Chapter Three

Central Processing

3.1 Introduction

Each complex task carried out by a computer needs to be broken down into a sequence of simpler tasks and a **binary machine instruction** is needed for the most primitive tasks. Consider a task that adds two numbers, held in memory locations designated by B and C and stores the result in memory location designated by A.

Example 1: A = B + C

This assignment can be broken down (compiled) into a sequence of simpler tasks or assembly instructions, e.g:

Assembly Instruction	Effect
LOAD R2, B	Copy the contents of memory location designated by B into Register 2
ADD R2, C	Add the contents of the memory location designated by C to the contents of Register 2 and put the result back into Register 2
STORE R2, A	Copy the contents of Register 2 into the memory location designated by A.

Each of these assembly instructions needs to be encoded into binary for execution by the Central Processing Unit (CPU). Each instruction is divided into a number of instruction fields that encode a different piece of information; the **OPCODE** field

identifies the CPU operation required. We can represent the above Assembly instruction using 16 bits word as below.

Field Name	**OPCODE**				**REG**		**ADDRESS**									
Field Width	4-bits				2-bits		10-bits									
	1	O	A	D	R	2										

In order to execute the program , its instructions and data needs to place within main memory. We'll place our 3-instruction program in memory starting at address 080H and we'll place the variables A, B and C at memory words 200H, 201H, and 202H respectively. Such placement results in the following memory layout prior to program execution.

Memory Address	Memory	Assembly Instruction
080	1A01	LOAD R2, [201H]
081	3A02	ADD R2, [202H]
082	2A00	STORE R2, [200H]
Etc	**Etc**	**Etc**
200	00	A = 0
201	09	B = 9
202	06	C = 6

Figure 1 Memory placement for example 1

3.2 CPU Organisation & Operation

The operations of CPU describes in terms of the Fetch-Execute cycle. Different CPU organisation has the same behaviour when it comes to program execution; this behaviour could be represented using flow charts (figure 1). CPU starts program execution by fetching the instruction first (if there any) , then execute it , check if there are any interrupts, and resume the fetch and execute cycle , if an interrupts occurs then the control unit will halt the current program and transfer control to

interrupt handler (to execute the interrupt) . We will talk about interrupts in next chapter.

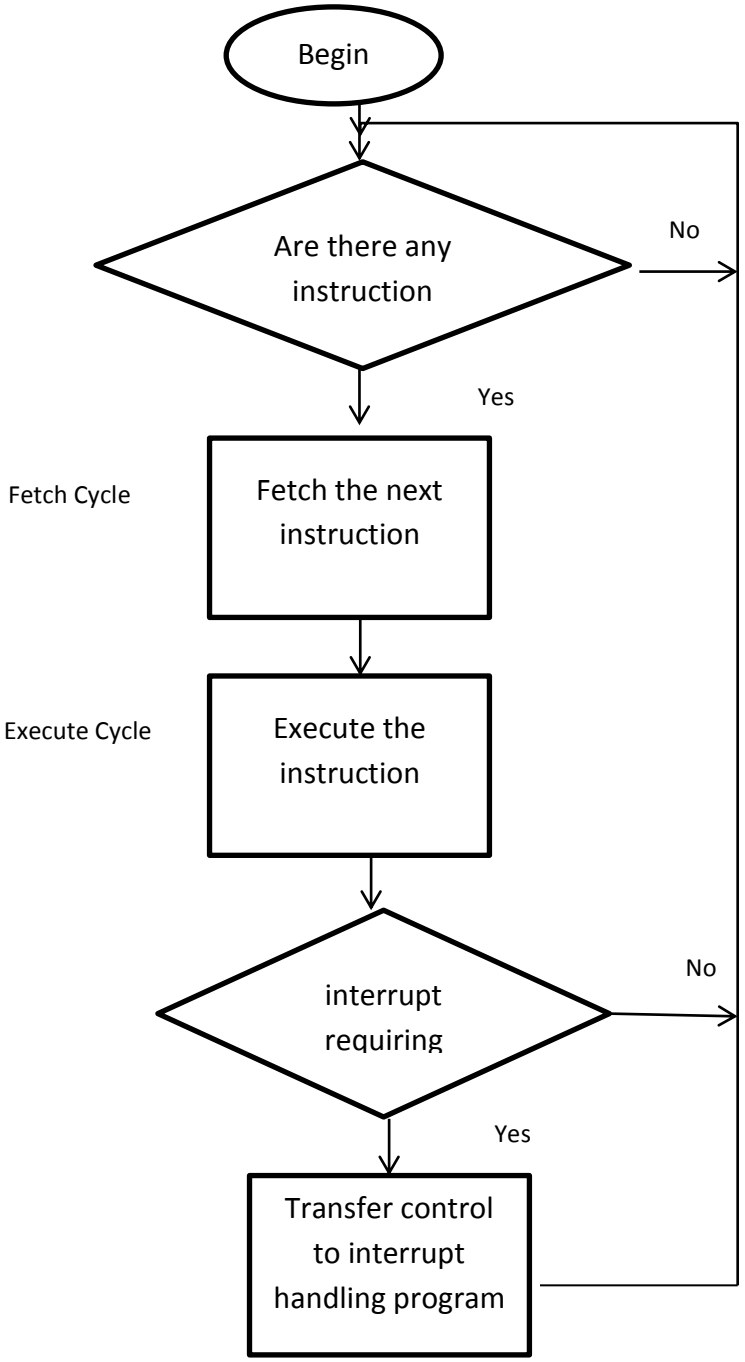


Figure 2 Overview of CPU behavior

3.3 Hardware Implementation

Registers consists of flip-flops, n bit register have **n** flip flops, these flip flops numbered in sequence from **0 to n-1**

Register Individual bits inside register

To initial H/W we must chick following:

1. The set of register
2. The sequence of microoperation
3. The control that initiates the sequence of microoperation

Basic symbol for microoperation:

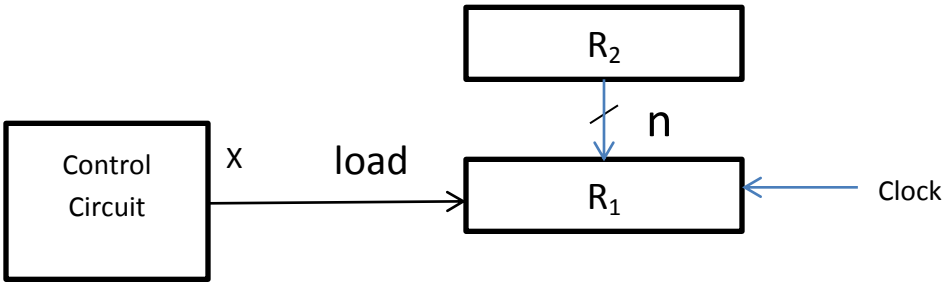
Symbol	Description	Example
Letters	Denotes a register	MAR, R_1
( )	Part of register	$R_{1(0-7)}$, $R_{2(L)}$
←	Transfer of information	$R_2 \leftarrow R_1$
,	Separate between to microoperation	

Control : operation

X : $R_1 \leftarrow R_2$

Information transfer from one register to another denoted by $\mathbf{R_2 \leftarrow R_1}$, If the transfer is occur under predetermined control signal (suppose the signal is **x**) then we designate it by **x:** $\mathbf{R_2 \leftarrow R_1}$.

