COE 561
Digital System Design &
Synthesis
Introduction to VHDL

Dr. Aiman H. El-Maleh
Computer Engineering Department
King Fahd University of Petroleum & Minerals

Outline …
- Hardware description languages
- VHDL terms
- Design Entity
- Design Architecture
- VHDL model of full adder circuit
- VHDL model of 1’s count circuit
- Other VHDL model examples
- Structural modeling of 4-bit comparator
- Design parameterization using Generic
... Outline

- Test Bench example
- VHDL objects
- Variables vs. Signals
- Signal assignment & Signal attributes
- Subprograms, Packages, and Libraries
- Data types in VHDL
- Data flow modeling in VHDL
- Behavioral modeling in VHDL
- FILE I/O
- VHDL coding styles for synthesis

Hardware Description Languages

- HDLs are used to describe the hardware for the purpose of modeling, simulation, testing, design, and documentation.
  - Modeling: behavior, flow of data, structure
  - Simulation: verification and test
  - Design: synthesis
- Two widely-used HDLs today
  - **VHDL**: VHSIC (Very High Speed Integrated Circuit) Hardware Description Language
  - **Verilog** (from Cadence, now IEEE standard)
Styles in VHDL

- **Behavioral**
  - High level, algorithmic, sequential execution
  - Hard to synthesize well
  - Easy to write and understand (like high-level language code)

- **Dataflow**
  - Medium level, register-to-register transfers, concurrent execution
  - Easy to synthesize well
  - Harder to write and understand (like assembly code)

- **Structural**
  - Low level, netlist, component instantiations and wiring
  - Trivial to synthesize
  - Hardest to write and understand (very detailed and low level)

---

VHDL Terms ...

- **Entity:**
  - All designs are expressed in terms of entities
  - Basic building block in a design

- **Ports:**
  - Provide the mechanism for a device to communicate with its environment
  - Define the names, types, directions, and possible default values for the signals in a component's interface

- **Architecture:**
  - All entities have an architectural description
  - Describes the behavior of the entity
  - A single entity can have multiple architectures (behavioral, structural, ...etc)

- **Configuration:**
  - A configuration statement is used to bind a component instance to an entity-architecture pair.
  - Describes which behavior to use for each entity
### VHDL Terms...

- **Generic:**
  - A parameter that passes information to an entity
  - Example: for a gate-level model with rise and fall delay, values for the rise and fall delays passed as generics

- **Process:**
  - Basic unit of execution in VHDL
  - All operations in a VHDL description are broken into single or multiple processes
  - Statements inside a process are processed sequentially

- **Package:**
  - A collection of common declarations, constants, and/or subprograms to entities and architectures.

---

### VHDL Terms...

- **Attribute:**
  - Data attached to VHDL objects or predefined data about VHDL objects
  - Examples:
    - Maximum operation temperature of a device
    - Current drive capability of a buffer

- **VHDL is NOT Case-Sensitive**
  - Begin = begin = beGiN

- **Semicolon “ ; ” terminates declarations or statements.**
- **After a double minus sign (--) the rest of the line is treated as a comment**
VHDL Models …

**PACKAGE**

DECLARATION

(Package

Generics

Entity

Ports

Architecture

ARCHITECTURE

(interface description)

(often used

functions,

constants,

components, ...)

Concurrent

Statements

Process

Sequential Statements

ARCHITECTURE

(functionality)

ARCHITECTURE

(structural)

... VHDL Models

**ENTITY**

(interface description)

**ARCHITECTURE**

(functionality)

**CONFIGURATION**

(connection entity ↔ architecture)
Design Entity …

- In VHDL, the name of the system is the same as the name of its entity.
- Entity comprises two parts:
  - parameters of the system as seen from outside such as bus-width of a processor or max clock frequency
  - connections which are transferring information to and from the system (system’s inputs and outputs)
- All parameters are declared as generics and are passed on to the body of the system
- Connections, which carry data to and from the system, are called ports. They form the second part of the entity.

Illustration of an Entity

entity Eight_bit_register is
  parameters

  CLK one-bit input

end [entity] [Eight_bit_register]
Entity Examples …

■ Entity FULLADDER is
  -- Interface description of FULLADDER
  port ( A, B, C: in bit;
        SUM, CARRY: out bit);
  end FULLADDER;

... Entity Examples

■ Entity Register is
  -- parameter: width of the register
  generic (width: integer);
  --input and output signals
  port ( CLK, Reset: in bit;
         D: in bit_vector(1 to width);
         Q: out bit_vector(1 to width));
  end Register;
Architecture Examples: Behavioral Description

- Entity FULLADDER is
  port (A, B, C: in bit;
       SUM, CARRY: out bit);
  end FULLADDER;

- Architecture CONCURRENT of FULLADDER is
  begin
  SUM <= A xor B xor C after 5 ns;
  CARRY <= (A and B) or (B and C) or (A and C) after 3 ns;
  end CONCURRENT;
### Architecture Examples: Structural Description ...

- architecture STRUCTURAL of FULLADDER is
  - signal S1, C1, C2 : bit;
  - component HA
    - port (I1, I2 : in bit; S, C : out bit);
  - component OR
    - port (I1, I2 : in bit; X : out bit);
  - begin
    - INST_HA1 : HA port map (I1 => B, I2 => C, S => S1, C => C1);
    - INST_HA2 : HA port map (I1 => A, I2 => S1, S => SUM, C => C2);
    - INST_OR : OR port map (I1 => C2, I2 => C1, X => CARRY);
  - end STRUCTURAL;

![Architecture Diagram]

---

### … Architecture Examples: Structural Description

**Entity HA is**

- PORT (I1, I2 : in bit; S, C : out bit);
- end HA;

**Architecture behavior of HA is**

- begin
  - S <= I1 xor I2;
  - C <= I1 and I2;
- end behavior;

**Entity OR is**

- PORT (I1, I2 : in bit; X : out bit);
- end OR;

**Architecture behavior of OR is**

- begin
  - X <= I1 or I2;
- end behavior;
One Entity Many Descriptions

- A system (an *entity*) can be specified with different *architectures*

```
Entity

Architecture A

Architecture B

Architecture C

Architecture D
```

Example: Ones Count Circuit

- Value of $C_1 C_0 =$ No. of ones in the inputs $A_2, A_1, and A_0$
- $C_1$ is the Majority Function ($=1$ iff two or more inputs $=1$)
- $C_0$ is a 3-Bit Odd-Parity Function (OPAR3))
- $C_1 = A_1 A_0 + A_2 A_0 + A_2 A_1$
- $C_0 = A_2 A_1' A_0' + A_2' A_1 A_0' + A_2' A_1' A_0 + A_2 A_1 A_0$

```
A0  C0 -->
A1  -->
A2  C1 -->
```
Ones Count Circuit Interface Specification

entity ONES_CNT is
  port ( A : in BIT_VECTOR(2 downto 0);
        C : out BIT_VECTOR(1 downto 0));
-- Function Documentation of ONES_CNT
-- (Truth Table Form)
--
-- | A2 A1 A0 | C1 C0 |
-- |-----------|-------|
-- | 0 0 0     | 0 0   |
-- | 0 0 1     | 0 1   |
-- | 0 1 0     | 0 1   |
-- | 0 1 1     | 1 0   |
-- | 1 0 0     | 0 1   |
-- | 1 0 1     | 1 0   |
-- | 1 1 0     | 1 0   |
-- | 1 1 1     | 1 1   |
end ONES_CNT;

Ones Count Circuit Architectural Body: Behavioral (Truth Table)

Architecture Truth_Table of ONES_CNT is
begin
  Process(A) -- Sensitivity List Contains only Vector A
  begin
      CASE A is
        WHEN "000" => C <= "00";
        WHEN "001" => C <= "01";
        WHEN "010" => C <= "01";
        WHEN "011" => C <= "10";
        WHEN "100" => C <= "01";
        WHEN "101" => C <= "10";
        WHEN "110" => C <= "10";
        WHEN "111" => C <= "11";
      end CASE;
  end process;
end Truth_Table;
Ones Count Circuit Architectural Body: Behavioral (Algorithmic)

Architecture Algorithmic of ONES_CNT is begin

Process(A) -- Sensitivity List Contains only Vector A
Variable num: INTEGER range 0 to 3;
begin
    num :=0;
    For i in 0 to 2 Loop
        IF A(i) = '1' then
            num := num+1;
        end if;
    end Loop;
    Transfer "num" Variable Value to a SIGNAL
    CASE num is
        WHEN 0 => C <= "00";
        WHEN 1 => C <= "01";
        WHEN 2 => C <= "10";
        WHEN 3 => C <= "11";
    end CASE;
end process;
end Algorithmic;

Ones Count Circuit Architectural Body: Data Flow

• C1 = A1 A0 + A2 A0 + A2 A1
• C0 = A2 A1 A0' + A2' A1 A0' + A2' A1' A0 + A2 A1 A0

Architecture Dataflow of ONES_CNT is begin

C(1)<=(A(1) and A(0)) or (A(2) and A(0))
or (A(2) and A(1));

C(0) <= (A(2) and not A(1) and not A(0))
or (not A(2) and A(1) and not A(0))
or (not A(2) and not A(1) and A(0))
or (A(2) and A(1) and A(0));
end Dataflow;
Ones Count Circuit Architectural Body: Structural ...

- \[ C_1 = A_1 A_0 + A_2 A_0 + A_2 A_1 = \text{MAJ}_3(A) \]
- \[ C_0 = A_2 A_1' A_0' + A_2' A_1 A_0' + A_2' A_1' A_0 + A_2 A_1 A_0 = \text{OPAR}_3(A) \]

**Structural Design Hierarchy**

```plaintext
Ones_CNT

C1
Majority Fun

C0
Odd-Parity Fun

AND2 OR3 NAND3 NAND4 INV
```

Ones Count Circuit Architectural Body: Structural ...

- Entity \texttt{MAJ3} is
  ```plaintext
  \text{PORT}( \texttt{X: in BIT\_Vector(2 downto 0);}\
  \texttt{Z: out BIT);}\
  \text{end MAJ3;}
  ```

- Entity \texttt{OPAR3} is
  ```plaintext
  \text{PORT}( \texttt{X: in BIT\_Vector(2 downto 0);}\
  \texttt{Z: out BIT);}\
  \text{end OPAR3;}
  ```
VHDL Structural Description of Majority Function ...

Architecture Structural of MAJ3 is

Component AND2
PORT ( I1, I2: in BIT;  O: out BIT);
end Component;

Component OR3
PORT ( I1, I2, I3: in BIT; O: out BIT);
end Component;

Declare Components To be Instantiated

begin

-- Instantiate Gates

g1: AND2 PORT MAP (X(0), X(1), A1);
g2: AND2 PORT MAP (X(0), X(2), A2);
g3: AND2 PORT MAP (X(1), X(2), A3);
g4: OR3 PORT MAP (A1, A2, A3, Z);

end Structural;

VHDL Structural Description of Majority Function

SIGNAL A1, A2, A3: BIT; Declare Maj3 Local Signals
begin

-- Instantiate Gates

g1: AND2 PORT MAP (X(0), X(1), A1);
g2: AND2 PORT MAP (X(0), X(2), A2);
g3: AND2 PORT MAP (X(1), X(2), A3);
g4: OR3 PORT MAP (A1, A2, A3, Z);

end Structural;

Wiring of Maj3 Components
VHDL Structural Description of Odd Parity Function …

Architecture Structural of OPAR3 is
Component INV
    PORT( Ipt: in BIT; Opt: out BIT);
end Component;
Component NAND3
    PORT( I1, I2, I3: in BIT;
          O: out BIT);
end Component;
Component NAND4
    PORT( I1, I2, I3, I4: in BIT;
          O: out BIT);
end Component;

SIGNAL A0B, A1B, A2B, Z1, Z2, Z3, Z4: BIT;
begin
    g1: INV PORT MAP (X(0), A0B);
g2: INV PORT MAP (X(1), A1B);
g3: INV PORT MAP (X(2), A2B);
g4: NAND3 PORT MAP (X(2), A1B, A0B, Z1);
g5: NAND3 PORT MAP (X(0), A1B, A2B, Z2);
g6: NAND3 PORT MAP (X(0), X(1), X(2), Z3);
g7: NAND3 PORT MAP (X(1), A2B, A0B, Z4);
g8: NAND4 PORT MAP (Z1, Z2, Z3, Z4, Z);
end Structural;
VHDL Top Structural Level of Ones Count Circuit

Architecture Structural of _ONES_CNT_ is
Component MAJ3
  PORT( X: in BIT_Vector(2 downto 0); Z: out BIT);
END Component ;
Component OPAR3
  PORT( X: in BIT_Vector(2 downto 0); Z: out BIT);
END Component ;
begin
  -- Instantiate Components
  c1: MAJ3 PORT MAP (A, C(1));
  c2: OPAR3 PORT MAP (A, C(0));
end Structural;

VHDL Behavioral Definition of Lower Level Components

Entity _INV_ is
  PORT( Ipt: in BIT;
       Opt: out BIT);
end _INV_;
Architecture behavior of _INV_ is
begin
  Opt <= not Ipt;
end behavior;

Entity _NAND2_ is
  PORT( I1, I2: in BIT;
        O: out BIT);
end _NAND2_;
Architecture behavior of _NAND2_ is
begin
  O <= not (I1 and I2);
end behavior;

Other Lower Level Gates Are Defined Similarly
**VHDL Model of 2x1 Multiplexer**

Entity mux2_1 IS
   Generic (dz_delay: TIME := 6 NS);
   PORT (sel, data1, data0: IN BIT; z: OUT BIT);
END mux2_1;

Architecture dataflow OF mux2_1 IS
Begin
   z <= data1 AFTER dz_delay WHEN sel='1' ELSE
      data0 AFTER dz_delay;
END dataflow;

**VHDL Model of D-FF**

Entity DFF IS
   Generic (td_reset, td_in: TIME := 8 NS);
   PORT (reset, din, clk: IN BIT; qout: OUT BIT :='0');
END DFF;

Architecture behavioral OF DFF IS
Begin
   Process(clk)
   Begin
      IF (clk = '0' AND clk'Event ) Then
         IF reset = '1' Then
            qout <= '0' AFTER td_reset ;
         ELSE
            qout <= din AFTER td_in ;
         END IF;
      END IF;
   END IF;
END process;
END behavioral ;
Divide-by-8 Counter

Entity counter IS
Generic (td_cnt: TIME := 8 NS);
PORT (reset, clk: IN BIT; counting: OUT BIT := '0');
Constant limit: INTEGER := 8;
END counter;
Architecture behavioral OF counter IS
Begin
  Process(clk)
    Variable count: INTEGER := limit;
    Begin
      IF (clk = '0' AND clk'Event) THEN
        IF reset = '1' THEN count := 0;
        ELSE IF count < limit THEN count:= count+1; END IF;
        END IF;
        IF count = limit Then  counting <= '0' AFTER td_cnt;
        ELSE counting <= '1' AFTER td_cnt;
        END IF;
      END IF;
    END process;
END behavioral;

