香港中文大學
The Chinese University of Hong Kong

CENG3430 Rapid Prototyping of Digital Systems

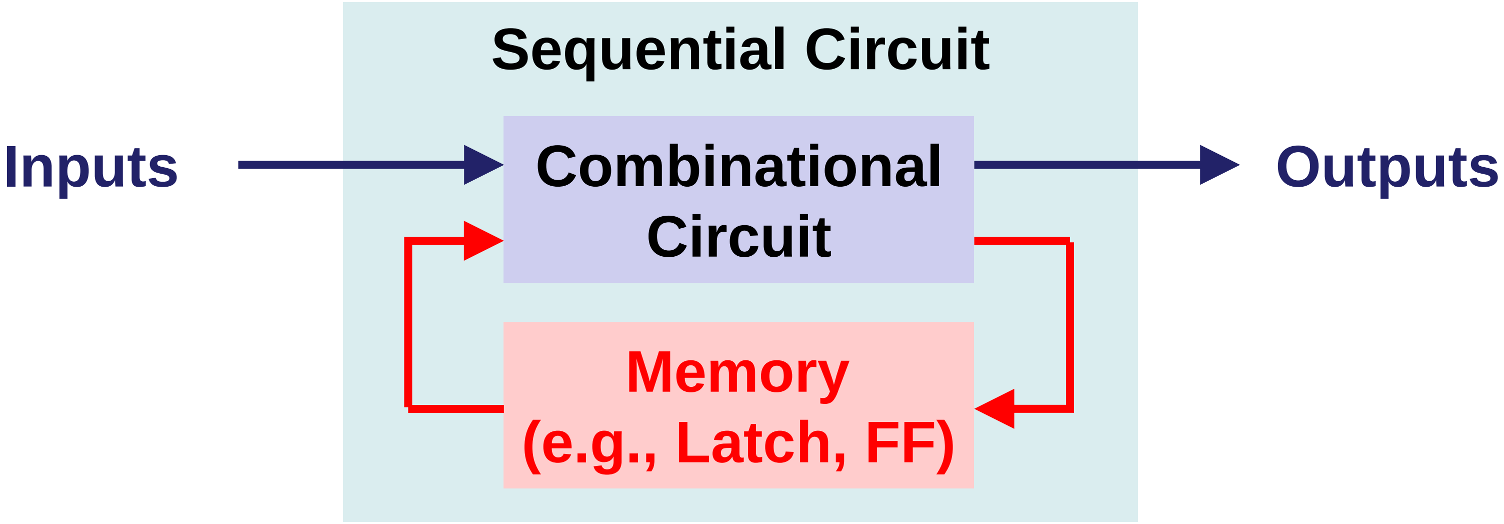
Lecture 04:
Finite State Machine

Ming-Chang YANG

mcyang@cse.cuhk.edu.hk

Recall: Comb. vs. Seq. Circuits (Lec03)

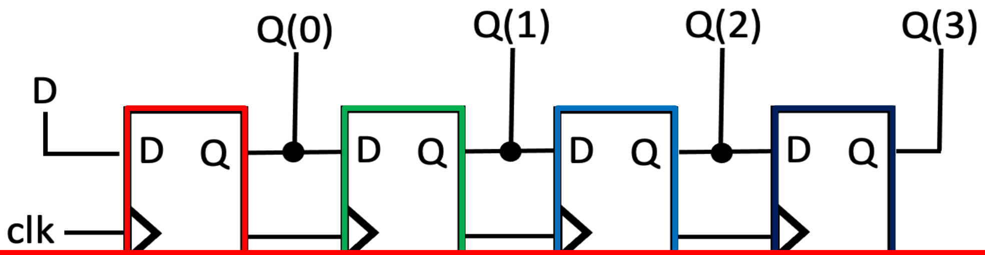
- **Combinational Circuit**: **no memory**
  - Outputs depend on the *present* inputs only.
  - **Rule**: Use **either** concurrent **or** sequential statements.
- **Sequential Circuit**: **has memory**
  - Outputs depend on *present* inputs and *previous* outputs.
  - **Rule**: **MUST** use sequential statements (i.e., `process`) .



