

**State university of New York at New Paltz  
Electrical and Computer Engineering Department**

---

# Introduction to VHDL

By

Dr. Yaser Khalifa

Electrical and Computer Engineering Department

State University of New York at New Paltz

l t z

# *What is VHDL?*

- VHDL stands for VHSIC Hardware Description Language.
- VHSIC is an abbreviation for Very High Speed Integrated Circuit, a project sponsored by the US Government and Air Force begun in 1980 to advance techniques for designing VLSI silicon chips.

VHDL is an IEEE standard.

# *Using VHDL for design synthesis*

The design process is a 6 step cycle:

- Define the design requirement
- Describe the design in VHDL code
- Simulate the source code
- Synthesize, optimize, and fit the design
- Simulate the design
- Implement the design

## *Design requirements*

Required setup, clock requirements, maximum operating frequency, critical paths.

## *Describe the design in VHDL code*

Prior to this a design methodology should be used to describe the system. The most common of these are:

- *Top-down* : In top-down, the main functional blocks are first defined, where each block has its defined inputs and outputs and is performing a specific function. Then a description of lower levels follows.

- *Bottom-up* : In this case first the individual blocks are defined and designed and then they are brought together to form the overall design.
- *Flat* : In this case the design is normally simple and does not need a hierarchical approach.

## *Simulate the source code*

- This is an important feature of Hardware Description Languages as the simulation is done before the synthesis and design stage.

## *Synthesize, optimize and fit the design*

- *Synthesis tools in VHDL*, allow designers to design logic circuits by creating design descriptions without necessarily having to perform Boolean algebra or create technology-specific, optimized functions.

*Optimization process*, is the process of minimizing the circuitry by means of reducing the sum of product terms. However, the process is multi-objective and dependent on the expression being realized and the device used in implementation

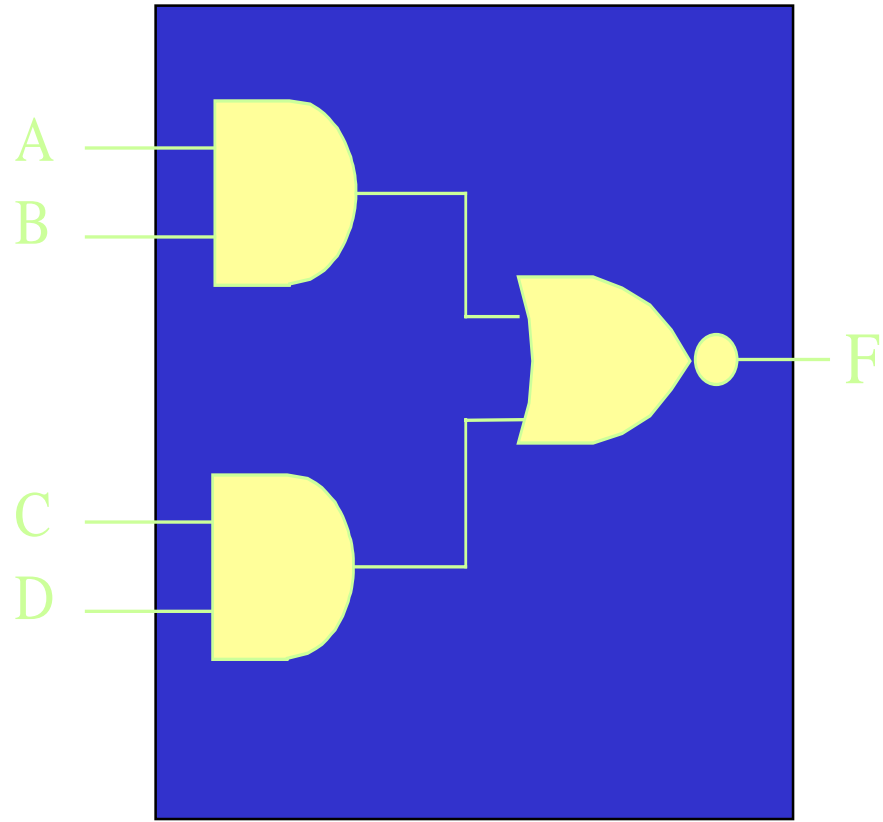
*Fitting*, is the process of taking the logic produced by the synthesis and optimization processes and placing it into a logic device.

*Place and route*, is fitting the logic into the logic device and placing the logic elements in optimal locations and routing the signals from one unit to another in an optimal way.

# Design Entity

A design entity is design element in VHDL and consists of:

- an entity declaration and
- an architecture body.



# Entity

An entity body describes the input and output pins (ports) of the design entity.

**entity** AOI **is**

**port** ( A , B , C , D : **in** STD\_LOGIC ;

    F : **out** STD\_LOGIC ) ;

**end** AOI ;

**architecture V1 of AOI is**

**begin**

**F <= not ( ( A and B ) or ( c and D ) ) ;**

**end V1 ;**

# Entity Declaration

Each port in an entity should include :

- name of the signal (identifier)
- direction (mode)
- data type

*Modes* : There are four default modes in VHDL

in : input into the entity ( unidirectional)

out : output from the entity( unidirectional)

inout : input and output to and from the entity  
( bidirectional signals )

buffer : behaves in a similar way as inout,  
but the source of the buffered signal is always  
determined by the driving value of the port.

# *Data Types*

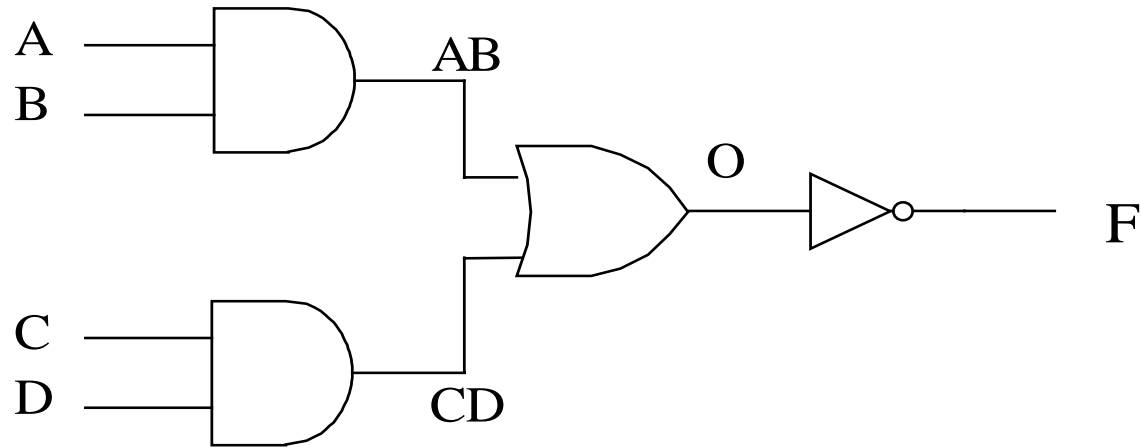
There are scalar and array data types in VHDL.

array types	example
string	“abc”
bit_vector	“1001”
std_logic_vector	“101Z”

Scalar type	example
character	'a'
bit	'1' '0'
std_logic	'0' forcing 0 '1' forcing 1 'x' forcing unknown 'z' high impedance '-' don't care 'L' weak 0 'H' weak 1 'U' uninitialized
boolean	true false
real	2.35 -1.0E+38
integer	832 -1
time	fs, ps, ns, us, ms, sec, min, hr

# Signals

An internal connection is described in VHDL as a signal defined inside the architecture.



**architecture V2 of AOI is**

**signals AB, CD, O : STD\_LOGIC ;**

**begin**

**AB <= A and B after 2 NS;**

**CD <= C and D after 2 NS;**

**O <= AB or CD after 2 NS;**

**F <= not O after 1 NS;**

**end V2 ;**

# Concurrent Statements

- VHDL statements are grouped into sequential and concurrent statements. Concurrent statements are used in data flow and structural descriptions. Sequential statements are used in behavioral descriptions

Concurrent statements are mainly:

Concurrent signal assignment statements with BOOLEAN equations

- Selective signal-assignment (with-select-when)  
Conditional statements (if-then-else)

# Concurrent Statements

A concurrent signal assignment is triggered by an event on a signal. An event is a change in value.

An event on the input port A would trigger the assignment to AB

```
port (A, B, C, D: in STD_LOGIC;
```

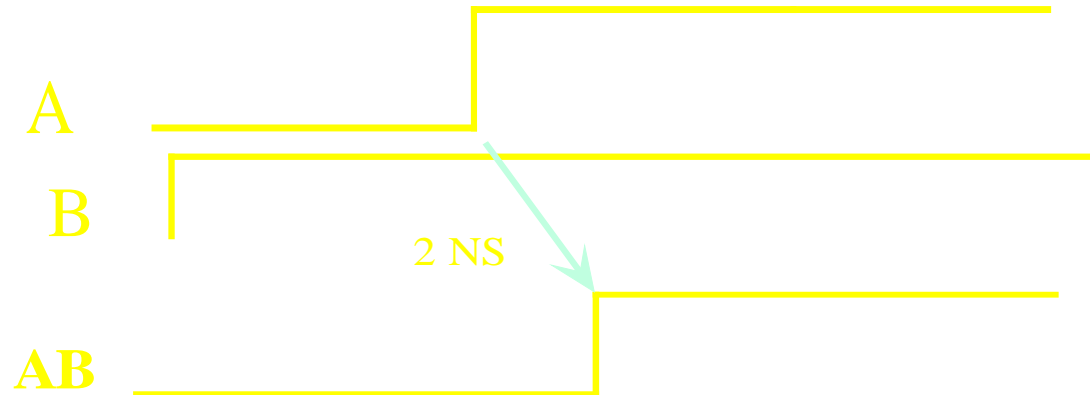
```
...
```

```
AB <= A and B after 2 NS;
```

```
port (A, B, C, D: in STD_LOGIC;
```

```
...
```

```
AB <= A and B after 2 NS;
```

