Computer Organization & Architecture EC 703 B

Introduction to Computer Organization & Architecture

Prepared by: Dr. Kaushik Roy

Basic Functional Unit

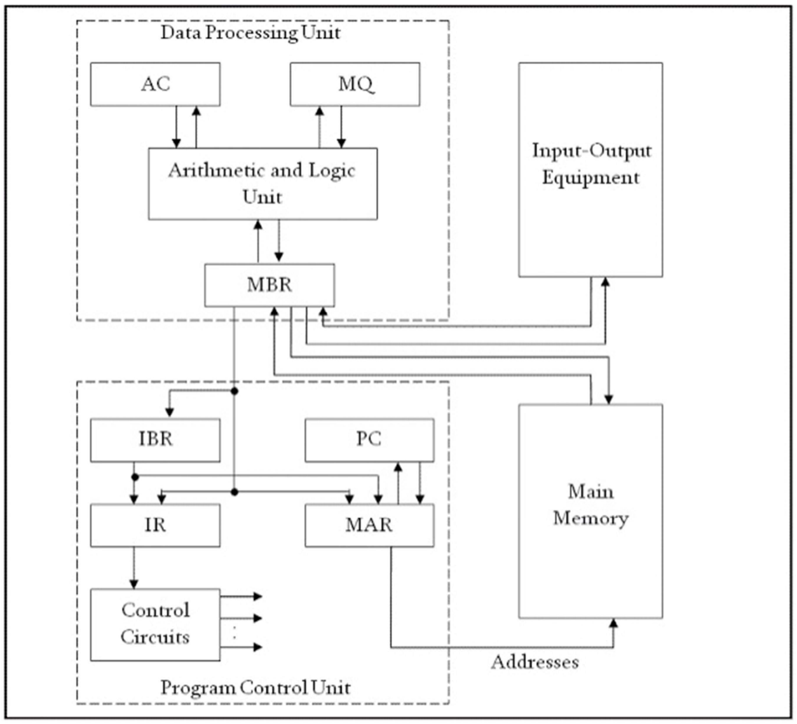


Figure 1: Expanded structure of IAS computer or Von Neumann architecture

1. MBR (Memory Buffer Register):
   MBR is a two-way register that holds the data fetched from memory and ready for the CPU to process or the data waiting to be stored in memory.

2. MAR (Memory Address Register): MAR specifies the address in memory of the word to be written from or read into the MBR.

3. IR (Instruction Register): IR contains the 8-bit op-code instruction being executed.

4. IBR (Instruction Buffer Register): IBR is employed to hold temporarily the right-hand instructions from a word in memory.

5. PC (Program Counter): PC is an counter that contains the address of the next instruction-pair to be fetched from memory to be executed.

6. AC and MQ (Accumulator and Multiplier Quotient): AC and MQ are employed to hold temporarily operands and results of ALU operations.

Harvard Architecture:

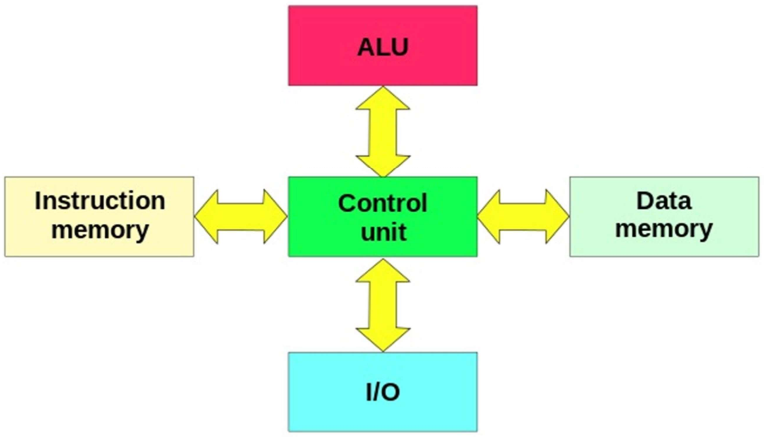


Figure 2 Harvard Architecture

The Harvard architecture is a computer architecture with physically separate storage and signal pathways for instructions and data. The term originated from the Harvard Mark I relay-based computer, which stored instructions on punched tape (24 bits wide) and data in electromechanical counters. These early machines had data storage entirely contained within the central processing unit, and provided no access to the instruction storage as data. Programs needed to be loaded by an operator; the processor could not initialize itself. Today, most processors implement such separate signal pathways for performance reasons, but actually implement a modified Harvard architecture, so they can support tasks like loading a program from disk storage as data and then executing it.

Contrast with von Neumann architectures

In a system with a pure von Neumann architecture, instructions and data are stored in the same memory, so instructions are fetched over the same data path used to fetch data. This means that a CPU cannot simultaneously read an instruction and read or write data from or to the memory. In a computer using the Harvard architecture, the CPU can both read an instruction and perform a data memory access at the same time, even without a cache. A Harvard architecture computer can thus be faster for a given circuit complexity because instruction fetches and data access do not contend for a single memory pathway.

Also, a Harvard architecture machine has distinct code and data address spaces: instruction address zero is not the same as data address zero. Instruction address zero might identify a twenty-four bit value, while data address zero might indicate an eight-bit byte that is not part of that twenty-four bit value.

