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 Conetnts

List of Figures

List of Tables

Introduction

This is the project report of the course **COEN 6501: Digital Design and Synthesis**. The project requires an implementation of an ALU circuit design that perform the equation of $Z = \frac{1}{4}[A^2 * B] + 1$. More details of the requirements will be discussed in the next section. Breaking down the project requirements, the team should implement the following component designs to complete the ALU:

Table 1
Components of the ALU Circuit

Functional Components	Other Components
n-bit adder design	2-bit right shifter
n-bit multiplier with 2 and 3 operands	n-bit incrementor
overflow handler	negative edge registers
END_FLAG generator	

As per adder design, the team chose the **"Carry Select Adder"** implementation. This adder design is suitable for bit width extension.

As per multiplier design, the team chose the **"Radix-4 Booth Algorithm"** implementation which is an elegant design for reducing both area occupation and delay.

As per the overflow handler, the team provides two solutions: 1. giving a negation which is impossible for the demanded equation to indicate the overflow; 2. dividing the full result into the higher part and the lower part and outputting them to the *Z* port alternately within two clock cycles.

As per the *END_FLAG* generator, the team chose to implement a FSM for indicating the valid *END_FLAG* signal. As per the register, falling edge clock event sensitive register will be used in the project, the registers for input *A* and *B* should use *LOAD* signal as the clock event which means those two registers are asynchronous, the rest of the registers should use *Clock* signal as the clock event.

After completing all component designs, the ALU will be implemented in both pipelined and non-pipelined designs.

The project is designed, implemented, simulated, and synthesized with the environments of:

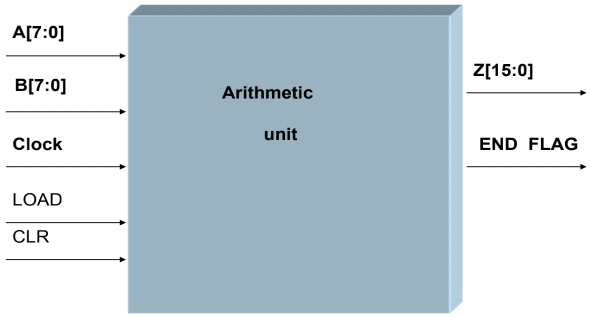
- ***ModelSim*** *SE-64 6.6g May 23 2012 Linux 3.10.0-1160.45.1.el7.x86_64*
- ***Precision RTL Synthesis*** *64-bit 2016.1.0.15 (Production Release)*
- ***XILINX ISE and Project Navigator*** *10.1 (lin64)*
- ***GHDL*** *2.0.0-dev (1.0.0.r849.gc56db233)*

The synthesized report is based on the "2VP20ff896" device provided by *Precision*.

Project Requirement

The project should implement ALU circuit described in Figure 1

Figure 1
The RTL Diagram of the Required ALU



The project contains the following requirements for signals:

- The operands *A* and *B* are latched into register RA and RB when *LOAD* signal transit from high to low.
- The unit outputs the results in 16-bit register RZ output port.
- Each calculation starts with a *LOAD* signal and ends with an *END_FLAG* signal.
- The *CLEAR* signal will clear all registers to '0'.
- The unit performs the arithmetic operation until *END_FLAG* becomes high.
- The 16-bit product shall be loaded into the 16-bit Z port.
- The design shall be structural.

The project contains the following extra features:

- **(Accomplished)** Expansion of the method for 16 bits operand.
- **(Accomplished)** Pipelining of the design.
- Multiply Accumulate for additional operands.

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 2.

Table 2
Full Adder Truth Table

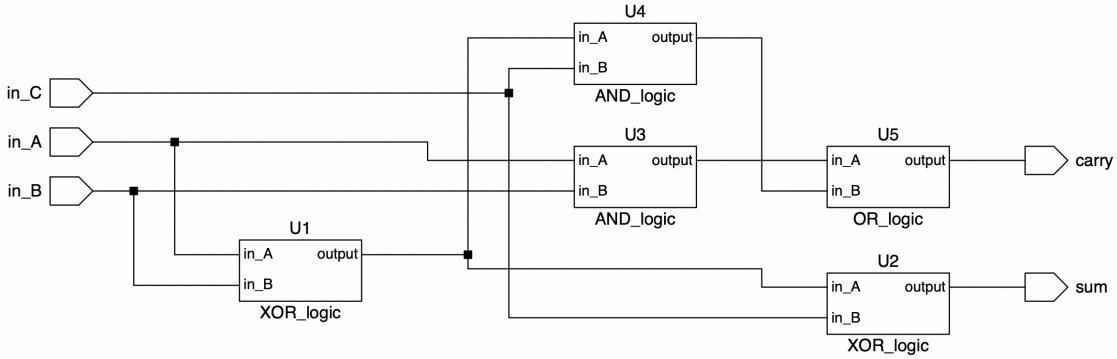
A	**B**	$Carry_{in}$	$Carry_{out}$	**Sum**
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

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

$$Sum = A \oplus B \oplus C_{in}$$
$$C_{out} = (A \bullet B) + (C_{in} \bullet (A \oplus B)) \quad (1)$$

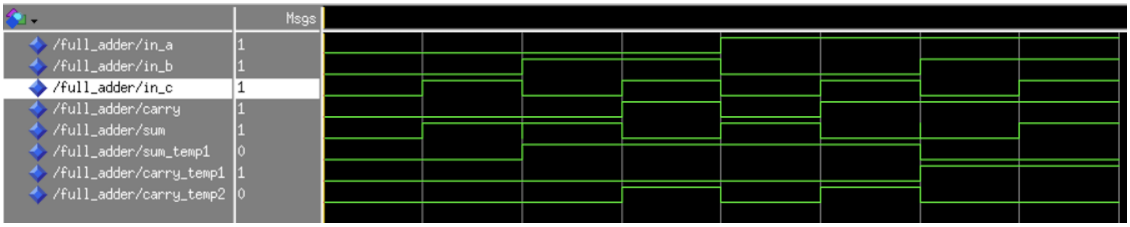
The circuit for the full adder is shown in Figure 2.

Figure 2
Synthesized RTL Diagram of Full Adder Block



The simulation results for the full adder are shown in Figure 3.

Figure 3
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 3.

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

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

Table 3
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 4.

The simulation results for the multiplexer are shown in Figure 5.

Figure 4
Synthesized RTL Diagram of 2-to-1 Multiplexer Block

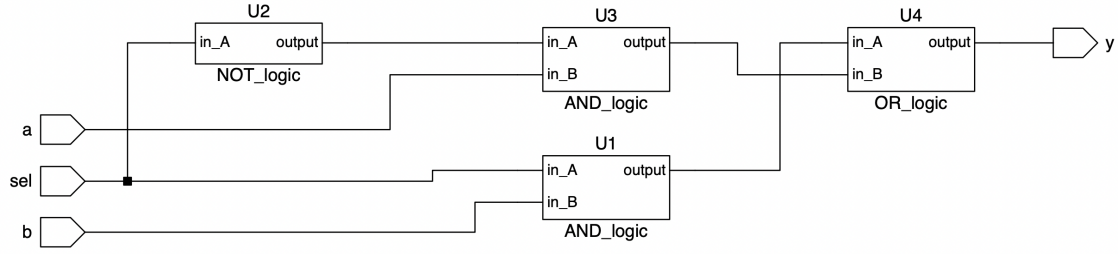
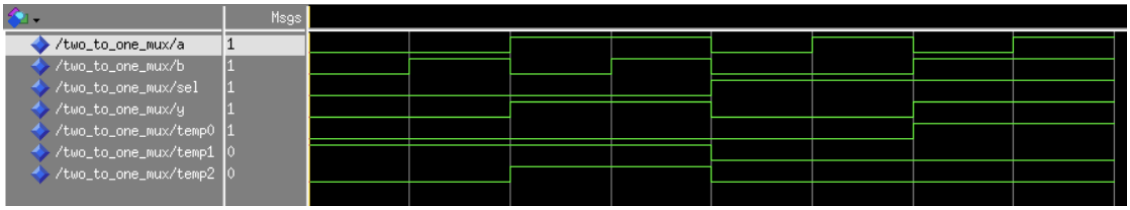


Figure 5
Simulation Wave Diagram of 2-to-1 Multiplexer Block

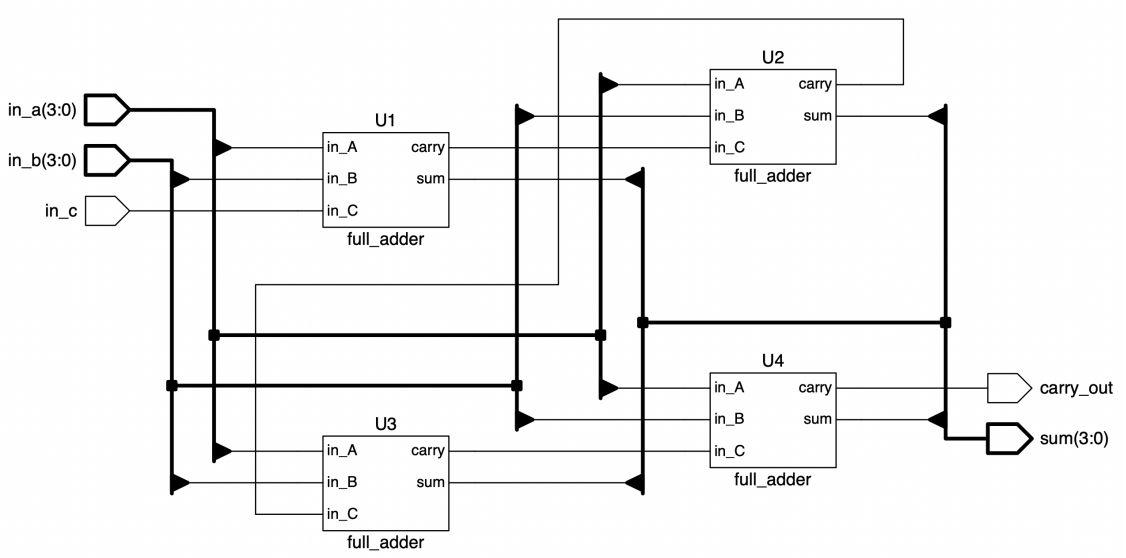


4-bit Ripple Carry Adder

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 6.

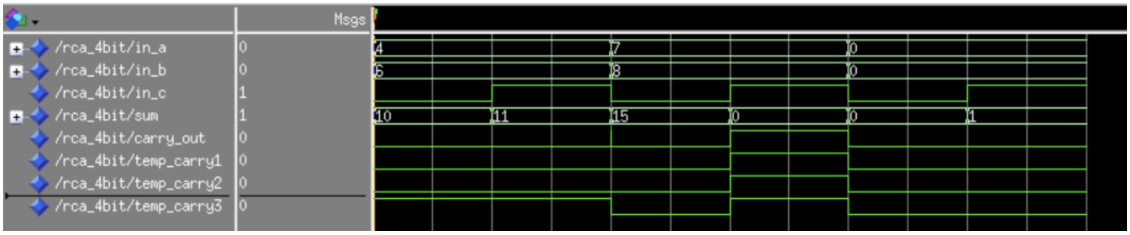
The selected simulation cases are shown below and the simulation results for the 4-bit ripple carry adder are shown in Figure 7.

