

Lab Assignment 3

Error Detection/Correction and Borrow Lookahead Subtractor

Due 11:59 PM November 5th, 2017

Introduction:

This lab assignment consists of two parts. In the first part, you are required to implement an error detection/correction circuit by modifying the Booth's multiplier circuit you have designed in LogicWorks 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, you will implement a 25-bit *borrow lookahead subtractor*, again in LogicWorks.

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. 22). 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 20 code-words in this coding scheme, represented by 10 bits each. This is indeed a rather inefficient scheme as only 5 bits would more than suffice to represent 20 different words but don't despair; the redundancy in these codewords will deliver certain error detection and correction capabilities.

Table 1: Error detection/correction codewords

-9	0 0 1 1 1 0 1 1 0 0	1	1 0 0 1 0 1 1 0 1 0
-8	0 0 1 1 1 1 0 0 1 0	2	1 0 0 1 1 0 0 1 1 0
-7	0 1 0 1 0 1 1 1 0 0	3	1 0 0 1 1 0 1 0 0 1
-6	0 1 0 1 1 0 1 0 1 0	4	1 0 1 0 0 1 0 1 1 0
-5	0 1 0 1 1 1 0 0 0 1	5	1 0 1 0 0 1 1 0 0 1
-4	0 1 1 0 0 1 1 0 1 0	6	1 0 1 0 1 0 0 1 0 1
-3	0 1 1 0 1 0 0 1 1 0	7	1 0 1 1 0 0 0 0 1 1
-2	0 1 1 0 1 0 1 0 0 1	8	1 1 0 0 0 0 1 1 1 0
-1	0 1 1 1 0 0 0 1 0 1	9	1 1 0 0 0 1 0 1 0 1
0	1 0 0 0 1 1 1 1 0 0	10	1 1 0 0 1 0 0 0 1 1

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 five 0's and five 1's. This invariant alone provides a single error detection capability, in a manner similar to the 2-out-of-5 code case. If you ponder this invariant a bit more, you will realize that (just as in the 2-out-of-5 code) it actually helps detect any *odd* number of errors.
2. **Sum of indices:** if we assign an index of 1 to 10 (represented by A in hexadecimal notation) 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 will get a total sum of 1.

Examples:

Indices: A 9 8 7 6 5 4 3 2 1	Indices: A 9 8 7 6 5 4 3 2 1
Codeword -9: 0 0 1 1 1 0 1 1 0 0	Codeword -8: 0 0 1 1 1 1 0 0 1 0
Indices for 1s: 3+4+6+7+8 =28 (Sum1)	Indices for 1s: 2+5+6+7+8 =28 (Sum1)
Indices for 0s: 1+2+5+9+A=27 (Sum2)	Indices for 0s: 1+3+4+9+A=27 (Sum2)
Sum1-Sum2 = 1	Sum1-Sum2=1

Calculating the Invariants:

Now that we have identified two invariants, we need algorithms to calculate them. You will be slightly altering your Booth's multiplier from Lab 2 in order to compute these invariants.

Invariant #1:

The number of 1's in a codeword can be easily calculated by the following algorithm: starting from the rightmost bit (the LSB), we add a 1 for each of the 1-bits, and subtract a 1 for each of the 0-bits. It is apparent that the resulting sum will be a 0 for every codeword given above, as each codeword consists of 5 0-bits and 5 1-bits.

Just as with Robertson's multiplication algorithm, this approach needs to perform a lot of additions for any consecutive sequence of '1's. On the other hand, you have learned from your Lab 2 assignment that Booth's multiplication algorithm enables us to replace a consecutive sequence of additions with only **two** operations. Can we calculate the number of 1's in a codeword using a method similar to Booth's algorithm? Let's examine several examples first.

Sequence: 0 1 0
Indices: +2 -1

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

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

Essentially, these examples indicate that for a sequence of consecutive 1's, subtracting the index of the rightmost '1' and adding the index of the '0' at the left-hand side of the sequence gives the number of 1's in that sequence. Therefore, the number of 1's in a codeword can be easily calculated by the following algorithm:

1s Count: starting from the rightmost bit, we examine 2 bits at a time, bit i and the bit immediately to its right. We perform a subtraction of the index i if $C_iC_{i-1} = '10'$, and an addition of the index i if $C_iC_{i-1} = '01'$. No operations are performed for $C_iC_{i-1} = '00'$ or $'11'$. This is illustrated in the following table:

C_iC_{i-1}	Operation
0 0	--
0 1	Add Index i
1 0	Subtract Index i
1 1	--

Table 1: Invariant #1 Operations

Obviously, this algorithm needs a pair of '01' and '10' to identify a sequence of consecutive 1's, implying that it can only correctly calculate the number of 1's in a codeword if the codeword begins with a '0' and ends with a '0'. Therefore, we need to modify our codewords here by inserting one '0' to the end (this one we have already included for Booth's multiplication) and another '0' to the beginning (this part is new, of course) of each codeword, i.e. the codeword becomes 0xxxxxxxxx0, where xxxxxxxxxxx is the original 10-bit codeword. Recall that our Booth's multiplier from Lab 2 worked with 11 bit numbers and our codewords here are 10 bits long, so the requisite space is already available for the new '0' needed at the beginning. (Note that this modification does not change the number of 1's.)

Indices:	B		A	9	8	7	6	5	4	3	2	1		0
Codeword 0:	0		1	0	0	1	1	1	1	0	0			0

Now, if we perform this **1s Count** algorithm on Codeword 0, we will get the number of 1's = $-1+3-6+7-9+B = 5$.

