2.1 Basic Processing Unit

The information's accepted from the users are processed with the help of processing unit. The instruction set of a processor may vary depends on the architecture. In this section we are dealing with how the instructions are processed within the processing unit.

2.1.1 Fundamental Concepts

To execute a program, the processor fetches one instruction at a time and performs the specified operations. Program Counter (PC) and Instruction Register (IR) are the main CPU registers who is participating in this task. PC contains the address of the instruction which is going to be executed next. The decoded instruction which is going to be currently executed is stored in register IR. The following are the general steps used to execute an instruction:

- Fetch the contents of memory location pointed by PC. This is actually referred as the instruction to be executed. Then they are loaded into IR. This can be symbolically represented as:

$$IR \leftarrow [[PC]]$$

- Update the value of PC. Suppose that every instruction has 4 bytes. Then PC will be updated by 4. Symbolically it can be represented as;

$$PC \leftarrow [PC] + 4$$

(The content of PC will be incremented by 4)

- Carry out the action specified in IR.

The fig 2.1 shows the single bus organization of data path inside the CPU. The ALU registers and the internal processor bus together known as "data path". The internal processor bus is used for communication between the various units inside the CPU. A separate external memory bus will be there to communicate between processor and memory.

Normally in the instruction execution following operations may be happen in different sequences:

- Register transfer
- Performing an arithmetic and logic operation
- Fetching a word from memory
- Storing a word in memory

The order of these operations may be different in separate sequences. In some of the operations, some steps may not be necessary too. Here, we are discussing each of these operations in detail.

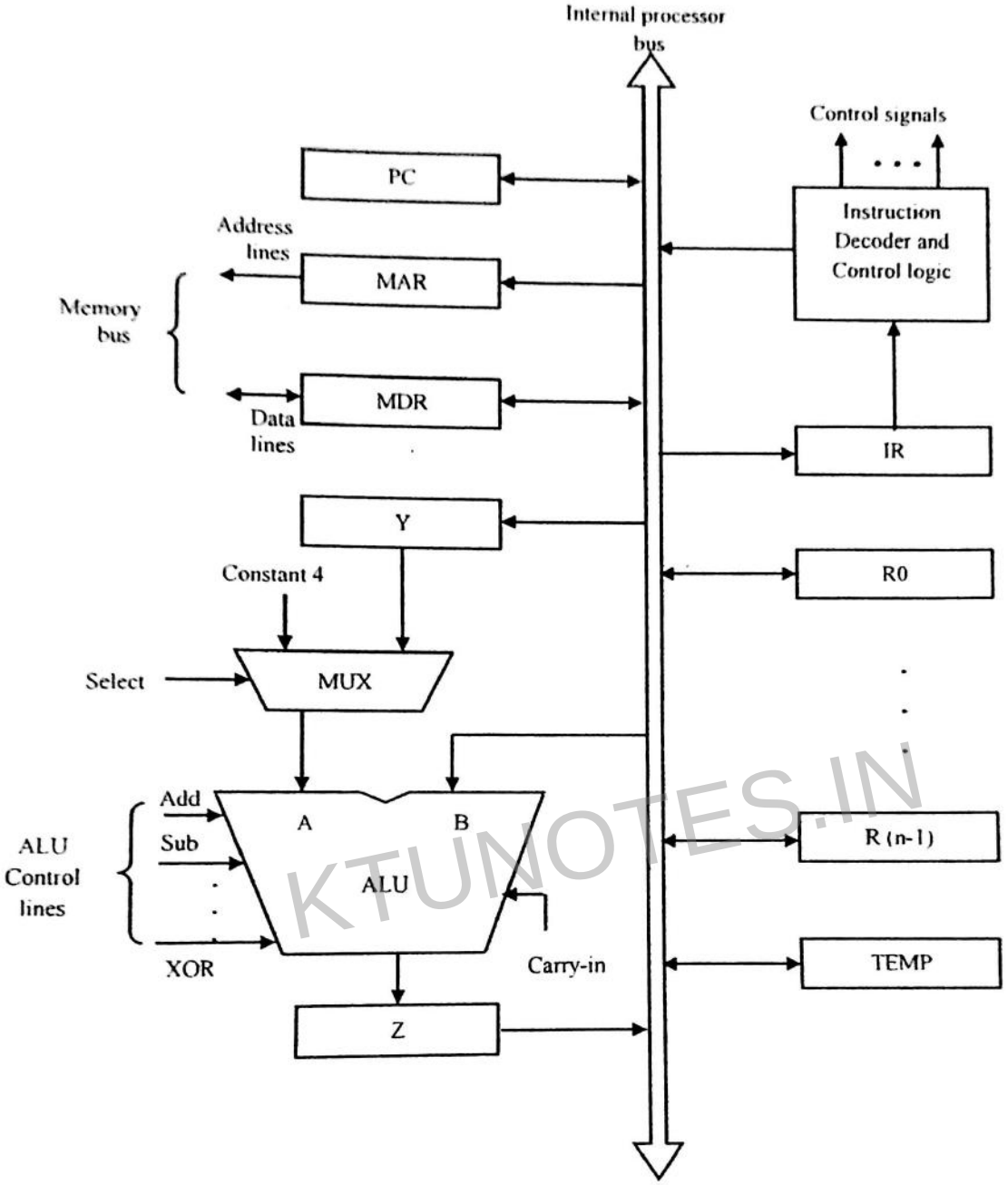


Fig 2.1: Single-bus organization of the datapath inside a processor

a) Register transfer

Associating with each registers there are two signals in and out. The input and output of the register R_i are connected to the bus via switches controlled by R_{in} and R_{iout}. When R_{in} is set to 1 the data on the bus are loaded in R_i. If R_{iout} is set to 1 the contents registers R_i are placed on bus.