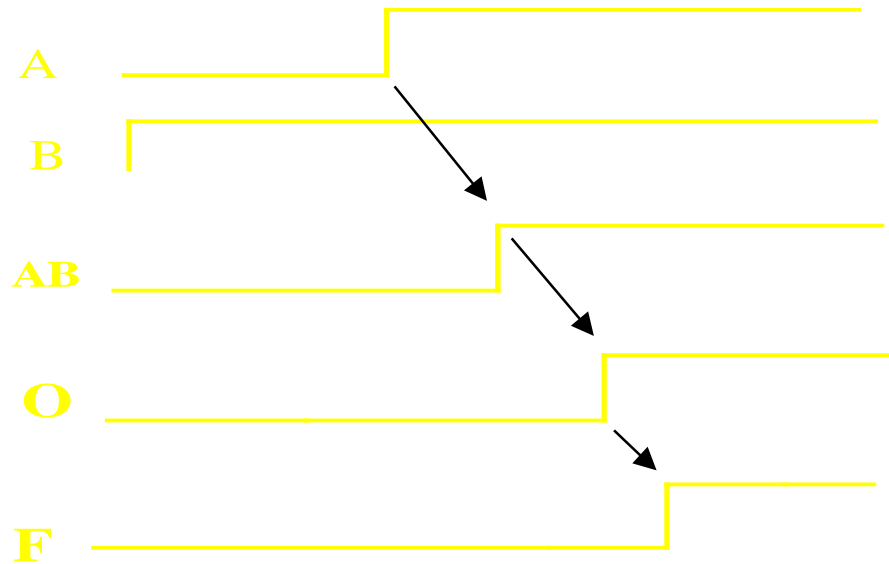


$AB \leq A \text{ and } B \text{ after } 2 \text{ NS};$

$CD \leq C \text{ and } D \text{ after } 2 \text{ NS};$

$O \leq AB \text{ or } CD \text{ after } 2 \text{ NS};$

$F \leq \text{not } O \text{ after } 1 \text{ NS};$



# Variables

Previously we have looked at signals as electrical connections or “pieces of wires”.

Variables can be pieces of wire too, but they can also be more abstract.

Variables can represent wires, registers, or be used to store intermediate values in abstract calculations.

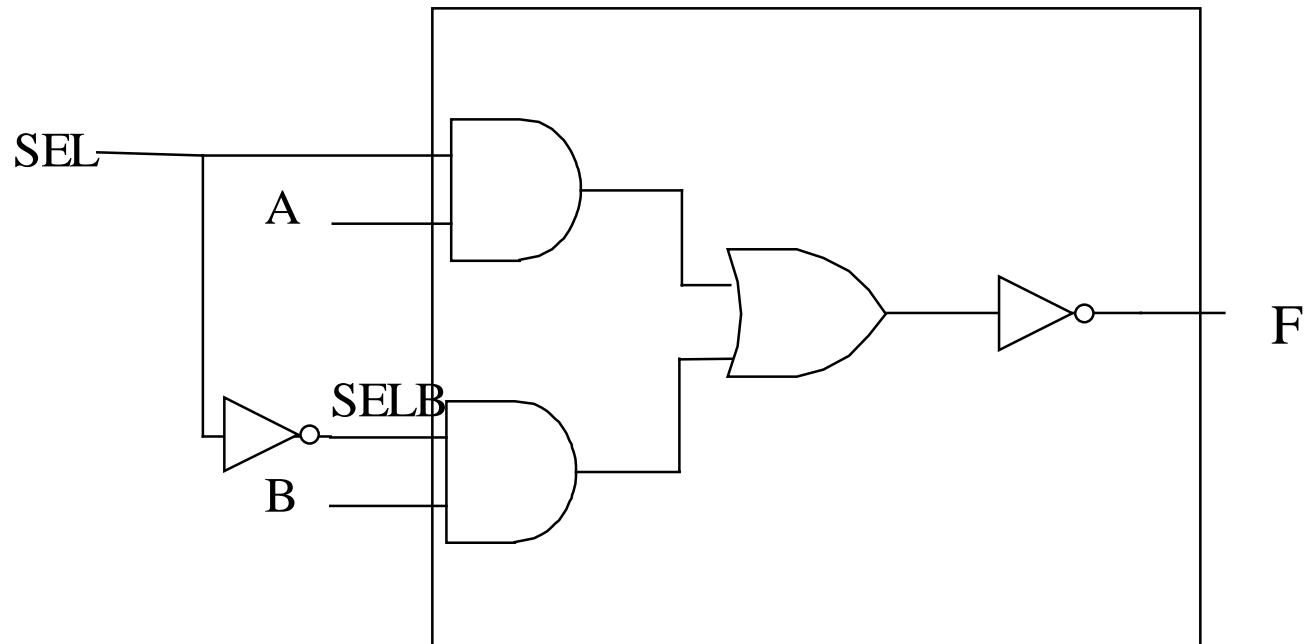
```
variable V: STD_LOGIC;
```

Variables must be defined inside a process, before begin (unlike signals which are defined in an architecture).

# Components

A component is analogous to a chip socket. It creates an extra interface which allows more flexibility when building hierarchical system out of its component parts.

```
library IEEE;  
use IEEE.STD_LOGIC_1164.all;  
  
entity MUX2 is  
    port (SEL, A, B: in STD_LOGIC;  
          F: out STD_LOGIC);  
end MUX2;
```



architecture STRUCTURE of MUX2 is

component INV

port (A: in STD\_LOGIC;

F: out STD\_LOGIC);

end component;

component AOI

port (A, B, C, D: in STD\_LOGIC;

F: out STD\_LOGIC);

end component;

signal SELB: STD\_LOGIC;

begin

G1: INV port map(SEL, SELB);

G2: AOI port map(SEL, A, SELB, B, F);

end STRUCTURE;

# Variables

Previously we have looked at signals as electrical connections or “pieces of wires”.

Variables can be pieces of wire too, but they can also be more abstract.

Variables can represent wires, registers, or be used to store intermediate values in abstract calculations.

```
variable V: STD_LOGIC;
```

Variables must be defined inside a process, before begin (unlike signals which are defined in an architecture).

**A variable has a name and a type, just like a signal.**

```
-- Assume A = '0', B = '0', C = '1'  
  
process (A, B, C)  
  variable V: Std_logic;           [ V = 'U' ]  
begin  
  V := A nand B;                   [ V = '1' ]  
  V := V nor C;                   [ V = '0' ]  
  F <= not V;  
end process;
```

**Variables behave quite differently to signals; variable assignments never have a delay. Signal assignments always have a delay (even if it is a delta delay ).**

architecture ...

```
signal F: Std_logic;
```

```
begin
```

```
  process (A, B, C)
```

```
    variable V: Std_logic;
```

```
  begin
```

```
    V := A nand B;
```

```
    V := V nor C;
```

```
    F <= not V;
```

```
  end process;
```

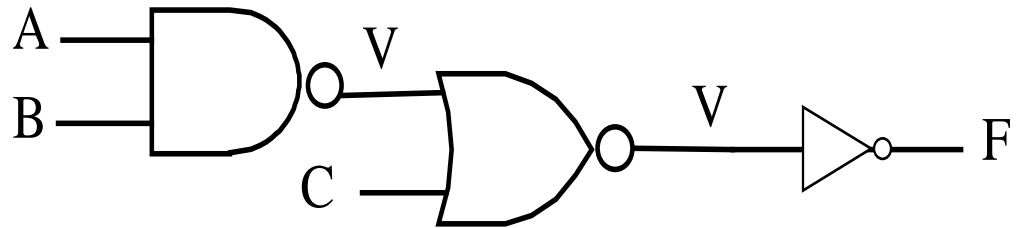
```
  process (F)
```

```
  begin
```

```
    ...
```

```
  end process;
```

When this process is synthesized, each variable assignment creates a new logic level. A new wire is synthesized to hold the value of the variable each time it is assigned.



A variable can be used *ONLY* inside the process. So, if we want a value passed between processes, we *MUST* use a signal.

## Structural Description

```
library IEEE;  
use IEEE.STD_LOGIC_1164.all;  
  
entity COMP4 is  
  port ( A, B : in STD_LOGIC_VECTOR ( 3 downto 0);  
         F : out STD_LOGIC) ;  
end COMP4;  
  
use WORK.GATESPKG.all ;  
architecture STRUCT of COMP4 is  
  signal x : STD_LOGIC_VEXTOR ( 0 to 3 );  
begin  
  u0: xnor2 port map (a(0) , b(0) , x(0)) ;  
  u1: xnor2 port map (a(1) , b(1) , x(1)) ;  
  u2: xnor2 port map (a(2) , b(2) , x(2)) ;  
  u3: xnor2 port map (a(3) , b(3) , x(3)) ;  
  u4: and4 port map (x(0) , x(1) , x(2) , x(3) , F) ;  
  
end STRUCT;
```

## Process Statement Execution

A process is an operation that involves a number of parameters identified in a sensitivity list:

```
proc_x : process (a , b , c )  
    begin  
        x <= a and b and c;  
    end process;
```

A process is either being executed or suspended. It is executed when one of the signals in its sensitivity list had an **event, a change of value.**

The process continues to execute until the last statement is reached or it encounters a wait statement. It then suspends itself.

A wait statement: is a sequential statement which causes a process to be suspended. It can wait for a period of time (wait for), an event on a signal in the sensitivity list (wait on), a boolean condition to become true (wait until), or forever (wait ;).

```
proc_x : process
    begin
        x <= a and b and c;
        wait on a , b , c ;
    end process;
```

The process is used to describe the behaviour of a part of a system without getting into the details of the implementation, so can be used to describe hardware at high level of abstraction.

Concurrent statement : It is a statement which execute in parallel with other concurrent statements in the design hierarchy. They are written between begin and end in an architecture. The most common concurrent statements are

**a) the process**

**b) the component instantiation**

**c) concurrent signal assignment**

# Sequential Statements

## If Statements

An if statement is a sequential statement which conditionally executes other sequential statements, based on the value of a Boolean condition.

An if statement can contain any number of other statements, including other nested if statements.

