

Lab Assignment 2

Two's Complement Multiplication and Error Correction

Due October 26, Sunday, 4:30pm

Introduction

This Fall quarter in preparation for the upcoming DECAF in February 2015 your best friend, who landed a dream internship last year with a top technology company, informs you that building a portfolio of projects is the best way to impress the companies there and tells you that many of the companies are looking for students who have a firm grasp of digital logic and circuit design. You ask this friend of yours for some suggestion on what kind of a projects you can do. His first suggestion to you is to make a multiplier circuit. He tells you that you could take the easy way out and implement the simpler Robertson's Multiplier circuit, but to set yourself apart from the rest and because you are always up for a challenge, he wants you to implement a 19 bit Modified Booth's Multiplier circuit instead. Below, your friend has provided you with an overview of the basic multiplication algorithm (a.k.a. Robertson's algorithm), followed by Booth's algorithm, and finally the Modified Booth's optimization; he has additionally provided you a number of guidelines on how to implement this project. His second suggestion is that you build a Hamming Code error correction circuit. Once again, you could take the easy way out, but he recommends to figure out a way to extend the Hamming Code instead of simply implementing the standard Hamming Code circuit you can find in textbooks. He has provided for you an explanation of the standard Hamming Code, some guidance on Extended Hamming Code, and also a few hints as to how to go about implementing this project below.

Section 1: Two's Complement Multiplication

The following sub-sections cover a number of fundamental issues regarding the multiplication algorithms that you will probably need in order to complete this first project. After reading and understanding these, move on to the implementation section and get started on this project right away.

Basic Multiplication Algorithm

As your friend recalls from his CSE 140 lectures and the handouts given to him, basic multiplication works as follows: A multiplication algorithm consists of a series of shift and add steps, wherein the addition steps are performed whenever the corresponding

multiplier bit is a '1'. A two's complement representation necessitates a distinct treatment for the leftmost multiplier bit, as a 1 value over there has a negative positional weight of magnitude 2^{n-1} for an n bit integer, i.e., a positional value of $-(2^{n-1})$. As we know, a n bit two's complement representation enables the representation of numbers in the range $[-(2^{n-1}), 2^{n-1})$. Note that the left boundary, denoted by the square bracket, is a hard boundary that includes the left element, whereas the right one is a soft boundary which includes numbers less than, but not equal to the right value. When multiplying these two n bit numbers the resulting product will fall in the range of $(-(2^{2n-2}), 2^{2n-2}]$. All of these values with the exception of the largest positive result fit within $2n - 1$ bits. Your friend assures you not to worry right now about this one little issue of the largest positive result as it is covered later on in his overview.

Now that we have defined the fundamentals of a functional definition, i.e. the input/output definition, it is time to turn our attention to some finer points that must be addressed when implementing this multiplication algorithm. One first needs to address the question of the bit width for the adder/subtractor component that is required. It is easy to observe that the first step of adding to the partial result will produce a n bit sum, while the final partial sum (the product of the multiplication) can be up to $2n - 1$ bits long. While the latter part of this discussion seems to argue in favor of a $2n - 1$ bit adder/subtractor, one can easily notice that at any point the addition will only involve n bits, as any bits of the partial sum to the right of the actual summation will remain fixed and not affected by the operation. Therefore, the inputs to the two's complement addition that underlies the multiplication algorithm can thus be reduced to n bit numbers.

Now that we've started delving into issues of implementation, it may be time to look at, not only the underlying adder/subtractor which is going to be performing the computations essentially, but also at the storage units necessary to facilitate the computation. It is trivial to observe that some storage is necessary for the n bit multiplicand and multiplier. The partial sum will need to grow to $2n - 1$ bits wrapping up the storage requirements, which we can sum up to a total of $4n - 1$ bits for the multiplier, multiplicand, and the final partial sum. While $4n - 1$ bits are certainly sufficient for multiplication, a simple reduction in the storage requirements can be effected when one notices two things. The first thing to notice, is that after each bit is analyzed and the multiplier's value is shifted rightward through the multiplier shift register, the most significant bits of the multiplier register become unused. The second thing to notice is, that after each stage the least-significant bit (LSB) of the accumulator will not be modified, since the accumulator grows by one bit each cycle and only the leftmost n bits get factored into the addition, or subtraction if the multiplicand's most-significant bit (MSB) is a '1', or no-op when a '0' is encountered in the multiplier. Thus, we can shift in the LSB of the accumulator to the newly made available multiplier register space. By the time the calculation is complete, the

multiplier value is completely shifted out the multiplier register and now holds the rightmost n bits of the product.

