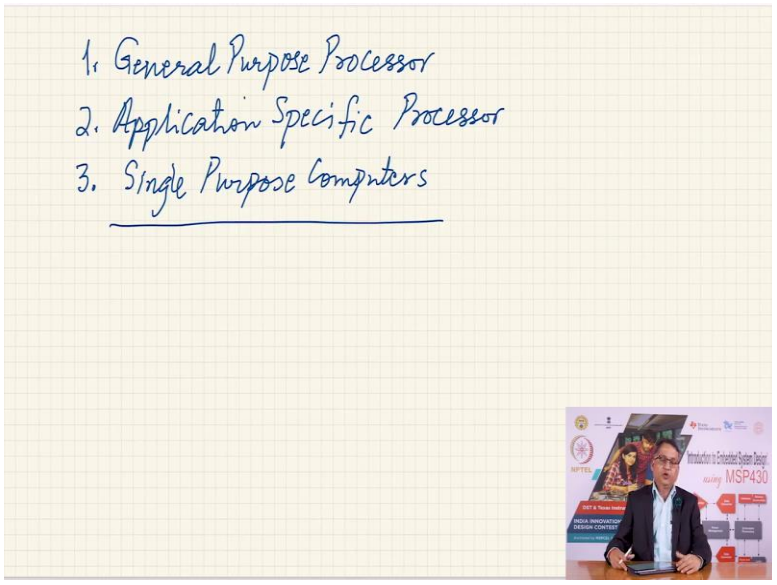
Introduction to Embedded System Design

Professor. Dhananjay V. Gadre
Netaji Subhas University of Technology
Professor. Badri Subudhi
Indian Institute of Technology, Jammu

Module 12
Lecture 38
Single Purpose Computers

Hello and welcome to a new session for this online course on Introduction to Embedded System Design. I am your instructor Dhananjay Gadre. Today we are going to look at a very different topic. Although we mentioned this topic in the first lecture, where we were talking about various ways of implementing the embedded computers and to revise that or to mention what we had repeat what we had mentioned, we had said that there are 3 ways of implementing an embed computer.

(Refer Slide Time: 0:57)



1. General Purpose Processor
2. Application Specific Processor
3. Single Purpose Computers

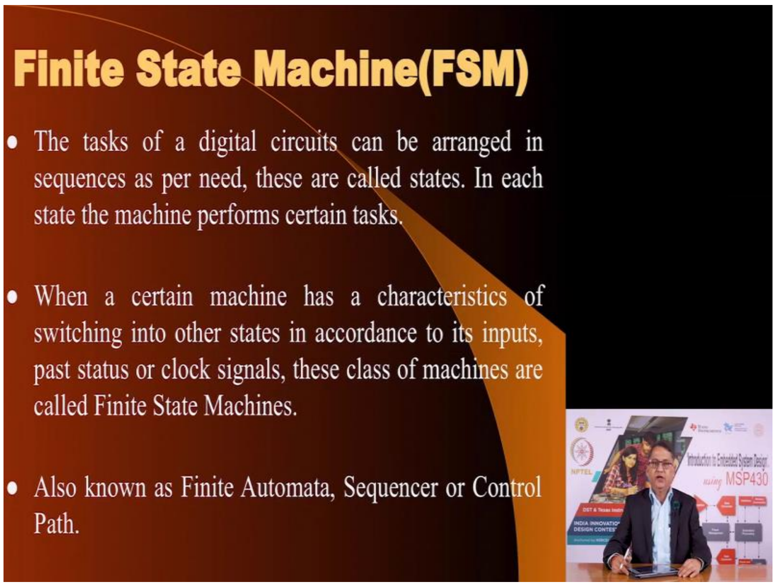
The first method is using a general general purpose computer plus processor, the second is to use an application application specific processor. For the first approach you might be using commonly available microprocessor such as 8086, 68000 and so on. For the second approach, we will be using a microcontroller which is what we are doing in this course. But we had mentioned a third approach and this is this was called single purpose computers, which means computers which are designed with only one objective in mind which is to solve the current problem at hand, it cannot do anything else. If you want to modify it, you will have to redesign everything.

Usually the single purpose computers are implemented using digital logic. And it follows paradigm which we also showed how that paradigm of looking at the computational model applies to the general purpose processor, as well as the application specific processor. That paradigm included 2 building blocks. One was called the sequencer or the controller, and the second was called the data path.

The sequencer decided which elements of the data path to be used for what computational tasks, the elements of data path would perform their tasks, process data as they are capable of, and convey the result or parts of the result to the sequencer to decide the next course of action.

In this lecture here, I am going to show you a sample project implemented using such an approach. The heart of a single purpose computer is the finite state machine. This finite state machine is a computational model with which you can create sequences. It uses a concept called states and outputs and conditions of transition.

(Refer Slide Time: 3:28)

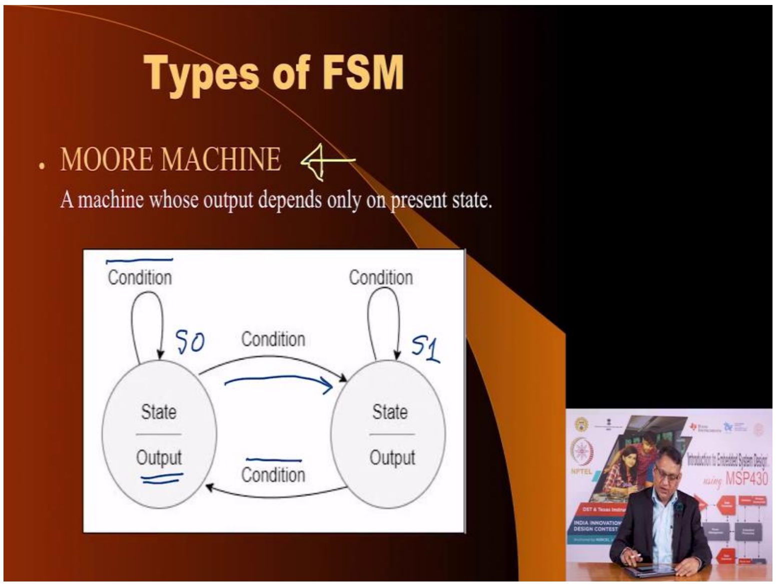


Finite State Machine(FSM)

- The tasks of a digital circuits can be arranged in sequences as per need, these are called states. In each state the machine performs certain tasks.
- When a certain machine has a characteristics of switching into other states in accordance to its inputs, past status or clock signals, these class of machines are called Finite State Machines.
- Also known as Finite Automata, Sequencer or Control Path.

When a certain machine has characteristics of switching into other states in accordance with its inputs, past status, clock signals, these class of machines are called finite state machines, they are known by other terms also such as finite automata sequencer as we will be using that term often in this lecture or control path.

(Refer Slide Time: 3:52)



Types of FSM

- MOORE MACHINE

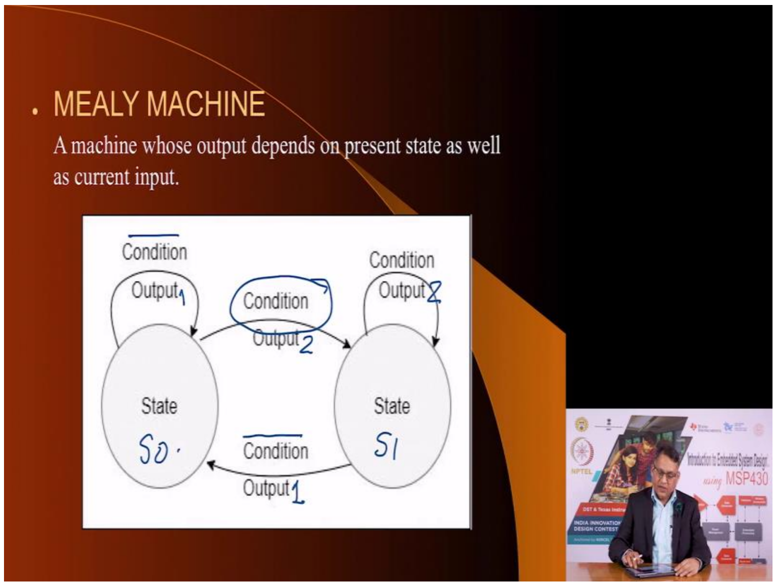
  A machine whose output depends only on present state.

Now broadly, there are 2 types of finite state machines. One is called the MOORE machine here. The characteristics of this MOORE machine is that the machine output depends only on the current state. So, the representation of the states, outputs and the transition conditions are illustrated using a diagram called state diagram.

So, here in this machine, there are 2 states, one is we could we could call it as 0, state 0 and another is state 1. And if you are in state 1, state 0, then it would produce certain outputs which you can define here. We are just given a label output, but you can mention what all outputs you have, what is the values of those outputs 0, 1 whatever.

And then when a certain condition is true, here, it should be that if the condition is say not true, then it will remain in the state but if the condition becomes true, it will transition to S 1. Similarly, we could make it that if the condition is true, it remains in the state, but if the condition becomes false, so I have used a condition bar, it will transition from S 1 to s 0. This is the complete description of a MOORE state MOORE type of finite state machine. There is another method of modelling state machines and that is called Mealy machine.

(Refer Slide Time: 5:28)



- MEALY MACHINE

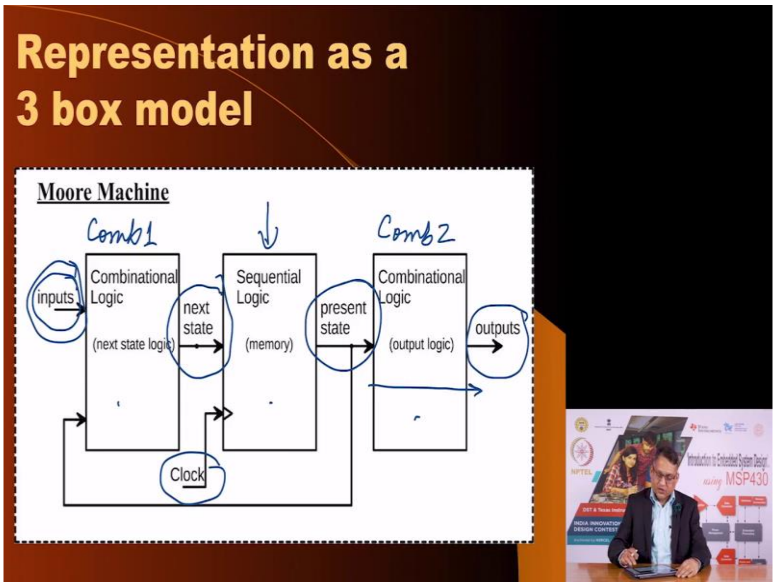
A machine whose output depends on present state as well as current input.

In Mealy machine, we have the states are still the same, but now, the outputs have moved from within these states, these circles which represent the state on the edges which represents the condition and again I will have to correct this, here is the condition bar and if there is condition, then say I can say output one, this is output 2, if it is condition you remain in so, this is state S 0, this is state S 1 and for condition 1 and you could have output 2 here, it continues to produce output 2. And if it goes back here it could be output 1 again and it goes back into state S 0.

