Still More VHDL

- Making circuit specifications more generic
- Arrays and records
- Using assertions to detect bugs
- Variables
- Functions and procedures

Jon Turner
Signal Attributes

- For signal x: std_logic_vector(15 downto 0),
  - x’left=15, x’right=0
  - these are referred to as attributes of x
  - other attributes include x’high (=15), x’low (=0),
    x’range (=15 downto 0) and x’length (=16)

- Signal attributes can be used to write code that is less dependent on specific signal lengths
  - for example
    x(x’high) <= '1'; x(x’low+3 downto x’low) <= x"c";
    x <= (x’high => '1', x’low => '1', others => '0');

- This can make code easier to change and is useful when writing modules with generic signal lengths
Types and Subtypes

- Define subtype for signals that should have same length and index range
  
  ```vhdl
  subtype word is std_logic_vector(15 downto 0);
  signal x, y: word;
  ```

- IEEE library has defines types *signed* and *unsigned*
  
  - variations on std_logic_vector but specialized for numeric data
    - signed signals are interpreted as negative if first bit is zero
  
  - type conversions required
    
    ```vhdl
    signal x: signed(7 downto 0); signal y: unsigned(7 downto 0);
    signal z: std_logic_vector(7 downto 0);
    x <= signed(z); z <= std_logic_vector(y); y <= unsigned(x);
    to convert to/from an integer i, use
    x <= to_signed(i,8); z <= to_unsigned(i,8); i <= to_integer(x);
    ```
Parameterized Components

- **Generic parameters in entity declarations used to vary component sizes**
  
  ```vhdl
  entity negate is
      generic ( wordSize => integer := 8 );
      port(x: in signed(wordSize-1 downto 0));
      neg_x: out signed(wordSize-1 downto 0));
  end negate;
  
  use wordSize within architecture to define signals
  
  specify alternate widths when instantiating component
  neg1: negate generic map(wordSize => 16) port map(...);
  ```

- **May define several parameters within one generic clause**
Exercises

1. Suppose \( i \) is an integer, \( x \) is an eight bit logic vector and \( y \) is a 16 bit unsigned. The following assignments cause the VHDL language processor to complain. Show how to fix them.
\[
y <= i; \quad i <= y; \quad i <= x; \quad x <= i; \quad x <= y;
\]

2. Write a VHDL module that specifies an increment circuit. Your module should have two generic parameters, a \( \text{radix} \) parameter that specifies the base of the numbers that your circuit handles, and a \( \text{length} \) parameter that specifies the number of digits that your increment function should support. So for example, if the two parameters are 5 and 6, your circuit should increment an input with 6 base-5 digits. The data input and output should be represented as unsigned vectors with \( 4*\text{length} \) bits.
Arrays of std_logic_vectors

-- using these declarations
subtype word is std_logic_vector(wordSize-1 downto 0);
type regFileType is array(0 to regFileSize-1) of word;
signal reg: regFileType;

-- we can write things like
reg(2) <= reg(1) + reg(4);
reg(3 downto 0) <= reg(7 downto 4);
reg(3)(5) <= '1';
reg(int(x)) <= reg(int(y)) -- int() converts to integer

- Synthesizer can implement such arrays using registers constructed from flip flops
- Sometimes can be implemented using memory blocks
  » depends on how the array is used in the VHDL code
  » for large arrays of words that are accessed one at a time, synthesizer will typically use a memory block
Using Array to Define Memory

defines a look-up table entity in VHDL:

```
entity lookupTable is port(
    clk, write: in std_logic;
    row: in rowIndex; inVal: in rowType;
    outVal: out rowType);
end lookupTable;
architecture a1 of lookupTable is
begin
    type tableType is array(0 to 15) of rowType;
    signal table: tableType;
    process(clk) begin
        if rising_edge(clk) then
            outVal <= table(int(row));
            if modTable = '1' then
                table(int(row)) <= inVal;
            end if;
        end if;
    end process;
end a1;
```

- **limited access pattern** allows use of memory components.
- **provides much more compact data storage than registers**.
Defining 2d Constant Arrays

- Constant arrays can be used to define tables of values

```vhdl
subtype asciiChar is std_logic_vector(7 downto 0);
type hex2AsciiMap is array(0 to 15) of asciiChar;
constant hex2Ascii : hex2AsciiMap := (
    x"30", x"31", x"32", x"33", x"34", -- 0-4
    x"35", x"36", x"37", x"38", x"39", -- 5-9
    x"61", x"62", x"63", x"64", x"65", x"66" -- a-f
);
...
ascii3 <= hex2Ascii(3);
```

- Constant arrays are “read-only” so, synthesizer can often implement using Read-Only Memory (ROM)
  - ROMs are generally less “expensive” than read/write memory
Defining Records

-- with these definitions

type entryType is record
  valid: std_logic;
  key, value: word
end record entryType;

constant tableSize: integer := 8;
type tableTyp is array(0 to tableSize-1) of entryType;
signal table: tableTyp

-- we can write

table(2).key <= table(1).value
if table(0).valid = '1' then
  table(5) <= ('1', x"abcd", x"0123");
end if;

- Synthesizer can implement using registers
Exercises

1. Write the declarations needed to define a constant array called `firstOne`, with 16 entries of 4 bits each, where the value of `firstOne(x)` is the position in x of the rightmost 1 bit. So for example, `firstOne("0010")="01"`, `firstOne("1100")="10"`. If none of the bits in x are 1, make `firstOne(x)="00"`.

