Course Summary Report - Day 3

By: **Gouthami Channakeshavamurthy**

A20445435

Instructor: Dr. Jafar Saniie

Class Date: 09-28-2019

Due Date: 10-18-2019

Acknowledgment: I acknowledge all of the work (including figures and codes) belongs to me and/or persons who are referenced. I did not receive any assistance from anyone or copy other sources and internet for completing this assignment.

Signature : Gouthami . C

Abstract *- This paper presents the summary of what was taught on Day-2 of Digital System-On-Chip Design course. So far in this subject, we have discussed the basic logic design and VHDL language. As a continuation, the design of multiplexers, registers, counters and loops using VHDL coding is explained. Then the lecture exposed us to the crucial topic of programmable logic devices which includes ROM, PLA and PAL architectures. The FPGA basic architectures and FPGA programming technology helps us to have better understanding about FPGA hardware.*

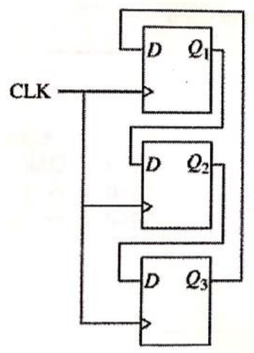
1. INTRODUCTION:

This lecture began with a continuation of the last week's lecture, further expanding on the concepts of VHDL programming. Many features of VHDL that would be useful in implementing more complex logic, such as the IEEE VHDL libraries and the functionality included in such. Furthermore more advanced language concepts such as arrays, loops as well as assert and report statements were covered along with a series of practical examples of using such language features to implement several of the logic functions discussed in earlier lectures. The remainder of the class covered the concepts governing the many different types of programmable logic devices. Additional use cases of programmable logic devices were presented, with a particular emphasis on how the logic function is physically implemented at the gate level within the programmable logic device.

2. TECHNICAL CONTENTS:

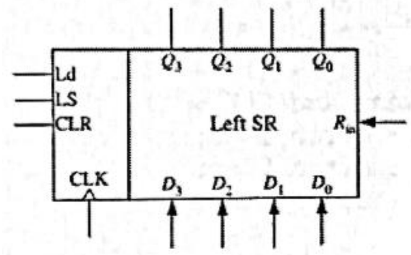
2.1 REGISTERS:

Normally, all the registers and counters will have clock signal. Process statements are used to model registers and counters in VHDL coding.



```
process(CLK)
begin
  if CLK'event and CLK = '1' then
    Q1 <= Q3 after 5 ns;
    Q2 <= Q1 after 5 ns;
    Q3 <= Q2 after 5 ns;
  end if;
end process;
```

Fig.1 Cyclic shift register



```
process(CLK)
begin
  if CLK'event and CLK = '1' then
    if CLR = '1' then Q <= "0000";
      elsif Ld = '1' then Q <= D;
      elsif LS = '1' then Q <= Q(2 downto 0) & Rin;
    end if;
  end if;
end process;
```

Fig.2 Left shift register with load and clear

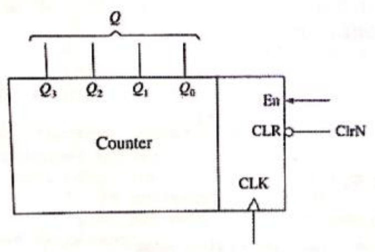
The registers are kind of speed memory which stores bit values using flipflops. The shift registers shift the bit values either right or left with specified no.of bit shifts. The shift registers are very much helpful for multiplication and division, as multiplication is the combination of addition and shifts, division is the combination of subtraction and shifts. The booth algorithm is the widely adapted method for multiplication in hardware. Such a useful shift registers can be designed in VHDL programming with ease.

As already noted, there are four operations in shift as explained in VHDL operators-sll,srl,sla,sra. And also as in fig.6 the signals, load and clear are crucial because while initialization we have to clear the register for correct operation without any errors due to previously stored values.

In fig.5, the VHDL code rotate function is shown. For eg:Q="110" then after cyclic shift Q="101" which is rotate left. In fig.6, when CLR='1' then the register is cleared. When Ld='1' then D is loaded to Q. If Ls='1', then it will shift left one bit position and fills the vacant LSB with Rin.

2.2 COUNTERS:

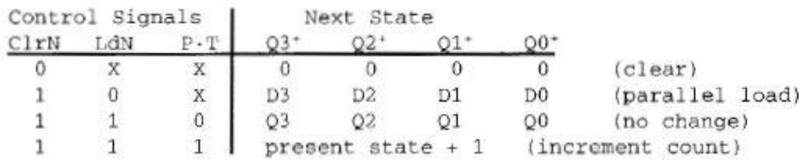
The counters have wide variety of application in digital electronics. It is normally used to find the no.of times the trigger signal appears. Here the counter will increment its value by 1 at every positive edge of the clock, which is the trigger.



```
signal Q: unsigned (3 downto 0);
-----------
process (CLK)
begin
  if CLK'event and CLK = '1' then
    if ClrN = '0' then Q <= "0000";
    elsif En = '1' then Q <= Q + 1;
    end if;
  end if;
end process;
```

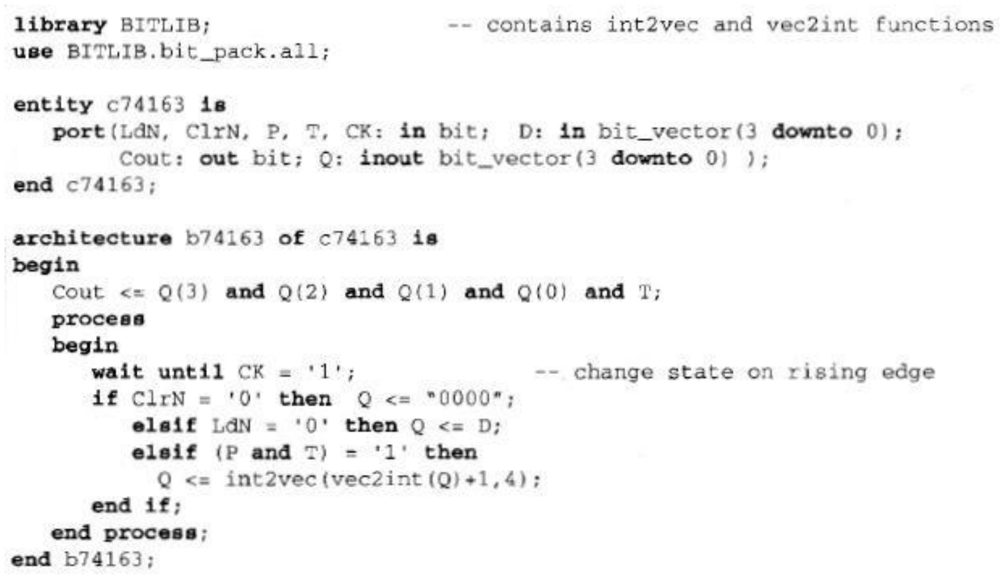
Fig.3 simple counter

Here we can note that, during positive edge of the clock, the condition checks whether clrN='0'. If that's true, then the counter is cleared. The 'N' denotes active low logic. The counters are cleared at the beginning of the operation. Or else, if enable signal is '1' then Q is incremented by 1. Now, let us discuss the famous counter IC 74163. The control signals and the corresponding operations are provided below.



Control Signals			Next State				
ClrN	LdN	P·T	Q3⁺	Q2⁺	Q1⁺	Q0⁺	
0	X	X	0	0	0	0	(clear)
1	0	X	D3	D2	D1	D0	(parallel load)
1	1	0	Q3	Q2	Q1	Q0	(no change)
1	1	1	present state + 1				(increment count)

All the control signals are active load. When LdN='0', Q<=D which is the parallel load. When P.T='0' , there is no change in the output. It can be used as a delay unit or buffer. When all the control signals are 'high', the counter increments the value of Q by 1 (Q<=Q+1). The corresponding VHDL code is provided below.



```
library BITLIB;                  -- contains int2vec and vec2int functions
use BITLIB.bit_pack.all;

entity c74163 is
   port(LdN, ClrN, P, T, CK: in bit;  D: in bit_vector(3 downto 0);
        Cout: out bit; Q: inout bit_vector(3 downto 0) );
end c74163;

architecture b74163 of c74163 is
begin
   Cout <= Q(3) and Q(2) and Q(1) and Q(0) and T;
   process
   begin
      wait until CK = '1';             -- change state on rising edge
      if ClrN = '0' then  Q <= "0000";
         elsif LdN = '0' then Q <= D;
         elsif (P and T) = '1' then
           Q <= int2vec(vec2int(Q)+1,4);
      end if;
   end process;
end b74163;
```

In the above code, int2vec() function is an in-built function which will convert integer value to binary vector of 4 bits specified within parentheses. Similarly, vec2int converts vector to integer. In order to make addition process easier, these conversions are made. Otherwise we can replace the statement by Q<=Q+"0001";

As stated clearly, IC 74163 is a four bit counter. To design a eight bit counter, two ICs are connected appropriately as shown in the figure and in VHDL code, the connection is made by port mapping.

