EE224 Handout
A routine way to express an RTL description in VHDL

Madhav P. Desai

April 4, 2018

1 Behavioural descriptions in VHDL: process statements

Consider the following fragment of VHDL code

```
-- a, b, c, d are signals of type std_logic
a <=  b and c and d;
```

This describes a driver which creates events on a based on events occurring at b, c and d.

Here is another way to express this driver:

```
-- a process statement which is sensitive
-- to b,c,d.
process(b,c,d)
   -- a variable for holding intermediate
   -- values.
   variable a_var : std_logic;
begin
   -- a_var is assigned the value b and c
   -- (immediately)
   a_var := b and c;

   -- a_var is assigned the value a_var and d
   -- (immediately)
   a_var := a_var and d;

   -- a transaction is created at a based on
   -- the value of a_var.
   a <= a_var;
end process;
```

This is an example of a *process* statement, whose syntax is

```
process ( <sensitivity-list-of-signals)
  <list-of-variable-declarartions>
begin
  <sequential-statements-in-process>
end process;
```

The interpretation of the process statement is as follows.

- At a time instant t, if there is an event at one of the signals in the sensitivity list, then the process statement is executed at time t.
- Execution of the process statement goes as follows:
  - The sequential statements in the process statement are executed serially, all at time t.
  - Variable assignment statements of the form

    ```
    a_var := b and c;
    ```

    take effect immediately, that is, if executed at time instant t, the value of the variable *a_var* is updated at t itself so that the variable *a_var* keeps track of the last value that was assigned to it.
  - Signal assignments of the form

    ```
    a <= a_var;
    ```

    create transactions as in the usual case.
- When the last statement in the sequential statement list is executed, the process becomes dormant until there is an event on a signal in the sensitivity list, and the beat goes on.

Sequential statements can be of the following types:

- Variable assignments, e.g.

  ```
  a_var := (b and c);
  ```

- Signal assignments, e.g.

  ```
  a <= a_var;
  ```

- If-then-else statements, e.g.

  ```
  if (p = '1') then
     q_var := r and s;
  else
     q_var := r or s;
  end if;
  ```

- Case statements, e.g.

  ```
  case p is
    when '1' =>
        q_var := r and s;
    when '0' =>
        q_var := r or s;
    default =>
        q_var := r and s;
  end case;
  ```

- For-loop statements, e.g.

  ```
  for I in 0 to 9 loop
    <sequence-of-statements>
  end loop;
  ```

- While-loop statements: e.g.

  ```
  -- I is an integer.
  while (I < 9) loop
    <sequence-of-statements>
    I := (I + 1);
  end loop;
  ```

There is lots more.. See the tutorial by Ashenden.

2 Combinational logic

To describe combinational logic, the simplest thing to do is to use If-then-else statements or case statements to express the logic function. For example

```
process (sel, b, c)
 variable mux_out_var: std_logic;
begin
   -- default value is ’0’..
   -- this ensures that mux_out_var
   -- has no memory of the past.
   mux_out_var := ’0’;

   -- mux_out_var is the muxed
   -- value of b/c determined by sel.
   case sel is
     when ’0’ =>
           mux_out_var := b;
     when ’1’ =>
           mux_out_var := c;
     default =>
           mux_out_var := ’0’;
   end case;

   -- muxed value is passed out of the
   -- process statement to its output.
   mux_out <= mux_var;
end process;
```

3 Recommended style of writing a process statement

Use variables to compute values to be passed to outputs and then do the output assignments. Variables should be initialized with default values to avoid introducing memory of the past.

```
process (a,b,c,d) -- don’t forget to include signals here
   -- variable declarations: declare a variable for
   -- each signal driven by this process.
   variable e_var: std_logic; -- variable declarations
begin
   e_var := '0'; -- default value.

   -- compute value to be passed to
   -- signal e.
   if(a = '1') then
      e_var := (c or d);
   else
      e_var := (c and d);
   end if;

   -- pass the value to the signal e
   e <= e_var;

end process;
```

4 Sequential logic

Here is a description of a D flip-flop.

```
process (D, CLK)
  -- Convention: variable is named next_Q_var because
  -- it will be passed to signal Q with
  -- a delay.
  variable next_Q_var: std_logic;
begin
  next_Q_var := D;

  if (CLK'event and (CLK='1')) then
    Q <= next_Q_var;
  end if;
end process;
```

