5.7. CONTROL UNIT

The function of control unit is to generate relevant timing and control signals to all operations in the computer. It controls the flow of data between the processor and memory and peripherals

Functions of Control Unit

- The control unit directs the entire computer system to carry out stored program instructions.
- The control unit must communicate with both the arithmetic logic unit (ALU) and main memory.
- The control unit instructs the arithmetic logic unit that which logical or arithmetic operation is to be performed.
- The control unit co-ordinates the activities of the other two units as well as all peripherals and auxiliary storage devices linked to the computer.

Design of a Control Unit

Control unit generates control signals using one of the two organizations:

- Hardwired Control Unit
- Micro-programmed Control Unit

5.7.1. Hardwired Control unit

- It is implemented as logic circuits (gates, flip-flops, decoders etc.) in the hardware.
- This organization is very complicated if we have a large control unit.
- In this organization, if the design has to be modified or changed, requires changes in the wiring among the various components. Thus the modification of all the combinational circuits may be very difficult.

5.7.2. Micro programmed Control Unit

- A micro-programmed control unit is implemented using programming approach. A sequence of microoperations is carried out by executing a program consisting of micro-instructions.
- Micro-program, consisting of micro-instructions is stored in the control memory of the control unit.
- Execution of a micro-instruction is responsible for generation of a set of control signals.

5.8. HARDWIRED CONTROL UNIT

In the hardwired control, the control units use fixed logic circuits to interpret instructions and generate control signals from them. Figure 5.10 shows the typical hardwired control unit. Here, the fixed logic circuit block includes combinational circuit that generates the required control outputs for decoding and encoding functions. By separating the decoding and encoding functions, we can draw more detail block diagram for hardwired control unit as shown in the Fig. 5.10.

The instruction decoder decodes the instruction loaded in the IR. If IR is an 8-bit register instruction decoder generates 2^8, *i.e.* 256 lines; one for each instruction. According to code in the IR, only one line among all output lines of decoder goes high *i.e.*, set to 1 and all other lines are set to 0. The step decoder provides a separate signal line for each step, or time slot, in a control sequence. The encoder gets in the input from instruction decoder, step decoder, external inputs and condition codes. it uses all these inputs to generate the individual control signals. After execution of each instruction end signal is generated which resets control step counter and make it ready for generation of control step for next instruction.

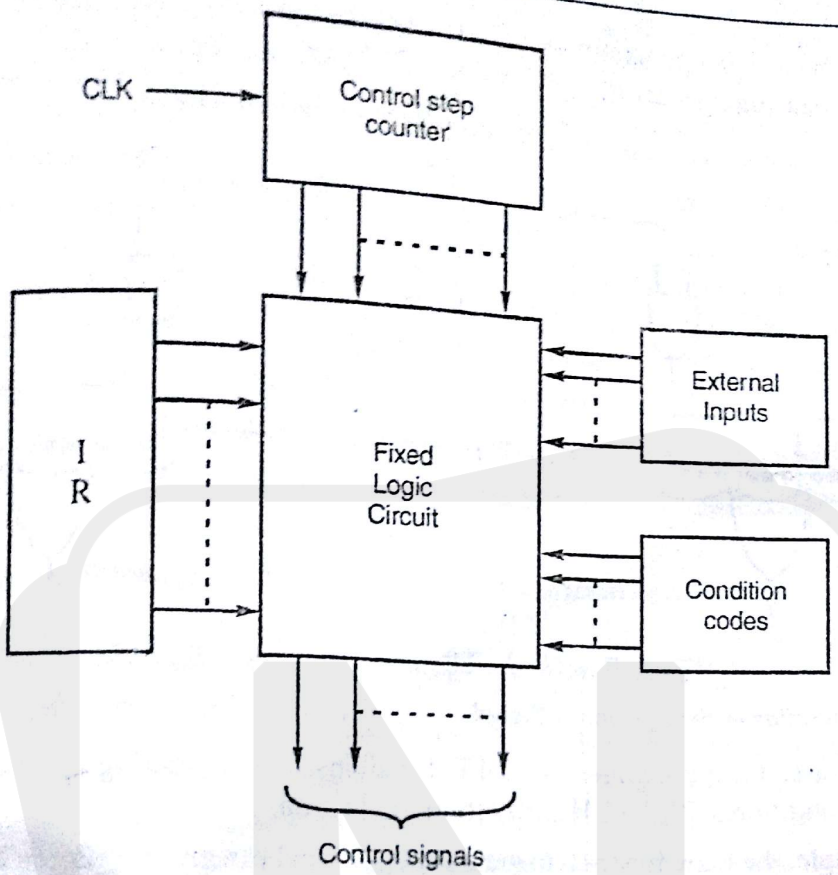


Fig. 5.10. Typical Hardwired Control Unit

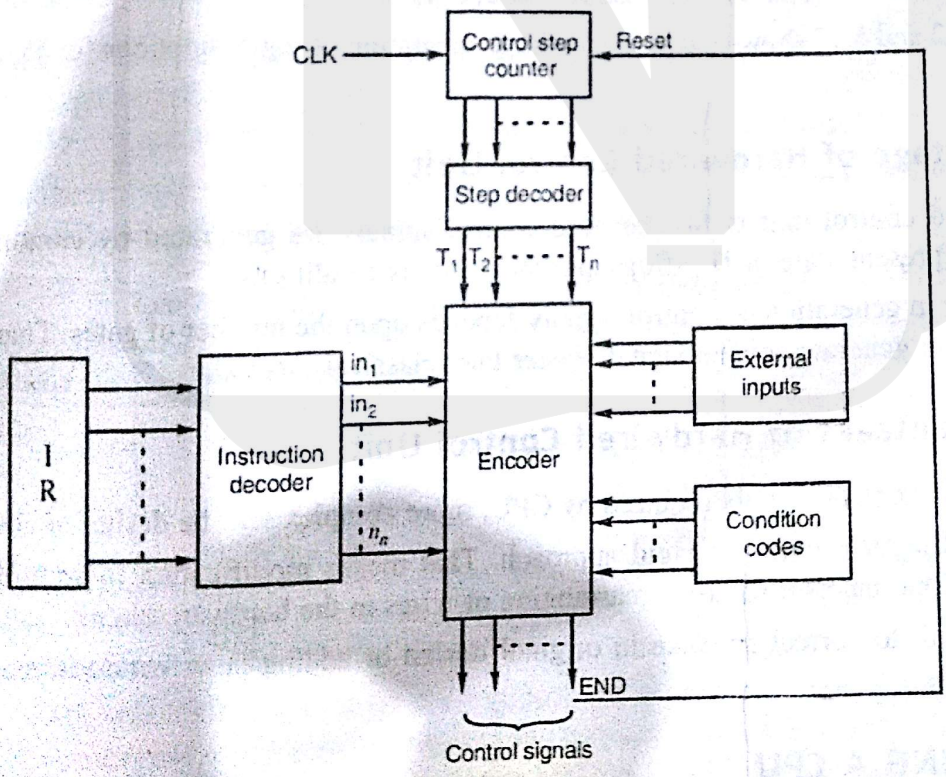


Fig. 5.11. Detail Block Diagram for Hardwired control unit

Let us see how the encoder generates signal for single bus processor organisation shown in Fig. 5.12. The encoder circuit implements the following logic function to generate Y_{in}.

$$Y_{in} = T_1 + T_5 . ADD + T_4 BRANCH + \ldots$$

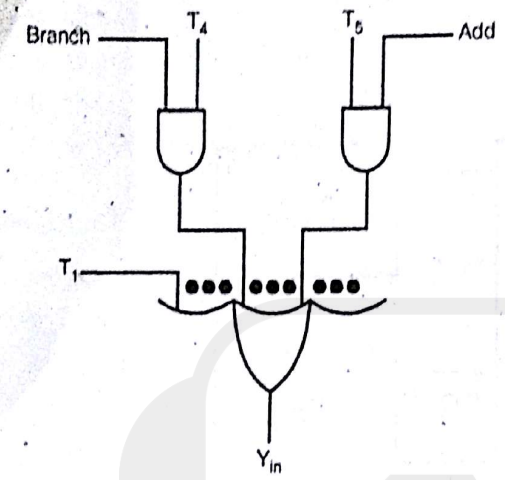


Fig. 5.12. Generation of the Y_{in} Control Signal

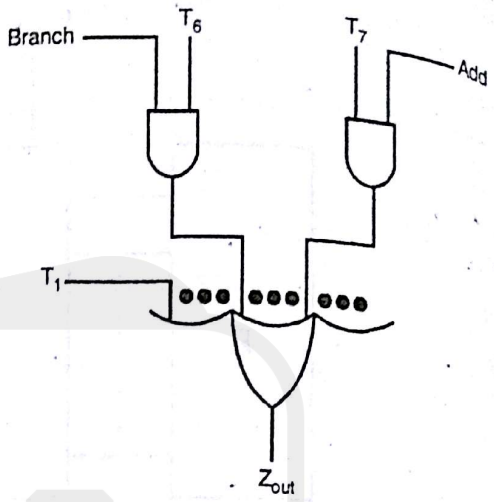


Fig. 5.13. Generation of the Z_{out} Control Signal

The Y_{in}, signal is asserted during time interval T_1 for all instructions, during T_5 for an ADD instruction, during T_4 for an unconditional BRANCH instruction, and so on.

As another example, the logic function to generate Z_{out} signal can given by $Z_{out} = T_2 + T_7 . ADD + T_6 . BRANCH + \ldots$

The Z_{out} signal is asserted during time interval T_2 of all instructions, during T_7 for an ADD instruction, during T_6 for an unconditional branch instruction, and so on.

The Fig. 5.12 and 5.13 shows the hardware implementation of logic functions for Y_{in} and Z_{out} control signals.

5.8.1. Advantage of Hardwired Control Unit

1. Hardwired control unit is fast because control signals are generated by combinational circuit based on present state of flip-flops and other inputs conditions.
2. The delay in generation of control signals depends upon the number of gates. That means one hot method can generate control signals faster than classical method.

5.8.2. Disadvantages of Hardwired Control Unit

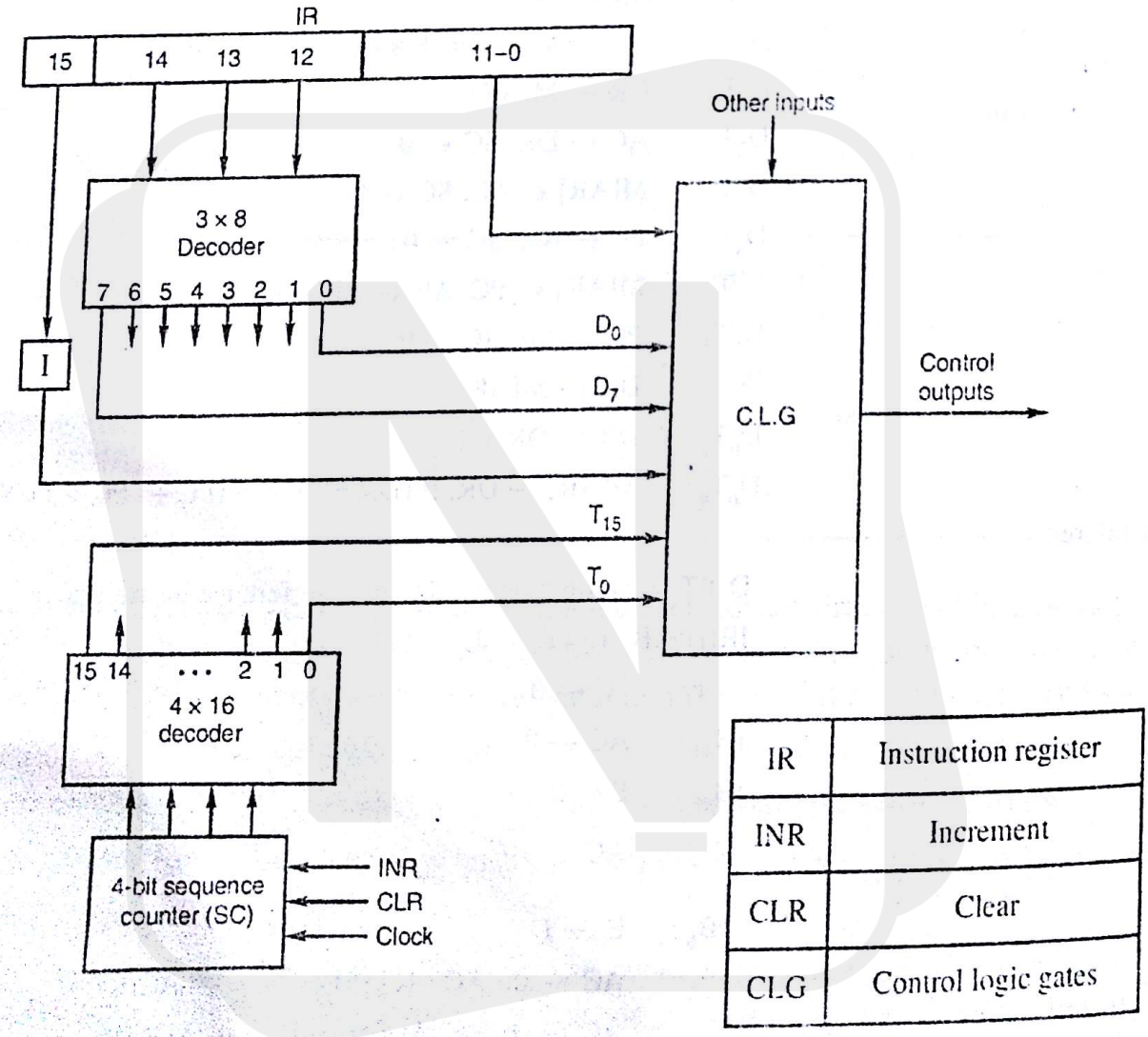
1. More is the control signals required by CPU, more complex will be design of control unit.
2. Hardwired control unit is a rigid approach. That means modification in control signal is very difficult. That means it requires rearranging of wires in the hardware circuit.
3. It is difficult to correct mistake in original design or adding new feature in existing design of control unit.

5.9. SPECIFYING A CPU

The first step in designing a CPU is to determine its applicatons. We don't need anything as complicated as Itanium microprocessor to control a microwave oven; a simple 4-bit processor would be

5.4. TIMING AND CONTROL

The timing of all registers in a basic computer system is controlled by a master clock generator. In the computer system the clock pulses are applied to all flip-flops and registers in the system, including the



IR	Instruction register
INR	Increment
CLR	Clear
CLG	Control logic gates

Fig. 5.5. Control Unit of Basic Computer

flip-flops in the control unit. The share of a register is changed if the register is enabled by a control signal. Basically the control signals are generated in the control unit and provides the control inputs for the multiplexers in the common bus, processor register and microoperations for the accumulator.

Hardwired control and microprogrammed control are two major types of control organization. The block diagram of the control unit is shown in Fig. 5.5.

The block diagram of the control unit is shown in Fig. 5.5. It consists of two decoders, a sequence counter, and a number of control logic gates. An instruction read from memory is placed in the instruction register (IR). The instruction register is shown again in Fig. 5.5 where it is divided into three parts : the I bit, the operation code, and bits 0 through 11. The operation code in bits 12 through 14 are decoded with a 3 × 8 decoder. The eight outputs of the decoder are designated by the symbols D_0 through D_7. The subscripted decimal number is equivalent to the binary value of the corresponding operation code. Bit 15 of the instruction is transferred to a flip-flop designated by the symbol I. Bits 0 through 11 are applied to the control logic gates. The 4-bit sequence counter can count in binary from 0 through 15. The outputs of the counter are decoded into 16 timing signals T_0 through T_{15}.

The sequence counter SC can be incremented or cleared synchronously. Most of the time, the counter is incremented to provide the sequence of timing signals out of the 4 × 16 decoder. Once in a while, the counter is cleared to 0, causing the next active timing signal to be.

For example:

Consider the case where SC is incremented to provide timing signals T_0, T_1, T_2, T_3, and T_4 in sequence. At time T_4, SC is cleared to 0 if decoder output D_3 is active. This is expressed symbolically by the statement:

$$D_3 T_4 : SC \leftarrow 0$$

The timing diagram shows the time relationship of the control signals.

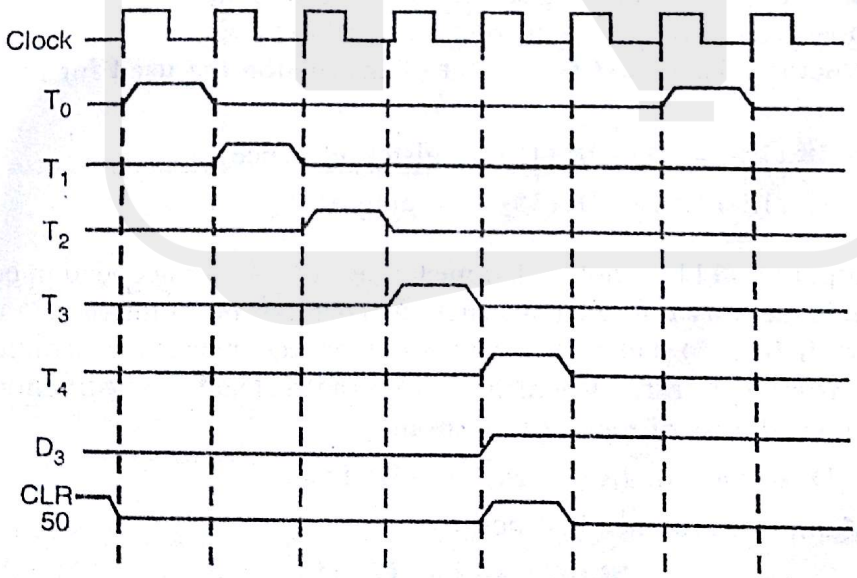


Fig. 5.6. Timing diagram for SC ← 0

5.10.7. Designing the Control Unit Using Hardwired Control

At this point, it is possible for the CPu to perform every operation necessary to fetch, decode and execute the entire instruction set. The next task is to design the circuitry to generate the control signals to cause the operations to occur in the proper sequence. This is **control unit** of the CPU.

There are two primary methodologies for designing control units. Hardwired control uses sequential and combinational logic to generate control signals, whereas microsequenced control uses a lookup memory to output the control signals. Each methodology has several design variants.

This Very simple CPU required only a very simple control unit. The simplest control unit has three components: a counter, which contains the current state; a decoder, which takes the current state and generates individual signals for each state; and some combinational logic to take the individual state signals and generate the control signals for each component, as well as the signals to control the counter. These signals cause the control unit to traverse the states in the proper order. A generic version of this type of hardwired control unit is shown in Fig. 5.20.

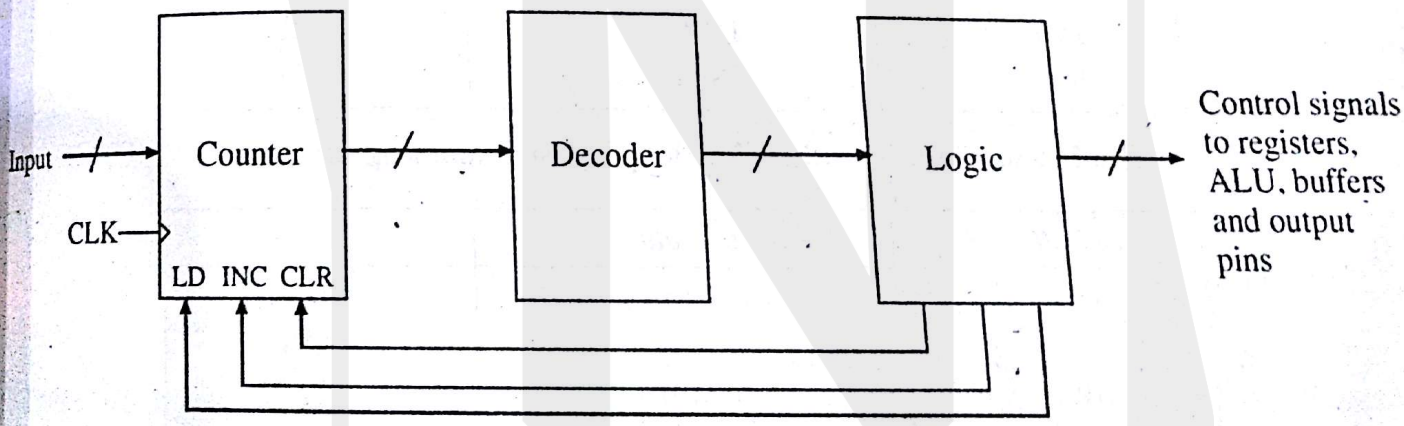


Fig. 5.20. Generic hardwired control unit.

For this CPU, there are a total of 9 states. Therefore, a 4-bit counter and a 4-to-16 bit decoder are needed. Seven of the outputs of the decoder will not be used.

The first task is to determine how best to assign states to the outputs of the decoder, and thus values in the counter. The following guidelines may help.

