

**COEN 6501 Project Fall 2021 Specification**

Jun Huang, Dawei Zuo, Yuelin Yao, and Xuesi Feng

Department of Electrical and Computer Engineering, Concordia University

COEN 6501: Digital Design and Synthesis

Dr. Marwan Ammar

December 6th, 2021

## Table of Contents

<b>Introduction</b>	<b>7</b>
Heading II . . . . .	7
Heading III . . . . .	7
<b>Project Requirement</b>	<b>8</b>
Heading II . . . . .	8
Heading III . . . . .	8
<b>Carry Select Adder</b>	<b>9</b>
Full Adder . . . . .	9
Two to one multiplexer . . . . .	10
4-bit Ripple Carry Adder . . . . .	11
4-bit Carry Select Adder . . . . .	12
16-bit Carry Select Adder . . . . .	14
24-bit Carry Select Adder . . . . .	15
24-bit CSA Incrementor . . . . .	16
<b>Multiplication in Three Operands</b>	<b>17</b>
Radix-4 Booth Algorithm Logic in Details . . . . .	17
Example in Signed Number Multiplication . . . . .	17
Example in Unsigned Number Multiplication . . . . .	18
8-bit Radix-4 Booth Multiplier Circuit Implementation . . . . .	19
Overall Circuit Design and RTL Description . . . . .	19
Blocks Design . . . . .	19
8-bit Triple Operands Multiplier Circuit Implementation . . . . .	23
16-bit Triple Operands Multiplier Circuit Implementation . . . . .	25
<b>Overflow Handling</b>	<b>26</b>
Heading II . . . . .	26
Heading III . . . . .	26
<b>End Flag Generator</b>	<b>27</b>

<b>Non-pipelined Implementation</b>	<b>28</b>
Operating Circuit . . . . .	28
Output Process . . . . .	28
<b>Pipelined Implementation</b>	<b>30</b>
8-bit Operands Implementation with Negation Overflow Handling . . . . .	30
8-bit Operands Implementation with Output Separation Overflow Handling . . . . .	30
16-bit Operands Implementation and Simulation . . . . .	33
<b>Synthesis and Analysis of the Arithmetic Circuit</b>	<b>34</b>
Heading II . . . . .	34
Heading III . . . . .	34
<b>References</b>	<b>35</b>

## List of Figures

1	Synthesized RTL Diagram of Full Adder Block . . . . .	10
2	Simulation Wave Diagram of Full Adder Block . . . . .	10
3	Synthesized RTL Diagram of 2-to-1 Multiplexer Block . . . . .	11
4	Simulation Wave Diagram of 2-to-1 Multiplexer Block . . . . .	11
5	Synthesized RTL Diagram of 4-bit Ripple Carry Adder Block . . . . .	12
6	Simulation Wave Diagram of 4-bit Ripple Carry Adder Block . . . . .	12
7	Synthesized RTL Diagram of 4-bit Carry Select Adder Block . . . . .	13
8	Simulation Wave Diagram of 4-bit Carry Select Adder Block . . . . .	14
9	Synthesized RTL Diagram of 16-bit Carry Select Adder Block . . . . .	14
10	Simulation Wave Diagram of 16-bit Carry Select Adder Block . . . . .	15
11	Synthesized RTL Diagram of 24-bit Carry Select Adder Block . . . . .	15
12	Simulation Wave Diagram of 24-bit Carry Select Adder Block . . . . .	16
13	Synthesized RTL Diagram of 24-bit Incrementor Block . . . . .	16
14	Simulation Wave Diagram of 24-bit Incrementor Block . . . . .	16
15	The Decoded Code Blocks of the Signed Multiplier . . . . .	18
16	Process of the Algorithm for Signed Number . . . . .	18
17	The Decoded Code Blocks of the Unsigned Multiplier . . . . .	19
18	Process of the Slgorithm for Unisgned Number . . . . .	19
19	Synthesized RTL Diagram of 8-bit Radix-4 Booth Multiplier . . . . .	20
20	Synthesized RTL Diagram of Complement Generator . . . . .	20
21	Simulation Wave Diagram of Complement Generator . . . . .	21
22	Synthesized RTL Diagram of Booth Stage 0 Block . . . . .	21
23	Simulation Wave Diagram of Booth Stage 0 Block . . . . .	22
24	Synthesized RTL Diagram of Booth Stage 1 Block . . . . .	22
25	Simulation Wave Diagram of Booth Stage 1 Block . . . . .	23
26	Synthesized RTL Diagram of Triple 8-bit Operands Multiplier Circuit . . . . .	24
27	Simulation Wave Diagram of Triple 8-bit Operands Multiplier Circuit . . . . .	24
28	Synthesized RTL Diagram of Triple 16-bit Operands Multiplier Circuit . . . . .	25
29	Simulation Wave Diagram of Triple 16-bit Operands Multiplier Circuit . . . . .	25
30	ASMD Chart of the FSM of Different End Flag Generator . . . . .	27
31	Synthesized RTL Diagram of Non-pipelined Operating Circuit . . . . .	28

32	Simulation Wave Diagram of Non-pipelined Operating Circuit . . . . .	29
33	Synthesized RTL Diagram of Non-pipelined Circuit . . . . .	29
34	Simulation Wave Diagram of Non-pipelined Circuit . . . . .	29
35	Simulation Wave Diagram of the 8-bit Pipelined ALU Circuit with Negation Output . . . . .	31
36	Synthesized RTL Diagram of the 8-bit Operands Pipelined ALU Circuit and the Arithmetic Operation Block . . . . .	31
37	Simulation Wave Diagram of the 8-bit Pipelined ALU Circuit with Separation Output . . . . .	32
38	Synthesized Diagram of the 8-bit Pipelined ALU Circuit with Separation Output . . . . .	32
39	Simulation Wave Diagram of the 16-bit Pipelined ALU Circuit with Negation Output . . . . .	33
40	Synthesized RTL Diagram of 16-bit Radix-4 Booth Multiplier . . . . .	36

**List of Tables**

1	Full Adder Truth Table . . . . .	9
2	2-to-1 Multiplexer Truth Table . . . . .	11
3	Code Table of Radix-4 Booth Algorithm . . . . .	17
4	A Visual Representation Table of the Pipelined Processe with Negation Overflow Handling . . . . .	30
5	A Visual Representation Table of the Pipelined Processing with Output Separation Overflow Handling . . . . .	32

## Introduction

Cervantes Saavedra, Raffel, and Wilson (1999) En un lugar de la Mancha, de cuyo nombre no quiero acordarme, no ha mucho tiempo que vivía un hidalgo de los de lanza en astillero, adarga antigua, rocín flaco y galgo corredor.

### **Heading II**

Una olla de algo más vaca que carnero, salpicón las más noches, duelos y quebrantos los sábados, lantejas los viernes, algún palomino de añadidura los domingos, consumían las tres partes de su hacienda.

### **Heading III**

El resto della concluían sayo de velarte, calzas de velludo para las fiestas, con sus pantuflos de lo mismo, y los días de entresemana se honraba con su vellorí de lo más fino.

**Heading IV.** Tenía en su casa una ama que pasaba de los cuarenta, y una sobrina que no llegaba a los veinte, y un mozo de campo y plaza, que así ensillaba el rocín como tomaba la podadera. Frisaba la edad de nuestro hidalgo con los cincuenta años; era de complexión recia, seco de carnes, enjuto de rostro, gran madrugador y amigo de la caza.

**Heading V.** Quieren decir que tenía el sobrenombre de Quijada, o Quesada, que en esto hay alguna diferencia en los autores que deste caso escriben; aunque por conjeturas verosímiles se deja entender que se llamaba Quijana.

## **Project Requirement**

**Heading II**

*Heading III*

**Heading IV.**

*Heading V.*

### Carry Select Adder

During the implementation of a multiplier, adders are needed. Different adder choices can have different effects on the delay and area. The outputs of ripple carry adder rely on the output carry of lower levels, so the RCA has an extremely long output chain and path. A carry select adder has a pair of Ripple Carry Adder performing the addition of a chunk of the two operands and a multiplexer to select the correct sum and carry out from the two RCAs. Compared to RCA, the carry select adder is a more efficient parallel adder.

In carry select adder, both sum and carry outputs are calculated for two alternatives: the input carry  $C_{in}$  ‘0’ and ‘1’. Once the input carry is loaded, the correct calculation is chosen by a multiplexer to produce the desired output. Instead of waiting for the carry to calculate the sum, the sum will be correctly output as soon as the input carry delivered. The time used to compute the sum is then reduced that results in a good improvement in speed. Also, it can be formed into higher bit adders by cascading. So that extending the algorithm will be easier with the usage of CSAs.

The following will introduce the structures of the desired carry select adders used in this project.

### Full Adder

The full adder is the fundamental digital component of various arithmetic logic unit, the circuit, which is composed of **XOR** gate, **AND** gate and **OR** gate, adds three inputs and produces two outputs. The full adder differs from the half adder in that the full adder has an input carry  $C_{in}$ , so that it can handle the carry in from the lower bit and output its carry out.

Multiple one-bit full adders can be cascaded to obtain a multi-bit full adder, which is used as a method to design subsequent circuits. The full adder is more widely used due to the feature that it can perform the addition of three bits. But at the same time, it requires additional gates. As a result, its delay increases. The truth table for the full adder is shown in Table 1.

