VHDL

An Introduction

Virendra Singh
Professor
Computer Architecture and Dependable Systems Lab
Department of Electrical Engineering
Indian Institute of Technology Bombay
http://www.ee.iitb.ac.in/~viren/
E-mail: viren@ee.iitb.ac.in

EE-309: Microprocessors

Lecture (24 Oct 2018)

CADSL

Modeling Digital Systems

- VHDL is for writing models of a system
- Reasons for modeling
  - requirements specification
  - documentation
  - testing using simulation
  - formal verification
  - synthesis
- Goal
  - most reliable design process, with minimum cost and time
  - avoid design errors!

What is VHDL?

- Very High Speed Integrated Circuit Hardware Description Language
- Used to describe a desired logic circuit
- Compiled, Synthesized and burned onto a working chip
- Simplifies hardware for large projects
- Examples: Combinatorial Logic, Finite State Machines

VHDL

- VHDL is a programming language that allows one to model and develop complex digital systems in a dynamic environment.
- Object Oriented methodology -- modules can be used and reused.
- Allows you to designate in/out ports (bits) and specify behavior or response of the system.

VHDL

- But VHDL is NOT C …
  There are some similarities, as with any programming language, but syntax and logic are quite different; so get over it !!

HDL Requirements

- Abstraction
- Modularity
- Concurrency
- Hierarchy

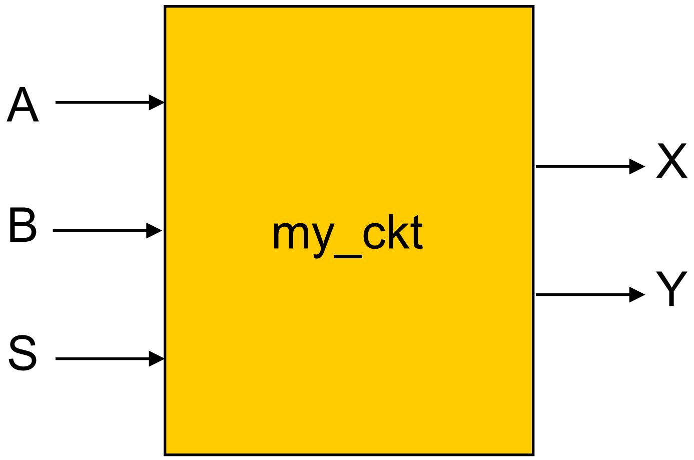
Abstraction

- VHDL supports description of components as well as systems at various level of abstraction
  - Gate and Component
  - Clock Cycle
  - Abstract behaviour without any notion of delay

Modularity

- Every component in VHDL is referred to as an entity and has clear interface
- Interface is called an entity declaration
- Internals of the component are referred to as an architecture declaration
- There can be multiple architecture at different level of abstraction associated with the same entity
- At the time of instantiation choose proper architecture

Input-Output specification of circuit



- Example: my_ckt
  - Inputs: A, B, C
  - Outputs: X, Y
- VHDL description:

```
entity my_ckt is
port (
        A:  in bit;
        B:  in bit;
        S:  in bit;
        X:  out bit;
        Y:  out bit);
end my_ckt ;
```

VHDL entity

```
• entity my_ckt is
  port (
          A: in  bit;
          B: in  bit;
          S: in  bit;
          X: out bit;
          Y: out bit
       );
  end my_ckt;
```

Datatypes:
- In-built
- User-defined

-
-
- Filename same as circuit name recommended
- Example:
  - Circuit name: my_ckt
  - Filename: my_ckt.vhd

S

Direction of port
3 main types:
- **in**: Input

Note the absence of semicolon ";" at the end of the last signal and the presence at the end of the closing bracket

Port names or Signal names

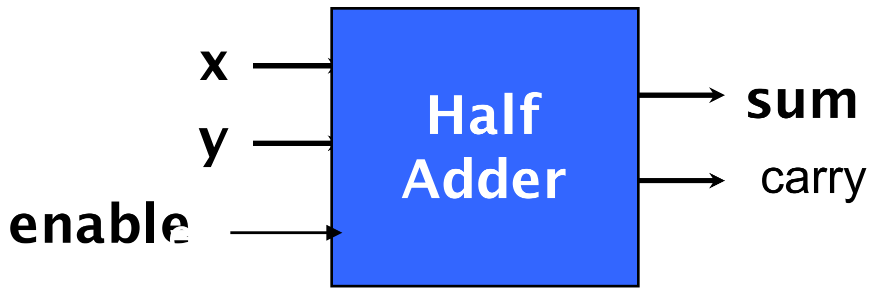
Modeling the Behavior way

- *Architecture body*
  - describes an implementation of an entity
  - may be several per entity
- *Behavioral* architecture
  - describes the algorithm performed by the module
  - contains
    - *process statements*, each containing
      - *sequential statements*, including
        - *signal assignment statements* and
        - *wait statements*

