Using VHDL to Design Digital Circuits – Part 2

- Constants and enumeration types
- For and case statements
- Synchronization conditions
- Structural VHDL

Jon Turner
Defining Constants

- Symbolic constants can be declared within architecture
  ```
  constant numBits: integer := 4;
  ```
- To define constants for use by multiple entities, use separate `package` (typically stored in a separate file)
  ```
  package commonConstants is
    constant wordSize: integer := 8;
  end package commonConstants;
  -------------------
  library IEEE;
  use IEEE...
  use work.commonConstants.all;
  entity calculator is port(
    clk, clear, load, add: in std_logic;
    dIn: in std_logic_vector(wordSize-1 downto 0);
    result: out std_logic_vector(wordSize-1 downto 0));
  end calculator;
  ```
Enumerated Types and Subtypes

-- these declarations
type color is (red, green, blue, black, white);
subtype primary is color range red to blue;
signal c1, c2, c3: color;
signal p: primary;

-- allow us to write
c1 <= red;
p <= green;
if c2 /= green then
  c3 <= p;
else
  c1 <= blue;
p <= blue;
end;

named values help clarify intention of signals with symbolic significance

allowed values for subranges are limited; so assignments
  p <= black; p <= c2;
  are not allowed
For-loops

-- Compute 2s-complement of input value
-- Output bit i is flipped if input bits 0..i-1 include a 1
entity negate is port(
  x : in std_logic_vector(wordSize-1 downto 0);
  x_neg: out std_logic_vector(wordSize-1 downto 0));
end negate;
architecture arch1 of negate is
signal foundOne: std_logic_vector(wordSize downto 0);
begin
  process(x, foundOne) begin
    foundOne(0) <= '0';
    for i in 0 to wordSize-1 loop
      x_neg(i) <= foundOne(i) xor x(i);
      foundOne(i+1) <= foundOne(i) or x(i);
    end loop;
  end process;
end arch1;

Note: foundOne must be a vector, not a single bit signal

foundOne(i)=1 if input bits 0..i-1 include a 1

For-loop defines multiple identical (or similar) sub-circuits
Loop does not imply sequential ordering of signal assignments
Loops with Exit Statements

- Exit statement enables/disables generated circuits

```vhdl
process(x) begin
  x_neg <= not x;
  for i in 0 to wordSize-1 loop
    if x(i) = '1' then exit end if;
    x_neg(i) <= x(i);
  end loop;
end process;
```

» defines logically equivalent circuit

- Alternate version without loop

```vhdl
process(x, foundOne) begin
  x_neg <= foundOne xor x;
  foundOne(wordSize downto 1)
  <= x or foundOne(wordSize-1 downto 0);
end process
```

» appears circular, but is not
Exercises

1. Suppose $x$ and $y$ are 16 bit logic vectors, where $y=x+1$. The bits of $y$ can be derived from the bits of $x$, as follows:
   \[
   \begin{align*}
   &\text{if bits } x(0) \ldots x(i-1) \text{ are all ones then } y(i) = \text{not } x(i) \\
   &\text{else } y(i) = x(i)
   \end{align*}
   \]
   Use this to define a VHDL module called plusOne that computes $x+1$ from $x$, using a for-loop. Do two versions, one using an exit statement and one that does not.

2. Draw a circuit diagram for a four bit version of the circuit defined by the VHDL spec from the previous problem.

3. Write a VHDL specification for a circuit module called findFirstOne that has a 16 bit input $dIn$, a four bit output called $x$ and a single bit output called valid. If $dIn$ is zero, then valid is low and $x$ is undefined. If $dIn$ is not zero, then $x$ should be the index of the first bit of $dIn$ that is not equal to zero, and valid should be high.
Understanding VHDL

- VHDL developed for circuit modeling & simulation
  - allows specification of hardware behavior independent of implementation
  - synthesis tools developed later
- currently, two primary uses of VHDL
  - circuit specifications define circuit components and how they are connected
  - testbenches define conventional programs that are used to generate input signals for circuit simulations

- Signals are an abstraction of the wires in real circuits
  - different meaning than variables in sequential programming
  - VHDL includes variables, as well as signals
    - common case: loop indexes are variables, not signals
    - precise meaning of variables in circuit specifications is not always obvious; best to use them sparingly
Understanding VHDL

- Signal assignments define logic circuits
  - signals on left side of assignment change as signals on right side change
    - on clock edge, in case of synchronous assignments
  - not like sequential program execution

- Strong typing in VHDL
  - signal types in expressions must match *exactly*
    - no automatic type conversions
  - bit and integer are built-in types
  - supports user-defined types, such as `std_logic`
    - `std_logic` defines 9 values, including 0, 1 and undefined
      - additional values mostly useful for simulation
Case Statement

