ECE381(CAD), Lecture 8:

Behavioral Modeling in VHDL

Mehdi Modarressi

Department of Electrical and Computer Engineering,

University of Tehran

Pictures and examples are taken from the slides of "VHDL: Analysis & Modeling of Digital Systems" and also from "VHDL by Example"

Behavioral modeling

- Readings:
  1. "VHDL: Analysis & Modeling of Digital Systems": Chapter 9.1 and 9.2
  2. "VHDL by Example": Chapter 3
  3. "Designers Guide To VHDL": chapter 3
- Recall:
  - A difference in taxonomy: Behavioral modeling is called "Sequential modeling" in readings 2 and 3

Behavioral modeling

- In sequential architecture bodies
- Defined using the PROCESS statement
  - Contains only sequential statements
- PROCESS statement is itself a concurrent statement
- PROCESS statement has a declaration section and a statement part
  - In the declaration section, types, variables, constants, subprograms, and so on can be declared
  - The statement part contains only sequential statements
    - CASE statements, IF THEN ELSE statements, LOOP statements, and so on

Behavioral modeling

- The basic structure of PROCESS have an explicit sensitivity list
  - List of the signals that will cause the process to execute

```
PROCESS (clk)
BEGIN
    b<= b+1;
END PROCESS;
```

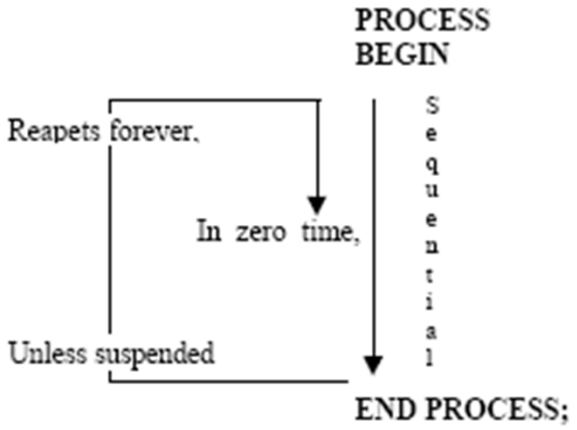
- PROCESS is concurrent with other assignments
- These two codes are equivalent:

```
ARCHITECTURE ...
BEGIN
   ...
   a <= b;
   ...
   c <= d;
   ...
END ...;
```

```
ARCHITECTURE ...
BEGIN
   ...
   PROCESS (b)
     ...
     a <= b;
   END PROCESS;
   ...
   c <= d;
   ...
END ...;
```

Behavioral modeling

- A PROCESS statement without sensitivity list loops forever!
- Sensitivity list stops the looping until the conditions are met
- The PROCESS statement iteration is completed in zero time
  - Unless explicit timing control statements are used



Behavioral modeling

- Have an explicit sensitivity list.
  - List of the signals that will cause the PROCESS to execute
- Whenever port a or b has a change in value, the statements inside of the PROCESS are executed
- Each statement is executed in serial order starting with the statement at the top of the PROCESS

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
ENTITY nand2 IS
  PORT( a, b : IN std_logic;
        c : OUT std_logic);
END nand2;

ARCHITECTURE nand2 OF nand2 IS
BEGIN
  PROCESS( a, b )
   VARIABLE temp : std_logic;
  BEGIN
   temp := NOT (a and b);

   IF (temp = '1') THEN
     c <= temp AFTER 6 ns;
   ELSIF (temp = '0') THEN
     c <= temp AFTER 5 ns;
   ELSE
     c <= temp AFTER 6 ns;
   END IF;

