Lab #6: Digital Logic

Goal: Design a combinational logic circuit that implements a decoder for the (15,8,4) error correcting code we used in Lab 5. Verify the design using the JSim logic simulator.

Instructions:

1. Read Introduction to JSim (in the General section of the Handouts webpage).

2. Complete the pre-lab (see first section below). There are questions to be answered, please write your responses on a separate sheet of paper and turn them in at the beginning of lab on Wednesday.

3. Complete the activities for Wednesday’s lab (see second section below).

4. Prepare the requested material and think about the questions posed on the Check-off Sheet, then find a staff member to complete your post-lab interview.

Pre-lab (due in lab, Wed., April 2, 2008)

The “Entering a circuit” section in Introduction to JSim will be useful background reading for the tasks below. A list of the pre-built gates in the 6.02 gates library appears at the end of this section – use these as the building blocks for your designs.

Question 1: Design a Q1 module that computes \( Z = A \cdot B + C \cdot \overline{B} \). Write your answer in the form of JSim .subckt:

\[
\text{.subckt Q1 a b c z}
\]
\[
* \text{ your netlist statements go here}
\]
\[
\text{.ends}
\]

Question 2: Draw a schematic diagram and provide a truth table for the function implemented by the following JSim .subckt. This function turns out to be implemented directly by one of the gates in the gates library. Which one?

\[
\text{.subckt Q2 a b z}
\]
\[
\text{X1 b binv inverter}
\]
\[
\text{X2 a b binv z mux2}
\]
\[
\text{.ends}
\]
Question 3: Design a Q3 module with three inputs (A, B, Cin) and two outputs (S, Cout) based on the following truth table. Write your answer in the form of a JSim .subckt.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Cin</th>
<th>S</th>
<th>Cout</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Question 4: A binary-to-seven-segment decoder takes 4 bits of input and produces seven outputs, one for each “segment” in a standard display:

Given the appropriate binary input, this decoder produces outputs that light up the display in the following manner:

Given the appropriate binary input, this decoder produces outputs that light up the display in the following manner:

Construct a truth table for segment 3. Create a JSim .subkt that implements the necessary circuit to map the four inputs to the control signal for segment 3. You might find it easiest to use muxes to create a 16-entry lookup table implementation.

This completes the pre-lab. Please turn in your answers at the start of lab on Wednesday.
## Gates defined by `/mit/6.02/Labs/gates.jsim`

<table>
<thead>
<tr>
<th>Netlist template</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Xid z constant0</code></td>
<td>( Z = 0 )</td>
</tr>
<tr>
<td><code>Xid z constant1</code></td>
<td>( Z = 1 )</td>
</tr>
<tr>
<td><code>Xid a z inverter</code></td>
<td>( Z = \overline{A} )</td>
</tr>
<tr>
<td><code>Xid a z buffer</code></td>
<td>( Z = A )</td>
</tr>
<tr>
<td><code>Xid a b z and2</code></td>
<td>( Z = A \cdot B )</td>
</tr>
<tr>
<td><code>Xid a b c z and3</code></td>
<td>( Z = A \cdot B \cdot C )</td>
</tr>
<tr>
<td><code>Xid a b c d z and4</code></td>
<td>( Z = A \cdot B \cdot C \cdot D )</td>
</tr>
<tr>
<td><code>Xid a b z nand2</code></td>
<td>( Z = A \cdot B )</td>
</tr>
<tr>
<td><code>Xid a b c z nand3</code></td>
<td>( Z = A \cdot B \cdot C )</td>
</tr>
<tr>
<td><code>Xid a b c d z nand4</code></td>
<td>( Z = A \cdot B \cdot C \cdot D )</td>
</tr>
<tr>
<td><code>Xid a b z or2</code></td>
<td>( Z = A + B )</td>
</tr>
<tr>
<td><code>Xid a b c z or3</code></td>
<td>( Z = A + B + C )</td>
</tr>
<tr>
<td><code>Xid a b c d z or4</code></td>
<td>( Z = A + B + C + D )</td>
</tr>
<tr>
<td><code>Xid a b z nor2</code></td>
<td>( Z = A + B )</td>
</tr>
<tr>
<td><code>Xid a b c z nor3</code></td>
<td>( Z = A + B + C )</td>
</tr>
<tr>
<td><code>Xid a b c d z nor4</code></td>
<td>( Z = A + B + C + D )</td>
</tr>
<tr>
<td><code>Xid a b z xor2</code></td>
<td>( Z = A \oplus B = A \cdot \overline{B} + \overline{A} \cdot B )</td>
</tr>
<tr>
<td><code>Xid a b z xnor2</code></td>
<td>( Z = A \oplus B = A \cdot B + \overline{A} \cdot \overline{B} )</td>
</tr>
</tbody>
</table>
| `Xid s d0 d1 z mux2` | \( Z = D0 \) when \( S = 0 \)  
\( Z = D1 \) when \( S = 1 \)  
\( Z = \overline{S} \cdot D0 + S \cdot D1 \) |
| `Xid s0 s1 d0 d1 d2 d3 z mux4` | \( Z = D0 \) when \( S_0 = 0, S_1 = 0 \)  
\( Z = D1 \) when \( S_0 = 1, S_1 = 0 \)  
\( Z = D2 \) when \( S_0 = 0, S_1 = 1 \)  
\( Z = D3 \) when \( S_0 = 1, S_1 = 1 \)  
(\textit{Note order of } s0 \textit{ and } s1!) |

To use one of these gates in your circuit, add a line to the netlist following the Netlist template shown for the gate. For example, to make an instance of a 3-input OR gate hooked to input signals Charlie, Chris and Hari, with an output of You:

\[ \text{Example} \text{ Charlie Chris Hari You or3} \]
In the lab (Wed., April 2, 2008)

Your task is to design a decode module that does error detection and correction on a (15,8,4) codeword with at most 2 bits of error and produces two outputs: an 8-bit corrected data word and an error signal indicating that the codeword contains a 2-bit error. The codeword has been encoded using the same (15,8,4) block code we used in Lab 5.

