1. Understand the problem
2. Determine the number of input and output variables that are needed
3. Give symbols for the stated input and output
4. Construct a truth table that defines the relationship between the input and output
5. Obtain the Boolean function or the logical expression from the truth table in (2) using Karnaugh Map or other known methods.
6. Draw a logic circuit based on the expression obtained from (5) above.

Below are examples of designing combinational circuits that are in the computer system that is the adder. Because computers use binary system for its data, its adder is based on the addition of the binary system. There are 2 kinds of addition, which are identified to be half addition and full addition.

Half addition is the addition of 2 bits data (doesn't involve carry) that produces 2 bits output, that is the result and the carrier. Full addition is the addition of 3 bits data (2 bits data and 1 bit carry) that produces 2 bits output (sum and carry). Logic circuit for half addition is known as Half Adder while the logic circuit for full addition is known as Full Adder

Designing a Circuit for Half Adder

The steps are as below:

Problem: to build a logic circuit for the addition of 2 bits data

1. Number of input : 2 Number of output : 2
2. Variables for input: x and y Variables for output : s (sum) and c (carry)
3. The Truth Table for the problem :

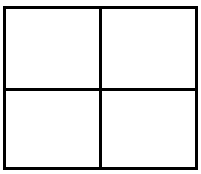
INPUT		1. OUTPUT	
	y	S	c
	0	0	0
	1	1	0
	0	1	0
	1	0	1

4. The expressions for r and c using Karnaugh Map

For

s

x



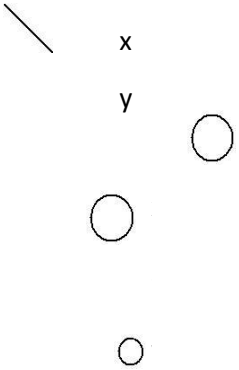
$\bar{s} = x$

y

+

x

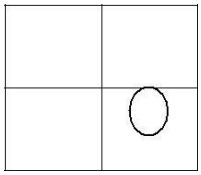
y



For c

x y

1



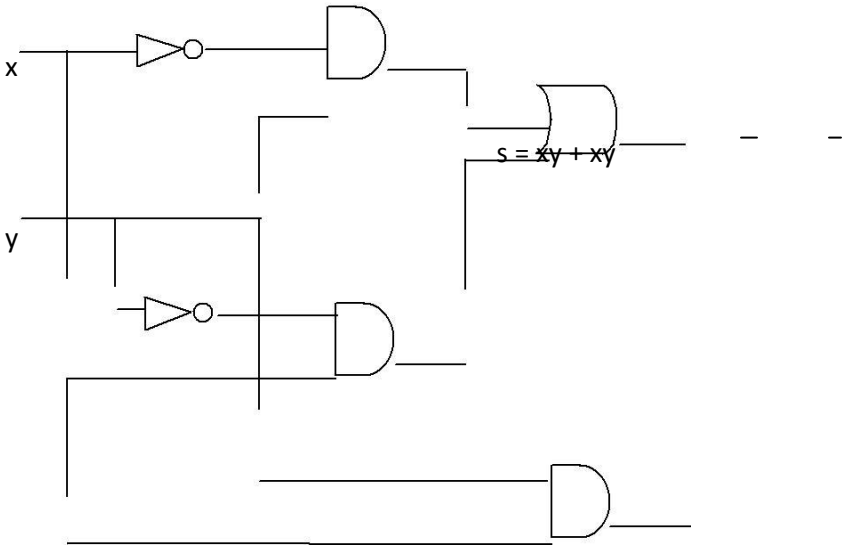
c

=

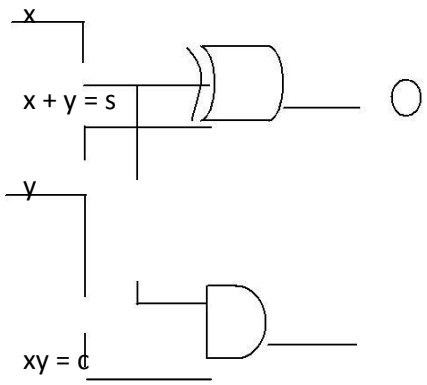
x

y

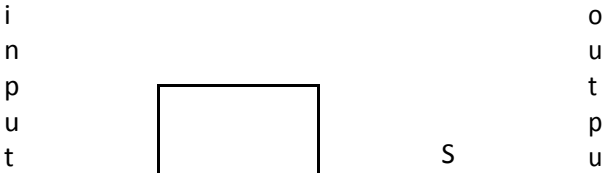
5. A logic circuits for Half Adder (HA)

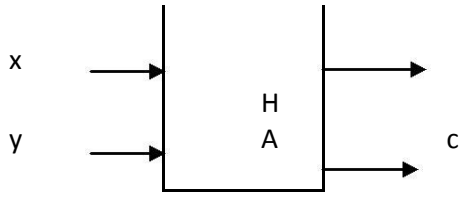


OR



A Block Diagram for HA is as below:





Designing a Circuit for Full Adder (FA)

The same method used to design HA.

