CEG 5010: Rapid System Prototyping
Section 3: VHDL Entities and Architectures

Philip Leong

“They know enough who know how to learn”
Henry Adams

A simple VHDL program (4 bit comparator), dataflow description

library ieee;
use ieee.std_logic_1164.all;
-- 4 bit comparator circuit
entity eqcomp4 is
  port (a, b : std_logic_vector(3 downto 0);
     equals : out std_logic); -- equals is active high
end eqcomp4;

architecture dataflow of eqcomp4 is
begin
  equals <= '1' when (a = b) else '0';
end dataflow;

Entities and Architectures

• Comments being with “--”
• Entity describes the I/O properties of the design
  - pinout/interface of the design
  - in means signal is an input (similarly out is an output)
  - std_logic_vector describes a 4 bit bus a[0]; a[1]; a[2]; a[3]
  - std_logic_vector is described as (3 downto 0) means 3 is the MSB
• Architecture describes its function
  - could be behavioural, structural or dataflow (more about this later)
  - “<=” is a signal assignment

Modes

• in - data only flows into the entity
• out - data only flows out of the entity
• buffer - an out signal that is used as in input to other parts of the cell
  - may not be driven by more than one output
  - can only connect to other buffer or an internal signals (not to out or inout signals)
• inout - for bidirectional signals (e.g. databus)
  - actually any signal can be replaced by inout but this is not encouraged for reasons of readability

Dataflow descriptions

• The assignment is a concurrent one
  - all assignments done in parallel (not sequentially like in a program)
  - ordering of the assignments does not matter

Behavioral description of eqcomp4

architecture behavioral of eqcomp4 is
begin
  comp: process(a, b)
  begin
    if (a = b) then
      equals <= '1';
    else
      equals <= '0';
    end if;
  end process comp;
end behavioral;
**Behavioral description**

- Can be synthesized
- Algorithmic way of describing the circuit
- Process is like a parallel task
- Process statement includes a sensitivity list which describes which inputs are used in the process
  - When an input changes value, the process is executed
  - Could also say (in this case ordering does matter)
    - \( a = b \)
    - \( \text{if } (a = b) \text{ equals } '0' \)
    - \( \text{end if} \)

**Structural description of eqcomp4**

Architecture structural of eqcomp4 is

```vhdl
signal x : std_logic_vector(3 downto 0);
begain
u0: xor2 port map (a(0), b(0), x(0));
u1: xor2 port map (a(1), b(1), x(1));
u2: xor2 port map (a(2), b(2), x(2));
u3: xor2 port map (a(3), b(3), x(3));
u4: and4 port map (x(0), x(1), x(2), x(3), equals);
end structural;
```

**Structural description**

- Hierarchical netlist description
- Components instantiated and connected together with signals
- This description cumbersome and cannot be used if the number of bits changed

**Behavioral, dataflow and structural descriptions for synthesizing circuits**

- We can use all of these forms for synthesis
  - Normally easiest to
    - Decompose circuit into small units
    - Make behavioral or dataflow descriptions of blocks
      - Simulate
      - Connect together using structural
      - Optimize by adding directives and/or changing descriptions

**VHDL Synthesis**

- VHDL is often used for simulation purposes only
  - Like a programming language
  - Only a subset is synthesizable
  - Synthesized programs may not simulate in the expected way
- Process statement execution
  - Statements executed sequentially until last statement then waits for
    a change in one of the signals in the sensitivity list
  - Although VHDL allows “wait” anywhere in the code, only makes sense in synthesis at the beginning or end of a process

**Simulation, “wait” and “after”**

```vhdl
proc1: process (a, b, c)
begain
    x <= a and b and c;
    end process;
has same semantics as
proc2: process
begain
    x <= a and b and c;
    wait on a, b, c;
end process;
```

- The following works only in simulation (it doesn’t make a lot of sense to synthesize such a circuit and synthesis software normally ignores “after”)
- `proc3`: process (a, b, c)
  - \( x <= a \text{ and } b \text{ and } c \) after 5 ns;
  - End process;
- `proc1` and `proc2` makes the assignment one delta delay after a change in a, b or c in simulation
Incomplete sensitivity lists

proc5: process(a, b)
begin
  x <= a and b and c;
end process;

- note c not in sensitivity list
  - can we make a circuit that does proc5?
- synthesis software will either
  - say this is the same as proc1
  - give a warning/compile error

Incomplete sensitivity lists

Simulation of VHDL

- New values assigned \( \delta \) time after the input changes
  - \( \delta \) is an infinitesimally small value
  - to process concurrent statements, time only increased after all signals have been processed
  - "transition time" of logic
  - note we use digital logic in the same way
  - e.g. a and b are high for proc1

