Project On

DESIGN OF BOOTH MULTIPLIER USING RIPPLE CARRY ADDER

By

Chinmay Kumar Behera(110ec0157)
Sandip Kumar Barman(110ei0235)
B.Tech 4th year

Project Guide
Prof. M. Nurul Islam
Asst. Professor
Dept. of Electronics & Communication Engineering

Submitted to

the Dept. of Electronics & Communication Engineering
National Institute of Technology, Rourkela

NATIONAL INSTITUTE OF TECHNOLOGY, ROURKELA
Rourkela,
Sundargarh-769008
Website- www.nitrkl.ac.in

CERTIFICATE

This is to certify that the thesis entitled "**DESIGN OF BOOTH MULTIPLIER USING RIPPLE CARRY ADDER**" submitted by **Mr. Chinmay Kumar Behera (110EC0157) and Mr. Sandip Kumar Barman(110EI0235)**, Final year students of Electronics and communication Engineering department, in partial fulfilments of the requirements for the award of Bachelor of Technology Degree in Electronics and Communication Engineering at National Institute of Technology, Rourkela is an authentic work carried out by them under my supervision and guidance.

To the best of my knowledge, the matter embodied in thesis has not been submitted to any other university/ Institute for the award of any degree or Diploma.

Prof M. NURUL ISLAM
Department of E.C.E
National Institute of Technology
Rourkela- 769008

ACKNOWLEDGEMENT

We would like to articulate our profound gratitude and indebtedness to our project guide Prof. M. Nurul Islam who has always been a constant motivation and guiding factor throughout the project time in and out as well. It has been a great pleasure for us to get an opportunity to work under him and complete the project successfully.

We wish to extend our sincere thanks to Prof. Dr. S. Meher, Head of our Department, for approving our project work with great interest.

We would like to mention Mr. Deepu S P for his cooperation and constantly rendered assistance.

An undertaking of this nature could never have been attempted without our reference to and inspirations from the works of others whose details are mentioned in references section. We acknowledge our indebtedness to all of them. Last but not the least, our sincere thanks to all our friends who have patiently extended all sorts of help for accomplishing this undertaking.

Chinmay Kumar Behera
(110EC0157)
NIT Rourkela

Sandip Kumar Barman
(110EI0235)
NIT Rourkela

CONTENTS

ABSTRACT

Modern IC Technology focuses on the planning of ICs considering additional space improvement and low power techniques. Multiplication may be a heavily used operation that figures conspicuously in signal process and scientific applications. Multiplication may be a terribly hardware intensive subject and thus we as users area unit largely involved with obtaining low-power, smaller space and better speed. The foremost necessary concern in classic multiplication largely accomplished by K-cycles of shifting and adding, is to hurry up underlying multi-operand addition of partial product. During this project we'll design the Booth multiplier using Ripple Carry Adder architecture. Additionally multipliers are designed for each radix-2 and radix-4. Results can show that the multiplier is able to multiply two 32 bit signed numbers and how this technique reduces the number of partial products, which is an important factor to be achieved in this project.

CHAPTER 1

INTRODUCTION

- **MOTIVATION**
- **MULTIPLIER DESIGN**
- **PROGRAMMING LANGUAGE AND ANALYSIS TOOLS USED**
- **REASEARCH APPROACH**

1.1 MOTIVATION

Day by day IC technology is obtaining additional advanced in terms of style and its performance analysis. A quicker style with lower power consumption and smaller space is implicit to the trendy electronic styles. Unceasing advancement in electronics style technology makes improved use of energy, code knowledge with success, communicate info way more firm, etc. significantly, several of those technologies address low-power consumption to fulfil the necessities of assorted transportable applications. In these application systems, a multiplier could be a basic arithmetic unit and wide employed in circuits that the multiplication method ought to be optimized properly. Multipliers typically have extended latency, huge space and consume substantial quantity of power. Thus low-power number style has become a very important half in VLSI system style. Everyday new approaches square measure being developed to style low-power multipliers at technological, physical, circuit and logic levels. Since multiplier is mostly the slowest component during a system, the system's performance is decided by performance of the multiplier. Additionally multipliers square measure the foremost space intense entity during a style. Therefore, optimizing speed and space of a multiplier could be a major style issue these days. However, space and speed square measure typically conflicting constraints in order that rising speed ends up in larger areas and vice-versa. Additionally space and power consumption of a circuit square measure linearly correlate. Therefore a compromise has got to be wiped out speed of the circuit for a larger improvement in reduction of space and power.

For implementing a digital number an oversized form of pc arithmetic algorithms may be used. Most techniques take into thought generating a collection of partial merchandise, and so adding the partial merchandise along once they need been shifted. During a number to extend its speed, the amount of partial product to be generated ought to be reduced. A better illustration base effectively indicates to fewer digits. Thus, a single-digit multiplication algorithmic rule necessitates fewer cycles as we tend to begin moving to a lot of higher radices, that mechanically ends up in a lesser variety of partial merchandise. Many algorithms are developed for this purpose like Booth's algorithmic rule, Wallace Tree methodology etc. For the summation method many adder architectures square measure on the market viz. Ripple Carry Addition, Carry Look-ahead Addition, Carry Save Addition etc. however to scale back the facility consumption the summation design of the number ought to be rigorously chosen.

