CARRY-FREE ARITHMETIC IMPLEMENTATIONS

Vikram Voleti[1]

[1]International Institute of Information Technology, Hyderabad
vikram.voleti@gmail.com

ABSTRACT

This paper uses Recoded Binary Signed Digits to perform carry-free arithmetic operations of addition, subtraction and (Karatsuba) multiplication on binary numbers. We provide detailed truth tables and logic circuits of the carry-free implementation of addition. Detailed circuit design of carry-free multiplication is also drawn. We compare the path delays and FPGA utilizations with that of standard implementation. The problem of implementing carry-free arithmetic is analyzed, and design improvements to our method are provided.

KEYWORDS

Carry-Free arithmetic, Redundancy, VLSI

1. INTRODUCTION

Even with recent development in technology of integrated circuits, various high-speed circuits with regular structures and low power design still suffer from problems of path delay, limited range of the number of bits, and complexity in hardware. It is well-known that the addition of radix-B numbers suffers from carry propagation. Since the speed of digital processors depends mostly on the speed of the adders used in the system, it is critical to optimize the addition operation to speed up all computations.

Simple carry-ripple adders take time proportional to the length of the longest path from input to output. Even though this can be reduced to a depth of *O(log(n))* by carry-lookahead adders [1], the depth still grows with the number of digits. In fact, for all basic operations on radix-*B* numbers, this minimal *O(log(n))* depth cannot be improved upon unless a different number system is used. Signed digit number systems [2-8] could be used to eliminate carry propagation. This means addition would be done in *O(1)* time. However, as [8] explains, the standard algorithm for addition do not work in the case of binary numbers ($B = 2$). Hence, [9, 10] suggested to recode the input numbers so that the later addition and subtraction of binary numbers will become carry-free.

Parhami [9], in particular, recodes the numbers into a format that allows carry-free arithmetic operations effectively. In this paper, this format is used to recode binary numbers for carry-free arithmetic operations (see section 4). Since the focus is on using carry-free logic to perform arithmetic operations on binary numbers, the goal is to make arithmetic modules that take binary numbers as input and perform addition and multiplication on them. To consider least memory requirement, only the minimal digit set {-1, 0, 1} has been considered. It is also of significant advantage to have these circuits as reusable modules in building circuits for more complex arithmetic operations, like modular reduction, which has significant relevance in cryptographic applications.

In this paper, circuits for addition of the recoded numbers, addition of binary numbers in a non-naive way, and multiplication of binary numbers have been designed. The modular nature of these circuits is exemplified by the use of Carry-Free Adders as modules to build the Carry-Free Multiplier. These circuits have been implemented on an FPGA, and their speeds and area requirements have been compared with those of standard addition. The carry-free implementations are then analyzed to probe their limitations and suggest remedies for future work.

The contents of this paper are: (i) details of the conversion between binary, signed digit, and Recoded Binary Signed Digit (see section 4) numbers, with truth table and logic equations, (ii) implementation of arithmetic operations of addition, subtraction, multiplication using carry-free logic (sections 5, 6), (iii) comparison of the carry-free implementations and their standard versions (section 7), (iv) analysis of the carry-free implementations (section 8).

2. Related Work

[11] begins to design a carry-free multiplier, however, we detail all the design features involved to create an end-to-end carry-free multiplier that is built using carry-free adders as modules. [12] uses the same format as us, that mentioned in [9], but uses trinary signed digits. [13] uses bit-pair recording to generate partial products, and then adds them in the form of a binary tree using a carry-free adder. In this paper, our carry-free multiplier uses the Karatsuba multiplication algorithm [14] (see section 6) and performs all additions using a carry-free adder.

The quarternary signed digit system has been explored quite significantly in light of carry-free implementations of addition [15-18] and multiplication [19-21]. We, however restrict our paper to the format specified in [9], which only uses a binary signed digit format, albeit recoded. In addition, we provide our own circuit designs for carry-free adders of recoded numbers, binary numbers, and multiplier that uses the carry-free adders as internal modules (see sections 5, 6).

3. Our Implementation

To achieve carry-free implementations of arithmetic operations on binary numbers, the input binary numbers are first converted to a binary signed digit format. This paper uses Recoded Binary Signed Digit (RBSD, see section 4) format as described in [5]. Arithmetic operations in the domain of RBSD numbers can be implemented very effectively in a carry-free way.

The RBSD numbers are then operated upon to produce binary signed digits. These arithmetic operations are summarized in the form of truth tables, since they are combinatorial in nature. The truth tables are then converted to Product-of-Sum equations so that an electronic circuit could be designed to implement those tables.

These electronic circuits are implemented in Verilog and run on an FPGA to measure the path delay and LUT usage. The results are compared with those of standard textbook implementation of addition. An analysis of the path delays is also done to reveal the costliest path in the implementation.

We first describe RBSD numbers and the conversions between binary, BSD and RBSD numbers in detail in section 4. We mention why using RBSD numbers lends itself to carry-free arithmetic. We then look at implementations of carry-free arithmetic operations on RBSD numbers in section 5. In section 6, we design end-to-end carry-free arithmetic operations on binary numbers.

4. Recoded Binary Signed Digit Numbers

Carry-free operations can be implemented by first converting binary numbers to binary signed digits {-D, …, 0, …, D}. The negative numbers in the digit set help extend the operation set of the field of numbers described by the set to carry-free operations.

