CDA 3200 Digital Systems

Instructor: Dr. Janusz Zalewski
Developed by: Dr. Dahai Guo
Spring 2012
Outline

• Overview of VHDL
• VHDL Description of Combinational Circuits
• VHDL Models for Multiplexers
• VHDL Modules
• Signals and Constants
• Arrays
• VHDL Operators
• Packages and Libraries
Overview of VHDL (1/6)

- VHDL
  - VHSIC Hardware Description Language
  - VHSIC: Very High Speed Integrated Circuit
  - Widely used in the industry
Overview of VHDL (2/6)

• Why VHDL?
  – Detailed design of the systems at the gate level has become very tedious and time consuming.
  – A hardware description language of top-down methodology is desired.
Overview of VHDL (3/6)

- The program that designs a 3-bit adder
  1. entity ADD is
  2. port (A, B : in integer range 0 to 7;
  3. Z : out integer range 0 to 15);
  4. end ADD;

  5. architecture ARITHMETIC of ADD is
  6. begin
  7. Z <= A + B;
  8. end ARITHMETIC;
Overview of VHDL (4/6)

• It seems that Boolean algebra, Karnaugh-Maps, and other techniques (Quine-McCluskey and Petrick’s methods) are useless, if VHDL is so powerful.
Where Boolean algebra, Karnaugh maps and other methods are integrated.
Overview of VHDL (6/6)

• Because the simplifying methods are systematic, they can be programmed into the synthesizer.
• Therefore, hardware designers can be freed of detailed work at the gate level.
• But the designers still can work at the gate level in VHDL.
VHDL Description of Combinational Circuits (1/6)

• Signal assignment operator
  1. \text{C} \leftarrow \text{A and B after 5 ns};
  2. \text{E} \leftarrow \text{C or D after 5 ns};

• Logically equivalent?
  1. \text{E} \leftarrow \text{C or D after 5 ns};
  2. \text{C} \leftarrow \text{A and B after 5 ns};
VHDL Description of Combinational Circuits (2/6)

- VHDL signal assignment statements are concurrent statements.
  - The VHDL simulator monitors the right side of each concurrent statement;
  - Any time a signal changes, the expression on the right side is immediately re-evaluated;
  - The new value is assigned to the signal on the left side after the delay.
  - The order of the concurrent statements does not matter.
The delay is optional in assignment statements.

- \( E \leq C \) or \( D \);
- \( C \leq A \) and \( B \);
- When no delay is specified, the simulator will assume an infinitesimal delay referred to as \( \Delta \) (delta).
VHDL Description of Combinational Circuits (4/6)

• Clock
  – CLK <= not CLK after 10 ns;

Can we remove the delay?

No, because a runtime error during simulation will be caused.
VHDL Description of Combinational Circuits (5/6)

• Bit vectors
  - B <= “0110”;
  - A <= “1100”;
  - C <= A and B;  \(\iff\)  C(3) <= A(3) and B(3);
    C(2) <= A(2) and B(2);
    C(1) <= A(1) and B(1);
    C(0) <= A(0) and B(0);
VHDL Description of Combinational Circuits (6/6)

- Comments are preceded by --
- An identifier, such as a variable, must start with a letter, and it cannot end with an underscore.
- Each statement is ended with a semicolon.
- NOT CASE SENSITIVE
VHDL Models for Multiplexers (1/6)

• 2-to-1 multiplexer
  – \( F \leq (\text{not } A \text{ and } I_0) \text{ or } (A \text{ and } I_1); \)
VHDL Models for Multiplexers (2/6)

- 2-to-1 (cont)
  1. -- conditional signal assignment statement
  2. F <= I0 when A='0' else I1;
VHDL Models for Multiplexers (3/6)

Design at the gate level.

