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

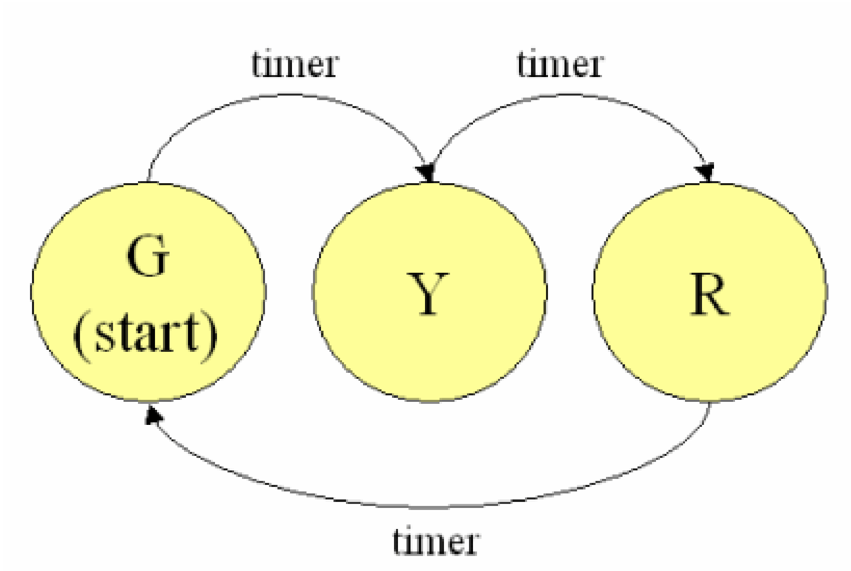
mcyang@cse.cuhk.edu.hk

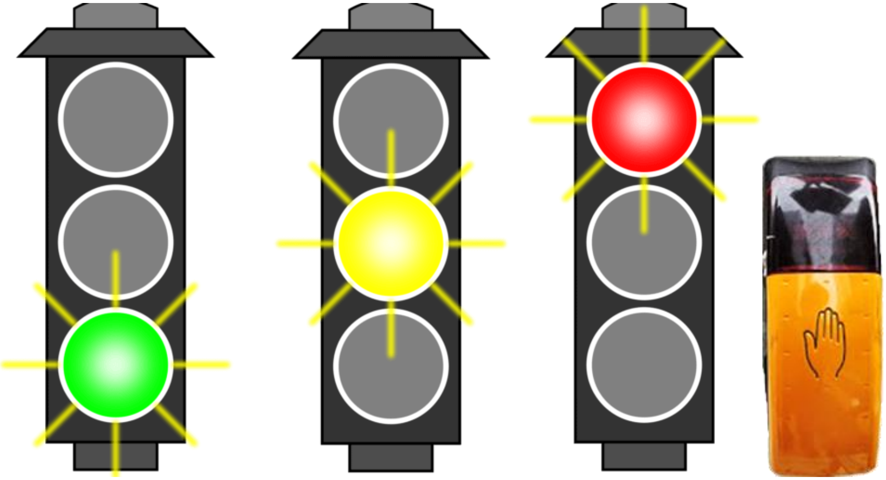
Outline

- Finite State Machine (FSM)
  - Clock Edge Detection
  - Feedback
- Use of Signals and Variables
  - Outside Process: Concurrent Statement
  - Inside Process: Sequential Statement
    - Combinational Process
    - Clocked Process
- Types of FSMs: Moore vs. Mealy
- Practical Examples
  - 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)*

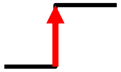
Outline

- Finite State Machine (FSM)
  - Clock Edge Detection
  - Feedback
- Use of Signals and Variables
  - Outside Process: Concurrent Statement
  - Inside Process: Sequential Statement
    - Combinational Process
    - Clocked Process
- Types of FSMs: Moore vs. Mealy
- Practical Examples
  - Up/Down Counter
  - Pattern Generator

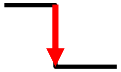
Clock Edge Detection

- "**if**" or "**wait until**" statements can be used to detect the clock edge of a clock signal (e.g., **CLK**):

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

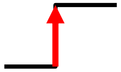


  - `if CLK'event and CLK = '0' -- falling edge`

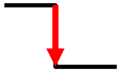


OR

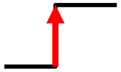
  - `if( rising_edge(CLK) ) -- rising edge`



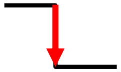
  - `if( falling_edge(CLK) ) -- falling edge`



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

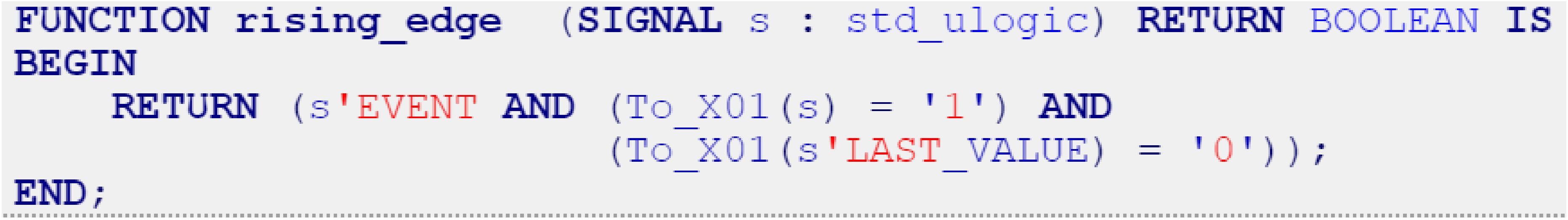


  - `wait until CLK = '0'; --falling edge`



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;
```

  - This function returns a value **TRUE** only when the present value is '1' and the last value is '0'.
  - If the past value is something like 'Z','U' etc. then it will return a **FALSE** value.

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

Just use **rising_edge()** *and* **falling_edge()** *functions!*

How to use "if" or "wait until"? (1/2)

- **Synchronous Process**: Computes values only on clock edges (i.e., only sensitive/sync. to clock signal).
  - Both "wait-until" or "if" statements can be used:

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.
   …
end process
```

How to use "if" or "wait until"? (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.
  - For async. processes, only "**if**" statements can be used:

Usage of "if"

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

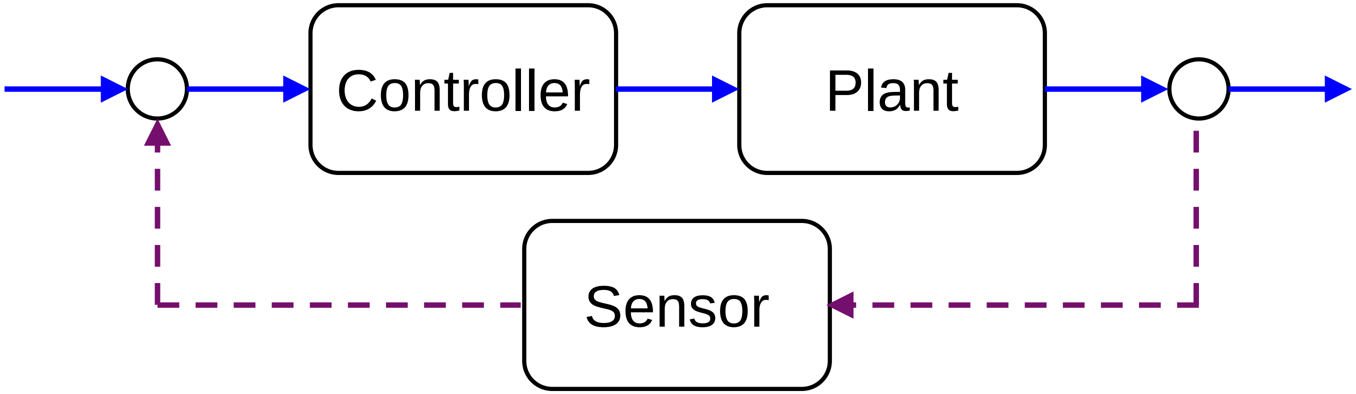
← The sensitivity list should include the clock signal, and all inputs that may affect asynchronous behavior.

Outline

- Finite State Machine (FSM)
  - Clock Edge Detection
  - Feedback
- Use of Signals and Variables
  - Outside Process: Concurrent Statement
  - Inside Process: Sequential Statement
    - Combinational Process
    - Clocked Process
- Types of FSMs: Moore vs. Mealy
- Practical Examples
  - Up/Down Counter
  - Pattern Generator

Feed-forward and Feedback Paths

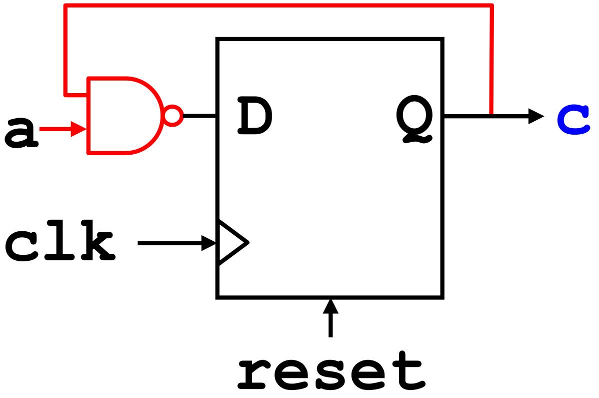
- So far, we only learned logic with feed-forward (or open-loop) paths.



- Now, we are going to learn feedback (or closed-loop) paths—*the key step of making a finite state machine*.
- There are three types of feedback paths:
  1) **Direct Feedback**
  2) **Feedback using Signals**
  3) **Feedback using Variables**

1) Direct Feedback

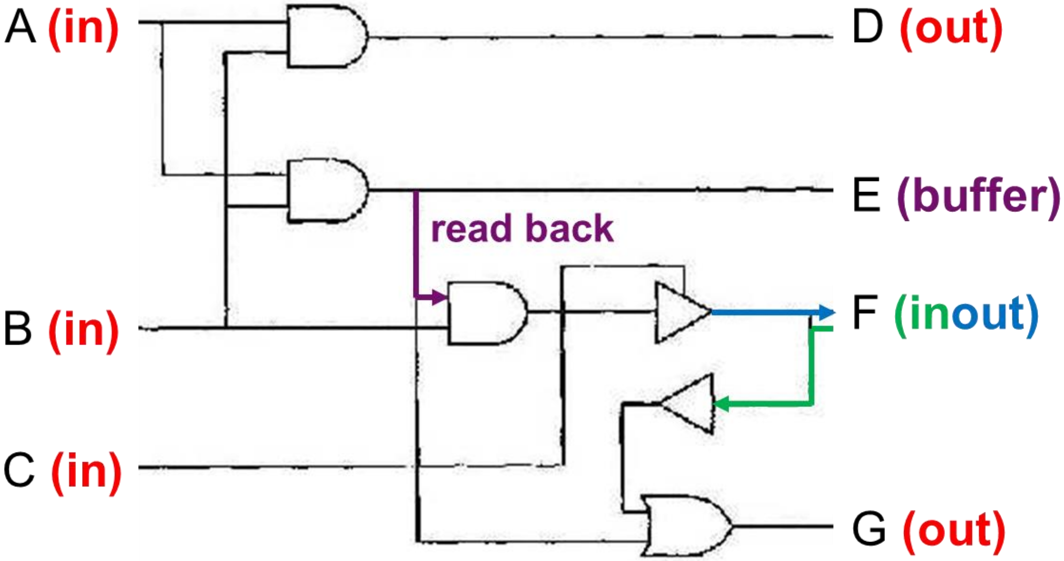
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity feedback_1 is
port(a,clk,reset: in std_logic;
     c: buffer std_logic);
end feedback_1;
architecture feedback_1_arch of feedback_1 is
begin
  process(clk, reset) -- async.
  begin
    if reset = '1' then c <= '0';
    elsif rising_edge(clk) then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



← not(a and c) will take effect and be assigned to c at the next rising clock edge.

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.

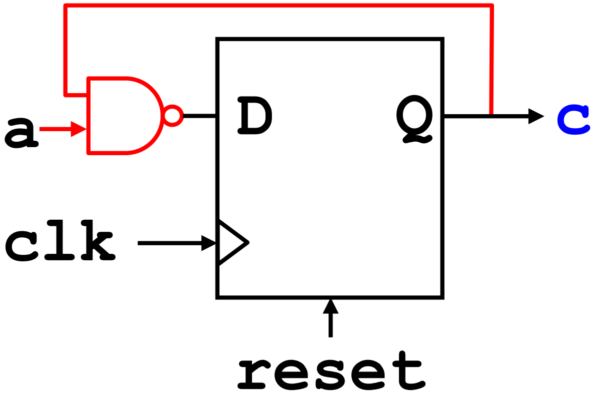
Class Exercise 5.1

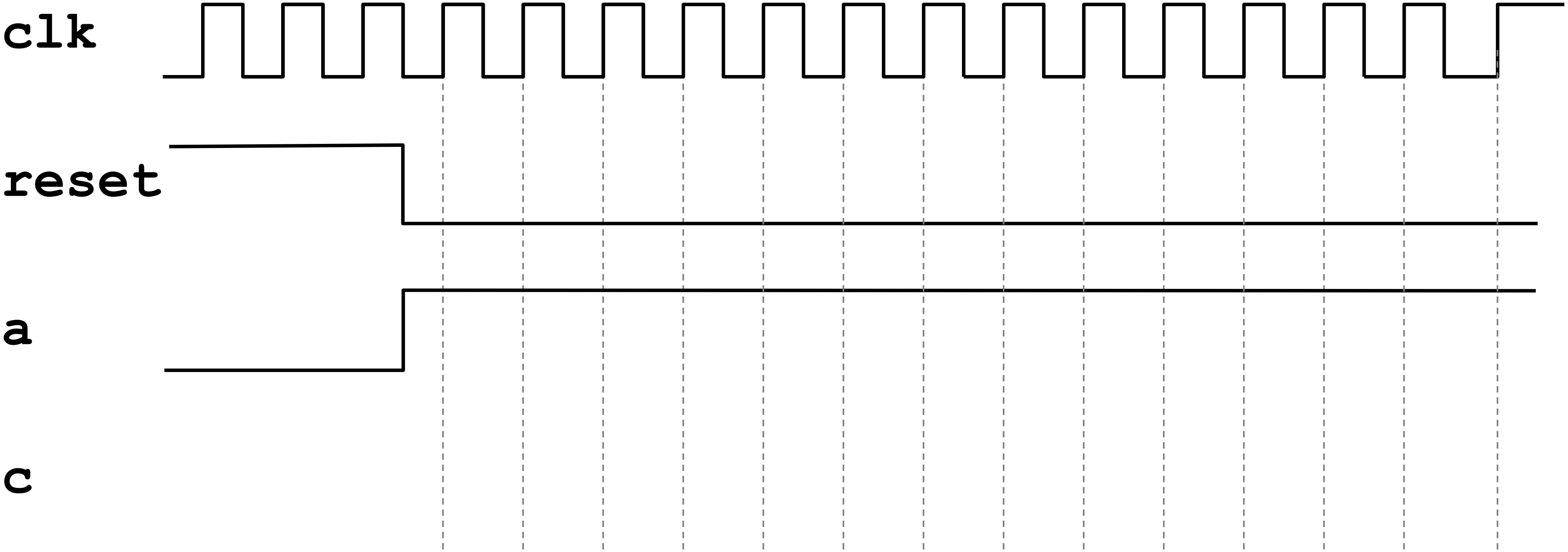
Student ID: ______________ Date:

Name: ____________________ ________

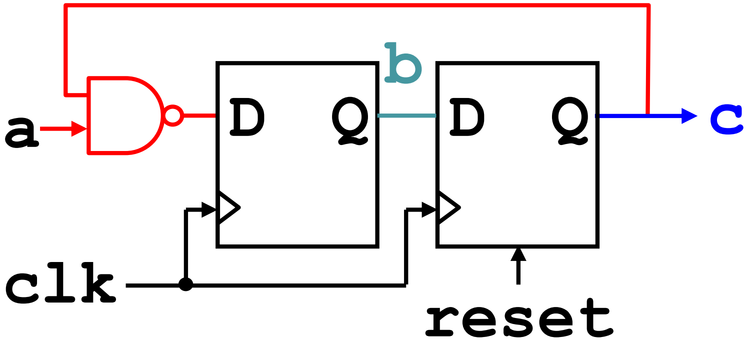
- Draw the signal **c**
  - Assume initially **c = 0**

```
elsif rising_edge(clk) then
   c <= not(a and c);
```





2) Feedback using Signals



```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity feedback_2 is
port(a,clk,reset: in std_logic;
     c: buffer std_logic);
end feedback_2;
architecture feedback_2_arch of feedback_2 is
signal b: std_logic; -- internal signal b
begin
  process(clk,reset)
  begin
    if reset = '1' then c <= '0';
    elsif rising_edge(clk) then
      b <= not(a and c);
      c <= b;
    end if;
  end process;
end feedback 2 arch ;
```

Why? Combinational logic

← not(a and c) will take effect and be assigned to b at the next rising clock edge.

← b will be assigned to c at the next rising clock edge.

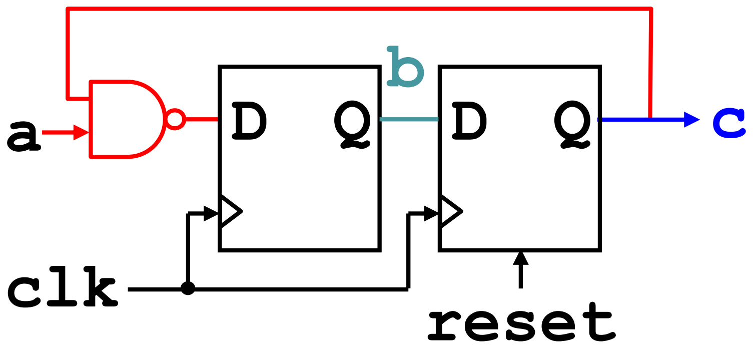
Class Exercise 5.2

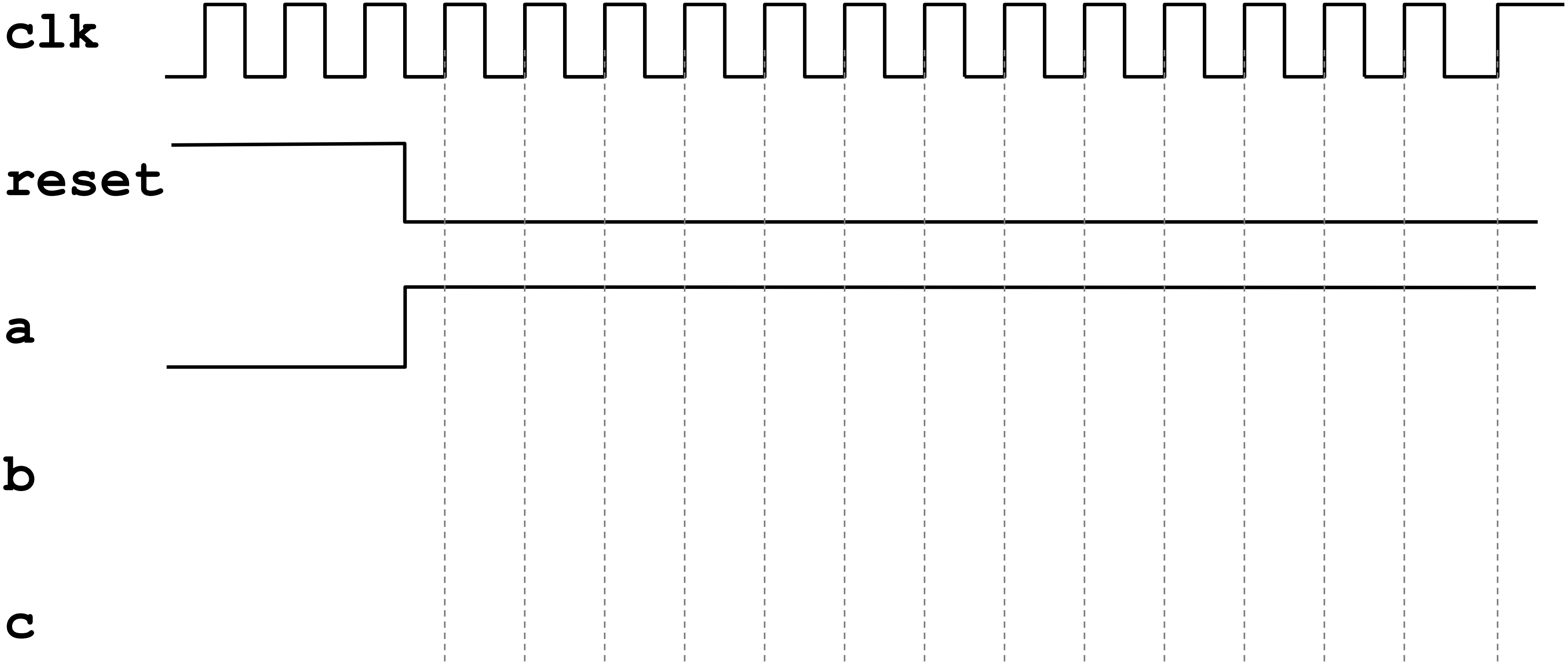
Student ID: ____________ Date:

Name: ________________ ________

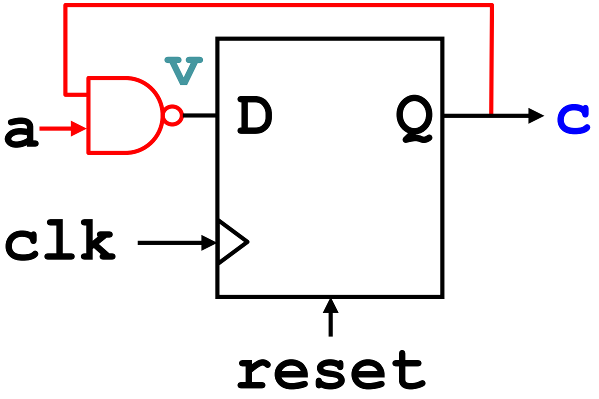
- Draw signals **b**, and **c**
  - Assume initially **b=1** and **c=0**

```
elsif rising_edge(clk) then
   b <= not(a and c);
   c <= b;
```





3) Feedback using Variables



```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity feedback_3 is
Port(a,clk,reset: in std_logic;
     c: buffer std_logic);
End feedback_3;
architecture feedback_3_arch of feedback_3 is
begin
  process -- no sensitivity list
  variable v: std_logic; -- local variable v
  begin
    wait until clk = '1';
    if reset = '1' then v := '0';
    else
      v := not(a and c);
      c <= v;
    end if;
  end process;
end feedback_3_arch ;
```

← not(a and c) affects v immediately at the next rising clock edge.

← The previous new v will be assigned to c at the same rising clock edge.

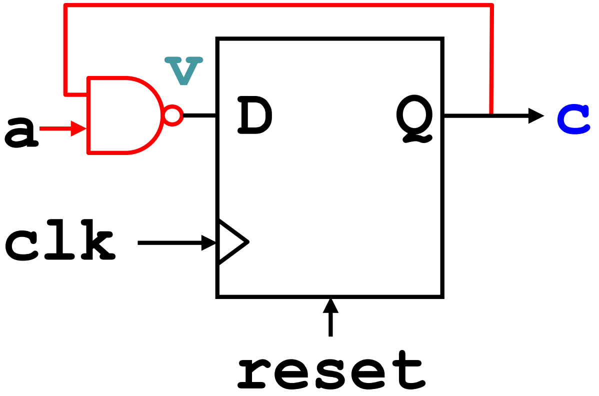
Class Exercise 5.3

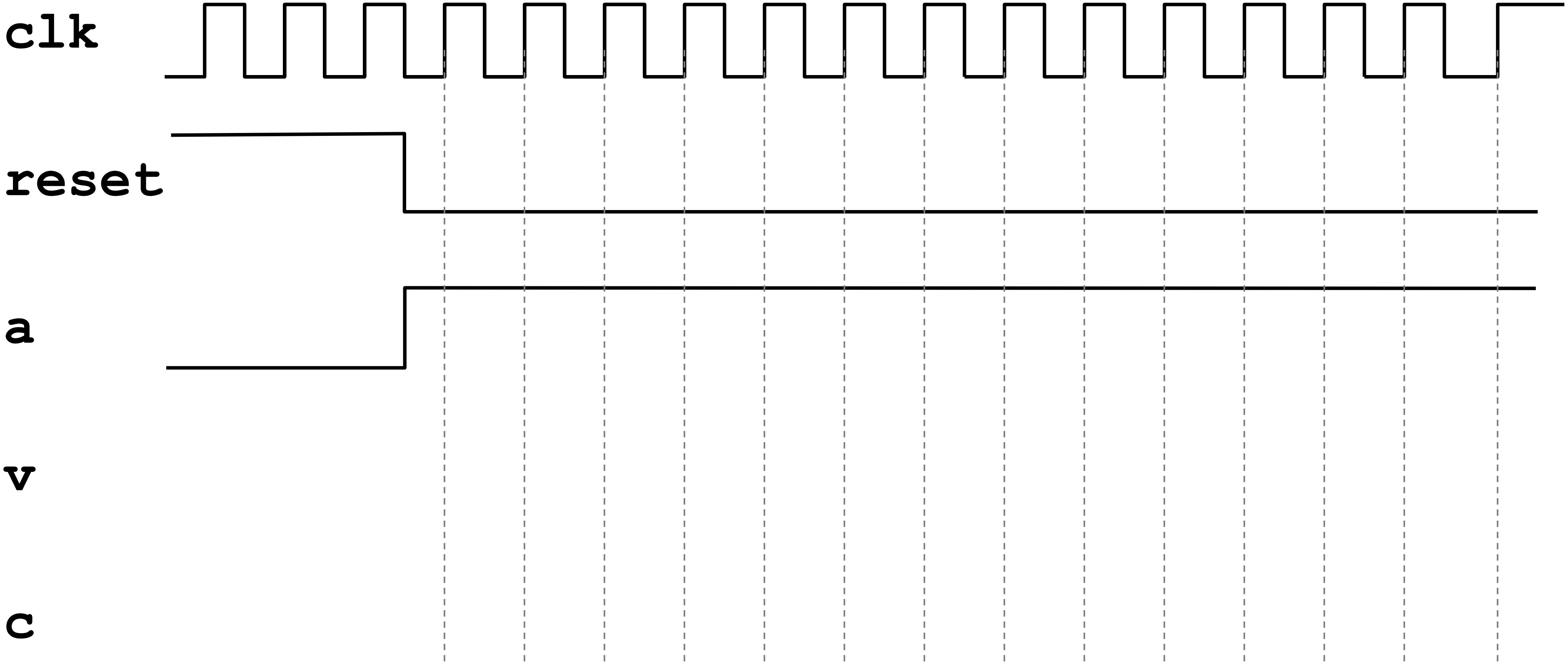
Student ID: ______________ Date:

Name: ____________________ _________

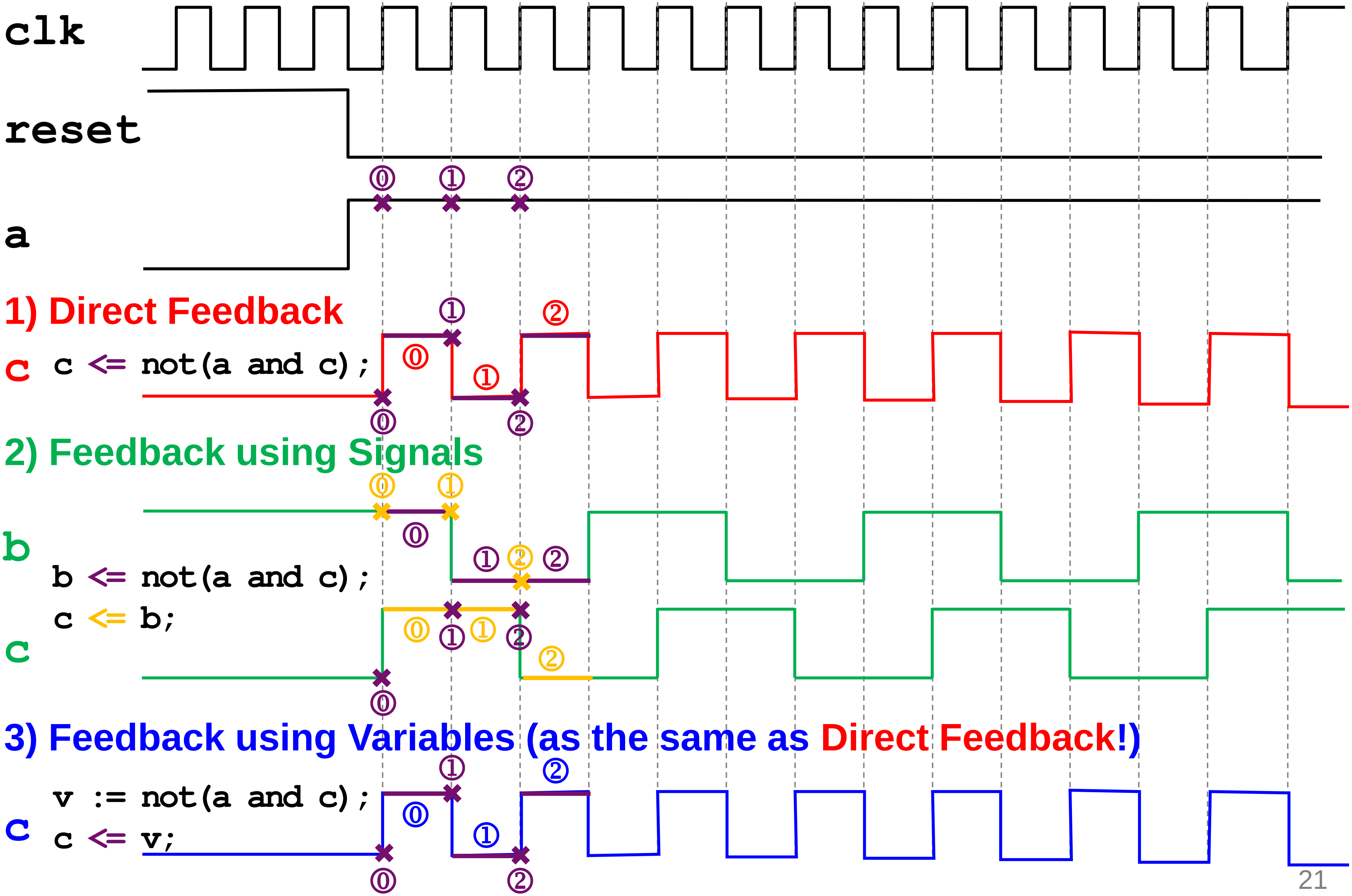
- Draw variable **v** and signal **c**
  - Assume initially `c = 0`

```
wait until clk = '1';
v := not(a and c);
c <= v;
```





Direct vs. Signal vs. Variable Feedback



1) Direct Feedback

```
c <= not(a and c);
```

2) Feedback using Signals

```
b <= not(a and c);
c <= b;
```

3) Feedback using Variables (as the same as Direct Feedback!)

```
v := not(a and c);
c <= v;
```

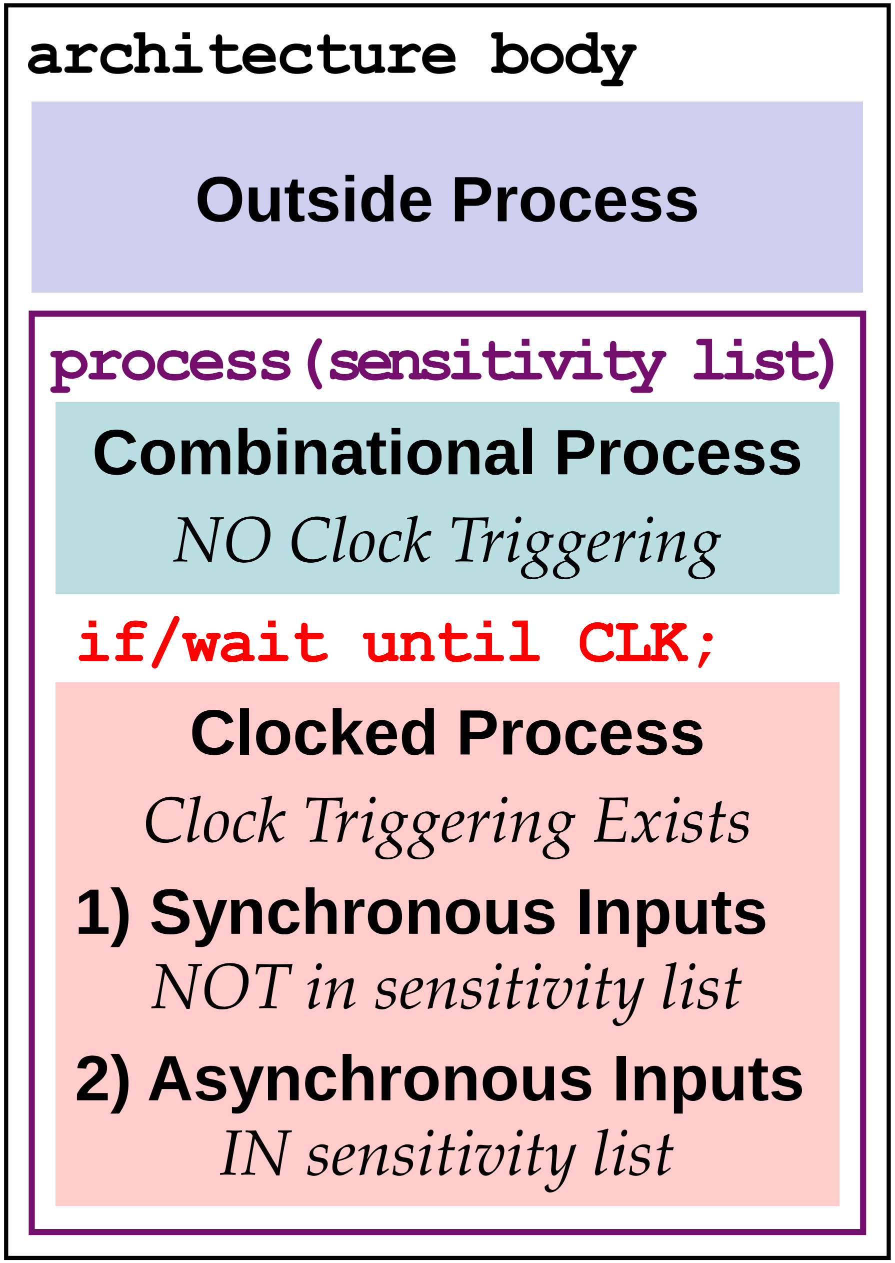
Signal Feedback vs. Variable Feedback

- Feedback using signals or variables will have different results.
- **Signals**
  - Signal assignment "**<=**" can be treated as a flip-flop.
    - Left-hand-side of "**<=**" is output
    - Right-hand-side of "**<=**" is input
  - A signal can be only updated once, when the process is performed at the triggering clock edge.
    - When a signal is assigned to different values by different statements in a process, only the last statement is effective.
- **Variables**
  - Variable assignment "**:=**" will take effect immediately.
  - A variable in a process can be updated many times.

Outline

- Finite State Machine (FSM)
  - Clock Edge Detection
  - Feedback
- Use of Signals and Variables
  - Outside Process: Concurrent Statement
  - Inside Process: Sequential Statement
    - Combinational Process
    - Clocked Process
- Types of FSMs: Moore vs. Mealy
- Practical Examples
  - Up/Down Counter
  - Pattern Generator

Overview: Use of Signals and Variables



`architecture body`

Outside Process

`process(sensitivity list)`

Combinational Process

NO Clock Triggering

`if/wait until CLK;`

Clocked Process

Clock Triggering Exists

1) Synchronous Inputs

NOT in sensitivity list

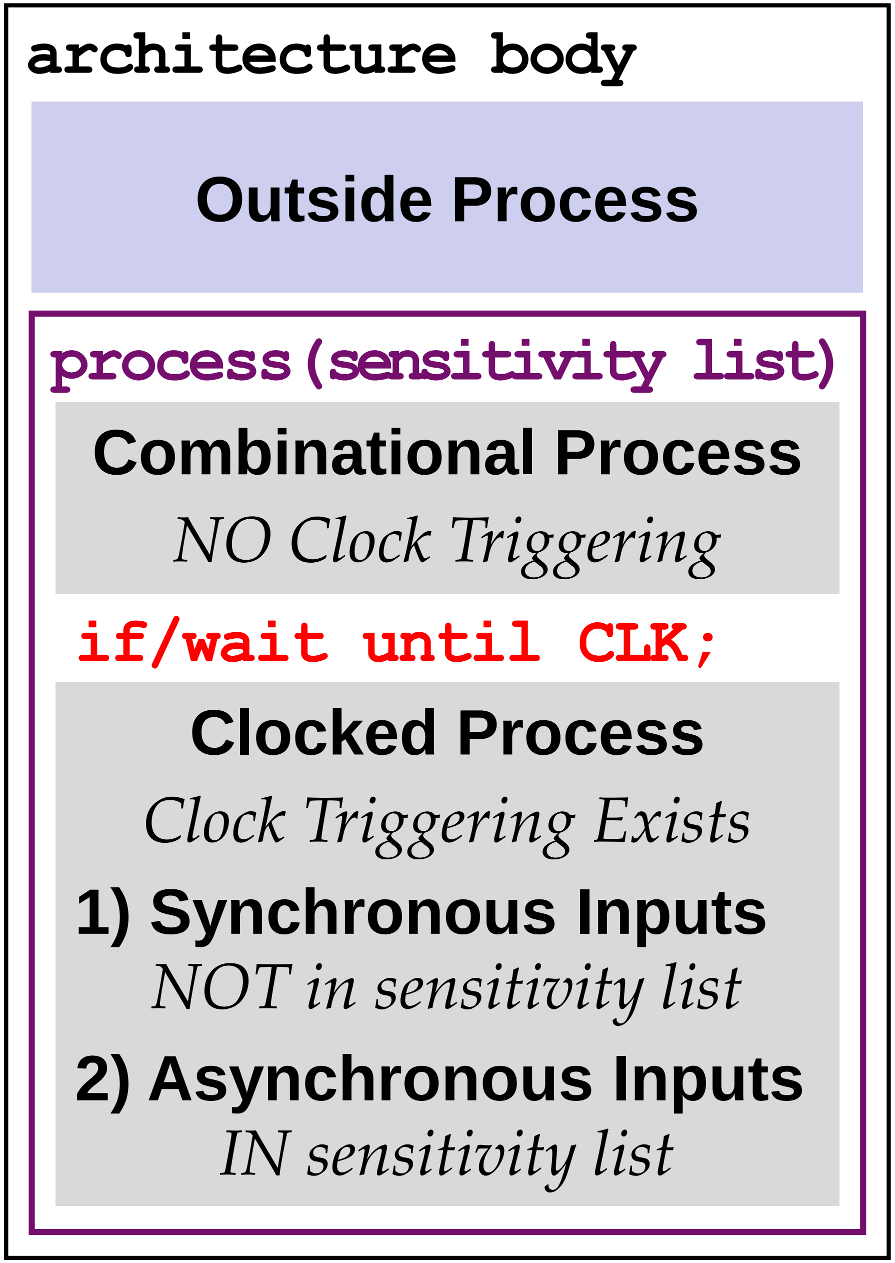
2) Asynchronous Inputs

IN sensitivity list

- **Outside Process**
  - Concurrent Statements
- **Inside Process**
  - Sequential Statements

1) **Combinational Process**: NO `CLK` triggering
   - "**<=**" is a combinational logic
   - All involved inputs should be in the sensitivity list
2) **Clocked Process**: Has `CLK` triggering
   - "**<=**" is a flip-flop
   - **Synchronous Inputs**: should NOT be in the sensitivity list
   - **Asynchronous Inputs**: should be in sensitivity list

Overview: Use of Signals and Variables



architecture body

Outside Process

process(sensitivity list)

Combinational Process

NO Clock Triggering

if/wait until CLK;

Clocked Process

Clock Triggering Exists

1) Synchronous Inputs

NOT in sensitivity list

2) Asynchronous Inputs

IN sensitivity list

- **Outside Process**
  - – Concurrent Statements
- **Inside Process**
  - – Sequential Statements
  1) **Combinational Process**: NO **CLK** triggering
     - "<=" is a combinational logic
     - All involved inputs should be in the sensitivity list
  2) **Clocked Process**: Has **CLK** triggering
     - "<=" is a flip-flop
     - **Synchronous Inputs**: should NOT be in the sensitivity list
     - **Asynchronous Inputs**: should be in sensitivity list

Outside Process: Concurrent Statement

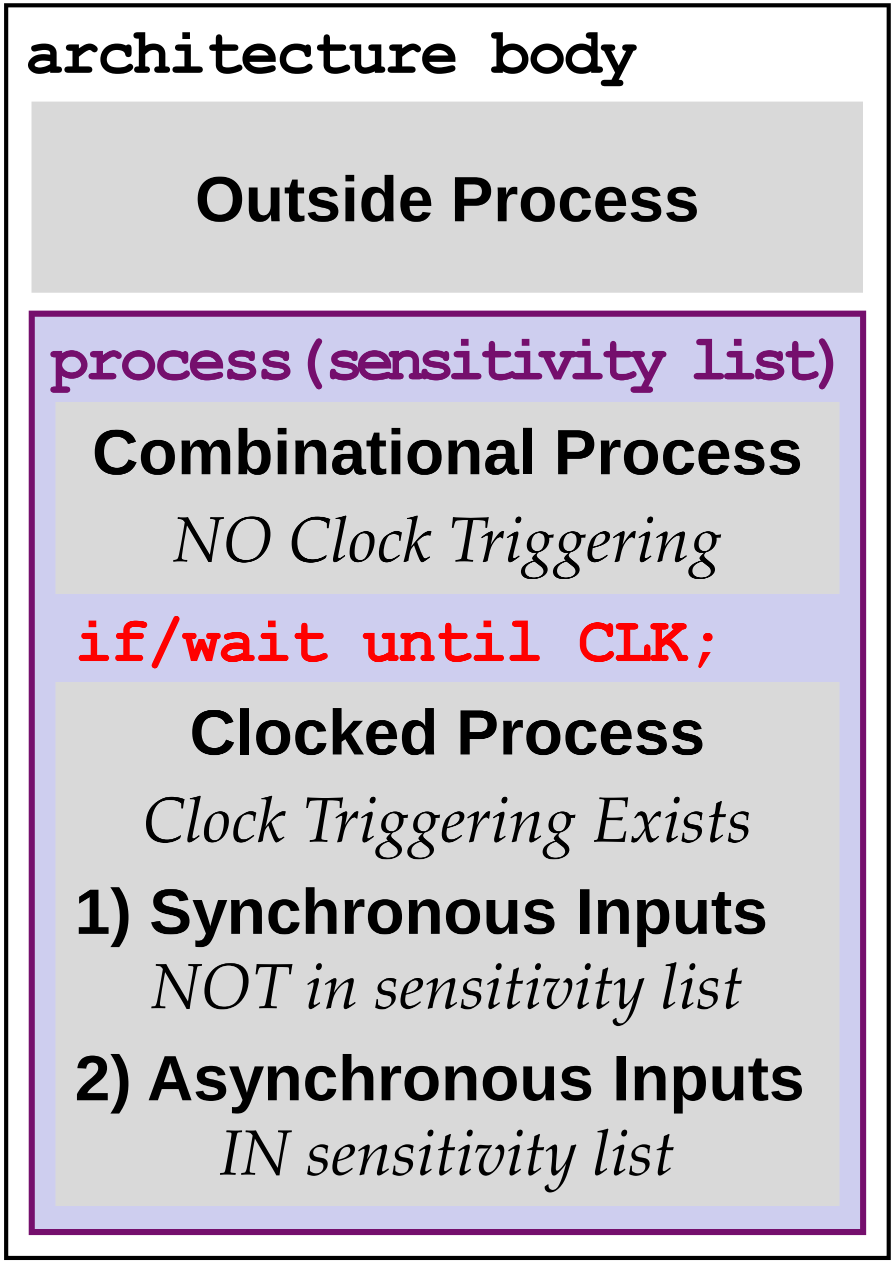
- **Signal Assignments outside a Process**
  - All the statements outside processes are "concurrent".
    - All concurrent statements can be interchanged freely.
    - Each statement will be executed once when any signal in it changes.
  - Signals can be assigned with multiple values if "resolved logic" (i.e., `std_logic` rather than `std_ulogic`) is allowed.