Controller Description

- Moore Sequence Detector
  - Detection sequence is 110

IF 110 found on x
Then Z gets ‘1’
Else z gets ‘0’
End

X       IF 110 found on x
|                    Then Z gets ‘1’
|                    Else z gets ‘0’
|                    End

clk       got1 /0
            got11 /0
get110 /1

Reset /0

0 1 1 1 0
0 0 0 1 0
**VHDL Description of Moore 110 Sequence Detector**

ENTITY moore_110_detector IS  
    PORT (x, clk : IN BIT; z : OUT BIT);  
END moore_110_detector;  
ARCHITECTURE behavioral OF moore_110_detector IS  
    TYPE state IS (reset, got1, got11, got110);  
    SIGNAL current : state := reset;  
BEGIN  
    PROCESS(clk)  
    BEGIN  
        IF (clk = '1' AND CLK'Event) THEN  
            CASE current IS  
                WHEN reset =>  
                    IF x = '1' THEN current <= got1;  
                    ELSE current <= reset; END IF;  
                WHEN got1 =>  
                    IF x = '1' THEN current <= got11;  
                    ELSE current <= reset; END IF;  
                WHEN got11 =>  
                    IF x = '1' THEN current <= got110; END IF;  
                WHEN got110 =>  
                    IF x = '1' THEN current <= got1;  
                    ELSE current <= reset; END IF;  
            END CASE;  
        END IF;  
    END PROCESS;  
    z <='1' WHEN current = got110 ELSE '0';  
END behavioral;  

**Structural 4-Bit Comparator**

![Structural 4-Bit Comparator Diagram](image-url)
A Cascadable Single-Bit Comparator

- When $a > b$ the $a_{gt\_b}$ becomes 1
- When $a < b$ the $a_{lt\_b}$ becomes 1
- If $a = b$ outputs become the same as corresponding inputs

---

Structural Single-Bit Comparator

- Design uses basic components
- The less-than and greater-than outputs use the same logic
### Structural Model of Single-Bit Comparator

ENTITY bit_comparator IS
  PORT (a, b, gt, eq, lt : IN BIT; a_gt_b, a_eq_b, a_lt_b : OUT BIT);
END bit_comparator;

ARCHITECTURE gate_level OF bit_comparator IS

-- Component Configuration
FOR ALL : n1 USE ENTITY WORK.inv (single_delay);
FOR ALL : n2 USE ENTITY WORK.nand2 (single_delay);
FOR ALL : n3 USE ENTITY WORK.nand3 (single_delay);

-- Intermediate signals
SIGNAL im1, im2, im3, im4, im5, im6, im7, im8, im9, im10 : BIT;

BEGIN
  -- a_gt_b output
  g0 : n1 PORT MAP (a, im1);
  g1 : n1 PORT MAP (b, im2);
  g2 : n2 PORT MAP (a, im2, im3);
  g3 : n2 PORT MAP (a, gt, im4);
  g4 : n2 PORT MAP (im2, gt, im5);
  g5 : n3 PORT MAP (im3, im4, im5, a_gt_b);

  -- a_eq_b output
  g6 : n3 PORT MAP (im1, im2, eq, im6);
  g7 : n3 PORT MAP (a, b, eq, im7);
  g8 : n2 PORT MAP (im6, im7, a_eq_b);

  -- a_lt_b output
  g9 : n2 PORT MAP (im1, b, im8);
  g10 : n2 PORT MAP (im1, lt, im9);
  g11 : n2 PORT MAP (b, lt, im10);
  g12 : n3 PORT MAP (im8, im9, im10, a_lt_b);

END gate_level;
**Netlist Description of Single-Bit Comparator**

ARCHITECTURE netlist OF bit_comparator IS
SIGNAL im1, im2, im3, im4, im5, im6, im7, im8, im9, im10 : BIT;
BEGIN

-- a_gt_b output
  g0 : ENTITY Work.inv(single_delay) PORT MAP (a, im1);
  g1 : ENTITY Work.inv(single_delay) PORT MAP (b, im2);
  g2 : ENTITY Work.nand2(single_delay) PORT MAP (a, im2, im3);
  g3 : ENTITY Work.nand2(single_delay) PORT MAP (a, gt, im4);
  g4 : ENTITY Work.nand2(single_delay) PORT MAP (im2, gt, im5);
  g5 : ENTITY Work.nand3(single_delay) PORT MAP (im3, im4, im5, a_gt_b);

-- a_eq_b output
  g6 : ENTITY Work.nand3(single_delay) PORT MAP (im1, im2, eq, im6);
  g7 : ENTITY Work.nand3(single_delay) PORT MAP (a, b, eq, im7);
  g8 : ENTITY Work.nand2(single_delay) PORT MAP (im6, im7, a_eq_b);

-- a_lt_b output
  g9 : ENTITY Work.nand2(single_delay) PORT MAP (im1, b, im8);
  g10 : ENTITY Work.nand2(single_delay) PORT MAP (im1, lt, im9);
  g11 : ENTITY Work.nand2(single_delay) PORT MAP (b, lt, im10);
  g12 : ENTITY Work.nand3(single_delay) PORT MAP (im8, im9, im10, a_lt_b);
END netlist;

**4-Bit Comparator Iterative Structural Wiring: “For …. Generate”Statement...**

ENTITY nibble_comparator IS
  PORT (a, b : IN BIT_VECTOR (3 DOWNTO 0)); -- a and b data inputs
  gt, eq, lt : IN BIT; -- previous greater, equal & less than
  a_gt_b, a_eq_b, a_lt_b : OUT BIT); -- a > b, a = b, a < b
END nibble_comparator;

ARCHITECTURE iterative OF nibble_comparator IS

COMPONENT comp1
  PORT (a, b, gt, eq, lt : IN BIT; a_gt_b, a_eq_b, a_lt_b : OUT BIT);
END COMPONENT;
FOR ALL : comp1 USE ENTITY WORK.bit_comparator (gate_level);
SIGNAL im : BIT_VECTOR (0 TO 8);
BEGIN
  c0: comp1 PORT MAP (a(0), b(0), gt, eq, lt, im(0), im(1), im(2));
4-Bit Comparator: “For ...... Generate” Statement

```
c1to2: FOR i IN 1 TO 2 GENERATE
   c: comp1 PORT MAP ( a(i), b(i), im(i*3-3), im(i*3-2), im(i*3-1),
                      im(i*3+0), im(i*3+1), im(i*3+2) );
END GENERATE;
c3: comp1 PORT MAP (a(3), b(3), im(6), im(7), im(8), a_gt_b,
a_eq_b, a_lt_b);
END iterative;
```

- USE BIT_VECTOR for Ports a & b
- Separate first and last bit-slices from others
- Arrays FOR intermediate signals facilitate iterative wiring
- Can easily expand to an n-bit comparator

---

4-Bit Comparator: “IF ...... Generate” Statement ...

```
ARCHITECTURE iterative OF nibble_comparator IS
  --
  COMPONENT comp1
    PORT (a, b, gt, eq, lt : IN BIT; a_gt_b, a_eq_b, a_lt_b : OUT BIT);
  END COMPONENT;
  --
  FOR ALL : comp1 USE ENTITY WORK.bit_comparator (gate_level);
  CONSTANT n : INTEGER := 4;
  SIGNAL im : BIT_VECTOR ( 0 TO (n-1)*3-1);
  --
  BEGIN
    c_all: FOR i IN 0 TO n-1 GENERATE
      l: IF i = 0 GENERATE
        least: comp1 PORT MAP (a(i), b(i), gt, eq, lt, im(0), im(1), im(2) );
      END GENERATE;
  ```
4-Bit Comparator: “IF … Generate” Statement

--

\[ m: \text{IF } i = n-1 \text{ GENERATE} \]
most: comp1 PORT MAP (a(i), b(i), im(i*3-3), im(i*3-2),
im(i*3-1), a_gt_b, a_eq_b, a_lt_b);
END GENERATE;

--

\[ r: \text{IF } i > 0 \text{ AND } i < n-1 \text{ GENERATE} \]
rest: comp1 PORT MAP (a(i), b(i), im(i*3-3), im(i*3-2),
im(i*3-1), im(i*3+0), im(i*3+1), im(i*3+2));
END GENERATE;

--

END GENERATE; -- Outer Generate
END iterative;

4-Bit Comparator: Alternative Architecture (Single Generate)

ARCHITECTURE Alt_iterative OF nibble_comparator IS
constant n: Positive :=4;
COMPONENT comp1
  PORT (a, b, gt, eq, lt : IN BIT; a_gt_b, a_eq_b, a_lt_b : OUT BIT);
END COMPONENT;
FOR ALL : comp1 USE ENTITY WORK.bit_comparator (gate_level);
SIGNAL im : BIT_VECTOR ( 0 TO 3*n+2);
BEGIN
  im(0 To 2) <= gt&eq&lt;
cALL: FOR i IN 0 TO n-1 GENERATE
  c: comp1 PORT MAP (a(i), b(i), im(i*3), im(i*3+1), im(i*3+2),
im(i*3+3), im(i*3+4), im(i*3+5));
  END GENERATE;
a_gt_b <= im(3*n);
a_eq_b <= im(3*n+1);
a_lt_b <= im(3*n+2);
END Alt_iterative ;
Design Parameterization …

- GENERICs can pass design parameters
- GENERICs can include default values
- New versions of gate descriptions contain timing

```vhdl
ENTITY inv_t IS
  GENERIC (tplh : TIME := 3 NS; tphl : TIME := 5 NS);
  PORT (i1 : IN BIT; o1 : OUT BIT);
END inv_t;
```

```vhdl
ARCHITECTURE average_delay OF inv_t IS
BEGIN
  o1 <= NOT i1 AFTER (tplh + tphl) / 2;
END average_delay;
```

```vhdl
ENTITY nand2_t IS
  GENERIC (tplh : TIME := 4 NS; tphl : TIME := 6 NS);
  PORT (i1, i2 : IN BIT; o1 : OUT BIT);
END nand2_t;
```

```vhdl
ARCHITECTURE average_delay OF nand2_t IS
BEGIN
  o1 <= i1 NAND i2 AFTER (tplh + tphl) / 2;
END average_delay;
```

```vhdl
ENTITY nand3_t IS
  GENERIC (tplh : TIME := 5 NS; tphl : TIME := 7 NS);
  PORT (i1, i2, i3 : IN BIT; o1 : OUT BIT);
END nand3_t;
```

```vhdl
ARCHITECTURE average_delay OF nand3_t IS
BEGIN
  o1 <= NOT (i1 AND i2 AND i3) AFTER (tplh + tphl) / 2;
END average_delay;
```

... Design Parameterization ...

```vhdl
ENTITY nand2_t IS
  GENERIC (tplh : TIME := 4 NS; tphl : TIME := 6 NS);
  PORT (i1, i2 : IN BIT; o1 : OUT BIT);
END nand2_t;
```

```vhdl
ARCHITECTURE average_delay OF nand2_t IS
BEGIN
  o1 <= i1 NAND i2 AFTER (tplh + tphl) / 2;
END average_delay;
```

```vhdl
ENTITY nand3_t IS
  GENERIC (tplh : TIME := 5 NS; tphl : TIME := 7 NS);
  PORT (i1, i2, i3 : IN BIT; o1 : OUT BIT);
END nand3_t;
```

```vhdl
ARCHITECTURE average_delay OF nand3_t IS
BEGIN
  o1 <= NOT (i1 AND i2 AND i3) AFTER (tplh + tphl) / 2;
END average_delay;
```
Using Default values ...

ARCHITECTURE default_delay OF bit_comparator IS
Component n1 PORT (i1: IN BIT; o1: OUT BIT);
END Component;
Component n2 PORT (i1, i2: IN BIT; o1: OUT BIT);
END Component;
Component n3 PORT (i1, i2, i3: IN BIT; o1: OUT BIT);
END Component;
FOR ALL : n1 USE ENTITY WORK.inv_t (average_delay);
FOR ALL : n2 USE ENTITY WORK.nand2_t (average_delay);
FOR ALL : n3 USE ENTITY WORK.nand3_t (average_delay):
-- Intermediate signals
SIGNAL im1,im2, im3, im4, im5, im6, im7, im8, im9, im10 : BIT;
BEGIN
-- a_gt_b output
g0 : n1 PORT MAP (a, im1);
g1 : n1 PORT MAP (b, im2);
g2 : n2 PORT MAP (a, im2, im3);
g3 : n2 PORT MAP (a, gt, im4);
g4 : n2 PORT MAP (im2, gt, im5);
g5 : n3 PORT MAP (im3, im4, im5, a_gt_b);
... Using Default values

-- a_eq_b output
g6 : n3 PORT MAP (im1, im2, eq, im6);
g7 : n3 PORT MAP (a, b, eq, im7);
g8 : n2 PORT MAP (im6, im7, a_eq_b);
-- a_lt_b output
g9 : n2 PORT MAP (im1, b, im8);
g10 : n2 PORT MAP (im1, lt, im9);
g11 : n2 PORT MAP (b, lt, im10);
g12 : n3 PORT MAP (im8, im9, im10, a_lt_b);
END default_delay;

• Component declarations do not contain GENERICs
• Component instantiation are as before
• If default values exist, they are used
Assigning Fixed Values to Generic Parameters ...

ARCHITECTURE fixed_delay OF bit_comparator IS
Component n1
Generic (tplh, tphl : Time); Port (i1: in Bit; o1: out Bit);
END Component;
Component n2
Generic (tplh, tphl : Time); Port (i1, i2: in Bit; o1: out Bit);
END Component;
Component n3
Generic (tplh, tphl : Time); Port (i1, i2, i3: in Bit; o1: out Bit);
END Component;
FOR ALL : n1 USE ENTITY WORK.inv_t (average_delay);
FOR ALL : n2 USE ENTITY WORK.nand2_t (average_delay);
FOR ALL : n3 USE ENTITY WORK.nand3_t (average_delay);
-- Intermediate signals
SIGNAL im1,im2, im3, im4, im5, im6, im7, im8, im9, im10 : BIT;
BEGIN
-- a_gt_b output
    g0 : n1 Generic Map (2 NS, 4 NS) Port Map (a, im1);
    g1 : n1 Generic Map (2 NS, 4 NS) Port Map (b, im2);
    g2 : n2 Generic Map (3 NS, 5 NS) Port Map (a, im2, im3);
    g3 : n2 Generic Map (3 NS, 5 NS) Port Map (a, gt, im4);
    g4 : n2 Generic Map (3 NS, 5 NS) Port Map (im2, gt, im5);
    g5 : n3 Generic Map (4 NS, 6 NS) Port Map (im3, im4, im5, a_gt_b);
    -- a_eq_b output
    g6 : n3 Generic Map (4 NS, 6 NS) Port Map (im1, im2, eq, im6);
    g7 : n3 Generic Map (4 NS, 6 NS) PORT MAP (a, b, eq, im7);
    g8 : n2 Generic Map (3 NS, 5 NS) PORT MAP (im6, im7, a_eq_b);
    -- a_lt_b output
    g9 : n2 Generic Map (3 NS, 5 NS) Port Map (im1, b, im8);
    g10 : n2 Generic Map (3 NS, 5 NS) PORT MAP (im1, lt, im9);
    g11 : n2 Generic Map (3 NS, 5 NS) PORT MAP (b, lt, im10);
    g12 : n3 Generic Map (4 NS, 6 NS) PORT MAP (im8, im9, im10, a_lt_b);
END fixed_delay;

... Assigning Fixed Values to Generic Parameters

\[ g3 : \text{n2 Generic Map (3 NS, 5 NS) Port Map } (\text{a, gt, im4}); \]
\[ g4 : \text{n2 Generic Map (3 NS, 5 NS) Port Map } (\text{im2, gt, im5}); \]
\[ g5 : \text{n3 Generic Map (4 NS, 6 NS) Port Map } (\text{im3, im4, im5, a_gt_b}); \]
\[ \quad \text{-- a_eq_b output} \]

\[ g6 : \text{n3 Generic Map (4 NS, 6 NS) Port Map } (\text{im1, im2, eq, im6}); \]
\[ g7 : \text{n3 Generic Map (4 NS, 6 NS) PORT MAP } (\text{a, b, eq, im7}); \]
\[ g8 : \text{n2 Generic Map (3 NS, 5 NS) PORT MAP } (\text{im6, im7, a_eq_b}); \]
\[ \quad \text{-- a_lt_b output} \]

\[ g9 : \text{n2 Generic Map (3 NS, 5 NS) Port Map } (\text{im1, b, im8}); \]
\[ g10 : \text{n2 Generic Map (3 NS, 5 NS) PORT MAP } (\text{im1, lt, im9}); \]
\[ g11 : \text{n2 Generic Map (3 NS, 5 NS) PORT MAP } (\text{b, lt, im10}); \]
\[ g12 : \text{n3 Generic Map (4 NS, 6 NS) PORT MAP } (\text{im8, im9, im10, a_lt_b}); \]
END fixed_delay;

