Chapter 2
Design and Implementation of Efficient Adders

Contents

Chapter 2 ................................................................................................................................. 4
2.1 Introduction ...................................................................................................................... 4
2.2 Review of Existing Adder Designs .................................................................................. 5
  2.2.1 Ripple Carry Adder ............................................................................................5
  2.2.2 Carry Select Adder (CSA) ..................................................................................6
  2.2.3 Carry Look-Ahead Adder ...................................................................................7
  2.2.4 Prefix Based Adders .........................................................................................11
2.3 Design and Implementation of Efficient Sum Computation Block for Higher Bit Sparse Adders ............................................................................................................ 14
2.4 Design and Implementation of Higher Bit Sparse Adder ................................................ 19
2.5 Simulation Results ....................................................................................................... 21
2.6 Conclusions .................................................................................................................... 27

2.1 Introduction

Arithmetic and Logic Unit (ALU) is a critical element in any CPU. In ALU, adders play a major role not only for addition but also in performing many other basic arithmetic operations like subtraction, multiplication, increment / decrement etc. Thus, realizing an efficient adder is required for better performance of a processor in general and ALU in particular [1-6]. Research into design of efficient adder algorithms for hardware implementation of Very Large Scale Integrated (VLSI) arithmetic circuits started in late 1950s. Many designs based on serial and parallel structures have been proposed to optimize different parameters from time to time [5]. Binary addition consists of four possible elementary operations, which are

\[
\begin{align*}
0 + 0 &= 0 \\
0 + 1 &= 1 \\
1 + 0 &= 1 \\
1 + 1 &= 10
\end{align*}
\]
2.2 Review of Existing Adder Designs

The first three operations produce only a ‘Sum’ whose length is one digit, but when both augend and addend bits are equal to 1, the binary sum consists of two digits. The higher significant bit of this result is called a ‘Carry’. A combinational circuit that performs the addition of two bits is called a half-adder while the one that performs the addition of three bits is known as a full-adder [5].

2.2 Review of Existing Adder Designs

Adders can be broadly classified into following four classes [5]:

- Ripple Carry Adder (RCA)
- Carry Select Adder (CSA)
- Carry Look-Ahead Adder (CLA)
- Parallel Prefix-based Adder (PPA)

2.2.1 Ripple Carry Adder

A full adder (FA) is a combinational circuit that takes two operand bits and a carry bit, say A, B and C\textsubscript{i} respectively, as inputs and gives Sum (S) and Carry bit (C\textsubscript{o}) as outputs. This output Carry bit C\textsubscript{o} will serve as input Carry bit for the successive full adder. The combinational circuit follows the Boolean equations 2.1 and 2.2 mentioned below to implement a full adder and the gate level implementation of the same is shown in Fig 2.1. A simple implementation of higher operand adder for two operands A and B is carried out by cascading n of these basic full adder units and is known as a ripple carry adder.

A simple 4-bit ripple carry adder is shown in Fig 2.2. The design of this adder is simple and implementation is easy, but it suffers from serious delay issues. This is because the next stage full adder needs to wait for Carry bit from the previous stage FA. By inspecting the FA shown in Fig 2.1 it can be observed that the gate delay from C\textsubscript{in} to C\textsubscript{o} is 2 gates. Therefore, each full adder contributes to a 2-gate delay in the process of rippling the carry [1-6].
2.2 Review of Existing Adder Designs

\[ C_o = (A \cdot B) + (C_i \cdot (A \oplus B)) = (A \cdot B) + (B \cdot C_i) + (C_i \cdot A) \quad (2.1) \]

\[ S = (A \oplus B) \oplus C_i \quad (2.2) \]

