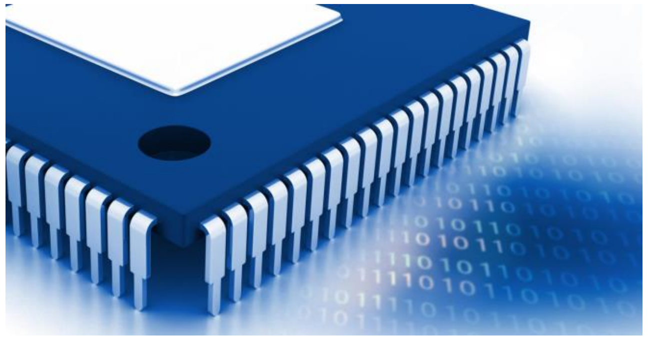
PROGRAMA JALISCO COMPETITIVO



# Chapter 8: State Machine

CINVESTAV-Unidad Guadalajara-Diciembre 2019

## Agenda

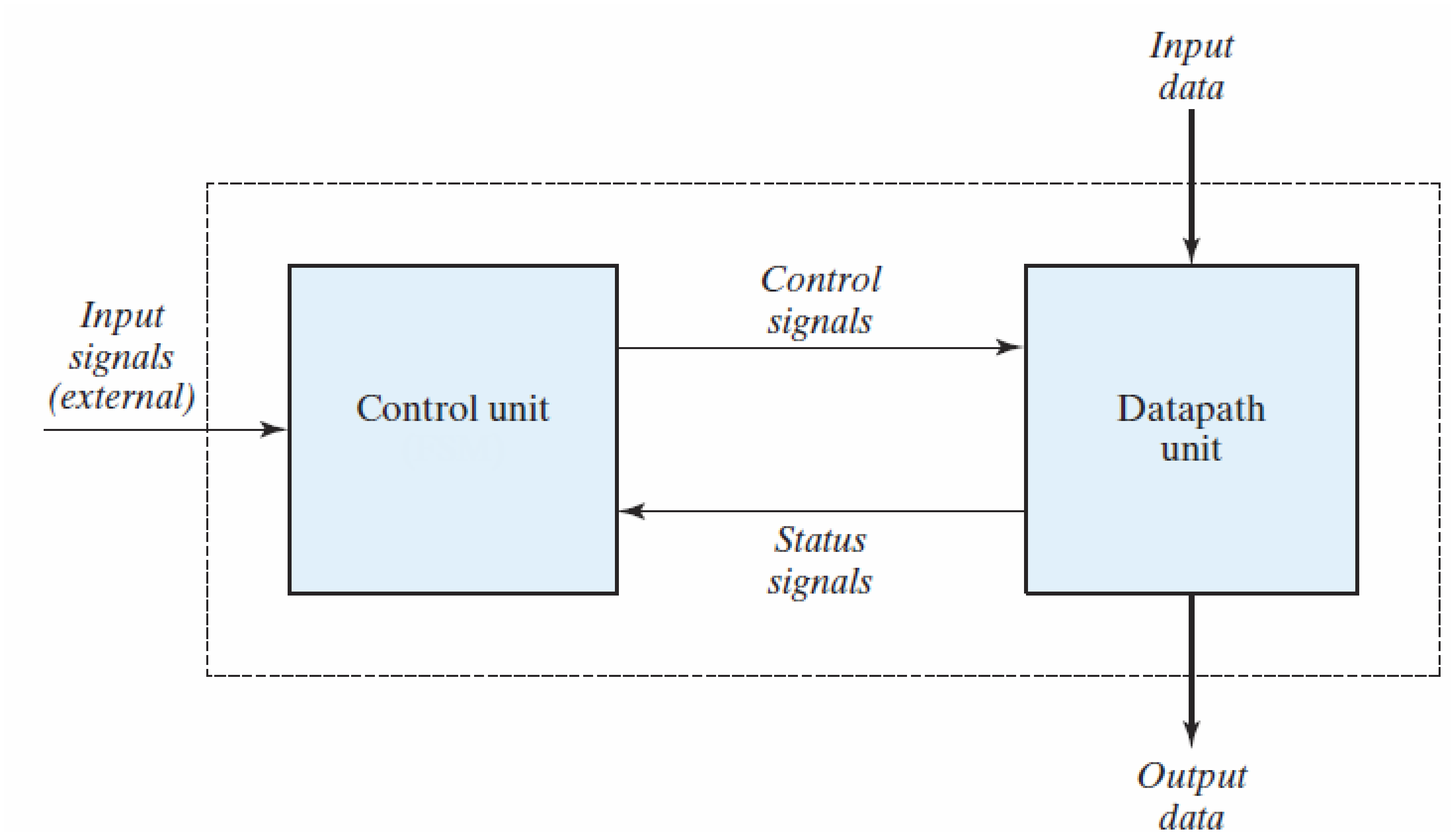
## Control and Data Path

The design of the logic of a digital system can be divided into two distinct efforts.

