A COMPARATIVE STUDY OF HIGH SPEED ADDERS

A Thesis Presented to
The Faculty of the
Fritz J. and Dolores H. Russ
College of Engineering and Technology
Ohio University
In Partial Fulfillment
of the Requirement for the Degree
Master of Science
by
Raja D.V. Bhupatiraju
March, 1999
Acknowledgments

I would like to thank my advisor, Dr. Janusz Starzyk for his guidance and support throughout the entire process towards the successful completion of this thesis. I probably would not have completed this thesis without his able guidance.
Contents

List of Tables iii
List of Figures iv

CHAPTER 1. Introduction 1
1.1 Tools used 2
1.1.1 Mentor Graphics 3
1.1.2 Xilinx Foundation Series 4
1.2 Organization of the thesis 4

CHAPTER 2. Adder Architectures 6
2.1 Ripple Carry Adder 6
2.2 Carry Bypass Adder 7
2.3 Carry Select Adder 10
2.4 Carry Look-Ahead Adder 12

CHAPTER 3. Logarithmic Adder 16
3.1 Introduction 16
3.2 Brent Kung Architecture 17
3.2.1 Generate/Propagate Generation 17
3.2.2 The Dot Operation 17
3.2.3 Sum Generation 21

CHAPTER 4. Design of Conditional Sum Adder 25
4.1 Conditional Sum Algorithm 25
4.2 Architecture of Conditional Sum Adder 27
4.2.1 Design of Full Adder Cell 30
4.2.2 Design of Selector 35
4.2.3 Multiplexer Design 39
4.3 Basic Structure Design 42
4.4 Comparison between Brent Kung and Conditional Sum Adder 55

CHAPTER 5. FPGA Implementation 58
5.1 Introduction to FPGA’s 58
5.2 Architecture 60
5.2.1 Configurable Logic Blocks 62
5.2.2 Input/Output Blocks 62
5.2.3 Programmable Interconnects 63
5.3 Design Implementation in FPGA’s 63

CHAPTER 6. Conclusion 73
References 76
### List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Example to illustrate the concept of conditional sum</td>
<td>26</td>
</tr>
<tr>
<td>4.2</td>
<td>Selector functionality</td>
<td>36</td>
</tr>
<tr>
<td>4.3</td>
<td>Multiplexer functionality</td>
<td>39</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Ripple Carry adder</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>4-bit carry bypass structure</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>16-bit carry bypass adder</td>
<td>9</td>
</tr>
<tr>
<td>2.4</td>
<td>Carry select module topology</td>
<td>10</td>
</tr>
<tr>
<td>2.5</td>
<td>16-bit carry select adder</td>
<td>11</td>
</tr>
<tr>
<td>2.6</td>
<td>Carry look-ahead adder concept</td>
<td>12</td>
</tr>
<tr>
<td>3.1</td>
<td>Design of dot operator</td>
<td>18</td>
</tr>
<tr>
<td>3.2</td>
<td>Layout of dot operator</td>
<td>19</td>
</tr>
<tr>
<td>3.3</td>
<td>8-bit Brent Kung structure</td>
<td>21</td>
</tr>
<tr>
<td>3.4</td>
<td>16-bit Brent Kung structure</td>
<td>22</td>
</tr>
<tr>
<td>3.5</td>
<td>Layout of 16-bit Brent Kung structure</td>
<td>24</td>
</tr>
<tr>
<td>4.1</td>
<td>Block diagram of 4-bit conditional sum adder</td>
<td>29</td>
</tr>
<tr>
<td>4.2</td>
<td>Transistor level design of full adder</td>
<td>32</td>
</tr>
<tr>
<td>4.3</td>
<td>Layout of full adder</td>
<td>33</td>
</tr>
<tr>
<td>4.4</td>
<td>Simulation results for full adder</td>
<td>34</td>
</tr>
<tr>
<td>4.5</td>
<td>Transistor level design of selector</td>
<td>37</td>
</tr>
<tr>
<td>4.6</td>
<td>Selector layout</td>
<td>38</td>
</tr>
<tr>
<td>4.7</td>
<td>Transistor level design of multiplexer</td>
<td>40</td>
</tr>
<tr>
<td>4.8</td>
<td>Layout of multiplexer</td>
<td>41</td>
</tr>
<tr>
<td>4.9</td>
<td>One bit conditional sum adder</td>
<td>42</td>
</tr>
<tr>
<td>4.10</td>
<td>Block level design of 2-bit adder</td>
<td>45</td>
</tr>
<tr>
<td>4.11</td>
<td>Layout of 2-bit adder</td>
<td>46</td>
</tr>
<tr>
<td>4.12</td>
<td>Simulation results for 2-bit adder</td>
<td>47</td>
</tr>
<tr>
<td>4.13</td>
<td>Block level design of 4-bit adder</td>
<td>48</td>
</tr>
<tr>
<td>4.14</td>
<td>Layout of 4-bit adder</td>
<td>49</td>
</tr>
<tr>
<td>4.15</td>
<td>Simulation results of 4-bit adder</td>
<td>50</td>
</tr>
<tr>
<td>4.16</td>
<td>Transistor level design of buffer</td>
<td>51</td>
</tr>
<tr>
<td>4.17</td>
<td>Layout of buffer</td>
<td>52</td>
</tr>
<tr>
<td>4.18</td>
<td>Block level design of 32-bit adder</td>
<td>53</td>
</tr>
<tr>
<td>4.19</td>
<td>Layout of 32-bit adder</td>
<td>54</td>
</tr>
<tr>
<td>4.20</td>
<td>Simulation results for 16-bit conditional sum adder</td>
<td>56</td>
</tr>
<tr>
<td>4.21</td>
<td>Simulation results for 16-bit Brent Kung adder</td>
<td>57</td>
</tr>
<tr>
<td>5.1</td>
<td>FPGA architecture</td>
<td>61</td>
</tr>
<tr>
<td>5.2</td>
<td>Gate level design of multiplexer</td>
<td>66</td>
</tr>
<tr>
<td>5.3</td>
<td>Gate level design of selector</td>
<td>67</td>
</tr>
<tr>
<td>5.4</td>
<td>Gate level design of full adder</td>
<td>68</td>
</tr>
<tr>
<td>5.5</td>
<td>Design of 8-bit adder</td>
<td>69</td>
</tr>
<tr>
<td>5.6</td>
<td>Design of 16-bit adder</td>
<td>70</td>
</tr>
<tr>
<td>5.7</td>
<td>Design of 24-bit adder</td>
<td>71</td>
</tr>
<tr>
<td>5.8</td>
<td>Design of 32-bit adder</td>
<td>72</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Addition is one of the simplest and commonly used operations and is in most cases a speed determining factor for arithmetic operations. The addition of two binary numbers is the fundamental arithmetic operation in microprocessors, digital signal processors, and data-processing application-specific integrated circuits. Several algorithms have been presented for high speed parallel addition, and there is generally a trade off between speed and area. Hence, binary adders are crucial building blocks in very large-scale integrated circuits.

