

Chapter – 1

Introduction

1.1 Computer Organization and Architecture

- Computer Architecture refers to those attributes of a system that have a direct impact on the logical execution of a program. Examples:
 - the instruction set
 - the number of bits used to represent various data types
 - I/O mechanisms
 - memory addressing techniques
- Computer Organization refers to the operational units and their interconnections that realize the architectural specifications. Examples are things that are transparent to the programmer:
 - control signals
 - interfaces between computer and peripherals
 - the memory technology being used
- So, for example, the fact that a multiply instruction is available is a computer architecture issue. How that multiply is implemented is a computer organization issue.
- Architecture is those attributes visible to the programmer
 - Instruction set, number of bits used for data representation, I/O mechanisms, addressing techniques.
 - e.g. Is there a multiply instruction?
- Organization is how features are implemented
 - Control signals, interfaces, memory technology.
 - e.g. Is there a hardware multiply unit or is it done by repeated addition?
- All Intel x86 family share the same basic architecture
- The IBM System/370 family share the same basic architecture
- This gives code compatibility
 - At least backwards
- Organization differs between different versions

1.2 Structure and Function

- Structure is the way in which components relate to each other
- Function is the operation of individual components as part of the structure
- All computer functions are:
 - **Data processing:** Computer must be able to process data which may take a wide variety of forms and the range of processing.
 - **Data storage:** Computer stores data either temporarily or permanently.
 - **Data movement:** Computer must be able to move data between itself and the outside world.
 - **Control:** There must be a control of the above three functions.

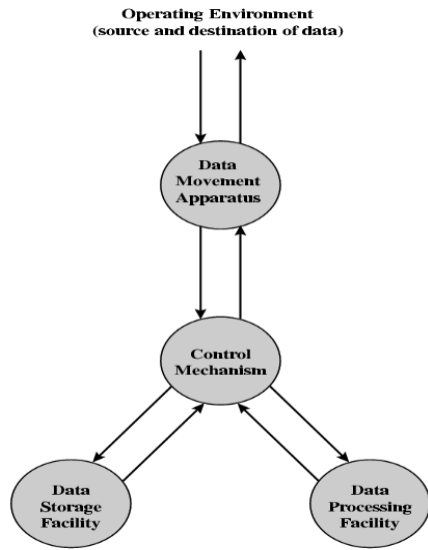


Fig: Functional view of a computer

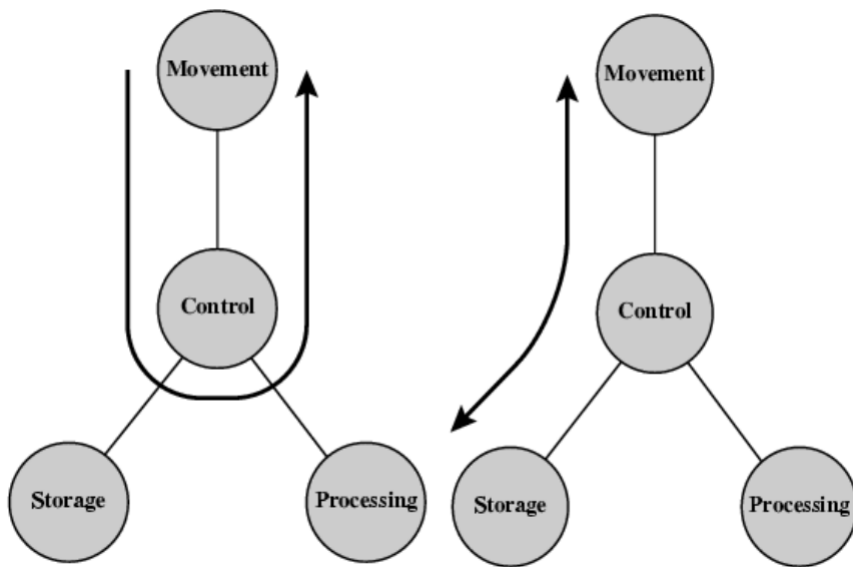


Fig: Data movement operation

Fig: Storage Operation

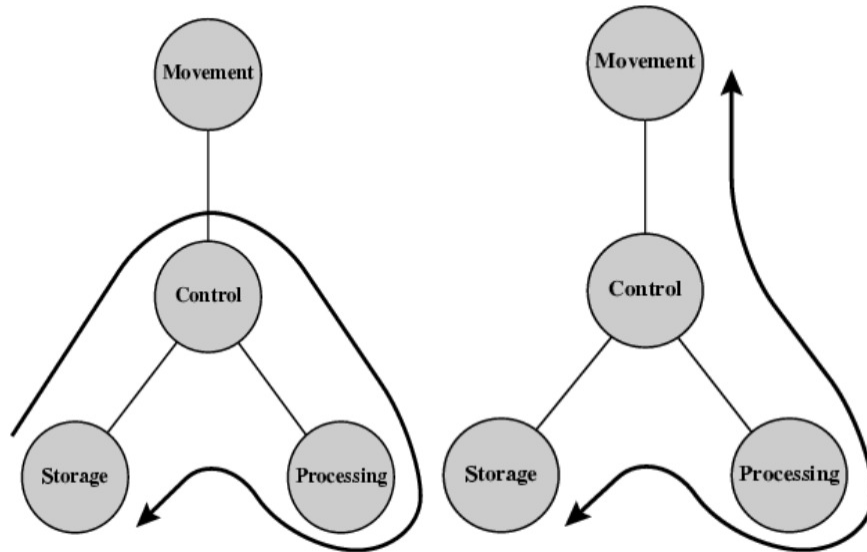


Fig: Processing from / to storage

Fig: Processing from storage to i/o

- Four main structural components:
 - Central processing unit (CPU)
 - Main memory
 - I / O
 - System interconnections
- CPU structural components:
 - Control unit
 - Arithmetic and logic unit (ALU)
 - Registers
 - CPU interconnections

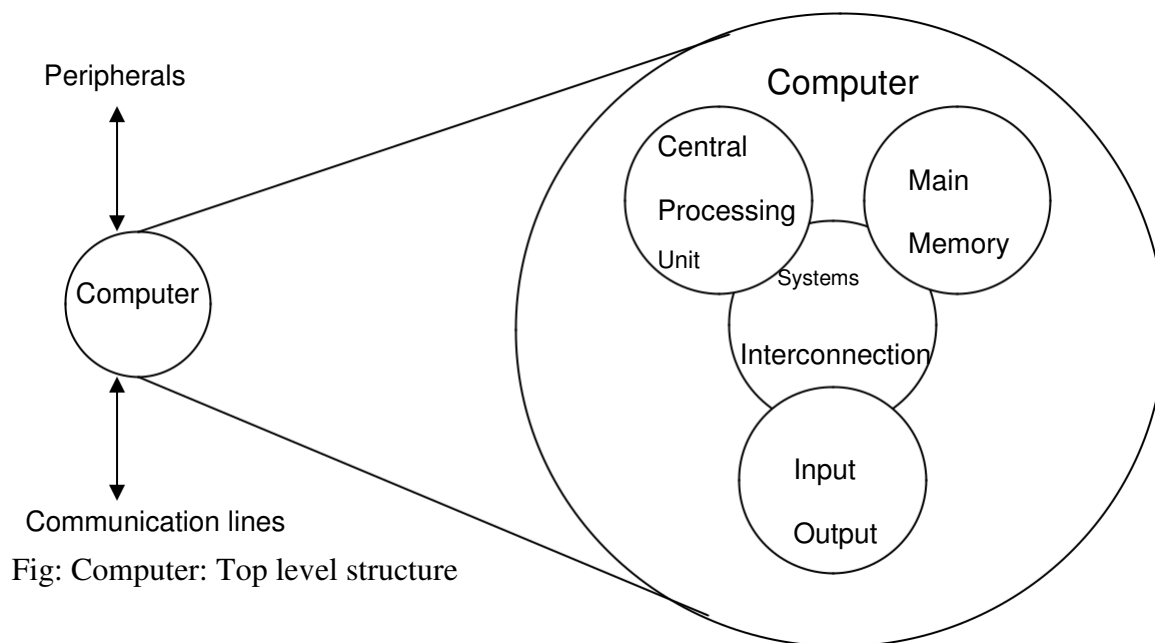


Fig: Computer: Top level structure

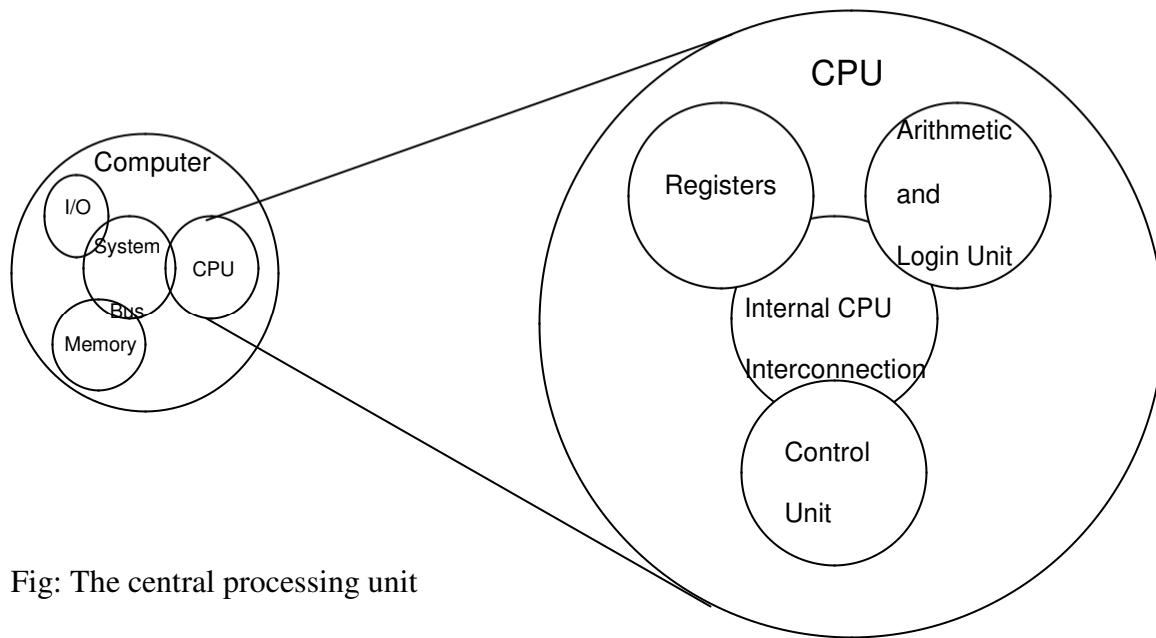


Fig: The central processing unit

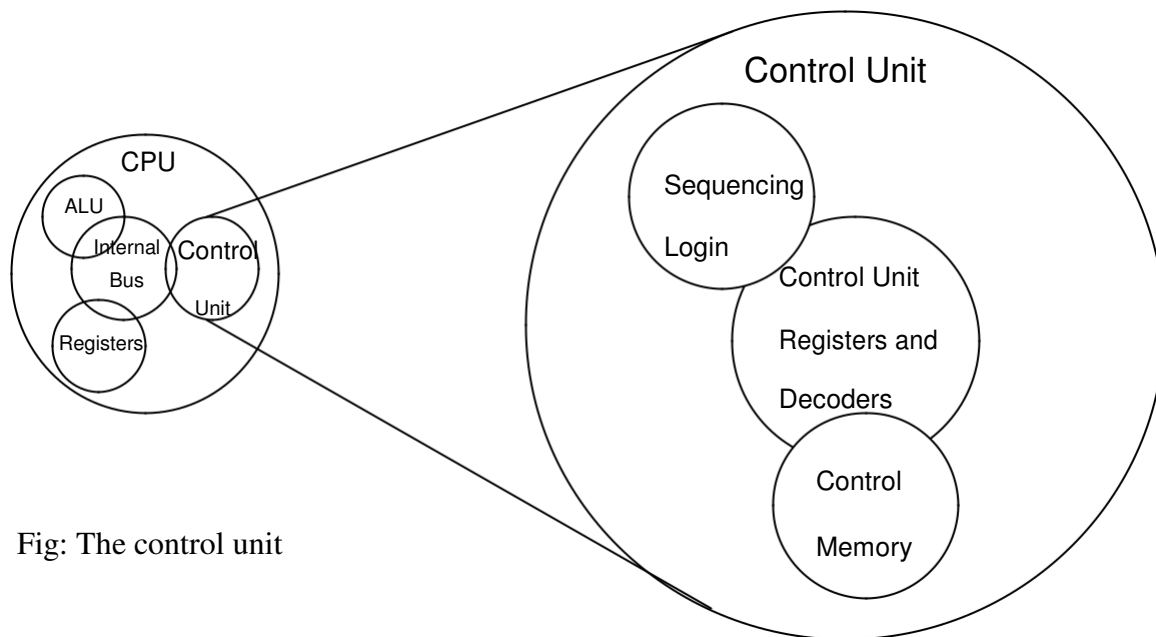


Fig: The control unit

1.3 Designing for performance

Some of the driving factors behind the need to design for performance:

- Microprocessor Speed
 - Pipelining
 - On board cache, on board L1 & L2 cache
 - Branch prediction: The processor looks ahead in the instruction code fetched from memory and predicts which branches, or group of instructions are likely to be processed next.
 - Data flow analysis: The processor analyzes which instructions are dependent on each other's results, or data, to create an optimized schedule of instructions to prevent delay.

- Speculative execution: Using branch prediction and data flow analysis, some processors speculatively execute instructions ahead of their actual appearance in the program execution, holding the results in temporary locations.
- Performance Mismatch
 - Processor speed increased
 - Memory capacity increased
 - Memory speed lags behind processor speed

Below figure depicts the history; while processor speed and memory capacity have grown rapidly, the speed with which data can be transferred between main memory and the processor has lagged badly.

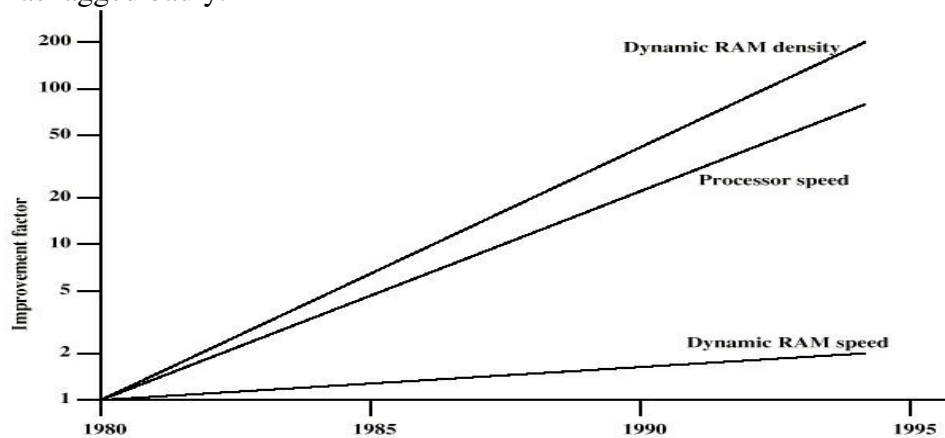


Fig: Evolution of DRAM and processor Characteristics

The effects of these trends are shown vividly in figure below. The amount of main memory needed is going up, but DRAM density is going up faster (number of DRAM per system is going down).