1. *Assign FETCH1 to counter value 0 and use the CLR input of the counter to reach this state.* Looking at the state diagram for this CPU, we see that every state except FETCH1 can only be reached from one other state. FETCH1 is reached from four states, the last state of each execute routine. By allocating FETCH1 to counter value 0, these four branches can be realized by asserting the CLR signal of the counter, which minimizes the amount of digital logic needed to design the control unit.
2. *Assign sequential states to sequential counter values and use the INC input of the counter to traverse these states.* If this is done, the control unit can traverse these sequential states by asserting the INC signal of the counter, which also reduces the digital logic needed in the control unit. This CPU would assign FETCH2 to counter value 1 and FETCH3 to counter value 2. It would also assign ADD1 and ADD2 to consecutive counter values, as well as AND1 and AND2.
3. *Assign the first state of each execute routine based on the instruction opcodes and the maximum number of states in the execute routines. Use the opcodes to generate the data input to the counter and the LD input of the counter to reach the proper execute routine.* This point squarely addresses the implementation of instruction decoding. Essentially, it implements a mapping of the opcode to the execute routine for that instruction. It occurs exactly once in this and all CPUs, at the last state of the fetch cycle.

To load in the address of the proper execute routine, the control unit must do two things. First, it must place the address of the first state of the proper execute routine on the data inputs of the counter. Second, it must assert the LD signal of the counter. The LD signal is easy; it is directly driven by the last state of the fetch cycle, FETCH3 for this CPU. The difficulty comes in allocating counter values to the states.

Toward that end, consider the list of instructions, their first states, and the value in register IR for those instructions, as shown in Table 5.5.

Table 5.5. Instructions, first states, and opcodes for the very simple CPU.

Instruction	*First State*	*IR*
ADD	ADD1	00
AND	AND1	01
JMP	JMP1	10
INC	INC1	11

Table 5.6. Counter values for the proposed mapping function.

IR[1...0]	*Counter Value*	*State*
00	1000 (8)	ADD1
01	1001 (9)	AND1
10	1010 (10)	JMP1
11	1011 (11)	INC1

Although this would get to the proper execute routine, it causes a problem. Since state ADD1 has a counter value of 8, and state AND1 has a counter value of 9, what value should we assign to ADD2 and how would it be accessed from ADD1 ? This could be done by incorporating additional logic, but this is not the best solution for the design.

Looking at the state diagram for this CPU, we see that no execute routine contains more than two states. As long as the first states of the execute routines have counter values at least two apart, it is possible to store the execute routines in sequential locations. This is accomplished by using the mapping function 1 IR[1...0]0, which results in counter values of 8, 10, 12 and 14 for ADD1, AND1, JMP1 and INC1 respectively. To assign the eecute routines to consecutive values, we assign ADD2 to counter value 9 and AND2 to counter value 11.

Now that we have decided which decoder output is assigned to each state, we can use these signals to generate the **control signals** for the counter of the control unit and for the components of the rest of the CPU. For the counter, we mut generate the INC, CLR and LD signals. INC is asserted when the control unit is traversing sequential states, during FETCH1, FETCH2, ADD1, and AND1. CLR is asserted at the end of each execute cycle to return to the fetch cycle; this happens during ADD2, AND2, JMP1 and INC1. Finally, as noted earlier, LD is asserted at the end of the fetch cycle during state FETCH3. Note that each state of the CPU's state diagram drives exactly one of these three control signals. The circuit diagram for the control unit at this point is shown in Fig. 5.21.

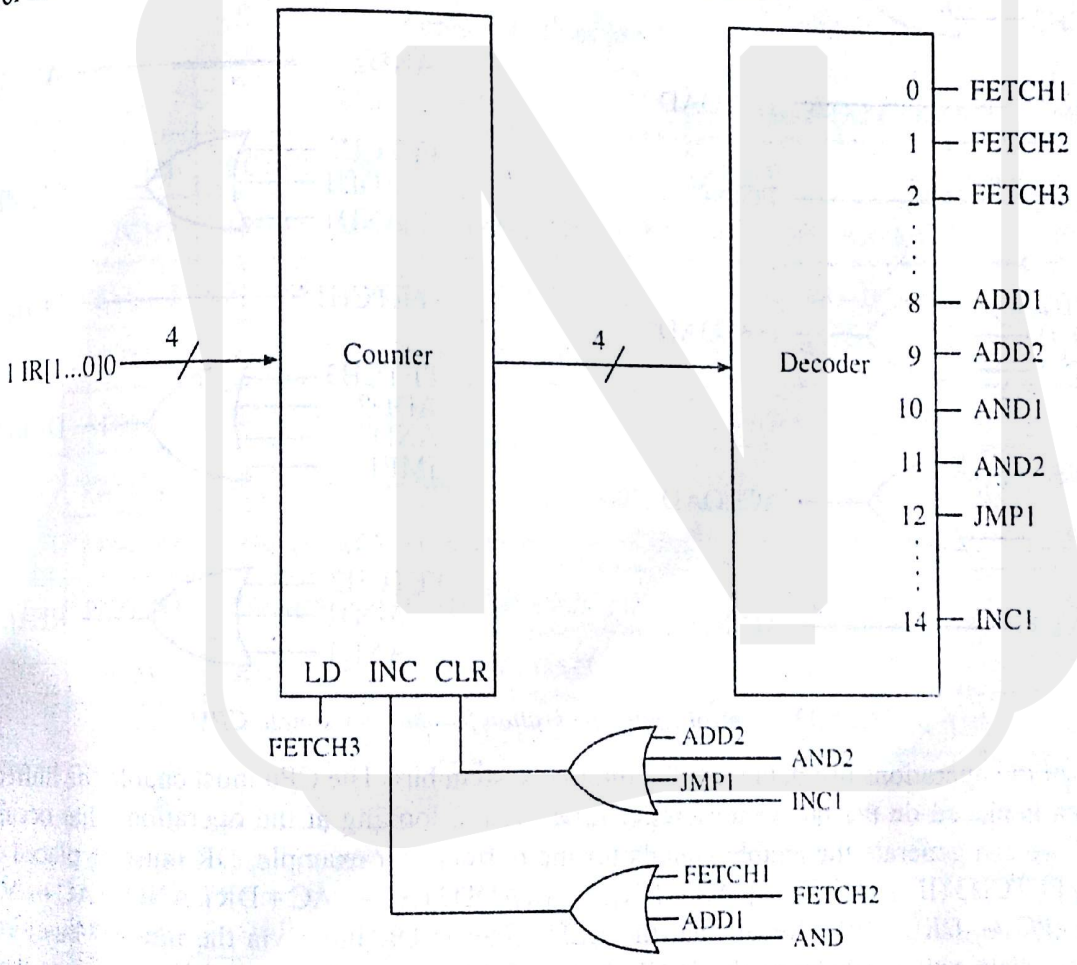


Fig. 5.21. Hardwired control unit for the very simple CPU

These state signals are also combined to create the control signals for AR, PC, DR, IR, M, the ALU and the buffers. First consider register AR. It is loaded during states FETCH1 (AR ← PC) and FETCH3 (AR ← DR[5...0]). By logically ORing these two state signals together, the CPu generates the LD signal for AR. It doesn't matter which value is to be loaded into AR, at least as far as the LD signal is concerned. When the designers create the control signals for the buffers, they will ensure that the proper data is placed on the bus and made available to AR. Following this procedure, we create the following control signals for PC, DR, AC and IR.

PCLOAD = JMP1

PCINC = FETCH2

DRLOAD = FETCH1 ∨ ADD1 ∨ AND1

ACLOAD = ADD2 ∨ AND2

ACINC = INC1

IRLOAD = FETCH3

The ALU has one control input, ALUSEL. When ALUSEL = 0, the output of the ALU is the arithmetic sum of its two inputs; if ALUSEL = 1, the output is the logical AND of its input. Setting ALUSEL = AND2 routes the correct data from the ALU to AC when the CPU is executing an ADD or AND instruction. At other times, during the fetch cycle and the other execute cycles, the ALU is still outputting a value to AC. However, since AC does not load this value, the value output by the ALU does not cause any problems.

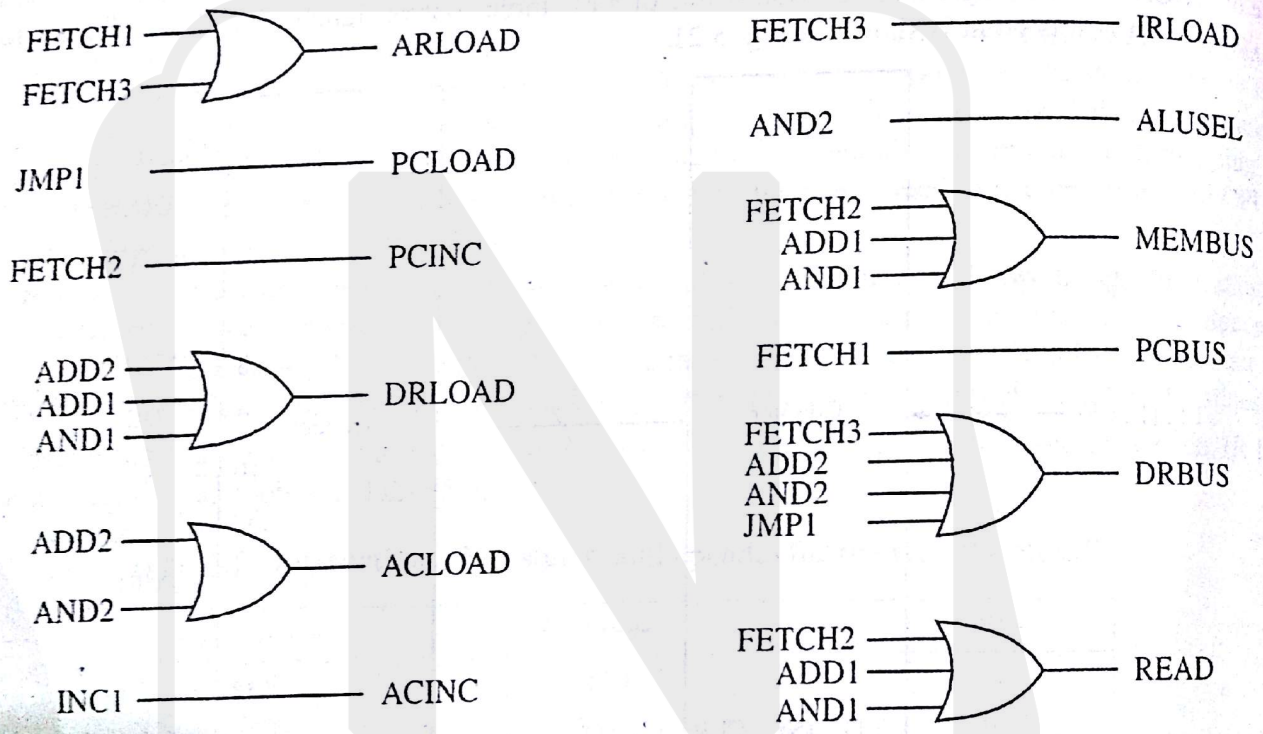


Fig. 5.22. Control signal generation for the very simple CPU

Many of the operations use data from the internal system bus. The CPu must enable the buffers so the correct data is placed on the bus at the proper time. Again, looking at the operations that occur during each state, we can generate the enable signals for the buffers. For example, DR must be placed onto the bus during FETCH3 (IR ← DR[7...6], AR←DR[5...0]), ADD2 (AC ← AC + DR), AND2 (AC ← AC ∧ DR) and JMP1 (PC ← DR[5...0]). (Recall that the ALU receives DR input via the internal bus.) Logically ORing these state values produces the DRBUS signal. This procedure is used to generate the enable signals for the other buffers as well:

MEMBUS = FETCH2 ∨ ADD1 ∨ AND1

PCBUS = FETCH1

Finally, the control unit must generate a READ signal, which is output from the CPU. This signal causes memory to output its data value. This occurs when memory is read during states FETCH2, ADD1 and AND1, so READ can be set as follows:

READ = FETCH2 ∨ ADD1 ∨ AND1

From this ... the adder and Logic circuit.

5.12. MICROPROGRAMMED CONTROL UNIT

A **micro-programmed control unit** is implemented using programming approach. A sequence of microoperations are carried out by executing a program consisting of micro-instructions.

Micro-program, consisting of micro-instructions is stored in the control memory of the control unit. Execution of a micro-instruction is responsible for generation of a set of control signals.

A **micro-instruction** consists of:

- One or more micro-operations to be executed.
- Address of next microinstruction to be executed.

Micro-Operations: The operations performed on the data stored inside the registers are called micro-operations.

Micro-Programs: Microprogramming is the concept for generating control signals using programs. These programs are called micro-programs.

Micro-Instructions: The instructions that make micro-program are called micro-instructions.

Micro-Code: Micro-program is a group of microinstructions. The micro-program can also be termed as micro-code.

Control Memory: Micro-programs are stored in the read only memory (ROM). That memory is called control memory.

5.12.1. Control Memory

Control Memory means a memory in which control signals are stored. Control signals are in the form of micro-operations. Number of control signals in a given system is finite. When control signal are generated by hardware then they are called hard-wired control signal. Control signals can also be generated by micro-programming.

The simplest way to represent a control signal is by a single binary bit. If binary bit is 1 then it means the corresponding control signal will be executed. If binary bit is 0 then it means the corresponding control signal is not executed. The limitations of using only one bit is that only two different states can be represented by it.

If we use group of bits (bus-organised system) then many combination can be made. Each combination represent a unique control signal.

If we have two binary bits then four different control signals can be made. Similarly If we have n binary bits then 2^n different control signals can be made

Control memory is available in the form of read only memory (ROM). In a computer, there are two types of memories :

1. Main Memory
2. Control Memory

The main memory is used for storing the programs written by the programmer. The contents of main memory may be charged when program is executed or when some another program is stored in it. The user program in main memory contains instructions and data.

The control memory contains the control signals in the form of micro-operations which are fixed and are not changed by the user.

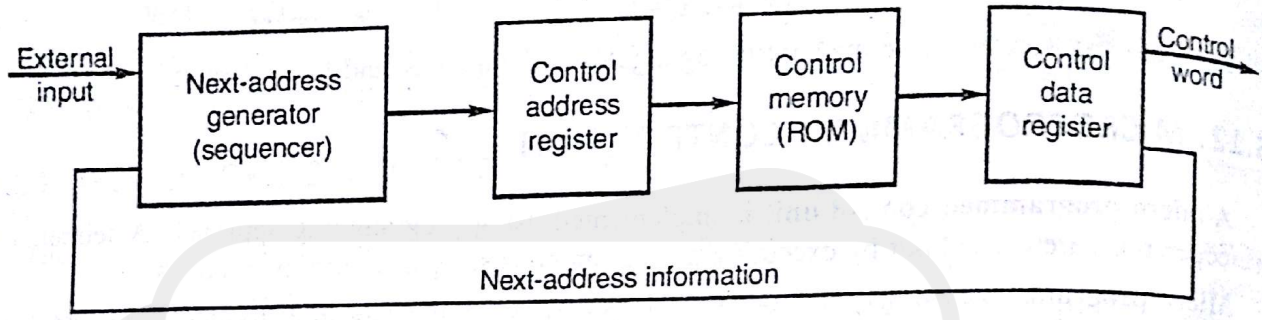


Fig. 5.24. Microprogrammed Control Organization

5.12.2. Address Sequencing

Each instruction has unique address. Making particular sequence of addresses of instructions is known as address sequencing. In the control memory micro-instructions are stored in groups. Each group represent specific routine. For execution of instruction each computer has its own routine. For proper execution of the instructions the hardware of the system must provide the facility to control the address sequencing of the control memory. Because the order of execution of instructions plays on important role. Control memory must also provide the facility of jumping from one routine to another routine if necessary.

Hardware is required to select the address of next micro-instruction in the control memory. It contains a set of control bits. The address of next micro-instruction can also be obtained explicitly by using an address field:

The address of next micro-instructions of control memory can be generated by different way. It can be received by Control Address Register through:

1. Mapping process
2. Branch Logic
3. Increment
4. Sub-routine

This can be shown in Fig. 5.25.

In general we can say that the address sequencing in control memory required the following facilities:

1. A mapping process between the bits of micro-operation and bits of ROM address.
2. Unconditional branch for the address field of micro-instruction.
3. Conditional branch for status bits in the registers.
4. Incrementing of control address register.
5. Facility for sub-routine calls and returns.

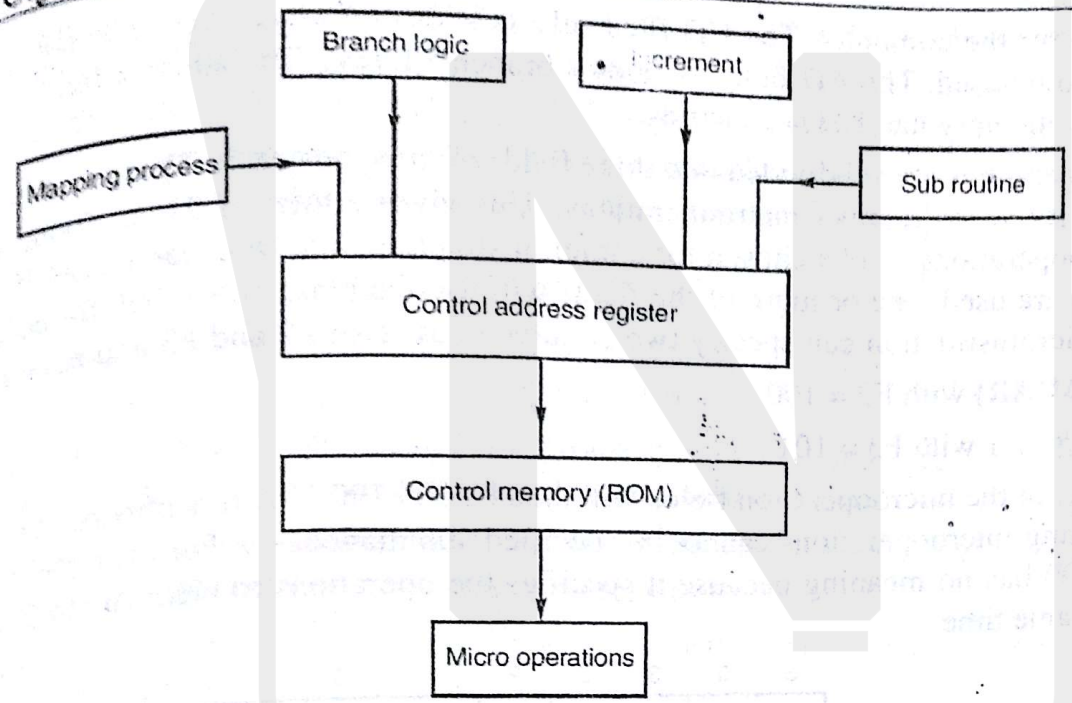


Fig. 5.25. Address Sequencing

7.2. CLASSIFICATIONS OF PARALLELISM

7.2.1. Flynn's Classification based on the Multiplicity of Instruction and Data Streams (1966)

Digital computer may be classified into four categories, according to the multiplicity of instructions and data stream. This scheme for classifying computer organizations was introduced by Michael J Flynn. The essential computing process is the execution of the sequence of instructions on a set of data. The term *stream* is used for here to denote a sequence of items (instructions or data) as executed or operated upon by a single processor. *Instructions* or *data* are defined with respect to the reference machine. An *instruction stream* is a sequence of instructions as executed by the machine; a *data stream* is a sequence of data including input, partial or temporary results, called for by the instruction stream.

Computer organizations are characterized by the multiplicity of the hardware provided to service the instruction and data streams. Flynn's four machine organizations are :

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

The capability of the processor varies in the these different types. The execution unit is indicated as a processor (PR) since it is often more complex than a traditional execution unit.

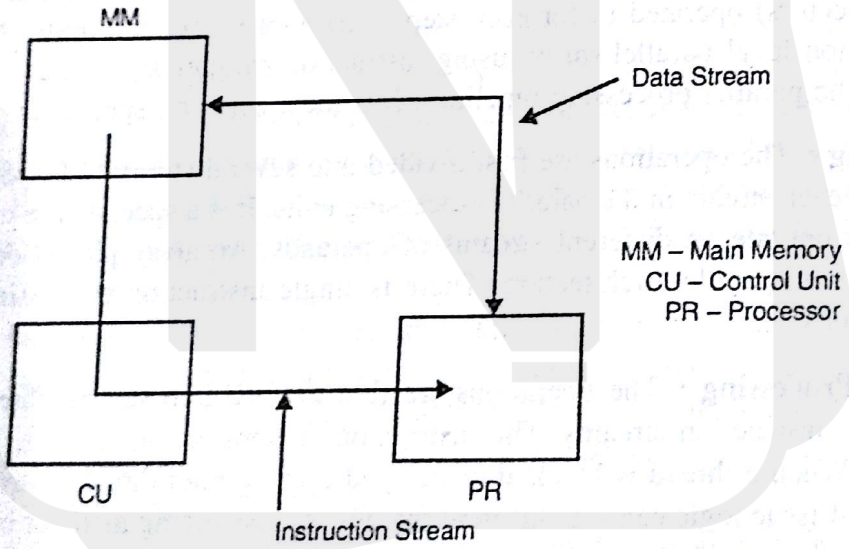


Fig. 7.1. Flynn's classification of computers

SISD represents the organization of a single computer that contains a control unit, a processor unit, and a memory unit. Instruction of Instructions are executed sequentially and the system may or may not have internal parallel processing capabilities. Parallel processing can be accomplished by means of multiple functional units or by pipelining.