Figure 2.1 One-Bit Full Adder

Figure 2.2 Four-Bit Ripple Carry Adder

2.2.2 Carry Select Adder (CSA)

Ripple carry adder waits for the input carry (Ci) and then computes the ‘Sum’ and the Carry out (Co) thus increasing its delay. In order to reduce the delay, carry select adder is introduced, which pre-computes the ‘Sum’ and ‘Co’ for the two possible cases i.e. Ci = 0 and Ci = 1. The calculated Sum is given to a multiplexer, which chooses the correct output depending upon the Ci coming from the previous stage. This pre-computation of Sum reduces the delay of rippling of Carry which is limited to only one multiplexer for each stage. Figure 2.3 below gives the gate level diagram of a 16-bit carry select adder. In this, each 4-bit adder is a bit ripple carry adder. Carry select adder uses more hardware even though it gives less delay compared to ripple carry adder. Thus, there is a tradeoff between area, power and delay between different adders[1-6].
2.2 Review of Existing Adder Designs

2.2.3 Carry Look-Ahead Adder

Various techniques have been proposed from time to time to decrease the overall delay in parallel addition [5]. One such technique is to derive the ‘Sum’ and ‘Carry’ outputs by using intermediate terms defined as ‘Generate (G)’ and ‘Propagate (P)’ terms [5-6]. Generate term produces a carry-out independent of the carry-in, i.e. no matter what the carry-in is, the carry-out is always ‘1’, when both of its inputs A and B are ‘1’ thus \( G = A \times B \). The Propagate term transfers the input Carry as output Carry when only one of the inputs is high and hence Propagate term is defined as \( P = A \oplus B \). Thus we have

\[
G(A, B) = A \times B \\
(2.3)
\]

\[
P(A, B) = A \oplus B \\
(2.4)
\]

Table 2.1 and the example shown below illustrate the concept of Propagate and Generate more clearly. In the Propagate case the ‘Carry-out’ depends on the ‘Carry-in’, i.e. when ‘Carry-in’ is 0 ‘Carry-out’ is 0 and when ‘Carry-in’ is 1 ‘Carry-out’ is 1 and in the case of Generate, no matter what the ‘Carry-in’ is ‘Carry-out’ is always 1.
2.2 Review of Existing Adder Designs

Table 2.1 Truth Table of a Full Adder

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Cin</th>
<th>Sum</th>
<th>Cout</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- **G-P=0**: Cout = 0 (no dependence on Cin)
- **P=1**: Cout = Cin
- **G=1**: Cout = 1 (no dependence on Cin)

Propagate

\[ A \oplus B \]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Generate

\[ A \cdot B \]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The output ‘Sum’ and ‘Carry’ of the full adder in terms of P and G, can be observed from Table 2.1 to be,

\[ S_i = P_i \oplus C_i \quad (2.5) \]

\[ C_{i+1} = G_i + (P_i \cdot C_i) \quad (2.6) \]
2.2 Review of Existing Adder Designs

Figure 2.4 One-bit Full Adder with Carry Propagate and Generate

Figure 2.4 above illustrates the implementation of above equations (2.5) and (2.6) which is essentially same as Fig 2.1 but derived from Table 2.1. This logic is also called carry look-ahead logic. For each bit in a binary sequence to be added, the carry look-ahead logic will determine whether that bit pair will generate or propagate a Carry. This allows the circuit to "pre-process" two numbers being added to determine the carry ahead of time. Thus, when the actual addition is performed, there is no delay from waiting for the ripple carry effect (time it takes for the carry from the first full adder to be passed on to the last Full Adder) [5-6].

Figure 2.5 Ripple Carry Adder with Carry Propagate and Generate
2.2 Review of Existing Adder Designs

The carry look-ahead type implementation of a ripple carry adder is shown in Fig 2.5. It can be seen from this figure that the carry propagation stage determines the critical path that determines the delay. To increase the speed of an adder, this stage has to be redesigned for fast carry propagation.

Keeping this in mind, Weinberger and Smith proposed a method for fast carry generation which states that the carry need not depend explicitly on the previous carry, but can be expressed as a function of only the relevant addend and augend digits and some lower order carry [7].