1. Problem: Build logic circuit for the addition of 3 bit data
2. Number of input : 3 Number of output : 2
3. Variables for input: x , y and c_i
   Variables for output : s (sum) and c_o (carry)
4. The truth table for the problem :

INPUT			OUTPUT	
X	**y**	**c_i**	**S**	
0	0	0	0	
0	0	1	1	
0	1	0	1	
0	1	1	0	
1	0	0	1	
1	0	1	0	
1	1	0	0	
1	1	1	1	

5. Obtain the expression for r and c_o using Karnaugh Map **(Students are required to try this out themselves)**:

w
i
l
l

o
b

s =
x y
p_i +
x y
c_i +

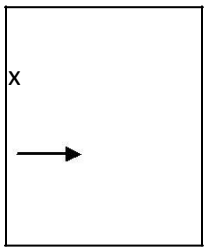
t a i n x y

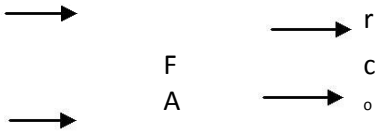
a n d $= x + y + c_i$

$c_o = x\,y + y\,c_i + x\,c_i$

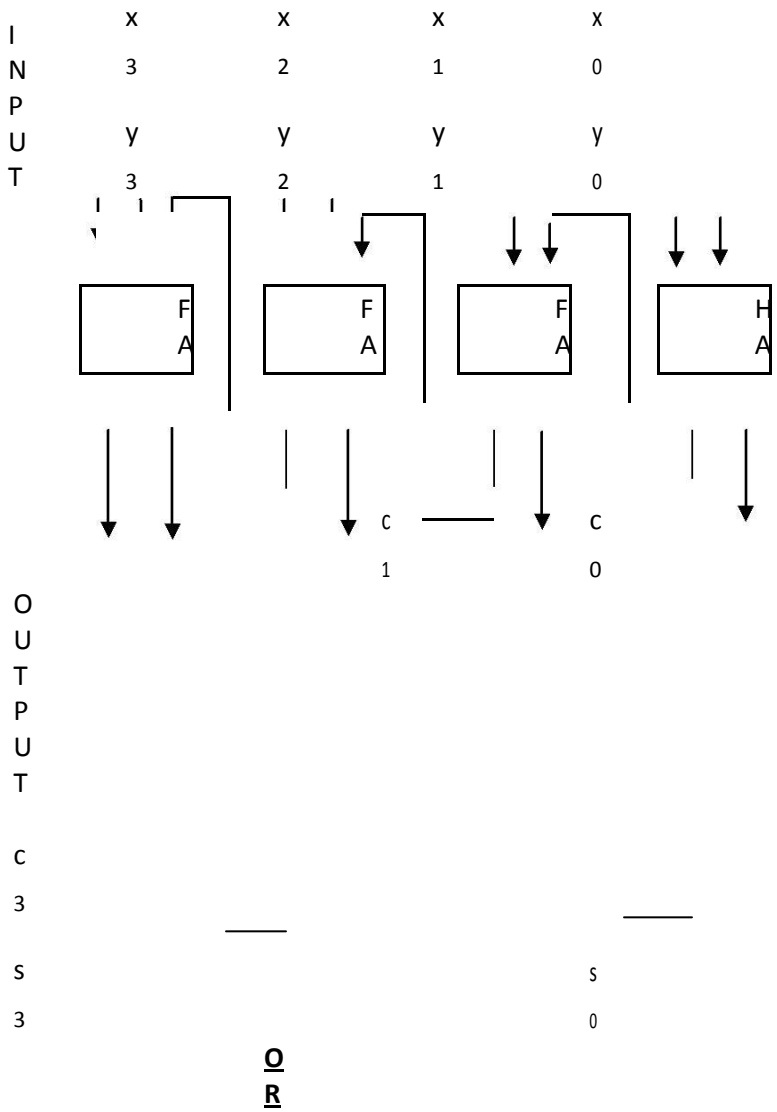
6. Draw the circuit for FA **(Students are required to try this out themselves)**:

Generally, the block diagram for FA is shown as below :

