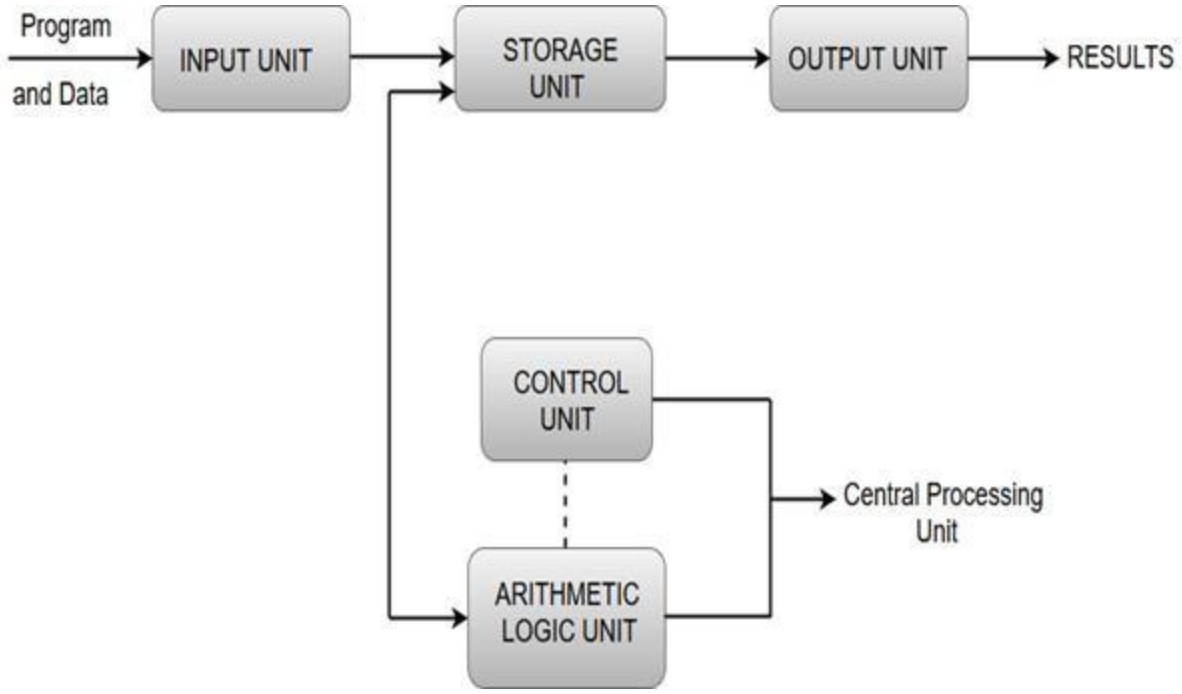
COA

UNIT - 1

Functional Units of Digital System

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



Input unit

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

Central processing unit

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

Memory unit

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

Arithmetic & logical unit

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

Control Unit

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

Output Unit

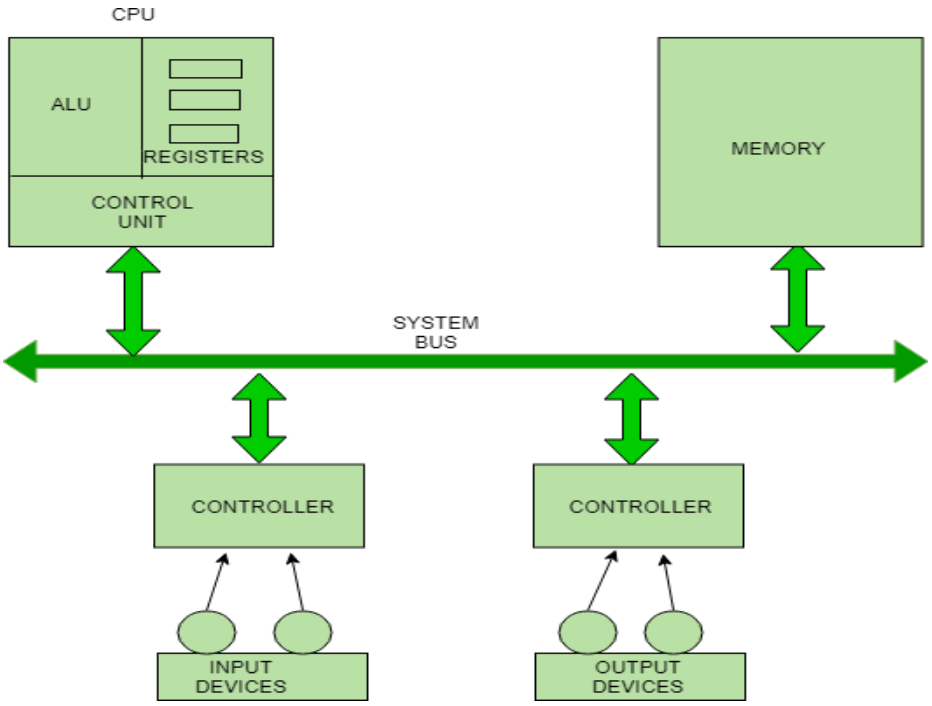
- The primary function of the output unit is to send the processed results to the user. Output devices display information in a way that the user can understand.
- Output devices are pieces of equipment that are used to generate information or any other response processed by the computer. These devices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

Interconnection between Functional Components :-

A computer consists of input unit that takes input, a CPU that processes the input and an output unit that produces output. All these devices communicate with each other through a common bus. A bus is a transmission path, made of a set of conducting wires over which data or information in the form of electric signals, is

passed from one component to another in a computer. The bus can be of three types – Address bus, Data bus and Control Bus.

Following figure shows the connection of various functional components:



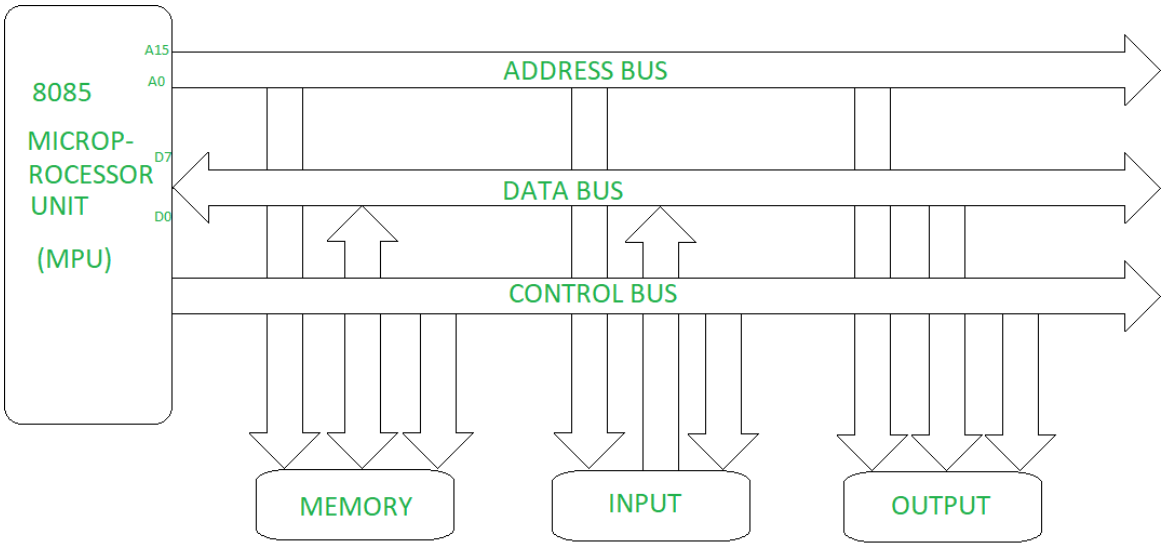
- The address bus carries the address location of the data or instruction. The data bus carries data from one component to another and the control bus carries the control signals. The system bus is the common communication path that carries signals to/from CPU, main memory and input/output devices. The input/output devices communicate with the system bus through the controller circuit which helps in managing various input/output devices attached to the computer.

Bus organization of 8085 microprocessor –

(Bus , Bus Architecture & Types of buses and bus atribution)

A digital system composed of many registers, and paths must be provided to transfer information from one register to another. The number of wires connecting all of the registers will be excessive if separate lines are used between each register and all other registers in the system.

Bus is a group of conducting wires which carries information, all the peripherals are connected to microprocessor through Bus. Diagram to represent bus Architecture



Bus organization system of 8085 Microprocessor

There are three types of buses.

1. Address Bus

It is a group of conducting wires which carries address only. Address bus is unidirectional because data flow in one direction, from microprocessor to memory or from microprocessor to Input/output devices (That is, Out of Microprocessor). Length of Address Bus of 8085 microprocessor is 16 Bit (That is, Four Hexadecimal Digits), ranging from 0000 H to FFFF H, (H denotes Hexadecimal). The microprocessor 8085 can transfer maximum 16 bit address which means it can address 65, 536 different memory location. The Length of the address bus determines the amount of memory a system can address.Such as a system with a 32-bit address bus can address 2^32 memory locations.If each memory location holds one byte, the addressable memory space is 4 GB.However, the actual amount of memory that can be accessed is usually much less than this theoretical limit due to chipset and motherboard limitations.

2. Data Bus

It is a group of conducting wires which carries Data only.Data bus is bidirectional because data flow in both directions, from microprocessor to memory or Input/Output devices and from memory or Input/Output devices to microprocessor. Length of Data Bus of 8085 microprocessor is 8 Bit (That is, two Hexadecimal Digits), ranging from 00 H to FF H. (H denotes Hexadecimal). When it is write operation, the processor will put the data (to be written) on the data bus, when it is read operation, the memory controller will get the data from specific memory block and put it into the data bus. The width of the data bus is directly related to the largest number that the bus can carry, such as an 8 bit bus can represent 2 to the power of 8 unique values, this equates to the number 0 to 255. A 16 bit bus can carry 0 to 65,535.

3. Control Bus –

It is a group of conducting wires, which is used to generate timing and control signals to control all the associated peripherals, microprocessor uses control bus to process data, that is what to do with selected memory location. Some control signals are:

• Memory read • Memory write • I/O read • I/O Write • Opcode fetch

If one line of control bus may be the read/write line.

If the wire is low (no electricity flowing) then the memory is read, if the wire is high (electricity is flowing) then the memory is written.

BUS Arbitration in Computer Organization -

Bus Arbitration refers to the process by which the current bus master accesses and then leaves the control of the bus and passes it to the another bus requesting processor unit. The controller that has access to a bus at an instance is known as Bus master.

A conflict may arise if the number of DMA controllers or other controllers or processors try to access the common bus at the same time, but access can be given to only one of those. Only one processor or controller can be Bus master at the same point of time. To resolve these conflicts, Bus Arbitration procedure is implemented to coordinate the activities of all devices requesting memory transfers. The selection of the bus master must take into account the needs of various devices by establishing a priority system for gaining access to the bus. The Bus Arbiter decides who would become current bus master. There are two approaches to bus arbitration:

1. Centralized bus arbitration – A single bus arbiter performs the required arbitration.

2. Distributed bus arbitration – All devices participate in the selection of the next bus master.

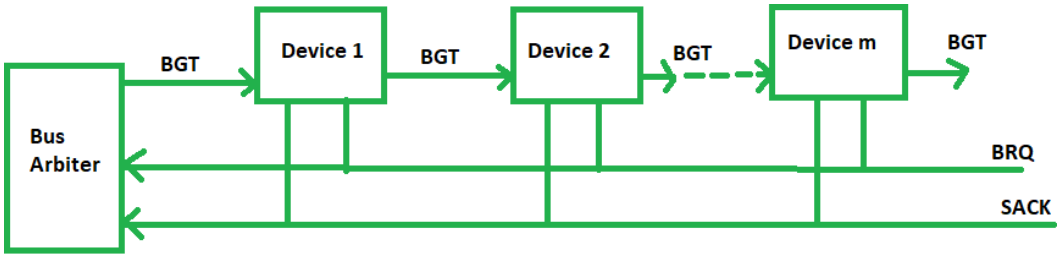
Methods of BUS Arbitration –

There are three bus arbitration methods:

(i) Daisy Chaining method – It is a centralized bus arbitration method. During any bus cycle, the bus master may be any device – the processor or any DMA controller unit, connected to the bus.

All devices are effectively assigned static priorities according to their locations along a bus grantcontrol line (BGT). The device closest to the central bus arbiter is assigned the highest priority.Requests for bus access are made on a common request line, BRQ. Similarly, the common acknowledge signal line (SACK) is used to indicate the use of bus. When no device is using the bus,the SACK is inactive.

The central bus arbiter propagates a bus grant signal (BGT) if the BRQ line is high and acknowledge signal (SACK) indicates that the bus is idle. The first device, which has issued a bus request, receives the BGT signal and stops the latter's propagation.This sets the bus-busy flag in the bus arbiter by activating SACK and the device assumes bus control. On completion, it resets the bus-busy flag in the arbiter and a new BGT signal is generated If other requests are outstanding (i.e., BRQ is still active). The first device simply passes the BGT signal to the next device in the line.



Daisy chained bus arbitration

Advantages –

- Simplicity and Scalability.
- The user can add more devices anywhere along the chain, up to a certain maximum value.

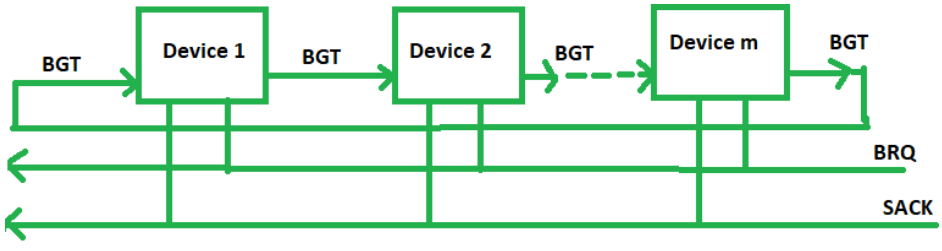
Disadvantages –

- The value of priority assigned to a device is depends on the position of master bus.
- Propagation delay is arises in this method.
- If one device fails then entire system will stop working.

(ii) Polling or Rotating Priority method –

In this method, the devices are assigned unique priorities and complete to access the bus, but the priorities are dynamically changed to give every device an opportunity to access the bus.

In the polling scheme, no central bus arbiter exists, and the bus-grant line (BGT) is connected from the last device back to the first in a closed loop.Whichever device is granted access to the bus serves as bus arbiter for the following arbitration (an arbitrary device is selected to have initial access to the bus). Each device's priority for a given arbitration is determined by that device's distance along the bus-grant line from the device currently serving as bus arbiter the latter device has the lowest priority.Hence, the priorities change dynamically with each bus cycle.



Rotating priority bus arbitration

Advantages –

- This method does not favor any particular device and processor.
- The method is also quite simple.
- If one device fails then entire system will not stop working.