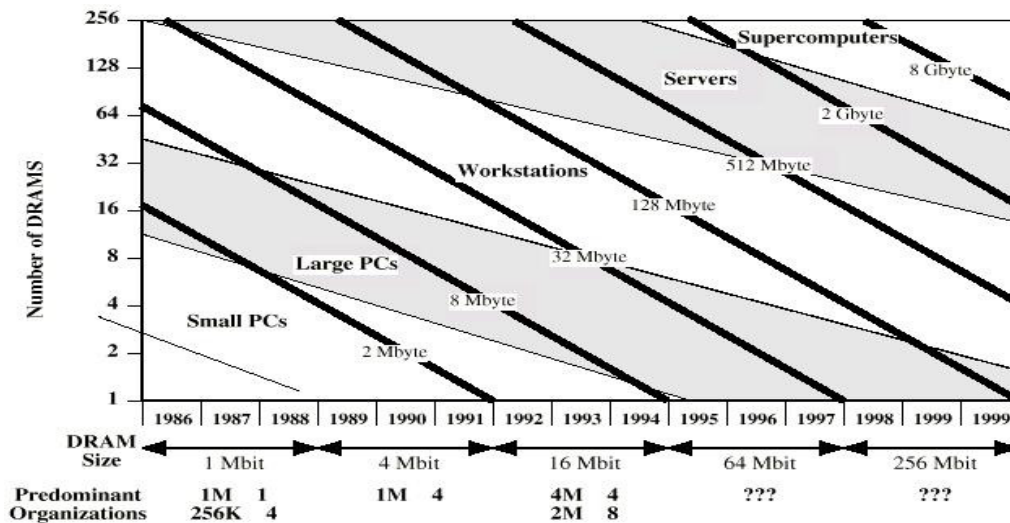


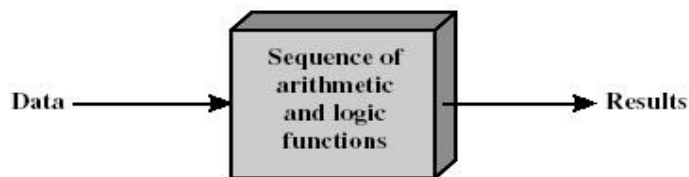
Fig: Trends in DRAM use

Solutions

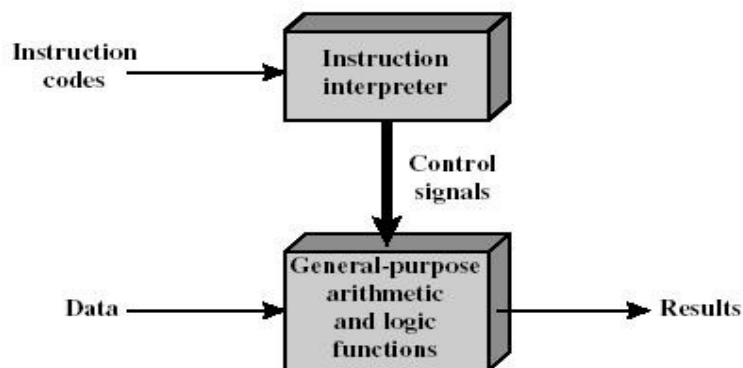
- Increase number of bits retrieved at one time
 - Make DRAM “wider” rather than “deeper” to use wide bus data paths.
- Change DRAM interface
 - Cache
- Reduce frequency of memory access
 - More complex cache and cache on chip
- Increase interconnection bandwidth
 - High speed buses
 - Hierarchy of buses

1.4 Computer Components

- The Control Unit (CU) and the Arithmetic and Logic Unit (ALU) constitute the Central Processing Unit (CPU)
- Data and instructions need to get into the system and results need to get out
 - Input/output (I/O module)
- Temporary storage of code and results is needed
 - Main memory (RAM)
- Program Concept
 - Hardwired systems are inflexible
 - General purpose hardware can do different tasks, given correct control signals
 - Instead of re-wiring, supply a new set of control signals



(a) Programming in hardware



(b) Programming in software

Fig: Hardware and Software Approaches

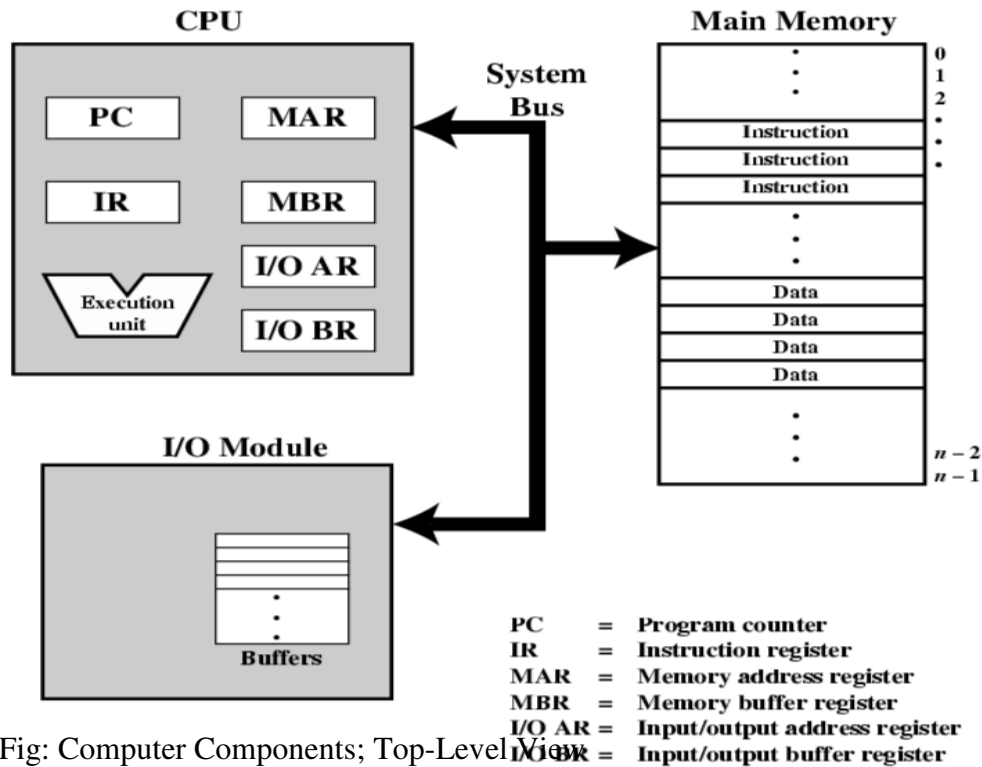


Fig: Computer Components; Top-Level View

1.5 Computer Function

The basic function performed by a computer is execution of a program, which consists of a set of instructions stored in memory.

- Two steps of Instructions Cycle:
 - Fetch
 - Execute

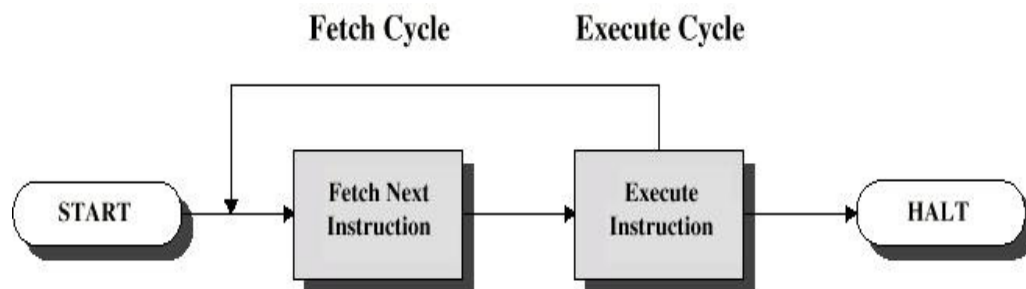


Fig: Basic Instruction Cycle

- Fetch Cycle
 - Program Counter (PC) holds address of next instruction to fetch
 - Processor fetches instruction from memory location pointed to by PC
 - Increment PC
 - Unless told otherwise
 - Instruction loaded into Instruction Register (IR)

- Execute Cycle
 - Processor interprets instruction and performs required actions, such as:
 - Processor - memory
 - data transfer between CPU and main memory
 - Processor - I/O
 - Data transfer between CPU and I/O module
 - Data processing
 - Some arithmetic or logical operation on data
 - Control
 - Alteration of sequence of operations
 - e.g. jump
 - Combination of above

Example of program execution.

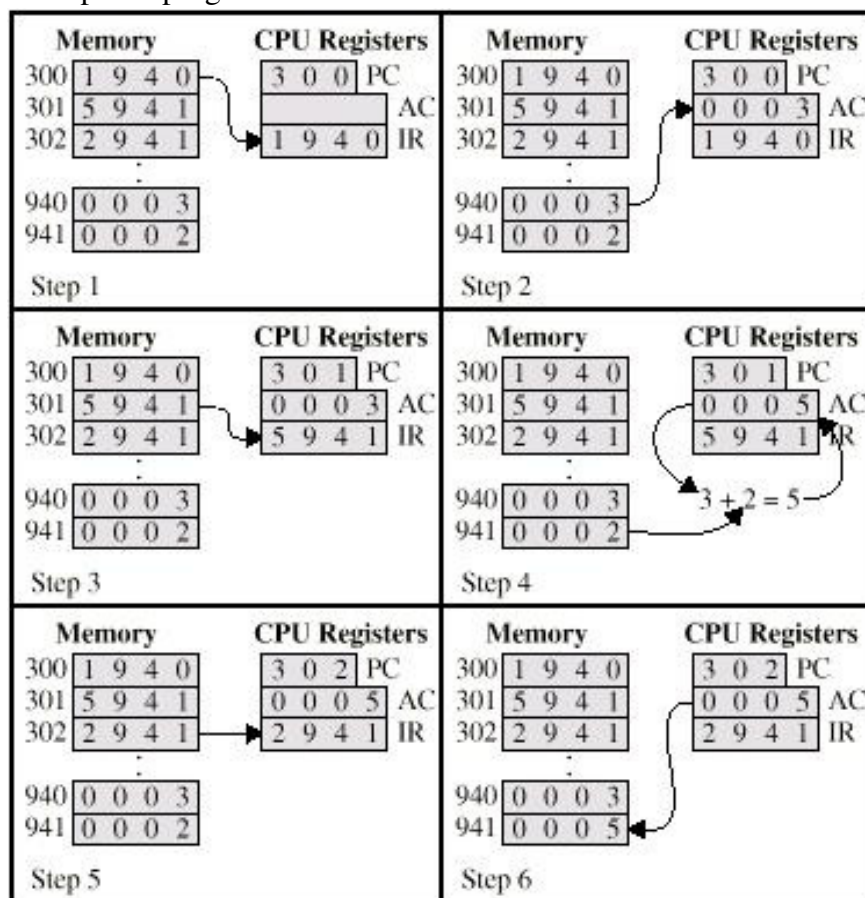


Fig: Example of program execution (consists of memory and registers in hexadecimal)

- The PC contains 300, the address of the first instruction. The instruction (the value 1940 in hex) is loaded into IR and PC is incremented. This process involves the use of MAR and MBR.
- The first hexadecimal digit in IR indicates that the AC is to be loaded. The remaining three hexadecimal digits specify the address (940) from which data are to be loaded.
- The next instruction (5941) is fetched from location 301 and PC is incremented.

- The old contents of AC and the contents of location 941 are added and the result is stored in the AC.
- The next instruction (2941) is fetched from location 302 and the PC is incremented.
- The contents of the AC are stored in location 941.

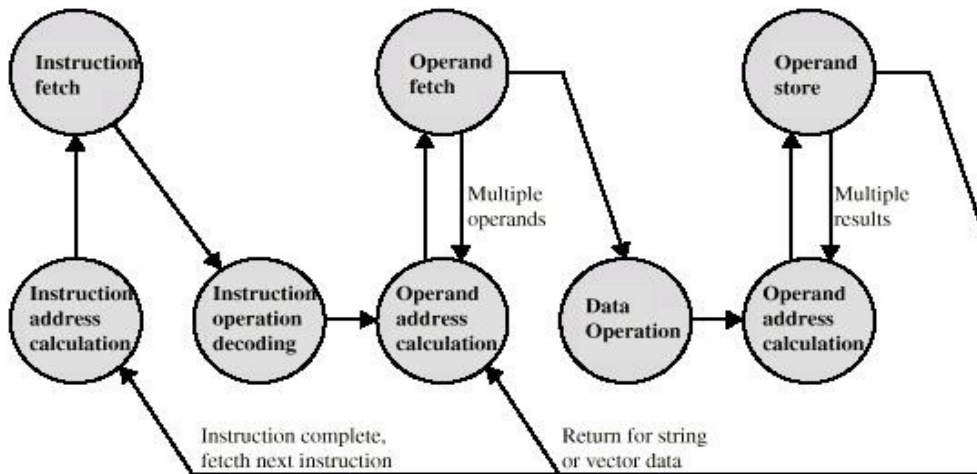


Fig: Instruction cycle state diagram

Interrupts:

- Mechanism by which other modules (e.g. I/O) may interrupt normal sequence of processing
- Program
 - e.g. overflow, division by zero
- Timer
 - Generated by internal processor timer
 - Used in pre-emptive multi-tasking
- I/O
 - from I/O controller
- Hardware failure
 - e.g. memory parity error
- Instruction Cycle
 - Added to instruction cycle
 - Processor checks for interrupt

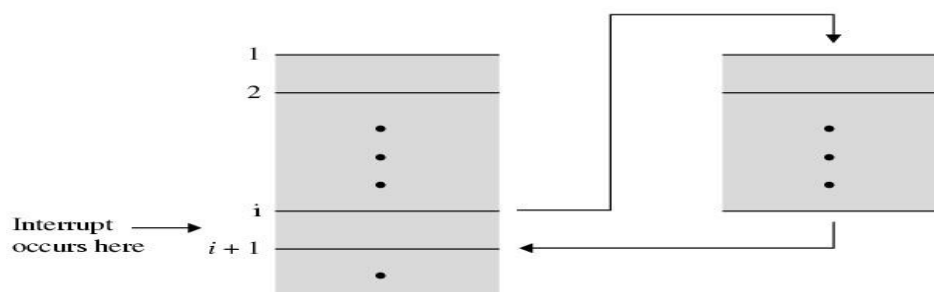
■ Indicated by an interrupt signal

- If no interrupt, fetch next instruction
- If interrupt pending:

- Suspend execution of current program
- Save context
- Set PC to start address of interrupt handler routine
- Process interrupt
- Restore context and continue interrupted program

User Program

Interrupt Handler



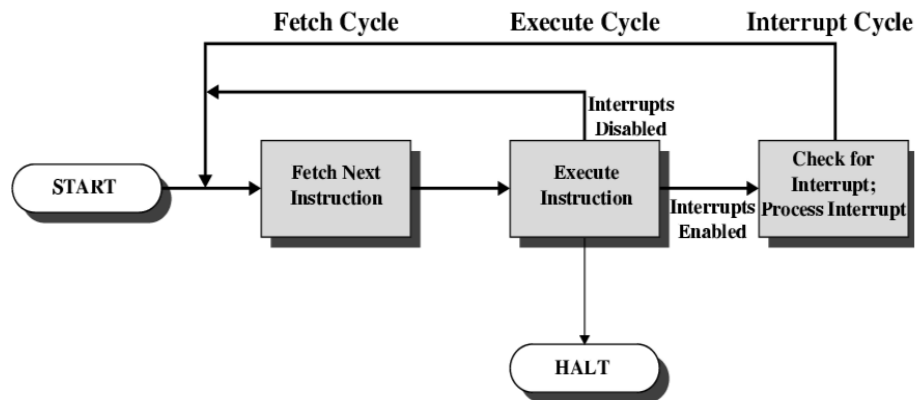


Fig: Instruction Cycle with Interrupts

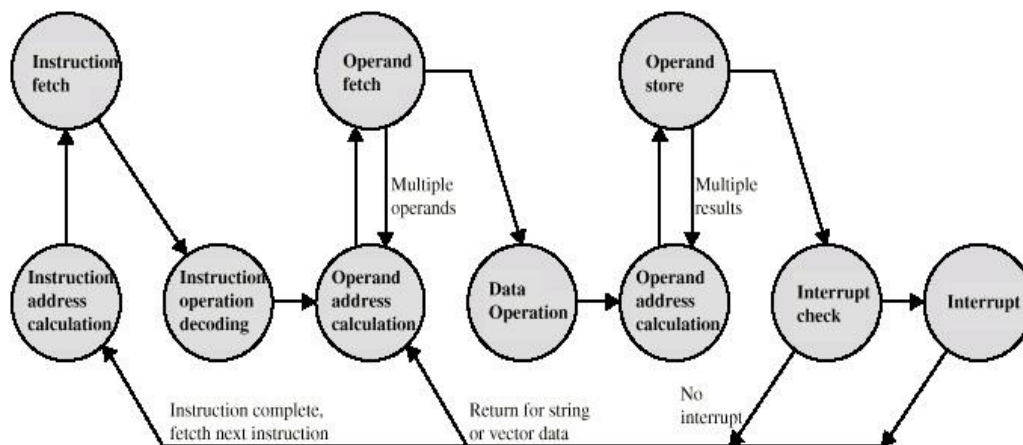


Fig: Instruction cycle state diagram, with interrupts

- Multiple Interrupts
 - Disable interrupts (approach #1)
 - Processor will ignore further interrupts whilst processing one interrupt
 - Interrupts remain pending and are checked after first interrupt has been processed
 - Interrupts handled in sequence as they occur
 - Define priorities (approach #2)
 - Low priority interrupts can be interrupted by higher priority interrupts
 - When higher priority interrupt has been processed, processor returns to previous interrupt

1.6 Interconnection structures

The collection of paths connecting the various modules is called the interconnecting structure.

