Chapter _2

DATA REPRESENTATION

i) Introduction To Number System

Why Binary number? : Inside the computer, there are **integrated circuits with thousands of transistors**. These transistors are made to operate on a two-state. By this design, all the input and output voltages are either HIGH or LOW. Low voltage represents binary 0 and high voltage represents binary 1.

Voltage	Low	High
Binary	0	1

In Computer the data is represented in binary format (1s and 0s). Even though we use characters, decimals, punctuation marks, symbols and graph, internally these things are represented in binary format. So we will go through **the number system**. There are different number systems namely decimal, binary, octal, hexadecimal, etc.

Decimal number system is important because it is universally used to represent quantities outside a digital system. Decimal system uses numbers from 0 to 9. This is to the base 10.
Binary numbers are based on the concept of ON or OFF. Binary number system has a base of 2. Its two digits are denoted by 0 & 1 and are called bits.
Octal Number system uses exactly eight symbols 0,1,2,3,4,5,6, and 7. i.e., it has a base of 8. Each octal digit has a unique 3 bit binary representation.
Hexadecimal number system is to the base 16. It spans from 0 to 9 and then A to E. In hexadecimal system A,B,C,D,E and F represents 10,11,12,13,14 and 15, i.e., it has a base of 16. Hexadecimal numbers are more convenient for people to recognize and interpret than the long strings of binary numbers

Decimal	Binary Code	Octal	Hexadecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000		8
9	1001		9
10	1010		A
11	1011		B
12	1100		C
13	1101		D
14	1110		E
15	1111		F

ii) Floating Point Representation

In this method, a real number is expressed as a combination of a mantissa and exponent. The mantissa is kept as less than 1 and greater than or equal to 0.1 and the exponent is the power of 10.
For example, the decimal number +4567.89 is represented in floating point with a fraction and an exponent as follows:

Fraction	Exponent	
+0.456789	+04	→ scientific notation is $+0.456789 \times 10^{+4}$

$\rightarrow$ $m \times r^e$

Here, Mantissa = 0.456789 Exponent = 4

While storing numbers, the leading digit in the mantissa is always made non-zero by approximately shifting it and adjusting the value of the exponent, i.e.. .004567 $\rightarrow$ 0.4567 * 10^{-2} $\rightarrow$ 0.4567E-2

This shifting of the mantissa to the left till its most significant digits is non-zero, is called ***Normalization.***

Only the mantissa **m** and the exponent **e** are physically represented in the register (including their signs). A floating point binary number is represented in a similar manner except that it uses base 2 for the exponent.

Example: the binary number +1001.11 is represented with an 8 bit fraction and 6 bit exponent as follows:

Fraction	Exponent
01001110	000100

Here again the fraction has a ***0*** in the Leftmost position to denote ***positive.***

The exponent has the equivalent binary number +4. The floating point

$M \times 2^e = +(.1001110)_2 \times 2^{+4}$

A floating point number is said to be ***Normalized*** if the most significant digit of the mantissa is nonzero.

Example: The 8 bit binary number 00011010 is ***not normalized*** because of the three 0' s.

The number can be normalized by shifting three positions to the left and discarding the leading 0' s obtain 11010000. The three shifts multiply the number by $2^3 = 8$. To keep the same value for the floating point number, the exponent must be subtracted by 3. Normalized numbers provide the maximum possible precision for the floating point number.

Many computers and all electronic calculators have the built-in capability of performing floating-point arithmetic operations.

iii) Fixed Point Representation

Positive Integers, including zero, can be represented as unsigned numbers. However, to represent negative integers, we need a notation for negative values. In ordinary arithmetic, a negative number is indicated by a minus sign and a positive number by a plus sign.

Because of hardware limitations, computers must represent everything with 1' s and 0' s, including the sign of a number. As a consequence, it is customary to represent the sign with a bit placed in the leftmost position of the number.

There are two ways of specifying the position of the binary point in a register by giving it a fixed position or by employing a floating point representation. The fixed point method assumes that the binary pint is always fixed in one position. The two positions most widely used are

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

Representation of number as an integer:

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

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

The negative number is represented in either the 1' s or 2' s complement of its positive value.

It' s customary to use 0 for the + sign and 1 for the – sign. Therefore -001, -010 and -011 are coded as 1001, 1010 and 1011.

Note: *Complement*

Sign – magnitude numbers are easy to understand, but they require too much hardware for addition and subtraction. It has led to the widespread use of complements for binary arithmetic.

For instance, if A = 0111 $\rightarrow$ *The 1' s complement is* $\bar{A} = 1000$

The 2's complement is defined as the new word obtained by adding 1 to 1's complement. As an equation

$A' = \bar{A} + 1$ *Where* A' *= 2's complement*

$\bar{A}$ *= 1's complement*

Here are some examples. If $A = 0111$

The 1's complement is $\bar{A} = 1001$

In terms of a binary the 2's complement is the next reading after the 1's complement.