<table>
<thead>
<tr>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
</tr>
<tr>
<td>( \delta )</td>
</tr>
</tbody>
</table>

What happens in the following?

interesting: process(a, b, c)
begin
  x <= '0';
  if (a = b or c = '1') then
    x <= '1';
  end if;
end process interesting;

- if a, b, c are initially '0' and then c becomes '1' at 5 ns
  - 1st statement makes x=0 as it stays at this value up to 5ns
  - 3rd statement makes x=1 at 5+ \( \delta \) ns

Execution model

1. all expressions on the RHS of a "<=" are computed
   - order does not matter
2. LHS updated at the end of a process statements to the last assignment of that variable after a delay of \( \delta \)
   - NB \( \delta \) is infinitely small

Another example

-- a, b, c initially zero
-- a goes high at 100ns
-- b goes high at 200ns

entity delta is port(
  a, b, c, d: in bit;
  u, v, w, x, y, z: buffer bit);
end delta;

architecture delta of delta is
begin
  z <= not y;
  y <= w or x;
  x <= u or v;
  w <= u and v;
  v <= c or d;
  u <= a and b;
end delta;

architecture xx of yy is
begin
  process(a_bus)
  begin
    x <= '1';
    for i in 7 downto 0 loop
      x <= a_bus(i) and x;
    end loop;
    end process;
end xx;

Why doesn’t this work?

- suppose initially a_bus is all 0’s
  - this always evaluates to x=0
- a_bus(i) always 0 at time T so x always 0
Multiple drivers

- The following will cause a "multiple driver" error
  
  architecture xx of yy is begin
  
  y <= a and b;
  y <= a or b;
  end xx;

- although obvious here, easy to do accidentally in different entities
  
  y <= a and b when oe = '1' else 'Z';

Adding don’t care values

-- s is std_logic_vector
if s = "00" or s = "11" then
  y <= '1';
elsif s = "01" then
  y <= '0';
else
  y <= '-';
end if;

- '1' means "don’t care"
- in this case can make sure we generate
  
  y = s1 + s0;
  instead if
  
  y = '1' / '0'

VHDL Identifiers

- Alphabetic, numerical and/or underscore
  
  - 1st character must be a letter
  
  - last character cannot be an underscore
  
  - 2 underscores in succession are not allowed

- upper and lower case are the same

Data Objects

- Constants
  
  - constant width: integer := 8;

- Signals
  
  - represent wires
    
    - signal count: std_logic_vector(3 downto 0);

    - initialized values ignored for synthesis but used in simulation
      
      - signal count: std_logic_vector(3 downto 0) := "1010";

    - ports are also signals
      
      - ports have modes
      
      - signals can be read or written

Data Objects (cont.)

- variables
  
  - used in functions and procedures
  
  - declared in the process or subprogram
  
  - do not represent wires
  
  - not well defined what they mean in terms of synthesis
  
  - we will avoid using them if possible since we aim to write portable VHDL code

Looped AND which does work using variables

architecture will_work_of my_and is begin
  
  process(a_bus)
  
  variable tmp: bit;

  begin
    
    tmp := '1';

    for i in 7 downto 0 loop
      tmp := a_bus(i) and tmp;
    end loop;

    x <= tmp;
  
  end process;

  end will_work;

- := means an immediate assignment
  
  - recall <= scheduled after a delay of \(\delta\)

  - this now works like a C program would

  - tmp gets assigned straight away
Data objects

- Files
  - used for test benches (will discuss later)
- Aliases
  - an alternate identifier for an existing object
  - e.g. signal address: std_logic_vector(31 downto 0);
  - alias std: std_logic_vector(7 downto 0) is address(31 downto 24);

Scalar Data types - enumeration

- enumeration
  - e.g. type states is (idle, preamble, data, jam, null, error);
  - these are ordered by the declaration
  - preamble < data

- standard enumerations
  - type boolean is (FALSE, TRUE);
  - type bit is (‘0’, ‘1');

- type std_logic is
  - ‘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

- ‘W’, ‘X’, ‘V’ not supported by synthesis

Other Scalar Data Types

- Integer
  - 32 bit signed
  - signals or variables that are integers should be constrained to a
    range for synthesis
    - variable a integer range -255 to 255;
- Floating point
  - only predefined is real
  - often not supported
- Time
  - type time is range -2147483647 to 2147483647 units fs;
  - fs, ps, ns, us, ms, sec, min, hr

