IMPLEMENTATION OF 32 BIT BRENT KUNG ADDER USING COMPLEMENTARY PASS TRANSISTOR LOGIC

By

NOEL DANIEL GUNDI

Bachelor of Engineering in Electronics and Communication,

Visvesvaraya Technological University,

Belgaum, Karnataka, India

June 2008

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 2015

IMPLEMENTATION OF 32 BIT BRENT KUNG

ADDER USING COMPLEMENTARY PASS

TRANSISTOR LOGIC

Thesis Approved:

Dr. Louis G. Johnson

Thesis Adviser

Dr. Rama G. Ramakumar

Dr. Weihua Sheng

ACKNOWLEDGEMENTS

I extend by heartfelt Thanks to Dr. Johnson for giving me an opportunity to work with him and guiding me all the while. I thank my family and all my friends for being with me throughout the journey. And most importantly I thank God for everything.

Name: NOEL DANIEL GUNDI

Date of Degree: MAY, 2015

Title of Study: IMPLEMENTATION OF 32 BIT BRENT KUNG ADDER USING COMPLEMENTARY PASS TRANSISTOR LOGIC

Major Field: ELECTRICAL AND COMPUTER ENGINEERING

Abstract: Adders are the most vital part of any digital system. Providing an efficient adder design which satisfies the tradeoff between speed and space aides in increasing the performance of the system. In the modern age in addition to the tradeoff between speed and space, power consumption plays a vital role. Devices with low power consumption and good performance are always preferred. Parallel Prefix adder are the ones widely used in Digital Design. This is primarily because of the flexibility in designing the Adders. Brent Kung Adder is a low power adder, as it uses minimum circuitry to obtain the result. The use of Complementary Pass transistor Logic aides in increasing the performance of the design by using the multiplexer approach in designing the various cells. The 16 bit design is extended to 32 bit, implemented in the physical level and successfully simulated. The area and delay results are accordingly illustrated.

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

CHAPTER I

INTRODUCTION

1.1 Adder is Digital Design

Addition is one of the four elementary operations in mathematics, the other being subtraction, multiplication and division. In digital systems, addition forms the most important operation. This is primarily because we can perform operations like subtraction, multiplication and division using the addition operation. Hence the design of a very fast, accurate and a lower power consumption adder directly results in the increased speed of the device for faster computational purpose as well as an improved life.

1.2 Design of an Adder

The basic complete form of addition can be implemented by the Half Adder and improvised by Full Adder. The generated Carry bit plays a very important role in the adder design. The speed of the adder is decided to a great extent by the Carry propagation. To reduce the Carry propagation delay multiple adders are designed. Some of the most commonly used adders are Carry Look-Ahead adder, Carry Skip Adder, Carry Select adder, Manchester chain adder and Parallel Prefix Adders.

1.3 Motivation

While designing an adder, lot of constraints comes into picture. The tradeoff between speed and size being the most important one. The most basic adder has a very small area and is very easy to implement. But the delay involved in obtaining the output is very huge. Hence the new designs came to picture. But in current age the power consumed by the device is also a very important constraint. Hence designing an adder which is very close in meeting all these requirements is a very important.

1.2 Adder Implementation

Apart from the design, the technology we use to design the adder also plays a huge role in the speed and size of the adder. In normal implementation procedure we use the CMOS logic. Additionally we have Static CMOS, Differential Pass Transistor logic, Lean Integration Pass Transistor logic and Complementary Pass Transistor Logic. These are some of the commonly used technologies out of the many. The Complementary Pass Transistor Logic will help in a great deal in increasing the signal strength at each stage. Also we will obtain two signals for each entity (one being the complement of the other).

The design can be extended to greater bits in pursuit of catering to the needs of the current processors. Also various other technologies can be integrated together in order to get an overall better result.

CHAPTER II

BACKGROUND

2.1 Half Adder

Half Adder is digital circuit which performs addition of two bits. A Half adder takes in two inputs 'A' and 'B'. It gives two outputs 'Sum' and 'Carry'. The overflow in case of a multi digit addition is indicated by the 'Carry' signal. The logic diagram and truth table of a half adder is as given in figure 1.1 and Table 1.

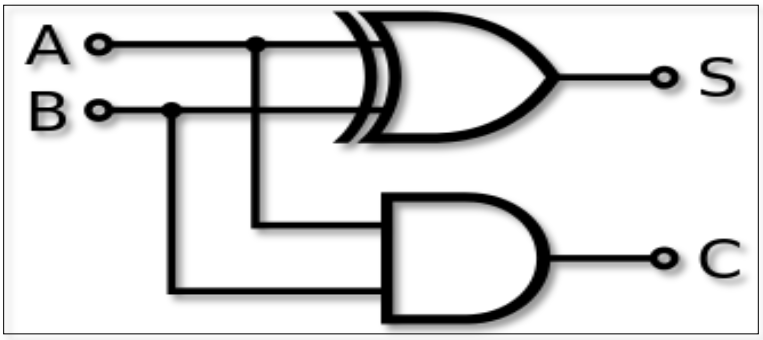

Figure 1.1 Half Adder

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table1. Truth Table of Half Adder

The half adders are very useful as long the addition is comprised of 2 bit addition. For adding more than two bits at time we will extend the half adder logic.

2.2 FULL ADDER

Full adder takes in three inputs 'A', 'B', 'C' and gives two outputs 'Sum' and 'Carry'. Normally the third input is often termed as 'Cin' to indicate it as a carry from the previous addition stage.

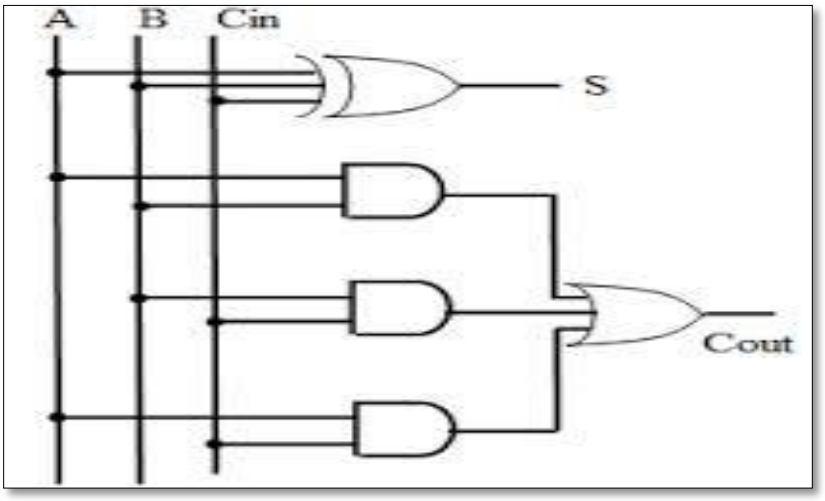

Figure 1.2 Full Adder

Inputs			SUM	CARRY
A	B	C		
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table2. Truth Table of Full Adder

The Full adder circuit can be implemented using half adders. Two half adders and an OR gate is used for this implementation.

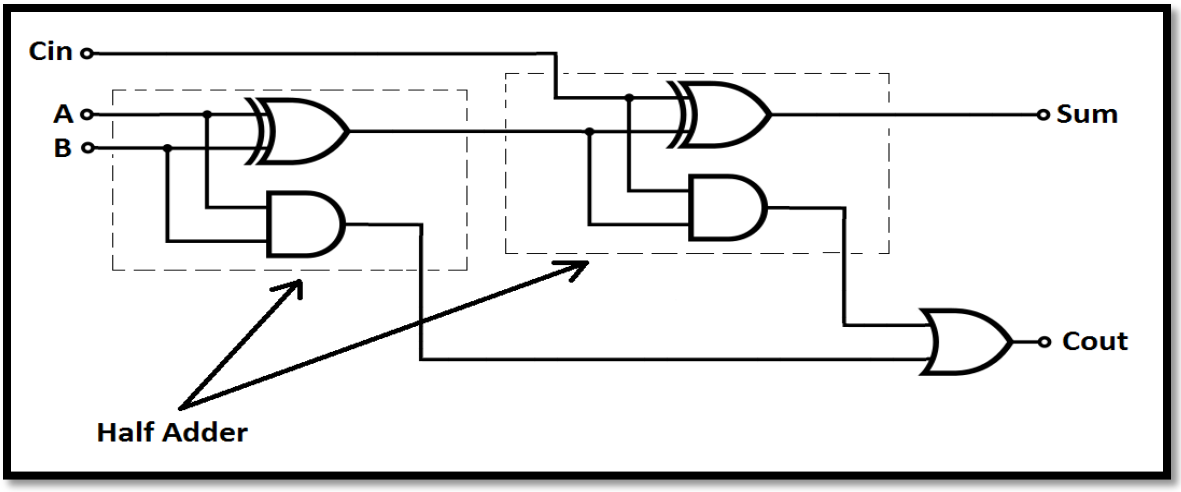

Figure 1.3 Full Adder using Half Adders

The optimized design of a Full Adder is as given in figure 1.4. Here, the Half Adder is derived using basic gates.

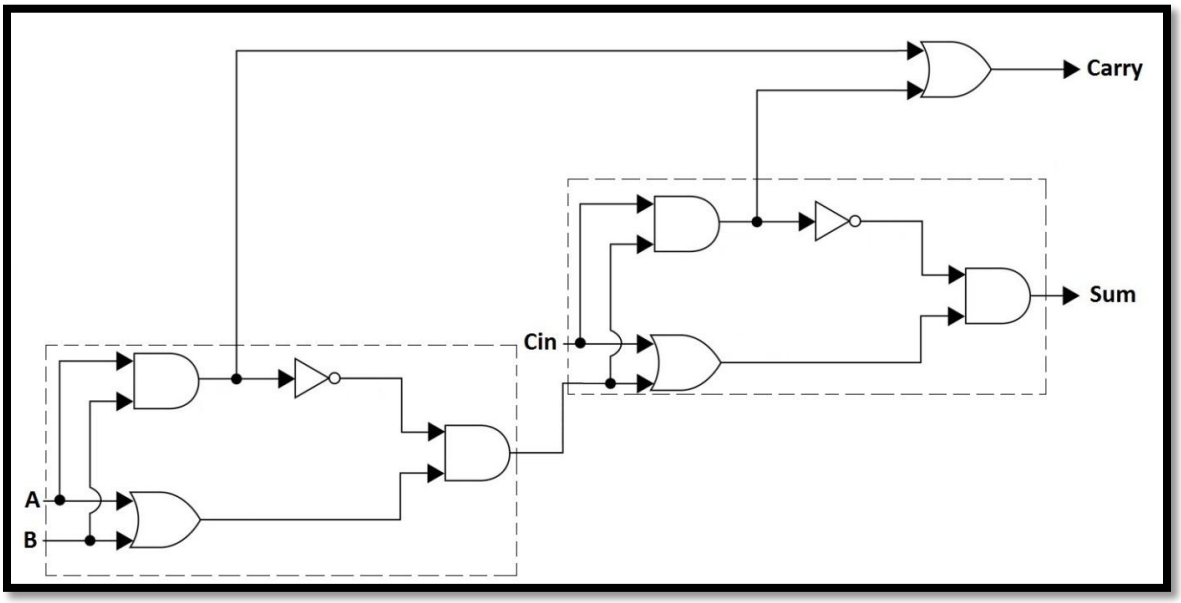

Figure 1.4 Full Adder using Basic Gates

$$(A+B).\overline{(A.B)} = (A+B).(\bar{A}+\bar{B})$$
$$= A.\bar{A} + A.\bar{B} + \bar{A}.B + B.\bar{B}$$
$$= A.\bar{B} + \bar{A}.B$$

2.3 Ripple Carry Adders

The most basic and simplest implementation of an 'n' bit adder is the Ripple Carry Adder. The Ripple Carry Adder is implemented using a series of Full Adders. The carry out of the i^{th} stage is given as the input to the carry in of the $(i+1)^{th}$ stage. The figure 1.5 shows the n bit Ripple Carry Adder.

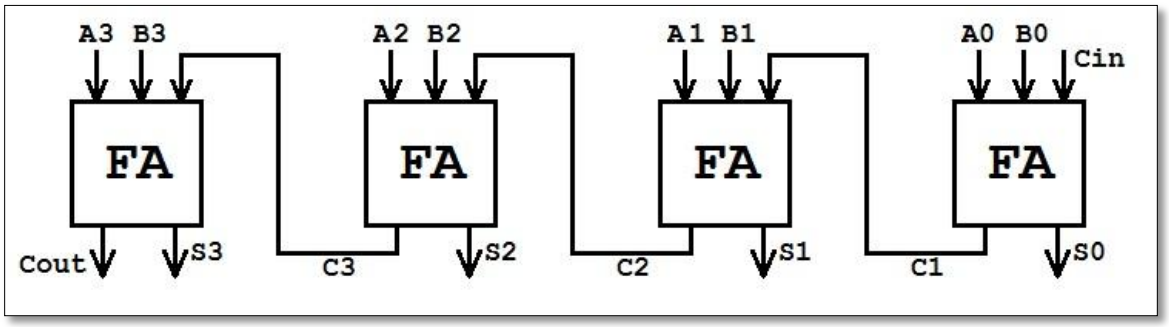

Figure 1.5 Ripple Carry Adder

Here we see that the delay in each Full Adder is primarily because of the carry output of the previous stage. The output is obtained the sequence as below.

C1-> S0 -> C2 -> S1 -> C3 -> S2 -> Cout -> S3

The form in which the output from each stage is obtained is similar to a ripple. Hence the name Ripple Carry Adder. The Ripple carry adder is very simple in its design, as it is implemented just using Full Adders. But the major drawback turns out to be the delay in obtaining the output. Each stage is dependent on the carry produced by the previous stage. This causes a major delay as we have to wait until the carries are generated along the way.

Based on the design, Ripple Carry Adder proves to be a major building block for the various future adder designs.

2.4 Carry look ahead adder

In the Ripple Carry Adder mechanism, we found that the primary cause of concern is the delay produced because of the Carry generated in each phases. Hence we now look into a scheme where we are generating Carry's ahead of time thus trying to mitigate the delays produced in the process.

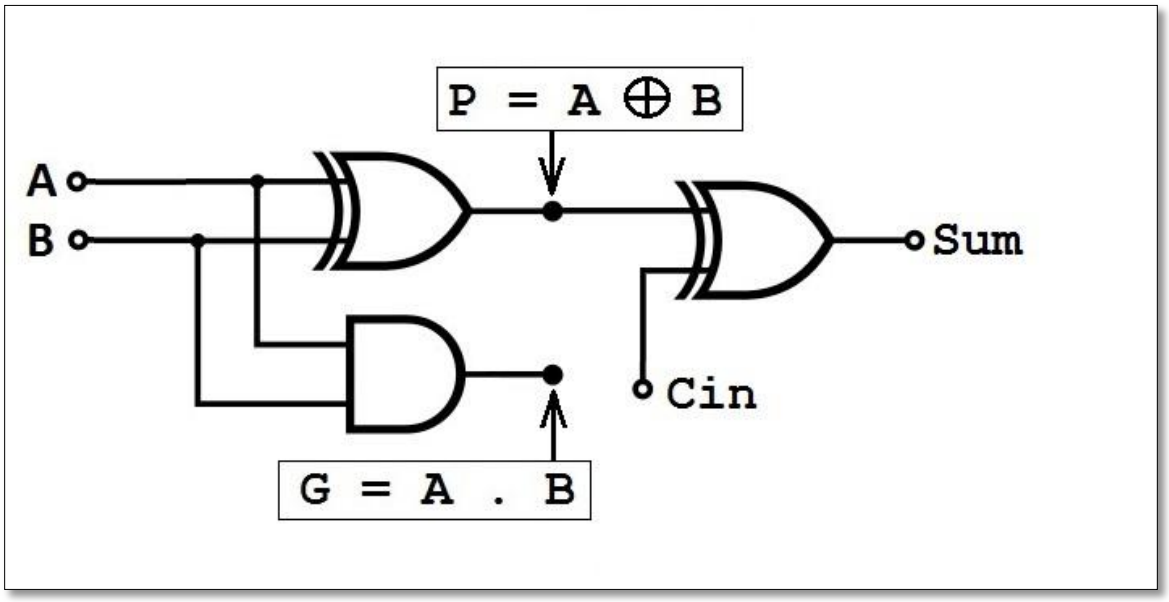

Figure 1.6 Full Adder with Generate and Propagate signal

In the figure 1.6, we see the Full Adder as modified to our needs. The output of the first EXOR gate is represented as a Propagate signal (P = A ⊕ B) and the output of the AND gate is represented as a Generate signal (G = A. B). We have already seen the design of a Full Adder using a Half Adder. The equation for carry signal is as given below. Considering the Inputs as A_0, B_0, C_{in}, A_1, B_1, A_2, B_2, A_3, B_3 the corresponding carries generated using the generate and propagate signals are as given below.

