1- Introduction

2- Basic Structures of VHDL

3- Combinational Circuits

4- Sequential Circuits

5- Memory

6- Writing Testbenches

7- Synthesis Issues

8- RTL Cores

# 4- Sequential Circuits

- **Basic Memory Elements**
  - Latch
  - D Flip-Flop
- **Registers, Shifters and Counters**
- **State Machine Coding**
  - Moore
  - Mealy
  - Huffman

# Basic Memory Elements

- Latch

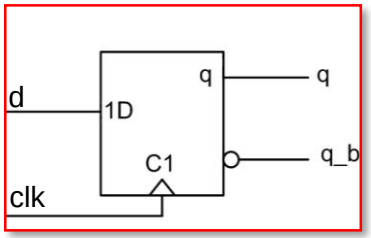
```
1  LIBRARY IEEE;
2  USE IEEE.std_logic_1164.ALL;
3
4  ENTITY latch1 IS
5      PORT (d, c: IN std_logic; q: OUT std_logic);
6  END latch1;
7
8  ARCHITECTURE behavioral OF latch1 IS
9  BEGIN
10
11     PROCESS (d, c)
12     BEGIN
13         IF c = '1' THEN
14             q <= d;
15         END IF;
16     END PROCESS;
17
18 END ARCHITECTURE behavioral;
```

- Use a process statement for describing a latch
- While *c* is one, *d* drives *q*
- Transparency when *c* is one

# Basic Memory Elements

- A Positive-Edge D Flip-Flop

```
1  LIBRARY IEEE;
2  USE IEEE.std_logic_1164.ALL;
3
4  ENTITY DFF1 IS
5      PORT (d, clk: IN std_logic; q : OUT std_logic);
6  END DFF1;
7
8  ARCHITECTURE behavioral OF DFF1 IS
9  BEGIN
10
11     PROCESS (clk)
12     BEGIN
13         IF (clk = '1' AND clk'EVENT) THEN
14             q <= d;
15         END IF;
16     END PROCESS;
17
18 END ARCHITECTURE behavioral;
```

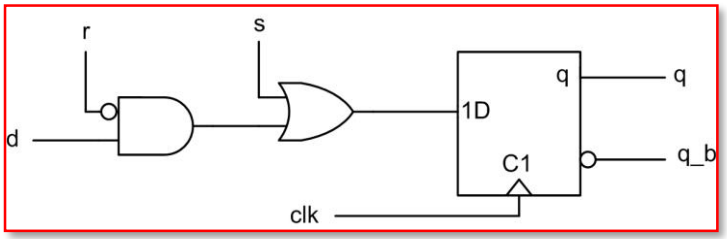



- Detect the edge of the clock and then put *d* into *q*
- An event on *clk* accompanied with *clk*=1, detects the rising edge
- This is a clocked process

# Basic Memory Elements

- DFF with Synchronous Control

```
1  LIBRARY IEEE;
2  USE IEEE.std_logic_1164.ALL;
3
4  ENTITY DFF1sr IS
5      PORT (d, clk, s, r: IN std_logic; q : OUT std_logic);
6  END DFF1sr;
7
8  ARCHITECTURE behavioral OF DFF1sr IS
9  BEGIN
10
11     PROCESS (clk)
12     BEGIN
13         IF clk = '1' AND clk'EVENT THEN
14             IF s = '1' THEN
15                 q <= '1';
16             ELSIF r = '1' THEN
17                 q <= '0';
18             ELSE
19                 q <= d;
20             END IF;
21         END IF;
22     END PROCESS;
23
24 END ARCHITECTURE behavioral;
```

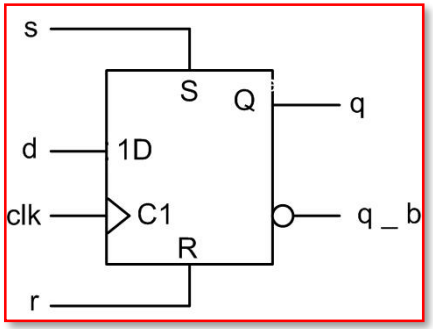



- Use sequential statements in a clocked process to add other functionalities
- This is synchronous control, *r*, *s*

# Basic Memory Elements

- DFF with Asynchronous Control

```
1  LIBRARY IEEE;
2  USE IEEE.std_logic_1164.ALL;
3
4  ENTITY DFF1sr IS
5      PORT (d, clk, s, r: IN std_logic; q : OUT std_logic);
6  END DFF1sr;
7
8  ARCHITECTURE asynchronous OF DFF1sr IS
9  BEGIN
10     PROCESS (clk, s, r) BEGIN
11         IF s = '1' THEN
12             q <= '1';
13         ELSIF r = '1' THEN
14             q <= '0';
15         ELSIF (clk = '1' AND clk'EVENT) THEN
16             q <= d;
17         END IF;
18     END PROCESS;
19 END ARCHITECTURE asynchronous;
```




- Implement asynchronous behavior by including corresponding signals in the sensitivity list.
- Clock edge detection becomes the last condition

# Registers, Shifters and Counters

- Register

```
1  LIBRARY IEEE;
2  USE IEEE.std_logic_1164.ALL;
3
4  ENTITY register8 IS
5      PORT (
6          d      : IN std_logic_vector (7 DOWNTO 0);
7          clk, s, r      : IN std_logic;
8          q      : OUT std_logic_vector ( 7 DOWNTO 0));
9  END register8;
10
11 ARCHITECTURE behavioral OF register8 IS
12 BEGIN
13     PROCESS (clk)
14     BEGIN
15         IF (clk = '1' AND clk'EVENT) THEN
16             IF s= '1' THEN
17                 q <= (OTHERS => '1');
18             ELSIF r = '1' THEN
19                 q <= (OTHERS => '0');
20             ELSE
21                 q <= d;
22             END IF;
23         END IF;
24     END PROCESS;
25 END behavioral;
```

- For registers use *std_logic_vector*
- Everything else basically remains the same

# Registers, Shifters and Counters

- Shift-Registers

```vhdl
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY shift_reg4 IS
   PORT (
      d   : IN std_logic_vector (3 DOWNTO 0);
      clk, ld, rst, l_r, s_in : IN std_logic;
      q   : OUT std_logic_vector (3 DOWNTO 0));
END shift_reg4;

ARCHITECTURE behavioral OF shift_reg4 IS
BEGIN
    PROCESS (clk)
        VARIABLE q_t: std_logic_vector (3 DOWNTO 0);
    BEGIN
        IF (clk = '1' AND clk'EVENT) THEN
            IF rst= '1' THEN
                q_t := (OTHERS => '0');
            ELSIF ld = '1' THEN
                q_t := d;
            ELSIF l_r = '1' THEN
                q_t :=  q_t (2 DOWNTO 0) & s_in ;
            ELSE
                q_t := s_in & q_t (3 DOWNTO 1);
            END IF;
        END IF;
        q <= q_t;
    END PROCESS;
END behavioral;
```