Composite types - arrays

- Arrays
  - array bit_vector is
    - array/natural range <>) of bit;
  - array std_logic_vector is
    - array/natural range <>) of
      std_logic;
  - range <> means its is not specified
  - you can define your own types
  - type word is array(15 downto 0) of std_logic;
  - signal w: word;
- two dimensional arrays
  - type table4 is
    - array(0 to 7, 0 to 3) of bit;
  - constant exclusive_or:
    - table4 <= (
      '000_0', '001_1',
      '010_1', '011_0',
      '100_1', '101_0',
      '110_0', '111_1');

Composite types - records

- Like a struct in C
  - type icell is record
    - buffer_in: bit_vector(7 downto 0);
    - enable: bit;
    - buffer_out: bit_vector(7 downto 0);
  - end record;
  - signal busig1, busig2: icell;

  - busig1.enable <= '0';
  - busig2 <= busig1;

Attributes

- provide information about entities, architectures, types and signals
  - several predefined value, signal and range attributes that are useful
- scalar type value attributes
  - ‘left’ leftmost value, ‘right’ rightmost value
  - ‘high’ highest value, ‘low’ lowest value
  - ‘length’ # els in the array
  - e.g. sig1 high

- signal attribute
  - ‘event is true if signal has just occurred

- range attribute
  - gives a signal’s range e.g. buffer_in’range is 7 downto 0
**Example: Shift Right**

- signal x: std_logic_vector(7 downto 0);
- y <= x(0) & x(x'high downto 1);  

**Binary, Octal and Hex Specifiers**

- Base specifiers can be used to describe the base of a string
  - B for binary
  - O for octal
  - X for hexadecimal
e.g.: a <= X"7A";

**CEG 5010: Rapid System Prototyping**

**Section 4: VHDL Combinatorial and Sequential Logic**

Philip Leong

**Combinatorial logic:**

Concurrent Statements

- Boolean equations can be used in concurrent and signal assignment statements
e.g.: 4-bit mux

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity mux is
  a, b, c, d: in std_logic_vector(3 downto 0);
  s: in std_logic_vector(1 downto 0);
  x: out std_logic_vector(3 downto 0));
end mux;
architecture archmux of mux is
begin
  x <= (a(3) and not(s(1)) and not(s(0)))
or (b(3) and not(s(1)) and s(0))
or (c(3) and s(1) and not(s(0)))
or (d(3) and s(1) and s(0));
end archmux;
```

**Logical Operations**

- Note that unlike C or other languages, logical operators in VHDL have no precedence!
- You must use parentheses to group the operators

**with-select-when**

- with selection_sig select
  - value a when sel_sig1,
  - value b when sel_sig2,
  - value c when sel_sig3,
  - value d when sel_sig4,
- architecture archmux of mux is
  with s select
  - x <= a when "00",
  - b when "01",
  - c when "10",
  - d when "11",
  - "-" when others;
end archmux;
- note if a is a std_logic_vector, there are a wide range of other values each el can take (e.g. `'`, `'Z'`, `'L'`, `'H'` etc)
**when-else**

- `x <= a when (s = "00") else b when (s = "00") else c when (s = "00") else d;
- this is different from with-select-when since the condition need not be mutually exclusive
  - in case of conflicts, earlier conditions have higher priority
- E.g. `j <= w when a = "1" else x when b = "1" else y when c = "1" else z when d = "1" else '0';
- priority encoder

**quiz**

A) what does the following do?

```vhdl
tmp <= (a, b, c, d);
with select
  j <= w when "0000",
    x when "0100",
    y when "0010",
    z when "0001",
  '0' when others;
```

B) how about

```vhdl
tmp <= (a, b, c, d);
with select
  j <= w when "1--",
    x when "1--",
    y when "1--",
    z when "1--",
  '0' when others;
```

**Answer**

A) `j <= (a and not(b) and not(c) and not(d) and w) or (not(a) and b and not(c) and not(d) and x) or ...`

B) `error`
- during simulation `j <= w` only if `a` is "1" and `b,c,d` are the value "--" NOT when `b,c,d` are "1" for instance!
- Synthesis needs to behave like simulation and "--" is not a value that `b,c,d` can ever take in synthesis
- "--" is not really a don’t care

**when-else conditions**

- with-select-when are like switch statements in C
- when-else are like if-else in C
  - more powerful since condition can be an expression
    - `a <= b when (c = d) and (e < f) else g;

**std_match**

- to get around the problem of b)
  - if `a = "l --l"` then `--` is always false in synthesis
  - if (std_match(a, "l --l")) then
    - evaluates true if `a(3)` and `a(0)` are equal to 1

**Component instantiation**

- VHDL does not define that certain components must be in the system
  - instantiation is system dependent
  - this makes it non-portable
  - however it would be optimized for the architecture
**Sequential statements**

- Sequential statements are those contained in a process, function or procedure
- Order of signals affects how the logic is synthesized
- Sequential statements can create combinatorial logic
- Following are equivalent

