香港中文大學
The Chinese University of Hong Kong

*CENG3430 Rapid Prototyping of Digital Systems*

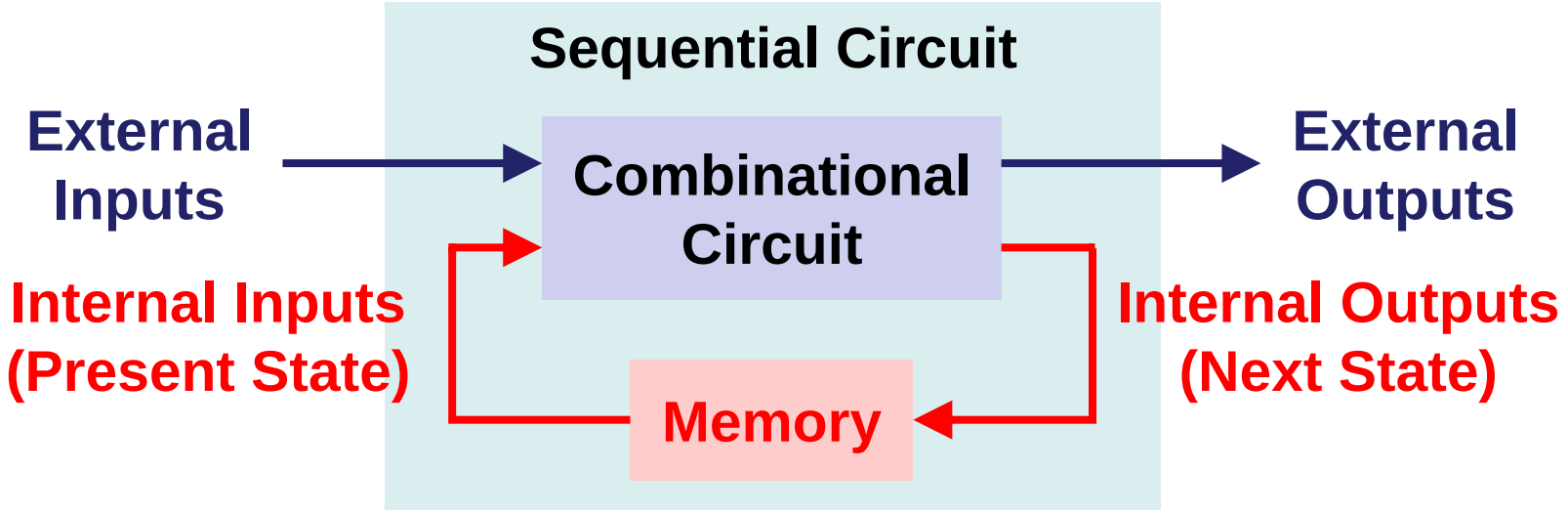
# Lecture 05: Finite State Machine

**Ming-Chang YANG**

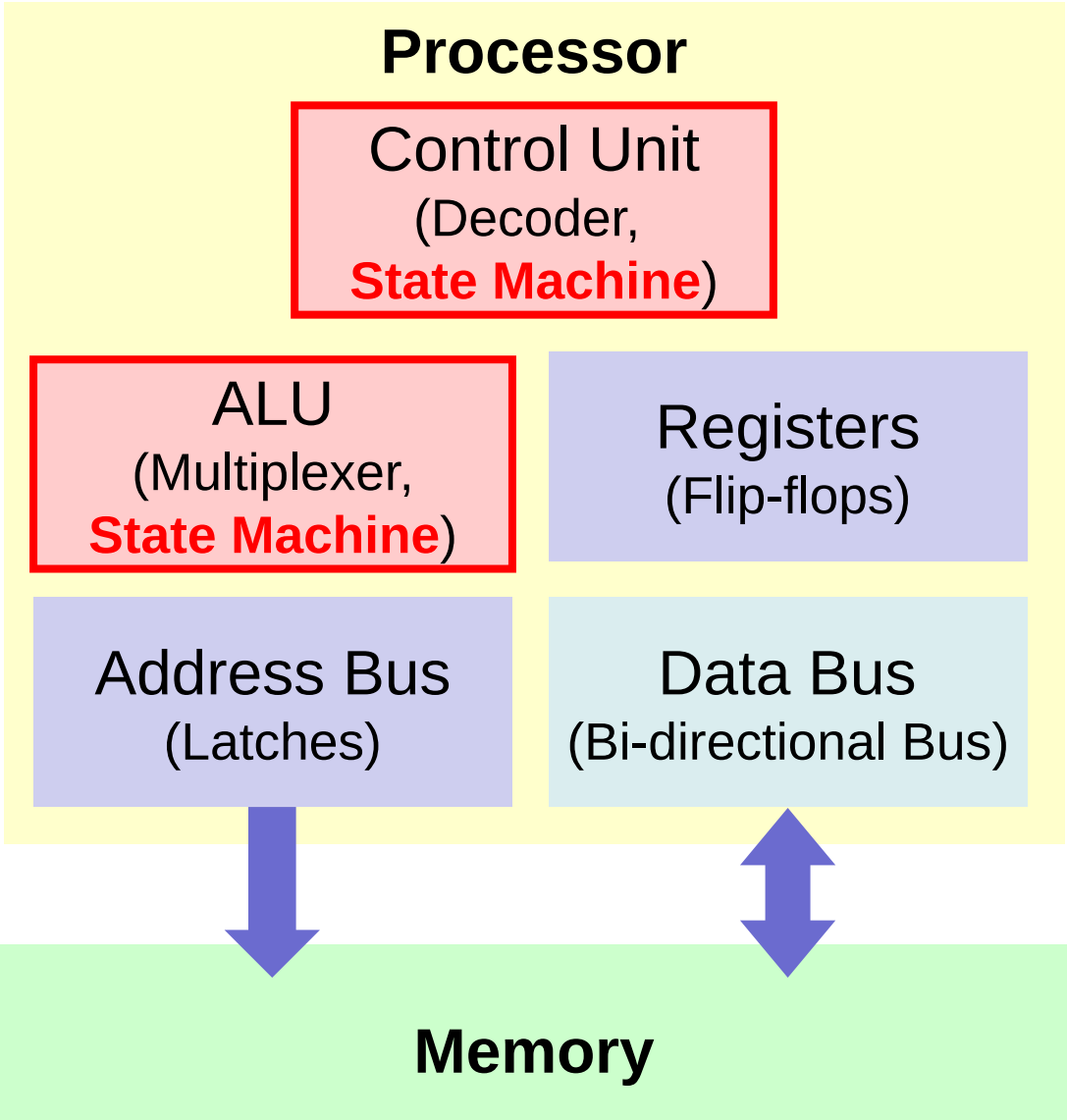
mcyang@cse.cuhk.edu.hk

# Recall: Combinational vs. Sequential

- **Combinational Circuit**: **no memory**
  - ① Outputs are a function of the *present* inputs only.
  - ② **Rule**: Use **either** concurrent **or** sequential statements.
- **Sequential Circuit**: **has memory**
  - ① Outputs are a function of the *present* inputs and the *previous* outputs (i.e., the **internal state**).
  - ② **Rule**: **Must** use sequential (i.e., `process`) statements.