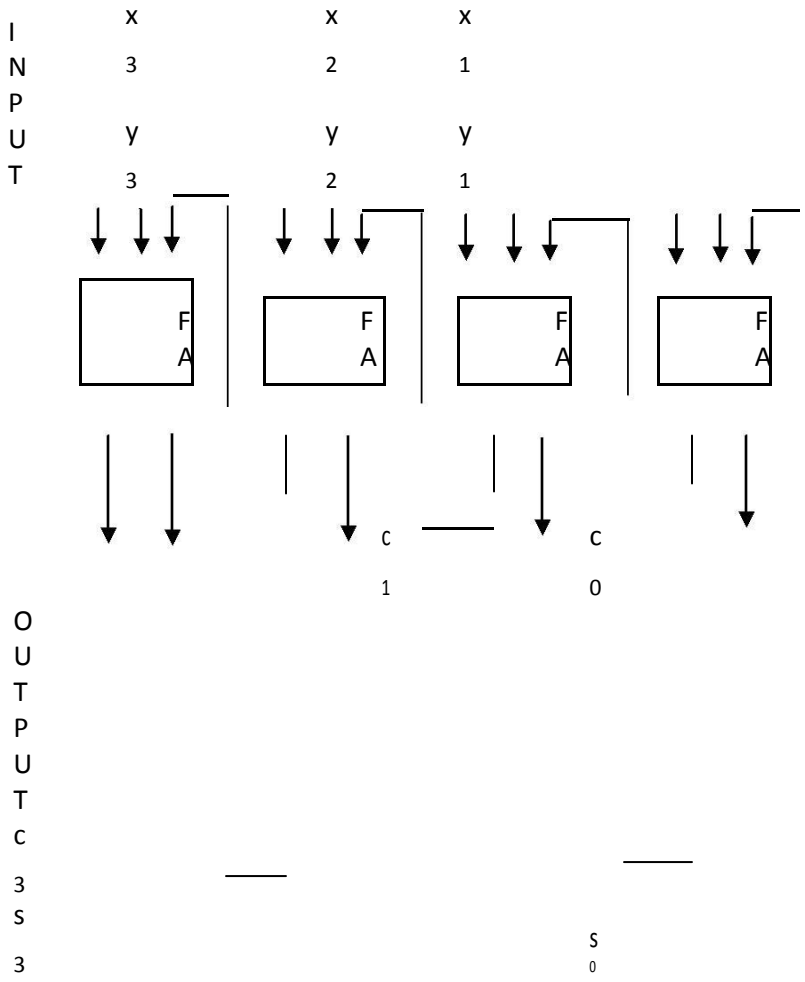




To construct a 2-bit parallel adder, 3 FA and 1 HA are required like the diagram below with the input as $X = x_3x_2x_1x_0$ and $Y = y_3y_2y_1y_0$ (X and Y are binary numbers 2-bit) and the output (addition result) is $r_3r_2r_1r_0$.



OR



Some Examples of Sequential Circuits: Flip-flop, Register, Serial Adder, etc.

Sequential circuits are a kind of logic circuit where the current output not only depends on the current input but also on the past history of inputs. Another and generally more useful way to view it is that the current output of a sequential circuit depends on the current input and the current state of that circuit. The simplest form of sequential circuit is the flip-flop. Flip-flop is a kind of logic circuit that is capable of exhibiting 2 stable conditions. It is also known as 1-bit memory element and is mostly used to make important computer components such as registers, counters, memory etc.

A few examples of Flip-flop (Sequential Circuit) usage

As priory stated, flip-flop is an example of the simplest form of sequential circuit. It is also a form of memory element where a flip-flop can store 1 bit of data. In this section, examples of sequential circuits that use flip-flop will be given:

Register

Register is an important component in the computer. Generally, it can be categorized into:

1. Storage Register (or Parallel Register)
2. Shift Register (or Serial Register)

Parallel register is made up of a set of 1-bit (flip-flop) that can be written on and read simultaneously. This register is used to store data (output=input). The amount of flip-flop used depends on the size of the register that is to be built. If a parallel register that can store 8 bits of data is to be built, then 8 flip-flops are needed. Diagram 6 below is a 2 bit parallel register that uses flip-flop D. (Note: all kinds of flip-flop can be used to build storage register, but its circuit will differ because ever flip-flop has its own features)

