University of San Carlos | Department of
COMPUTER ENGINEERING

CpE 3101L – Introduction to HDL

Unit 7: Finite State Machines (FSMs)

Outline

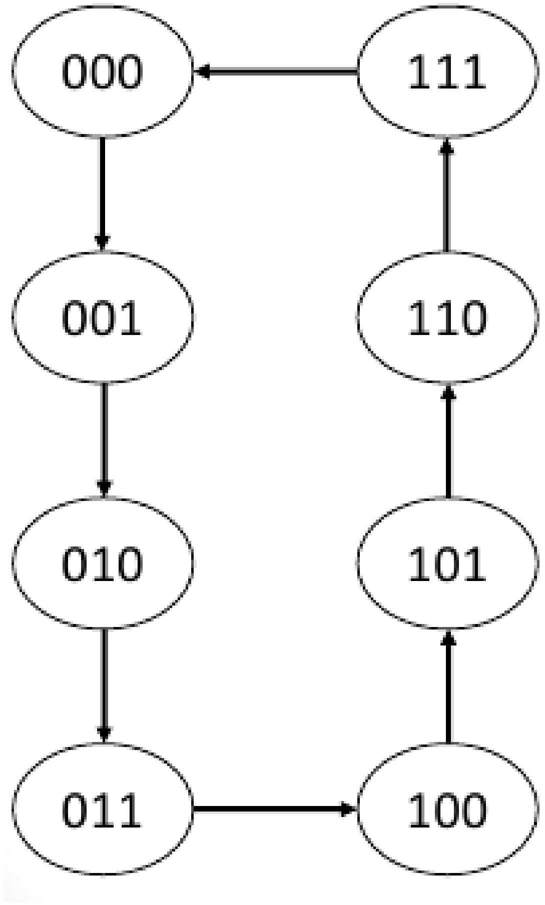
- FSMs in Digital Systems
- Mealy and Moore Models
- State Diagrams as FSM Representations
- Algorithmic State Machines (ASM) and ASM Charts
- Behavioral Modeling of FSMs
- Synthesis of FSMs
- FSM Partitioning Schemes
- State Assignment and Encoding
- Algorithmic State Machine and Datapath (ASMD) Charts
- Design Example with ASMD Chart
- HDL Description of Design Example
- Modular Approach in Complex FSMs

Finite State Machines (FSMs)

- A **state machine** is another term for a sequential circuit, which is the basic structure of a digital system.
- The term ***finite*** refers to a definite, exact, quantifiable number of states this machine/system has.
- **States** are determined by the number of memory elements *(flip-flops)*.
  - Maximum number of states is 2^n where n is the number of flip-flops.

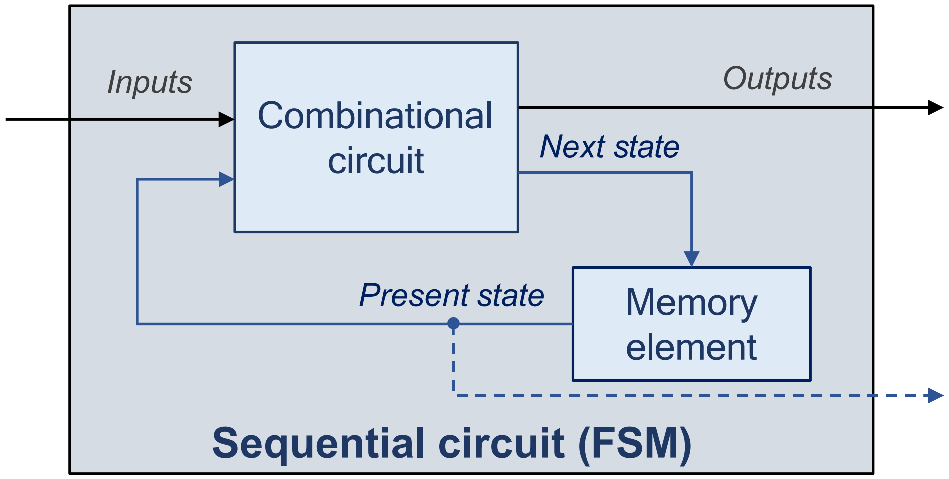
Finite State Machines (FSMs)

- FSMs are usually specified by **state diagrams.**
- Generally composed of three components/ divisions.


$$\begin{tikzcd} 000 \arrow[d] & 111 \arrow[l] \\ 001 \arrow[d] & 110 \arrow[u] \\ 010 \arrow[d] & 101 \arrow[u] \\ 011 \arrow[r] & 100 \arrow[u] \end{tikzcd}$$

Mealy and Moore Models

- The **most general model** of a sequential circuit or an FSM has:
  - Inputs
  - Outputs
  - Internal states

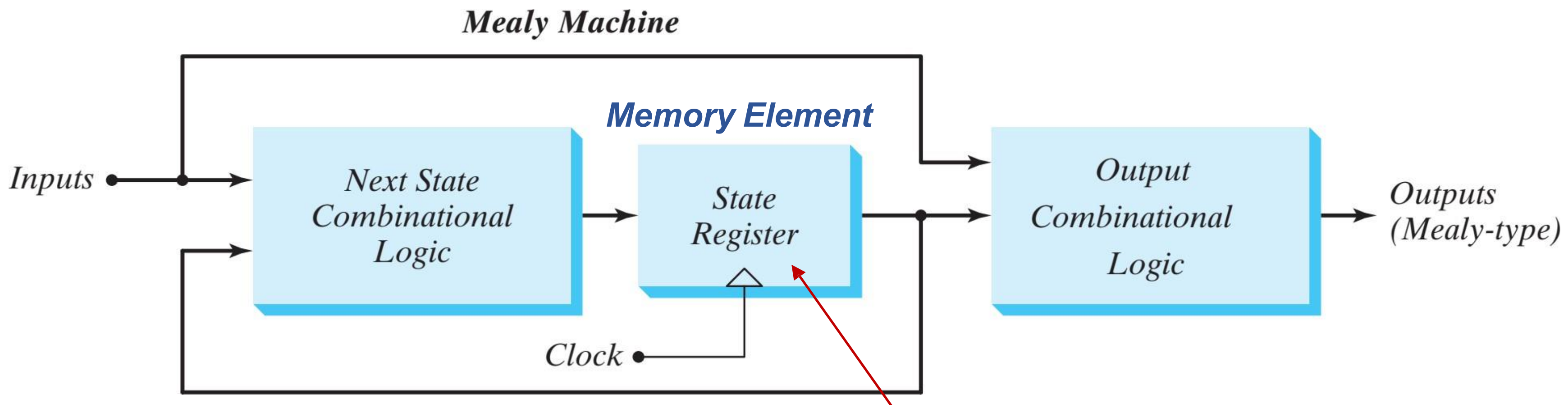


- **Two models:**
  - Mealy
  - Moore

- ***They differ in the way their output is generated.***

Mealy Model

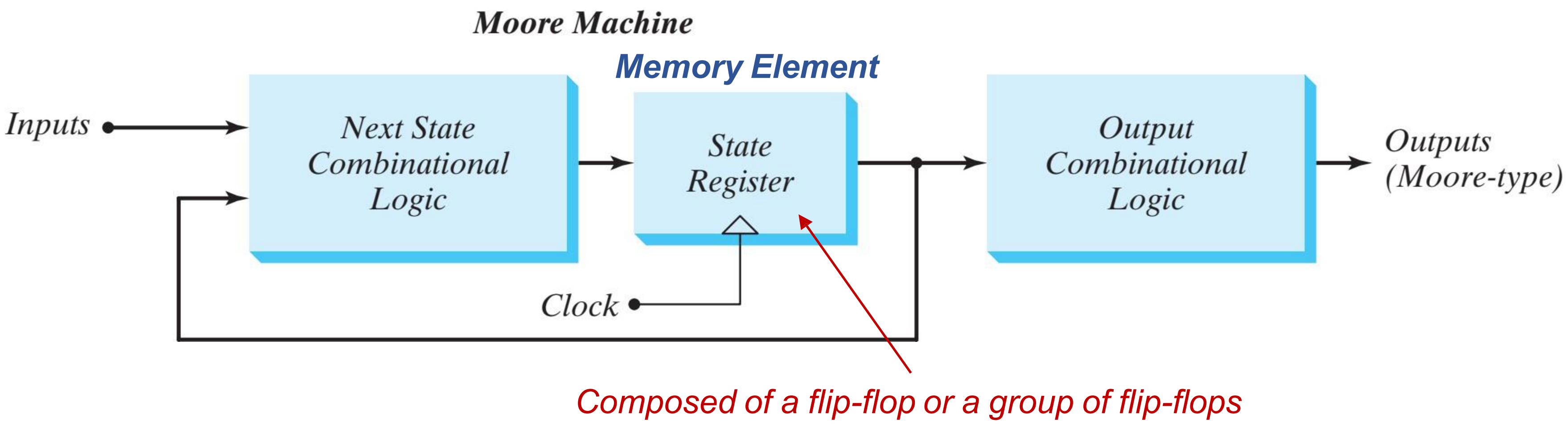
- Also known as **Mealy FSM** or **Mealy machine**
- The **output** is a function of *(is dependent on)* both the present state and the input.
  - **Outputs** of a Mealy FSM may change if the inputs change at any time *(harder to synchronize)*.
  - **States** cannot change until a triggering clock edge.



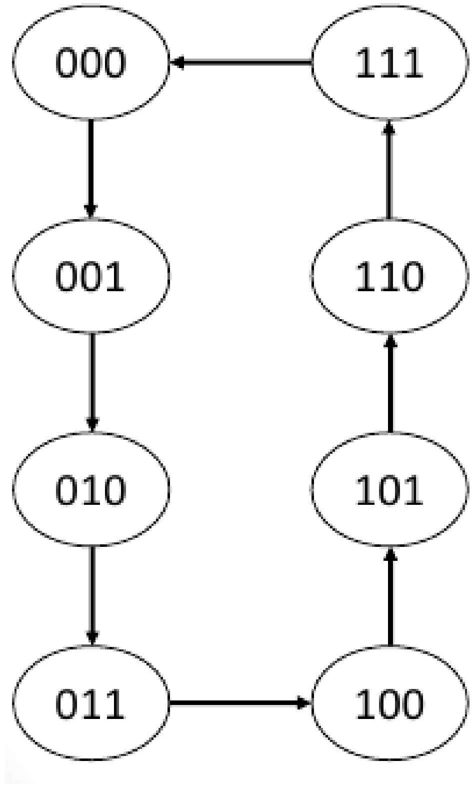
Composed of a flip-flop or a group of flip-flops

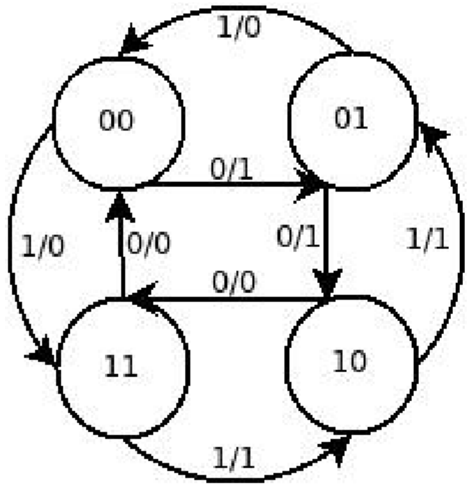
Moore Model

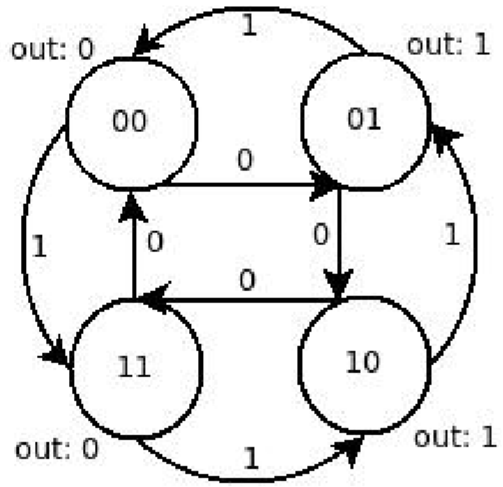
- Also known as **Moore FSM** or **Moore machine**
- The **output** is a function of *(is dependent on)* only the present state.
    - **Outputs** and **states** of the Moore FSM are synchronized with the clock.


Moore Machine
Memory Element
Inputs
Next State Combinational Logic
State Register
Output Combinational Logic
Outputs (Moore-type)
Clock
Composed of a flip-flop or a group of flip-flops

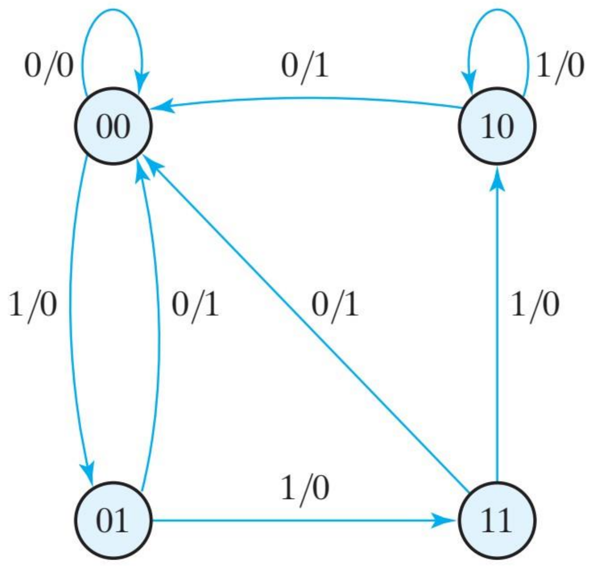
Mealy or Moore?

