

# Digital Design with VHDL

---



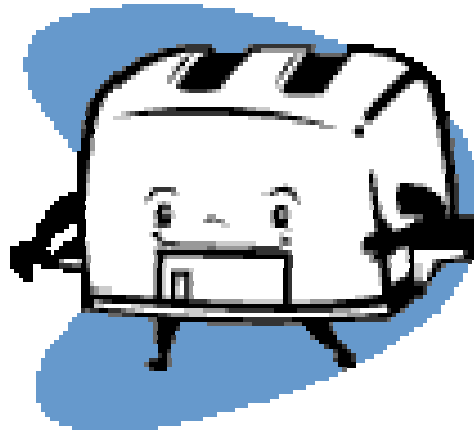
Sequential Structures

**By Benjamin Abramov**

All Rights reserved ©

No duplication copying and distribution of this document is allowed without the proper authorization of the author.

# Processes



# Processes

*Process – a behavioral region in the architecture where statements are executed in sequence.*

Syntax of process declaration:

```
label: process ( sensitivity list ) is  
    --declaration of process resources  
    begin  
        -- sequential statements;  
    end process label;
```



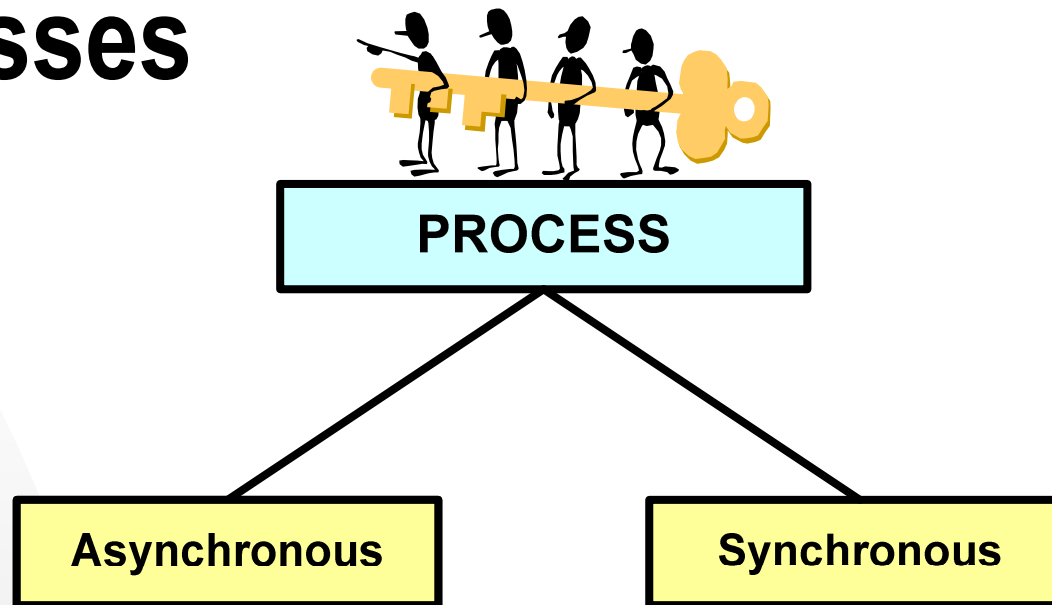
# Processes

The **process** declarative part defines local items for the **process** and may contain declaration of:

- subprograms
- types
- subtypes
- constants
- variables
- files
- aliases
- use clauses
- groups



# Processes



## **Asynchronous processes:**

**Describe a combinatorial logic hardware.**

- mux,
- decoders,
- encoders,
- arithmetic devices

## **Synchronous processes:**

**Describe a “clocked” (Memory) logic.**

- FF,
- registers,
- shift registers,
- counters

# Processes

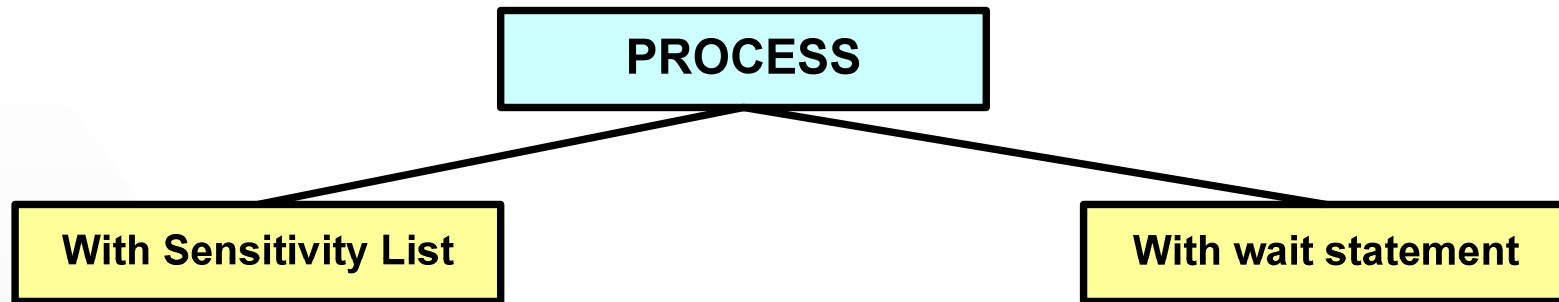
A process declaration may contain an optional *sensitivity list*. The list contains identifiers of signals to which the **process** is sensitive. A change of value by any of those signals causes the suspended process to resume.

Alternatively, process activation and suspension may be controlled via a wait statement.

*Sensitivity list and explicit wait statements may not be specified in the same process.*



# Processes



```
warn_process : process (tempr, max_v) is  
begin  
    alert <= (tempr >= max_v);  
end process warn_process;
```

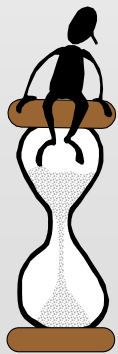
```
warn_process : process  
begin  
    alert <= (tempr >= max_v);  
    wait on tempr, max_v;  
end process warn_process;
```

# Processes



## Processes rules:

- **Statements are executed sequentially**
- **Execution time is zero !!!**
- **All signals within the process— are updated at its end**



# Processes

architecture arc\_test of test is

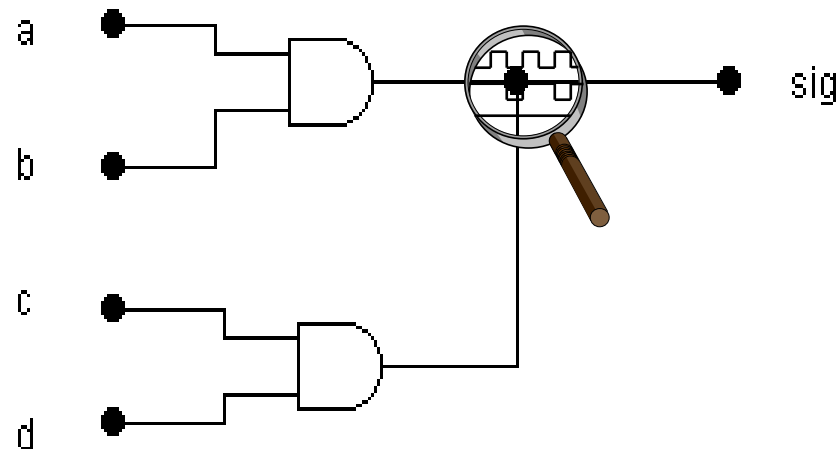
begin

sig <= a and b;

sig <= c and d; -- wrong !!! unresolved contention

end architecture arc\_test;

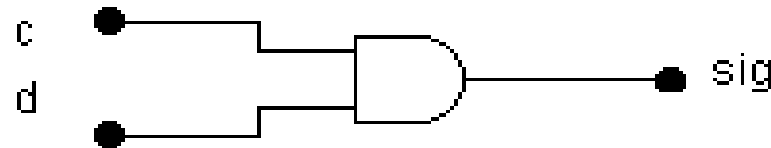
- The above code will pass simulation compile, resulting with ('X' – when values differ)
- but !!! It will **not** pass synthesis compile.



# Processes

Acceptable VHDL code!

```
architecture arc_test of test is  
begin  
  and_gate : process (a, b, c, d) is  
    begin  
      sig <= a and b;  
      sig <= c and d;  
    end process and_gate;  
end architecture arc_test;
```



- The above is a correct assignment yet the designer should be aware that only the last assignment will take place.
- (i.e. – sig <= c and d; -- The synthesizer will be implement this line.
- while, sig <= a and b; -- This line is ignored and treated as if not existing at all.)

# Processes

- Don't use in a combinatorial process a signal that is driven by that process, ( The result of such an act is a feedback loop).

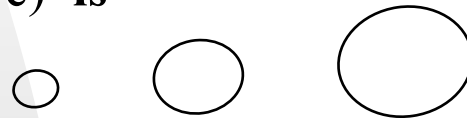
**process** (a, b, c) **is**

**begin**

temp <= (a **xor** b) **xor** temp;

s <= temp **xor** c;

**end process** ;



If *temp* is used inside of the process, then *temp* should be included into sensitivity list, but if *temp* is listed in the sensitivity list, it feeds itself !??

# Processes

Combinatorial processes , a.k.a. asynchronous processes must have a sensitivity list containing **all** the signals that the process uses (i.e. – use - read, compared, assigned from).

The process will always update the signals that it assigns when there is an event on any of the signals in the sensitivity list.

**A missing signal in the sensitivity list can lead to an unintentional latch !!!**



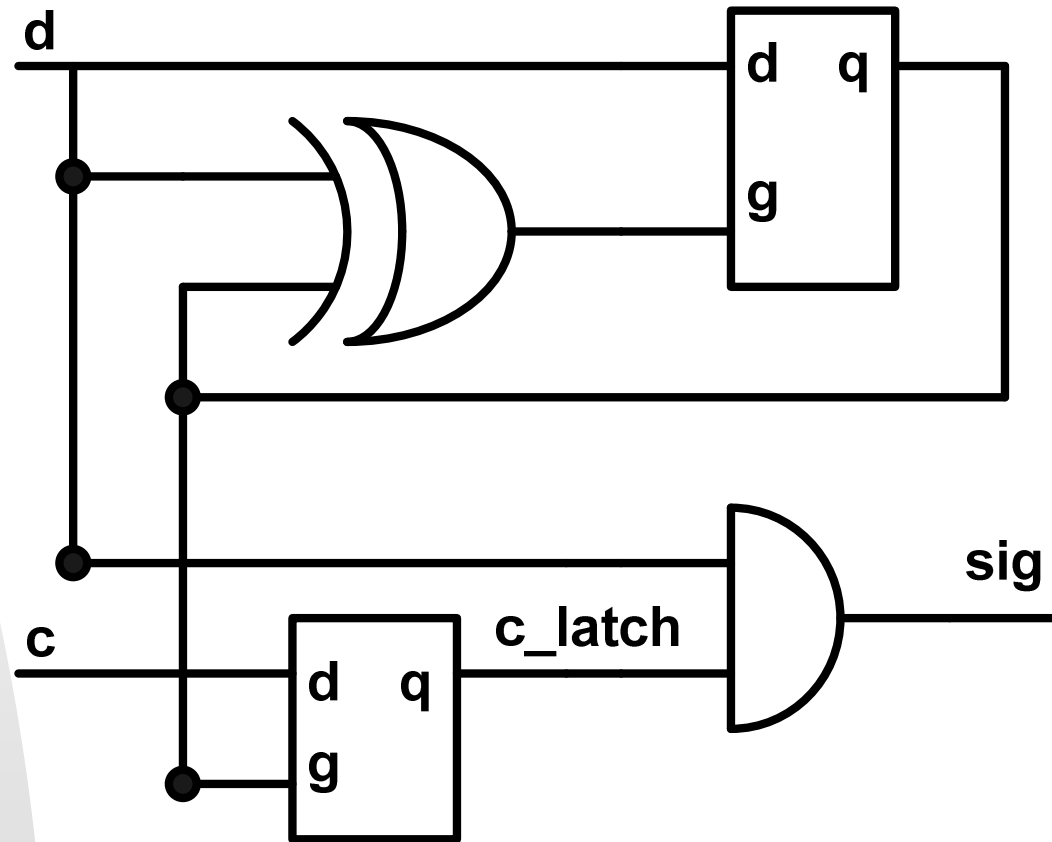
# Processes

```
architecture arc_and_gate of and_gate is
begin
    process (d) is
    begin
        sig <= c and d;
    end process;
end architecture arc_and_gate;
```

Signal **c** is not included in the process sensitivity list, therefore the process does not respond when **c** changes its value.

So what get is the following circuit:

# Processes



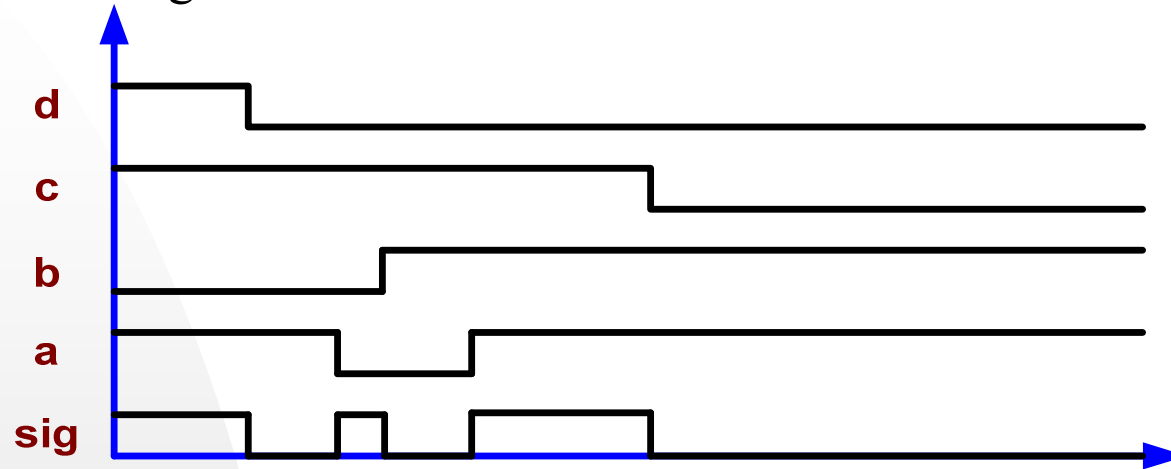
# Processes

Another example of uncompleted sensitivity list : in this case the output values (c, d) are not included in the sensitivity list.