This is the graphical representation of any finite state machine. As you can see, it uses 2 symbols, one is a circle to represent the state and one is these arrows which represent the transition conditions. These transition conditions are basically inputs, which when they change, they transition from one state to another state.

However, one must also keep in mind that if these finite state machines are implemented using a synchronous logic, then the transition happens on 2 conditions. One is this condition being checked, but also at the rising or falling edge of the clock whichever may be the case in your implementation. So, it requires 2 conditions to be true, the physical condition that is the input to have a certain value and the transition of a clock from 0 to 1, then only it will transition from state S 0 to S 1, this must be kept in mind. It is not mentioned here, but this is what is the assumption.

(Refer Slide Time: 7:27)



Representation as a 3 box model

The physical representation of such a state machine can be done by using 3 building blocks. One of them is the memory as you see here, this is the memory, this is the combinational logic and this also is so this I would say combinational logic 1 and this is combinational logic 2. The combinational logic receives the inputs, these inputs are external inputs, which means we call them asynchronous inputs, which means they are coming from an outside world outside of this system.

And therefore, they can change whenever they wish to. They are applied to this combinational logic block. This combinational logic block produces an output, which becomes the input of the memory. And so since this is the value that is currently known, this is called next state I will come to that in a moment.

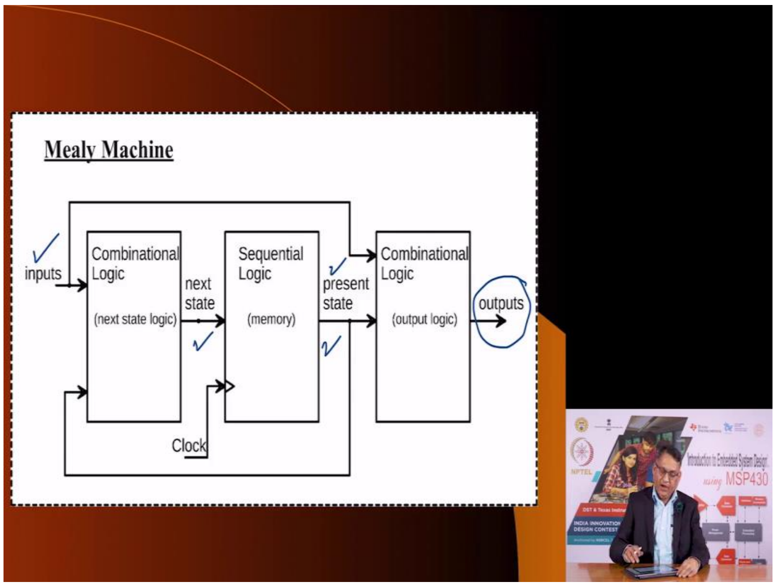
The output of the sequential logic which is which could be implemented with D type flip flops is passed through another combinational logic block and that produces the output. And since we said in the Moore machine, the outputs are dependent only on the present state. So, this is your present state. The combinational logic does not change the state, it simply transforms these present state values into expected output.

And these are the outputs that you mentioned in the circles that represented each of these states. These present state values also are fed back into the first combinational block and together with the inputs they decide the next course of action that is, it provides next state inputs.

And as you see, the memory is clocked and this clock is locally generated and the rate of this clock will decide how quickly the state machine can transition. All of this can be implemented using logic circuit, this is combinational logic, this is memory that is flip flops and this is another set of combinational logic.

Now, in this block diagram, if you want to represent a Mealy machine, all you have to do is the inputs now are able to affect the outputs, not only the outputs are affected by the current state, but current inputs may also modify the outputs.

(Refer Slide Time: 9:53)

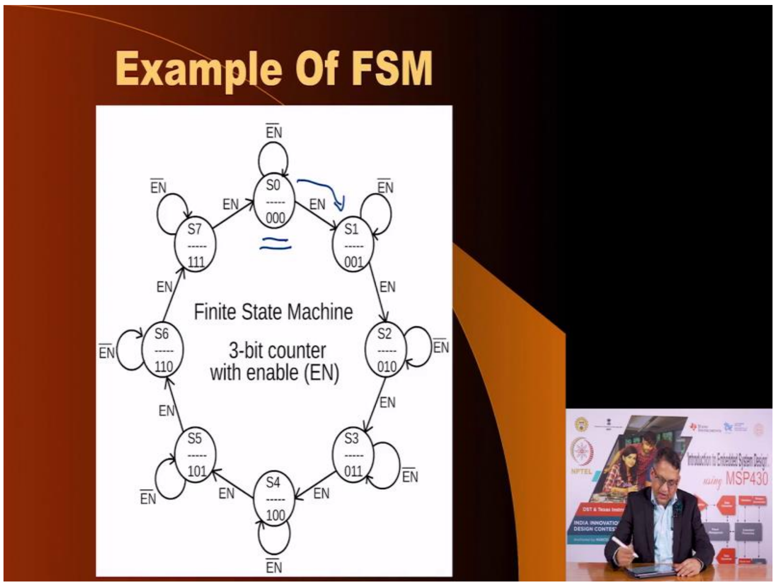


And this is represented in this second diagram here. And you see, the rest of everything remains the same, this is my next state. This is my present state, that present state is being fed back into the combinational logic, but some of some or all the inputs are being fed into the output combinational logic and therefore, these outputs are dependent not only on the present state, but also on the present inputs.

And these are two alternative methods of implementing the finite state machine using the Mealy model or the MOORE model. Now, when you probably would study this as part of digital circuits and systems, and they would talk about various examples, and I want to take one example, what are simple examples that you already know?

A counter is an example of a finite state machines, it has finite number of states, which can be limited by the will be limited by the maximum number of flip flops being used in that counter. And so, the state diagram can be drawn to represent the transition of the states that is how the counter is working? Is it counting up or is it counting down? Or is there any sequence, up or down sequence or you want to change that sequence, you can create a counter with arbitrary outputs also. Outputs that do not seem to have a pattern of being increasing or decreasing, you can choose that.

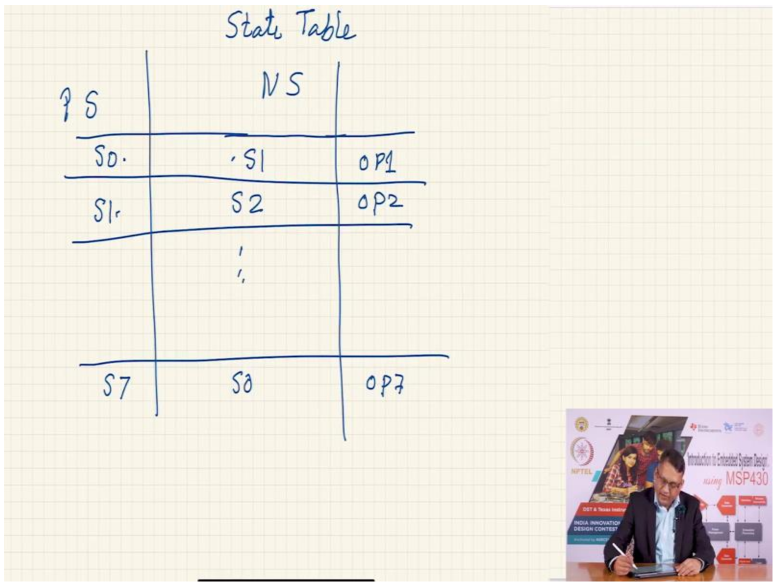
(Refer Slide Time: 11:22)


Example Of FSM

And let me show you an example of a finite state machine. Essentially, it is a counter except with the little addition that I have added an enable signal. If the enable signal is 1, it transition from one state to the other. If the enable signal is held to 0, it will retain the current state, and if it retains the current state, it will keep on producing the outputs as mentioned in the state circle in the symbol that represents the state.

Now, how do you implement such a finite state machine? How do you implement that counter? Now you can actually draw from the state diagram you can create a state table, in the state table you will create a table, let me show you what that table would look like.

(Refer Slide Time: 12:10)



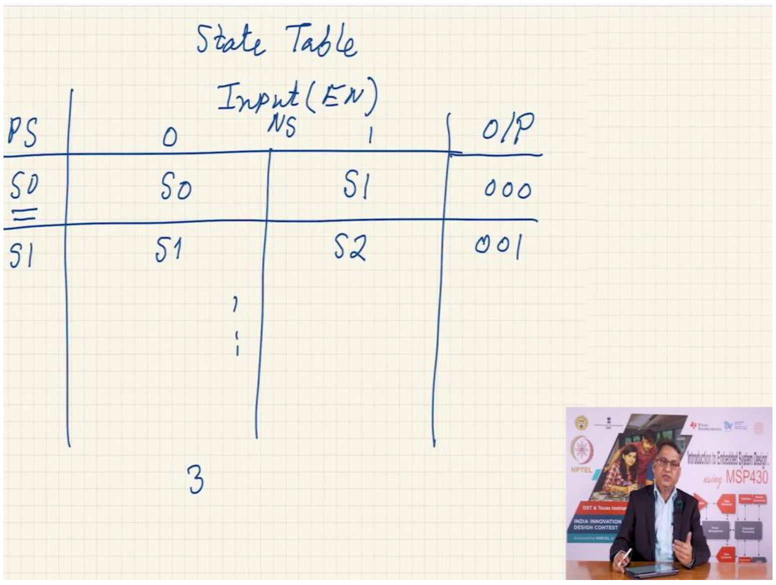
State Table

PS	NS	
S0	S1	OP1
S1	S2	OP2
	⋮	
S7	S0	OP7

In the table you will create what is your present state, we call it PS, what is your next state and what are the outputs. And if it is a MOORE machine, then the outputs will can be mentioned as a function of the states only. So, if I know my present state S 0, I know my output so maybe output 1 and my next state could be S 1 and if my present state is S 1, this the next state is S 2 and you can have output 2 and so on, and eventually it will somewhere it will say let us say s 7 and I want to go to S 0 and output will be O P 7.

Now with this table I have actually represented a counter with no enable signal, why? Because whenever you are in state S 0, the clock will take you to S 1 if you are in S 1 it will take to S 2, there is no condition to hold the current state. If we want to modify it to be representative, this is the state diagram. This is the state table from the state diagram we drew earlier, if you want to create the state table for the earlier state diagram, then it will have to have the following representation.