```vhdl
similar: process(addr)
begin
  step <= '0';
  if (addr > x'0F') then
    step <= '1';
  end if;
  end process;

similar2: process(addr)
begin
  if (addr > x'0F') then
    step <= '1';
  else
    step <= '0';
  end if;
  end process;
```

**Implied memory**

```vhdl
not similar: process(addr)
begin
  if (addr > x'0F') then
    step <= '1';
  end if;
  end process;
- no initial values in VHDL
- this will cause step to become '1' and stay there forever when addr > x'0F'
  - what circuit can do this? (hint must have memory)
```

**if-then-elseif**

```vhdl
if (cond1) then
  action1;
elsif (cond2) then
  action2;
else
  action3;
```

- What logic statement does this generate?
- How is this related to when-else?
- Which is more powerful?

**case-when**

```vhdl
process(addr)
begin
  case addr is
    when x'001' => decode <= X'11';
    when x'111' => decode <= X'11';
    when x'011' => decode <= X'11';
    when other => decode <= X'00';
  end case;
end process;
```

- Similar to when-else but can change the output expression

**Synchronous logic - a D flip-flop**

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity dff_logic is port (d, clk: in std_logic; q: out std_logic);
end dff_logic;
architecture example of dff_logic is begin
  process (clk) begin
    if (clk'event and clk = '1') then
      if (d = '1') then
        q <= d;
      else
        q <= a;
      end if;
    end if;
  end process;
end example;
```

- Only clk is in sensitivity list
  - changes it d should not run the process (many will anyway)
- if (clk'event and clk = '1')
  - standard way to recognize a rising edge
  - how do we do a falling edge?
- How about a level sensitive latch?
- A T flip flop?

**Bad VHDL code**

- Putting and else after an if (clk'event and clk='1') is not clear don't use it since it may not be portable

```vhdl
if (clk'event and clk='1') then
  q <= d;
else
  q <= a;
end if;
```
Wait statements and rising_edge

- Instead of using a process with a sensitivity list and 'event can use
  architecture example2 of diff_logic
  begin
  process begin
  wait until (clk = '1');
  q <= d;
  end process;
  end example2;
- Can also use rising_edge or falling_edge
  process (clk)
  begin
  if rising_edge(clk) then
    q <= d;
  end if;
  end process;

An 8 bit counter with preload of 5

entity cnt8 is
  clk, grst in std_logic;
  enable, load in std_logic;
  data in std_logic_vector(7 downto 0);
  cnt buffer
    std_logic_vector(7 downto 0);
end cnt;
architecture a_cnt8 of cnt is
  begin
  count process(grst, txclk)
    begin
      if (grst = '1') then
        cnt <= 00000001;
      elsif (txclk = '1') then
        if (load = '1') then
          cnt <= data;
        elsif (enable = '1') then
          cnt <= cnt + 1;
        end if;
      end if;
      end process count;
    end architecture;

Asynchronous reset

process (clk, reset)
begin
  if (reset = '1') then
    q <= '0';
  elsif (clk'event and clk = '1') then
    q <= d;
  end if;
end process;

- How do we do a synchronous reset?

Aggregates

- cnt <= (others => '0'); -- all elements of cnt set to 0
- cnt <= ('1', '0', others => '1'); -- cnt becomes '10' & all 1's deep on cnt's size
- cnt <= (others => 'Z'); -- high impedance output
- cnt <= data(7) & data -- & is the concatenation operator

Tristate and Bidirectional signals

- How do we make the counter have a tristate output?
- How about using the same bus for output and loading?

Tristate and Bidirectional signals

- Tristate
  - cnt_out: buffer std_logic_vector(7 downto 0);
  - cnt_out <= (others => 'Z') when oe = '0' else cnt;

- Bidirectional
  - cnt_out: inout std_logic_vector(7 downto 0);
  if (load = '1') then
    cnt <= cnt_out;
  elsif (enable = '1') then
    cnt <= cnt + 1;
  end if;
  - cnt_out <= (others => 'Z') when oe = '0' else cnt;
**For loops**

- Variables automatically declared in a loop
  - for i in 7 downto 0 loop
    - data(i) <= '0';
  end loop;

- Next is like continue in C
  - for i in 7 downto 0 loop
    - if i = 4 then
      - next;
    - else
      - data(i) <= '0';
    end loop;

**While loops**

- Variables not needed often in synthesis
  - an exception is loop counters

reg_array: process (not, clk)
  variable i: integer := 0;  -- note => not <= for variables
begin
  while i < 7 loop
    - data(i) := 0;
    - i := i + 1;
  end loop;
end process;