**LSU EE 4755** **Homework 1** **Due: 17 September 2025**

Solution *Formatted 15:44, 18 October 2025*

*For instructions visit* `https://www.ece.lsu.edu/koppel/v/proc.html`. *For the complete Verilog for this assignment without visiting the lab follow* `https://www.ece.lsu.edu/koppel/v/2025/hw01.v.html`.

### Collaboration Rules

Each student is expected to complete his or her own assignment. It is okay to work with other students and to ask questions in order to get ideas on how to solve the problems or how to overcome some obstacle (be it a question of Verilog syntax, interpreting error messages, how a part of the problem might be solved, etc.) It is also acceptable to seek out digital design resources for help on Verilog, digital design, etc. It is okay to make use of AI LLM tools such as ChatGPT and Copilot to generate sample Verilog code. (Do not assume LLM output is correct. Treat LLM output the same way one might treat legal advice given by a lawyer character in a movie: it may sound impressive, but it can range from sage advice to utter nonsense.)

After availing oneself to these resources **each student is expected to be able to complete the assignment alone**. Test questions will be based on homework questions and **the assumed time needed to complete the question will be for a student who had solved the homework assignment on which it was based**.

**Problem 0:** Following instructions at `https://www.ece.lsu.edu/koppel/v/proc.html`, set up your class account, copy the assignment, and run the Verilog simulator on the unmodified homework file, `hw01.v`. Do this early enough so that minor problems (*e.g.*, password doesn't work) are minor problems.

### Homework Introduction

This assignment is a straightforward Verilog coding task. Those familiar with Verilog should be able to solve it in a few minutes. Most of the time will be spent on tasks such as finding the lab, logging in, and familiarizing yourself with the software.

### Testbench

To compile your code and run the testbench press F9 in an Emacs buffer in a properly set up account. (Of course the testbench won't run until any compilation errors are fixed.) The testbench will test module for Problem 1 in this assignment. The beginning of the testbench output, which may quickly scroll by, will look something like this:

```
Compilation started at Tue Sep  9 17:17:20

xrun -sv -batch -exit hw01.v
TOOL: xrun(64) 25.03-s001: Started on Sep 09, 2025 at 17:17:20 CDT
xrun(64): 25.03-s001: (c) Copyright 1995-2025 Cadence Design Systems, Inc.
Recompiling... reason: file './hw01.v' is newer than expected.
expected: Tue Sep  9 16:09:19 2025
actual:   Tue Sep  9 16:10:01 2025
file: hw01.v
```

At the end of the testbench output is a tally of the number of errors in each module. For a correctly solved assignment the output will be:

```
After 128 tests, 0 errors.
xmsim: *W,RNQUIE: Simulation is complete.
xcelium> exit
TOOL: xrun(64) 25.03-s001: Exiting on Sep 09, 2025 at 17:17:22 CDT  (total: 00:00:02)
```

Further up in the output the testbench shows the details for any modules that produced incorrect output (which is not the case above). The testbench will only show details of the first few errors in each module.

An unmodified assignment file will show errors. Here is an except from of the output:

```
For a=0, b=0, c=0, d=0, e=0, f=0, g=1 error x=0 != 1 (correct)
For a=0, b=0, c=0, d=0, e=0, f=1, g=1 error x=0 != 1 (correct)
For a=0, b=0, c=0, d=0, e=1, f=0, g=1 error x=0 != 1 (correct)
For a=0, b=0, c=0, d=0, e=1, f=1, g=0 error x=0 != 1 (correct)
For a=0, b=0, c=0, d=0, e=1, f=1, g=1 error x=0 != 1 (correct)
For a=0, b=0, c=0, d=1, e=0, f=0, g=1 error x=0 != 1 (correct)
For a=0, b=0, c=0, d=1, e=0, f=1, g=1 error x=0 != 1 (correct)
For a=0, b=0, c=0, d=1, e=1, f=0, g=1 error x=0 != 1 (correct)
For a=0, b=0, c=0, d=1, e=1, f=1, g=0 error x=0 != 1 (correct)
After 128 tests, 64 errors.
```

### Common Problems

Here are some common errors messages, which will be encountered when the code is compiled (for example, by pressing F9).