**Table 1**

*Full Adder Truth Table*

A	B	Carry <sub>in</sub>	Carry <sub>out</sub>	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

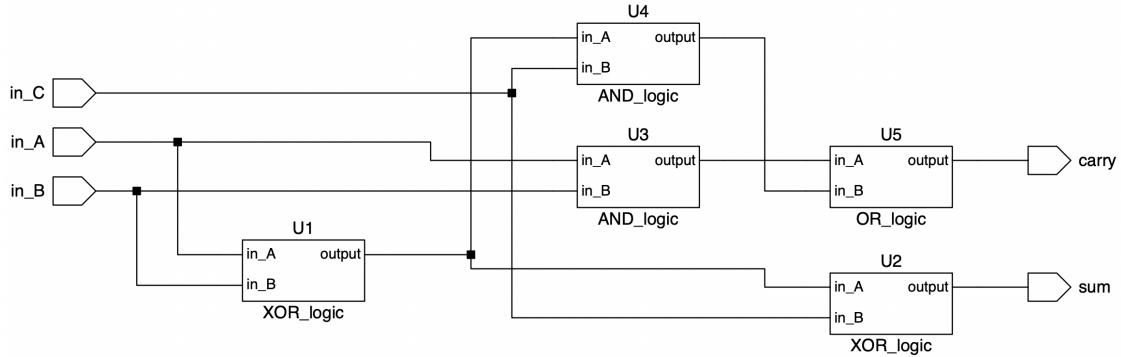
The equation for the full adder is defined at Expression (1).

$$\begin{aligned} \text{Sum} &= A \oplus B \oplus C_{in} \\ C_{out} &= (A \bullet B) + (C_{in} \bullet (A \oplus B)) \end{aligned} \quad (1)$$

The circuit for the full adder is shown in Figure 1.

**Figure 1**

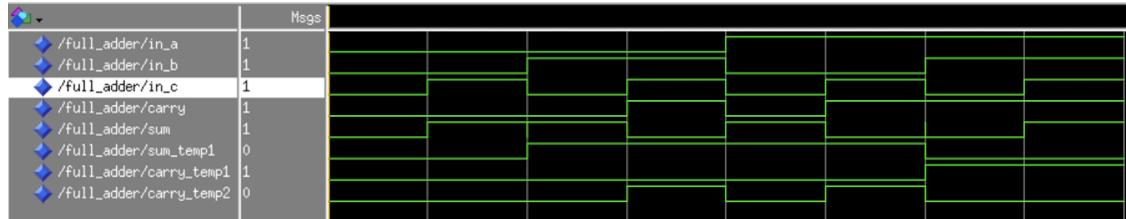
*Synthesized RTL Diagram of Full Adder Block*



The simulation results for the full adder are shown in Figure 2.

**Figure 2**

*Simulation Wave Diagram of Full Adder Block*



### Two to one multiplexer

The 2-to-1 multiplexer is a selector that has a switch to control the input, the circuit which consists of **AND** gate, **OR** gate and **NOT** gate. For a 2-to-1 multiplexer, the inputs are  $A$  and  $B$ ,  $Sel$  is the select signal and  $Z$  is the output. Depending on the select signal, the output is connected to either of the inputs. If  $Sel = 0$ , then the output will be switched to input  $a$ , whereas if  $Sel = 1$ , then the output will be switched to input  $b$ . The truth table is shown in Table 2.

The equation for the multiplexer is defined at Expression (2)

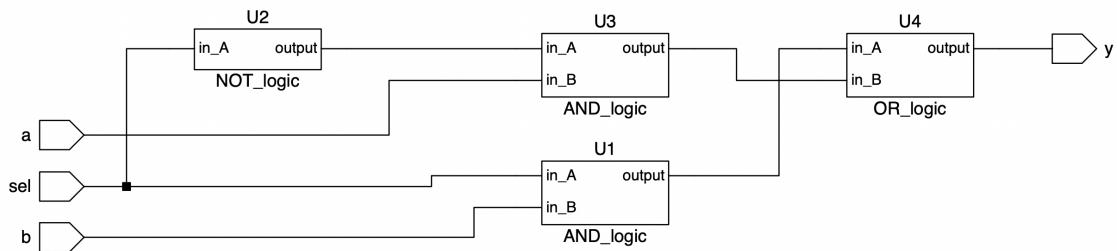
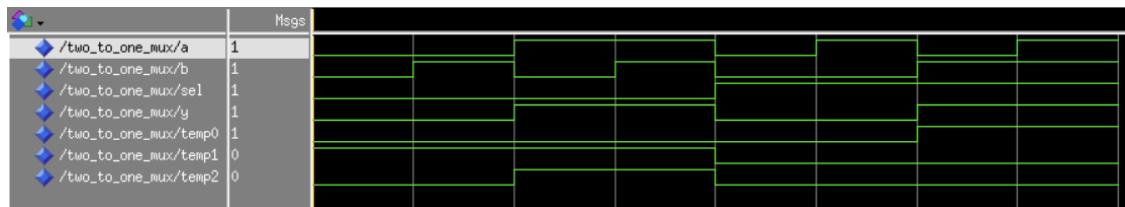
$$Y = (\overline{Sel} \bullet A) + (Sel \bullet B) \quad (2)$$

**Table 2***2-to-1 Multiplexer Truth Table*

A	B	Select	Output
0	-	0	0
1	-	0	1
-	0	1	0
-	1	1	1

The circuit for the multiplexer is shown in Figure 3.

The simulation results for the multiplexer are shown in Figure 4.

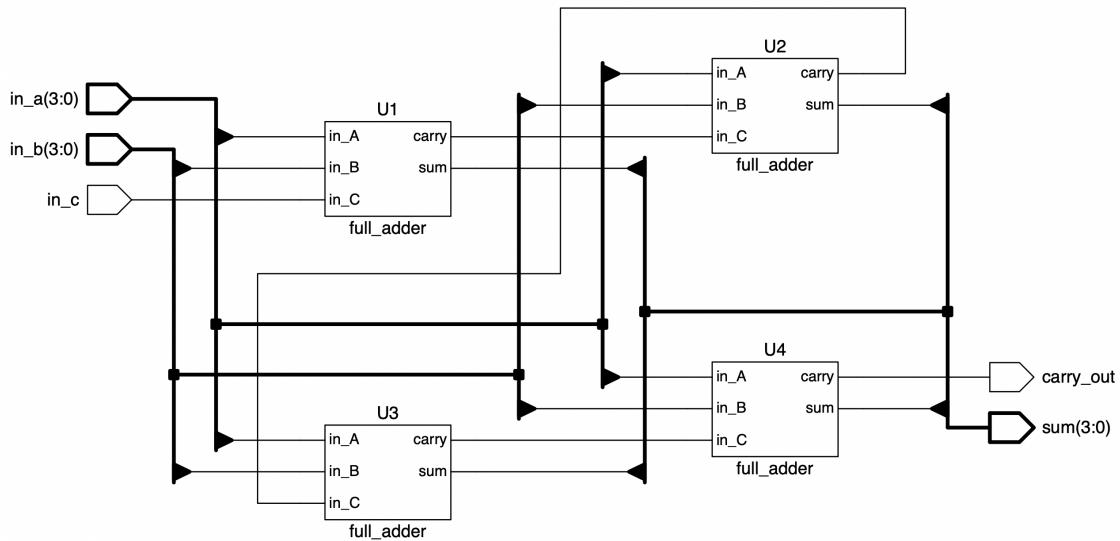
**Figure 3***Synthesized RTL Diagram of 2-to-1 Multiplexer Block***Figure 4***Simulation Wave Diagram of 2-to-1 Multiplexer Block*

#### 4-bit Ripple Carry Adder

In order to build n-bit carry select adders which are suitable for this project, a 4-bit ripple carry adder block is the basic component. A ripple carry adder is cascaded in parallel by multiple full adder circuits, in which the carry out of each full adder is the carry in of the succeeding next most significant full adder. Four full adders are tied together to build the 4-bit ripple carry adder block for this project. Where  $A$  and  $B$  are 4-bit inputs, sum is the addition output of  $A$  and  $B$ ,  $Carry_{out}$  is the output carry which depends on  $C_{in}$ ,  $C_1$ ,  $C_2$ ,  $C_3$ . The circuit for the 4-bit ripple carry adder block is shown in Figure 5.

**Figure 5**

*Synthesized RTL Diagram of 4-bit Ripple Carry Adder Block*

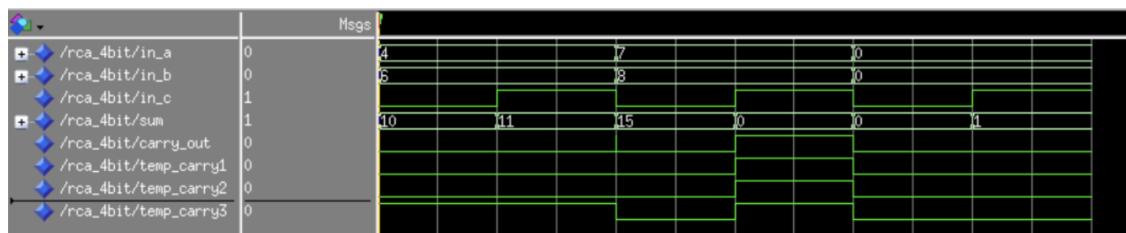


The selected simulation cases are shown below and the simulation results for the 4-bit ripple carry adder are shown in Figure 6.

- A general purpose test addition
- Overflow test
- Zeros test

**Figure 6**

*Simulation Wave Diagram of 4-bit Ripple Carry Adder Block*



#### 4-bit Carry Select Adder

A basic building block size of carry select adder is four. A 4-bit carry select adder consists of two parallel 4-bit ripple carry adders and 2-to-1 multiplexers to perform the calculation twice. One of the 4-bit RCA block assumes that the input carry is 0 (RCA\_0), the other assumes that the input carry is 1 (RCA\_1). After the two results

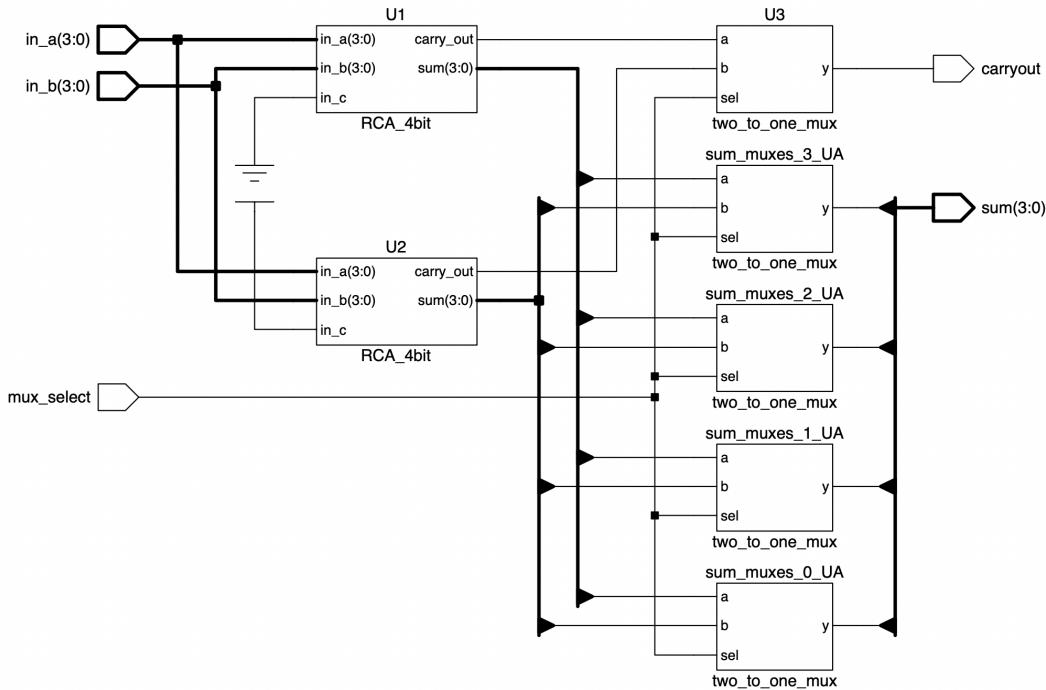
are calculated, the correct sum, as well as the correct carry out, is then selected with the multiplexer once the correct carry in is known. The delay equation for the 4-bit carry select adder is defined below:

$$T_{CSA} = T_{mux} + 4 \times T_{full\_adder} \quad (3)$$

The circuit for the 4-bit carry select adder is shown in Figure 7.