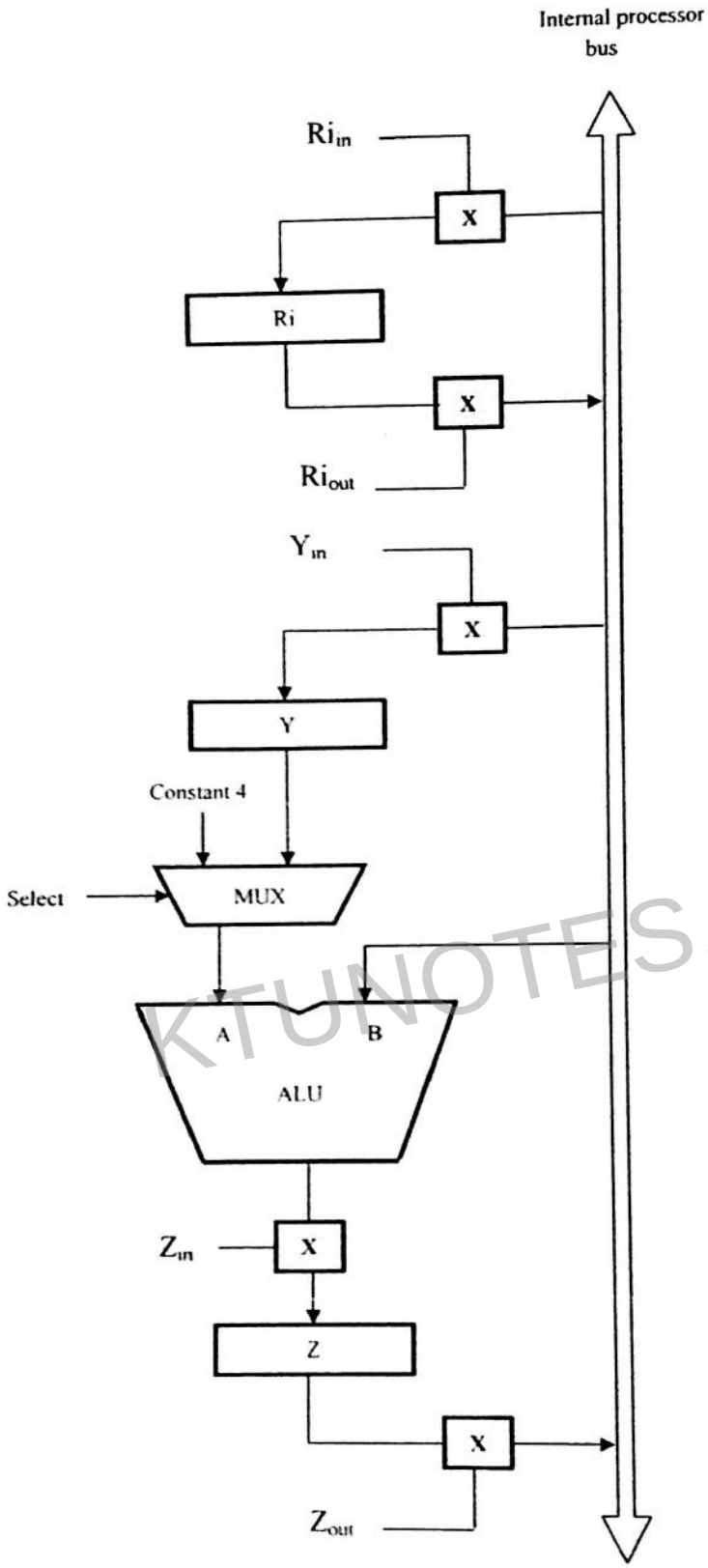


Fig 2.2: Input and output gating for the registers in Fig 2.1

Example

Transfer the contents of register R_1 to R_4.

- Set R_{1out} to 1. This places the contents of R_1 to the bus.

- Set R_{4in} to 1. This loads data from the processor bus to register R_4

b) Performing an arithmetic and logic operation

ALU is a combinational circuit and has no internal storage. It performs arithmetic and logic operations on the two operands applied to its A and B inputs. In the above figure we can see that, one of the operand is the output of the MUX and the other operand is obtained directly from the bus. The result produced by ALU is temporarily stored in register 'Z'. Therefore a sequence of operations to add the contents of register R_1 to those of register R_2 and store the result in register R3 is:

- R_{1out} , Y_{in}
- R_{2out} ,Select Y ,add, Z_{in}
- Z_{out} , R_{3in}

Example

Subtract the contents of R4 from R_5 and store the result in R_6.

- R_{4out} , Y_{in}
- R_{5out} ,Select Y ,sub, Z_{in}
- Z_{out} , R_{6in}

c) Fetching a word from memory

To fetch a word of information from the memory the processor has to specify the memory location, where this information is stored and requests a read operation. This applies whether the information to be fetched represents an instruction in a program or an operand specified by an instruction. The processor transfers the required address to MAR whose output is connected to address lines of external memory bus. At the same time the processor uses the control lines of the memory bus to indicate that a read operation is needed. When the requested data is received from the memory they are stored in the register MDR. From there they can be transferred to other registers in the processor. The following figure shows the connection and control signals for register MDR.

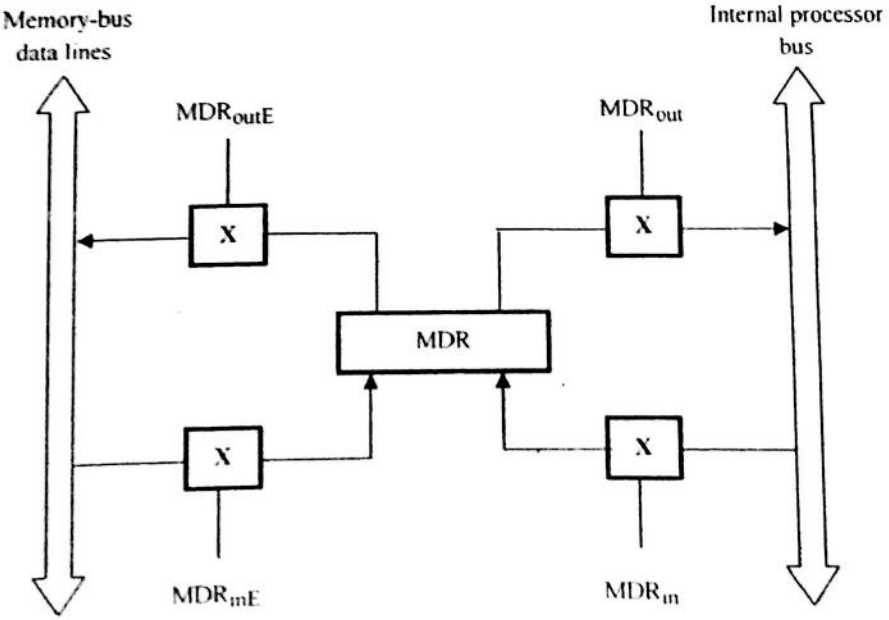


Fig 2.3: Connection and control signals for register MDR

Example MOVE (R_1), R_2

$R1_{out}$, MAR_{in}, READ
MDR_{inE}, WMFC
MDR_{out}, $R2_{in}$

MFC (Memory Function Complete) signal is used for indicating that the requested read or write operation has been completed.

d) Storing a word in memory

Writing a word into a memory location follows a similar procedure. The desired address will be loaded in MAR, the data to be written are loaded in MDR, and then a write command is issued.

Example MOVE R2, (R1)

$R1_{out}$, MAR_{in}
$R2_{out}$, MDR_{in}, WRITE
MDR_{outE}, WMFC

The action specified in one step can be completed in one clock cycle. Exemption may come for some steps like MFC depending on the speed of the addressed device.

2.1.2 Instruction Cycle

An instruction cycle (sometimes called a fetch–decode–execute cycle) is the basic operational process of a computer. It is the process by which a computer retrieves a program instruction from its memory, determines what actions the instruction dictates, and carries out those actions. In simpler CPUs the instruction cycle is executed sequentially, each instruction being processed before the next one is started. In most modern CPUs the instruction cycles are executed concurrently, and often in parallel, through an instruction pipeline: the next instruction starts being processed before the previous instruction has finished, which is possible because the cycle is broken up into separate steps.

The main steps in the instruction cycle are Fetch and Execution. Initially an instruction is fetched from the memory unit based on the address contained in PC. The PC value is updated by a value of size of the instruction so that PC can point to the next instruction to be executed. Then the decoded instruction is placed in IR, which is ready to be executed. (Execution phase: Refer2.1.3)

2.1.3 Execution of a Complete Instruction

A sequence of elementary operations we have discussed so far which are required to execute an instruction. They can put together to execute one instruction.

Consider the instruction,

ADD (R3), R1

Which adds the contents of the memory location pointed by R3 to register R1.The execution of this instruction requires the following actions:

1. Fetch the instruction.

2. Fetch the first operand (contents of the memory location pointed by R3)
3. Perform addition.
4. Load the result into R1.

The following figure shows the sequence of control steps to perform these operations for the single bus architecture.

Step	Action
1	PC_{out}, MAR_{in}, READ, SELECT 4, ADD, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	R_{1out}, MAR_{in}, READ
5	R_{1out}, Y_{in}, WMFC
6	MDR_{out}, SELECT Y, ADD, Z_{in}
7	Z_{out}, R_{1in}, END

Table 2.1: Control sequence for execution of the instruction Add (R3), R1

Explanation: The first three steps are common to every instruction. Instruction fetch operation is initiated by loading the contents of PC into MAR (Memory Address Register) and sending a read request to the memory. The select signal is set to 4 so that the multiplexer select the input as constant 4. This value is added to the operand at B (this is the value of PC) and the result is storing in register Z. Then this updated value is moved to PC from register Z. This is specified in control sequence step 2. In step 3, word fetched from the memory is loaded into IR.

From step 4 onwards the sequence (execution sequences) will be different for every instruction. Here, in the example instruction, contents of register R3 are transferred to MAR in step 4, and a read operation is specified. Then the contents of register R1 are transferred to register Y in step 5. When the read operation is completed, memory operand is available in MDR and the addition operation can be performed. The contents of MDR are gated to the bus, and are taken as input B of ALU circuit. Register Y is selected as the second input to the ALU by choosing select Y. The sum will be getting in register Z and will transfer to register R1 in step 7. The End signal causes a new instruction fetch cycle to begin by returning to step 1.