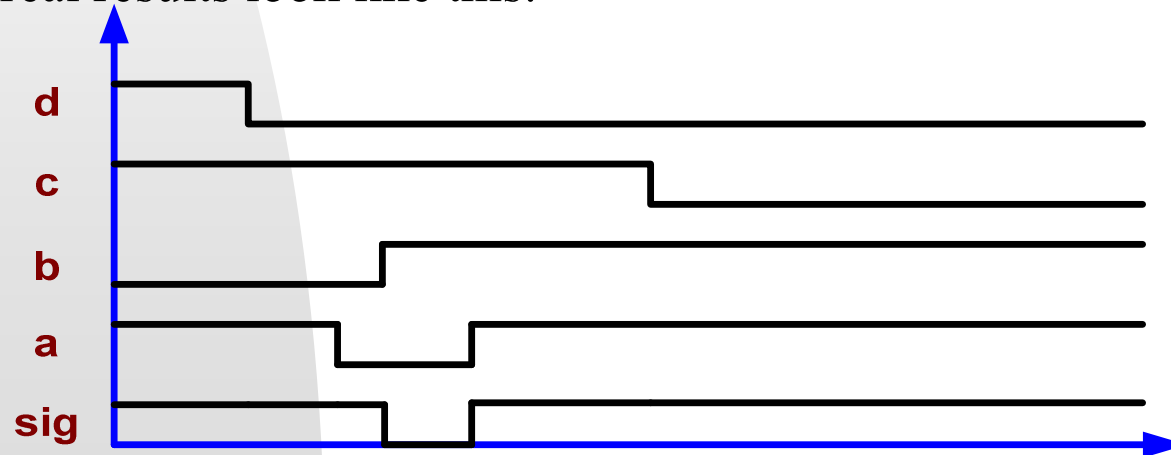
```
process(a, b) is  
begin  
  if (a = b) then  
    Sig <= c;  
  else  
    Sig <= d;  
  end if;  
end process;
```

# Processes

The design if coded correctly was expected to behave like the following wave form :



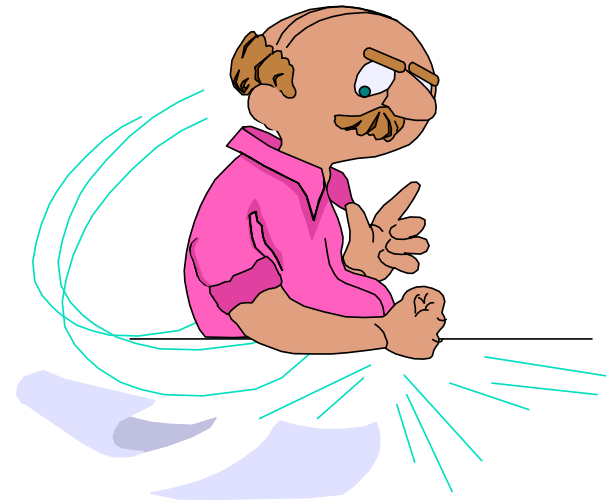
But real results look like this:



# Processes

Most common mistakes when using an asynchronous processes are:

- Uncompleted sensitivity list.
- Redundant signals included into sensitivity list.
- Redundant logic is described.



# Processes

```
entity comp is
port( a   : in  std_logic_vector(3 downto 0);
      b   : in  std_logic_vector(3 downto 0);
      aeb : out boolean;
      agb : out boolean;
      alb : out boolean );
end entity comp;
```

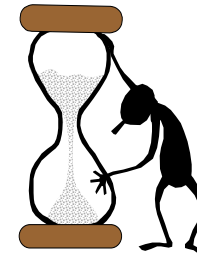
```
architecture arc_comp of comp is
begin
    process(a, b) is
    begin
        aeb <= (a = b);
        agb <= (a > b);
        alb <= (a < b);
    end process;
end architecture arc_comp;
```

- A complete and correct example of an asynchronous comparator.

# Processes

Synchronous process – current output is sequenced by a clock (rising edge or falling edge) and depends on past inputs when there is no change in clock and current inputs when the appropriate clock event occurs. Such a process in actuality is describing a memory structure such as :-

- flip-flops
- registers
- counters
- shift registers
- state machines



- ✓ Sensitivity list include only clock and asynchronous signals such as reset/preset.
- ✓ All the logic must be assigned after clock edge ( rising or falling)

# Processes

Clock change detecting:

- By the predefined attribute - **event** :

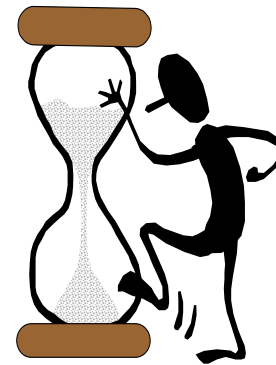
`clk'event and clk='1'` -- rise detect

`clk'event and clk='0'` -- fall detect

- By library functions:

`rising_edge(arg : std_logic)` return **boolean**

`falling_edge(arg : std_logic)` return **boolean**

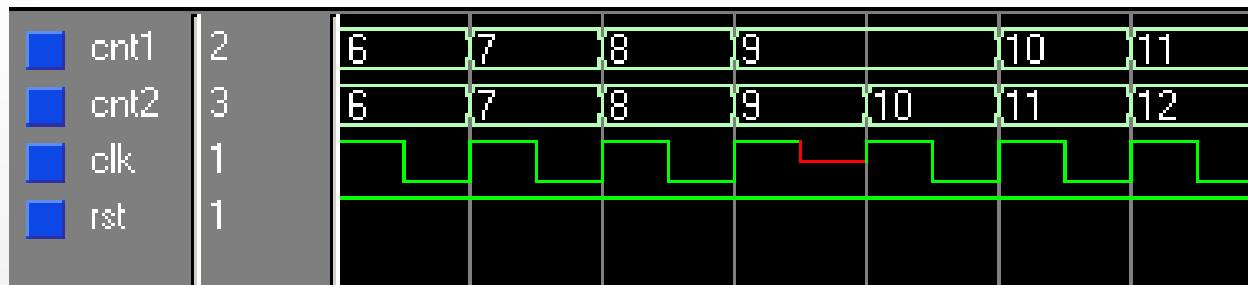


At simulation with type **std\_logic** use of functions **rising\_edge** and **falling\_edge** is more preferable.

# Processes

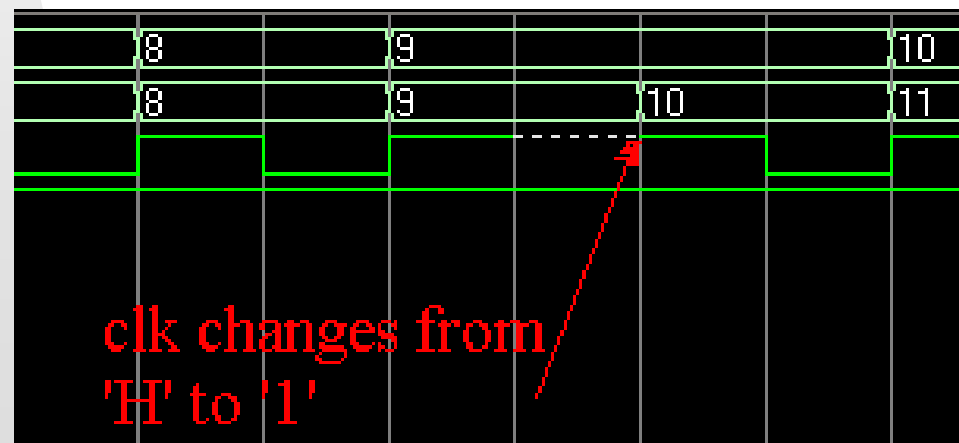
Clock changes from **uncertain** value to '1' when both counters are equal to 9.

- Cnt1 does not change its value. Cnt1 is described using the function **rising\_edge**.
- Cnt2 does change its value. Cnt2 is described using **clk'event and clk='1'**.



**clk'event and clk='1'** returns **TRUE** at change to '1' from any other value.

**rising\_edge** function returns **TRUE** only on recognition of a transition from '0' to '1'.



# Processes

Example :

a = 1

b = 2

c = 3

d = 1 and d changes from 1 to 2

**process**(clk) **is**

**begin**

**if** **rising\_edge**(clk) **then**

a <= d;

b <= a + 1;

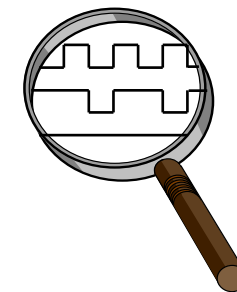
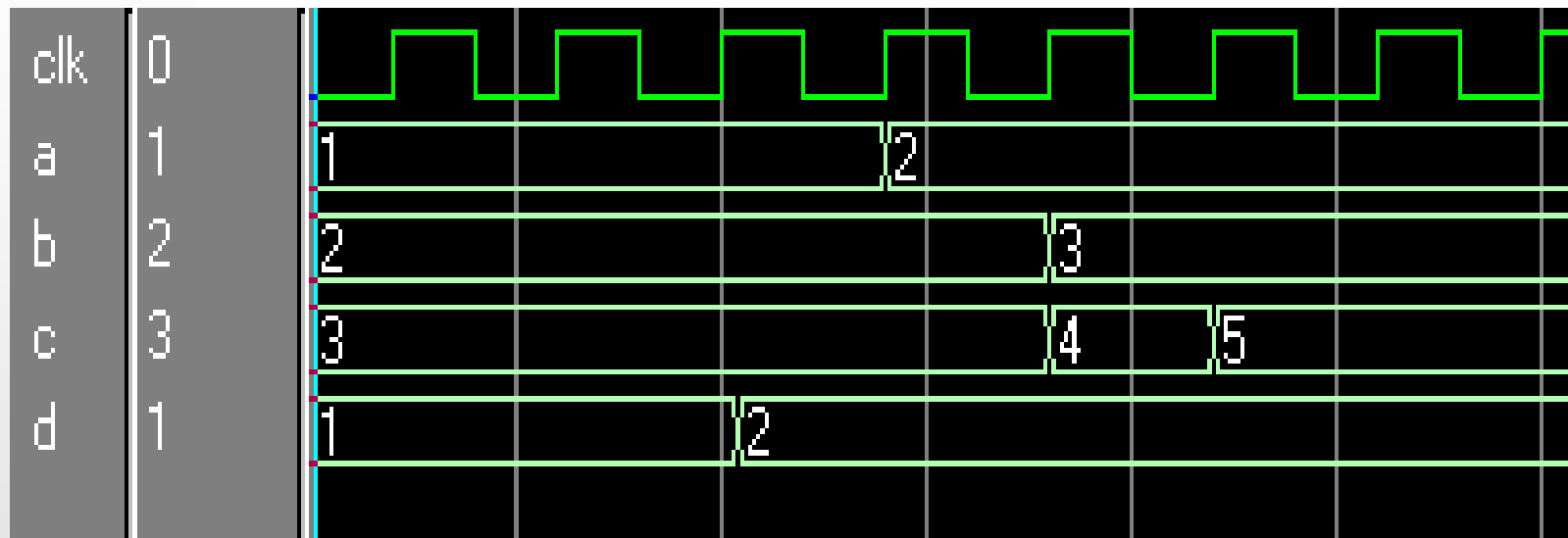
c <= a + b;

**end if;**

**end process;**

# Processes

Assignment behavior of signals in the previously described synchronous process.



# Processes

DFF with asynchronous reset (active high);

**entity dff is**

```
    port( clk: in std_logic;  
          rst : in std_logic;  
          d  : in std_logic;  
          q  : out std_logic );
```

**end entity dff;**

**architecture arc\_dff of dff is**

**begin**

```
    process(clk, rst) is
```

```
        begin
```

```
            if (rst= '1') then
```

```
                q<='0';
```

```
            elsif rising_edge(clk) then
```

```
                q<=d;
```

```
            end if;
```

```
        end process;
```

```
    end architecture arc_dff;
```



# Processes

DFF with synchronous reset (active high)

-- only clock signal should be declared in the sensitivity list, reset is synchronous.

```
process (clk) is -- begin  
    if rising_edge(clk) then  
        if (rst = '1') then  
            q <= '0';  
        else  
            q <= d;  
        end if;  
    end if;  
end process;
```

# Processes

```
set_rst_dff : process(clk, set, rst) is
  begin
    if (rst = '0') then
      q <= '0';
    elsif (set = '0') then
      q <= '1';
    elsif rising_edge(clk) then
      q <= d;
    end if;
  end process set_rst_dff;
```

# Processes

The typical errors of synchronous processes:

- Using different signals as clock instead of the global clock
- Both set and reset assigned to the same signal and can be active at the same time.
- Signal do not have a reset value.
- Two edges of clock are used in the process.
- Gated clock is created.
- More then one clock is used in a process.