# Recall: Typical Processor Organization



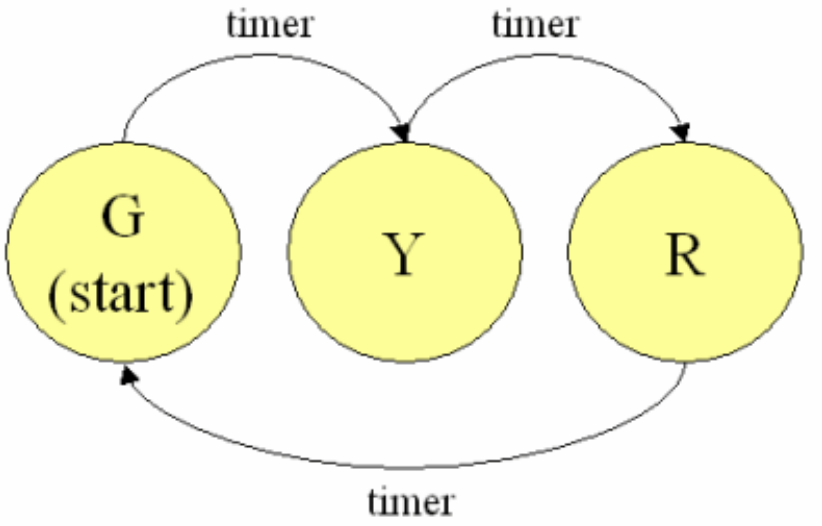

**How to maintain the internal state explicitly?**

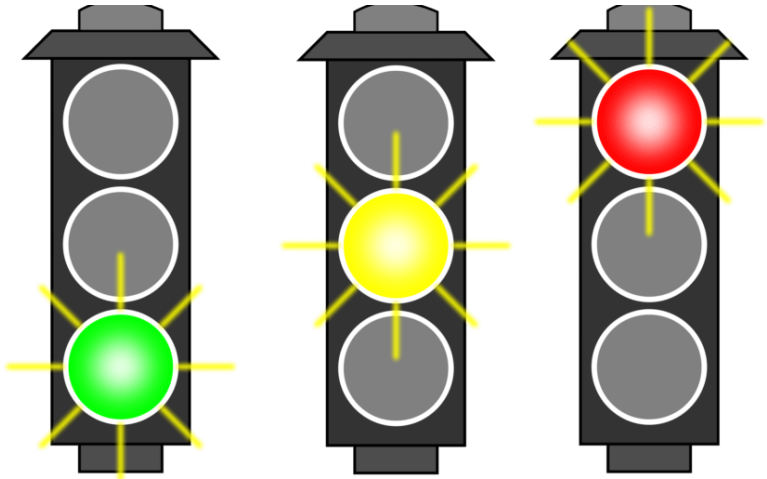
# Outline

- Finite State Machine (FSM)
  - Clock Edge Detection
    - "**`if`**" statement vs. "**`wait until`**" statement
    - **`rising_edge(CLK)`** vs. **`CLK'event`**
  - Direct Feedback Path
- Types of FSM
  - Moore vs. Mealy
- Examples of FSM
  - Up/Down Counter
  - Pattern Generator

# Finite State Machine (FSM)

- **Finite State Machine (FSM)**: A system jumps from one **state** to another:
  - Within a pool of finite states, and
  - Upon clock edges and/or input transitions.
- Example of FSM: traffic light, digital watch, CPU, etc.



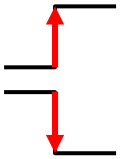
- Two crucial factors: *time (clock edge)* and *state (feedback)*

# Outline

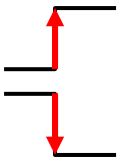
- Finite State Machine (FSM)
  - Clock Edge Detection
    - "**`if`**" statement vs. "**`wait until`**" statement
    - **`rising_edge(CLK)`** vs. **`CLK'event`**
  - Direct Feedback Path
- Types of FSM
  - Moore vs. Mealy
- Examples of FSM
  - Up/Down Counter
  - Pattern Generator

# Clock Edge Detection

- 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`

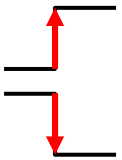


- "**if**" **statement:**
  - `if CLK'event and CLK = '1' -- rising edge`
  - `if CLK'event and CLK = '0' -- falling edge`



