A

Project Report on

Embedded processor design and Implementation of CAM

In partial fulfillment of the requirements of

Bachelor of Technology (Electronics and Instrumentation Engg.)

Submitted By

Jaswant Kumar Rout (Roll No.10507008)

Session: 2008-09

Department of Electronics and Communication Engineering

National Institute of Technology

Rourkela-769008

Orissa

A

Project Report on

Embedded processor design and Implementation of CAM

In partial fulfillment of the requirements of

Bachelor of Technology (Electronics and Instrumentation Engg.)

Submitted By

Jaswant Kumar Rout (Roll No.10507008)

Session: 2008-09

Under the guidance of

Prof. (Dr.) K.K.Mahapatra

NATIONAL INSTITUTE OF TECHNOLOGY ROURKELA

Department of Electronics and Communication Engineering

National Institute of Technology
Rourkela-769008
Orissa

National Institute of Technology
Rourkela

CERTIFICATE

This is to certify that that the work in this thesis report entitled “Embedded processor design and implementation of CAM” submitted by Jaswant Kumar Rout in partial fulfillment of the requirements for the degree of Bachelor of Technology in Electronics and instrumentation engineering Session 2005-2009 in the department of Electronics and Communication Engineering, National Institute of Technology Rourkela, Rourkela is an authentic work carried out by them under my supervision and guidance.
To the best of my knowledge the matter embodied in the thesis has not been submitted to any other University /Institute for the award of any degree.

Date:

Prof. (Dr) K.K.Mahapatra
Department of Electronics and Communication
National Institute of Technology
Rourkela - 769008

ACKNOWLEDGEMENT

I owe a debt of deepest gratitude to my thesis supervisor, **Dr. K.K. Mahapatra**, Professor, Department of Electronics and communication engineering , for his guidance, support, motivation and encouragement through out the period this work was carried out. His readiness for consultation at all times, his educative comments, his concern and assistance even with practical things have been invaluable.

. I am grateful to **Prof. S.K.Patra**, Head of the Department, Electronics and Communication Engineering for providing us the necessary opportunities for the completion of our project. I also thank the other staff members of my department for their invaluable help and guidance.

Jaswant Kumar Rout (Roll No. 10507008)

B.Tech. Final Yr. EIE

CONTENTS

LIST OF FIGURES

ABSTRACT

KEYWORDS: Microprocessor, Digital circuits, Combinational circuits, Sequential circuits, Implementation technologies

Microprocessors are the heart of all “smart” devices, whether they be electronic devices or otherwise. Their smartness comes as a direct result of the decisions and controls that microprocessors make. For example, we usually do not consider a car to be an electronic device. However, it certainly has many complex, smart electronic systems, such as the anti-lock brakes and the fuel-injection system. Each of these systems is controlled by a microprocessor. Yes, even the black, hardened blob that looks like a dried-up and pressed-down piece of gum inside a musical greeting card is a microprocessor.

There are generally two types of microprocessors: **general-purpose microprocessors** and **dedicated microprocessors**. General-purpose microprocessors, such as the Pentium CPU, can perform different tasks under the control of software instructions. General-purpose microprocessors are used in all personal computers. Dedicated microprocessors, also known as **application-specific integrated circuits** (**ASIC**s), on the other hand, are designed to perform just one specific task. For example, inside your cell phone, there is a dedicated microprocessor that controls its entire operation. The embedded microprocessor inside the cell phone does nothing else but controls the operation of the phone. Dedicated microprocessors are, therefore, usually much smaller than and not as complex as general-purpose microprocessors. However, they are used in every smart electronic device, such as the musical greeting cards, electronic toys, TVs, cell phones, microwave ovens, and anti-lock break systems in your car. From this short list, I’m sure that you can think of many more devices that have a dedicated microprocessor inside them. Although the small dedicated microprocessors are not as powerful as the general-purpose microprocessors, they are being sold and used in a lot more places than the powerful general-purpose microprocessors that are used in personal computers.

Our projects aim at designing an embedded processor and implement the embedded processor for a particular application and the project is divided into following modules

1) Designing the basic component of the microprocessor such as

- ALU
- RAM
- ROM
- REGISTERS
- DATA PATH
- CONTROL PATH

2) Component design as per the fixed size.

3) Connect the data path and control path

4) Fixed the number of instruction required

5) Implementation of CAM

In order to analyze the performance issues, and testing we will implement our code in FPGA / ASIC

CHAPTER 1

LITURATURE REVIEW

1.1 INTRODUCTION

The logic circuit for the microprocessor can be divided into two parts: the **data path** and the **control unit**, as the data path is responsible for the actual execution of all data operations performed by the microprocessor, such as the addition of two numbers inside the arithmetic logic unit (ALU). The data path also includes registers for the temporary storage of your data. The functional units inside the data path, which in our example includes the ALU and the register, are connected together with multiplexers and data signal lines. The data signal lines are for transferring data between two functional units. Data signal lines in the circuit diagram are represented by lines connecting two functional units. Sometimes, several data signal lines are grouped together to form a **bus**. The width of the bus (that is, the number of data signal lines in the group) is annotated next to the bus line. In the example, the bus lines are thicker and are 8-bits wide. Multiplexers, also known as MUX, are for selecting data from two or more sources to go to one destination. In the sample circuit, a 2-to-1 multiplexer is used to select between the input data and the constant '0' to go to the left operand of the ALU. The output of the ALU is connected to the input of the register. The output of the register is connected to three different destinations: (1) the right operand of the ALU, (2) an OR gate used as a comparator for the test "not equal to 0," and (3) a tri-state buffer. The tri-state buffer is used to control the output of the data from the register.

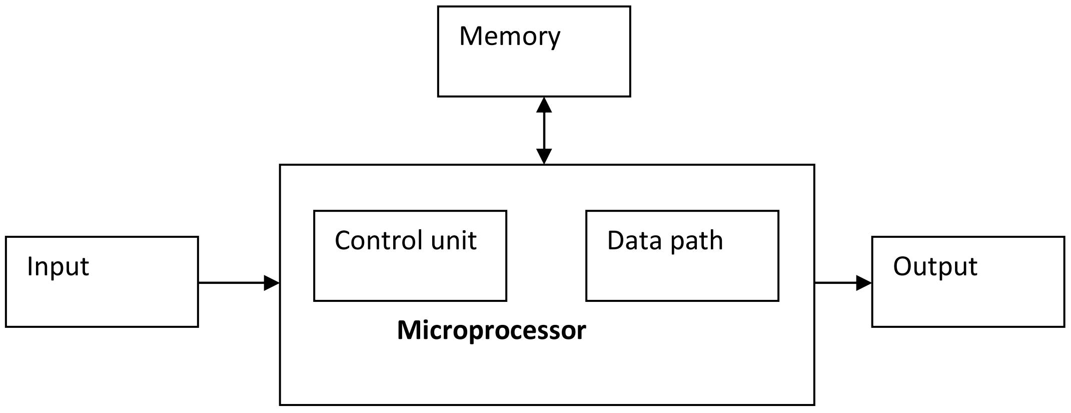


Fig: 1.1 Model of computer

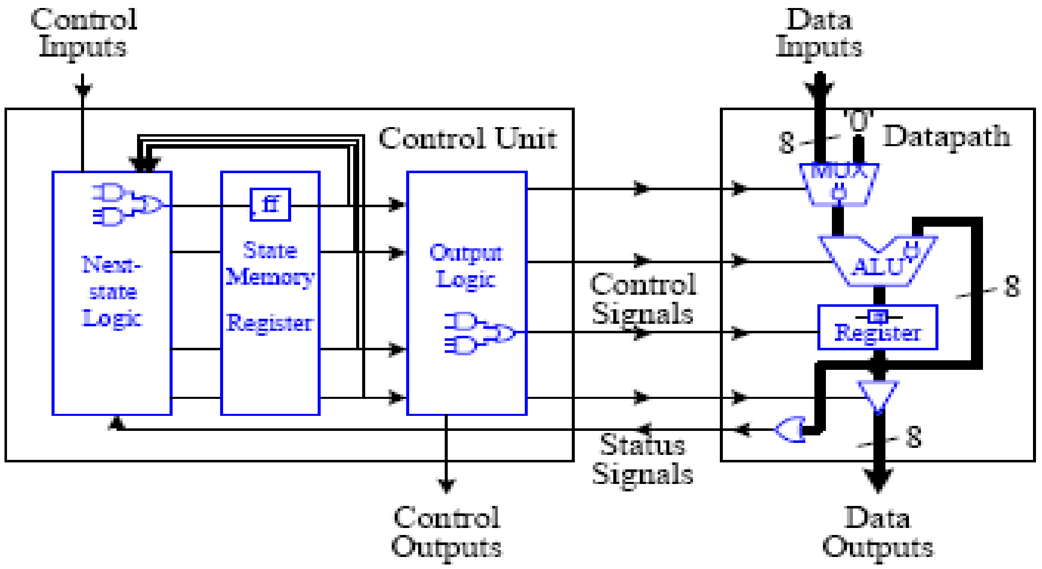


Fig: 1.2 internal parts of microprocessor

Even though the data path is capable of performing all of the data operations of the microprocessor, it cannot, however, do it on its own. In order for the data path to execute the operations automatically, the control unit is required. The control unit, also known as the controller, controls all of the operations of the data path, and therefore, the operations of the entire microprocessor. The control unit is a **finite state machine** (FSM) because it is a machine that executes by going from one state to another and that there are only a finite number of states for the machine to go to. The control unit is made up of three parts: the **next-state logic**, the **state memory**, and the **output logic**. The purpose of the state memory is to remember the current state that the FSM is in. The next-state logic is the circuit for determining what the next state should be for the machine. And the output logic is the circuit for generating the actual control signals for controlling the data path. Every digital logic circuit, regardless of whether it is part of the control unit or the data path, is categorized as either a **combinational circuit** or a **sequential circuit**. A combinational circuit is one where the output of the circuit is dependent only on the current inputs to the circuit. For example, an adder circuit is a combinational circuit.

It takes two numbers as inputs. The adder evaluates the sum of these two numbers and outputs the result. A sequential circuit, on the other hand, is dependent not only on the current inputs, but also on all the previous inputs. In other words, a sequential circuit has to remember its past history. For example, the up-channel button on a TV remote is part of a sequential circuit. Pressing the up-channel button is the input to the circuit. However, just having this input is not enough for the circuit to determine what TV channel to display next. In addition to the up channel button input, the circuit must also know the current channel that is being displayed, which is the history. If the current channel is channel 3, then pressing the up-channel button will change the channel to channel 4. Since sequential circuits are dependent on the history, they must therefore contain memory elements for remembering the history; whereas combinational circuits do not have memory elements. Examples of combinational circuits inside the microprocessor include the next-state logic and output logic in the control unit, and the ALU, multiplexers, tri-state buffers, and comparators in the data path. Examples of sequential circuits include the register for the state memory in the controller and the registers in the data path. model is also a sequential circuit.

Irregardless of whether a circuit is combinational or sequential, they are all made up of the three basic logic gates: **AND**, **OR**, and **NOT** gates. From these three basic gates, the most powerful computer can be made. Furthermore, these basic gates are built using transistors — the fundamental building blocks for all digital logic circuits. Transistors are just electronic binary switches that can be turned on or off. The on and off states of a transistor are used to represent the two binary values: 1 and 0. Figure 1.3 summarizes how the different parts and components fit together to form the microprocessor. From transistors, the basic logic gates are built. Logic gates are combined together to form either combinational circuits or sequential circuits. The difference between these two types of circuits is only in the way the logic gates are connected together. Latches and flip-flops are the simplest forms of sequential circuits, and they provide the basic building blocks for more complex sequential circuits. Certain combinational circuits and sequential circuits are used as standard building blocks for larger circuits, such as the microprocessor. These standard combinational and sequential components usually are found in standard libraries and serve as larger building blocks for the microprocessor. Different combinational components and sequential components are connected together to form either the data path or the control unit of a microprocessor. Finally, combining the data path and the control unit together will produce the circuit for either a dedicated or a general microprocessor.

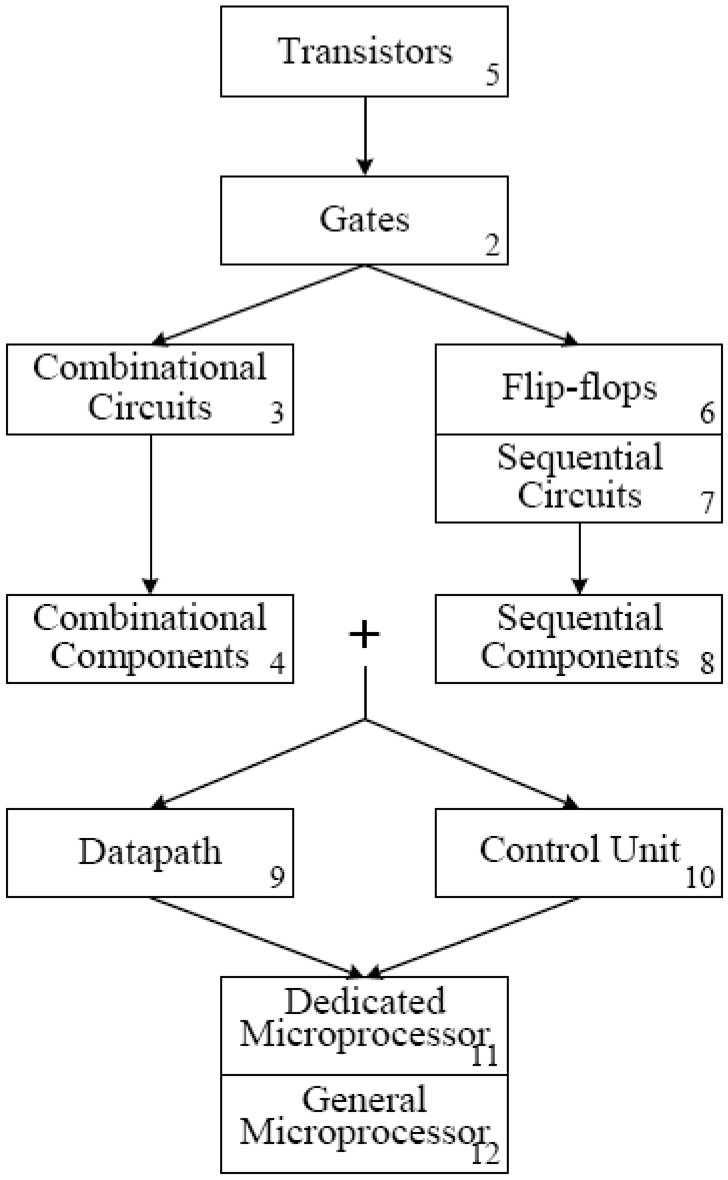


Fig 1.3 Summary of how parts of a microprocessor fit together

1.2 OBJECTIVE

The objective of the project is to design an 8-bit embedded processor which is highly efficient and reliable and implementation for content addressable memory.

1.3 ORGANISATION OF THE REPORT

The present study is about the microprocessor, introduction about the embedded system . Chapter 2 deals with the multiphase modeling of the fluidized bed reactor which includes the detailed derivation of the continuity and momentum equations . Chapter 3 describes the programming code for the simulation of the fluidized bed reactor. Chapter 4 deals with the results and discussion.

1.4 DESIGN ABSTRATION LEVEL

Digital circuits can be designed at any one of several abstraction levels. When designing a circuit at the **transistor level**, which is the lowest level, you are dealing with discrete transistors and connecting them together to form the circuit. The next level up in the abstraction is the **gate level**. At this level, you are working with logic gates to build the circuit. At the gate level, you also can specify the circuit using either a truth table or a Boolean equation. In using logic gates, a designer usually creates standard combinational and sequential components for building larger circuits. In this way, a very large circuit, such as a microprocessor, can be built in a hierarchical fashion. Design methodologies have shown that solving a problem hierarchically is always easier than trying to solve the entire problem as a whole from the ground up. These combinational and sequential components are used at the **register-transfer level** in building the data path and the control unit in the microprocessor. At the register-transfer level, we are concerned with how the data is transferred between the various registers and functional units to realize or solve the problem at hand. Finally, at the highest level, which is the **behavioral level**, we construct the circuit by describing the behavior or operation of the circuit using a hardware description language. This is very similar to writing a computer program using a programming language.