Invariant #2:

The straightforward approach for computing the second invariant is to look at each bit of the codeword and add (or subtract) its index to (or from) the partial sum if that bit is a '1' (or '0'). This approach requires an addition or subtraction to be performed for each bit of the codeword. The operation to be performed on the partial sum for each bit can be shown in the following table:

Value of C_i	Operation
0	Subtract Index i
1	Add Index i

Table 2: Invariant #2 Operations

Normally, when we compute invariant #2 in this manner, we get a final result of '1'. However, note that in order to do invariant #1 properly, we added an extra leading '0' to the codeword (which has an index of 'B', or 11 in decimal). According to the previous table, since the index 'B' would be subtracted from our final product, we actually now end up with $1 - B = -A$, or in decimal $1 - 11 = -10$, as the correct result of calculating invariant #2 on the extended codeword. In other words, a machine operating on the principles of Table 2 on the extended codewords with an additional 0 on the left should always yield the value of $-A$ (-10 in decimal).

The question is how to utilize the Booth's hardware already in place to perform the invariant #2 calculation according to Table 2. Booth's multiplication does not examine one bit of the multiplier at a time, but rather it examines 2 bits at a time. We can easily expand the above table to look at 2 bits instead:

$C_i C_{i-1}$	Operation
0 0	Subtract Index i
0 1	Subtract Index i
1 0	Add Index i
1 1	Add Index i

Table 3: Invariant #2 Operations Expanded

In a sense, we are still subtracting or adding based upon the value of the bit C_i . However, we may be able to utilize the second bit, C_{i-1} . If you look back at Table 1, you'll notice that additions and subtractions were performed for $C_i C_{i-1} = 01$ and 10 , but the operation was the *opposite* of what we would want to be done for Table 3. However, if we *negate* the result we got when calculating invariant #1, the new result is the same as if we completed rows $C_i C_{i-1} = 01$ and 10 of Table 3! We already know that the result of invariant #1 should be '5'. We can avoid having to redo the operations for $C_i C_{i-1} = 01$ and 10 during the computation of invariant #2 because we already know the result of those two operations (it is the *negation* of the result of invariant #1: '-5'). Our updated table of operations for this new, "degraded" algorithm is shown below:

$C_i C_{i-1}$	Operation
0 0	Subtract Index i
0 1	--
1 0	--
1 1	Add Index i

Table 4: "Degraded" Invariant #2 Operations

So, what should our final result be after performing the degraded invariant #2 algorithm? As previously mentioned, the result of the original invariant #2 algorithm would give us '1', but after taking into account the extra leading 0, we will be left with '-A'. However, the degraded invariant #2 algorithm does not perform operations for $C_i C_{i-1} = 01$ and 10 , so the result will be *missing* the value from those operations. As mentioned before, that missing value is the *negated* result of invariant #1, '-5'. Thus, the degraded invariant #2 algorithm will be *missing* '-5' from what the original invariant #2 algorithm would compute. That is, the result of our degraded invariant #2 algorithm is $-10 - (-5) = -5$. Let's try a few examples to build confidence that this is correct:

- For the codeword -8, the degraded algorithm produces $-1 - 4 + 6 + 7 + 8 - A - B = -5$.
- For the codeword -9, the degraded algorithm produces $-1 - 2 + 4 + 7 + 8 - A - B = -5$.
- For the codeword 0, the degraded algorithm produces $-1 - 2 + 4 + 5 + 6 - 8 - 9 = -5$.
- For the codeword -1, the degraded algorithm produces $-5 - 6 + 8 + 9 - B = -5$.

Our analysis that the degraded invariant #2 algorithm will result in '-5' are thus confirmed.

There is one more optimization we can incorporate. We know that the result of our invariant #1 algorithm should be '5', and the result of our degraded invariant #2 algorithm is '-5'. If we left the result of invariant #1 in the accumulator before calculating our degraded invariant #2, then there will already be a value of '5' in the partial product before we begin invariant #2. After completing invariant #2, there will be an additional net value of '-5' in the accumulator. Thus, after computing both invariants in succession, our final result left in the accumulator will be $5 + (-5) = 0$! Thus, we can simply check if the final result is 0 after the degraded invariant #2 algorithm is completed to detect if the codeword is correct.

To summarize, the operations performed for checking both invariants are listed as follows:

Table 5

Invariant #1 – Total number of 1's	
$C_i C_{i-1} = 00$	No-op
01	Add i (index of C_i)
10	Subtract i (index of C_i)
11	No-op

Table 6

Invariant #2 – Sum of indices	
$C_i C_{i-1} = 00$	Subtract i (index of C_i)
01	No-op
10	No-op
11	Add i (index of C_i)

Tables 5 and 6: Rules for calculating the two invariants

Error Detection and Correction:

How can the two rules above help detect and correct errors? For invariant #1, all codewords have a total of five '1's. Therefore, we can detect any odd number of errors, as this situation will change the number of 1s in the codeword. Now, let's see what the original invariant #2 contributes in terms of error detection. 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 1. Furthermore, because the indices of any two bits are 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 1 for the second invariant. However, the first invariant can always detect this type of error 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. You can verify that the code is indeed **exactly** a distance-4 code by finding two codewords in the table that differ exactly by 4 bits.

In order to detect 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. To correct these errors, we need to perform an additional step. Basically, a minimum *Hamming distance* of 4 between any two codewords provides the ability to correct any single-bit error. To achieve this goal, we need to understand how a single error changes the invariants.

An invariant can be used for error correction if the position of the bit that has been corrupted is preserved in the calculation of that invariant. 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 would be changed to 4. 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.