Figure 6
Synthesized RTL Diagram of 4-bit Ripple Carry Adder Block



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

Figure 7
Simulation Wave Diagram of 4-bit Ripple Carry Adder Block



4-bit Carry Select Adder

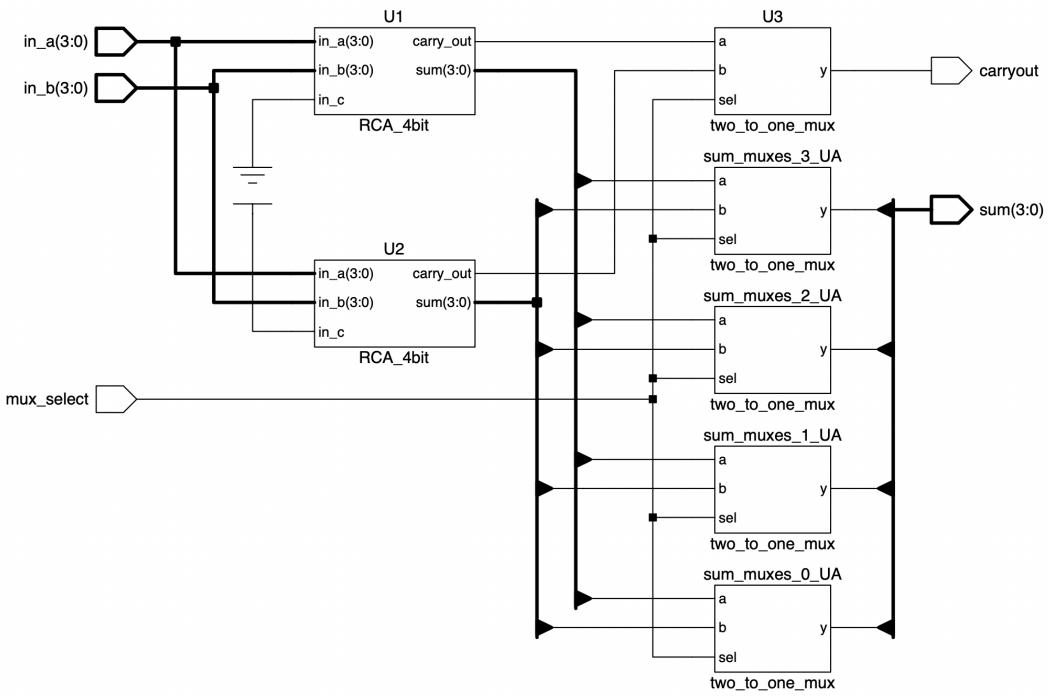
A basic building block size of carry select adder is four. A 4-bit carry select adder consists of two parallel 4-bit ripple carry adders and 2-to-1 multiplexers to perform the calculation twice. One of the 4-bit RCA block assumes that the input carry is 0 (RCA_0), the other assumes that the input carry 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:

$$T_{CSA} = T_{mux} + 4 \times T_{full\_adder} \tag{3}$$

The circuit for the 4-bit carry select adder is shown in Figure 8.

Figure 8

Synthesized RTL Diagram of 4-bit Carry Select Adder Block

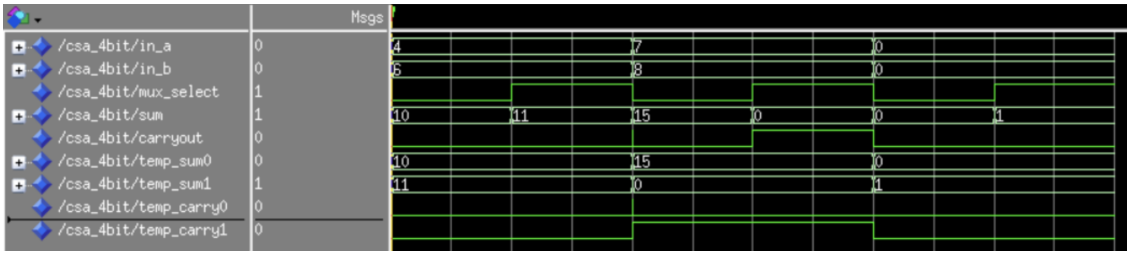


If the input carry C_{in} from the lower level is 0, the output carry of the RCA_0 is selected as the output carry of this 4-bit CSA block. If the input carry C_{in} from the lower level is 1, the output carry of the RCA_1 is selected as the output carry. At the same time C_{in} is used as the selection 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 9.

Figure 9

Simulation Wave Diagram of 4-bit Carry Select Adder Block



16-bit Carry Select Adder

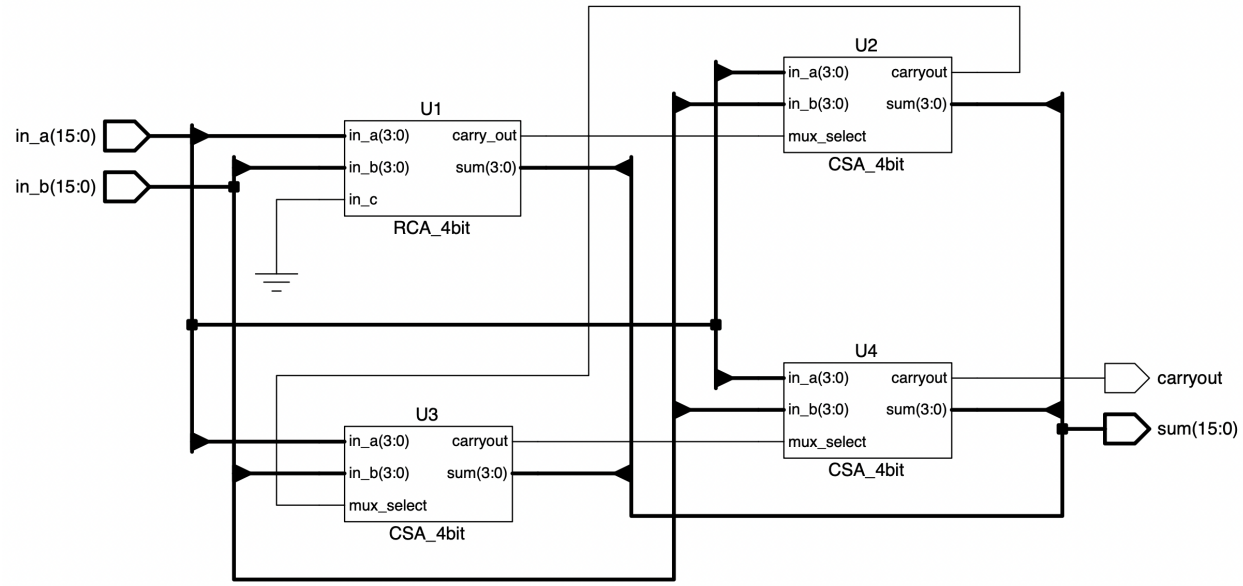
In this project, 16-bit carry select adder is used. A 16-bit carry select adder can be created using three 4-bit CSA blocks and one 4-bit RCA blocks. The first block is a 4-bit RCA, the inputs are two binary numbers from multiplier, so that there is no input carry and the C_{in} can be set to 0. Then 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:

$$T_{CSA} = 3 \times T_{mux} + 4 \times T_{full\_adder} \tag{4}$$

The circuit for the 16-bit carry select adder is shown in Figure 10.

Figure 10

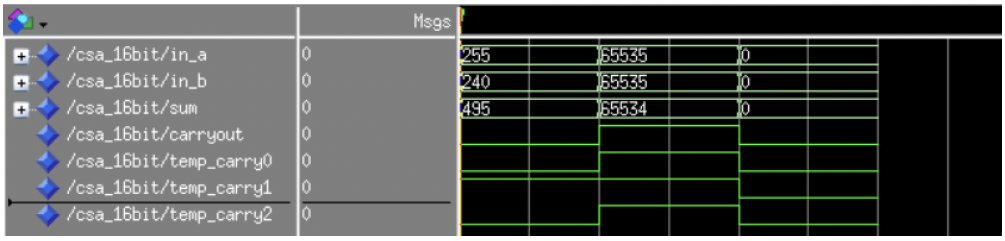
Synthesized RTL Diagram of 16-bit Carry Select Adder Block



The simulation results for the 16-bit carry select adder are shown in Figure 11.

Figure 11

Simulation Wave Diagram of 16-bit Carry Select Adder Block



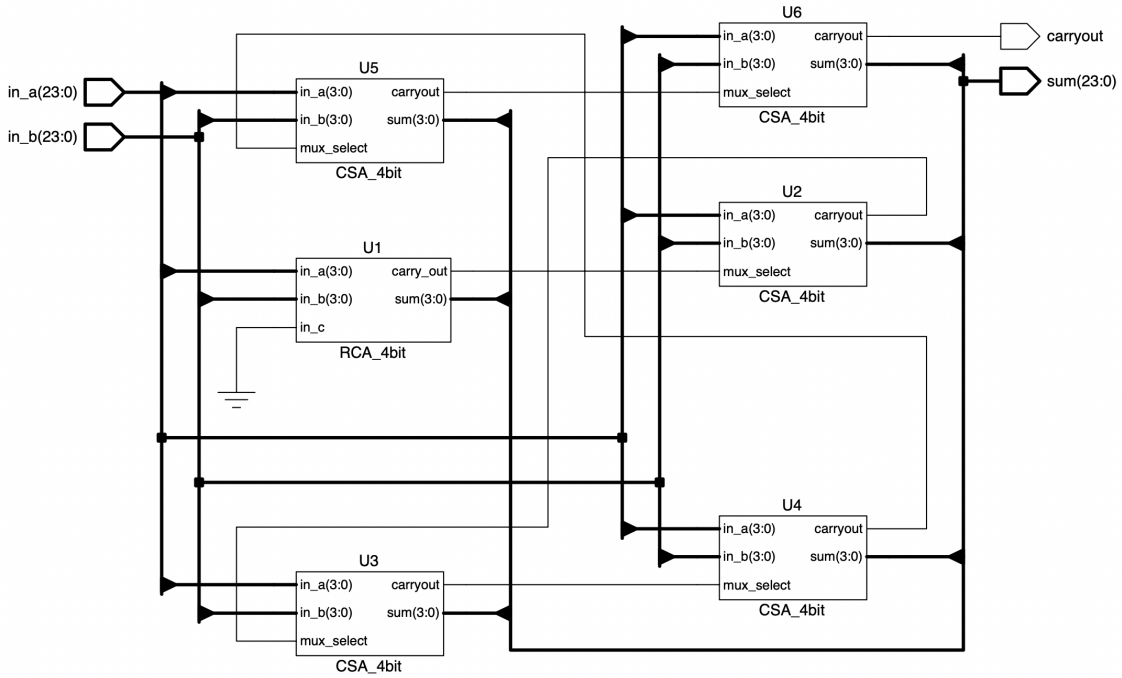
24-bit Carry Select Adder

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:

$$T_{CSA} = 5 \times T_{mux} + 4 \times T_{full\_adder} \tag{5}$$

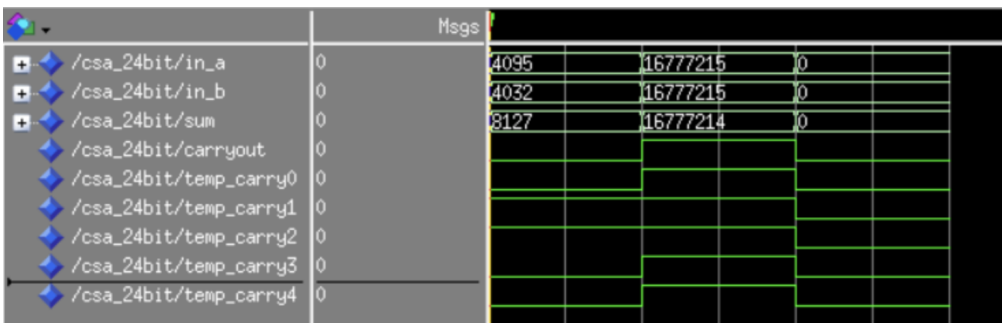
The circuit for the 24-bit carry select adder is shown in Figure 10.

Figure 12
Synthesized RTL Diagram of 24-bit Carry Select Adder Block



The simulation results for the 24-bit carry select adder are shown in Figure 13.

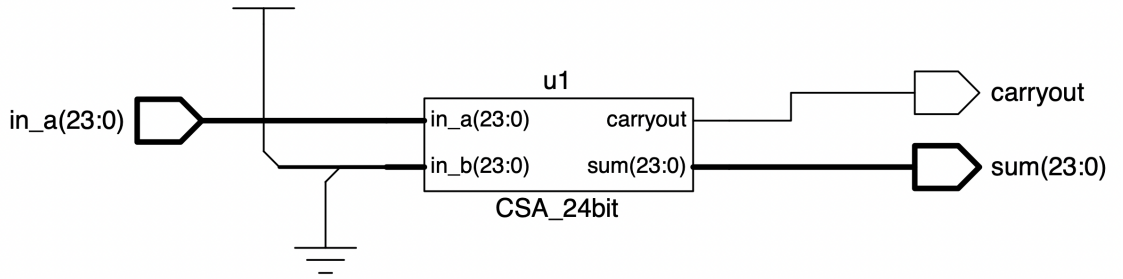
Figure 13
Simulation Wave Diagram of 24-bit Carry Select Adder Block



24-bit CSA Incrementor

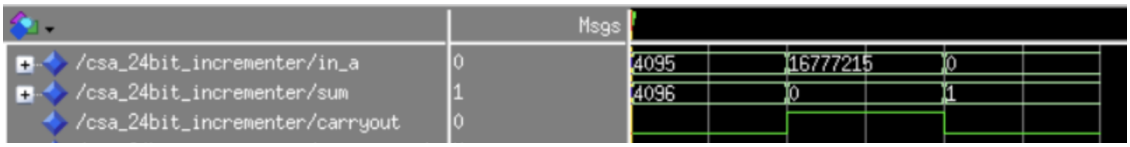
The incrementor used in this project were designed on the basis of the CSA. The principle is to define the input B of the CSA as a constant with the lowest bit of 1 and the rest of 0. Due to different bit requirements, a N-bit incrementor can be implemented by N-bit CSA. A 24-bit incrementor is used in this project. The circuit for the 24-bit incrementor is shown in Figure 14.

Figure 14
Synthesized RTL Diagram of 24-bit Incrementor Block



The simulation results for the 24-bit incrementor are shown in Figure 15.

Figure 15
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 sighed 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".

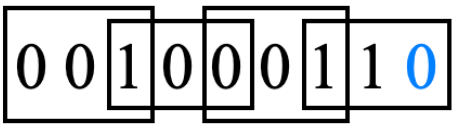
Example in Signed Number Multiplication

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 16 is shown. Then all partial products can be derived from the code blocks and the "Code Table" which is presented in Table 4.

Table 4
Code Table of Radix-4 Booth Algorithm

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

Figure 16
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 17.

Figure 17
Process of the Algorithm for Signed Number

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

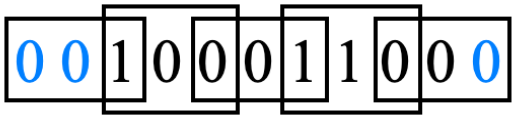
Note. Bits in green color represent the product is extended to 16 bits. Bits in red color represent the shifted 2 bits leftward.

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 19 and Figure 19 are shown.

Figure 18
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 19
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 1 0 1 0 1 1 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
	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 20 presents the synthesized RTL diagram of the multiplier circuit.

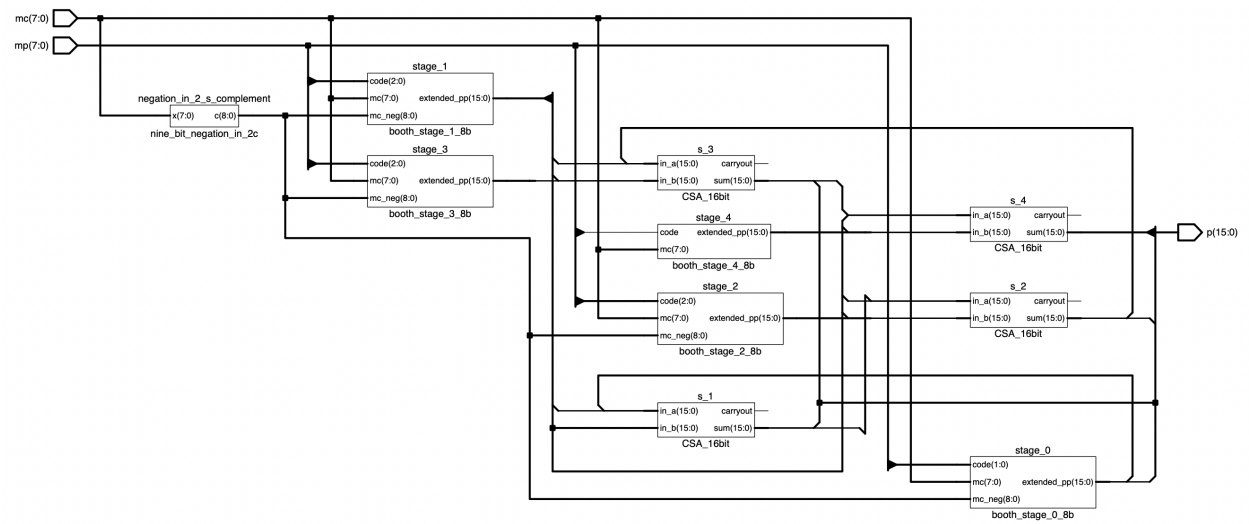
The circuit will first calculate a 9-bit negation of operand multiplicand in 2's complement. Then pass through all five booth stage blocks to get five 16-bit partial products. And finally sums those partial products up.

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 21. The logic of the generator that uses a 2-to-1 mux is straightforward as Expression (6) described. The simulation result is shown in Figure 22.

Figure 20
Synthesized RTL Diagram of 8-bit Radix-4 Booth Multiplier



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

Figure 21
Synthesized RTL Diagram of Complement Generator

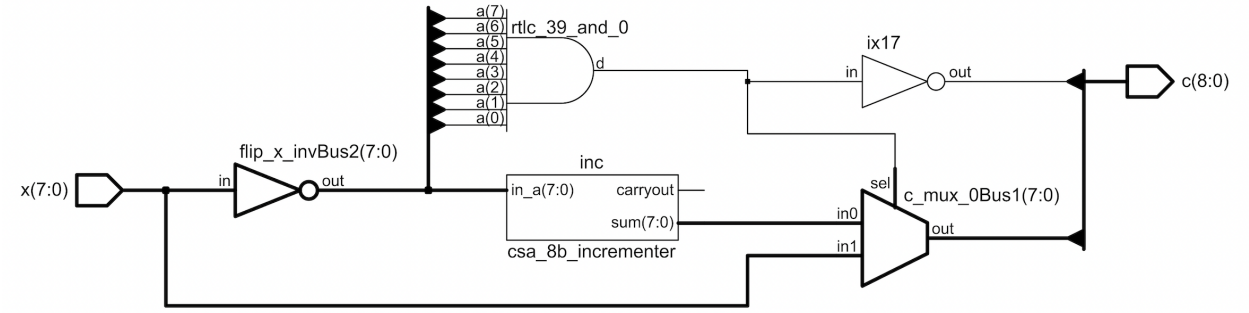
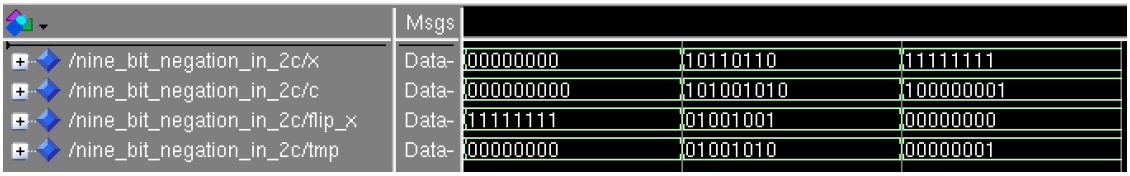
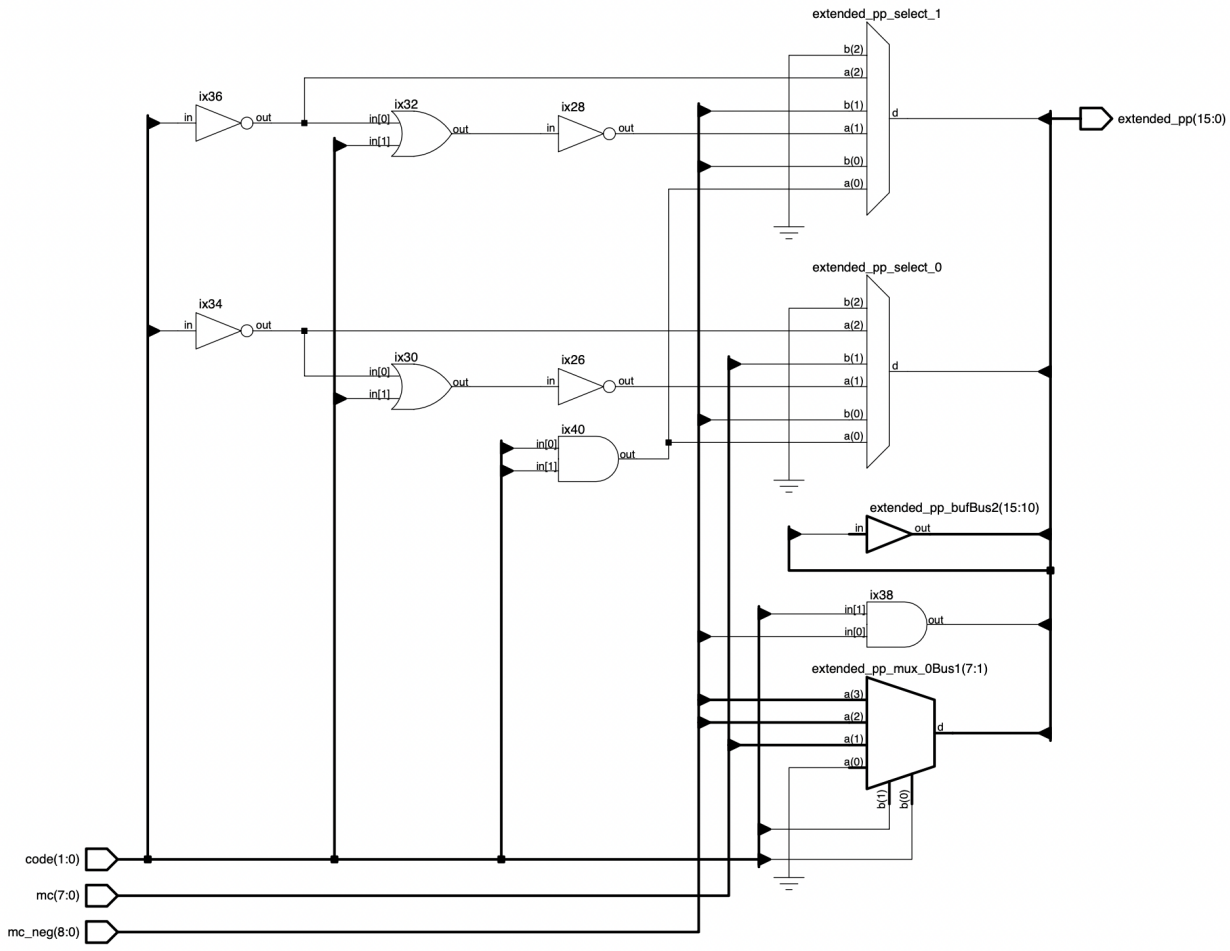