VHDL Design Example
Entity Declaration

- As a first step, the entity declaration describes the interface of the component
  - input and output *ports* are declared

```
ENTITY half_adder IS
    PORT( x, y, enable: IN BIT;
            carry, sum: OUT BIT);
END half_adder;
```



Syntax of the Architecture

```
architecture <architecture_name> of <entity_identifier> is
[<architecture_declarative_part>]
begin
  <architecture_statement_part>  -- The body of the arch.
end [architecture] [<architecture_name>];
```

- The word "architecture" in the last line is not supported before the VHDL-93 standard

VHDL Design Example
Behavioral Specification

```
ARCHITECTURE half_adder_a OF half_adder IS
   BEGIN
     PROCESS (x, y, enable)
       BEGIN
             IF enable = ‘1’ THEN
                   result <= x XOR y;
                   carry <= x AND y;
             ELSE
                   carry <= ‘0’;
                   result <= ‘0’;
             END IF;
       END PROCESS;
   END half_adder_a;
```

Concurrency in VHDL

- Achieved through processes
- Concurrent assignments are also process by itself
- These are non-terminating
- Communicating through signals
- Variables are allowed inside the processes
- Multiple processes are active at the same time

Signal Assignment

- Signals
  - Used to communicate between ***concurrently executing processes***.
  - Within a process they continue to have the form

    sig <= waveform ;

  - Means that for the signal a sequence of value updating events is to be scheduled for the future.

Variable Assignment

- Variables:
  - Exist within procedural bodies, like processes, functions, and procedures. Not visible to others.
  - Variable assignment statements appear as follows:

    var := expression;

  - Used within the sequential body just as in other procedural languages.

```
X <= Y;          X := Y;
Y <= X;          Y := X;
```

Process Statement

Is the "wrapper" around a sequential routine to compute the behavior desired for the design at a specific moment in time.

```
label: process [ (signal list) ] is
    { declarations }
begin
    { sequential statements }
    -- (typically ended by a wait statement)
end process [ label ];
```

Process Statement - A Concurrent Statement

- A process is a kind of concurrent statement.
  - *includes declarations, sequential body, and all*
- Evaluation of a process is triggered when one of a list of signals in the wait statement changes value
- Note: Just because a process is sequential does NOT mean it is modeling the sequential behavior of a design.
  - *a description of functional behavior*
  - *For example: the half-adder process example is the model of a combinational logic element.*

Process Execution Model

- Executes once (at TIME = 0) -- initialization, running till it hits a WAIT statement.
- Time advances until the wait condition is satisfied, then execution resumes.
- Executes in an endless loop,
  - interrupted only by WAIT statements;
  - bottom of the process contains an implicit "go to the top.”
- TIME DOES NOT ADVANCE within a process; it advances during a WAIT statement.

Wait Statements

```
wait_stmt <=
    [ label : ] wait [ on signal_name{ , … } ]
                     [ until boolean_expr ]
                     [ for time_expr ] ;
```

- wait;
- wait on a, b, c;
- wait until x = 1;
- wait for 100 ns;

Wait on

- process being suspended until an event takes place on any one of the signals.
- The list of signals is also called a sensitivity list.

```
half_adder: process (a, b) is
begin
  s <= a xor b after 10 ns;
  c <= a and b after 10 ns;
end process;
```

```
half_adder: process is
begin
   s <= a xor b after 10 ns;
   c <= a and b after 10 ns;
   wait on a, b;
   end process;
```

Procedural Modeling USE: High level abstraction of behavior

```
entity traffic_light_controller
    generic (    yellow_time : time;
                 min_hwygreen : time;
                 max_hwyred : time );
     port (
                 farmroad_trip : in boolean;
                 farmroad_light : out color;
                 highway_light : out color   );
end traffic_light_controller;

architecture specification of traffic_light_controller is begin
    . . .
```