SIMD represents an organization that is inclusive of any processing units under the supervision of a common control unit. All processors receive the same instruction from the control unit but operate on different items of data. The shared memory unit must contain multiple modules so that it can communicate with all the processors simultaneously. MISD structure is only of theoretical interest since no practical system has been constructed using this organization. MIMD organization refers to a computer system

capable of processing several programs at the same time. Most multiprocessor and multi-computer systems can be classified in this category.

Flynn's classification is dependent on the difference between the performance of the control unit and the data-processing unit. It stresses on the behavioural characteristics of the computer system rather than its operational and structural interconnections. One type of parallel processing that does not fit Flynn's classification is pipelining. The only two categories used from this classification are SIMD array processors and MIMD multiprocessors.

7.3.1. Pipelining

It is a technique of breaking down the sequential process into the sub operations, with each sub process being executed in a segmented manner that operates concurrently with all other segments. This process is basically a flow of binary information through the collection of processing segments. In the pipelining process, each segment performs partial processing dictated by the way the task is divided. In the pipelining process the result obtained from one segment is transferred to another in the pipeline. When the data passed through all the segments then we receive the final result.

The process pipelining has been arrived from the assembly language concept, as in the assembly language the code is executed in segmented format the same in the case with the pipelining. The major advantage of the pipelining process is that we can do the as many computations as we want in distinct segments at the same time. The concept of overlapping of computations ia made possible by associating a register with each segment in the pipeline, as the rgisters provide the isolation between each segment so tha each can operate on distinct data simultaneously.

A pipelining process with five units, which are also known as stages, is shown in Fig. 7.3(*b*), where a stage fetches the instruction from the memory and places it in a buffer until it is needed. Stage 2 decodes the instruction, determining its types and what operands it need. Stage 3 locates and fetches the operands, either from the registers or from the memory. Stage 4 actually does the work of carrying out the instruction, typically by running the oeprands through the data path. Finally the stage 5 writes the result back to the proper register.

7.3.1.1. Space-Time Diagram

The behaviour of a pipeline can be illustrated with a space-time diagram. This is a diagram that shows the segment utilization as function of time. The space-time diagrma of a four segment pipeline is demonstrated in figure below. The horizontal axis displays the time in clock cycles and the vertical axis gives the segment number. In the figure below it is shown that how the pipeline operates as the function of time.

- During the clock cycle 1, stage S1 is working on instruction 1 fetching it from memory.
- During clock cycle 2, stage S2 decodes the instruction 1, while stage S1 fetches instruction 2.
- During clock cycle 3, stage S3 fetches the operand for instruction 1, while stage S2 decodes instruction 2, and stage S1 fetches the third instruction 2.
- During clock cycle 4, stage S4 executes the instruction 1, S3 fetches the operand for instruction 2, while stage S2 decodes instruction 3, and stage S1 fetches the fourth instruction.
- Now, finaly during clock cycle 5, stage S5 writes the result of instruction 1 back, while the other stages work on the following instructions.

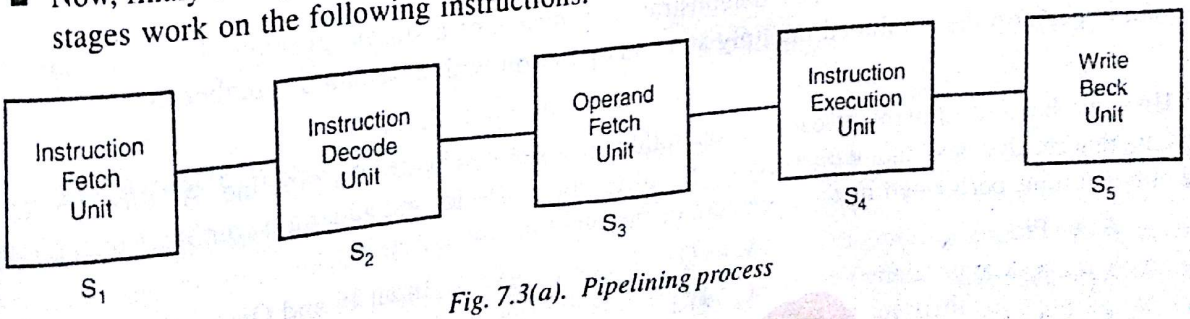


Fig. 7.3(a). Pipelining process

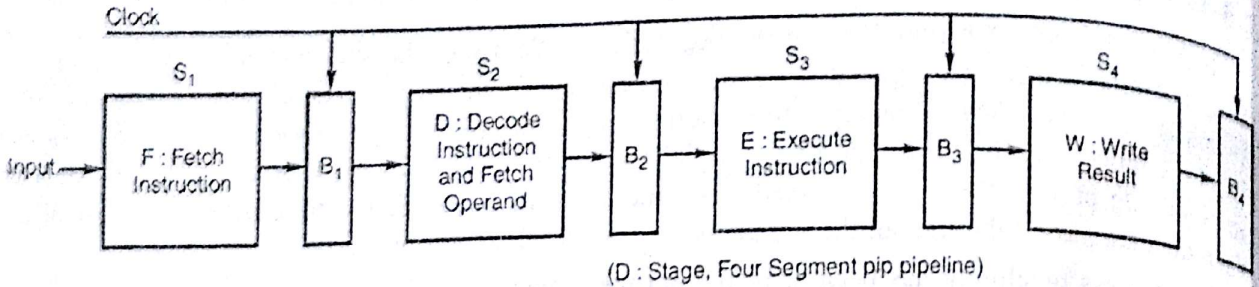


Fig. 7.3(b). Pipelining process

Now suppose that the cycle time of this machine is 2 nsec, then it takes 10 nsecs for an instruction to progress all the way through the five-stage pipeline. At first glance, with an instruction taking 10 nsecs, it might appear that the machine can run at 100 MIPS, but in fact it can do much better than this. At every clock cycle (2 nsec), one new instruction is completed, so the actual rate of processing is 500 MIPS, not 100 MIPS. The process of pipelining allows a tradeoff between **latency** *i.e.*, how long it takes to execute an instruction and processor bandwidth *i.e.* how many MIPS the CPU has. With a cycle time of T nsecs, and n stages in the pipeline, the latency is nT nsecs and the bandwidth is 1000/T MIPS.

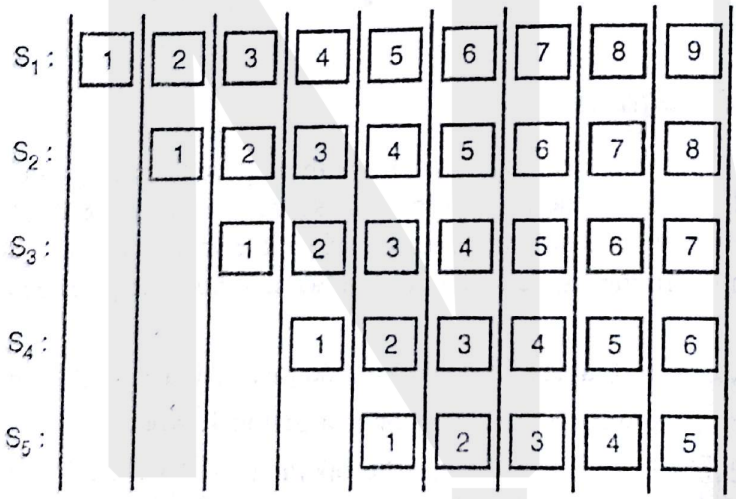


Fig. 7.4. Stage of Each stage as a function of time

7.3.1.2. Example of Pipeline Processing

The pipeline organization will be demonstrated by means of a simple example. Let us suppose that we want to perform the combined multiply and add operation with a stream of number as

$$Pi * Qi + Ri \quad \text{for } i = 1, 2, ..., 7$$

Here we have to implement each suboperation in a segment within a pipeline. A_1 through A_5 are registers that receive new data with every clock pulse. The multipler and adder are combinational circuits. The suboperatons performed in each segment of the pipeline are as follows :

$A_1 \leftarrow Pi,$	$A_2 \leftarrow Qi$	Input Pi and Qi
$A_3 \leftarrow A_1 * A_2,$	$A_4 \rightarrow Ri$	Multiply and input Ri
$A_5 \leftarrow A_3 + A_4$		Add Ri to product

Each segment has one or two registers and a combinational circuits as shown in Fig. 7.5 :

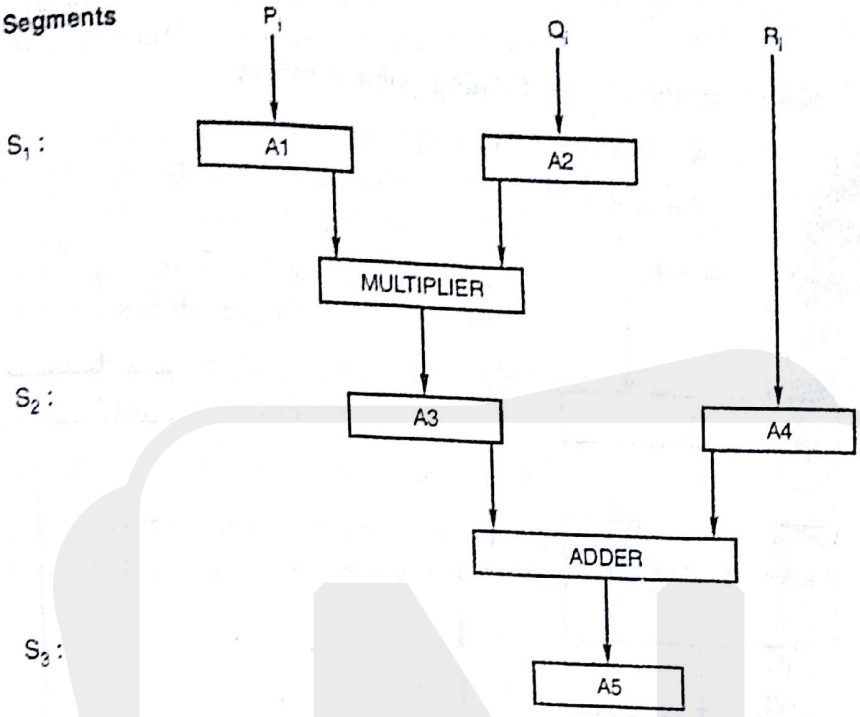


Fig. 7.5. Example of pipeline processing

7.3.1.3. Pipeline Performance

The pipelined processor completes the execution of one instruction in each clock cycle, which shows that the rate of instruction processing is four times the sequential operation. The performance resulting form pipelining is proportional to number of pipeline stages and this entrancement would be achieved only if the pipeline would be sustained without interrupting the program execution.

Once the pipe is filled up, it will output one result per clock period, independent of number of stages in pipe. Although a linear pipeline with k stages can process n tasks in $T_k = k + (n-1)$ clock peridos, where k cycles are required to fill the pipeline or to complete execution of first task and $(n-1)$ cycles are required for $(n-1)$ tasks. The same number of tasks can be executed in a non-pipelined processors with an equivalent function in $T = nk$ time delay. Speed up of a k stage linear pipeline processor over an equivalent non-pipeline processor as :

$$S_k = \frac{T_n}{T_k} = \frac{nk}{k+(n-1)} \text{ max. speed up as } n \geq k \text{ is } k.$$

7.4. TYPES OF PIPELINING

7.4.1. Arithmetic Pipelining

Large numerical applications often make use of repeated arithmetic operations for processing the elements of vectors and arrays. Architectures specialized for applications of this type often provide pipelines to speed processing of floating-point arithmetic sequences. This type of pipelining is called **arithmetic pipelining.** In arithmetic pipelining different stages of an arithmetic operation are handled along the stages of pipeline. In arithmetic pipelining different stages of an arithmetic operation are handled along the stages of pipeline. Floating point operations can be easily divided into suboperations

to carry out each suboperation in single stage of pipelining. Let us take an example to understand the concept of arithmetic pipeline :

The inputs to adder pipeline are two floating-point numbers.

$$X = A \times 2^a$$

$$Y = B \times 2^b$$

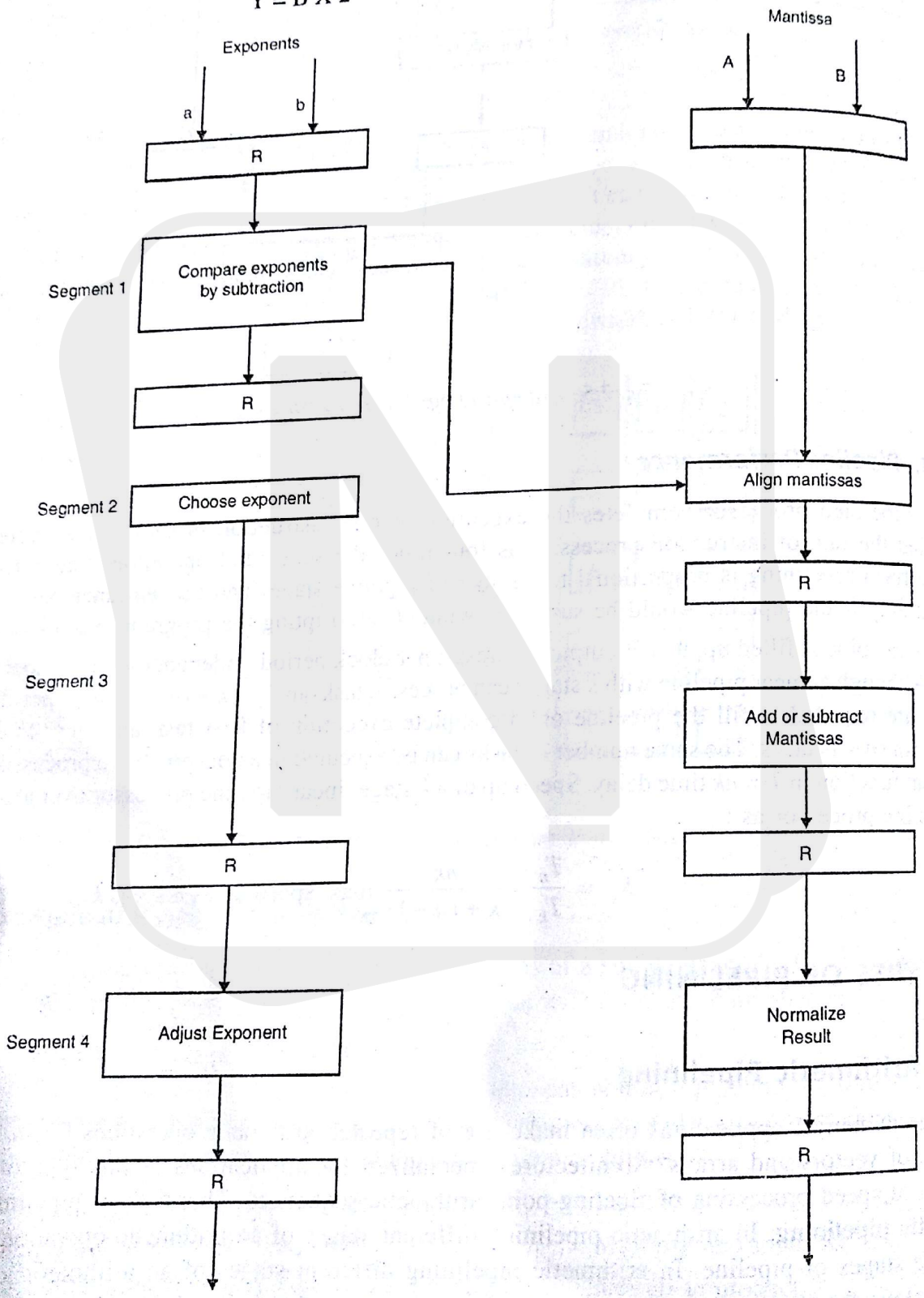


Fig. 7.6. Pipeline for Floating-point addition and subtraction.

Here A and B are the fractions that represents mantissas in the range $0.5 < f < 1.0$ and a and b are the exponents. The floating point addition and subtraction can be done in four segments. The registers named R are used to store the intermediate results. The operations that are performed in four segments are as follow :

1. Compute the larger exponent and the exponent difference. Shift the fraction corresponding to the smaller exponent right for a number of places equal to the difference. Both fractions are now adjusted to match the same (larger) exponent. Output the exponent and the two fractions.
2. Add or subtract the two fractions (mantissas), producing a sum. Pass through the exponent unchanged. Output the exponent and fractions.
3. Count leading zeros in the result fraction. Shift the fraction to normalize. Output the original exponent, the fraction, and the count.
4. Add the exponent and count. Output the adjusted exponent and the normalized fraction.

These steps follows the sequence of operations shown in the flowchart of Fig. 7.6. Let us consider the following example to understand the concept more clearly. Consider the two floating-point numbers :

$$X = 0.4532 \times 10^4$$

$$Y = 0.8267 \times 10^3$$

The two exponents are subtracted in first segment to obtain $4 - 3 = 1$. The larger exponent 4 is chosen as the exponent of the result. The next segment shifts the mantissa of Y to the right to obtain

$$X = 0.9532 \times 10^4$$

$$Y = 0.0826 \times 10^4$$

This aligns the two mantissas under the same exponent. The addition of two mantissas in segment 3 produces the sum

$$Z = 1.0358 \times 10^4$$

The sum is adjusted so that the result is a first nonzero digit in mantissa. This is done by shifting the mantissa once to the right and incrementing the exponent by one to obtain the normalized result

$$Z = 0.1035 \times 10^5$$

7.4.1.1. Performance improvement

The comparator, shifter, adder-subtractor, incrementer, and decrementer in the floating-point pipeline are implemented with combinational circuits. Suppose that the time delays of the four segments are $t_1 = 60$ ns, $t_2 = 70$ ns, $t_3 = 100$ ns, $t_4 = 80$ns, and the interface registers have a delay of $t_p = 10$ ns. The clock cycle is chosen to be $t_p = t_3 + t_r = 110$ ns. An equivalent nonpipeline floating- point adder-subtractor will have a delay time $t_n = t_1 + t_2 + t_3 + t_4 + t_r = 320$ ns. In this case the pipelined adder has a speedup of 320/110 = 2.9 over the nonpipelined adder.

7.4.2. Instruction Pipelining

In instruction pipeline each instruction is divided into a series of steps, so that all steps can execute simultaneously to execute complete instruction in very little time. An **instruction pipeline** is a technique used in the design of computers and other digital electronic devices to increase their instruction throughput (the number of instructions that can be executed in a unit of time).The fundamental idea is to split the

7.4.2.1. Hazards in Instruction Pipelining

1. **Structural hazards :** These hazards are because of conflicts due to insufficient resources when even with all possible combination, it may not be possible to overlap the operation.

2. **Data or Data dependent hazards :** These results when instruction in the pipeline depends on the result of previous instructions which are still in pipeline and not completed.

3. **Instruction or Control hazards :** They arise while pipelining branch and other instructions that change the contents of program counter. The simplest way to handle these hazards is to stall the pipeline. Stalling of the pipeline allows few instructions to proceed to completion while stopping the execution of those which results in hazards.

7.4.4. Pipeline Hazards

Hazards are problems with the instruction pipeline in central processing unit (CPU) microarchitectures that potentially result in incorrect computation. There are typically three types of hazards:

- data hazards

- structural hazards
- control hazards (branching hazards)

There are several methods used to deal with hazards, including pipeline stalls, pipeline bubbling, register forwarding, and in the case of out-of-order execution.

A hazard occurs when two or more of these simultaneous (possibly out of order) instructions conflict.

7.4.4.1. Data Hazards (Data Dependency)

Data hazards occur when instructions that exhibit data dependence modify data in different stages of a pipeline *i.e.* when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on the unpipelined machine. Data dependency is the condition in which the outcome of the current operation is dependent on the outcome of a previous instruction that has not yet been executed to completion because of the effect of the pipeline.

There are three situations in which a data hazard can occur:

Consider two instructions i1 and i2, with i1 occurring before i2 in program order.

(a) *Read After Write (RAW)*

(i2 tries to read a source before i1 writes to it) A read after write (RAW) data hazard refers to a situation where an instruction refers to a result that has not yet been calculated or retrieved. This can occur because even though an instruction is executed after a previous instruction, the previous instruction has not been completely processed through the pipeline.

(b) *Write After Read (WAR)*

(i2 tries to write a destination before it is read by i1) A write after read (WAR) data hazard represents a problem with concurrent execution.

(c) *Write After Write (WAW)*

(i2 tries to write an operand before it is written by i1) A write after write (WAW) data hazard may occur in a concurrent execution environment.

7.4.4.2. Structural Hazards (e.g. Resource dependency)

A structural hazard occurs when a part of the processor's hardware is needed by two or more instructions at the same time. A canonical example is a single memory unit that is accessed both in the fetch stage where an instruction is retrieved from memory, and the memory stage where data is written and/or read from memory. They can often be resolved by using two separate memories.