OR

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



# rising_edge(CLK) vs. CLK'event

- **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.*

**Use rising_edge() / falling_edge() with "if" statements!**

# 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
```

← The sensitivity list should include the clock signal, and all inputs that may affect asynchronous behavior.

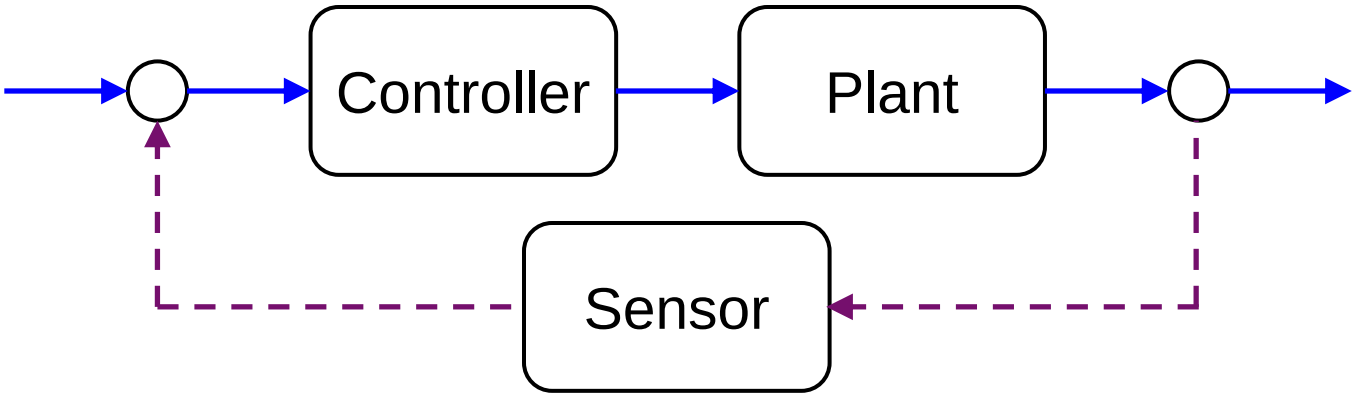
**Simply use "if" statements for both sync. and async. processes!**

# Outline

- Finite State Machine (FSM)
  - Clock Edge Detection
    - "`if`" statement vs. "`wait until`" statement
    - `rising_edge(CLK)` vs. `CLK'event`
  - Direct Feedback Path
- Types of FSM
  - Moore vs. Mealy
- Examples of FSM
  - Up/Down Counter
  - Pattern Generator

# Feed-forward and Feedback Paths

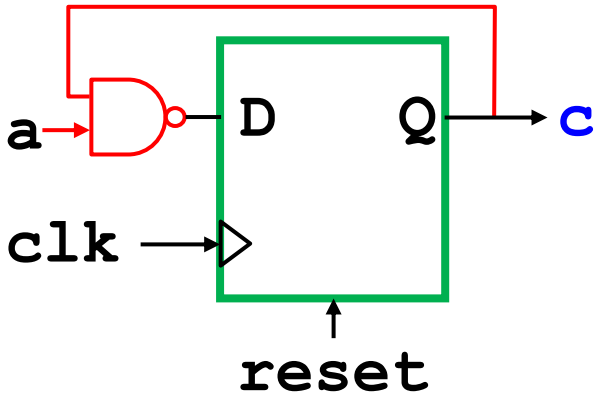
- So far, we mostly focus on logic with feed-forward (or open-loop) paths.




- Now, we are going to learn feedback (or closed-loop) paths—*the key step of making a finite state machine*.

# Direct Feedback Path

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity feedback_1 is
port(a,clk,reset: in std_logic;
     c: buffer std_logic);
end feedback_1;
architecture feedback_1_arch of feedback_1 is
begin
  process(clk, reset) -- async.
  begin
    if reset = '1' then c <= '0';
    elsif rising_edge(clk) then
      c <= not(a and c);
    end if;
  end process;
end feedback_1_arch ;
```