Procedural Modeling USE: High level abstraction of behavior

```
architecture specification of traffic_light_controller is begin
    cycle:  process is
    begin
                    highway_light <= green;
                    farmroad_light <= red;
        wait for min_green;
        wait  until farmroad_trip;
                    highway_light <= yellow;
        wait for yellow_time_light ;
                    highway<= red;
                    farmroad_light <= green;
        wait until not farmroad_trip for max_hwyred;
              farmroad_light <= yellow;
        wait for yellow_time;
    end process;
end specification;
```

Procedural Modeling Use: Detailed Modeling of Behavior

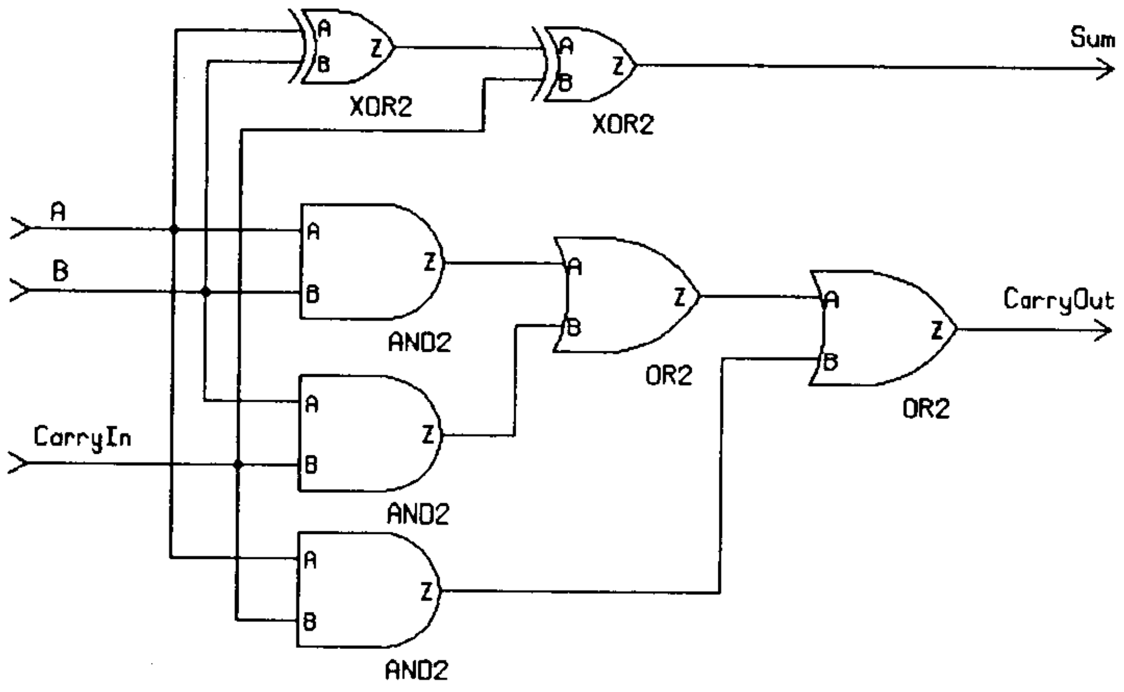
Example: Timed Behavior of Primitive Elements

```
AND_n:  process (x) is   -- x is an array of bit
            variable Zvar : bit;
begin
            Zvar := ‘1’;
            for i in x'range loop  -- for every i in the range of x
                        if x(i) = '0' then
                                    Zvar := '0' ;
                                    exit ;
                        end if;
            end loop;
            Z <= Zvar after Tprop ;
end process AND_n;
```

Logical Operators

```
library IEEE;
use IEEE.Std_logic_1164.all;
entity Full_Adder is
port (A, B, CINL: in Std_logic;
        sum, cout: out Std_logic);
end;
architecture Dataflow of Full_adder is
begin
 sum <= (A xor B) xor CIN;
Cout <= (A and B) or (B and CIN) or (A and CIN);
end;
```

Logical Operators



Logical Operators

```
signal BI, STDY, TAP: bit_vector (0 to 3)
 . . . . .
. . . . .
TAP <= BI xor STDY;
```

Arithmetic Operators

```
library IEEE;
Use IEEE.numeric_std.all;
entity Unsigned_Adder is
Port (A, B: in unsigned (0 to 3);
        sum: out  unsigned (0 to 3));
end:
architecture Simple of unsigned_adder is
begin
    sum <= A + B;
end Simple;
```