For the optimization of speed in adders, the most important factor is carry generation. For the implementation of a fast adder, the generated carry should be driven to the output as fast as possible, thereby reducing the worst path delay which determines the ultimate speed of the digital structure. In the design for timing optimization, a network can be optimized either at circuit or logic level. Logic level optimization is done by manipulating boolean equations, whereas, circuit level optimization can be carried out by manipulating transistor sizes and circuit topologies. In the optimization for area, care should be taken in the design of the building blocks of the structure, which determine the area occupied by the architecture and, finally, also effect the speed.

The basic objective in realization of this goal is to present an algorithm for high speed parallel addition in which the computation is performed in $\log_2 n$ operating steps with
being the number of inputs. The algorithm uses the conditional sum concept, which is explained in Chapter 4. A comprehensive overview and a qualitative evaluation of the different existing adder architectures are given in this thesis along with the proposed adder architecture. One particular adder structure known as Brent Kung structure is also studied in depth. It has been determined that compared with other adders, such as the ripple-carry and carry-look ahead, the proposed conditional sum adder shows the best overall performance characteristics. Transistor level design and layout for the proposed conditional sum adder and the Brent-Kung structure are presented and discussed in depth along with the timing and simulation results. The overall goal of this thesis is to present a comparative study between high speed digital adders and discuss their implementation in VLSI and FPGA technologies.

1.1 Tools used

For the purpose of design, simulation, analysis and a programmable gate array implementation, the tools used in this thesis are:

1. Mentor Graphics
2. Xilinx Foundation Series

These tools are installed in the VLSI lab on SUN workstations and in FPGA lab on PC machines. Tools are industry standards for VLSI design and for XILINX FPGA design.
1.1.1 Mentor Graphics

Mentor Graphics tool has a number of toolsets among which design architect, IC station, and accusim are widely used in this thesis. Design architect is a schematic design software, which is used for all transistor level designs in this thesis. The schematic design once made is checked for any errors and rectified if necessary. IC station is the tool used to make a complete layout of the transistor level design. During this process, the layout is examined for design rule violations using design rule check (DRC), which basically checks the required minimum dimensions of the interconnects and spacing. Once the layout is made and it passes the DRC, we know that the design has been made according to the rules, but this does not tell if the layout is in accordance with the schematic design. To check for this, a program known as the LVS (Layout Versus Schematic) check is used. This program compares the layout and schematic for each device and net in the design and provides a comparative report of the layout versus schematic.

Finally, to check the functionality of the designed circuit, the circuit needs to be simulated. Simulation of the circuit before and after the layout is carried out by accusim which is a simulation tool. Accusim takes in the forced input values and provides a corresponding output values. The timing analysis is achieved using accusim. It is used to analyze the worst case delay in VLSI circuits.

1.1.2 Xilinx Foundation Series

Xilinx foundation series software is used in this thesis for the implementation of the circuit in field programmable gate arrays (FPGA), which will be discussed in Chapter
5. In the Xilinx foundation series software, the digital design is created using the schematic design editor, a hardware description language such as ABEL, VHDL or verilog, or a state machine editor. The outputs of these programs produce netlists. This netlist is converted into a bit stream file, which configures the FPGA. The configuration process involves mapping of the design, placement of the logic blocks known as configurable logic blocks which will be discussed in Chapter 5, and routing of the interconnection paths between the logic blocks. The resulting file is a logic cell array file, which is converted into a bit stream file for configuring the FPGA. In the configuration process, the circuit design is downloaded into the FPGA. The FPGA used in this thesis is the Xilinx XC4010XLPC84, which is a member of the Xilinx 4000 series family. The FPGA is configured by a programmable read only memory (PROM), which can write data into the FPGA or from which the FPGA can actively read its configuration data.

1.2 Organization of the thesis

In Chapter 2 of this thesis a description of the various types of existing adders[2] is presented. The architectures of different adders along with their functionality are given.

In Chapter 3 an architecture known as the Brent Kung structure[3] is presented and the design methodology for this structure is discussed.

Chapter 4 introduces the proposed conditional sum adder, its algorithm and the design methodology. The complete architecture is discussed in detail. Towards the end of this chapter the simulation results from the proposed conditional sum adder are compared
with the simulation results obtained from the structure known as the Brent Kung structure, discussed in Chapter 3, to make a qualitative analysis.

Chapter 5 discusses the new concept of FPGA’s[4] and necessity to implement a structure in FPGA’s. The implementation of the conditional sum adder in FPGA’s is presented.

Finally, Chapter 6 leads to the conclusion of this thesis.
Before discussing the proposed conditional sum adder, it would be necessary to look at various existing architectures and show how the proposed architecture differs from the existing structures[2]. Some of the important adder architectures are discussed below.

### 2.1 Ripple Carry Adder

The ripple carry adder is one of the earliest structures, and in this configuration an N-bit adder is constructed by cascading N-full adder circuits in series. Here the carry generated ripples from one stage to the next. So an adder actually waits for the carry signal to arrive from the previous adder and, hence, the delay through this structure depends on the number of logic stages. A block diagram for the ripple carry adder is shown in Figure 2.1.

![Figure 2.1 Ripple carry adder](image-url)
The basic building blocks here are full adder cells that compute the sum and the carry, given as

\[
\text{Sum} = A \oplus B \oplus C \quad (2.1)
\]

and

\[
\text{Carry} = AB + BC + AC \quad (2.2)
\]

where A, B, C are the input bits.

The delay of the ripple carry adder is proportional to the number of input bits and is given by

\[
T_{\text{adder}} = (N-1) t_{\text{carry}} + t_{\text{sum}} \quad (2.3)
\]

where \( t_{\text{carry}} \) is the propagation delay of the signal from the carry input \( C_{\text{in}} \) to the carry output \( C_{\text{out}} \) and \( t_{\text{sum}} \) is the propagation delay of the signal from the carry input \( C_{\text{in}} \) to the sum \( S \). This kind of adder would be good and area efficient when computing for fewer bits or when speed is not of a prime concern. But as the number of bits increases, the delay associated with this adder would be large and, hence, when speed is a major factor a different adder must be adopted.