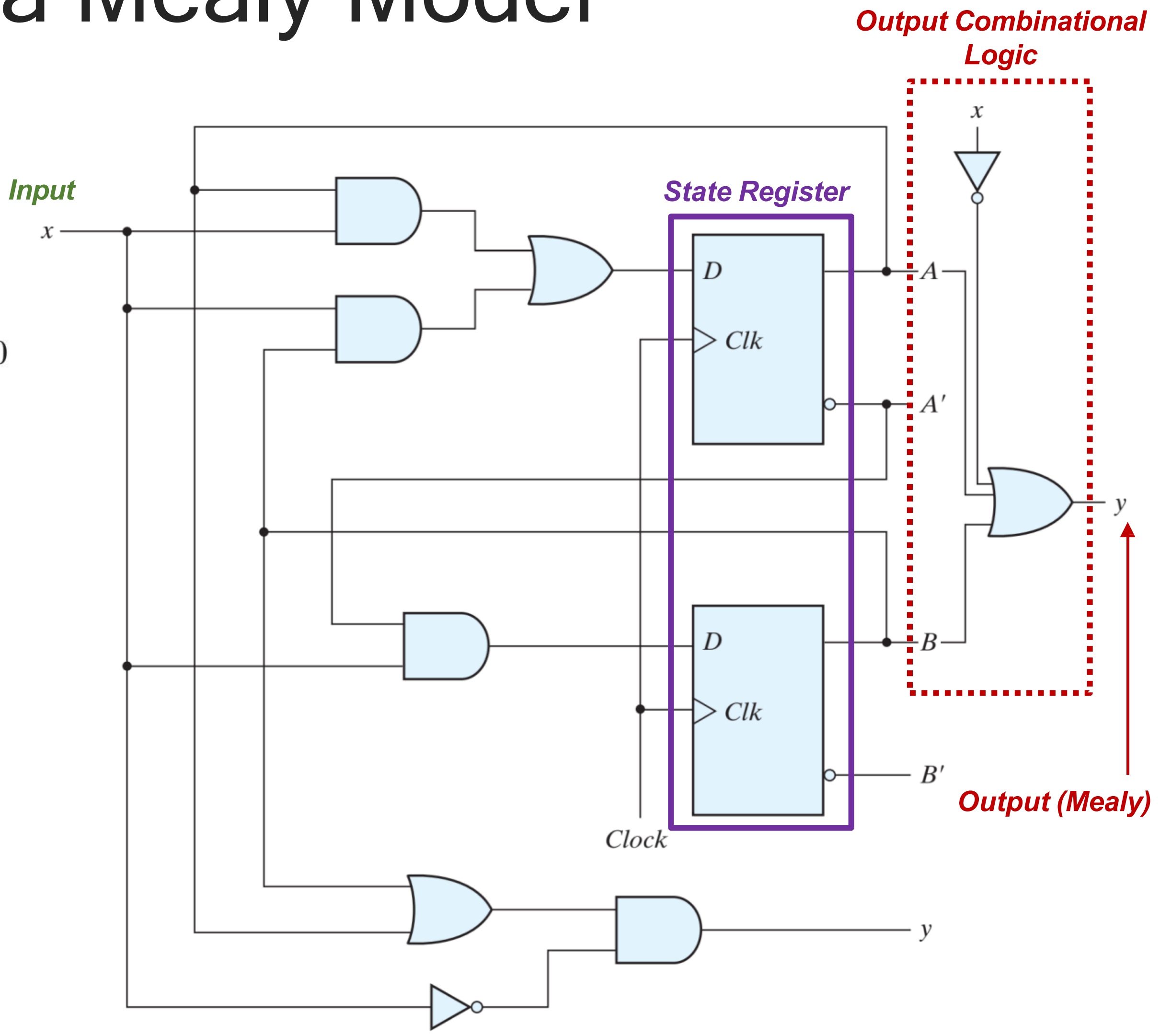






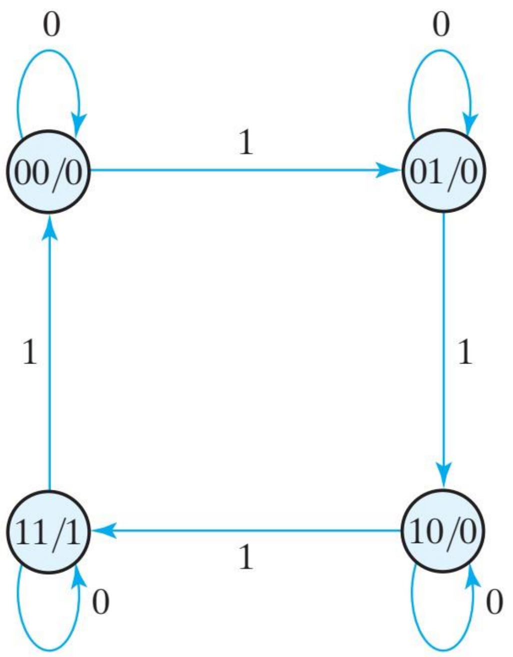
Example of a Mealy Model

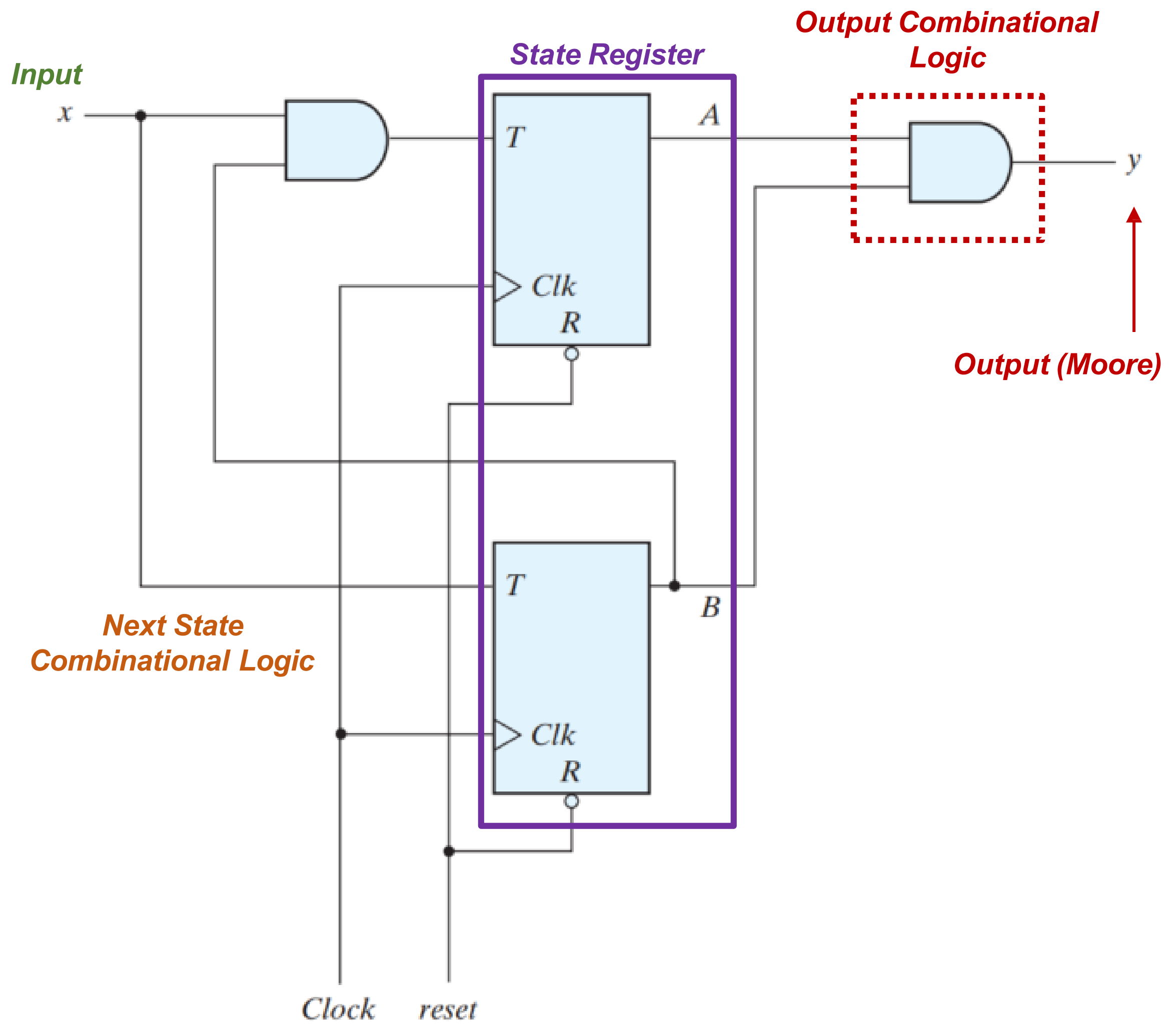




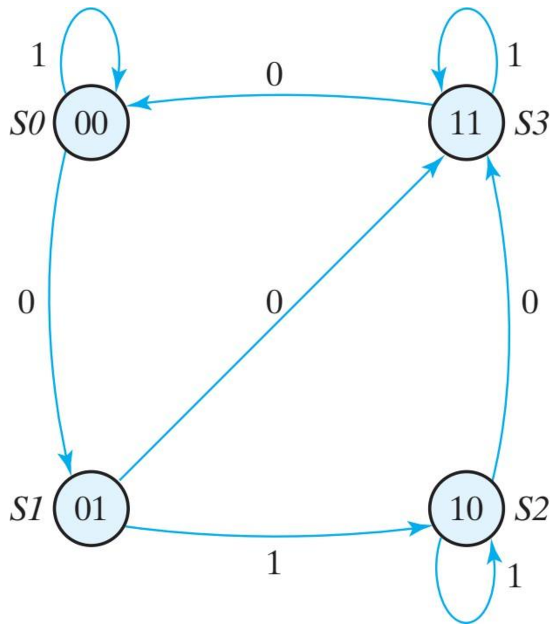
Next State Combinational Logic

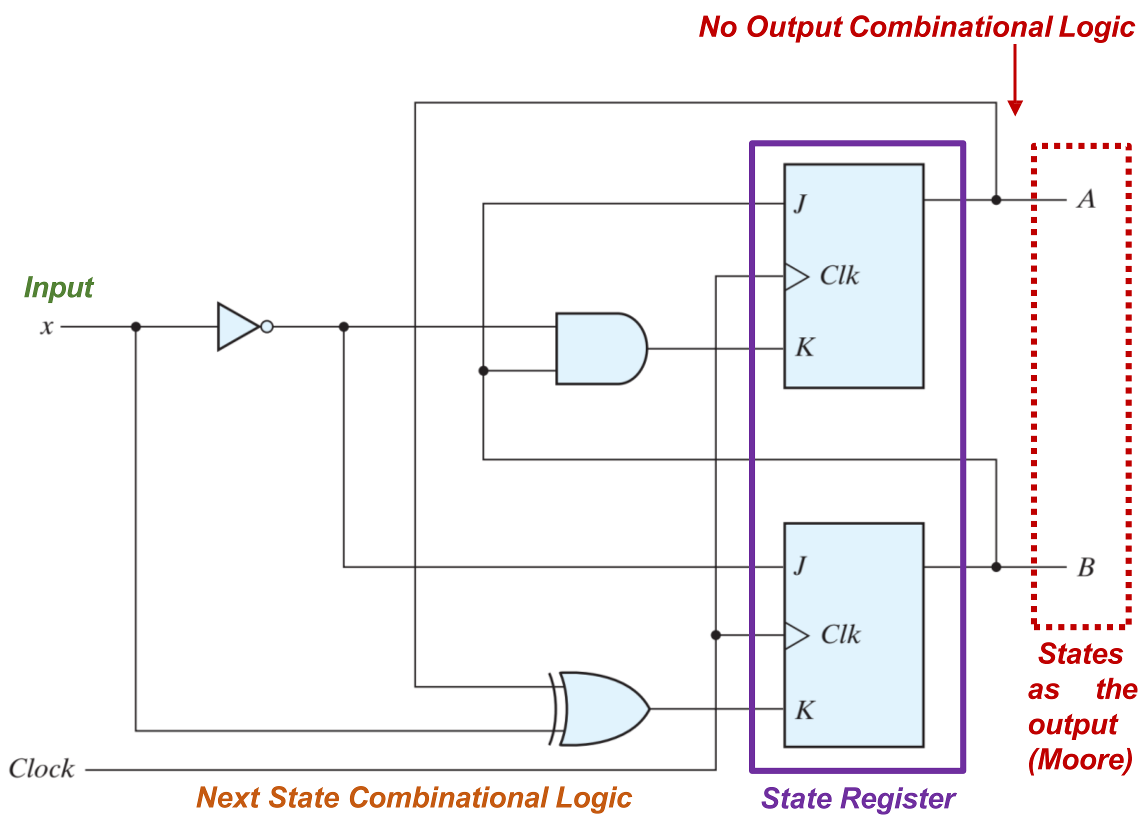
Example of a Moore Model





Another Example of a Moore Model



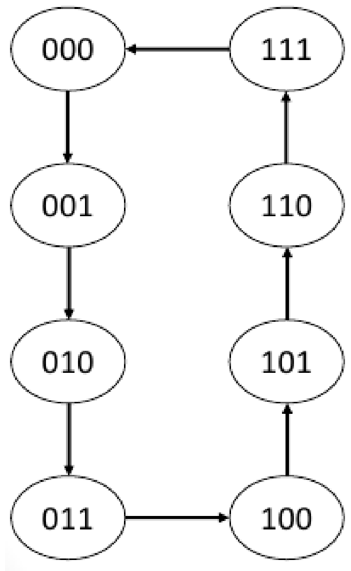

No Output Combinational Logic
Input
x
J
Clk
K
J
Clk
K
A
B
Clock
Next State Combinational Logic
State Register
States as the output (Moore)

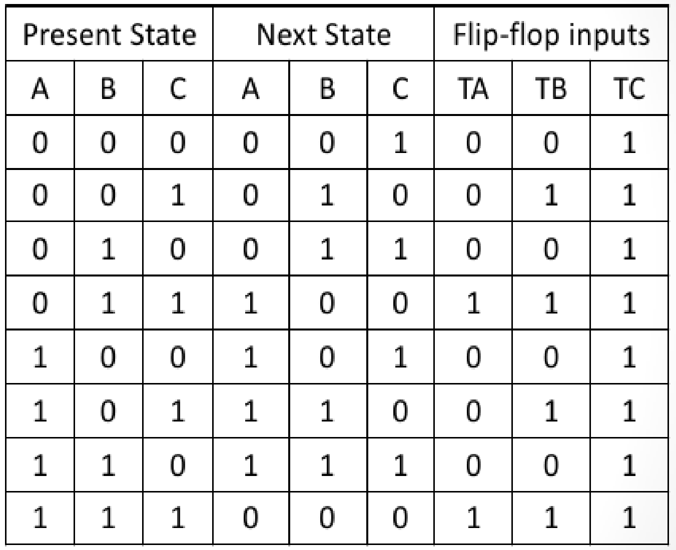
Synthesis of Sequential Circuits

1. Describe circuit into FSM and draw state diagram.
2. State minimization *(if possible)*
3. Draw state table and determine flip-flop excitation equations.
4. Construct circuit using derived equations.

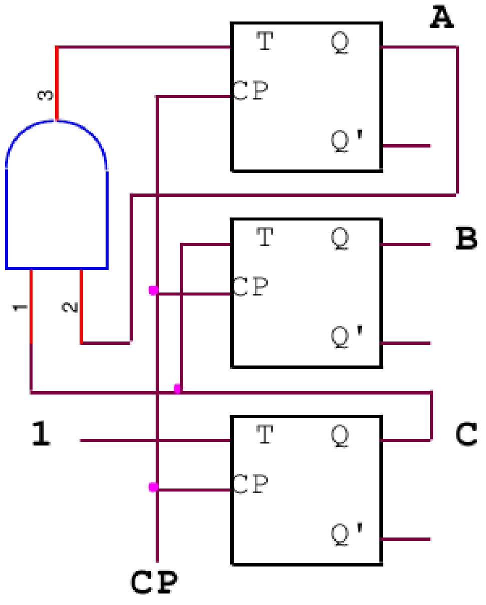
Traditional (Manual) Method

- Example: **3-Bit Counter (Moore Model)**





Present State			Next State			Flip-flop inputs		
A	B	C	A	B	C	TA	TB	TC
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	0	0	1
1	1	1	0	0	0	1	1	1



In this case, state minimization (step #2) is no longer needed.

BC

A	00	01	11	10
0			1	
1			1	

TA = BC

Bx

A	00	01	11	10
0		1	1	
1		1	1	

TB = C

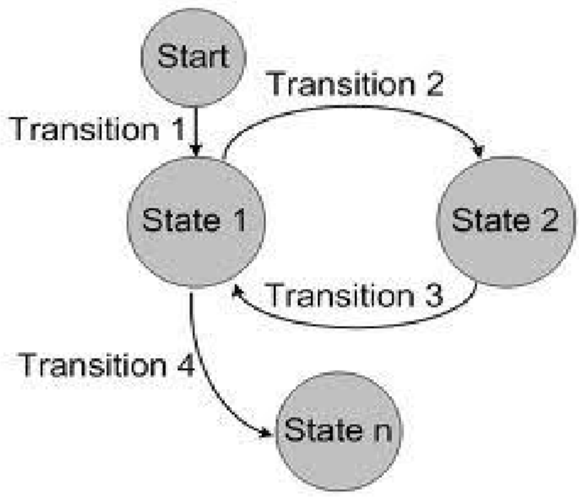
Bx

A	00	01	11	10
0	1	1	1	1
1	1	1	1	1

TC = 1

FSM Description in Verilog HDL

- No need to manually determine flip-flop excitations.
- The **HDL synthesis tool** will automatically perform steps for synthesis and can even make optimizations.
- Describe FSM using the **behavioral model**.

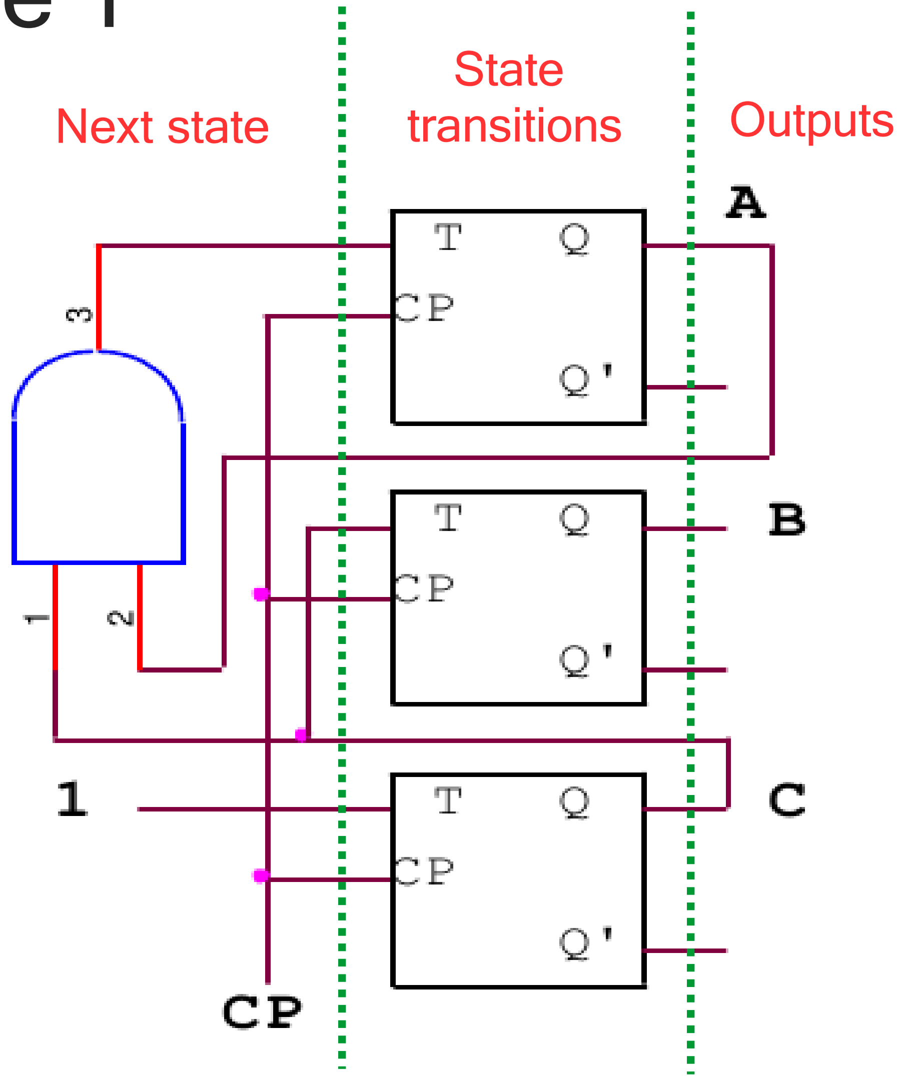


```
always @ (posedge Clk or negedge nReset) begin
     if (!nReset)
          Q <= 1'b0;
     else
          Q <= D;
end
```

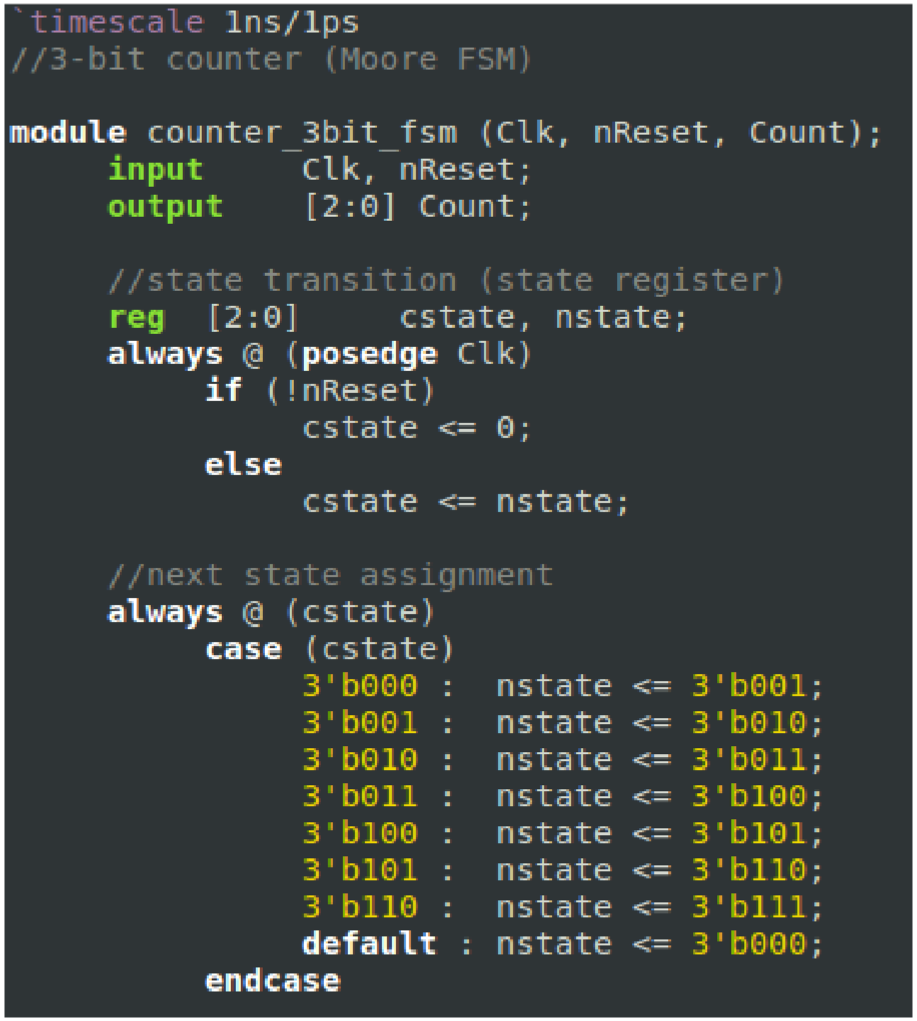
```
d_ff u1 (Clk, nReset, x1, x0);
and  u2 (x1, x10, x99);
or   u3 (x10, x70, x13);
     ...
```

Partitioning Scheme 1

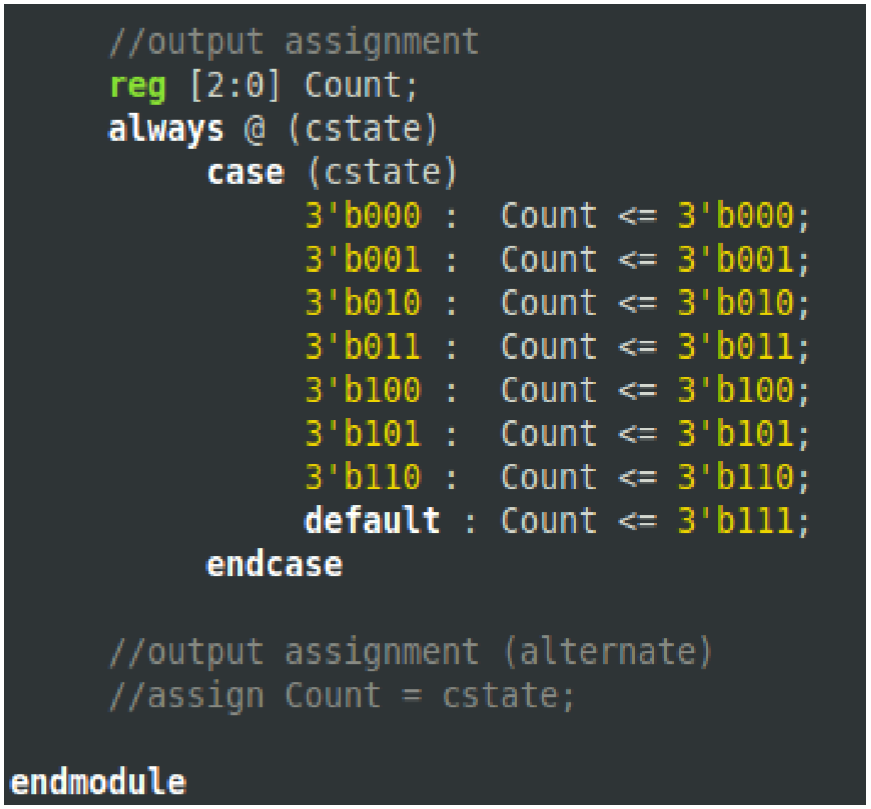
- There are many ways to describe FSMs in Verilog.
- One partitioning scheme is to **separate the description of**:
  - State transitions
  - Determining the next states, and
  - Determining the outputs



Partitioning Scheme 1



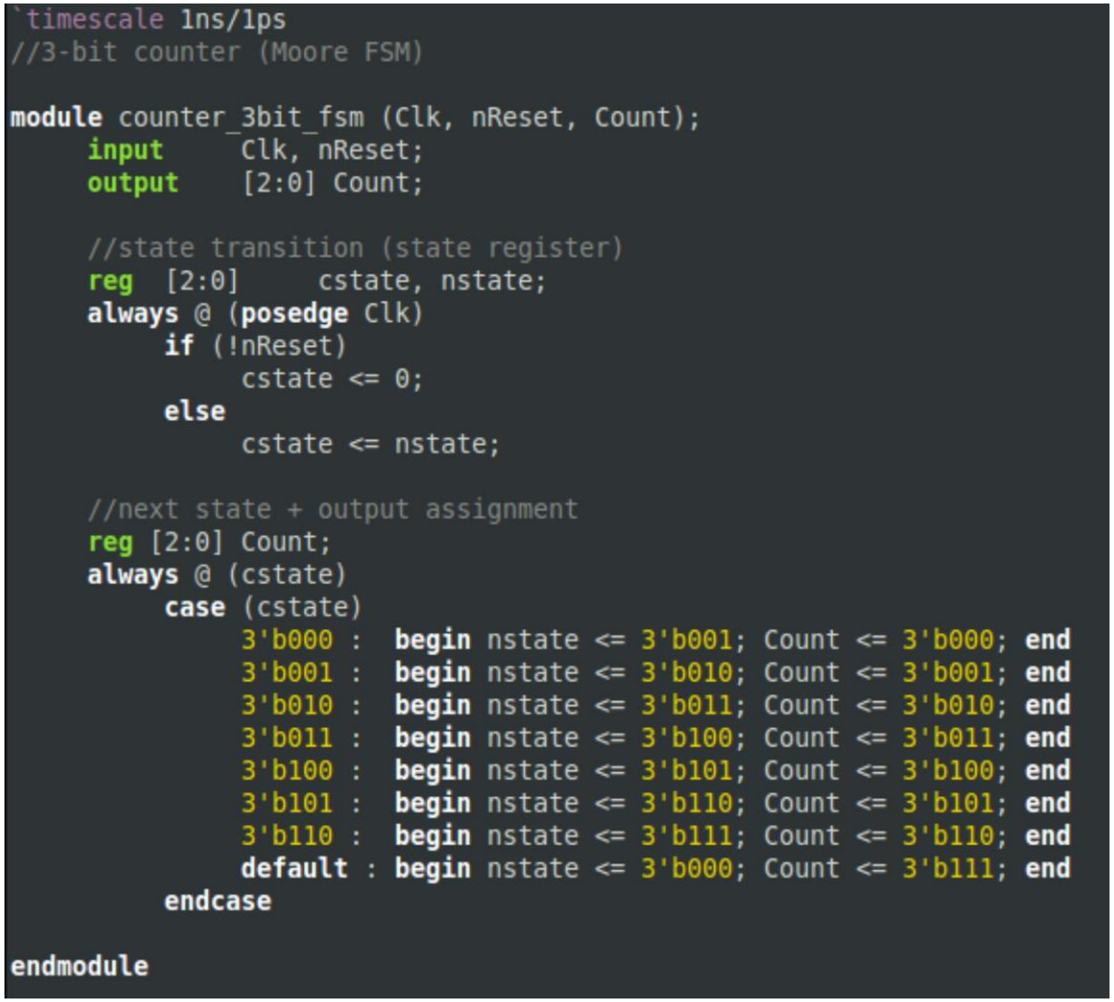
```
`timescale 1ns/1ps
//3-bit counter (Moore FSM)

module counter_3bit_fsm (Clk, nReset, Count);
     input      Clk, nReset;
     output     [2:0] Count;

     //state transition (state register)
     reg   [2:0]     cstate, nstate;
     always @ (posedge Clk)
          if (!nReset)
               cstate <= 0;
          else
               cstate <= nstate;

     //next state assignment
     always @ (cstate)
          case (cstate)
               3'b000 :  nstate <= 3'b001;
               3'b001 :  nstate <= 3'b010;
               3'b010 :  nstate <= 3'b011;
               3'b011 :  nstate <= 3'b100;
               3'b100 :  nstate <= 3'b101;
               3'b101 :  nstate <= 3'b110;
               3'b110 :  nstate <= 3'b111;
               default : nstate <= 3'b000;
          endcase
```



```
     //output assignment
     reg [2:0] Count;
     always @ (cstate)
          case (cstate)
               3'b000 :  Count <= 3'b000;
               3'b001 :  Count <= 3'b001;
               3'b010 :  Count <= 3'b010;
               3'b011 :  Count <= 3'b011;
               3'b100 :  Count <= 3'b100;
               3'b101 :  Count <= 3'b101;
               3'b110 :  Count <= 3'b110;
               default : Count <= 3'b111;
          endcase

     //output assignment (alternate)
     //assign Count = cstate;

endmodule
```