Signed Arithmetic Operators

```
library IEEE;
use IEEE.std_logic_1164.all, IEEE.numeric_std.all;
entity Diff_Adder is
port (A: in signed (2 downto 0); B: in unsigned (2 downto 0);
        C: out signed (2 downto 0);  D: out unsigned (2 downto 0));
end Diff_Adder:
architecture Two_Adder of Diff_Adder is
begin
   C <= A + “11”;
   D <= B + “11”;
end Two_Adder;
```

Relational Operators

```
library IEEE;
use IEEE.numeric_std.all;
entity GT is
port (A, B: in unsigned (3 downto 0);
        Z: out  Boolean);
end
architecture DF of GT is
begin
 Z <= A (1 down to 0) > B (3 downto 2);
end Simple;
```

Relational Operators

```
library IEEE;
use IEEE.numeric_std.all;
Entity NE is
port (A, B: in signed (0 to 7);
        Z: out  boolean);
End NE;
architecture DF of NE is
begin
 Z <= A  /= B ;
end NE;
```

Vector and Slices

```
library IEEE;
use IEEE.std_logic_1164.all;
package ARRAYS is
type BANK is array (0 to 1) of Std_logic_vector (3 downto 0);
end ARRAYS;
library IEEE;
use IEEE.std_logic_1164.all , work.arrays.all;
entity GT is
port (A, B, C: in std_logic_vector (3 downto 0);
            REG_FILE: inout BANK;
         Z: out  std_logic_vector (3 downto 0));
end GT;
```

Vector and Slices

```
architecture Example of  Vectors is
begin
 Z (3 downto 1) <= A (2) & B (3 downto 2);
    -- reading of an element and a slice and assign to slice
Z(0) <= REG_FILE (1)(3);
   --- reading an element of an array and assign to an
    element of array
REG_FILE (0) <= A and B and C;
     -- assign to one dimension of an array of an array
end Simple;
```

Non-constant Index

```
library IEEE;
use IEEE.std_logic_1164.all;
entity Non_compute-right is
port (data: in std_logic_vector (0  to 3);
          index: in natural range 0 to 3;
          dout: out  std_logic);
end Non_compute-right;
architecture Example of Non_compute-right  is
begin
   dout <= data(index);
end Example;
```

Non-constant Index

```
library IEEE;
use IEEE.std_logic_1164.all;
entity Non_compute-left is
port (addr: in natural range 0 to 7;
          store: in bit;
          mem: out  bit_vector (7 downto 0));
end Non_compute-left;
architecture Example of Non_compute-right  is
begin
    mem(addr) <= data(index);
end Example;
```

Process Statement

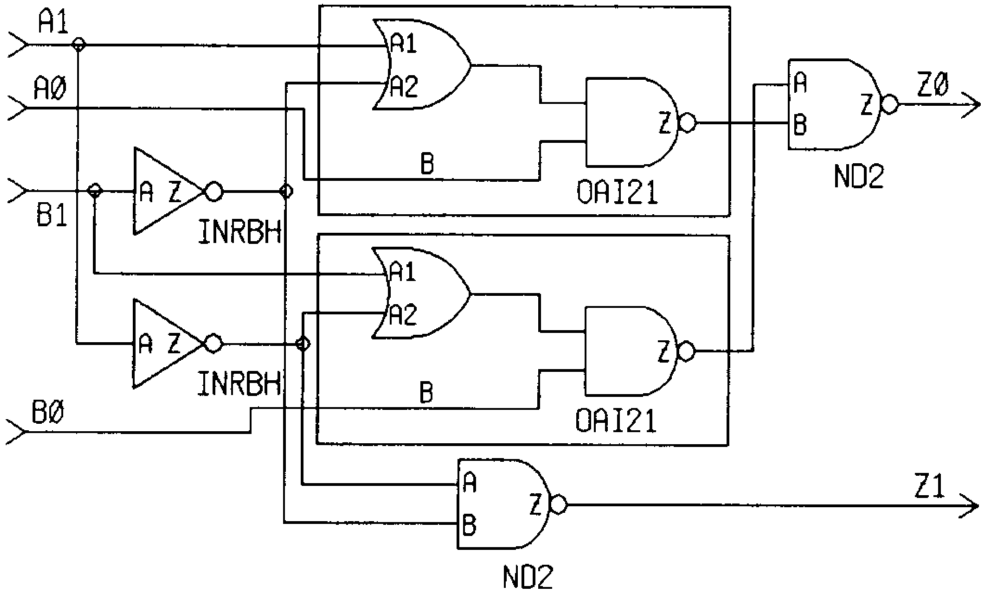
```
entity PAR is
port (A,B,C, D: in bit;
          Z: out bit);
end PAR;
architecture SEQ of PAR  is
begin
    process (A, B, C, D)
          variable Temp1, Temp2: bit;
    begin
          Temp1 := A xor B;
          Temp2 := C xor D;
          Z <= Temp1 xor Temp2;
    end process;
end Example;
```