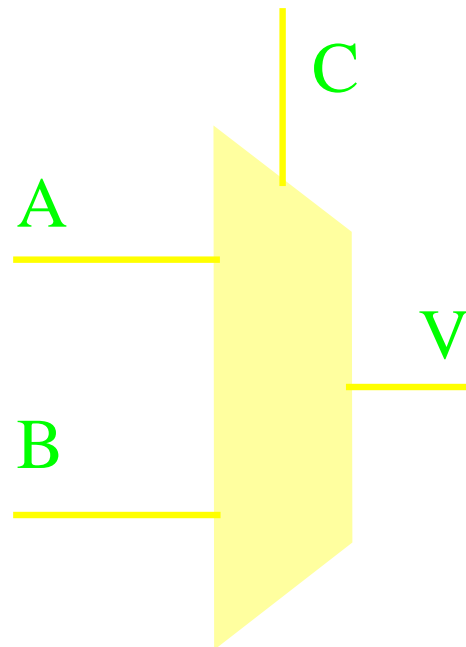
```
if C1 = '0' then  
  V := A;  
end if;
```

```
if C2 = '0' then  
  V := B;  
  W := C;  
  if C3 = '0' then  
    X := D;  
    Y := E;  
  end if;  
else  
  V := C;  
  W := D;  
end if;
```

**An if statement is synthesised by generating a multiplexer for every signal or variable assigned within the statement.**

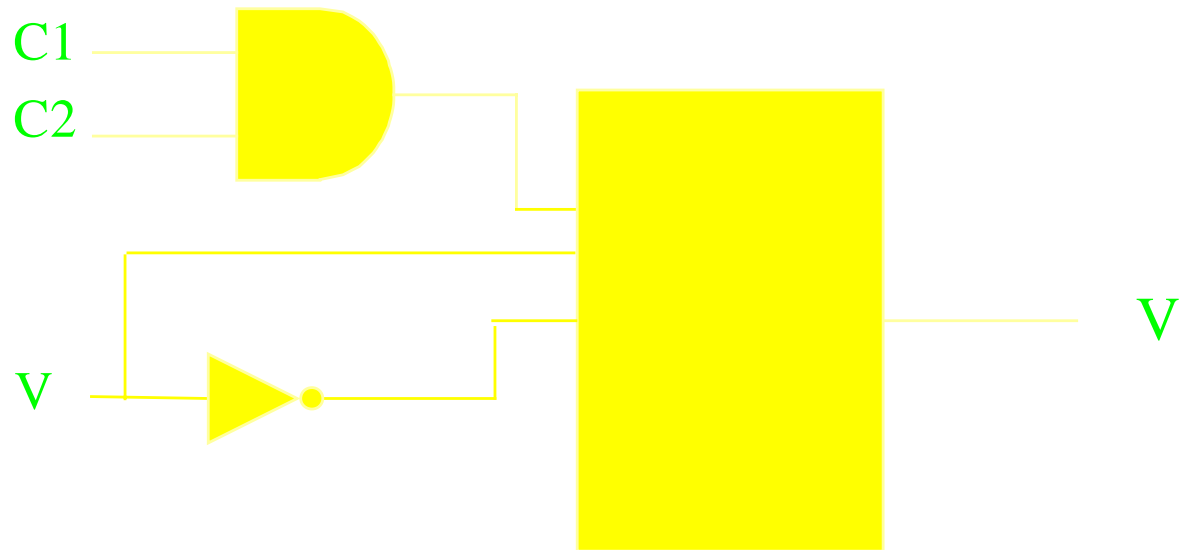
**The condition given at the top of the if statement forms the select input to the multiplexer.**

```
if C1 = '0' then  
    V := A;  
else  
    V := B;  
end if
```



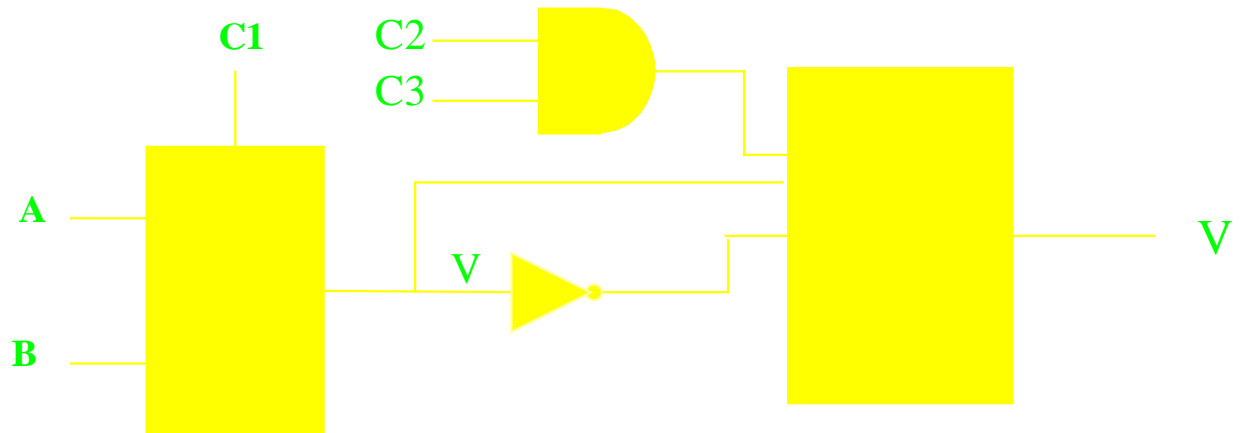
**In the example below the if statement does not have an else part, but the assignment  $V$  within the if still synthesizes to a multiplexer.**

```
if C2 = '1' and C3 = '1' then  
  V := not V;  
end if;
```



**If the condition is false, the previous value of  $V$  is fed through the multiplexer unaltered.**

```
process (C1, C2, C3, A, B)
  variable V: Std_logic;
begin
  if C1 = '0' then
    V := A;
  else
    V := B;
  end if;
  if C2 = '1' and C3 = '1'
  then
    V := not V;
  end if;
  F <= V;
end process;
```



## Nested if Statements

**Each if must be balanced with an end if.**

```
process (C0, C1, C2, A, B, C, D)
```

```
begin
```

```
  if C0 = '1' then
```

```
    F <= A;
```

```
  elseif C1 = '1' then
```

```
    F <= B;
```

```
  elseif C2 = '1' then
```

```
    F <= C;
```

```
  else
```

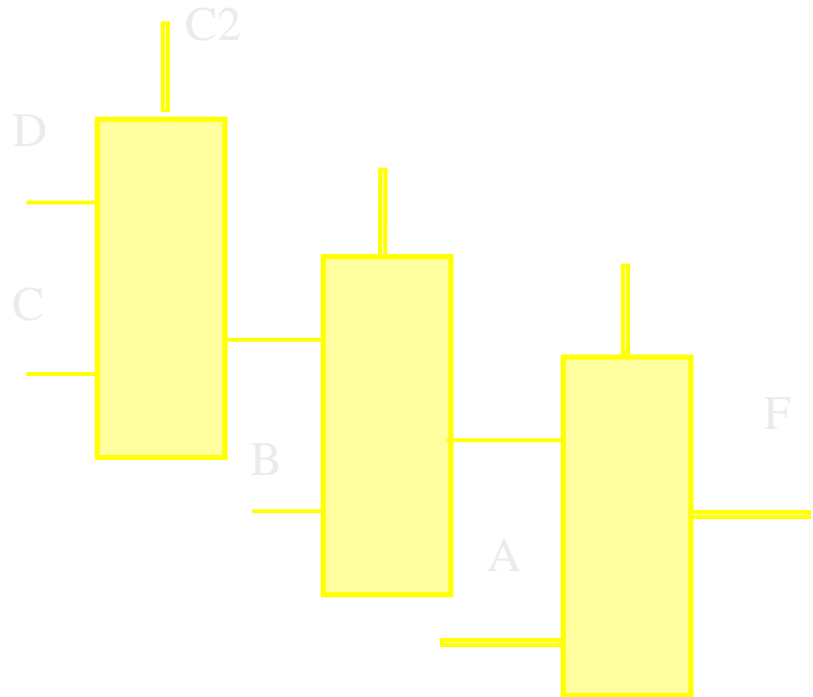
```
    F <= D;
```

```
  end if;
```

```
  end if;
```

```
end if;
```

```
end process
```



**The elseif allows multiple tests to be cascaded together, such that only the branch following the first true condition is executed.**

```
process (C0, C1, C2, A, B, C, D)
begin
  if C0 = '1' then
    F <= A;
  elsif C1 = '1' then
    F <= B;
  elsif C2 = '1' then
    F <= C;
  else
    F <= D;
  end if;
end process;
```

**Here there is no end if after each elseif.**

**Example:**

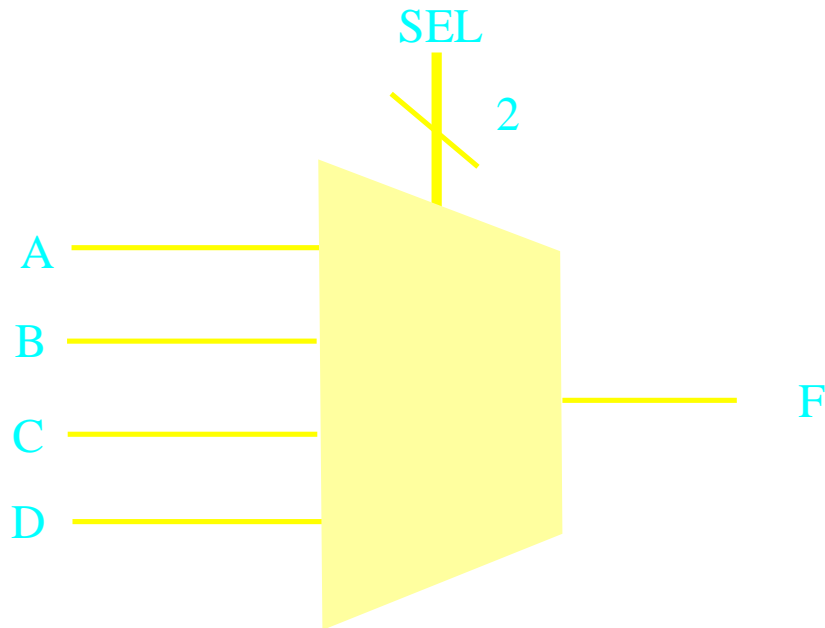
```
process  
begin  
    if Clock = '0' then  
        F <= '0';  
    else  
        F <= A;  
    end if;  
end process;
```

**What kind of hardware would be synthesized?**

# Case Statements

The expression at the top of the case is evaluated and compared with the expressions following the whens. The statements within the matching when branch are executed, then control jumps to the statement following end case.

```
case SEL is
when "00" =>
  F <= A;
when "01" =>
  F <= B;
when "10" =>
  F <= C;
when "11" =>
  F <= D;
when others =>
  F <= 'X';
end case;
```



The others branch will catch any cases not already mentioned explicitly by the when.

