

Lab Assignment 3

Error Detection/Correction

Due November 9, 2014, Sunday, 4:30pm

Introduction:

This lab assignment consists of two parts. In the first part, you are expected to implement an error detection/correction circuit by slightly altering the modified Booth's multiplier circuit you have designed in your Lab 2 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. Second, you will implement a Java program to minimize a given Boolean expression with the absorption and consensus theorems. A basic framework for this program will be provided, and only the core functions need to be designed.

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 textbook (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 32 values in this coding scheme, represented by 18 bits each. This is indeed a rather inefficient scheme as only 5 bits would more than suffice to represent 32 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	010001111110001010	-1	110110000001101110
1	001001111110001100	-2	111010000001011110
2	000101111111000001	-3	111100000000111110
3	000011111111000010	-4	001101110101000011
4	010111000011110001	-5	001110110000111100
5	001111000011101100	-6	001111010000111010
6	001110100011110100	-7	001111100000110110
7	001101100011111000	-8	010111100000101110
8	101000011111100001	-9	011011100000011110
9	011000011111100010	-10	011101100000011101
10	010100011111100100	-11	011110100000011011
11	001100011111101000	-12	011111000000010111
12	100000111111110000	-13	101111000000001111
13	110100000111110001	-14	001101111000001110
14	101100000111110010	-15	001110111000001101
15	011100000111110100	-16	001111011000001011

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 we examined in your textbook 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 (and all odd number of 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 codeword consists of exactly nine 0's and nine 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 (just as in the 2-out-of-5 code) helps detect any *odd* number of errors.
2. **Sum of index values:** if we assign an index of 1 to 18 to each bit (from least significant to most significant) and perform a summation of indices where a 1 appears and subtract the indices where a 0 appears, the resulting sum is always 1.

Examples for Invariants follow:

Indices:	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
Codeword 15:	0	1	1	1	0	0	0	0	0	1	1	1	1	1	0	1	0	0
Count of 1s:	count(3, 5, 6, 7, 8, 9, 15, 16, 17) = 9																	
Count of 0s:	count(1, 2, 4, 10, 11, 12, 13, 14, 18) = 9																	
Invariant 1 =	Count 1 – Count 0 = 0																	

Indices:	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
Codeword -12:	0	1	1	1	1	1	0	0	0	0	0	0	0	0	1	0	1	1
Indices for 1s:	1 + 2 + 3 + 5 + 13 + 14 + 15 + 16 + 17 = 86 (Sum1)																	
Indices for 0s:	4 + 6 + 7 + 8 + 9 + 10 + 11 + 12 + 18 = 85 (Sum2)																	
Invariant 2 =	Sum1 – Sum2 = 1																	

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. You can actually identify the value of the codeword by summing the indices (in a 2's complement representation) by repeating the Invariant 2 calculation for the lower half (rightmost 9 bits) of the codeword and shifting the sum right once.

Indices:	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
Codeword -1:	1	1	0	1	1	0	0	0	0	0	0	1	1	0	1	1	1	0
Indices for 1s:	2 + 3 + 4 + 6 + 7 = 22 (Sum1)																	
Indices for 0s:	1 + 5 + 8 + 9 = 23 (Sum2)																	
Value:	(Sum1-Sum2)>>1 = (22-23)>>1 = -1>>1 = -1																	

In addition to these invariants, the codewords exhibit an additional property. At a glance, you will notice that each contains at least one long string of 0s or 1s. Were you to take all 18 bit numbers and select only those that satisfy both Invariant 1 and Invariant 2, you would notice

that the codewords in the table above have the longest possible strings of 1's or 0's. Since modified Booth's multiplication provides significant reduction in the number of operations required for multipliers with long runs of 1 or 0, we have selected codewords so that using a Booth's-like circuit to calculate the invariants will deliver speedups.

Calculating the Invariants:

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 Modified Booth's multiplier from Lab 2 can help you compute some of these invariants/properties. Of course your alterations should be limited so we constrain you to keep intact the inputs to FF, 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 Modified Booth's multiplication circuitry to help check these invariants. Because we need additional 'virtual' bits for Booth's-like operations, and due to issues related to calculating our invariants (explained below), we'll be operating on modified versions of the 18-bit codewords. Our input 18-bit code will be expanded by inserting a 0 to the right of the least significant bit and two 0s to the left of its most significant bit, giving us a numeral of the form 00xxxxxxxxxxxxxxxxxx0, where xxxxxxxxxxxxxxxxxxxx is our original 18-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 increment the accumulator every time we see a 1. Yet Booth's multiplication (and its extension to Modified Booth's) 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, 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 a set of rules, examining 2 bits at a time, moving from right to left.

Indices:	20	19		18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1		0
Codeword -4:	0	0		0	0	1	1	0	1	1	1	0	1	0	1	0	0	0	0	1	1		0

If we work on Codeword -4 as an example, these rules will result in $(-1+3-7+8-9+10-11+14-15+17 = 9)$.

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 '9', 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 '0' 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 1's in the codeword!