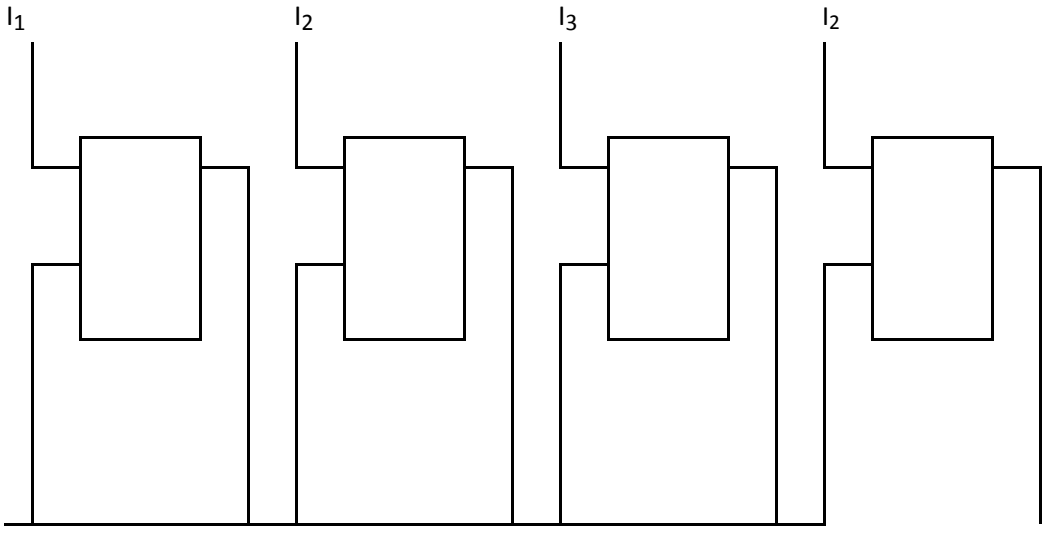


Diagram 6: A 2-bit parallel register
that uses D Flip-flop

Q
2

In the above diagram, 2 bits of input is admitted simultaneously, that is I_1, I_2, I_3 and I_2, whereas its output is also is simultaneous or parallel, that is Q_1, Q_2, Q_3 and Q_2.

In shift register, only one output is produced at a time. There are 2 types of shift register that is shift to right and shift to left. Shift to right register means the rightmost bit of the stated will be taken out first followed by the

following bits after a given clock beat. It's vice versa for move to shift to left register. Diagram 7 below is an example of 2-bit shift to right register that utilizes J-K flip-flop.

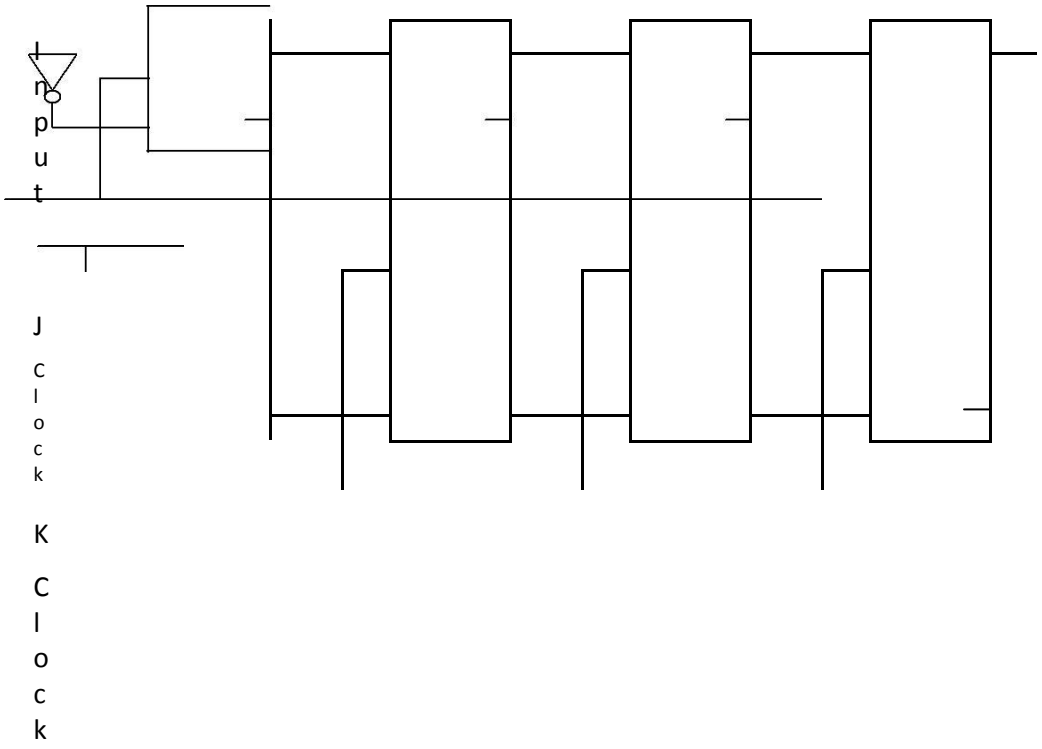


Diagram 7: Shift to Right Register Using J-K Flip-flop

Parallel Adder

In the computer environment, there are 2 types of adders:

1. Parallel Adder
2. Serial Adder

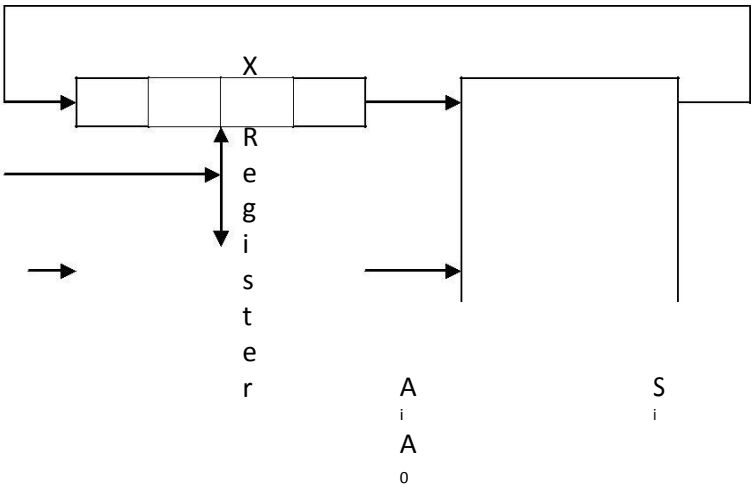
Parallel adder is an adder that performs addition concurrently for each bit involved. Adder in section 2.2 is called a serial adder. Serial Adder performs addition bit by bit starting with the rightmost bit, followed by the following bits. Diagram 8 below is an example of a serial 2-bit adder. This adder uses two Shift to Right Registers, X and Y to hold operand 1 (A = $A_3A_2A_IA_0$) and operand 2 (B = $B_3B_2B_1B_0$), a full adder (see section 2.2) and a flip-flop (usually D flip-flop) to hold the carrier value.

The addition process in the adder are as below :

X = X + Y

that is the X and Y registers will hold operand 1 and operand 2 and the addition result will be kept in the X register. Hence, in the addition, the value in the Y (Operand 2) register cannot change while the X register holds the addition result (the value of operand 1 will be lost)

Note: observe and understand the data movement in the stated circuit after every clock pulse is given.



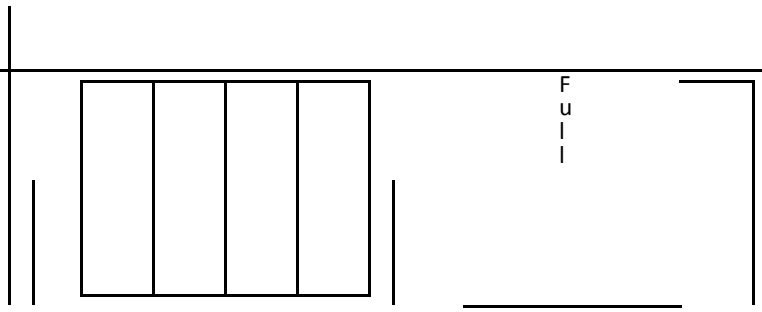


Diagram 8 : 2-bit Serial Adder

UNIT-3 PART-1

3.1.1. Algorithms for fixed point and floating point addition

3.1.1.1. Algorithms for fixed point addition

3.1.1.2. Algorithms for floating point addition

3.1.2. Subtraction, multiplication and division operations.

3.1.3. Hardware Implementation of arithmetic and logic operations,

3.1.4. High Performance arithmetic

UNIT-3 PART-2

Instruction set & Addressing

3.2.1. Memory Locations and Addresses

Mainmemory is the second major subsystem in a computer. It consists of a collection of storage locations, each with a unique identifier, called an address.

Data is transferred to and from memory in groups of bits called words. A word can be a group of 8bits, 16bits, 32bits or 64bits (and growing).

•If the word is 8bits, it is referred to as a byte. The term “byte” is so common in computer science that sometimes a 16-bit word is referred to as a 2-byte word, or a 32-bitword is referred to as a 4 byteword.

