

Lab Assignment 2

Two's Complement Multiplication and Error Correction

Due August 13, 2014, Wednesday, 10:30am

Introduction

In this lab you are asked to complete an assignment composed of two parts. In Part 1, you are asked to build several 24-bit two's complement multipliers in LogicWorks using Robertson's multiplication algorithm. We start the write-up with the basic Robertson's multiplication algorithm, and will guide you through several possible multiplier implementations for two's complement numbers. In the second part, you will use Verilog to implement a Hamming(15, 11) error corrector using the Altera tools provided in the basement labs.

Two's Complement Robertson's Multiplication

In this lab, you will build several 24-bit two's complement multipliers in LogicWorks using Robertson's algorithm. We are providing you with the major components for the circuits: the 24-bit shift register, the control unit, and the Adder/Subtractor. Each implementation will differ slightly from its predecessors and will make certain hardware optimizations. We are also providing you with a general explanation of how to implement a Robertson's multiplier and will guide you through the optimizations at each step. We also present a somewhat incomplete schematic of the design, in order to help you understand the fundamentals of Robertson's algorithm. In return, we are asking you to do your bit and help us by connecting the given components and supplying the glue logic that you will develop for a few connections. At the completion of each step you will produce an implementation of a Robertson's multiplier that may or may not be complete. We then ask that you ponder the changes at each step, and determine the faults and successes of each design.

The Basic Idea: Robertson's Multiplication

Fundamentally, a multiplication algorithm needs to check each digit of the multiplier (conventionally, from the least significant to the most significant digit) and accumulate a partial sum. The pencil and paper method that you should be quite familiar with from your childhood days does the same thing, except that you would typically sum all the partial products in one scoop rather than iteratively step-by-step.

If we turn our attention to the binary two's complement multiplication hardware, we note that the process described in the previous paragraph holds except with an important simplification. Since each digit in a binary system can only be a 0 or a 1, deriving the partial products is trivial to accomplish. The partial sum which accumulates the result of our multiplication circuit iteratively as is to be expected starts off at 0. Starting from the least significant bit (LSB) of the multiplier, if the i^{th} bit (counting from the right of the multiplier) is a '1', we shift the multiplicand left i times and add it to our partial sum. This operation is repeated until the most significant bit (MSB) is encountered which is the sign bit of the multiplier. Since the sign bit has a negative weight, we need to subtract the shifted multiplicand from the partial sum (instead of the usual addition) if it is 1.

Note that if this algorithm were implemented in hardware as described, we would need a distinct storage element to save the partial sum at every step. A much simpler implementation allows us to use fewer resources. We keep a running total (an accumulation) in a single accumulator register which can be reused in each step. Hence in order to perform n -bit multiplication, a $2n-1$ bit **accumulator** register, a

$2n-1$ bit **multiplicand** register (because the n -bit multiplicand needs to be shifted left $n-1$ times) and an n bit **multiplier** register would be required.

Can we reduce the hardware even further? The answer is a definite yes if you consider the following two observations:

- Instead of shifting the *multiplicand* 1-bit **left** on each step, we can just as easily shift the *accumulator* 1-bit **right**. In this way, the addition still lines up correctly, while the multiplicand, which does not need to be shifted, can be stored in an n -bit register.
- The *multiplier* bits that have already been examined in previous steps are no longer useful for subsequent steps. Therefore, if at every step we shift the multiplier **right**, the higher bits of the multiplier register can be used to store the extra bits required by the partial sum.

Based on these two observations, we can design a multiplication circuit that only needs three n -bit registers respectively for the partial sum, multiplier and multiplicand. During the multiplication process, both the **accumulator** and the **multiplier** registers are shifted 1-bit right on every step, while no shifting is performed on the **multiplicand** register. The bit being shifted out of the **accumulator** register on each step is fed to the leftmost bit of the **multiplier** register for storage. An extra advantage of this scheme is that the bit of the multiplier at each step that we are interested in to determine whether to add the **multiplicand** or not is consistently found at the same physical bit line of the **multiplier** register (i.e. the rightmost bit line), thus simplifying the control and the interconnect. If the rightmost bit of the multiplier register is a '1', the contents of the **multiplicand** and the **accumulator** registers are added (or subtracted if this happens to be the MSB of the multiplier), and the results are written back to the **accumulator** register. A partial datapath that illustrates this multiplication design is shown in Figure 1. Notice that the accumulator register and the multiplier register are connected so that the bits being shifted out of the accumulator get shifted into the multiplier register for storage. Both registers hold a part of our end result.