$C_1 = G_0 + P_0C_0$

$C_2 = G_1 + P_1G_0 + P_1P_0C_0$

$C_3 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$

$G_{3:0}$ and $P_{3:0}$ are the Group Generate and Group Propagate signals which will be used when we extend the adder to more number of bits.

$G_{3:0} = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$

$P_{3:0} = P_3P_2P_1P_0$

These carries are generated using a Carry Look-Ahead Block. The Carry's generated using the Carry Look-Ahead block are given to the respective modified Full Adders. The diagram of 4 bit Carry Look-Ahead Adder is as given in figure 1.7.

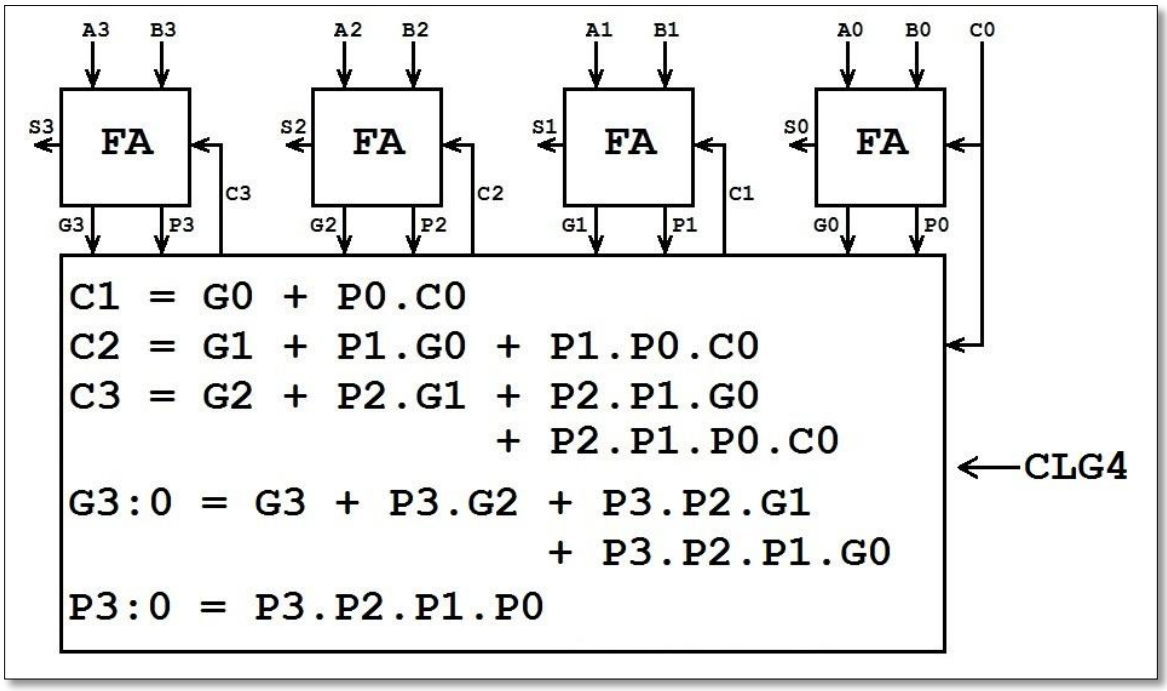

Figure 1.7 4-bit Carry Look-Ahead Adder

For example, if we were using a 2 bit Carry Look-Ahead Generator the Group Generate and Group Propagate signals would be as given below.

$G_{1:0} = G_1 + P_1G_0$

$P_{1:0} = P_1P_0$

Similar to 4 bit Carry Look-Ahead Generator we can have higher bit Carry Look-Ahead generators based on the number of input bits and the design. Here the delay involved in obtaining the output is reduced but the circuit number of gates involved in circuit drastically increases.

2.5 Parallel Prefix Adders

Until now we have seen that when we add two numbers we get a Sum and Carry bit. Based on the Carry Look-Ahead Adder operation we can categorize the addition operation in three different stages. They are:-

- First stage is the Pre-processing stage where we obtain the Group Generate and Group Propagate signals.
- Second stage is the Carry generation stage where we generate the carry using the Group Generate and Group Propagate signals.
- Third and Final stage is where we obtain the Sum bit using the Carry bit and the Propagate signal.

The steps illustrated above are as shown in the figure 1.8.

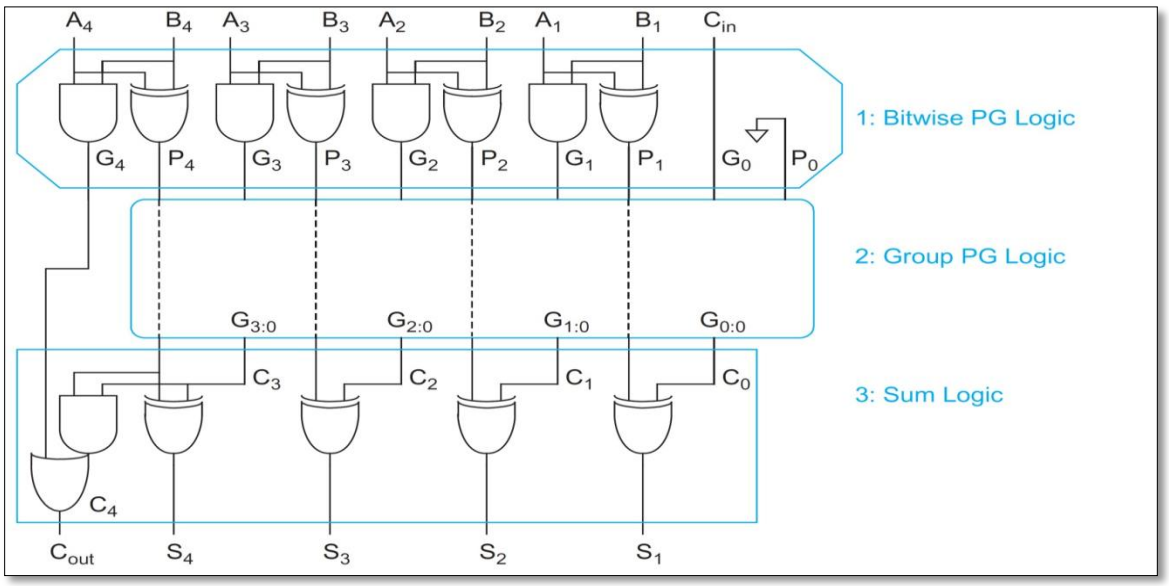

Figure 1.8 Parallel Prefix Adder Stages[1]

2.6 Carry Generation and Propagation

In the discussion of Carry Look-Ahead Adder we have seen the Group Generate and Group Propagate signals. Now if we generalize the signals we would get the generalized equations as

$G_{i:j} = G_{i:k} + P_{i:k} G_{k-1:j}$

$P_{i:j} = P_{i:k} P_{k-1:j}$

The basic case is,

$G_{i:i} = G_i = A_i B_i$

$P_{i:i} = P_i = A_i \oplus B_i$

The group generates a carry if the upper (i.e. the more significant) or the lower portion generates and the upper portion propagates the carry. If both the upper and lower portions propagate the carry, the group propagates.

The input carry is also equally important. Let us take the input carry C_{in} as $C_0 = C_{in}$ and $C_N = C_{out}$.

The Generate and Propagate signals for the bit 0 will be

$G_{0:0} = C_{in}$

$P_{0:0} = 0$.

The carry into the i^{th} bit is the carry-out of $(i-1)^{th}$ bit and $C_{i-1} = G_{i-1:0}$. This is an important relationship. The group generate signals and carries will be used synonymously in the subsequent sections.

The Sum for the i^{th} bit is computed as

$S_i = P_i \oplus G_{i-1:0}$

Hence the addition is reduced to the three step process as previously mentioned.

The equation previously defined is the valency-2 group PG logic as it combines a pair of smaller groups. We can also define higher valency group logic so that it can be possible to fewer stages of more complex gates. Given below is the valency-4 group generates and propagate logics.

$$G_{i:j} = G_{i:k} + P_{i:k} G_{k-1:l} + P_{i:k} P_{k-1:l} G_{l-1:m} + P_{i:k} P_{k-1:l} P_{l-1:m} G_{m-1:j}$$

$$= G_{i:k} + P_{i:k} (G_{k-1:l} + P_{k-1:l} (G_{l-1:m} + P_{l-1:m} G_{m-1:j})) \quad (i \geq k > l > m > j)$$

$$P_{i:j} = P_{i:k} P_{k-1:l} P_{l-1:m} P_{m-1:j}$$

As per Logical Effort, the best stage effort is about 4.

2.7 PG Carry Ripple Addition

The 4 bit carry ripple adder using PG logic is as shown below. The critical path of the ripple carry adder propagates through the carry-in to the carry-out along the carry chain of the major gates. The G and P signals will be stabilized by the time the carry arrives.

The Carry signal is generated using these G and P signals. We can simplify the majority function using an And-Or gate.

$C_i = A_i B_i + (A_i \oplus B_i) C_{i-1}$

$C_i = G_i + P_i C_{i-1}$

C_i is nothing but $G_{i:0}$. Hence ripple carry addition can now be seen as an extreme case of group PG logic in which 1-bit group is combined with and *i*-bit to obtain an (i + 1)-bit group.

$G_{i:0} = G_i + P_i G_{i-1:0}$

The figure 1.9 shows the 4-bit carry ripple adder using the AND-OR gates.

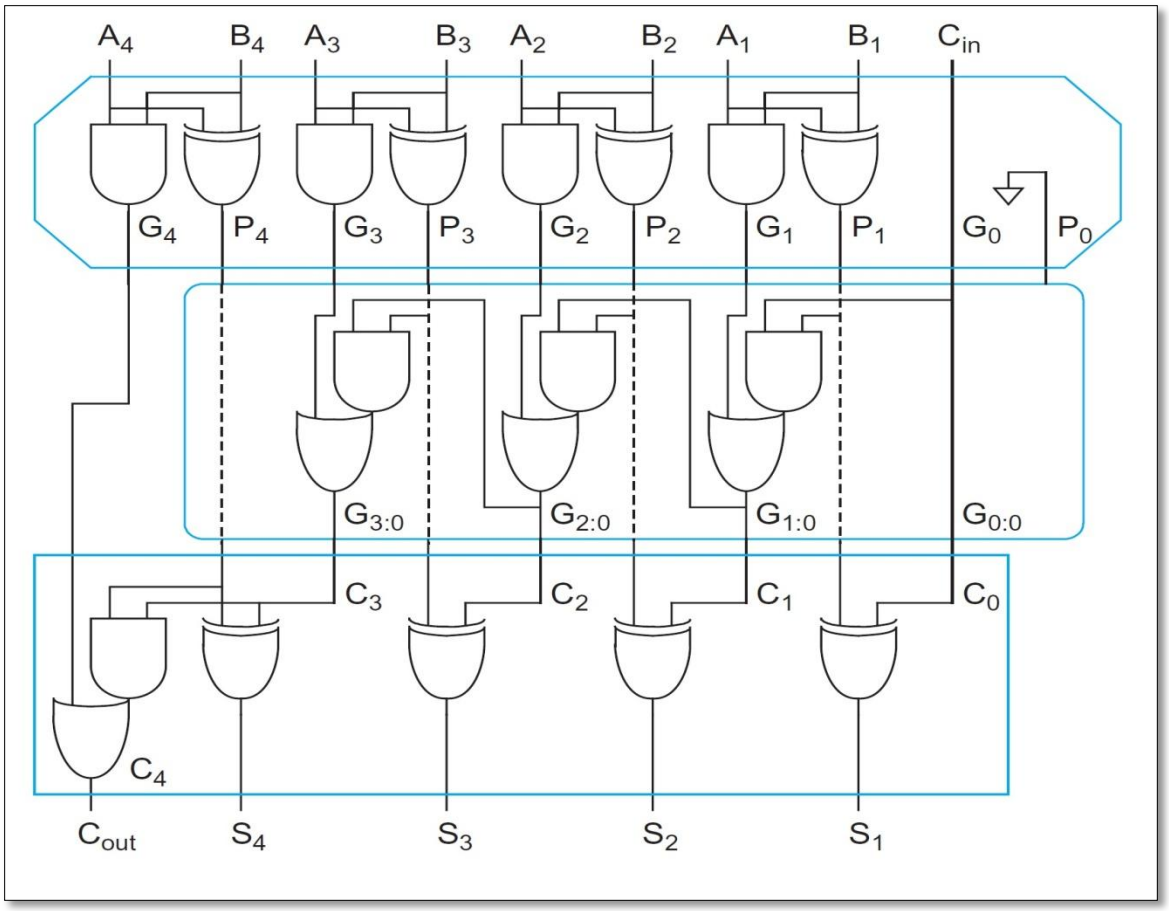

Figure 1.9 4-bit Ripple Carry Adder using Parallel Prefix Stage[1]

The critical path of the adder will now be the path through a chain of AND-OR gates rather than only a chain of major gates.

The figure 2.0 shows the 16-bit ripple carry adder using Gray Cells.

Using diagrams like the above 16 bit ripple carry adder we can compare a lot of architectures. In addition to Gray Cells, we will use Black Cells and White buffers as defined in the figures below.

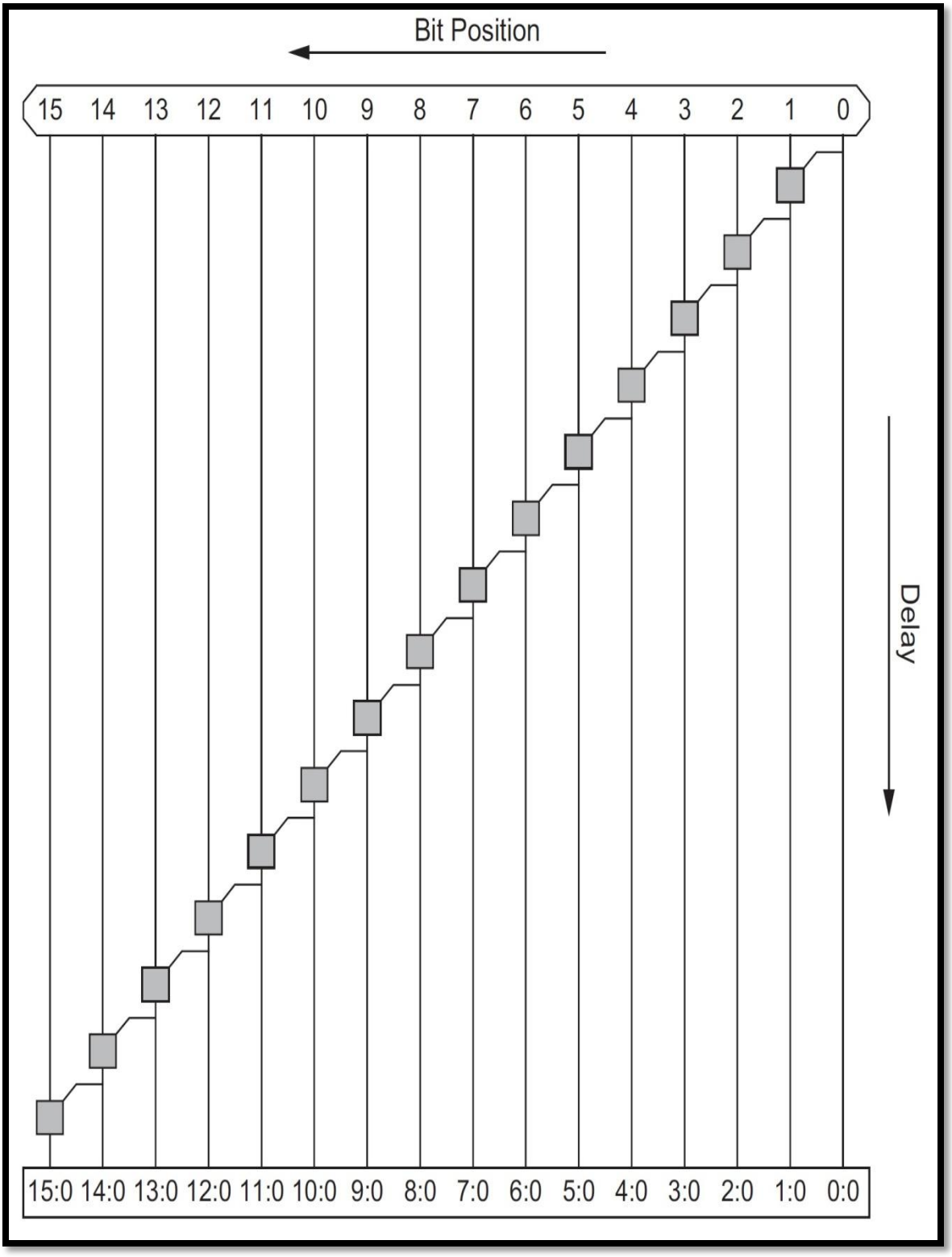

Figure 2.0 16-bit Adder using Gray Cells[1]