Among the different methods of converting binary numbers into signed digits [7, 8, 9], we chose the solution provided by Parhami [8] to recode a given binary number *x* of length *n* to an equivalent Signed Digit number *z* of length *n+1* such that there are no two neighboring digits z_{i+1} and z_i with $z_{i+1} * z_i = 1$. As proven in [8], this is essential to implementing carry-free addition, and shall be called "Recoded Binary Signed Digits", or RBSD. We limit our digit set to {-1, 0, +1}, and show that computational gains by using the least number of signed digits are significant in themselves to bother with more digits.

In the following subsections, conversions between Binary, BSD and RBSD formats are detailed. As these are combinatorial circuits, there is no delay due to any carry propagation.

4.1. Binary to RBSD

Binary numbers are first converted to RBSD before performing any arithmetic operation on them. The table of conversion from two consecutive binary digits to one RBSD digit at the higher binary digit's position is:

Table 1. Conversion from Binary to RBSD

X_i	X_{i-1}	Z_i
0	0	0
0	1	1
1	0	-1 ($\bar{1}$)
1	1	0

For example, the binary number 10110 is converted to $1\bar{1}$ 10 $\bar{1}$ 0. Moving bit-by-bit from lower to higher bit, 0(0) → 0, 10 → $\bar{1}$, 11 → 0, 01 → 1, 10 → $\bar{1}$, (0)1 → 1. As can be seen the RBSD number contains one bit more than the original binary number.

The conversion from binary to RBSD is such that the value of the number remains the same, but there are no two consecutive 1's or -1's in the number. This, in effect, eliminates the possibility of a Carry beyond one position. It is easy to see the conversion from binary to RBSD:

$$Z_i^s = X_i \,\&\, {\sim}X_{i-1} \tag{1}$$

$$Z_i^v = X_i \,\&\, {\sim}X_{i-1} \tag{2}$$

Here, Z_i^s is the sign bit and Z_i^v is the value bit of the RBSD number.

4.2. BSD to RBSD

It is important to note that the field of RBSD numbers is not closed under addition. This means that the addition of two RBSD numbers shall result in a BSD sum, but which is not necessarily an RBSD number. Thus, a Carry-Free Addition module of binary numbers (converted to RBSD) needs to be followed by a BSD-to-RBSD Converter. Although this makes for another overhead in computation, we shall see in section 7 that the overall computation time is significantly lesser than that for two BSD numbers.

The conversion from BSD to RBSD is a more complex process, because a binary number has only two possible digits: 0 and 1, while a BSD number has three possible inputs: -1, 0 and 1. Fortunately, the conversion table has been provided in [8], and is summarized in Table 2. In the table, 'X' implies "don't care", i.e. either of {-1, 0, 1}. The conversion of a BSD bit y_i into an RBSD bit z_i requires the 3 following bits in the BSD number, y_{i-1}, y_{i-2}, y_{i-3}, as explained in [9]. It can be seen that this is also a combinatorial circuit, and takes *O(1)* time to run.

Table 2. Conversion from BSD to RBSD

y_i	y_{i-1}	y_{i-2}	y_{i-3}	z_i
-1	-1	-1	X	0
		0	-1	0
		0	0	1
		0	1	1
		1	X	1
-1	0	-1	X	1
		0	X	-1
		1	X	-1
-1	1	-1	X	-1
		0	-1	-1
		0	0	0
		0	1	0
		1	X	0
0	-1	-1	X	-1
		0	-1	-1
		0	0	0
		0	1	0
		1	X	0
0	0	X	X	0
0	1	-1	X	0
		0	-1	0
		0	0	1
		0	1	1
		1	X	1
1	-1	-1	X	0
		0	-1	0
		0	0	1
		0	1	1
		1	X	1
1	0	-1	X	1
		0	X	-1
		1	X	-1
1	1	-1	X	-1
		0	-1	-1
		0	0	0
		0	1	0
		1	X	0

4.3. BSD to Binary

The following table can be used to convert a BSD or RBSD number into its Binary form. The process involves carry propagation, and the table details the conversion of one bit of the BSD digit along with the carry bit at its position (0 for the least significant) into one bit at the same position in its Binary form, and a carry bit for the next digit.

Table 3. Conversion from BSD (or RBSD) to Binary.

BSD (Z)	**carry_in (c)**	**Binary (B)**	**carry_out (C)**
-1	0	1	1
-1	1	0	1
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

These conversions can be represented through logical equations, Z^s is the sign bit, and Z^v is the value bit of Z as:

$$B = (\sim Z^v \,\&\, c) \mid (Z^v \,\&\, \sim c) \tag{3}$$

$$C = (\sim Z^v \,\&\, c) \mid Z^s \tag{4}$$

5. Carry-Free Arithmetic Operations of RBSD Numbers

In the following subsections we descibe the carry-free implementations of addition and subtraction of RBSD numbers.