Disadvantages –

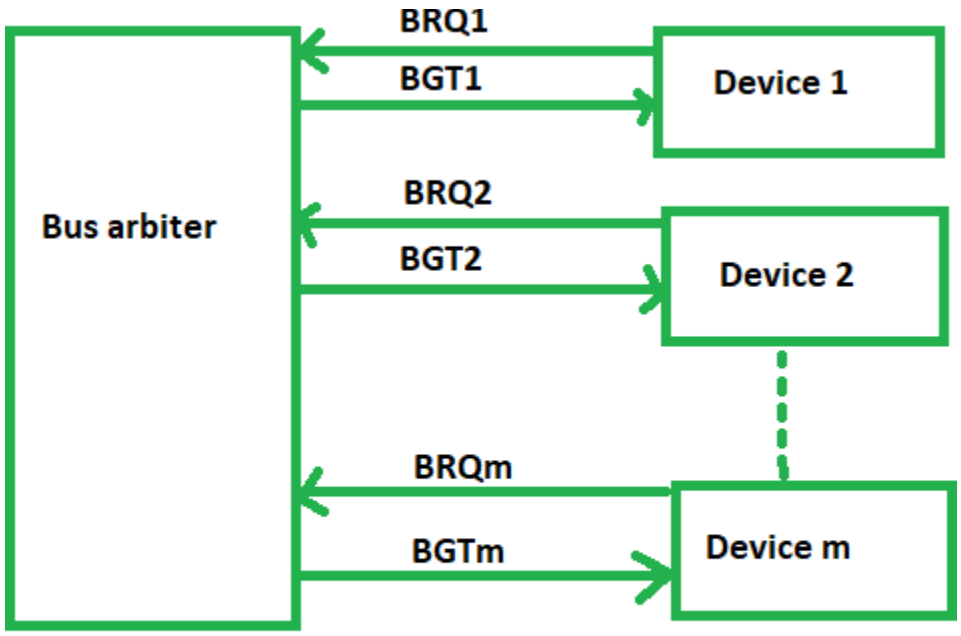
- Adding bus masters is different as increases the number of address lines of the circuit.

(iii) Fixed priority or Independent Request method -

In this method, the bus control passes from one device to another only through the centralized bus arbiter.

In bus independent request method, the bus control passes from one device to another only through the centralized bus arbiter.Each device has a dedicated BRQ output line and BGT input line. If there are m devices, the bus arbiter has m BRQ inputs and m BGT outputs.The arbiter follows a priority order with different priority level to each device.At a given time, the arbiter issues bus grant (BGT) to the highest priority device among the devices who have issued bus requests.

This scheme needs more hardware but generates fast response.



Fixed priority bus arbitration method

Advantages –

- This method generates fast response.

Disadvantages –

- Hardware cost is high as large no. of control lines are required.

Registers – Registers are a type of computer memory used to quickly accept, store, and transfer data and instructions that are being used immediately by the CPU. The registers used by the CPU are often termed as Processor registers.

A processor register may hold an instruction, a storage address, or any data (such as bit sequence or individual characters).

The computer needs processor registers for manipulating data and a register for holding a memory address. The register holding the memory location is used to calculate the address of the next instruction after the execution of the current instruction is completed.

Bus and Memory Transfers –

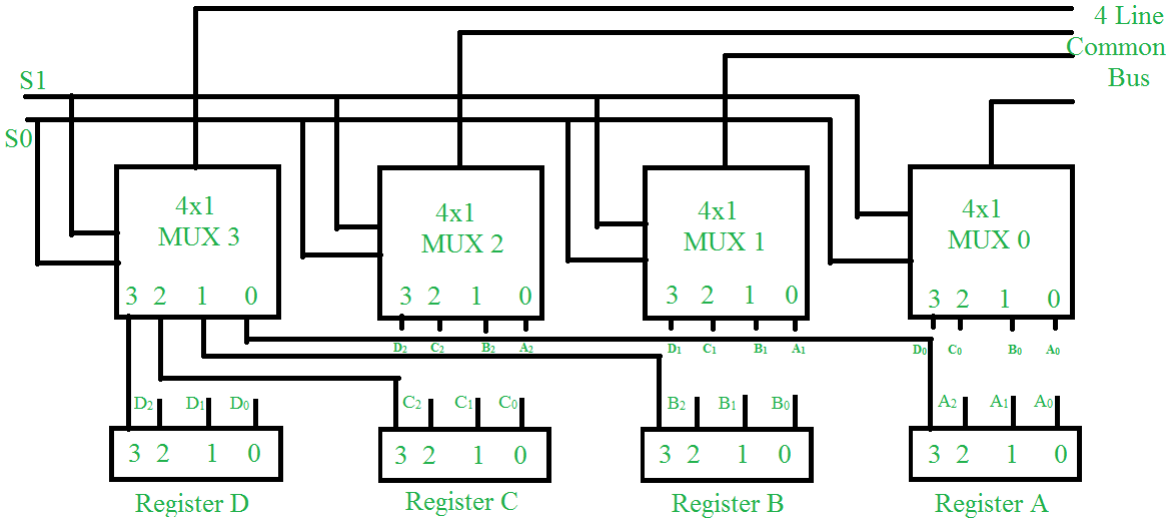
A digital system composed of many registers, and paths must be provided to transfer information from one register to another. The number of wires connecting all of the registers will be excessive if separate lines are used between each register and all other registers in the system.

A bus structure, on the other hand, is more efficient for transferring information between registers in a multi-register configuration system.

A bus consists of a set of common lines, one for each bit of register, through which binary information is transferred one at a time. Control signals determine which register is selected by the bus during a particular register transfer.

The following block diagram shows a **Bus system for four registers**. It is constructed with the help of four 4 * 1 Multiplexers each having four data inputs (0 through 3) and two selection inputs (S1 and S2).

We have used labels to make it more convenient for you to understand the inputoutput configuration of a Bus system for four registers. For instance, output 1 of register A is connected to input 0 of MUX1.



The two selection lines S1 and S2 are connected to the selection inputs of all four multiplexers. The selection lines choose the four bits of one register and transfer them into the four-line common bus.

When both of the select lines are at low logic, i.e. S1S0 = 00, the 0 data inputs of all four multiplexers are selected and applied to the outputs that forms the bus. This, in turn, causes the bus lines to receive the content of register A since the outputs of this register are connected to the 0 data inputs of the multiplexers.

Similarly, when S1S0 = 01, register B is selected, and the bus lines will receive the content provided by register B.

Note:-

The number of multiplexers needed to construct the bus is equal to the number of bits in each register. The size of each multiplexer must be 'k * 1' since it multiplexes 'k' data lines. For instance, a common bus for eight registers of 16 bits each requires 16 multiplexers, one for each line in the bus. Each multiplexer must have eight data input lines and three selection lines to multiplex one significant bit in the eight registers.

Memory Transfer –

Most of the standard notations used for specifying operations on memory transfer are stated below.

The transfer of information from a memory unit to the user end is called a Read operation.

The transfer of new information to be stored in the memory is called a Write operation.

A memory word is designated by the letter M.

We must specify the address of memory word while writing the memory transfer operations.

The address register is designated by AR and the data register by DR.

Thus, a read operation can be stated as:

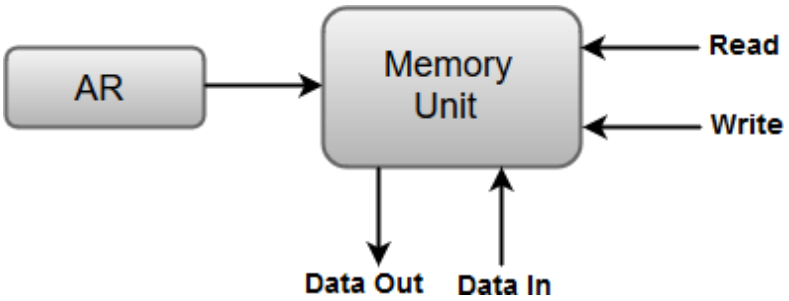
1. Read: DR ← M [AR]

The Read statement causes a transfer of information into the data register (DR) from the memory word (M) selected by the address register (AR).

And the corresponding write operation can be stated as:

1. Write: M [AR] ← R1

The Write statement causes a transfer of information from register R1 into the memory word (M) selected by address register (AR).



Processor Organization –

Processor organization refers to the design and structure of a computer's central processing unit (CPU), which is responsible for executing instructions and managing data. It encompasses various aspects, including the architecture, components, and how these components interact to perform computations.

There are following three types of processor (CPU) organization used in general :

1. Stack based organization
2. General Register based organization

General Register Organization

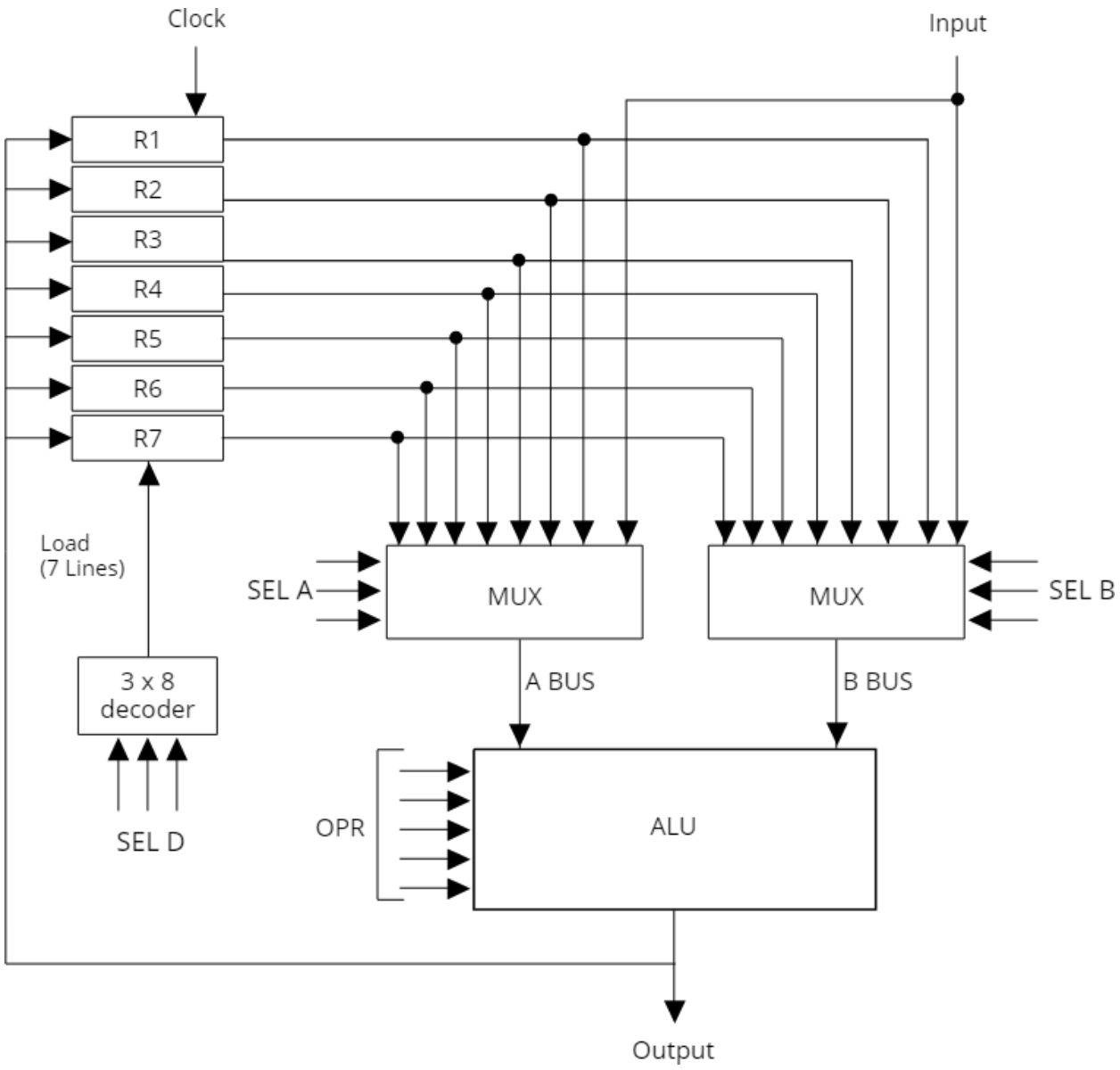
- To understand General Register Organization, it's essential to grasp the major components within a CPU -

1. **Storage Components:** These include registers and flip-flops, serving as temporary storage for data.
2. **Execution Components:** The Arithmetic Logic Unit (ALU) is responsible for carrying out calculations and logical operations.
3. **Transfer Components:** The bus facilitates the transfer of data between storage and execution components.
4. **Control Component:** The control unit oversees and directs the functioning of other components within the CPU.

- Memory locations play a crucial role in storing various data types such as pointers, counters, return addresses, temporary results, and partial products. However, accessing memory is a time-consuming task. To enhance efficiency, intermediate values are stored in processor registers. These registers are interconnected through a common bus system, allowing seamless communication not only for direct data transfer but also for coordinating various microoperations.
- **Definition of General Register Organization:** In computing, General Register Organization refers to the systematic arrangement and utilization of registers

within the CPU. These registers serve as high-speed, temporary storage for data and play a vital role in enhancing computational efficiency by minimizing the need for frequent memory access.

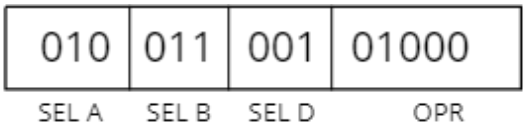
A Bus Organization for Seven CPU Registers –



- The depicted bus organization features seven CPU registers, and its functionality is detailed as follows:
- The output of each register is linked to two multiplexers (MUX), both of which play a crucial role in transferring register data into the Arithmetic Logic Unit (ALU).
- Two buses, A and B, are utilized for data transfer. The selection lines in each multiplexer determine whether to choose data from a register or from input data. Data is transmitted to the ALU via buses A and B.
- The OPR (Operation) signal serves to define the type of operation to be executed by the ALU.
- The result of the operation conducted by the ALU can be directed to other units within the system or stored in any of the processor registers.
- A decoder is employed to select the register where the result will be stored. The decoder activates one of the register load inputs, specifying the destination register for storing the result.

Example, Let we want to perform the operation R1 ← R2 + R3

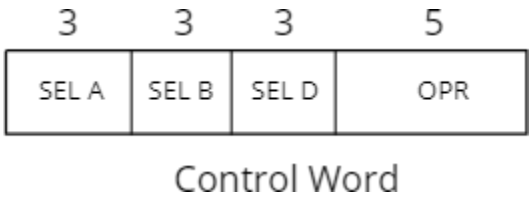
- To do this operation, Control Unit generates following singal (Control Word).



1. SEL A = 010. So, MUX A transfer the content of R2 into bus A.
2. SEL B = 011. So, MUX B transfer the content of R3 into bus B.
3. OPR = 01000. So, ALU performs addition.
4. SEL D = 001. to transfer the result in register R1.

Control Word

- A control word, designed for the aforementioned CPU organization, consists of four fields as illustrated below:



Control Word

- The three bits of SEL A are dedicated to transferring the contents of a register into BUS A.
- The three bits of SEL B are assigned to transferring the contents of a register into BUS B.
- The three bits of SELD are utilized for selecting a destination register. This facilitates the decision of whether to store the result in a register or to transmit it outside the ALU.
- The five bits of OPR define the type of operation to be performed by the ALU. This field governs the arithmetic or logical operation executed by the ALU based on the specified opcode.

