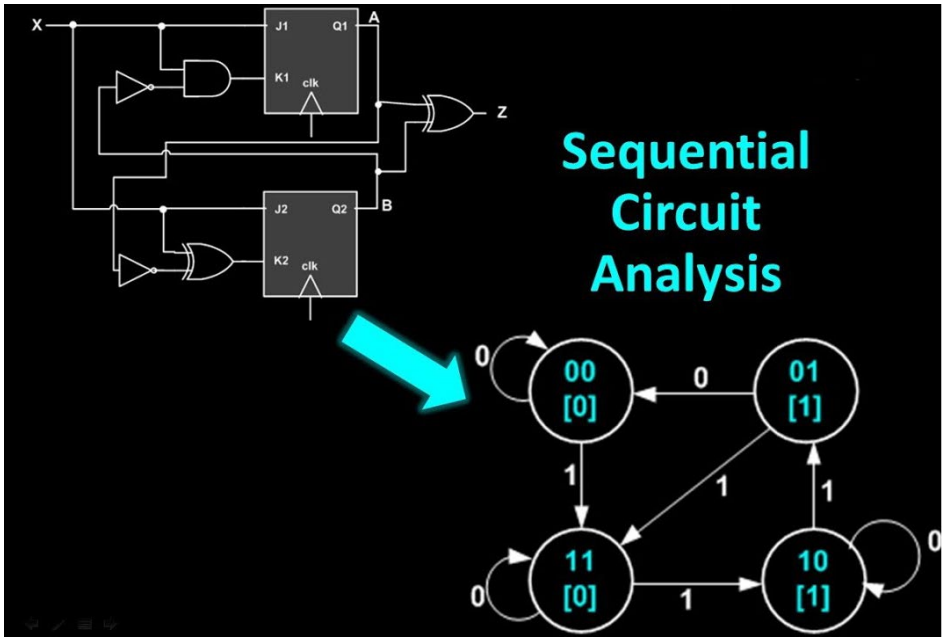
EE2000 Logic Circuit Design

Lecture 9 – Sequential Logic Circuit Design



Exercise (Sequence Detector)

Design a Mealy machine to detect the sequence “111” (Overlapping)

In other words,

Design a system with one input x and one output z such that z = 1 if x has been 1 for at least three consecutive clock times.

x	0	1	1	0	1	1	1	0	1	1	1	1	1	0
z	0	0	0	0	0	0	1	0	0	0	1	1	1	0

Exercise (Sequence Detector)

x	0	1	1	0	1	1	1	0	1	1	1	1	1	0
z	0	0	0	0	0	0	1	0	0	0	1	1	1	0

STEP 1: Determine what needs to be stored in memory and how to store them.

A: input is '0' *if next input is 0, remains at A else B

B: one '1' is detected *if next input is 0, back to A else C

C: two '1's are detected

* if next input is 0, back to A and output '0', else remains at C and output '1'

Exercise (Sequence Detector)

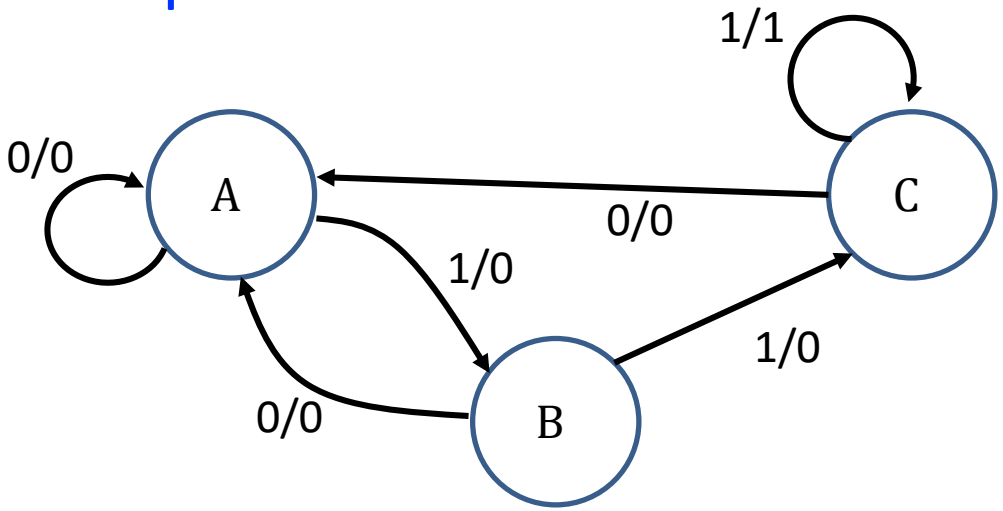
STEP 2: Work out the State Diagram

A: Input is '0' *if next input is 0, remains at A else B

B: one '1' is detected *if next input is 0, back to A else C

C: two '1's are detected

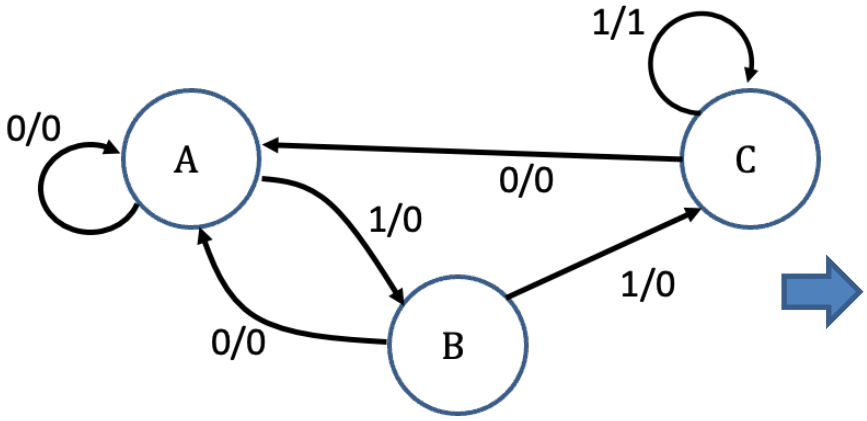
* if next input is 0, back to A and output '0', else remains at C and output '1'



Exercise (Sequence Detector)

STEP 3: Work out the analysis table with assigned FFs

3 states → 2 FFs (We use D-FFs in this example)



Present State (Q_1Q_2)	Input X	Next stage Q_1^* Q_2^*	Output Z
A (0 0)	0		
A (0 0)	1		
B (0 1)	0		
B (0 1)	1		
(1 0)	X		
C (1 1)	0		
C (1 1)	1		

Assign State A:

$Q_1 \rightarrow 0$ and $Q_2 \rightarrow 0$ etc

Exercise (Sequence Detector)

STEP 4: Work out D_1 and D_2

Present State (Q_1Q_2)	Input X	Next State Q_1^*	Next State Q_2^*	Output Z
A (0 0)	0	0	0	0
A (0 0)	1	0	1	0
B (0 1)	0	0	0	0
B (0 1)	1	1	1	0
(1 0)	x	x	x	x
C (1 1)	0	0	0	0
C (1 1)	1	1	1	1

Q_1^*

x \ Q_1Q_2	00	01	11	10
0	0	0	0	x
1	0	1	1	x

$$D_1 = Q_1^* = xQ_2$$

Q_2^*

x \ Q_1Q_2	00	01	11	10
0	0	0	0	x
1	1	1	1	x

$$D_2 = Q_2^* = x$$

Exercise (Sequence Detector)