Table 4. Carry-Free addition of two RBSD numbers

X_i	Y_i	X_{i-1}	Y_{i-1}	S_i
-1	-1	0	0	0
		0	1	0
		1	0	0
		1	1	0
-1	0	0	X	-1
		1	-1	-1
		1	0	-1
		1	1	0
-1	1	0	-1	0
		0	0	0
		1	-1	0
		1	0	0
0	-1	X	0	-1
		-1	1	-1
		0	1	-1
		1	1	0
0	0	-1	-1	-1
		X	0	0
		-1	1	0
		0	X	0
		1	-1	0
		1	1	1
0	1	-1	-1	0
		X	0	1
		0	-1	1
		1	-1	1
1	-1	-1	0	0
		1	1	0
		0	0	0
		0	1	0
1	0	-1	-1	0
		-1	0	1
		-1	1	1
		0	X	1
1	1	-1	-1	-1
		-1	0	0
		0	-1	0
		0	0	0

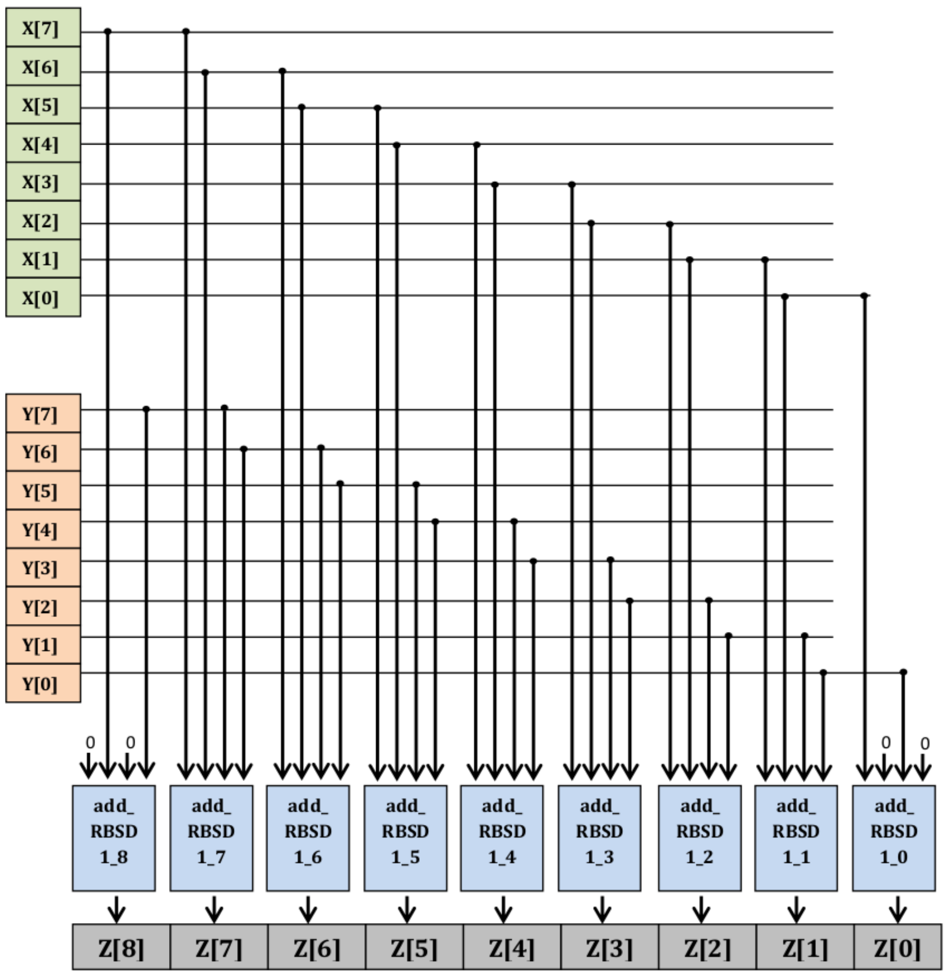

Figure 1. Carry-free addition of two 8-bit RBSD numbers

5.1. Carry-Free Addition of RBSD numbers

Before dealing with Binary numbers, we first tackle the easier problem of designing carry-free addition of two RBSD numbers using the truth table in Table 1. The sum digit of two numbers depends on two consecutive digits of each of the input numbers, since carry has to be accounted for. So we consider two bits each from the two inputs, $X_{i,}$ X_{i-1}, Y_i, Y_{i-1}, to compute the sum at the higher bit's position, S_i. The resulting truth table is documented in Table 4.

Table 4 can transformed into a Product-of-Sum form to implement an electronic circuit that performs Carry-Free RBSD Addition. The most optimized Product-of-Sum form of Table 4 is:

$$S0 = \sim X_i^s \mid Y_i^s \mid \sim Y_i^v \tag{5}$$

$$S1 = \sim X_i^v \mid Y_i^v \mid \sim X_{i-1}^s \mid \sim Y_{i-1}^s \tag{6}$$

$$S2 = X_i^v \mid \sim Y_i^v \mid \sim X_{i-1}^s \mid \sim Y_{i-1}^s \tag{7}$$

$$S3 = \sim X_i^s \mid Y_i^s \mid \sim X_{i-1}^v \mid Y_{i-1}^s \mid \sim Y_{i-1}^v \tag{8}$$

$$S4 = X_i^v \mid \sim Y_i^s \mid X_{i-1}^s \mid \sim X_{i-1}^v \mid \sim Y_{i-1}^v \tag{9}$$

$$S_i^s = S0 \,\&\, S1 \,\&\, S2 \,\&\, S3 \,\&\, S4 \,\&\, (\sim X_i^v \mid \sim Y_i^s) \,\&\, (X_i^s \mid Y_i^s \mid X_{i-1}^s) \,\&\, (X_i^s \mid Y_i^s \mid Y_{i-1}^s) \tag{10}$$