# FF coding rules



# FF coding rules

badDFFstyle : process(clk) is

begin

if rising\_edge(clk) then

if (rst\_n='0') then

q1<='0';

else

q1 <= d;

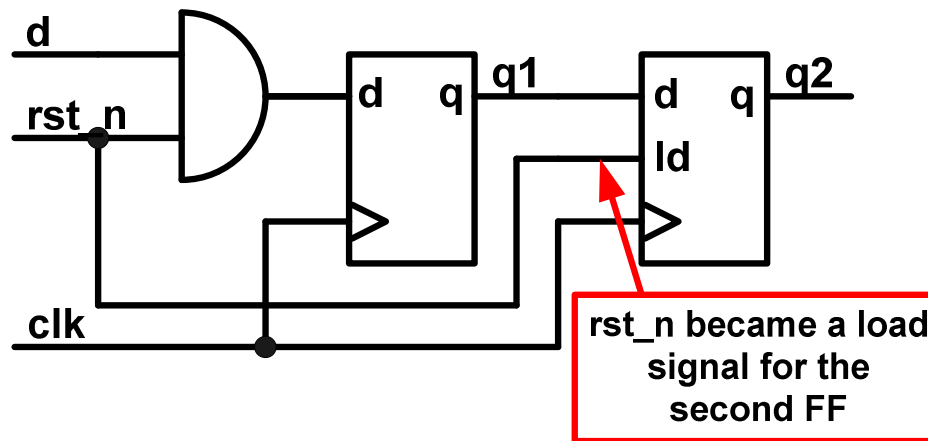
q2 <= q1;

end if;

end if;

end process badDFFstyle ;

**Bad VHDL coding style to model dissimilar FF's.**



# FF coding rules

**goodDFFstyle :process(clk) is**

**begin**

**if rising\_edge(clk) then**

**q2 <= q1;**

**if (rst\_n = '0') then**

**q1 <= '0';**

**else**

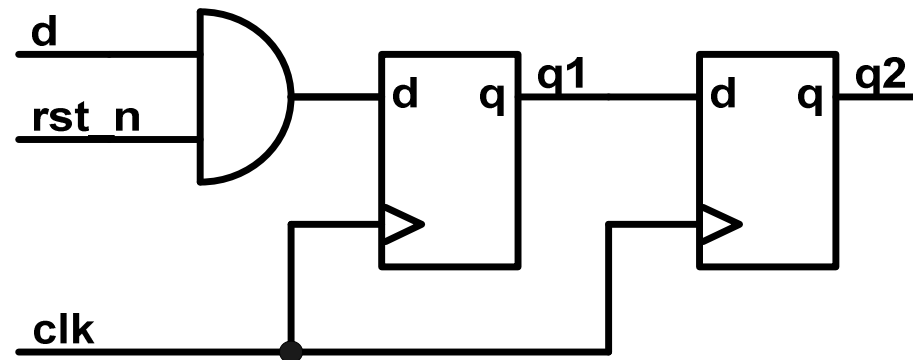
**q1 <= d;**

**end if;**

**end if;**

**end process goodDFFstyle ;**

**Good VHDL coding style to model dissimilar FF**

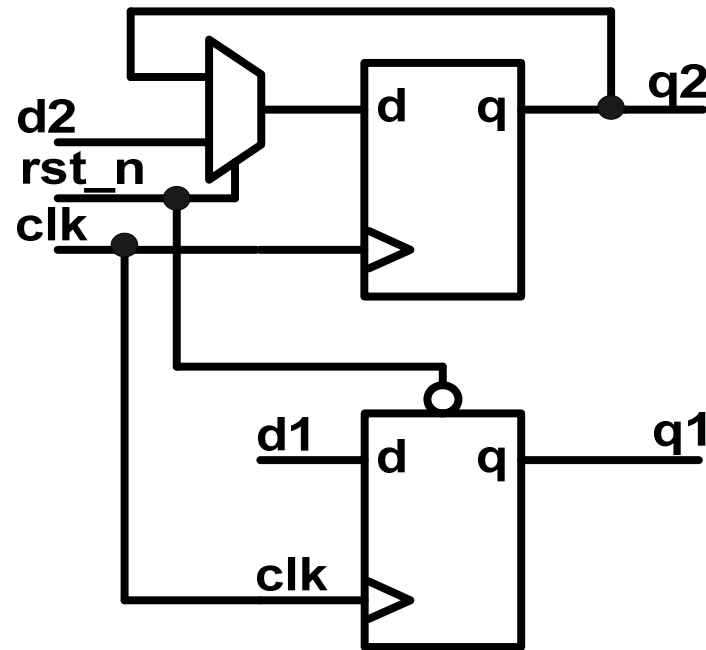


# FF coding rules

Some designers code all registers in a block in the same process.

If some registers require reset and some do not, the following code can result:

```
TwoReg : process(clk, rst_n) is
begin
  if (rst_n = '0') then
    q1 <= '0' ;
  elsif rising_edge(clk) then
    q1 <= d1 ;
    q2 <= d2 ;
  end if ;
end process TwoReg;
```



**This code results in extra logic being created.  
The extra logic (multiplexer) maintains the value of Q2 during reset.**

# FF coding rules

Good coding style, split to two separate processes :

```
first : process( clk, rst_n) is
```

```
begin
```

```
  if (rst_n = '0') then
```

```
    q1 <= '0' ;
```

```
  elsif rising_edge(clk) then
```

```
    q1 <= d1 ;
```

```
  end if ;
```

```
end process first ;
```

```
second:process(clk) is
```

```
begin
```

```
  if rising_edge(clk) then
```

```
    q2 <= d2;
```

```
  end if;
```

```
end process second;
```

# FF coding rules

Mixed Synchronous and Asynchronous processes

```
bad_style: process (clk, rst, ld, load_val) is
```

```
begin
```

```
  if (rst = '0') then
```

```
    cnt <= 0;
```

```
  elsif rising_edge(clk) then
```

```
    cnt <= cnt + 1;
```

```
  elsif (ld = '1') then
```

```
    cnt <= load_val;
```

```
  end if;
```

```
end process bad_style;
```

**Legal VHDL but  
non-deterministic  
synthesis result !!!**

# FF coding rules

Asynchronous load in synchronous process

dangerous\_style: **process** (clk, ld) **is**

**begin**

**if** (ld = '0') **then**

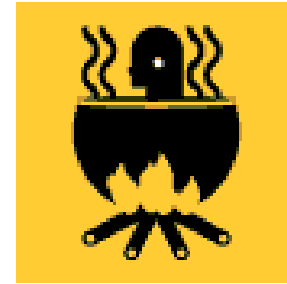
cnt <= load\_val;

**elsif** **rising\_edge** (clk) **then**

cnt <= cnt + 1;

**end if;**

**end process** dangerous\_style;



**Legal VHDL when load\_val is a constants value .**

**Otherwise non-deterministic synthesis result !!!**

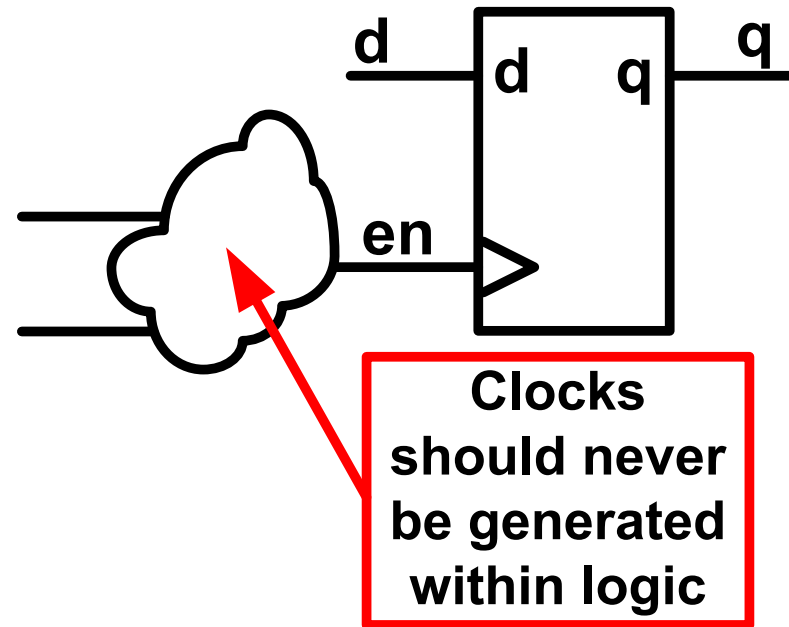
# FF coding rules

The following is an example of bad design practice which will result in gated clock.

```
p_reg: process ( clk, en) is
begin
    if (en = '1') then
        if rising_edge (clk) then
            q <= d;
        end if;
    end if;
end process p_reg;
```

# FF coding rules

```
en<= '0' when state = '0' else
    '1' when state = '1' else
    '0';
p_reg : process (en) is
begin
    if rising_edge(en) then
        q <= d;
    end if;
end process p_reg;
```



# FF coding rules

Gated clock examples:

①

```
process (clk) is
```

```
.....
```

```
if rising_edge (clk) and (en = '1') then
```

```
  --Signal assignment
```

```
end proces...
```

②

```
clk_internal <= clk and en;
```

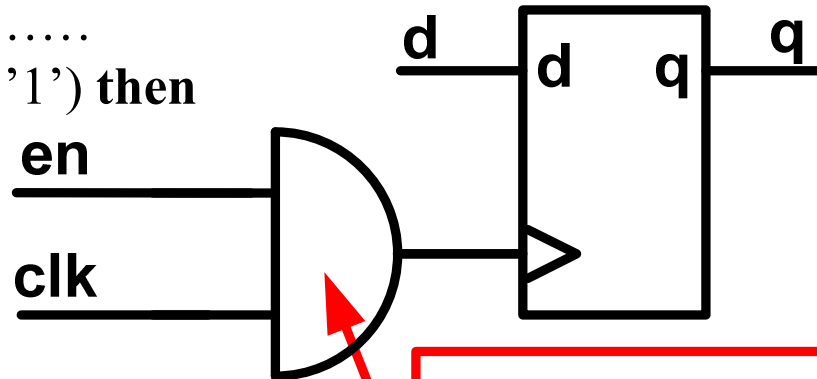
```
-----  
process (clk_internal) is
```

```
.....
```

```
if rising_edge (clk_internal) then
```

```
  --Signal assignment
```

```
end proces...
```



**Gated Clocks  
result in clock  
skew and timing  
violations**



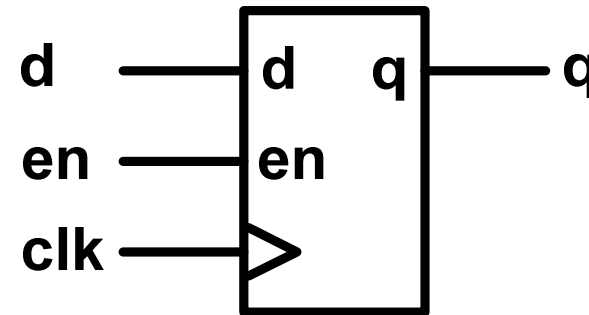
# FF coding rules

The correct description for an enabled FF:

Correct\_proc : proces...

.....

```
if rising_edge(clk) then
    if (en = '1') then
        -- signal assignment
```



# FF coding rules

Is the following a register or a latch?

```
reg_or_latch_proc : process (clk) is
```

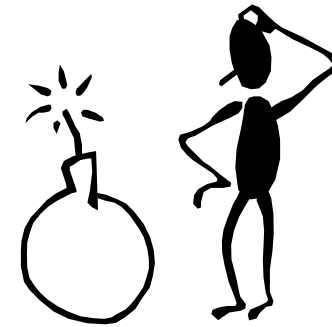
```
begin
```

```
  if (clk = '1') then
```

```
    q <= d;
```

```
  end if;
```

```
end process reg_or_latch_proc;
```



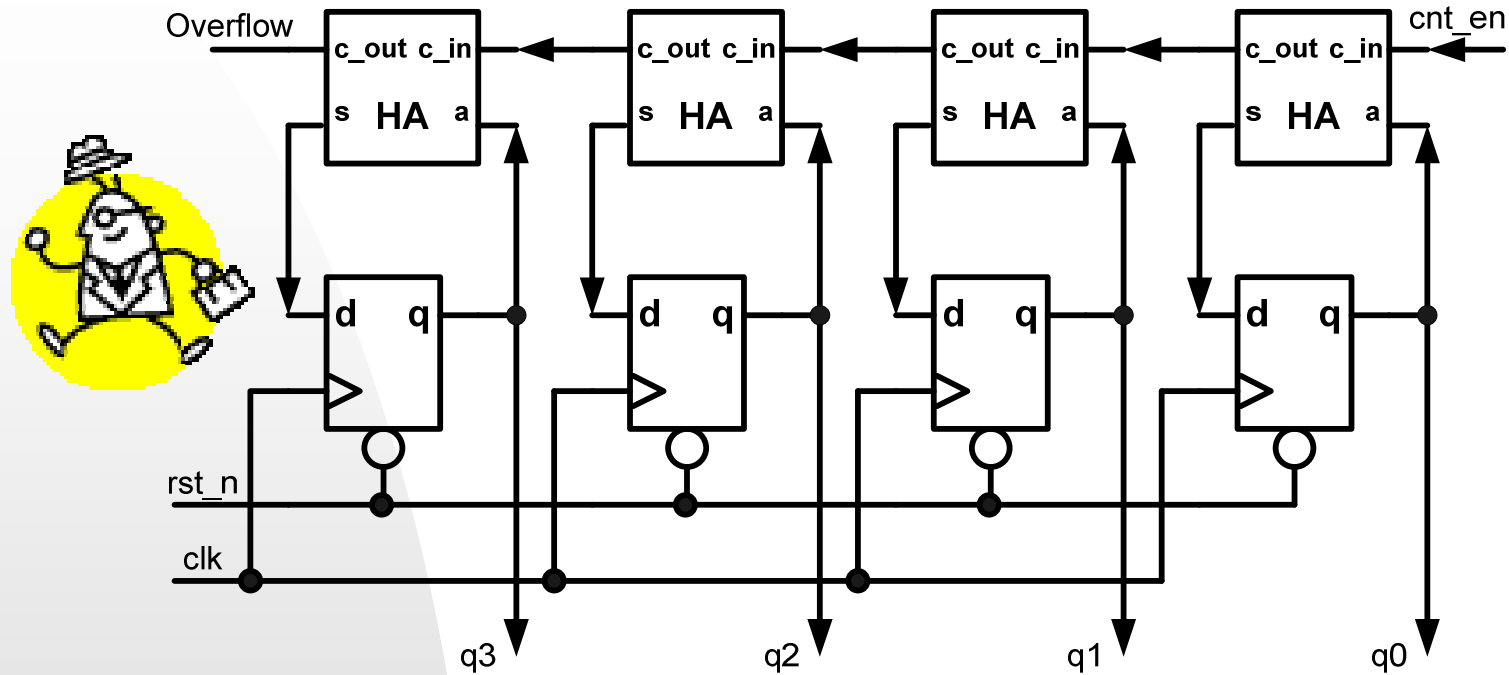
# FF coding rules

- ✓ Cannot be a latch since it has an incomplete sensitivity list
- ✓ Cannot be a register since has no clock edge specification