• Component declarations contain **GENERICs**
• Component instantiation contain **GENERIC Values**
• **GENERIC Values** overwrite default values
Instances with OPEN Parameter Association

ARCHITECTURE iterative OF nibble_comparator IS

BEGIN
  c0: comp1
  GENERIC MAP (Open, Open, 8 NS, Open, Open, 10 NS)
  PORT MAP (a(0), b(0), gt, eq, lt, im(0), im(1), im(2));
END iterative;

ARCHITECTURE iterative OF nibble_comparator IS

BEGIN
  c0: comp1
  GENERIC MAP (tplh3 => 8 NS, tphl3 => 10 NS)
  PORT MAP (a(0), b(0), gt, eq, lt, im(0), im(1), im(2));
END iterative;

• A GENERIC Map may specify only some of the parameters
• Using OPEN causes use of default component values
• Alternatively, association by name can be used
• Same applies to PORT MAP

Structural Test Bench

- A Testbench is an Entity without Ports that has a Structural Architecture
- The Testbench Architecture, in general, has 3 major components:
  • Instance of the Entity Under Test (EUT)
  • Test Pattern Generator (Generates Test Inputs for the Input Ports of the EUT)
  • Response Evaluator (Compares the EUT Output Signals to the Expected Correct Output)
Testbench Example …

Entity nibble_comparator_test_bench IS
  End nibble_comparator_test_bench;
End ARCHITECTURE input_output OF nibble_comparator_test_bench IS
  COMPONENT comp4 PORT (a, b : IN bit_vector (3 DOWNTO 0);
  gt, eq, lt : IN BIT;
  a_gt_b, a_eq_b, a_lt_b : OUT BIT);
  END COMPONENT;
  FOR a1 : comp4 USE ENTITY WORK.nibble_comparator(iterative);
  SIGNAL a, b : BIT_VECTOR (3 DOWNTO 0);
  SIGNAL eql, lss, gtr, gnd : BIT;
  SIGNAL vdd : BIT := '1';
  BEGIN
    a1 : comp4 PORT MAP (a, b, gnd, vdd, gnd, gtr, eql, lss);
  END input_output;

…Testbench Example

  a2 : a <= "0000", -- a = b (steady state)
  "1111" AFTER 0500 NS, -- a > b (worst case)
  "1110" AFTER 1500 NS, -- a > b (need bit 1 info)
  "0000" AFTER 3500 NS, -- a < b (worst case)
  "1100" AFTER 4000 NS, -- a > b (need bit 2 info)
  "1101" AFTER 4500 NS, -- a > b (need bit 3 info)
  "1111" AFTER 6000 NS, -- a > b (need bit 3 info, best case)
  "0000" AFTER 6500 NS, -- a = b (worst case)
  "1111" AFTER 7000 NS, -- a > b (worst case)
  END input_output;
VHDL Predefined Operators

- **Logical Operators:** NOT, AND, OR, NAND, NOR, XOR, XNOR
  - **Operand Type:** Bit, Boolean, Bit_vector
  - **Result Type:** Bit, Boolean, Bit_vector

- **Relational Operators:** =, /=, <, <=, >, >=
  - **Operand Type:** Any type
  - **Result Type:** Boolean

- **Arithmetic Operators:** +, -, *, /
  - **Operand Type:** Integer, Real
  - **Result Type:** Integer, Real

- **Concatenation Operator:** &
  - **Operand Type:** Arrays or elements of same type
  - **Result Type:** Arrays

- **Shift Operators:** SLL, SRL, SLA, SRA, ROL, ROR
  - **Operand Type:** Bit or Boolean vector
  - **Result Type:** same type

VHDL Reserved Words

<table>
<thead>
<tr>
<th>abs</th>
<th>disconnect</th>
<th>label</th>
<th>package</th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
<td>downto</td>
<td>library</td>
<td>Poll</td>
</tr>
<tr>
<td>after</td>
<td>linkage</td>
<td>procedure</td>
<td>until</td>
</tr>
<tr>
<td>alias</td>
<td>else</td>
<td>loop</td>
<td>process</td>
</tr>
<tr>
<td>all</td>
<td>elsif</td>
<td>variable</td>
<td></td>
</tr>
<tr>
<td>and</td>
<td>end</td>
<td>map</td>
<td>range</td>
</tr>
<tr>
<td>architecture</td>
<td>entity</td>
<td>mod</td>
<td>record</td>
</tr>
<tr>
<td>array</td>
<td>exit</td>
<td>nand</td>
<td>register</td>
</tr>
<tr>
<td>assert</td>
<td>new</td>
<td>rem</td>
<td>while</td>
</tr>
<tr>
<td>attribute</td>
<td>file</td>
<td>next</td>
<td>report</td>
</tr>
<tr>
<td>begin</td>
<td>for</td>
<td>nor</td>
<td>return</td>
</tr>
<tr>
<td>block</td>
<td>function</td>
<td>not</td>
<td>select</td>
</tr>
<tr>
<td>body</td>
<td>generate</td>
<td>null</td>
<td>severity</td>
</tr>
<tr>
<td>buffer</td>
<td>generic</td>
<td>of</td>
<td>signal</td>
</tr>
<tr>
<td>bus</td>
<td>guarded</td>
<td>on</td>
<td>subtype</td>
</tr>
<tr>
<td>case</td>
<td>if</td>
<td>open</td>
<td>then</td>
</tr>
<tr>
<td>component</td>
<td>in</td>
<td>or</td>
<td>to</td>
</tr>
<tr>
<td>configuration</td>
<td>inout</td>
<td>others</td>
<td>transport</td>
</tr>
<tr>
<td>constant</td>
<td>is</td>
<td>out</td>
<td>type</td>
</tr>
</tbody>
</table>
VHDL Language Grammar

- Formal grammar of the IEEE Standard 1076-1993 VHDL language in BNF format

VHDL Objects …

- VHDL OBJECT: Something that can hold a value of a given Data Type.
- VHDL has 3 classes of objects
  - CONSTANTS
  - VARIABLES
  - SIGNALS
- Every object & expression must unambiguously belong to one named Data Type
- Every object must be Declared.
... VHDL Object ...

Syntax

```
Obj_Class <id_list> : Type/SubType [signal_kind] ::= expression;
```

- **Constant**: Values of Constants **must** be specified when declared.
- **Initial values**: Initial values of Variables or Signals **may** be specified when declared.
- **Default values**: If not explicitly specified, Initial values of Variables or Signals **default** to the value of the Left Element in the type range specified in the declaration.

**Examples**:
```
• Constant Rom_Size : Integer := 2**16;
• Constant Address_Field : Integer := 7;
• Constant Ovfl_Msg : String (1 To 20) := "Accumulator OverFlow";
• Variable Busy, Active : Boolean := False;
• Variable Address : Bit_Vector (0 To Address_Field) := "00000000";
• Signal Reset : Bit := '0';
```
Variables vs. Signals

<table>
<thead>
<tr>
<th>VARIABLES</th>
<th>SIGNALS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables are only Local and May Only Appear within the Body of a Process or a SubProgram</td>
<td>Signals May be Local or Global</td>
</tr>
<tr>
<td>Variable Declarations Are Not Allowed in Declarative Parts of Architecture Bodies or Blocks</td>
<td>Signals May not be Declared within Process or Subprogram Bodies</td>
</tr>
<tr>
<td>All Port Declarations Are for Signals</td>
<td></td>
</tr>
</tbody>
</table>

A Variable Has No Hardware Correspondence A Signal Represents a Wire or a Group of Wires (BUS)

Variables Have No Time Dimension Associated With Them (Variable Assignment occurs instantaneously) Signals Have Time Dimension (Minimum Delay - Δ)

Variable Assignment Statement is always SEQUENTIAL Signal Assignment Statement is Mostly CONCURRENT (Within Architectural Body). It Can Be SEQUENTIAL (Within Process Body)

Variable Assignment Operator is := Signal Assignment Operator is :=

Variables Within Process Bodies are STATIC, i.e., a Variable Keeps its Value from One Process Call to Another. Variable Values are Not held from one Call to Another.

Signal Assignments ...

- **Syntax:**
  Target Signal <= [ Transport ] Waveform;
  Waveform := Waveform_element {, Waveform_element}
  Waveform_element := Value_Expression [ After Time_Expression ]

- **Examples:**
  - X <= '0'; -- Assignment executed After δ delay
  - S <= '1' After 10 ns;
  - Q <= Transport '1' After 10 ns;
  - S <= '1' After 5 ns, '0' After 10 ns, '1' After 15 ns;

- **Signal assignment statement**
  - mostly concurrent (within architecture bodies)
  - can be sequential (within process body)
... Signal Assignments

- Concurrent signal assignments are order independent
- Sequential signal assignments are order dependent
- Concurrent signal assignments are executed
  - Once at the beginning of simulation
  - Any time a signal on the right hand side changes

* A Signal Has a
  1- Current Value (CV), and
  2- Projected WaveForm (P_Wfm)

* A WaveForm: Consists of a Set of Signal Transactions
* Signal Transaction = Pair (Value, Time)

Signal Transaction

- When the time element of a signal transaction expires (t=0)
  - Its associated value is made the current value (CV) of a signal
  - The transaction is deleted from the list of transactions forming the Projected Waveform (P_Wfm) of the signal
Signal Transaction & Event ...

- When a new value is assigned to a signal, it is said that
  - a Transaction has been *Scheduled* for this signal
  - or a Transaction has been placed on this *Signal Driver*

- A Transaction which does not cause a signal transition (Event) is still a Transaction
- A Transaction *May/May not* cause a signal transition (Event) on the target signal

![Transaction Diagram](image1)

**Delta Delay**

- If no Time Delay is *explicitly specified*, Signal assignment is executed after a $\delta$-delay
  - Delta is a simulation cycle, and not a real time
  - Delta is used for scheduling
  - A million deltas do not add to a femto second

```vhdl
ARCHITECTURE concurrent
  OF timing_demo IS
  SIGNAL a, b, c : BIT := '0';
  BEGIN
    a <= '1';
    b <= NOT a;
    c <= NOT b;
    END concurrent;
```
Signal Assignment & Delay Types

- Two types of signal delay: Inertial and Transport

**Transport Delay**
- Exact delayed version of input signal no matter how short the input stimulus
- **Transport** keyword must be used
  - Example: S<= TRANSPORT waveform after 5 ns;
- Models delays through transmission lines and networks with virtually infinite frequency response

**Inertial Delay**
- A delayed version of the input waveform
- Signal changes (Glitches) that do not persist for a specified duration are missed
- Default delay type
  - Example: S<= waveform after 5 ns;
- Models capacitive networks and represents hardware inertial delay
- Can have additional Reject specification
  - Example: S<= REJECT 3 ns INERTIAL waveform after 5 ns;

Transport & Inertial Delay …

ARCHITECTURE delay OF example IS
SIGNAL target1, target2, waveform : BIT;
-- this is a comment
BEGIN
-- the following illustrates inertial delay
target1 <= waveform AFTER 5 NS;
-- the following illustrates transport delay
 target2 <= TRANSPORT waveform AFTER 5 NS;
END delay;
Entity example is
End example;
Architecture ex1 of example is
SIGNAL a, b, c, wave : BIT;
BEGIN
a <= wave after 5 ns;
b <= REJECT 2 ns INERTIAL wave after 5 ns;
c <= transport wave after 5 ns;
wave <= '1' after 5 ns, '0' after 8 ns, '1' after 15 ns, '0' after 17 ns, '1' after 25 ns;
END ex1;
### Sequential Placement of Transactions

<table>
<thead>
<tr>
<th>New Transaction is Before Already Existing</th>
<th>Transport</th>
<th>Inertial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overwrite existing transaction</td>
<td>Append new transaction</td>
<td></td>
</tr>
<tr>
<td>New Transaction is After Already Existing</td>
<td>Append new transaction</td>
<td></td>
</tr>
<tr>
<td>Overwrite existing transaction</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- $V_{\text{new}} /= V_{\text{old}}$
- $T_{\text{new}}-T_{\text{old}} > 0$ 
  - Reject
- $V_{\text{new}} /= V_{\text{old}}$
  - Overwrite existing transaction

### Signal Attributes...

- Attributes are named characteristics of an Object (or Type) which has a value that can be referenced.

- **Signal Attributes**
  - S`Event -- Is TRUE if Signal S has changed.
  - S`Stable(t) -- Is TRUE if Signal S has not changed for the last single time period. If t=0; it is written as S`Stable
  - S`Last_Value -- Returns the previous value of S before the last change.
  - S`Active -- -- Is TRUE if Signal S has had a transaction in the current simulation cycle.
  - S`Quiet(t) -- -- Is TRUE if no transaction has been placed on Signal S for the last single time period. If t=0; it is written as S`Quiet
  - S`Last_Event -- Returns the amount of time since the last value change on S.
...Signal Attributes

architecture ex of example is
signal a, a4: bit; signal a1, a2, a3 : Boolean;
begin
  a <= '0' after 5 ns, '1' after 10 ns, '1' after 15 ns, '0' after 20 ns;
  a1 <= a'event;
  a2 <= a'stable(2 ns);
  a3 <= a'quiet;
  a4 <= a'last_value;
end ex;

Subprograms...

