Advanced VLSI Design

Lecture 2: VHDL Recapitulation

Aleksandar Milenkovic

Outline

- Introduction to VHDL
- Modeling of Combinational Networks
- Modeling of FFs
- Delays
- Modeling of FSMs
- Wait Statements
- VHDL Data Types
- VHDL Operators
- Functions, Procedures, Packages

Intro to 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

VHDL Description of Combinational Networks

Entity-Architecture Pair

Full Adder Example

```
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;
```

CPE 626

Advanced VLSI Design

Lecture 2: VHDL Recapitulation

Aleksandar Milenkovic

http://www.ece.uah.edu/~milenka

http://www.ece.uah.edu/~milenka/cpe626-04F/

milenka@ece.uah.edu

Assistant Professor
Electrical and Computer Engineering Dept.
University of Alabama in Huntsville
Whenever one of the signals in the sensitivity list changes, the sequential statements are executed in sequence one time.
**JK Flip-Flop Model**

**Concurrent Statements vs. Process**

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

**Simulation Results**

<table>
<thead>
<tr>
<th>time</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

**MUX Models (1)**

```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;
```

**MUX Models (2)**

```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 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 with;
end RTL3;
```
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 Code
- 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

Timing Model
- VHDL uses the following simulation cycle to model the stimulus and response nature of digital hardware

Delay Types
- All VHDL signal assignment statements prescribe an amount of time that must transpire before the signal assumes its new value
- This prescribed delay can be in one of three forms:
  - Transport -- prescribes propagation delay only
  - Inertial -- prescribes propagation delay and minimum input pulse width
  - Delta -- the default if no delay time is explicitly specified
Inertial Delay

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

\[
\text{target} <= \text{REJECT time_expression} \quad \text{INERTIAL waveform};
\]

- Inertial delay is default and REJECT is optional:

\[
\text{Input} \quad \text{Output}
\]

\[
\begin{array}{cccccccc}
0 & 5 & 10 & 15 & 20 & 25 & 30 & 35
\end{array}
\]

\[
\text{Output} <= \text{NOT Input AFTER 10 ns;}
\]

\[
\text{--- Propagation delay and minimum pulse width are 10ns}
\]

Delta Delay

- Default signal assignment propagation delay if no delay is explicitly prescribed
- VHDL signal assignments do not take place immediately
- Delta is an infinitesimal VHDL time unit so that all signal assignments can result in signals assuming their values at a future time

- E.g.: \( \text{Output} <= \text{NOT Input; \quad \text{Output assume new value in one delta cycle} } \)

- Supports a model of concurrent VHDL process execution

- Order in which processes are executed by simulator does not affect simulation output

Simulation Example

- Supports a model of concurrent VHDL process execution
- Order in which processes are executed by simulator does not affect simulation output

Modeling a Sequential Machine

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.

Simulation of the VHDL Model

Simulation command file:
```
wave CLK X State NextState Z
force CLK 0 0, 1 100 -repeat 200
force X 0 0, 1 350, 0 550, 1 750, 0 950, 1 1350
run 1600
```

Waveforms:

Dataflow VHDL Model

- The following is a description of the sequential machine of
- Figure 1.7.4 terms of its local state equations.
- The following state assignment was used:
  - S0 = S0; S1 = S1; S2 = S2; S3 = S3

Entity and Architecture:
```
entity SM_2 is
  port(Clk: in bit; Z: out bit);
architecture Behavior of SM_2 is
begin
  process(Clk)
  begin
    if Clk = '1' then
      Q<1> := Q<1> or not (Q<0> and Q<3>); 
      Q<3> := Q<3> or not (Q<0> and Q<1>); 
      Z := Q<1> or Q<3> or (Q<0> and Q<1>). 
    else
      Z := Q<0> and Q<3> or (Q<0> and Q<1>). 
    end if;
  end process;
end architecture;
```

Simulation of the Structural Model

Simulation command file:
```
wave CLK X Q2 Q3 Z
force CLK 0 0, 1 100 -repeat 200
force X 0 0, 1 350, 0 550, 1 750, 0 950, 1 1350
run 1600
```

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
    ...
  end process;
  ```
  ```
  How wait statements work?
  - Execute seq. statement 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

- 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
- 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

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

Using Wait Statements (1)

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

Using Wait Statements (2)