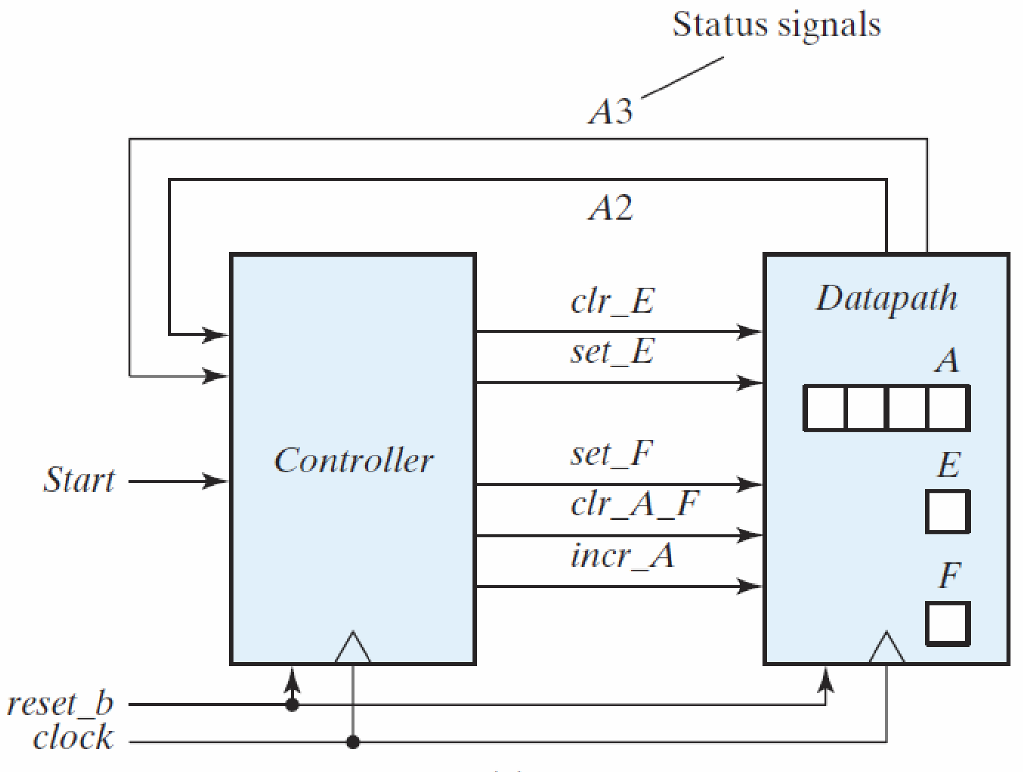
One part is concerned with designing the digital circuits that perform the data-processing operations.

The other part is concerned with designing the control circuits that determine the sequence in which the various manipulations of data are performed.

## Control and Data Path



## Control and Data Path



(a)

## Control and Data Path

The data-processing path, commonly referred to as the *datapath unit*, manipulates data in registers according to the system’s requirements.

The *control unit* issues a sequence of commands to the datapath unit.

Note that an internal feedback path from the datapath unit to the control unit provides status conditions that the control unit uses together with the external (primary) inputs to determine the sequence of control signals (outputs of the control unit) that direct the operation of the datapath unit.

## Control and Data Path

The control logic that generates the signals for sequencing the operations in the datapath unit is a finite state machine (FSM), i.e., a synchronous sequential circuit.

The control commands for the system are produced by the FSM as functions of the primary inputs, the status signals, and the state of the machine.

In a given state, the outputs of the controller are the inputs to the datapath unit and determine the operations that it will execute.

## Control and Data Path

Depending on status conditions and other external inputs, the FSM goes to its next state to initiate other operations.

