

CSE 140L
Lab Assignment 2
Booth's Multiplication and Non-Restoring Division
Due Sunday, October 22nd, 11:59 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 11-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 get started using the tools Altera Quartus II and Modelsim to practice Verilog. You will implement a number of basic digital components using *structural* descriptions. Afterwards, you will write a division algorithm using *behavioral* Verilog. This way, you'll get a taste of the two flavors of Verilog modeling. These tools are available on the lab computers in the basement.

Section 1: LogicWorks Implementation of Multiplication Algorithms

Basic Multiplication Algorithm: Robertson's

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 needs no more than three n -bit registers in total 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 to be shifted in for storage. 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. **Figure 1** below illustrates this optimized data flow between the **accumulator** and **multiplier** registers:



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 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 $[-2^{(2n-2)}, 2^{(2n-2)}]$. (Note that the square bracket '[' indicates a hard boundary, including the right element, whereas the parenthesis '(' is a soft boundary that includes numbers greater than, but not equal to, the left value.) This necessitates a truly $2n$ -bit representation to account for the somewhat exceptional case 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).

There is one more implementation detail we must consider. 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 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 constant 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^{th} bit replicated, effectively performing a sign extension at the input space.

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 (or subtraction, in the case of the MSB) 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) implies that a standard binary number can be converted into a **three-valued** (0, 1, and 1) 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 illustration in Equation (2), imply 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 as we scan from right to left (i.e. the sequence '10' as it appears written in left-to-right form). Similarly, we need a 1 for the finishing point of the sequence of 1's, that is, when we encounter a 1->0 transition (i.e. '01'). 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 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 two's complement negative number, one or more consecutive bits of 1's are in the most significant place. Since there is no 1->0 transition at the most significant bit, the last nonzero bit in the right-to-left sweep through the encoded multiplier will be a 1. This guarantees that the resultant representation is indeed negative. For example, the two's complement number 11111000110 converts

into 00001001010. Both bit strings represent 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, for example resulting in a representation for decimal '29' as '011101.0' (plus however many leading bits may be required). This bit is not part of the value of the multiplier, so we need a storage element to keep track of it outside of the multiplier register. It will be stored in a D flip-flop ("F1" in **Figure 2**). Each time the **multiplier** is shifted right by a single bit, 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, as shown in **Table 1**. Keep in mind that though we are inspecting two bits, we only shift by a single bit between each operation, i.e. each bit is examined twice (once in the M_0 position, then once in the M_{-1} position), leading to exactly n inspections of the bit pairs.

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

Table 1: Booth's multiplication operation based on the LSB of the multiplier register and flip-flop F1

The functions determined by **Table 1** can be broken down between the Rightmost Bit and the value in F1 as follows: If the Rightmost Bit is a 0, then the circuit will be performing an addition. If the Rightmost Bit is a 1, then the circuit will be performing a subtraction. Whether this operation is actually saved by being committed to the accumulator register is determined by a single 2-input gate with the Rightmost Bit and the value in F1 as inputs. It is left to you to figure out what gate will perform the necessary function. Considering the logic in this way will help with your implementation, detailed in the next section.

LogicWorks Implementation

Part 1:

Figure 2 on the next page shows an incomplete schematic of a Booth's Multiplier with 11-bit inputs and a 22-bit output. The output is contained within the concatenation of the Accumulator with the Multiplier/Accumulator registers. The arrows pointing down into the 11-bit Multiplicand and the 11-bit Multiplier/Accumulator registers indicate where the inputs to the algorithm are initially loaded. The Adder/Subtractor module is also shown with its inputs, but the output is not shown. Its output, along with the other glue logic indicated by the floating arrows with question marks, must be implemented and connected by you. More details about the Control Unit, and what its inputs/outputs are used for, can be found in the section **Implementation Guidelines** that follows **Part 3**.