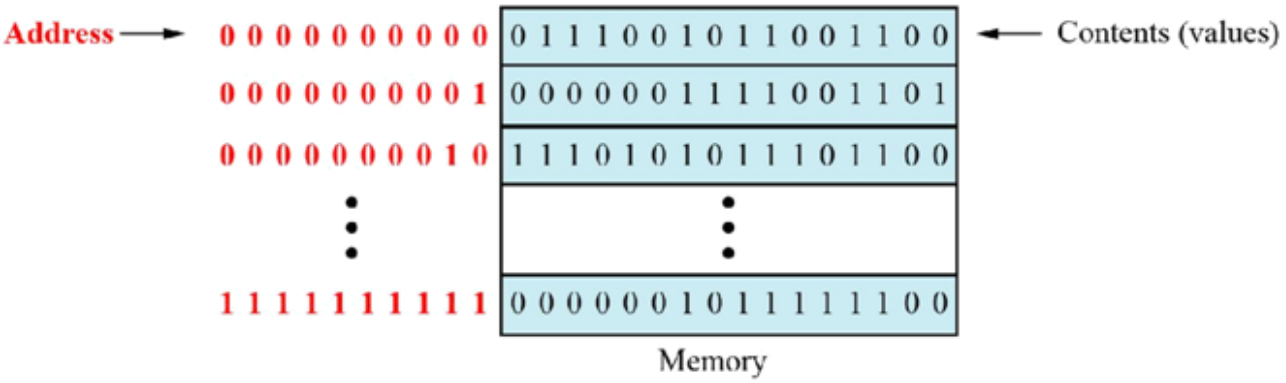


Figure 5.3 Main memory

Address space

•To access a word in memory requires an identifier. Although programmers use a name to identify a word (or a collection of words), at the hardware level each word is identified by an address.

•The total number of uniquely identifiable locations in memory is called the **addressspace**.

•For example, a memory with 64kilobytes (16address line required) and a wordsize of 1byte has an address space that ranges from 0 to 65,535

Table 5.1 Memory units

Unit	*Exact Number of Bytes*	*Approximation*
kilobyte	2^{10} (1024) bytes	10^3 bytes
megabyte	2^{20} (1,048,576) bytes	10^6 bytes
gigabyte	2^{30} (1,073,741,824) bytes	10^9 bytes
terabyte	2^{40} bytes	10^{12} bytes

Assignment of byte addresses

Little Endian (e.g., in DEC, Intel)

a) low order byte stored at lowest address

b) byte0 byte1 byte2 byte3

Big Endian

Big Endian (e.g., in IBM, Motorolla, Sun, HP)

a) high order byte stored at lowest address

b) byte3 byte2 byte1 byte0

Programmers/protocols should be careful when transferring binary data between Big Endian and Little Endian machines

In case of 16 bit data, aligned words begin at byte addresses of 0,2,4,…….

a) In case of 32 bit data, aligned words begin at byte address of 0,4,8,…….

b) In case of 64 bit data, aligned words begin at byte addresses of 0,8,16,……..

c) In some cases words can start at an arbitrary byte address also then, we say that word locations are unaligned

MEMORY OPERATIONS

Today, **general-purpose computers**use a set of instructions called a **program**to process data.

a) A computer executes the program to create output data from input data

b) Both program instructions and data operands are stored in memory

Two basic operations requires in memory accessLoad operation (Read or Fetch)-Contents of specified memory location are read by processor Store operation (Write)-Data from the processor is stored in specified memory location

INSTRUCTION SET ARCHITECTURE:-Complete instruction set of the processor

BASIC 4 TYPES OF OPERATION:-

i) Data transfer between memory and processor register

ii) Arithmetic and logic operation

iii) Program sequencing and control

iv) I/O transfer

Register transfer notation (RTN)

Transfer between processor registers & memory, between processor register & I/O devices Memory locations, registers and I/O register names are identified by a symbolic name in uppercase alphabets

LOC,PLACE,MEM are the address of memory location R1 , R2,… are processor registers

DATA_IN, DATA_OUT are I/O registers

Contents of location is indicated by using square brackets [ ] RHS of RTN always denotes a values, and is called Source

LHS of RTN always denotes a symbolic name where valueis to be storedand is called destination not modified

ASSEMBLY LANGUAGE NOTATION (ALN)

RTN is easy to understand and but cannot be used to represent machine instructions

Mnemonics can be converted to machine language, which processor understands using assembler

Eg:

1. MOVE LOCN, R2
2. ADD R3, R2, R4

3.2.2. Machine addresses and sequencing

Each machine instruction is executed through the application of a sequence of microinstructions. Clearly, we must be able to sequence these; the collection of microinstructions which implements a particular machine instruction is called a routine.

The MCU typically determines the address of the first microinstruction which implements a machine instruction based on that instruction's opcode. Upon machine power-up, the CAR should contain the address of the first microinstruction to be executed.

The MCU must be able to execute microinstructions sequentially (e.g., within routines), but must also be able to ``branch'' to other microinstructions as required; hence, the need for a sequencer.

The microinstructions executed in sequence can be found sequentially in the CM, or can be found by branching to another location within the CM. Sequential retrieval of microinstructions can be done by simply incrementing the current CAR contents; branching requires determining the desired CW address, and loading that into the CAR.