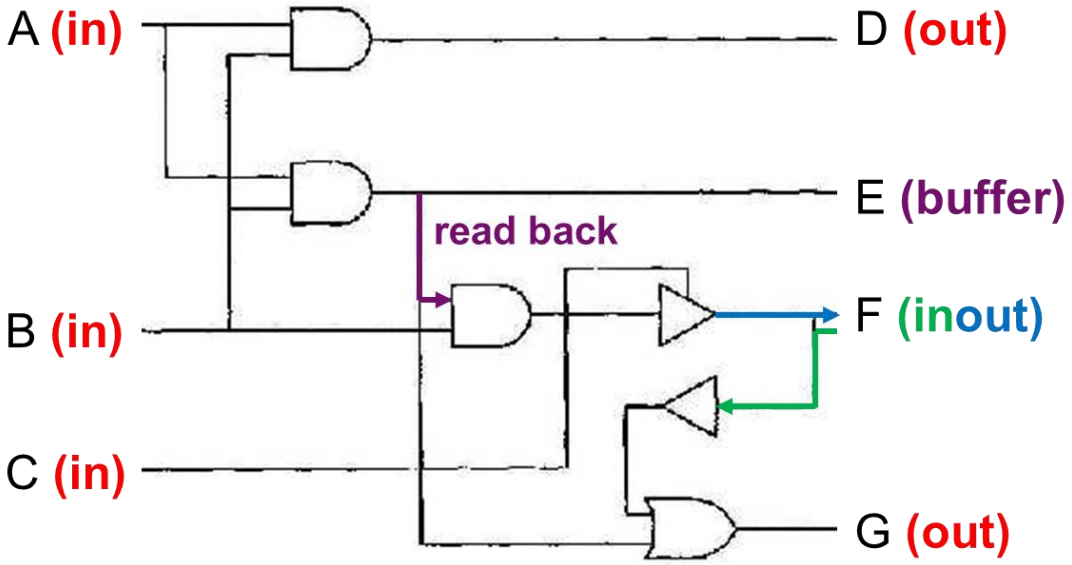



① Signal `c` forms a **closed loop**.
- `not(a and c)` takes effect at the next rising clock edge.
- The current `c` holds for one cycle.

② "`<=`" is like a flip-flop.

# Internal Feedback: inout or buffer

- Recall (*Lec01*): There are 4 modes of I/O pins:
  1) **in**: Data flows **in** only
  2) **out**: Data flows **out** only (cannot be read back by the entity)
  3) inout: Data flows **bi-directionally** (i.e., in or out)
  4) buffer: Similar to **out** but it can be **read back** by the entity



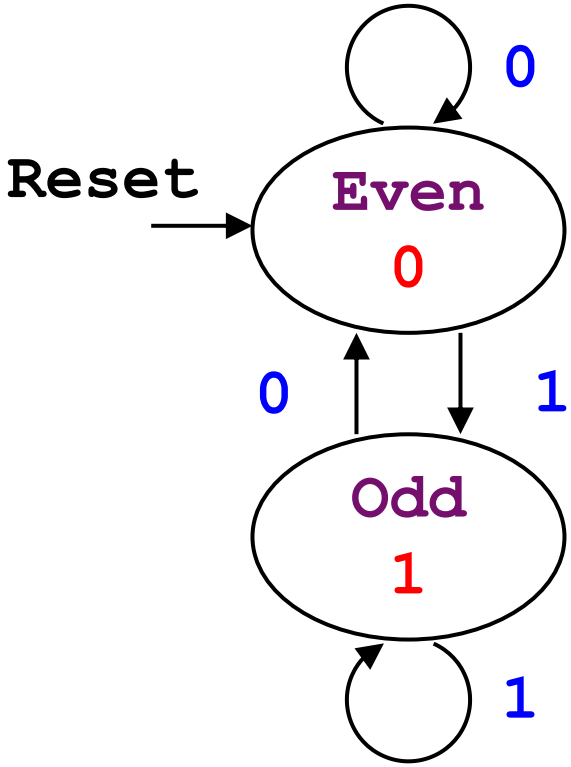

- Both buffer and inout can be read back internally.
  - inout can also read external input signals.