7.4.4.3. Control Hazards (Branch Hazards)

Branching hazards (also known as control hazards) occur with branches. On many instruction pipeline microarchitectures, the processor will not know the outcome of the branch when it needs to insert a new instruction into the pipeline (normally the fetch *stage).*

7.4.4.4. Eliminating or Resolving of hazards

Pipeline bubbling or stall or break.

This is a method for preventing data, structural, and branch hazards from occurring. As instructions are fetched, control logic determines whether a hazard could/will occur. If this is true, then the control

logic inserts NOPs into the pipeline. Thus, before the next instruction (which would cause the hazard) is executed, the previous one will have had sufficient time to complete and prevent the hazard. If the number of NOPs is equal to the number of stages in the pipeline, the processor has been cleared of all instructions and can proceed free from hazards. This is called *flushing the pipeline*. All forms of stalling introduce a delay before the processor can resume execution.

(a) Data Hazards

There are several main solutions and algorithms used to resolve data hazards:

- insert a pipeline bubble whenever a read after write (RAW) dependency is encountered, guaranteed to increase latency, or
- utilize out-of-order execution to potentially prevent the need for pipeline bubbles
- *utilize* register forwarding *to use data from later stages in the pipeline*

In the case of out-of-order execution, the algorithm used can be:

- **scoreboarding**, in which case a *pipeline bubble* will only be needed when there is no functional unit available
- the **Tomasulo algorithm**, which utilizes register renaming allowing the continual issuing of instructions

We can delegate the task of removing data dependencies to the compiler, which can fill in an appropriate number of NOP instructions between dependent instructions to ensure correct operation, or re-order instructions where possible.

(b) Register Forwarding

Forwarding involves feeding output data into a previous stage of the pipeline. Forwarding is implemented by feeding back the output of an instruction into the previous stage(s) of the pipeline as soon as the output of that instruction is available.

(c) Control Hazards (Branch Hazards)

To avoid control hazards:

- insert a *pipeline bubble* (discussed above), guaranteed to increase latency, or
- use branch prediction and essentially guesstimate which instructions to insert, in which case a *pipeline bubble* will only be needed in the case of an incorrect prediction

5.13. DIFFERENCE BETWEEN HARDWIRED AND MICROPROGRAMMED CONTROL UNIT

Attribute	*Hardwired Control*	*Microprogrammed Control*
Speed	Fast	Slow
Control Functions	Implemented in hardware	Implemented in software
Flexibility	Not flexible, to accommodate new system specifications or new instructions.	More flexible, to accommodate new system specification or new instructions redesign is required.
Ability to Handle Large/ Complex Instruction sets	Some what difficult	Easier
Ability to Support Operating Systems and Diagnostic Features	Very difficult (unless anticipated during design)	Easy
Design Process	Somewhat complicated	Orderly and systematic
Applications	Mostly RISC microprocessors	Mainframes, some microprocessors
Instruction set Size	Usually under 100 instructions	Usually over 100 instructions
ROM size	—	2K to 10K by 20-400 bit microinstructions
Chip Area Efficiency	Uses least area	Uses more area

2.2. COMPLEMENTS

In digital computer complements are used to simply the operation of subtraction and for logical manipulation. For each bse system r two types of complements are defined.

1. r's complement
2. $(r-1)$'s complement.

When the value of base r is 2 then 2's complement and 1's complement are defined. When the value of r is 10 then 10's complement and 9's complement are defined.

2.2.1. r's Complement

It is radix complement. The r's complement of a given number N having n digits and base r is defined as

$$= r^n - N \quad \text{for } N \neq 0$$
$$= 0 \quad \text{for } N = 0$$

$$r\text{'s complement} = r^n - N$$
$$= [(r^{n-1}) - N] + 1$$
$$= (r-1)\text{'s complement} + 1$$

So adding 1 to $(r-1)$'s complement, we get r's complement of a number.

2.2.1.1. 2's Complement

It is defined for binary number system. The 2's complement of a given number N is defined as

$$2^n - N$$

where n = number of bits in number

2's complement can be findout using two method.

Method 1. (*i*) Findout the 1's complement of the given number by changing 1 into 0 and 0 into 1.

(*ii*) Add 1 to 1's complement to get 2's complement of the number.

Method 2. (*i*) Start from the LSB of the given number.

(*ii*) The first 1 occurs will remain unchanged.

(*iii*) After the first 1 change all significant bits 0 by 1 and 1 by 0.

2.2.1.2. 10's Complement

10's complement is defined for decimal number system. 10's complement is defined as

$$= 10^n - N$$

where N = Given number

n = number of digits

10's complement can be written as

$$= [10^n - 1] - N + 1$$

$$= 9\text{'s complement} + 1$$

So 10's complement of a given number can be obtained by adding 1 to 9's complement.

Example 2.13. *Find out 2's complement of* $(100101.11)_2$.

Solution. *Direct Method.* Scan the number starting from LSB. LSB is 1 so it will remain unchanged and all other bits will change from 0 to 1 and 1 to 0.

So	1	0	0	1	0	1	.	1	1
	↓	↓	↓	↓	↓	↓	.	↓	↓
	0	1	1	0	1	0	.	0	1

Decimal point is placed at the same place so 2's complement is 011010.01.

Example 2.14. *Find out 10's complement of 232140.*

Solution. Here

$$N = 232140$$

$$n = 6$$

So 10's complement is

$$= 10^6 - 232140$$

$$= 1000000 - 232140$$

$$= 767860$$

2.2.2. (r – 1)'s Complement

Given a number N having base r and n digit, the $(r-1)$'s complement of N is defined as $(r^n - 1) - N$.

2.2.2.1. 1's Complement

It is defined for binary number system. The 1's complement of any given number N is $(2^n - 1) - N$. 2^n represents a binary number whose MSB is 1 and all other bits are zero. For example if $N = 1001$ then

$$2^n = 2^4(16)_{10} = (10000)_2$$

and

$$2^n - 1 = 10000 - 1 = (1111)_2$$

Then 1's complement of $(1001)_2$ is

$$= [2^n - 1] - N = 1111 - 1001 = 0110.$$

$(2^n - 1)$ has n 1's, so 1's complement of a given number can be obtained when each bit of N is subtracted from 1. Bits are only two either 0 or 1.

So

$$1 - 0 = 1$$

$$1 - 1 = 0$$

Implies 0 is got converted into 1 and 1 is converted into 0.

Therefore, the 1's complement of a given number can be obtained by replacing each bit of the number *i.e.*, '1' into '0' and '0' into 1. It is direct method bny which easily we can get 1's complement of a given binary number.

2.2.2.2. 9's Complement

9's complement of a given number is defined as $[10^n - 1] - N$

where n = Number of digits in number

N = Given decimal number

10^n represents a number that consists a single 1 followed by n 0's. $[10^n - 1]$ will represent a number having n 9's. For example,

$$N = (1234)_{10}$$

then $$10^n - 1 = 10^4 - 1 = 10000 - 1 = 9999$$

9's complement of $(1234)_{10}$ is

$$= [10^n - 1] - N$$
$$= 9999 - 1234$$
$$= 8765$$

So 9's complement of a given number N can be obtained by subtracting each digit of given number by 0. It is very easy and direct method.

Example 2.15. *Find out 1's complement of* $(10\ 101.1011)_2$.

Solution.

1	0	1	0	1	.	1	0	1	1
↓	↓	↓	↓	↓	.	↓	↓	↓	↓
0	1	0	1	0	.	0	1	0	0

Example 2.16. *Find out the 9's complement of* $(413052)_{10}$.

Solution. 9's complement is

$$= [10^6 - 1] - 413052$$
$$= 999999 - 413052$$
$$= 586947.$$

2.2.3. Subtraction Using 2's Complement

The subtraction of two binary number can be obtained using 2's complement. 2's complement of number can be directly obtained by adding 1 to 1's complement.

Let we have to perform $(A - B)$ then using 2's complement method we have to follow the following steps :

(*i*) Find out the 2's complement of B.

(*ii*) Add $A + B$.

(*iii*) If carry is generated then ignore it. If there is no carry then take 2's complement of the result and change the sign of result.

Example 2.17. *Subtract* $(1010)_2$ *from* $(1111)_2$ *using 2's complement method.*

Solution.

$$B = (1010)_2$$
$$A = (1111)_2$$
2's complement of B $= 0101 + 1 = 0110$

Addition of A and B $= 1111 + 0110$

$$= 10101$$
$$\uparrow$$
carry generation

The addition operation is generating carry.

So after ignoring carry bit we will get answer is $(0101)_2$.

Example 2.18. *Subtract 5 from 3 using 2's complement method.*

Solution.

$(A) = (3)_{10} = (0011)_2$

$(B) = (5)_{10} = (0101)_2$

2's complement of 5 $= 1010 + 1$

$= 1011$

Adding A and B

$$\begin{array}{r} 0\,0\,1\,1 \\ 1\,0\,1\,1 \\ \hline 1\,1\,1\,0 \end{array}$$

There is no carry so we take 2's complement of the result and will change the sign of the result.

2's complement of 1110 is $0\,0\,0\,1$

$$\begin{array}{r} +\,1 \\ \hline 0\,0\,1\,0 \end{array}$$

$(0010)_2 = (2)_{10} = (-2)_{10}$ [after changing the sign]

2.3. FIXED POINT NUMBERS

A number having a fixed length is called as fixed point number. In selecting a number representation to be used in a computer, the following factors should be taken into account.

- The numbers types to be represented; for example, integers or real numbers.
- The range of values likely to be encountered.
- The precision of the numbers, which refers to the maximum accuracy of the representation.
- The cost of the hardware require to store and process the numbers.

We have two formats:

- fixed-point and
- floating point.

Fixed point formats allow a limited range of values and also require less hardware for processing. On the other hand, floating-point numbers allow large range of values but require costly processing hardware or lengthy software implementations.

In binary number system, a number can be represented as an integer or a fraction. Depending on the design, the hardware can interpret numbers as an integer or fraction. The radix point is never explicitly specified. It is easier in the design and the hardware interprets it accordingly. In integer numbers, the radix point is fixed and assumed to be the right of the rightmost digit. As radix point is fixed, the number system is referred to as fixed point number system.

Fixed Point Representation

The fixed point/integer numbers are represented in two forms:

- *signed integer and*
- *unsigned integer.*

Unsigned integer numbers represent positive numbers. To represent negative numbers various techniques are used because computers do not have any provision to represent negative sign.

2.3.1. Signed Binary Numbers

A binary number can have only two symbols 0 and 1, these symbols are used to indicate positive binary number and negative binary number. In the n bits of signed binary number, one bit is reserved to indicate positive or negative value and the remaining bits are magnitude. The sign bit is placed as the most significant bit. The MSB, 0 indicates that the number is positive and MSB, 1 indicates negative binary number. The signed binary number can be represented in one of the three ways.

1. Sign-magnitude representation.
2. 1's complement representation.
3. 2's complement representation.

Sign-magnitude Representation

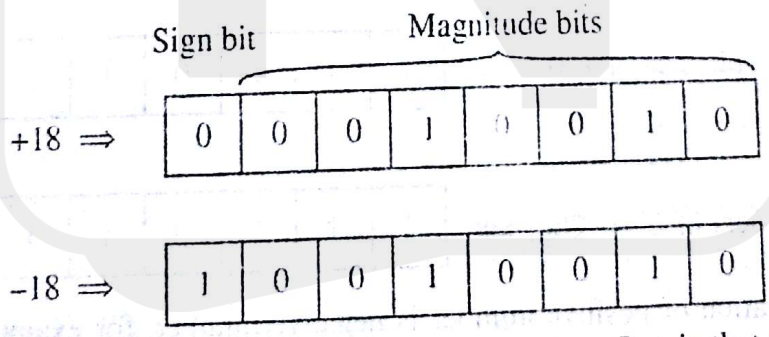
The use of unsigned integer is insufficient in many cases when we need to represent negative as well as positive numbers. The simplest form of representation that employs a sign bit is the sign magnitude representation. In an n bit word, the right most $n-1$ bit holds the magnitude of the given number and nth bit is assigned for sign bit. If the sign bit (nth bit) is

0	⇒ The given number is positive
1	⇒ The given number is negative

For example, +18 and –18 are represented as 8 bit signed number as follows.

18 equivalent binary number = 10010

In the given 8 bit signed number, one bit is assigned to be sign bit (MSB). The remaining 7 bits represent the magnitude, but the binary equivalent of 18 is 10010 (5 bits). Two 0's are placed left most to make a 7 bit group (0010010), now we can represent the positive and negative number.



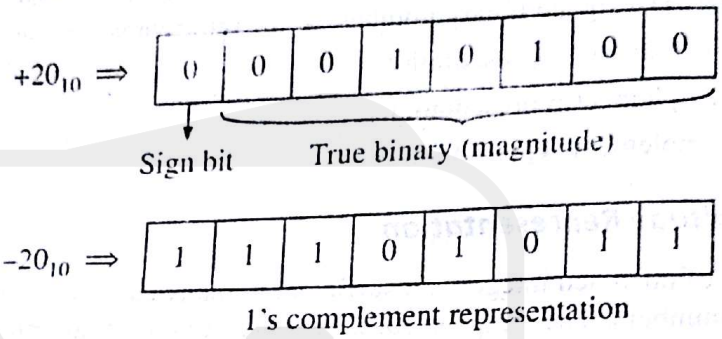
There are several drawbacks in sign-magnitude representation. One is that addition and subtraction require consideration of both the signs of the numbers and their relative magnitude in order to carry out the required operation. Another drawback is that there are two representations for 0,

$$+0 = 0\,0\,0\,0\,0\,0\,0\,0\,0$$
$$-0 = 1\,0\,0\,0\,0\,0\,0\,0\,0$$

This is inconvenient, because it is slightly more difficult to test for 0. The signed magnitude is not normally used because the circuit implementation is more complex than the other systems.

1's Complement Representation

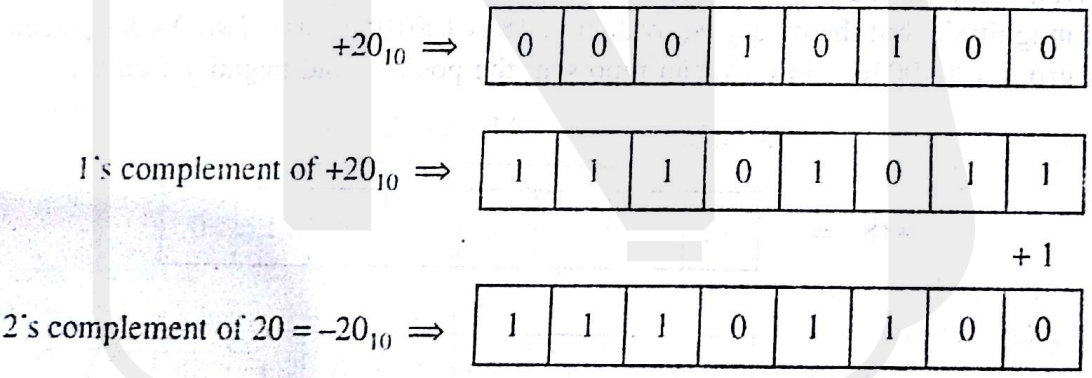
The 1's complement of a binary number is obtained by changing each 0 to 1 and each 1 to 0. For 1's complement representation of signed number we have to find the 1's complement including sign bit. For example, let us see how 20 is represented in 8 bit 1's complement form. The magnitude of 20 is 10100 (5 bits). The 8 bit signed number consists of one bit (MSB) for sign bit and the remaining 7 bits represents magnitude of the given number. Two 0's are placed to make a 7 bit group, and the 1's complement representation of +20 and –20 is given as follows.



2's Complement Represenation

If 1 is added to 1's complement of a binary number, the resulting number is known as the 2's complement of the binary number.

For example, $(-20)_{10}$ is represented as 8 bit 2's complement by adding 1 in the LSB to 1's complement. Whenever a signed number has a 1 in the sign bit and all 0's for the magnitude bits, its decimal equivalent is -2^N, where N is the number of bits in the magnitude.



Note: Complementation of positive number is negative number, for example complement of +8 is the –8.

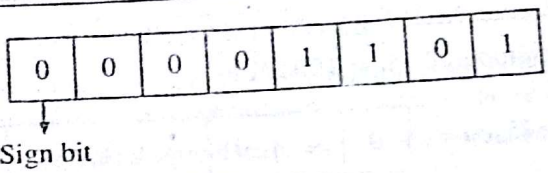
Example 2.19. *Represent each of the following signed decimal numbers as a signed bginary number in 8 bit sign magnitude representation and 1's complement representation.*

(a) +13, (b) –13 (c) +48 (d) –75

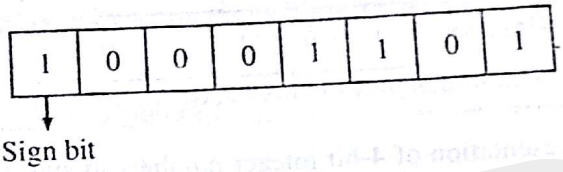
Solution.

(a) +13

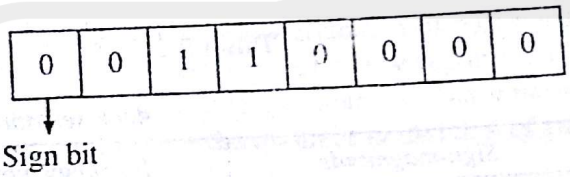
Magnitude of 13 = 1101

$+13_{10} \Rightarrow$ 

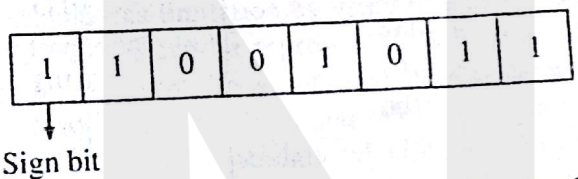
(*b*) –13

$-13_{10} \Rightarrow$ 

(*c*) +48

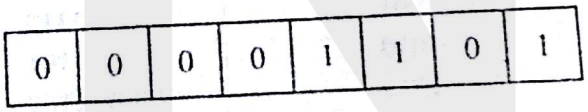
$+48_{10} \Rightarrow$ 

(*d*) –75

$-75_{10} \Rightarrow$ 

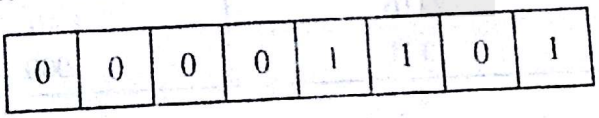
1's Complement Representation: In the 1's complement, positive number representation is the same as for signed magnitude representation but 1's complement representation is diffewrent from signed magnitude reprsentation for negative numbers.

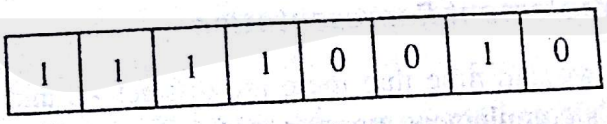
(*a*) +13

$+13_{10} \Rightarrow$ 

(*b*) –13

Magnitude of 13 = 1101

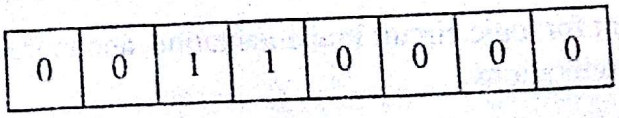
$+13_{10} \Rightarrow$ 

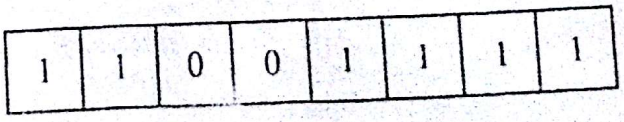
$-13_{10} \Rightarrow$ 

1's complement

(*c*) +48

Magnitude of 48 = 11000

$+48_{10} \Rightarrow$ 

$-48_{10} \Rightarrow$ 

(d) –75

Magnitude of 75 = 1001011

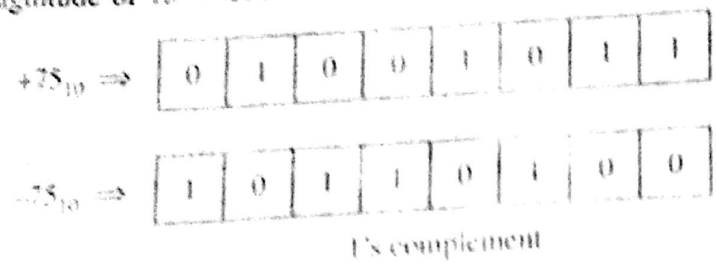


Table 2.2 shows representation of 4-bit integer numbers in sign-magnitude, 1's complement and 2's complement form.

Table 2.2

Decimal	Representations		
	Sign-magnitude	*1's complement*	*2's complement*
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000
–0	1000	1111	—
–1	1001	1110	1111
–2	1010	1101	1110
–3	1011	1100	1101
–4	1100	1011	1100
–5	1101	1010	1011
–6	1110	1001	1010
–7	1111	1000	1001

dvantage of 2's Complement Representation

- From Table 2.2, we can note that there are distinct +0 and –0 representation in both the sign magnitude and 1's complement representations, but the 2's complement representation has only one representation.
- For 4-bit numbers, the value –8 is representatble only in the 2's complement representation and not in other representations.
- It is more efficient for logic circuit implementation, and more often used in computers for addition and subtraction operations.