### 2.2 Carry-Bypass Adder

The main concept in a carry-bypass adder is that the carry generation in a block is based on generate and propagate signals. Suppose signals \( A_i \) and \( B_i \) are the inputs to an adder and the values of \( A \) and \( B \) are such that all the propagate signals are high; then the carry-out would be equal to the carry-in. Hence, when all the propagate signals are equal
to one, the carry coming in is forwarded directly to the next block rather than passing through all the individual adder cells. When all the propagate signals are not equal to one, the carry propagates through all the cells. This is illustrated clearly in the 4-bit carry-bypass structure in Figure 2.2. When \( P_0 P_1 P_2 P_3 \) are all equal to one, then the carry \( C_{in} \) appears at the output through the bypass rather than propagating through all the blocks. This basically is used to increase the speed of operation of the adder.

![4-bit carry-bypass structure](image)

**Figure 2.2 4-bit carry-bypass structure**

In order to build a bypass adder with \( N \) number of bits, the structure is achieved by cascading \( N/M \) equal length stages, where \( M \) is the number of inputs to each stage. Figure 2.2 represents a single stage with 4 adder cells and a multiplexer for the bypass. Thus, for a 16-bit bypass adder, \( N=16 \) and \( M=4 \) to obtain 4 equal stages of 4 inputs each. The block diagram is shown in Figure 2.3.
Hence, for such a structure with $N$ inputs divided into $N/M$ equal length stages where $M$ is the number of inputs to each stage, the propagation delay through the total adder is given by

$$T_p = t_{\text{setup}} + M t_{\text{carry}} + (N/M - 1)t_{\text{bypass}} + M t_{\text{carry}} + t_{\text{sum}} \quad (2.4)$$

where $t_{\text{carry}}$ is the propagation delay through a single bit, $t_{\text{bypass}}$ is the propagation through the bypass multiplexer, $t_{\text{sum}}$ is the time for the sum to be generated, and $t_{\text{setup}}$ is the time required for the generate and propagate signals to be generated.
2.3 Carry Select adder

The reason carry propagation is slow in a ripple adder is because each stage needs to have the inputs -- say A, B and C -- available before it can compute the carry generated from the addition. One way of eliminating this dependency on the carry input is to calculate both \( A + B + 0 \) and \( A + B + 1 \) which determine the results when there is no carry into the system and when there is a carry into the system respectively and then choose the appropriate result when the input carry becomes available. This is the basic concept of the carry select adder. Figure 2.4 gives the basic idea of a four bit carry select module-topology.

![Carry select module-topology](image)

**Figure 2.4** Carry select module-topology
As seen in Figure 2.4, the carry select adder calculates two possibilities for the high-end values—One for the case when low-end has a carry value and one when it does not. The correct output is determined by multiplexer connected to the carry input of the low end carry. Hence the computation here is performed based on a pre-evaluation. Figure 2.4 shows the concept involved in adding the bits ‘k’ to ‘k+3’ and represents a single block. Here the structure adds four bits starting from a value ‘k’ to ‘k+3’ where k= 0,1,2..n. Hence, to build a 16-bit carry select adder we would need four such blocks with an equal number of inputs, as shown in Figure 2.5.

Figure 2.5 16-bit carry select adder
The propagation delay of this adder is also linearly proportional to \( N \), and an adder built by cascading a number of equal length adder stages would have a total adder delay given by,

\[
T_{\text{add}} = t_{\text{setup}} + M t_{\text{carry}} + (N/M) t_{\text{mux}} + t_{\text{sum}}
\]

(2.5)

where \( t_{\text{setup}} \), \( t_{\text{carry}} \) and \( t_{\text{sum}} \) are fixed delays, \( N \) is the total number of bits, \( M \) is the total number of bits per stage, and \( t_{\text{carry}} \) is the delay through a single full adder cell.

### 2.4 Carry Look-Ahead Adder

The basic goal in the carry look-ahead adder is to improve the speed of simple ripple-carry adder design through use of more complicated and hardware-costly design techniques. The methodology involved here is to determine individual carry bits by examining all data inputs rather than waiting for carry to propagate through. Figure 2.6 shows the basic concept involved in a carry look-ahead adder, and is explained below.

![Figure 2.6 Carry look-ahead adder concept](image-url)
The carry look-ahead adder consists of a propagate/generate generator, a sum generator, and a carry generator. The gray colored area in Figure 2.6 indicates where the main concept of the carry look-ahead adder is involved. Here the propagate generator XOR’s the input bits together to determine if this bit position propagates a previous carry. The result of the propagate generator is used directly in generating the sum bits. The generate generator AND’s input bits together in order to determine if this bit position will generate a new carry. Hence the propagate and generate signals—$P_i$ and $G_i$—respectively, are formed from the individual input operand bits, $A_i$ and $B_i$ as follows:

\[ P_i = A_i \oplus B_i \quad (2.6) \]

and

\[ G_i = A_i \cdot B_i \quad (2.7) \]

The sum generator XOR’s the carry-in calculated from the previous two bits and the propagate signal of the current two bits; hence, the name carry look-ahead adder. The sum is given by the relation,

\[ S_i = P_i \oplus C_i \quad (2.8) \]

The carry generator in the carry look-ahead adder takes the propagate/generate signals as its inputs and generates a carry for the next bit. The XOR of this carry bit with the next propagate bit is now determined to generate the sum.
A carry out of the $i^{th}$ bit is generated when both $A_i$ and $B_i$ are 1 or when a carry-in is received and is propagated through; thus,

$$C_{i+1} = G_i + P_1 \cdot C_i$$  \hspace{1cm} (2.9)

and a carry within the stage is generated, or a carry-in is propagated through the stage. Hence,

$$C_1 = G_0 + P_0 \cdot C_0$$  \hspace{1cm} (2.10)

Similarly in the second stage it is possible to generate a carry in that stage, or propagate the $C_{in}$ to the $C_{out}$:

$$C_2 = G_1 + P_1 \cdot C_1$$  \hspace{1cm} (2.11)

Substituting the value of $C_1$ in $C_2$

$$C_2 = G_1 + P_1 (G_0 + P_0 \cdot C_0)$$  \hspace{1cm} (2.12)

$$= G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0$$  \hspace{1cm} (2.13)

Similarly,

$$C_3 = G_2 + P_2 \cdot C_2$$  \hspace{1cm} (2.14)

$$= G_2 + P_2 \cdot (G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0)$$  \hspace{1cm} (2.15)
and so on for all the stages of the adder. The delay time of an n-bit carry look ahead adder is at least linear with the number of bits.