Contrast with modified Harvard architecture

Computer Organization & Architecture EC 703 B

A modified Harvard architecture machine is very much like a Harvard architecture machine, but it relaxes the strict separation between instruction and data while still letting the CPU concurrently access two (or more) memory buses. The most common modification includes separate instruction and data caches backed by a common address space. While the CPU executes from cache, it acts as a pure Harvard machine. When accessing backing memory, it acts like a von Neumann machine (where code can be moved around like data, which is a powerful technique). This modification is widespread in modern processors, such as the ARM architecture, Power Architecture and x86 processors. It is sometimes loosely called a Harvard architecture, overlooking the fact that it is actually "modified".

BUS Architecture Fundamentals

What is Computer Bus: The electrically conducting path along which data is transmitted inside any digital electronic device. A Computer bus consists of a set of parallel conductors, which may be conventional wires, copper tracks on a PRINTED CIRCUIT BOARD, or microscopic aluminum trails on the surface of a silicon chip. Each wire carries just one bit, so the number of wires determines the largest data WORD the bus can transmit: a bus with eight wires can carry only 8-bit data words, and hence defines the device as an 8-bit device.

Types of Computer Bus

There are a variety of buses found inside the computer.

Data Bus: The data bus allows data to travel back and forth between the microprocessor (CPU) and memory (RAM).

Address Bus: The address bus carries information about the location of data in memory.

Control Bus: The control bus carries the control signals that make sure everything is flowing smoothly from place to place.

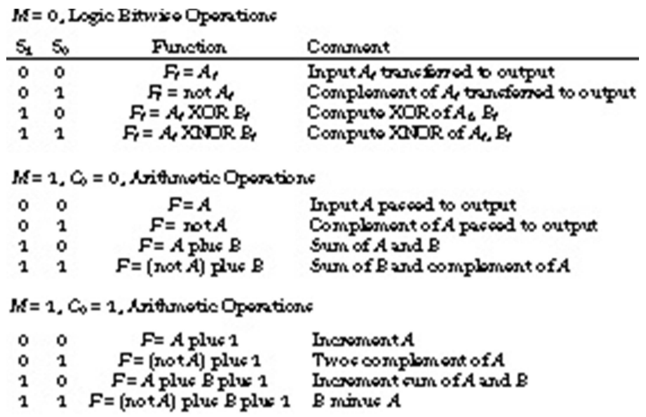
ALU Design:

An arithmetic logic unit, or ALU (sometimes pronounced "Al Loo"), is a combinational network that implements a function of its inputs based on either logic or arithmetic operations. ALUs are at the heart of all computers as well as most digital hardware systems. In this section, we learn how to design these very important digital subsystems.

A Sample ALU

An n-bit ALU typically has two input words, denoted by A = An - 1, , A0 and B = Bn - 1, , B0. The output word is denoted by F = Fn, Fn - 1, , F0, where the high-order output bit, Fn, is actually the carry-out. In addition, there is a carry-in input C0.

Besides data inputs and outputs, an ALU must have control inputs to specify the operations to be performed. One input is M, a mode selector. When M = 0, the operation is a logic function; when M = 1, an arithmetic operation is indicated. In addition, there are operation selection inputs, Si, which determine the particular logic or arithmetic function to be performed.



S_1	S_0	Function	Comment
$M = 0$, Logic Bitwise Operations			
0	0	$F_i = A_i$	Input A_i transferred to output
0	1	$F_i =$ not A_i	Complement of A_i transferred to output
1	0	$F_i = A_i$ XOR B_i	Compute XOR of A_i, B_i
1	1	$F_i = A_i$ XNOR B_i	Compute XNOR of A_i, B_i
$M = 1$, $C_0 = 0$, Arithmetic Operations			
0	0	$F = A$	Input A passed to output
0	1	$F =$ not A	Complement of A passed to output
1	0	$F = A$ plus B	Sum of A and B
1	1	$F =$ (not A) plus B	Sum of B and complement of A
$M = 1$, $C_0 = 1$, Arithmetic Operations			
0	0	$F = A$ plus 1	Increment A
0	1	$F =$ (not A) plus 1	Twos complement of A
1	0	$F = A$ plus B plus 1	Increment sum of A and B
1	1	$F =$ (not A) plus B plus 1	B minus A

Figure 3: Simple ALU functions

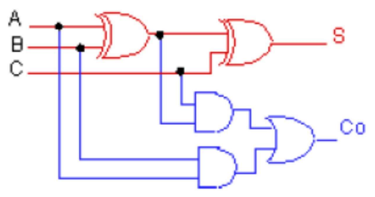
To make the discussion more concrete, Figure 3 contains the specification of a simple ALU bit slice, that is, the behavior of a single bit of the ALU. The list of operations is partitioned into three sections: logic operations, arithmetic operations where the carry-in is 0, and arithmetic operations where the carry-in is 1. Some of the operations do not appear to be useful, such as the sum of B and the ones complement of A. However, if we set carry-in to 1, we obtain a very useful operation indeed: B minus A (B plus the 2's complement of A).

Implementation of an ALU: ALUs are relatively simple to implement: design a 1-bit slice and cascade as many of these as you need to build a multi-bit structure. Of course, the limiting performance factor will be the propagation of carries among the ALU stages.

Using the specification of Figure 3, a single bit slice has six inputs, A_i, B_i, C_i, M, S_1, and S_0, and two outputs, F_i and C_{i+1}.

In ALU design, different arithmetic and logic operations are included. Among arithmetic operations, addition, subtraction, multiplication, division, exponentiation etc. are included. Among logical operations, AND, OR, NOR, NAND, XOR, XNOR, Left/ Right Shift operations, Rotate operation etc. are included.