1.2 MULTIPLIER DESIGN

Multiplication is thought of to incorporates 3 basic steps: generation of partial product (PPG), partial product reduction (PPR), and at last at the top addition of carry propagate(CPA).In general we've got combinatory and ordered multiplier factor implementations. Here we have a tendency to area unit taking into thought the combinatory case solely, as a result of the size of integration currently has become large enough to begin accommodating parallel multiplier factor applications in digital VLSI circuits. Completely different multiplication algorithms vary within the approaches of generation and reduction of Partial product and also the addition method. So as to diminish the amount of PPs concerned and so reduce the area/delay of the circuit, one quantity is sometimes recoded into high-radix digit sets. One amongst the foremost used and widespread radix-2n algorithmic rule is that the radix-4 that features a set of digits given by for PPG. For PPR, 2 decisions exist which might be implemented: reduction by rows, which might be performed by taking into thought Associate in Nursing adder array and reduction by columns, which might be performed by taking into thought a counter array. The closing method of addition necessitates a quick adder arrangement as a result of it's on the crucial path. In an exceedingly few cases, last summation is delayed if it's valuable to stay redundant results from PPG to hold out any arithmetic operations.

1.3 PROGRAMMING LANGUAGE AND ANALYSIS TOOLS USED

To write program for the implementation of any digital circuit there square measure varied languages accessible, referred to as Hardware Description Language e.g. Verilog, VHDL. For our style we've used VHDL (Very High Specific microcircuit HDL) for programming. VHDL is one among the common techniques utilized in digital system aborning method. The technique is enforced in program mistreatment bound package that carries out simulation and examination of the designed system. The designer solely must describe the digital circuit design in matter type which may take away while not the trouble to change the hardware. VHDL is very most well-liked as a result of this system has the power to scale back value and time, is simple to troubleshoot, portable, lots of platforms package support the VHDL operate and high references square measure accessible. We tend to used XILINX ten.1 platform to jot down our programs. All the RTL simulations has been done mistreatment this package solely. Conjointly for delay report the synthesis tool embedded in Xilinx was used.

We used for Scirocco and VirSim, that square measure logic simulators, for the practicality simulation of our style. Conjointly we tend to used Synopsys style Vision tool to estimate power of all our arithmetic circuits. Synopsys style Vision could be a logic synthesis tool. It takes alpha-lipoprotein styles and synthesizes them to gate-level net-lists. Conjointly it supports each Verilog and VHDL. It will synthesize generic gates or different style libraries. The tool exists within a interface and command version. The interface version is thought as style vision and therefore the command version is referred as dc_shell-xg-t. For each space and power estimation we tend to used style Vision. The essential steps for analyzing a style are:

Analyze: This step begin checking the planning files for syntax. We can conjointly save modules (Verilog) associate degree entities (VHDL) in an intermediate format into an area folder.

Elaborate: we will build a style from the intermediate format files created within the previous Analyze step.

Compile: this is often the synthesizing step, wherever we will map the planning to a gate library or cell library.

Save: once aggregation a style we will save the synthesized style into alpha-lipoprotein or different formats. Synthesized styles square measure elementary for making ASICS or effecting completely different simulations for temporal order and power. After compilation mistreatment commands like report_power or report_area we will get power and space consequently.

1.4 RESEARCH APPROACH

Speed of the multiplier is highly dependent upon the number of partial products generated and the adder architecture used to add these partial products. The main intention of the project is to use booth multiplier algorithm for designing the binary multiplier with the help of Ripple carry adder. The reason for using the booth’s algorithm is that, using booth’s algorithm we can reduce the number of partial products during multiplication. The adder here we have used is ripple carry adder. This adder has a very simple architecture and is very easy to implement. As here we are dealing with high bits, this adder is very useful because of its simple architecture. If we see overall including the adder and booth’s algorithm we get a binary multiplier which has comparatively high speed because of less partial products and less power consumption because of the adder architecture we have used. We have checked the results for both signed and unsigned numbers.

CHAPTER 2

ADDERS

- **ADDERS CLASSIFICATION**

- **RIPPLE CARRY ADDER**

- **ANALYSIS OF RIPPLE CARRY ADDER**

2.1 ADDERS CLASSIFICATION

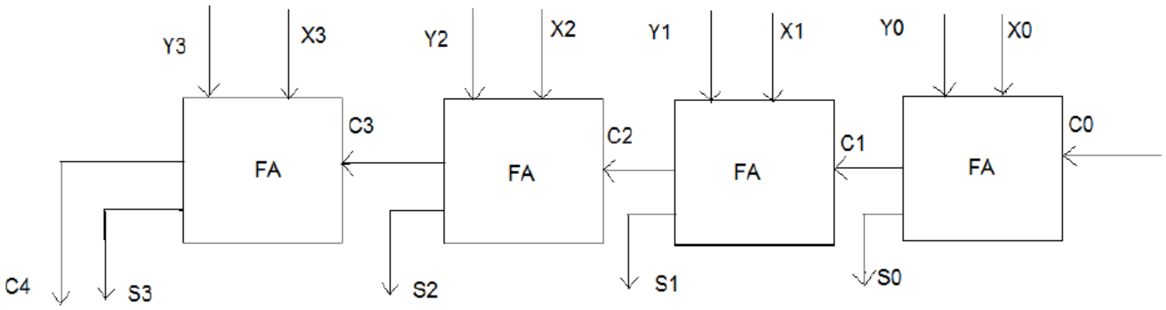
Addition is one in every of the foremost normally used mathematical process in silicon chip, digital signal processor etc. It also can be used as a building block for synthesis of all alternative arithmetic operations. Therefore, as so much because the economical implementation of associate degree arithmetic unit is bothered, the binary adder structure becomes an awfully essential hardware unit. In any book on pc arithmetic, we will observe that there happens an oversized variety of quite completely different circuit architectures relating different performance characteristics. Whereas adders is made for lots of numerical expressions like Binary-coded decimal or excess-3, the foremost oft used adders operate numbers that area unit binary. In sure cases wherever two's complement is being employed to represent negative numbers, it's trivial to convert associate degree adder into associate degree adder-subtractor.

