CSE 140L

Lab Assignment 3

Error Correction Circuitry and Boolean Simplification

Due February 13, 2014, 4:00pm

Introduction:

This lab assignment consists of two parts. In the first part, you are asked to implement an error detection/correction circuit by slightly altering the Booth's multiplier circuit you have designed in your Lab2 assignment. We will provide you with a correctly functioning multiplier circuit for this part in case you do not have a fully working circuit handy from the second lab assignment. In the second part, we ask you to implement a program in Java that simplifies Boolean equations according to the absorption and consensus theorems and generates its corresponding Verilog code.

Part 1: Error Detection/Correction Circuit

Physical failures of digital systems can lead to errors during data transmission. Oftentimes there may be a difference between the data that is transmitted and the data that is received. In order to deal with such problems, one can use some sort of coding system when transmitting data. One such code is the *2-out-of-5* code introduced in Chapter 1 of your text book (p. 21). Although the *2-out-of-5* code is inefficient in terms of the number of bits used, the code *does* allow you to detect all single bit errors. Keep in mind that the ability to detect errors is based on the *Hamming distance* between code words. In addition to the ability of detecting errors, some codes even allow errors to be corrected as well.

In this part of the experiment, you will build an error detection/correction circuit for the coding scheme in Table 1 by using LogicWorks. You see only 10 code-words in this coding scheme, represented by 15 bits each. This is indeed a rather inefficient scheme as only 4 bits would more than suffice to represent 10 different words but don't despair; the redundancy in these code-words will deliver certain error detection and correction capabilities.

Table 1: Error detection/correction codewords

0	001111000110100	-0	110000111001100
1	001110100111000	-1	110001011010001
2	001101101011000	-2	110010011001001
3	001011101101000	-3	110100011000101
4	000111101110000	-4	111000011000011

1.1 Theory

Invariants

To implement the appropriate detection and correction circuit, you need to understand the properties of these codewords. These properties are typically referred to as *invariants* in coding theory; for example, the invariant of the *2-out-of-5* code is that there are exactly two 1's in any codeword (and consequently three 0's). This invariant enables the *2-out-of-5* code to detect single bit errors, as a single flip of any bit will violate the invariant of exactly two 1's.

On the code that we show you above there are actually two *invariant* properties embedded;

1. *Total number of 1's:* each code-word consists of exactly eight 0's and seven 1's. This invariant alone provides a single error detection capability, in a manner similar to the *2-out-of-5* code case. Actually, a moment's reflection yields the fact that this invariant helps detect any *odd* number of errors.
2. *Sum of indices:* if we assign an index of 1 to 15 to each bit (from least significant bit to most significant) and perform a summation of indices where a 1 appears and subtract the indices where a 0 appears, we get a sum of 0.

   Examples:

Indices: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1	Indices: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
Codeword -4: 1 1 1 0 0 0 0 1 1 0 0 0 0 1 1	Codeword 3: 0 0 1 0 1 1 1 0 1 1 0 1 0 0 0
Indices for 1s: 1+2+7+8+13+14+15=60 (Sum1)	Indices for 1s: 4+6+7+9+10+11+13=60 (Sum1)
Indices for 0s: 3+4+5+6+9+10+11+12=60 (Sum2)	Indices for 0s: 1+2+3+5+8+12+14+15=60 (Sum2)
Sum1-Sum2 = 0	Sum1-Sum2 = 0

Value Embedding property

A moment's reflection (oh, well, maybe a bit more than a moment) should convince you that the assignment of values to codewords is not arbitrary and actually you can identify the value of the codeword by summing the appropriate bits. If you stop the accumulation for an instance, at the end of summing the least significant 7 bits, you would at that point be staring at twice the value of the codeword. Well, let's try it and see.

Examples:

Indices:	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
Codeword 3:	0 0 1 0 1 1 1 0 1 1 0 1 0 0 0
Indices for 1s:	4+6+7=17 (Sum1)
Indices for 0s:	1+2+3+5=11 (Sum2)
Value:	(Sum1-Sum2)/2 = (17-11)/2 = 6/2 = 3

Indices:	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
Codeword -4:	1 1 1 0 0 0 0 1 1 0 0 0 0 1 1
Indices for 1s:	1+2+7=10 (Sum1)
Indices for 0s:	3+4+5+6=18 (Sum2)
Value:	(Sum1-Sum2)/2 = (10-18)/2 = -8/2 = -4

Other properties of the code:

Besides the value embedding property, the codewords listed in Table 1 have some additional properties. A quick comparison of the two codewords on each row of Table 1 should convince you that the codewords have a sign-magnitude property, wherein every positive number has an MSB of 0 while every negative number has an MSB of 1. Additionally, with respect to the value-embedding property, the number of 1's in the least significant 7 bits is three.

1.2 Computing Invariants, Deciphering Codeword Values

Now that we have identified two invariants and the Value Embedding property, we need algorithms to calculate them. Interestingly, it turns out that some slight alterations to your Booth's multiplier from Lab 2 can help you compute them. Of course your alterations should be limited so we constrain you to keep intact the inputs to F, the flip-flop containing your right-most examined bit. You are, however, allowed to change the logic controlling when an operation is performed and which operation is performed. The interesting part of this exercise in this lab is the use of Booth's multiplication circuitry to help check these invariants. Booth's multiplication does a right-to-left sweep of the multiplier in pairs with each bit element of the pair being examined twice. To accommodate this pairwise sweep, we had already inserted an extra right bit (initialized to 0) to the Booth's multiplication circuit. It turns out that an additional leftmost bit needs to be inserted as well (also initialized to 0) when we use the circuitry to do the invariant calculation for our error correction code. We will use this extra bit as well when we illustrate the invariant calculation. The insertion of dummy bits results in a bit string of the form 0xxxxxxxxxxxxxx0 as input to a Booth's-like right-to-left traversal, where xxxxxxxxxxxxxxx is our original 15-bit input.

Calculating Invariant #1:

Invariant #1 necessitates counting the number of 1s (or equivalently the number of 0s) in order to check for any codeword violations. Perhaps a way to do this would be to fix the value in the multiplicand to 1 and then effectively increment the accumulator by adding the multiplicand every time we see a 1. Yet Booth's multiplication effectively jumps over a whole bunch of 1s and a whole bunch of 0s, thus dooming this approach to failure. Now we know that when checking for invariant #2, the multiplicand would need to be incremented every step to reflect the index value of the bit. Since we need to have such multiplicand incrementation properties built in for computing invariant #2 anyways, perhaps we can try for fun to see how Booth's multiplication

hardware would behave with this multiplicand incrementation. We know that we would be subtracting the index of the left bit when we encounter the pair “10” and adding when we see the pair “01”. We do not perform any operations when the sequence is “11” or “00”. Observe what happens when we apply this set of rules, examining 2 bits at a time, moving from right to left.