# Outline

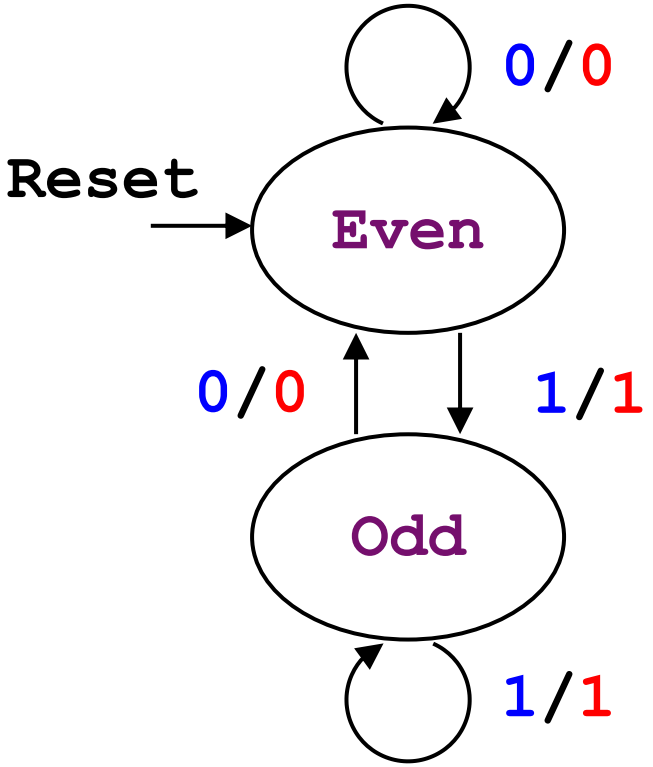
- Finite State Machine (FSM)
  - Clock Edge Detection
    - "`if`" statement vs. "`wait until`" statement
    - `rising_edge(CLK)` vs. `CLK'event`
  - Direct Feedback Path
- Types of FSM
  - Moore vs. Mealy
- Examples of FSM
  - Up/Down Counter
  - Pattern Generator

# Types of Finite State Machines

- **Moore Machine**:
  - Outputs are a function of the present state *only*.
- **Mealy Machine**:
  - Outputs are a function of the present state *and* the present inputs.




State / input / output




**Suggestion: Maintain the internal state explicitly!**

# Moore Machine

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

```
architecture moore_arch of fsm is
signal s: bit; -- internal state
begin
  process (s)                                  Combinational Logic
  begin
    OUTX <= not s; -- output
  end process;
  process (CLOCK, RESET)                       Sequential Logic
  begin
    if RESET = '1' then s <= '0';
    elsif rising_edge(CLOCK) then
      s <= not (INX and s); -- feedback
    end if;
  end process;
end moore_arch;
```

# Mealy Machine

- **Mealy Machine**: *outputs depend on state and inputs.*

```
architecture mealy_arch of fsm is
signal s: bit; -- internal state
begin
  process (INX, s)                    Combinational Logic
  begin
    OUTX <= (INX or s); -- output
  end process;
  process (CLOCK, RESET)              Sequential Logic
  begin
    if RESET = '1' then s <= '0';
    elsif rising_edge(CLOCK) then
      s <= not (INX and s); -- feedback
    end if;
  end process;
end mealy_arch;
```

# Rule of Thumb: VHDL Coding Tips

① **Maintain 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.

# Outline

- Finite State Machine (FSM)
  - Clock Edge Detection
    - "`if`" statement vs. "`wait until`" statement
    - `rising_edge(CLK)` vs. `CLK'event`
  - Direct Feedback Path
- Types of FSM
  - Moore vs. Mealy
- Examples of FSM
  - Up/Down Counter
  - Pattern Generator