**Resolution :**

**Synthesis tool should produce an error and not produce any results.**

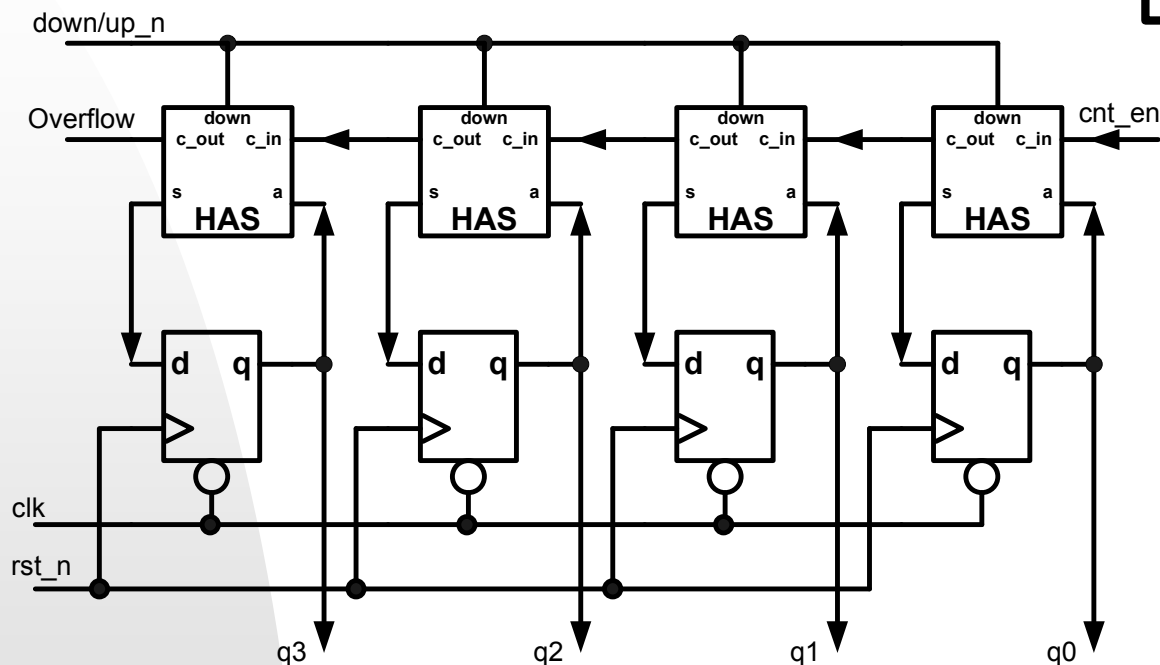
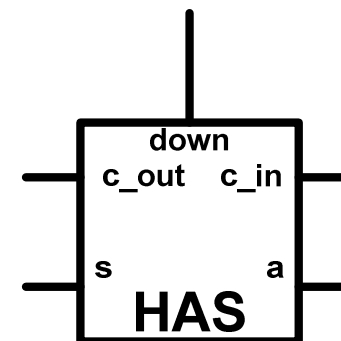
# Self Work



Describe the binary counter according to the above-stated circuit.  
Implement the Half Adder using gates.

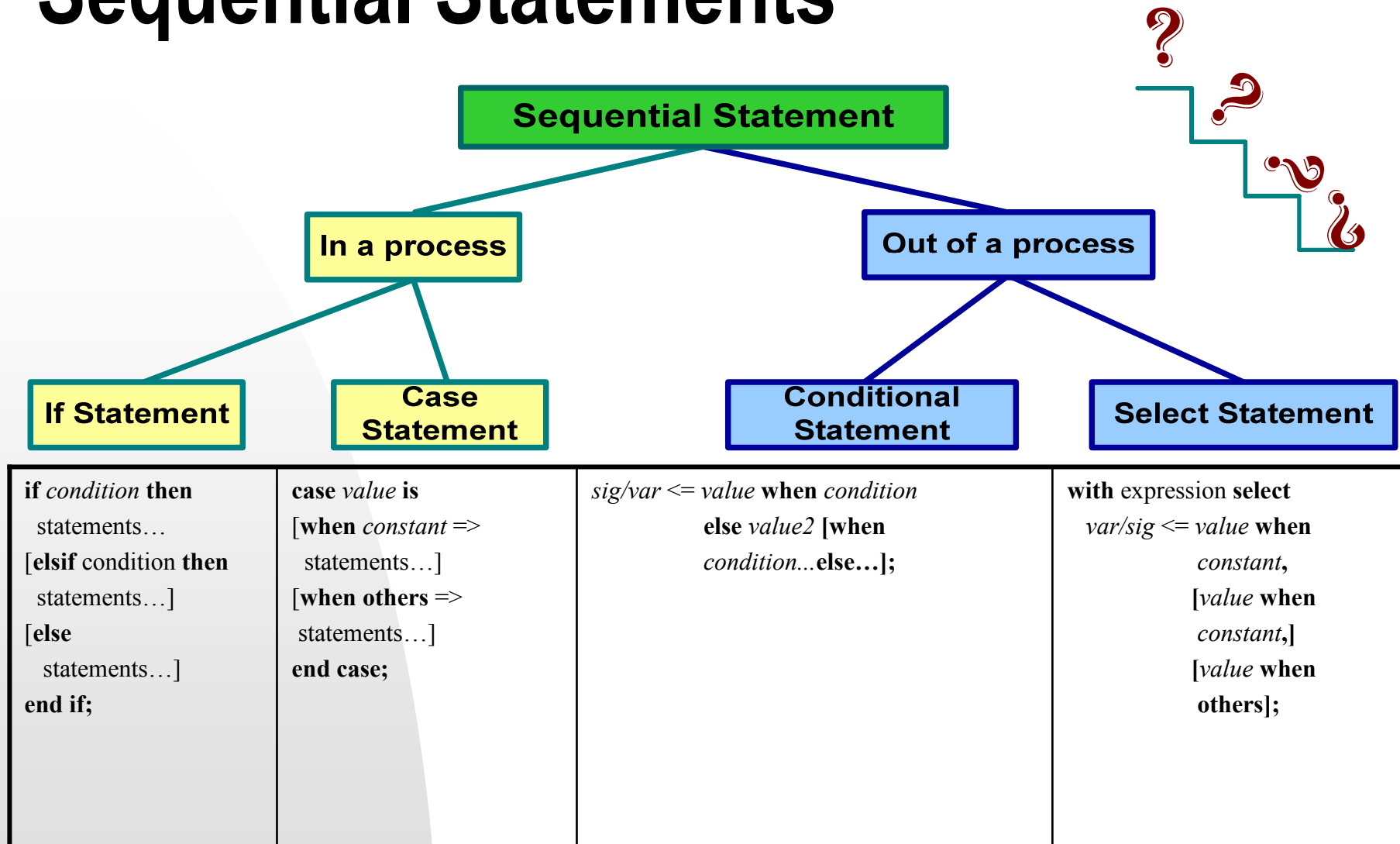
# Self Work

The half adder-subtractor



- Build the functions to implement a Half Adder-Subtraction model (using gates).
- Describe the binary up/down counter according to the above-stated circuit.

# Sequential Statements



# Sequential Statements

The *if statement* allows selection of statements to be executed based on one or more conditions.

If statement generates priority structure.

The syntax is:

```
if_statement ::=  
    if condition then  
        sequence_of_statements  
    { elsif condition then  
        sequence_of_statements }  
    [ else  
        sequence_of_statements ]  
    end if ;
```



**If Statement**

*The IF Statement must be enclosed in a process !!!*

# Sequential Statements

The conditions are expressions that evaluate to *boolean values*.

The conditions are evaluated successively until one found that yields the value *true* or until the end of the process is reached.

If an expression evaluates to *true* the statements in that branch are the only ones to be executed.

## If Statement

*The elsif and else clauses are optional.*



# Sequential Statements



## Rules:

- **Only one of the branches is executed if any.**
- **The order is important – the first true condition causes the sequential statements after it to be executed.**
- **Any following condition after the first true one is ignored.**
- **Any statement not in the true evaluated branch is ignored.**
- **The first condition has the highest priority.**

**If Statement**

# Sequential Statements

## IF Condition example :

If **a** equals '1' only the first branch will execute due to the priority structure.

```
process (a, b, c, d) is  
begin
```

```
  if (a = '1') then
```

```
    res <= c;
```

```
  elsif (b = '1') then
```

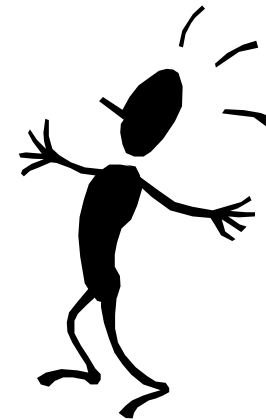
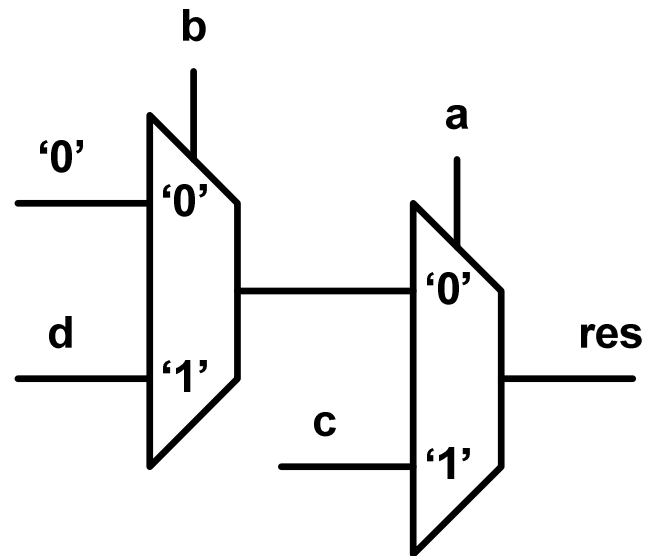
```
    res <= d;
```

```
  else
```

```
    res <= '0';
```

```
  end if;
```

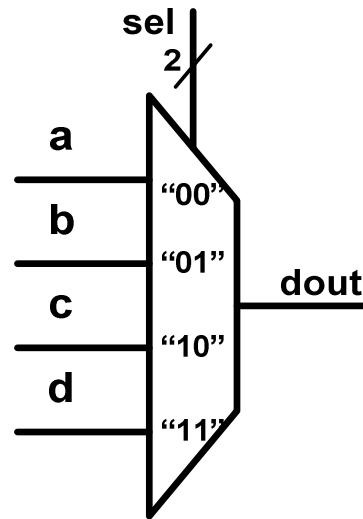
```
end process;
```



# Sequential Statements

The **if** statements will *not always create prioritized structures*.

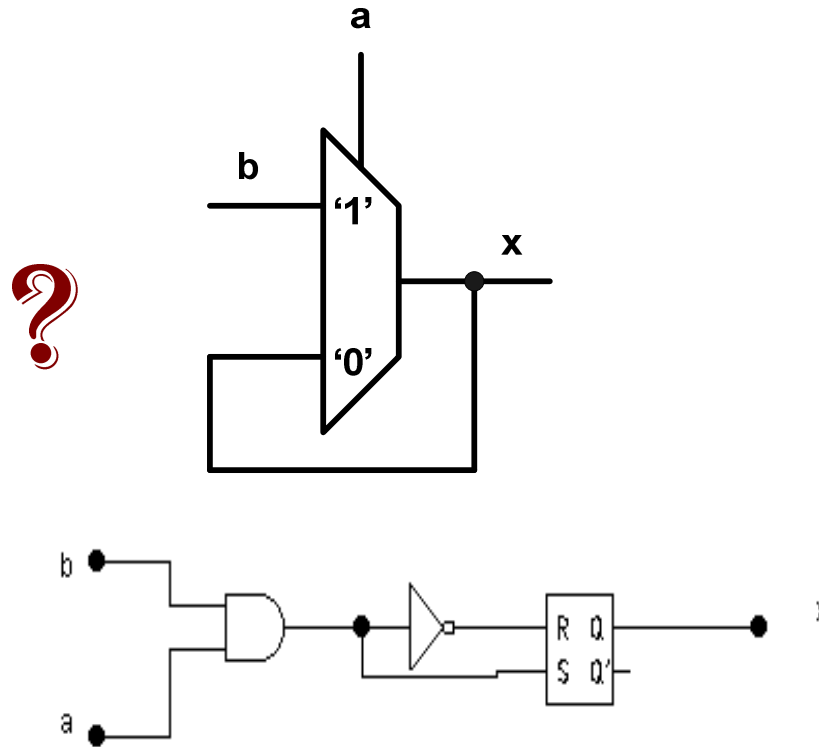
```
process (a, b, c, d, sel) is  
begin  
    if (sel = "00") then  
        dout <= a;  
    elsif (sel = "01") then  
        dout <= b;  
    elsif (sel = "10") then  
        dout <= c;  
    else  
        dout <= d;  
    end if;  
end process;
```



# Sequential Statements

Using an **if** statement without an **else** clause in a combinatorial process can result in latches being inferred, unless all signals driven by the process are given unconditional *default assignment*.