Indices: 16 | 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 | 0
Codeword -4: 0 | 1 1 1 0 0 0 0 1 1 0 0 0 0 1 1 | 0

If we work on Codeword -4 as an example, these rules will result in: -1+3-7+9-13+16 = 7

Indices: 16 | 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 | 0
Codeword 1: 0 | 0 0 1 1 1 0 1 0 0 1 1 1 0 0 0 | 0

Similarly, working with Codeword 1 will result in: -4+7-9+10-11+14=7. Perhaps you can try a couple of examples yourselves, but essentially it should be evident that performing the calculations on any codeword will result in a sum of ‘7’, which happens to be the total number of 1’s in the codeword! How have these rules managed to count the number of 1s in the codeword? Observe the sequences and the marked indices below:

Sequence: 0 1 0	Sequence: 0 1 1 0	Sequence: 0 1 1 1 0
Indices: +2 -1	Indices: +3 -1	Indices: +4 -1

If you ponder for a moment these examples, you should notice that in a sequence of consecutive 1’s, subtracting the index of the rightmost ‘1’ and adding the index of the zero at the left hand side of the sequence gives the number of 1’s in that sequence. The rules just described above therefore count the number of 1’s in each sequence of 1’s in the codeword. Totaling them across the complete codeword should give you consequently the total number of 1s in the codeword!

Calculating Invariant #2:

Well, if the Booth’s-like algorithm with the multiplicand incremented at each step managed to count the number of 1s in the codeword, how are we to obtain the sum of the indices in this case? A moment’s observation should tell you that applying Booth’s addition/subtraction rules as in the original algorithm fails to obtain this sum, since it skips over a slew of 1 and 0 bits. Obviously, one could modify the rules such that each “11” or “10” pair performs an addition of the right index while each “01” or “00” pair performs a subtraction of the index, resulting in an algorithm that would be equivalent to the invariant calculation. However, the goal with Booth's multiplication is to reduce operations, so it is in our best interest to turn two of these cases into “noops.” As you realize, turning “00” and “11” into a “noop” is nearly identical to Booth's algorithm, which we used for Invariant 1 and will thus not produce a usable result for the computation of Invariant 2. However, perhaps if we instead do nothing for the “10” and “01” bit strings while adding for “11” and subtracting for “00”, we could come close to a usable result. Let’s observe what happens when we apply these modified rules to this sequence. We again will use our modified codewords here in the form of “0xx..xx0”.

Indices: 16 | 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 | 0
Codeword 0: 0 | 0 0 1 1 1 1 0 0 0 1 1 0 1 0 0 | 0

So for codeword 0 given above, these rules will result in: -1-2+6-8-9+11+12+13-15-16= -9.

Indices: 16 | 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 | 0
Codeword -2: 0 | 1 1 0 0 1 0 0 1 1 0 0 1 0 0 1 | 0

If we examine the codeword that denotes -2 in the same way, the sum of indices will be: -3-6+8-10-13+15= -9. Again a few more trials of your own would convince you that all the codewords listed here generate an identical result of -9.

Yet when defining the second invariant, we calculated the difference of Sum1 and Sum2 above, and found the result to be **0**. Now we have a result of **-9** in this part; how are these two values related? First, we should account for the effect the additional virtual bits have on our original invariant. Even though we are adding 2 virtual bits, we are actually only processing bits 1-15 and using the rightmost virtual bit solely for context, so the 0 should not change our result. Yet since the 16^{th} (the leftmost, extra) bit is always 0, the expected sum is now reduced to yield **-16**, since that additional operation would subtract the positional value of the 16^{th} bit from the expected invariant of 0. But neither does the sum of -16 that this extra mental adjustment yields match the observed sum of -9. The question arises as to how to account for the 7 units of difference between the now expected -16 and the observed sum of -9. To understand this part of the puzzle, we need to take a somewhat more precise look as to what we are counting when we apply this Booth's-like calculation method for our Invariant 2. As you recall, the second invariant computation would have yielded the correct result had we performed an operation for each possible pair of combinations. Yet we skipped over the computation in the cases of "01" and "10". What is the impact of these missing computations?

If we have a sequence such as 010, where 1 is at the n^{th} index, with our new algorithm, we skip the addition of index **n** ("10" is a no-op) and the subtraction of index (**n+1**) ("01" is a no-op as well). This is in comparison to simply adding when encountering a '1' and subtracting at a '0'. Skipping these operations will result in an invariant of **s-n+n+1**, or **s+1**, where **s** denotes the original invariant. When we have a sequence such as 01110, this time we skip the addition at **n**, where **n** is the rightmost 1 in the sequence, and we also skip the subtraction at (**n+3**). This time we will have an invariant of **s-n+n+3**, that is **s+3**. Essentially, in a sequence 01..10, we will have an invariant of **s+k**, where **k** is the total number of 1s.

So this difference between the original invariant described and the invariant this Booth's-like algorithm computes is always the number of 1s in the word, since the skipping of the operations for "01" and "10" boosts the expected sum by the number of 1s in the codeword. So, here in our case, we will always get a sum that is 7 (the number of 1s in our codewords) more than the original invariant described in Section 1.1. When one puts together the -16 that is inserted due to the leftmost extra bit and the +7 adjustment due to the vagaries of the Booth's-like calculation method for our invariant, it should be then obvious that the expected sum should be -9. Take a

look at the example below:

Indices: 16 | 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 | 0
Codeword 0: 0 | 0 0 1 1 1 1 0 0 0 1 1 0 1 0 0 | 0
Expected Sum: -1-2+3-4+5+6-7-8-9+10+11+12+13-14-15-16 = -16

The invariant calculated on the modified codeword (0xx..xx0) by the Booth's like algorithm:

-1-2+6-8-9+11+12+13-15-16 = –9

Try the same calculations yourself for several codewords and see that the sums are always -16 and -9 respectively.

