Outline

• What we know
  – How to model Combinational Networks in VHDL
    • Structural, Dataflow, Behavioral
  – How to model Flip-flops in VHDL
  – Processes
  – Delays (delta, transport, inertial)

• What we do not know
  – How to model FSM in VHDL
  – Wait statements
  – Variables, Signals, Arrays
  – VHDL Operators
  – Procedures, Functions
  – Packages, Libraries
  – Additional Topics (if time)

Review: VHDL Program Structure

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;

Review: JK Flip-Flop Model

Note 1: J is defined as input (rather than output because it appears on both the left and right sides of an assignment within a process.

Note 2: The wire delay (J → Q) is fixed and (Q → J) has just changed from 0 to 1.

Note 3: The characteristic equation which describes the behavior of a flip-flop.

Note 4: Everyone (J changes, Q will not change). If the statement were placed within the process, the old Q would be used. The || symbol represents the OR gate.

Review: VHDL Models for a MUX

MUX model using a conditional signal assignment statement:

\[ F = \begin{cases} 
1 & \text{if } \text{Sel} = 0 \\
\text{not } F & \text{if } \text{Sel} = 1 \\
\text{else if } \text{Sel} = 2 & \\
\text{else } & \text{end case,}
\end{cases} \]

If a MUX model is used inside a process, the MUX can be modeled using a CASE statement (cannot use a concurrent statement):

case Sel is
  when 0 => F := 1;
  when 1 => F := \text{not } F;
  when 2 => F := X;
  when others => F := \text{other operator statement(s)}
end case;

Timing Model

VHDL uses the following simulation cycle to model the stimulus and response nature of digital hardware:

Start Simulation

Delay

Update Signals

Execute Processes

End Simulation
Review: 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

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 300
force X 0.0, 1.350, 0.550, 1.750, 0.950, 1.1350, run 1800
```

Wavforms:

Dataflow VHDL Model

```vhdl
entity Dataflow is
  port (A, B: in bit; C, D: out bit);
end Dataflow;
architecture Behavior of Dataflow is
begin
  process (CLK)
  begin
    if rising_edge (CLK) then
      C <= A and B;
      D <= (not A) or B;
    end if;
  end process;
end Behavior;
```
**Structural Model**

Package `bit_pack` is a part of library BITLIB – includes gates, flip-flops, counters

(See Appendix B for details)

**Simulation of the Structural Model**

Simulation command file:

```plaintext
wave CLK X Q1 Q2 Q3 Z
force CLK 0 0 1 180 -repeat 200
force X 0 0 1 350 0 530 1 750 0 951 1 1350
run 1600
```

Wavetables:

![Wavetables](image)

**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)**

```plaintext
process
  begin
    sequential-statements
    wait statement
    sequential-statements
    wait-statement
    ...
  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
  - 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 (1)**

```
library BITLIB;
use BITLIB.bit_pack.all;
entity STRM_2 is
  port (CLK, X: in bit;
        Q1, Q2, Q3, Z: out bit);
end entity;
architecture Structural of STRM_2 is
  begin
    process
      begin
        case X is
          when '0' =>
            if Z = '1' then
              if Y = '1' then
                if X = '1' then
                  if Y = '1' then
                    WAIT_STATE (wait_state); end if;
                else
                  WAIT_STATE (wait_state); end if;
              else
                WAIT_STATE (wait_state); end if;
            end if;
          when '1' =>
            if Z = '1' then
              if Y = '1' then
                if X = '1' then
                  if Y = '1' then
                    WAIT_STATE (wait_state); end if;
                else
                  WAIT_STATE (wait_state); end if;
              else
                WAIT_STATE (wait_state); end if;
            end if;
        end case;
      end begin
    end process;
  end architecture;
end STRM_2;
```

**Using Wait Statements (2)**

```
library BITLIB;
use BITLIB.bit_pack.all;
entity STRM_2 is
  port (CLK: in bit;
        X, Y, Z: in bit;
        Q1, Q2, Q3: out bit);
end entity;
architecture Structural of STRM_2 is
  begin
    process
      begin
        case X is
          when '0' =>
            if Y = '1' then
              if Z = '1' then
                if Y = '1' then
                  if X = '1' then
                    WAIT_STATE (wait_state); end if;
                else
                  WAIT_STATE (wait_state); end if;
              else
                WAIT_STATE (wait_state); end if;
            end if;
          when '1' =>
            if Y = '1' then
              if Z = '1' then
                if Y = '1' then
                  if X = '1' then
                    WAIT_STATE (wait_state); end if;
                else
                  WAIT_STATE (wait_state); end if;
              else
                WAIT_STATE (wait_state); end if;
            end if;
        end case;
      end begin
    end process;
  end architecture;
end STRM_2;
```
Variables