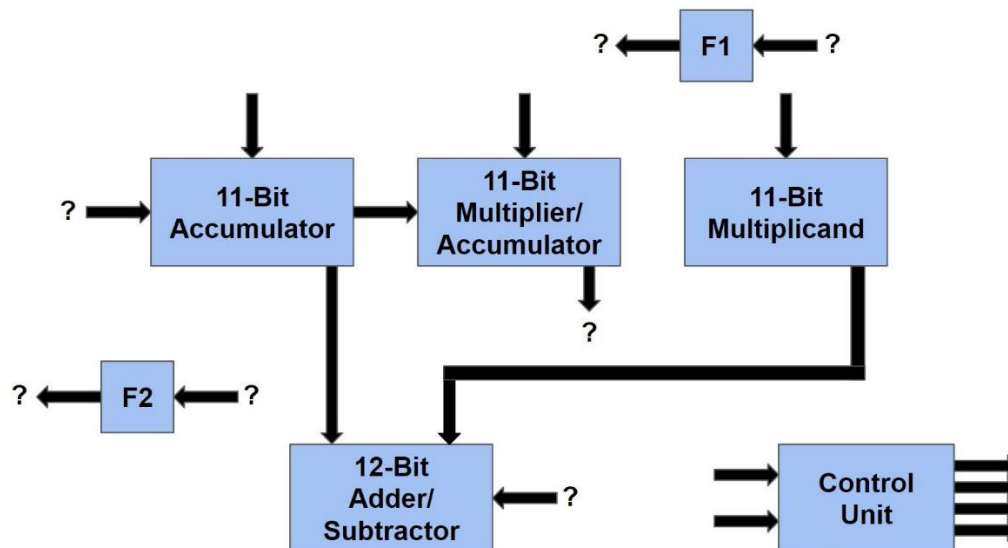


Figure 2: Incomplete Datapath for Booth's Multiplier

You should be ready to complete this schematic by filling in the missing glue logic. Specifically, you must implement glue logic determining the following:

- When the Accumulator register should “Load” a new value
- Whether the Adder-Subtractor should add or subtract
- What bit should be shifted into 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, and not all inputs/outputs are shown.

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 is sufficient for the various needed functionalities for this task without cluttering the design with unnecessary extra functionalities. 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 the issue of overflow during addition in the original multiplication algorithm was a much more nuanced undertaking. You decide to explore this overflow issue a bit more in depth while considering the particularities of the consecutive operations which occur during a Booth's multiplication. Your efforts quickly bear fruit and you are able to do an extra optimization on your Booth's multiplier from Part 1. Please implement the appropriate optimization to your circuit. You will need to modify a component we have provided in the component library, but the modifications should be straightforward. If you need to know how the components are implemented to carry out your modifications, you should be able to find the needed information by opening the components in *LogicWorks* and studying the circuit inside.

Hint: Consider the order of additions and subtractions in both the original multiplication algorithm and in Booth's algorithm. Is there a pattern in the sequence of additions/subtractions that Booth's algorithm guarantees that the original multiplication algorithm doesn't?

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 seems 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 overflowing additions 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 sliver of hope you decide to analyze what the sign bit should be and to determine a logic implementation to feed your sign bit. You quickly realize that simple sign extension of the accumulator when shifting will not be sufficient.

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 Control Unit:

The control unit (also called a 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**. As you must have heard, the basic multiplication operation implementation is colloquially referred to as an Add/Shift algorithm. As the name denotes, the **Shift** signal controls the latter step of the algorithm. The former step of "Add" (or perhaps more accurately called "Add/Subtract") is controlled by the **LoadA** signal, even though the name of the signal is to be interpreted a bit more broadly. For starters, as we know, the actual operation to be executed in Booth's multiplication alternates between subtraction and addition (hopefully with long bouts of doing nothing in between). In this particular implementation, for simplicity, we will forgo luxuriating in laziness and instead have the algorithm execute an operation every cycle. To ensure correct execution of the overall algorithm, we will end up throwing away (i.e. **not** load) the result of the addition/subtraction in many of the cases. All **LoadA** denotes is that we are on an operation (i.e. an "Add/Subtract") cycle; It is up to you, as discussed at the end of the description of Booth's algorithm, to determine the gate whose output, in conjunction with the **LoadA** signal indicating that we are on an "Add/Subtract" cycle, will determine whether the result of the addition/subtraction is loaded into the accumulator register.