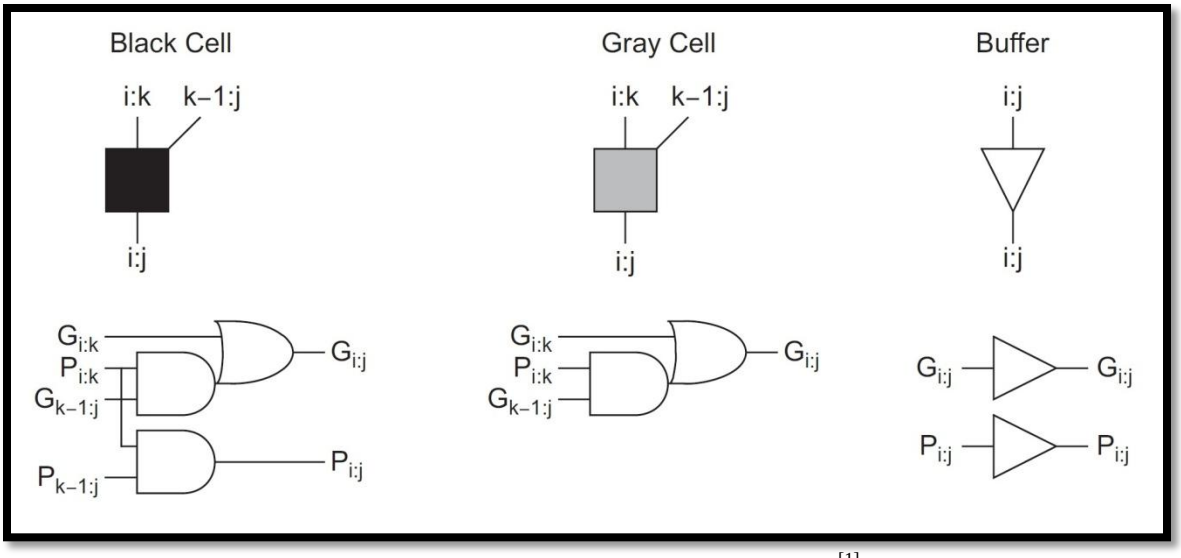

Figure 2.1 Black Cells, Gray Cells and Buffer[1]

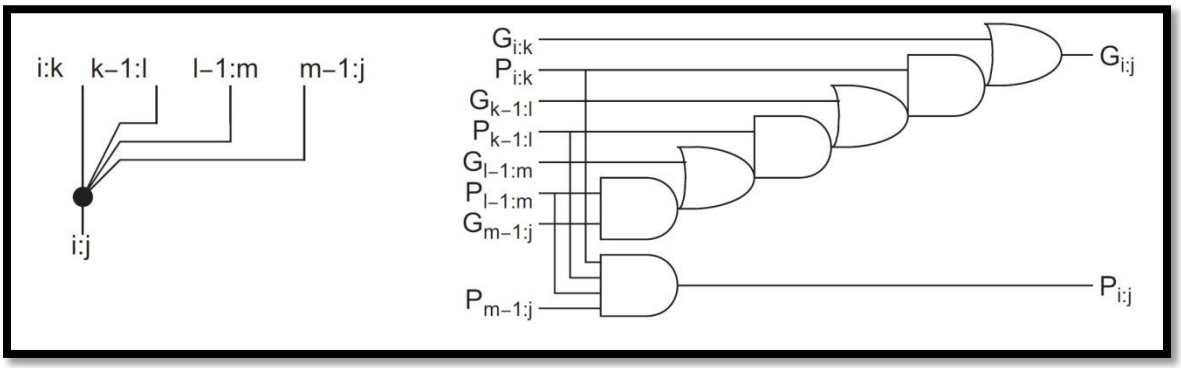

Figure 2.2 Mulitple Valency Operation[1]

Black Cells contains the Group Generate (AND-OR gate) and the Group Propagate (And gate) as defined the equation.

Gray Cells contains a Group Generate which are used at the final cell position in each column as only Group Generate will be the carry and will be used to obtain the Sums.

Buffers are used at different stages in order to reduce the load at the critical paths.

The bitwise PG is abstracted in the top box and the Sum XOR's are abstracted in the bottom boxes. It is assumed that the AND-OR gate will work in parallel in order to generate the carry-out.

The figure 2.3 shows the Carry Look-Ahead adder using group PG network.

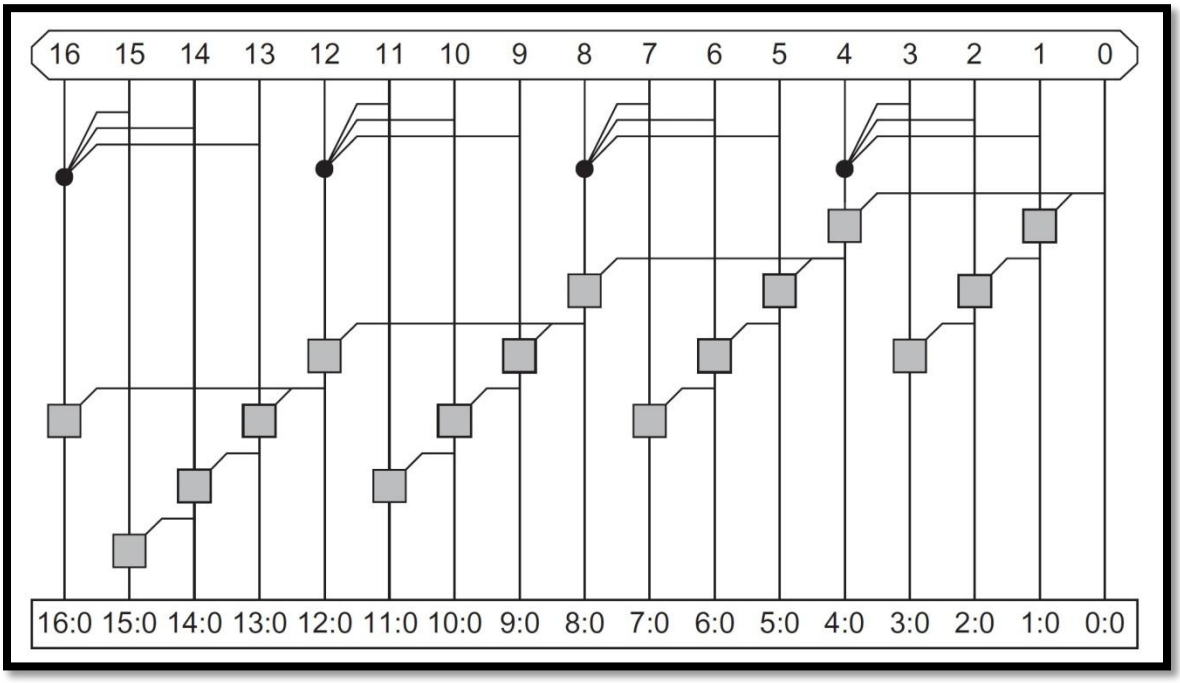

Figure 2.3 Carry Look-Ahead Adder using PG Network[1]

2.8 Tree Adders

When we design the Carry Look-Ahead Adders with larger bits (N > 16bits), the delay in the adders becomes dominant because of the delay of the Carry through the Look-Ahead blocks. This delay can be reduced by overstepping the Look-Ahead blocks. Generally we can create a multilevel tree of Look-Ahead Structures in order to obtain a delay that increases with *log N*. These adders are commonly referred to as tree adders, logarithmic adders, look-ahead adders, multilevel-look-ahead adders or simply Parallel-Prefix Adders.

Parallel Prefix Adders can be built in many ways based on the number of levels of the logic, number of logic gates involved, number of the fanout from each gate and the number of wiring between the levels. The three fundamental Tree adders are Brent-Kung, Sklansky and Kogge-Stone architectures.

The most commonly used Tree(Parallel Prefix) Adders are

1. Brent Kung
2. Sklansky
3. Kogge Stone
4. Han Carlson
5. Knowles
6. Ladner-Fischer

2.8.1 Brent Kung

The Brent Kung adder computes the prefixes for 2 bit groups. These prefixes are used to find the prefixes for the 4 bit groups, which in turn are used to compute the prefixes for 8 bit groups and so on. These prefixes are then used to compute the carry out of the particular bit stage. These carries will be used along with the Group Propagate of the next stage to compute the Sum bit of that stage. Brent Kung Tree will be using $2log_2N - 1$ stages. Since we are designing a 32-bit adder the number of stages will be 9. The fanout for each bit stage is limited to 2. The diagram below shows the fanout being minimized and the loading on the further stages being reduced. But while actually implemented the buffers are generally omitted.

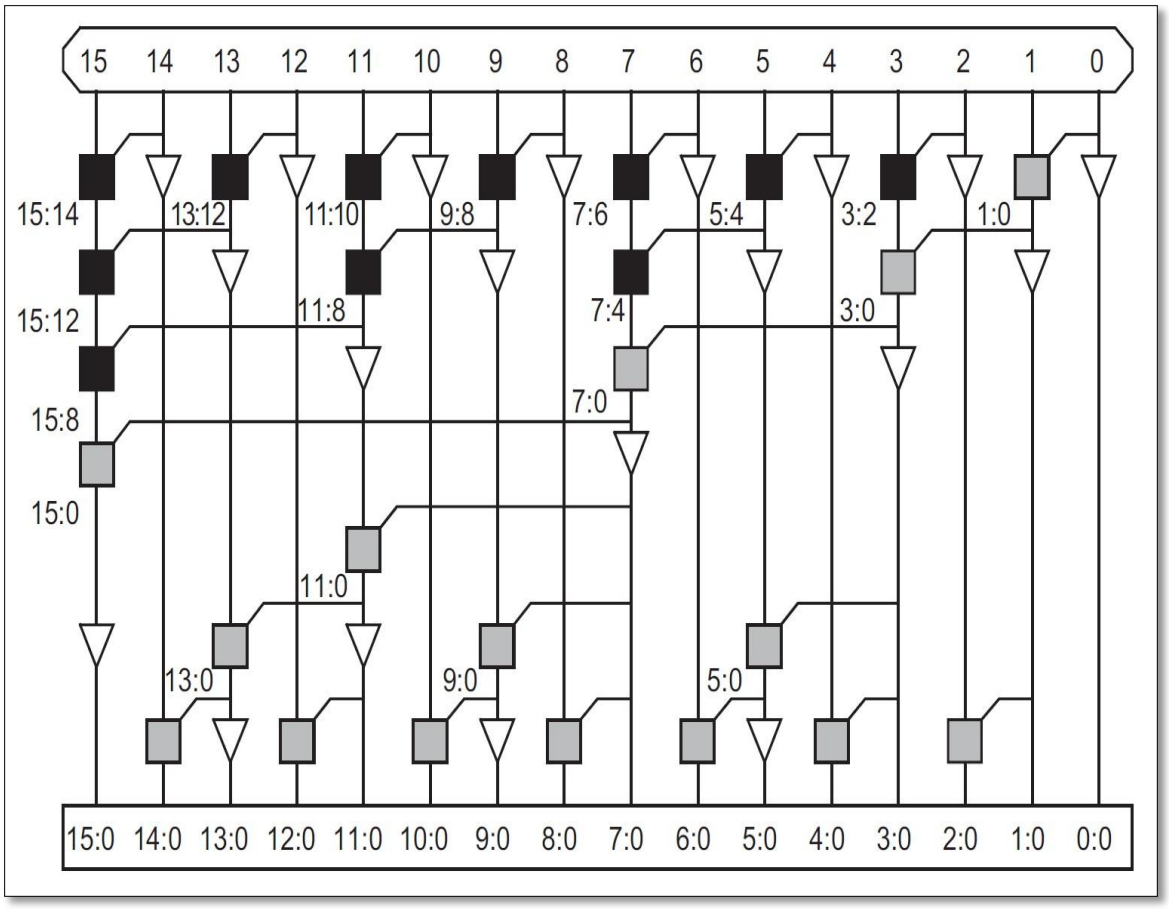

Figure 2.4 16-bit Brent Kung Adder[1]

2.8.2 Sklansky

The Sklansky tree is commonly known as the divide-and-conquer tree because it reduces the delay to log_2N stages. It computes the intermediate prefixes along with the large group prefixes. The fanouts double at each stage. The higher fanouts result in poor performance for wide adders. If the fanout gates are appropriately sized or if the critical signals are buffered before being used for the intermediate prefixes then we can see an improved performance. Transistor sizing cuts into the regularity of the design layout because the multiple sizes of each cell are required. Although the larger gates can be spread into adjacent columns. The fanouts can be reduced with appropriate buffering.

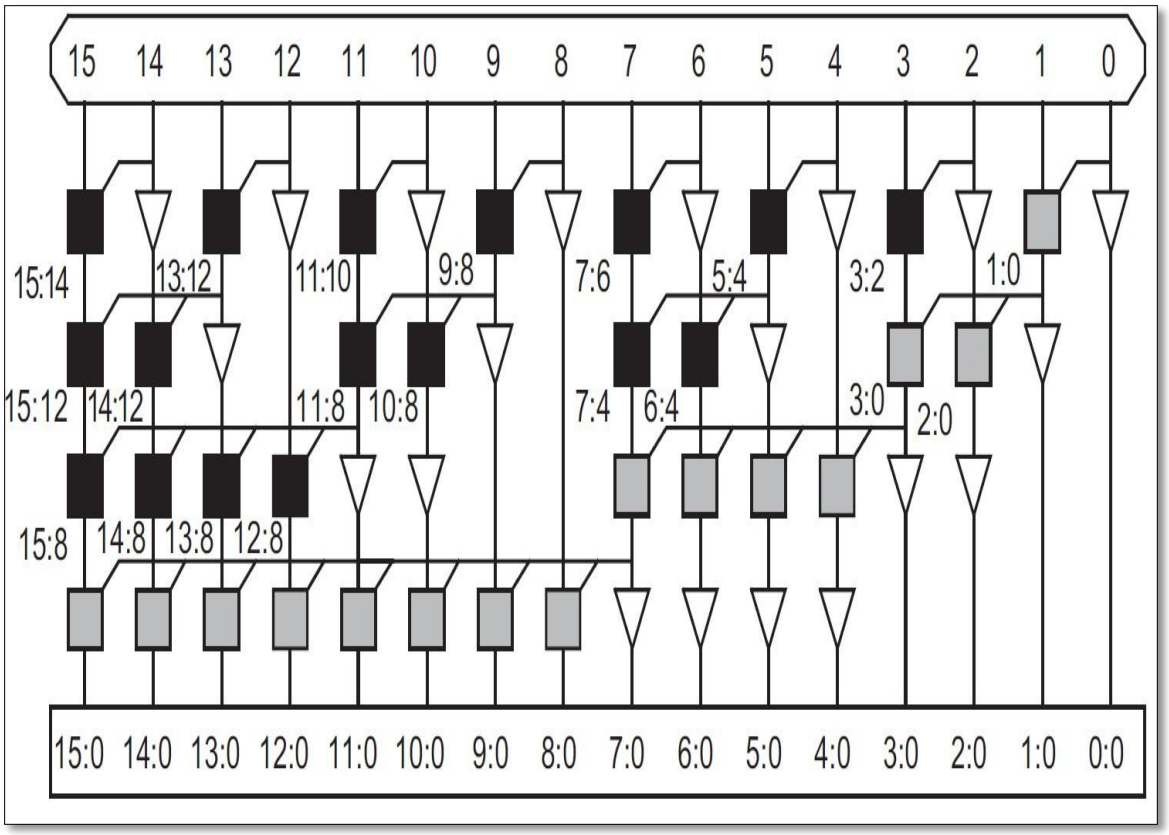

Figure 2.5 16-bit Sklansky Adder[1]

2.8.3 Kogge Stone

The fanout of 2 at each stage as well as the log_2N stages are both achieved in the Kogge Stone tree. But there is a cost involved. There are many long wires routed between the stages. This tree contains more PG cells. The layout area is not affected. But the number of gates increases to a great extent. More importantly, the power consumption is very high. Even at such high costs the Kogge-Stone tree is widely used in high performance 32-bit and 64-bit adders.

