Motivation for VHDL

- Technology trends
  - 1 billion transistor chip running at 20 GHz in 2007
- Need for Hardware Description Languages
  - Systems become more complex
  - Design at the gate and flip-flop level becomes very tedious and time consuming
- HDLs allow
  - Design and debugging at a higher level before conversion to the gate and flip-flop level
  - Tools for synthesis do the conversion
- VHDL, Verilog
- VHDL — VHSIC Hardware Description Language
Facts About VHDL

- Developed originally by DARPA
  — for specifying digital systems
- International IEEE standard (IEEE 1076-1993)
- Hardware Description, Simulation, Synthesis
- Provides a mechanism for digital design and reusable design documentation
- Support different description levels
  — Structural (specifying interconnections of the gates),
  — Dataflow (specifying logic equations), and
  — Behavioral (specifying behavior)
- Top-down, Technology Independent

VHDL Description of Combinational Networks

Concurrent Statements
- C <= A and B after 5 ns;
- E <= C or D after 5 ns;
If delay is not specified, “delta” delay is assumed
- C <= A and B;
- E <= C or D;
Order of concurrent statements is not important
- E <= C or D;
- C <= A and B;
This statement executes repeatedly
- CLK <= not CLK after 10 ns;
This statement causes a simulation error
- CLK <= not CLK;
**Entity-Architecture Pair**

**Full Adder Example**

```vhdl
entity FullAdder is
  port (X, Y, Cin: in bit;  -- Inputs
       Cout, Sum: out bit);  -- Outputs
end FullAdder;

architecture Equations of FullAdder is
begin  -- Concurrent Assignments
  Sum  <= X xor Y xor Cin after 10 ns;
  Cout <= (X and Y) or (X and Cin) or (Y and Cin) after 10 ns;
end Equations;
```

---

**Hierarchy of VHDL Models**

```vhdl
entity entity-name is
  [port(interface-signal-declaration);]
end [entity] [entity-name];

architecture architecture-name of entity-name is
  [declarations]
begin
  architecture body
end [architecture] [architecture-name];
```
**4-bit Adder**

```
entity Adder4 is
  port (A, B: in bit_vector(3 downto 0); Ci: in bit;); -- Inputs
  S: out bit_vector(3 downto 0); Co: out bit); -- Outputs
end Adder4;
```

**Structural Architecture of a 4-bit Adder**

```
entity Adder4 is
  port (A, B: in bit_vector(3 downto 0); Ci: in bit;); -- Inputs
  S: out bit_vector(3 downto 0); Co: out bit); -- Outputs
end Adder4;

architecture Structure of Adder4 is
component FullAdder
  port (X, Y, Cin: in bit; Cout, Sum: out bit); -- Inputs
end component;

signal C: bit_vector(3 downto 1);
begin --Instantiate four copies of the FullAdder
  FA0: FullAdder port map (A(0), B(0), Ci, C(1), S(0));
  FA1: FullAdder port map (A(1), B(1), C(1), C(2), S(1));
  FA2: FullAdder port map (A(2), B(2), C(2), C(3), S(2));
  FA3: FullAdder port map (A(3), B(3), C(3), Co, S(3));
end Structure;
```
4-bit Adder Simulation - Providing the Stimuli using the Simulator

list A B Co C Ci S -- put these signals on the output list
force A 1111 -- set the A inputs to 1111
force B 0001 -- set the B inputs to 0001
force Ci 1 1 -- set the Ci to 1
run 50 0 50 -- run the simulation for 50 ns
force Ci 0
force A 0101
force B 1110
run 50

ns delta a b c o c c i s
0 +0 0000 0000 0 000 0 0000
0 +1 1111 0001 0 000 1 0000
10 +0 1111 0001 0 001 1 1111
20 +0 1111 0001 0 111 1 1101
30 +0 1111 0001 0 111 1 1001
40 +0 1111 0001 1 111 1 0001
50 +0 0101 1110 1 111 0 0001
60 +0 0101 1110 1 110 0 0101
70 +0 0101 1110 1 100 0 0111
80 +0 0101 1110 1 100 0 0111

4-bit Adder Simulation - Providing the Stimuli using a VHDL Testbench

entity ADDER is
end ADDER;

architecture TESTBENCH of ADDER is
component Adder4 is
port (A, B: in bit_vector(3 downto 0); Ci: in bit; --Inputs
S: out bit_vector(3 downto 0); Co: out bit); --Output
end component;
signal A, B, S : bit_vector(3 downto 0);
signal Ci, Co : bit;
begin
-- signals
A <= "1111" after 0 ns, "0101" after 50 ns;
B <= "0001" after 0 ns, "1110" after 50 ns;
Ci <= '1' after 0 ns, '0' after 50 ns;
-- component being tested
ctl: Adder4 port map (A, B, Ci, S, Co);
end TESTBENCH;
Chapter 2