Partitioning Scheme 2

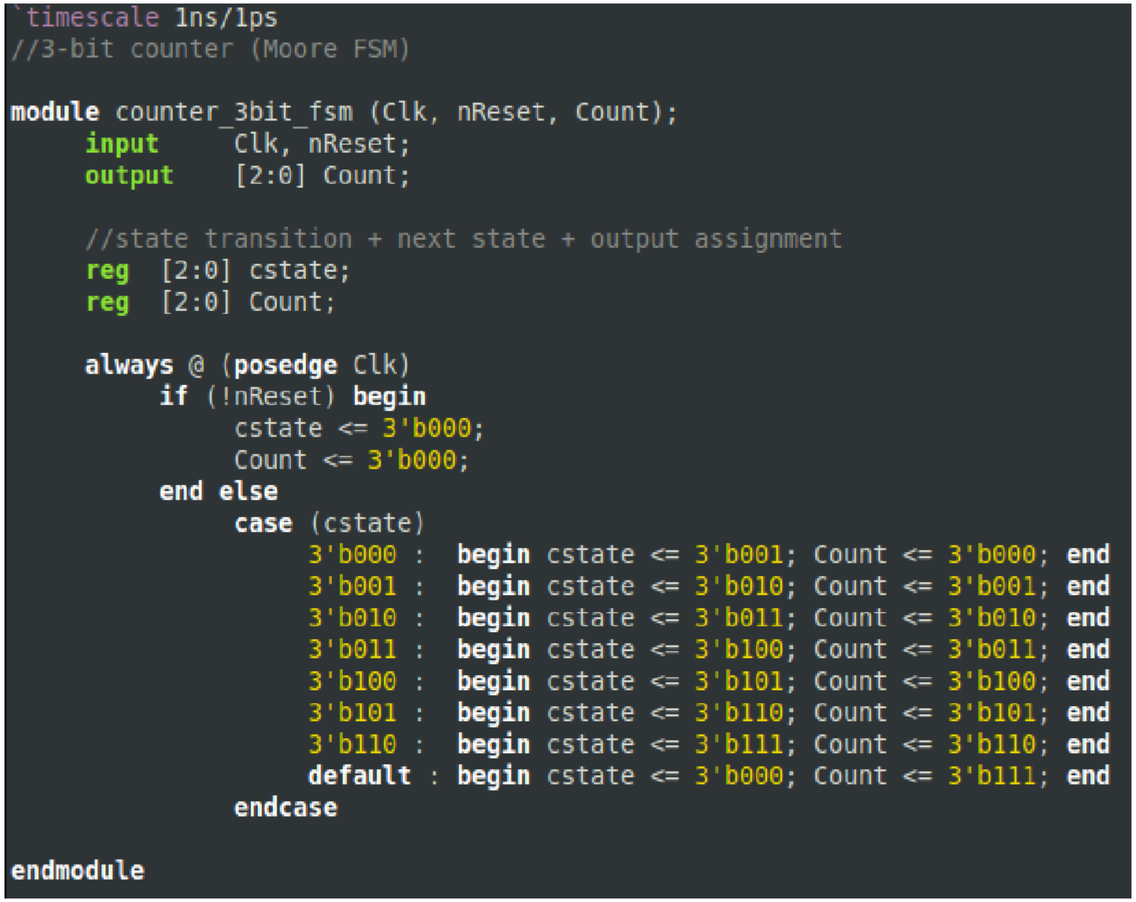
- There are **2 partitions:**
    - State transitions
    - Combined description of the next state and output assignments



```
`timescale 1ns/1ps
//3-bit counter (Moore FSM)

module counter_3bit_fsm (Clk, nReset, Count);
    input     Clk, nReset;
    output    [2:0] Count;

    //state transition (state register)
    reg   [2:0]     cstate, nstate;
    always @ (posedge Clk)
        if (!nReset)
            cstate <= 0;
        else
            cstate <= nstate;

    //next state + output assignment
    reg [2:0] Count;
    always @ (cstate)
        case (cstate)
            3'b000 :  begin nstate <= 3'b001; Count <= 3'b000; end
            3'b001 :  begin nstate <= 3'b010; Count <= 3'b001; end
            3'b010 :  begin nstate <= 3'b011; Count <= 3'b010; end
            3'b011 :  begin nstate <= 3'b100; Count <= 3'b011; end
            3'b100 :  begin nstate <= 3'b101; Count <= 3'b100; end
            3'b101 :  begin nstate <= 3'b110; Count <= 3'b101; end
            3'b110 :  begin nstate <= 3'b111; Count <= 3'b110; end
            default : begin nstate <= 3'b000; Count <= 3'b111; end
        endcase

endmodule
```

Partitioning Scheme 3

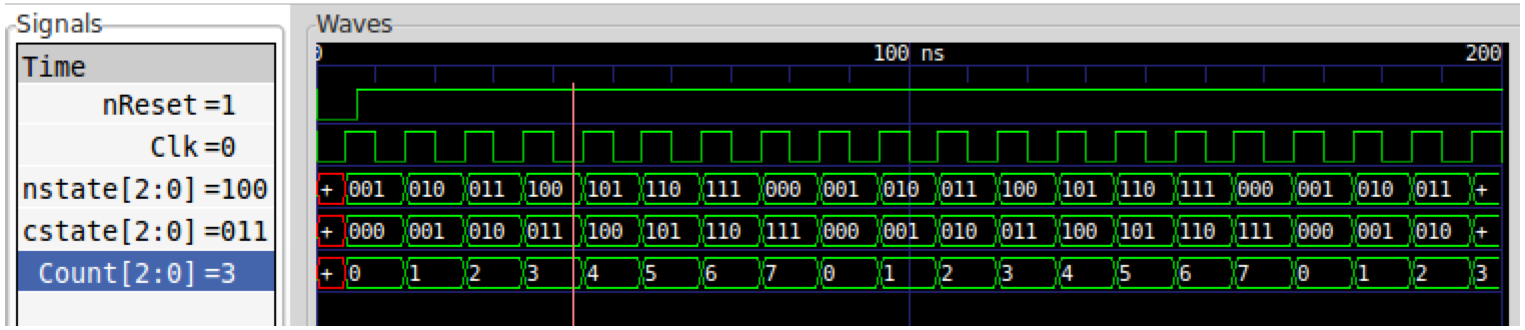
- There is just **1 partition:**
  - All 3 components (state transitions, next state, and output assignments) are combined into a single ***always*** block.



```
`timescale 1ns/1ps
//3-bit counter (Moore FSM)

module counter_3bit_fsm (Clk, nReset, Count);
     input     Clk, nReset;
     output    [2:0] Count;

     //state transition + next state + output assignment
     reg  [2:0] cstate;
     reg  [2:0] Count;

     always @ (posedge Clk)
          if (!nReset) begin
               cstate <= 3'b000;
               Count <= 3'b000;
          end else
               case (cstate)
                    3'b000 :  begin cstate <= 3'b001; Count <= 3'b000; end
                    3'b001 :  begin cstate <= 3'b010; Count <= 3'b001; end
                    3'b010 :  begin cstate <= 3'b011; Count <= 3'b010; end
                    3'b011 :  begin cstate <= 3'b100; Count <= 3'b011; end
                    3'b100 :  begin cstate <= 3'b101; Count <= 3'b100; end
                    3'b101 :  begin cstate <= 3'b110; Count <= 3'b101; end
                    3'b110 :  begin cstate <= 3'b111; Count <= 3'b110; end
                    default : begin cstate <= 3'b000; Count <= 3'b111; end
               endcase

endmodule
```

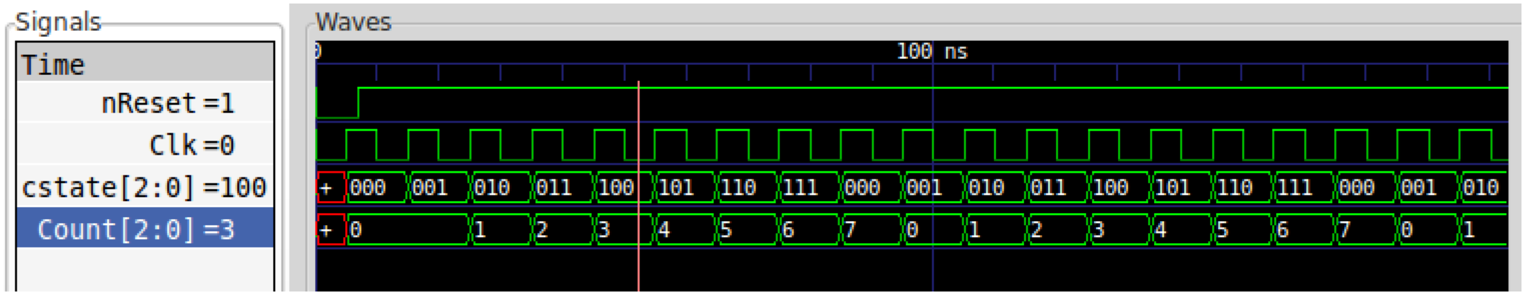
Compare Waveforms

Partitioning Scheme 1 or 2



```
Time
nReset=1
Clk=0
nstate[2:0]=100
cstate[2:0]=011
Count[2:0]=3
```

Partitioning Scheme 3



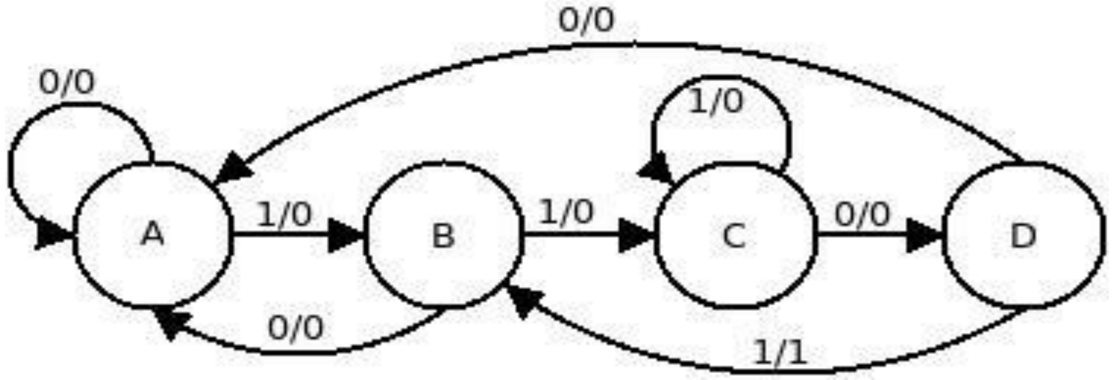
```
Time
nReset=1
Clk=0
cstate[2:0]=100
Count[2:0]=3
```

Example 1: **Sequence Recognizer Problem**

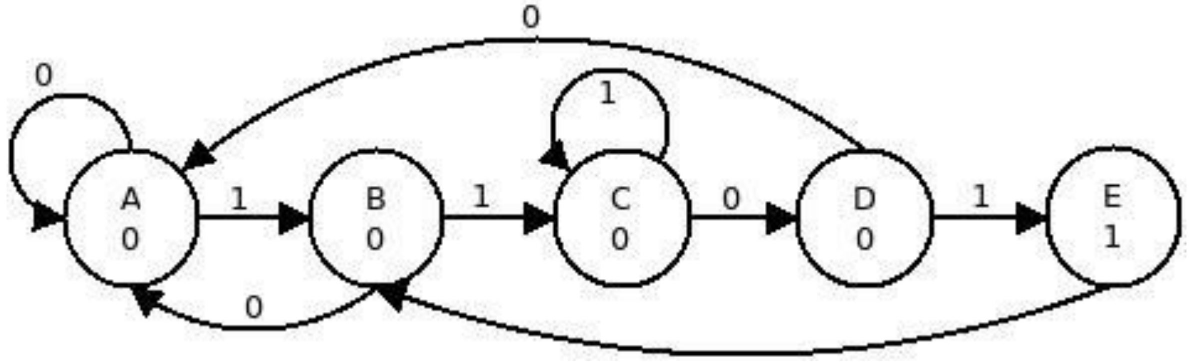
- Design an FSM that recognizes the occurrence of a particular sequence of bits, regardless of where it occurs in a longer sequence. It has to have one input X, output Z, and direct resets on its flip-flops to initialize the state of the circuit to all zeros. The circuit is to recognize the occurrence of the sequence of bits **1101** on X by making Z equal to 1 when the previous inputs of the circuit were 110 and the current input is a 1. Otherwise, Z is 0.

Example 1: State Diagram

- **Mealy FSM**



- **Moore FSM**



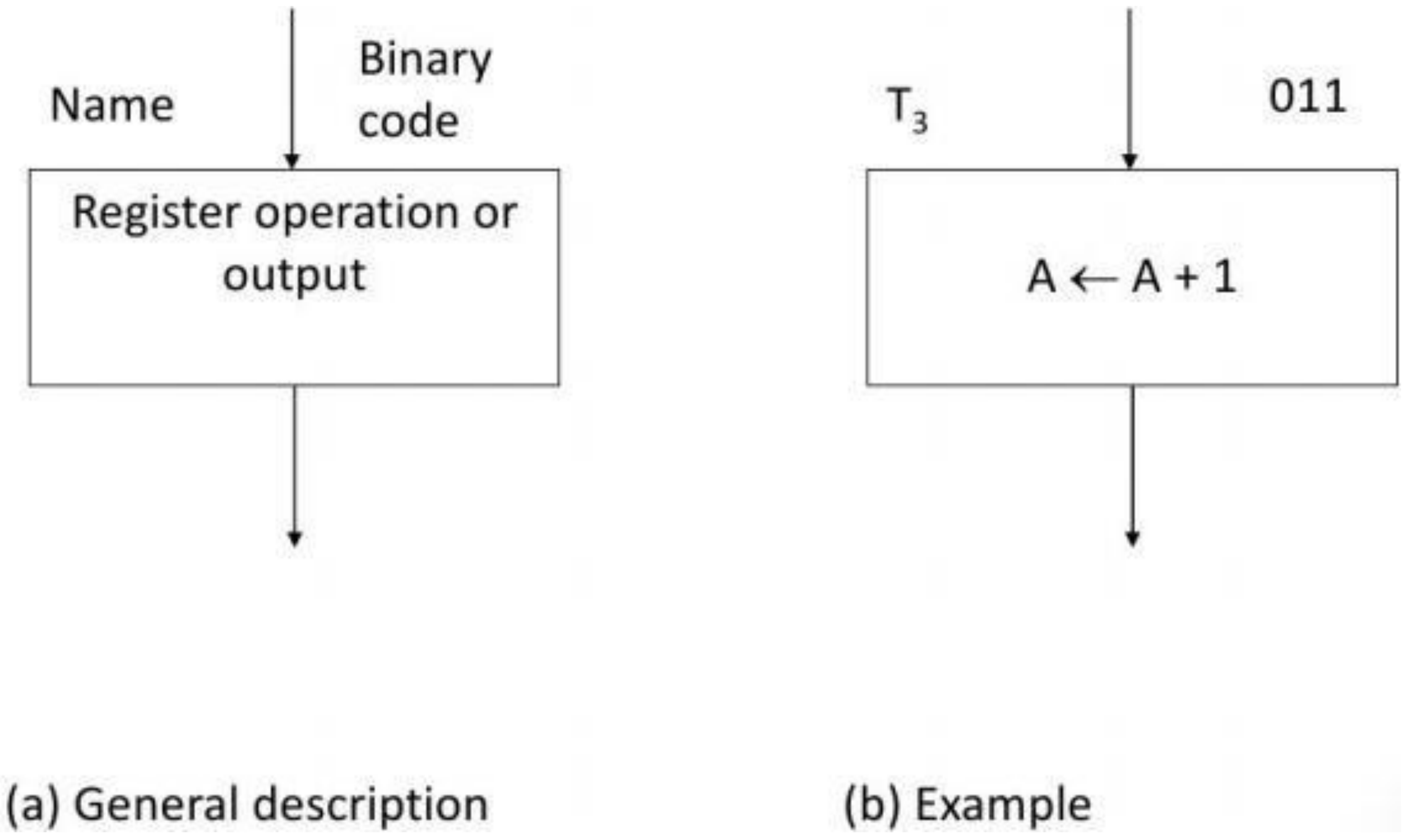
Algorithmic State Machine (ASM) Chart

- A **flowchart** is a convenient way to specify the sequence of procedural steps and decision paths for an algorithm.

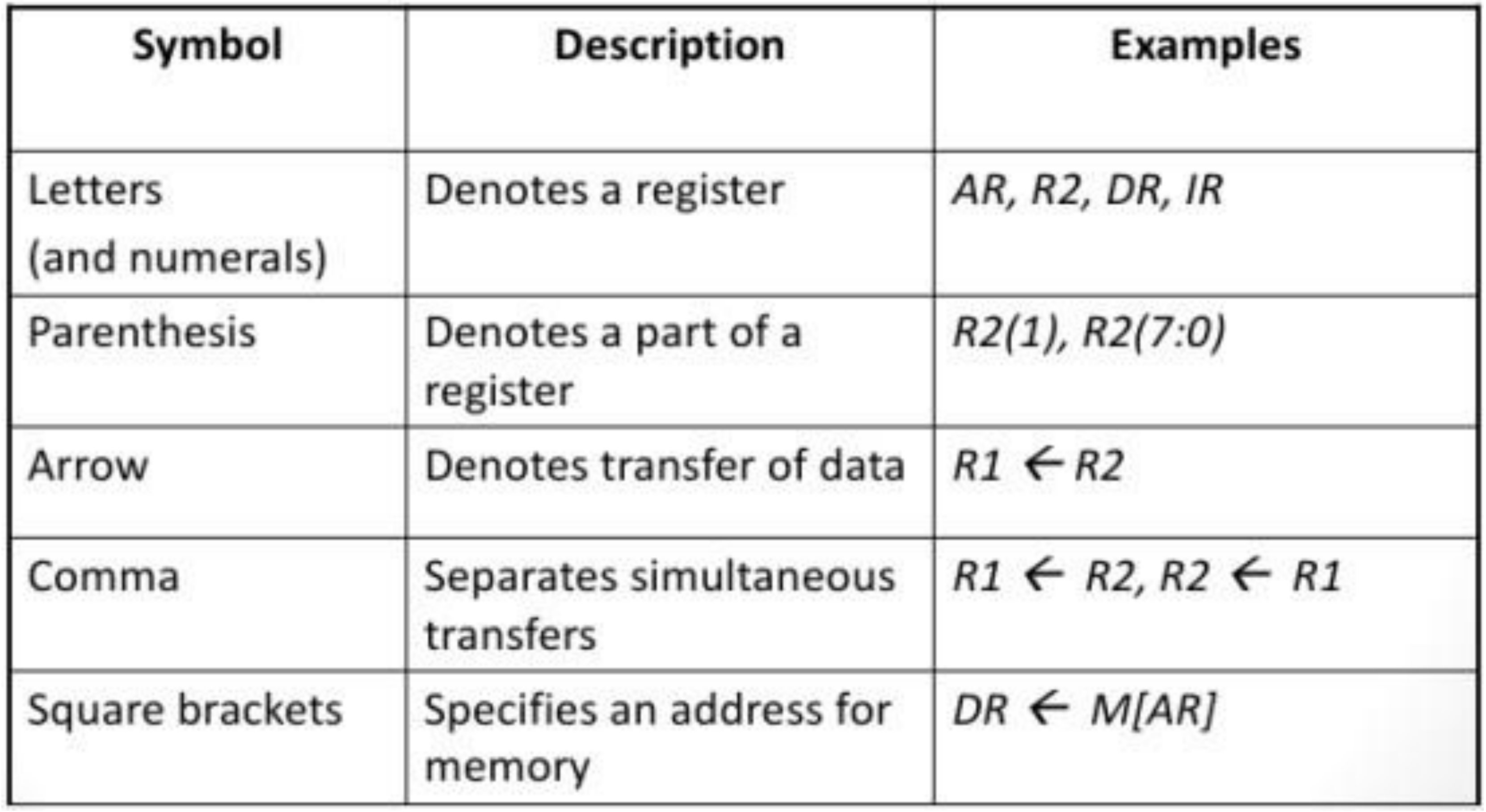
- A special flow chart that has been developed specifically to define digital hardware algorithms is called an **algorithmic state machine (ASM) chart**. It is suitable for describing the sequential operations in a digital system.

- 3 basic elements:
  - State box
  - Decision box
  - Conditional box

State Box

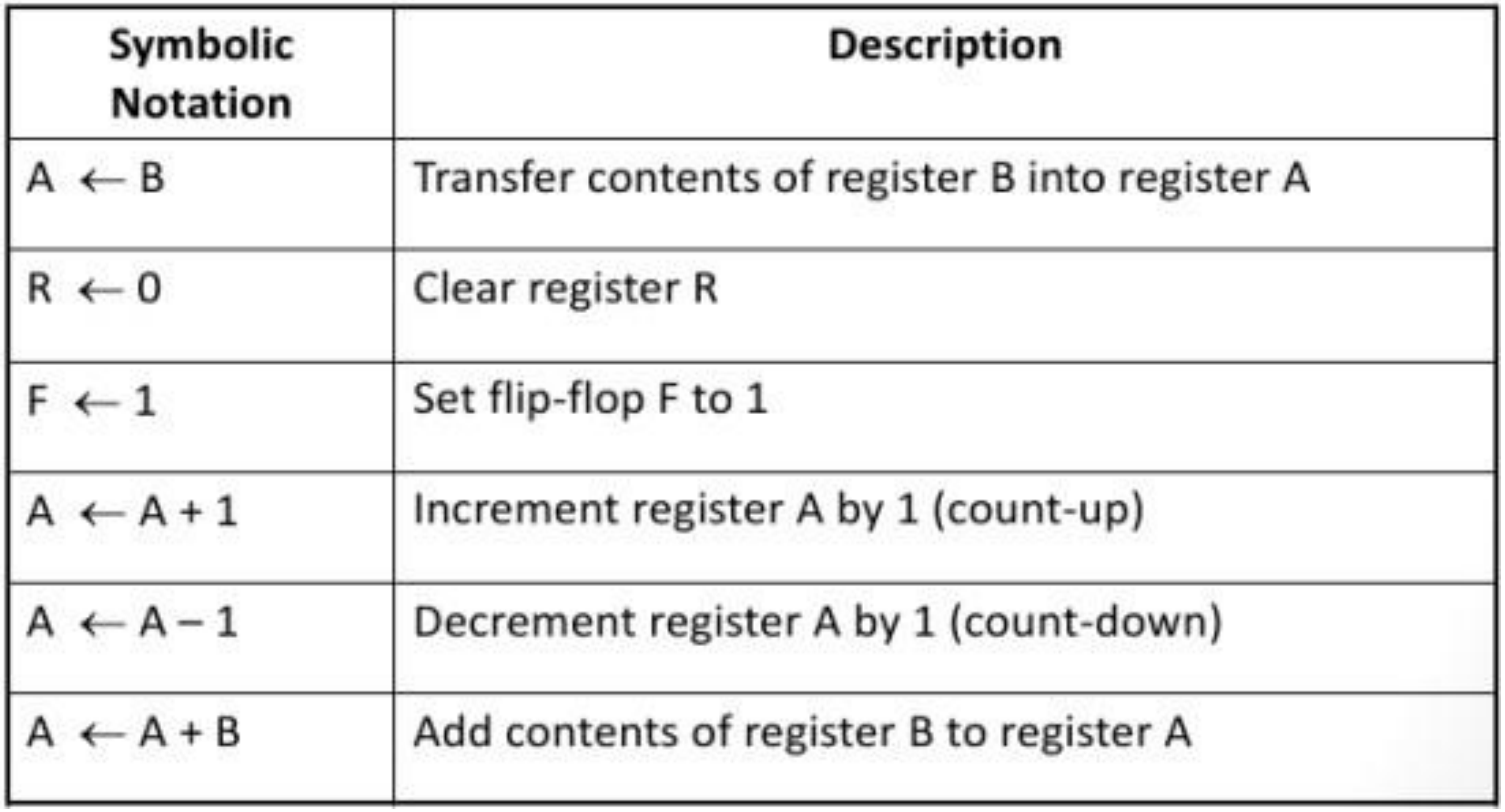

Name
Binary code
Register operation or output
(a) General description
T_3
011
$A \leftarrow A + 1$
(b) Example