Addressing Sequencing

CAR

Control Address Register

control ROM

control memory (CM); holds CWs

opcode

opcode field from machine instruction

mapping logic

hardware which maps opcode into microinstruction address

branch logic

determines how the next CAR value will be determined from all the various possibilities

multiplexors

implements choice of branch logic for next CAR value

incrementer

generates CAR + 1 as a possible next CAR value

SBR

used to hold return address for subroutine-call branch operations

Conditional branches are necessary in the microprogram. We must be able to perform some sequences of micro-ops only when certain situations or conditions exist (e.g., for conditional branching at the machine instruction level); to implement these, we need to be able to conditional execute or avoid certain microinstructions within routines.

Subroutine branches are helpful to have at the microprogram level. Many routines contain identical sequences of microinstructions; putting them into subroutines allows those routines to be shorter, thus saving memory.

Mapping of opcodes to microinstruction addresses can be done very simply. When the CM is designed, a ``required'' length is determine for the machine instruction routines (i.e., the length of the longest one). This is rounded up to the next power of 2, yielding a value k such that 2 k microinstructions will be sufficient to implement any routine.

The first instruction of each routine will be located in the CM at multiples of this ``required'' length. Say this is N. The first routine is at 0; the next, at N; the next, at 2*N; etc. This can be accomplished very easily. For instance, with a four-bit opcode and routine length of four microinstructions, k is two; generate the microinstruction address by appending two zero bits to the opcode:

addressing

Alternately, the n-bit opcode value can be used as the ``address'' input of a 2n x M ROM; the contents of the selected ``word'' in the ROM will be the desired M-bit CAR address for the beginning of the routine implementing that instruction. (This technique allows for variable-length routines in the CM.) >pp We choose between all the possible ways of generating CAR values by feeding them all into a multiplexor bank, and implementing special branch logic which will determine how the muxes will pass on the next address

to the CAR. As there are four possible ways of determining the next address, the multiplexor bank is made up of N 4x1 muxes, where N is the number of bits in the address of a CW. The branch logic is used to determine which of the four possible ``next address'' values is to be passed on to the CAR; its two output lines are the select inputs for the muxes

3.2.3. Addressing Modes

The term ***addressing modes*** refers to the way in which the operand of an instruction is specified. Information contained in the instruction code is the value of the operand or the address of the result/operand. Following are the main addressing modes that are used on various platforms and architectures.

1) Immediate Mode

The operand is an immediate value is stored explicitly in the instruction:

Example: SPIM (opcode dest, source)

li $11, 3 // loads the immediate value of 3 into register $11

li $9, 8 // loads the immediate value of 8 into register $9

Example : (textbook uses instructions type like, opcode source, dest)

move #200, R0; // move immediate value 200 in register R0

2) Index Mode

The address of the operand is obtained by adding to the contents of the general register (called index register) a constant value. The number of the index register and the constant value are included in the instruction code. Index Mode is used to access an array whose elements are in successive memory locations. The content of the instruction code, represents the starting address of the array and the value of the index register, and the index value of the current element. By incrementing or decrementing index register different element of the array can be accessed.

Example: SPIM/SAL - Accessing Arrays

```
.data
array1: .byte 1,2,3,4,5,6
.text
__start:
move $3, $0          # $3 initialize index register with 0
add $3, $3,4         # compute the index value of the fifth element
sb $0, array1($3)  # array1[4]=0
# store byte 0 in the fifth element of the array
# index addressing mode
done
```

3) Indirect Mode

The effective address of the operand is the contents of a register or main memory location, location whose address appears in the instruction. Indirection is noted by placing the name of the register or the memory address given in the instruction in parentheses. The register or memory location that contains the address of the operand is a pointer. When an execution takes place in such mode, instruction may be told to go to a

specific address. Once it's there, instead of finding an operand, it finds an address where the operand is located.

NOTE:

Two memory accesses are required in order to obtain the value of the operand (fetch operand address and fetch operand value).

Example: (textbook) ADD (A), R0

(address A is embedded in the instruction code and (A) is the operand address = pointer variable)

Example: SPIM - simulating pointers and indirect register addressing

The following "C" code:

int *alpha=0x00002004, q=5; *alpha = q;

could be translated into the following assembly code:

alpha: .word 0x00002004 # alpha is and address variable # address value is 0x00002004 q: .word 5
....
lw $10,q # load word value from address q in into $10
# $10 is 5
lw $11,alpha # $11 gets the value 0x0002004
# this is similar with a load immediate address value
sw $10,($11) # store value from register $10 at memory location
# whose address is given by the contents of register $11
# (store 5 at address 0x00002004)

Example: SPIM/SAL - - array pointers and indirect register addressing

.data
array1: .byte 1,2,3,4,5,6
.text
__start:
la $3, array1 # array1 is direct addressing mode
add $3, $3,4 # compute the address of the fifth element
sb $0, ($3) # array1[4]=0 , byte accessing
indirect addressing mode
done

4) Absolute (Direct) Mode

The address of the operand is embedded in the instruction code.

