

**CSE 140L**  
**Lab Assignment 2**  
**Booth's Multiplication**  
**Due Thursday, January 30<sup>th</sup>, 4:00 PM**

## Introduction

Over the summer, you find yourself interning at NumeriCorp, a hardware company which specializes in devising hardware implementations of common arithmetic and numerical algorithms. While you are excited about the application areas that they are working in as it will give you a nice forum to practice the skills that you learned in CSE 140(L), you are concerned a bit about NumeriCorp's reputation. The gossip is that they cut corners and while they produce solutions fast and cheap, they do not always work for every corner case. Indeed, to your dismay you notice that the company has unilaterally modified slightly two's complement representation and made it into a symmetric range of  $(-2^{(n-1)}, 2^{(n-1)})$  for an n-bit number by simply declaring the smallest representable number illegal. You remember the rigor of the 140 class you had taken in your days of youth, and your love of precision and are dismayed at such redefinitions, but you plug on as the boom cycles of the 2010s have long since transmuted into a bust cycle where jobs are hard to find.

Your first assignment as an intern is to implement a two's-complement multiplier based on Booth's multiplication algorithm. In Parts 1, 2, and 3 of the **LogicWorks Implementation** section, you are asked to build a 16-bit two's complement multiplier in LogicWorks using Booth's Algorithm for non-zero inputs (zero inputs are efficiently handled upfront by being detected, obviating the need to perform more complicated multiplication algorithms). NumeriCorp provides you (below) with background training on general traits of multiplication algorithms, leading gradually into the particularities of Booth's algorithm. The bulk of the ideas and optimizations presented are fundamental and constitute the basis for Booth's algorithm which you will eventually implement.

In the last section, called **Verilog Implementation**, you will implement the same multiplier in Verilog using Altera Quartus II and Modelsim. In real life, these environments are available on the lab computers in the basement.

---

## Multiplication Algorithms

As you recall from the lecture and the handout given, a multiplication algorithm consists of a series of shift and load steps, wherein the addition steps are performed whenever the corresponding multiplier bit is 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, yet with a negative phase, i.e., a positional value of  $-2^{(n-1)}$ . An interesting issue that should be handled upfront is the issue of the size of the product of the multiplication of two n-bit two's complement numbers. As we know, an 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 including the left element, whereas the right is a soft boundary which includes numbers less than, but not equal to, the right value). With a slight simplification in the name of symmetry, we notice that n-bit numbers in the range of  $(-2^{(n-1)}, 2^{(n-1)})$  when multiplied, will result in numbers in the range of  $(-2^{(2n-2)}, 2^{(2n-2)})$ , thus fitting in a  $2n-1$  bit representation.

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 the highlights of some implementation issues that are raised by this multiplication algorithm. As each step consists of the partial sum being added to the multiplicand (after

appropriate alignment, of course) one needs to address the question of the bit width of the needed adder component. It is easy to observe that the first step of adding to the partial result will produce an  $n$ -bit sum, while the final partial sum (and product of the multiplication) which we just discussed can be  $2n-1$  bits long. While the latter part of this discussion seems to argue in favor of  $2n-1$  bit adders, one can easily notice that at any point the addition will only involve  $n$ -bit inputs, as any bits of the partial sum to the right of the actual summation will remain fixed and not affected by the operation.

While the inputs to the two's complement addition that underlies the multiplication algorithm can thus be reduced to  $n$ -bit numbers, we do know that  $n$ -bit number addition is prone to overflow in the case of an  $n$ -bit output. To preclude this overflow possibility, we will start you off in this exercise with an  $n+1$  bit adder whose inputs will have their most significant  $n^{\text{th}}$  bit replicated, effectively performing a sign extension at the input space.

Now that we've started delving into issues of implementation, it may be time to look at not only the underlying adder 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 partial sum accumulation.