CHAPTER 2

COMPONENT DESIGN

2.1 STANDARD COMBINTIONAL COMPONENTS

2.1 (a) ALU

The **arithmetic logic unit** (**ALU**) is one of the main components inside a microprocessor. It is responsible for performing arithmetic and logic operations, such as addition, subtraction, logical AND, and logical OR. The ALU, however, is not used to perform multiplications or divisions. It turns out that, in constructing the circuit for the ALU, we can use the same idea as for constructing the adder-subtractor combination circuit, as discussed in the previous section. Again, we will use the ripple-carry adder as the building block and then insert some combinational logic circuitry in front of the two input operands to each full adder. This way, the primary inputs will be modified accordingly, depending on the operations being performed before being passed to the full adder. The general, overall circuit for an 8-bit ALU is shown in Figure 1.4(a), and its logic symbol in (b).As we can see in the figure, the two combinational circuits in front of the full adder (FA) are labeled LE and AE. The logic extender (LE) is for manipulating all logical operations; whereas, the arithmetic extender (AE) is for manipulating all arithmetic operations. The LE performs the actual logical operations on the two primary operands, *Ai* and *bi*, before passing the result to the first operand, *xi*, of the FA. On the other hand, the AE only modifies the second operand, *bi*, and passes it to the second operand, *yi*, of the FA where the actual arithmetic operation is perform

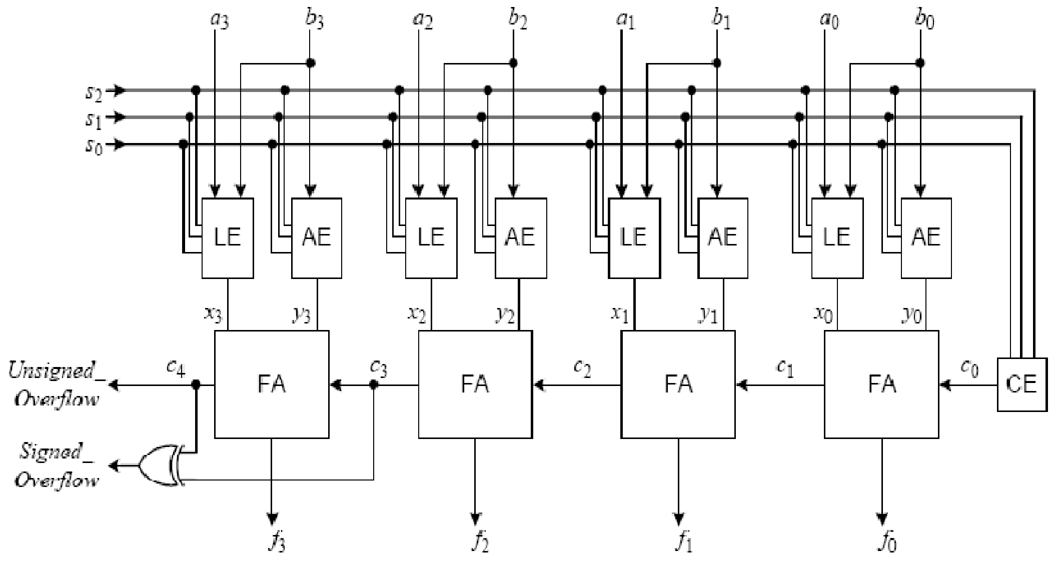


Fig: 1.4-bit ALU circuit

We saw from the adder-subtract or circuit that, to perform additions and subtractions, we only need to modify *yi* (The second operand to the FA) so that all operations can be done with additions. Thus, the AE only takes the second operand of the primary input, *bi*, as its input and modifies the value depending on the operation being performed. Its output is *yi*, and it is connected to the second operand input of the FA. As in the adder-subtract or circuit, the addition is performed in the FA. When arithmetic operations are being performed, the LE must pass the first operand unchanged from the primary input *ai* to the output *xi* for the FA. Unlike the AE (where it only modifies the operand) the LE performs the actual logical operations. Thus, for example, if we want to perform the operation *A* OR *B*, the LE for each bit slice will take the corresponding bits, *ai* and *bi*, and OR them together. Hence, one bit from both operands, *ai* and *bi*, are inputs to the LE. The output of the LE is passed to the first operand, *xi*, of the FA. Since this value is already the result of the logical operation, we do not want the FA to modify it but to simply pass it on to the primary output, *fi*. This is accomplished by setting both the second operand, *yi*, of the FA, and *c*0 to 0 since adding a 0 will not change the resulting value. The combinational circuit labeled CE (for carry extender) is for modifying the primary carry-in sign*c*0, so that arithmetic operations are performed correctly. Logical operations do not use the carry signal, so *c*0 is set to 0 for all logical operations.

s_2	s_1	s_0	Operation Name	Operation	x_i (LE)	y_i (AE)	c_0 (CE)
0	0	0	Pass(A)	Pass A to output	a_i	0	0
0	0	1	AND	A AND B	a_i AND b_i	0	0
0	1	0	OR	A OR B	a_i OR b_i	0	0
0	1	1	NOT	A'	a_i'	0	0
1	0	0	Addition	$A + B$	a_i	b_i	0
1	0	1	Subtraction	$A - B$	a_i	b_i'	1
1	1	0	Increment	$A + 1$	a_i	0	1
1	1	1	Decrement	$A - 1$	a_i	1	0

(A)

s_2	s_1	s_0	x_i
0	0	0	a_i
0	0	1	$a_i b_i$
0	1	0	$a_i - b_i$
0	1	1	a_i'
1	×	×	a_1

(B)

s_2	s_1	s_0	c_0
0	×	×	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

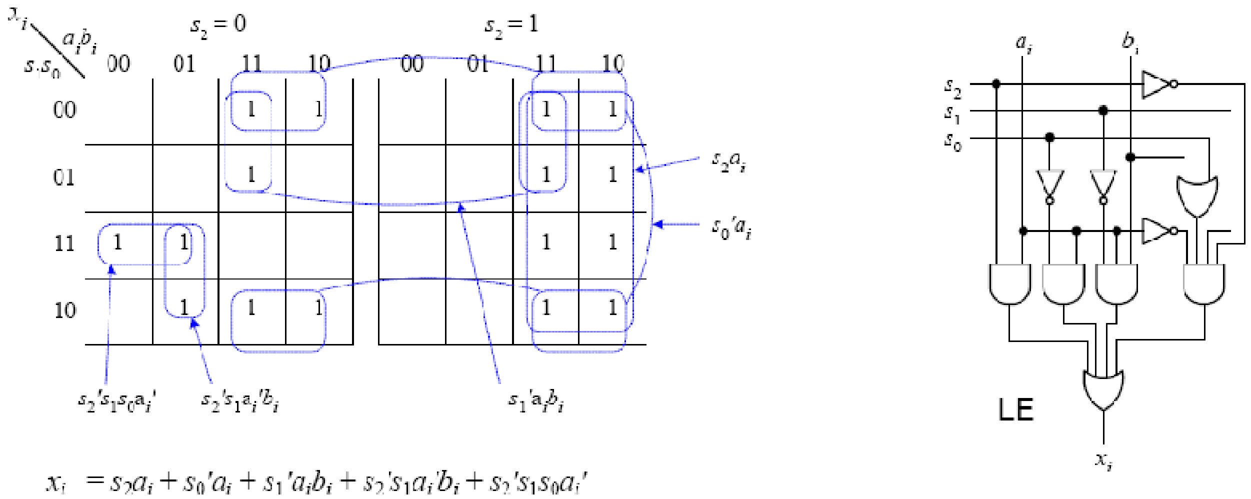
(C)

s_2	s_1	s_0	b_i	y_i
0	×	×	×	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

(D)

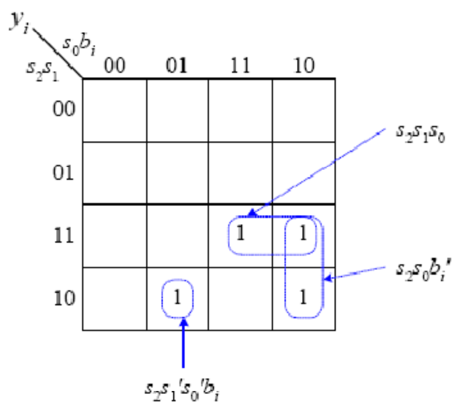
Fig:1.5 ALU operations a).functional table b).LE truth table c) AE truth table d) CE truth table

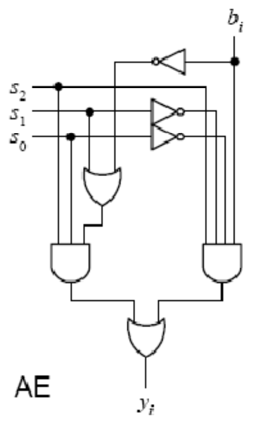
In the circuit shown in Figure, three select lines, *s*2, *s*1, and *s*0, are used to select the operations of the ALU. With these three select lines, the ALU circuit can implement up to eight different operations. Suppose that the operations that we want to implement in our ALU are as defined in Figure. The *xi* column shows the values that the LE must generate for the different operations. The *yi* column shows the values that the AE must generate. The *c*0 column shows the carry signals that the CE must generate. For example, for the pass-through operation, the value of *ai* is passed through without any modifications to *xi*. For the AND operation, *xi* gets the result of *ai* AND *bi*. As mentioned before, both *yi* and *c*0 are set to 0 for all of the logical operations, because we do not want the FA to change the results. The FA is used only to pass the results from the LE straight through to the output *F*. For the subtraction operation, instead of subtracting *B*, we want to add –*B*. Changing *B* to –*B* in two’s complement format requires flipping the bits of *B* and then adding a 1. Thus, *yi* gets the inverse of *bi*, and the 1 is added through the carry-in *c*0. To increment *A*, we set *yi* to all 0’s, and add the 1 through the carry-in *c*0. To decrement *A*, we add a –1 instead. Negative one in two’s complement format is a bit string with all 1’s. Hence, we set *yi* to all 1’s and the carry-in *c*0 to 0. For all the arithmetic operations, we need the first operand, *A*, unchanged for the FA. Thus, *xi* gets the value of *ai* for all arithmetic operations. Figure (b), (c) and (d) show the truth tables for the LE, AE, and CE respectively. The LE circuit is derived from the *xi* column of Figure (b); the AE circuit is derived from the *yi* column of Figure(c); and the CE circuit is derived from the *c*0 column of Figure (d). Notice that *xi* is dependent on five variables, *s*2, *s*1, *s*0, *ai*, and *bi*; whereas *yi* is dependent on only four variables, *s*2, *s*1, *s*0, and *bi*; and *c*0 is dependent on only the three select lines *s*2, *s*1, and *s*0. The K-maps, equations, and schematics for these three circuits are shown in Figure. The behavioral VHDL code for the ALU is shown in Figure, and a sample simulation trace for all the operations using the two inputs 5 and 3 is shown in Figure .



$$x_i = s_2a_i + s_0'a_i + s_1'a_ib_i + s_2's_1a_i'b_i + s_2's_1s_0a_i'$$
$$= s_2a_i + s_0'a_i + s_1'a_ib_i + s_2's_1a_i'(b_i + s_0)$$

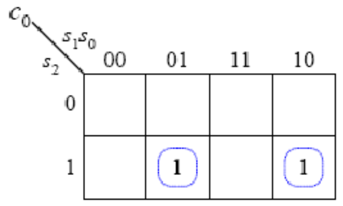
(a)

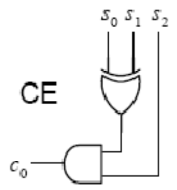




$$
\begin{aligned}
y_i &= s_2s_1s_0 + s_2s_0b_i' + s_2s_1's_0'b_i \\
&= s_2s_0(s_1 + b_i') + s_2s_1's_0'b_i
\end{aligned}
$$

(b)





$$
\begin{aligned}
c_0 &= s_2s_1's_0 + s_2s_1s_0' \\
&= s_2(s_1 \oplus s_0)
\end{aligned}
$$

(c)

Fig: 1.6 k- map equations and schematics for a) L.E b) A.E c) C.E

Behavioral VHDL code for an ALU

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
-- The following package is needed so that the STD_LOGIC_VECTOR signals
-- A and B can be used in unsigned arithmetic operations.
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY  alu IS PORT (
S: IN STD_LOGIC_VECTOR(2 DOWNTO 0); -- select for operations
A, B: IN STD_LOGIC_VECTOR(7 DOWNTO 0); -- input operands
F: OUT STD_LOGIC_VECTOR(7 DOWNTO 0)); -- output
END  alu;
ARCHITECTURE Behavior OF alu IS
BEGIN
PROCESS(S, A, B)
BEGIN
```

```
CASE  S  IS
WHEN "000" => -- pass A through
F <= A;
WHEN "001" => -- AND
F <= A AND B;
WHEN "010" => -- OR
F <= A OR B;
WHEN "011" => -- NOT A
F <= NOT A;
WHEN "100" => -- add
F <= A + B;
WHEN "101" => -- subtract
F <= A - B;
WHEN "110" => -- increment
F <= A + 1;
WHEN OTHERS => -- decrement
F <= A - 1;
END CASE;
END PROCESS;
END Behavioral;
```

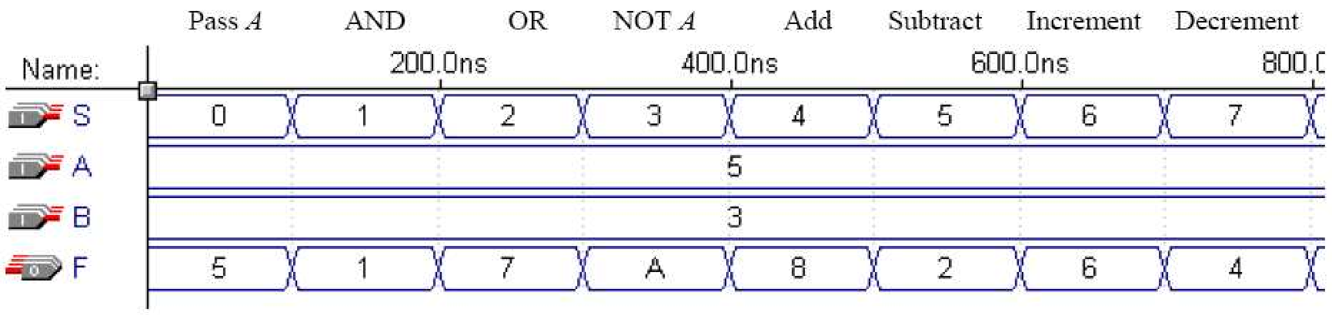


Fig: 1.7 Sample simulation trace with the two input operands A and B for all of the eight operations.

2.1(B) <u>MULTIPLEXER</u>

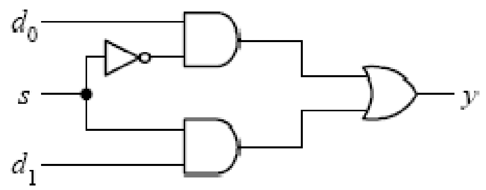
The **multiplexer**, or **MUX** for short, allows the selection of one input signal among *n* signals, where *n* > 1, and is a power of two. Select lines connected to the multiplexer determine which input signal is selected and passed to the output of the multiplexer. In general, an *n*-to-1 multiplexer has *n* data input lines, *m* select lines where *m* = log2 *n*, i.e. 2*m* = *n*, and one output line. For a 2-to-1 multiplexer, there is one select line, *s*, to select between the two inputs, *d*0 and *d*1. When *s* = 0, the input line *d*0 is selected, and the data present on *d*0 is passed to the output *y*. When *s* = 1, the input line *d*1 is selected and the data on *d*1 is passed to *y*. The truth table, equation, circuit, and logic symbol for a 2-to-1 multiplexer are shown in Figure 4.20. Constructing a larger size multiplexer, such as the 8-to-1 multiplexer, can be done similarly. In addition to having the eight data input lines, *d*0 to *d*7, the 8-to-1 multiplexer has three (23 = 8) select lines, *s*0, *s*1, and *s*2. Depending on the value of the three select lines, one of the eight input lines will be selected and the data on that input line will be passed to the output. For example, if the value of the select lines is 101, then the input line *d*5 is selected, and the data that is present on *d*5 will be passed to the output

s	d_1	d_0	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

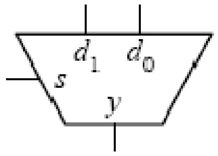
(a)

$$\begin{aligned} y &= s'd_1'd_0 + s'd_1d_0 + sd_1d_0' + sd_1d_0 \\ &= s'd_0(d_1' + d_1) + sd_1(d_0' + d_0) \\ &= s'd_0 + sd_1 \end{aligned}$$

(b)



(b)



(c)

Fig: 1.8 multiplexer state table

.

The truth table, circuit, and logic symbol for the 8-to-1 multiplexer are shown in Figure 1.8. The truth table is written in a slightly different format. Instead of including the *d*'s in the input columns and enumerating all 211 = 2048 rows (the eleven variables come from the eight *d*'s and the three *s*'s), the *d*'s are written in the entry under the output column. For example, when the select line value is 101, the entry under the output column is *d*5, which means that *y* takes on the value of the input line *d*5. To understand the circuit in Figure 1.8(b), notice that each AND gate acts as a switch and is turned on by one combination of the three select lines. When a particular AND gate is turned on, the data at the corresponding *d* input is passed through that AND gate. The outputs of the remaining AND gates are all 0's

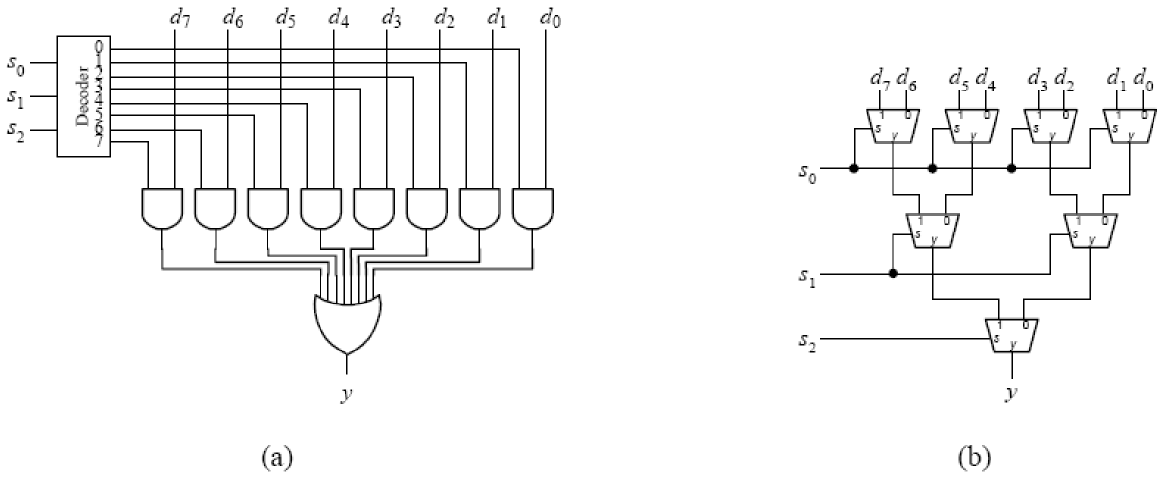


Fig: 1.9 An 8-to-1 multiplexer implemented using: (a) a 3-to-8 decoder; (b) seven 2-to-1 multiplexers

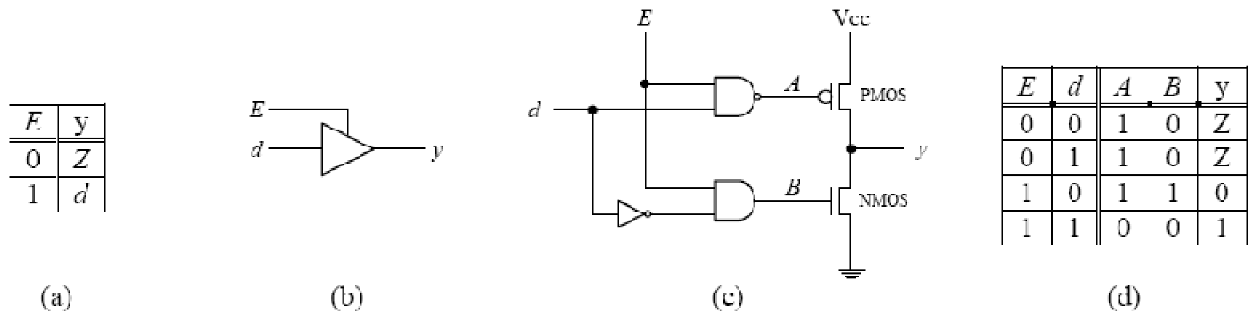
<u>Behavioral level VHDL code for an 8-bit wide 4-to-1 multiplexer</u>