STEP 5: Work out z

Present State (Q_1Q_2)	Input X	Next State Q_1^*	Q_2^*	Output Z
A (0 0)	0	0	0	0
A (0 0)	1	0	1	0
B (0 1)	0	0	0	0
B (0 1)	1	1	1	0
(1 0)	x	x	x	x
C (1 1)	0	0	0	0
C (1 1)	1	1	1	1

z

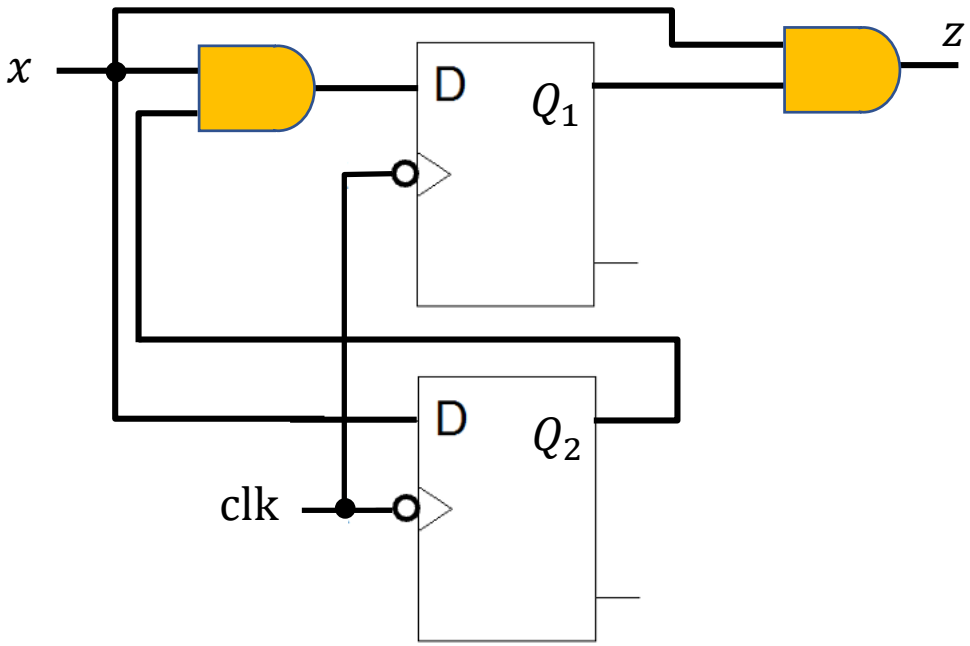
x \ Q_1Q_2	00	01	11	10
0	0	0	0	x
1	0	0	1	x

$$z = xQ_1$$

Exercise (Sequence Detector)

STEP 6: Draw the sequential logic circuits

$D_1 = xQ_2$ $D_2 = x$ $z = xQ_1$



Exercise (Sequence Detector)

Use T FFs to design a Mealy machine to detect the sequence “111” (Overlapping)

Present State (Q_1Q_2)	Input *X*	Next State Q_1^*	Q_2^*	Flip-Flops T_1	T_2	Output *Z*
A (0 0)	0	0	0	0	0	0
A (0 0)	1	0	1	0	1	0
B (0 1)	0	0	0	0	1	0
B (0 1)	1	1	1	1	0	0
(1 0)	x	x	x	x	x	x
C (1 1)	0	0	0	1	1	0
C (1 1)	1	1	1	0	0	1

T	Q_{t+1}	$\overline{Q_{t+1}}$	State
0	Q_t	$\overline{Q_t}$	Hold
1	$\overline{Q_t}$	Q_t	Toggle

Exercise (Sequence Detector)

Present State (Q_1Q_2)	Input X	Flip-Flops T_1	T_2
A (0 0)	0	0	0
A (0 0)	1	0	1
B (0 1)	0	0	1
B (0 1)	1	1	0
(1 0)	x	x	x
C (1 1)	0	1	1
C (1 1)	1	0	0

T_1

x \ Q_1Q_2	00	01	11	10
0	0	0	1	x
1	0	1	0	x

$$T_1 = xQ_1'Q_2 + x'Q_1$$

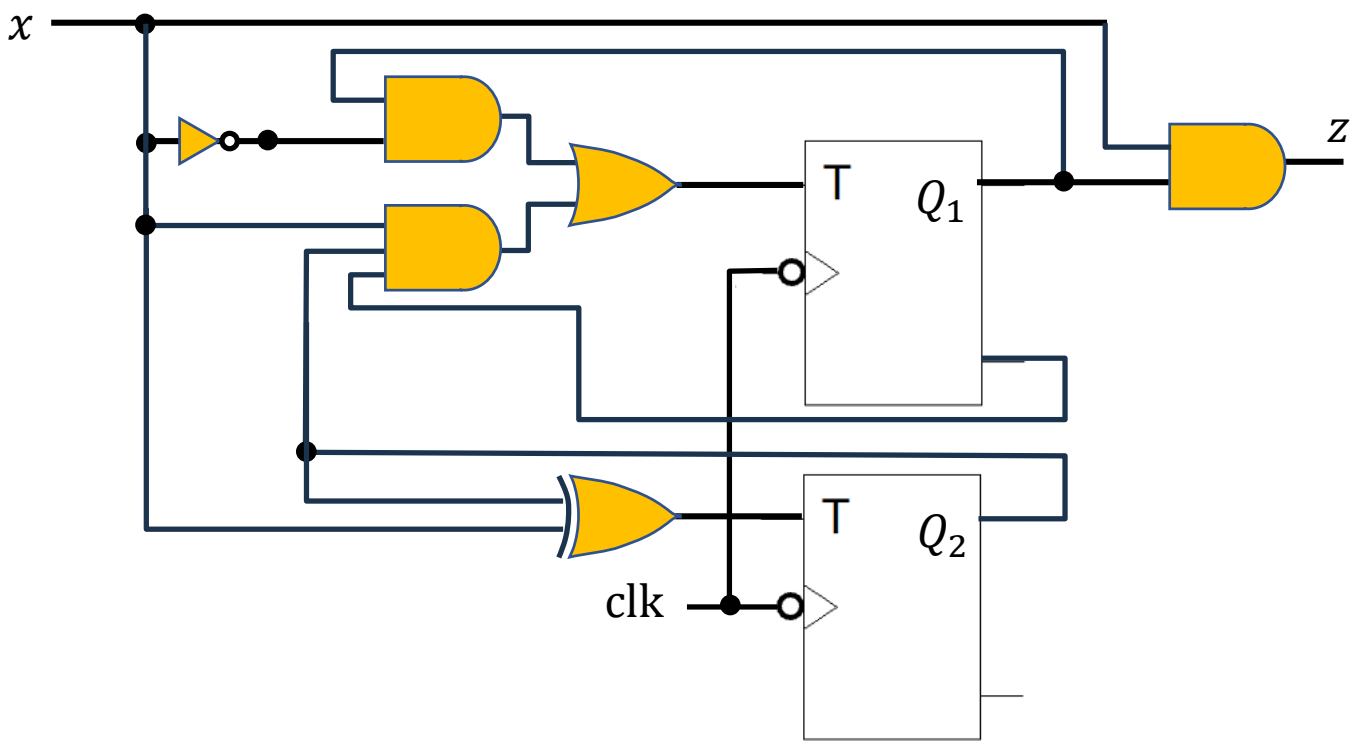
T_2

x \ Q_1Q_2	00	01	11	10
0	0	1	1	x
1	1	0	0	x

$$T_2 = xQ_2' + x'Q_2$$

Exercise (Sequence Detector)