• What are they 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;
  ```

• 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

• Variable assignment statement
  ```vhdl
  variable_name := expression;
  ```

• Signal assignment statement
  ```vhdl
  signal_name <= expression [after delay];
  ```

  – expression is evaluated and the variable is instantaneously updated
  (no delay, not even delta 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 vs. Signals

• 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, uppercase letters, digits, special characters, ...
  – time – an integer with units fs, ps, ns, us, ms, sec, min, or hr

Predefined VHDL Types
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 : 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

```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));
```

- Unconstrained array type

```vhdl
type intvec is array (natural range<>) of integer;
variable intvec5 : intvec(1 to 5) := (3,2,6,8,1);
```

Predefined Unconstrained Array Types

- Bit_vector, string

```vhdl
type bit_vector is array (natural range<>) of bit;
type string is array (positive range<>) of character;
constant str5 : string := "This string is 5 characters.";
```

- Subtypes

```vhdl
subtype SHORT_WORD is bit_vector(5 to 0);
```

- POSITIVE, NATURAL – predefined subtypes of type integer

Sequential Machine Model Using State Table

```vhdl
entity seq is
port (clk, reset : in bit);
end seq;
architecture StateMachine of seq is
begin
process
begin
wait until rising_edge(clk);
ReportErr <= StateMachine.X; -- read next state from state table;
nextState <= StateMachine.Y; -- read output from output table;
report (nextState);
end process;
end StateMachine;
```

VHDL Operators

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 **

- Class 7 has the highest precedence (applied first), followed by class 6, then class 5, etc
Example of VHDL Operators

In the following expressions, A, B, C, and D are bit_vectors:

(A & not B or C and D) = "110011"

The operator would be applied in the order:

not, or, and, =

if A = "111", B = "111", C = "0110000", and D = "1101111", the computation would proceed as follows:

not B = "000" (bit-by-bit complement)
A & not B = "1110000" (concatenation)
C or not B = "0110000" (not-right-or-places)
(A & B) or C = "1110111" (bit-by-bit or)
(A & B) or C and D = "1110110" (bit-by-bit and)
not B and (A & C) or (B and D) = "1100110" = TRUE

(remember nodes for the equality test to be done last, and the result is TRUE)

Example of Shift Operators

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 lsh 2 is "01010100" (shift left logical, filled with '0')
A lsh 3 is "00010101" (shift right logical, filled with '0')
A sla 3 is "11010111" (shift left arithmetic, filled with right bit)
A sar 3 is "11100101" (shift right arithmetic, filled with left bit)
A rot 3 is "11010100" (rotate left)
A rot 5 is "11011001" (rotate right)

VHDL Functions

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

  function rotate_right (inp: bit_vector) return bit_vector is
  begin
    return reg reg;  
  end rotate_right;

- A = "10010101"
  B <= rotate_right(A);

- General form

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

Add Function

- This function adds 2-bit vectors and a carry:
  - It returns a 3-bit sum

  function add (A, B: bit_vector; cin: bit) return bit_vector is
  variable dout: bit;
  variable cinh: bit := cin;
  variable sum: bit_vector := downto 0 = "0000";
  begin
    loop: for i in 0 to 2 loop
      cinh := (A(i) xor B(i)) or (cin) or (B(i) and cin);
      dout := (A(i) and B(i)) xor cin;
      Sum(i) := cinh;  
      return cinh;
  end loop;

Example function call:

  Sum(3) <= add(A[1], B[1], cin);

For Loops

General form of a for loop:

  for loop-index in range loop
  sequential-statements
  end loop;

-- Exit statement has the form:
-- exit when condition;

For Loop Example:

-- compare two 8-character strings and return TRUE if equal
function comp_string (string1: string2, string2: string1 to 0)
return boolean is
  variable B: boolean;
  begin
    loop: for i in 1 to 8 loop
      B = string1(i) = string2(i);
      exit when B = FALSE;
    end loop;
    return B;
  end comp_string;

VHDL Procedures

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

General form

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

  procedure_name (actual-parameter-list);
Packages and Libraries

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

- Package declaration

  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;

Library BITLIB – bit_pack package

package bit_pack is
  function (bit, bit) returns (bit);
  function (signal, bit) returns (signal);
  function (const, bit) returns (const);
  function (signal, signal) returns (signal);
  function (const, signal) returns (const);