- Can add functionality such as increment, decrement, shift, etc.
- This is a right/left shift register
- In VHDL cannot use output on RHS, so use a temporary variable

# Registers, Shifters and Counters

- Counter

```
1  LIBRARY IEEE;
2  USE IEEE.std_logic_1164.ALL;
3  USE IEEE.std_logic_unsigned.ALL;
4
5  ENTITY counter4 IS
6      PORT (clk, reset : IN std_logic;
7            count : OUT std_logic_vector (3 DOWNTO 0));
8  END ENTITY;
9
10 ARCHITECTURE procedural OF counter4 IS
11     SIGNAL cnt_reg : std_logic_vector (3 DOWNTO 0);
12 BEGIN
13     PROCESS (clk)
14     BEGIN
15         IF (clk = '1' AND clk'EVENT) THEN
16             IF (reset='1') THEN
17                 cnt_reg <="0000";
18             ELSE
19                 cnt_reg <= cnt_reg + "0001";
20             END IF;
21         END IF;
22     END PROCESS;
23     count <= cnt_reg;
24 END ARCHITECTURE procedural;
```

- Synchronous reset
- Functionality is to increment
- Cannot use output on RHS
- Use a temporary signal, i.e., *cnt_reg*

# Registers, Shifters and Counters

- Unconstrained Counter

```
1  LIBRARY IEEE;
2  USE IEEE.std_logic_1164.ALL;
3  USE IEEE.std_logic_unsigned.ALL;
4
5  ENTITY counterN IS
6      PORT (clk, reset, load : IN std_logic;
7            d : IN std_logic_vector;
8            count : OUT std_logic_vector);
9  END ENTITY;
10
11 ARCHITECTURE procedural OF counterN IS
12     SIGNAL cnt_reg : std_logic_vector( d'RANGE );
13 BEGIN
14     PROCESS (clk)
15     BEGIN
16         IF (clk = '1' AND clk'EVENT) THEN
17             IF (reset = '1') THEN
18                 cnt_reg <= (OTHERS=>'0');
19             ELSIF (load = '1') THEN
20                 cnt_reg <= d;
21             ELSE
22                 cnt_reg <= cnt_reg + 1;
23             END IF;
24         END IF;
25     END PROCESS;
26     count <= cnt_reg;
27 END ARCHITECTURE procedural;
28
```

- N-bit up counter uses unconstrained vectors
- Size is fixed when instantiated

# Registers, Shifters and Counters

- Unconstrained Counter with Generic

```
1  LIBRARY IEEE;
2  USE IEEE.std_logic_1164.ALL;
3  USE IEEE.std_logic_unsigned.ALL;
4  USE IEEE.numeric_std.ALL;
5
6  ENTITY counterNupM IS
7      GENERIC( M: integer:=4 );
8      PORT (clk, reset, load : IN std_logic;
9            d : IN std_logic_vector;
10           count : OUT std_logic_vector);
11 END ENTITY;
12
13 ARCHITECTURE procedural OF counterNupM IS
14     SIGNAL cnt_reg : std_logic_vector( d'RANGE );
15 BEGIN
16     PROCESS (clk)
17     BEGIN
18         IF (clk = '1' AND clk'EVENT) THEN
19             IF (reset = '1') THEN
20                 cnt_reg <= (OTHERS=>'0');
21             ELSIF (load = '1') THEN
22                 cnt_reg <= d;
23             ELSE
24                 cnt_reg <= cnt_reg + std_logic_vector(to_unsigned(M, d'length));
25             END IF;
26         END IF;
27     END PROCESS;
28     count <= cnt_reg;
29 END ARCHITECTURE procedural;
```

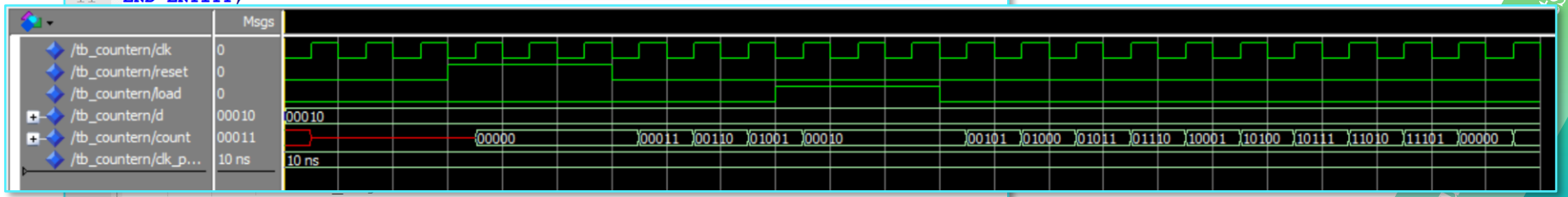
- Use **generic** for count up
- **generic** *M* will be determined by **generic map** when counter is instantiated

# Registers, Shifters and Counters

- Unconstrained Counter with G

```
uut: ENTITY WORK.counterNupM
        GENERIC MAP (M => 3)
        PORT MAP (
            clk => clk,
            reset => reset,
            load => load,
            d => "00010",
            count => count
            );
```

```
1  LIBRARY IEEE;
2  USE IEEE.std_logic_1164.ALL;
3  USE IEEE.std_logic_unsigned.ALL;
4  USE IEEE.numeric_std.ALL;
5
6  ENTITY counterNupM IS
7      GENERIC( M: integer:=4 );
8      PORT (clk, reset, load : IN std_
9            d : IN std_logic_vector;
10           count : OUT std_logic_vect
11 END ENTITY;
```