- Subprograms consist of functions and procedures.
- Subprograms are used to
  - Simplify coding,
  - Achieve modularity,
  - Improve readability.
- Functions return values and cannot alter values of their parameters.
- Procedures used as a statement and can alter values of their parameters.
- All statements inside a subprogram are sequential.
...Subprograms

- **Subprograms**
  - Concurrent
  - Sequential

- **Concurrent subprograms exist outside of a process or another subprogram.**

- **Sequential subprograms exist in a process statement or another subprogram.**

- **A procedure exists as a separate statement in architecture or process.**

- **A function usually used in assignment statement or expression.**

---

Functions

- **Function specification:**
  - Name of the function
  - Formal parameters of the function
    - Name of the parameter
    - Type of the parameter
    - Mode IN is default & only allowed mode
    - Class constant is default
  - Return type of the function
  - Local declarations

- **A function body**
  - Must contain at least one return statement
  - May not contain a wait statement
A Left-Shift Function

Subtype Byte IS Bit_Vector (7 Downto 0);
Function SLL (V: Byte; N: Natural; Fill: Bit) Return Byte IS
    Variable Result: Byte := V;
    Begin
        For I IN 1 To N Loop
            Result := Result (6 Downto 0) & Fill;
        End Loop;
        Return Result;
    End SLL;

Using the Function

Architecture Functional Of LeftShifter IS
    Subtype Byte IS Bit_Vector (7 Downto 0);
    Function SLL (V: Byte; N: Natural; Fill: Bit) Return Byte is
        Variable Result: Byte := V;
        Begin
            For I IN 1 To N Loop
                Result := Result (6 Downto 0) & Fill;
            End Loop;
            Return Result;
        End SLL;
    Begin
        Sout <= SLL(Sin, 1, '0') After 12 ns;
    End Functional;
A Single-Bit Comparator

Entity Bit_Comparator IS111

Port ( a, b, -- data inputs
    gt, -- previous greater than
    eq, -- previous equal
    lt: IN BIT; -- previous less than
    a_gt_b, -- greater
    a_eq_b, -- equal
    a_lt_b: OUT BIT); -- less than

End Bit_Comparator;


a_gt_b = a . gt + b` . gt + a . b`
a_eq_b = a . b . eq + a` . b` . eq
a_lt_b = b . lt + a` . lt + b . a`

A Single-Bit Comparator using Functions

Architecture Functional of Bit_Comparator IS

Function fgl (w, x, gl: BIT) Return BIT IS
  Begin
    Return (w AND gl) OR (NOT x AND gl) OR (w AND NOT x);
  End fgl;

Function feq (w, x, eq: BIT) Return BIT IS
  Begin
    Return (w AND x AND eq) OR (NOT w AND NOT x AND eq);
  End feq;

Begin
  a_gt_b <= fgl (a, b, gt) after 12 ns;
  a_eq_b <= feq (a, b, eq) after 12 ns;
  a_lt_b <= fgl (b, a, lt) after 12 ns;
End Functional;
Binary to Integer Conversion Function

Function To_Integer (Bin : BIT_VECTOR) Return Integer IS
  Variable Result: Integer;
  Begin
    Result := 0;
    For I IN Bin'Range Loop
      If Bin(I) = '1' then
        Result := Result + 2**I;
      End if;
    End Loop;
    Return Result;
  End To_Integer;

Procedure Specification

- **Name of the procedure**
- **Formal parameters of the procedure**
  - Class of the parameter
    - optional
    - defaults to constant
  - Name of the parameter
  - Mode of the parameter
    - optional
    - defaults to IN
  - Type of the parameter
- **Local declarations**
A Left-Shift Procedure

Subtype Byte is Bit_Vector (7 downto 0);
Procedure SLL (Signal Vin : In Byte; Signal Vout : out Byte; N: Natural; Fill: Bit;
ShiftTime: Time) IS
  Variable Temp: Byte := Vin;
Begin
  For I IN 1 To N Loop
    Temp := Temp (6 downto 0) & Fill;
  End Loop;
  Vout <= Temp after N * ShiftTime;
End SLL;

Using the Procedure

Architecture Procedural of LeftShifter is
Subtype Byte is Bit_Vector (7 downto 0);
Procedure SLL (Signal Vin : In Byte; Signal Vout : out Byte; N: Natural;
Fill: Bit; ShiftTime: Time) IS
  Variable Temp: Byte := Vin;
Begin
  For I IN 1 To N Loop
    Temp := Temp (6 downto 0) & Fill;
  End Loop;
  Vout <= Temp after N * ShiftTime;
End SLL;
Begin
  Process (Sin)
  Begin
    SLL(Sin, Sout, 1, '0', 12 ns) ;
  End process;
End Procedural;
Binary to Integer Conversion
Procedure

Procedure Bin2Int (Bin : IN BIT_VECTOR; Int: OUT Integer) IS
  Variable Result: Integer;
  Begin
    Result := 0;
    For I IN Bin`RANGE Loop
      If Bin(I) = '1' Then
        Result := Result + 2**I;
      End If;
    End Loop;
    Int := Result;
  End Bin2Int;

Integer to Binary Conversion
Procedure

Procedure Int2Bin (Int: IN Integer; Bin : OUT BIT_VECTOR) IS
  Variable Tmp: Integer;
  Begin
    Tmp := Int;
    For I IN 0 To (Bin`Length - 1) Loop
      If ( Tmp MOD 2 = 1) Then
        Bin(I) := '1';
      Else Bin(I) := '0';
      End If;
      Tmp := Tmp / 2;
    End Loop;
  End Int2Bin;
Packages...

- A package is a common storage area used to hold data to be shared among a number of entities.
- Packages can encapsulate subprograms to be shared.
- A package consists of
  - Declaration section
  - Body section
- The package declaration section contains subprogram declarations, not bodies.
- The package body contains the subprograms’ bodies.
- The package declaration defines the interface for the package.

...Packages

- All items declared in the package declaration section are visible to any design unit that uses the package.
- A package is used by the USE clause.
- The interface to a package consists of any subprograms or deferred constants declared in the package declaration.
- The subprogram and deferred constant declarations must have a corresponding subprogram body and deferred constant value in the package body.
- Package body May contain other declarations needed solely within the package body.
  - Not visible to external design units.
**Package Declaration**

The package declaration section can contain:

- Subprogram declaration
- Type, subtype declaration
- Constant, deferred constant declaration
- Signal declaration creates a global signal
- File declaration
- Alias declaration
- Component declaration
- Attribute declaration, a user-defined attribute
- Attribute specification
- Use clause

**Package Body**

The package body main purpose is

- Define the values of deferred constants
- Specify the subprogram bodies for subprograms declared in the package declaration

The package body can also contain:

- Subprogram declaration
- Subprogram body
- Type, subtype declaration
- Constant declaration, which fills in the value for deferred constants
- File declaration
- Alias declaration
- Use clause
Existing Packages

- **Standard Package**
  - Defines primitive types, subtypes, and functions.
  - e.g. Type Boolean IS (false, true);
  - e.g. Type Bit is ('0', '1');

- **TEXTIO Package**
  - Defines types, procedures, and functions for standard text I/O from ASCII files.

Package Example for Component Declaration

Package simple_gates is
COMPONENT n1 PORT (i1: IN BIT; o1: OUT BIT); END COMPONENT ;
COMPONENT n2 PORT (i1,i2: IN BIT;o1:OUT BIT);END COMPONENT;
COMPONENT n3 PORT (i1, i2, i3: IN BIT; o1: OUT BIT); END COMPONENT;
end simple_gates;

Use work.simple_gates.all;
ENTITY bit_comparator IS
  PORT (a, b, gt, eq, lt : IN BIT; a_gt_b, a_eq_b, a_lt_b : OUT BIT);
END bit_comparator;
ARCHITECTURE gate_level OF bit_comparator IS
  FOR ALL : n1 USE ENTITY WORK.inv (single_delay);
  FOR ALL : n2 USE ENTITY WORK.nand2 (single_delay);
  FOR ALL : n3 USE ENTITY WORK.nand3 (single_delay);
  --Intermediate signals
  SIGNAL im1,im2, im3, im4, im5, im6, im7, im8, im9, im10 : BIT;
BEGIN
  -- description of architecture
END gate_level;
Package Example…

Package Shifters IS
   Subtype Byte IS Bit_Vector (7 Downto 0);
   Function SLL (V: Byte; N: Natural; Fill: Bit := '0') Return Byte;
   Function SRL (V: Byte; N: Natural; Fill: Bit := '0') Return Byte;
   Function SLA (V: Byte; N: Natural; Fill: Bit := '0') Return Byte;
   Function SRA (V: Byte; N: Natural) Return Byte;
   Function RLL (V: Byte; N: Natural) Return Byte;
   Function RRL (V: Byte; N: Natural) Return Byte;
End Shifters;

…Package Example…

Package Body Shifters IS
   Function SLL (V: Byte; N: Natural; Fill: Bit) Return Byte is
      Variable Result: Byte := V;
      Begin
         If N >= 8 Then
            Return (Others => Fill);
         End If;
         For I IN 1 TO N Loop
            Result := Result (6 Downto 0) & Fill;
         End Loop;
         Return Result;
      End SLL;
      .
      .
      .
End Shifters;
…Package Example

USE WORK.Shifters.ALL
Architecture Functional of LeftShifter IS
Begin
  Sout <= SLL(Sin, 1, '0') After 12 ns;
End Functional;

Another Package Example...

Package Basic_Utilities IS
  Type Integers IS Array (0 to 5) of Integer;
  Function fgl (w, x, gl: BIT) Return BIT;
  Function feq (w, x, eq: BIT) Return BIT;
  Procedure Bin2Int (Bin : IN BIT_VECTOR;  Int: OUT Integer);
  Procedure Int2Bin (Int: IN Integer; Bin : OUT BIT_VECTOR);
  Procedure Apply_Data (
    Signal Target: OUT Bit_Vector (3 Downto 0);
    Constant Values: IN Integers;
    Constant Period: IN Time);
  Function To_Integer (Bin : BIT_VECTOR) Return Integer;
End Basic_Utilities;
...Another Package Example...

Package Body Basic_Utilities IS
  Function fgl (w, x, gl: BIT) Return BIT IS
    Begin
      Return (w AND gl) OR (NOT x AND gl) OR (w AND NOT x);
    End fgl;
  Function feq (w, x, eq: BIT) Return BIT IS
    Begin
      Return (w AND x AND eq) OR (NOT w AND NOT x AND eq);
    End feq;
  .
  .
  .
End Basic_Utilities;

...Another Package Example

USE WORK.Basic_Utilities.ALL
Architecture Functional of Bit_Comparator IS
Begin
  a_gt_b <= fgl (a, b, gt) after 12 ns;
  a_eq_b <= feq (a, b, eq) after 12 ns;
  a_lt_b <= fgl (b, a, lt) after 12 ns;
End Functional;
Design Libraries…

- VHDL supports the use of design libraries for categorizing components or utilities.

- Applications of libraries include
  - Sharing of components between designers
  - Grouping components of standard logic families
  - Categorizing special-purpose utilities such as subprograms or types

- Two Types of Libraries
  - Working Library (WORK) (A Predefined library into which a Design Unit is Placed after Compilation.),
  - Resource Libraries (Contain design units that can be referenced within the design unit being compiled).

… Design Libraries…

- Only one library can be the Working library
- Any number of Resource Libraries may be used by a Design Entity
- There is a number of predefined Resource Libraries
- The Library clause is used to make a given library visible
- The Use clause causes Package Declarations within a Library to be visible
- Library management tasks, e.g. Creation or Deletion, are not part of the VHDL Language Standard → Tool Dependent
... Design Libraries...

- Exiting libraries
  - STD Library
    - Contains the STANDARD and TEXTIO packages
    - Contains all the standard types & utilities
    - Visible to all designs
  - WORK library
    - Root library for the user

- IEEE library
  - Contains VHDL-related standards
  - Contains the std_logic_1164 (IEEE 1164.1) package
    - Defines a nine values logic system
    - De Facto Standard for all Synthesis Tools

To make a library visible to a design
- LIBRARY libname;

The following statement is assumed by all designs
- LIBRARY WORK;

To use the std_logic_1164 package
- LIBRARY IEEE
- USE IEEE.std_logic_1164.ALL

By default, every design unit is assumed to contain the following declarations:
- LIBRARY STD, work;
- USE STD.Standard,All;
Arithmetic & Logical Operators for std_logic : Example

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity example is
port (a, b: IN std_logic_vector (7 downto 0));
end example;
architecture try of example is
signal x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12 : std_logic_vector (7 downto 0);
begin
  x1 <= not a;
  x2 <= a and b;
  x3 <= a nand b;
  x4 <= a or b;
  x5 <= a nor b;
  x6 <= a xor b;
  x7 <= a xnor b;
  x8 <= a + b;
  x9 <= a - b;
  x10 <= "*" (a, b);
end try;

Data Types

- A Data Type defines a set of values & a set of operations.
- VHDL is a strongly-typed Language. Types cannot be mixed in Expressions or in assigning values to Objects in general
Scalar Data Types

- **SYNTAX**
  - TYPE Identifier IS Type-Definition

- **Numeric Data Type**
  - Type-Definition is a Range_Constraint as follows:
  - Type-Definition := Range Initial-Value < To | DownTo> Final-Value

- **Examples**
  - TYPE address IS RANGE 0 To 127;
  - TYPE index IS RANGE 7 DownTo 0;
  - TYPE voltage IS RANGE -0.5 To 5.5;

Number Formats

- Integers have no Decimal Point.
- Integers may be Signed or Unsigned (e.g. -5 356 )
- A Real number must have either a Decimal Point, a -ive Exponent Term (Scientific Notation), or both.
- Real numbers may be Signed or Unsigned (e.g. -3.75 1E-9 1.5E-12 )

- Based Numbers:
  - Numbers Default to Base 10 (Decimal)
  - VHDL Allows Expressing Numbers Using Other Bases
  - Syntax
    - B#nnnn# -- Number nnnn is in Base B
  - Examples
    - 16#DF2# -- Base 16 Integer (HEX)
    - 8#7134# -- Base 8 Integer (OCTAL)
    - 2#10011# -- Base 2 Integer (Binary)
    - 16#65_3EB.37# -- Base 16 REAL (HEX)
Predefined Numeric Data Types

- **INTEGER** -- Range is Machine limited but At Least \((-2^{31} - 1)\) To \((2^{31} - 1)\)
- **POSITIVE** -- INTEGERS > 0
- **NATURAL** -- INTEGERS >= 0
- **REAL** -- Range is Machine limited

Enumeration Data Type

- Parenthesized ordered list of literals.
  - Each may be an identifier or a character literal.
  - The list elements are separated by commas
- A Position # is associated with each element in the List
- Position #’s begin with 0 for the Leftmost Element
- Variables & Signals of type ENUMERATION will have the leftmost element as their Default (Initial) value unless, otherwise explicitly assigned.

Examples
- **TYPE Color IS ( Red, Orange, Yellow, Green, Blue, Indigo, Violet);**
- **TYPE Tri_Level IS ( ‘0’, ‘1’, ‘Z’);**
- **TYPE Bus_Kind IS ( Data, Address, Control);**
- **TYPE state IS ( Init, Xmit, Receive, Wait, Terminal);**
Predefined Enumerated Data Types

- TYPE BIT IS ( '0' , '1') ;
- TYPE BOOLEAN IS ( False, True) ;
- TYPE CHARACTER IS (128 ASCII Chars......) ;
- TYPE Severity_Level IS (Note, Warning, Error, Failure) ;
- TYPE Std_U_Logic IS ( 
  'U', -- Uninitialized  
  'X', -- Forcing Unknown  
  '0', -- Forcing 0  
  '1', -- Forcing 1  
  'Z', -- High Impedence  
  'W', -- Weak Unknown  
  'L', -- Weak 0  
  'H', -- Weak 1  
  '-', -- Don’t Care  
) ;
- SUBTYPE Std_Logic IS resolved Std_U_Logic ;

Physical Data Type

- Specifies a Range Constraint, one Base Unit, and 0 or more secondary units.
- Base unit is indivisible, i.e. no fractional quantities of the Base Units are allowed.
- Secondary units must be integer multiple of the indivisible Base Unit.
- Examples
  TYPE Resistance IS Range 1 To Integer'High Units  
  Ohm; -- Base Unit  
  Kohm = 1000 Ohm; -- Secondary Unit  
  Mohm = 1000 Kohm; -- Secondary Unit  
  end Units ;
Predefined Physical Data Types

- Time is the ONLY predefined Physical data type

```plaintext
TYPE Time IS Range 0 To 1E20

Units
  fs;    -- Base Unit (Femto Second = 1E-15 Second)
  ps = 1000  fs; -- Pico_Secound
  ns = 1000  ps; -- Nano_Secound
  us = 1000  ns; -- Micro_Secound
  ms = 1000  us; -- Milli_Secound
  sec = 1000  ms;-- Second
  min = 60  sec; -- Minuite
  hr = 60  min; -- Hour

end Units;
```

Composite Data Types: Arrays

- Elements of an Array have the same data type
- Arrays may be Single/Multi-Dimensional
- Array bounds may be either Constrained or Unconstrained.

