Assignment 3

Multiplier Circuits

Justin Clark

Introduction

The purpose of this assignment was to use Logisim to design circuits that multiply two binary numbers together. Logisim is circuit design software that has everything from simple gates to more complicated circuits.

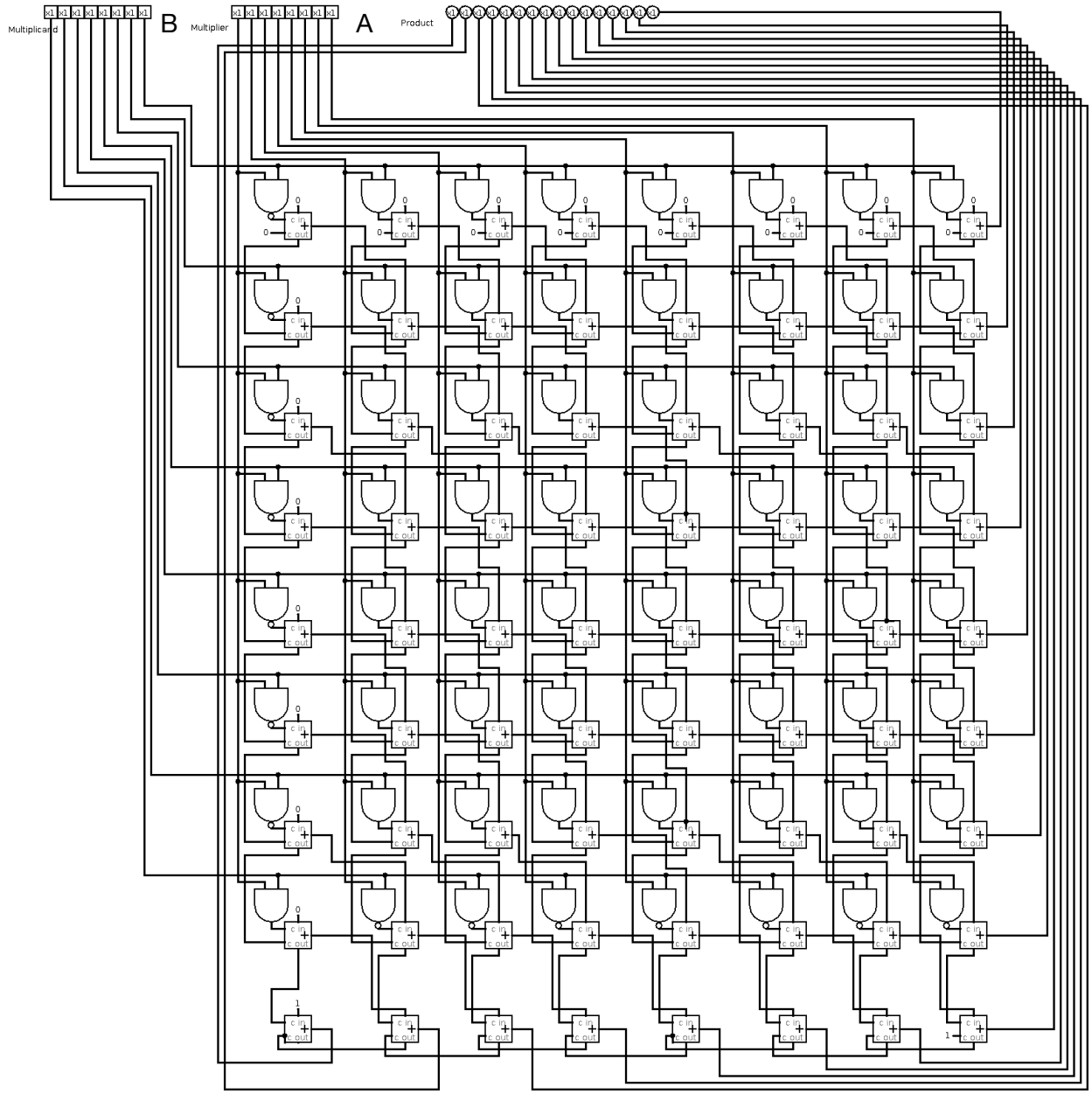
In Part A of this project, we were to design a circuit that multiplies two 8-bit 2s complement binary numbers. The most high-level (predesigned) circuit in this implementation is an adder. All other gates are NAND and AND gates.

In Part B, we designed a IEEE 754 single precision floating point multiplier. There are more high-level circuits used. For example, a predesigned multiplier circuit is used which has even more functionality than all of Part A. The real heart of Part B came down to renormalizing exponents and mantissas of the product.