In the above execution of add instruction there is no need of Yin signal in step 2.But in the case of branch instruction ,the updated value of PC is needed to compute the branch target address. For this purpose only we are transferring the value of PC into register Y. This is the reason for adding Y_{in} in step 2. The fetch phase is common for all instructions.

Branch Instructions

Usually the address of the next instruction to be executed will be obtained from PC register. When branching takes place, the address is obtained by adding an offset X (which is given in the branch instruction) to the updated value of PC.

The following figure shows the control sequence that implements an unconditional branch instruction. Fetch phase is same as the above instruction. In step4, offset value is extracted from IR by the instruction decoding circuit. Updated PC value is already available in register Y. Offset X is gated onto the bus in step 4 and an addition operation is performed. The result, which is branch target address, is loaded into the PC.

Step	Action
1	PC_{out}, MAR_{in}, READ, SELECT 4, ADD, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Offset-field –of-IR_{out}, Add, Z_{in}
5	Z_{out}, PC_{in}, END

Table 2.2: Control sequence for an unconditional Branch instruction

Now a conditional branch can consider. Here we need to check the status of condition codes before loading a new value into the PC. This can be done by the control sequence

Offset-field –of-IR_{out}, Add, Z_{in}, If N=0 then End

in step 4 of above table 2.2. In this case if N=0 the processor returns to step1 immediately after step4. If N=1, step 5 is performed to load a new value into the PC, thus performing the branch operation.

2.1.4 Multiple-Bus Organization

In the case of single bus organization, only one data item can be transferred over the bus in a clock cycle. The resulting control sequence will be lengthy in such cases. To reduce the number of steps needed, most of the processors are providing multiple internal paths so that several transfers are possible during a clock cycle.

The following figure shows a three bus structure. Registers and the ALU are connected to this bus. All general purpose registers are combined into a single block called register file. The register files have 3 ports. There are two output ports and one input port. Contents of two different registers can be accessed simultaneously and have their contents placed on buses A and B by using two output ports. The third port allows the data on bus C to be loaded into a third register during the same clock cycle.

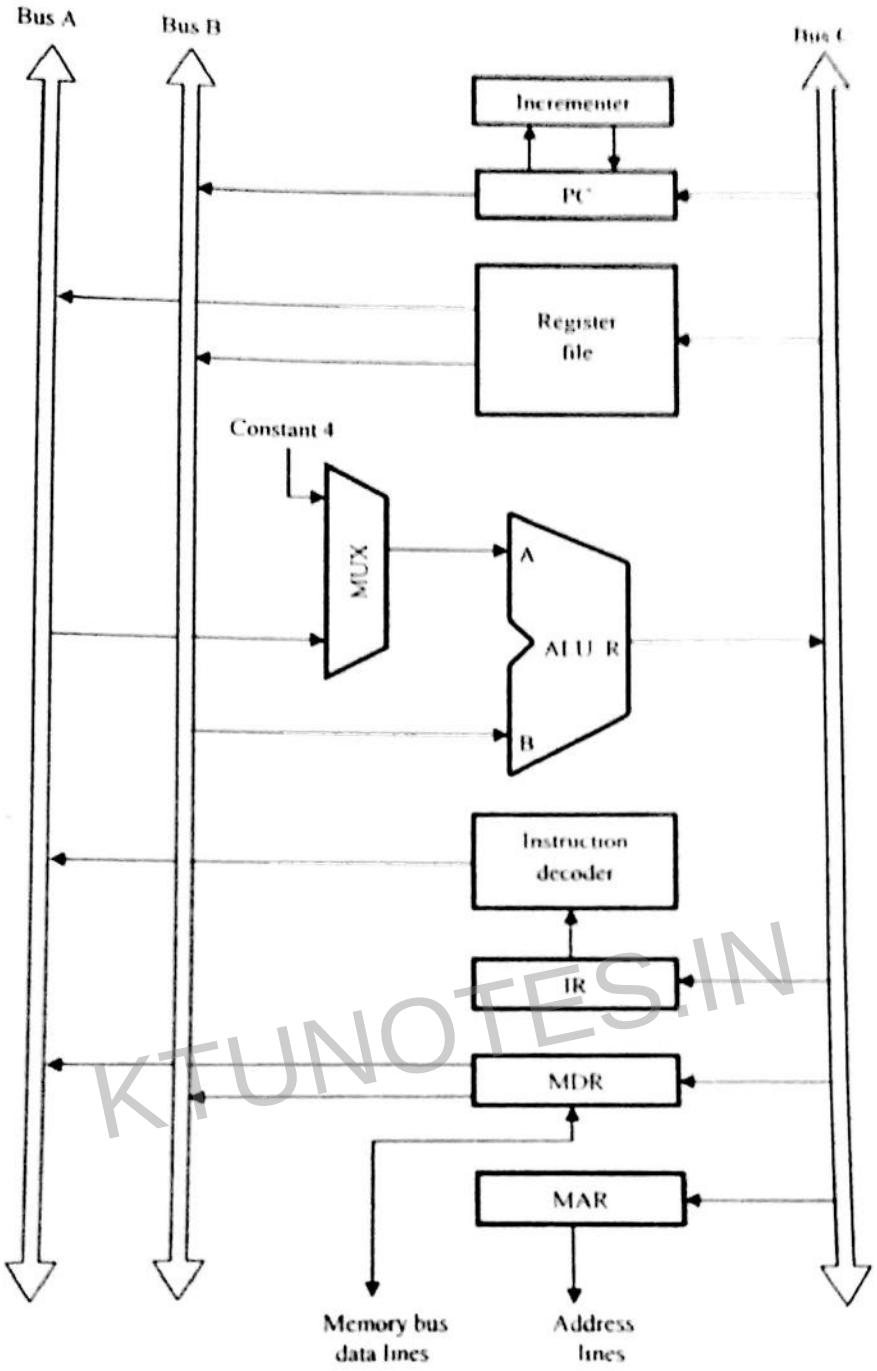


Fig 2.4: Three-bus organization of the datapath

The need of temporary registers Y and Z are not there with three bus arrangement. Buses A and B are used to transfer the source operands to the A and B inputs of the ALU.ALU performs the operation and the result is transferred to the destination over bus C.

Another feature of multiple buses is the introduction of incrementer unit. The purpose of this unit to increment the value of PC by 4. (We are assuming the word size as 4 bytes). This eliminates the need to add 4 to the PC using the main ALU. The constant4 in the ALU input multiplexer is still useful because it can be used to increment other memory addresses in LoadMultiple and StoreMultiple instructions.

Example

Write the control sequence for the instruction Add R4, R5, R6

Ans

Step	Action
1	PC_{out}, R=B, MARin, Read, IncPC
2	WMFC
3	MDR_{outB}, R=B, IR_{in}
4	$R4_{outA}$, $R5_{outB}$, Select A, Add, $R6_{in}$, END

Table 2.3: Control sequence for the instruction Add R4, R5, R6 for the three-bus organization in fig 2.4

Explanation: In step 1, the contents of the PC are passed through the ALU using R=B signal, and loaded into MAR to start a memory read operation. At the same time PC is incremented by 4. The value loaded into MAR is the original content of PC. The incremented value is loaded into the PC at the end of the clock cycle only. In step2 the processor waits for MFC signal and loads the data received into MDR. The data in MDR is transferred to IR in step3. The execution phase needs only one step as in step4 Content of R4 (one operand of add) is placed on bus A, content of R5 (one operand of add) in bus B and then addition operation is performed on the specified operands. Result is transferred to register R6.

The number of clock cycles for instruction execution is significantly reduced in multiple bus organization.

2.1.5 Sequencing of Control Signals

Binary information stored in a digital computer can be classified as either data or control information. Data is manipulated in a data path by using micro operations that are implemented with:

- Adder-Subtracters
- Shifters
- Registers
- Multiplexers
- Buses

The control unit provides signals that activate the various microoperations within the datapath. The control unit also determines the sequence in which the actions are performed.