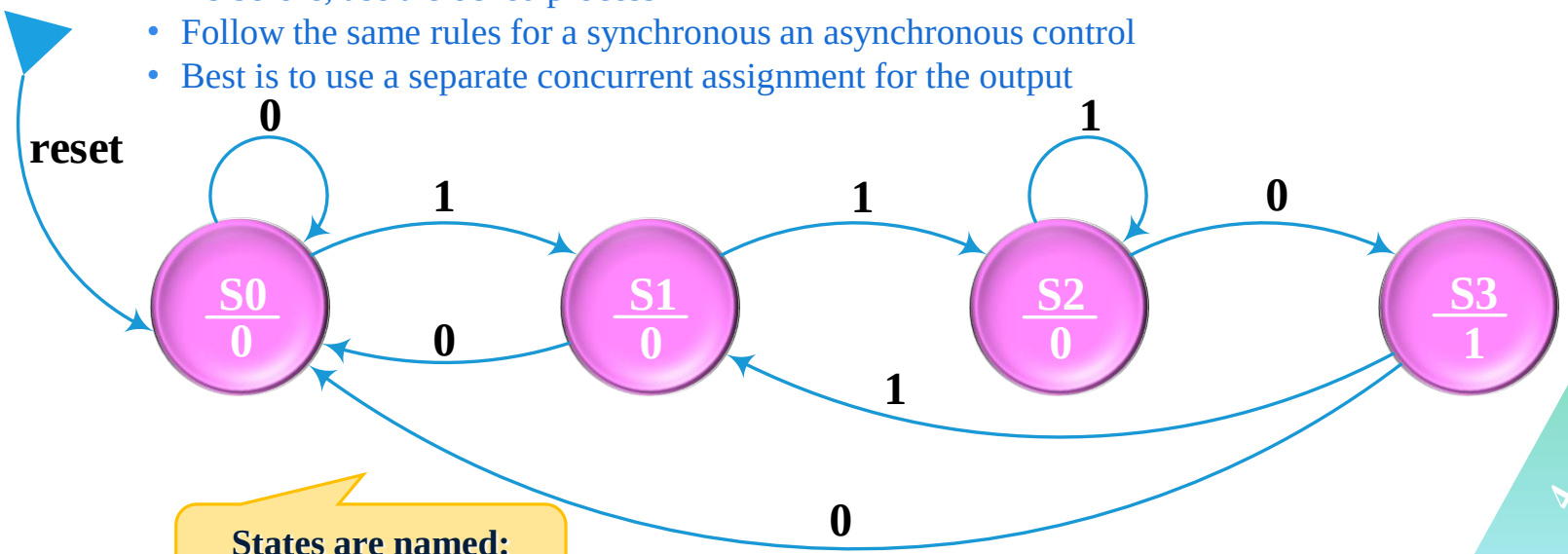



```
21             ELSIF (load = '1') THEN
22                 cnt_reg <= d;
23             ELSE
24                 cnt_reg <= cnt_reg + std_logic_vector(to_unsigned(M, d'length));
25             END IF;
26         END IF;
27     END PROCESS;
28     count <= cnt_reg;
29 END ARCHITECTURE procedural;
```

- Use **generic** for count up
- **generic** *M* will be determined by **generic map** when counter is instantiated

# State Machines

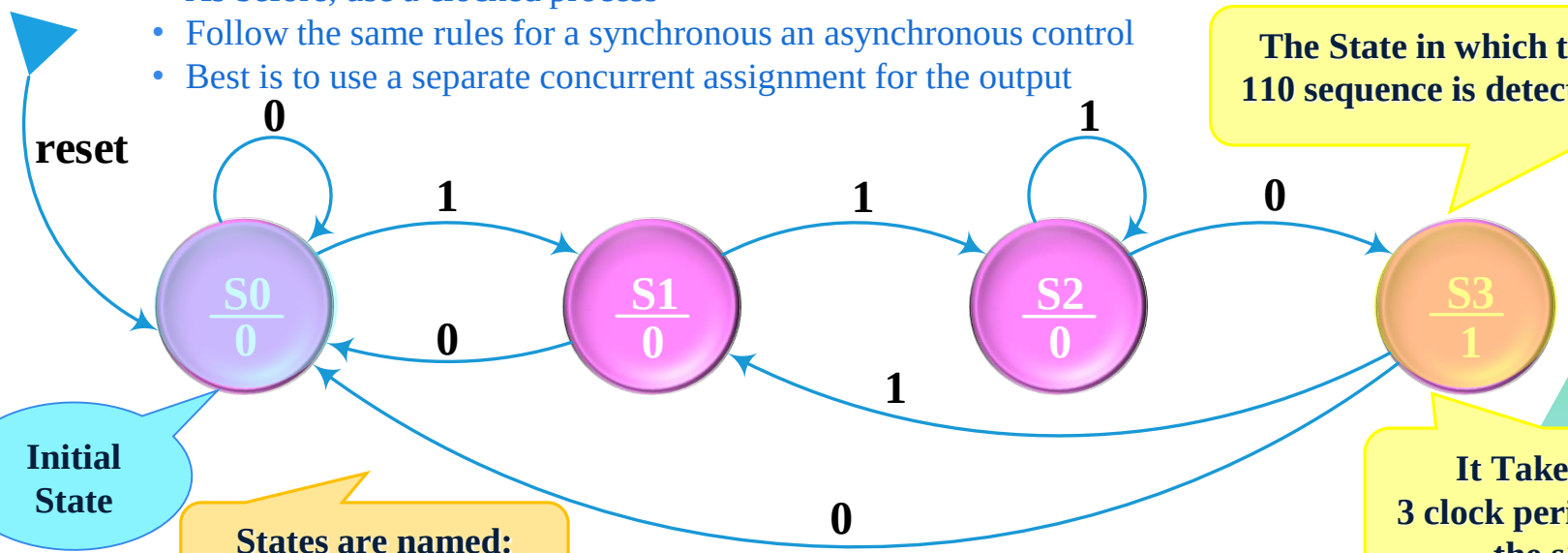
- **A Moore Machine 110 Sequence Detector**
- Define states by an enumeration type
- As before, use a clocked process
- Follow the same rules for a synchronous an asynchronous control
- Best is to use a separate concurrent assignment for the output




**States are named:**
***s0 , s1 , s2 , s3***

# State Machines

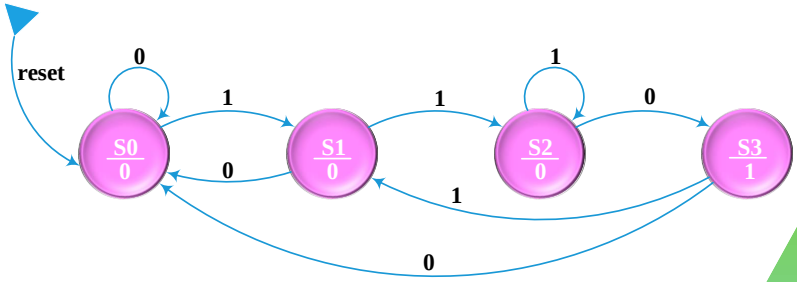
- **A Moore Machine 110 Sequence Detector**
- Define states by an enumeration type
- As before, use a clocked process
- Follow the same rules for a synchronous an asynchronous control
- Best is to use a separate concurrent assignment for the output



# State Machines

- **A Moore Machine 110 Sequence Detector**

```
1  LIBRARY IEEE;
2  USE IEEE.STD_LOGIC_1164.ALL;
3
4  ENTITY detector110 IS
5      PORT (a, clk, reset : IN std_logic; w : OUT std_logic);
6  END ENTITY;
7
8  ARCHITECTURE procedural OF detector110 IS
9      TYPE state IS (S0, S1, S2, S3);
10     SIGNAL current : state := S0;
11 BEGIN
12     PROCESS (clk) BEGIN
13         IF (clk = '1' AND clk'EVENT) THEN
14             IF reset = '1' THEN current <= S0;
15             ELSE
16                 CASE current IS
17                     WHEN S0 =>
18                         IF a='1' THEN current <= S1;
19                         ELSE current <= S0; END IF;
```