IMPORTANT NOTES

- **Fixed point numbers:** Radix point is a point which separates the integer and fraction part of a number. If the radix point is fixed then the number is called fixed point number.
- Signed number can be represented as signed number, 1's complement representation and 2's complement representation.
- 1's complement representation has two representation for +0 and –0, this is drawback of this representation.
- 2's complement representation is best representation of signed number.

2.4. FLOATING POINT NUMBERS

The floating point numbers contains the binary point variable in its position and hence these numbers are called floating point numbers. The range of numbers that can be represented by a fixed-point number is insufficient. This *fixed point has limitations, very large numbers cannot be represented, nor can very small fractions.* Furthermore, the fractional part of the quotient in division of two large numbrs could be lost.

For decimal numbers, one gets around this limitation by using scientific notation. Thus, 976000 can be represented as 9.76×10^5 and 0.0000976 can be represented as 9.76×10^{-5}. To represent binary numbers, it is necessary to consider binary point. 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 points variable and is automaticlaly adjusted as computation proceeds. In this case, *the binary point is said to float, and the numbers are called floating point numbers.*

The floating point representation has the following three fields:

1. Sign
2. Mantissa
3. Exponent.

The general structure of floating point number is

$$\pm M \times B^{\pm E}$$

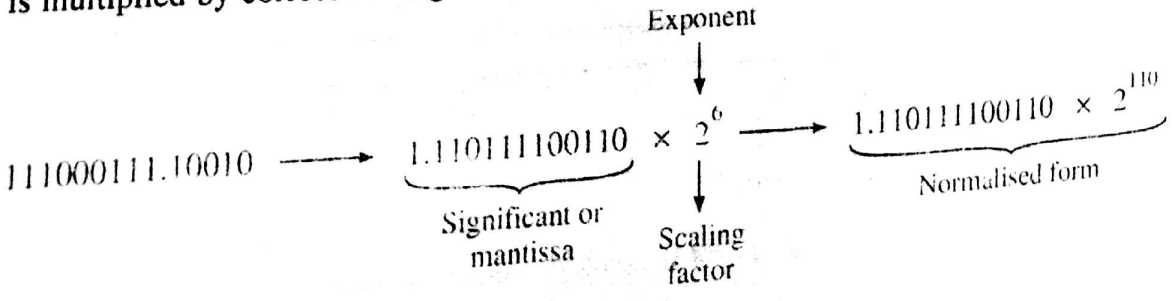
Where M is Significant (Mantissa) digits

E is Exponent

B is scaling factor, which is

- 2 for binary number
- 10 for decimal number.

Let us consider the number 1110111.100110 to be represented in the floating point format. To represent the number in floating point format, first binary point is shifted to left of the sign bit and the number is multiplied by correct scaling factor to get the same value.



It is important to note that the base in the scaling factor is fixed as 2 and therefore, it does not need to appear explicity in the machine representation of a floating point number. The string of the significant digits is commonly known as mantissa.

In the above representation, we can say that

Sign = 0

Mantissa = 110111100110

Exponent = 110 (6)

In floating point number, bias value is added to the true exponent. Due to this the magnitude of the numbers can be compared by doing arithmetic on the exponent part.

The range of floating point is

$$-1 \le F \le 1 - 2^{-(n-1)}$$

If $n = 32$ bit, from

$-1 \le F \le 1 - 4.65661 \times 2873 \times 10^{-10}$

$-1 \le F \le 0.9999$

IMPORTANT NOTES

- **Mantissa:** The floating point representation of a number has two parts. The first part represents a signed, fixed-point number called the mantissa.
- **Exponent:** The second part of floating point designates the position of the decimal (or binary) point is called exponent.
- **Floating point number:** A number contains the binary point variable in its position and hence these numbers are called floating point numbers.
- **Normalization:** A floating point number is said to be normnalized if the most significant digit of the mantissa is non-zero. For example, the decimal number 350 is normalized but 00035 is not normalized.

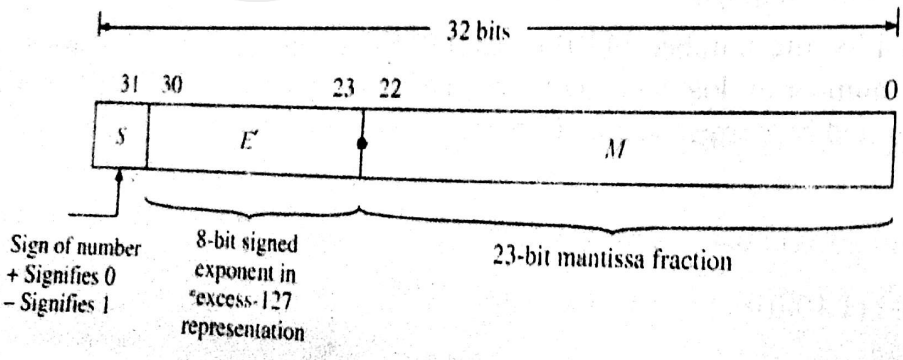
2.4.1. IEEE Standards of Floating Point Representation

The standards for representing floating point numbers is 32 bits and 64 bits have been developed by the Institute of Electrical and Electronics Engineers (IEEE), refers to as IEEE 754 standards.

Based on the total number of bits, we can say the floating point numbers are **single precision** (32 bit) and **double precision** (64 bit).

Single Precision

The 32 bit IEEE standard floating point representation is shown in Fig. 2.1.



Value represented = $\pm 1.M \times 2^{E-127}$

Fig. 2.1. IEEE 754 standard floating point numbers

- The sign of a number is given in the first bit followed by a representation for the exponent (to the base 2) of the scaling factor.
- Instead of the signed exponent E, the value actually stored in the exponent field is $E' = E + \text{bias}$.
- In the 32-bit floating point system (single precision) bias is 127. Hence $E' = E + 127$.
- This representation of exponent is also called the excess-127 format.
- The end values of E' namely 0 and 255 are used to indicate the floating point values of exact zero and infinity respectively in single precision.
- Thus E' is in the range $0 \leq E' \leq 255$.
- The actual exponent E is in the range $-126 \leq E$ 127. The last 23 bits represent the mantissa.

Note: The values 0 and 255 are used to indicate the floating point values of exact 0 and infinite respectively.

Double Precision

The 64-bit standard representation shown in Fig. 2.2 is called a double precision representation because it occupies two 32-bit words.

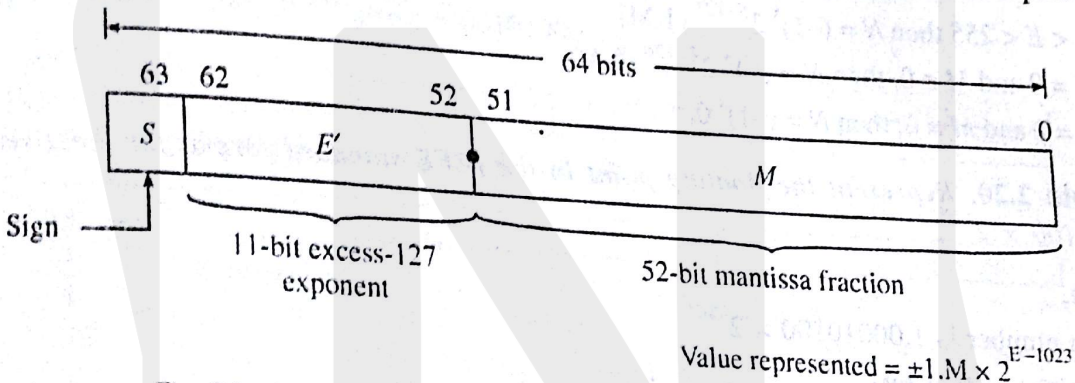


Fig. 2.2. Double precision floating point representation (IEEE 754).

- In the double precision format value actually stored in the exponent field is given as $E' = E + 1023$.
- Here bias value is IO23 and hence it is also called excess-1023.
- The end values of E', namely 0 and 2047 are used to indicate the floating point exact values of zero and infinity respectively.
- Thus the range of E' for normal values in double precision is $0 < E' < 2047$. This means that for 64-bit representation the actual exponent E is the range

$$-1022 \leq E \leq 1023$$

Paramater	Format	
	Single	Double
Word width (bits)	32	64
Exponent width (bits)	8	11
Exponent bias	127	1023
Maximum Exponent	127	1023
Minimum Exponent	−126	−1022
Number of Exponents	254	2046
Number of Fractions	2^{23}	2^{52}
Number of Values	1.98×2^{31}	1.99×2^{63}

The IEEE floating-point defined formats are specified for the results of overflow, underflow and other exceptional conditions. The IEEE standard's exception formats are intended to set flags in the processor, which subsequent insturctions can use for error control.

- If the result of a floating-point operation is not a valid floating-point number, then a special code referred to as Not a Number (NaN). Examples of operations that result in NaNs are dividing zero by zero and taking the square root of a negative number.
- NaN formats are identified in the standard by $M \neq 0$ and $E = 255$ (32-bit format) or $E = 2047$ (64-bit format).
- When overflow occurs which means that a number has been produced whose magnitude is too big to represent this result is referred to an infinity and is identified by $M = 0$ and $E = 255$ (32-bit format) or $E = 2047$ (64-bit format).

In summary, the number N represented by a 32-bit IEEE-standard floating-point number has the following set of interpretations.

If $E = 255$ and $M \neq 0$, then $N = \text{NaN}$

If $E = 255$ and $M = 0$, then $N = (-1)^s \infty$

If $0 < E < 255$ then $N = (-1)^s 2^{E-127} (1.M)$

If $E = 0$ and $M \neq 0$, then $N = (-1^s 2^{E-126} (0.M)$

If $E = 0$ and $M = 0$, then $N = (-1)^s 0$.

Example 2.20. *Represent the floating point in the IEEE standard format for the given number* $1.00010100 \times 2^{-10}$.

Solution.

Given number is $1.00010100 \times 2^{-10}$

Sign bit(s) $\Rightarrow$ 0 (1 bit)

Mantissa (M) $\Rightarrow$ 00010100 ... 0 (23 bits)

Actual Exponent $E = -10$

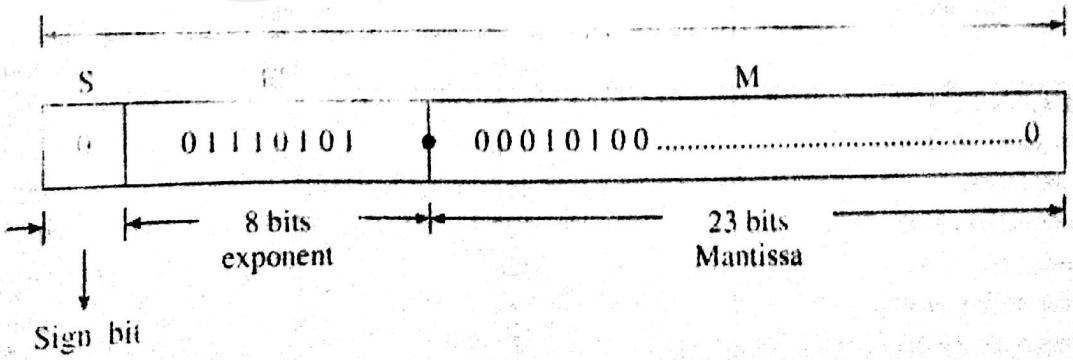
Modified Exponent $E' = E + 127 = -10 + 127$

$= +117$

Equivalent binary representation

$117 \Rightarrow 1110101$

8 bits are allotted to represent the exponent values, but in this example we have only 7 bits. Padding zero to the above 7 bits we to make a group of 8 bits.



Example 2.21. *Represent the floating point in the IEEE standard format for the given number.* $1.001010 \ldots 0 \times 2^{-87}$.

Solution.

Given number's $1.001010 \ldots 0 \times 2^{-87}$

Sign bit $S = 0$ (The given number is positive number)

Mantissa $M = 0010100 \ldots 0$ (23 bits)

Modified Exponent $E' = 127 + E$

Where E is actual given exponent

$E = -87$

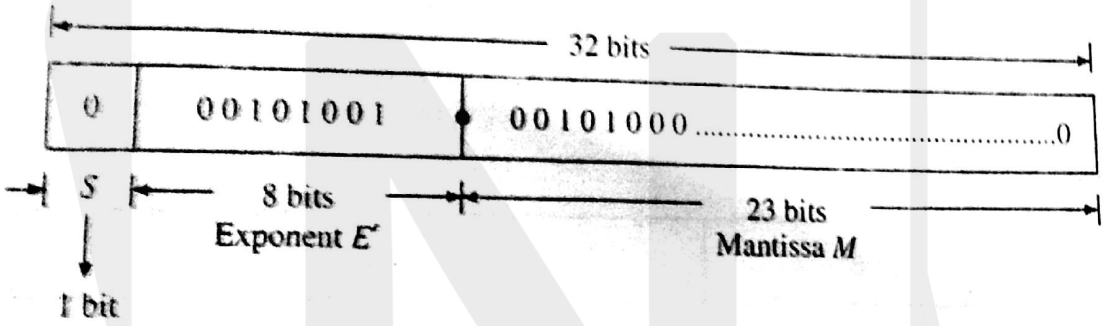
$E' = -87 + 127$

$E' = 40$

Equivalent binary representation $E' = (40)_{10}$

$= 101000$

Padding two zeros to make 8 bit group = 00101000



Example 2.22. *Represent* 1460.125_{10} *in single precision and double precision formats.*

Solution.

Step 1. Convert the decimal number in binary format

For integer

2	1460
2	730 – 0
2	365 – 0
2	182 – 1
2	91 – 0
2	45 – 1
2	22 – 1
2	11 – 0
2	5 – 1
2	2 – 1
2	1 – 0

$(1460)_{10} = (10110110100)_2$

For fraction

$0.125 \times 2 = 0.250 \Rightarrow 0$

$0.250 \times 2 = 0.500 \Rightarrow 0$

$0.500 \times 2 = 1.000 \Rightarrow 1$

$(0.125)_{10} = (0.001)_2$

$(1460.125)_{10} = (10110110100.001)_2$

Step 2. Normalise the number.

$$10110110100.001 = 1.0110110100001 \times 2^{10}$$

Now we represent the above floating point in single precision and double precision.

Single precision: From the above floating point

$S = 0$

$E = 10$

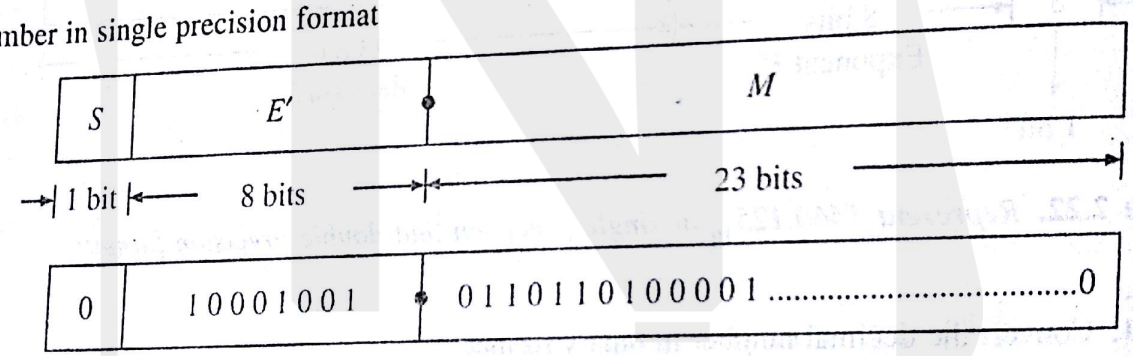
$M = 0110110100001$

The modified exponent (an bias for single precision)

$E' = 127 + E = 127 + 10 = 137_{10}$

$= 10001001_2$

Number in single precision format



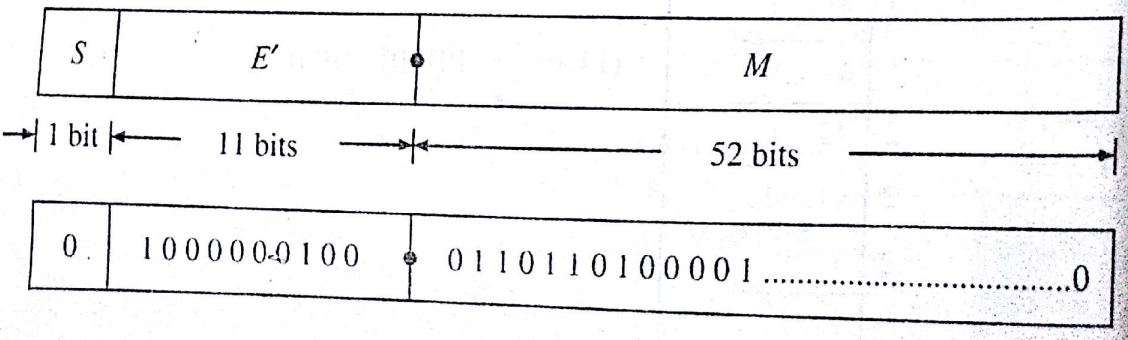
Double Precision: For a given number.

$S = 0$, $E = 10$ and $M = 0110110100001$

Bias (or) modified exponent for double precision

$E' = E + 1023 = 10 + 1023$

$= (1033)_{10} = (10000001001)_2$



Example 2.23. *Represent -307.1875_{10} in single precision and double precision formats.*

Solution.

Step 1. Convert the decimal number in binary format.

Integer format

2	307
2	153 – 1
2	76 – 1
2	38 – 0
2	19 – 0
2	9 – 1
2	4 – 1
2	2 – 0
2	1 – 0

$(307)_{10} = (100110011)_2$

Fractional format

$$0.1875 \times 2 = 0.370 \Rightarrow 0$$
$$0.3750 \times 2 = 0.750 \Rightarrow 0$$
$$0.750 \times 2 = 1.5 \Rightarrow 1$$
$$0.5 \times 2 = 1.0 \Rightarrow 1$$
$$(0.1875)_{10} = (0.0011)_2$$
$$(307.1875)_{10} = (100110011.0011)_2$$
$$-307.1875_{10} = -100110011.0011_2$$

Step 2. Normalise the number

$$-100110011.0011 = -1.001100110011 \times 2^8$$

Now we will see the represntation of the numbers in single precision and double precision formats.

Single precision: For a given number

$S = 1$ (given number is negative number)

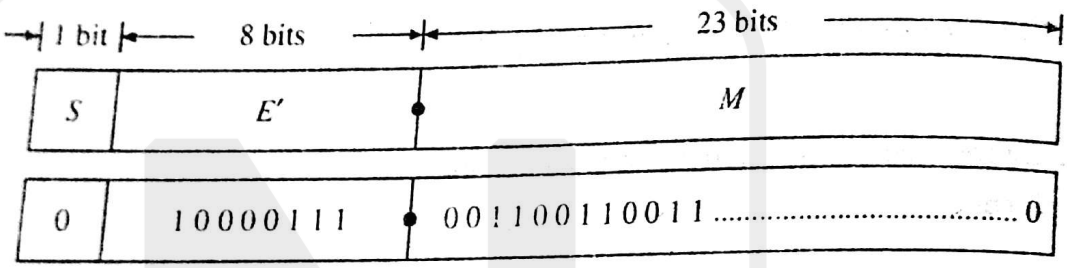
$E = 8$

$M = 001100110011$

Bias of single precision format is = 127

$$E' = 127 + E$$
$$= 127 + 8$$
$$= (135)_{10}$$
$$= (10000\ 0111)_2$$

Number in single precision representation



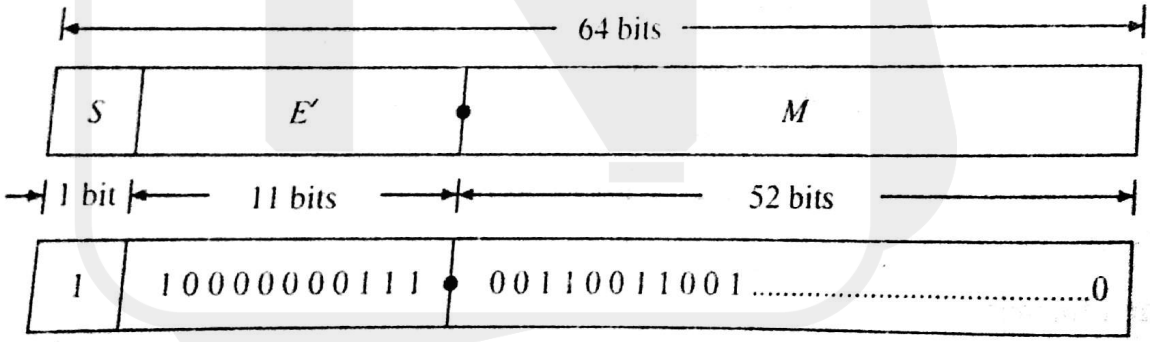
Double precision: For a given number

$S = 1, E = 8, M = 001100110011$

Bias for double precision format is = 1023

$$E' = 1023 + E = 1023 = 8$$
$$= 1031_{10} = 10000000111$$

Number in double precision



8 Computer Arithmetic

Arithmetic instruction in digital computers manipulate data to produce results necessary for the solution ofr computational problems. These instructions perform the basic arithmetic calculations. Basic arithmetic operations are addition, subtraction, multiplication and division. Other arithmetic functions are generated from these basic operations.