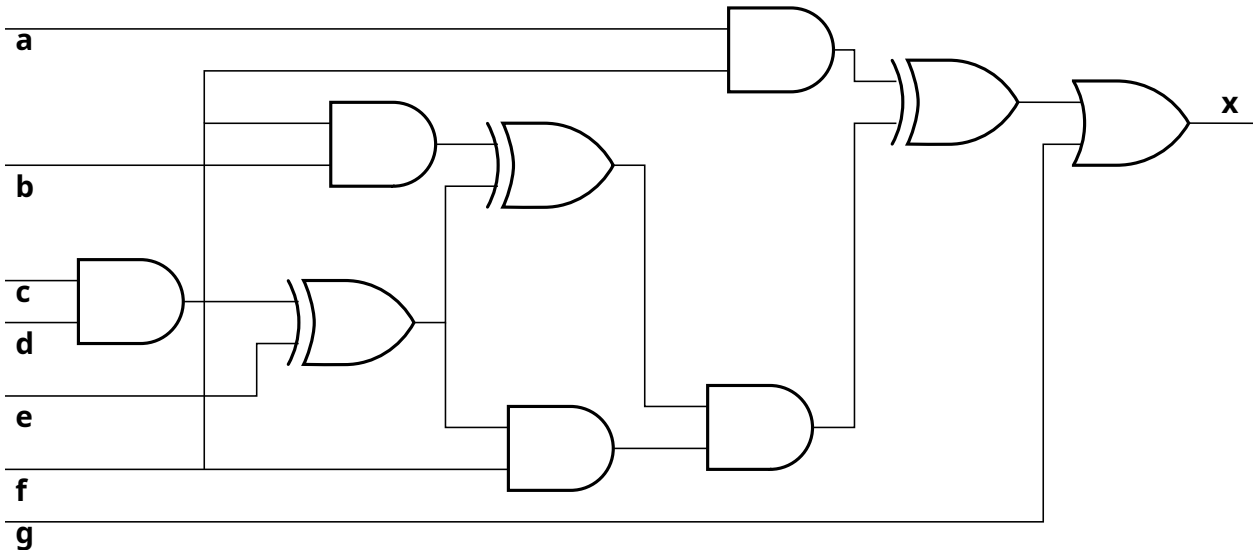
```
file: hw01.v
   mult mym( p, a[0], b[0] );
              !
xmvlog: *E,NODFNT (hw01.v,105!13): Implicit net declaration (p) is NOT allowed, since
‘default_nettype is declared as NONE [19.2(IEEE 2001)].
```

The problem above is that `p` was never declared. “Implicit net declaration” refers to a Verilog feature in which an object is assumed to be a `wire` if it had not been declared. That feature has been turned off since it can hide typos. The solution is to declare something like `uwire [um-1:0] p;`.

### Helpful Examples

A good past assignment to look at is 2024 Homework 1. Like this assignment, a module, say `dot4`, is completed by instantiating other modules. But in this assignment the (minor) challenge is to figure out what those other modules are.

**Problem 1:** Appearing below is a logic diagram of what we'll call CC1 (combinational circuit 1). Notice that pieces of logic in the diagram seem to repeat, such as around the exclusive or gate. Write a Verilog description of CC1 using two Verilog modules, `mCC1`, which should compute CC1 output `x`, and `mP1` (module part 1) which should implement those pieces of logic that seem to repeat. Starter code for module `mCC1` is in `hw01.v`, including the module ports. You'll have to *be able to (meaning you, not just a classmate or some large language model)* write module `mP1` from scratch.




☑ Complete `mCC1` so it implements CC1.

☑ The testbench should show zero errors.

☑ Module `mCC1` should instantiate module `mP1` several times.

☑ Module `mP1` should have at least two gates ☑ and one of those should connect to the other.

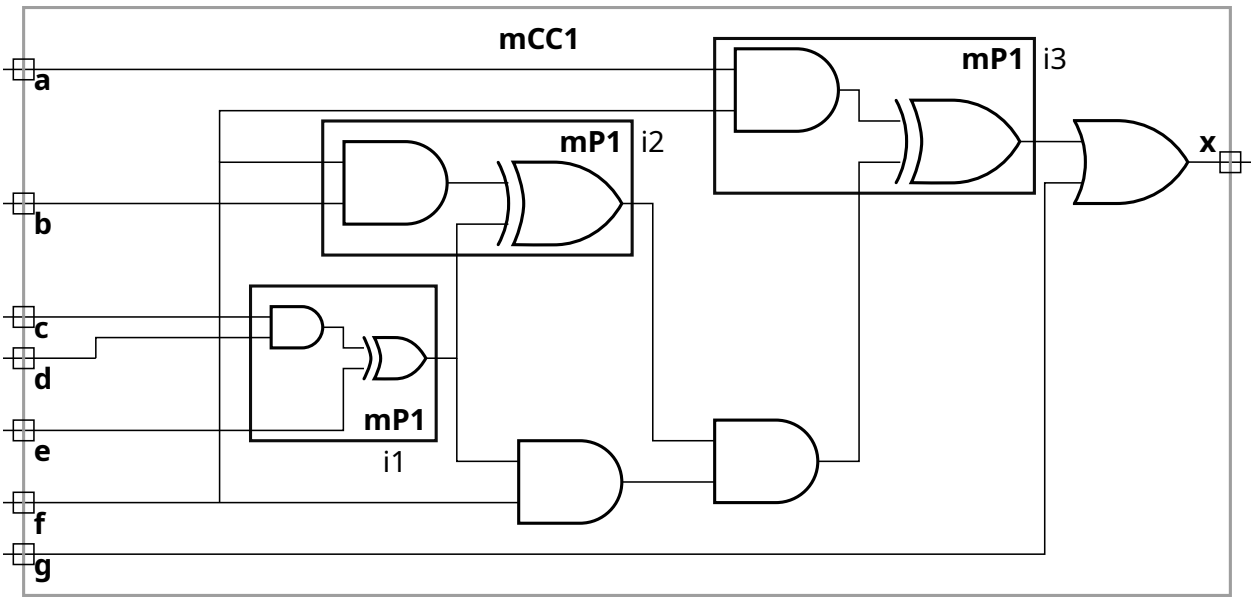
☑ Both modules can also contain primitive instantiations. Yes, this is one of those rare times you are told to use primitives.