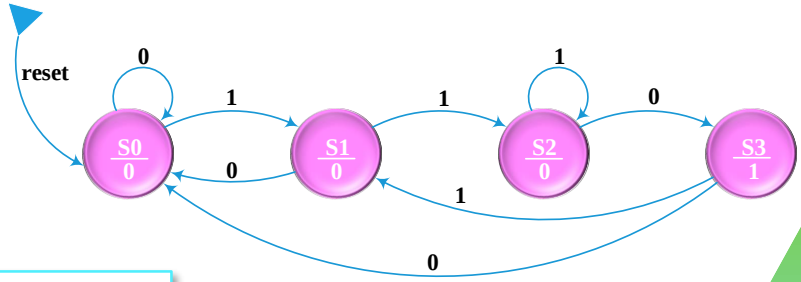



- Declare *current* as the register that holds the FSM state; two bits
- User a process statement for transitions
- Use a signal assignment for the output
- Can use a process statement for the output if output is more complex

# State Machines

- **A Moore Machine 110 Sequence Detector**

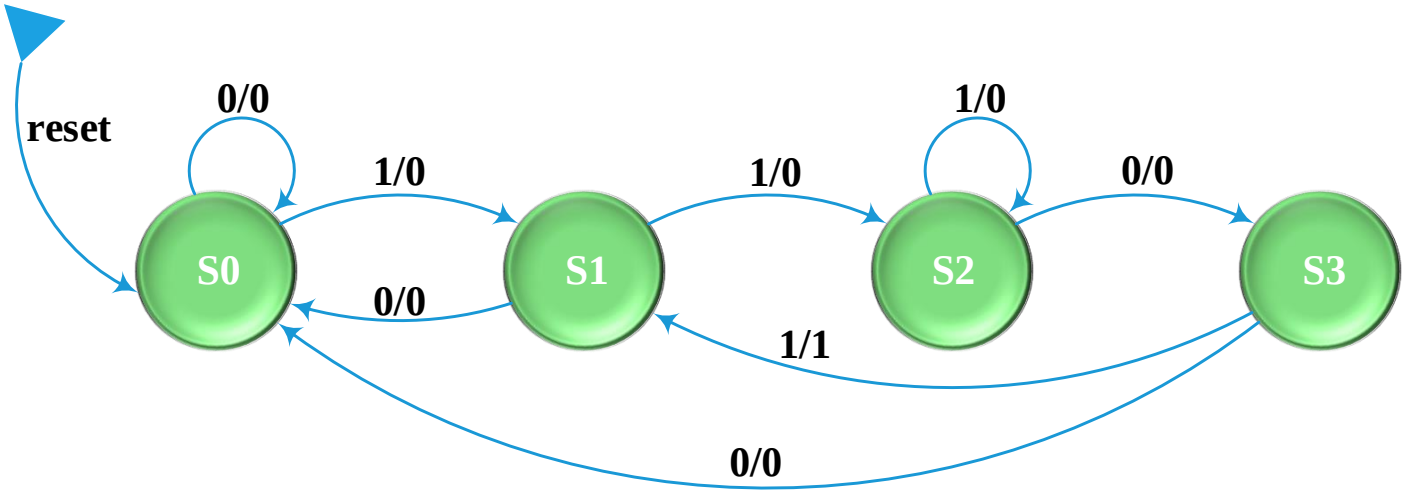
```
1  LIBRARY IEEE;
2  USE IEEE.STD_LOGIC_1164.ALL;
3
4  ENTITY detector110 IS
5      PORT (a, clk, reset : IN std_logic; w : OUT std_logic);
6  END ENTITY;
7
8  ARCHITECTURE procedural OF detector110 IS
9      TYPE s
10     SIGNAL
11 BEGIN
12     PROCES
13         IF
14
15
16
17
18
19
20                         WHEN S1 =>
21                             IF a='1' THEN current <= S2;
22                             ELSE current <= S0; END IF;
23                         WHEN S2 =>
24                             IF a='1' THEN current <= S2;
25                             ELSE current <= S3; END IF;
26                         WHEN S3 =>
27                             IF a='1' THEN current <= S1;
28                             ELSE current <= S0; END IF;
29                         WHEN OTHERS => current <= S0;
30                 END CASE;
31             END IF;
32         END IF;
33     END PROCESS;
34     w <= '1' WHEN current = S3 ELSE '0';
35 END ARCHITECTURE procedural;
```




- Declare *current* as the register that holds the FSM state; two bits
- User a process statement for transitions
- Use a signal assignment for the output
- Can use a process statement for the output if output is more complex

# State Machines

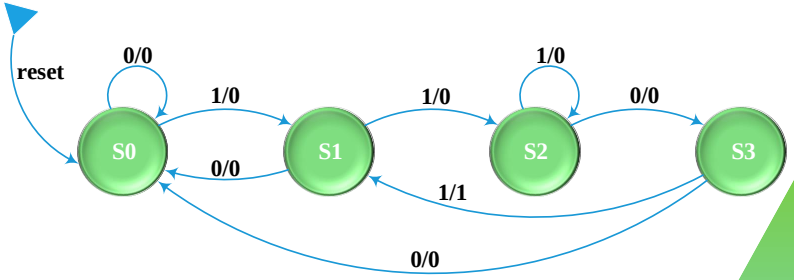
- **A Mealy Machine 1101 Sequence Detector**
- Output is issued on the edges
- Use the same coding style



# State Machines

- **A Mealy Machine 1101 Sequence Detector**

```
1  LIBRARY IEEE;
2  USE IEEE.STD_LOGIC_1164.ALL;
3
4  ENTITY detector1101 IS
5      PORT (a, clk, reset : IN std_logic; w : OUT std_logic);
6  END ENTITY;
7
8  ARCHITECTURE procedural OF detector1101 IS
9      TYPE state IS (S0, S1, S2, S3);
10     SIGNAL current : state := S0;
11 BEGIN
12     PROCESS (clk) BEGIN
13         IF (clk = '0' AND clk'EVENT) THEN
14             IF reset = '1' THEN current <= S0;
15             ELSE
16                 CASE current IS
17                     WHEN S0 =>
18                         IF a='1' THEN current <= S1;
19                         ELSE current <= S0; END IF;
```



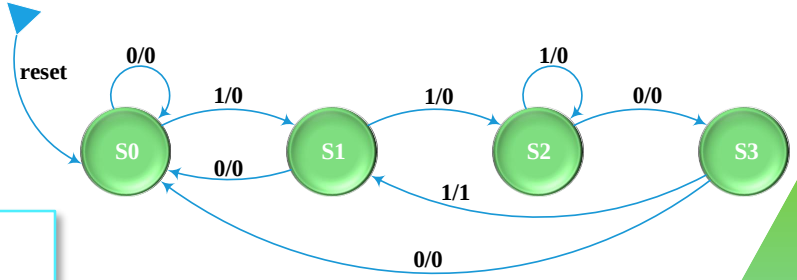

- Use the same style as registers and counters
- Use a cocked process
- Synchronous/Asynchronous control rules are applied
- A separate concurrent process is used for output

