EE 459/500 – HDL Based Digital Design with Programmable Logic

Lecture 7
Sequential circuits II

Read before class:

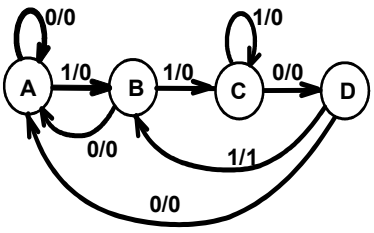
Chapter 2 from textbook

Overview

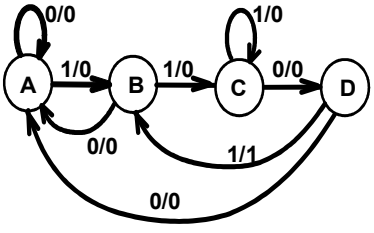
- Sequential circuits
  - Finite State Machines (continued)
- More on sequential statements
  - IF-THEN-ELSE Statement
  - CASE Statement
  - LOOP Statement
  - WAIT Statement
  - ASSERT Statement

VHDL code using 3 processes: sequential recognizer

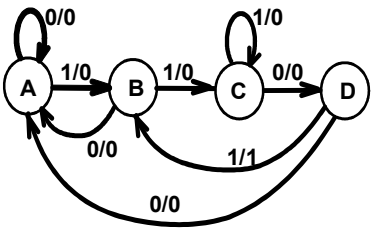
```
library ieee;
use ieee.std_logic_1164.all;

entity seq_rec_MEALY is
 port (CLK, RESET, X: in std_logic;
       Z: out std_logic);
end seq_rec;

architecture process_3 of seq_rec_MEALY is
  type state_type is (A, B, C, D);
  signal state, next_state: state_type;

begin
```



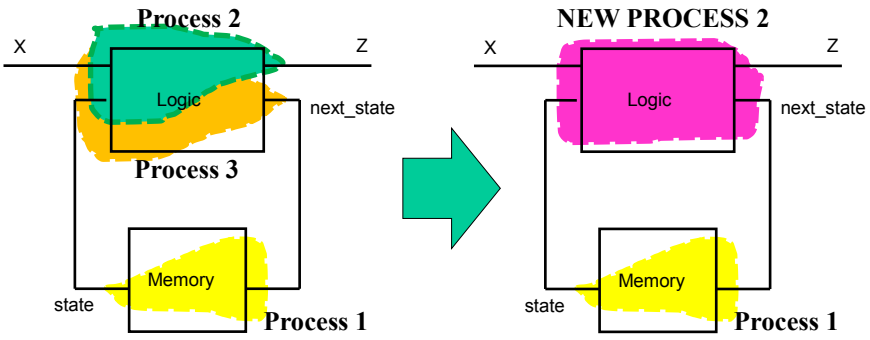

```
-- process 1: implements positive edge-triggered
-- flipflop with asynchronous reset
state_register: process (CLK, RESET)
begin
 if (RESET = '1') then
  state <= A;
 elsif (CLK'event and CLK = '1') then
  state <= next_state;
 end if;
end process;

-- process 2: implement output as function
-- of input X and state
output_function: process (X, state)
begin
 case state is
  when A => Z <= '0';
  when B => Z <= '0';
  when C => Z <= '0';
  when D => if X = '1' then Z <= '1';
            else Z <= '0';
            end if;
  end case;
end process;
```



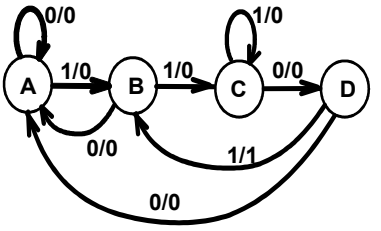
```
-- process 3: next state-function implemented
-- as a function of input X and state
next_state_function: process (X, state)
begin
 case state is
   when A =>
     if X = '1' then next_state <= B;
     else next_state <= A;
     end if;
   when B =>
     if X = '1' then next_state <= C;
     else next_state <= A;
     end if;
   when C =>
     if X = '1' then next_state <= C;
     else next_state <= D;
     end if;
   when D =>
     if X = '1' then next_state <= B;
     else next_state <= A;
     end if;
 end case;
end process;

end architecture;
```



