

## CSC211 Laboratory A

### Background Information

In this lab, we return to Logisim. If you need a refresher on it (eg, how to run it, where to find the on-line tutorial for it), take a look at Laboratory 5 on shift registers and counters.

Here are two additional tips regarding Logisim:

- *Save your work early and often* because Logisim occasionally crashes during a save operation.
- Occasionally, Logisim may incorrectly label some lines within your circuit as having an error condition (coloring the lines red). When this happens it seems that saving the file, closing Logisim, and then restarting Logisim will clear the error.

### Laboratory Exercises

1. Recall, or review, the Hamming single-error correction code we studied in class. With this scheme, we create a codeword that includes both data bits and check bits for each data word we wish to store. When we read this codeword back from memory, we recompute the check bits and compare them with the stored check bits. If differences are found, we know that at least one bit was corrupted in memory. If only one bit was corrupted, we can determine which bit it was and correct it.

In this laboratory you will create a circuit that implements this scheme for 4-bit data words. Recall that this requires generating a 7-bit codeword that includes 4 data bits and 3 check bits. To enable you to test your circuit, you should begin by working out on paper the codewords that correspond to several input datawords. Your examples should include some pairs of data words that have Hamming distance one from one another (ie, they differ in just one bit). Other pairs should differ by two bits.

Recall that the order of bits in the codeword should be:  $D_4 D_3 D_2 C_4 D_1 C_2 C_1$ . Also, each check bit should “cover” a specific set of data bits based on their location within the codeword. We will use even parity, so the value of each check bit should be that which makes the number of ones in the set of covered bits even.

2. Start a new project in Logisim, and create a circuit that accepts four input (data) bits and outputs the corresponding 7-bit codeword. Note that the following boolean expression can be used to compute even parity, given that check bit  $C_1$  is being used to cover data bits  $D_1$ ,  $D_2$ , and  $D_4$ .

$$C_1 = D_1 \oplus D_2 \oplus D_4$$

Test your work based on the codewords you computed manually in part 1.

3. Create a separate subcircuit in Logisim that computes a syndrome word based on the check bits associated with two codewords. In other words, this circuit should accept 6 input bits (three from each codeword) and determine which check bits differ between the two codewords. Recall that if the resulting syndrome word is 101, this indicates that the two versions of  $C_4$  and  $C_1$  differ, but the two versions of  $C_2$  match.
4. Here you will create a main circuit that uses your two previous subcircuits in an single-error correction scheme. To do this your circuit should:
  - (a) Pass 4 input (data) bits to your codeword generator subcircuit. Use input toggle switches for these inputs, so you can set their values manually.
  - (b) Store the output codeword into 7 D-flipflops. You will want to use individual flipflops here, rather than using a Logisim register, because this will allow you to manually modify (corrupt) one of the stored values.

- (c) So far, your circuit simulates saving data into a memory register. Next, to simulate reading the data back out and recomputing the check bits based on the stored data bits, add a second codeword generator subcircuit. It should accept the data bits stored in the D-flipflops and generate a new codeword from them.
- (d) Now add a syndrome word generator subcircuit. It should compare the check bits stored in the D-flipflops with the recomputed check bits. You may want to display this output using a set of Logisim “output pins” (ie, circular LEDs) to make them easily visible.

To test your circuit: give it an input datum, “clock” this datum into the D-flipflop register, and then modify (corrupt) the codeword stored in the register. The output syndrome word should reflect the bit position of the bit you modified. Further, if you correct that bit by (again) modifying the value in the corresponding D-flipflop, the syndrome word should display 000.

5. In this exercise, you will enhance your circuit to produce a SEC-DED (single-error correcting, double-error detecting) circuit. I suggest that you save a separate copy of your SEC circuit as is before continuing.

To create a SEC-DED circuit:

- (a) Enhance your codeword generator subcircuit so that it outputs a fourth check bit. The value of this check bit should be that which makes the number of ones in the entire (8-bit) codeword even.
- (b) Store the additional check bit as bit 8 in your D-flipflop register.
- (c) Enhance your syndrome word generator to also accept the fourth check bit from each input codeword, and to produce a new output bit in the high-order position. This bit should be a 1 iff the three lower-order bits of the syndrome word are 0 and the two new input check bits differ.

The logic behind this is that the three lower-order bits will be 0 if there are no errors in the stored data, or if there were two errors in the stored data and we then “fixed” the erroneously named single error. In the former case the stored and reconstructed C8 bits will match, but in the latter case they won’t.

To test your enhanced (SEC-DED) circuit:

- (a) Input a new data word, clock the corresponding codeword into your D-flipflop register, and then corrupt two of the stored bits.
- (b) Note that the syndrome word now assumes that a single bit was corrupted and indicates which bit could have caused the observed symptoms. Now “fix” the bit indicated by the syndrome word, knowing that doing so will actually cause a third error.

When you do so, the syndrome word should become 1000. This indicates there is still a problem: correcting a single bit did not make the two codewords match, so there must have been two corrupted bits rather than one.