Register-Transfer Language



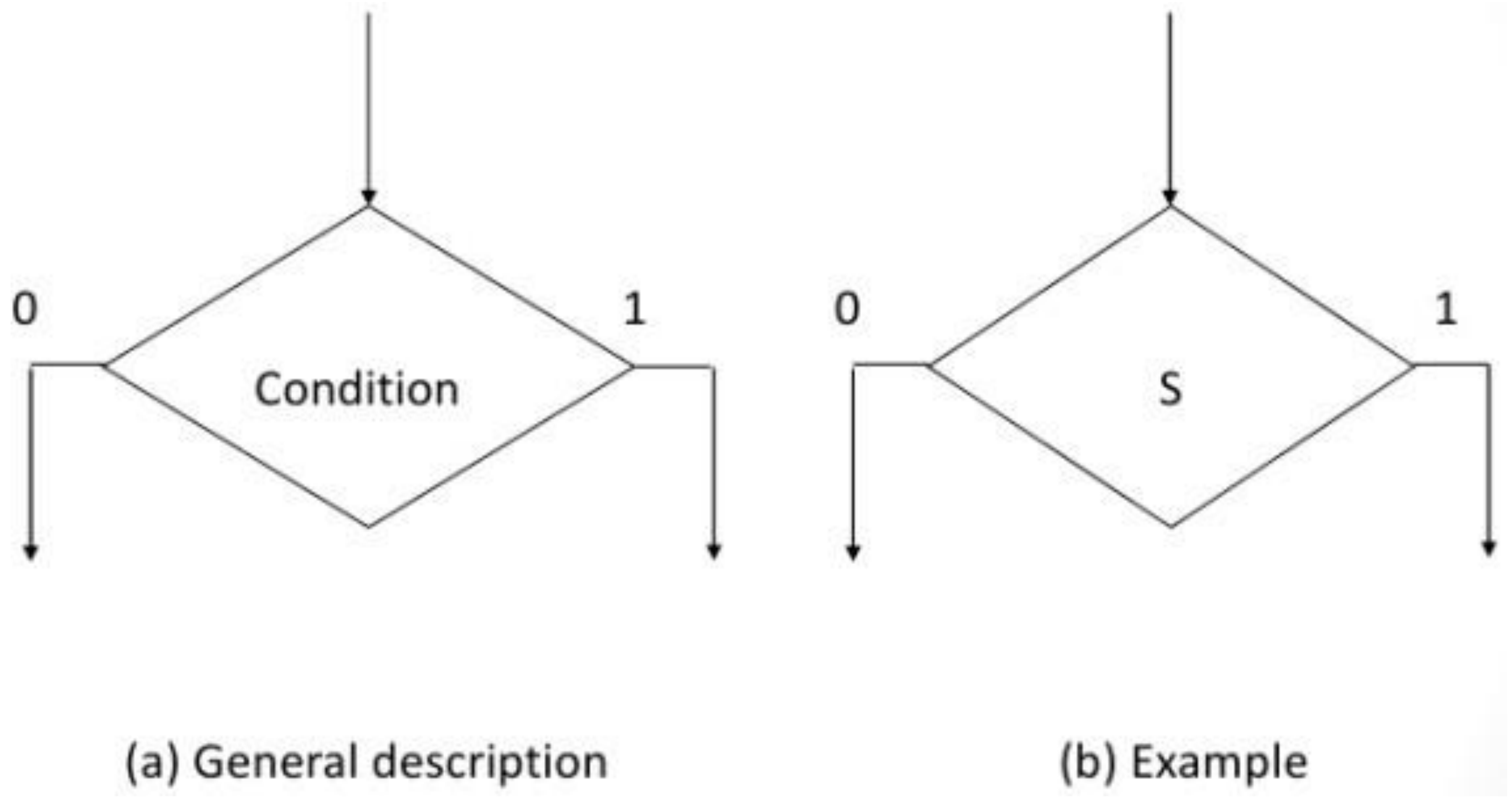
Symbol	Description	Examples
Letters (and numerals)	Denotes a register	*AR, R2, DR, IR*
Parenthesis	Denotes a part of a register	*R2(1), R2(7:0)*
Arrow	Denotes transfer of data	*R1* $\leftarrow$ *R2*
Comma	Separates simultaneous transfers	*R1* $\leftarrow$ *R2, R2* $\leftarrow$ *R1*
Square brackets	Specifies an address for memory	*DR* $\leftarrow$ *M[AR]*

Register-Transfer Language

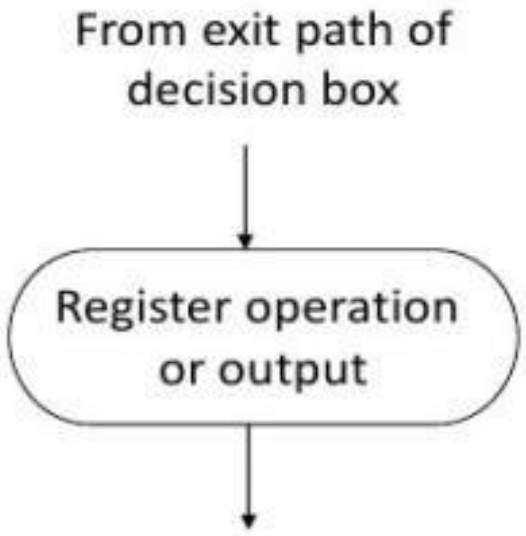


Symbolic Notation	Description
$A \leftarrow B$	Transfer contents of register B into register A
$R \leftarrow 0$	Clear register R
$F \leftarrow 1$	Set flip-flop F to 1
$A \leftarrow A + 1$	Increment register A by 1 (count-up)
$A \leftarrow A - 1$	Decrement register A by 1 (count-down)
$A \leftarrow A + B$	Add contents of register B to register A

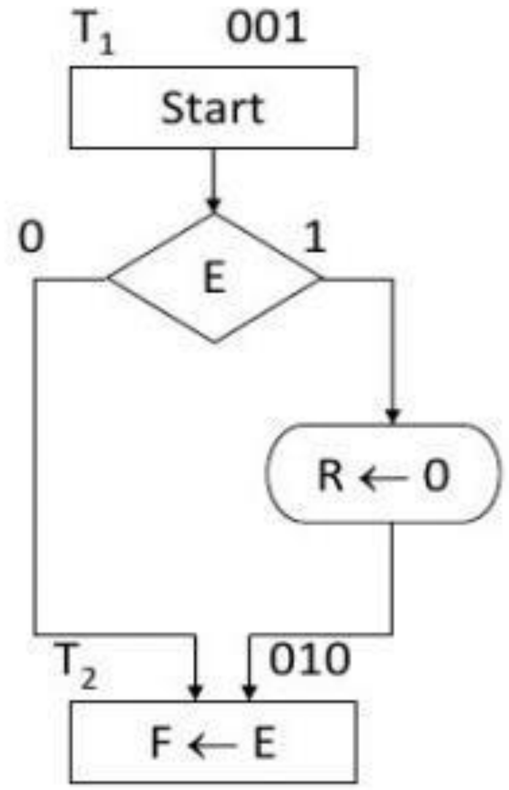
Decision Box



Conditional Box

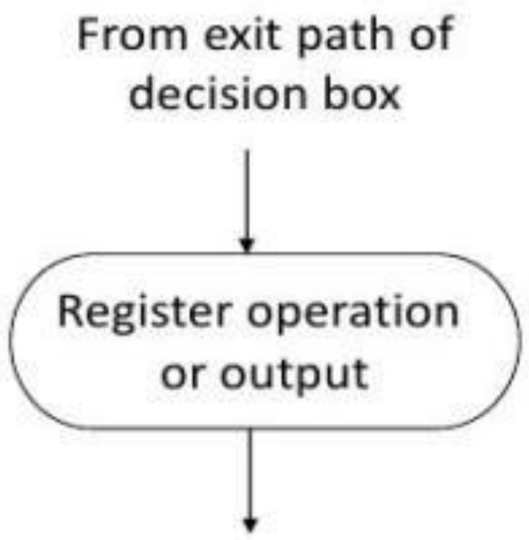


(a) General description

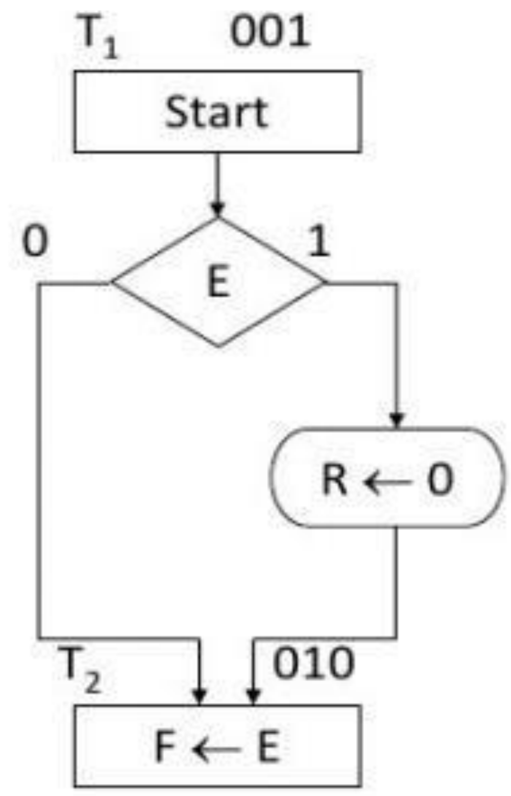


(b) Example with all boxes

Conditional Box



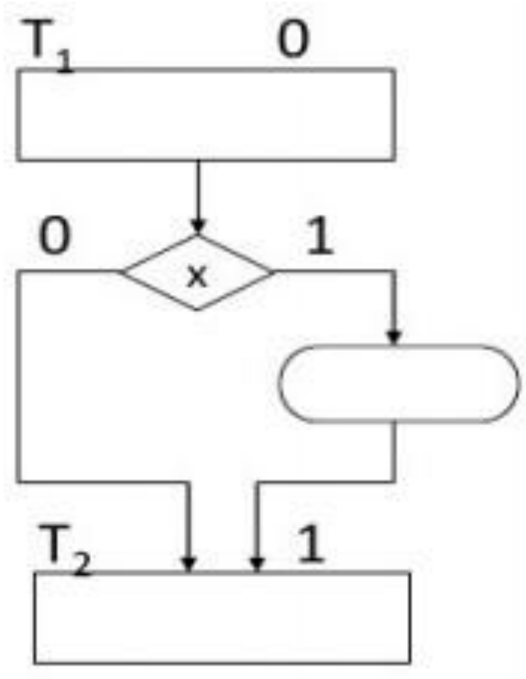
(a) General description



(b) Example with all boxes

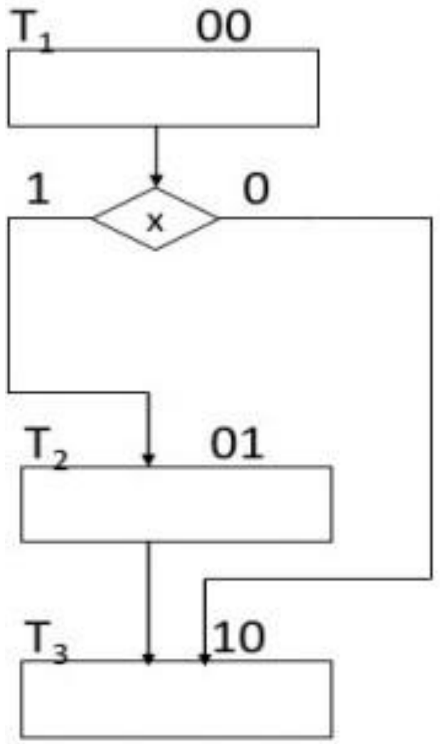
Example 1:

- If x = 0, control goes from state T1 to state T2; if x = 1, generate a conditional operation and go from T1 to T2.



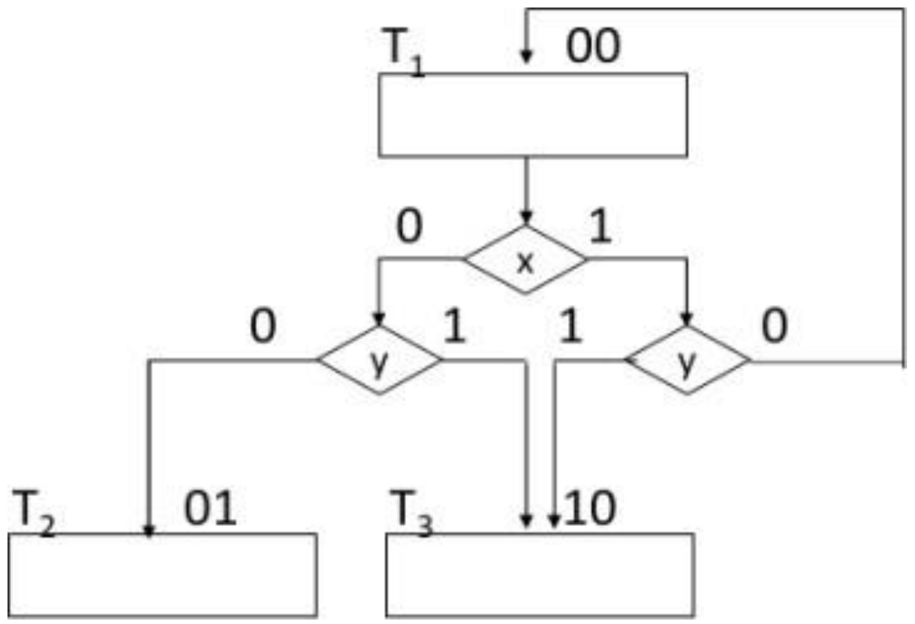
Example 2:

- If x = 1, control goes from T1 to T2 and then to T3; if x = 0, control goes from T1 to T3.

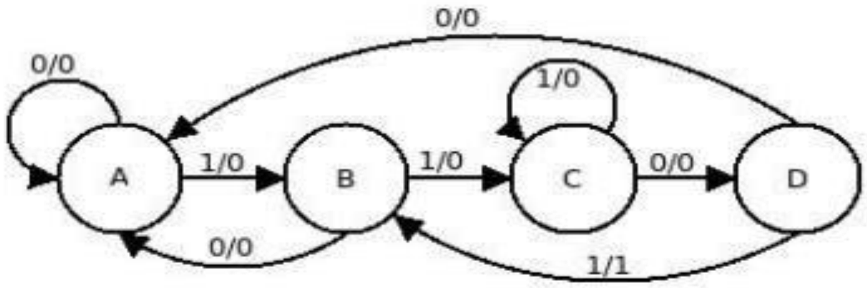


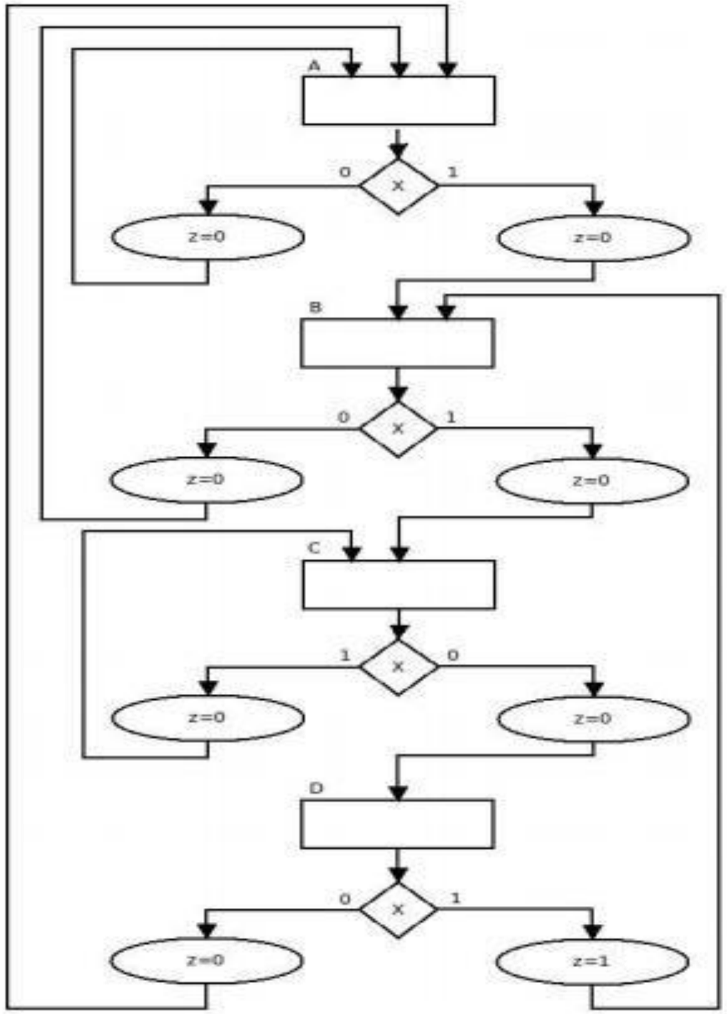
Example 3:

- Start from state T1; then: if xy = 00, go to T2; if xy = 01, go to T3; if xy = 10, go to T1; otherwise, go to T3.

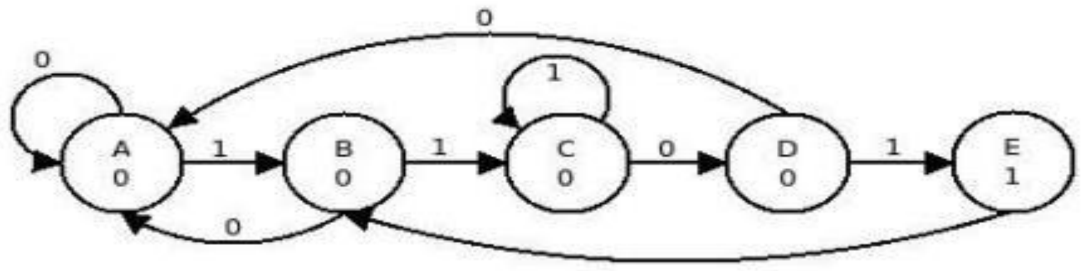


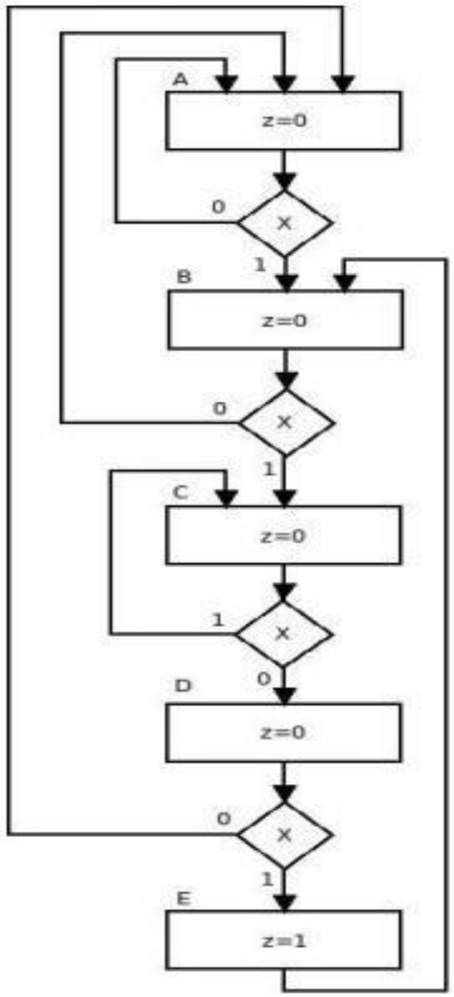
Mealy State Diagram vs. ASM Chart





Moore State Diagram vs. ASM Chart





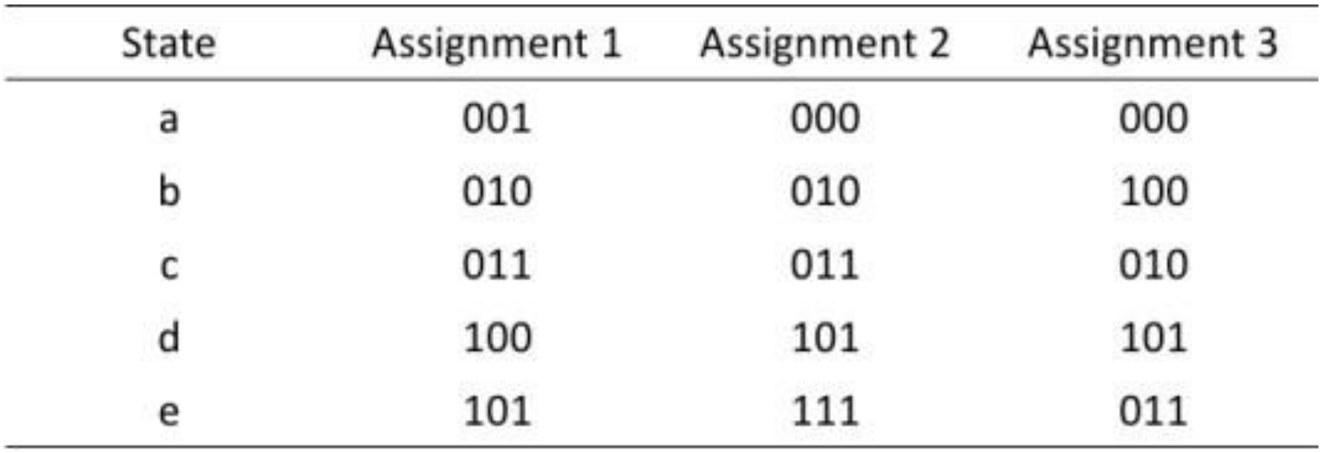
State Assignment and Encoding

- There are **2 ways**:

- **Hard encoding**
  - Using binary values to represent states
  - Different schemes: *Sequential (binary), Random, Gray, Johnson, One hot, One cold*

- **Soft encoding**
  - Usually used in HDLs through abstract state representation *(the enumerated states)*
  - Use ***parameter*** or ***localparam*** keywords to represent states

