Department of Computer Science and Engineering

RGUKT – SRIKAKULAM

COMPUTER ORGANIZATION AND ARCHITECTURE
LECTURE NOTES

Prepared by
Mr. Dileep Kumar Koda *M. Tech., (Ph. D)*
Asst. Professor
Computer Science & Engineering

Rajiv Gandhi University of Knowledge Technologies

Srikakulam

Catering to the Educational Needs of Gifted Rural Youth of Andhra Pradesh

(Established by the Govt. of Andhra Pradesh and recognized as per Section 2(f) of UGC Act, 1956)

COMPUTER ORGANIZATION AND ARCHITECTURE

Course code	Course name Course	Category L-T-P	Credits
20CS2201	COMPUTER ORGANIZATION AND ARCHITECTURE	PCC 3-0-0	3

Course Content

UNIT-I

Basic Functional blocks of a computer: CPU, Memory, Input -Output Subsystems, Control Unit. **Data Representation:** Number Systems, Signed Number Representation, Fixed and Floating Point Representations, Character Representation.

UNIT-II

ALU: Computer Integer Arithmetic: addition, subtraction, multiplication, division, floating point arithmetic: Addition, subtraction, multiplication, division.

Instruction set architecture of a CPU registers, instruction execution cycle, RTL interpretation of instructions, addressing modes, instruction set. RISC and CISC architecture. Case study instruction sets of some common CPUs.

UNIT-III

CPU control unit design: Introduction to CPU design, Processor Organization, Execution of Complete Execution, Design of Control Unit: hardwired and micro-programmed control, Case study design of a simple hypothetical CPU.

UNIT-IV

Memory system design: Concept of memory: Memory hierarchy, SRAM vs DRAM, Internal organization of memory chips, cache memory: Mapping functions, replacement algorithms, Memory management, virtual memory.

UNIT-V

Input -output subsystems, I/O transfers: programmed I/O, interrupt driven and DMA. I/O Buses, Peripheral devices and their characteristics, Disk Performance

UNIT-VI

Performance enhancement techniques: Pipelining: Basic concepts of pipelining, through put and speedup, pipeline hazards.

Parallel processing: Introduction to parallel processing, Introduction to Network, Cache coherence

Text Books:

1. V. C. Hamacher, Z. G. Vranesic and S. G. Zaky, “Computer Organization,” 5/e, McGraw Hill, 2002.
2. William Stallings, “Computer Organization and Architecture”: Designing for Performance, 8/e, Pearson Education India. 2010.
3. **Morris Mano**, “Computer System Architecture”, Pearson Education India, Third edition. References: A. S. Tanenbaum, “Structured Computer Organization”, 5/e, Prentice Hall of India, 2009.
4. D. A. Patterson and J. L. Hennessy, “Computer Organization and Design,” 4/e, Morgan Kaufmann, 2008.
5. J. L. Hennessy and D. A. Patterson,” Computer Architecture: A Quantitative Approach",4/e, Morgan Kaufmann, 2006.

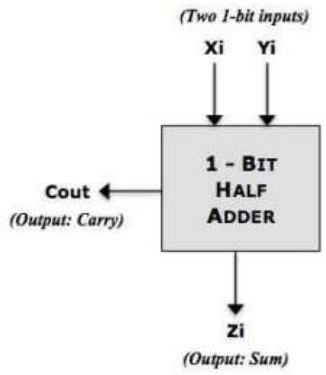
6. D. V. Hall, “Microprocessors and Interfacing”, 2/e, McGraw Hall, 2006 “8086 Assembler Tutorial for Beginners “By Prof. Emerson Giovani Carati.

UNIT – II

Integer Data Computation:

ONE BIT ADDITION: HALF ADDER

1) It is a simple 1-bit adder circuit.
2) It adds two 1-bit inputs Xi & Yi and produces a sum Zi and a Carry Cout.
3) As it does not consider any carry input, it can’t be combined to add large numbers.
4) Hence it is called a Half Adder.

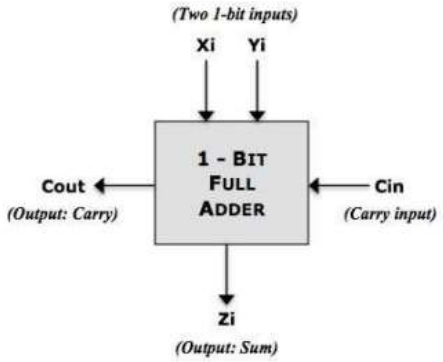


Inputs bits : X_i and Y_i.
Output (Sum) : Z_i
Output (Carry): Cout

Formula:
Sum (Zi) = X_i Ex-Or Y_i.
Carry (Cout) = $X_i.Y_i$

One-bit Addition: Full Adder

1. It is a 1-bit adder circuit.
2. It adds two 1-bit inputs Xi & Yi, along with a Carry Input Cin.
3. It produces a sum Zi and a Carry output Cout.
4. As it considers a carry input, it can be used in combination to add large numbers.
5. Hence it is called a Full Adder.



Inputs bits : X_i and Y_i.
Input Carry : C_{in}

Output (Sum) : Z_i
Output (Carry): C_{out}

Formula for Sum (Z_i):

$$Zi = Xi \div Yi \div Cin$$

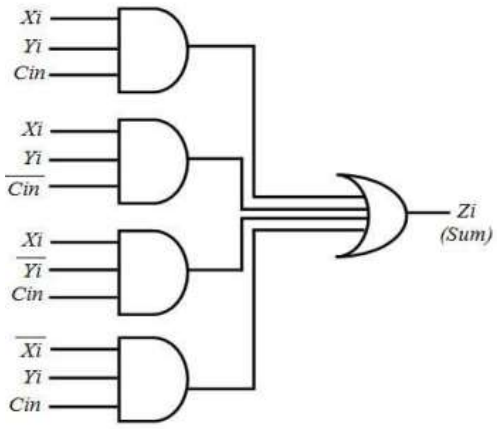
$$\therefore Zi = Xi \cdot Yi \cdot Cin + Xi \cdot Yi \cdot \overline{Cin} + Xi \cdot \overline{Yi} \cdot Cin + \overline{Xi} \cdot Yi \cdot Cin$$

Formula for Carry (C_{out}):

$$Cout = Xi \cdot Yi + Xi \cdot Cin + Yi \cdot Cin$$

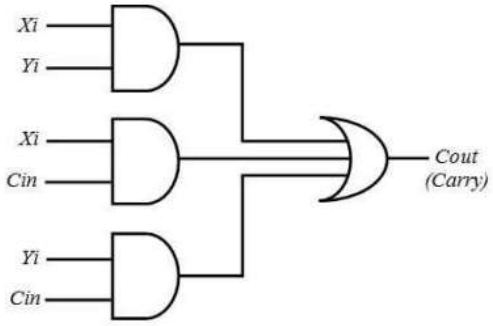
CIRCUIT FOR A FULL ADDER:

Circuit for Sum (Z):



Circuit for Carry (C_{out}):

Multiple Bit Addition: Serial Adder / Ripple Carry Adder:



1. A Full Adder can add two "1-bit" numbers with a Carry input.
2. It produces a "1-bit" Sum and a Carry output.
3. Combining many of these Full Adders, we can add multiple bits.
4. One such method is called Serial Adder.
5. Here, bits are added one-by-one from LSB.
6. The Carry of each stage is propagated (Rippled) into the next stage.
7. Hence, these adders are also called as Ripple Carry Adders.
8. Advantage: they are very easy to construct.
9. Drawback: As addition happens bit-by-bit, they are slow
10. Number of cycles needed for the addition is equal to the number of bits to be added

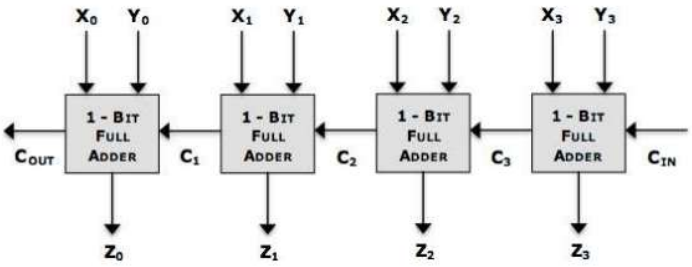
Inputs:
Assume X and Y are two "4-bit" numbers to be added, along with a Carry input C_{in}.

$X = X_0\ X_1\ X_2\ X_3$ (X_0 is the MSB ... X_3 is the LSB)
$Y = Y_0\ Y_1\ Y_2\ Y_3$ (Y_0 is the MSB ... Y_3 is the LSB)
C_{IN} = Carry Input

Outputs:
Assume Z to be a "4-bit" output, and COUT to be the output Carry

$Z = Z_0\ Z_1\ Z_2\ Z_3$ (Z_0 is the MSB ... Z_3 is the LSB)
C_{OUT} = Carry Output

Circuit for 4-bit Serial Adder/ Ripple Carry Adder:



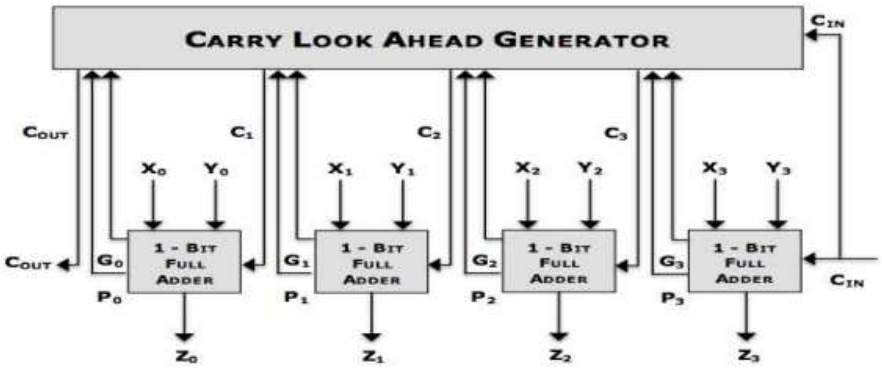
Multiple Bit Addition: Carry Look Ahead Adder / Parallel Adder:

1. It is used to add multiple bits simultaneously.
2. While adding multiple bits, the main issue is that of the intermediate carries.
3. In Serial Adders, we therefore added the bits one-by-one.
4. This allowed the carry at any stage to propagate to the next stage.
5. But this also made the process very slow.
6. If we "PREDICT" the intermediate carries, then all bits can be added simultaneously.
7. This is done by Carry Look Ahead generator.
8. Once all carries are determined beforehand, then all bits can be added simultaneously.
9. Advantage: This makes the addition process extremely fast.
10. Drawback: Circuit is Complex

Inputs:
Assume X and Y are two "4-bit" numbers to be added, along with a Carry input CIN.
$X = X_0\ X_1\ X_2\ X_3$ (X_0 is the MSB … X_3 is the LSB); $Y = Y_0\ Y_1\ Y_2\ Y_3$ & C_{IN} = Carry Input

Outputs:
Assume Z to be a "4-bit" output, and COUT to be the output Carry
$Z = Z_0\ Z_1\ Z_2\ Z_3$ & C_{OUT} = Carry Output

Circuit for 4-bit Carry Look Ahead Adder:



CALCULATIONS:
We can "Predict" (Look Ahead) all the intermediate carries in the following manner.