Figure 22
Simulation Wave Diagram of Complement Generator



Booth Stage 0. Figure 23 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 shown in Figure 24.

Figure 23
Synthesized RTL Diagram of Booth Stage 0 Block

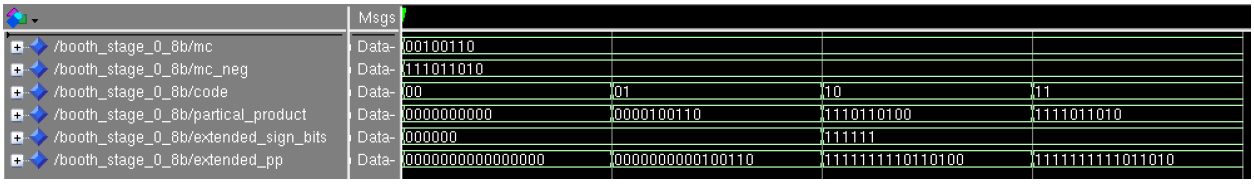


$$partial\_product = \begin{cases} 0000000000, & \text{if } code = 00 \\ concat(00, mc), & \text{if } code = 01 \\ concat(mc\_neg, 0), & \text{if } code = 10 \\ concat(mc\_neg(8), mc\_neg), & \text{else } code = others \end{cases} \tag{7}$$

$$extended\_sign\_bits = (others => partical\_product(9))$$

$$extended\_pp = concat(extended\_sign\_bits, partial\_product)$$

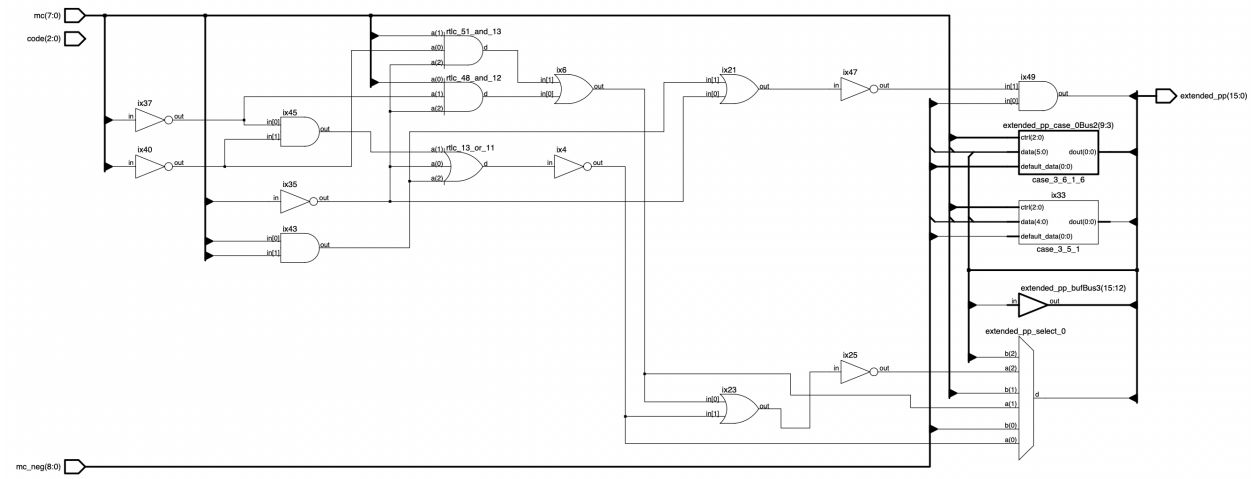
Figure 24
Simulation Wave Diagram of Booth Stage 0 Block



Note. Four code values were provided to simulate the *extended_pp*.

Booth Stage 1. Figure 25 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 26.

Figure 25
Synthesized RTL Diagram of Booth Stage 1 Block



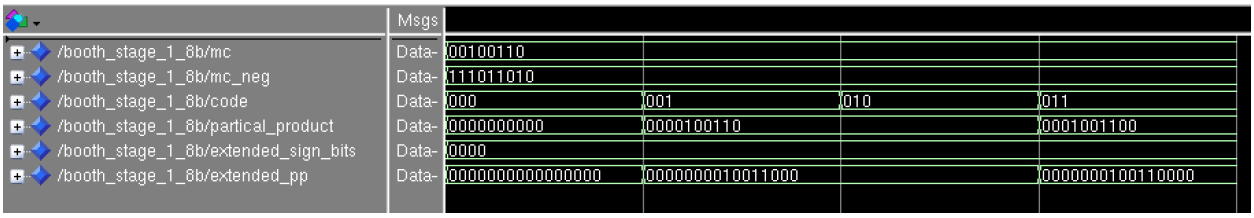
$$partial\_product = \begin{cases} 0000000000, & \text{if } code = 000|111 \\ concat(00, mc), & \text{if } code = 001|010 \\ concat(0, mc, 0), & \text{if } code = 011 \\ concat(mc\_neg, 0), & \text{if } code = 100 \\ concat(mc\_neg(8), mc\_neg), & \text{else } code = others \end{cases} \tag{8}$$

$$extended\_sign\_bits = (others => partical\_product(9))$$

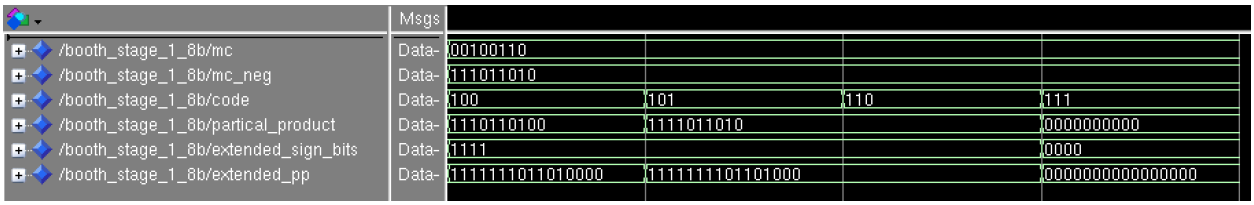
$$extended\_pp = concat(extended\_sign\_bits, partial\_product, 00)$$

Figure 26
Simulation Wave Diagram of Booth Stage 1 Block

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



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



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 27 presents the RTL description of the circuit and Figure 28 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 27
Synthesized RTL Diagram of Triple 8-bit Operands Multiplier Circuit

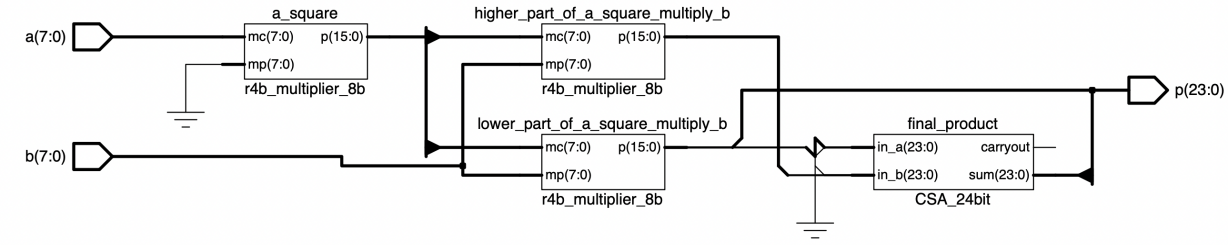
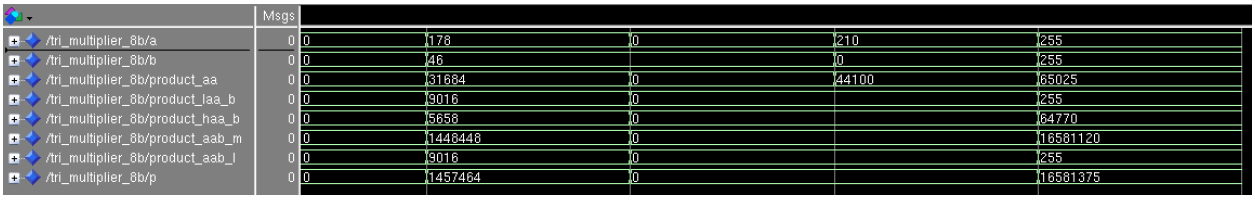


Figure 28
Simulation Wave Diagram of Triple 8-bit Operands Multiplier Circuit



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 29 presents the RTL description of the triple 16-bit operands multiplier circuit and Figure 30 presents the simulation result. Figure 47 presents the RTL description of the 16-bit booth multiplier circuit.