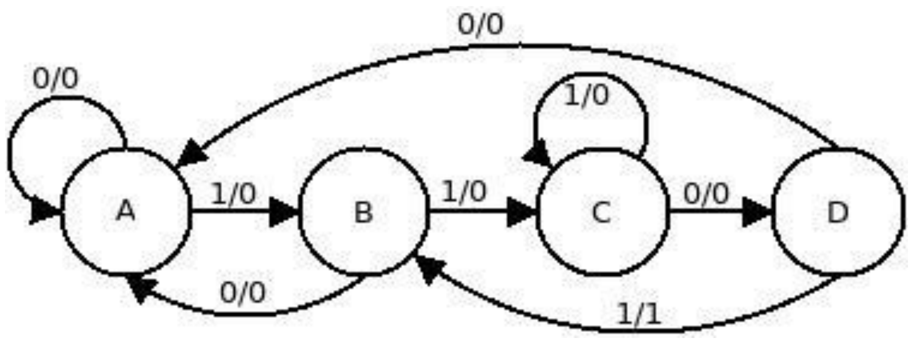
State Assignment and Encoding

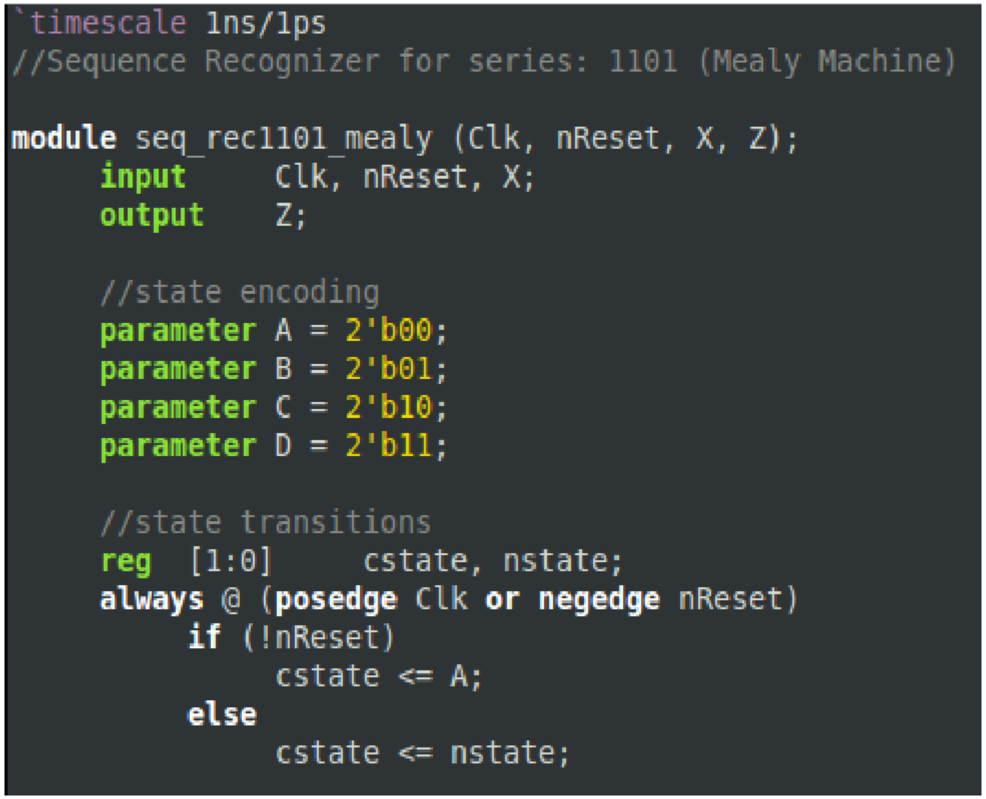


State	Assignment 1	Assignment 2	Assignment 3
a	001	000	000
b	010	010	100
c	011	011	010
d	100	101	101
e	101	111	011

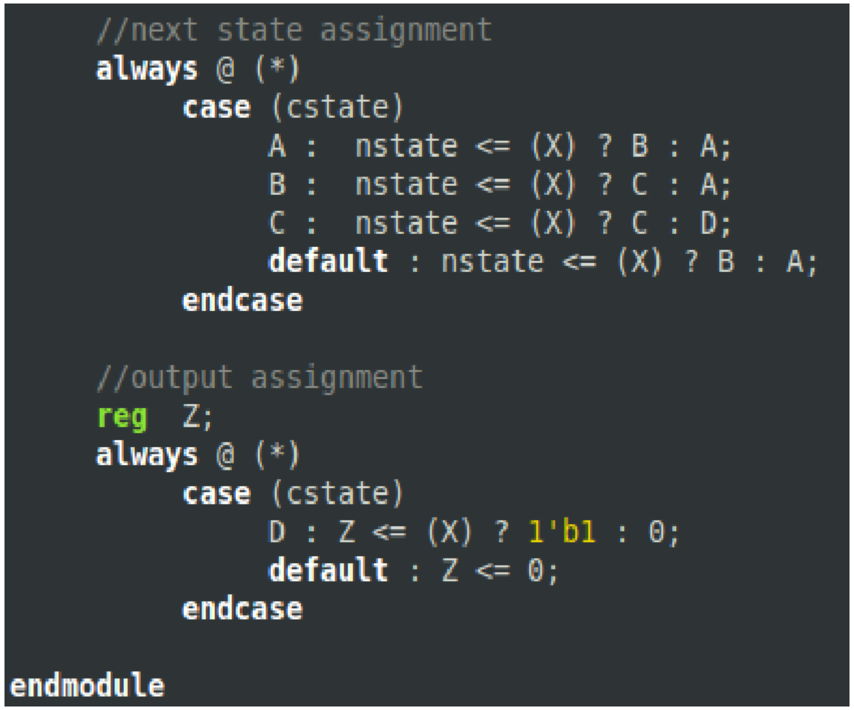
Example 1:

- **Mealy Model**





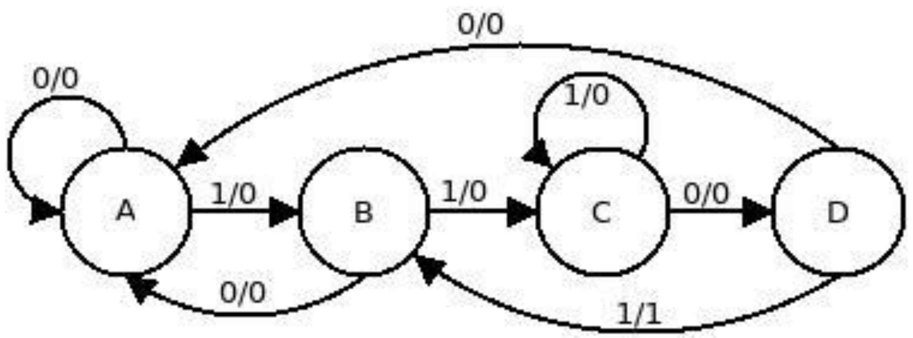
```
`timescale 1ns/1ps
//Sequence Recognizer for series: 1101 (Mealy Machine)

module seq_rec1101_mealy (Clk, nReset, X, Z);
     input      Clk, nReset, X;
     output     Z;

     //state encoding
     parameter A = 2'b00;
     parameter B = 2'b01;
     parameter C = 2'b10;
     parameter D = 2'b11;

     //state transitions
     reg   [1:0]     cstate, nstate;
     always @ (posedge Clk or negedge nReset)
          if (!nReset)
               cstate <= A;
          else
               cstate <= nstate;
```

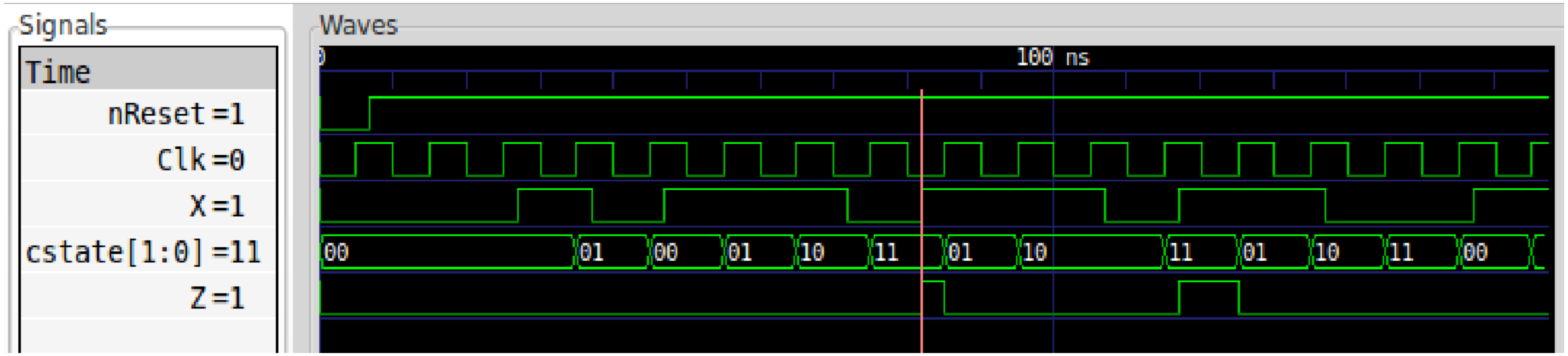


```
     //next state assignment
     always @ (*)
          case (cstate)
               A :  nstate <= (X) ? B : A;
               B :  nstate <= (X) ? C : A;
               C :  nstate <= (X) ? C : D;
               default : nstate <= (X) ? B : A;
          endcase

     //output assignment
     reg  Z;
     always @ (*)
          case (cstate)
               D : Z <= (X) ? 1'b1 : 0;
               default : Z <= 0;
          endcase

endmodule
```

Example 1:

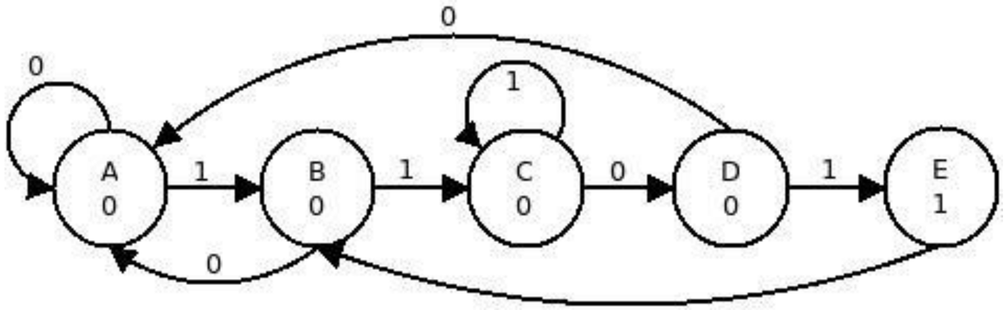
- **Mealy Model**

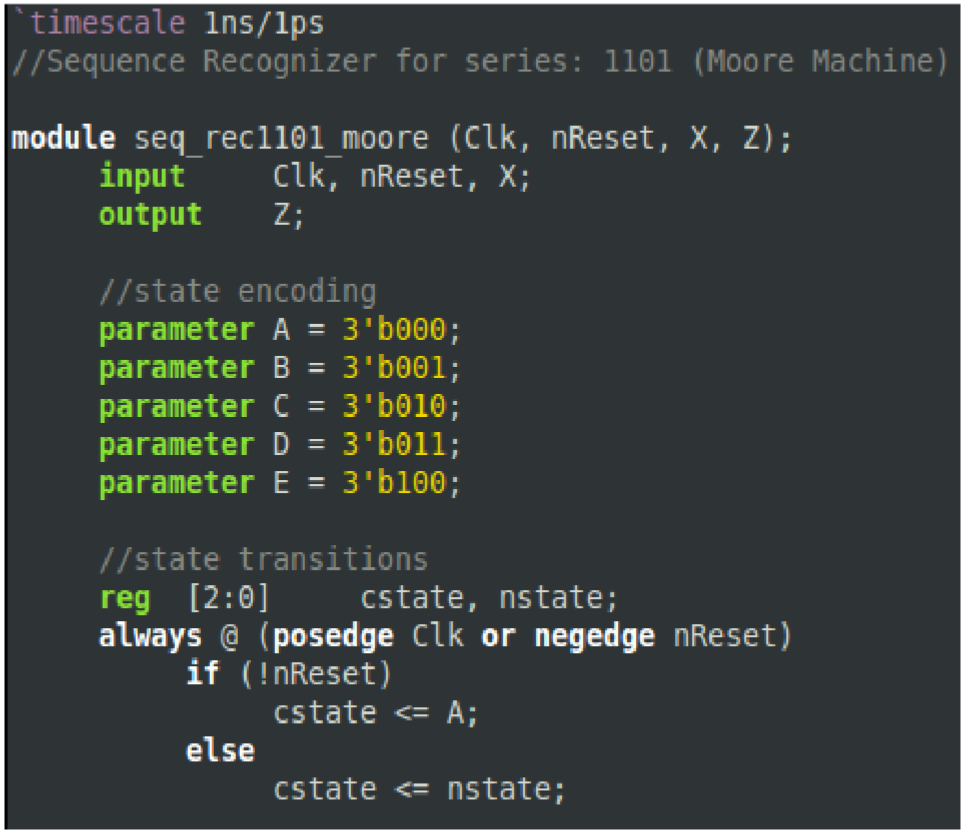




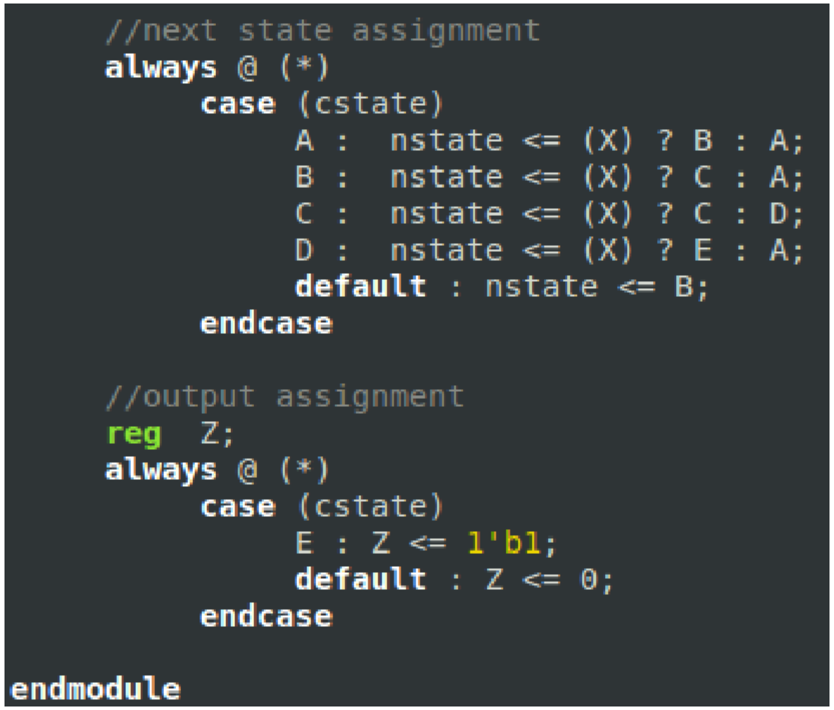
Example 1:

- **Moore Model**





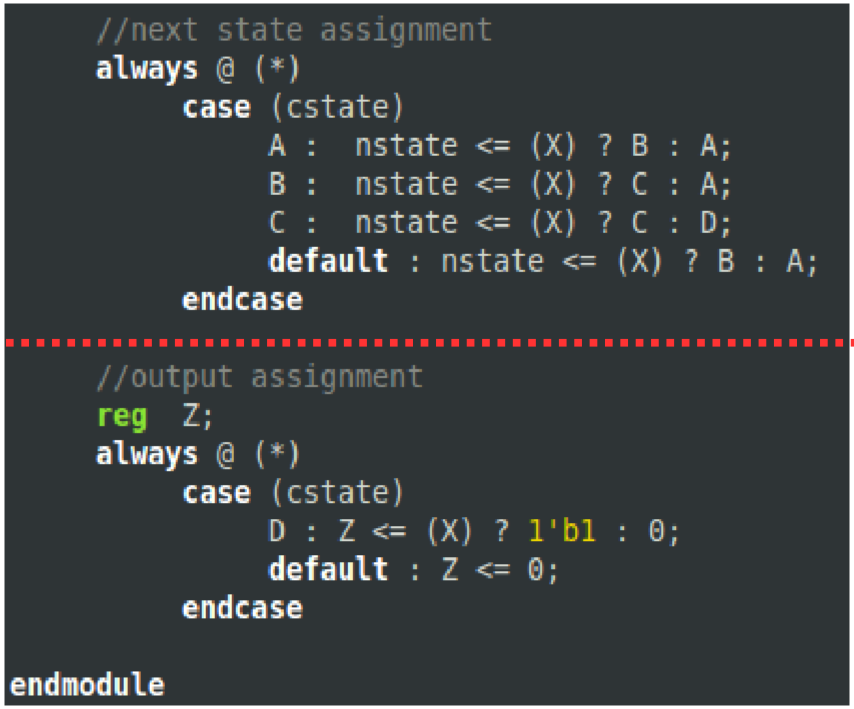
```
`timescale 1ns/1ps
//Sequence Recognizer for series: 1101 (Moore Machine)

module seq_rec1101_moore (Clk, nReset, X, Z);
     input     Clk, nReset, X;
     output    Z;

     //state encoding
     parameter A = 3'b000;
     parameter B = 3'b001;
     parameter C = 3'b010;
     parameter D = 3'b011;
     parameter E = 3'b100;

     //state transitions
     reg  [2:0]     cstate, nstate;
     always @ (posedge Clk or negedge nReset)
          if (!nReset)
               cstate <= A;
          else
               cstate <= nstate;
```



```
     //next state assignment
     always @ (*)
          case (cstate)
               A :  nstate <= (X) ? B : A;
               B :  nstate <= (X) ? C : A;
               C :  nstate <= (X) ? C : D;
               D :  nstate <= (X) ? E : A;
               default : nstate <= B;
          endcase

     //output assignment
     reg  Z;
     always @ (*)
          case (cstate)
               E : Z <= 1'b1;
               default : Z <= 0;
          endcase

endmodule
```

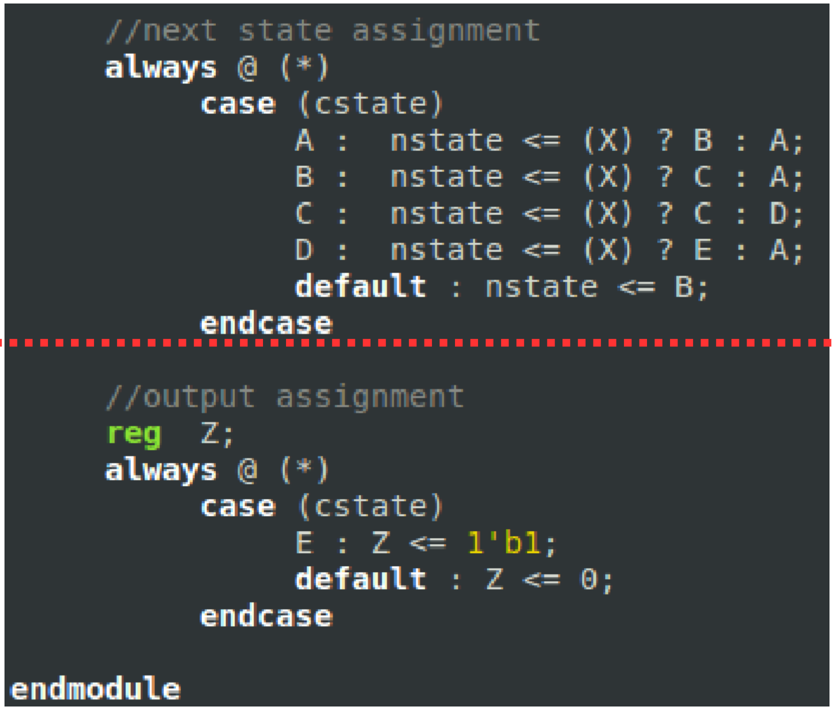
Example 1: Mealy vs. Moore Output

- **Mealy Model**



```
    //next state assignment
    always @ (*)
         case (cstate)
              A :  nstate <= (X) ? B : A;
              B :  nstate <= (X) ? C : A;
              C :  nstate <= (X) ? C : D;
              default : nstate <= (X) ? B : A;
         endcase

    //output assignment
    reg  Z;
    always @ (*)
         case (cstate)
              D : Z <= (X) ? 1'b1 : 0;
              default : Z <= 0;
         endcase

endmodule
```

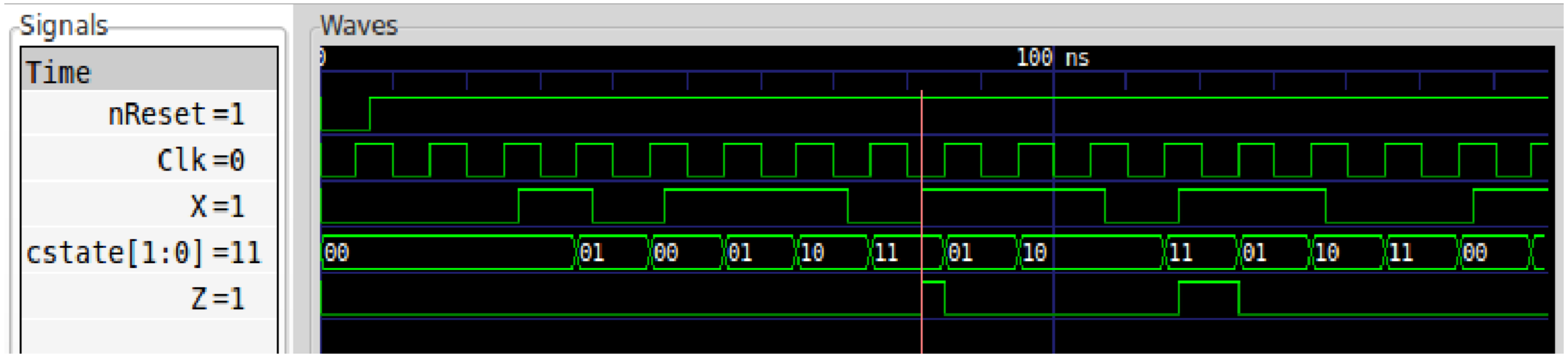
- **Moore Model**



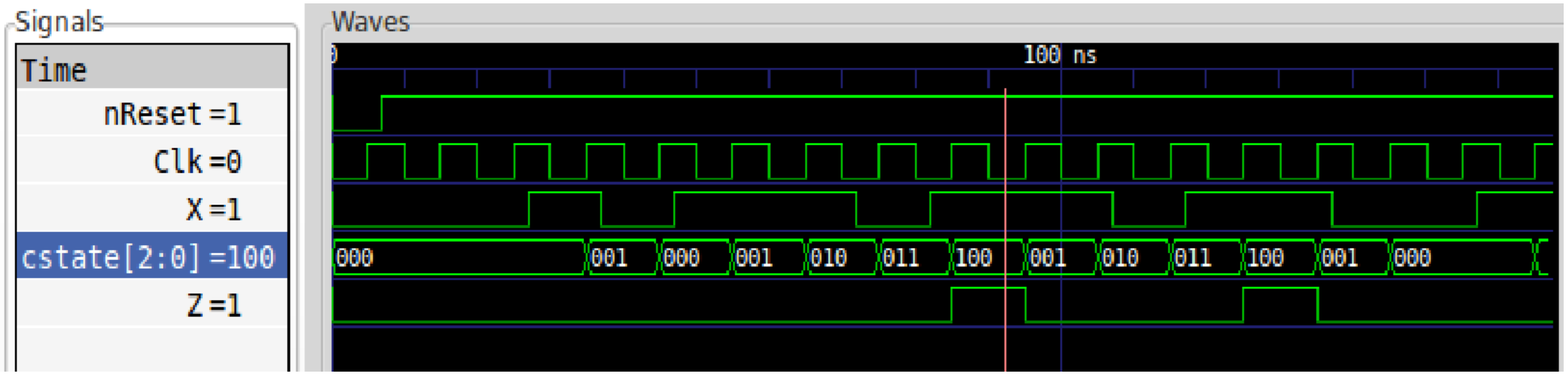
```
    //next state assignment
    always @ (*)
         case (cstate)
              A :  nstate <= (X) ? B : A;
              B :  nstate <= (X) ? C : A;
              C :  nstate <= (X) ? C : D;
              D :  nstate <= (X) ? E : A;
              default : nstate <= B;
         endcase

    //output assignment
    reg  Z;
    always @ (*)
         case (cstate)
              E : Z <= 1'b1;
              default : Z <= 0;
         endcase

endmodule
```

Example 1: Comparing Waveforms

Mealy Output



Moore Output



Partitioning Further

- As the sequential circuit grows in complexity, **multiple output assignments** can be partitioned into multiple separate ***always*** blocks.