Computing the Value Embedding property:
Since the embedded value is attained by summing up the indices for the least significant seven bits, we can apply exactly the same set of rules for invariant 2 checking in the calculation of the embedded value. However, instead of performing 16 add/subtract operations as in calculating invariant #2, only the first 8 operations are needed while the effect of all the subsequent operations must be ignored. However, several complexities arise when performing this computation. Firstly, bit 8 will now have to be used as the extra leftmost bit, meaning it must be padded with 0 for the purpose of the embedded value computation. Additionally, because there are 3 ones within the least significant 7-bits of every codeword, our embedded value (actually, twice the embedded value) will be off by a value of -8+3= -5. This means we will need to add 5 to our obtained sum before performing a division by 2 to yield the embedded value. Let's supplement this theory with the following examples:

Indices: 16 | 15 14 13 12 11 10 9 8 **7 6 5 4 3 2 1** | 0
Codeword 0: 0 | 0 0 1 1 1 1 0 0 **0 1 1 0 1 0 0** | 0
Indices: 8 | 7 6 5 4 3 2 1 | 0
Least Significant 7 bits: 0 | 0 1 1 0 1 0 0 | 0
Booth's-like embedded value calculation: -1-2+6-8 = -5

As we can see, (-5 + 5)/2 = 0, which is the embedded value.

Indices: 16 | 15 14 13 12 11 10 9 8 **7 6 5 4 3 2 1** | 0
Codeword -2: 0 | 1 1 0 0 1 0 0 1 **1 0 0 1 0 0 1** | 0
Indices: 8 | 7 6 5 4 3 2 1 | 0
Least Significant 7 bits: 0 | 1 0 0 1 0 0 1 | 0
Booth's-like embedded value calculation: -3-6 = -9

Here, (-9+5)/2 = -2, our embedded value!

Error Detection and Correction:

How can the rules described above assist in detecting errors? Recall that all codewords have two common invariants. For invariant #1, all codewords have a total of seven '1's. Thus, if any odd number of errors occurs, this situation will change the number of 1s in the codeword,

signaling an error. For invariant #2, an error may change the sum of indices when the rules are applied. Due to varying the indices among distinct bits, any double errors would also result in an incorrect summation result of Invariant #2. Such a fact provides us the capability of detecting double errors. From the discussion above, it can be seen that our codewords form a distance-4 coding scheme, and the two invariants in combination guarantee the detection of single, double and triple errors, and the correction of any single error. So in order to detect and correct the errors, what we need to do is to calculate the invariants for a codeword, and compare the actual result with the expected one for each invariant.

For error detection the procedure just described is sufficient. We need to perform an additional step though if we are interested in correcting these errors. It is essential to understand how a single error changes the invariant in order to do the correction. We will be using invariant #2 for correction. Firstly, consider a word with three zeros at bit positions j+1, j and j-1.

j+1	j	j-1
0	0	0

According to our rules, this sequence will result in a subtraction for index j and another subtraction for j+1. This means they contribute a total of (-2j-1) to the calculation. Now let's see what happens when there is a bit flip at position j:

j+1	j	j-1
0	1	0

Here, since the 2 bit sequences we observe are "01" and "10", we do not perform any operation and the contribution of this word is 0, indicating a boosting in the overall total by 2j+1. So, a bit flip in position j alters the overall result by changing the computations for both j and j+1 indices. To try another example, consider a codeword containing the sequence '110'. The contribution of these bits to the sum consists of the operations at positions j and j+1, which are a no-op and an addition of (j+1), respectively. Flipping the bit at j (the middle bit) changes the sequence to '100', producing the operations of subtracting j followed by a no-op. Once again, the difference is (2j+1), even though this time around the sum has been reduced by (2j+1) instead since now the presumed error flipped a "1" to a "0". Try other initial values for j+1, j, and j-1 to verify that the change in value is always 2j+1 in magnitude.

Since the calculation for the erroneous codeword will differ by 2j+1 from the original sum, taking the absolute value of the difference of the original and erroneous sums and dividing it by 2 identifies position j, pinpointing the location of the error.

Absolute value: $|-2j-1| = |2j+1| = 2j+1$
Divide by 2: $\text{floor}\ \{(2j+1)/2\} = j$

For the reason we have discussed, the original sum has an offset of -9. Thus the erroneous sum will be -9+(2j+1) or -9-(2j+1), which necessitates the elimination of the offset for the identification

of the error location. To attain this goal, we can initialize the accumulator to 9 instead of 0 at the beginning of the invariant 2 calculation. This non-zero initialization will compensate for the offset, delivering a final value of 2j+1 or –(2j+1) in case of a single bit error at position j.

1.3 Circuit Overview

As you may have noticed, calculations for both invariants are very similar to the calculations performed in the Booth's multiplier you implemented in Lab 2. For these invariants, we are again examining two bits at a time and performing an addition, subtraction or no-op accordingly. We need one additional bit in Booth's algorithm, which we stored by using a flip-flop (F). In this error correction exercise, we can utilize this flip-flop to keep our extra bit on the right. The 15-bit codeword needs to be expanded to a 16-bit Multiplier register to hold the codeword and the additional zeroed bit on the left (this is simply initialized to 0 and not sign extended). You should use your existing Lab 2 implementation as a basis for this lab unless your Lab 2 was not properly completed. We are providing you a Lab 2 implementation on the website in case you don't have a working Lab 2.

Given an input word, the circuit that you will implement should check whether the two invariants and the Value Embedding property are satisfied. You can implement the various checks through a *three-phase algorithm*. In the first phase, your circuit should compute the number of 1's and check if invariant 1 holds. The second phase examines invariant 2 and corrects any single bit error. The Value Embedding property will be examined in the third phase.

The Control Unit from Lab 2 is still useful here, as it will direct your circuit to perform 16 operations and shifts, but your circuit needs to go through the 16 operations three times, with each run checking a distinct invariant or a property. For this reason, we are providing you with a "Super Controller", which will attach to the inputs of your controller and simply direct it to perform its duties three times. It also has two outputs that lets your circuit know which phase it is in.

In Lab 2, the Multiplier register was shifted right at each step and 2 bits were examined to determine the operation. The same idea applies here to our input codeword. But the specific arithmetic operations determined by these 2 bits vary across computation phases of the error detection/correction circuit. In Phase 1 where the first invariant is examined, the operations are the same as the standard Booth's algorithm, whereas in the second and third phases, the mapping between the 2-bit pattern and the arithmetic operations is flipped, thus necessitating extra glue logic in the datapath to perform the correct operations in each phase.