Addition: Boolean Equations: □ = □ □ □ □ □ , □ = □□ + (□ □ □)□



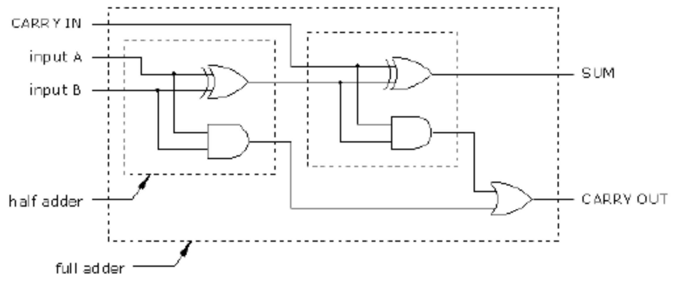


Figure 4 Circuit of Full Adder

Here, □ = □ and □ = □ .

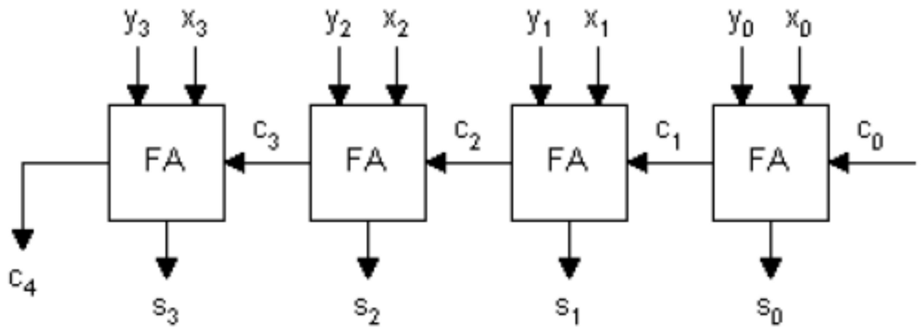


Figure 5 4 bit Parallel Adder

In Figure 5, 4 bit parallel adder using the adder shown in Figure 4 is shown.

Algorithm: (VHDL code) library IEEE;

```
use IEEE.STD_LOGIC_1164.ALL;

entity adder4bit is generic(N : integer := 4);

port(a,b : in std_logic_vector(N-1 downto 0);

r : out std_logic_vector(N downto 0)); end

adder4bit; architecture Behavioral of adder4bit

is begin process(a,b) variable cin:

std_logic_vector(N downto 0); begin

cin(0):='0'; for i in 0 to N-1 loop

r(i)<= a(i) xor b(i) xor cin(i); cin(i+1):= (a(i) and b(i))

or ((a(i) xor b(i)) and cin(i)); end loop; r(N)<= cin(N);

end process; end Behavioral;
```

Subtraction: Boolean Equations: □ = □ □ □ □ □□□ , □□□ = □□ + (□ □ □)□□□

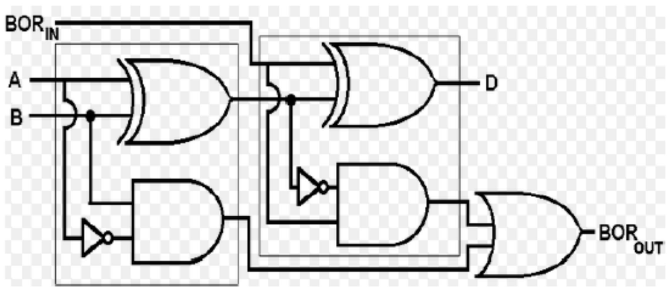


Figure 6 Full Subtractor

Here, □□□ = □□□ and □□□ = □□□ .

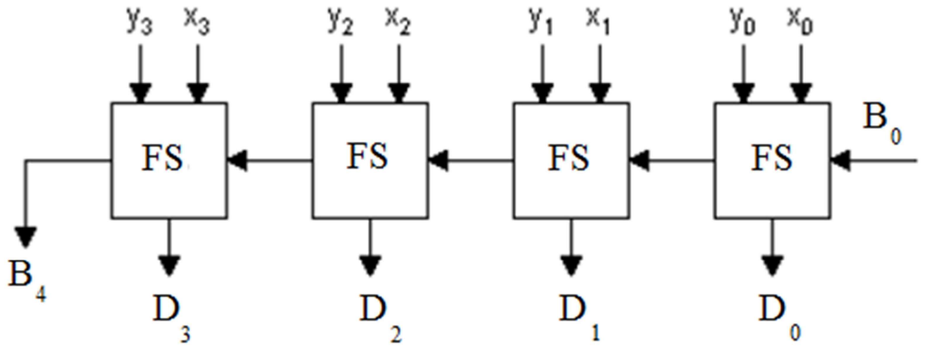


Figure 7 4 bit Parallel Subtractor

In Figure 7, 4 bit parallel subtractor using the subtractor shown in Figure 6 is shown.

Algorithm: General algorithm (VHDL code) library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

entity subtractor4bit is generic(N : integer := 4); port(a,b : in

std_logic_vector(N-1 downto 0); r : out

std_logic_vector(N downto 0)); end subtractor4bit; architecture

Behavioral of subtractor4bit is begin process(a,b) variable bin:

std_logic_vector(N downto 0); begin bin(0):='0'; for i in 0 to N-1

loop r(i)<= a(i) xor b(i) xor bin(i); bin(i+1):= ((not a(i)) and b(i))