**It is possible to cover several cases in the same when branch by separating the values with vertical bars. In this example the branch is executed if ADDRESS is:  
16, 20, 24 or 28.**

```
case ADDRESS is
```

```
when 16 | 20 | 24 | 28 =>
```

```
  A <= '1';
```

```
  B <= '1';
```

```
when others =>
```

```
end case;
```

**Each branch can also include any number of sequential statements including other nested case statements as well!!**

```
case ADDRESS is
```

```
when 16 | 20 | 24 | 28 =>
```

```
  A <= '1';
```

```
  B <= '1';
```

```
when others =>
```

```
end case;
```

# The null statement means *do nothing*

```
A <= '0';
```

```
B <= '0';
```

```
case ADDRESS is
```

```
when 0 to 7 =>
```

```
  A <= '1';
```

```
when 8 to 15 =>
```

```
  B <= '1';
```

```
when 16 | 20 | 24 | 28 =>
```

```
  A <= '1';
```

```
  B <= '1';
```

```
when others =>
```

```
  null;
```

```
end case;
```

**It is possible also to cover a whole range using the case statement.**

**case ADDRESS is**

**when 0 to 7 =>**

**A <= '1';**

**when 8 to 15 =>**

**B <= '1';**

**when 16 | 20 | 24 | 28 =>**

**A <= '1';**

**B <= '1';**

**when others =>**

**end case;**

**If ADDRESS is:**

1) =9

2) =19

**What will be the values of A and B?**

```
A <= '0';
```

```
B <= '0';
```

```
case ADDRESS is
```

```
when 0 to 7 =>
```

```
  A <= '1';
```

```
when 8 to 15 =>
```

```
  B <= '1';
```

```
when 16 | 20 | 24 | 28 =>
```

```
  A <= '1';
```

```
  B <= '1';
```

```
when others =>
```

```
  null;
```

```
end case;
```

## **Case versus if:**

- 1) if is used when priority is desired.**
- 2) in cases of more than three branches, case is preferred.**

# *FOR Loops*

**The FOR loop is a sequential statement which is used to execute a set of sequential statements repeatedly.**

```
for I in 0 to 3 loop  
  F(I) <= A(I) and B(3-I);  
  V := V xor A(I);  
end loop;
```

**The range can be ascending or descending, but the loop parameter cannot change in increments greater than 1.**

**for I in 0 to 3 loop**

**F(I) <= A(I) and B(3-I);**

**V := V xor A(I);**

**end loop;**

**for I in 3 downto 0 loop**

**F(I) <= A(I) and B(3-I);**

**V := V xor A(I);**

**end loop;**

**The loop parameter is technically a constant, which means that its value cannot be changed using an assignment.**

**You cannot jump out of a loop by forcing the value of the loop parameter**

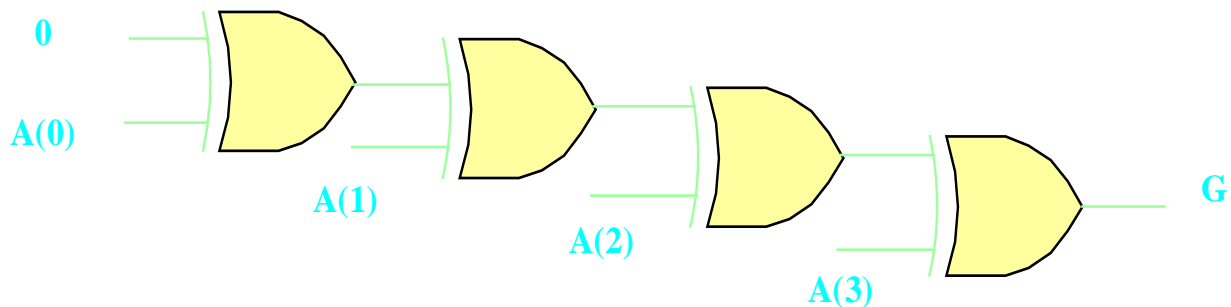
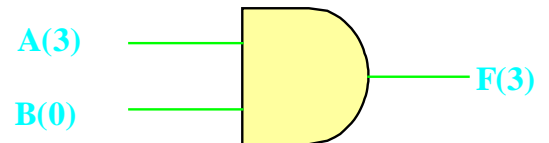
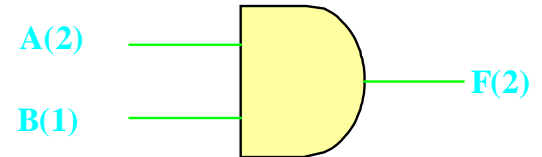
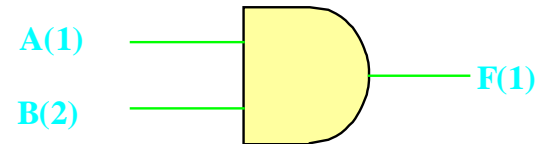
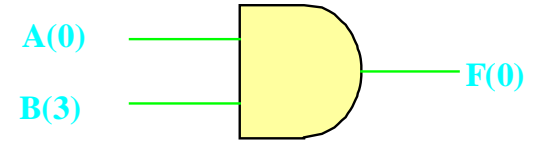
```
for I in 0 to 3 loop  
  F(I) <= A(I) and B(3-I);  
  V := V xor A(I);  
  if V = 'X' then  
    I := 4;  
  end if;  
end loop;
```

**FOR Loop are synthesized by making multiple copies of the logic synthesized inside the loop, one copy for each possible value of the loop parameter.**

```

process (A, B)
  variable V: Std_logic;
begin
  V := '0';
  for I in 0 to 3 loop
    F(I) <= A(I) and B(3-I);
    V := V xor A(I);
  end loop;
  G <= V;
end process;

```



# LOOPS

**There are 3 different kinds of loop statements in VHDL:**

- 1) The while loop, tests a boolean condition at the top of the loop, and only leaves the loop when the condition is false.**

```
while CONDITION loop
```

```
  . . .
```

```
end loop;
```

**2) The unbounded loop statement simply loops forever.**

Loop

...

**exit;**

...

**exit when CONDITION;**

...

end loop;

**3) for loop**

**Only the for loop is synthesizable, and that only if the bounds are constant. The other loops can be useful in test benches and behavioural models.**

**L1: for I in 0 to 7 loop**

**L2: for J in 0 to 7 loop**

**C := C + 1;**

**exit L2 when A(J) = B(I);**

**exit L1 when B(C) = 'U';**

**end loop L2;**

**end loop L1;**

```
ClockGenerator_1: process  
begin  
  for I in 1 to 1000 loop  
    Clock <= '0';  
    wait for 5 NS;  
    Clock <= '1';  
    wait for 10 NS;  
  end loop;  
  wait;  
end process ClockGenerator_1;
```

```
ClockGenerator_2: process  
begin  
  while NOW < 15 US loop  
    Clock <= '0';  
    wait for 5 NS;  
    Clock <= '1';  
    wait for 10 NS;  
  end loop;  
  wait;  
end process ClockGenerator_2;
```

```
ClockGenerator_3: process  
begin  
  loop  
    Clock <= '0';  
    wait for 5 NS;  
    Clock <= '1';  
    wait for 10 NS;  
    exit when NOW >= 15 US;  
  end loop;  
  wait;  
end process ClockGenerator_3;
```

# Sequential Logic:Latches

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity LATCH is
    port (ENB, D: in STD_LOGIC;
          Q: out STD_LOGIC);
end;
architecture BEHAVIOUR of LATCH
is
begin
    process (ENB, D)
    begin
        if ENB = '1' then
            Q <= D;
        end if;
    end process;
end;
```



# Edge Triggered Flip-Flops

This is achieved by using the signal attribute 'EVENT' as follows:

```
process (Clock)
begin

    if Clock'EVENT and Clock = '1' then

        Q0 <= D0;
        Q1 <= D1;

    end if;

end process;
```

Each signal has a signal attribute S'EVENT, which has a boolean value True or False.

S'Event changes during any delta in which there is an event on the signal (i.e. signal changes value).

```
process
```

```
begin
```

```
...
```

```
S <= '1';
```

```
wait for 10 NS;
```

```
S <= '1';
```

```
wait for 10 NS;
```

```
S <= '0';
```

```
wait for 10 NS;
```

```
S <= '0';
```

```
...
```

# Rising edge and Falling edge Functions

Testing for a clock edge is such a common requirement that the package `std_logic_1164` includes standard functions `Rising_edge` and `Falling_edge`.

The functions are given a signal of type `STD_LOGIC` and return a value which is either `True` or `False`.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
...
process (Clock)
begin

    if RISING_EDGE(Clock) then
        Q <= D;
    end if;

end process;
```

```
process (Clock)
begin
  if Clock'EVENT and Clock = '1' then
    Q <= D;
  end if;
end process;
```

```
process (Clock)
begin
  if RISING_EDGE(Clock) then
    Q <= D;
  end if;
end process;
```

Not all synthesis tools support the RISING\_EDGE and FALLING\_EDGE functions.

# Asynchronous Reset

Describing an asynchronous reset requires that the reset signal is added to the sensitivity list of the process.

```
process (Clock, Reset)
begin

    if Reset = '0' then
        Count <= "00000000";

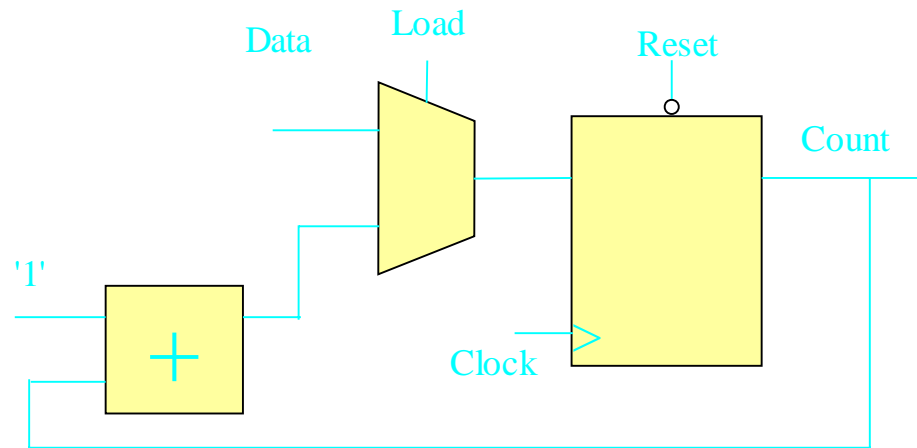
    elsif RISING_EDGE(Clock) then

        if Load = '1' then
            Count <= Data;
        else
            Count <= Count + '1';
        end if;
    end if;
end process;
```

Synchronous operations and Asynchronous operations can be described within a single process.