If the j^{th} bit is flipped from a 1 to a 0, the index of that bit, j , is subtracted although it should have been added to the *sum of indices*, thus resulting in the value of the original *second invariant* being **$2j$ less** than it should have been. Similarly, if the j^{th} bit is flipped from a 0 to a 1, the value of the second invariant would be **$2j$ more** than it should have been. In either case, the sum of indices will therefore deviate from 1 by **twice** as much as the index value. This implies that if we subtract 1 from the resulting sum, compute the absolute value, and then divide it by 2, we would exactly get the index (position) of the error bit. However, because we are using a degraded version of the second invariant algorithm, as well as not clearing the accumulator between the first and second invariant computations, there are some ramifications to the error correction.

In the description of our degraded invariant #2 algorithm, we have confirmed that for any codeword, the result of the algorithm produces a result of **-5** (which is cancelled by the 5 that is already in the accumulator from invariant #1, leaving a total result of **0**). The next issue to contemplate is, in the case of a single bit error, what would the result of our degraded algorithm be? We have shown that in the case of a single bit flip from 1 to 0 at the j^{th} bit, the *total number of 1's* would decrease by 1 (resulting in a total of **4**), while the *sum of indices* would decrease by

2j. However, as described earlier, our degraded invariant #2 algorithm is *missing the negation of the number of 1's*. Therefore, instead of missing a value of -5, we are instead **missing a value of -4**. In other words, the result of our degraded algorithm (without considering the value in the accumulator after invariant #1) would be $(1 - 2j) - B - (-4) = -6 - 2j$ (1 is the original invariant #2 result, and -B is due to the extra leading 0, as described in the details of the degraded invariant #2 algorithm). But, since we are not clearing the accumulator, we are also adding the result of *the number of 1's* to our sum, which in this case will be 4 (instead of the usual 5). So, *our final result after computing both invariants, without clearing the accumulator, will be $(1 - 2j) - B - (-4) + 4 = -2 - 2j = -2(j+1)$* , instead of the usual value of 0 for a correct codeword.

Similarly, if a bit flips from 0 to 1, the *total number of 1's* would increase from 5 to 6, and the sum of indices would increase by **2j**. Our final result would be $(1 + 2j) - B - (-6) + 6 = 2(j+1)$. Thus, if there is a single bit flip at index j, our final result will be $\pm 2(j+1)$. We can find the index j of the erroneous bit by finding the absolute value of the result, dividing by 2, and then subtracting 1. Let's observe this with an example by taking codeword 0 and flipping its most significant bit from 0 to 1.

Indices:	B A 9 8 7 6 5 4 3 2 1 0
Codeword -9 with single bit error:	0 1 0 1 1 1 0 1 1 0 0 0
Number of 1's calculates:	$-3 + 5 - 6 + 9 - A + B = 6$
Degraded invariant #2 calculates:	$-1 - 2 + 4 + 7 + 8 = 16$
Value in accumulator:	$6 + 16 = 22$
Position of the error bit:	$ 22 / 2 - 1 = A$ (10 in decimal)

1.2 Implementation

As you may have noticed, calculations for both invariants are very similar to the calculations performed by the Booth's multiplier you implemented in Lab 2. For both Booth's multiplication and the two algorithms, we are examining two bits at a time and performing an addition, subtraction or no-op accordingly. One difference is that in the error detection/correction circuit, the value that should be added/subtracted is the **index of the bit** being analyzed rather than the value of the multiplicand. Thus, the register holding the multiplicand should be replaced with a counter that keeps track of the index of the bit being analyzed. Another difference here is that *no shift operation is to be performed for the partial 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 (instead of doubling) as we proceed from the rightmost to the leftmost bit, thus eliminating the necessity of shift operations. Finally, for each codeword, the circuit will evaluate twice, once for each invariant. The controller from Lab 2 will provide control signals to evaluate once. So, we will have an additional **super-controller** that will be connected to the original controller, signalling the original controller to run twice. The super-controller is described in greater detail later in section **1.3: Components**.

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 (after the late deadline on Wednesday night, 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 are satisfied. You can implement the various checks through a *two-phase algorithm* that subsequently checks the two invariants in each phase. After checking the second invariant, your circuit should furthermore perform error correction using the content of the accumulator **A** (used in the implementation of Booth's multiplication) at the end of the second phase.

In the first phase, your circuit should compute the number of 1's by performing the operations listed in Table 5. At the end of this phase, the content of the accumulator **A** should equal 5 for a correct codeword. At that time, the **Error Detection** logic should compare the content of **A** with 5 and output a 1 if they do not match.

Next, the second phase will begin. As described in the detailing of the error detection/correction algorithms, you should **not** clear the content of **A** before proceeding with computing invariant #2 in the second phase. As we have discovered, we will end up getting a 0 for any correct codeword or $\pm 2(j+1)$ for any codeword with a single bit error at the j^{th} bit. Therefore, at the end of the second phase, the **Error Detection** logic should compare the content of **A** with 0 and output a 1 if they do not equal. Furthermore, in case **A** is nonzero, the **Error Correction** logic should compute the index of the corrupted bit by evaluating the absolute value of **A**, dividing the value by 2, and then subtracting by 1.