We no longer need a Multiplicand as part of our calculation, but the value that we are adding at each step to the accumulator must start at 1 and increment on every shift cycle. In lab 2, the accumulator was shifted right after every operation because it was part of a positional number system. We are now simply summing a set of numbers with no underlying positional semantics, thus no longer needing the shift of the accumulator. What about the shifted-in bit of the Multiplier? Does the value shifted into the Multiplier matter?

1.4 The Control Unit

Since the algorithm consists of three phases now, the original controller that we had for Lab2 may need to be assisted a bit. The basic operations within each phase are essentially the same, so rather than implementing a new controller, we add a master controller, called the *SuperController.* The SuperController is responsible for controlling our original controller of Lab 2 and it also keeps track of the phase information. While connecting the two control units, you should pay attention to the following:

- The SuperController will begin the error detection/correction process when its **Begin** input makes a transition from 1 to 0, which will be controlled by the user (a switch). It will start the first phase by setting its **BeginPhase** output to 0. The Controller should take this input as its Begin input. The SuperController also broadcasts a two-bit signal [**Ph_bit1**, **Ph_bit0**] to indicate the current phase information. In this first phase, this two-bit signal outputs '01'.
- When the Controller raises the **End** signal for the first time, this shows the SuperController that the first phase has been completed. The SuperController has a **PhaseEnd** input to read this information. So it starts the second phase by setting its **BeginPhase** output first to 1 and then to 0 again. In the second phase, the SuperController outputs '10' at its [**Ph_bit1**, **Ph_bit0**] output, indicating that we are currently in the second phase.
- The third phase starts the same way as the second phase. The only difference is that in this phase, the [**Ph_bit1**, **Ph_bit0**] signal outputs '11'.
- The output signals and the cycle information within each phase are the same as Lab 2, i.e., you can refer to the table in Lab 2 to remember what happens at which cycle.
- When the Controller raises its **End** output to 1 at the end of the third phase, the SuperController realizes that this is the third and final time that the **End** signal has been generated. The error detection & correction is over now and the SuperController sets its **End** output to '1' to indicate that operation is over.

1.5 Implementation Guidelines

Beyond the Booth's-like algorithm, each phase needs to have additional glue logic connected to the accumulator to perform some final operations to detect and correct errors.

Phase 1:

The only thing the Phase 1 datapath needs to do is to compare the accumulator with the constant value 7, which produces a one bit result (1 for correct, 0 for incorrect). At the end of Phase 1, however, the Phase 2 algorithm needs to reset the accumulator in order to perform its calculations, which means that the Phase 1 datapath has to ignore the content of the accumulator after the termination of Phase 1. You'll need a register to load this value until Phase 2 begins. The [**Ph_bit1**, **Ph_bit0**] signal from the controller can be used to control the loading operation of this register. The register can continually overwrite itself as the accumulator

changes until Phase 2 is reached; the last value in the register will be the one displayed when Phase 1 is complete. The output of this register can be compared with the number of 1's in the codeword (i.e., 7) to determine whether there are any odd-bit errors in the input codeword.

Phase 2:

In Phase 2, the circuit will examine our second invariant. Our goal is to be able to correct single-bit errors, as well as detect up to two-bit errors using this phase. In order to compensate for the offset effect we have discussed, the accumulator should be initialized to 9 at the beginning of Phase 2. This will ensure that the final expected invariant value would come out to zero if there are no single or double bit errors. Therefore, we can detect errors using this phase simply by comparing the summation to zero.

Since the accumulator will be initialized in the next phase, the Phase 2 result also needs to be stored in another register. This register should be able to continually overwrite itself as long as the error detection/correction process is in Phase 2, whereas its loading operation must be deactivated in other phases. Again, the [**Ph_bit1**, **Ph_bit0**] signal from the controller can be used to control the loading operation of this register. The comparison of the accumulator to zero will be connected to another binary probe in a manner similar to Phase 1.

What about correcting single-bit errors? As we've seen from the explanation of the invariant above, any single bit error will result in a summation whose absolute value is one more than twice the index of the corrupted bit. For example, if our input contains a codeword whose least significant bit is flipped (index 1), the summation after the appropriate adjustments for the leftmost extra bit and implicit count of 1s should yield either +3 or -3. Similarly, if the incorrect bit is at index 5, the summation should yield +11 or -11. How do we use this to correct the corrupted bit and output a corrected codeword?

There are several steps to follow: first we must take the absolute value of the accumulator's output; then we have to divide that result by two. The resulting binary value will provide us the index of the corrupted bit. After this, we have to find a way to flip the bit poinpointed by the index of the corrupted bit we just identified. Each step is explained below.

Absolute Value:

There are several ways we could perform an absolute value operation in a hardware implementation. Let us assume a numeric value represented by a series of bit lines in *LogicWorks*. The method we recommend is the following: An adder/subtractor can be programmed to either add or subtract its inputs. Imagine if we had the choice to perform A – B or A + B, where A is set to 0. If B is positive, we add, resulting in B, whereas if B is negative we subtract, resulting in –B, which for a negative B yields its magnitude. We could attach such an adder to the outputs of the accumulator to begin this branch of the datapath that will correct our codewords. The A/S line that determines whether a subtraction will be performed is to be controlled by the sign bit of the input word. This should provide you a device that effectively takes a single input into B and outputs its absolute value.

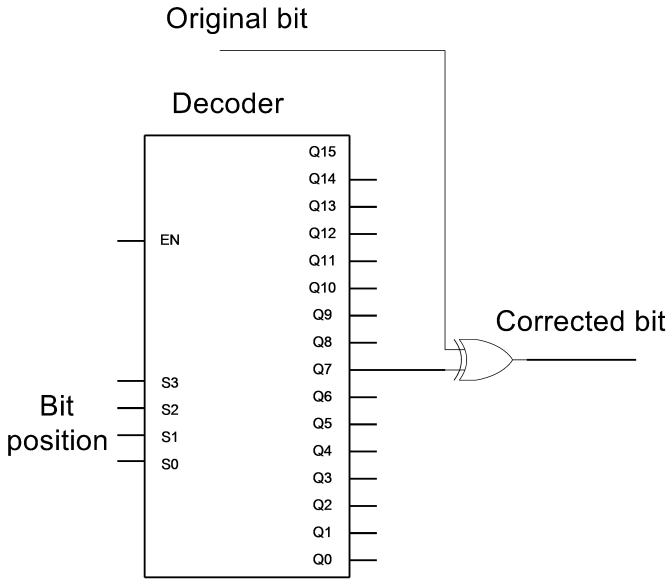
Divide by Two:

With binary numbers, multiplication and division with powers of 2 is very convenient, i.e. we add zeros to the end or cut off zeros from the right hand side. Here, we need to divide by 2 and take the floor, which is essentially equivalent to cutting the LSB off. For example, 1100 divided by 2 is 110; 0101 divided by 2 and floored is 010. So we need to eliminate the last bit of the absolute value result. We can accomplish this by simply connecting the adder output to the decoder (see correction part right below) accordingly.