Although several researches associated with the binary adder structures are allotted, the studies supported their comparative performance analysis area unit solely quite few in variety. During this project, assessments of the classified binary adder architectures area unit given. From the large member of adders we've got, we tend to enforce the VHDL (Hardware Description Language) code for Ripple-carry adder. Throughout consequent section, we offer you with a quick description of the studied adder design.

2.2 RIPPLE CARRY ADDERS (RCA)

This well-liked adder design, ripple carry adder consists of cascaded full adders as shown in figure2.1. It is shaped by cascading full adder blocks nonparallel with each other. The output carry of 1 stage is fed on to the input carry of following stage. AN N-bit parallel adder needs N full adders.

FIGURE 2.1



The given adder design isn't terribly economical once sizable amount of bits square measure used. The gate delay will simply be calculated by inspecting the total adder circuit. we all know that every full adder needs 3 levels of logic. Considering a 64-bit ripple-carry adder, we all know that it's sixty four full adders, therefore the crucial path (worst case) delay is three (from input to hold just in case of the primary adder) + sixty three * two (for carry propagation within the later adders) = 127 gate delays.

2.3 ANALYSIS OF RIPPLE CARRY ADDER ADDERS

We know that combinational logic circuits cannot cipher the outputs instantly. There's some delay between the time the inputs square measure sent to the circuit, and therefore the time the output is computed.

Let's say the delay is T units of your time.

Suppose we wish to implement associate degree n-bit ripple carry adder. Since associate degree n-bit ripple carry adder consists of n adders, there'll be a delay of nongovernmental organization. This is often O(n) delay. While the adders square measure operating in parallel, the carrys should "ripple" their means from the smallest amount vital bit and work their thanks to the foremost vital bit. It takes T units for the perform of the right column to create it as input to the adder within the next to right column. Thus, the carries abate the circuit, creating the addition linear with the quantity of bits within the adder. This is not a giant drawback, usually, as a result of hardware adders square measure mounted in size. They add, say, thirty two bits at a time. There is not associate degree thanks to create an adder add an impulsive variety of bits. It will be exhausted code, not in hardware. In effect, this is often what makes hardware "hard". It is not suitable to alter. Even though there square measure a set variety of bits to feature in an adder, there square measure ways that to create the adder add a lot of quickly (at least, by a constant).

CHAPTER 3

THE MULTIPLIERS

- **BASIC MULTIPLICATION ALGORITHM**
- **BOOTH'S ENCODING**
- **MODIFIED BOOTH'S ALGORITHM**

3.1 BASIC ALGORITHM FOR BINARY MULTIPLICATION

A Binary number is associate degree device utilized in digital physics or in an exceedingly pc or different electronic devices to hold out multiplication of 2 numbers represented in binary format. It's engineered mistreatment binary adders. The foremost basic technique involves generating a collection of partial merchandise, so summing the partial merchandise at the same time. This method is comparable to the tactic that is instructed to lower classes' students in class for conducting long multiplication on base-10 integers, however has been changed here for application to a base-2 (binary) numeral system.

The rules for binary multiplication are expressed as given:

1. If the number digit is one, the number is derived down and it provides the merchandise.
2. If the number digit is zero then we have a tendency to get a product that is additionally zero.

For planning such a number circuit we should always have the electronic equipment to hold out or do the subsequent four things:

1. It ought to be capable of recognizing whether or not a little is zero or one.
2. It ought to be capable of shifting the left partial product.
3. It ought to be capable of adding all the partial-products to provide the merchandise as a addition of the partial merchandise.
4. It ought to examine sign bits and if they're similar, the sign of the merchandise are going to be a Positive illustration and if the sign bits are opposite then the merchandise are going to be negative. The sign little bit of the merchandise that has been keep with the higher than criteria ought to be displayed together with the merchandise.

From the higher than discussion we will observe that it's not necessary to attend till all the partial merchandise are shaped before polishing off the add. in truth the addition of the partial merchandise is allotted as before long as a partial product is created.

3.2 BOOTH'S ENCODING

Booth's encryption or Booth's multiplication rule could be a multiplication algorithm which might multiply 2 signed binary numbers during a two's complement notation. Booth's rule has the power to perform fewer additions and subtractions as compared to traditional multiplication rule. It's AN encryption method which might be accustomed minimize the no of partial product during a multiplication method. it's based mostly upon the relation

$$2^n = 2^{n-1} - 2^n$$

Example:

0 0 1 1 1 1 1 1 0 0

+1 -1

+1 -1

+1 -1

+1 -1

+1 -1

+1 -1

0 +1 0 0 0 0 0 -1 0 0

Booth's algorithmic program examines consecutive bits of the N-bit number Y in signed two's complement illustration, which has Associate in Nursing implicit bit below the smallest amount important bit, y-1 = 0. For every bit Lolo, as i runs from zero to N-1, the bits Lolo and yi-1 square measure thought of. Once these 2 bits square measure equal, the merchandise accumulator P stays unchanged. Wherever Lolo = zero and yi-1 = one, the number times 2i is additional to P; and wherever Lolo = one and yi-1 = zero, the number times 2i gets subtracted from P. The ultimate price of P are the signed product.

The illustration of the number and products don't seem to be specified; usually, these also are in two's complement illustration, sort of a number, however any system of numeration that supports addition and subtraction can work furthermore. The order of the steps isn't determined. Generally, it issue from LSB to mutual savings bank, beginning at i = 0; the multiplication by 2i is then replaced by progressive shifting of the P accumulator to the proper between steps; low bits are shifted out, and resultant additions or subtractions will then be done simply on the best N bits of P. There square measure several variations and optimizations on these details.