Ex:

```
architecture test_arch of test is
  out1 <= in1 and in2; -- concurrent statement
  out2 <= in1 or in2;  -- concurrent statement
  out2 <= in2;         -- multi-value assignment
end test_arch;
```

- **Variable Assignments outside a Process**
  - Variables can only live *inside* processes!

Overview: Use of Signals and Variables



`architecture body`

Outside Process

`process (sensitivity list)`

Combinational Process

NO Clock Triggering

`if/wait until CLK;`

Clocked Process

Clock Triggering Exists

1) Synchronous Inputs

NOT in sensitivity list

2) Asynchronous Inputs

IN sensitivity list

- Outside Process
  - – Concurrent Statements
- **Inside Process**
  - – Sequential Statements
  1) **Combinational Process**: NO `CLK` triggering
     - "<=" is a combinational logic
     - All involved inputs should be in the sensitivity list
  2) **Clocked Process**: Has `CLK` triggering
     - "<=" is a flip-flop
     - **Synchronous Inputs**: should NOT be in the sensitivity list
     - **Asynchronous Inputs**: should be in sensitivity list

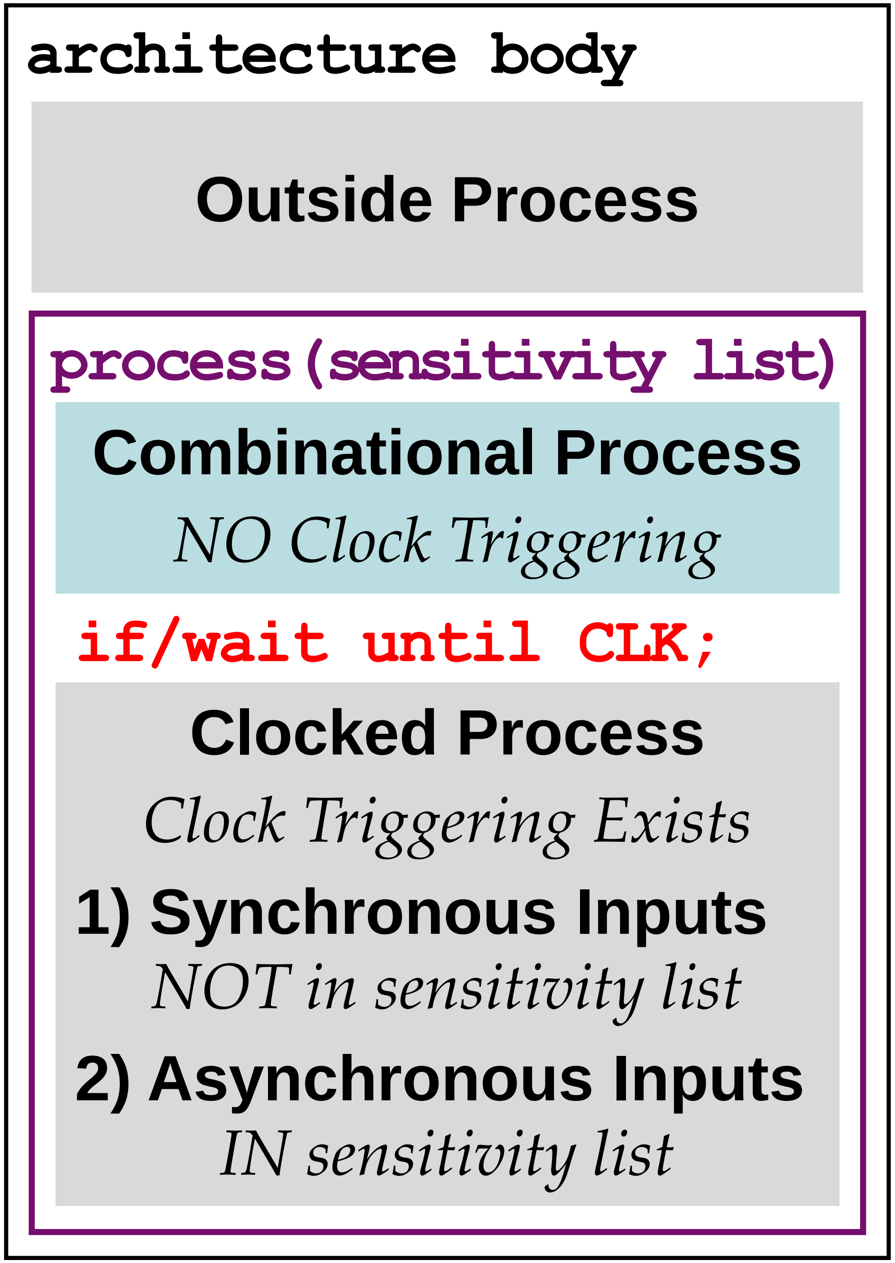
Inside Process: Sequential Statement

- Statements inside **process** are executed sequentially.
  - The process will be executed once when one or more signals in the sensitivity list changes.

Ex:
```
process(in1, in2) -- sensitivity list
variable v1, v2: std_logic;
begin
  s1 <= in1 and in2;
  s1 <= in1 or in2;
  v1 := in1 and in2;
  v1 := in1 or in2;
end process
```

  - **Signals Assignments (<=) inside a Process:** Only ***the last*** assignment for a particular signal takes effect.
  - **Variables Assignments (:=) inside a Process:** ***All*** assignments take effect immediately and sequentially.
- A process can be: "combinational" or "clocked".

Overview: Use of Signals and Variables



architecture body

Outside Process

process (sensitivity list)

Combinational Process

NO Clock Triggering

if/wait until CLK;

Clocked Process

Clock Triggering Exists

1) Synchronous Inputs

NOT in sensitivity list

2) Asynchronous Inputs

IN sensitivity list

- Outside Process
  - Concurrent Statements
- **Inside Process**
  - Sequential Statements
  1) **Combinational Process**: NO `CLK` triggering
     - “<=” is a combinational logic
     - All involved inputs should be in the sensitivity list
  2) **Clocked Process**: Has `CLK` triggering
     - “<=” is a flip-flop
     - **Synchronous Inputs**: should NOT be in the sensitivity list
     - **Asynchronous Inputs**: should be in sensitivity list

1) Combinational Process

- **Combinational Process**
  - *NO* clock triggering condition can be found inside.
    - **Clock Triggering Condition**: `if (clk='1' and clk'event)`, `(wait until clk='1')`, etc.
  - Each “<=” is a combinational logic.
  - *All* involved inputs should be in the sensitivity list.
    - Otherwise the results will be unpredictable.

Ex:

```
combinational_process: process(in1, in2)
begin
   out3 <= in1 xor in2;
   out3 <= ’1’;
end process;
```

Class Exercise 5.4

Student ID: ___________ Date:

Name: _______________ _______

```
 1 signal S1, S2: bit;
 2 signal S_OUT: bit_vector(1 to 8);
 3 process (S1, S2)
 4 variable V1, V2: bit;
 5 begin
 6   V1 := '1';
 7   V2 := '1';
 8   S1 <= '1';
 9   S2 <= '1';
10   S_OUT(1) <= V1;
11   S_OUT(2) <= V2;
12   S_OUT(3) <= S1;
13   S_OUT(4) <= S2;
14   V1 := '0';
15   V2 := '0';
16   S2 <= '0';
17   S_OUT(5) <= V1;
18   S_OUT(6) <= V2;
19   S_OUT(7) <= S1;
20   S_OUT(8) <= S2;
21 end process;
```

- Which line(s) will NOT take effect?
  Answer: ________________

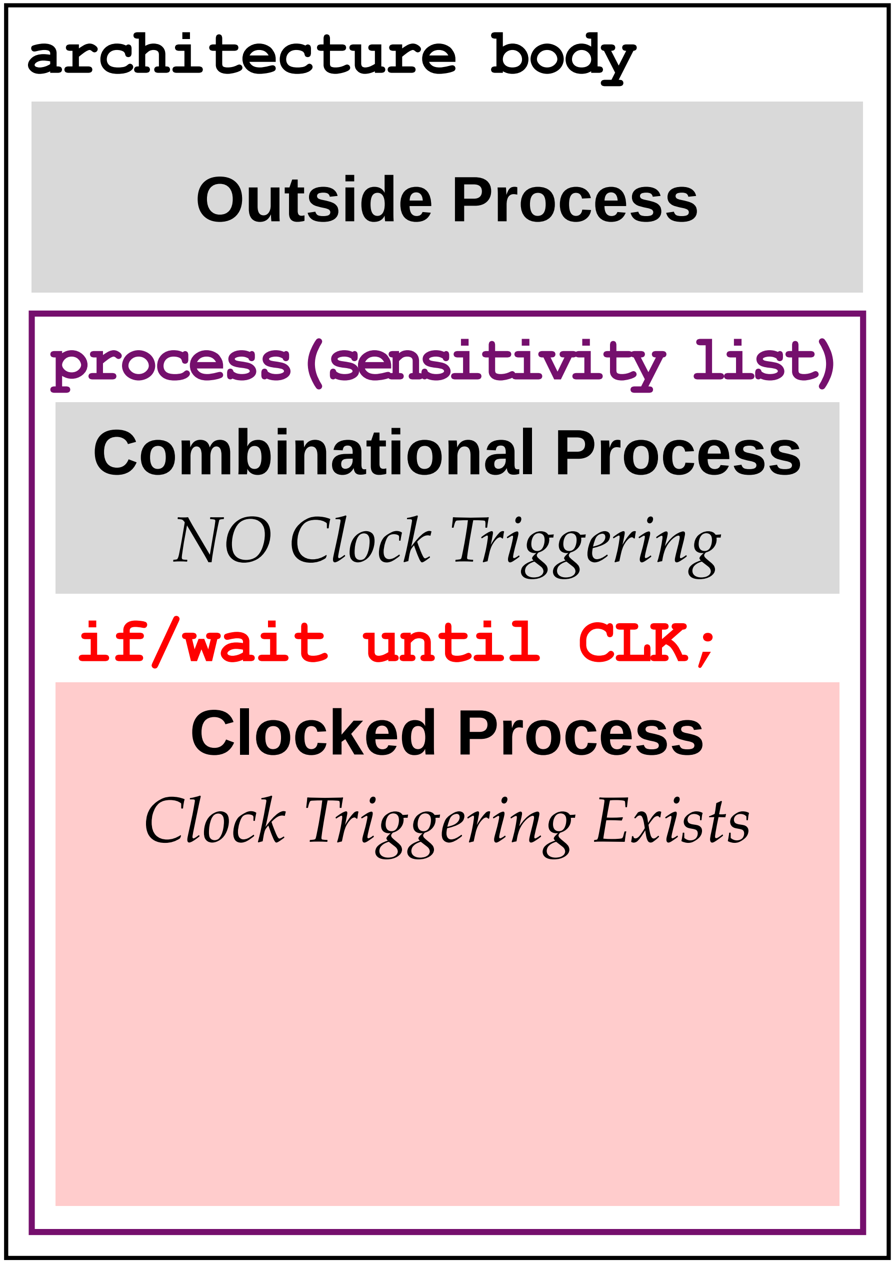
- When will the process be executed?
  Answer: ________________
  ____

- What are the values of **`S_OUT`** after execution?
  Answer:

`S_OUT(1):`
`S_OUT(2):`
`S_OUT(3):`
`S_OUT(4):`

`S_OUT(5):`
`S_OUT(6):`
`S_OUT(7):`
`S_OUT(8):`

Overview: Use of Signals and Variables



architecture body

Outside Process

process (sensitivity list)

Combinational Process

NO Clock Triggering

if/wait until CLK;

Clocked Process

Clock Triggering Exists

- Outside Process
  - Concurrent Statements
- Inside Process
  - Sequential Statements

1) **Combinational Process**: NO `CLK` triggering
   - "<=" is a combinational logic
   - All involved inputs should be in the sensitivity list

2) **Clocked Process**: Has `CLK` triggering
   - "<=" is a flip-flop
   - **Synchronous Inputs**: should NOT be in the sensitivity list
   - **Asynchronous Inputs**: should be in sensitivity list

2) Clocked Process

- **Clocked Process**
  - A clock edge expression can be found inside:
    - "**if**" statement:

```
clocked_process: process(sensitivity list)
begin
  … -- same as combinational process
  if (clk='1' and clk'event) then
    out1 <= in1 and in2;
  end if;
  … -- same as combinational process
end process;
```

1) Each "**<=**" is a flip-flop.

2) The assignment takes effect on next clock edge.