Another example, If $A = 0000\ 1000$

Then $\bar{A} = 1111\ 0111$

$1 \leftarrow$ *(adding 1)*

And $A' = 1111\ 1000$

Example: Consider the signed number 14 stored in an ***8 bit*** register.

+14 is represented by a sign bit of 0 in the leftmost position followed by the binary equivalent of 14 $\rightarrow$ 00001110. Note that each of the eight bits of the register must have a value and, therefore, 0' s must be inserted in the most significant positions following the sign bit.

Although there is only one way to represent +14, there are three different ways to represent -14 with eight bits.

	Sign bit	
In signed magnitude representation	1	0001110
In signed 1' s complement representation	1	1110001
In signed 2' s complement representation	1	1110010

$\rightarrow$The signed magnitude representation of -14 is obtained from +14 by complementing only the sign bit.

$\rightarrow$The signed 1' s complement representation of -14 is obtained by complementing all the bits of +14, including the sign bit.

$\rightarrow$The signed 2' s complement representation is obtained by taking the 2' s complement of the positive number, including its sign bit.

Arithmetic Addition:

The addition of two numbers in the signed-magnitude system follows the rules of ordinary arithmetic. If the signs are same, we add the two magnitudes and give the sum the common sign. If the sign are different, we subtract the smaller magnitude from the larger and give the result the sign of the larger magnitude.

Add the two numbers, including their sign bits, and discard any carry out of the sign bit position. Numerical examples for addition are shown below. Note that negative numbers must initially be in 2' s complement and that if the sum obtained after the addition is negative, it is in 2' s complement form.

+6	00000110	-6	11111010	+6	00000110	-6	11111010
+13	00001101	+13	00001101	-13	11110011	-13	11110011
+19	00010011	+7	00000111	-7	11111001	-19	11101101

In each of the four cases, the operation performed is always addition, including the sign bits. Any carry out of the sign bit position is ***discarded***, and negative results are automatically in 2' s complement form.

Arithmetic subtraction:

Subtraction of two signed binary numbers when negative numbers are in 2' s complement form is very simple.

The complement of a negative number in complement form produces the equivalent positive number.
Ex: (-6) – (-13) = +7
In binary with 8 bits this is written as 11111010 – 11110011.
The subtraction is changed to addition by taking the 2' s complement of (-13) to give (+13).
Ie., 11110011 → 2' s complement is → 0000 1101
In binary this is 11111010 + 00001101 = 100000111.
Removing the end carry, we obtain the correct answer 00000111 (+7)

It is worth noting that binary numbers in the signed 2' s complement system are added and subtracted by the same basic addition and subtraction rules as unsigned numbers.
Therefore, computers need only one common hardware circuit to handle both types of arithmetic.

Overflow:

When two numbers of n digits each are added and the sum occupies n+1 digits, we say that an overflow occurred.
An overflow is a problem in digital computers because the width of registers is finite.

A result that contains n+1 bits cannot be accommodated in a register with a standard length of n bits. For this reason, many computers detect the occurrence of an overflow, and when it occurs, a corresponding flip-flop is set which can then be checked by the user.

An overflow may occur if the two numbers added are both positive or both negative.

Example: Two signed binary numbers, +70 and +80, are stored in two 8bit registers.

```
carries :   0     1                      carries:      1     0
+70               0     1000110                -70           1     0111010
+80               0     1010000                -80           1     0110000
                  -------------                              -------------
+150              1     0010110                -150          0     1101010
```

Note that the 8 bit result that should have been positive has a negative sign bit and the 8bit result that should have been negative has a positive sign bit. If, however, the carry out of the sign bit position is taken as the sign bit of the result, the 9 bit answer so obtained will be correct.
Since the answer cannot be accommodated within 8 bits, we say that an overflow occurred. If the two carries are applied to an exclusive-OR gate, an overflow will be detected when the output of the gate is equal to 1.

iv) Representation Of Numbers, Operand Of Code And Address

Other binary codes for decimal numbers and alphanumeric characters are sometimes used.
Gray code:: This is an unweighted code. It means that there are no specific weights assigned to the bit position. This code is not suitable for arithmetic operations, but it is very useful for input-output devices, Analog to digital conversion etc., and the machines which use Shaft encoders as sensors.
The following table gives the comparison of decimal, binary and gray codes. 4bit Gray code

Decimal	Binary Code	Gray Code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111

6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101

Note: *Left most bit is the first binary digit as the first gray code. Then move from left to right and add each adjacent pair of binary digit to get the next gray code digit,* ***neglect carries if any****.*

Ex: 1010 = ?

(1) (2) (3) (4)

1 0 1 0

↓

1

Add (1) + (2) → 1 + 0 = 1 and Add (2) + (3) → 0 +1 = 1 and Add (3) + (4) → 1 + 0 = 1 then result is 1111 → gray code

BCD(Binary – Coded decimal)

it is based on the idea of converting each decimal digits into its equivalent binary number. It means each decimal is represented b binary code of 4 bits. The devices such as Electronic calculators, Digital voltmeters, frequency counters, electronic counters, digital clocks etc, work with BCD numbers. BCD codes have also been used in early computers. Modern computers do not use BCD numbers as they have to process names and other non numeric data.