Part A: Signed 8 Bit 2's Complement Multiplier

APPROACH

My approach was to design a circuit based on a "Baugh-Wooley" multiplier which is one way of directly handling signed bits (specifically in 2's complement format) as discussed in section 3.2.2.2 of [1]. The Baugh-Wooley multiplier is a modification of a Wallace Tree which is series of *carry save adders* (CSA) followed by a *carry lookahead adder* (CLA). My circuit design is below:



The output for each of the first 8 rows of adders (with a corresponding NAND or AND gate) represents a partial product. The carryout of each adder is the input of the the adder

calculating the next significant bit for the next partial product. A more clear way of creating this circuit could have been to shift gates in subsequent rows one to the left in order to visualize how the output corresponds to the bit significance of the final product, but there was a tradeoff to be made in terms of space and clear wiring.

One might notice that the input for the adders corresponding to the most significant bit for the first $n-1$ partial products are calculated with a NAND gate. The first $n-1$ bits of the last partial products are identical. This is the where the numerical adjustment takes place as shown in figure 3.25 of [1].

The way the final 16-bit product is actually calculated comes in 2 parts. First, the output of the of the least significant bit each partial product represents the corresponding bit in the final product. e.g. the output of $PP0$ represents $P0$ and so on. This gives us the first 8 digits of the product. The seconds part is shown on the bottom row, which represents a carry lookahead adder. The carryout of each adder is the input of the adder corresponding to the next significant bit (moving from right to left). The output of each of these 8 adders represents the most significant 8 bits of the 16 bit product.

Example results of this implementation are in the following table:

A (Binary)	B (Binary)	Product (Binary)	A (Decimal)	B (Decimal)	Product (Decimal)
10001110	10000110	0011011001010100	-114	-122	13908
10011001	00010100	1111011111110100	-103	20	-2060
00000000	01111111	0000000000000000	0	127	0
01110000	00001111	0000011010010000	112	15	1680
01111011	10000100	1100010001101100	123	-124	-15252
10101010	10101010	0001110011100100	-86	-86	7396
00000011	00000111	0000000000010101	3	7	21

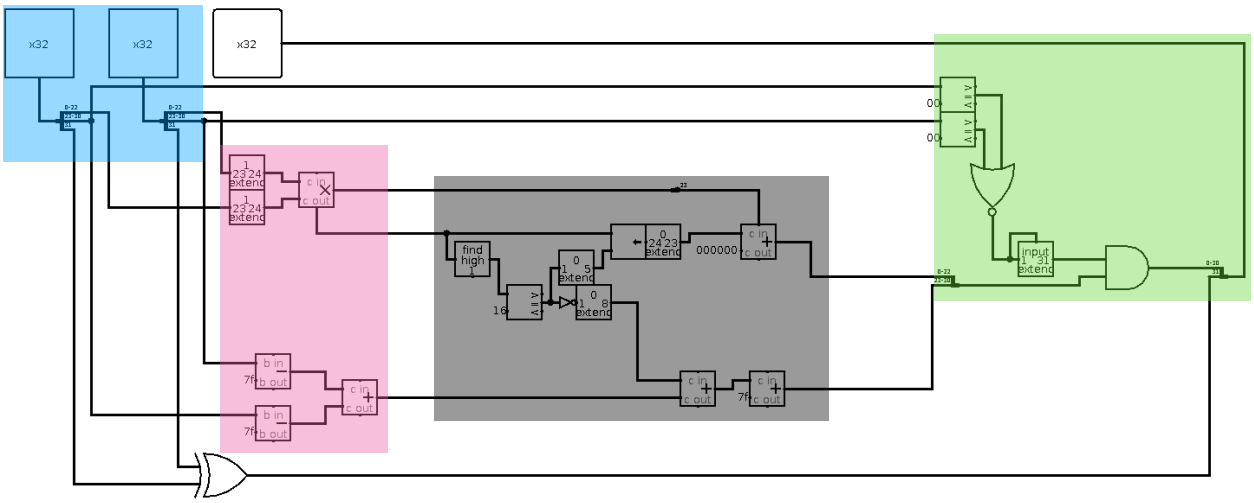
Table 1

This problem could have been solved another way in place of the Baugh-Wooley multiplier. The other method is called Booth's algorithm which is one of the more popular algorithms in practice. However, as discussed in [2], the performance increase of Booth's algorithm (due to decreased number of partial products) has plateaued as array multipliers have fallen out of style. The authors also discuss how the Baugh-Wooley multiplier can lead to better performance in a "High Performance Multiplier (HPM) tree".