- All the units must be connected
- Different type of connection for different type of unit
 - Memory
 - Input/Output
 - CPU

- Memory Connection
 - Receives and sends data
 - Receives addresses (of locations)
 - Receives control signals
 - Read
 - Write
 - Timing

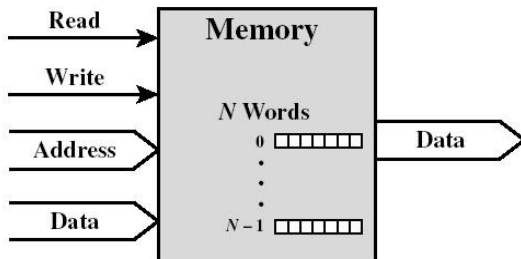


Fig: Memory Module

- I/O Connection
 - Similar to memory from computer's viewpoint
 - Output
 - Receive data from computer
 - Send data to peripheral
 - Input
 - Receive data from peripheral
 - Send data to computer
 - Receive control signals from computer
 - Send control signals to peripherals
 - e.g. spin disk
 - Receive addresses from computer
 - e.g. port number to identify peripheral
 - Send interrupt signals (control)

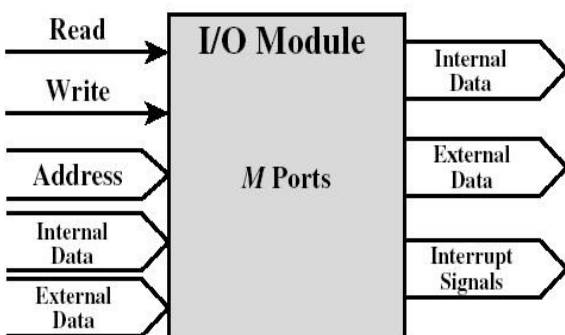


Fig: I/O Module

- CPU Connection
 - Reads instruction and data
 - Writes out data (after processing)
 - Sends control signals to other units
 - Receives (& acts on) interrupts

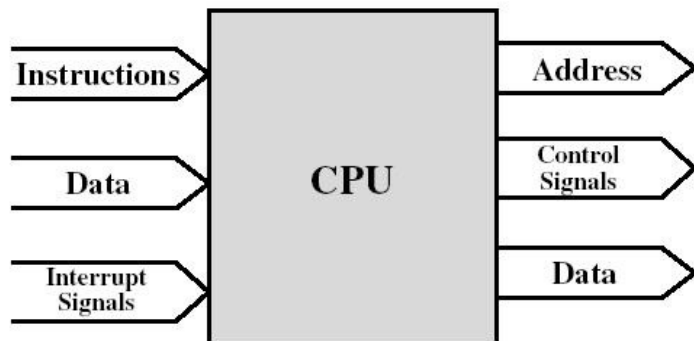


Fig: CPU Module

1.7 Bus interconnection

- A bus is a communication pathway connecting two or more devices
- Usually broadcast (all components see signal)
- Often grouped
 - A number of channels in one bus
 - e.g. 32 bit data bus is 32 separate single bit channels
- Power lines may not be shown
- There are a number of possible interconnection systems
- Single and multiple BUS structures are most common
- e.g. Control/Address/Data bus (PC)
- e.g. Unibus (DEC-PDP)
- Lots of devices on one bus leads to:
 - Propagation delays
 - Long data paths mean that co-ordination of bus use can adversely affect performance
 - If aggregate data transfer approaches bus capacity
- Most systems use multiple buses to overcome these problems

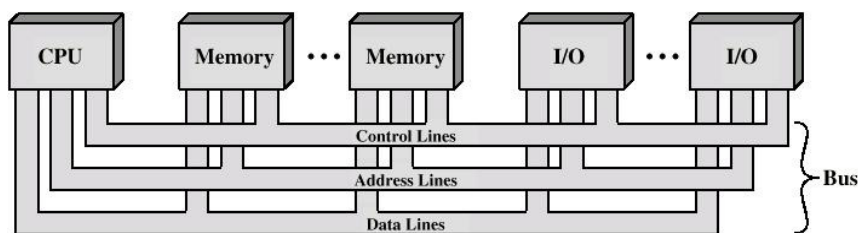


Fig: Bus Interconnection Scheme

- Data Bus
 - Carries data
 - Remember that there is no difference between “data” and “instruction” at this level
 - Width is a key determinant of performance
 - 8, 16, 32, 64 bit
- Address Bus
 - Identify the source or destination of data
 - e.g. CPU needs to read an instruction (data) from a given location in memory
 - Bus width determines maximum memory capacity of system
 - e.g. 8080 has 16 bit address bus giving 64k address space
- Control Bus
 - Control and timing information
 - Memory read
 - Memory write
 - I/O read
 - I/O write
 - Transfer ACK
 - Bus request
 - Bus grant
 - Interrupt request
 - Interrupt ACK
 - Clock
 - Reset

Central Processing Unit

The part of the computer that performs the bulk of data processing operations is called the Central Processing Unit (CPU) and is the central component of a digital computer. Its purpose is to interpret instruction cycles received from memory and perform arithmetic, logic and control operations with data stored in internal register, memory words and I/O interface units. A CPU is usually divided into two parts namely processor unit (Register Unit and Arithmetic Logic Unit) and control unit.

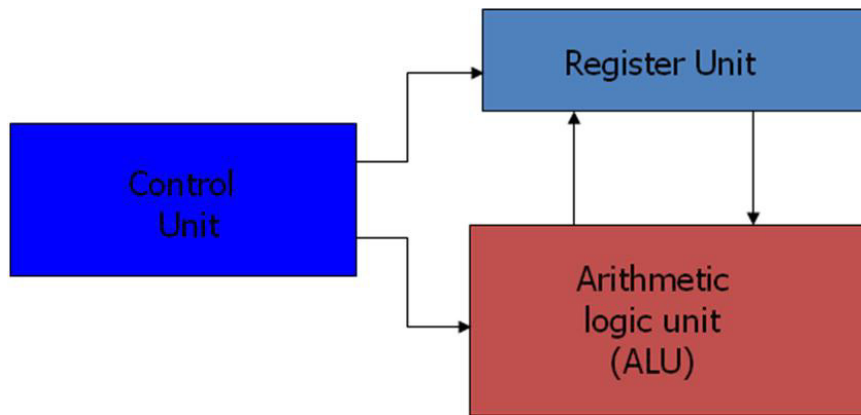


Fig: Components of CPU

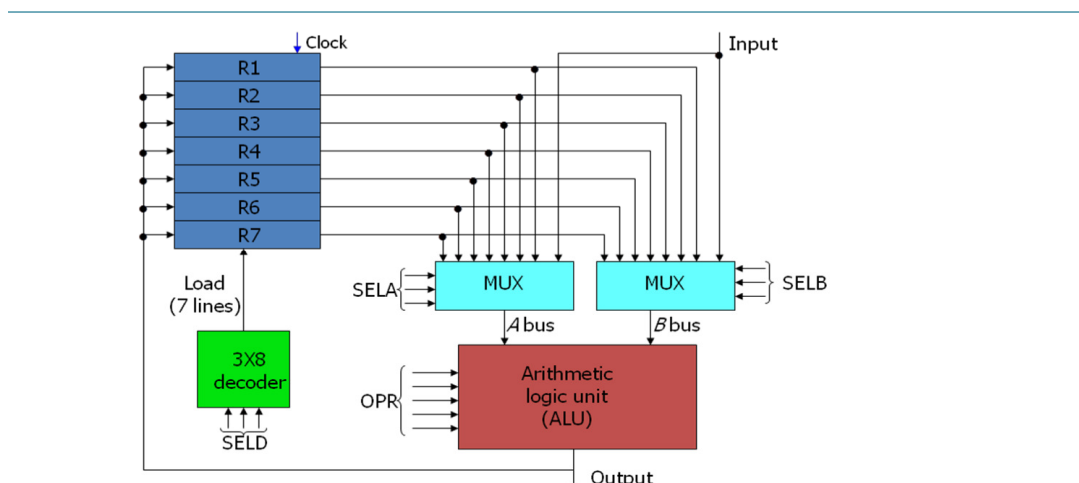
Processor Unit:

The processor unit consists of arithmetic unit, logic unit, a number of registers and internal buses that provides data path for transfer of information between register and arithmetic logic unit. The block diagram of processor unit is shown in figure below where all registers are connected through common buses. The registers communicate each other not only for direct data transfer but also while performing various micro-operations.

Here two sets of multiplexers select register which perform input data for ALU. A decoder selects destination register by enabling its load input. The function select in ALU determines the particular operation that to be performed.

For an example to perform the operation: $R_3 \leftarrow R_1 + R_2$

- ❑ MUX A selector (SELA): to place the content of R_1 into bus A.
- ❑ MUX B selector (SELB): to place the content of R_2 into bus B.
- ❑ ALU operation selector (OPR): to provide arithmetic addition $A + B$.
- ❑ Decoder destination selector (SELD): to transfer the content of the output bus into R_3 .



Control unit:

The control unit is the heart of CPU. It consists of a program counter, instruction register, timing and control logic. The control logic may be either hardwired or micro-programmed. If it is a hardwired, register decodes and a set of gates are connected to provide the logic that determines the action required to execute various instructions. A micro-programmed control unit uses a control memory to store micro instructions and a sequence to determine the order by which the instructions are read from control memory.

The control unit decides what the instructions mean and directs the necessary data to be moved from memory to ALU. Control unit must communicate with both ALU and main memory and coordinates all activities of processor unit, peripheral devices and storage devices. It can be characterized on the basis of design and implementation by:

- Defining basic elements of the processor

- Describing the micro-operation that processor performs

- Determining the function that the control unit must perform to cause the micro-operations to be performed.

Control unit must have inputs that allow determining the state of system and outputs that allow controlling the behavior of system.

The input to control unit are:

Flag: flags are headed to determine the status of processor and outcome of previous ALU operation.

Clock: All micro-operations are performed within each clock pulse. This clock pulse is also called as processor cycle time or clock cycle time.

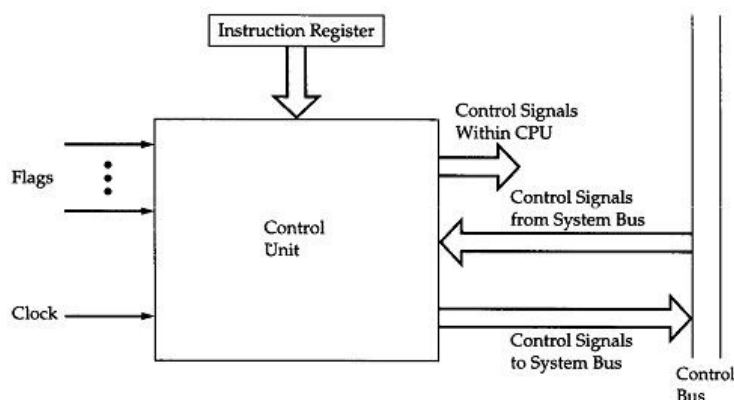
Instruction Register: The op-code of instruction determines which micro-operation to perform during execution cycle.

Control signal from control bus: The control bus portion of system bus provides interrupt, acknowledgement signals to control unit.

The outputs from control unit are:

Control signal within processor: These signals causes data transfer between registers, activate ALU functions.

Control signal to control bus: These are signals to memory and I/O module. All these control signals are applied directly as binary inputs to individual logic gate.



CPU Structure and Function

Processor Organization

Things a CPU must do:

- Fetch Instructions
- Interpret Instructions
- Fetch Data
- Process Data
- Write Data

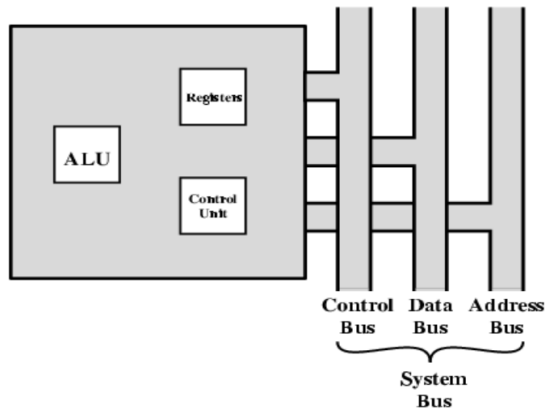


Fig: The CPU with the System Bus

- π A small amount of internal memory, called the registers, is needed by the CPU to fulfill these requirements

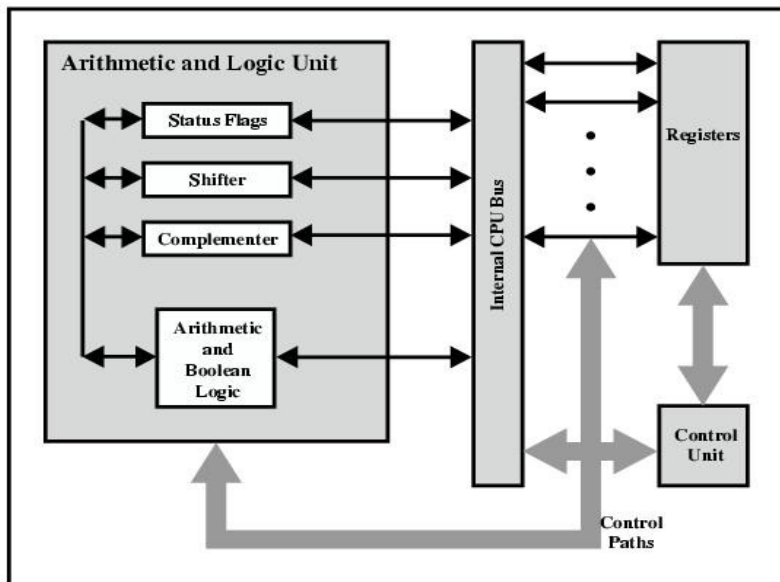


Fig: Internal Structure of the CPU

- π Components of the CPU

Arithmetic and Logic Unit (ALU): does the actual computation or processing of data

Control Unit (CU): controls the movement of data and instructions into and out of the CPU and controls the operation of the ALU.

Register Organization

- π Registers are at top of the memory hierarchy. They serve two functions:

User-Visible Registers - enable the machine- or assembly-language programmer to minimize main-memory references by optimizing use of registers

Control and Status Registers - used by the control unit to control the operation

User-Visible Registers

Categories of Use

- p General Purpose registers - for variety of functions
- q Data registers - hold data
- r Address registers - hold address information
- s Segment pointers - hold base address of the segment in use
- t Index registers - used for indexed addressing and may be auto indexed
- u Stack Pointer - a dedicated register that points to top of a stack. Push, pop, and other stack instructions need not contain an explicit stack operand.
- v Condition Codes (flags)

Design Issues

Completely general-purpose registers or specialized use?