Altera Simulation of the Full Adder

Altera Simulation of the 4-bit Adder
Behavioral Description of a 4-bit Adder

-- Behavioral Description of 4 Bit Adder

-- Libraries
  library IEEE;
  use IEEE.std_logic_1164.all; -- to allow STD_LOGICVectors
  use IEEE.std_logic_arith.all; -- to allow arithmetic operations
    with the vectors
  use IEEE.std_logic_unsigned.all; -- to allow integer conversions

entity ADDER4X is
  port(A,B: in STD_LOGIC_VECTOR (3 downto 0)); -- Inputs
    CI : in STD_LOGIC; -- Input
    S : out STD_LOGIC_VECTOR (3 downto 0); -- Outputs
    CO: out STD_LOGIC; -- Output
end ADDER4X;

architecture BEHAVIORAL of ADDER4X is
  -- temporary signal vectors which are 5 bits to keep up with CO
  signal AA,BB,SS : STD_LOGIC_VECTOR (4 downto 0) := "00000";

begin
  -- converting from 4 bit inputs to 5 bit ones to allow full
  -- 5 bit arithmetic
    AA <= '0'&A; -- appending a leading 0 at most significant
    BB <= '0'&B; -- bit position of A and B
  -- main equation for 4 bit adder (describes desired behavior)
    SS <= AA + BB + CI;

  -- relating this internal output to the port output signals
    S <= SS(3 downto 0); -- 4 bit Sum
    CO <= SS(4); -- carry out
end BEHAVIORAL;
Simulation of Behavioral Architecture

Whenever one of the signals in the sensitivity list changes, the sequential statements are executed in sequence one time.

General form of process

```
process(sensitivity-list)
begin
    sequential-statements
end process;
```

¥ Whenever one of the signals in the sensitivity list changes, the sequential statements are executed in sequence one time.
Chapter 2

Concurrent Statements versus Processes

A, B, C, D are integers A=1, B=2, C=3, D=0 D changes to 4 at time 10

\[
\begin{align*}
A &= B; \quad \text{-- statement 1} \\
B &= C; \quad \text{-- statement 2} \\
C &= D; \quad \text{-- statement 3}
\end{align*}
\]

\[
\text{process (B, C, D)}
\begin{align*}
\text{begin} \\
A &= B; \quad \text{-- statement 1} \\
B &= C; \quad \text{-- statement 2} \\
C &= D; \quad \text{-- statement 3}
\end{align*}
\text{end process;}
\]

Modeling a D Flip-Flop

\[
\text{entity DFF is}
\begin{align*}
& \text{port (D, CLK: in bit;}
& \quad Q: out bit; \text{ QN: out bit := '1');}
& \quad \text{-- initialize QN to '1' since bit signals are initialized to '0' by default}
\end{align*}
\text{end DFF;}
\]

\[
\text{architecture SIMPLE of DFF is}
\begin{align*}
\text{begin} \\
& \text{process (CLK)} \quad \text{-- process is executed when CLK changes}
& \text{begin} \\
& \quad \text{if CLK = '1' then} \\
& \quad \quad Q &= D \text{ after 10 ns;}
& \quad \quad QN &= \text{not D after 10 ns;}
& \quad \text{end if;}
& \text{end process;}
\end{align*}
\text{end SIMPLE;}
\]
Modeling a D Latch

entity DLATCH is
  port (D, Q, QN: in bit;
       Q: out bit; QN: out bit := '1');
end DLATCH;

architecture SIMPLE of DLATCH is
begin
  process(G, D) -- process is executed when either G or D changes
  begin
    if G = '1' then -- pass D through when G=1
      Q := D after 10 ns;
      QN := not D after 10 ns;
    end if;
  end process;
end SIMPLE;

---

D Flip-Flop versus D Latch

D Flip-Flop

D Latch
Chapter 2

Altera Simulation of D Flip-Flop

Altera’s Simulation of D Flip-Flop

Chapter 2

Altera Simulation of D Latch

Altera’s Simulation of D Latch

Electrical and Computer Engineering
Building a Shift Register with D Flip-flop Building Blocks