If you notice, the rule above is similar to what Booth's algorithm does. Once you understand how the invariant is calculated using the Booth-like algorithm, it will be easy to move to modified Booth's, where we examine 3 bits at a time. As you remember, modified Booth's is essentially identical to Booth's algorithm except for 2 special cases; therefore, all the rules still apply as in the Booth's-like algorithms but for the special cases, '010' and '101', which require special handling.

For these special cases, like it or not, the next shift will toggle the value being put into flip flop FF, which means 101 after shift will become x11, while 010 will become x00 after the shift operation. How does this affect the operation of invariant calculation? Let's look at how Booth's behaves in these cases and then decide what we need to do in modified Booth's to get the same result. This is what happens for these cases in the standard Booth's algorithm:

Index: j+1	j	j-1	Index: j+1	j	j-1
Case: 1	0	1	Case: 0	1	0
Actions: $-(j+1)$,	$+j$	$= -1$	Actions: $+(j+1)$,	$-j$	$= +1$

In order to produce the same results, what we need to do is to add one to or subtract one from the accumulator for the next two actions. Since the complemented FF input cancels the next operation, this is straightforward: we just do this immediately. Notice that this results in a reduction of the number of operations, just as Modified Booth's does for multiplication. The reasoning behind adding 1 for the '010' case is straightforward; when we cancel the middle 1 in the flip flop, we will essentially be removing a 1, which requires us to add it back when we detect the '010' case. For the '101' case, the reasoning is a little more complicated; the value going into the flip flop will be a '1' that should not actually be counted, so we need to remove it from consideration.

Another thing to give a thought to is the issue of why are we using two virtual zero bits to the left of our codeword? Doesn't one suffice, since we only need a numeral to our left when examining bit 18? Imagine that we are computing the number of 1s in a codeword that begins with two 1s (such as Codeword 13). If we use one constant 0 to the left of the codeword and process bits 1-18, the last case handled would be '011', which according to the rule defined above, would not perform the final addition necessary to compute the size of the sequence of 1s. By inserting two 0s on the left of the codeword, we get '001' as the last case handled, which is guaranteed to properly complete our calculation.

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?

So far, we have shown how invariant 1 was able to be computed by using Booth's (and with a slight extension Modified Booth's) algorithm. Now we are confronted with the task of computing invariant 2 by using a Booth-like algorithm. We'll walk you again through a bunch of mental steps to show how you can do it with slight alterations of Booth's algorithm and then move you on to alterations of Modified Booth's to compute Invariant 2.

Perhaps if we invert the rules and do nothing for the "x10" and "x01" bit strings and add for "x11" and subtract for "x00", which bears a strong resemblance to the complement of Booth's algorithm, we would come closer to the Invariant #2 sum. Let's observe what happens when we apply these modified rules to this sequence. We again will use our modified codewords here as "00xx..xx0".

Indices: 20 19 | 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 | 0

Codeword -4: 0 0 | 0 0 1 1 0 1 1 1 0 1 0 1 0 0 0 0 1 1 | 0

For codeword -4 given above, the rule will give: $2-4-5-6+12+13+16-18-19 = -9$.

Again a few more trials of your own would convince you that all the codewords listed here display the same property of the index sum calculated according to Booth's algorithm being equal to -9.

While defining the second invariant, we calculated the difference of Sum1 and Sum2 above, and found the result to be **1**. Now we have a result of **-9** in this part. How are these two values related? First, we should determine the effect the additional virtual bits have on our original invariant. Even though we are adding 3 virtual bits, we are actually only processing bits 1-18 and using the rightmost virtual bit for context. Since the 19th and 20th (the leftmost 2 extra) bits are always 0, the expected sum is now $1-19 = -18$ (we are processing the middle bit of each triplet, so the 20th bit is never truly accounted for). But that extra mental adjustment seems to yield a sum of -18 which does not match the observed sum of -9. The question arises as to where the extra 9 units difference between the expected and observed sum comes from. To understand that we need to take a somewhat more careful look as to what we are counting when we apply this Booth's-like calculation method for our Invariant 2. A moment's reflection will show that we actually end up skipping some of the operations. Let's look at several examples to realize which operations we end up skipping.

If we have a sequence such as 0110, where the rightmost 1 is at the n^{th} index, with our new algorithm, we skip the addition of index n ("10" is no-op) and the subtraction of index $(n+2)$. 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 $s-n+n+2$, or $s+2$, where s is 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. So, here in our case, we will always get a sum that is 9 (the number of 1s in our codewords) more than the invariant value of -18 which we just described. When one puts together the -18 that emanates from the leftmost extra bit inserted and the +9 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. Considering that a correct codeword produces an Invariant 1 of +9, we can simplify the result of Invariant 2 by leaving this number in the accumulator between the calculation of each invariant. For any correct codeword, at the end of the computation of Invariant 2 by the method we have just outlined will result in a value of 0.