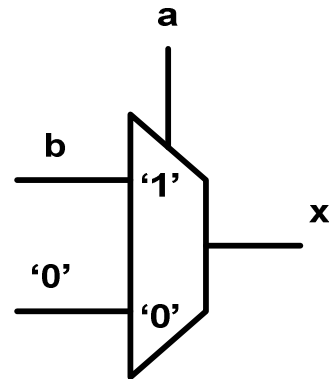
Example:  
**process** (a, b) **is**  
**begin**  
    **if** (a = '1') **then**  
        x <= b;  
    **end if**;  
**end process**;



# Sequential Statements

The complete if statement presented below, describes mux.

```
process (a, b) is
begin
  if (a = '1') then
    x <= b;
  else
    x <= '0';
  end if;
end process;
```



Same result gives the *default assignment described bellow*:

```
process (a, b) is
begin
  x <= '0'; -- default assignment latches prevented
  if (a = '1') then
    x <= b;
  end if;
end process;
```



# Sequential Statements

The same functionality as an if statement can be done in the dataflow environment (concurrent statement) can be accomplished using the **conditional signal assignment** statement:

```
signal a : integer ;  
signal output_signal, x, y, z : bit_vector (3 downto 0) ;  
....  
output_signal <= x when (a=1) else y when (a=2) else  
z when (a=3) else "0000" ;
```

**Conditional  
Statement**

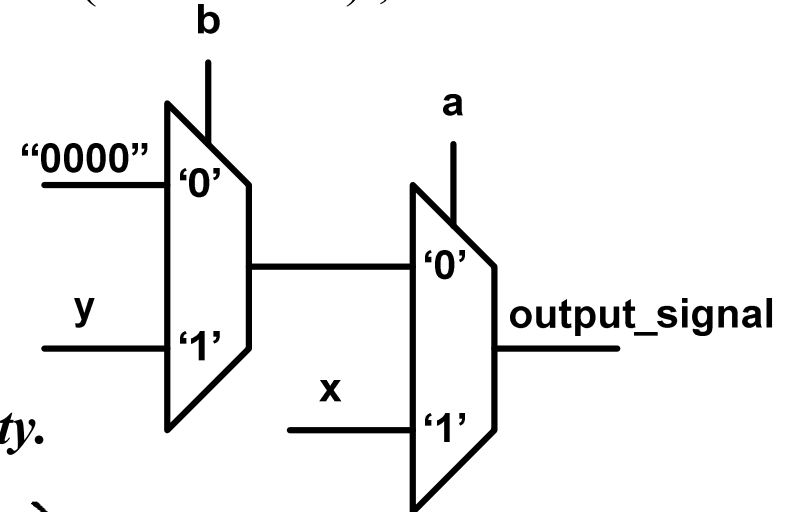
# Sequential Statements

```
signal a ,b: std_logic ;  
signal output_signal, x, y : std_logic_vector (3 downto 0) ;
```

....

```
output_signal <= x when (a = '1') else  
                y when (b = '1') else  
                "0000" ;
```

*The first condition gets the highest priority.*



**Conditional  
Statement**



# Sequential Statements

In **conditional signal assignment** statement:

- Conditions may overlap, as for the **if** statement.
- The expression corresponding to the first “true” condition is assigned, and the following statements are ignored.
- There must be a final unconditional else expression:

`y<= a when (x = 5); -- wrong`

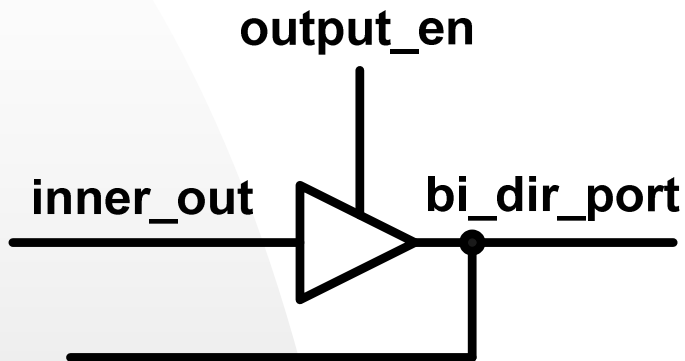
**Conditional  
Statement**



# Sequential Statements

Conditional signal assignment may be used to define tri-state buffers:

```
bi_dir_port <= inner_out when (output_en='1') else 'Z';
```



Conditional Statement

*If a signal is conditionally assigned to itself, latches may be inferred.*



# Sequential Statements

The case statement allows selection of statements to be executed depending on the value of the selecting expression.

If the selected value exist the corresponding statement list will be executed. Otherwise, if the others clause is present, its corresponding statement list will be executed.

## Case Statement

The case syntax is:

```
case_statement ::=
    case expression is
        case_statement_alternative
    { case_statement_alternative }
    end case;
```

***The CASE Statement must be enclosed in a process !!!***

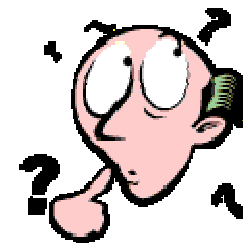


# Sequential Statements

```
case_statement_alternative ::=  
  when choices =>  
    sequence_of_statements
```

```
choices ::= choice { | choice }  
choice ::=
```

```
  simple_expression | discrete_range | element_simple_name | others
```



**Case  
Statement**

# Sequential Statements

The selection expression must result in either a discrete type, or a one-dimensional array of characters.

The alternative whose choice list includes the value of the expression is selected and the statement list executed.

Note : *All choices must be distinct, that is, no value may be duplicated.*

All values must be represented in the choice lists, or the special choice **others** must be included as the last alternative. If no choice list includes the value of the expression, the others alternative is selected. If the expression results in an array, then the choices must be strings or bit strings.

**Case  
Statement**

# Sequential Statements

case example :

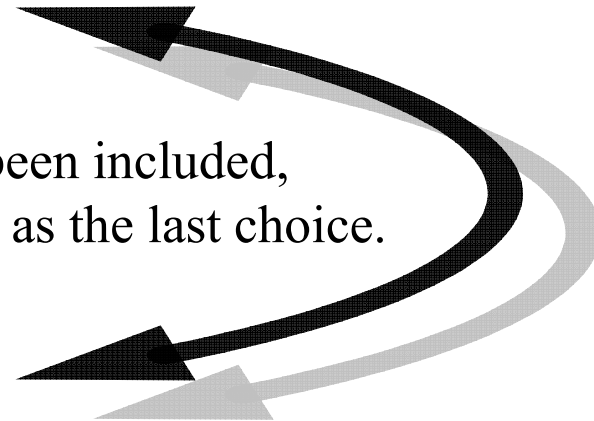
```
multiplexor : case sel is
  when "00" => dout <= a;
  when "01" => dout <= b;
  when "10" => dout <= c;
  when "11" => dout <= d;
end case multiplexor;
```

Sel is defined as a bit\_vector( 1 downto 0).

In the above example all possible choices have been included, otherwise the others clause must have been used as the last choice.

Multiplexor with others clause:

```
multiplexor: case sel is
  when "00"  => dout <= a;
  when "01"  => dout <= b;
  when "10"  => dout <= c;
  when others => dout <= d;
end case multiplexor;
```



# Sequential Statements

A range or selections may be specified as well as the '|' (or) choice:

```
case sel is
  when 0      => y <= a;
  when 1 to 4 => y <= b;
  when 5|7|9  => y <= c;
  when others => y <= d;
end case;
```

all values  
1, 2, 3, and 4  
are included

only values  
5, 7, 9  
are included



# Sequential Statements

A range may not be used with a vector type:

```
case sel is
  when "000"      => y <= a;
  when "001" to "100" => y <= b; -- wrong!
  when "101" | "110" => y <= c; -- correct
  when others     => y <= d;
end case;
```

**WRONG !!!**  
sel is defined as  
a vector type  
NOT as a range

**CORRECT !!!**  
The '|' (or - pipe)  
statement is legal also  
for vector type

Choice may not overlap :

```
case sel is
  when 0      => y <= a;
  when 1 to 4 => y <= b;
  when 3|5|7|9 => y <= c; -- error, 3 included into previous choice
  when others => y <= d;
end case;
```

**WRONG !!!**  
3 is included  
in more than  
one choice

# Sequential Statements

If many conditional clauses have to be performed on the same selection signal, a **case** statement is a better solution than the **if-then-else** construct.

A **case** statement with repeated assignments to a target signal, will synthesize to a large multiplexor with logic on the select inputs to evaluate the conditions for the different choices in the **case** statement branches.

*No priority* will be inferred from the order of the branches.

# Sequential Statements

Hardware with priority:

```
if (a = '1') then  
    dout <= c;  
elsif (b = '1') then  
    dout <= d;  
else  
    dout <= e;  
end if;
```

Hardware without priority:

```
ab <= a & b;  
.....  
case ab is  
    when "10" | "11" => dout <= c;  
    when "01"      => dout <= d;  
    when others   => dout <= e;  
end case;
```



# Sequential Statements

The **case** statement may be used with multiple targets, embedded **if** statements and nested **case** statement.

Example of mixed case/if process with multiple targets.

```
arbiter : process (addr, power_up, valid) is  
begin
```

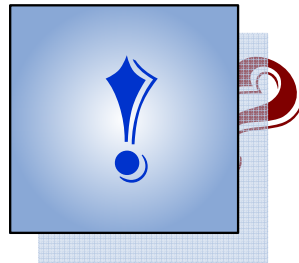
```
-- preventing transparent latches with default assignments
```

```
bios      <= '0';
```

```
flash    <= '0';
```

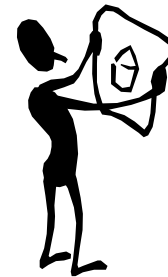
```
sram     <= '0';
```

```
eeprom   <= '0';
```



# Sequential Statements

```
if (valid = '1') then
  case addr is
    when 0 to 5000    =>
      if (power_up = '1') then
        bios <= '1';
      else
        flash <= '1';
      end if;
    when 5001 to 8000 =>
      sram <= '1';
    when 8001 to 9000 =>
      eeprom <= '1';
    when others       =>
      flash <= '1';
  end case;
end if;
end process arbiter;
```



# Sequential Statements

When a multiplexer is not a power of 2, some synthesis tools might implement a non-optimal logic. The solution is best taken care of by using the others clause.

```
EncodedMux : process (muxsel, a, b, c)
```

```
begin
```

```
    case muxsel is
```

```
        when "00" => y <= a;
```

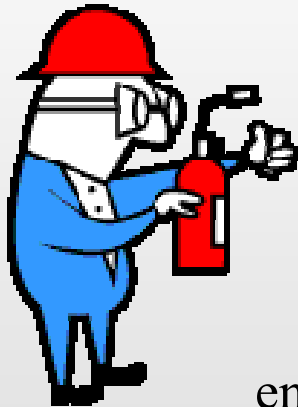
```
        when "01" => y <= b;
```

```
        when "10" => y <= c;
```

```
        when others => y <= (others => 'X');
```

```
    end case;
```

```
end process;
```



# Sequential Statements

The case statement can only be used in a sequential environment. In the dataflow environment (concurrent assignment), the **selected signal assignment** statement has the equivalent behavior:

```
signal output_signal, sel, x, y, z : std_logic_vector(3 downto 0) ;  
...  
with sel select  
output_signal <= x           when "0010",  
                           y           when "0100",  
                           z           when "1000",  
x and y and z when "1010" | "1100"|"0110",  
"0000"           when others ;
```

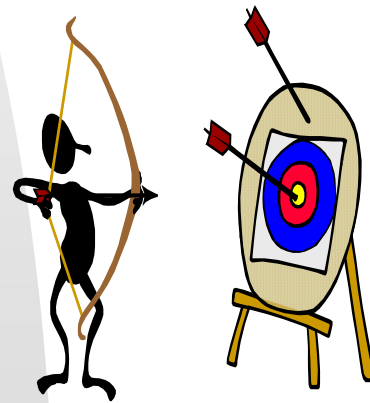
**Select Statement**

# Sequential Statement

The rules for the **selected signal assignment** are the same as for the case statement:

- All possible choices must be included, unless the others clause is used as the last choice.
- A range or a selection may be specified as a choice.
- Choices may not overlap.
- There is not implied priority to the choices.

A selected signal assignment will usually result in combinatorial logic being generated.



**Select Statement**

# Sequential Statements

Select statement and qualified expression:

```
entity full_adder is
port ( a, b, cin : in bit;
      s, cout  : out bit);
end entity full_adder;

architecture truth_table of full_adder is
begin
  with bit_vector'(a, b, cin) select
    (s, cout) <= bit_vector'("00") when "000",
               bit_vector'("10") when "001",
               bit_vector'("10") when "010",
               bit_vector'("01") when "011",
               bit_vector'("10") when "100",
               bit_vector'("01") when "101",
               bit_vector'("01") when "110",
               bit_vector'("11") when others;
end architecture truth_table;
```

# Sequential Statements

**entity mux is**

```
    port (din : in  std_logic_vector(3 downto 0);
          sel : in  std_logic_vector(1 downto 0);
          dout: out std_logic);
```

**end entity mux;**

**architecture arc\_mux of mux is**

```
    signal decoded : std_logic_vector(3 downto 0);
```

**begin**

```
    decoder_2_to_4: process(sel) is
```

```
    begin
```

```
        decoded <= x"0"; -- Default assignment to bits not assigned on next line
```

```
        decoded(conv_integer(sel)) <= '1';
```

```
    end process decoder_2_to_4;
```

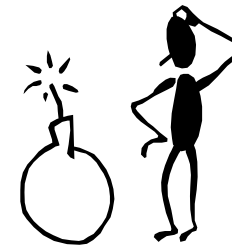
```
    dout <= din(0) when (decoded = x"1") else 'Z';
```

```
    dout <= din(1) when (decoded = x"2") else 'Z';
```