Process Statement

```
VAR_EX: process (A, B, C)
                variable T1, T2: bit;
        begin
                T1 := A and B;
                T2 := T1 xor C;
                T1 := T2 nand A;
                Z <= T1 nor T2;
        end process VAR_EX;
```

If Statement

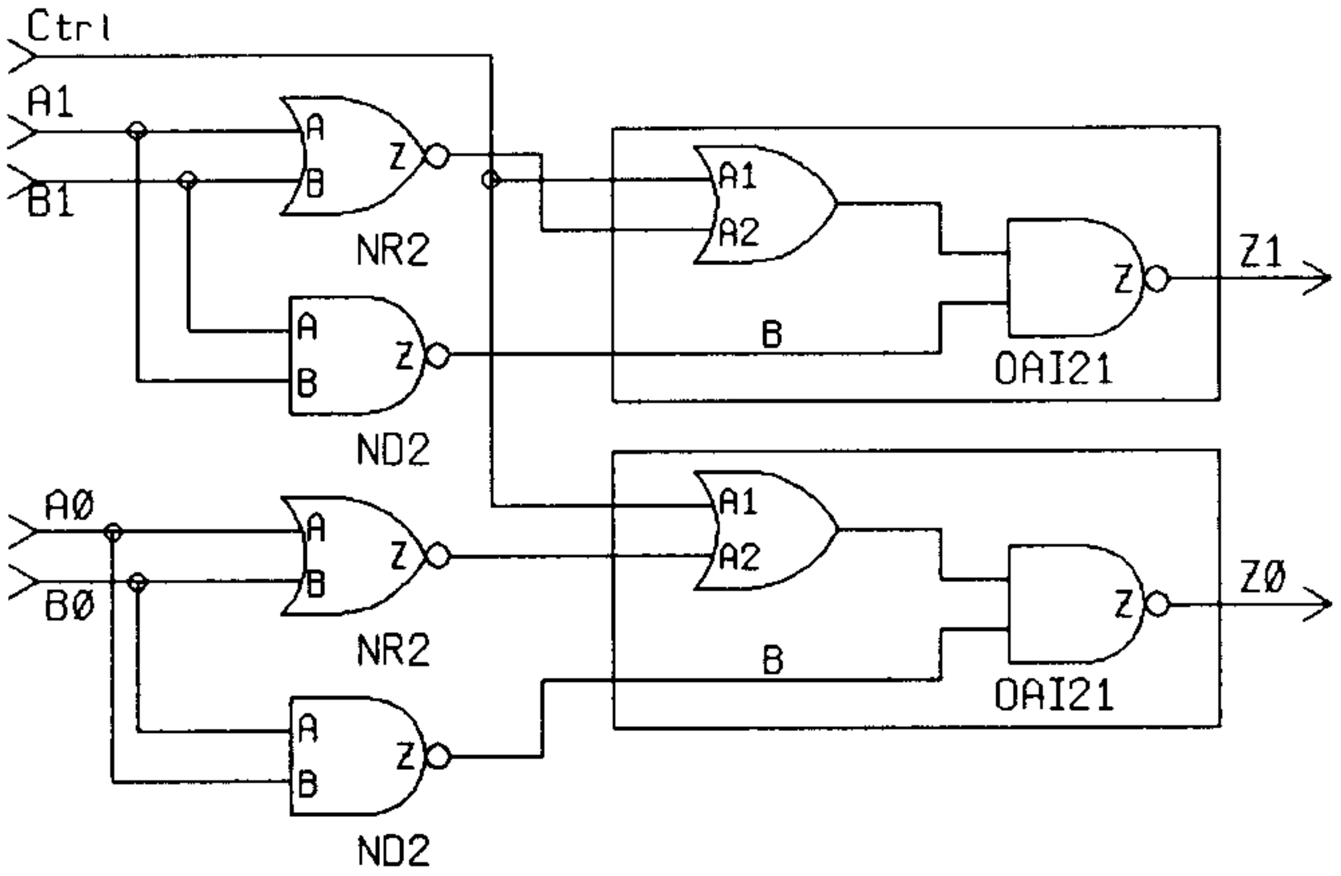
```
if  A> B
        Z := A;
else
        Z := B;
end if;
```