Recall: SIPO Shift Register (Lab03)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity SIPO_ASYNC is
  port(D, CLK, RST : IN STD_LOGIC;
       Q : OUT STD_LOGIC_VECTOR(3 downto 0) );
end SIPO_ASYNC;
architecture SIPO_ASYNC_ARCH of SIPO_ASYNC is
component DFF_ASYNC is
  port(D, clk, reset : in STD_LOGIC;
       Q : out STD_LOGIC );
end component;
signal dout : STD_LOGIC_VECTOR(3 downto 0);
begin
  DFF0: DFF_ASYNC port map
        (D, CLK, RST, dout(0));
  DFF1: DFF_ASYNC port map
        (dout(0), CLK, RST, dout(1));
  DFF2: DFF_ASYNC port map
        (dout(1), CLK, RST, dout(2));
  DFF3: DFF_ASYNC port map
        (dout(2), CLK, RST, dout(3));
  Q <= dout;
end SIPO_ASYNC_ARCH;
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity DFF_ASYNC is
  port(D, CLK, RESET: in std_logic;
       Q: out std_logic);
end DFF_ASYNC;
architecture DFF_ASYNC_ARCH of DFF_ASYNC is
begin
  process(CLK, RESET) -- sensitivity list
  begin
    if (RESET = '1') then
      Q <= '0'; -- Reset Q anytime
    elsif CLK = '1' and CLK'event then
      Q <= D; -- Q follows input D
    end if;
  end process;
end DFF_ASYNC_ARCH;
```



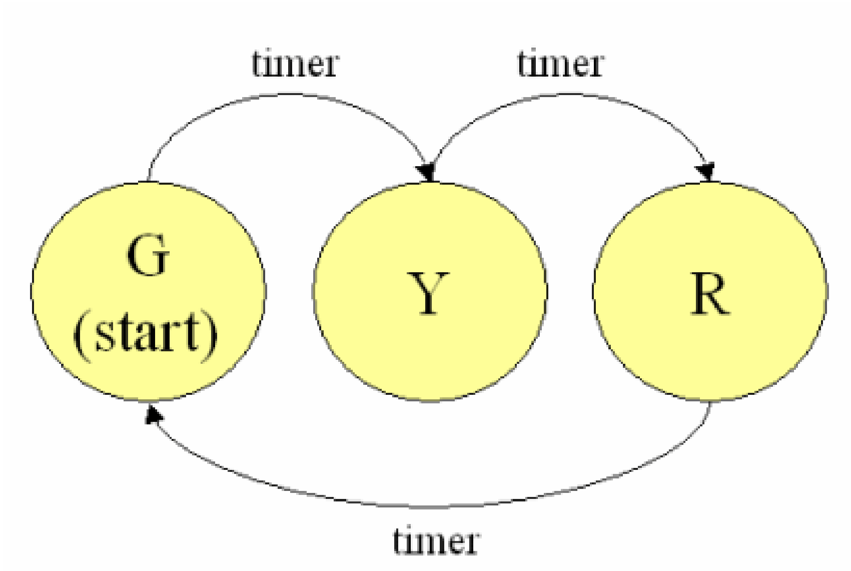
Can we model a sequential circuit in a more “abstract” way?

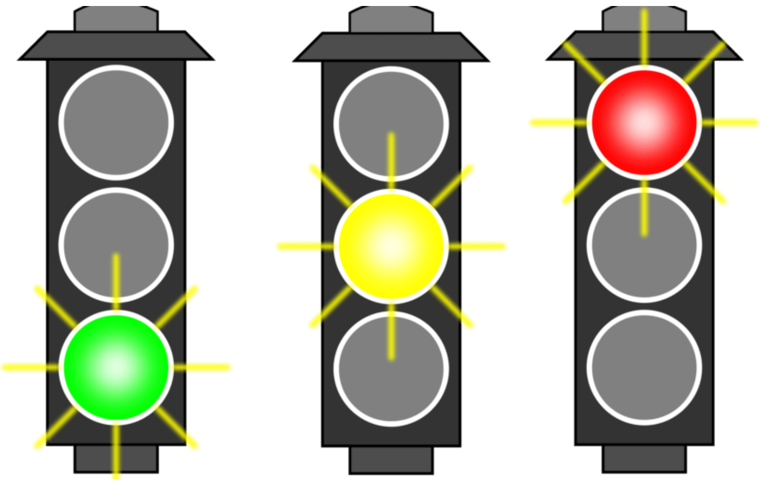
Outline

- Finite State Machine (FSM)
  - Time Controlling
  - State Maintenance
- FSM Types
  - Moore Machine
  - Mealy Machine
- FSM Examples
  - Up/Down Counter
  - Pattern Generator
- Rule of Thumb: FSM Coding Tips

Finite State Machine (FSM)

- **Finite State Machine (FSM)** is an abstract model of a sequential circuit.
  - It jumps from one state to another within a finite pool.
  - Real-life example: Traffic light





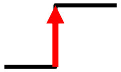
- Two crucial factors of FSM:

  ① **time controlling** and ② **state maintenance**

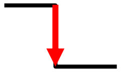
① Time Controlling

- Both "**wait until**" and "**if**" statements can be used to detect the clock edge (e.g., **CLK**):

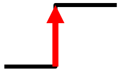
- **"wait until" statement:**
  - `wait until CLK = '1'; -- rising edge`



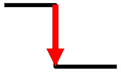
  - `wait until CLK = '0'; -- falling edge`



- **"if" statement:**
  - `if CLK'event and CLK = '1' -- rising edge`

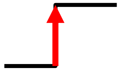


  - `if CLK'event and CLK = '0' -- falling edge`

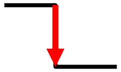


OR

  - `if( rising_edge(CLK) )  -- rising edge`



  - `if( falling_edge(CLK) ) -- falling edge`



When to use “wait until” or “if”? (1/2)

- **Synchronous Process**: Computes values only on clock edges (i.e., only sensitive/sync. to clock signal).
  - **Rule:** Use “wait-until” or “if” for **synchronous** process:

Usage of “wait until”

```
process ← NO sensitivity list implies that there is one clock signal.
begin
   wait until clk='1';← The first statement must be wait until.
   …
end process
```

*Note: IEEE VHDL requires that a process with a wait statement must not have a sensitivity list, and the **first statement** must be **wait until**.*

Usage of “if”

```
process (clk)← The clock signal must be in the sensitivity list.
begin
   …
   if( rising_edge(clk) )← NOT necessary to be the first line.
   …
end process
```

When to use “wait until” or “if”? (2/2)

- **Asynchronous Process**: Computes values on clock edges or when asynchronous conditions are TRUE.
  - That is, it must be sensitive to the clock signal (if any), and to all inputs that may affect the asynchronous behavior.
  - **Rule:** Only use “if” for **asynchronous** process:

Usage of “if”

