UC Santa Cruz
UC Santa Cruz Electronic Theses and Dissertations

Title
OpenWare Library

Permalink
https://escholarship.org/uc/item/3dx5775c

Author
Mantri, Garvit Rajendra

Publication Date
2018

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
SANTA CRUZ

OPENWARE LIBRARY

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Garvit Rajendra Mantri

June 2018

The Thesis of Garvit Rajendra Mantri
is approved:

Professor Jose Renau, Chair

Professor Anujan Varma

Professor Heiner Litz

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Copyright © by

Garvit Rajendra Mantri

2018

Table of Contents

List of Figures **iv**

List of Tables **v**

Abstract **vi**

Acknowledgments **vii**

1 Introduction **1**

2 Binary Adders **3**

3 Ripple Carry Adder **7**

4 Carry Save Adder **11**

5 Parallel Prefix Adders **14**

6 Bit Shifts **21**

7 Booth Multiplier **31**

8 Results **34**

9 Conclusion **36**

Bibliography **37**

List of Figures

2.1 Combinational Block Diagram 3
2.2 Implementation of Half Adder 4
2.3 Implementation of Full Adder 5

3.1 4-bit Ripple Carry Adder 7

4.1 4-bit Carry Save Adder 12

5.1 Parallel Prefix Block Diagram 15
5.2 16-bit Kogge Stone Prefix Graph 16
5.3 16-bit Han Carlson Prefix Graph 17

6.1 Example: Shift Logical/Arithmetic Left 22
6.2 Example: Shift Logical Right 25
6.3 Example: Shift Rotate Left 27
6.4 Example: Shift Rotate Right 29

7.1 Example: Multiplication 32

List of Tables

2.1 Truth Table: Half Adder 4
2.2 Truth Table: Full Adder 5
2.3 ABC Results: Full Adder 6

3.1 Statistics: Ripple Carry Adder 10

4.1 Statistics: Carry Save Adder 13

5.1 ABC Results: Carry Save Adder 18
5.2 Statistics: Kogge Stone Adder 19
5.3 Statistics: Han Carlson Adder 20

6.1 Statistics: Bit Shift Left 24
6.2 Statistics: Bit Shift Arithmetic Right 26
6.3 Statistics: Bit Shift Rotate Left 28
6.4 Statistics: Bit Shift Rotate Right 30

7.1 Radix-4 Booth Encoding Values 32
7.2 Statistics: RADIX-4 Booth Multiplier 33

8.1 Results:Adders 34
8.2 Results 35

Abstract

OpenWare Library

by

Garvit Rajendra Mantri

Encapsulating various implementation of basic modules has always been a key to generic programming. It has helped developing complex designs faster, efficiently and more robust.

There can be instances where the module required for your design might already have been a necessity to another person earlier. In such a case, there are high chances of an already existing bug free implementation. Hence, you need not reinvent the wheel. Most of the synthesis tools in industry have some sort of library that helps an engineer to develop the bigger or complex designs by instantiating basic modules.

OpenWare addresses this by incorporating various modules so that an engineer's time is saved by not creating the redundant modules repeatedly. The library also has other advantages such as having a common legacy so that the whole organization goes with basic standard and have a bug free implementation. It is always reliable to use something which is tested extensively, and its working is established by a group of people.

Acknowledgments

I, Garvit Mantri take this opportunity to acknowledge the people without whom these journey would not have been possible.

I am extremely thankful to have Prof. Jose Renau as my advisor who helped me a lot from coursework to granting me permission to work in his lab. I am thankful to him to let me work on this project and ESESC. He has helped me recover from my mistakes and learn from them. I have come out to be more hardworking while watching him work on his research.

I would like to express my gratitude to the readers of the thesis, Professor Heiner Litz and Professor Anujan Varma for their extremely valuable suggestions. I would like to appreciate Rafael Trapani for helping me setting up the system.

I would like to specially say thanks to my friend, Isaak Cherdak for helping me throughout the thesis by providing valuable suggestions towards testbench.

I feel grateful and lucky to have full support from my family and housemates who have helped me in taking rational decisions throughout the Masters' journey.

Chapter 1

Introduction

OpenWare Library is an attempt to create a library where we have designed basic modules which can come quite handy to instantiate while designing complex modules and bigger designs.

All the modules designed are configurable in terms of the number of bits. We have designed the modules using verilog. A significant amount of time is spent to make sure that the modules are purely combinational and can be synthesized.

OpenWare helps in removing the redundancy to create the basic modules repeatedly. Modules such as Adders, Parallel Prefix Adder, Shifters, and Multipliers are designed which can be configured theoretically to any number of bits. Complete synthesis results of each module that includes the area, critical path, critical path delay, input capacitance and output capacitance are published.Each module is tested via a C++ test-bench.

Yosys is the framework used for verilog RTL synthesis. Yosys is free software licensed under the ISC license (a GPL compatible license that is similar in terms to the MIT license, or the 2-clause BSD license). We have a version 0.7+ for synthesis results. A 15 nm OCL standard library is used for the cells.

We have currently included Adders, Shifter and Multiplier to our library which

provides a standard framework for future designs. The library specifically focuses on modules which are synthesizable (realizable) using common logic gates. We have also included synthesis results so that a designer can select the module based on the requirement. The added advantage of this library is that the modules created have a re-configurable instantiation in terms of the number of bits.

The following chapters will go over the individual modules such as binary adders, ripple carry adder, carry save adder, parallel prefix adder which includes two types i.e kogge-stone and han-carlson adder. Further the shifters and multiplier are explained briefly. Each chapter explains the basics regarding the circuit and how it operates. Basic building blocks used to designed those modules are also included along with the synthesis results.

OpenWare tries to create a legacy for the complete organization where one can go with a default standard for modules used repeatedly with no or known bugs.

Chapter 2

Binary Adders

A combinational circuit consists of various inputs, logic gates and outputs variables. Combinational logic gates respond to the values at their inputs and produce the value of the output signal, transforming binary information from the given input data to a required output data.[5] For n input variables, there are 2^n possible binary input combinations. For each possible input combination, there is one possible output value. Thus a combinational circuit can be specified with a truth table that lists the output values for each combination of input variables.

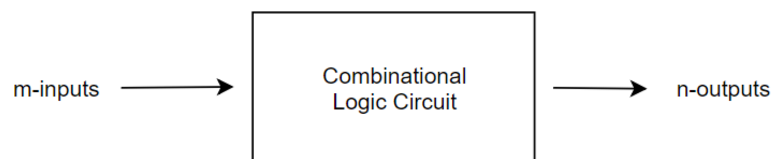

Figure 2.1: Combinational Block Diagram

There are several combinational circuits that are used as a basic building blocks to bigger designs. In this chapter, we will brief over two of them i.e Half Adder and Full Adder. These are the building blocks to modules explained in later chapters. Also, All the modules designed in this library are purely combinational. A block diagram of a combinational circuit is shown in Fig. 2.1.