Specialized registers save bits in instructions because their use can be implicit

General-purpose registers are more flexible

Trend is toward use of specialized registers

Number of registers provided?

More registers require more operand specifier bits in instructions

8 to 32 registers appears optimum (RISC systems use hundreds, but are a completely different approach)

Register Length?

Address registers must be long enough to hold the largest address

Data registers should be able to hold values of most data types

Some machines allow two contiguous registers for double-length values

Automatic or manual save of condition codes?

Condition restore is usually automatic upon call return

Saving condition code registers may be automatic upon call instruction, or may be manual

Control and Status Registers

Essential to instruction execution

- o Program Counter (PC)
- o Instruction Register (IR)
- o Memory Address Register (MAR) - usually connected directly to address lines of bus
- o Memory Buffer Register (MBR) - usually connected directly to data lines of bus

Program Status Word (PSW) - also essential, common fields or flags contained include:

- Sign - sign bit of last arithmetic operation
- Zero - set when result of last arithmetic operation is 0
- Carry - set if last op resulted in a carry into or borrow out of a high-order bit
- Equal - set if a logical compare result is equality
- Overflow - set when last arithmetic operation caused overflow
- Interrupt Enable/Disable - used to enable or disable interrupts
- Supervisor - indicates if privileged ops can be used

Other optional registers

- o Pointer to a block of memory containing additional status info (like process control blocks)

- p An interrupt vector
- q A system stack pointer
- r A page table pointer
- s I/O registers

Design issues

- o Operating system support in CPU
- How to divide allocation of control information between CPU registers and first part of main memory (usual tradeoffs apply)

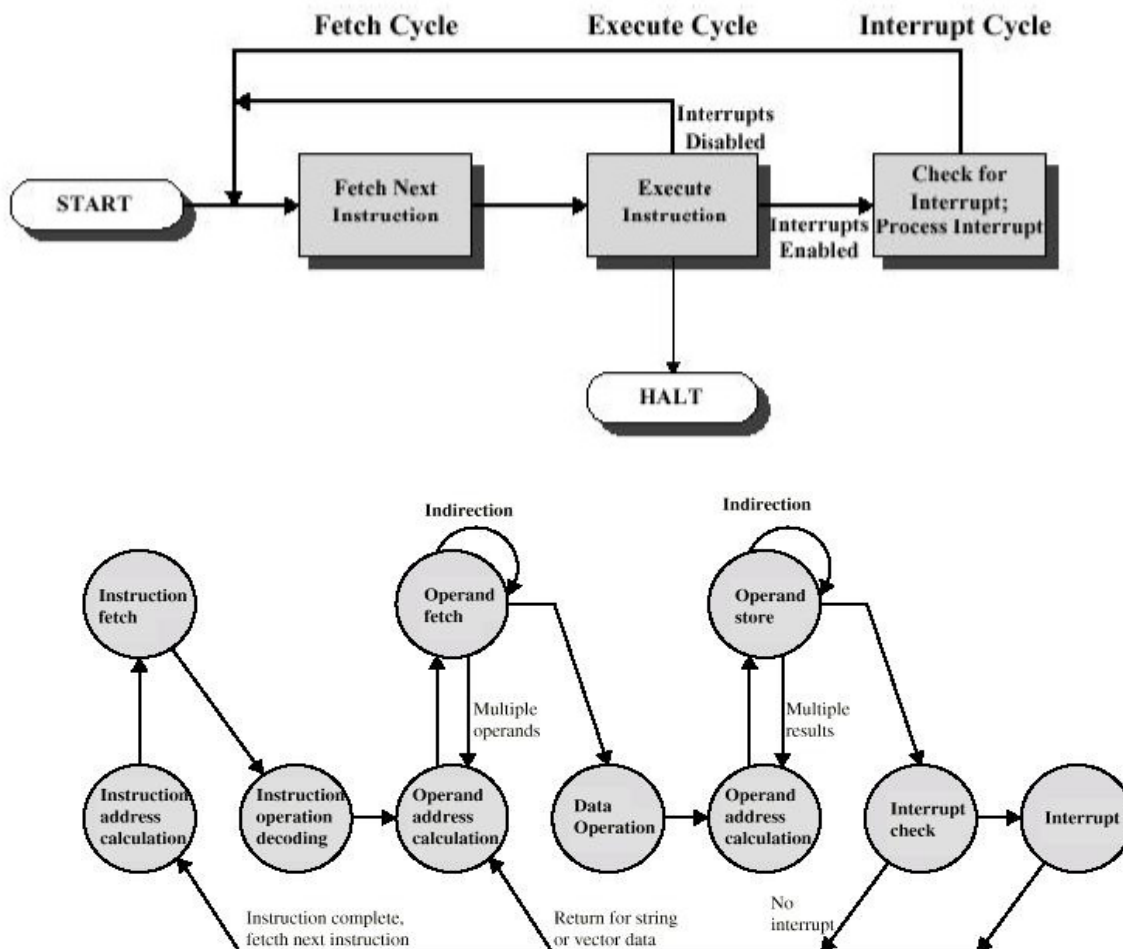
The Instruction Cycle

Basic instruction cycle contains the following sub-cycles.

Fetch - read next instruction from memory into CPU

Execute - Interpret the opcode and perform the indicated operation

Interrupt - if interrupts are enabled and one has occurred, save the current process state and service the interrupt



The Indirect Cycle

- Think of as another instruction sub-cycle
- May require just another fetch (based upon last fetch)
- Might also require arithmetic, like indexing

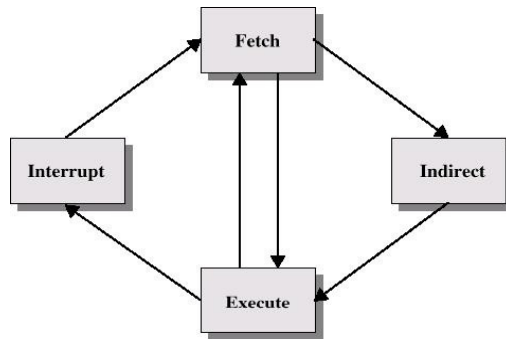
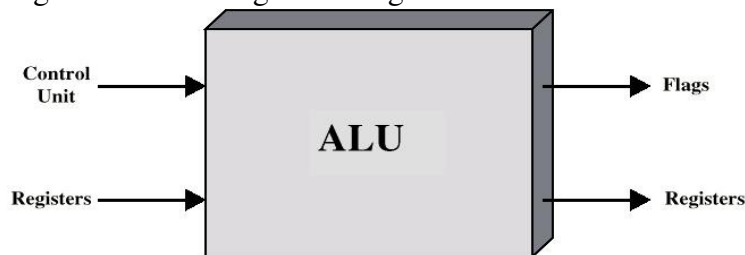


Fig: Instruction Cycle with Indirect

Arithmetic and Logic Unit

ALU is the combinational circuit of that part of computer that actually performs arithmetic and logical operations on data. All of the other elements of computer system- control unit, registers, memory, I/O are their mainly to bring data into the ALU for it to process and then to take the result back out. An ALU & indeed all electronic components in computer are based on the use of simple digital logic device that can store binary digit and perform simple Boolean logic function. Figure indicates in general in general term how ALU is interconnected with rest of the processor.



Data are presented to ALU in register and the result of operation is stored in register. These registers are temporarily storage location within the processor that are connected by signal path to the ALU. The ALU may also set flags as the result of an operation. The flags values are also stored in registers within the processor. The control unit provides signals that control the operation of ALU and the movement of data into an out of ALU.

The design of ALU has three stages.

o Design the arithmetic section

The basic component of arithmetic circuit is a parallel adder which is constructed with a number of full adder circuits connected in cascade. By controlling the data inputs to the parallel adder, it is possible to obtain different types of arithmetic operations. Below figure shows the arithmetic circuit and its functional table.

Fig: Block diagram of Arithmetic Unit

Functional table for arithmetic unit:

Select		Input Y	Output		Microoperation	
S ₁	S ₀		Cin = 0	Cin = 1	Cin = 0	Cin = 1

0	0	0	A	A+1	Transfer A	Increment A
0	1	B	A+B	A+B+1	Addition	Addition with carry
1	0	B'	A+B'	A+B'+1	Subtraction with borrow	Subtraction
1	1	-1	A-1	A	Decrement A	Transfer A

- **Design the logical section**

The basic components of logical circuit are AND, OR, XOR and NOT gate circuits connected accordingly. Below figure shows a circuit that generates four basic logic micro-operations. It consists of four gates and a multiplexer. Each of four logic operations is generated through a gate that performs the required logic. The two selection input S1 and S0 choose one of the data inputs of the multiplexer and directs its value to the output. Functional table lists the logic operations.

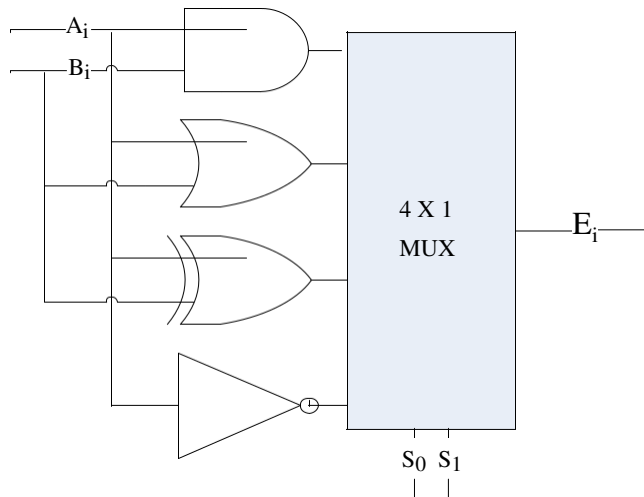


Fig: Block diagram of Logic Unit

Functional table for logic unit:

S1	S0	output	Microoperation
0	0	$A_i \& B_i$	AND
0	1	$A_i \parallel B_i$	OR
1	0	$A_i \text{ XOR } B_i$	XOR
1	1	A_i'	NOT

- **Combine these 2 sections to form the ALU**

Below figure shows a combined circuit of ALU where n data input from A are combined with n data input from B to generate the result of an operation at the G output line. ALU has a number of selection lines used to determine the operation to be performed. The selection lines are decoded with the ALU so that selection lines can specify distinct operations. The mode select S_2 differentiate between arithmetic and logical operations. The two functions select S_1 and S_0 specify the particular arithmetic and logic operations to be performed. With three selection lines, it is

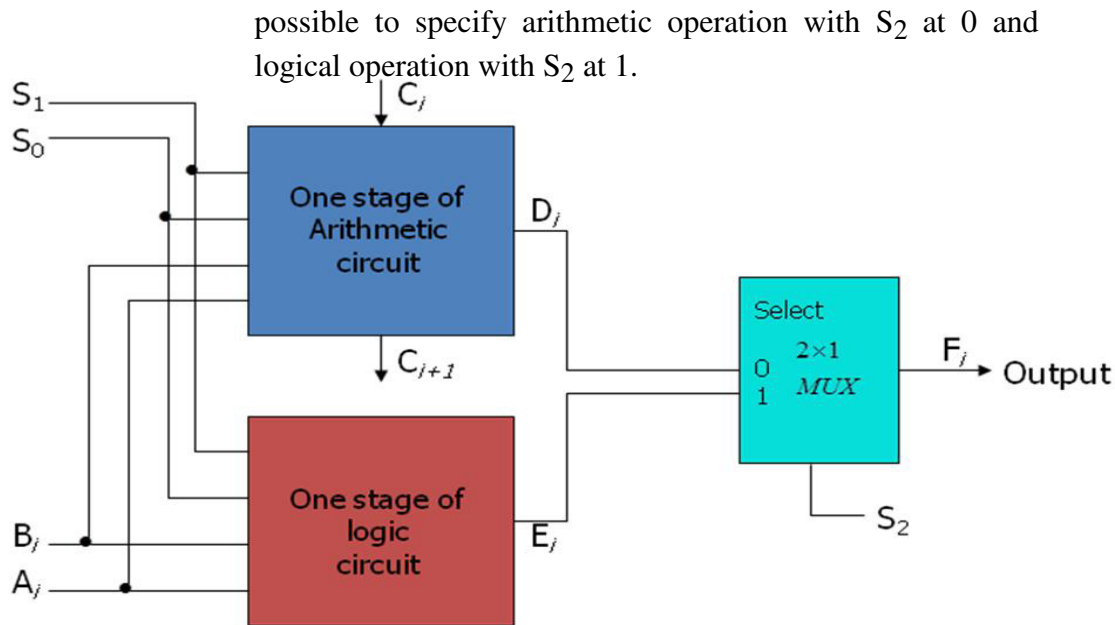


Fig: Block diagram of ALU

Instruction Formats

The computer can be used to perform a specific task, only by specifying the necessary steps to complete the task. The collection of such ordered steps forms a 'program' of a computer. These ordered steps are the instructions. Computer instructions are stored in central memory locations and are executed sequentially one at a time. The control reads an instruction from a specific address in memory and executes it. It then continues by reading the next instruction in sequence and executes it until the completion of the program.

A computer usually has a variety of Instruction Code Formats. It is the function of the control unit within the CPU to interpret each instruction code and provide the necessary control functions needed to process the instruction. An n bit instruction that k bits in the address field and m bits in the operation code field come addressed 2^k location directly and specify 2^m different operation.

- π The bits of the instruction are divided into groups called fields.
- θ The most common fields in instruction formats are:
 - An **Operation code** field that specifies the operation to be performed.
 - An **Address field** that designates a memory address or a processor register.
- p A **Mode field** that specifies the way the operand or the effective address is determined.

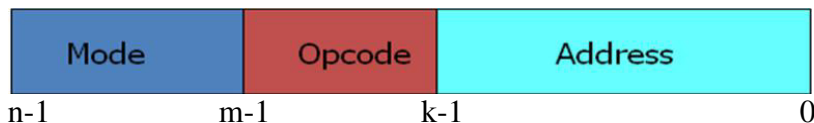


Fig: Instruction format with mode field

The operation code field (Opcode) of an instruction is a group of bits that define various processor operations such as add, subtract, complement, shift etcetera. The bits that define the mode field of an instruction code specify a variety of alternatives for choosing the operands from the given address. Operation specified by an instruction is executed on some data stored in the processor register or in the memory location. Operands residing in memory are specified by their memory address. Operands residing in processor register are specified with a register address.

Types of Instruction

- π Computers may have instructions of several different lengths containing varying number of addresses.
- θ The number of address fields in the instruction format of a computer depends on the internal organization of its registers.
- ρ Most computers fall into one of 3 types of CPU organizations:

Single accumulator organization:- All the operations are performed with an accumulator register. The instruction format in this type of computer uses one address field. For example: ADD X, where X is the address of the operands .

General register organization:- The instruction format in this type of computer needs three register address fields. For example: ADD R1,R2,R3

Stack organization:- The instruction in a stack computer consists of an operation code with no address field. This operation has the effect of popping the 2 top numbers from the stack, operating the numbers and pushing the sum into the stack. For example: ADD

Computers may have instructions of several different lengths containing varying number of addresses. Following are the types of instructions.

1) Three address Instruction

With this type of instruction, each instruction specifies two operand location and a result location. A temporary location T is used to store some intermediate result so as not to alter any of the operand location. The three address instruction format requires a very complex design to hold the three address references.

Format: Op X, Y, Z; X ← Y Op Z

Example: ADD X, Y, Z; X ← Y + Z

ADVANTAGE: It results in short programs when evaluating arithmetic expressions.

DISADVANTAGE: The instructions requires too many bits to specify 3 addresses.