```
process (clk, input_a, input_b, …)
begin
   …
   if( rising_edge(clk) )
   …
end process
```

← It should include the clock signal and all inputs that involve.

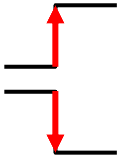
← The clock edge detection doesn’t have to be the first statement of this process.

Simply use “if” statement for both sync/async process!

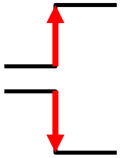
`CLK'event` vs. `rising_edge(CLK)` (1/2)

- Both “`wait until`” and “`if`” statements can be used to detect the clock edge (e.g., `CLK`):

- “`wait until`” statement:
  - `wait until CLK = '1'; -- rising edge`
  - `wait until CLK = '0'; -- falling edge`

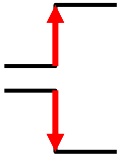


- “`if`” statement:
  - `if CLK'event and CLK = '1' -- rising edge`
  - `if CLK'event and CLK = '0' -- falling edge`



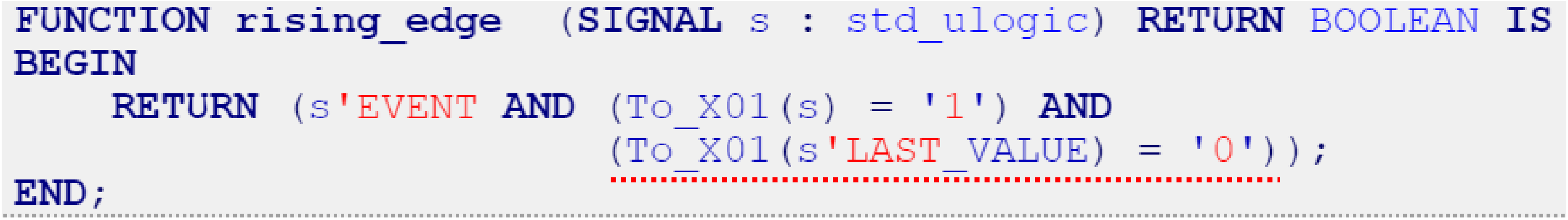
OR

  - `if( rising_edge(CLK) )   -- rising edge`
  - `if( falling_edge(CLK) ) -- falling edge`



CLK'event vs. rising_edge(CLK) (2/2)

- **rising_edge()** function in std_logic_1164 library



```
FUNCTION rising_edge  (SIGNAL s : std_ulogic) RETURN BOOLEAN IS
BEGIN
    RETURN (s'EVENT AND (To_X01(s) = '1') AND
                        (To_X01(s'LAST_VALUE) = '0'));
END;
```

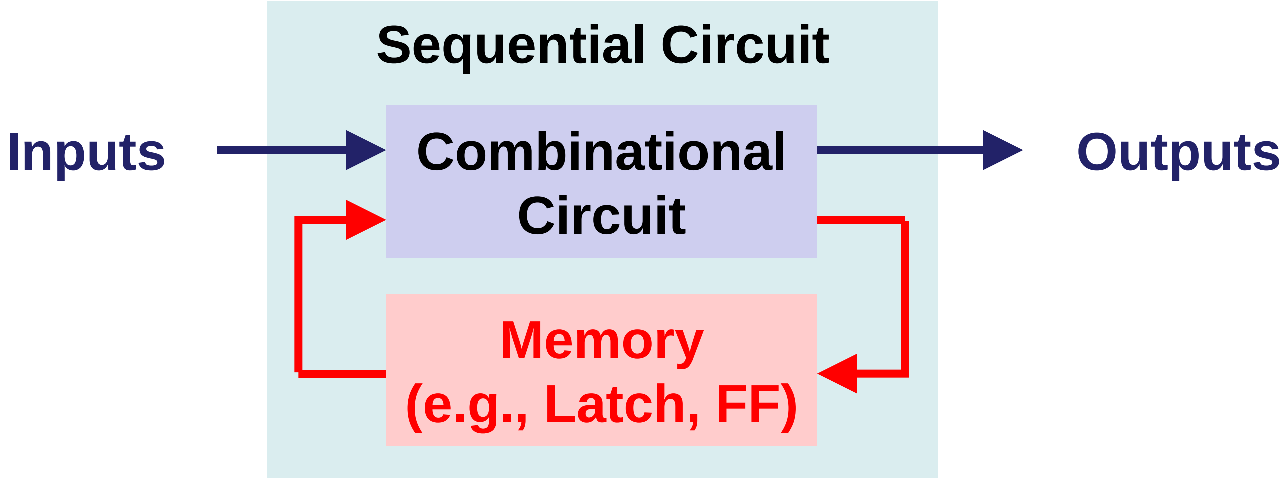
  - It results **TRUE** when there is an edge transition in the signal **s**, the present value is '**1**' and the last value is '**0**'.
  - If the last value is something like '**Z**' or '**U**', it returns a **FALSE**.

- The statement (**clk'event and clk='1'**)
  - It results **TRUE** when the there is an edge transition in the **clk** and the present value is '**1**'.
  - *It does not see whether the last value is '0' or not.*

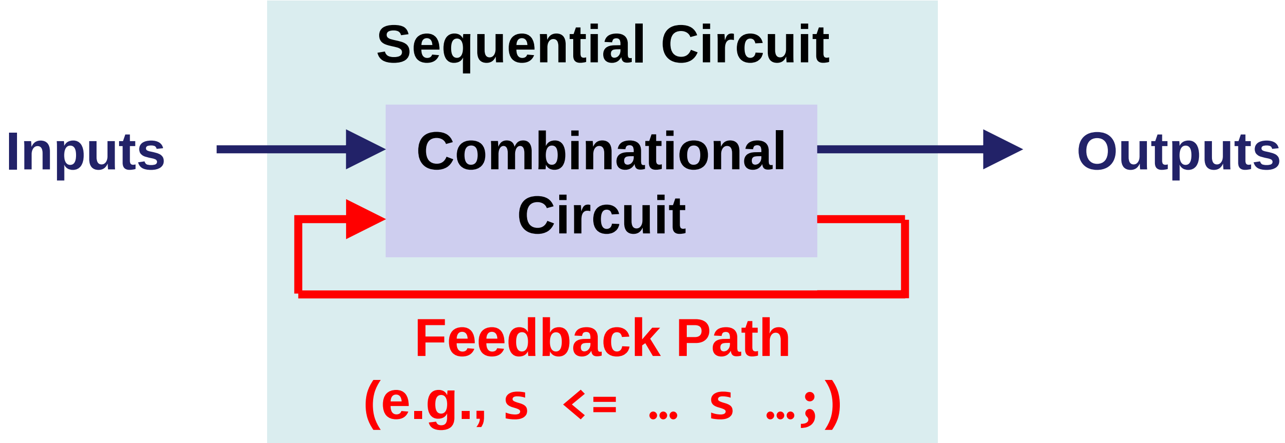
Suggested to use rising/falling_edge() with "if"!

② State Maintenance (1/2)

- **Method 1**: Use **"memory device(s)"** (e.g., FF)



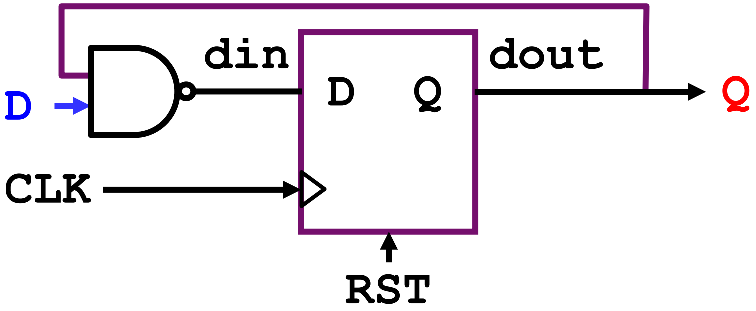
- **Method 2**: Form **"feedback path(s)"** in a clocked process (i.e., sensitive to the clock signal)



✓ **More abstract & convenient!**

② State Maintenance (2/2)