A half adder have two binary inputs and two binary outputs. It can be implemented with an exclusive-OR and an AND gate. The Boolean logic for sum(S) will be A'B+AB' and for Carry(C) will be AB. Implementation of half adder is shown in Fig. 2.2. Truth table for a half adder is shown in Table 2.1. The simplified expression are:

$S = A \oplus B$

$C = AB$

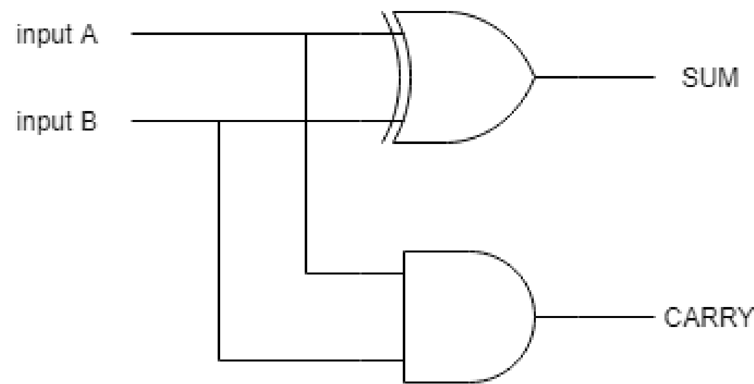

Figure 2.2: Implementation of Half Adder

Truth Table: Half Adder			
input A	input B	SUM	CARRY
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 2.1: Truth Table: Half Adder

A full adder is a combinational circuit that adds three bits. In other words, it incorporates the carry bit. It consists of three inputs and two outputs. It can be implemented using two half adders and an OR gate. The simplified expression are:

SUM = A ⊕ (B ⊕ C)

SUM = A'B'C + A'BC' + AB'C' + ABC

CARRY = AB + AC + BC

A full adder is a basic building block for cascade and parallel prefix adders which adds 8,16,32, etc bit binary numbers.Implementation of full adder is shown in Fig. 2.3. Truth table for a full adder is shown in Table. 2.2.

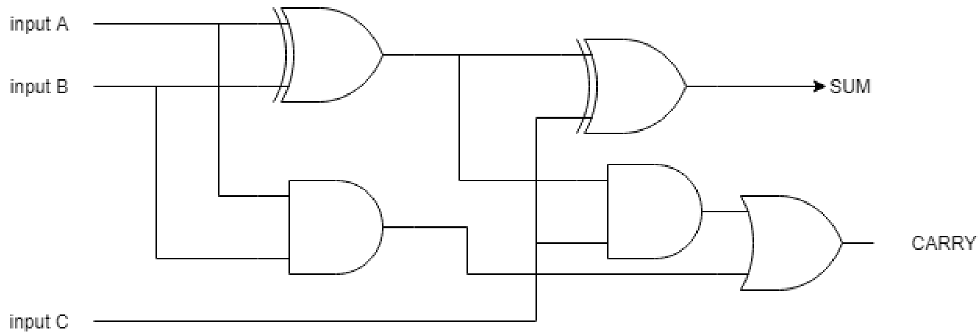

Figure 2.3: Implementation of Full Adder

Truth Table: Full Adder				
input A	input B	input C	SUM	CARRY
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

Table 2.2: Truth Table: Full Adder

ABC Results: Full Adder	
Delay (ps)	11.11
NAND2_X1 cells:	3
XOR2_X1 cells:	2
internal signals:	3
input signals:	3
output signals:	2

Table 2.3: ABC Results: Full Adder

ABC Results I.e logic synthesis and timing analysis results are shown below. The results clearly shows the Gates which is the number of gates in the circuit. For example, the full adder contains 5 gates. Delay of the critical path is 11.1 picoseconds. A refers to the area factor. Df refers to the delay factor. Cin and Cout are input capacitance and output capacitance respectively. The results show multiple paths of the circuit as path 0, path 1 and so on.

```
ABC Results: Full Adder
Gates = 5  Cap = 1.3 ff  Area = 1.47  Delay = 11.11 ps
Path 0 -- 2 pi      A = 0.00  Df =  0.0 ps Cin = 0.0 ff  Cout = 2.4 ff
Path 1 -- 2 XOR2_X1 A = 0.44  Df =  5.7 ps Cin = 1.4 ff  Cout = 2.0 ff
Path 2 -- 1 XOR2_X1 A = 0.44  Df = 11.1 ps Cin = 1.4 ff  Cout = 0.0 ff
```

Chapter 3

Ripple Carry Adder

A ripple carry adder is an adder that produces the arithmetic sum of two binary adders. Multiple(n) full adders are cascaded to built a n-bit ripple carry adder. For example, a 4-bit ripple carry adder is shown in Fig. 3.1

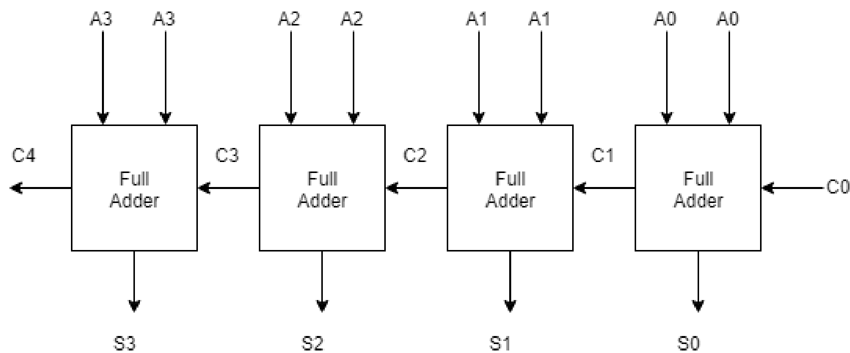

Figure 3.1: 4-bit Ripple Carry Adder

Similarly for n full adders will be cascaded for an n-bit ripple carry adder. The name is ripple carry adder is so called because carry is rippled into subsequent stages. k^{th} stage result will be valid only if the carry has propagated through the previous (k-1) stages. Therefore, S3 in the Fig. 3.1 is valid only when C3 has been generated by C2. The delay in the logical circuit from input to output i.e propagation delay is because of the carry being propagated.

As a general rule of thumb for the complete openware library, we have designed and synthesized a purely combinational circuit. Synthesis is done using Yosys Open Synthesis Suite [9]. Test-bench is developed in C++ and the logic is explained using pseudo code in Algorithm 1.