  END PROCESS;
END nand2;
```

Behavioral modeling

- Variables : a software variable for temporarily storage
- Defined like signals, can have the same types
- Variable definition is only allowed inside a PROCESS

Using Objects In VHDL		BODY					
		Concurrent			Sequential		
		Declare	Assign to	Use	Declare	Assign to	Use
OBJECT	Signal	YES	YES	YES	NO	YES	YES
	Variable	NO	NO	YES	YES	YES	YES
	Constant	YES	--	YES	YES	--	YES
	File	YES	--	YES	YES	--	YES

Variable vs. signal

- Signal assignment <=
- Variable assignment :=
- PROCESS is performed in 0 physical time

  → Signal assignment is done at the end of process
- Variable assignment has no delay; it happens immediately

Example- a multiplexer

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
ENTITY mux IS
PORT (i0, i1, i2, i3, a, b : IN std_logic;

      q : OUT std_logic);
END mux;

ARCHITECTURE wrong of mux IS
  SIGNAL muxval : INTEGER;
BEGIN
  PROCESS ( i0, i1, i2, i3, a, b )
  BEGIN
   muxval <= 0;
   IF (a = '1') THEN
     muxval <= muxval + 1;
   END IF;

   IF (b = '1') THEN
     muxval <= muxval + 2;
   END IF;

   CASE muxval IS
     WHEN 0 =>
       q <= I0 AFTER 10 ns;
     WHEN 1 =>
       q <= I1 AFTER 10 ns;
     WHEN 2 =>
       q <= I2 AFTER 10 ns;
     WHEN 3 =>
       q <= I3 AFTER 10 ns;
     WHEN OTHERS =>
       NULL;
   END CASE;
  END PROCESS;
END wrong;
```

- *muxval* is a signal
- Its new value is assigned at the end of the PROCESS
- The case sees the value of the *muxval* at the beginning of the PROCESS
- This code doesn't work!

Example- a multiplexer

```
LIBRARY IEEE;
USE IEEE.std_logic_1164ALL;
ENTITY mux IS
PORT (i0, i1, i2, i3, a, b : IN std_logic;
      q : OUT std_logic);
END mux;

ARCHITECTURE better OF mux IS
BEGIN
  PROCESS ( i0, i1, i2, i3, a, b )
   VARIABLE muxval : INTEGER;
  BEGIN
   muxval := 0;
   IF (a = '1') THEN
     muxval := muxval + 1;
   END IF;

   IF (b = '1') THEN
     muxval := muxval + 2;
   END IF;

   CASE muxval IS
     WHEN 0 =>
       q <= I0 AFTER 10 ns;
     WHEN 1 =>
      q <= I1 AFTER 10 ns;
    WHEN 2 =>
      q <= I2 AFTER 10 ns;
    WHEN 3 =>
      q <= I3 AFTER 10 ns;
    WHEN OTHERS =>
      NULL;
  END CASE;
 END PROCESS;
END better;
```

- *muxval* is a variable
- Its new value is assigned immediately
- The case sees the updated value of the *muxval*
- This code works correctly!

Sequential statements

- Used inside a PROCESS
- All constructs used in software languages
  - IF
  - CASE
  - LOOP
  - EXIT
  - ASSERT
  - WAIT

IF-then-else statement

```
IF (day = sunday) THEN
  weekend := TRUE;
ELSIF (day = saturday) THEN
  weekend := TRUE;
ELSE
  weekday := TRUE;
END IF;
```

- The IF statement can have multiple ELSIF statement parts, but only one ELSE
- More than one sequential statement can exist between each statement part

Case statement

```
CASE instruction IS
  WHEN load_accum =>
    accum <= data;
  WHEN store_accum =>
    data_out <= accum;
  WHEN load|store =>
    process_IO(addr);
  WHEN OTHERS =>
    process_error(instruction);
END CASE;
```

- It is an error if an OTHERS clause does not exist, when the given choices do not cover every possible value of the expression type

Case example

```
TYPE vectype IS ARRAY(0 TO 1) OF BIT;
VARIABLE bit_vec : vectype;
.
.

CASE bit_vec IS
   WHEN "00" =>
    RETURN 0;
   WHEN "01" =>
    RETURN 1;
   WHEN "10" =>
    RETURN 2;
   WHEN "11" =>
    RETURN 3;
END CASE;
```

Type declaration

- One way to convert an array of bits into an integer

Loop statement

- Like the loops in software
  - Iteration capability
  - Whenever an operation needs to be repeated
- Two types:
  - For
  - While

```
FOR i IN 1 to 10 LOOP
  i_squared(i) := i * i;
END LOOP;
```

```
WHILE (day = weekday) LOOP
  day := get_next_day(day);
END LOOP;
```