```
entity Method_1 is -- use D-FF
  port(D, CLK, RST : IN STD_LOGIC;
       Q : OUT STD_LOGIC );
end Method_1;
architecture Arch of Method_1 is
component DFF_ASYNC is
  port(D, clk, reset : in STD_LOGIC;
       Q : out STD_LOGIC );
end component;
signal din, dout: STD_LOGIC;
begin
  din <= not ( D and dout );
  DFF_ASYNC port map(din,CLK,RST,dout);
  Q <= dout; -- output
end Arch;
```



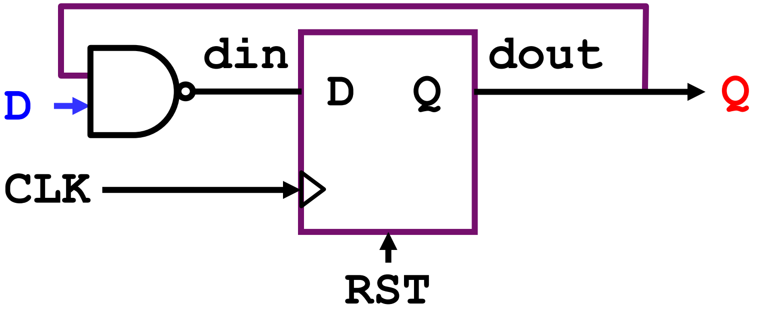
```
entity Method_2 is -- form feedback path
   ... -- same as Method_1
end Method_2;
architecture Arch of Method_2 is
signal s: STD_LOGIC; -- state
begin
  process(CLK, RST) -- clocked process
  begin
    if (RST = '1') then
      s <= '0'; -- async. Reset
    elsif rising_edge(CLK) then
      s <= not ( D and s ); -- feedback
    end if;
  end process;
  Q <= s; -- output
end Arch;
```

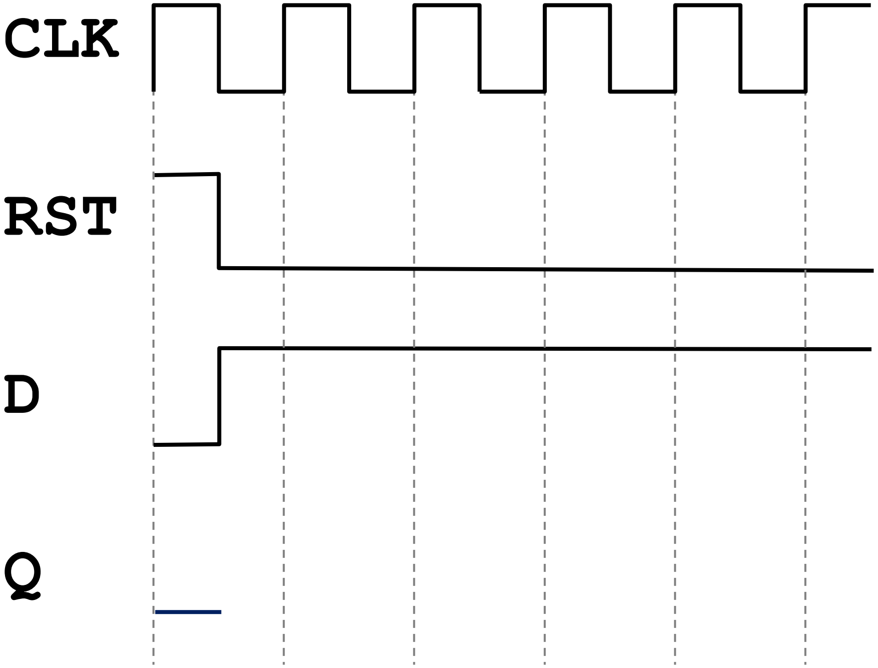
Signal **s** (i.e., state) forms a **feedback path** in a clocked process!

- The value of **s** will last for one clock cycle.
  - i.e., not(D and **s**) will takes effect at the next edge.
- **<=** here can be treated as a flip-flop!

Class Exercise 4.1

- Determine the signal **Q**:





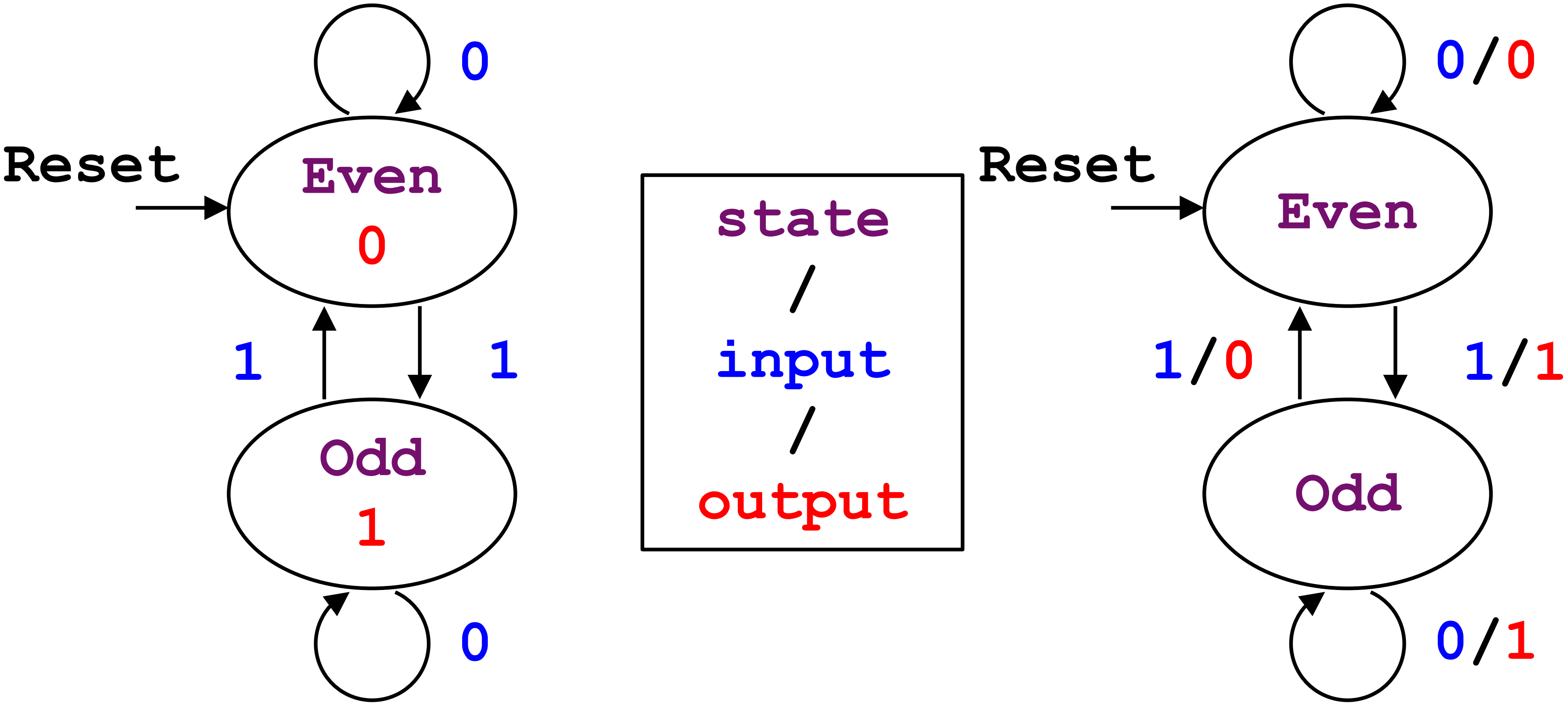
```
entity Method_2 is -- form feedback path
  port(D, CLK, RST : IN STD_LOGIC;
        Q : OUT STD_LOGIC );
end Method_2;
architecture Arch of Method_2 is
signal s: STD_LOGIC; -- state
begin