...

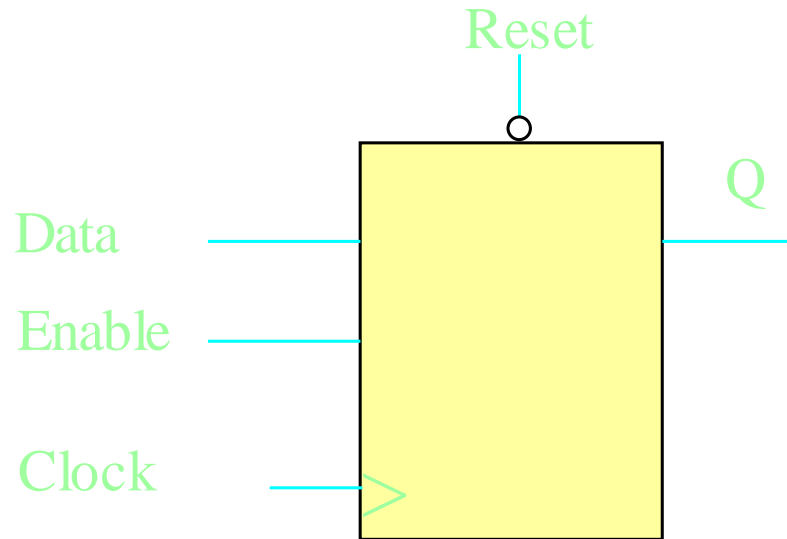
```
signal Count :  
    STD_LOGIC_VECTOR(7 downto 0);  
begin  
    process (Clock, Reset)  
    begin  
        if Reset = '0' then  
            Count <= "00000000";  
        elsif RISING_EDGE(Clock) then  
            if Load = '1' then  
                Count <= Data;  
            else  
                Count <= Count + '1';  
            end if;  
        end if;  
    end process;
```



```
process (Clock)
begin
  if RISING_EDGE(Clock) then

    if Reset = '1' then
      Q <= '0';

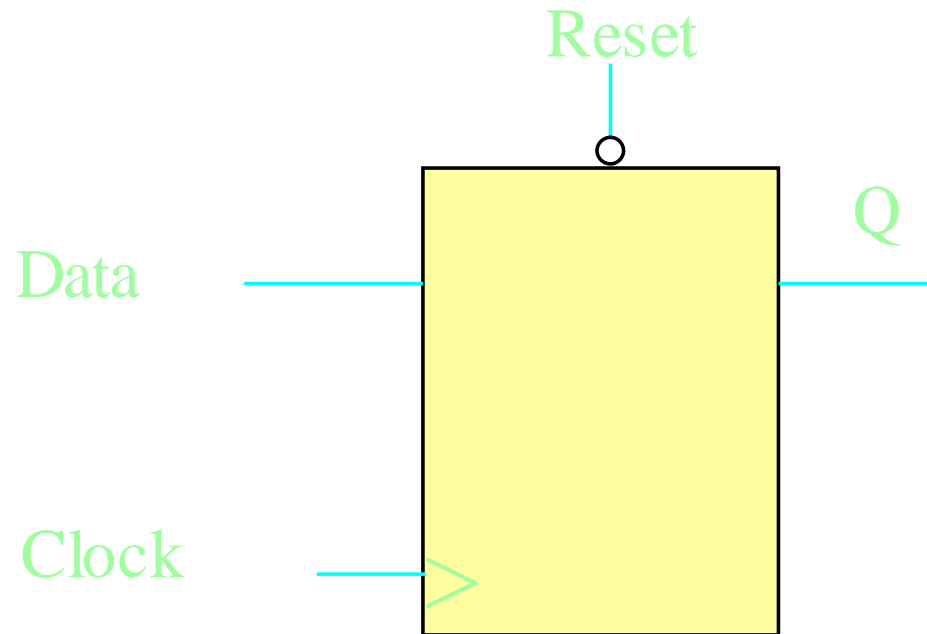
    elsif Enable = '1' then
      Q <= Data;
    end if;
  end if;
end process;
```



# Wait until Statement

It suspends the process until a condition becomes True. The condition can be any Boolean expression.

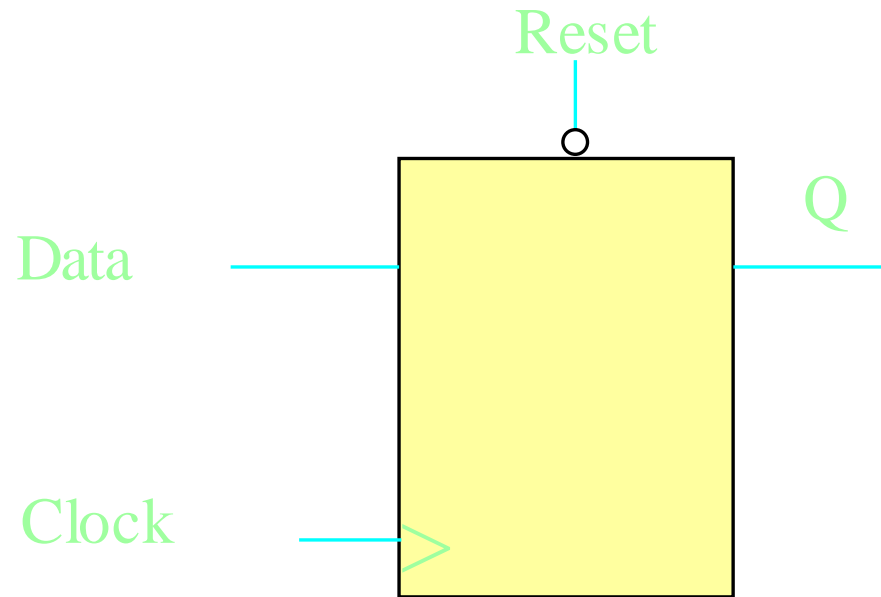
```
Another_Flipflop: process
begin
  wait until Clock = '1';
  if Reset = '1' then
    Q <= '0';
  else
    Q <= D;
  end if;
end process;
```



## Another\_Flipflop: process

```
begin
  wait until RISING_EDGE(Clock);

  if Reset = '1' then
    Q <= '0';
  else
    Q <= D;
  end if;
end process;
```



# Finite State Machines

- A finite state machine is a sequential logic circuit which moves between a finite set of states, dependent upon the values of the inputs and the previous state. The state transitions are synchronized on a clock.

There are many ways to describe a finite state machine in VHDL. The most convenient is with a process statement.

The state of the machine can be stored in a variable or signal, and the possible states conveniently represented with an enumeration type.

architecture Explicit of FSM is

begin

  process

    type StateType is (Idle, Start, Stop, Clear);

    variable State: StateType;

  begin

    wait until RISING\_EDGE(Clock);

    if Reset = '1' then

      State := Idle; F <= '0'; G <= '1';

    else

      case State is

        when Idle => State := Start; G <= '0';

        when Start => State := Stop;

        when Stop => State := Clear; F <= '1';

        when Clear => State := Idle; F <= '0';

                  G <= '1';

      end case;

    end if;

  end process;

end;

## *State Names*

Using enumeration type allows you to give symbolic names to the states, but say nothing about the hardware implementation.

You should choose meaningful names, as this makes the VHDL code easy to understand. The names will also be visible during simulation, which makes debugging easier.

In practice, it is important that finite state machines are initialized by means of an explicit reset signal.

Otherwise, there is no reliable way to get the VHDL and gate level representations of the FSM into the same known state, and thus no way to verify their equivalence.

architecture Explicit of FSM is

begin

  process

    type StateType is (Idle, Start, Stop, Clear);

    variable State: StateType;

  begin

    wait until RISING\_EDGE(Clock);

**if Reset = '1' then**

**State := Idle; F <= '0'; G <= '1';**

    else

      case State is

        when Idle => State := Start; G <= '0';

        when Start => State := Stop;

        when Stop => State := Clear; F <= '1';

        when Clear => State := Idle; F <= '0';

                  G <= '1';

      end case;

    end if;

  end process;

end;

The following description of an FSM consists of a process synchronized on a clock edge, and assigning the variable state (the state vector) and signal F and G (the outputs).

Thus, four flip flops will be synthesized, two for the state vector, and one each of the two outputs.

architecture Explicit of FSM is

begin

  process

    type StateType is (Idle, Start, Stop, Clear);

    variable State: StateType;

  begin

    wait until RISING\_EDGE(Clock);

    if Reset = '1' then

**State** := Idle; **F** <= '0'; **G** <= '1';

    else

      case State is

        when Idle => **State** := Start; **G** <= '0';

        when Start => **State** := Stop;

        when Stop => **State** := Clear; **F** <= '1';

        when Clear => **State** := Idle; **F** <= '0';  
                          **G** <= '1';

      end case;

    end if;

  end process;

end;

```

architecture SeparateDecoding of FSM is
  type StateType is (Idle, Start, Stop, Clear);
  signal State: StateType;
begin
  Change_state: process
  begin
    wait until RISING_EDGE(Clock);
    if  State = Clear or Reset = '1' then
      State <= Idle;
    elsif State = Idle then State <= Start;
    elsif State = Start then State <= Stop;
    else      State <= Clear;
    end if;
  end process;

  Output: process (State)
  begin
    F <= '0'; G <= '0';
    if  State = Clear then F <= '1';
    elsif State = Idle then G <= '1';
    end if;
  end process;
end;

```

```

architecture RegistersPlusLogic of FSM is
  type StateType is (Idle, Start, Stop, Clear);
  signal State: StateType;
begin
  Registers: process
  begin
    wait until RISING_EDGE(Clock);
    if Reset = '0' then
      State <= Idle;
    else
      State <= NextState;
    end if;
  end process;
  C_logic: process (State)
  begin
    if State = Clear then NextState <= Idle;
      F <= '1';
    elsif State = Idle then NextState <= Start;
      G <= '1';
    elsif State = Start then NextState <= Stop;
    else
      NextState <= Clear;
    end if;
  end process;
end;

```

# Concatenation

- The concatenation operator “&” is used to join together two arrays end to end to make one longer array

```
Signal A,B: STD_LOGIC_VECTOR(7 downto 0);
```

```
Signal F: STD_LOGIC_VECTOR(15 downto 0);
```