The algorithmic program is commonly delineate as changing strings of 1's within the number to a high-order +1 and a low-order –1 at the ends of the string. once the string runs through the mutual savings bank, there's no high-order +1, and also the web result is interpretation as a negative of the acceptable price.

RADIX-2 ALGORITHM IMPLEMENTATION

Let x be the quantity of bits of the number, and y be the quantity of bits of the multiplier:

• Draw a grid of 3 rows, every with columns for x + y + one bits. Label the lines severally A (add), S (subtract), and P (product).

• In two's complement notation, fill the primary x bits of every line with :

o A: the number

o S: negative of the multiplicand(2's complement format)

o P: zeroes

• Fill future y bits of every line with :

o A: zeroes

o S: zeroes

o P: the number

• Fill the last little bit of every line with a zero.

For example think about the given 2 numbers: 5 and -2.

On ending the higher than directions we'd notice the subsequent values of A, S and P.

A = 0101 0000 0

S = 1011 0000 0

P = 0000 1110 0

Now do each of those steps y times :

If the last 2 bits within the product are:

- 00 or 11: do nothing.
- 01: P = P + A. Ignore any overflow.
- 10: P = P + S. Ignore any overflow.
- Arithmetically shift the merchandise right one position.
- Drop the primary (we count from right to left once handling bits) bit from the merchandise for the ultimate result.
- Do each of those steps y times :

For Example: Find 5 × -2, with x = 4 and y = 4:

We get:

A = 0101 0000 0

S = 1011 0000 0

P = 0000 1110 0

Perform the loop four times:

1- P = 0000 1110 0. The last two bits are 00.

P = 0000 0111 0. A right shift.

2- P = 0000 0111 0. The last two bits are 10.

P = P+S. Right shift.

3- P = 0000 0011 0. The last two bits are 10.

P = 1101 1001 0.

P = P + A. Right shift.

4- P = 1110 1101 1. The last two bits are 11.

P = 1111 0110 1. Right shift.

The final product is 1111 0110, which is -10.

3.3 MODIFIED BOOTH'S ALGORITHM

One of the various solutions of realizing high speed multipliers is enhancing correspondence that helps in decreasing the quantity of ulterior calculation levels. the initial version of Booth formula (Radix-2) had 2 specific drawbacks. They were:

• the quantity of add-subtract operations and shift operations become variable and causes inconvenience in coming up with parallel multipliers.

• The formula becomes inefficient once there square measure isolated 1's.

These issues square measure swamped by victimization changed Radix4 Booth formula that scans strings of 3 bits victimization the formula given below:

1) Lengthen the sign bit one position if necessary to confirm that n is even.

2) Add a zero to the correct of the LSB of the multiplier factor.

3) akin to the worth of every vector, every Partial Product are zero, +M, -M, +2M or -2M. The negative values of M square measure created by taking its 2's complement. The multiplication of M is completed by shifting M by one bit to the left (in case it's increased with 2). Thus, in any case, in coming up with associate degree n-bit parallel multiplier factor, solely n/2 partial merchandise square measure generated.

TABLE 3.1: Modified Booth's Recoding Table

i+1	*I*	*i-1*	*add*
0	0	0	0*M
0	0	1	1*M
0	1	0	1*M
0	1	1	2*M
1	0	0	-2*M
1	0	1	-1*M
1	1	0	-1*M
1	1	1	0*M

CHAPTER 4

IMPLEMENTATION AND RESULTS

- **PROGRAMS FOR MULTIPLIERS**

- **OUTPUT WAVEFORMS**

4.1 PROGRAM FOR RADIX-4 MULTIPLIER

MAIN CODE FOR BOOTH MULTIPLIER

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity R4MUL_RCA is
Port (a, b : in STD_LOGIC_VECTOR (31 downto 0);
mul: inout std_logic_vector(63 downto 0);
overflow: out std_logic);
end R4MUL_RCA;

architecture Behavioral of R4MUL_RCA is
component RADIX4_ENCODER is
Port ( x : in STD_LOGIC_VECTOR (31 downto 0);
arg : in STD_LOGIC_VECTOR (2 downto 0);
pp : inout STD_LOGIC_VECTOR (63 downto 0));
end component;
component fulladder
Port (a, b, cin: in STD_LOGIC;
sum, cout: out STD_LOGIC);
end component;
```

```
component RCA64  is
    Port ( a,b : in  STD_LOGIC_VECTOR (63 downto 0);

           add : out  STD_LOGIC_VECTOR (63 downto 0);
                                   cout: out std_logic);
end component ;

signal arg1, arg2, arg3, arg4: std_logic_vector(2 downto 0);
signal arg5, arg6, arg7, arg8: std_logic_vector(2 downto 0);
signal arg9, arg10, arg11, arg12: std_logic_vector(2 downto 0);
signal arg13, arg14, arg15, arg16: std_logic_vector(2 downto 0);
signal tt: std_logic_vector(32 downto 0);
signal s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13,s14,s15: std_logic_vector(63 downto 0);
signal sum1,sum2,sum3,sum4,sum5,sum6,sum7,sum8: std_logic_vector(63 downto 0);
signal sum9,sum10,sum11,sum12,sum13,sum14,sum15: std_logic_vector(63 downto 0);
signal y: std_logic_vector(15 downto 0);
signal pp1, pp2, pp3, pp4, pp5, pp6, pp7, pp8 : STD_LOGIC_VECTOR (63 downto 0);
signal pp9, pp10, pp11, pp12, pp13, pp14, pp15, pp16: STD_LOGIC_VECTOR (63 downto
0);
begin
tt<= a(31 downto 0)&'0';
arg1<=tt(2 downto 0);
arg2<=tt(4 downto 2);
arg3<=tt(6 downto 4);
arg4<=tt(8 downto 6);
```