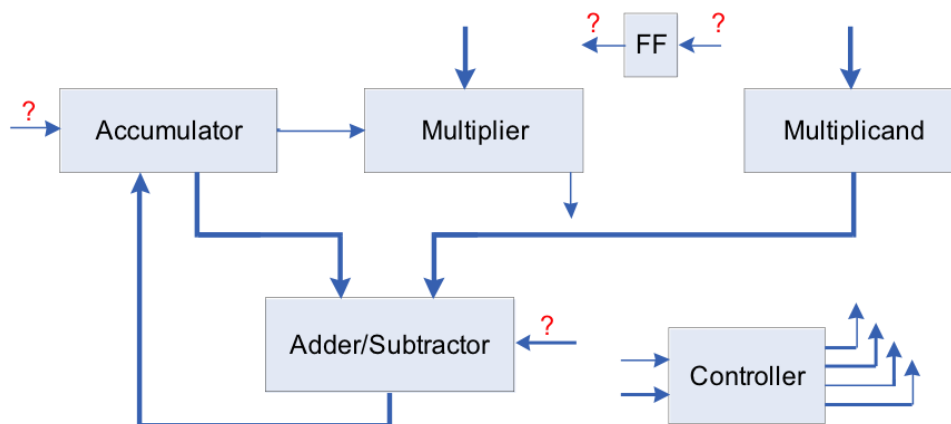


Figure 1: A partial datapath of 2's complement multiplier

When a register performs a right shift, every bit is replaced with its left neighbor, except for the MSB, which has no such neighbor. For this reason the register has a **SI** pin ("Shift-in"), and whatever value is on this pin during a shift gets placed in the MSB of the register. We saw previously that the multiplier's MSB gets shifted in from the LSB of the accumulator, but what about the MSB of the accumulator itself? There are a few different ideas about what could possibly be shifted into the accumulator. In this lab we will present you with several possible solutions for the Shift-in value. We ask that you ponder each circuit you will be implementing and draw conclusions about what multiplier/multiplicand inputs will work for each circuit. To guide your thinking through this process, we have provided a series of test values that you should use as inputs. These inputs will produce different results for each circuit that will help lead you to see the faults and/or benefits of that design.

The Controller:

The controller coordinates the multi-cycle process, dictating to each device what to do on every cycle until the entire process is finished. Every other device simply does what it's told on each cycle, and has no concept of how far into the multiplication algorithm it has progressed, how to coordinate itself with other devices, or when they are finished.

The controller has output signals that can be used to coordinate the actions of the other devices. These outputs must be connected to the correct inputs of the devices in order for control to be effective. For example, when the algorithm begins, each sequential device must initialize itself: the multiplier and multiplicand by loading input values, the accumulator by clearing to zero, and the D flip-flop by resetting to zero. The control unit provides a single line ('Init') that can be used for this purpose, which goes high ('1') on the first cycle of the algorithm.

The controller's other important outputs are 'LoadA' and 'Shift'. 'LoadA' goes high to indicate that the circuit is on an operation (add/subtract) cycle. Using this knowledge together with your glue logic, you should determine whether an operation should actually be performed based upon the multiplication algorithm being implemented. The Adder/Subtractor component always outputs the result of an operation on its inputs, but whether or not the result is saved back into the accumulator depends upon your glue logic feeding into the LD input of the accumulator, which effectively determines whether an operation is performed. A load should *only* be performed on these operation cycles, and only when an operation (add or subtract) is needed according to the algorithm you are implementing.