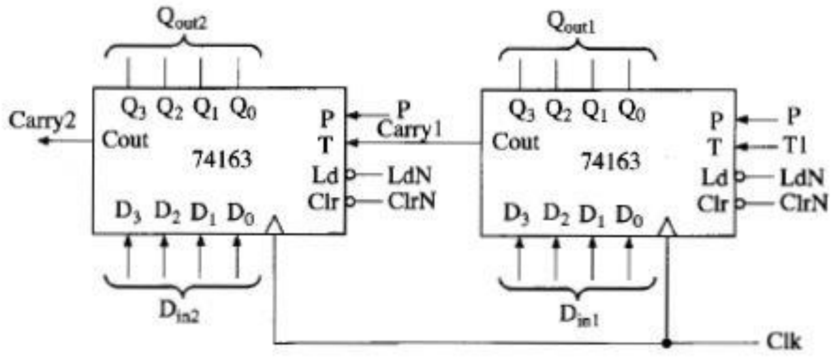
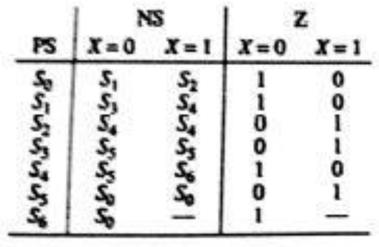


Fig.4 Eight bit counter

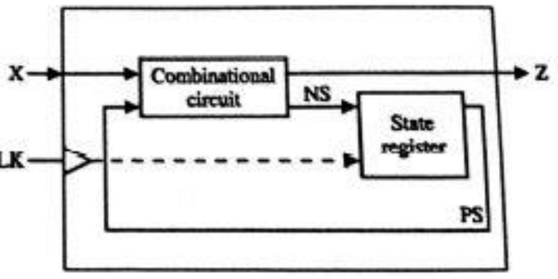
Here the carry1 from first stage to second stage determines the total delay of the counter. Which means only when the first stage completes its execution, the second stage can start its execution.

2.3 MODELING A SEQUENTIAL MACHINE:

The sequential machine can be modeled in behavioral or structural model. The excess-3 code converter is discussed in the previous lectures. The behavioral model is given below. Here there are two processes used, one for combinational network and another state register. These two combine to form a sequential machine.



	NS		Z	
PS	$X=0$	$X=1$	$X=0$	$X=1$
S_0	S_1	S_2	1	0
S_1	S_3	S_4	1	0
S_2	S_4	S_4	0	1
S_3	S_5	S_5	0	1
S_4	S_5	S_6	1	0
S_5	S_0	S_0	0	1
S_6	S_0	—	1	—



In the above behavioral code, case statements are used to determine the output and next state depending on the input X. The following command file for the simulator is used to simulate the behavioral code.

```
entity SM1_2 is
   port(X, CLK: in bit;
        Z: out bit);
end SM1_2;

architecture Table of SM1_2 is
   signal State, Nextstate: integer := 0;
begin
   process(State,X)                          --Combinational Network
   begin
     case State is
      when 0 =>
         if X='0' then Z<='1'; Nextstate<=1; end if;
         if X='1' then Z<='0'; Nextstate<=2; end if;
      when 1 =>
         if X='0' then Z<='1'; Nextstate<=3; end if;
         if X='1' then Z<='0'; Nextstate<=4; end if;
      when 2 =>
         if X='0' then Z<='0'; Nextstate<=4; end if;
         if X='1' then Z<='1'; Nextstate<=4; end if;
      when 3 =>
         if X='0' then Z<='0'; Nextstate<=5; end if;
         if X='1' then Z<='1'; Nextstate<=5; end if;
      when 4 =>
         if X='0' then Z<='1'; Nextstate<=5; end if;
         if X='1' then Z<='0'; Nextstate<=6; end if;
      when 5 =>
         if X='0' then Z<='0'; Nextstate<=0; end if;
         if X='1' then Z<='1'; Nextstate<=0; end if;
      when 6 =>
         if X='0' then Z<='1'; Nextstate<=0; end if;
      when others => null;               -- should not occur
     end case;
  end process;
```

```
  process(CLK)                          -- State Register
    begin
     if CLK='1' then                    -- rising edge of clock
       State <= Nextstate;
     end if;
  end process;
end Table;
```

In the first statement, the signals in the waveform are specified. The second command specifies the CLK signal with 200ns period. CLK is ‘0’ at 0 ns and ‘1’ at 100 ns. Likewise, the input X is also defined and the simulator is run for 1600 ns.

```
wave CLK X State NextState Z
force CLK 0 0, 1 100 -repeat 200
force X 0 0, 1 350, 0 550, 1 750, 0 950, 1 1350
run 1600
```

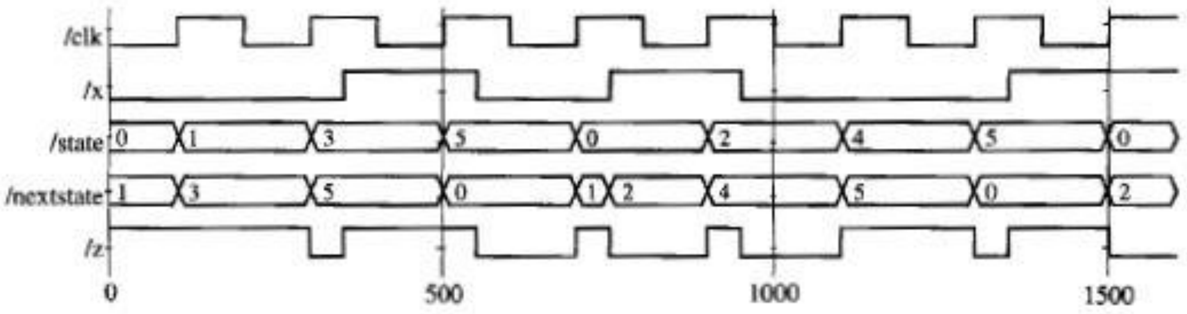


Fig.5 Simulator output of excess-3 code

Excess-3 code converter can also be modeled using logic equations obtained from the truth table. The process of obtaining the equations are explained in the previous lectures.

```
entity SM1_2 is
  port(X,CLK: in bit;
       Z: out bit);
end SM1_2;

architecture Equations1_4 of SM1_2 is
  signal Q1,Q2,Q3: bit;
begin
  process(CLK)
  begin
    if CLK='1' then                    -- rising edge of clock
      Q1<=not Q2 after 10 ns;
      Q2<=Q1 after 10 ns;
      Q3<=(Q1 and Q2 and Q3) or ((not X) and Q1 and (not Q3)) or
            (X and (not Q1) and (not Q2)) after 10 ns;
    end if;
  end process;
  Z<=((not X) and (not Q3)) or (X and Q3) after 20 ns;
end Equations1_4;
```

2.4 SIGNAL VS VARIABLE:

The signal is the one that connects two components within the entity. They are represented using physical wire. The variable is the one, that will not at all be there in the circuit. They are defined only for the convenience of the programming.

2.5 ARRAYS:

In VHDL, we must declare the array type and array object to use an array. For eg: type MAT is array (7 downto 0) of bit;

Here the MAT is the data type which is an array of 8 integer values and the index value from 7 to 0. It is a bit vector of size 8.

Example of array objects of type MAT signal data:MAT:=”00000101”

2.6 LOOPS:

Infinite loops:

The infinite loops are not reliable for software architectures. The system goes into malfunction and cannot be used for other processes. But in hardware design, infinite loop is very much encouraged because the device works continuously until power off.

for loop:

In for loop, the no.of iterations can be specified.

Eg:

```
loop1: for I in 0 to 3 loop Statements;
End loop loop1;
```

while loop:

The loop index can be changed in the while loop unlike for loop. In while the condition is first checked, and then starts the iteration. The loop exits when the condition is false.

2.7 ASSERT AND REPORT STATEMENTS:

After modeling the VHDL system, we need to test the model for correct functioning. For this assert, report and severity statements are used.