Lets move on to describe the following state-machine:

```
// state_sig is the state signal.
// States are {A,B,C}
// A is the reset state.
// Input is X
// Input symbols {U,D}
// Output is Y
// Output symbols {Y,N}
// Let reset be the reset signal used
// to put the machine in state A.
//
// By default, Y is N.
A: if (X == U) then
     $state := B
   else
     $state := C
   end if
B: if (X == U) then
     $state := C
   else
     $state := A
   end if
C: if (X == U) then
     $state := A
     Y = U
   else
     $state := B
   end if
```

The description in VHDL is direct and simple.

```
architecture Behave of ... is
   -- reset and clock are assumed to
   -- be ports or signals declared here.

   -- first define some types..
   type FsmState is {A,B,C};
   type InputSymbols is {U,D};
   type OutputSymbols is {Y,N};

   -- state signal
   signal fsm_state: FsmState;
   signal X: InputSymbols;
   signal Y_sig: OutputSymbols;

begin

   -- FSM process.
   process(clk,X,fsm_state,reset)
     variable next_fsm_state_var: FsmState;
     variable Y_sig_var: OutputSymbols;
   begin
       -- default values.
       next_fsm_state_var := fsm_state;
       Y_sig_var := N;

       -- calculate values of next-state and
       -- Y vars
       case fsm_state is
         when A =>
              if (X == U) then
                  next_fsm_state_var := B;
              else
                  next_fsm_state_var := C;
              end if;
         when B =>
              if (X == U) then
                  next_fsm_state_var := C;
              else
                  next_fsm_state_var := A;
              end if;
         when C =>
              if (X == U) then
                  next_fsm_state_var := A;
                  Y_sig_var := Y;
              else
                  next_fsm_state_var := A;
              end if;
       end case;
```

```
        -- update signals

        -- immediate update
        Y_sig <= Y_var;

        -- delayed update
        if (clk'event and clk = '1') then
           -- notice that reset is brought
           -- in here.
           if (reset = '1') then
              fsm_state <= A;
           else
              -- non-reset case.
              fsm_state <= next_fsm_state_var;
           end if;
        end if;
    end process;

    ....

end Behave;
```

Finally, lets look at an RTL machine.

```
// count is a register with 8 bits.
register count[9:0];
thread DCount {
 rst: if (start)  then
        // x := y means x(k+1) = y(k)
        count := 1023,
        // goto decr means
        //   q(k+1) = decr
        $state := decr
     else goto rst end if
 decr: if(count == 0) then
        // done = 1 means done(k) = 1
        done = 1, $state := rst
      else
```

```
        count := count - 1, $state := decr
      end if
}
```

A VHDL description of this RTL machine can be routinely obtained in a manner similar to that used for the FSM earlier.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

architecture Behave of ... is
   -- reset and clock are assumed to
   -- be ports or signals declared here.

   -- first define some types..
   type RtlState is {rst, decr};
   signal rtl_state: RtlState;

   -- input start coded as a single bit.
   signal start: std_logic;

   -- output done coded as a single bit.
   signal done: std_logic;

   -- signal declaration for register.
   -- (note: The unsigned type is a vector of
   --    std-logic for which add/sub/mul/div
   --    functions are defined).
   signal count: unsigned (9 downto 0);

   -- a useful constant.
   constant Z9 : unsigned (9 downto 0) := (others => '0');

begin

   -- the process description of the RTL
   process(clk, reset, rtl_state, start, count)
```

```
    variable next_rtl_state_var: RtlState;
    variable next_count_var: unsigned (9 downto 0);
    variable done_var: std_logic;
  begin
    next_rtl_state := rtl_state;
    next_count_var := count;
    done_var := '0';

    case rtl_state is
        when rst =>
            if (start = '1') then
               next_rtl_state := decr;
               next_count_var := (others => '1');
    end if;
        when decr =>
            if (count = Z9) then
                next_rtl_state := rst;
                done_var := '1';
    end case;

    done <= done_var;

    if (clk'event and clk = '1') then
        if (reset = '1') then
           rtl_state <= rst;
        else
           rtl_state <= next_rtl_state_var;
        end if;
        count <= next_count_var;
    end if;
  end process;
end Behave;
```

Thats it.

5 Disclaimers

This is not the only way to translate an RTL algorithm to VHDL, but this is the easiest.