    - "**wait until**" statement:

```
clocked_process: process -- no sensitivity list
begin
  wait until clk='1';
  out1 <= in1 and in2;
end process;
```

1) Each "**<=**" is a flip-flop.

2) The assignment takes effect on next clock edge.

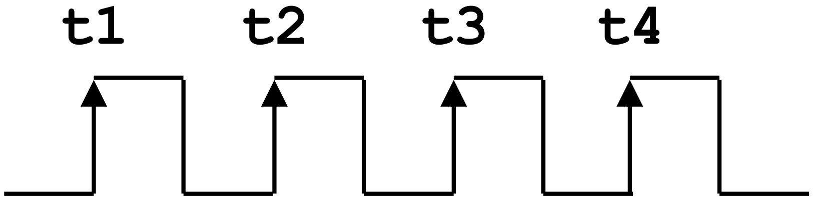
Class Exercise 5.5

Student ID: ____________ Date:

Name: ________________ ________

- Find the signal results after clock edges **t1** ~ **t4**:

```
process
signal s1: integer:=1;
signal s2: integer:=2;
signal s3: integer:=3;
begin
wait until rising_edge(clk);
   s1 <= s2 + s3;
   s2 <= s1;
   s3 <= s2;
   sum <= s1 + s2 + s3;
end process
end
```



	t1	t2	t3	t4
s1				
s2				
s3				
sum				

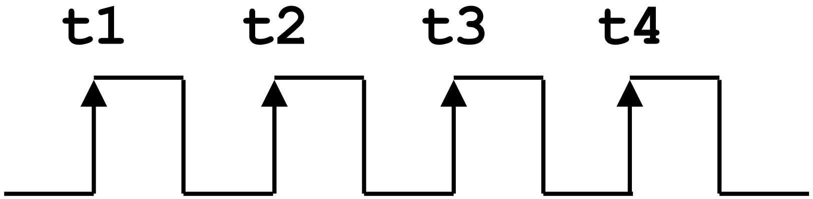
Signals Assignments (<=) inside a Process:
Only *the last* assignment for a particular signal takes effect.
Variables Assignments (:=) inside a Process:
All assignments take effect immediately and sequentially.

Class Exercise 5.6

Student ID: ____________ Date:
Name: ________________ ________

- Find the signal results after clock edges **t1** ~ **t4**:

```
process
variable v1: integer:=1;
variable v2: integer:=2;
variable v3: integer:=3;
begin
wait until rising_edge(clk);
  v1 := v2 + v3;
  v2 := v1;
  v3 := v2;
  sum <= v1 + v2 + v3;
end process
end
```



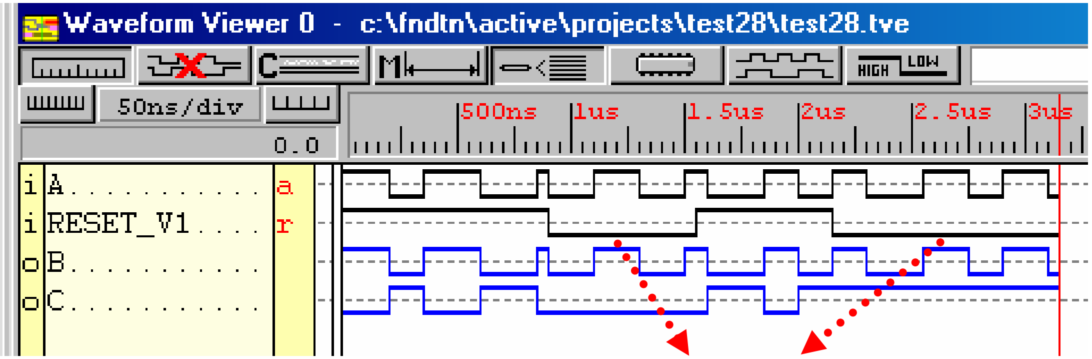
	t1	t2	t3	t4
v1				
v2				
v3				
sum				

Signals Assignments (<=) inside a Process:
Only *the last* assignment for a particular signal takes effect.
Variables Assignments (:=) inside a Process:
All assignments take effect immediately and sequentially.

Do Variables Have Memory?

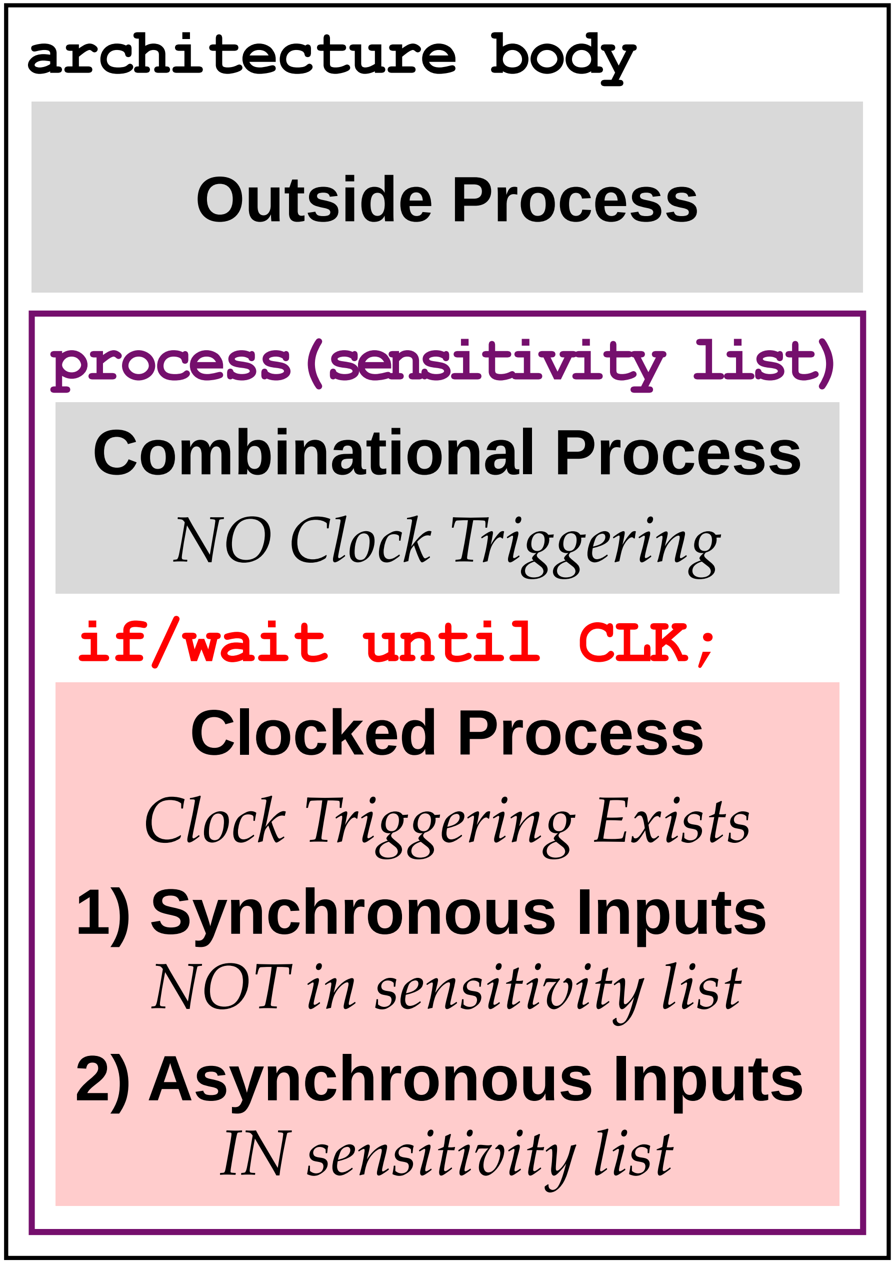
- **Yes.** After a process is called, the state of a variable will be kept for being used again next time.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity test is port (a, reset_v1: in std_logic;
                     b, c: out std_logic); end test;
architecture test_arch of test is
begin
label_proc1: process (a, reset_v1)
variable v1 : std_logic;
begin
  if reset_v1 ='1' then
    v1:= not a;
  end if;
  b <= a;
  c <= v1;
end process label_proc1;
end test_arch;
```



v1 stays at two different levels depending on previous result.

Overview: Use of Signals and Variables



architecture body

Outside Process

process (sensitivity list)

Combinational Process

NO Clock Triggering

if/wait until CLK;

Clocked Process

Clock Triggering Exists

1) Synchronous Inputs

NOT in sensitivity list

2) Asynchronous Inputs

IN sensitivity list

- **Outside Process**
  - Concurrent Statements
- **Inside Process**
  - Sequential Statements

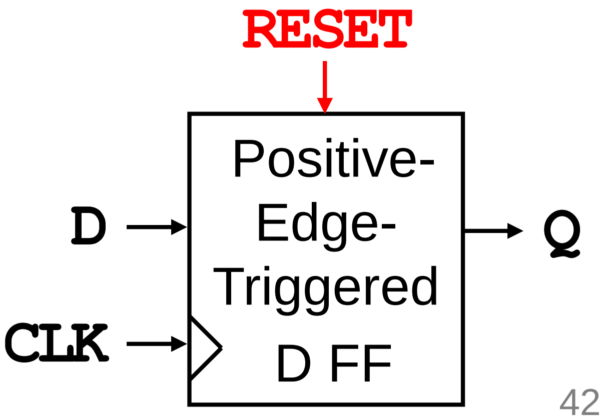
1) **Combinational Process**: NO `CLK` triggering
   - "<=" is a combinational logic
   - All involved inputs should be in the sensitivity list
2) **Clocked Process**: Has `CLK` triggering
   - "<=" is a flip-flop
   - **Synchronous Inputs**: should NOT be in the sensitivity list
   - **Asynchronous Inputs**: should be in sensitivity list

Synchronous & Asynchronous Inputs

- Besides of the clock signal (**CLK**), other signals in a **clocked process** can be classified into two types:

1) **Synchronous Inputs** (e.g., **D** input of flip-flops)
   - Inputs that should be checked only at the next clock edge.
   - ***NO*** need to put synchronous input signals in the sensitivity list.
2) **Asynchronous Inputs** (e.g., **RESET** input of flip-flops)
   - Inputs that should be checked either at the next clock edge or when any asynchronous input in the sensitivity list changes.
   - Asynchronous inputs ***NEVER*** exist in **wait-until** clocked processes.

```
process(CLK, RESET) -- no need to put D, why?
begin
  if (RESET = '1') then
    Q <= '0'; -- Reset Q immediately
  elsif CLK = '1' and CLK'event then
    Q <= D;    -- Q follows input D
  end if;
end process;
```



Class Exercise 5.7

Student ID: __________ Date:
Name: ______________ ________

- What are processes p1 and p2 (combinational or clocked)?
- Which signals are sync., async., or combinational inputs?

```
…
port(clock,reset: in std_logic;
         t_light: out std_logic_vector (2 downto 0));
…
type traffic_state_type is (s0, s1,s2,s3);
signal t_state: traffic_state_type; -- internal signal
```

```
p1: process(t_state)
begin
  case (t_state) is
    when s0 => t_light <= "100";
    when s1 => t_light <= "110";
    when s2 => t_light <= "001";
    when s3 => t_light <= "010";
  end case;
end process;
```

```
p2: process
begin
  wait until clock='1';
  if reset = '1' then
     t_state <= s0;
  else
    case t_state is
      when s0 => t_state <= s1;
      when s1 => t_state <= s2;
      when s2 => t_state <= s3;
      when s3 => t_state <= s0;
    end case;
  end if;
end process;
```

Class Exercise 5.8

Student ID: ___________ Date:
Name: ________________ _______

- Based on Class Exercise 5.7, rewrite process p2 using asynchronous reset.

```
sync_p2: process
begin
  wait until clock='1';
  if reset = '1' then
     t_state <= s0;
  else
    case t_state is
      when s0 => t_state <= s1;
      when s1 => t_state <= s2;
      when s2 => t_state <= s3;
      when s3 => t_state <= s0;
    end case;
  end if;
end process;
```