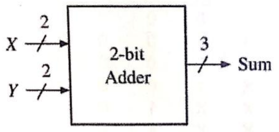
The assert statement checks whether the condition is true or not. If it is wrong, then an error message will be displayed along with string-expression and the severity level which may be note, warning, error or failure. For test bench programs, these statements are very much useful.

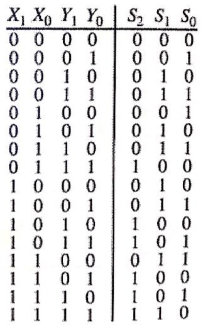
2.8 PROGRAMMABLE LOGIC DEVICES:

The programmable logic devices like ROM, PLA, PAL and PLD are used to realize complex logic in a small area making it possible to fabricate in single IC. PLA, PAL and FPGA are field programmable devices, where the user can program them to his needs. ROM is a factory programmable device, which can be programmed only at factory during manufacturing.

2.9 READ ONLY MEMORY (ROM):

A ROM consists of an array of flipflops interconnected to each other to store bit values. The inputs and outputs are defined as n and m for a ROM having 2^n words with each word m bits long. Typically, a ROM has a decoder and a memory array. The address of the memory to be fetched is fed into the decoder and the corresponding line to that memory word is made 1 to select that word and get the output.





X_1	X_0	Y_1	Y_0	S_2	S_1	S_0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0

Fig.7 2 bit adder with truth table

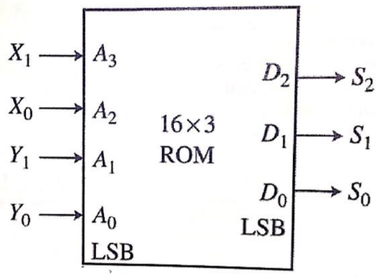


Fig.8 ROM implementation of 2 bit full adder

The 2 bit full adder truth table is provided in fig.10. For eg: X1X0=11 and Y1Y0=10 then the decimal equivalent is 3+2=5, thus S2S1S0=101. This complex logical function can be easily implemented in ROM with 16 input address lines with each word output of 3 bits.

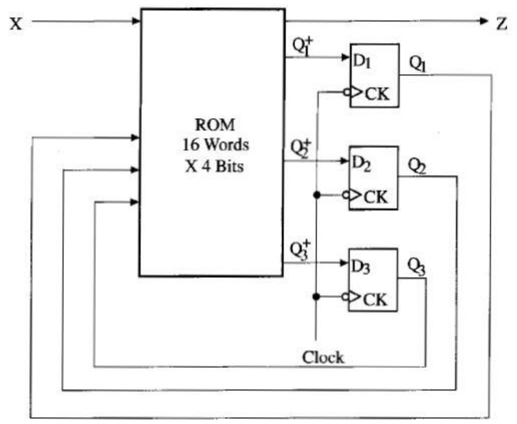


Fig.9 ROM realization of mealy sequential circuit

The ROM can be used to compute the combinational part of the sequential circuit i.e. to compute the output and next state. The state is stored in the flipflops and then fed back to the ROM. The ROM inputs are the present state and X and the outputs are next state and Z.

Q_1	Q_2	Q_3	X	Q_1^+	Q_2^+	Q_3^+	Z
0	0	0	0	1	0	0	1
0	0	0	1	1	0	1	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	1
0	1	0	1	0	0	0	0
0	1	1	0	0	0	0	0
0	1	1	1	0	0	0	1
1	0	0	0	1	1	1	1
1	0	0	1	1	1	0	0
1	0	1	0	1	1	0	0
1	0	1	1	1	1	0	1
1	1	0	0	0	1	1	1
1	1	0	1	0	1	0	0
1	1	1	0	0	1	1	0
1	1	1	1	0	1	1	1

Fig.10 Truth table of excess 3 code converter

2.10 PROGRAMMABLE LOGIC ARRAY:

The programmable logic array(PLA) consists of a programmable AND and OR array. The user can program in two levels of gates. This is a field programmable device where the user can program using switches. The decoder in the ROM is replaced by AND array to realize minterms, which will be ORed by OR array.

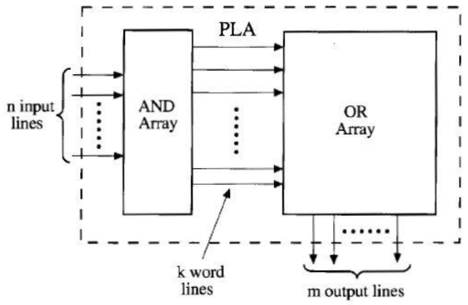


Fig.14 PLA architecture

Let us realize the below logic functions using a PLA.

$$F_0 = \Sigma m(0, 1, 4, 6) = A'B' + AC'$$

$$F_1 = \Sigma m(2, 3, 4, 6, 7) = B + AC'$$

$$F_2 = \Sigma m(0, 1, 2, 6) = A'B' + BC'$$

$$F_3 = \Sigma m(2, 3, 5, 6, 7) = AC + B$$

From the below two figures we can implement that, each minterm is computed in the AND array and the appropriate minterms are ORed in the OR array to form the logic functions.

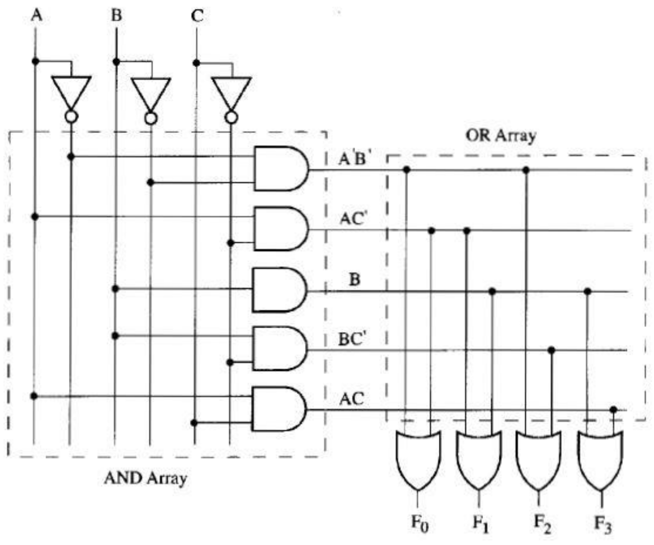
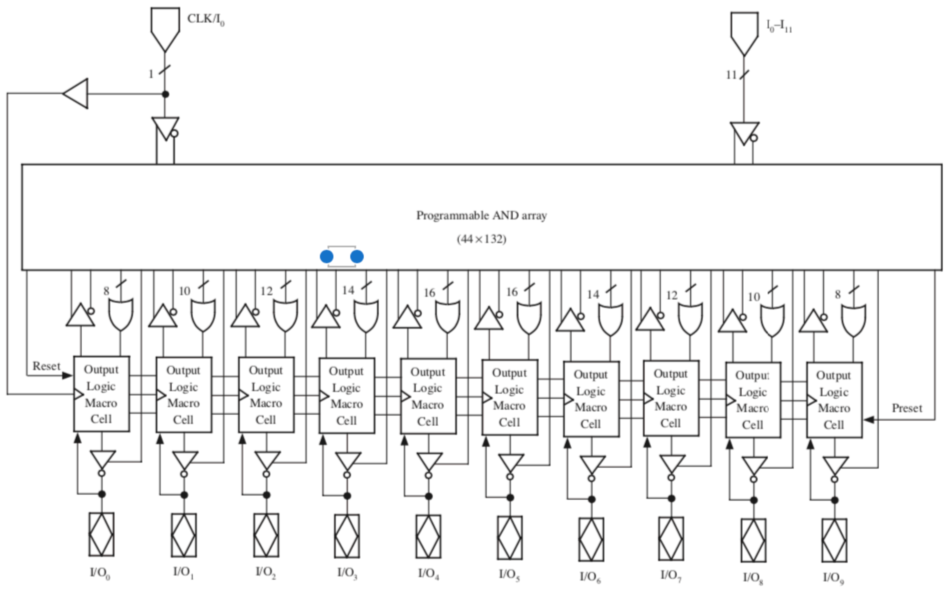


Fig.11 PLA implementation

Product Term	Inputs A	B	C	Outputs F_0	F_1	F_2	F_3
$A'B'$	0	0	–	1	0	1	0
AC'	1	–	0	1	1	0	0
B	–	1	–	0	1	0	1
BC'	–	1	0	0	0	1	0
AC	1	–	1	0	0	0	1

Fig.12 PLA table

The 22CEV10 is a CMOS electrically erasable PLD that can be used to realize both combinational and sequential circuits. The abbreviation PLD has been used as a generic term for all programmable logic devices and also refers to specific devices such as the 22CEV10. In addition to the AND-OR arrays that the PALs have, most PLDs have some type of a macroblock



that contains some multiplexers and some additional programmability. These PLDs are named with reference to their input and output capability. For instance, the 22CEV10 has 12 dedicated input pins and 10 pins that can be programmed as either inputs or outputs. It contains 10 D flip-flops and 10 OR gates. The number of AND gates that feeds each OR gate ranges from 8 through 16. Each OR gate drives an *output logic macrocell*. Each macrocell contains one of the 10 D flip-flops. The flip-flops have a common clock, a common asynchronous reset (AR) input, and a common synchronous preset (SP) input. The name 22V10 indicates a versatile PAL with a total of 22 input and output pins, 10 of which are bidirectional I/O (input/output) pins.