```
-- A 4-to-1 8-bit wide multiplexer
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY Multiplexer IS
PORT(S: IN STD_LOGIC_VECTOR (1 DOWNTO 0); -- select lines
D0, D1, D2, D3: IN STD_LOGIC_VECTOR (7 DOWNTO 0); -- data bus input
Y: OUT STD_LOGIC_VECTOR (7 DOWNTO 0)); -- data bus output
END Multiplexer;
-- Behavioral level code
ARCHITECTURE Behavioral OF Multiplexer IS
```

```
BEGIN
PROCESS (S, D0, D1, D2, D3)
BEGIN
CASE S IS
WHEN "00" => Y <= D0;
WHEN "01" => Y <= D1;
WHEN "10" => Y <= D2;
WHEN "11" => Y <= D3;
WHEN OTHERS => Y <= (OTHERS => 'U');
END CASE;
END PROCESS;
END Behavioral;
```

2.1(C) TRI-STATE BUFFER

A **tri-state** buffer, as the name suggests, has three states: 0, 1, and a third state denoted by *Z*. The value *Z* represents a high-impedance state, which for all practical purposes acts like a switch that is opened or a wire that is cut. Tri-state buffers are used to connect several devices to the same bus. A *bus* is one or more wire for transferring signals. If two or more devices are connected directly to a bus without using tri-state buffers, signals will get corrupted on the bus because the devices are always outputting either a 0 or a 1. However, with a tri-state buffer in between, devices that are not using the bus can disable the tri-state buffer so that it acts as if those devices are physically disconnected from the bus. At any one time, only one active device will have its tri-state buffers enabled, and thus, use the bus. The truth table and symbol for the tri-state buffer is shown in Figure 1.10(a) and (b). The active high enable line *E* turns the buffer on or off. When *E* is de-asserted with a 0, the tri-state buffer is disabled, and the output *y* is in its high-impedance *Z* state. When *E* is asserted with a 1, the buffer is enabled, and the output *y* follows the input *d*. A circuit consisting of only logic gates cannot produce the high impedance state required by the tri-state buffer, since logic gates can only output a 0 or a 1. To provide the high impedance state, the tri-state buffer circuit uses two transistors in conjunction with logic gates, as shown in Figure 1.10(c). Section 5.3 will discuss the operations of these two transistors in detail. For now, we will keep it simple. The top PMOS transistor is enabled with a 0 at the node labeled *A*, and when it is enabled, a 1 signal from Vcc passes down through the transistor to *y*. The bottom NMOS transistor is enabled with a 1 at the node labeled *B*, and when it is enabled, a 0 signal from ground passes up through the transistor to *y*. When the two transistors are disabled (with *A* = 1 and *B* = 0) they will both output a high impedance *Z* value; so *y* will have a *Z* value. Having the two transistors, we need a circuit that will control these two transistors so that together they realize the tri-state buffer function. The truth table for this control circuit is shown in Figure 4.25(d). The truth table is derived as follows. When *E* = 0, it does not matter what the input *d* is, we want both transistors to be disabled so that the output *y* has the *Z* value. The PMOS transistor is disabled when the input *A* = 1; whereas, the NMOS transistor is disabled when the input *B* = 0. When *E* = 1 and *d* = 0, we want the output *y* to be a 0. To get a 0 on *y*, we need to enable the bottom NMOS transistor and disable the top PMOS transistor so that a 0 will pass through the NMOS transistor to *y*. To get a 1 on *y* for when *E* = 1 and *d* = 1, we need to do the reverse by enabling the top PMOS transistor and disabling the bottom NMOS transistor. The resulting circuit is shown in Figure 1.10(c). When *E* = 0, the output of the NAND gate is a 1, regardless of what the other input is, and so the top PMOS transistor is turned off. Similarly, the output of the AND gate is a 0, so the bottom NMOS transistor is also turned off. Thus, when *E* = 0, both transistors are off, so the output *y* is in the *Z* state. When *E* = 1, the outputs of both the NAND and AND gates are equal to *d'*. So if *d* = 0, the output of the two gates are both 1, so the bottom transistor is turned on while the top transistor is turned off. Thus, *y* will have the value 0, which is equal to *d*. On the other hand, if *d* = 1, the top transistor is turned on while the bottom transistor is turned off, and *y* will have the value 1.



E	y
0	*Z*
1	*d*

E	*d*	*A*	*B*	y
0	0	1	0	Z
0	1	1	0	Z
1	0	1	1	0
1	1	0	0	1

Tri-state buffer: Fig 1.10 (a) truth table; (b) logic symbol; (c) circuit; (d) truth table for the control portion of the tri-state buffer circuit.

VHDL code for an 8-bit wide tri-state buffer

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY TriState_Buffer IS PORT (
E: IN STD_LOGIC;
d: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
y: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END TriState_Buffer;
ARCHITECTURE Behavioral OF TriState_Buffer IS
BEGIN
PROCESS (E, d)
BEGIN
IF (E = '1') THEN
y <= d;
ELSE
y <= (OTHERS => 'Z'); -- to get 8 Z values
END IF;
END PROCESS;
END Behavioral;
```

1.3 (D) SHIFTER

The **shifter** is used for shifting bits in a binary word one position either to the left or to the right. The operations for the shifter are referred to either as **shifting** or **rotating**, depending on how the end bits are shifted in or out. For a shift operation, the two end bits do not wrap around; whereas for a rotate operation, the two end bits wrap around. Figure 1.11 shows six different shift and rotate operations. For example, for the “Shift left with 0” operation, all the bits are shifted one position to the left. The original leftmost bit is shifted out (i.e. discarded) and the rightmost bit is filled with a 0. For the “Rotate left” operation, all the bits are shifted one position to the left. However, instead of discarding the leftmost bit, it is shifted in as the rightmost bit (i.e. it rotates around).For each bit position, a multiplexer is used to move a bit from either the left or right to the current bit position. The size of the multiplexer will determine the number of operations that can be implemented. For example, we can use a 4-to-1 multiplexer to implement the four operations, as specified by the table in Figure 1.11(a). Two select lines, *s*1 and *s*0, are needed to select between the four different operations. For a 4-bit operand, we will need to use four 4-to-1 multiplexers as shown in Figure 4.31(b). How the inputs to the multiplexers are connected will depend on the given operations.

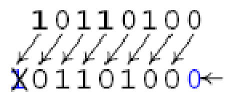
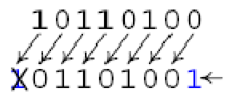
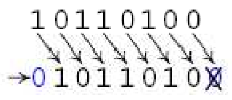
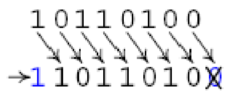
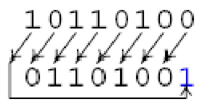
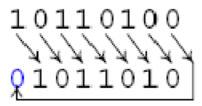
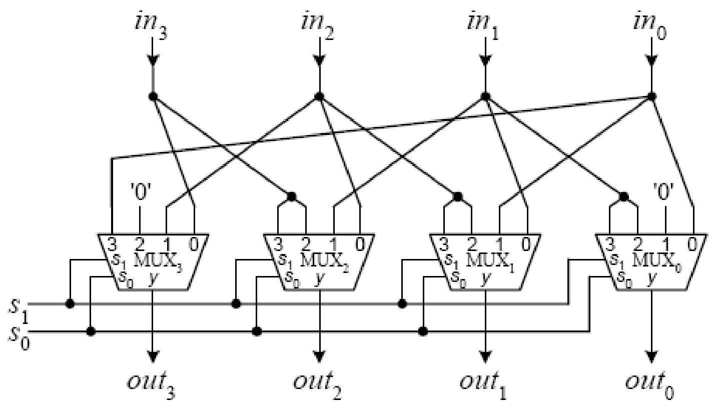
Operation	Comment	Example
Shift left with 0	Shift bits to the left one position. The leftmost bit is discarded and the rightmost bit is filled with a 0.	
Shift left with 1	Same as above, except that the rightmost bit is filled with a 1.	
Shift right with 0	Shift bits to the right one position. The rightmost bit is discarded and the leftmost bit is filled with a 0.	
Shift right with 1	Same as above, except that the leftmost bit is filled with a 1.	
Rotate left	Shift bits to the left one position. The leftmost bit is moved to the rightmost bit position.	
Rotate right	Shift bits to the right one position. The rightmost bit is moved to the leftmost bit position.	

Fig: 1.11 Shifter and rotator operations

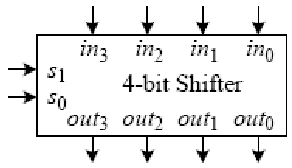
In this example, when *s*1 = *s*0 = 0, we want to pass the bit straight through without shifting, i.e. we want the value for *ini* to pass to *outi*. Given *s*1 = *s*0 = 0, *d*0 of the multiplexer is selected, hence, *ini* is connected to *d*0 of mux*i* which outputs to *outi*. For *s*1 = 0 and *s*0 = 1, we want to shift left, i.e. we want the value for *ini* to pass to *outi*+1. With *s*1 = 0 and *s*0 = 1, *d*1 of the multiplexer is selected, hence, *ini* is connected to *d*1 of MUX*i*+1 which outputs to *outi*+1. For this selection, we also want to shift in a 0 bit, so *d*1 of MUX0 is connected directly to a 0.

s_1	s_0	Operation
0	0	Pass through
0	1	Shift left and fill with 0
1	0	Shift right and fill with 0
1	1	Rotate right

(a)



(b)



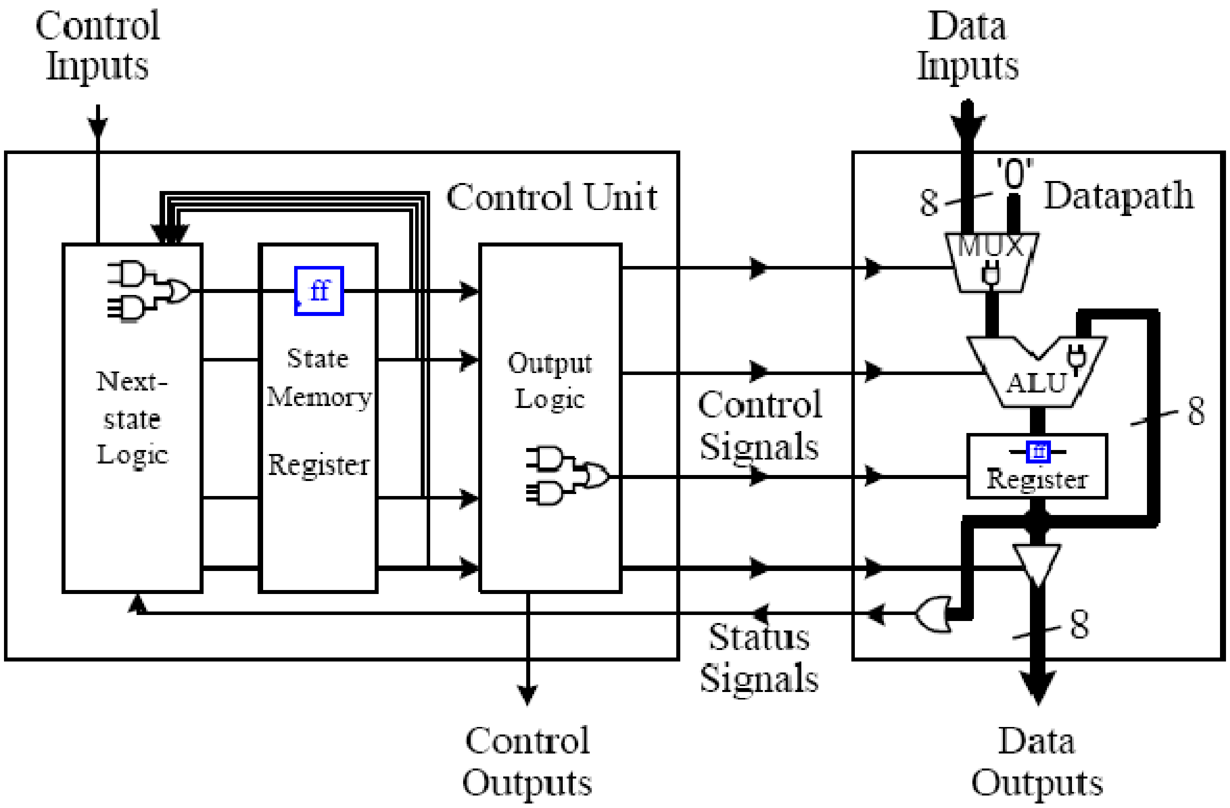
(c)

Fig 1.12 shifter a)operation table b)circuit c)logic symbol

Behavioral VHDL code for an 8-bit shifter

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY shifter IS PORT (
S: IN STD_LOGIC_VECTOR(1 DOWNTO 0); -- select for operations
input: IN STD_LOGIC_VECTOR(7 DOWNTO 0); -- input
output: OUT STD_LOGIC_VECTOR(7 DOWNTO 0)); -- output
END shifter;
ARCHITECTURE Behavior OF shifter IS
BEGIN
PROCESS(S, input)
BEGIN
CASE S IS
WHEN "00" => -- pass through
output <= input;
WHEN "01" => -- shift left with 0
output <= input(6 downto 0) & '0';
WHEN "10" => -- shift right with 0
output <= '0' & input(7 downto 1);
WHEN OTHERS => -- rotate right
output <= input(0) & input(7
DOWNTO 1);
END CASE;
END PROCESS;
END Behavior;
```

2.2 STANDARD SEQUENTIAL COMPONENTS



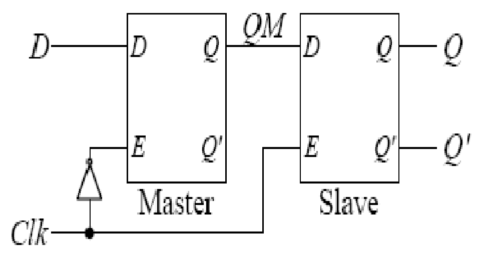
In order to remember this history of inputs, sequential circuits must have memory elements. Memory elements, however, are just like combinational circuits in the sense that they are made up of the same basic logic gates. What makes them different is in the way these logic gates are connected together. In order for a circuit to "remember" its current value, we have to connect the output of a logic gate directly or indirectly back to the input of that same gate. We call this a **feedback loop** circuit, and it forms the basis for all memory elements. Combinational circuits do not have any feedback loops. **Latches** and **flip-flops** are the basic memory elements for storing information. Hence, they are the fundamental building blocks for all sequential circuits. A single latch or flip-flop can store only one bit of information. This bit of information that is stored in a

latch or flip-flop is referred to as the **state** of the latch or flip-flop. Hence, a single latch or flip-flop can be in either one of two states: 0 or 1. We say that a latch or a flip-flop changes state when its content changes from a 0 to a 1 or vice versa. This state value is always available at the output. Consequently, the content of a latch or a flip-flop is the state value, and is always equal to its output value .The main difference between a latch and a flip-flop is that for a latch, its state or output is constantly affected by its input as long as its enable signal is asserted. In other words, when a latch is enabled, its state changes immediately when its input changes. When a latch is disabled, its state remains constant, thereby, remembering its previous value. On the other hand, a flip-flop changes state only at the active edge of its enable signal, i.e., at
precisely the moment when either its enable signal rises from a 0 to a 1 (referred to as the rising edge of the signal), or from a 1 to a 0 (the falling edge). However, after the rising or falling edge of the enable signal, and during the time when the enable signal is at a constant 1 or 0, the flip-flop's state remains constant even if the input changes. In a microprocessor system, we usually want changes to occur at precisely the same moment. Hence, flip-flops are used more often than latches, since they can all be synchronized to change only at the active edge of the enable signal. This enable signal for the flip-flops is usually the global controlling clock signal.
Historically, there are basically four main types of flip-flops: SR, D, JK, and T. The major differences between them are the number of inputs they have and how their contents change. Any given sequential circuit can be built using any of these types of flip-flops (or combinations of them). However, selecting one type of flip-flop over another type to use in a particular sequential circuit can affect the overall size of the circuit. Today, sequential circuits are designed mainly with D flip-flops because of their ease of use. This is simply a tradeoff issue between ease of circuit design versus circuit size. Thus, we will focus mainly on the D flip-flop.

2.1 (A)D FLIP-FLOP

Unlike the latch, a flip-flop is not level-sensitive, but rather **edge-triggered**. In other words, data gets stored into a flip-flop only at the active edge of the clock. An **edge-triggered D flip-flop** achieves this by combining in series a pair of D latches. Figure 6.10(a) shows a **positive-edge-**

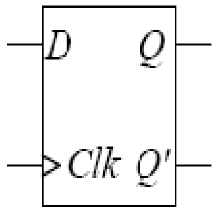
triggered D flip-flop, where two D latches are connected in series. A clock signal *Clk* Is connected to the *E* input of the two latches: one directly, and one through an inverter. The first latch is called the *master* latch. The master latch is enabled when *Clk* = 0 because of the inverter, and so *QM* follows the primary input *D*. However, the signal at *QM* cannot pass over to the primary output *Q*, because the second latch (called the *slave* latch) is disabled when *Clk* = 0. When *Clk* = 1, the master latch is disabled, but the slave latch is enabled so that the output from the master latch, *QM*, is transferred to the primary output *Q*. The slave latch is enabled all the while that *Clk* = 1, but its content changes only at the rising edge of the clock, because once *Clk* is 1, the master latch is disabled, and the input to the slave latch, *QM*, will be constant. Therefore, when *Clk* = 1 and the slave latch is enabled, the primary output *Q* will not change because the input *QM* is not changing. The circuit shown in Figure 6.10(a) is called a positive-edge-triggered D flip-flop because the primary output *Q* on the slave latch changes only at the rising edge of the clock. If the slave latch is enabled when the clock is low (i.e., with the inverter output connected to the *E* of the slave latch), then it is referred to as a **negative-edge triggered** flip-flop. The circuit is also referred to as a **master-slave** D flip-flop because of the two D latches used in the circuit. Figure 6.10(b) shows the operation table for the D flip-flop. The _ symbol signifies the rising edge of the clock. When *Clk* is either at 0 or 1, the flip-flop retains its current value (i.e., *Qnext* = *Q*). *Qnext* changes and follows the primary input *D* only at the rising edge of the clock. The logic symbol for the positive-edge-triggered D flip-flop is shown in Figure 6.10(c). The small triangle at the clock input indicates that the circuit is triggered by the edge of the signal, and so it is a flip-flop. Without the small triangle, the symbol would be that for a latch. If there is a circle in front of the clock line, then the flip-flop is triggered by the falling edge of the clock, making it a negative-edgetriggered flip-flop. Figure 6.10(d) shows a sample trace for the D flip-flop. Notice that when *Clk* = 0, *QM* follows *D*, and the output of the slave latch, *Q*, remains constant. On the other hand, when *Clk* = 1, *Q* follows *QM*, and the output of the master latch, *QM*, remains constant.



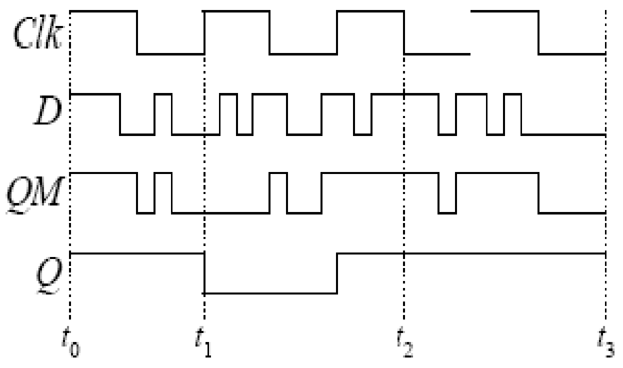
(a)

Clk	D	Q	Q_{next}	Q_{next}'
0	×	0	0	1
0	×	1	1	0
1	×	0	0	1
1	×	1	1	0
↑	0	×	0	1
↑	1	×	1	0

(b)



(c)



(d)

Fig: 1.13 Master-slave positive-edge-triggered D flip-flop: (a) circuit using D latches; (b) operation table; (c) logic symbol; (d) sample trace.

<u>VHDL Code for a D Latch with Enable</u>

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY D_latch_with_enable IS PORT (
D, Enable: IN STD_LOGIC;
Q: OUT STD_LOGIC);
```

```
END D_latch_with_enable;
ARCHITECTURE Behavior OF D_latch_with_enable IS
BEGIN
PROCESS(D, Enable)
BEGIN
IF Enable = '1' THEN
Q <= D;
END IF;
END PROCESS;
END Behavior;
```