```
arg5<=tt(10 downto 8);
arg6<=tt(12 downto 10);
arg7<=tt(14 downto 12);
arg8<=tt(16 downto 14);
arg9<=tt(18 downto 16);
arg10<=tt(20 downto 18);
arg11<=tt(22 downto 20);
arg12<=tt(24 downto 22);
arg13<=tt(26 downto 24);
arg14<=tt(28 downto 26);
arg15<=tt(30 downto 28);
arg16<=tt(32 downto 30);
u1: RADIX4_ENCODER port map(b(31 downto 0), arg1, pp1);
u2: RADIX4_ENCODER port map(b(31 downto 0), arg2, pp2);
u3: RADIX4_ENCODER port map(b(31 downto 0), arg3, pp3);
u4: RADIX4_ENCODER port map(b(31 downto 0), arg4, pp4);
u5: RADIX4_ENCODER port map(b(31 downto 0), arg5, pp5);
u6: RADIX4_ENCODER port map(b(31 downto 0), arg6, pp6);
u7: RADIX4_ENCODER port map(b(31 downto 0), arg7, pp7);
u8: RADIX4_ENCODER port map(b(31 downto 0), arg8, pp8);
u9: RADIX4_ENCODER port map(b(31 downto 0), arg9, pp9);
u10: RADIX4_ENCODER port map(b(31 downto 0), arg10, pp10);
u11: RADIX4_ENCODER port map(b(31 downto 0), arg11, pp11);
u12: RADIX4_ENCODER port map(b(31 downto 0), arg12, pp12);
u13: RADIX4_ENCODER port map(b(31 downto 0), arg13, pp13);
```

```
u14: RADIX4_ENCODER port map(b(31 downto 0), arg14, pp14);

u15: RADIX4_ENCODER port map(b(31 downto 0), arg15, pp15);

u16: RADIX4_ENCODER port map(b(31 downto 0), arg16, pp16);

s1<= pp2(61 downto 0)&"00";

s2<= pp3(59 downto 0)&"0000";

s3<= pp4(57 downto 0)&"000000";

s4<= pp5(55 downto 0)&"00000000";

s5<= pp6(53 downto 0)&"0000000000";

s6<= pp7(51 downto 0)&"000000000000";

s7<= pp8(49 downto 0)&"00000000000000";

s8<= pp9(47 downto 0)&"0000000000000000";

s9<= pp10(45 downto 0)&"000000000000000000";

s10<= pp11(43 downto 0)&"00000000000000000000";

s11<= pp12(41 downto 0)&"0000000000000000000000";

s12<= pp13(39 downto 0)&"000000000000000000000000";

s13<= pp14(37 downto 0)&"00000000000000000000000000";

s14<= pp15(35 downto 0)&"0000000000000000000000000000";

s15<= pp16(33 downto 0)&"000000000000000000000000000000";


h1: RCA64 port map(pp1, s1, sum1, y(0));

h2: RCA64 port map(sum1, s2, sum2, y(1));

h3: RCA64 port map(sum2, s3, sum3, y(2));

h4: RCA64 port map(sum3, s4, sum4, y(3));

h5: RCA64 port map(sum4, s5, sum5, y(4));

h6: RCA64 port map(sum5, s6, sum6, y(5));
```

```
h7: RCA64 port map(sum6, s7, sum7, y(6));

h8: RCA64 port map(sum7, s8, sum8, y(7));

h9: RCA64 port map(sum8, s9, sum9, y(8));

h10: RCA64 port map(sum9, s10, sum10, y(9));

h11: RCA64 port map(sum10, s11, sum11, y(10));

h12: RCA64 port map(sum11, s12, sum12, y(11));

h13: RCA64 port map(sum12, s13, sum13, y(12));

h14: RCA64 port map(sum13, s14, sum14, y(13));

h15: RCA64 port map(sum14, s15, mul, overflow);

end Behavioral;
```

CODE FOR RIPPLE CARRY ADDER

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

use IEEE.STD_LOGIC_ARITH.ALL;

use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity RCA64  is

   Port ( a,b : in  STD_LOGIC_VECTOR (63 downto 0);

                         cout: out std_logic;

          add : out  STD_LOGIC_VECTOR (63 downto 0));

end RCA64 ;

architecture Behavioral of RCA64  is

begin

process(a,b)
```