Design at the logic level.
VHDL Models for Multiplexers (4/6)

- The general form of a conditional signal assignment statement is
  1. `signal_name <= expression1 when condition1`
  2. `else expression2 when condition2`
  3. ……
  4. `[else expressionN];`
- This concurrent statement is executed whenever a change occurs in a signal used in the selected expression or its condition.
VHDL Models for Multiplexers (5/6)

• 4-to-1

\[ F \leq I_0 \text{ when } A \& B = \text{"00"} \]
\[ \quad \text{else } I_1 \text{ when } A \& B = \text{"01"} \]
\[ \quad \text{else } I_2 \text{ when } A \& B = \text{"10"} \]
\[ \quad \text{else } I_3; \]
VHDL Models for Multiplexers (6/6)

• 4-to-1

```
sel <= A&B;
with sel select
  F <= I0 when "00",
      I1 when "01",
      I2 when "10",
      I3 when "11";
```
VHDL Modules (1/17)

• Entities and architectures
  – An entity declares the interface between a module and its environment. What are the input(s) and output(s)?
  – The architecture contains the implementation for an entity.
VHDL Modules (2/17)

• An entity does not deal with design details.
  – Example:
  – `entity two_gates is`
  – `port (A, B, D: in bit; E: out bit);`
  – `end two_gates;`

• Key words that are used to define an entity.
• The name of the entity.
• The input and output list. *port* is a key word that is used to declare input/output.
• A, B, D are the input signals and E is the output signal.
• Be careful where you need semicolons, commas, and colons.
VHDL Modules (3/17)

- Associated with each entity is one or more architecture declarations
  - `architecture gates of two_gates is`
  - `signal C: bit;`
  - `begin`
  - `C <= A and B; -- concurrent`
  - `E <= C or D; -- statements`
  - `end gates;`

Key words used to define an architecture.
- Every architecture must be associated with an entity.
- The architecture’s name.
- Signal used within this architecture.
- Design details
VHDL Modules (4/17)

• Full adder
  – entity FullAdder is
  –   port(X, Y, Cin: in bit; -- inputs
  –     Cout, Sum: out bit); -- outputs
  – end FullAdder;
VHDL Modules (5/17)

• Full adder
  – architecture Equations of FullAdder is
  – begin
  – 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;

VHDL does not specify an order of precedence for the logic operators, therefore parentheses are required.
VHDL Modules (6/17)

- 4-bit fuller adder
1. entity Adder4 is
2. port (A, B: in bit_vector(3 downto 0); Ci: in bit -- Inputs
3. S: out bit_vector(3 downto 0); Co: out bit) -- Outputs
4. Adder4
5. architecture Structure of adder4 is
6. begin

1. S(0) <= A(0) xor B(0) xor Ci(0) after 10 ns;
2. Co(0) <= (A(0) and B(0)) or (A(0) and Ci(0)) or (B(0) and Ci(0)) after 10 ns;
   \textbf{Find all the syntax errors in lines 1-6 and 15.}
3. S(1) <= A(1) xor B(1) xor Ci(1) after 10 ns;
4. Co(1) <= (A(1) and B(1)) or (A(1) and Ci(1)) or (B(1) and Ci(1)) after 10 ns;
5. S(2) <= A(2) xor B(2) xor Ci(2) after 10 ns;
6. Co(2) <= (A(2) and B(2)) or (A(2) and Ci(2)) or (B(2) and Ci(2)) after 10 ns;
7. S(3) <= A(3) xor B(3) xor Ci(3) after 10 ns;
8. Co(3) <= (A(3) and B(3)) or (A(3) and Ci(3)) or (B(3) and Ci(3)) after 10 ns;
9. Structure
1. entity Adder4 is
2. port (A, B: in bit_vector(3 downto 0); Ci: in bit; -- Inputs
3. S: out bit_vector(3 downto 0); Co: out bit); -- Outputs
4. end Adder4;
5. architecture Structure of adder4 is
6. begin
7. S(0) <= A(0) xor B(0) xor Ci(0) after 10 ns;
8. Co(0) <= (A(0) and B(0)) or (A(0) and Ci(0)) or (B(0) and Ci(0)) after 10 ns;
9. S(1) <= A(1) xor B(1) xor Ci(1) after 10 ns;
10. Co(1) <= (A(1) and B(1)) or (A(1) and Ci(1)) or (B(1) and Ci(1)) after 10 ns;
11. S(2) <= A(2) xor B(2) xor Ci(2) after 10 ns;
12. Co(2) <= (A(2) and B(2)) or (A(2) and Ci(2)) or (B(2) and Ci(2)) after 10 ns;
13. S(3) <= A(3) xor B(3) xor Ci(3) after 10 ns;
14. Co(3) <= (A(3) and B(3)) or (A(3) and Ci(3)) or (B(3) and Ci(3)) after 10 ns;
15. end Structure;
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
begin
  \begin{align*}
    S(0) &= A(0) \text{xor} B(0) \text{xor} Ci(0) \text{ after } 10 \text{ ns;} \\
    Co(0) &= (A(0) \text{ and } B(0)) \text{ or } (A(0) \text{ and } Ci(0)) \text{ or } (B(0) \text{ and } Ci(0)) \text{ after } 10 \text{ ns;} \\
    S(1) &= A(1) \text{xor} B(1) \text{xor} Ci(1) \text{ after } 10 \text{ ns;} \\
    Co(1) &= (A(1) \text{ and } B(1)) \text{ or } (A(1) \text{ and } Ci(1)) \text{ or } (B(1) \text{ and } Ci(1)) \text{ after } 10 \text{ ns;} \\
    S(2) &= A(2) \text{xor} B(2) \text{xor} Ci(2) \text{ after } 10 \text{ ns;} \\
    Co(2) &= (A(2) \text{ and } B(2)) \text{ or } (A(2) \text{ and } Ci(2)) \text{ or } (B(2) \text{ and } Ci(2)) \text{ after } 10 \text{ ns;} \\
    S(3) &= A(3) \text{xor} B(3) \text{xor} Ci(3) \text{ after } 10 \text{ ns;} \\
    Co(3) &= (A(3) \text{ and } B(3)) \text{ or } (A(3) \text{ and } Ci(3)) \text{ or } (B(3) \text{ and } Ci(3)) \text{ after } 10 \text{ ns;}
  \end{align*}
end Structure;
VHDL Modules (10/17)