Correction:

At this point in the datapath, we'll have a number that identifies the index of the corrupted bit of the input. We basically need to flip thereupon the erroneous bit and display the correct result. There are two parts to this process. First, what we have is a binary representation of the index of the faulty bit, whereas what we need is one line per bit of input. For example, if the third bit has been corrupted, our aforementioned computation at this point should yield the value "0011". We can use a decoder to transform this binary number into a form in which the third bit of the decoder outputs a value 1, correcting (by flipping) bit #3 of the codeword.

The next part of this process involves correcting the input codeword. By connecting each output pin of the decoder from Q1 to Q15 (Q0 is active when there are no errors) to one of the inputs of an XOR gate, and each bit of the input word to the other pins, we can perform conditional bit flipping on each line. This is shown in the figure below in the context of a 4-to-16-bit decoder.



For a bit that does not have an error, the decoder output pin corresponding to that bit of the input will be zero. If that is the case, the original input bit will remain unchanged. If the bit was corrupted, however, the decoder will output a 1 on this line, and whatever value came from the input will be flipped.

The outputs of the XOR gates can now be attached to hex displays and the corrected codeword displayed.

Phase 3:

The third phase calculates the embedded value of the codeword. Since the initial values for the counter for phase 2 and 3 differ, you will need to reset it to the correct value of 5. Moreover, since in phase 3 we are only summing up 7 of the bits in the word, you will need to implement the appropriate termination logic to ignore the bits that are not used in the embedded value calculation.

The controller will instruct the circuit to perform 16 summation steps in each phase, whereas in Phase 3, only the first 8 steps are needed. Therefore, we need a register to store the accumulator value immediately after the eighth summation step is performed. But how do we know which step the circuit is currently performing? Well, remember that we use a counter (in the multiplicand) to keep track of the index value of each bit for the first two invariants. This index value directly corresponds to the number of steps we have performed in each phase. Hence we can use its value to control the loading operation of the register. More concretely, the load input of the register should be activated only at the 'Load' cycle of the eighth step.

Finally, because you are using a separate phase to compute the embedded value, it is possible to calculate the embedded value of **the corrected codeword**, so we ask that you do so.

1.6 Components

Since the Lab 2 implementation is used as a basis for this lab, we inherit most components of the Lab 2 library. You will be replacing at least one of your components in this lab, and there are three new components you may need as well. We provide them to you in the Lab 3 library file.

1. **Lab 2 components that will still be used in this design**

You can use the 16-bit **multiplier** register in your Lab 2 implementation to store the 15-bit codeword and the leading dummy bit. The additional 0 at the right end can be stored in the flip-flop **F1**. The **accumulator** register will still be used to store the partial sum of your error detection computation. The **Adder/Subtractor** is still needed to perform addition or subtraction during error detection/correction. The **controller** in Lab 2 will be used as the datapath-controller in this design.

2. **Lab 2 components that will be replaced in this design**

The **multiplicand** register in the design of Lab 2 will be replaced by a counter as we need to increment our indices by 1 as we traverse through each bit of our codeword. As the lowest bit of our original codeword is denoted as index 1, this counter should start counting from 1. As we only need to count from 1 up to 16, a 4-bit counter barely misses the mark, necessitating an 8-*bit counter* that you can find in the *LogicWorks* standard library.

3. **New components that we will provide to you**
   - A **4-to-16 bit decoder** to be used in the design of the error correction logic.
   - A **SuperController** to be used for controlling the three-phase operation by indicating the start and the end of each phase. Within each phase, the operation of the circuit is actually controlled by the datapath controller (which is identical to the controller used in Lab 2). You need to hook up the two controllers so that they operate correctly.
   - A 2-to-1 multiplexer, **MUX-2x16**, which receives two groups of 16-bit inputs and selects one to output depending on the selection signal S.
4. **New components that you need to design and insert into the circuit**
   - The **Error Detection** logic which should output a 1 if an error is detected in either the first or the second phase
   - The **Error Correction** logic which corrects the input word based on the result of Phase 2
   - The logic that captures through the accumulator the embedded value of the codeword in Phase 3

1.7 Items to be Displayed:

Your circuit should display the following results:

- The input word (on 4 hex keyboards);
- The result of the first calculation (the number of 1s on a hex display);
- The result of the second calculation (the index summation on 2 hex displays);
- The corrected code word (on 4 hex keyboards) according Phase 2;
- Buzzer signal showing if a bit was corrected in Phase 2 (on a binary probe);
- The embedded value of the corrected codeword calculated in Phase 3 (on a hex display for the value and a binary probe for the sign);
- Buzzer signals showing the result of error detection in each phase (on binary probes).

Part 2: Boolean Simplification

This part of the lab will transport you back to the realm of software development, as you will be tasked with implementing a program in Java that simplifies **Two-Level Sum-of-Products** boolean expressions by utilizing **the absorption theorem** and **the consensus theorem**.

In order to implement this part of the lab, you will need to remember some additional vocabulary terms. An **implicant** is a product of one or more variables (e.g. *a*, *a'bc*), and a **Two-Level-Sum-of-Products** expression is a boolean expression that consists of only a sum of Implicants (e.g. *ab + a'c*). Implicants can be decomposed into smaller implicants (ones with more terms), and the smallest such decomposition for a given number of variables is called a **minterm**. For example, for a 5-variable function, the implicants *abc'de* and a'bc'd'e' would be considered minterms. This means that an **implicant** can be seen as a set of **minterms**.

2.1 Implicant Encoding

Let's explore the notion of implicants further through the implicant example of *ac'*. There are two visible variables within this implicant, *a* and *c*, where *a* is in its uncomplemented form and *c* is in its complemented form. What about *b*? A cursory glance at this implicant would yield the conclusion that *b* is not present within this implicant. However, as you recall from CSE 140, *ac'* is equivalent to *abc' + ab'c'*, which means that the variable *b* actually **is** present within the implicant, as it is present in **both** its complemented and uncomplemented form.