The digital circuits that act as the control logic provide a time sequence of signals for initiating the operations in the datapath and also determine the next state of the control subsystem itself.

## Control and Data Path

The control sequence and datapath tasks of a digital system are specified by means of a hardware algorithm.

An algorithm consists of a finite number of procedural steps that specify how to obtain a solution to a problem.

## Automatons

**Automatons** are abstract models of machines that perform computations on an input by moving through a series of states or configurations.

At each state of the computation, a transition function determines the next configuration on the basis of a finite portion of the present configuration.

As a result, once the computation reaches an accepting configuration, it accepts that input.

## Automatons

There are **four major families of automaton** :

- Finite-state machine
- Pushdown automata
- Linear-bounded automata
- Turing machine

The most general and powerful automata is the **Turing machine**.

## Automatons

An automaton in which the state set Q contains only a *finite* number of elements is called a **FSM.**

FSMs are abstract machines, consisting of a set of states, set of input events, a set of output events and a state transition function.

## Automatons

The state transition function takes the current state and an input event and returns the new set of output events and the next state. Therefore, it can be seen as a function which maps an ordered sequence of input events into a corresponding sequence, or set, of output events.

In digital design two types of FSMs are used:

- Moore machine
- Mealy machine

## Moore machine

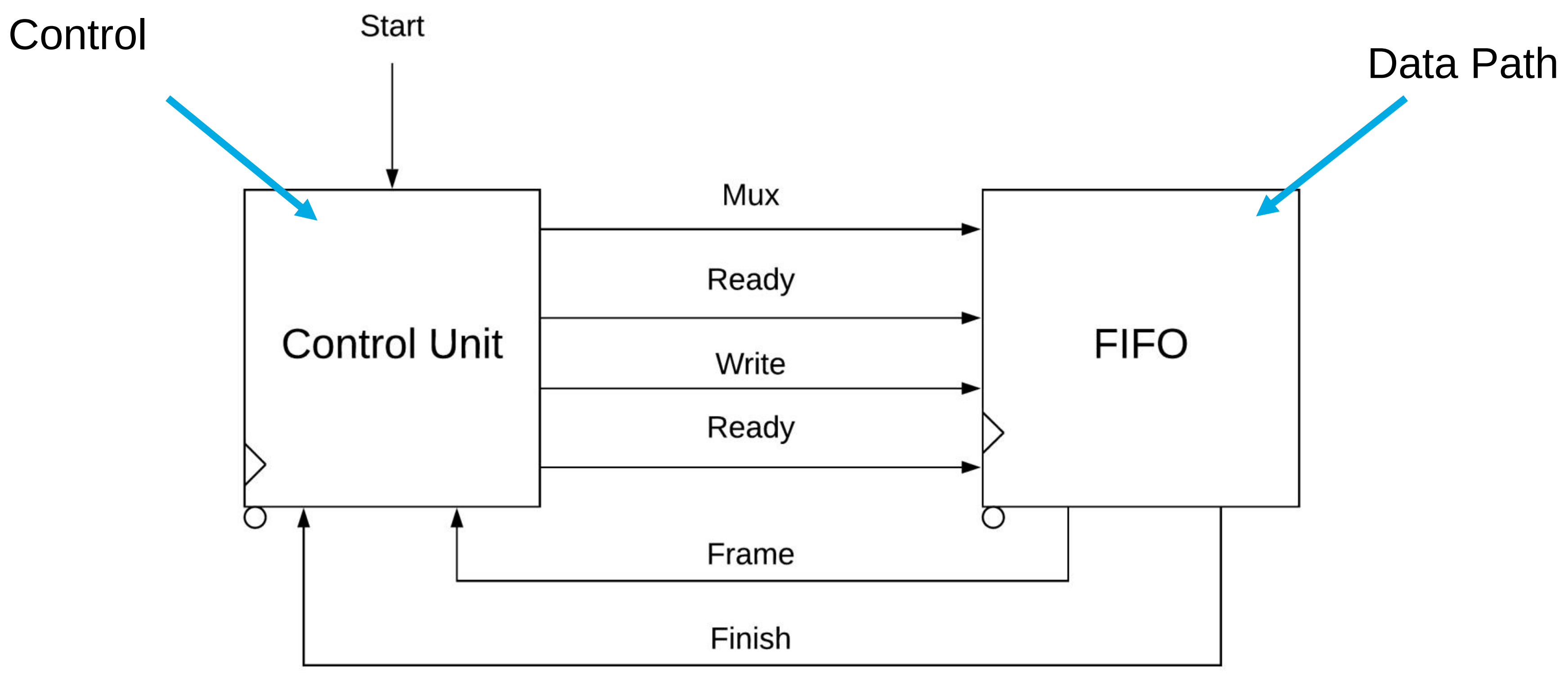
Moore machine's output is based upon the current state alone.