2.11 PROGRAMMABLE ARRAY LOGIC (PAL) :

The programmable array logic has only AND programmable array and OR array cannot to programmed. Thus, PAL is less expensive and easy to use.

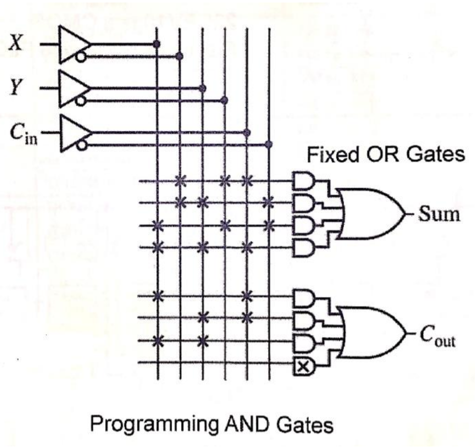
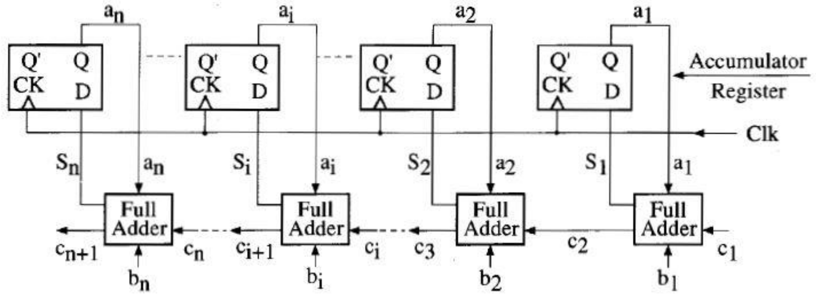


Fig.13 Full adder implementation using PAL

The Boolean equations of full adder is known. Here since the OR gates are fixed, we can see that one of the AND gate is wasted while computing Cout. This is not the case in PLA. This is the trade-off between PAL and PLA.

2.12 COMPLEX PLD:

Parallel adder with accumulator:



In parallel addition, A is loaded into the accumulator and B and Cin is fedinto the adder. After calculation, the SUM is stored into the accumulator. The Cout goes to the next state. The accumulator used here is the 22V10. It consists of AND-OR array with flipflops.

The logic equations of full adder are

Sum=X’Y’Cin+X’YCin’+XY’Cin’+XYCin Cout=X’YCin+XY’Cin+XYCin’+XYCin

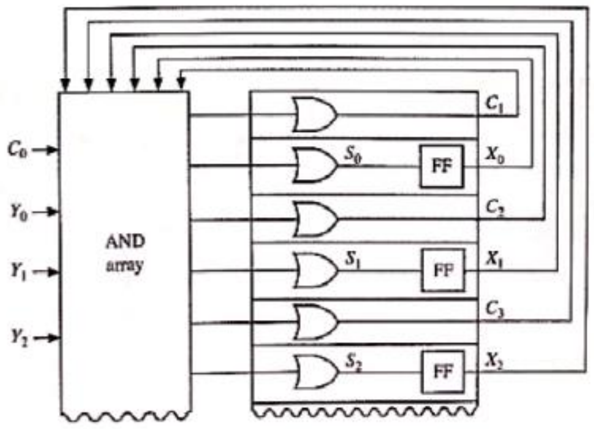


Fig.14 CPLD implementation of parallel adder

We can note that, sum and carry is calculated for every stage and the SUM is stored in the FF of the accumulator. The carry will propagate to next stages. The AND array is used to implement the minterms of the logic equations.

<u>2.13 FPGA BASIC ARCHITECTURES:</u>

<u>Matrix based architectures:</u>

Most of this FPGAs are matrix based in which the logic blocks are arranged in matrix like structure. The routing between these logic blocks are in vertical and horizontal directions called two dimensional channeled routing.

<u>Row based architectures:</u>

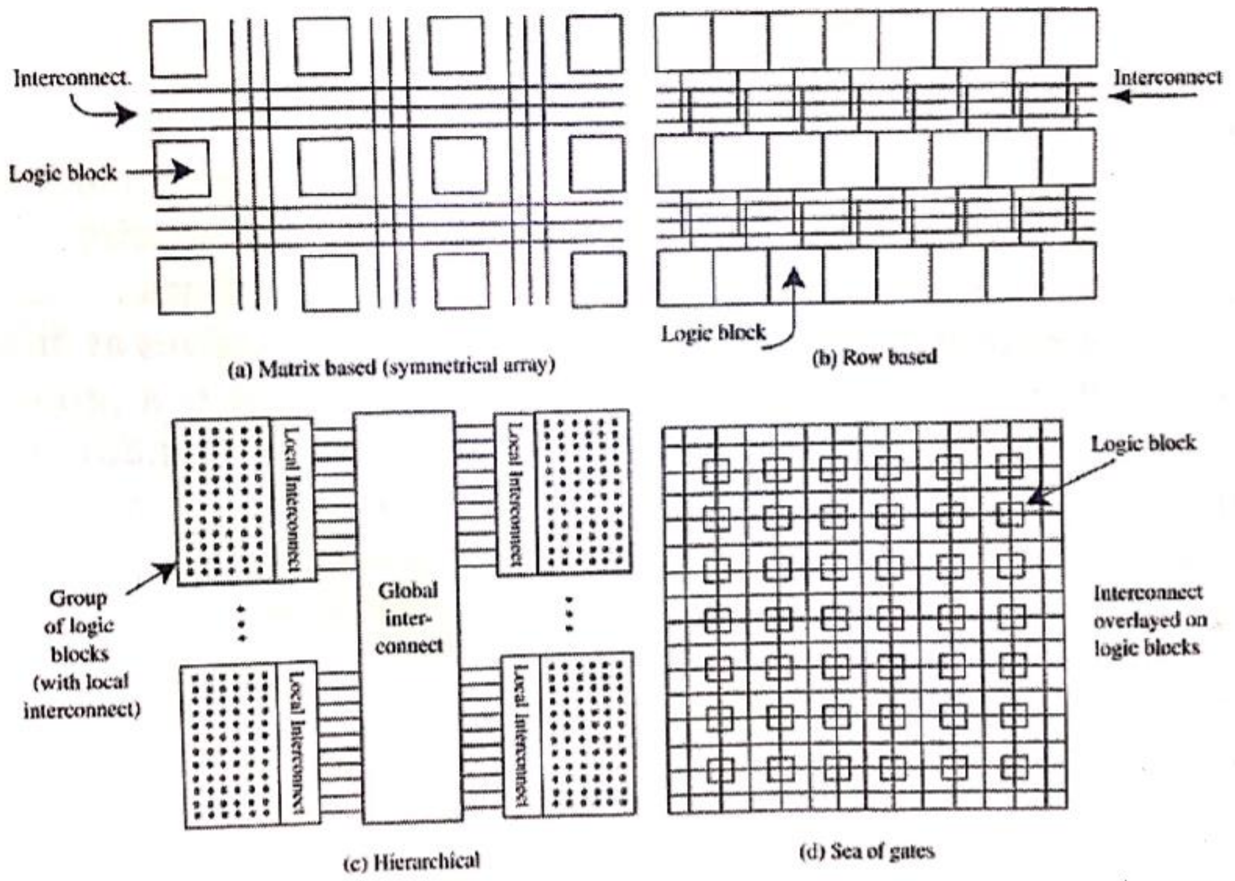
In this gate array like architecture, the logic blocks are arranged in rows with horizontal routing lines between adjacent rows. This is called one dimensional channeled routing.