Loop statement

- VHDL does not allow any assignment to the loop index
- Te index value *i* is locally declared by the FOR statement
  - Does not need to be declared explicitly

```
erroneous : process is
    variable i, j : integer;
begin
    i := loop_param;                    -- error!
    for loop_param in 1 to 10 loop
        loop_param := 5;                -- error!
    end loop;
    j := loop_param;                    -- error!
end process erroneous;
```

Next statement in loops

- Like "continue" in C++
- Stops the current iteration and starts from the next one

```
PROCESS(A, B)
  CONSTANT max_limit : INTEGER := 255;
BEGIN
  FOR i IN 0 TO max_limit LOOP
   IF (done(i) = TRUE) THEN
     NEXT;
   ELSE
     done(i) := TRUE;
   END IF;

   q(i) <= a(i) AND b(i);

  END LOOP;
END PROCESS;
```

Constant declaration

- New syntax: constant
- Like constant in C++
- Can be defined in architecture declaration part or process declaration part

```
PROCESS(A, B)
  CONSTANT max_limit : INTEGER := 255;
BEGIN
   FOR i IN 0 TO max_limit LOOP
    IF (done(i) = TRUE) THEN
      NEXT;
    ELSE
      done(i) := TRUE;
    END IF;

    q(i) <= a(i) AND b(i);

   END LOOP;
END PROCESS;
```

Exit statement

- Like "break" in C++
- Exit from the loop

```
PROCESS(a)
  variable int_a : integer;
BEGIN
  int_a := a;

  FOR i IN 0 TO max_limit LOOP
   IF (int_a <= 0) THEN -- less than or
     EXIT;              -- equal to
   ELSE
     int_a := int_a -1;
     q(i) <= 3.1416 / REAL(int_a * i); -- signal
   END IF;                             -- assign
  END LOOP;

  y <= q;

END PROCESS;
```

Assert statement

- Added by developer for verification
  - Very useful statement for reporting textual strings to the designer
  - Informing the user about what is currently happening in the model
- ASSERT statement checks the value of a boolean expression for true or false
  - True: do nothing
  - False: output a text output specified by user

Assert statement

- Severity levels:
  - Note
  - Warning
  - Error
  - Failure

```
ASSERT assertion_condition REPORT "reporting_message"
SEVERITY severity_level;
```

```
MAKE SURE THAT assertion_condition IS TRUE,
OTHERWISE REPORT "reporting_message"
AND TAKE THE ACTION AT THIS severity_level;
```

- Simulators may handle the severity levels differently
  - Note and warning often result in a message
  - Failure and error often result in stopping the simulation

Assert statement

- An SR flipflop using assert

```
ARCHITECTURE behavioral OF d_sr_flipflop IS
  SIGNAL state : BIT := '0';
BEGIN
  dff: PROCESS (rst, set, clk)
  BEGIN
    ASSERT
      (NOT (set = '1' AND rst = '1'))
    REPORT
      "set and rst are both 1"
    SEVERITY NOTE;
    IF set = '1' THEN
      state <= '1' AFTER sq_delay;
    ELSIF rst = '1' THEN
      state <= '0' AFTER rq_delay;
    ELSIF clk = '1' AND clk'EVENT THEN
      state <= d AFTER cq_delay;
    END IF;
  END PROCESS dff;
  q <= state;
  qb <= NOT state;
END behavioral;
```

Assert statement

- ASSERT can also be written in the architecture concurrent body
- The concurrent ASSERT statement executes whenever any signals that exist inside of the condition expression have an event upon them
- The ASSERT statement is often ignored by synthesis tools
  - Assertion synthesis methods in some research work

Wait statement

- Another way to control the timing of the sequential body
  - Suspends the sequential execution of a process
- The conditions for resuming execution by 3 means:
  - WAIT ON signal changes
  - WAIT UNTIL an expression is true
  - WAIT FOR a specific amount of time

```
WAIT UNTIL (( x * 10 ) < 100 );

WAIT ON a, b;

WAIT FOR 10 ns;
```