-- Structural Description of 4 Bit Shift Register

-- Libraries
library IEEE;
USE IEEE.std_logic_1164.all; -- to allow STD_LOGIC

entity SHIFT4S is
    port(CLK, D : in STD_LOGIC; -- Inputs
        Q : out STD_LOGIC_VECTOR (3 downto 0)); -- Outputs
end SHIFT4S;

architecture STRUCT of SHIFT4S is
    signal Q0 : STD_LOGIC_VECTOR (3 downto 0) := "0000";
    component DFF is
        port(D, CLK : in STD_LOGIC; -- Inputs
            Q : out STD_LOGIC; QN : out STD_LOGIC := '1'); -- Outputs
    end component;

    begin
        C1: DFF port map (Q0,CLK,Q(0),Q0(0));
        C2: DFF port map (Q0,CLK,Q(1),Q0(1));
        C3: DFF port map (01,Q0,Q(2),01(1));
        C4: DFF port map (Q0,CLK,Q(3),Q0(3));
    end STRUCT;
Testing the Shift Register

-- Libraries
library IEEE;
USE IEEE.std_logic_1164.all; -- to allow STD_LOGIC_VECTORS
USE IEEE.std_logic_arith.all; -- to allow arithmetic operations
USE IEEE.std_logic_unsigned.all; -- to allow integer conversions

entity SHIFT48 is
end SHIFT48;

architecture BEHAVIORAL of SHIFT48 is
begin
process (CLR)
begin
if CLR = '1' then
O(0) <= D1;
O(1) <= O(0);
O(2) <= O(1);
O(3) <= O(2);
end if;
end process;
end BEHAVIORAL;

A Behavioral Shift Register Model

-- A Behavioral Description of 4 Bit Shift Register

-- Libraries
library IEEE;
USE IEEE.std_logic_1164.all; -- to allow STD_LOGIC_VECTORS
USE IEEE.std_logic_arith.all; -- to allow arithmetic operations
USE IEEE.std_logic_unsigned.all; -- to allow integer conversions

entity SHIFT48 is
end SHIFT48;

architecture BEHAVIORAL of SHIFT48 is
begin
process (CLR)
begin
if CLR = '1' then
O(0) <= D1;
O(1) <= O(0);
O(2) <= O(1);
O(3) <= O(2);
end if;
end process;
end BEHAVIORAL;
Chapter 2

Another Behavioral Shift
Register Model

-- Another Behavioral Description of a 4 Bit Shift Register

-- Libraries
library IEEE;
USE IEEE.std_logic_1164.all; -- to allow STD_LOGIC_VECTORS
USE IEEE.std_logic_arith.all; -- to allow arithmetic operations
USE IEEE.std_logic_unsigned.all; -- to allow integer conversions

entity SHIFT4B is
  port(CLK,DI : in STD_LOGIC; -- Inputs
       O : out STD_LOGIC_VECTOR (3 downto 0)); -- Outputs
end SHIFT4B;

architecture BEHAVIORAL of SHIFT4B is
begin
process (CLK)
begin
  if (CLK = '1') then
    O <= O(2 downto 0) & DI;
  end if;
end process;
end BEHAVIORAL;

Shift Register Simulation Results

<table>
<thead>
<tr>
<th>clk</th>
<th>ps</th>
<th>delta</th>
<th>di</th>
<th>o</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5000</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5001</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10001</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15001</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Electrical and Computer Engineering

Spring 2004 Slide #27

Spring 2004 Slide #28
**Modeling a JK Flip-Flop**

```
entity JKFF is
  port (SN, RN, J, K, CLK: in bit; Q: inout bit; QN: out bit := '1'); -- inputs
end JKFF;

architecture JKFF1 of JKFF is
begin
  process (SN, RN, CLK) -- see Note 2
  begin
    if RN = '0' then Q<= '0' after 10 ns; -- RN=0 will clear the FF
    elsif SN = '0' then Q<= '1' after 10 ns; -- SN=0 will set the FF
    elsif CLK = '0' and CLK'event then
      Q <= (J and not Q) or (not K and Q) after 10 ns; -- see Note 3
    end if;
    end process;
    QN <= not Q; -- see Note 5
end JKFF1;
```

**Nested If-then-else Statements**

```
if (C1) then S1; S2;
  else if (C2) then S3; S4;
    else if (C3) then S5; S6;
      else S7; S8;
    end if;
  end if;
end if;
```

```
if (C1) then S1; S2;
  elsif (C2) then S3; S4;
    elsif (C3) then S5; S6;
      else S7; S8;
    end if;
  end if;
end if;
```
Modeling a Multiplexer

F <= (not A and not B and I0) or (not A and B and I1) or (A and not B and I2) or (A and B and I3);

MUX model using a conditional signal assignment statement:
F <= 10 when Sel = 0
  else 11 when Sel = 1
  else 12 when Sel = 2
  else 13;

The case statement has the general form:

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity SELECTOR is
  port(
    A : in std_logic_vector(15 downto 0);
    SEL: in std_logic_vector(3 downto 0);
    Y : out std_logic);
end SELECTOR;
architecture RTL1 of SELECTOR is
begin
  p0 : process (A, SEL)
  begin
    if (SEL = "0000") then       Y <= A(0);
    elsif (SEL = "0001") then    Y <= A(1);
    elsif (SEL = "0010") then    Y <= A(2);
    elsif (SEL = "0011") then    Y <= A(3);
    elsif (SEL = "0100") then    Y <= A(4);
    elsif (SEL = "0101") then    Y <= A(5);
    elsif (SEL = "0110") then    Y <= A(6);
    elsif (SEL = "0111") then    Y <= A(7);
    elsif (SEL = "1000") then    Y <= A(8);
    elsif (SEL = "1001") then    Y <= A(9);
    elsif (SEL = "1010") then    Y <= A(10);
    elsif (SEL = "1011") then    Y <= A(11);
    elsif (SEL = "1100") then    Y <= A(12);
    elsif (SEL = "1101") then    Y <= A(13);
    elsif (SEL = "1110") then    Y <= A(14);
    else                        Y <= A(15);
    end if;
  end process;
end RTL1;
```
Multiplexer Model With A Conditional Concurrent Statement

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity SELECTOR is
  port (A : in std_logic_vector(15 downto 0);
        SEL : in std_logic_vector( 3 downto 0);
        Y   : out std_logic);
end SELECTOR;

architecture RTL3 of SELECTOR is
begin
  with SEL select
  Y <= A(0)  when "0000",
      A(1)  when "0001",
      A(2)  when "0010",
      A(3)  when "0011",
      A(4)  when "0100",
      A(5)  when "0101",
      A(6)  when "0110",
      A(7)  when "0111",
      A(8)  when "1000",
      A(9)  when "1001",
      A(10) when "1010",
      A(11) when "1011",
      A(12) when "1100",
      A(13) when "1101",
      A(14) when "1110",
      A(15) when others;
end RTL3;

Modeling a Multiplexer with a Case Statement

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity SELECTOR is
  port (A : in std_logic_vector(15 downto 0);
        SEL : in std_logic_vector( 3 downto 0);
        Y   : out std_logic);
end SELECTOR;

architecture RTL2 of SELECTOR is
begin
  p1 : process (A, SEL)
  begin
    case SEL is
      when "0000" => Y <= A(0);
      when "0001" => Y <= A(1);
      when "0010" => Y <= A(2);
      when "0011" => Y <= A(3);
      when "0100" => Y <= A(4);
      when "0101" => Y <= A(5);
      when "0110" => Y <= A(6);
      when "0111" => Y <= A(7);
      when "1000" => Y <= A(8);
      when "1001" => Y <= A(9);
      when "1010" => Y <= A(10);
      when "1011" => Y <= A(11);
      when "1100" => Y <= A(12);
      when "1101" => Y <= A(13);
      when "1110" => Y <= A(14);
      when others => Y <= A(15);
    end case;
  end process;
end RTL2;
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity SELECTOR is
  port (  
    A   : in  std_logic_vector(15 downto 0);
    SEL : in  std_logic_vector( 3 downto 0);
    Y   : out std_logic);
end SELECTOR;

architecture RTL4 of SELECTOR is
begin
  Y <= A(conv_integer(SEL));
end RTL4;

Compilation and Simulation of VHDL

- Compiler (Analyzer) — checks the VHDL source code
  — does it conforms with VHDL syntax and semantic rules
  — are references to libraries correct
- Intermediate form used by a simulator or by a synthesizer
- Elaboration
  — create ports, allocate memory storage, create interconnections, ...
  — establish mechanism for executing of VHDL processes
Transport Delay

- Transport delay must be explicitly specified
  - I.e. keyword TRANSPORT must be used
- Signal will assume its new value after specified delay

--- TRANSPORT delay example
Output <= TRANSPORT NOT Input AFTER 10 ns;

Inertial Delay

- Provides for specification propagation delay and input pulse width, i.e. inertia of output:

  target <= [REJECT time_expression] INERTIAL waveform;

- Inertial delay is default and REJECT is optional:

  Output <= NOT Input AFTER 10 ns;
  -- Propagation delay and minimum pulse width are 10ns
Inertial Delay Modeling

Propagation Delay

- Example of gate with inertia smaller than propagation delay
  - e.g. Inverter with propagation delay of 10ns which suppresses pulses shorter than 5ns

```
Output <= REJECT 5ns INERTIAL NOT Input AFTER 10ns;
```

Note: the REJECT feature is new to VHDL 1076-1993

Simulation Example

