Behavioral Modeling and Timing Constraints

Introduction

Behavioral modeling was introduced in Lab 1 as one of three widely used modeling styles. Additional capabilities with respect to testbenches were further introduced in Lab 4. However, there more constructs available in this modeling style which are particularly useful for complex sequential digital circuits design. Sequential circuits require clocking, and with clocking there is always a frequency or speed involved at which the circuit can be run. The expected speed can be communicated to the tools through specific timing constraints via the .ucf file. In this lab you will learn more language constructs and timing constraints concepts. *Please refer to the PlanAhead tutorial on how to use the PlanAhead tool for creating projects and verifying digital circuits*.

Objectives

After completing this lab, you will be able to:
- Use various language constructs using behavioral modeling
- Communicate timing expectations through timing constraints

Behavioral Modeling Part 1

As mentioned in previous labs, the primary mechanism through which the behavior of a design can be modeled is via the `process` statement. The `process` statement is used in testbenches and behavioral code to describe the functionality of the circuit. The `process` statement must start with a **`begin`** and **`end`** `process` to denote the beginning and end of the procedural statement(s).

A procedural statement is one of:
procedural_assignment (blocking or non-blocking)
conditional_statement
case_statement
loop_statement
wait_statement
event_trigger
sequential_block
task (user or system)

When multiple procedural statements are enclosed between begin … end, they execute sequentially. Since a `process` statement executes continuously, they are typically controlled using either delay control or event control mechanisms. Here is an example of a delay controlled procedural statement:

```
process begin
   wait for 5 ns; CLK <= not(CLK);
end process;
```

In the above example, the statement will execute after every 5 units of time (nanoseconds) specified in the VHDL code, inverting the signal value every time it executes, thus generating a clock of 10 ns period. This is considered a delay control, meaning the time delay between the statement encountered and actually executed is 5 time units. VHDL supports delays in the form of the `wait for` statement. When the wait for statement is inserted between two statements, it forces the test bench to wait for a specified period of time before executing the next statement.

```
process begin
      target <= ‘1’;
      wait for 20 ns;
      target <= ‘0’;
end process;
```

Below is the example that illustrates the effect of the inter-statement delay:

```
process begin
      wait for 5 ns; SIG1 <= 3;
      wait for 4 ns; SIG1 <= 7;
      wait for 2 ns; SIG1 <= 4;
end process;
```

The SIG1 signal will get the value of 3 at 5 ns, value of 7 at 9 ns, and value of 4 at 11 ns.

```
signal test : std_logic;
process(test) begin
      wait for 5 ns;
      CLK <= not(CLK);
end process;
```

The above `process` statement will execute only when there is a change in value (an event) on a signal *test*. The change in value can be 0 -> 1, 1 -> 0, 0 -> x, x->1, x -> 0, 1 -> z, z -> 0, 0 -> z, z -> 1 or 1->x. When the event occurs, the logical value of CLK will be flipped after 5 ns.

```
signal test : std_logic;
process(test) begin
      if rising_edge(test) then
            wait for 5 ns;
            CLK <= not(CLK);
      end if;
end process;
```

The above `process` statement will execute only when there is a rising edge change (0 -> 1, 0 -> x, 0 -> z, z->1, x->1) in value on a signal *test*. When the event occurs, the logical value of CLK will be flipped after 5 ns. Such events are called edge-triggered events. In contrast to edge-triggered events, there can be another type of event called a level-sensitive event control.

```
wait until (SUM > 22)
   SUM <= 0;

wait until (DATA_READY = ‘1’)
   DATA <= BUS;
```

In the above examples, SUM is assigned 0 only when SUM is greater than 22, and DATA is assigned whatever the value is on BUS when DATA_READY is asserted.

The “<=” assignment operator in testbench procedural assignment statements are used as blocking procedural assignments. As the name indicates, the subsequent statement is blocked until the current assignment is done. The statements are executed sequentially. Here is an example that explains the concept:

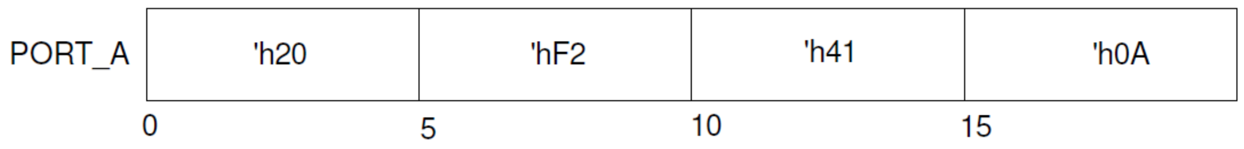
```
signal T1, T2, T3 : std_logic;
process(A, B, CIN) begin
   T1 <= A and B;
   T2 <= B and CIN;
   T3 <= A and CIN;
end process;
```