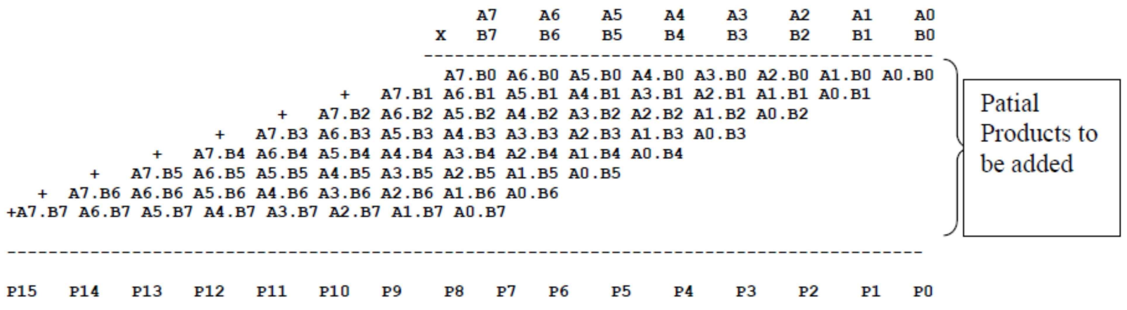
or ((not (a(i) xor b(i))) and bin(i)); end loop; r(N)<= bin(N); end

process; end Behavioral;

The output of the algorithm may be positive or negative. In binary number system, negative number can be represented by 2’s complement. To convert it to positive number the output should be converted again by 2’s complement method. Algorithm: 2’s Complement Subtraction (VHDL code)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity subtractor4bit_signadjust is generic(N : integer := 4);

port(a,b : in std_logic_vector(N-1 downto 0);     r : out

std_logic_vector(N downto 0)); end subtractor4bit_signadjust;

architecture Behavioral of subtractor4bit_signadjust is begin

process(a,b) variable bin,res,bin1: std_logic_vector(N downto 0);

variable rs: std_logic_vector(N-1 downto 0); begin bin(0):='0';

for i in 0 to N-1 loop res(i):= a(i) xor b(i) xor bin(i); bin(i+1):=

((not a(i)) and b(i)) or ((not (a(i) xor b(i))) and bin(i)); end loop;

res(N):= bin(N); for i in 0 to N-1 loop rs(i):= res(N) xor res(i);

end loop; bin1(0):=res(N); for i in 0 to N-1 loop

r(i)<= rs(i) xor bin1(i);

bin1(i+1):= rs(i) and bin1(i);

end loop; r(N)<= res(N); end

process; end Behavioral;
```

Multiplier: Mathematical Expression: □ = ∑ ∑ □ □ 2 where □ and □ are bits of the operands □ = ∑ □ 2 and □ = ∑ □ 2 .



```
                                          A7    A6    A5    A4    A3    A2    A1    A0
                                      X   B7    B6    B5    B4    B3    B2    B1    B0
                                     ------------------------------------------------
                                        A7.B0 A6.B0 A5.B0 A4.B0 A3.B0 A2.B0 A1.B0 A0.B0
                                  +  A7.B1 A6.B1 A5.B1 A4.B1 A3.B1 A2.B1 A1.B1 A0.B1
                              +  A7.B2 A6.B2 A5.B2 A4.B2 A3.B2 A2.B2 A1.B2 A0.B2
                          +  A7.B3 A6.B3 A5.B3 A4.B3 A3.B3 A2.B3 A1.B3 A0.B3
                      +  A7.B4 A6.B4 A5.B4 A4.B4 A3.B4 A2.B4 A1.B4 A0.B4
                  +  A7.B5 A6.B5 A5.B5 A4.B5 A3.B5 A2.B5 A1.B5 A0.B5
               +  A7.B6 A6.B6 A5.B6 A4.B6 A3.B6 A2.B6 A1.B6 A0.B6
+A7.B7 A6.B7 A5.B7 A4.B7 A3.B7 A2.B7 A1.B7 A0.B7
-------------------------------------------------------------------------------------
P15  P14  P13  P12  P11  P10  P9   P8  P7  P6   P5   P4   P3   P2   P1  P0
```

Figure 8 Structure for simple array multiplier

Algorithm: Array Multiplier (VHDL code) library

```
IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity arraymultiplier4bit is generic(N : integer :=
4); port(a,b : in std_logic_vector(N-1 downto 0);
  p : out std_logic_vector(2*N-1 downto 0)); end
arraymultiplier4bit; architecture Behavioral of
arraymultiplier4bit is begin
process(a,b) variable res: std_logic_vector(N-1 downto 0);
variable pp: std_logic_vector(N-1 downto 0); variable rs:
std_logic_vector(N-1 downto 0); variable cin:
std_logic_vector(N downto 0); begin for i in 0 to N-1 loop
res(i):= '0'; end loop; for i in 0 to N-1 loop for j in 0 to N-1
loop pp(j):= a(j) and b(i); end loop; cin(0):= '0'; for j in 0 to
N-1 loop rs(j):= res(j) xor pp(j) xor cin(j); cin(j+1):= (res(j)
and pp(j)) or ((res(j) xor pp(j)) and cin(j)); end loop; p(i)<=
rs(0); res(N-2 downto 0):= rs(N-1 downto 1); res(N-1):=
cin(N); end loop; p(2*N-1 downto N)<= res(N-1 downto 0);
```

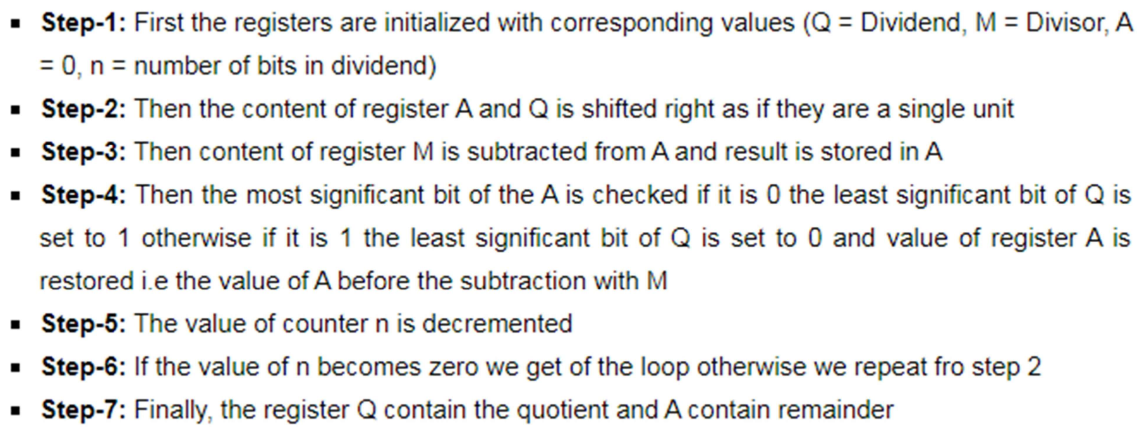
```
end process; end