In sum, Robertson's multiplication algorithm consists of the following steps in the given order:

- Load multiplier and multiplicand, clear accumulator A and flip-flop F;
- Repeat **23** times the following two sub-steps (a & b):
 - a) If the rightmost bit of the multiplier is 1 then $A = A + \text{Multiplicand}$;
 - b) Shift A and multiplier 1-bit right;
- Correction step: if the sign bit of the multiplier is 1 then $A = A - \text{Multiplicand}$;
- Remember that the LSB of the multiplier is not taken into account as part of the product (you should find a way to clear multiplier[0] to '0' at the end of the multiplication process);
- The 48-bit fractional result is therefore given by A(23:0),multiplier(23:1),0.

The Test Cases:

The following test cases will be of use to you when verifying your implementations. We ask that after each iteration of the multiplier circuit you record which test cases passed/failed. We then ask that you record any products that deviated from the correct results. This will help you when theorizing about why certain implementations do not work and will also be required for you to submit in your lab 2 write-up. Keep in mind that the last digit of your result may differ if you did not zero out multiplier[0] at the end of the multiplication process.

Test Case:	Multiplier:	Multiplicand:	Product:
1	0x000006	0x600000	0x000004800000
2	0x000100	0x001000	0x000000200000
3	0x000001	0x800000	0xFFFFFFFF000000
4	0xFFFFFFFF	0xABCDEF	0x000000A86422
5	0x000002	0x800000	0xFFFFFE000000
6	0x00000A	0x00000A	0x0000000000C8
7	0x7FFFFFFF	0x000001	0x000000FFFFFE
8	0x000008	0xFEDCBA	0xFFFFFFFFFEDCBA0
9	0x000004	0xABCDEF	0xFFFFFD5E6F78
10	0xFFFFFFFF	0x123456	0xFFFFFDB9754

LogicWorks Implementation

Part 1:

Now that you understand the fundamentals of the Robertson's algorithm it is time to move onto the implementation of the first circuit. This first design will be somewhat primitive and will not take advantage of a number of hardware optimizations. Nevertheless, this design should get you thinking about possible optimizations that could be made to the multiplier circuit.

After pondering some examples of Robertson's algorithm, you should observe that there are cases where addition of two same-signed 24 bit numbers can result in overflow behavior, making a number that should be positive look like it is negative and vice versa. Perhaps a simple example can illustrate this a bit more concretely. If the two LSBs of the multiplier are '11', Robertson's algorithm will perform two consecutive addition operations, which could force the partial sum to exceed the value range of the Adder/Subtractor and accumulator (e.g., if we have a positive multiplicand, 01111, the partial sum will be 101101 after the two additions, which looks like a negative number), leading to an incorrect sign for the partial sum.

As you may have already thought about, simply increasing the size of the accumulator by one bit can resolve the overflow problem by allowing the Adder/Subtractor some additional headroom to keep track of the sign bit when an overflow occurs. For this basic design, we present you with 25-bit Adder/Subtractor hardware. We ask that you implement a design that utilizes the extra hardware we have given you to correctly generate a Shift-in value to the accumulator. Keep in mind that the accumulator and multiplicand are still 24 bits, so you will need to deduce a way to use 24 bit words as input to a 25 bit adder.

Note that while you will receive additional hardware for your Adder/Subtractor, you will not receive a 25-bit register for your accumulator. This will require that you devise a scheme to store the MSB of the Adder/Subtractor to be shifted into the accumulator. For this part of the design we ask that you familiarize yourself with the component in the Figure 1 schematic that you may not have recognized, namely, the **FF** (flip-flop) component. Flip-flops will be discussed more in depth in the latter part of the course, but for now what you need to know is that these components have the ability to hold a state value. Essentially they work by taking an input value and storing this value on a clock edge. This will become useful in the lab when you need to cause a value to be "remembered". Experiment with the basic cells in the LogicWorks library, such as a D flip-flop with the D (Data), C (Clock), and R (Reset) inputs.