Therefore, the complete error detection/correction algorithm can be summarized as follows:

Phase 1: Calculate the number of 1's

- Clear accumulator **A**, load the codeword to be inspected into the **multiplier**, initialize the **counter** to 1, reset flip-flop **F** to 0.
- Repeat 11 times the following two steps:
 - If the rightmost bit of the **multiplier** is 1 and **F**=0, **A**=**A**-**counter**;
If the rightmost bit of the **multiplier** is 0 and **F**=1, **A**=**A**+**counter**;
Else, the result of the **adder/subtractor** is discarded (not saved by the accumulator **A**)
 - Shift the **multiplier** to the right (the rightmost bit of multiplier is shifted into **F**), increment the **counter** by 1.
- Perform error **detection** on **A**.

Phase 2: Calculate the sum of indices

- Load the codeword to be inspected into the **multiplier**, initialize the **counter** to 1, reset flip-flop **F** to 0 (No change to the content of the accumulator **A**).
- Repeat 11 times the following two steps:
 - If the rightmost bit of the **multiplier** is 0 and **F**=0, **A**=**A**-**counter**;
If the rightmost bit of the **multiplier** is 1 and **F**=1, **A**=**A**+**counter**.
Else, the result of the **adder/subtractor** is discarded (not saved by the accumulator **A**)
 - Shift the **multiplier** to the right (the rightmost bit of multiplier is shifted into **F**), increment the **counter** by 1.
- Perform error **detection** on **A**, identify the corrupted bit and display the "corrected" word.

Note that just as with Lab 2, the result of the **adder/subtractor** will not be saved in the accumulator **A** on every add/subtract cycle. More specifically, even though it was computed, the result will not be saved when no operation should be performed as indicated in tables 5 and 6 and in the preceding pseudocode.

Implementation Guidelines:

Now let us summarize the preceding information and try to provide you a somewhat sharper picture regarding the implementation of this design. Your starting point is the multiplier circuit in Lab 2. Your modification of the circuit could be based on the following guidance.

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

You can use the 11-bit **multiplier** register in your Lab 2 implementation to store the 10-bit codeword and the extra 0 at the beginning. The extra 0 at the end can be stored in the flip-flop **F**. The **accumulator** register will still be used to store the partial sum of your error detection computation. The **Adder/Subtractor** you have developed for Lab 2 is still needed to perform addition/subtraction during error detection/correction. The **controller** in Lab 2 will be used as the datapath-controller in this design.

2. Lab2 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 11, a *4-bit counter* provided in the LogicWorks standard Library would suffice.

3. New components that we will provide to you

- A **16-bit decoder** to be used in the design of error correction logic.
- A **super-controller** to be used for controlling the two-phase operation by indicating the start and the end of each phase. Within each phase, the operation of the circuit is controlled by the datapath controller (which is identical to the controller used in Lab 2). You need to hook up the two controllers to enable their correct operation.

4. New components that you need to design and insert into the circuit

- You are asked to design the **Error Detection** logic which should output a 1 if an error is detected in either the first or the second phase.
- You also need to design the **Error Correction** logic which corrects the input word based on the value in **A** at the end of the second phase.

1.3 Components

Control Units:

The controller unit that you will use for the error detection circuit is designed in a hierarchical manner. It consists of a **super-controller** and the **datapath-controller**; the latter one is the same control unit that you have used in your Lab 2 assignment. You are asked to hook up the two controllers to implement the control unit. We will provide you with the **super-controller**, which is responsible for controlling which phase is executed by the **datapath-controller** by providing appropriate signals to the **datapath-controller**:

- The super-controller 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 datapath-controller should take this input as its **Begin** input. In this phase, the SuperController broadcasts the **PhaseOne** output as 1.
- The super-controller has a **PhaseEnd** input to monitor the end of each phase indicated by the datapath-controller. When the datapath-controller raises the **End** signal for the first time, the super-controller knows that the first phase has been completed. So it initializes the second phase by setting its **BeginPhase** output first to 1 and then to 0 again. Meanwhile, the super-controller broadcasts the **PhaseOne** output as 0, indicating that we are currently in the second phase.

- When the datapath-controller raises the **End** signal at the end of the second phase, the super-controller understands the operation to be over. Therefore, it sets its **End** output to 1.

Within each phase, the operation of the circuit is actually controlled by the **datapath-controller**. Fundamentally, the task you need to accomplish here is to control which operation should be performed by the adder/subtractor (through driving the **A/S** signal), and whether the result of the adder/subtractor should be loaded into the accumulator (through driving the **Mult** input of the controller). You can refer to the previous lab write-up to remember the input and output signals of the **datapath-controller**. As you can observe from Table 2 in this write-up, in different phases the operations performed by the adder/subtractor and the conditions for loading the adder/subtractor output into **A** are distinct. In order to drive these signals appropriately, you need to consider the least significant bit of the **multiplier**, the value in flip-flop **F**, and the **PhaseOne** output of the super-controller. In your implementation, you should drive these signals using basic gates (no more than 3 gates are needed).

Error Detection/Correction Components:

Error Detection:

You are asked to design the **Error Detection** logic which produces a one bit result (0 for correct, 1 for incorrect) if an error is detected in either the first or the second phase. For *phase1*, you need to compare the accumulator with the constant number 5 using only basic logic gates (no more than 5 gates would be optimal). As the content of **A** will change in the second phase, this comparison result should be latched in a flip-flop **Err** at the end of *phase1*. For *phase2*, you need to compare the accumulator with the constant number 0 using again only basic logic gates. If there is no error reported in *phase1* but an error is reported in *phase2*, the content of the flip-flop **Err** should be updated at the end of *phase2*. Finally, a buzzer, **B**, controlled by the output of **Err**, should be used to signal the detection of error by buzzing.