Behavioral;
```

Algorithm: Booth’s Multiplier (radix 2) (VHDL code)

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;

entity boothmultiplier4bit is generic(N : integer :=

4); port(a,b : in std_logic_vector(N-1 downto 0);

  p : out std_logic_vector(2*N-1 downto 0)); end

boothmultiplier4bit; architecture Behavioral of

boothmultiplier4bit is begin process(a,b) variable

res: std_logic_vector(N-1 downto 0); variable pp:

std_logic_vector(N-1 downto 0); variable rs:

std_logic_vector(N-1 downto 0); variable cin:

std_logic_vector(N downto 0); begin for i in 0 to N-

1 loop res(i):= '0'; end loop; for i in 0 to N-1 loop

if(b(i)='1') then

for j in 0 to N-1 loop pp(j):= a(j); end loop; else for j in 0 to

N-1 loop pp(j):= '0'; end loop; end if; cin(0):= '0'; for j in 0

to N-1 loop rs(j):= res(j) xor pp(j) xor cin(j); cin(j+1):=

(res(j) and pp(j)) or ((res(j) xor pp(j)) and cin(j)); end loop;

p(i)<= rs(0); res(N-2 downto 0):= rs(N-1 downto 1); res(N-

1):= cin(N); end loop; p(2*N-1 downto N)<= res(N-1

downto 0); end process; end Behavioral;
```

Divider:

Algorithm: Restoring Division:



- **Step-1:** First the registers are initialized with corresponding values (Q = Dividend, M = Divisor, A = 0, n = number of bits in dividend)
- **Step-2:** Then the content of register A and Q is shifted right as if they are a single unit
- **Step-3:** Then content of register M is subtracted from A and result is stored in A
- **Step-4:** Then the most significant bit of the A is checked if it is 0 the least significant bit of Q is set to 1 otherwise if it is 1 the least significant bit of Q is set to 0 and value of register A is restored i.e the value of A before the subtraction with M
- **Step-5:** The value of counter n is decremented
- **Step-6:** If the value of n becomes zero we get of the loop otherwise we repeat fro step 2
- **Step-7:** Finally, the register Q contain the quotient and A contain remainder