- Case statement provides convenient way to express alternatives that depend only on value of a single signal

```vhdl
architecture al of foo is
begin
  process(c,d,e) begin
    b <= '1'; -- provide default value for b
    case e is
      when "00" => a <= c; b <= d;
      when "01" => a <= d; b <= c;
      when "10" => a <= c \xor\ d;
      when others => a <= '0';
    end case;
  end process;
end al;
```

- Can produce more efficient circuits than equivalent if-then-else
Circuit Implementing Case Statement

Case can be implemented with *decoder* and selection logic to choose among alternatives

```vhdl
b <= '1';
case e is
  when "00" => a <= c; b <= d;
  when "01" => a <= d; b <= c;
  when "10" => a <= c xor d;
  when others => a <= '0';
end case;
```

- decoder with *n* inputs generates $2^n$ outputs, one for each of the distinct input values
- one output is high, the rest are low
  - the input value determines which output is high
Exercises

1. Rewrite the VHDL code fragment on slide 10 as a pair of selected signal assignments.
2. Draw a circuit that implements the two selected signal assignments from the previous problem. Compare to the implementation on slide 10.
3. Write a VHDL module using a case statement that is implemented by the circuit at right.
Synchronization Conditions

entity serialParity is port (  
    clk, reset: in std_logic;  
    dIn: in std_logic;  
    parityOut: out std_logic);  
end serialParity;  
architecture arch of serialParity is  
begin  
    process (clk) begin  
        if rising_edge(clk) then  
            if reset = '1' then  
                parity <= '0';  
            else  
                parity <= dIn xor parity;  
            end if;  
        end if;  
        parityOut <= parity;  
    end process;  
end arch;
Understanding Synchronization Conditions

- Signal assigned value in scope of synchronization condition is connected to flip flop output
  - so, cannot assign to same signal outside a sync condition
  - and cannot assign same signal a value in scope of sync condition involving a different clock signal
  - cannot assign same signal in different processes
  - and usually, cannot assign same signal on both rising and falling edges of same clock signal
- Changes to signals assigned within scope of sync condition are delayed until clock occurs
  - to make signal change immediately in response to another signal, place signal assignment outside sync condition
- Be careful with “mixed” processes
  - synchronous assignments require clock in sensitivity list
  - asynchronous assignments require all asynchronous inputs
Example of Mixed Process

```
sensitivity list for both synchronous and asynchronous parts

process (clk, t, x) begin
  if rising_edge(clk) then
    t <= s;
    if en = '0' then
      s <= '0'; t <= '1';
    else
      s <= x xor s;
    end if;
  end if;
  y <= t and x;
end process;
```

```
purely synchronous process

process (clk) begin
  if rising_edge(clk) then
    t <= s;
    if en = '0' then
      s <= '0'; t <= '1';
    else
      s <= x xor s;
    end if;
  end if;
end process;
```

```
asynchronous process

process (t, x) begin
  y <= t and x;
end process;
```

asynchronous assignment
Structural VHDL

entity top is port(...); end top;

architecture al of top is
component calculator port( ... ); end component;
component binaryInMod port( ... ); end component;
component binaryOutMod port( ... ); end component;

signal reset, clear, load, add: std_logic;
signal dBtn, pulse: std_logic_vector(3 downto 1);
signal inBits, outBits: word;
begin
  imod: binaryInMod port map(clk,btn,knob,reset,dBtn,pulse,inBits);
  calc: calculator port map(clk,clear,load,add,inBits,outBits);
  omod: binaryOutMod port map(clk,reset,inBits,outBits,led);

  clear <= dBtn(1) or reset; load <= pulse(0); add <= pulse(3);
  led(7 downto 4) <= inBits(3 downto 0);
  led(3 downto 0) <= outBits(3 downto 0);
end al;
Exercises

1. Draw a circuit that implements the VHDL process shown below. Assume that all
signals are of type std_logic (not std_logic_vector). You may use simple
gates, multiplexors and flip flops in your circuit diagram.

```vhdl
process (...) begin
  x <= a and b;
  if a /= b then x <= b or c; end if;
  if rising_edge(clk) then
    if b > c then y <= a and d;
    elsif a /= c then y <= x;
    end if;
  end if;
end process;
```

2. What signals are needed in the sensitivity list of the process in exercise 1?

3. Suppose we added the assignment `y <= a or b`, to the process in exercise 1 right
after the first assignment to `x`. Would this make sense as a circuit? Why or why not?

4. Suppose we added the assignment `x <= a or b`, to an architecture containing the
process in exercise 1 right after the end of the process. Would this make sense as
a circuit? Why or why not?