While  $4n-1$  bits are certainly sufficient for multiplication, a simple reduction in the storage requirements can be effected when one notices two things. First, as the multiplier's value is shifted rightward through the multiplier register after each bit is analyzed, the most significant bits of the multiplier register become unused. Second, after each stage, the least-significant bit of the accumulator will not be modified, since the accumulator grows by one bit each cycle, and only the most significant  $n$  bits get factored into the addition (or subtraction for an MSB of 1, or no-op if a zero bit is encountered). Thus, we can shift in the LSB of the Accumulator to the newly open multiplier register space. By the time the calculation is complete, the multiplier value is completely shifted out, and the multiplier register holds the least significant  $n$  bits of the product. Figure 1 below illustrates this optimized data flow:



Figure 1: Accumulator Cannibalization of Multiplier Register

In the process of cannibalization of the multiplier, one can see that a  $(2n-1)$ -bit partial sum storage has inherited an extra bit from the cannibalized multiplier, thus turning into a  $2n$  bit accumulator. This is just as well, since as we discussed in the first paragraph of the discussion, 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 possible result of  $2^{(2n-2)}$  as the square of the largest negative number.

Of course, you realize that the corner-cutting of NumeriCorp has made this issue moot, but you keep it in the back of your mind, in the hopes of enabling you to contribute to your company, by setting them on the path of the true and righteous (and earning for yourself, of course, that long coveted promotion in these lean times).

---

## Booth's Algorithm

Now that we have the fundamentals of two's complement multiplication algorithm overviewed, we turn to the particularities of Booth's multiplication algorithm. We will start by elaborating on an opportunity for optimization which arises from the generic multiplication algorithm previously discussed.

In the generic multiplication algorithm, each step decides whether an addition/subtraction operation is needed by examining the individual bit corresponding to that multiplier position; for an  $n$ -bit number,  $n/2$  addition 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 '1's 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 '1's, starting from bit  $i$  to bit  $i+k$  (counting from the right). If we use 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:

$$0011111110 = 010000000\underline{1}0 \quad (2)$$

Given this type of transformation, what is its impact on our multiplication algorithm? In the generic algorithm, we needed an add/subtract operation for each 1 in the multiplier. On the other hand, Equation (1), and its more concrete formulation 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 1 at bit  $i$  and a **1** at bit  $i+k+1$ . Therefore, all the additions needed for bit  $i$ ,  $i+1$ , ...,  $i+k$  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 1? The discussion heretofore indicates that we need a 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 what 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. If we see a sequence of 1's 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 1's) and an addition when we are examining  $M_{i+k+1} M_{i+k} = '01'$  (the finishing point of the sequence of 1's).

One thing that perhaps you may worry about is what happens to negative 2'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 number one or more bits of 1's are in the most significant bits, thus ensuring that the last nonzero bit generated in the right-to-left sweep through the multiplier will be a 1, ensuring that the resultant representation is indeed negative. Perhaps a more concrete way to visualize this is by looking at an example, such as perhaps 11111000110 and realizing that this converts into 00001001010, both bit strings representing of course the negative 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 least significant bit,  $M_0$ . 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. This bit can be thought of as being after a binary point, thus making the representation for decimal '29', for example, be '011101.0' (plus however many leading bits may be required). Furthermore, because we need to save this bit without changing the value of the multiplier, this extra bit is stored in a D flip-flop ("F1" in Figure 1). Each time the **multiplier** is shifted right, the least significant bit is fed into this flip-flop. 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:

| Rightmost Bit | Value in F1 | Arithmetic Operation |
|---------------|-------------|----------------------|
| 0             | 0           | No operation         |
| 0             | 1           | Addition             |
| 1             | 0           | Subtraction          |
| 1             | 1           | No operation         |

Table 1: The operation determined by the two consecutive bits

## LogicWorks Implementation

### Part 1:

Below, Figure 1 shows a incomplete schematic of a Booth's Multiplier, taking 16-bit inputs and producing a 32-bit output, with all of the components laid out but with some missing glue logic which you will have to figure out, indicated by the question marks.