Example: (SPIM)

```
beta: .word 2000

lw $11, beta    # load word (32 -bit quantity) at address beta into register $11
# address of the word is embedded in the instruction code
# (register $11 will receive value 2000)
```

5) Register Mode

The name (the number) of the CPU register is embedded in the instruction. The register contains the value of the operand. The number of bits used to specify the register depends on the total number of registers from the processor set.

Example (SPIM)

```
add $14,$14,$13  # add contents of register $13 plus contents of
# register $14 and save the result in register $14
```

No memory access is required for the operand specified in register mode.

6) Displacement Mode

Similar to index mode, except instead of a index register a base register will be used. Base register contains a pointer to a memory location. An integer (constant) is also referred to as a displacement. The address of the operand is obtained by adding the contents of the base register plus the constant. The difference between index mode and displacement mode is in the number of bits used to represent the constant. When the constant is represented a number of bits to access the memory, then we have index mode. Index mode is more appropriate for array accessing; displacement mode is more appropriate for structure (records) accessing.

Example: SPIM/SAL - Accessing fields in structures

```
.data
student: .word 10000 #field code
.ascii "Smith" #field name
.byte # field test
.byte 80,80,90,100 # fields hw1,hw2,hw3,hw4
.text __start:
la $3, student  # load address of the structure in $3
# $3 base register
add $17,$0,90 # value 90 in register $17
# displacement of field "test" is 9 bytes
#
sb $17, 9($3)   # store contents of register $17 in field "test"
```

```
# displacement addressing mode
done
```

7) Autoincrement /Autodecrement Mode

A special case of indirect register mode. The register whose number is included in the instruction code, contains the address of the operand. Autoincrement Mode = after operand addressing , the contents of the register is incremented. Decrement Mode = before operand addressing, the contents of the register is decrement.

Example: SPIM/SAL - - simulating autoincrement/autodecrement addressing mode

(**MIPS** has no autoincrement/autodecrement mode)

```
lw $3, array1($17) #load in reg. $3 word at address array1($17)
addi $17,$17,4        #increment address (32-bit words) after accessing
#operand this can be re-written in a "autoincrement like mode":
lw+ $3,array1($17)    # lw+ is not a real MIPS instruction
subi $17,$17,4        # decrement address before accessing the operand
lw $3,array1($17)
```

NOTE: the above sequence can be re-rewritten proposing an "autodecrement instruction", **not real** in MIPS architecture.

-lw $3, array1($17)

3.2.4. Instruction Formats

The most common fields found in instruction format are:-

(1) An operation code field that specified the operation to be performed

(2) An address field that designates a memory address or a processor registers.

(3) A mode field that specifies the way the operand or the effective address is determined.

Computers may have instructions of several different lengths containing varying number of addresses. The number of address field in the instruction format of a computer depends on the internal organization of its registers. Most computers fall into one of three types of CPU organization.

(1) Single Accumulator organization ADD X AC ® AC + M [×]

(2) General Register Organization ADD R1, R2, R3 R ® R2 + R3

(3) Stack Organization PUSH X

Three address Instruction

Computer with three addresses instruction format can use each address field to specify either processor register are memory operand.

ADD R1, A, B A1 ® M [A] + M [B]

ADD R2, C, D R2 ® M [C] + M [B] X = (A + B) * (C + A)

MUL X, R1, R2 M [X] R1 * R2

The advantage of the three address formats is that it results in short program when evaluating arithmetic expression. The disadvantage is that the binary-coded instructions require too many bits to specify three addresses.

Two Address Instruction

Most common in commercial computers. Each address field can specify either a processes register on a memory word.

R
1
®
R M
1 [
, A
A]
R
1
®
R
1
+
R M
1 [
, B
B]
R R X
2 2
, =
®
C (

M A

[+

C

] B

)

*

(

C

+

D

)

R

2

®

R

2

+

R M

2

, [

D

D]

R

1

®

R

R 1

1

, *

R R

2 2

X M

1
[
R X
1]
®
R
1

One Address instruction

It used an implied accumulator (AC) register for all data manipulation. For multiplication/division, there is a need for a second register.

A
C
®
M
[
A
]
A
C
®
A
C
+
M
[
B
]
S M X
T

O R E T [T] ® A C = (A + B) × (C + A)

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

A C ® M (C) A C ® A C + M

(D)AC ® AC + M (T)M

STORE X [×] ® AC

Zero – Address Instruction

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