• Two address instruction

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. One address must do double duty as both operand and result. The two address instruction format reduces the space requirement. To avoid altering the value of an operand, a MOV instruction is used to move one of the values to a result or temporary location T, before performing the operation.

Format: Op X, Y; X ← X Op Y

Example: SUB X, Y; X ← X - Y

2) One address Instruction

It was generally used in earlier machine with the implied address been a CPU register known as accumulator. The accumulator contains one of the operand and is used to store the result. One-address instruction uses an implied accumulator (Ac) register for all data manipulation. All operations are done between the AC register and a memory operand. We use LOAD and STORE instruction for transfer to and from memory and Ac register.

Format: Op X; Ac ← Ac Op X

Example: MUL X; Ac \leftarrow Ac * X

3) Zero address Instruction

It does not use address field for the instruction like ADD, SUB, MUL, DIV etc. The PUSH and POP instructions, however, need an address field to specify the operand that communicates with the stack. The name “Zero” address is given because of the absence of an address field in the computational instruction.

Format: Op; TOS \leftarrow TOS Op (TOS – 1)

Example: DIV; TOS \leftarrow TOS DIV (TOS – 1)

Example: To illustrate the influence of the number of address on computer programs, we will evaluate the arithmetic statement $X=(A+B)*(C+D)$ using Zero, one, two, or three address instructions.

1. Three-Address Instructions:

ADD R1, A, B; $R1 \leftarrow M[A] + M[B]$
 ADD R2, C, D; $R2 \leftarrow M[C] + M[D]$
 MUL X, R1, R2; $M[X] \leftarrow R1 * R2$

It is assumed that the computer has two processor registers R1 and R2. The symbol M[A] denotes the operand at memory address symbolized by A.

2. Two-Address Instructions:

MOV R1, A; $R1 \leftarrow M[A]$
 ADD R1, B; $R1 \leftarrow R1 + M[B]$

MOV R2, C; $R2 \leftarrow M[C]$
 ADD R2, D; $R2 \leftarrow R2 + M[D]$
 MUL R1, R2; $R1 \leftarrow R1 * R2$
 MOV X, R1; $M[X] \leftarrow R1$

3. One-Address Instruction:

LOAD A; $Ac \leftarrow M[A]$
 ADD B; $Ac \leftarrow Ac + M[B]$
 STORE T; $M[T] \leftarrow Ac$
 LOAD C; $Ac \leftarrow M[C]$
 ADD D; $Ac \leftarrow Ac + M[D]$
 MUL T; $Ac \leftarrow Ac * M[T]$
 STORE X; $M[X] \leftarrow Ac$

Here, T is the temporary memory location required for storing the intermediate result.

4. Zero-Address Instructions:

PUSH A; $TOS \leftarrow A$
 PUSH B; $TOS \leftarrow B$
 ADD; $TOS \leftarrow (A+B)$
 PUSH C; $TOS \leftarrow C$
 PUSH D; $TOS \leftarrow D$
 ADD; $TOS \leftarrow (C+D)$
 MUL; $TOS \leftarrow (C+D)*(A+B)$
 POPX; $M[X] \leftarrow TOS$

2.4 Addressing Modes

- ♣ 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 the following purposes:-
 - To give programming versatility to the user by providing such facilities as pointers to memory, counters for loop control, indexing of data and various other purposes.
 - To reduce the number of bits in the addressing field of the instructions.
 - Other computers use a single binary for operation & Address mode.
 - The mode field is used to locate the operand.
 - Address field may designate a memory address or a processor register.
 - There are 2 modes that need no address field at all (Implied & immediate modes).

Effective address (EA):

- ♣ The effective address is defined to be the memory address obtained from the computation dictated by the given addressing mode.

The effective address is the address of the operand in a computational-type instruction

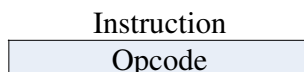
The most well known addressing mode are:

Implied Addressing Mode.
 Immediate Addressing Mode
 Register Addressing Mode
 Register Indirect Addressing Mode
 Auto-increment or Auto-decrement Addressing Mode
 Direct Addressing Mode
 Indirect Addressing Mode
 Displacement Address Addressing Mode
 Relative Addressing Mode
 Index Addressing Mode
 Stack Addressing Mode

Implied Addressing Mode:

In this mode the operands are specified implicitly in the definition of the instruction.

For example:- CMA - “complement accumulator” is an implied-mode instruction because the operand in the accumulator register is implied in the definition of the instruction. In fact, all register reference instructions that use an accumulator are implied-mode instructions.



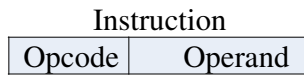
Advantage: no memory reference. Disadvantage: limited operand

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.

This instruction has an operand field rather than an address field. The operand field contains the actual operand to be used in conjunction with the operation specified in the instruction.

These instructions are useful for initializing register to a constant value;
For example MVI B, 50H

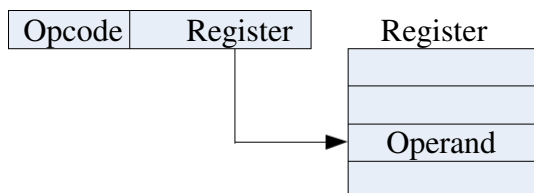


It was mentioned previously that the address field of an instruction may specify either a memory word or a processor register. When the address field specifies a processor register, the instruction is said to be in register-mode.

Advantage: no memory reference. Disadvantage: limited operand

Register direct addressing mode:

In this mode, the operands are in registers that reside within the CPU.
The particular register is selected from the register field in the instruction. For example MOV A, B



Effective Address (EA) = R

Advantage: no memory reference. Disadvantage: limited address space

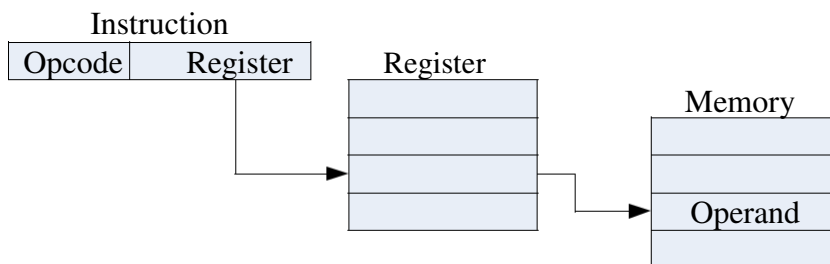
Register indirect addressing mode:

In this mode the instruction specifies a register in the CPU whose contents give the address of the operand in the memory.

In other words, the selected register contains the address of the operand rather than the operand itself.

Before using a register indirect mode instruction, the programmer must ensure that the memory address of the operand is placed in the processor register with a previous instruction.

For example LDAX B



Effective Address (EA) = (R)

Advantage: Large address space.

The address field of the instruction uses fewer bits to select a register than would have been required to specify a memory address directly.

Disadvantage: Extra memory reference

Auto increment or Auto decrement Addressing Mode:

This is similar to 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 registers refers to a table of data in memory, it is necessary to increment or decrement the registers after every access to the table.

This can be achieved by using the increment or decrement instruction. In some computers it is automatically accessed.

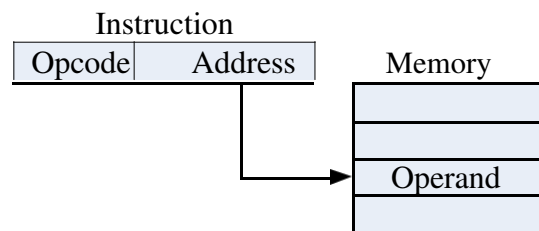
The address field of an instruction is used by the control unit in the CPU to obtain the operands from memory.

Sometimes the value given in the address field is the address of the operand, but sometimes it is the address from which the address has to be calculated.

Direct Addressing Mode

- π 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.

For example LDA 4000H



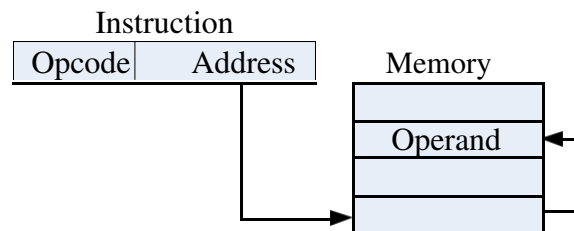
Effective Address (EA) = A

Advantage: Simple.

Disadvantage: limited address field

Indirect Addressing Mode

- π In this mode the address field of the instruction gives the address where the effective address is stored in memory.
- θ Control unit fetches the instruction from the memory and uses its address part to access memory again to read the effective address.



Effective Address (EA) = (A)

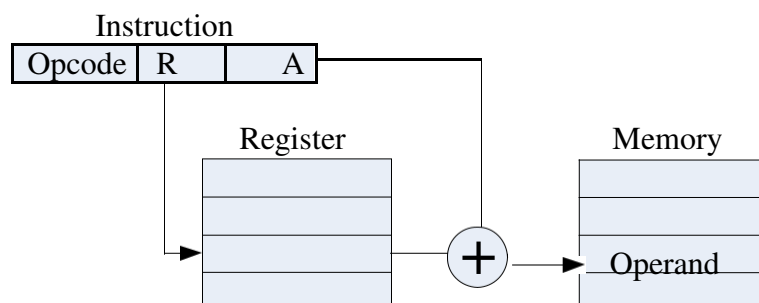
Advantage: Flexibility.

Disadvantage: Complexity

Displacement Addressing Mode

A very powerful mode of addressing combines the capabilities of direct addressing and register indirect addressing.

The address field of instruction is added to the content of specific register in the CPU.



Effective Address (EA) = A + (R) Advantage: Flexibility.

Disadvantage: Complexity

Relative Addressing Mode

- π In this mode the content of the program counter (PC) is added to the address part of the instruction in order to obtain the effective address.
- θ The address part of the instruction is usually a signed number (either a +ve or a -ve number).
- ρ When the 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.
Effective Address (EA) = PC + A

Indexed Addressing Mode

- π In this mode the content of an index register (XR) is added to the address part of the instruction to obtain the effective address.
- θ The index register is a special CPU register that contains an index value.
- ρ Note: If an index-type instruction does not include an address field in its format, the instruction is automatically converted to the register indirect mode of operation. Effective Address (EA) = XR + A

Base Register Addressing Mode

In this mode the content of a base register (BR) is added to the address part of the instruction to obtain the effective address.

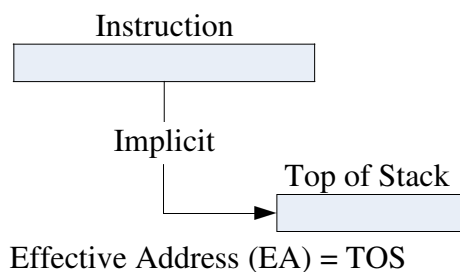
This is similar to the indexed addressing mode except that the register is now called a base register instead of the index register.

The base register addressing mode is used in computers to facilitate the relocation of programs in memory i.e. when programs and data are moved from one segment of memory to another.

Effective Address (EA) = BR + A

Stack Addressing Mode

The stack is the linear array of locations. It is some times referred to as push down list or last in First out (LIFO) queue. The stack pointer is maintained in register.



RISC and CISC

- π Important aspect of computer – design of the instruction set for processor.
- θ Instruction set – determines the way that machine language programs are constructed.
- ρ Early computers – simple and small instruction set, need to minimize the hardware used.
- σ Advent of IC – cheaper digital software, instructions intended to increase both in number of complexity.

- τ Many computers – more than 100 or 200 instructions, variety of data types and large number of addressing modes.

Complex Instruction Set Computers (CISC)

- π The trend into computer hardware complexity was influenced by various factors:
 - Upgrading existing models to provide more customer applications
 - Adding instructions that facilitate the translation from high-level language into machine language programs
 - Striving to develop machines that move functions from software implementation into hardware implementation
- θ A computer with a large number of instructions is classified as a *complex instruction set computer* (CISC).

One reason for the trend to provide a complex instruction set is the desire to simplify the compilation and improve the overall computer performance.

- π The essential goal of CISC architecture is to attempt to provide a single machine instruction for each statement that is written in a high-level language.
- θ Examples of CISC architecture are the DEC VAX computer and the IBM 370 computer. Other are 8085, 8086, 80x86 etc.

The major characteristics of CISC architecture

- ρ A large number of instructions—typically from 100 to 250 instructions
- σ Some instructions that perform specialized tasks and are used infrequently
- τ A large variety of addressing modes—typically from 5 to 20 different modes
- υ Variable-length instruction formats
- ϖ Instructions that manipulate operands in memory
- ω Reduced speed due to memory read/write operations
- ξ Use of microprogram – special program in control memory of a computer to perform the timing and sequencing of the microoperations – fetch, decode, execute etc.
- ψ Major complexity in the design of microprogram
- ζ No large number of registers – single register set of general purpose and low cost

Reduced Instruction Set Computers (RISC)

A computer uses fewer instructions with simple constructs so they can be executed much faster within the CPU without having to use memory as often. It is classified as a *reduced instruction set computer* (RISC).

- π RISC concept – an attempt to reduce the execution cycle by simplifying the instruction set
- θ Small set of instructions – mostly register to register operations and simple load/store operations for memory access
- ρ Each operand – brought into register using load instruction, computations are done among data in registers and results transferred to memory using store instruction
- σ Simplify instruction set and encourages the optimization of register manipulation
- τ May include immediate operands, relative mode etc.

The major characteristics of RISC architecture

Relatively few instructions
 Relatively few addressing modes
 Memory access limited to load and store instructions

All operations done within the registers of the CPU
 Fixed-length, easily decoded instruction format
 Single-cycle instruction execution
 Hardwired rather than microprogrammed control

Other characteristics attributed to RISC architecture

- π A relatively large number of registers in the processor unit
- θ Use of overlapped register windows to speed-up procedure call and return
- ρ Efficient instruction pipeline – fetch, decode and execute overlap
- σ Compiler support for efficient translation of high-level language programs into machine language programs
- τ Studies that show improved performance for RISC architecture do not differentiate between the effects of the reduced instruction set and the effects of a large register file.
- υ A large number of registers in the processing unit are sometimes associated with RISC processors.
- ϖ RISC processors often achieve 2 to 4 times the performance of CISC processors.
- ω RISC uses much less chip space; extra functions like memory management unit or floating point arithmetic unit can also be placed on same chip. Smaller chips allow a semiconductor mfg. to place more parts on a single silicon wafer, which can lower the per chip cost dramatically.
- ξ RISC processors are simpler than corresponding CISC processors, they can be designed more quickly.

Comparison between RISC and CISC Architectures

S.N.	RISC	CISC
1	Simple instructions taking one cycle	Complex instructions taking multiple cycles
2	Only load and store memory references	Any instructions may reference memory
3	Heavily pipelined	Not/less pipelined
4	Multiple register sets	Single register set
5	Complexity is in compiler	Complexity is in micro-programming
6	Instructions executed by hardware	Instructions interpreted by micro-programming
7	Fixed format instructions	Variable format instructions
8	Few instructions and modes	Large instructions and modes

Computer Arithmetic

Integer Representation: (Fixed-point representation):

An eight bit word can be represented the numbers from zero to 255 including

00000000 = 0

00000001 = 1

11111111 = 255

In general if an n-bit sequence of binary digits $a_{n-1}, a_{n-2}, \dots, a_1, a_0$; is interpreted as unsigned integer A.

$$A = \sum_{i=0}^{n-1} 2^i a_i$$

Sign magnitude representation:

There are several alternative convention used to represent negative as well as positive integers, all of which involves treating the most significant (left most) bit in the word as sign bit. If the sign bit is 0, the number is +ve and if the sign bit is 1, the number is -ve. In n bit word the right most n-1 bit hold the magnitude of integer.