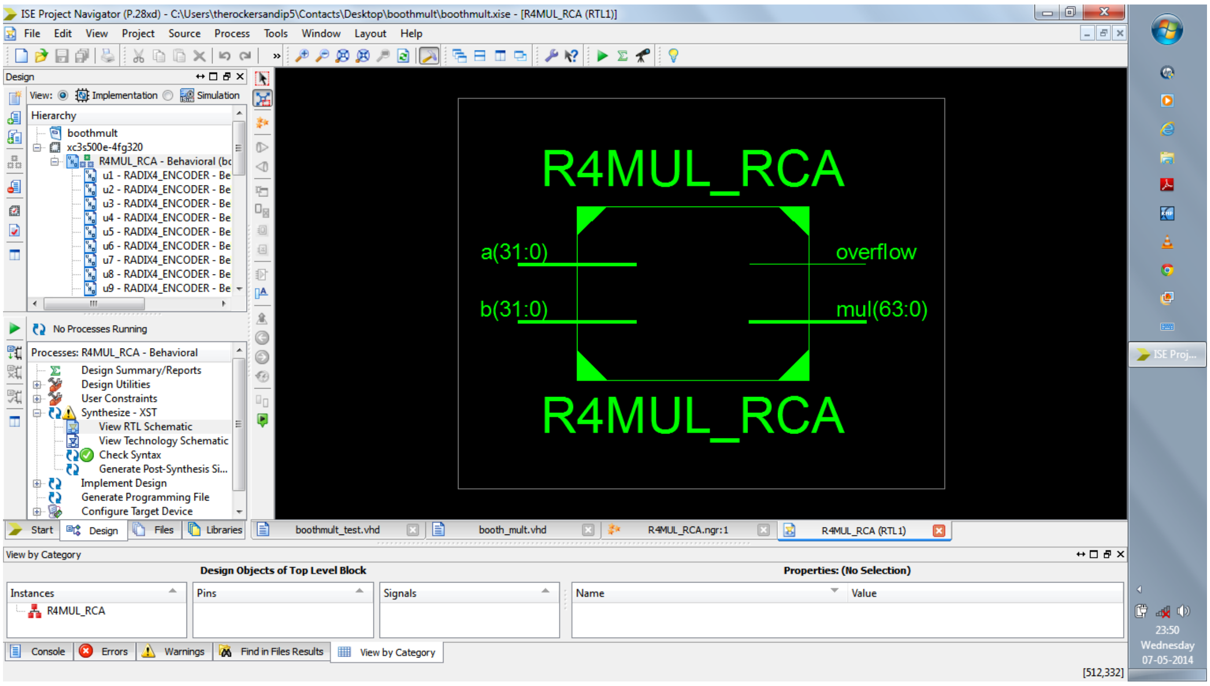
```
variable x :std_logic_vector(64 downto 0);
begin
x(0):='0';
for i in 0 to 63 loop
add(i)<=a(i) xor b(i) xor x(i);
x(i+1):= (a(i)and b(i)) or (a(i) and x(i)) or  (x( i) and b(i));
end loop;
cout<=x(64);
end process;
end behavioral;
```

BOOTH LOGIC CODE

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity RADIX4_ENCODER is
generic(N: integer:= 32);
Port ( x : in STD_LOGIC_VECTOR (N-1 downto 0);
arg : in STD_LOGIC_VECTOR (2 downto 0);
pp : inout STD_LOGIC_VECTOR (2*N-1 downto 0));
end RADIX4_ENCODER;
```

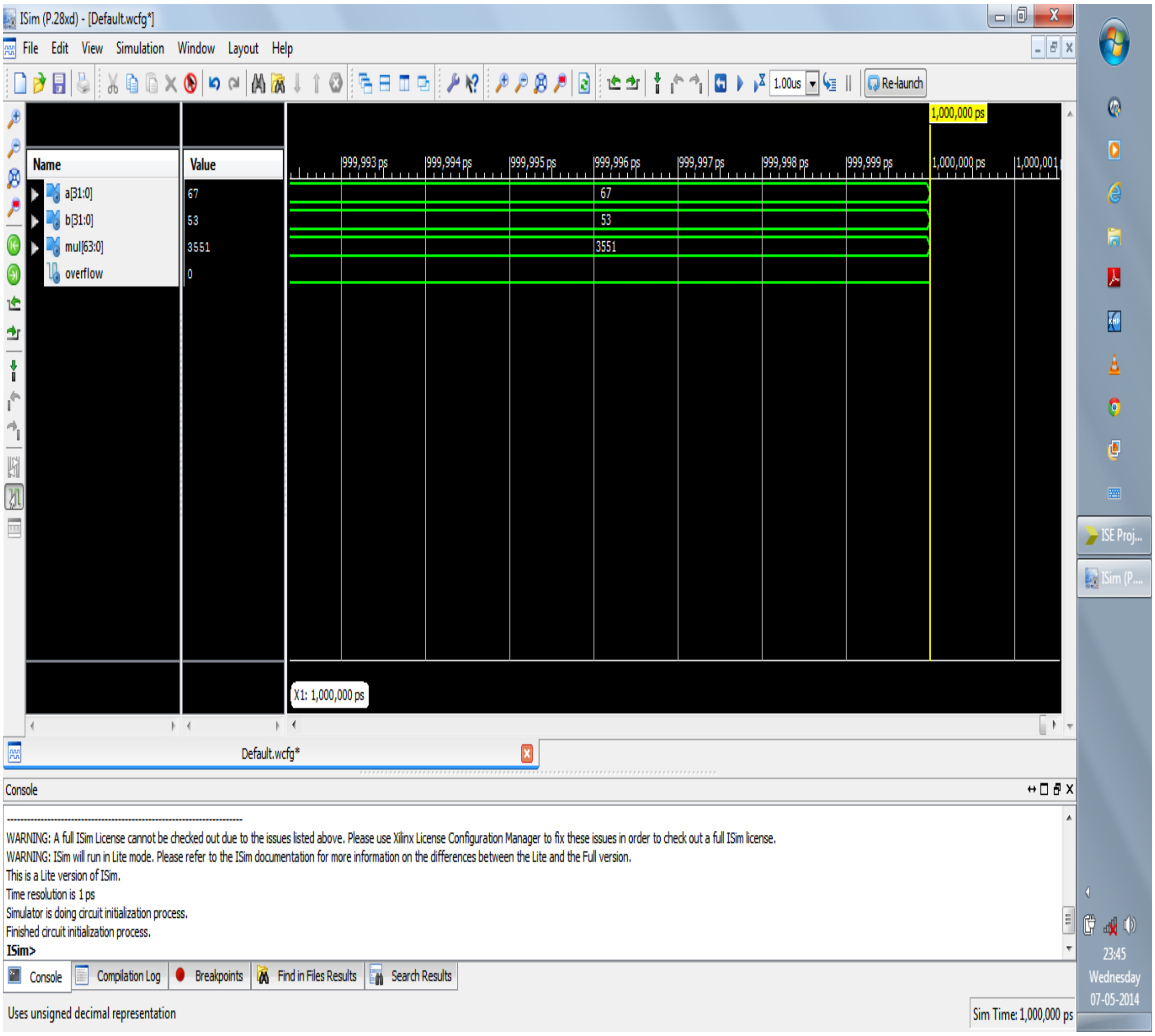
```
architecture Behavioral of RADIX4_ENCODER is