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

Problem #1

Using the labels, list the order in which the following signal assignments are evaluated if in2 changes from a '0' to a '1'. Assume in1 has been a '1' and in2 has been a '0' for a long time, and then at time t in2 changes from a '0' to a '1'.

```
entity not_another_prob is
port (in1, in2: in bit;
 a: out bit);
end not_another_prob;
architecture oh_behave of not_another_prob is
signal b, c, d, e, f: bit;
begin
L1:  d <= not(in1);
L2:  c <= not(in2);
L3:  f <= (d and in2) ;
L4:  e <= (c and in1) ;
L5:  a <= not b;
L6:  b <= e or f;
ed end_behave;
```

Problem #2

Under what conditions do the two assignments below result in the same behavior? Different behavior? Draw waveforms to support your answers.

```
out <= reject 5 ns inertial (not a) after 20 ns;
out <= transport (not a) after 20 ns;
```

Variables

- What are they for: Local storage in processes, procedures, and functions
- Declaring variables

```
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;
  ```
  • 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 vs. Signals
- Variable assignment statements
  - expression is evaluated and the variable is instantaneously updated (no delay, not even delta delay)
  ```vhdl
  variable_name := expression;
  ```
  • Signal assignment statement
  ```vhdl
  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

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 \(-1.0\text{E}38\) to \(+1.0\text{E}38\)
  - character – legal VHDL characters including lowercase, uppercase letters, digits, special characters, ...
  - time – an integer with units fs, ps, ns, vs, sec, min, or hr

User Defined Type
- Common user-defined type is `enumerated`
  ```vhdl
  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 language => signals and variables of different types cannot be mixed in the same assignment statement, and no automatic type conversion is performed
Arrays

- Example
  ```vhdl
type SHORT_WORD is array (15 downto 0) of bit;
signal DATA_WORD : SHORTWORD;
variable ALT_WORD : SHORTWORD := "0101010101010101";
constant ONE_WORD : SHORTWORD := (others => '1');
```
- General form
  ```vhdl
type arrayTypeName is array index_range of element_type;
signal arrayName : arrayTypeName [:=InitialValues];
```

Arrays (cont’d)

- Multidimensional arrays
  ```vhdl
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));
constant A : matrix4x3 :=
((1, 2, 3), (4, 5, 6), (7, 8, 9), (10, 11, 12));
```

- Unconstrained array type
  ```vhdl
type matrix is array (natural range<>), natural range<> of integer;
variable matrixB : matrix :=
((1, 2, 3), (4, 5, 6), (7, 8, 9), (10, 11, 12));
```

Sequential Machine Model Using State Table

- Predefined Unconstrained Array Types
  ```vhdl```

  **Bit_vector, string**
  ```vhdl
type bit_vector is array (natural range<>) of bit;
type string is array (natural range<>) of character;
```

  **Subtypes**
  ```vhdl```

  - include a subset of the values specified by the type
  ```vhdl```

  **POSITIVE, NATURAL** — predefined subtypes of type integer

VHDL Operators

- Binary logical operators: and or nand nor xor xnor
- Relational: /= <= < <= > >=
- Shift: sll srl sla sra rol ror
- Adding: + - & (concatenation)
- Unary sign: + -
- Multiplying: * / mod rem
- Miscellaneous: not abs **
- Class 7 has the highest precedence (applied first), followed by class 6, then class 5, etc
Example of Shift Operators (cont'd)

The shift operators can be applied to any bit, vector or boolean vector. In the following examples, A is a bit, vector equal to ‘00010101’:

- **A sll 2** is ‘01001010’ (shift left logical, filled with ‘0’)
- **A srl 3** is ‘00001010’ (shift right logical, filled with ‘0’)
- **A sla 3** is ‘10110111’ (shift left arithmetic, filled with right bit)
- **A sra 2** is ‘11100111’ (shift right arithmetic, filled with left bit)
- **A roh 5** is ‘10101100’ (rotate right)

VHDL Functions

- Functions execute a sequential algorithm and return a single value to calling program
- General form
  - function function-name (formal-parameter-list)
    return return-type is
    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
  - exit statement has the form:
    - exit; -- or
    - exit when condition;

  *For Loop Example:*

  ```
  begin
  for i in 1 to 10 loop
    if i = 5 then
      exit when i = 5;
  end loop;
  end for i loop;
  ```

VHDL Procedures

- Facilitate decomposition of VHDL code into modules
- Procedures can return any number of values using output parameters

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

  procedure_name (actual-parameter-list);
  ```