Figure 29
Synthesized RTL Diagram of Triple 16-bit Operands Multiplier Circuit

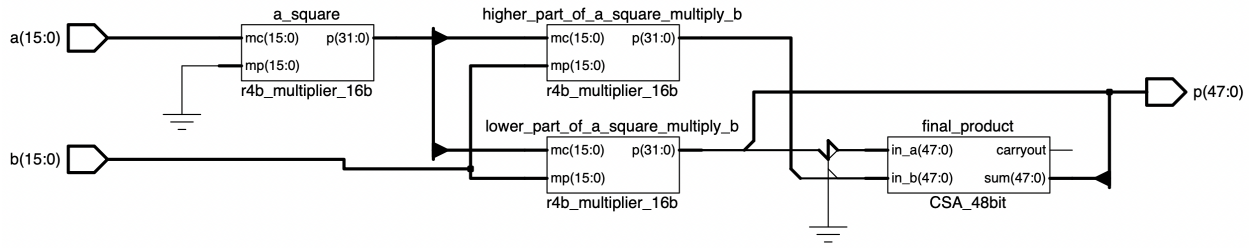
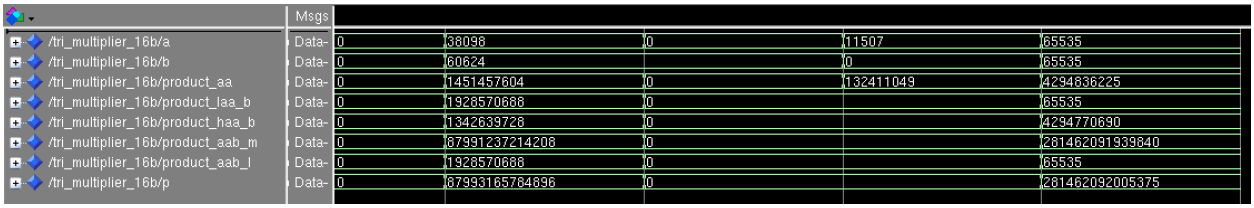


Figure 30
Simulation Wave Diagram of Triple 16-bit Operands Multiplier Circuit



Overflow Handling

In normal process, output *Z* is a 24-bit unsigned number from the equation since *A* and *B* are 8-bit numbers. Here are two methods to convert 24-bit unsigned number into 16-bit signed number.

Method 1: Negation Output

Overflow Detection

To detect whether a 24-bit unsigned number is overflow for a 16-bit signed slot, just simply using a 9-bit or gate for the higher 9-bit of the 24-bit unsigned number.

The RTL description is shown in Figure 31 and the simulation result of block is shown in Figure 32.

Figure 31
Synthesized RTL Diagram of the Zero Detector

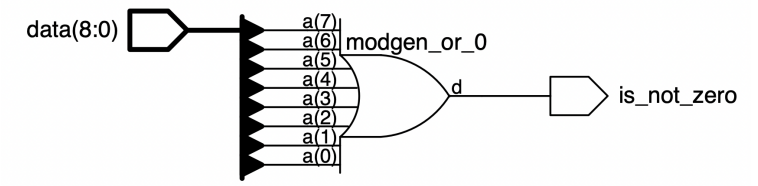


Figure 32
Simulation Wave Diagram of the Zero Detector

Handler Implementation

Once the 24-bit result is detected as an overflowing number, the circuit will just output a negative number to indicate the overflow happens. The RTL description of the implementation is shown in Figure 33 and the simulation result of block is shown in Figure 34.

Method 2: Display Z Separately with Two Clock Cycles

In this method, the circuit will output the 24-bit number into two parts. The higher part of the 24-bit result which is a 9-bit number will be stored in a register with concatenating a “1000000” to its front; The rest of the 15-bit will be stored in another register with concatenating a “0” to its front. Then the circuit will output those two parts to *Z* alternately by the clock cycles.

Figure 33
Synthesized RTL Diagram of the Overflow Handler of Method 1

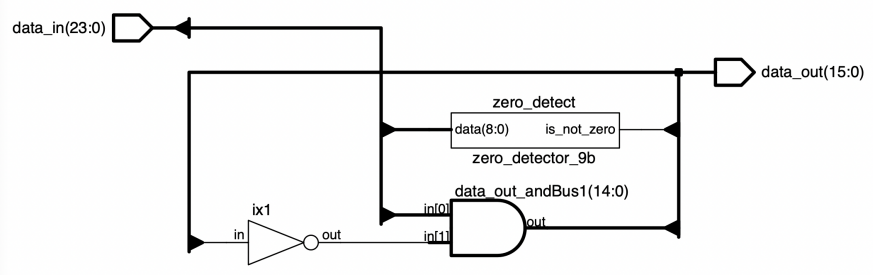
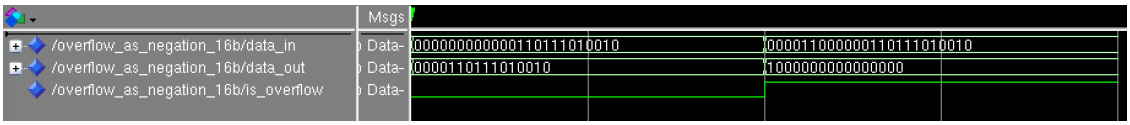


Figure 34
Simulation Wave Diagram of the Overflow Handler of Method 1



The drawback of this method is obvious: **the result will need two clock cycles to be fully obtained**. Hence the input of the ALU circuit should have a clock of the time interval between every set of *A* and *B* input to avoid output overlaps. This method acts like a 2 states FSM which RTL description is shown in Figure 35. The simulation result of block is shown in Figure 36.

Figure 35
Synthesized RTL Diagram of the Overflow Handler of Method 2

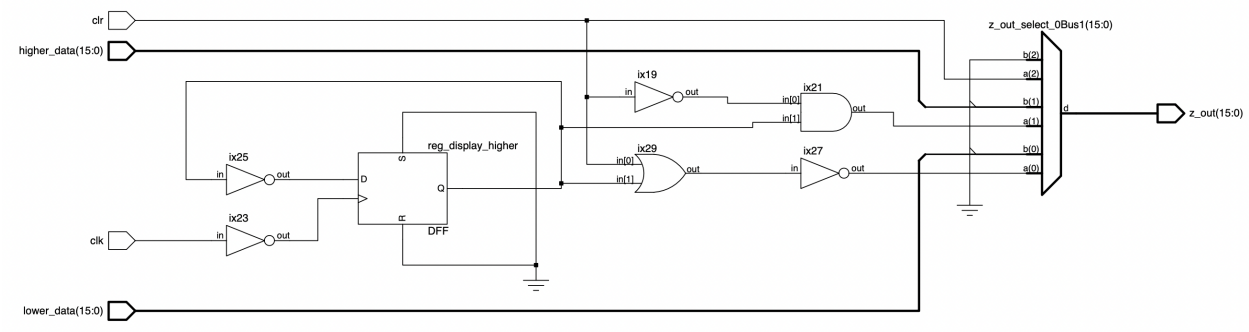
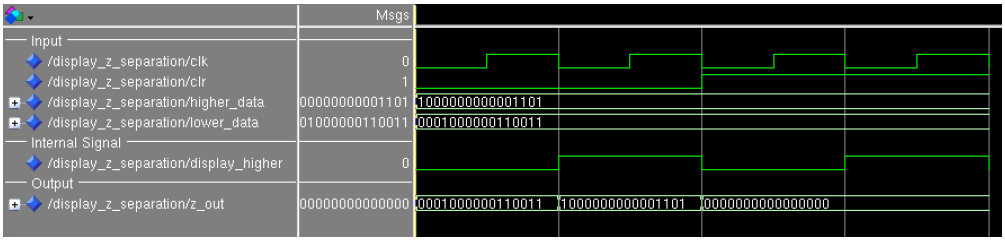


Figure 36
Simulation Wave Diagram of the Overflow Handler of Method 2



End Flag Generator

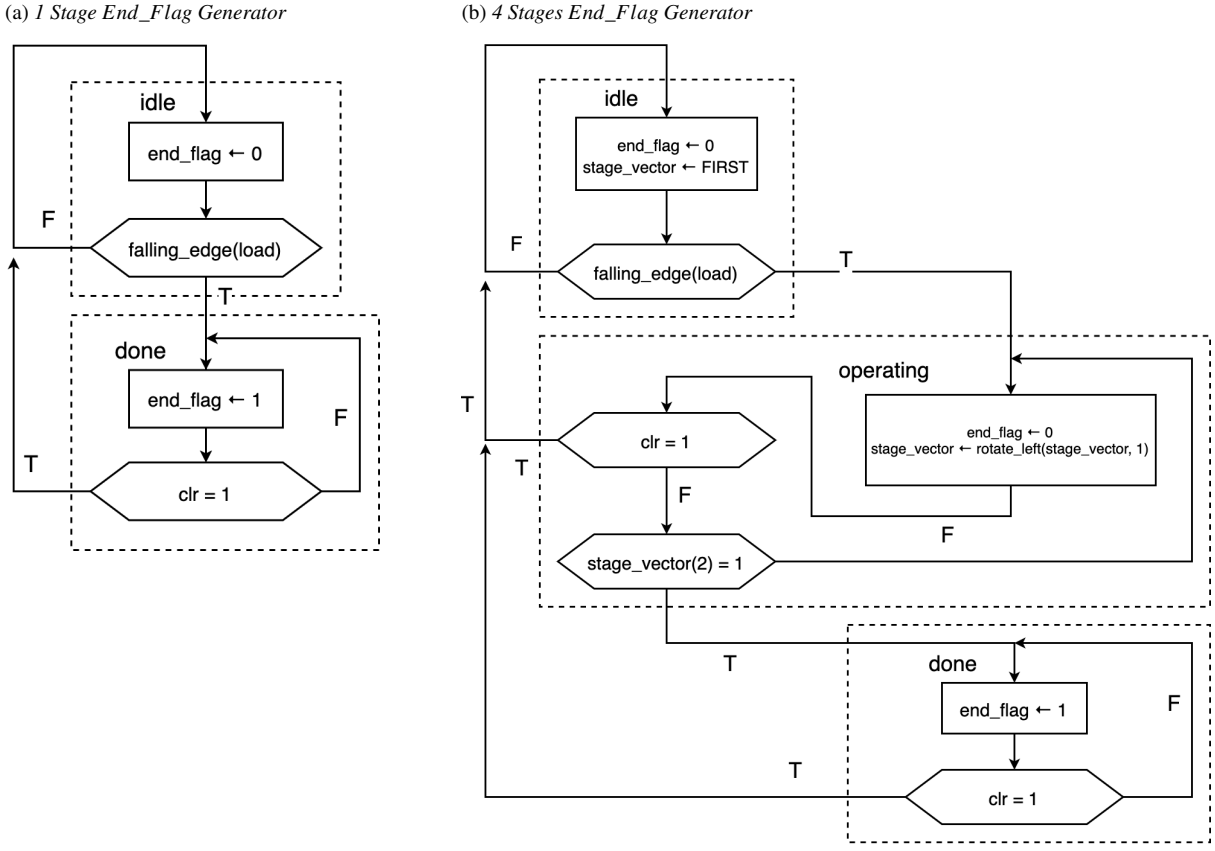
To generate a valid *END_FLAG* signal for the ALU, a Timed Mealy State Machine is introduced into the circuit. The ASMD charts of the 1 stage and 4 stages *END_FLAG* generator are presented in Figure 37.