<u>VHDL code for a positive-edge-triggered D flip-flop</u>

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY D_flipflop IS PORT (
D, Clock: IN STD_LOGIC;
Q: OUT STD_LOGIC);
END D_flipflop;
ARCHITECTURE Behavior OF D_flipflop IS
BEGIN
PROCESS(Clock) -- sensitivity list is used
BEGIN
IF Clock'EVENT AND Clock = '1' THEN
Q <= D;
END IF;
END PROCESS;
END Behavior;
```

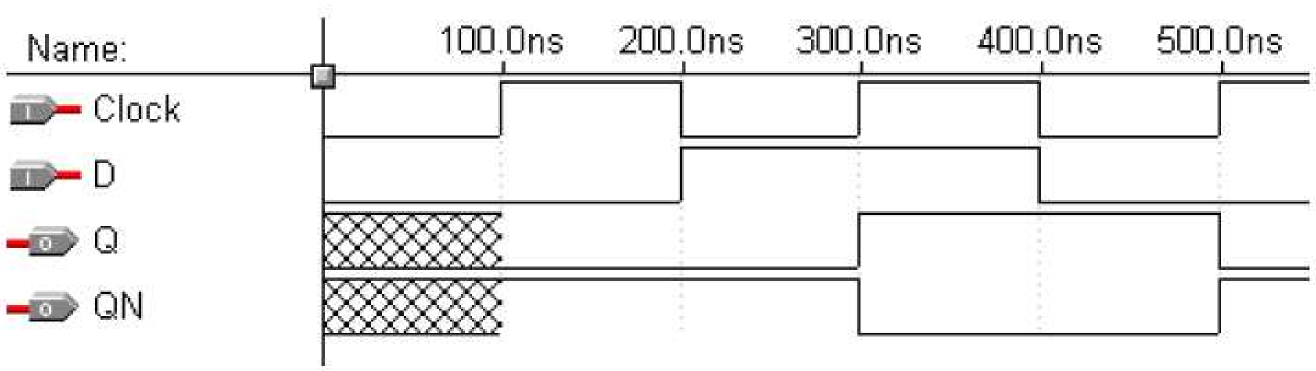


Fig: 1.14 Simulation trace for the positive-edge-triggered D flip-flop.

2.4 <u>REGISTER FILES</u>

When we want to store several numbers concurrently in a digital circuit, we can use several individual registers in the circuit. However, there are times when we want to treat these registers as a unit, similar to addressing the individual locations of an array or memory. So, instead of having several individual registers, we want to have an array of registers. This array of registers is known as a **register file**. In a register file, all of the respective control signals for the individual registers are connected in common. Furthermore, all of the respective data input and output lines for all of the registers are also connected in common. For example, the *Load* lines for all of the registers are connected together, and all of the *d*3 data lines for all of the registers are connected together. So the register file has only one set of input lines and one set of output lines for all of the registers. In addition, address lines are used to specify which register in the register file is to be accessed. In a microprocessor circuit requiring an ALU, the register file usually is used for the source operands of the ALU. Since the ALU usually takes two input operands, we like the register file to be able to output two values from possibly two different locations of the register file at the same time. So, a typical register file will have one write port and two read ports. All three ports will have their own enable and address lines. When the read enable line is de-asserted, the read port will output a 0. On the other hand, when the read enable line is asserted, the content of the register specified by the read address lines is passed to the output port. The write enable line is used to load a value into the register specified by the write address lines. The logic symbol for a 4 × 8 register file (four registers, each being 8-bits wide) is shown in Figure 1.13. The 8-bit write port is labeled *In*, and the two 8-bit read ports are labeled *Port A* and *Port B*. *WE* is the active-high write enable line. To write a value into the register file, this line must be asserted. The *WA*1 and *WA*0 are the two address lines for selecting the write location. Since there are four locations in this register file, two address lines are needed. The *RAE* line is the read enable line for *Port A*. The two read address select lines for *Port A* are *RAA*1 and *RAA*0. For *Port B*, we have the *Port B* enable line, *RBE*, and the two address lines, *RBA*1 and *RBA*0.

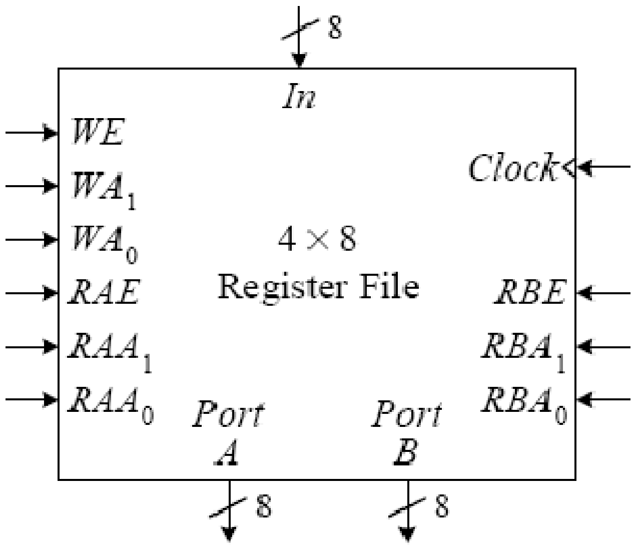


Fig: 1.15 Logic symbol for a 4 × 8 register file

. The register circuit from Figure 8.1 does not have any control for the reading of the data to the output port. In order to control the output of data, we can use a 2-input AND gate to enable or disable each of the data output lines, *Qi*. We want to control all the data output lines together, therefore, one input from all of the 2-input AND gates are connected in common. When this common input is set to a 0, all the AND gates will output a 0. When this common input is set to a 1, the output for all of the AND gates will be the value from the other input. An alternative to using AND gates to control the read ports is to use tri-state buffers. Instead of outputting a 0 when disabled, the tri-state buffers will have a high impedance .Our register file has two read ports, that is, two output controls for each register. So, instead of having just one 2-input AND gate per output line, *Qi*, we need to connect two AND gates to each output line: one for *Port A*, and one for *Port B*. An 8-bit wide register file cell circuit will have eight AND gates for *Port A*, and another eight AND gates for *Port B*, as shown in Figure 1.14. *AE* and *BE* are the read enable signals for *Port A* and *Port B*, respectively. For each read port, the read enable signal is connected in common to one input of all of the eight AND gates. The second input from each of the eight AND gates connects to the eight output lines, *Q*0 to *Q*7.For a 4X 8 register file, we need to use four 8-bit register file cells. In order to select which register file cell we want to access, three decoders are used to decode the addresses, *WA*1, *WA*0, *RAA*1, *RAA*0, *RBA*1, and *RBA*0. One decoder is used for the write addresses, *WA*1 and *WA*0; one for the *Port A* read addresses, *RAA*1 and *RAA*0; and one for the *Port B* read addresses, *RBA*1 and *RBA*0. The decoders' outputs are used to assert the individual register file cell's write line, *Load*, and read enable lines, *AE*, and *BE*. The complete circuit for the 4 X 8 register file is shown in Figure 1.14 The respective read ports from each register file cell are connected to the external read port through a 4- input 8-bit OR gate.

In terms of the timing issues, the data on the read ports are available immediately after the read enable line is asserted, whereas, the write occurs at the next active (rising) edge of the clock. Because of this, the same register can be accessed for both reading and writing at the same time; that is, the read and write enable lines can be asserted at the same time using the same read and write address. When this happens, then the value that is currently in the register is read through the read port, and a new value will be written into the register at the next rising clock edge. This timing is shown in Figure 1.14. The important point to remember is that, when the read and write operations are performed at the same time on the same register, the read operation always reads the current value stored in the register and never the new value that is to be written in by the write operation. The new value written in is available only after the next rising clock edge.

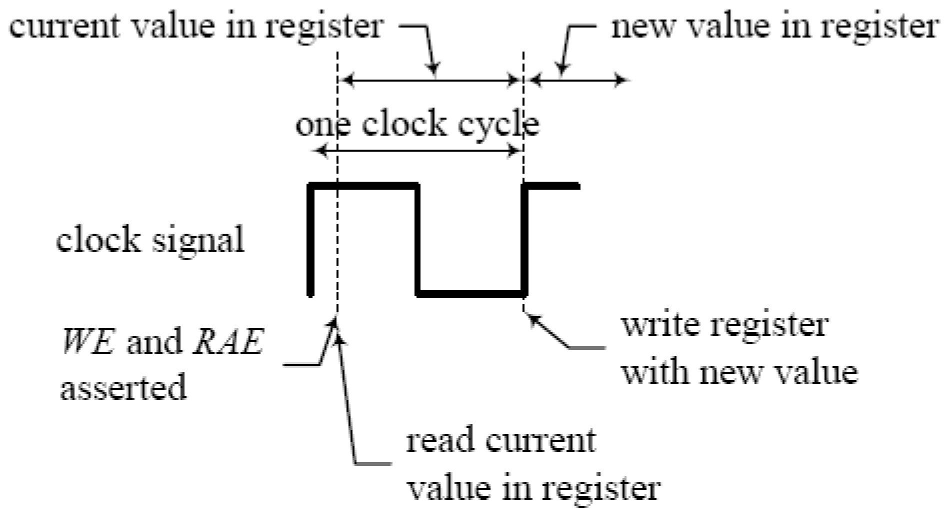

current value in register
new value in register
one clock cycle
clock signal
WE and RAE asserted
write register with new value
read current value in register

Fig: 1.16 Read and write timings for a register file cell

. The main code is composed of three processes: the write process and the two read port processes. These three processes are similar to three concurrent statements in that they are executed in parallel. The write process is sensitive to the clock, and because of the IF *clock* statement in the process, a write occurs only at the rising edge of the clock signal. The two read port processes are not sensitive to the clock but only to the read enable and read address signals. So the read data is available immediately when these lines are asserted. The function CONV_INTEGER(WA) converts the STD_LOGIC_VECTOR *WA* to an integer so that the address can be used as an index into the *RF* array.

A sample simulation trace is shown in Figure 1.15. In the simulation trace, both the write address, *WA*, and *Port A* read address, *RAA*, are set to Register 3. At 0 ns, the input data, *D*, is 5. With write enable, *WE*, asserted, the data 5 is stored into RF(3) at the next rising edge of the clock, which happens at 100 ns. When *RAE* is asserted at 200 ns, the data 5 from RF(3) is available on *Port A* immediately. At 400 ns, both *WE* and *RAE* are asserted at the same time. The current data 5 from RF(3) appears immediately on *Port A*. However, the new data 7 is written into RF(3) at 500 ns, the next rising clock edge. The new data 7 is available on *Port A* only after time 500 ns.

VHDL code for a 4 × 8 register file with one write port and two read ports

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL; -- needed for CONV_INTEGER()
ENTITY regfile IS port(
clock: IN std_logic; --clock
WE: IN std_logic; --write enable
WA: IN std_logic_vector(1 DOWNTO 0); --write address
D: IN std_logic_vector(7 DOWNTO 0); --input
RAE, RBE: IN std_logic; --read enable ports A & B
RAA, RBA: IN std_logic_vector(1 DOWNTO 0); --read address port A & B
PortA, PortB: OUT std_logic_vector(7 DOWNTO 0));--output port A & B
END regfile;
ARCHITECTURE Behavioral OF regfile IS
SUBTYPE reg IS std_logic_vector(7 DOWNTO 0);
TYPE regArray IS array(0 to 3) OF reg;
```

```
ELSE
PortB <= (others => '0');
END IF;
END PROCESS;
END Behavioral;
```

```
SIGNAL RF: regArray; --register file contents
BEGIN
WritePort: PROCESS (clock)
BEGIN
IF (clock'EVENT AND clock='1') THEN
IF (WE = '1') THEN
RF(CONV_INTEGER(WA)) <= D; -- fn to convert from vector to integer
END IF;
END IF;
END PROCESS;
ReadPortA: PROCESS (RAA, RAE)
BEGIN
-- Read Port A
IF (RAE = '1') THEN
PortA <= RF(CONV_INTEGER(RAA)); -- fn to convert from vector to integer
ELSE
PortA <= (others => '0');
END IF;
END PROCESS;
ReadPortB: PROCESS (RBE, RBA)
BEGIN
-- Read Port B
IF (RBE = '1') THEN
PortB <= RF(CONV_INTEGER(RBA)); -- fn to convert from vector to integer
```

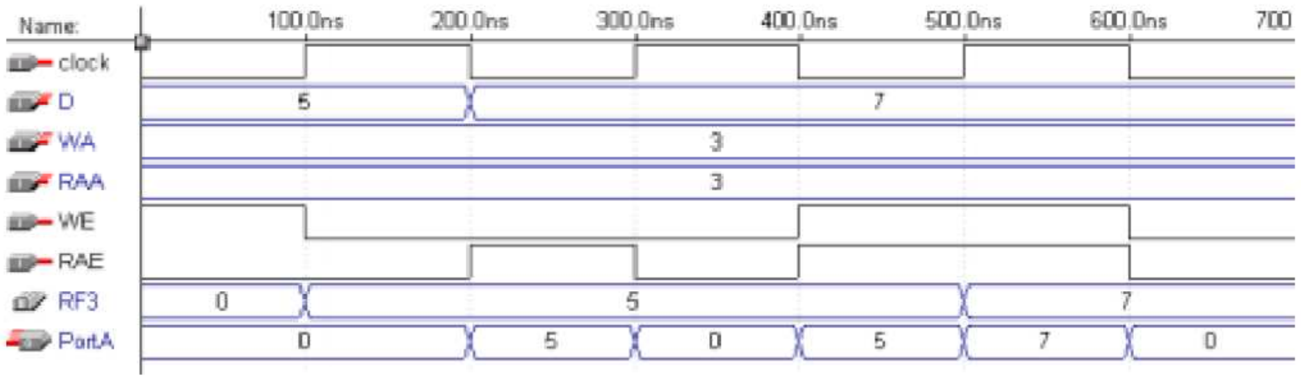
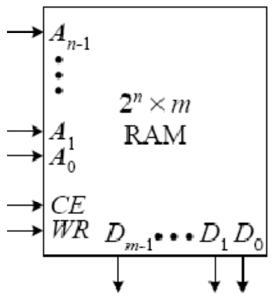


Fig: 1.17 Sample simulation trace for the 4 × 8 register file.