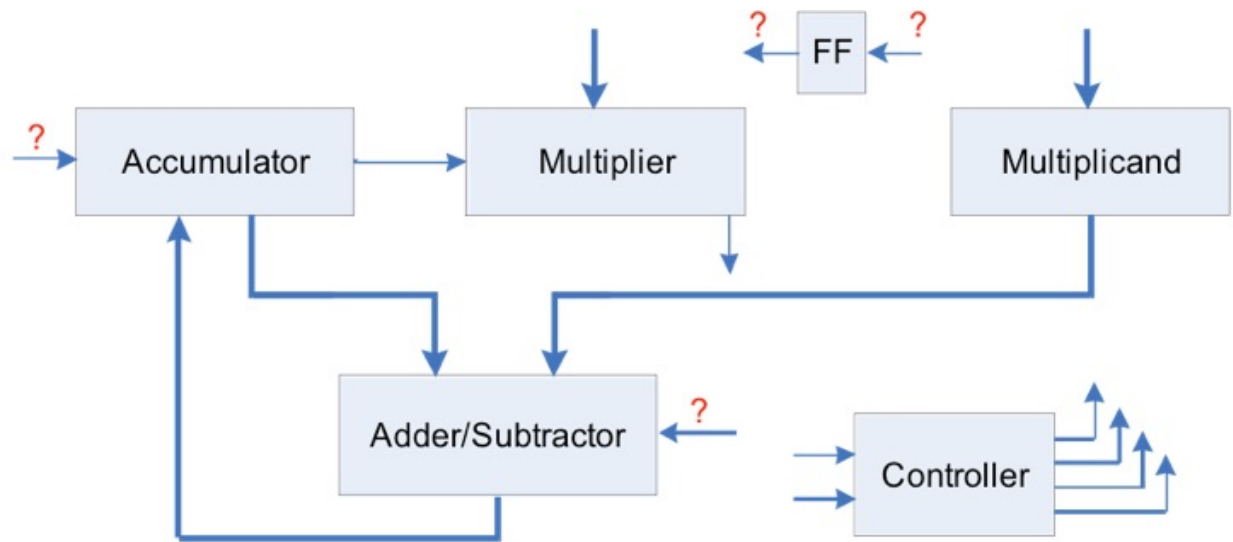


Figure 1: Accumulator Cannibalization of Multiplier Register

In the process of cannibalization of the multiplier, one can see that the $2n - 1$ bit partial sum storage has inherited an extra bit from the cannibalized multiplier register, thus effectively turning the two registers into a combined $2n$ bit accumulator containing the final product of the multiplication. This is just as well, since as we discussed in the previous paragraph, true n bit two's complement numbers actually lie in the range of $[-(2^{n-1}), 2^{n-1})$, whose product will produce results in the range $-(2^{2n-2}), 2^{2n-2}]$, thus necessitating truly a $2n$ bit representation for the one extremal result of 2^{2n-2} , the square of the largest negative number. Now let's move on to a somewhat more interesting and complicated algorithm for multiplication, namely; Booth's algorithm!

Booth's Algorithm

We will start this discussion by elaborating on an opportunity for an optimization which arises from a shortcoming of the generic multiplication algorithm previously discussed. In the generic multiplication algorithm, each step decides whether an addition, subtraction, or no-op is needed by examining the individual bit corresponding to that multiplier position; for a n bit number, $n/2$ of these steps are to be expected on average. Yet a moment's thought helps us realize that, in the binary numbering system, the sum of a consecutive sequence of ones can be easily calculated using the following equation, perhaps enabling us to reduce the number of addition/subtraction steps:

$$2^{i+k} + 2^{i+k-1} + \dots + 2^{i+1} + 2^i = 2^{i+k+1} - 2^i \quad (1)$$

In the binary numbering system, the left hand side of equation (1) is represented by a consecutive sequence of ones, starting from bit i to bit $i+k$ (counting from the right). If we use $\bar{1}$ to represent -1, equation (1) actually tells us that a standard binary number can be converted into a three-valued codeword representation. For instance, assuming $i = 1$ and $k = 7$, we can get the following transformation:

$$00111111110 = 010000000\bar{1}0 \quad (2)$$

Given this type of transformation, what is its impact on our multiplication algorithm? In the generic algorithm, we needed a possible add or subtract operation for each '1' value in the multiplier. On the other hand, equation (1), and its more concrete illustration in equation (2), tell us that a consecutive sequence of '1's in the multiplier, starting from bit i to bit $i+k$, can be converted into nothing more than a $\bar{1}$ at bit i and a '1' at bit $i+k+1$. Therefore, all the additions needed for bit $i, i+1, \dots, i+k$ can be replaced by an addition for bit $i+k+1$ and a subtraction for bit i . In this way, we can replace the consecutive sequence of additions with only two operations!

Scanning the multiplier from right to left, how do we detect where to place a '1' and where to place a $\bar{1}$? The discussion heretofore indicates that we need a $\bar{1}$ for the starting point of the sequence of '1's, that is, when we encounter a '0'→'1' transition. Similarly, we need a '1' for the finishing point of the sequence of '1's, that is, when we encounter a '1'→'0' transition. This is the optimization to Robertson's algorithm that is implemented in Booth's algorithm. Rather than simply looking at bit i of the multiplier to determine whether to add or not, in Booth's we examine two bits at a time: bit i and the bit immediately to its right at position $i-1$. If we see a sequence of ones starting from bit i to bit $i+k$, we can simply perform a subtraction when we are examining $M_i M_{i-1} = '10'$ (the starting point of the sequence of ones) and an addition when we are examining $M_{i+k+1} M_{i+k} = '01'$ (the finishing point of the sequence of ones).