Again, will any changes be needed to handle the special cases when we move to Modified Booth's? According to the rule defined above, for '101' and '010', the Booth's-like algorithm doesn't perform any operations, but because of our input logic to F, the next case will be 'x11' or 'x00', respectively, after the shift, thus triggering an operation. So if in the n^{th} cycle, a '101' is encountered, the following cycle's 'x11' results in an addition of $n+1$. Yet if you go back to our original discussion of Booth's, you will remember that neither alternating bit pairs (i. e. '01' and '10') will have performed an actual operation, implying no change to the running sum for invariant 2 for the two underlying pairs of '101.' Yet as we just discussed, the latter pair of the computation will result in the addition of $n+1$, necessitating a subtraction of $n+1$ when the first '01' pair of the '101' triplet in this context of Modified Booth's is encountered. Think about the '010' case, which can be similarly analyzed.¹

Now that we have discussed all the rules and operations performed for calculating both of the invariants, we can summarize them as in Table 2:

Table 2: Rules for calculating the two invariants

Invariant # 1 – Total number of 1s		Invariant # 2 – Zero sum indices	
000	No-op	000	Subtract the middle index
001	Add the middle index	001	No-op
010	Add one	010	Add (middle index + one)
011	No-op	011	Add the middle index
100	No-op	100	Subtract the middle index
101	Subtract one	101	Subtract (middle index + one)
110	Subtract the middle index	110	No-op
111	No-op	111	Add the middle index

¹ Interestingly, we can use our observations of the '010' and '101' cases to optimize even further. Upon encountering either case on the n^{th} cycle, we add or subtract $n+1$, respectively. In the next step, we have 'x00' or 'x11', and we perform the opposite operation, subtracting or adding our current n value. This value of n is incremented from the last step, so the addition of $n+1$ in our first step is canceled by the subtraction of n in the second step. This means any time we see the combinations '101' or '010', we can perform no operation on the current and next step. However, we still perform addition/subtraction on the steps where a 'x11' or 'x00' are encountered without a '010' or '101' in the previous step, so we cannot simply change our operation table. You can take advantage of this observation in your circuit if you want to further optimize things, but do not worry about it if you already have your plate full at this point.

Error Detection and Correction:

How can the rules described above be helpful in detecting errors? Recall that all codewords have two common invariants. For invariant #1, all codewords have a total of nine 1s. Therefore, we can detect any *odd* number of errors, as this situation will change the number of 1s in the codeword. For invariant #2, an error may change the sum of indices when the rules are applied. Obviously, if a single bit flips from 0 to 1 or from 1 to 0, adding the indices of the 1's and subtracting the indices of the 0's would no longer sum to 0. Furthermore, because the value at each index is distinct, any two bits of errors can also be detected by checking the *sum of indices*. It is possible to flip three bits of a codeword (for example, bits 1, 2, 3) and still get a sum of 0 for the second invariant. However, the first invariant can always detect this type of errors as it can detect any *odd* number of errors. Therefore, the two invariants together provide a capability of detecting *all errors up to 3*, implying that the code listed in Table 1 is **at least** a distance-4 code. *To show that the code is **exactly** distance-4, provide an example of a 4-bit error that isn't detected in either invariant.*

Perhaps the simplest form for detecting errors is the calculation of the invariants for a codeword, and comparing the actual result with the expected one for each invariant. On the other hand, if we are interested in correcting these errors, we need to perform an additional step. In this exercise we ask you to implement the correction of single bit errors. To achieve this goal, we thus need to understand how a single error changes the invariants.

An invariant can be used for error correction by using the invariant calculation to infer the position of the error based on how different the calculated value is from the expected value. Among the two invariants we have, which one can be used for error correction? Let's examine the *total number of 1's* first. If the j^{th} bit which should have been a 1 is flipped to a 0, the value of the first invariant (the number of ones) would be changed by -1. Likewise, if it was flipped from a 0 to a 1, the first invariant would be changed by +1. As you can see, this change does not preserve any information of the index j . Therefore we need to turn to the *second invariant* to see if there is any chance of identifying the faulty bit.

Invariant 2 error analysis is less obvious, so we start you off with the impact of an error on your computations in case you were using the Booth's-like algorithm, then graduate you off to a Modified Booth's algorithm. 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 for Invariant 2, 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 (error) 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, making the overall total $2j+1$ more than the original sum. So, a bit flip in position j alters the overall result by changing the computations for both the j and the $j+1$ index. To try another example, consider a codeword containing the sequence '110'. The contribution of these bits are the operations at positions j and $j+1$,

which are no-op and $+(j+1)$, respectively. Flipping the bit at j (the middle bit) changes the sequence to '100', producing the operations $-j$ and no-op. The difference in the sum is again $(2j+1)$. It is straightforward to see the intuition underlying the $2j+1$ deviation in the case of a single bit error if one for a moment strips away the mechanics of the process outlined above. Essentially, a flip from 0 to 1 in a particular bit position j will end up adding j units for the incorrectly encountered 1 while skipping the subtraction of j units that should have taken place if the 0 had been correctly encountered. This explanation accounts for a $2j$ deviation which gets augmented by 1 more unit, accounted for by the fact that Invariant 2 calculations according to our algorithm not only count the index sum but also add in the quantity of 1s encountered during our traversal, a quantity which has been incremented due to the incorrect '1.' Try other initial values for the $j+1$, j , and $j-1$ triplet and walk through the mechanics outlined in the beginning of this paragraph to verify that the change in value is indeed always $2j+1$.