Decimal	**BCD**
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	0001 0000
11	0001 0001
12	0001 0010
….	……
77	0111 0111

Hexadecimal Versus BCD

The Hexadecimal system utilizes the full capacity of four binary bits, whereas BCD codes do not utilize the same. The BCD codes do not utilize the binary codes from 1010 to 1111. In the hexadecimal system an 8 bit word can represent up to FF, that is 11111111 (255 decimal) whereas in BCD only up to 10011001 (99 decimal). Hence the hexadecimal is a compact form of representation, and it occupies less memory space, thereby reducing the hardware cost. The arithmetic operations are also simpler in hexadecimal system.

Alpha numeric Code: ASCII (American standard code for information interchange)

It is the standardized alphanumeric code and most widely used by several computer manufacturers as their computer' s internal code. It is a 7 bit alphanumeric code which has 2^7 =128 different characters to encode 85 characters. (52 lowercse and uppercase alphabets, 10 numerals and 23 punctuation and others symbols as marked on keyboards printers,video display etc.,) ASCII code can be classified into 2 types as specified below.

(i) ASCII -7 : 7 bit code, which represents 2^7 =128 different characters. In 7 bits, the first 3 bit represent zone bit and next 4 bit represent digit.

(ii) ASCII-8 : 8 bit code, which represent 2^8 = 256 different characters. In 8 bits, the first 4 bit represent zone bit and next 4 bit represent digit.

ASCII 7 code for characters

LSB	**MSB $b_6b_5b_4$**					
$b_3b_2b_1b_0$	**010**	**011**	**100**	**101**	**110**	**111**
0000	SPACE	0	@	P	-	P
0001	!	1	A	Q	A	Q
0010	“	2	B	R	B	R
0011	#	3	C	S	C	S
0100	$	4	D	T	D	T
0101	%	5	E	U	E	U
0110	&	6	F	V	F	V
0111	‘	7	G	W	G	W
1000	(	8	H	X	H	X
1001	)	9	I	Y	I	Y
1010	*	:	J	Z	J	Z
1011	+	;	K	[	K	{
1100	‘	<	L	\	L	|
1101	-	=	M	]	M	}
1110	.	>	N	^	N	~
1111	/	?	O	_	O	Del

For instance, the letter ‘ A’ has the bits $b_6b_5b_4$ of 100 and the bits $b_3b_2b_1b_0$ of 0001. Therefore, A is represented in ASCII code as 100 0001 → 41 (hexadecimal)

Character	**ASCII -7 Code**		**Hexadecimal Equivalent**
	Zone	Digit	
0	011	0000	30
1	011	0001	31
2	011	0010	32
…	…	…	..
A	100	0001	41
B	100	0010	42
C	100	0011	43
…	…	…	…

The 8 bit ASCII-8 codes…..

Character	**ASCII -8 Code**		**Hexadecimal Equivalent**
	Zone	Digit	

0	0101	0000	50
1	0101	0001	51
2	0101	0010	52
…	…	…	..
A	1010	0001	A1
B	1010	0010	A2
C	1010	0011	A3
…	…	…	…

v) <u>Instruction Formats And Types</u>

A computer will usually have a variety of instruction code formats. It is the function of the control unit within the CPU to interpret each instruction code and provide the necessary control functions needed to process the instruction.

The bits of the instruction are divided into groups called fields. The most common fields found in instruction formats are:

1. An operation code field that specifies the operation to be performed
2. An address field that designates a memory address or a processor register
3. A mode field that specifies the way the operand or the Effective Address is determined.

The operation code field of an instruction is a group of bits that define various processor operations, such as add, subtract, complement, and shift.
The bits that define the mode field of an instruction code specify a variety of alternatives for choosing the operands from the given address.

- Operations specified by computer instructions are executed on some data stored in memory or processor registers.
- Operands residing in memory are specified by their ***memory address***.
- Operands residing in processor registers are specified with a ***register address***.

A register address is a binary number of k bits that defines one of 2^k registers in the CPU. Thus a CPU with 16 processor registers R0 through R15 will have a register address field of 4 bits.

Most computers fall into one of 3 types of CPU organizations:

1. Single accumulator organization
2. General register organization
3. Stack organization

All operations are performed with an implied accumulator register. The instruction format in this type of computer uses one address field.

- ➔ For example, the instruction that specifies an arithmetic addition is defined by an assembly language instruction as

  ADD X

Where X is the address of the operand. The ADD instruction in this case results in the operation AC $\leftarrow$AC + M[X] .
AC is the accumulator register and M[X] symbolizes the memory word located at address X

- ➔ The instruction format in this type of computer needs three register address fields.

  ADD R1, R2, R3

to denote the operatin R1 $\leftarrow$ R2 + R3.

