

Notes on Computer Project #3

Comments about the assignment and responses to frequently asked questions will be added to this file as necessary.

***** comments added on 09/10/18 *****

1- Please note the following statement from the assignment handout:

The deliverable for this assignment is the following file:

proj03.netlist.c -- the source code file containing your solution

Be sure to use the specified file name, and to submit your file for grading via the handin program.

It is possible to submit your solution files multiple times: the last version of a file which you submit will be graded.

2- Please review the information about the "sim" software on the course website:

click on the "Course Information" tab, then click on "intro.sim.pdf"

As noted in that document, the "sim" software is only available on the Pi array and there are certain steps that you must take to run "sim" remotely.

3- If you have not already done so, you should complete the first two self-study modules ("lab01" and "lab02") before developing your solution to this project. The files for those exercises are available as:

```
/user/cse320/Labs/lab01*  
/user/cse320/Labs/lab02*
```

Those exercises will help you become familiar with the "sim" software.

4- Please note that you are developing the combinational logic to generate the correct signals that could be used as inputs to an LED component, not the LED component itself.

That is, you are developing the eight functions that produce a "present" signal and the seven signals associated with the seven segments of an LED.

If the "sim" package contained an LED component (which it does not), you would connect the eight output signals from your circuit to the eight inputs of that LED component.

Since "sim" does not contain an LED component, you will examine ~~the~~ output signals of your circuit using eight probes. To generate the four bits of input required by your circuit, you will use a counter and pulsers. From the project handout:

Function "simnet" will serve as a test fixture which allows the user to produce hexadecimal digits and observe the resulting outputs. The input will be generated using a four-bit counter (controlled by pulsers), and the results will be displayed using eight probes.

The pulsers used in conjunction with the four-bit counter will be placed vertically on the left edge of the "sim" window.

The eight probes will be placed near the right edge of the "sim" window. The "Present" probe will be placed at the top of the "sim" window, while the probes for the other seven functions ("a", "b", and so on) will be placed below the "Present" probe in the same pattern used by a seven-segment display.

All pulsers and probes will be appropriately labeled in the "sim" window.

Thus, the output produced by "sim" will be a pattern of "off" and "on" values on the eight probes, where seven of the probes are arranged in a pattern similar to a seven-segment display component:

```

+-----+
+-----+
+-+      +-+
| |      | |
+-+      +-+
+-----+
+-----+
+-+      +-+
| |      | |
+-+      +-+
+-----+
+-----+

```

4) A counter is a sequential circuit which uses flip-flops to retain the current state (value in the count sequence); it moves to the next state (next value in the count sequence) when the counter is strobed. It always outputs the current state.

A pulser is used to generate a temporary value of "One" on a specified signal line (signal value: Zero ==> One ==> Zero).

The file "/user/cse320/Labs/lab03.counter.c" contains a C++ module which is a test bed for experimenting with the "Counter" and "Pulser" components.

5) Please note the following statement from the assignment handout:

Your implementation will be formalized by creating a C++ source code file which represents the minimized version of each function in your circuit and serves as input to the "sim" package.

That is, your circuit must contain eight functions which match the eight minimized functions in your design document, and those functions must be constructed out of gates from the set {NOT, AND, OR}.

You should not use XOR gates, decoders, or other components which do not correspond to the operations in the minimized functions in your design document.

6) There are several features of "sim" illustrated in the self-study modules which might be useful for this project.

For example, the constant signals "Zero" (false) and "One" (true) are defined in the "Sim.h" interface file.

Also, "composition" (with parentheses and commas) can be used to group signals into larger units.

Consider the following statement:

```
Mux (SD("2b-4b"), (C1, C0), (One, D2, D1, D0), MuxOut);
```

The notation "(C1, C0)" forms a two-bit signal and the notation "(One, D2, D1, D0)" forms a four-bit signal (where the most significant bit is always asserted).

Please note that composition can be also used for output signals. Assuming that "w", "x", "y" and "z" are one-bit signals, then "(w, x, y, z)" could serve as the output of a four-bit counter.

~~7~~ Please note that you are not allowed to change the interface between function "simnet" and function "circuits":

```
void circuits( SD, Signal, Signal, Signal, Signal, Signal,  
              Signal, Signal, Signal, Signal, Signal, Signal, Signal );
```

That is, function "circuits" must receive four 1-bit signals, and it must send back eight 1-bit signals. If one or more of the eight output signals is a constant value (such as the constant signal "One"), simply send back that value.

8) One way to "rename" a signal is to OR the signal with itself. For example, if the "Present" signal returned from function "circuits" was always supposed to be true, then you could use:

```
Or ( SD("1a"), (One, One), Present );
```

The SD coordinates ("1a") depend on the rest of your circuit layout.

9) I recommend using an incremental approach to this project. Perhaps the first step is to make sure that the interface between function "simnet" and function "circuits" is working correctly. After that, you can incrementally add to your solution.