```vhdl
entity simulation_example is
  signal A, B: bit;
end simulation_example;

architecture test1 of simulation_example is
  signal A, B: bit;
begin
  P1: process(B)
  begin
    A <= '1';
    A <= transport '0' after 5 ns;
  end process P1;

  P2: process(A)
  begin
    if A = '1' then B <= not B after 10 ns; end if;
  end process P2;
end test1;
```
Mealy Machine for
8421 BCD to 8421 BCD + 3 bit serial converter

How to model this in VHDL?

Behavioral VHDL Model

Two processes:
- the first represents the combinational network;
- the second represents the state register
-- The following is a description of the sequential machine of
-- Figure 1-17 in terms of its next state equations.
-- The following state assignment was used:
-- S0-0->D; S1-->4; S2-->5; S3-->7; S4-->6; S5-->3; S6-->2

entity SM1_2 is
  port(X, CLK: in bit;
          Z: out bit);
end SM1_2;

architecture Equations1_4 of SM1_2 is
  signal Q1, Q2, Q3: bit;
begin
  process(CLK)
  begin
    if CLK='1' then -- rising edge of clock
      Q1 <= not Q2 after 10 ns;
      Q2 <= Q1 after 10 ns;
      Q3 <= (Q1 and Q2 and Q3) or (not X and Q1 and not Q3) or
           (X and not Q1 and not Q2) after 10 ns;
    end if;
  end process;
  Z <= (not X and not Q3) or (X and Q3) after 20 ns;
end Equations1_4;

library BITLIB;
use BITLIB.bit_pack.all;

entity SM1_2 is
  port(X, CLK: in bit;
          Z: out bit);
end SM1_2;

architecture Structure of SM1_2 is
  signal A1, A2, A3, A5, A6, D3: bit= '0';
  signal Q1, Q2, Q3: bit= '0';
  signal Q1N, Q2N, Q3N, XN: bit= '1';
begin
  H1: Inverter port map (X,XN);
  G1: NAND3 port map (Q1, Q2, Q3, A1);
  G2: NAND3 port map (Q1, Q2N, XN, A2);
  G3: NAND3 port map (X, Q3N, Q2N, A3);
  G4: NAND3 port map (A1, A2, A3, D3);
  FF1: DFF port map (Q2N, CLK, Q1, Q1N);
  FF2: DFF port map (Q1, CLK, Q2, Q2N);
  FF3: DFF port map (D3, CLK, Q3, Q3N);
  G5: NAND2 port map (X, Q3, A5);
  G6: NAND2 port map (XN, Q3N, A6);
  G7: NAND2 port map (A5, A6, Z);
end Structure

Package bit_pack is a part of library
BITLIB — includes gates, flip-flops, counters
(See Appendix B for details)
Chapter 2

VHDL Model for A 74163 Counter

Control Signals

<table>
<thead>
<tr>
<th>ClrN</th>
<th>LdN</th>
<th>P+T</th>
<th>D3+</th>
<th>Q2+</th>
<th>Q1+</th>
<th>Q0+</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>D3</td>
<td>D2</td>
<td>D1</td>
<td>D0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Q3</td>
<td>Q2</td>
<td>Q1</td>
<td>Q0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Generate a Cout in state 15 if T=1

Cout = Q₃Q₂Q₁Q₀T

VHDL Model for a 74163 Counter

-- 74163 FULLY SYNCHRONOUS COUNTER
library BITLIB;
use BITLIB.bit_pack.all;
entity c74163 is
port(ClnN, ClrN, P+, T, CK: in bit; D: in bit_vector(3 downto 0);
    Cout: out bit; Q: inout bit_vector(3 downto 0));
end c74163;
architecture b74163 of c74163 is
begin
    Cout <= Q(3) and Q(2) and Q(1) and Q(0) and T;
    process
        begin
            wait until CK = '1';
            if ClrN = '0' then Q <= "0000";
            elsif LdN = '0' then Q <= D;
            elsif (P and T) = '1' then
                Q <= int2vec(vec2int(Q)+1,4);
            end if;
        end process;
    end b74163;
Cascade Counters (Block Diagram)

Cascaded Counters (VHDL)