Constrained Arrays
- Array Bounds Are Specified
- Syntax:
  - TYPE id  Is Array ( Range_Constraint) of Type;

Examples
- TYPE word Is Array ( 0 To 7) of Bit;
- TYPE pattern Is Array ( 31 DownTo 0) of Bit;
- 2-D Arrays
  - TYPE col Is Range 0 To 255;
  - TYPE row Is Range 0 To 1023;
  - TYPE Mem_Array Is Array (row, col) of Bit;
  - TYPE Memory Is Array (row) of word;
Unconstrained Arrays

- Array Bounds not specified through using the notation RANGE<>.
- Type of each Dimension is specified, but the exact Range and Direction are not Specified.
- Useful in Interface_Lists → Allows Dynamic Sizing of Entities, e.g. Registers.
- Bounds of Unconstrained Arrays in such entities assume the Actual Array Sizes when wired to the Actual Signals.
- Example
  - TYPE Screen Is Array ( Integer Range<> , Integer Range<> ) of BIT;

Predefined Array Types

- Two UNCONSTRAINED Array Types are predefined
  - BIT_VECTOR
    - TYPE Bit_Vector Is Array ( Natural Range<> ) of Bit;
  - String
    - TYPE String Is Array ( Positive Range<> ) of Character;
- Example
  - SUBTYPE Pixel Is Bit_Vector (7 DownTo 0);
Use of Unconstrained Arrays

**TYPE integer_vector IS ARRAY (NATURAL RANGE <>) of INTEGER;**

**PROCEDURE apply_data (**

**SIGNAL target : OUT BIT_VECTOR;**

**CONSTANT values : IN integer_vector; CONSTANT period : IN TIME) IS**

**VARIABLE buf : BIT_VECTOR (target'RANGE);**

**BEGIN**

**FOR i IN values'RANGE LOOP**

**int2bin (values(i), buf);**

**target <= TRANSPORT buf AFTER i * period;**

**END LOOP;**

**END apply_data;**

- Example shows use of unconstrained arrays:
  - **target** : BIT_VECTOR
  - **values** : integer_vector
- Use 'RANGE to look up passed range
- Range will be determined when procedure is called

Unconstrained Comparator

**ENTITY n_bit_comparator IS**

**PORT (a, b : IN BIT_VECTOR; gt, eq, lt : IN BIT;**

**a_gt_b, a_eq_b, a_lt_b : OUT BIT);**

**END n_bit_comparator;**

**ARCHITECTURE structural OF n_bit_comparator IS**

**COMPONENT comp1**

**PORT (a, b, gt, eq, lt : IN BIT; a_gt_b, a_eq_b, a_lt_b : OUT BIT);**

**END COMPONENT;**

**FOR ALL : comp1 USE ENTITY WORK.bit_comparator (functional);**

**CONSTANT n : INTEGER := a’LENGTH;**

**SIGNAL im : BIT_VECTOR ( 0 TO (n-1)*3-1);**

**BEGIN**

**c_all: FOR i IN 0 TO n-1 GENERATE**

**l: IF i = 0 GENERATE**

**least: comp1 PORT MAP (a(i), b(i), gt, eq, lt, im(0), im(1), im(2) );**

**END GENERATE;**

**m: IF i = n-1 GENERATE**

**most: comp1 PORT MAP**

**a(i), b(i), im(i*3-3), im(i*3-2), im(i*3-1), a_gt_b, a_eq_b, a_lt_b);**

**END GENERATE;**

**r: IF i > 0 AND i < n-1 GENERATE**

**rest: comp1 PORT MAP**

**a(i), b(i), im(i*3-3), im(i*3-2), im(i*3-1), im(i*3+0), im(i*3+1), im(i*3+2) );**

**END GENERATE;**

**END GENERATE;**

**END structural;**

- Size of input vectors are not specified
- Comparator length is a’LENGTH
- Size will be determined when instantiated
Unconstrained Comparator Test Bench

ENTITY n_bit_comparator_test_bench IS
END n_bit_comparator_test_bench;

USE WORK.basic_utilities.ALL;
-- FROM PACKAGE USE : apply_data which uses integer_vector
ARCHITECTURE procedural OF n_bit_comparator_test_bench IS
COMPONENT comp_n
PORT (a, b : IN bit_vector; gt, eq, lt : IN BIT; a_gt_b, a_eq_b, a_lt_b : OUT BIT);
END COMPONENT;
FOR a1 : comp_n USE ENTITY WORK.n_bit_comparator(structural);
SIGNAL a, b : BIT_VECTOR (5 DOWNTO 0);
SIGNAL eql, lss, gtr : BIT;
SIGNAL vdd : BIT := '1';
SIGNAL gnd : BIT := '0';
BEGIN
a1: comp_n PORT MAP (a, b, gnd, vdd, gnd, gtr, eql, lss);
apply_data (a, 00&15&57&17, 500 NS);
apply_data (b, 00&43&14&45&11&21&44&11, 500 NS);
END procedural;

SIGNAL a determines size of comparator
SIGNAL a determines target size of apply_data
SIGNAL b determines target size of apply_data
Concatenated integers form integer_vector

Referencing Arrays & Array Elements …

- VHDL allows referencing an Array in its Entirety or by a SLICE, or Element.

- Examples
  - TYPE clock_state IS (Low, Rising, High, Falling);
  - TYPE Conversion_Array IS Array (Clock_state) of Bit;
  - Signal C_A : Conversion_Array := ('0', '1', '0', '1');
  - C_A <= ('1', '1', '0', '0'); -- Positional Association List
  - C_A <= (Low => '0', Rising => '1', High => '1', Falling => '0'); -- Named Association List
  - C_A <= (Low => '0', High => '0', OTHERS=> '1'); -- Alternative 3
  - C_A(Low) <= '0';
  - TYPE Register4 IS Array (3 Downto 0) of Bit;
  - TYPE Reg4 IS Array (0 To 3) of Bit;
  - Signal A: Register4 := ('0', '1', '0', '1'); --A(0)=’1’, A(3)=’0’
  - Signal B: Reg4 := ('0', '1', '0', '1'); --B(0)=’0’, B(3)=’1’
... Referencing Arrays & Array Elements

- **2-D Arrays**
  - TYPE Reg32 Is Array (31 DownTo 0) of Bit;
  - TYPE ROM Is Array (0 To 3) of Reg32;
  - TYPE ROM2 Is Array (0 To 4, 0 To 2) of Bit;
  - Signal A: ROM := (X"2F3C_5456", X"FF32_E7B8", X"109A_BA15", X"FFFF_FFFF");
  - Signal B: ROM2 := ( (`1`, `0`, `0`),
    (`0`, `1`, `0`),
    (`0`, `1`, `1`),
    (`1`, `0`, `1`),
    (`1`, `1`, `1`) );
  - B(1, 2) <= '0'; -- Referencing a 2-D Array Element

---

Type Compatibility & Conversion

- VHDL is a strongly-typed language.
- Compiler flags an Error whenever different types are mixed.
- Subtypes are Type compatible with their higher level Subtypes and their Root Type.
- Two Subtypes of the same type are Type- Compatible
Important Type Compatibility Rules …

- Type of an Expression assigned to an Object must be the Same as the Type of the Object.
- Operands of Predefined Operators Must be of the Same Type
- The Type of an Actual Signal is The SAME as the Type of the Formal Port it Connects To.

Example

- SubType X_int Is Integer Range 1 To 30;
- SubType Y_int Is Integer Range 1 To 30;
- Type X2_int Is Range 1 To 30;
- Type Y2_int Is Range 1 To 30;
- SubType Z_int Is X_int Range 1 To 20;

- Signal I : Integer;
- Signal x : X_int;
- Signal y : Y_int;
- Signal z : Z_int;
- Signal x2 : X2_int;
- Signal y2 : Y2_int;

… Important Type Compatibility Rules

- x <= z ; -- Legal Since Z_int is a Subtype of X
- x <= y ; --Legal
- x2 <= y2; --Illegal, Operands are not compatible
- l <= x + z ; -- Legal, both x and z are Vertical subtypes of Integer
- l <= x + y ; -- Legal, Operands are subtypes of same type
- l <= x2 + y2 ; -- Illegal, Operands are not compatible
- z <= x ; -- Legal, value of x should be within the range of z
- z <= y ; -- Legal, value of y should be within the range of z
Closely Related Types …

- A Type is closely related to itself.
- Any two Numeric types are closely related
  - I := Integer(X); -- Rounds X
  - X := Real(I);
  - I := Integer(3.5); -- Ambiguous (Implementation Dependent)
- Array Types are closely related iff:
  - Same dimensionality
  - Index Types are closely related for each dimension
  - Elements Types are the same

… Closely Related Types

- Example
  - Type Minutes Is Range 0 To 59;
  - Type Seconds Is Range 0 To 59;
  - SubType X_int Is Integer Range 1 To 30;
  - Variable x : X_int;
  - Variable M : Minutes;
  - Variable S : Seconds;
  - Variable I : Integer;
  - I := 60*M + S; -- Illegal M & S are incompatible types
  - I := 60*Integer(M) + Integer(S); -- Valid
  - I := M; -- Illegal -- Minutes is not a "SubType" of Integer
  - M := 60*S; -- Illegal -- Minutes & Seconds are not Compatible
  - M := Minutes(S/60); -- illegal -- Casting Closely Related Types
  - M := x; -- Illegal -- Minutes & X_int are not Type Compatible
  - M := Minutes(x); -- legal -- Casting Closely Related Types
Mixed Type Arithmetic

- Explicit Type Conversion is done between Closely-Related Types, e.g. REALs & INTEGERs
  - Example:
    - Variable x, y : Real;
    - Variable n,m : Integer;
    - n := INTEGER (x) * m; -- x is first converted to Integer
    - y := REAL (n) * x; -- n is first converted to Real
  - Example:
    - TYPE qit_byte IS ARRAY (7 DOWNTO 0) OF qit;
    - TYPE qit_octal IS ARRAY (7 DOWNTO 0) OF qit;
    - Signal qb: qit_byte;
    - Signal qo: qit_octal;
    - qb <= qit_byte(qo); -- Explicit Type Conversion (Type Casting)
      - - of closely-related types

Custom Type Conversion

- Custom Type Conversions can be defined Using either:
  - Constant Conversion Arrays, or
  - Subprograms (Functions or Procedures)
- Type conversion Arrays or Subprograms may be placed within packages, e.g. functions already in predefined standard packages may also be used
  - e.g. the package std_logic_1164 defined within the `ieee` Library
Type Conversion Using Functions

Type MVL4 (’X’, ’0’, ’1’, ’Z’);
Function MVL4_To_Bit(B: in MVL4)
Return Bit IS
Begin
Case B is
  when ’X’ => return ’0’;
  when ’0’ => return ’0’;
  when ’1’ => return ’1’;
  when ’Z’ => return ’0’;
End Case;
End MVL4_To_Bit;

Function Bit_To_MVL4 (B: in Bit) Return MVL4 IS
Begin
Case B is
  when ’0’ => return ’0’;
  when ’1’ => return ’1’;
End Case;
End MVL4_To_Bit;

Signal B4: MVL4;
Signal B: Bit;
B <= MVL4_To_Bit(B4);
B4 <= Bit_To_MVL4 (B);

DATA FLOW MODEL

- Represents Register Transfer operations
- There is Direct Mapping between Data Flow Statements && Register Structural Model
  - Implied Module Connectivity
  - Implied Muxes & Buses
- Main Data Flow VHDL Constructs:
  - Concurrent Signal Assignment Statements
  - Block Statement
Signal Assignment ...

- Unconditional: Both Sequential & Concurrent
- Conditional: Only Concurrent; Conditions Must Be Boolean, May Overlap and Need Not Be Exhaustive
- Selected: Only Concurrent; Cases Must Not Overlap and Must Be Exhaustive

Conditional Signal Assignment

```
[ Label: ] target <= [Guarded] [Transport]
  Wave1 when Cond1 Else
  Wave2 when Cond2 Else
  ................................
  Waven-1 when Condn-1 Else
  Waven ; -- Mandatory Wave
```

... Signal Assignment

- Selected Signal Assignment

```
With Expression Select
target <= [Guarded] [Transport]
  Wave1 when Choice1,
  Wave2 when Choice2,
  ................................
  Waven-1 when Chosen-1,
  Waven when OTHERS;
```

VHDL-93: Any Wavei Can Be Replaced By the Keyword UNAFFECTED (Which Doesn’t Schedule Any Transactions on the Target Signal.)
Signal Assignment Examples

Example: A 2x4 Decoder
Signal D : Bit_Vector(1 To 4) := “0000”;
Signal S0, S1 : Bit;

Decoder: D <= “0001” after T When S1='0' and S0='0' else
“0010” after T When S1='0' else
“0100” after T When S0='0' else “1000”;

Example: 4-Phase Clock Generator
Signal Phi4 : Bit_Vector(1 To 4) := "0000”;

ClkGen: With Phi4 Select
Phi4 <= “1000” after T When “0000”,
“0100” after T When “1000”,
“0010” after T When “0100”,
“0001” after T When “0010”,
“1000” after T When “0001”,
“0000” When Others ; -- Exhaustive

Multiplexing …

- Multiplexers are used for data selection

![Multiplexer Diagram]
... Multiplexing

USE WORK.basic_utilities.ALL;
-- FROM PACKAGE USE: qit, qit_vector
ENTITY mux_8_to_1 IS
PORT (i7, i6, i5, i4, i3, i2, i1, i0 : IN qit;
    s7, s6, s5, s4, s3, s2, s1, s0 : IN qit;
    z : OUT qit);
END mux_8_to_1;

ARCHITECTURE dataflow OF mux_8_to_1 IS
SIGNAL sel_lines : qit_vector ( 7 DOWNTO 0);
BEGIN
    sel_lines <= s7&s6&s5&s4&s3&s2&s1&s0;
    WITH sel_lines SELECT
    z <= '0' AFTER 3 NS WHEN "00000000",
    i7 AFTER 3 NS WHEN "10000000" | "Z0000000",
    i6 AFTER 3 NS WHEN "01000000" | "0Z000000",
    i5 AFTER 3 NS WHEN "00100000" | "00Z00000",
    i4 AFTER 3 NS WHEN "00010000" | "000Z0000",
    i3 AFTER 3 NS WHEN "00001000" | "0000Z000",
    i2 AFTER 3 NS WHEN "00000100" | "00000Z00",
    i1 AFTER 3 NS WHEN "00000010" | "000000Z0",
    i0 AFTER 3 NS WHEN "00000001" | "0000000Z",
    'X' WHEN OTHERS;
END dataflow;

3-to-8 Decoder

USE WORK.basic_utilities.ALL;
-- FROM PACKAGE USE: qit_vector
ENTITY dcd_3_to_8 IS
PORT (adr : IN qit_vector (2 DOWNTO 0);
    so : OUT qit_vector (7 DOWNTO 0));
END dcd_3_to_8;

ARCHITECTURE dataflow OF dcd_3_to_8 IS
BEGIN
    WITH adr SELECT
    so <= "00000001" AFTER 2 NS WHEN "000",
    "00000010" AFTER 2 NS WHEN "00Z" | "011",
    "00000100" AFTER 2 NS WHEN "0Z0" | "010",
    "00001000" AFTER 2 NS WHEN "ZZ0" | "Z01" | "01Z" | "011",
    "00010000" AFTER 2 NS WHEN "100" | "Z00",
    "00100000" AFTER 2 NS WHEN "ZZZ" | "Z01" | "10Z" | "101",
    "01000000" AFTER 2 NS WHEN "ZZ0" | "Z10" | "1Z0" | "110",
    "10000000" AFTER 2 NS WHEN "ZZZ" | "ZZ1" | "Z1Z" | "111",
    "XXXXXXX" WHEN OTHERS;
END dataflow;
**Block Statement**

- Block Statement is a Concurrent VHDL Construct Which is Used Within an Architectural Body to Group (Bind) a Set of Concurrent Statements

```
Block_Label: Block [ (Guard_Condition) ] [ IS ]
    Block Header;
    Block_Declarations;
    Begin
    Concurrent_Statements;
    END Block Block_Label;
```

- A Guard Condition May be Associated with a Block Statement to Allow Enabling/Disabling of Certain Signal Assignment Statements.
- The Guard Condition Defines an Implicit Signal Called GUARD.
- In the Simplest Case, Binding (Packing !) Statements Within A Block Has No Effect On the Model.
- Blocks Can Be Nested.