A Moore machine is a state machine in which all outputs are fully synchronized with the circuit clock.

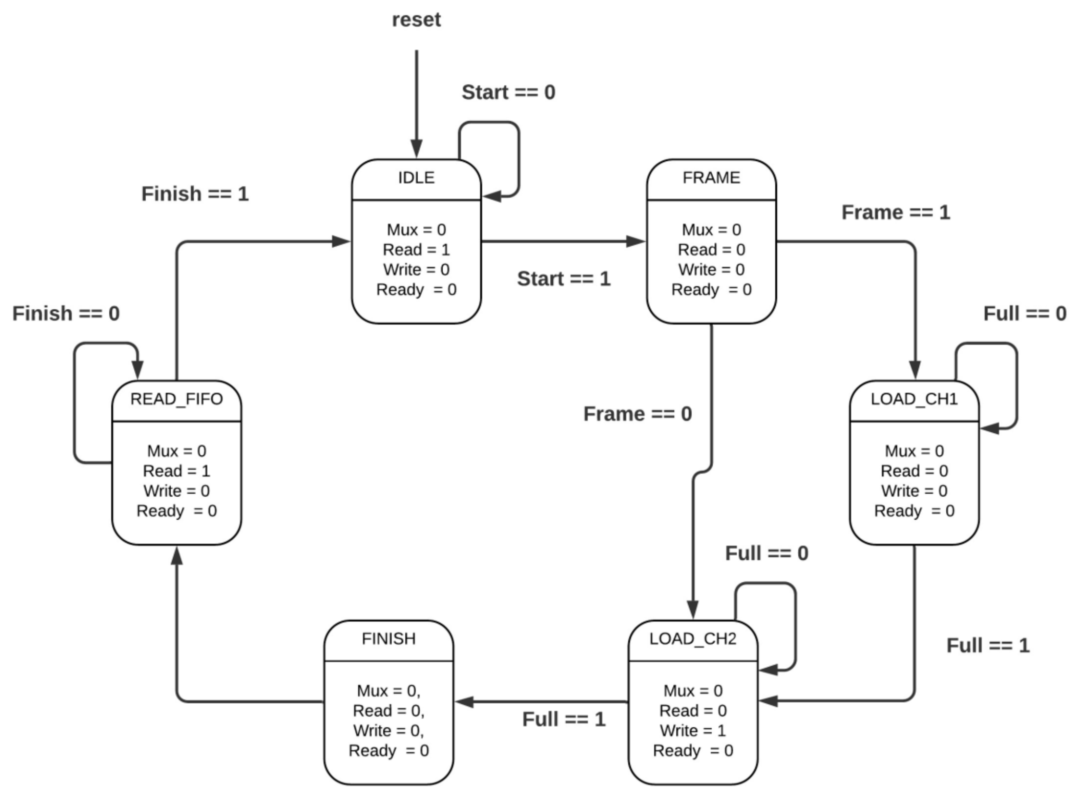
In the state diagram form, each state of the machine specifies its output(s) independent of circuit inputs.

In the Verilog code of a Moore state machine, only circuit state variables participate in the output expression of the circuit.

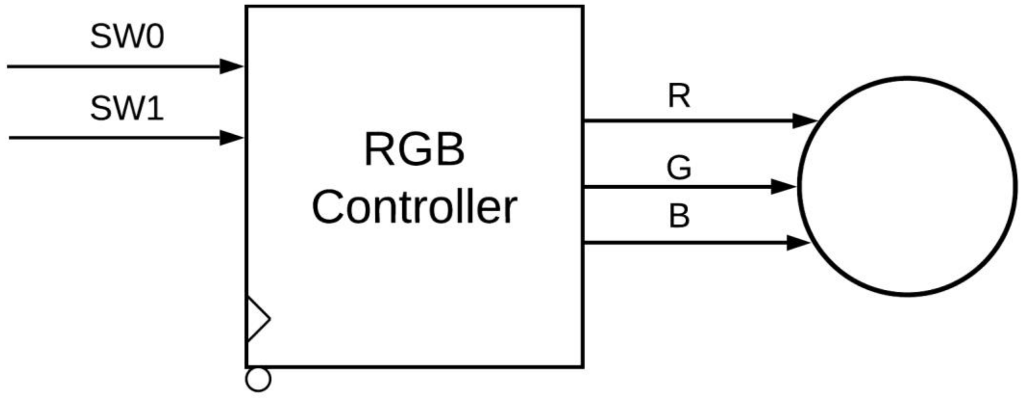
## Example of a System with Data Path and Control Unit