One thing that perhaps you may worry about is what happens to negative two's complement multipliers and whether any special treatment needs to be accorded to them. It turns out that Booth's encoding has the extra benefit that negative numbers require no special handling whatsoever and Booth's encoding will automatically generate representations with the correct value embedded. Perhaps one way to realize this is that in

the case of a negative multiplier one or more bits of value '1' occupy the leftmost bits, thus ensuring that the last nonzero bit generated in the right-to-left sweep through the multiplier will be a $\overline{1}$, delivering a Booth's encoded representation corresponding indeed to a negative number. Perhaps a more concrete way to visualize this is by looking at an example, such as perhaps '11111000110' and realizing that this converts into '0000 $\overline{1}$ 0010 $\overline{1}$ 0', both bit strings representing, of course, the value -58 .

Because we are always looking at two bits at a time, we need a value for M_{-1} to help us complete the inspection of the LSB, M_0 . Therefore, in Booth's algorithm we add an extra '0' to the right of our multiplier input to help get the first step of our multiplication algorithm going. Furthermore, because we need to save this bit without changing the value of the multiplier, this extra bit is stored into a flip-flop which we refer to as FF in Figure 1 above instead of M_{-1} for convenience. Take note that a flip-flop is just a basic storage element for holding a value of a bit that can be read out or written over at any time. Each time the multiplier is shifted right, the LSB is fed into this flip-flop and in this way, we can always inspect the rightmost bit of the multiplier register and the value in the flip-flop to determine the arithmetic operation to be performed. The algorithm finishes when the last bit of the multiplier is shifted out of the multiplier and into the flip-flop. Once again the result is a $2n$ bit product across the two registers as it was in the previous algorithm. Now that we have basic Booth's out of the way, let's move onto what Modified Booth's brings to the table.

Rightmost Bit	Value in FF	Arithmetic Operation
0	0	No operation
0	1	Addition
1	0	Subtraction
1	1	No operation

Table 1: Booth's Multiplication Operation Table

Modified Booth's Optimization

Booth's algorithm is great for dealing with consecutive ones (and zeroes, for that matter, but hey even Robertson's does a good job at that), but when an input multiplier has alternating ones and zeroes, it would end up necessitating even more operations than simply performing an addition at each '1' in the multiplier that Robertson's algorithm would require. A bit of lookahead in examining the bit sequence may provide some help to the algorithm to obviate these unnecessary operations for the cases discussed. More concretely, looking at three bits at a time and avoiding the two alternating cases of '101'

and '010' can fix this. This is exactly the approach Modified Booth's takes to solve this issue. In Modified Booth's algorithm, we need to inspect $M_{i+1}M_iM_{i-1}$ to determine the current operation that needs to be performed. This means that inspecting M_{n-1} , the MSB of a n bit multiplier, is a problem if we don't add a dummy bit to the left of our input multiplier as well. To do this, you must increase the bit width of the multiplier from n bits to $n+1$ bits, where this additional bit is filled in through sign extension. Keep in mind that Modified Booth's still makes use of the flip-flop denoted as FF to the right of the multiplier like standard Booth's.

Implementing the modified version of Booth's algorithm means that, in addition to examining three bits to determine whether to perform an addition or subtraction (or no operation whatsoever), you also need to be able to modify the value going into the flip-flop FF, which is M_{-1} as a reminder. This is so, because in the case of an alternating sequence, we need to kill the alternation by lying about the previously seen triplet by treating '101' as if it were '111', for example, eliminating the operation that would have occurred on the next cycle, 'x10' in this example. In other words, we end up treating '101' effectively as '111' with all the associated benefits of converting long strings of ones to zeros, i.e. no operation for the multiplier. We have to of course pay back the additional '1' we have borrowed in the middle of the string '101', by forcing a $\bar{1}$ output at this point. The three bits we'll take into account to determine the appropriate operation will be the two rightmost bits of the multiplier and the value that resides in the flip-flop FF. Again as in standard Booth's algorithm, negative multipliers work perfectly with these rules and no special care needs to be taken. Take a moment to convince yourself of this with a few examples.

Second Rightmost Bit	Rightmost Bit	Value in FF	Arithmetic Operation	Value to shift into FF
0	0	0	No operation	0
0	1	0	Addition	0
1	0	0	No operation	0
1	1	0	Subtraction	1
0	0	1	Addition	0
0	1	1	No operation	1
1	0	1	Subtraction	1
1	1	1	No operation	1

Table 2: Modified Booth's Multiplication Operations

The next implementation point to touch on, which we skipped over in both Robertson's and Booth's algorithms, is how to fill in the leading bit of the accumulator register after the partial sum has been shifted right. Your friend admits his memory is a little hazy on what to do here exactly in the cases of Booth's and Modified Booth's algorithms so some experimentation may be required to get this correct. Your friend suggests that you should think about what this may be in Robertson's and standard Booth's algorithm as it might help you understand what to do in Modified Booth's or at least provide some insight. Furthermore, your friend is a bit hazy as to whether the Modified Booth's algorithm is prone to integer addition overflow or not, and what to do in the case that it is as this will surely result in an incorrect result if it happens. He does remember in the case of standard Booth's algorithm that an overflow is not possible but cannot remember the explanation for why this is so, just that he was told it couldn't happen by his eccentric CSE 140 professor and TAs. Now with these high level overviews, let's move on to making this circuit.

LogicWorks Implementation

With this comprehensive understanding of multiplication algorithms provided to you, we are ready to start implementing this project. Your friend suggests for this first project you use LogicWorks 5 to implement the Modified Booth's multiplication algorithm. LogicWorks can be found in the CSE basement labs on all of the machines on the Microsoft Windows environment.