2. Write a VHDL declaration for an array of records, where the index range for the record is the same as the index range for a `std_logic_vector`, called `z`. Each record in your array should have two fields; field A is a `std_logic_vector` with a descending range that has the same limits as `z`, field B is a `std_logic_vector` with an ascending range that has the same limits as `z`.

3. Write the declarations to define a table in which each entry has three fields, x, y and z, where x is a single bit signal and y and z are eight bit signals. Entries should be numbered 1, 2, ..., 8. Write an assignment that copies the y field of the second entry in the table to the z field of the fifth entry if the x fields of both entries are equal.
Assertions

- Assertion statements are useful for debugging a circuit specification
  - `assert z < x and x < y;`
  - the condition in the assert statement is checked during simulation and triggers an error message if violated
- Add report clause to specify error message
  - `assert z < x and x < y report “x out of range”;`
- Add severity clause to force early termination
  - `assert z < x and x < y severity failure;`
- Also useful in testbenches to compare circuit output with expected output
Variables

- VHDL provides variables in addition to signals
  - unlike signals, variables do not correspond to wires
  - best to think of variable assignment as an abbreviation
    
    ```vhdl
    a <= x"3a";
y := a + x"01";  -- assignment to variable y
b <= y;
y := y + x"10";  -- defines new abbreviation for y
c <= y;
    -- equivalent code segment without variables
a <= x"3a";
b <= a + x"01";
c <= (a+x"01") + x"10";
    ```

- To reduce confusion, no variable should be used in both a synchronous and asynchronous context
Use of Variables in Loops

architecture al of findFirstOne is
signal got1: std_logic_vector(wordSize downto 0);
beg
  process (dIn,got1) begin
    got1(0) <= '0'; index <= (others => '0');
    for i in 0 to wordSize-1 loop
      if got1(i) < dIn(i) then index <= slv(i,lgWordSize); end if;
      got1(i+1) <= dIn(i) or got1(i);
    end loop;
    valid <= got1(wordSize);
  end process;
end al;

architecture al of findFirstOne is begin
process (dIn)
variable got1: std_logic;
beg
  got1 := '0'; index <= (others => '0');
  for i in 0 to wordSize-1 loop
    if got1 < dIn(i) then index <= slv(i,lgWordSize); end if;
    got1 := dIn(i) or got1;
  end loop;
  valid <= got1;
end process;
end al;
Exercises

1. In the VHDL code segment shown below, determine the value assigned to each of the signals x and y on the next clock tick assuming that their initial values are x"4" and x"7".
   if rising_edge(clk) then
     a := x + y;
     x <= a + x"1";
     y <= a + x;
     a := y - x"3";
     x <= a + x"2";
   end if;

2. Unroll the loop in the second example on the previous slide. Then eliminate all occurrences of the variable. Assume wordSize=4.
Functions

- Functions can be used to define sub-circuits that are used repeatedly within a larger circuit
  
  ```
  function max(x, y: word) return word is
  begin
    if x > y then return x; else return y; end if;
  end function max;
  ```

- Each distinct use of a function causes a copy of the subcircuit to be synthesized
  
  ```
  q <= max(s, t);
  r <= max(u, q);
  ```

  » exception: multiple uses with same inputs don’t require separate copies
Procedures

- Procedures can be used to define subcircuits with multiple outputs

```vhdl
procedure inRange(x: in word;
                 signal inRange: out std_logic;
                 signal tooHigh: out std_logic;
                 signal tooLow: out std_logic) is
constant loBound: word := x"1234";
constant hiBound: word := x"abcd";
begin
  tooLow <= '0'; inRange <= '0'; tooHigh <= '0';
  if x < loBound then tooLow <= '1';
  elsif x <= hiBound then inRange <= '1';
  else tooHigh <= '1';
end if;
end procedure;
```

- Procedure parameters are variables by default
  » in this case, must use variable assignment in procedure
Packages

- Packages can be used to define constants, types and functions/procedures used by multiple modules

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;

package commonDefs is
  constant wordsize: integer := 16;
  constant nBtn: integer := 4;
  subtype word is std_logic_vector(wordSize-1 downto 0);
  subtype buttons is std_logic_vector(nBtn-1 downto 0);
  function int(d: std_logic_vector) return integer;
end package commonDefs;

library IEEE; ...
package body commonDefs is
  function int(d: std_logic_vector) return integer is
    return integer(unsigned(d));
end function int;
end package body commonDefs;
```

- Typically placed in a separate source file
Exercises

1. Write a VHDL procedure \texttt{minMax} that takes two eight bit inputs and produces two eight bit outputs. The first of the outputs is equal to the smaller of the two input values, while the second is equal to the larger of the two inputs.

2. Suppose some process contains the code fragment
   \begin{verbatim}
   minMax(a,b,u,v); minMax(c,u,w,x); minMax(c,v,y,z);
   \end{verbatim}
   Draw a block diagram of the resulting circuit. Which signal is the overall minimum of \(a\), \(b\) and \(c\)? Which is the maximum?

3. Let \(x\) be an array 10 of unsigned vectors, each 16 bits long. Write a VHDL code fragment that uses a for-loop to add all the elements in the array and saves the sum in a register called sum. Draw a diagram of the circuit specified by this VHDL.

4. Repeat the last problem, but design the circuit to do one sum computation on each clock tick, instead of doing all the sums at once.