```
async_p2: process
begin









end process;
```

Recall: How to use "if" or "wait until"?

- **Asynchronous Process**: Computes values on clock edges and 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.
  - Only "**if**" statements can be used: ②

Usage of "if"

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

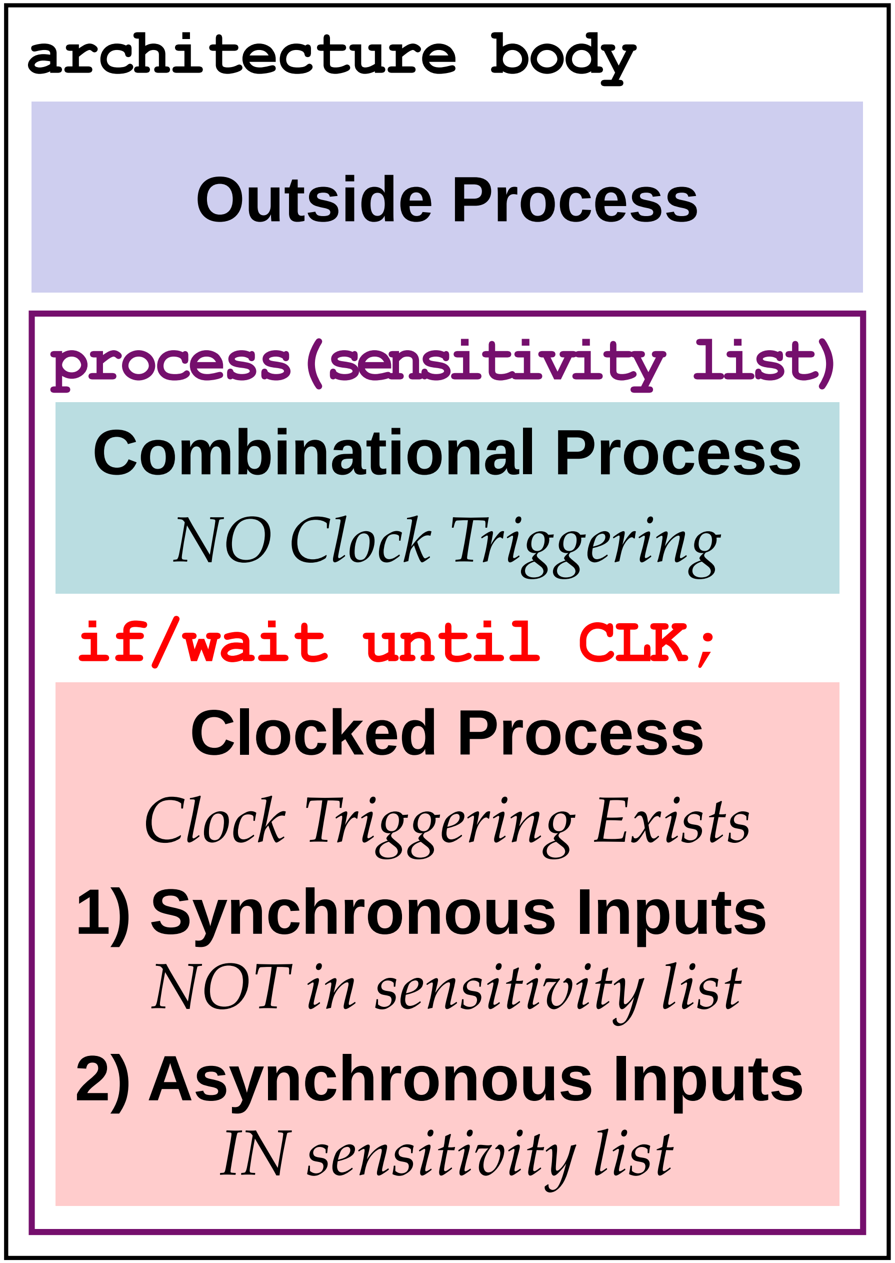
①

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

Summary: Inside Process

- **Signals Assignments (<=) inside a Process**
  - Only the *last* assignment for a particular signal takes effect.
  - **Combinational Process**: *No* clock (**`CLK`**) triggering
    - Each "<=" is a combinational logic.
    - All involved inputs should be in the sensitivity list.
  - **Clocked Process**: Has clock (**`CLK`**) triggering
    - Signal assignments *before* or *outside* the clock edge detection:
      - As the same as combinational process.
    - Signal assignments *after* or *inside* the clock edge detection:
      - Each "<=" can be treated as a flip-flop: The signal assignment will take effect at the next clock edge.
      - **Synchronous inputs** should *NOT* be in the sensitivity list.
      - **Asynchronous inputs** should be in the sensitivity list.
- **Variables Assignments (`:=`) inside a Process**
  - *All* assignments take effect immediately and sequentially.

Summary: Use of Signals and Variables



`architecture body`

Outside Process

`process (sensitivity list)`

Combinational Process

NO Clock Triggering

`if/wait until CLK;`

Clocked Process

Clock Triggering Exists

1) Synchronous Inputs

NOT in sensitivity list

2) Asynchronous Inputs

IN sensitivity list

- **Outside Process**
  - – Concurrent Statements
- **Inside Process**
  - – Sequential Statements

1) **Combinational Process**: NO `CLK` triggering
   - "**<=**" is a combinational logic
   - All involved inputs should be in the sensitivity list
2) **Clocked Process**: Has `CLK` triggering
   - "**<=**" is a flip-flop
   - **Synchronous Inputs**: should NOT be in the sensitivity list
   - **Asynchronous Inputs**: should be in sensitivity list

Summary: Multiple Assignments

- **Signals**
  - **Outside Process**
    - Signals can be assigned with multiple values if “*resolved logic*” is allowed.
  - **Inside Process**
    - Only *the last* assignment for that particular signal will take effect.
- **Variables**
  - **Outside Process**
    - Variables can only live *inside* processes! 🚫
  - **Inside Process**
    - *ALL* variable assignments will take effect immediately and sequentially.

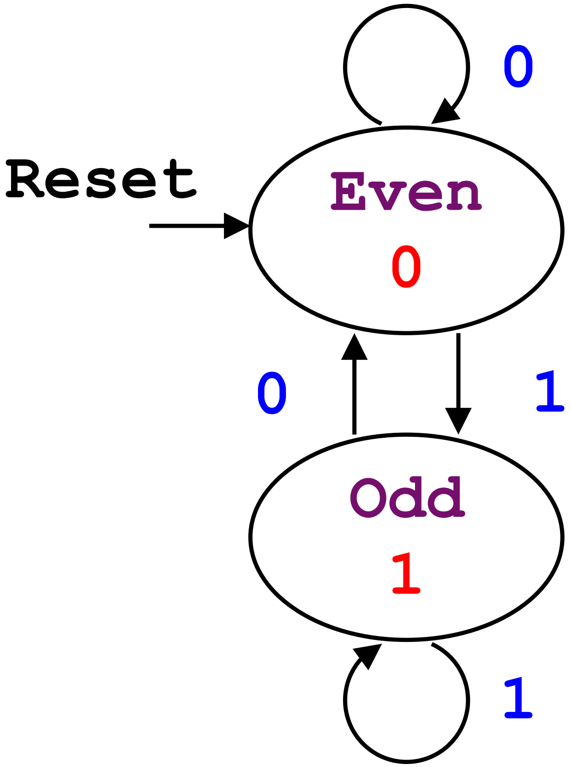
Outline

- Finite State Machine (FSM)
  - Clock Edge Detection
  - Feedback
- Use of Signals and Variables
  - Outside Process: Concurrent Statement
  - Inside Process: Sequential Statement
    - Combinational Process
    - Clocked Process
- Types of FSMs: Moore vs. Mealy
- Practical Examples
  - Up/Down Counter
  - Pattern Generator

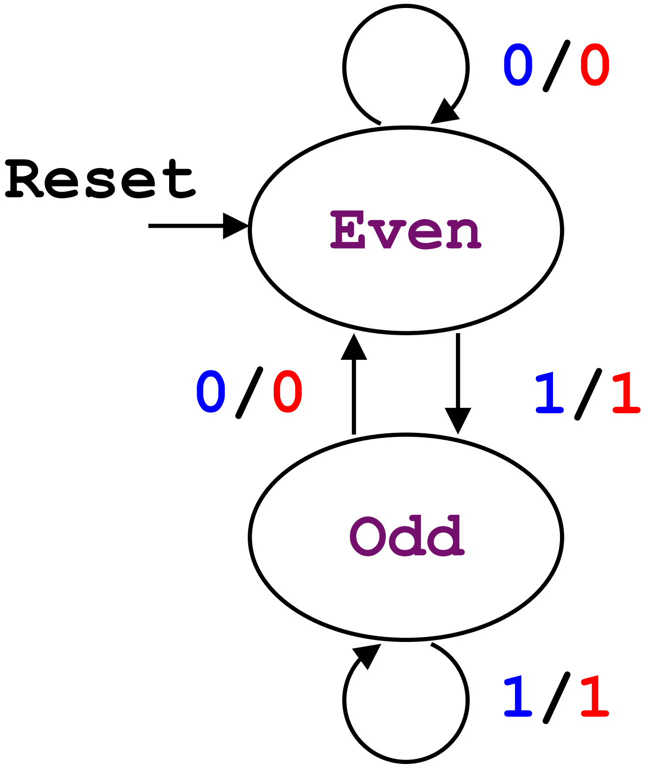
Types of Finite State Machines

- **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

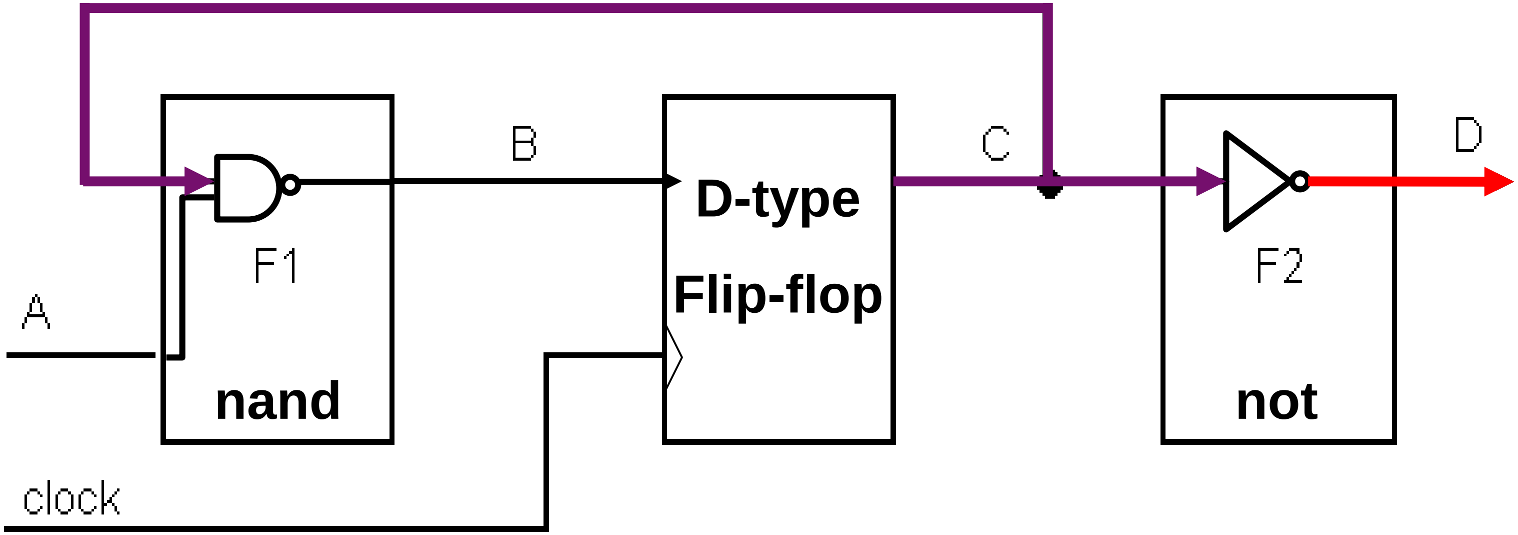


https://www.slideshare.net/mirfanjum1/moore-and-mealy-machines-29553482

Moore Machine (1/3)

- **Moore Machine**:
  - Outputs are a function of the present state ***only***.
- An Example of Moore Machine:

```
F1: C <= not (A and C); -- "<=" is a flip-flop
F2: D <= not C; -- Moore Machine
```



B is the current output of not(A and C), but B does not need to exist.

Writing **`C <= not(A and C)`** ***is enough.***

Moore Machine (2/3)

- The simplest Moore machine uses only one process:

```
architecture moore_arch of system is
signal C: bit; -- state
begin
   D <= not C; -- combinational logic   ← F2
   process - sequential logic
   begin
     wait until rising_edge(clock);       ← F1
    C <= not (A and C); -- flip-flop
   end process;
end moore_arch;
```

Moore Machine (3/3)

- Using two processes is flexible and easier to design.

```
process (C) -- combinational
begin
  D <= not C; -- Moore Machine
end process;
```

← F2