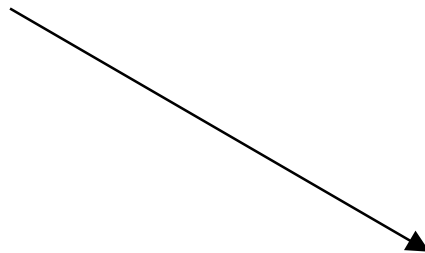
```
F<= A & B;
```

- Concatenation can also be used to concatenate arrays with single bits, or even bits with bits.

```
Signal A,B,C: STD_LOGIC;
```

```
Signal F: STD_LOGIC_VECTOR(3 downto 0);
```

```
F(3 downto 1) <= (A & B) & C;
```



F(3) = A

F(2) = B

F(1) = C

F(0) = unchanged

- Shift and rotate operations can be performed in one line by combining concatenation with slice names.

```
Signal Reg: STD_LOGIC_VECTOR(7 downto 0);
```

```
Reg <= Reg(6 downto 0) & '0';
```

Shift left one digit

```
Reg <= Reg(6 downto 0) & Reg(7);
```

Rotate left one digit

# Shift and Rotate

- In VHDL '93 these operations can be performed also using the shift and rotate operators:
- Shift left / right logical : sll srl
- Shift left / right arithmetic : sla sra
- Rotate left / right : rol ror

```
Signal Reg: STD_LOGIC_VECTOR(7 downto 0);
```

```
Reg <= Reg(6 downto 0) & '0';
```

Shift left one digit

```
Reg <= Reg sll 1;
```

```
Reg <= Reg(6 downto 0) & Reg(7);
```

Rotate left one digit

```
Reg <= Reg rol 1;
```

# Example

```
Variable A: STD_LOGIC_VECTOR (3 downto 0);
```

```
    ...  
    A:= "0110";
```

- What is the value of this expression?

```
A(0) & '1' & A(2 downto 1)
```

```
"0111"
```

# Operator Overloading

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity ADDER is
  port(A, B: in
        STD_LOGIC_VECTOR(7 downto 0);
        SUM: out
        STD_LOGIC_VECTOR(7 downto 0));
end;

architecture A1 of ADDER is
begin
  SUM <= A + B;
end;
```

- This is illegal because the operator “+” is not defined to work on type Std\_logic\_vector.

- The Std\_logic\_vector type is declared in the package Std\_logic\_1164.

```
library IEEE;  
use IEEE.STD_LOGIC_1164.all;
```

- The package also declares a set of operators which work on the type.
- Defining operators to work on new types is called operator overloading.

- There are also some operators that are implicitly defined for all array types.
- These are the relational operators

>

>=

<

<=

=

/=

- “+” is not defined in `std_logic_1164`, but it can be defined for `std_logic_vector` and put in a package.
- All VHDL synthesis tools provide a package that overloads certain arithmetic operators on `std_logic_vector` or on related types.

- Package Numeric\_Std includes new array types signed and unsigned to represent numbers and overloaded operators.

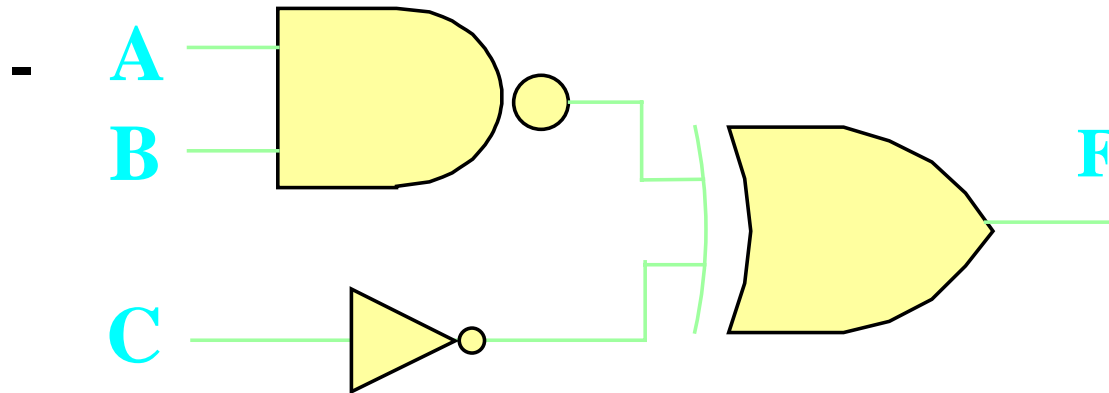
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity ADDER is
    port(A, B: in UNSIGNED(7
        downto 0);
         SUM: out UNSIGNED(7
            downto 0));
end;

architecture A1 of ADDER is
begin
    SUM <= A + B;
end;
```

**signal A, B, C: STD\_LOGIC;**

**F <= (A nand B ) xor (not C);**



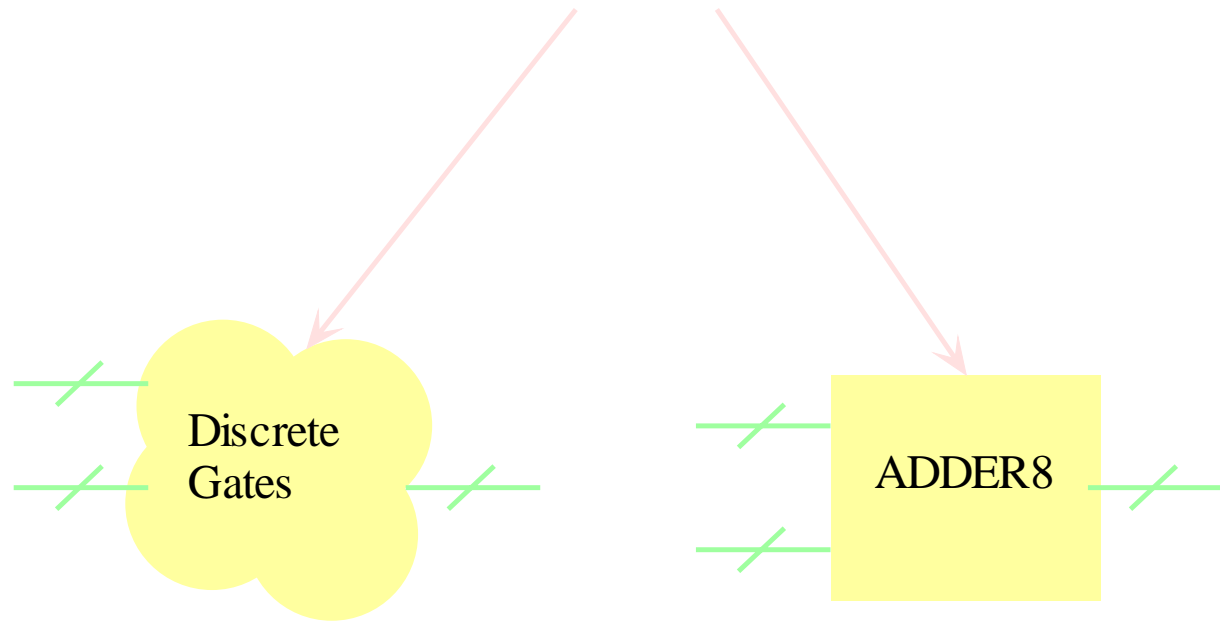
**signal A, B: STD\_LOGIC\_VECTOR;**

-  
**F <= A + B ;**

# Synthesis of Arithmetic

- The synthesis of the arithmetic operators depend on the synthesis tool.
- Some tools will map the operators to discrete gates. Others will make use of macro-cells optimized for the target technology.
- A plus operator (+) can be implemented in hardware using a ripple carry or a carry look-ahead scheme. The implementation will be smaller or faster, depending on which one you choose.

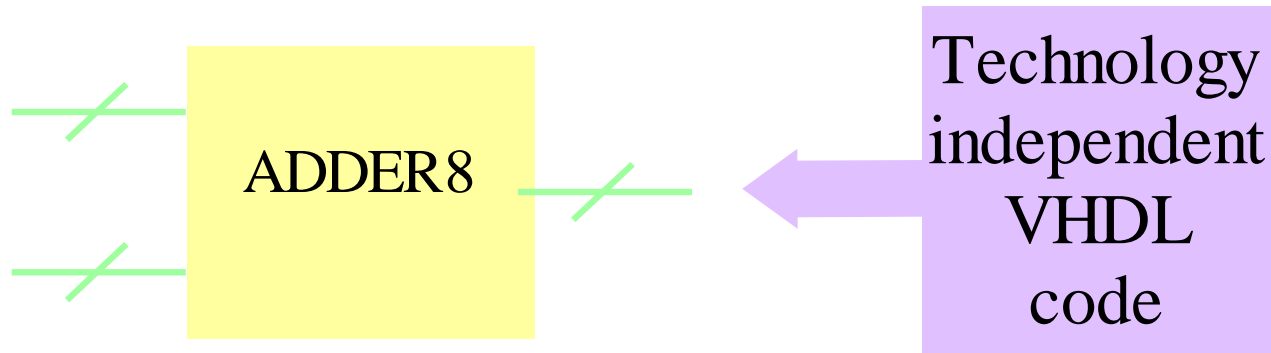
$$F \leq A + B ;$$



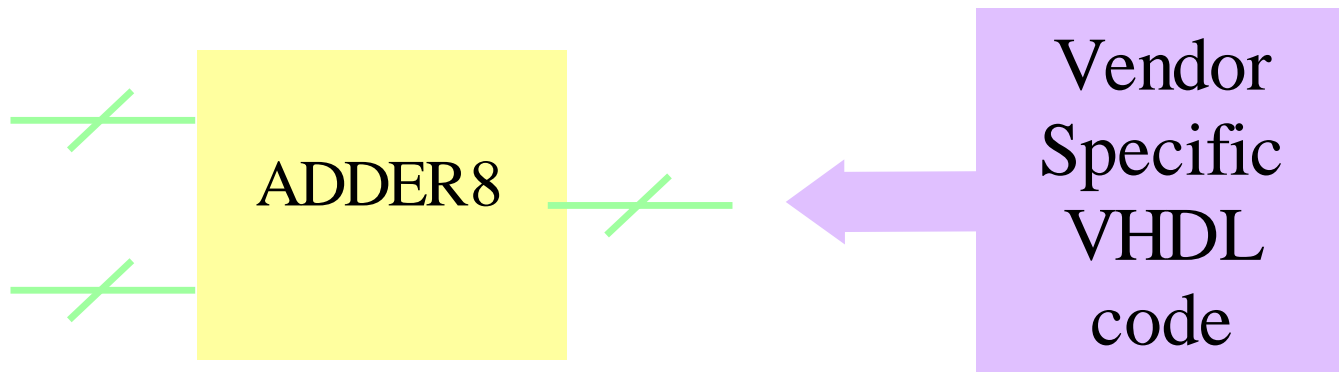
Non-optimal  
connection of  
cells

Optimized  
structure  
of cells

$$F \leq A + B ;$$

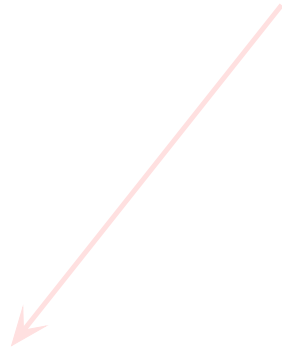


C1: ADDER8 port map (A,B,B);