VHDL code using 2 processes: combine processes 2 & 3



```
-- NEW PROCESS 2: Z and next_state implemented
-- as functions of input X and state
X_and_next_state_functions: process (X, state)
begin
 case state is
   when A =>
     Z <= '0';
     if X = '1' then next_state <= B;
     else next_state <= A;
     end if;
   when B =>
     Z <= '0';
     if X = '1' then next_state <= C;
     else next_state <= A;
     end if;
   when C =>
     Z <= '0';
     if X = '1' then next_state <= C;
     else next_state <= D;
     end if;
   when D =>
     if X = '1' then Z <= '1'; next_state <= B;
     else Z <= '0'; next_state <= A;
     end if;
 end case;
end process;
```



Overview

- Sequential circuits
  - Finite State Machines
- More on sequential statements
  - IF-THEN-ELSE Statement
  - CASE Statement
  - LOOP Statement
  - WAIT Statement
  - ASSERT Statement

1. IF Statement

```
if CONDITION then
   -- sequential statements
end if;

if CONDITION then
  -- sequential statements
else
  -- sequential statements
end if;

if CONDITION then
   -- sequential statements
elsif CONDITION then
  -- sequential statements
   . . .
else
  -- sequential statements
end if;
```

- Condition is a boolean expression
- Optional elsif sequence
  - Conditions may overlap
  - priority
- Optional else path
  - executed, if all conditions evaluate to false

Example 1

Write VHDL IF-THEN-ELSE code to model a D Flip-flop (input and output D and Q, respectively)

```
IF (clock'event and clock = 1)  THEN
     Q <=  D AFTER 5 ns;
END IF;
```

Example 2:

Clocked 4-to-1 MUX

```
ENTITY clocked_mux IS
 PORT (inputs  : IN BIT_VECTOR (0 to 3);
       sel     : IN BIT_VECTOR (0 to 1);
       clk     : IN BIT;
       output  : OUT BIT);
END clocked_mux;

ARCHITECTURE example OF clocked_mux IS
BEGIN
      PROCESS (clk)
            VARIABLE temp : BIT;
       BEGIN
            IF (clk = '1') THEN
                 IF  sel = "00" THEN
                     temp := inputs (0)
                 ELSIF sel = "01" THEN
                     temp := inputs(1)
                 ELSIF sel = "10" THEN
                     temp := inputs (2)
                 ELSE
                     temp := inputs(3)
                 END IF;

                 output <= temp AFTER 5 ns;
            END IF;
       END PROCESS;
END example;
```

Example 3

```
entity IFSTMT is
    port (A, B, C, X : in bit_vector (3 downto 0);
          Z : out bit_vector (3 downto 0));
end IFSTMT;

architecture EX1 of IFSTMT is
begin
    process (A, B, C, X)
      begin
        Z <= A;
        if (X = "1111") then
          Z <= B;
        elsif (X > "1000") then
          Z <= C;
        end if;
    end process;
end EX1;
```

=

```
architecture EX2 of IFSTMT is
begin
    process (A, B, C, X)
      begin
        if (X = "1111") then
          Z <= B;
        elsif (X > "1000") then
          Z <= C;
        else
          Z <= A;
        end if;
    end process;
end EX2;
```

2. Case Statement

```
CASE expression IS

    WHEN constant_value => sequential statements
    WHEN constant_value => sequential statements
    WHEN others => sequential statements

END CASE;
```

- The keyword **WHEN** is used to identify constant values that the expression might match. The expression evaluates a choice, and then the associated statements will be executed.
- The **CASE** statement will exit when all statements associated with the first matching constant value are executed.