Add Function

- This function adds 2 i-bit vectors and a carry.
- It returns a 5-bit sum
  ```
  function add (A, B, bit_vector; carry: bit) return bit_vector is
    variable Sum, Carry: bit;
    variable Sum, Carry: bit_vector; carry: bit;
    begin
    for i in 0 to 3 loop
      Sum(i) := A(i) + B(i) + Carry(i);
      Carry(i) := Sum(i) and Carry(i);
      end loop;
    end add;
  ```

Procedure for Adding Bit_vectors

- This procedure adds two i-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 dimension n-1 downto 0

  ```
  procedure Addvec
  (Add1, Add2 in bit_vector;
  Carry in bit;
  signal Sum: out bit_vector;
  signal C: out bit;
  n-in position) is
  variable C: bit;
  begin
  C <= Carry;
  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;
  end Addvec;
  ```

Example procedure call:

  ```
  Addvec(A, B, C, Sum, C, 4);
  ```
Parameters for Subprogram Calls

<table>
<thead>
<tr>
<th>Mode</th>
<th>Class</th>
<th>Procedure Parameter</th>
<th>Function Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>constant</td>
<td>expression</td>
<td>expression</td>
</tr>
<tr>
<td>out/inout</td>
<td>variable</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 declaration
    ```vhd
    package package-name is
    package declarations;
    end package;
    ```

  • Package body [optional]
    ```vhd
    package body package-name is
    package body declarations;
    end package body;
    ```

Library BITLIB – bit_pack package

```vhd
package bit_pack is
  function bit_pack(...) return bit;
  function bit_vector(...) return bit_vector;
  return bit_vector(...);
  function sig2numeric(...) return sig;
  function numeric2sig(...) return numeric;
  return numeric(...);
  function numeric2signal(...) return signal;
  function signal2numeric(...) return numeric;
  return numeric(...);
  function signal2sig(...) return sig;
  function sig2signal(...) return sig;
  return sig(...);
  procedure bit_pack(...);
  procedure sig2numeric(...);
  procedure numeric2sig(...);
  procedure signal2numeric(...);
  procedure numeric2signal(...);
  procedure signal2sig(...);
end package;
```

Additional Topics in VHDL

- Attributes
- Transport and Inertial Delays
- Operator Overloading
- Multivalued Logic and Signal Resolution
- IEEE 1164 Standard Logic
- Generics
- Generate Statements
- Synthesis of VHDL Code
- Synthesis Examples
- Files and Text IO
Signal Attributes

Attributes associated with signals that return a value

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>S'EVENT</td>
<td>True if an event occurred during the current delta, else False</td>
</tr>
<tr>
<td>S'ACTIVE</td>
<td>True if a transaction occurred during the current delta, else false</td>
</tr>
<tr>
<td>S'LAST_EVNT</td>
<td>Time elapsed since the previous event on S</td>
</tr>
<tr>
<td>S'LAST_VALUE</td>
<td>Value of S before the previous event on S</td>
</tr>
<tr>
<td>S'LAST_ACTIVE</td>
<td>Time elapsed since previous transaction on S</td>
</tr>
</tbody>
</table>

A'event – true if a change in S has just occurred

A'active – true if A has just been reevaluated, even if A does not change

Review: Signal Attributes (cont’d)

Attributes that create a signal

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Creates</th>
</tr>
</thead>
<tbody>
<tr>
<td>S'DELAYED (time)*</td>
<td>Signal same as S delayed by specified time</td>
</tr>
<tr>
<td>S'SMALLE (time)*</td>
<td>Boolean signal that is true if S had no events for the specified time</td>
</tr>
<tr>
<td>S'QUIET (time)*</td>
<td>Boolean signal that is true if S had no transactions for the specified time</td>
</tr>
<tr>
<td>S'TRANSACTION</td>
<td>Signal of type BIT that changes for every transaction on S</td>
</tr>
</tbody>
</table>

* Delay is used if no time is specified

Array Attributes

A can be either an array name or an array type.

Array attributes work with signals, variables, and constants.