In the above basic study of the different existing adders, it is seen that faster architectures would generally require a more complex structure and a larger area. Hence, the desirable properties for an adder are small area and greater speed. This is the main idea in view of the approach to the design of the proposed conditional sum adder.
Chapter 3
Logarithmic Adder

3.1 Introduction

The type of structure of any adder greatly affects the speed of the circuit. The logarithmic structure is considered to be one of the fastest structures since it reaches theoretical limit for binary addition. The logarithmic concept is used to combine its operands in a tree-like fashion. The delay obtained is equal to ($\log_2 N$) $t$, where ‘$N$’ is the number of input bits to the adder and $t$ is the propagation delay time. Hence, for a 16-bit structure, the logarithmic adder has a delay equal to ‘4t’, while for a simple ripple carry adder the delay is given by (N-1)t and is equal to ‘15t’ for ‘N’ and ‘t’ being the number of input bits and the delay time, respectively. Therefore this structure greatly reduces the delay, and would be especially beneficial for an adder with large number of inputs. This advantage is, however, obtained at the expense of large area and a complex design orientation.

In the following section, a structure known as the Brent Kung Structure [3], which was first proposed by Brent and Kung in 1982 and which uses the logarithmic concept, is discussed. This structure used an operator known as the dot ($\bullet$) operator, which is explained in the architecture, for its basic blocks.
3.2 Brent Kung Architecture

The Brent Kung architecture is divided into three separate stages:

1. Generate/Propagate Generation
2. The Dot (•) Operation
3. Sum generation

3.2.1 Generate/Propagate Generation

If the inputs to the adder are given by the signals A and B, then the generate and propagate signals are obtained according to the following equations.

\[ G = A \cdot B \]  \hspace{1cm} (3.1)

\[ P = A \oplus B \]  \hspace{1cm} (3.2)

3.2.2 The Dot (•) Operation

The most important building block in the Brent Kung Structure is the dot (•) operator. The basic inputs to this structure are the generate and propagate signals obtained in the previous stage. The • operator is a function that takes in two sets of inputs-- (g, p) and (g', p')-- and generates a set of outputs-- (g + pg', pp'). The transistor level design of this structure is shown in Figure 3.1 and the layout is shown in Figure 3.2. These building blocks are used for the generation of the carry signals in the structure. For the generation of the carry signals, the carry for the kth bit from the carry look-ahead concept is given by,
Figure 3.1 Design of the dot (•) operator
Figure 3.2 Layout of the dot operator
\[ C_{o,k} = G_k + P_k (G_{k-1} + P_{k-1} + P_{k-1} (\ldots + P_1 (G_0 + P_0 Ci_o))) \]  

(3.3)

Using the dot operator explained above the Equation 3.3 can be written for the different carry signals as

\[ C_{o,0} = G_0 + P C_{i,0} = \alpha (G_0, P_0) \]

\[ C_{o,1} = G_1 + G_0 P_1 = \alpha ((G_1, P_1) \bullet (G_0, P_0)) \]

\[ \ldots \]

\[ C_{o,k} = \alpha ((G_k, P_k) \bullet (G_{k-1}, P_{k-1}) \bullet \ldots \bullet (G_0, P_0)) \]  

(3.4)

where \( \alpha \) is a function defined in order to access all the tuples. The 8-bit Brent Kung Structure is shown in Figure 3.3. This figure shows all the carry signals generated at different stages in the structure. In the structure, two binary tree organizations are represented -- the forward and the reverse trees. The forward binary tree alone is not sufficient for generation of all the carry signals. The forward binary tree alone can only generate the signals shown as \( C_{o,0}, C_{o,1}, C_{o,3} \) and \( C_{o,7} \). The remaining carry signals are generated by the reverse binary tree.
3.2.3 Sum generation

The final stage in this architecture is the sum generation stage. The sum is given by

\[ S = A \oplus B \oplus C \]  

(3.5)

where A and B are the input signals, and C is the carry signal. The carry is obtained from the dot operator stage discussed earlier, and the exclusive of A and B is actually the propagate signal itself. Hence the sum ‘S’ can finally be represented and realized as

\[ S = P \oplus C \]  

(3.6)

Using the above three stages, the complete architecture is built. The complete block level design of a 16-bit Brent Kung Structure is shown in Figure 3.4, while the layout is
Figure 3.4 16-bit Brent Kung Structure
shown in Figure 3.5. The simulation results for this structure are also available and will be discussed towards the end of Chapter 4 in comparison to the simulation results obtained from the proposed conditional sum adder.
Figure 3.5 Layout of 16-bit Brent Kung Structure
Chapter 4

Design of Conditional Sum Adder

4.1 Conditional Sum Algorithm

The conditional sum algorithm is the algorithm used in the proposed design. It is considered to be one of the fastest methods to perform addition. Here, for a binary number each of the digit positions is associated with two results, one computed based on the assumption that there is a carry into that position and the other one that there is no carry into that position. The concept involved in the conditional sum algorithm is illustrated in the example shown in Table 4.1.

1. At a time $t = t_0$ in Table 4.1 a computation or the addition is performed on each pair of bits taking into consideration that there is a carry input into the system and that there is no carry input. The resulting pairs of outputs obtained at $t_1$ in Table 4.1 represent a sum and a carry bit for each pair of input bits.

2. Now pairs of bits are grouped and again the computation is performed based on whether or not there is a carry into the system. The resulting output is as shown at $t_2$ in Table 4.1.

3. This process is repeated by grouping the bits together until the final result comes out at $t_5$, as indicated in Table 4.1. At $t_6$ we have both the results computed assuming that there is a carry and that there is no carry into the system. To obtain the
correct result, it is necessary to simply multiplex the required output based on the input carry.