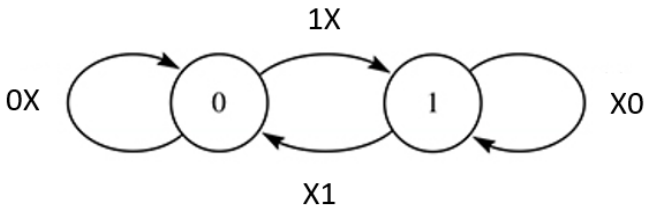
$$T_1 = xQ_1'Q_2 + x'Q_1 \qquad T_2 = xQ_2' + x'Q_2 = x \oplus Q_2 \qquad z = xQ_1$$



Exercise (Sequence Detector)

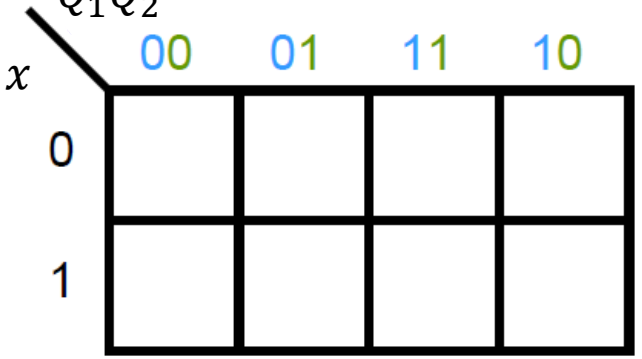
Use JK FFs to design a Mealy machine to detect the sequence “111” (Overlapping)

Present State (Q_1Q_2)	Input X	Next State Q_1^*	Q_2^*	Flip-Flops J_1	K_1	J_2	K_2	Output Z
A (0 0)	0	0	0					0
A (0 0)	1	0	1					0
B (0 1)	0	0	0					0
B (0 1)	1	1	1					0
(1 0)	x	x	x					x
C (1 1)	0	0	0					0
C (1 1)	1	1	1					1



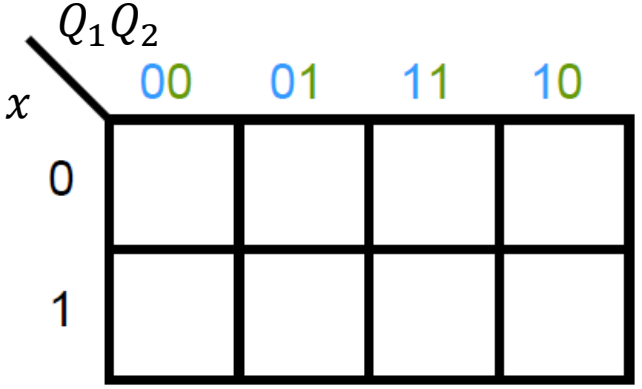
Present State (Q_1Q_2)	Input X	Flip-Flops J_1	K_1	J_2	K_2
A (0 0)	0				
A (0 0)	1				
B (0 1)	0				
B (0 1)	1				
(1 0)	x				
C (1 1)	0				
C (1 1)	1				

J_1



$J_1 =$

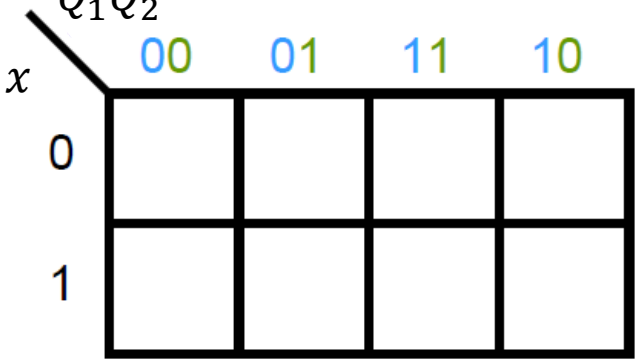
K_1



$K_1 =$

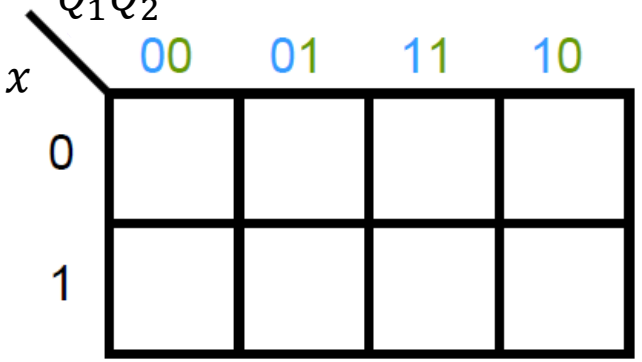
Present State (Q_1Q_2)	Input X	Flip-Flops J_1	K_1	J_2	K_2
A (0 0)	0				
A (0 0)	1				
B (0 1)	0				
B (0 1)	1				
(1 0)	x				
C (1 1)	0				
C (1 1)	1				

J_2



$J_2 =$

K_2



$K_2 =$

Mealy *vs* Moore Machines

Mealy machine

Present State	Input *X*	
	0	**1**
A	A/0	B/0
B	A/0	C/0
C	A/0	C/1

Moore machine

Present State	Input *X*		Present Output *Z*
	0	**1**	
A	A	B	0
B	A	C	0
C	A	D	0
D	A	D	1

Moore machine

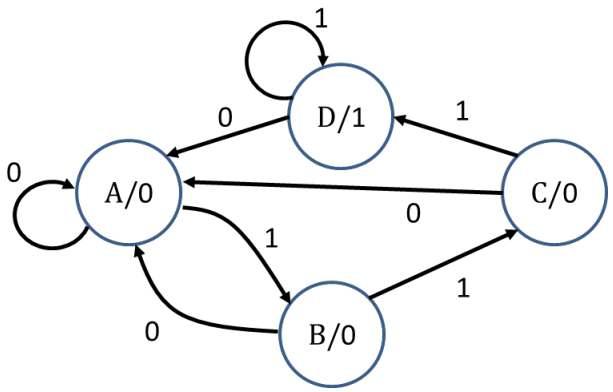
- Typically more states, more complex logic circuits