$$S_i^v = S0 \,\&\, S1 \,\&\, S2 \,\&\, S3 \,\&\, S4 \,\&\, (X_i^s \mid \sim X_i^v \mid \sim Y_i^s) \,\&\, (\sim X_i^v \mid \sim Y_i^v \mid X_{i-1}^v) \,\&\, (\sim X_i^v \mid \sim Y_i^v \mid Y_{i-1}^v) \,\&\, (X_i^v \mid Y_i^v \mid X_{i-1}^v) \,\&\, (X_i^v \mid Y_i^v \mid Y_{i-1}^v) \,\&\, (X_i^v \mid Y_i^v \mid \sim X_{i-1}^s \mid Y_{i-1}^s) \,\&\, (X_i^v \mid Y_i^v \mid X_{i-1}^s \mid \sim Y_{i-1}^s) \tag{11}$$

Equations (10) and (11) describe the *i*-th sign and value bits of the Sum of two binary numbers *X* and *Y* at the *i*-th position.

As an example, Figure 1 shows the addition of two 8-bit numbers. The computation of the sum bit at each position is performed by the "add_RBSD" module that uses circuits described by equations (10) and (11). It is important to note that the computation of the first and last bits require phantom 0s to complete their inputs.

5.2. Carry-Free Subtraction of RBSD Numbers

The negative of an RBSD number can be obtained by interchanging the 1's and -1's in it. For example, RBSD [1 0 -1] is decimal 3, while RBSD [-1 0 1] is decimal -3. Thus, by carrying out Carry-Free Addition of the negative of the number to subtract and the second number, Carry-Free Subtraction is achieved.

6. Carry-Free Arithmetic Operations of Binary Numbers

In the following subsections, carry-free implementations of addition, subtraction, multiplication of two binary numbers are described.

6.1. Carry-Free Addition of Binary Numbers

The addition of two binary numbers can be carried out by simply connecting in cascade a Binary-to-RBSD Converter module to the Carry-Free Adder for RBSD Numbers designed previously in Section 4.1. Although this is functionally correct, a better approach would be to combine the truth tables of these two modules to make a single truth table that takes as input binary digits, and produces the sum.

Table 5. Carry-Free addition of two binary numbers

A_i	A_{i-1}	A_{i-2}	B_i	B_{i-1}	B_{i-2}	S_i
0	0	0	0	0	0	0
			0	0	1	0
			0	1	0	1
			0	1	1	1
			1	0	0	$\bar{1}$
			1	0	1	$\bar{1}$
			1	1	0	0
			1	1	1	0
0	0	1	0	0	0	0
			0	0	1	1
			0	1	0	1
			0	1	1	1
			1	0	0	$\bar{1}$
			1	0	1	0
			1	1	0	0
			1	1	1	0
0	1	0	0	0	0	1
			0	0	1	1
			0	1	0	$\bar{1}$
			0	1	1	0
			1	0	0	0
			1	0	1	0
			1	1	0	0
			1	1	1	1
0	1	1	0	0	0	1
			0	0	1	1
			0	1	0	0
			0	1	1	0
			1	0	0	0
			1	0	1	0
			1	1	0	1
			1	1	1	1
1	0	0	0	0	0	$\bar{1}$
			0	0	1	$\bar{1}$
			0	1	0	0
			0	1	1	0
			1	0	0	0
			1	0	1	0
			1	1	0	$\bar{1}$
			1	1	1	$\bar{1}$
1	0	1	0	0	0	$\bar{1}$
			0	0	1	0
			0	1	0	0
			0	1	1	0
			1	0	0	0
			1	0	1	1
			1	1	0	$\bar{1}$
			1	1	1	$\bar{1}$
1	1	0	0	0	0	0
			0	0	1	0
			0	1	0	0
			0	1	1	1
			1	0	0	$\bar{1}$
			1	0	1	$\bar{1}$
			1	1	0	$\bar{1}$
			1	1	1	0
1	1	1	0	0	0	0
			0	0	1	0
			0	1	0	1
			0	1	1	1
			1	0	0	$\bar{1}$
			1	0	1	$\bar{1}$
			1	1	0	0
			1	1	1	0

Thus, a new truth table, Table 5, is constructed by merging the two truth tables in Tables 1 and 4. This new table takes as input three consecutive bits in a binary number. This is because every bit of an RBSD number depends on two bits in the binary number, and RBSD addition at a bit requires two RBSD bits. The sum bit is to be placed in the most significant position among the three input positions, in the sum.

As can be observed from Table 5, there are 64 combinations of input to be taken care of. Each combination results in two RBSD numbers, which then needed to be added using Table 4. Then the optimal Product-of-Sum form of Table 5 was made as equations (19) and (20).

$$S0 = A_i \mid A_{i-1} \mid {\sim}B_i \mid {\sim}B_{i-1} \tag{12}$$

$$S1 = A_i \mid A_{i-1} \mid {\sim}A_{i-2} \mid {\sim}B_i \mid B_{i-1} \mid {\sim}B_{i-2} \tag{13}$$

$$S2 = A_i \mid {\sim}A_{i-1} \mid A_{i-2} \mid B_i \mid {\sim}B_{i-1} \mid {\sim}B_{i-2} \tag{14}$$

$$S3 = {\sim}A_i \mid A_{i-1} \mid B_i \mid {\sim}B_{i-1} \tag{15}$$

$$S4 = {\sim}A_i \mid A_{i-1} \mid {\sim}A_{i-2} \mid B_i \mid B_{i-1} \mid {\sim}B_{i-2} \tag{16}$$

