VLSI Final Project: 8-Bit Simple Calculator

By Rushabh Patel

Abstract

Performing calculations is a very important process when solving problems, calculators provide the ability to perform a variety of calculations from one interface. Calculators provide a variety of different operations that can be performed, they can perform operations as simple as addition to more complicated operations such as differentiation. In order to optimize the different operations there is special hardware that is used to reduce the amount of work that needs to be done. In this project I have implemented a 8-bit simple calculator that performs the following operations: addition, subtraction, multiplication, 2^Y. This implementation of the 8-bit calculator performs addition and subtraction on a fast adder, and it performs multiplication on a fast multiplier.

Introduction

Calculators are not a new invention, they have been around for a long time. They support a variety of different operations that can be performed on them. In this project a 8-bit simple calculator was built with System Verilog, it supports the following operations: addition, subtraction, multiplication and 2^Y. Optimization is a key technique in anything that we do, and the operations that are performed by a calculator can be optimized by using specific hardware that enables faster calculations. To optimize the addition and subtraction operations of the calculator a fast adder was used, and to optimize the multiplication a fast multiplier with booth encoding was used.

To begin, let's discuss the idea of using a fast adder to optimize addition/subtraction operations. There are a variety of different adders that fall under the category of a fast adder, they include carry-look-ahead, carry-skip, and carry-bypass adder. These all have the same goal of providing the fastest possible carry path in order to optimize the addition/subtraction operation, but the level of optimization that they provide is different from one another. For this project the fast adder that was chosen was the 16-bit carry-look-ahead adder, the reasoning behind this is that this is an adder that I have built before and am comfortable working with. From the Lecture 8 lecture notes from the VLSI course, we can see that the carry-look-ahead adder provides that the delay is $T(O(\sqrt{N}))$. We can perform both addition/subtraction operations on this carry-look-ahead adder, to perform subtraction the 2's complement of the subtrahend is used.

Next, let's discuss the idea of using a fast multiplier to optimize the multiplication operation. The standard unoptimized method of performing multiplication is to multiply bit by bit to produce an array of partial products. These partial products get added up to produce the final product. The process of multiplying by one bit at a time is inefficient, instead we can group bits together to perform the multiplication thus cutting down on the number of partial products generated.

Reducing the number of partial products that are generated then reduces the total number of additions that need to be performed to get the final product. Note that in the operation x*y, x is the multiplicand and y is the multiplier. The fast multiplier in this project uses the following two steps in order to optimize the multiplication applications:

1. Use Radix-16 Booth encoding to generate the partial products. The Radix-16 groups the 8-bit y (multiplier) value into groups of 5 bit encodings, each encoding is compared to a table of encondings to determine which partial product to generate. The table of Radix-16 encodings can be found below, I found these from [1]. Note that because Radix-16 Booth encoding is being used there will only be two partial products that will be generated.
2. The two generated partial products are then added up using a fast adder, in this situation it is the 16-bit Carry-Look-Ahead adder mentioned earlier in the report.

For this fast multiplier Radix-16 Booth encoding was specifically chosen because it generates two partial products, if Radix-8 or Radix-4 Booth encoding was used then an intermediate step of using a Wallace tree would be needed in order simplify the partial products before using the CLA to add them. A Wallace tree uses a combination of full adders and half adders to simplify an array of partial products down to two partial products that can then be added in a CLA.

5 bit Encoding	**Operation**
00000	+0
00001	+1x
00010	+1x
00011	+2x
00100	+2x
00101	+3x
00110	+3x
00111	+4x
01000	+4x
010001	+5x
01010	+5x
01011	+6x
01100	+6x

01101	+7x
01110	+7x
01111	+8x
10000	-8x
10001	-7x
10010	-7x
10011	-6x
10100	-6x
10101	-5x
10110	-5x
10111	-4x
11000	-4x
11001	-3x
11010	-3x
11011	-2x
11100	-2x
11101	-1x
11110	-1x
11111	0

Figure 1. The Radix-16 Booth Encoding used in order to generate the partial products in the fast multiplier, this table comes from the following paper: Design and Comparison of High Speed Radix-8 and Radix-16 Booth's Multipliers by Chaudhary,Kularia,Choudhary,Kaur and Vats. This paper is located in the reference section and it is [1].