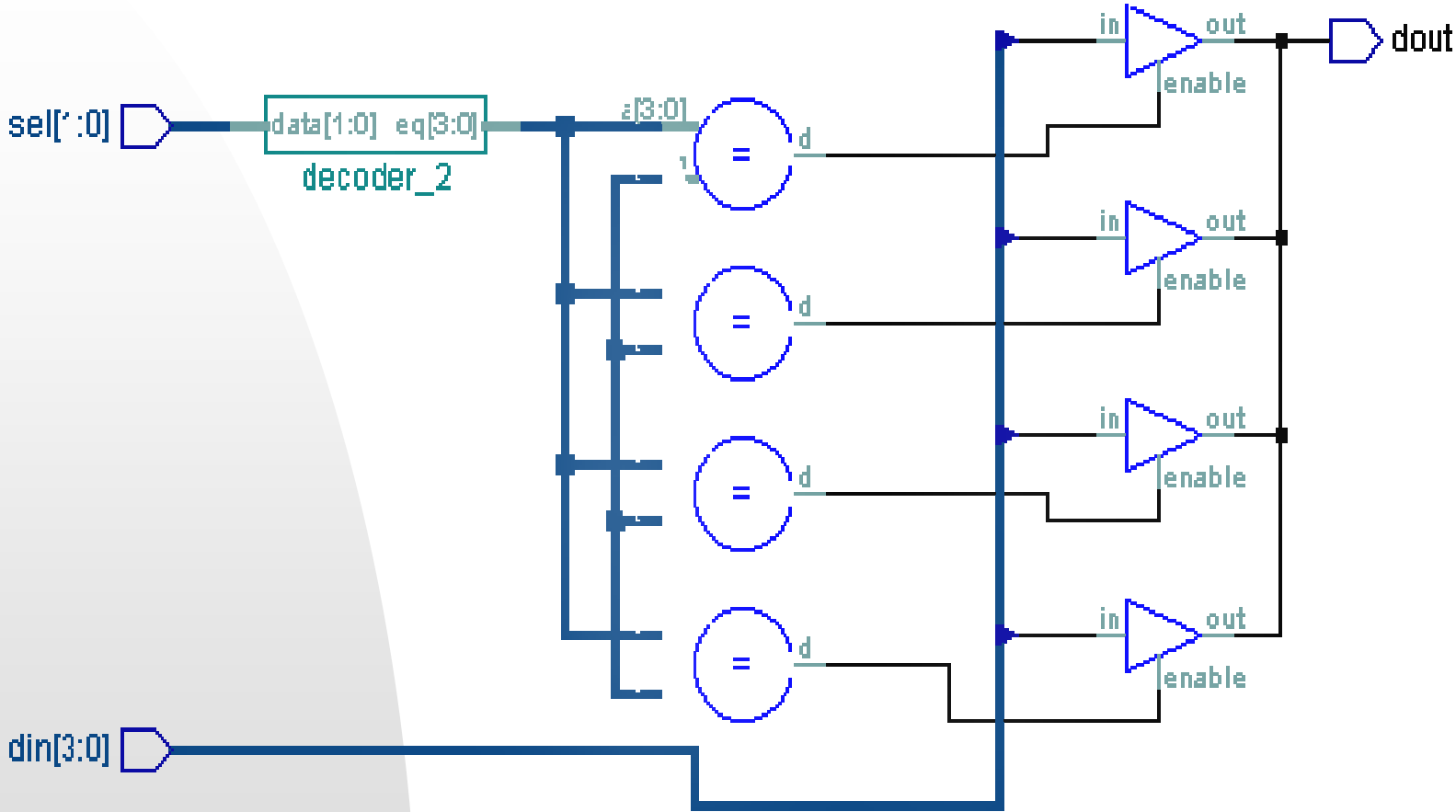
```
    dout <= din(2) when (decoded = x"4") else 'Z';
```

```
    dout <= din(3) when (decoded = x"8") else 'Z';
```

**end architecture arc\_mux;**



# Sequential Statements



# Sequential Statements

**entity** mux **is**

```
    port (din      : in  std_logic_vector(3 downto 0);
          sel      : in  std_logic_vector(1 downto 0);
          dout     : out std_logic);
```

**end entity** mux;

**architecture** mux\_arc **of** mux **is**

```
    signal decoded : std_logic_vector(3 downto 0);
```

**begin**

```
    decoder_2_to_4: process(sel) is
```

```
    begin
```

```
        decoded <= x"0";
```

```
        decoded(conv_integer(sel)) <= '1';
```

```
    end process decoder_2_to_4;
```

```
    dout <= din(0) when (decoded(0) = '1') else 'Z'; --optimized description
```

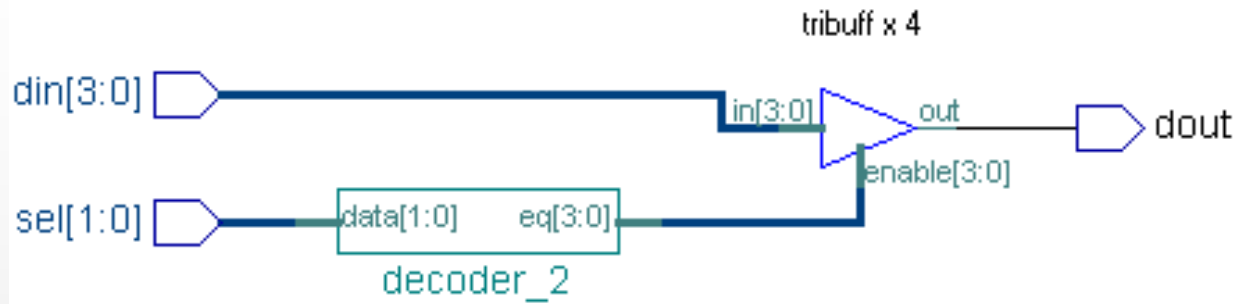
```
    dout <= din(1) when (decoded(1) = '1') else 'Z'; --without comparators
```

```
    dout <= din(2) when (decoded(2) = '1') else 'Z';
```

```
    dout <= din(3) when (decoded(3) = '1') else 'Z';
```

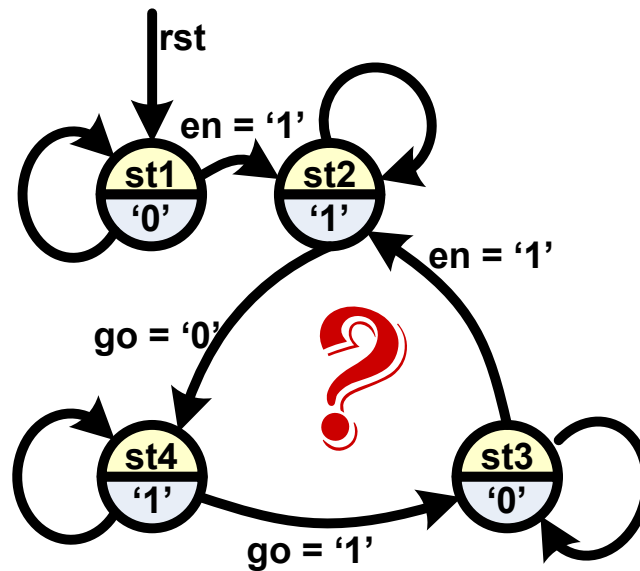
**end architecture** mux\_arc;

# Sequential Statements



# FSM

- A **finite state machine** (FSM) or **finite automaton** is a model of behavior composed of states and transitions. A state stores information about the past, i.e. it reflects the input changes from the system start to the present moment. A transition indicates a state change and is described by a condition that would need to be fulfilled to enable the transition.



# FSM

Defined:

$x = \{x_1, x_2, \dots, x_n\}$  - set of input signals

$A = \{a_1, a_2, \dots, a_m\}$  - set of states

$y = \{y_1, y_2, \dots, y_k\}$  - set of output signals

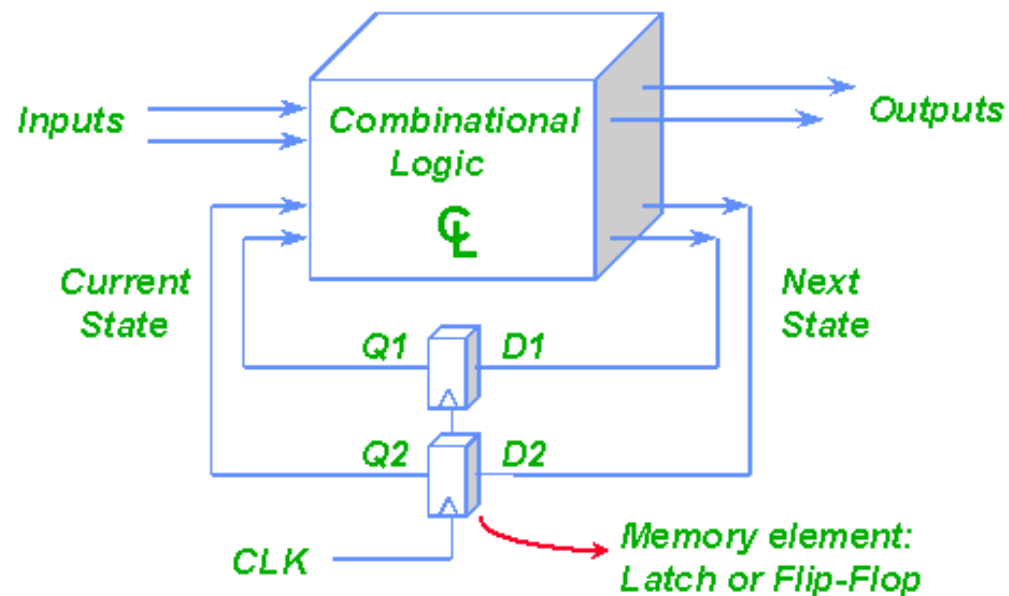
The transition function -  $\delta$

The output function -  $\lambda$

The initial state -  $a_1$

# FSM – General diagram representation

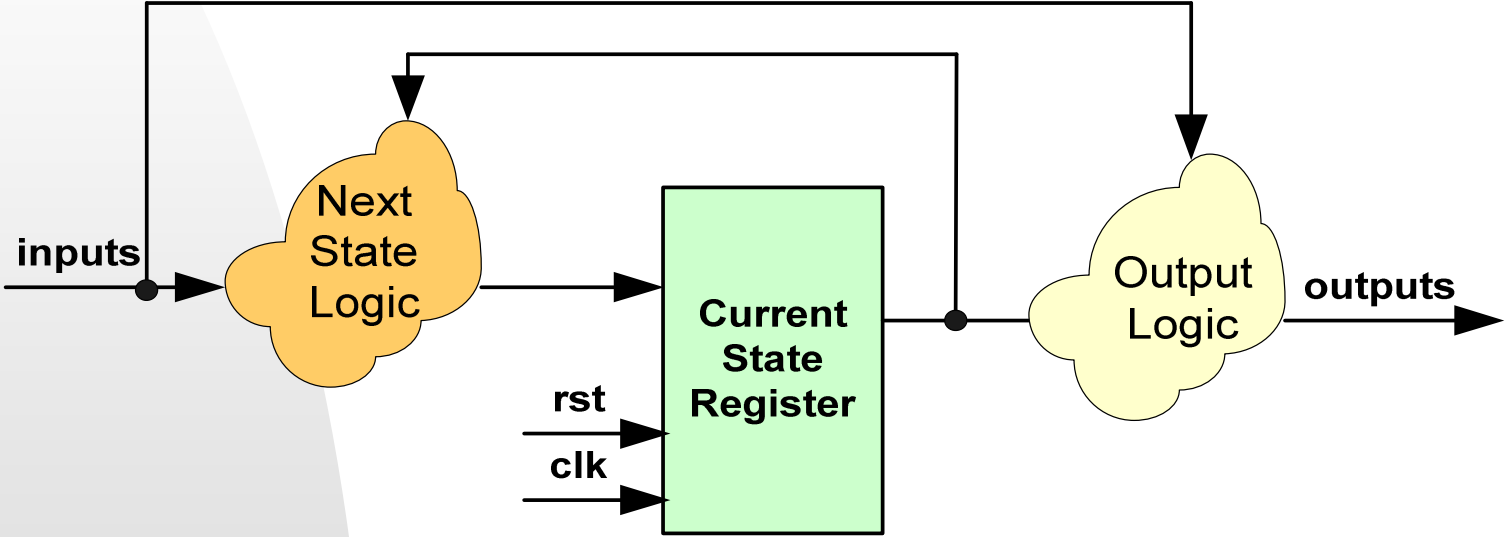
Many designs expressed at the register-transfer level consist of combinatorial data paths controlled by *finite-state-machines* -> *FSM*. There are basically three forms of state machines, Mealy, Moore and Medvedev (Full Moore) machines (MMM 😊).



Mealy  
FSM

# FSM forms

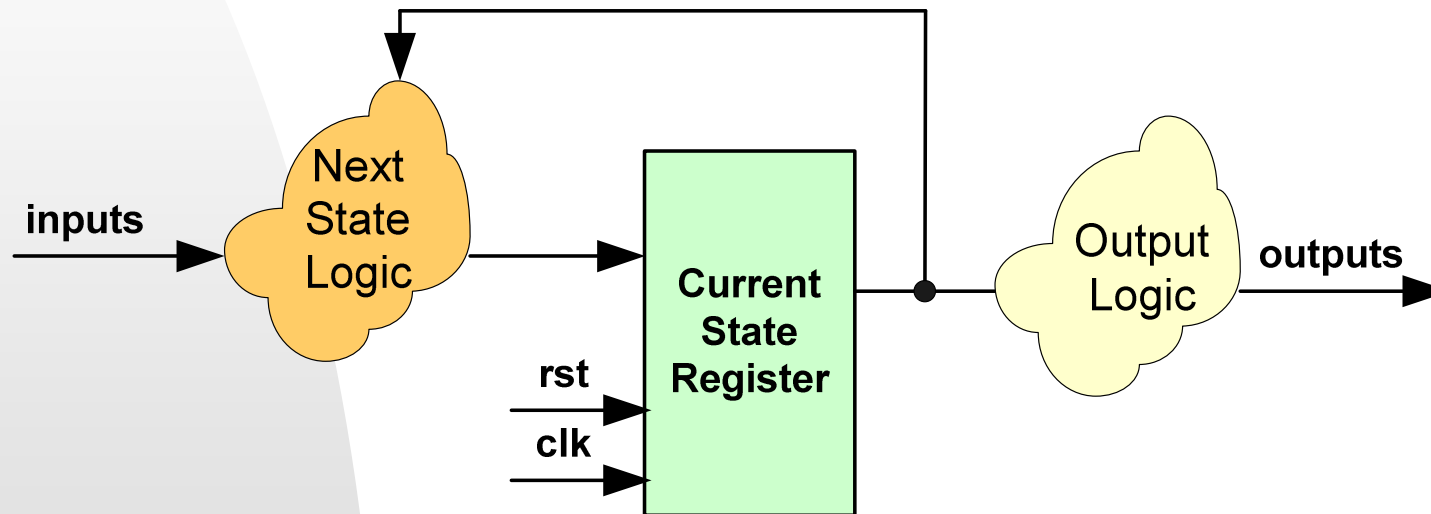
Mealy State Machine form - Output logic is determined based on :  
Present state and Input signals values.