As with each of the circuits you will be building we ask that you use the test cases presented above to observe some behaviors of this circuit. It was mentioned previously that extra hardware was used in this design. Think about what purpose this extra hardware might serve. Would you expect any test cases to give an incorrect product for this implementation? *Use the test cases presented above to verify your expectations. You will be expected to discuss these cases in your report.*

Part 2:

As we discussed previously, the design in Part 1 uses an $(N+1)$ 25-bit Adder/Subtractor to resolve the overflow problem. Unfortunately, such a strategy necessitates a larger Adder/Subtractor, which imposes penalties in both the hardware cost and circuit delay. Therefore, we need to rethink the hardware strategy for our circuit. After pondering the implications of using a 24-bit Adder/Subtractor, you should observe that the most natural way to provide a correct Shift-in value for the accumulator would be to perform a simple sign extension.

We ask that for the second circuit you create a Robertson's implementation that optimizes the hardware, by replacing the 25-bit Adder/Subtractor with a 24-bit Adder/Subtractor component. We ask that you retain the same design as you created in Part 1, but instead of the accumulator Shift-in value being produced from the adder, simply sign extend the resulting sum/difference.

Again, we ask that you test the cases presented above on this multiplier circuit. Record which, if any, cases did not give a correct product. If there were incorrect results produced, look at the multiplier and multiplicand for each case and theorize about why these cases did not pass. *Record the results for the test cases that did not produce the expected product. You will be expected to discuss these cases and the reasons for failure in your report.*

Part 3:

It should be clear from the test cases that although the circuit designed in Part 2 contains optimized hardware over that in Part 1, the circuit contains somewhat glaring flaws. You probably realize now that while sign extension works in the absence of overflow, when an overflow does occur then sign extension of the partial sum leads to an incorrect result due to exceeding the value range that the Adder/Subtractor can output. Obviously we need to revisit the design and create another iteration of the Robertson's circuit.

An alternative solution would be to consider that the sign of the partial sum should be the carry out bit from the Adder/Subtractor; in this case, you could simply grab the carry out of the adder and use that to load the most significant bit of the accumulator. You should immediately notice that this will not work for all cases. Try the test cases we provided to you above and describe why this case occurs, by identifying at least the first point at which the partial sum calculations falter.

Part 4:

While using the carry out resolved the issue that arose when an overflow occurred by using sign extension as the Shift-in value, you will notice that using the carry out causes the multiplier to no longer give the correct product when the multiplicand is negative (but not the largest representable negative number). To develop another strategy for forcing the correct sign bit, consider how 2's complement numbers work. One way to view the value of 2's complement numbers is to sum the positional weight of each bit except for the MSB and then subtract the positional weight of the MSB. Therefore, during the multiplication process each bit of the multiplier (except for the MSB) can be considered positive. This implies that the sign of a non-zero partial sum is identical to the sign of the multiplicand. (Notice that we don't need to worry about the final step as it is not followed up by a shift as you can see in the description of the given algorithm.)

We ask that you implement this new method of determining the Shift-in value, and comment on the results of the various test cases as was done in the previous three parts. 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:

After once again testing your multiplier, you should observe that while using the MSB of the multiplicand resolves the issues that arose from using sign extension or the carry out, some other test cases no longer give accurate results. Notice that there are only two values that the MSB of the multiplicand can take on: '0' for positive numbers, and '1' for negative numbers. Furthermore, a bit must be shifted in during every shift cycle. If the multiplicand is positive, a '0' can always be shifted in, since the partial sum after each shift must always remain positive; however, simply shifting in the MSB of a negative multiplicand will in some cases turn a partial sum of zero into the largest representable negative number. Therefore, a minor addition to your Robertson's multiplication circuit that allows it to "remember" whether or not the partial sum has become non-zero will allow you to determine what should be shifted in.

With your newfound appreciation of the subtle nuances of multiplication, you should modify your circuit from part 4 to determine the correct accumulator MSB for all stages of Robertson's multiplication. Be sure to thoroughly test your implementation, in addition ensuring the results for the test cases we have provided are correct.