8.1. ARITHMETIC TYPES

Theoretically, a computer can operate on any type of data through appropriate programming. A computer is practically designed with a limited type of arithmetic. Figure 8.1 shows popular arithmetic

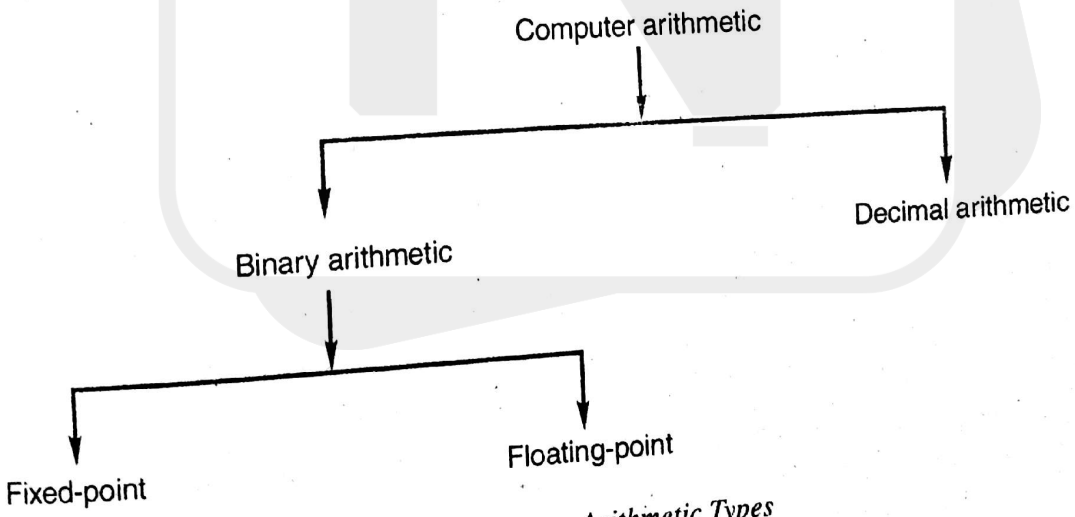


Fig. 8.1. Computer Arithmetic Types

types used in computers. Since the internal circuits of computers are binary in nature, binary arithmetic is available in almost all the computers. The user uses decimal number system while all the input to computers are decimal. These are covered into binary numbers by the computer before performing the

arithmetic operations. The final result is converted into decimals and presented to the user. This concept works well due to limited number of conversions. In special business applications, the amount of data is large but the extent of arithmetic operation is limited. In these cases, a lot of time is wasted in conversions if we only use binary arithmetic. Hence computers are designed with decimal arithmetic for such applications.

At an early edge we are taught how to perform the basic arithmetic operations is signed magnitude representation usually an algorithm will contain a number of procedural steps which are dependent upon the results of the previous steps. A convenient method for presenting algorithms is a flow chart. The computational steps are specified in the flow chart inside rectangular boxes.

In this chapter we develop the various arithmetic algorithms and show the procedure for implementing them with digital hardware. We consider addition, subtraction, multiplication and division for the following types of data :

1. Fixed point binary data in signed magnitude representation.
2. Fixed point binary data in signed 2's complement representation.
3. Floating point binary data.

8.2. REPRESENTATION OF A NUMBER

When an integer binary number is positive, the sign bit is represented by 0 and the magnitude by a positive binary number. When the number is negative, the sign is represented by 1 but the rest of the number may be represented in one of the three possible ways :

1. Signed magnitude representation.
2. Signed 1's complement representation.
3. Signed 2's complement representation.

The signed magnitude representation of a negative number consists of a magnitude and a negative sign. A positive number is represented by a sign bit 0 and the magnitude part for example, the signed positive number 14 is represented in 8 bit form then

Sign bit = 0

Magnitude = 0001110

Representation = 00001110

while a negative number can be represented by three different ways in eight bits. So –4 can be represented by the following three ways.

Signed magnitude representation	1	0000100
Signed 1's complement representation	1	1111011
Signed 2's complement representation	1	1111100

The signed magnitude representation system is used in ordinary arithmetic but is awkward when employed in computer arithmetic.

When two number of n digits each are added and the sum occupies $n + 1$ digits, we say that an overflow occurred. When the addition is performed with paper and pencil, an overflow is not a problem since there is no limit to the width of the page to write down the sum. An overflow is a problem in digital computers because the width of registers is finite. A result that contains n + 1 bits cannot be accommodated in a register with a standard length of n bits. When an overflow occurs, a corresponding flip-flop is set

If a positive number is added to a negative number, overflow does not occur because the sum is smaller than the largest of the two original numbers. An overflow occurs when numbers to be added are both positive or both negative and the result exceeds the storing capacity of the register.

Generally 2's complement representation is used when performing arithmetic operations with integers. For floating point operations, most computers use the signed magnitude representation for mantissa.

8.2.1. Fixed Point Representation

In fixed point representation all numbers are represented as integers or fraction. Signed integers or BCD numbers are referred to as fixed point numbers because they contain no information regarding the location of the binary point or decimal point. The binary or decimal point is assumed at the extreme right or left of the number. The fixed point method assumes that the binary point is always fixed in one position. The two positions most widely used are :

1. A binary point in the extreme left of the register to make the stored number a fraction.
2. A binary point in the extreme right of the register to make the stored number an integer.

The binary point is not actually present, but its presence is assumed from the fact that the number stored in the register is treated as a fraction or as an integer. If the bianry or decimal point is at the extreme right of the computer word all numbers are positive or negative integers. If the radix point is assumed to be at the extreme left, all numbers are positive or negative fraction.

Suppose we have to multiply 2.58 × 50.38. The number will be represented as 258 × 5038. The result will be 1299804. The decimal point has to be placed by the programmer to get the correct result that is 1299.804. Thus, in the fixed point representation the user has to keep the track of radix point.

In scientific applications of computers fractions are frequnetly used. So a system that keeps track about the position of binary or decimal point is required. Such a system is called floating point representation of numbers. Many computers and all modern electronic calculators use floating point arithmatic operations.

8.3. FIXED POINT ARITHMETIC

8.3.1. Addition and Subtraction

A negative fixed point binary number can be represented in signed magnitude, signed 1's complement or signed 2's complement form. Generally 2's complement representation is used when performing arithmatic operations with integers. For floating point operations, most computers use the signed magnitude representation for mantissa. In this section, we study the addition and subtraction algorithm when data is represented in signed magnitude form or when data is represented in signed 2's complement form.

It is important to realise that the adopted representation for negative numbers refers to the representation of numbers in the registers before and after the execution of the arithmetic operation. It does not mean that complement arithmetic may not be used in an intermediate step. For example, it is convenient to employ complement arithmetic when reforming a subtraction operation with numbers in signed magnitude representation. As long as the initial minuend and subtrahend have been used in an intermediate steps it does not alter the fact that the representation is in signed magnitude.

8.3.1.1. Addition and Subtraction using Signed Magnitude Data

In signed magnitude form numbers are represented in everyday arithmetic calculations. Here we explain the addition and subtraction with the magnitude of two numbers A and B. There are eight

different conditions to consider when we do addition and subtraction depending on the sign of the numbers and the operation performed. These different conditions are listed in Table 8.1. in column first. The other columns in the table show the actual operation to be performed with the magnitude of the numbers. The last column is needed to prevent a negative zero, when two numbers are equal the result should be zero.

The algorithm for addition and subtraction are derived from the table and can be stated as follows :

1. Addition Subtraction Algorithm

Table 8.1. Addition and Subtraction of signed magnitude data

Operation	*Add Magnitude*	*Subtracted magnitudes when*		
		A > B	*A < B*	*A = B*
(+A) + (+B)	+ (A + B)			
(+A) + (–B)		+ (A – B)	– (B – A)	+ (A – B)
(–A) + (+B)		– (A – B)	+ (B – A)	+ (A – B)
(–A) + (–B)	– (A + B)			
(+A) – (+B)		+ (A – B)	– (B – A)	+ (A – B)
(+A) – (–B)	+ (A + B)			
(–A) – (+B)	– (A + B)			
(–A) – (–B)		–(A – B)	+(B – A)	+(A – B)

We derive the algorithm for addition and subtraction from this above table. When the sign of A and B are identical in case of addition and when the sign of A and B are different in case of subtraction, then add the two magnitudes and attach the sign of A to the result. That are represented in add magnitude column. When the sign of A and B are different in case of addition operation and identical in case of subtraction operation then compare the magnitudes and subtract the smaller number from the larger.

If A > B

Sign of result = sign of A

If A < B

Sign of result = complement the sign of A

If the two magnitudes are equal, subtract B from A and make the sign of the result positive.

The two algorithms are similar except for the sign comparison. The procedure to be followed for identical signs in the addition algorithm is the same for different signs in the subtraction algorithm and vice versa.

2. Hardware Implementation:

To implement the addition and subtraction operation with hardware first of all we have to store these two numbers A and B in two registers. Let A and B are two registers that holds the numbers A and B. Sign of these two numbers are stored in two flip-flops As and Bs. The result of addition or subtraction can be transferred into the third register, however a saving is achieved if the result is transferred into A and As. Thus, A and As together form an accumulator register. The following hardware components are used for implementation :

1. Complementer
2. Parallel adder
3. 4 1 bit flip-flops
4. Two registers

For a parallel adder is needed to perform the microoperation A + B. A comparator circuit is needed to establish if A > B, A = B or A < B. Figure 8.2 shows a block diagram of the hardware for implementing the addition and subtraction operation. A complementor is needed for performing complement of the numbers. As we know subtraction can be accomplished by means of complement and addition, secondly the comparision can be determined from the carry after subtraction. So there is no need for comparator circuit. The figure consists of registers A and B and sign flip-flops As and Bs. Subtraction is done by adding A to the 2's complement of B. The output carry is transferred to flip-flop E, where it can be checked to determine the relative magnitudes of the two numbers. The add overflow flip-flop AVF holds the overflow bit when A and B are added. The A register provides other micro-operations that may be needed when we specify the sequence of steps in the algorithm.

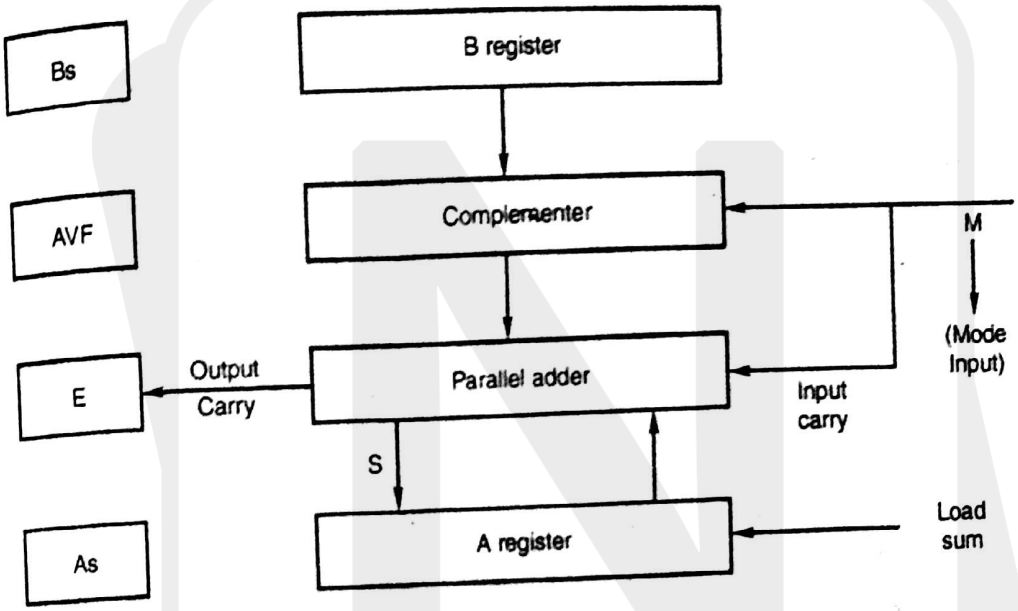


Fig. 8.2. Hardware implementation

The addition of A plus B is done through the parallel adder. The sum output of the adder is applied to the input of the A register. The complementer provides an output of B or the complement of B depending upon the value of mode input M. The complementer consists of EX-OR gates and the parallel adder consists of full adder circuits that we have studied in Chapter 3. In case of addition initial carry is assumed 0 so the M signal is also applied to the input carry of the adder.

When $M = 0$

$\text{Input Carry} = 0$

$\text{Output} = A + B$ (Addition operation)

When $M = 1$

$\text{Input Carry} = 1$

$\text{Output of B Register} = \overline{B}$

$\text{Output} = A + \overline{B} + 1 = A - B$ (Subtraction operation).

3. Hardware Algorithm/Flow Chart :

The flow chart for the hardware algorithm is shown in Fig. 8.3. First of all we have to compare the sign of A and B that are stored in flip-flops As and Bs. The two signs are compared with exclusive OR gate.

Table 8.2. Truth table for EX-OR gate

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

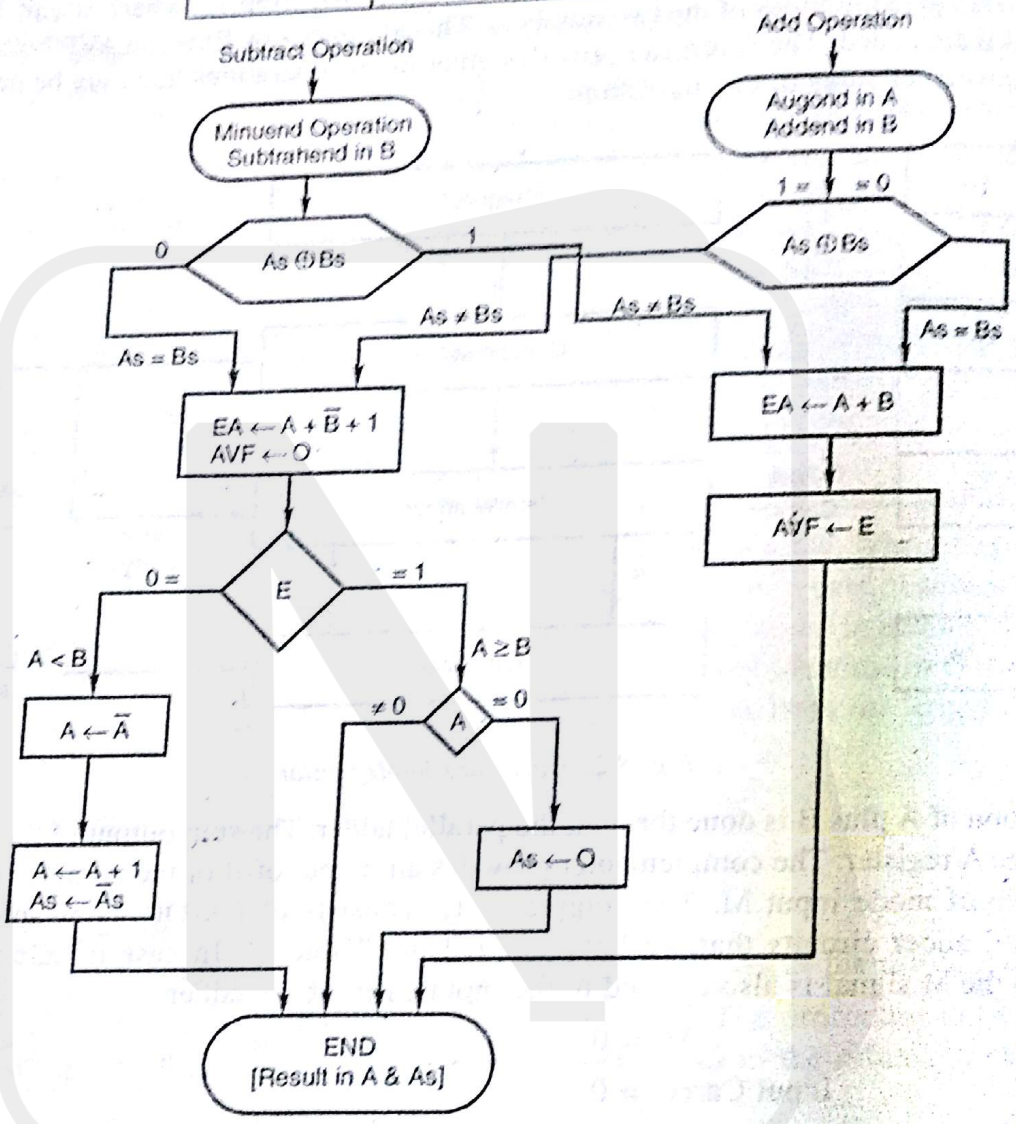


Fig. 8.3. Flow chart for add and subtract operation

If the output of the gate is 0, the signs are identical; if it is 1, the signs are different. For an *add* operation, identical signs dictate that the magnitude be added. For a *subtract* operation, different signs dictate the magnitudes be added. The magnitudes are added with a microoperation EA ← A + B, where EA is a register that combines E and A. The carry in E after the addition constitutes an overflow if it is equal to 1. The value of E is transferred into the add-overflow flip-flop AVF.

8.3.1.2. Addition and Subtraction with Signed 2's complement Data

The signed-2's complement representation of numbers together with arithmetic algorithms for addition and subtraction. They are summarized here. The leftmost bit of a binary number represents the sign : 0 to

denote to denote positive and 1 to denote negative. If the sign bit is 1, then we represent number in 2's complement form. Thus + 33 is represented as 00100000 and -33 as 11011110. Note that 11011110 is the 2's complement of 00100000, and vice versa.

The addition of two numbers in signed 2's complement form by adding the numbers with the sign bits treated the same as the other bits of the number. We discard the carry of the sign-bit position. The subtraction consists of first taking the 2's complement of the subtrahend and then adding it to the minuend.

When we add two numbers of n digits then the sum occupies n + 1 digits, in this case an overflow occurs. The effect of an overflow on the sum of two signed 2's complement numbers has been discussed already.

We can detect an overflow by inspecting the last two carries of the addition. When the two carries are applied to an exclusive-OR gate, the overflow is detected when the output of the gate is equal to 1.

The register configuration for the hardware implementation is given in Fig. 8.4. This is the same configuration as in Fig. 8.4 except that the sign bits are not separated from the rest of the registers. We call the A register AC (accumulator) and the B register BR. The two sign bits are added or subtracted together with the other bits in the complementer and parallel adder. The overflow flip-flop V is set to 1 if there is an overflow. The output carry in this case is discarded.

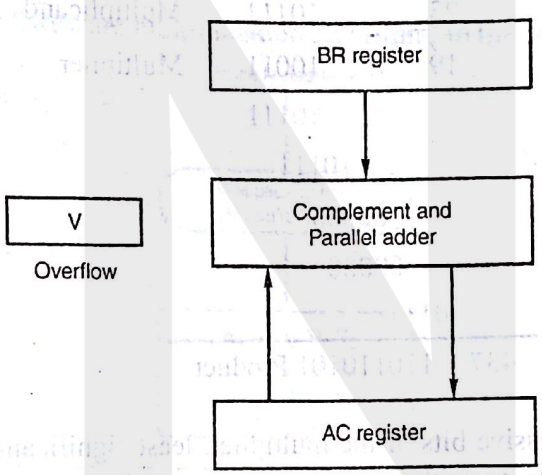


Fig. 8.4. Hardware for signed 2's complement addition and subtraction

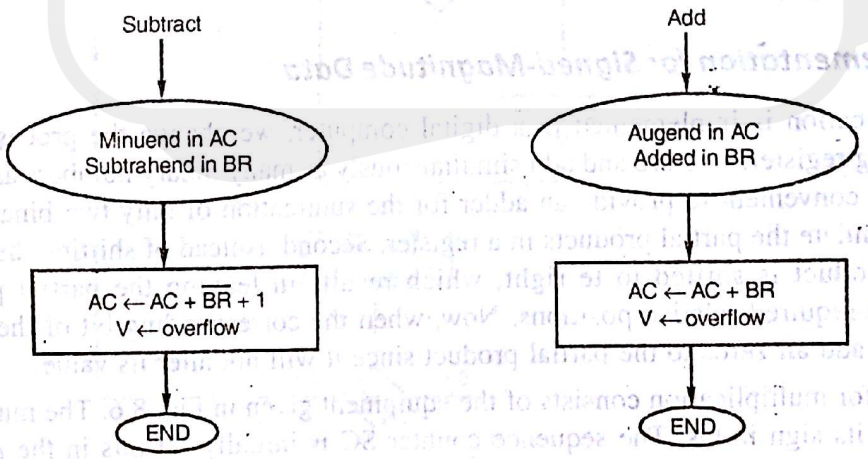


Fig. 8.5. Addition and Subtraction is in Signed 2's complement representation

The algorithm for adding and subtracting two binary numbers in signed 2's complement representation is shown in the flowchart of Figure 8.5. We obtain the sum by adding the contents of AC and BR (including their sign bits). The overflow bit V is set to 1 if the exclusive OR of the last two carries is 1, otherwise it is cleared. The subtraction operation is performed by adding the content of AC to the 2's complement of BR. Taking the 2's complement of BR has the effect of changing a positive number to negative, and vice versa. We have to check an overflow during this operation because the two numbers added may have the same sign. It should be noted that if an overflow occurs, there is an erroneous result in the AC register.