2.5 STATIC RANDOM ACCESS MEMORY

Another main component in a computer system is memory. This can refer to either random access memory (RAM) or read-only memory (ROM). We can make memory the same way we make the register file but with more storage locations. However, there are several reasons why we don’t want to. One reason is that we usually want a lot of memory and we want it very cheap, so we need to make each memory cell as small as possible. Another reason is that we want to use a common data bus for both reading data from, and writing data to the memory. This implies that the memory circuit should have just one data port (and not two or three like the register file) for both reading and writing of data. The logic symbol, showing all of the connections for a typical RAM chip is shown in Figure .There is a set of data lines, *Di*, and a set of address lines, *Ai*. The data lines serve for both input and output of the data to the location that is specified by the address lines. The number of data lines is dependent on how many bits are used for storing data in each memory location. The number of address lines is dependent on how many locations are in the memory chip. For example, a 512-byte memory chip will have eight data lines (8 bits = 1 byte) and nine address lines (29 = 512). In addition to the data and address lines, there are usually two control lines: chip enable (*CE*), and write enable (*WR*). In order for a microprocessor to access memory, either with the read operation or the write operation, the active-high *CE* line must first be asserted. Asserting the *CE* line enables the entire memory chip. The active-high *WR* line selects which of the two memory operations is to be performed. Setting *WR* to a 0 selects the read operation, and data from the memory is retrieved. Setting *WR* to a 1 selects the write operation, and data from the microprocessor is written into the memory. Instead of having just the *WR* line for selecting the two operations, read and write, some memory chips have both a read enable and a write enable line. In this case, only one line can be asserted at any one time. The memory location in which the read and write operations are to take place, of course, is selected by the value of the address lines. The operation of the memory chip is shown in Figure



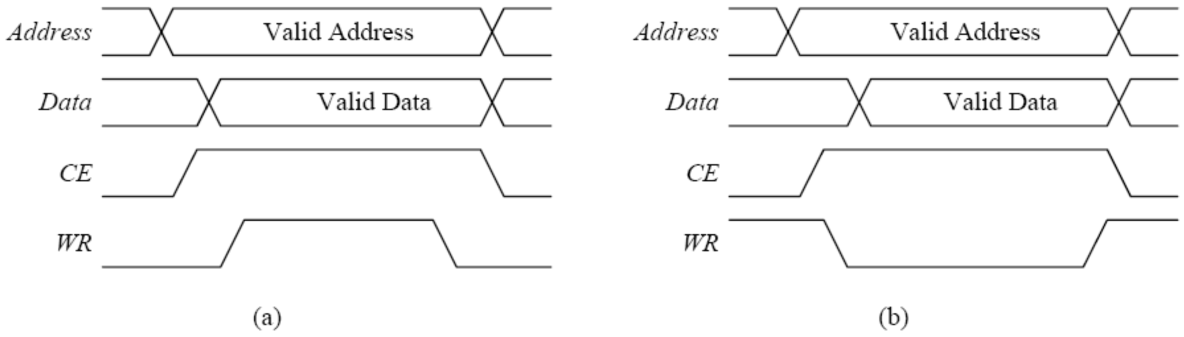
(a)

CE	*WR*	Operation
0	×	None
1	0	Read from memory location selected by address lines
1	1	Write to memory location selected by address lines

(b)

A RAM 2^n X m chip a)logic symbol b)operation table

A memory read operation also begins with setting a valid address on the address lines, followed by *CE* going high. The *WR* line is then pulled low, and shortly after, valid data from the addressed memory location is available on the data lines. The timing diagram for the read operation is shown in Figure 1.6 (a). Each bit in a static RAM chip is stored in a memory cell similar to the circuit shown in Figure 1.6(b). The main component in the cell is a D latch with enable. A tri-state buffer is connected to the output of the D latch so that it can be selectively read from. The *Cell enable* signal is used to enable the memory cell for both reading and writing. For reading, the *Cell enable* signal is used to enable the tri-state buffer. For writing, the *Cell enable* together with the *Write enable* signals are used to enable the D latch so that the data on the *Input* line is latched into the cell. The logic symbol for the memory cell is shown in Figure 1.6 Each row forms a single storage location, and the number of memory cells in a row determines the bit width of each location. So all of the memory cells in a row are enabled with the same address. Again, a decoder is used to decode the address lines, *A*0 and *A*1. In this example, a 2-to-4 decoder is used to decode the four address locations. The *CE* signal is for enabling the chip, specifically to enable the read and write functions through the two AND gates. The internal *WE* signal, asserted when both the *CE* and *WR* signals are asserted, is used to assert the *Write enables* for all of the memory cells. The data comes in from the external data bus, *Di*, through the input buffer and to the *Input* line of each memory cell. The purpose of using an input buffer for each data line is so that the external signal coming in only needs to drive just one device (the buffer) rather than having to drive several devices (i.e., all of the memory cells in the same column). Which row of memory cells actually gets written to will depend on the given address. The read operation requires *CE* to be asserted and *WR* to be de-asserted. This will assert the internal *RE* signal, which in turn will enable the four output tri-state buffers at the bottom of the circuit diagram. Again, the location that is read from is selected by the address lines.



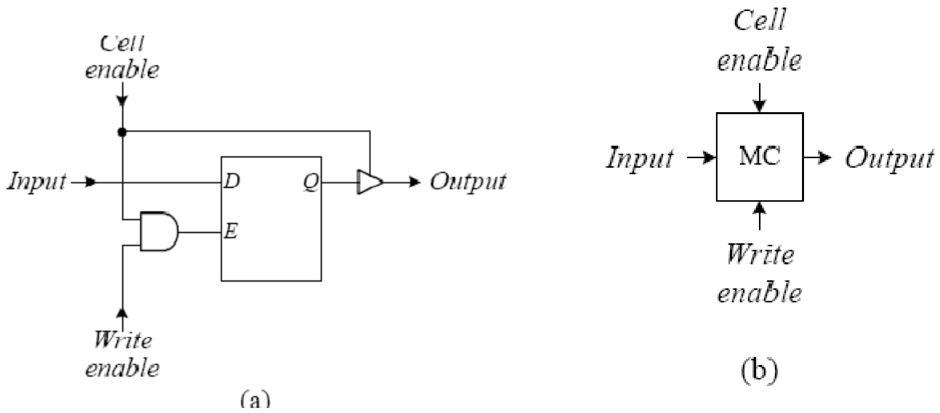


Fig:1.18 memory timing diagram a) read operation b) write operation

VHDL code for a 16 × 4 RAM chip.

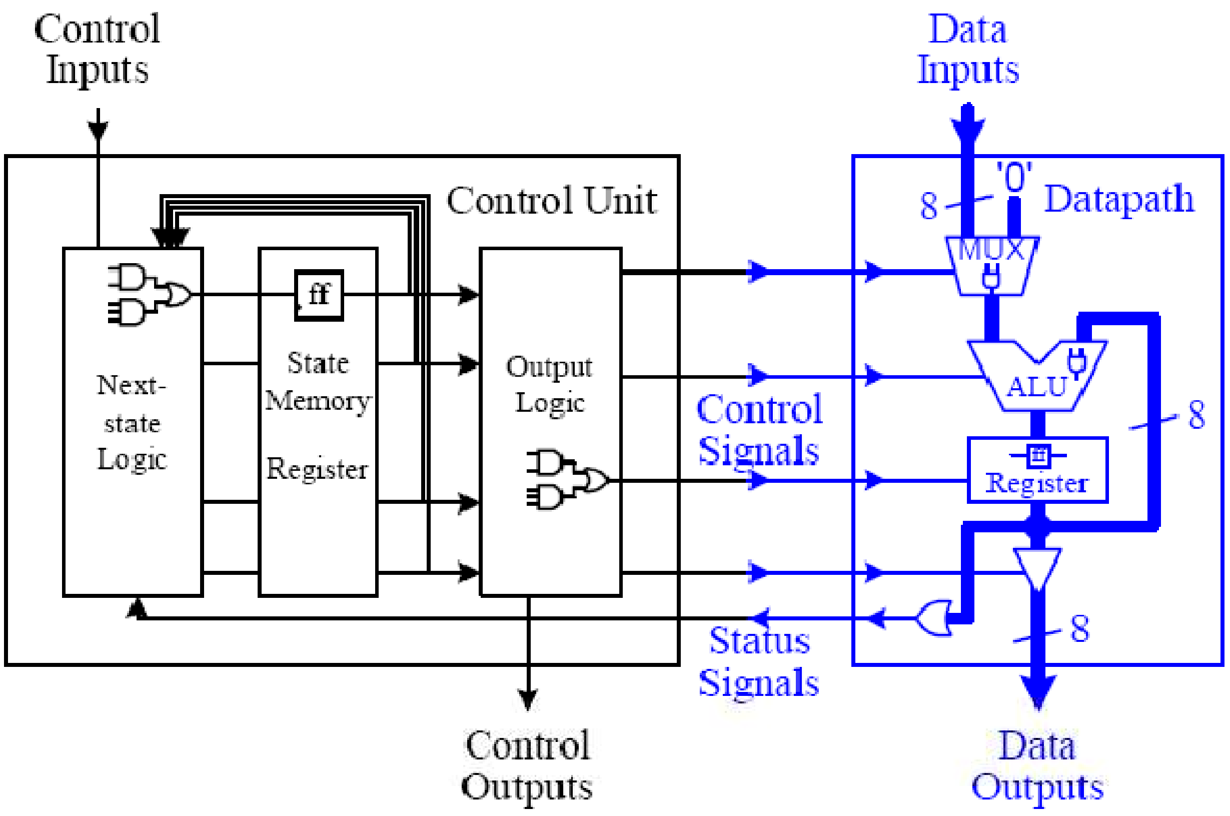
```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL; -- needed for CONV_INTEGER()
ENTITY memory IS PORT (
CE, WR: IN STD_LOGIC; --chip enable, write enable
A: IN STD_LOGIC_VECTOR(3 DOWNTO 0); --address
D: BUFFER STD_LOGIC_VECTOR(3 DOWNTO 0)); --data
END memory;
ARCHITECTURE Behavioral OF memory IS
BEGIN
PROCESS (CE, WR)
```

```
SUBTYPE cell IS STD_LOGIC_VECTOR(3 DOWNTO 0);
TYPE memArray IS array(0 TO 15) OF cell;
VARIABLE mem: memArray; --memory contents
VARIABLE ctrl: STD_LOGIC_VECTOR(1 DOWNTO 0);
BEGIN
ctrl := CE & Wr; -- group signals for CASE decoding
CASE ctrl IS
WHEN "10" => -- read
D <= mem(CONV_INTEGER(A));-- fn TO convert from bit vector TO integer
WHEN "11" => -- write
mem(CONV_INTEGER(A)) := D;-- fn TO convert from bit vector TO integer
WHEN OTHERS => -- invalid or not enable
D <= (OTHERS => 'Z');
END CASE;
END PROCESS;
END Behavioral
```

CHAPTER 3

DATA PATH

3.1 DATAPATH



we learned how to design functional units for performing single, simple data operations, such as the adder for adding two numbers or the comparator for comparing two values. The next logical question to ask ishow do we design a circuit for performing more complex data operations or operations that involve multiple steps? For example, how do we design a circuit for adding four numbers or a circuit for adding a million numbers? For adding four numbers, we can connect three adders together, as shown in Figure 1.18 (a) However, for adding a million numbers, we really don’t want to connect a million minus one adders together like that. Instead, we want a circuit with just one adder and to use it a million times. A **datapath** circuit allows us to do just that, that is, for performing operations involving multiple steps. Figure 1.18(b) shows a simple datapath using one adder to add as many numbers as we want. In order for this to be possible, a register is needed to store the temporary result after each addition.

The temporary result from the register is fed back to the input of the adder so that the next number can be added to the current sum.

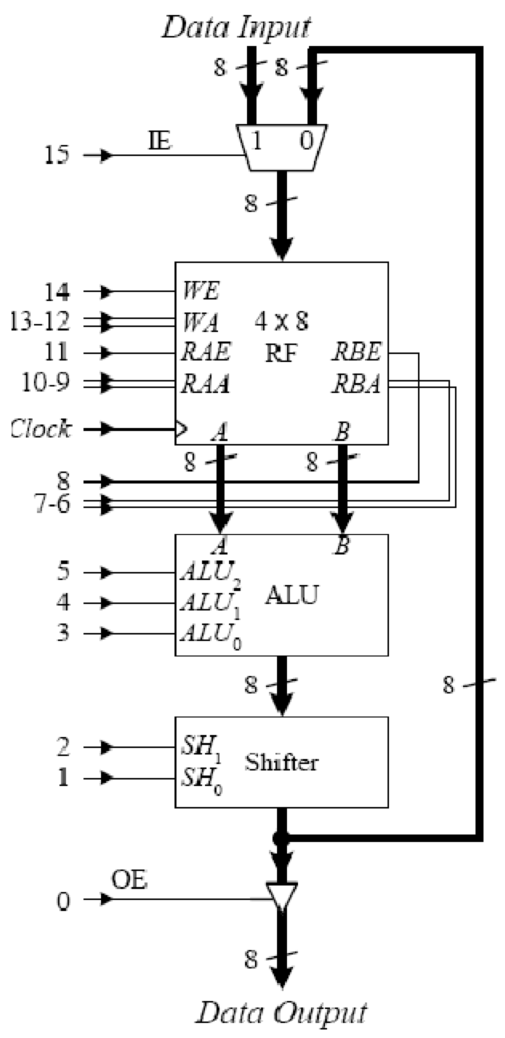
In this chapter, we will look at the design of datapaths. Recall that the datapath is the second main part in a microprocessor. The datapath is responsible for the manipulation of data. It includes (1) functional units such as adders, shifters, multipliers, ALUs, and comparators, (2) registers and other memory elements for the temporary storage of data, and (3) buses, multiplexers, and tri-state buffers for the transfer of data between the different

components in the datapath, and the external world. From the microprocessor road map figure at the beginning of this chapter, we see that external data enters the datapath through the **data input** lines. Results from the datapath operations are provided through the **data output** lines. These signals serve as the primary input/output data ports for the microprocessor.

In order for the datapath to function correctly, appropriate **control signals** must be asserted at the right time.Control signals are needed for all of the select and control lines for all of the components used in the datapath. This includes all of the select lines for multiplexers, ALUs, and other functional units having multiple operations; all of the read/write enable signals for registers and register files; address lines for register files; and enable signals for tristate buffers. Thus, the operation of the datapath is determined by which control signals are asserted or de-asserted and at what time. In a microprocessor, these control signals are generated by the **control unit**.