The Carry at any stage can be calculated as:

$C_i = X_i . Y_i + X_i . C_{in} + Y_i . C_{in}$
$C_i = X_i . Y_i + C_{in} (X_i + Y_i)$

$$\boxed{C_i = G_i + P_i . C_{IN}}$$

Here $Gi = X_i . Y_i$ ... (Generate)
And $Pi = X_i + Y_i$ ... (Propagate)

We need to predict the Carries: C_3, C_2, C_1 and C_0

$$C_3 = G_3 + P_3 C_{IN} \quad \text{... I}$$

$$C_2 = G_2 + P_2 C_3$$

Substituting the value of C3, we get:

$$C_2 = G_2 + P_2 G_3 + P_2 P_3 C_{IN} \quad \text{... II}$$

$$C_1 = G_1 + P_1 C_2$$

Substituting the value of C2, we get:

$$C_1 = G_1 + P_1 G_2 + P_1 P_2 G_3 + P_1 P_2 P_3 C_{IN} \quad \text{... III}$$

$$C_0 = G_0 + P_0 C_1$$

Substituting the value of C1, we get:

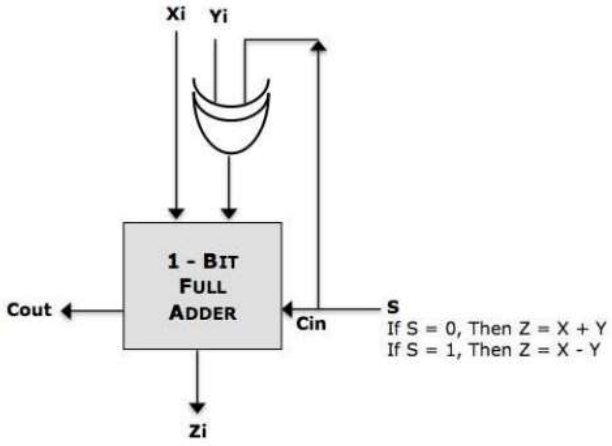
$$C_0 = G_0 + P_0 G_1 + P_0 P_1 G_2 + P_0 P_1 P_2 G_3 + P_0 P_1 P_2 P_3 C_{IN} \quad \text{... IV}$$

From the above four equations, it is clear that the values of all the four Carries (C3, C2, C1, C0) can be determined beforehand even without doing the respective additions. To do this we need the values of all G's (Xi.Yi) and all P's (Xi+Yi) and the original carry input CIN. This is done by the Carry Look Ahead Generator Circuit.

ADDER / SUBTRACTOR CIRCUIT:

1. Subtraction in binary numbers is simply performed by addition of two's complement.
2. That means, a special circuit for subtraction is not needed.
3. The same circuit that is used for Addition, can also be used for subtraction.
4. The following circuit is called Adder/ Subtractor circuit.
5. It can perform Addition as Z = X + Y.
6. It can also perform subtraction as Z = X + (2's Complement of Y)
7. The Variable "S" determines if Addition or Subtraction will be performed.
8. If S = 0, then Addition will be performed.
9. If S = 1, then Subtraction will be performed.

10. If S = 1, then the operation is Z = X + (1's Complement of Y) + 1. Hence Z = X – Y.



Unsigned Multiplication: Conventional Method / Pencil–Paper Method:

1. The Conventional (Pencil-Paper) method is used to multiply two unsigned numbers.
2. When we multiply two "N-bit" numbers, the answer is "2 x N" bits.
3. Three registers A, Q and M, are used for this process.
4. Q contains the Multiplier and M contains the Multiplicand.
5. A (Accumulator) is initialized with 0.
6. At the end of the operation, the Result will be stored in (A & Q) combined.
7. The process involves addition and shifting.

Algorithm

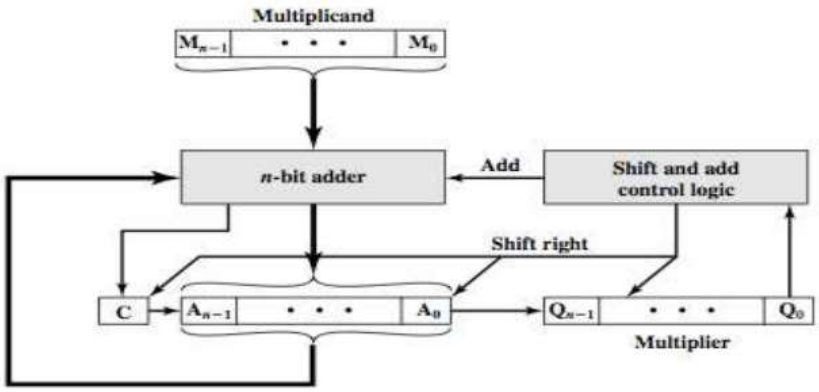
The number of steps required is equal to the number of bits in the multiplier.

1. At each step, examine the current multiplier bit starting from the LSB.
2. If the current multiplier bit is "1", then the Partial-Product is the Multiplicand itself.
3. If the current multiplier bit is "0", then the Partial-Product is the Zero.
4. At each step, ADD the Partial-Product to the Accumulator.
5. Now Right-Shift the Result produced so far (A & Q combined).

Repeat steps 1 to 5 for all bits of the multiplier.

The final answer will be in A & Q combined.

Circuit Diagram for Unsigned Multiplication:



Example: 7 x 6 = 42

				0	1	1	1	… **Multiplicand (7)**
		x		0	1	1	0	… **Multiplier (6)**
				0	0	0	0	… Partial-Product
			0	1	1	1	x	”
		0	1	1	1	x	x	”
+	0	0	0	0	x	x	x	”
	0	1	0	1	0	1	0	… **Result (42)**

SIGNED MULTIPLICATION: BOOTH'S ALGORITHM

1. Booth's Algorithm is used to multiply two SIGNED numbers.
2. When we multiply two "N-bit" numbers, the answer is "2 x N" bits.
3. Three registers A, Q and M, are used for this process.
4. Q contains the Multiplier and M contains the Multiplicand.
5. A (Accumulator) is initialized with 0.
6. At the end of the operation, the Result will be stored in (A & Q) combined.
7. The process involves addition, subtraction and shifting.

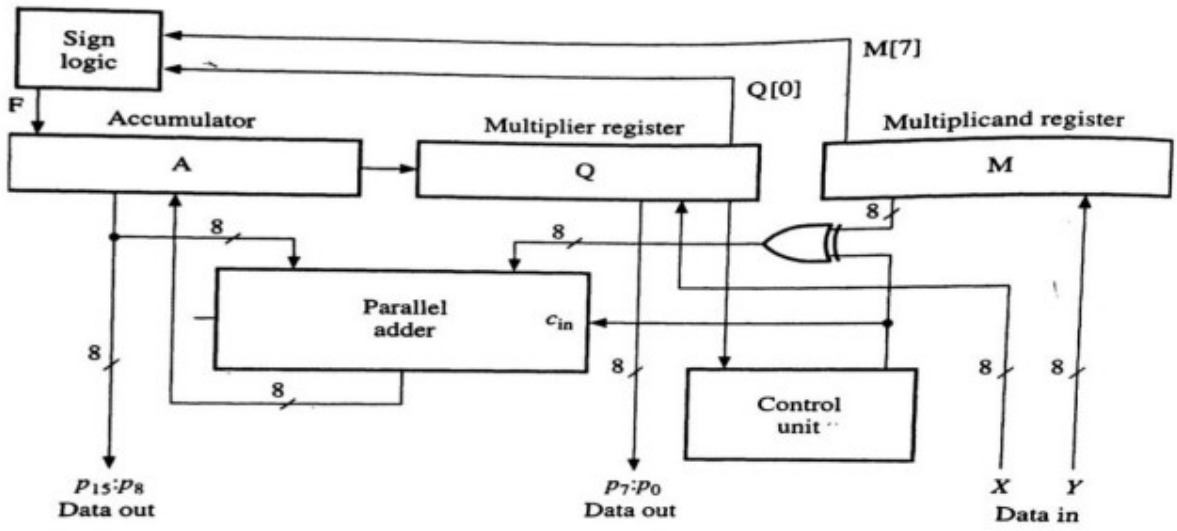
Algorithm:

The number of steps required is equal to the number of bits in the multiplier. At the beginning, consider an imaginary "0" beyond LSB of Multiplier

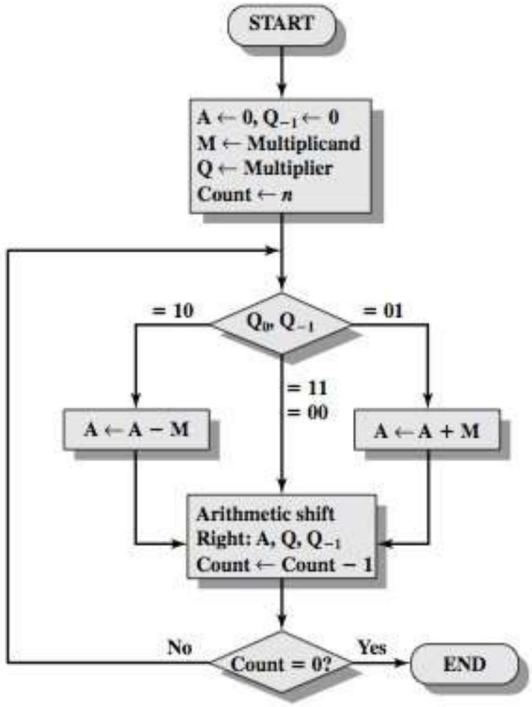
1. At each step, examine two adjacent Multiplier bits from Right to Left.
2. If the transition is from "0 to 1" then Subtract M from A and Right-Shift (A & Q) combined.
3. If the transition is from "1 to 0" then ADD M to A and Right-Shift.
4. If the transition is from "0 to 0" then simply Right-Shift.
5. If the transition is from "1 to 1" then simply Right-Shift. Repeat steps 1 to 5 for all bits of the multiplier.

The final answer will be in A & Q combined.

Circuit Diagram for Signed Multiplication using Booth's Algorithm



FLOWCHART FOR BOOTH'S ALGORITHM:



Example: 7 x 6 = 42

Multiplicand (M): **7 = 0111.** **-7 = 1001** (Two's Complement Form)
Multiplier (Q): **6 = 0110.** **-6 = 1010** (Two's Complement Form)

Step	A Accumulator	Q Multiplier	Q_{-1} Imaginary	M Multiplicand
Initial	0000	0110	0	**0111**
1) (0 ← 0) No Add or Sub Right-Shift	0000 0000	0110 0011	0 0	
2) (1 ← 0) Perform (A - M) Right-Shift	1001 1100	0011 1001	0 1	
3) (1 ← 1) No Add or Sub Right-Shift	1100 1110	1001 0100	1 1	
4) (0 ← 1) Perform (A + M) Right-Shift	0101 **0010** **FINAL ANSWER :-)**	0100 **1010**	1 **0**	

Example: 5 x 7 = 35

Multiplicand (M): **5 = 0101.** **-5 = 1011** (Two's Complement Form)
Multiplier (Q): **7 = 0111.** **-7 = 1001** (Two's Complement Form)

Step	A Accumulator	Q Multiplier	Q_{-1} Imaginary	M Multiplicand
Initial	0000	0111	0	**0101**
1) (1 ← 0) Perform (A - M) Right-Shift	1011 1101	0111 1011	0 1	
2) (1 ← 1) No Add or Sub Right-Shift	1101 1110	1011 1101	1 1	
3) (1 ← 1) No Add or Sub Right-Shift	1110 1111	1101 0110	1 1	
4) (0 ← 1) Perform (A + M) Right-Shift	0100 0010 **FINAL ANSWER :-)**	0110 0011	1 **0**	

Example: 9 x 10 = 90

Multiplicand (M): **9 = 01001.** **-9 = 10111** (Two's Complement Form)
Multiplier (Q): **10 = 01010.** **-10 = 10110** (Two's Complement Form)