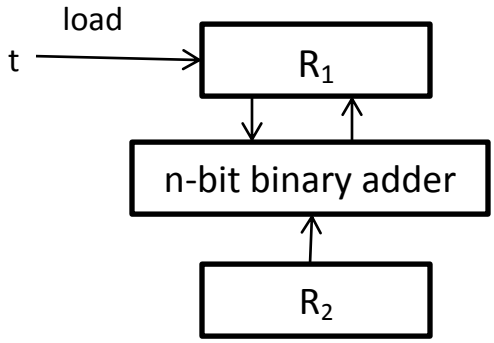
Control circuits controls register operations, transferring $R_1 \leftarrow R_2$ under a control signal x, required microoperation called load which will load the content of R_2 to R_1 , thus the H.W implementation of this operation is :



Block diagram for Hardware implementation of Register transfer

Q/ Suppose you have an addition of 2 register, what is the HW implementation of t:$R1 \leftarrow R_1+R_2$

We need: an adder, R1, R2 Registers, specify where the result will be, accordingly the h/w implementation is:



3.3.1 Bus and memory transfer

If we need to move data from and to multiple registers, problem arises. The number of wires will be so large if separate lines are used to connect all registers with each other. To completely connect **n** registers we need **n(n-1)**

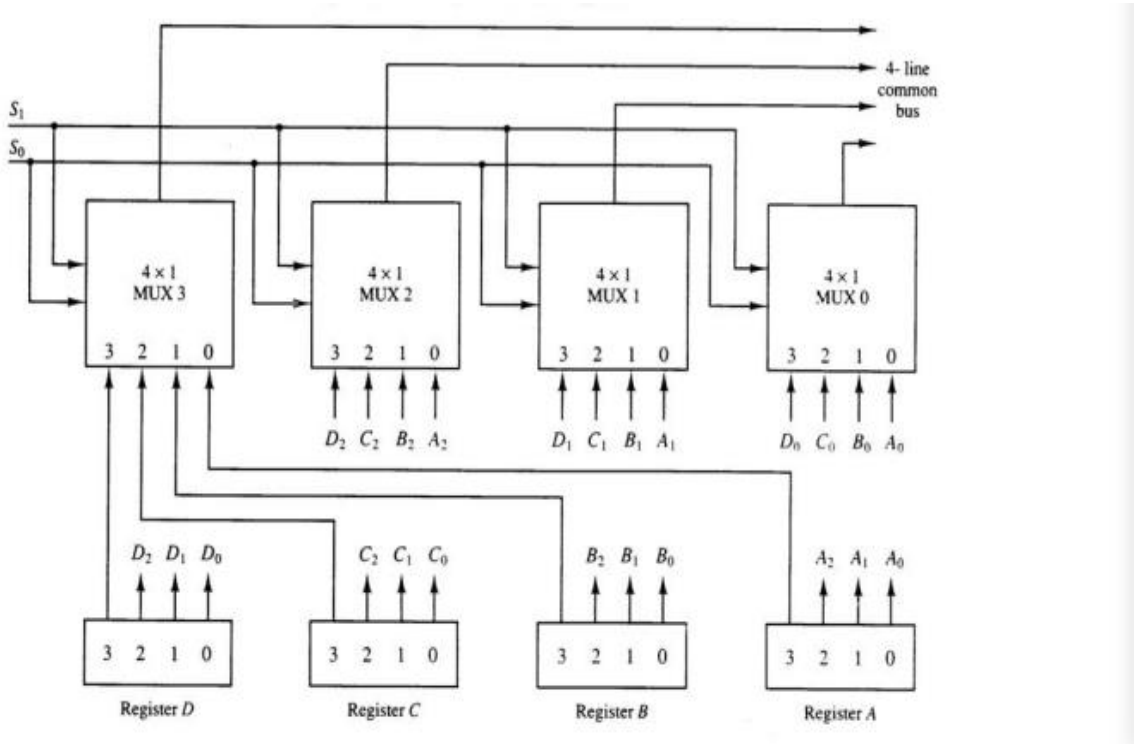
lines. So the cost is in order of $O(n^2)$. This is not a realistic approach to be used in a large digital system. The solution is to use a common "Bus".

Registers are connected using bus; the bus structure consists of a set of common lines, one for each bit of a register, control signal used to determine which register is selected by the bus during each transfer. One way of constructing such a bus is by using **multiplexers**. For **k-registers** each with **n-bits** we need **n-line bus**, which mean we need **n multiplexer** (one for each bit), the size of each multiplexer must be **k X 1**. The number of selected lines required is $\log_2 k$.

Bus: is a path (of a group of wires) over which information is transferred, from any of several sources to any of several destinations.

e.g./ draw a bus system that connect 4-registers (4-bits each).

Solution:



Another way of constructing a bus is by using ***buffers or 3-state gates***. Three state gates is a digital circuit that represent three states. Two of the states are logic 0 and 1, the third state is high impedance state (behave as an open circuits). The following figure represents three state gates.

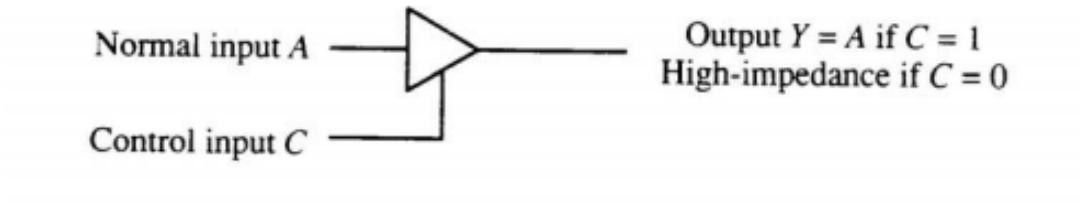


Figure 3 Three State Gate

With control 1, the output (of the three state buffer gate) equals the normal input, while in 0 controls the gate goes to a high-impedance state.

3.3.2 Microoperation:

Microoperations are the basic operations that can be performed by a system on data stored in registers. Each microoperation describes a simple operation performed on data in one or more registers. Such operation can be executed by one clock pulse.

There are **four** categories of the most common microoperations:

1- **Register Transfer** (B←A, A←X, …) : Transfer ***binary*** information from one register to another.
2- **Arithmetic Operation (A← A-B, A← A+B, …):** perform arithmetic operation on ***numeric*** data.
3- **Logical Operation (compare, …):** perform bit manipulation on ***numeric*** data.
4- **Shift Micro-Operation**: to shift the temporary data which are present in register.

The basic Arithmetic microoperations are addition, subtraction, increment, decrement and Arithmetic Shift. Multiply and divide are not included as microoperations.

Multiply implemented by a sequence of add and shift microoperations, while divide is implemented as a sequence of subtract and shift.

Normally subtraction ($R_3 \leftarrow R_1 - R_2$) implemented through 1's complement, and then adding 1 to 1's complement produce 2's complement, adding the number in R1 to 2's complement of R2 is equivalent to subtracting.

3.3.2.1 Implementation of microoperation

Internal hardware organization of a digital computer is best defined by:

- Registers it contains and their functions
- Sequence of micro operations performed on data inside registers
- Control that define the sequence of micro operations

3.3.2.2 Arithematic Microoperation

To implement add microperation we will review basic components needed which are:

a) A ***Binary adder i***s a digital circuit that generates the arithmetic sum of two binary number of any length. (Remember A full adder adds two bits only and previous carry). The binary adder consists of full adder circuits connected in cascade, accordingly to build an n-bit binary adder we need n full-adders.

e.g Build digital Circuit that can perform the following operation

A+B, A-B (assume both A & B are 4-bits size)

Solution: First remember that we need binary adder to perform the adding operation for 4-bits, figure 5 represent the required adder

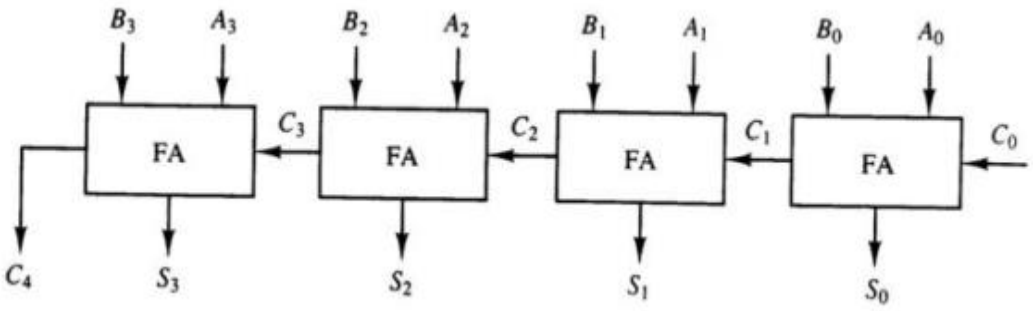


Figure (5): 4-bits binary adder

The subtraction of A-B required three steps:

$A + \overline{B} + 1$

- implements 1's complements for B,
- add 1 to get 2's complements
- and finally add result to A.