GENERAL DATAPATH

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY datapath IS PORT (
clock: IN STD_LOGIC;
input: IN STD_LOGIC_VECTOR( 7 DOWNTO 0 );
IE, WE: IN STD_LOGIC;
WA: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
RAE: IN STD_LOGIC;
RAA: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
RBE: IN STD_LOGIC;
RBA: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
aluSel: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
shSel: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
OE: IN STD_LOGIC;
output: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END datapath;
ARCHITECTURE Structural OF datapath IS
COMPONENT mux2 PORT (
S: IN STD_LOGIC; -- select lines
D1, D0: IN STD_LOGIC_VECTOR (7 DOWNTO 0); -- data bus input
Y: OUT STD_LOGIC_VECTOR (7 DOWNTO 0)); -- data bus output
END COMPONENT;
COMPONENT regfile PORT (
clk: IN STD_LOGIC; --clock
WE: IN STD_LOGIC; --write enable
WA: IN STD_LOGIC_VECTOR (1 DOWNTO 0); --write address
input: IN STD_LOGIC_VECTOR(7 DOWNTO 0); --input
RAE: IN STD_LOGIC; --read enable ports A & B
RAA: IN STD_LOGIC_VECTOR (1 DOWNTO 0); --read address port A & B
RBE: IN STD_LOGIC; --read enable ports A & B
RBA: IN STD_LOGIC_VECTOR (1 DOWNTO 0); --read address port A & B
Aout, Bout: OUT STD_LOGIC_VECTOR (7 DOWNTO 0)); --output port A & B
END COMPONENT;
COMPONENT alu PORT (
ALUSel: IN STD_LOGIC_VECTOR(2 DOWNTO 0); -- select for operations
A, B: IN STD_LOGIC_VECTOR(7 DOWNTO 0); -- input operands
F: OUT STD_LOGIC_VECTOR (7 DOWNTO 0)); -- output
END COMPONENT;
COMPONENT shifter PORT (
SHSel: IN STD_LOGIC_VECTOR (1 DOWNTO 0); -- select for operations
input: IN STD_LOGIC_VECTOR(7 DOWNTO 0); -- input operands
output: OUT STD_LOGIC_VECTOR(7 DOWNTO 0)); -- output
END COMPONENT;
COMPONENT tristatebuffer PORT (
E: IN STD_LOGIC;
D: IN STD_LOGIC_VECTOR(7 downto 0);
Y: OUT STD_LOGIC_VECTOR(7 downto 0));
END COMPONENT;
SIGNAL muxout, rfAout, rfBout: STD_LOGIC_VECTOR( 7 DOWNTO 0 );
SIGNAL aluout, shiftout, tristateout: STD_LOGIC_VECTOR( 7 DOWNTO 0 );
BEGIN
-- doing structural modeling here
U0: mux2 PORT MAP( IE, input, shiftout, muxout );
U1: regfile PORT MAP(clock,WE,WA,muxout,RAE,RAA,RBE,RBA,rfAout,rfBout );
U2: alu PORT MAP( ALUsel, rfAout, rfBout, aluout );
U3: shifter PORT MAP(SHSel,aluout,shiftout);
U4: tristatebuffer PORT MAP(OE, shiftout, tristateout);
output <= tristateout;
END Structural;
```



(a)

ALU_2	ALU_1	ALU_0	Operation
0	0	0	Pass through A
0	0	1	A AND B
0	1	0	A OR B
0	1	1	NOT A
1	0	0	$A + B$
1	0	1	$A - B$
1	1	0	$A + 1$
1	1	1	$A - 1$

(b)

SH_1	SH_0	Operation
0	0	Pass through
0	1	Shift left and fill with 0
1	0	Shift right and fill with 0
1	1	Rotate right

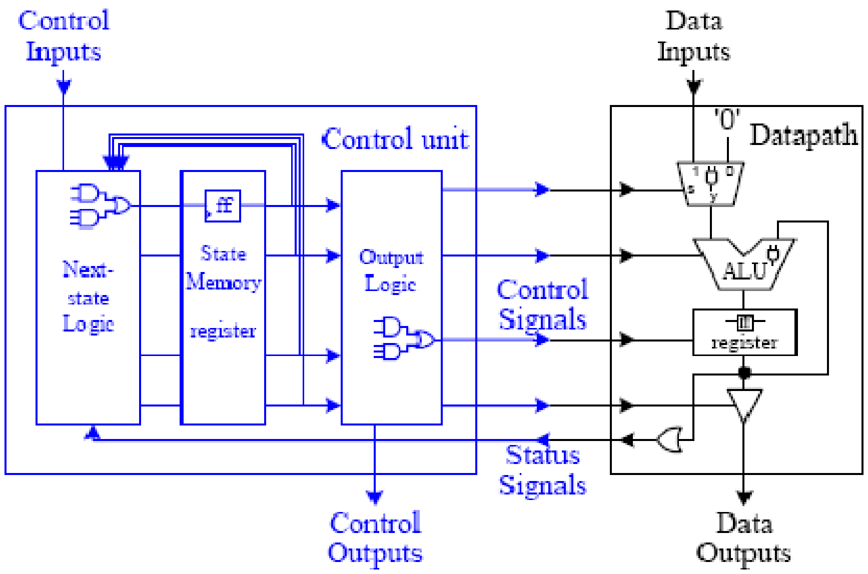
(c)

Fig:1.18 Complex general datapath with a register file: (a) circuit; (b) ALU operations; (c) Shifter operations

CHAPTER 4

CONTROL PATH

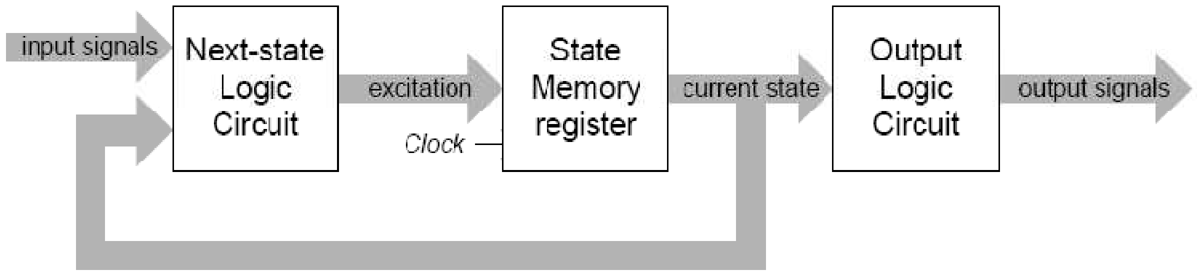
CONTROL PATH: The control unit is a **sequential circuit** in which its outputs are dependent on both its current and past inputs. This history of past inputs is stored in the **state memory** and is said to represent the **state** of the circuit.



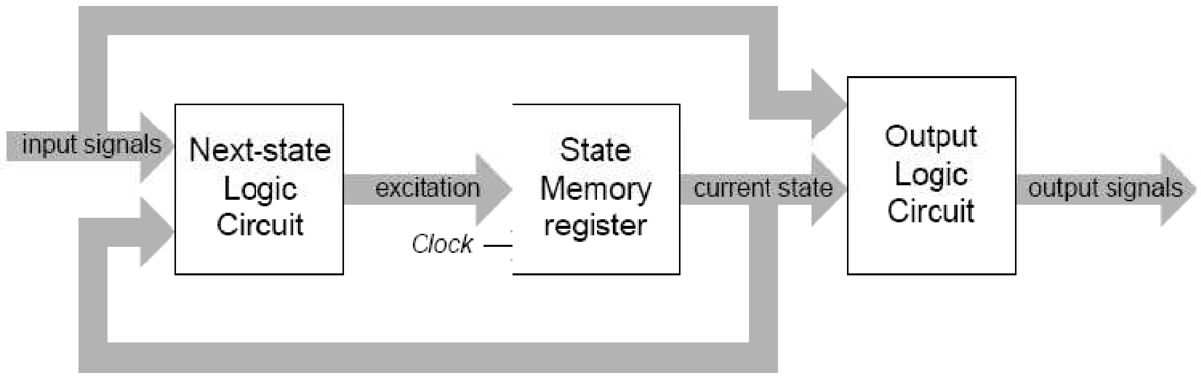
Thus, the circuit changes from one state to the next when the content of the memory changes. Depending on the current state of the circuit and the input signals, the **next-state logic** will determine what the next state ought to be by changing the content of the state memory. Hence, a sequential circuit executes by going through a sequence of states. Since the state memory is finite, therefore the total number of different states that the circuit can go to is also finite. This is not to be confused with the fact that the sequence length can be infinitely long. However, because of the reason of having only a finite number of states, a sequential circuit is also referred to as a **finite-state machine** (**FSM**).

The **control unit** inside the microprocessor is thus an example of a finite-state machine. By stepping through a sequence of states, the control unit controls the operations of the data path. For each state that the control unit is in, the **output logic** that is inside the control unit will generate all the appropriate control signals for the data path to perform one operation. The speed in which the finite-state machine sequences through the states is determined by the clock signal. At each active edge of the clock signal, the state memory register is enabled and

the next state value is stored in. The limiting factor for the clock speed is whether all the operational units inside the data path can finish their operations within one clock period.
In this chapter, we will look at the design of finite-state machines in general. In the next chapter, we will combine what we have learned about data paths and finite-state machines together to construct a complete microprocessor
Finite-state machines are classified into two main types: Moore and Mealy. A **Moore** type FSM is one where the output of the machine is dependent only on the current state, whereas a **Mealy** type FSM is one where the output is dependent on both the current state and the input signals.



(a)



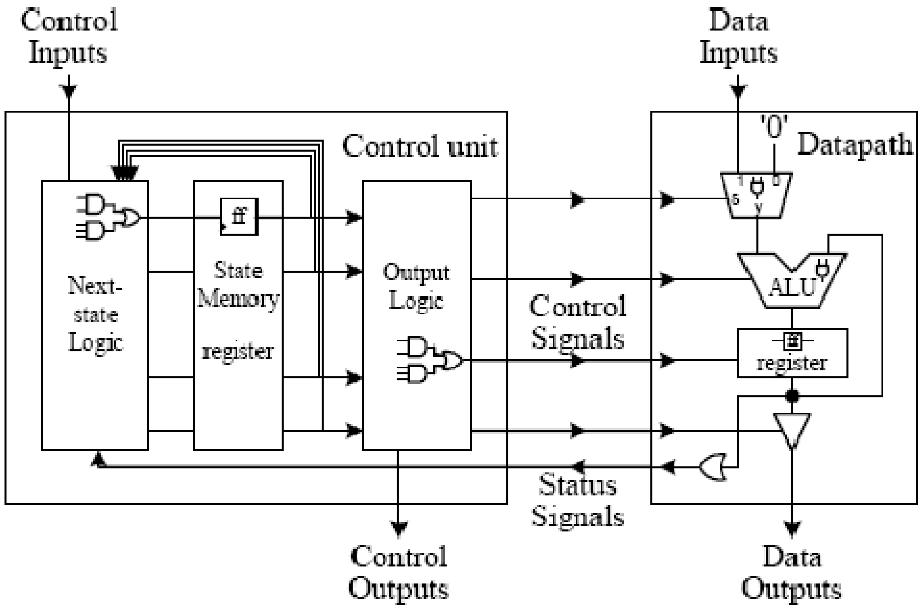
(b) **Fig: 1.19 a) Moore type FSM b) Mealy type FSM**

4.1 Dedicated Microprocessors:

All microprocessors can be divided into two main categories: **general-purpose microprocessors** and **dedicated microprocessors** also known as application specific integrated circuits (**ASIC**s). A general-purpose microprocessor is capable of performing a variety of computations. In order to achieve this goal, each computation is not hardwired into the processor, but rather is represented by a sequence of instructions in the form of a program
that is stored in the memory and executed by the microprocessor. The program in memory can be easily changed so that another computation can be performed. Because of the general nature of the processor, it is likely that in performing a specific computation, not all of the resources available inside the microprocessor are used.
An ASIC, on the other hand, is dedicated to performing only one task. The instructions for performing that one task are, therefore, hardwired into the processor itself and once manufactured, cannot be modified again. In other words, no program memory is required because the program is built right into the circuit itself. If the ASIC is completely customized, then only those resources that are required by the computation are included in the ASIC and so no resources are wasted.

The **control unit** (or **controller**) is responsible for controlling all the operations of the datapath by providing appropriate control signals to the datapath at the appropriate time. At any one time, the control unit is said to be in a certain **state** as determined by the content of the **state memory**. The state memory is simply a register with one or more (D) flip-flops. The control unit operates by transitioning from one state to another – one state per clock cycle.
Depending on the current state, the control inputs and the status signals, the **next-state logic** in the control unit will determine what state to go to next in the next clock cycle. Thus, the control unit is also referred to as a **finite-state machine** (**FSM**) because of this. In every state, the **output logic** that is in the control unit generates all the appropriate control signals for controlling the datapath. The datapath, in return, provides status signals for the nextstate logic. Upon completion of the computation, the control output line is asserted to notify external devices that the value on the data output lines is valid

4.2 Manual Construction of a Dedicated Microprocessor:



datapath is designed and how it is used to execute a particular algorithm by specifying the control words to manipulate the datapath at each clock cycle. In that chapter, we tested the datapath by setting the control word signals manually. However, to actually have the datapath automatically operate according to the control words, a control unit is needed that will generate the control signals that correspond to the control words at each clock cycle. Thus, we need to construct this control unit, and when combined with the datapath, forms the complete dedicated microprocessor that will execute the algorithm. These two components are connected together by the control signals and status signals as shown in Figure 1 using structural VHDL. The control signals are generated by the control unit to control the operations of the datapath, while the status signals are generated by the datapath to inform the next-state logic in the FSM in determining what the next state should be in the execution of the algorithm. The control unit is constructed exactly using the FSM synthesis method . Having designed the datapath and the control words for solving a given problem, we are now ready to build the control unit for it. We start by constructing the state diagram for the FSM. One control word is assigned to one state in the state diagram. Every state is given a symbolic name for convenience. The sequence in which the states are connected follows the sequence of the statements in the algorithm. Conditional branches in the algorithm will have two edges going out of a state with the conditions labeled on the edges; one edge for when the condition is true and the other for when the condition is false. These conditions are the status signals that are generated by the datapath and passed to the next-state logic in the FSM. A general datapath will need additional combinational circuitry corresponding status signal to be used by the next-state logic. There are also conditions from external inputs such as the *Start* signal for starting the execution of the algorithm/circuit. In addition, an external *Reset* signal is used to reset the FSM to its starting state. From the state diagram, we construct the next-state table, which has the same information as the state diagram but with the actual bit encodings assigned to the states. The total number of states in the state diagram will determine the number of flop-flops needed for the state memory. Assigning different encodings to the states may produce a more optimized circuit. The state diagram, the next-state table tells us what the next state of the FSM is to be, given the current state that the FSM is in and the current values of the status and input signals.

We have learned how to design the datapath and the control unit separately. In this chapter, you will learn how to put them together to form a dedicated microprocessor. There are several levels at which a microprocessor can be designed. At the lowest level, you manually construct the circuit for both the control unit and the datapath and then connect them together. This method of construction uses the **FSM+D** (FSM *plus* datapath) model since the FSM and the datapath are constructed separately. This also ties together everything that you have learned so far from this book. The next level of microprocessor design also uses the FSM+D model. As before, you manually construct the datapath. However, instead of manually constructing the FSM, you synthesize the FSM from behavioral VHDL.There will be a next-state process and an output process in the behavioral code. The next-state process will generate the next-state logic and the output process will generate all the control signals for driving the datapath. The FSM and the datapath are connected together in an enclosing entity module using the control and status signals. In practice, this is probably the lowest level in which you would want to design a dedicated microprocessor. The advantage of using the FSM+D model is that you have full control as to how the control unit and the datapath are built.

The third level of microprocessor design uses the **FSMD** (FSM *with* datapath) model. Using this model, you would still design the FSM using behavioral VHDL code. However, instead of constructing the datapath manually as a separate module, all the datapath operations are embedded within the FSM entity using the built-in VHDL operators. During the synthesis process, the synthesizer will automatically generate a separate FSM and datapath. The advantage of this model is that you do not have to design the datapath, but you still have full control as to what operation is executed in what state or in what clock cycle. In other words, you have control over the timing of the circuit.

Finally, a microprocessor can be described completely at the behavioral level using VHDL. This process synthesizes the full microprocessor with its control unit and datapath automatically. Keep in mind that whether you write VHDL code for a microprocessor using the FSM+D model, the FSMD model, or the behavioral model, after synthesis, the resulting microprocessor circuit still contains both the FSM and the datapath as two separate components and are connected together via the control and status signals

From the state diagram, we construct the next-state table, which has the same information as the state diagram but with the actual bit encodings assigned to the states. The total number of states in the state diagram will determine the number of flop-flops needed for the state memory. Assigning different encodings to the states may produce a more optimized circuit. the state diagram, the

next-state table tells us what the next state of the FSM is to be, given the current state that the FSM is in and the current values of the status and input signals.Up until this point, the FSM design has been independent of what flip-flop type is used. However, a FSM can be implemented using any of the four types of flip-flops or combinations of them, and using different flip-flops can produce a smaller circuit. We will however, use only D flip-flops because of their ease of use and because this is the current trend in microprocessor design. We need to convert the next-state table to the implementation table for the D flip-flop. The implementation table shows the necessary inputs for the (D) flip-flops that will produce the next states as given in the next-state table. It turns out that the implementation table for the D flip-flop is identical to the next-state table except for the labeling of the entries. In the next-state table, the label for the entries is *Qnext* for the next state to go to, whereas, in the implementation table (see Figure 3(c)), the label for the entries is *D* for the input to the flip-flop. We want to assign to the input *D* the value that will cause the corresponding *Qnext* value in the next-state table. However, since the characteristic equation for the D flip-flop (i.e. the equation that describes the operation of the D (flip-flop) is

$$Qnext = D$$

therefore, the entries in these two tables are the same. If one of the other types of flip-flops is used, the two tables will not be the same. The implementation table is used to derive the excitation equations (i.e. the equation that causes the flip-flop to change state) for all the inputs of all the flip-flops. These equations are dependent on the current state encodings and the values of the status signals . The next-state logic circuit, which is a combinational circuit, is then constructed from these equations.

FSM + D Model: The complete microprocessor can also be designed by writing VHDL code in a truly behavioral style so that both the FSM and the datapath are synthesized automatically. Using the behavioral model to design a circuit is quite similar to writing computer programs using a high-level language. The synthesizer, like the compiler, will translate the VHDL behavioral description of the circuit automatically to a netlist. This netlist can then be programmed directly to a FPGA chip. Since the synthesizer automatically constructs both the FSM and the datapath, therefore, you have no control over what parts are used in the datapath and what control words are executed in what state of the FSM. Not being able to decide what components are used in the datapath is not too big of a problem because the synthesizer does do a good job in deciding that for you. The issue is with not being able to say what control words are executed in what state of the FSM. This is purely a timing issue. In some timing critical applications such as communication protocols and real-time controls, we need to control exactly in what clock cycle a certain operation is performed. In other words, we need to be able to assign a control word to a specific state of the FSM.

This is an example for evaluating the GCD (Greatest Common Denominator) of two values