Step	A Accumulator	Q Multiplier	Q_{-1} Imaginary	M Multiplicand
Initial	00000	01010	**0**	**01001**
1) (0 ← 0) No Add or Sub Right-Shift	00000 00000	01010 00101	**0** **0**	
2) (1 ← 0) Perform (A - M) Right-Shift	10111 11011	00101 10010	**0** **1**	
3) (0 ← 1) Perform (A + M) Right-Shift	00100 00010	10010 01001	**1** **0**	
4) (1 ← 0) Perform (A - M) Right-Shift	11001 11100	01001 10100	**0** **1**	
5) (0 ← 1) Perform (A + M) Right-Shift	00101 **00010** **FINAL ANSWER :-)**	10100 **11010**	**1** **0**	

Expected Ans: 90 = 00010 11010

Example: -9 x 10 = -90

Multiplicand (M): **-9 = 10111** **9 = 01001.** (Two's Complement Form)
Multiplier (Q): **10 = 01010.** **-10 = 10110** (Two's Complement Form)

Step	A Accumulator	Q Multiplier	Q_{-1} Imaginary	M Multiplicand
Initial	00000	01010	**0**	**10111**
1) (0 ← 0) No Add or Sub Right-Shift	00000 00000	01010 00101	**0** **0**	
2) (1 ← 0) Perform (A - M) Right-Shift	01001 00100	00101 10010	**0** **1**	
3) (0 ← 1) Perform (A + M) Right-Shift	11011 11101	10010 11001	**1** **0**	
4) (1 ← 0) Perform (A - M) Right-Shift	00110 00011	11001 01100	**0** **1**	
5) (0 ← 1) Perform (A + M) Right-Shift	11010 **11101** **FINAL ANSWER :-)**	01100 **00110**	**1** **0**	

Expected Ans: -90 = 11101 00110

Example: 9 x -10 = -90

Multiplicand (M): **9 = 01001** **-9 = 10111.** (Two's Complement Form)
Multiplier (Q): **-10 = 10110** **10 = 01010.** (Two's Complement Form)

Step	A Accumulator	Q Multiplier	Q_{-1} Imaginary	M Multiplicand
Initial	00000	10110	0	01001
1) (0 ← 0) No Add or Sub Right-Shift	00000 00000	10110 01011	0 0	
2) (1 ← 0) Perform (A - M) Right-Shift	10111 11011	01011 10101	0 1	
3) (1 ← 1) No Add or Sub Right-Shift	11011 11101	10101 11010	1 1	
4) (0 ← 1) Perform (A + M) Right-Shift	00110 00011	11010 01101	1 0	
5) (1 ← 0) Perform (A - M) Right-Shift	11010 **11101** **FINAL ANSWER :-)**	01101 **00110**	0 **1**	

Example: -9 x -10 = 90

Multiplicand (M): **-9 = 10111** **9 = 01001.** (Two's Complement Form)
Multiplier (Q): **-10 = 10110** **10 = 01010.** (Two's Complement Form)

Step	A Accumulator	Q Multiplier	Q_{-1} Imaginary	M Multiplicand
Initial	00000	10110	0	10111
1) (0 ← 0) No Add or Sub Right-Shift	00000 00000	10110 01011	0 0	
2) (1 ← 0) Perform (A - M) Right-Shift	01001 00100	01011 10101	0 1	
3) (1 ← 1) No Add or Sub Right-Shift	00100 00010	10101 01010	1 1	
4) (0 ← 1) Perform (A + M) Right-Shift	11001 11100	01010 10101	1 0	
5) (1 ← 0) Perform (A - M) Right-Shift	00101 **00010** **FINAL ANSWER :-)**	10101 **11010**	0 **0**	

RESTORING DIVISION

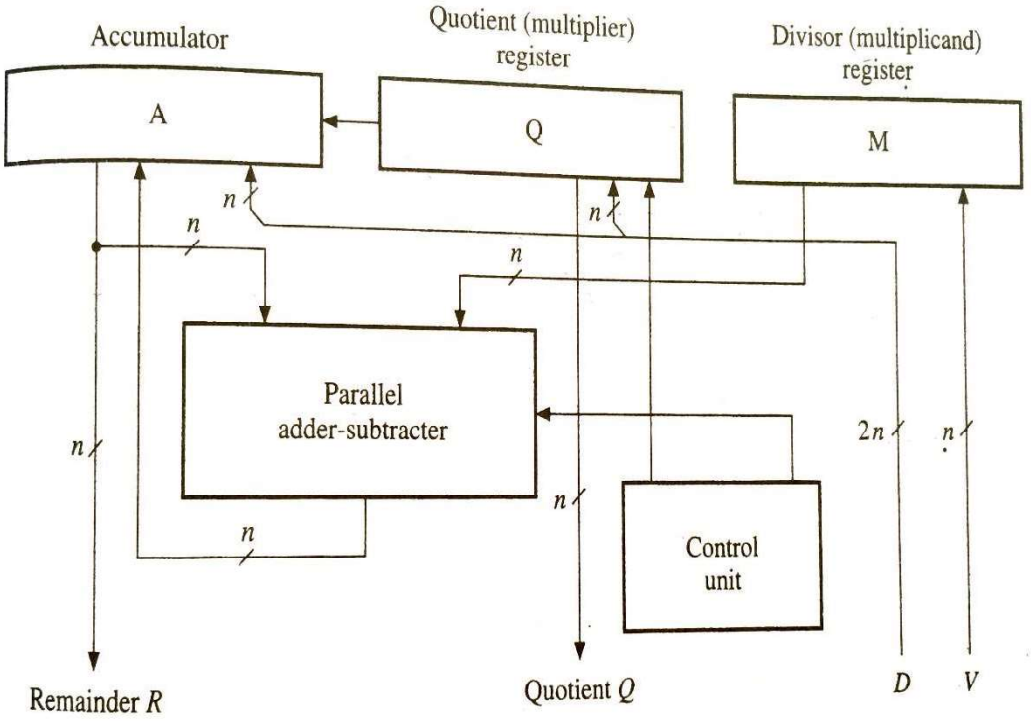
1) Let Q register hold the divided, M register holds the divisor and A register is 0.
2) On completion of the algorithm, Q will get the quotient and A will get the remainder.

Algorithm:

The **number of steps** required is equal to the **number of bits in the Dividend.**

1) At each step, **left shift the dividend by 1 position.**
2) **Subtract the divisor from A** (perform A - M).
3) If the **result is positive** then the step is said to be **"Successful".**
   In this case **quotient bit will be "1"** and **Restoration is NOT Required.**
4) If the **result is negative** then the step is said to be **"Unsuccessful".**
   In this case **quotient bit will be "0".**
   Here Restoration is performed by adding back the divisor.
   Hence the method is called Restoring Division.

Repeat steps 1 to 4 for **all bits** of the Dividend.



Example: (6) / (4)

Dividend (Q) = 6
Divisor (M) = 4
Accumulator (A) = 0

6 = 0110 4 = 0100
-6 = 1010 -4 = 1100

	ACCUMULATOR	DIVIDEND	DIVISOR
	A (0)	Q (6)	M (4)
Initial Values	0000	0110	0100
Step 1:			
Left-Shift	0000	110_	
A – M	+1100		
Unsuccessful (-ve)	1100		
Restoration	0000	1100	
Step 2:			
Left-Shift	0001	100_	
A – M	+1100		
Unsuccessful (-ve)	1101		
Restoration	0001	1000	
Step 3:			
Left-Shift	0011	000_	
A – M	+1100		
Unsuccessful (-ve)	1111		
Restoration	0011	0000	
Step 4:			
Left-Shift	0110	000_	
A – M	+1100		
Successful (+ve)	0010		
No Restoration	0000	0001	
	Remainder(0)	Quotient(1)	

Example: (19) / (7)

Dividend (Q) = 19Divisor (M) = 7 Accumulator (A) = 0

19 = 010011 7 = 000111
-19 = 101101 -7 = 111001

	ACCUMULATOR A (0)	DIVIDEND Q (19)	DIVISOR M (7)
Initial Values	000000	010011	000111
Step 1:			
Left-Shift	000000	10011_	
A – M	+111001		
Unsuccessful (-ve)	111001		
Restoration	000000	100110	
Step 2:			
Left-Shift	000001	00110_	
A – M	+111001		
Unsuccessful (-ve)	111010		
Restoration	000001	001100	
Step 3:			
Left-Shift	000010	01100_	
A – M	+111001		
Unsuccessful (-ve)	111011		
Restoration	000010	011000	
Step 4:			
Left-Shift	000100	11000_	
A – M	+111001		
Unsuccessful (-ve)	111101		
Restoration	000100	110000	
Step 5:			
Left-Shift	001001	10000_	
A – M	+111001		
Successful (+ve)	000010		
No Restoration	000010	100001	
Step 6:			
Left-Shift	000101	00001_	
A – M	+111001		
Unsuccessful (-ve)	111110		
Restoration	000101	000010	
	Remainder(5)	Quotient(2)	

Example: (23) / (3)

Dividend (Q) = 23Divisor (M) = 3 Accumulator (A) = 0

23 = 010111 3 = 000011
-23 = 101001 -3 = 111101

	ACCUMULATOR	DIVIDEND	DIVISOR
	A (0)	Q (23)	M (3)
Initial Values	000000	010111	000011
Step 1:			
Step 2:			
Step 3:			
Step 4:			
Step 5:			
Step 6:			
	Remainder(2)	Quotient(7)	

NON-RESTORING DIVISION

1) Let Q register hold the divided, M register holds the divisor and A register is 0.
2) On completion of the algorithm, Q will get the quotient and A will get the remainder.

Algorithm:

The **number of steps** required is equal to the **number of bits in the Dividend**.

1) At each step, **left shift the dividend by 1 position**.
2) **Subtract the divisor from A** (perform A - M).
3) If the **result is positive** then the step is said to be **"Successful"**.

In this case **quotient bit will be "1"** and **Restoration is NOT Required**.

The **Next Step** will also be **Subtraction**.

4) If the **result is negative** then the step is said to be **"Unsuccessful"**.

In this case **quotient bit will be "0".**

Here Restoration is NOT Performed.

Instead the next step will be ADDITION in place of subtraction.

As restoration is not performed, the method is called Non-Restoring Division.

Repeat steps 1 to 4 for **all bits** of the Dividend.

Note: Restoration may only be performed in the FINAL step, if it is unsuccessful, as there is no next step thereafter. In such a case, restoration is performed by adding back the divisor.