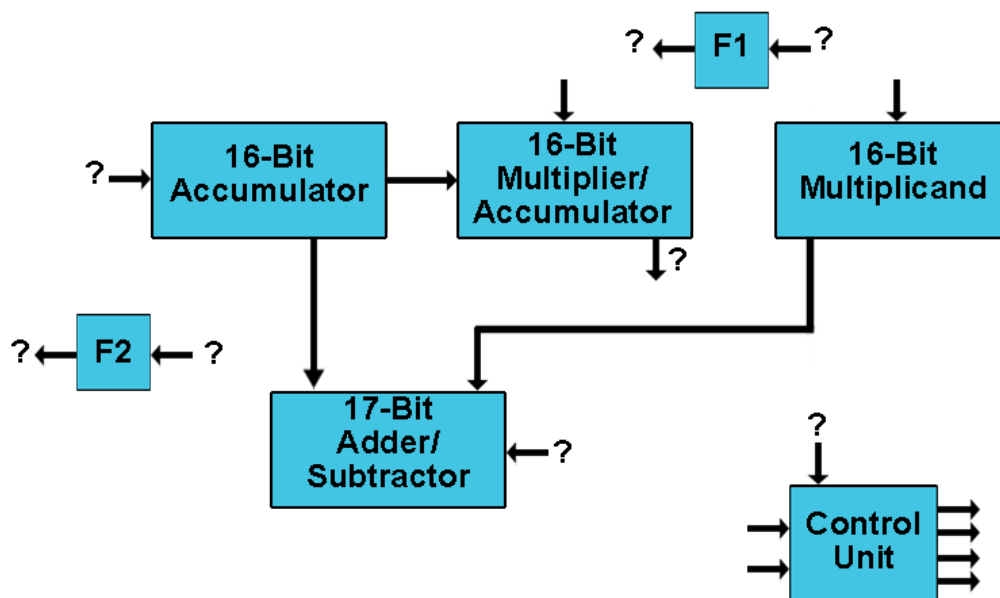


Figure 2: Incomplete Datapath for Booth's Multiplier

With the discussion in the previous section regarding the particularities of Booth's algorithm, you should be ready to complete this schematic by filling in the missing glue logic. Specifically, you must implement glue logic for the "Load" input of the accumulator, the "A/S" input of the Adder-Subtractor, and the shift-in logic for the Accumulator. You should additionally complete the interconnections among the components since as you can see a number of components are floating unconnected to the rest.

We recommend that the Flip Flop F1 (as well as F2) be implemented using the LogicWorks **D Flip Flop En wo/SQ/** component, as this component best implements the various needed functionalities for this task. You may wish to review the **Implementation Guidelines** section, which specifies details about the components we have provided for you to use, as well as general hints.

## **Part 2:**

While you quickly finish up the task of the implementation in Part 1, something seems to be bothering you as you remember from your discussion in CSE 140 that this whole discussion of overflow during addition was a much more nuanced undertaking. You decide to explore this overflow issue a bit more in depth, in light of the particularities of the consecutive operations which occur during a Booth's multiplication, and your suspicion that they have interesting implications for overflow. Your efforts bear quickly fruit and are able to do significant optimization on your Booth's multiplier from Part 1. Please implement the appropriate modifications to your circuit from Part 1 in order to effect the optimizations you have identified. You will need to modify the components we have provided in the component library, but the modifications should be straightforward. If you need to know how these components are implemented in order to carry out your modifications, you should be able to find the needed information by opening up the components in LogicWorks and studying the circuit inside.

## **Part 3:**

Now that you are going full speed and establishing a reputation for yourself as the young genius of the company, you realize it is time for you to deliver the final *coup de grace* to all the old-timers, impress them with your capabilities, and quickly rise to the position of CEO and establish NumeriCorp as the unquestioned master of the world, if not the universe, with you at the helm of course. You decide that you need to deal with this pesky limitation of NumeriCorp's two's complement representation, which forced all the salesmen of your company to quickly run for the exit when a competitor started mumbling something about large negative numbers. Namely, it is time to make NumeriCorp a full service company capable of representing and manipulating all (non-zero) two's complement numbers, including  $-2^{(n-1)}$ .

Now that you have this lean and mean multiplier from Part 2, life is going to be a bit challenging since including the largest representable negative number in your representation seem to break your algorithm. But since you have clearly identified the causes of the problem, you persevere in your quest to fixing things up. With two's complement multiplication, you soon realize that additions that overflow can be restored to health if you have extra headroom to accommodate the result (and of course assuming you can figure out what the correct sign bit should be). You soon realize that the extra headroom to accommodate the restoration of the result will magically appear as a result of the right shift. With this silver lining of hope you decide to analyze what the sign bit should be and to determine a logic implementation to feed your sign bit.

Armed with the result of your analysis, you should modify your circuit from Part 2 to include the logic implementation satisfying the requirements identified in the last paragraph for determining the correct Accumulator MSB at all stages of Booth's multiplication. Your additional logic should comfortably fit within the constraints of 4 gates and a flip-flop.

## **Implementation Guidelines:**

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 (and extra flip-flop) 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. The rules for this determination have been more or less spelled out for you earlier in the lab.