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 11 times the following two sub-steps:
 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 21-bit result will appear as the concatenation of A(9:0) and Multiplier(10:0)

The Adder-Subtractor:

The **Adder-Subtractor** component we provide has no clock input, meaning it is purely a *combinational* logic component. This means that the sum/difference of the inputs always shows on the outputs with minimal delay. However, whether the result is committed back into the accumulator depends upon the value of the **LoadA** signal from the control unit, the value of the LSB of your multiplier (the Rightmost Bit in **Table 1**), and the value in the flip flop F1. You should take all three signals into account when designing the glue logic to load the adder/subtractor result into the accumulator, as described in the above description of the **Control Unit**.

You will also need to be able to determine whether each operation is an addition or a subtraction. The **A/S** input of your adder-subtractor dictates whether the adder-subtractor will perform an addition or a subtraction (more details about the inputs/outputs can be found in the **Provided Component Library** section). The rules for this determination have been largely spelled out for you in **Table 1**.

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 Piazza or the course website. A description of each component in the Library can be found in the **Component Library** section of this lab.

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 circuit 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. Part 3 explores additional special cases of the multiplier, so you will have to use special test cases to test the extra functionalities.

Component Libraries

Provided Component Library:

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

Shifter11: May be used for both the accumulator register and the multiplier register.

Inputs:

- **LD:** when set, the input connected to D11-D0 is loaded into the register;
- **D10...D0:** 11-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:

- **Q10...Q0:** Output bus that shows the current value of the register.

Adder-Subtractor12

Inputs:

- **A11...A0, B11...B0:** two 12-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:

- **Q11...Q0:** the 12-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. A '1' on this signal means 'initialize';
- **LoadA:** a '1' on this signal indicates that the accumulator should load the value of the output of the adder-subtractor if the algorithm (explained by **Table 1**) indicates it should;
- **Shift:** a '1' on this signal indicates that the contents of the accumulator and the multiplier should be shifted 1-bit to the right.
- The outputs of the controller on each cycle will be as follows:

Cycle	End	Init	LoadA	Shift
0	0	1	0	0
1,3,5,7,9,11,13,15,17,19,21	0	0	1	0
2,4,6,8,10,12,14,16,18,20,22	0	0	0	1
23	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 flip flop 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:** Reset pin. When R = 0, Q is set to 0.
 - Note: Input is asynchronous and does not depend on C and EN;
 - Note: There is a small circle on this input, indicating that the semantics are complemented. A value of '0' on this pin means, "Reset pin is active," and the content of the flip flop is immediately overridden to hold '0'.

Outputs

- **Q:** The value currently stored in the Flip Flop.

Section 2: Verilog Implementations of Booth's Multiplication and Non-Restoring Division

There are two main approaches to implementing a digital circuit using Verilog: structural and behavioral implementations. In a structural approach, the components, modules, and their connections are explicitly declared. For the implementation of Booth's with Verilog in this context, a structural implementation is the natural choice since you already have a structural depiction from your LogicWorks implementation. Please do a structural implementation in Verilog for the Booth's multiplier.

The other Verilog option is a behavioral implementation. Nowadays, most hardware designers use behavioral semantics which can be synthesized into physical designs using software tools. A Hardware Description Language (HDL) like Verilog enables designers to describe hardware using programming language semantics quite similar to those that you have learned in your introductory CSE courses. Being veteran students within the UCSD computer science and electrical engineering departments, we expect that Verilog may prove to be even more convenient for you than LogicWorks. For the division portion of this exercise, we ask that you use behavioral Verilog to implement the division algorithm.

Booth's Algorithm

You realize that your elation regarding your status within NumeriCorp, as well as the status of the company within the engineering 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 of Section 1, but this time in Verilog instead of LogicWorks. Doing so will require translating the behavior given in the **Component Library** section in the LogicWorks implementation 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 in Verilog your solution to Part 1 or 2 of Section 1, for partial credit.)

Translating Logicworks to Verilog:

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 = 11
)
(
    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 Section 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 Shifter11 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 Shifter11 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.