```vhdl
library BITLIB;
use BITLIB.bit_pack.all;

entity c74163test is
  port(
    Clrn, Ldn, P, T1, Clk: in bit;
    Din1, Din2: in bit_vector(3 downto 0);
    Qout1, Qout2: inout bit_vector(3 downto 0);
    Carry2: out bit);
end c74163test;

architecture tester of c74163test is
  component c74163
    port(
      Ldn, Clrn, P, T, Clk: in bit;
      D: in bit_vector(3 downto 0);
      Cout: out bit; Q: inout bit_vector(3 downto 0));
  end component;
  signal Carry1: bit;
  signal Count: integer;
  signal temp: bit_vector(7 downto 0);
begin
  c1: c74163 port map (Ldn, Clrn, P, T, Clk, Din1, Carry1, Qout1);
  c2: c74163 port map (Ldn, Clrn, P, Carry1, Clk, Din2, Carry2, Qout2);
  temp <= Qout2 & Qout1;
  Count <= vec2int(temp);
end tester;
```
Wait Statements

- ... an alternative to a sensitivity list
  - Note: a process cannot have both wait statement(s) and a sensitivity list
- Generic form of a process with wait statement(s)

```vhdl
process
begin
  sequential-statements
  wait statement
  sequential-statements
  wait-statement
  ...
end process;
```

How do wait statements work?
- Execute sequential statements until a wait statement is encountered.
- Wait until the specified condition is satisfied.
- Then execute the next set of sequential statements until the next wait statement is encountered.
- When the end of the process is reached start over again at the beginning.

Forms of Wait Statements

```vhdl
wait on sensitivity-list;
wait for time-expression;
wait until boolean-expression;
```

- Wait on
  - until one of the signals in the sensitivity list changes
- Wait for
  - waits until the time specified by the time expression has elapsed
  - What is this: wait for 0 ns;
- Wait until
  - the boolean expression is evaluated whenever one of the signals in the expression changes, and the process continues execution when the expression evaluates to TRUE
Using **Wait** Statements

```vhdl
wait on CLK, X;
if rising_edge(CLK) then
    State <= Nextstate;
    wait for 0 ns;
end if;
end process;
```

---

**Variables**

- What they are for:
  - Local storage in processes, procedures, and functions
- Declaring variables
  ```vhdl
  variable list_of_variable_names : type_name
  [ := initial value ];
  ```
- Variables must be declared within the process in which they are used and are local to the process
  —Note: exception to this is **SHARED** variables
Signals

¥ Signals must be declared outside a process
¥ Declaration form

```vhdl
signal list_of_signal_names : type_name [
  := initial value ];
```

¥ Declared in an architecture can be used anywhere within that architecture

Constants

¥ Declaration form

```vhdl
constant constant_name : type_name := constant_value;
constant delay1 : time := 5 ns;
```

¥ Constants declared at the start of an architecture can be used anywhere within that architecture
¥ Constants declared within a process are local to that process
Variables versus Signals

- Variable assignment statement
  
  ```
  variable_name := expression;
  ```

  - expression is evaluated and the variable is
    instantaneously updated
    (no delay, not even delta delay)

- Signal assignment statement

  ```
  signal_name <= expression [after delay];
  ```

  - expression is evaluated and the signal is scheduled to
    change after delay; if no delay is specified the signal is
    scheduled to be updated after a delta delay

Variables versus Signals (continued)

Process Using Variables

```
entity dummy is
end dummy;

architecture var of dummy is
signal trigger, sum: integer:=0;
begin
  process
    variable var1: integer:=1;
    variable var2: integer:=2;
    variable var3: integer:=3;
    begin
      wait on trigger;
      var1 := var2 + var3;
      var2 := var1;
      var3 := var2;
      sum <= var1 + var2 + var3;
    end process;
  end var;
  Sum = ?
end entity
```
Predefined VHDL Types

- Variables, signals, and constants can have any one of the predefined VHDL types or they can have a user-defined type.

Predefined Types:
- bit —\{0, 1\}
- boolean —\{TRUE, FALSE\}
- integer —\{-2^{31} - 1.. 2^{31} - 1\}
- real —floating point number in range \(-1.0E38 \) to \(+1.0E38\)
- character —legal VHDL characters including lowercase letters, digits, special characters, ...
- time —an integer with units fs, ps, ns, us, ms, sec, min, or hr

User Defined Type

Common user-defined type is enumerated:

type state_type is (S0, S1, S2, S3, S4, S5);
signal state : state_type := S1;

If no initialization, the default initialization is the leftmost element in the enumeration list (S0 in this example).

VHDL is strongly typed => signals and variables of different types cannot be mixed in the same assignment statement, and no automatic type conversion is performed.
Arrays

Example

type SHORT_WORD is array (15 downto 0) of bit;
signal DATA_WORD : SHORT WORD;
variable ALT_WORD : SHORT WORD := "0101010101010101";
constant ONE_WORD : SHORT WORD := (others => '1');

ALT_WORD(0) — rightmost bit
ALT_WORD(5 downto 0) — low order 6 bits

General form

type arrayTypeName is array index_range of element_type;
signal arrayName : arrayTypeName [:=InitialValues];

Multidimensional arrays

type matrix4x3 is array (1 to 4, 1 to 3) of integer;
variable matrixA: matrix4x3 :=
((1,2,3), (4,5,6), (7,8,9), (10,11,12));

matrixA(3, 2) = ?

Unconstrained array type

type intvec is array (natural range<>) of integer;
type matrix is array (natural range<> ,natural range<> )
of integer;

range must be specified when the array object is declared

signal intvec5 : intvec(1 to 5) :=
(3,2,6,8,1);
**Predefined Unconstrained Array Types**