Example: (7) / (5)

Dividend (Q) = 7
Divisor (M) = 5
Accumulator (A) = 0

 7 = 0111 5 = 0101
-7 = 1001 -5 = 1011

	ACCUMULATOR	DIVIDEND	DIVISOR
	A (0)	Q (7)	M (5)
Initial Values	0000	0111	0101
Step 1:			
Left-Shift	0000	111_	
A – M	+1011		
Unsuccessful (-ve)	1011	1110	
Next Step: "Add"			
Step 2:			
Left-Shift	0111	110_	
A + M	+0101		
Unsuccessful (-ve)	1100	1100	
Next Step: "Add"			
Step 3:			
Left-Shift	1001	100_	
A + M	+0101		
Unsuccessful (-ve)	1110	1000	
Next Step: "Add"			
Step 4:			
Left-Shift	1101	000_	
A + M	+0101		
Successful (+ve)	0010	0001	
	Remainder(2)	Quotient(1)	

Example: (7) / (3)

Dividend (Q) = 7
Divisor (M) = 3
Accumulator (A) = 0

7 = 0111 3 = 0011
-7 = 1001 -3 = 1101

	ACCUMULATOR	DIVIDEND	DIVISOR
	A (0)	Q (7)	M (3)
Initial Values	0000	0111	0011
Step 1:			
Left-Shift	0000	111_	
A – M	+1101		
Unsuccessful (-ve)	1101	1110	
Next Step: "Add"			
Step 2:			
Left-Shift	1011	110_	
A + M	+0011		
Unsuccessful (-ve)	1110	1100	
Next Step: "Add"			
Step 3:			
Left-Shift	1101	100_	
A + M	+0011		
Successful (+ve)	0000	1001	
Next Step: "Sub"			
Step 4:			
Left-Shift	0001	001_	
A - M	+1101		
Unsuccessful (-ve)	1110		
Restore: (Add back divisor)	0001	0010	
	Remainder(1)	Quotient(2)	

Example: (19) / (4)

Dividend (Q) = 19Divisor (M) = 4 Accumulator (A) = 0

19 = 010011 4 = 000100
-19 = 101101 -4 = 111100

	ACCUMULATOR	DIVIDEND	DIVISOR
	A (0)	Q (19)	M (4)
Initial Values	000000	010011	000100
Step 1:			
Left-Shift	000000	10011_	
A – M	+111100		
Unsuccessful (-ve)	111100	100110	
Next Step: "Add"			
Step 2:			
Left-Shift	111001	00110_	
A + M	+000100		
Unsuccessful (-ve)	111101	001100	
Next Step: "Add"			
Step 3:			
Left-Shift	111010	01100_	
A + M	+000100		
Unsuccessful (-ve)	111110	011000	
Next Step: "Add"			
Step 4:			
Left-Shift	111100	11000_	
A + M	+000100		
Successful (+ve)	000000	110001	
Next Step: "Sub"			
Step 5:			
Left-Shift	000001	10001_	
A – M	+111100		
Unsuccessful (-ve)	111101	100010	
Next Step: "Add"			
Step 6:			
Left-Shift	111011	00010_	
A + M	+000100		
Unsuccessful (-ve)	111111		
Restore: (Add back divisor)	000011	000100	
	Remainder(3)	Quotient(4)	

Example: (23) / (3)

Dividend (Q) = 23Divisor (M) = 3 Accumulator (A) = 0

23 = 010111 3 = 000011
-23 = 101001 -3 = 111101

	ACCUMULATOR	DIVIDEND	DIVISOR
	A (0)	Q (23)	M (3)
Initial Values	000000	010111	000011
Step 1:			
Step 2:			
Step 3:			
Step 4:			
Step 5:			
Step 6:			
	Remainder(2)	Quotient(7)	

RESTORING DIVISION FOR SIGNED NUMBERS

1) Let M register hold the divisor, Q register hold the divided.
2) A register should be the signed extension of Q.
3) On completion of the algorithm, Q will get the quotient and A will get the remainder.

Algorithm:

The **number of steps** required is equal to the **number of bits in the Dividend**.

1) At each step, **left shift the dividend by 1 position**.

2) If Sign of A and M is the same then **Subtract the divisor from A** (perform A - M),
Else **Add M to A**

3) After the operation,

If **Sign of A remains the same** or the **dividend** (in A and Q) **becomes zero**,
then the step is said to be **"Successful"**.
In this case **quotient bit will be "1"** and **Restoration is NOT Required**.

4) If **Sign of A changes**, then the step is said to be **"Unsuccessful"**.
In this case **quotient bit will be "0".**
Here Restoration is Performed.
Hence, the method is called Restoring Division.

Repeat steps 1 to 4 for **all bits** of the Dividend.

Note: The result of this algorithm is such that, the quotient will always be positive and the remainder will get the same sign as the dividend.

Example: (5) / (3)

5 = 0101 3 = 0011
-5 = 1011 -3 = 1101

	ACCUMULATOR	DIVIDEND	DIVISOR
	A (Sign Extension)	Q (5)	M (3)
Initial Values	0000	0101	0011
Step 1:			
Left-Shift	0000	101_	
Sign (A, M) Same: A - M	+1101		
Sign Changes: Unsuccessful	1101		
Restore	0000	1010	
Step 2:			
Left-Shift	0001	010_	
Sign (A, M) Same: A - M	+1101		
Sign Changes: Unsuccessful	1110		
Restore	0001	0100	
Step 3:			
Left-Shift	0010	100_	
Sign (A, M) Same: A – M	+1101		
Sign Changes: Unsuccessful	1111		
Restore	0010	1000	
Step 4:			
Left-Shift	0101	000_	
Sign (A, M) Same: A - M	+1101		
Sign still Same: Successful	0010		
Restore not required	0010	0001	
	Remainder(2)	Quotient(1)	

Example: (-19) / (7)

19 = 010011 7 = 000111
-19 = 101101 -7 = 111001

	ACCUMULATOR	DIVIDEND	DIVISOR
	A (Sign Extension)	Q (-19)	M (7)
Initial Values	111111	101101	000111
Step 1:			
Left-Shift	111111	01101_	
Sign (A, M) Different: A + M	+000111		
Sign Changes: Unsuccessful	000110		
Restore	111111	011010	
Step 2:			
Left-Shift	111110	11010_	
Sign (A, M) Different: A + M	+000111		
Sign Changes: Unsuccessful	000101		
Restore	111110	110100	
Step 3:			
Left-Shift	111101	10100_	
Sign (A, M) Different: A + M	+000111		
Sign Changes: Unsuccessful	000100		
Restore	111101	101000	
Step 4:			
Left-Shift	111011	01000_	
Sign (A, M) Different: A + M	+000111		
Sign Changes: Unsuccessful	000010		
Restore	111011	010000	
Step 5:			
Left-Shift	110110	10000_	
Sign (A, M) Different: A + M	+000111		
Sign still Same: Successful	111101		
Restore not required	111101	100001	
Step 6:			
Left-Shift	111011	00001_	
Sign (A, M) Different: A + M	+000111		
Sign Changes: Unsuccessful	000010		
Restore	111011	000010	
	Remainder(-5)	Quotient(2)	

Example: (-8) / (-4)

8 = 01000 4 = 00100
-8 = 11000 -4 = 11100

	ACCUMULATOR	DIVIDEND	DIVISOR
	A (Sign Extension)	Q (-8)	M (-4)
Initial Values	11111	11000	11100
Step 1:			
Left-Shift	11111	1000_	
Sign (A, M) Same: A - M	+00100		
Sign Changes: Unsuccessful	00011		
Restore	11111	10000	
Step 2:			
Left-Shift	11111	0000_	
Sign (A, M) Same: A - M	+00100		
Sign Changes: Unsuccessful	00011		
Restore	11111	00000	
Step 3:			
Left-Shift	11110	0000_	
Sign (A, M) Same: A - M	+00100		
Sign Changes: Unsuccessful	00010		
Restore	11110	00000	
Step 4:			
Left-Shift	11100	0000_	
Sign (A, M) Same: A - M	+00100		
Dividend(A,Q)=0: Successful	00000		
Restore not required	00000	00001	
Step 5:			
Left-Shift	00000	0001_	
Sign (A, M) Different: A + M	+11100		
Sign Changes: Unsuccessful	11100		
Restore	00000	00010	
	Remainder(0)	Quotient(2)	

Example: (-14) / (2)

14 = 01110 2 = 00010
-14 = 10010 -2 = 11110

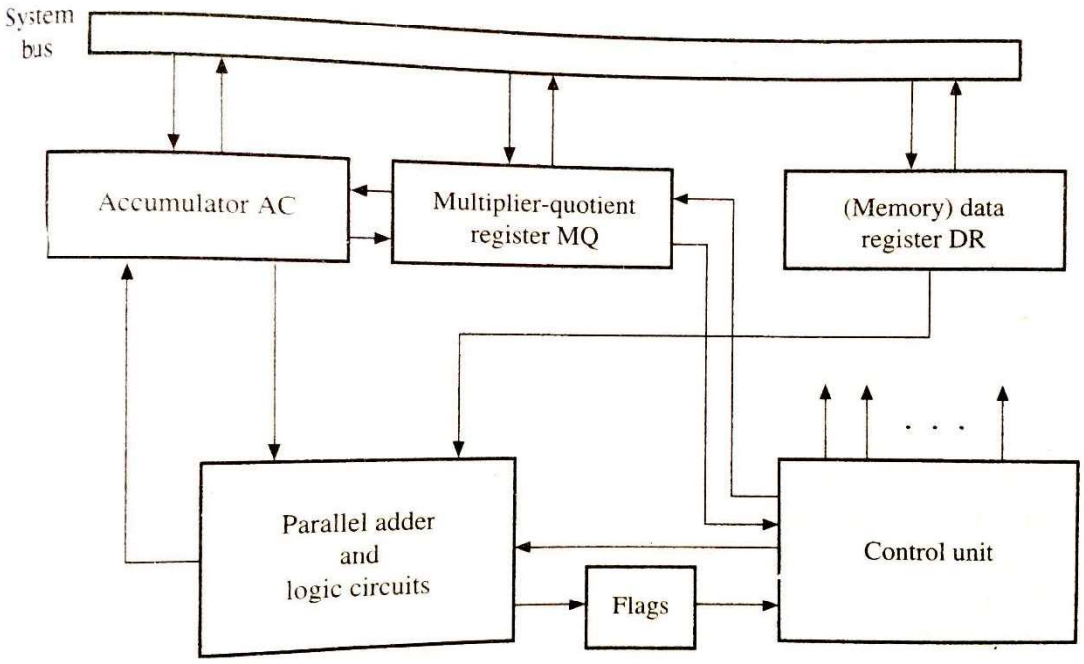
	ACCUMULATOR	DIVIDEND	DIVISOR
	A (Sign Extension)	Q (-14)	M (2)
Initial Values	11111	10010	00010
Step 1:			
Left-Shift	11111	0010_	
Sign (A, M) Different: A + M	+00010		
Sign Changes: Unsuccessful	00001		
Restore	11111	00100	
Step 2:			
Left-Shift	11110	0100_	
Sign (A, M) Different: A + M	+00010		
Sign Changes: Unsuccessful	00000	*Note that dividend part in (A, Q) is not zero*	
Restore	11110	01000	
Step 3:			
Left-Shift	11100	1000_	
Sign (A, M) Different: A + M	+00010		
Sign still Same: Successful	11110		
Restore not required	11110	10001	
Step 4:			
Left-Shift	11101	0001_	
Sign (A, M) Different: A + M	+00010		
Sign still Same: Successful	11111		
Restore not required	11111	00011	
Step 5:			
Left-Shift	11110	0011_	
Sign (A, M) Different: A + M	+00010		
Dividend(A,Q)=0: Successful	00000	*Note that dividend part in (A, Q) is Zero!*	
Restore not required	00000	00111	
	Remainder(0)	Quotient(7)	

Example: (14) / (-2)

14 = 01110 2 = 00010
-14 = 10010 -2 = 11110

	ACCUMULATOR	DIVIDEND	DIVISOR
	A (Sign Extension)	Q (-14)	M (-2)
Initial Values	00000	01110	11110
Step 1:			
Step 2:			
Step 3:			
Step 4:			
Step 5:			
	Remainder(0)	Quotient(7)	

TYPICAL ALU DESIGN



Addition	AC := AC + DR
Subtraction	AC := AC – DR
Multiplication	AC.MQ := DR × MQ
Division	AC.MQ := MQ/DR
AND	AC := AC ***and*** DR
OR	AC := AC ***or*** DR
EXCLUSIVE-OR	AC := AC ***xor*** DR
NOT	AC := ***not***(AC)

RESTORING DIVISION FOR SIGNED NUMBERS

1) Let M register hold the divisor, Q register hold the divided.
2) A register should be the signed extension of Q.
3) On completion of the algorithm, Q will get the quotient and A will get the remainder.