begin

process(arg, x)

variable temp, temp1, temp2: std_logic_vector(N downto 0);

begin

if x(N-1)='1' then

temp:= '1'&x(N-1 downto 0);

else

temp:= '0'&x(N-1 downto 0);

end if;

if(arg="001"or arg="010") then

temp1:= temp;

elsif(arg="101" or arg="110") then

temp1:= not(temp) + "000000001";

elsif(arg="011") then

temp1:= temp(N-1 downto 0)&'0';

elsif(arg="100") then

temp2:= not(temp) + "000000001";

temp1:= temp2(N-1 downto 0)&'0';

else

temp1:= (others=>'0');

end if;

pp<= sxt(temp1, 2*N);

end process;

end Behavioral;
```

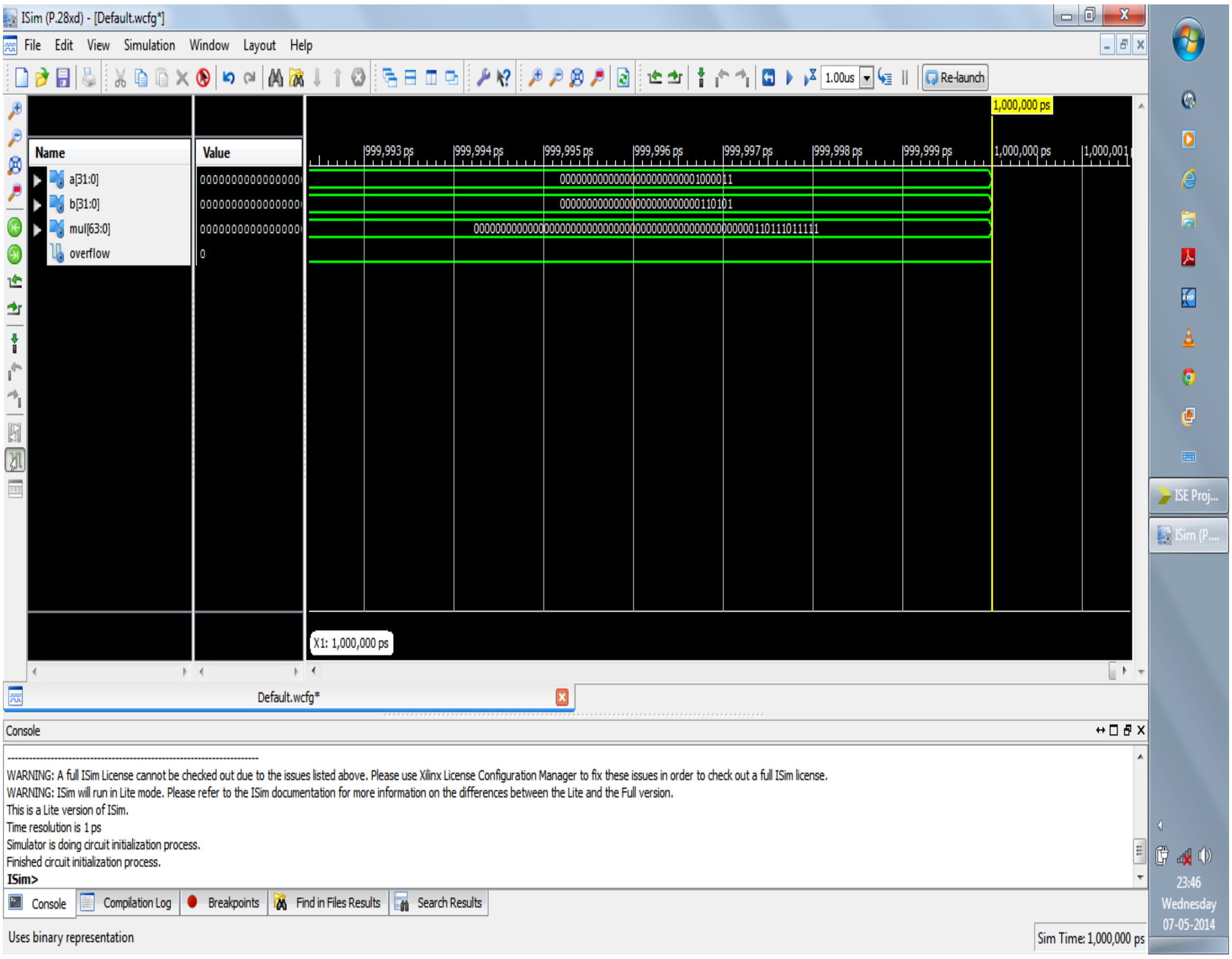
RTL SCHEMATIC

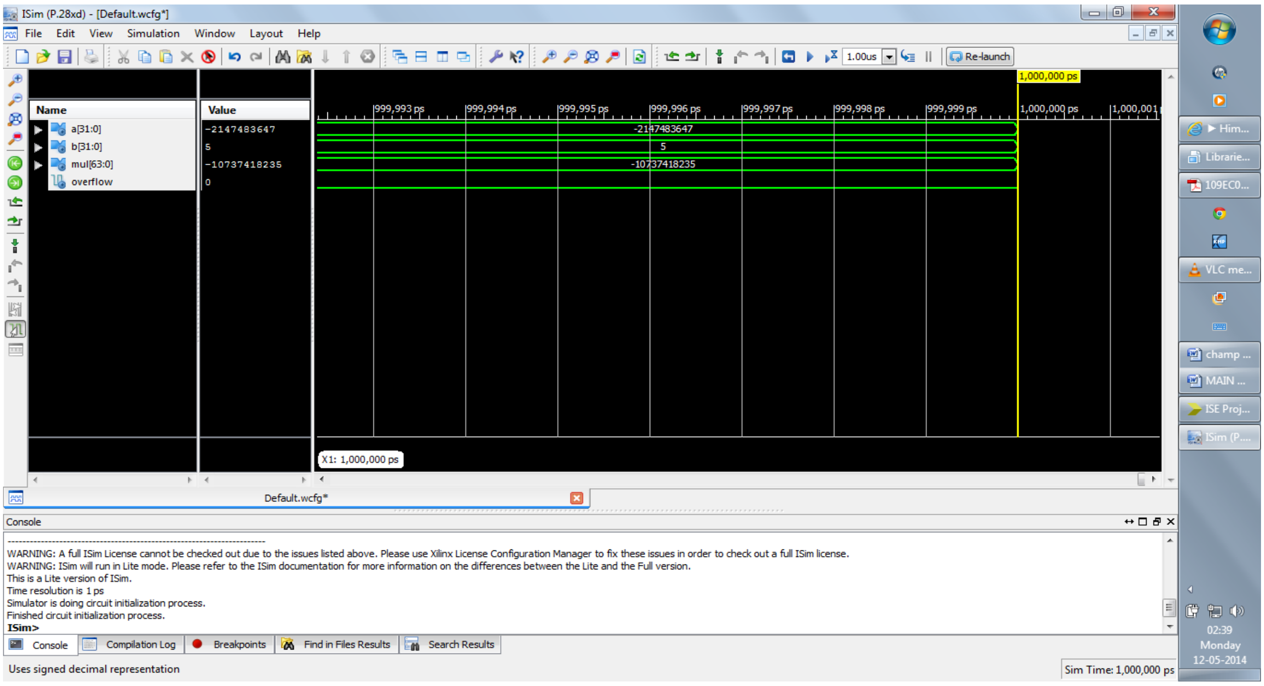


4.2 OUTPUT WAVEFORMS

Testbench Waveform generation using XIlinx




ISim (P.28xd) - [Default.wcfg*]
File Edit View Simulation Window Layout Help
Name
Value
a[31:0]
b[31:0]
mul[63:0]
overflow
1,000,000 ps
X1: 1,000,000 ps
Default.wcfg*
Console
WARNING: A full ISim License cannot be checked out due to the issues listed above. Please use Xilinx License Configuration Manager to fix these issues in order to check out a full ISim license.
WARNING: ISim will run in Lite mode. Please refer to the ISim documentation for more information on the differences between the Lite and the Full version.
This is a Lite version of ISim.
Time resolution is 1 ps
Simulator is doing circuit initialization process.
Finished circuit initialization process.
ISim>
Console
Compilation Log
Breakpoints
Find in Files Results
Search Results
Uses binary representation
Sim Time: 1,000,000 ps


ISim (P.28xd) - [Default.wcfg*]
File Edit View Simulation Window Layout Help
Re-launch
1,000,000 ps
Name
Value
a[31:0]
b[31:0]
mul[63:0]
overflow
-2147483647
5
-10737418235
0
999,993 ps
999,994 ps
999,995 ps
999,996 ps
999,997 ps
999,998 ps
999,999 ps
1,000,000 ps
X1: 1,000,000 ps
Default.wcfg*
Console
WARNING: A full ISim License cannot be checked out due to the issues listed above. Please use Xilinx License Configuration Manager to fix these issues in order to check out a full ISim license.
WARNING: ISim will run in Lite mode. Please refer to the ISim documentation for more information on the differences between the Lite and the Full version.