  function (signal, const) returns (signal);
end (package);bit_pack;

component ADUS
  generic (input bits: in 8)
  port (i, j: in 8; k: out 8)
begin
  k <= i AND j;
end (component);

Library BITLIB – bit_pack package

component ADUS
  generic (input bits: in 8)
  port (i, j: in 8; k: out 8)
begin
  k <= i AND j;
end (component);
VHDL Model for a 74163 Counter

- 74163 – 4-bit fully synchronous binary counter
- Counter operations

<table>
<thead>
<tr>
<th>Clk</th>
<th>Load</th>
<th>Nxt State</th>
<th>Q&lt;sub&gt;3&lt;/sub&gt;</th>
<th>Q&lt;sub&gt;2&lt;/sub&gt;</th>
<th>Q&lt;sub&gt;1&lt;/sub&gt;</th>
<th>Q&lt;sub&gt;0&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>0 0 0 0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>0 0 0 0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0 0 0 0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Q&lt;sub&gt;3&lt;/sub&gt; Q&lt;sub&gt;2&lt;/sub&gt; Q&lt;sub&gt;1&lt;/sub&gt; Q&lt;sub&gt;0&lt;/sub&gt;</td>
<td>Increment count</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Generate a Cout in state 15 if T=1
  - Cout = Q<sub>3</sub>Q<sub>2</sub>Q<sub>1</sub>Q<sub>0</sub>T

Cascaded Counters

Signal Attributes

- Attributes associated with signals that return a value
  - S'EVEN – True if an event occurred during the current data set
  - S'ACTION – True if a transaction occurred during the current data set
  - S'LAST_EVENT – Time elapsed since the previous event on S
  - S'LAST_ACTIVE – Time elapsed since the previous transaction on S

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)

- Event
  - occurs on a signal every time it is changed
- Transaction
  - occurs on a signal every time it is evaluated
- Example:

  \[
  A \leftarrow B \quad \text{B changes at time T}
  \]

<table>
<thead>
<tr>
<th>T</th>
<th>Event</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>T+1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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;

Signal Attributes (cont'd)

Attributes that create a signal

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Creates</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGNAL_DELAY([time])</td>
<td>signal same as \text{S} delayed by specified time</td>
</tr>
<tr>
<td>SIGNAL_HIGH([time])</td>
<td>signal same as \text{S} delayed or specified time</td>
</tr>
<tr>
<td>SIGNAL_LOW([time])</td>
<td>signal same as \text{S} delayed or specified time</td>
</tr>
<tr>
<td>SIGNAL_TRANSAC</td>
<td>signal of type BIT that changes for every transaction on S</td>
</tr>
</tbody>
</table>

Examples of Signal Attributes

VHDL Code for Attribute Test

entity attr_test is
port (A : in bit);
end attr_test;
architecture test of attr_test is
begin
  A <= B and C;
end test;

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;
Array Attributes

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

Recap: Adding Vectors

Add1 and Add2 vectors must be dimensioned as N-1 downto 0.

Procedure for Adding Bit Vectors

Transport and Inertial Delay (cont’d)

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_operate is
  function '+'(A, B : bit_vector) return bit_vector;
  function '+'(A, B, C : bit_vector) return integer;
  end bit_operate;

  library IEEE;
  use IEEE.std_logic_1164.all;

  package body bit_operate is
    -- The add is performed bit by bit with an internal carry
    function '+'(A, B, C : bit_vector) return bit_vector is
      variable ret : bit_vector(Vlength(A) + 1 downto 0);
      variable c : bit;
      begin
        ret(0) <= A(0) + B(0) + C(0);
        for i in 1 to Vlength(A) loop
          c := A(i) + B(i) + C(i) + ret(i-1);
          ret(i) := (not c) and (A(i) or B(i) or C(i));
        end loop;
        ret(Vlength(A) + 1) := c;
        return ret;
      end function;

    -- This function returns a bit_vector of a bit_vector and an integer:
    function '+'(A : bit_vector; B : integer) return bit_vector is
      begin
        return (A <= B + C + 3 ?)
      end function;

    -- 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

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

- 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'

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)

- 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
AND and OR Functions Using X01Z

<table>
<thead>
<tr>
<th>X</th>
<th>0</th>
<th>1</th>
<th>X</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>Z</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

IEEE 1164 Standard Logic

- **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: X01Z

Resolution Function for IEEE 9-valued

```vhdl
current_resolution_table := (cursor, current_table) => |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
```

AND Function for std_logic_vectors

```vhdl```

Resolution Function for X01Z

```vhdl```

AND Table for IEEE 9-valued

```vhdl```