<u>Hierarchical architectures:</u>

The logic cells are connected by local interconnect to form a group which in- turn is connected by a next level or global interconnect to form a logic array block.

Sea-of-gates architectures:

The interconnections between the gates in the FPGA is called sea-of-gates because of the numerous no. of gates in an FPGA.



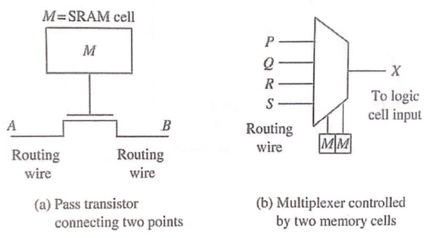
2.14 FPGA PROGRAMMING TECHNOLOGIES:

The collection of large no. of programmable logic blocks with programmable interconnect constitutes an FPGA. This helps to realize complex digital circuitry for testing and debugging the errors without physically changing the interconnection. The input signals can be changed to correct the errors. This highly effective reconfigurability is possible by changing the contents of static RAM or flash memory cells or fusing metal links.

Clock skew:

FPGA is such a large logic block, where the clocks connected to different logic blocks arrive at different times due to wiring delay. The difference in time or delay in the arrival of the clock edge is called clock skew. Thus clock distribution circuitry is designed with at-most care to minimize clock skew and maintain the strength of the clock amplitude. Otherwise, it will be a potential problem in FPGAs making it unreliable to work.

2.15 SRAM PROGRAMMING TECHNOLOGY:


M = SRAM cell
M
A
B
Routing wire
Routing wire
(a) Pass transistor connecting two points
P
Q
R
S
X
To logic cell input
Routing wire
M M
(b) Multiplexer controlled by two memory cells

In the above circuit we can note that M is made ‘1’ to connect A and B. This is called a memory cell. By using two memory cells, we can realize a 4 to 1 MUX. The memory cells are pass transistors with NMOS transistor which is ON when the gate signal is high, connecting source and drain by the inversion channel.

2.16 EPROM PROGRAMMING TECHNOLOGY:

The erasable programmable ROM , the transistor is made ON and OFF by the floating gate charged by high voltage and discharged by ultraviolet light. The floating gate is electrically isolated from the circuit, charged only by capacitance between floating gate and circuitry. Thus the signal value at gate maintains for long period of time, and is discharged when exposed to ultraviolet light.

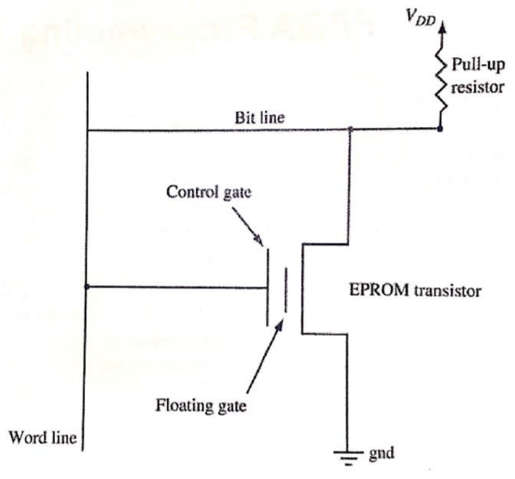
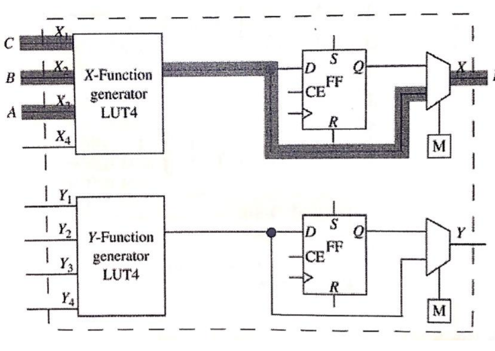


Fig.15 EPROM technology

2.17 PROGRAMMING LOGIC LOOK-UP TABLE BASED:

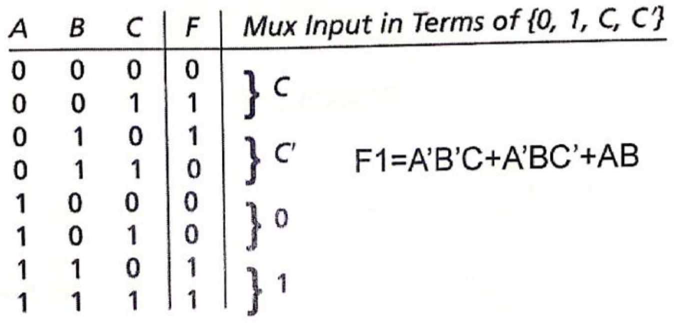
To design a logic circuit of the above function F1, we fed the inputs ABC to the LUT. The LUT generates the minterms and OR them to get the logic function. It is then fed into D flipflop to store it. Using the memory cell implementation of MUX, we can select the function F1.

F1=A’B’C+A’BC’+AB



2.18 LOGIC BLOCKS USING MUX:

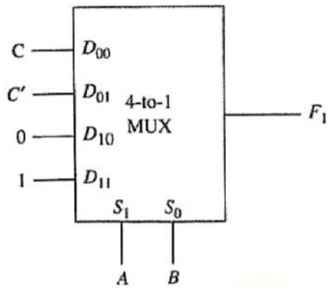
The logic blocks can easily be realized using MUX, by critically analyzing the truth table of the logic function. By doing so, we can fabricate the circuit with less no. of gates, leading to save chip area.



A	B	C	F	Mux Input in Terms of {0, 1, C, C’}
0	0	0	0	C
0	0	1	1	
0	1	0	1	C’
0	1	1	0	
1	0	0	0	0
1	0	1	0	
1	1	0	1	1
1	1	1	1	

F1=A’B’C+A’BC’+AB

Multiplexer Implementing Function F_1



For the function F1, we first create the truth table. The minterm A’B’C corresponds to 001 in the truth table and the output F is 1 in that case. Similarly all the minterms are considered and the truth table is completed. Now if we analyze the truth table, when AB=00, the output F=C, AB=10, then output F=0. Thus it is almost like a MUX function where AB are the select lines and C is the input. C’ is another input when F is the complement of C. To make the output 0 or 1, independent of C, we hardwired two inputs to 0 and 1 respectively. Likewise, we can implement wide variety of logic functions using MUX. The difficulty lies in analyzing the truth table and finding the appropriate input and select lines to the MUX. Once it is done, the logic circuit is easily implemented in MUX form.

2.19 INTERCONNECTS:

The interconnection between the logic blocks is done by three methods. The first one is the switch matrix, where the switches are used to establish connection between the logic blocks. Another simple method is using the direct connection between the logic blocks. But they are usually avoided in complex circuits, because they are not aligned and may lead to shorting. The last method is the most widely used global lines. All the logic blocks are connected to the common bus with a tristate buffer to establish and detach connection.

2.20 I/O STANDARDS:

*Stub series terminated logic(SSTC):*These devices are used to drive transmission lines like SDRAM modules in memories. These devices come with different voltage levels like 3.3v, 2.5v and 1.8v.

Accelerated graphics port: It is used to connect graphics card with mother board for high speed connection.

Gunning transceiver logic: It is the standard for bus connecting the i/o pads to provide high bandwidth of 100 MHz with less power swing and dissipation.

2.21 BCD to Seven-Segment Display Decoder:

Seven-segment displays are often used to display digits in digital counters, watches, and clocks. A digital watch displays time by turning on a combination of the segments on a seven-segment display. For this example, the segments are labeled as follows, and the digits have the forms as indicated in Figure below.

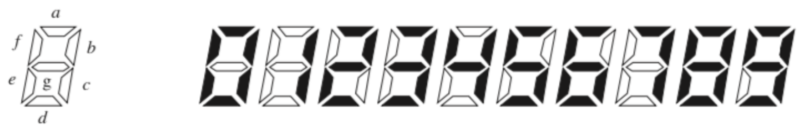


Fig.15: Seven-Segment Display

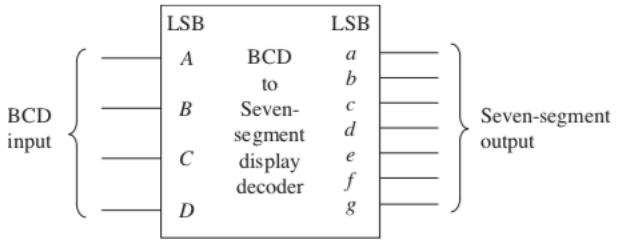


Fig.16: Block Diagram of a BCD to Seven-Segment Display Decoder

Let us design a BCD to seven-segment display decoder. BCD stands for binary- coded decimal. In this format, each digit of a decimal number is encoded into 4-bit binary representation. This decoder is a purely combinational circuit, and hence no state machine is involved here. A block diagram of the decoder is shown in the Figure. The decoder for one BCD digit is presented.