If Statement

```
Library IEEE
use IEEE.std_logic_1164.all;
entity Simple_alu is
port ( ctrl: in boolean;
      A, B: in bit_vector (0 to 1); Z: out bit_vector (0 to 1); ;
End Simple_alu;
architecture Example of Simple_alu  is
begin
      process (Ctrl, A, B)
      begin
            if Ctrl then
                        Z <= A and B;
            else
                        Z <= A or B;
            end if;
      end process;
end Example;
```

If Statement



If Statement

```
entity Prority is
port (Sel: in bit_vecor (0 to 3); Z: in bit_vecor (0 to 3); ;
end Pruority;
architecture SEQ of PAR  is
begin
    process (Sel)
            if Sel (0) = ‘1’ then
                        Z <= “000”;
            elsif Sel (1) = ‘1’ then
                        Z <= “001”;
            elsif Sel (2) = ‘1’ then
                        Z <= “010”;
            elsif Sel (3) = ‘1’ then
                        Z <= “011”;
            else
                        Z <= “011”; end if;
    end process;
end SEQ;
```

Inferring Latches from If Statements

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity INCR is
port (A: in bit;
          Z: out  unsigned (0 to 1));
End INCR;
```

Inferring Latches from If Statements

```
architecture Example of INCR  is
Begin
    INCR_L: process (A)
        variable ONES: unsigned (0 to 1)
        begin
            if A = ‘1’ then
                ONES := ONES + 1;
            end if;
            Z <= ONES;
        end process INCR_L;
end Example;
```

Inferring Latches from If Statements

```
Package EXAM is
    type GRADE_TYPE is (FAIL, PASS, EXCELLENT);
end;
library IEEE;
use IEEE.std_logic_1164.all;
use work.exam.all;
Entity compute is
port (marks: in natural in range 0 to 10;;
          grade: out  GRADE_TYPE);
End compute;
```

Inferring Latches from If Statements

```
architecture Example of compute  is
begin
    process (marks)
    begin
          if marks < 5 then
                     grade <= FAIL;
          elsif marks >= 5 and marks < 7 then
                     grade <= PASS;
          end if;
    end process ;
end Example;
```

Inferring Latches from If Statements

```
Package EXAM is
    type GRADE_TYPE is (FAIL, PASS, EXCELLENT);
end;
library IEEE;
use IEEE.std_logic_1164.all;
use work.exam.all;
Entity compute_mod is
port (marks: in natural in range 0 to 10;
          grade: out  GRADE_TYPE);
End compute_mod;
```

Inferring Latches from If Statements

```
architecture Example of compute_mod  is
begin
    process (marks)
    begin
        if marks < 5 then
                grade <= FAIL;
        elsif marks >= 5 and marks < 7 then
                grade <= PASS;
        else grade <= EXCELLENT;
        end if;
    end process ;
end Example;
```

Inferring Latches from If Statements: Exception for Variables

```
signal A, B, clk;
. . . .
P1: process (A, clk)
    variable p: std_logic;
begin
    if clk = ‘1’ then
           B <= p;
           p := A;
    end if;
    end process ;
```

Inferring Latches from If Statements: Exception for Variables

```
P2: process (A, clk)
    variable q:std_logic;
begin
    if clk = '1' then
          q := A;
          B <= q;
    end if;
end process P2;
end
```

Case Statements

```
Package PACK_A is
    type OP_TYPE is (ADD, SUB, MUL, DIV);
end;
library IEEE;
use IEEE.std_logic_1164.all, IEEE.numeric_std.all;
use work.exam.all;
Entity ALU is
port (OP: in OP_TYPE;
          A, B: in  unsigned (0 to 1);
          Z: out unsigned (0 to 1));
End ALU;
```

Case Statements