The addition and subtraction operation could be combined into one common circuit by including an XOR gate with each full adder. So the solution will be figure 6 below:

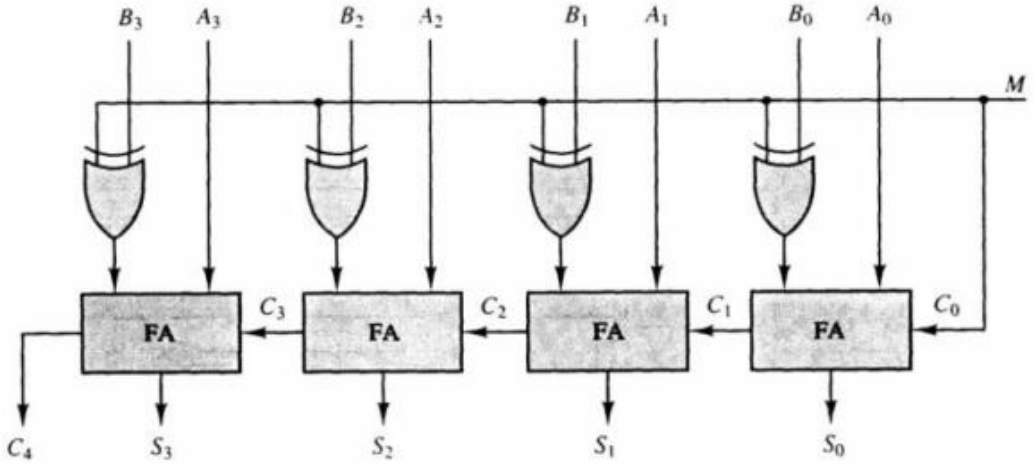


Figure 4: 4-bit binary Adder-Subtractor

b) **Binary Incrementor**: The binary incrementor always adds one to the number in a register. To implement the increment microoperation we will use:

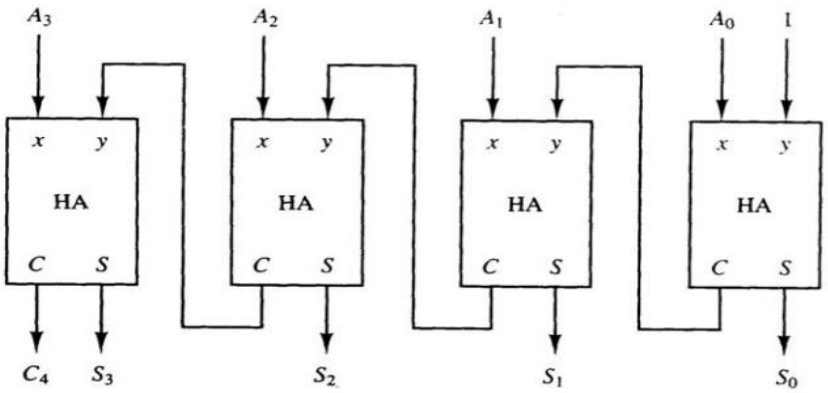
1) a binary counter. When clock transition arrives the count is incremented by one.

2) if the increment is to be performed independent of a particular register, then use half adders connected in cascade.

3) An n-bit binary incrementor requires **n** half adder.

least significant adder always have one of its input as "1" while its carry is cascaded to other half adders.

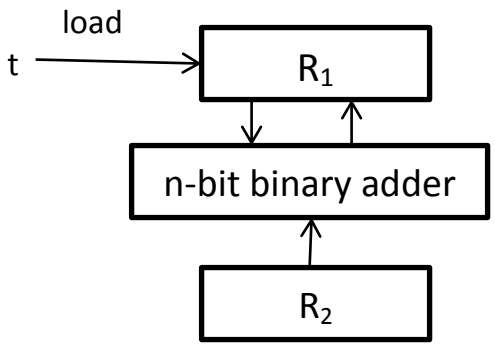
e.g.: Design an incremental for the number A (it is 4-bit number)



4-bit Binary incrementer

Ex: Give the H/W required for implementing the following micro-operations.

a) t : $R_1 \leftarrow R_1 + R_2$



b) Arithmetic micro-operation with overflow detection.

1. Unsigned Binary Addition overflow

When the "Binary Addition Algorithm" is used with **unsigned** binary integer representation: The result is CORRECT only if the **CARRY OUT** of the high order column is **ZERO**. Unsigned overflow occurred when carry out =1; For example:

1		
1101 0010	210_{10}	
0110 1101	109_{10}	
0011 1111	63_{10}	**The carry bit of 1 indicates overflow.**

2. Signed Binary Addition overflow

There are many schemes for representing negative integers with patterns of bits. Two's complement is one of many ways to represent negative integers with bit patterns. With **two's complement** representation the result of addition is correct if the ***carry into*** the ***high order column*** is the ***same as the carry out of the high order column***. Overflow is detected by comparing these two bits. Here are some more examples:

No Overflow	No Overflow	Overflow	Overflow
11111 111 0011 1111 (63_{10}) 1101 0101 (-43_{10}) 0001 0100 (20_{10})	00000 011 1100 0001 (-63_{10}) 0010 1011 (43_{10}) 1110 1100 (-20_{10})	01111 100 0011 1111 (63_{10}) 0110 0100 (100_{10}) 1010 0011 (-93_{10})	10000 000 1100 0001 (-63_{10}) 1001 1100 (-100_{10}) 0101 1101 (93_{10})

The truth table of tow's sign bits (A_{sign} and B_{sign} bit) is shown below:

INPUTS			OUTPUTS		
A_{sign}	B_{sign}	CARRY IN	CARRY OUT	SUM_{sign}	OVERFLOW
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	0	1	0
1	0	1	1	0	0
1	1	0	1	0	1
1	1	1	1	1	0

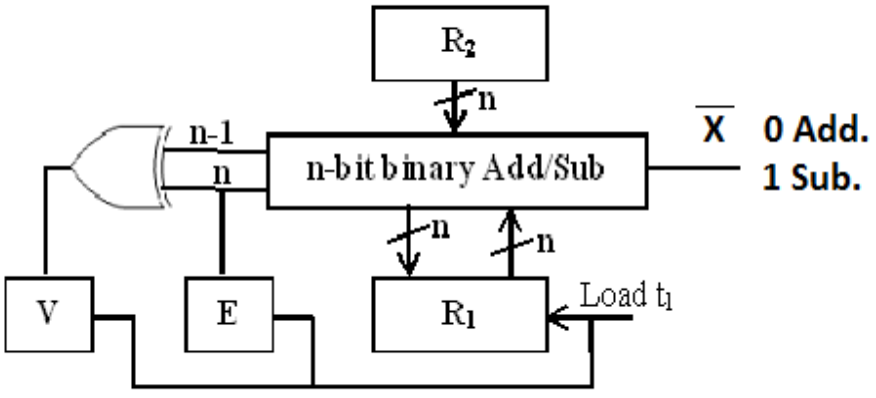
From the above truth table:

- Notice that **overflow** occurs only when: $\mathbf{CARRY_{in} \neq CARRY_{out}}$
- or simply: $\mathbf{V = C_{in}\ XOR\ C_{out}}$; where **V** is the overflow signal.

So that, the Arithmetic micro-operation with overflow detection can be design as:

t_1X: $ER_1 \leftarrow R_1+R_2$

t_1X: $ER_1 \leftarrow R_1+R_2+1$



Each of the **arithmetic micro operations** can be implemented in one **composite** arithmetic circuit. This circuit comprised of:

- Parallel full adders and
- Multiplexers are used to choose between the different operations.

The multiplexer controls which data is fed into input of the adder (suppose B, A represent that inputs). The output of the binary adder is computed from

$$\mathbf{D = A + B + C_{in}}$$

The B input can have one of 4 different values: B,$\overline{B}$, always "1", or always "0"
Figure (7); below represent 4-bit arithmetic circuit.