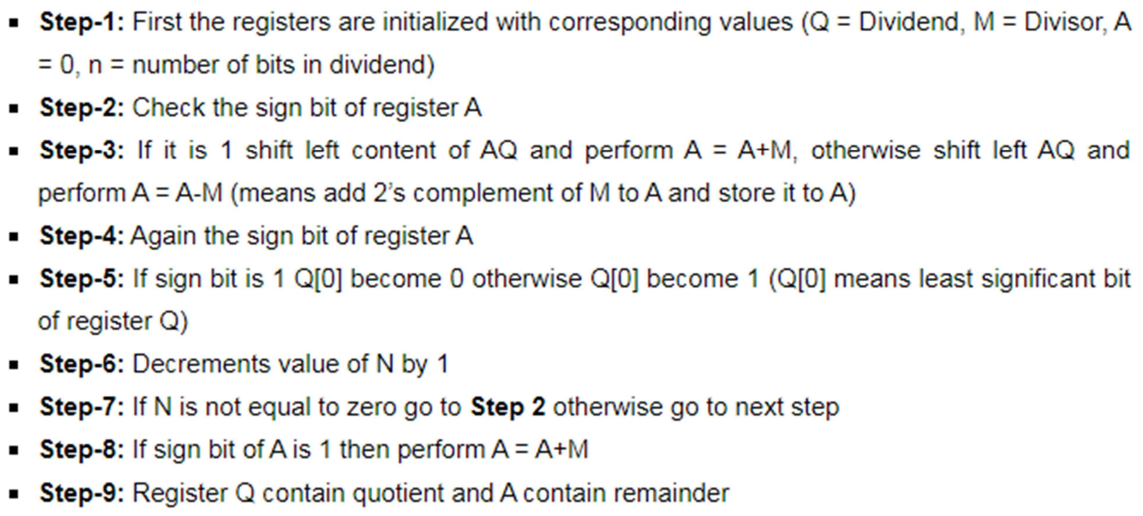
Figure 9 Restoring Algorithm

Restoring Division (VHDL code): library

```
IEEE;

use IEEE.STD_LOGIC_1164.ALL;

entity restoring_division is generic(N : integer :=

4); port(a,b : in std_logic_vector(N-1 downto 0);

  q : out std_logic_vector(N-1 downto 0);           r

: out std_logic_vector(N downto 0)); end

restoring_division;


architecture Behavioral of restoring_division is begin

process(a,b) variable pp,div,twos,rs,res,re:

std_logic_vector(N downto 0); variable bin,bin1:

std_logic_vector(N+1 downto 0); variable dv,d:

std_logic_vector(N-1 downto 0);
```

```
begin for i in 0 to N-1 loop div(i):=
not b(i); end loop; div(N):= '1'; dv(N-
1 downto 0):= a(N-1 downto 0);
bin(0):= '1'; for i in 0 to N loop twos(i):= div(i) xor bin(i);
bin(i+1):= div(i) and bin(i); end loop; for i in 0 to N loop pp(i):=
'0'; end loop; for i in 0 to N-1 loop rs(N downto 0):= pp(N-1
downto 0) & dv(N-1); d(N-1 downto 1):= dv(N-2 downto 0);
bin1(0):= '0'; for j in 0 to N loop res(j):= rs(j) xor twos(j) xor
bin1(j); bin1(j+1):= (rs(j) and twos(j)) or ((rs(j) xor twos(j)) and
bin1(j)); end loop; if(res(N)='0') then
pp(N downto 0):= res(N downto 0);
else pp(N downto 0):= rs(N downto
0);
end if; d(0):= not res(N); dv(N-1
downto 0):= d(N-1 downto 0); end
loop; r(N downto 0)<= pp(N downto
0); q(N-1 downto 0)<= dv(N-1 downto
0); end process; end Behavioral;
```

Algorithm: Non-Restoring Division:



- **Step-1:** First the registers are initialized with corresponding values (Q = Dividend, M = Divisor, A = 0, n = number of bits in dividend)
- **Step-2:** Check the sign bit of register A
- **Step-3:** If it is 1 shift left content of AQ and perform A = A+M, otherwise shift left AQ and perform A = A-M (means add 2's complement of M to A and store it to A)
- **Step-4:** Again the sign bit of register A
- **Step-5:** If sign bit is 1 Q[0] become 0 otherwise Q[0] become 1 (Q[0] means least significant bit of register Q)
- **Step-6:** Decrements value of N by 1
- **Step-7:** If N is not equal to zero go to **Step 2** otherwise go to next step
- **Step-8:** If sign bit of A is 1 then perform A = A+M
- **Step-9:** Register Q contain quotient and A contain remainder

Figure 9 Non-Restoring Algorithm

Non-Restoring Division (VHDL code):

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity nonrestoring_division is generic(N : integer := 4); port(a,b

: in std_logic_vector(N-1 downto 0);      q : out

std_logic_vector(N-1 downto 0);           r : out

std_logic_vector(N downto 0)); end nonrestoring_division;

architecture Behavioral of nonrestoring_division is begin

process(a,b) variable pp,div,twos,rs,res,re,rsd:

std_logic_vector(N downto 0); variable bin,bin1,bin2:

std_logic_vector(N+1 downto 0); variable dv,d:

std_logic_vector(N-1 downto 0); begin for i in 0 to N-1 loop

div(i):= not b(i); end loop; div(N):= '1'; re(N-1 downto 0):= b(N-

1 downto 0); re(N):= '0'; dv(N-1 downto 0):= a(N-1 downto 0);
```

```
bin(0):= '1'; for i in
0 to N loop
twos(i):= div(i)
xor bin(i);
bin(i+1):= div(i)
and bin(i); end
loop; for i in 0 to
N loop pp(i):= '0';
end loop; for i in 0
to N-1 loop rs(N
downto 0):= pp(N-
1 downto 0) &
dv(N-1); d(N-1
downto 1):= dv(N-
2 downto 0);
bin1(0):= '0'; for j
in 0 to N loop
res(j):= rs(j) xor
twos(j) xor
bin1(j);
bin1(j+1):= (rs(j)
and twos(j)) or
((rs(j) xor twos(j))
and bin1(j)); end

loop; for j in 0 to

N loop rsd(j):=

res(N) and re(j);

end loop;

bin2(0):= '0'; for j

in 0 to N loop

pp(j):= res(j) xor

rsd(j) xor bin2(j);

bin2(j+1):= (res(j)

and rsd(j)) or

((res(j) xor rsd(j))

and bin2(j)); end

loop; d(0):= not

res(N);

dv(N-1 downto 0):= d(N-1 downto 0);