Transport and Inertial Delay

Z3 <= transport * after 10 ns;  -- transport delay
Z2 <= 8 after 10 ns;            -- inertial delay
Z1 <= zero if x > after 10 ns;  -- delay with specified rejection pulse width

Review: Operator Overloading

- Operators +, - operate on integers
- Write procedures for bit vector addition/subtraction
- advec, subvec
- Operator overloading allows using + operator to implicitly call an appropriate addition function
- How does it work?
- When compiler encounters a function declaration in which the function name is an operator enclosed in double quotes, the compiler treats the function as an operator overloading ("+")
- When a "+" operator is encountered, the compiler automatically checks the types of operands and calls appropriate functions

VHDL Package with Overloaded Operators

```
-- This package provides two overloaded functions for the plus operator
package bit_vector is
  function + (A : in bit_vector; B : in bit_vector) return bit_vector;
  function + (A : in bit_vector; B : in bit_vector; C : in bit_vector) return bit_vector;
end bit_vector;
```

-- operator overloads
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
package body bit_vector is
  function + (A : in bit_vector; B : in bit_vector) return bit_vector is
    variable sum : bit_vector := "0";
    variable carry : bit := '0';
  begin
    for i in A'range loop
      sum(i) := (A(i) + B(i)) + carry;
      if (A(i) + B(i) + carry) /= sum(i) then
        carry := '1';
      else
        carry := '0';
      end if;
    end loop;
    return sum;
  end function +;
end body;
```
Multivalued Logic

- Bit (0, 1)
- Tristate buffers and buses => high impedance state ‘Z’
- Unknown state ‘X’
  - e.g., a gate is driven by ‘Z’, output is unknown
  - a signal is simultaneously driven by ‘0’ and ‘1’

Tristate Buffers

Resolution function to determine the actual value of f since it is driven from two different sources

Signal Resolution

- VHDL signals may either be resolved or unresolved
- Resolved signals have an associated resolution function
- Bit type is unresolved —
  - there is no resolution function
  - if you drive a bit signal to two different values in two concurrent statements, the compiler will generate an error

Signal Resolution (cont’d)

- `signal R : X01Z := 'Z';`
- `R <= transport '0' after 2 ns, 'Z' after 6 ns;`
- `R <= transport '1' after 4 ns;`
- `R <= transport '1' after 8 ns, '0' after 10 ns;`

Resolution Function for X01Z