☑ The modules cannot contain explicit structural code (that is, no `assign` statements) or behavioral code (which we haven't covered yet anyway), and so primitives are the only way to solve it.

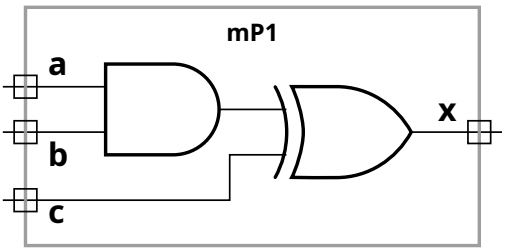
☑ Avoid logically correct but overly complex solutions. Also, don't try to optimize.

*Solution appears on the next page.*

The original logic drawn as a module, and showing three instantiations of `mP1` appears below.




The Verilog for `mP1` and `mCC1` appear below.




```
module mP1( output uwire x, input uwire a, b, c );
   uwire ab;
   and a1( ab, a, b );
   xor x1( x, ab, c );
endmodule

module mCC1( output uwire x,
             input uwire a, b, c, d, e, f, g );

   uwire y1, y2, y3, y5, y6;

   mP1 i1( y1, c, d, e);
   and a2( y2, y1, f );
   mP1 i2( y3, b, f, y1 );
   and a1( y5, y3, y2);
   mP1 i3( y6, a, f, y5 );
   or o1( x, y6, g );

endmodule
```

*There's another problem on the next page.*

**Problem 2:** Run synthesis on your solution to Problem 1. Synthesis can be run using the command `genus -files syn.tcl` in the assignment directory. After a slow and wordy start (it admits to taking 17 seconds before even getting around to actually looking at your code), if all goes well it will write:

```
Module Name           Area   Delay   Delay     Synth
                             Actual  Target     Time
mCC1                  1854    0.71   100.0 ns     5 s
mCC1_1                4604    0.38     0.1 ns     5 s
```

And then finish gracelessly with:

```
Normal exit.

Segmentation Fault accessing address 0.

Fatal internal error, code 11 (Segmentation fault)
Encountered a problem while exiting.
```

If the last line reads `@genus:root: 15>` then type `exit` and enter to exit.

The table above was actually produced by script `syn.tcl`, which is in the assignment directory alongside `hw01.v`. In this case the script synthesized `mCC1` twice, once with a delay target of 100 ns, and again with a delay target of 0.1 ns. Each row of the table corresponds to a synthesis. The column `Delay Actual` shows the actual delay achieved. Notice that for the 100 ns target the result was much faster than the target, but for the 0.1 ns target the actual module was slower. The `Area` column indicates the chip are of the two syntheses. Finally `Synth Time` shows how long it took to synthesize, it does not indicate anything about the synthesized module, just how much of our time synthesis took.

The results of the syntheses can be found in the `fv` directory, with one subdirectory for each synthesis. The directory names match the text under `Module Name`. A Verilog version of the synthesized module can be found in file `fv_map.v.gz`. (If you open the file in Emacs it will automatically decompress the file.) The file contains the `mCC1` module, but with instantiations of the Oklahoma State University ASIC modules. Descriptions of the modules can be found at
`file:///apps/linux/cadence/standard_cell/osu_soc_v2.7/cadence/lib/ami035/html/indexframe.html`

Draw a circuit diagram of the design using fewer modules. That is, look at one of the `fv_map.v.gz` files, and the Web page describing the Oklahoma modules, and draw a circuit diagram of the simpler module. Your circuit diagram should consist of AND, OR, XOR, and NOT gates, not boxes labeled with the Oklahoma AMI035 module names.