Does the use of the Modified Booth's algorithm instead of basic Booth's examined heretofore change this fact? We know that the modified Booth's algorithm mimics Booth's steps except for the two special cases. Let's consider the example below, which again shows the effect of a single bit flip on the summation, but this time using our algorithm that will examine 3 bits at a time. Because each bit ends up being examined three times, our example has to be wide enough to consider the effect of the bit change on 3 operations. The bit that will be changed will once again be marked as ' j ':

$j+2$	$j+1$	j	$j-1$	$j-2$
1	1	0	0	0

By the rules given for calculating 'Invariant #2', the following operations will take place while scanning from right to left through the indices $(j-1)$ through $(j+1)$: $-(j-1)$, $-j$, no-op. This totals to $-(2j + 1)$. Now we flip the bit at position j to examine the difference it makes in the summation:

$j+2$	$j+1$	j	$j-1$	$j-2$
1	1	1	0	0

The operations that will be performed now become: $-(j-1)$, no-op, $+(j+1)$. This totals $+2$. The difference between them is $|(-2j + 1) - (+2)| = |-2j-1| = 2j+1$. While we have not illustrated an example for the special cases '101' or '010', you are encouraged to explore this and confirm that a bit change produces the same difference.

Once you are convinced that the calculation for the erroneous codeword will always differ by $2j+1$ from the original sum, we need to locate the bit position of error.

If the j^{th} bit is in error, the result of the index sum should deviate from 0 by $(2j+1)$ according to our calculations shown above. However for any correct codeword, in this calculation of Invariant 2, we expect to find 0 which is the result of the cancellation of the $+9$ left in the accumulator from Invariant 1 and the -9 from the calculation of Invariant 2. If one bit is in error, the Invariant 1 calculation will **also** be off by ± 1 in addition to the $\pm(2j+1)$ difference from Invariant 2. The result is that the cumulative Invariant 1 & 2 calculations deviate from 0 by $\pm(2j+2)$ for any 1-bit error at position j . We can use this final value to determine the index j and correct the error by flipping that bit.

Embedded Value:

Our final calculation is that of the value embedded in our codeword. **This value can be found by calculating the index sum of the lower 9 bits (in a 2's complement representation) and shifting right once.** Unfortunately, as we observed in the calculation of Invariant 2, calculating the index sum using a modified Booth's-like algorithm doesn't produce exactly the expected invariant value (expected value would be the positional index sum of 1, but as we discussed before the calculations consistently yield -9 instead). Let us review the cause of our Invariant 2 results:

1. The sum of the indices should be equal to 1.
2. The addition of three bits of padding (two left, one right) results in the extra observation of the 19th bit and an additional constant offset of -19.
3. The result is offset by the total number of 1s in the codeword (constant 9).
4. Summing 1-3 yields a result of $1-19+9 = -9$.

In a manner similar to the calculation of Invariant 2, to obtain the embedded value we must perform the index sum of the first 9 bits and adjust according to the above offsets. First, we recognize that performing the index sum for the first 9 bits will require the same left and right padding, and thus our observation of the 10th bit, always 0, will result in a constant offset of -10.

Similarly, we will observe an offset of the total number of 1s in the lower 9 bits of the codeword. For each code, however, the number of 1s is not constant. Fortunately for us, we already have a method of calculating the number of ones in a codeword in Invariant 1. Let us now consider how to use both invariants to find the embedded value.

From our Invariant 2 calculation of only the first 10 steps, with appropriate corrections to the padding, we have determined that we will have an offset of -10 plus the number of 1s in the lower 9 bits of the codeword. Since there is a constant offset of 10, we can simply implement a constant-10 addition. However, we must now account for the addition of the count of ones. Fortunately, we can obtain this value through re-use of our Invariant 1 calculations on the first 9 bits of the codeword. Perhaps if we were to first perform Invariant 1 calculations in order to obtain the number of ones in the lower half of the codeword, we could complement the result to compensate for the addition of the count of ones in our Invariant 2 calculations. We must also add 10 to compensate for the constant offset of -10, then perform Invariant 2 calculations starting up with $10-(\# \text{ ones})$ pre-loaded into the accumulator and thus find the embedded value.

In fact the correctness of this can be easily verified by looking more closely at Table 2. Complementing Invariant 1 results is equivalent to subtraction of that result, so what would the table look like if we subtracted each operation in the left column from the operation in the right column? Indeed we see that for a middle index of 0 or 1, we always simply subtract or add the index, respectively. This boils down to a basic Robertson's-like implementation of adding the index for each 1 seen and subtracting for each 0. Instead of changing your circuitry to match a Robertson's-like algorithm, you will be expected to reuse your modified Booth's-like Invariant calculations to find the embedded value.

1.2 Implementation

As you may have noticed, calculations for both invariants and the embedded value are very similar to the calculations performed by the modified Booth's multiplier you implemented in Lab 2. The difference though is that in the error detection/correction circuit, the value that

should be added/subtracted is sometimes a **constant 1** (for invariant #1), the **index of the bit** being analyzed (for invariants #1 and #2), or the **index of the bit+1** (for Invariant #2) rather than the value of the multiplicand. Actually there is no multiplicand operand in this lab; error detection/correction just operates on a single word. Another difference here is that no shift operation is to be performed for the intermediate sums. This is because for multiplication, the shift of the partial product is to account for the doubling of the bit weights in a positional number system. In this coding scheme, however, the bit weights keep on increasing by 1 (at least for the embedded value calculation) and not as a power of the radix as we proceed from the rightmost to the leftmost bit, thus eliminating the necessity for shift operations.