(Refer Slide Time: 13:41)



State Table

Input (EN)

PS	NS 0	NS 1	O/P
S0	S0	S1	000
S1	S1	S2	001
	⋮		

3

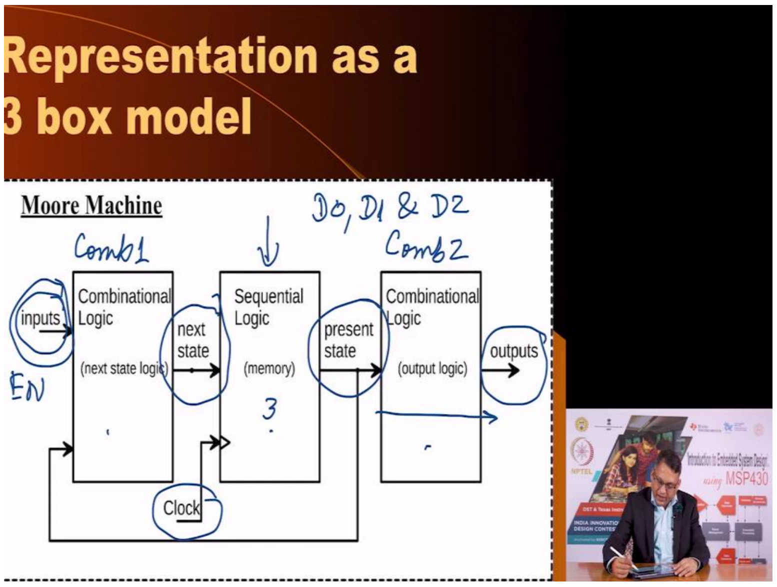
So, this is a state table, here you will write present state and if the input is enabled, if it is 0 and if it is 1 because there is only one input what will be the next state, and here is your output. So if the present state is S 0, if the enable signal is 0 is the next state will be S 0. However, if the enable signal is 1, the next state will be S 1, but if you know as 0, you can say I want the output to be 000. If the present state is S 1 and if enable signal is 0, it will retain that state and if it is 1, it will go to the next state and I can have my output 001 and so on and so forth.

Now once you create the state table, you know the relationship between the inputs and outputs of your first combinational block, the relationship between inputs and outputs of your second combinational block and the middle block is just few flip flops and we can decide how many flip flops to use. One method is for the total number of states that we have, we take log 2. So like we have 8 states, we take log 2, you get 3. So, all you have to do is use 3 flip flops, this is the encoded method of representing the states.

The other option is for all the states you have one flip flop, and that is called one hot method of encoding representing the state, we are not going to go there, it may appear that one hot encoding uses more memory. But the advantage of that oftentimes is that when you use more memory, your decode logic that is the first block of combinational circuit, maybe even the second block of combinational circuit, the complexity of that circuit will simplify.

So, it is a trade off. If you use the encoded method that is used less number of flip flops, your combination circuit might increase. And so you decide whether you want to use the encoded method of State representation or the one hot method. Anyway, this can be manually implemented meaning you could create a table which which implements the relationship here between inputs.

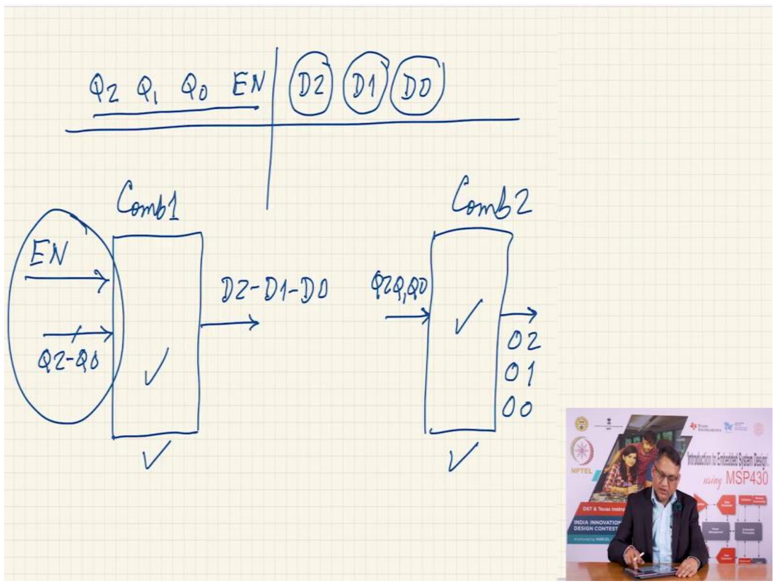
(Refer Slide Time: 16:17)



Representation as a 3 box model

Now, your input here is enable as well as the present states and you are producing next state. So, if I use the encoded method, the number of flip flops will be 3 therefore, you have to deal with 3 d type inputs to the flip flops, we can label them D 0, D 1 and D 2. The outputs of these flip flops will be Q 0, Q 1 and Q 2 and these outputs here they go into the combinational circuit including the signal enable.

(Refer Slide Time: 16:57)



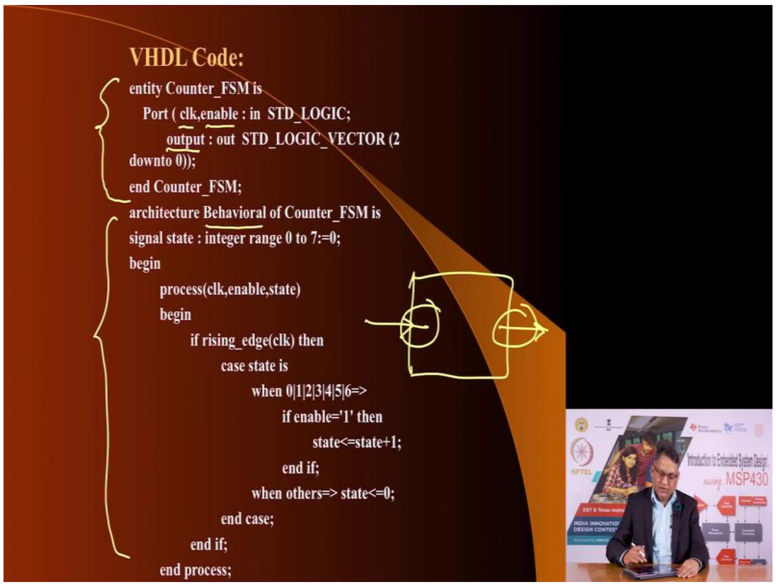
So, you have to create a table which has Q 2, Q 1, Q 0 and enable to produce outputs which is D 2, D 1 and D 0. Which means it is a matter of implementing 4 combinational circuits with 16 states here, these 16 states will produce one output the same 16 states will also produce another output and the third output and that resultant combination logic circuit will go in the first combinational block one. So, here you are putting E N enable, we are also putting Q 2, and it is producing D 2, D 1 and D 0.

So, implementing finite state machine is an exercise in starting from creating the state diagram from there to create the state table. That state table will give you the relationship between inputs of the combinational circuit and the outputs, it will also give you another relationship for the second building block combinational circuit 2. The inputs to this will be Q 2, Q 1 and Q 0 and the outputs will be say O 2, O 1and O 0.

So, you solve this combinational circuit using minimization techniques that you are aware of, you solve this combinational circuit using the same approach and the middle path is just sufficient number of D type flip flops. So you see, solving implementing a finite state machine is all about implementing a couple of combinational circuits. Now, this is possible manually using a pen and paper method.

If the number of inputs on your combination circuit are few meaning 4 or 5 or 6, because with four inputs you can draw a K map, but as the number of inputs increase, then you may have to resort to other methods. And in the contemporary world, this is only for learning purpose, real implementation is done with the help of representing the requirements of this state machine as a hardware description language code. So, I am going to show you that for this counter what kind of VHDL code is implemented.

(Refer Slide Time: 19:41)



VHDL Code:

```
entity Counter_FSM is
   Port ( clk,enable : in  STD_LOGIC;
        output : out  STD_LOGIC_VECTOR (2
downto 0));
end Counter_FSM;
architecture Behavioral of Counter_FSM is
signal state : integer range 0 to 7:=0;
begin
      process(clk,enable,state)
      begin
            if rising_edge(clk) then
                  case state is
                        when 0|1|2|3|4|5|6=>
                              if enable='1' then
                                    state<=state+1;
                              end if;
                        when others=> state<=0;
                  end case;
            end if;
      end process;
```

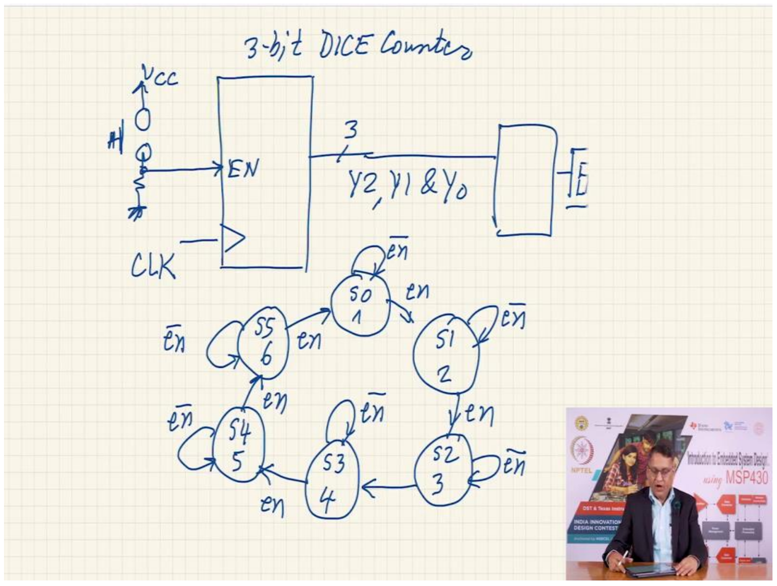
Here is the VHDL code here, as you are aware VHDL code consists of 2 parts; the entity declaration here, which defines the relationship of this circuit with the outside world. So, in this case the entity will have all those signals which enter the circuit and which exit the circuit. In our case, we have a clock and enable signal and the outputs are these actual outputs which are the output of the counter.

And then the second part of a VHDL code is the architectural description architectural block which decides the relationship between these input signals and the output signal. And you can express that in various ways various methods of writing your code, one of a common popular method is called behavioural style of modelling.

In this you have represented your code and once this code is compiled, it will give you a net list of components or digital logic building blocks which when connected, as the net list suggests, it will implement a 3 bit counter. Now, what can I do with that counter? Let me show you an interesting variation to that counter. We have a counter with 3 outputs and it uses all the possible states that is going from 000 to 7. So, essentially what was implemented was an up counter with an enable signal, whenever enable is 1 it will transition to the next state, if the enable is held to 0, it will hold that current state.

Let me say that I modify that counter to not ha3 three d type flip flops, because the number of states is more than 4 and less than 8. What would I use that for? I could use this circuit to actually implement a dice, how?

(Refer Slide Time: 21:44)



So, it is like this that I have created a counter. The counter requires a clock. Yes, it can be high frequency clock, but now I have a enabled signal in the following form, I have VCC, I have a switch, and I have a pull down resistor. And this is going into the enable signal of the counter. Now since this is 0, it is going to hold the state, this 3 bit, we will call it 3 bit dice counter. The outputs are, let us say, y 2, y 1 and y 0, so there are three outputs. And the state diagram for this is as follows. You have 1, 2, 3, 4, 5 and 6 states, if the enable signal is 0, so enable bar is going to remain in that state this is S 0.