Suppose A and B are the inputs and \( C_{in} \) is the carry input in the following example. The following results will be obtained after each level of conditional sum addition:

**Table 4.1** Example to illustrate the concept of conditional sum

<table>
<thead>
<tr>
<th>( T=t_0 )</th>
<th>( A = 0111110111101111 )</th>
<th>( B = 10011110111011 ) and ( C_{in} = 0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 )</td>
<td>01 01 01 10 10 10 01 01 10 10 01 10 10 10</td>
<td>without ( C_{in} )</td>
</tr>
<tr>
<td></td>
<td>10 10 11 11 11 10 10 11 11 10 11 11</td>
<td>with ( C_{in} )</td>
</tr>
<tr>
<td>( t_2 )</td>
<td>011 100 110 011 110 100 101 110</td>
<td>without ( C_{in} )</td>
</tr>
<tr>
<td></td>
<td>100 101 111 100 111 101 110 111</td>
<td>with ( C_{in} )</td>
</tr>
<tr>
<td>( t_3 )</td>
<td>10000 11011 11100 11010</td>
<td>without ( C_{in} )</td>
</tr>
<tr>
<td></td>
<td>10001 11100 11101 11011</td>
<td>with ( C_{in} )</td>
</tr>
<tr>
<td>( t_4 )</td>
<td>1000111011 111011010</td>
<td>without ( C_{in} )</td>
</tr>
<tr>
<td></td>
<td>100011100 111011011</td>
<td>with ( C_{in} )</td>
</tr>
<tr>
<td>( t_5 )</td>
<td>10001110011011010</td>
<td>without ( C_{in} )</td>
</tr>
<tr>
<td></td>
<td>10001110011011011</td>
<td>with ( C_{in} )</td>
</tr>
<tr>
<td>( t_6 )</td>
<td>10001110011011010</td>
<td>without ( C_{in} )</td>
</tr>
</tbody>
</table>
There are \((\log_2 n + 1)\) levels of conditional sum computations with the last stage which perform the final computation. Finally, from the example it is seen that the Sum = 0001110011011010 with the Carry out = 1, which is the required result.

### 4.2 Architecture of the Conditional Sum Adder

The basic idea behind the proposed adder is the concept of Conditional Sum explained earlier. It is a synchronous parallel adder in which the addition is performed in parallel and carried out in \(\log_2 n + 1\) operating stages. Since the ultimate performance of any circuit would be based on its algorithm, it is necessary to have such a structured algorithm. Also, some of the desirable properties of a VLSI circuit are to have a regular network and to implement it with only a few different types of cells since this would allow a smaller compact structure and would be both area and time efficient. This architecture has been build based on the above principles. The entire architecture of the system is divided into sub-systems. In the process of each the sub-systems design, the steps specified below have been followed:

1. A complete transistor level design of the circuit is derived.
2. The layout for this transistor level design is made.
3. The post layout circuit is simulated an optimized to obtain the required results.

The basic building blocks or sub-systems of the adder architecture are;

1. Full Adders
2. Selectors

3. Multiplexers

1. Full Adders

   The Full Adder blocks have been designed to generate two sets of sum and carry signals based on the assumption that there is a carry and there is no carry into the system. Hence, this is a kind of pre-evaluation of the result of the circuit.

2. Selectors

   Sum and carry signals from the full adder stage are combined in this stage for which the control is provided by the lower carry signals or the carry signals from the previous stage.

3. Multiplexers

   In this stage, based on the real carry input signal into the system, the appropriate sum and carry signals are multiplexed out to provide the final result. This stage represents the final step in the algorithm in the process to obtain the sum and carry signal.

   Hence, the entire structure of this proposed conditional sum adder is built using these three building blocks. Careful design of these building blocks is essential since these blocks ultimately define the speed and the area occupied by the circuit. The hierarchical structure of a 4-bit adder is shown in Figure 4.1, and a similar concept is
used to extend the circuit for a 32-bit implementation, which is shown towards the end of this chapter. The details and design of each of the building blocks are also shown. FA is the full adder, INV the inverter, S is the selector, and M is the multiplexer in Figure 4.1.

**Figure 4.1** Block diagram of 4-bit conditional sum adder
4.2.1 Design of Full Adder Cell

The full adder cell is the most important part of the design, and influences the speed of the circuit. The function of this full adder cell is to perform the arithmetic addition of two operands A and B. The realization of this full adder circuit is based on the requirements of the circuit or, to be precise the circuit is needed to generate the sum and carry signals under the assumptions that there is a carry and that there is no carry. Hence, both the output and the inverted value of the output of the sum and the carry signals are made available from the full adder circuit. In order to arrive at the required results, the output equations for the full adder are as shown below.

If A and B are the inputs to the adder cell, then we have the required sum and carry signals as

\[ S_1 = A_i \oplus B_i \]  \hspace{1cm} (4.1)  
\[ S_2 = (A_i \oplus B_i)' \]  \hspace{1cm} (4.2)  
\[ C_1 = A_i \cdot B_i \]  \hspace{1cm} (4.3)  
\[ C'_1 = (A_i \cdot B_i)' \]  \hspace{1cm} (4.4)  
\[ C_2 = A_i + B_i \]  \hspace{1cm} (4.5)  
\[ C'_2 = (A_i + B_i)' \]  \hspace{1cm} (4.6)
In the above equations, \( S_1, C_1 \) and \( C'_1 \) are the output sum, carry, and the complement of the carry assuming that there is no carry input, while \( S_2, C_2, \) and \( C'_2 \) are the sum, carry, and the complement of the carry assuming that there is a carry input into the system. At a logic level, the circuit therefore utilizes two 2-input NOR gates, one 2-input NAND gate, and three inverters. The delay associated with the circuit would be equal to four gate delays assuming equal delays for the gates.

The transistor level design of the circuit is shown in Figure 4.2, the complete CMOS layout is shown in Figure 4.3 and the simulation results are shown in Figure 4.4. The design of the circuit could be achieved in different logic structures but the purpose of using the circuit shown in the Figure 4.2 is to obtain a glitch free output. Any kind of glitches of hazards in the output of the logic level design would seriously effect the final output of the system. In the Figures below \( C'_1 \) is represented as \( C_{1\text{inv}} \), \( C'_2 \) as \( C_{2\text{inv}} \), \( S_1 \) as \( S_{\text{inv}} \) and \( S_2 \) as \( S \).
Figure 4.2 Transistor level design of Full Adder
Figure 4.3 Layout of Full Adder
Figure 4.4 Simulation results for the Full Adder
4.2.2 Design of Selector

The main purpose of the selector is to select the data inputs based on the two control inputs which are the carry signals generated at the lower significant bits. The data inputs once selected are either fed to another selector before reaching the multiplexer discussed later or directly to a multiplexer as in the case of a single bit adder to be multiplexed out to obtained the required result.