$S5 = {\sim}A_i \mid {\sim}A_{i-1} \mid A_{i-2} \mid {\sim}B_i \mid {\sim}B_{i-1} \mid {\sim}B_{i-2}$ (17)

$S6 = {\sim}A_i \mid {\sim}A_{i-1} \mid {\sim}A_{i-2} \mid {\sim}B_i \mid {\sim}B_{i-1}$ (18)

$S_i^s = S0 \;\&\; S1 \;\&\; S2 \;\&\; S3 \;\&\; S4 \;\&\; S5 \;\&\; S6 \;\&\; (A_i \mid A_{i-1} \mid B_i) \;\&\; (A_i \mid {\sim}A_{i-1} \mid A_{i-2} \mid B_i \mid B_{i-1}) \;\&\; (A_i \mid {\sim}A_{i-1} \mid A_{i-2} \mid {\sim}B_i) \;\&\; (A_i \mid {\sim}A_{i-1} \mid {\sim}A_{i-2}) \;\&\; ({\sim}A_i \mid A_{i-1} \mid {\sim}B_i \mid B_{i-1}) \;\&\; ({\sim}A_i \mid {\sim}A_{i-1} \mid B_i)$ (19)

$S_i^v = S0 \;\&\; S1 \;\&\; S2 \;\&\; S3 \;\&\; S4 \;\&\; S5 \;\&\; S6 \;\&\; (A_i \mid A_{i-1} \mid A_{i-2} \mid B_i \mid B_{i-1}) \;\&\; (A_i \mid A_{i-1} \mid {\sim}A_{i-2} \mid B_i \mid B_{i-1}) \;\&\; (A_i \mid {\sim}A_{i-1} \mid {\sim}B_i \mid B_{i-1}) \;\&\; (A_i \mid A_{i-1} \mid A_{i-2} \mid {\sim}B_i \mid {\sim}B_{i-1} \mid B_{i-2}) \;\&\; (A_i \mid {\sim}A_{i-1} \mid {\sim}A_{i-2} \mid B_i \mid {\sim}B_{i-1}) \;\&\; ({\sim}A_i \mid A_{i-1} \mid A_{i-2} \mid {\sim}B_i \mid B_{i-1}) \;\&\; ({\sim}A_i \mid A_{i-1} \mid {\sim}A_{i-2} \mid {\sim}B_i \mid B_{i-1} \mid B_{i-2}) \;\&\; ({\sim}A_i \mid {\sim}A_{i-1} \mid B_i \mid B_{i-1}) \;\&\; ({\sim}A_i \mid {\sim}A_{i-1} \mid A_{i-2} \mid B_i \mid {\sim}B_{i-1} \mid B_{i-2})$ (20)

6.2. Carry-Free Subtraction of Binary Numbers

It is fairly straightforward to see that subtraction of two binary numbers can be achieved by the addition of one number with the negative of the other, as mentioned in Section 4.2. In this case, the two binary numbers must be converted to their 1's complement or 2's complement forms before converting them into RBSD numbers. Section 7.2 deals with this.

6.3. Carry-Free Multiplication of Binary Numbers

Multiplication forms a very critical part of arithmetic operations used in any signal-processing application. The basic (school-book) multiplication algorithm is time-intensive ($O(n^2)$), and so is not a preferable option for many core operations.

Karatsuba Algorithm [14] for multiplication proves to be the ideal algorithm, providing a much faster $O(n^{\mathrm{Log}_2 3}$) time. Making this algorithm carry-free can prove quite useful. Since multiplication involves repeated addition, acascading of carry-free addition modules produces carry-free multiplication.

The Karatsuba algorithm implements multiplication of two numbers, x and y, by first splitting them into two parts: $x = x_1B^m + x_0$, $y = y_1B^m + y_0$, at some digit place B^m. The standard multiplication of x and y would require 4 multiplications, as can be seen from equation (21).

$$x * y = (x_1B^m + x_0) * (y_1B^m + y_0) = (x_1 * y_1)B^{2m} + (x_1{*}y_0 + x_0{*}y_1)B^m + x_0{*}y_0 \quad (21)$$

The Karatsuba algorithm requires only 3 multiplications, given in equations (22), (23), and (24).

$$z0 = x1 * y1 \quad (22)$$

$$z1 = x0 * y0 \quad (23)$$

$$z2 = (x1 + x0) * (y1 + y0) \quad (24)$$

Using z0, z1 and z2, the product of x and y is computed in equation (25).

$$x * y = z_0B^{2m} + (z_2 - z_1 - z_0)B^m + z_1 \quad (25)$$

Thus, the Karatsuba algorithm decreases the number of multiplications by 1, while increasing the number of additions (subtractions are counted as additions) by 1. Since addition is an *O(1)* process using the carry-free adder designed in the previous section, this is a highly feasible compromise.

