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 clock cycle.</td>
</tr>
<tr>
<td>S/ACTIVE</td>
<td>True if a transaction occurred during the current clock cycle.</td>
</tr>
<tr>
<td>S/LAST_EVENT</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
Signal Attributes (cont’d)

entity test is
end;
architecture bmtest of test is
signal A : bit;
signal B : bit;
signal C : bit;
begin
A <= not A after 20 ns;
B <= '1';
C <= A and B;
process(A, B, C)
variable Aev : bit;
variable Aac : bit;
variable Bev : bit;
variable Bac : bit;
variable Cev : bit;
variable Cac : bit;
begin
if (A'event) then Aev := '1';
else Aev := '0';
end if;
if (A'active) then Aac := '1';
else Aac := '0';
end if;
if (B'event) then Bev := '1';
else Bev := '0';
end if;
if (B'active) then Bac := '1';
else Bac := '0';
end if;
if (C'event) then Cev := '1';
else Cev := '0';
end if;
if (C'active) then Cac := '1';
else Cac := '0';
end if;
end process;
end bmtest;

Examples of Signal Attributes

Ack: entity Ack is
port(I, O : in bit);
end Ack;
architecture test of Ack is
begin
A <= I;
end test;

Attributes that create a signal

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Creates</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDELAYED (time)*</td>
<td>signal stays S delayed to specified time</td>
</tr>
<tr>
<td>SYSTABLE ([true])*</td>
<td>boolean signal that is true if S had no events for the specified time</td>
</tr>
<tr>
<td>SQUIET ([true])*</td>
<td>boolean signal that is true if S had no transactions for the specified time</td>
</tr>
<tr>
<td>STRANSACTION</td>
<td>Signal of new SST that changes for every transaction on S</td>
</tr>
</tbody>
</table>

* Beta is used if no time is specified

Using Attributes for Error Checking

check: process
begin
wait until rising_edge(Clk);
assert (D'stable(setup_time))
report("Setup time violation")
severity error;
wait for hold_time;
assert (D'stable(hold_time))
report("Hold time violation")
severity error;
end process check;
Assert Statement

assert boolean-expression
report string-expression
severity severity-level

- If boolean expression is false
display the string expression on the monitor
- Severity levels: Note, Warning, Error, Failure

Array Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Return</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALEFT(N)</td>
<td>left bound of ith index range</td>
<td>ROM11LEFT(1) = 0</td>
</tr>
<tr>
<td>ARIGHT(N)</td>
<td>right bound of ith index range</td>
<td>ROM11RIGHT(1) = 7</td>
</tr>
<tr>
<td>AIHIGH(N)</td>
<td>largest bound of ith index range</td>
<td>ROM11IHIGH(1) = 15</td>
</tr>
<tr>
<td>AILOW(N)</td>
<td>smallest bound of ith index range</td>
<td>ROM11ILOW(1) = 0</td>
</tr>
<tr>
<td>AIWIDE(N)</td>
<td>width of ith index range</td>
<td>ROM11IWIDTH(1) = 8</td>
</tr>
<tr>
<td>AILEFT(RANGE)</td>
<td>left bound of ith index range</td>
<td>ROM11ILEFT(1) = 0</td>
</tr>
<tr>
<td>AIRIGHT(RANGE)</td>
<td>right bound of ith index range</td>
<td>ROM11IRIGHT(1) = 7</td>
</tr>
<tr>
<td>AIHIGH(RANGE)</td>
<td>largest bound of ith index range</td>
<td>ROM11IHIGH(1) = 15</td>
</tr>
<tr>
<td>AILOW(RANGE)</td>
<td>smallest bound of ith index range</td>
<td>ROM11ILOW(1) = 0</td>
</tr>
<tr>
<td>AIWIDE(RANGE)</td>
<td>width of ith index range</td>
<td>ROM11IWIDTH(1) = 8</td>
</tr>
<tr>
<td>ALength</td>
<td>size of ith index range</td>
<td>ROM11ILNGTH(1) = 16</td>
</tr>
</tbody>
</table>

A can be either an array name or an array type.
Array attributes work with signals, variables, and constants.

Recap: Adding Vectors

- This procedure adds two n-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 Add1:
    (Add1: in bit_vector;
     C: in bit;
     signal Sum: out bit_vector;
     signal Carry: out bit;
    variable C: bit;
    begin
        C := C;
        for i in 1 to n 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;
        C := C;
    end Add1;
Note: Add1 and Add2 vectors must be dimensioned as N-1 downto 0. Use attributes to write more general procedure that places no restrictions on the range of vectors other than the lengths must be same.

Procedure for Adding Bit Vectors

- This procedure adds two bit vectors and a carry and returns a sum and a carry. Both bit_vectors should be of the same length.

procedure Add2:
    (Add1, Add2: in bit_vector;
     C: in bit;
     signal Sum: out bit_vector;
    variable C: bit;
    begin
        for i in 1 to n 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;
        C := C;
    end Add2;
Note: Add1 and Add2 vectors must be dimensioned as N-1 downto 0. Use attributes to write more general procedure that places no restrictions on the range of vectors other than the lengths must be same. 