If the two selector inputs are $S_1$ and $S_2$, the two selector outputs are $S_x$ and $S_y$ and the controls to the selector are $C_1$ and $C_2$ then the output equations for the selector can be given as

\[
S_x = S_1C'_1C'_2 + S_1C'_1C_2 + S_2C_1C'_2 + S_2C_1C_2
= S_1C'_1 + S_2C_1\quad (4.7)
\]

\[
S_y = S_1C'_1C'_2 + S_2C'_1C_2 + S_1C_1C'_2 + S_2C_1C_2
= S_1C'_2 + S_2C_2\quad (4.8)
\]

This functionality can be more clearly seen in Table 4.2.
Table 4.2 Selector Functionality Table

<table>
<thead>
<tr>
<th>Select control</th>
<th>Output signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>C2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The transistor level design of the selector is shown in Figure 4.5 and the layout is shown in Figure 4.6. As seen, the selector is built using simple pass transistors and, hence, only NMOS transistors are sufficient for the realization. This advantage of building a small simple structure using only NMOS transistors comes at an expense. NMOS only pass transistors are subjected to voltage loss, which is dealt at the complete design stage by introducing buffers.
Figure 4.5 Transistor level design of selector
Figure 4.6 Selector Layout
4.2.3 Multiplexer Design

The final stage of the Conditional sum adder is the multiplexer stage. The multiplexer is basically used here to select and multiplex out the final required output based on the fact whether or not there is a carry into the system. If $X_{in}$ and $Y_{in}$ are the two inputs to the multiplexer and $C$ is the control signal, then the output, $M_{out}$ could be realized according to the equation

$$M_{out} = X_{in}C + Y_{in}C'$$  \hspace{1cm} (4.9)

The functionality table for the multiplexer is shown below,

<table>
<thead>
<tr>
<th>C</th>
<th>$M_{out}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$Y_{in}$</td>
</tr>
<tr>
<td>0</td>
<td>$X_{in}$</td>
</tr>
</tbody>
</table>

One of the simplest methods to design the multiplexer would be to use transmission gates, and the design could be realized using six transistors as shown in the transistor level design, Figure 4.7. The layout of the multiplexer is shown in Figure 4.8. Transmission gates used in the design of the multiplexer reduce the parasitic capacitances
and result in a fast circuit as compared to a multiplexer implemented using only basic logic gates.

Figure 4.7 Transistor level design of Multiplexer
Figure 4.8 Layout of Multiplexer
4.3 Basic Structure design

In the design of a basic structure using all the above building blocks, to form a one-bit conditional sum adder we would require only the full adder, the multiplexer, and an inverter which basically provides the carry into the system. The block diagram of the one-bit adder is shown in Figure 4.9.

![Diagram of one-bit conditional sum adder](image)

**Figure 4.9** One bit conditional sum adder
When designing for a higher number of input bits, we obviously would require all the building blocks. The number of building blocks required in the architecture depends on the number of inputs into the system, and is different for each of the stages in the circuit.

1. The number of building blocks of the Full Adder is proportional to the total number of input bits. Thus an N-bit Conditional Sum Adder require 'N' number of Full Adder cells.

2. The number of selector cells required in the structure is given by the equation

\[ S_N = 2S_N/2 + 0.5N + 2 \quad \text{(4.10)} \]

where \( S_N \) is the number of selector cells required, and \( N \) is the number of bits. The total number of selector stages in the design of the entire system would be given by the relation \( \log_2N \) with \( N \) being the number of input bits into the system.

3. The number of multiplexer cells required is \( M_N = N + 1 \), where \( M_N \) is the number of multiplexer cells and \( N \) is the number of input bits into the system.

4. Finally, one inverter is required to provide the carry into the system and does not depend on the number of input bits.

Using the above structural concept, the conditional sum adder was designed for 32 bits. The total design of the 32-bit conditional sum adder is achieved using one stage of full adders, \( \log_2 n \) stages of selectors which is equal to 5 in this case, and one stage of
multiplexers. The complete design and layout and simulation of the 2-bit and 4-bit adders are shown in Figures 4.10 to Figures 4.15.

In the layout of the 32-bit adder, an additional stage is introduced into the structure. This additional stage is the buffer stage. The necessity of introducing buffers in the circuit is to increase the speed of the circuit and to pull up the output voltage levels. The reduction in the voltage levels and the speed in the circuit are attributed to the selectors built using only NMOS transistors. NMOS only pass transistors are subjected voltage loss and the delay associated kills the performance of the circuit. Thus, the buffers are introduced to speed up the circuit performance and also to pull up the voltage levels as much as possible thus acting as voltage level restorers. The buffers used here are a cascade of two inverters and, thus, a non-inverting buffer. The transistor level design and layout of the buffer are shown in Figure 4.16 and Figure 4.17. The block level design of the 32-bit adder is shown in Figure 4.18, the layout in Figure 4.19, and the simulation results in Figure 4.20. In the simulation results the signals shown starting from the bottom are, the input signals \( A_0, A_1, B_0, B_1 \), the input carry signal \( C_0 \), the output bits \( \text{SUM}_0, \text{SUM}_1 \) and the output carry CARRY.
Figure 4.10 Block level design of 2-bit adder
Figure 4.11 Layout of 2-bit adder
Figure 4.12 Simulation results for 2-bit adder
Figure 4.13 Block level design of 4-bit adder
Figure 4.14 Layout of 4-bit adder
Figure 4.15 Simulation results for 4-bit adder
Figure 4.16 Transistor level design of Buffer
Figure 4.17 Layout of Buffer
Figure 4.18 Block level design of 32-bit adder
Figure 4.19 Layout of 32-bit adder
4.4 Comparison between Brent Kung and Conditional Sum adder

One of the objectives in this thesis is to provide a comparison between adders. Hence, it is necessary here to provide a comparison between the Brent Kung adder and the conditional sum adder which have been studied in detail. Two most important factors worth comparing would be the area, signal quality and the timing for the two adders.

Firstly, it is observed that the conditional sum adder occupies a lesser area than the Brent Kung structure. This is because of the smaller size of the selector cells is used in building of the conditional sum adder when compared to the carry generator cells in the Brent Kung structure. The area of the Brent Kung structure is obtained to be $600\mu m \times 2450\mu m$ (height \times length) while the area of the conditional sum adder without buffers is $395\mu m \times 2625\mu m$ (height \times width) for a sixteen input bits. This shows a considerable difference in the design area. When the buffers are introduced into the conditional sum adder the area obviously increases and is found to be $480\mu m \times 2625\mu m$ (height \times width). Even after the increase in the area of the conditional sum adder due to the buffers, the total area is still found to be lesser than the area of the Brent Kung structure.