# State Machines

- **A Mealy Machine 1101 Sequence Detector**

```
1  LIBRARY IEEE;
2  USE IEEE.STD_LOGIC_1164.ALL;
3
4  ENTITY detector1101 IS
5      PORT (a, clk, reset : IN std_logic; w : OUT std_logic);
6  END ENTITY;
7
8
9
10
11
12
13
14
15
16
17
18
19
```

```
20                      WHEN S1 =>
21                          IF a='1' THEN current <= S2;
22                          ELSE current <= S0; END IF;
23                      WHEN S2 =>
24                          IF a='1' THEN current <= S2;
25                          ELSE current <= S3; END IF;
26                      WHEN S3 =>
27                          IF a='1' THEN current <= S1;
28                          ELSE current <= S0; END IF;
29                      WHEN OTHERS => current <= S0;
30                  END CASE;
31              END IF;
32          END IF;
33      END PROCESS;
34      w <= '1' WHEN (current = S3 AND a='1') ELSE '0';
35  END ARCHITECTURE procedural;
```




- Use the same style as registers and counters
- Use a cocked process
- Synchronous/Asynchronous control rules are applied
- A separate concurrent process is used for output

# State Machines

- **Mealy vs. Moore**

**Moore Machine**

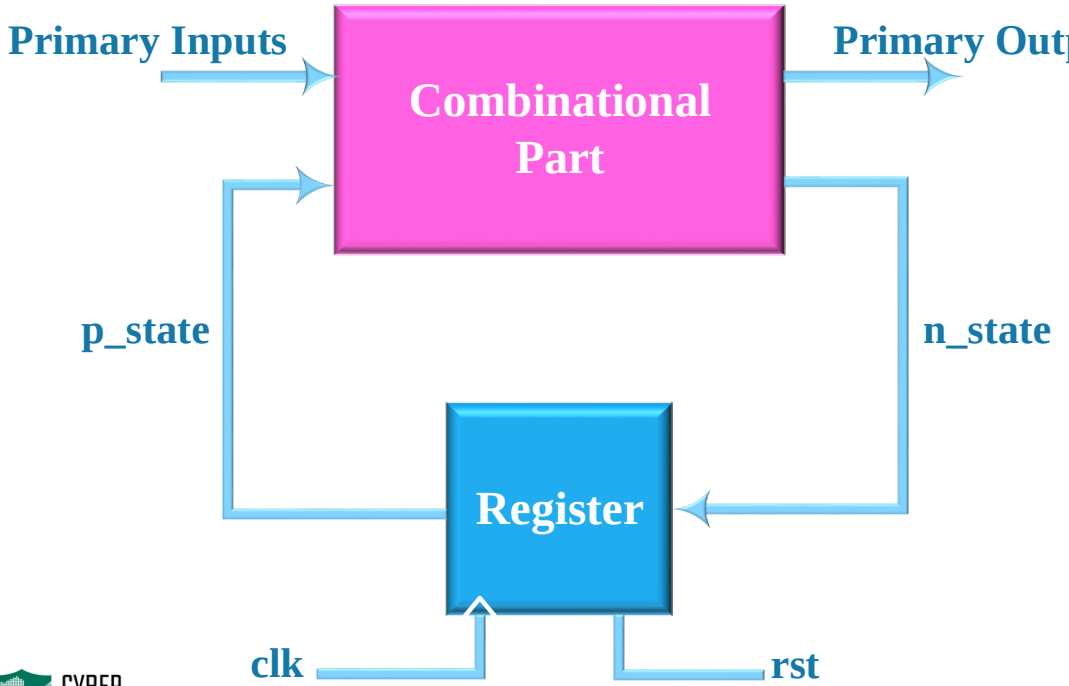
1. Output depends on the present state only
2. Input changes between clock don't propagate to the output
3. Generally, it has more states than Mealy
4. It reacts slower to inputs (One clock cycle later)
5. Output is always synchronous with the clock

**Mealy Machine**

1. Output depends on the present state and present input
2. Synchronous input changes can propagate to the output
3. Generally, it has fewer states than Moore Machine
4. They react faster to inputs
5. Asynchronous output generation

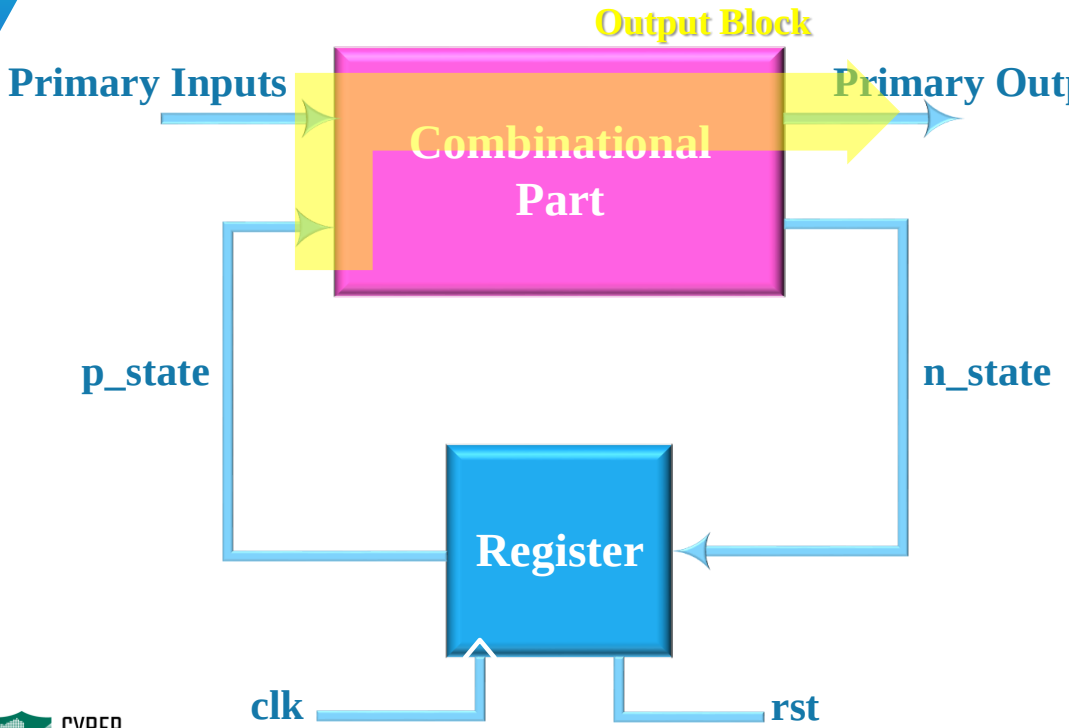
- Don't use mealy unless its input comes from a circuit synchronized with the same clock