```
package body X01Z is
    type x01z_vector is array (0 to 3) of x01z;
    constant resolution : x01z_vector :=
        (0 => '0', 1 => '0', 2 => '0', 3 => 'Z');
    function resolve (x : x01z_vector) return x01z is
    begin
        return resolution (x);  
    end function;
end X01Z;
```

AND and OR Functions Using X01Z

Define AND and OR for 4-valued inputs?
IEEE 1164 Standard Logic

- 9-valued logic system
  - 'U' – Uninitialized
  - 'X' – Forcing Unknown
  - '0' – Forcing 0
  - '1' – Forcing 1
  - 'Z' – High impedance
  - 'W' – Weak unknown
  - 'L' – Weak 0
  - 'H' – Weak 1
  - '-' – Don't care

If forcing and weak signal are tied together, the forcing signal dominates.

Useful in modeling the internal operation of certain types of ICs.

The IEEE 9-valued logic system has useful digits for various applications. It can be used in modeling devices where more than just '0' and '1' values are meaningful. The digits are as follows:

- 'U' – Uninitialized
- 'X' – Forcing Unknown
- '0' – Forcing 0
- '1' – Forcing 1
- 'Z' – High impedance
- 'W' – Weak unknown
- 'L' – Weak 0
- 'H' – Weak 1
- '-' – Don’t care

If forcing and weak signal are tied together, the forcing signal dominates.

Useful in modeling the internal operation of certain types of ICs.

In this course we use a subset of the IEEE values: X10Z

Resolution Function for IEEE 9-valued

AND Table for IEEE 9-valued

AND Function for std_logic_vectors

Generics

- Used to specify parameters for a component in such a way that the parameter values must be specified when the component is instantiated

Example: rise/fall time modeling
Generate Statements
- Provides an easy way of instantiating components when we have an iterative array of identical components
- Example: 4-bit RCA

4-bit Adder
- entity Adder4 is
  port (A, B: in bit_vector(3 downto 0); C: in bit; -- Inputs
  S, O: out bit_vector(3 downto 0); Ci: out bit); -- Outputs
end Adder4;
- architecture Structure of Adder4 is
  component FullAdder
  port (Xi, Yi, Ci: in bit;
  Cout, Sum, S: out bit);
  -- Inputs
  -- Outputs
end component;
signal C: bit_vector(3 downto 1);
begin -- instantiate four copies of the FullAdder
FA0: FullAdder port map (A(0), B(0), C(0), S(0));
FA1: FullAdder port map (A(1), B(1), C(1), S(1));
FA2: FullAdder port map (A(2), B(2), C(2), S(2));
FA3: FullAdder port map (A(3), B(3), C(3), S(3));
end Structure;

4-bit Adder using Generate
- entity Adder4 is
  port (A, B: in bit_vector(3 downto 0); C: in bit; -- Inputs
  S, O: out bit_vector(3 downto 0); Ci: out bit); -- Outputs
end Adder4;
- architecture Structure of Adder4 is
  component FullAdder
  port (Xi, Yi, Ci: in bit;
  Cout, Sum, S: out bit);
  -- Inputs
  -- Outputs
end component;
signal C: bit_vector(3 downto 0);
begin
  C(0) <= C;
  -- generate four copies of the FullAdder
  generate
  for i in 0 to 3 generate
  begin
  FAi: FullAdder port map (A(i), B(i), C(i), S(i));
  end generate
  end generate FullAdder;
  C(0) <= C(0);
end Structure;

Files
- File input/output in VHDL
- Used in test benches
- Source of test data
- Storage for test results
- VHDL provides a standard TEXTIO package
- read/write lines of text

Standard TEXTIO Package
- Contains declarations and procedures for working with files composed of lines of text
- Defines a file type named text:
  type text is file of string;
- Contains procedures for reading lines of text from a file of type text and for writing lines of text to a file
Reading TEXTO file

- `Readline` reads a line of text and places it in a buffer with an associated pointer.
- Pointer to the buffer must be of type line, which is declared in the textio package as:
  ```
  type line is access string;
  ```
- When a variable of type line is declared, it creates a pointer to a string.
- Code
  ```
  variable buff: line;
  ...
  readline (test_data, buff);
  ```
  reads a line of text from test_data and places it in a buffer which is pointed to by buff.

Extracting Data from the Line Buffer

- To extract data from the line buffer, call a read procedure one or more times.
- For example, if `bv4` is a bit_vector of length four, the call
  ```
  read (buff, bv4)
  ```
  extracts a 4-bit vector from the buffer, sets `bv4` equal to this vector, and adjusts the pointer `buff` to point to the next character in the buffer. Another call to `read` will then extract the next data object from the line buffer.

Extracting Data from the Line Buffer (cont'd)

- TEXTIO provides overloaded read procedures to read data of types bit, bit_vector, boolean, character, integer, real, string, and time from buffer.
- Read forms:
  ```
  read(pointer, value)
  read(pointer, value, good)
  ```
  good is boolean that returns TRUE if the read is successful and FALSE if it is not.
- Type and size of value determines which of the read procedures is called:
- Character, strings, and bit_vectors within files of type text are not delimited by quotes.

Writing to TEXTO files

- Call one or more write procedures to write data to a line buffer and then call writeln to write the line to a file.
- Code sequence: an example
  ```
  12AC 7 (7 hex bytes follow)
  AE 03 B6 01 C7 02 SC (LDX imm, LDA dir, STA ext)
  005B 2 (2 bytes follow)
  01 FC_
  ```

An Example

- Procedure to read data from a file and store the data in a memory array.
- Format of the data in the file:
  ```
  address N comments
  byte1 byte2 ... byteN comments
  ```
  • address – 4 hex digits
  • N – indicates the number of bytes of code
  • byte - 2 hex digits
  • each byte is separated by one space
  • the last byte must be followed by a space
  • anything following the last state will not be read and will be treated as a comment.

An Example (cont'd)

- Code sequence: an example
  ```
  12AC 7 (7 hex bytes follow)
  AE 03 B6 01 C7 02 SC (LDX imm, LDA dir, STA ext)
  005B 2 (2 bytes follow)
  01 FC_
  ```

- TEXTIO does not include read procedure for hex numbers.
- We will read each hex value as a string of characters and then convert the string to an integer.
- How to implement conversion?
  ```
  • table lookup – constant named lookup is an array of integers indexed by characters in the range '0' to 'F'
  ```
  • this range includes the 23 ASCII characters:
    ```
    '0', '1', ... '9', ':', ';', '<', '=', '>', '?', '@', 'A', ... 'F'
    ```
  • corresponding values:
    ```
    0, 1, ... 9, -1, -1, -1, -1, -1, -1, 10, 11, 12, 13, 14, 15
    ```
### VHDL Code to Fill Memory Array

**Library** uses:
- std_logic_1164.all
- std_logic_arith.all
- conv_std_logic_vector(16, size)

**Entity** `addrfill` is