```vhdl
type bit_vector is array (natural range <>) of bit;
type string is array (positive range <>) of character;
constant string1: string(1 to 29) := "This string is 29 characters."

constant A : bit_vector(0 to 5) := "10101";
  -- ('1', '0', '1', '0', '1', '0', '1');

¥ Subtypes

¥ include a subset of the values specified by the type

```vhdl
subtype SHORT_WORD is : bit_vector(15 to 0);
```
VHDL Operators

- Class 7 has the highest precedence (applied first), followed by class 6, then class 5, etc.

1. Binary logical operators: **and** or **nand** **nor** **xor** **xnor**
2. Relational: **=** /= **<** <= **>** >=
3. Shift: **sll** **srl** **sla** **sra** **rol** **ror**
4. Adding: **+** - & (concatenation)
5. Unary sign: **+** -
6. Multiplying: ***** / **mod** **rem**
7. Miscellaneous: **not** **abs** ****

VHDL Operator Example

In the following expression, A, B, C, and D are bit vectors:

\[(A \& \text{not } B \text{ or } C \text{ ror } 2 \text{ and } D) = "110010"\]

The operators would be applied in the order:

**not**, & **ror**, **or**, **and**, **=**

If \(A = "110", B = "111", C = "011000",\) and \(D = "110111",\) the computation would proceed as follows:

- **not** B = "000" (bit-by-bit complement)
- A & **not** B = "110000" (concatenation)
- C ror 2 = "000110" (rotate right 2 places)
- (A & **not** B) or (C ror 2) = "110110" (bit-by-bit or)
- (A & **not** B or C ror 2) and D = "110010" (bit-by-bit and)
- [(A & **not** B or C ror 2 and D) = "110010"] = TRUE

(with the parentheses force the equality test to be done last and the result is TRUE)
Shift Operator Example

The shift operators can be applied to any bit_vector or boolean_vector. In the following examples, A is a bit_vector equal to "10010101":

A sll 2 is "01010100" (shift left logical, filled with '0')
A srl 3 is "00010010" (shift right logical, filled with '0')
A sia 3 is "10101111" (shift left arithmetic, filled with right bit)
A sra 2is "11100101" (shift right arithmetic, filled with left bit)
A rol 3 is "10101100" (rotate left)
A ror 5 is "10101100" (rotate right)

VHDL Functions

Functions execute a sequential algorithm and return a single value to calling program

```vhdl
function rotate_right (reg: bit_vector)
return bit_vector is
begin
return reg ror 1;
end rotate_right;
B <= rotate_right(A);
```

General form

```vhdl
function function-name (formal-parameter-list)
return return-type is
[declarations]
begin
sequential statements -- must include return return-value;
end function-name;
```

For Loops

General form of a for loop:

[loop-label:] for loop-index in range loop
  sequential statements
end loop [loop-label];

For Loop Example:

-- compare two 8-character strings and return TRUE if equal
function comp_string(string1: string, string2: string(1 to 8))
  return boolean is
variable B: boolean;
begin
  loopex: for j in 1 to 8 loop
    B := string1(j) = string2(j);
    exit when B = FALSE;
  end loop loopex;
  return B;
end comp_string;

Add Function

-- This function adds 2 4-bit vectors and a carry.
-- It returns a 5-bit sum
function add4 (A,B: bit_vector(3 downto 0); carry: bit)
  return bit_vector is
variable cout: bit;
variable cin: bit := carry;
variable Sum: bit_vector(4 downto 0):="00000";
begin
  loop1: for i in 0 to 3 loop
    cout := (A(i) and B(i)) or (A(i) and cin) or (B(i) and cin);
    Sum(i) := A(i) xor B(i) xor cin;
    cin := cout;
  end loop loop1;
  Sum(4):= cout;
  return Sum;
end add4;

Example function call:

Sum1 <= add4(A1, B1, cin);
VHDL Procedures

Procedures can return any number of values using output parameters

General form

```vhdl
procedure procedure_name (formal-parameter-list) is
    [declarations]
begin
    Sequential-statements
end procedure_name;
procedure_name (actual-parameter-list);
```