Timing of all registers in a synchronous digital system in controlled by a master clock generator. Clock pulses are applied to all flip-flops and registers in the system, including those in the control unit. To prevent clock pulses from changing the state of all registers on every clock cycle, some registers have a load control signal that enables and disables the

loading of new data into the register. The binary variables that control the selection inputs of the components are generated by the control unit. The control unit that generates the signals for sequencing the microoperations is a sequential circuit that dictates the control signals for the system. Using status conditions and control inputs the sequential control unit determines the next state.

In a programmable system, a portion of the input to the processor consists of a sequence of instructions. Each instruction specifies the operation that the system is to perform, which operands to use, where to place the results of the operation. Instructions are stored in memory. The address of the next instruction is the PC. There is a parallel transfer from memory to the instruction register. The control unit contains decision logic to interpret the instruction. Executing an instruction means activating the necessary sequence of micro operations in the datapath required to perform the operation specified by the instruction.

In nonprogrammable systems, the control unit is not responsible for obtaining instructions from memory, nor is it responsible for sequencing execution of those instructions. There is no PC in nonprogrammable systems. Instead, the control determines the operations to be performed and the sequence of those operations, based on its inputs and status bits from the data path.

Mainly there are four sequences of control signals as we described in section 2.1.3.

(a) Fetch the instruction from memory location pointed by PC

(b) Read the operand from memory address pointed by the source register.

(c) Perform Arithmetic operations on the operands.

(d) Write back the result to the memory address pointed by destination register.

Refer the example in 2.1.3 for more details.

2.2 Arithmetic Algorithms

In this section we are describing the different methods to perform multiplication and division of binary numbers, BCD numbers and floating point numbers.

2.2.1 Multiplication Algorithms for Binary Numbers

Multiplication of two fixed point binary number in signed magnitude representation is done with paper and pencil by a process of successive shift and add operation as shown below. The process consists of looking at successive bits of multiplier in LSB first fashion. If the multiplier bit is 1, the multiplicand is copied down otherwise 0's are copied down. The numbers copied down in successive lines are shifted one position to the left from the previous number. Finally the numbers are added and their sum forms the product.

The sign of the product determined from the signs of multiplicand and multiplier. If they are same, sign of product is +ve. If the sign are not same, sign will be –ve.

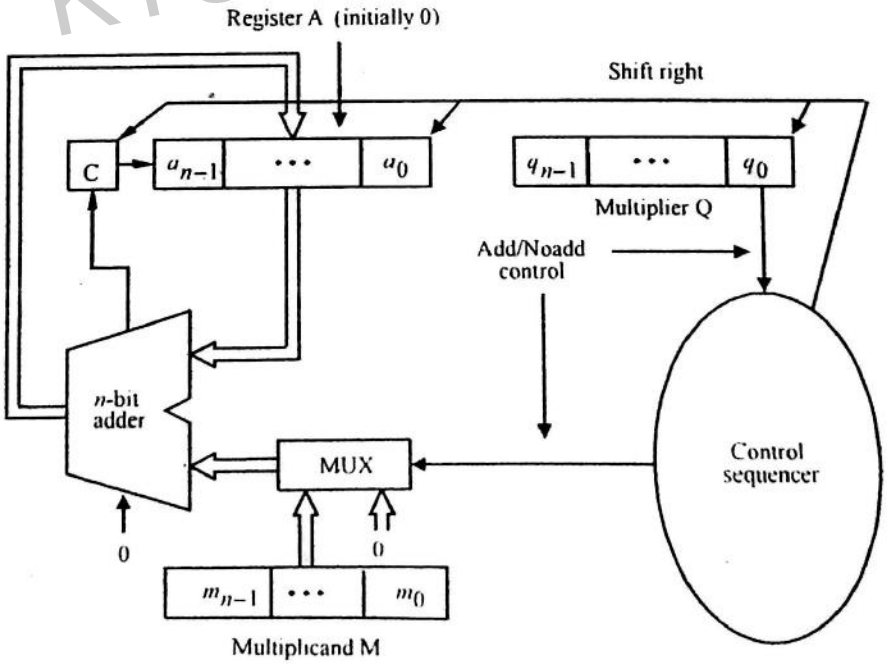
Let us consider an example, 13×6

```
   01101
   00110
  -------
   00000
  01101
 01101
00000
00000
-----------
001001110
```

$2^6+2^3+2^2+2^1=78$

- **Register Configuration**

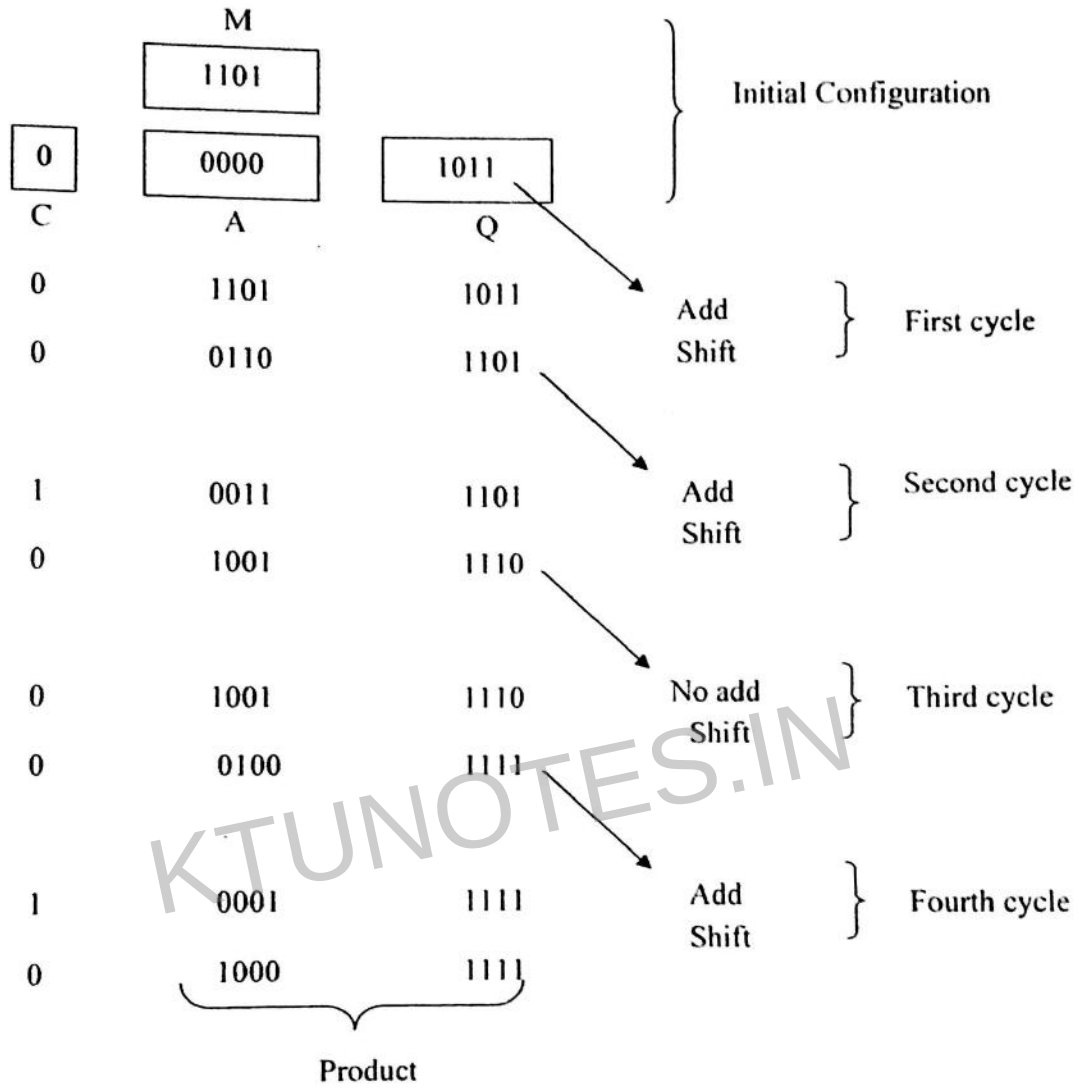
Simplest way to perform multiplication is to use an adder circuitry to ALU for no of sequential steps. The below circuit performs multiplication by using a single n-bit adder. The partial result PP_i is stored in combined A and Q register. The multiplier bit q_i generates add/no add signal. This signal controls the addition of multiplicand M to PP_i to generate PP_{i+1}. The product is computed in 'n' cycles. The carry out of the adder is stored in Flip Flop C. At the end of each cycle, C, A and Q are shifted right one bit position. Because of this shifting multiplier bit q_i appears in LSB position of Q to generate the add/no add signal at the correct time starting with q_0 during 1st cycle,q_1 during 2nd cycle and so on. After they are used, the multiplier bits are discarded by right shift operation. After end of the cycle, higher order half of the product is stored in register A and lower order in Register Q.