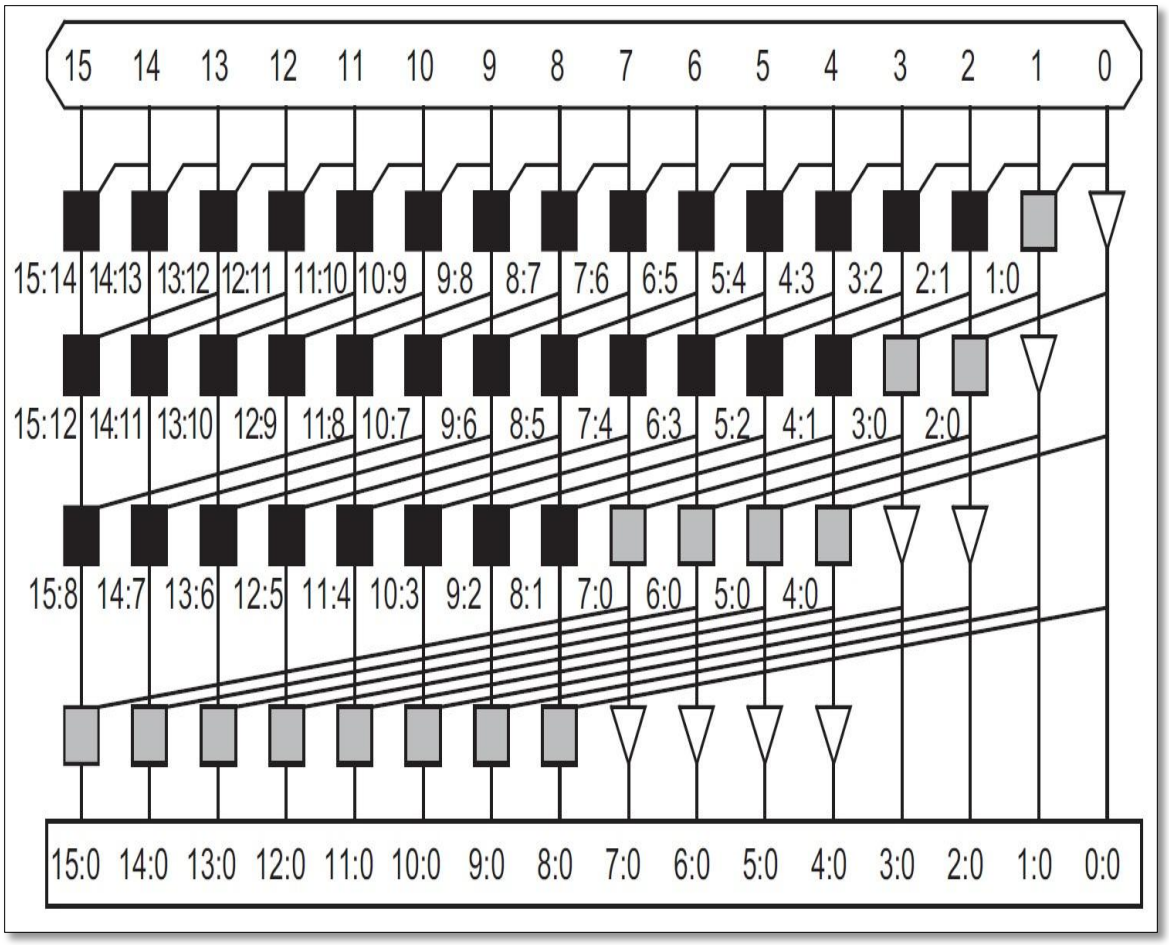

Figure 2.6 16-bit Kogge Stone Adder[1]

2.8.4 Han Carlson

Han Carlson tree are a family of tree networks between Brent-Kung and Kogge-Stone trees. This tree performs Kogge-Stone on the odd numbered bits and then to ripple between even positions one more stage is used.

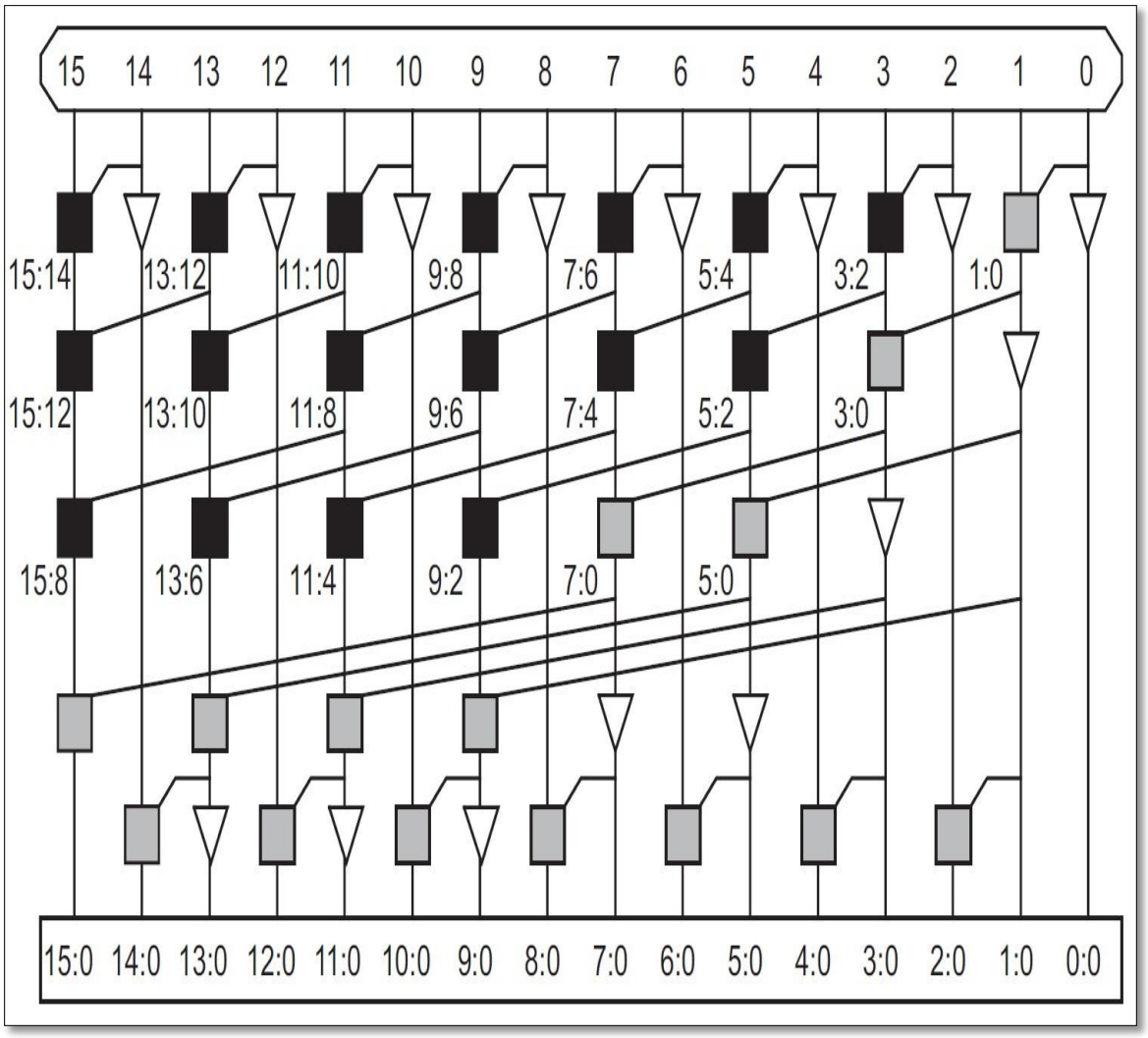

Figure 2.7 16-bit Han Carlson Adder[1]

2.8.5 Knowles

Knowles tree are a family of tree networks between Kogge-Stone and Sklansky trees. The tree differs in fanout and number of wires but has log_2N stages.

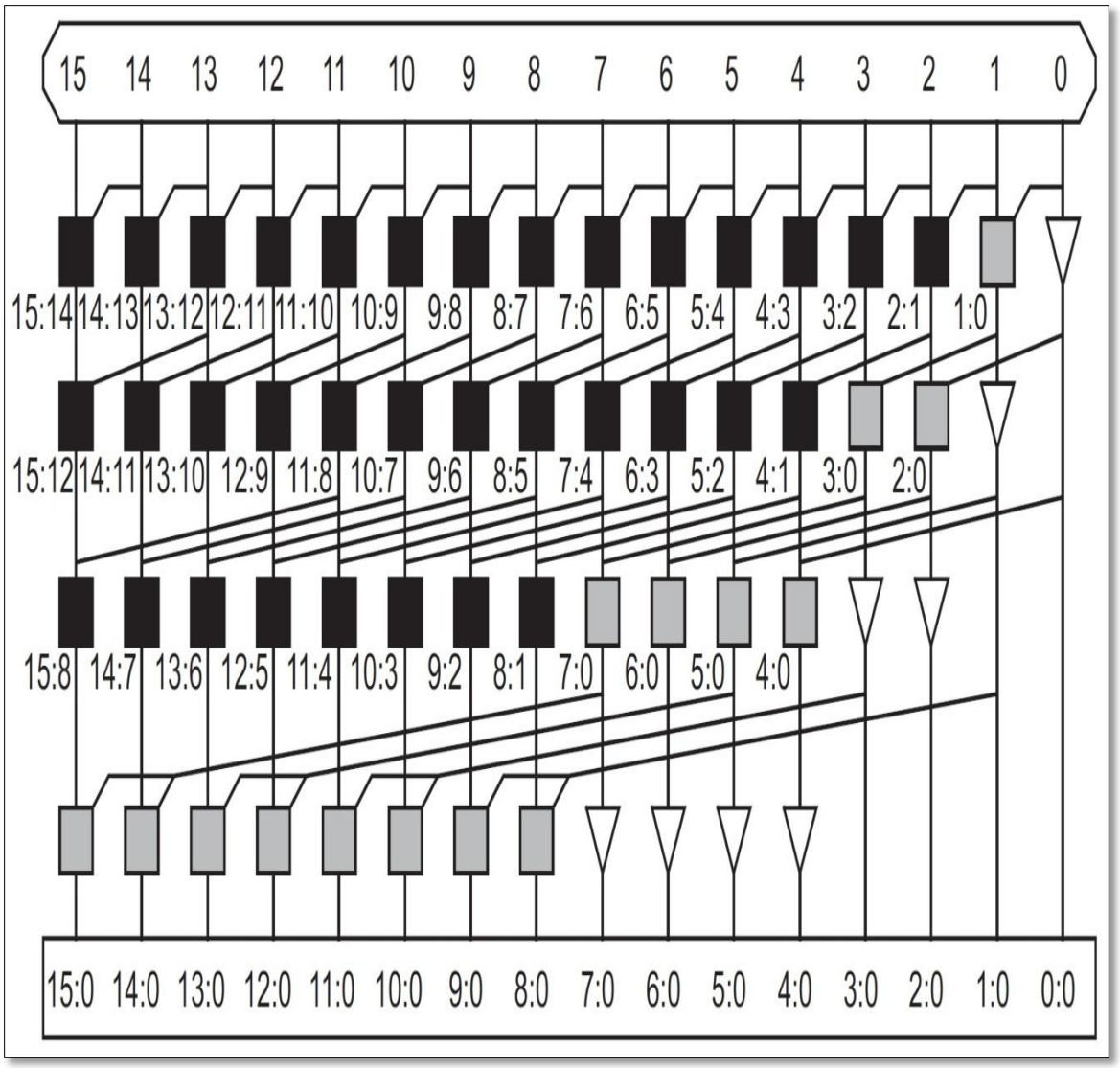

Figure 2.8 16-bit Knowles Adder[1]

2.8.6 Ladner-Fischer

Ladner-Fischer trees are a family of tree networks between Brent Kung and Sklansky tree. It is very similar to Sklansky, but it computes prefixes for odd number bits and then uses one more stage to ripple into the even positions. At higher fanout nodes, cells must still be appropriately sized or ganged to get better speeds.

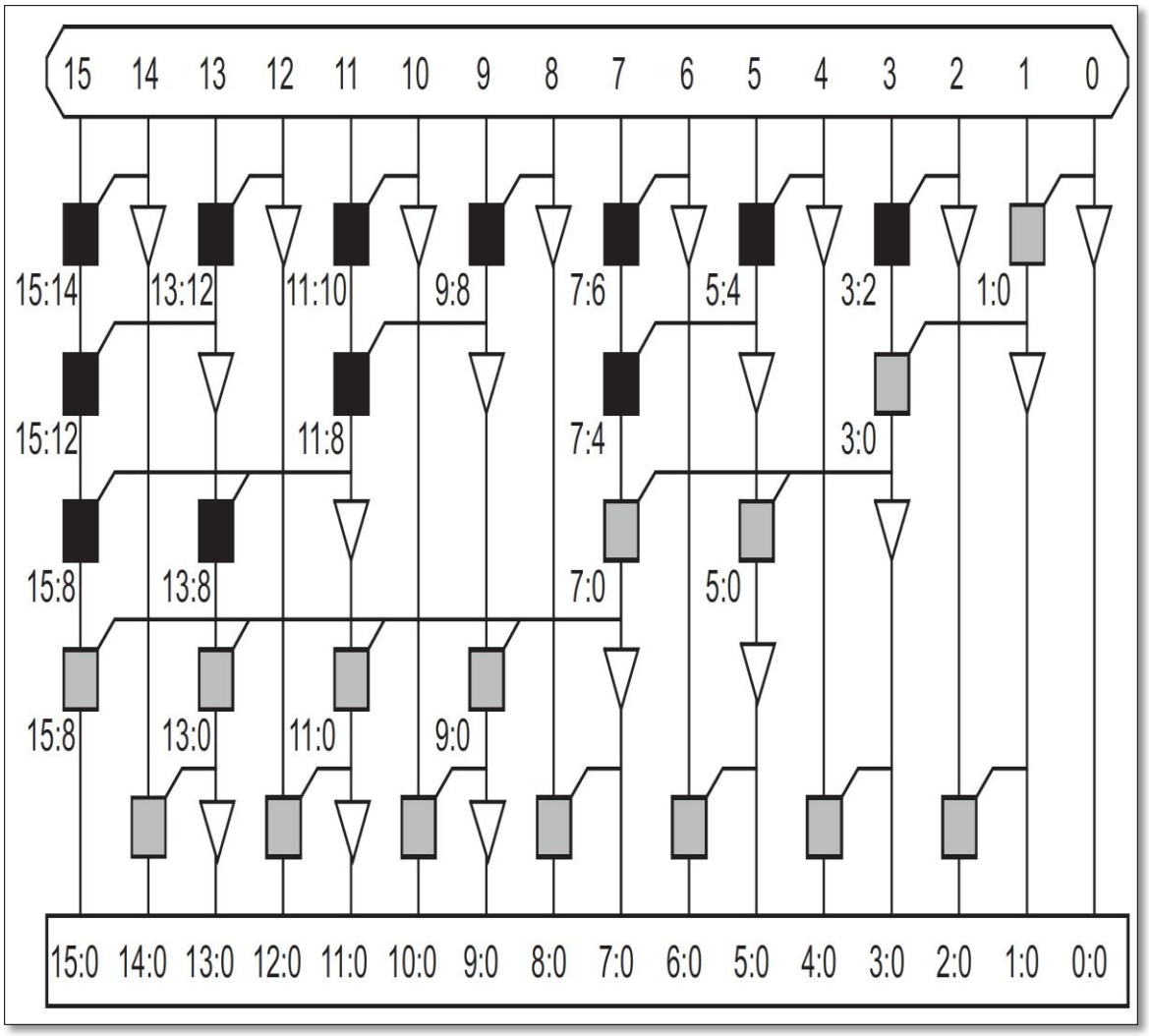

Figure 2.9 16-bit Ladner Fischer Adder[1]

2.9 Comparison of Tree Networks

The ideal tree network would have log_2N levels of logic, fanout of 2 and not more than 1 wiring track between in each row. Brent Kung has many levels. Sklansky has many much more fanouts. Kogge Stone has many wires. In between these extreme cases, Han-Carlson, Land Fishcer and Knowles trees will fill out the design space with different compromises between number of stages, fanout and the wire count.

An advantage of Brent Kung network to the others is that for any given row there is never more than one cell in each pair of the columns. It is low gate count. The length is half as wide reducing

the length of the horizontal wires throughout the adder. This greatly reduces the wire capacitance, which might be the major component of delay in 32-bit, 64-bit and larger adders.

CHAPTER III

METHODOLOGY

3.1 Pass transistor logic

The Pass transistor logic often runs faster, consumes less power and uses less transistors for a particular functional design than the complementary CMOS logic which takes more transistors. There are two main pass transistor logics. The complementary Pass Transistor logic uses only nmos transistors. The Dual Pass transistor logic and dual value logic uses both nmos as well as pmos transistors.