Algorithm:

The **number of steps** required is equal to the **number of bits in the Dividend**.

1) At each step, **left shift the dividend by 1 position**.
2) If Sign of A and M is the same then **Subtract the divisor from A** (perform A - M),
   Else **Add M to A**
3) After the operation,

If **Sign of A remains the same** or the **dividend** (in A and Q) **becomes zero**,then the step is said to be **"Successful"**.

In this case **quotient bit will be "1"** and **Restoration is NOT Required**.

4) If **Sign of A changes**, then the step is said to be **"Unsuccessful"**.
   In this case **quotient bit will be "0".**

Here Restoration is Performed.

Hence, the method is called Restoring Division.

Repeat steps 1 to 4 for **all bits** of the Dividend.

Note: The result of this algorithm is such that, the quotient will always be positive and the remainder will get the same sign as the dividend.

Example: (5) / (3)

5 = 0101 3 = 0011
-5 = 1011 -3 = 1101

	ACCUMULATOR	DIVIDEND	DIVISOR
	A (Sign Extension)	Q (5)	M (3)
Initial Values	0000	0101	0011
Step 1:			
Left-Shift	0000	101_	
Sign (A, M) Same: A - M	+1101		
Sign Changes: Unsuccessful	1101		
Restore	0000	1010	
Step 2:			
Left-Shift	0001	010_	
Sign (A, M) Same: A - M	+1101		
Sign Changes: Unsuccessful	1110		
Restore	0001	0100	
Step 3:			
Left-Shift	0010	100_	
Sign (A, M) Same: A – M	+1101		
Sign Changes: Unsuccessful	1111		
Restore	0010	1000	
Step 4:			
Left-Shift	0101	000_	
Sign (A, M) Same: A - M	+1101		
Sign still Same: Successful	0010		
Restore not required	0010	0001	
	Remainder(2)	Quotient(1)	

Example: (-19) / (7)

19 = 010011 7 = 000111
-19 = 101101 -7 = 111001

	ACCUMULATOR	DIVIDEND	DIVISOR
	A (Sign Extension)	Q (-19)	M (7)
Initial Values	111111	101101	000111
Step 1:			
Left-Shift	111111	01101_	
Sign (A, M) Different: A + M	+000111		
Sign Changes: Unsuccessful	000110		
Restore	111111	011010	
Step 2:			
Left-Shift	111110	11010_	
Sign (A, M) Different: A + M	+000111		
Sign Changes: Unsuccessful	000101		
Restore	111110	110100	
Step 3:			
Left-Shift	111101	10100_	
Sign (A, M) Different: A + M	+000111		
Sign Changes: Unsuccessful	000100		
Restore	111101	101000	
Step 4:			
Left-Shift	111011	01000_	
Sign (A, M) Different: A + M	+000111		
Sign Changes: Unsuccessful	000010		
Restore	111011	010000	
Step 5:			
Left-Shift	110110	10000_	
Sign (A, M) Different: A + M	+000111		
Sign still Same: Successful	111101		
Restore not required	111101	100001	
Step 6:			
Left-Shift	111011	00001_	
Sign (A, M) Different: A + M	+000111		
Sign Changes: Unsuccessful	000010		
Restore	111011	000010	
	Remainder(-5)	Quotient(2)	

Example: (-8) / (-4)

8 = 01000 4 = 00100
-8 = 11000 -4 = 11100

	ACCUMULATOR	DIVIDEND	DIVISOR
	A (Sign Extension)	Q (-8)	M (-4)
Initial Values	11111	11000	11100
Step 1:			
Left-Shift	11111	1000_	
Sign (A, M) Same: A - M	+00100		
Sign Changes: Unsuccessful	00011		
Restore	11111	10000	
Step 2:			
Left-Shift	11111	0000_	
Sign (A, M) Same: A - M	+00100		
Sign Changes: Unsuccessful	00011		
Restore	11111	00000	
Step 3:			
Left-Shift	11110	0000_	
Sign (A, M) Same: A - M	+00100		
Sign Changes: Unsuccessful	00010		
Restore	11110	00000	
Step 4:			
Left-Shift	11100	0000_	
Sign (A, M) Same: A - M	+00100		
Dividend(A,Q)=0: Successful	00000		
Restore not required	00000	00001	
Step 5:			
Left-Shift	00000	0001_	
Sign (A, M) Different: A + M	+11100		
Sign Changes: Unsuccessful	11100		
Restore	00000	00010	
	Remainder(0)	Quotient(2)	

Example: (-14) / (2)

14 = 01110 2 = 00010
-14 = 10010 -2 = 11110

	ACCUMULATOR	DIVIDEND	DIVISOR
	A (Sign Extension)	Q (-14)	M (2)
Initial Values	11111	10010	00010
Step 1:			
Left-Shift	11111	0010_	
Sign (A, M) Different: A + M	+00010		
Sign Changes: Unsuccessful	00001		
Restore	11111	00100	
Step 2:			
Left-Shift	11110	0100_	
Sign (A, M) Different: A + M	+00010		
Sign Changes: Unsuccessful	00000	*Note that dividend part in (A, Q) is not zero*	
Restore	11110	01000	
Step 3:			
Left-Shift	11100	1000_	
Sign (A, M) Different: A + M	+00010		
Sign still Same: Successful	11110		
Restore not required	11110	10001	
Step 4:			
Left-Shift	11101	0001_	
Sign (A, M) Different: A + M	+00010		
Sign still Same: Successful	11111		
Restore not required	11111	00011	
Step 5:			
Left-Shift	11110	0011_	
Sign (A, M) Different: A + M	+00010		
Dividend(A,Q)=0: Successful	00000	*Note that dividend part in (A, Q) is Zero!*	
Restore not required	00000	00111	
	Remainder(0)	Quotient(7)	

Example: (14) / (-2)

14 = 01110 2 = 00010
-14 = 10010 -2 = 11110

	ACCUMULATOR	DIVIDEND	DIVISOR
	A (Sign Extension)	Q (-14)	M (-2)
Initial Values	00000	01110	11110
Step 1:			
Step 2:			
Step 3:			
Step 4:			
Step 5:			
	Remainder(0)	Quotient(7)	

FLOATING POINT NUMBERS

- In some numbers, which have a fractional part, the position of the decimal point is not fixed as the number of bits before (or after) the decimal point may vary.
- **Eg: 0010.01001, 0.0001101, -1001001.01** etc.
- As shown above, the position of the decimal point is not fixed, instead it **"floats"** in the number.

- Such numbers are called Floating Point Numbers.
- Floating Point Numbers are stored in a "Normalized" form.

NORMALIZATION OF A FLOATING POINT NUMBER

- Normalization is the process of shifting the point, left or right, so that there is only one non-zero digit to the left of the point.

Floating Point Number		Normalized Number
01010.01	⟶	$(-1)^{0} \times 1.\mathbf{01001} \times 2^{3}$
11111.01	⟶	$(-1)^{0} \times 1.\mathbf{111101} \times 2^{4}$
0.00101	⟶	$(-1)^{0} \times 1.\mathbf{01} \times 2^{-3}$
-10.01	⟶	$(-1)^{1} \times 1.\mathbf{001} \times 2^{1}$

- As seen above a Normalized Form of a number is:

$$(-1)^{S} \times 1.M \times 2^{E}$$

Where: S = Sign, M = Mantissa and E = Exponent.

- As Normalized numbers are of the 1.M format, the "**1**" is not actually stored, it is instead **assumed**. This saves the storage space by 1 bit for each number.

- Also the Exponent is stored in the biased form by adding an appropriate bias value to it so that -ve exponents can be easily represented.

Advantages of Normalization.

1. Storing all numbers in a standard for makes **calculations easier** and **faster**.
2. By **not storing** the **1** (of 1.M format) for a number, considerable **storage space** is **saved**.
3. The **exponent** is **biased** so there is **no need** for **storing** its **sign bit** (as the biased exponent cannot be -ve).

SHORT REAL FORMAT / SINGLE PRECISION FORMAT / IEEE 754: 32 BIT FORMAT

S	Biased Exponent	Mantissa
(1)	(8) Bias value = 127	(23 bits)

- **32 bits** are used to store the **number**.
- **23 bits** are used for the **Mantissa**.
- **8 bits** are used for the Biased **Exponent**.
- **1 bit** used for the **Sign** of the number.
- The **Bias** value is $(\mathbf{127})_{10}$.
- The range is $\pm\mathbf{1 \times 10^{-38}}$ **to** $\pm\mathbf{3 \times 10^{38}}$ approximately.
- It is called as the **Single Precision Format** for Floating-Point Numbers.

LONG REAL FORMAT / DOUBLE PRECISION FORMAT / IEEE 754: 64 BIT FORMAT

S	Biased Exponent	Mantissa
(1)	(11) Bias value = 1023	(52 bits)

- **64 bits** are used to store the **number**.
- **52 bits** are used for the **Mantissa**.
- **11 bits** are used for the Biased **Exponent**.
- **1 bit** used for the **Sign** of the number.
- The **Bias** value is $(\mathbf{1023})_{10}$.
- The range is $\pm\mathbf{10^{-308}}$ **to** $\pm\mathbf{10^{308}}$ approximately.
- It is called as the **Double Precision Format** for Floating-Point Numbers.

Numericals on Floating Point Number Representation

1) Convert 2A3BH into Short Real and Temp Real formats {Exam question}

Short real:

Converting the number into binary we get:
0010 1010 0011 1011

Normalizing the number we get:
$(-1)^{0}$ **x** 1.**0101000111011** x 2^{13}
Here S = 0; M = 0101000111011; True Exponent = 13.

Bias value for Short Real format is 127:
Biased Exponent (BE) = True Exponent + Bias
= 13 + 127
= 140.

Converting the Biased exponent into binary we get:
Biased Exponent (BE) = (1000 1100)

Representing in the required format we get:

0	**10001100**	**01010001110110O…**
S	Biased Exp	Mantissa
(1)	(8)	(23)

Converting the number into hexadecimal form we get:
4628EC00H … 32 bits.

Temp real:

Bias value for Temp Real format is 16383:
Biased Exponent (BE) = True Exponent + Bias
= 13 + 16383
= 16396.

Converting the Biased exponent into binary we get:
Biased Exponent (BE) = (100 0000 0000 1100)

Representing in the required format we get:

0	100000000001100	**1**010100011101100…
S	Biased Exp	Mantissa
(1)	(15)	(64)

Converting the number into hexadecimal form we get:
400C A8EC 0000 0000 0000H … 80 bits.

2) Convert $(12.125)_d$ into Temp Real format {Exam question}

Temp real:

Converting the number into binary we get:
1100.001

Normalizing the number we get:
$(-1)^{0}$ **x** $1.\mathbf{100001} \times 2^{3}$
Here S = 0; M = 100001; True Exponent = 3.

Bias value for Temp Real format is 16383:
Biased Exponent (BE) = True Exponent + Bias
= 3 + 16383
= 16386.

Converting the Biased exponent into binary we get:
Biased Exponent (BE) = (100 0000 0000 0010)

Representing in the required format we get:

0	100000000000010	**1**10000100000…
S	Biased Exp	Mantissa
(1)	(8)	(23)

Converting the number into hexadecimal form we get:
4002 C200 0000 0000 0000H … 80 bits.