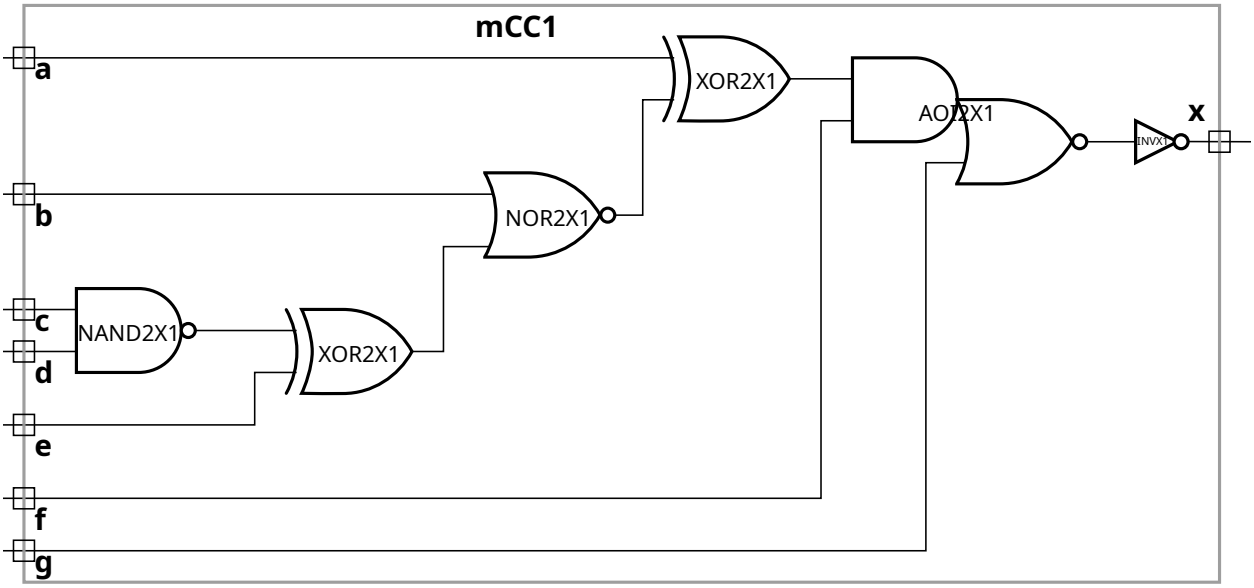
☑ Draw a circuit diagram of the simpler module. ☑ The diagram should use AND, NAND, OR, NOR, XOR, XNOR, and NOT gates.

☑ Did the synthesis program ⊗ *actually optimize your design*, or did it ◯ *merely replace primitives with closely matching Oklahoma AMI035 cells*? ☑ Explain.

Solution on next page.

The synthesis output followed by a diagram of this logic appears below.

```
// Generated by Cadence Genus(TM) Synthesis Solution 25.10-p002_1
// Generated on: Oct 17 2025 16:48:32 CDT (Oct 17 2025 21:48:32 UTC)
module mCC1(x, a, b, c, d, e, f, g);
  input a, b, c, d, e, f, g;
  output x;
  wire a, b, c, d, e, f, g;
  wire x;
  wire n_0, n_1, n_2, n_3, n_4;
  INVX1 g199(.A (n_4), .Y (x));
  AOI21X1 g200(.A (f), .B (n_3), .C (g), .Y (n_4));
  XOR2X1 g201(.A (n_2), .B (a), .Y (n_3));
  NOR2X1 g202(.A (b), .B (n_1), .Y (n_2));
  XOR2X1 g203(.A (n_0), .B (e), .Y (n_1));
  NAND2X1 g204(.A (d), .B (c), .Y (n_0));
endmodule
```




Notice that input `f` just connects to one gate, in contrast to the original diagram, where `f` connected to three gates. So it looks like the synthesis program actually optimized the design.

*Grading note:* In many solutions students commented on how what looked like unnecessary inversions, such as the AOI followed by a NOT gate near the output. The reason for that is that in many logic families it is less costly and faster to make a gate that inverts, such as NAND and a NOR. Though in the logic above it might seem wasteful to put in that final NOT gate, the savings by using inverting gates (NAND and NOR) are greater than the cost of the NOT.

*Note: In the original assignment NOR, NAND, and XNOR were not in the list of gates that could be used. That was an oversite. Full credit would be given both for a NAND and for an AND followed by a NOT. Points were deducted, on the other hand if, say, the final NOT was omitted.*