end loop; r(N downto 0)<= pp(N

downto 0); q(N-1 downto 0)<= dv(N-1

downto 0); end process; end

Behavioral;
```

IEEE-754 format for floating point numbers:

A floating point signed number A can be represented as, □ = □ 2 + ∑ □ 2 , where N is the number of precision. IEEE standardized the representation of floating point numbers as 32 bit for single precision and 64 bit for double precision.

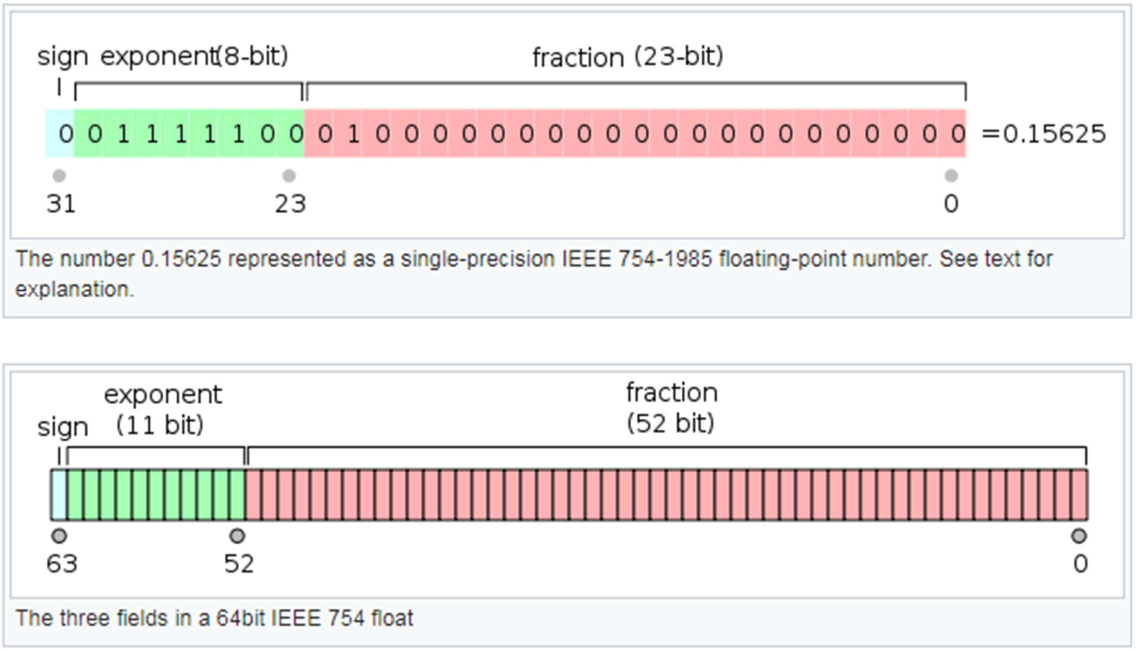

sign exponent(8-bit)
fraction (23-bit)
0 0 1 1 1 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
=0.15625
31
23
0
The number 0.15625 represented as a single-precision IEEE 754-1985 floating-point number. See text for explanation.
sign
exponent (11 bit)
fraction (52 bit)
63
52
0
The three fields in a 64bit IEEE 754 float

Figure 10 IEEE-754 format for single precision and double precision floating point numbers

As per IEEE format, for single precision, the MSB is the sign bit, next 8 bits are the exponent and the last 23 bits are mantissa or fractional bits. For double precision, the MSB is the sign bit, next 11 bits are the exponent and the last 52 bits are mantissa or fractional bits. If A is a floating point number, then $\square = (-1) \times 1.\ \square \times 2^{()}$, i=31th bit value, m=mantissa and e=exponent for single precision representation. For double precision, $\square = (-1) \times 1.\ \square \times 2^{()}$, i=63th bit value.
Algorithm: Binary to IEEE-754 single precision format for 33 bit number (one sign bit) (VHDL code):

```
library IEEE; use

IEEE.STD_LOGIC_1164.ALL;

entity ieee_floating_point_format is port(a : in

std_logic_vector(32 downto 0);             ifnr : out

std_logic_vector(31 downto 0)); end

ieee_floating_point_format; architecture Behavioral of

ieee_floating_point_format is begin process(a) variable

inta,binexp: integer; begin ifnr(31)<= a(32); for i in 0 to
```

```
31 loop if(a(i)='1') then inta:= i; end if; end loop;

binexp:= inta + 127; for i in 0 to 7 loop if((binexp mod

2)=0) then ifnr(i+23)<= '0'; else

ifnr(i+23)<= '1'; end if; binexp:=

binexp / 2; end loop; ifnr(22 downto

0)<= a(31 downto 9); end process;

end Behavioral;
```

Truncation Technique:

Consider, a floating point number, □ = ∑ □ 2 . If the number is truncated up to n bits then it can be approximated as, □̅ = ∑ □ 2 . Then the truncation error can be represented as, □ = □ − □̅ = ∑ □ 2 − ∑ □ 2 = ∑ □ 2 < ∑ 2 = 2 ( )(1 + 2 + 2 + ⋯ ∞) ≅ 2 which implies that if the number of precision (n) of the floating point number is increased then the truncation error can be reduced.

Now, let us consider a floating point number is of the form □ = ∑ □ 2 + 0 × 2 ( ) + ∑ 2 . It can be approximated as □̅ = ∑ □ 2 + 2 ( ). Since, □̅ > □, then here the truncation error is □ = □̅ − □ = ∑ □ 2 + 2 ( ) − ∑ □ 2 − 0 × 2 ( ) − ∑ 2 = 2 ( ) − ∑ 2 = 2 ( ) − 2 ( )(1 + 2 + 2 + □) □ 2 ( ) − 2 ( )=0 which implies that the truncation error for this kind of approximation is almost 100% accurate if the number of precision (n) is increased. As for example, let □ = 0.10110010101111111 ….. It can be approximated as □̅ = 0.1011001011. Therefore the finding of truncation should be performed using large precision to eliminate error. Algorithm: (VHDL code) library IEEE; use IEEE.STD_LOGIC_1164.ALL;