- ➔ The number of address fields in the instruction can be reduced from 3 to 2 if the destination register is the same as one of the source registers.

  ADD R1, R2

would denote the operation R1 ← R1 + R2. Only register addresses for R1 and R2 need be specified in this instruction.

- ➔ Computers with multiple processor registers use the move instruction with a mnemonic MOV to symbolize a transfer instruction.
  MOV R1 , R2 denotes the transfer R1 ← R2
- ➔ General register type computers employ two or three address feidls.
  ADD R1, X

Would specify the operation R1 ← R1 + M[X]. It has two address fields, one for register R1 and the other for the memory address X.

- ➔ Computers with stack organization would have PUSH and POP instructions which require an address field.
  PUSH X

will push the word at address X to the top of the stack. The stack pointer is updated automatically.

- ➔ Operation type instructions do not need an address field in stack organized computers. This is because the operation is performed on the two items that are on top of the stack.

Ie., **ADD**

in a stack computer consists of an operation code only with no address field. This operation has the effect of popping the two top numbers from the stack, adding the numbers, and pushing the sum into the stack. There is no need to specify with an address field since all operands are implied to be in the stack.

Some computers combine features from more than one organizational structure.
For ***Example***, the intel 8080 microprocessor has 7 CPU registers, one of which is an accumulator register. The processor has some of the characteristics of a general register type and some of the characteristics of an accumulator type. Moreover, the intel 8080 processor has a stack pointer and instructions to push and pop from a memory stack.

- ❖ *To illustrate the influence of the number of addresses on computer programs, we will evaluate the arithmetic statement*
  X = (A + B) * (C + D)

We will use the symbols ADD, SUB, MUL and DIV for the 4 arithmetic operations.
MOV for the transfer type operation
LOAD and STORE for transfers to and from memory and AC register.
We will assume that the operands are in memory addresses A,B,C, and D, and the result must be stored in memory at address X

Three Address Instructions:

ADD	R1, A, B	ie., R1 ← M[A] + M[B]
ADD	R2, C, D	i.e., R2 ← M[C] + M[D]
MUL	X, R1, R2	i.e., M[X] ← R1 * R2

It is assumed that the computer has two processor registers, R1, and R2. The symbol M[A] denotes the operand at memory address symbolized by A.

Two Address Instructions:

MOV	R1, A	ie., R1 ← M[A]
ADD	R1, B	i.e., R1 ← R1 + M[B]
MOV	R2, C	ie., R2 ← M[C]
ADD	R2, D	i.e., R2 ← R2 + M[D]
MUL	R1, R2	i.e., R1 ← R1 * R2
MOV	X, R1	i.e., M[X] ← R1

One Address Instructions:

LOAD	A	ie., AC ← M[A]
ADD	B	i.e., AC ← AC + M[B]
STORE	T	i.e., M[T] ← AC
LOAD	C	ie., AC ← M[C]

ADD	D	i.e., AC ← AC + M[D]
MUL	T	i.e., AC ← AC * M[T]
STORE	X	i.e., M[X] ← AC

All operations are done between the AC register and a memory operand. T is the address of a temporary memory location required for storing the intermediate result.

Zero Address Instructions:

A stock organized computer does not use an address field for the instructions ADD and MUL. The PUSH and POP instructions, however, need an address field to specify the operand that communicates with the stack.
(*TOS stands for top of stack*)

PUSH	A	i.e., TOS ← A
PUSH	B	i.e., TOS ← B
ADD		i.e., TOS ← (A + B)
PUSH	C	i.e., TOS ← C
PUCH	D	i.e., TOS ← D
ADD		i.e., TOS ← (C + D)
MUL		i.e., TOS ← (C + D) * (A + B)
POP	X	i.e., M[X] ← TOS

vi) Addressing Modes

An operand reference in an instruction either contains the actual value of the operand or a reference to the address of the operand. Once we have determined the number of addresses contained in an instruction, the manner in which each address field specifies memory location must be determined

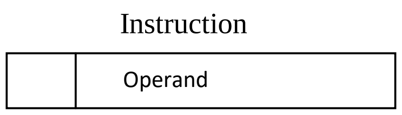
Want the ability to reference a large range of address locations
The most common addressing techniques:

1. Immediate
2. Direct
3. Indirect
4. Register
5. Register indirect
6. Displacement
7. Stack

We Use The Following Notation

A = contents of an address field in the instruction
R = contents of an address field in the instruction that refers to a register
EA = actual (effective) address of the location containing the referenced operand
(X) = contents of memory location X or register X