FLOATING POINT NUMBERS

- In some numbers, which have a fractional part, the position of the decimal point is not fixed as the number of bits before (or after) the decimal point may vary.
- **Eg: 0010.01001, 0.0001101, -1001001.01** etc.
- As shown above, the position of the decimal point is not fixed, instead it **"floats"** in the number.

- Such numbers are called Floating Point Numbers.
- Floating Point Numbers are stored in a "Normalized" form.

NORMALIZATION OF A FLOATING POINT NUMBER

- Normalization is the process of shifting the point, left or right, so that there is only one non-zero digit to the left of the point.

Floating Point Number		Normalized Number
01010.01	→	$(-1)^{0} \times 1.\mathbf{01001} \times 2^{3}$
11111.01		$(-1)^{0} \times 1.\mathbf{111101} \times 2^{4}$
0.00101		$(-1)^{0} \times 1.\mathbf{01} \times 2^{-3}$
-10.01		$(-1)^{1} \times 1.\mathbf{001} \times 2^{1}$

- As seen above a Normalized Form of a number is:

$$\mathbf{(-1)^{S} \times 1.M \times 2^{E}}$$

Where: S = Sign, M = Mantissa and E = Exponent.

- As Normalized numbers are of the 1.M format, the "**1**" is not actually stored, it is instead **assumed.** This saves the storage space by 1 bit for each number.

- Also the Exponent is stored in the biased form by adding an appropriate bias value to it so that -ve exponents can be easily represented.

Advantages of Normalization.

1. Storing all numbers in a standard for makes **calculations easier** and **faster**.
2. By **not storing** the **1** (of 1.M format) for a number, considerable **storage space** is **saved**.
3. The **exponent** is **biased** so there is **no need** for **storing** its **sign bit** (as the biased exponent cannot be -ve).

SHORT REAL FORMAT / SINGLE PRECISION FORMAT / IEEE 754: 32 BIT FORMAT

S	Biased Exponent	Mantissa
(1)	(8) Bias value = 127	(23 bits)

- **32 bits** are used to store the **number**.
- **23 bits** are used for the **Mantissa**.
- **8 bits** are used for the Biased **Exponent**.
- **1 bit** used for the **Sign** of the number.
- The **Bias** value is $(\mathbf{127})_{10}$.
- The range is $\pm\mathbf{1 \times 10^{-38}}$ **to** $\pm\mathbf{3 \times 10^{38}}$ approximately.
- It is called as the **Single Precision Format** for Floating-Point Numbers.

LONG REAL FORMAT / DOUBLE PRECISION FORMAT / IEEE 754: 64 BIT FORMAT

S	Biased Exponent	Mantissa
(1)	(11) Bias value = 1023	(52 bits)

- **64 bits** are used to store the **number**.
- **52 bits** are used for the **Mantissa**.
- **11 bits** are used for the Biased **Exponent**.
- **1 bit** used for the **Sign** of the number.
- The **Bias** value is $(\mathbf{1023})_{10}$.
- The range is $\pm\mathbf{10^{-308}}$ **to** $\pm\mathbf{10^{308}}$ approximately.
- It is called as the **Double Precision Format** for Floating-Point Numbers.

Numericals on Floating Point Number Representation

1) Convert 2A3BH into Short Real and Temp Real formats {Exam question}

Short real:

Converting the number into binary we get:
0010 1010 0011 1011

Normalizing the number we get:
$(-1)^{0}$ **x** 1.**0101000111011** x 2^{13}
Here S = 0; M = 0101000111011; True Exponent = 13.

Bias value for Short Real format is 127:
Biased Exponent (BE) = True Exponent + Bias
= 13 + 127
= 140.

Converting the Biased exponent into binary we get:
Biased Exponent (BE) = (1000 1100)

Representing in the required format we get:

0	**10001100**	**010100011101100…**
S	Biased Exp	Mantissa
(1)	(8)	(23)

Converting the number into hexadecimal form we get:
4628EC00H … 32 bits.

Temp real:

Bias value for Temp Real format is 16383:
Biased Exponent (BE) = True Exponent + Bias
= 13 + 16383
= 16396.

Converting the Biased exponent into binary we get:
Biased Exponent (BE) = (100 0000 0000 1100)

Representing in the required format we get:

0	100000000001100	**1**010100011101100…
S	Biased Exp	Mantissa
(1)	(15)	(64)

Converting the number into hexadecimal form we get:
400C A8EC 0000 0000 0000H … 80 bits.

2) Convert $(12.125)_d$ into Temp Real format {Exam question}

Temp real:

Converting the number into binary we get:
1100.001

Normalizing the number we get:
$(-1)^{0}$ **x** $1.\mathbf{100001} \times 2^{3}$
Here S = 0; M = 100001; True Exponent = 3.

Bias value for Temp Real format is 16383:
Biased Exponent (BE) = True Exponent + Bias
= 3 + 16383
= 16386.

Converting the Biased exponent into binary we get:
Biased Exponent (BE) = (100 0000 0000 0010)

Representing in the required format we get:

0	100000000000010	**1**10000100000…
S	Biased Exp	Mantissa
(1)	(8)	(23)

Converting the number into hexadecimal form we get:4002 C200 0000 0000 0000H … 80 bits.

Instruction Set Architecture (ISA)

DILEEP KUMAR K || Asst. Prfessor || Dept. of CSE || RGUKT SKLM

An Instruction

- An instruction can be considered as a word in processor's language
- What information an Instruction should covey to the CPU?

opcode	Addr of OP1	Addr of OP2	Dest Addr	Addr of next intr

- The instruction is too long if everything specified explicitly
  - More space in memory
  - Longer execution time
- How can you reduced the size of an instruction?
  - Specifying information implicitly
  - How?
- Using PC, ACC, GPRs, SP

Instruction Set Architecture

"Instruction Set Architecture is the structure of a computer that a machine language programmer (or a compiler) must understand to write a correct (timing independent) program for that machine."

- The ISA defines:
  - Operations that the processor can execute
  - Data Transfer mechanisms + how to access data
  - Control Mechanisms (branch, jump, etc)
  - "Contract" between programmer/compiler and HW
- ISA is important:
  - Not only from the programmer's perspective.
  - From processor design and implementation perspectives as well.

Instruction Set Architecture

- Programmer visible part of a processor:
  - Registers (where are data located?)
  - Addressing Modes (how is data accessed?)
  - Instruction Format (how are instructions specified?)
  - Exceptional Conditions (what happens if something goes wrong?)
  - Instruction Set (what operations can be performed?)

ISA Design Choices

- Types of operations supported
  - e.g. arithmetic/logical, data transfer, control transfer, system, floating-point, decimal (BCD), string
- Types of operands supported
  - e.g. byte, character, digit, halfword, word (32-bits), doubleword, floating-point number
- Types of operand storage allowed
  - e.g. stack, accumulator, registers, memory
- Implicit vs. explicit operands in instructions and number of each
- Orthogonality of operands, operand location, and addressing modes

Classification of ISAs

- Determined by the means used for storing data in CPU:
- The major choices are:
  - A stack, an accumulator, or a set of registers
- Stack architecture:
  - Operands are implicitly on top of the stack
- Accumulator architecture:
  - One operand is in the accumulator (register) and the others are elsewhere
  - Essentially this is a 1 register machine
  - Found in older machines...
- General purpose registers:
  - Operands are in registers or specific memory locations.

Evolution of ISAs

- Accumulator Architectures (EDSAC, IBM-701)
- Special-purpose Register Architectures
- General purpose registers architectures
  - Register-memory (IBM 360, DEC PDP-11, Intel 80386
  - Register-register (load-store) (CDC6600, MIPS, DEC alpha)

Classification of ISAs

Types of Architecture	Source Operands	Destination
Stack	Top two elements in stack	Top of stack
Accumulator	Accumulator (1) Memory (other)	Accumulator
Register set	Register or Memory	Register or Memory

Comparison of Architectures

Consider the operation: C = A + B

Stack	Accumulator	Register-Memory	Register-Register
Push A	Load A	Load R1, A	Load R1, A
Push B	Add B	Add R1, B	Load R2, B
Add	Store C	Store C, R1	Add R3, R1, R2
Pop C			Store C, R3

Classification of Operations

- Data Transfer – Load, store, mov
- Data Manipulation
  - Arithmetic – Add, subtract, multiply, divide
  - Signed, unsigned, integer, floating-point
  - Logical - Conjunction, disjunction, shift left, shift right
- Status manipulation
- Control Transfer
  - Conditional Branch
  - Branch on equal, not equal
  - Set on less than
  - Unconditional Jump

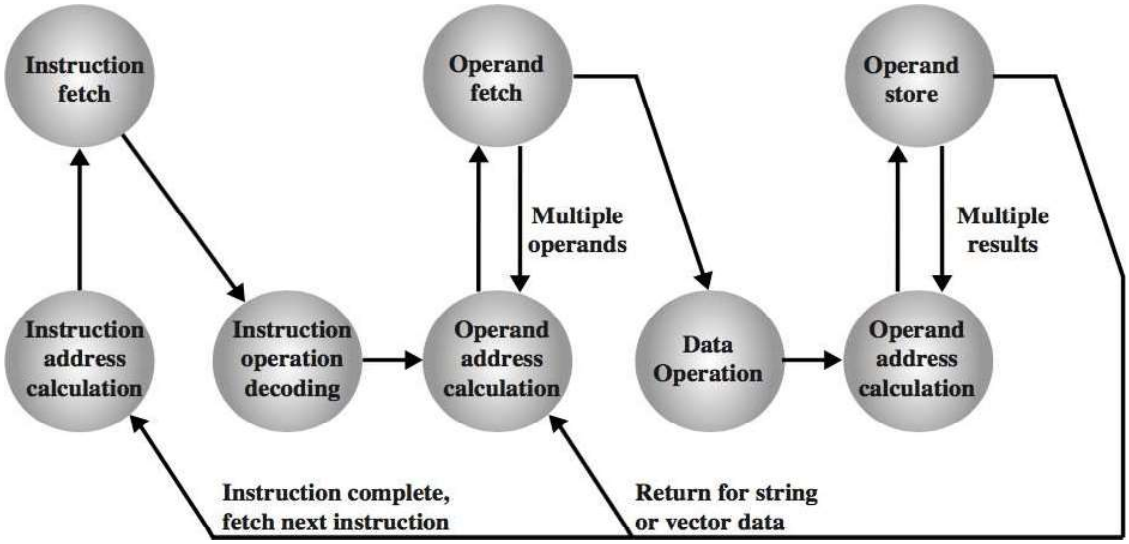
Instruction Format

- Instruction length needs to be in multiples of bytes.
- Instruction encoding can be:
  - **Variable or Fixed**
- Variable encoding tries to use as few bits to represent a program as possible:
  - **But at the cost of complexity of decoding**
- Alpha, ARM, MIPS instructions:

Operation and Modes	Addr1	Addr2	Addr3

INSTRUCTION CYCLE

An instruction cycle is the complete process of fetching, decoding and executing the instruction.