```
entity truncation is generic(N : integer := 32); port(a : in

std_logic_vector(63 downto 0);          approx : out

std_logic_vector(N-1 downto 0)); end truncation;

architecture Behavioral of truncation is begin
```

```
process(a) variable appa: std_logic_vector(N-1 downto

0); variable cnt: integer; begin appa(31 downto 1):=

a(63 downto N+1); for i in 0 to N-1 loop if(a(i)='1')

then cnt := cnt + 1; end if; end loop; if(cnt>((N*3)/4))

then appa(0) := '1'; else appa(0) := a(N); end if;

approx(31 downto 0)<= appa(31 downto 0); cnt:= 0;

end process; end Behavioral;
```

Instruction Set:

An Instruction is a command given to the computer to perform a specified operation on given data. The instruction set of a microprocessor is the collection of the instructions that the microprocessor is designed to execute. The instructions described here are of Intel 8085. These instructions are of Intel Corporation. The instructions can be classified into the following groups:

1. Data Transfer Group
2. Arithmetic Group
3. Logical Group
4. Branch Control Group
5. I/O and Machine Control Group

Data Transfer Group:

This group includes the function of data transfer and loading from one register or input port to other registers. In normal 8085 programming, MOV, MVI, LXI, LDA, STA etc. instructions are called data transfer instructions.

Arithmetic Group:

This group includes the arithmetic operations. In 8085 programming, ADD, SUB, INR, DCR, DAD etc. are called arithmetic instructions. Here Accumulator and Flag registers have important contributions.

Logical Group:

This group includes the logical operations. In 8085 programming, ANA, XRA, ORA, CMP, RAL, RAR, RLC, RRC etc. are called logical instructions. Here also Accumulator and Flag registers have important contributions.

Branch Control Group:

This group includes the instructions like conditional and unconditional jump, subroutine call, return, restart etc. In 8085 programming, JMP, JC, JZ, CALL, CZ, RST etc. are the branch instructions.

I/O and Machine Control Group:

This group includes the input/ output port declaration, stack, machine control etc. IN, OUT, PUSH, POP, HLT etc. are these instructions.

Instruction Format:

An instruction format defines the layout of the bits of an instruction, in terms of its constituents. An instruction format must include an opcode and, implicitly or explicitly, zero or more operands.

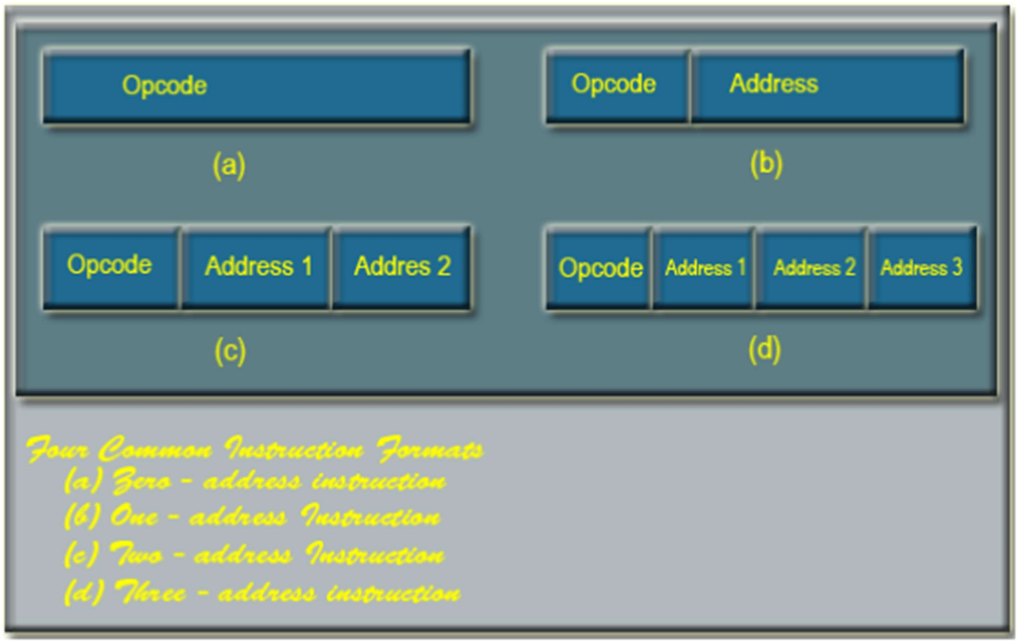

Opcode
(a)
Opcode
Address
(b)
Opcode
Address 1
Addres 2
(c)
Opcode
Address 1
Address 2
Address 3
(d)
Four Common Instruction Formats
(a) Zero - address instruction
(b) One - address Instruction
(c) Two - address Instruction
(d) Three - address instruction

Figure 11 Instruction Format

Figure 11 shows different instruction formats. The first format (Figure 11(a)) consists of zero address instruction. The second format (Figure 11(b)) consists of single address instruction. Similarly Figure 11(c) and (d) show the formats for two and three address instructions respectively. Opcode is the encoded form of the instruction to be executed by the microprocessor or microcontroller.

In CISC (Complex Instruction Set Computer) architecture, few extra instructions are required for a fixed program but in RISC (Reduced Instruction Set Computer) architecture, the same program can be executed with less number of instructions. Let us take an example with the following C program.

if(a>=b) { a=a-b; } else { a=b-a;

}

In CISC processor like 8085, this program can be written as follows.

```
MVI B, a;  // Load a

MVI C, b;  //Load b

MOV A, B;  //Copy a to Accumulator

SUB C;  //Subtract b from a;

JC Loop;  //Jump to Loop line if carry flag is set

CMA;  //Complement A if carry flag is reset

ADI #01h;  //Add 1 with the accumulator content

Loop: OUT 80h;  //Output through port address 80h
```

Therefore 8 instructions are required to complete the program in CISC processor. In RISC processor like ARM, this program can be written as follows.

```
MOV R1, a;

MOV R2, b;

CMP R1, R2;

SUBGE R1, R1, R2;

SUBLT R1, R2, R1;

OUT Port address, R1;
```

Thus only 6 instructions are required to complete the program in RISC processor.