The strong similarity between the error detection/correction and multiplication operations suggests that an easy way to implement this lab assignment is to take the previous lab and make the appropriate modifications. For this reason, we will provide you a fully working Lab 2 (once all the late labs have trickled in), in case you do not have one of your own.

Circuit Overview:

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 *four-phase algorithm*. In the first phase, your circuit should compute the number of 1's and check whether invariant #1 holds or not. The second phase examines invariant #2 and in the process corrects any single bit error if necessary. The third phase calculates Invariant 1 for the rightmost 9 bits, and the fourth phase utilizes this result in combination with Invariant 2 for the same 9 bits to calculate the embedded value.

The Control Unit from Lab 2 is still useful here, as it will direct your circuit to perform 19 operations and shifts, but your circuit needs to go through the 19 operations twice for error detection and correction, and 10 of the 19 operations twice more for the embedded value calculation, with each run checking an 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 four times. It also has an output that lets your circuit know which phase it is in. You are asked to hook up the two controllers to implement the control unit.

The Super Controller:

Since the algorithm consists of four phases now, a new controller is needed in this lab. The basic operations within either phase are essentially the same, so rather than implementing four new controllers, 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. Obviously, the Controller should take this input as its **Begin** input. The SuperController also broadcasts a signal **Ph_bit0** to indicate the current phase information. In the first phase, this signal outputs a '1'.
- When the Controller raises the **End** signal at the end of each phase, this shows the SuperController that the current phase has been completed. The super-controller has a **PhaseEnd** output to read this information. So it initializes the next phase by setting its **BeginPhase** output first to 1 and then to 0 again. In the next phase, the

SuperController outputs an incremented Phase indicator as 001, 010, 011, and 100 for phases 1, 2, 3, and 4, respectively.

- For phases 3 and 4, we need to instruct the controller to end the current phase after the 10th shift/load cycle. The control logic implementation is left up to you.
- 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 the **End** signal at the end of the fourth phase, the SuperController understands that this is the final time that the **End** signal will have been generated. The error detection & correction is over now and the SuperController sets its **End** output to '1' to indicate that the full operation has been completed.

Implementation Guidelines:

Beyond the Booth's-like algorithm, each phase needs to have an additional data-path connected to the accumulator to perform some 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 numeral 9 and produce a one bit result (1 for correct, 0 for incorrect). You need to perform this comparison using only logic gates (no more than 17 gates would be optimal). The Phase 2 algorithm needs to use the result of the accumulator in order to perform its calculation, so be sure not to clear the accumulator between phases. However, during the Phase 2 calculations, the accumulator will once again change, so you will need a flip-flop to hold the value indicating whether an error has been detected in Phase 1. The phase output from the controller can be used to control the loading operation of this flip-flop. The flip-flop value can be continually overwritten as the accumulator changes until Phase 2 is reached; the final value will be the one displayed when Phase 1 is complete. The output of this flip-flop can be compared with the numeral 9 to determine whether there are odd-bit errors in the input codeword.

Phase 2:

In Phase 2, the circuit will examine our second invariant, which can be checked in a manner similar to the check of the first invariant: starting with the Invariant 1 result in the accumulator, perform the operations listed in the right part of Table 2 step by step. Our goal is to be able to correct single-bit errors, as well as detect up to 2-bit errors using this phase.

To detect errors, we only need to compare the accumulator with zero at the end of Phase 2 using only basic logic gates. But what about correcting single-bit errors? As we've seen from the explanation of the invariant above, if there is a single bit error at the j^{th} bit, the content of the **Accumulator** should be equal to $\pm (2j+2)$ at the end of the second phase. How do we use this value to correct the corrupted bit and output a corrected codeword? There are several steps to follow: first we need to compute the absolute value of the accumulator. Then we must divide it by two, and use that value to determine the index of the error. Fortunately, we will provide you a specialized 5-to-20 decoder that will receive the value thus obtained and output the index of that value. After determining the position of the faulty bit we need to perform the correction step by XORing the faulty bit with the single enabled pin of the decoder output. Instead of subtracting a constant value (to correct the constant ± 2), we can alter which outputs of this decoder correspond to which inputs of our "XOR" correction circuitry. A large XOR component has also been provided in the component library.

Phase 3:

For Phase 3, the circuit will repeat the Invariant 1 calculation for the first 9 bits. This requires

a total of 10 operations, and the left-padding must be accounted for in the 10th and 11th bits. After the 10th cycle, you must raise the signal to end the phase and begin Phase 4.

In a manner similar to the collaboration between Invariants 1 and 2, we must somehow use the result of Phase 3 in our Phase 4 calculations.

Phase 4:

For Phase 4, the circuit will calculate our embedded value, which can be checked in a manner similar to the check of the second invariant: we first initialize the **Accumulator** to a specific offset (10 - #1s in rightmost 9 bits), then perform the operations listed in the right part of Table 2 step by step. Our goal is to determine the embedded value by summing the indices.