Moore  
FSM

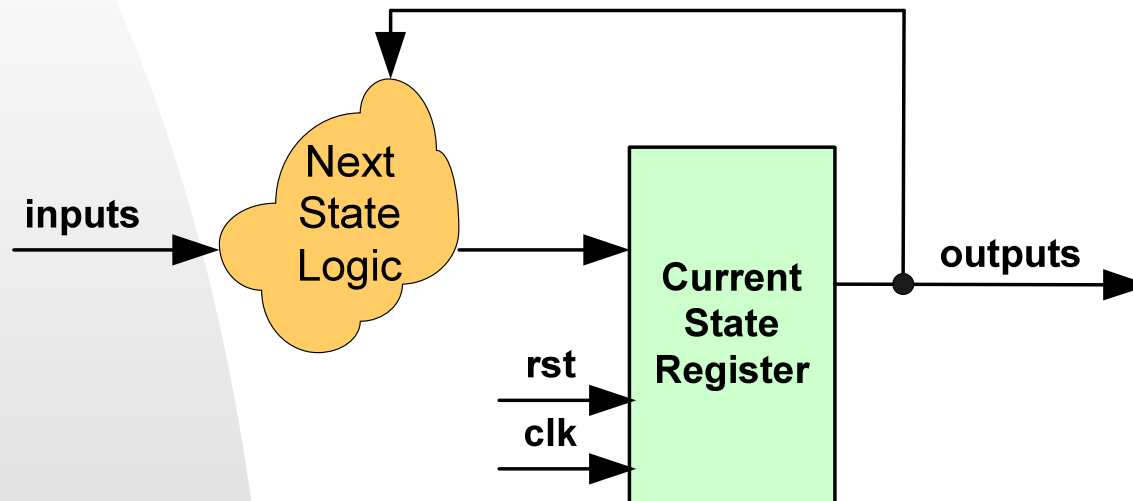
# FSM forms

Moore State Machine form - Output logic is determined based on :  
Present state only.



# FSM forms

Medvedev State Machine form (Full Moore) - Output logic is the value of the Present state.



# FSM outputs

Mealy FSM : The output vector (Y) is a function of state vector (S) and the input vector (X)

$Y = f(S, X)$  (*asynchronous to state, unregistered output*)

Full Mealy: The output vector (Y) resembles the next state vector (NS) or the slice of next state vector.

$Y = NS$



Moore FSM : The output vector (Y) is a function of state vector (S) only

$Y = f(S)$  (*synchronous to state, unregistered outputs*)

Medvedev FSM (Full Moore) : The output vector (Y) resembles the state vector (S) or a slice of the state vector.

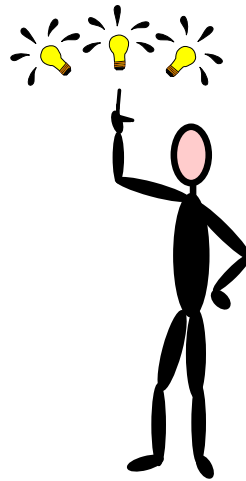
$Y = S$  (*synchronous to state, registered output*)

*Good choice for latency dependent systems*

# How many processes ?

The preferred style for beginners, is to separate the implementation into three processes:

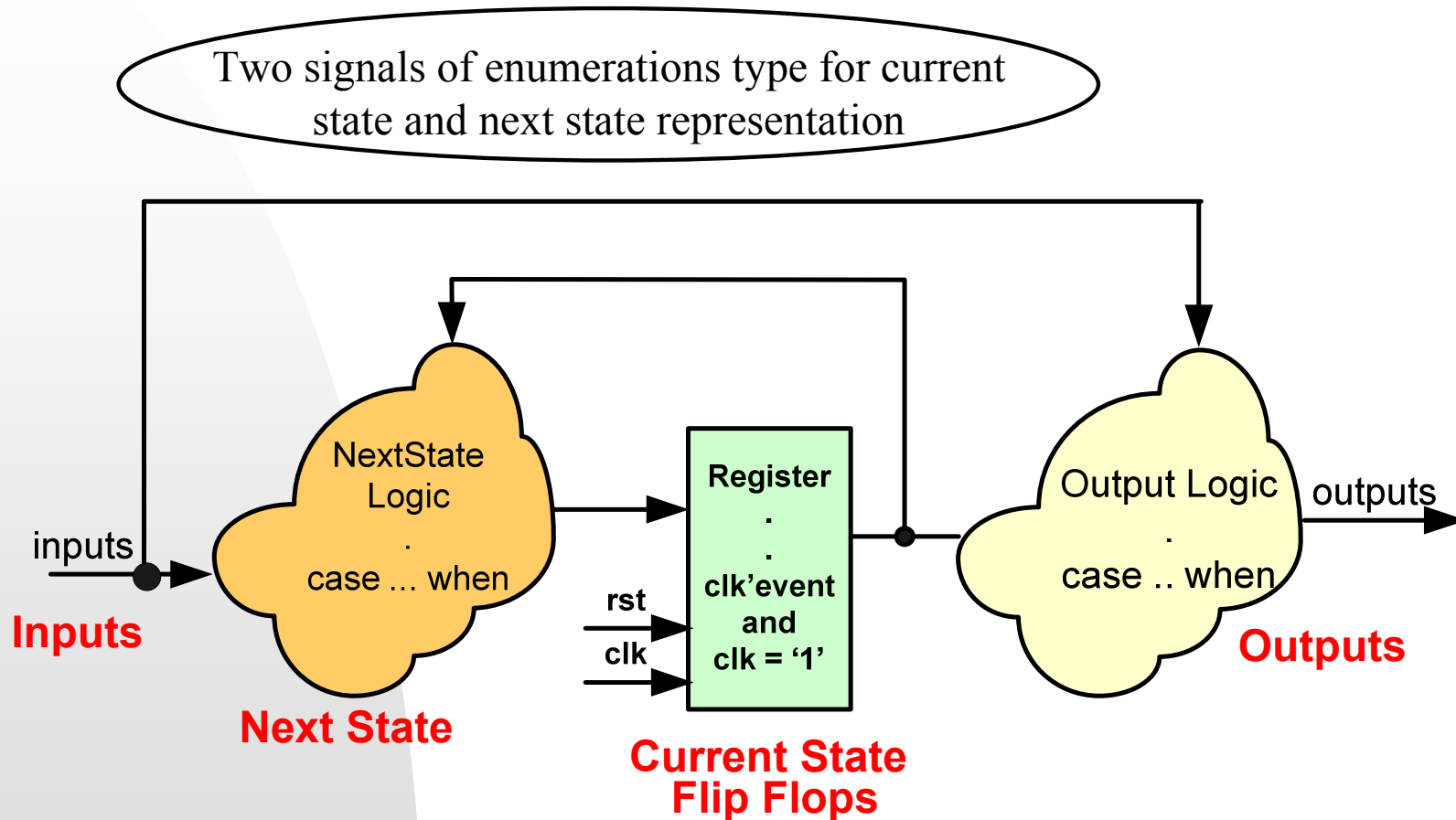
- First - the *register* that stores the state.
- Second - the logic combination for determining the *next state*.
- Third - the logic combination for determining the *outputs values*



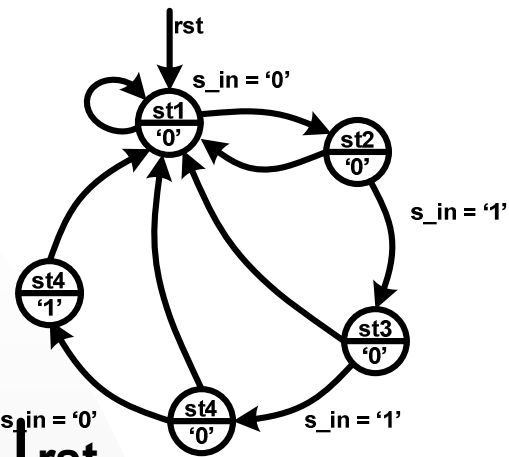
# How many processes ?

- One process – describes state register, state transaction and output logic.
  - ✓ Advantage : registered outputs
  - ✓ Disadvantage : verbose syntax, poorly debugging, 1 clock latency for outputs
- Two process – the first describes state register, the second combinatorial logic.
  - ✓ Advantage : easy to debugging, simply and readable code.
  - ✓ Disadvantage : non registered outputs, needs assignment to next state and outputs for all possible cases.
- Three processes – one for state register , one for next state logic, one for outputs
  - ✓ Advantage : easy to debugging, simply and readable code.
  - ✓ Disadvantage : non registered outputs, redundant code.
- Three processes – first for state register, second for next state logic, third for synchronous outputs.
  - ✓ Advantage : fully synchronous, readable code, easy for debugging.
  - ✓ Disadvantage : 1 clock cycle latency for output assertion

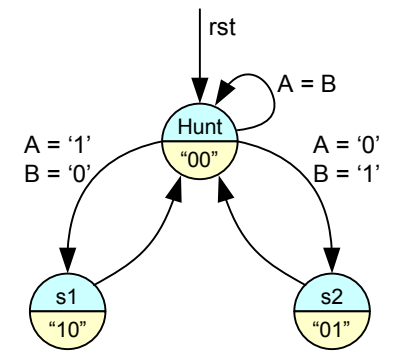
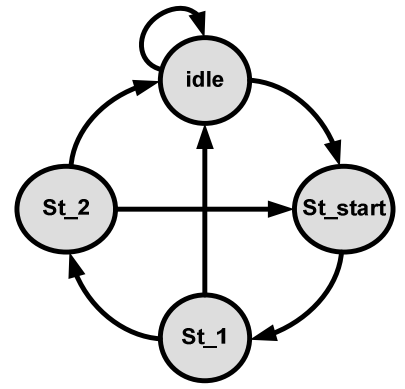
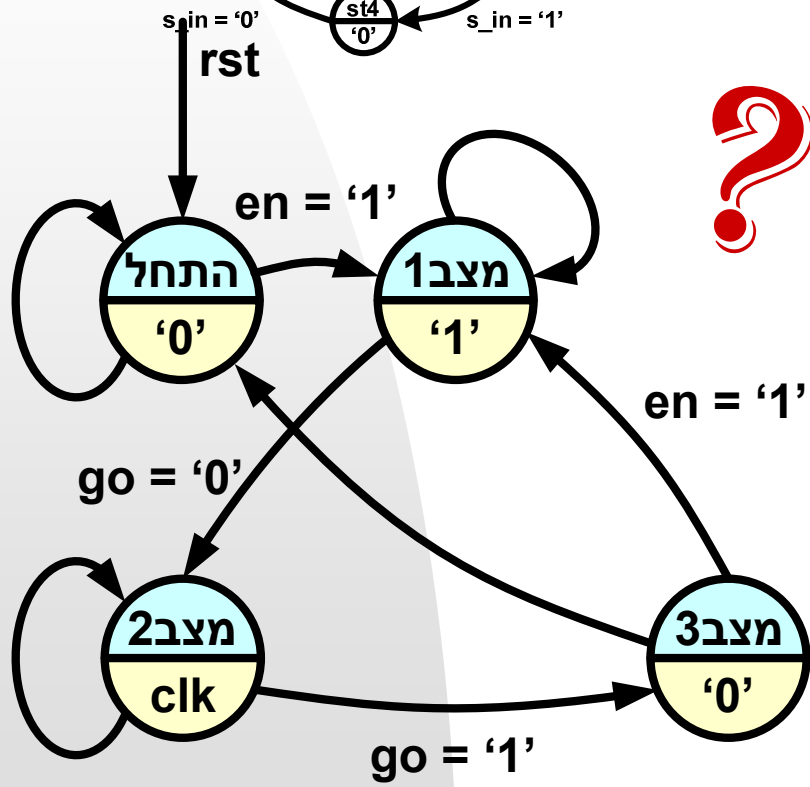
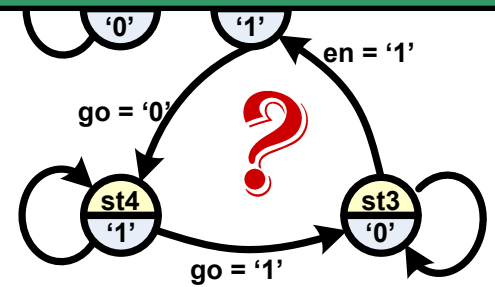
# Case Statement in FSM



# FSM Example

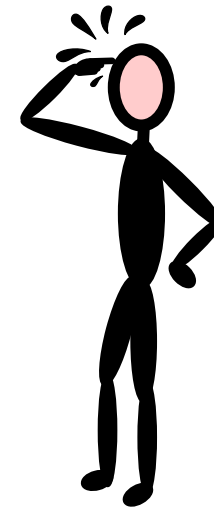
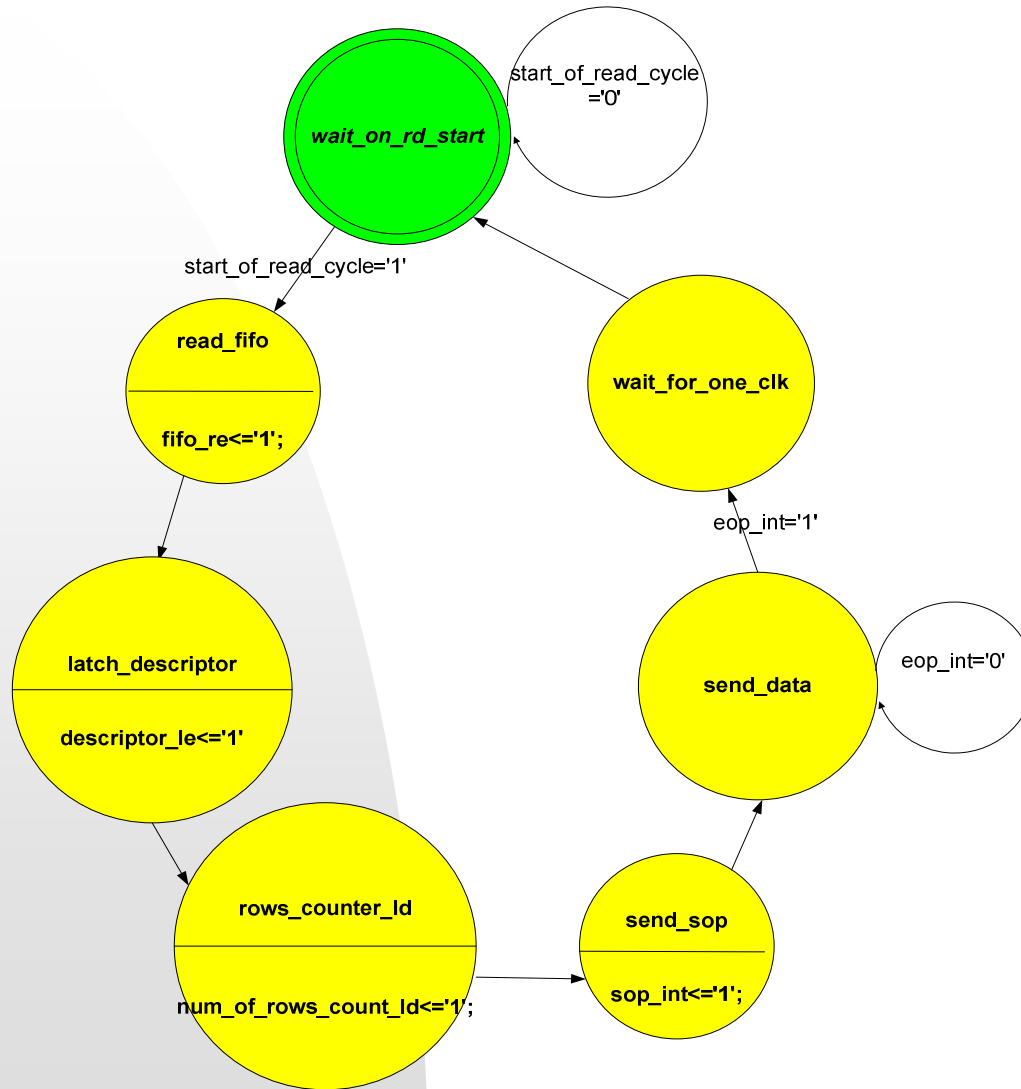