**Figure 7**

*Synthesized RTL Diagram of 4-bit Carry Select Adder Block*

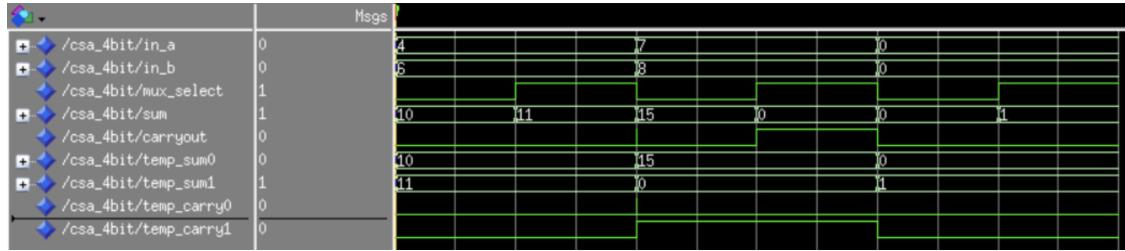


If the input carry  $C_{in}$  from the lower level is 0, the output carry of the RCA\_0 is selected as the output carry of this 4-bit CSA block. If the input carry  $C_{in}$  from the lower level is 1, the output carry of the RCA\_1 is selected as the output carry. At the same time  $C_{in}$  is used as the selection signal of 2-to-1 multiplexer to control whether the output of S3 to S0 comes from the RCA\_0 or the RCA\_1.

The simulation results for the 4-bit carry select adder are shown in Figure 8.

**Figure 8**

*Simulation Wave Diagram of 4-bit Carry Select Adder Block*



### 16-bit Carry Select Adder

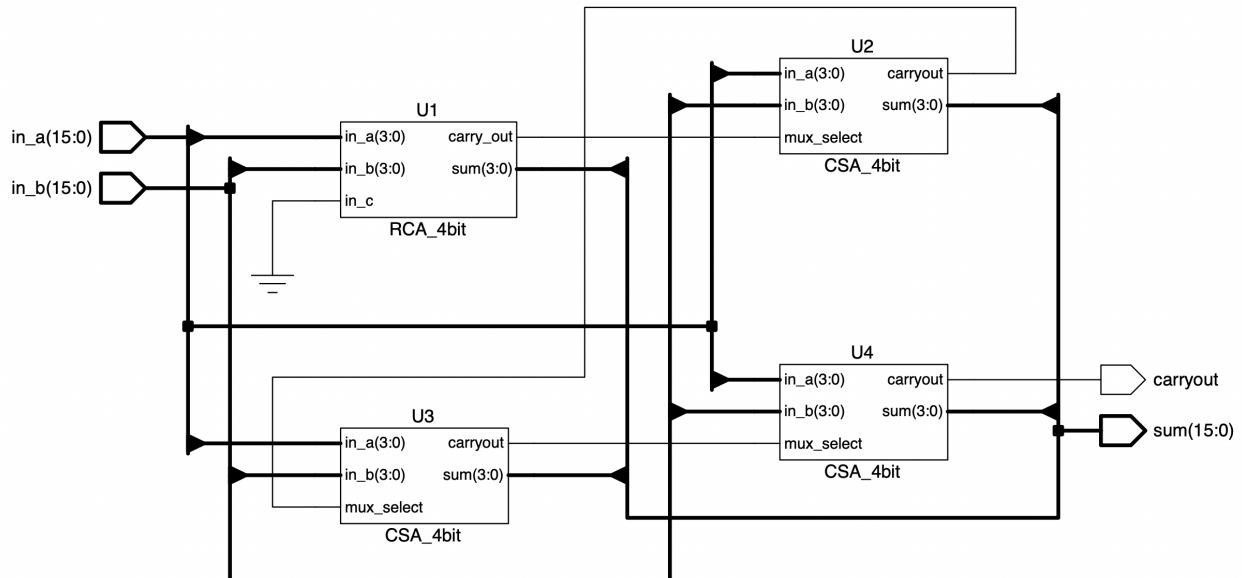
In this project, 16-bit carry select adder is used. A 16-bit carry select adder can be created using three 4-bit CSA blocks and one 4-bit RCA blocks. The first block is a 4-bit RCA, the inputs are two binary numbers from multiplier, so that there is no input carry and the  $C_{in}$  can be set to 0. Then the delay of this adder will be the delay of the four full adders, plus the delay of the three MUXs. The delay equation for the 16-bit carry select adder is defined below:

$$T_{CSA} = 3 \times T_{mux} + 4 \times T_{full\_adder} \quad (4)$$

The circuit for the 16-bit carry select adder is shown in Figure 9.

**Figure 9**

*Synthesized RTL Diagram of 16-bit Carry Select Adder Block*



The simulation results for the 16-bit carry select adder are shown in Figure 10.

**Figure 10**

*Simulation Wave Diagram of 16-bit Carry Select Adder Block*



## 24-bit Carry Select Adder

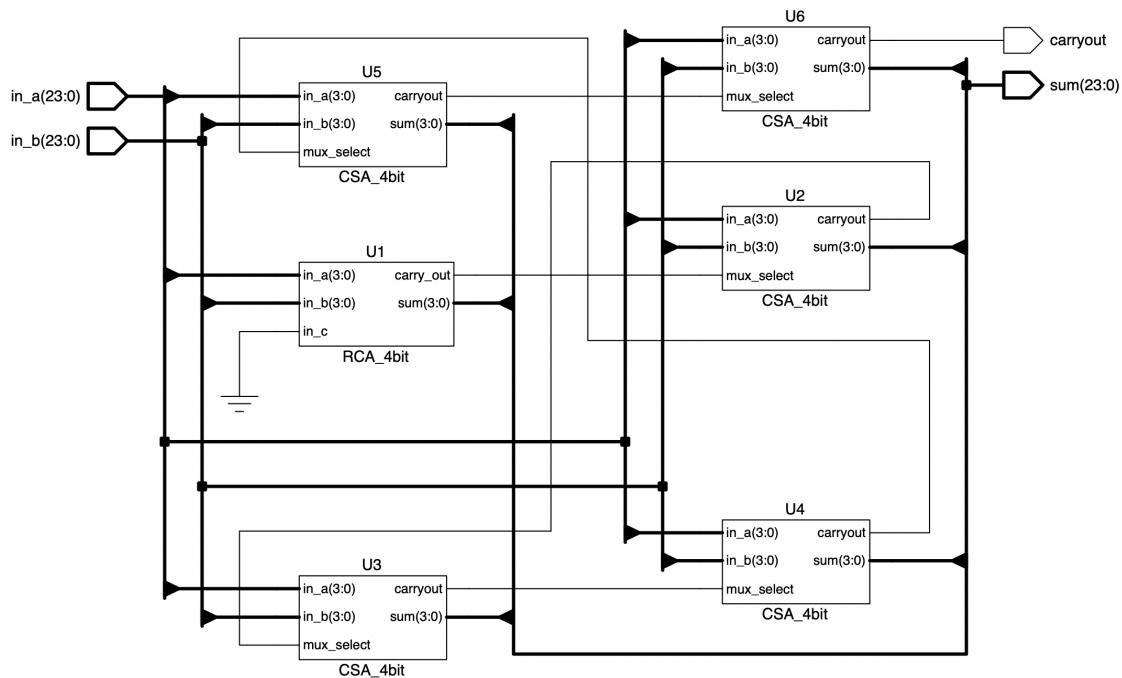
Same principle as a 16-bit CSA, a 24-bit CSA is built up with five 4-bit CSA blocks and one 4-bit RCA block. Also, the first block is 4-bit RCA, the inputs are two binary numbers from multiplier, so that there is no input carry and the  $C_{in}$  can be set to 0. The delay equation for the 16-bit carry select adder is defined below:

$$T_{CSA} = 5 \times T_{mux} + 4 \times T_{full\_adder} \quad (5)$$

The circuit for the 24-bit carry select adder is shown in Figure 9.

**Figure 11**

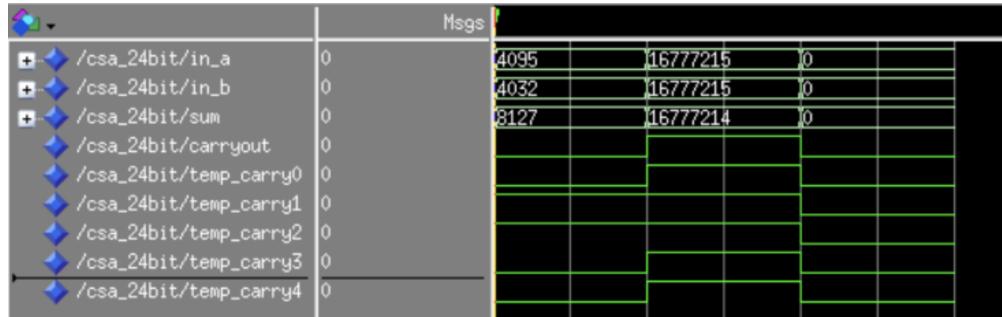
*Synthesized RTL Diagram of 24-bit Carry Select Adder Block*



The simulation results for the 24-bit carry select adder are shown in Figure 12.