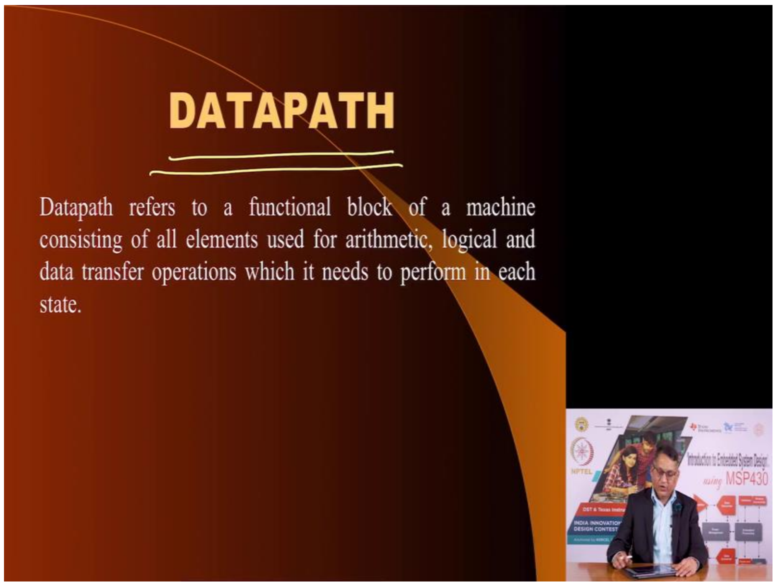
Now instead, I can choose whatever outputs I want in those states, and I want because the dice will give you random numbers between 1 and 6. I want the first state to give me 1, 2, 3, 4, 5 and 6, and if the enable signal is 1, it goes to the next. Here again, e n bar so it is S 1, e n, here e n bar. Again, this is S 2, S 3, S 4, S 5, e n, this is going to stay here for e n bar, e n, e bar, e n, and I am going to stay here for e n bar. Now, basically what I have done is I have repurposed that counter to become a dice. The only condition is that the clock frequency should be very high.

Now, when the user presses the switch, the counter will start working, it will go transition through all the states and when I remove the switch press, when I release the switch, the enable signal will become 0 and it will hold a state where it had reached and those outputs will become stable. Now, these 3 bits can be decoded passed through another combinational circuit and can be represented on a 7 segment display if you like.

And so, what you have done is started with a simple finite state machine and convert it into a useful piece of equipment or gadget. However, please understand that this is not really what full computer would do. This does serve a purpose but this is a very simple applications. In real applications, the finite state machine is only a sequencer which means it tells somebody what to do and that somebody has to actually perform those operations.

In this simple application, the sequence itself could be converted into numbers, but nothing more. And so, what we are going to look at is a bigger model where we are able to not only decide the sequence, we can program what sequence we want by implementing the state machine and we have other elements of data processing, which are under the control of the outputs from the state machine to perform useful computational operations. And that approach involves a building block which we call as the data path.

(Refer Slide Time: 25:42)

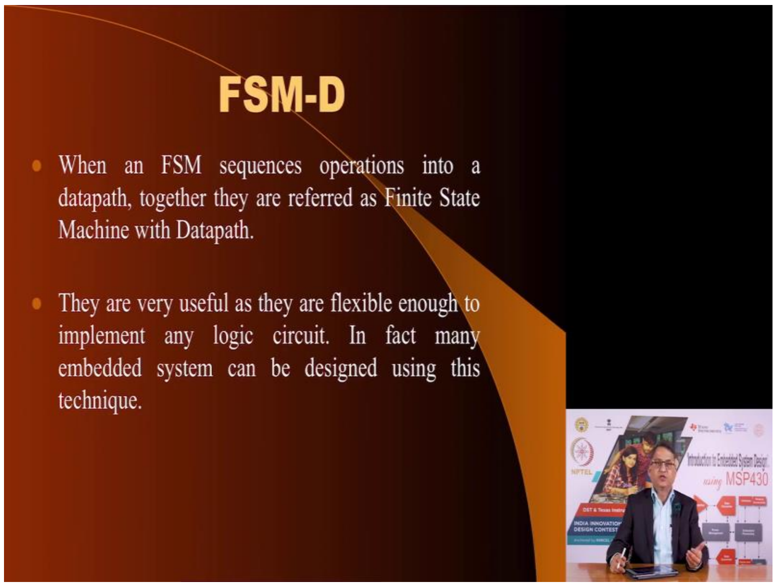


DATAPATH

Datapath refers to a functional block of a machine consisting of all elements used for arithmetic, logical and data transfer operations which it needs to perform in each state.

So, data path nothing is nothing but it represents all the functional elements which process data which are under the control of the finite state machine. And when you glue these 2 building blocks together, one which consists of the sequencer or the control path and the other which consists of data path elements, then you are able to implement a single purpose computer.

(Refer Slide Time: 26:23)

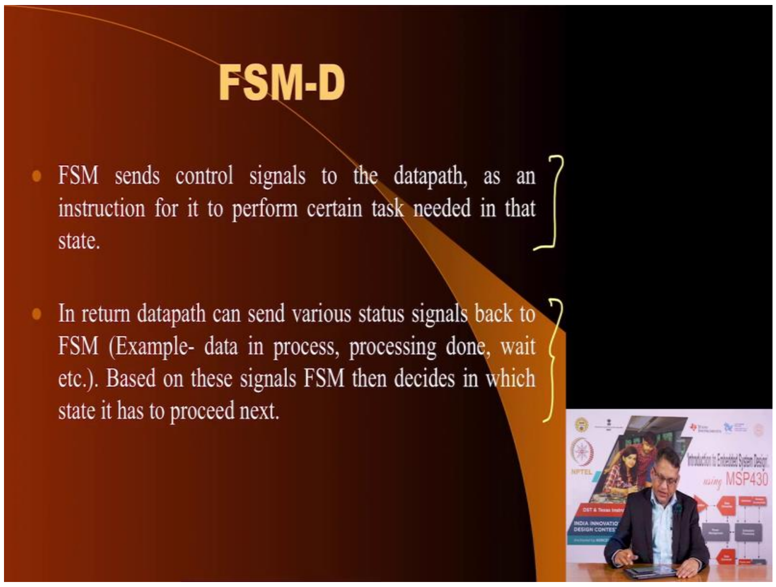


FSM-D

- When an FSM sequences operations into a datapath, together they are referred as Finite State Machine with Datapath.
- They are very useful as they are flexible enough to implement any logic circuit. In fact many embedded system can be designed using this technique.

And that is represented or that is called FSMD that is Finite State Machine with Data Path. A finite state machine sequences operations into data path through signals called control signals, the data path elements under the control of these control signals perform certain useful operations, they may give the feedback back to the sequencer using what we call as status signals. And the happy marriage of these 2 elements will be used to implement a single purpose computer.

(Refer Slide Time: 26:47)

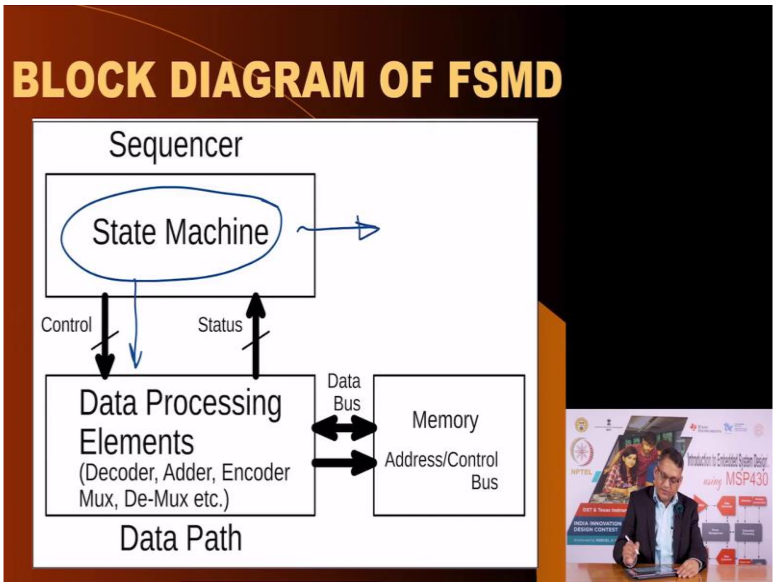


FSM-D

- FSM sends control signals to the datapath, as an instruction for it to perform certain task needed in that state.
- In return datapath can send various status signals back to FSM (Example- data in process, processing done, wait etc.). Based on these signals FSM then decides in which state it has to proceed next.

So, we need to send control signals to the data path from the sequencer, and the data path will need to perform useful operations and give feedback to the sequencer to show that the sequencer can decide what is the next course of action. Here is a block diagram

(Refer Slide Time: 27:12)

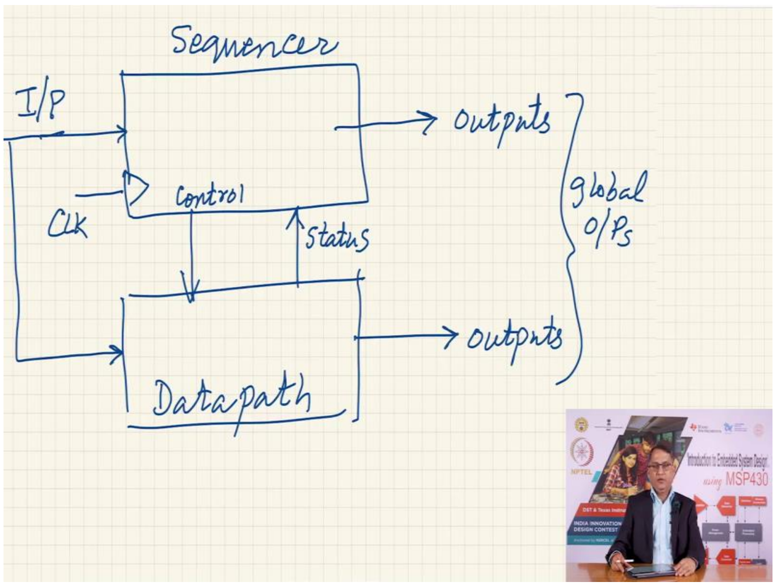


BLOCK DIAGRAM OF FSMD

I this is a repeat of what I had shown earlier that we have a sequencer, which is nothing but a state machine. And the state machine produces control signals and feeds various elements of data processing elements. And here is a list, I will show you a bigger list, these elements will operate on that data which may be coming from outside or may be provided by the sequencer and it will give a feedback to the state machine.

State machine can also provide some outputs or these data processing elements will provide outputs either stored them in memory or provided in the outside world. Let me draw a representation of FSM-D which is useful for implementing single purpose computers.

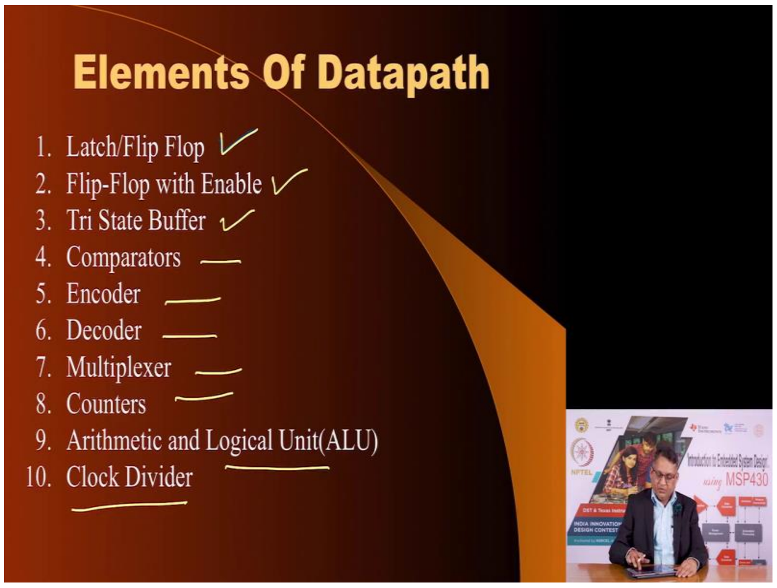
(Refer Slide Time: 27:59)



So, here is my sequencer, here are inputs and these might be these would turn out to be asynchronous inputs, I would need a clock signal, this is my clock, here is my elements of data path data path, these data path elements will get control signals from the sequencer. So, these are control, the data path elements will perform certain operations for which it may have some inputs from the same inputs which are going to the sequencer, or it may have certain other inputs, it will provide feedback and we call this status.