Error Correction:

The error correction logic tries to correct the input word based on the content of **A** at the end of *phase2*. If our input codeword is correct, the content of **A** at the end of phase 2 is **0**. Otherwise, if there is a single bit error at the j^{th} bit, the content of **A** should be equal to $2(j+1)$ or $-2(j+1)$. If we evaluate its absolute value, divide the value by 2 and then decrement the value by 1, we would obtain the index of the corrupted bit of the input. At that point, we can correct the input code-word by flipping the corrupted bit and displaying the correct result. You are asked to implement these functions using an absolute value evaluator, a decoder and some basic logic gates.

There are several ways to take the absolute value of a binary number. The method we recommend is to use an adder/subtractor to selectively compute $A - B$ or $A + B$, wherein **A** is always 0. The A/S line that determines the operation is to be controlled by the sign of **B**. In this way, if **B** is positive, the result of the adder/subtractor will be **B**, whereas if **B** is negative, the result of the adder/subtractor will be $-B$.

Given the index of the corrupted bit of the input, we basically need to flip the erroneous bit and display the correct result. This process consists of two parts. First, we need to use a **decoder** to transform a binary index i into a form in which only the i^{th} output line is active. Because our codeword is 10 bits long, we need a decoder that has at least 10 output pins; as typically decoders come in sizes that are powers of 2, we provide you a *16-bit* decoder for this lab.

The second part of this process involves correcting the input codeword. We need to conditionally flip each bit of the input word by XORing it with the corresponding output of the **decoder**. For a bit that does not have an error, the decoder output pin corresponding to that bit of the input word should be 0. XORing 0 with that bit would not affect its original value. If the bit

was corrupted, on the other hand, the decoder will output a 1 for that line, and whatever value came from the input will be flipped. This error correction process is shown in Figure 1 below:

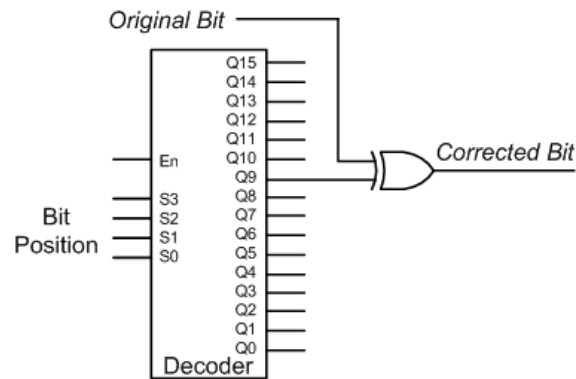


Figure 1: Using a Decoder to Flip a Single Bit

We have discussed the way to compute the absolute value of the content of **A** and the way to correct the codeword given the index of the corrupted bit. The question left open is how to divide the absolute value of **A** by 2 and then decrement it by 1 to get the index of the corrupted bit. While a straightforward implementation of this somewhat complex operation would need a divider and a subtractor, most hardware designers (like you, for example) prefer to simplify their design by using a clever way of connecting signals. As you may have noticed, dividing an even binary number by 2 is essentially equivalent to cutting the least significant bit off. In fact, both the division and the subtraction can be implemented by a clever way of hooking up the absolute value evaluator, the decoder, and the XOR arrays. Neither a divider, an extra adder, nor a shifter is needed.

Items to be Displayed:

Your circuit should display the following results:

- The input word (on 3 hex keyboards);
- A buzzer signal showing the result of error detection (on 1 binary probe);
- The content of **A** after the second phase (on 3 hex displays);
- The corrected code word (on 3 hex displays).

Part 2: Borrow Look-Ahead Subtractor

One important concept in digital logic design is the circuit's latency. Latency is the time the circuit takes to provide an output when a new set of inputs are presented. In a combinational circuit, this is simply the propagation delay of the longest path from input to output. In a sequential circuit, which we will see later in the course, it is the number of clock cycles the circuit takes to produce the result multiplied by the clock period.

In this part of the lab, you will be designing a borrow look-ahead subtractor. To start, we will analyze the concept of carry look-ahead. We will then graduate to borrow look-ahead using similar concepts.

The interesting part of a binary carry look-ahead adder is that it can in some sense precompute the carries of each bit, thus drastically speeding up the overall adder through enabling a number of operations to happen concurrently. You need to think about how to design a *borrow look-ahead* logic for binary numbers based on the ideas you have learned from carry look-ahead adders.

Carry look-ahead binary adder

You have learned from your textbook that a single bit **full adder** (FA) takes as inputs two operands x_i , y_i and the carry c_i , and produces as outputs the sum s_i and carry c_{i+1} . An n -bit binary adder can be implemented as a serial connection of FAs, arranged such that the output carry of each FA serves as the input carry for the next-higher significant FA. This type of adder is denoted as **ripple-carry adder**, since any change in the least significant bits c_0 , x_0 , or y_0 has to ripple through all the FAs.

