CS 207 Digital Logic - Spring 2020
Lab 2

James Yu

Feb. 26, 2020

Objective

1. Learn how to design in logic gate level.
2. Test Boolean algebra properties.

Lab Exercise Submission

1. You should name all source code as instructed. Mis-named files will not be recognized.
2. You should submit all source code files with an extension "`.v`".
3. You should submit all source code directly into the sakai system below. **Do not compress them into one folder.**

   `https://sakai.sustech.edu.cn/portal/site/ebf68254-68c5-4cfe-9d42-b26758f854ee`
4. Lab exercises should be submitted before the deadline, typically one week after the lab session. No late submission policy applies to lab exercises.

1 Gate Level Design

In Verilog a module can be defined using various levels of abstraction. There are four levels of abstraction in verilog. They are:

- Behavioral or algorithmic level: This is the highest level of abstraction. A module can be implemented in terms of the design algorithm. The designer no need to have any knowledge of hardware implementation.

- Data flow level: In this level the module is designed by specifying the data flow. Designer must how data flows between various registers of the design.
- Gate level: The module is implemented in terms of logic gates and interconnections between these gates. Designer should know the gate-level diagram of the design.
- Switch level: This is the lowest level of abstraction. The design is implemented using switches/-transistors. Designer requires the knowledge of switch-level implementation details.

Gate-level modeling is virtually the lowest-level of abstraction, because the switch-level abstraction is rarely used. In general, gate-level modeling is used for implementing lowest level modules in a design like, full-adder, multiplexers, etc. Verilog has gate primitives for all basic gates.

1.1 Gate Primitives

Gate primitives are predefined in Verilog, which are ready to use. They are instantiated like modules. There are two classes of gate primitives: *Multiple input gate primitive*s and *Single input gate primitive*s.

Multiple input gate primitives include `and`, `nand`, `or`, `nor`, `xor`, and `xnor`. These can have multiple inputs and a single output. They are instantiated as follows:

```verilog
// Two input AND gate.
and and_1 (out, in0, in1);

// Three input NAND gate.
nand nand_1 (out, in0, in1, in2);

// Two input OR gate.
or or_1 (out, in0, in1);

// Four input NOR gate.
nor nor_1 (out, in0, in1, in2, in3);

// Five input XOR gate.
xor xor_1 (out, in0, in1, in2, in3, in4);

// Two input XNOR gate.
xnor and_1 (out, in0, in1);
```

Note that instance name is not mandatory for gate primitive instantiation, which means that you can also create an `and` gate as follows:

```verilog
// Two input AND gate without a name.
and (out, in0, in1);
```

Single input gate primitives include not, buf, notif1, bufif1, notif0, and bufif0. These have a single input and one or more outputs. Gate primitives notif1, bufif1, notif0, and bufif0 have a control signal. The gates propagate if only control signal is asserted, else the output will be high impedance state (z).

```
// Inverting gate.
not not_1 (out, in);

// Two output buffer gate.
buf buf_1 (out0, out1, in);

// Single output Inverting gate with active-high control signal.
notif1 notif1_1 (out, in, ctrl);

// Double output buffer gate with active-high control signal.
bufif1 bufif1_1 (out0, out1, in, ctrl);

// Single output Inverting gate with active-low control signal.
notif0 notif0_1 (out, in, ctrl);

// Single output buffer gate with active-low control signal.
bufif0 bufif1_0 (out, in, ctrl);
```

1.2 Delays

In Verilog delays can be introduced with #num as in the examples below, where # is a special character to introduce delay, and num is the number of ticks simulator should delay current statement execution. This is especially useful when testing your Verilog code, or model delays of real logic gates.

```
module buf_gate ();
reg in;
wire out;

// This delays the out by 5 ticks from in.
buf #(5) (out, in);

initial begin
  $monitor ("Time = %g in = %b out=%b", $time, in, out);
  in = 0;
  // This delays the execution by 10 ticks.
  #10  in = 1;
  #10  in = 0;
  #10  $finish;
```

```
end
endmodule
```

1.3 Procedural Assignment Groups

In the above example, we come across three new keywords, `initial`, `begin`, and `end`.

- `initial` defines a series of Verilog behavior that is executed only once at time zero.
- `begin…end` encloses more than one statement as a group. The enclosed statements are evaluated sequentially, and any timing within the sequential group is relative to the previous statement. We will take about procedural assignment groups with more details later.

1.4 System Task and Function

In the above example, we also notice two keywords prepended with a dollar sign `$`. These are tasks and functions that are used to generate input and output during simulation. The following are some widely used ones:

- `$display`: displays a message once every time it is executed.
- `$monitor`: displays every time **one of its parameters** changes.

They can be used with the following syntax:

```
$display ("format_string", par_1, par_2, ... );
$monitor ("format_string", par_1, par_2, ... );
$displayb (as above but defaults to binary..);
$monitoro (as above but defaults to octal..);
```

The format string is like that in Java, and may contain format characters. Format characters include `%d` (decimal), `%h` (hexadecimal), `%b` (binary), `%c` (character), `%s` (string) and `%t` (time), `%m` (hierarchy level). `%5d`, `%5b` etc. would give exactly 5 spaces for the number instead of the space needed. Append `b`, `h`, `o` to the task name to change default format to binary, octal or hexadecimal.