The data path may produce certain outputs for global consumption that is for the outside world. The sequencer may also produce some outputs if it so wishes, and these 2 outputs are your global outputs or user outputs. So, this is the block diagram that we are going to use in this lecture and we will implement a useful single purpose computer.

(Refer Slide Time: 29:28)



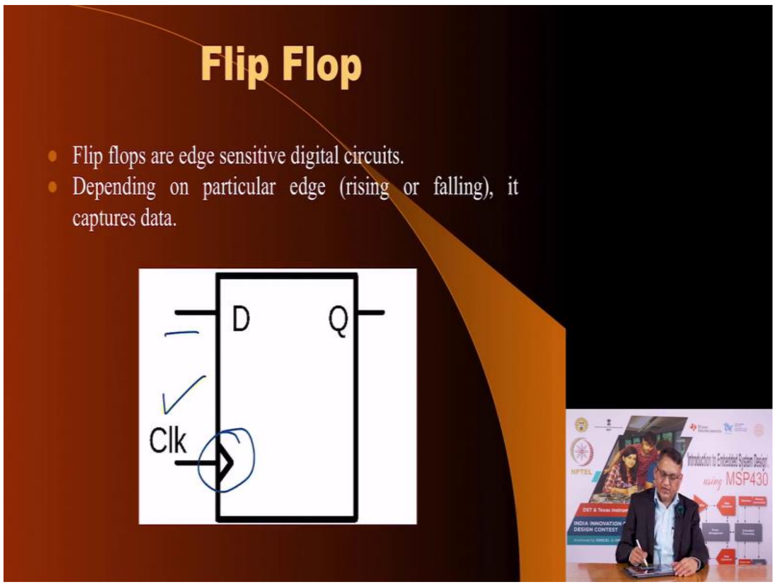
Elements Of Datapath

1. Latch/Flip Flop ✓
2. Flip-Flop with Enable ✓
3. Tri State Buffer ✓
4. Comparators
5. Encoder
6. Decoder
7. Multiplexer
8. Counters
9. Arithmetic and Logical Unit(ALU)
10. Clock Divider

What are the elements of data path? This is not this is not the only 10 elements, this is just a sample of such data processing elements. Usually data path elements are elements of combinational circuit, you can implement elements of data path using simple combinational logic, but that is not necessary. Sometimes you may have to store some information for further processing. So, for that you may require latch and flip flops. Sometimes you want a flip flop with the enable signal so that you only capture the data whenever the enable signal allows you to.

You may use tri state buffers, you may use comparators for comparing 2 numbers, you may use encoder and decoder, you may use multiplexers and demultiplexers and counters and you can use adders and subtractors or you could make generic building block and arithmetic logical logic unit which would not only perform mathematical operations, but may also perform logical operations. You could also divide the clock for use in certain parts of the data path. And this is just a sample of some of the data path building blocks.

(Refer Slide Time: 30:46)

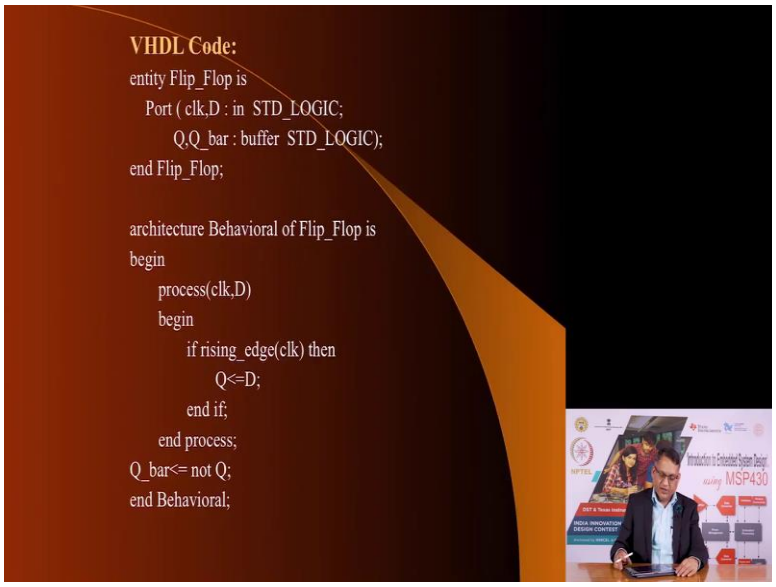


Flip Flop

- Flip flops are edge sensitive digital circuits.
- Depending on particular edge (rising or falling), it captures data.

So, here is the representation of a flip flop. And, like I mentioned, we do not manually connect all the building blocks such as flip flops or gates, but we write VHDL code to model them. So for our flip flop here, it is a rising edge flip flop as this clock indicates here, this is the rising edge. And so it is going to capture whatever data here on every rising edge and present it to the output.

(Refer Slide Time: 31:15)



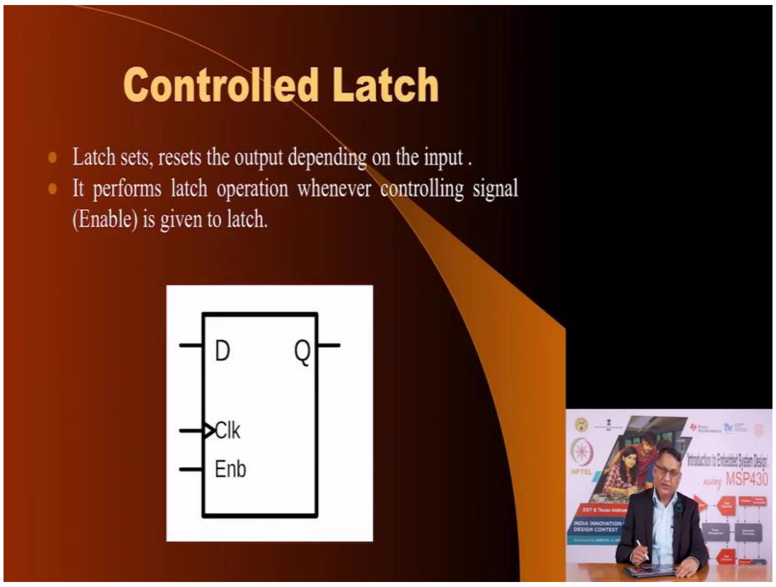
VHDL Code:

```
entity Flip_Flop is
   Port ( clk,D : in  STD_LOGIC;
          Q,Q_bar : buffer  STD_LOGIC);
end Flip_Flop;

architecture Behavioral of Flip_Flop is
begin
      process(clk,D)
      begin
            if rising_edge(clk) then
                  Q<=D;
            end if;
      end process;
Q_bar<= not Q;
end Behavioral;
```

Here is a VHDL code to represent that. I am not going to go in detail and I strongly urge you and encourage you to learn VHDL or other hardware description language such as Verilog and there are many, many software programs available which you can which you can use to practice and develop logic systems.

(Refer Slide Time: 31:45)

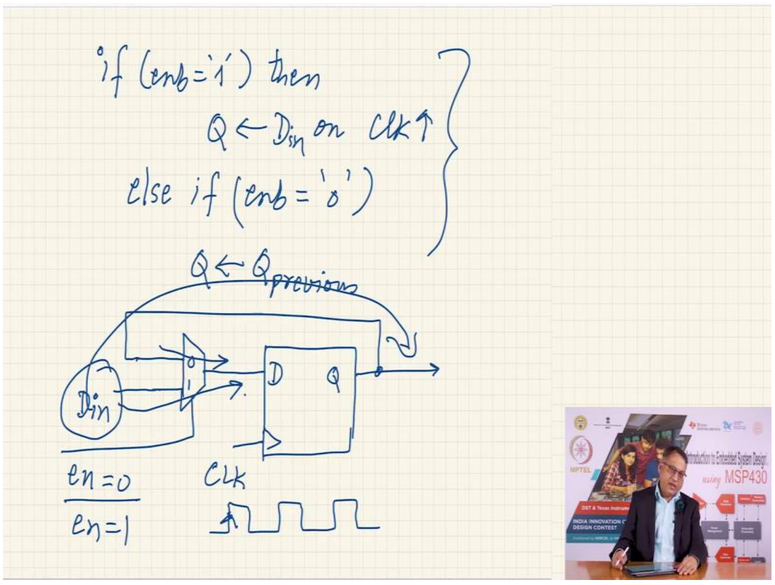


Controlled Latch

- Latch sets, resets the output depending on the input .
- It performs latch operation whenever controlling signal (Enable) is given to latch.

Here is a controlled latch which means it is going to capture information at the rising edge but only when enable is 1, let me show you what does it mean?

(Refer Slide Time: 31:56)



It means if enable is equal to 1, then Q gets the value of D on clock rising edge, else if enable is equal to 0, Q gets the value of the previous value, so I will call it Q previous which means it does not capture the input. Now, how does it how does it use a normal D type flip flop, how does a normal D type flip flop get modified to implement such a such a requirement? Well, this can be implemented as a finite state machine by itself.

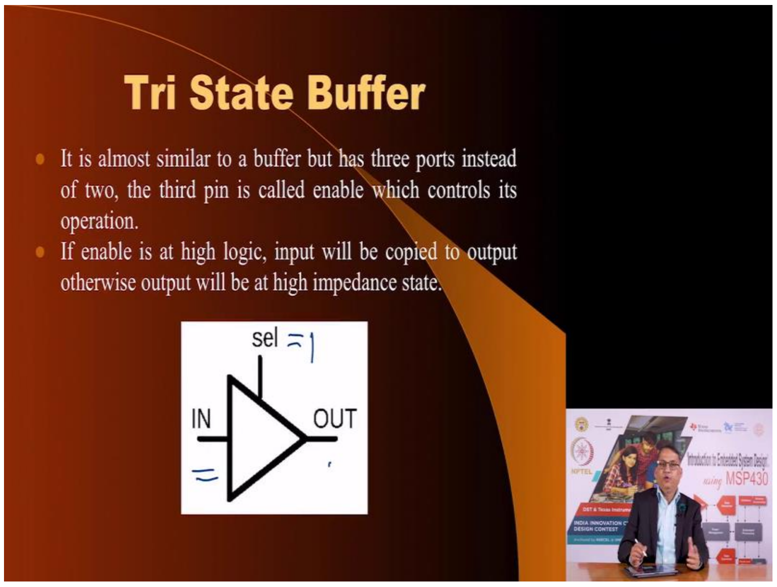
And you could draw the state diagram for this final state machine and implement it, and the implementation will look something like this that it does use a D type flip flop, but it uses an additional component which in this case is a 2 input multiplexer; 0 and 1 are the inputs, the output of the multiplexer provides the connection to the D input of the D type flip flop.