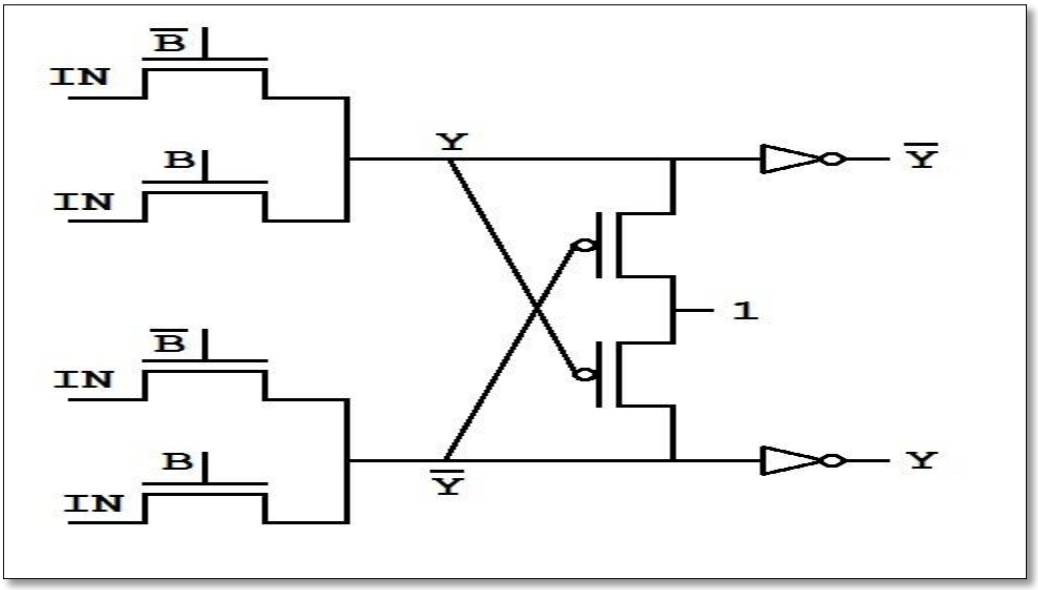

Figure 3.0 Complementary Pass Transistor Logic

3.2 Magic

Magic is design tool which is used to create and modify VLSI layouts. We have various color graphics display which are defined into various metal layers and contacts. We use magic to design basic cells. Then we combine these basic cells hierarchically to make larger structures. The design rules are built-in in Magic. The built-in design-rule-check feature continuously checks for the rule violations when we are building the layouts. Hence we will get to know the error immediately and we can act upon it. The layout is saved in (.mag) format.

The extractor will create a suitable file from the (.mag) file. The suitable file can be a simulation file, spice file etc. for detailed analysis of the layout. We can use various simulators like IRSIM or Spice simulators based on the need.

3.3 IRSIM

IRSIM is a tool which is used to simulate digital circuits. It is a switch level simulator. The transistors are treated as ideal switches by IRSIM. In order to make the switch realistic, lumped resistance and extracted capacitances values are used. MAGIC is used to produce the design while IRSIM is used to read the ".sim" file format.

3.4 32-Bit Extension from 16-Bit Adder

Figure 2.4 shows the diagram of a 16-bit Brent Kung Adder. 16-bit adder will contain 7 stages of operation. 32-bit will contain 9 stages of operation as per the equation $2log_2N - 1$. Now we will add a Gray Cell of 16 bit prefix at 31^{st} bit to the 5^{th} and 8 bit prefix at 23^{rd} bit to the 6^{th} stage. A Black cell of 8 bit prefix will be added to the 5^{th} stage at the 31^{st} bit. The 32-Bit Brent Kung adder is shown in figure 3.1.

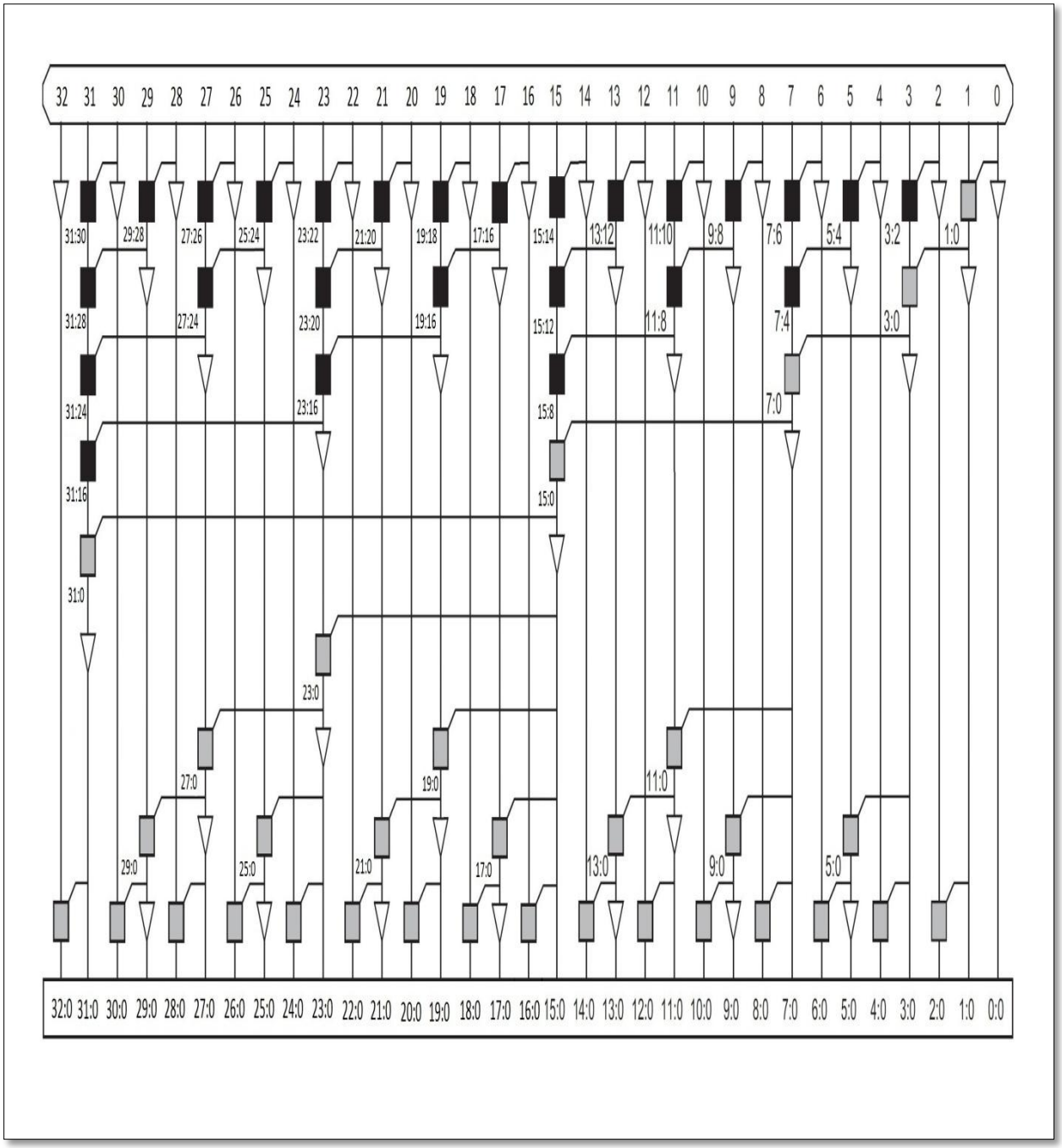

Figure 3.1 32-Bit Brent Kung Adder

CHAPTER IV

IMPLEMENTATION

4.1 Complementary Pass Transistor Logic (CPL)

The use of a normal CMOS technology to implement a logic gate will normally result in a huge delay as each gate is implemented in terms of a universal gate and then the NOT gate. This is basically because CMOS follows an inverted logic. For example, AND is implemented using NAND gate and a NOT gate. The inputs are given to NAND gate and the output of NAND gate is given as input to the NOT gate. Hence we obtain the AND gate.

The use of CPL reduces the delay of the circuit to a great extent. The number of transistors increases slightly but is negligent compared to the increase in the speed of the circuit. This is greatly because of the use of NMOS to realize the basic gates as well as the EXOR gate. The size of the circuit in terms of the transistors remains the same irrespective of whether it is a AND, OR and EXOR gate.

4.2 Implementation of CPL

The implementation of the different gates is done in parts. The basic equation which represents the function executed by CPL is as given in figure 3.2.

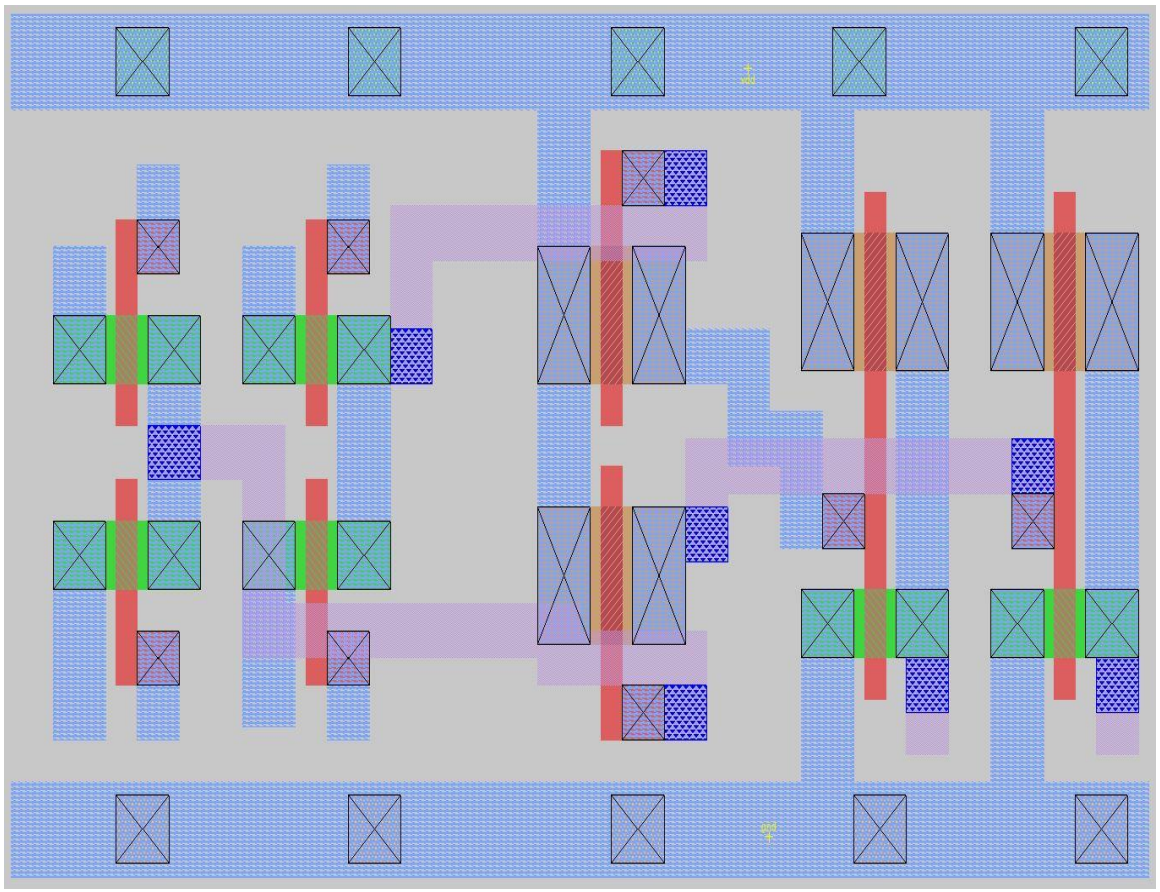

Figure 3.2 Layout of Complementary Pass Transistor Logic (Source : MAGIC)

Now using this equation we will obtain the different equations corresponding to AND, OR and EXOR gate.

4.3 AND Gate

The equation representing the AND gate using CPL is as given below.

$$Y = A.B = A.B + 0.\bar{B}$$

$$\bar{Y} = \bar{A} + \bar{B} = \bar{A}.B + 1.\bar{B}$$

The schematic of an AND gate using CPL is as given in figure 3.3.

The Layout of an AND gate is as given in figure 3.4.

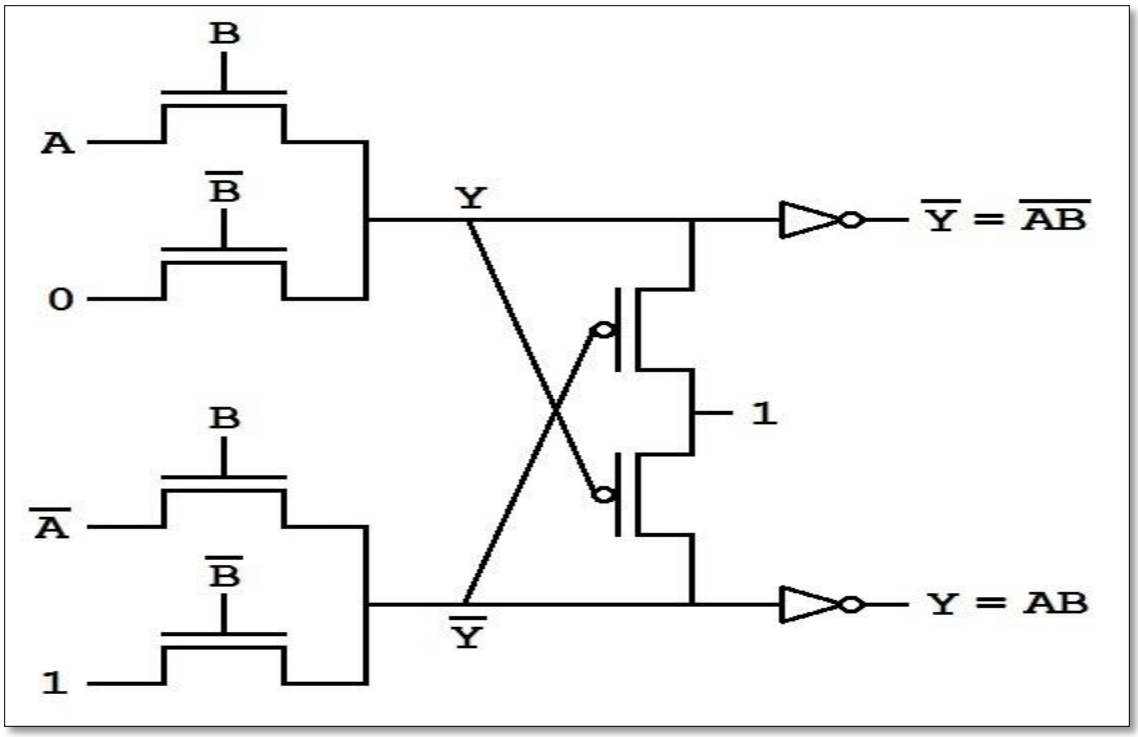

Figure 3.3 Schematic of CPL And gate

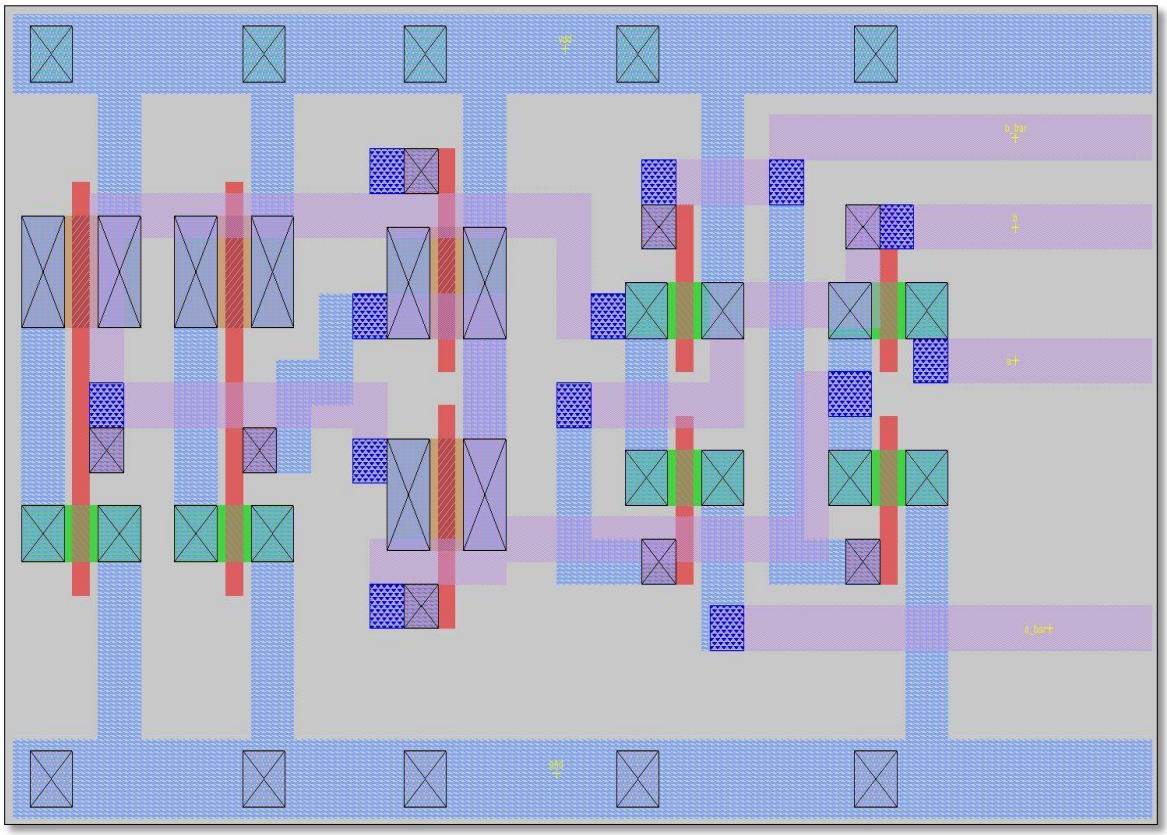

Figure 3.4 Layout of CPL And gate (Source : MAGIC)

4.4 OR Gate

The equation representing the OR gate using CPL is as given below.

$$Y = A + B = 1.B + A.\bar{B}$$

$$\bar{Y} = \bar{A}.\bar{B} = 0.B + \bar{A}.\bar{B}$$

The schematic of an OR gate using CPL is as given in figure 3.5.