One problem with Phase 4 is determining what to set the accumulator value to before beginning our calculations. This value is a constant 10 - (#1's in lower 9 bits). Luckily, the number of ones is stored in Invariant 1. It is left up to you to determine how you should implement this initialization for Phase 4, and to discuss the reason we use this value in your lab report.

In order to calculate the embedded value, we will need to either add or subtract each of the first 9 indices depending on whether there is a 1 or 0 at that index respectively. This value is then divided by 2 to get the codeword. For this last phase, you are to use the **corrected** codeword you determined from your phase 2 implementation.

Absolute Value:

There are several ways to take the absolute value of a numeral represented by a series of bit lines in LogicWorks. The method we recommend is to use an adder/subtractor to selectively compute $A - B$ or $A + B$, wherein A is always 0. If B is positive, we add, resulting in B , whereas if B is negative we subtract, resulting in $-B$, with either case thus providing a positive value. To do this variable operation we need to connect the A/S line to the sign of B . This should provide you a device that effectively takes a single input into B and outputs its absolute value. For the purpose of minimizing the hardware cost, you are encouraged to explore the possibility of reusing the adder/subtractor that is used for invariant calculation for the absolute value computation.

Divide by Two:

With binary numbers, multiplication and division with powers of 2 is very convenient, i.e. we add zeros to or cut off zeros from the right hand side. In this exercise, we need to divide by 2, which is essentially equivalent to cutting the LSB off. We can accomplish this by simply connecting the adder to the decoder (see correction part below) accordingly.

Correction:

We basically need to flip 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, the output of the adder used to compute the absolute value at this point (after our division by 2) would contain the value "000100" (accounting for constant 2 offset of $2j+2$, which causes a difference of 1 after our division by 2 in the previous step). We can use a decoder to transform this binary number into a form in which the fifth bit (one more bit offset to account for the fact that decoders are 0-indexed while our enumeration scheme is 1-indexed as we discuss in a bit more detail later on) of the decoder outputs a value 1 to the third bit of the XOR component, thus helping to correct bit #3 of the codeword by flipping it. Note that our **5-to-20 Decoder** is larger than our 18-bit codeword. This is not coincidental,

but a result of intelligent design. For example, if the result of our invariant calculation, after taking the absolute value and dividing by two, is 4, we can then “subtract” by 1 in the decoder by shifting the outputs by one (i.e. Q0..17 map to Q1..18). However, we also don’t have a 0th bit, as we start counting our indices from 1. Thus, we must shift one more bit over: Q0..17 of the register inputs will map to Q2..19 of the decoder.

How does this toggling-based correction work? By connecting each output pin of the decoder from Q2 to Q19 to one of the inputs of an XOR gate array, and each bit of the input word to the other pins, we can perform conditional bit flipping on each line.

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.

1.3 Components

Since the Lab 2 implementation is used as a basis for this lab, we inherit most components provided in the Lab 2 library. There are four new components you may need in this lab. We provide them to you in the Lab 3 library file.

1. Lab 2 components that continue to be used in this design

You can use the 20-bit **multiplier** register in your Lab 2 implementation to store the 18-bit codeword. The **multiplicand** register should be replaced with a counter, which will be used to represent the current index. The **accumulator** register will also still be used to store the partial sum of your error detection computation. A new register will also be used to store the value of Invariant 2 so the error correction circuitry has a persistent input. The **Flip Flop** containing your FF value and the logic driving it will not change from the Lab 2 implementation. Your glue logic for modified Booth’s will also be the same for the calculations in **Table 1**, but this logic must be augmented for **Table 2** calculations. The **Adder/Subtractor** you have used for Lab 2 is still needed to perform addition/subtraction during error detection/correction. The **controller** in Lab 2 will be used as the data-path controller in this design.

2. New components needed, either provided or available in LogicWorks library

- A **super-controller** to be used for controlling the four 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 controller that you have used in Lab 2. You need to hook up the super-controller to the Lab 2 controller to enable correct operation.
- A **5-to-20 decoder** to be used to determine the index corresponding to the given value. Only the bit corresponding to the 5-bit input will be true.
- **XOR-19**, a simple bitwise XOR of two 19-bit inputs. Can also be accomplished with individual gates, but much less cleanly.
- **MUX-2x19**, a two-input MUX that selects between two 19-bit values.

3. 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 Invariant 2 calculations, which should be stored in a register.
- The Counter, Controller, and Register **logic** to control dataflow.

Items to be displayed:

Your circuit should display the following results:

- The input word (on 5 hex keyboards);
- Two buzzer signals showing the error detection result in each phase (on binary probes);
- The content of a register holding the result of phase 1 calculations (on 5 hex displays);
- The content of a register holding the result of phase 2 calculations (on 5 hex displays);
- The corrected code word (on 5 hex displays) according to the second phase;
- The embedded value calculated in Phase 4 (on 1 binary probe and 1 hex display).
 - The probe signifies whether the codeword is negative (1) or positive (0), as the codeword should be displayed in a two's complement format.

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**.

2.0 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'.

Conversely, we must consider a list of implicants containing both abc' and $ab'c'$. These and other like pairs can be simplified (e.g. to the form ac') by searching for implicants with only the same variables, without regard for whether or not they are complemented, then check that they are complementary and replace both with the simplified form.