If we compare this algorithm with its signed-magnitude part, we note that it is much simpler to add and subtract numbers if we keep negative numbers in signed 2's complement representation. Therefore most computers adopt this representation over the more familiar signed-magnitude.

8.3.2. Multiplication

Multiplication of two fixed-point binary numbers in signed magnitude representation is done with paper and pencil by a process of successive shift and add operations. This process is best illustrated with a numerical example:

23		10111	Multiplicand
19	×	10011	Multiplier
		10111	
		10111	
		00000	
		00000	
		10111	
437		110110101	Product

This process looks at successive bits of the multiplier, least significant bit first. If the multiplier bit is 1, the multiplicand is copied as it is; otherwise, we copy zeros. Now we shift numbers copied down one position to the left from the previous numbers. Finally, the numbers are added and their sum produces the product.

Hardware Implementation for Signed-Magnitude Data

When multiplication is implemented in a digital computer, we change the process slightly. Here, instead of providing registers to store and add simultaneously as many binary numbers as there are bits in the multiplier, it is convenient to provide an adder for the summation of only two binary numbers, and successively accumulate the partial products in a register. Second, isntead of shifting the multiplicand to left, the partial product is shifted to te right, which results in leaving the partial product and the multiplicand in the required relative positions. Now, when the corresponding bit of the multiplier is 0, there is no need to add all zeros to the partial product since it will not alter its value.

The hardware for multiplication consists of the equipment given in Fig. 8.6. The multiplier is stored in the register and its sign in Qs. The sequence counter SC is initially set bits in the multiplier. After forming each partial product the counter is decremented. When the content of the counter reaches zero, the product is complete and we stop the process.

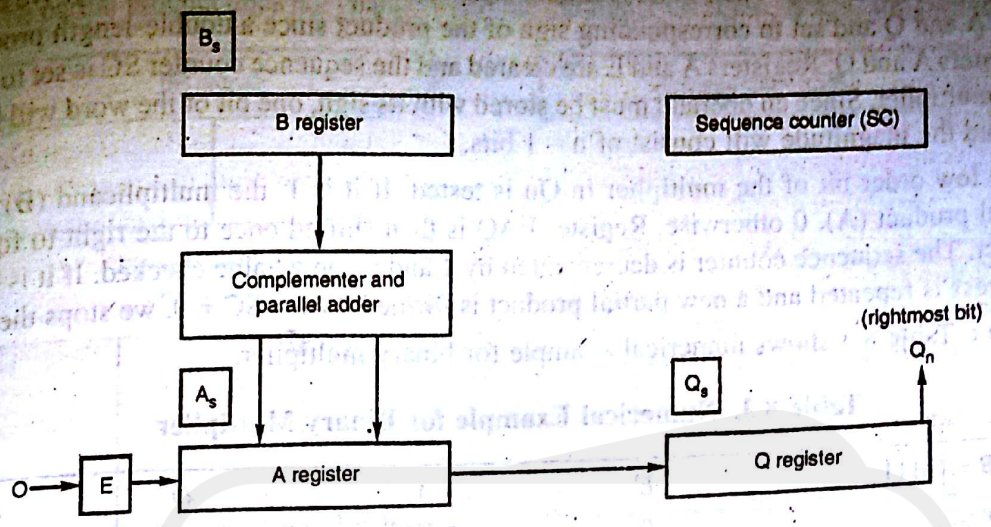


Fig. 8.6. Hardware for multiply operation

Hardware Algorithm/Flow Chart

Figure 8.7 is a flow chart of the hardware multiplication algorithm. In the beginning, the multiplicand

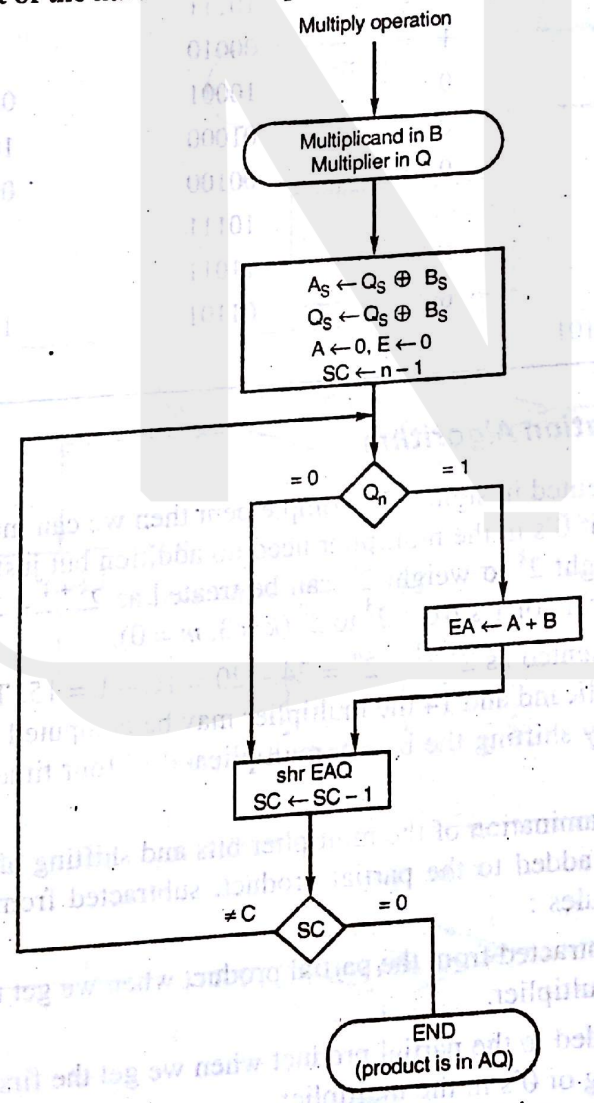


Fig. 8.7. Flowchart for multiply operation

is in B and the multiplier in Q. Their corresponding signs are in Bs and Qs respectively. We compare the signs of both A and Q and set to corresponding sign of the product since a double-length product will be stored in registers A and Q. Registers A and E are cleared and the sequence counter SC is set to the number of bits of the multiplier. Since an operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will consist of n – 1 bits.

Now, the low order bit of the multiplier in Qn is tested. If it is 1, the multiplicand (B) is added to present partial product (A), 0 otherwise. Register EAQ is then shifted once to the right to form the new partial product. The sequence counter is decremented by 1 and its new value checked. If it is not equal to zero, the process is repeated and a new partial product is formed. When SC = 0, we stops the process.

Example : Table 8.3 shows numerical example for binary multiplier.

Table 8.3. Numerical Example for Binary Multiplier

	E	*A*	*Q*	*SC*
Multiplicand $B = 10111$				
Multiplier in Q	0	00000	10011	101
$Q_n = 1$; add B		10111		
First partial product	0	10111		
Shift right EAQ	0	01011	11001	100
Qn = 1; add B		10111		
Second partial product	1	00010		
Shift right EAQ	0	10001	01100	011
Qn = 0; shift right EAQ	0	01000	10110	010
Qn = 0; shift right EAQ	0	00100	01011	001
Qn = 1; add B		10111		
Fifth partial product	0	11011		
Shift right EAQ	0	01101	10101	000
Final product in AQ = 0110110101				

8.3.2.1. Booth Multiplication Algorithm

If the numbers are represented in signed 2's complement then we can multiply them by using Booth algorithm. In fact the strings of 0's in the multiplier need no addition but just shifting, and a string of 1's in the multiplier from bit weight 2^k to weight 2^m can be treated as $2^{k+1} - 2^m$. For example, the binary number 001111 (+15) has a string of 1's from 2^3 to 2^0 ($k = 3$, $m = 0$).

The number can be represented as $2^{k+1} - 2^m = 24 - 20 = 16 - 1 = 15$. Therefore, the multiplication $M \times 14$, where M is the multiplicand and 14 the multiplier may be computed as $M \times 24 - M \times 21$. That is, the product can be obtained by shifting the binary multiplicand M four times to the left and subtracting M shifted left once.

Booth algorithm needs examination of the multiplier bits and shifting of the partial product. Prior to the shifting, the multiplicand added to the partial product, subtracted from the partial product, or left unchanged by the following rules :

1. The multiplicand is subtracted from the partial product when we get the first least significant 1 in a string of 1's in the multiplier.
2. The multiplicand is added to the partial product when we get the first Q (provided that there was a previous 1) in a string of 0's in the multiplier.

3. The partial product does not change when the multiplier bit is the same as the previous multiplier bit.

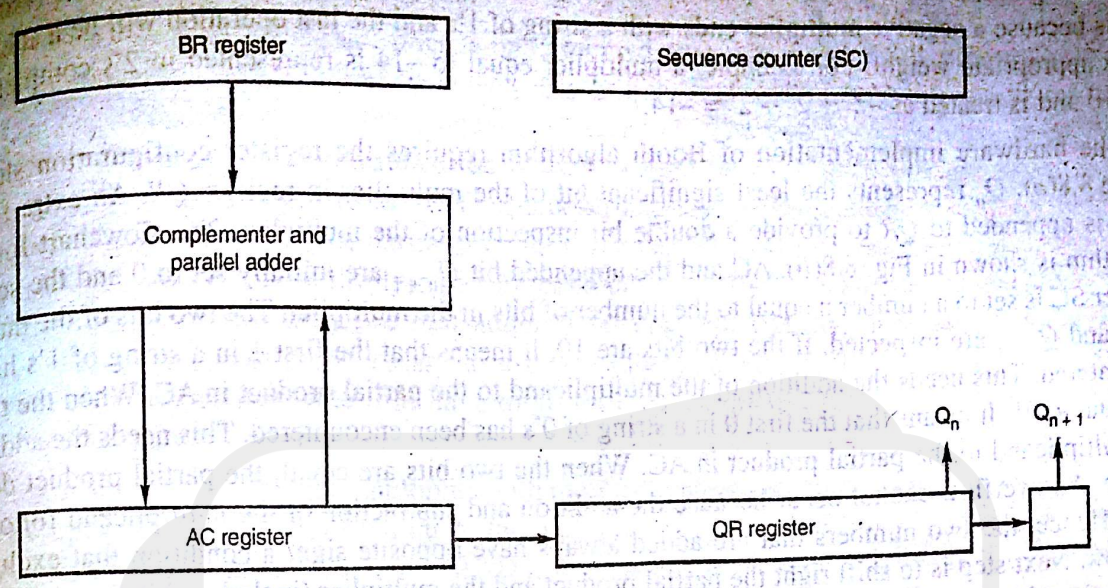


Fig. 8.8(a). Hardware for Booth algorithm

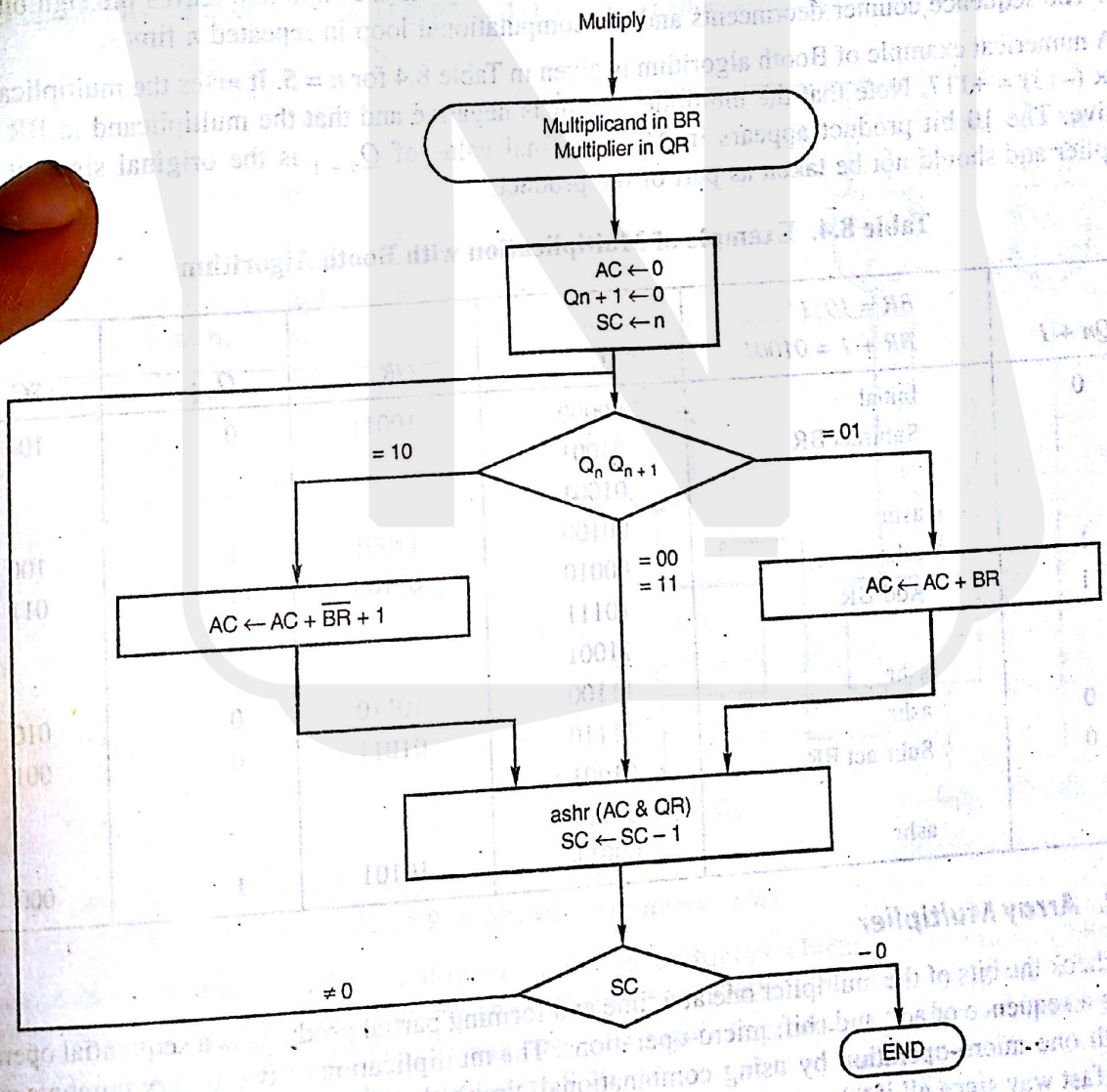


Fig. 8.8(b).

The algorithm applies to both positive and negative multipliers in 2's complement representation. This is because a negative multiplier ends with a string of 1's and the last operation will be a subtraction of the appropriate weight. For example, a multiplier equal to −14 is represented in 2's complement as 110010 and is treated as $-2^4 + 2^2 - 2^1 = -14$.

The hardware implementation of Booth algorithm requires the register configuration shown in Figure 8.8(*a*). Q_n represents the least significant bit of the multiplier in register *QR*. An extra flip-flop Q_{n+1} is appended to *QR* to provide a double bit inspection of the multiplier. The flowchart for Booth algorithm is shown in Fig. 8.8(*b*). AC and the appended bit Q_{n+1} are initially set to 0 and the sequence counter SC is set to a number n equal to the number of bits in the multiplier. The two bits of the multiplier in Q_n and Q_{n+1} are inspected. If the two bits are 10, it means that the first 1 in a string of 1's has been encountered. This needs the addition of the multiplicand to the partial product in AC. When the two bits are equal to 01. It means that the first 0 in a string of 0's has been encountered. This needs the addition of the multiplicand to the partial product in AC. When the two bits are equal, the partial product does not change. An overflow cannot occur because the addition and subtraction of the multiplicand follow each other. Hence, the two numbers that are added always have opposite sign, a condition that excludes an overflow. Next step is to shift right the partial product and the multiplier (including bit Q_{n+1}). This is an arithmetic shift right (ashr) operation which shifts AC and QR to the right and leaves the sign bit in AC same. The sequence counter decrmeents and the computational loop in repeated *n* times.

A numerical example of Booth algorithm is given in Table 8.4 for $n = 5$. It gives the multiplication of $(-9) \times (-13) = +117$. Note that the multiplier in QR is negative and that the multiplicand in BR is also negative. The 10 bit product appears in AC. The final value of Q_{n+1} is the original sign bit of the multiplier and should not be taken as part of the product.

Table 8.4. Example of Multiplication with Booth Algorithm

Qn	*Qn + 1*	*BR = 10111* *BR + 1 = 01001*	*AC*	*QR*	Q_{n+1}	*SC*
1	0	Initial	00000	10011	0	101
		Subtract BR	01001			
			01001			
		ashr	00100	11001	1	100
1	1	ashr	00010	01100	1	011
0	1	Add BR	10111			
			11001			
		ashr	11100	10110	0	010
0	0	ashr	11110	01011	0	001
1	0	Subtract BR	01001			
			00111			
		ashr	00011	10101	1	000

8.3.2.2. Array Multiplier

To check the bits of the multiplier one at a time and forming partial products is a sequential operation requiring a sequence of add and shift micro-operations. The multiplication of two binary numbers can be done with one micro-operation by using combinational circuit that forms the product bits all at once. This is a fast way since all it takes is the time for the signals to propagate through the gates that form the

multiplication array. However, an array multiplier requires a large number of gates, and so it is not an economical unit for the development of ICs.

Now we see how an array multiplier is implemented with a combinational circuit. Consider the multiplication of two 2-bit numbers as shown in Fig. 8.9. The multiplicand bits are b1 and b0, the multiplier bits are a1 and a0, and the product is c3 c2 c1 c0. The first partial product is obtained by multiplying a0 by b1b0. The multiplication of two bits gives a 1 if both bits are 1; otherwise, it produces a 0. This is identical to an AND operation and can we implement it with an AND gate. As shown in the diagram, the first partial product is formed by means of two AND gates. The second partial product is formed by multiplying a1 by b1b0 and is shifted one position to the left. The two partial products are added with two half-adder (HA) circuits. Usually, there are more bits in the partial products and it will be necessary to use full-adders to produce the sum. Note that the least significant bit of the product does not have to go through an adder since it is formed by the output of the first AND gate.

A combinational circuit binary multiplier with more bits can be constructed in a similar fashion. A bit of the multiplier is ANDed with each bit of the multiplicand in as many levels as there are bits in the multiplier. The binary output in each level AND gates is added in parallel with the partial product of the previous level to form a new partial product. The last level produces the product. For j multiplier bits and k multiplicand bits we need j * k AND gates and (j – 1) k-bit adders to produce a product of j + k bits.

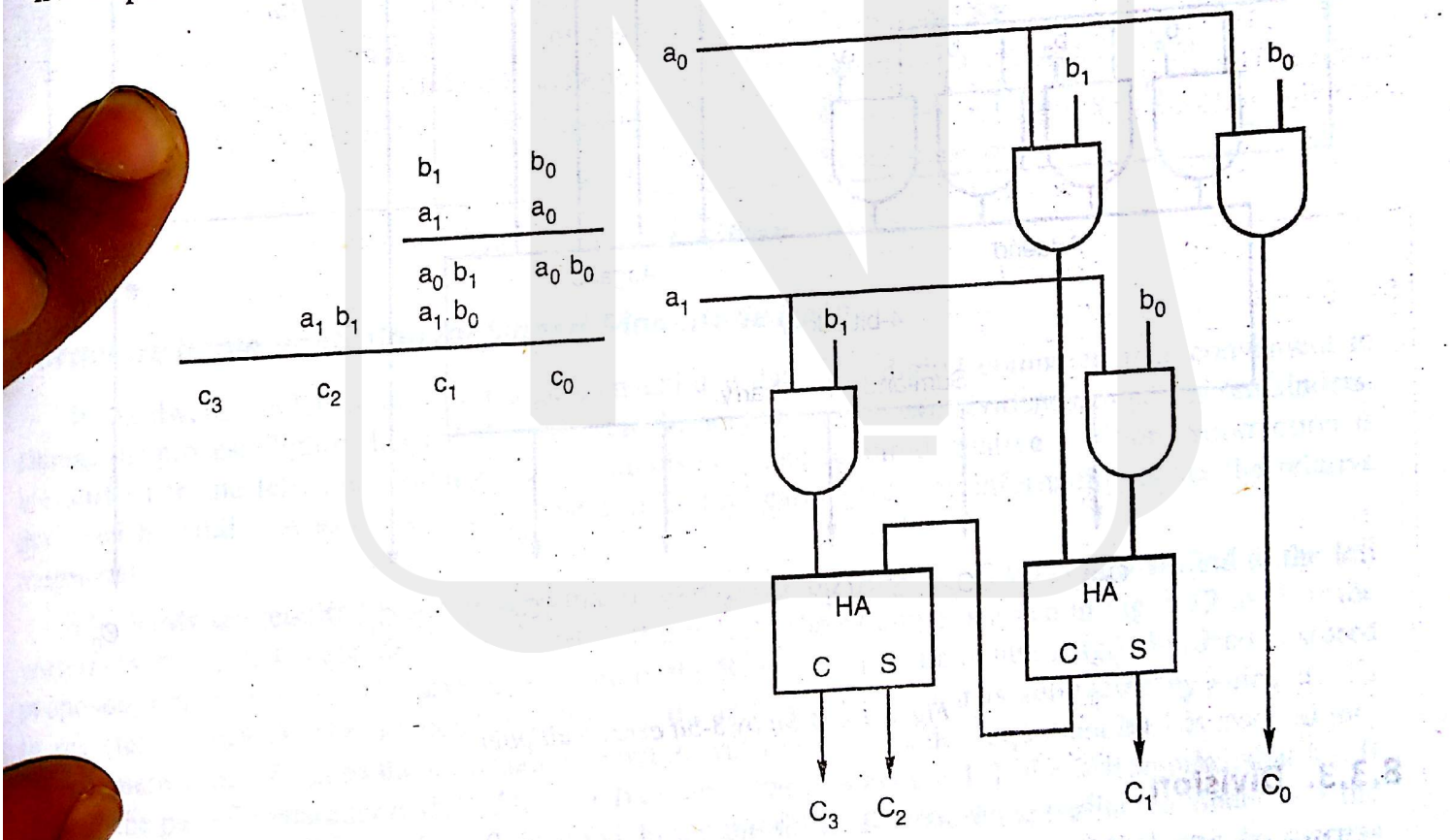


Fig. 8.9. 2-bit by 2-bit array multiplier

As a second example, consider a multiplier circuit that multiplies a binary number of four bits with a number of three bits. Let the multiplicand be represented by b3b2b1b0 and the multiplier by a2a1a0. Since k=4 and j=3, we need 12 AND gates and two 4-bit adders to produce a product of seven bits. The logic diagram of the multiplier is shown in Figure 8.10.