---

**Block Statement Example**

Architecture DF of D_Latch is

Begin
    B : Block (Clk = '1')
    Signal I_State :Bit; Block Local Signal
    Begin
        I_State <= Guarded D ;
        Q  <= I_State after 5 ns;
        QB <= not I_State after 5 ns;
    END Block B ;
END DF ;

- If Guard Condition (Clk=’1’) is TRUE, Guarded Statements within block are Enabled (Made Active)
- Guarded Statements (e.g., I_State) execute when
  - Guard Condition Becomes True, AND
  - While Guard Condition is True, a Signal on the RHS Changes Value

- UnGuarded Signal Targets (e.g., Q, QB) are independent of the Guard Condition
Positive-Edge-Triggered DFF ...

Library IEEE;
Use IEEE.Std_Logic_1164.ALL;
Entity DFF is
  Generic(TDel: Time:= 5 NS);
  Port(D, Clk: in Std_Logic; Q, QB: out Std_Logic);
End DFF;

• We will show several dataflow architectures with and without Block statement
• Will show why some of these architectures do not work

... Positive-Edge-Triggered DFF ...

Arch 1
Architecture DF1_NO_Block of DFF is
Signal I_State: Std_Logic:= '1';
begin
  I_State <= D when (Clk='1' and Clk'Event) else I_state;
  Q <= I_state after TDel ;
  QB <= not I_state after TDel ;
End DF1_NO_Block ;

Works

Signal Evaluated 2-Times Per Clock Cycle
... Positive-Edge-Triggered DFF ...  

**Arch 2**  
Architecture DF2_NO_Block of DFF is  
Signal I_State: Std_LOGIC := '1';  
begin  
I_State <= D after TDel when (Clk='1' and (not(Clk'Stable)) else I_state;  
Q <= I_state;  
QB <= not I_state;  
End DF2_NO_Block ;  

Signal Evaluated 4-Times Per Clock Cycle

**Doesn't Work**

**Arch 3**  
Architecture DF3_NO_Block of DFF is  
Signal I_State: Std_LOGIC := '1';  
begin  
I_State <= D when (Clk='1' and (not(Clk'Stable)) else I_state;  
Q <= I_state after TDel;  
QB <= not I_state after TDel ;  
End DF3_NO_Block ;

**Works**

I_State gets the value of D after 1 delta and its value does not get overwritten
... Positive-Edge-Triggered DFF ...

Arch4

Architecture DF1_Block of DFF is
Signal I_State: Std_Logic:= '1';
begin
  D_Blk: Block(Clk='1' and Clk'Event)
    Begin
      Q <= Guarded D after Tdel;
      QB <= Guarded not D after Tdel;
    End
  End
End DF1_Block;

GUARD <= Clk='1' and Clk'Event

Signal Evaluated Continuously while Clk = '1' !!!

Positive-Edge-Triggered DFF ...

Arch5

Architecture DF2_Block of DFF is
Signal I_State: Std_Logic:= '1';
begin
  D_Blk: Block(Clk='1' and not Clk'Stable)
    Begin
      Q <= Guarded D after Tdel;
      QB <= Guarded not D after Tdel;
    End
  End
End DF2_Block;

GUARD <= Clk='1' and not Clk'Stable

Signal Evaluated Once Per Clock Cycle
(At Rising Edge of the Clock)

Works

Doesn’t Work
Use of Nested Blocks For Composite Enabling Conditions

ARCHITECTURE guarding OF DFF IS
BEGIN
  edge: BLOCK ( c = '1' AND NOT c'STABLE )
  BEGIN
    gate: BLOCK ( e = '1' AND GUARD )
    BEGIN
      q <= GUARDED d AFTER delay1;
      qb <= GUARDED NOT d AFTER delay2;
      END BLOCK gate;
    END BLOCK edge;
  END guarding;

• Inner Guard Signal <= (e = '1') AND ( c= '1' AND NOT c'STABLE)
• Can nest block statements
• Combining guard expressions must be done explicitly
• Implicit GUARD signals in each block

Data Flow Example …

Model A System with 2 8-Bit Registers R1 and R2, a 2-Bit Command signal “COM” and an external 8-Bit Input “INP”

• When Com= “00” → R1 is Loaded with External Input
• When Com= “01” → R2 is Loaded with External Input
• When Com= “10” → R1 is Loaded with R1+R2
• When Com= “11” → R1 is Loaded with R1-R2

Use Work.Utils_Pkg.ALL
Entity DF_Ex is
  Port (Clk: IN Bit; Com: IN Bit_Vector (1 DownTo 0); INP: IN Bit_Vector(7 DownTo 0));
End DF_Ex;
… Data Flow Example …

Architecture DF of DF_Ex is
Signal  Mux_R1, R1, R2, R2C, R2TC, Mux_Add,
       Sum: Bit_Vector(7 DownTo 0);
Signal  D00, D01, D10, D11, LD_R1: Bit;
Begin
  D00 <= not Com(0) and not Com(1); -- Decoder
  D01 <= not Com(0) and Com(1); -- Decoder
  D10 <= Com(0) and not Com(1); -- Decoder
  D11 <= Com(0) and Com(1); -- Decoder
  R2C  <= not R2;
  R2TC <= INC(R2C); -- Increment Function Defined in the Package
  Mux_Add <=R2TC when D11 = ‘1’ Else  R2 ;
Rising Edge:
End DF;

… Data Flow Example

Sum <= ADD(R1, Mux_Add); -- ADD Function-- Defined in Package
Mux_R1 <= INP when D00 = ‘1’ Else  Sum;
R1E  <= D00 OR D10 OR D11;
Rising Edge:  BLOCK(Clk='1' and not Clk'Stable)
  R1_Reg:  BLOCK(R1E='1' AND GUARD)
    R1 <= Guarded Mux_R1 ;
  End Block R1_Reg ;
  R2_Reg:  BLOCK(D01='1' AND GUARD)
    R2 <= Guarded INP ;
  End Block R2_Reg ;
End Block Rising Edge;
End DF;
**Concurrent Versus Sequential Statements**

**Sequential Statements**
- Used Within Process Bodies or SubPrograms
- Order Dependent
- Executed When Control is Transferred to the Sequential Body
  - Assert
  - Signal Assignment
  - Procedure Call
  - Variable Assignment
  - IF Statements
  - Case Statement
  - Loops
  - Wait, Null, Next, Exit, Return

**Concurrent Statements**
- Used Within Architectural Bodies or Blocks
- Order Independent
- Executed Once At the Beginning of Simulation or Upon Some Triggered Event
  - Assert
  - Signal Assignment
  - Procedure Call (None of Formal Parameters May be of Type Variable)
  - Process
  - Block Statement
  - Component Statement
  - Generate Statement
  - Instantiation Statement

---

**Process Statement …**

- **Main Construct for Behavioral Modeling.**
- **Other Concurrent Statements Can Be Modeled By an Equivalent Process.**
- **Process Statement is a Concurrent Construct which Performs a Set of Consecutive (Sequential) Actions once it is Activated. Thus, Only Sequential Statements Are Allowed within the Process Body.**

```
PROCESS (Sensitivity_List)
Process_Declarations;
Begin
  Sequential Statements;
END Process;
```

Optional

Constant/Variables No Signal Declarations Allowed
… Process Statement …

- Unless sequential part is suspended
  - It executes in zero real and delta time
  - It repeats itself forever

![Diagram showing process flow]

BEGIN
sequential part
END PROCESS

… Process Statement

- Whenever a SIGNAL in the Sensitivity_List of the Process Changes, The Process is Activated.
- After Executing the Last Statement, the Process is SUSPENDED Until one (or more) Signal in the Process Sensitivity_List Changes Value where it will be REACTIVATED.
- A Process Statement Without a Sensitivity_List is ALWAYS ACTIVE, i.e. After the Last Statement is Executed, Execution returns to the First Statement and Continues (Infinite Looping).
- It is ILLEGAL to Use WAIT-Statement Inside a Process Which Has a Sensitivity_List.
- In case no Sensitivity_List exists, a Process may be activated or suspended Using the WAIT-Statement
- Conditional and selective signal assignments are strictly concurrent and cannot be used in a process
Process Examples

**Sequential Processing:**
- First A is Scheduled to Have a Value '1'
- Second B is Scheduled to Have a Value '0'
- A & B Get their New Values At the SAME TIME (1 Delta Time Later)

**Sequential Processing Example:**
```
Process
Begin
    A<= '1';
    B <= '0';
End Process;
```

**Assuming a '0' Initial Value of A,**
- First A is Scheduled to Have a Value '1'
  One Delta Time Later
- Thus, Upon Execution of IF_Statement, A Has a Value of '0' and Action 2 will be Taken.
- If A was Declared as a Process Variable, Action 1 Would Have Been Taken

Wait Statement

**Syntax of Wait Statement:**
- **WAIT;**
- **WAIT ON Signal List;**
- **WAIT UNTIL Condition;**
- **WAIT FOR Time_Out_Expression;**
- **WAIT FOR 0 any_time_unit;** -- Process Suspended for 1 delta

**When a WAIT-Statement is Executed, The process Suspends and Conditions for its Reactivation Are Set.**

**Process Reactivation conditions may be Mixed as follows**
- **WAIT ON Signal_List UNTIL Condition FOR Time_Expression ;**
- **wait on X,Y until (Z = 0) for 70 NS;** -- Process Resumes After 70 NS OR (in Case X or Y Changes Value and Z=0 is True) **Whichever Occurs First**
- Process Reactivated IF:
  - Event Occurred on the Signal_List while the Condition is True, OR
  - Wait Period Exceeds 'Time_Expression'
Positive Edge-Triggered D-FF Examples

\[
\text{D_FF: PROCESS (CLK)} \\
\text{Begin} \\
\text{IF (CLK'Event and CLK = '1') Then} \\
\text{Q <= D After TDelay;} \\
\text{END IF;} \\
\text{END Process;} \\
\]

\[
\text{D_FF: PROCESS -- No Sensitivity List} \\
\text{Begin} \\
\text{WAIT UNTIL CLK = '1';} \\
\text{Q <= D After TDelay;} \\
\text{END Process;} \\
\]

\[
\text{D_FF: PROCESS (Clk, Clr) -- FF With Asynchronous Clear} \\
\text{Begin} \\
\text{IF Clr = '1' Then Q <= '0' After TD0;} \\
\text{ELSIF (CLK'Event and CLK = '1') Then Q <= D After TD1;} \\
\text{END IF;} \\
\text{END Process;} \\
\]

Sequential Statements

- **CONTROL STATEMENTS**
  - **Conditional**
    - IF statements
    - CASE statement
  - **Iterative**
    - Simple Loop
    - For Loop
    - While Loop
Conditional Control – IF Statement

- **Syntax:** 3-Possible Forms
  1. IF condition Then
     statements;
     End IF;
  2. IF condition Then
     statements;
     Else
     statements;
     End IF;
  3. IF condition Then
     statements;
     Elsif condition Then
     statements;
     ........
     Elsif condition Then
     statements;
     End IF;

Conditional Control – Case Statement

- **Syntax:**
  1. CASE Expression is
     when value => statements;
     when value1 | value2| ...|valuen => statements;
     when discrete range of values => statements;
     when others => statements;
     End CASE;

  - Values/Choices should not overlap (Any value of the Expression should Evaluate to only one Arm of the Case statement).
  - All possible choices for the Expression should be accounted for Exactly Once.
Conditional Control – Case Statement

- If `others` is used, it must be the last `arm` of the CASE statement.
- There can be any number of arms in any order (except for the `others` arm which should be last)

```
CASE x is
  when 1 => y := 0;
  when 2 | 3 => y := 1;
  when 4 to 7 => y := 2;
  when others => y := 3;
End CASE;
```

Loop Control ...

- Simple Loops
- Syntax:
  
  `[Loop_Label:] LOOP
  statements;
  End LOOP [Loop_Label];

- The `Loop_Label` is Optional
- The exit statement may be used to exit the Loop. It has two possible forms:
  - exit [Loop_Label]; -- This may be used in an if statement
  - exit [Loop_Label] when condition;
...Loop Control

Process

variable A : Integer := 0;
variable B : Integer := 1;

Begin

Loop1: LOOP
    A := A + 1;
    B := 20;
Loop2: LOOP
    IF B < (A * A) Then
        exit Loop2;
    End IF;
    B := B - A;
End LOOP Loop2;
exit Loop1 when A > 10;
End LOOP Loop1;

End Process;

FOR Loop

- Syntax:

[Loop_Label]: FOR Loop_Variable in range LOOP statements;
End LOOP Loop_Label;

Process

variable B : Integer := 1;

Begin

Loop1: FOR A in 1 TO 10 LOOP
    B := 20;
    Loop2: LOOP
        IF B < (A * A) Then
            exit Loop2;
        End IF;
        B := B - A;
    End LOOP Loop2;
End LOOP Loop1;

End Process;
WHILE Loop

- **Syntax:**

  
  \[
  [\text{Loop\_Label}]: \text{WHILE condition LOOP statements; End LOOP Loop\_Label;}
  \]

  
  Process variable B:Integer :=1;
  Begin
    Loop1: FOR A in 1 TO 10 LOOP
      B := 20;
      Loop2: WHILE B < (A * A) LOOP
        B := B - A;
      End LOOP Loop2;
    End LOOP Loop1;
  End Process;

A Moore 1011 Detector using Wait

**ENTITY** moore_detector **IS**

  \[
  \text{PORT (x, clk : IN BIT; z : OUT BIT); END moore_detector;}
  \]

  
  *Can use WAIT in a Process statement to check for events on clk*

**ARCHITECTURE** behavioral_state_machine **OF** moore_detector **IS**

  \[
  \text{TYPE state IS (reset, got1, got10, got101, got1011); SIGNAL current : state := reset; BEGIN}
  \]

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
A Moore 1011 Detector using Wait

PROCESS
BEGIN
CASE current IS
  WHEN reset => WAIT UNTIL clk = '1';
    IF x = '1' THEN current <= got1; ELSE current <= reset; END IF;
  WHEN got1 => WAIT UNTIL clk = '1';
    IF x = '0' THEN current <= got10; ELSE current <= got1; END IF;
  WHEN got10 => WAIT UNTIL clk = '1';
    IF x = '1' THEN current <= got101; ELSE current <= reset; END IF;
  WHEN got101 => WAIT UNTIL clk = '1';
    IF x = '1' THEN current <= got1011; ELSE current <= got10; END IF;
  WHEN got1011 =>
    IF x = '1' THEN current <= got1; ELSE current <= got10; END IF;
END CASE;
WAIT FOR 1 NS; z <= '0';
END PROCESS;
END behavioral_state_machine;

A Moore 1011 Detector without Wait

ARCHITECTURE most_behavioral_state_machine OF moore_detector IS
  TYPE state IS (reset, got1, got10, got101, got1011);
  SIGNAL current : state := reset;
BEGIN
PROCESS (clk)
BEGIN
  IF (clk = '1' and CLK'Event) THEN
    CASE current IS
      WHEN reset =>
        IF x = '1' THEN current <= got1; ELSE current <= reset; END IF;
      WHEN got1 =>
        IF x = '0' THEN current <= got10; ELSE current <= got1; END IF;
      WHEN got10 =>
        IF x = '1' THEN current <= got101; ELSE current <= reset; END IF;
      WHEN got101 =>
        IF x = '1' THEN current <= got1011; ELSE current <= got10; END IF;
      WHEN got1011 =>
        IF x = '1' THEN current <= got1; ELSE current <= got10; END IF;
      END CASE;
      WAIT FOR 1 NS; z <= '0';
  END IF;
END PROCESS;
END most_behavioral_state_machine;
Generalized VHDL Mealy Model

Architecture Mealy of fsm is
  Signal D, Y: Std.Logic_Vector(...); -- Local Signals
Begin
Register: Process( Clk)
  Begin
   IF (Clk'EVENT and Clk = '1') Then Y <= D;
   End IF;
  End Process;
Transitions: Process(X, Y)
  Begin
   D <= F1(X, Y);
   End Process;
Output: Process(X, Y)
  Begin
   Z <= F2(X, Y);
   End Process;
End Mealy;

---

Generalized VHDL Moore Model

Architecture Moore of fsm is
  Signal D, Y: Std.Logic_Vector(...); -- Local Signals
Begin
Register: Process( Clk)
  Begin
   IF (Clk'EVENT and Clk = '1') Then Y <= D;
   End IF;
  End Process;
Transitions: Process(X, Y)
  Begin
   D <= F1(X, Y);
   End Process;
Output: Process(Y)
  Begin
   Z <= F2(Y);
   End Process;
End Moore;
FSM Example ...