  process(CLK, RST) -- clocked process
  begin
    if (RST = '1') then
      s <= '0'; -- async. Reset
    elsif rising_edge(CLK) then
      s <= not ( D and s ); -- feedback
    end if;
  end process;
  Q <= s; -- output
end Arch;
```

Outline

- Finite State Machine (FSM)
  - Time Controlling
  - State Maintenance
- FSM Types
  - Moore Machine
  - Mealy Machine
- FSM Examples
  - Up/Down Counter
  - Pattern Generator
- Rule of Thumb: FSM Coding Tips

FSM Types

- **Moore Machine**:
  - – Outputs rely on the present state *only*.
- **Mealy Machine**:
  - – Outputs rely on both the present state *and* inputs.

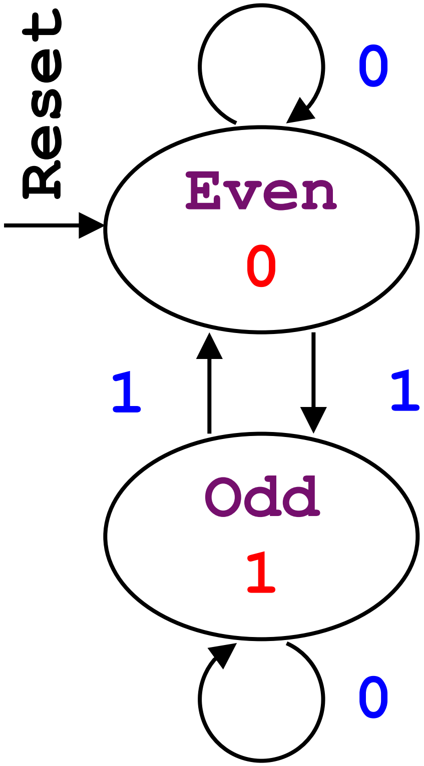


Example: An FSM that outputs a ‘0’ (resp. to ‘1’) if an even (resp. to odd) number of 1’s have been received.

https://www.slideshare.net/mirfanjum1/moore-and-mealy-machines-29553482

Moore Machine

- **Moore Machine**: *Outputs rely on present state only.*



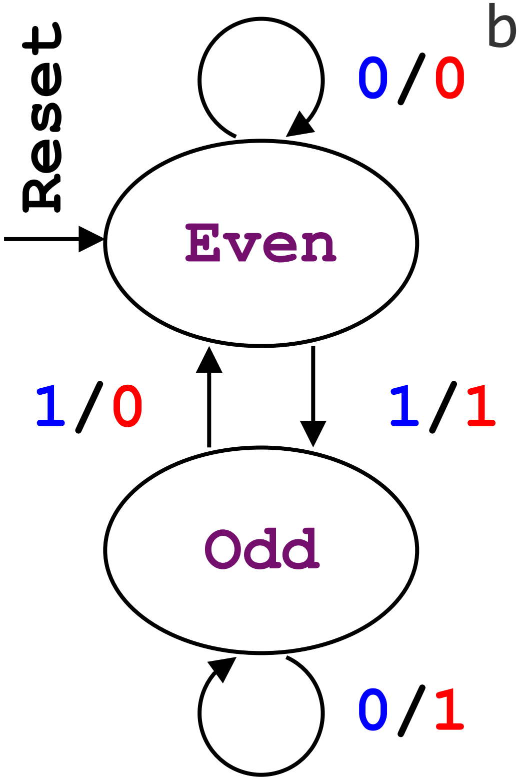
```
architecture moore_arch of fsm is
signal s: std_logic := ‘0’; -- internal state
begin
   process (s)
   begin
     OUTX <= s; -- output
   end process;
   process (CLOCK, RESET)
   begin
     if RESET = '1' then s <= '0';
     elsif rising_edge(CLOCK) then
       s <= INX xor s; -- feedback
     end if;
   end process;
end moore_arch;
```

Combinational Logic

Sequential Logic

Mealy Machine

- **Mealy Machine**: *Outputs rely on both state and inputs.*