The final operation to be implemented was 2^Y, this was very simple to implement because I used Verilog built in exponentiation operation (2**y).

Related Work

As mentioned before calculators are not a new invention, they are something that everyone has used for a long time. There have been a variety of implementations of calculators using Verilog, and they also probably implement the same optimization ideas that I use to make their calculators more efficient.

Data / Method / Model Description

Let's begin with the simple_calculator module itself. The first two inputs are 8-bit x and y, the third input is the 3-bit operation to be performed, and there is the 16-bit out for the output from the calculator. The 3-bit codes for the operations are as follows:

- 3'b000 → Addition
- 3'b001 → Subtraction
- 3'b011 → Multiplication
- 3'b100 → 2^Y

In order to perform the addition/subtraction operations a 16-bit CLA, and to perform a Radix-16 Booth multiplier is used. These modules will be discussed later. **Note that the 8-bit operands have an additional eight zeros added to the front to make them 16-bit.** Performing addition simply requires feeding the two operands to the CLA and then setting the trigger to 1. Performing subtraction follows a similar methodology as addition but the two's complement of the subtrahend is sent to the CLA. Performing the multiplication requires feeding the two operands to the Booth encoding multiplier and then setting the booth trigger to 1. Performing the 2^Y operation is straightforward and is done with the following line of code: out = 2**y.

The next module is the carry_lookahead_adder_16bit() and this is used for fast addition/subtraction. The inputs/outputs for this module are:

- x → 16-bit operand
- y → 16-bit operand
- cin → Carry In
- sum → 16-bit output Final Sum
- cout → Carry Out
- trigger → Used to trigger the CLA

The process of summing x and y is very straightforward and is as follows:

1. Compute the propagation for each stage. This means that each respective bit from x and y is XORed. A total of 16 propagation bits are generated.Below is the specific computation performed using Verilog code.

```
//Compute the propogation for each stage
p0 = x[0] ^ y[0];
p1 = x[1] ^ y[1];
p2 = x[2] ^ y[2];
p3 = x[3] ^ y[3];
p4 = x[4] ^ y[4];
p5 = x[5] ^ y[5];
p6 = x[6] ^ y[6];
p7 = x[7] ^ y[7];
p8 = x[8] ^ y[8];
p9 = x[9] ^ y[9];
p10 = x[10] ^ y[10];
p11 = x[11] ^ y[11];
p12 = x[12] ^ y[12];
p13 = x[13] ^ y[13];
p14 = x[14] ^ y[14];
p15 = x[15] ^ y[15];
```

2. Compute the generated bit for each stage. This means that each respective bit from x and y is ANDed. A total of 16 generated bits are generated. Below is the specific computation performed using Verilog code.

```
// Compute the generated for each stage
g0 = x[0] & y[0];
g1 = x[1] & y[1];
g2 = x[2] & y[2];
g3 = x[3] & y[3];
g4 = x[4] & y[4];
g5 = x[5] & y[5];
g6 = x[6] & y[6];
g7 = x[7] & y[7];
g8 = x[8] & y[8];
g9 = x[9] & y[9];
g10 = x[10] & y[10];
g11 = x[11] & y[11];
g12 = x[12] & y[12];
g13 = x[13] & y[13];
g14 = x[14] & y[14];
g15 = x[15] & y[15];
```