```
architecture Example of ALU  is
begin
    process (OP, A, B)
          variable tmp: unsigned (3 downto 0);
    begin
          case OP is
                when ADD =>
                          Z <= A + B;
                when SUB =>
                          Z <= A - B;
```

Case Statements

```
                when MUL =>
                        tmp := A * B;
                        Z <= tmp (1 downto 0);
                when DIV =>
                        Z <= A / B;
        end case;
    end process ;
end Example;
```

Case Statements

```
package COLLECT is
    type STATES is (S0, S1, S2, S3);
end;
library IEEE;
use IEEE.std_logic_1164.all, work.pack_b.all;
entity state_update is
port (curr_state: in STATES;
           Z: out integer range 0 to 3);
end state_update;
```

Case Statements

```
architecture Example of state_update  is
begin
    process (curr_state)
            variable tmp: unsigned (3 downto 0);
    begin
            case curr_state is
                    when S0 | S3 =>
                            Z <= 0;
                    when S1 =>
                            Z <= 3;
                    when others =>
                            null;
            end case;
    end process;
end Example;
```

Case Statements

```
architecture Example of state_update  is
begin
    process (curr_state)
            variable tmp: unsigned (3 downto 0);
    begin
            Z <= 0;
            case curr_state is
                    when S0 | S3 =>
                            Z <= 0;
                    when S1 =>
                            Z <= 3;
                    when others =>
                            null;
            end case;
    end process;
End Example;
```

FSM Implementation

```
architecture fsm_1 of fsm is
      signal cur_state: fsm_state;
      signal nxt_state: fsm_state;

State_change: process(clk, reset)
begin
  if (clk'event and clk=1)
     if reset = '1' then
              cur_state <= S0;
     else
              cur_state <= nxt_state;
     end if;
  end if;
end process state_change;
```

FSM Implementation

```
Next_state_logic: process(cur_state, reset)
begin
 case cur_state is
          when S0 =>
            if ready = ‘1’ then
              nxt_state <= S1;
            else
            nxt_state <= S0;
            end if;
          when S1 =>
            nxt_state <= S2;
      . . . . . . . . . . .
  end case;
end process;
```

FSM Implementation

```
Output_logic: process(cur_state)
begin
  case cur_state is
              when S0 =>
                          A <= ‘0’;
                           B <= ‘0’;
                          C <= ’00”;
                          D <= “00”;
              when S1 =>
                          A <= ‘1’;
                           B <= ‘0’;
                          C <= ’11”;
                          D <= “01”;
. . . .
   end case
end process;
end;
```

Loop Statements

3 kinds of loop in VHDL

- While-loop
- For – loop
- Loop

For-loop is supported by synthesis

Loop Statements

```
library IEEE;
use IEEE.std_logic_1164.all, IEEE.numeric_std.all;
entity DEMUX is
port (A: in unsigned (1 downto 0);
          Z: out unsigned (3 downto 0));
end DEMUX;
```

Loop Statements

```
architecture For_Loop of DEMUX  is
begin
    process (A)
        variable tmp: integer range 0 to 3;
    begin
        tmp := To_integer  (A);
        for J in Z`range loop
            if tmp = J then
                Z(J) <= '1';
            end if;
        end loop;
    end process ;
end Example;
```

Loop Statements

```
library IEEE;
use IEEE.std_logic_1164.all, IEEE.numeric_std.all;
Entity INCR is
port (clk: in std_logic;
            counter: out unsigned (1 downto 0));
end DEMUX;
architecture FLOP of INCR is
Begin
    process
    begin
            wait until clk = ‘1’;
            counter <= counter + 1;
    end process;
End FLOP;
```

Modeling Memories

- RAM can be modeled as registers
- Storage represented by array, bit vectors, or integers
- An address vector, converted to integer, is used to index the array
- Declaration of an array type and a signal of that type

```
type mem-array is array (0 to 2**depth -1) of
                std_ulogic_vector(width -1 downto 0);
signal RAM: mem_array;
```

Asynchronous RAM

- Level sensitive device
- Model like a latch
- Behaviour model

```
Asynch_RAM:   process (addr, din, we)
    begin
          if we = ‘1’ then
                    RAM(to_integer (addr)) <= din;
          end if;
    end process ;
    dout <= RAM (to_integer (addr));
```

Synchronous RAM with Asynchronous Read

- Write operation is synchronous
- Have embedded registers that store address and data
- Read is asynchronous