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 37

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.

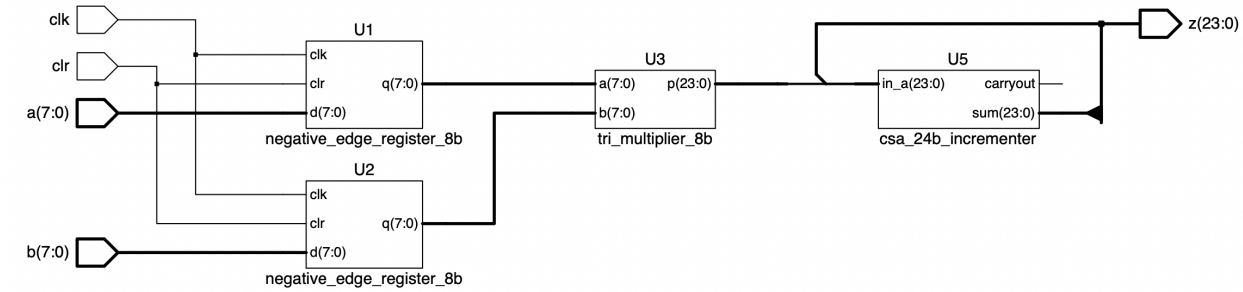
Operating Circuit

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 38.

Figure 38

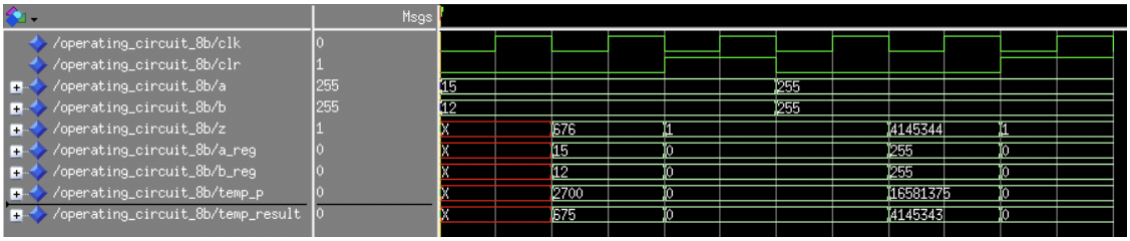
Synthesized RTL Diagram of Non-pipelined Operating Circuit



The test simulation results for the non-pipelined operating circuit are shown in Figure 39.

Figure 39

Simulation Wave Diagram of Non-pipelined Operating Circuit

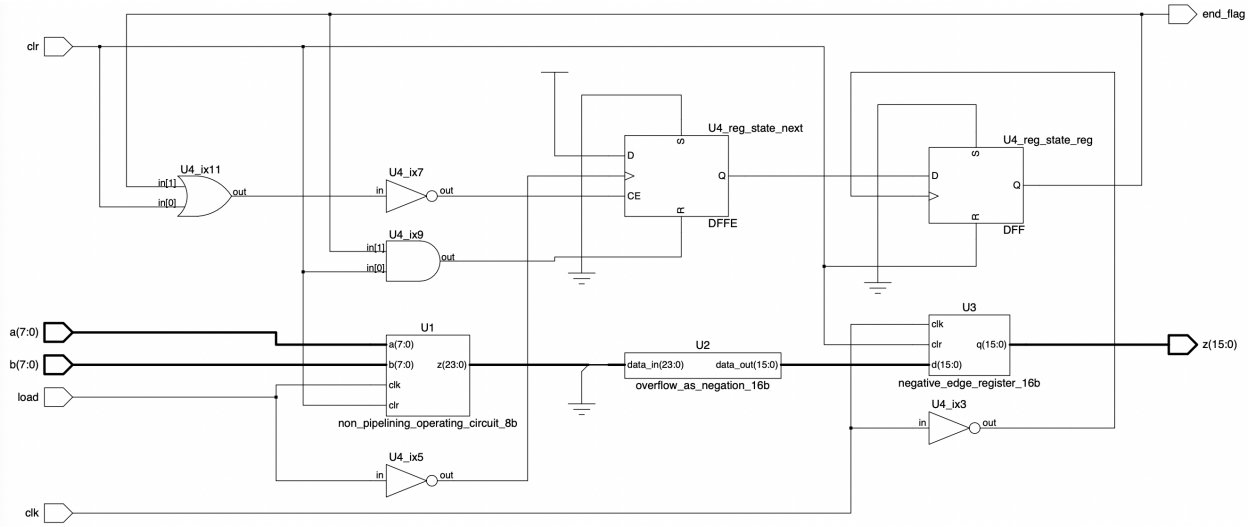


Note. Sigal with suffix "_after_reg" means the value of the signal is an output of a register

Output Process

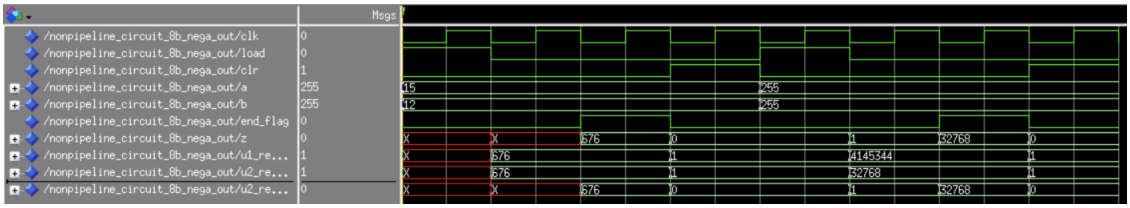
In the non-pipelined implementation, negation output is used to meet the project requirements of 16-bit *Z*-port outputs, which just represent the result as overflow. The implementation of the whole circuit is shown in Figure 40. The test simulation results for the negation output are shown in Figure 41

Figure 40
Synthesized RTL Diagram of Non-pipelined Circuit



Note. A 1 stage *END_FLAG* generator is used.

Figure 41
Simulation Wave Diagram of Non-pipelined Circuit



Note. Sigal 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 Handler

By this design, the circuit will obtain the result in 4 synchronized clock cycles after the falling edge event of the *load* signal. Table 5 shows how the registers in this pipelined circuit will latch the values. Figure 43 presents the RTL description of the circuit and its arithmetic block.

Table 5
A Visual Representation Table of the Pipelined Processe with Negation Overflow Handler

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 42 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.

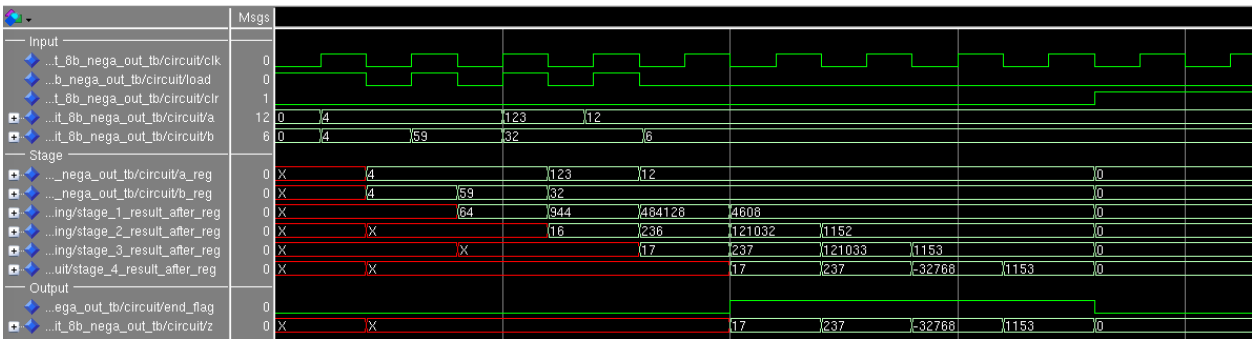
8-bit Operands Implementation with Output Separation Overflow Handler

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 6 presents how the z will take over the data and Figure 44 shows the simulation of the design. Figure 45 presents the RTL description of the decign.

Figure 42

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

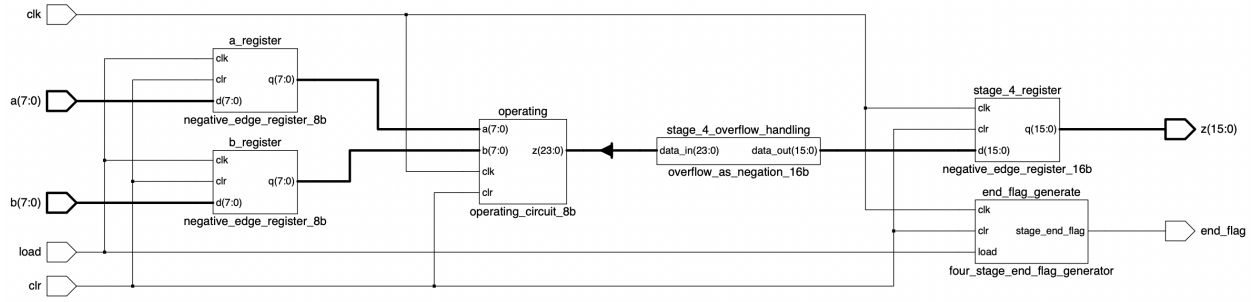


Note. Sigal with suffix "_after_reg" means the value of the signal is an output of a register.