State Transition diagram may be good way for starting of FSM design



# Moore FSM Example

start\_of\_read\_cycle<='1' when (xgmii\_ready='1' and fifo\_empty='0') else '0';



# Three processes style

```
type rd_ctrl_type is (wait_on_rd_start, read_fifo, latch_descriptor,  
                      rows_counter_ld, send_sop, send_data, ait_for_one_clk);  
signal next_st, curr_st : rd_ctrl_type;
```

.....

```
-- Present state logic registers
```

```
rd_ctrl_fsm_curr_st_reg : process (sys_clk, sys_rst) is
```

```
begin
```

```
  if (sys_rst = RESET_INIT) then
```

```
    curr_st <= wait_on_rd_start;
```

```
  elsif rising_edge(sys_clk) then
```

```
    curr_st <= next_st;
```

```
  end if;
```

```
end process rd_ctrl_fsm_curr_st_reg;
```

```

ns_logic: process (curr_st, xgmii_rdy, fifo_empty, rows_cnt_eq_1, seq_5, cxe_int) is
begin
    next_st<=curr_st;
    case (curr_st) is
        when wait_on_rd_start =>
            if( (not fifo_empty) and xgmii_rdy and seq_5 and (not cxe_int)) then
                next_st <= read_fifo;
            end if;
        when read_fifo => next_st <= latch_descriptor;
        when latch_descriptor => next_st <= rows_counter_ld;
        when rows_counter_ld => next_st <= send_sop;
        when send_sop => next_st <= send_data;
        when send_data => if (rows_cnt_eq_1) then
            next_st <= wait_for_one_clk;
        end if;
        when wait_for_one_clk => next_st <= wait_on_rd_start;
        when others => next_st <= wait_on_rd_start;
    end case;
end process ns_logic;

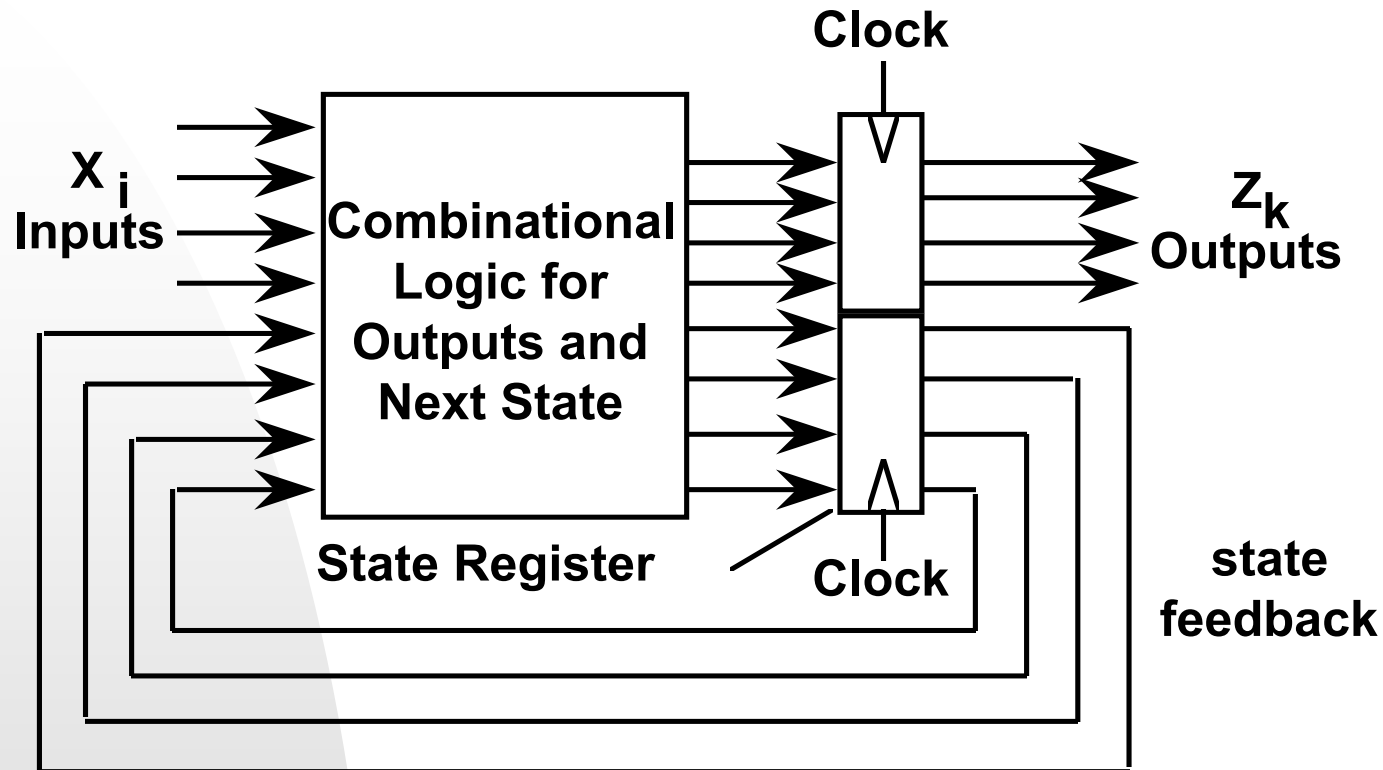
```

```

output_logic : process(curr_st) is
begin
    fifo_re      <= '0';
    descriptor_le <= '0';
    num_of_rows_cnt_ld <= '0';
    xgmii_sop_int <= '0';
    case (curr_st) is
        when read_fifo =>
            fifo_re <= '1';
        when latch_descriptor =>
            descriptor_le <= '1';
        when rows_counter_ld =>
            num_of_rows_cnt_ld <= '1';
        when send_sop =>
            xgmii_sop_int <= '1';
        when others => fifo_re <= '0';
    end case;
end process output_logic;

```

# Synchronous Mealy FSM, three processes style



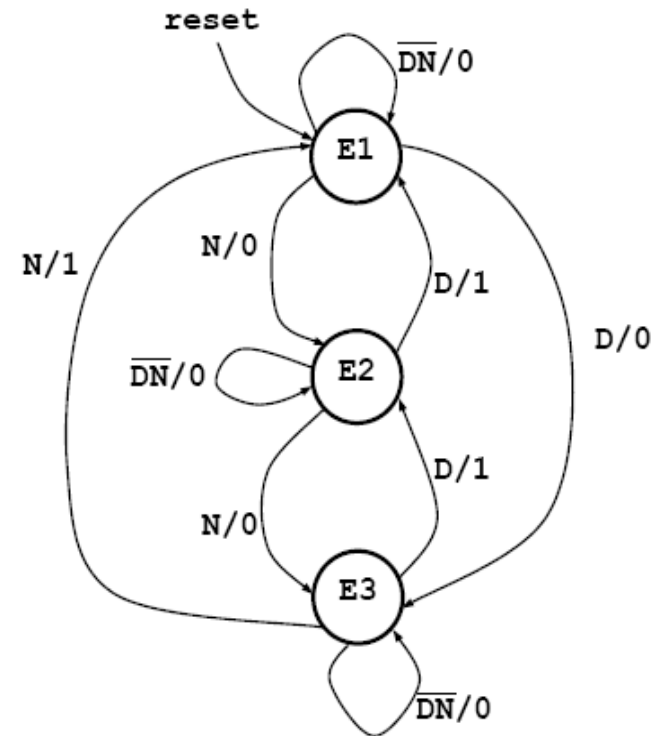
**Registered state AND outputs avoids glitchy outputs!**

# Two processes style

```
comb_logic: process (curr_st, xgmii_rdy, fifo_empty, rows_cnt_eq_1, seq_5, cxe_int) is  
begin  
  next_st          <= curr_st; fifo_re <= '0';          descriptor_le <= '0';  
  num_of_rows_cnt_ld <= '0';          xgmii_sop_int <= '0';  
  case (curr_st) is  
    when wait_on_rd_start =>  
      if( (not fifo_empty) and xgmii_rdy and (not cxe_int)) then  
        next_st <= read_fifo;  
      end if;  
    when read_fifo => next_st <= latch_descriptor; fifo_re <= '1';  
    when latch_descriptor => next_st <= rows_counter_ld; descriptor_le <= '1';  
    when rows_counter_ld => next_st <= send_sop; num_of_rows_cnt_ld <= '1';  
    when send_sop => next_st <= send_data; xgmii_sop_int <= '1';  
    when send_data =>  
      if (rows_cnt_eq_1) then  
        next_st <= wait_for_one_clk;  
      end if;  
    when wait_for_one_clk => next_st <= wait_on_rd_start;  
    when others => next_st <= wait_on_rd_start;  
  end case;  
end process comb_logic;
```

# Mealy FSM, Two processes

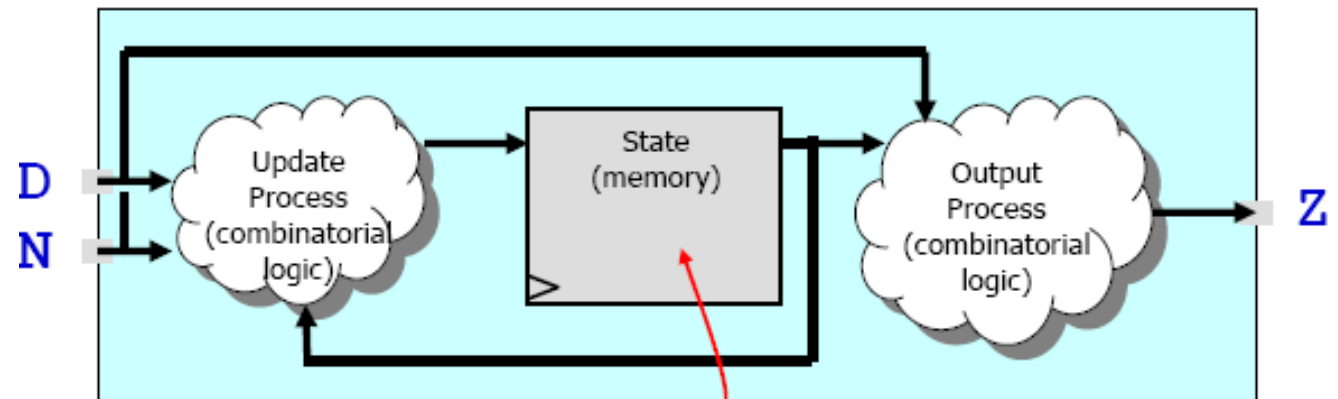
- Write a VHDL program to implement the FSM described by the state graph on the right
- The two inputs **D** and **N** are never active at once
- The FSM has a single output **Z**



# Mealy FSM (cont)

Two combinatorial blocks (processes)

One 2-bit state register (sequential process)

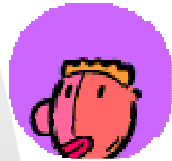


Three states  $\rightarrow$  2 bits  
( $2^2 = 4 \geq 3$ )



# Mealy FSM (cont)

Even if it is not mentioned,  
there is always a `reset` signal  
in a sequential design!



```
library ieee;
use ieee.std_logic_1164.all;

entity fsm is
  port (
    clk           : in  std_logic;
    reset         : in  std_logic;
    D             : in  std_logic;
    N             : in  std_logic;
    Z             : out std_logic);
end fsm;

architecture synth of fsm is
  type t_State is (E1, E2, E3);
  signal s_State      : t_State;
  signal s_NextState  : t_State;
begin
```

# Mealy FSM (cont)

```
process (s_State, D, N)
begin
  Z      <= '0';
  s_NextState <= E1;
  case s_State is
    when E1 =>
      if (D = '1') then
        s_NextState <= E3;
      elsif (N = '1') then
        s_NextState <= E2;
      end if;
    when E2 =>
      if (D = '1') then
        Z      <= '1';
      elsif (N = '1') then
        s_NextState <= E3;
      else s_NextState <= E2;
      end if;
    when E3 =>
      if (D = '1') then
        s_NextState <= E2;
        Z      <= '1';
      elsif (N = '1') then
        Z      <= '1';
      else s_NextState <= E3;
      end if;
  end case;
end process;
```

```
process (clk, reset)
begin
  if (reset = '1') then
    s_State <= E1;
  elsif (clk'event and clk = '1') then
    s_State <= s_NextState;
  end if;
end process;
end synth;
```

Only one process, because the two combinatorial blocks test exactly the same input variables in the same way—but we could as well have written two...

We make sure that in a sequential process we always assign something—no matter what...

# One Process style

```
type state_type is (s1,s2,s3,s4);
signal state: state_type;
.....
process (clk, reset) is
begin
  if (reset = '1') then
    state <= s1; outp <= '1';
  elsif rising_edge(clk) then
    case state is
      when s1 => if (x1='1') then
                    state <= s2; outp <= '1';
                else
                    state <= s3; outp <= '0';
                end if;
      when s2 => state <= s4; outp <= '0';
      when s3 => state <= s4; outp <= '0';
      when s4 => state <= s1; outp <= '1';
    end case;
  end if;
end process;
```