(a) Register configuration

Perform the multiplication of the following numbers by sequential multiplication approach.

Multiplicand = 1101 Multiplier = 1011



(b) Multiplication example

Fig 2.5: Sequential circuit binary multiplier

2.2.2 Booth's Multiplication Algorithm

By sequential network structure a multiplying instruction takes much more time to execute than an ADD instruction. Several techniques have been developed to speed up multiplication. One of them is Booth Multiplication. This is a technique which works well for both +ve and –ve multipliers. The booth algorithm generates a 2n bit product and treat both -ve and +ve 2's complement n bit operands uniformly.

In booth scheme '-1' times the shifted multiplicand is selected when moving from 0 to 1 and '+1' times shifted multiplicand is selected when moving from 1 to 0. As the multiplicand is scanned from right to left booth multiplier can be found out by following table.

Multiplier bit i, bit i-1	Version of multiplier selected by bit i
0 0	0 x M
0 1	+1 x M
1 0	-1 x M
1 1	0 xM

Table 2.4: Booth multiplier recording table

Eg: 00101111001

The booth recording of multiplier can be in the form as follows. Always assume that there is a right most zero if the right most bit is'1'.

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

The above transformation is called skipping over 1's. Here only few versions of the shifted multiplicand have to be added to generate the product. Thus speeding up the multiplication operation. In some situations, booth algorithm will work as worse (when alternate 0's and 1's), in some cases such as ordinary multiplier. (consecutive combinations of 0's and 1's and some cases larger block of 1's).Let us consider an example,

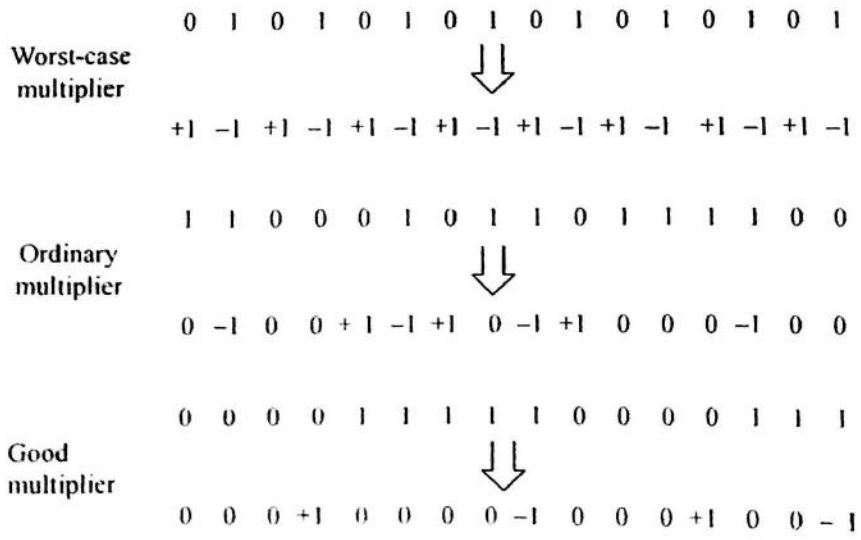


Fig 2.6: Booth recorded multiplier

Advantages of Booth Algorithm

It handles both +ve and –ve multipliers uniformly. It achieves some efficiency in the no of additions required when the multiplier has few large block of 1's. The speed gained by skipping over once depends on the data.

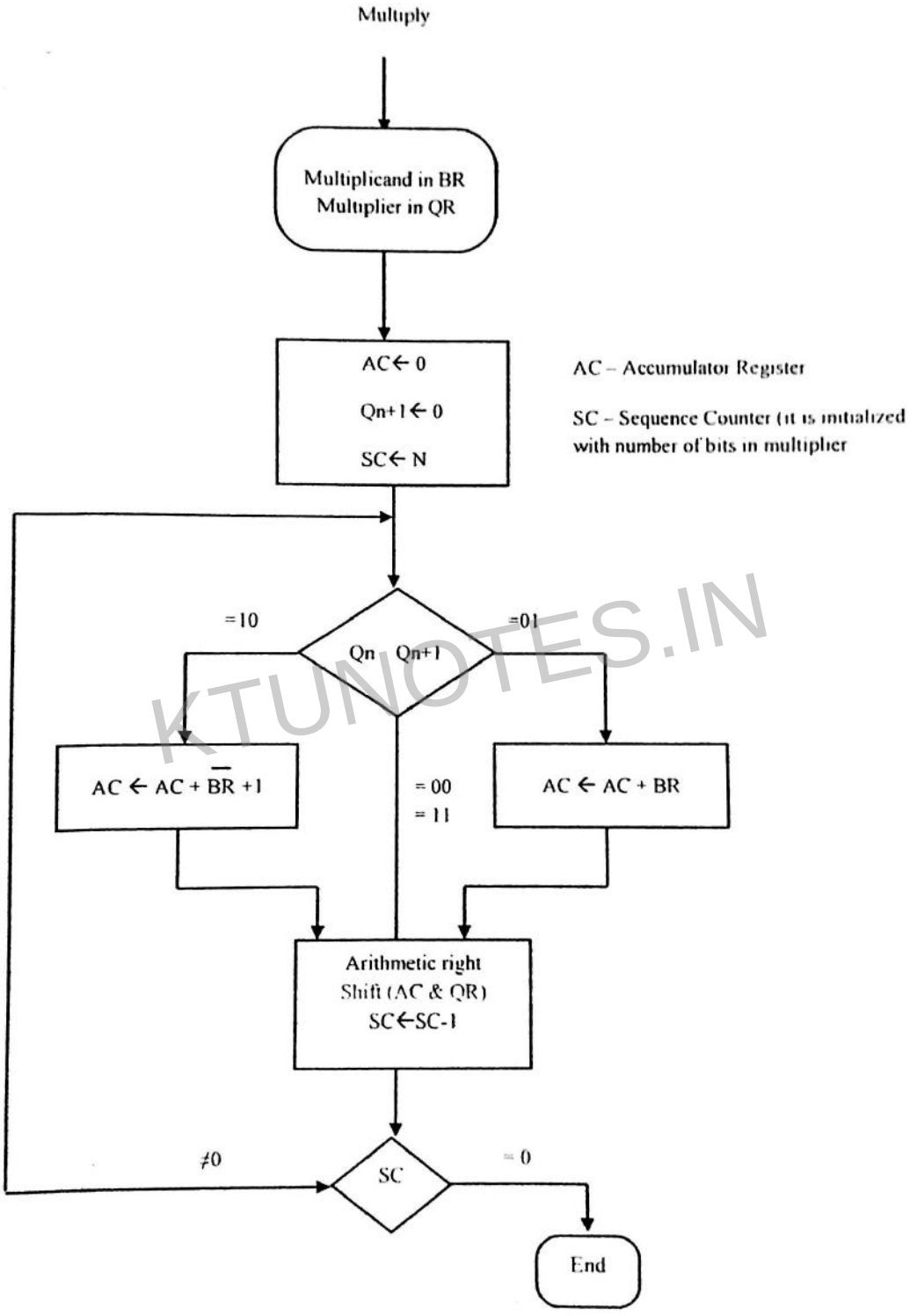


Fig 2.7: Flow chart of Booth Algorithm

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

Result is 0001110101 = 117

The above example show as the multiplication of (-9)*(13) = +117. Note that multiplier in QR is negative and multiplicand in BR is also negative. Then 10 bit product appears in AC and QR and it is positive. The final value of Q_{n+1} is the original sign bit of the multiplier and shouldn't be taken as part of the product.