Code for four Register Selection –

Binary Code	SEL A	SEL B	SEL D
000	Input	Input	None
001	R1	R1	R1
010	R2	R2	R2
011	R3	R3	R3
100	R4	R4	R4
101	R5	R5	R5
110	R6	R6	R6
111	R7	R7	R7

Operation Code for ALU –

OPR Select	Operation	Symbol
00000	Transfer A	TSFA
00001	Increment A	INCA
00010	A+B	ADD
00101	A-B	SUB
00110	Decrement A	DECA
01000	A.B	AND
01010	OR A and B	OR
01100	XOR A B	XOR
01110	Complement A	COMA
10000	Shift Right A	SHRA
11000	Shift Left A	SHLA

Stack Organization -

- The memory of a CPU can be organized as a STACK, a structure where information is stored in a Last-In-First-Out (LIFO) manner. This means that the item last stored is the first to be removed or popped.
- To manage the items, a stack uses a stack pointer (SP) register. The stack pointer stores the address of the last item in the stack, essentially pointing to the topmost element. Stack operations involve two main actions:

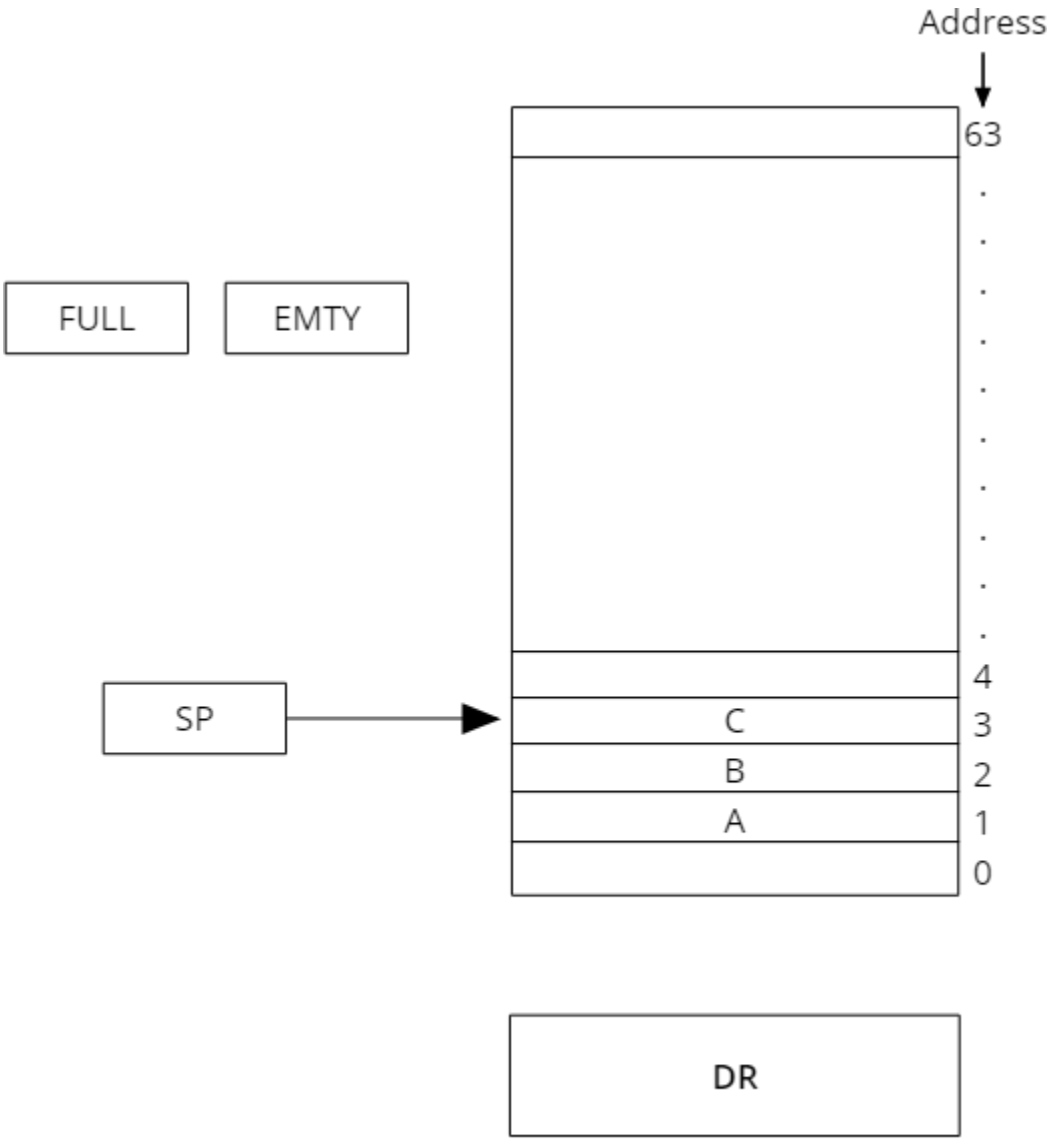
1. **Insertion (Push):** When an item is added to the stack, it is referred to as insertion or a push operation.
2. **Deletion (Pop):** When an item is removed from the stack, it is known as deletion or a pop operation.

- There are two main types of stacks:

1. **Register Stack:** Utilizes processor registers to create a stack structure, enhancing speed and efficiency in certain operations.

2. **Memory Stack:** Involves using dedicated memory locations to implement the stack structure.

1 - Register Stack



Block diagram of a 64-word stack.

- When processor registers are organized in a stack-like fashion, it is termed a register stack. The diagram above illustrates a 64-word register stack.
- The stack pointer (SP) contains the address of the topmost element in the stack.
- When the stack is empty, the EMPTY flag is set to 1, and when the stack is full, the FULL flag is set to 1.
- The DR (Data Register) contains the data either being popped from or pushed into the stack.
- Additional benefits of a register stack include faster access times and reduced memory bus contention, making it suitable for certain computing tasks requiring high-speed data manipulation.
- For example, in the figure, three items (A, B, and C) are placed in the stack, with item C at the top. Thus, the stack pointer (SP) holds the address of C (SP = 3).

PUSH Operation :

- When performing a PUSH operation to add an element (let's say E) to the stack, the following steps are executed:
    - **Step 1:** Increment the Stack Pointer (SP) by 1 so that it points to an empty slot.
      SP ← SP + 1 [Increment stack pointer]
    - **Step 2:** Store the value of the Data Register (DR) at the address pointed to by SP.
      M[SP] ← DR [Write the item on top of the stack]
    - **Step 3:** Check boundary conditions.
      If (SP = 0), then set FULL ← 1 indicating the stack is full.
      EMPTY ← 0 signifies that the stack is not empty.

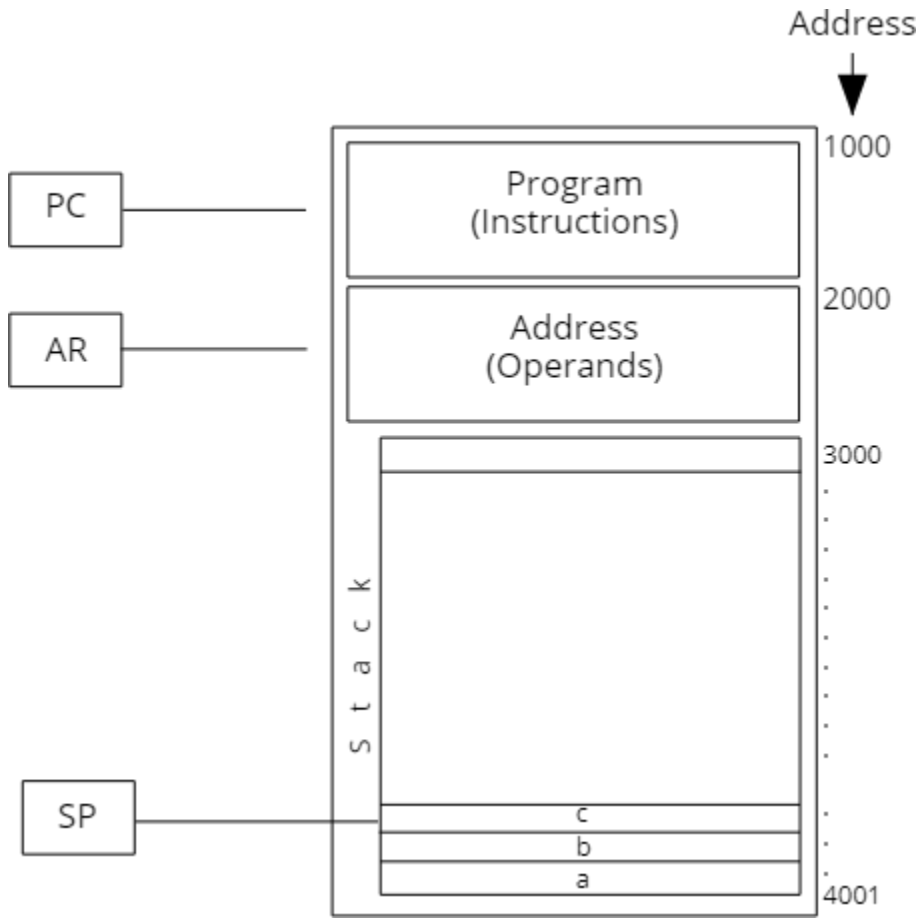
POP Operation:

- When performing a POP operation to remove an element from the stack, the following steps are executed:
    - **Step 1:** Retrieve the data from the address stored in the Stack Pointer (SP) and store it in the Data Register (DR).
      DR ← M[SP] [Fetch item from the top of the stack]

- **Step 2:** Decrement the Stack Pointer (SP) by 1.
  SP → SP - 1
- **Step 3:** Check boundary conditions.
  if (SP = 0) then (EMPTY ← 1) [Check if the stack is empty]
  FULL ← 0 [Mark the stack as not full]

2 - Memory Stack

- When primary memory (RAM) is organized in the form of a stack, it is referred to as a Memory Stack.



- The Program Counter (PC) indicates the address of the next instruction in the program.
- The Address Register (AR) points to an array of data within the memory stack.
- The Stack Pointer (SP) identifies the top of the stack.

- In the illustrated figure, the initial value of SP is 4001, and the stack grows with decreasing addresses. Consequently, the first item stored in the stack is at address 4000, the second item at address 3999, and the last item at address 3000.

PUSH Operation

```
SP → SP - 1
M[SP] ← DR
```

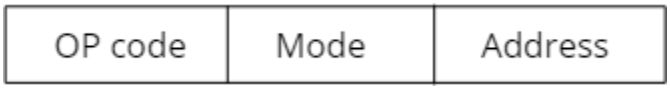
POP Operation

```
DR ← M[SP]
SP → SP + 1
```

- The PUSH operation involves decrementing the Stack Pointer (SP) to allocate space for a new item and storing the value from the Data Register (DR) at the address pointed to by the updated SP.
- The POP operation retrieves the item from the top of the stack by copying the data from the address indicated by SP to DR. Subsequently, SP is incremented to free up space in the stack.

Addressing Modes –

An instruction format is a collection of bits that defines the type of instruction, operands, and the type of operation. The instruction format is represented by a rectangular box, and a basic instruction format includes the following fields: Opcode, Mode, and Address.



- **Opcode:** Defines the type of operation to be performed, such as add, subtract, complement, and shift.
- **Address field:** Defines the address of operands.
- **Mode (or addressing mode) field:** Defines the method by which operands are fetched, modifying the address field of the instruction to determine the actual address of the data.

Addressing Modes:

- **1. Implied Addressing Mode:** The zero-address instruction and all instructions using the accumulator are implied-mode instructions. For example, the "complement accumulator" instruction is implied-mode because the operand is in the accumulator.
- **2. Immediate Addressing Mode:** In this mode, the operand is specified in the instruction itself, having an operand field instead of an address field.
  For example: `ADD 10, 20`.
- **3. Register Addressing Mode:** Used when data is stored in processor registers, and the address part of the instruction contains the address of the processor register.
  For example: `SUB R1, R2`.
- **4. Register Indirect Addressing Mode:** The instruction has the address of a processor register, which contains the address of the operand in memory.
- **5. Direct Address Mode:** The instruction has the address of a memory cell where the data is stored, and the effective address is the address stored in the instruction.
- **6. Indirect Address Mode:** The address field of the instruction has a memory address where the data is stored.
- **7. Autoincrement or Autodecrement Address Mode:** Used when fetching a series of data, and the address part of the instruction gives the starting address, which is incremented or decremented to fetch the next data from memory.
- **8. Relative Address Mode:** The content of the program counter is added to the address part of the instruction to obtain the effective address of data.
- **9. Indexed Addressing Mode:** The content of an index register is added to the address part of the instruction to obtain the effective address, useful for accessing data arrays in memory.
- **10. Base Register Addressing Mode:** Similar to indexed addressing mode, the content of a base register is added to the address part of the instruction to obtain the effective address.

Data Transfer Instructions

Data transfer and manipulation are fundamental aspects of computer architecture, integral to the execution of instructions within a computing system.

- These instructions are typically categorized into three main types:

1. Data Transfer Instructions

2. Data Manipulation Instructions
3. Program Control Instructions

Data Transfer Instructions -

Data transfer instructions facilitate the movement of data from one location to another within the computer system. These instructions are essential for controlling the flow of information, including:

- Data transfer between memory and processor registers
- Data transfer between processor registers and input or output devices
- Data transfer between different processor registers

The table below presents a list of eight common data transfer instructions widely utilized across various computer architectures

Load (LD)	It transfer data from memory to processor register
Store (ST)	It transfer data from processor register to memory
Move (MOV)	It transfer data between processor registers
Exchange (XCH)	It swaps information between two registers or a register and a memory
Input (IN) / Output (OUT)	It transfer data between processor register and input or output units.
Push (PUSH) / Pop (POP)	It transfer data between processor registers and a memory stack.

Data Manipulation Instructions -

Data manipulation instructions play a critical role in performing operations on data within a computer system. These instructions can be broadly categorized into three types:

1. **Arithmetic Instructions**
2. **Logical and Bit Manipulation Instructions**
3. **Shift Instructions**

Arithmetic Instructions -

Arithmetic instructions encompass fundamental operations such as addition, subtraction, multiplication, and division. The table below provides a list of typical arithmetic instructions

Name	Menomics	Micro Operations
Increment	INC	R1 ←R1 + 1
Decrement	DEC	R1 ←R1 - 1
Add	ADD	R2 + R3
Subtract	SUB	R2 - R3
Multiply	MUL	R2 * R3
Divide	DIV	R2 / R3
Add with carry	ADDC	R2 + R3 + 1
Subtract with borrow	SUBB	R2 + R3'- 1
Negate (2's complement)	NEG	(R3')'

Logical and Bit Manipulation Instructions –

Logical instructions are designed to perform binary operations on data stored in registers. These instructions consider each bit of the operand individually. Here are some common logical and bit manipulation instructions

Name	Menomics
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Complement carry	COMC
Enable interrupt	EI
Disable interrupt	DI

Shift Instructions -

Shift instructions move bits within a register either to the left or right. Logical shifts insert 0 to the end bit position. The table below illustrates various types of shift instructions

Name	Menomics
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right through carry	RORC
Rotate left through carry	RORL

Program Control Instructions -

In a computer system, instructions are typically stored in successive memory locations, and the execution of a program involves fetching instructions from these consecutive memory locations. As each instruction is fetched, the program counter is incremented to contain the address of the next instruction in sequence. Program control instructions play a crucial role in directing the flow of a program and managing the execution process.

