Outline

- What we know
  - How to model Combinational Networks in VHDL
    - Structural, Dataflow, Behavioral
  - How to model Flipflops in VHDL
  - Processes
  - Delays (delta, transport, inertial)
  - How to model FSM in VHDL
  - Wait statements
    - Variables, Signals, Arrays
- What we do not know
  - VHDL Operators
  - Procedures, Functions
  - Packages, Libraries
  - Additional Topics (if time)

Review: Modeling a Sequential Machine

Mealy Machine for 8421 BCD to 8421 BCD + 3 bit serial converter

How to model this in VHDL?

Review: Behavioral VHDL Model

Two processes:
- the first represents the combinational network
- the second represents the state register
Review: 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)
  
  ```
  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.

Review: 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

Review: 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

Review: Signals

- Signals must be declared outside a process
- Declaration form
  ```
  signal list_of_signal_names : type_name
  [ := initial_value ];
  ```
- Declared in an architecture can be used anywhere within that architecture
Review: Constants

- Declaration form
  
  ```vhdl
  constant constant_name : type_name := constant_value;
  ```

  ```vhdl
  constant delay1 : time := 5 ns;
  ```

- Constants declared at the start of an architecture can be used anywhere within that architecture
- Constants declared within a process are local to that process

Review: Variables vs. Signals

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

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

- 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

Review: Variables vs. Signals (cont’d)

Process Using Variables

```vhdl
entity dummy is
  signal trig, sum: integer;
end dummy;

architecture var of dummy is
  begin
    process
      variable var1: integer := 1;
      variable var2: integer := 2;
      begin
        wait on trig;
        var1 := var2 + var3;
        var2 := var1;
        Sum := var1 + var2 + var3;
      end process;
    end var;
end dummy;
```

Process Using Signals

```vhdl
entity dummy is
  signal trig, sum: integer;
end dummy;

architecture sig of dummy is
  begin
    process
      signal sig1, sig2, sig3: integer := 1;
      signal sig4: integer := 2;
      begin
        wait on trig;
        sig1 := sig2 + sig3;
        sig2 := sig1;
        Sum := sig1 + sig2 + sig4;
      end process;
    end sig;
end dummy;
```

Predefined VHDL Types

- Variables, signals, and constants can have any one of the predefined VHDL types or they can have a user-defined type
- Predefined Types
  - bit – {0, 1}
  - boolean – {TRUE, FALSE}
  - integer – [-2^{31} - 1.. 2^{31} - 1]
  - real – floating point number in range -1.0E38 to +1.0E38
  - character – legal VHDL characters including lowercase letters, digits, special characters, ...
  - time – an integer with units fs, ps, ns, us, ms, sec, min, or hr
User Defined Type

- Common user-defined type is enumerated

  ```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;
  ```

  - range must be specified when the array object is declared

    ```vhdl
    signal intvec5 : intvec(1 to 5) := (3,2,6,8,1);
    ```

Sequential Machine Model

Using State Table

<table>
<thead>
<tr>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>S4</th>
<th>S5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x0</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>S2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>S3</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>S4</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S5</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
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 A : bit_vector(0 to 5) := "101011";
-- ('1', '0', '1', '0', '1', '1');
```

- Subtypes
  - include a subset of the values specified by the type
    ```vhdl
    subtype SHORT_WORD is : bit_vector(15 to 0);
    ```

- Predefined subtypes of type integer
  - POSITIVE (all positive integers)
  - NATURAL (all positive integers and 0)

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 **

Example of VHDL Operators

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

(A or not B or C or 2 and 0) = "1100101"

The operators would be applied in the order:

not, or, and, =

if A = "110", B = "111", C = "011000", and D = "110113", the computation would proceed as follows:

not B = "000" (bit-by-bit complement)
A or not B = "110000" (concatenation)
A and C = "000111" (bit-by-bit and)
A or B or C or 2 = "111111" (bit-by-bit or)

(substitute the values for A, B, C, and D)

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 "10101011".