Entity fsm is
port ( Clk, Reset : in Std_Logic;
    X : in Std_Logic_Vector(0 to 1);
    Z : out Std_Logic_Vector(1 downto 0));
End fsm;

Architecture behavior of fsm is
Type States is (st0, st1, st2, st3);
    Signal Present_State, Next_State : States;
Begin
    register: Process(Reset, Clk)
    Begin
        IF Reset = '1' Then
            Present_State <= st0; -- Machine Reset to st0
        elsif (Clk EVENT and Clk = '1') Then
            Present_State <= Next_state;
        End IF;
    End Process;
End behavior;

Transitions: Process(Present_State, X)
Begin
    CASE Present_State is
        when st0 =>
            Z <= '00';
            IF X = '11' Then Next_State <= st0;
            else Next_State <= st1; End IF;
        when st1 =>
            Z <= '01';
            IF X = '11' Then Next_State <= st0;
            else Next_State <= st2; End IF;
        when st2 =>
            Z <= '10';
            IF X = '11' Then Next_State <= st2;
            else Next_State <= st3; End IF;
        when st3 =>
            Z <= '11';
            IF X = '11' Then Next_State <= st3;
            else Next_State <= st0; End IF;
    End CASE;
End Process;
End behavior;
Using Wait for Two-Phase Clocking

```vhdl
BEGIN
    phase2: PROCESS
    BEGIN
        WAIT UNTIL c1 = '0';
        WAIT FOR 10 NS;
        c2 <= '1';
        WAIT FOR 480 NS;
        c2 <= '0';
    END PROCESS phase2;
```

File Type & External File I/O ...

- Specifying files is a two step process of
  - File type declaration
  - File declaration

- Data is associated with an identifier that is defined as a file type.

- File Type Declaration Example
  - Type logic_data is FILE of Character;

- File Declaration Examples
  - FILE file1 : logic_data; -- file must be opened to be associated with a physical file
  - FILE file2 : logic_data IS “Input.dat”;
  - FILE file3 : logic_data OPEN READ_MODE IS “Input.dat”;
    - File can be opened in READ_MODE, WRITE_MODE, APPEND_MODE
• FILE file4 : logic_data OPEN WRITE_MODE IS "output.dat";

■ Opening and Closing Files
• FILE_OPEN(file1, "input.dat", READ_MODE);
• FILE_OPEN(file1, "output.dat", WRITE_MODE);
• FILE_CLOSE(file1);

■ File Read and Write Operations
• VHDL provides three operations READ, WRITE and ENDFILE for file types
• READ takes a file name and an object of file data type as its argument, reads next data from file to its data argument
• WRITE takes a file name and an object of file data type as its argument, write next data from its data argument to file
• ENDFILE takes a file name and returns TRUE if a subsequent read cannot be done
.. File Type & External File I/O

-- File Type logic_data is Visible
PROCEDURE assign_bits (  
  SIGNAL target : OUT BIT; file_name : IN STRING; period : IN TIME)  IS  
  VARIABLE char : CHARACTER;  
  VARIABLE current : TIME := 0 NS;  
  FILE input_value_file : logic_data IS  file_name;  
BEGIN  
  WHILE NOT ENDFILE (input_value_file) LOOP  
    READ (input_value_file, char);  
    IF char = '0' OR char = '1' THEN  
      current := current + period;  
      IF char = '0' THEN  
        target <= TRANSPORT '0' AFTER current;  
      ELSIF char = '1' THEN  
        target <= TRANSPORT '1' AFTER current;  
      END IF;  
    END IF;  
  END LOOP;  
END assign_bits;

Formatted I/O ...

- USE STD.TEXTIO.ALL;
- I is LINE, f is FILE
- The following functions provided:
  - READLINE (f, I)
  - READ (I, v)
  - WRITE (I, v),
  - WRITELINE (f, I)
  - ENDFILE (f)
- READ or WRITE can read values of type:
  - BIT, BIT_VECTOR, BOOLEAN, CHARACTER, INTEGER, REAL, STRING, TIME
TYPE state IS (reset, got1, got10, got101);
TYPE state_vector IS ARRAY (NATURAL RANGE <>) OF state;
FUNCTION one_of (sources : state_vector) RETURN state IS
USE STD.TEXTIO.ALL;
VARIABLE l : LINE;
FILE flush : TEXT IS OUT "/dev/tty";
BEGIN
FOR i IN sources'RANGE LOOP
WRITE (l, state'IMAGE(sources(I), LEFT, 7);
END LOOP;
WRITELINE (flush, l);
RETURN sources (sources'LEFT);
END one_of;

• Add screen output to resolution function
• The 'IMAGE type attribute translates a state to its corresponding string
• The keyword LEFT specifies left justification
• 7 specifies the string length

USE STD.TEXTIO.ALL;
PROCEDURE display (SIGNAL value1, value2 : BIT) IS
FILE flush : TEXT IS OUT "/dev/tty;
VARIABLE filler : STRING (1 TO 3) := " ..";
VARIABLE l : LINE;
BEGIN
WRITE (l, NOW, RIGHT, 8, NS);
IF value1'EVENT THEN
WRITE (l, value1, RIGHT, 3);
WRITE (l, filler, LEFT, 0);
ELSE
WRITE (l, filler, LEFT, 0);
WRITE (l, value2, RIGHT, 3);
END IF;
WRITE (l, filler, LEFT, 0);
WRITE (l, value2, RIGHT, 3);
END display;

• An EVENT on value1 or value2 puts the following in l:
  NOW
• An EVENT on value1 puts the following in l:
  v1 ...
• An EVENT on value2 puts the following in l:
  ... v2
• WRITELINE writes:
  time v1 ...
  time ... v2
VHDL Synthesis Subset

- VHDL is a complex language but only a subset of it is synthesizable.
- Primary VDHL constructs used for synthesis:
  - Constant definition
  - Port map statement
  - Signal assignment: A <= B
  - Comparisons: = (equal), /= (not equal), > (greater than), < (less than), >= (greater than or equal), <= (less than or equal)
  - Logical operators: AND, OR, NAND, NOR, XOR, XNOR, NOT
  - 'if' statement
    - if (presentstate = CHECK_CAR) then ....
    - end if | elsif ....
  - 'for' statement (used for looping in creating arrays of elements)
  - Other constructs are 'with', 'when', 'when else', 'case', 'wait'. Also ":=" for variable assignment.

Constant Definition...

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity constant_ex is
  port (in1 : in std_logic_vector (7 downto 0); out1 : out std_logic_vector (7 downto 0));
end constant_ex;
architecture constant_ex_a of constant_ex is
  constant A : std_logic_vector (7 downto 0) := "00000000";
  constant B : std_logic_vector (7 downto 0) := "11111111";
  constant C : std_logic_vector (7 downto 0) := "00001111";
begin
  out1 <= A when in1 = B else C;
end constant_ex_a;
```
library ieee;
use ieee.std_logic_1164.all;
entity sub is
  port (a, b : in std_logic; c : out std_logic);
end sub;
architecture sub_a of sub is
begin
  c <= a and b;
end sub_a;
library ieee;
use ieee.std_logic_1164.all;
entity portmap_ex is
  port (in1, in2, in3 : in std_logic; out1 : out std_logic);
end portmap_ex;
architecture portmap_ex_a of portmap_ex is
  component sub
    port (a, b : in std_logic; c : out std_logic);
  end component;
  signal temp : std_logic;
begin
  u0 : sub port map (in1, in2, temp);
  u1 : sub port map (temp, in3, out1);
end portmap_ex_a;
When Statement

library ieee;
use ieee.std_logic_1164.all;
entity when_ex is
  port (in1, in2 : in std_logic; out1 : out std_logic);
end when_ex;
architecture when_ex_a of when_ex is
begin
  out1 <= '1' when in1 = '1' and in2 = '1' else '0';
end when_ex_a;

With Statement

library ieee;
use ieee.std_logic_1164.all;
entity with_ex is
  port (in1, in2 : in std_logic; out1 : out std_logic);
end with_ex;
architecture with_ex_a of with_ex is
begin
  with in1 select out1 <= in2 when '1',
      '0' when others;
end with_ex_a;
Case Statement...

library ieee;
use ieee.std_logic_1164.all;
entity case_ex is
  port (in1, in2 : in std_logic; out1, out2 : out std_logic);
end case_ex;
architecture case_ex_a of case_ex is
  signal b : std_logic_vector (1 downto 0);
  begin
    process (b)
    begin
      case b is
      when "00" | "11" => out1 <= '0'; out2 <= '1';
      when others => out1 <= '1'; out2 <= '0';
      end case;
    end process;
    b <= in1 & in2;
  end case_ex_a;

...Case Statement
library ieee;
use ieee.std_logic_1164.all;
entity for_ex is
  port (in1 : in std_logic_vector (3 downto 0); out1 : out std_logic_vector (3 downto 0));
end for_ex;
architecture for_ex_a of for_ex is
begin
  process (in1)
  begin
    for0 : for i in 0 to 3 loop
      out1 (i) <= not in1(i);
    end loop;
  end process;
end for_ex_a;
Generate Statement

signal A,B:BIT_VECTOR (3 downto 0);
signal C:BIT_VECTOR (7 downto 0);
signal X:BIT;

... 

GEN_LABEL:
for I in 3 downto 0 generate
  C(2*I+1) <= A(I) nor X;
  C(2*I) <= B(I) nor X;
end generate GEN_LABEL.

If Statement

library ieee;
use ieee.std_logic_1164.all;
entity if_ex is
  port (in1, in2 : in std_logic; out1 : out std_logic);
end if_ex;
architecture if_ex_a of if_ex is
begin
  process (in1, in2)
  begin
    if in1 = '1' and in2 = '1' then out1 <= '1';
    else out1 <= '0';
    end if;
  end process;
end if_ex_a;
library ieee;
use ieee.std_logic_1164.all;

entity variable_ex is
  port ( a : in std_logic_vector (3 downto 0); b : in std_logic_vector (3 downto 0); c : out std_logic_vector (3 downto 0));
end entity variable_ex;

architecture variable_ex_a of variable_ex is
begin
  process (a,b)
  variable carry : std_logic_vector (4 downto 0);
  variable sum : std_logic_vector (3 downto 0);
  begin
    carry (0) := '0';
    for i in 0 to 3 loop
      sum (i) := a(i) xor b(i) xor carry(i);
      carry (i+1) := (a(i) and b(i)) or (b(i) and carry (i))
      or (carry (i) and a(i));
    end loop;
    c <= sum;
  end process;
end variable_ex_a;
library ieee;
use ieee.std_logic_1164.all;
entity mux is
    port (in1, in2, ctrl : in std_logic; out1 : out std_logic);
end mux;
architecture mux_a of mux is
begin
    process (in1, in2, ctrl)
    begin
        if ctrl = '0' then out1 <= in1;
        else out1 <= in2;
        end if;
    end process;
end mux_a;
**Multiplexor Synthesis**

entity mux2to1_8 is
  port ( signal s: in std_logic; signal zero, one: in std_logic_vector(7 downto 0); signal y: out std_logic_vector(7 downto 0));
end mux2to1_8;
architecture behavior of mux2to1 is
begin
  y <= one when (s = '1') else zero;
end behavior;

2x1 Multiplexor using Booleans

architecture boolean_mux of mux2to1_8 is
  signal temp: std_logic_vector(7 downto 0);
begin
  temp <= (others => s);
  y <= (temp and one) or (not temp and zero);
end boolean_mux;

- The s signal cannot be used in a Boolean operation with the one or zero signals because of type mismatch (s is a std_logic type, one/zero are std_logic_vector types)
- An internal signal of type std_logic_vector called temp is declared. The temp signal will be used in the Boolean operation against the zero/one signals.
- Every bit of temp is set equal to the s signal value.
2x1 Multiplexor using a Process

architecture process_mux of mux2to1_8 is
begin
    comb: process (s, zero, one)
    begin
        y <= zero;
        if (s = '1') then
            y <= one;
        end if;
    end process comb;
end process_mux;

Decoder Synthesis...

library ieee;
use ieee.std_logic_1164.all;
entity decoder is
    port (in1, in2 : in std_logic; out00, out01, out10, out11 : out std_logic);
end decoder;
architecture decoder_a of decoder is
begin
    process (in1, in2)
    begin
        if in1 = '0' and in2 = '0' then out00 <= '1';
        else out00 <= '0';
        end if;
        if in1 = '0' and in2 = '1' then out01 <= '1';
        else out01 <= '0';
        end if;
        if in1 = '1' and in2 = '0' then out10 <= '1';
        else out10 <= '0';
        end if;
        if in1 = '1' and in2 = '1' then out11 <= '1';
        else out11 <= '0';
        end if;
end process;
**Decoder Synthesis**

```vhdl
if in1 = '1' and in2 = '0' then out10 <= '1';
else out10 <= '0';
end if;
if in1 = '1' and in2 = '1' then out11 <= '1';
else out11 <= '0';
end if;
end process;
end decoder_a;
```

**3-to-8 Decoder Example...**

```vhdl
entity dec3to8 is
  port (signal sel: in std_logic_vector(2 downto 0); signal en: in std_logic; signal y: out std_logic_vector(7 downto 0));
end dec3to8;
architecture behavior of dec3to8 is
begin
  process (sel, en)
    y <= "1111111";
    if (en = '1') then
      case sel is
      when "000" => y(0) <= '0';
      when "001" => y(1) <= '0';
      when "010" => y(2) <= '0';
      when "011" => y(3) <= '0';
      when "100" => y(4) <= '0';
      when "101" => y(5) <= '0';
      when "110" => y(6) <= '0';
      when "111" => y(7) <= '0';
      end case;
    end if;
  end process;
end behavior;
```
3-to-8 Decoder Example

Architecture of Generic Decoder

```vhdl
architecture behavior of generic_decoder is
begin
    process (sel, en)
    begin
        y <= (others => '1');
        for i in y'range loop
            if (en = '1' and bvtoi(To_Bitvector(sel)) = i) then
                y(i) <= '0';
            end if;
        end loop;
    end process;
end behavior;
```

`bvtoi` is a function to convert from `bit_vector` to `integer`
A Common Error in Process Statements…

- When using processes, a common error is to forget to assign an output a default value.
  - ALL outputs should have DEFAULT values
- If there is a logical path in the model such that an output is not assigned any value
  - the synthesizer will assume that the output must retain its current value
  - a latch will be generated.
- Example: In `dec3to8.vhd` do not assign 'y' the default value of B"11111111"
  - If `en` is 0, then 'y' will not be assigned a value
  - In the new synthesized logic, all 'y' outputs are latched

...A Common Error in Process Statements...