General program control instructions encompass a variety of operations that dictate the execution flow. Some of these instructions are outlined in the table below

Name	Menomics
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Compare (by subtration)	CMP

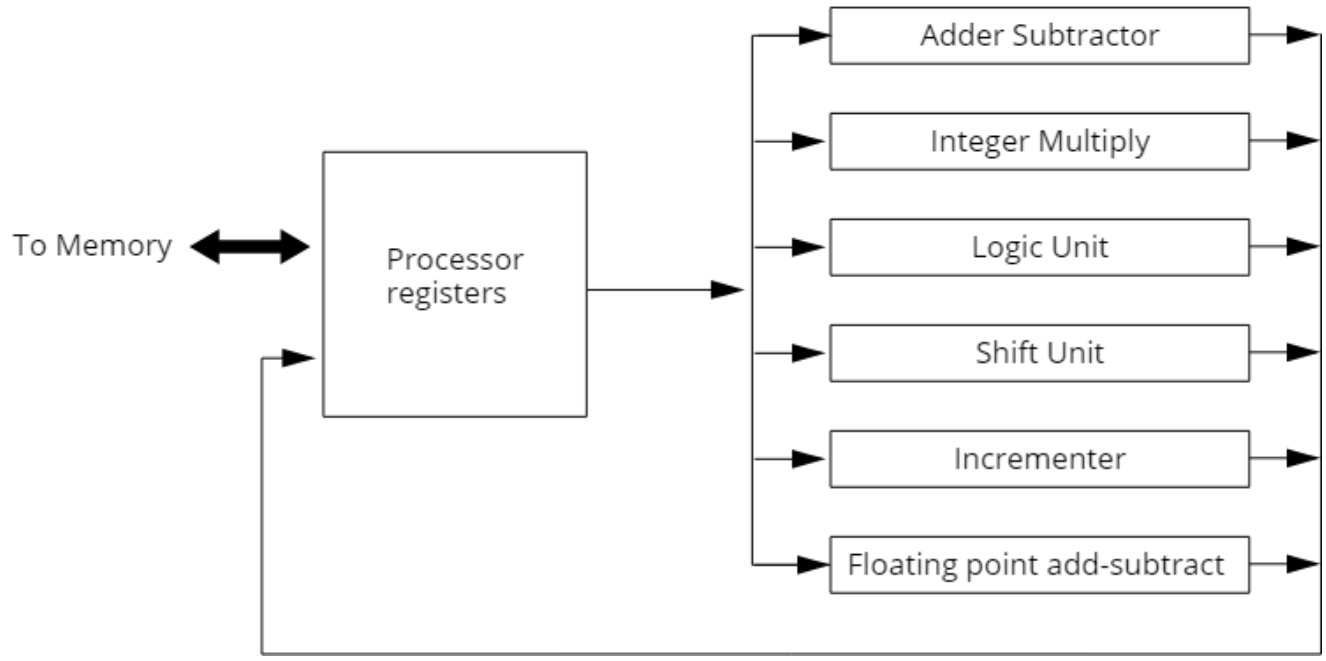
Parallel Processing -

- In older computers, only a single instruction used to be executed at a time, leading to the wastage of ALU time and an inability to fully utilize processing capabilities. To address this inefficiency, the concept of parallel processing was introduced.
- Parallel processing involves the simultaneous execution of multiple instructions, allowing for concurrent data processing and faster execution times. Instead of processing instructions sequentially, parallel processing techniques enable more efficient use of computing resources. For example:
  - While an instruction is being executed in the ALU, the next instruction can be read from memory.
  - A system may have two or more ALUs, capable of executing multiple instructions simultaneously.
  - Multiple processors may operate concurrently, enhancing overall system performance.

The primary purpose of parallel processing is to accelerate computer capabilities by leveraging increased hardware resources.

- Parallel processing can be examined at various levels of complexity:
  - At a lower level, the distinction between parallel and serial operations is based on the type of registers used.
  - Shift registers operate in a serial fashion, processing one bit at a time, while registers with parallel load operate with all bits of the word simultaneously.
  - At a higher level of complexity, parallel processing can involve a multiplicity of functional units performing identical or different operations simultaneously.

The following diagram illustrates a processor with multiple functional units, showcasing the additional components added to increase productivity and enable parallel processing



Parallel processing can be classified in various ways, one of which is introduced by M.J. Flynn. Flynn's classification divides computers into four major groups based on the sequence of instructions read from memory and the operations performed in the instruction and data streams:

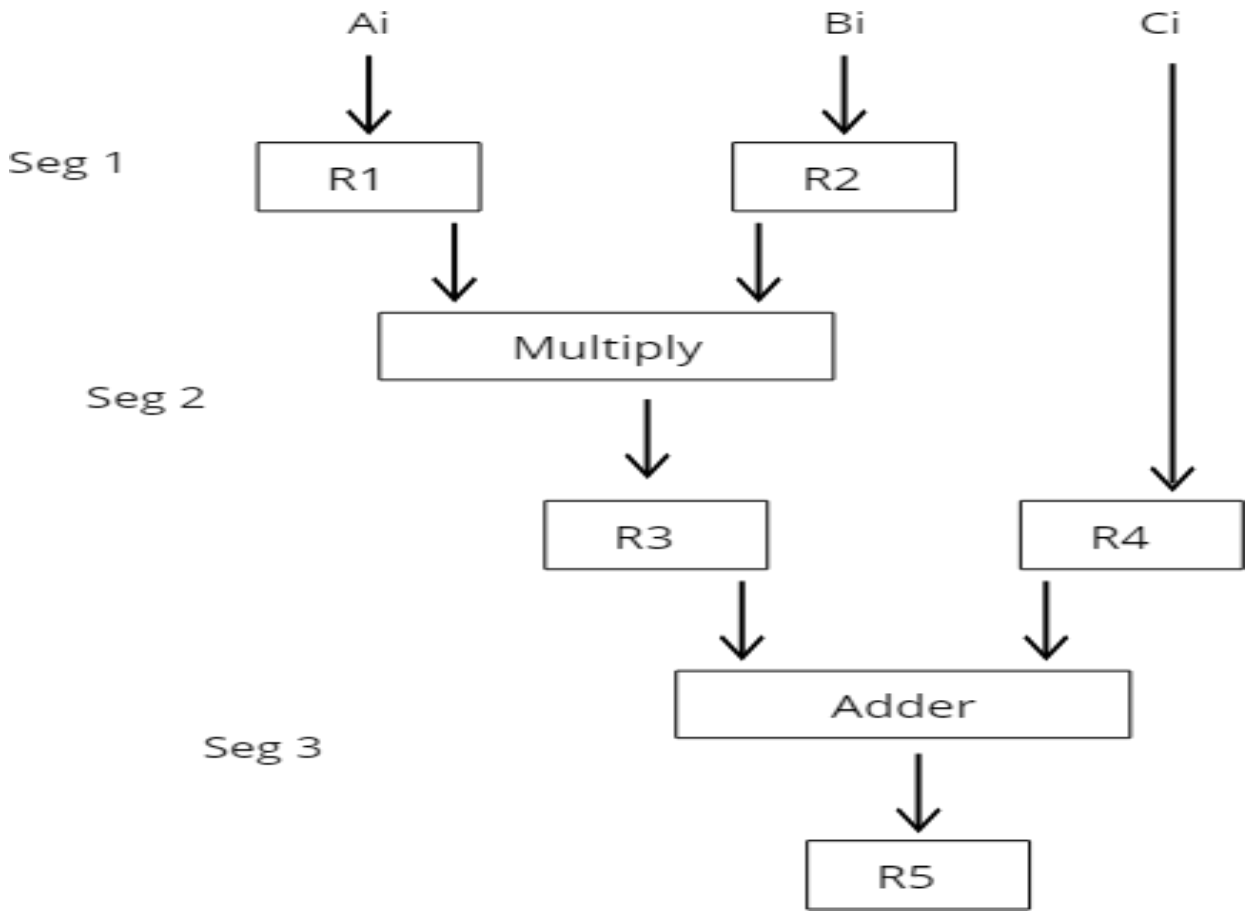
1. **Single Instruction Stream, Single Data Stream (SISD):** In SISD architecture, a single instruction stream is executed on a single data stream. This is the traditional von Neumann architecture where one instruction is processed at a time.
2. **Single Instruction Stream, Multiple Data Streams (SIMD):** SIMD architecture involves the processing of a single instruction simultaneously on multiple data streams. This is commonly seen in vector processors, where the same operation is applied to multiple data elements concurrently.
3. **Multiple Instruction Streams, Single Data Stream (MISD):** MISD architecture, although rare in practice, involves multiple instruction streams operating on a single data stream. This concept is not widely implemented due to its complexity and limited applicability.
4. **Multiple Instruction Streams, Multiple Data Streams (MIMD):** MIMD architecture allows for the simultaneous execution of multiple instruction streams on multiple data

streams. This is a versatile and widely used parallel processing architecture found in modern multi-core processors and parallel computing systems.

Pipelining

- Pipelining involves dividing a process into several suboperations, with each suboperation associated with a segment.
- The output of each segment is stored in a register, and this register information is passed to the next segment, facilitating a continuous flow of data.
- Each segment operates independently, allowing for concurrent execution of all segments in the pipeline.
- The term "pipelining" is derived from the sequential transfer of information from one segment to another.

Example: Performing Ai * Bi + Ci; for i = 1 to 7.
Segment 1: R1 ← Ai, R2 ← Bi
Segment 2: R3 ⇆ R1 * R2, R4 ← Ci
Segment 3: R5 ← R3 + R4



Pipelining is an efficient technique that allows for the overlap of different stages of instruction execution, thereby improving overall throughput. Each segment operates concurrently, enabling the processor to handle multiple instructions simultaneously. This approach significantly enhances the speed and efficiency of data processing in modern computer architectures.

Arithmetic

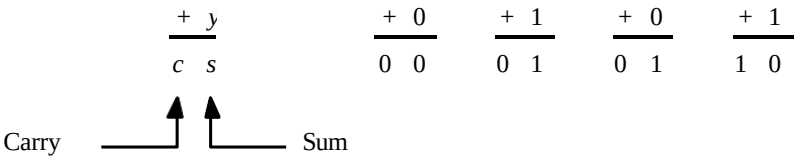
A basic operation in all digital computers is the addition and subtraction of two numbers They are implemented, along with the basic logic functions such as AND,OR, NOT,EX- OR in the ALU subsystem of the processor. In this chapter we will study how to implement these operations by using different techniques.

Addition and Subtraction of Signed Numbers

Half Adder

Figure 1(a),(b),(c),(d): Implementation of Half Adder

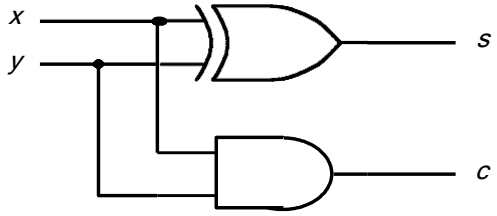
x		0	0	1	1



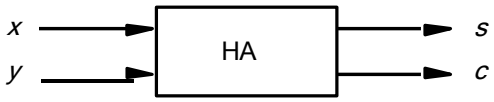
(a) The four possible cases

x	y	Carry c	Sum s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(b) Truth table



(c) Circuit



(d) Graphical symbol

Full Adder

The following figure 2 shows the logic truth table for the sum and carry-out functions for adding equally weighted bits x_i and y_i in two numbers X and Y. The figure also shows logic expressions for these functions, along with an example of addition of the 4-bit unsigned numbers 7 and 6.

x_i	y_i	Carry-in c_i	Sums s_i	Carry-out 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

$$s_i = \bar{x}_i\bar{y}_i c_i + \bar{x}_i y_i \bar{c}_i + x_i \bar{y}_i \bar{c}_i + x_i y_i c_i = x_i \oplus y_i \oplus c_i$$

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

Example:

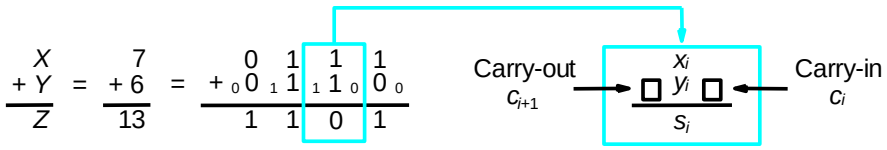


Figure 2

Implementation

The logic expression for S_i in figure 2 can be implemented with a 3-input XOR gate,used in figure 3(a)as a part of the logic required for a single stage of binary addition. The carry-out function, Ci+1, is implemented with a two level AND-OR logic circuit.

Using AND -OR gate

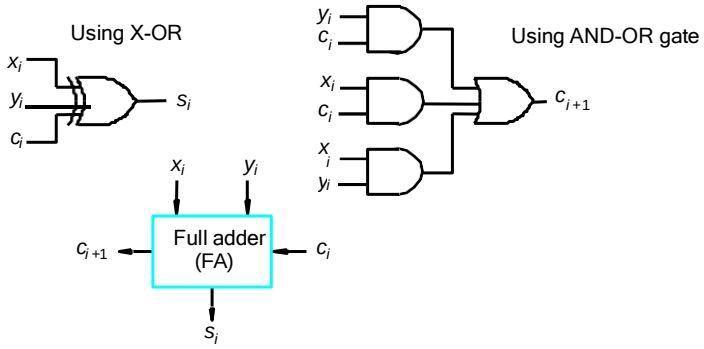


Figure 3(a) Logic For a single stage

A cascaded connection of n full adder blocks, as shown in figure 4(a), can be use to add two n-bit numbers. Since the carries must propagate, or ripple through this cascade, the configuration is called n-bit ripple carry adder.