- 4-bit full adder
  1. $S(0) \leq A(0) \oplus B(0) \oplus Ci(0)$ after 10 ns;
  2. $Co(0) \leq (A(0) \land B(0)) \lor (A(0) \land Ci(0)) \lor (B(0) \land Ci(0))$ after 10 ns;
    - $Ci$ and $Co$ are defined as bit signals and cannot work with indices.
VHDL Modules (11/17)

• 4-bit full adder
  – architecture Structure of Adder4 is
  – signal C: bit_vector(2 downto 0);
  – begin
  – end Structure;
VHDL Modules (12/17)

- $S(0) \leq A(0) \text{ xor } B(0) \text{ xor } C_i$ after 10 ns;
- $C(0) \leq (A(0) \text{ and } B(0)) \text{ or } (A(0) \text{ and } C_i) \text{ or } (B(0) \text{ and } C_i)$ after 10 ns;

- $S(1) \leq A(1) \text{ xor } B(1) \text{ xor } C(0)$ after 10 ns;
- $C(1) \leq (A(1) \text{ and } B(1)) \text{ or } (A(1) \text{ and } C(0)) \text{ or } (B(1) \text{ and } C(0))$ after 10 ns;

- $S(2) \leq A(2) \text{ xor } B(2) \text{ xor } C(1)$ after 10 ns;
- $C(2) \leq (A(2) \text{ and } B(2)) \text{ or } (A(2) \text{ and } C(1)) \text{ or } (B(2) \text{ and } C(1))$ after 10 ns;

- $S(3) \leq A(3) \text{ xor } B(3) \text{ xor } C(2)$ after 10 ns;
- $C_o \leq (A(3) \text{ and } B(3)) \text{ or } (A(3) \text{ and } C(2)) \text{ or } (B(3) \text{ and } C(2))$ after 10 ns;
VHDL Modules (13/17)

• 4-bit full adder
  – Simulation commands
    • 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         -- set Ci to 1
    • run 50 ns          -- run the simulation for 50 ns
VHDL Modules (14/17)

- 4-bit full adder
  - Simulation output

<table>
<thead>
<tr>
<th>Current</th>
<th>0ns</th>
<th>10ns</th>
<th>20ns</th>
<th>30ns</th>
<th>40ns</th>
<th>50ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1111</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0001</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ci</td>
<td>'1'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>0001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Co</td>
<td>'1'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
VHDL Modules (15/17)