The purpose of designing a carry look-ahead (CLA) adder is to reduce the delay of the carry chain in a ripple adder as this is the critical path and in fact still remains the critical path even in a CLA adder. This technique takes advantage of the fact that the computation for all the carry values in the chain can be precomputed in parallel and be available in a much shorter time than the ripple method. Let's take as an example a 4-bit adder that computes $x_3x_2x_1x_0 + y_3y_2y_1y_0 + c_0$. We first define two functions as follows: (a) the carry-generate function $g_i = x_i y_i$, and (b) the carry-propagate function $p_i = x_i \oplus y_i$. Using these two functions, the four carries $c_1 \dots c_4$ can be expressed as follows:

$$c_1 = g_0 + p_0 c_0 \quad (1)$$

$$c_2 = g_1 + p_1 c_1 = g_1 + p_1(g_0 + p_0 c_0) = g_0 + p_1 g_0 + p_1 p_0 c_0 \quad (2)$$

$$c_3 = g_2 + p_2 c_2 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0 \quad (3)$$

$$c_4 = g_3 + p_3 c_3 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0 \quad (4)$$

These four equations show that the four carries $c_1 \dots c_4$ can be computed directly and in parallel from input bits $x_3x_2x_1x_0$, $y_3y_2y_1y_0$, and carry-in c_0 without incurring a ripple. In other words, the carry value for each bit can be generated essentially simultaneously, which in turn allows the addition of each bit to be performed in parallel, thus drastically reducing the latency of the adder circuit! Figure 2 shows how these CLA generators can be daisy chained together to make larger adders but in doing so the circuit still suffers a similar, yet attenuated, delay as the ripple adder. The carry delay of c_4 which is needed in order to enable the next CLA generator to begin computation is still the critical path and the use of larger fan-in gates causes the delay of this path (c_0 to c_4) to be greater than necessary. In order to speed up the generation of c_4 , the equation for c_4 can be transformed to:

$$c_4 = G_{0,3} + P_{0,3} c_0 \quad (5)$$

where $G_{0,3}$ and $P_{0,3}$ are defined in a form that can be precomputed as:

$$G_{0,3} = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 \quad (6)$$

$$P_{0,3} = p_3 p_2 p_1 p_0 \quad (7)$$

This transformation speeds up the generation of c_4 by reducing the maximum fan-in requirements from 5-inputs down to 4-inputs for a 4-bit CLA. In the case of your implementation, a similar transformation will obviate the need for 6-input gates, enabling you to implement your design with gates of up to 5 inputs.

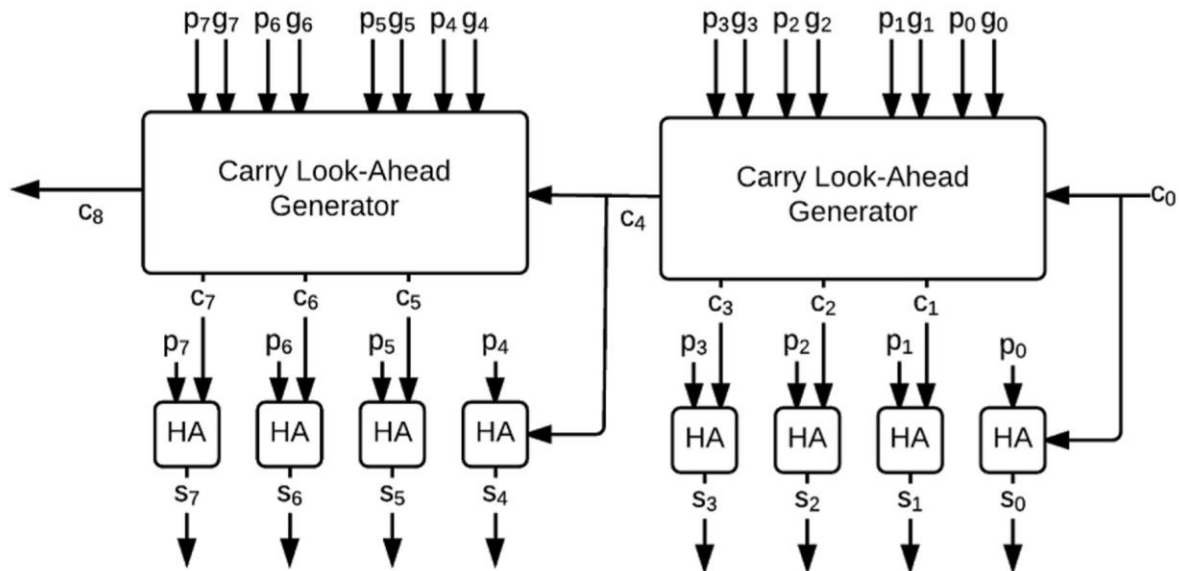


Figure 2: Example 8-bit single level CLA Adder

In general, a carry look-ahead adder circuit consists of three parts:

1. Carry-generate functions and carry-propagate functions for each bit
 - a. The carry-generate function $g_i = x_i y_i$ determines whether the two input values x_i and y_i will generate a carry to a higher bit.
 - b. The carry-propagate function $p_i = x_i \oplus y_i$ determines whether the two input values can propagate a carry coming from a lower bit to a higher bit.
2. A CLA Generator which takes as inputs the carry-generate and carry-propagate functions of each bit, and generates the carry value for each bit in parallel using Equations (1) - (4).
3. Single-bit Half Adders (HA) for each bit, taking as inputs the carry-generate function p_i and the carry value c_i (provided by the CLA Generator), producing the sum $s_i = p_i \oplus c_i$.