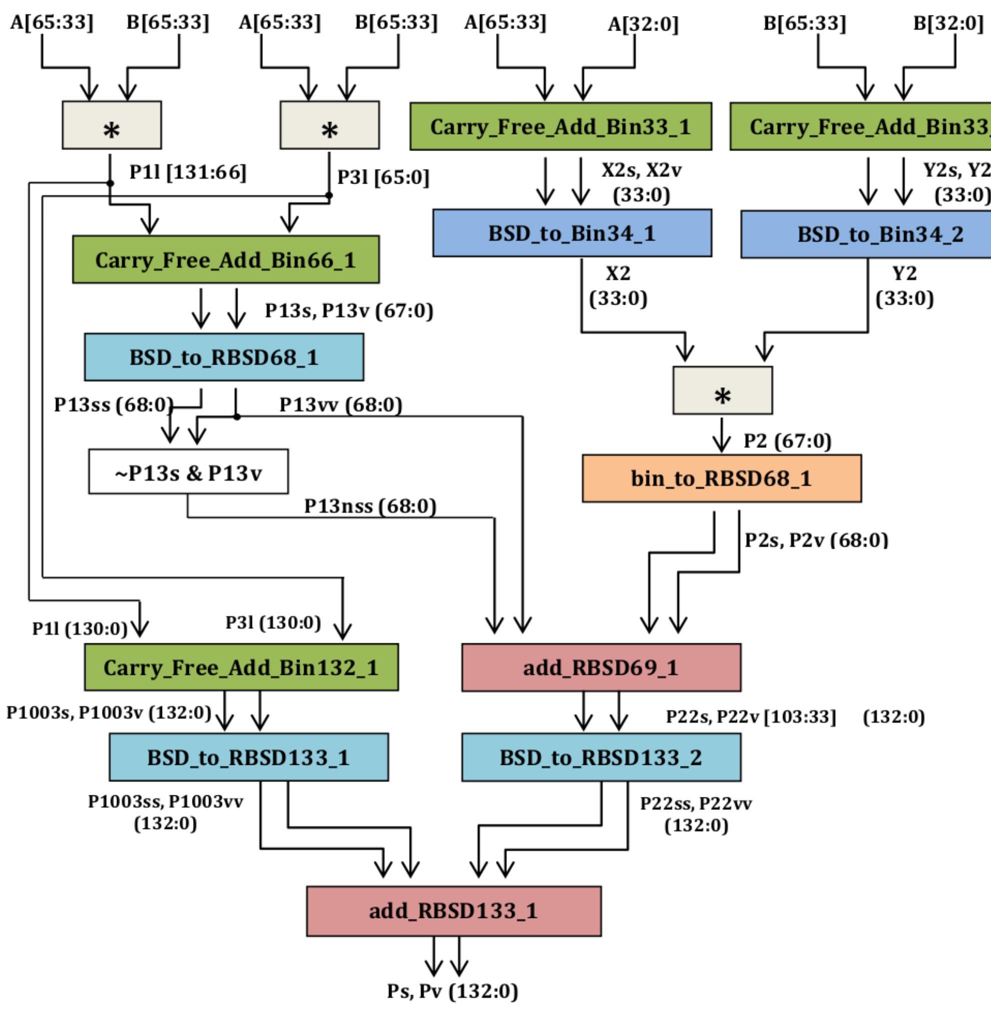

Figure 2. Circuit design of carry-free muliplication of binary numbers

Figure 2 is the circuit implementation of the Karatsuba Algorithm, for carry-free multiplication of two 66-bit binary numbers A and B. As can be seen, the three multiplications are performed on binary numbers. One of these required the conversion from BSD to binary, which is a carry-full step and contributes significantly to the path delay. However, we shall see in section 7 that despite this, the path delay of carry-free multiplication is lesser than that of standard addition. It can be seen that at the end of every module, the output is converted to RBSD, to ensure carry-free operation in the subsequent modules.

7. Results

All circuits were designed in Verilog and run using Xilinx-9 on a Virtex-5 FPGA (5vlx30ff324). All the modules were also first coded in Octave to verify their computation.

7.1 Addition

Table 6 lists the comparison of computation times of addition and multiplication between standard implementations and carry-free implementations described in the previous sections. Two metrics were analysed to compare the performance – path delay, Slice Logic Utilization, i.e. the percentage of Slice LUTs on the FPGA utilized, which is equivalent to the Slice Logic Distribution, i.e. the number of LUT Flip-Flop pairs utilized, out of the 19200 present. As can be seen from Table 6, carry-binary binary addition takes lesser memory, as well as significantly lesser time than standard addition.

Table 6. Comparison of results

Metric	Standard addition	Carry-free RBSD Addition	Carry-Free Binary Addition	Carry-Free Multiplication
Path delay	71.306 ns	4.784 ns	4.028	24.973 ns
Slice LUTs	2%	4%	1%	11%
LUT FF pairs	384	928	265	2223

7.2 Multiplication

Table 6 also lists the path delay and LUT usage for the carry-free implementation of multiplication. As can be seen, carry-free multiplication takes lesser time than even standard addition, it can be inferred that it takes significantly lesser time than standard multiplication.

Table 7. Path delay of individual modules in multiplication

Module	Path delay
Bin_to_RBSD68	3.607 ns
Carry_Free_Add_Bin33	4.028 ns
Carry_Free_Add_Bin66	4.028 ns
Carry_Free_Add_Bin132	4.028 ns
BSD_to_RBSD68	4.028ns
BSD_to_RBSD132	4.028 ns
add_RBSD_69	4.784 ns
add_RBSD_133	4.784 ns
BSD_to_Bin33	12.824 ns

Table 7 lists the path delays of the individual modules in the Carry-Free Multiplier, in ascending order of time taken. It can be observed that the most time-consuming module is the BSD_to_Bin33 module, which involves carry-full operation. All carry-free operations clearly take less than a third of the time taken by a simple BSD to binary conversion. The advantage of carry-free operation is highlighted in this result.