S_1	S_2	C_{in}	Input B	D =A+B+C_{in}	Operation
0	0	0	B	A+B	ADD
0	0	1	B	A+B+1	ADD with carry
0	1	0	$\overline{B}$	A+$\overline{B}$	Sub with borrow (1's comp)
0	1	1	$\overline{B}$	A+$\overline{B}$+1	Sub in 2's comp.
1	0	0	0	A	Transfer
1	0	1	0	A+1	Increment
1	1	0	1	A-1	Decrement
1	1	1	1	A	Transfer

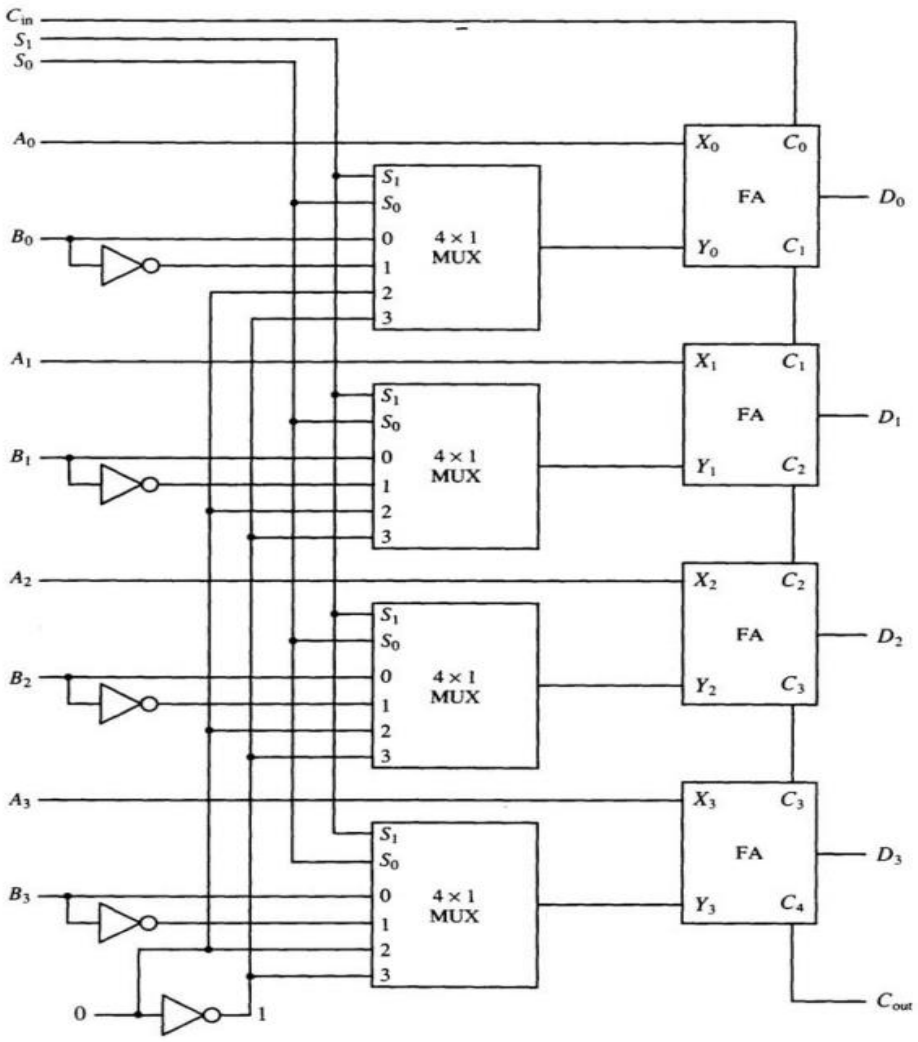


Figure 5 : 4-bit arithmetic circuit

3.3.2.3 Logical Microoperation

Logic micro operation specifies binary operations on the strings of bits in registers. Logic micro operations are bit-wise operations, i.e., they work on the individual bits of data. These are useful for bit manipulations on binary data and also useful for making logical decisions based on the bit value. There are many different

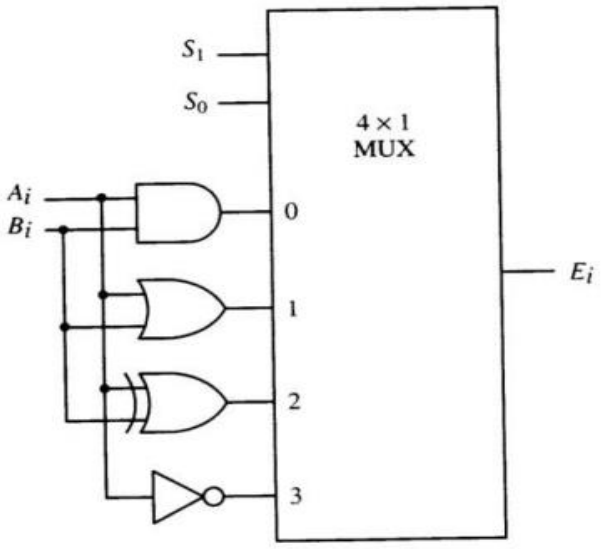
logic functions that can be defined over two binary input variables. However, most systems only implement four of these: ***AND, OR, XOR, Complement/NOT***.

The others can be created from combination of these. The hardware implementation of logic micro operation requires the insertion of the most important gates like AND, OR, EXOR, and NOT for each bit or pair of bits in the registers.

Build a logical circuit to generate the four basic logic micro operations required:

- ***four gates*** (**AND, OR, XOR, NOT**) and
- ***a multiplexer***.

The two selection lines of the multiplexer selects one of the four logic operations available at one time. The circuit shows one stage for bit "i" but for logic circuit of n bits the circuit should be repeated n times but with one remark; the selection pins will be shared with all stages.



(a) Logic diagram

S_1	S_0	Output	Operation
0	0	$E = A \wedge B$	AND
0	1	$E = A \vee B$	OR
1	0	$E = A \oplus B$	XOR
1	1	$E = \overline{A}$	Complement

(b) Function table

Figure 6 Simple Logic circuit

3.3.2.4 Shift Microoperations

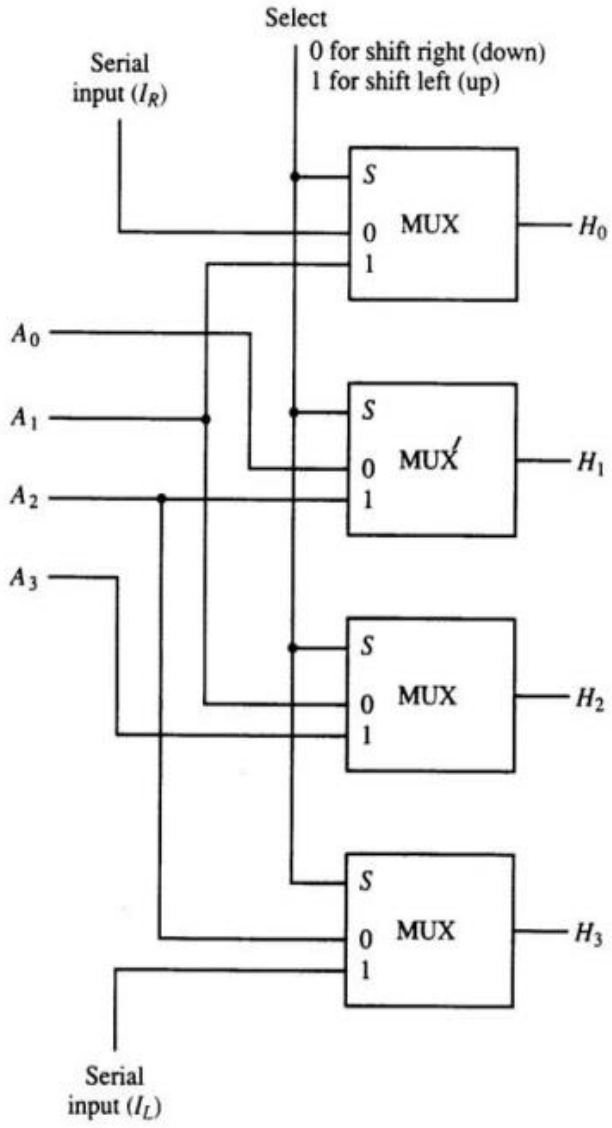
Shift micro-operations are used for **serial transfer** of data beside they are used in conjunction with arithmetic, logic, and other data processing operations. There are **3 types** of shift micro operations.

- **Logical shift** : Logical shift is one that transfers 0 through the serial input
- **Circular shift** : The circular shift rotates of the register around the two ends without loss of information
- **Arithmetic shift**: Arithmetic shift is a micro-operation that shifts a signed binary number to the left or right. Arithmetic shift must leave sign bit unchanged.

We can implement Shift microoperation using:

- Use **bidirectional shift register** with parallel load, In that case we need two clocks pulse one to **load** the value and another to **shift**.
- Another solution which is **more efficient** is to implement the shift operation with ***combinational circuits*** (combinational circuit will construct using multiplexers.)