1. Immediate Mode

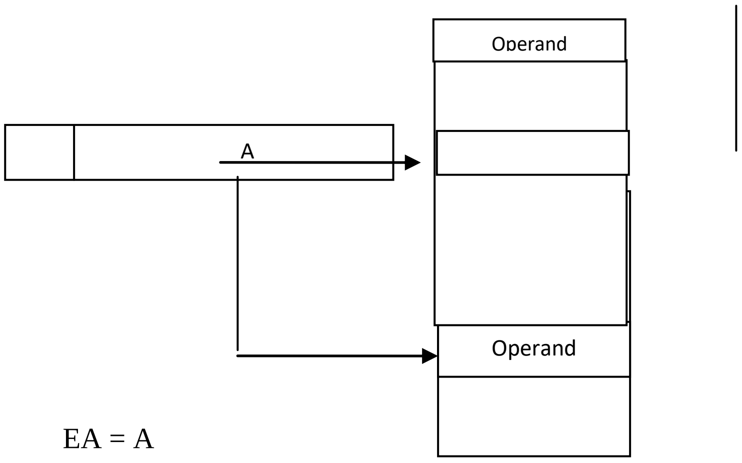


Algorithm : operand = A

– The operand is contained within the instruction itself
– Data is a constant at run time

Advantage : No additional memory references are required after the fetch of the instruction itself
Disadvantage : Size of the operand (thus its range of values) is limited

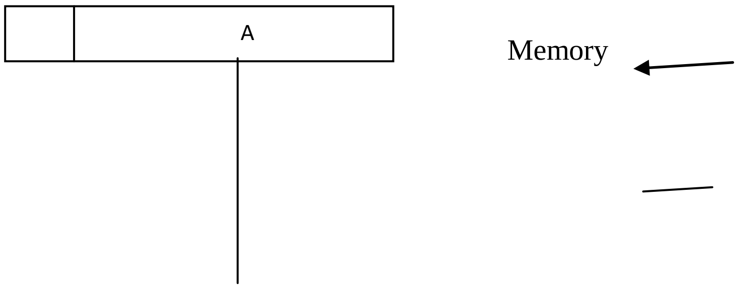
2. Direct mode



Algorithm: EA = A

– The address field of the instruction contains the effective address of the operand
– No calculations are required
– Address is a constant at run time but data itself can be changed during program execution
Advantage : One additional memory access is required to fetch the operand
Disadvantage :Address range limited by the width of the field that contains the address reference

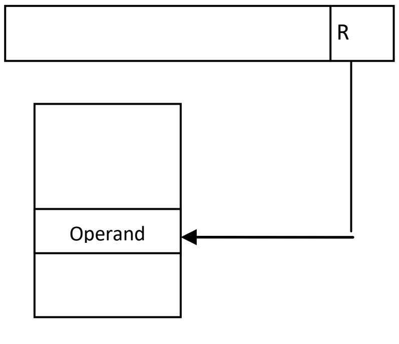
3. Indirect addressing



Algorithm: EA = (A)

– The address field in the instruction specifies a memory location which contains the address of the data
– Two memory accesses are required
» The first to fetch the effective address
» The second to fetch the operand itself
–Advantage: Range of effective addresses is equal to 2^n, where n is the width of the memory data word i.e, large address space.
-Disadvantage : instruction execution requires two memory references to fetch the operand

4. Register-based addressing modes



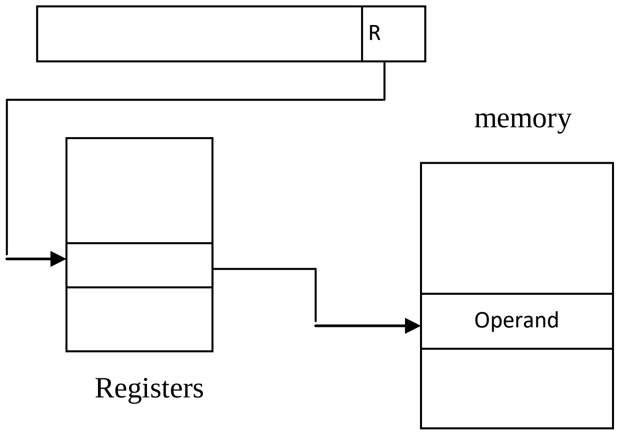
Registers

Algorithm: EA = R

– Register addressing: like direct, but address field specifies a register location

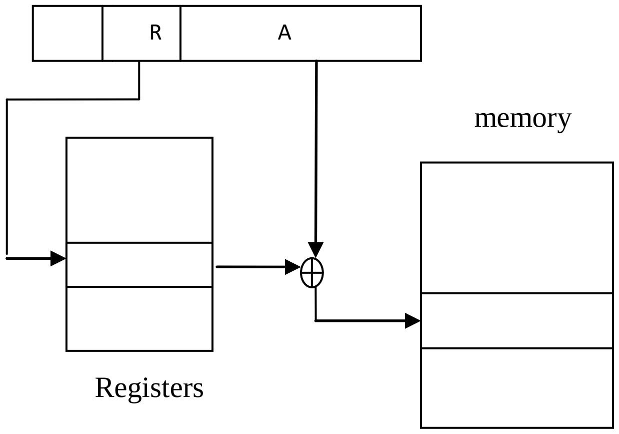
– No memory reference
– Faster access to data, smaller address fields in the instruction word