Algorithm 1: Ripple Carry Adder Testbench

```
1  Input A[63:0], Input B[63:0];
2  lower_sum = A[63] + B[63];
3  upper_A = Shift A right by 1 bit;
4  upper_B = Shift B right by 1 bit;
5  shifted_sum = upper_A + upper_B;
6  if lower_sum is 2 then
7  |   shifted_sum ++;
8  end
9  if top_carry == (shifted_sum » 63) && (A + B = Sum) then
10 |   Pass;
11 else
12 |   Fail;
13 end
```

Ripple carry adder signed testbench has to take the extra measures as illustrated in the code:

```
uint8_t sign_a = !!(top_a >> 63);
uint8_t sign_b = !!(top_b >> 63);
// evaluate correctness
printf("Test %d: ", i);
uint64_t lower_a = (top_a & ~(1 << 63));
uint64_t lower_b = (top_b & ~(1 << 63));
uint64_t unsigned_sum = lower_a + lower_b;
```

```
printf("A_sign is %d, B_sign is %d, Sum_sign is %d\n", sign_a, sign_b,
    (unsigned_sum >> 63));
int8_t our_carry = !!(sign_a == sign_b && sign_a != (unsigned_sum >> 63));
if (top_carry == our_carry &&
    top_a + top_b == top_sum) {
  printf("PASSED\n");
} else {
  printf("FAILED\n");
}
printf("With values:\n");
printf("A = %lld, B = %lld, Expected Carry = %s, Actual Carry = %s"
    ", Expected Sum = %lld, Actual Sum = %lld\n",
    top_a, top_b, our_carry ? "TRUE" : "FALSE",
    top_carry ? "TRUE" : "FALSE", (top_a + top_b), top_sum);
```

Statistics: Ripple Carry Adder				
	8-bit	16-bit	32-bit	64-bit
Delay (ps)	48.59	86.82	120.21	240.48
Number of wires	15	15	15	15
Number of wire bits	75	107	171	299
Number of public wires	07	07	07	07
Number of public wire bits	36	68	132	260
Number of memories	0	0	0	0
Number of memory bits	0	0	0	0
Number of processes	0	0	0	0
Number of cells	11	19	35	67
NOR2_X1	1	1	1	1
XNOR2_X1	1	1	1	1
XOR2_X1	1	1	1	1
Full_Adder_X1	8	16	32	64

Table 3.1: Statistics: Ripple Carry Adder

Chapter 4

Carry Save Adder

A carry save adder is so called because the carry is saved at the individual stages and latter computed in the end. In fact the result from each addition is split into two parts i.e half-sum bit and a carry bit. The half sum bits and carry bit are not combined until very end. In the end a ripple carry adder is used to take care of all the carry bits.

Carry save adders are commonly used for high speed and less delay , where they generally are able to operate faster than "ripple carry" adders because a carry save adder does not completely perform the relatively time-exhaustive process of combining carries with sum bits between successive additions in the multiplication process but instead defer it until the final cycle of the operation.

The whole motivation lies in the fact that the carry is delayed until the very end and the signals don't have to move farther. This helps in a smaller delay in comparison to ripple carry adder.

n-bit carry save adder can be implemented by using n full adders by using the following techniques:

1. Use a ripple carry adder.
2. Add 0 at the beginning (MSB) of the sum array after first stage.

3. Shift the carry array left by one bit.

We have utilized the ripple carry adder designed in the previous chapter to design carry save adder. Note, that there are three inputs to the carry save adder. This is the reason for area and delay to be in comparison with ripple carry adder. Refer to block diagram in Fig.4.1. The idea can be scaled to n-bit.

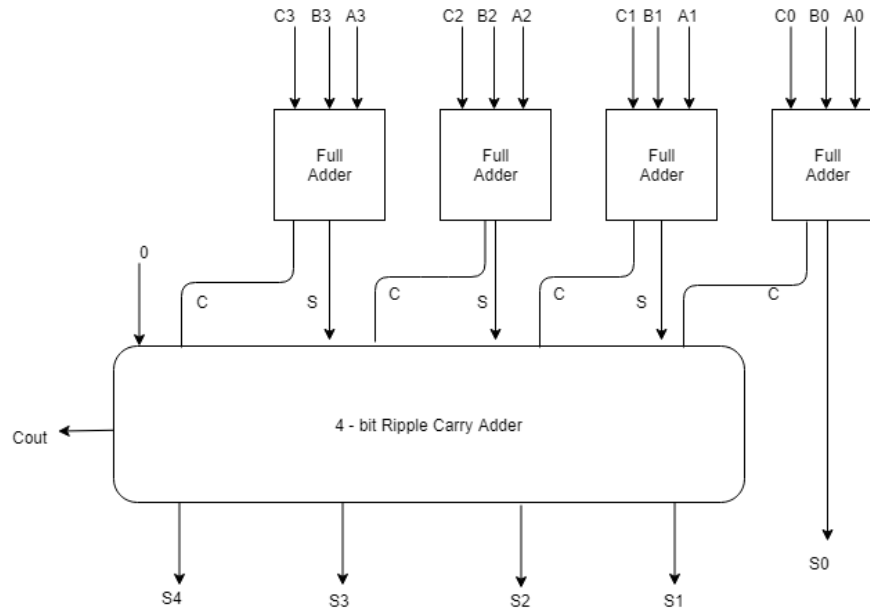

Figure 4.1: 4-bit Carry Save Adder

The test-bench used are same for all the adders in the library. Care must be taken to incorporate right data types when using more than 128 bit adders. Statistics for 8, 16, 32 and 64 bits are shown in the table below.

Statistics: Carry Save Adder				
	8-bits	16-bits	32-bits	64-bits
Delay (ps)	61.22	86.66	127.05	248.21
AND2_X1	8	12	29	61
AND3_X2	0	1	0	0
AOI21_X1	5	21	49	98
AOI21_X2	0	0	15	
INV_X1	4	37	78	155
INV_X2	0	0	2	5
NAND2_X1	24	18	50	82
NOR2_X1	18	57	119	247
NOR3_X1	0	1	6	6
OAI21_X1	0	8	39	71
XNOR2_X1	0	1	0	0
OR2_X1	0	0	3	3
XOR2_X1	22	34	58	122
Number of cells	81	190	448	880

Table 4.1: Statistics: Carry Save Adder

Chapter 5

Parallel Prefix Adders

Parallel prefix adders are most important because of the speed at which they operate. The sum of n-bit number can be computed in time O(log n)[1]. This reduction in time is achieved due to its use of a tree network known as prefix operation graph. The reduction in time helps in addition of wider word lengths. A block diagram for parallel prefix adder is shown in 5.1

Every parallel prefix adder can be designed using three stages as described in the figure 5.1. The first stage is simple half adder. The core of the parallel prefix adder is the prefix graph that propagates the carry to the final stages. An example of the graph is show in Fig. 5.2. In the prefix operation graph, each node is a basic logical circuit described as prefix operation.

