Introduction to Verilog

The purpose of this lab is to introduce verilog and the Xilinx software environment to you. Before starting this lab you should complete the Xilinx Tutorial found here:

http://www.doe.carleton.ca/~wknisely/elec3500/xilinx_2010_tutorial.html

(Note: you do not need to complete the up/down counter test bench. That will be done in a later lab.)

Verilog HDL is a hardware description language (HDL). A HDL is a programming language used to describe a digital system. The syntax of Verilog is very similar to the C programming language, but Verilog does not compile or execute like C. In general, Verilog is used to define the behaviour of a circuit and the software then uses abstraction to synthesize a gate level implementation of the design.

1 Verilog Concepts

This section describes some typical verilog statements and demonstrates with examples how to use them. To get started:

1. Create a new folder in your directory (eg. W:\ELEC3500\Lab3).
2. Download the following files to the new directory (eg. W:\ELEC3500\Lab3):

demo1.v	demo5.v	demo8.v
demo2.v	demo6.v	FSM_tb.v
demo3.v	demo7.v	counterFSM.v
demo4.v		

3. Open Xilinx and start a new project. Use the directory above (eg. W:\ELEC3500\Lab3).
4. Add the above files as new sources.

1.1 Data Types for Synthesis

Registers and wires are used to deliver information around a module. In verilog they appear as variables of types:

reg: a variable that has storage capability. It can keep the value for 1 clock cycle or longer and drive a wire. This can synthesize as latches or flip flops.

wire: a variable with no storage. It is just a connection. Wires need a driver that sends/assigns information.

The data in a register or wire can have 4 possible values: 0, 1, x, z:

0: logical “0”
1: logical “1”
x: don’t care/undefined/unknown
z: high impedance (x input on a gate causes z output)

Demo 1

The pound symbol (#) means delay or wait. It is always followed by an integer representing time. Thus "#5" means wait 5 time units. Note the default time unit is picoseconds, but can be changed.

a) Simulate demo1 and observe at what time each of the assign statements is executed.
b) Create a table of each register's value (a1,a2,…a6) versus the execution time.

Execution Time	a1	a2	a3	a4	a5	a6
0ps						
Etc.						

c) Note the time the simulation finishes. What time did you expect the simulation to finish?
d) How are the initial blocks executed: sequentially or concurrently?

Demo 2

e) Simulate demo2 and note when the value of a1 changes.
f) Create a table, similar to that of part b, of a1's value versus the execution time for each change in a1's value.

1.2 Behavioural and Gate Level Simulations

Behavioural modeling represents a circuit at a high level of abstraction and is done through programming using the Verilog HDL language. It has no relation to the hardware details and implementation. The gate level modeling (low level of abstraction) the description of the module is implemented in terms of gates and interconnections between the gates, which is closer to the actual hardware.

Behavioural simulation of a module uses the procedural statements 'initial' and 'always'. Code within an initial block executes once starting at time 0, while code within an always block executes as a loop. Only the reg data type can be assigned a value within these blocks.

Demo 3

Code within an always statement block starts at time 0 and executes continuously. In demo3 the always block generates a running clock. Note that wire cannot appear in an always block.

a) Simulate demo3.
b) What is the measured period of the clk?
c) What happens if the $finish statement is removed (do not remove ';' after $finish)?

Demo 4

To change the value of a wire, the statement assign is used. Assign is the same as connecting two wires together; they will always have the same value.

d) Simulate demo4.

e) Create a table of a's value versus the time when a's value changes.
f) Does a's value always change on a clk edge? Why or why not?

1.3 Assignment Statements: Blocking '=' and Non-blocking '<='

The value of a register can be changed by using the blocking assignment operator '=' or the non-blocking operator '<='. Each operator has a different behaviour.

Demo 5

a) Simulate demo5.
b) Observe at what time each statement in the initial block is executed. Create a table of statement versus execution time.

Demo 6

c) Simulate demo6. Note that some of the blocking assignment statements from demo 5 are now non-blocking.
d) Observe at what time each statement in the initial block is executed. Create a table of statement versus execution time.
e) What difference in waveforms between blocking (demo5) and non-blocking (demo6) statements have you found?
f) Which statements, blocking or non-blocking, execute in sequence?

1.4 Sensitivity List

The 'always' statement is used to control the execution of certain statements by the occurrence of specified events. For example, always (@ posedge clk) is used to execute the code in the always block every time a positive edge of the clock (i.e. rising edge) happens. You can list several events in the bracket, connected with or. This list is called the sensitivity list.

Demo 7

a) Simulate demo7.
b) Which value of reg a1 is assigned to reg a3, the old one (before the assignment to b1) or updated (after the assignment to b1)?
c) Do final values of reg a1, a2, a3 depend on the order of assignment statements '<='?