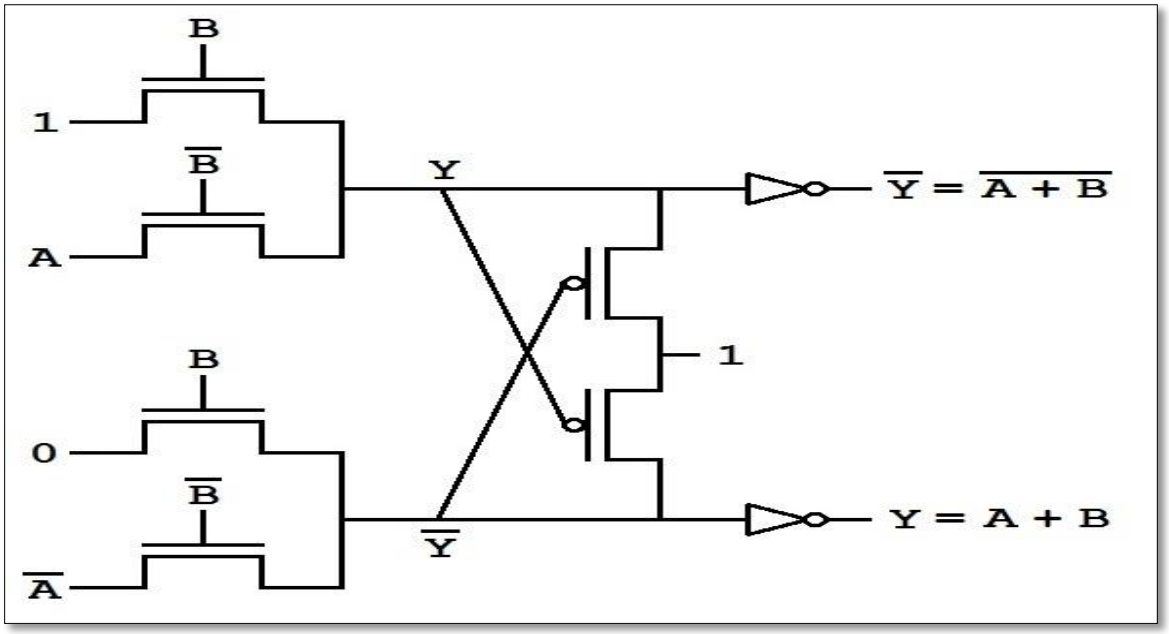

Figure 3.5 Schematic of CPL OR gate

The Layout of an OR gate is as given in figure 3.6.

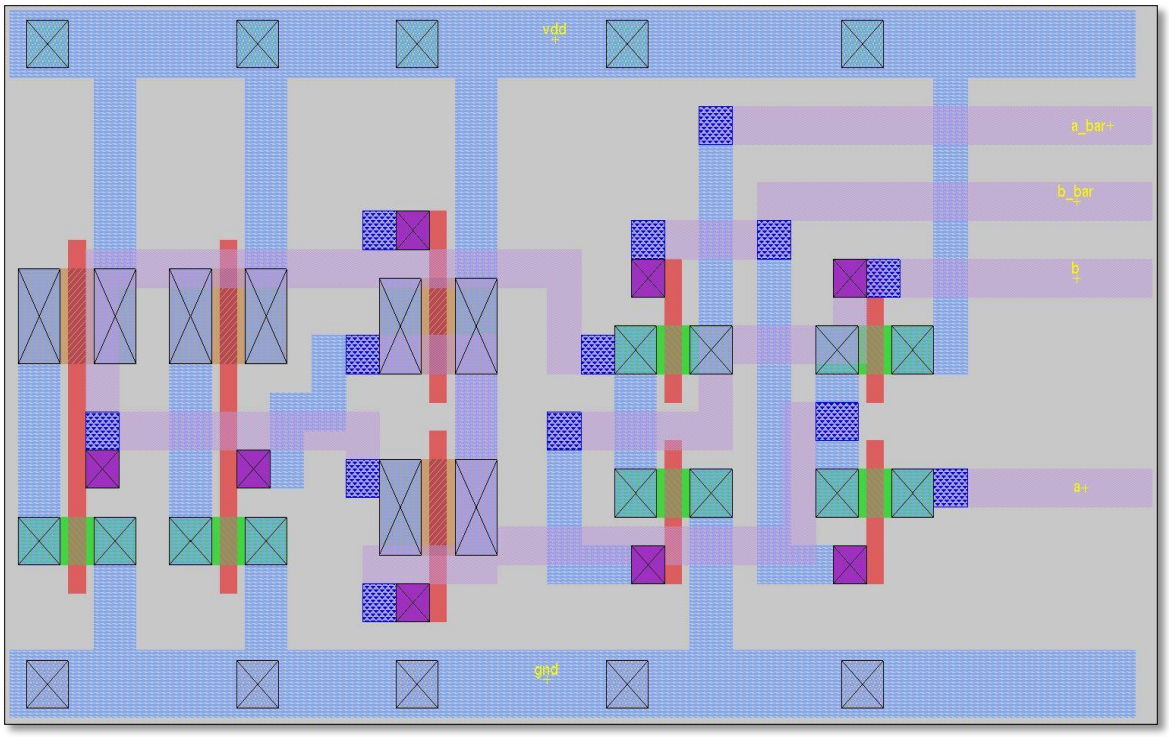

Figure 3.6 Layout of CPL OR gate (Source : MAGIC)

4.5 EXOR Gate

The equation representing the EXOR gate using CPL is as given below.

$$Y = \bar{A}.B + A.\bar{B}$$

$$\bar{Y} = (\bar{A} + B).(A + \bar{B}) = A.B + \bar{A}.\bar{B}$$

The schematic of an EXOR gate using CPL is as given in figure 3.7.

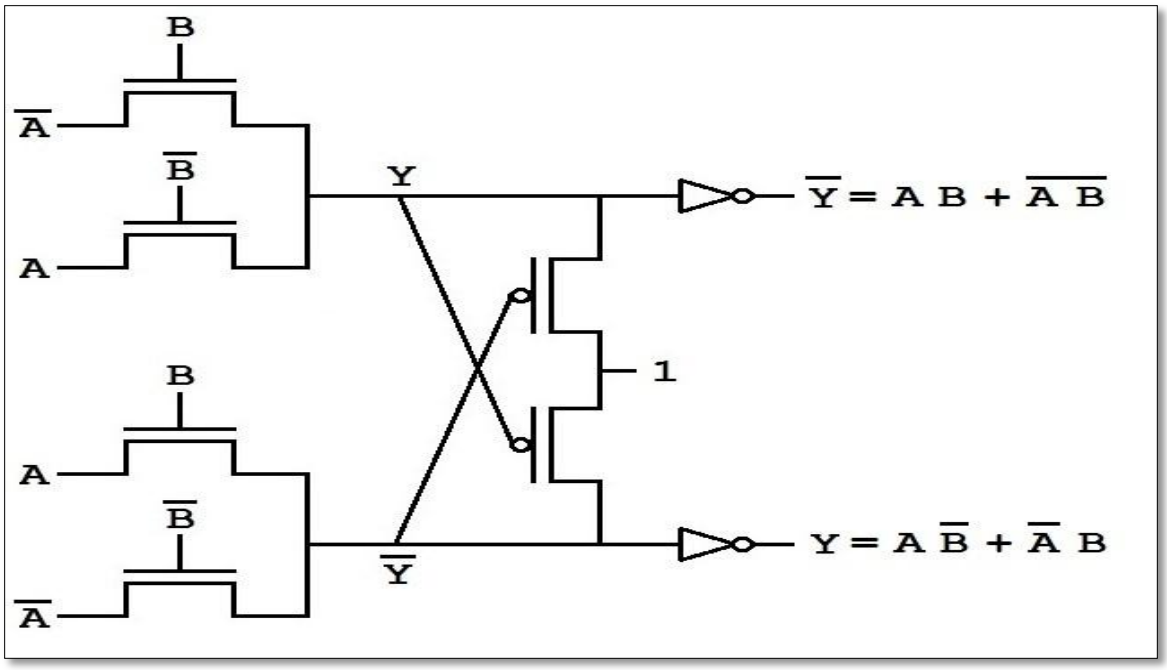

Figure 3.7 Schematic of CPL EXOR gate

The Layout of an EXOR gate is as given figure 3.8.

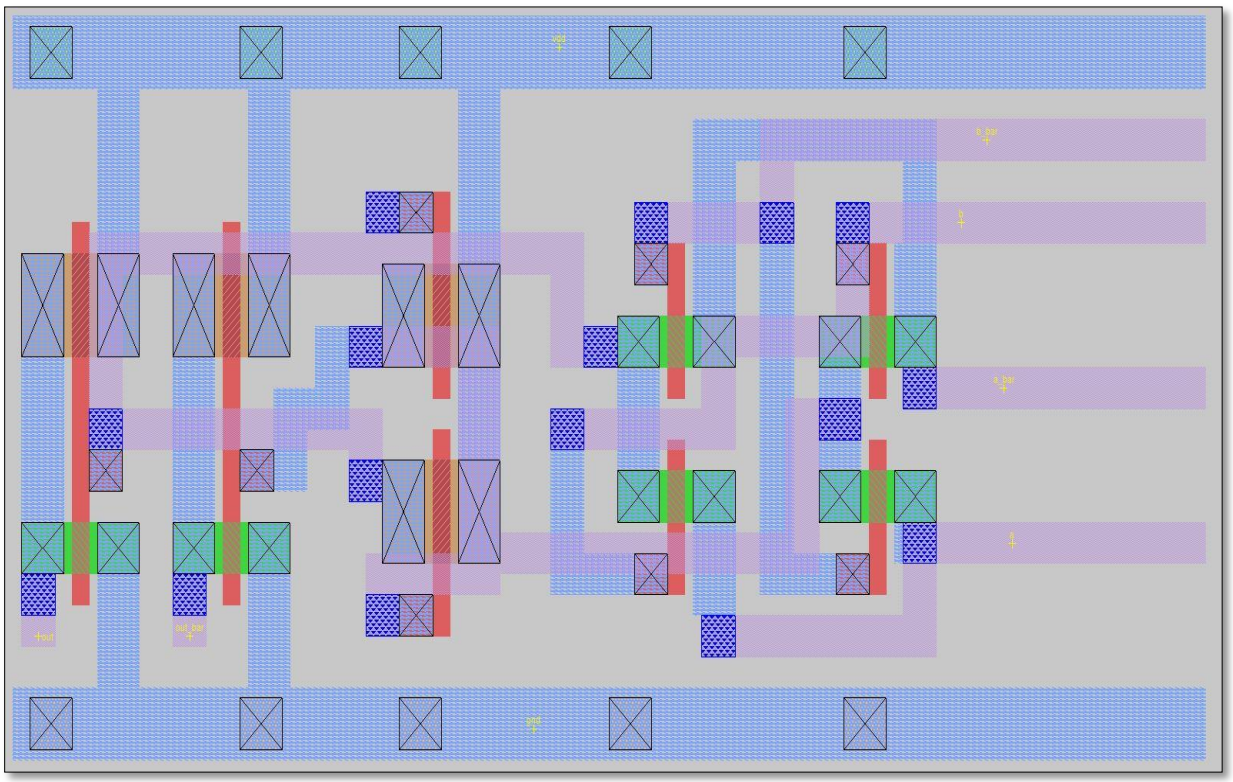

Figure 3.8 Layout of CPL EXOR gate (Source : MAGIC)

4.6 Gray Cell

Gray Cell is the combination of AND-OR gates. The output of AND gate is given as one of the input to the OR gate.

The schematic of a Gray Cell using CPL is as given in figure 3.9.

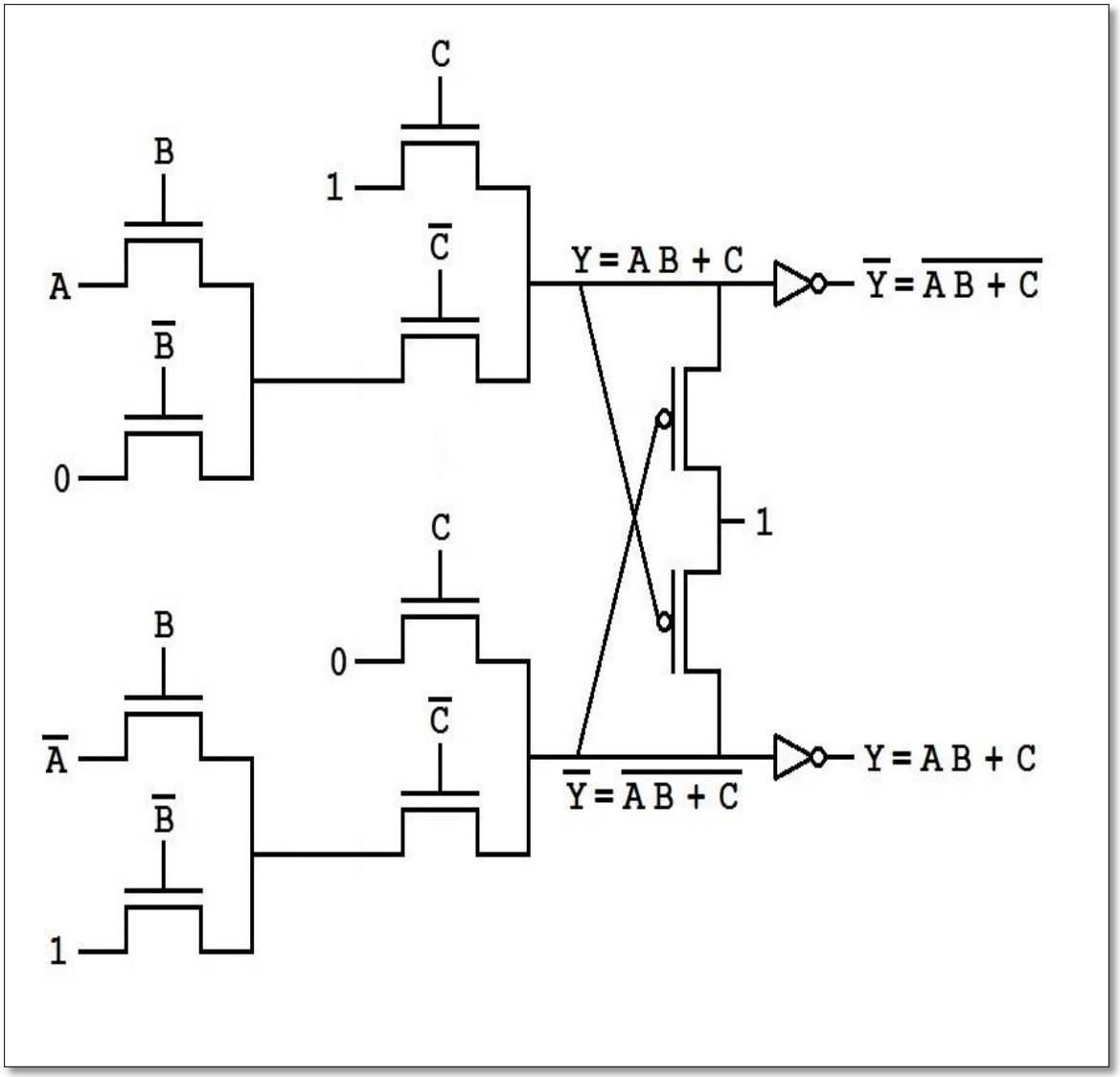

Figure 3.9 Schematic of Gray Cell

The Layout of a Gray Cell is as given in figure 4.0.