Example 2: Vending Machine (Mealy)

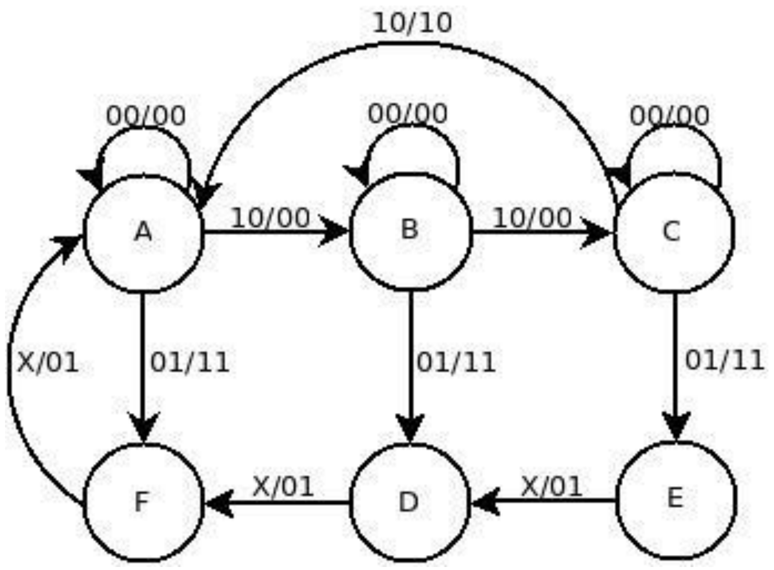
- Implement a **vending machine** which dispenses a 3-peso cost item.

- Vending machine has **2 inputs**: one for a 1-peso coin *(input **p1**)* and another for a 5-peso coin *(input **p5**)*
- It has **2 outputs**: dispensed item *(output **disp**)* and a 1-peso change *(output **change**)*

- Whenever a coin is inserted, the corresponding input is set to '1' for that clock cycle. Only one coin can be inserted every clock cycle and only from a single denomination.
  - Thus **{p1,p5}** can only be **“00”**, **“01”**, or **“10”**

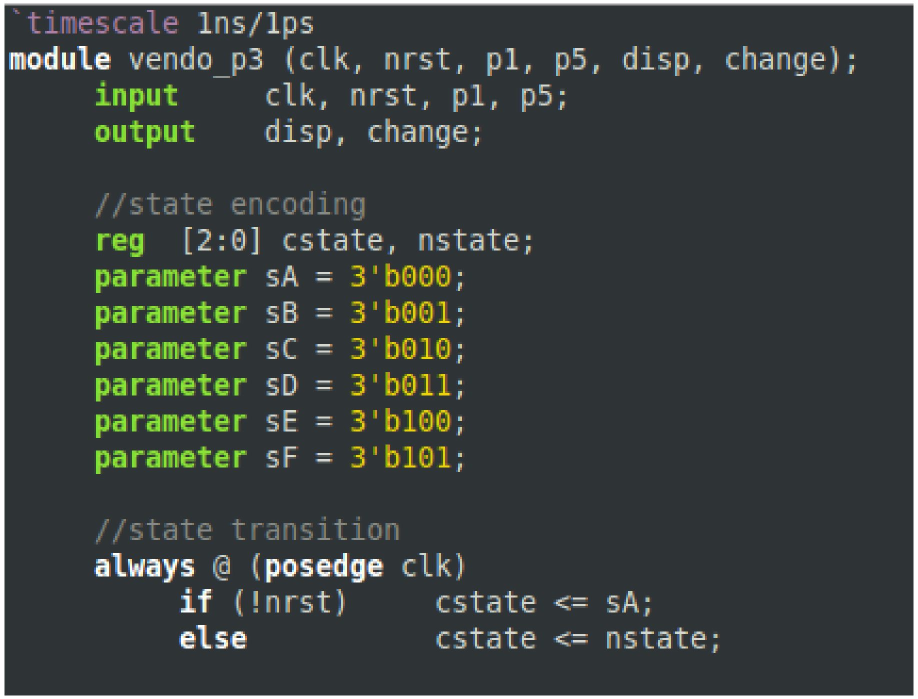
Example 2: Vending Machine (Mealy)

- **Possible scenarios:**
  - Insert p5, dispense and change, change
  - Insert p1, p5, dispense and change, change 2x
  - Insert p1 2x, p5, dispense and change, change 3x
  - Insert p1 3x, dispense
- **Assume I/O format:**
  - {p1, p5 / disp, change}

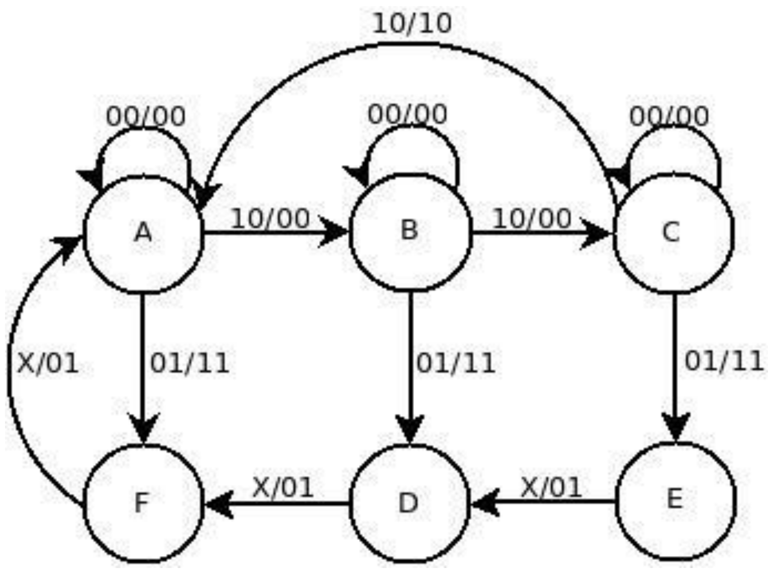


Example 2: Vending Machine (Mealy)

- **State Encoding and State Transition**

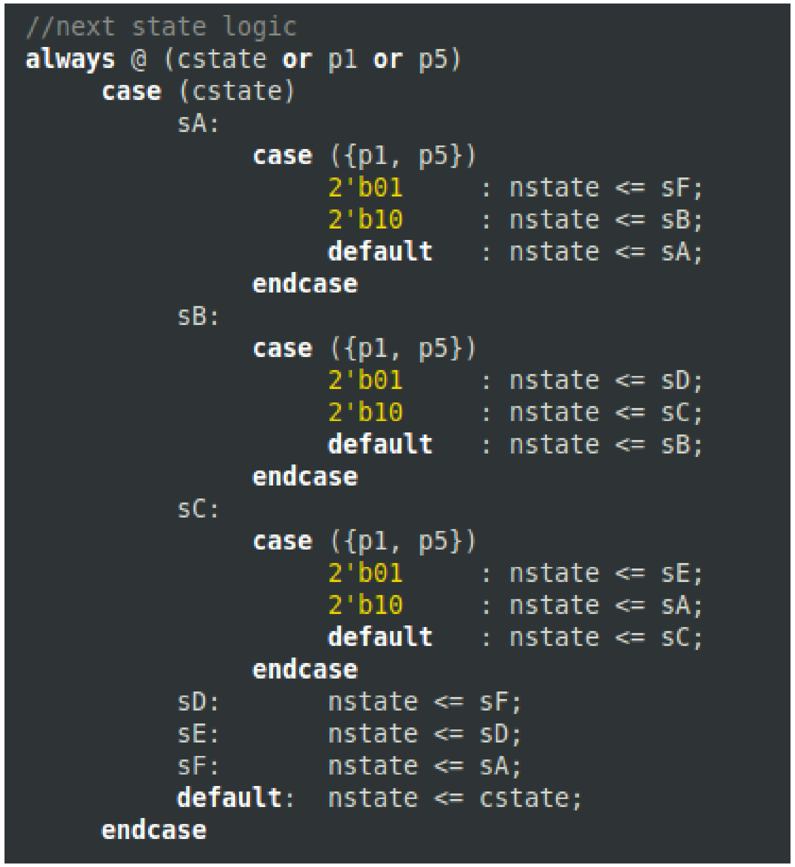


```
`timescale 1ns/1ps
module vendo_p3 (clk, nrst, p1, p5, disp, change);
     input     clk, nrst, p1, p5;
     output    disp, change;

     //state encoding
     reg  [2:0] cstate, nstate;
     parameter sA = 3'b000;
     parameter sB = 3'b001;
     parameter sC = 3'b010;
     parameter sD = 3'b011;
     parameter sE = 3'b100;
     parameter sF = 3'b101;

     //state transition
     always @ (posedge clk)
          if (!nrst)     cstate <= sA;
          else           cstate <= nstate;
```

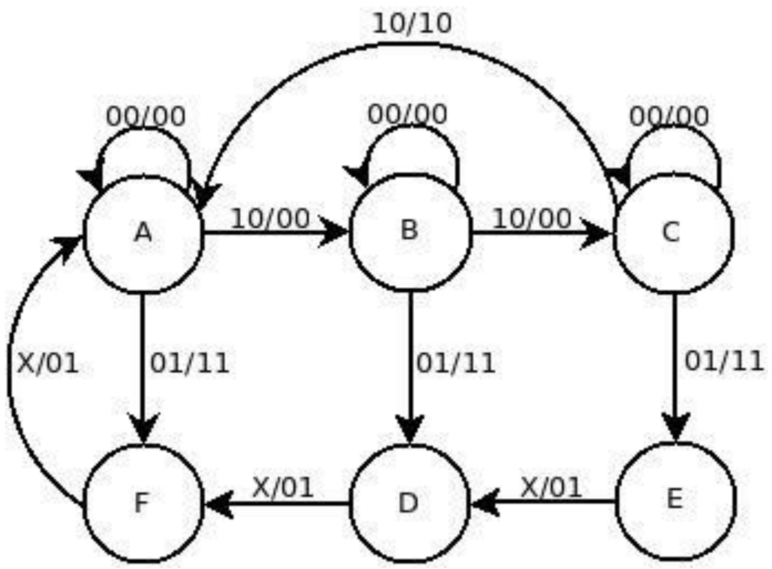


Example 2: Vending Machine (Mealy)

- **Next State Logic**

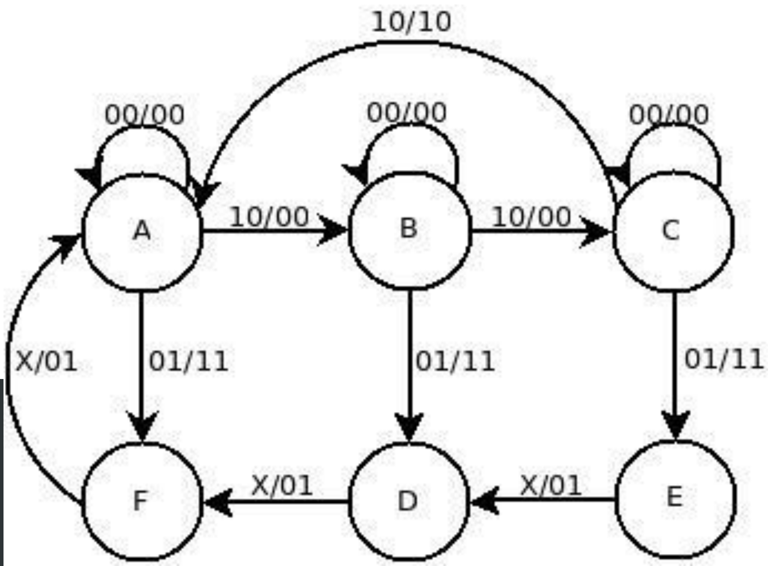


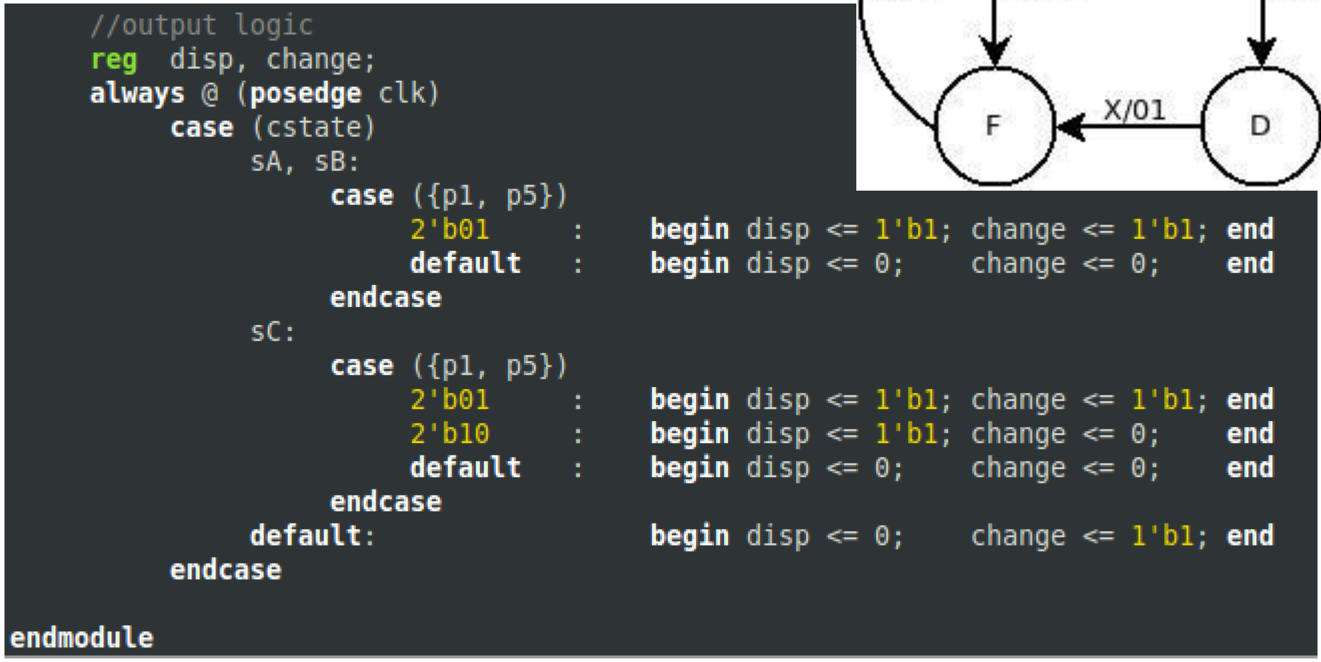
```
//next state logic
always @ (cstate or p1 or p5)
    case (cstate)
        sA:
            case ({p1, p5})
                2'b01      : nstate <= sF;
                2'b10      : nstate <= sB;
                default    : nstate <= sA;
            endcase
        sB:
            case ({p1, p5})
                2'b01      : nstate <= sD;
                2'b10      : nstate <= sC;
                default    : nstate <= sB;
            endcase
        sC:
            case ({p1, p5})
                2'b01      : nstate <= sE;
                2'b10      : nstate <= sA;
                default    : nstate <= sC;
            endcase
        sD:     nstate <= sF;
        sE:     nstate <= sD;
        sF:     nstate <= sA;
        default:  nstate <= cstate;
    endcase
```



Example 2: Vending Machine (Mealy)

- **Output Logic**

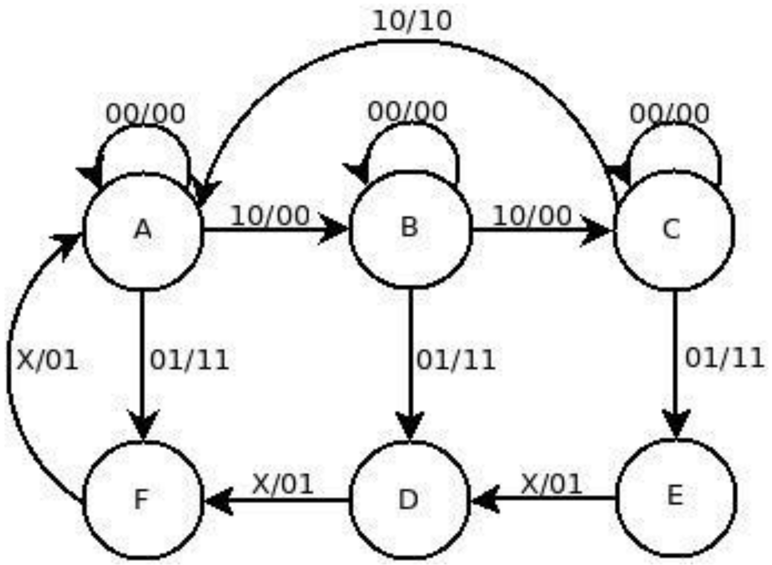


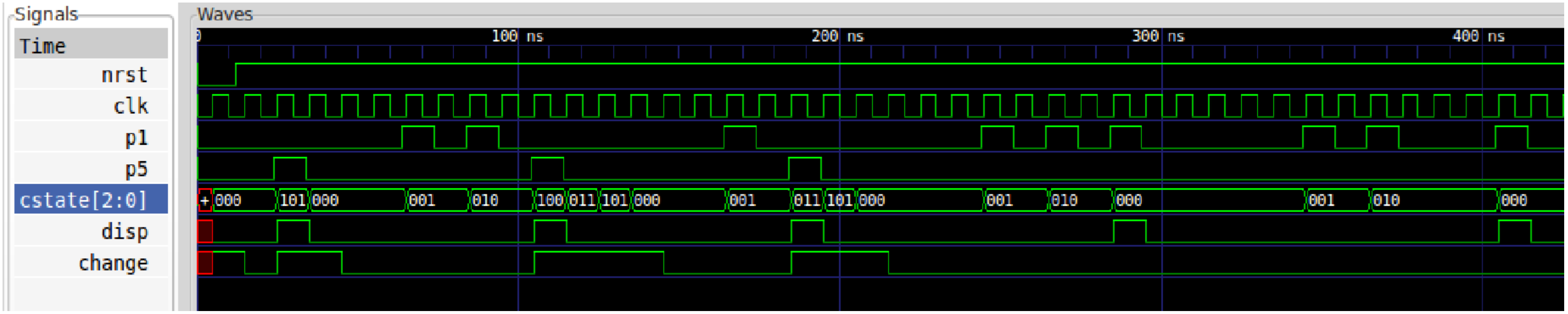


```
    //output logic
    reg  disp, change;
    always @ (posedge clk)
        case (cstate)
            sA, sB:
                case ({p1, p5})
                    2'b01      :    begin disp <= 1'b1; change <= 1'b1; end
                    default    :    begin disp <= 0;    change <= 0;    end
                endcase
            sC:
                case ({p1, p5})
                    2'b01      :    begin disp <= 1'b1; change <= 1'b1; end
                    2'b10      :    begin disp <= 1'b1; change <= 0;    end
                    default    :    begin disp <= 0;    change <= 0;    end
                endcase
            default:                begin disp <= 0;    change <= 1'b1; end
        endcase

endmodule
```

Example 2: Vending Machine (Mealy)

- **Possible scenarios:**
  - Insert p5, dispense and change, change
  - Insert p1, p5, dispense and change, change 2x
  - Insert p1 2x, p5, dispense and change, change 3x
  - Insert p1 3x, dispense



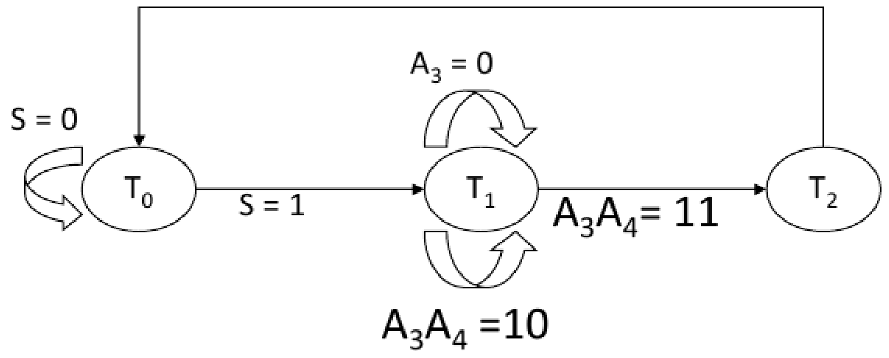


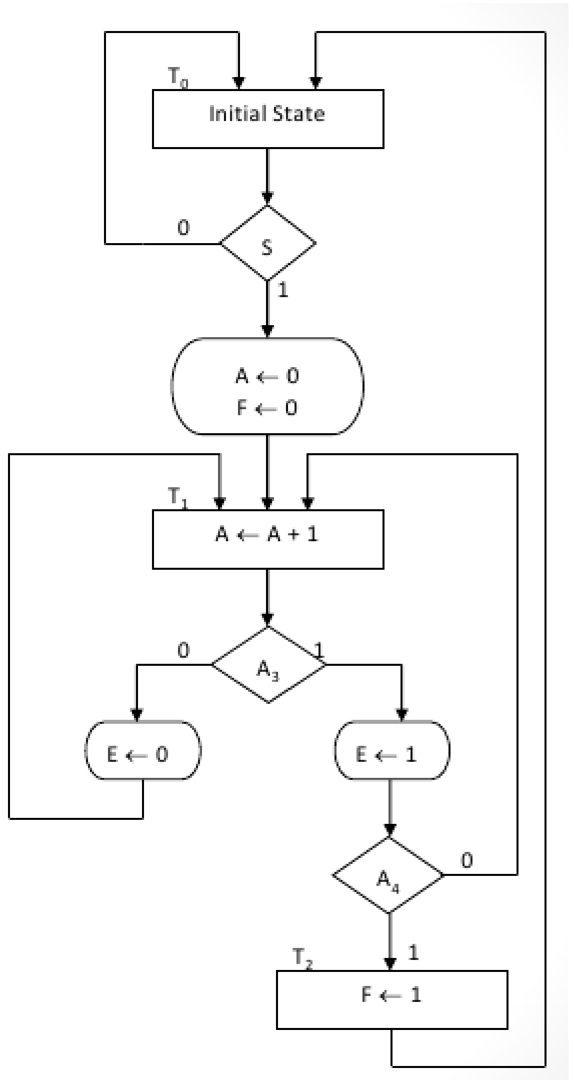
Control Unit as an FSM

Design a controller of a digital system with two flip-flops, E and F, and one 4-bit binary counter, A. The individual flip-flops in A are denoted by A4, A3, A2, and A1, with A4 holding the most significant bit of the count. A start signal S initiates the system operation by clearing the counter A and flip-flop F. The counter is then incremented by 1 starting from the next clock pulse and continues to increment until the operations stop. Counter bits A3 and A4 determine the sequence of operations:

- If A3 = 0, E is cleared to 0 and the count continues.
- If A3 = 1, E is set to 1; then if A4 = 0, the count continues, but if A4 = 1, F is set to 1 on the next clock pulse and the system stops counting.