**Figure 12**

*Simulation Wave Diagram of 24-bit Carry Select Adder Block*

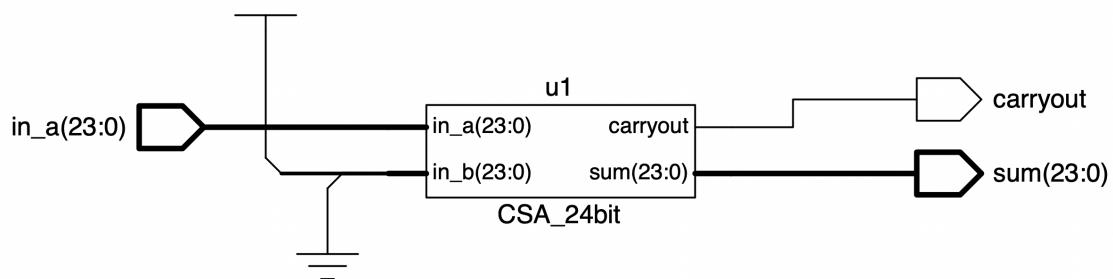


#### 24-bit CSA Incrementor

The incrementor used in this project were designed on the basis of the CSA. The principle is to define the input B of the CSA as a constant with the lowest bit of 1 and the rest of 0. Due to different bit requirements, a N-bit incrementor can be implemented by N-bit CSA. A 24-bit incrementor is used in this project. The circuit for the 24-bit incrementor is shown in Figure 13.

**Figure 13**

*Synthesized RTL Diagram of 24-bit Incrementor Block*



The simulation results for the 24-bit incrementor are shown in Figure 14.

**Figure 14**

*Simulation Wave Diagram of 24-bit Incrementor Block*



### Multiplication in Three Operands

The very first component of the ALU should be the circuit that calculates the result of  $A^2 * B$ . Hence the multiplication of two 8-bit operands circuits should be designed first, then the circuit for multiplying the product of the square A with the B should be designed later.

In general, the multiplication of two 8 bits operands in the shift-and-add algorithm or the radix-2 booth algorithm requires 8 steps of calculating eight partial products and then adding them together. By using the modified booth algorithm which is also known as radix-4 booth algorithm, the number of the partial products can be reduced to  $2/n$  where n is the bit length of the operand.

Not only because it's faster, but also because it saves more area compared with the previous two algorithms. Hence the project chooses to implement the radix-4 booth algorithm as the multiplication component of the ALU.

There is an extra partial product the circuit should consider since the booth algorithm is designed for the signed number. In this case, the implemented algorithm requires  $2/n + 1$  partial product for the unsigned number to reach the final answer. This will be discussed in the following sections.

### Radix-4 Booth Algorithm Logic in Details

Booth algorithm calculates the partial product by examining the “Code Table” on the second operand, the multiplier. The table requires certain blocks of bits from the right side to the left side of the multiplier, and for each block, the table provides the partial product respectively. Radix-2 requires 2 bits while radix-4 requires 3 bits. This is how the radix-4 algorithm reduces the partial products to a half. In this manner, the overlaps will occur in the partial products, hence the algorithm will do subtraction according to the “Code Table”.

#### *Example in Signed Number Multiplication*

When the algorithm is used for calculating two 8-bit signed number *0b01010110* which is 86 in decimal, and *0b00100011* which is 35 in decimal, the multiplier will be decoded as Figure 15 is shown. Then all partial products can be derived from the code blocks and the “Code Table” which is presented in Table 3.

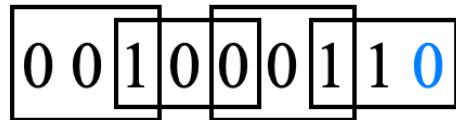
**Table 3**

*Code Table of Radix-4 Booth Algorithm*

Code Blocks	Partial Product
000/111	0
001/010	$1 * \text{multiplicand}$
011	$2 * \text{multiplicand}$
100	$-2 * \text{multiplicand}$
101/110	$-1 * \text{multiplicand}$

**Figure 15**

*The Decoded Code Blocks of the Signed Multiplier*



*Note.* The blue bit is an extra bit which is added on the right of the LSB of the multiplier for completing the first code block.

The algorithm takes four blocks of code from the right to the left and retrieves the corresponding product by shifting two bits more than the previous product. Then perform the addition. The process of the algorithm is described in Figure 16.

**Figure 16**

*Process of the Algorithm for Signed Number*

110	1 1 1 1 1 1 1 1	1 0 1 0 1 0 1 0 1 0	partial product 1
001	0 0 0 0 0 0	1 0 1 0 1 1 0 0 0	partial product 2
100	1 1 1 1 0 1 0 1 0 1 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	partial product 3
001	0 0 0 1 0 1 0 1 1 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	partial product 4
			0 0 0 0 1 0 1 1 1 1 0 0 0 0 0 1 0

*Note.* Bits in green color represent the product is extended to 16 bits. Bits in red color represent the shifted 2 bits leftward.

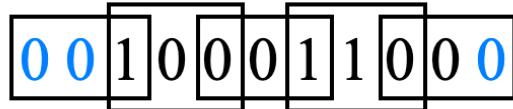
After the performance, the result  $0b0000101111000010$  is 3010 in decimal which is the product of 86 and 35. As can be observed at the table, subtraction can be done by adding the negation of the number which is represented by 2's complement.

#### **Example in Unsigned Number Multiplication**

Applying the algorithm to unsigned operands is a little bit different. Because the MSB of the operand is treated as a valid number rather than the sign, the operands should extend to 9-bit by adding a “0” to the MSB. Hence an extra partial product will be added to the product. For instance, multiplicand  $0b010010101$  which is 149 in decimal and multiplier  $0b011001100$  which is 204 in decimal can be operated in the process as Figure 18 and Figure 18 are shown.

**Figure 17**

*The Decoded Code Blocks of the Unsigned Multiplier*



*Note.* Two “0” are added to the MSB to complete the last code block. The second zero represent the sign bit of the operand which in this case will always be positive number.

**Figure 18**

*Process of the Slgorithm for Unisgned Number*

000	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	partial product 1
110	1 1 1 1 1 1 1 1 0 1 0 1 1 0 0 0	partial product 2
001	0 0 0 0 1 0 0 1 0 1 0 1 0 0 0 0	partial product 3
100	1 1 0 1 1 0 1 0 1 1 0 0 0 0 0 0	partial product 4
001	1 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0	partial product 5
<hr/>		
	0 1 1 1 0 1 1 0 1 0 1 1 1 1 0 0	

*Note.* Bits in green color represent the product is extended to 16 bits. Bits in red color represent the shifted 2 bits leftward.

### 8-bit Radix-4 Booth Multiplier Circuit Implementation

#### *Overall Circuit Design and RTL Description*

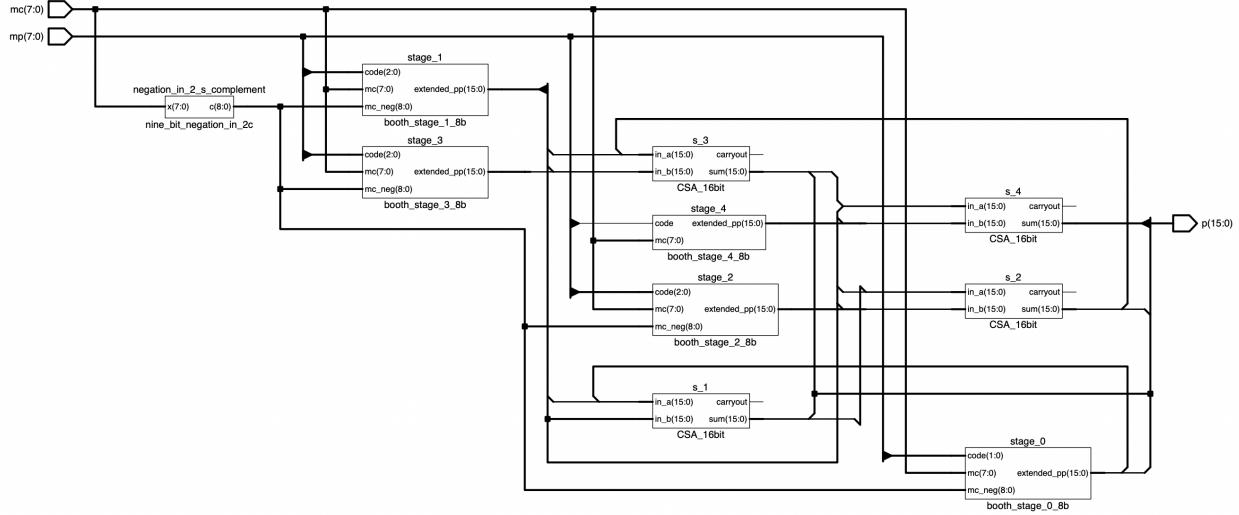
As the previous discussion, the booth multiplier component should contain the following blocks: 1. A 9-bit complement generator for the negation of the multiplicand; 2. Five booth stage units for 5 partial products; 3. Four 16-bit adders to sum up the partial products. Figure 19 presents the synthesized RTL diagram of the multiplier circuit.

The circuit will first calculate a 9-bit negation of operand multiplicand in 2’s complement. Then pass through all five booth stage blocks to get five 16-bit partial products. And finally sums those partial products up.