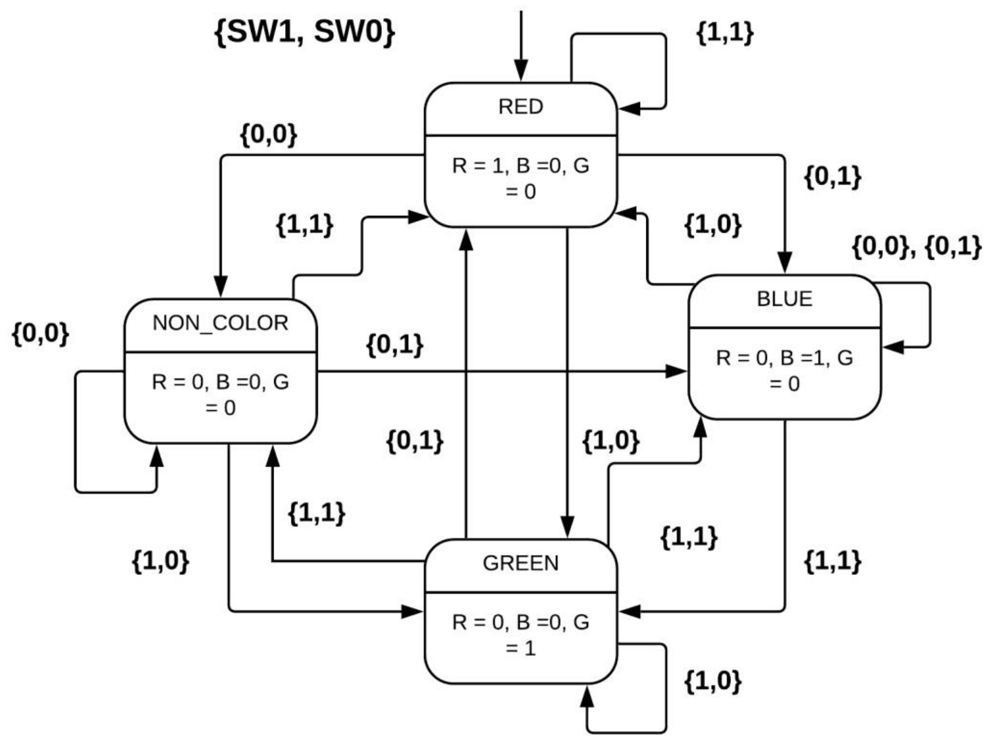
# State Machine



## State Machine



# State Machine



# Moore State Machine

```verilog
module MooreSequenceDetector
(
   // Input Ports
   input clk,
   input rst,
   input x,

   // Output Ports
   output z
);

localparam RESET = 0;
localparam GOT_1 = 1;
localparam GOT_10 = 2;
localparam GOT_101 = 3;

reg [1:0] state;

reg z_b;
```

```verilog
always@(posedge clk, negedge rst) begin

   if(rst == 1'b0)
         state <= RESET;
   else
      case(state)

         RESET:
            if(x == 1'b1)
               state <= GOT_1;
            else
               state <= RESET;
         GOT_1:
            if (x == 1'b1)
               state <= GOT_1;
            else
               state <= GOT_10;
         GOT_10:
            if(x == 1'b1)
               state <= GOT_101;
            else
               state <= RESET;
         GOT_101:
            if(x == 1'b1)
               state <= GOT_1;
            else
               state <= GOT_10;

         default:
               state <= RESET;

         endcase
end//end always
```

```verilog
always@(state)begin

    if(state == GOT_101)
      z_b = 1'b1;
   else
      z_b = 1'b0;


end

assign z = z_b;

endmodule
```

## Mealy State Machine

A Mealy machine is different from a Moore machine in that its output(s) depend on its current state and inputs while in that state.