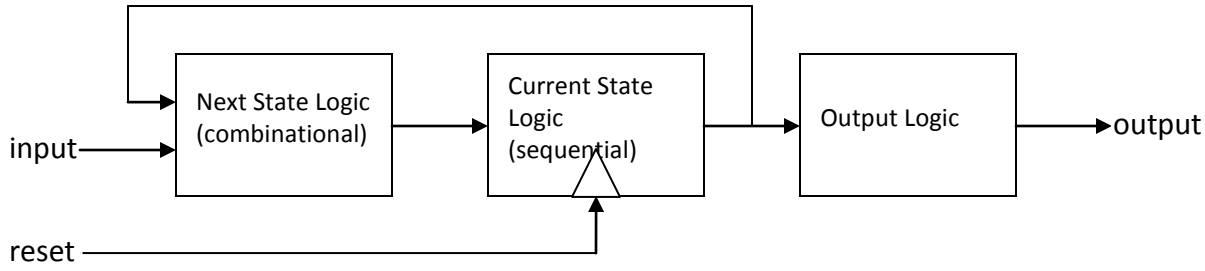
Demo 8

d) Simulate demo8. The values of the registers are meant to toggle at each positive clock edge.
e) Which assignment blocking or non-blocking can do a toggle operation?
f) Based on demonstrations 4 to 8 describe what blocking and non-blocking assignments are and how they are different.

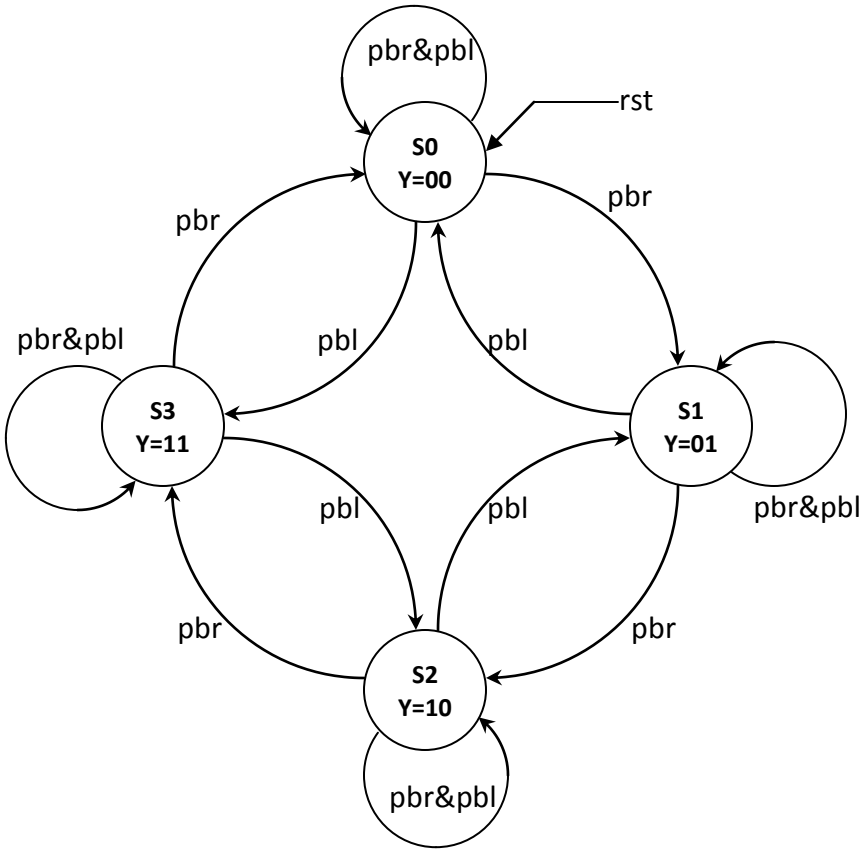
2 Part 2.

This section introduces the concept of the finite state machine (FSM) and implements a 2 bit counter as a demonstration. The implementation of the counter is found in counterFSM.v, while the test bench is in FSM_tb.v

A finite state machine is an object that has a limited number of states. A typical block diagram for a FSM in verilog is shown below.



To describe each of the blocks in a FSM always statements are used. The sequential block generates the current state the FSM is in. In the case in which the reset button is pressed, the current state is forced to a pre-specified state. Otherwise, on the next clock, it is equal to next state, which is calculated by the combinational block. Calculation of the next state depends only on transition rules/specifications of the FSM. Transition from one state to another can be graphically represented by a transition graph. The output block calculates the output signals given the current state of the FSM.



Note that some inputs for some transitions are implied! For example, rst is shown with no source state. This implies that rst can occur from any state. Also, the transitions only show the inputs that are on, which implies the other inputs should be off (for example pbr & ~pbl).

In this example we designed a 2-bit up/down counter using a FSM. These two bits are the most significant bits, i.e. 6 and 7 (our LED has 7 bits). If the 'pbr' button is pressed the counter should count up: 0, 32, 64, 96, 0, 32, 64, 96, … If the 'pbl' button is pressed the counter should count down: 96, 64, 32,0, 96, 64, 32, 0, … If both 'pbl' and 'pbr' buttons are pressed, counter should display current value and should not count. If the reset is pressed, the counter should display zero. For a 2-bit counter we need 4 states in total. The state transition graph is shown above.

Sequential block (current state).

```
always @(posedge clk or posedge rst)
        begin
                if(rst) state <= s0;
                else state <= next_state;
        end
```

Combinational block (next state)

```
always @(state or pbl or pbr)
        begin
                case(state)
                        s0: if (pbr&~pbl) next_state = s1;
                            else if (~pbr&pbl) next_state = s3;
                            else next_state = s0;
                            …
```

Output block

```
always @(state)
        begin
                case(state)
                        s0: count = 7'b0000000;
                        …
```

a) The test bench that is given tests two transitions: s0 to s1 and s1 to s0. Complete all other possible cases.