Division Algorithm

Next, you are asked to implement a division algorithm for integers using Verilog. For simplicity, the inputs will be interpreted as 2's complement signed integers with only positive values constituting valid inputs. The basic principle is not unlike normal long division done by hand. First, we take the **divisor** and align it with the leftmost (most significant) bits of the **dividend**, and only consider the bits of the dividend that are aligned with the divisor. If the divisor is larger than the number formed by the aligned bits of the dividend, then we mark the corresponding bit of the quotient as a 0. Instead of then shifting the divisor one bit to the right (effectively halving the value) as we traditionally do by hand, we can shift the dividend one bit to the left (doubling the value of the dividend) before re-evaluating. If the divisor is instead smaller than the aligned bits of the dividend, then we note that the corresponding bit of the quotient is a 1, and subtract the divisor from the dividend to create a partial remainder. This process is repeated, with the new partial remainder taking the place of the dividend, until we have considered the least significant bit of the partial remainder. For simplicity in implementation, we can initialize the partial

remainder to equal the dividend so that all operations are performed on the partial remainder register. Any value left over in the partial remainder at the end is recorded as the **remainder** of the division.

In order to determine if the divisor is greater or less than the number formed by the aligned bits of the dividend, we can perform a preliminary subtraction. If the result is positive, then it can be concluded that the divisor was indeed smaller, and our subtraction step to form the partial remainder is already done! However, if the result is negative, it indicates that the divisor was less than the dividend, and we need to undo the subtraction, shift the dividend (actually by the now more properly named partial remainder) to the left by one bit, then re-evaluate. There are two varieties of division algorithms to handle the 'undo' step: Restoring and Non-Restoring. You will implement the non-restoring version because of its speed advantages. But first, to better understand the non-restoring algorithm, we will look at the restoring version.

Restoring Division Algorithm:

In the restoring algorithm, when the new partial remainder is negative after the subtraction of the divisor from the partial remainder, an addition of the divisor is then performed to restore the partial remainder to its previous value. Then the algorithm can continue as usual with a shift and another preliminary subtraction.

Non-Restoring Division Algorithm:

The non-restoring variant is based on the observation that the subtraction of the divisor resulting in a negative partial remainder, which is followed by a restoring addition before a shift of the partial remainder to set things up for the next subtraction (as in the restoring algorithm) can be combined into just a shift of the negative partial remainder followed by an *addition* of the divisor (instead of performing a subtraction next). This is because if we shift the partial remainder left without a restoring addition, it is as if we doubled the divisor deficit before we subtracted it, in addition to doubling the partial remainder as we normally do. Adding the divisor on the next step cancels half of the previous double subtraction, resulting in the same value as the single subtraction from the partial remainder that the restoring algorithm would have given. Thus, instead of a restoring addition when the partial remainder is negative, the incorrect result can be retained and the restoring step can be omitted, since doing an addition on the next step instead of a subtraction will compensate. No changes whatsoever need to be made for the case of a positive partial remainder. We can now see that the restoring algorithm can be altered to take advantage of this observation as follows:

1. After a subtraction/addition of the divisor:
 - If the partial remainder is positive, we keep the value in the partial remainder, shift it one bit to the left, and mark the corresponding bit of the quotient as a 1. The next step for computing the next partial remainder will be a subtraction of the divisor.
 - If the partial remainder is negative, we still keep the value in the partial remainder and shift it one bit to the left. We mark the corresponding bit of the quotient as a 0. The next step for computing the next partial remainder will be an addition of the divisor.
2. Perform either an addition or subtraction of the divisor, as outlined in the previous step.
3. Repeat until the least significant bit of the dividend is examined.

There is one caveat to the non-restoring algorithm as outlined above. If the partial remainder is negative after examining the last, least significant bit of the dividend, then a single restoring step of adding the divisor back is necessary. This is because there won't be another addition on the next bit to compensate.

Implementation in Verilog

We will provide you with two incomplete Verilog modules, *Division.v* and the testbench *division_tb.v*. Your job will be to complete *division.v* with your implementation of the non-restoring division algorithm, and to add sufficient test cases to *division_tb.v* to thoroughly test it. Do not change the definitions of the two modules, including any inputs and outputs, as we will use them to test your design. However, you may add any *wires*, *regs*, and logic that you deem necessary.

Provided in *Division.v* is a template of sorts. It will define some parameters, wires, regs, and logic blocks for you to get started. Feel free to change anything except for the inputs/outputs. You will primarily be interested in the *always @(*)* and *always @(posedge CLK)* blocks, as this is where the bulk of your logic will be placed. These blocks are separating the combinational logic (*@(*)*) and the sequential logic (*@(posedge CLK)*).