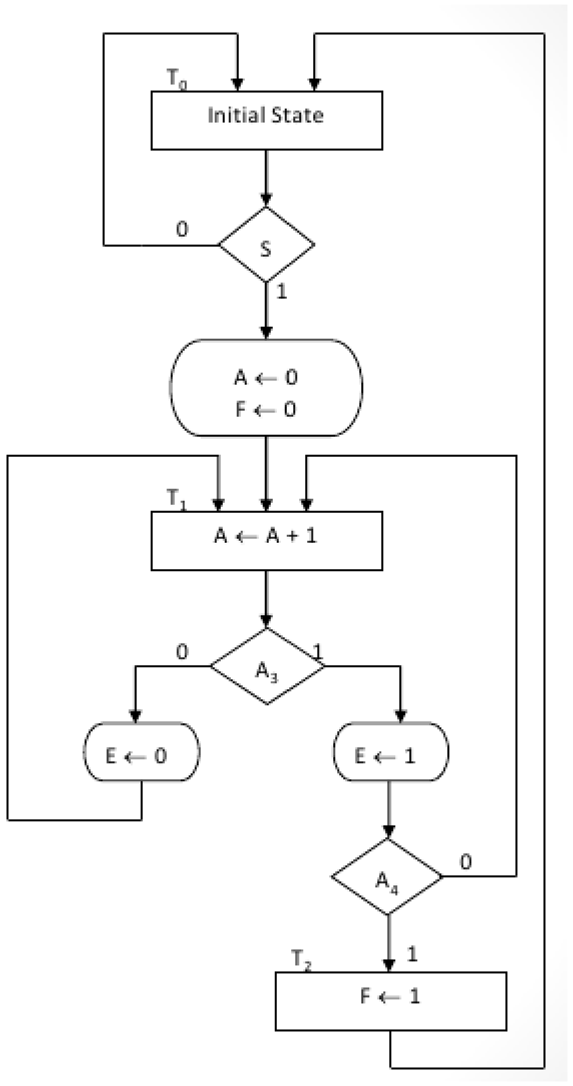
ASM Chart and State Diagram



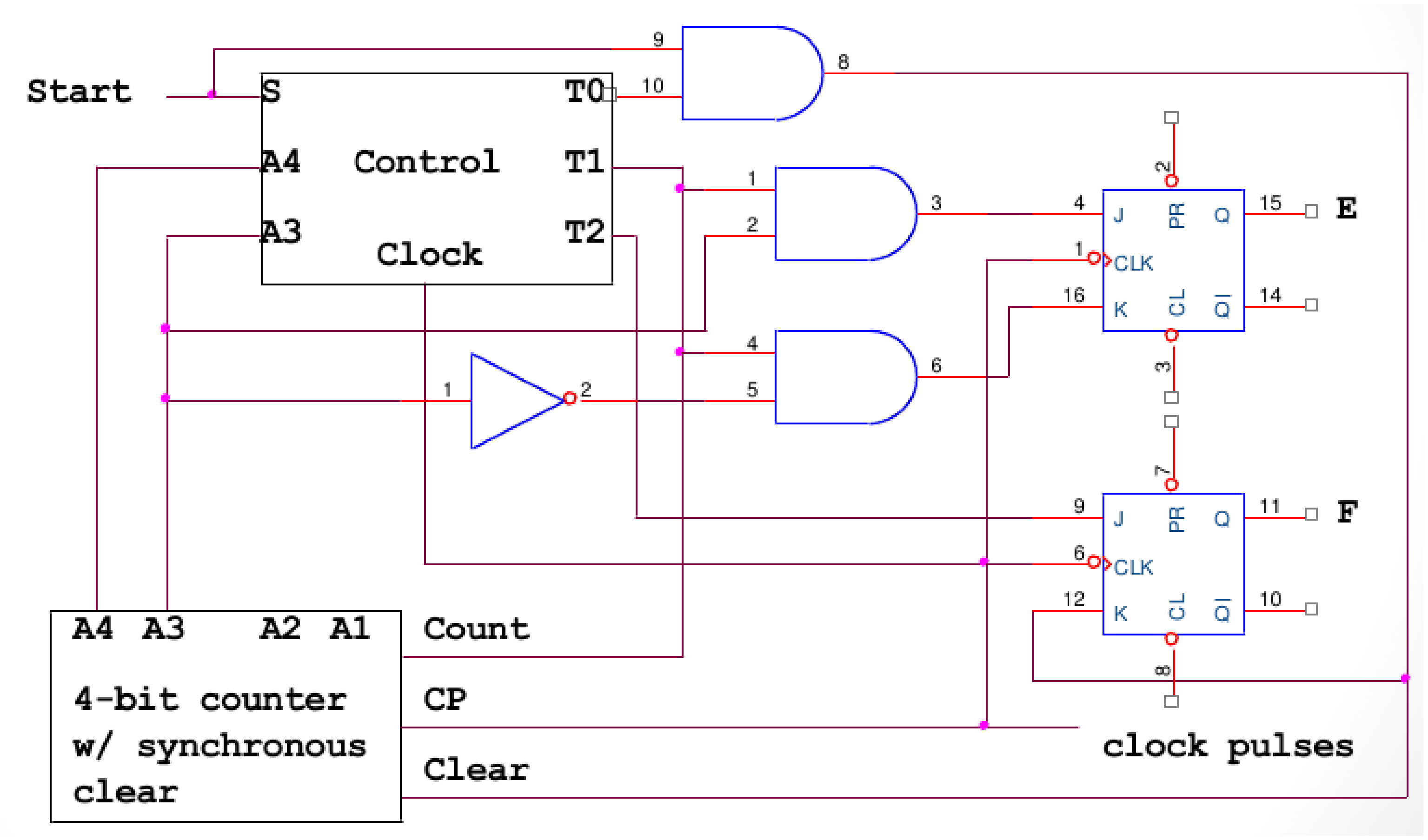


Sequence of Operations

Counter				Flip-flops		Conditions	State
A_4	A_3	A_2	A_1	E	F		
0	0	0	0	1	0	$A_3 = 0, A_4 = 0$	T_1
0	0	0	1	0	0		
0	0	1	0	0	0		
0	0	1	1	0	0		
0	1	0	0	0	0	$A_3 = 1, A_4 = 0$	
0	1	0	1	1	0		
0	1	1	0	1	0		
0	1	1	1	1	0		
1	0	0	0	1	0	$A_3 = 0, A_4 = 1$	
1	0	0	1	0	0		
1	0	1	0	0	0		
1	0	1	1	0	0		
1	1	0	0	0	0	$A_3 = 1, A_4 = 1$	
1	1	0	1	1	0		T_2
1	1	0	1	1	1		T_0



Control Unit as an FSM

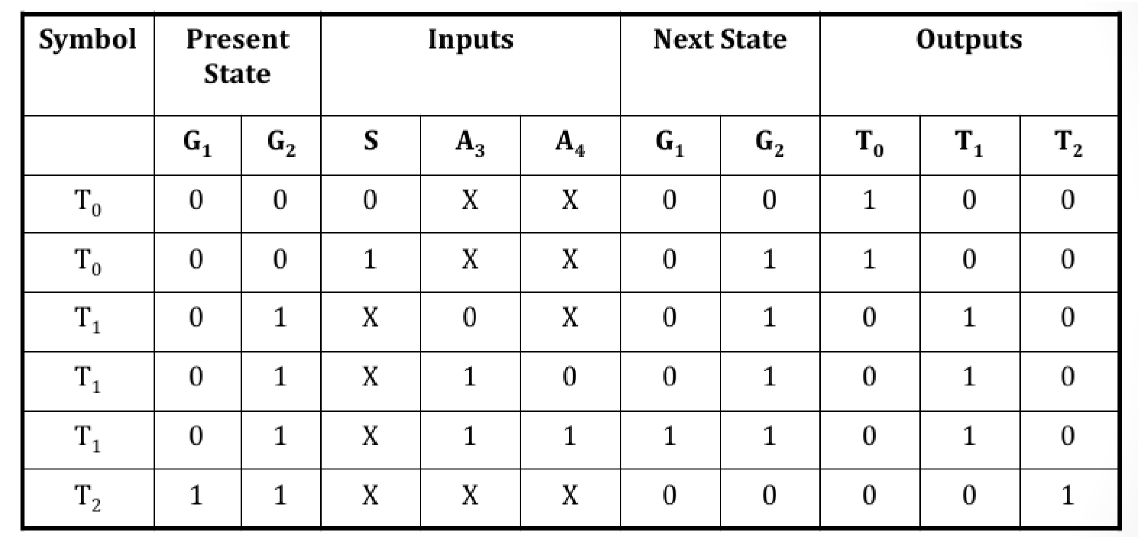


Control Implementation

- Hardwired Control
  - D flip-flop and a Decoder
  - One flip-flop per state

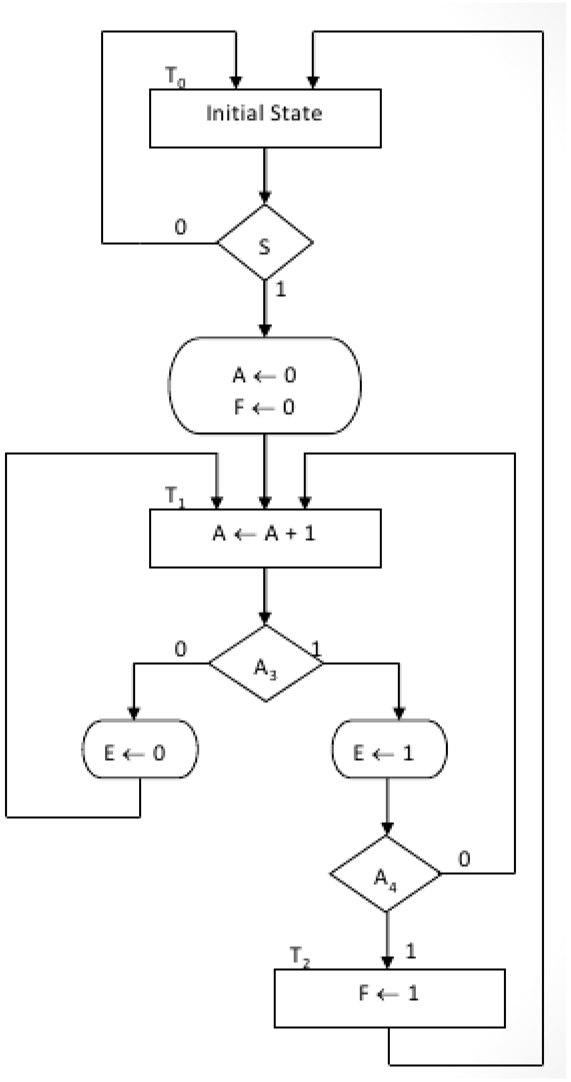
- HDL Implementation

D flip-flop and a Decoder



Symbol	Present State		Inputs			Next State		Outputs		
	G_1	G_2	S	A_3	A_4	G_1	G_2	T_0	T_1	T_2
T_0	0	0	0	X	X	0	0	1	0	0
T_0	0	0	1	X	X	0	1	1	0	0
T_1	0	1	X	0	X	0	1	0	1	0
T_1	0	1	X	1	0	0	1	0	1	0
T_1	0	1	X	1	1	1	1	0	1	0
T_2	1	1	X	X	X	0	0	0	0	1

Note: E and F are not included since they are external outputs (not part of the control logic block)



D flip-flop and a Decoder: K-map of D f/f inputs

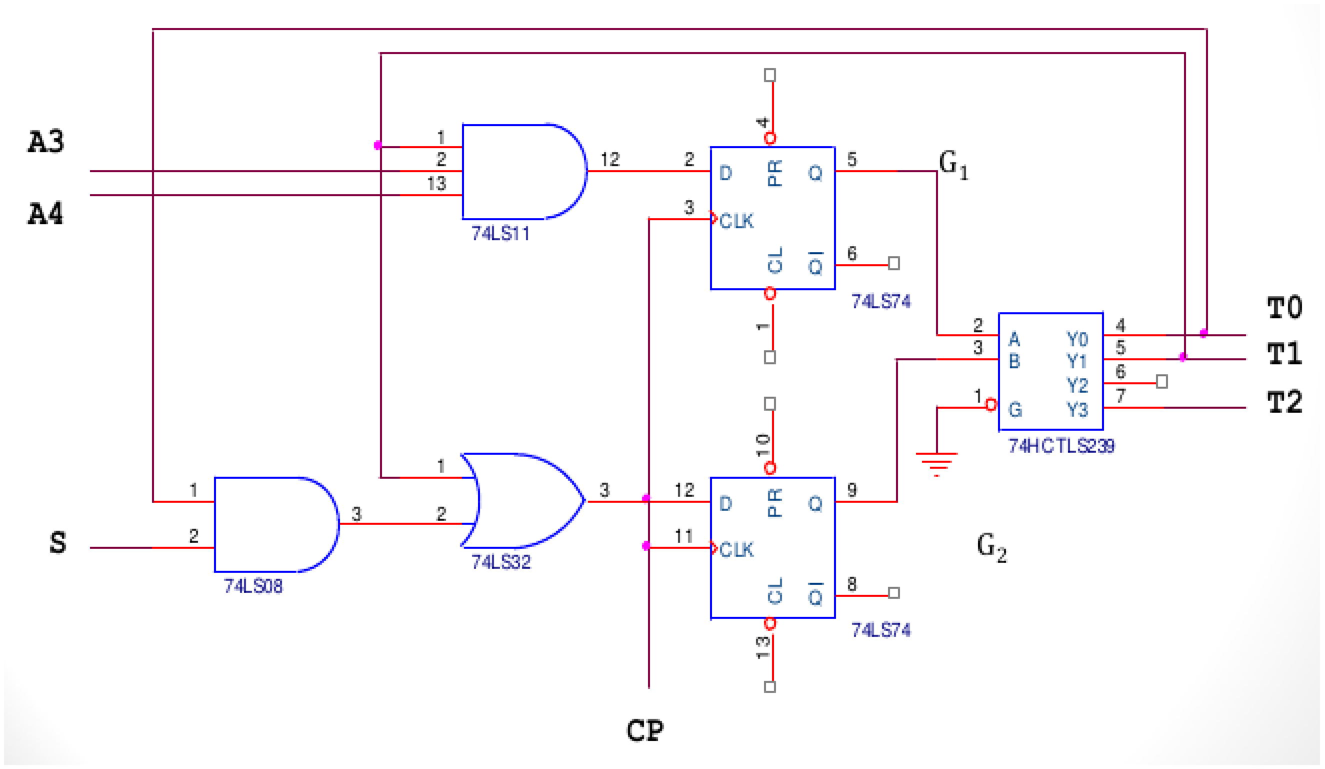
G_1G_2	A_3A_4 when S=0				A_3A_4 when S=1			
	00	**01**	**11**	**10**	**10**	**11**	**01**	**00**
00	0	0	0	0	0	0	0	0
01	0	0	1	0	0	1	0	0
11	0	0	0	0	0	0	0	0
10	x	x	x	x	x	x	x	x

$DG_1 = G_1'G_2A_3A_4 = A_3A_4T_1$

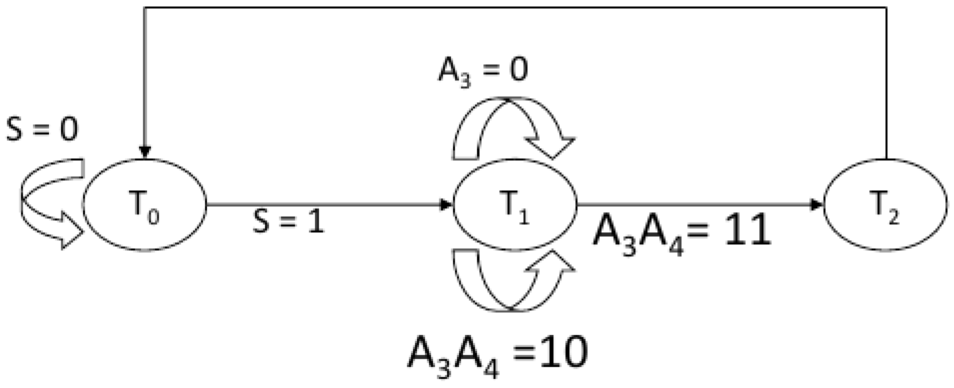
G_1G_2	A_3A_4 when S=0				A_3A_4 when S=1			
	00	**01**	**11**	**10**	**10**	**11**	**01**	**00**
00	0	0	0	0	1	1	1	1
01	1	1	1	1	1	1	1	1
11	0	0	0	0	0	0	0	0
10	x	x	x	x	x	x	x	x

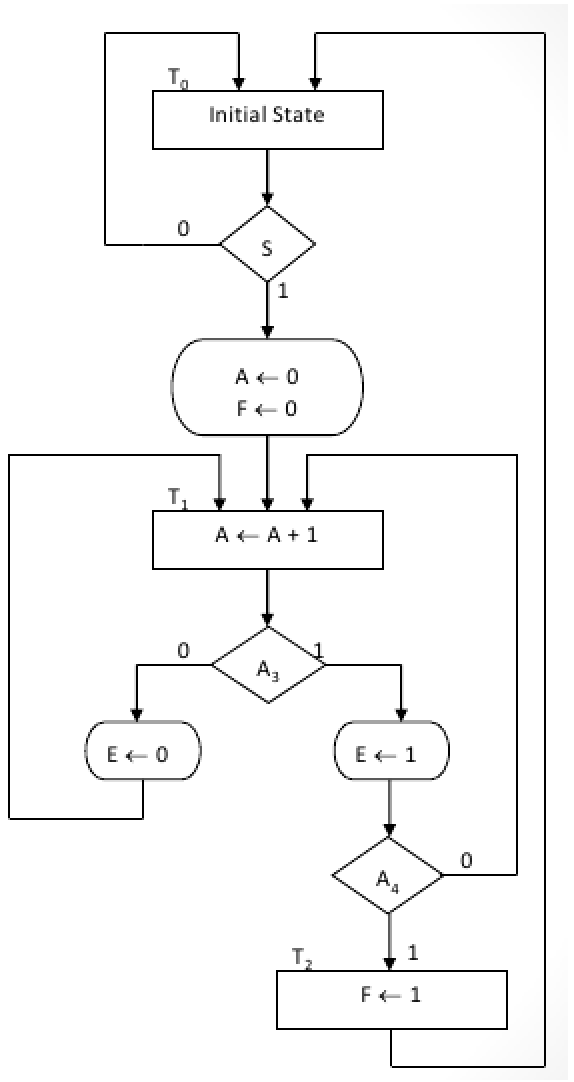
$DG_2 = G_1'G_2'S + G_1'G_2 = ST_0 + T_1$

D flip-flop and a Decoder: Logic diagram



One flip-flop per state



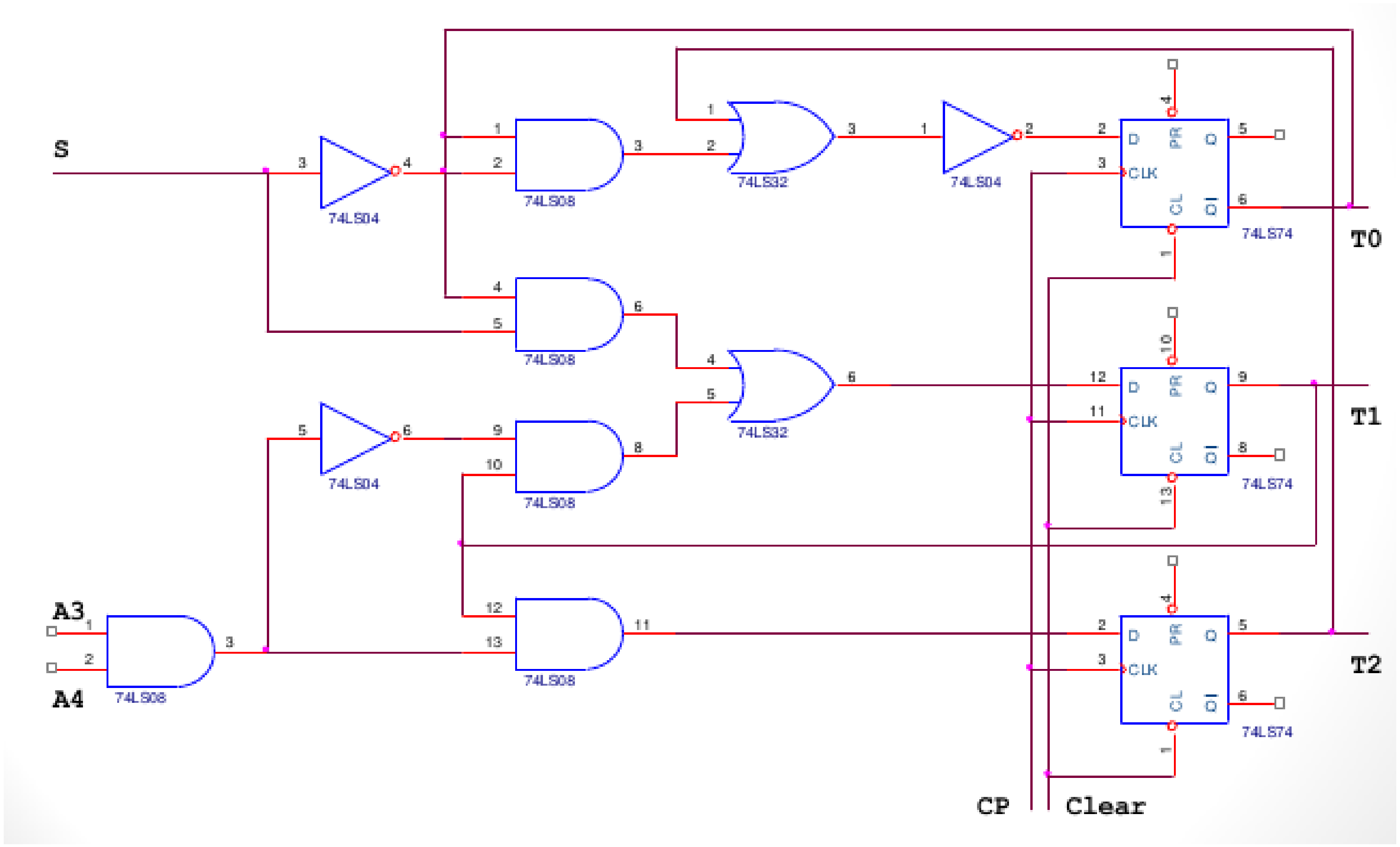


$DT_0 = S'T_0 + T_2$

$DT_1 = ST_0 + A_3'T_1 + A_3A_4'T_1 = ST_0 + (A_3A_4)'T_1$

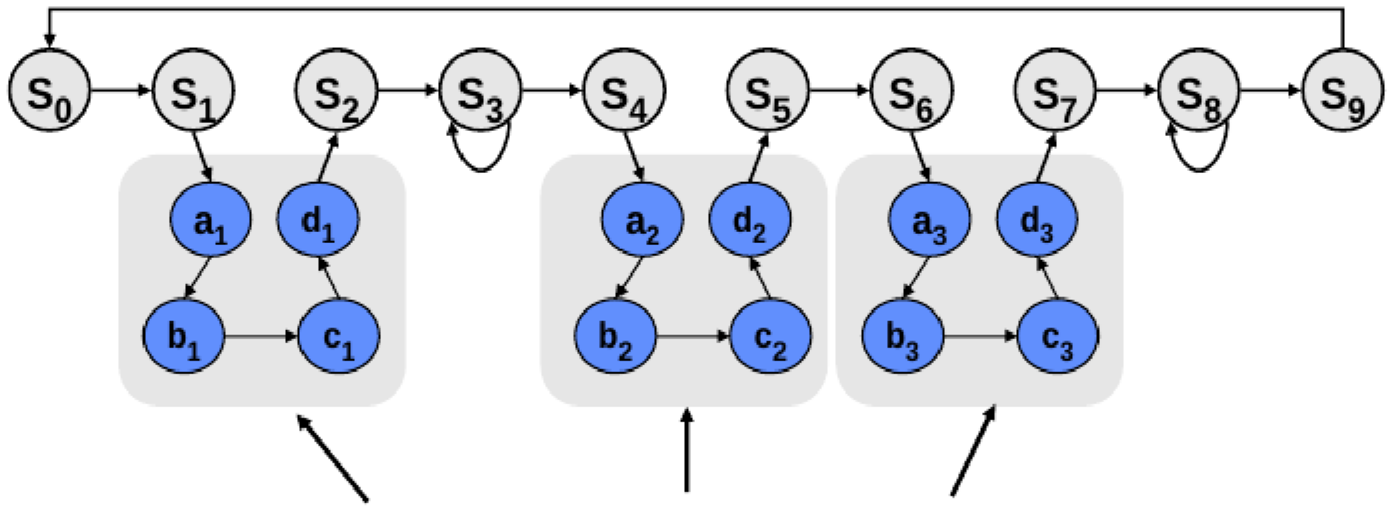
$DT_2 = A_3A_4T_1$

One flip-flop per state: Logic diagram



Towards FSM Modularity

- Consider the following abstract FSM:



- Suppose that each set of states ax...dx is a "sub-FSM" that produces exactly the same outputs.