The components you will use to implement your circuit can be found in the native LogicWorks libraries and the Component Library which you can download from moodle. A description of each component in

the Library can be found in the **Component Library** section of this lab, after **Phase IV** and before the **Implementation Deliverables** section.

### 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 flip-flops 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, Booth's multiplication algorithm consists of the following steps in the given order:

1. Load multiplier and multiplicand, clear accumulator A and flip-flops F1 and F2;
2. Repeat 16 times the following two sub-steps (a & b):
  1. Update the accumulator:
    - If the LSB of the multiplier register is 0 and  $F1 = 1$ ,  $A = A + \text{Multiplicand}$
    - If the LSB of the multiplier register is 1 and  $F1 = 0$ ,  $A = A - \text{Multiplicand}$
    - /\* Else, don't update A \*/
  2. Shift A and multiplier 1-bit rightward. The LSB of the accumulator should flow into the MSB of the multiplier, and the LSB of the multiplier should flow into F1 as shown in Figure 2. (The MSB of the Accumulator will be updated at this stage, but its behavior will differ between Parts 1, 2, and 3).
3. The 31-bit result will appear as the concatenation of A(14:0) and Multiplier(15:0).

Note that most of the behavior of flip-flop F2 is left unspecified. This is because it will not remain the same for all parts of this lab. We will leave it to you to figure out how, when, and whether it is needed in each part. Nonetheless, we recommend you use the **D Flip Flop En wo/SQ/** as the flip flops F1 and F2, as they contain inputs which enable efficient input logic for the required functionality.

### Testing Your Solution

To make sure you have implemented your circuits correctly, you should perform test multiplications. Your test inputs should show a high degree of variability to test regular inputs as well as corner cases. We recommend testing multiplier/multiplicand pairs characterized by alternating 1s and 0s, as well as those which contain long runs of 1s and 0s. Also test multiplications between inputs of the same sign and of opposite signs.

Good examples of input pairs for all three parts might be:  $2AAA * 5555$ ;  $FF00 * 00FF$ ;  $FF00 * FF00$ . Part 3 explores additional special cases of the multiplier, so you will have to use special test cases to test the extra functionalities.

## **Provided Component Library**

This is the library of LogicWorks components we have provided to enable you to complete parts 1-3 of the lab.

### **Shifter16**

*Inputs:*

- **LD:** when set, the input connected to D15-0 is loaded into the register;
- **D15...D0:** 16-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:*

- **Q15...Q0:** Output bus that shows the current value of the register.

### **Adder-Subtractor17**

*Inputs:*

- **A16...A0, B16...B0:** two 17-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:*

- **Q16..Q0:** the 17-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:

| <b>Cycle</b>                                       | <b>End</b> | <b>Init</b> | <b>LoadA</b> | <b>Shift</b> |
|----------------------------------------------------|------------|-------------|--------------|--------------|
| 0                                                  | 0          | 1           | 0            | 0            |
| 1,3,5,7,9,11,13,15,17,19,21,<br>23, 25, 27, 29, 31 | 0          | 0           | 1            | 0            |
| 2,4,6,8,10,12,14,16,18,20,22,<br>24,26,28,30,32    | 0          | 0           | 0            | 1            |
| 33                                                 | 1          | 0           | 0            | 0            |

## **Native Component Libraries**

This section contains information about the components you will need from the native LogicWorks libraries.

### **D Flip Flop En wo/SQ/**

#### *Inputs:*

- D: The value Q takes on the next enabled positive clock edge
- En: Enables a positive clock edge to update the value of Q
- C: The clock, which, when it transitions from 0 to 1, updates the value of Q (if enabled)
- R: When R = 1, Q is set to 0
  - Note: Behavior does not depend on C or En
  - Note: Input is complemented before input, so a value of “0” on the pin is interpreted as a value of “1” in the circuit.

#### *Outputs*

- Q: The value currently stored in the Flip Flop

---

## **Verilog Implementation**

You realize that your elation regarding your status within the company, as well as the status of the company within the universe, was a bit premature, given that you have implemented your design in LogicWorks. Nobody in the real world uses LogicWorks to design hardware in a commercial enterprise. Thus, you decide that you must re-implement the more robust two's-complement Booth's multiplier you developed in Part 3, but this time in Verilog instead of LogicWorks. Doing so will require translating the behavior given in the Component Library section of Part 1 into Verilog, hooking the components together in an identical way as you did in Part 3, and thoroughly testing your results in Modelsim in order to verify the correctness of your implementation. (If you were unable to complete Part 3, you may instead re-implement your solution to Part 1 or 2, for partial credit.)