```
Synch_RAM:     process (clk)
    begin
          if rising_edge (clk)  then
                    if we = ‘1’ then
                              RAM(to_integer (addr)) <= din ;
                    end if;
          end if;
    end process ;
    dout <= RAM (to_integer (addr));
```

Synchronous RAM with Synchronous Read

- Have embedded registers that store address and data
- Differ in data they read when write is also performed

```
Synch_RAM:    process (clk)
    begin
        if rising_edge (clk)  then
                dout <= RAM (to_integer (addr));
                if we = '1' then
                        RAM(to_integer (addr)) <= din ;
                end if;
        end if;
    end process Synch_RAM;
```

Synchronous RAM with Synchronous Read

- Have embedded registers that store address and data
- Differ in data they read when write is also performed

```
Synch_RAM:    process (clk)
    begin
            if rising_edge (clk)  then
                    if we = '1' then
                            RAM(to_integer (addr)) <= din ;
                    end if;
                    dout <= RAM (to_integer (addr));
            end if;
    end process Synch_RAM;
```

Synchronous RAM with Synchronous Read

- Have embedded registers that store address and data
- Differ in data they read when write is also performed
- Provides new data

```
Synch_RAM:    process (clk)
    begin
        if rising_edge (clk)  then
            if we = '1' then
                RAM(to_integer (addr)) <= din ;
            else
                dout <= RAM (to_integer (addr));
            end if;
        end if;
    end process Synch_RAM;
```

Synchronous RAM with Synchronous Read

- Add enable input

```
Synch_RAM:    process (clk)
    begin
          if rising_edge (clk)  then
                if en = ‘1’ then
                      dout <= RAM (to_integer (addr));
                      if we = ‘1’ then
                            RAM(to_integer (addr)) <= din ;
                      end if;
                end if;
          end if;
    end process Synch_RAM;
```

Pipelined RAM

- Add enable input
- Data may arrive too late un clock cycle
- Add a storage register to use this data in subsequent cycle

```
Pipelined_RAM: process (clk)
    variable pipelined_en: std_ulogic;
    variable pipelined_dout: std_ulogic_vector (width -1 downto 0)
    begin
        if rising_edge (clk)  then
            if pipelined_en = ‘1’ then
                dout <= pipelined_dout;
            end if;
            pipelined_en := en;
```

Pipelined RAM

```
                    if en = '1' then
                              pipelined_dout: =RAM(to_integer (addr));
                    end if;
                    if we = '1' then
                              RAM(to_integer (addr)) <= din;
                    end if;
          end if;
          end if;
end process Pipelined_RAM;
```

RAM Initialization

- For FPGA technology
- RAM can be loaded by initial contents
- Provides new data

```
Signal  RAM : RAM_array := (X"0020", X"FC01", X"A0A0",
                            . . . .
                            others => X"0000");
```

ROM

- ROM can be modeled as RAM
- Omit code for write operation
- We can use constants as code does not change

```
constant  ROM :  mem_array := (X"0020", X"FC01", X"A0A0",
                               . . . .
                               others => X"0000");
```

- Asynchrounous read

```
dout <= ROM (to_integer (addr));
```

- For a small ROM synthesis tool can optimize it as a combinational logic

Small ROM using case statements

```
decoder: process (bcd) is
    begin
        case bcd is
                when X”0” => seg <=  “0111111”;
                when X”0” => seg <= “0000110”;
                when X”0” => seg <= “0000110”;
                . . . . . . .
                . . . . . . .
                when others => seg <= “10000000”
        end case;
    end process;
```

Small ROM using case statements

```
with bcd select
    seg <=        "0111111" when X"0",
                  "0000110" when X"1",
                  . . . . . . .
                  . . . . . .  . .
                  "10000000" when others
        end case;
    end process;
```

Resolution Function

```
Signal Z: wired_or bit;

. . . .
Z <= To-bit (Y)
. . . .


Z <= LA or BL
```

Block Statements

```
entity Block_Statement is
port (cin, din: in bit;
          rec_out: out bit);
end Block_Statement;
architecture Example of Block_Statement is
   signal stat: bit;
begin
   B1: block
          signal stat: bit;
   begin
          stat <= cin and din;
          Example.stat <= cin or din;
```

Block Statements

```
    B2: block
            signal stat: bit;
    begin
            stat <= B1.stat xor B2.stat;
            rec_out <= B2.stat;
    end block B2;
end block B1;
end Example;
```

Thank You