The second important factor between the two adders would be the timing comparison. The simulation results for the 16 bit Brent Kung structure and the conditional sum adder are shown in Figure 4.20 and Figure 4.21. In the waveforms of the Brent Kung structure it is observed that certain unwanted transitions occur at certain points. These are hazards resulting from the logic organization of the architecture the Brent Kung structure. No such hazards are observed in the case of the conditional sum adder.
Figure 4.20 Simulation results for 16-bit conditional sum adder
**Figure 4.21** Simulation results for 16-bit Brent Kung adder
Chapter 5

FPGA implementation

5.1 Introduction to FPGA’s

Field programmable gate arrays (FPGA’s) are a class of general purpose devices that can be configured for a wide variety of applications. Field programmable gate arrays were first introduced by Xilinx in the mid-1980’s [4]. Before the availability of FPGA’s, a designer had the options for implementing digital logic in discrete logic devices (VLSI or SSI), programmable devices (PAL’s or PLD’s), and cell-based Application Specific Integrated Circuits (ASIC’s). At this stage it is necessary to see the difference between an ASIC and an FPGA and determine the need to carry out the implementation of the circuit in FPGA’s. Up to this point in this thesis, what was presented was more of a custom hardware circuit design—often known as application specific integrated circuits. ASIC’s provide the exact functionality required for a specific task. They are smaller, faster, cheaper and consume less power than a programmable processor and will solve the specific problem for which it was designed. But in a situation where we would require a slightly modified alternative to the developed ASIC the approach would probably require rebuilding the entire chip which would be both costly and time consuming.

It is in such a situation that FPGA’s could come into play. A discrete device can be used to implement a small amount of logic, while a programmable device, by comparison, is a general-purpose device capable of implementing extremely large logic. The flexibility here is that it is capable of being programmed by the users at their site.
using programming hardware. Hence, FPGA's provide the benefits of custom VLSI, while avoiding the initial cost and time delay associated with ASIC design. They allow the implementation of integrated digital electronic circuits without requiring the complex approach used in a conventional chip fabrication. These are highly tuned hardware circuits that can be modified at any point during use and consist of configurable logic blocks (CLB's) which implement the logical functions of gates. This architecture would be discussed in the next section. In FPGA's, the logic function performed within the logic blocks as well as the interconnections between the blocks can be programmed repeatedly, and this configuration within the chip can be accomplished in a few milliseconds. ASIC's definitely have their advantages over FPGA's, but FPGA's are highly recommended where time and money are a factor. In fact, the field programmable gate array is the preferred first step into application specific integrated circuits.

In order to clearly illustrate the above discussion with a simple example, imagine yourself on a warship in the middle of an ocean. The ship obviously has a great deal of sophisticated equipment built using a number of important IC’s. When such an IC fails to perform its desired task and the ship does not happen to have a stock of the required chips, it obviously would be extremely beneficial to program an FPGA and use it in place of the custom IC rather than the ship returning to the dock for the device.
5.2 Architecture (Xilinx 4000 series FPGA)

In the FPGA’s the architecture and technology determine the methods of interconnections and programming. The most important technologies are

1. SRAM technology
2. Anti-fuse technology
3. EPROM/EEPROM technology

1. SRAM Technology

In the static RAM technology, programmable interconnections are made using pass transistors, transmission gates or multiplexers that are controlled by SRAM cells. The advantage is that it allows fast in circuit reconfiguration.

2. Anti-Fuse Technology

In this technology an anti-fuse resides in high impedance and can be programmed into low impedance or fused state.

3. EPROM/EEPROM Technology

This concept is similar to that used in EPROM memories. In this technology there is no necessity for an external storage of the configuration.
The FPGA used in this thesis which is the Xilinx XC4010XLPC84 is SRAM based.

The major building blocks in an FPGA are

1. Configurable Logic Blocks (CLB's).
2. Input/Output Blocks (IOB's).
3. Programmable interconnects.

![FPGA architecture](image)

**Fig 5.1** FPGA architecture

Xilinx FPGA's consisted of a matrix of logic cells or the above mentioned CLB's surrounded by vertical and horizontal channels of programmable interconnects and the periphery being surrounded by IOB's. A basic block diagram of this architecture is shown above in Figure 5.1. FPGA's that are fine grained in structure have large number of simple CLB's while those with a coarse grained structure have smaller number of powerful blocks.
5.2.1 Configurable Logic Blocks

Each CLB contains a pair of flip flops and two independent four input function generators. The flip flops are accessed through the thirteen inputs and four outputs of the configurable logic blocks. The configurable logic blocks are responsible for implementing most of the logic in an FPGA. A third function generator is also available and has three inputs. One or two of these inputs can be the outputs from the other two function generators while the other input(s) are from outside the CLB. Hence each CLB would be capable of implementing functions of up to nine variables. The outputs from these function generators are stored in flip-flops within the CLB. Implementing large functions in a single CLB would reduce the number of cells needed and the delay associated therefore resulting in both area and speed efficiency.

5.2.2 Input / Output Blocks

The input/output blocks in an FPGA provide interface between the external package pins and the internal logic. Each IOB is defined as either as an input, an output or a bi-directional signal. Here two paths are responsible for bringing the input signals into the array and also connect to an input register that is capable of being programmed either as an edge-triggered flop-flop or as a level sensitive latch. The inputs can be globally configured for either TTL or CMOS logic.
5.2.3 Programmable interconnects

Internally the connection are achieved using metal segments with programmable switching points. These switching points or switching matrices basically consists of six pass transistors that can turned on and off to provide the desired routing. The major interconnections within the FPGA are provided by single length lines which are vertical lines that intersect at a switch matrix, double length lines which are twice as long as the single length lines and long lines that run the entire length or width of the array of cells. The various interconnections inside an FPGA are made using these routing channels. CLB outputs are routed to the long lines through tri-state buffers or the single length interconnect lines. In addition there is also a routing resource around the IOB known as the versa ring which facilitates the swapping of the pins and facilitates redesign.