3. Compute the carry for each stage. A total of 16 carry bits are generated. Below is the exact computation for computing each carry bit, note that the code gets cut off but you get the general idea.

```
// Compute the carry for each stage
c1 = g0 | (p0 & cin);
c2 = g1 | (p1 & g0) | (p1 & p0 & cin);
c3 = g2 | (p2 & g1) | (p2 & p1 & g0) | (p2 & p1 & p0 & cin);
c4 = g3 | (p3 & g2) | (p3 & p2 & g1) | (p3 & p2 & p1 & g0) | (p3 & p2 & p1 & p0 & cin);
c5 = g4 | (p4 & g3) | (p4 & p3 & g2) | (p4 & p3 & p2 & g1) | (p4 & p3 & p2 & p1 & g0) | (p4 & p3 & p2 & p1 & p0 & cin);
c6 = g5 | (p5 & g4) | (p5 & p4 & g3) | (p5 & p4 & p3 & g2) | (p5 & p4 & p3 & p2 & g1) | (p5 & p4 & p3 & p2 & p1 & g0) | (p5 & p4 & p3 & p2 & p1 & p0 & cin);
c7 = g6 | (p6 & g5) | (p6 & p5 & g4) | (p6 & p5 & p4 & g3) | (p6 & p5 & p4 & p3 & g2) | (p6 & p5 & p4 & p3 & p2 & g1) | (p6 & p5 & p4 & p3 & p2 & p1 & g0) | (p6
c8 = g7 | (p7 & g6) | (p7 & p6 & g5) | (p7 & p6 & p5 & g4) | (p7 & p6 & p5 & p4 & g3) | (p7 & p6 & p5 & p4 & p3 & g2) | (p7 & p6 & p5 & p4 & p3 & p2 & g1) | (p7
c9 = g8 | (p8 & g7) | (p8 & p7 & g6) | (p8 & p7 & p6 & g5) | (p8 & p7 & p6 & p5 & g4) | (p8 & p7 & p6 & p5 & p4 & g3) | (p8 & p7 & p6 & p5 & p4 & p3 & g2) | (p8
c10 = g9 | (p9 & g8) | (p9 & p8 & g7) | (p9 & p8 & p7 & g6) | (p9 & p8 & p7 & p6 & g5) | (p9 & p8 & p7 & p6 & p5 & g4) | (p9 & p8 & p7 & p6 & p5 & p4 & g3) | (p
c11 = g10 | (p10 & g9) | (p10 & p9 & g8) | (p10 & p9 & p8 & g7) | (p10 & p9 & p8 & p7 & g6) | (p10 & p9 & p8 & p7 & p6 & g5) | (p10 & p9 & p8 & p7 & p6 & p5 & g
c12 = g11 | (p11 & g10) | (p11 & p10 & g9) | (p11 & p10 & p9 & g8) | (p11 & p10 & p9 & p8 & g7) | (p11 & p10 & p9 & p8 & p7 & g6) | (p11 & p10 & p9 & p8 & p7 &
c13 = g12 | (p12 & g11) | (p12 & p11 & g10) | (p12 & p11 & p10 & g9) | (p12 & p11 & p10 & p9 & g8) | (p12 & p11 & p10 & p9 & p8 & g7) | (p12 & p11 & p10 & p9 &
c14 = g13 | (p13 & g12) | (p13 & p12 & g11) | (p13 & p12 & p11 & g10) | (p13 & p12 & p11 & p10 & g9) | (p13 & p12 & p11 & p10 & p9 & g8) | (p13 & p12 & p11 & p1
c15 = g14 | (p14 & g13) | (p14 & p13 & g12) | (p14 & p13 & p12 & g11) | (p14 & p13 & p12 & p11 & g10) | (p14 & p13 & p12 & p11 & p10 & g9) | (p14 & p13 & p12 &
```

4. Compute the sum for each stage. This means that for each respective bit in the sum the respective propagation bit and carry bit are XORed. Below is the specific computation performed using Verilog code.

```
// Compute the sum
sum[0] = p0 ^ cin;
sum[1] = p1 ^ c1;
sum[2] = p2 ^ c2;
sum[3] = p3 ^ c3;
sum[4] = p4 ^ c4;
sum[5] = p5 ^ c5;
sum[6] = p6 ^ c6;
sum[7] = p7 ^ c7;
sum[8] = p8 ^ c8;
sum[9] = p9 ^ c9;
sum[10] = p10 ^ c10;
sum[11] = p11 ^ c11;
sum[12] = p12 ^ c12;
sum[13] = p13 ^ c13;
sum[14] = p14 ^ c14;
sum[15] = p15 ^ c15;
```

The next important module is the booth_multplier(), this is responsible for performing the fast multiplication. The inputs/outputs for this module are:

- x → 8-bit multiplicand
- y → 8-bit multiplier
- product → 16-bit product generated by multiplying x and y
- trigger → Used to trigger the booth_multplier module

The booth_multplier performs multiplication in a very straightforward manner, it follows the following steps:

1. Breaks the multiplier into two 5-bit Booth encodings. Each Booth encoding is compared against in the Radix-16 Booth encoding table, the appropriate calculation is applied to the multiplicand and it is added to the partial products array.

2. The two partial products are passed to a CLA to perform the fast addition, the sum from the CLA is the final product.

The table used to look up the 5-bit Booth encodings is not something that I came up with, I referenced it from [1].