Wait statement

- A D-FF

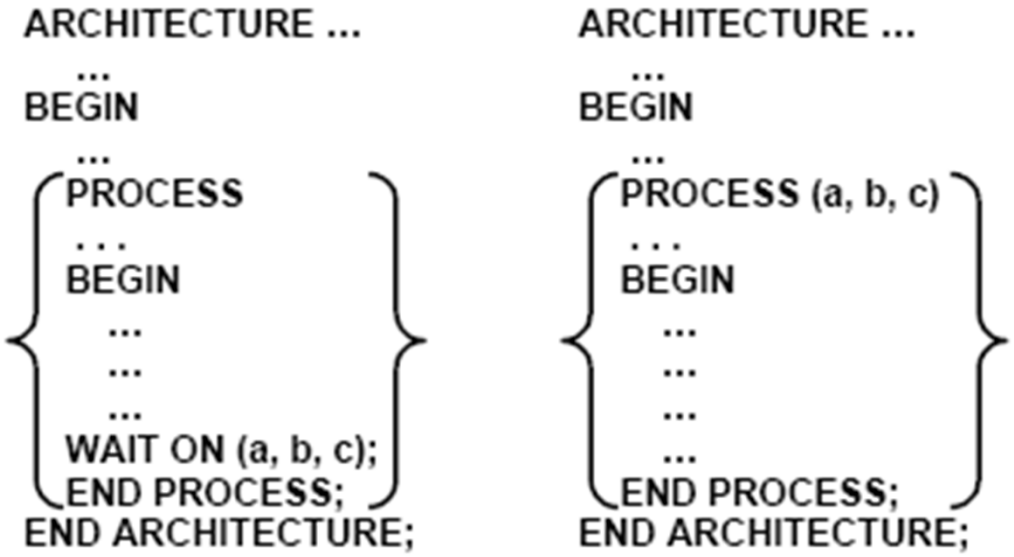
```
PROCESS
BEGIN
   WAIT UNTIL clock = '1' AND clock'EVENT;
   q <= d;
END PROCESS;
```

- A clock generator

```
PROCESS
BEGIN
    clock= NOT clock;
    WAIT for 10 ns;
END
```

Wait statement

- WAIT ON at the end of a process is equivalent to sensitivity list



- WAIT and sensitivity list cannot be used together

Sensitivity list vs *Wait*

```
PROCESS (clk)
  VARIABLE last_clk : std_logic := 'X';
BEGIN
  IF (clk /= last_clk ) AND (clk = '1') THEN
   q <= din AFTER 25 ns;
END IF;

last_clk := clk;

END PROCESS;
```

```
PROCESS
  VARIABLE last_clk : std_logic := 'X';
BEGIN
  IF (clk /= last_clk ) AND (clk = '1') THEN
   q <= din AFTER 25 ns;
  END IF;

  last_clk := clk;

  WAIT ON clk;
END PROCESS;
```

- The WAIT statement at the end of the process is equivalent to the sensitivity list at the beginning of the process
- Why is the WAIT statement at the end of the process and not at the beginning?
  - During initialization of the simulator, all processes are executed once
  - To allow the PROCESS statement to execute once like a process with a sensitivity list

Signal assignment problem in process

Recall:

- This code doesn't work
- Why?

```
ENTITY mux IS
  PORT (I0, I1, I2, I3, A, B : IN std_logic;
        Q : OUT std_logic);
END mux;

ARCHITECTURE mux_behave OF mux IS
  SIGNAL sel : INTEGER RANGE 0 TO 3;
BEGIN
  B : PROCESS(A, B, I0, I1, I2, I3)
  BEGIN

   sel <= 0;
   IF (A = '1') THEN sel <= sel + 1; END IF;
   IF (B = '1') THEN sel <= sel + 2; END IF;

   CASE sel IS
     WHEN 0 =>
       Q <= I0;
     WHEN 1 =>
       Q <= I1;
     WHEN 2 =>
       Q <= I2;
     WHEN 3 =>
       Q <= I3;
   END CASE;
  END PROCESS;
END mux_behave;
```

Signal assignment problem in process

- This code works correctly
- "*WAIT for 0 ns*" makes a delta delay
  - Signal assignment is done in one delta: allows the signal value to be updated

```
ARCHITECTURE mux_fix1 OF mux IS
  SIGNAL sel : INTEGER RANGE 0 TO 3;
BEGIN
  PROCESS
  BEGIN
   sel <= 0;
   WAIT FOR 0 ns;  -- or wait on sel

   IF (a = '1') THEN sel <= sel + 1; END IF;
   WAIT for 0 ns;

   IF (b = '1') THEN sel <= sel + 2; END IF;
   WAIT FOR 0 ns;

   CASE sel IS
     WHEN 0 =>
       Q <= I0;
     WHEN 1 =>
       Q <= I1;
     WHEN 2 =>
       Q <= I2;
     WHEN 3 =>
       Q <= I3;
   END CASE;

   WAIT ON A, B, I0, I1, I2, I3;
  END PROCESS;
END mux_fix1;
```

Signal assignment problem in process

- Why we have a WAIT statement at the end of the process?