Figure 43

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*

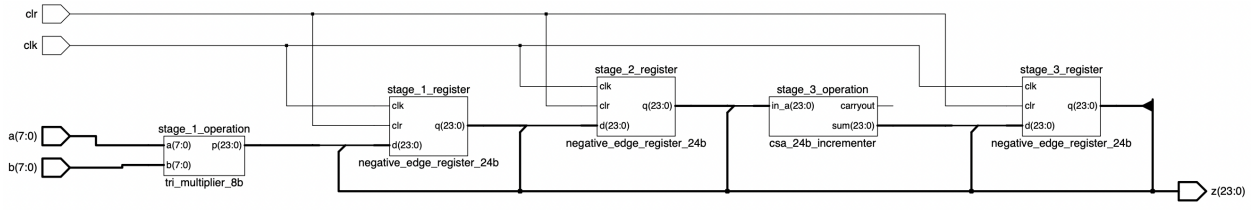


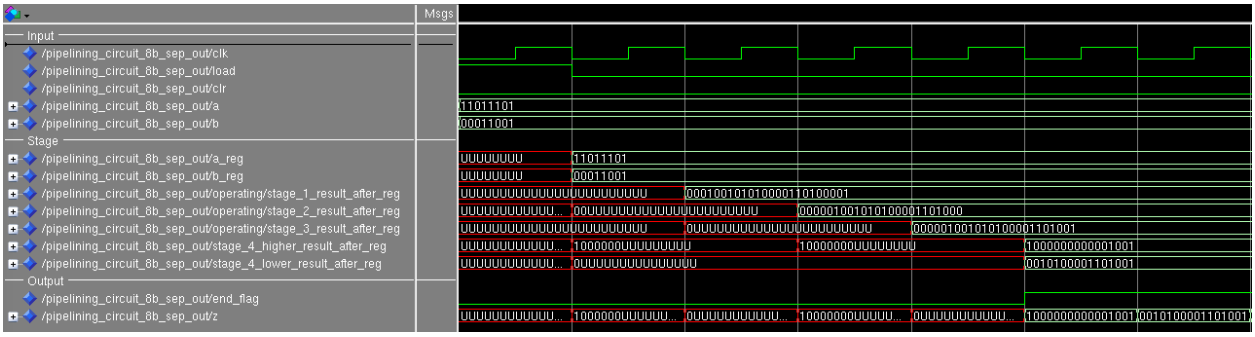
Table 6

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

clk	Stage 3 Reg	Stage 4 Reg for Higher	Stage 4 Reg for Lower	Z
...				
3	000110110000110100101101	-	-	-
4	-	1000000000110110	0000110100101101	1000000000110110
6	-	1000000000110110	0000110100101101	0000110100101101

Figure 44

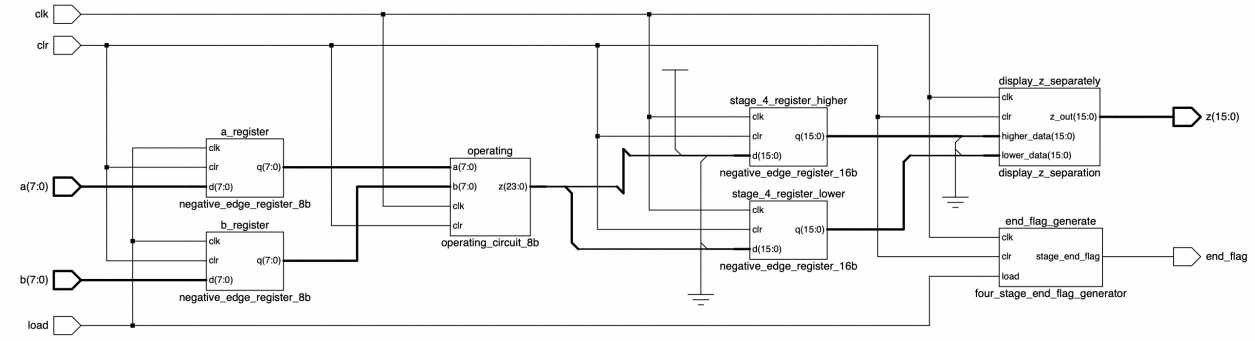
Simulation Wave Diagram of the 8-bit Pipelined ALU Circuit with Separation Output



Note. Sigal 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 45

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

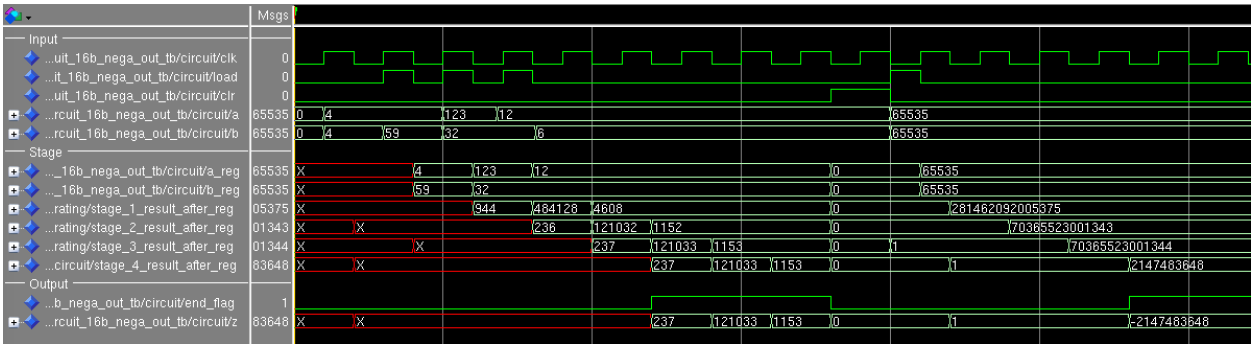


16-bit Operands Implementation and Simulation

By now there will no necessary for discussing the implementation of the extended design since all details were presented in the previous sections. Figure 46 presents the testbench simulation of the bit-width extended circuit.

Figure 46

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



Note. Sigal with suffix "_after_reg" means the value of the signal is an output of a register.

Synthesis and Analysis of the Arithmetic Circuit

Introduction

The arithmetic circuit is synthesized for implementation on a field-programmable grid array (FPGA). Xilinx Virtex-II Pro is used as the FPGA in this project. Mentor Graphics PrecisionRTL is used to generate and synthesize the circuits, which also details the area usage and can perform timing analysis if needed. Xilinx ISE is used for getting the timing and power information.

Results for the Non-Pipelined Version

Implementation with Negation Output

Area. When synthesizing the non-pipelined version, the total number of slices used was 377 or 4.06% of all available logic slices. The full report from Mentor Graphics PrecisionRTL regarding the area and resource used is shown in Appendix Area Report for Non-Pipelined Version with Negation Output.

Power. The power information was performed with Xilinx ISE. The results are shown in the Table 7.

Table 7
Power Information for Non-pipelined Version

Source	**Voltage(V)**	**Power(W)**
Vccint	1.5	0.06
Vccaux	2.5	0.025
Vcco25	2.5	0.003
		Total power: 0.088W

Results for the Pipelined Version

Implementation with Negation Output

Area. When synthesizing the pipelined negation output version, the total number of slices used was 380 or 4.09% of all available logic slices. The full report from Mentor Graphics PrecisionRTL regarding the area and resource used is shown in Appendix Area Report for Pipelined Version with Negation Output.

Power. The power information was performed with Xilinx ISE. The results are shown in the Table 8.

Implementation with Separation Output

When synthesizing the pipelined separation output version, the total number of slices used was 379 or 4.08% of all available logic slices. The full report from Mentor Graphics PrecisionRTL regarding the area and resource used is shown in Appendix Area Report for Pipelined Version with Separation Output.

Table 8
Power Information for Pipelined Version with Negation Output

Source	**Voltage**(V)	**Power**(W)
Vccint	1.5	0.06
Vccaux	2.5	0.025
Vcco25	2.5	0.003
		Total power: 0.088W

Power. The power information was performed with Xilinx ISE. The results are shown in the Table 9.

Table 9
Power Information for Pipelined Version with Separation Output

Source	**Voltage**(V)	**Power**(W)
Vccint	1.5	0.06
Vccaux	2.5	0.025
Vcco25	2.5	0.003
		Total power: 0.088W

Appendices

Area Report for Non-Pipelined Version with Negation Output

```
//  Precision RTL Synthesis  64-bit 2016.1.0.15 (Production Release) Wed Jun  8 09:35:56 PDT 2016
//
//  Copyright (c) Mentor Graphics Corporation, 1996-2016, All Rights Reserved.
//              Portions copyright 1991-2008 Compuware Corporation
//                        UNPUBLISHED, LICENSED SOFTWARE.
//             CONFIDENTIAL AND PROPRIETARY INFORMATION WHICH IS THE
//           PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS
//
//  Running on Linux hu_ju@grace.encs.concordia.ca #1 SMP Tue Oct 12 08:40:46 CDT 2021
    3.10.0-1160.45.1.el7.x86_64 x86_64
//
//  Start time Mon Dec  6 09:38:30 2021

*****************************************************************
Device Utilization for 2VP20ff896
*****************************************************************
Resource                          Used    Avail    Utilization
-----------------------------------------------------------------
IOs                               36      556       6.47%
Global Buffers                    2       16       12.50%
LUTs                              754     18560     4.06%
CLB Slices                        377     9280      4.06%
Dffs or Latches                   34      20228     0.17%
Block RAMs                        0       88        0.00%
Block Multipliers                 0       88        0.00%
Block Multiplier Dffs             0       3168      0.00%
GT_CUSTOM                         0       8         0.00%
-----------------------------------------------------------------

*****************************************************************

Library: work     Cell: nonpipeline_circuit_8b_nega_out     View: arch

*****************************************************************

  Cell                            Library   References      Total Area

 BUFGP                            xcv2p      2 x
 CSA_24bit                        work       1 x     40      40 gates
                                                     40      40 LUTs
 CSA_24bit_unfolded0              work       1 x     41      41 gates
                                                     39      39 LUTs
 FDCE_1                           xcv2p      1 x      1       1 Dffs or Latches
 FDC_1                            xcv2p     33 x      1      33 Dffs or Latches
 IBUF                             xcv2p     17 x
 LUT2                             xcv2p      2 x      1       2 LUTs
 LUT3                             xcv2p     14 x      1      14 LUTs
 LUT4                             xcv2p      5 x      1       5 LUTs
 OBUF                             xcv2p     17 x
 VCC                              xcv2p      1 x
 r4b_multiplier_8b                work       1 x      7       7 MUXF5
                                                    207     207 LUTs
                                                    218     218 gates
 r4b_multiplier_8b_unfolded0      work       1 x      2       2 MUXF5
                                                    214     214 LUTs
                                                    229     229 gates
 r4b_multiplier_8b_unfolded1      work       1 x      4       4 MUXF5
                                                    233     233 LUTs
                                                    252     252 gates

 Number of ports :                            36
 Number of nets :                            236
 Number of instances :                        97
 Number of references to this view :           0

Total accumulated area :
 Number of Dffs or Latches :                  34
 Number of LUTs :                            754
 Number of MUXF5 :                            13
 Number of gates :                           803
 Number of accumulated instances :           840
```

```
****************************
 IO Register Mapping Report
****************************
Design: work.nonpipeline_circuit_8b_nega_out.arch
```