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.

2.1 The Absorption Theorem

To further 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:

$$\begin{aligned} \text{imp1} &= ab = 01\ 01\ 11 \\ \text{imp2} &= abc = 01\ 01\ 01 \end{aligned}$$

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 $S1 \cup S2 = S1$. A dual subset check exists as well, in which $S2$ is a subset of $S1$ if $S1 \cap S2 = S2$. 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:

$$\text{imp1} \mid \text{imp2} = 01\ 01\ 11 \mid 01\ 01\ 01 = 01\ 01\ 11 = \text{imp1}$$

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.

2.2 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:

$$\begin{aligned} ab + a'c + bc &= ab + a'c \\ ab + a'c + bc &\text{ (Given)} \\ ab + a'c + 1bc &\text{ (Identity)} \\ ab + a'c + (a + a')bc &\text{ (Complement)} \\ ab + a'c + abc + a'bc &\text{ (Distributivity)} \\ ab + a'c &\text{ (Absorption)} \end{aligned}$$

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:

$$01\ 01\ 11 \ \&\ 11\ 01\ 01 = 01\ 01\ 01 = \textit{templmp1}$$

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 *templmp2*:

$$10\ 11\ 01 \ \&\ 11\ 01\ 01 = 10\ 01\ 01 = \textit{templmp2}$$

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

Now that we have our decompositions, *templmp1* and *templmp2*, 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 *templmp1*, 01 01 01, and *templmp2*, 10 01 01:

$$01\ 01\ 01 \ \wedge\ 10\ 01\ 01 = 11\ 00\ 00$$

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 *templmp1* and *templmp2* and compare it to *imp3*. With the encoding, a union can be effected through a **bitwise-or**. So, if we take *templmp1*, encoded as 01 01 01, and union it with *templmp2*, 10 01 01, we obtain:

$$01\ 01\ 01 \ \mid\ 10\ 01\ 01 = 11\ 01\ 01$$

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 *templmp1*:

$$10\ 10\ 11\ 11\ 11 \ \&\ 11\ 10\ 01\ 01\ 01 = 10\ 10\ 01\ 01\ 01$$

Next, we take the intersection of *imp2* and *imp3* to obtain *templmp2*:

$$01\ 01\ 01\ 01\ 11 \ \&\ 11\ 10\ 01\ 01\ 01 = 01\ 00\ 01\ 01\ 01 = 00\ 00\ 00\ 00\ 00$$

Because *templmp2*'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.

Once we skip the bitwise-xor step, we can proceed to take the union of *templmp1* and *templmp2*; upon doing so, we obtain:

$$10\ 10\ 01\ 01\ 01\ 01 \mid 00\ 00\ 00\ 00\ 00\ 00 = 10\ 10\ 01\ 01\ 01\ 01$$

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):

templmp1 = imp1 & imp3

templmp2 = imp2 & imp3

if (templmp1 != 0 && templmp2 != 0):

if (templmp1 and templmp2 differ by more than one variable):

return false

return (templmp1 | templmp2) == 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:

$$imp1 = ab = 01\ 01\ 11\ 11$$

$$imp2 = a'c = 10\ 11\ 01\ 11$$

$$imp3 = bcd = 11\ 01\ 01\ 01$$

$$templmp1 = imp1 \& imp3 = 01\ 01\ 11\ 11 \& 11\ 01\ 01\ 01 = 01\ 01\ 01\ 01$$

$$templmp2 = imp2 \& imp3 = 10\ 11\ 01\ 11 \& 11\ 01\ 01\ 01 = 10\ 01\ 01\ 01$$

$$templmp1 \wedge templmp2 = 01\ 01\ 01\ 01 \wedge 10\ 01\ 01\ 01 = 11\ 00\ 00\ 00 = \text{differ by one variable}$$

$$templmp1 \mid templmp2 = 01\ 01\ 01\ 01 \mid 10\ 01\ 01\ 01 = 11\ 01\ 01\ 01 = imp3$$

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 **evaluate** methods. We ask that you do not alter any existing code, as we will be unit testing both classes to verify that the structure of the classes is 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 **01 01 01**, 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 0xFFFFFFFFFFFFFFF7, 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 (bitCountNewMSB == 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 **evaluate**. This function should take a minterm expression as input and determine what the result of evaluation of the Boolean Expression will be for that minterm. That is, if the minterm is covered by the Boolean Expression then the function should return true; otherwise, it should return false. This function can then be used to determine whether or not the minimized solution is equivalent to the original expression. In your report, please *include pseudocode for a **verify** function that takes as input the number of variables in the Boolean Expression (**numvars**), the original Boolean Expression (**originalExpression**), and the minimized Boolean Expression (**minimizedSolution**), that verifies that the minimizedSolution returns the same result as the originalExpression.*

Part 3: Boolean Simplification with Don't Cares

In this part of the lab, you will be extending the program you wrote in Part 2 that simplifies **Two-Level Sum-of-Products** boolean expressions by adding methods to perform **Reverse Absorption** and **Factoring** steps that take any *Don't Cares* in the original expression into account.