```
process (clock, reset) -- sequential
begin
  if  reset = '1' then c <= '0';
  elsif rising_edge(clock) then
    C <= not (A and C); -- flip-flop
  end if;
end process;
```

← F1

Class Exercise 5.9

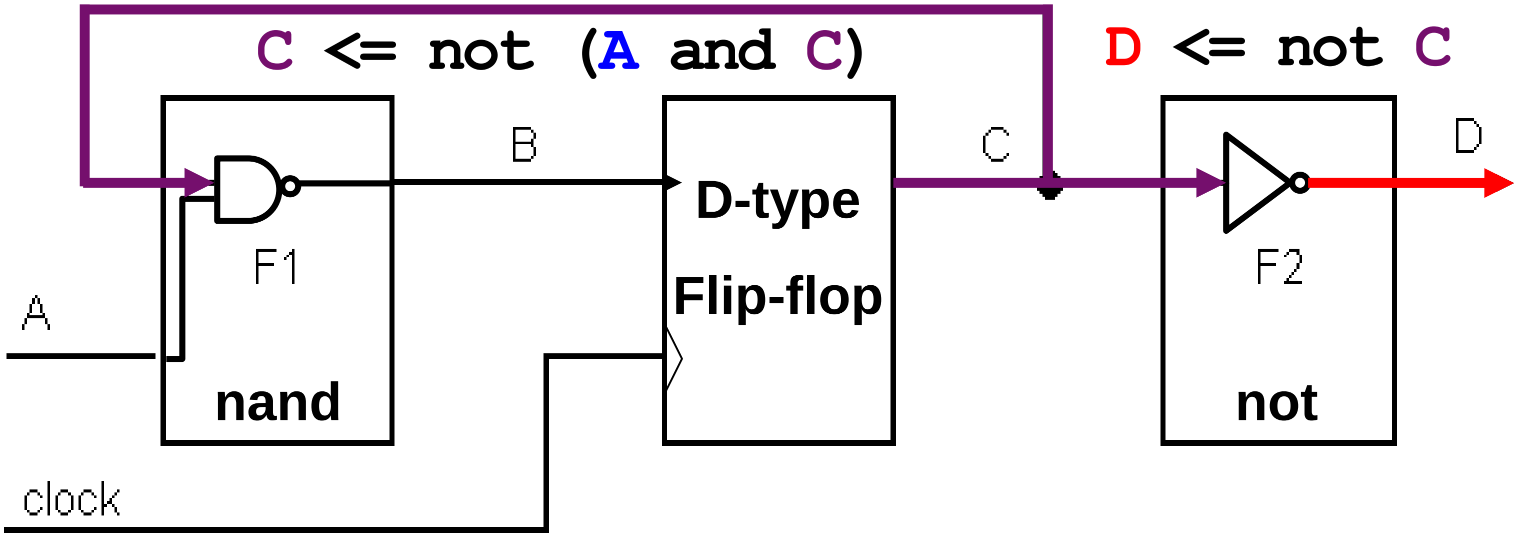
Student ID: ______________ Date:

Name: ____________________ ________

- Draw the waveform of **C** (initially **C=0**)

`C <= not (A and C)`

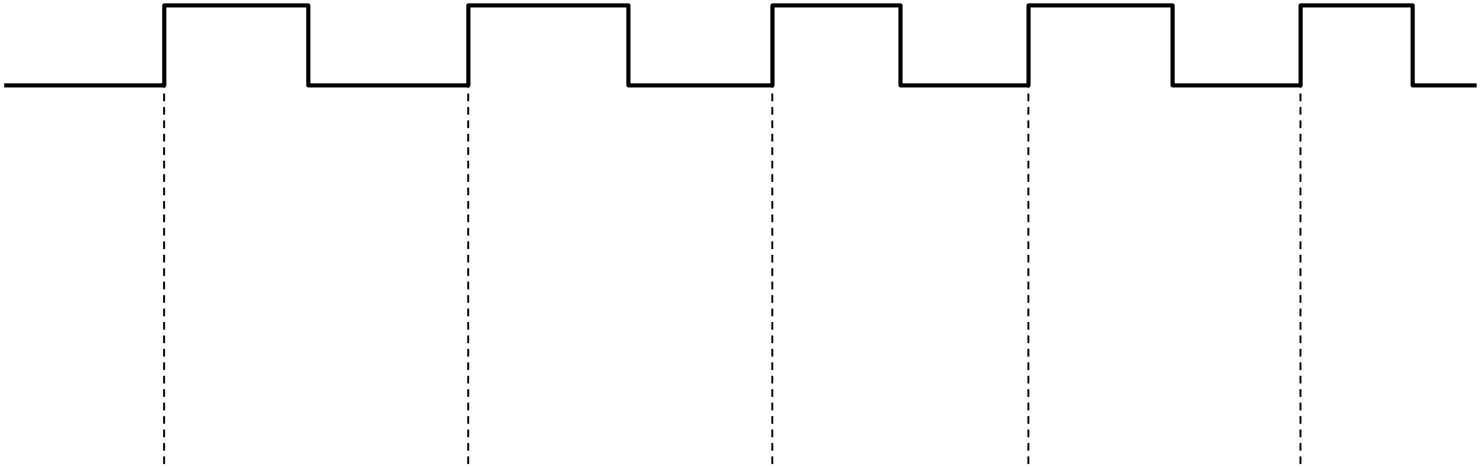
`D <= not C`



clock

C=/D
when A=1

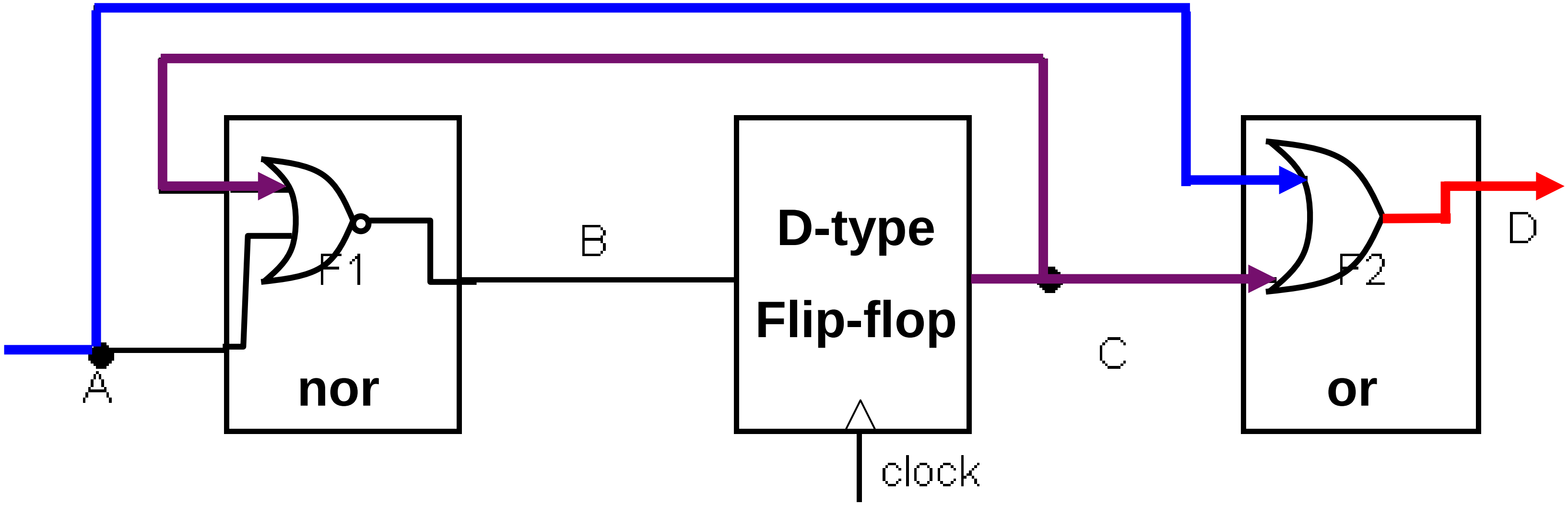
C=/D
when A=0



Mealy Machine (1/2)

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

```
F1: C <= not(A or C); -- "<=" is a flip-flop
F2: D <= (A or C); -- Mealy Machine
```



B is the current output of not(A or C), but B does not need to exist.

Writing `C <= not(A or C)` *is enough.*

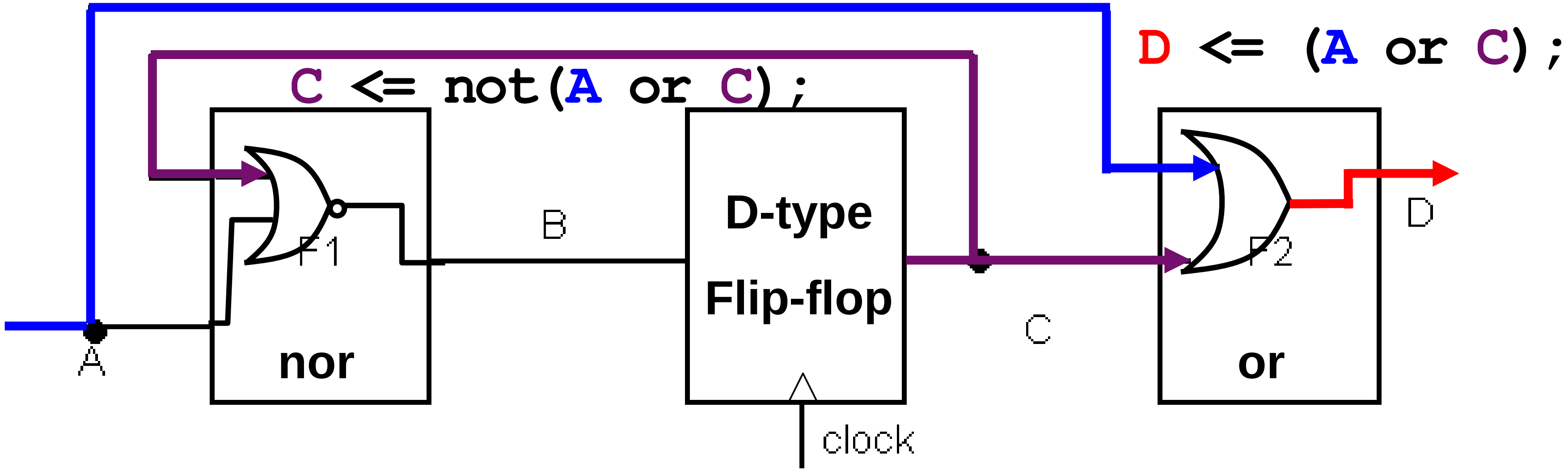
Mealy Machine (2/2)

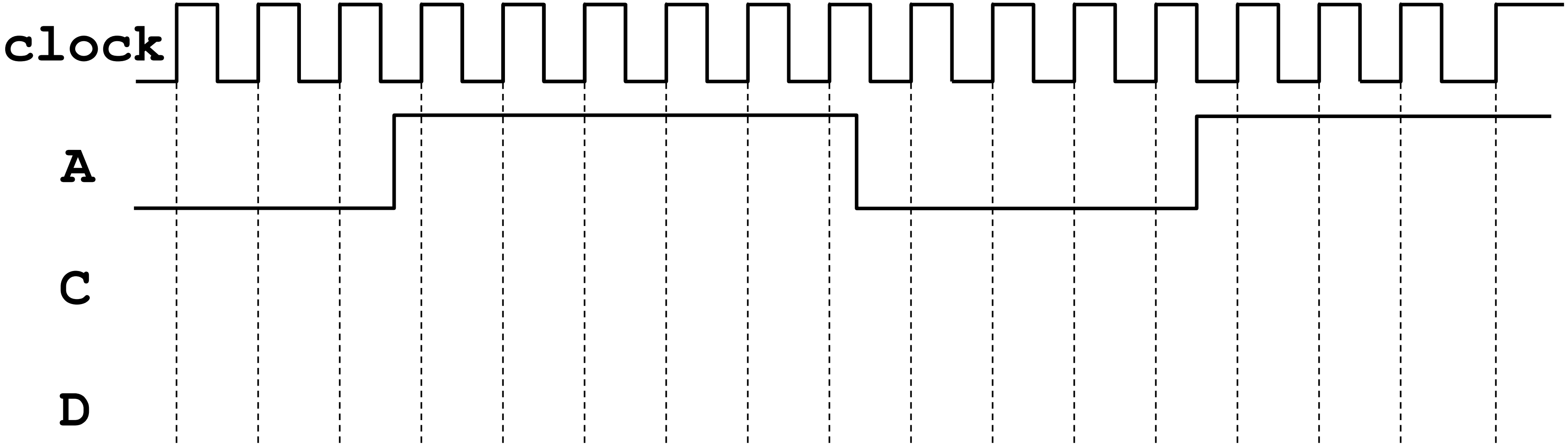
```
architecture mealy_arch of some_entity is
signal  C: std_logic;
begin
  process (A,C) -- combinational logic
  begin
    D <= (A or C); -- Mealy Machine                ← F2
  end process;
  process(clock,reset) -- sequential logic
  begin
    if reset = '1' then c <= '0';
    elsif rising_edge(clock) then                  ← F1
      C <= not(A or C); -- flip-flop
    end if;
  end process;
 end mealy_arch;
```

Class Exercise 5.10

Student ID: ____________ Date:
Name: __________________ ________

- Draw the waveforms of `C` and `D` (initially `C=0`)

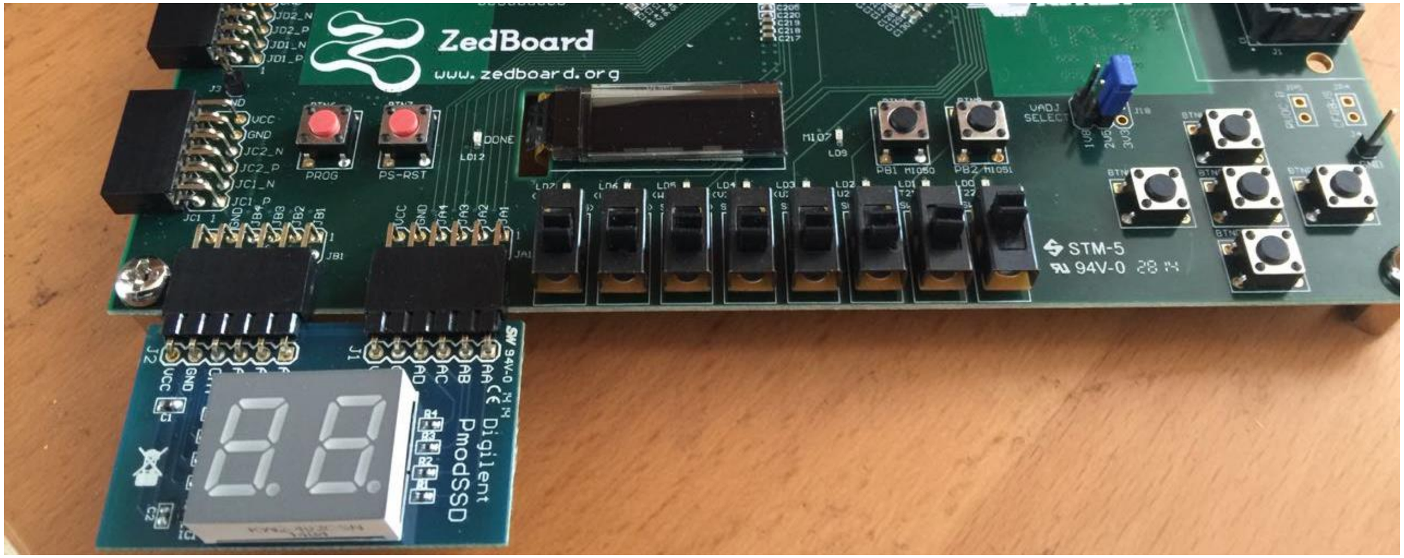




VHDL Coding Tips and Styles

- Separating the combinational and sequential logics.
  - Try to use at least two processes: one contains all combinational logic & the other contains all sequential logic.
- Keeping each process as simple (small) as possible.
  - Try to partition a large process into multiple small ones based on the signals in the sensitivity list.
- Putting every signal that your process needs to know about changes to be in the sensitivity list.
- Avoiding assigning a signal from multiple processes.
  - It may cause the "multi-driven" issue.

Recall: What we have done in Lab06



```
entity sevenseg is
port(     clk : in STD_LOGIC;
       switch : in STD_LOGIC_VECTOR (7 downto 0);
          btn : in STD_LOGIC;
          ssd : out STD_LOGIC_VECTOR (6 downto 0);
       ssdcat : out STD_LOGIC );
end sevenseg;
```

underline: external I/O pins

- *Task1: Display the input number (XY) in hexadecimal*
- *Task2: Count down from the input number (XY) to (00)*

Example: Lab06 (1/2)

```
-- generate 1ms and 1s clocks
process(clk)
begin
  if rising_edge(clk) then
    if (s_count = 49999999) then
      s_pulse <= not s_pulse;
      s_count <= 0;
    else
      s_count <= s_count + 1;
    end if;
    if (ms_count = 49999) then
      ms_pulse <= not ms_pulse;
      ms_count <= 0;
    else
      ms_count <= ms_count + 1;
    end if;
  end if;
end process;

-- read button (combinational)
process(btn)
begin
  if rising_edge(btn) then
    counter_en <= not counter_en;
  end if;
end process;
```

```
-- count down
process(s_pulse)
begin
  if rising_edge(s_pulse) then
    if (counter_en = '1') then
      if(counter=0) then
        counter <= to_integer(
                     unsigned(switch));
      else
        counter <= counter - 1;
      end if;
    end if;

  counter_vec <= std_logic_vector(
                   to_unsigned(counter,8));
  end if;
end process;
```

```
-- update the seven segment display
process(ms_pulse)
  … -- see the next page
end process;
```

```
-- output ssd (combinational)
process(digit)
 … -- see the next page
end process;
```

Example: Lab06 (2/2)