The module definition for *division.v* is as follows:

```
module division #(DIVIDEND_LEN=16) (  
    input          start,  
    input          CLK,  
    input [DIVIDEND_LEN-1:0] divisor,  
    input [DIVIDEND_LEN-1:0] dividend,  
    input [DIVIDEND_LEN-1:0] divisor_len,  
  
    output reg      done,  
    output reg [DIVIDEND_LEN-1:0] quotient,  
    output reg [DIVIDEND_LEN-1:0] remainder  
);
```

Below is a description of each of the inputs and outputs, and how these dictate the behavior of the circuit. Note the parameter *DIVIDEND_LEN*, which will determine the maximum length of the dividend and divisor.

Inputs:

- *start* – When set to 1, the algorithm will begin executing. The *start* signal does not need to remain at 1 for the algorithm to continue executing.
- *CLK* – External clock input.
- *divisor* – The positive divisor. If it uses fewer bits than *DIVIDEND_LEN*, then the bits representing the divisor will be right-aligned (leading zeros, but no trailing zeros). Because it is a positive number represented in 2's complement notation, there must be at least one leading zero.
- *dividend* – The positive dividend. Will also be right-aligned. As with the divisor, there must be at least one leading zero.
- *divisor_len* – The exact number of bits needed to express the positive divisor. This should include **exactly one** leading zero of the divisor. For example, if the *DIVIDEND_LEN* = 6, and the divisor input is 000111, the *divisor_len* input should be 4 (000100). This is used to trim the leading zeros in the divisor input.

Outputs:

- *done* – When the algorithm is done computing, it will output *done* = '1'. Else, output *done* = '0'.

- *quotient* – The quotient. It displays the correct value when done = '1'.
- *remainder* – The remainder. It displays the correct value when done = '1'.

Implementation Guidelines

- The *divisor_len* input must account for a single leading zero in the divisor input. This is to prevent any overflow issues when aligning the divisor with the dividend in the initial setup.
- The leftmost bit in the partial remainder will be shifted out after each left-shift operation. This value may be discarded freely. Thus, the partial remainder register only needs to be of the size specified by DIVIDEND_LEN.
- To keep the values and the behavioral description of the addition/subtraction consistent, the divisor may be left aligned and placed into a register of size DIVIDEND_LEN as well. This register can be aligned exactly with the partial remainder register, as they would then be the same size. The divisor register will stay constant, while the partial remainder register will be shifted. This, however, is not essential: The divisor bits may be aligned with the partial remainder regardless of size, but this implementation will require more bit juggling on your part.
- Inside the *division.v* template are two *always* blocks. The *always @(*)* block is used to calculate the values that will be registered on the next clock cycle. The *always @(posedge CLK)* block is used to register the values calculated inside the *always @(*)* block. This separates the combinational logic from the sequential logic. It is good practice to keep these two types of logic separate in your Verilog code.
- It is possible, although not necessary, for the addition/subtraction of the divisor and the following shift of the partial remainder to be combined into a single step, executed in a single clock cycle.

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<1,2 or 3>_<first initial+last name>_<partner's first initial+last name>.cct". For example, if the team members' names were John Doe and Darren Eck, the circuit for Part 3 would be titled "Part3_JDoe_DEck.cct". If you did not work with a partner, then the partner's last name is omitted, of course.

Verilog Implementation: Please include all relevant Verilog files. Do not change the inputs in the *Multiplier.v* or the *Division.v* modules, as we will need them in order to adequately grade your design. Be sure to thoroughly test your designs as well, especially with overflow cases. Include an annotated screenshot of the ModelSim waveform showing correct operation. Also, synthesize your Multiplier and Division circuits and put a screenshot of the resulting RTL View (Tools → Netlist Viewers → RTL Viewer). The .v files do not need to be renamed, as it is good practice to keep the module name and filename the same in Verilog.

Include all files, including a PDF of your lab write-up, inside a .zip file titled "Lab2_<first initial+last name>_<partner's first initial+last name>.zip". Upload the final .zip file to Gradescope.

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 optimization(s) 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. As asked in Part 3, give an example of a pair of inputs (besides where both are $-2^{(n-1)}$) where a sign-extension during the shift operation will result in an incorrect value.
3. 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?
4. 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 of the multiplier, 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. For your implementation of the non-restoring division algorithm, please describe how you specified the behavioral description of your circuit. Does the synthesized circuit look like how you imagined it would?