The carry precomputation ideas enable us to derive the Boolean functions that can generate each carry bit essentially in parallel in a 25-bit CLA adder. However, because of fan-in and fanout limitations, it is practically infeasible to directly implement a 25-bit CLA generator. The simpler method just connects five 5-bit CLA adders in serial so that the carry-out of each CLA adder serves as the carry-in of the next-higher significant CLA adder. This 25-bit adder is denoted as a single-level CLA adder since it is constructed in such a way that carries within each 5-bit adder are computed by a 5-bit CLA generator based on the values of g 's and p 's, while the 5-bit carries $c_5, c_{10}, c_{15}, c_{20}$, and c_{25} ripple through the 5-bit slices

Borrow look-ahead binary subtractor

A borrow look-ahead subtractor shares similar design principles with a carry look-ahead adder. In order to precompute the borrow information, for every bit, a **borrow-generate** function g_i and a **borrow-propagate** function p_i are needed. The borrow-generate function g_i takes as its inputs the i^{th} bit of the minuend x , denoted by x_i , and the i^{th} bit of the subtrahend y , denoted by y_i . When this function is 1, we definitely need to borrow from the higher bit. The borrow-propagate function p_i also takes x_i and y_i as its inputs, and lets us know whether we should consult the borrow-in bit from the lower bit. When this function is 1, the borrow-in bit will be directly propagated to the borrow-out. The borrow-generate and borrow-propagate functions of each bit are fed into a borrow look-ahead logic (BLA) which generates the borrow value for each bit of

the subtraction in parallel. Finally, for each bit, the propagate signal can be combined with the borrow value through an appropriate gate to compute the final difference.

The borrow-generate function is defined as $g_i = x_i' y_i$. This is because, when x is 0 and y is 1, we will always borrow, regardless of the borrow-in. The borrow-propagate function is defined as $p_i = x_i' \oplus y_i = x \text{ XNOR } y$, because we will otherwise propagate the borrow-in if x and y are both 0, or they are both 1. The borrow functions are defined in exactly the same way as the carry functions:

$$b_1 = g_0 + p_0 b_0 \quad (8)$$

$$b_2 = g_1 + p_1 b_1 = g_1 + p_1(g_0 + p_0 b_0) = g_1 + p_1 g_0 + p_1 p_0 b_0 \quad (9)$$

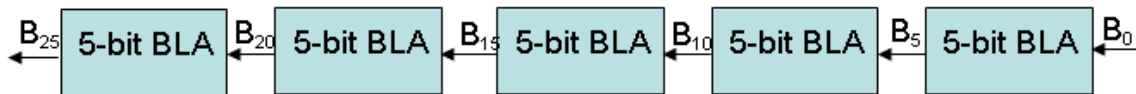
$$b_3 = g_2 + p_2 b_2 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 b_0 \quad (10)$$

The b_4 and b_5 bits can be computed in a similar manner as above.

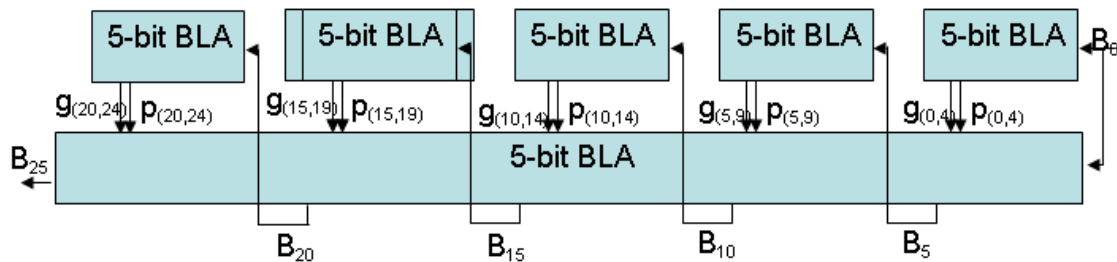
Looking at Figure 2, the carry-propagate signal is combined with the carry value produced by the CLA generator through a half adder to compute the final sum. Similarly, the borrow-propagate signal must be combined with the borrow value to compute the final difference for each bit. You will only need a single gate for this, which is left for you to figure out.

Putting it all together:

Similarly to the CLA adder described above, it is practically infeasible to directly implement a 25-bit BLA generator. Therefore, in this part of the lab, we will use two other methods to implement a 25-bit BLA binary subtractor. First, we will connect five 5-bit BLA binary subtractors in serial such that the borrow-out of each BLA subtractor serves as the borrow-in of the next-higher significant BLA subtractor. This 25-bit binary subtractor is denoted as a **single-level BLA subtractor** since it is constructed in such a way that borrows within each 5-bit subtractor are computed by a 5-bit BLA generator based on the values of g 's and p 's, while the 5-bit borrows $b_5, b_{10}, b_{15}, b_{20}$, and b_{25} ripple through the 5-bit slices. The advantage of this method is that it requires no extra hardware besides the five 5-bit BLA subtractors. However, each 5-bit BLA subtractor still needs to wait for the next-lower significant BLA subtractor to generate the borrow, thus limiting the speed of this subtractor. A high-level schematic of this is shown below:



In contrast, the somewhat more complicated but faster method, the **two-level BLA subtractor**, uses an extra BLA generator in addition to the five 5-bit BLA subtractors in order to speed up the generation of 5-bit borrows $b_5, b_{10}, b_{15}, b_{20}$, and b_{25} . A moment's reflection should show that the two-level BLA generator can be constructed in a manner similar to that shown below:



Obviously, the two designs of 25-bit BLA binary subtractor display distinct hardware complexity and timing characteristics. In this part of the lab, you are asked to implement **both** designs and compare their timing characteristics.

Implementation Guidelines

Now let us summarize the preceding information and try to provide you a somewhat sharper picture regarding the implementation of this design. Your implementation of the 25-bit borrow look-ahead subtractor should be composed of the following modules:

- **1 single-bit borrow-generate and borrow-propagate component**
- **1 BLA generator:** It should take as input the results of 5 of the 1-bit generate and propagate components, and output the borrow for each single-bit subtraction. It also outputs the combined generate and propagate signals for those 5 bits.
- **1 5-bit BLA subtractor:** Combine 5 of the single-bit components, the BLA generator, and the appropriate gates to compute the difference for each bit, to produce a complete 5-bit BLA subtractor.
- **2 25-bit BLA subtractors:** Both 25-bit subtractors will combine 5 of the 5-bit subtractors to create a complete 25-bit subtractor. However, one 25-bit subtractor will link the borrow out of one 5-bit subtractor to the next in a ripple-borrow fashion (single-level BLA subtractor), while the other will employ an additional BLA generator to compute the borrows between the 5 first-level BLA subtractors (two-level BLA subtractor).