```
architecture mealy_arch of fsm is
signal s: std_logic := ‘0’; -- internal state
begin
  process (s, INX)
  begin
    OUTX <= INX xor s; -- output
  end process;
  process (CLOCK, RESET)
  begin
    if RESET = '1' then s <= '0';
    elsif rising_edge(CLOCK) then
      s <= INX xor s; -- feedback
    end if;
  end process;
end mealy_arch;
```

Combinational Logic

Sequential Logic

Outline

- Finite State Machine (FSM)
  - Time Controlling
  - State Maintenance
- FSM Types
  - Moore Machine
  - Mealy Machine
- FSM Examples
  - Up/Down Counter
  - Pattern Generator
- Rule of Thumb: FSM Coding Tips

FSM Example 1) Up/Down Counter (1/3)

- **Up/Down Counter**: Generates a sequence of up/down counting patterns.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.Numeric_Std.ALL;
entity counter is
  port(
  CLK: in std_logic;
  RESET: in std_logic;
  COUNT: out std_logic_vector
         (3 downto 0) );
end counter;
architecture counter_arch of counter is
signal s: std_logic_vector(3 downto 0) ) := “0000”; -- state
begin
```

Sequential Logic

```
  process(CLK, RESET)
  begin
    if(RESET = '1') then s <= “0000”;
    else
      if( rising_edge(CLK) ) then
        s <= std_logic_vector(
        unsigned(s)+1); -- feedback
      end if;
    end if;
  end process;
```