Figure 9- below represents 4-bit combinational circuit shifter



Function table

Select	Output			
S	H_0	H_1	H_2	H_3
0	I_R	A_0	A_1	A_2
1	A_1	A_2	A_3	I_L

Figure 7: 4-bit combinational circuit shifter

3.3.3 Arithmetic Logic Shift Unit

Instead of having individual registers performing micro-operations directly, computer systems employ a ***number of storage registers connected to a unit*** called **Arithmetic Logic Unit (ALU**). This unit has ***2 operands input ports*** and ***one output port*** and a ***number of select lines*** to help in ***selecting different operations***. The ALU is made of ***combinational circuit*** so that the entire register transfer operation ***from the sources to the destination is performed in one clock cycle***. The arithmetic, logic, and shift

circuits (implemented previously) will be combined in one ALU with common selection inputs. Simple stage (bit) of ALU with its table is shown blow (figure 10). The arithmetic and logic units will select their operations simultaneously when $\mathbf{S_0}$ and $\mathbf{S_1}$ are applied; while $\mathbf{S_2}$ and $\mathbf{S_3}$ will select one of those unit outputs or a shift left bit stage or shift right bit stage. The circuit shown provides **8 arithmetic operations, 4 logic operations, and 2 shift operations.**

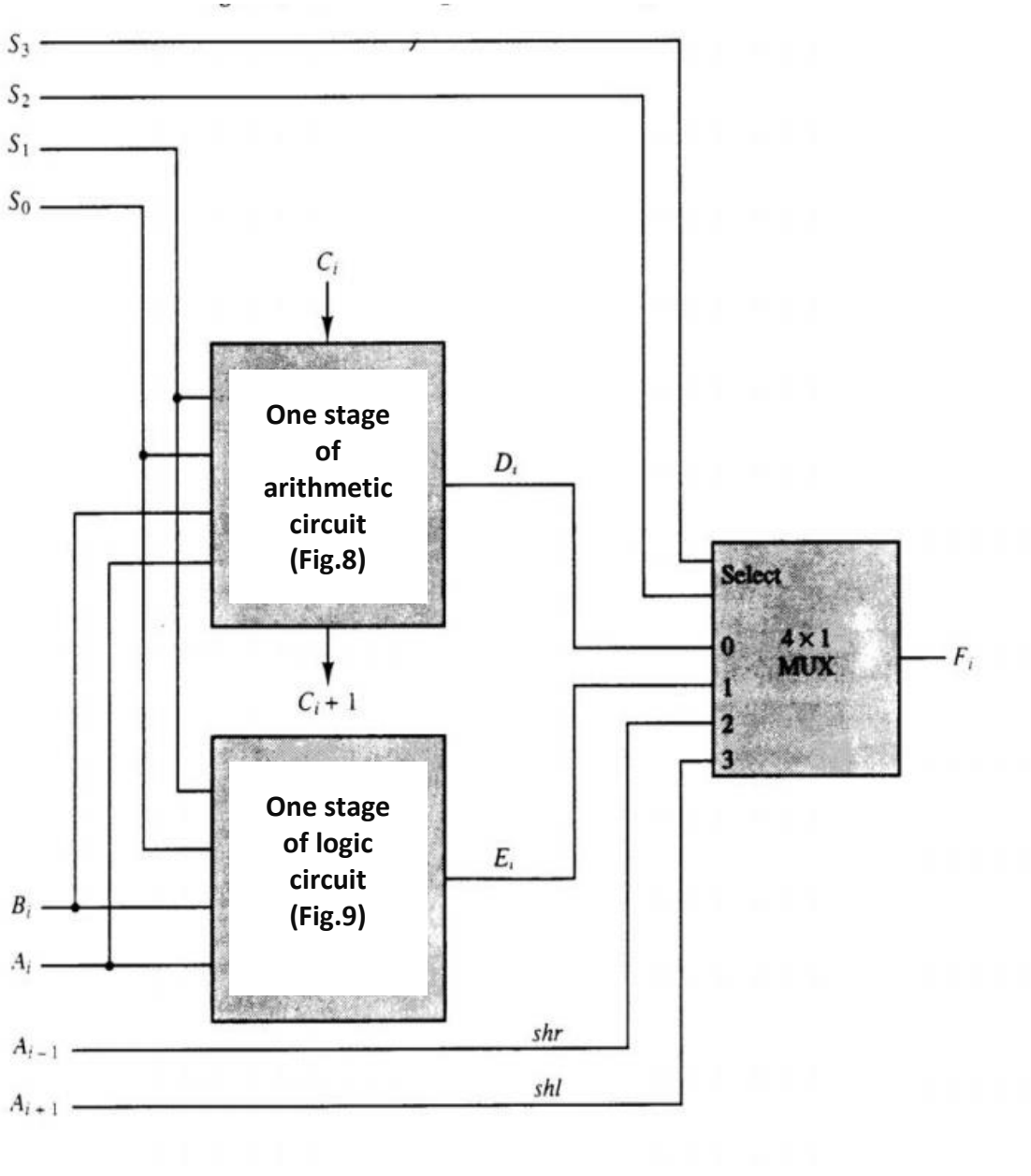


Figure 8 simple Arithmetic and logic unit

Table 2 Function table for Arithmetic -Logical (AND SHIFT) Unit

Operation select						
S_3	S_2	S_1	S_0	C_{in}	Operation	Function
0	0	0	0	0	$F = A$	Transfer A
0	0	0	0	1	$F = A + 1$	Increment A
0	0	0	1	0	$F = A + B$	Addition
0	0	0	1	1	$F = A + B + 1$	Add with carry
0	0	1	0	0	$F = A + \overline{B}$	Subtract with borrow
0	0	1	0	1	$F = A + \overline{B} + 1$	Subtraction
0	0	1	1	0	$F = A - 1$	Decrement A
0	0	1	1	1	$F = A$	Transfer A
0	1	0	0	×	$F = A \wedge B$	AND
0	1	0	1	×	$F = A \vee B$	OR
0	1	1	0	×	$F = A \oplus B$	XOR
0	1	1	1	×	$F = \overline{A}$	Complement A
1	0	×	×	×	$F = \text{shr } A$	Shift right A into F
1	1	×	×	×	$F = \text{shl } A$	Shift left A into F

3.4 Instruction Set

If **m** is the maximum number of explicit main memory addresses allowed in any processor instruction, the processor may be called an **m-address machine**.

3.4.1 Op-code

The number of bits in the op-code filed in any computer can be choose into two different ways

1- Select a **constant** number of bits (e.g. **4 bits** make 2^4 different instructions).
2- Choose **variable** length op-code size depends on the probability of instruction (e.g. the frequency have large probability have less number of codes & vice-versa).

Ex: Mors code; the shortest code; a single dot; is assigned to the most common letter in English (e), while the largest code are assigned to the least frequently occurring letters.

Ex: variable length based on instruction occurrence probability?

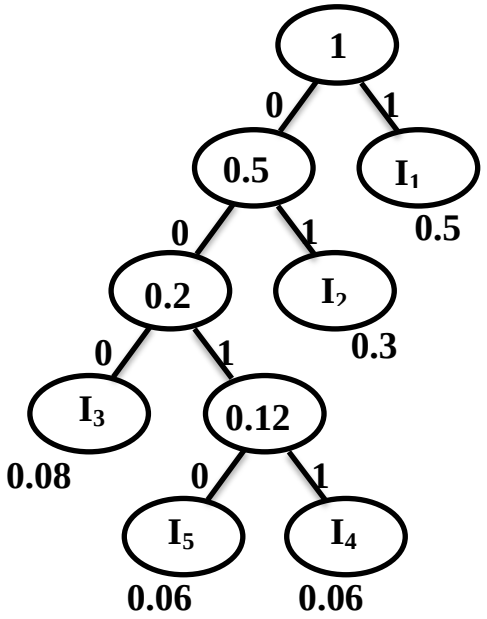
Instruction	Probability of occurrence (pi)	Op-code (ci)
I_1	0.5	1
I_2	0.3	01
I_3	0.08	000
I_4	0.06	0011
I_5	0.06	0010

The codes in example above are generated on Huffman Coding Algorithm.