WAIT ON A, B, I0, I1, I2, I3;

- Sensitivity list is not allowed

```
ARCHITECTURE mux_fix1 OF mux IS
  SIGNAL sel : INTEGER RANGE 0 TO 3;
BEGIN
  PROCESS
  BEGIN
   sel <= 0;
   WAIT FOR 0 ns;  -- or wait on sel

   IF (a = '1') THEN sel <= sel + 1; END IF;
   WAIT for 0 ns;

   IF (b = '1') THEN sel <= sel + 2; END IF;
   WAIT FOR 0 ns;

   CASE sel IS
     WHEN 0 =>
       Q <= I0;
     WHEN 1 =>
       Q <= I1;
     WHEN 2 =>
       Q <= I2;
     WHEN 3 =>
       Q <= I3;
   END CASE;

   WAIT ON A, B, I0, I1, I2, I3;
  END PROCESS;
END mux_fix1;
```

Timing analysis of a *Process*

```
ARCHITECTURE average_delay_behavioral OF d_sr_flipflop IS
BEGIN
  dff: PROCESS (rst, set, clk)
    VARIABLE state : BIT := '0';
  BEGIN
    IF set = '1' THEN
      state := '1';
    ELSIF rst = '1' THEN
      state := '0';
    ELSIF clk = '1' AND clk'EVENT THEN
      state := d;
    END IF;
    q <= state AFTER (sq_delay + rq_delay + cq_delay) /3;
    qb <= NOT state AFTER (sq_delay + rq_delay + cq_delay) /3;
  END PROCESS dff;
END average_delay_behavioral;
```

Timing analysis of a *Process*

```
ARCHITECTURE average_delay_behavioral OF d_sr_flipflop IS
BEGIN
  dff: PROCESS (rst, set, clk)
    VARIABLE state : BIT := '0';
  BEGIN
    IF set = '1' THEN
      state := '1';
    ELSIF rst = '1' THEN
      state := '0';
    ELSIF clk = '1' AND clk'EVENT THEN
      state := d;
    END IF;
    q <= state AFTER (sq_delay + rq_delay + cq_delay) /3;
    qb <= NOT state AFTER (sq_delay + rq_delay + cq_delay) /3;
  END PROCESS dff;
END average_delay_behavioral;
```

TIME (NS)	ss	rr	cc	dd	q1	q2	qb1	qb2
0	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'
+1δ	...	...	...	...	...	...	'1'	...
6	...	...	...	...	...	...	...	'1'
200	'1'	...	...	...	...	...	...	...
206	...	...	...	...	...	'1'	...	'0'
+1δ	...	...	...	...	'1'	...	'0'	...
500	...	...	'1'	...	...	...	...	...
1000	...	...	'0'	...	...	...	...	...
1200	'0'	...	...	...	...	...	...	...
1400	...	'1'	...	...	...	...	...	...
1406	...	...	...	...	...	'0'	...	'1'
+1δ	...	...	...	...	'0'	...	'1'	...
1500	...	...	'1'	...	...	...	...	...
2000	...	...	'0'	...	...	...	...	...
2200	...	'0'	...	...	...	...	...	...
2400	...	...	...	'1'	...	...	...	...
2500	...	...	'1'	...	...	...	...	...
2506	...	...	...	...	...	'1'	...	'0'
+1δ	...	...	...	...	'1'	...	'0'	...
3000	...	...	'0'	...	...	...	...	...
3300	...	...	...	'0'	...	...	...	...
3500	...	...	'1'	...	...	...	...	...
3506	...	...	...	...	...	'0'	...	'1'
+1δ	...	...	...	...	'0'	...	'1'	...
4000	...	...	'0'	...	...	...	...	...

Postponed process

- Wait until the last event in a real time increment