3.0 Don't Care Implicants

In a two-level boolean expression that is representative of some gate logic, we often find that there are certain combinations we never expect to see as input (or that we don't care

what their output is). For these cases, we simply consider the output a *Don't Care*. This can be used to our advantage in logic minimization, as we have the option to include or exclude the terms from the *Don't Care* expressions to simplify other implicants.

For example, if $ab'c'$ is one of our implicants, but abc' is a *Don't Care*, we can observe that simplifying $ab'c'$ to ac' causes the expression to be valid for abc' . This has an effect on the output, but we should never observe this input (or we just don't care what the circuit outputs, hence the term *Don't Care*) and thus the change allows us to simplify the implementation of our circuit.

We can also observe an expression where *Don't Care* implicants have no effect. Given the expression $a'b' + ab$, the *Don't Care* term abc has no impact on simplification because it is already included in the ab term. It can be easily verified that neither will its inclusion lead to a simpler Boolean expression formulation.

3.1 Reverse Absorption

However, there are some cases where a *Don't Care* may be included in terms that we do not want to keep, and it may be beneficial to temporarily remove them. For example, the expression $A'B' + B'C + AC$ with *Don't Cares* of $A'B'C'$ and ABC can be reduced to $B'C$, because $A'B'$ and AC each have a minterm that is already covered by $B'C$. Furthermore, the other term in $A'B'$ and AC is a *Don't Care* that does not need to be covered, allowing us to exclude them. To handle these kinds of situations, we can split up implicants containing a *Don't Care*, as long as we break even with the number of terms in our initial expression. With the example above, if we remove *Don't Cares* we end up with $A'B'C + B'C + ABC$. The hope is that in the subsequent Absorption step the new implicant covering a small number of minterms will be absorbed by an already existing implicant; otherwise, the factoring step will simply recombine the *Don't Care* and the new implicant if it is not absorbed.

3.2 Factoring

One simple simplification where *Don't Cares* are beneficial that you should not forget is factoring. If you have AB' in the expression and AB is a *Don't Care*, then AB' can be replaced by A . We can show that this is equivalent by factoring $AB' + AB$ to $A(B' + B)$, which simplifies to A .

3.3 Implementation Guidelines

For implementing the heuristics mentioned above involving *Don't Cares*, your functions will take as input a list of *Don't Cares* in a format similar to the implicant list from Part 2. You will need to implement **reverseAbsorption** and **factor** methods that handle *Don't Cares*, in addition to extending your **doSimplification** method to handle the *Don't Cares*. To extend your **doSimplification** method, we would like you to structure it so that it performs the following sequence of steps: Reverse Absorption, Absorption, Consensus, Factoring. Note that you do not need to modify your Absorption or Consensus methods to handle *Don't Cares*.

The **evaluate** method should not require any changes to correctly output the result of the minimized solution. However, the pseudocode you wrote for verifying that the minimized solution is equivalent to the original expression will require some minor modifications to account for *Don't Cares*. With this in mind, please include pseudocode for a **verify** function that takes as input the number of variables in the Boolean Expression (**numvars**), the original Boolean Expression (**originalExpression**), a list of don't cares (**dontCares**), and the minimized Boolean Expression (**minimizedSolution**), that verifies that the **minimizedSolution** returns the same result as the **originalExpression** for all input combinations that are not *Don't Cares*.

Deliverables:

- **Part 1 - LogicWorks Files:**

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.

- **Part 3:** 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:**

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 need to test the cases of 1-bit, 2-bit and 3-bit errors respectively, and report the behavior of each error detection/correction phase. Discuss the result of the following examples in your report:

- 0x287E0
- 0x0F45A
- 0x2F60B
- 0x11F8A
- 0x1C274

For part 2, your report should detail how you implemented the four required methods. For the **evaluate** method we ask that you *include pseudocode for a **verify** function that takes as input the number of variables in the Boolean Expression (**numvars**), the original Boolean Expression (**originalExpression**), and the minimized Boolean Expression (**minimizedSolution**), that verifies that the **minimizedSolution** returns the same result as the **originalExpression***. 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?

For part 3, your report should detail how you extended the 4 required methods to handle *Don't Cares*. For the **evaluate** method we ask that you *include pseudocode for a **verify** function that takes as input the number of variables in the Boolean Expression (**numvars**), the original Boolean Expression (**originalExpression**), a list of don't cares*

(**dontCares**), and the minimized Boolean Expression (**minimizedSolution**), that verifies that the *minimizedSolution* returns the same result as the *originalExpression*. Please test your program and ensure it works properly for the examples given in parts 2 and 3. Examples are provided inside the **main.java** class. Keep in mind that these are not exhaustive, and you should come up with additional tests to ensure your code is bug free.

Report Questions (answer these questions in your reports):

- 1) Since our codewords are distance-4, can we sometimes/always/never correct 2-bit errors? Justify your answer.
- 2) In Phase 4 why do we need to initialize the accumulator to a constant 10 – (#1s in lower half) before beginning our calculations? *Hint: Review the Invariant 2 theory section.*
- 3) In what cases will a don't care implicant cause the simplification of your expression through the consensus theorem? The absorption theorem?
- 4) Is there any case where a don't care causes the final expression to be less simplified than if optimization had been done without the don't care?