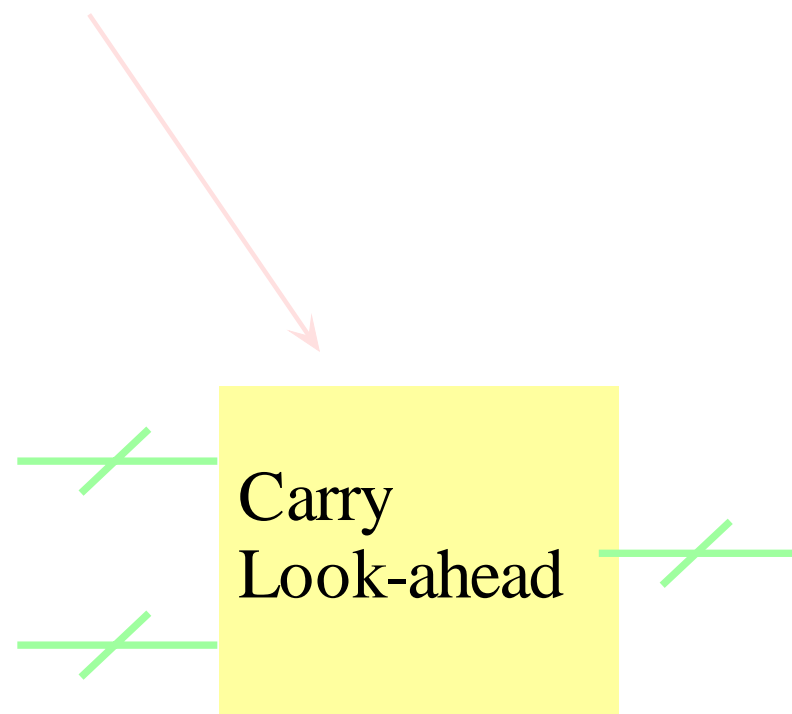
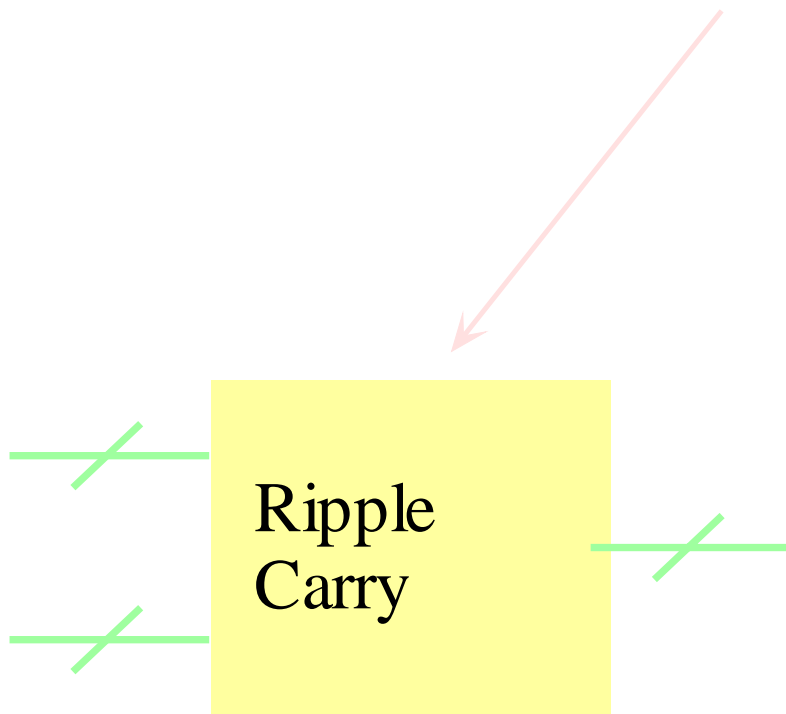
- If your synthesis tool can not map an arithmetic operator to an optimized macro-cell, you will need to instantiate the macro-cell directly in your VHDL code.
- This would make your code vendor specific.
- However, sometimes you are forced to compromise technology independence in favour of efficiency.

$F \leq A + B ;$   
+ Constraints



C1: SMALL-ADDER ... (Ripple Carry Adder)

C1: FAST\_ADDER8 ... (Carry Look-Ahead Adder)



# Resource Sharing

- Resource sharing allows a single hardware to be shared by more than one VHDL operator
- Some tools share resources automatically.

```
process (A, B, C, D, K)
```

```
begin
```

```
  if K then
```

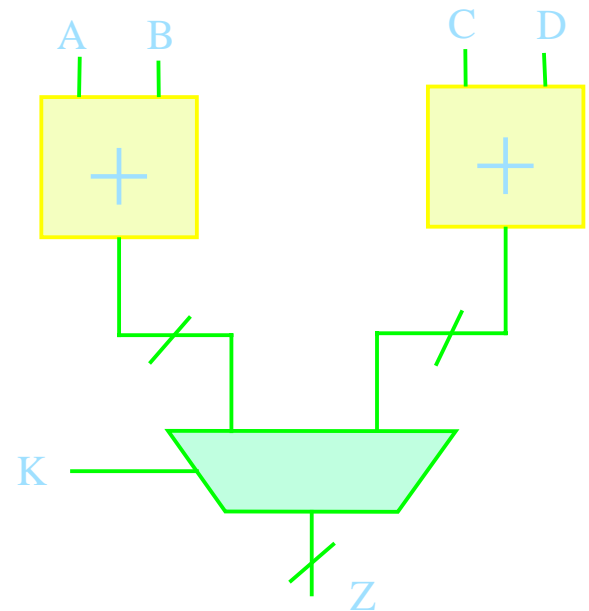
```
    Z <= A + B; -- 8 Bit
```

```
  else
```

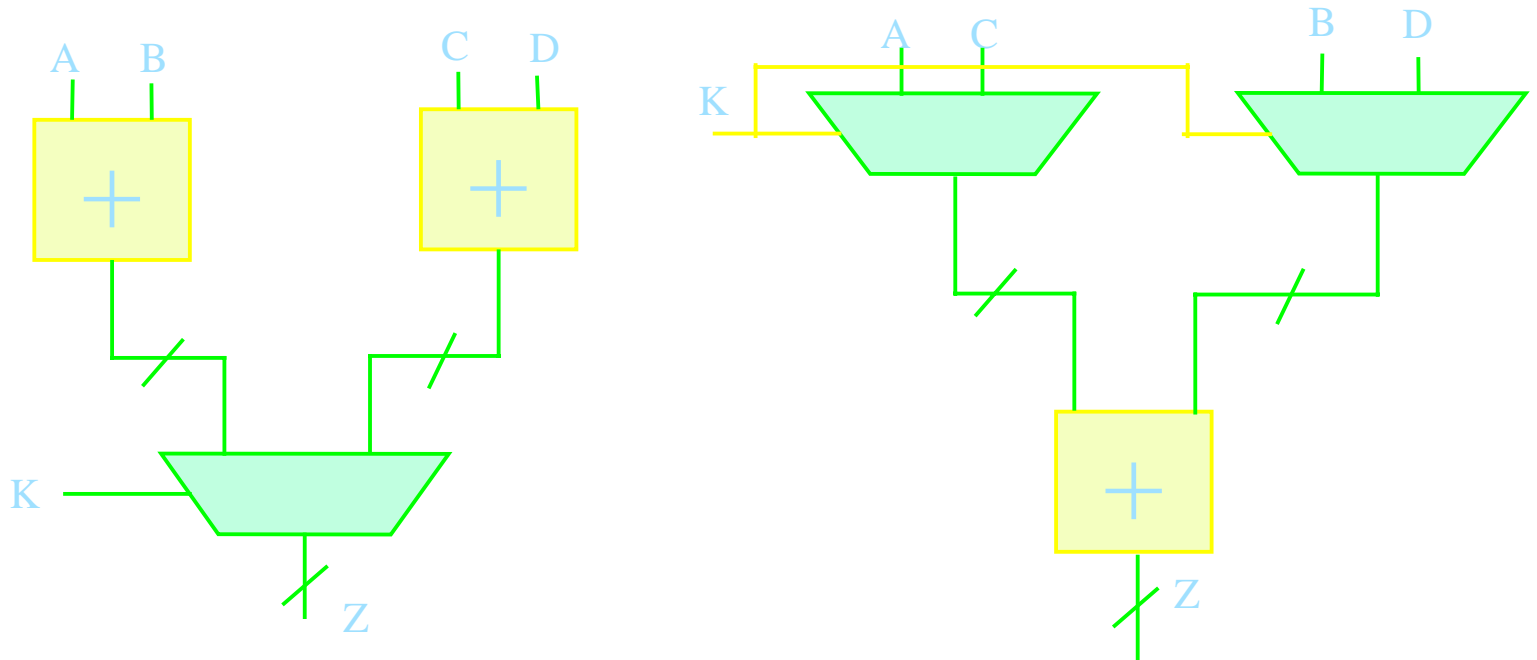
```
    Z <= C + D;
```

```
  end if;
```

```
end process;
```



```
process (A, B, C, D, K)
begin
  if K then
    Z <= A + B; -- 8 Bit
  else
    Z <= C + D;
  end if;
end process;
```



- If your tool does not do resource sharing, you must rewrite your code to achieve same results.

```
process (A, B, C, D, K)
```

```
  variable V1, V2 : ...
```

```
begin
```

```
  if K then
```

```
    V1 := A;
```

```
    V2 := B;
```

```
  else
```

```
    V1 := C;
```