Elements	Probability	Huffman Tree	Code
I_1	0.5		1
I_2	0.3		01
I_3	0.08		000
I_4	0.06		0011
I_5	0.06		0010

Another Method

Instruction	Probability	Code
I1	0.5	1
I2	0.3	01
I3	0.08	000
I4	0.06	0011
I5	0.06	0010



H.W

1) Find the Huffman code of:

Elements	Probability	Huffman Tree	Code
A	40		
B	20		
C	14		
D	10		
E	06		
F	05		
G	03		
H	02		

2) Find the Huffman code of

Symbol	Freq.	Huffman Tree	Code
A	**45**		
B	**13**		
C	**12**		
D	**16**		
E	**9**		
F	**5**		

3.5 Fundamental Concepts & Examples

3.5.1 CPU Organization According To Number of Buses

There are several components inside a CPU, namely, ALU, control unit, general purpose register, Instruction registers etc. Now we will see how these components are organized inside CPU. There are several ways to place **t**hese components and interconnect them.

A. Single-Bus Organization

In this case, the arithmetic and logic unit (ALU), and all CPU registers are connected via a single common bus. This bus is internal to CPU and this internal bus is used to transfer the information between different components of the CPU. This organization is termed as single bus organization, since only one internal bus is used for transferring of information between different components of CPU. We have external bus or buses to CPU also to connect the CPU with the memory module and I/O devices. The external memory bus is also shown in the **figure (A)** connected to the CPU via the memory data and address register **MDR** and **MAR**.

In this organization, two registers, namely Y and Z are used which are transparent to the user. Programmer can not directly access these two registers. These are used as **input** and **output** buffer to the ALU which will be used in ALU operations. They will be used by CPU as ***temporary storage*** for some instructions.

Execution of an instruction requires the following three steps to perform by the CPU:-

1- **Fitch** the contain of memory location pointed by the (PC) and store the instruction code in (IR): $IR \leftarrow M[PC]$
2- **Increment** the contain of (PC) by one: $PC \leftarrow [PC] + 1$
3- **Carry out** the actions specified by the instruction stored in the (IR)

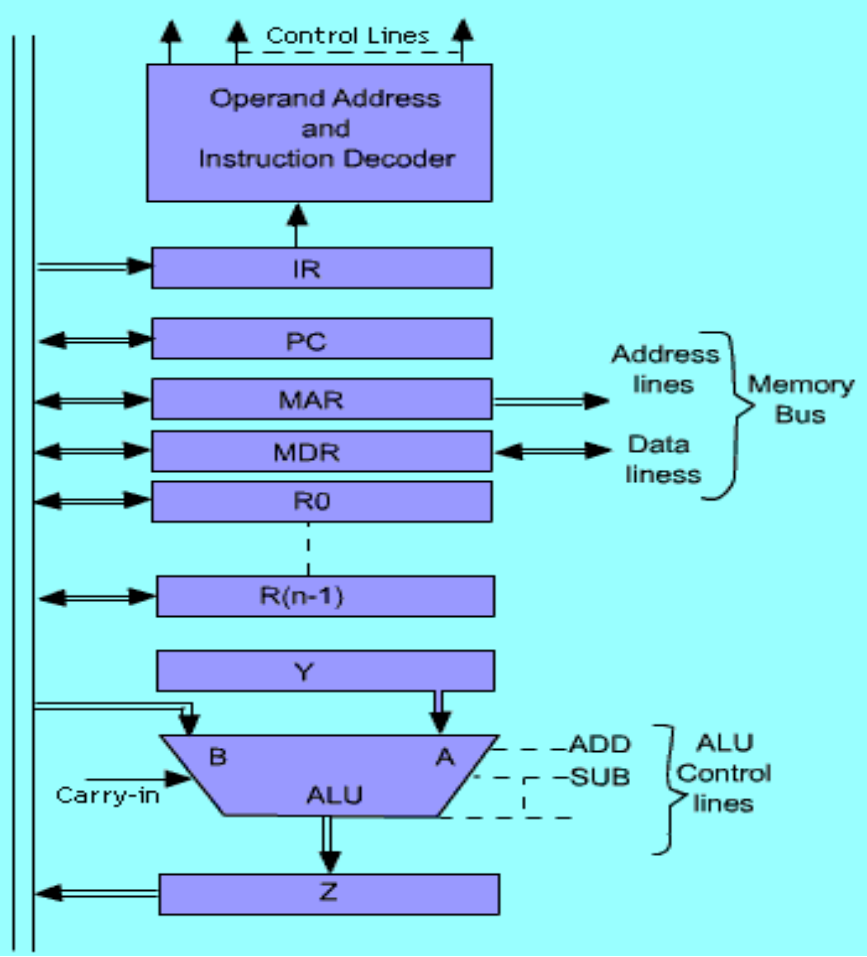


Figure (A): Single Bus Organization of the data path inside the CPU.

❖ ***Fetching a word into memory***

Example1: assume that the address of the memory location to be accessed is kept in register R1 and that the memory contents to be loaded into register R2. This is done by the following sequence of operations:

1. MAR ← (R_1)
2. Read
3. The CPU waits for MFC (memory function complete) signal from memory.
4. R_2 ← (MDR)

❖ ***Storing a word into memory***

Example2: assumes that the data word to be stored in the memory is in register R2 and that the memory address is in register R1. The memory write operation requires the following sequence:

1. MAR ← (R1)
2. MDR ← (R_2)
3. Write
4. Wait for MFC

❖ ***Register transfer operation***

Register transfer operations enable data transfer between various blocks connected to the common bus of CPU. We have several registers inside CPU and it is needed to transfer information from one register another. As for example during memory write operation data from appropriate register must be moved to MDR.

Since the input output lines of all the register are connected to the common internal bus, we need appropriate input output gating. The input and output gates for register $\mathbf{R_i}$ are controlled by the signal $\mathbf{R_{i\text{-}in}}$ and $\mathbf{R_{i\text{-}out}}$ respectively.

Thus, when $\mathbf{R_{i\text{-}in}}$ set to 1 the data available in the common bus is loaded into $\mathbf{R_i}$. Similarly when, $\mathbf{R_{i\text{-}out}}$ is set to 1, the contents of the register $\mathbf{R_i}$ are placed on the bus. To transfer data from one register to other register, we need to generate the appropriate register gating signal.

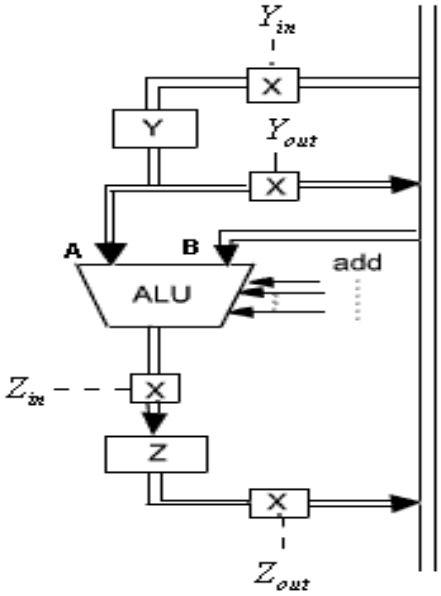
Example: to transfer the contents of register R1 to register R2, the following actions are needed:

- Enable the output gate of register R_1 by setting R_{1out} to 1.
  -- This places the contents of R1 on the CPU bus.
- Enable the input gate of register R_2 by setting $R_{2\ in}$ to 1.
  -- This loads data from the CPU bus into the register R2.

❖ ***Arithmetic or logic operation***

Generally ALU is used inside CPU to perform arithmetic and logic operation. ALU is a combinational logic circuit which does not have any internal storage. Therefore, to perform any arithmetic or logic operation (say binary operation) both the input should be made available at the **two inputs** of the ALU simultaneously. Once both the inputs are available then appropriate signal is generated to perform the required operation. We may have to use **temporary storage** (register) to carry out the operation in ALU.

The sequence of operations that have to carry out to perform one ALU operation depends on the organization of the CPU. Consider an organization in which one of the operand of ALU is stored in some **temporary register Y** and **other operand** is directly taken from **CPU internal bus**. The result of the ALU operation is stored in another temporary **register Z**.



Ex: $R_3 \leftarrow R_1 + R_2$

1. $R_{1\ out}$, Y_{in}
2. $R_{2\ out}$, ADD, Z_{in}
3. Z_{out}, $R_{3\ in}$

B. Two -Bus Organization

It is an alternative structure, where two different internal buses are used in CPU. All register outputs are connected to bus A, add all registered inputs are connected to bus B. as shown in figure (B).