Some suggestions:

~~a~~. In function "simnet", you might be wise to start with four switches to generate the four 1-bit input signals (w, x, y, z) and eight probes to display the eight 1-bit output signals. Don't worry about the layout of the probes at this point.

In function "circuits", use "signal renaming" (discussed above) to generate arbitrary values for the eight output signals.

Run your simulation and see if the interface is working correctly.

b. Revise function "circuits" to actually handle the eight required functions (perhaps in several steps). Run your simulation after each step to check the new functionality.

c. Revise function "simnet" so that the eight probes are in the right places:

The eight probes will be placed near the right edge of the "sim" window. The "Present" probe will be placed at the top of the "sim" window, while the probes for the other seven functions ("a", "b", and so on) will be placed below the "Present" probe in the "Figure 8" pattern used by a seven-segment display.

Run your simulation to check the new layout.

d. Revise function "simnet" to use a 4-bit counter instead of four switches:

Function "simnet" will serve as a test fixture which allows the user to produce hexadecimal digits and observe the resulting outputs. The input will be generated using a four-bit counter (controlled by pulsers), and the results will be displayed using eight probes.

The pulsers used in conjunction with the four-bit counter will be placed vertically on the left edge of the "sim" window.

Run your simulation to check the new functionality.

e. As a final step, review the project handout to make sure you have met all of the specifications!

10) Be sure to avoid invalid "positions" for components, which results in an unreadable display. Positions (row and column indicators) are restricted to two characters. Thus, "9c" is valid, but "10c" is not.

Here's the official statement from the "sim" manual (available as a PDF under "Related Links"):

The Schematic Descriptor (SD) entry positions the component on a two-dimensional grid in the simulation-time display-window. In their simplest form, descriptors are given as two-character, quoted ASCII strings. This two character sequence specifies the row and column coordinates of a virtual grid overlaying the display window. For example, "1a", "1b", "2c", specify row 1 column a, row 1 column b, and row 2 column c. The grid is virtual in the sense that the maximum extent of placements occurring throughout all the SDs encountered in a system description are mapped to the full extent of the window of the display. Row and column extents in a window will include all the ASCII characters between the extremes of those used. Row and column labeling are independent of each other and the same labels could be used for each.

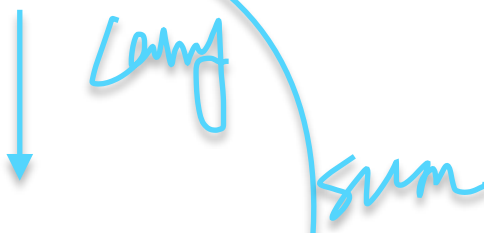
If you find that you need more rows than columns, you might consider using letters for rows and digits for columns (for example, "a1" and "k7"). As noted in the paragraph above, you can even use notation such as "aa" and "dk", where letters are used to identify both rows and columns.


11) The "Module" component allows a given C++ function to be treated the same way as a standard component, which is quite convenient for more complex circuits. Here's the relevant text from the "sim" manual:

3.1. Hierarchy

For simulation models including more than a few gates, it is desirable to be able to form hierarchical modules that may be used directly as model components. As an example consider the following HalfAdder module that is defined in terms of primitive Xor and And gates.

```
void HalfAdder( const SD & sd, const Signals & in, const Signals & out )
{
    Module( sd, in, out ); // Module display at current schematic level
    Xor( SD( sd, "1a" ), in, out[1] ); // Nested <Schematic Descriptor>
    And( SD( sd, "2a" ), in, out[0] ); // Nested <Schematic Descriptor>
}
```





The above definition allows the subsequent use of the module in a manner similar to the use of primitive components, such as:

```
HalfAdder( "1b", ( x, y ), ( carry, sum ) ); // Module usage
```

Please note that it is necessary to list an additional parameter for each "SD" component inside the module: the schematic descriptor from the next level up in the hierarchy. From the "Xor" component above:

```
... SD( sd, "1a" ) ...
```

where "sd" is the formal argument name in the definition of "HalfAdder".

One of the files from the third self-study module ("lab03.mux_bank.c") illustrates this technique.

To summarize, each "SD" component inside function "circuits" will require two arguments: "sd" and a two-character row and column indicator.

12) Since function "circuits" uses a "Module" component, it will appear as a "black box" in the initial circuit diagram. To move down one level in the module hierarchy, use the "Page Down" key and then redraw the circuit diagram using the "F12" key. To move back up in the module hierarchy, use the "Page Up" key (and then redraw the circuit diagram with the "F12" key).

13) If the circuit diagram becomes difficult to read due to the lines connecting the various gates and components, you can remove them by using the "F2" key to toggle the drawing of lines and the "F12" key to redraw the screen.

14) Some students have trouble with the idea of a hierarchy of modules. Compare and contrast these two different implementations of the first function from Lab Exercise #1:

```
/user/cse320/Labs/lab01.circuit.c
/user/cse320/Labs/lab01.modules.c
```

The second file uses a two-level hierarchy, similar to the the approach you will use in Project #3.

--M. McCullen