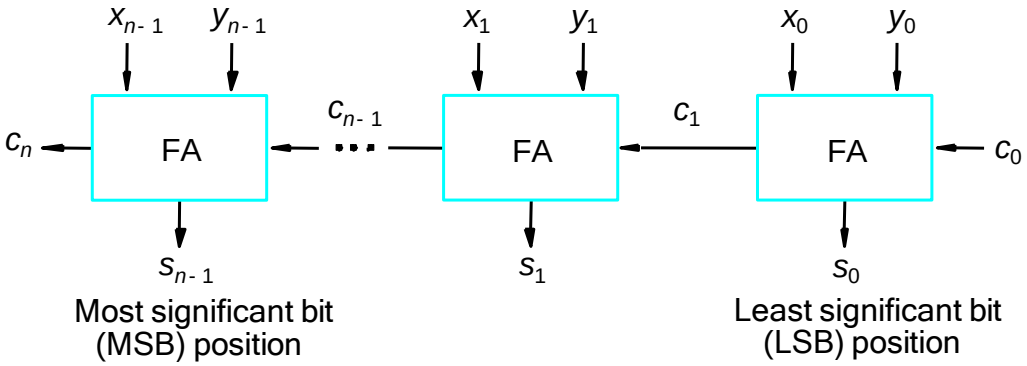


Figure 4(a) An n-bit ripple carry adder

Overflow – Overflow occurs in signed numbers having same signs, and sign of the result is different, and also it is shown that carry bits C_n and C_{n-1} are different. A circuit is added to detect overflow, eg. $C_{n-1} \oplus C_n$

In order to perform the subtract operation X-Y on 2's complement numbers X and Y, we form the 2s-complement of Y and add it to X. The logic circuit network shown in figure (5) can be used to perform either addition or subtraction based on the value applied to the Add/Sub input control line. This line is set to 0 for addition, applying the Y vector un changed to one of the adder inputs along with a carry-in signal,C0 of 0 . When Add/Sub control line is set to 1, the Y vector is 1's complemented by the XOR gates and C0 is set to 1 to complete the 2's complementation of Y. Remember that 2's complementing a negative number is done exactly same manner as for positive number. An XOR gate can be added to Figure(5) to detect the overflow condition $C_{n-1} \oplus C_n$

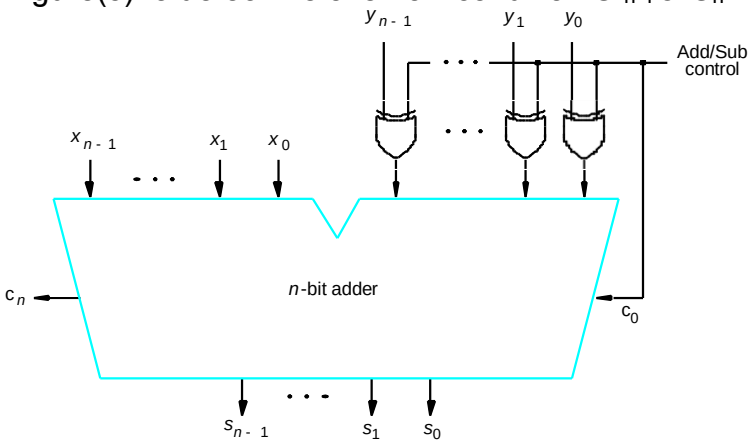


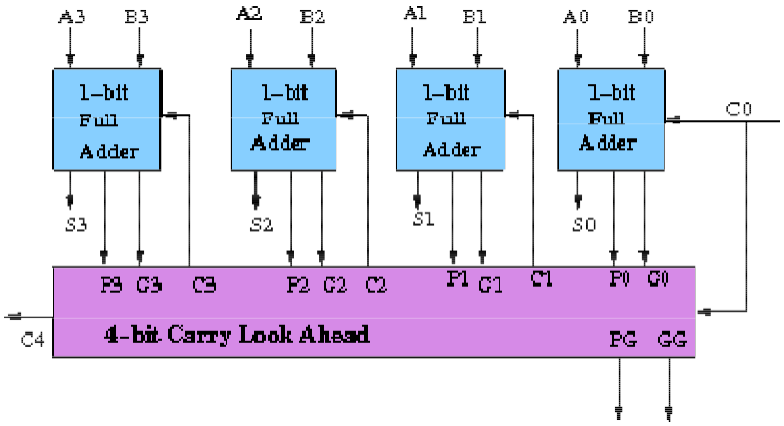
Figure 6.3. Binary addition-subtraction logic netw ork.

Design of Fast Adders

If an n-bit ripple carry adder is used in the addition /subtraction unit of Figure (3), it may have too much delay in developing its outputs, s_0 through s_{n-1} and c_n. The delay through any combinational logic network constructed from gates in a particular technology is determined by adding up the number of logic gate delays along the longest signal propagation path through the network. In the case of n-bit ripple-carry adder, the longest path is from inputs x0,y0, and c0 at the LSB position to outputs c_n and s_{n-1} at the most- significant-bit(MSB) position.

Design of Carry Lookahead Adders

To reduce the computation time, there are faster ways to add two binary numbers by using carry lookahead adders. They work by creating two signals P and G known to be Carry Propagator and Carry Generator. The **carry propagator** is propagated to the next level whereas the **carry generator** is used to generate the output carry, regardless of input carry. The block diagram of a 4-bit Carry Lookahead Adder is shown here below -



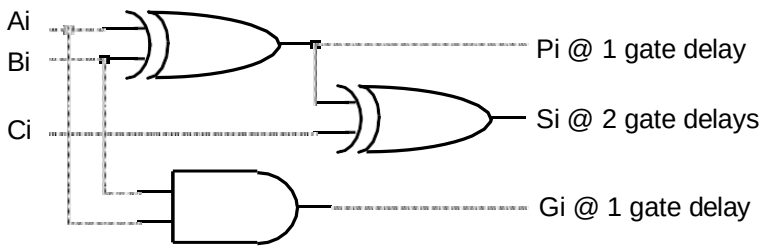
Let us consider the design of a 4 bit adder is shown in figure (6). The carries can be implemented as C1=G0+P0C0

$C_2=G_1+P_1G_0+P_1P_0C_0$

$C_3=G_2+P_2G_1+P_2P_1G_0+P_2P_1P_0C_0$

$C_4=G_3+P_3G_2+P_3P_1G_1+P_3P_2P_1G_0+P_3P_2P_1P_0C_0$

Each of the carry equations can be implemented in a two-level logic network.Variables are the adder inputs and carry in to next stage



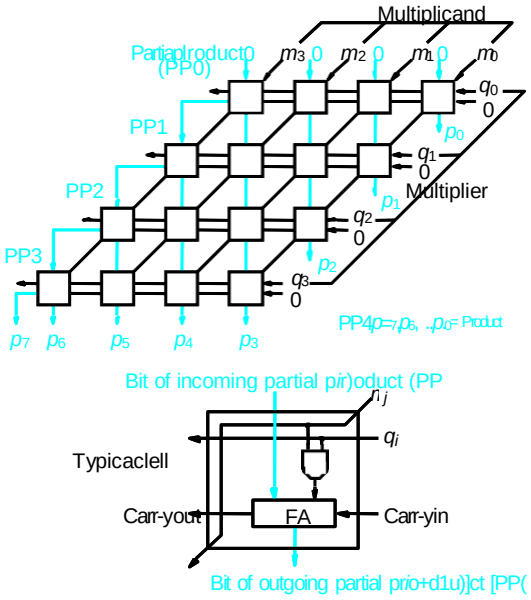
The number of gate levels for the carry propagation can be found from the circuit of full adder. The signal from input carry Cin to output carry Cout requires an AND gate and an OR gate, which constitutes two gate levels. So if there are four full adders in the parallel adder, the output carry C_5 would have 2 X 4 = 8 gate levels from C_1 to C_5. For an n-bit parallel adder, there are 2n gate levels to propagate through..

Multiplication of Positive numbers

The usual algorithm for multiplying integers by hand is illustrated in figure7(a) for the binary system.This algorithm applies to unsigned numbers and to positive signed numbers. The product of two n-digit numbers can be accommodated in 2n digits, so the product of the 4 bit numbers in this example fits into 8 bits.

				1	1	0	1	(13)
			×	1	0	1	1	(11)
				1	1	0	1	
			1	1	0	1		
		0	0	0	0			
	1	1	0	1				
1	0	0	0	1	1	1	1	(143)

Figure 7(a) Manual multiplication algorithm



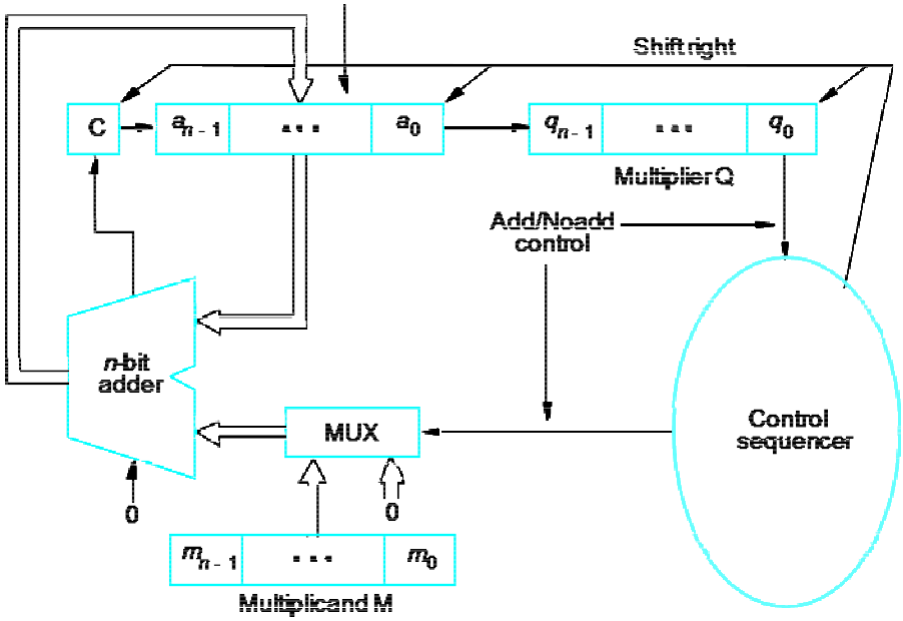
Figure

Binary multiplication of positive operands can be implemented in a combinational two dimensional logic array as shown in figure7(b). The main component in each cell is a full adder FA. The AND gate in each cell determines whether a multiplicand bit m_j , is added to the incoming partial product bit, based on the value of the multiplierbit qj. Each row I,where $0 \leq I \leq 3$, adds the multiplicand to the incoming partial product, PP_i to generate the outgoing partial product, $PP_{(i+1)}$, if $q_i=1$. If $q_i=0$, PP_i is passed vertically downward unchanged. PP_0 is all 0s, and PP_4 is the desired product. The multiplicand is shifted left one position per row by the diagonal signal path.

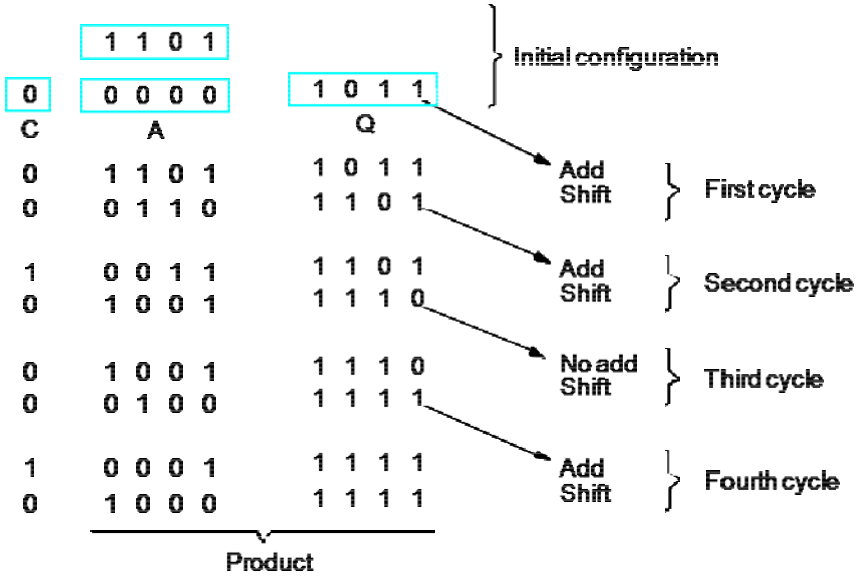
Worst case signal propagation delay path is from the upper right corner of the array to the higher order product bit output at the bottom left corner of the array.

Sequential Circuit Binary multiplier

Registers A and Q combined hold PP_i multiplier bit qi generates the signal Add/Noadd. This signal controls the addition of the multiplicand, M to PP_i to genertae $PP_{(i+1)}$). The product is computed in n cycles. The partial product grows in length by one bit per cycle from the initial vector,PP_0 of n 0s in register A. The carry-out from the adder is stored in flip-flop C, shown at the left end of register A. At the start, the multiplier is loaded into register Q, the multiplicand into register M, and C and A are cleared to 0. At the end of each cycle, C,A and Q are shifted right one bit position to allow for growth of the partial product as the multiplier is shifted out of register Q. Because of this shifting, multiplier bit qi appears at the LSB position Q to generate the Add/Noadd signal at the correct time, starting with q0 during the first cycle, q1 during the second cycle, and so on. After they are used , the multiplier bits are discarded by the right shift operation. Note that the carry-out from the adder is the leftmost bit of $PP_{(i+1)}$, and must be held in the C flip-flop to be shifted right with the contents of A and Q. After n cycles, the high-order half of the product is held in register A and the low order half is in register Q.



(a) Register configuration



(b) Multiplication example

Signed Multiplication

Booth Algorithm

A powerful algorithm for signed -number multiplication is a Booth's algorithm which generates a 2n bit product and treats both positive and negative numbers uniformly. This algorithm suggest that we can reduce the number of operations required for multiplication by representing multiplier as a difference between two numbers.

For example, multiplier 0 0 1 1 1 0(14) can be represented as follows.

```
      0 1 0 0 0 0 (16)
  -   0 0 0 0 1 0 (2)
  --------------------------------
      0 0 1 1 1 0 (14)
```

Therefore, the product can be computed by adding 24 times the multiplicand to the 2s complement of 21 times the multiplicand. In simple notations, we can describe the sequence of required operations be recoding the preceding multiplier as

0 + 1 0 0 -1 0

In general , For Booth's algorithm recoding scheme can be given as
-1 times the shifted multiplicand is selected when moving from 0 to 1,+1 times the shifted multiplicand is selected when moving from 1 to 0, and 0 timesw the shifted muluiplicand is selected for none of the above case,as multiplier is scanned from right to left.

Fast Multiplication -- Booth's Algorithm

The Booth's algorithm serves two purposes:

1. Fast multiplication (when there are consecutive 0's or 1's in the multiplier).
2. Signed multiplication.

$98,765 \times 10,001$ $98,765 \times 9,999$

First consider two decimal multiplications: and . It is obvious that If straight forward multiplication is used, the first one is easier than the second as only two single-digit multiplications are needed for the first while four are needed for the second. However, as we also realize that:

$$98765 \times 10001 = 98765 \times (10000 + 1) = 98765 \times 10000 + 98765$$

$$98765 \times 9999 = 98765 \times (10000 - 1) = 98765 \times 10000 - 98765$$

the two multiplications should be equally easy.