#### *Blocks Design*

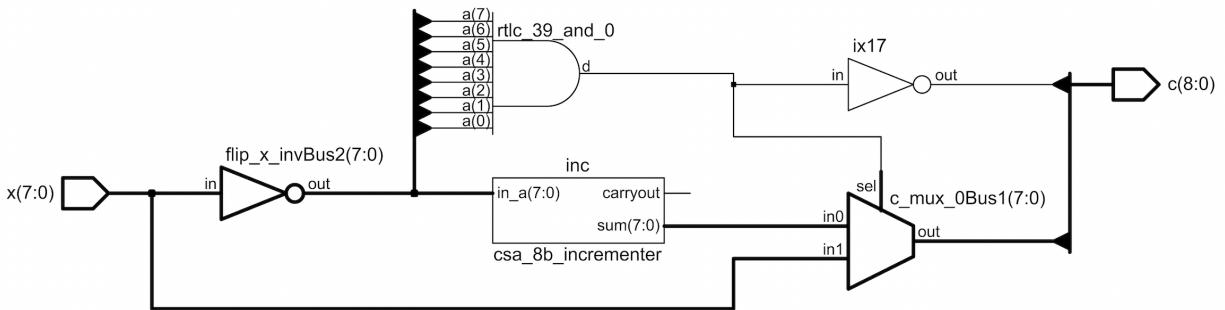
This section will discuss the design of the complement generator and five of the booth stages, the 16-bit CSA will be discussed in Section “Carry Select Adder”.

**The 9-bit Complement Generator for the Negation of the Multiplier.** Since the circuit uses negation addition to represent the subtraction, a complement generator should be introduced. The RTL diagram of the generator is shown in Figure 20. The logic of the generator that uses a 2-to-1 mux is straightforward as Expression

**Figure 19***Synthesized RTL Diagram of 8-bit Radix-4 Booth Multiplier*

(6) described. The simulation result is shown in Figure 21.

$$c = \begin{cases} concat(0, x), & \text{if } x = 00000000 \\ concat(1, (not x)) + 1, & \text{otherwise} \end{cases} \quad (6)$$

**Figure 20***Synthesized RTL Diagram of Complement Generator*

**Booth Stage 0.** Figure 22 and Expression (7) present the hardware implementation and the logic expression of the block. As the figure suggested, a booth stage block takes the 8-bit operand multiplicand and its 9-bit negation and two of the rightmost bits of the operand multiplier as input, and composes the 16-bit extended partial product as output by a 4-to-1 mux. The width of the *extended\_sign\_bits* will be 6. The simulation result is

**Figure 21**

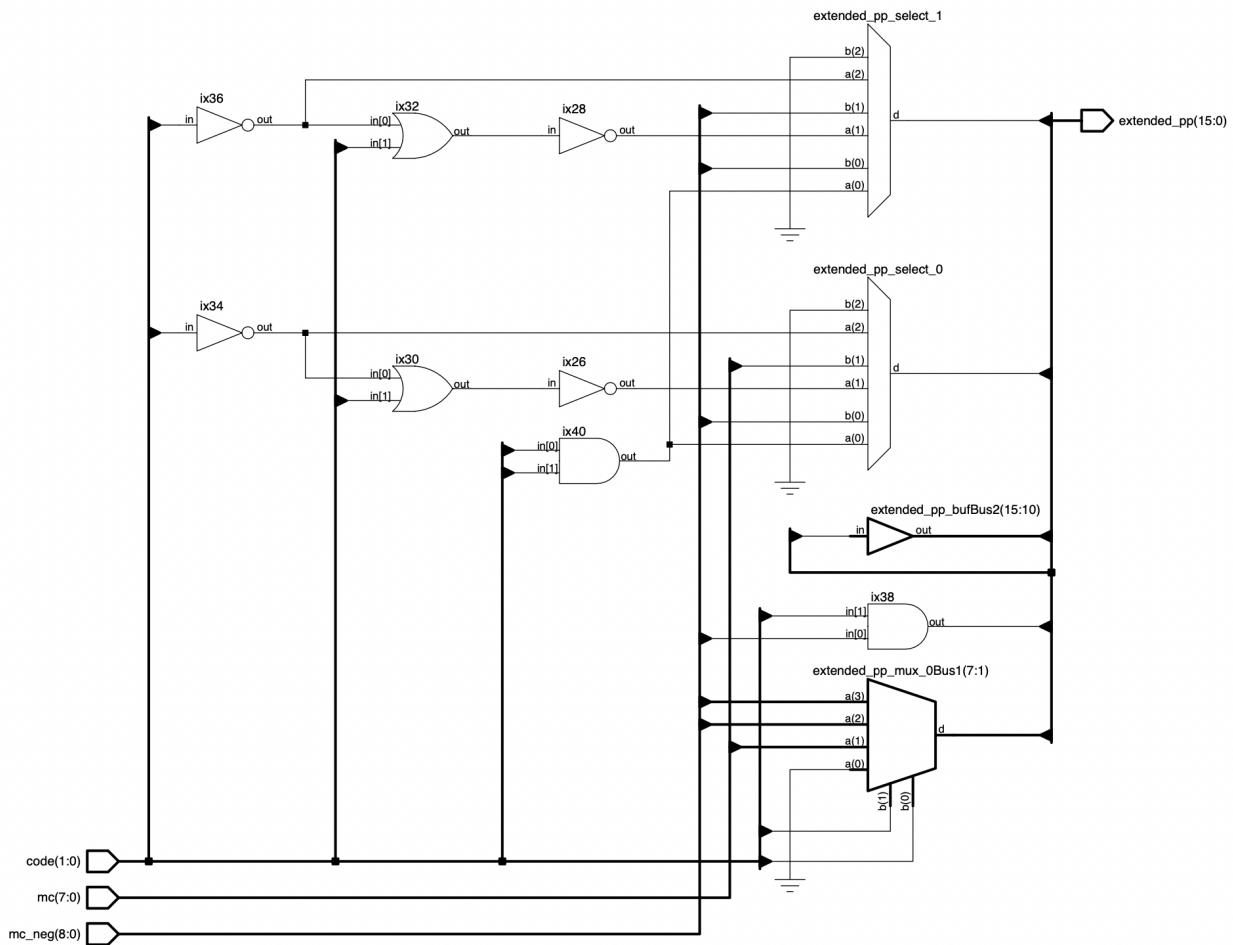
*Simulation Wave Diagram of Complement Generator*

	Msgs		
Data-	00000000	10110110	11111111
Data-	00000000	101001010	100000001
Data-	11111111	01001001	00000000
Data-	00000000	01001010	00000001

shown in Figure 23.

**Figure 22**

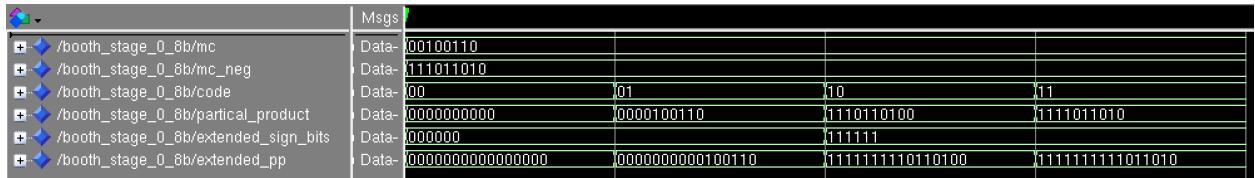
*Synthesized RTL Diagram of Booth Stage 0 Block*



$$\begin{aligned}
partial\_product &= \begin{cases} 0000000000, & \text{if } code = 00 \\ concat(00, mc), & \text{if } code = 01 \\ concat(mc\_neg, 0), & \text{if } code = 10 \\ concat(mc\_neg(8), mc\_neg), & \text{else } code = others \end{cases} \\
extended\_sign\_bits &= (others \Rightarrow partial\_product(9)) \\
extended\_pp &= concat(extended\_sign\_bits, partial\_product)
\end{aligned} \tag{7}$$

**Figure 23**

### *Simulation Wave Diagram of Booth Stage 0 Block*

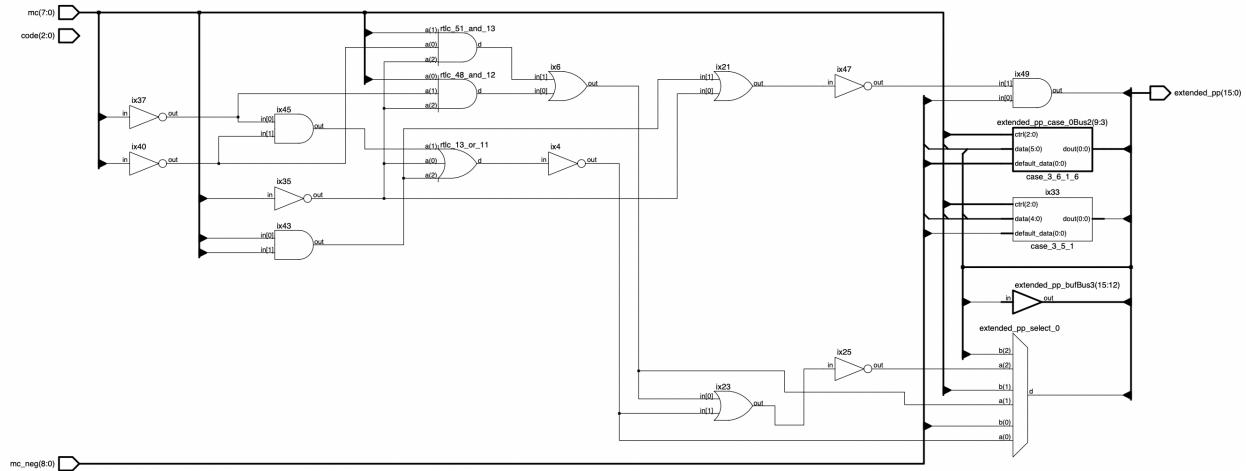


*Note.* Four code values were provided to simulate the *extended\_pp*.

**Booth Stage 1.** Figure 24 and Expression (8) present the hardware implementation and the logic expression of the block. Different from booth stage 0 block, this stage takes 3 bits from the operand multiplier. With considering the right shift during the algorithm, the width of the *extended\_sign\_bits* will be 4, and “00” will be added to the end. The simulation result is shown in Figure 25.