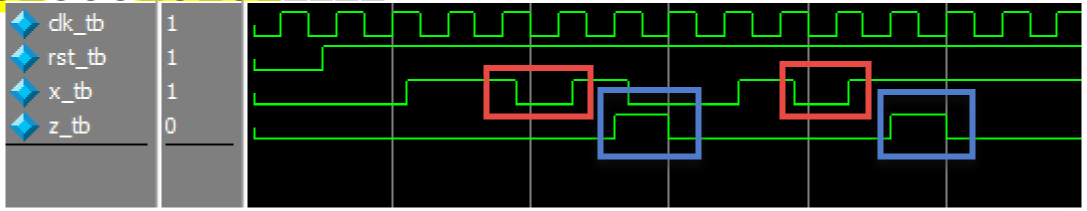
In Mealy machine output are not  synchronized with the circuit clock.

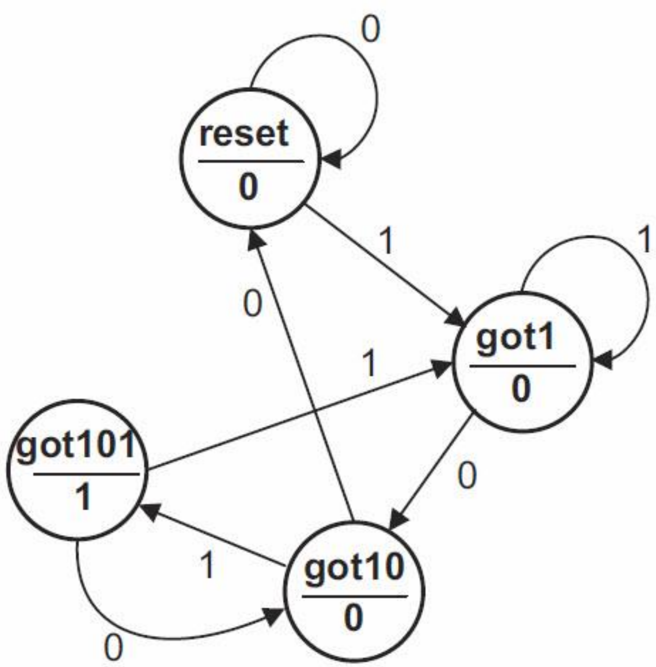
In the state diagram form, each transition of the machine specifies its output(s) depending on circuit inputs.