The goal of addition is to compute the sum, S, of two operands A and B, both of which are binary words of length n. For n-bit addition, the first stage of the adder computes the generate (G) and propagate (P) terms for each bit of the operands according to the following equations:

Gi = Ai AND Bi

Pi = Ai XOR Bi

Stage 2 consists of the basic prefix operation, pref, is defined as follows:

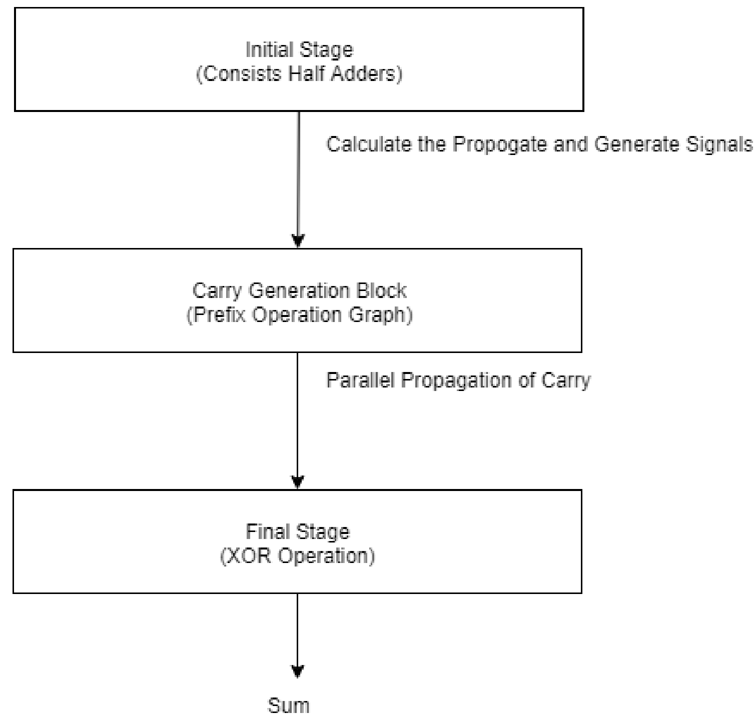

Figure 5.1: Parallel Prefix Block Diagram

(Gi,Pi) pref (Gj,Pj) = (Gi + Pi . Gj , Pi.Pj)

In the above equation, + refers to logical OR and . refers to logical AND.

In the end, the carry is equal to Gi's and sum is calculated by XOR with initial propagate which is the final stage.

We have designed two parallel prefix adders:

1. Kogge-Stone Adder
2. Han-Carlson Adder

P.M. Kogge and H.S. Stone were he first to use the property of commutativity and design parallel prefix adders where the computation of the prefixes is considered to be a recurrence that can be performed in parallel[4]. The Kogge-Stone computation uses log2n stages, where n is the number of bits in the operands.

Han-Carlson adder is a hybrid of Kogge-Stone and another parallel prefix adder I.e Brent-Kung. Kogge-Stone takes log2n stages and the Brent-Kung construction

takes 2log2n-1 stages[3]. Han-Carlson adder takes less area for the combinational circuits as compared to Kogge-Stone design.

Each prefix tree consist of the some basic building blocks such as prefix_op (Bigger Circle), Square Box, Buffer and Diamond (Last stage XOR). Prefix tree graph for 16-bit Kogge stone is shown in Fig. 5.2

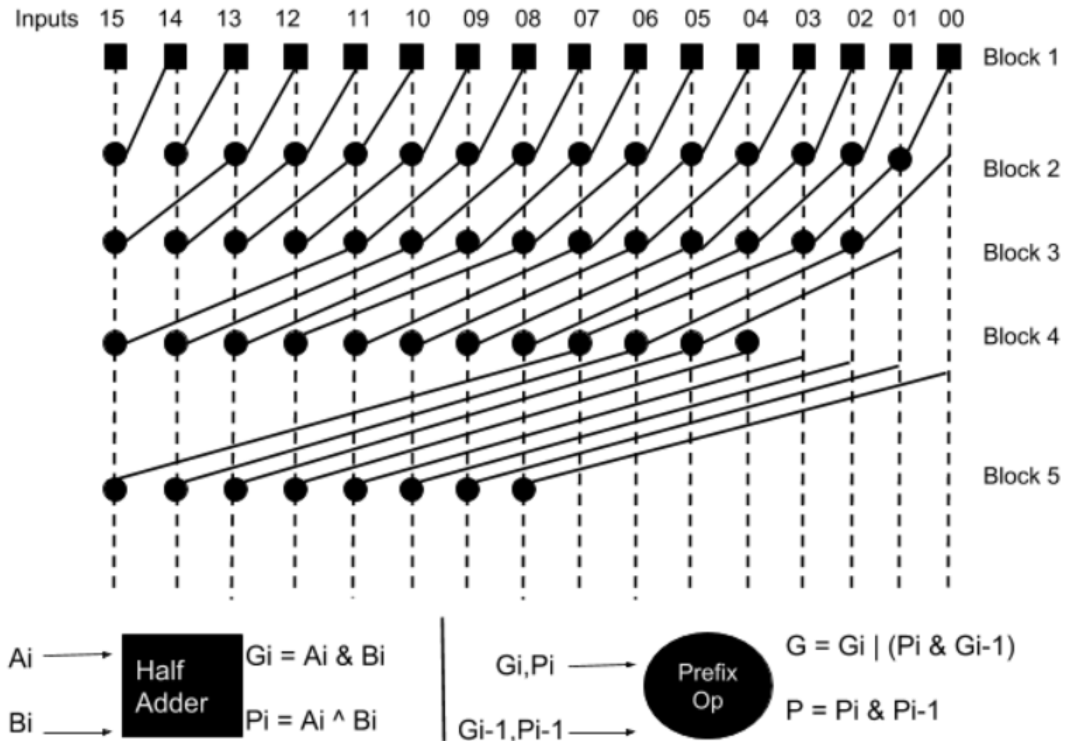

Figure 5.2: 16-bit Kogge Stone Prefix Graph

```
ABC Results: First Stage Square Operation
Gates = 3  Cap = 0.9 ff  Area = 0.69 Delay = 4.70 ps
Path 0 -- 2 pi      A = 0.00  Df = 0.0 ps  Cin = 0.0 ff  Cout = 1.6 ff
Path 1 -- 2 AND2_X1 A = 0.29  Df = 3.8 ps  Cin = 0.8 ff  Cout = 0.7 ff
Path 2 -- 1 NOR2_X1 A = 0.20  Df = 4.7 ps  Cin = 0.8 ff  Cout = 0.0 ff
```

