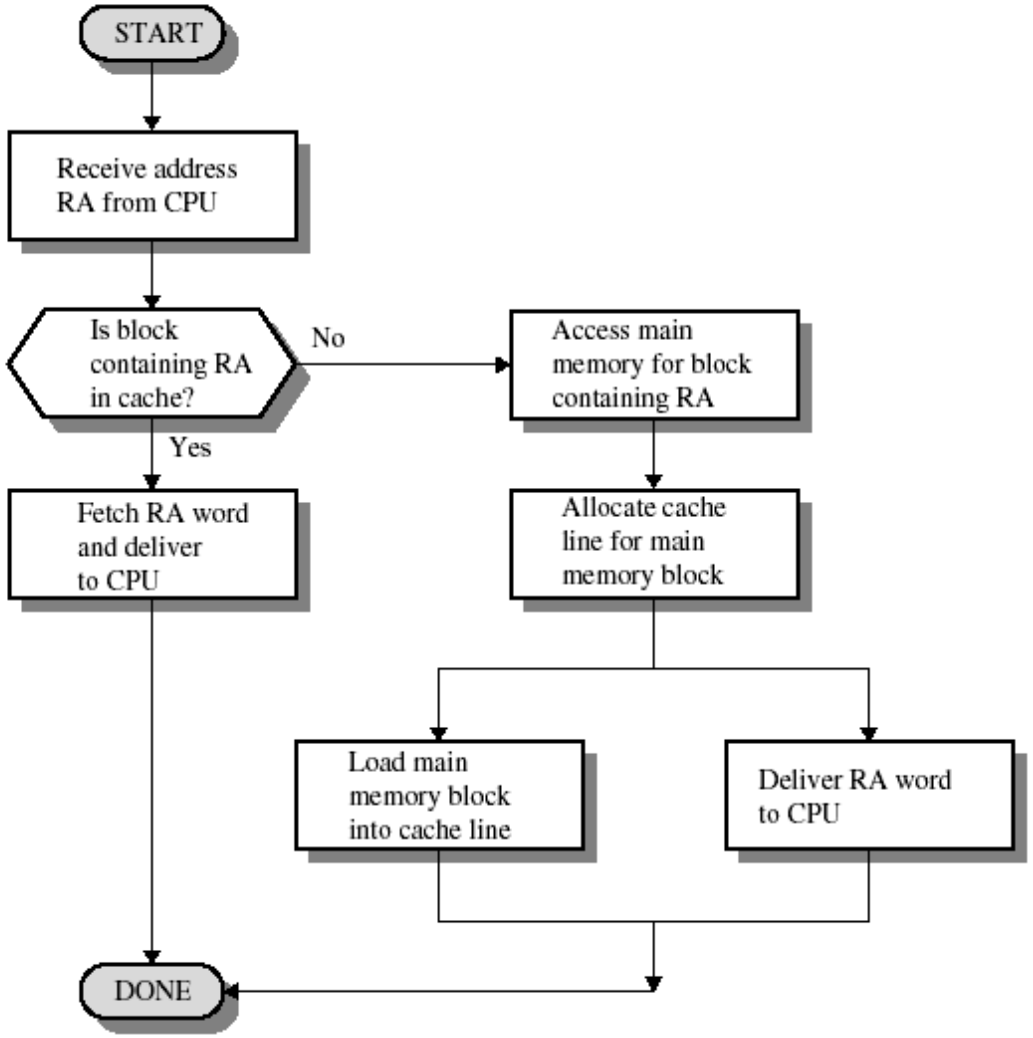
Access time (t_A): Avg. time taken to read a unit of information from the memory.

$$\text{Access rate } (r_A) = 1 / t_A$$

Cycle time (T_C): Avg. time lapse b/w two successive read operations.

$$\text{Data transfer rate (or BW), } r_C = 1/T_C$$

Cache operations



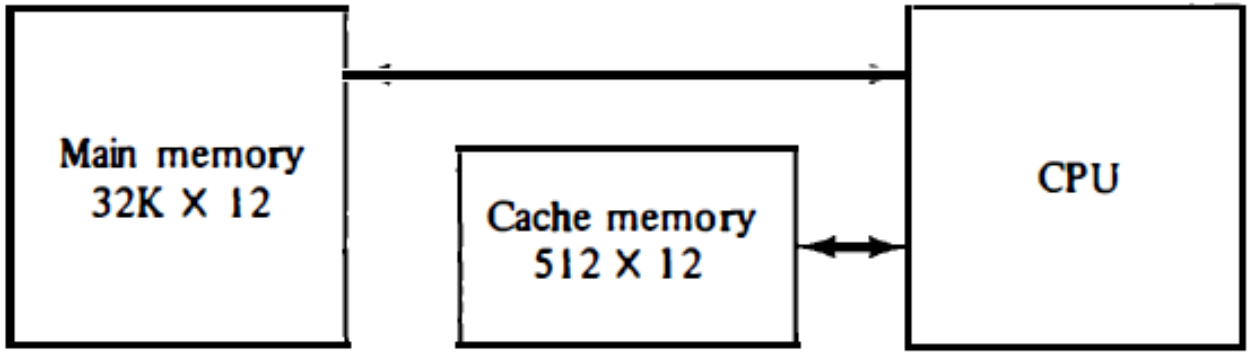
- The average memory access time of a computer system can be improved considerably by use of a cache.
- If the hit ratio is high enough so that most of the time the CPU accesses the cache instead of main memory,
- the average
- access time is closer to the access time of the fast cache memory.