```
entity dec3to8 is
  port (signal sel: in std_logic_vector(3 downto 0); signal en: in std_logic;
        signal y: out std_logic_vector(7 downto 0));
end dec3to8;
architecture behavior of dec3to8 is
begin
  process (sel, en)
  -- y <= "11111111";
  if (en = '1') then
    case sel is
      when "000" => y(0) <= '0';
      when "001" => y(1) <= '0';
      when "010" => y(2) <= '0';
      when "011" => y(3) <= '0';
      when "100" => y(4) <= '0';
      when "101" => y(5) <= '0';
      when "110" => y(6) <= '0';
      when "111" => y(7) <= '0';
    end case;
  end if;
end process;
end behavior;
```

No default value assigned to y!!
...A Common Error in Process Statements

Another Incorrect Latch Insertion Example...

entity case_example is
    port (in1, in2 : in std_logic; out1, out2 : out std_logic);
end case_example;
architecture caseLatch of case_example is
    signal b : std_logic_vector (1 downto 0);
begins
    process (b)
    begin
        case b is
            when "01" => out1 <= '0'; out2 <= '1';
            when "10" => out1 <= '1'; out2 <= '0';
            when others => out1 <= '1';
        end case;
    end process;
b <= in1 & in2;
end caseLatch;

out2 has not been assigned a value for others condition!!
Another Incorrect Latch Insertion

Example

architecture case_nolatch of case_example is
    signal b : std_logic_vector (1 downto 0);
begin
    process (b)
    begin
        case b is
            when "01" => out1 <= '0'; out2 <= '1';
            when "10" => out1 <= '1'; out2 <= '0';
            when others => out1 <= '1'; out2 <= '0';
        end case;
    end process;
    b <= in1 & in2;
end case_nolatch;

Avoiding Incorrect Latch Insertion

architecture case_nolatch of case_example is
    signal b : std_logic_vector (1 downto 0);
begin
    process (b)
    begin
        case b is
            when "01" => out1 <= '0'; out2 <= '1';
            when "10" => out1 <= '1'; out2 <= '0';
            when others => out1 <= '1'; out2 <= '0';
        end case;
    end process;
    b <= in1 & in2;
end case_nolatch;
Eight-Level Priority Encoder…

Entity priority is
  Port (Signal y1, y2, y3, y4, y5, y6, y7: in std_logic;
       Signal vec: out std_logic_vector(2 downto 0));
End priority;

Architecture behavior of priority is
Begin
  Process(y1, y2, y3, y4, y5, y6, y7)
  begin
    if (y7 = '1') then vec <= "111"; elsif (y6 = '1') then vec <= "110";
    elsif (y5 = '1') then vec <= "101"; elsif (y4 = '1') then vec <= "100";
    elsif (y3 = '1') then vec <= "011"; elsif (y2 = '1') then vec <= "010";
    elsif (y1 = '1') then vec <= "001"; else vec <= "000";
  end if;
  end process;
End behavior;

...Eight-Level Priority Encoder...
Eight-Level Priority Encoder…

Architecture behavior of priority is
Begin
  Process(y1, y2, y3, y4, y5, y6, y7)
  begin
    vec <= “000”;
    if (y1 = ‘1’) then vec <= “001”; end if;
    if (y2 = ‘1’) then vec <= “010”; end if;
    if (y3 = ‘1’) then vec <= “011”; end if;
    if (y4 = ‘1’) then vec <= “100”; end if;
    if (y5 = ‘1’) then vec <= “101”; end if;
    if (y6 = ‘1’) then vec <= “110”; end if;
    if (y7 = ‘1’) then vec <= “111”; end if;
  end process;
End behavior;

Equivalent 8-level priority encoder.

Ripple Carry Adder…

library ieee;
use ieee.std_logic_1164.all;
entity adder4 is
  port (Signal a, b: in std_logic_vector (3 downto 0);
         Signal cin : in std_logic_vector;
         Signal sum: out std_logic_vector (3 downto 0);
         Signal cout : in std_logic_vector);
end adder4;
architecture behavior of adder4 is
  Signal c: std_logic_vector (4 downto 0);
begi
  C is a temporary signal to hold the carries.
process (a, b, cin, c)
begin
  \( c(0) \leftarrow cin; \)
  for \( l \) in 0 to 3 loop
    \( sum(l) \leftarrow a(l) \oplus b(l) \oplus c(l); \)
    \( c(l+1) \leftarrow (a(l) \land b(l)) \lor (c(l) \land (a(l) \lor b(l))); \)
  end loop;
end process;

\( cout \leftarrow c(4); \)
End behavior;

- The Standard Logic 1164 package does not define arithmetic operators for the std_logic type.
- Most vendors supply some sort of arithmetic package for 1164 data types.
- Some vendors also support synthesis using the '+' operation between two std_logic signal types (Synopsis).
Tri-State Buffer Synthesis

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity tri_ex is
  port (in1, control : in std_logic; out1 : out std_logic);
end tri_ex;
architecture tri_ex_a of tri_ex is
begin
  out1 <= in1 when control = '1' else 'Z';
end tri_ex_a;
```

Bi-directional Buffer Synthesis

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity inout_ex is
  port (io1, io2 : inout std_logic; ctrl : in std_logic);
end inout_ex;
architecture inout_ex_a of inout_ex is
begin
  io1 <= io2 when ctrl = '1' else 'Z';
io2 <= io1 when ctrl = '0' else 'Z';
end inout_ex_a;
```
Sequential Circuits

- Sequential circuits consist of both combinational logic and storage elements.
- Sequential circuits can be
  - Moore-type: outputs are a combinatorial function of Present State signals.
  - Mealy-type: outputs are a combinatorial function of both Present State signals and primary inputs.

Template Model for a Sequential Circuit

```vhdl
entity model_name is
  port ( list of inputs and outputs );
end model_name;
architecture behavior of model_name is
  internal signal declarations
  begin
    -- the state process defines the storage elements
    state: process ( sensitivity list – clock, reset, next_state inputs)
    begin
      vhdl statements for state elements
    end process state;
    -- the comb process defines the combinational logic
    comb: process ( sensitivity list – usually includes all inputs)
    begin
      vhdl statements which specify combinational logic
    end process comb;
  end behavior;
```
Latch Synthesis…

library ieee;
use ieee.std_logic_1164.all;
entity latch_ex is
  port (clock, in1 : in std_logic; out1 : out std_logic);
end latch_ex;
architecture latch_ex_a of latch_ex is
begin
  process (clock, in1)
  begin
    if (clock = '1') then
      out1 <= in1;
    end if;
  end process;
end latch_ex_a;

…Latch Synthesis
library ieee;
use ieee.std_logic_1164.all;
entity dff_asyn is
  port( reset, clock, d: in std_logic; q: out std_logic);
end dff_asyn;
architecture dff_asyn_a of dff_asyn is
begin
  process
  begin
    if (reset = '1') then
      q <= '0';
    elsif clock = '1' and clock'event then
      q <= d;
    end if;
  end process;
end dff_asyn_a;

• Note that the reset input has precedence over the clock in order to define the asynchronous operation.
Flip-Flop Synthesis with Synchronous Reset...

library ieee;
use ieee.std_logic_1164.all;
entity dff_syn is
  port( reset, clock, d: in std_logic; q: out std_logic);
end dff_syn;
architecture dff_syn_a of dff_syn is
begin
  process
  begin
    if clock = '1' and clock'event then
      if (reset = '1') then q <= '0';
      else q <= d;
      end if;
    end if;
  end process;
end dff_syn_a;

...Flip-Flop Synthesis with Synchronous Reset
8-bit Loadable Register with Asynchronous Clear...

library ieee;
use ieee.std_logic_1164.all;
entity reg8bit is
  port( reset, clock, load: in std_logic;
     din: in std_logic_vector(7 downto 0);
     dout: out std_logic_vector(7 downto 0));
end reg8bit;
architecture behavior of reg8bit is
  signal n_state, p_state: std_logic_vector(7 downto 0);
begnin
  dout <= p_state;
  comb: process (p_state, load, din)
  begin
    n_state <= p_state;
    if (load = '1') then n_state <= din end if;
  end process comb;
  state: process (clk, reset)
  begin
    if (reset = '0') then p_state <= (others => '0');
    elsif (clock = '1' and clock'event) then
      p_state <= n_state;
    end if;
  end process state;
end behavior;

...8-bit Loadable Register with Asynchronous Clear...

state: process (clk, reset)
begin
  if (reset = '0') then p_state <= (others => '0');
  elsif (clock = '1' and clock'event) then
    p_state <= n_state;
  end if;
end process state;

• The state process defines a storage element which is 8-bits wide, rising edge triggered, and had a low true asynchronous reset.
• Note that the reset input has precedence over the clock in order to define the asynchronous operation.
...8-bit Loadable Register with Asynchronous Clear

library ieee;
use ieee.std_logic_1164.all;
entity shift4 is
  port( reset, clock: in std_logic; din: in std_logic;
       dout: out std_logic_vector(3 downto 0));
end shift4;
architecture behavior of shift4 is
  signal n_state, p_state: std_logic_vector(3 downto 0);
begin
  dout <= p_state;
  state: process (clk, reset)
  begin
    if (reset = '0') then   p_state <= (others => '0');
    elsif (clock = '1' and clock'event) then
      p_stateq <= n_state;
    end if;
  end process state;
end behavior;
...4-bit Shift Register...

comb: process (p_state, din)
begin
  n_state(0) <= din;
  for i in 3 downto 0 loop
    n_state(i) <= p_state(i-1);
  end loop;
end process comb;

End behavior;

- Serial input din is assigned to the D-input of the first D-FF.
- For loop is used to connect the output of previous flip-flop to the input of current flip-flop.

...4-bit Shift Register
Register with Tri-State Output...

library ieee;
use ieee.std_logic_1164.all;
entity tsreg8bit is
  port( reset, clock, load, en: in std_logic;
       signal din: in std_logic_vector(7 downto 0);
       signal dout: out std_logic_vector(7 downto 0));
end tsreg8bit;
architecture behavior of tsreg8bit is
  signal n_state, p_state: std_logic_vector(7 downto 0);
begin
  dout <= p_state when (en='1') else "ZZZZZZZZ";
  comb: process (p_state, load, din)
  begin
    n_state <= p_state;
    if (load = '1') then n_state <= din end if;
  end process comb;
end behavior;

...Register with Tri-State Output...

state: process (clk, reset)
begin
  if (reset = '0') then p_state <= (others => '0');
  elsif (clock = '1' and clock'event) then
    p_state <= n_state;
  end if;
end process state;
End behavior;
...Register with Tri-State Output

Finite State Machine Synthesis...

- Mealy model
- Single input, two outputs
- Synchronous reset

Reset=0
library ieee;
use ieee.std_logic_1164.all;
entity state_ex is
    port (in1, clock, reset : in std_logic; out1 :
        out std_logic_vector (1 downto 0));
end state_ex;
architecture state_ex_a of state_ex is
    signal cur_state, next_state : std_logic_vector (1 downto 0);
begin
    process (clock, reset)
        begin
            if clock = '1' and clock'event then
                if reset = '0' then cur_state <= "00";
                else cur_state <= next_state;
                end if;
            end if;
        end process;
    process (in1, cur_state)
        begin
            case cur_state is
                when "00" => if in1 = '0' then next_state <= "10"; out1 <= "00";
                else next_state <= "01"; out1 <= "10";
                end if;
                when "01" => if in1 = '0' then next_state <= cur_state;
                            out1 <= "01";
                else next_state <= "10"; out1 <= "10";
                end if;
                when "10" => next_state <= "11"; out1 <= "10";
                when "11" => next_state <= "00"; out1 <= "10";
                when others => null;
            end case;
        end process;
    end state_ex_a;
Key Synthesis Facts

- Synthesis ignores the after clause in signal assignment
  - C <= A AND B after 10ns
  - May cause mismatch between pre-synthesis and post-synthesis simulation if a non-zero value used
  - The preferred coding style is to write signal assignments without the after clause.

- If the process has a static sensitivity list, it is ignored by the synthesis tool.

- Sensitivity list must contain all read signals
  - Synthesis tool will generate a warning if this condition is not satisfied
  - Results in mismatch between pre-synthesis and post-synthesis simulation
Synthesis Static Sensitivity Rule

<table>
<thead>
<tr>
<th>Original VHDL Code</th>
<th>Pre-Synthesis Simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process(A, B)</td>
<td></td>
</tr>
<tr>
<td>Begin</td>
<td></td>
</tr>
<tr>
<td>D &lt;= (A AND B) OR C;</td>
<td></td>
</tr>
<tr>
<td>End process;</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Synthesis View of Original VHDL Code</th>
<th>Post-Synthesis Simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process(A, B, C)</td>
<td></td>
</tr>
<tr>
<td>Begin</td>
<td></td>
</tr>
<tr>
<td>D &lt;= (A AND B) OR C;</td>
<td></td>
</tr>
<tr>
<td>End process;</td>
<td></td>
</tr>
</tbody>
</table>

Latch Inference & Synthesis Rules...

- A latch is inferred to satisfy the VHDL fact that signals and process declared variables maintain their values until assigned new ones.
- Latches are synthesized from if statements if all the following conditions are satisfied
  - Conditional expressions are not completely specified
  - An else clause is omitted
  - Objects conditionally assigned in an if statement are not assigned a value before entering this if statement
  - The VHDL attribute `EVENT is not present in the conditional if expression
- If latches are not desired, then a value must be assigned to the target object under all conditions of an if statement (without the `EVENT attribute).
...Latch Inference & Synthesis Rules

- **For a case statement**, latches are synthesized when it satisfies all of the following conditions:
  - An expression is not assigned to a VHDL object in every branch of a case statement
  - VHDL objects assigned an expression in any case branch are not assigned a value before the case statement is entered.

- **Latches are synthesized whenever a for...loop statement** satisfies all of the following conditions
  - for...loop contains a next statement
  - Objects assigned inside the for...loop are not assigned a value before entering the enclosing for...loop

---

**For...Loop Statement Latch Example**

```
Process(Data_In, Copy_Enable)
Begin
  for k in 7 downto 0 loop
    next when Copy_Enable(k)='0'
    Data_Out(k) <= Data_in(k);
  end loop;
End process;
```

**Seven latches will be synthesized**

```
Data_In(k)   Data_Out(k)

Copy_Enable(k)
```

LATCH
Flip-Flop Inference & Synthesis Rules...

- Flip-flops are inferred by either
  - Wait until….
    - Wait on… is not supported by synthesis
    - Wait for… is not supported by synthesis
  - If statement containing `EVENT

- Synthesis accepts any of the following functionally equivalent statements for inferring a FF
  - Wait until Clock='1';
  - Wait until Clock’Event and Clock='1';
  - Wait until (not Clock’Stable) and Clock='1';

...Flip-Flop Inference & Synthesis Rules

- Synthesis does not support the following Asynchronous description of set and reset signals
  - Wait until (clock='1') or (Reset='1')
  - Wait on Clock, Reset

- When using a synthesizable wait statement only synchronous set and reset can be used.

- If statement containing the VHDL attribute `EVENT cannot have an else or an elsif clause.