A sll 2 is "01010101" (shift left logical, filled with '0')
A srl 3 is "000000010" (shift right logical, filled with '0')
A sla 2 is "101010111" (shift left arithmetic, filled with right bit)
A sra 2 is "11100101" (shift right arithmetic, filled with left bit)
A rol 3 is "10101100" (rotate left)
A ror 5 is "10101100" (rotate right)
VHDL Functions

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

  ```vhdl
  function rotate_right (rep : bit_vector)
  return bit_vector is
  begin
    return reg xor 1;
  end rotate_right;
  ```

- A = "10010101"

- General form:

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

For Loops

- General form of a for loop:

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

- Exit statement has the form:

  ```vhdl
  exit:
  -- or
  exit when condition;
  ```

- For Loop Example:

  ```vhdl
  variable B : boolean;
  variable A : bit_vector(0 downto 0) := "000001";
  begin
    loop : for j in 1 to 1 loop
      B := string(1) = string(2);
      exit when B = FALSE;
    end loop loop;
    return B;
  end procedure_name;
  ```

Add Function

- This function adds 2 8-bit vectors and a carry.

  ```vhdl
  function add (A,B : bit_vector(7 downto 0); carry : bit) return bit_vector is
  variable res : bit_vector(7 downto 0);
  variable S : bit := carry;
  begin
    loop : for i in 0 to 3 loop
      res := (A(i) and B(i)) or (A(i) and S) or (B(i) and S);
      S := res;
      end loop loop;
    return res;
  end add;
  ```

- Example function call:

  ```vhdl
  Sum_1 <= add(A1, B1, CN);
  ```

VHDL Procedures

- Facilitate decomposition of VHDL code into modules.

- Procedures can return any number of values using output parameters.

- General form:

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

  ```vhdl
  procedure-name (actual-parameter-list);
  ```
Procedure for Adding Bit_vectors

procedure Addvec;
  Add1: in bit vector;
  Add2: in bit vector;
  Sum: out bit vector;
  Caud: out bit;
begin
  C := Caud;
  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
  Caud := C;
end Addvec;

Parameters for Subprogram Calls

<table>
<thead>
<tr>
<th>Mode</th>
<th>Class</th>
<th>Procedure Call</th>
<th>Function Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>signal</td>
<td>constant</td>
<td>signal</td>
<td>signal</td>
</tr>
<tr>
<td>variable</td>
<td></td>
<td>variable</td>
<td>n/a</td>
</tr>
<tr>
<td>inout</td>
<td>signal</td>
<td>signal</td>
<td>n/a</td>
</tr>
<tr>
<td>variable</td>
<td></td>
<td>variable</td>
<td>n/a</td>
</tr>
</tbody>
</table>

1 default mode for functions  2 default for in mode  3 default for inout mode

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-package-name;
  ```

Library BITLIB – bit_pack package

```
Library BITLIB – bit_pack package

package body bit_pack is

function func1 (in : bit_vector) return bit

variable a, b : bit;

end bit_pack;

Library BITLIB – bit_pack package

component is library BITLIB model

end component;

architecture of 74163 is

begin

end architecture;

VHDL Model for a 74163 Counter

- 74163 – 4-bit fully synchronous binary counter

- Counter operations

<table>
<thead>
<tr>
<th>Control Signals</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cn</td>
<td>Lst</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- Generate a Cout in state 15 if T=1

\[ \text{Cout} = Q_0 Q_3 Q_2 Q_1 T \]
Cascaded Counters

Cascaded Counters (cont’d)

library IEEE;
use IEEE.std_logic_arith.all;

entity c4163test is
port(D16, D15, D14, D13, Clk, Cn, Rst, S : in bit);
out (Down0, Down1, Down2, Down3) : out bit;
end;

end c4163test:

architecture Lester of c4163test is
component C4163
port (D16, D15, D14, D13, Clk, Cn, Rst, S : in bit;
out (Down0, Down1, Down2, Down3) : out bit
end;

end component;
n signal Carry : bit;
n signal CIn : integer;
n signaltemp : bit vector(7 downto 0);

begin
U1 : C4163 port map (D16, D15, D14, D13, Clk, Cn, Rst, S, Carry, CIn, temp);
carry <= (temp(2) and CIn);
CIn <= Carry;
end;

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, else false</td>
</tr>
<tr>
<td>S'ACTIVE</td>
<td>True if a transaction occurred during the current cycle, else false</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 the 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)

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

Example:

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

<table>
<thead>
<tr>
<th>Event</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<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

- DELAYED [(time)*]
  - Signal same as S delayed by specified time
- STABLE [(time)*]
  - Boolean signal that is true if S had no events for the specified time
- QUIET [(time)*]
  - Boolean signal that is true if S had no transactions for the specified time
- TRANSACTION
  - Signal of type BIT that changes for every transaction on S