Cache Memory

- The basic characteristic of cache memory is its fast access time.
- very little or no time must be wasted when searching for words in the cache.
- **Mapping process:** It is a process of transformation of data from main memory to cache memory
- Three types of mapping procedures
  1. Associative mapping
  2. Direct mapping
  3. Set-associative mapping

Cache Memory

- **Example of cache memory.**



For every word stored in cache, there is a duplicate copy in main memory.

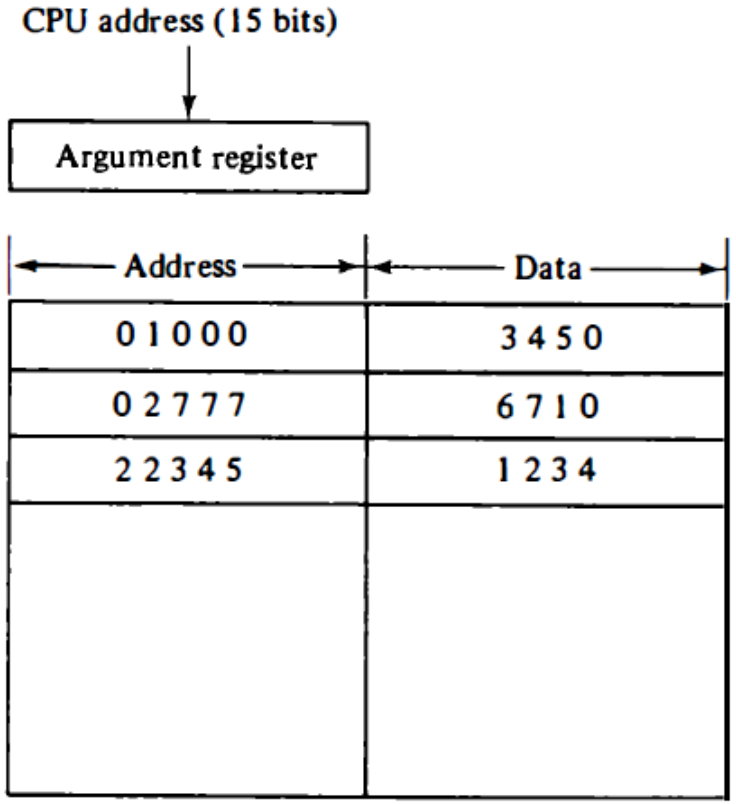
The CPU communicates with both memories.

1. It first sends a 15-bit address to cache.
2. If there is a hit, the CPU accepts the 12-bit data from cache.
3. If there is a miss, the CPU reads the word from main memory and the word is then transferred to cache.

Cache Memory

Associative Mapping

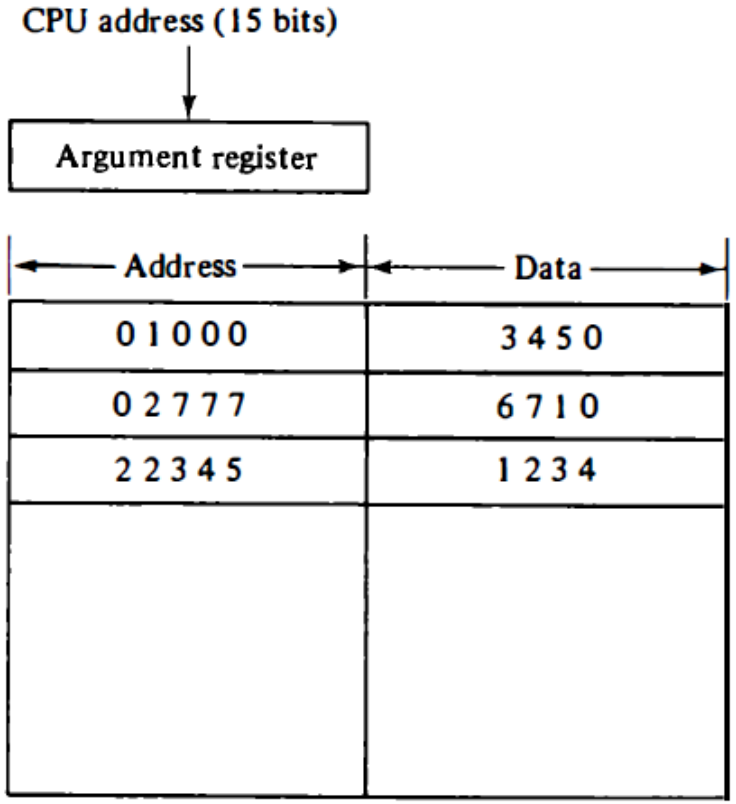
- The fastest and most flexible cache organization uses an associative memory.
- Associative memory stores both the **address and content (data) of the memory word**.
- **Example:**
- In the example Address and data word are represented in octal system
- address value of 15 bits is shown as a five-digit octal number and its corresponding 12 -bit word is shown as a four-digit octal number.
- A CPU address of 15 bits is placed in the argument register.
- *If the address is found, the corresponding 12-bit data is read and sent to the CPU.*
- *If no match occurs, the main memory is accessed for the word.*
- The address--data pair is then transferred to the associative cache memory.



Cache Memory

Associative Mapping

- The fastest and most flexible cache organization uses an associative memory.
- Associative memory stores both the **address and content (data) of the memory word**.
- **Example:**
- In the example Address and data word are represented in octal system
- address value of 15 bits is shown as a five-digit octal number and its corresponding 12 -bit word is shown as a four-digit octal number.
- A CPU address of 15 bits is placed in the argument register.
- *If the address is found, the corresponding 12-bit data is read and sent to the CPU.*
- *If no match occurs, the main memory is accessed for the word.*
- The address--data pair is then transferred to the associative cache memory.



Cache Memory