Here is the Q which will go to the outside, but it will also be fed back into the 0 input of the multiplexer. The one input of the multiplexer is actually the D in, which you want to capture here so D in. The clock will be applied to the clock input and the control signal of the select line of the multiplexer will become the enable signal.

Now, let us go through this circuit to understand that it actually implements this requirement. Now, if enable is equal to 0, the 0th input gets connected here. And the 0th input is nothing but the current value of Q. So on every rising edge, it is going to just capture D and transfer it to queue which means Q value is not going to change, but the moment enable signal becomes 1. Now it is going to route the D in input to the Q and now on every rising edge every rising edge it is going to capture this value and put it at the output, this is exactly what we wanted.

And so, this is the method of implementing a latch with enable signal. Similarly, you can add more signals to this by saying that I want a reset signal that is if reset is one, then irrespective of the state of the rest of the input signals, the output should be 0 and so on and so forth. And you can implement VHDL code to implement whatever functionality that you wish, here this is the code for control the flip flop that is a flip flop where there is a enable signal, then one important building block is the tri state buffer.

(Refer Slide Time: 34:58)

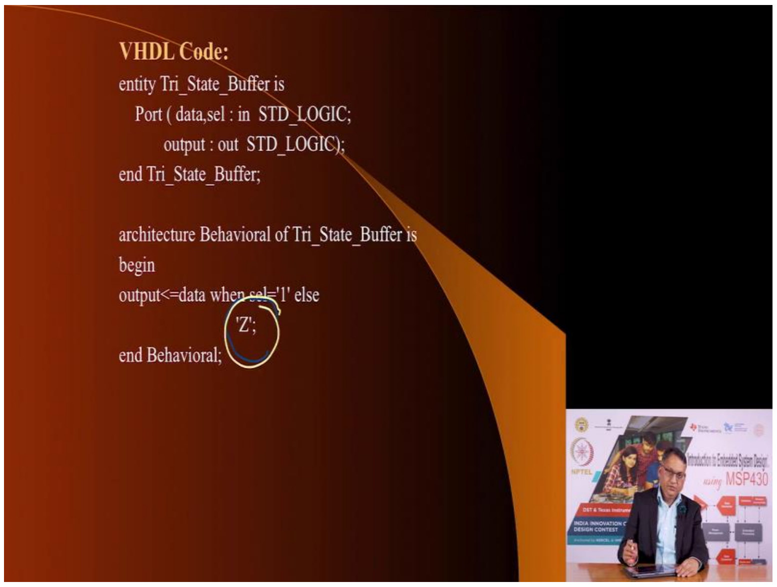


Tri State Buffer

- It is almost similar to a buffer but has three ports instead of two, the third pin is called enable which controls its operation.
- If enable is at high logic, input will be copied to output otherwise output will be at high impedance state.

Here output becomes gets the value of input provided select line is equal 1, but if select line is 0, then the output becomes high impedance.

(Refer Slide Time: 35:09)



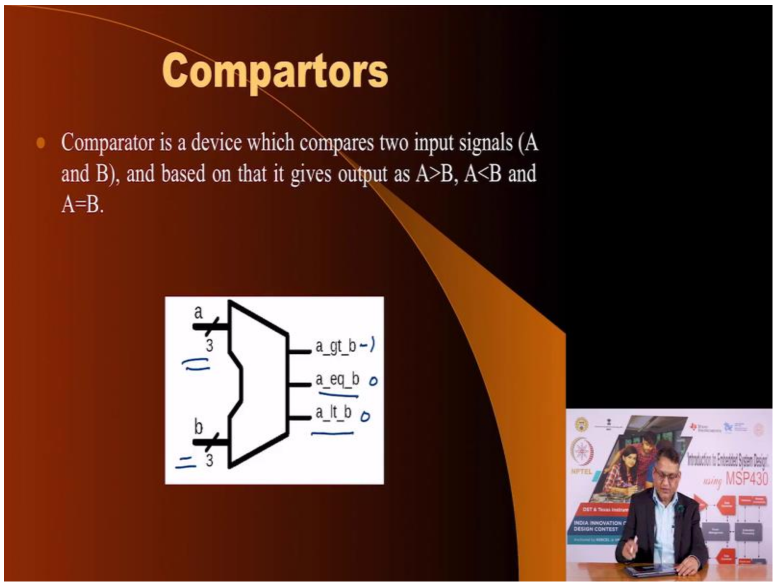
VHDL Code:

```
entity Tri_State_Buffer is
   Port ( data,sel : in  STD_LOGIC;
         output : out  STD_LOGIC);
end Tri_State_Buffer;

architecture Behavioral of Tri_State_Buffer is
begin
output<=data when sel='1' else
                    'Z';
end Behavioral;
```

Here is a VHDL code for that, this is Z here represents indicates the high impedance.

(Refer Slide Time: 35:16)



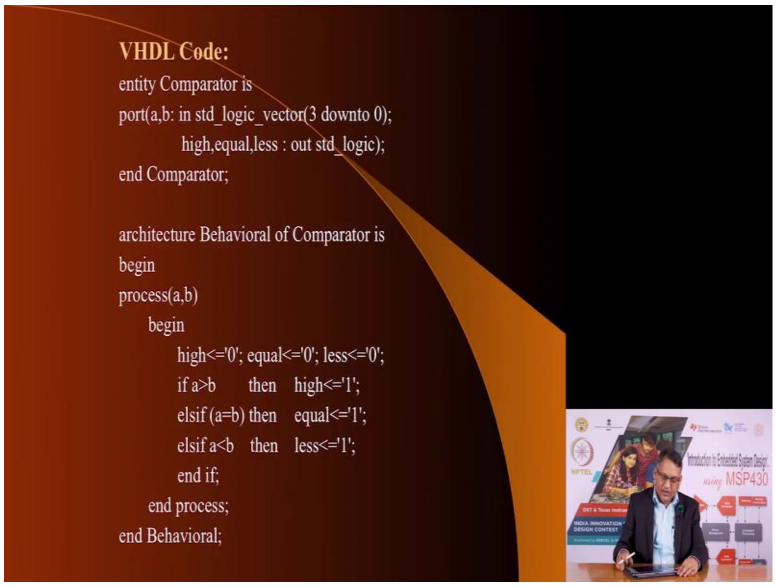
Compartors

- Comparator is a device which compares two input signals (A and B), and based on that it gives output as A>B, A<B and A=B.

Then you would require comparators to compare digital numbers. Here in this example, two 3 bit numbers A and B are being compared. And a comparator typically has three outputs, A is greater than B, A is equal to B and A is less than B. Whenever, any of these three conditions is true, that particular output becomes 1 and the others are 0.

So if numerical value of A those three bits as a number is more than the numerical value of B, AGT will be 1 and the rest of the 2 outputs will be 0. If the numerical value of A is equal to B, AGT will be 0, A equal to B will become 1 and A less than B will also remain 0.

(Refer Slide Time: 36:06)



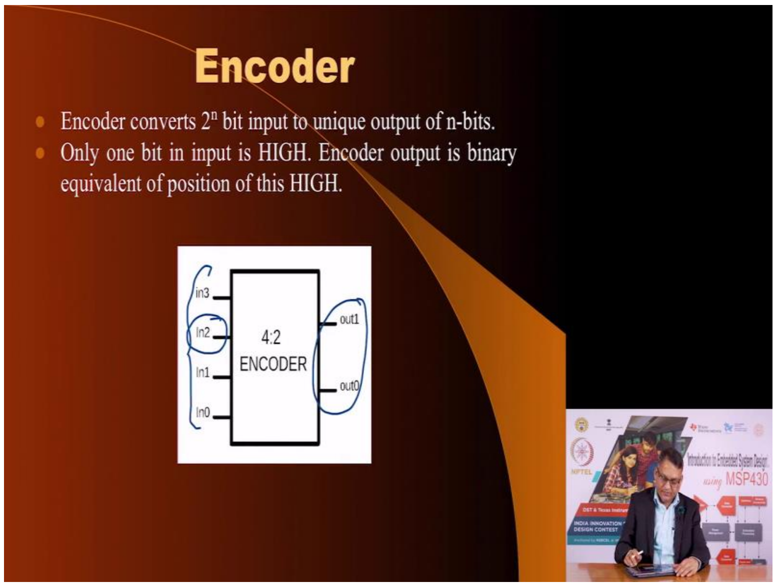
VHDL Code:

```
entity Comparator is
port(a,b: in std_logic_vector(3 downto 0);
          high,equal,less : out std_logic);
end Comparator;

architecture Behavioral of Comparator is
begin
process(a,b)
    begin
        high<='0'; equal<='0'; less<='0';
        if a>b      then   high<='1';
        elsif (a=b) then   equal<='1';
        elsif a<b   then   less<='1';
        end if;
    end process;
end Behavioral;
```

Likewise, this is the VHDL code for implementing such a three bit comparator and you can increase the size by having n bits on the 2 inputs.

(Refer Slide Time: 36:17)

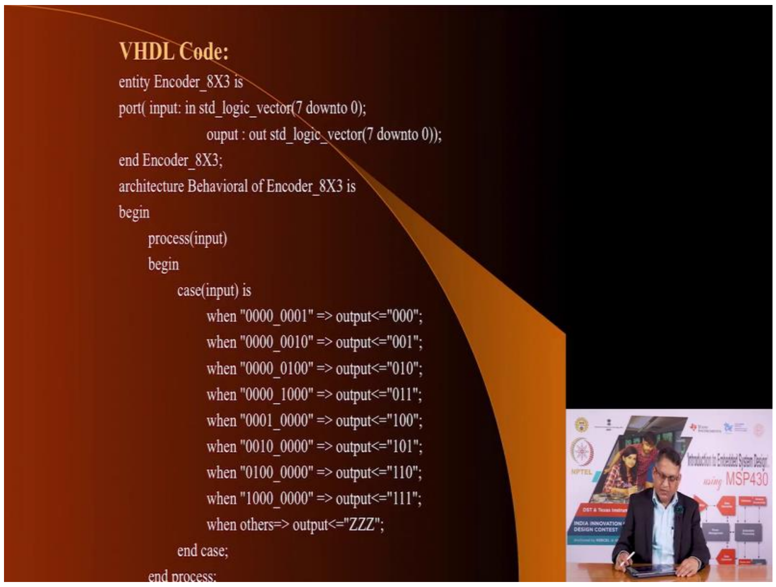


Encoder

- Encoder converts 2^n bit input to unique output of n-bits.
- Only one bit in input is HIGH. Encoder output is binary equivalent of position of this HIGH.

Similarly, this is an encoder which encodes the value here depending upon the combination here of 0s and 1s, it provides an output and you could make a priority encoder meaning if this is 1, then the value of this will indicate that I n 2 is 1 and so on.