Example 1

```
CASE vect IS
       WHEN "00" => int := 0;
       WHEN "01" => int := 1;
       WHEN "10" => int := 2;
       WHEN "11" => int := 3;
END CASE;
```

- vect is a two element bit-vector. By evaluating vect and the matching **WHEN** value or choice causes the variable int to be assigned the matching integer value.

Example 2: Clocked 4-to-1 MUX

```
ENTITY clocked_mux IS
      PORT ( inputs : IN BIT_VECTOR (0 to 3);
             sel    : IN BIT_VECTOR (0 to 1);
             clk    : IN BIT;
             output : OUT BIT);
END clocked_mux;

ARCHITECTURE behave OF clocked_mux IS
  BEGIN
       PROCESS (clk)
           VARIABLE temp : BIT;
           BEGIN
                CASE clk IS
                       WHEN '1' =>
                           CASE sel  IS
                                WHEN "00" => temp := inputs(0);
                                WHEN "01" => temp := inputs(1);
                                WHEN "10" => temp := inputs(2);
                                WHEN "11" => temp := inputs(3);
                           END CASE;
                           output <= temp AFTER 5 ns;
                       WHEN OTHERS => NULL;
                 END CASE;
       END PROCESS;
  END behave;
```

Example 3

```
entity CASE_STATEMENT is
      port (A, B, C, X : in integer range 0 to 15;
            Z : out integer range 0 to 15;
end CASE_STATEMENT;
architecture EXAMPLE of CASE_STATEMENT is
begin
   process (A, B, C, X)
    begin
     case X is
       when 0 =>
                    Z <= A;
       when 7 | 9 =>
                    Z <= B;
       when 1 to 5 =>
                    Z <= C;
       when others =>
                    Z <= 0;
    end case;
  end process;
end EXAMPLE;
```

3. Loop Statement

- The LOOP statement provides a mechanism to repeatedly execute a sequence of statements. VHDL provides two types of loop statements:
  - FOR LOOP
  - WHILE LOOP

FOR LOOP Statement

- *FOR LOOP syntax:*

```
[loop_label :]
   FOR variable_name IN range LOOP
           sequential_statements
   END LOOP [loop_label];
```

- The sequential_statements within the loop will be repeatedly executed within the range specified.

Example

```
FOR i IN 0 to 3 LOOP
    IF vect(i) = '1' THEN
        value := value + 2**i;
    ENDIF;
END LOOP;
```

- After the fourth pass, the loop range will be exceeded and the loop will terminate.
- A feature of VHDL: unlike most programming languages, the range variable i was not declared. Any range variable used within the FOR construct does not have to be declared. The same range identifier can be used repeatedly from one loop statement to the next.

WHILE LOOP Statement

- *WHILE LOOP Syntax:*

```
[loop_label :]
  WHILE boolean_expression LOOP
          sequential_statements
  END LOOP [loop_label];
```

- The `boolean_expression` condition is evaluated, and if it is true the sequential_statements within the loop statement are evaluated until the condition is no longer true.

NEXT & EXIT Loop Termination Statements

- `NEXT` and `EXIT` statements can be used inside the loop statement
  - NEXT: terminate a loop iteration
  - EXIT: completely terminate the loop statement

4. Sensitivity List vs. Wait Statement

- The process statement contains only one sensitivity list. A process with a sensitivity list can only be triggered by an event on a signal in the list.
- Once triggered, the process will sequentially execute all of statements in the statement region and then suspend until another event is detected on those signals.
- If multiple signals are included in the sensitivity list, any one of those signals in the list can trigger the process. Therefore, the use of sensitivity list in a process is fairly limited.
  - To provide greater flexibility for the control of execution of a process, a **`WAIT`** statement can be used.

Wait Statement

- The **WAIT** statement provides the user with more options than the process sensitivity list.

- **Advantage:**
  - It can be placed anywhere within the process body.
  - With the process sensitivity list the process suspends at the end of the process.
  - With the **WAIT** statement, the suspension occurs where a **WAIT** statement is encountered.
  - There is no limitation to the number of **WAIT** statements within a process.
  - **WAIT** statements are more flexible.