For an example,

+18 = 00010010

- 18 = 10010010 (sign magnitude)

The general case can be expressed as follows:

$$A = \sum_{i=0}^{n-2} 2^i a_i \quad \text{if } a_{n-1} = 0$$
$$= - \sum_{i=0}^{n-2} 2^i a_i \quad \text{if } a_{n-1} = 1$$

$$A = -2^{n-1} a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i \quad (\text{Both for +ve and -ve})$$

There are several drawbacks to sign-magnitude representation. One is that addition or subtraction requires consideration of both signs of number and their relative magnitude to carry out the required operation. Another drawback is that there are two representation of zero. For an example:

+0₁₀ = 00000000

-0₁₀ = 10000000 this is inconvenient.

2's complement representation:

Like sign magnitude representation, 2's complement representation uses the most significant bit as sign bit making it easy to test whether the integer is negative or positive. It differs from the use of sign magnitude representation in the way that the other bits are interpreted. For negation, take the Boolean complement (1's complement) of each bit of corresponding positive number, and then add one to the resulting bit pattern viewed as unsigned integer. Consider n bit integer A in 2's complement representation. If A is +ve then the sign bit a_{n-1} is zero. The remaining bits represent the magnitude of the number.

$$A = \sum_{i=0}^{n-2} 2^i a_i \quad \text{for } A \geq 0$$

The number zero is identified as +ve and therefore has zero sign bit and magnitude of all 0's. We can see that the range of +ve integer

that may be represented is from 0 (all the magnitude bits are zero) through $2^{n-1} - 1$ (all of the magnitude bits are 1).

Now for -ve number integer A, the sign bit a_{n-1} is 1. The range of -ve integer that can be represented is from -1 to -2^{n-1} .

$$2\text{'s complement, } A = -2^{n-1} a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

Defines the twos complement of representation of both positive and negative number.

For an example:

$$+18 = 00010010$$

$$1\text{'s complement} = 11101101$$

$$2\text{'s complement} = 11101110 = -18$$

5.1 Addition Algorithm

5.2 Subtraction Algorithm

$$1001 = -7$$

$$\underline{0101 = +5}$$

$$1110 = -2$$

$$(a) (-7) + (+5)$$

$$1100 = -4$$

$$\underline{0100 = +4}$$

$$10000 = 0$$

$$(b) (-4) + (4)$$

$$0011 = 3$$

$$\underline{0100 = 4}$$

$$0111 = 7$$

$$(c) (+3) + (+4)$$

$$1100 = -4$$

$$\underline{1111 = -1}$$

$$11011 = -5$$

$$(d) (-4) + (-1)$$

$$0101 = 5$$

$$\underline{0100 = 4}$$

$$1001 = \text{overflow}$$

$$(e) (+5) + (+4)$$

$$1001 = -7$$

$$\underline{1010 = -6}$$

$$10011 = \text{overflow}$$

$$(f) (-7) + (-6)$$

The first four examples illustrate successful operation if the result of the operation is +ve then we get +ve number in ordinary binary notation. If the result of the operation is -ve we get negative number in twos complement form. Note that in some instances there is carry bit beyond the end of what which is ignored. On any addition the result may larger then can be held in word size being use. This condition is called over flow. When overflow occur ALU must signal this fact so that no attempt is made to use the result. To detect overflow the following rule observed. If two numbers are added, and they are both +ve or both -ve; then overflow occurs if and only if the result has the opposite sign.

The data path and hardware elements needed to accomplish addition and subtraction is shown in figure below. The central element is binary adder, which is presented two numbers for addition and produces a sum and an overflow indication. The binary adder treats the two numbers as unsigned integers. For addition, the two numbers are presented to the adder from two registers A and B. The result may be stored in one of these registers or in the third. The overflow indication is stored in a 1-bit overflow flag V (where 1 = overflow and 0 = no overflow). For subtraction, the subtrahend (B register) is passed through a 2's complement unit so that its 2's complement is presented to the adder ($a - b = a + (-b)$).

Multiplication Algorithm

The multiplier and multiplicand bits are loaded into two registers Q and M. A third register A is initially set to zero. C is the 1-bit register which holds the carry bit resulting from addition. Now, the control logic reads the bits of the multiplier one at a time. If Q_0 is 1, the multiplicand is added to the register A and is stored back in register A with C bit used for carry. Then all the bits of CAQ are shifted to the right 1 bit so that C bit goes to A_{n-1} , A_0 goes to Q_{n-1} and Q_0 is lost. If Q_0 is 0, no addition is performed just do the shift. The process is repeated for each bit of the original multiplier. The resulting $2n$ bit product is contained in the QA register.

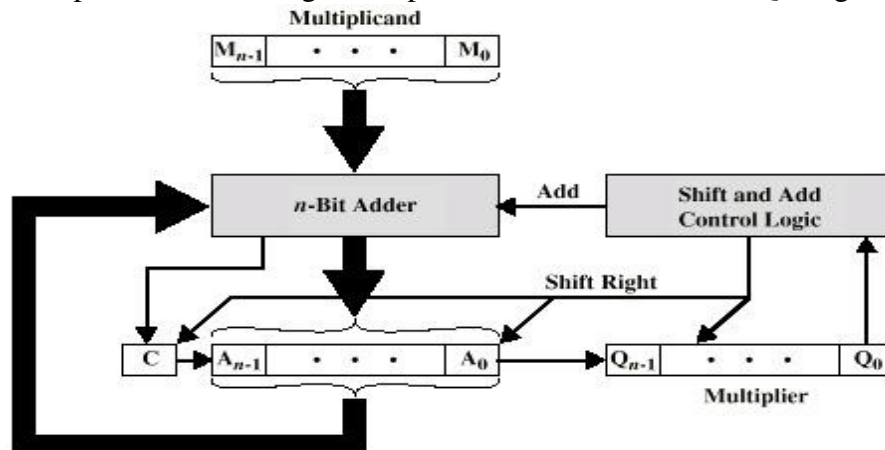


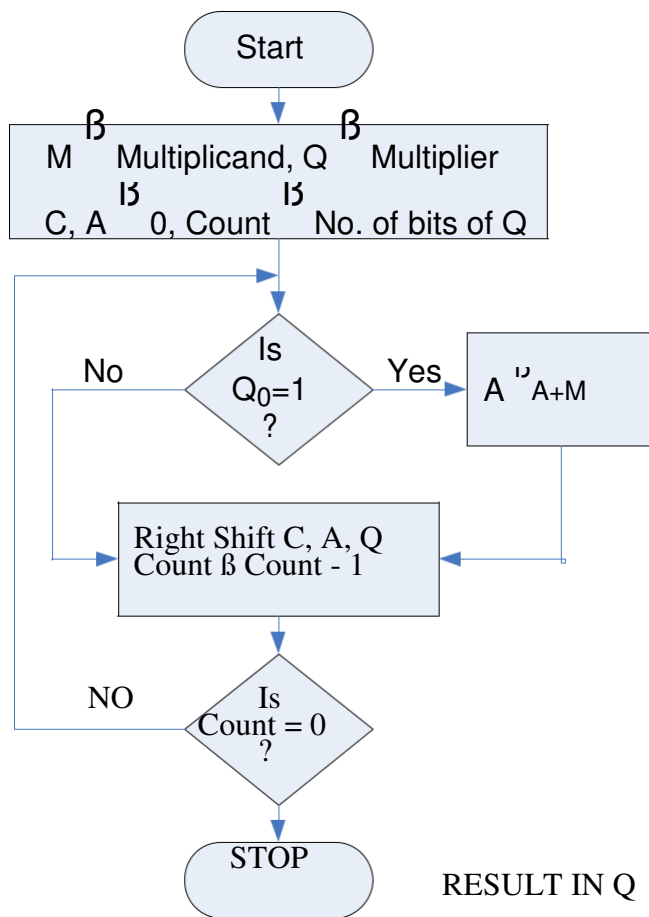
Fig: Block diagram of multiplication

There are three types of operation for multiplication.

- It should be determined whether a multiplier bit is 1 or 0 so that it can designate the partial product. If the multiplier bit is 0, the partial product is zero; if the multiplier bit is 1, the multiplicand is partial product.
- It should shift partial product.
- It should add partial product.

Unsigned Binary Multiplication

1011	Multiplicand 11
x 1101	Multiplier 13
1011	Partial Product
0000	
1011	
• 1011	
10001111	Product (143)

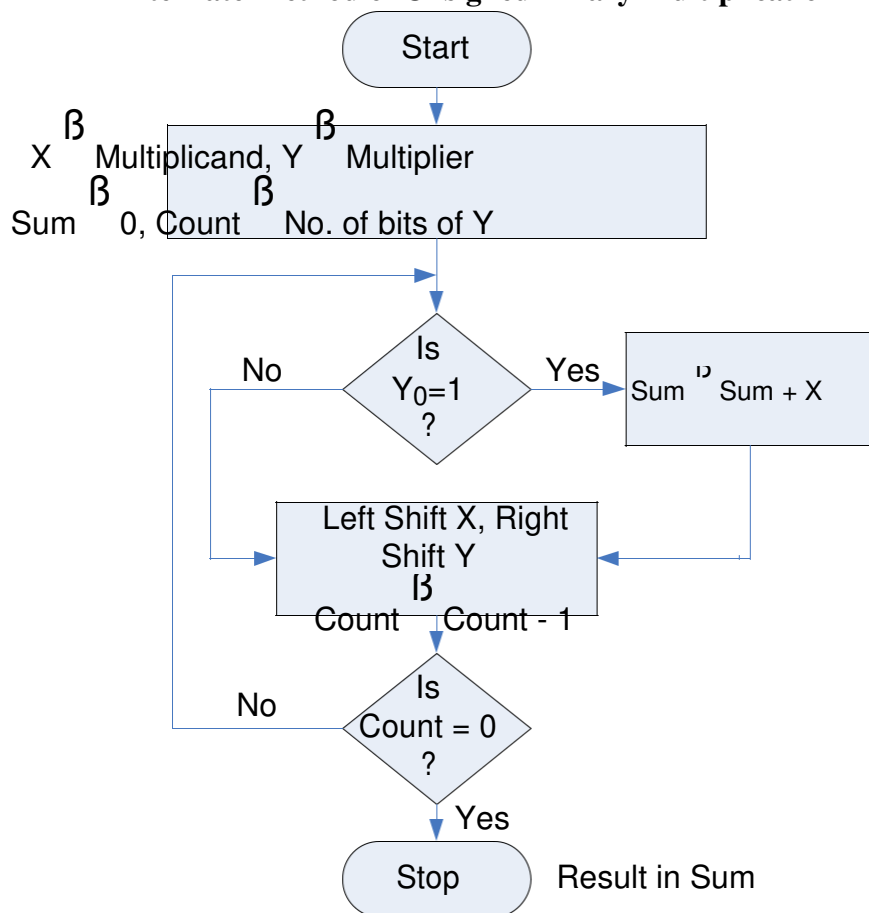


Example: Multiply 15 x 11 using unsigned binary method

C	A	Q	M	Count	Remarks
0	0000	1011	1111	4	Initialization
0	1111	1011	-	-	Add ($A \oplus A + M$)
0	0111	1101	-	3	Logical Right Shift C, A, Q
1	0110	1101	-	-	Add ($A \oplus A + M$)
0	1011	0110	-	2	Logical Right Shift C, A, Q
0	0101	1011	-	1	Logical Right Shift C, A, Q
1	0100	1011	-	-	Add ($A \oplus A + M$)
0	1010	0101	-	0	Logical Right Shift C, A, Q

$$\text{Result} = 1010\ 0101 = 2^7 + 2^5 + 2^2 + 2^0 = 165$$

Alternate Method of Unsigned Binary Multiplication



Algorithm:

Step 1: Clear the sum (accumulator A). Place the multiplicand in X and multiplier in Y.

Step 2: Test Y_0 ; if it is 1, add content of X to the accumulator A.

Step 3: Logical Shift the content of X left one position and content of Y right one position.

Step 4: Check for completion; if not completed, go to step 2.

Example: Multiply 7×6

Sum	X	Y	Count	Remarks
000000	000111	110	3	Initialization
000000	001110	011	2	Left shift X, Right Shift Y
001110	011100	001	1	Sum \leftarrow Sum + X, Left shift X, Right Shift Y
101010	111000	000	0	Sum \leftarrow Sum + X, Left shift X, Right Shift Y

$$\text{Result} = 101010 = 2^5 + 2^3 + 2^1 = 42$$

Signed Multiplication (Booth Algorithm) – 2's Complement Multiplication

Multiplicand and multiplier are placed in Q and M register respectively. There is also one bit register placed logically to the right of the least significant bit Q_0 of the Q register and designated as Q_{-1} .

The result of multiplication will appear in A and Q register. A and Q_{-1} are initialized to zero if two bits (Q_0 and Q_{-1}) are the same (11 or 00) then all the bits of A, Q and Q_{-1} registers are shifted to the right 1 bit. If the two bits differ then the multiplicand is added to or subtracted from the A register depending on whether the two bits are 01 or 10. Following the addition or subtraction the arithmetic right shift occurs. When count reaches to zero, result resides into AQ in the form of signed integer $[-2^{n-1} * a_{n-1} + 2^{n-2} * a_{n-2} + \dots + 2^1 * a_1 + 2^0 * a_0]$.

Example: Multiply $9 \times -3 = -27$ using Booth

Algorithm $+3 = 00011$, $-3 = 11101$ (2's complement of +3)

A	Q	Q_{-1}	Add (M)	Sub ($\overline{M} + 1$)	Count	Remarks
00000	11101	0	01001	10111	5	Initialization
10111 11011	11101 11110	0 1	- -	- -	- 4	Sub ($A \leftarrow A - M$) as $Q_0Q_{-1} = 10$ Arithmetic Shift Right A, Q, Q_{-1}
00100 00010	11110 01111	1 0	- -	- -	- 3	Add ($A \leftarrow A + M$) as $Q_0Q_{-1} = 01$ Arithmetic Shift Right A, Q, Q_{-1}
11001 11100	01111 10111	0 1	- -	- -	- 2	Sub ($A \leftarrow A - M$) as $Q_0Q_{-1} = 10$ Arithmetic Shift Right A, Q, Q_{-1}
11110	01011	1	-	-	1	Arithmetic Shift Right A, Q, Q_{-1} as $Q_0Q_{-1} = 11$
11111	00101	1	-	-	0	Arithmetic Shift Right A, Q, Q_{-1} as $Q_0Q_{-1} = 11$

$$\text{Result in AQ} = 11111\ 00101 = -2^9 + 2^8 + 2^7 + 2^6 + 2^5 + 2^2 + 2^0 = -512 + 256 + 128 + 64 + 32 + 4 + 1 = -27$$

5.4 Division Algorithm

Division is somewhat more than multiplication but is based on the same general principles. The operation involves repetitive shifting and addition or subtraction.

First, the bits of the dividend are examined from left to right, until the set of bits examined represents a number greater than or equal to the divisor; this is referred to as the divisor being able to divide the number. Until this event occurs, 0s are placed in the quotient from left to right. When the event occurs, a 1 is placed in the quotient and the divisor is subtracted from the partial dividend. The result is referred to as a *partial remainder*. The division follows a cyclic pattern. At each cycle, additional bits from the dividend are appended to the partial remainder until the result is greater than or equal to the divisor. The divisor is subtracted from this number to produce a new partial remainder. The process continues until all the bits of the dividend are exhausted.