* Delta is used if no time is specified
Examples of Signal Attributes

entity attr_ex is
port (B, C : in bit);
end attr_ex;
architecture test of attr_ex is
signal A, delayed_A, A_trans : bit;
signal A_stable, A_stable_set : boolean;
begin
A <= B and C;
A_delayed <= &delay(5 ns);
A_trans <= A_trans_crit;
A_stable_set <= &stable(5 ns);
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.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Format</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_INDEX(N)</td>
<td>array</td>
<td>index range</td>
<td>ROM1 предн 1 = 0</td>
</tr>
<tr>
<td>A_RIGHT(N)</td>
<td>array</td>
<td>right bound of index range</td>
<td>ROM1 чтот (1) = 15</td>
</tr>
<tr>
<td>A_HIGH(N)</td>
<td>array</td>
<td>upper bound of valid index range</td>
<td>ROM1 чтот (1) = 15</td>
</tr>
<tr>
<td>A_LOW(N)</td>
<td>array</td>
<td>lower bound of index range</td>
<td>ROM1 Чтот (1) = 0</td>
</tr>
<tr>
<td>A_RANGE(N)</td>
<td>array</td>
<td>index range</td>
<td>ROM1 Чтот (1) = 1 to 15</td>
</tr>
<tr>
<td>A_REVERSE_RANGE(N)</td>
<td>array</td>
<td>reverse range</td>
<td>ROM1 Чтот (1) = 15 downto 1</td>
</tr>
<tr>
<td>A_LENGTH(N)</td>
<td>array</td>
<td>length of index range</td>
<td>ROM1 Чтот (1) = 16</td>
</tr>
</tbody>
</table>

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+1-bit sum and a carry. Add1 and Add2 are assumed
to be of the same length and dimension n-1 downto 0.

procedure Add12;
begin
signal Sum_out, bit, n positive;
signal Carry_out, bit;
for i in 0 to n loop
Sum_out(i) <= Add1(i) xor Add2(i) xor C;
C <= (Add1(i) and Add2(i) and C) or (Add1(i) and C) or (Add2(i) and C);
end loop;
end Add12;

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

The 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 addvec();
  var i, j, carry: bit;
  carry := 0;
  for i := 1 to length do
    sum[i] := vec1[i] + vec2[i] + carry;
    if sum[i] > 1 then
      sum[i] := 0;
      carry := 1;
    else
      carry := 0;
    end if;
  end for;
end addvec;
```

Transport and Inertial Delay

```
z3 <= reject 4 ns x after 10 ns;
```

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

```
z3 <= x after 4 ns;
z3 <= transport x 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 $T'$:
- $C$ at $T'$;
- $A$ <= $C$ after 2 ns;
- $A$ <= transport $C$ after 1 ns;
- $A$ <= transport $C$ after 2 ns;
- $A$ <= transport $B$ after 1 ns;
- $A$ <= $C$ after 2 ns;
- $A$ <= transport $C$ after 1 ns;

Transport and Inertial Delay (cont’d)

```
Z3 <= reject 4 ns X 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

The package provides two overloaded functions for the plus operator:

```vhdl
package ...

function "+" (A[1..n], B[1..m], C[1..p]) return B[1..m];
function "+" (A[1..n], B[1..m], C) return B[1..m];
end package;
```

library "std":

package body ...

-- This function returns a bit_vector sum of two bit_vector operands.
function "+" (A[1..n], B[1..m]) return bit_vector is
  variable c: bit_vector(A.length + B.length - 1 downto 0);
  begin
    for i in A'range LOOP
      c(A'first + i) := A(i) + B(i);
    end LOOP;
    return c;
end function;
```

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

Resolution Function for X01Z

AND and OR Functions Using X01Z
IEEE 1146 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

AND Table for IEEE 9-valued

```plaintext
CONSTANT and_table : stdlogic_table := ( 
  0, X, 0, 1, Z, W, L, H, -
);```

AND Function for std_logic_vectors

```plaintext
function 'and' ( l : std_logic ; r : std_logic ) return ( std_logic ) is begin
  return ( and_bit(l, r) ) ;
  end 'and' ;
```

```plaintext
function 'and' ( l : std_logic_vector ; r : std_logic_vector ) return ( std_logic_vector ) is begin
  if length(l) /= length(r) then
    assert FALSE
    report "arguments of overloaded 'and' operator are not of the same length";
    assert vr
  end if;
  variable result : std_logic_vector ( 0 to length(l) ) ;
  begin
    for i in reverse RANGE loop
      result(i) := and_table(l(i), r(i)) ;
    end loop;
    return result ;
  end 'and' ;
```