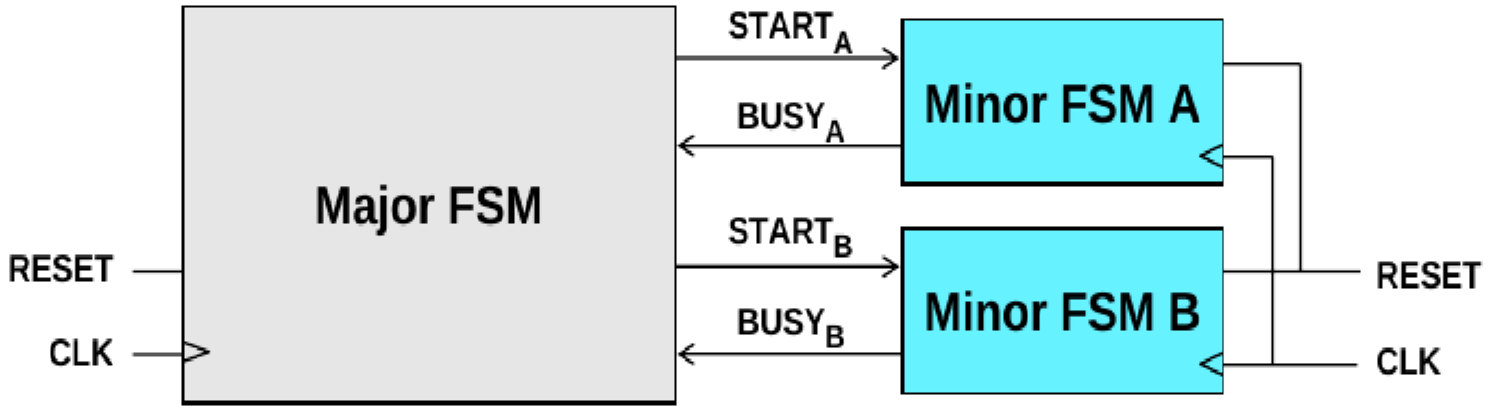
Towards FSM Modularity

- Can we simplify the FSM by removing equivalent states?
  - **No!** The outputs may be the same, but the next-state transitions are not.

- This situation closely resembles a **procedure call** or **function call** in software. How can we apply this concept to FSMs?

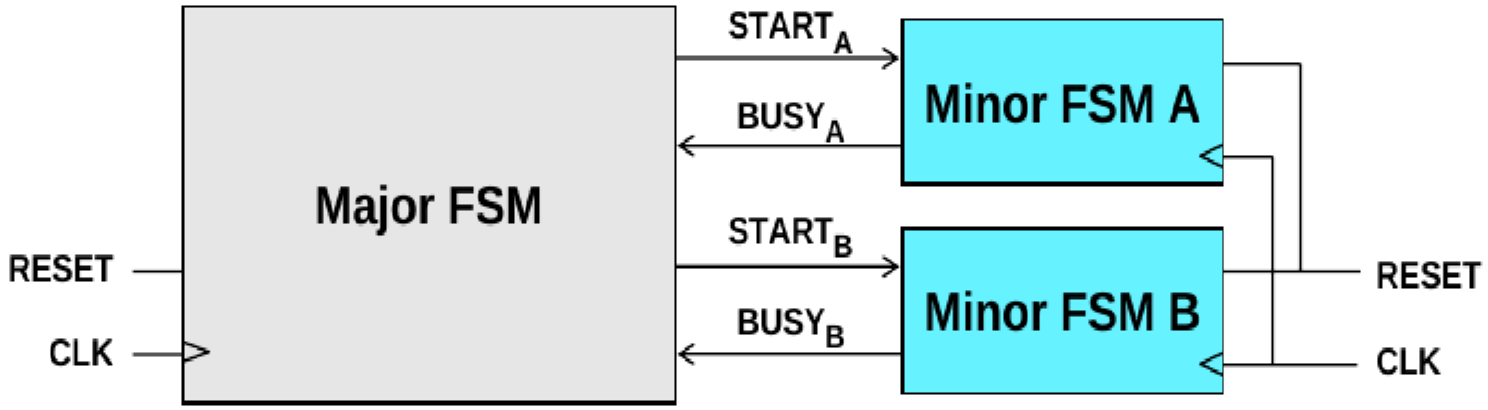
Major/Minor FSM Abstraction

- **Subtasks** are encapsulated in minor FSMs with common reset and clock.
- **Simple communication abstraction:**
  - **START:** tells the minor FSM to begin operation *(call)*
  - **BUSY:** tells the major FSM whether the minor is done *(return)*

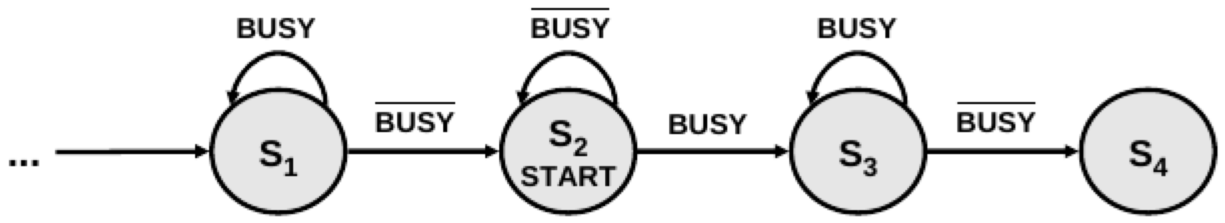


Major/Minor FSM Abstraction

- **The major/minor abstraction is great for:**
  - Modular designs *(always a good thing)*
  - Tasks that occur often but in different contexts
  - Tasks that require a variable/unknown period of time
  - Event-driven systems



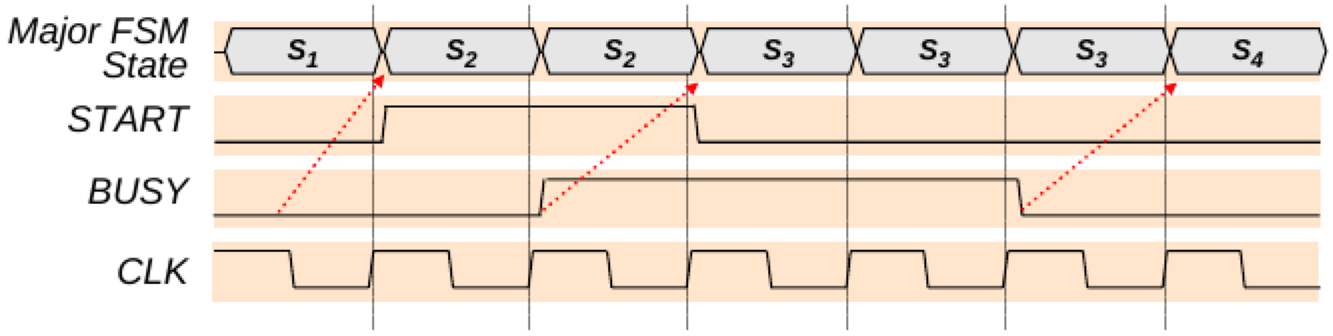
Inside the Major FSM



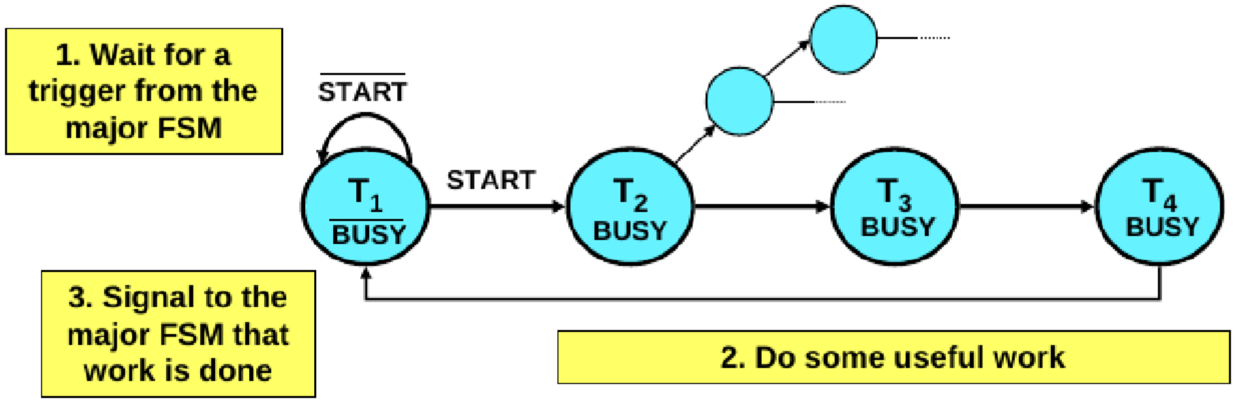
1. Wait until the minor FSM is ready

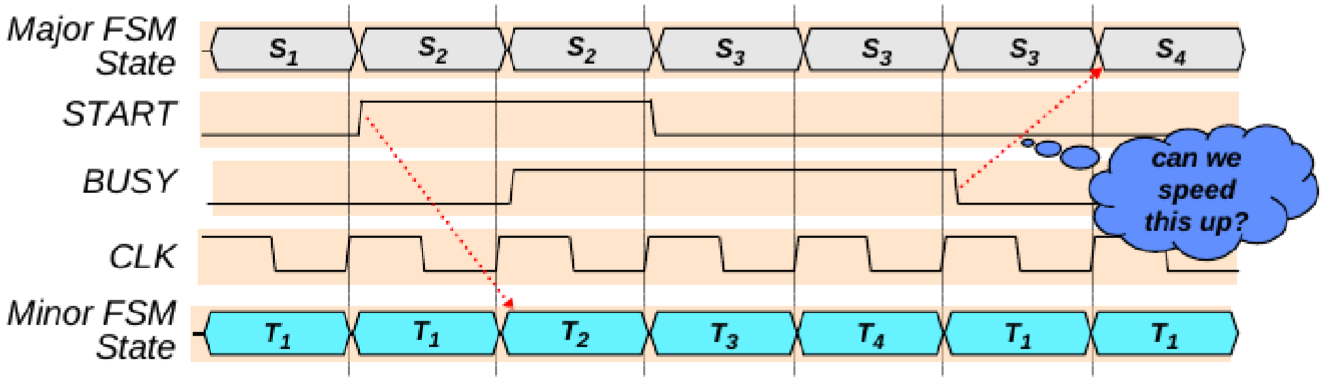
2. Trigger the minor FSM (and make sure it's started)

3. Wait until the minor FSM is done



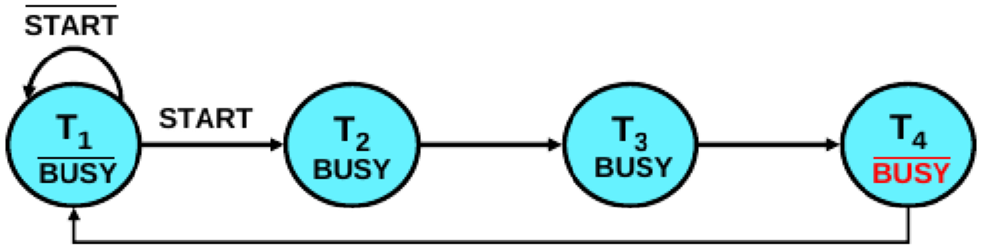
Inside the Minor FSM


1. Wait for a trigger from the major FSM
$\overline{\text{START}}$
START
T_1
$\overline{\text{BUSY}}$
T_2 BUSY
T_3 BUSY
T_4 BUSY
3. Signal to the major FSM that work is done
2. Do some useful work


Major FSM State
S_1 S_2 S_2 S_3 S_3 S_3 S_4
START
BUSY
CLK
Minor FSM State
T_1 T_1 T_2 T_3 T_4 T_1 T_1
can we speed this up?

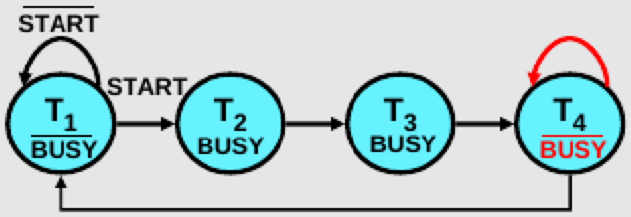
Optimizing the Minor FSM

Good idea: de-assert BUSY one cycle early



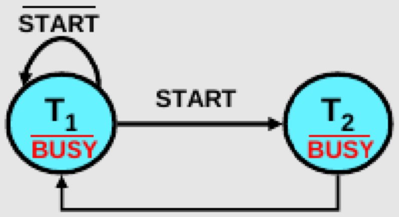
Bad idea #1:

T_4 may not immediately return to T_1

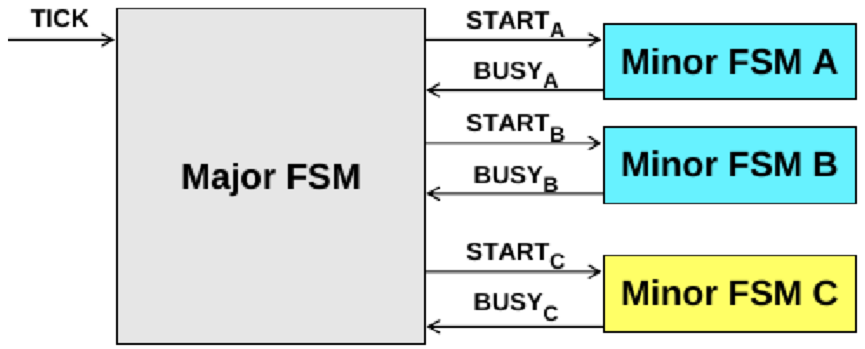


Bad idea #2:

BUSY never asserts!

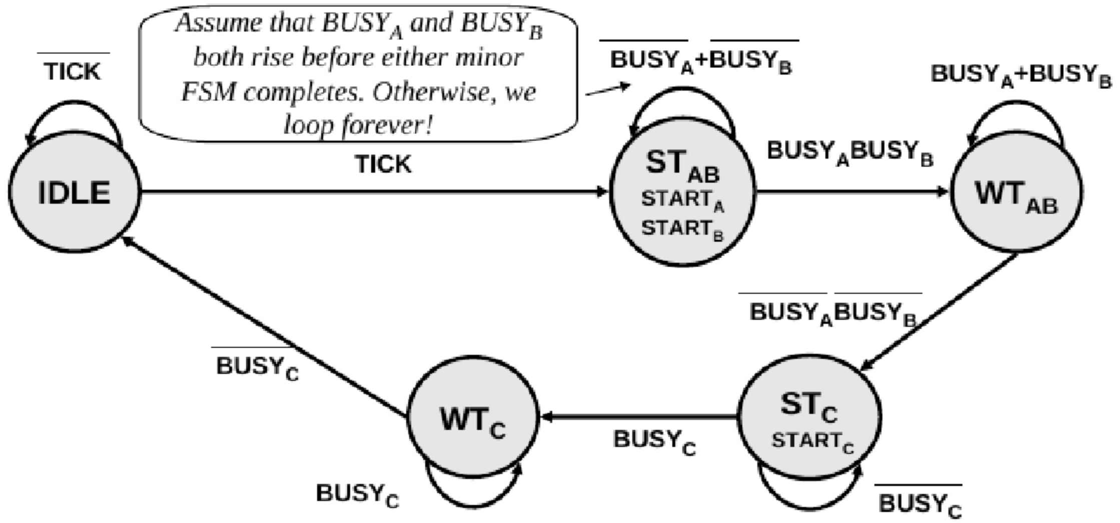


A Four-FSM Example

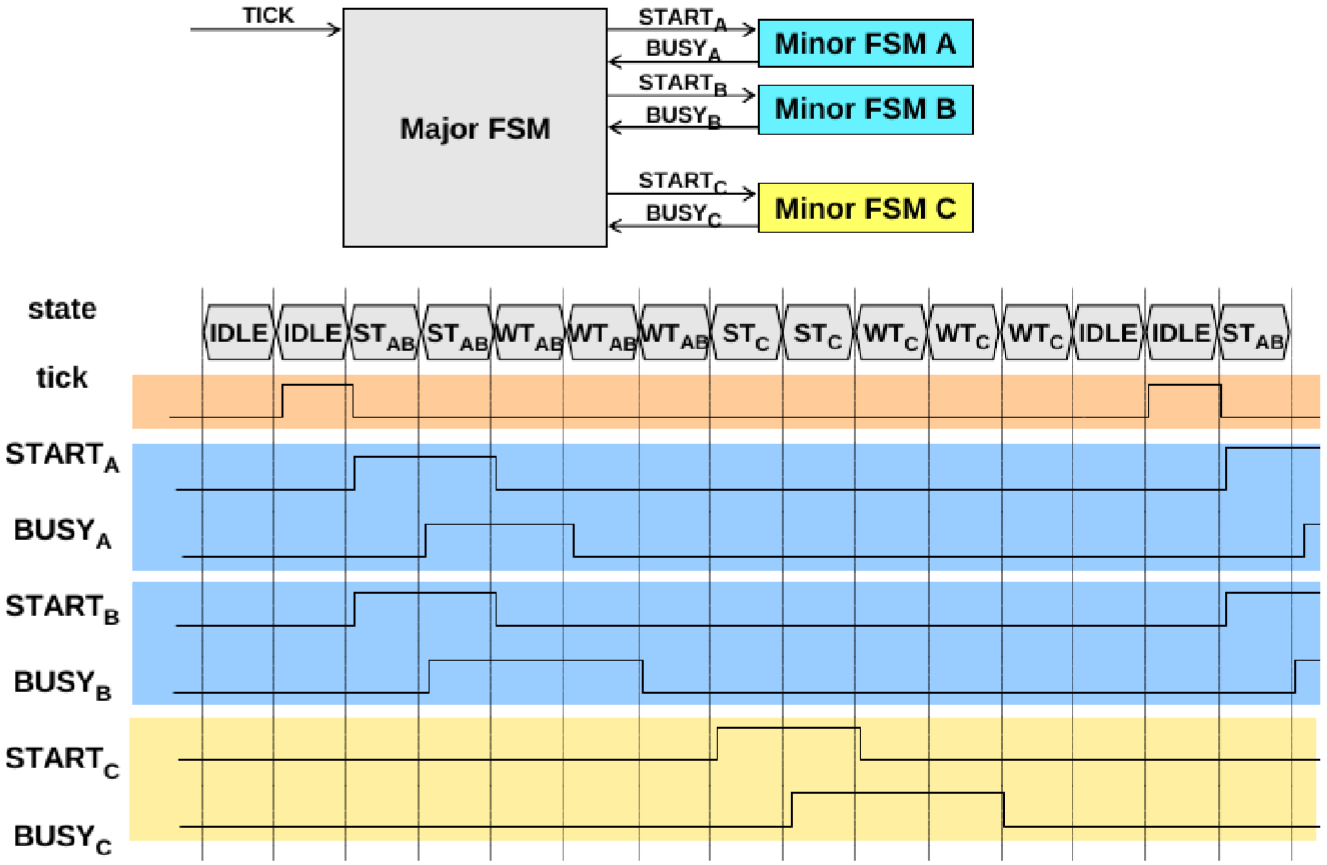


Operating Scenario:

- Major FSM is triggered by TICK
- Minors A and B are started simultaneously
- Minor C is started once both A and B complete
- TICKs arriving before the completion of C are ignored


Assume that BUSY_A and BUSY_B both rise before either minor FSM completes. Otherwise, we loop forever!
$\overline{\text{TICK}}$
IDLE
TICK
$\overline{\text{BUSY}_A}+\overline{\text{BUSY}_B}$
ST_AB START_A START_B
BUSY_A BUSY_B
BUSY_A+BUSY_B
WT_AB
$\overline{\text{BUSY}_A}\,\overline{\text{BUSY}_B}$
ST_C START_C
$\overline{\text{BUSY}_C}$
BUSY_C
WT_C
BUSY_C
$\overline{\text{BUSY}_C}$

Four-FSM Sample Waveform



End of Unit 7