2.22 BCD Adder:

BCD adder A 4-bit binary adder that is capable of adding two 4-bit words having a BCD (binary-coded decimal) format. The result of the addition is a BCD-format 4-bit output word, representing the decimal sum of the addend and augend, and a carry that is generated if this sum exceeds a decimal value of 9. Decimal addition is thus possible using these devices.

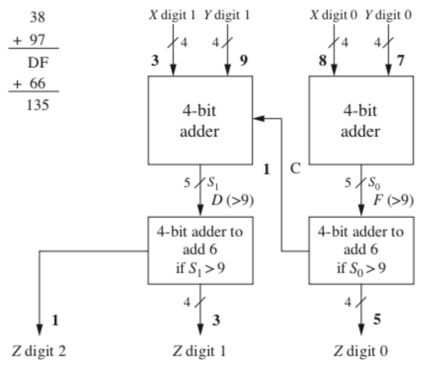


Fig.17: Addition of Two BCDNumbers

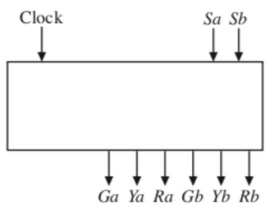
```
library IEEE;
use IEEE.numeric_bit.all;

entity BCD_Adder is
  port(X, Y: in unsigned(7 downto 0);
       Z: out unsigned(11 downto 0));
end BCD_Adder;

architecture BCDadd of BCD_Adder is
alias Xdig1: unsigned(3 downto 0) is X(7 downto 4);
alias Xdig0: unsigned(3 downto 0) is X(3 downto 0);
alias Ydig1: unsigned(3 downto 0) is Y(7 downto 4);
alias Ydig0: unsigned(3 downto 0) is Y(3 downto 0);
alias Zdig2: unsigned(3 downto 0) is Z(11 downto 8);
alias Zdig1: unsigned(3 downto 0) is Z(7 downto 4);
alias Zdig0: unsigned(3 downto 0) is Z(3 downto 0);
signal S0, S1: unsigned(4 downto 0);
signal C: bit;
begin
  S0 <= '0' & Xdig0 + Ydig0; -- overloaded +
  Zdig0 <= S0(3 downto 0) + 6 when S0 > 9
      else S0(3 downto 0); -- add 6 if needed
  C <= '1' when S0 > 9 else '0';
  S1 <= '0' & Xdig1 + Ydig1 + unsigned'(0=>C);
                           -- type conversion done on C before adding
  Zdig1 <= S1(3 downto 0) + 6 when S1 > 9
      else S1(3 downto 0);
  Zdig2 <= "0001" when S1 > 9 else "0000";
end BCDadd;
```

When BCD numbers are added, each sum digit should be adjusted to skip the six unused codes. For instance, if 6 is added with 8, the sum is 14 in decimal form. A binary adder would yield 1110, but the lowest digit of the BCD sum should read 4. In order to obtain the correct BCD digit, 6 should be added to the sum whenever it is greater than 9. Below Figure illustrates the hardware that will be required to perform the addition of two BCD digits. A binary adder adds the least significant digits. If the sum is greater than 9, an adder adds 6 to yield the correct sum digit and a carry digit to be added with the next digit. The addition of the higher digits is performed in a similar fashion.

2.23 Traffic Light Controller

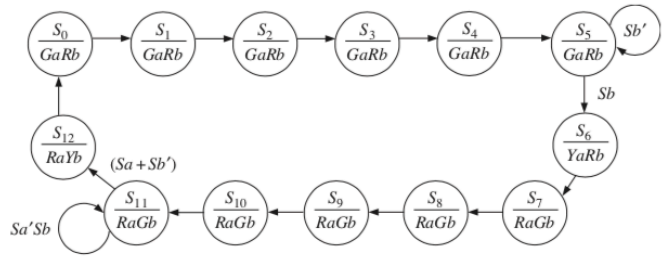
Each street has traffic sensors, which detect the presence of vehicles approaching or stopped at the intersection. Sa '1' means a vehicle is approach- ing on street A, and Sb '1' means a vehicle is approaching on street B. Street A is a main street and has a green light until a car approaches on B. Then the lights change, and B has a green light. At the end of 50 seconds, the lights change back unless there is a car on street B and none on A, in which case the B cycle is extended for 10 additional seconds. If cars continue to arrive on street B and no car appears on street A, B continues to have a green light. When A is green, it remains green at least 60 seconds, and then the lights change only when a car approaches on B. Figure 4-13 shows the external connections to the controller. Three of the outputs (Ga, Ya, and Ra) drive the green, yellow, and red lights on street A. The other three (Gb, Yb, and Rb) drive the corresponding lights on street B.

FIGURE 4-13: Block Diagram of Traffic Light Controller



Moore state graph for the controller. For timing purposes, the sequential circuit is driven by a clock with a 10-second period. Thus, a state change can occur at most once every 10 seconds. The following notation is used: *GaRb* in a state means that *Ga Rb* 1 and all the other output variables are 0. *Sa'Sb* on an arc implies that *Sa* 0 and *Sb* 1 will cause a transition along that arc. An arc without a label implies that a state transition will occur when the clock occurs, independent of the input variables. Thus, the green A light will stay on for six clock cycles (60 seconds) and then change to yellow if a car is waiting on B street.

FIGURE 4-14: State Graph for Traffic Light Controller



The VHDL code for the traffic light controller (Figure 4-15) represents the state machine with two processes. Whenever the state, *Sa*, or *Sb* changes, the first process updates the outputs and *nextstate*. When the rising edge of the clock occurs, the sec- ond process updates the state register. The case statement illustrates use of a **when** clause with a range. Since states S_0 through S_4 have the same outputs, and the next states are in numeric sequence, we use a **when** clause with a range instead of five separate **when** clauses:

when 0 **to** 4 => Ga <= '1'; Rb <= '1'; nextstate <= state + 1;

```
entity traffic_light is
  port(clk, Sa, Sb: in bit;
       Ra, Rb, Ga, Gb, Ya, Yb: inout bit);
end traffic_light;

architecture behave of traffic_light is
signal state, nextstate: integer range 0 to 12;
type light is (R, Y, G);
signal lightA, lightB: light;  -- define signals for waveform output
begin
  process(state, Sa, Sb)
  begin
    Ra <= '0'; Rb <= '0'; Ga <= '0'; Gb <= '0'; Ya <= '0'; Yb <= '0';
    case state is
      when 0 to 4 => Ga => '1'; Rb => '1'; nextstate => state+1;
      when 5 => Ga <= '1'; Rb <= '1';
        if Sb = '1' then nextstate <= 6; end if;
      when 6 => Ya <= '1'; Rb <= '1'; nextstate <= 7;
      when 7 to 10 => Ra <= '1'; Gb <= '1'; nextstate <= state+1;
      when 11 => Ra <= '1'; Gb <= '1';
        if (Sa='1' or Sb='0') then nextstate <= 12; end if;
      when 12 => Ra <= '1'; Yb <= '1'; nextstate <= 0;
    end case;
  end process;
  process(clk)
  begin
    if clk'event and clk = '1' then
      state <= nextstate;
    end if;
  end process;
  lightA <= R when Ra='1' else Y when Ya='1' else G when Ga='1';
  lightB <= R when Rb='1' else Y when Yb='1' else G when Gb='1';
end behave;
```

For each state, only the signals that are '1' are listed within the case statement. Since in VHDL a signal will hold its value until it is changed, we should turn off each signal when the next state is reached. In state 6 we should set *Ga* to '0', in state 7 we should set *Ya* to '0', and so on. This could be accomplished by insert- ing appropriate statements in the when clauses. For example, we could insert

Ga <= ‘0’ in the **when** 6 => clause.

An easier way to turn off the outputs is to set them all to ‘0’ before the case statement, as shown in Figure 4-15. At first, it seems that a glitch might occur in the output when we set a signal to ‘0’ that should remain ‘1’. However, this is not a problem because the sequential statements with- in a process execute instantaneously. For example, suppose that at time 20 ns a state change from S_2 to S_3 occurs. *Ga* and *Rb* are ‘1’, but as soon as the process starts executing, the first line of code is executed and *Ga* and *Rb* are scheduled to change to ‘0’ at time 20 . The case statement then executes, and *Ga* and *Rb* are scheduled to change to ‘1’ at time 20 . Since this is the same time as before, the new value (‘1’) preempts the previously scheduled value (‘0’), and the signals never change to ‘0’.