8. ANALYSIS

8.1 Use of 2's Complement binary numbers

Throughout the project, binary numbers have been assumed to be positive. In this case, there is an increase in the bit size from *n*-bits of the binary number to *2*(n+1)* bits of the RBSD number (*n+1* sign bits and *n+1* value bits). Here, there are n+1 bits in each of sign and value since every 2 bits of the binary number are required to make 1 RBSD bit. However, the incorporation of negative numbers, i.e. the possible use of binary numbers as being in their 1's complement or 2's complement forms, has not been explored.

The good news here is that this incorporation is particularly easy. If the binary number being used is assumed to be in its 2's complement form, then there shall be no increase in number of bits in its conversion from binary to RBSD. The binary number already has an extra bit included. If the number is negative, the most significant bit of its RBSD form shall inevitably be -1. Thus, Binary to RBSD conversion shall then entail only the doubling of the number of bits, one set each for sign and value. The number of bits would increase to just *2*n.*

8.2 Combined table of Carry Free Addition to produce RBSD numbers

The field of RBSD numbers is not closed under addition. Hence, the sum of two RBSD numbers, either carry-free or with carry propagation, shall result in a BSD number that is not necessarily RBSD. Since many applications require furthur use of the sum (for example, computing sum as an intermediate step in multiplication), it is imperative to then convert the sum into RBSD before using it elsewhere.

Fortunately, the conversion of a BSD number to RBSD is combinational, and hence shall not contribute much to the delay of the circuit. Thus, every time Carry-Free addition has taken place, a module for the corresponding bit size has been attached to its output that functions as a BSD-to-RBSD converter.

It can be argued that a Carry-Free Adder and the subsequent BSD-to-RBSD converter can be combined to make one single truth table. After all, we combined the tables for Binary-to-RBSD conversion and Carry-Free Addition of RBSD numbers to make the table for Carry-Free Addition of Binary Numbers. So a combined table of Carry-Free Addition (Binary or RBSD) and BSD-to-RBSD conversion could prove to make a faster circuit. This could also provide binary-to-binary addition, with the major intermediate step of addition being carry-free.

However, making such a concise table has many hurdles. It can be observed that the BSD-to-RBSD converter takes 4 consecutive digits of the input BSD number, and produces one output digit at the most significant digit position. It can also be observed that a Carry-Free Adder takes 2 consecutive RBSD digits from each RBSD input, or 3 consecutive binary bits from each binary input, to produce one output digit at the most significant position. Since these act as inputs to the BSD-to-RBSD Converter, 5 consecutive input RBSD digits in case of RBSD Carry-Free Adder, and 6 consecutive bits in case of Binary Carry-Free Adder are required to produce one output digit at the most significant position. This shall result in a table having *3^5* = 243 rows in case of RBSD Adder, and *2^6* = 64 rows in case of Binary adder. It is practically infeasible, and quite unneccesary, to manually construct a Sum-of-Products or Product-of-Sum form for the combined truth table.

8.3 Karatsuba Multiplication with carry-free addition incorporated

8.3.1 Multiplication of BSD numbers

The first step of Karatsuba multiplication involves splitting the input number into two parts. In case of BSD numbers, however, simply splitting the number at the middle is mathematically incorrect, and shall lead to erroneous results. Moreover, mathematically sound splitting involves carry propagation, which opposes our goal of fast computation.

8.3.2 Carry-full 34-bit BSD-to-Binary converter

As seen from Table 7, the two 34-bit RBSD-to-Binary converters used in the multiplier (see Figure 2) are the major contributors to the path delay of this circuit, since they involve carry propagation.

Since the conversion from Binary to RBSD in done so as to eliminate carry propagation, it can be expected that the opposite direction would involve carry propagation. Unfortunately, it cannot be avoided. In the future, if a method is devised to eliminate the need for carry propagation in this conversion, then instead of using it here that method should directly be applied for the addition of two numbers.

It could be argued that the use of two 17-bit RBSD-to-Binary converters can help decrease the path delay. However, this would involve splitting the RBSD number, which, as discussed above, results in some carry propagation of its own. However, it is to be seen whether the overall path delay is still lesser.

8.3.3 Multiplication of integers

Ultimately, multiplication operation is performed using integer arithmetic, which involves carry propagation. It can be argued that the numbers can be split into two 17-bit numbers and integer multiplication can be carried out over them instead of the 34-bit numbers. However, this is equivalent to adding one more level to the Karatsuba multiplication being carried out. This shall result in thrice the number of multiplications involved. It needs to be checked through experimentation whether this incurs lesser delay.

8.3.4 Multiplication of binary numbers with more than 64 bits

The original design was meant for 64-bit binary numbers. But since the conversion from binary to RBSD requires an extra bit, and there are two such conversions to be done within the algorithm (see Figure 2), the inputs were changed to 66 bits. The extra two bits can be set to 0 to add two n-bit numbers, where $n <= 64$. For $n > 64$, each input number could be split into the lowest 64 bits and the rest. The Karatsuba algorithm can then be recursively employed on the two parts of the numbers.