There are also some other system tasks and functions you may find useful:

- `$time, $stime, $realtime`: these return the current simulation time as a 64-bit integer, a 32-bit integer, and a real number, respectively.
- `$reset`: resets the simulation back to time 0.
- `$finish`: exits the simulator back to the operating system. **You must include at least one `$finish` in your testbench.**

They are used without any parameters or parentheses.

2 Exercises

2.1 Test the Commutative Property

In the lecture we learnt the basics of Boolean algebra. Boolean operations have some properties, e.g., commutative, associative, distributive, DeMorgan, and absorption. In the first exercise, we try to test the commutative property with gate level models of Verilog.

The commutative property of AND and OR is represented by two equations: $x + y = y + x$, and $xy = yx$. While looks intuitive, the LHS and RHS lead to different wirings in a hardware. We first create a module that implements both of the LHS and RHS:

commutative.v

```verilog
module commutative(and_lhs, and_rhs, or_lhs, or_rhs, A, B);
input A, B;
output and_lhs, and_rhs, or_lhs, or_rhs;

and and_1 (and_lhs, A, B);
and and_2 (and_rhs, B, A);
or  or_1  (or_lhs,  A, B);
or  or_2  (or_rhs,  B, A);
endmodule
```

Then we write a testbench to test all possible combinations of A and B values:

commutative.v

```verilog
module commutative_tb;
reg A, B;
wire AL, AR, OL, OR;

commutative com_1(AL, AR, OL, OR, A, B);

initial begin
  $monitor("%3t: A is %b, B is %b, LHS of AND is %b, RHS of AND is %b, LHS of OR is %b
      , RHS of OR is %b.", $time, A, B, AL, AR, OL, OR);
  # 5  A = 0; B = 0;
  # 5  A = 1; B = 0;
  # 5  A = 0; B = 1;
  # 5  A = 1; B = 1;
  # 10 $finish;
end
endmodule
```

Here we use a new data type called reg. It a language construct denoting variables that are on the left hand side of inside an initial, always, or forever block (we will visit the latter two later. You declare reg, like wire, at the top level of a module, but you use them within initial, always, or forever blocks. You cannot assign reg values at the top level of a module, and you cannot assign wire while inside an initial, always, or forever block.

Finally, we use Icarus Verilog for simulation. We execute the following commands one by one:

```
> iverilog -o commutative.o commutative.v
> vvp commutative.o
  0: A is x, B is x, LHS of AND is x, RHS of AND is x, LHS of OR is x, RHS of OR is x.
  5: A is 0, B is 0, LHS of AND is 0, RHS of AND is 0, LHS of OR is 0, RHS of OR is 0.
 10: A is 1, B is 0, LHS of AND is 0, RHS of AND is 0, LHS of OR is 1, RHS of OR is 1.
 15: A is 0, B is 1, LHS of AND is 0, RHS of AND is 0, LHS of OR is 1, RHS of OR is 1.
 20: A is 1, B is 1, LHS of AND is 1, RHS of AND is 1, LHS of OR is 1, RHS of OR is 1.
```

Does it meet your expectation?

2.2 Test the DeMorgan and Absorption Properties

The DeMorgan property of AND and OR is represented by $(x+y)' = x'y'$ and $(xy)' = x'+y'$. The absorption property of AND and OR is represented by $x+xy = x$ and $x(x+y) = x$. Write two modules with file names demorgan.v and absorption.v to test the equivalency of LHS and RHS. The module head should be defined as follows:

demorgan.v

```
module demorgan(and_lhs,and_rhs,or_lhs,or_rhs,x,y);
// Module Code
endmodule
```

absorption.v

```
module absorption(and_lhs,and_rhs,or_lhs,or_rhs,x,y);
// Module Code
endmodule
```

Write a testbench for each of the module that outputs all possible combinations of x and y.

Assignment
Save the source code in **demorgan.v** (for DeMorgan property) and **absorption.v** (for absorption property), respectively. Upload these two files to Sakai under **Assignments $\rightarrow$ Lab Exercise 2**.

2.3 Sum-of-Products and Product-of-Sums

Use gate-level model Verilog to verify the sum-of-products and product-of-sums transformation. Write a module to implement $(b+d)(a'+b'+c)$ in a file pos.v defined as follows:

pos.v

```
module pos(f,a,b,c,d);
// Module Code
endmodule
```

where `f` outputs the product-of-sum result of the Boolean function. Then, derive the sum-of-products form of the above function and write a module to implement the SOP in a file `sop.v` defined as follows:

sop.v

```
module sop(f,a,b,c,d);
// Module Code
endmodule
```

Write a testbench for each of the module that outputs all possible combinations of a, b, and c.

Assignment
Save the source code in **`pos.v`** (for product-of-sums) and **`sop.v`** (for sum-of-products), respectively. Upload these two files to Sakai under **Assignments** $\rightarrow$ **Lab Exercise 2**.

Assignment
When you finished Lab Exercise 2, there should be four `.v` files in the Sakai system: **`demorgan.v`**, **`absorption.v`**, **`pos.v`**, and **`sop.v`**.