**Figure 24**

### *Synthesized RTL Diagram of Booth Stage 1 Block*



$$\begin{aligned}
partial\_product = & \begin{cases} 0000000000, & \text{if } code = 000|111 \\ concat(00, mc), & \text{if } code = 001|010 \\ concat(0, mc, 0), & \text{if } code = 011 \\ concat(mc\_neg, 0), & \text{if } code = 100 \\ concat(mc\_neg(8), mc\_neg), & \text{else } code = others \end{cases} \\
extended\_sign\_bits &= (others \Rightarrow partial\_product(9)) \\
extended\_pp &= concat(extended\_sign\_bits, partial\_product, 00)
\end{aligned} \tag{8}$$

**Figure 25***Simulation Wave Diagram of Booth Stage 1 Block*

(a) With Code Values: 000/001/010/011

Msgs	Data	Data	Data	Data
/booth_stage_1_8b/mc	00100110			
/booth_stage_1_8b/mc_neg	111011010			
/booth_stage_1_8b/code	000	001	010	011
/booth_stage_1_8b/partial_product	0000000000	0000100110		0001001100
/booth_stage_1_8b/extended_sign_bits	0000			
/booth_stage_1_8b/extended_pp	0000000000000000	0000000010011000		0000000100110000

(b) With Code Values: 100/101/110/111

Msgs	Data	Data	Data	Data
/booth_stage_1_8b/mc	00100110			
/booth_stage_1_8b/mc_neg	111011010			
/booth_stage_1_8b/code	100	101	110	111
/booth_stage_1_8b/partial_product	1110110100	1111011010		0000000000
/booth_stage_1_8b/extended_sign_bits	1111			0000
/booth_stage_1_8b/extended_pp	1111111011010000	1111111101101000		0000000000000000

*Note.* Eight code values were provided to simulate the *extended\_pp*.

**Booth Stage 2, 3, and 4.** Booth stage 2 and 3 blocks share the same idea of booth stage 1 except they shift more bit to the right. As for the booth stage 4 block, it only takes the MSB from the operand multiplier. Expression (9) shows its logic.

$$extended\_pp = \begin{cases} 0000000000000000, & \text{if } code = 0 \\ concat(mc, 00000000), & \text{else } code = others \end{cases} \tag{9}$$

### 8-bit Triple Operands Multiplier Circuit Implementation

Once the 16-bit product of  $A^2$  which is marked as *product\_aa* is calculated, it will then multiply with the 8-bit input *B*. To perform multiplication with a 16-bit operand and an 8-bit operand, the circuit divides the 16-bit

operand into two 8-bit operands. This is to reuse the 8-bit multiplier block that is designed before.

The product of multiplying the higher 8-bit of *product\_aa* and *B* will be marked as *product\_haa\_b*, and it will be extended to 24 bits by shifting 8 bits rightwards. The product of multiplying the lower 8-bit of *product\_aa* and *B* will be marked as *product\_laa\_b*, and it will be extended to 24 bits by adding 8 zeros to its left. Then adds those two extended 24-bit numbers together will be the result of  $A^2 * B$ .

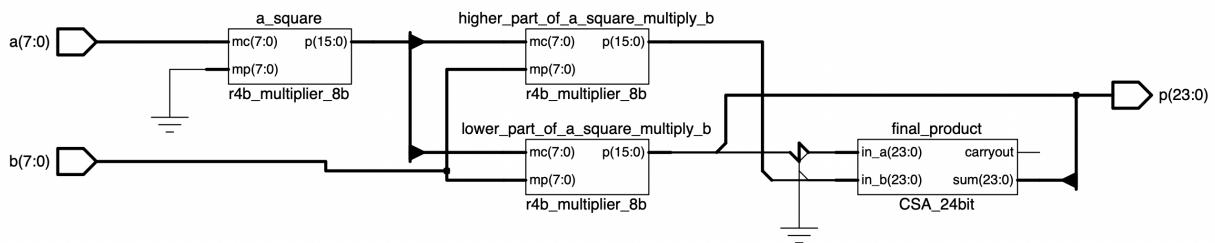
Figure 26 presents the RTL description of the circuit and Figure 27 presents the simulation result.

Expression (10) shows the arithmetic process of the  $A^2 * B$ .

$$\begin{aligned}
 aa &= a * a \\
 laa\_b &= aa[7,0] * b \\
 haa\_b &= aa[15,8] * b \\
 aab\_m &= concat(haa\_b, 00000000) \\
 aab\_l &= concat(00000000, laa\_b) \\
 p &= aab\_m + aab\_l
 \end{aligned} \tag{10}$$

**Figure 26**

Synthesized RTL Diagram of Triple 8-bit Operands Multiplier Circuit



**Figure 27**

Simulation Wave Diagram of Triple 8-bit Operands Multiplier Circuit

	Msgs				
+ /tri_multiplier_8b/a	0 0	178	0	210	255
+ /tri_multiplier_8b/b	0 0	46	0	0	255
+ /tri_multiplier_8b/product_aa	0 0	31684	0	44100	65025
+ /tri_multiplier_8b/product_laa_b	0 0	9016	0	0	255
+ /tri_multiplier_8b/product_haa_b	0 0	5658	0	0	64770
+ /tri_multiplier_8b/product_aab_m	0 0	1448448	0	0	16581120
+ /tri_multiplier_8b/product_aab_l	0 0	9016	0	0	255
+ /tri_multiplier_8b/p	0 0	1457464	0	0	16581375

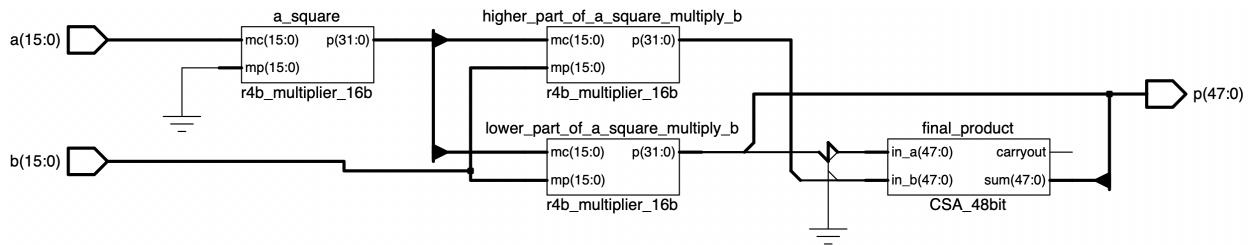
## 16-bit Triple Operands Multiplier Circuit Implementation

Like the 8-bit version, the 16-bit implementation of the booth multiplier will need 9 blocks of booth stage to calculate 9 partial products and a 17-bit negation complement generator. Because of the characteristic of the algorithm, the 16-bit version can not reuse or extend from the 8-bit version circuit designed before. Hence the circuit will need to build every block from scratch. **This is one of the drawbacks of the booth algorithm: lack of expandability.**

Figure 28 presents the RTL description of the triple 16-bit operands multiplier circuit and Figure 29 presents the simulation result. Figure 40 presents the RTL description of the 16-bit booth multiplier circuit.

**Figure 28**

*Synthesized RTL Diagram of Triple 16-bit Operands Multiplier Circuit*



**Figure 29**

*Simulation Wave Diagram of Triple 16-bit Operands Multiplier Circuit*

Msgs	Data-0	Data-1	Data-2	Data-3
+ /tri_multiplier_16b/a	38096	0	11507	65535
+ /tri_multiplier_16b/b	60624	0	0	65535
+ /tri_multiplier_16b/product_aa	1451457604	0	132411049	4294836225
+ /tri_multiplier_16b/product_aab_l	1928570688	0	0	65535
+ /tri_multiplier_16b/product_aaa_b	1342639728	0	0	4294770690
+ /tri_multiplier_16b/product_aab_m	8799123714208	0	0	281462091339840
+ /tri_multiplier_16b/product_aab_l	1928570688	0	0	65535
+ /tri_multiplier_16b/p	87993165784896	0	0	281462092005375

## Overflow Handling

**Heading II**

*Heading III*

**Heading IV.**

*Heading V.*

### End Flag Generator

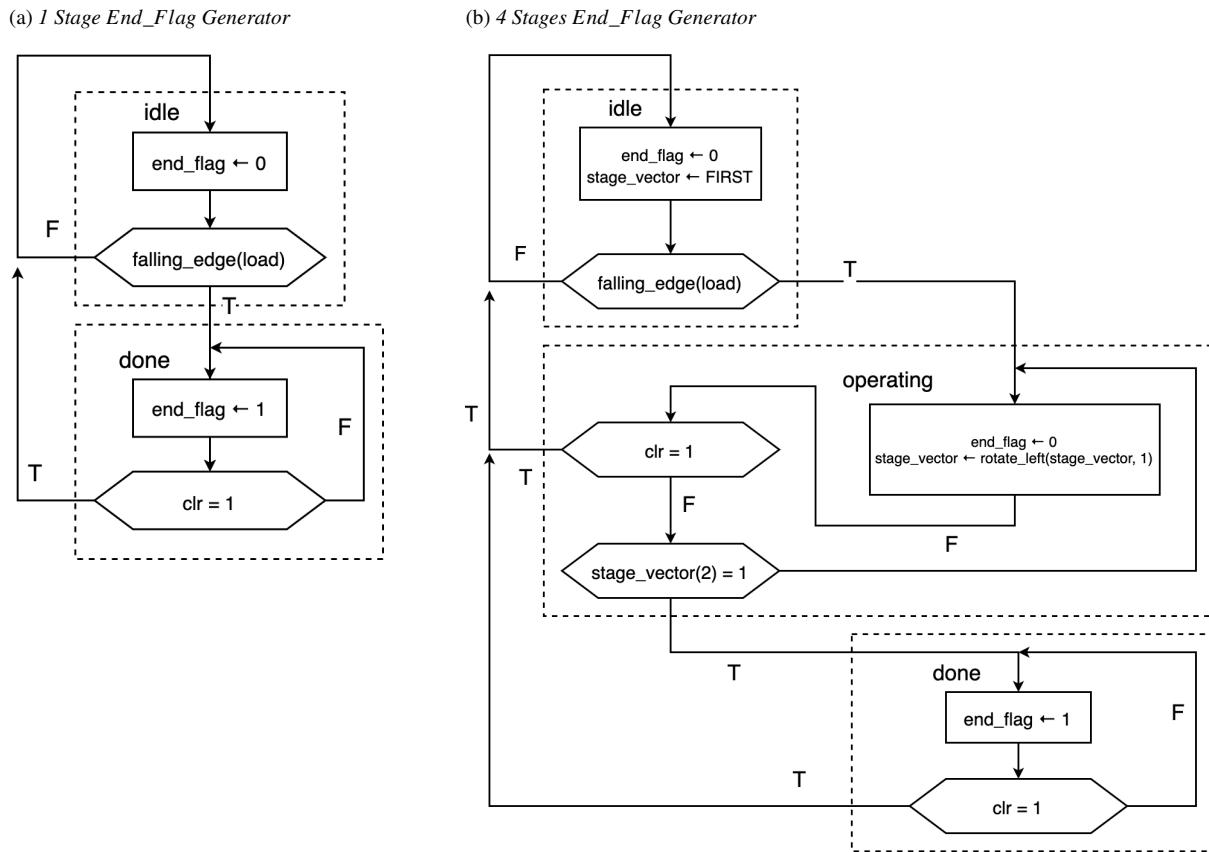
To generate a valid *END\_FLAG* signal for the ALU, a Timed Mealy State Machine is introduced into the circuit. The ASMD charts of the 1 stage and 4 stages *END\_FLAG* generator are presented in Figure 30.