The decode module has fifteen inputs:

- \( d_1 \) \( d_2 \) \( d_3 \) \( d_4 \) \( d_5 \) \( d_6 \) \( d_7 \) \( d_8 \)
  - The eight data bits of the codeword. For the parity calculations think of these as being organized as two rows of four bits each: \( d_1, d_2, d_3, d_4 \) on row 1; \( d_5, d_6, d_7, d_8 \) on row 2.
- \( r_1 \) \( r_2 \)
  - The two row parity bits (even parity: \( r_x \) was set by the transmitter to make the number of 1’s in row \( x \) even)
- \( c_1 \) \( c_2 \) \( c_3 \) \( c_4 \)
  - The four column parity bits (even parity: \( c_x \) was set by the transmitter to make the number of 1’s in column \( x \) even)
- \( p \)
  - The overall parity bit (even parity: \( p \) was set by the transmitter to make the number of 1’s in the entire 15-bit codeword even).

There are nine outputs:

- \( \text{two-bit-error} \)
  - A 1-bit signal indicating than a 2-bit error has been detected in the input codeword. Error is set to 1 if the codeword contains a 2-bit error, 0 otherwise.
- \( \text{dout}_1 \) \( \text{dout}_2 \) \( \text{dout}_3 \) \( \text{dout}_4 \) \( \text{dout}_5 \) \( \text{dout}_6 \) \( \text{dout}_7 \) \( \text{dout}_8 \)
  - The eight corrected data bits. Note that their values are unspecified if a 2-bit error has been detected (error = 1), so your logic can do anything that’s convenient in this case.

To get started, set up the 6.02 environment and copy over the files you’ll need:

```
athena% add 6.02
athena% cp /mit/6.02/Labs/Lab6/lab6_template.jsim lab6.jsim
athena% jsim lab6.jsim &
```

lab6_template.jsim is a skeleton circuit file that .include’s two files: one that defines the gate library and another that sets up a test jig for your circuit. It also contains a .subckt
The test jig in lab6checkoff.jsim creates an instance of the decode module which it then tests on 408 different codewords during a simulation run that lasts for 40800ns, supplying a test codeword every 100ns. You can look in the test jig file to see a listing of the test codewords and the expected results.

Try running the template code as-is: click on \( \square \) to start the simulation. Once the simulation is complete, you can check the results by clicking the verify button (\( \checkmark \)). The test jig file also contains information about the expected results and JSim compares those results with the results from your decoder module, reporting any differences it finds. In this case, the comparisons are made 99ns after each test codeword has been supplied to the decoder module. With the dummy decoder module supplied by the template JSim reports:

![Checkoff failure](image)

indicating that at a simulated time of 99ns node \texttt{error} was expected to have the value 0 but actually had the value 1 (in other words, the dummy circuit “detected” an error but the test codeword actually didn’t have a 2-bit error).

Here are some suggestions about how to proceed:

1. Create modules to do the necessary parity checks: XOR3 for checking column parity, XOR5 for checking row parity and XOR15 for checking overall parity. Each module simply computes the XOR of all of its inputs, so its output will be 1 if there aren’t an even number of 1’s on its inputs, i.e., the outputs are 1 if a parity error has been detected. You can build these modules using the XOR2 gate supplied in the library and perhaps instances of each other (e.g., the XOR15
module can easily be built from XOR3 and XOR5 modules).

2. Wire up instances of the XOR3, XOR5 and XOR15 modules to perform the necessary parity checks, generating row, column and overall error signals.

3. Create a module called FIX that has three inputs: one column error signal, one row error signal and one incoming data bit. The module has one output: the corrected data bit. Hint: if the two error signals are both 1, the output is the inversion of the incoming data bit, otherwise the incoming data bit is passed through without change. You can implement this module using just two gates!

4. Create a module called CORRECT that has 7 input signals: the two row error signals, the four column error signals, and the overall error signal. The module has one output: a signal that’s 1 if a 2-bit error has been detected.

Here’s a diagram showing the internals of the decode module:

After completing your design, run a simulation to have the test jig verify that it’s operating correctly. Once your decode module passes all the tests, you’re done! Call over a staff member to get your circuit checked off.
Check-off Sheet for 6.02 Lab #6

Names of team members: _______________________________

Check-off for decode module (staff initials): ______________________________

Post-lab interview questions:

   a. Draw a schematic diagram or write a Boolean equation for your CORRECT module. Use it to explain your logic for the ERROR signal.

   b. Assume that each gate in the gate library has a propagation delay of 1. What’s the propagation delay for your decoder module?

   c. Suppose one wanted to decode words at twice the rate as your decoder module works. Describe a way to use two copies of your decoder module to get the job done.

Interview signoff (staff initials, score): _________________________________