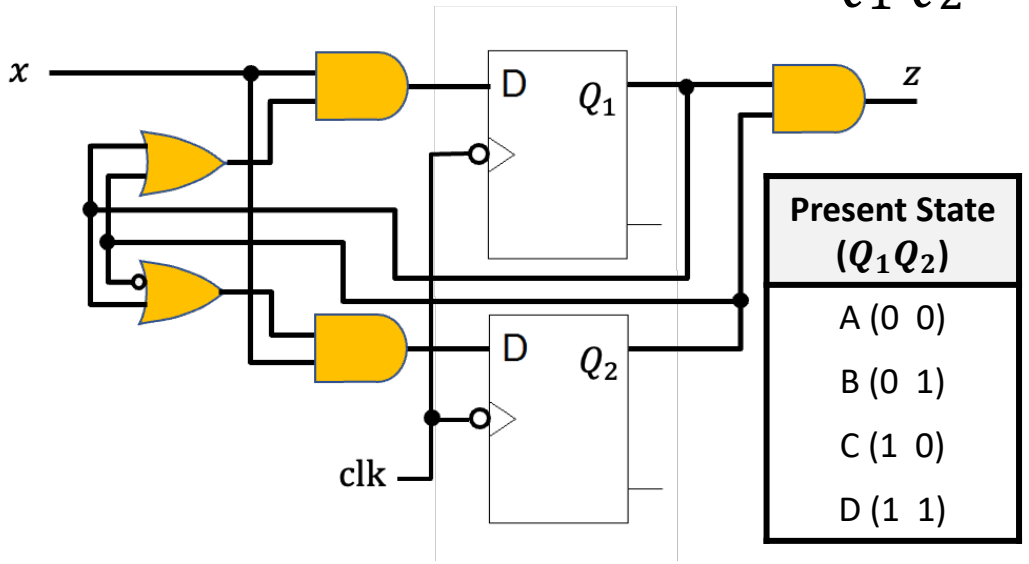
Mealy machine

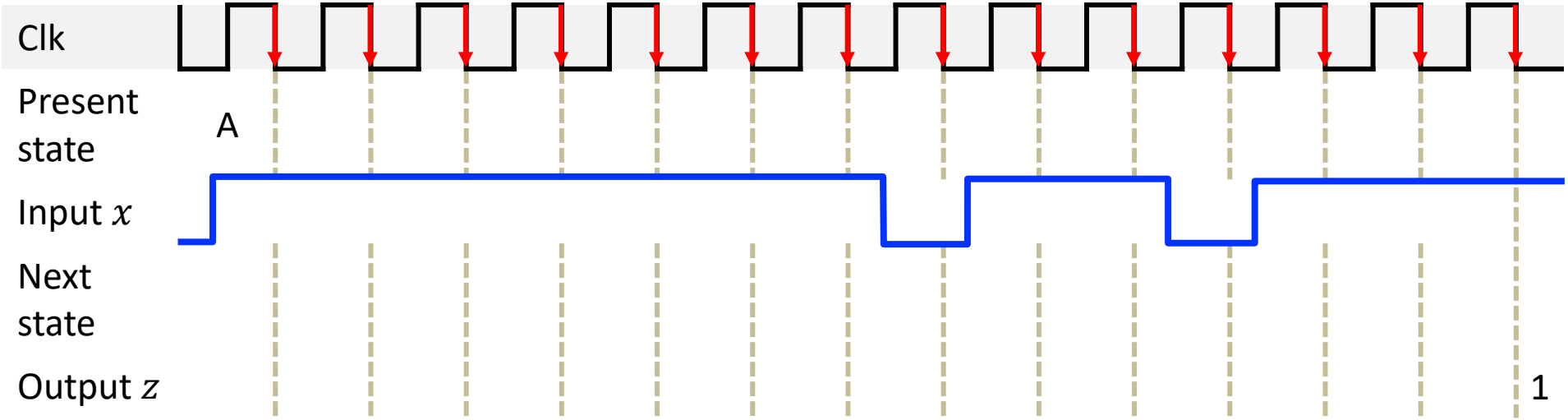
+ Typically fewer states, simpler logic circuits

Example (Timing Diagram)

$$z = Q_1 Q_2$$

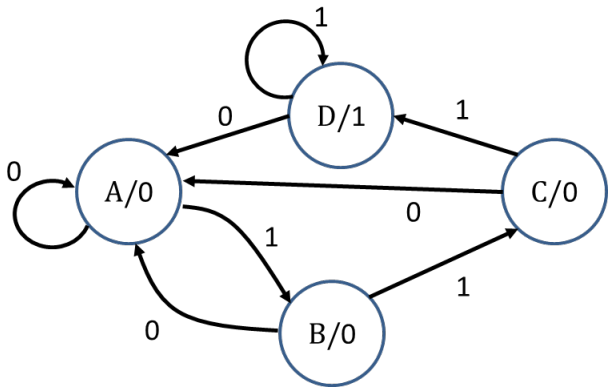


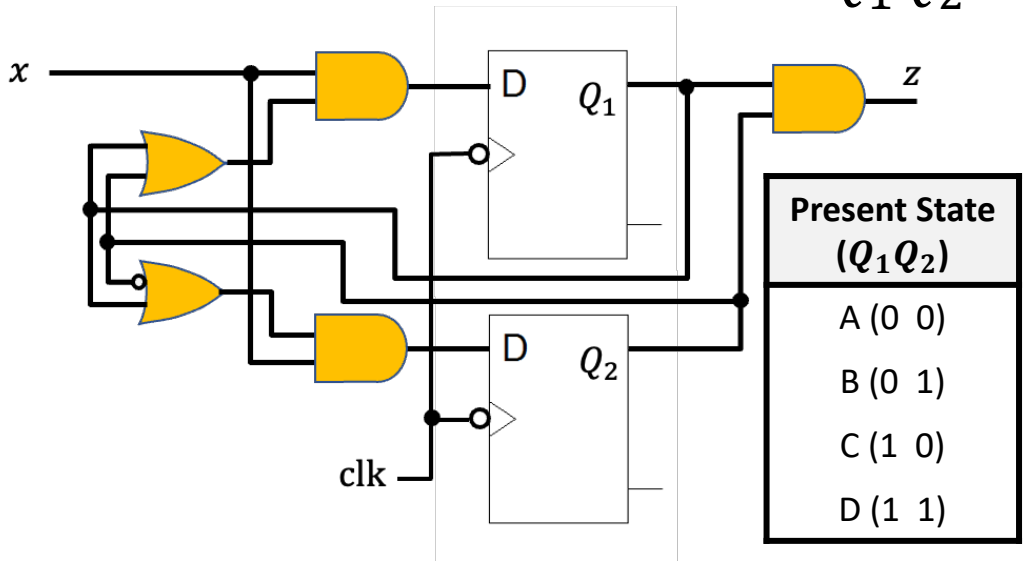




Example (Timing Diagram)

$$z = Q_1 Q_2$$

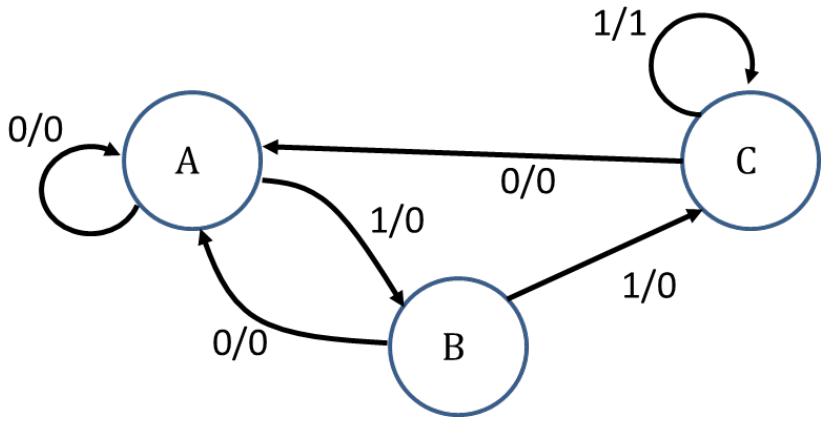


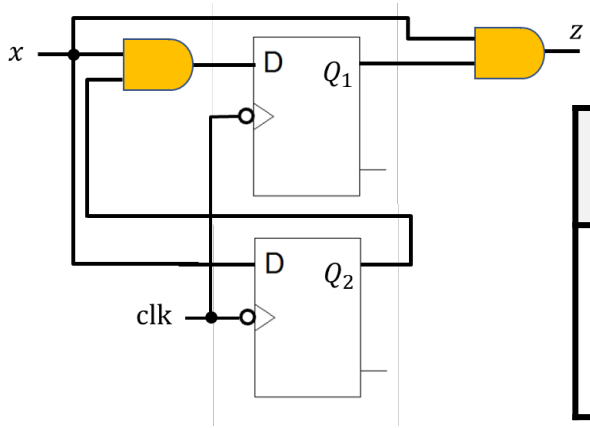


Clk														
Present state	A	B	C	D	D	D	D	D	A	B	C	A	B	C
Input x	1	1	1	1	1	1	1	0	1	1	0	1	1	1
Next state	B	C	D	D	D	D	D	A	B	C	A	B	C	D
Output z	0	0	0	1	1	1	1	1	0	0	0	0	0	0