Overall, I found the Baugh-Wooley algorithm to be much more intuitive due to its very clear iterative approach to multiplying numbers. Although some performance is possibly sacrificed when compared to Booth's Algorithm (or some variant), the simplicity is hard to beat.

Part B: IEEE 754 Single Precision Multiplier

In this part, the circuit had to multiply two signed floating point numbers. There are 4 steps necessary in multiplying these numbers. First, separate each number into three parts: the sign bit b_{31}, the exponent b_{23-30}, and the mantissa b_{0-22}. The second step is to multiply the mantissa bits and add the exponents. Third, renormalize the exponent and mantissa based on the results of the multiplication of the mantissas. Finally, recombine the bits (including the XOR of the sign bits) into IEEE 754 format. My circuit design is shown in the following diagram:



This **blue** area is the part of the multiplier that separates out the bits into three parts.

The **pink** area corresponds to the adding of exponents and multiplying of mantissas.

The **grey** area represents the renormalization of the exponents and mantissas.

The **green** area represents recombination and the handling of 0s and the sign bits.

The blue part is pretty self-explanatory. In the pink area, there are two things going on. First, before the multiplication of mantissas, the 23 bits are extended by one bit by setting the new b_{23} to 1. As discussed in [1], we can use an extra bit for normalized numbers (which are assumed for inputs). On the lower part of the pink box, the actual exponent is calculated for each input by subtracting 127 (“7f” in hex), from the encoded input. Then, these raw exponents are added together.

In the grey box, renormalization takes place. Depending on the output of the multiplier (specifically, depending on whether b_{22} of the carryout is equal to 1), the final exponent might have an additional change and the multiplier carryout may or may not be shifted left. In regards to regularization of the mantissa, the most significant bit of the multiplication output is taken as a carry in for the adder to produce a the final mantissa.

Finally, in the green box, the exponent and mantissa are recombined. Additionally, this part looks for any zeros (when the exponents are all zero bits) and if either of the inputs are zero, the

final result is set to positive or negative zero - depending on the output of the XOR of the sign bits.

Example results of this implementation are in the following tables.

Input A	Input B	Product
00010010010010100010110001010100	01001001000100000010000000000100	00011011111000111010010001110000
11000000000010100000000000000000	01001001000100000000000000000000	11001001110011011010000000000000
11001010001010100010001000010000	11001001000100101000001000100000	01010011110000101011101111001111
00000110100000110010001011000111	01011000111101111100111100101001	00011111111111011110000101011000
01000000100000000000000100000000	01000111010100100001000101000100	01001000010100100001001011101000
00000000000000000000000000000000	01000000111111111111111111111111	00000000000000000000000000000000
01111000100000001111111100000000	01000000100000000000000000000000	01111001100000001111111100000000

Table 2: IEEE 754 Format

Input A	Input B	Product
6.37946E-28	590336.25	4.0005956E-22
-2.15625	589824.0	-1271808.0
-2787460.0	-600098.0	1.67274912E+12
4.9327824E-35	2.17975295E+15	1.0752247E-19
4.000122	53777.266	215115.62
0	7.9999995	0
2.0930813E+34	4	8.372325E+34

Table 3: Decimal Representations

The results come out as expected. One of my initial struggles was in cases where the most significant bit in the mantissa (and some other cases) being set to one was causing issues. My fix was to find where the highest bit of the multiplication carryout was and adjust my exponent/shift depending on that case. There is likely a more efficient way of solving this case by not using a “find high” gate and instead just looking directly at b_{22}, but this was was cleaner (in my opinion). My handling of zeros could have likely been integrated with the rest of the logic, but I found it more clear to handle it separately.

In the future, I think it would be interesting to look into building a circuit that can handle non-regularized numbers, NaNs and infinities.

References

[1] Dumas, Joseph D. II. Computer Architecture: Fundamentals and Principles of Computer Design, Second Edition, CRC/Taylor & Francis, 2017. ISBN 978-1-4987-7271-6.

[2] Sjalander, M., & Larsson-Edefors, P. (2008). High-speed and low-power multipliers using the Baugh-Wooley algorithm and HPM reduction tree. Electronics, Circuits and Systems, 2008. ICECS 2008. 15th IEEE International Conference on (pp. 33–36). IEEE. doi:10.1109/ICECS.2008.4674784