Wait Statement

- **WAIT** statements stop the process execution.
- Four types of wait statements:
  - wait on signal_list; -- wait for a signal event

```
WAIT ON clock, clear, reset, D;
```

  - wait until condition; -- wait for true condition (requires an event)

```
WAIT UNTIL (clock = '1');
WAIT UNTIL (clock ='1') or (clear = '0');
```

  - wait for specific_time; -- wait for a specific time

```
WAIT FOR 10ns;
```

  - wait; -- indefinite (process is never reactivated)
- Wait statements must not be used in processes with sensitivity list

Example

```
WAIT ON clock UNTIL (clear='0') FOR 10 ns;
```

- This is a combination of three types of WAIT statements. In this example, the wait statement will suspend the process and resume if:
  - Simulation time has advanced 10 ns or
  - There is an event on clock and

    The Boolean expression clear = 0 is true

Sensitivity List & Wait Statement

A process with sensitivity is functionally equivalent to a process statement with a WAIT statement as the last statement within the process.

```
PROCESS (clk)
BEGIN
  clk <= NOT (clk) AFTER 50ns;
END PROCESS;
```

```
PROCESS
BEGIN
  clk <= NOT (clk) AFTER 50ns;
  WAIT ON clk;
END PROCESS;
```

If a process does not have a sensitivity list and does not have a WAIT statement contained within it, the process will loop forever during initialization.

This is important to remember !

Example: D Flip-Flop Model

```
entity FF is
    port (D, CLK : in bit;
          Q : out bit);
end FF;
```

```
architecture BEH_1 of FF is
  begin
     process
       begin
         wait on CLK;
         if (CLK='1') then
           Q <= D;
         end if;
     end process;
  end BEH_1;
```

=

```
architecture BEH_2 of FF is
begin
    process
      begin
        wait until CLK='1';
          Q <= D;
    end process;
end BEH_2;
```

Example: Stimuli Generation in Testbenches

```
STIMULUS: process
  begin
     SEL <= `0`;
     BUS_B <= "0000";
     BUS_A <= "1111";
     wait for 10 ns;

     SEL <= `1`;
     wait for 10 ns;

     SEL <= `0`;
     wait for 10 ns;

     wait;
  end process STIMULUS;
```

- Via '**wait for**' construct it is very easy to generate simple input patterns for design verification purposes.
- Wait for constructs are excellent tool for describing timing specifications.

WAIT Statements and Behavioral Modeling

```
READ_CPU : process
begin
       wait until CPU_DATA_VALID = `1`;
          CPU_DATA_READ <= `1`;
       wait for 20 ns;
          LOCAL_BUFFER <= CPU_DATA;
       wait for 10 ns;
          CPU_DATA_READ <= `0`;
end process READ_CPU;
```

- It is easy to implement a bus protocol for simulation.
- This behavioral modeling can only be used for simulation purposes as it is definitely not synthesizable!

5. Assertion Statement

- Check that expected conditions are met within the model
- Both concurrent and sequential statement, can be included anywhere in a process body
- `[label:] ASSERT boolean_expression`
  `[REPORT expression]`
  `[SEVERITY severity_level];`
- Severity_level: predefined enumeration type
  - TYPE severity_level IS (note, warning, error, failure)

Example

```
assert (last_position-first_position + 1) =
  number_of_entries
report "inconsistency in buffer model"
severity failure;
```

- Both **report** and **severity** clauses are optional
  - Default report string is: "Assertion violation"
  - Default severity level is: error

Concurrent Assertion Statement Example

```
architecture functional of S_R_flipflop is

begin
  q<='1' when s='1' else
     '0' when r='1';
  q_n<='0' when s='1' else
       '1' when r='1';

  check: assert not (s='1' and r='1')
         report "Incorrect use of S_R_flip_flop:
                 s and r both '1'";
end architecture functional;
```

Summary

- FSM description with two processes is the most popular
- Wait statements offer more flexibility; very useful to construct testbenches
- Assert statement useful for debugging and to construct testbenches