Procedure for Adding Bit_Vectors

```vhdl
-- This procedure adds two n-bit bit_vectors and a carry and
-- returns an n-bit sum and a carry. Add1 and Add2 are assumed
-- to be of the same length and dimensioned n-1 downto 0.

procedure Addvec
    (Add1,Add2: in bit_vector;
     Cin: in bit;
     Sum: out bit_vector;
     Cout: out bit;
     n: in positive) is
    variable C: bit;
begin
    C := Cin;
    for i in 0 to n-1 loop
        Sum(i) <= Add1(i) xor Add2(i) xor C;
        C := (Add1(i) and Add2(i)) or (Add1(i) and C) or (Add2(i) and C);
    end loop;
    Cout <= C;
end Addvec;
```

Example procedure call:
```
Addvec(A1, B1, Cin, Sum1, Cout, 4);
```
## Parameters for Subprogram Calls

<table>
<thead>
<tr>
<th>Mode</th>
<th>Class</th>
<th>Procedure Call</th>
<th>Function Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>in$^1$</td>
<td>constant$^2$</td>
<td>expression</td>
<td>expression</td>
</tr>
<tr>
<td></td>
<td>signal</td>
<td>signal</td>
<td>signal</td>
</tr>
<tr>
<td></td>
<td>variable</td>
<td>variable</td>
<td>n/a</td>
</tr>
<tr>
<td>out/inout</td>
<td>signal</td>
<td>signal</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>variable$^3$</td>
<td>variable</td>
<td>n/a</td>
</tr>
</tbody>
</table>

$^1$ default mode for functions  $^2$ default for in mode  $^3$ default for out/inout mode

---

## Packages and Libraries

- Provide a convenient way of referencing frequently used functions and components

- **Package header**
  ```
  package package-name is
  package declarations
  end [package][package-name];
  ```

- **Package body [optional]**
  ```
  package body package-name is
  package body declarations
  end [package body][package name];
  ```
package bit_pack is

function add4 (reg1,reg2: bit_vector(3 downto 0);carry: bit)
  return bit_vector;
function falling_edge(signal clock:bit)
  return Boolean;
function rising_edge(signal clock:bit)
  return Boolean;
function vec2bit(vec1: bit_vector)
  return integer;
function int2vec(int1, NBits: integer)
  return bit_vector;
procedure Add2vec
  (Add1,Add2: in bit_vector;
   Cn: in bit;
   signal Sum: out bit_vector;
   signal Cout: out bit;
   n: in natural);

component jkff
  generic DELAY:time := 10 ns);
  port(SN, RN, J,K,CLK: in bit; Q, QN: inout bit);
  end component;

component dff
generic DELAY:time := 10 ns);
  port(D, CLK: in bit; Q: out bit; QN: out bit := '1');
  end component;

end bit_pack;

-- This function adds 2 4-bit numbers, returns a 5-bit sum
function add4(lreg1,lreg2: bit_vector(3 downto 0);carry) return bit_vector is
variable cout: bit:= '0';
variable cin: bit:= carry;
variable retval: bit_vector(4 downto 0):= "00000";
begin
  for i in 0 to 3 loop
    cout := (lreg1(i) and lreg2(i)) or (lreg1(i) and cin) or
      (lreg2(i) and cin);
    retval(i) := lreg1(i) xor lreg2(i) xor cin;
  loop b1;
    retval(4):= cout;
  end loop b1;
  cout(:= cout;
return retval;
end add4;

function falling_edge(signal clock:bit)
  return Boolean is
begin
  return clock'event and clock = '0';
end falling_edge;

end bit_pack;
Library BITLIB – bit_pack package

Components in Library BITLIB include:
-- 3 input AND gate
entity And3 is
  generic(Delay_time);
  port (A1, A2, A3: in bit; Z: out bit);
end And3;
architecture conc_of And3 is
begin
  Z <= A1 and A2 and A3 after Delay;
end;
-- D flip-flop
entity DFF is
  generic(Delay_time);
  port (Q, CLK: in bit;
   Qn: out bit; Qn: out bit := 'Y');
-- initialize Qn to 'Y' since bit signals are initialized to '0' by default
end DFF;
architecture simple_of DFF is
begin
  process(CLk)
  begin
    if CLK = '1' then
      Q <= Q after Delay;
      Qn <= not Q after Delay;
    end if;
  end process;
end SIMPLE;

Design of a Keypad Scanner
Design of a Keypad Scanner

Electrical and Computer Engineering

Spring 2004 Slide #78