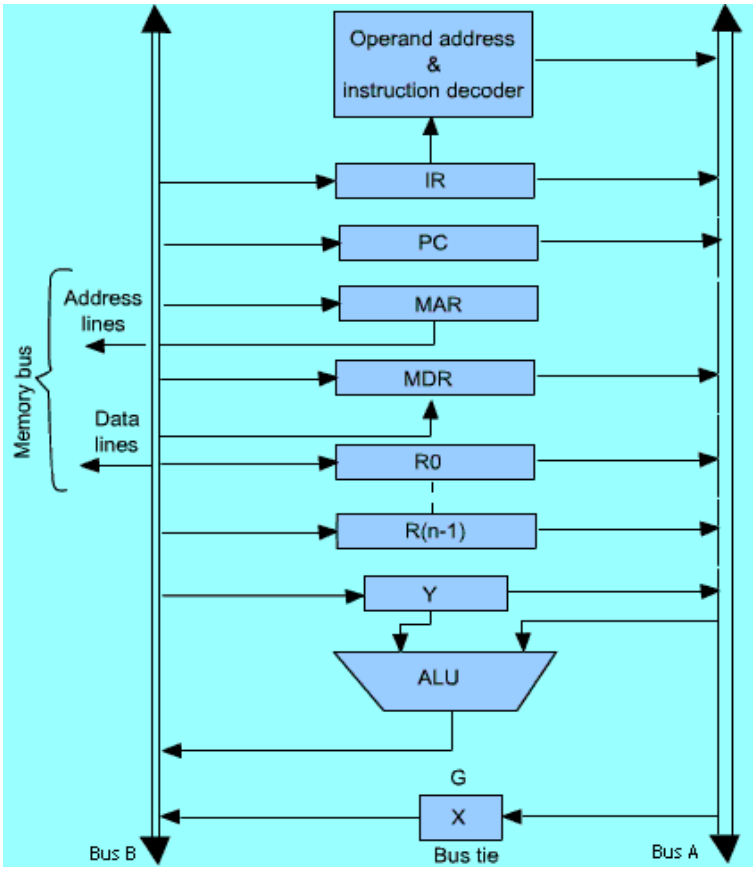


Figure (B): Two Bus Structure.

There is a special arrangement to transfer the data from one bus to the other bus. The buses are connected through the bus tie G. When this tie is enabled data on bus A is transfer to bus B. When G is disabled, the two buses are electrically isolated.

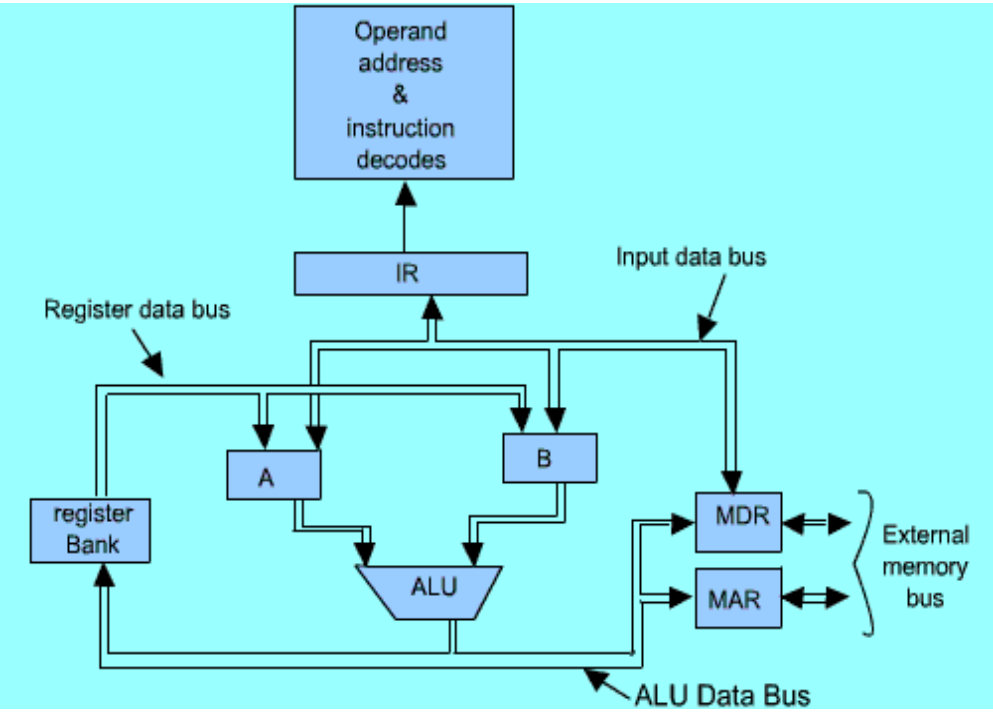
Since two buses are used here the temporary register Z is **not required** here which is used in single bus organization to store the result of ALU. Now result can be directly transferred to bus B, since one of the inputs is in bus A. With the bus tie disabled, the result can directly be transferred to destination register.

Ex: $R_3 \leftarrow R_1 + R_2$

Step	Action
1	$R_{1\ out}$, G_{enable}, Y_{in}
2	$R_{2\ out}$, ADD, ALU_{out}, $R_{3\ in}$

C. Three – Bus Organization

In this organization each bus connected to only **one output** and number of inputs. The elimination of the need for connecting more than one output to the same bus leads to faster bus transfer and simple control. A simple three-bus organization is shown in the figure (C).



Figure(C): Three Bus Structure.

A **multiplexer** is provided at the input to each of the two work registers **A** and **B**, which allow them to be loaded from either the input data bus or the register data bus.

Two separate input data buses are present – **one** is for ***external data transfer***, i.e. retrieving from memory and the **second** one is for ***internal data transfer*** that is transferring data from general purpose register to other building block inside the CPU.

We may use one bus **tie G1** between input data bus and ALU output bus and another bus **tie G2** between register data bus and ALU output data bus.

3.6.2 Execute Of Complete Instruction

To execute a complete instruction we need to take help of the following basic operations in some particular order to execute an instruction.

- Fetch information from memory to CPU
- Store information to CPU register to memory
- Transfer of data between CPU registers.
- Perform arithmetic or logic operation and store the result in CPU registers.

Ex: Give the control sequence for the execution "Add contain of memory location addressed in memory direct mode to register R_1"?

Execution of this instruction requires the following action:

1. Fetch instruction
2. Fetch first operand (Contents of memory location pointed at by the address field of the instruction)
3. Perform addition
4. Load the result into R1.

Following sequence of control steps are required to implement the above operation for the single-bus architecture that we have discussed in earlier section.

Step	Action
1	PC_{out}, MAR_{in}, Read, Clear Y, Set carry, ADD, Z_{in}
2	Z_{out}, PC_{in}, Wait for MFC
3	MDR_{out}, IR_{in}
4	Address field of IR_{out}, MAR_{in}, Read
5	$R_{1\ out}$, Y_{in}, wait for MFC
6	MDR_{out}, ADD, Z_{in}
7	Z_{out}, $R_{1\ in}$
8	END

Instruction execution proceeds as follows:

In Step1:

The instruction fetch operation is initiated by loading the contents of the PC into the MAR and sending a read request to memory.

To perform this task first of all the contents of PC have to be brought to internal bus and then it is loaded to MAR. To perform this task control circuit has to generate the PC_{out} signal and MAR_{in} signal.

After issuing the read signal, CPU has to wait for some time to get the MFC signal. During that time PC is updated by 1 through the use of the ALU. This is accomplished by setting one of the inputs to the ALU (Register Y) to 0 and the other input is available in bus which is current value of PC. At the same time, the carry-in to the ALU is set to 1 and an add operation is specified.

In Step 2:

The updated value is moved from register Z back into the PC. Step 2 is initiated immediately after issuing the memory Read request without waiting for completion of memory function. This is possible, because step 2 does not use the memory bus and its execution does not depend on the memory read operation.

In Step 3:

Step3 has been delayed until the MFC is received. Once MFC is received, the word fetched from the memory is transferred to IR (Instruction Register), because it is an

instruction. Steps 1 through 3 constitute the instruction fetch phase of the control sequence. The instruction fetch portion is same for all instructions. Next step onwards, instruction execution phase takes place.

In Step 5:

The destination field of IR, which contains the address of the register R1, is used to transfer the contents of register R1 to register Y and wait for Memory function Complete. When the read operation is completed, the memory operand is available in MDR.