Example 1

If there is a sequence of 0's in the multiplier, the multiplication is easy as all 0's can be skipped.

```
      2  2  ||                          0  1  0  1  1  0
    x 1  7  ||                       x  0  1  0  0  0  1
-------------||-------------------------------------------
      1  5  4||                          0  1  0  1  1  0
+     2  2    ||    +  0  1  0  1  1  0
-------------||-------------------------------------------
      3  7  4||       0  1  0  1  1  1  0  1  1  0
```

Example 2

However, it does not help if there is a sequence of 1's in the multiplier. We have to go through each one of them:

$$
\begin{array}{cccc|cccccccccccc}
 & & 2 & 2 & & & & & & 0 & 1 & 0 & 1 & 1 & 0 \\
 & \times & 1 & 4 & & & & & \times & 0 & 0 & 1 & 1 & 1 & 0 \\
\hline
 & & 8 & 8 & & & & & 0 & 1 & 0 & 1 & 1 & 0 & \\
 & 2 & 2 & & & & & 0 & 1 & 0 & 1 & 1 & 0 & & \\
 & & & & + & 0 & 0 & 1 & 0 & 1 & 1 & 0 & & & \\
\hline
 & 3 & 0 & 8 & & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0
\end{array}
$$

How can we enjoy a sequence of 1's as well as a sequence of 0's? We first Realize that $001110 = 010000 - 000010$, or in general a string of 1's in the multiplier A can be written as:

$$
\begin{array}{ccccccccccc}
 & d & d & \underline{0} & \underline{1} & 1 & \cdots & 1 & \underline{1} & \underline{0} & d & d \\
= & d & d & \underline{1} & \underline{0} & 0 & \cdots & 0 & 0 & 0 & d & d \\
- & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & \underline{1} & \underline{0} & 0 & 0
\end{array}
$$

where d is ``don't care'' (either 0 or 1). If we define the first part of the above as $A_{leftend} = dd10\cdots00dd$ and the second part as $A_{rightend} = 0000\cdots1000$, then the multiplication becomes

$$B \times A = B \times (A_{leftend} - A_{rightend}) = B \times A_{leftend} - B \times A_{rightend}$$

In other words, only the two ends of a string of 1's in the multiplier need to be taken care of. At the left end the multiplicand is added to the partial product, while at the right end the multiplicand is subtracted from the partial product. The above multiplication can therefore be written as:

$$
\begin{array}{ccccccccccc|ccccccccccc}
 & & & & & 0 & 1 & 0 & 1 & 1 & 0 & & & & & & 0 & 1 & 0 & 1 & 1 & 0 \\
 & & & & \times & 0 & \underline{0} & \underline{1} & 1 & \underline{1} & \underline{0} & & & & & \times & 0 & \underline{0} & \underline{1} & 1 & \underline{1} & \underline{0} \\
\hline
 & 0 & 1 & 0 & 1 & 1 & 0 & & & & & & 0 & 1 & 0 & 1 & 1 & 0 & & & & \\
- & & & & 0 & 1 & 0 & 1 & 1 & 0 & & + & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & \\
\hline
 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0
\end{array}
$$

On the right side above the subtraction is carried out by adding 2's complement. We observe that there is a sequence of 1's in the multiplier, only the two ends need to be taken care of, while all 1's in between do not require any operation. The Booth's algorithm for multiplication is based on this observation. To do a $B \times A$ multiplication,

where

- $B = b_{n-1}b_{n-2}\cdots b_1 b_0$ is the multiplicand
- $A = a_{n-1}a_{n-2}\cdots a_1 a_0$ is the multiplier

we check every two consecutive bits in A at a time:

a_i	a_{i-1}	$a_{i-1} - a_i$	Operations
0	0	0	in middle of string of 0. No operation.
1	0	-1	beginning of string of 1. Subtract B from partial product
1	1	0	in middle of string of 1. No operation.
0	1	1	end of string of 1. Add B to partial product

where $i = 0, 1, \cdots, n-1$, and when $i = 0$, $a_{i-1} = a_{-1} \equiv 0$.

Why does it work? What we did can be summarized as the following

$$\begin{aligned} Product &= (a_{-1} - a_0) \times B \times 2^0 + (a_0 - a_1) \times B \times 2^1 + (a_1 - a_2) \times B \times 2^2 + \\ &\quad \cdots + (a_{n-2} - a_{n-1}) \times B \times 2^{n-1} \\ &= B \times [-a_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} a_i \times 2^i] \\ &\stackrel{*}{=} B \times Val(A) \end{aligned}$$

* Recall that the value of a signed-2's complement number (either positive or negative) can be found by:

$$Val(A = a_{n-1}\cdots a_0) = -a_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} a_i \times 2^i$$

Another Example:

$$B = 22 = (0010110)_2$$

Assume $n = 7$bits available. Multiply by $A = -34 = -(0100010)_2$. First represent both operands and their negation in signed 2's complement:

22: 00101110, -22: 1101010
34: 01000010, -34: 1011110

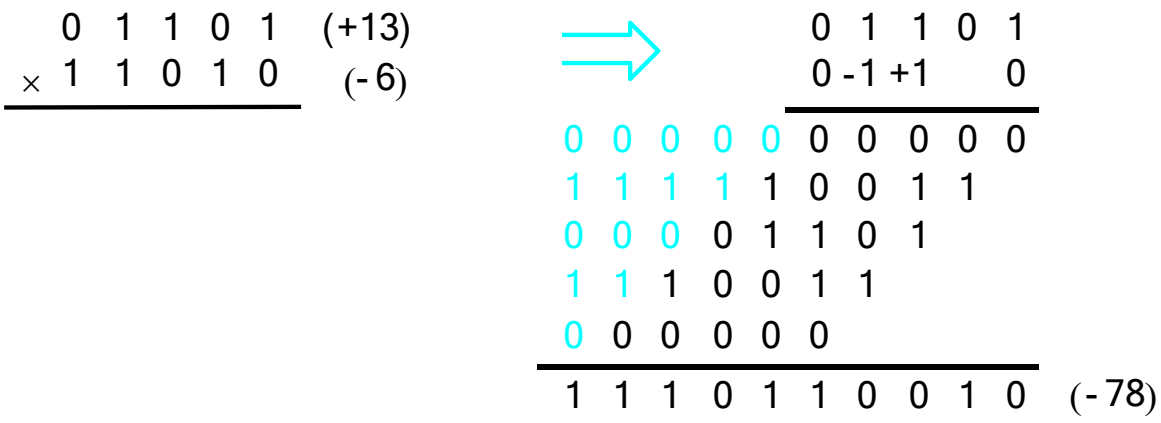
Then carry out the multiplication in the hardware:

q_iq_{i-1}	Action		[M] 0010110			
		[A]	0000000	[Q]	1011110	0
00	right shift		0000000		0101111	0
10	-B	+	1101010			
			1101010		0101111	0
	right shift		1110101		0010111	1
11	right shift		1111010		1001011	1
11	right shift		1111101		0100101	1
11	right shift		1111110		1010010	1
01	+B	+	0010110			
			0010100		1010010	1
	right shift		0001010		0101001	0
10	-B	+	1101010			
			1110100		0101001	0
	right shift		1111010		0010100	1

The upper half of the final result 1111010 0010100 is in register [A] while the lower half is in register [Q]. The product is given in signed 2's complement and its actual value is negative of the 2's complement:

$$B \times A = -\overline{1111010\,0010100} = -0000101\,1101100 = -748_{10}$$

Another Example



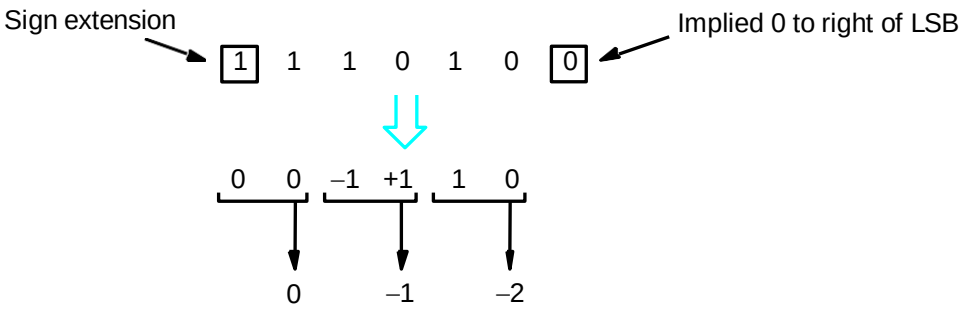
Also note that:

- As the operands are in signed 2's complement form, the *arithmetic shift* is used for the right shifts above, i.e., the MSB bit (sign bit) is always repeated while all other bits are shifted to the right. This guarantees the proper sign extension for both positive and negative values represented in signed 2's complement.
- When the multiplicand is negative represented by signed 2's complement, it needs to be complemented again for subtraction (when the LSB of multiplier is 1 and the extra bit is 0, i.e., the beginning of a string of 1's).

- Best case – a long string of 1's (skipping over 1s)
- Worst case – 0's and 1's are alternating

Bit-Pair Recoding of Multipliers

- Group the booth recoded multiplier bits in pairs, and can be observed, that, the pair (+1, -1) is same to the pair (0, +1), i.e., Instead of adding -1 x M at shift position i with +1 x M at i+1, it can be added with +1 x M at position i.
- Bit-pair recoding halves the maximum number of summands (versions of the multiplicand).



Multiplier bit-pair		Multiplier bit on the right	Multiplicand selected at position *i*
i + 1	*i*	*i* – 1	
0	0	0	0 × M
0	0	1	+ 1 × M
0	1	0	+ 1 × M
0	1	1	+ 2 × M
1	0	0	2 × M
1	0	1	– 1 × M
1	1	0	– 1 × M
1	1	1	0 × M

(b) Table of multiplicand selection decisions

Example

```
    0  1  1  0  1   (+13)
    1  1  0  1  0   (- 6)
  ---------------
```

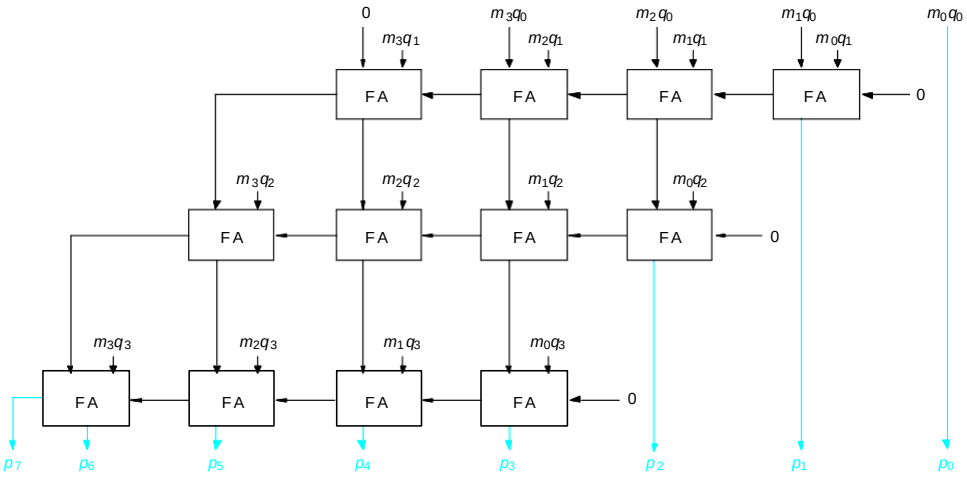
```
                0  1  1  0  1
                0 -1 +1 -1  0
             ---------------
 0  0  0  0  0  0  0  0  0  0
 1  1  1  1  1  0  0  1  1
 0  0  0  0  1  1  0  1
 1  1  1  0  0  1  1
 0  0  0  0  0  0
-----------------------------
 1  1  1  0  1  1  0  0  1  0   (- 78)
```

```
                0  1  1  0  1
                0    -1    -2
             ---------------
 1  1  1  1  1  0  0  1  1  0
 1  1  1  1  0  0  1  1
 0  0  0  0  0  0
-----------------------------
 1  1  1  0  1  1  0  0  1  0   (-78)
```

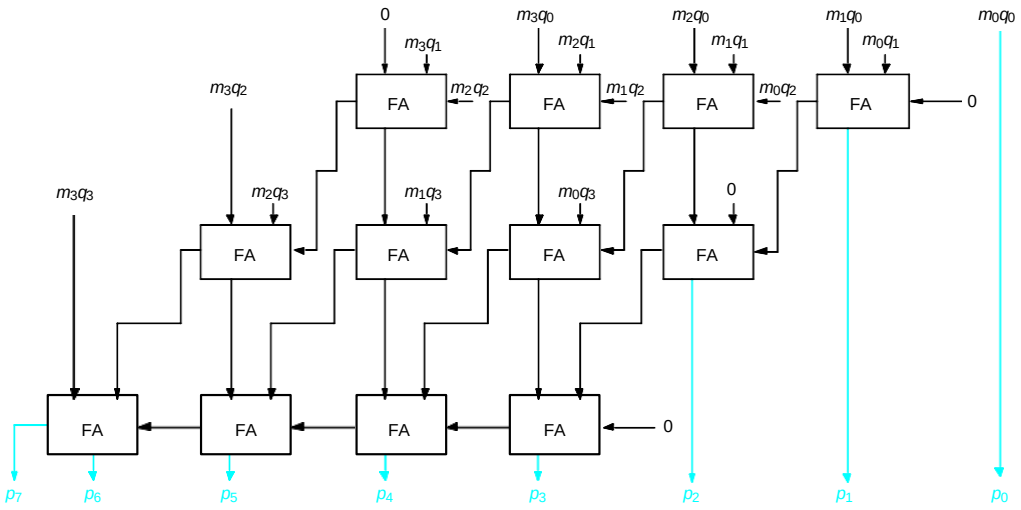
Carry-Save Addition (CSA) of Summands

Carry save addition speeds up the addition process. In CSA, instead of letting the carries ripple along the rows, they can be *saved* and introduced into next row, at correct weighted positions. The full adder is input with three partial bit products in the first row.

- Multiplication requires the addition of several summands.
- CSA speeds up the addition process.
- Consider the array for 4x4 multiplication shown in fig(1).
- First row consisting of just the AND gates that implement the bit products m3q0,m2q0,m1q0 and m0q0 .



(a) Ripple-carry array (Figure 6.6 structure)



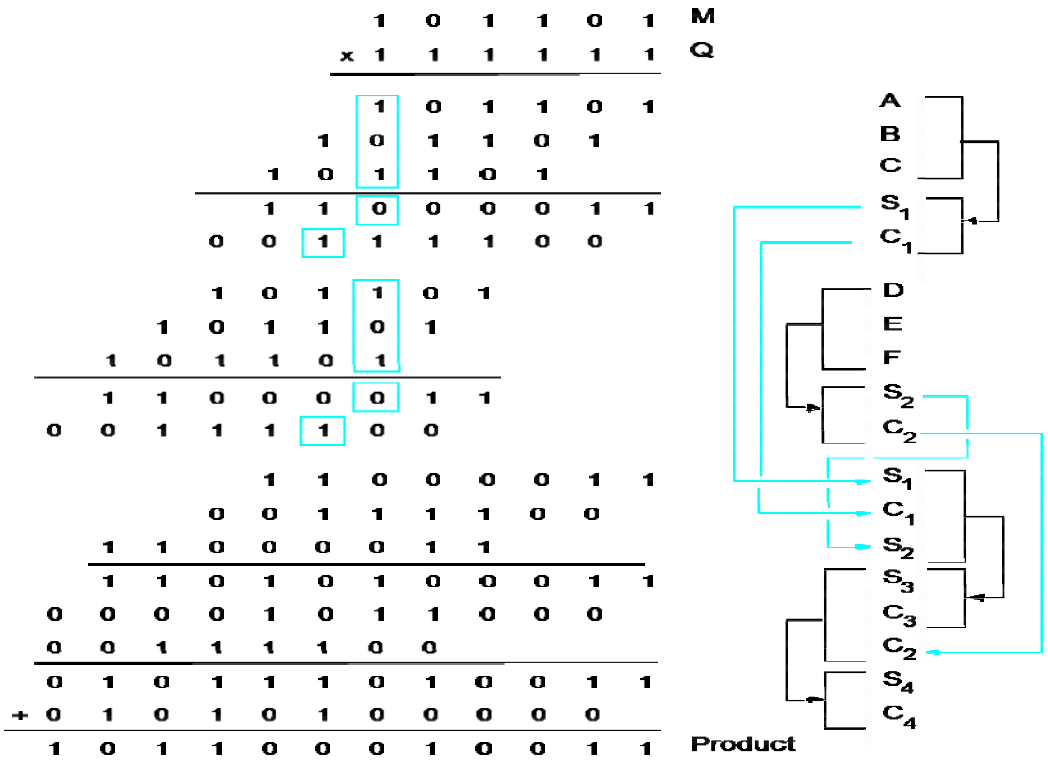
(b) Carry-save array

Figure 6.16. Ripple-carry and carry-save arrays for the multiplication operation M x Q = P for 4-bit operands.

