Lab 7 (All Sections) Prelab: Introduction to Verilog

Name:

Sign the following statement:

On my honor, as an Aggie, I have neither given nor received unauthorized aid on this academic work.

1 Objective

The objective of this lab is to gain further experience with Verilog HDL and VCS. In particular, you will explore procedural Verilog, and simulation timing constructs. For this lab you are expected to have some knowledge of Verilog programming. You are advised to go through the Verilog tutorial posted on the course website.

2 Procedural Blocks

In digital design there are two types of elements: combinational elements and sequential elements. Verilog provides mechanisms to model these elements using the always, assign and initial statements:

- Combinational elements can be modeled using assign and always statements.
- Sequential elements can be modeled using only always statements.
- Non-synthesizable constructs can use initial statements.
2.1 Initial statements

An initial block as the name suggests, is executed only once, at the beginning of the simulation. This is usually useful for writing test benches. If we have multiple initial blocks then all of them are executed in parallel at the beginning of the simulation.

\[
\text{initial begin} \\
\text{clk} = 0; \\
\text{reset} = 0; \\
\text{req.0} = 0; \\
\text{req.1} = 0; \\
\text{end}
\]

In the above example, at the beginning of simulation, (i.e., at time zero), all the variables inside the block marked by `begin` and `end` are driven zero.

2.2 Always statements

As the name suggests, an `always` block executes periodically throughout the program, unlike `initial` blocks which are executed only once. A second difference is that an `always` block should have a sensitivity list or a delay associated with it. The sensitivity list specifies the conditions which trigger the execution the `always` block.

\[
\text{always} @ (a \text{ or } b \text{ or } sel) \\
\text{begin} \\
y = 0; \\
\text{if} (sel == 0) \text{ begin} \\
y = a; \\
\text{end else begin} \\
y = b; \\
\text{end} \\
\text{end}
\]

As shown above, the `@` symbol after reserved word `always` indicates that the block will be executed whenever the condition in parenthesis after symbol `@` is satisfied. In this case whenever the variables `a` or `b` or `sel` are modified the `always` block is executed.

Note that `always` blocks cannot drive `wire` data type, but instead can drive `reg` data type. In addition, we note that the statements that need to be executed sequentially must be enclosed inside a begin-end block.

2.3 Continuous assignments

The Verilog includes the `continuous assign` construct, which usually represent the combinatorial logic. For example, consider the following assignment:

\[
\text{assign } x = y \& z
\]

This assignment describes an “AND” gate whose inputs are connected to wires `y` and `z` and whose output is connected to wire `x`. In continuous assignments, the left-hand size of
the expression must be a wire. As a rule, the register are given values at discrete times, while the nets are always given a value.

The continuous assignments often use the following conditional operator:

\[
\text{condition} \ ? \ \text{expression1} \ : \ \text{expression2}
\]

The condition is the logical expression which has the true or false result. If the result is true, expression1 is evaluated, otherwise expression2 is evaluated. For example, a multiplexor with input signals \( \text{in0}, \text{in1} \), output signal \( \text{out} \), and control signal \( s \) can be described by the following statement:

\[
\text{assign out} = s \ ? \ \text{in1} \ : \ \text{in0} ;
\]

### 3 Timing Control

The timing control is an important and useful feature provided by Verilog. Timing control is needed to simulate propagation or switching delay in chips/gates or wires. Timing control is also used in testbench code to simulate timing delays to produce input waveforms.

The Verilog language provides three types of explicit timing control. The first type is a delay control in which an expression specifies the time duration between initially encountering the statement and when the statement actually executes. The second type of timing control is the event expression, which allows statement execution. The third type is a wait statement which suspends execution until a certain condition is met. Simulated time can only progress by one of the following:

1. Gate or wire delay, if specified;
2. A delay control, introduced by the \(#\) symbol;
3. An event control, introduced by the \(@\) symbol;
4. The wait statement.

#### 3.1 Delay Control (\#)

A delay control expression specifies the time duration between initially encountering the statement and when the statement actually executes. For example:

\[
\#10 \ A = A + 1 ;
\]

Specifies to delay 10 time ticks before executing the procedural assignment statement. The \# may be followed by an expression with variables.
### 3.1.1 Timescale

In Verilog there are no default units of for the “ticks”, instead they are typically defined explicitly in the testbench for each particular design with a “timescale” Verilog directive thus:

```
timescale time_unit base / precision base
```

This command specifies the time units and precision of time tick delays:

- **time unit**: The amount of time a tick delay of 1 represents. Must be 1 10 or 100.
- **base**: The time base for each unit. Must be: s ms us ns ps or fs.
- **precision**: Represents how many decimal points of precision to use relative to the time units.

Example timescale with 1ns units and 10ps of precision:

```
timescale 1 ns / 10 ps
```

### 3.2 Events

The execution of a procedural statement can be triggered with a value change on a wire or register, or the occurrence of a named event.

Some examples:

```
@r begin // Controlled by any value change in r
A = B & C;
end

@ (posedge clock2) A = B & C;  // Controlled by positive edge of clock2
@ (negedge clock3) A = B & C;  // Controlled by negative edge of clock3

forever @ (negedge clock) // Controlled by negative egde
begin
A = B & C;
end
```

**Note**: In the forms using `posedge` and `negedge`, the must be followed by a 1-bit expression, typically a clock.

### 3.3 User-defined events

Verilog also provides features to name an event and then to trigger the occurrence of that event. We must first declare the event:

```
event event6;
```

To trigger the event, we use the `->` symbol:

```
->event6;  // To control a block of code for this event, we use the `@` symbol:
```
@([event6]) begin
<some procedural code>
end

We assume that the event occurs in one thread of control, i.e., concurrently, and the controlled code is in another thread. Several events may be or-ed inside the parentheses (after @ symbol).

3.4 Wait Statement

The wait statement allows a procedural statement or a block to be delayed until a condition becomes true.

```plaintext
wait (A == 3)
begin
A = B & C;
end
```

4 Blocking and non-blocking Assignments

In the previous sections we saw the assignments which use the “=” operator. Such assignments are referred to as blocking assignments. With such assignments, the next statement after the assignment statement does not begin executing until the assignment has finished. The blocking assignments have the form

```plaintext
left-hand side = [control] right-hand side;
```

For example, consider the following assignment with intra-assignment delay:

```plaintext
y = #10 x + z;
```

The execution of this statement requires the following steps:

1. Evaluation of the right-hand side and saving the result. In this case, the value of $x + z$ will be evaluated;
2. Wait for the specified delay - in this case 10 ticks;
3. Perform the assignment.

The execution of this statement includes waiting for 10 time ticks. That is, the next statement will execute only after 10 ticks. Thus, this statement blocks the next one until the waiting is over.

A blocking statement may wait for a specific event to occur. For example, consider the following statement:

```plaintext
y = @(posedge clk) y + z;
```
This statement first evaluates the right-hand side expression \((y + z)\) and then waits for rising edge of clk and performs the assignment.

*Non-blocking* assignments use the “<=” operator. Such assignments do not block the following statement even in case of intra-assignment wait. Non-blocking assignments have the following form:

\[
\text{left-hand side} \leq \text{[control]} \text{ right-hand side};
\]

For example consider the following assignment with intra-assignment delay:

\[
y \leq \#10 x+z;
\]

The execution of this statement requires the following steps:

1. Evaluate the right-hand side and save the result. In this case, the value of \(x + z\) will be evaluated;

2. Put the assignment on the waiting list to be executed at \(t + 10\) time units, where \(t\) is the current time.

3. Continue the next assignment at time \(t\).

## 5 System Tasks

Verilog provides a means for executing system level operations on the host machine during simulation via system tasks. All system tasks use the form `$\langle\text{keyword}\rangle$. A commonly used system task is `$\text{display}$', which prints variables, strings, and expressions to the terminal window when invoked. The syntax for `$\text{display}$' follows very closely to that of `printf` in C. Another useful system task is `$\text{monitor}$', which continuously monitors the signals specified in the parameter list and prints to the terminal whenever any of these signals change. `$\text{monitor}$' needs to only be invoked once and can be toggle on and off by other system tasks not mentioned here. Another useful system task is `$\text{time}$', which provides the current simulation time when invoked. Finally, for exiting simulation, Verilog provides the `$\text{finish}$' system task.
6 Questions

1. The goal of the following model is generate a clock waveform that has the clock high 4 time units and low 4 time units, with the first rising edge at time 20. Complete the missing lines.

   ```
   module q2;
   reg clock;
   initial begin
   clock = 0;
   ...
   ...
   ...
   end
   end
   endmodule
   ```

2. The following model produces a 4 bit output that counts the number of rising clock edges which have occurred. The output should be produced on the falling edge of the clock. Complete the missing fields.

   ```
   module Counter (count, clock);
   input clock;
   output count;
   reg [3:0] ...., count_reg;

   initial count_reg = 0;

   always @(posedge clock)
   ....;

   always @(....)
   .... = count_reg;

   endmodule
   ```
3. Consider the following code snippets:

(a)

```c
A = 1;
B = 2;
A = B;
B = A;
```

What will be the values in A and B after the above code is executed?

(b)

```c
A = 11;
B = 22;
A <= B;
B <= A;
```

What will be the values in A and B after the above code is executed?
4. What will the following code display to the terminal?

```verilog
module test1(i1, i2);
input i1, i2;
reg [31:0] a;
always @(i1) begin
  #1 a = 1;
  #5 a = a + 1;
  #5 a = a + 2;
end
initial begin
  $monitor ($time, "\t a = %d", a);
a = 2;
end
endmodule

module testtest1();
reg i1, i2;
test1 t1(i1, i2);
initial begin
  i1 = 1;
  #2 i1 = 0;
  #10 $finish;
end
endmodule
```
5. What are the values of the variable \( out \) in the following code

```
module test();
reg clock;
reg out;

initial begin
    clock = 1'b1;
    forever #5 clock = !clock;
end

initial begin
    out = 1'b0;
    #50 $finish;
end

always @(posedge clock) begin
    #5 out = !out;
end
endmodule
```

6. Assume an MxN register file where \( M \) is the number of registers and \( N \) is the number of bits in each register. The register file can read 2 registers and optionally write a single register in each clock cycle on a clock edge if write is enabled. This results in 2 data outputs and 6 inputs.

What are the different input and outputs of the register file? Include the width of each port. Express appropriate values in terms of \( M \) and \( N \).