# State Machines - Huffman Model



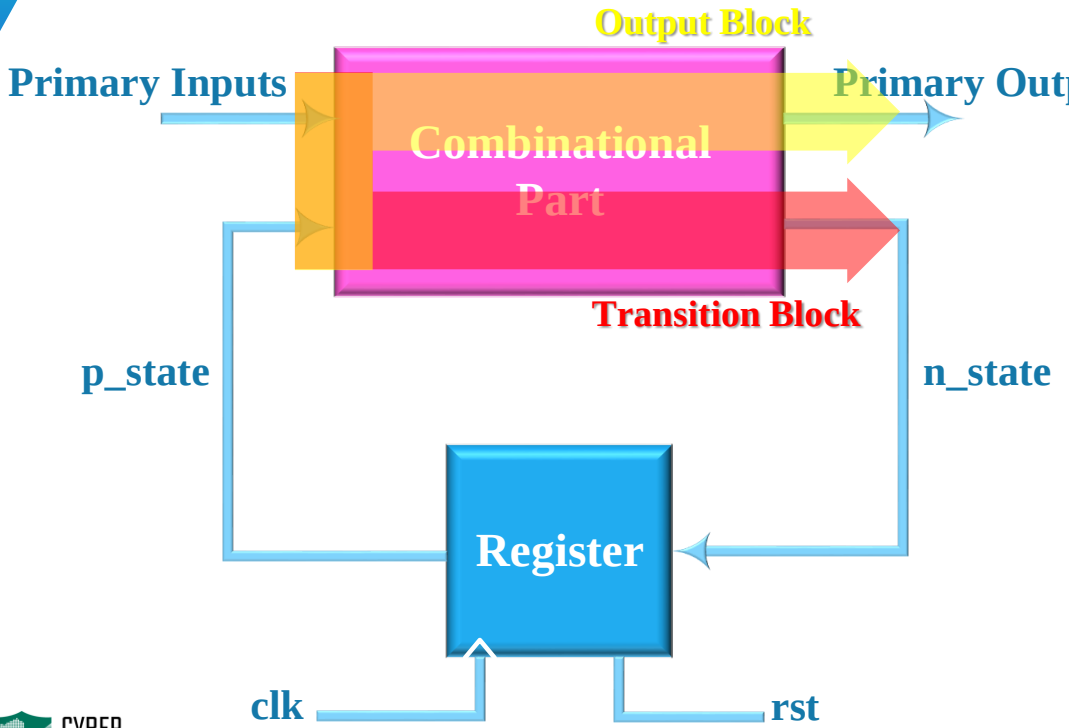

- A better coding style for sequential circuit; separates combinational and sequential parts
- This is called Huffman model
- Alternatively, can further separate outputs from transitions

# State Machines - Huffman Model




- A better coding style for sequential circuit; separates combinational and sequential parts
- This is called Huffman model
- Alternatively, can further separate outputs from transitions

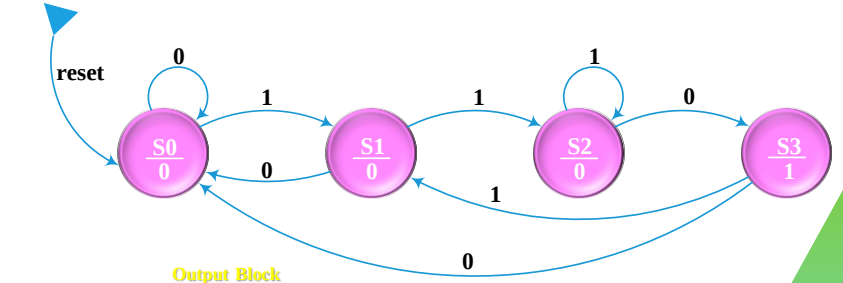
# State Machines - Huffman Model

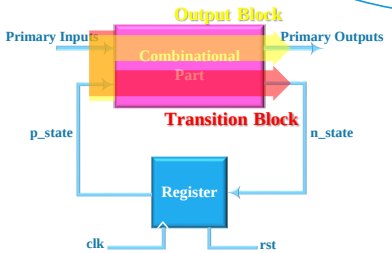



- A better coding style for sequential circuit; separates combinational and sequential parts
- This is called Huffman model
- Alternatively, can further separate outputs from transitions

# Huffman Model

```
1  LIBRARY IEEE;
2  USE IEEE.std_logic_1164.ALL;
3
4  ENTITY moore_detector IS
5      PORT (a, rst, clk : IN std_logic; w : OUT std_logic);
6  END ENTITY ;
7
8  ARCHITECTURE procedural OF moore_detector IS
9      TYPE state IS (S0, S1, S2, S3);
10     SIGNAL p_state, n_state : state;
```




- The same Moore detector using Huffman
- All combinational inputs appear on the sensitivity list
- All combinational outputs receive a value no matter what
- The register block is just a clocked process

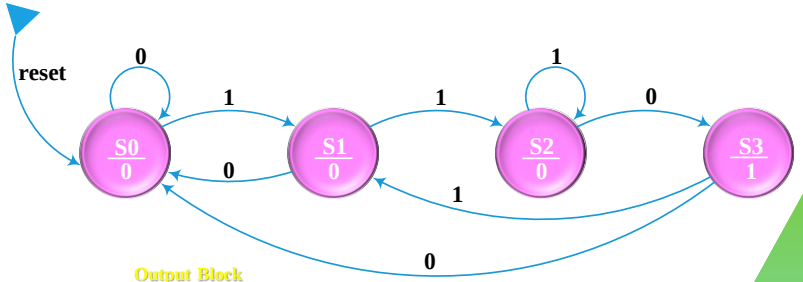
4- Sequential Circuits

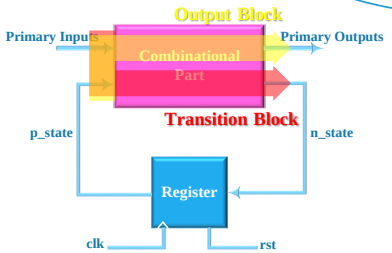
State Machines

# Huffman Model

```
1  LIBRARY IEEE;
2  USE IEEE.std_logic_1164.ALL;
3
4
5
6
7
8
9
10
```

```
13      combinational: PROCESS (p_state, a) BEGIN
14          n_state <= S0;
15          w <= '0';
16          CASE p_state IS
17              WHEN S0 =>
18                  IF a='1' THEN n_state <= S1;
19                  ELSE n_state <= S0; END IF;
20                  w <= '0';
21              WHEN S1 =>
22                  IF a='1' THEN n_state <= S2;
23                  ELSE n_state <= S0; END IF;
24                  w <= '0';
25              WHEN S2 =>
26                  IF a='1' THEN n_state <= S2;
27                  ELSE n_state <= S3; END IF;
28                  w <= '0';
29              WHEN S3 =>
30                  IF a='1' THEN n_state <= S1;
31                  IF a='1' THEN n_state <= S1;
32                  ELSE n_state <= S0; END IF;
33                  w <= '1';
34              WHEN OTHERS => n_state <= S0;
35          END CASE;
36      END PROCESS combinational;
```