Given the aforementioned 3 states of a variable within an implicant, we introduce the *positional-cube notation*, which we will see is a mechanism for representing the state of variables within a given implicant in our program. These three cases require two bits to represent the state of each variable within an implicant. We ask that you use the following representation:

uncomplemented = 01
complemented = 10
both = 11

While the encoding might seem arbitrary, a moment of reflection will reveal to you that the two bits of the encoding follow a certain pattern. We can see that the most significant bit (MSB) of the representation will be a '1' if and only if the complemented form of the variable is within the implicant and is a '0' if it is not within the implicant. Likewise, a '1' in the least significant bit (LSB) denotes that the uncomplemented form of the variable is within the implicant with conversely a '0' at the LSB denoting its absence.

One question you might now ask is what '00' should correspond to. Given the observation in the previous paragraph, '00' would correspond to a variable that is present in neither its complemented nor uncomplemented form within the given implicant. But what does this mean? Let's go back to our example with *ac' = abc' + ab'c'*. Let's say we want to modify this implicant so that neither *b* nor b′ are present. This would require removing both *abc'* and *ab'c'* from the implicant, both of which are needed to make the implicant *ac'*. Thus, the resulting implicant with both *b* and *b'* removed is one in which **no variables** are present within the implicant, meaning that the implicant does not exist. So, we can thus consider '00' for any variable to be a pattern that renders the implicant **invalid**. Additionally, without loss of generality, if one variable has an encoding of '00' in an implicant, then we can replace the implicant such that all variables have an encoding of '00'.

Let's apply this notation to some examples. Let's take the 5-variable implicant *ab'e*. The variable *a* is not complemented, so it would be represented as '01', while the variable *b* is complemented, giving it a representation of '10'. The variables *c* and *d* are not shown, which means they should

be encoded as '11', and *e* is not complemented, meaning it should be again represented as '01'. So, the implicant *ab'e* would be represented as 01 10 11 11 01.

2.2 The Absorption Theorem

To illustrate the power of this encoding, we will guide you in utilizing it to implement the **absorption theorem**. As you recall, the absorption theorem dictates the following:

$$ab + abc = ab$$

If we convert the two implicants, *ab* and *abc* into their corresponding encodings, we obtain:

$$imp1 = ab = 01\ 01\ 11$$
$$imp2 = abc = 01\ 01\ 01$$

As we can see, because the *c* variable in the *ab* implicant is 11, it is present in the implicant in both its uncomplemented and complemented form. Meanwhile, the implicant *abc* contains only the uncomplemented version of *c*. This means that *abc* is a subset of *ab*, which is fundamentally the reason why it is redundant and can be eliminated from the boolean equation.

As you recall from CSE 20, a set S2 is a subset of S1 if S1 U S2 = S1. A dual subset check exists as well, in which S2 is a subset of S1 if S1 ∩ S2 = S2. Our encoding becomes useful when implementing this function, as a union can be effected through a **bitwise-or**, while an intersection can be effected using a **bitwise-and**.

Thus we can write a quick algorithm to determine subset relationships between implicants. Let's look again at *ab* (*imp1*) and *abc* (*imp2*). If we compute a bitwise-or of them, we obtain:

$$imp1 \mid imp2 = 01\ 01\ 11 \mid 01\ 01\ 01 = 01\ 01\ 11 = imp1$$

As you can see, the result of the union is simply the implicant *ab*, encoded as 01 01 11, which means that *abc* (*imp2*) is a subset of *ab* (*imp1*). Equivalently, one can utilize a bitwise-and to perform an intersection and obtain the same result, as shown below:

$$imp1 \ \&\ imp2 = 01\ 01\ 11\ \&\ 01\ 01\ 01 = 01\ 01\ 01 = imp2$$

Because the result of the intersection is equivalent to *abc* (*imp2*), we can again conclude that *abc* (*imp2*) is a subset of *ab* (*imp1*).The algorithms are summarized below in pseudocode:

Implicant Subset Algorithm 1 (returns true if imp2 is a subset of imp1):

```
isSubset(imp1, imp2):
    return (imp1 | imp2) == imp1
```

Implicant Subset Algorithm 2 (returns true if imp2 is a subset of imp1):

```
isSubset(imp1, imp2):
    return (imp1 & imp2) == imp2
```

A corner case to this algorithm exists when imp1 or imp2 possess two 0s in the same variable location, as this will render the implicant invalid, and performing a union or intersection on an invalid implicant may result in mistakenly obtaining a union that is otherwise valid. However, as we stated before, implicants with a pair of 0's can be replaced with one containing all 0's, and doing so will rectify this problem.

2.3 The Consensus Theorem

Our encoding will also prove to be useful in identifying consensus elements. As you recall from CSE 140, the standard consensus theorem and its corresponding proof are:

$$ab + a'c + bc = ab + a'c$$

$ab + a'c + bc$ (Given)
$ab + a'c + 1bc$ (Identity)
$ab + a'c + (a + a')bc$ (Complement)
$ab + a'c + abc + a'bc$ (Distributivity)
$ab + a'c$ (Absorption)

As you can see above, the implicant *bc* is decomposable into two implicants, *abc* and *a'bc*. Why can these two terms be eliminated? Well, the two other implicants, *ab* and *a'c* can both be decomposed into smaller implicants as well, where $ab = abc + abc'$ and $a'c = a'bc + a'bc'$. As we can see, *abc* is already present within *ab*, meaning that it is a subset of *ab* and is thus redundant. Similarly, *a'bc* is already within *a'c* and is redundant as well. So, the correctness of the theorem is due to the fact that the decomposition of *bc* into smaller implicants yields implicants that are **subsets** of two other implicants within the function. We can thus establish that *bc* is a **consensus element**.

Now the question revolves around how one leverages the aforementioned encoding to identify the consensus elements. Specifically, let's formulate the problem as follows:

Problem: given three implicants, *imp1*, *imp2*, and *imp3*, we want to return *true* if *imp3* is a consensus element of *imp1* and *imp2* and *false* otherwise.

How does one go about implementing an algorithm for solving this problem? In order for the problem to return an affirmative response, one must identify whether *imp3* can be decomposed into two smaller implicants, one of which is a subset of *imp1* and the other of which is a subset of *imp2*.