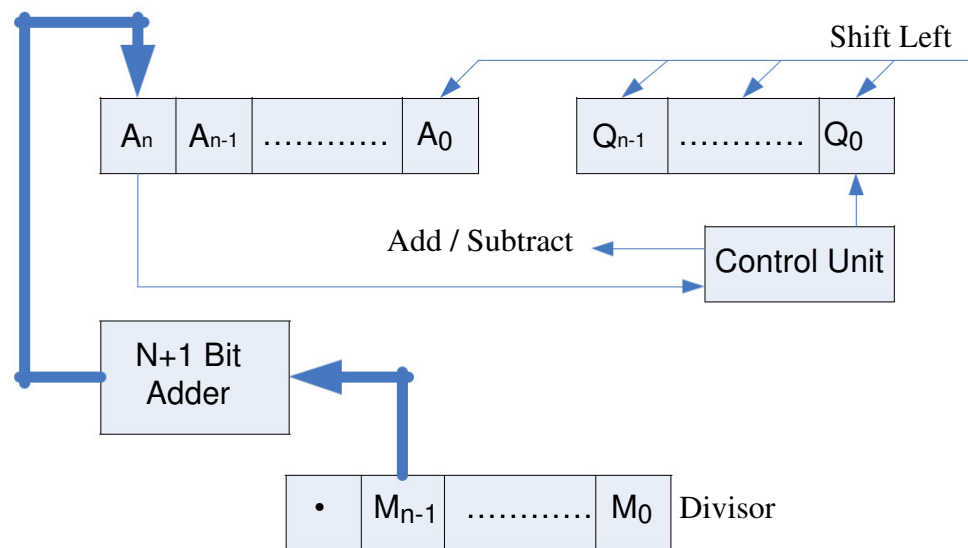
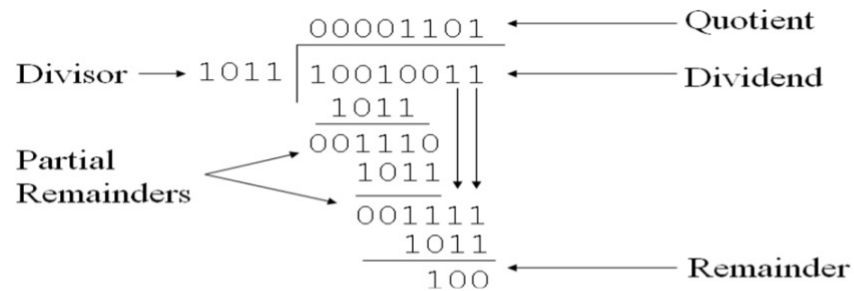
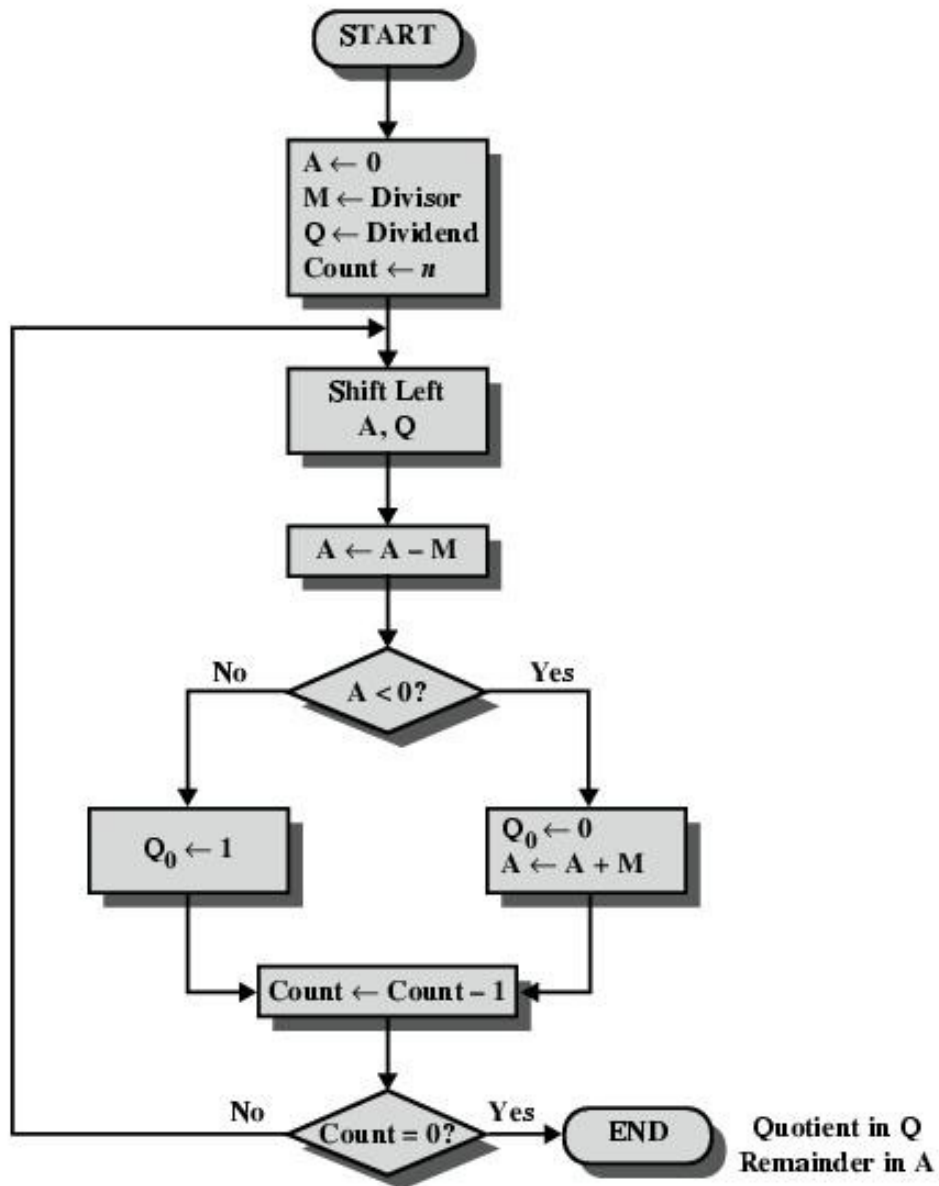


Fig.: Block Diagram of Division Operation



Algorithm:

Step 1: Initialize A, Q and M registers to zero, dividend and divisor respectively and counter to n where n is the number of bits in the dividend.

Step 2: Shift A, Q left one binary position.

Step 3: Subtract M from A placing answer back in A. If sign of A is 1, set Q_0 to zero and add M back to A (restore A). If sign of A is 0, set Q_0 to 1.

Step 4: Decrease counter; if counter > 0 , repeat process from step 2 else stop the process. The final remainder will be in A and quotient will be in Q.

Example: Divide 15 (1111) by 4 (0100)

A	Q	M	M + 1	Count	Remarks
00000	1111	00100	11100	4	Initialization
00001 1 1101 00001	111□ 111□ 111 0	- - -	- - -	- - 3	Shift Left A, Q Sub ($A \leftarrow A - M$) $Q_0 \leftarrow 0$, Add ($A \leftarrow A + M$)
00011 1 1111 00011	110□ 110□ 110 0	- - -	- - -	- - 2	Shift Left A, Q Sub ($A \leftarrow A - M$) $Q_0 \leftarrow 0$, Add ($A \leftarrow A + M$)
00111 0 0011 00011	100□ 100□ 100 1	- - -	- - -	- - 1	Shift Left A, Q Sub ($A \leftarrow A - M$) $Q_0 \leftarrow 1$
00111 0 0011 00011	001□ 001□ 001 1	- - -	- - -	- - 0	Shift Left A, Q Sub ($A \leftarrow A - M$) $Q_0 \leftarrow 1$

Quotient in Q = 0011 = 3

Remainder in A = 00011 = 3

Non – Restoring Division (Signed Binary Division)

Algorithm

Step 1: Initialize A, Q and M registers to zero, dividend and divisor respectively and count to number of bits in dividend.

Step 2: Check sign of A;

If $A < 0$ i.e. b_{n-1} is 1

q Shift A, Q left one binary position.

r Add content of M to A and store back in A.

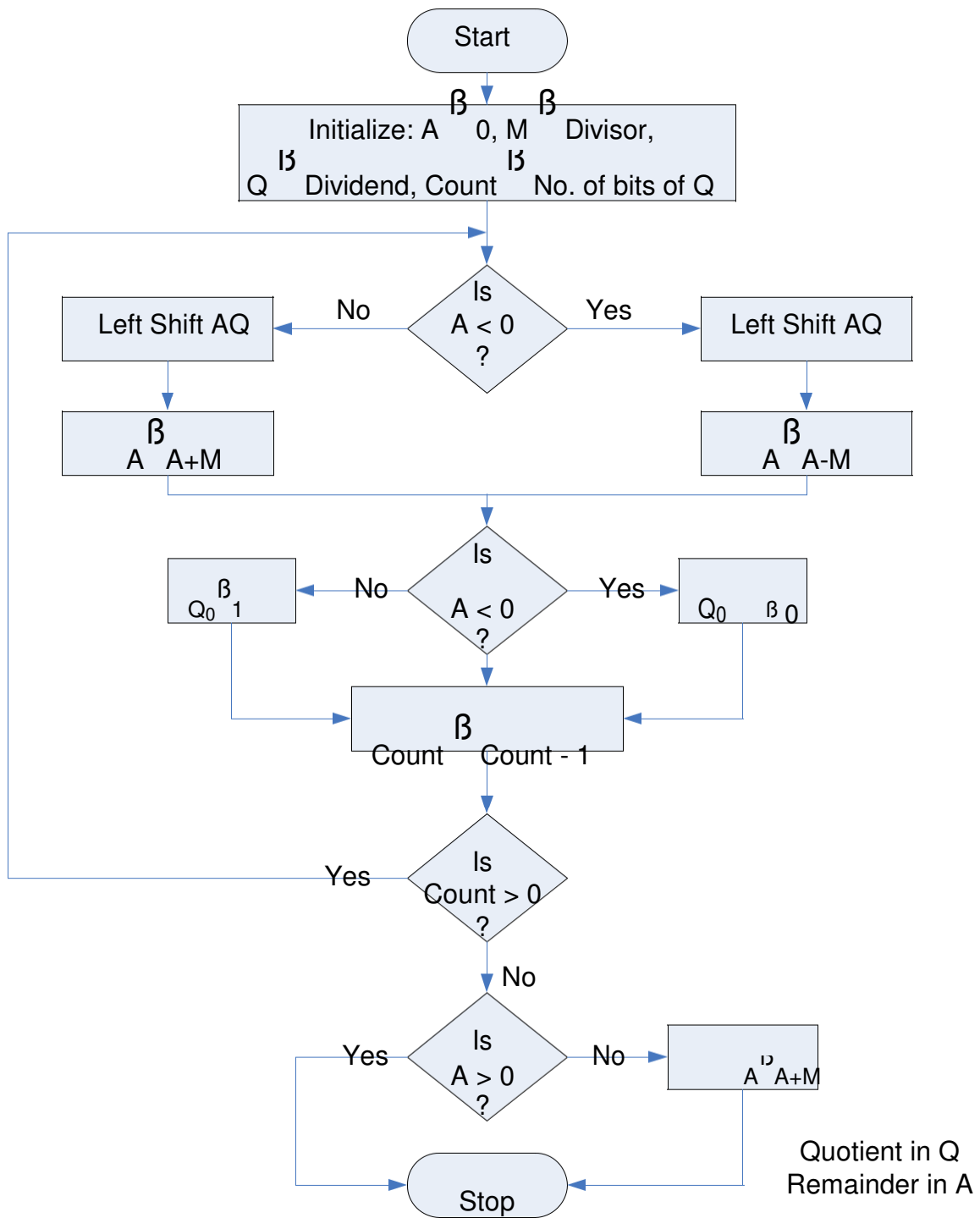
q Shift A, Q left one binary position.

r Subtract content of M to A and store back in A.

Step 3: If sign of A is 0, set Q_0 to 1 else set Q_0 to 0.

Step 4: Decrease counter. If counter > 0 , repeat process from step 2 else go to step 5.

Step 5: If $A \geq 0$ i.e. positive, content of A is remainder else add content of M to A to get the remainder. The quotient will be in Q.



Example: Divide 1110 (14) by 0011 (3) using non-restoring division.

A	Q	M	M + 1	Count	Remarks
0 0000	1110	00011	11101	4	Initialization
00001	110□	-	-	-	Shift Left A, Q
1 1110	110□	-	-	-	Sub (A ← A - M)
1 1110	1100	-	-	3	Set Q ₀ to 0
11101	100□	-	-	-	Shift Left A, Q
0 0000	100□	-	-	-	Add (A ← A + M)
0 0000	1001	-	-	2	Set Q ₀ to 1
00001	001□	-	-	-	Shift Left A, Q
1 1110	001□	-	-	-	Sub (A ← A - M)
1 1110	0010	-	-	1	Set Q ₀ to 0
11100	010□	-	-	-	Shift Left A, Q
1 1111	010□	-	-	-	Add (A ← A + M)
1 1111	0100	-	-	0	Set Q ₀ to 0
00010	0100	-	-	-	Add (A ← A + M)

Quotient in Q = 0011 = 3

Remainder in A = 00010 = 2

Floating Point Representation

The floating point representation of the number has two parts. The first part represents a signed fixed point numbers called mantissa or significand. The second part designates the position of the decimal (or binary) point and is called exponent. For example, the decimal no + 6132.789 is represented in floating point with fraction and exponent as follows.

Fraction	Exponent
+0.6132789	+04

This representation is equivalent to the scientific notation $+0.6132789 \times 10^{+4}$

The floating point is always interpreted to represent a number in the following form $\pm M \times R^{\pm E}$. Only the mantissa M and the exponent E are physically represented in the register (including their sign). The radix R and the radix point position of the mantissa are always assumed.

A floating point binary no is represented in similar manner except that it uses base 2 for the exponent.

For example, the binary no +1001.11 is represented with 8 bit fraction and 0 bit exponent as follows.

$$0.1001110 \times 2^{100}$$

Fraction	Exponent
01001110	000100

The fraction has zero in the leftmost position to denote positive. The floating point number is equivalent to $M \times 2^E = +(0.1001110)_2 \times 2^{+4}$

Floating Point Arithmetic

The basic operations for floating point arithmetic are

<i>Floating point number</i>	<i>Arithmetic Operations</i>
$X = X_s \times B^{\pm E}$	$X + Y = (X_s \times B^{\pm E - 1E} + Y_s) \times B^{1E}$

$$Y = Y_s \times B^{1E}$$

$$X - Y = (X_s \times B^{E-1E} - Y_s) \times B^{1E}$$

$$X * Y = (X_s \times Y_s) \times B^{E+1E}$$

$$X / Y = (X_s / Y_s) \times B^{E-YE}$$

There are four basic operations for floating point arithmetic. For addition and subtraction, it is necessary to ensure that both operands have the same exponent values. This may require shifting the radix point on one of the operands to achieve alignment. Multiplication and division are straighter forward.

A floating point operation may produce one of these conditions:

Exponent Overflow: A positive exponent exceeds the maximum possible exponent value.

Exponent Underflow: A negative exponent which is less than the minimum possible value.

Significand Overflow: The addition of two significands of the same sign may carry in a carry out of the most significant bit.

Significand underflow: In the process of aligning significands, digits may flow off the right end of the significand.

Floating Point Addition and Subtraction

In floating point arithmetic, addition and subtraction are more complex than multiplication and division. This is because of the need for alignment. There are four phases for the algorithm for floating point addition and subtraction.

- o Check for zeros:

Because addition and subtraction are identical except for a sign change, the process begins by changing the sign of the subtrahend if it is a subtraction operation. Next; if one is zero, second is result.

- p Align the Significands:

Alignment may be achieved by shifting either the smaller number to the right (increasing exponent) or shifting the large number to the left (decreasing exponent).

- q Addition or subtraction of the significands:

The aligned significands are then operated as required.

- r Normalization of the result:

Normalization consists of shifting significand digits left until the most significant bit is nonzero.