• Since a 4-bit full adder is a sequence of 1-bit full adders, why do not we build the 4-bit adder based on 1-bit adders in VHDL?
  – -- specify the FullAdder as a component
  – -- within the architecture of Adder4
  – architecture Structure of Adder4 is
    – component FullAdder
      – port (X, Y, Cin: in bit; -- Inputs
        – Cout, Sum: out bit); -- Outputs
    – end component;
  – -- other code
  – end Structure;
VHDL Modules (16/17)

- other code
- signal C: bit_vector(3 downto 1);
- begin
  -- instantiate four copies of the FullAdder
  FA0: FullAdder port map (A(0), B(0), Ci, C(0), S(0));
  FA1: FullAdder port map (A(1), B(1), C(0), C(1), S(1));
  FA2: FullAdder port map (A(2), B(2), C(1), C(2), S(2));
  FA3: FullAdder port map (A(3), B(3), C(2), Co, S(3));
VHDL Modules (17/17)

• Component instantiation statements
  – label: component-name *port map* (list-of-actual-signals)
4-bit Full Adder Simulation (1/7)
4-bit Full Adder Simulation (2/7)
Full Adder

Co

Full Adder

C2

Full Adder

C1

Full Adder

C0

Full Adder

Ci

A3 1
B3 0

A2 1
B2 0

A1 1
B1 0

A0 1
B0 1

S3 1
S2 1
S1 1
S0 1

C2 C1 C0
S3 S2 S1 S0
Co

0ns 0 0 0 0 0 0 0
10ns 0 0 1(1) 1 1 1 1(F) 0

<table>
<thead>
<tr>
<th>Current</th>
<th>0ns</th>
<th>10ns</th>
<th>20ns</th>
<th>30ns</th>
<th>40ns</th>
<th>50ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>A 1111</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B 0001</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ci '1'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S 1111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1(F)</td>
</tr>
<tr>
<td>Co '0'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(F)</td>
</tr>
<tr>
<td>C 001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

4-bit Full Adder Simulation (3/8)
Nothing changes to the inputs of Full Adders 0,2,3. Only need to look at Full Adder 1.

<table>
<thead>
<tr>
<th>Current</th>
<th>0ns</th>
<th>10ns</th>
<th>20ns</th>
<th>30ns</th>
<th>40ns</th>
<th>50ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ci</td>
<td>'1'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>1101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Co</td>
<td>'0'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4-bit Full Adder Simulation (4/8)
(1) (F)
(2) 0ns
(3) (D)
(4) 20ns
(5) (7)
(6) (9)

Nothing changes to the inputs of Full Adders 0,1,3. Only need to look at Full Adder 2.

4-bit Full Adder Simulation (5/8)
Nothing changes to the inputs of Full Adders 0,1,2. Only need to look at Full Adder 3.

<table>
<thead>
<tr>
<th>Current</th>
<th>0ns</th>
<th>10ns</th>
<th>20ns</th>
<th>30ns</th>
<th>40ns</th>
<th>50ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>A 1111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B 0001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ci '1'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S 0001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Co '1'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C 111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4-bit Full Adder Simulation (6/8)
The circuit is steady. Nothing changes from 40-50 ns.
Signals and Constants (1/5)

• Signals
  – Interface: input/output
    • entity FullAdder is
    •   port(X, Y, Cin: in bit; -- inputs
    •   Cout, Sum: out bit); -- outputs
    • end FullAdder;
  – Internal signals
    • architecture gates of two_gates is
    •   signal C: bit;
    • begin
    •   C <= A and B; -- concurrent
    •   E <= C or D; -- statements
    • end gates;
Signals and Constants (2/5)

• The predefined VHDL types
  – bit: ‘0’ or ‘1’
  – boolean: FALSE or TRUE
  – integer
  – positive
  – natural
  – real
  – character
  – time
Signals and Constants (2/5) (cont)

- Types default value
- bit ‘0’
- Boolean false
- integer not defined
- positive 1
- natural 0
- Real 0.000
- character not defined
- Time 0
Signals and Constants (3/5)