(Refer Slide Time: 36:40)

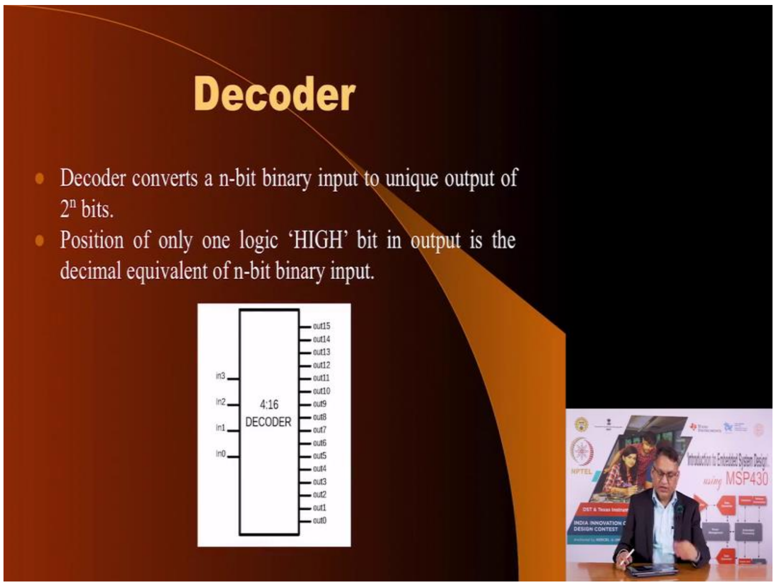


VHDL Code:

```
entity Encoder_8X3 is
port( input: in std_logic_vector(7 downto 0);
            ouput : out std_logic_vector(7 downto 0));
end Encoder_8X3;
architecture Behavioral of Encoder_8X3 is
begin
    process(input)
    begin
        case(input) is
            when "0000_0001" => output<="000";
            when "0000_0010" => output<="001";
            when "0000_0100" => output<="010";
            when "0000_1000" => output<="011";
            when "0001_0000" => output<="100";
            when "0010_0000" => output<="101";
            when "0100_0000" => output<="110";
            when "1000_0000" => output<="111";
            when others=> output<="ZZZ";
        end case;
    end process;
```

Here is the code for the encoder.

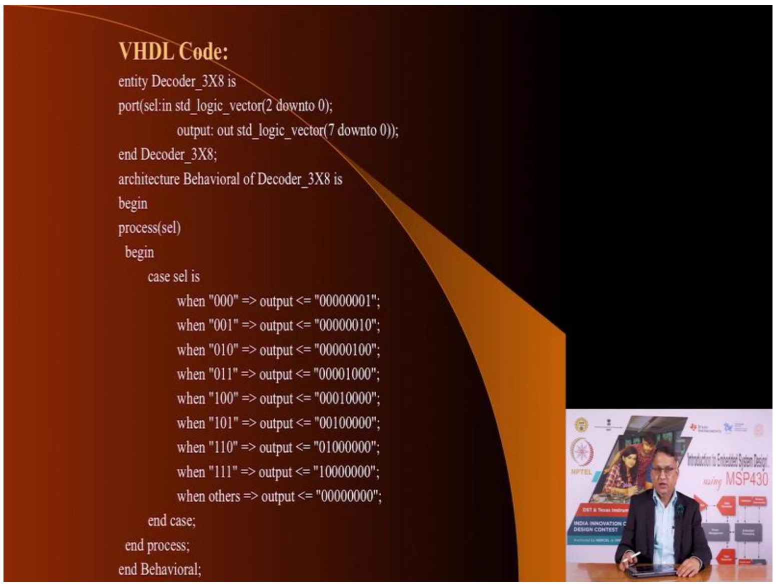
(Refer Slide Time: 36:46)



Decoder

- Decoder converts a n-bit binary input to unique output of 2^n bits.
- Position of only one logic 'HIGH' bit in output is the decimal equivalent of n-bit binary input.

in3
in2
in1
in0
4:16 DECODER
out15
out14
out13
out12
out11
out10
out9
out8
out7
out6
out5
out4
out3
out2
out1
out0

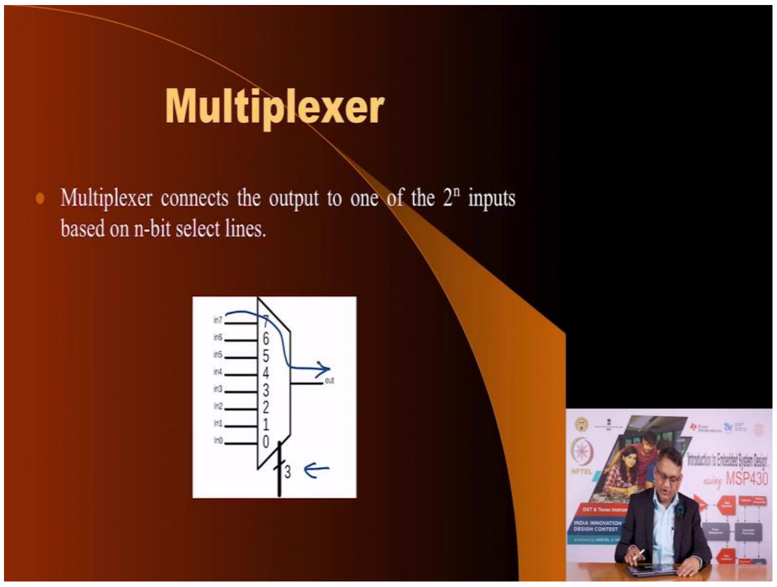


VHDL Code:

```
entity Decoder_3X8 is
port(sel:in std_logic_vector(2 downto 0);
          output: out std_logic_vector(7 downto 0));
end Decoder_3X8;
architecture Behavioral of Decoder_3X8 is
begin
process(sel)
 begin
     case sel is
          when "000" => output <= "00000001";
          when "001" => output <= "00000010";
          when "010" => output <= "00000100";
          when "011" => output <= "00001000";
          when "100" => output <= "00010000";
          when "101" => output <= "00100000";
          when "110" => output <= "01000000";
          when "111" => output <= "10000000";
          when others => output <= "00000000";
     end case;
 end process;
end Behavioral;
```

This is the opposite of encoder, this is a decoder. In this case is a 4 to 16 decoder and again VHDL representation for such a decoder.

(Refer Slide Time: 36:58)



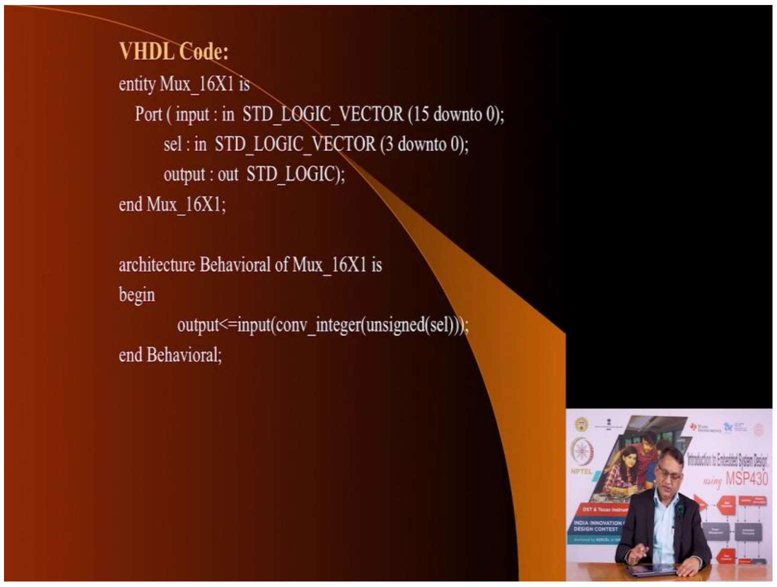
Multiplexer

- Multiplexer connects the output to one of the 2^n inputs based on n-bit select lines.

in7
in6
in5
in4
in3
in2
in1
in0
7
6
5
4
3
2
1
0
out
3
Introduction to Embedded System Design using MSP430
NPTEL
INDIA INNOVATION DESIGN CONTEST

Then we have a multiplexer, this is an eight input multiplexer, each of the inputs is single bits and the select lines here will decide which of the inputs get routed to the output, we use such a multiplexer 2 input multiplexer in a previous example, and multiplexers are very, very useful building blocks and we are going to see their use in our implementation.

(Refer Slide Time: 37:20)



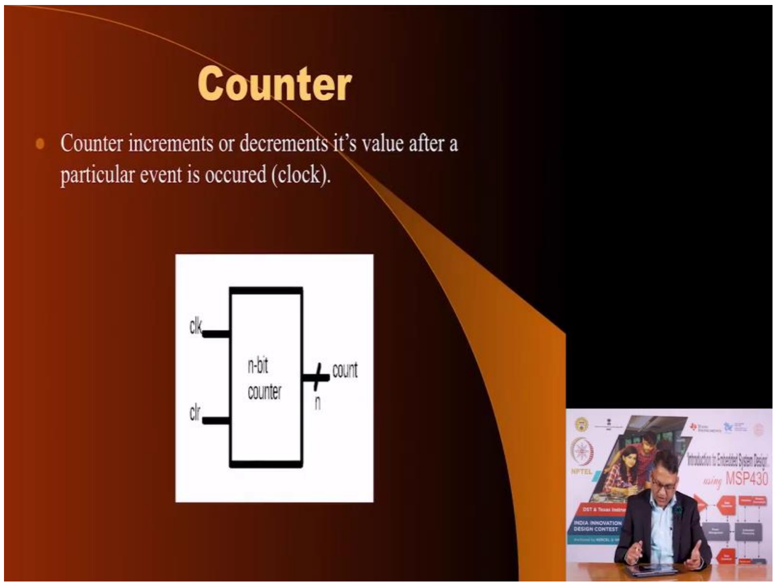
VHDL Code:

```
entity Mux_16X1 is
    Port ( input : in  STD_LOGIC_VECTOR (15 downto 0);
           sel : in  STD_LOGIC_VECTOR (3 downto 0);
           output : out  STD_LOGIC);
end Mux_16X1;

architecture Behavioral of Mux_16X1 is
begin
           output<=input(conv_integer(unsigned(sel)));
end Behavioral;
```

This is the VHDL representation.

(Refer Slide Time: 37:23)

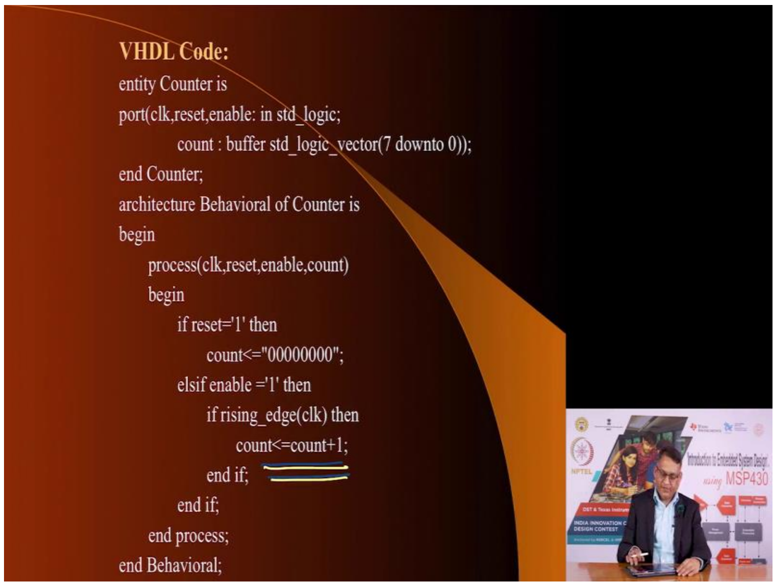


Counter

- Counter increments or decrements it's value after a particular event is occured (clock).