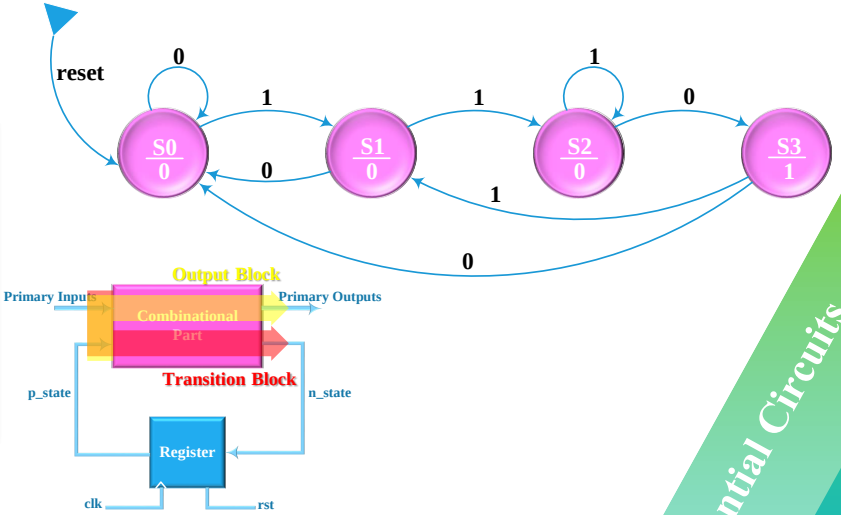



- The same Moore detector using Huffman
- All combinational inputs appear on the sensitivity list
- All combinational outputs receive a value no matter what
- The register block is just a clocked process

# Huffman Model

```
1  LIBRARY IEEE;
2  USE IEEE.std_logic_1164.ALL;
3
4  ENTITY Huffman IS
5      PORT (clk, rst, a: IN BIT; w: OUT BIT);
6  END Huffman;
7
8  ARCHITECTURE Huffman OF Huffman IS
9      TYPE state IS (S0, S1, S2, S3);
10     SIGNAL p_state, n_state: state;
11
12 BEGIN
13     combinational: PROCESS (p_state, a) BEGIN
14         n_state <= S0;
15
16     sequential: PROCESS (clk) BEGIN
17         IF (clk = '1' AND clk'EVENT) THEN
18             IF rst = '1' THEN
19                 p_state <= S0;
20             ELSE
21                 p_state <= n_state;
22             END IF;
23         END IF;
24     END PROCESS sequential;
25
26 END ARCHITECTURE;
27
28             w <= '0';
29         WHEN S3 =>
30             IF a='1' THEN n_state <= S1;
31             IF a='1' THEN n_state <= S1;
32             ELSE n_state <= S0; END IF;
33             w <= '1';
34         WHEN OTHERS => n_state <= S0;
35     END CASE;
36 END PROCESS combinational;
```

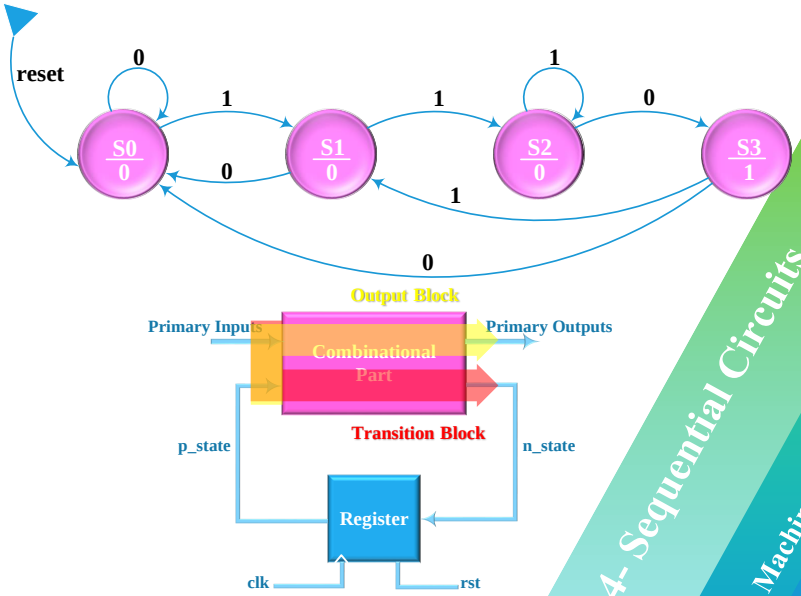



- The same Moore detector using Huffman
- All combinational inputs appear on the sensitivity list
- All combinational outputs receive a value no matter what
- The register block is just a clocked process

# Huffman Model

- A More Modular Moore State Machine Style for 110 Sequence Detector

```
1  LIBRARY IEEE;
2  USE IEEE.std_logic_1164.ALL;
3
4  ENTITY moore_modular IS
5      PORT (a, clk, rst : IN std_logic; w : OUT std_logic);
6  END ENTITY moore_modular;
7
8  ARCHITECTURE procedural OF moore_modular IS
9      TYPE state IS (S0, S1, S2, S3);
10     SIGNAL p_state, n_state : state;
```



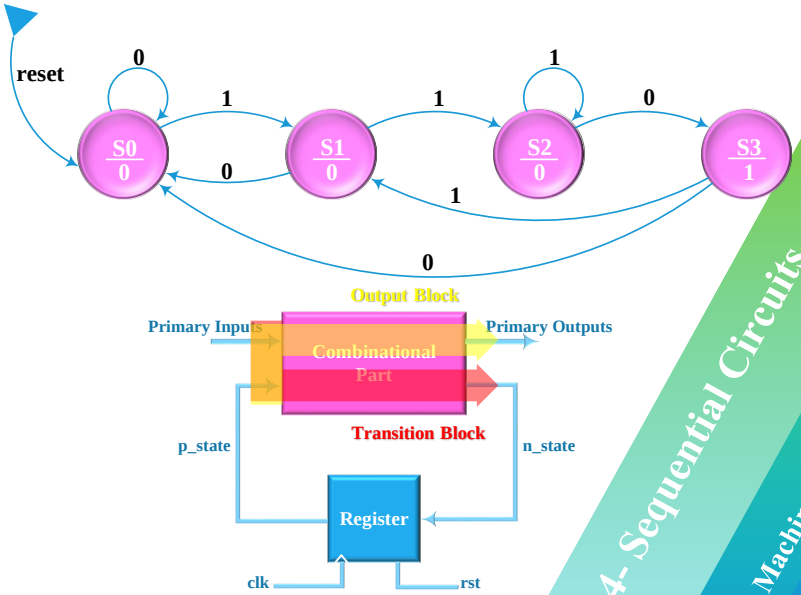

- Separate output from transition

# Huffman Model

- A More Modular Moore State Machine Style for 110 Sequence Detector

```
LIBRARY IEEE;
BEGIN

    PROCESS (p_state, a) BEGIN
        CASE p_state IS
            WHEN S0 =>
                IF a='1' THEN n_state <= S1;
                ELSE n_state <= S0; END IF;
            WHEN S1 =>
                IF a='1' THEN n_state <= S2;
                ELSE n_state <= S0; END IF;
            WHEN S2 =>
                IF a='1' THEN n_state <= S2;
                ELSE n_state <= S3; END IF;
            WHEN S3 =>
                IF a='1' THEN n_state <= S1;
                ELSE n_state <= S0; END IF;
            WHEN OTHERS => n_state <= S0;
        END CASE;
    END PROCESS;
```