```
while (A != B){
        if (A > B)
                A = A – B;
        else
                B = B – A;
}
GCD = A;
```

Data path for GCD:

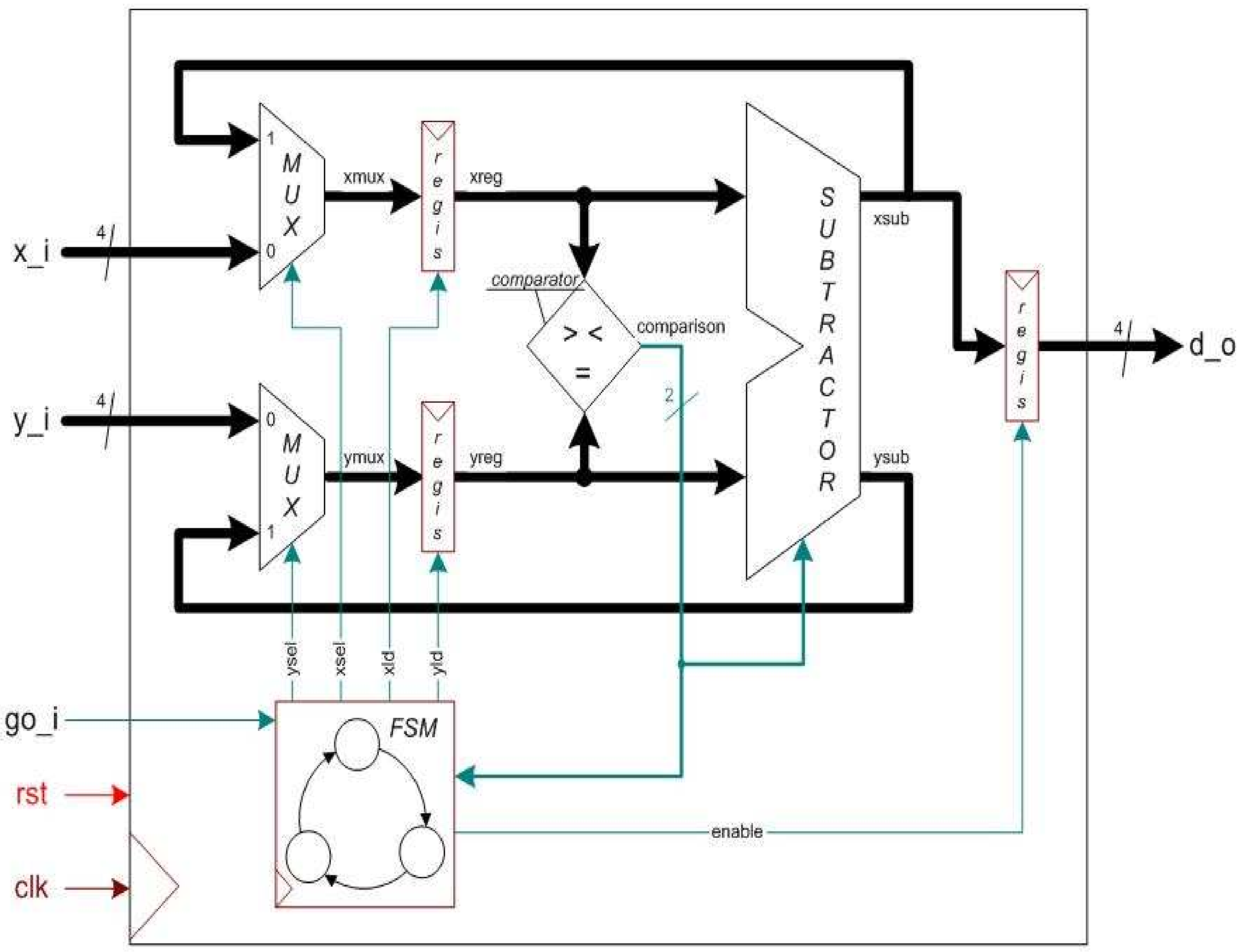


Fig: 1.20 DATA PATH FOR GCD

Control logic for GCD:

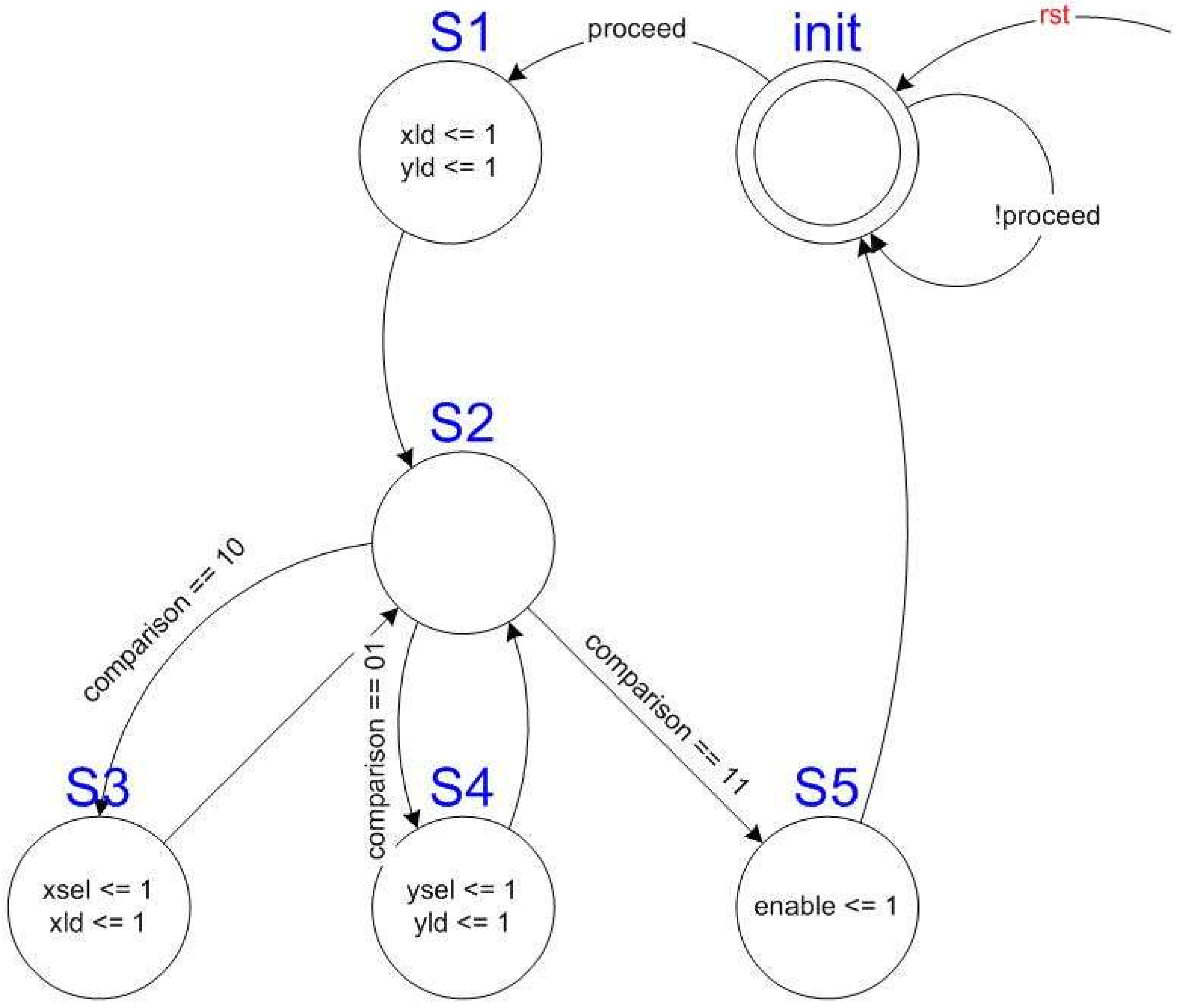


FIG 1.21 CONTROL LOGIC (FSM)

VHDL code for Data path:

```
entity mux is
          port ( rst, sLine: in std_logic;
                    load, result: in std_logic_vector( 3 downto 0 );
                    output: out std_logic_vector( 3 downto 0 )
   );
end mux;
architecture mux_arc of mux is
begin
   process( rst, sLine, load, result )
   begin
          if( rst = '1' ) then
             output <= "0000";              -- do nothing
          elsif (sLine = '0' ) then
             output <= load;                -- load inputs
          else
             output <= result;              -- load results
          end if;
   end process;
end mux_arc;
entity comparator is
   port(rst: in std_logic;
                    x, y: in std_logic_vector( 3 downto 0 );
                    output: out std_logic_vector( 1 downto 0 )
   );
```

```
end comparator;

architecture comparator_arc of comparator is

begin

   process( x, y, rst )

   begin

          if( rst = '1' ) then

             output <= "00";                  -- do nothing

          elsif( x > y ) then

             output <= "10";                  -- if x greater

          elsif( x < y ) then

             output <= "01";                  -- if y greater

          else

             output <= "11";                  -- if equivalance

          end if;

   end process;

end comparator_arc

entity subtractor is

   port(rst: in std_logic;

                 cmd: in std_logic_vector( 1 downto 0 );

                 x, y: in std_logic_vector( 3 downto 0 );

                 xout, yout: out std_logic_vector( 3 downto 0 )

   );

end subtractor;

architecture subtractor_arc of subtractor is

begin

   process( rst, cmd, x, y )
   begin
          if( rst = '1' or cmd = "00" ) then          -- not active
             xout <= "0000";
             yout <= "0000";
          elsif( cmd = "10" ) then                    -- x is greater
             xout <= ( x - y );
             yout <= y;
          elsif( cmd = "01" ) then                    -- y is greater
             xout <= x;
             yout <= ( y - x );
          else
             xout <= x;                          -- x and y are equal
             yout <= y;
          end if;
   end process;
end subtractor_arc;
entity regis is
   port(rst, clk, load: in std_logic;
                input: in std_logic_vector( 3 downto 0 );
                    output: out std_logic_vector( 3 downto 0 )
   );
end regis;
architecture regis_arc of regis is
begin
   process( rst, clk)
   begin
```

```
            if( rst = '1' ) then
                output <= "0000";
            elsif( clk'event and clk = '1') then
                if( load = '1' ) then
                    output <= input;
                end if;
     end if;
    end process;
end regis_arc;
```

VHDL code for control path:

```
entity fsm is
    port(rst, clk, proceed: in std_logic;
                        comparison: in std_logic_vector( 1 downto 0 );
                        enable, xsel, ysel, xld, yld: out std_logic
    );
end fsm;
architecture fsm_arc of fsm is
    type states is ( init, s1, s2, s3, s4, s5 );
    signal nState, cState: states;
begin
    process( rst, clk )
    begin
            if( rst = '1' ) then
                cState <= init;
            elsif( clk'event and clk = '1' ) then
                cState <= nState;
```

```
        end if;
  end process;
 process( proceed, comparison, cState )
  begin
        enable <= '0';
        xsel <= '0';
        ysel <= '0';
        xld <= ‘0';
        yld <= ‘0';
        case cState is
        when init =>    if( proceed = '0' ) then
                           nState <= init;
                        else
                           nState <= s1;
                        end if;

        when s1 =>      xld <= '1';
                        yld <= '1';
                        nState <= s2;

        when s2 =>      if( comparison = "10" ) then
                           nState <= s3;
                        elsif( comparison = "01" ) then
                           nState <= s4;
                        elsif( comparison = "11" ) then
                           nState <= s5;
```

```
                    end if;
```

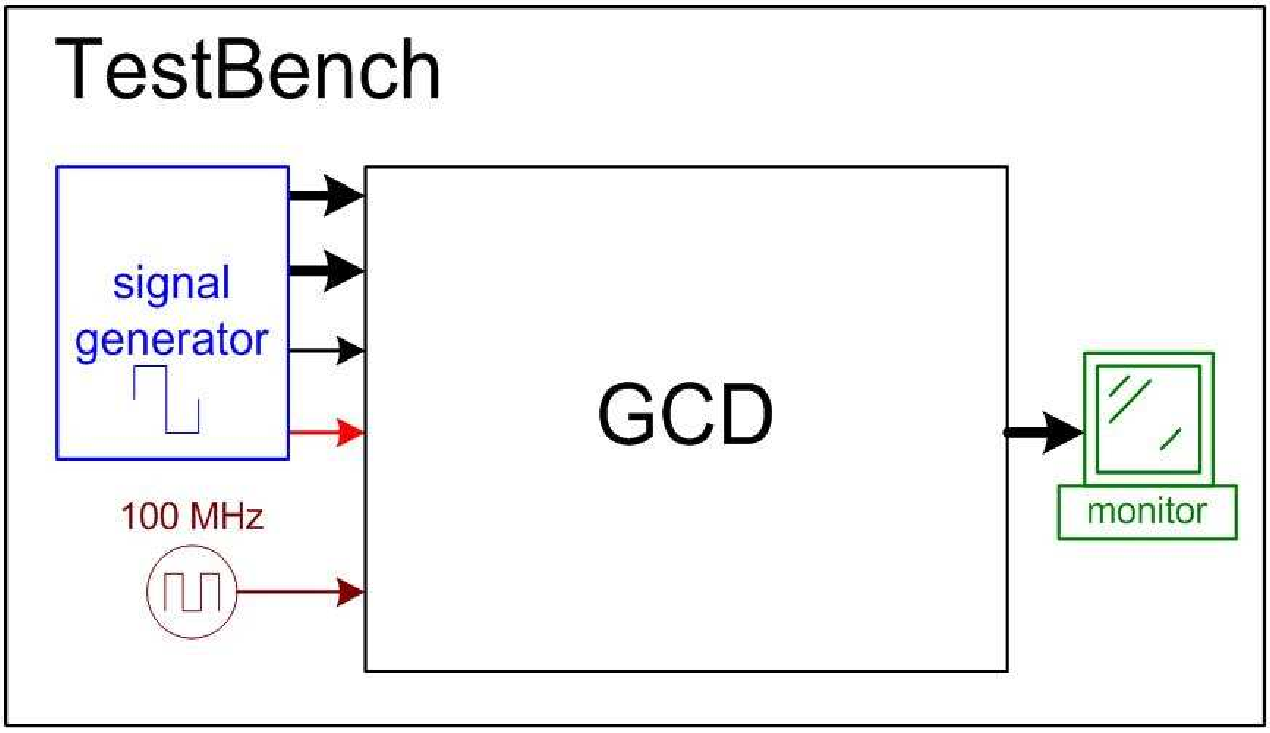
VHDL code for GCD:

```
entity gcd is
  port(rst, clk, go_i: in std_logic;
              x_i, y_i: in std_logic_vector( 3 downto 0 );
              d_o: out std_logic_vector( 3 downto 0 )
  );
end gcd;
architecture gcd_arc of gcd is
component fsm
  port(rst, clk, proceed: in std_logic;
              comparison: in std_logic_vector( 1 downto 0 );
              enable, xsel, ysel, xld, yld: out std_logic
  );
end component;
component mux
  port(rst, sLine: in std_logic;
              load, result: in std_logic_vector( 3 downto 0 );
              output: out std_logic_vector( 3 downto 0 )
  );
end component;
component comparator
  port(rst: in std_logic;
              x, y: in std_logic_vector( 3 downto 0 );
              output: out std_logic_vector( 1 downto 0 )
  );
```

```
end component;
component subtractor
   port(           rst: in std_logic;
                   cmd: in std_logic_vector( 1 downto 0 );
                   x, y: in std_logic_vector( 3 downto 0 );
                   xout, yout: out std_logic_vector( 3 downto 0 )
   );
end component;
component regis
   port(rst, clk, load: in std_logic;
                   input: in std_logic_vector( 3 downto 0 );
                   output: out std_logic_vector( 3 downto 0 )
   );
end component;
signal xld, yld, xsel, ysel, enable: std_logic;
signal comparison: std_logic_vector( 1 downto 0 );
signal result: std_logic_vector( 3 downto 0 );
signal xsub, ysub, xmux, ymux, xreg, yreg: std_logic_vector( 3 downto 0 );
begin
   -- FSM controller
   TOFSM: fsm port map(   rst, clk, go_i, comparison,
                         enable, xsel, ysel, xld, yld );
   -- Datapath
   X_MUX: mux port map(
          rst      => rst,
          sLine    => xsel,
          load     => x_i,
          result   => xsub,
          output => xmux );
   Y_MUX: mux port map( rst, ysel, y_i, ysub, ymux );
   X_REG: regis port map( rst, clk, xld, xmux, xreg );
   Y_REG: regis port map( rst, clk, yld, ymux, yreg );
   U_COMP: comparator port map( rst, xreg, yreg, comparison );
   X_SUB: subtractor port map( rst, comparison, xreg, yreg, xsub, ysub );
   OUT_REG: regis port map( rst, clk, enable, xsub, result );

   d_o <= result;
end gcd_arc;
```

TEST BENCH FOR GCD:



Code for test bench:

```
entity test_GCD is                    -- entity declaration
end test_GCD;
architecture Bench of test_GCD is
  component gcd
```

```
   port(   rst, clk, go_i: in std_logic;
             x_i, y_i: in std_logic_vector( 3 downto 0 );
             d_o: out std_logic_vector( 3 downto 0 )
   );
   end component;
   signal T_clk,T_rst,T_Go_i: std_logic;
   signal T_x_i,T_y_i,T_D_o: std_logic_vector(3 downto 0);
begin
   U1: gcd port map (T_rst, T_clk, T_Go_i, T_x_i, T_y_i, T_D_o);
   Clk_sig: process
   begin
      T_clk<='1';                                    -- clock signal
      wait for 5 ns;
      T_clk<='0';
      wait for 5 ns;
   end process;
process
   variable err_cnt: integer := 0;
   begin

      T_rst <= '1' ;
           T_Go_i <= '0' ;
      wait for 10 ns;
      T_rst <= '0' ;
      wait for 10 ns;
          -- case 1: input C & 8, output should be 4
```