To help get you started your friend has provided you with a library file, *Lab2Fa14.clf*, that has custom components that he built within it that will be crucial in implementing this circuit and will speed the process along. LogicWorks also has all the standard gates, flip-flops, switches, binary probes, and LEDs available in its built-in libraries that will aid you in this project and become instrumental during the testing of this circuit.

Your friend suggests for starting off, for you to take a look at both the library mentioned above (described below in the *Components provided in Library section*) and Figure 1 and become intimately familiar with both of these. Afterwards begin building the basic circuit as seen in Figure 1 with the provided custom components along with the built-in components of LogicWorks. Your friend tells you that for right now use the 20 bit adder/subtractor; he will explain later on why there are two adder/subtractors and why you are starting off with the 20 bit version. For the input entry use five hex keypads each for the multiplicand and multiplier. Take note that for the 19 bit shift registers one of the hex keypad pins will not be used and be careful not to wire these components backwards. For the 38 bit result display use 10 hex displays and once again be careful not to wire these up backwards either. Remember that the result is read from the outputs of the $2n$ bit partial sum, which resides across the concatenation of the accumulator and multiplier registers in

the leftmost 38 bits. For FF use a *D Flip Flop En wo/SQ/* from the built-in libraries. The controller, in the provided library, must be wired up to make this circuit run. Take a look at Table 3 in the *Components provided in Library* section below to gain an understanding of the inputs and outputs of this controller and to get an idea of where they should be connected.

To complete the design you need to implement some glue logic regarding three issues. The add/subtract unit needs to be handled appropriately, the load decision needs to be made for the accumulator register, and the FF flip-flop needs to be appropriately loaded at each step. Perhaps the last question is the easiest to get started with as the Modified Booth's table can be used as the truth table for the logic implementation; if you do this carefully you will notice that the logic that you are implementing is actually a logic you have studied for a quite basic arithmetic operation and should cost you no more than 4 AND/OR logic gates. The remaining two questions are a bit more thought provoking, as they benefit from being considered together. You will notice in the current way of implementing multiplication an operation (either addition or subtraction) will take place every step but its effects may be discarded by foregoing the loading of the result. Thus in selecting whether an add or subtract operation is to take place for the cases when the result will be discarded, one can exercise foresight to reduce the control of the add/subtract unit to nothing but the appropriate wiring of a bit on the multiplier. The loading operation can be easily derived from observing in the truth table that it is quite symmetric, resulting in you using nothing but a simple gate (albeit maybe a bit more complex than your typical AND/OR gate).

Additionally, take note of the two different shift register sizes and be sure to use the appropriate sized registers for multiplicand, multiplier, and the accumulator. Finally, for the 20 bit shift register you will have to think about what value to load into the MSB initially (hint: it is not from an external input like the 19 rightmost bits).

Now with all tasks above complete and points made above considered, it is time to move on. Part 1 will be the start of tackling the last two remaining concerns, the issue of the accumulator shift-in value and addition/subtraction overflow possibility.

Part 1

This first design, as mentioned above, uses a $n+1$ bit, i.e. a 20 bit, adder/subtractor to sidestep the errors that may occur with addition/subtraction overflow in the case that Modified Booth's Multiplication is prone to this behavior. As you may have already thought about, by simply increasing the size of the adder/subtractor by one bit the overflow issue is precluded as the adder/subtractor has sufficient headroom to keep track of the sign bit in the event that an overflow would have occurred. Also, using the extended $n+1$ bit adder/subtractor not only takes care of masking whether overflow is an issue, it

also contains the correct value that should be shifted into the MSB of the accumulator during the following shift phase. This value that should be shifted into the accumulator is located in the MSB of the extended $n + 1$ bit adder/subtractor. This value should be stored into a flip-flop anytime the adder/subtractor result is saved into the accumulator. The flip-flop's output will become the input for the accumulator's shift-in port. Think about why the MSB of the extended adder is the correct value and perhaps this knowledge may help you in the latter parts of this project. Be sure to *include any explanation you have for why this shift in value is correct in your report*. Keep in mind that input registers are still only 19 bits wide, so you will need to deduce a way to use a 19 bit value from the register as an input for the extended 20 bit adder/subtractor.

Your friend tells you, that since this circuit will always produce correct results if everything is connected properly, you should use it to verify that all of your logic and wiring described above is correct using the test cases he has provided for you below in the *Test Cases* section. Your friend soon recalls that there is one corner case that the algorithm is not designed to handle involving either the multiplicand or the multiplier set to zero. He tells you this corner case is usually handled specifically but for this project to not worry about it and just take note which one doesn't work. *Make sure to include in your report which one of these two corner cases does not work and an explanation for why you think it doesn't*. Once you have proven your multiplier design works correctly in this configuration you can then use this working design to test if overflow is possible by examining some test cases cycle by cycle. You should work out some pen and paper examples to convince yourself whether it can or cannot overflow as well. Knowing this answer upfront will help you tackle Part 2 below. *Be sure to include in your report if an overflow was or wasn't observed. If it was, provide a written out test case to demonstrate this; if it cannot occur provide a reasoning for why in Modified Booth's overflow is not possible*.