Before completing the design of the traffic controller, we will test the VHDL code to see that it meets specifications. As a minimum, our test sequence should cause all of the arcs on the state graph to be traversed at least once. We may want to perform additional tests to check the timing for various traffic condi- tions, such as heavy traffic on both A and B, light traffic on both, heavy traffic on A only, heavy traffic on B only, and special cases such as a car failing to move when the light is green, a car going through the intersection when the light is red, and so on.

To make it easier to interpret the simulator output, we define a type named light with the values *R, Y*, and *G* and two signals, *lightA* and *lightB*, which can assume these values. Then we add code to set *lightA* to *R* when the light is red, to *Y* when the light is yellow, and to *G* when the light is green. The following simulator com- mand file first tests the case where both self-loops on the graph are traversed and then the case where neither self-loop is traversed:

```
add wave clk SA SB state lightA lightB
force clk 0 0, 1 5 sec -repeat 10 sec
force SA 1 0, 0 40, 1 170, 0 230, 1 250 sec
force SB 0 0, 1 70, 0 100, 1 120, 0 150, 1 210, 0 250, 1 270 sec
```

The test results in Figure 4-16 verify that the traffic lights change at the specified times.

2.24 Array Multiplier:

An array multiplier is a parallel multiplier that generates the partial products in a parallel fashion. The various partial products are added as soon as they are avail- able. Consider the process of multiplication as illustrated in Table 4-3. Two 4-bit unsigned numbers, $X_3X_2X_1X_0$ and $Y_3Y_2Y_1Y_0$, are multiplied to generate a product that is possibly 8 bits. Each of the X_iY_j product bits can be generated by an AND gate. Each partial product can be added to the previous sum of partial products using a row of adders. The sum output of the first row of adders, which adds the first two partial products, is $S_{13}S_{12}S_{11}S_{10}$, and the carry output is $C_{13}C_{12}C_{11}C_{10}$. Similar results occur for the other two rows of adders. (We have used the notation S_{ij} and C_{ij} to represent the sums and carries from the *i*th row of adders.)

TABLE 4-3: Four-bit Multiplier Partial Products

				X_3	X_2	X_1	X_0	Multiplicand
				Y_3	Y_2	Y_1	Y_0	Multiplier
				X_3Y_0	X_2Y_0	X_1Y_0	X_0Y_0	Partial product 0
			X_3Y_1	X_2Y_1	X_1Y_1	X_0Y_1		Partial product 1
			C_{12}	C_{11}	C_{10}			First row carries
		C_{13}	S_{13}	S_{12}	S_{11}	S_{10}		First row sums
		X_3Y_2	X_2Y_2	X_1Y_2	X_0Y_2			Partial product 2
		C_{22}	C_{21}	C_{20}				Second row carries
	C_{23}	S_{23}	S_{22}	S_{21}	S_{20}			Second row sums
	X_3Y_3	X_2Y_3	X_1Y_3	X_0Y_3				Partial product 3
	C_{32}	C_{31}	C_{30}					Third row carries
C_{33}	S_{33}	S_{32}	S_{31}	S_{30}				Third row sums
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0	Final product

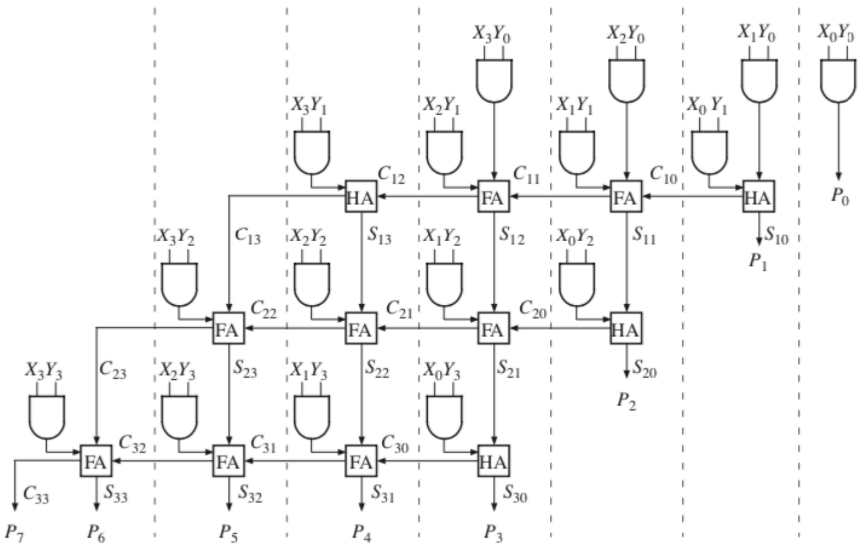


FIGURE 4-29: Block Diagram of 4 × 4 Array Multiplier

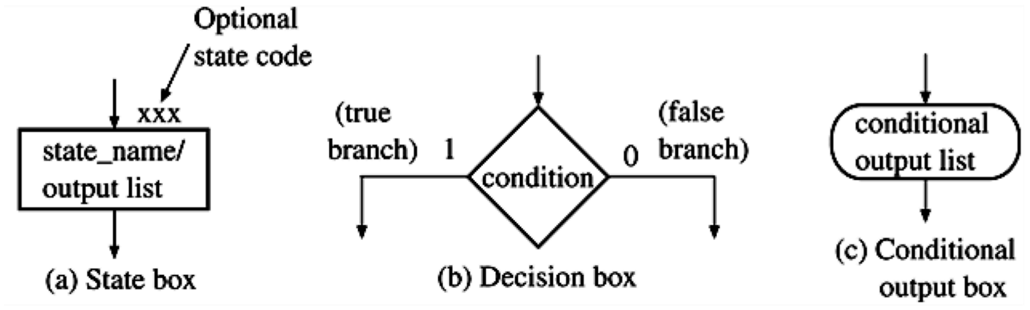
2.25 State Machine Charts:

SM charts resemble software flow charts. Flow charts have been very useful in soft- ware design for decades, and in a similar fashion, SM charts have been useful in hardware design. This is especially true in behavioral-level design entry.

SM charts offer several advantages over state graphs. It is often easier to under- stand the operation of a digital system by inspection of the SM chart instead of the equivalent state graph. A proper state graph has to obey some conditions: (1) One and exactly one transition from a state must be true at any time, and (2) the next state must be uniquely defined for every input combination. These conditions are automatically satisfied for an SM chart. An SM chart also directly leads to a hard- ware realization. A given SM chart can be converted into several equivalent forms, and different forms might naturally result in different implementations. Hence, a designer may optimize and transform SM charts to suit the implementation styletechnology that he or she is looking for.

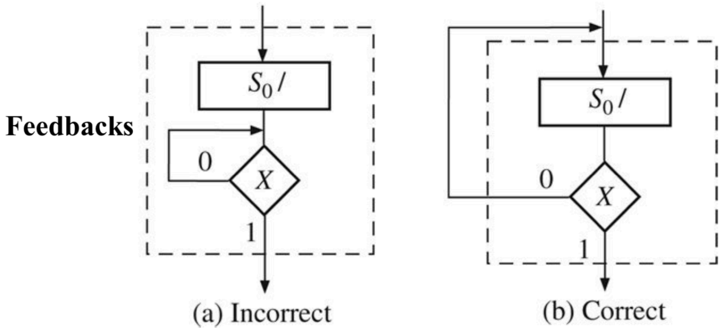
An SM chart differs from an ordinary flow chart in that certain specific rules must be followed in constructing the SM chart. When these rules are followed, the SM chart is equivalent to a state graph, and it leads directly to a hardware realization.

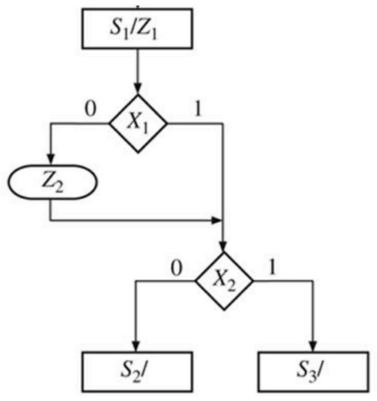
Principle Components of State Machine Chart:



State Machine Chart Rules

- A state consists of 1 state box and all the other boxes coming from it (until another state box is encountered)
- There must be exactly 1 exit path active for a state at any time, no matter what the input
- No internal loops (feedbacks)
- All paths are traversed simultaneously
- Outputs on active paths or in the state box are asserted; all other outputs are de-asserted





3. APPLICATIONS

Applications of FPGAs:

FPGAs have become a popular mode of circuit implementation for various applications:

Rapid Prototyping

FPGAs are very useful for building rapid prototypes of large systems. A designer can build proof-of-concept systems very quickly using field programmable gate arrays. Since FPGAs are large enough to contain 5 million or more gates, many large real-world systems can be prototyped using a single FPGA. If a single FPGA will not suffice, multiple FPGAs can be interconnected to realize large systems. Rapid prototyping of large systems is done by using boards with multiple FPGAs and plugging multiple boards into a backplane (motherboard).