```vhdl
architecture fillmem of addrfill is
  type RAMtype is array (0 to 15) of std_logic_vector(1 downto 0);
  signal mem: RAMtype := others => (others => '0');
procedure fill_memory(signal mem: RAMtype; character range is n of integer)
  -- valid hex char: 0, 1, ..., A, B, C, D, E, F (upper case only)
constant setup: std_logic_vector(0 to 7) :=
  (0, 1, 2, 3, 4, 5, 6, 7);
-1, -1, -1, -1, 10, 11, 12, 13, 14, 15);
file input: text open read mode is "memory.txt" -- open file for reading
-- file name: set in "memo.txt" -- VHDL '97 version
variable buf: length is integer range 0 downto 0);
variable addr_s: string(4 downto 1));
variable data_s: string(16 downto 1));
-- data_s[15] has a space
variable addr, byte_cnt, integer_range: integer range 255 downto 0);
```

### Synthesis of VHDL Code

- **Synthesizer**
  - take a VHDL code as an input
  - synthesize the logic: output may be a logic schematic with an associated wirelist
- **Synthesizers accept a subset of VHDL as input**
- **Efficient implementation?**
- **Context**
  ```vhdl
  -- A <= B and C;
  wait until clk'event and clk = '1';
  A <= B and C;
  Implies CM for A Implies a register or flip-flop
  ```

### Advanced VLSI Design

**Synthesis of VHDL Code (cont'd)**

- When use integers specify the range
- if not specified, the synthesizer may infer 32-bit register
- When integer range is specified, most synthesizers will implement integer addition and subtraction using binary adders with appropriate number of bits
- General rule: when a signal is assigned a value, it will hold that value until it is assigned new value

### Unintentional Latch Creation

- **If Statements**
  ```vhdl
  if A = '1' then NextState <= 3;
  end if;
  What if A = 1?
  Retain the previous value for NextState?
  Synthesizer might interpret this to mean that NextState is unknown!
  if A = '1' then NextState <= 3;
  else NextState <= 2;
  end if;
  ```
Advanced VLSI Design

Synthesis of an If Statement

```vhdl
entity if_example is
port(
    C: bit_vector; Z: bit_vector;
    E: bit_vector;
); end if_example;
architecture test of if_example is begin
  if E = '0' then Z := C;
  elsif E = '1' then Z := '0';
  else Z := E;
  end if;
end test;
```

Synthesized code before optimization

Advanced VLSI Design

Synthesis of a Case Statement

```vhdl
entity case_example is
port(
    A: integer range 0 to 3;
    B: bit_vector;
); end case_example;
architecture test of case_example is begin
  case A is
    when 0 => B <= '1';
    when 1 => B <= '0';
    when 2 => B <= '1';
    when 3 => B <= '0';
  end case;
end test;
```

Advanced VLSI Design

Case Statement: Before and After Optimization