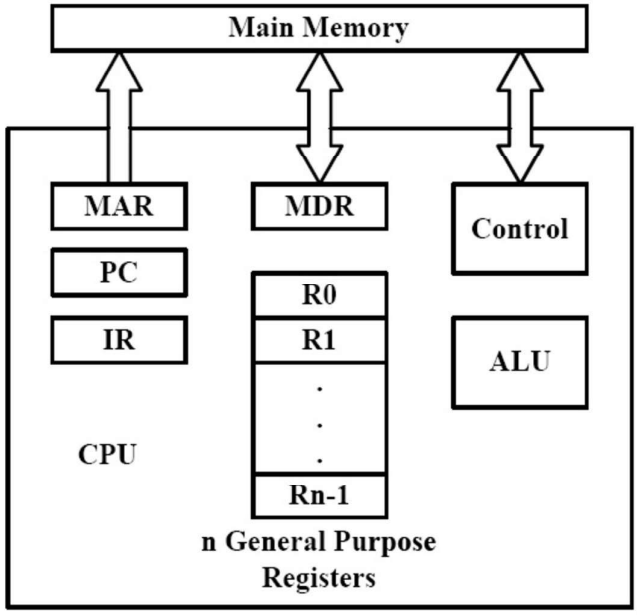
1) **PC gives the address to fetch** an instruction from the memory.
2) Once fetched, the **instruction opcode is decoded**.
3) This **identifies**, if there are **any operands to be fetched** from the memory.
4) The **operand address is calculated.**
5) **Operands are fetched** from the memory.
6) Now the **data operation is performed** on the operands, and a **result is generated**.
7) If the result has to be stored in a **register**, the **instruction ends** here.
8) If the **destination is memory**, then first the **destination address has to be calculated**.
9) The result is then **stored in the memory**.
10) Now the current instruction has **been executed**.
11) **Side by side PC is incremented** to calculate address of the next instruction.
12) The above instruction cycle then **repeats for further instructions**.

Register transfer language (RTL) Instruction

Register

- Register is a very fast computer memory, used to store data/instruction in execution
- A register is a group of flip-flops with each flip-flop capable of storing one bit of information
- An n-bit register ha a group of n flip-flops and is capable of storing binary information of n-bits
- A register is consists of a group of flip-flops and gates
- The flip-flops holds the binary information and gates controls when and how new information is transferred into a register
- The simplest register is one that consists of only one flip-flop with no external gates

- **Some of the common registers:**
  - **Accumulator**: this is the most common register, used to store data taken out from the memory
  - **General Purpose Registers**: this is used to store data intermediate results during the program execution. It can be accessed via assembly programming.
- **Some of the common Special purpose registers:** (For Computer)
  - **MAR**: Memory Address registers (holds address of the address of the memory unit)
  - **MDR**: Memory data register (stores instructions and data)
  - **PC** : Program Counter
  - **IR** **:** Instruction register (Holds instruction to be executed)



Internal connection between processor and MM

Register Transfer language (RTL)

- Symbolic notation used to describe the micro-operation transfer amongst registers is called **RTL**
- **Register Transfer** means the availability of **Hardware Logic Circuits** that can perform a stated micro-operation and transfer the result of the operation to the same or another register.
- The word **Language** is barrowed from programmers who apply this term to programming languages.
- This programming language is a procedure for writing symbols to specify a given computational process.

PIPELINING

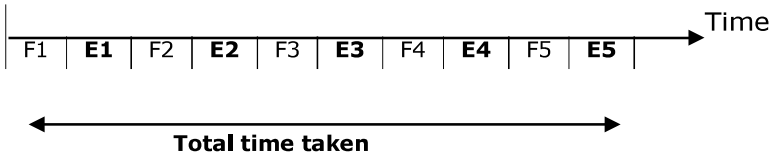
Overlapping different stages of an instruction is called pipelining.

An instruction requires several steps which mainly involve fetching, decoding and execution.
If these steps are performed one after the other, they will take a long time.
As processors became faster, several of these steps started to get overlapped, resulting in faster processing. This is done by a mechanism called pipelining.

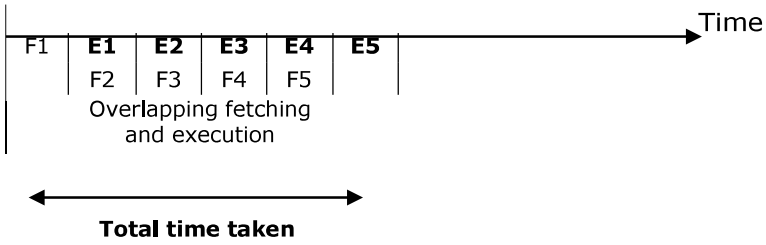
2 STAGE PIPELINING - 8086

Here the instruction process in divided into two stages of fetching and execution.
Fetching of the next instruction takes place while the current instruction is being executed. Hence two instructions are being processed at any point of time.

NON-PIPELINED PROCESSOR EG: 8085

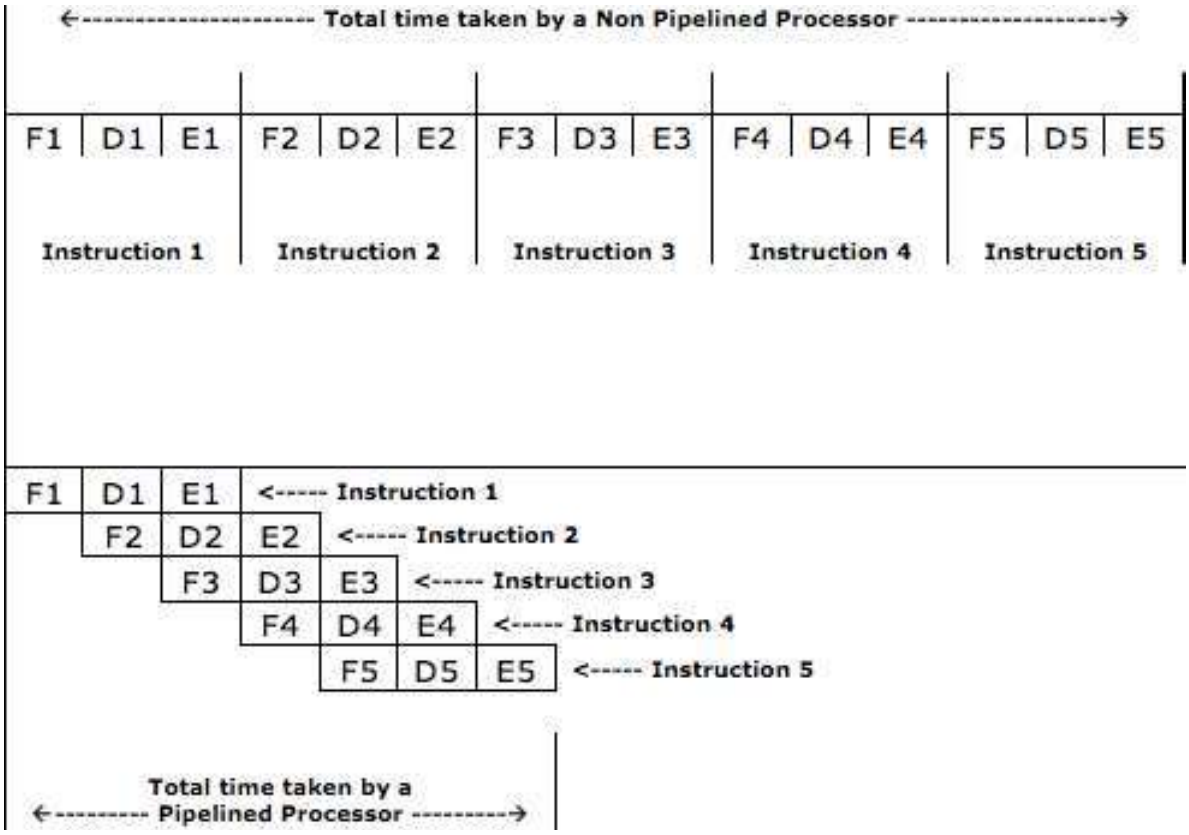


PIPELINED PROCESSOR EG: 8086



3 Stage Pipelining – 80386/ ARM 7

Here the instruction process in divided into three stages of **fetching, decoding and execution** and are overlapped. Hence **three instructions** are being processed at any point of time.



4 Stage Pipelining

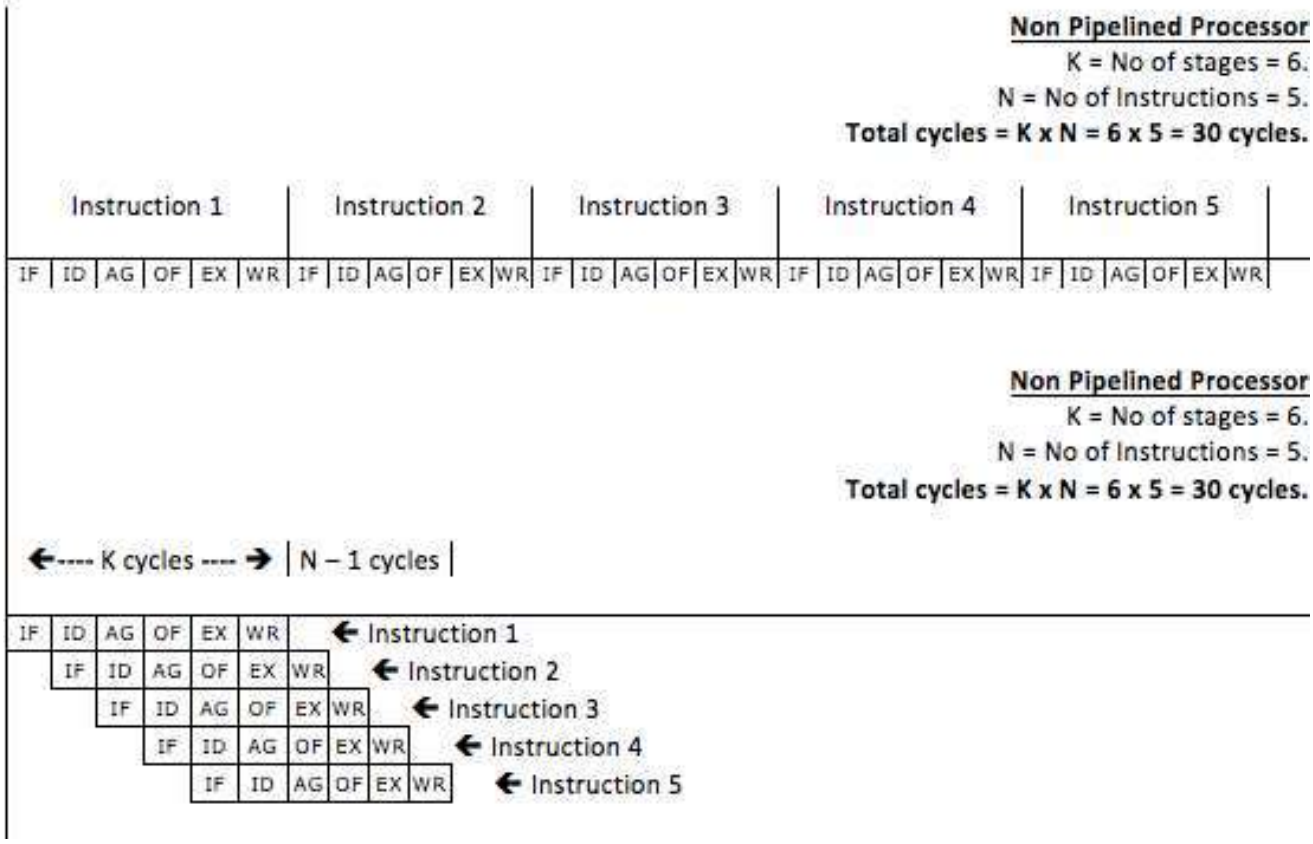
Fetch, Decode, Execute, Store

5 Stage Pipelining - Pentium

Fetch, Decode, Address Generation, Execute, Store

6 Stage Pipelining – Pentium Pro

Instruction Fetch (IF):	Fetch the instruction
Instruction Decode (ID):	Decode the instruction.
Address Generation (AG):	Calculate address of Memory operand
Operand Fetch (OF):	Fetch memory operand
Execute (EX):	Execute the operation
Write Back (WR):	Write back/ Store the result



Modern processors have a very deep pipeline.
Ex: Pentium 4 (Net burst Architecture) has a 20 stage pipeline.