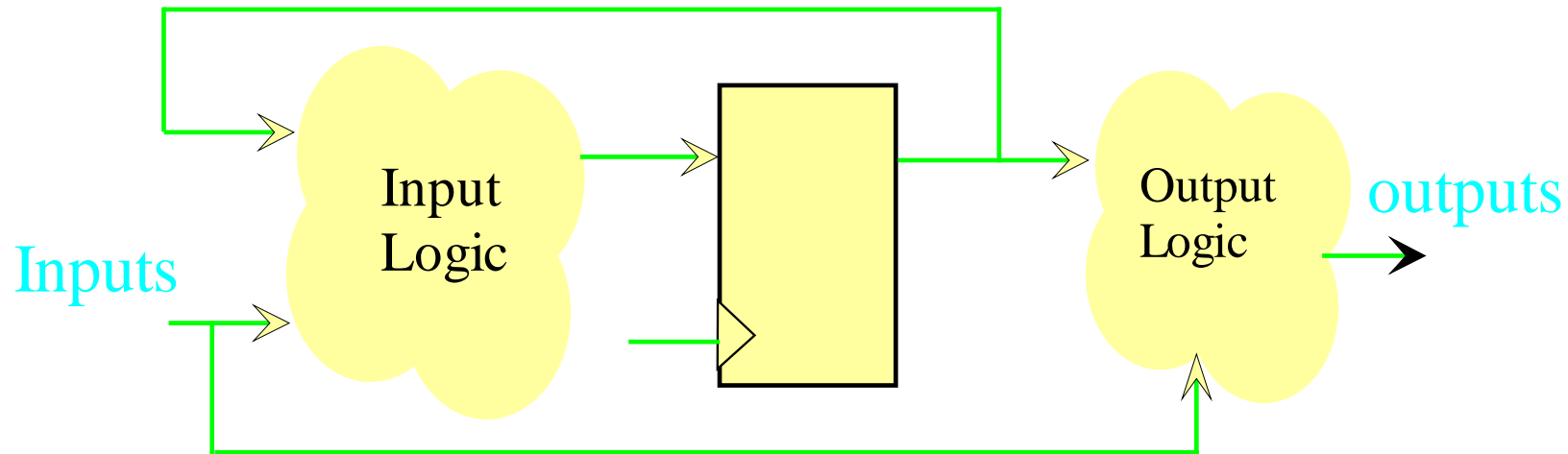
```
    V2 := D;
```

```
  end if;
```

```
  Z <= V1 + V2;
```

```
end process;
```

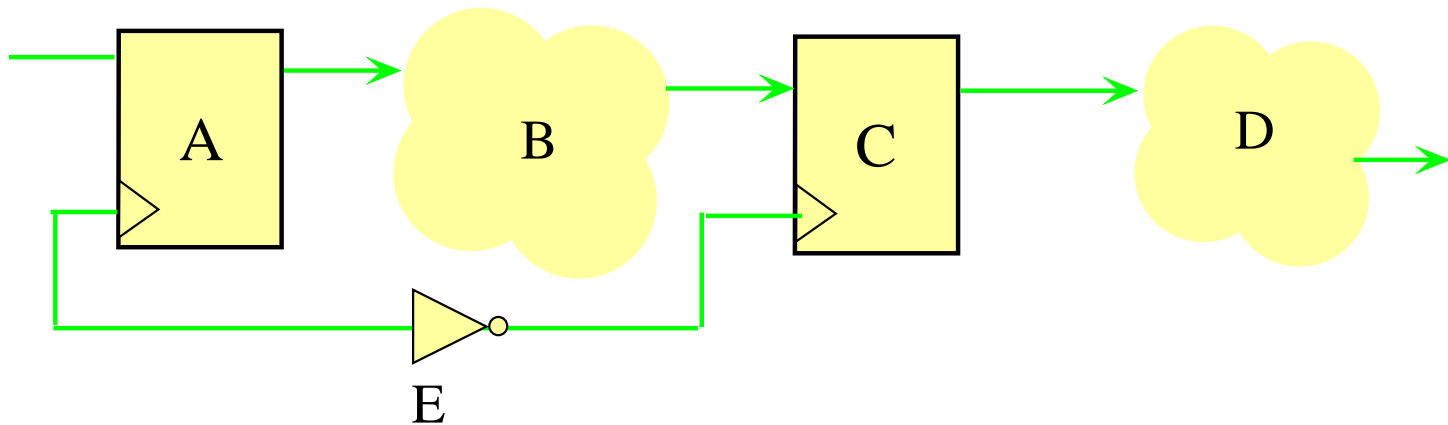
# State Machines (Again!)



- A State Machine should not have registers at the output stage.
- In order to eliminate the output registers and synthesize a pure state machine, we must re-write the VHDL description.
- In order to synthesize the correct hardware, we must split the description into at least two processes.

# Exercise

- What is the minimum number of processes necessary to describe this architecture for synthesis?



- 2 processes (AC) (BDE)
- 4 processes (AC) (B) (D) (E)
- 3 processes (A) (CE) (BD)
- 4 processes (A) (BC) (D) (E)
- 2 processes (ABCE) (D)
- 3 processes (A) (BCE) (D)

```
entity FSM1 is
port (Clock : in std_logic;
      SlowRAN: in std_logic;
      Read, Write: out std_logic);
end entity;
```

```
architecture RTL of FSM1 is
begin
```

```
  SEQ_abd_COMB: process
type StateType is (ST_Read, ST_Write, ST_Delay);
variable State:StateType := ST_Read;
```

```
begin
```

```
  wait until rising_edge*Clock);
```

```
  case State is
```

```
    when ST_Read => Read <= '1';
                                State := ST_Write;
    when ST_Write => Read <= '0';
```

```
                                Write <= '0';
```

```
                                Write <= '1';
                                if (SlowRam = '1') then
                                    State := ST_Delay;
                                else
                                    State := ST_Read
```

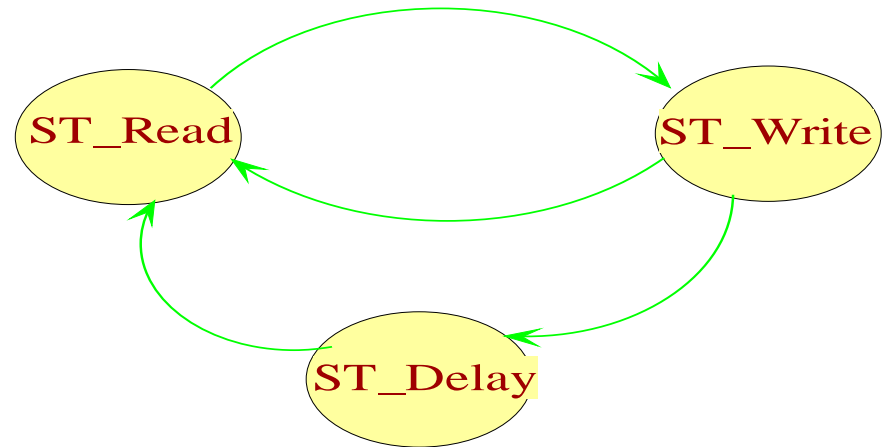
```
                                end if;
```

```
    when ST_Delay => Read <= '0';
                                Write <= '0';
                                State := ST_Read;
```

```
  end case;
```

```
  end process SEQ_AND_COMB;
```

```
end architecture RTL;
```



```

entity FSM2 is
port (Clock : in std_logic;
      SlowRAN: in std_logic;
      Read, Write: out std_logic);
end entity;

```

```

architecture RTL of FSM2 is
  type StateType is (ST_Read, ST_Write, ST_Delay);
  signal CurrentState, NextState: StateType;
begin
  SEQ: process (Clock)
begin
  if rising_edge(Clock) then
    if (Reset = '1') then
      CurrentState <= ST_Read;
    else
      CurrentState <= NextState;
    end if;
  end process SEQ;

```

```

COMB: process (CurrentState)
begin

```

```

  case CurrentState is
    when ST_Read => Read <= '1';
                    NextState := ST_Write;
    when ST_Write => Read <= '0';

```

```

                    Write <= '0';

```

```

                    Write <= '1';
    if (SlowRam = '1') then
      State := ST_Delay;
    else
      NextState := ST_Read;
    end if;

```

```

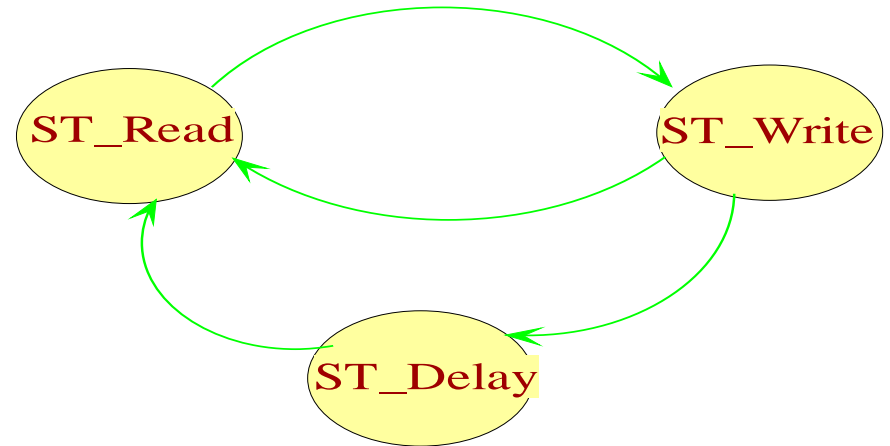
    when ST_Delay => Read <= '0';
                    Write <= '0';
                    NextState := ST_Read;

```

```

  end case;
end process COMB;
end architecture RTL;

```



```
SEQ: process (Clock)
```

```
begin
```

```
  if rising_edge(Clock) then
```

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

```
      CurrentState <= ST_Read;
```

```
    else
```

```
      CurrentState <= NextState;
```

```
    end if;
```

```
end process SEQ;
```

```
COMB: process (CurrentState)
```

```
begin
```

```
  case CurrentState is
```

```
    when ST_Read => Read <= '1';
```

```
      Write <= '0';
```

```
      NextState := ST_Write;
```

```
    when ST_Write => Read <= '0';
```

```
      Write <= '1';
```

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

```
        State := ST_Delay;
```

```
      else
```

```
        NextState := ST_Read
```

```
      end if;
```

```
    when ST_Delay => Read <= '0';
```

```
      Write <= '0';
```

```
      NextState := ST_Read;
```

```
  end case;
```

```
end process COMB;
```