Example: Addition

$$X = 0.10001 * 2^{110}$$

$$Y = 0.101 * 2^{100}$$

Since $E_Y < E_X$, Adjust Y

$$Y = 0.00101 * 2^{100} * 2^{010} = 0.00101 * 2^{110}$$

So, $E_Z = E_X = E_Y = 110$

$$\text{Now, } M_Z = M_X + M_Y = 0.10001 + 0.00101 = 0.10110$$

$$\text{Hence, } Z = M_Z * 2^{E_Z} = 0.10110 * 2^{110}$$

Example: Subtraction

$$X = 0.10001 * 2^{110}$$

$$Y = 0.101 * 2^{100}$$

Since $E_Y < E_X$, Adjust Y

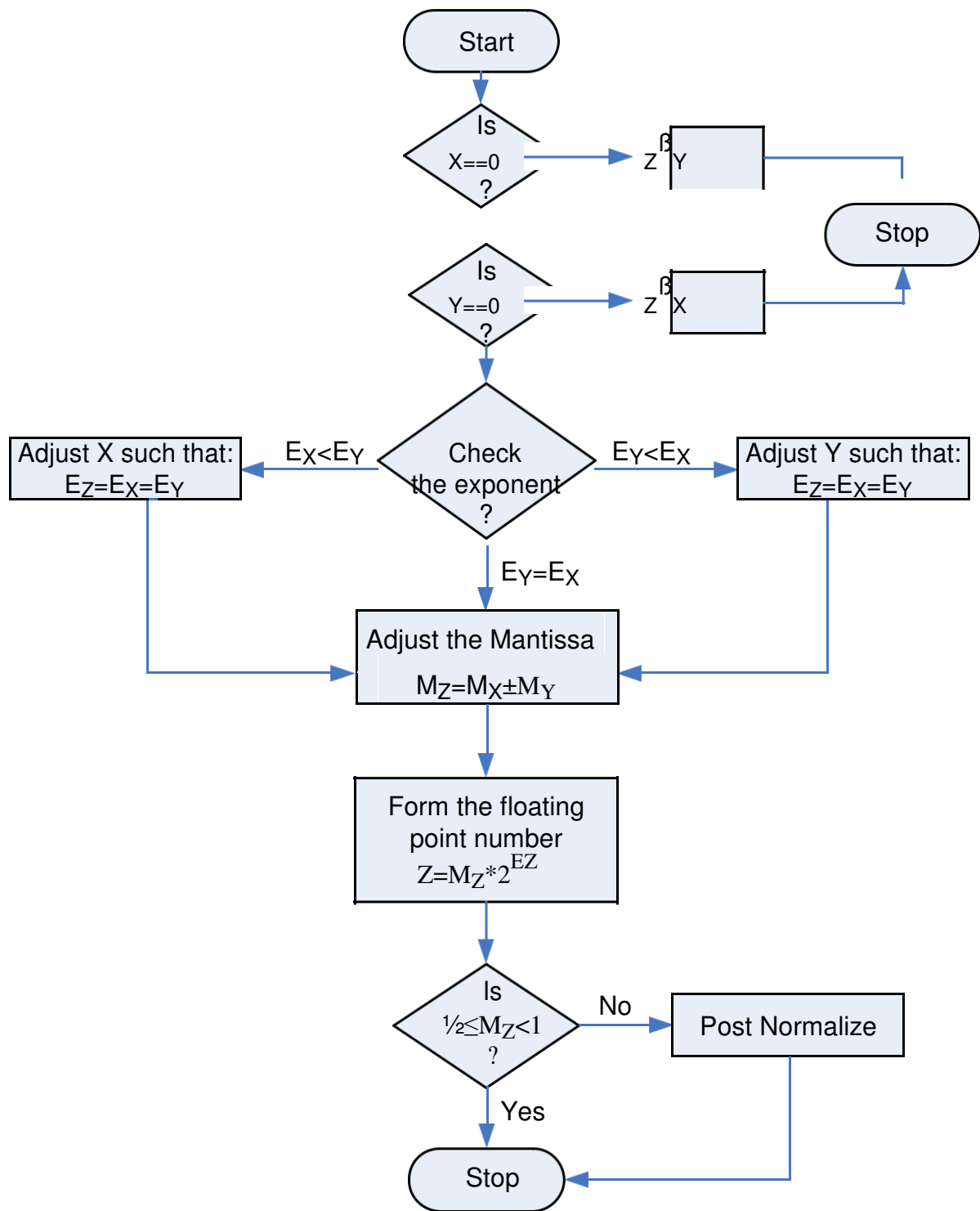
$$Y = 0.00101 * 2^{100} * 2^{010} = 0.00101 * 2^{110}$$

So, $E_Z = E_X = E_Y = 110$

$$\text{Now, } M_Z = M_X - M_Y = 0.10001 - 0.00101 = 0.01100$$

$$Z = M_Z * 2^{E_Z} = 0.01100 * 2^{110} \text{ (Un-Normalized)}$$

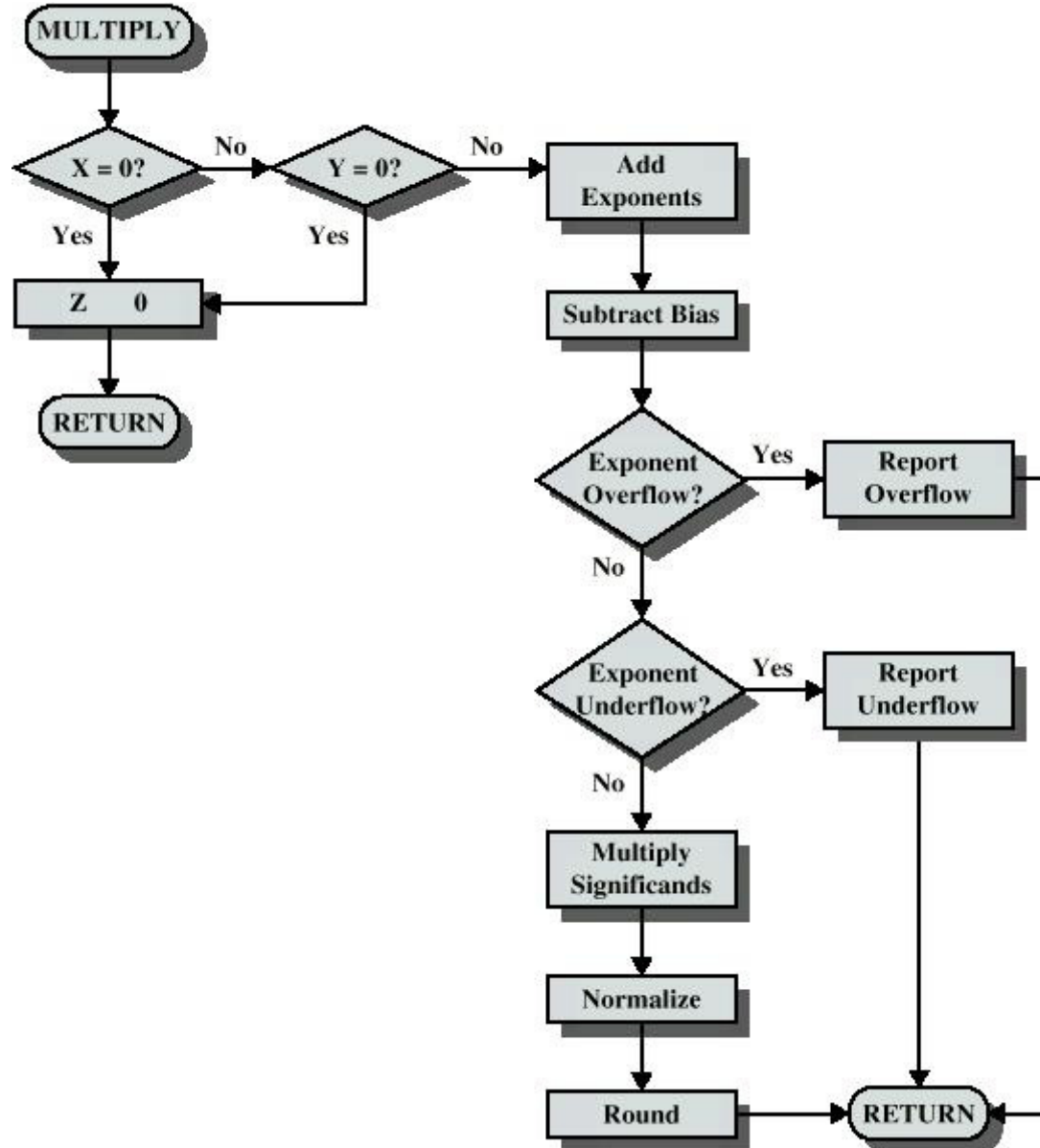
$$\text{Hence, } Z = 0.1100 * 2^{110} * 2^{-001} = 0.1100 * 2^{101}$$



Floating Point Multiplication

The multiplication can be subdivided into 4 parts.

- Check for zeroes.
- Add the exponents.
- Multiply mantissa.
- Normalize the product.



Example:

$$X = 0.101 * 2^{110}$$

$$Y = 0.1001 * 2^{-010}$$

$$\text{As we know, } Z = X * Y = (M_X * M_Y) * 2^{(EX + EY)}$$

$$Z = (0.101 * 0.1001) * 2^{(110-010)}$$

$$= 0.0101101 * 2^{100}$$

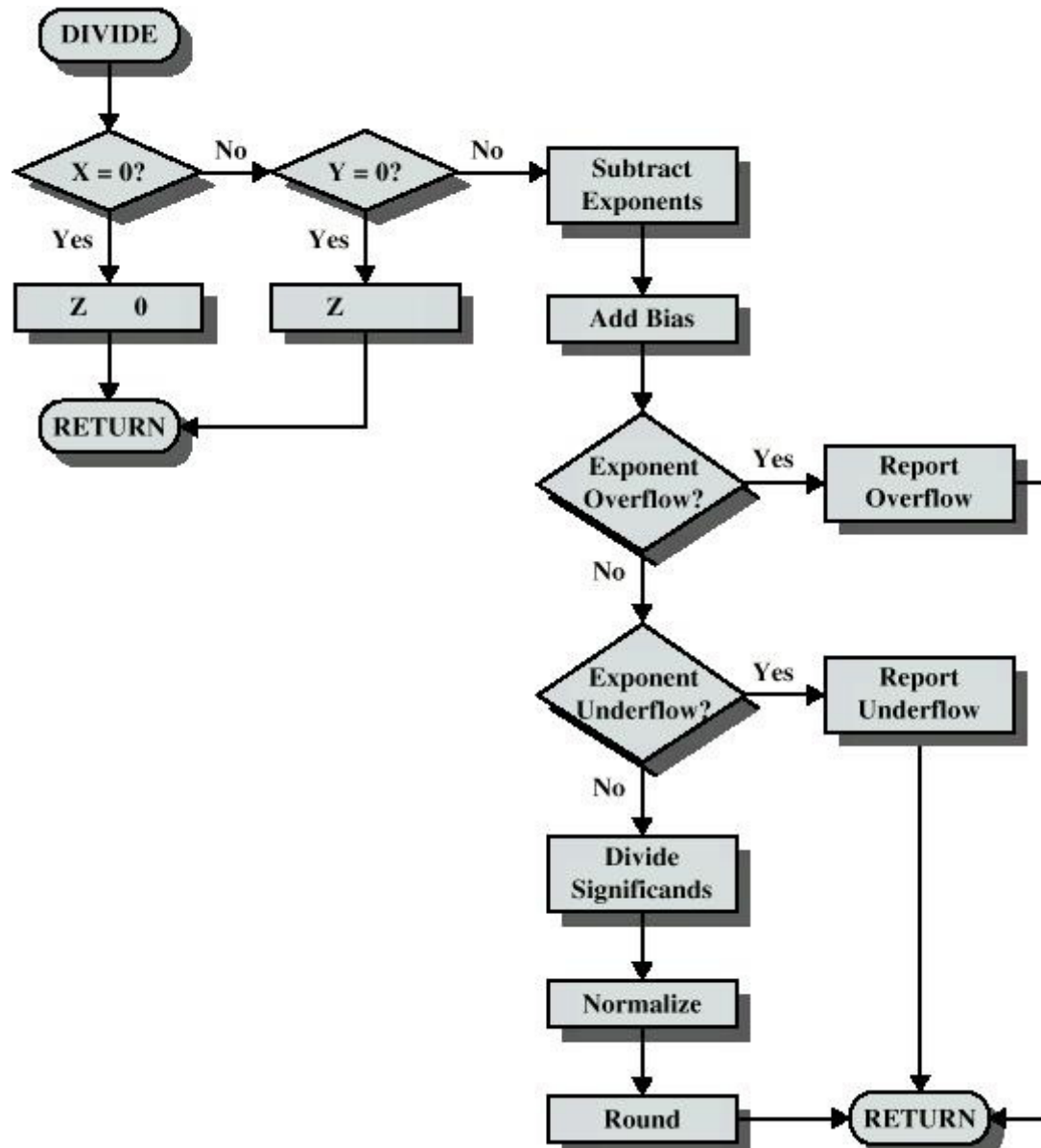
$$= 0.101101 * 2^{011} \text{ (Normalized)}$$

$$\begin{array}{r}
 0.1001 \\
 0.101 \\
 \hline
 1001 \\
 0000^* \\
 +1001^{**} \\
 \hline
 101101 = 0.0101101
 \end{array}$$

Floating Point Division

The division algorithm can be subdivided into 5 parts

- Check for zeroes.
- Initial registers and evaluates the sign.
- Align the dividend.
- Subtract the exponent.
- Divide the mantissa.



Example:

$$X = 0.101 * 2^{110}$$

$$Y = 0.1001 * 2^{-010}$$

$$\text{As we know, } Z = X / Y = (M_X / M_Y) * 2^{(EX - EY)}$$

$$M_X / M_Y = 0.101 / 0.1001 = (1/2 + 1/8) / (1/2 + 1/16) = 1.11 = 1.00011 \text{ } 0.11$$

$$* 2 = 0.22 \text{ } 0$$

$$0.22 * 2 = 0.44 \text{ } 0$$

$$0.44 * 2 = 0.88 \text{ } 0$$

$$0.88 * 2 = 1.76 \text{ } 1$$

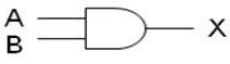
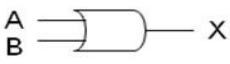
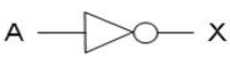
$$0.76 * 2 = 1.52 \text{ } 1$$

$$E_X - E_Y = 110 + 010 = 1000$$

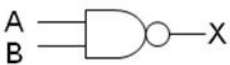
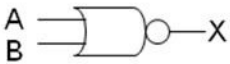

$$\text{Now, } Z = M_Z * 2^{EZ} = 1.00011 * 2^{1000} = 0.100011 * 2^{1001}$$

5.5 Logical Operation

Gate Level Logical Components

Name	Symbol	VHDL Equation	Truth Table															
AND		$X \leq A \text{ and } B$	<table><tr><th>A</th><th>B</th><th>X</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	X	0	0	0	0	1	0	1	0	0	1	1	1
A	B	X																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
OR		$X \leq A \text{ or } B$	<table><tr><th>A</th><th>B</th><th>X</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	X	0	0	0	0	1	1	1	0	1	1	1	1
A	B	X																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
NOT		$X \leq \text{not } A$	<table><tr><th>A</th><th>X</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	X	0	1	1	0									
A	X																	
0	1																	
1	0																	

Composite Logic Gates

Name	Symbol	VHDL Equation	Truth Table															
NAND		$X \leq \text{not } (A \text{ and } B)$	<table><tr><th>A</th><th>B</th><th>X</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	X	0	0	1	0	1	1	1	0	1	1	1	0
A	B	X																
0	0	1																
0	1	1																
1	0	1																
1	1	0																
NOR		$X \leq \text{not } (A \text{ or } B)$	<table><tr><th>A</th><th>B</th><th>X</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	X	0	0	1	0	1	0	1	0	0	1	1	0
A	B	X																
0	0	1																
0	1	0																
1	0	0																
1	1	0																
XOR		$X \leq A \text{ xor } B$	<table><tr><th>A</th><th>B</th><th>X</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	X	0	0	0	0	1	1	1	0	1	1	1	0
A	B	X																
0	0	0																
0	1	1																
1	0	1																
1	1	0																