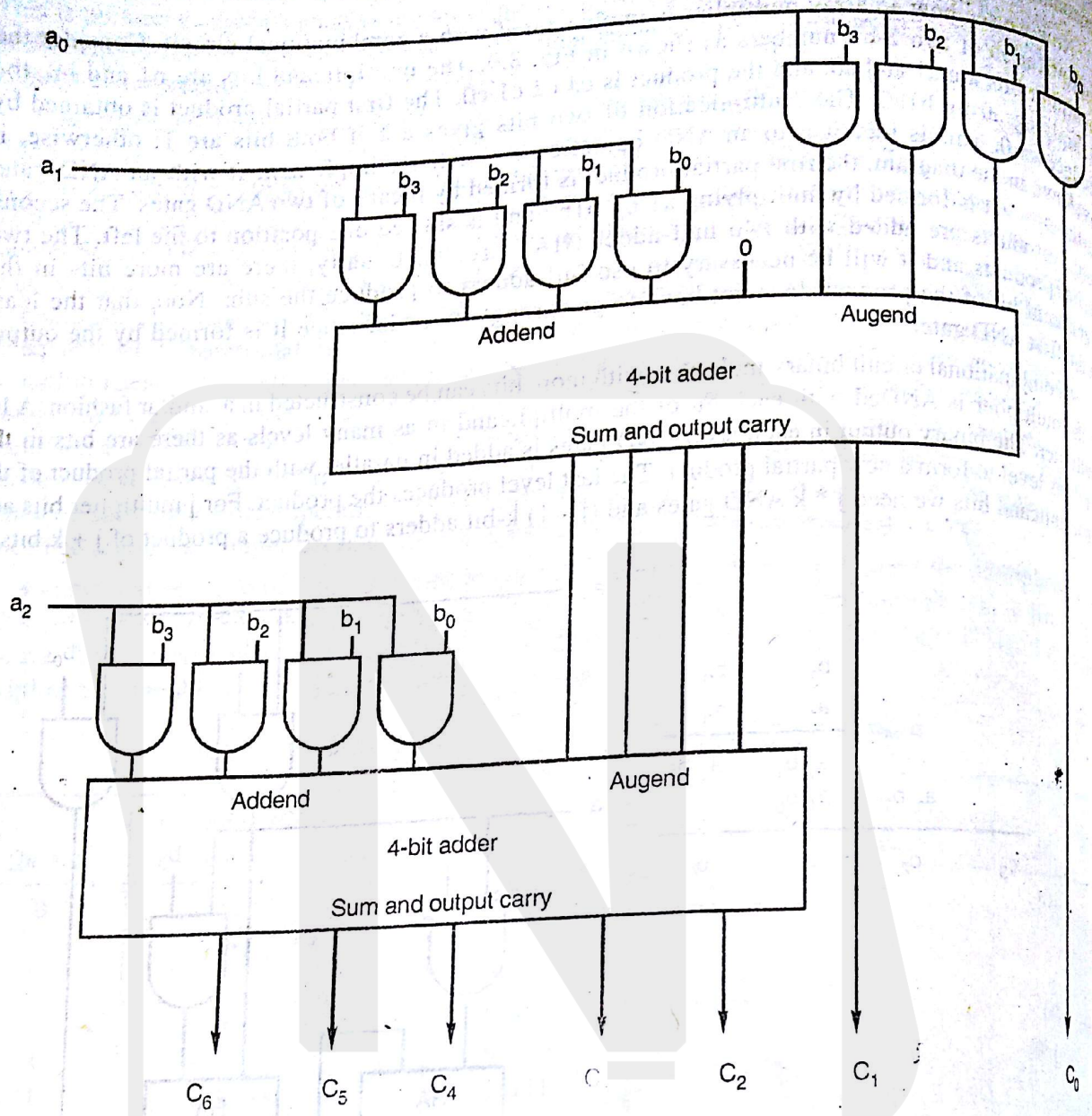


Fig. 8.10. 4-bit by 3-bit array multiplier

8.3.3. Division

Division of two fixed-point binary numbers in signed magnitude representation is performed with paper and pencil by a process of successive compare, shift and subtract operations. Binary division is much simpler than decimal division because here the quotient digits are either 0 or 1 and there is no need to estimate how many times the dividend or partial remainder fits into the divisor. The division process is described in Figure 8.11. The divisor B has five bits and the dividend A has ten.

The divisor is compared with the five most significant bits of the dividend. Since the 5-bit number is smaller than B, we again repeat the same process. Now the 6-bit number is greater than B, so we place a 1

for the quotient bit in the sixth position above the dividend. Now we shift the divisor once to the right and subtract it from the dividend. The difference is known as a partial remainder because the division could have stopped here to obtain a quotient of 1 and a remainder equal to the partial remainder. Comparing a partial remainder with the divisor continues the process. If the partial remainder is greater than or equal to the divisor, the quotient bit is equal to 1. The divisor is then shifted right and subtracted from the partial remainder. If the partial remainder is smaller than the divisor, the quotient bit is 0 and no subtraction is needed. The divisor is shifted once to the right in any case. Obviously the result gives both a quotient and a remainder.

Division : B = 10001	11010) 011000000 01110 011100 - 10001 - 010110 - - 10001 - - 001010 010100 - - - - 10001 - - - - 000110 - - - - - 00110	Quotient – Q Dividend = A 5 bits of A < B. quotient has 5 bits 6 bits of A ≥ B Shift right B and subtract; enter 1 in Q 7 bits of remainder ≥ B Shift right B and subtract; enter 1 in Q Remainder < B; enter O in Q; shift right B Remainder ≥ B Shift right B and subtract; enter 1 in Q Remainder < B; enter 0 in Q Final remainder

Fig. 8.11. Example of Binary division

Hardware Implementation for Signed-Magnitude Data

In hardware implementation for signed-magnitude data in a digital computer, it is convenient to change the process slightly. Instead of shifting the divisor to the right, two dividends, or partial remainders, are shifted to the left, thus leaving the two numbers in the required relative position. Subtraction is achieved by adding A to the 2's complement of B. End carry gives the information about the relative magnitudes.

The hardware required is identical to that of multiplicaiton. Register EAQ is now shifted to the left with 0 inserted into Qn and the previous value of E is lost. The example is given in Fig. 8.12 to clear the proposed divisor process. The divisor is stored in the B register and the doube-length dividend is stored in registers A and Q. The dividend is shifted to the left and the divisor is subtracted by ading its 2's complement value. E keeps the information about the relative magnitude. A quotient bit 1 is inserted into Qn and the partial remainder is shifted to the left to repeat the process E = 1. If E = 0, it signifies that A < B so that the quotient in Qn remains a 0 (inserted during the shift). To restore the partial remainder in A the value of B is then added to its previous value. The partial remainder is shifted to the left and the process is repeated again until we get all five quotient-bits. Note that while the partial remainder is shifted left, the quotient bits are shifted also and after five shifts, the quotient is in Q and A has the final remainder.

Before showing the algorithm in flowchart form, we have to consder the sign of the result and a possible overflow condition. The sign of the quotient is obtained from the signs of the dividend and the divisor. If the two signs are same, the sign of the quotient is plus. If they are not identical, the sign is minus. The sign of the remainder is the same as that of the dividend.

Divisor B = 10001 $\quad \bar{B} + 1 = 01111$

	E	A	Q	SC
Dividend:		01110	00000	5
shl EAQ	0	11100	00000	
add $\bar{B} + 1$		01111		
E = 1	1	01011		
Set $Q_n = 1$	1	01011	00001	4
shl EAQ	0	10110	00010	
Add $\bar{B} + 1$		01111		
E = 1	1	00101		
Set $Q_n = 1$	1	00101	00011	3
shl EAQ	0	01010	00110	
Add $\bar{B} + 1$		01111		
E = Q; leave $Q_n = 0$	0	11001	00110	
Add B		10001		2
Restore remainder	1	01010		
shl EAQ	0	10100	01100	
Add $\bar{B} + 1$		01111		
E = 1	1	00011		
Set $Q_n = 1$	1	00011	01101	1
Shl EAQ	0	00110	11010	
Add $\bar{B} + 1$		01111		
E = 0; leave $Q_n = 0$	0	10101	11010	
Add B		10001		
Restore remainder	1	00110	11010	0
Neglect E				
Remainder in A:		00110		
Quotient in Q:			11010	

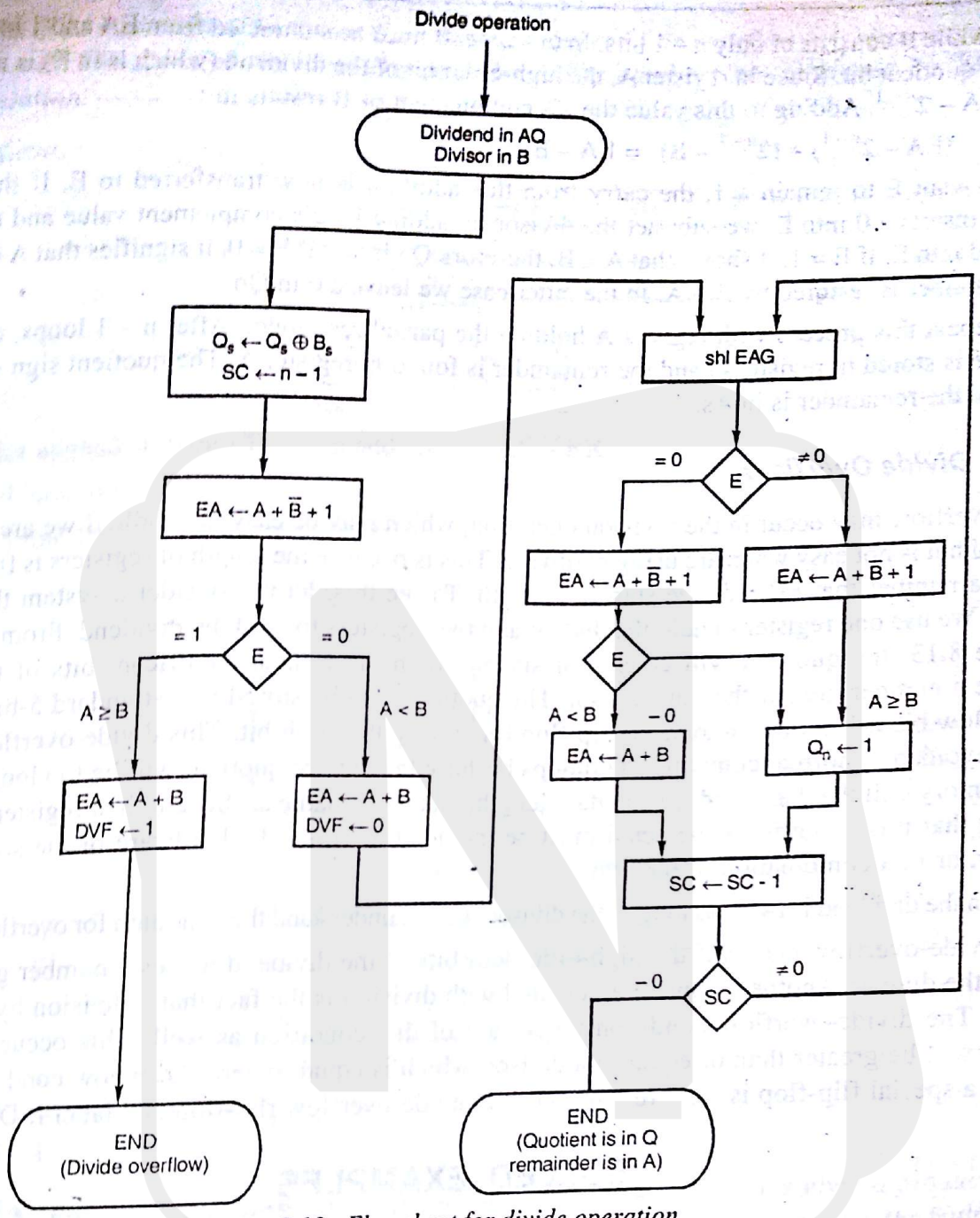


Fig. 8.13. Flowchart for divide operation

Hardware Algorithm/Flow Chart

The hardware divide algorithm is given in Fig. 8.13. A and Q contain the dividend and B has the divisor. The sign of the result is transferred into Q. A constant is set into the sequence counter SC to specify the number of bits in the quotient. As in multiplication, we assume that operands are transferred to registers from a memory unit that has words of n bits. Since an operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will have $n - 1$ bits.

We can check a divide-overflow condition by subtracting the divisor (B) from half of the bits of the dividend stored (A). If A < B, the divide-overflow occur and the operation is terminated. If A ≥ B, no divide overflow occurs and so the value of the dividend is restored by adding B to A.

The division of the magnitudes begins by shifting the dividend in AQ to the left with the high-order bit shifted into E. If the bit shifted into E is 1, we know that EA > B because EA consists of 1 followed by

$n-1$ bits while B consists of only $n-1$ bits. In this case, B must be subtracted from EA and 1 inserted into Qn for the quotient bit. Since in register A, the high-order bit of the dividend (which is in E) is missing the vlaue is $EA - 2^{n-1}$. Adding to this value the 2's complement of B results in :

$$(EA - 2^{n-1}) + (2^{n-1} - B) = EA - B$$

If we want E to remain a 1, the carry from this addition is now transferred to E. If the shift-left operation inserts a 0 into E, we subtract the divisor by adding its 2's complement value and the carry is transferred into E. If E = 1, it shows that A < B, therefore Qn is set. If E = 0, it signifies that A < B and the original number is restored by B + A. In the latter case we leave a 0 in Qn.

We repeat this process with register A holding the partial remainder. After n – 1 loops, the quotient magnitude is stored in register Q and the remainder is found in register A. The quotient sign is in Qs and the sign of the remainder is in As.

8.3.3.1. Divide Overflow

An overflow may occur in the division operation, which may be easy to handle if we are using paper and pencil but is not easy when are using hardware. This is because the length of registers is finite and will not hold a number that exceeds the standard length. To see this, let us consider a system that has 5 bit registers. We use one register to hold the divisor and two registers to hold the dividend. From the example of Figure 8.13, the quotient will consist of six bits if the five most significant bits of the dividend constitute a number greater than the divisor. The quotient is to be stored in a strandard 5-bit register, so the overflow bit will require one more flip-flop for storing the sixth bit. This divide-overflow condition must be avoided in normal computer operations because the entyire quotient will be too long for transfer into a memory unit that has words of standard length, that is, the same as the length of registers. Provisions to ensure that this condition is detected must be included in either the hardware or the software of the computer, or in a combination of the two.

When the dividend is twice as long as the divisor, we can understand the condition for overflow as follows:

A divide-overflow occurs if the high-order half bits of the dividend makes a number greater than or equal to the divisor. Another problem associated with division is the fact that a division by zero must be avoided. The divide-overflow condition takes care of this condition as well. This occurs because any dividend will be greater than or equal to a divisor, which is equal to zero. Overflow condition is usually detected a special flip-flop is set. We will call it a divide-overflow flip-flop and label it DVF.

SOLVED EXAMPLES

Example 8.1. *Perform the arithmetic operations below with binary numbers and with negative numbers in signed-2's complement representation. Use seven bits to accommodate each number together with its sign. In each case, determine if there is an overflow by checking the carries into and out of the sign bit position.*

(a) (+35) + (+40) *(b) (–35) + (–40).*

Solution.

(a)

+35	0 100011
+40	0 101000
+75	1 001011

F = 0, E = 1

F ⊕ E = 1 ; overflow

(b)

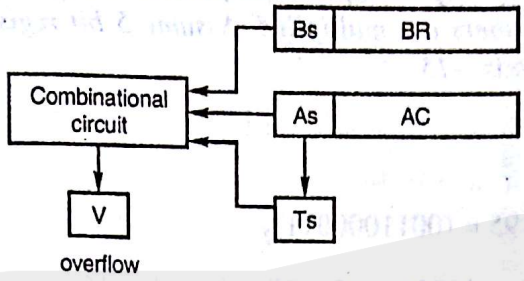
–35	1 011101
–40	1 011000
–70	0 110101

Carries → F = 1, E = 0

F ⊕ E = 1 ; overflow

Example 8.2. *Formulate a hardware procedure for detecting an overflow by comparing the sign of the sum with the signs of the augend and addend. The numbers are in signed-2's complement representation.*

Solution.



Transfer augend sign into Ts. Then add : AC ← AC + BR

As will have sign of sum.

Truth Table for combinational circuit

T_S	B_S	A_S	V	
0	0	0	0	
0	0	1	1	Change of sign
0	1	0	0	
0	1	1	0	quantities subtracted
1	0	0	0	
1	0	1	0	
1	1	0	1	Change of sign
1	1	1	0	

Boolean function for circuit :

$V = T'_s B'_s A_s + T_s B_s A'_s$

Example 8.3. *Prove that the multiplication of two n-digit numbers in base r gives a product no more than 2n digits in length. Show that this statement implies that no overflow can occur in the multiplication operation.*

Solution. Maximum value of numbers is $r^n - 1$. It is necessary to show that maximum product is less than or equal to $r^{2n} - 1$. Maximum product is :

$$(r^n - 1)(r^n - 1) = r^{2n} - 2r^n + 1 \le r^{2n} - 1$$

which gives : $2 \le 2r^n$ or $1 \ge r^n$

This is always true since $r \ge 2$ and $n \ge 1$

Example 8.4. *Show that adding B after the operation $A + \overline{B} + 1$ restores the original value of A. What should be done with the end carry ?*

Solution. $A + \overline{B} + 1$ performs : $A + 2^n - B = 2^n + A - B$

adding B : $(2^n + A - B) + B = 2^n + A$

remove end-carry 2^n to obtain A.

Example 8.5. *Show the step-by-step multiplication process using Booth algorithm (as in Table 8.3) when the following binary numbers are multiplied. Assume 5 bit registers that hold signed numbers. The multiplicand in both cases is +15.*

(a) (+15) × (+13)

(b) (+15) × (–13)

Solution. $(+15) \times (+13) = +195 = (0011000011)_2$

BR = 01111 (+15); $\overline{BR} + 1$ = 10001 (–15); QR = 01101 (+15)

Q_n	Q_{n+1}		AC	QR	Q_{n+1}	SC
		Initial	00000	01101	0	101
1	0	Subtract BR	10001			
			10001			
		ashr	11000	10110	1	100
0	1	Add BR	01111			
			00111			
		ashr	00011	11011	0	011
1	0	Subtract BR	10001			
			10100			
		ashr	11010	01101	1	010
1	1	ashr	11101	00110	1	001
0	1	Add BR	01111			
			01100			
		ashr	00110	00011	0	000
			+ 195			

(*b*) $(+15) \times (-13) = -195 = (1100111101)_{2\text{'s css}}$

BR = 011111 (+15); $\overline{BR} + 1$ = 10001 (–15); QR = 10011 (–13)

Q_n	Q_{n+1}		AC	QR	Q_{n+1}	SC
		Initial	00000	10011	0	101
1	0	Subtract BR	10001			
			10001			
		ashr	11000	11001	1	100
1	1	ashr	11100	01100	1	011
0	1	Add BR	01111			
			01011			
		ashr	00101	10110	0	010
0	0	ashr	00010	11011	0	001
1	0	Subtract BR	10001			
			10011			
		ashr	11001	11101	1	010
			–195			