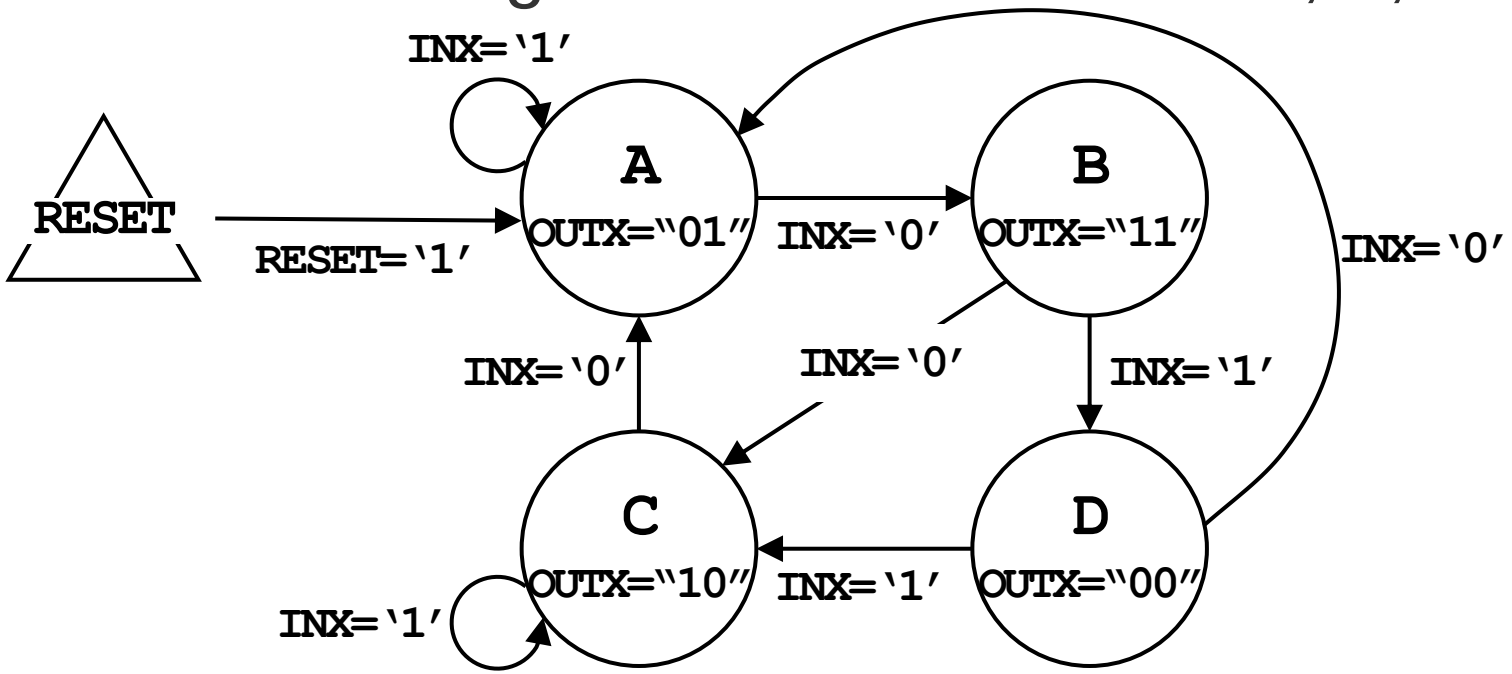
# Example 1) Up/Down Counter

- **Up/Down Counters**: Generate a sequence of counting patterns according to the clock and inputs.

```
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); -- internal state
begin
  COUNT <= s; -- output                          Combinational Logic
  process(CLK, RESET)                            Sequential Logic
  begin
    if(RESET = '1') then s <= "0000";
    else
      if( rising_edge(CLK) ) then
        s <= s + 1; -- feedback
      end if;
    end if;
  end process;
end counter_arch;
```

# Example 2) Pattern Generator (1/3)

- **Pattern Generator**: Generates any pattern we want.
  - Example: the control unit of a CPU, traffic light, etc.
- 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**.

# Example 2) Pattern Generator (2/3)

```
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
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;
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;
end arch;
```

**Sequential Logic**

**Combinational Logic**

# Example 2) Pattern Generator (3/3)

- Encoding methods for representing patterns/states:
  - **Binary Encoding**: Using N flip-flops to represent $2^N$ states.
    - Less flip-flops but more combinational logics
  - **One-hot Encoding**: Using N flip-flops for **N** states.
    - More flip-flops but less combination logic
  - *Xilinx default seeting is one-hot encoding.*
    - *Change at* ***`synthesis → options`***
    - *http://www.xilinx.com/itp/xilinx4/data/docs/sim/vtex9.html*

# Rule of Thumb: VHDL Coding Tips

① **Maintain 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)
  - Clock Edge Detection
    - "**`if`**" statement vs. "**`wait until`**" statement
    - **`rising_edge(CLK)`** vs. **`CLK'event`**
  - Direct Feedback Path
- Types of FSM
  - Moore vs. Mealy
- Examples of FSM
  - Up/Down Counter
  - Pattern Generator