Part 2

As we discussed previously, the design in Part 1 uses an $n + 1$ bit adder/subtractor to resolve the possibility of an overflow issue and to calculate the shift-in value for the accumulator. Unfortunately, such a strategy necessitates a larger adder/subtractor, which imposes penalties in both the hardware cost and circuit delays. Your friend asks that for this second part you alter your current Modified Booth's implementation to optimize the design by replacing the 20 bit adder/subtractor with a 19 bit adder/subtractor component. He informs you that the accumulator shift-in logic will need to be replaced as a result of reducing the bit width of the adder/subtractor as well. After pondering the implications of this change for a few minutes, your friend believes that the most natural way to provide a correct shift-in value for the accumulator would be to perform a simple sign extension of the accumulator's current value. You will have to come up with some glue logic for making this happen but it shouldn't be too difficult. You should *record and discuss results for the same*

test cases used in Part 1. Report test cases that do not produce the expected product and why that may be. Analyze closely any test cases that fail cycle by cycle and try to determine when they fail and what may have caused the failure. Be sure to record these findings in your lab report.

Part 3

Although the circuit designed in Part 2 optimizes hardware over that in Part 1, the circuit contains somewhat conspicuous flaws that your friend has noticed. As you may have also discovered, sign extension works in some test cases but it fails in others. Obviously you will need to revisit your design and create another iteration of the Modified Booth's circuit.

This time your friend suggests that an alternative solution for the shift-in value would be to consider the carryout value from the adder/subtractor as this, he believes, will mimic the values produced in the MSB of the 20 bit adder/subtractor used in Part 1. With your friend's suggestion seeming logical, modify the circuit this time to use the carryout value of the adder/subtractor for the shift-in of the accumulator. *Report what did not produce the expected product in your report and why it may be. Examine closely any test cases that have failed cycle by cycle and try to determine when these cases failed and what may have caused the failures. Be sure to record any examples of failures in your lab report. Also mention in your report if you have any disputes with your friend's suggestion for the shift-in value used and what that dispute is.*

Part 4

There still seems to be something wrong with this multiplier circuit your friend tells you after observing your last round of tests. Perhaps the shift-in value for the accumulator is not dependent on the outcome of addition/subtraction. He suggests that this time you build your circuit to shift in the MSB of the multiplicand as he noticed that during some of your testing the MSB of the multiplicand after a bunch of additions was always the correct value that needed to be shifted into the accumulator.

With this new recommendation from your friend, modify your circuit from Part 3 to shift in the MSB of the multiplicand into the accumulator. Again, you should *record and discuss the results for the test cases that did not produce the expected product in your report* by identifying at least the first point at which the partial sum calculations falter.

Part 5

At this point you are probably starting to question your friend's understanding of this algorithm. But being the good friend that you are and keeping in mind that he did land that dream internship last year, you decide to press on and keep listening to his advice; after all he did admittedly say he wasn't sure about this part of the circuit.

As your friend was reflecting upon why he chose to use the MSB in the first place he noticed one slight flaw in his observations. The test cases that he observed where the MSB was the correct shift in value all had one thing in common; they all had only addition operations and no subtractions at all. From these observations your friend thinks he has this problem licked this time with his new solution. He tells you with great enthusiasm to modify your current design to shift in the multiplicand's MSB when you last performed addition or shift in the complement of the multiplicand's MSB if you last performed subtraction. He tells you that this can be implemented by adding only one additional logic gate, specifically an XOR gate, to the design of Part 4 and challenges you to figure how this is done with just one gate. Again, you should *record and discuss the results for the test cases that did not produce the expected products in your report* by identifying at least the first point at which the partial sum calculations falter and what may have caused these failures.

Part 6

Your friend has come to you with that look of disappointment in his face, the sad puppy dog kind of look if you will. He tells you that as you probably have figured out by now he was wrong once again. He was so sure last time and excited about his prior solution being correct that he admits, he overlooked one last thing that has been plaguing this design since Part 2. With such a sad look on his face and his constant murmuring that, this fix will be the last and it will be correct, you decide to give him one last shot. If he is wrong this time though you are just going to figure this out on your own. Your friend tells you that some of the test cases begin to fail when the accumulator is still a zero value, i.e. no addition/subtraction operations have occurred yet, but a '1' is shifted into the accumulator which of course makes the accumulator a non-zero value and that should not be the case. This time however, your friend says you are all on your own on figuring this part out but he will give you a couple of hints; first, you need some way of detecting or keeping track of whether the accumulator has become non-zero; second, you need to amend the accumulator shift-in logic from Part 5 with the previously mentioned hint so that it only shifts in zeros until the accumulator has become non-zero. Your friend tells you good luck and promises that this should fix it this time, which is a good thing because that dream internship he got last year is flying him to a conference in Vegas all expenses paid and he will be gone for one week after today, but of course back just in time to help you with your second project, i.e. the Extended Hamming Code project. So you are all on your own for now so take these last pieces of advice and what you have learned from the previous parts and hopefully finish this project up.

Be sure to thoroughly test your implementation ensuring the results for the test cases your friend has provided are correct. Your friend also recommends you come up with and try many more test cases than the ones he provided to ensure this last design is finally correct.