Implementation Guidelines:

Up until now we have described the datapath of our multiplier, the details of Robertson's algorithm, and the controller component. As you can see from Figure 1, some signals in the datapath are marked with '?', which are left for you to figure out. Essentially, you are asked in this lab to connect the output of the controller to each component, and provide appropriate inputs for the accumulator, the Adder/Subtractor, and the controller.

The Adder/Subtractor component we provide always outputs the result of an operation on its inputs without any clock signal. However, whether or not the result is saved back into the accumulator depends upon both the value of the **LoadA** signal and the value of the LSB of your multiplier. You should take both signals into account when designing the glue logic to load the Adder/Subtractor result into the accumulator.

You will also need to be able to determine whether each operation is an addition or a subtraction so you can hook that into the **A/S** input of your Adder/Subtractor. As you remember, every step will be an addition except for the last step, which will be a subtraction. To determine what the **A/S** signal should be, you should utilize a counter that increments at every step of the multiplication, and when the last step is reached, it outputs a 1 to the **A/S** signal in order to indicate that a subtraction should occur.

Components in Provided Library (Lab2Su14.clf)

Shifter24

Inputs:

- **LD**: when set, the input connected to D23-0 is loaded into the register;
- **D23...D0**: 24-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:

- **Q23...Q0:** Output bus that shows the current value of the register.

Adder/Subtractor24

Inputs:

- **A23...A0, B23...B0:** two 24-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:

- **Q23..Q0:** the 24-bit sum of the two operands;
- **CO:** indicates whether or not there was a carry (or borrow) out from the operation performed.

Adder/Subtractor25

Inputs:

- **A24...A0, B24...B0:** two 25-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:

- **Q24..Q0:** the 25-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,...,45,47	0	0	1	0
2,4,6,8,10...,44,46	0	0	0	1
48	1	0	0	0

Part 6: Verilog Implementation

Background:

As you might recall from the in-class discussion, Hamming codes are used to implement an error correction algorithm. This algorithm can be used to detect and correct a single bit error and also can detect double bit errors. 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 1s, these bits are said to have an odd parity. If the examined bits have an even number of 1s, they are said to have even parity. The goal of 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 the Hamming(7,4) code, four bits are used to actually encode the data and three parity bits are added to the data bits to give seven bits total. In a general sense, a Hamming formula can be represented as Hamming($n+k$, k), with k representing the number of data bits and n the number of parity bits that need to be added. The number of parity bits that must be added adhere to the formula: ($k \leq 2^n - n - 1$). Thus in this example for $k=4$ data bits, it is clear that 3 parity bits must be generated in order to satisfy the formula ($4 \leq 2^3 - 3 - 1$).

Remember that the positioning of these parity bits are always located at positions that are powers of 2. Position 1 for the first parity bit, Position 2 for the second, Position 4 for the third, and so on. Each parity bit is then 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 $P_4P_2P_1$ shown below: P_1P_2 check data bit 3, P_1P_4 check data bit 5, and $P_1P_2P_4$ 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 presented in class 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 the 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 1. 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 1s, which

violates the even parity that we wanted each formula to have. For the Hamming(7, 4) formula presented in class, 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 1s. 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. If an error did occur, say in position 1, 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 1 (corresponding to the parity bit 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. Note that this is the definition of a standard Hamming code, which is what you were presented in class, and what you will be required to implement in the lab.

The Hamming codewords we will deal with in this exercise will be the Hamming(15, 11) codewords. This means the 15-bit codeword will contain 11 data bits and 4 parity bits. In order to start designing a Verilog module, you will first want to consider the other differences when dealing with 15-bits instead of 7. Of course, an additional parity bit will be necessary, but you will need to consider a number of additional differences. Recall that the parity bits will still be located at the powers of 2, meaning that positions 1, 2, 4, and 8 will be used to record the parity of the codewords. 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. The last important idea to remember is 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 we have asked you to identify:

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

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

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

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

Lab Structure:

This part of the lab will serve as an introduction to the Verilog language. This will consist of creating a simple module that can perform basic error correction using the Hamming code technique. Because you should already be familiar with the Hamming(7, 4) code from class, this lab will focus on the implementation of a Hamming(15, 11) code error corrector. The parity bits and correction bits for the Hamming(15,11) code should have similar properties to the Hamming(7,4) code, but the exact formulas for computing these values will need to be determined by you.