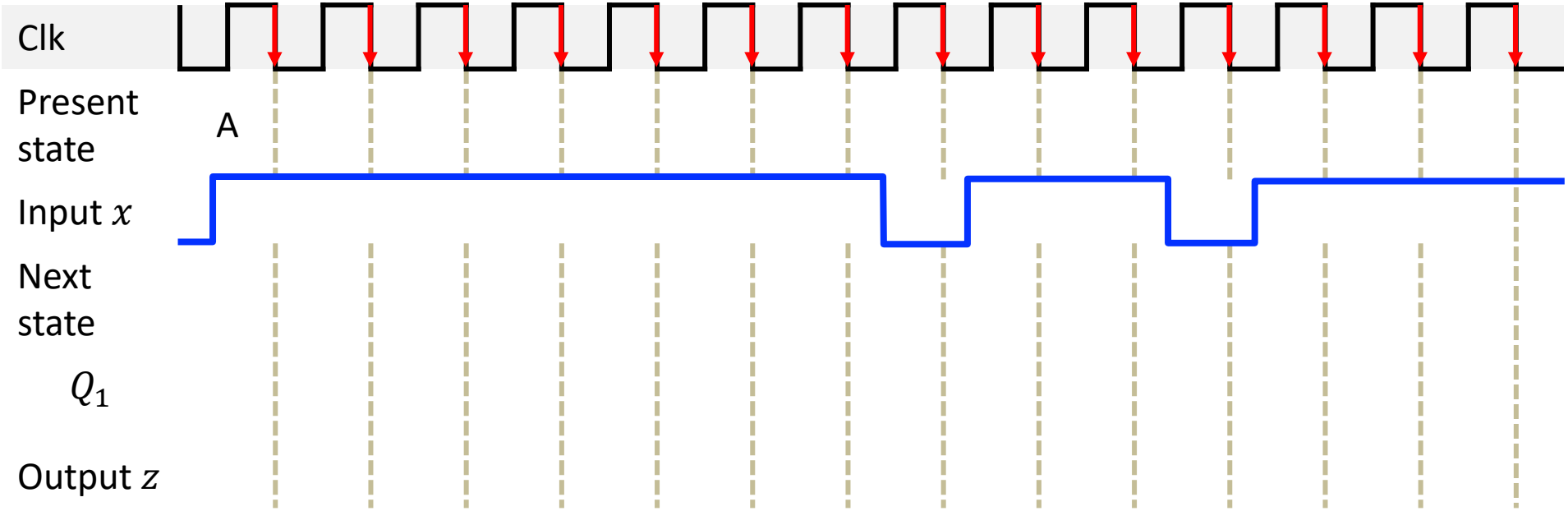
Exercise (Timing Diagram)

$$z = xQ_1$$

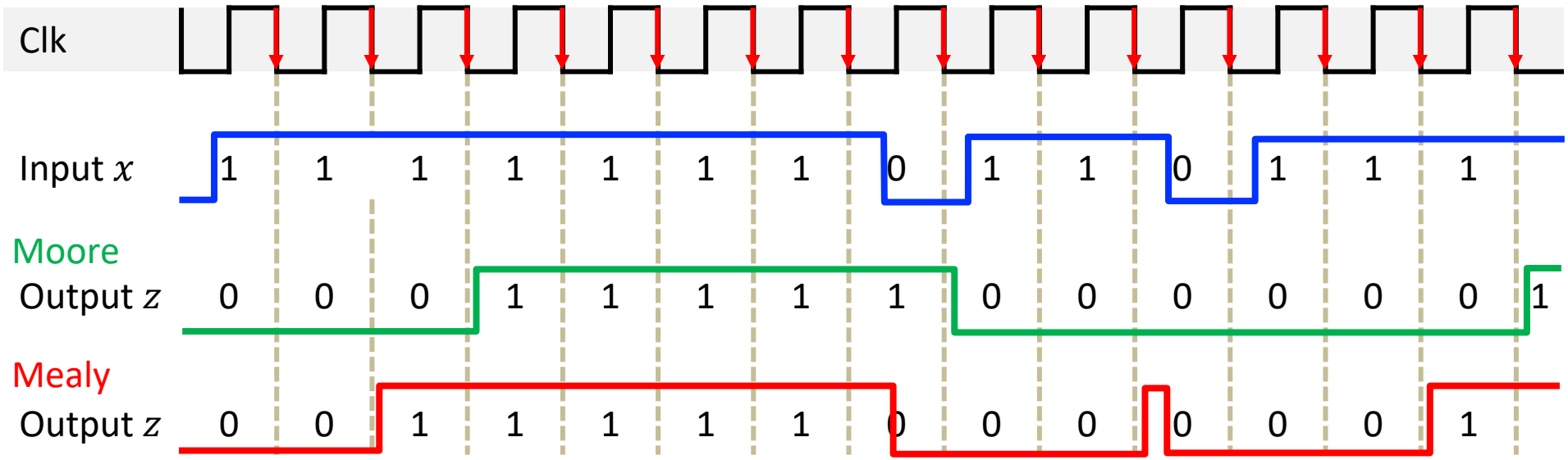




Present State (Q_1Q_2)
A (0 0)
B (0 1)
C (1 1)



Mealy *vs* Moore Machines



Observation

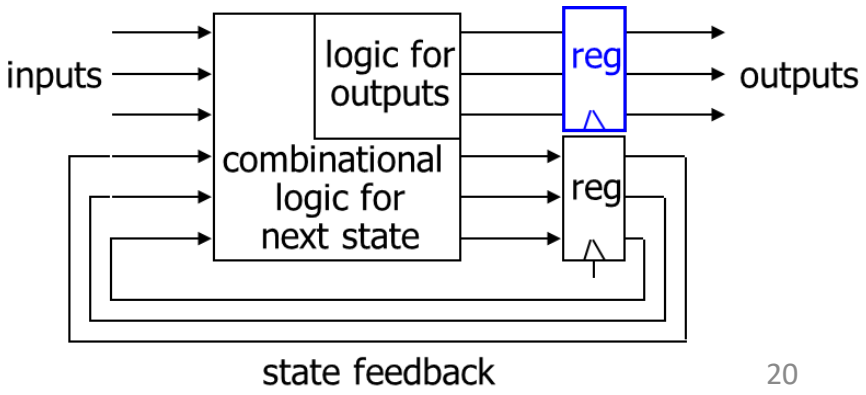
- Moore: Output changes occur only after clk edge
- Mealy: Output changes occur whenever input changes
- Mealy: Faster response but glitch might occurs

Mealy *vs* Moore Machines

Mealy machine	Moore machine
Output depends on inputs and present state	Output depends only on present state
Typically fewer states, simpler logic circuits	Typically more states, more complex logic circuits
React faster to inputs	React one clock cycle later
Asynchronous	Synchronous
Glitches might present	No glitch

Better solution?