Advantage of Pipelining

The obvious advantage of pipelining is that it increases the performance. As shown by the various examples above, deeper the pipelining, more is the level of parallelism, and hence the processor becomes much faster.

DRAWBACKS/ HAZARDS OF PIPELINING

There are various hazards of pipelining, which **cause a dip** in the performance of the processor. These hazards become even **more prominent** as the **number of pipeline stages increase**. They may occur due to the following reasons.

1) DATA HAZARD/ DATA DEPENDENCY HAZARD

Data Hazard is caused when **the result (destination) of one instruction becomes the operand (source) of the next instruction**.

Consider two instructions I1 and I2 (I1 being the first).

Assume I1: INC [4000H]
Assume I2: MOV BL , [4000H]

Clearly in I2, BL should get the incremented value of location [4000H].
But this can only happen once I1 has completely finished execution and also written back the result at [4000H].
In a multistage pipeline, I2 may reach execution stage before I1 has finished storing the result at location [4000H], and hence get a wrong value of data.
This is called **data dependency hazard**.
It is solved by inserting NOP (No operation) instructions between such data dependent instructions.

2) CONTROL HAZARD/ CODE HAZARD

Pipelining assumes that the program will always flow in a sequential manner.
Hence, it performs various stages of the forthcoming instructions before-hand, while the current instruction is still being executed.
While programs are sequential most of the times, it is not true always.
Sometimes, branches do occur in programs.
In such an event, all the forthcoming instructions that have been fetched/ decoded etc have to be flushed/ discarded, and the process has to start all over again, from the branch address. This causes pipeline bubbles, which simply means time of the processor is wasted.

Consider the following set of instructions:

```
Start:
        JMP Down
        INC  BL
        MOV CL, DL
        ADD AL, BL
        …
        …
        …

Down:   DEC CH
```

JMP Down is a branch instruction.
After this instruction, program should jump to the location "Down" and continue with DEC CH instruction.

But, in a multistage pipeline processor, the sequentially next instructions after JMP Down have already been fetched and decoded. These instructions will now have to be discarded and fetching will begin all over again from DEC CH. This will keep several units of the architecture idle for some time. This is called a pipeline bubble.

The **problem of branching is solved** in higher processors by a method called "**Branch Prediction Algorithm**". It was introduced by **Pentium** processor. It relies on the **previous history** of the instruction as most programs are repetitive in nature. It then **makes a prediction** whether branch will be **taken or not** and hence puts the correct instructions in the pipelines.

3) STRUCTURAL HAZARD

Structural hazards are caused by **physical constraints in the architecture like the buses**. Even in the most basic form of pipelining, we want to execute one instruction and fetch the next one. Now as long as execution only involves registers, pipelining is possible. But **if execution requires to read/ write data from the memory, then it will make use of the buses, which means fetching cannot take place at the same time**. So the fetching unit will have to wait and hence a pipeline bubble is caused. This problem is solved in complex Harvard architecture processors, which use separate memories and separate buses for programs and data. This means fetching and execution can actually happen at the same time without any interference with each other.
E.g.: PIC 18 Microcontroller.

ADDRESSING MODES OF 8086 (ADDRESSING MODES OF IA 32 ARCHITECTURE)

Addressing modes is the manner in which operands are given in an instruction.
Most processors have various addressing modes, used in their instruction set.
The addressing modes of 8086 are as follows.

I IMMEDIATE ADDRESSING MODE

In this mode the **operand** is specified in the **instruction** itself.
Instructions are **longer** but the **operands** are **easily identified**.

```
Eg:   MOV CL, 12H        ; Moves 12 immediately into CL register
      MOV BX, 1234H      ; Moves 1234 immediately into BX register
```

II REGISTER ADDRESSING MODE

In this mode **operands** are specified using **registers**.
Instructions are **shorter** but **operands cant** be **identified** by looking at the instruction.

```
Eg:   MOV CL, DL      ; Moves data of DL register into CL register
      MOV AX, BX      ; Moves data of BX register into AX register
```

III DIRECT ADDRESSING MODE

In this mode **address** of the operand is directly specified **in the instruction**.

```
Eg:   MOV CL, [2000H]     ; CL Register gets data from memory location 2000H
                          ; CL ← [2000H]
Eg:   MOV [3000H], DL     ; Memory location 3000H gets data from DL Register
                          ; [3000H] ← DL
```

IV INDIRECT ADDRESSING MODE

In Indirect Addressing modes, **address is given by a register.**
The register can be incremented in a **loop** to access a **series of locations.**
There are various sub-types of Indirect addressing mode.

REGISTER INDIRECT ADDRESSING MODE

This is the most basic form of indirect addressing mode.
Here address is simply **given by a register**.

```
Eg:   MOV CL, [BX]     ; CL gets data from a memory location pointed by BX
                       ; CL ← [BX]. If BX = 2000H, CL ← [2000H]
Eg:   MOV [BX], CL     ; CL is stored at a memory location pointed by BX
                       ; [BX] ← CL. If BX = 2000H, [2000H] ← CL.
```

REGISTER RELATIVE ADDRESSING MODE

Here address is given by a **register plus a numeric displacement**.

```
Eg:   MOV CL, [BX + 03H] ; CL gets data from a location BX + 03H
                         ; CL ← [BX+03H]. If BX = 2000H, then CL ← [2003H]
Eg:   MOV [BX + 03H], CL ; CL is stored at location BX + 03H
                         ; [BX+03H] ← CL. If BX = 2000H, then [2003H] ← CL.
```

BASE INDEXED ADDRESSING MODE

Here address is given by **a sum of two registers**.
This is typically useful in accessing an array or a look up table.
One register acts as the base of the array holding its starting address and the **other acts as an index** indicating the element to be accessed.

```
Eg:   MOV CL, [BX + SI]  ; CL gets data from a location BX + SI
                         ; CL ← [BX+SI].
                         ; If BX = 2000H, SI = 1000H, then CL ← [3000H]

Eg:   MOV [BX + SI], CL  ; CL is stored at location BX + SI
                         ; [BX+SI] ← CL.
                         ; If BX = 2000H, SI = 1000H, then [3000H] ← CL.
```

BASE RELATIVE PLUS INDEX ADDRESSING MODE

Here address is given by a sum of base register plus index register plus a numeric displacement.

```
Eg: MOV CL, [BX+SI+03H]  ; CL gets data from a location BX + SI + 03H
                         ; CL ← [BX+SI+03H].
                         ; If BX = 2000H, SI = 1000H, then CL ← [3003H]

Eg: MOV [BX+SI+03H], CL  ; CL is stored at location BX + SI + 03H
                         ; [BX+SI+03H] ← CL.
                         ; If BX = 2000H, SI = 1000H, then [3003H] ← CL.
```

V IMPLIED ADDRESSING MODE

In this addressing mode, the operand is not specified at all, as it is an implied operand. Some instructions operate only on a particular register. In such cases, specifying the register becomes unnecessary as it becomes implied.

```
Eg:   STC                ; Sets the Carry flag.
                         ; This instruction can only operate on the Carry Flag.
Eg:   CMC                ; Complements the Carry flag.
                         ; This instruction can only operate on the Carry Flag.
```

ADDRESSING MODES OF 8085

Addressing modes is the manner in which operands are given in an instruction.
Most processors have various addressing modes, used in their instruction set.
The addressing modes of 8085 are as follows.

IMMEDIATE ADDRESSING MODE

In this mode, the **Data** is specified **in** the **Instruction** itself.

Eg: **MVI A, 35H** *; Move immediately the value 35 into the Accumulator. i.e. A ← 35H*
LXI B, 4000H *; Move immediately the value 4000 into BC register pair. i.e. BC ← 4000H*

Advantage:
Programmer can easily **identify** the **operands**.

Disadvantage:
Always more than one byte hence requires **more space**.
The µP requires **two or three machine cycles** to fetch the instruction hence **slow**.

REGISTER ADDRESSING MODE

In this mode, the **Data** is specified **in Registers**.

Eg: **MOV B, C** *; Move the Contents of C-Register into B-Register. i.e. B ← C*
INR B *; Increments the contents of B-Register. i.e. B ← B + 1*

Advantage:
Instructions are of **one byte** so only one cycle is required to fetch them.

Disadvantage:
Operands **cannot** be easily **identified**.

DIRECT ADDRESSING MODE

In this mode, the **Address** of the operand is specified **in** the **Instruction** itself.

Eg: **LDA 2000H** *; Loads "A" register with Contents of Location 2000. i.e. A ← [2000]*
STA 2000H *; Stores the Contents of "A" register at the Location 2000. i.e. [2000] ← A*

Advantage:
The programmer **can identify** the address of the operand.

Disadvantage:
These are **three byte instructions** hence require three fetch cycles.

INDIRECT ADDRESSING MODE

In this mode, the **Address** of the operand is specified **in Registers.**
Hence, the instruction indirectly points to the operands.

E.g.:: **LDAX B** *; Loads "A" register with the contents of the memory Location pointed by BC Pair*
; So if BC pair = 4000 i.e. [BC] = 4000 then A ← [4000].
#Please refer Bharat Sir's Lecture Notes for this ...

STAX B *; Stores the contents of the Accumulator at the memory location pointed by BC pair.*
; So if contents of BC pair = 4000 i.e. [BC] = 4000 then [4000] ← A.
#Please refer Bharat Sir's Lecture Notes for this ...

Advantage:
Address of the operand is **not fixed** and hence can be used in a **loop**.

Disadvantage:
Requires initialization of the register pair hence more **complex**.

IMPLIED ADDRESSING MODE

In this mode, the **Operand** is **implied** in the instruction.
This instruction will work only on that implied operand, and not on any other operand.

Eg: **STC** *; Sets the Carry Flag in the Flag register, Cy ← 1.*
CMC *; Complements the Carry Flag in the Flag register.*

Advantage:
Instructions are generally **only one byte**.

Disadvantage:
Rigid, as it works only on a fixed operand.

	RISC	CISC
1	**Instructions of a fixed size**	Instructions of **variable size**
2	Most instructions take **same time to fetch.**	Instructions have **different fetching times.**
3	Instruction set **simple and small.**	Instruction set **large and complex.**
4	**Less addressing modes** as most operations are register based.	**Complex addressing modes** as most operations are memory based.
5	**Compiler design is simple**	**Compiler design is complex**
6	**Total size of program is large** as many instructions are required to perform a task as instructions are simple.	**Total size of program is small** as few instructions are required to perform a task as instructions are complex & more powerful.
7	Instructions use a **fixed number of operands.**	Instructions have **variable** number of operands.
8	Ideal for processors performing a **dedicated operation.**	Ideal for processors performing a **verity of operations.**
9	Since instructions are simple, they can be decoded by a **hardwired control unit.**	Since instructions are complex, they require a **Micro-programmed Control Unit.**
10	Execution speed is **faster** as most operations are register based.	Execution speed is **slower** as most operations are memory based.
11	As No. of cycles per instruction is fixed, it gives a **better degree of pipelining**	Since number of cycles per instruction varies, **pipelining has more bubbles** or stalls.
12	E.g.:: **ARM7**, **PIC 18** Microcontrollers.	E.g.: Intel **8085, 8086** Microprocessors.