For pipelined circuit, the *END\_FLAG* will result in the output after counting 4 synchronized clock cycles when the *LOAD* signal goes from high to low. The counting of the clock cycle is implemented by rotating a vector signal to avoid introducing an extra addition circuit.

For non-pipelined circuit, the *END\_FLAG* will result in the output of a clock later after the *LOAD* signal goes from high to low.

**Figure 30**

*ASMD Chart of the FSM of Different End Flag Generator*



*Note.* In (b), the “stege\_vector” is a 3 bit vector signal and the “FIRST” is “001”.

### Non-pipelined Implementation

For non-pipelined implementation, the arithmetic logic block functions with a clock and an active low signal *LOAD* to load the operands *A* and *B* into the operand registers. After finishing the arithmetic operations, the result will be stored in the output register with a high *END\_FLAG* signal. The *END\_FLAG* signal is used to denote whether the output is valid, it is asserted when the calculation has finished. The *CLEAR* signal will clear all the operand and output registers to ‘0’ when it is active high.

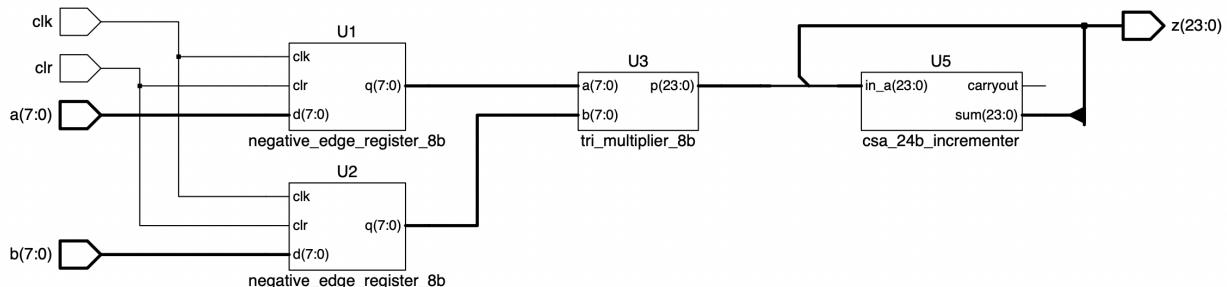
### Operating Circuit

Once the operands *A* and *B* are loaded into the operand registers. The output of these registers is then fed into the designed multiplier to perform the multiplication calculation required by the arithmetic unit. The output of the multiplication unit is then fed into the shifter unit. After being shifted by four bits to the right, the result is then fed into the 24-bit incrementor. The resulting result then passed into a result register pending final overflow processing. The result output of the overflow process is then fed into the *Z*-port.

The operating circuit for the non-pipelined implementation is shown in Figure 31.

**Figure 31**

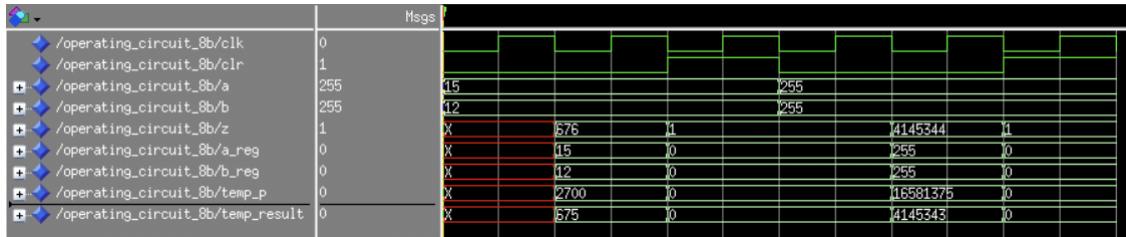
*Synthesized RTL Diagram of Non-pipelined Operating Circuit*



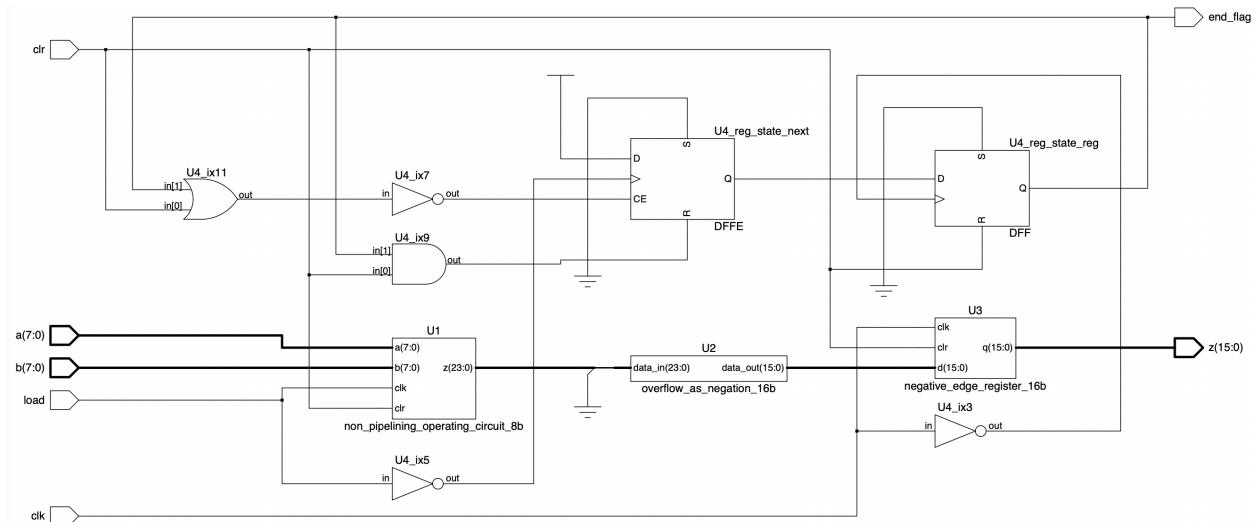
The test simulation results for the non-pipelined operating circuit are shown in Figure 32.

### Output Process

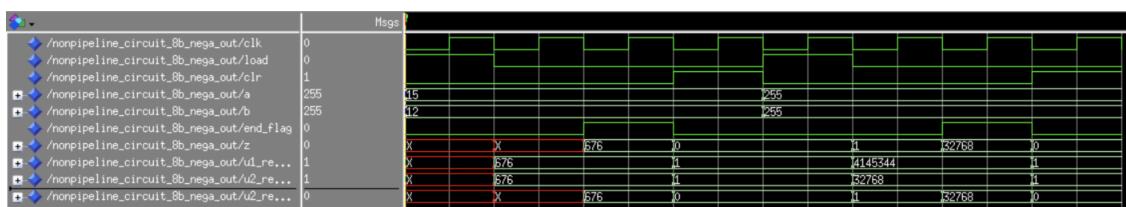
In the non-pipelined implementation, negation output is used to meet the project requirements of 16-bit *Z*-port outputs, which just represent the result as overflow. The implementation of the whole circuit is shown in Figure 33. The test simulation results for the negation output are shown in Figure 34

**Figure 32***Simulation Wave Diagram of Non-pipelined Operating Circuit*

*Note.* Signal with suffix “\_after\_reg” means the value of the signal is an output of a register

**Figure 33***Synthesized RTL Diagram of Non-pipelined Circuit*

*Note.* A 1 stage END\_FLAG generator is used.

**Figure 34***Simulation Wave Diagram of Non-pipelined Circuit*

*Note.* Signal with suffix “\_after\_reg” means the value of the signal is an output of a register.

## Pipelined Implementation

For pipelined implementation, a set of stage registers is placed after every arithmetic operation block for memorizing the outcome of every stage. The required operation  $Z = \frac{1}{4}[A^2 * B] + 1$  can be divided into three stages of calculation with one overflow handling stage. Stage 1 should obtain the product of  $A$  and  $A$  and  $B$ . Then stage 2 should perform two bits right shifting operation on the result of stage 1, this operation represents the  $\frac{1}{4}$ . After that, stage 3 will add number 1 to the result of stage 2. Finally, stage 4 should handle the bit width of the arithmetic result into the required width.

### 8-bit Operands Implementation with Negation Overflow Handling

By this design, the circuit will obtain the result in 4 synchronized clock cycles after the falling edge event of the *load* signal. Table 4 shows how the registers in this pipelined circuit will latch the values. Figure 36 presents the RTL description of the circuit and its arithmetic block.

**Table 4**

*A Visual Representation Table of the Pipelined Process with Negation Overflow Handling*

clk	A Reg	B Reg	Stage 1 Reg	Satge 2 Reg	Satge 3 Reg	Stage 4 Reg(Z)
0	12	3	-	-	-	-
1	100	100	432	-	-	-
2	-	-	1000000	108	-	-
3	-	-	-	250000	109	-
4	-	-	-	-	250001	109
5	-	-	-	-	-	-32768

Figure 35 shows the simulation wave result of the pipelined ALU circuit. As can be observed with the figure, once the *load* signal goes from 1 to 0, after 4 clock cycles, *z* obtains the result of the input. And the circuit is pipelining the data by the falling edge event of *load* signal and the *clock*. If the *clk* signal was asserted, all registers will be reset and *end\_flag* turned into 0.