```vhdl
begin
  A <= '0';
end;
```

Advanced VLSI Design

Standard VHDL Synthesis Package

- Every VHDL synthesis tool provides its own package of functions for operations commonly used in hardware models
- IEEE is developing a standard synthesis package, which includes functions for arithmetic operations on bit_vectors and std_logic vectors
  - `numeric_bit` package defines operations on bit_vectors
    - Type `unsigned` is array (natural range<> of bit
    - Type `signed` is array (natural range<> of bit
  - Package includes overloaded versions of arithmetic, relational, logical, and shifting operations, and conversion functions
  - `numeric_std` package defines similar operations on std_logic vectors

Advanced VLSI Design

Numeric_bit, Numeric_std

- Overloaded operators
  - Unary: abs, -
  - Arithmetic: +, -, *, /, rem, mod
  - Relational: >, <, >=, <=, =, /=
  - Logical: not, and, or, nand, nor, xor, xnor
  - Shifting: shift_left, shift_right, rotate_left, rotate_right, sll, srl, rol, ror

Advanced VLSI Design

Numeric_bit, Numeric_std (cont’d)

If the left and right signed operands are of different lengths, the shortest operand will be sign-extended before performing an arithmetic operation. For unsigned operands, the shortest operand will be extended by filling in 0’s on the left. Examples:

Signed:
- Signed: “0110111” + “0111” becomes “0110111” + “0000001”
- Signed: “0011111” + “0000001” becomes “0011111” + “0000001”

When addition is performed on unsigned or signed operands, the final carry is discarded and overflow is ignored. If a carry is needed, an extra bit can be added to one of the operands. Examples:
**Synthesis Examples (1)**

- **library ieee;**
- **use ieee.std_logic_1164.all;**
- **use ieee.std_logic_arith.all;**

- **entity example is**
  - **port (signal clk : in std_logic;**
  - **signal a, b : in std_logic_vector(15 downto 0);**
  - **signal c : in std_logic_vector(15 downto 0);**
  - **signal d : out std_logic_vector(15 downto 0);**

- **end example:**
  - **architecture hv of example is**
    - **begin**
      - **wait until clock'event and clock = '1';**
      - **if a = b and c = d then**
        - **d <= a + b;**
        - **d <= a + b;**
      - **end if;**
      - **end process;**

**Synthesis Examples (2a)**

**Mealy machine:**
- **BCD to BCD+3 Converter**

- **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 RTL of SELECTOR is**
  - **begin**
    - **Y <= A(conv_integer(SEL));**
  - **end RTL;**

**Synthesis Examples (2b)**

**Mealy machine:**
- **BCD to BCD+3 Converter**

- **library ieee;**
- **use ieee.std_logic_1164.all;**
- **use ieee.std_logic_unsigned.all;**

- **entity example is**
  - **port (X: in std_logic_vector(15 downto 0);**
  - **Y: out std_logic_vector(15 downto 0);**

- **end example:**
  - **architecture hv of example is**
    - **begin**
      - **when X = 0 ->**
        - **Y <= X + 1;**
      - **when others ->**
        - **Y <= X;**
    - **end process;**

**Writing Test Benches**

**MUX 16 to 1**
- **16 data inputs**
- **4 selection inputs**

- **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 hv of SELECTOR is**
  - **begin**
    - **Y <= A(conv_integer(SEL));**
  - **end hv;**
Assert Statement

- Checks to see if a certain condition is true, and if not causes an error message to be displayed
  
  ```vhdl
  assert boolean-expression
  report string-expression
  severity severity-level;
  ```

- Four possible severity levels
  - NOTE
  - WARNING
  - ERROR
  - FAILURE

- Action taken for a severity level depends on the simulator

Writing Test Benches

begin
  process
  variable cnt : std_logic_vector(4 downto 0);
  begin
    for j in 0 to 31 loop
      cnt := conv_std_logic_vector(j, 5);
      TSEL <= cnt(3 downto 0);
      Y <= cnt(4);
      A <= (A'range => not cnt(4));
      A(conv_integer(cnt(3 downto 0))) <= cnt(4);
      wait for PERIOD;
    end loop;
    wait;
  end process;
begin
  process
  variable err_cnt : integer := 0;
  begin
    wait for STROBE;
    for j in 0 to 31 loop
      assert FALSE report "comparing" severity NOTE;
      if (Y /= TY) then
        assert FALSE report "not compared" severity WARNING;
        err_cnt := err_cnt + 1;
      end if;
      wait for PERIOD;
    end loop;
    assert (err_cnt = 0) report "test failed" severity ERROR;
    assert (err_cnt /= 0) report "test passed" severity NOTE;
    wait;
  end process;
sel: SELECTOR port map (A => TA, SEL = TSEL, Y => TY);
end process;
end BEH;

Things to Remember

- Attributes associated to signals
  - allow checking for setup, hold times, and other timing specifications

- Attributes associated to arrays
  - allow us to write procedures that do not depend on the manner in which arrays are indexed

- Inertial and transport delays
  - allow modeling of different delay types that occur in real systems

- Operator overloading
  - allow us to extend the definition of VHDL operators so that they can be used with different types of operands

Things to Remember (cont’d)

- Multivalued logic and the associated resolution functions
  - allow us to model tri-state buses, and systems where a signal is driven by more than one source

- Generics
  - allow us to specify parameter values for a component when the component is instantiated

- Generate statements
  - efficient way to describe systems with iterative structure

- TEXTIO
  - convenient way for file input/output