- Signal initialization
  - signal A: bit := ‘1’; -- A <= ‘1’
  - signal A: bit; -- A <= ‘0’. ‘0’ is the default value
  - signal A, B, C: bit = ‘1’; -- anything wrong?
  - signal A, B, C: bit := ‘1’; -- what are A and B?
  - -- all A, B, and C will be
  - -- initialized to ‘1’
Signals and Constants (4/5)

• Constants:
  – \texttt{constant} limit:integer := 17;
  – \texttt{constant} delay1:time := 5 ns;
  – \texttt{A} <= \texttt{B} after \texttt{delay1};
Signals and Constants (5/5)

• Enumeration data type
  – Defined by users
  – `type state_type is (S0,S1,S2,S3,S4,S5);`
  – `signal state : state_type := S1;`
  – `state <= s3;`
Arrays (1/11)

• Differences with arrays in Java/C++
  – Index values can be customized.
  – Must create an array type before declaring an array object.
Arrays (2/11)

• Differences with arrays in Java/C++ (cont)
  – Correct declaration
    • type SHORT_WORD is array (15 downto 0) of bit;
    • signal DATA_WORD : SHORT_WORD;
  – Wrong declaration
    • signal DATA_WORD : array (15 downto 0) of bit;
Arrays (3/11)

• Differences with arrays in Java/C++ (cont)
  – type SHORT_WORD is array (15 downto 0) of bit;
  – signal DATA_WORD : SHORT_WORD; -- default value “00…0”
  – signal ALT_WORD : SHORT_WORD := “0101010101010101”;

“The initializer for an array of bits must be double quoted.

ALT_WORD(15)

ALT_WORD(0)
Arrays (4/11)

• Differences with arrays in Java/C++ (cont)
  – A portion of the array can be accessed in one statement
  – ALT_WORD(5 downto 0) <= "111111";
Arrays (5/11)

- Differences with arrays in Java/C++ (cont)
  - The slice direction must be consistent.
    - downto
    - to

- type SHORT_WORD is array (0 to 15) of bit;
- signal DATA_WORD : SHORT_WORD;
- DATA_WORD (5 downto 0) <= "111111";

Wrong slice direction.
Arrays (6/11)

- Differences with arrays in Java/C++ (cont)
  - Unconstrained array types
  - type intvec is array (natural range <>) of integer;
  - type matrix is array (natural range <> , natural range <> ) of integer;
Arrays (7/11)

• Predefined unconstrained array types in VHDL
  – type bit_vector is array (natural range<> ) of bit;
  – type String is array (positive range <> ) of character;
Arrays (8/11)

• Predefined unconstrained array types in VHDL
  – When declaring an array signal, the range must be specified
  – signal A: bit_vector(0 to 5) := "101011";
  – constant string1:string(1 to 29)="……";
Example 1: design a ROM as illustrated in Figure 9-17.
Arrays (10/11)

- entity ROM9_17
  - port (A,B,C: in bit; F: out bit_vector(0 to 3));
- end entity;

- or

- entity ROM9_17
  - port (A: in bit_vector(0 to 2); F: out bit_vector(0 to 3));
- end entity;
Arrays (11/11)

- architecture ROM of ROM9_17 is
- type ROM8X4 is array (0 to 7) of bit_vector (0 to 3);
- signal index: integer range 0 to 7;
- begin
-  index <= vec2int(A&B&C); --A&B&C is a 3-bit vector
-  -- vec2int is a function that converts this vector to integer
-  F<=ROM1(index);
-  end ROM;
VHDL Operators (1/2)

- Predefined VHDL operators
  1. and or nand nor xor xnor
  2. = /= < <= > >=
  3. sll srl sla sra rol ror
  4. + - &
  5. * / mod rem
  6. not abs **

Precedence
VHDL Operators (2/2)

- Predefined VHDL operators
  1. and or nand nor xor xnor
  2. = /= < <= > >=
  3. sll srl sla sra rol ror
  4. + - &
  5. * / mod rem
  6. not abs **
Packages and Libraries (1/2)

- Packages and libraries provide a convenient way of referencing frequently used functions and components.
- In this class, some of your programs may need package *bit_pack* in library *BITLIB*. 
Packages and Libraries (2/2)

• The syntax is
  – library BITLIB;
  – use BITLIB.bit_pack.all;

• They are needed before each entity that uses library functions.