Let's tackle one portion of this question first; how can we find a portion of *imp3* that is a subset of *imp1*? Of course, since implicants are merely sets of minterms, an intersection of *imp3* and imp1 will yield this fourth implicant, *tempImp1*, that is a subset of both *imp1* and *imp3*. If we use the

aforementioned encoding, we can compute the intersection using a **bitwise-and** of the bits in the representation. For example, let's take the implicants *ab* and *bc*, which are represented as 01 01 11 and 11 01 01, respectively. The bitwise-and of these two implicants is:

01 01 11 & 11 01 01 = 01 01 01 = *tempImp1*

This is the encoded representation of *abc*, which just so happens to be the decomposed implicant we found in our proof earlier! Likewise, if we do the same thing for *a'c* and *bc*, we get as *tempImp2*:

10 11 01 & 11 01 01 = 10 01 01 = *tempImp2*

This is the encoded representation of *a'bc*, the other decomposed implicant in our earlier proof!

Now that we have our decompositions, *tempImp1* and *tempImp2*, we need to verify that they represent the entirety of our potential consensus element. This requires two steps.

Firstly, two implicants are decompositions of another implicant if and only if the two implicants differ by a single variable. For example, in the consensus proof, the decomposed implicants are *abc* and *a'bc*, which differ in a single variable, *a*. How do we check this? Well, for a variable to differ between two implicants, it must be present in its complemented form in one of the implicants and in its uncomplemented form in the other implicant. Because of our encoding, a variable in its complemented form is encoded as '10' while a variable in its uncomplemented form is encoded as '01'. An xor of these two values will produce '11'. What about the remainder of the variables? Well, since they should be identical between the two implicants, their xor should produce '00'. Thus, we can compute whether two implicants differ by a single variable by taking a **bitwise-xor** of the two implicants and verifying that the two bits corresponding to a variable are both 1's, as shown in the example below with *tempImp1*, 01 01 01, and *tempImp2*, 10 01 01:

01 01 01 ^ 10 01 01 = 11 00 00

As you can see, the two bits corresponding to the variable *a* are both 1, meaning that the two implicants only differ by the term *a*. As you can also see, the two bits corresponding to the remaining variables are both 0, meaning that they are identical between the two implicants.

Now that we know the two implicants differ by a single variable, we can perform the second step, which is to take the union of *tempImp1* and *tempImp2* and compare it to *imp3*. With the encoding, a union can be effected through a **bitwise-or.** So, if we take *tempImp1*, encoded as 01 01 01, and union it with *tempImp2*, 10 01 01, we obtain:

01 01 01 | 10 01 01 = 11 01 01

This happens to correspond to *imp3*, the implicant *bc*! We thus know that *bc* is a consensus element.

Let's try this algorithm on another example, where *imp1* is *a'b'* (encoded 10 10 11 11 11), *imp2* is *abcd* (encoded 01 01 01 01 11), and *imp3* is *b'cde* (encoded 11 10 01 01 01). If we want to determine if *imp3* is a consensus element we would first perform the intersection of *imp1* and *imp3* to obtain *tempImp1*:

10 10 11 11 11 & 11 10 01 01 01 = 10 10 01 01 01

Next, we take the intersection of *imp2* and *imp3* to obtain *tempImp2*:

01 01 01 01 11 & 11 10 01 01 01 = 01 **00** 01 01 01 = 00 00 00 00 00

Because *tempImp2*'s *b* variable is denoted with a pair of 0's, according to our aforementioned encoding, the implicant is invalid and can be rewritten as containing all 0's.

As you recall, however, the next step of the algorithm is to perform a bitwise-xor of the resulting intersections. However, if we recall why we were doing this, it was to verify that the two implicants could be combined accurately. However, if one of our implicants is the empty set, we can always combine it, so in this case, the bitwise-xor step is unnecessary.

Because we do not need to do a bitwise-xor, we can now take the union of *tempImp1* and *tempImp2*, and upon doing so, we obtain:

10 10 01 01 01 | 00 00 00 00 00 = 10 10 01 01 01

Because this is not equivalent to our potential consensus element, encoded as 11 10 01 01 01, it is not a consensus element.

In summary, the aforementioned algorithm for determining a consensus element can be summarized by the following pseudocode:

```
//determine if imp3 is a consensus element of imp1 and imp2
isConsensus(imp1, imp2, imp3):
        tempImp1 = imp1 & imp3
        tempImp2 = imp2 & imp3
        if (tempImp1 != 0 && tempImp2 != 0):
                if (tempImp1 and tempImp2 differ by more than one variable):
                        return false
        return (tempImp1 | tempImp2) == imp3
```

Being the astute CSE 140L students that you are, you have probably noticed that the aforementioned algorithm is actually a bit more powerful in identifying redundant elements than simply those mandated by the standard consensus theorem. For example, let's look at the following function:

$$ab + a'c + bcd$$

Since we know that *bc* a consensus element of *ab* and *a'c*, and since *bcd* is a subset of *bc*, we know that *bcd* is redundant, but this example does not specifically conform to the example of consensus in the textbook. The algorithm discussed previously, however, will indeed identify *bcd* as a consensus element, as shown below:

imp1 = *ab* = 01 01 11 11
imp2 = *a'c* = 10 11 01 11
imp3 = *bcd* = 11 01 01 01
tempImp1 = *imp1* & *imp3* = 01 01 11 11 & 11 01 01 01 = 01 01 01 01
tempImp2 = *imp2* & *imp3* = 10 11 01 11 & 11 01 01 01 = 10 01 01 01
tempImp1 ^ *tempImp2* = 01 01 01 01 ^ 10 01 01 01 = 11 00 00 00 = differ by one variable
tempImp1 | *tempImp2* = 01 01 01 01 | 10 01 01 01 = 11 01 01 01 = *imp3*

2.3 Removing Redundant Elements

Now that we know how to identify redundant elements via absorption and consensus, the next step is to remove them from our boolean expression. We ask that you implement a two-pass algorithm. The first pass will be used eliminate any redundant terms via absorption. The second pass should eliminate consensus elements. We ask that you use a greedy algorithm for doing this, in which you perform a left-to-right traversal on the implicant list, identify whether the element currently indexed is a consensus element of any remaining pair of implicants, and removing the implicant immediately if it is. This behavior can be characterized by the following pseudocode:

```
doSimplification(implicantList):
    //Iterate starting from the left to perform absorption
    for implicant in implicantList:
        for implicant2 in (implicantList – implicant):
            if isSubset(implicant, implicant2):
                implicantList.remove(implicant)
                break
    //Iterate starting from the left to perform consensus
    for implicant in implicantList:
        for every pair (imp1, imp2) in (implicantList – implicant):
            if isConsensus(imp1, imp2,implicant):
                implicantList.remove(implicant)
                break
```

While the above algorithm may look simple, special care must be taken when implementing it in Java, as the removal of an implicant from a list that is being iterated over can be error-prone if one is not careful. This is because removing an object from a list will implicitly shift the indices, which can cause your program to skip elements during an iteration. While we recommend using an iterator object in order to accurately implement this behavior, ultimately, the choice of implementation is up to you, so long as the algorithm produces the correct output.