- Severity error;
- For all i in 1 to n loop
  C := (Sum(i) and Add1(i)) or (Sum(i) and C) or (Add2(i) and C);
  C := C;
end Add2;
Transport and Inertial Delay

Reject is equivalent to a combination of inertial and transport delay:

\[ Z_3 \leq \text{reject 4 ns } X \text{ after 10 ns}; \]
\[ Z_2 \leq \text{transport } Z_3 \text{ after 6 ns}; \]

Statements executed at time \( T \):

- \( B \) at \( T+1 \), \( C \) at \( T+2 \):
  - \( A \) <= transport \( B \) after 1 ns;
  - \( A \) <= transport \( C \) after 2 ns;

Statements executed at time \( T-1 \):

- \( A \) <= transport \( B \) after 2 ns;
- \( A \) <= transport \( C \) after 1 ns;

Transport and Inertial Delay (cont'd)

\[ Z_2 \leq \text{transport } Z_1 \text{ after 10 ns}; \]

Statements executed at time \( T \):

- \( X \) at \( T+4 \):
  - \( A \) <= transport \( X \) after 6 ns;

\[ Z_1 \leq \text{reject 4 ns } X \text{ after 10 ns}; \]

Operator Overloading

- Operators +, - operate on integers
- Write procedures for bit vector addition/subtraction – addvec, 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_overload is
  function add (*: bit_VECTOR; add: bit_VECTOR) return bit_VECTOR;
end bit_overload;

element + bit_pkg.all;
package body bit_overload is
  function add (*: bit_VECTOR; add: bit_VECTOR) return bit_VECTOR is
    -- The add is performed bit by bit with an internal carry
    function add (*: bit_VECTOR; add: bit_VECTOR) return bit_VECTOR is
      local carry: bit := '0';
      local result: bit_VECTOR := (others => '0');
      for i in result'range loop
        result(i) := (add(i) + carry) mod 2;
        carry := (add(i) + carry) >= 2;
      end loop;
    return result;
  end function;
begin
  -- This function returns a bit_vector sum of a bit_vector and an integer
  -- Using the previous function alternate range is converted
  function add (*: bit_VECTOR; add: bit_VECTOR) return bit_VECTOR is
    local result: bit_VECTOR := (others => '0');
    local carry: bit := '0';
    for i in result'range loop
      result(i) := (add(i) + carry) mod 2;
      carry := (add(i) + carry) >= 2;
    end loop;
    return result;
  end function;
end bit_overload;
Overloaded Operators

- A, B, C – bit vectors
- A <= B + C + 3 ?
- A <= 3 + B + C ?

- Overloading can also be applied to procedures and functions
  - procedures have the same name
  - type of the actual parameters in the procedure call determines which version of the procedure is called

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 '1' after 2 ns, 'Z' after 6 ns;
R <= transport '1' after 8 ns, '0' after 10 ns;

Resolution Function for X01Z

Define AND and OR for 4-valued inputs?

Resolution Function using X01Z

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>W</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

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.

In this course we use a subset of the IEEE values: X10Z
Resolution Function for IEEE 9-valued

\[
\text{function } \text{and'}(\text{a}, \text{b}) : \text{std_logic} \text{ return} \text{std_logic} \text{ is}
\begin{align*}
\text{begin} \\
\text{return}(\text{a'} \text{ or } \text{b'})\};
\text{end } \text{and'};
\end{align*}
\]

AND Function for std_logic_vectors

\[
\text{function } \text{and}(\text{a}, \text{b}) : \text{std_logic_vector} \text{ return} \text{std_logic_vector} \text{ is}
\begin{align*}
\text{begin} \\
\text{result} := \text{a} \text{ and } \text{b}; \\
\text{end } \text{and};
\end{align*}
\]

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
Rise/Fall Time Modeling Using Generics

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

architecture Structure of Adder4 is
component FullAdder
  port (A, B, Cin: in bit; 
  Cout, Sum, out bit); 
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(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;

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 Adder is
  port (A, B: in bit_vector(3 downto 0); 
  S, C: out bit_vector(3 downto 0); 
  Co, out bit); 
end Adder;

architecture Structure of Adder is
component FullAdder
  port (A, B, Cin: in bit; 
  Cout, Sum, out bit); 
end component;

signal C, bit_vector(4 downto 0); 
bEGIN 
C(0) = Ci; 
-- generate four copies of the FullAdder
FullAdder: for i in 0 to 3 generate 
FullAdder port map (A(i), B(i), C(i), C(i+1), S(i)); 
end generate FullAdder;
C(0) := C(3); 
end Structure;
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

---

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

What if \( a = 3 \)?

The previous value of \( b \) should be held in the latch, so \( G \) should be 0 when \( a = 3 \).

---

If Statements

```vhdl```
```java
if A = '1' then NextState <= 3;
end if;
```

What if \( A \neq 1 \)?

Retain the previous value for **NextState**?

Synthesizer might interpret this to mean that **NextState** is unknown!

```vhdl```
```java
if A = '1' then NextState <= 3;
else NextState <= 2;
end if;
```
Synthesis of a Case Statement

```vhdl
entity case_example is
    port (a: bit; b, c: bit_vector; z: bit_vector);
end case_example;
architecture test of case_example is
begin
    case a is
    when 1 => z <= b + c;
    when 2 => z <= b - c;
    when 3 => z <= b * c;
    end case;
end test;
```

Case Statement: Before and After Optimization

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 include overloaded versions of arithmetic, relational, logical, and shifting operations, and conversion functions:
    - numeric_std package defines similar operations on std_logic_vectors.
• 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

If the left and right operand values 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 0s on the left. Examples:

```
signed 0110111 + 1011 => becomes 0110111 + 0100000
```

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:

```