LAB K

Basic Verilog Programming

Perform the following groups of tasks:

LabK1.v

1. Create a directory to hold the files of this lab.

2. Launch your favourite editor and a command-prompt console; you will need to frequently switch back and forth between these two windows. Change the working directory of the console to that of this lab.

3. In the editor, type the following **Verilog** program. Note that Verilog is free form, case-sensitive, and has a syntax that is very similar to Java (and C). The major differences at this stage is that **module** is used instead of "class" and compound statements are surrounded by `begin` / `end` instead of braces:

```
module labK;
reg [31:0] x; // a 32-bit register

initial
begin
   $display($time, " ", x);
   x = 0;
   $display($time, " ", x);
   x = x + 2;
   $display($time, " ", x);
   $finish;
end

endmodule
```

4. This module declares the variable `x` to be of type `reg` (register) and size `[31:0]` (32 bits labelled left to right 31 to 0). It then starts an `initial` block. The block contains one (compound) statement and, as a matter of style, it is indented right by one tab position. The above code uses a tab size of 3. Save the program under the name: LabK1.v. Unlike Java, the filename does not have to match the module's name.

5. Switch to the console and issue the following commands:

```
iverilog labK1.v
vvp a.out
```

The first compiles the program into **`a.out`** and the second runs **`a.out`**. This is reminiscent of compiling and running a Java program using the *javac* and *java* commands. If you prefer to give your compiled program a different name, e.g. **`circuit`**, then you would use the **`-o`** switch:

```
iverilog -o circuit labK1.v
vvp circuit
```

6. Examine the generated output. Note, in particular, that:

   - The output indicates that all statements were executed at the same time, time 0. Statements in an initial block are executed *sequentially* (one after the other) but each takes 0 time to execute.

   - Unlike Java, it is legal to use a variable before initializing it. In that case, its value would be unknown and is shown as **`x`** in the output.

7. Add output formatting by replacing each of the three **`$display`** statements with:

   ```
   $display($time, " %b", x);
   ```

   Recompile and run the modified program. The outputs should now be in binary.

8. Replace **`%b`** with **`%h`** in the last output and then recompile and run. The outputs should now be in hexadecimal. In order to display in decimal, use no formatting at all or specify **`%d`**. Furthermore, you can specify the width of a decimal output by placing it between the **`%`** and the **`d`**, e.g.

   ```
   $display("%5d", $time, " %b", x);
   ```

   This displays the time right justified in a field of 5 columns.

9. Replace all three output statements with the following statement:

   ```
   $display("time = %5d, x = %b", $time, x);
   ```

   As you can see, you can combine all format specifiers and text literals into one string. Recompile and run.

10. Replace the initialization of **`x`** with:

    ```
    x = 32'hffff0000;
    ```

    In general, the syntax for integer literals is:

    ```
    [size in bits]['base]value
    ```

    where base is either **`d`** (decimal) or **`b`** (binary) or **`h`** (hexadecimal). Recompile and run.

11. Declare three new reg variables: **one** (of size 1 bit), **two** (2 bits), and **three** (3 bits). Assign their values such that **one** is the and of all the bits of **x**, **two** is the least-significant two bits in **x**, and **three** is the concatenation of **one** and **two**; i.e. add the following statements before $**finish**:

    ```
    one = &x;              // and reduction
    two = x[1:0];          // part-select
    three = {one, two};    // concatenate
    ```

    Verilog has the same arithmetic, relational, boolean, and bit-wise logical operators as Java. In addition, it has: **[]** for part selecting, **{}** for concatenating, *unary* **&** , **|** for reduction, and **===**, **!==** for testing equality in the presence of the **x** unknown value. Add an output statement to display the three new variables in binary and re-run.

LabK2.v

12. Create the program LabK2.v as follows:

    ```
    module labK;
    reg [31:0] x, y, z;

    initial
    begin
       x = 5;
       $display("%2d: x=%1d y=%1d z=%1d", $time, x, y, z);
       y = x + 1;
       $display("%2d: x=%1d y=%1d z=%1d", $time, x, y, z);
       z = y + 1;
       $display("%2d: x=%1d y=%1d z=%1d", $time, x, y, z);
       $finish;
    end

    endmodule
    ```

    Compile and run the program. Verify that all statements were executed sequentially.

13. Modify the program by adding a delay of 10 units of time before each variable is assigned. Prefix each of the three assignment statements with **#10**, e.g.

    ```
    #10 x = 5;
    ```

    When Verilog encounters such a delay, it suspends the execution of the thread associated with this block, executes any other pending threads, and then resumes executing 10 units later. Compile and run the program and interpret its output. Note that since statements in a block are executed sequentially, a delay in one causes the rest to be delayed.

14. Modify the program by adding the second **initial** block shown below. Note that different blocks in Verilog execute in *parallel*:

    ```
    initial
       #15 x = x + 1;
    ```

Compile and run the program. The output should change as shown below:

```
10: x=5 y=x z=x                        10: x=5 y=x z=x
20: x=5 y=6 z=x         -->            20: x=6 y=7 z=x
30: x=5 y=6 z=7                        30: x=6 y=7 z=8
```

15. Change the delay in the second block from #15 to #5. Compile, run, and explain why the output reverted back to the original one (as if the second block played no role).

16. Change the delay in the second block from #5 to #25. Predict the output then verify by re-running.

17. Change the delay in the second block from #25 to #35. Predict the output then verify by re-running.

18. Modify the second block so that it increments x every 7 units of time. Since the simulation lasts for 30 units, this means we must use 4 statements, i.e.

```
initial
begin
   #7 x = x + 1;
   #7 x = x + 1;
   #7 x = x + 1;
   #7 x = x + 1;
end
```

This can be done more conveniently by using a **`repeat`** or a **`for`** loop:

```
initial                    |       initial
begin                      |       begin
   repeat (4)              |       for (i = 0; i < 4; i = i + 1)
      #7 x = x + 1;        |           #7 x = x + 1;
end                        |       end
```

The variable **`i`** of the **`for`** loop is not a **`reg`** but a simple loop counter and is declared using: **`integer i`**. Use either loop to implement the change and you should obtain the output shown below. Explain why it came the way it did.

```
10: x=5 y=x z=x
20: x=6 y=7 z=x
30: x=8 y=7 z=8
```

19. Instead of using loops, we can use an **`always`** block, which can be thought of as a circular initial block, one that repeats forever. Comment out the second initial block (surround it by **`/*`** and **`*/`**) and add the following block:

```
always
   #7 x = x + 1;
```

Compile and run and verify that the output is the same as before.

20. When the simulation involves several blocks, it may become cumbersome to generate output using `$display`. Another system task is `$monitor`. It generates output whenever any of the variables it outputs changes. You can have only one such a statement in a program. Comment out all the `$display` statements (prefix each with `//`) and add the following initial block:

```
initial
   $monitor("%2d: x=%1d y=%1d z=%1d", $time, x, y, z);
```

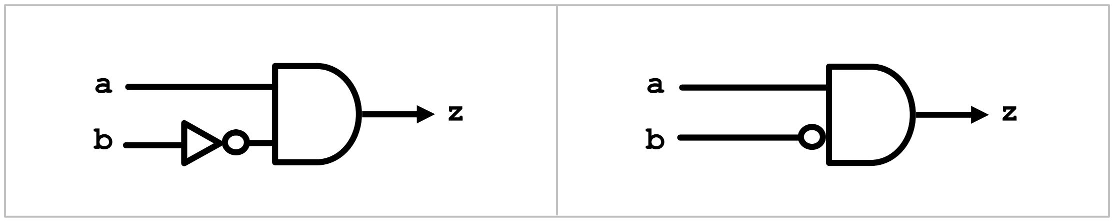
In order to give a chance for this statement to detect the change in `z`, delay the execution of `$finish` by any number of units, e.g. by 1 unit:

```
#1 $finish;
```

The idea is to relinquish control momentarily so that the `$monitor` thread can take control and generate its output. Recompile and run the program.

LabK3.v

21. We will now write a Verilog program that simulates and tests a circuit. The circuit shown in the left half of the figure below takes two inputs `a` and `b` each of which is one-bit, and generates an output `z` which is also one-bit. The circuit involves a `not` gate and an `and` gate as shown. The symbol for `not` (a triangle and a circle) can be reduced to just a circle touching the next gate as shown in the right half of the figure.



22. Create a new program named LabK3.v as follows:

```
module labK;
reg a, b;  // reg without size means 1-bit
wire tmp, z;

not my_not(tmp, b);
and my_and(z, a, tmp);

initial
begin
   a = 1; b = 1;
   $display("a=%b b=%b z=%b", a, b, z);
   $finish;
end

endmodule
```

Compile and run the program. The output should show `x` (unknown) for `z`.

23. Note that the program starts by simulating the circuit. The statement:

    ```
    not my_not(tmp, b);
    ```

    is similar to instantiating an object in Java: we specify the name of the class to be instantiated (`not`), give a name to the created instance (`my_not`), and then provide parameters (a.k.a. `ports`) to the constructor (the gate's output followed by its input). Note that the instance name is optional; i.e. you can omit it: `not (tmp, b)`.

24. Using the same name, `tmp`, to denote both the output of the `not` gate and an input of the `and` gate effectively connects these two gates. Note that variables that are set by the circuit, such as `z` and `tmp`, are declared using `wire` rather than `reg`. These variables are known as ***nets*** and are read-only; i.e. you cannot modify them in code.

25. It is sometimes preferable not to have instances sharing wire names and to use the `assign` statement to *explicitly* connect ports. Using this approach, the above code turns into the following equivalent code:

    ```
    module labK;
    reg a, b;  // reg without size means 1-bit
    wire notOutput, lowerInput, tmp, z;

    not my_not(notOutput, b);
    and my_and(z, a, lowerInput);
    assign lowerInput = notOutput;

    ... (as before)
    ```

26. Why did the program fail to capture the output of the circuit? Think about the timing and make the necessary correction. Compile, and re-run. The correct output should be 0 because: `1 and (not 1) = 1 and 0 = 0`.

LabK4.v

27. Save the program as LabK4.v

28. We tested the circuit for the `a=1`, `b=1` case only. In general, we shall use one of three methods to test a circuit:

    - Exhaustive Testing
      In this case we create loops (usually `for` loops) that generate every possible input to the circuit. We will use this approach to test the circuit of this section.
    - Sampled Testing
      In this case we look only at a subset of the possible test cases, ones chosen by the end user. We will implement this approach to test the next circuit in this Lab.
    - Random Testing
      In this case we create loops (usually `repeat` loops) that generate test cases randomly. We will implement this approach in the next Lab.

29. Since the size of the input space for this circuit is very small, only 4 possibilities, let us conduct an exhaustive test. We will use two nested `for` loops:

```
module labK;
reg a, b;
wire z;
integer i, j;

not my_not(tmp, b);
and my_and(z, a, tmp);

initial
begin
        for (i = 0; i < 2; i = i + 1)
        begin
            for (j = 0; j < 2; j = j + 1)
            begin
                a = i; b = j;
                #1 $display("a=%b b=%b z=%b", a, b, z);
            end
        end
        $finish;
end
endmodule
```

Compile the program and run it. Did the circuit pass all tests?

30. Argue that it would be incorrect to get rid of `i` and `j` and use `a` and `b` directly as loop counters. *Hint:* `reg` *variables have a fixed size and, hence, they wrap.*

LabK5.v

31. Save the program as LabK5.v.

32. Our testing module does not have an oracle; i.e. it prints the input and the output of each test case but cannot decide if the circuit has or has not passed. We can remedy this by computing the correct output ourselves using the language operators and then comparing it with the circuit's output. Add a declaration for the oracle's output:

```
reg a, b, expect;
```

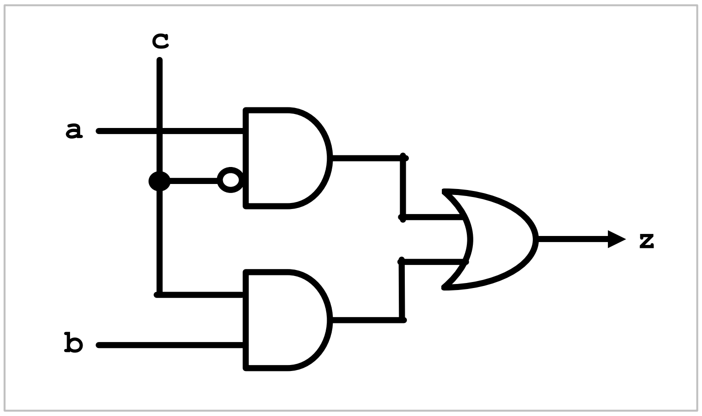
33. Change the body of the innermost `for` loop so it becomes:

```
a = i; b = j;
expect = i & ~b;
#1; // wait for z
if (expect === z)
   $display("PASS: a=%b b=%b z=%b", a, b, z);
else
   $display("FAIL: a=%b b=%b z=%b", a, b, z);
```

Compile the program and run it. Did the automated testing work?

LabK6.v

34. Create the program LabK6.v so that it simulates the following circuit:



Note that the circuit has three 1-bit inputs and one 1-bit output.

35. Did you use the **`assign`** statement (a.k.a. ***continuous assignment***) to connect the wires explicitly? If not, re-implement your code using (and declaring) distinct names for the gates' inputs and outputs and then connect wires using **`assign`**.

36. Start with the following test case:

```
a = 1; b = 0; c = 0;
```

Compile and run your program. The output should be 1 in this case.

37. Argue that if **`c`** is 0 then the output should be the same as **`a`** regardless of the value of **`b`**. You can do so by examining how the circuit is wired.

LabK7.v

38. Save the program as LabK7.v. Rather than hard-coding the test case, we seek to take it from the user via command-line arguments. Replace the initialization of the three inputs with the following:

```
flag = $value$plusargs("a=%b", a);
flag = $value$plusargs("b=%b", b);
flag = $value$plusargs("c=%b", c);
```

and declare **`flag`** as a **`reg`** variable.

39. Compile the program as usual but when you run it, supply the desired values of the three inputs as command-line arguments with a + switch:

```
vvp a.out +a=1 +b=0 +c=0
```

40. Note that the `$value$plusargs` system task has two parameters: the first is like a format specifier that enables *vvp* to parse the command line and extracts the desired argument. The second specifies the variable to be initialized. Hence, `a=%b` instructs *vvp* to look for `a=` followed by a single bit. This specifier can also be `%h` or `%d` to indicate that the entered value is in hex or decimal.

41. What happens if the user forgot to specify one or more of the needed arguments? Run your program without specifying `+a=1` and examine its output.

42. Is it possible to detect if one or more of the needed arguments is missing? Modify the program so that it will warn the user in that case.

    Hint: examine the value of `flag`.

LabK8.v

43. Save the program as LabK8.v and embed an oracle in it using the bitwise operators of Verilog. Have the program output PASS or FAIL accordingly. You can keep taking the input from the command line or switch to a loop-based exhaustive test.

44. Compile your program and run it. Verify that it works as expected.

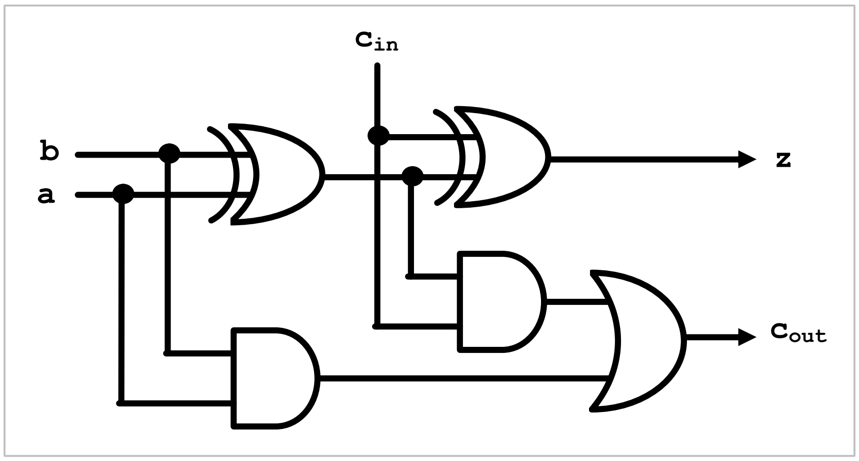
45. The function computed by this circuit can be expressed without using any bitwise operator. Comment out the bitwise-based oracle and add a new line that computes the expected output differently.

    Hint: Use an `if` statement or the `?` ternary operator.

46. Compile your program and run it. Verify that it works as expected. This circuit is known as a ***2-to-1 multiplexor***, or ***mux*** for short (more on this in the next Lab).

LabK9.v

47. Create the program LabK9.v so that it simulates the following circuit:



Note that the circuit has three 1-bit inputs and two 1-bit outputs. It is made up of two **`xor`** gates, two **`and`** gates, and an **`or`** gate.

48. It is up to you to use **`assign`** or shared names to simulate the circuit but make sure you connect its ports correctly. Either way, you may find it helpful to label the wires on the diagram and/or to give them meaningful names.

49. Since the circuit has three 1-bit inputs, the input space has a size of $2^3 = 8$. Use plus arguments to initialize these inputs and display the circuit outputs.

50. Note that if c_{in} is zero then **`z`** is nothing but the sum of **`a`** and **`b`** with a carry of c_{out}. That is why this circuit is known as a ***full adder*** (more on this in the next Lab).

51. Do an exhaustive test using three nested loops and verify the addition functionality of the circuit. This functionality can be expressed as:

```
expect[0] === z && expect[1] === cout
```

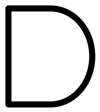
where **`expect`** is a two-bit vector (i.e. declared as **`reg[1:0]`**) computed using the statement: **`expect = a + b +`** c_{in}.

LAB K
Notes

- Do not confuse bitwise with logical operators. Verilog has three logical operators, `!`, `&&`, `||`, and they are used in conditional expressions to indicate negation, conjunction, and disjunction. The bitwise operators are `~` (not), `&` (and), `|` (or), `^` (xor), and they operate on operands bit by bit.
- In addition to the single-input `not`, Verilog has six built-in, primitive gates. Their logic names, truth tables, operators, and circuit symbols are shown below.
- All these gates have one output and two or more inputs. Upon instantiation the first wire you specify is the output and it is followed by the inputs. Supplying an instance name upon instantiation is optional.

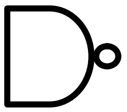
and	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

&



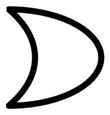
nand	0	1	x	z
0	1	1	1	1
1	1	0	x	x
x	1	x	x	x
z	1	x	x	x

~&



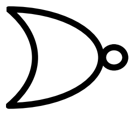
or	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

|



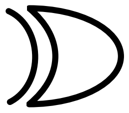
nor	0	1	x	z
0	1	0	x	x
1	0	0	0	0
x	x	0	x	x
z	x	0	x	x

~|



xor	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

^



xnor	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

~^