5. Register-Indirect addressing modes



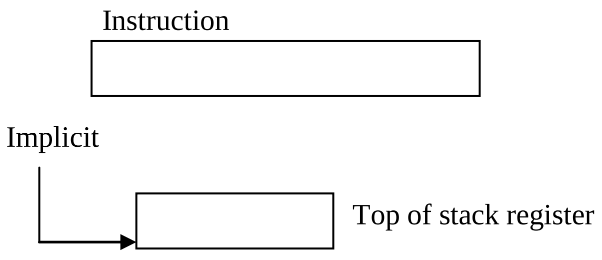
Algorithm: EA = (R)
– Register indirect: like indirect, but address field specifies a register that contains the effective address
-advantages : Large address space
-disadvantage : extra memory reference

6. Displacement or address relative addressing



Algorithm: EA = A + (R)
– Two address fields in the instruction are used
 » One is an explicit address reference
 » The other is a register reference
 » EA = A + (R)
– *Relative addressing:*
 » A is added to the program counter contents to cause a branch operation in fetching the next instruction
– *Base-register addressing:*
 » A is a displacement added to the contents of the referenced " base register" to form the EA
 » Used by programmers and O/S to identify the start of user areas, segments, etc. and provide accesses within them
-Indexing:
 » Indexing is used within programs for accessing data structures

7. STACK:



Algorithm: EA = top of stack

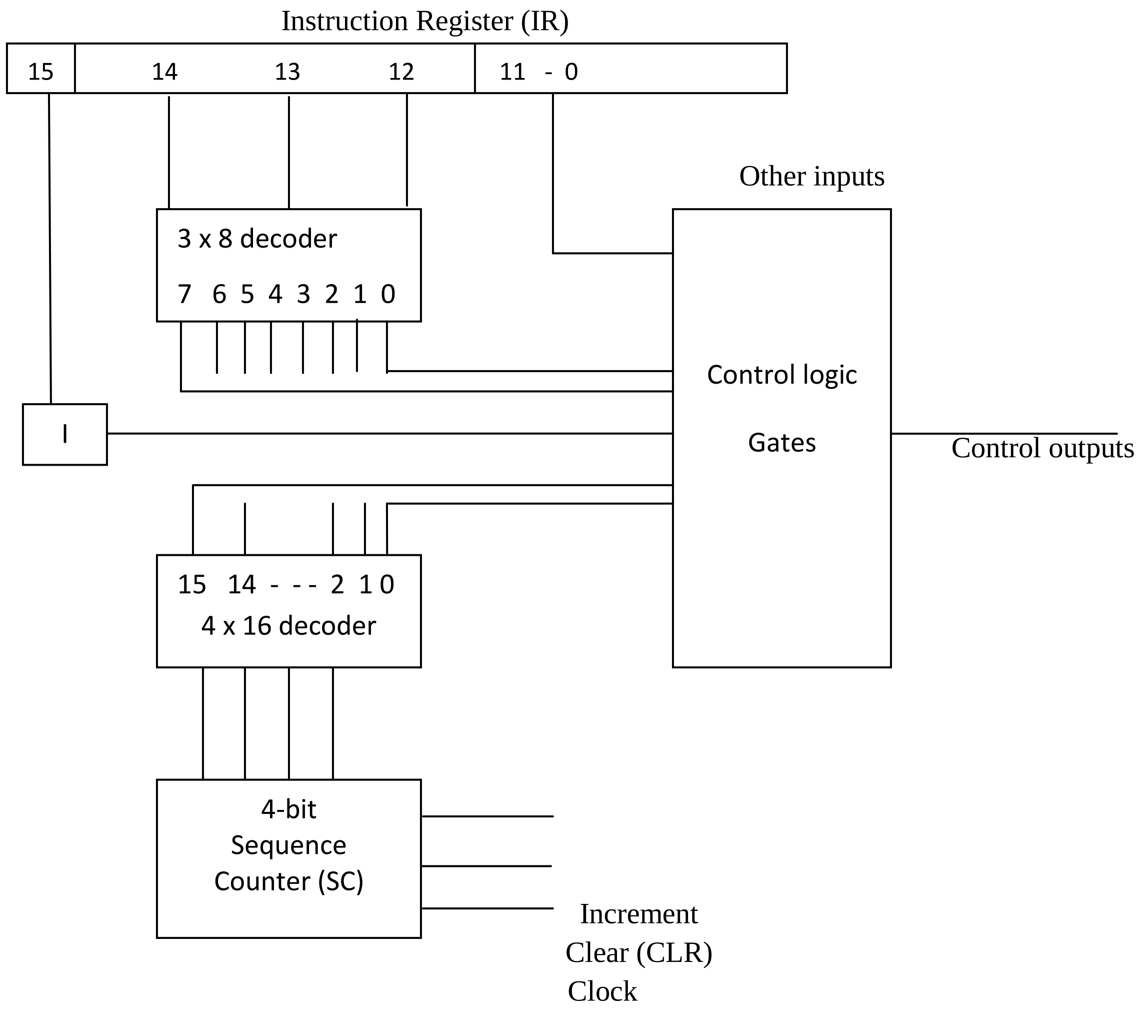
A stack is a linear array of locations. Items are appended to the top of the stack so that, at any given time, the block is partially filled. Associated with the stack is a pointer whose value is the address of the top of the stack. The top two elements of the stack may be in processor registers, in which case the stack pointer references the third element of the stack. The stack pointer is maintained in a register.