Experimental Procedure and Results

To test the simple calculator a testbench was generated, this is the sc_testbench.sv file that is included in the final project submission. Testing a calculator is straightforward, for each operation we need to test a variety of different operands and check if the output from the calculator matches what we would expect. In the sc_testbench.sv there are a total of five tests for each operation for a total of twenty tests, the output from the calculator was then compared against the expected outputs. After running the tests in sc_testbench.sv on edaplayground the results below were generated, they show that the outputs from the calculator match the outputs from the respective built in operations. The results verify that the simple calculator was implemented correctly.

```
**TO GET EXPECTED VALUES THE BUILT IN OPERATIONS PROVIDED BY VERILOG WERE USED **
Testing Operation: Addition
Test 1: X = 00000001 ,Y = 00000001 ,X+Y = 00000010, Expected = 00000010
Test 2: X = 00000010 ,Y = 00000010 ,X+Y = 00000100, Expected = 00000100
Test 3: X = 10000000 ,Y = 01000000 ,X+Y = 11000000, Expected = 11000000
Test 4: X = 00000111 ,Y = 01101011 ,X+Y = 01110010, Expected = 01110010
Test 5: X = 10001100 ,Y = 01111000 ,X+Y = 00000100, Expected = 00000100
Testing Operation: Subtraction - - - - - - - - - - - - - - - - - -
Test 6:   X = 00000001 ,Y = 00000001 ,X-Y = 00000000, Expected = 00000000
Test 7:   X = 00000010 ,Y = 00000001 ,X-Y = 00000001, Expected = 00000001
Test 8:   X = 00000100 ,Y = 00000010 ,X-Y = 00000010, Expected = 00000010
Test 9:   X = 00001000 ,Y = 00000001 ,X-Y = 00000111, Expected = 00000111
Test 10:  X = 10000000 ,Y = 00010000 ,X-Y = 01110000, Expected = 01110000
Testing Operation: Multiplication - - - - - - - - - - - - - - - - -
Test 11:  X = 00000010 ,Y = 00000010 ,X*Y = 00000100, Expected = 00000100
Test 12:  X = 00000100 ,Y = 00000010 ,X*Y = 00001000, Expected = 00001000
Test 13:  X = 00010000 ,Y = 00000100 ,X*Y = 01000000, Expected = 01000000
Test 14:  X = 00010000 ,Y = 00000010 ,X*Y = 00100000, Expected = 00100000
Test 15:  X = 00000010 ,Y = 00000000 ,X*Y = 00000000, Expected = 00000000
Testing Operation: 2^y - - - - - - - - - - - - - - - - - - - -
The operand x does not matter, only y is important
Test 16:  X = 00000000 ,Y = 00000001 ,2^Y = 0000000000000010, Expected = 00000000000000000000000000000010
Test 17:  X = 00000000 ,Y = 00000010 ,2^Y = 0000000000000100, Expected = 00000000000000000000000000000100
Test 18:  X = 00000000 ,Y = 00000011 ,2^Y = 0000000000001000, Expected = 00000000000000000000000000001000
Test 19:  X = 00000000 ,Y = 00000100 ,2^Y = 0000000000010000, Expected = 00000000000000000000000000010000
Test 20:  X = 00000000 ,Y = 00000101 ,2^Y = 0000000000100000, Expected = 00000000000000000000000000100000
Done
```

Conclusion

In conclusion, I was able to successfully implement a 8-bit simple calculator that supports the following operations: addition, subtraction, multiplication and 2^y. In order to support fast addition and subtraction I was able to successfully build a 16-bit Carry-Look-Ahead adder. Also to support fast multiplication I was able to implement a fast multiplier that utilized Radix-16 Booth encoding to generate the partial products and a 16-bit Carry-Look-Ahead adder to sum the partial products. Each operation of the simple calculator was tested with a variety of test cases, and each operation successfully worked as intended.

Reference

[1] Chaudhary, lla, Kularia, D., Choudhary, R., Kaur, G., & Vats, A. (2018, July 2). *Design and comparison of high speed radix-8 and radix-16 booth's ... - IJCA*. International Journal of Computer Applications. Retrieved December 28, 2022, from https://www.ijcaonline.org/archives/volume181/number2/chaudhary-2018-ijca-917410.pdf