# Moore State Machine

1110000111101 1100

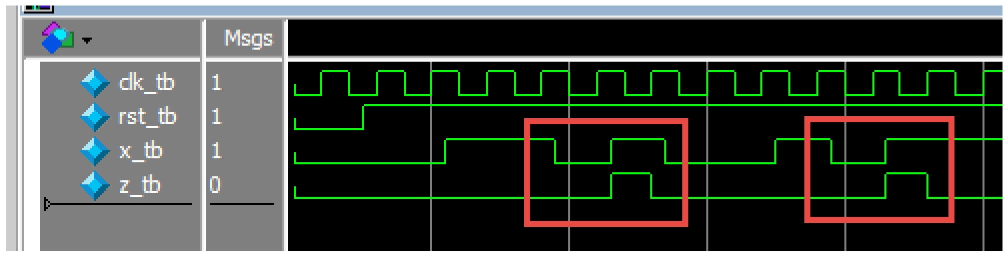
0001010001010 11111

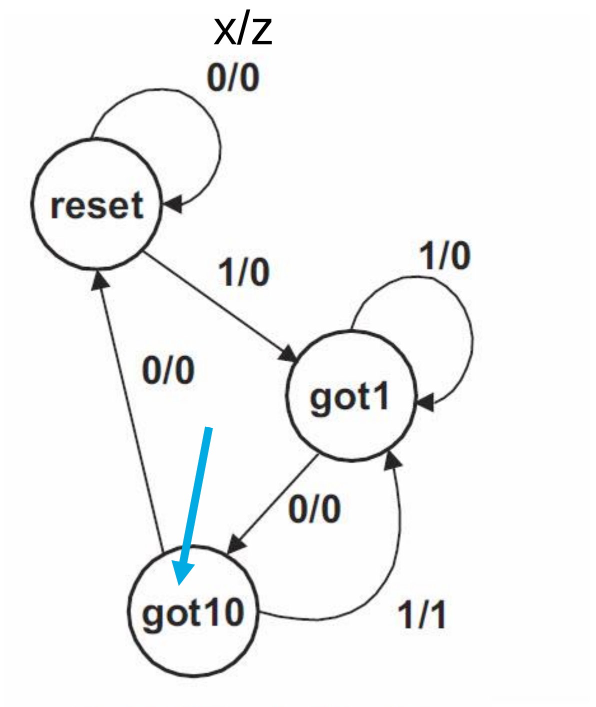




A Moore 101 Detector

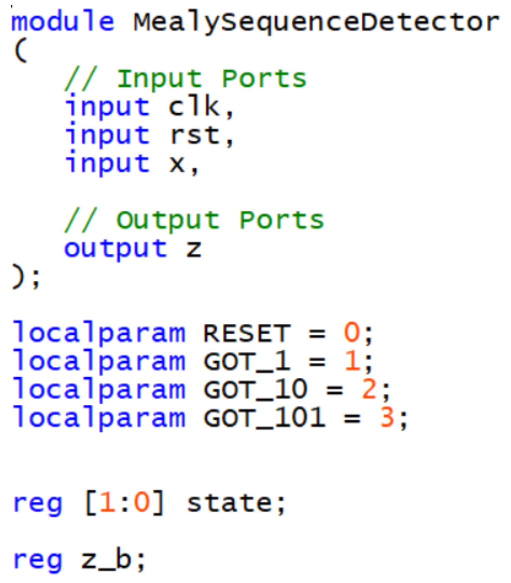
# Mealy State Machine



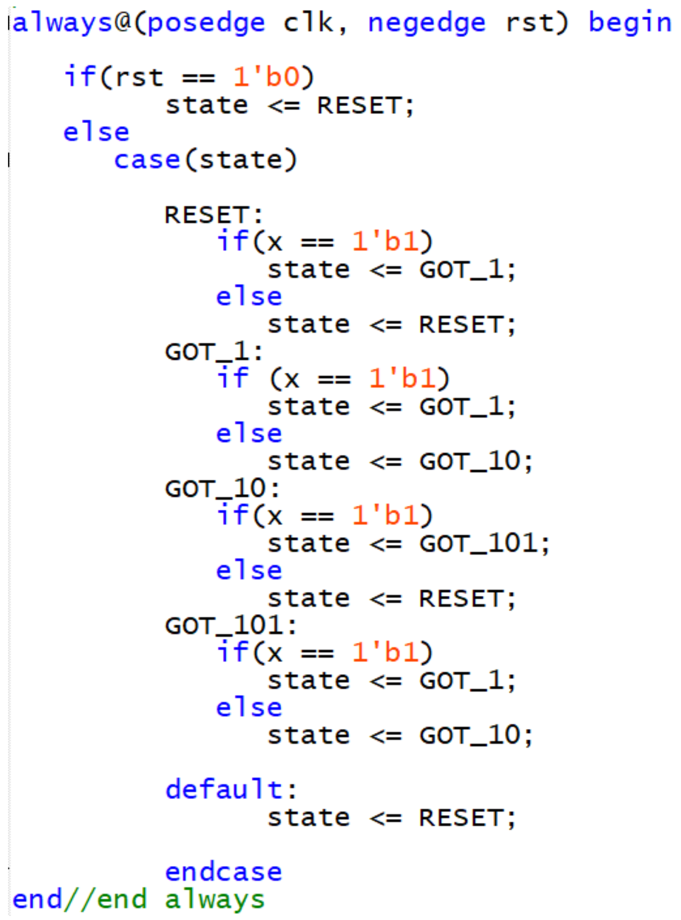


A 101 Mealy Machine

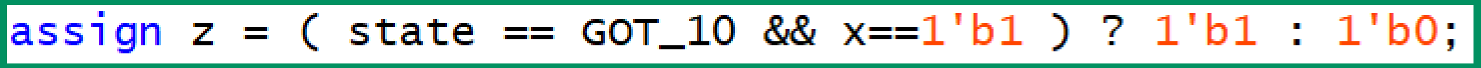
# Mealy State Machine



```
module MealySequenceDetector
(
   // Input Ports
   input clk,
   input rst,
   input x,

   // Output Ports
   output z
);

localparam RESET = 0;
localparam GOT_1 = 1;
localparam GOT_10 = 2;
localparam GOT_101 = 3;


reg [1:0] state;

reg z_b;
```