Example

BR = 10111
QR = 10011

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

Result is 0001110101 = 117

The above example show as the multiplication of (-9)*(13) = +117. Note that multiplier in QR is negative and multiplicand in BR is also negative. Then 10 bit product appears in AC and QR and it is positive. The final value of Q_{n+1} is the original sign bit of the multiplier and shouldn't be taken as part of the product.

Example

BR = 00111(+7)
QR = 10101(-11)

Q_n	Q_{n+1}	$\overline{BR}+1$	AC	QR	Q_{n+1}	SC
		Initial	00000	10101	0	101
1	0	sub BR	11001			
			11001	10101	0	101
		ashr	11100	11010	1	100
0	1	add BR	00111			
			00011	11010	1	100
		ashr	00001	11101	0	011
1	0	sub BR	11001			
			11010	11101	0	011
		ashr	11101	01110	1	010
0	1	add BR	00111			
			00100	01110	1	010
		ashr	00010	00111	0	001
1	0	sub BR	11001			
			11011	00111	0	001
		ashr	11101	10011	1	000

Result is 1110110011 = -77

2.2.3 Array Multiplier

Checking the bits of the multiplier one at a time and forming the partial product is a sequential operation, that requires a sequence of add and shift micro operation. The multiplication of two binary no's can be done with one micro operation by means of a combinational circuit that forms the product bits all at once. This is the fast way of multiplying two no's since all it takes is the time for the signal to propagate trough the gates

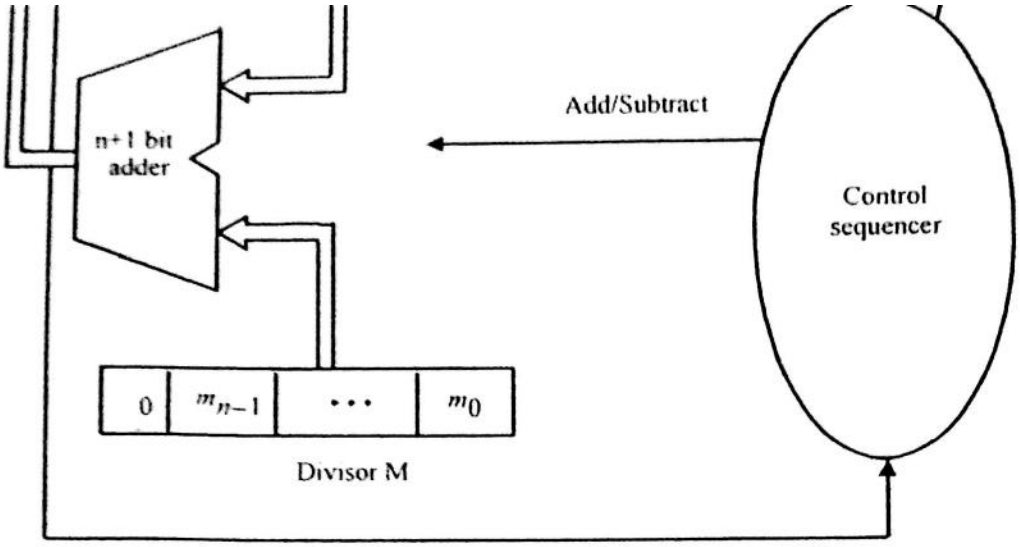


Fig 2.10: Circuit arrangement for binary division

An n-bit positive divisor is loaded in register M and n-bit positive dividend is loaded in register Q. Register A is set to 0. After the division is complete, n bit quotient is in register Q and the remainder is in register A. The required subtractions are facilitated by using 2's complement arithmetic. The extra bit position at the left end of both A and M accommodates the sign bit during subtractions.

Division Algorithms are of 2 types:-

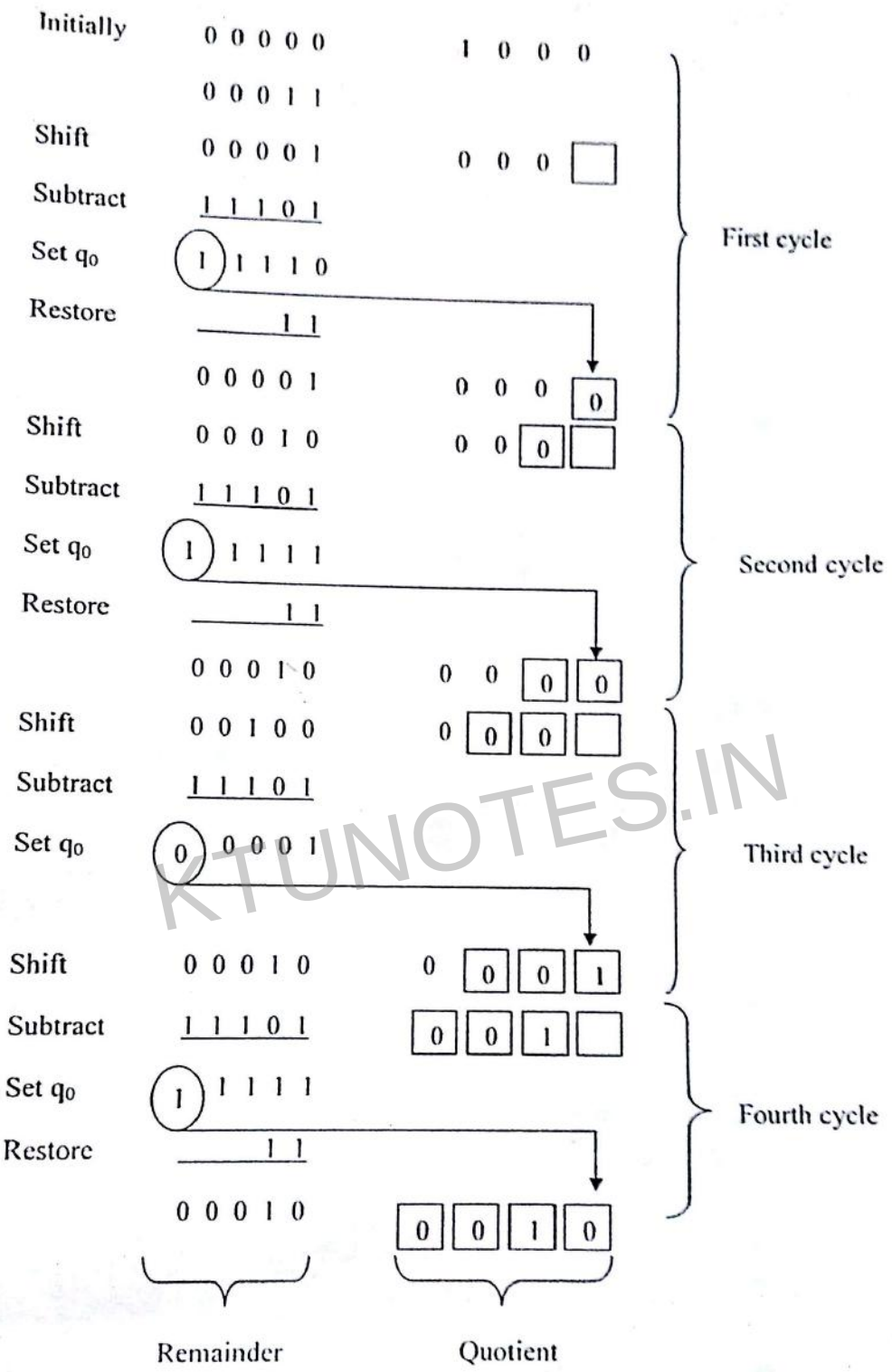
1. Restoring Division

Algorithm

Do the following n times:

1. Shift A and Q left one binary position.
2. Subtract M from A and place the answer back in A.
3. If the sign of A is 1, set q0 to 0 and add M back to A (i.e. restore A), Otherwise set q0 to 1.