### **Translating Logicworks to Verilog:**

While Logicworks provides a solid interface for designing structural schematics for circuit implementations, nowadays most hardware designers use behavioral semantics which can be synthesized into physical designs using software tools. Behavioral descriptions can generally be expressed through Hardware Description Languages (HDLs), an example of which is Verilog. HDLs enable designers to describe hardware using very similar programming language semantics that you have learned in your introductory CSE courses. Being veteran students within the UCSD computer science department, we expect that Verilog may prove to be even more convenient than *Logicworks*. For this part, we will provide you with three Verilog modules, *MultiplierTest.v*, *Multiplier.v*, and *Controller.v*. The descriptions of these modules are below:

*MultiplierTest.v*: The testbench that you can use to test the functionality of your multiplier in ModelSim. You do not have to modify this file, but you may want to change the test cases in order to more thoroughly test your design.

*Multiplier.v*: This will be where your glue logic should go. The module should have the following definition:

```

module Multiplier#(
    parameter width = 16
)
(
    input CLK,
    input [width-1:0] Multiplicand,
    input [width-1:0] Multiplier,
    input Begin,

    output reg [(2*width)-1:0] Product
);

```

Please do not modify the input or output names, as we will use them to test your design.

*Controller.v*: This controller is identical in functionality to the Controller described in the Component Library in Part 1. Feel free to look at the source code in order to acclimate yourself to Verilog and garner inspiration for writing your own modules.

In order to complete the multiplier circuit, we ask for you to implement the Shifter16 and AdderSubtractor modules yourself. Please refer to the Component Library for the specific functionality, and try playing with the components in Logicworks so you can more accurately perform the translation into Verilog. Keep in mind that the implementation of these two modules will be different, as the Shifter16 will end up being a sequential module while the AdderSubtractor will be a combinational module. Once you create these modules, you will need to instantiate them and implement the relevant glue logic in the *Multiplier.v* module in order to obtain the correct multiplier functionality.

---

## Implementation Deliverables

**LogicWorks Implementation:** For each of the first three parts of the lab, include a LogicWorks design file (.cct) with your solution to that part. The inputs loading into the multiplier and multiplicand registers in each circuit should be hex keyboards and the outputs should be hex displays (these components can be found under "LogicWorks I/O" in the drop-down list of libraries). Please name your circuits "Part<part#>\_<firstname>\_<lastname>.cct", with "part#" corresponding to the part of the LogicWorks implementation the circuit is associated with, and "firstname" and "lastname" substituted with your first and last name, respectively.

**Verilog Implementation:** Please include all relevant verilog files. Do not change the inputs into the *Multiplier.v* module, as we will need them in order to adequately grade your design. Be sure to thoroughly test your design as well, especially with overflow cases. Also, please synthesize your Multiplier circuit and put a screenshot of the resulting RTL View (Tools → Netlist Viewers → RTL Viewer). Please include your first and last name in your submitted filenames as outlined above.

---

## Report Deliverables

For Part 1, explain your behavioral design for the multiplier, your reasoning about the glue logic and how you came to develop it. Specifically, explain your derivation of logic determining the Add/Subtract signal, the “LD” input of the Accumulator, and the shift-in input of the Accumulator.

For Part 2, explain the optimizations you were able to make. Why were these optimizations possible?

For Part 3, please answer the following questions:

1. It is obvious that introducing the largest negative number means that your Part 2 circuit will no longer produce the correct output for all inputs. However, this does not mean that all input combinations which use the largest negative number will be incorrect. Can you distinguish between input combinations making use of the largest negative number that will produce the correct answer from those which will produce an incorrect answer?
2. Outline the rules that govern the behavior of the sign bit of the partial product over the course of a Booth's multiplication. How did you translate these rules into the corresponding logic in your circuit? What does this reveal about the nature of the error occurring?
3. At the beginning of the report, we explained that NumeriCorp handles the input “0” differently than other numbers. Which of the three circuits above can correctly handle an input of 0? Does it matter whether 0 is the multiplier or the multiplicand? Why does each circuit behave this way?

For your Verilog Implementation, please describe your strategy for translating your Logicworks schematic into Verilog. Furthermore, describe the similarities and differences between your generated Verilog schematic and your Logicworks Schematic.