The stack mode of addressing is a form of implied addressing. The machine instructions need not include a memory reference but implicitly operate on the top of the stack.

Advantage : No memory reference

Disadvantage : Limited applicability

vii) Timing cycle



The timing for all registers in the basic computer is controlled by a master clock generator. The clock pulses are applied to all flip-flops and registers in the system. The clock pulses do not change the state of a register unless the register is enabled by a control signal.

The control signals are generated in the control unit and provide control inputs for the multiplexers in the common bus, control inputs in processor registers and micro-operations for the accumulator.

There are two major types of control organization: hardwired control and micro-programmed

Hardwired organization : the control logic is implemented with gates, flip-flops, decoders and other digital circuits.

Microprogrammed organization: the control information is stored in a control memory. The control memory is programmed to initiate the required sequence of micro-operations.

In the block diagram of the control unit , it consists of two decoders, a sequence counter and a number of control logic gates.

An instruction read from memory is placed in the instruction register (IR). IR is divided into 3 parts; the 1 bit, the operation code and bits 0 through 11. The operation code in bits 12 through 14 are decoded with a 3 x 8 decoder. The eight outputs of the decoder are designated by the symbols D0 through D7. Bit 15 of the instruction is transferred to a flip-flop designated by the symbol I.

Bits 0 through 11 are applied to the control logic gates. The 4 bit sequence counter can count in binary from 0 through 15.

The outputs of the counter are decoded into 16 timing signals T0 through T15. The sequence counter SC can be incremented or cleared synchronously. Most of the time, the counter is incremented to provide the sequence of timing signals out of the 4 x 16 decoder.

Once in a while, the counter is cleared to 0, causing the next active timing signal to be T0. As an example consider the case where SC is incremented to provide timing signals T0,T1,T2,T3 and T4 in sequence. At time T4, SC is cleared to 0 if decoder output D3 is active. This is expressed symbolically by the statement

D3T4 : SC $\leftarrow$ 0

viii) <u>Conversion Of Number System</u>

The right ost bit is called Least Significant Bit (LSB) and left Most Bit is called Most Significant Bit (MSB).

Ie.,

$$\begin{tikzcd} & 100111 \arrow[dl] \arrow[dr] & \\ \text{MSB} & & \text{LSB} \end{tikzcd}$$

<u>Conversion from Binary to Decimal</u>

The binary number system have a base 2, the position weights are used on the power of 2.

Ex: 100111 = ?

$1 \times 2^5 + 0\text{x } 2^4 + 0\text{x } 2^3 + 1\text{x}2^2 + 1\text{x}2^1 + 1\text{x}2^0$

1 x 32 + 0x 16 + 0x 8 + 1x4 + 1x2 + 1x1 = 39

100111 = 39 $\rightarrow$ Ans.

Ex- 2: 0.1010 = ?

$1 \times 2^{-1} + 0\text{x } 2^{-2} + 1\text{x } 2^{-3} + 0\text{x}2^{-4}$

1x(1/2) + 0x(1/4) + 1x (1/8) + 0 x (1/16)

1x0.5 + 0x0.25 + 1x 0.125 + 0 x 0.0625

(8 + 4 + 0 + 1) + (0.5 + 0 + 0. 125 + 0) = 13.625

0.1010 = 0.625 $\rightarrow$ Ans.

Ex-3: 1101.1010 = ?

$(1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0) + (1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4})$

$(1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1) + (1 \times 0.5 + 0 \times 0.25 + 1 \times 0.125 + 0 \times 0.0625)$

$(8 + 4 + 0 + 1) + (0.5 + 0 + 0.125 + 0) = 13.625$

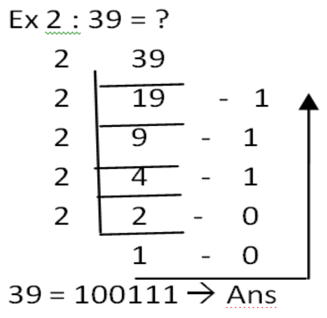
1101.1010 = 13.625 → Ans.

<u>Conversion from Decimal to Binary</u>

Ex: 15 = ?

```
  2 | 15
  2 | 7   - 1
  2 | 3   - 1
      1   - 1
```

15 = 1111 → Ans



Ex 2 : 39 = ?

```
2   39
2 | 19  - 1
2 | 9   - 1
2 | 4   - 1
2 | 2   - 0
    1   - 0
```

39 = 100111 → Ans

Ex 2: 0.625 = ?