The following example will perform the division on numbers 8 and 3. After the operation you can see quotient 2 is in register Q and reminder 2 will be in register A.



2. Non Restoring Division

The restoring division algorithm can be improved by avoiding the need for restoring A after an unsuccessful subtraction. Subtraction is said to be unsuccessful if the result is negative.

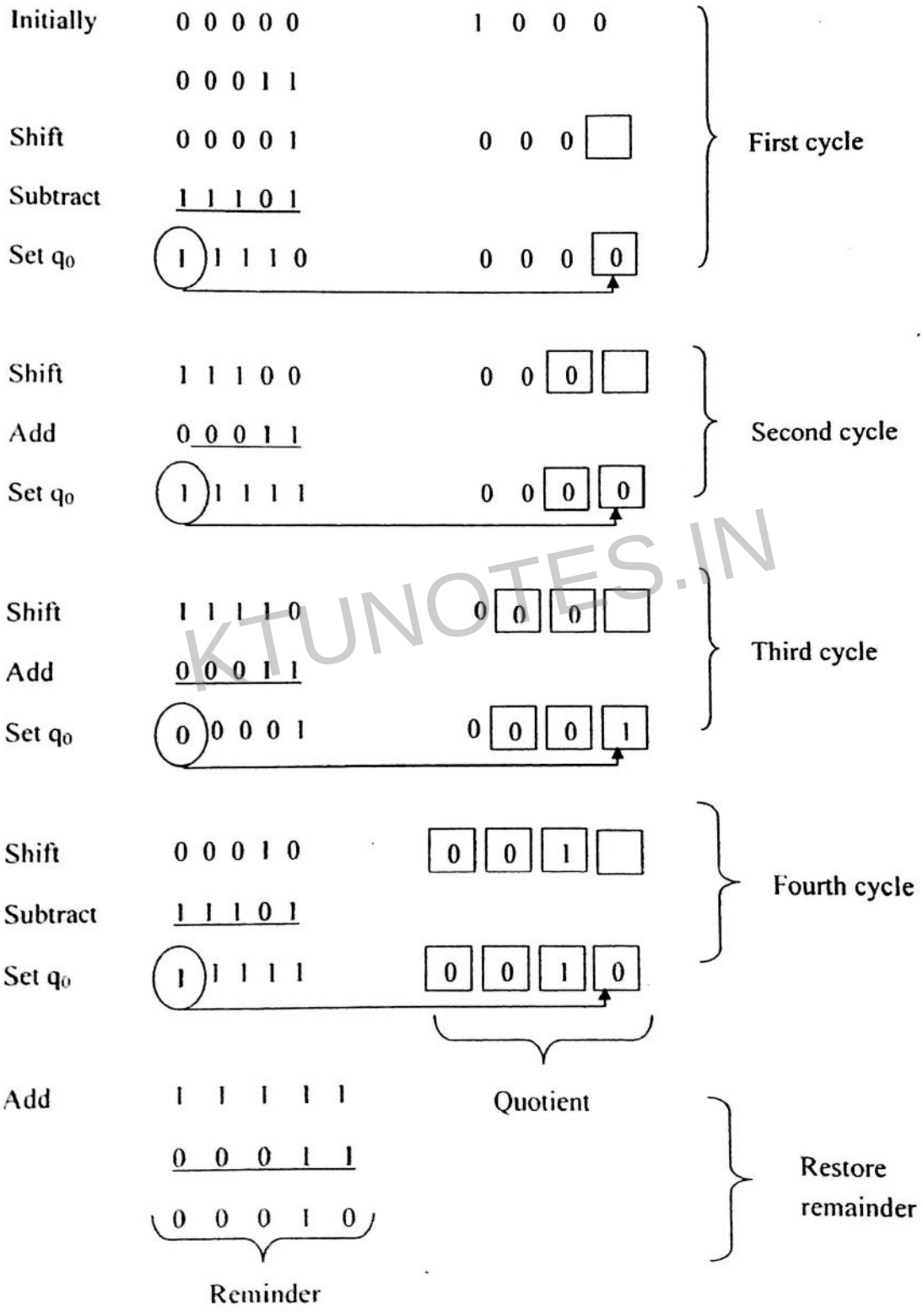
Algorithm

Step 1: Do the following n times:

1. If the sign of A is zero, shift A and Q left one bit position and subtract M from A ; otherwise Shift A and Q left and add M to A.
2. Now, if the sign of A is 0, set q0 to 1; otherwise set q_0 to 0.

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

Example: Perform division on numbers 8 and 3 using non restoring technique.



2.2.5 Representation of BCD Numbers

BCD means that Binary Coded Decimal. It represents each of the digits of an unsigned decimal as the 4-bit binary equivalents. BCD numbers can be represented in two formats. They are packed BCD numbers and unpacked BCD numbers.

- **Unpacked BCD**
  Unpacked BCD representation contains only one decimal digit per byte. The digit is stored in the least significant 4 bits; the most significant 4 bits are not relevant to the value of the represented number.
- **Packed BCD**
  Packed BCD representation packs two decimal digits into a single byte.

Following figure will give more clarification.

Decimal	Binary	BCD	
		Packed	Unpacked
0	0000 0000	0000 0000	0000 0000
1	0000 0001	0000 0001	0000 0001
2	0000 0010	0000 0010	0000 0010
3	0000 0011	0000 0011	0000 0011
4	0000 0100	0000 0100	0000 0100
5	0000 0101	0000 0101	0000 0101
6	0000 0110	0000 0110	0000 0110
7	0000 0111	0000 0111	0000 0111
8	0000 1000	0000 1000	0000 1000
9	0000 1001	0000 1001	0000 1001
10	0000 1010	0000 0001 0000 0000	0001 0000
11	0000 1011	0000 0001 0000 0001	0001 0001
12	0000 1100	0000 0001 0000 0010	0001 0010
13	0000 1101	0000 0001 0000 0011	0001 0011
14	0000 1110	0000 0001 0000 0100	0001 0100
15	0000 1111	0000 0001 0000 0101	0001 0101
16	0001 0000	0000 0001 0000 0110	0001 0110
17	0001 0001	0000 0001 0000 0111	0001 0111
18	0001 0010	0000 0001 0000 1000	0001 1000
19	0001 0011	0000 0001 0000 1001	0001 1001
20	0001 0100	0000 0010 0000 0000	0010 0000

Table 2.5: Representation of BCD numbers

- **Packing a Two-Byte BCD**

  To pack a two-byte unpacked BCD number into a single byte creating a packed BCD number, shift the upper byte left four times, then **OR** the results with the lower byte.

Example

$0000\ 0111\ 0000\ 1001_{(\text{unpacked BCD})}$	$0111\ 1001_{(\text{packed BCD})}$
$0000\ 0111 << 4$	$0111\ 0000$ (SHIFT LEFT 4)
$0111\ 0000 + 0000\ 1001$	$0111\ 1001$ (OR)

In BCD, digits from 0 to 9 only are allowed. The binary numbers 1010, 1011, 1100, 1101, 1110, 1111are not allowed in BCD system. So they are called invalid BCD numbers.

2.2.6 Addition and Subtraction of BCD Numbers

BCD Addition

Either packed or unpacked BCD numbers can be summed. BCD addition follows the same rules as binary addition. However, if the addition produces a carry and/or creates an invalid BCD number, an adjustment is required to correct the sum. The correction method is to add 6 to the sum in any digit position that has caused an error.

Example

24 + 13 = 37	15 + 9 = 24	19 + 28 = 47
0010 0100 = 24	0001 0101 = 15	0001 1001 = 19
+0001 0011 = 13	+0000 1001 = 9	-0010 1000 = 28
0011 0111 = 37	0001 1110 = 1?(invalid)	0100 0001 = 41(error)
	0001 1100 = 1?(invalid)	0100 0001 = 41(error)
	+ 0000 0110 = 6 (adjustment)	+0000 0110 = 6 (adjustment)
	0010 0100 = 24	0100 0011 = 47

BCD Subtraction

Either packed or unpacked BCD numbers can be subtracted. BCD subtraction follows the same rules as binary subtraction. However, if the subtraction causes a borrow and or creates an invalid BCD number, an adjustment is required to correct the answer. The correction method is to subtract 6 from the difference in any digit position that has caused an error.