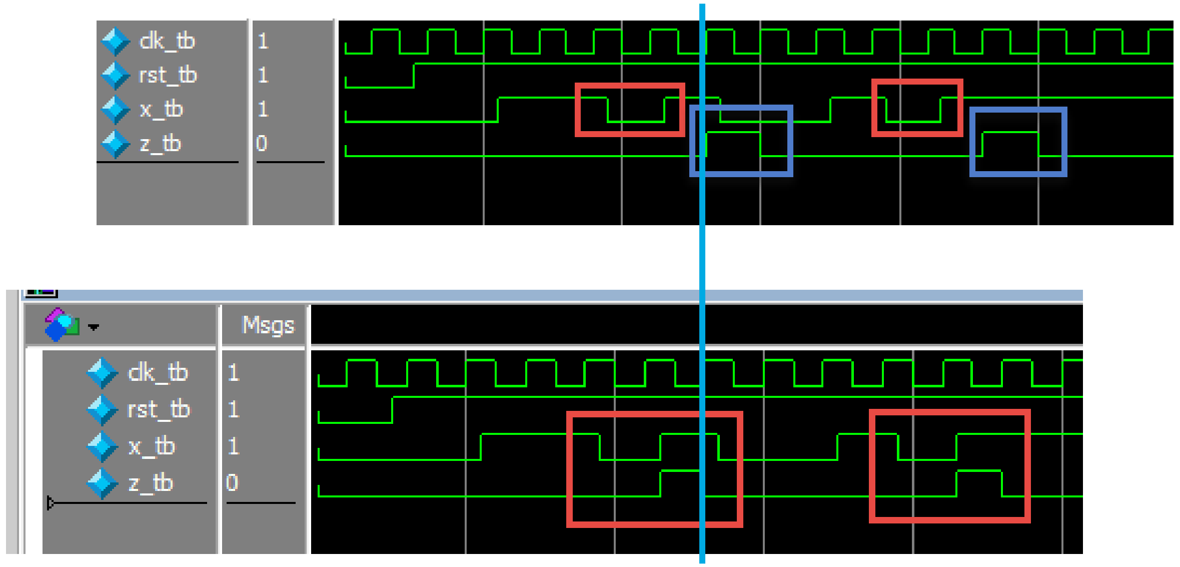


```
always@(posedge clk, negedge rst) begin

   if(rst == 1'b0)
         state <= RESET;
   else
      case(state)

         RESET:
            if(x == 1'b1)
               state <= GOT_1;
            else
               state <= RESET;
         GOT_1:
            if (x == 1'b1)
               state <= GOT_1;
            else
               state <= GOT_10;
         GOT_10:
            if(x == 1'b1)
               state <= GOT_101;
            else
               state <= RESET;
         GOT_101:
            if(x == 1'b1)
               state <= GOT_1;
            else
               state <= GOT_10;

         default:
               state <= RESET;

         endcase
end//end always
```



```
assign z = ( state == GOT_10 && x==1'b1 ) ? 1'b1 : 1'b0;
```

# Mealy and Moore



## State Machines on Quartus

When a synthesis tool recognizes a piece of code as a state machine, it can implement techniques that improve the design area and performance. For example, the tool can recode the state variables to improve the  quality of results, or use the known properties of state machines to optimize other parts of the design.

## State Machines on Quartus

To ensure proper recognition and inference of state machines and to improve the quality of results, observe the following guidelines:

- Assign default values to outputs derived from the state machine so that synthesis does not generate unwanted latches.
- Separate the state machine logic from all arithmetic functions and datapaths, including assigning output values.
- If your design contains an operation that more than one state uses, define the operation outside the state machine and cause the output logic of the state machine to use this value.

## State Machines on Quartus

- Use a simple asynchronous or synchronous reset to ensure a defined power-up state. If your state machine design contains more elaborate reset logic, such as both an asynchronous reset and an asynchronous load, the Quartus Prime software generates regular logic rather than inferring a state machine.
- If you are using the SystemVerilog standard, use enumerated types to describe state machines.
- Do not directly use integer values for state variables, such as next_state <= 0. However, using an integer does not prevent inference in the Quartus Prime software.