In Step 6:

The result of addition operation is performed in this step.

In Step 7:

The result of addition operation is transfered from temporary register **Z** to the destination register **R1** in this step.

In step 8:

It indicates the end of the execution of the instruction by generating End signal. This indicates completion of execution of the current instruction and causes a new fetch cycle to be started by going back to step 1.

Branching

Branching is accomplished by replacing the current contents of the PC by the branch address, that is, the address of the instruction to which branching is required. Consider a branch instruction in which branch address is obtained by adding an offset **X**, which is given in the address field of the branch instruction, to the current value of PC.

EX: Consider the following unconditional branch instruction: JUMP **X**

The control sequence that enables execution of an unconditional branch instruction using the single – bus organization is as follows:

Step	Action
1	PC_{out}, MAR_{in}, Read, Clear Y, Set Carry-in to ALU, Add ,Z_{in}
2	Z_{out}, PC_{in}, Wait for MFC
3	MDR_{out}, IR_{in}
4	PC_{out}, Y_{in}
5	Address field-of IR_{out}, Add, Z_{in}
6	Z_{out}, PC_{in}
7	End

Execution starts as usual with the fetch phase, ending with the instruction being loaded into the IR in step 3.

To execute the branch instruction, the execution phase starts in step 4.

In Step 4: The contents of the PC are transferred to register Y.

In Step 5: The offset **X** of the instruction is gated to the bus and the addition operation is performed.

In Step 6: The result of the addition, which represents the branch address, is loaded into the PC.

In Step 7: It generates the End signal to indicate the end of execution of the current instruction.

3.6.3 Design of Control Unit

To execute an instruction, the control unit of the CPU must generate the required control signal in the proper sequence. As for example, during the fetch phase, CPU has to generate $\boldsymbol{PC_{out}}$ signal along with other required signal in the first clock pulse. In the second clock pulse CPU has to generate $\boldsymbol{PC_{in}}$ signal along with other required signals. So, during fetch phase, the proper sequence for generating the signal to ***retrieve from*** and ***store to PC*** is $\boldsymbol{PC_{out}}$ and $\boldsymbol{PC_{in}}$ **.**

To generate the control signal in proper sequence, a wide variety of techniques exist. Most of these techniques, however, fall into one of the two categories,

1- Hardwired Control
2- Microprogrammed Control

A. Hardwired Control

In this hardwired control techniques, the control signals are generated by means of hardwired circuit. The main objective of control unit is to generate the ***control signal*** in ***proper sequence***.

The control sequence for execution of two instructions is different. ***Of course***, the ***fetch phase*** of all the instructions ***remains same***. It is clear that ***control signals depend on the instruction***, i.e., the contents of the instruction register. It is also observed that execution of some of the instructions depend on the contents of condition code or status flag register, where the control sequence depends in conditional branch instruction.

Hence, the required control signals are usually determined by the following information:

- Contents of the control counter
- Contents of the instruction register
- Contents of the condition code and other status flags. These include the status flags like carry, overflow, zero, etc.

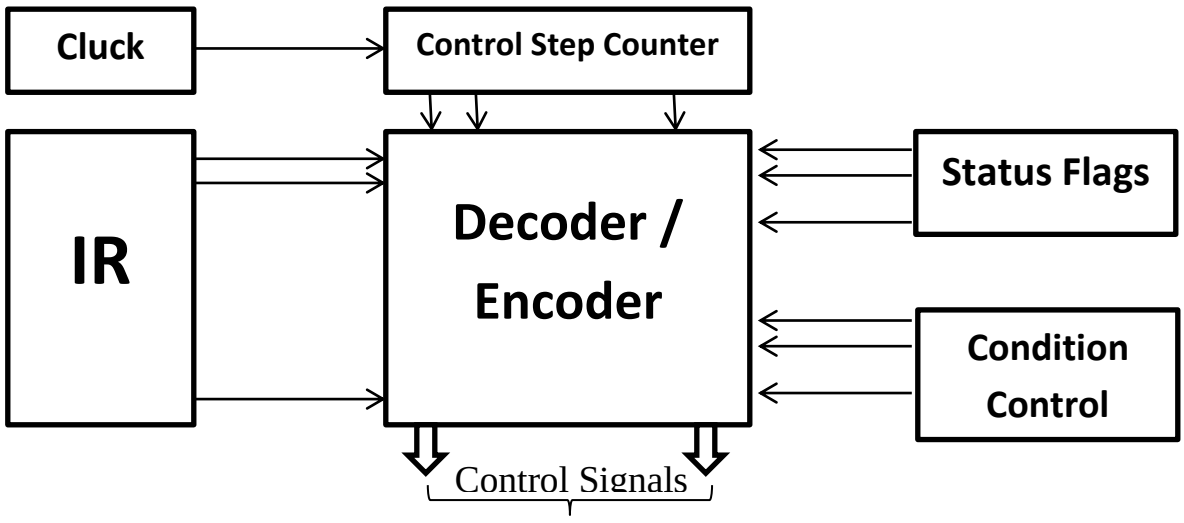


Fig (3.15): Control Unit Organization.

The simplified view of the control unit is given in the figure (3.15) (Prev. page). The decoder/encoder block is simply a combinational circuit that generates the required control outputs depending on the state of all its input.

The decoder part of decoder/encoder part provides a separate signal line for each control step, or time slot in the control sequence. Similarly, the output of the instructor decoder consists of a separate line for each machine instruction loaded in the IR, one of the output lines *INS1* to *INSm* is set to 1 and all other lines are set to 0. The detailed view of Control Unit organization is shown in the figure (3.16) below:

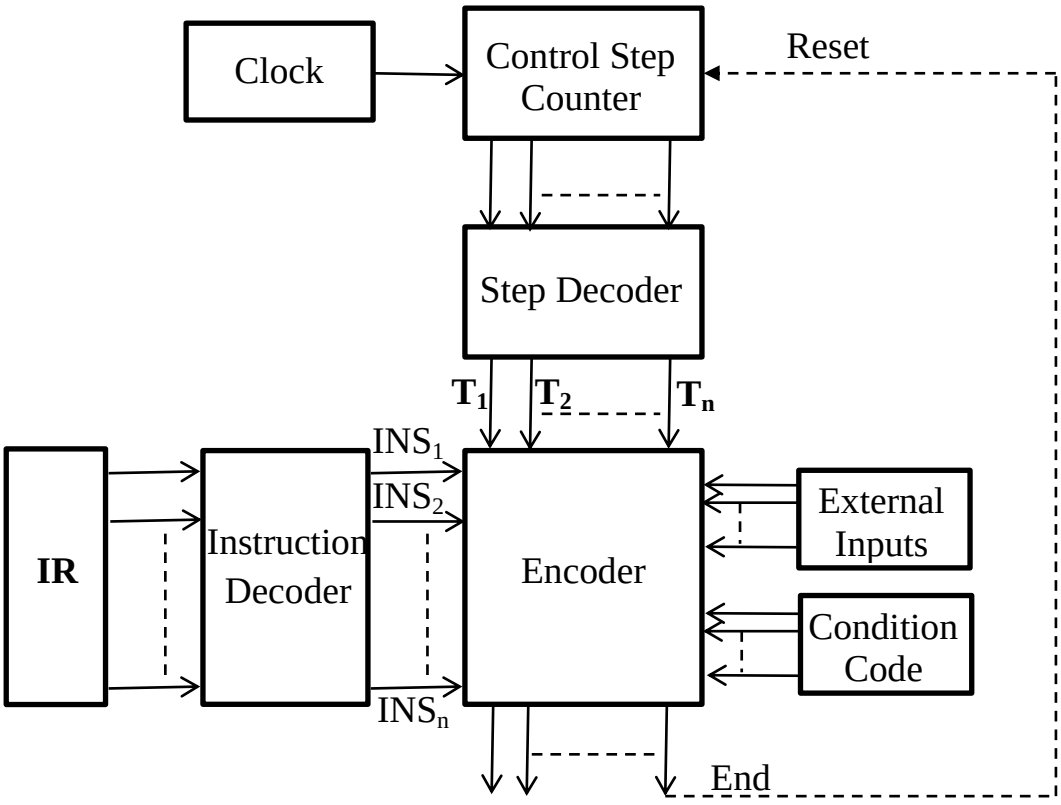


Figure (3.16): Detailed view of Control Unit organization.