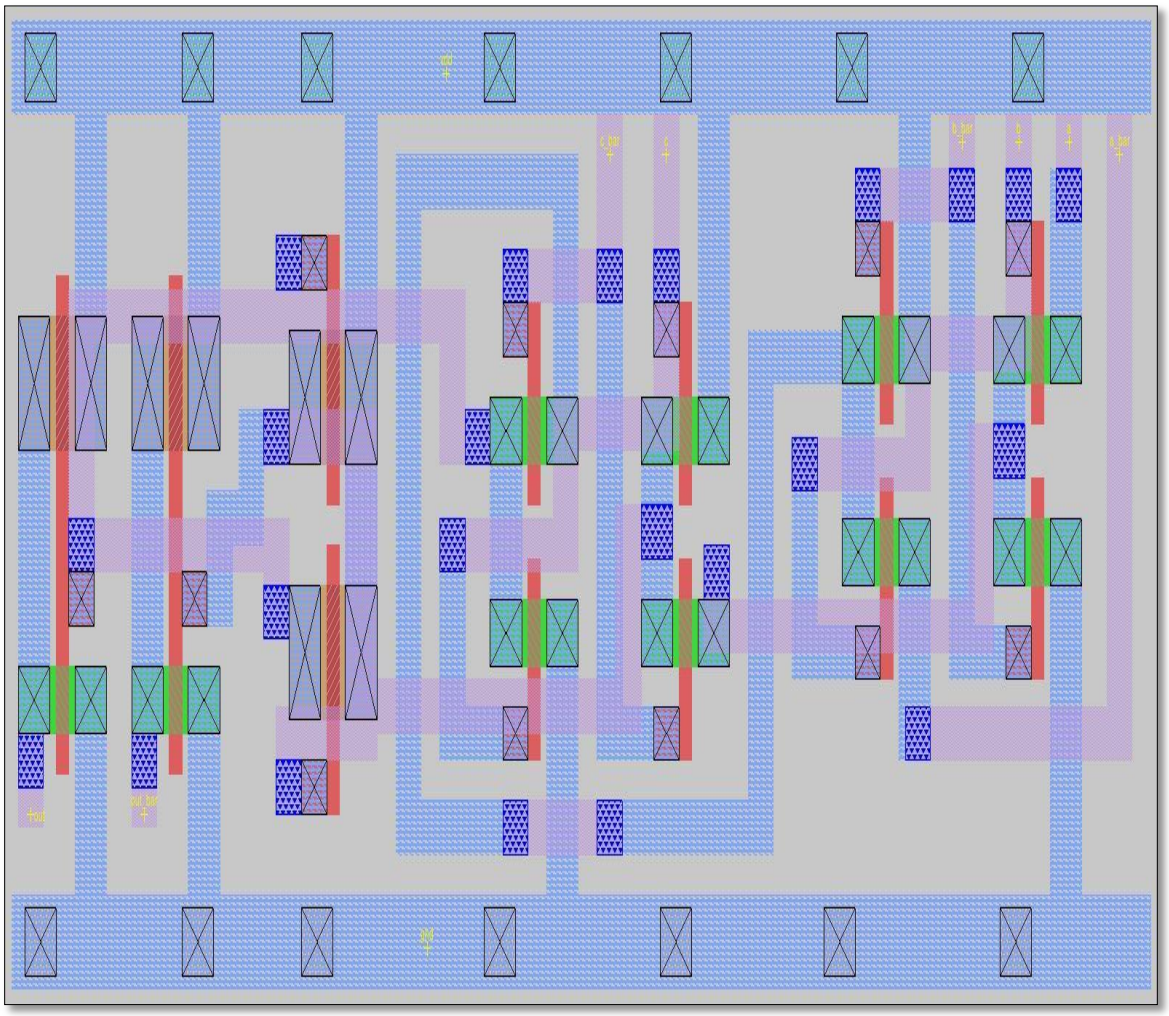

Figure 4.0 Layout of Gray Cell (Source : MAGIC)

4.7 Black Cell

Black Cell is the combination of Gray cell and an AND gate.

The Layout of a Black Cell is as given in figure 4.1.

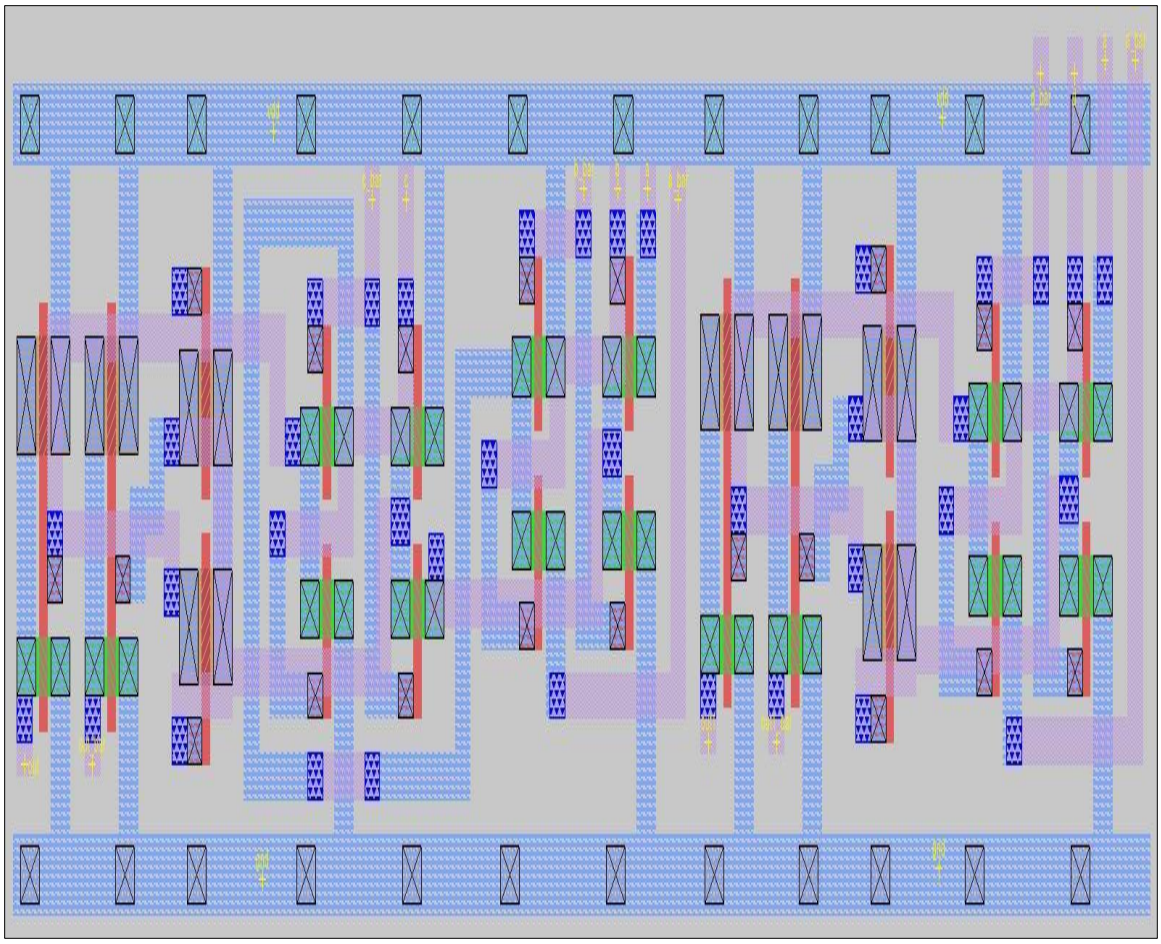

Figure 4.1 Layout of Black Cell (Source : MAGIC)

4.8 Buffer

The buffer is the basic NOT gate, connected in series, which we use in the normal electronic circuit. Apart from performing the Inversion operation the NOT gate also helps in boosting the strength of the electronic signal.

The layout of the NOT gate is as shown in figure 4.2.

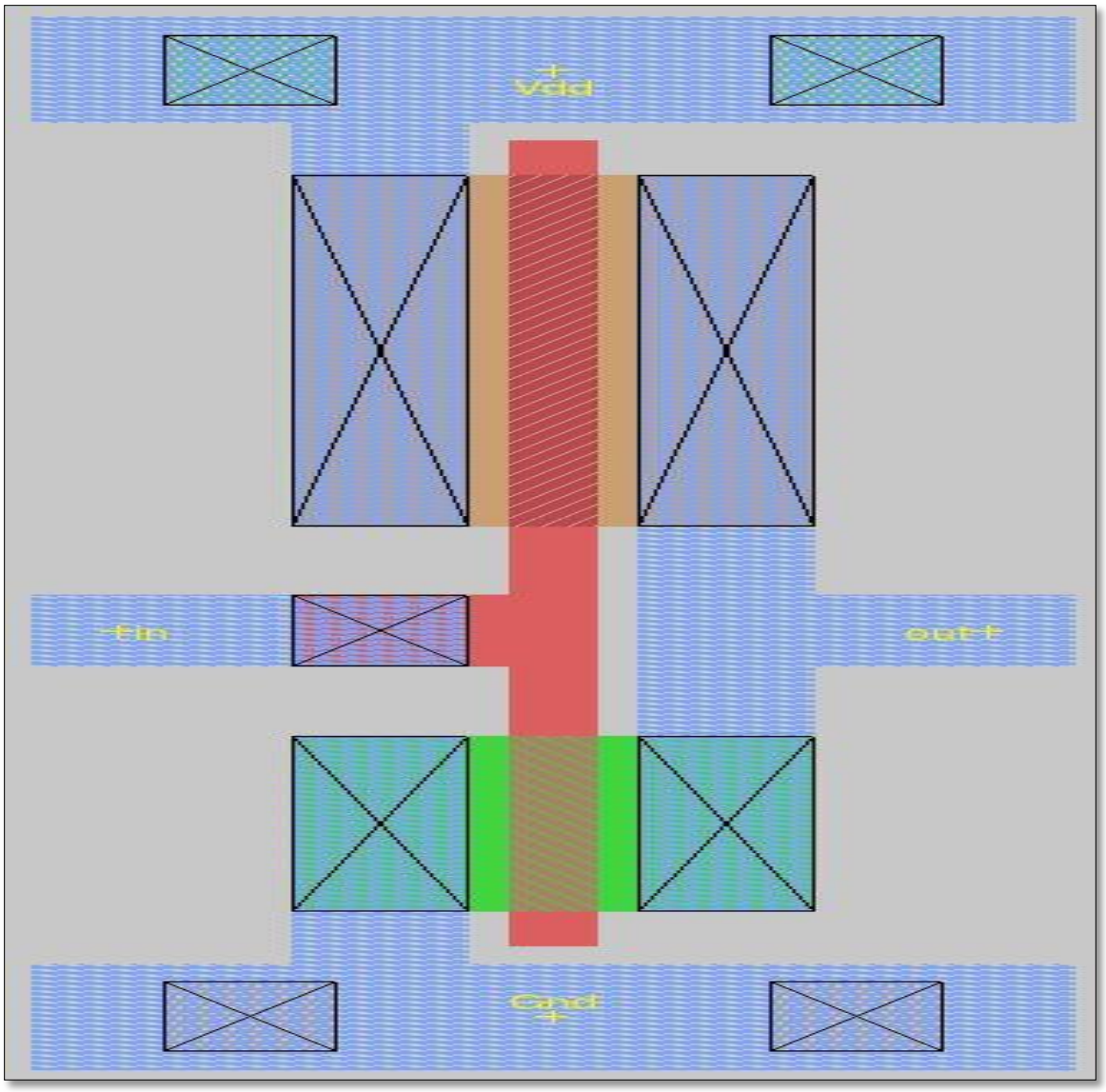

Figure 4.2 Layout of NOT Gate (Source : MAGIC)

4.9 Design Implementation

We will be the linux terminal to open the MAGIC tool. First, open a file using the command *magic <filename>.mag* . The various cells are copied into the design file using the command *getcell <filename>* (Example; *getcell And*, where 'And' implies 'And.mag' which is the magic file for the And gate for the design under consideration). The cells as seen in the previous figures are interconnected as per the 32 bit Brent Kung Adder Design as in figure 3.1. The outputs and the inputs are appropriately conncected using the various metal layers in MAGIC tool. Once all

the interconnections are obtained save the design using the command *save* . The design is checked for design rules by selecting the design layer under consideration and using the command *drc count* . If any errors exist, corrections are made. Then the circuit is extracted using the command *extract all*.

The magic window is closed and using the command *ext2sim <filename>.ext* the simulator file is named *<filename>.sim* is obtained. Now the tool IRSIM is used to simulate the design. Type the command *irsim scmos100.prm <filename>.sim* in the terminal. In the changed command prompt, enter the command *@ <filename>.cmd* to run the script file '<filename>.cmd'. Hence we obtain the final results.

To obtain the total time delay in obtaining the output from the input we can use the command *path inputnode ouputnode*. For example, to find the delay between input 'b1' and output 'sum7' we give the command '*path b1 sum7*'.

CHAPTER V

RESULTS

5.1 IRSIM Output

The 32-Bit Brent Kung Adder implemented using Complementary Pass Transistor Logic contains 2162 transistors. The adder design is tested for different values.

The different inputs and the Theoretical output is as given below.

1) A = 11111111111111111111111111111110 (FFFFFFFE)

   B = 11101010100000000000000000101001 (EA800029)

   Cin = 0

   Cout = 1

   Output = 11101010100000000000000000100111 (EA800027)

2) A = 10101010101010101010101010101010 (AAAAAAAA)

   B = 01010101010101010101010101010101 (55555555)

   Cin = 0

   Cout = 0

   Output = 11111111111111111111111111111111 (FFFFFFFF)

3) A = 11111001111111111111111111101001 (F9FFFFE9)

   B = 00011011010010111101010001010101 (1B4BD455)

Cin = 0

Cout = 1

Output = 00011011010010111101010001010101 (1B4BD455)

4) A = 00001110000000000000100100010110 (0E000916)

   B = 00000011011001001010101110111010 (0364ABBA)

   Cin = 0

   Cout = 0

   Output = 00010001011001001011010011010000 (1164B4D0)

5) A = 11111111111111111111111111111111 (FFFFFFFF)

   B = 11111111111111111111111111111101 (FFFFFFFD)

   Cin = 0

   Cout = 1

   Output = 11111111111111111111111111111101 (FFFFFFFD)

6) A = 11111111111111111111111111111111 (FFFFFFFF)

   B = 11111111111111111111111111111111 (FFFFFFFF)

   Cin = 0

   Cout = 1

   Output = 11111111111111111111111111111110 (FFFFFFFE)

7) A = 11111111111111111111111111111111 (FFFFFFFF)

   B = 11111111111111111111111111111111 (FFFFFFFF)

   Cin = 1

   Cout = 1

   Output = 11111111111111111111111111111111 (FFFFFFFF)

The output for different values of inputs are as given in figure 4.3. The signal G32:0 is the Carry of the 32 bit stage which is nothing but the Output Carry.

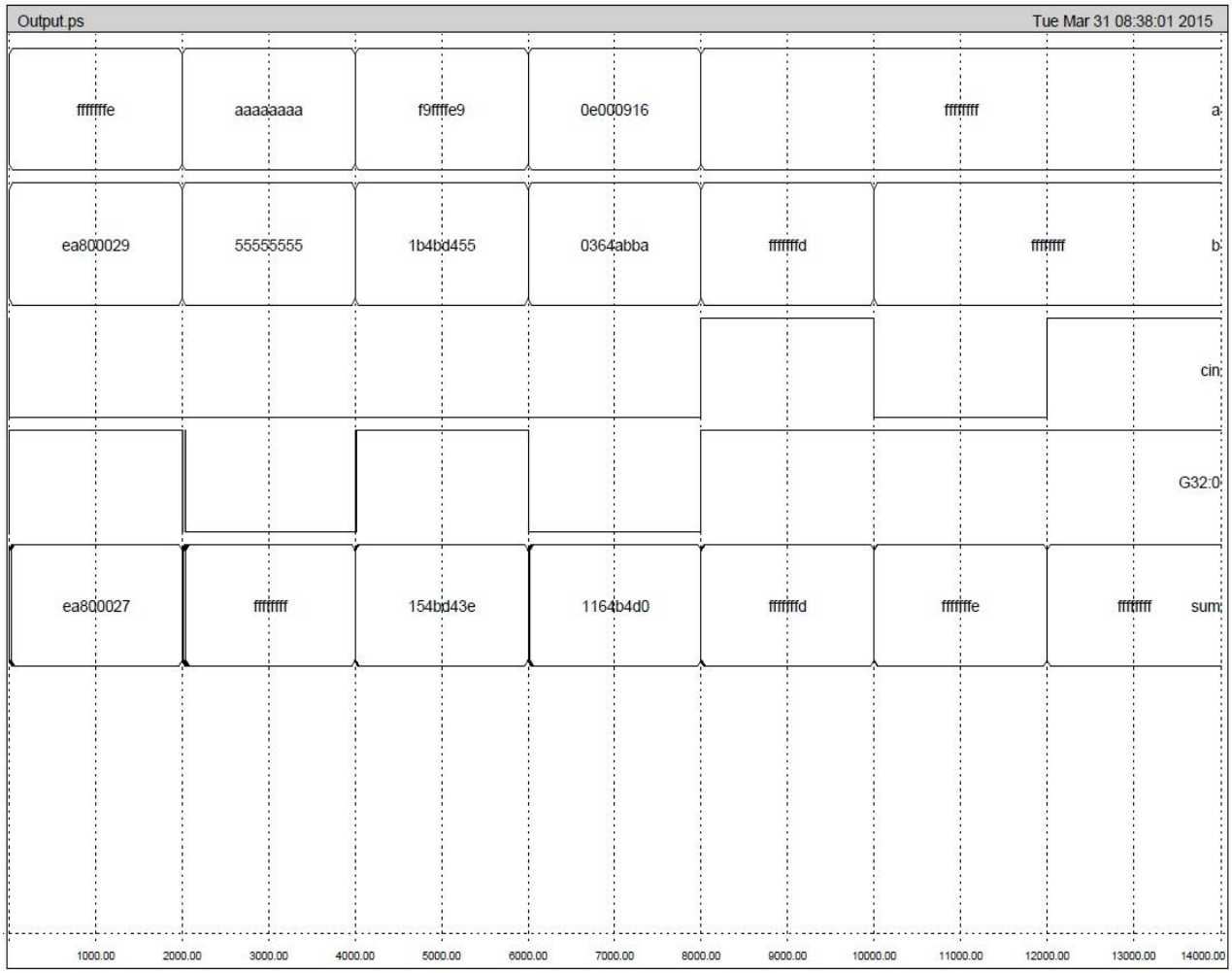

Figure 4.3 IRSIM Output (Source : IRSIM)

The theoretical and IRSIM outputs match, thus proving proper and efficient working of the design.