Synchronous Mealy machine



Exercise (Sequence Detector)

Design a Moore machine to detect the sequence “11” or “000” (Overlapping)

In other words,

Design a system with one input x and one output z such that z = 1 if x has been 1 for at least two consecutive clock times or 0 for at least three consecutive clock times.

x	0	0	0	0	1	0	1	1	0	0	1	1	1	0
z	?	?	?	1	1	0	0	0	1	0	0	0	1	1

Exercise (Sequence Detector)

x	0	0	0	0	1	0	1	1	0	0	1	1	1	0
z	?	?	?	1	1	0	0	0	1	0	0	0	1	1

(Hint: 5 states)

Exercise (Sequence Detector)

Design a Mealy machine to detect the sequence “00110” (No overlapping)

In other words,

Design a system with one input x and one output z such that z = 1 if x has been 0 for two consecutive clock times, follows by two 1’s and then a 0.

x	0	0	1	1	0	0	1	1	0	0	1	1	0	1
z	0	0	0	0	1	0	0	0	0	0	0	0	1	0

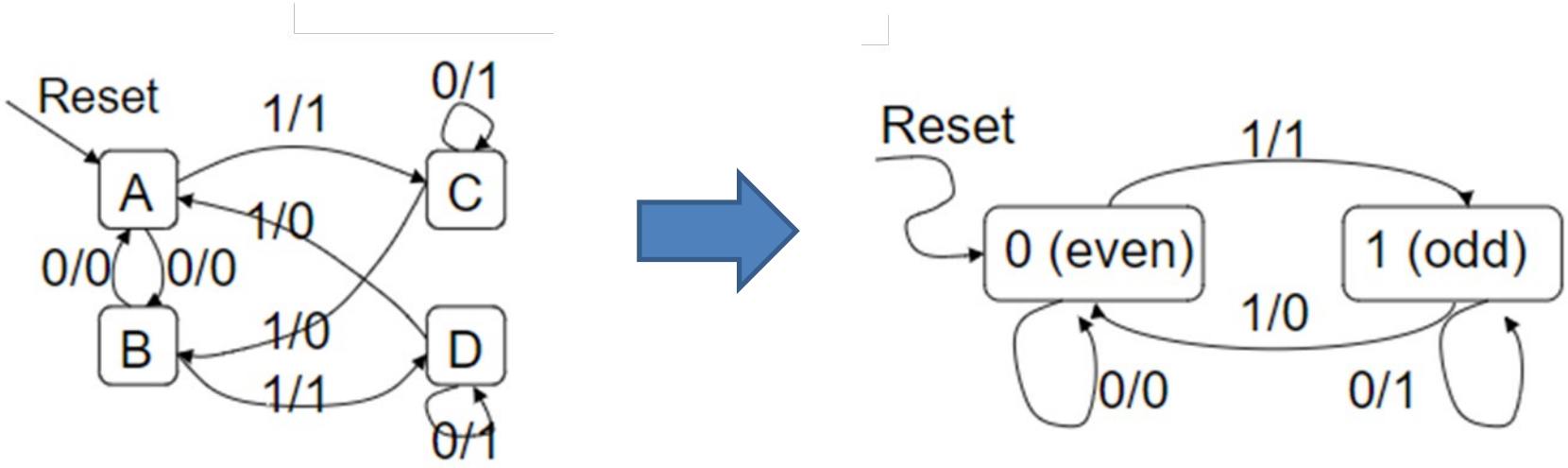
Exercise (Sequence Detector)

x	0	0	1	1	0	0	1	1	0	0	1	1	0	1
z	0	0	0	0	1	0	0	0	0	0	0	0	1	0

(Hint: 5 states)

State Minimization

- No of FFs ∝ No of states
- Combinational logic complexity ∝ No of states
- More FFs, more complex logic circuits → higher COST!
- Solution: Aims to remove redundant states



State Minimization

- Direct observation

Identify same output combinations and same state

	0	1
A	A/0	E/1
B	E/1	C/0
C	A/1	D/1
D	F/0	G/1
E	B/1	C/0
F	F/0	E/1
G	A/1	D/1

	0	1
A	A/0	E/1
B	E/1	C/0
C	A/1	D/1
D	F/0	G/1
E	B/1	C/0
F	F/0	E/1
G	A/1	D/1

	0	1
A	A/0	E/1
B	E/1	C/0
C	A/1	D/1
D	F/0	G/1
E	B/1	C/0
F	F/0	E/1
G	A/1	D/1

	0	1
A	A/0	E/1
B	E/1	C/0
C	A/1	D/1
D	F/0	G/1
E	B/1	C/0
F	F/0	E/1
G	A/1	D/1

	0	1
A=F	A/0	B/1
B=E	B/1	C/0
C=G	A/1	D/1
D	A/0	C/1

Partitioning Method

	0	1
A	A/0	E/1
B	E/1	C/0
C	A/1	D/1
D	F/0	G/1
E	B/1	C/0
F	F/0	E/1
G	A/1	D/1

- Separate states with different outputs to different partitions

$$P_0 = (\mathrm{A}\ \mathrm{B}\ \mathrm{C}\ \mathrm{D}\ \mathrm{E}\ \mathrm{F}\ \mathrm{G})$$

- A, D and F have outputs (0 1); B and E have outputs (1 0); C and G have outputs (1 1)

$$P_1 = (\mathrm{A}\ \mathrm{D}\ \mathrm{F}\ )(\mathrm{B}\ \mathrm{E})(\mathrm{C}\ \mathrm{G})$$

Partitioning Method

P		0	1
1	A	A/0	E/1
	D	F/0	G/1
	F	F/0	E/1
2	B	E/1	C/0
	E	B/1	C/0
3	C	A/1	D/1
	G	A/1	D/1

$$P_1 = (\mathrm{A\ D\ F})(\mathrm{B\ E})(\mathrm{C\ G})$$

- Next check the next state of each state in each partition
  - A(0) → A (P1) A(1) → E (P2)
  - D(0) → F (P1) D(1) → G (P3)
  - F(0) → F (P1) F(1) → E (P2)

  ∴ A and F same partitions; D is different

$$P_2 = (\mathrm{A\ F})(\mathrm{D})(\mathrm{B\ E})(\mathrm{C\ G})$$

Partitioning Method

P		0	1
1	A	A/0	E/1
	F	F/0	E/1
2	B	E/1	C/0
	E	B/1	C/0
3	C	A/1	D/1
	G	A/1	D/1
4	D	F/0	G/1

$$P_2 = (\mathrm{A\ F})(\mathrm{D})(\mathrm{B\ E})(\mathrm{C\ G})$$

This is the final with no more changes!

	I	J
A=F	A/0	B/1
B=E	B/1	C/0
C=G	A/1	D/1
D	A/0	C/1

Exercise

Present State	Input *X*		Present Output *Z*
	0	1	
A	I	C	0
B	B	I	0
C	C	G	0
D	I	C	1
E	D	E	1
F	I	C	1
G	E	F	1
H	H	A	0
I	A	C	0