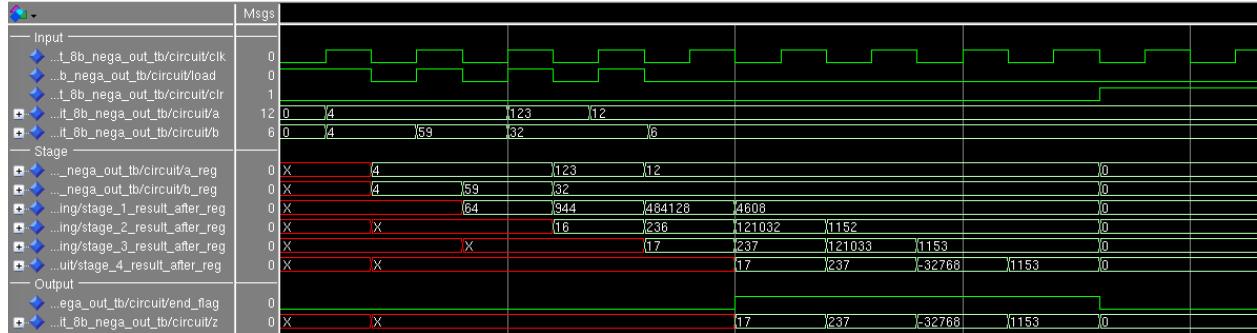
### 8-bit Operands Implementation with Output Separation Overflow Handling

With separation overflow handling discussed before, the full result can be obtained after three stages of calculation with two overflow handling stages. Before the full data reach the *Z* port, the 24-bit result will be separated into 2 parts: the higher part stored in one register and the lower part stored in another register. The handler takes those two data and transfers them to *Z* port alternately by the clock.

Table 5 presents how the *z* will take over the data and Figure 37 shows the simulation of the design. Figure 38 presents the RTL description of the design.

**Figure 35**

*Simulation Wave Diagram of the 8-bit Pipelined ALU Circuit with Negation Output*

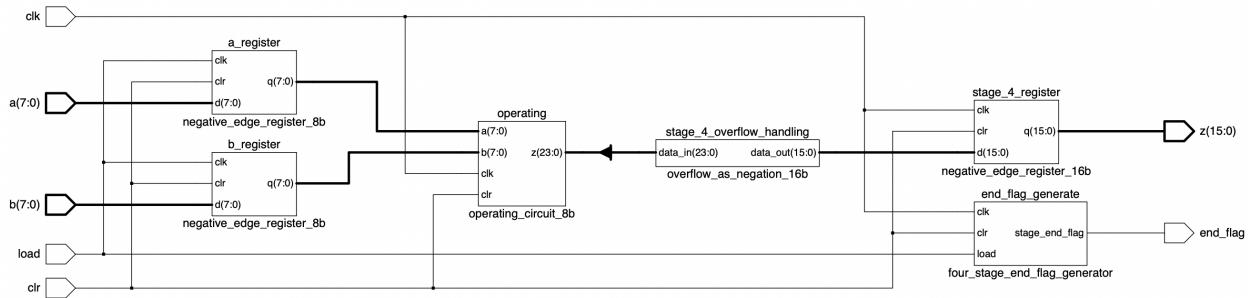


Note. Signal with suffix “\_after\_reg” means the value of the signal is an output of a register.

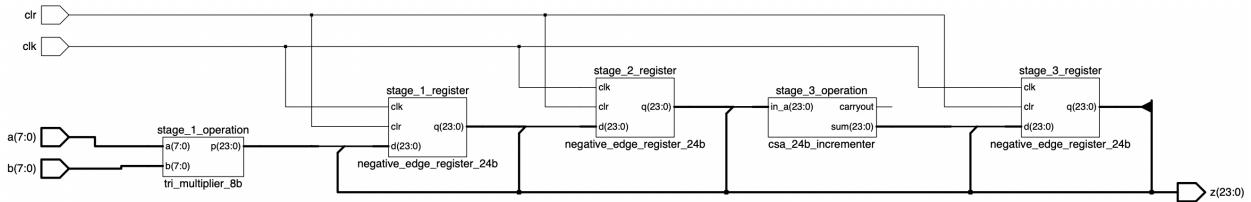
**Figure 36**

*Synthesized RTL Diagram of the 8-bit Operands Pipelined ALU Circuit and the Arithmetic Operation Block*

(a) Top Design of the Circuit



(b) Arithmetic Operation Block



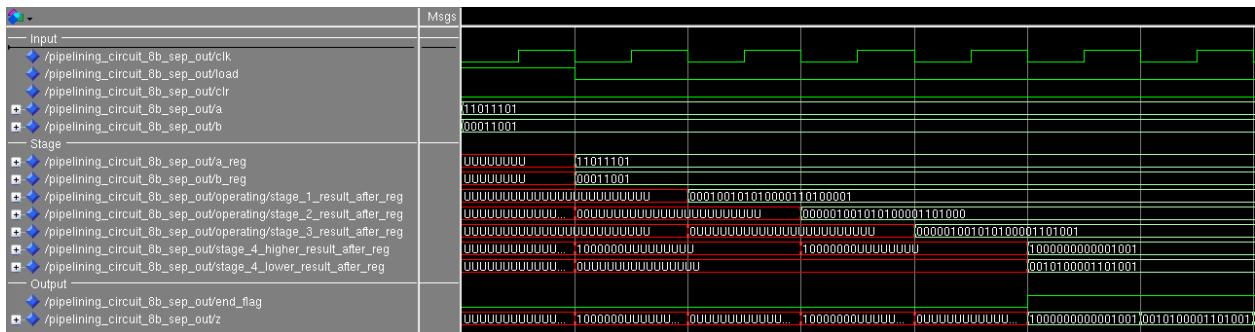
**Table 5**

A Visual Representation Table of the Pipelined Processing with Output Separation Overflow Handling

clk	Stage 3 Reg	Stage 4 Reg for		Z
		Higher	Lower	
...				
3	000110110000110100101101	-	-	-
4	-	10000000000110110	0000110100101101	10000000000110110
6	-	10000000000110110	0000110100101101	0000110100101101

**Figure 37**

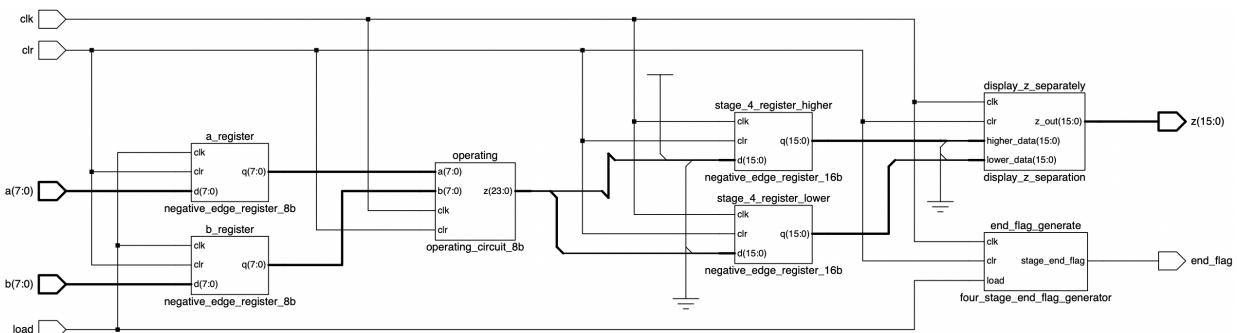
Simulation Wave Diagram of the 8-bit Pipelined ALU Circuit with Separation Output



Note. Signal with suffix “\_after\_reg” means the value of the signal is an output of a register. The value of Z will retrieve from the previous two registers alternately.

**Figure 38**

Synthesized Diagram of the 8-bit Pipelined ALU Circuit with Separation Output

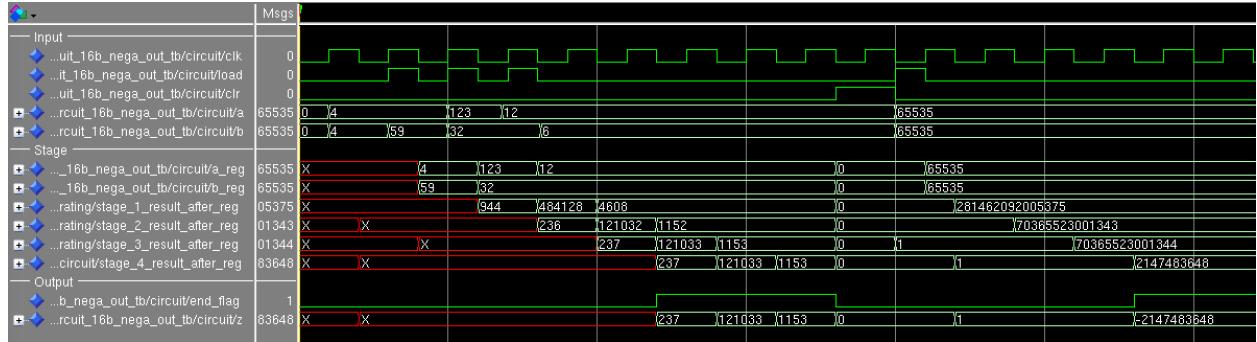


## 16-bit Operands Implementation and Simulation

By now there will no necessary for discussing the implementation of the extended design since all details were presented in the previous sections. Figure 39 presents the testbench simulation of the bit-width extended circuit.

**Figure 39**

*Simulation Wave Diagram of the 16-bit Pipelined ALU Circuit with Negation Output*



*Note.* Signal with suffix “\_after\_reg” means the value of the signal is an output of a register.

## Synthesis and Analysis of the Arithmetic Circuit

**Heading II**

*Heading III*

**Heading IV.**

*Heading V.*

### References

Cervantes Saavedra, M. d., Raffel, B., & Wilson, D. d. A. (1999). *Don Quijote: a new translation, backgrounds and contexts, criticism* (1st ed ed.). New York: W.W. Norton.

**Figure 40**

Synthesized RTL Diagram of 16-bit Radix-4 Booth Multiplier