Combinational Logic

```
  COUNT <= s; -- Moore or Mealy?
```

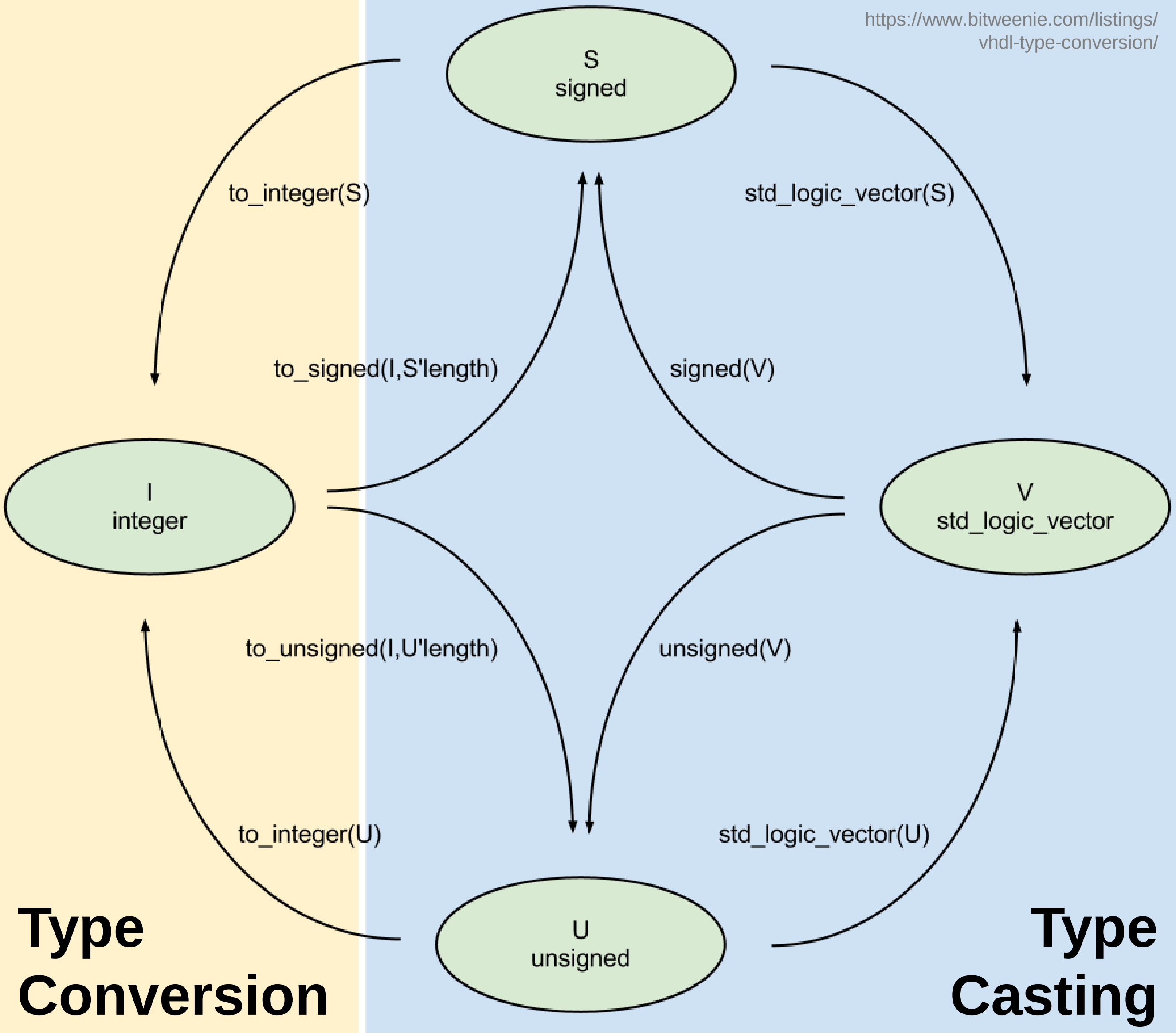
```
end counter_arch;
```

FSM Example 1) Up/Down Counter (2/3)

```
use IEEE.Numeric_Std.ALL;
signal s: std_logic_vector(3 downto 0)) := “0000”; -- state
s <= std_logic_vector(unsigned(s)+1); -- feedback
```

- A std_logic_vector is merely a collection of std_logic.
  - The individual positions have no predefined meaning.
- The IEEE NUMERIC_STD package includes overloading functions for data types that are more convenient to use.
  - Such as unsigned/signed types and integer type.
- VHDL is a strongly-typed language.
  - Signals of different types CANNOT be assigned to each other without using type casting/conversion.

FSM Example 1) Up/Down Counter (3/3)



Remember to "`use IEEE.Numeric_Std.ALL`"!

Class Exercise 4.2

- Complete the counter FSM by filling in the missing line if the state is declared as an unsigned type:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.Numeric_Std.ALL;
entity counter is
  port(
  CLK: in std_logic;
  RESET: in std_logic;
  COUNT: out std_logic_vector
         (3 downto 0) );
end counter;
architecture counter_arch of counter
is
signal s: unsigned(3 downto 0) :=
“0000”; -- state
```

```
begin
  process(CLK, RESET)
  begin
    if(RESET = '1') then s <= “0000”;
    else
      if( rising_edge(CLK) ) then
        s <= s + 1; -- feedback
      end if;
    end if;
  end process;
```

Sequential Logic

Combinational Logic

  ____

```
end counter_arch;
```

Class Exercise 4.3

- Complete the counter FSM by filling in the missing line if the state is declared as an integer type:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.Numeric_Std.ALL;
entity counter is
  port(
  CLK: in std_logic;
  RESET: in std_logic;
  COUNT: out std_logic_vector
         (3 downto 0) );
end counter;
architecture counter_arch of counter
is
signal s: integer range 0 to 15
          := 0; -- state
```

```
begin
```

Sequential Logic

```
  process(CLK, RESET)
  begin
    if(RESET = '1') then s <= “0000”;
    else
      if( rising_edge(CLK) ) then
        s <= s + 1; -- feedback
      end if;
    end if;
  end process;
```

Combinational Logic

```

  __________________________________
end counter_arch;
```

Integer Type

- An integer type can be defined with or without specifying a range.
  - If a range is not specified, VHDL allows integers to have a minimum rage of

$$-2{,}147{,}483{,}647\ to\ 2{,}147{,}483{,}647$$

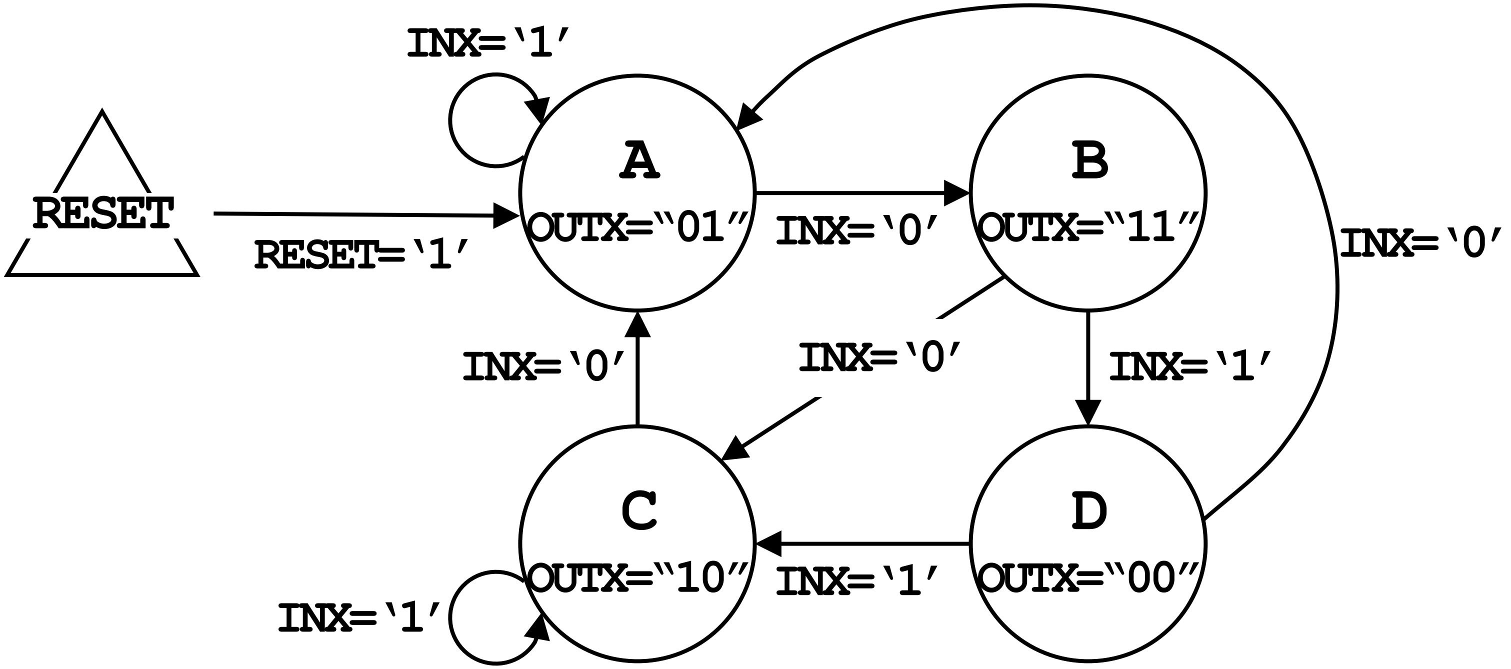
$$-(2^{31}-1)\ to\ (231-1)$$

  - Or a range can be specified, e.g.,

```
signal int: integer range 0 to 255;
```