Reduce the state table using partitioning method

$$P_0 = (\mathrm{A\ B\ C\ D\ E\ F\ G\ H\ I})$$

Group states based on same output

Exercise

Present State	Input *X*		Present Output *Z*
	0	1	

Reduce the state table using partitioning method

$$P_1 =$$

Exercise

Present State	Input *X* 0	Input *X* 1	Present Output *Z*

Reduce the state table using partitioning method

$P_2 =$

Exercise

Present State	Input *X*		Present Output *Z*
	0	1	

Reduce the state table using partitioning method

$P_3 =$

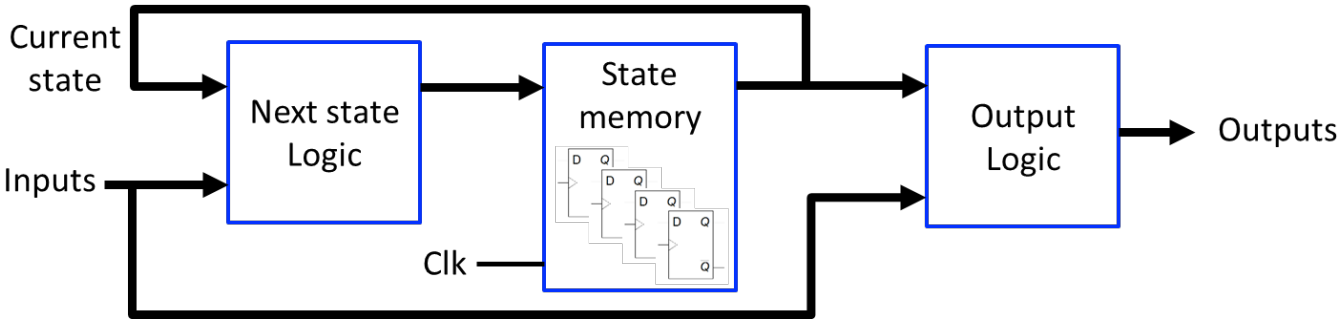
Exercise

Present State	Input *X*		Present Output *Z*
	0	1	

Reduce the state table using partitioning method

$P_3 =$

VHDL for Finite State Machine

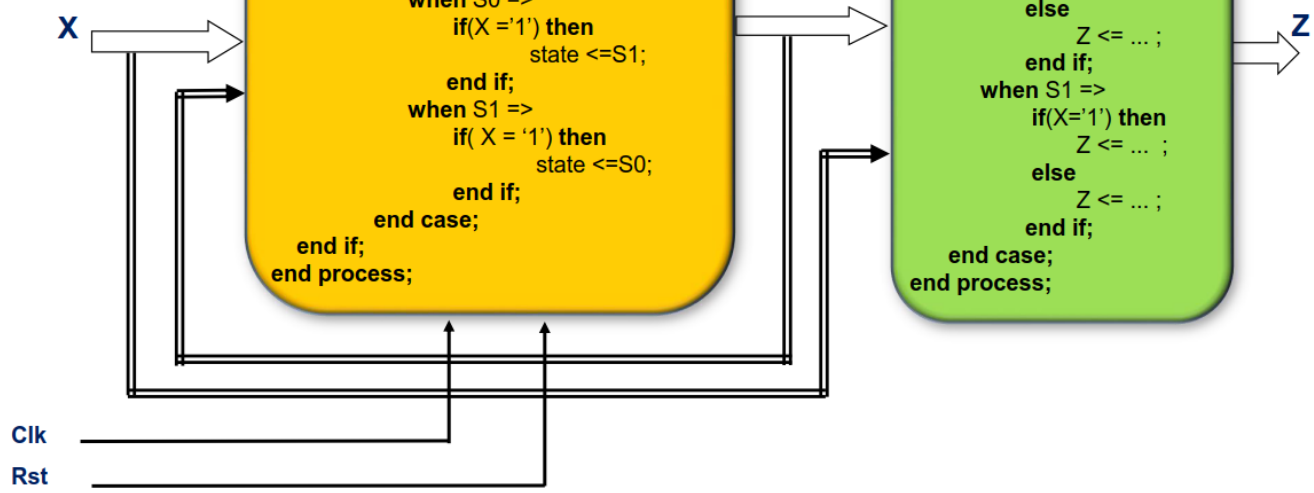


Seq. Present State and Next State Logic

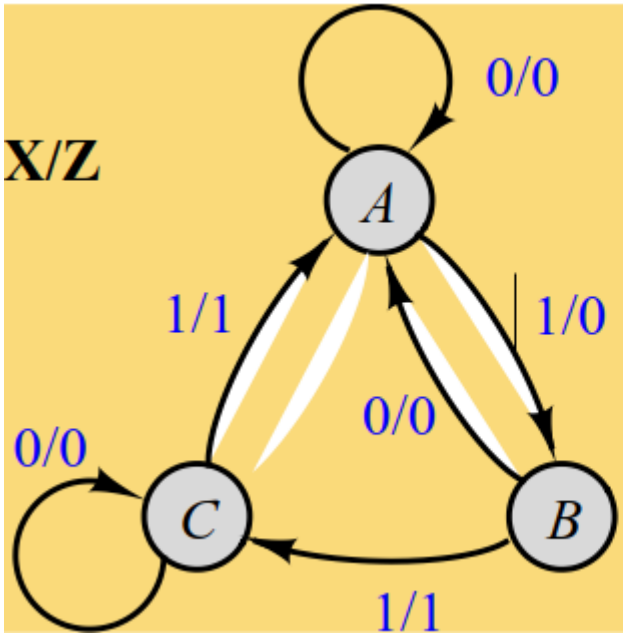
```
process (clk, rst)
begin
   if(rst = '1') then
            State <= S0;
   elsif (rising_edge(clk)) then
            case state is
               when S0 =>
                     if(X ='1') then
                              state <=S1;
                     end if;
               when S1 =>
                     if( X = '1') then
                              state <=S0;
                     end if;
            end case;
   end if;
end process;
```

Comb. Output Logic

```
process (state, X)
begin
   case state is
         when S0 =>
               if(X='1') then
                     Z <= ... ;
               else
                     Z <= ... ;
               end if;
         when S1 =>
               if(X='1') then
                     Z <= ... ;
               else
                     Z <= ... ;
               end if;
   end case;
end process;
```



Example

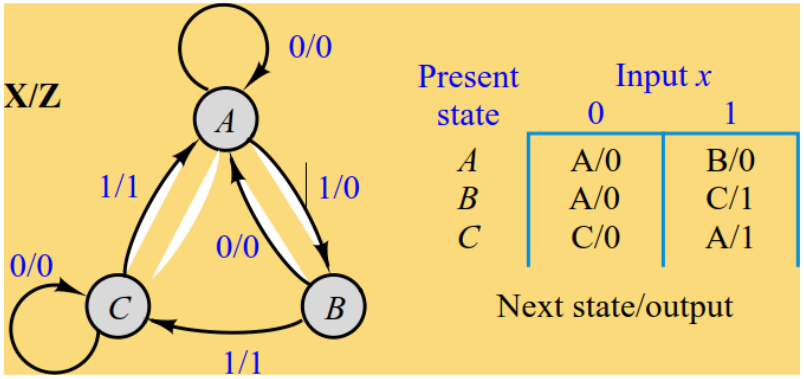


Present state	Input x = 0	Input x = 1
A	A/0	B/0
B	A/0	C/1
C	C/0	A/1

Next state/output

```
entity seqckt is
port (     x: in std_logic; -- FSM input
           z: out std_logic; -- FSM output
         clk: in std_logic ); -- clock
end seqckt;
```

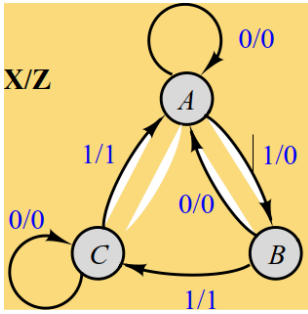
Output Logic




```
architecture behave of seqckt is

  type states is (A,B,C);   -- symbolic state names (enumerate)
  signal state: states;     --state variable

Begin

-- Output Logic

z <= '1' when ((state = B) and (x = '1'))    --all conditions
           or ((state = C) and (x = '1'))    --for which z = 1
        else '0';                            --otherwise z = 0
```