We have provided you with a 25-bit borrow subtractor that uses the ripple-borrow mechanism to use as reference (and to compare with your timing results from the look-ahead implementations). You are asked to implement both the single-level and the two-level 25-bit BLA binary subtractors. Once you have finished both implementations, you are asked to compare their performance. In order to obtain the correct timings, you must manually specify the delays for each gate. We would like you to emulate the gate timings specified by Table 9 at the end of this write-up, which requires you to specify a delay for each gate in your design. To specify a delay, place a gate in your schematic in LogicWorks, right click it and select “Attributes”, select “Delay.Dev”, and enter the time in nanoseconds.

When you change the inputs, it will take some time for the effect of that change to reflect into the output. This difference in time is your total delay. Usually different input-output pairs have distinct delays. Even for the same input-output pair, its delay varies with distinct input combinations. The **maximum** delay value of each input-output pair is considered as its real delay value. After you finish your design, you need to measure and/or calculate the delays on the paths specified in Table 7 and fill in that table. In Table 7, x_i and y_i respectively denote the i^{th} bit of your two inputs (each input is 25 bits in total), b_i denotes the borrow-in for the i^{th} bit of your binary subtractor, and d_i denotes the i^{th} bit of your difference.

Path	Delay		Path	Delay		Path	Delay	
	Single-level	Two-level		Single-level	Two-level		Single-level	Two-level
b_0 to b_5			x_0 to b_5			y_0 to b_5		
b_0 to b_{10}			x_0 to b_{10}			y_0 to b_{10}		
b_0 to b_{15}			x_0 to b_{15}			y_0 to b_{15}		
b_0 to b_{20}			x_0 to b_{20}			y_0 to b_{20}		
b_0 to b_{25}			x_0 to b_{25}			y_0 to b_{25}		
b_0 to d_4			x_0 to d_4			y_0 to d_4		
b_0 to d_9			x_0 to d_9			y_0 to d_9		
b_0 to d_{14}			x_0 to d_{14}			y_0 to d_{14}		
b_0 to d_{19}			x_0 to d_{19}			y_0 to d_{19}		
b_0 to d_{24}			x_0 to d_{24}			y_0 to d_{24}		

Table 7: Path delay of 25-bit borrow look-ahead subtractor

In order to emphasize the differences in delays between the three subtractors (the ripple-borrow, single-level borrow lookahead, and the two-level borrow lookahead), we ask that you also measure the delay in some of the worst-case scenarios for different inputs changing. Simulate the input changes in Table 8 below, and record the delay from switching the inputs to the correct output. All inputs should change simultaneously.

Inputs	Delay		
	Ripple-Borrow	Single-Level	Two-Level
x: 000...000 → 000...000 y: 000...000 → 000...000 b ₀ : <u>0</u> → <u>1</u>			
x: 111...11 <u>1</u> → 111...11 <u>0</u> y: 111...111 → 111...111 b ₀ : 0 → 0			
x: 000...000 → 000...000 y: 000...00 <u>0</u> → 000...00 <u>1</u> b ₀ : 0 → 0			

Table 8: Delays from Specific Input Changes

Deliverables:

- LogicWorks Files:**

Part 1: 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: Include a LogicWorks design file (.cct) for each of the two 25-bit BLA subtractor circuits (with single- and two- level BLA generators), your 5-bit BLA subtractor, your 1-bit subtractor that generates your *g* and *p* values, and your BLA generator. Fill out tables 7 and 8 and include them in your report.

- 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, by examining the following cases and observing the output of your circuit:

- Take Codewords -8, -4, 2, 6, 10 and flip the 5th bit. Apply the resulting word to your circuit. Report the value in **A** after the completion of the first and the second phases.
- Take Codewords -6, -3, 3, 7, 9 and flip the 3rd and 7th bits. Apply the resulting word to your circuit. Report the value in **A** after the completion of the first and the second phases.
- Take Codewords -9, -7, 0, 8 and flip the 1st, 4th and 6th bits. Apply the resulting word to your circuit. Report the value in **A** after the completion of the first and the second phases.
- Find a code-word and flip **three** bits so that when you apply the resulting word to your circuit, the first phase reports an error while the second phase reports no error. Report this codeword, which code bits you flipped, and the value in **A** after the completion of the first and the second phases.

- Find a code-word and flip **four** bits so that when you apply the resulting word to your circuit, the first phase reports no error while the second phase reports an error. Report this codeword, which code bits you flipped, and the value in **A** after the completion of the first and the second phases.

For Part 2, you should provide your measured delay data by filling in Table 3 for both implementations of your BLA subtractor circuits. (See Table 9 below.)

Name	Delay (ns)	Name	Delay (ns)
2-input AND	24	2-input NAND	14
3-input AND	28	3-input NAND	18
4-input AND	32	4-input NAND	22
5-input AND	36	5-input NAND	26
2-input OR	24	2-input NOR	14
3-input OR	28	3-input NOR	18
4-input OR	32	4-input NOR	22
5-input OR	36	5-input NOR	26
2-input XOR	42	2-input XNOR	32
Inverter	10		

Table 9: Gate delay list