- The delay through the carry-save array is somewhat less than delay through the ripple-carry array. This is because the S and C vector outputs from each row are produced in parallel in one full-adder delay.
- Consider the addition of many summands, we can:
- Group the summands in threes and perform carry-save addition on each of these groups in parallel to generate a set of S and C vectors in one full-adder delay
- Group all of the S and C vectors into threes, and perform carry-save addition on them, generating a further set of S and C vectors in one more full-adder delay
- Continue with this process until there are only two vectors remaining

- They can be added in a RCA or CLA to produce the desired product.

```
                    1 0 1 1 0 1      (45)      M
                x   1 1 1 1 1 1      (63)      Q
              -----------------
                  | 1 0 1 1 0 1 |    A
                1 | 0 1 1 0 1   |    B
              1 0 | 1 1 0 1     |    C
            1 0 1 | 1 0 1       |    D
          1 0 1 1 | 0 1         |    E
        1 0 1 1 0 | 1           |    F
      -------------------------
1 0 1 1 0 0 0 1 0 0 1 1          (2,835)   Product
```



- When the number of summands is large, the time saved is proportionally much greater.

- Delay: AND gate + 2 gate / CSA level + CLA gate delay, Eg., 6 bit number require 15 gate delay, array 6x6 require 6(n-1)-1 = 29 gate D.
- In general CSA takes 1.7 *log*2 k -1.7 levels of CSA to reduce k summands

Integer Division

Manual Division

```
   21
  ----
13) 27
    4
  ----
   14
   13
  ----
    1
```

```
          10101
       ----------
1101) 100010010
      1101
      ------
        10000
         1101
        ------
         1110
          1101
         ------
            1
```

Longhand Division
Steps

- Position the divisor appropriately with respect to the dividend and performs a subtraction.
- If the remainder is zero or positive, a quotient bit of 1 is determined, the remainder is extended by another bit of the dividend, the divisor is repositioned, and another subtraction is performed.
- If the remainder is negative, a quotient bit of 0 is determined, the dividend is restored by adding back the divisor, and the divisor is repositioned for another subtraction.

Restoring Division

- Similar to multiplication circuit
- N-bit positive divisor is loaded into register M and an n-bit positive dividend is loaded into register Q at the start of the operation.
- Register A is set to 0
- After the division operation is complete, the n-bit quotient is in register Q and the remainder is in register A.
- The required subtractions are facilitated by using 2's complement arithmetic.
- The extra bit position at the left end of both A and M accomodates the sign bit during subtraction.
- Shift A and Q left one binary position
- Subtract M from A, and place the answer back in A
- If the sign of A is 1, set q0 to 0 and add M back to A (restore A); otherwise, set q0 to 1

- Repeat these steps *n* times

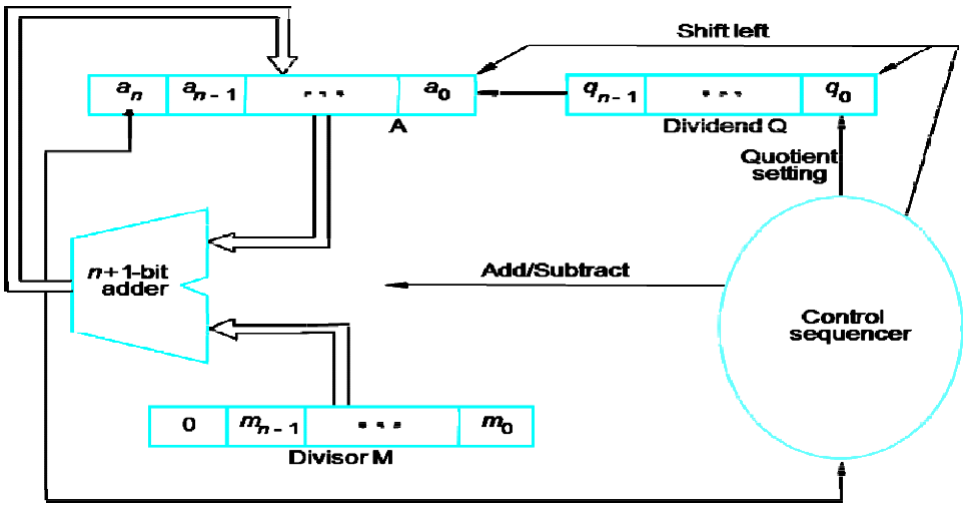


Figure 6.21. Circuit arrangement for binary division.

Example

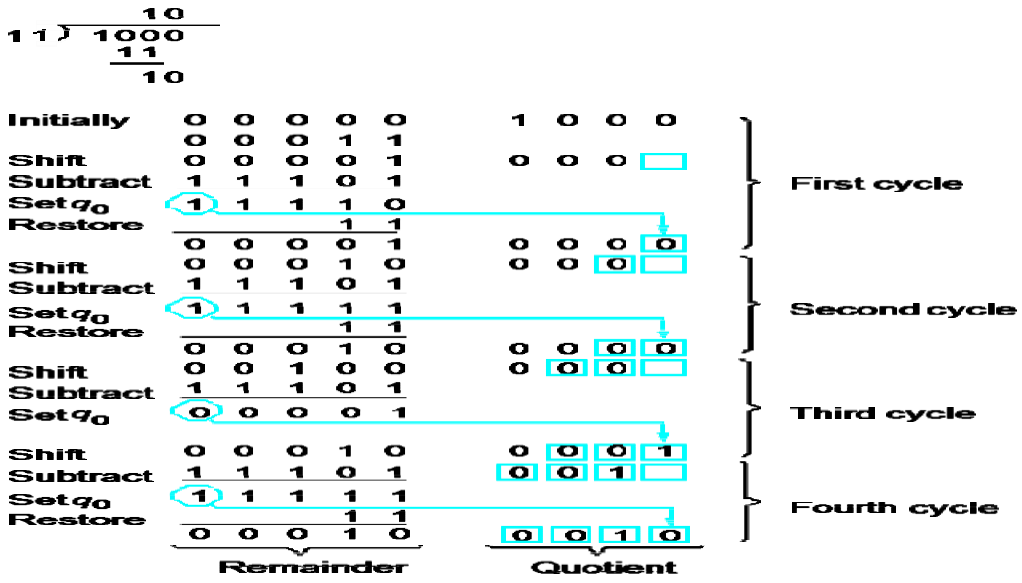


Figure 6.22. A restoring-division example.

Non-restoring Division

- Restoring division algorithm can be improved by avoiding the need for restoring A after an unsuccessful subtraction
- Subtraction is said to be unsuccessful if the result is negative
- If A is positive, we shift left and subtract M that is we perform 2A-M.
- If A is negative, we restore it by performing A+M, and then we shift it left and subtract M.
- This is equivalent to performing 2A+M.
- Q0 is set to 0 or 1 after the correct operation has been performed.

Algorithm for Non Restoring Division

Step 1:(Repeat n times)

- If the sign of A is 0, shift A and Q left one bit position and subtract M from A; Otherwise , shift A and Q left and add M to A.
- Now if the sign of A id 0 set q0 to 1; otherwise , set q0

to 0 Step 2: If the sign of A is 1, add M to A

Example

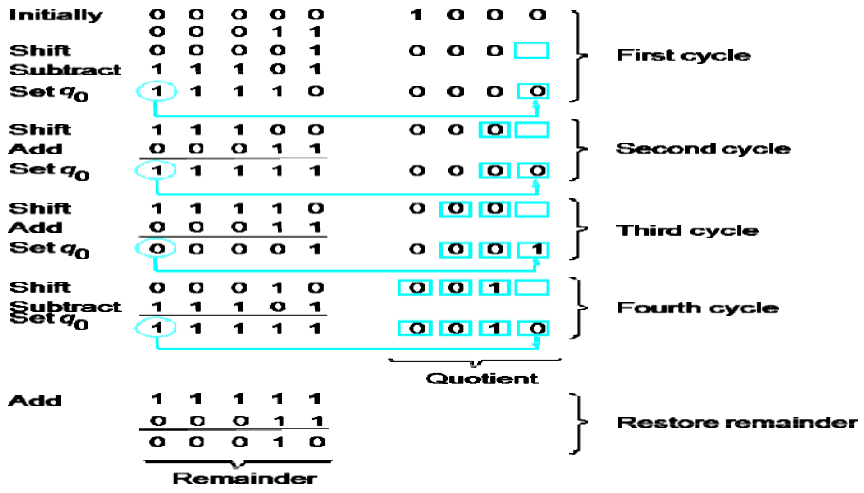


Figure 6.23. A nonrestoring-division example.

Comparision

- Needs restoring of reg A if the result of subtraction is -ve.
- In each cycle content of reg A is first shifted left and then divisor is subtracted from it

- Does not need restoring of remainder
- Slower algorithm

- Does not need restoring
- Needs restoring of remainder if remainder is –ve

- In each cycle the content of reg A is first shifted left and then the divisor is added or subtracted with the content of reg A depending on the sign of A
- Faster algorithm

Floating-Point Numbers and Operations

- So far we have dealt with fixed-point numbers and have considered them as integers.
- Floating-point numbers: the binary point is just to the right of the sign bit.
- In the 2's complement system, the signed value F,represented n-bit binary

  fraction $B=b_0b_{-1}b_{-2}.....b_{-(n-1)}$

  $F(B)= -b_0 x2^0+b_{-1}x2^{-1}+b_{-2}x2^{-2}\cdots\cdots b_{-(n-1)}x2^{-(n-1)}$

- Where the range of F is:$2^{-(n-1)} \leq F \leq 1-2^{-(n-1)}$
- The position of the binary point is variable and is automatically adjusted as computation proceeds.

- If n=32, then the value range is

  approximately $2^{(-31)} \leq F \leq 1-2^{-(31)}$ (1-

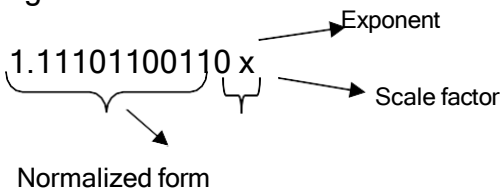
  $2.3283X10^{-10}$)

- But this range is not sufficient to represent fractional numbers,
- To accommodate very large integers and very small fractions, a computer must be able to represent numbers and operate on them in such a way that the position of the binary point is variable and is automatically adjusted as computation proceeds.
- In this case the binary point is said to float, and the numbers are called floating point numbers.
- What are needed to represent a floating-point decimal number?
- It needs three fields
- Sign
- Mantissa (the significant digits)
- Exponent to an implied base (scale factor)

"Normalized" – the decimal point is placed to the right of the first (nonzero) significant digit

- Let us consider the number 111101.1000110 to be represented in floating point format.
- To represent the number in floating point format, first binary point is shifted to right of the first bit and the number is multiplied by the scaling factor to get the same value.
- The number is said to be Normalized form and is given as

111101.1000110
2^5

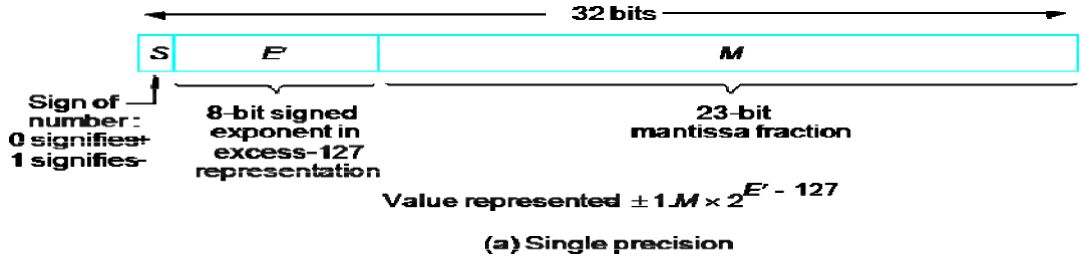


IEEE Standard for Floating-Point Numbers

Think about this number (all digits are decimal): $\pm X_1.X_2X_3X_4X_5X_6X_7\times10^{\pm Y1Y2}$.It is possible to approximate this mantissa precision and scale factor range in a binary representation that occupies 32 bits: 24-bit mantissa (1 sign bit for signed number), 8-bit exponent.

Instead of the signed exponent, E, the value actually stored in the exponent field is an unsigned integer E'=E+127, so called excess-127 format.