Again, this is a counter and in this case the counter has a clock and a clear signal which means if there is 1, all output bits of the counter will become 0, else it will count. Now, this does not indicate whether it is going to count up or count down or may have a different counting sequence that will be decided by the kind of VHDL we have.

(Refer Slide Time: 37:46)

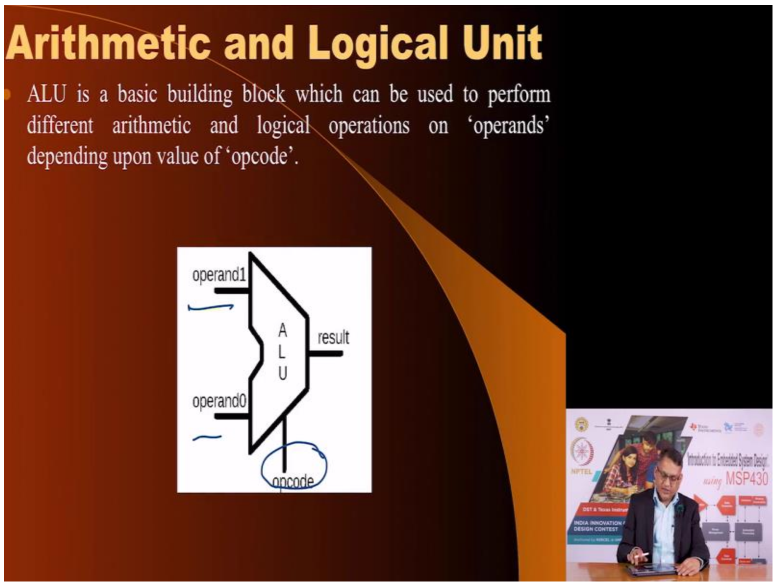


VHDL Code:

```
entity Counter is
port(clk,reset,enable: in std_logic;
        count : buffer std_logic_vector(7 downto 0));
end Counter;
architecture Behavioral of Counter is
begin
    process(clk,reset,enable,count)
    begin
        if reset='1' then
            count<="00000000";
        elsif enable ='1' then
            if rising_edge(clk) then
                count<=count+1;
            end if;
        end if;
    end process;
end Behavioral;
```

And as we see here the count is being incremented on each clock edge. Therefore, this is a up counter, VHDL code for up counter.

(Refer Slide Time: 38:01)

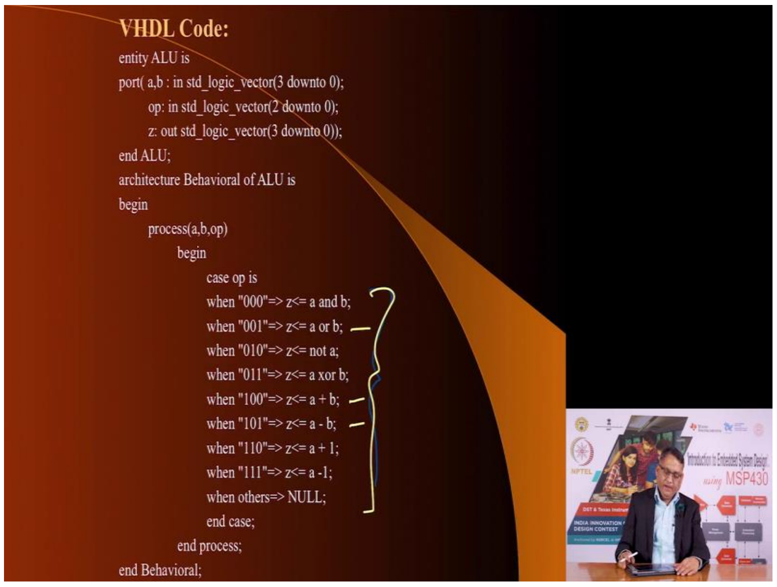


Arithmetic and Logical Unit

- ALU is a basic building block which can be used to perform different arithmetic and logical operations on 'operands' depending upon value of 'opcode'.

This is a arithmetic and logical unit and there are 2 operands, operand 1 and operand 0, and the opcode that is the bits here will decide what operation it performs and it will depend on you, if you want to do add, subtract, and to the bits of operands 1 and 0, it will depend on all of you.

(Refer Slide Time: 38:20)

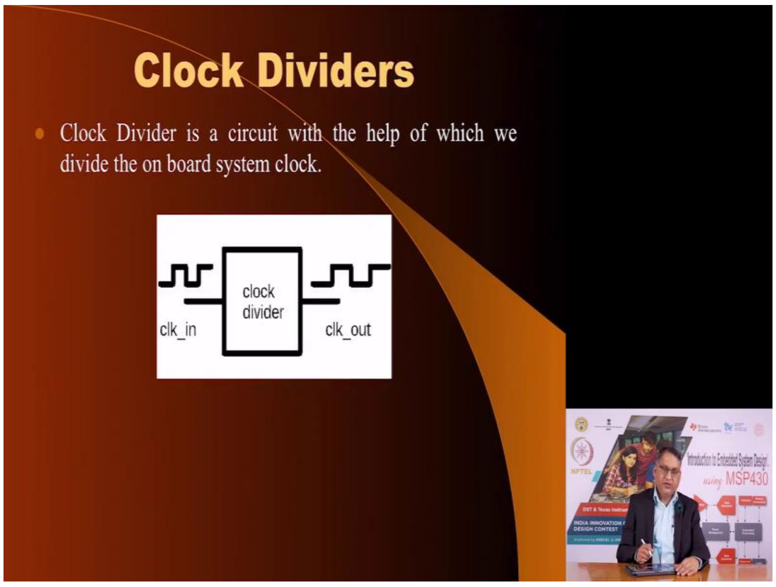


VHDL Code:

```
entity ALU is
port( a,b : in std_logic_vector(3 downto 0);
      op: in std_logic_vector(2 downto 0);
      z: out std_logic_vector(3 downto 0));
end ALU;
architecture Behavioral of ALU is
begin
      process(a,b,op)
            begin
                  case op is
                  when "000"=> z<= a and b;
                  when "001"=> z<= a or b;
                  when "010"=> z<= not a;
                  when "011"=> z<= a xor b;
                  when "100"=> z<= a + b;
                  when "101"=> z<= a - b;
                  when "110"=> z<= a + 1;
                  when "111"=> z<= a -1;
                  when others=> NULL;
                  end case;
            end process;
end Behavioral;
```

As you can see here, a lot of functions are being performed, here you are ANDing you are ORing, you are adding subtracting you are incrementing, you are decrementing and so on. So, you can implement a wide variety of functions using a ALU.

(Refer Slide Time: 38:39)

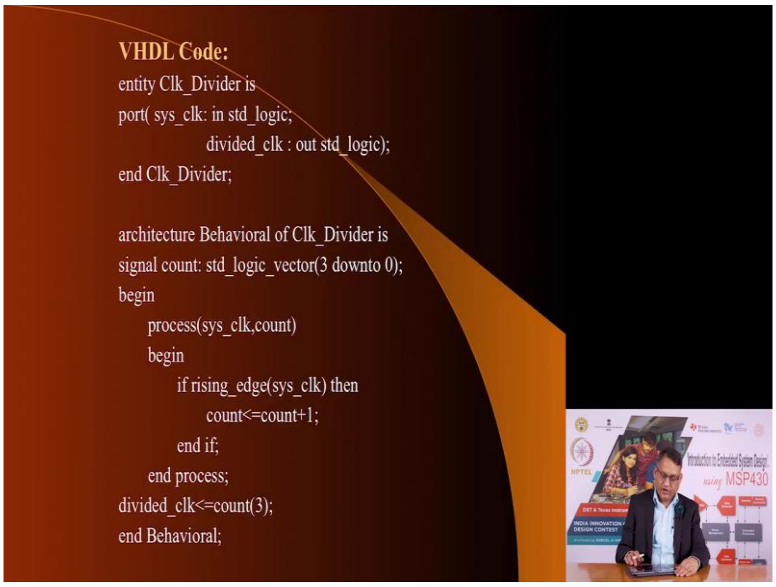


Clock Dividers

- Clock Divider is a circuit with the help of which we divide the on board system clock.

Clock bigger divider is used to divide the clock frequency from a higher value to a lower value and it is often useful in FSM-D applications.

(Refer Slide Time: 38:50)



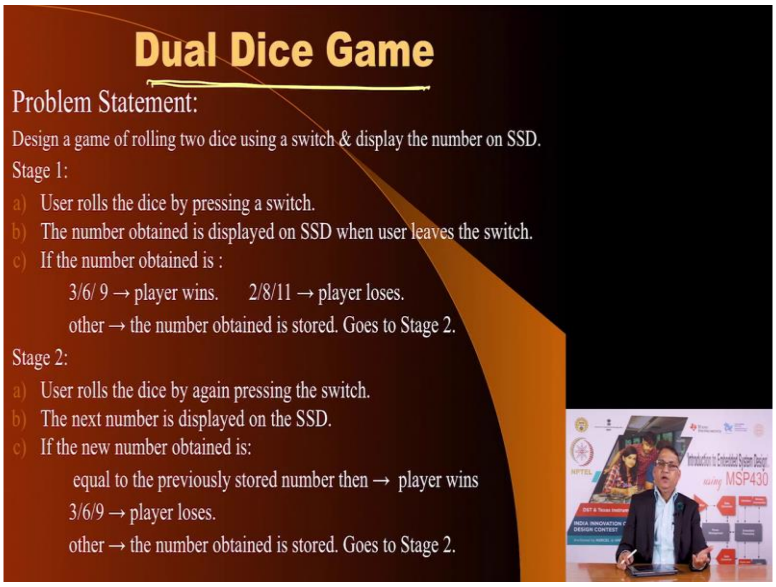
VHDL Code:

```
entity Clk_Divider is
port( sys_clk: in std_logic;
            divided_clk : out std_logic);
end Clk_Divider;

architecture Behavioral of Clk_Divider is
signal count: std_logic_vector(3 downto 0);
begin
    process(sys_clk,count)
    begin
        if rising_edge(sys_clk) then
            count<=count+1;
        end if;
    end process;
divided_clk<=count(3);
end Behavioral;
```

Now what we are going to do this is the VHDL code for a clock divider.

(Refer Slide Time: 38:54)



Dual Dice Game

Problem Statement:

Design a game of rolling two dice using a switch & display the number on SSD.

Stage 1:

a) User rolls the dice by pressing a switch.
b) The number obtained is displayed on SSD when user leaves the switch.
c) If the number obtained is :

   3/6/ 9 → player wins.   2/8/11 → player loses.

   other → the number obtained is stored. Goes to Stage 2.

Stage 2:

a) User rolls the dice by again pressing the switch.
b) The next number is displayed on the SSD.
c) If the new number obtained is:

   equal to the previously stored number then → player wins

   3/6/9 → player loses.

   other → the number obtained is stored. Goes to Stage 2.

What we are going to do is, in our next lecture, we are going to implement a single purpose computer to implement a particular requirement and that is to implement a dual dice scheme. That is a game in which there are 2 dice outputs and the rules of the game will be mentioned and we will see how these are implemented very easily using the finite state machine with data path approach. So, I will see you soon in the next lecture.