FSM Example 2) Pattern Generator (1/2)

- **Pattern Generator**: Generates any pattern we want.
- Given the following machine of 4 states: **A**, **B**, **C** and **D**.



  - The machine has an asynchronous **RESET**, a clock signal **CLK**, and a 1-bit synchronous input signal **INX**.
  - The machine also has a 2-bit output signal **OUTX**.

FSM Example 2) Pattern Generator (2/2)

```
library IEEE;
use IEEE.std_logic_1164.all;
entity pat_gen is port(
RESET,CLOCK,INX: in STD_LOGIC;
OUTX: out STD_LOGIC_VECTOR(1
downto 0));
end pat_gen;
architecture arch of pat_gen is
type state_type is (A,B,C,D);
signal s: state_type; -- state
```

Sequential Logic

```
begin
process(CLOCK, RESET)
begin
  if RESET = '1' then
    s <= A;
  elsif rising_edge(CLOCK) then
    -- feedback
    case s is
    when A =>
      if INX = '1' then s <= A;
      else s <= B; end if;
    when B =>
      if INX = '1' then s <= D;
      else s <= C; end if;
    when C =>
      if INX = '1' then s <= C;
      else s <= A; end if;
    when D =>
      if INX = '1' then s <= C;
      else s <= A; end if;
    end case;
  end if;
end process;
```

Combinational Logic

```
process(s)
begin
  case s is
    when A => OUTX <= "01";
    when B => OUTX <= "11";
    when C => OUTX <= "10";
    when D => OUTX <= "00";
  end case;
end process; -- Moore Machine
end arch;
```

Enumeration Type

- An enumeration type introduces abstraction into circuits by allowing users defining a list of values.
  - Example:

```
type colors is (RED, GREEN, BLUE);
signal my_color: colors;
```

- An enumerated type is ordered.
  - The order in which the values are listed in the type declaration defines their relation:

    Each values is greater than the one to the left,
    and less than the one to the right.

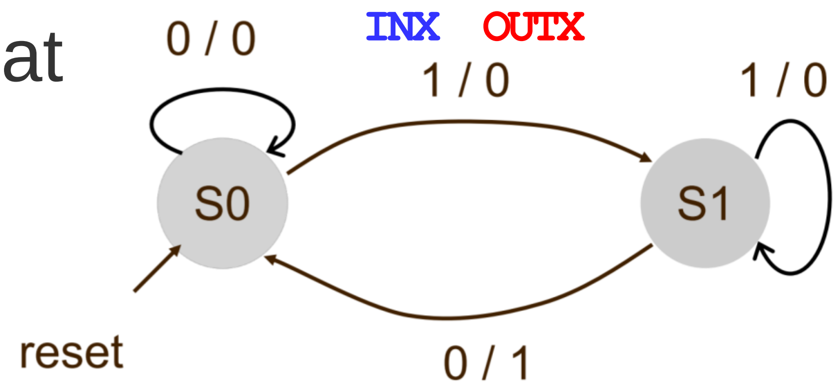
  - Example: a comparison can be:

```
my_color > RED and my_color < BLUE
```

Class Exercise 4.4

Student ID: ____________ Date:
Name: ________________ ________

- Complete the Mealy FSM that recognizes sequence “10”:



```
architecture arch of mealy_fsm is
type state_type is (S0, S1);
signal s: std_logic; -- state
begin
process(CLK, RESET) -- seq
begin
  if(RESET = '1') then s <= S0;
  else
    if( rising_edge(CLK) ) then
      case s is
      when S0 =>
        if INX = '1' then
          s <= S1; -- feedback
        else
          s <= S0; -- feedback
        end if;
      when S1 =>
        if INX = '0' then
          s <= S0; -- feedback
        else
          s <= S1; -- feedback
        end if;
      end case;
    end if;
  end if;
end process;
OUTX <= '1' when(s=__ and INX=__)
        else '0';   -- Mealy
end arch;
```

Outline

- Finite State Machine (FSM)
  - Time Controlling
  - State Maintenance
- FSM Types
  - Moore Machine
  - Mealy Machine
- FSM Examples
  - Up/Down Counter
  - Pattern Generator
- Rule of Thumb: FSM Coding Tips

Rule of Thumb: FSM Coding Tips

① **Maintain/define the internal state(s) explicitly**

② **Separate combinational and sequential logics**

- – Write **at least two processes**: one for combinational logic, and the other for sequential logic
  - Maintain the internal state(s) using a sequential process
  - Drive the output(s) using a combination process

③ **Keep every process as simple as possible**

- – Partition a large process into **multiple small ones**

④ **Put every signal** (*that your process must be sensitive to its changes*) **in the sensitivity list**

⑤ **Avoid assigning a signal from multi-processes**

- – It may cause the "**multi-driven**" issue.

Summary

- Finite State Machine (FSM)
  - Time Controlling
  - State Maintenance

- FSM Types
  - Moore Machine
  - Mealy Machine

- FSM Examples
  - Up/Down Counter
  - Pattern Generator

- Rule of Thumb: FSM Coding Tips