0.625 x 2	=	1.25	→ 0.25	with a carry of 1 (MSB)
0.25 x 2	=	0.50	→ 0.50	with a carry of 0
0.50 x 2	=	1.00	→ 0.00	with a carry of 1 (LSB)

0.625 = 0.101

Ex 2: 0.39 = ?

0.39 x 2	=	0.78	→ 0.78	with a carry of 0
0.78 x 2	=	1.56	→ 0.56	with a carry of 1
0.56 x 2	=	1.12	→ 0.12	with a carry of 1

0.39 = 0.011

Note: in this example it is seen that the fraction has not become zero, and the will continue further. For such a case an approximation is made. For this example, we may take the result up to 3 binary bits after the binary point.

<u>Conversion from Decimal to Octal:</u>

Ex: 3977 = ?

```
  8 | 3977
  8 | 497    -       1
  8 | 62     - 1
      7    - 6
```

3977 = 7611 → Ans

Ex 2: 0.39 = ?

0.39 x 8	=	3.12	→ 0.12	with a carry of 3
0.12 x8	=	0.96	→ 0.96	with a carry of 0
0.96 x 8	=	7.68	→ 0.68	with a carry of 7
0.68 x 8	=	5.44	→ 0.44	with a carry of 5

0.39 = 0.3075

<u>Conversion from Octal to Decimal:</u>

The binary number system has a base 8, the position weights are used on the power of 8.

Ex: 3077 = ?

$3 \times 8^3 + 0 \times 8^2 + 7 \times 8^1 + 7 \times 8^0$

$3 \times 512 + 0 \times 64 + 7 \times 8 + 7 \times 1 = 1599$

3077 = 1599 → Ans.

<u>Conversion from Binary to Octal :</u>

Ex: 10011.11001 = ?

➔ 010 011 . 110 010

➔ 2 3 6 2

= 23.62

<u>Conversion from Octal to Binary :</u>

Ex1: 37

= 011 111

=011111

Ex2: 377.77

011 111 111 . 111 111

=011111111.111111

<u>Conversion from Decimal to Hexadecimal:</u>

Ex: 39 = ?

```
16 | 39
   ----
     2   -  7
```

39 = 27 → Ans

Ex: 256 = ?

```
16 | 256
16 | 16    - 0
   ----
     1    -  0
```

256 = 100 → Ans

Ex: 1723.256 = ?

```
16 | 1723
16 | 107     -     11
   ----
     6       -  11
```

0.256 x 16	=	4.096	→ 0.096	with a carry of 4
0.096 x16	=	1.536	→ 0.536	with a carry of 1
0.536 x 16	=	8.576	→ 0.576	with a carry of 8
0.576 x 16	=	9.216	→ 0.216	with a carry of 9

1723.256 = 6BB.4189 → Ans

<u>Conversion from Hexadecimal to Decimal:</u>

The binary number system has a base 16, the position weights are used on the power of 16.

Ex: 39A = ?

$3 \times 16^2 + 9 \times 16^1 + A \times 16^0$

$3 \times 256 + 9 \times 16 + A \times 1$

$3 \times 256 + 9 \times 16 + 10 \times 1 = 922$ (i.e., decimal 10 for A Hex)

39A = 922 → Ans.

Ex: 3A.2F = ?

$3x\ 16^1 + Ax\ 16^0 . 2 X 16^{-1} + 15 X 16^{-2}$

=48+10 . (2/10) + (15 / 16^2)

= 58.1836

Conversion from Binary to Hexadecimal :

Ex: 11100101110111

=0011 1001 0111 0111

=3 9 7 7

=3977

Ex: 110101001.0010110

=0001 1010 1001 . 0010 1100

=1 A 9 . 2 C

Conversion from Hexadecimal to Binary :

Ex: 39A

=0011 1001 1010

Octal → Binary → Hexadecimal:

7521 → Octal

111 101 010 001 → Binary

1111 0101 0001 → Hexadecimal

Hexadecimal → Octal:

CBAED → Hexadecimal

C B A E D

1100 1011 1010 1110 1101

011 001 011 101 011 101 101

3 1 3 5 3 5 5

CBAED 3135355

Binary to Gray Code:

Ex: 1010 = ?

(1)	*(2)*	*(3)*	*(4)*
1	*0*	*1*	*0*

$\downarrow$

1

Add (1) + (2) → 1 + 0 = 1

Add (2) + (3) → 0 +1 = 1

Add (3) + (4) → 1 + 0 = 1 then result is 1111 → gray code

Gray Code to binary:

Ex: 1111 = ?

(1)	*(2)*	*(3)*	*(4)*
1	*1*	*1*	*1*

$\downarrow$

1 $\rightarrow$ (5)

Add (5) + (2) $\rightarrow$ 1 +1 =0 $\rightarrow$ (6) note: discard carry 1

Add (6) + (3) $\rightarrow$ 0 +1 = 1 $\rightarrow$ (7)

Add (7) + (4) $\rightarrow$ 1 +1 = 0 note: discard carry 1

then result is 1010 $\rightarrow$ Binary code