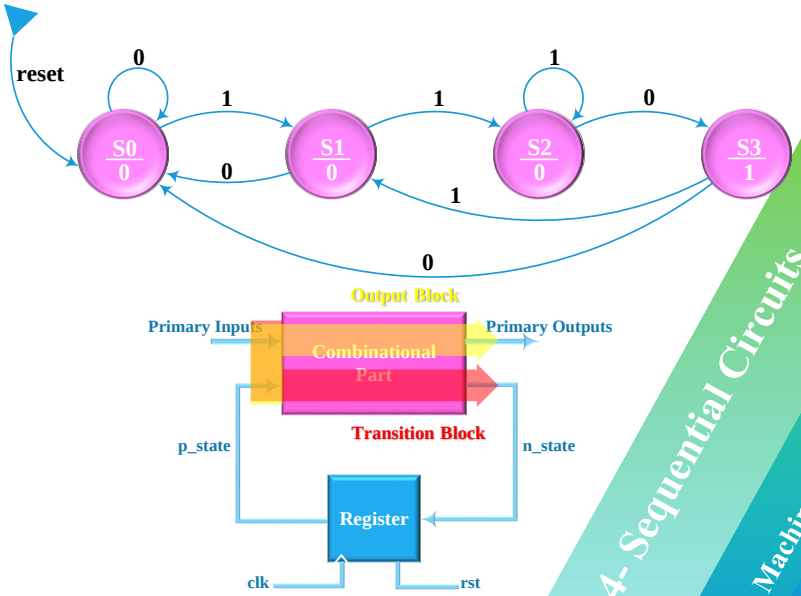

- Separate output from transition

# Huffman Model

- A More Modular Moore State Machine Style for 110 Sequence Detector

```
LIBRARY IEEE;
BEGIN

    PROCESS (p_state, a) BEGIN
        CASE p_state IS
            WHEN S0 =>
                IF a='1' THEN n_state <= S1;
                ELSE n_state <= S0; END IF;
            WHEN S1 =>
                IF a='1' THEN n_state <= S2;
                ELSE n_state <= S0; END IF;
            WHEN S2 =>
                IF a='1' THEN n_state <= S2;
                ELSE n_state <= S3; END IF;
            WHEN S3 =>
                IF a='1' THEN n_state <= S1;
                ELSE n_state <= S0; END IF;
            WHEN OTHERS => n_state <= S0;
        END CASE;
    END PROCESS;
```




- Separate output from transition

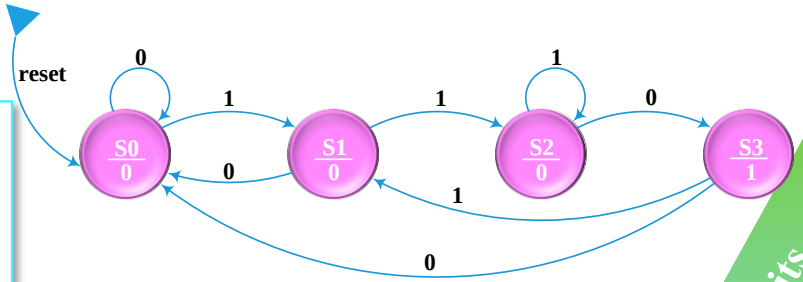
# Huffman Model

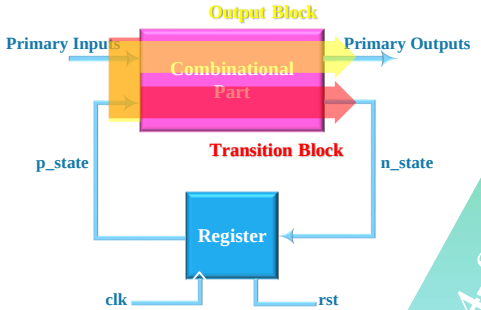
- A More Modular Moore State Machine Style for 110 Sequence Detector

```
LIBRARY IEEE;
BEGIN
    PROCESS (p_state, a) BEGIN
        CASE p_state IS
            WHEN S0 => w <= '0';
            WHEN S1 => w <= '0';
            WHEN S2 => w <= '0';
            WHEN S3 => w <= '1';
            WHEN OTHERS => w <= '0';
        END CASE;
    END PROCESS;

    sequential: PROCESS (clk) BEGIN
        IF (clk = '1' AND clk'EVENT) THEN
            IF rst = '1' THEN
                p_state <= S0;
            ELSE
                p_state <= n_state;
            END IF;
        END IF;
    END PROCESS sequential;

END ARCHITECTURE;
```




- Separate output from transition

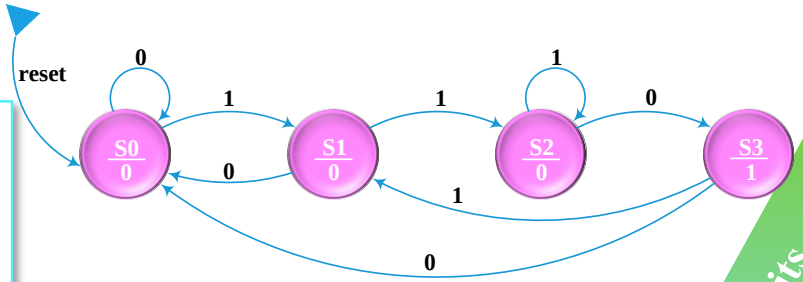
# Huffman Model

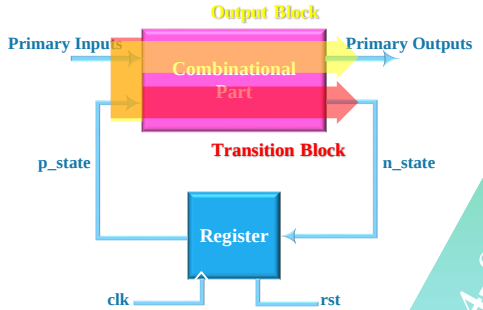
- A More Modular Moore State Machine Style for 110 Sequence Detector

```vhdl
LIBRARY IEEE;
BEGIN
    PROCESS (p_state, a) BEGIN
        CASE p_state IS
            WHEN S0 => w <= '0';
            WHEN S1 => w <= '0';
            WHEN S2 => w <= '0';
            WHEN S3 => w <= '1';
            WHEN OTHERS => w <= '0';
        END CASE;
    END PROCESS;

    sequential: PROCESS (clk) BEGIN
        IF (clk = '1' AND clk'EVENT) THEN
            IF rst = '1' THEN
                p_state <= S0;
            ELSE
                p_state <= n_state;
            END IF;
        END IF;
    END PROCESS sequential;

END ARCHITECTURE;
```




- Separate output from transition