This is a Lite version of ISim.
Time resolution is 1 ps
Simulator is doing circuit initialization process.
Finished circuit initialization process.
ISim>
Console
Compilation Log
Breakpoints
Find in Files Results
Search Results
Uses signed decimal representation
Sim Time: 1,000,000 ps
02:39
Monday
12-05-2014

CHAPTER 6

CONCLUSION & FUTURE WORK
REFERENCES

CONCLUSION AND FUTURE WORK

After browsing all the toil and when facing plenty of issues, we have a tendency to managed to complete the objectives of the project that square measure to implement Booth's formula for the look of a binary multiplier factor mistreatment ripple carry adder design. in any case we have a tendency to came to a conclusion that Ripple Carry Adders square measure best suited to our Applications. Then we have a tendency to turned our focus into the look of Multipliers. initial of all we have a tendency to designed a Booth's Radix-4 multiplier factor. If we have a tendency to comparison information between Radix-2 and Radix-4 booth multipliers we have a tendency to noted that radix-4 consumes less power than radix-2, as a result of radix-4 uses virtually a 0.5 variety of iterations than radix-2. As radix-4 appeared a lot of appropriate for the look we have a tendency to dispensed additional analysis on radix-4 multiplier factor by mistreatment ripple carry adder design.

Further work will be dispensed on this project within the power estimation section. Power will be calculable at the gate-level by generating gate-level netlist and conjointly the post layout analysis will be in serious trouble this style. Another attainable direction will be pursued for higher base encryption.

REFERENCES

[1] ***A. D. Booth*** *"A signed binary multiplication technique," Quart. J. Mech. Appl. Math., vol.4*

[2] *"Design, Analysis and Switching Activity based Power Estimation of Booth Multiplier using Different adder techniques" by* ***Arun Kumar P.S, J K Das, Sudeendra Kumar, K K Mahapatra****.*

[3] *"Controllability-driven power virus generation for digital circuits" by K.Najeeb, Karthik Gururaj, V.Kamakoti.*

[4] *Power Compiler Manuals, www.synopsys.com*