Next State Logic

```
process (clk) - Present & next states logic
  begin
    if rising_edge(clk) then -- clock edge
      case state is
        when A => if (x = '0') then
                      state <= A;
                  else
                      state <= B;   -- x = '1'
                  end if;
        when B => if (x='0') then
                      state <= A;
                  else
                      state <= C;   -- x = '1'
                  end if;
        when C => if (x='0') then
                      state <= C;
                  else
                      state <= A;   -- x = '1'
                  end if;
      end case;
    end if;
end process;
end behave;
```

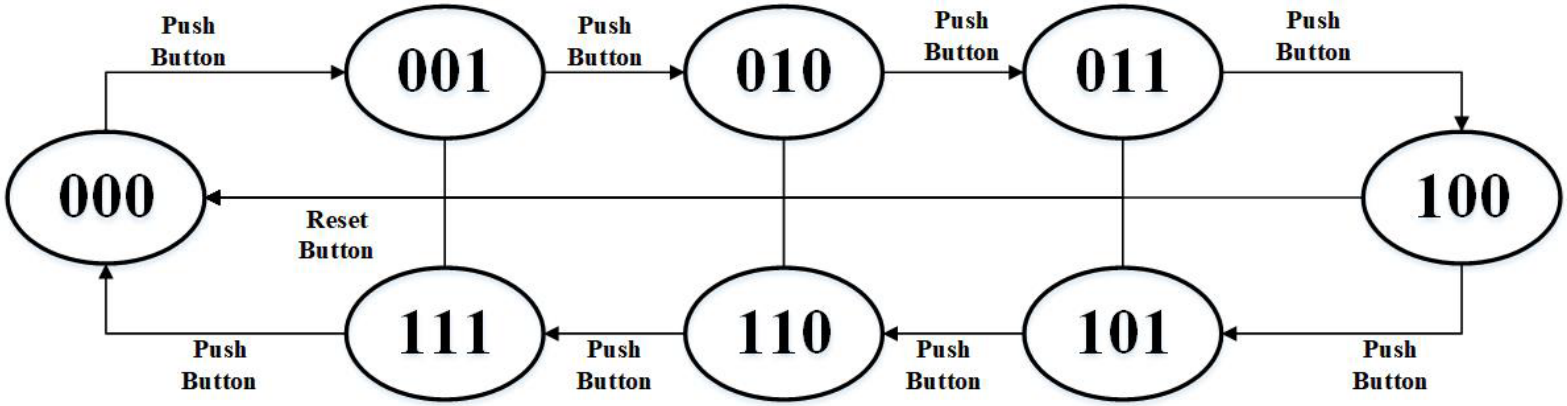


Present state	Input x = 0	Input x = 1
A	A/0	B/0
B	A/0	C/1
C	C/0	A/1

Next state/output

Lab Session 4

- Implement a **simple finite-state machine** with a good coding style
- Implement **two buttons** to control the state change of the FSM
- Display the state transition with the **LEDs** on the BASYS3 board



VHDL Codes

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity fsm is
Port (
clk: in STD_LOGIC;
rst: in STD_LOGIC;
ctrlButton: in STD_LOGIC;
led: out STD_LOGIC_VECTOR (7 DOWNTO 0)
);
end fsm;
```

Inputs

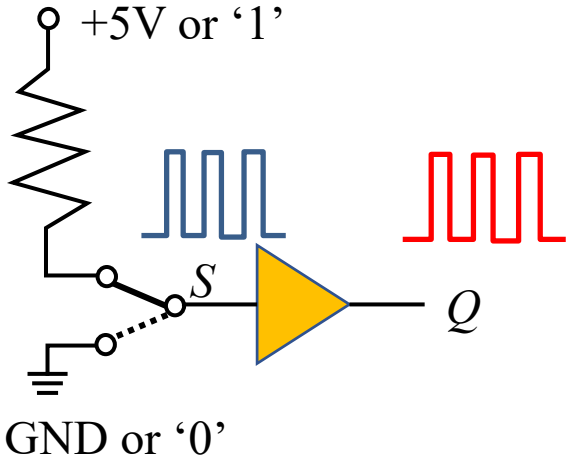
- clock signal (clk)
- Master reset button (rst)
- Control button (ctrlButton)

Output

- 8 LEDS for 8 states

Key Debouncing

When pressing a key or switch, its internal spring may cause the contact point to vibrate for a short period of time, causing a series of “ON” and “OFF” state before settling down



Inputs

- clock signal (clk)
- Master reset button (rst)
- Control button (ctrlButton)

Output

- 8 LEDS for 8 states

VHDL Codes

```
Constant TIME_10ms: integer := 1000000;
Signal key_counter: integer range 0 to 1000001;
Signal btnDetection: boolean;

begin
  Key_detection: process (clk) begin
   if rising_edge(clk) then
    if (ctrlButton='1') then
       key_counter <= 0;
    elsif (ctrlButton='0') and (key_counter <= TIME_10MS) then
       key_counter <= key_counter +1;
       end if; end if;
end process;
btnDetection <= (key_counter = TIME_10ms);
```

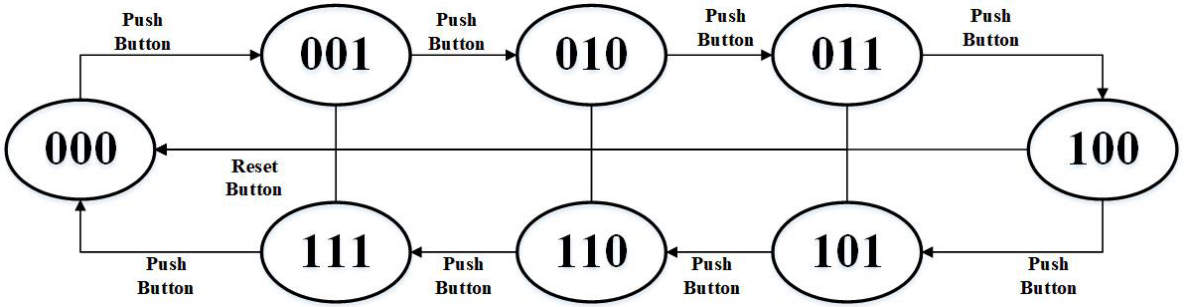
When control button is pushed, **key_counter** will set to 0. Then, the timer will start counting and if the stable time reach to **TIME_10ms**, **btnDetection** sets to true for that cycle.

VHDL Codes

Internal signals
- data types: states
- present_state
- next_state

```
Architecture Behavioral of fsm is

type states is (zero, one, two, three, four, five, six, seven);
signal present_state, next_state: states;
```

VHDL Codes



The sequential logic of our FSM is very simple. Every time the ***btnDetection*** is true, the FSM will jump to the next state. We define a type called ***states*** enumerating all possible states. Then we define two signals present_state and next_state which are both ***states*** type. Every time the btnDetection is true, the value of next_state will be provided to present_state (meaning the state jumps to the next state). The assignment of next_state will be introduced in the next section

```
state_transition: process (rst, clk)
begin
    if (rst='1') then
        --Write your code here to describe state transition.
    elsif (rising_edge(clk) and btnDetection = true ) then
        --Write your code here to describe state transition.
    end if;
end process;
```

VHDL Codes

present_state	LED Output
zero	00000001
one	00000010
two	00000100
three	00001000
four	00010000
five	00100000
six	01000000
seven	10000000

The corresponding VHDL template and you can complete the following code.

```
    decoder: process (present_state)
    begin
        case present_state is
            --Write your code here to assign LED and next_state
        end case;
    end process;
end Behavioral;
```