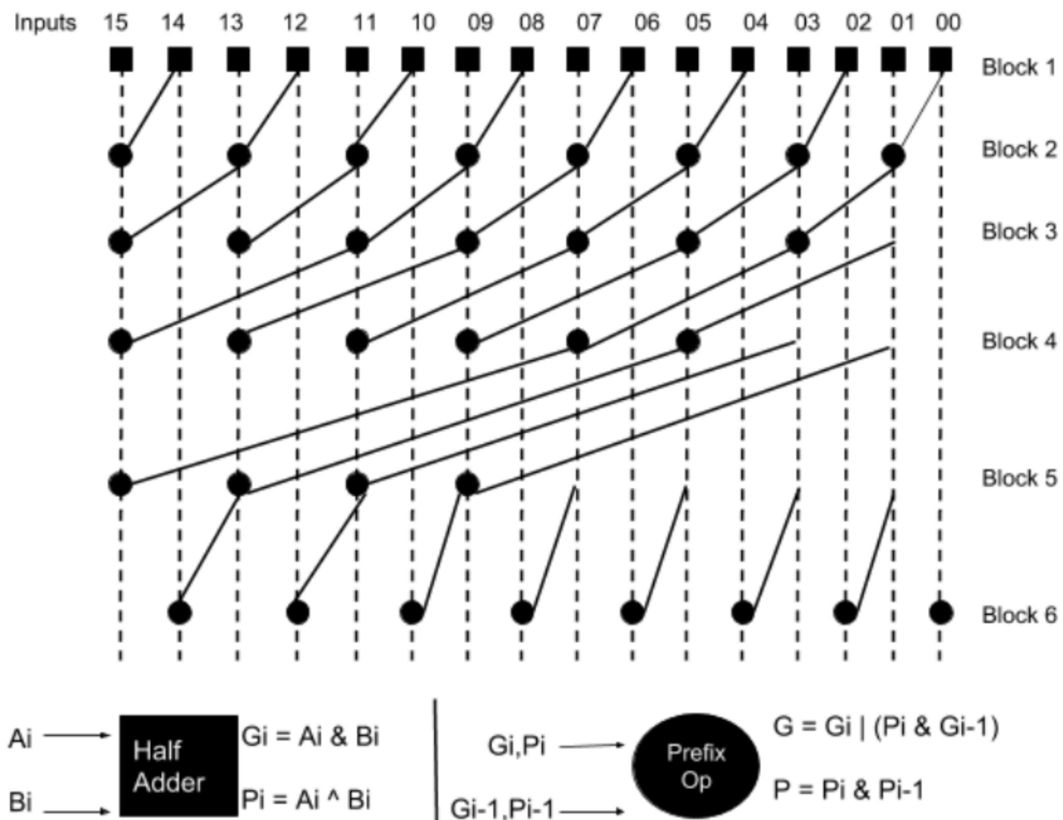

Figure 5.3: 16-bit Han Carlson Prefix Graph

```
ABC Results: Last Stage XOR Operation
Gates = 1  Cap = 1.0 ff  Area = 0.44  Delay = 4.28 ps
Path 0 -- 1 pi      A = 0.00  Df = 0.0 ps  Cin = 0.0 ff  Cout = 1.6 ff
Path 1 -- 1 XOR2_X1 A = 0.44  Df = 4.3 ps  Cin = 1.4 ff  Cout = 0.0 ff
```

```
ABC Results: Prefix Operation
Gates = 4  Cap = 0.7 ff  Area = 0.84  Delay = 3.08 ps
Path 0 -- 1 pi       A = 0.00  Df = 0.0 ps Cin = 0.0 ff  Cout = 0.8 ff
Path 1 -- 1 NAND2_X1 A = 0.20  Df = 2.0 ps Cin = 0.8 ff  Cout = 0.7 ff
Path 2 -- 1 NAND2_X1 A = 0.20  Df = 3.1 ps Cin = 0.8 ff  Cout = 0.0 ff
```

ABC Results: Building Blocks for Prefix Adder			
	Diamond	Prefix Operation	Square Operation
Delay (ps)	4.28	3.08	4.70
NAND2_X1 cells:	0	2	0
AND2_X1 cells:	0	1	1
BUF_X2 cells:	0	1	1
INV_X1 cells:	0	1	0
XOR2_X1 cells:	1	0	0
NOR2_X1 cells:	0	0	2
internal signals:	0	1	0
input signals:	2	4	2
output signals:	1	2	2

Table 5.1: ABC Results: Carry Save Adder

Statistics: Kogge Stone Adder				
	8-bit	16-bit	32-bit	64-bit
Delay (ps)	31.97	39.49	48.07	57.01
Number of wires	14	16	18	20
Number of wire bits	98	226	514	1154
Number of public wires	14	16	18	20
Number of public wire bits	98	226	514	1154
Number of memories	0	0	0	0
Number of memory bits	0	0	0	0
Number of processes	0	0	0	0
Number of cells	37	92	219	506
Buffer	7	15	31	63
Diamond	8	16	32	64
Pref_Operation	14	45	124	315
Square_Operation	8	16	32	64

Table 5.2: Statistics: Kogge Stone Adder

Statistics: Han Carlson Adder		
	16-bit	32-bit
Delay (ps)	77.28	62.67
Number of wires	18	20
Number of wire bits	258	578
Number of public wires	18	20
Number of public wire bits	258	578
Number of memories	0	0
Number of memory bits	0	0
Number of processes	0	0
Number of cells	107	250
Buffer	47	111
Diamond	16	32
Pref_Operation	28	75
Square_Operation	16	32

Table 5.3: Statistics: Han Carlson Adder

Chapter 6

Bit Shifts

Bitwise Operation operates on a binary number in a manner that each individual bits can be manipulated. We have designed some of the basic shifts that come in very handy. The shifts simple move the number of bits in the left or right direction and fill the empty spots as per required. In other words, the shifters will shift the data word by specified number of bits purely by a combinational logic.

Four basic modules designed are:

1. Barrel Shift left:

A barrel shift left moves the bits to its left by the specified number of bits and append zeroes at the right. This is exactly same as shift arithmetic left. A pseudo-code for implementing shift logical/arithmetic left is shown below. Refer to Fig.6.1.

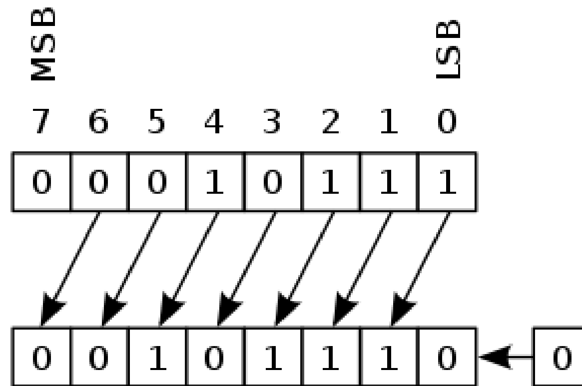

Figure 6.1: Example: Shift Logical/Arithmetic Left

Algorithm 2: Shift Logical/Arithmetic Left

```
1  Input a[Bits-1:0];
2  Input shift_bits;
3  Input b[Bits-1:0];
4  //Adding 0's to the top:
5  temp = 0;
6  reg [(2*Bits-1):0] a_double;
7  a_double = {a,temp};
8  for(i=0; i<Bits; i++)
9  array[i] = a_double[2*Bits-1-1:Bits-i];
10 b = array[shift_bits];
```

```
ABC Results Bit Shift Left (64-bits):
Gates = 4739  Cap = 1.5 ff  Area = 940.77  Delay = 67.98 ps
Path  0 --  3 pi       A = 0.00  Df = 1.2  ps  Cin = 0.0 ff  Cout =  2.5 ff
Path  1 --  2 INV_X1   A = 0.15  Df = 3.3  ps  Cin = 0.8 ff  Cout =  1.6 ff
Path  2 --  1 NOR2_X1  A = 0.20  Df = 5.9  ps  Cin = 0.8 ff  Cout =  0.8 ff
Path  3 --  7 INV_X1   A = 0.15  Df = 11.0 ps  Cin = 0.8 ff  Cout =  5.8 ff
Path  4 -- 10 BUF_X2   A = 0.25  Df = 18.5 ps  Cin = 0.8 ff  Cout =  8.3 ff
Path  5 -- 10 NOR2_X1  A = 0.20  Df = 29.4 ps  Cin = 0.8 ff  Cout =  7.3 ff
```

```
Path  6 -- 10 INV_X1    A = 0.15  Df = 39.0 ps  Cin = 0.8 ff  Cout =  7.3 ff
Path  7 -- 10 CLKBUF_X2 A = 0.25  Df = 47.3 ps  Cin = 0.8 ff  Cout =  7.3 ff
Path  8 --  1 NOR2_X1   A = 0.20  Df = 50.5 ps  Cin = 0.8 ff  Cout =  0.8 ff
Path  9 --  1 NOR2_X1   A = 0.20  Df = 52.7 ps  Cin = 0.8 ff  Cout =  0.7 ff
Path 10 --  1 NAND2_X1  A = 0.20  Df = 54.4 ps  Cin = 0.8 ff  Cout =  0.7 ff
Path 11 --  1 NOR2_X1   A = 0.20  Df = 56.0 ps  Cin = 0.8 ff  Cout =  0.7 ff
Path 12 --  1 NAND2_X1  A = 0.20  Df = 57.6 ps  Cin = 0.8 ff  Cout =  0.7 ff
Path 13 --  1 NOR2_X1   A = 0.20  Df = 59.3 ps  Cin = 0.8 ff  Cout =  0.8 ff
Path 14 --  1 NAND2_X1  A = 0.20  Df = 68.0 ps  Cin = 0.8 ff  Cout = 10.0 ff
```