Port	Direction	INFF	OUTFF	TRIFF
clk	Input			
load	Input			
clr	Input			
a(7)	Input			
a(6)	Input			
a(5)	Input			
a(4)	Input			
a(3)	Input			
a(2)	Input			
a(1)	Input			
a(0)	Input			
b(7)	Input			
b(6)	Input			
b(5)	Input			
b(4)	Input			
b(3)	Input			
b(2)	Input			
b(1)	Input			
b(0)	Input			
end_flag	Output			
z(15)	Output			
z(14)	Output			
z(13)	Output			
z(12)	Output			
z(11)	Output			
z(10)	Output			
z(9)	Output			
z(8)	Output			
z(7)	Output			
z(6)	Output			
z(5)	Output			
z(4)	Output			
z(3)	Output			
z(2)	Output			
z(1)	Output			

```
+-------------+-----------+----------+----------+----------+
| z(0)        | Output    |          |          |          |
+-------------+-----------+----------+----------+----------+
Total registers mapped: 0
```

Area Report for Pipelined Version with Negation Output

```
//  Precision RTL Synthesis  64-bit 2016.1.0.15 (Production Release) Wed Jun  8 09:35:56 PDT 2016
//
//  Copyright (c) Mentor Graphics Corporation, 1996-2016, All Rights Reserved.
//              Portions copyright 1991-2008 Compuware Corporation
//                        UNPUBLISHED, LICENSED SOFTWARE.
//             CONFIDENTIAL AND PROPRIETARY INFORMATION WHICH IS THE
//           PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS
//
//  Running on Linux hu_ju@grace.encs.concordia.ca #1 SMP Tue Oct 12 08:40:46 CDT 2021
    3.10.0-1160.45.1.el7.x86_64 x86_64
//
//  Start time Mon Dec  6 09:38:30 2021

*****************************************************************
Device Utilization for 2VP20ff896
*****************************************************************
Resource                          Used    Avail    Utilization
-----------------------------------------------------------------
IOs                               36      556       6.47%
Global Buffers                    2       16       12.50%
LUTs                              760     18560     4.09%
CLB Slices                        380     9280      4.09%
Dffs or Latches                   106     20228     0.52%
Block RAMs                        0       88        0.00%
Block Multipliers                 0       88        0.00%
Block Multiplier Dffs             0       3168      0.00%
GT_CUSTOM                         0       8         0.00%
-----------------------------------------------------------------

*****************************************************************

Library: work    Cell: pipelining_circuit_8b_nega_out    View: arch

*****************************************************************

  Cell                    Library  References      Total Area

 BUFGP                    xcv2p     2 x
 FDCE_1                   xcv2p     1 x       1      1 Dffs or Latches
 FDCPE_1                  xcv2p     1 x       1      1 Dffs or Latches
 FDC_1                    xcv2p    34 x       1     34 Dffs or Latches
 FDE_1                    xcv2p     3 x       1      3 Dffs or Latches
 GND                      xcv2p     1 x
 IBUF                     xcv2p    17 x
 LUT2                     xcv2p     5 x       1      5 LUTs
 LUT3                     xcv2p    18 x       1     18 LUTs
 LUT4                     xcv2p     4 x       1      4 LUTs
 OBUF                     xcv2p    17 x
 VCC                      xcv2p     1 x
 operating_circuit_8b     work      1 x      67     67 Dffs or Latches
                                             781    781 gates
                                             734    734 LUTs
                                              13     13 MUXF5

 Number of ports :                            36
 Number of nets :                            146
 Number of instances :                       105
 Number of references to this view :           0

Total accumulated area :
 Number of Dffs or Latches :                 106
 Number of LUTs :                            760
 Number of MUXF5 :                            13
 Number of gates :                           811
 Number of accumulated instances :           920


****************************
```

IO Register Mapping Report
****
Design: work.pipelining_circuit_8b_nega_out.arch

Port	Direction	INFF	OUTFF	TRIFF
clk	Input			
load	Input			
clr	Input			
a(7)	Input			
a(6)	Input			
a(5)	Input			
a(4)	Input			
a(3)	Input			
a(2)	Input			
a(1)	Input			
a(0)	Input			
b(7)	Input			
b(6)	Input			
b(5)	Input			
b(4)	Input			
b(3)	Input			
b(2)	Input			
b(1)	Input			
b(0)	Input			
end_flag	Output			
z(15)	Output			
z(14)	Output			
z(13)	Output			
z(12)	Output			
z(11)	Output			
z(10)	Output			
z(9)	Output			
z(8)	Output			
z(7)	Output			
z(6)	Output			
z(5)	Output			
z(4)	Output			
z(3)	Output			
z(2)	Output			
z(1)	Output			
z(0)	Output			

```
Total registers mapped: 0
```

Area Report for Pipelined Version with Separation Output

```
//  Precision RTL Synthesis  64-bit 2016.1.0.15 (Production Release) Wed Jun  8 09:35:56 PDT 2016
//
//  Copyright (c) Mentor Graphics Corporation, 1996-2016, All Rights Reserved.
//              Portions copyright 1991-2008 Compuware Corporation
//                        UNPUBLISHED, LICENSED SOFTWARE.
//           CONFIDENTIAL AND PROPRIETARY INFORMATION WHICH IS THE
//         PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS
//
//  Running on Linux hu_ju@poise.encs.concordia.ca #1 SMP Tue Oct 12 08:40:46 CDT 2021
    3.10.0-1160.45.1.el7.x86_64 x86_64
//
//  Start time Sat Dec  4 21:22:54 2021

************************************************************
Device Utilization for 2VP20ff896
************************************************************
Resource                          Used    Avail    Utilization
------------------------------------------------------------
IOs                               36      556       6.47%
Global Buffers                    2       16       12.50%
LUTs                              757     18560     4.08%
CLB Slices                        379     9280      4.08%
Dffs or Latches                   115     20228     0.57%
Block RAMs                        0       88        0.00%
Block Multipliers                 0       88        0.00%
Block Multiplier Dffs             0       3168      0.00%
GT_CUSTOM                         0       8         0.00%
------------------------------------------------------------

*****************************************************************

Library: work    Cell: pipelining_circuit_8b_sep_out    View: arch

*****************************************************************

  Cell                    Library  References      Total Area

 BUFGP                    xcv2p     2 x
 FDCE_1                   xcv2p     1 x       1      1 Dffs or Latches
 FDCPE_1                  xcv2p     1 x       1      1 Dffs or Latches
 FDC_1                    xcv2p    42 x       1     42 Dffs or Latches
 FDE_1                    xcv2p     3 x       1      3 Dffs or Latches
 FD_1                     xcv2p     1 x       1      1 Dffs or Latches
 GND                      xcv2p     1 x
 IBUF                     xcv2p    17 x
 LUT1                     xcv2p     1 x       1      1 LUTs
 LUT2                     xcv2p     4 x       1      4 LUTs
 LUT3                     xcv2p    11 x       1     11 LUTs
 LUT4                     xcv2p     9 x       1      9 LUTs
 OBUF                     xcv2p    17 x
 VCC                      xcv2p     1 x
 operating_circuit_8b     work      1 x      67     67 Dffs or Latches
                                             781    781 gates
                                             734    734 LUTs
                                              13     13 MUXF5

 Number of ports :                            36
 Number of nets :                            153
 Number of instances :                       112
 Number of references to this view :           0

Total accumulated area :
 Number of Dffs or Latches :                 115
 Number of LUTs :                            757
 Number of MUXF5 :                            13
 Number of gates :                           809
 Number of accumulated instances :           927


****************************
 IO Register Mapping Report
```

```
****************************
Design: work.pipelining_circuit_8b_sep_out.arch

+-------------+-----------+---------+---------+---------+
| Port        | Direction |   INFF  |  OUTFF  |  TRIFF  |
+-------------+-----------+---------+---------+---------+
| clk         | Input     |         |         |         |
+-------------+-----------+---------+---------+---------+
| load        | Input     |         |         |         |
+-------------+-----------+---------+---------+---------+
| clr         | Input     |         |         |         |
+-------------+-----------+---------+---------+---------+
| a(7)        | Input     |         |         |         |
+-------------+-----------+---------+---------+---------+
| a(6)        | Input     |         |         |         |
+-------------+-----------+---------+---------+---------+
| a(5)        | Input     |         |         |         |
+-------------+-----------+---------+---------+---------+
| a(4)        | Input     |         |         |         |
+-------------+-----------+---------+---------+---------+
| a(3)        | Input     |         |         |         |
+-------------+-----------+---------+---------+---------+
| a(2)        | Input     |         |         |         |
+-------------+-----------+---------+---------+---------+
| a(1)        | Input     |         |         |         |
+-------------+-----------+---------+---------+---------+
| a(0)        | Input     |         |         |         |
+-------------+-----------+---------+---------+---------+
| b(7)        | Input     |         |         |         |
+-------------+-----------+---------+---------+---------+
| b(6)        | Input     |         |         |         |
+-------------+-----------+---------+---------+---------+
| b(5)        | Input     |         |         |         |
+-------------+-----------+---------+---------+---------+
| b(4)        | Input     |         |         |         |
+-------------+-----------+---------+---------+---------+
| b(3)        | Input     |         |         |         |
+-------------+-----------+---------+---------+---------+
| b(2)        | Input     |         |         |         |
+-------------+-----------+---------+---------+---------+
| b(1)        | Input     |         |         |         |
+-------------+-----------+---------+---------+---------+
| b(0)        | Input     |         |         |         |
+-------------+-----------+---------+---------+---------+
| end_flag    | Output    |         |         |         |
+-------------+-----------+---------+---------+---------+
| z(15)       | Output    |         |         |         |
+-------------+-----------+---------+---------+---------+
| z(14)       | Output    |         |         |         |
+-------------+-----------+---------+---------+---------+
| z(13)       | Output    |         |         |         |
+-------------+-----------+---------+---------+---------+
| z(12)       | Output    |         |         |         |
+-------------+-----------+---------+---------+---------+
| z(11)       | Output    |         |         |         |
+-------------+-----------+---------+---------+---------+
| z(10)       | Output    |         |         |         |
+-------------+-----------+---------+---------+---------+
| z(9)        | Output    |         |         |         |
+-------------+-----------+---------+---------+---------+
| z(8)        | Output    |         |         |         |
+-------------+-----------+---------+---------+---------+
| z(7)        | Output    |         |         |         |
+-------------+-----------+---------+---------+---------+
| z(6)        | Output    |         |         |         |
+-------------+-----------+---------+---------+---------+
| z(5)        | Output    |         |         |         |
+-------------+-----------+---------+---------+---------+
| z(4)        | Output    |         |         |         |
+-------------+-----------+---------+---------+---------+
| z(3)        | Output    |         |         |         |
+-------------+-----------+---------+---------+---------+
| z(2)        | Output    |         |         |         |
+-------------+-----------+---------+---------+---------+
| z(1)        | Output    |         |         |         |
+-------------+-----------+---------+---------+---------+
| z(0)        | Output    |         |         |         |
+-------------+-----------+---------+---------+---------+
Total registers mapped: 0
```

Figure 47
Synthesized RTL Diagram of 16-bit Radix-4 Booth Multiplier