The T1 assignment occurs first, T1 is computed, then the second statement is executed, T2 is assigned and then the third statement is executed and T3 is assigned Here is another of a blocking example.

```
process begin
   wait for 5 ns;
   CLR <= 0;
   wait for 4 ns;
   CLR <= 1;
   wait for 10 ns;
   CLR <= 0;
end process;
```

In RTL VHDL code, the assignment operator "<=" used is called non-blocking. The statement that uses the non-blocking operator does not block the execution; however the assignment is scheduled to occur in the future. When the non-blocking assignment is executed, the right-hand side expression is evaluated at that time and its value is scheduled to be assigned to the left-hand side target and the execution continues with the next statement. The non-blocking statements are widely used for content transfer across multiple registers (often in parallel) when a desired clock event occurs.

1-1. Write a testbench using delays to produce the following waveform.



So far we have seen constructs which allow generation of stimulus unconditionally. However, many times we like to have different stimulus generation upon certain conditions. VHDL provides control altering statements such as **if, if … else,** and **if … elsif**. All variations of the **if** statement need to end with an **end if**. The general syntax of an **if** statement is:

```
if (condition-1) then
  procedural_statement
[ elsif (condition-2) then
  procedural_statement ]
[ else
  procedural_statement ]

end if;
```

A begin…end block is not necessary for **if** statements. Any statements between **if, if … else,** and **if … elsif** are considered to be part of the block

It is possible to have nested **if** statements. In such case, the **else** part is associated to the closest **if** part. For example, below, the else part is associated to the **if** (RESET) condition,

```
if (CLK = '1') then
  if (RESET = '1') then
      Q <= 0;
  else
      Q <= D;
 end if;
end if;
```

The if statement is commonly used to create a priority structure, giving higher priority to the condition listed first.

1-2. Write a behavioral model to design a 1-bit 4-to-1 mux using the if-else-if statement. Develop a testbench to verify the design. Assign the four input channels to SW3-SW0 (SW3 is assigned to most significant channel and will have the lowest priority, and so on) and select lines to SW5-SW4 and output to LED0. Verify your design in the hardware using the Nexys3 board. Look at the Project Summary report and make sure that no latches or registers resources are used or inferred.

Another widely used statement is a `case` statement. The `case` statement is generally used when we want to create a parallel structure (unlike priority). Case statements are commonly used in creating finite state machines. The syntax of the `case` statement is:

```
case [ case_expression] is
    when case_item_expression => procedural_statement;
    …
    …
    when others => procedural_statement;
end case;
```

The case_expression is evaluated first (whenever there is an event on it), and the value is matched with the case_item_expr in the order they are listed. When the match occurs, the corresponding procedural_statement is executed. Like the **if** statement, multiple procedural statements are enclosed between **"=>"** and the next **"when"** statement, no begin … end blocks are needed. The others case covers all values that are not covered by any of the case_item_expr.

1-3. Design a gray code generator using the case statement. The design will take a 4-bit BCD input through SW3-SW0 and will output the corresponding gray code value on the four LEDS, LED3-LED0, provided that the enable input on SW4 is TRUE. If the enable input is FALSE or the input is not BCD then LED3-LED0 should all be turned ON and LED4 should also be turned ON. Look at the Project Summary report and make sure that no latches or registers resources are used.

VHDL testbenches also support various loop statements to do the same function a number of times. The supported loop statements are:

basic loop
`while` loop
`for` loop

The basic loop statement is used when the procedural statement(s) need to be executed continuously. Some kind of timing control must be used within the procedural statement if a periodic output is desired. For example, to generate a clock of 20 units period, the following code can be used:

```
signal CLK : std_logic := '0';
process begin
  loop
      wait 10 ns;
      CLK <= not(CLK);
  end loop;
 end process;
```

Notice you need an end loop statement to terminate the loop.

The `while` loop statement's procedural statement(s) are executed until certain conditions become false.

```
while (COUNT < COUNT_LIMIT) loop
   SUM <= SUM +5;
end loop;
```

The `for` loop statement is used when the procedural statement(s) need to be executed for a specified number of times. An index variable is used which can be initialized to any desired value, it can be further updated by whatever value is required, and a condition can be given to terminate the loop statement. The loop index variable is normally defined as an integer type. Here is an example of the loop statement.

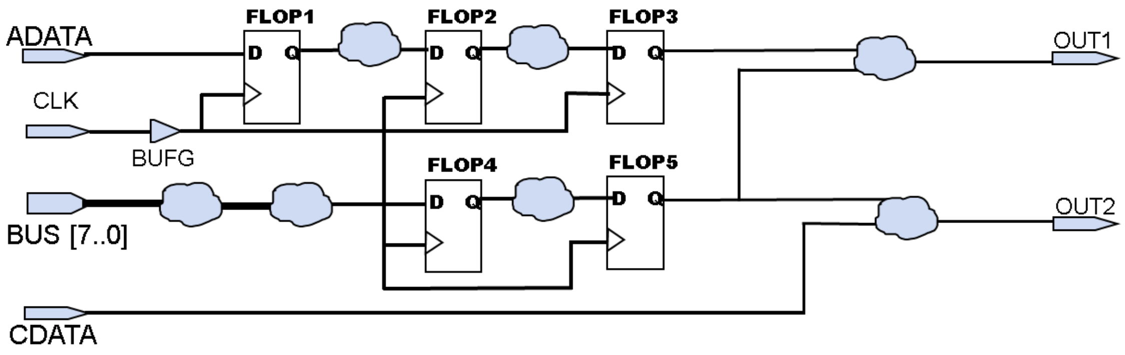
```
signal K : integer := '0';
for K in 0 to 15 loop
   SUM <= SUM + K;
end loop;
```

1-4. Write a model of a counter which counts in the sequence mentioned below. The counter should use behavioral modeling and a case statement. Develop a testbench to test it. The testbench should display the counter output in the simulator console output. Simulate using the clock period of 10 units for 200 ns. 000, 001, 011, 101, 111, 010, (repeat 000). The counter will have an enable signal (SW2), a reset signal (SW1), and a clock signal (SW0). The output of the counter will be on LED2-LED0.

Timing Constraints Part 2

In combinatorial logic design, delays through the circuits will depend on the number of logic levels, the fan-out (number of gate inputs a net drives) on each net, and the capacitive loading on the output nets. When such circuits are placed between flip-flops or registers, they affect the clock speeds at which sequential designs can be operated. The synthesis and implementation tools will pack the design in LUT, flip-flops, and registers, as well as place them if the expected performance is communicated to them via timing constraints. Timing constraints can be categorized into global timing or path specific constraints. The path specific constraints have higher priority over global timing constraints, and the components which are used in those specific paths are placed and routed first.

The global timing constraints cover most of the design with very few lines of instructions. In any pure combinatorial design, the path-to-path constraint is used to describe the delay the circuit can tolerate. In sequential circuits, period, offset-In, and offset-out constraints are used. All four kinds of the timing constraints are shown in the figure below.



In the above figure, the paths which are covered between ADATA input and D port of FLOP1, BUS input and D port of FLOP4 can be constrained by a constraint called OFFSET IN. The paths between the port Q of FLOP3 and output OUT1, Q port of FLOP5 and OUT1, Q port of FLOP5 and OUT2 can be constrained by OFFSET OUT. The paths between CDATA and OUT2 can be constrained by PAD to PAD constraint. The paths between Q port of FLOP1 and D port of FLOP2, Q port of FLOP2 and D port of FLOP3, Q port of FLOP4 and D port of FLOP5 can be constrained by PERIOD constraint. The syntax of each type of constraints are given below:

```
NET "CLK" TNM_NET = CLK;
TIMESPEC TS_CLK = PERIOD "CLK" 10 ns HIGH 50%;
OFFSET = IN 7 ns VALID 10 ns BEFORE "CLK" RISING;
OFFSET = OUT 15 ns AFTER "CLK";
TIMESPEC TS_PAD = FROM  PADS("CDATA") TO  PADS("OUT2]") 5 ns;
```

Conclusion

In this lab you learned about various constructs available in behavioral modeling. You also learned about blocking and non-blocking operators as well as concepts and the need of timing constraints. Providing the timing constraints to the implementation tools the generated output can be made to meet the design's timing specifications.