Statistics: Bit Shift Left				
	8-bits	16-bits	32-bits	64-bits
Delay (ps)	32.46	45.39	58.38	67.98
Number of wires	161	459	1587	5425
Number of wire bits	321	1142	4423	17122
Number of public wires	12	20	36	68
Number of public wire bits	99	324	1157	4358
Number of memories	0	0	0	0
Number of memory bits	0	0	0	0
Number of processes	0	0	0	0
Number of cells	89	310	1259	4739
AND2_X1	2	1	5	2
BUF_X1	0	0	1	1
BUF_X2	0	0	26	70
CLKBUF_X1	0	0	0	1
CLKBUF_X2	0	0	45	263
INV_X1	13	38	80	155
NAND2_X1	47	110	612	1780
NOR2_X1	27	161	490	2467

Table 6.1: Statistics: Bit Shift Left

2. Barrel shift right:

A barrel shift right moves the bits to its right by the specified number of bits. In the case of arithmetic shift right, MSB is appended to the left. In the case of logical shift right, zeroes are appended to the MSB. Refer to Fig.6.2.

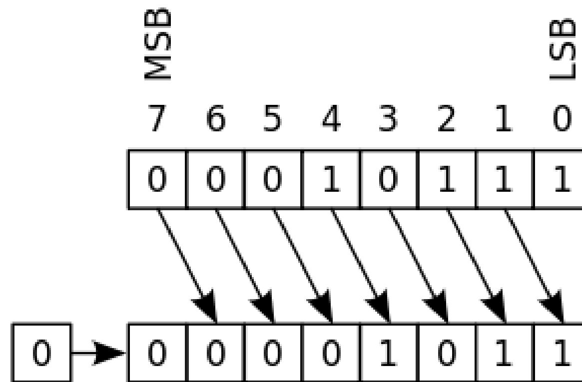

Figure 6.2: Example: Shift Logical Right

```
ABC Results Bit Shift Right (64 bits):
Gates = 4935  Cap = 1.5 ff  Area = 987.41  Delay = 72.56 ps
Path  0 --  3 pi        A = 0.00  Df = 1.2  ps  Cin = 0.0 ff  Cout = 2.5 ff
Path  1 --  2 INV_X1    A = 0.15  Df = 3.3  ps  Cin = 0.8 ff  Cout = 1.6 ff
Path  2 --  1 NOR2_X1   A = 0.20  Df = 5.9  ps  Cin = 0.8 ff  Cout = 0.8 ff
Path  3 --  7 INV_X1    A = 0.15  Df = 11.0 ps  Cin = 0.8 ff  Cout = 5.8 ff
Path  4 -- 10 BUF_X2    A = 0.25  Df = 18.5 ps  Cin = 0.8 ff  Cout = 8.3 ff
Path  5 -- 10 NOR2_X1   A = 0.20  Df = 29.9 ps  Cin = 0.8 ff  Cout = 7.7 ff
Path  6 --  2 NOR2_X1   A = 0.20  Df = 34.7 ps  Cin = 0.8 ff  Cout = 1.5 ff
Path  7 --  1 NAND2_X1  A = 0.20  Df = 37.8 ps  Cin = 0.8 ff  Cout = 0.8 ff
Path  8 --  1 NOR2_X1   A = 0.20  Df = 40.0 ps  Cin = 0.8 ff  Cout = 0.8 ff
Path  9 --  2 NAND2_X1  A = 0.20  Df = 42.5 ps  Cin = 0.8 ff  Cout = 1.4 ff
Path 10 --  1 NOR2_X1   A = 0.20  Df = 44.4 ps  Cin = 0.8 ff  Cout = 0.8 ff
Path 11 --  1 NAND2_X1  A = 0.20  Df = 46.6 ps  Cin = 0.8 ff  Cout = 0.9 ff
Path 12 --  9 BUF_X2    A = 0.25  Df = 52.8 ps  Cin = 0.8 ff  Cout = 7.6 ff
Path 13 -- 10 BUF_X2    A = 0.25  Df = 60.0 ps  Cin = 0.8 ff  Cout = 8.3 ff
Path 14 --  1 NAND2_X1  A = 0.20  Df = 72.6 ps  Cin = 0.8 ff  Cout =10.0 ff
```

Statistics: Bit Shift Arithmetic Right				
	8-bits	16-bits	32-bits	64-bits
Delay (ps)	33.75	50.90	59.06	72.56
Number of wires	166	519	1663	5679
Number of wire bits	338	1239	4601	17639
Number of public wires	13	21	37	69
Number of public wire bits	107	340	1189	4422
Number of memories	0	0	0	0
Number of memory bits	0	0	0	0
Number of processes	0	0	0	0
Number of cells	89	358	1308	4935
AND2_X1	3	3	2	14
BUF_X1	0	0	0	3
BUF_X2	1	4	13	218
CLKBUF_X1	0	0	1	6
CLKBUF_X2	0	11	67	182
INV_X1	9	31	47	92
NAND2_X1	49	83	315	3392
NOR2_X1	26	226	863	1026

Table 6.2: Statistics: Bit Shift Arithmetic Right

3. Shift Rotate Left:

A shift rotate left move bits circularly by the specified number of times. In this case, the Least Significant Bit(LSB) becomes the Most Significant Bit(MSB). Refer to the Fig.6.3.