```
-- generate 1ms and 1s clocks
process(clk)
begin
  if rising_edge(clk) then
    if (s_count = 49999999) then
      s_pulse <= not s_pulse;
      s_count <= 0;
    else
      s_count <= s_count + 1;
    end if;
    if (ms_count = 49999) then
      ms_pulse <= not ms_pulse;
      ms_count <= 0;
    else
      ms_count <= ms_count + 1;
    end if;
  end if;
end process;

-- read button (combinational)
process(btn)
begin
  if rising_edge(btn) then
    counter_en <= not counter_en;
  end if;
end process;
```

```
-- count down
process(s_pulse)
  … -- see the previous page
end process;
-- update the seven segment display
process(ms_pulse)
begin
  if ms_pulse='1' then
    if(counter_en = '1') then
    digit <= counter_vec(7 downto 4);
    else
    digit <= switch(7 downto 4);
    end if;
  else
    if(counter_en = '1') then
    digit <= counter_vec(3 downto 0);
    else
    digit <= switch(3 downto 0);
    end if;
  end if;
  ssdcat <= ms_pulse; -- select display
end process;
-- output ssd (combinational)
process(digit)
begin
  case digit is
    when "0000" => ssd <= "1111110";
    …
  end case;
end process;
```

Outline

- Finite State Machine (FSM)
  - Clock Edge Detection
  - Feedback
- Use of Signals and Variables
  - Outside Process: Concurrent Statement
  - Inside Process: Sequential Statement
    - Combinational Process
    - Clocked Process
- Types of FSMs: Moore vs. Mealy
- Practical Examples
  - Up/Down Counter
  - Pattern Generator

Example 1) Up/Down Counter

- **Up/Down Counters**: Generate a sequence of gradually increasing or decreasing counting patterns according to the clock and inputs.
  - **Synchronous Clock**: *All clock inputs of state registers (i.e., flip-flops) are connected.*
    - More complex to design
    - More logic
    - Less time delay at outputs
  - **Asynchronous Clock**: *The output of one state register (i.e., flip-flop) is the clock of another state register.*
    - Easier to design
    - Less logic
    - More time delay at outputs

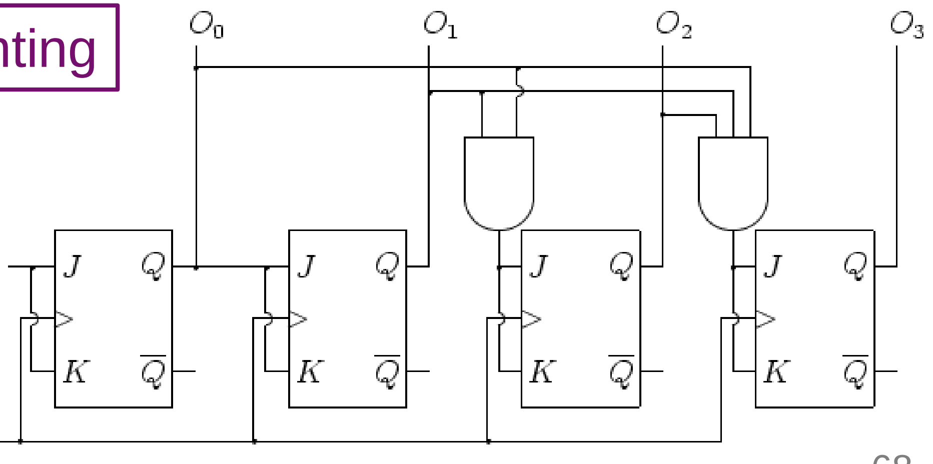
4-Bit Sync. Clock Down Counter

```
entity sync_counter is
  port(CLK: in std_logic;
       RESET: in std_logic;
       COUNT: inout std_logic_vector(3 downto 0));
end sync_counter ;
architecture sync_counter_arch of sync_counter is
begin
  process(CLK, RESET)
  begin
    if(RESET = '1') then COUNT <= "0000";
    else
      if( rising_edge(CLK) ) then
        COUNT <= COUNT - 1;
      end if;
    end if;
  end process;
end sync_counter_arch;
```

synchronous clock, asynchronous reset

All clock inputs of state registers (i.e., flip-flops) are connected.

counting



http://web.cs.mun.ca/~paul/cs3724/

4-Bit Async. Clock Down Counter

Asynchronous Clock Counter
(*a.k.a.* Ripple Counter)

```
entity async_counter is
  port(CLK: in std_logic;
       RESET: in std_logic;
       COUNT: inout std_logic_vector(3 downto 0));
end async_counter ;
architecture async_counter_arch of async_counter is
begin
process(RESET, CLK, COUNT(0), COUNT(1), COUNT(2))
begin
  if RESET ='1' then
    COUNT <= "0000";
  else
    if(rising_edge(CLK))       then COUNT(0)<=not COUNT(0); end if;
    if(rising_edge(COUNT(0))) then COUNT(1)<=not COUNT(1); end if;
    if(rising_edge(COUNT(1))) then COUNT(2)<=not COUNT(2); end if;
    if(rising_edge(COUNT(2))) then COUNT(3)<=not COUNT(3); end if;
  end if;
end process;
end async_counter_arch;
```

Asynchronous clocks

The output of one state register is the clock of another state register.

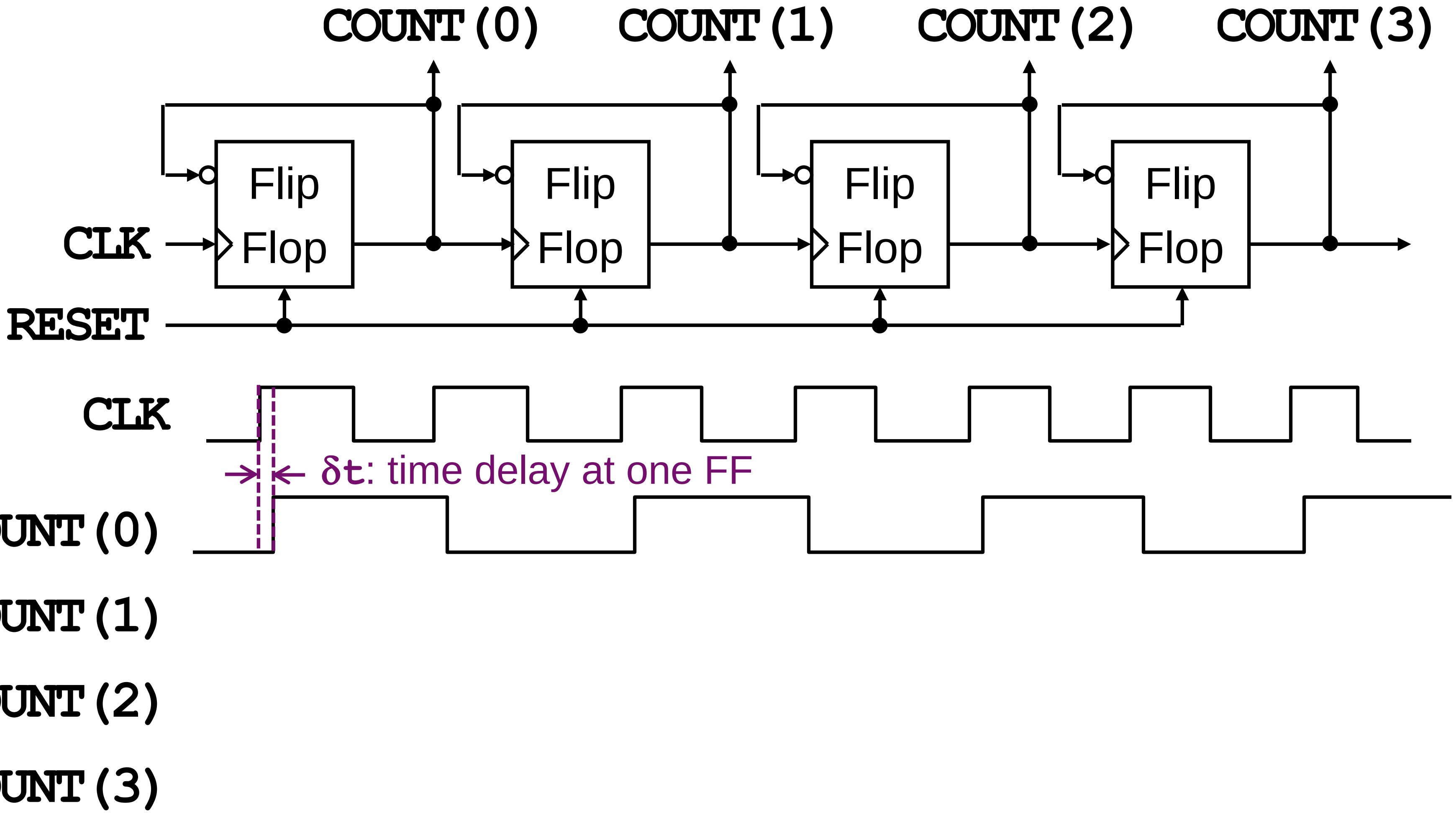
Each assignment is a flip-flop

Class Exercise 5.11

Student ID: ___________ Date:

Name: _______________ ________

- Draw the waveforms of **COUNT(0)** ~ **COUNT(3)** to show the time delays of a 4-bit async. clock counter:



Example 2) Pattern Generator

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

- 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-lops 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*

Class Exercise 5.12

Student ID: __________ Date:

Name: ________________ ________

- How many states can a 4-bit counter have?
  Answer: ______________________________

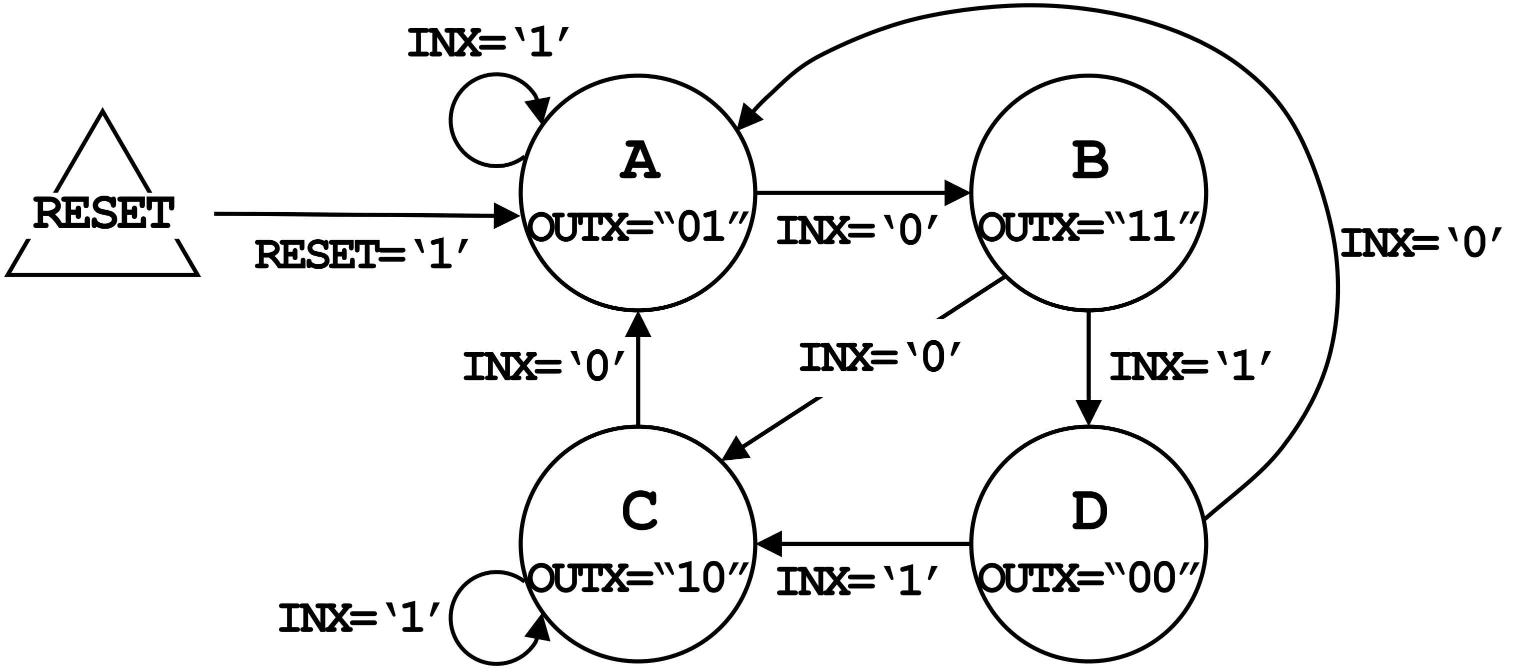
- Under binary and one-hot encoding schemes, how many bits for the state registers are required, respectively, if you need:

	Binary	One-Hot
4 States		
9 States		
21 States		

Class Exercise 5.13

Student ID: ____________ Date:
Name: ________________ ________

- 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**.
- Write the complete VHDL program for the design.
- Is this a Moore or Mealy Machine?

Class Exercise 5.13

Student ID: ___________ Date:

Name: ______________ _______

```
library IEEE;
use IEEE.std_logic_1164.all;
entity ex is port(
  RESET,CLOCK,INX: in STD_LOGIC;
  OUTX: out STD_LOGIC_VECTOR(1 downto 0));
end x7e;
architecture ex_arch of ex is
type state_type is (A,B,C,D);
signal s: state_type;
begin
process(CLOCK, RESET) -- sequential
begin
  if _____________________ then
    s <= _;
    OUTX <= "__";
  elsif __________________ then
    case s is
    when A =>
      if INX = '_' then s <= _;
      else s <= _; end if;
    when B =>
      if INX = '_' then s <= _;
      else s <= _; end if;
    when C =>
      if INX = '_' then s <= _;
      else s <= _; end if;
    when D=>
      if INX = '_' then s <= _;
      else s <= _; end if;
    end case;
  end if;
end process;
process(s) -- combinational
begin
  case s is
  when A => OUTX <= "__";
  when B => OUTX <= "__";
  when C => OUTX <= "__";
  when D => OUTX <= “__";
end process;
end ex_arch;
```

Moore or Mealy?

Summary

- Finite State Machine (FSM)
  - Clock Edge Detection
  - Feedback
- Use of Signals and Variables
  - Outside Process: Concurrent Statement
  - Inside Process: Sequential Statement
    - Combinational Process
    - Clocked Process
- Types of FSMs: Moore vs. Mealy
- Practical Examples
  - Up/Down Counter
  - Pattern Generator