5.3 Design Implementation in FPGA ’s

In the process for the implementation of the design a sequence of basic steps are followed. The 32-bit conditional sum adder described in Chapter 4 is implemented here in FPGA ’s. The implementation is done using the following procedure:

1. Firstly, the logic level circuit is created using either a schematic design software or a hardware description language. In this case, for the design of the 32-bit adder both the methods have been tested. A schematic design of the 32-bit adder is implemented in a hierarchical fashion. In the approach using hardware description language, VHDL code has been generated from the schematic circuit previously implemented using Mentor
Graphics tools. Mentor Graphics tools have the capability to generate VHDL code from the schematic design. This code is generated by selecting the design and using the 'generate VHDL' command from the pull down menu. The generated code has a .vhd file extension. The .vhd files generated are exported and used in the HDL editor of the Xilinx Software to carry out the implementation. Netlists are generated from the code and it is necessary to be sure that the library sets of the targeted FPGA are available in the tool.

2. The netlist produced by the design entry is transformed into a bit stream file which is used to configure the FPGA. The design here is initially mapped onto the FPGA. This is followed by the placement of the logic blocks created in the mapping process and, finally, the routing takes place. This entire process is shown in the Xilinx tool as a design flow. The logic cell array file thus obtained is converted into the bit stream file to configure the FPGA.

3. In the final stage the circuit configuration is downloaded onto the FPGA. The chip used for the configuration is the Xilinx XC4010XLPC84. The demo board used here, in addition to the FPGA, also has the PROM, which is used to configure the FPGA. The FPGA can actively read the configuration data from the PROM, or the configuration could be written into the FPGA.

The procedure described above is illustrated here with the implementation of the 32-bit adder. The gate level design of the multiplexer and the design of a selector are shown in
Figure 5.2 and Figure 5.3 respectively and Figure 5.4 shows the design of the full adder. Using these circuits as the basic building blocks, the 32-bit adder is designed. The VHDL files generated are sel2.vhd, mux2.vhd, modFA.vhd, invert.vhd and SuBu.vhd which are lower level selector, multiplexer, full adder, inverter and buffer blocks and are necessarily the behavioral models. PA32bit.vhd is the 32-bit adder code which is hierarchically built with PA8bit.vhd, PA16bit.vhd and PA24bit.vhd. These files describe only the connectivity between the behavioral blocks and the connectivity between the hierarchical blocks. Since the structure is considerably large, the design is implemented in a hierarchical fashion. Figure 5.5 shows the design of an 8-bit adder. A macro of this 8-bit adder is created and used in the design of the 16-bit adder as shown in Figure 5.6. A macro could be created by selecting the ‘create macro’ option in the pull down menu while the schematic is open. Similarly, a macro is created from the 16-bit adder and used in the design of the 24-bit adder as in Figure 5.7 and so on until the final design of the 32-bit adder is obtained as in Figure 5.8. In this process of creating the entire design, initially a new version is selected from the pull down menu. This appears as a directory named ‘version1’ in the design manager. Under the new version1 it is possible to have any number of new revisions which appear as ‘revision1’, ‘revision2’ etc. It would be necessary to go to a new revision only if the current revision stops because of some error. In the implementation of the design on an FPGA, the 32-adder was downloaded onto the Xilinx XC4010XLPC84 FPGA using the xilinx design editor. In order to download the design onto the FPGA, the command used is ‘xsload [file]’ at the DOS prompt. The file is a ‘bit’ file. The design can also be downloaded onto the FPGA by using the download
command button which shows the entire flow of placement and routing of the design in the form of a flow engine. Once the design is downloaded onto the FPGA, this chip could be used as the 32-bit adder chip. This brings to light, the simplicity of converting a design into a chip, ready to use, using an FPGA.

Figure 5.2 Gate level design of multiplexer
Figure 5.3 Gate level design of selector
Figure 5.4 Gate level design of full adder
Figure 5.5 Design of 8-bit adder
Figure 5.6 Design of 16-bit adder
Figure 5.7 Design of 24-bit adder
Figure 5.8 Design of 32-bit adder
Chapter 6

Conclusion

This thesis introduces a novel adder structure known as the conditional sum adder which may become a critical element of the data-path processors built using adders, registers etc. Hence the performance of many parallel architectures such as DSP chips depends on the data-path which in turn depends on the individual units such as the adder. Thus there is a necessity to make a constant effort to improve the adder performance.

The CMOS implementation and simulation results of the conditional sum adder are put forward in this thesis. Along with this new adder structure, the already existing adder structures such as the Brent Kung structure and various other adders are also studied in comparison with the conditional sum adder. The implementation of the Brent-Kung adder structure is also presented in this thesis. The difference between the Brent-Kung adder and the proposed conditional sum adder is observed and the necessity to introduce a new adder structure is justified.

In the design and implementation of the conditional sum adder, factors such as area, timing and hazards in the output have been taken into account. The transistor level realization of the subsystems are shown and discussed in chapters 3 and 4. Various designs for the subsystems have been used before approaching the final designs. Also, before approaching the transistor level designs of the subsystems the logic of the blocks was verified at a gate level. This was necessary because hazards in the design of the circuit lead to unwanted glitches in the output which affect the operation of the total
system. Since different signals in the system arrived at different times, it was also necessary to design the circuits so as to synchronize the signals in order to obtain a correct result. The introduction of buffers in the circuit has improved the overall performance of the system and resulted in better voltage levels at the output nodes. By careful design a compact adder layout was obtained. The main objective was to build a VLSI circuit with better performance and smaller area and this was in comparison to the existing Brent Kung structure. From the simulation results and layouts of the conditional sum adder and the Brent Kung structure seen in chapters 3 and 4, a considerable difference in the performance and the area of the two structures is observed. This difference is explained in chapter 4. This difference is obviously large when compared with the adders such as the ripple carry adder and other adders discussed in chapter 2.

In the FPGA implementation of the conditional sum adder, the entire circuit is downloaded onto the Xilinx XC4010 FPGA. This implementation of the circuit allows the designer to quickly build a hardware and illustrates the difference between an application specific integrated circuit and a circuit implemented on an FPGA and was discussed in chapter 5.

In the design of the conditional sum adder it was observed that the introduction of buffers in the circuit has increased the area considerably though the overall area is still smaller than that of the Brent Kung structure. In the absence of the buffers the area taken by the conditional sum adder is far smaller than the area taken by the Brent Kung Structure. One possible suggestion to future designers would be to try and design the entire system so as to eliminate the need of buffers without sacrificing the performance of the system.
To probe further, literature on computing and arithmetic which is relevant to this topic is available in the IEEE journal of computers and other IEEE resources. Specific resources are also provided in the section on bibliography along with closely related references for a future study on adders.
References