```
    T_x_i <= "1100" ;
        T_y_i <= "1000" ;
        T_Go_i <= '1' ;
        wait for 20 ns;
        T_Go_i <= '0' ;
        wait for 300 ns;
        assert (T_D_o = "0100" ) report "Error1" severity error;
        if (T_D_o /= "0100" ) then
          err_cnt := err_cnt + 1;
        end if;
-- summary of all the tests
       if (err_cnt = 0) then
          assert false
          report "Testbench of GCD completed successfully!"
          severity note;
       else
          assert false
          report "Something wrong, try again"
          severity error;
       end if;
       wait;
  end process;
end Bench;
```

result:

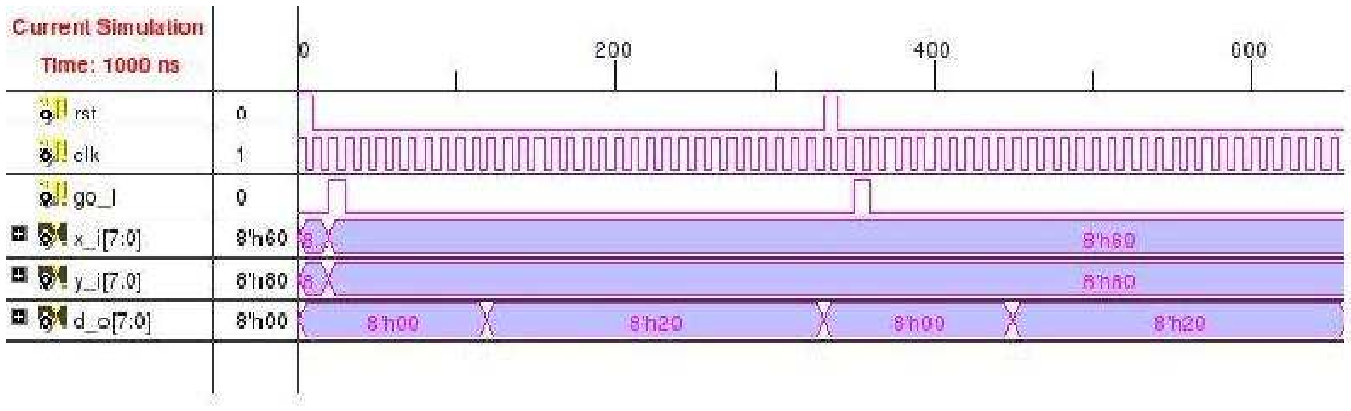


FIG 1.22 SIMULATION TRACE FOR GCD

CHAPTER 5

CONTENT ADDRESSABLE MEMORY

5.1 Content addressable memory:

A typical CAM architecture is shown in Fig. The Address Decoder and Data I/O are similar to those in a RAM. The Bit Control Logic determines the state of data lines for all operations. The Control unit generates signals to control all functional blocks. The Mask Register stores a binary pattern determining whether the corresponding bits in a word are to be masked from further Write and Search operations or not. Each of Valid bits (Vi) indicates whether the match signal (Mi) of the corresponding word is valid or invalid. When multiple match signals occurs for a search operation, the CAM exports the address with the highest priority through the Match Address Priority Encoder and the Hit output is high. Figure shows a $m \times n$ -bit CAM cell array, where *m* is the number of words and *n* is the word length. Each cell is composed of a storage element and a comparison logic. A CAM usually has the following basic operations: Write, Read, Search, Erase, Mask Write, and Mask Search

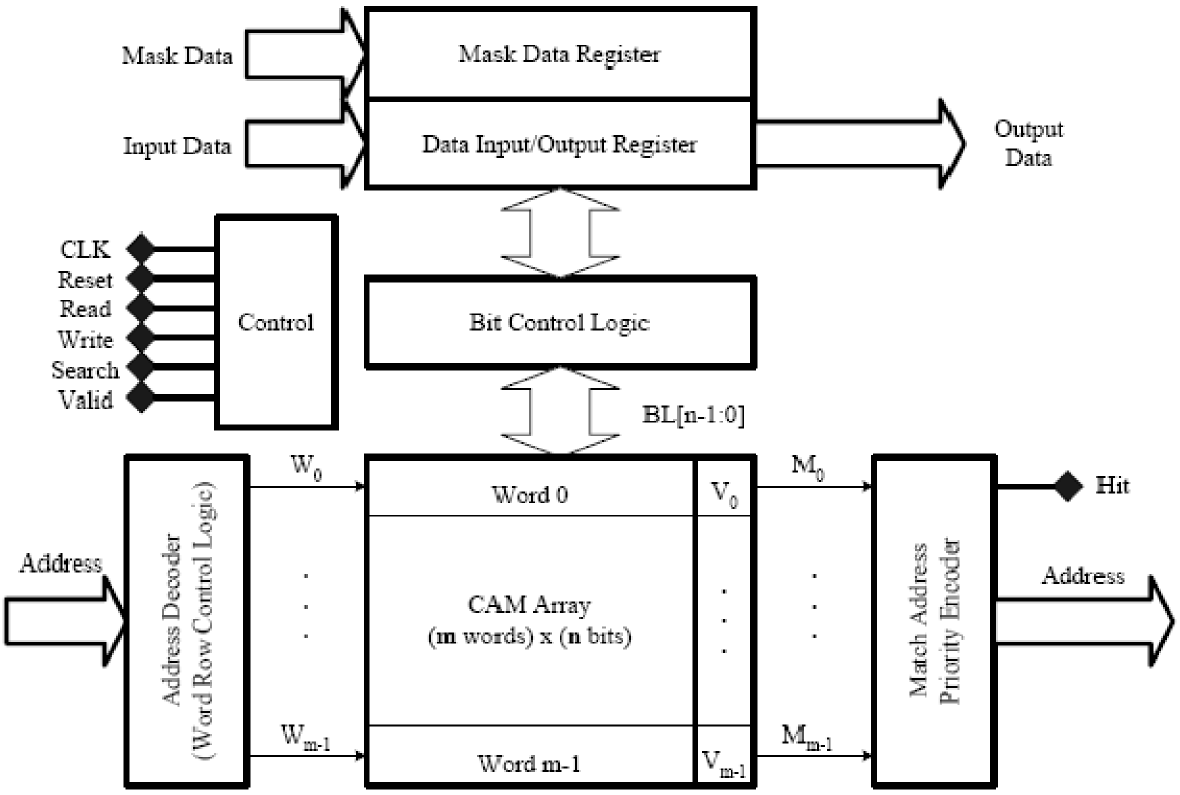


FIG 1.23 Typical CAM architecture

The Write operation of CAMs is similar to that of RAMs. The input data are applied to bit lines (BLs) and Address Decoder generates a signal Wi to activate the corresponding CAM word, such that the data are written into the activated word. Also the Write operation sets the corresponding Valid bit of the word. In contrast, the Erase operation resets the corresponding Valid bit flip-flop of a specified word. The Read operation in a CAM also is identical to the Read operation in a RAM. The Search operation is executed with sending a search pattern to the input register. Then the search pattern is compared with the contents of all words in the CAM. A match (mismatch) is asserted on each match line Mi if the search pattern is the same as (different from) the content of the corresponding word. The Mask Search operation is that search data (comparand) guarantees a match regardless of the data contents of cells in masked bits. A typical CAM implements the Mask Search operation with mask registers which stores binary mask pattern. If the mask pattern has 0s, the corresponding bits are masked from Search operation.

Similarly, the Mask Write operation writes the data into the addressed word and the data of the cells whose corresponding mask register bits with mask data 0s are not changed. Table 2-1 summarizes the states of bit lines for a CAM cell executing different
operations.

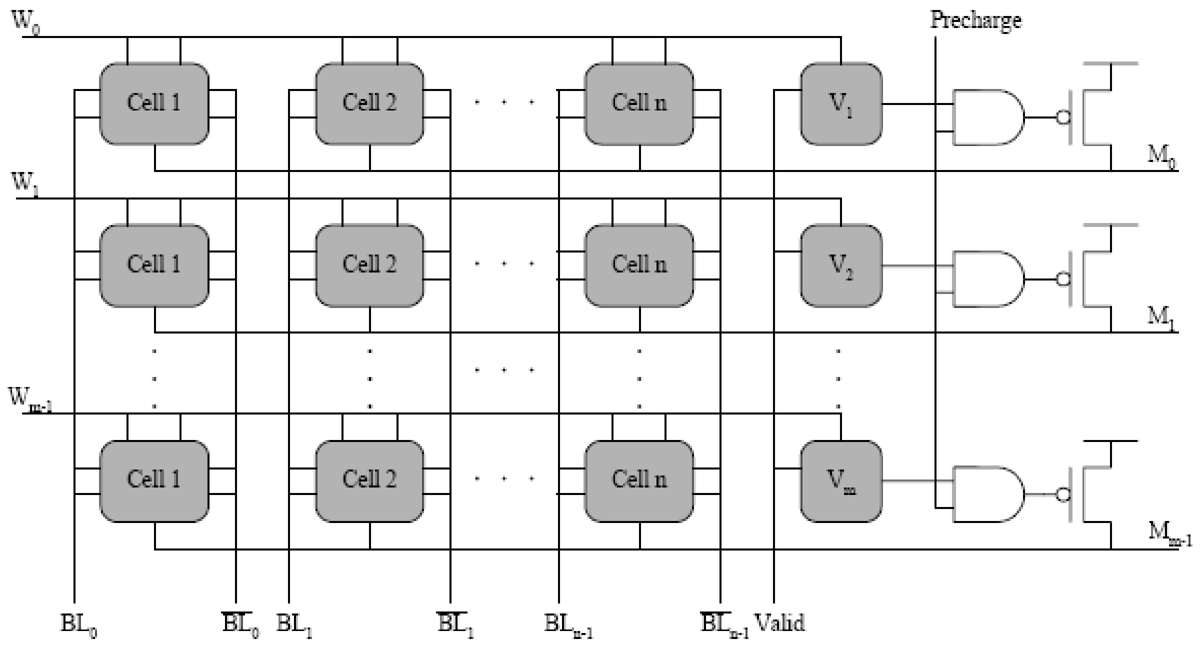


FIG 1.24 An *m*X *n* CAM array

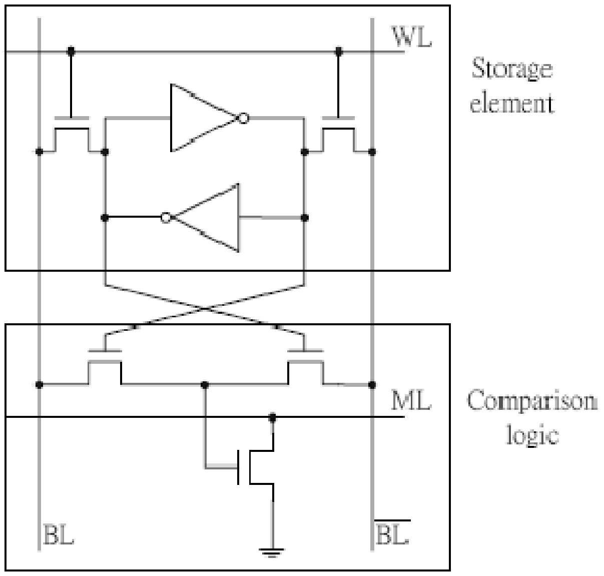
BIT line status with respect to different operations

Precharge for WRITE/READ			Precharge for SEARCH			Mask Write		Mask Search	
BL	$\overline{BL}$	ML	BL	$\overline{BL}$	ML	BL	$\overline{BL}$	BL	$\overline{BL}$
H	H	L	L	L	H	H	H	L	L

5.2 CAM type:

A CAM cell consists of two basic components: a storage element and a comparison logic. The storage element can be implemented with a SRAM cell or a DRAM cell and the comparison logic usually executes XNOR function. Therefore, we can classify CAMs into the static CAMs and dynamic CAMs according to their storage elements. For example, shows a static CAM cell and a dynamic CAM cell. A static CAM usually represents significant silicon cost, since the number of transistors of a static cell is large. A dynamic CAM provides a low-area solution, since the number of transistors of a dynamic CAM cell is small. According to the number of states stored by the storage element, CAMs can be divided into binary and ternary CAMs. A binary CAM cell which can represent two states . In some applications, e.g., image compression, network, etc., the extension of CAM ability to process ternary data, zero (**0**), one (**1**), and don't care (X) is required. a typical TCAM cell. When the Search operation is executed, the stored values (QL, QR) are compared with the comparand data (CL, CR) in the bit lines and the search result is reported by the match output. For example, an encoding scheme of storage is **0** (QL =0, QR =1), **1** (QL =1, QR =0), and X (QL =0, QR =0). If the cell data is **0**, and the comparand is **1**, the ML is discharged to logic 0 through the pull down path m2 and m4. In contrast, if the comparand is **0**, m2 and m3 are turned-off. The ML is still at logic 1 (match). Therefore, if the data is equivalent to the comparand, the two pull-down paths are turned-off and the ML signals a match. In contrast, one of the two pull-down paths is turned-on and the ML signals a mismatch. On the other hand, if the cell stores X data, the m3 and m4 are turned-off, the cell is masked and does not

anticipate the Search operation. Table 2-2 shows the truth table of the search operation.A single-port CAM cell mounts two drain capacitances on both bit lines. Hence, the equivalent capacitance of BL is large. A dual-port CAM cell uses separate bit line pairs to reduce the load on bit line as well as to eliminate the crosstalk effect between long parallel wires. a dual-port CAM cell. One pair of bit lines is used for Read/Write operation, and another pair of search lines is used to feed the comparand to all words for parallel comparision.



1.25 Static binay CAM

VHDL code for CAM:

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity cam_new is
port(
clk,reset:  in std_logic;
wr_en: in std_logic;
key_in : in std_logic_vector(15 downto 0);
hit: out std_logic;
addr_out: out std_logic_vector(1 downto 0));
end cam_new;

architecture Behavioral of cam_new is
type reg_file_type is array(2**2 -1  downto 0) of std_logic_vector(16-1 downto 0);
signal array_reg: reg_file_type;
signal array_next: reg_file_type;
signal en:std_logic_vector(4-1 downto 0);
signal match:std_logic_vector(4-1 downto 0);
signal rep_reg,rep_next: std_logic_vector(1 downto 0);
signal addr_match: std_logic_vector(1 downto 0);
signal wr_key ,hit_flag: std_logic;
begin
process(clk,reset)
begin
if(reset='1')then
array_reg(3)<=(others=>'0');
array_reg(2)<=(others=>'0');
array_reg(1)<=(others=>'0');
array_reg(0)<=(others=>'0');
elsif (clk'event and clk='1') then
array_reg(3)<=array_next(3);
array_reg(2)<=array_next(2);
array_reg(1)<=array_next(1);
array_reg(0)<=array_next(0);
end if;
end process;
process(array_reg,en,key_in)
begin
array_next(3)<=array_reg(3);
array_next(2)<=array_reg(2);
array_next(1)<=array_reg(1);
array_next(0)<=array_reg(0);
if en(3)='1' then
array_next(3)<= key_in;
end if ;
if en(2)='1' then
array_next(2)<= key_in;
end if ;
if en(1)='1' then
array_next(1)<= key_in;
end if ;
if en(0)='1' then
array_next(0)<= key_in;
end if ;
end process;
wr_key <= '1' when( wr_en='1' and hit_flag='0' )else
'0';

process(wr_key,rep_reg)
```

```
begin
if(wr_key='0')then
en<=(others=>'0');
else
case rep_reg is
when "00"=> en<="0001";
when "01"=> en<="0010";
when "10"=> en<="0100";
when others=> en<="1000";
end case;
end if;
end process;
process(clk,reset)
begin
if(reset='1')then
rep_reg<=(others=>'0');
elsif(clk'event and clk='1')then
rep_reg<=rep_next;
end if;
end process;
rep_reg <= rep_reg + 1 when wr_key='1' else
rep_reg ;
---end Behavioral;
process(array_reg,key_in)
begin
match<=(others=>'0');
if array_reg(3)=key_in then
match(3)<='1';
end if;
if array_reg(2)=key_in then
match(2)<='1';
end if;
if array_reg(1)=key_in then
match(1)<='1';
end if;
if array_reg(0)=key_in then
match(0)<='1';
end if;
end process;
with match select
addr_match <=
"00" when "0001",
"01" when "0010",
"10" when "0100",
"11" when others;
hit_flag <= '1' when match /="0000" else '0';
hit <= hit_flag;
addr_out<=addr_match when(hit_flag='1')else
  std_logic_vector(rep_reg);
end Behavioral;
```

Result :

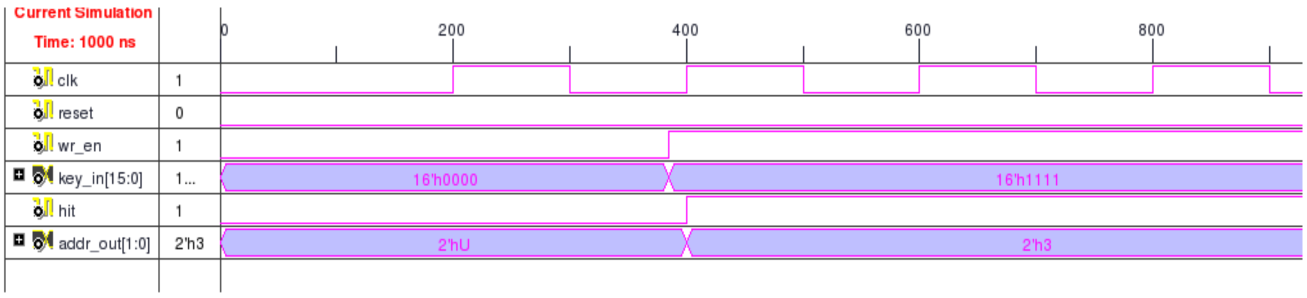


FIG : 1.26 SIMULATION TRACE FOR CAM

CHAPTER 6

CONCLUSION AND REFERENCES

CONCLUSION:

Design of a circuit to implement a state diagram requires sequential diagram, which consists of drawing an implementation model with a state register and combinational logic block assigning a binary encoding to each stae,drawing a state table with input and output and repeating our combinational design process for this table .finally design of a single-processor circuit to implement a program requires us to first schedule the program statement into a complex state diagram, construct a data path from the diagram, create a new state diagram that replaces complex actions and conditions by data path control actions, and then design control path for the new state diagram using sequential design.

Much optimization can be performed at each level of design. Implemented the CAM(content addressable memory),which can be used for applications which require high speed matching .Cam array is same as RAM array but it have match mode is unique to associate memories.camparand block is fill with data pattern to match and mask word indicate which bits are significant, all match happens in a single clock .

REFERENCES

1. T. Kohonen, *Content-Addressable Memories*, 2nd ed. New York: Springer-Verlag, 1987.
2. L. Chisvin and R. J. Duckworth, “Content-addressable and associative memory: alternatives to the ubiquitous RAM,” *IEEE Computer*, vol. 22, no. 7, pp. 51–64, Jul. 1989.
3. I. N. Robinson, “Pattern-addressable memory,” *IEEE Micro*, vol. 12, no. 3, pp. 20–30, Jun. 1992
4. S. Stas, “Associative processing with CAMs,” in *Northcon/93 Conf. Record*, 1993, pp. 161–167
5. T.-B. Pei and C. Zukowski, “VLSI implementation of routing tables: tries and CAMs,” in *Proc. IEEE INFOCOM*, vol. 2, 1991, pp. 515–524.
6. www.wikipedia.org
7. “*Embedded system design*” a unified hardware/software introduction. Frank vahid /Tony givargis.
8. “*Digital logic and computer design*” M.Morris Mano.
9. Peatman , J.B .,*Microcomputer-based design*. New York: McGraw-Hill Book co.,1977
10. Klingman,E.K., *Microprocessor System Design*. Englewood Cliffs, N.J : Prentice-Hall, Inc.,1977
11. *Digital Logic and Microprocessor Design With VHDL*, Enoch O. Hwang,La Sierra University, Riverside
12. Phister M., The Logic Design of Digital Computers . New York : John Wiley and Sons,1958
13. L.-Y. Liu, J.-F.Wang, R.-J.Wang, and J.-Y. Lee, “CAM-based VLSI architectures for dynamic Huffman coding,” *IEEE Trans. Consumer Electron.*, vol. 40, no. 3, pp. 282–289, Aug. 1994.