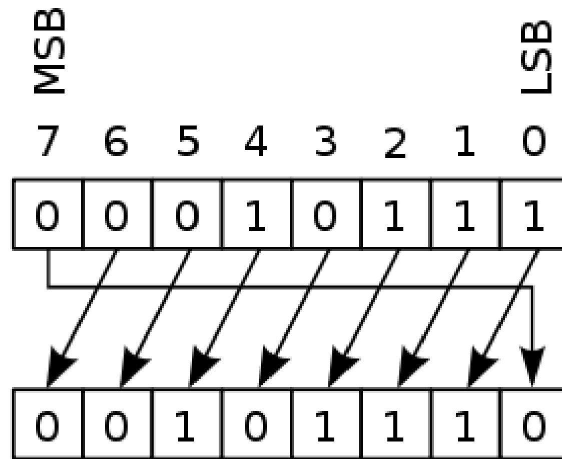

Figure 6.3: Example: Shift Rotate Left

```
ABC Results Bit Rotate Left (64 bits):
Gates = 9214  Cap = 1.5 ff  Area = 1860.45  Delay = 64.64 ps
Path  0 -- 3 pi         A = 0.00  Df = 1.1  ps  Cin = 0.0 ff  Cout = 2.4 ff
Path  1 -- 2 INV_X1     A = 0.15  Df = 3.2  ps  Cin = 0.8 ff  Cout = 1.5 ff
Path  2 -- 1 NOR2_X1    A = 0.20  Df = 5.7  ps  Cin = 0.8 ff  Cout = 0.8 ff
Path  3 -- 7 INV_X1     A = 0.15  Df = 10.8 ps  Cin = 0.8 ff  Cout = 5.8 ff
Path  4 --10 BUF_X2     A = 0.25  Df = 18.4 ps  Cin = 0.8 ff  Cout = 8.3 ff
Path  5 -- 2 NOR2_X1    A = 0.20  Df = 22.5 ps  Cin = 0.8 ff  Cout = 1.6 ff
Path  6 --10 BUF_X2     A = 0.25  Df = 29.3 ps  Cin = 0.8 ff  Cout = 7.7 ff
Path  7 --10 CLKBUF_X2 A = 0.25  Df = 36.3 ps  Cin = 0.8 ff  Cout = 7.3 ff
Path  8 -- 1 NAND2_X1  A = 0.20  Df = 39.2 ps  Cin = 0.8 ff  Cout = 0.8 ff
Path  9 -- 1 NAND2_X1  A = 0.20  Df = 41.6 ps  Cin = 0.8 ff  Cout = 0.8 ff
Path 10 -- 1 NOR2_X1    A = 0.20  Df = 43.5 ps  Cin = 0.8 ff  Cout = 0.7 ff
Path 11 -- 1 NAND2_X1  A = 0.20  Df = 45.3 ps  Cin = 0.8 ff  Cout = 0.8 ff
Path 12 -- 1 NOR2_X1    A = 0.20  Df = 47.3 ps  Cin = 0.8 ff  Cout = 0.8 ff
Path 13 -- 1 NAND2_X1  A = 0.20  Df = 49.7 ps  Cin = 0.8 ff  Cout = 0.8 ff
Path 14 -- 1 NOR2_X1    A = 0.20  Df = 52.8 ps  Cin = 0.8 ff  Cout = 0.8 ff
Path 15 -- 1 NAND2_X1  A = 0.20  Df = 64.6 ps  Cin = 0.8 ff  Cout =10.0 ff
```

Statistics: Bit Shift Rotate Left				
	8-bits	16-bits	32-bits	64-bits
Delay (ps)	32.15	42.94	57.19	64.64
Number of wires	212	733	2709	9958
Number of wire bits	379	1444	5630	21855
Number of public wires	12	20	36	68
Number of public wire bits	99	324	1157	4358
Number of memories	0	0	0	0
Number of memory bits	0	0	0	0
Number of processes	0	0	0	0
Number of cells	135	572	2354	9214
AND2_X1	0	0	3	66
BUF_X2	0	16	2	513
CLKBUF_X2	0	16	192	384
INV_X1	3	20	63	36
NAND2_X1	108	96	447	6740
NOR2_X1	24	424	1646	1474

Table 6.3: Statistics: Bit Shift Rotate Left

4. Shift Rotate Right:

A shift rotate left move bits circularly by the specified number of times. In this case, the Least Significant Bit(LSB) becomes the Most Significant Bit(MSB). Refer to the Fig.6.4

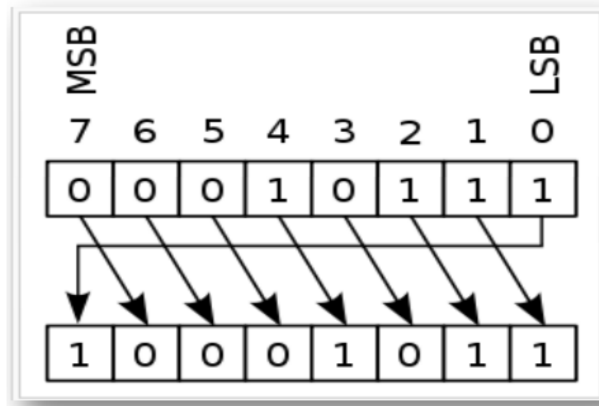

Figure 6.4: Example: Shift Rotate Right

```
ABC Results Bit Rotate Right (64 bits):
Gates = 9214  Cap = 1.5 ff  Area = 1860.45  Delay = 64.64 ps
Path  0 -- 3 pi         A = 0.00  Df = 1.1  ps  Cin = 0.0 ff  Cout = 2.4 ff
Path  1 -- 2 INV_X1     A = 0.15  Df = 3.2  ps  Cin = 0.8 ff  Cout = 1.5 ff
Path  2 -- 1 NOR2_X1    A = 0.20  Df = 5.7  ps  Cin = 0.8 ff  Cout = 0.8 ff
Path  3 -- 7 INV_X1     A = 0.15  Df = 10.8 ps  Cin = 0.8 ff  Cout = 5.8 ff
Path  4 --10 BUF_X2     A = 0.25  Df = 18.4 ps  Cin = 0.8 ff  Cout = 8.3 ff
Path  5 -- 2 NOR2_X1    A = 0.20  Df = 22.5 ps  Cin = 0.8 ff  Cout = 1.6 ff
Path  6 --10 BUF_X2     A = 0.25  Df = 29.3 ps  Cin = 0.8 ff  Cout = 7.7 ff
Path  7 --10 CLKBUF_X2  A = 0.25  Df = 36.3 ps  Cin = 0.8 ff  Cout = 7.3 ff
Path  8 -- 1 NAND2_X1   A = 0.20  Df = 39.2 ps  Cin = 0.8 ff  Cout = 0.8 ff
Path  9 -- 1 NAND2_X1   A = 0.20  Df = 41.6 ps  Cin = 0.8 ff  Cout = 0.8 ff
Path 10 -- 1 NOR2_X1    A = 0.20  Df = 43.5 ps  Cin = 0.8 ff  Cout = 0.7 ff
Path 11 -- 1 NAND2_X1   A = 0.20  Df = 45.3 ps  Cin = 0.8 ff  Cout = 0.8 ff
Path 12 -- 1 NOR2_X1    A = 0.20  Df = 47.3 ps  Cin = 0.8 ff  Cout = 0.8 ff
Path 13 -- 1 NAND2_X1   A = 0.20  Df = 49.7 ps  Cin = 0.8 ff  Cout = 0.8 ff
Path 14 -- 1 NOR2_X1    A = 0.20  Df = 52.8 ps  Cin = 0.8 ff  Cout = 0.8 ff
Path 15 -- 1 NAND2_X1   A = 0.20  Df = 64.6 ps  Cin = 0.8 ff  Cout =10.0 ff
```