You will need to create a simple Verilog module that is able to take a 15 bit input and produce a 15 bit corrected output. *You will need to derive the correction formulas for the 15-bit codewords. The formulas for these correction codewords will be then implemented using Verilog.* This should be a relatively simple task as the generation of the correction bits will follow the pattern outlined above, but will use the Hamming(15, 11) codewords. Remember that the parity functions will be a simple function of the XOR gate, which is included in the Verilog bit-wise operators.

After the formulas for the correction bits are generated, the input codeword will then need to be corrected. *To correct this codeword you will use the correction bits that you implemented in Verilog. The examination of the correction bits you generated will allow identification of an error during transmission. Remember that it is possible for errors to exist in either the parity bits or data bits of a codeword. If no error was found in the transmitted codeword then the output should be the same as the input. If an error is present in the input codeword, the bit that was found to be incorrect will need to be corrected. This correction can be implemented through an examination of the correction bits, which will notify you at what position the error occurred. This newly corrected codeword should be the output of your Verilog module.*

In order to complete this task you will use the Altera Quartus II development tools, which are installed on all lab computers. The Quartus II tool can 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. We will provide 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 15-bit codewords as inputs to your Verilog module. For each input we have provided, your module will produce a corresponding 15-bit 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 Hamming code implementation is correct. This should not be too complicated of a task, and understanding the code written in the testbench will prove quite valuable when writing your own testbench modules in other parts of the course.

Implementation Deliverables

Part 1-5: As part of your submission, include a LogicWorks design file (.cct) for each part that contains a complete multiplier circuit using the components provided. The inputs loading into the multiplier and multiplicand registers should be hex keyboards and the outputs should be hex displays (both under "LogicWorks I/O" in the drop-down list of libraries). Be sure to test your circuit with several different kinds of inputs in addition to the test cases we have provided to detect problems and aid in your understanding of the advantages and/or shortcomings of each implementation.

Part 6: Please include the Verilog module that contains your Hamming(15, 11) codeword implementation. Remember that this module should have a 15-bit input and a 15-bit output. Do not include any other inputs/outputs in your module, as we will only use these two in order to adequately grade your design. Be sure to thoroughly test your design, as the testbench we have provided you is somewhat limited in its functionality.

Report Deliverables:

For parts 1-5, you should submit a report explaining your behavioral design for the multiplier, your reasoning about the glue logic and how you came to develop it. Additionally, please briefly address the following questions.

- Assume the accumulator **only** has a defect which has disabled the shifting port, while the remaining part of the circuit is working correctly (e.g., your implementation from part 1 or part 5). How would this problem impact the final result? (Hint: you can try to figure it out by setting the multiplicand to 1 and observing the value of the multiplier and the final product.)
- A fair amount of discussion in this lab has been concerned with overflow considerations for Robertson's Algorithm. It should be evident that very small multiplicands will never result in any overflows during partial sum formation. Conversely, certain multiplicands are guaranteed to deliver partial sum overflows, even if only 2 consecutive addition operations took place. Can you characterize them by letting us know the ranges of the multiplicand which would guarantee an overflow even with nothing but 2 consecutive 1s anywhere in the unsigned part of the multiplier (for example 0x003 or 0x201)? Can you characterize the ranges of the multiplicand which will guarantee the absence of an overflow for Robertson's algorithm, as per the discussion in this paragraph? Please include an answer to this question in your report.

For part 6, please include a general description of how you implemented the error correction algorithm. Also please list the functions for the Hamming(15,11) parity bit formulas that we asked you to derive. Include a brief description of what led you to these derivations. Also using the formula described above, explain what would be the maximum number of data bits you could have and still be able to correct single bit errors if you were using 6 parity bits. What would be the smallest number of data bit you could have? A brief paragraph here will suffice.