As Final Product in Medium-Speed Systems

Circuits realized using FPGAs typically operate in the 150–200-MHz clock rate. For applications where this speed is sufficient, FPGAs can be used for the final product itself as opposed to the prototype. When an FPGA is used as the final product, enhancements to the system can be done as software updates rather than hardware changes. Modern FPGA speeds are adequate for many applications.

Reconfigurable Circuits and Systems

The reprogrammability of FPGAs lends itself to building dynamically reconfigurable circuits and systems. SRAM-based FPGAs make it possible to implement "soft" hardware. FPGAs have been used to design circuits and systems that need multiple functionalities at various times.

As an example, consider a reprogrammable Tomahawk missile that the Navy designed using FPGAs. [46] The conventional Tomahawk is a long-range Navy cruise missile designed to perform a variety of missions. The Navy designed a reconfigurable Tomahawk, which can operate in one of two modes, depending on the mission at hand. Rather than designing separate logic for each mode, the missile designers used FPGAs so that the configuration for each mode can be kept on-board in ROM. Depending on the mode of operation, the FPGA could be configured in midflight.

Glue Logic

FPGAs have become the medium of choice for implementing interface or glue logic between modules and components. Small changes in interface protocols or formats would conventionally necessitate building new interface logic. With SRAM FPGAs, the new interface logic can be implemented on the same FPGA as in a software update.

Hardware Accelerators/Coprocessors

A software application running on a conventional system can be accelerated if a coprocessor/ accelerator can implement some key routines/kernels from the applica- tion in hardware. An FPGA can be used to implement the key kernel. A SRAM-based, reconfigurable FPGA is well suited for this type of use because depending on the application running, different kernels can be dynamically programmed into the FPGA. This approach has been demonstrated for applications, such as pattern matching. FPGA-based hardware is used for several applications, including computer architecture simulator acceleration, emulation boards, hardware test/verification, and so on.

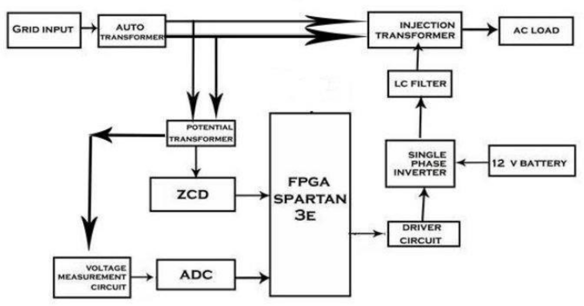
3.1 FPGA BASED GRID-TIE INVERTER:

An inverter is an electronic circuit which consists of four power MOSFETs or IGBTs to convert DC to AC. Normally in a battery, the voltage is stored as DC voltage. But when the load connected to the AC grid is higher, the smart systems draws current from the battery to avoid drop in voltage. Unfortunately, the voltage in battery is DC voltage but the voltage from grid is AC. All the appliances work with AC only. Thus we inevitably require an inverter which converts DC to AC.

This AC voltage from the inverter has to be in-phase with normal AC grid voltage to avoid shorting. This imposes the condition that the grid AC and inverter AC should be in-phase for proper functioning of the appliances. Thus we use FPGA SPARTAN 3E kit to generate PWM waves, which will switch the MOSFETs in full bridge inverter for converting DC voltage in batteries to AC voltage, in-phase with normal AC grid.

The grid AC voltage is taken as the reference to generate PWM waves. The AC grid wave is passed through a ZCD to get the information about when the AC voltage reaches zero and starts a new cycle. The same AC grid wave is passed through an ADC to convert it into 12 bit values. With these two inputs, the FPGA generates PWM signals.

The PWM signals do not go straight to the inverter. It passes through the driver circuit, which ensures protection to the inverters. Eventually, the DC voltage from the battery is converted to AC voltage which will be added to the grid AC voltage. Thus the extra voltage from the solar panels can be used to counteract the voltage drop due to heavy load or to use it as a back-up during power cuts.



The advantage of using FPGA is its reprogramming ability and cost. Using oscillators for PWM generation is a tedious process and we don't have full control over the functioning.

3.2 RF-ID BASED ILLEGAL PARKING DETECTOR:

The traffic at big cities are high. The cars parked in the no parking area creates more congestion to road transport. Everyday, there are millions of cars parked at no parking zone creating much disturbance for transiting. This new system provides a way to eradicate and send alert to the drivers when they parked in no parking zone.

A large number of personnel are deployed to check for unauthorized parking and fine those owners. Also, Towing vans need to manually search for illegally parked vehicles. This system requires large overhead costs in manpower payment, fuel and other physical surveillance. Here we propose a system that allows for automatic illegal parking detection and alerting. The system consists of integrating an RFID transmitter in every vehicle. RFID receiver circuit is mounted on every area where parking is prohibited. If a vehicle is parked in an area where parking is prohibited, the rfid transmitter comes in range of the receiver circuit. Once this happens the rfid reader reads the transmitter id with mobile no. stored in RF-id and can instantly send text message to the owner using GSM module to alert them. This also includes one IR sensor to detect the vehicle, incase no rfid is placed in that vehicle. During that time, the police can come to that place and tow the vehicle. The place can be identified by GPS module integrated to the micro controller.

The above processor consists of the micro controller arduino, to integrate RF- id reader, IR sensor and GSM module. The simple arduino code is written to make this system work. The entire system is low cost with multiple purposes. The same RF-id can be used for many other detections about the vehicles.

4. CONCLUSIONS

A thorough, in depth understanding of hardware description languages as well as a solid understanding of how various programmable logic devices operate internally is essential to successful embedded systems design. In real world use cases, the implementation of complex functionality, such as FFT or other complex mathematical functions will potentially be required. While these cases are far more complex than the cases we have studied during the limited time available during these lectures, the same concepts governing hardware description languages still apply and will need to be heavily relied upon to implement and debug such systems. Additionally, when implementing highly performance critical functionality, manual optimization may become necessary in order to achieve the required levels of performance. Whether these optimizations are the implementation of all or part of the logic at the structural level or supplying additional timing or placement constraints to the synthesis tools, a solid understanding of the internal workings of programmable logic devices is required.

5. REFERENCES

1. Digital Systems Design Using VHDL, *Charles H. Roth, Lizy Kurian John*. 2008 Thomson Learning.

2. FPGA-Based Configurable Frequency-Diverse Ultrasonic Target-Detection System, Joshua Weber, Erdal Oruklu, and Jafar Saniie. IEEE TRANSACTIONS ON INDUSTRIAL ELECTRONICS, VOL. 58, NO. 3, MARCH 2011

3. *Automated Vehicle Parking System And Unauthorized Parking Detector*, Ishraq Haider Chowdhury, International Conference on Advanced Communications Technology(ICACT). http://icact.org/upload/2018/0621/20180621_finalpaper.pdf

4. https://faculty.weber.edu/snaik/ECE3610/09Lec9.pdf

5. *Design a grid tie inverter for PMSG wind turbine using FPGA & DSP builder*, Ilhami Colak ; Eklas Hossain ; Ramazan Bayindir ; Jakir Hossain, https://ieeexplore.ieee.org/document/7752026

6. https://semiwiki.com/efpga/achronix/7541-when-why-and-how-should-you-use-embedded-fpga-technology/

7. https://en.wikipedia.org/wiki/Wire_recording

8. http://www.xilinx.com/video/software/sdsoc-developing-in-c-and-c-plus-plus.html

9. https://www.xilinx.com/applications/smarter-vision.html

10. https://www.youtube.com/watch?v=u0bW6lQvsVI

11. https://hackaday.io

12. http://ww1.microchip.com/downloads/en/DeviceDoc/31002a.pdf

13. https://www.quora.com/How-do-artificial-neural-networks-learn

14. https://en.wikipedia.org/wiki/Spiking_neural_network

15. https://www.quora.com/How-do-artificial-neural-networks-learn