Components in Provided Library (Lab2Fa14.clf)

Shifter19

Inputs:

- **LD:** when set, the input connected to D18-0 is loaded into the register;
- **D18...D0:** 19 bit data to be loaded when LD = 1;
- **SH:** when set, the value of the register is shifted 1 bit to the right, with the most significant bit fed by SI;
- **SI:** the shift-in value to be placed in the most significant bit during a shift operation;
- **CLR:** resets the value of the registers to 0.
- **CLK:** a standard clock input that should be connected to a global clock in your circuit.

Outputs:

- **Q18...Q0:** Output bus that shows the current value of the register.

Shifter20

Inputs:

- **LD:** when set, the input connected to D19-0 is loaded into the register;
- **D19...D0:** 20 bit data to be loaded when LD = 1;
- **SH:** when set, the value of the register is shifted 1 bit to the right, with the most significant bit fed by SI;
- **SI:** the shift-in value to be placed in the most significant bit during a shift operation;
- **CLR:** resets the value of the registers to 0.
- **CLK:** a standard clock input that should be connected to a global clock in your circuit.

Outputs:

- **Q19...Q0:** Output bus that shows the current value of the register.

Adder/Subtractor19

Inputs:

- **A18...A0, B18...B0:** two 19 bit operands to be added;
- **A/S:** the signal that indicates which operation to perform: A/S = 0 for an add and A/S = 1 for a subtract.

Outputs:

- **Q18..Q0:** the 19 bit sum of the two operands;

- **CO:** indicates whether or not there was a carry (or borrow) out from the operation performed.

Adder/Subtractor20

Inputs:

- **A19...A0, B19...B0:** two 20 bit operands to be added;
- **A/S:** the signal that indicates which operation to perform: $A/S = 0$ for an add and $A/S = 1$ for a subtract.

Outputs:

- **Q19..Q0:** the 20 bit sum of the two operands.

Controller

Inputs:

- **CLK:** a standard clock input that should be connected to a global clock in your circuit;
- **Begin:** a transition from 1 to 0 on this input will start the multiplication.

Outputs:

- **End:** indicates that the multiplication is completed;
- **Init:** the signal to initialize the multiplication process, which should be used to clear the accumulator, load the input values, and reset the flip-flop;
- **LoadA:** a '1' on this signal indicates that the results of the adder should be loaded into accumulator A;
- **Shift:** a '1' on this signal indicates that the contents of the accumulator and the multiplier should be shifted 1 bit to the right.

In sum, the outputs of the controller on each cycle will be as follows:

Cycle	End	Init	LoadA	Shift
0	0	1	0	0
1,3,5,7,9,...,35,37	0	0	1	0
2,4,6,8,10...,36,38	0	0	0	1
39	1	0	0	0

Table 3: Controller Output Table

Section 2: Error Correction Code

Standard Hamming Code

As your friend recalls from his in-class discussions and handouts given during CSE 140 and CSE 140L, Hamming Code is used to implement an error correction algorithm. This algorithm can be used to detect and correct a single bit error. The way this algorithm works is by generating a set of additional bits that record the parity of a codeword. If the bits being examined have an odd number of ones, these bits are said to have an odd parity. If the examined bits have an even number of ones, they are said to have even parity. The goal of the standard Hamming Code is to generate parity bits that are used to give each set of numbers being examined an even parity. For example the set $\{1, 1, 0\}$ clearly has an even parity. Thus the generated parity bit for this set would be 0. If we add this bit to the existing set we get the set $\{1, 1, 0, 0\}$ which also has an even parity. In *Hamming* $(7, 4)$, there are four data bits and three parity bits that are added to the data bits to give seven bits total. In general, a Hamming Code formula can be represented as *Hamming* $(n + c, c)$, with n representing the number of data bits and c the number of parity bits that needs to be added. The number of parity bits that must be added adheres to the follow formula:

$$n \leq 2^c - c - 1$$

Thus in the case of *Hamming* $(7, 4)$ where $n = 4$, it is clear that 3 parity bits must be generated in order to satisfy the formula above ($4 \leq 2^3 - 3 - 1 = 4$). Each parity bit is responsible for checking some subset of bits in the original codeword. This subset of bits that each parity bit must check is called the parity function. This allows the potentially incorrect bits in a codeword to be identified, because each data bit has a unique combination of parity bits that are responsible for checking it. For example, in the set of parity bits $P_4 P_2 P_1$ shown below: $P_1 P_2$ check data bit 3, $P_1 P_4$ check data bit 5, and $P_1 P_2 P_4$ check data bit 7. This is the main way that parity of the data bits is recorded and then checked by the correction formulas. Of course, the generation of these parity bits can be performed by taking an XOR of the bits in the parity functions. The formulas for the parity bits of the *Hamming* $(7, 4)$ codewords are:

$$P_1 = \text{XOR of bits } (3, 5, 7)$$

$$P_2 = \text{XOR of bits } (3, 6, 7)$$

$$P_4 = \text{XOR of bits } (5, 6, 7)$$

Using these additional parity bits that were generated we can detect and correct any single bit error. This is made possible because we can compute a formula for the correction bits. This new formula is identical to the original formula for the generation of the parity bits, but also includes an additional XOR of the parity bits itself. We know that after the parity bit has been generated all the parity functions should have an even parity. Thus if one of the bits has been flipped in the transmission process, the correction bit will be a one. This is because the subset of bits this correction bit is checking will have an odd parity. This means that the given formula contains an odd number of ones, which violates the even parity that we wanted each formula to have. For the *Hamming* (7,4) formula, the correction bits were computed as:

$$C_0 = \text{XOR of bits } (1, 3, 5, 7)$$

$$C_1 = \text{XOR of bits } (2, 3, 6, 7)$$

$$C_2 = \text{XOR of bits } (4, 5, 6, 7)$$

Notice that the only difference between the correction bit formulas and the parity bit formulas, is that in the correction bits, the parity bits are being XORed with the original function. Thus for C_0 , the bit at position 1 (the location of P_1) is included in the formula. Each of the correction bits should have an even number of ones. If one of the bits does not have this even parity property, it means an error has occurred during the transmission. Thus we can also derive from these formulas that $C_2C_1C_0 = '000'$ implies that no error has occurred. Take note that bit positions are counted starting off from a position numbered 1, not 0, because as just mentioned, a result of '000' for the correction bits denotes the absence of an error. However, If an error did occur, say in position one, this error can be identified because the correction bits will be computed as $C_2C_1C_0 = '001'$. This means that the subset of bits {1, 3, 5, 7} that C_0 is responsible for checking produced an odd parity. Notice that bits C_2 and C_1 correctly produce an even parity. This absolves bits {2, 3, 4, 5, 6, 7} from being the source of the error, thus leaving only bit one (corresponding to the parity bit itself, P_1) as the only possible cause of the error. Similarly, in case $C_2C_1C_0 = 101$, the incorrectness of C_0 and C_2 implicates the intersection of the bits involved in the two equations, namely, {5, 7}. Yet the correctness of bit C_1 absolves bit 7, leaving data bit 5 as the only explanation for the failure.

Extended Hamming Code

For many applications standard Hamming Code would be considered sub-standard error detection and correction as these applications dictate higher requirements for their correction operation. A Single Error Correction, Double Error Detection (SEC-DED) code

seems safer and it is the type of correction and detection method most often used in modern computer memory architectures. The *Hamming (7,4)* can be converted to a SEC-DED code by adding one additional bit, which is a parity bit, let's assume even, on all 7 bits in the codeword. This code is called an Extended Hamming Code and is what your friend is suggesting you to implement. To understand how Extended Hamming Code is SEC-DED, consider Table 3 below.

Errors	Overall Parity	Correction Bits	Conclusion
0	even	All zeros	No Error
1	odd	All zeros	Overall parity bit is in error, error corrected
1	odd	non-zero	Parity equation is in error, error corrected
2	even	non-zero	Double error, not correctable

Table 3: Extended Hamming Code Possibilities and Outcomes

If there are no errors present, the overall parity will be even and the corrections bits of the codeword will be all zeros. If there is only one error, then the overall parity of the received codeword will be odd. If the single error occurred in the overall parity, then the correction bits will be all zeros. If the single error occurred in some other bit of the codeword, then the correction bits will be nonzero and they will indicate the index of which bit is in error as per standard Hamming Code. If there are two errors, then the overall parity of the received codeword will be even but the correction bits will be non-zero. Take a moment to convince yourself that no matter where the 2 errors have occurred, they are still not correctable.

By now you may be wondering why bother with the additional overall parity if two bit errors are still not correctable, what have we gained? Let's take a moment and examine the benefit Extended Hamming Code has over standard Hamming Code with the addition of the overall parity. In standard Hamming Code, when two bits of a codeword have changed, the circuitry will detect that an error has occurred and attempt to perform a correction; however, this correction will be incorrectly performed and the output codeword will still contain errors. In Extended Hamming Code, you know that if the corrections bits indicate an error, i.e. they are non-zero, but the overall parity is still even, there must have been 2 errors that have occurred in the codeword to transform the overall parity back to

even. Since you know that two errors have occurred, you can signal to other circuits that this codeword has errors in it that can't be corrected and it can be discarded.

The addition of overall parity changes the Hamming distance of the codewords which is the underlying theory behind why Extended Hamming Code has gained the additional capability of Double Error Detection. *Be sure to record in your report what happens to the Hamming distance in Extended Hamming Code, i.e. what does it change to? What was it originally? What is the problem with standard Hamming Code when it performs error correction in the event that two errors have occurred?*

Now that you know exactly how Extended Hamming Code works, let us move on to making this project.

Verilog Implementation

Your friend recommends that you implement this in Verilog to serve as an introduction to Verilog Hardware Description Language (HDL) while at the same time gaining an understanding of Hamming and Extended Hamming Code. This project will consist of creating a single Verilog module that can perform basic error correction using the Extended Hamming Code technique. The Hamming codewords we will deal with in this project will be the *Hamming* (15,11) codewords. This means the 15-bit codeword will contain 11 data bits and 4 parity bits ($n = 11$, $c = 4$, $11 \leq 2^4 - 4 - 1 = 11$).