5.2 Critical Path

The critical path delay for the circuit is calculated using the vectors

a = 11111111111111111111111111111111

b = 00000000000000000000000000000001

cin = 0

The critical path for the 32-Bit Brent Kung Adder is as shown in figure 4.4.

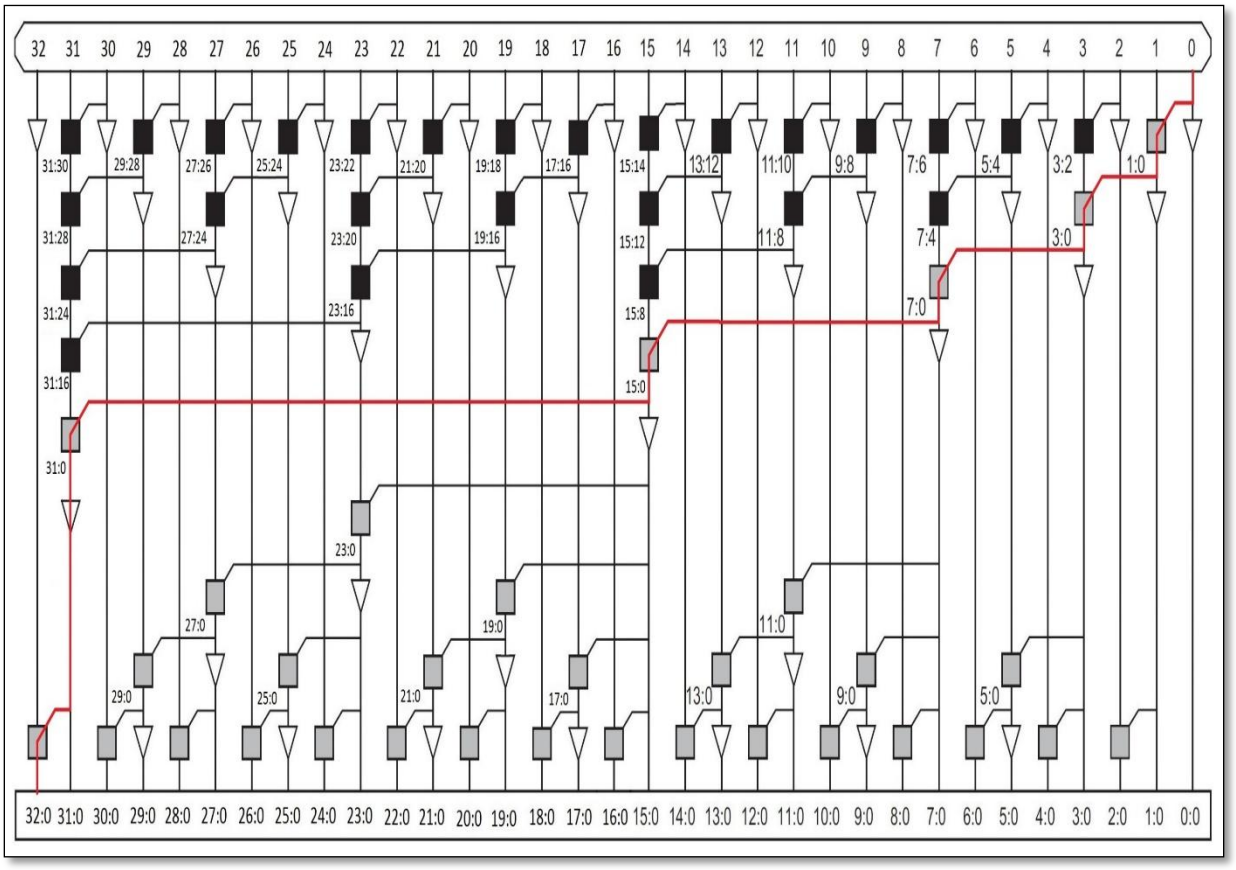

Figure 4.4 Critical Path of 32-Bit Brent Kung Adder

The output of IRSIM for a critical path of a 32-Bit Brent Kung Adder using CMOS logic and CPL is as given in figure 4.5 and figure 4.6 respectively.

The path starts from the a1 and follows the path as the Gray Cell producing the G1:0 signal to the Gray Cell producing the G3:0 signal. Then it follows the Gray Cell producing the G7:0 signal to the Gray Cell producing the G15:0 signal to the Gray Cell producing the G31:0 signal. Finally it moves into the Exor gate producing the Sum32 bit.

```
critical path for last transition of a1:
  a1 -> 1 @ 0.000ns , node was an input
critical path for last transition of Sum32:
  a3 -> 1 @ 0.000ns , node was an input
  And-Exor_2/a_n7_n2# -> 0 @ 0.161ns    (0.161ns)
  P3 -> 1 @ 1.575ns    (1.414ns)
  Black_0/a_88_34# -> 0 @ 2.328ns    (0.753ns)
  P3:2 -> 1 @ 2.615ns    (0.287ns)
  Grey_2/a_13_53# -> 0 @ 2.980ns    (0.365ns)
  Grey_2/a_40_9# -> 1 @ 3.267ns    (0.287ns)
  Grey_2/a_58_9# -> 0 @ 3.501ns    (0.234ns)
  G3:0 -> 1 @ 4.435ns    (0.934ns)
  Grey_6/a_13_53# -> 0 @ 4.934ns    (0.499ns)
  Grey_6/a_40_9# -> 1 @ 5.221ns    (0.287ns)
  Grey_6/a_58_9# -> 0 @ 5.455ns    (0.234ns)
  G7:0 -> 1 @ 6.561ns    (1.106ns)
  Grey_14/a_13_53# -> 0 @ 7.088ns    (0.527ns)
  Grey_14/a_40_9# -> 1 @ 7.375ns    (0.287ns)
  Grey_14/a_58_9# -> 0 @ 7.609ns    (0.234ns)
  G15:0 -> 1 @ 8.888ns    (1.279ns)
  Grey_30/a_13_53# -> 0 @ 9.442ns    (0.554ns)
  Grey_30/a_40_9# -> 1 @ 9.729ns    (0.287ns)
  Grey_30/a_58_9# -> 0 @ 9.963ns    (0.234ns)
  G31:0 -> 1 @ 10.551ns    (0.588ns)
  Sum32 -> 0 @ 10.560ns    (0.009ns)
```

Figure 4.5 Critical Path Output using CMOS Technology (Source : IRSIM)

```
critical path for last transition of a1:
  a1 -> 1 @ 0.000ns , node was an input
critical path for last transition of sum32:
  b3 -> 0 @ 0.000ns , node was an input
  And-Exor_2/a_383_188# -> 1 @ 0.229ns    (0.229ns)
  And-Exor_2/a_455_155# -> 1 @ 0.872ns    (0.643ns)
  And-Exor_2/a_437_155# -> 0 @ 1.411ns    (0.539ns)
  P3 -> 1 @ 1.954ns    (0.543ns)
  Black_0/a_n19_15# -> 1 @ 3.455ns    (1.501ns)
  Black_0/a_n1_15# -> 0 @ 4.264ns    (0.809ns)
  G3:2_bar -> 1 @ 4.582ns    (0.318ns)
  Grey_2/a_39_88# -> 1 @ 6.062ns    (1.480ns)
  Grey_2/a_21_88# -> 0 @ 6.871ns    (0.809ns)
  Exor_3/a_24_4# -> 1 @ 8.082ns    (1.211ns)
  G3:0_bar -> 0 @ 8.999ns    (0.917ns)
  Grey_6/a_39_88# -> 1 @ 10.468ns    (1.469ns)
  Grey_6/a_21_88# -> 0 @ 11.277ns    (0.809ns)
  Grey_7/a_39_88# -> 1 @ 13.062ns    (1.785ns)
  Grey_14/a_21_88# -> 0 @ 14.805ns    (1.743ns)
  Grey_15/a_39_88# -> 1 @ 16.738ns    (1.933ns)
  G15:0_bar -> 0 @ 18.065ns    (1.327ns)
  Grey_30/a_39_88# -> 1 @ 19.534ns    (1.469ns)
  Grey_30/a_21_88# -> 0 @ 20.343ns    (0.809ns)
  Exor_31/a_24_4# -> 1 @ 21.420ns    (1.077ns)
  sum32 -> 0 @ 21.427ns    (0.007ns)
```

Figure 4.6 Critical Path Output using CPL (Source : IRSIM)

5.3 Comparison

The 32 Bit Brent Kung Adder design is compared for the two implementations. i.e. CPL and CMOS. The parameters for which the comparison is done is shown in table 3.

	CPL	**CMOS**
Operating Voltage	5 V	5 V
Technology	0.180 micron	0.180 micron
NMOS and PMOS Widths	5λ for NMOS, 10λ for PMOS	5λ for NMOS, 10λ for PMOS
No. of Transistors	2162	1812
Critical Path delay	21.427 ns	10.560 ns

Table 3. Comparison of CPL and CMOS designs

CHAPTER VI

CONCLUSION

In the results section, we see that the design using CPL has a more delay compared to the design using CMOS. Also, the number of transistors are also more. This is primarily because in CPL the has additional two NOT gates in its design. If we remove these NOT gates and use the buffers at only the required slots then we can have excessively less number of transistors and the delay will also be considerably reduced. The use of NOT gate is to increase the signal strength at each level. Thus by only using the buffers even more wisely, the performance of the design can be noticeably increased.

The main advantage of using the CPL's is that we will be using less number of transistors to implement the logical functions as compared to CMOS. This can be evidently seen in the design of the EXOR where the CPL design takes ten transistors and CMOS design takes twelve transistors. The CPL also has very less power consumption compared to CMOS.

Hence CPL will significantly reduce the power consumption of the circuit. Also the circuit delay and area of the layout will be reduced by more than fifty percent. Also experimenting with the different types of Pass Transistor logics is also good option as it will help in considering more options for the design.

6.2 Future Work

The design can be further enhanced for 64 bit as well 128 bit. Even other design variants can also be tried for even better analysis. In fact by combining the various tree adders as well as the technology used to implement them, a very suitable adder with significant less delay can be achieved.

As we have used Complementary Pass Logic to design the adder, we can even design the adder using LEAP technology, Double Pass Transistor logic and many more technologies. Using these technologies for the design may further increase the speed of the Brent Kung adder.

REFERENCES

[1] Neil H.E. Weste and David Money Harris, "CMOS VLSI Design: A circuits and Systems Perspective", 4th Ed: Addition-Wesley, 2004.

[2] Richard P. Brent and H.T. Kung, "*A Regular Layout for Parallel Adders*", IEEE Transactions on Computers Volume 31 Issue 3, March 1982, Pages 260-264.

[3] J. Grad and J.E. Stine, "*A Standard Cell Library for Student Projects*", International Conference on Microelectronics Systems Education, pages 98–99. IEEE Computer Society Press 2003, 2003.

[4] Kogge P and Stone H, "*A Parallel Algorithm for the Efficient Solutions of a General Class of Recurrence Relations*", IEEE Transactions on Computers, Vol. C-22, No.8, 1973.

[5] S. Knowles, "*A family of adders*", Proc. 15th IEEE Symp. Comp. Arith., June 2001, Pages. 277-281.

[6] Han, Carlson, "*Fast Area-Efficient VLSI Adders*", IEEE, 1987.

[7] G. Dimitrakopoulos and D. Nikolos, "*High-speed parallel-prefix VLSI ling adders*", IEEE Trans. Comput., vol. 54, no. 2, Feb.2005, Pages. 225–231.

[8] V. Dave, E. Oruklu, and J. Saniie, "Performance evaluation of flagged prefix adders for constant addition," in Proc. IEEE Int. Conf. Electro/ inf. Technol., 2006, Pages 415–420.

[9] Geeta Rani , Sachin Kumar, "*Delay Analysis of Parallel-Prefix Adders*", International Journal of Science and Research, Volume 3 Issue 6, June 2014, Pages 2339-2342.

APPENDICES

List of Commands used in Magic:-

- *magic* <filename>.mag - Creates a layout file by the name <filename> or loads the file <filename>.mag
- *paint* layers|cursor - paints layer types in the current edit cell inside the area of the cursor box
- *save* <filename> - saves the current edit cell to disk. If *filename* is specified, then the layout will be saved to "*filename*.mag", and the current edit cell will be renamed to *filename*.
- *drc* <option> - check -Recheck area under box in all cells
  count [total] - Count error tiles in each cell under box.
  Option total returns the total number of DRC error tiles in the cell.
- *extract* <option> - all - Extract the root cell and all its children. This bypasses the incremental extraction and ensures that a new .ext file is written for every cell definition.
  cell *name*- Extract the indicated cell into file *name*
- *extract* <option> - all - Extract the root cell and all its children. This bypasses the incremental extraction and ensures that a new .ext file is written for every cell definition.
  cell *name*- Extract the indicated cell into file *name*

- *ext2sim* <filename>.ext - convert extracted file(s) to a ".sim" format file

List of commands used in IRSIM:-

- *irsim* <prm_file>.prm <filename>.sim - prm_file - The electrical parameters file that configure the devices to be simulated
  filename - All file names *not* beginning with a '-' are assumed to be sim (netlist) files.
- *path* <input-node> <output-node> - Displays the critical path between the given input node and output node

Contents of IRSIM command file:-

###

stepsize 2000.0

vector a a32 a31 a30 a29 a28 a27 a26 a25 a24 a23 a22 a21 a20 a19 a18 a17 a16 a15 a14 a13 a12 a11 a10 a9 a8 a7 a6 a5 a4 a3 a2 a1

vector b b32 b31 b30 b29 b28 b27 b26 b25 b24 b23 b22 b21 b20 b19 b18 b17 b16 b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1

vector sum sum32 sum31 sum30 sum29 sum28 sum27 sum26 sum25 sum24 sum23 sum22 sum21 sum20 sum19 sum18 sum17 sum16 sum15 sum14 sum13 sum12 sum11 sum10 sum9 sum8 sum7 sum6 sum5 sum4 sum3 sum2 sum1

ana -h a b cin G32:0 sum

analyzer

| addition

set a 11111111111111111111111111111110

set b 11101010100000000000000000101001

l cin

c

s

set a 10101010101010101010101010101010

set b 01010101010101010101010101010101

l cin

c

s

set a 11111001111111111111111111101001

set b 00011011010010111101010001010101

l cin

c

s

set a 00001110000000000000100100010110

set b 00000011011001001010101110111010

l cin

c

s

set a 11111111111111111111111111111111

set b 11111111111111111111111111111101

h cin

c

s

set a 11111111111111111111111111111111

set b 11111111111111111111111111111111

l cin

c

s

set a 11111111111111111111111111111111

set b 11111111111111111111111111111111

h cin

c

s

##

VITA

Noel Daniel Gundi

Candidate for the Degree of

Master of Science

Thesis: IMPLEMENTATION OF 32 BIT BRENT KUNG ADDER USING COMPLEMENTARY PASS TRANSISTOR LOGIC

Major Field: ELECTRICAL AND COMPUTER ENGINEERING

Biographical:

Education:

Completed the requirements for the Master of Science in Electrical and Computer Engineering at Oklahoma State University, Stillwater, Oklahoma in May, 2015.

Completed the requirements for the Bachelor of Engineering in Electronics and Communication at Visvesvaraya Technological University, Belgaum, Karnataka, India in June 2008.

Experience: Senior Engineer at Tata Elxsi Ltd.
Computer Assistant in IT Labs at Oklahoma State University
Tutor in LASSO Tutoring at Oklahoma State University

Professional Memberships: IEEE, Phi Kappa Phi, NSBE