Single Precision



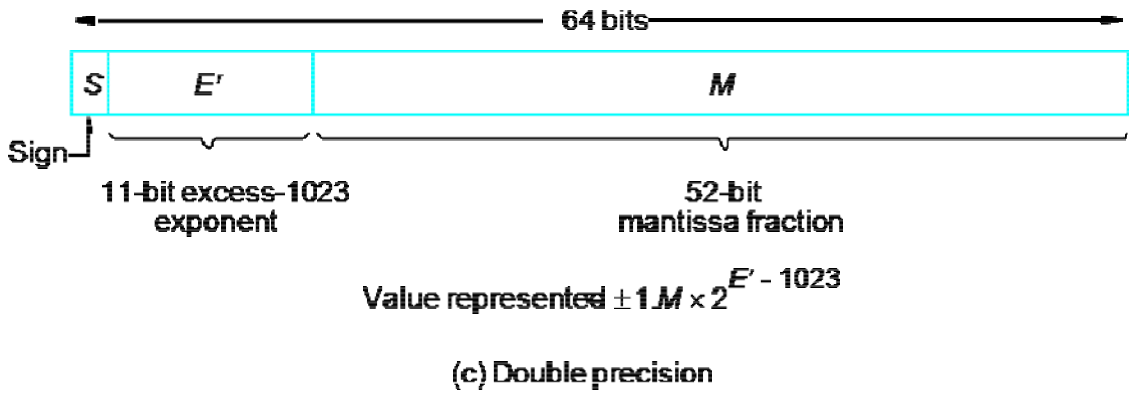
(a) Single precision

Value represented $1.001010\ldots0\times2^{-87}$

(b) Example of a single-precision number

$101000)_2=40_{10}$; 40-127=-87

Double Precision



(c) Double precision

Problem

1) Represent 1259.125_{10} in single precision and double precision formats

- Step 1 :Convert decimal number to binary format

  $1259_{(10)}=10011101011_{(2)}$

  Fractional Part

  $0.125_{(10)}=0.001$
- Binary number = 10011101011+0.001

  =10011101011.001

Step 2:Normalize the number

$10011101011.001=1.0011101011001 \times 2^{10}$

Step3:Single precision format:

For a given number S=0,E=10 and M=0011101011001 Bias for single precision format is = 127 E'=E+127=10+127=137 $_{(10)}$

$=10001001_{(2)}$

- Number in single precision format

0 | 10001001 | 0011101011001…. 0

Sign | Exponent | Mantissa(23 bit)

Step 4:Double precision format:

For a given number S=0,E=10 and M=0011101011001 Bias for double precision format is = 1023 E'=E+1023=10+1023=1033 $_{(10)}$

$=10000001001_{(2)}$

- Number in double precision format is given as

0 | 1000100 | 0011101011001…0

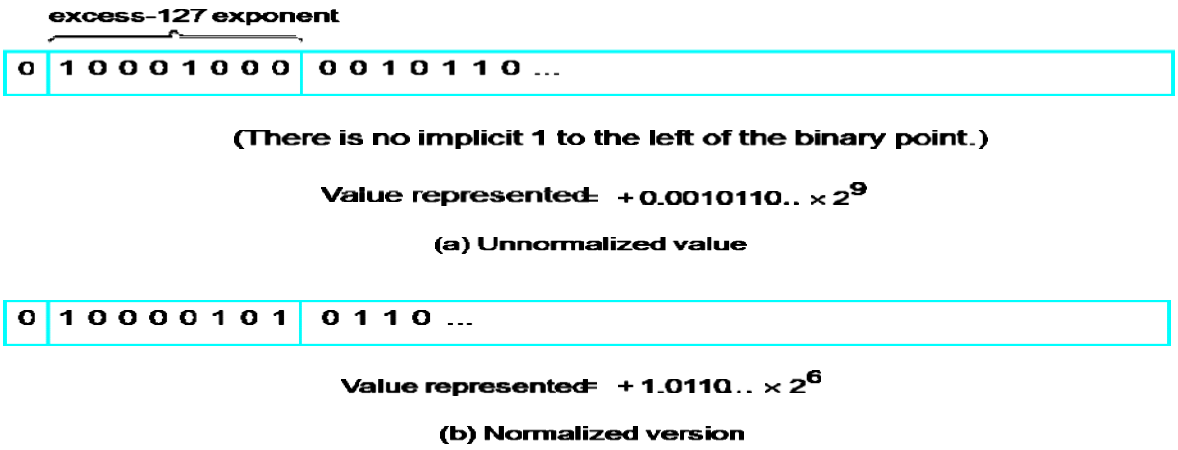
Sign | Exponent | Mantissa(23 bit)

IEEE Standard

- For excess-127 format, $0 \le E' \le 255$. However, 0 and 255 are used to represent special value. So actually $1 \le E' \le 254$. That means $-126 \le E \le 127$.
- Single precision uses 32-bit. The value range is from 2^{-126} to 2^{+127}.
- Double precision used 64-bit. The value range is from 2^{-1022} to 2^{+1023}.

Normalization

- If a number is not normalized, it can always be put in normalized form by shifting the fraction and adjusting the exponent. As computations proceed, a number that does not fall in the representable range of normal numbers might be generated.
- In single precision, it requires an exponent less than -126 (underflow) or greater than +127 (overflow). Both are exceptions that need to be considered.



Special Values

- The end value 0 and 255 are used to represent special values.
- When E'=0 and M=0, the value exact 0 is represented. (±0)
- When E'=255 and M=0, the value ∞ is represented. (± ∞)
- When E'=0 and M≠0, de normal numbers are represented. The value is $\pm 0.M \times 2^{-126}$. (allow for Gradual underflow)
- When E'=255 and M≠0, Not a Number (NaN).
- NaN is the result of performing an invalid operation, such as 0/0 or square root of -1.

Exceptions

- A processor must set exception flags if any of the following occur in performing operations: underflow, overflow, divide by zero, inexact, invalid.
- When exception occurs, the results are set to special values.

Arithmetic Operations on Floating-Point Numbers

Add/Subtract rule

1. Choose the number with the smaller exponent and shift its mantissa right a number of steps equal to the difference in exponents.
2. Set the exponent of the result equal to the larger exponent.
3. Perform addition/subtraction on the mantissas and determine the sign of the result.
4. Normalize the resulting value, if necessary.

Subtraction of floating point numbers

- Similar process is used for subtraction
- Two mantissas are subtracted instead of addition
- Sign of greater mantissa is assigned to the result

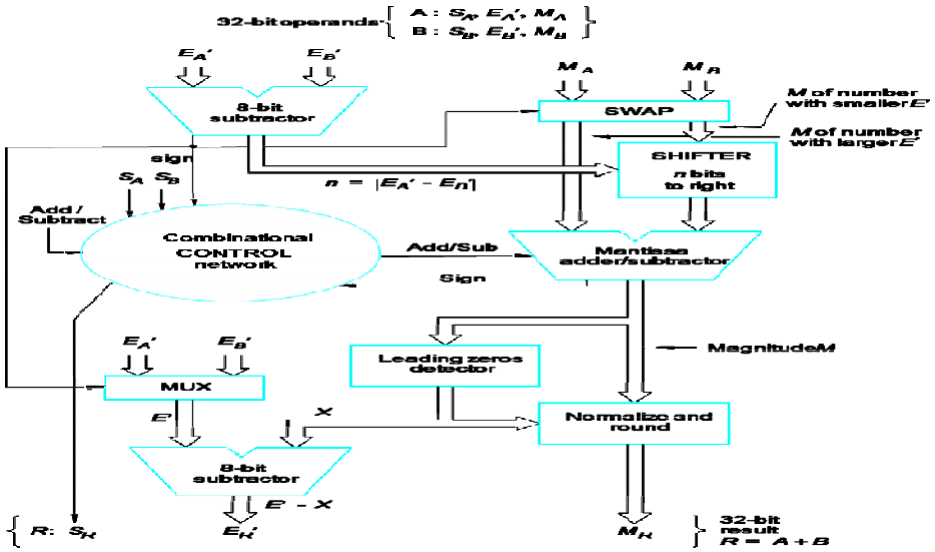


Figure 6.26. Floating-point addition-subtraction unit.

Step 1: Compare the exponent for sign bit using 8bit subtractor Sign is sent to SWAP unit to decide on which number to be sent to SHIFTER unit.
Step2: The exponent of the result is determined in two way multiplexer depending on the sign bit from step1

Step3: Control logic determines whether mantissas are to be added or subtracted. Depending on sign of the operand.
There are many combinations are possible here, that depends on sign bits, exponent values of the operand.
Step4: Normalization of the result depending on the leading zeros, and some special case like 1.xxxxx operands. Where result is 1x.xxx and X = -1, therefore will increase the exponent value.

Example

Add single precision floating point numbers A and B, where A=44900000 H and B = 42A00000H. Solution
Step 1 :Represent numbers in single precision format
A = 0 1000 1001 0010000….0
B = 0 1000 0101 0100000….0
Exponent for A = 1000 1001 =137
Therefore actual exponent = 137-127(Bias) =10
Exponent for B = 1000 0101 = 133
Therefore actual exponent = 133-127(Bias) = 6

With difference 4. Hence its mantissa is shifted right by 4 bits as shown below Step 2:Shift mantissa
Shifted mantissa of B = 0 0 0 0 0 1 0 0…0
Step 3: Add mantissa
Mantissa of A = 00100000…0
Mantissa of B = 00000100…0
Mantissa of result = 00100100…0
As both numbers are positive, sign of the result is positive Result =0100 0100 1001 0010 0…0
=44920000H

Multiply rule

- Add the exponents and subtract 127.
- Multiply the mantissas and determine the sign of the result.
- Normalize the resulting value, if necessary.

Divide rule

- Subtract the exponents and add 127.
- Divide the mantissas and determine the sign of the result.

Normalize the resulting value, if

necessary Guard Bits

- During the intermediate steps, it is important to retain extra bits, often called guard bits, to yield the maximum accuracy in the final results.
- Removing the guard bits in generating a final result requires truncation of the extended mantissa.

Truncation

- Chopping – Remove the guard

bits $0.b_{-1}b_{-2}b_{-3}000$ -- $0.b_{-1}b_{-2}b_{-3}111$à$0.b_{-1}b_{-2}b_{-3}$

Error ranges from 0 to 0.000111.

Chopping is biased because is not

symmetrical about 0, 0 to 1 at LSB.

- Von Neumann Rounding - All 6-bit fractions with $b_{-4}b_{-5}b_6$ not equal to 000 are truncated to to $0.b_{-1}b_{-2}1$
- This truncation is unbiased, error ranges: -1 to +1 at LSB.
- unbiased rounding is better because positive error tend to offset negative errors as the computation proceeds.

- Rounding (A 1 is added to the LSB position of the bits to be retained if there is a 1 in the MSB position of the bits being removed) – unbiased, -½ to +½ at LSB.

$0.b_{-1}b_{-2}b_{-3}1$ .... is rounded to $0.b_{-1}b_{-2}b_{-3}$+ 0.001

- Round to the nearest number or nearest even number in case of a

tie ($0.b_{-1}b_{-2}0100$ -> $0.b_{-1}b_{-2}0$; $0.b_{-1}b_{-2}1100$ -> $0.b_1b_2$1+0.001)

- Best accuracy
- Most difficult to implement

Addition and Subtraction

Floating point addition is analogous to addition using scientific notation. For example, to add 2.25x 10^{0} to 1.340625x 10^{2} :

1. Shift the decimal point of the smaller number to the left until the exponents are equal. Thus, the first number becomes .0225x 10^2.
2. Add the numbers with decimal points aligned:

```
      0.0225      x 10^2
+     1.340625    x 10^2
-------------------------
      1.362125    x 10^2
```

3. Normalize the result.

Once the decimal points are aligned, the addition can be performed by ignoring the decimal point and using integer addition.

The addition of two IEEE FPS numbers is performed in a similar manner. The number 2.25 in IEEE FPS is:

```
S       E                       M
0   1000 0000   (1) 001 0000 0000 0000 0000 0000
```

The number 134.0625 in IEEE FPS is:

```
S       E                       M
0   1000 0110   (1) 000 0110 0001 0000 0000 0000
```

1. To align the binary points, the smaller exponent is incremented and the mantissa is shifted right until the exponents are equal. Thus, 2.25 becomes:

```
S       E                       M
0   1000 0110   (0) 000 0010 0100 0000 0000 0000
```

2. The mantissas are added using integer addition:

```
      S       E                       M
      0   1000 0110   (0) 000 0010 0100 0000 0000 0000
+     0   1000 0110   (1) 000 0110 0001 0000 0000 0000
-------------------------------------------------------
      0   1000 0110   (1) 000 1000 0101 0000 0000 0000
```

3. The result is already in normal form. If the sum overflows the position of the hidden bit, then the mantissa must be shifted one bit to the right and the exponent incremented. The mantissa is always less than 2, so the hidden bits can sum to no more than 3 (11).

The exponents can be positive or negative with no change in the algorithm. A smaller exponent means more negative. In the bias-127 representation, the smaller exponent has the smaller value for E, the unsigned interpretation.

An important case occurs when the numbers differ widely in magnitude. If the exponents differ by more than 24, the smaller number will be shifted right entirely out of the mantissa field, producing a zero mantissa. The sum will then equal the larger number. Such *truncation errors* occur when the numbers differ by a factor of more than 2^{24}, which is approximately 1.6×10^{7}. The precision of IEEE single precision floating point arithmetic is approximately 7 decimal digits.

Negative mantissas are handled by first converting to 2's complement and then performing the addition. After the addition is performed, the result is converted back to sign-magnitude form.

When adding numbers of opposite sign, cancellation may occur, resulting in a sum which is arbitrarily small, or even zero if the numbers are equal in magnitude. Normalization in this case may require shifting by the total number of bits in the mantissa, resulting in a large loss of accuracy.

When the mantissa of the sum is zero, no amount of shifting will produce a 1 in the hidden bit. This case must be detected in the normalization step and the result set to the representation for 0, E = M = 0. This result does not mean the numbers are equal; only that their difference is smaller than the precision of the floating point representation.

Floating point subtraction is achieved simply by inverting the sign bit and performing addition of signed mantissas as outlined above.

Multiplication

The multiplication of two floating point numbers is analogous to multiplication in scientific notation. For example, to multiply 1.8×10^{1} times 9.5×10^{0}:

1. Perform unsigned integer multiplication of the mantissas. The decimal point in the sum is positioned so that the number of decimal places equals the sum of the number of decimal places in the numbers.

```
  1.8
x 9.5
-----
17.10
```

2. Add the exponents:

```
  1
+ 0
---
 1
```

3. Normalize the result:

$$17.10 \times 10^{1} = 1.710 \times 10^{2}$$

10. Set the sign of the result.

The multiplication of two IEEE FPS numbers is performed similarly. The number 18.0 in IEEE FPS format is:

```
S       E                          M
0   1000 0011   (1) 001 0000 0000 0000 0000 0000
```

The number 9.5 in IEEE FPS format is:

```
S       E                          M
0   1000 0010   (1) 001 1000 0000 0000 0000 0000
```

1. The product of the 24 bit mantissas produces a 48 bit result with 46 bits to the right of the binary point:

   01.010101[illegible]...0000

   Truncated to 24 bits with the hidden bit in (), the mantissa is:

   (1).010101100000000000000000.

2. The biased-127 exponents are added. Addition in biased-127 representation can be performed by 2's complement with an additional bias of -127 since:

   $$E = (e_1 + e_2) + 127 = (e_1 + 127) + (e_2 + 127) - 127 = (E_1 + E_2) - 127$$

   The sum of the exponents

   ```
       is: E
    1000 0011 (4)
   + 1000 0010 (3)
   ---------------
    0000 0101
   + 1000 0001 (-127)
   ---------------
    1000 0110 (+7)
   ```

3. The mantissa is already in normal form. If the position of the hidden bit overflows, the mantissa must be shifted right and the exponent incremented.
4. The sign of the result is the xor of the sign bits of the two numbers.

When the fields are assembled in IEEE FPS format, the result is:

```
S       E                          M
0   1000 0110   (1) 010 1011 0000 0000 0000 0000
```

Rounding occurs in floating point multiplication when the mantissa of the product is reduced from 48 bits to 24 bits. The least significant 24 bits are discarded.

Overflow occurs when the sum of the exponents exceeds 127, the largest value which is defined in bias-127 exponent representation. When this occurs, the exponent is set to 128 (E = 255) and the mantissa is set to zero indicating + or - infinity.

Underflow occurs when the sum of the exponents is more negative than -126, the most negative value which is defined in bias-127 exponent representation. When this occurs, the exponent is set to -127 (E = 0). If M = 0, the number is exactly zero.

If M is not zero, then a *denormalized* number is indicated which has an exponent of -127 and a hidden bit of 0. The smallest such number which is not zero is 2^{-149}. This number retains only a single bit of precision in the rightmost bit of the mantissa.