Example

37 - 12 = 25	65 - 19 = 46	41 - 18 = 23
0011 0111 = 37	0110 0101 = 65	0100 0001 = 41
-0001 0010 = 12	-0001 1001 = 19	-0001 1000 = 18
0010 0101 = 25	0100 1100 = 4?(invalid)	0010 100 = 29(error)
	0100 1100 = 4?(invalid)	0010 100 = 29(error)
	- 0000 0110 = 6 (adjustment)	- 0000 0110 = 6 (adjustment)
	0100 0110 = 46 ✓	0010 0011 = 23 ✓

2.2.7 Multiplication of BCD Numbers

This algorithm implements BCD multiplication using a shift-add philosophy. Here we are using the notation Q_L as the least significant digit and A_e as carry digit. Finally, k is the number of digits and thus the number of iterations of the shift-add loop. Step 1 sets the sign, initializes the running total and sets the loop counter. Step 2 performs the add portion of the shift-add. Note that it loops back to itself in order to perform the multiple adds required by digits greater than one. Step 3 performs the shift, a decimal shift this time, and decrements the loop counter. Finally, step 4 loops back if not done.

Algorithm

1. As ← Bs ⊕Qs, A ← 0, A_e ← 0, SC ← k
2. IF ($Q_L \neq 0$) THEN (A_e A ← A+B, Q_L ← Q_L-1, GOTO 2)
3. dshr (A_e AQ), SC ← SC - 1
4. IF (SC ≠ 0) THEN GOTO 2

Example

B = 57 * Q = 12

STEP	A_eA	Q	SC
1	000	12	2
2	057	11	
2	114	10	
3,4	011	41	1
2	068	40	
3,4	006	84	0

 Scanned by CamScanner

Consider this example. We begin by clearing A_e and A and by setting the loop counter to 2. The first loop performs step 2 twice because the least significant digit of Q is 2. Steps 3 and 4 perform the shift and loop counter maintenance. During the second iteration, we perform step 2 only once, since the least significant digit of Q is now 1. Notice that the running product is underlined. Just as before, we shift the extra bit of the product into Q just as the least significant digit of the multiplier is processed and no longer needed.

2.2.8 Division of BCD Numbers

BCD division is an extension of binary signed-magnitude division. Again we use a decimal shift instead of a linear shift and, just as in the multiplication algorithm; more than one subtraction may be required.

Instead of performing subtraction using 2's complement, we perform it using 10's complement. The 10's complement of a number is equal to its 9's complement + 1. For example, a 3-digit number XYZ has a 10's complement of (999-XYZ) + 1.

Algorithm

Q ← A&Q div B, A ← remainder

1. IF (OVERFLOW) THEN {ERROR}
2. Qs ← As ⊕Bs, Be ← 0, SC ← k
3. dshl(AQ)
4. EA← A+B+1
5. IF (E=1) THEN (Q_L ← Q_L+1, EA← A+B+1, GOTO 5)
6. A← A+B, SC ← SC - 1
7. IF (SC≠0) THEN GOTO 3

This algorithm implements BCD division. As before, we check for overflow and exit if it present. Otherwise we proceed to step 2, where we set the sign, initialize the running quotient and set the loop counter.

Steps 3 to 7 comprise the loop. In step 3 we perform the shift and in step 4 we form A-B using 10's complement. Step 5 checks to see if the subtraction was valid. If so, it increments the quotient, performs another subtraction and loops back to itself. This step implements the multiple subtractions in this algorithm.

When we have subtracted one too many times, we proceed to step 6, where we restore the extra subtraction and decrement the loop counter. Step 7 either branches back, if we are not done, or exits the algorithm.

Example

AQ = 0769 ÷ B = 036; B+1 = 964

STEP	A	Q	SC
1,2	007	69	2
3	076	90	
4	040	90	
5	004	91	
5	968	92	
6,7	004	92	1

In this example, step 1 finds no overflow, so step 2 initializes the necessary values. The first iteration of the loop begins by shifting the value one position to the left and subtracting via 10's complement. Step 5 checks and sees that the first subtraction was valid, so it performs a second subtraction and loops back to itself. During the second iteration of step 5, we find that this subtraction is also valid, so we update Q and perform a third subtraction. The next iteration of step 5 finds that this is invalid, so step 6 restores the last subtraction. This step and step 7 finish the first iteration of the loop.

The second and final iteration performs similarly to the first iteration, except that step 5 is only executed once in this iteration. This is shown below in the continuing example above. The final result is 769/36 = 21 with a remainder of 13.

STEP	A	Q	SC
3	049	20	
4	013	20	
5	977	21	
6	013	21	0

2.2.9 Floating Point Arithmetic

A floating point number in a computer register consists of two parts:

1. Mantissa (m)
2. Exponent (e)

The two parts represents a number obtained from multiplying m times a radix r raised to the value of e, i.e., $m \times r^e$. The mantissa may be a fraction or an integer. A floating point number is normalized if the most significant digit of the mantissa is nonzero.

Register Configuration for floating point arithmetic

The same registers and adders used for the integer arithmetic can also be used in processing the mantissa part. The only difference lies in the way the exponents are

handled. There are three registers BR, AC and QR. Each register is subdivided into two parts: One for mantissa (rep by uppercase) and one for exponent (rep by lowercase).

Each floating point number has a mantissa in signed magnitude representation and a biased exponent. Thus AC has a mantissa part whose sign is in A_s, and magnitude in A. The exponent is in the part of the register and represents by lower symbol a. A_1 is the most significant digit and parallel for other registers. A parallel Adder adds the two mantissas and transfers the sum into A and carry into E. A separate parallel adder is used for exponents. The exponents are biased. (So don't have a distinct sign bit)

In this representation, the sign bit is removed from being a separate entity. The bias is a positive number that is added to each exponent as the floating point number is formed, so that internally all exponents are positive.

Consider an exponent that ranges from -50 to 49. Internally it is represented by 2 digits (without a sign) by adding it to a bias of 50.Then the exponent will become e+50, into the range of 00 to 99.

The advantage of bias exponent is that, they contain only positive numbers. It is then simpler to compare their relative magnitude without being concerned with their signs.

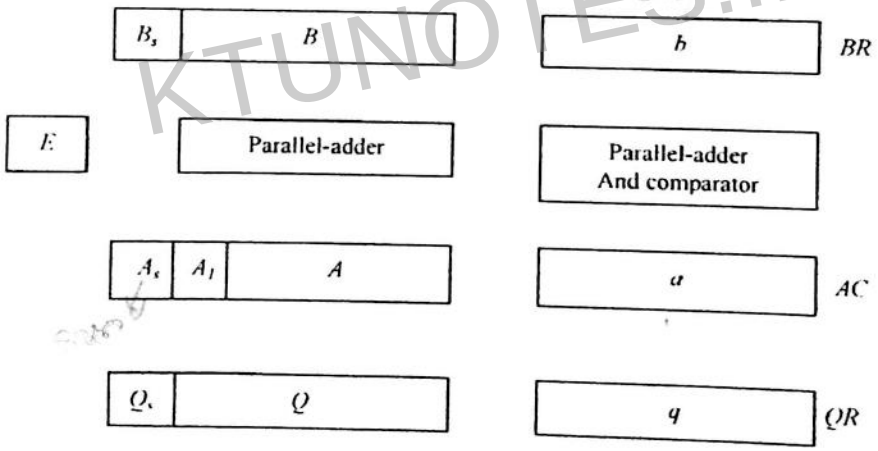


Fig 2.11: Registers for floating point arithmetic operations

Floating Point Addition and Subtraction

In addition and subtraction, operands are stored in AC and BR. The sum or difference is formed in AC. The algorithm can be divided into four parts:

1. Check for zeros
2. Align the mantissa
3. Add or subtract the mantissa
4. Normalize the result

The flow chart can be drawn as follows: