Introduction to Verilog

We are going to use Verilog to model our hardware systems and also to simulate and test them. Use of Verilog as simulation language is easier to understand than its use for hardware synthesis so we will start there.

Verilog has familiar C-like syntax (as does Java) and is case sensitive. For a simulation we need to (i) instantiate our hardware system, (ii) simulate the system clock, (iii) load a program to run and (iv) otherwise get ready to test the program by keeping track of the state of the simulation and inputs and outputs.

Types of components are defined by Verilog modules. **Modules** can correspond to hardware components when synthesized or to model or simulation components. The `module` keyword declares a new module which is given a name. The module declaration needs to be terminated by `;` (as do other statements).

```
// This is a comment /* also works */
module testbench;
```

We need to then define some of the important external signals that we will use to stimulate or test our accumulator. For instance the accumulator will need a system clock and a reset signal to put it into a known state.

```
reg clk, reset;
```

These signals are defined as type `reg` which is necessary as they will be set within an `always` or `initial` blocks (`reg` will also be used for registers). Among other signals we need to provide the instruction to execute (32 bit).

```
reg  [31:0] instruction;
```

The notation `[31:0]` indicates a 32-bit number for the register and is numbered from bit `31` down to `0` as we normally do for binary numbers.

Let's do some simple manipulations of the instruction. Verilog has special blocks that are useful in simulation `initial` and `always`. The initial block runs once when the simulation starts and we can use it to simulate a sequence of instructions and display them. We can display variables and text using `$display()` which provides console output of specified text or variable:

```
initial begin
    instruction = 10;
    $display(\Time: ", $time);
    $display(\Instruction: ", instruction);
    instruction = 20;
    $display(\Time: ", $time);
    $display(\Instruction: ", instruction);
    instruction = 30;
    $display(\Time: ", $time);
    $display(\Instruction: ", instruction);
$finish;
end
```

The group of statements in the always or initial block are grouped into a compound statement by enclosing them within a begin/end pair. The `$finish` statement at the end of the initial block indicates that the simulation is complete. The module is completed as follows: `endmodule`.

Note there is no semicolon after `endmodule` or `end`.

```
module test;
  reg clk, reset;
  reg [31:0] instruction;
  initial begin
  $display("Time: ", $time);
  $display("Instruction: ", instruction);
  instruction = 10;
  $display("Time: ", $time);
  $display("Instruction: ", instruction);
  instruction = 20;
  $display("Time: ", $time);
  $display("Instruction: ", instruction);
  instruction = 30;
  $display("Time: ", $time);
  $display("Instruction: ", instruction);
  $finish;
  end
endmodule
```

Save the above example as a file named **k1a01.v**.

In a terminal window navigate to the directory containing your Verilog code and test your program by:

```
iverilog k1a01.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 k1a01.v
vvp circuit
```

The result should look something like:

```
Time:              0
Instruction:       x
Time:              0
Instruction:       10
Time:              0
Instruction:       20
Time:              0
Instruction:       30
```

Correct, if there are any errors, and save the code to **k1a01.v**.

Adding timing and formatting

The output is rather simply formatted and the time is `0` for all instructions. In Verilog simulations everything happens conceptually at the same time (the statements in the initial block are executed sequentially so that the behaviour is consistent). When writing a simulation to test our system design, we want to make changes at specific times (for example execute a new instruction in every clock interval). Time intervals are specified as `#n` which indicates an n unit time delay.

Create a copy of your previous program as **k1b01.v** to introduce a time delay of `10`, `20` and `30` units before the first, second and third instructions, respectively. We can also format the `$display` a little more

flexibly. First, we can combine the text and the variables on the same line. Replace the first two `$display` lines with:

```
$display("Time: ", $time, ", Instruction: ", instruction);
```

We can also format more flexibly using C-style formatting specifiers (similar to `printf()` in C) like `%d`, `%h`, `%b` to indicate decimal, hex or binary. Other specifiers prescribing length can also be used, e.g. `%64b`. The specifiers are applied to the variables listed on the right in the order of their appearance.

For example, we can replace the remaining `$display` pairs with `$display` lines with:

```
$display("Time: %5d Instruction: %16h", $time, instruction);
```

The code should now look like this

```
// Testbench
module testbench;
      // Inputs to device under test (DUT) reg, outputs wire reg clk, reset;
      reg [31:0] instruction;
      initial begin
      $display("Time: ", $time, ", Instruction: ", instruction);
      #10 instruction = 10;
      $display("Time: %5d Instruction: %16h", $time, instruction);
      #20 instruction = 20;
      $display("Time: %5d Instruction: %16h", $time, instruction);
      #30 instruction = 30;
      $display("Time: %5d Instruction: %16h", $time, instruction);
      $finish;
  end
endmodule
```

Save the above example as a file named **k1b01.v**. In a terminal window navigate to the directory containing your Verilog code and test your program by:

```
iverilog k1b01.v
vvp a.out
```

The first line compiles your program and gives it a specific name (myfile) using the `-o` option, and the second simulates the resulting output. The result should look something like:

```
Time:     0, Instruction:          x
Time:    10 Instruction:          0000000a
Time:    30 Instruction:          00000014
Time:    60 Instruction:          0000001e
```

We see the formatting differences in the output. The instructions are padded with spaces because they are only `32`-bit and thus need `8` hexadecimal digits (each corresponding to `4` bits) not `16` as we specified.

Why does the first instruction printout return `x`? This is the Verilog symbol for *unknown* or *don't care* which is appropriate here as instruction has not been initialized at the first `$display` statement.

Why are the times for the last two instructions `30` and `60` rather than the `#20` and `#30` we specified? Recall that the times specified are delays (and thus cumulative) rather than absolute times.

Modify your code to (a) specify the instructions at absolute times `10`, `20` and `30` and (b) to print the `instruction` variable out in decimal, binary and hex at `10`, `20` and `30` respectively with appropriate text labels. Compile and test your program. When it works save it **k1c01.v**.

Monitoring variables and signals

Writing `$display` statements to trace a simulation is a little tedious if there are many changes. For example, if we changed the instruction variable each step in a complex program (here we will just use simple loop). We can monitor changes in variables whenever they occur using `$monitor`.

Create a copy of your previous program as **k1d01.v** and replace the `initial` block with: `$display` lines with:

```
integer i;
initial begin
$monitor("Time: %5d Instruction: %d (decimal)", $time, instruction);
#5
for (i = 0; i< 20; i= i + 1)
    #10 instruction = 1000+i;
$finish;
end
```

Test the program and verify you see a value displayed every time the variable changes. Save it as **k1d01.v**.

Circuit simulation and testing

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.

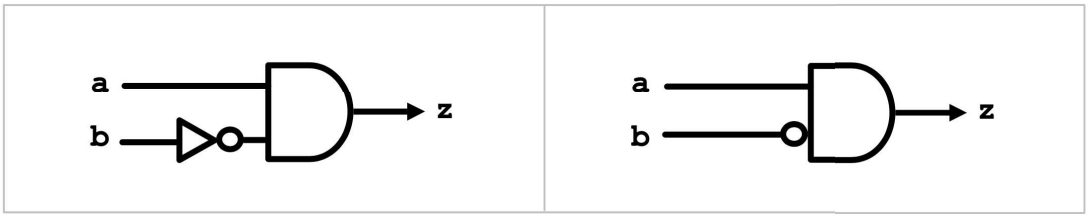


Figure 1: Circuits

Create a new program named **k2a01.v** as follows: `$display` lines with:

```
module labK;
reg a, b;  // reg without size means 1-bit
wire tmp, z;
// tmp is an output; b is an input
not my_not(tmp, b);
//z output; a, tmp are inputs
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`. 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)`.

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.

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)
```

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`.

User Input

In your previous code, 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);
```

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

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=1
```

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.

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.

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.*

Testing the Model

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.

Sampled Testing In this case we look only at a subset of the possible test cases, ones chosen by the end user.

Random Testing In this case we create loops (usually `repeat` loops) that generate test cases randomly.

Since the size of the input space for circuits in Figure 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
```

Create a new program named **k2a02.v**. Compile the program and run it. Did the circuit pass all tests?

The Oracle

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;
```

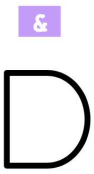
In your previous code, change the body of the inner most for `loop` so it becomes:

```
a = i; b = j;
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);
```

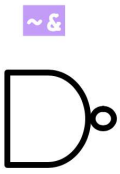
Notes

- 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

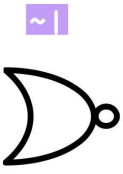


nand	0	1	x	z
0	1	1	1	1
1	1	0	x	x
x	1	x	x	x
z	1	x	x	x



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

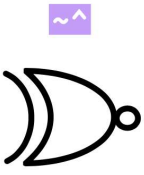


Figure 2: Logic gates