In order to start designing this Verilog module, you will first want to consider the other differences when dealing with 15 bits instead of 7 bits. The exact parity functions for these bits will be slightly different than the *Hamming* (7,4) codewords. This is because you will now be dealing with 11 data bits instead of 4, and thus the subset of data bits that each parity bit must check is greatly increased. Also remember that each bit in the codeword must be uniquely identifiable. This is necessary because in order to properly correct an error we must be able to identify what position it is coming from. Thus when deriving the formulas for your *Hamming* (15,11) parity bits, keep this property in mind. Of course, because differences exist when generating the parity bits, these same discrepancies will be reflected in the correction bits. As explained earlier, the formulas for the correction bits are simply the XOR of the original parity functions, with the parity bit that was generated. This is captured in the formulas shown below, where P_8 , P_4 , P_2 , and P_1 stand for the value of the parity bit, and $F(P)$ stands for the parity function you must identify:

$$C_0 = P_1 \text{ XOR } F(P_1)$$

$$C_2 = P_4 \text{ XOR } F(P_4)$$

$$C_1 = P_2 \text{ XOR } F(P_2)$$

$$C_3 = P_8 \text{ XOR } F(P_8)$$

You will also want to consider how many additional overall parity bits will need to be added to make these *Hamming* (15, 11) codewords an Extended Hamming Code. Remember this means changing the Hamming distance of the codewords so that they are SEC-DED.

You will create a Verilog module that is able to take an input codeword and produce a corrected output codeword. Though you could just output the data portion let's just assume for this project you want to see the entire codeword at the output of this module. As you will find out later this will make your testing easier as well. Remember that parity functions will be a simple function of the XOR gate, which is included in the standard Verilog bit-wise operations. If no errors are found in the transmitted codeword then the output codeword should be the same as the input. If one error is present in the input codeword, the bit that was found to be incorrect will need to be corrected before output. If two errors are detected then an error output indicating error in transmission should be raised to logic high to indicate that this codeword is not usable. A few words about the correction circuitry; your friend has decided to leave the mechanics of how this is done to you but recommends you take a look at what a decoder does and how it might be used in conjunction with, say an XOR gate, to perform correction. You should also consider the additional logic needed to make use of the overall parity information and how to output a logic high on the error output when any two bit errors occur.

In this project you will use the Altera Quartus II development tools, which are installed on all lab computers in the CSE building's basement in the Microsoft Windows environment. Altera Quartus II development tools can be downloaded for free from www.altera.com for both Windows and Linux if you so choose to use your own computer for this project. The Quartus II tool will be used for the creation and compilation of a Verilog project. After a successful compilation in Quartus II, your module can be simulated using the ModelSim tool. Two files have been provided for you; a skeleton Verilog module for the Extended Hamming code circuit and a small testbench file for you to use to verify your design before turning it in. This testbench will work by simulating a series of codewords as inputs to your Verilog module. For each input provided, your module will produce a corresponding output codeword. The output will then be examined for correctness by the testbench which will notify you of the test results. Be sure to add additional test cases to the testbench before submitting your code, because the provided testbench is somewhat limited. Remember that checking the design of your Verilog module is critical to making sure your Extended Hamming code implementation is correct. Understanding the code written in the testbench will prove quite valuable when writing your own testbench modules in any future Verilog work you may find yourself doing. So with all that said, your friend suggests you get started on this right now!

Implementation Deliverables

Section 1: As part of your submission, include a LogicWorks design file (.cct) for each part that contains a complete multiplier circuit using the components provided all together in one zip file (Parts 1 - 6, six files). Be sure to test your circuit with several different combinations of inputs in addition to the test cases that were provided to detect problems and aid in your understanding of the advantages and/or shortcomings of each implementation.

Section 2: Submit the Verilog files that contain your Extended Hamming Code error detection/correction module and your testbench module. Start with the provided skeleton code for the Hamming Code module and the testbench module and modify these for your implementation. Note that the interface for the Hamming Code module should not change drastically when implementing your version. Be sure to thoroughly test your design as the testbench we have provided you is somewhat limited in its functionality and the testbench we will use is much more exhaustive.

Report Deliverables:

Section 1: Submit a report in PDF format explaining your designs of the various multiplier implementation for Part 1 - 6, include your reasoning about the glue logic and how you came to develop it. Answer all the questions contained in the lab and report findings, examples, and other items pointed out in *italics* in Parts 1 - 6. Make sure you run your implementation on all the test cases shown in the next section (and hopefully many others) and let us know which of the parts they failed and why.

Section 2: Submit a report in PDF format that includes a general description of how you implemented the Extended Hamming Code error correction/detection algorithm. List the functions for the parity bit formulas that we asked you to derive and include a brief description of what led you to these derivations. Include answers to all questions in the lab and everything pointed out in *italics*. Lastly, answer the following question: Can three data bit errors be detected in Extended Hamming Code; if so, can any of the bits be corrected? A brief paragraph here will suffice.

Test Cases

<u>Test Case:</u>	<u>Multiplicand:</u>	<u>Multiplier:</u>	<u>Product:</u>
1	0x0006B	0x0006B	0x0000002CB9
2	0x43657	0x06580	0x3E7F8B7E80
3	0x7FFFE	0x00005	0x3FFFFFFF6
4	0x43210	0x7FFD3	0x0000AB3330
5	0x50000	0x00001	0x3FFFD0000
6	0x7FFFF	0x7FFFF	0x0000000001
7	0x7FF95	0x0100B	0x3FFF94B67
8	0x60591	0x000F9	0x3FFE136A09
9	0x70010	0x11100	0x3EEF111000

Table 4: Given Multiplier Test Cases