Statistics: Bit Shift Rotate Right				
	8-bits	16-bits	32-bits	64-bits
Delay (ps)	32.15	42.94	57.19	64.64
Number of wires	212	733	2709	9958
Number of wire bits	379	1444	5630	21855
Number of public wires	12	20	36	68
Number of public wire bits	99	324	1157	4358
Number of memories	0	0	0	0
Number of memory bits	0	0	0	0
Number of processes	0	0	0	0
Number of cells	135	572	2354	9214
AND2__X1	0	0	3	66
BUF__X2	0	16	2	513
CLKBUF__X2	0	16	192	384
INV__X1	3	20	63	36
NAND2__X1	108	96	447	6740
NOR2__X1	24	424	1646	1474

Table 6.4: Statistics: Bit Shift Rotate Right

Chapter 7

Booth Multiplier

Multiplication can be computed by simplifying the numbers to base 2. The product of A x B with X and Y bits respectively can be performed in two simple steps. First, by generating Y number of partial products of X bits each. Second, by simply adding the generated partial products from step one. The partial products are generated using the Radix encoding and shifting the bits appropriately for the negative sign. Each column of partial products must then be added and, if necessary, any carry values passed to the next column. An Example is show in Fig.7.1.[7]

Booth Multiplier is an algorithm that multiplies two signed binary in two's complement form. The bottleneck of a multiplier is the addition of partial products. The more the partial products are the more time it takes to compute the results. The whole point of using a booth multiplier is to reduce the number of partial products generated. We have designed the multiplier using Radix-4 Booth encoding which further reduces the number of partial products to n/2 if we are multiplying two n bit numbers. Radix-4 Booth encoding values are show in table. 7.1

```
Multiplicand          011001  :  25_10
Multiplier          x 100111  :  39_10
              ---------------------
                    011001
Partial            011001
Products          011001
                 000000
                000000
              + 011001
              ---------------------
Product       001111001111 : 975_10
```

Figure 7.1: Example: Multiplication

Radix-4 Booth Encoding Values			
X(2i+1)	X(2i)	X(2i-1)	Partial Product
0	0	0	0
0	0	1	Y
0	1	0	Y
0	1	1	2Y
1	0	0	-2Y
1	0	1	-Y
1	1	0	-Y
1	1	1	0

Table 7.1: Radix-4 Booth Encoding Values

The multiplier designed is completely configurable where we can give the parameters required for 8,16,32,64 and so on bits to be multiplied. The synthesis results are shown in Figure below.

Statistics: Booth Multiplier				
	8-bits	16-bits	32-bits	64-bits
Delay (ps)	118.16	258.49	561.12	1171.70
Number of wires	944	3640	15070	60986
Number of wire bits	1484	5684	23194	93688
Number of public wires	18	30	54	102
Number of public wire bits	177	545	1857	6785
Number of memories	0	0	0	0
Number of memory bits	0	0	0	0
Number of processes	0	0	0	0
Number of cells	665	2624	11093	45050
AND2__X1	07	48	187	599
AND2__X2	0	0	45	97
BUF__X2	0	6	105	623
CLKBUF__X1	0	0	0	4
CLKBUF__X2	0	51	259	896
CLKBUF__X4	0	0	1	5
INV__X1	89	346	1323	5850
INV__X2	06	07	96	267
INV__X4	0	9	09	29
NAND2__X1	209	792	2840	11399
NAND2__X2	31	71	620	1212
NOR2__X1	262	1130	4766	21182
NOR2__X2	34	36	389	917
OR2__X1	07	33	109	327
OR2__X2	0	01	09	25
XNOR2__X1	01	10	34	187
XOR2__X1	19	92	298	1431

Table 7.2: Statistics: RADIX-4 Booth Multiplier

Chapter 8

Results

Results of all the modules are tabulated below along with the bits, delay and number of cells for each.

Results:Adders			
		Delay(ps)	Number of Cells
Ripple Carry Adder	8-bit	48.59	40
	16-bit	86.82	76
	32-bit	120.21	150
	64-bit	240.48	294
Carry Save Adder 3 - Inputs	8-bit	61.22	81
	16-bit	86.66	190
	32-bit	127.05	448
	64-bit	248.21	880
Kogge-Stone Adder	8-bit	31.97	59
	16-bit	39.49	168
	32-bit	48.07	454
	64-bit	57.01	1100
Han-Carlson Adder	16-bit	77.28	96
	32-bit	62.67	286

Table 8.1: Results:Adders

Results			
		Delay(ps)	Number of Cells
Bit-Shift Left	8-bit	32.46	89
	16-bit	45.39	310
	32-bit	58.38	1259
	64-bit	67.98	4739
Bit-Shift Right	8-bit	33.75	89
	16-bit	50.90	358
	32-bit	59.06	1308
	64-bit	72.56	4935
Bit-Shift Rotate Left	8-bit	32.15	135
	16-bit	42.94	572
	32-bit	57.19	2354
	64-bit	64.64	9214
Bit-Shift Rotate Right	8-bit	32.15	135
	16-bit	42.94	572
	32-bit	57.19	2354
	64-bit	64.64	9214
Radix-4 Booth Multiplier	8-bit	118.16	665
	16-bit	258.49	2624
	32-bit	561.12	11093
	64-bit	1171.70	45050

Table 8.2: Results

Chapter 9

Conclusion

A base has been created by creating this library where the modules can be used as per the required number of bits. An engineer would not need to create this modules redundantly and can be directly instantiated while designing complex modules.

Current modules include Binary Adders, Parallel Prefix Adder, Shifters and Radix-4 Booth Multiplier. All the modules are designed successfully and are synthesizable. Extensive synthesis results are attached and also available at the repository.

In future, we can continue to grow our library as per required by adding more modules to it.

Bibliography

[1] R. P. Brent and H. T. Kung. A regular layout for parallel adders. *IEEE Trans. Comput.*, 31(3):260–264, March 1982.

[2] Leininger Joel Calvin and Taylor George Phillips. Carry save adder.

[3] T. Han and D. A. Carlson. Fast area-efficient vlsi adders. In *1987 IEEE 8th Symposium on Computer Arithmetic (ARITH)*, pages 49–56, May 1987.

[4] P. M. Kogge and H. S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, C-22(8):786–793, Aug 1973.

[5] M. Morris Mano. *Digital Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 3rd edition, 2001.

[6] S. Muthyala Sudhakar, K. P. Chidambaram, and E. E. Swartzlander. Hybrid han-carlson adder. In *2012 IEEE 55th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 818–821, Aug 2012.

[7] Neil Weste and David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison-Wesley Publishing Company, USA, 4th edition, 2010.

[8] Wikipedia. Adder. `https://en.wikipedia.org/wiki/Adder_(electronics)`, 2018. [Online; accessed 05-May-2018].

[9] Clifford Wolf. Yosys open synthesis suite. `http://www.clifford.at/yosys/`.