9. Conclusions

This paper details the conversions between binary numbers, binary signed digits, and RBSD numbers [9]. Detailed circuit diagrams and logic circuits have been designed for the carry-free implementation of addition and multiplication. These implementations are compared with standard addition in terms of path delay and memory, and their advantage is highlighted. The implementations have also been analyzed for flaws and improvements

Acknowledgements

The author would like to thank Sujoy Sinha Roy and Prof. Ingrid Verbauwhede from KU Leuven, Belgium, for the opportunity to collaborate with them for this work.

References

[1] P. Kogge and H. Stone (1973), "A parallel algorithm for the efficient solution of a general class of recurrences", *IEEE Transactions on Computers (T-C)*, vol. 22, pp. 786–793

[2] B. Parhami (2000), *Computer Arithmetic – Algorithms and Hardware Designs*. Oxford University Press

[3] A. Avizienis (1961), "Signed-digit number representations for fast parallel arithmetic," I*RE Transactions on Electronic Computers*, vol. 10, no. 3, pp. 389–400

[4] B. Parhami (1990), "Generalized signed-digit number systems: A unifying framework for redundant number representations," *IEEE Transactions on Computers (T-C)*, vol. 39, no. 1, pp. 89–98

[5] F. Kharbash, G. M. Chaudhry (2007), "Reliable Binary Signed Digit Number Adder Design", *IEEE Computer Society Annual Symposium on VLSI*, pp 479-484

[6] J. Moskal, E. Oruklu and J. Saniie (2007), “Design and Synthesis of a Carry-Free Signed-Digit Decimal Adder”, *IEEE International symposium on Circuits and Systems*, pp1089-1092

[7] R. Rani, N. Sharma, L. K. Singh (2009), “Fast Computing using Signed Digit Number System” *IEEE proceedings of International Conference on Control, Automation, Communication And Energy Conservation* - 2009

[8] K. Schneider, A. Willenbucher (2014), “A New Algorithm for Carry-Free Addition of Binary Signed-Digit Numbers”, *IEEE 22nd International Symp. Field-Programmable Custom Computing Machines*.

[9] B. Parhami, (1988) “Carry-free addition of recoded binary signed-digit numbers,” *IEEE Transactions on Computers (T-C)*, vol. 37, no. 11, pp. 1470–1476.

[10] M. Joye and S.-M. Yen (2000), “Optimal left-to-right binary signed-digit recoding,” *IEEE Transactions on Computers (T-C)*, vol. 49, no. 7, pp. 740–748

[11] J. U. Ahmed, A. A. S. Awwal (1993), "Multiplier design using RBSD number system", *Proceedings of the 1993 National Aerospace and Electronics Conference*, vol. 1, pp. 180-184

[12] A.K. Cherri, M.S. Alam (1998), "Recoded and nonrecoded trinary signed-digit multipliers designs using redundant bit representations", *Aerospace and Electronics Conference 1998. NAECON 1998. Proceedings of the IEEE 1998 National*, pp. 505-512, 1998, ISSN 0547-3578.

[13] Rajashekhar, T.N. and Kal, O. (1990), “Fast Multiplier Design using Redundant Signed-Digit Numbers”, *International Journal of Electronics*, vol .69, no. 3, pp–359-368

[14] A. Karatsuba and Yu. Ofman (1962). "Multiplication of Many-Digital Numbers by Automatic Computers". *Proceedings of the USSR Academy of Sciences.* 145: 293–294. Translation in the academic journal Physics-Doklady, 7 (1963), pp. 595–596

[15] A.A.S. Awwal and J.U. Ahmed (1993), “Fast carry free adder design using QSD number system,” *Proceedings of the IEEE 1993 National Aerospace and Electronic Conference*, vol 2, pp 1085-1090

[16] Reena Rani, L.K. Singh and Neelam Sharma (2010), “A Novel design of High Speed Adders Using Quaternary Signed Digit Number System,” *International Journal of Computer and Network Security (IJCNS)*, Vol. 2, No. 9, pp.62-66

[17] Neha W. Umredkar, M. A. Gaikwad (2013), “Review of Quaternary Adders in Voltage Mode Multi-Valued Logic”, *International Journal of Computer Applications* (0975–8887) “Recent Trends in Engineering Technology-2013”.

[18] Sachin Dubey, Reena Rani (2013) “VLSI Implementation of Fast Addition using Quaternary Signed Digit Number System”, *IEEE International Conference on Emerging Trends in Comp*...ICECCN 2013.

[19] O. Ishizuka, A. Ohta, K. Tannno, Z. Tang, D. Handoko (1997), “VLSI design of a quaternary multiplier with direct generation of partial products,” Proceedings of the 27th International Symposium on Multiple-Valued Logic, pp. 169-174

[20] Satyendra R.P. Raju Datla, Mitchell A. Thornton (2010), "Quaternary Voltage-Mode Logic Cells and Fixed-Point Multiplication Circuits", *Multiple-Valued Logic (ISMVL) 2010 40th IEEE International Symposium on*, pp. 128-133, 2010, ISSN 0195-623X.

[21] Neelam Sharma, B. S. Rai and Arun Kumar (2006), “Design of RBSD Adder and Multiplier Circuits for High Speed Arithmetic Operations and Their Timing Analysis”, Special Russian Issue: *Advances in* Computer *Science and Engineering*, *Research in Computing Science*, pp. 243-25

Author

Vikram Voleti is currently a researcher at IIIT Hyderabad. He has worked on this problem at KU Leuven, Belgium.