2.4 Implementation Guidelines

We provide you with three Java classes, *Implicant.java* and *BooleanExpression.java*. In the *Implicant.java* class, you are tasked with implementing the **isConsensus** and **isSubset** methods. In the *BooleanExpression.java* class, we ask you to implement the **doSimplification** and **genVerilog** methods. We ask that you do not alter any existing coour de, as we will be unit testing both classes to verify that the structure of the classes are identical to how they are provided to you. However, you are of course free to add whatever helper functions you desire. We will document the important aspects of this starter code in this section of the writeup.

Implicant.java

This class is used to represent implicants. The major piece of understanding that you should note is the implementation of the encoding. In order to facilitate the computation of some aspects of your algorithm, we ask that you split up the representation into an MSB and an LSB and store them in two integers. For example, the implicant *abc* is **0***1* **0***1* **0***1*, and we ask you to store the bold and italicized portion of the encoding into separate longs, which are called *myMSB* and *myLSB* within the *Implicant.java* class. In the aforementioned example, the corresponding 3 bits of the *myMSB* value will be 000, and the corresponding 3 bits of the *myLSB* integer would be 111. One question you may have is what the remaining 61 bits of the

long should be, but since we know that an implicant implicitly contains both the complemented and uncomplemented forms of the remaining variables, it is obvious that the remaining bits should be set to 1. Thus, the implicant *abc* will have a *myMSB* value of 0xFFFFFFFFFFFFFFF8 and a *myLSB* value of 0xFFFFFFFFFFFFFFFF, with the relevant literals being placed in the least significant bits of the longs.

This representation simplifies two operations. One operation that is simplified is how one establishes whether an implicant is valid. As you recall, an implicant is invalid if at least one variable is represented with '00'. If this occurs in our new representation, then the index of myMSB and myLSB will be 0 at the same place. If we therefore perform a bitwise-or of the two values, we would get a result that has a 0 in some spot, whereas a valid implicant would have a 1 at every spot, giving a result of -1 (in two's complement). Thus, to establish the validity of an implicant, we simply need to bitwise-or the two longs and compare the result with -1.

The second operation that is simplified is the determination of whether or not two implicants differ by a single variable. Let's consider two implicants, *imp1* and *imp2*. As you recall, each implicant will have a *myMSB* and *myLSB* value. One can perform a bitwise-xor of the two *myMSB* and two *myLSB* values to obtain two new values. If the original implicants differ by one variable, these new values will both be equivalent and will both possess a single 1. Thus, the following algorithm will successfully check if *imp1* and *imp2* differ by a single variable:

```
differBySingleVariable(imp1, imp2)
        newMSB = imp1.myMSB ^ imp2.myMSB
        newLSB = imp1.myLSB ^ imp2.myLSB
        bitCountNewMSB = bitCount(newMSB)
        bitCountNewLSB = bitCount(newLSB)
        return (bitCoutNewMSB == 1) && (bitCountNewLSB == 1) && (newMSB == newLSB)
```

In Java, one can use *Long.bitCount* in order to count the number of 1's in a long.

BooleanExpression.java

This class represents a two-level sum-of-products function as a list of implicants, and its constructor is able to parse valid two-level-sum-of-product strings into this representation. The constructor builds each implicant's *myMSB* and *myLSB* values by embedding the literals into the necessary number of least significant bits, with all the remaining unused bits set to 1.

The **doSimplification** method that you have to implement will check to see if any element is redundant via the absorption or consensus theorems, and if it is, the method will remove the implicant. The removal should be done **in place**, meaning that the current *BooleanExpression*

element that you are performing the **doSimplification** function on should be modified with the consensus elements removed.

The other method that you need to implement is **genVerilog**. This function should output your boolean function into a sum-of-products form that you can then simulate in Modelsim in order to validate its behavior. We have already provided the skeleton of how the function should look; it is up to you to parse your list of implicants into the appropriate sum-of-products expression. Because your **doSimplification** method modifies your **BooleanExpression** object in-place, you can conceivably generate two verilog files, one before performing the simplification and the other after. You can then write a test bench to compare the outputs of these functions. To get you started, we have provided a skeleton for a testbench that you can use, *Lab3Test.v*.

Deliverables:

- **LogicWorks Files:**

  Part 1: As part of your submission, include a LogicWorks design file (.cct) that contains a complete error detection/correction circuit using the components provided. Be sure to test your circuit with several different kinds of inputs to detect problems.

  Part 2: Please submit all code required in order to run your program (excluding any Main.java wrapper classes that you used for testing). Be sure to test your code thoroughly.

- **Reports:**

  For part 1, you should submit a report explaining how you designed the error detection/ correction components of your circuit. You also need to provide explanations for the error detection and correction capabilities of your circuit. Specifically, you should examine the cases of 1-bit, 2-bit and 3-bit errors respectively, and report the behavior of each error detection/correction phase. In addition, for the following 15-bit words, specify whether they possess no error, a 1-bit error, a 2-bit error, or a 3-bit error: 1F34, 0F70, 62CC, 77C3

  For part 2, your report should detail how you implemented the four required methods. Additionally, in section 2.3, you were guided through the steps for implementing a greedy algorithm for removing redundant consensus elements that utilized a left-to-right traversal. However, greedy algorithms usually sacrifice optimality for time complexity. With this in mind, please provide answers to the following questions:

  - What is a boolean expression on which the greedy algorithm will **not** yield a minimal solution (in other words, will not yield a solution with the fewest implicants)?
  - What is a boolean expression on which performing a greedy right-to-left traversal instead will yield a solution with fewer implicants than the left-to-right traversal you implemented?