![Figure 2.6 4-bit Weinberger-Smith CLA](image)

The carry generation is done by first calculating Propagate \( p_i \) and Generate \( g_i \) terms.

\[
g_i = A_i B_i \tag{2.7}
\]

\[
p_i = A_i \oplus B_i \tag{2.8}
\]

After the parallel generation of Propagate and Generate terms, the carries can be generated using the equations below. In the following a 4-bit adder is considered as an example:

\[
C_1 = g_0 + p_0 C_0 \tag{2.9}
\]

\[
C_2 = g_1 + p_1 g_0 + p_1 p_0 C_0 \tag{2.10}
\]
2.2 Review of Existing Adder Designs

\[
\begin{align*}
C_3 &= g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0C_0 \\
C_4 &= g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0C_0 \\
\end{align*}
\] (2.11)

After the carries are generated, the sum is calculated using the equation

\[ S_i = A_i \oplus B_i \oplus C_i \] (2.13)

A typical 4-bit CLA implementing the above equations is shown in Fig 2.6. For wide adders where \( N > 16 \) (\( N \) is the input operand size), the delay of the carry look-ahead adders becomes dominated by the delay of passing the carry through the look-ahead stages and the implementation needs high fan-in gates [1, 5]. To overcome these problems, a new breed of networks has been designed that pass the carry through the look-ahead stage in around \( \log(N) \) stages. These networks are called Tree Networks and the adders that utilize these networks are called tree-adders or prefix-adders [5]. There are many ways to build the tree adders which offer tradeoffs among parameters like, the number of stages of logic, number of logic gates, the maximum fan-out of each gate and the amount of wiring between the stages.

2.2.4 Prefix Based Adders

A prefix adder consists of 3 stages i.e, pre-computation stage, prefix network stage and post-computation stage as shown in Fig 2.7 [5-11].

![Figure 2.7 Block level diagram of a prefix adder](image)

The pre-computation stage computes the carry ‘Propagate’ and carry ‘Generate’ bits for each input pair as given below.

\[
\begin{align*}
\text{Generate}, \ g &= ab \\
\text{Propagate}, \ p &= a \oplus b
\end{align*}
\] (2.14)

(2.15)

The prefix network stage computes the final carries from the carry ‘Propagate’ and carry ‘Generate’ bits. Carry computation can be transformed to a prefix problem [5-9] using the
2.2 Review of Existing Adder Designs

associative operator ‘○’, which associates pairs of ‘Generate’ and ‘Propagate’ bits as given below:

\[(g, p) \circ (g', p') = (g + p \cdot g', p \cdot p')\]  \hspace{1cm} (2.16)

where \(g\) and \(g'\) represent the ‘Generate’ terms and \(p\) and \(p'\) represent the ‘Propagate’ terms. Using the operator ‘○’ consecutive ‘Propagate’ and ‘Generate’ pairs can be grouped to generate carry as follows:

\[C_i = (g_i, p_i) \circ (g_{i-1}, p_{i-1}) \circ \ldots \circ (g_1, p_1) \circ (g_0, p_0)\]  \hspace{1cm} (2.17)

Representing the operator ‘○’ as node ●, and signal pairs \((g, p)\) as edges of a graph, parallel prefix carry computation can be represented as graphs. One of the prefix networks, Kogge-Stone [9] represented as a graph is shown in Fig. 2.8. The white color node in the graph represents a feed through node with no logic (generally realized with a buffer in hardware).

![Figure 2.8 Example of a Kogge–Stone prefix adder](image)

The final post computation stage computes the final Sum from carry generated in prefix network stage. These designs are very efficient in terms of delay and area when compared to carry-select and carry look-ahead adders.

As operand size increases (32-bits and above) these prefix adders suffer from complexity in prefix network due to an increase in number of logic cells and wiring [5, 9]. This problem can be addressed with a hybrid adder (also called as a sparse adder) which is a combination of prefix and carry-select adders [12-17]. These adders consist of two segments, one being carry generation block (CGB) that has prefix network and the other the conditional sum computation block (SCB) that has carry-select adders shown in Fig 2.9. As seen from the figure in CGB, where a Kogge-Stone network structure is used, all ‘carry’’s are not computed as in prefix adders (shown in Fig. 2.8) but only a few (in this case \(C_3\) and \(C_7\)) are computed depending on the degree
2.2 Review of Existing Adder Designs

of sparseness where the degree of sparseness is the number of sums selected conditionally. For example, degree of sparseness 4 means that the carry will select four sum bits conditionally as shown in Fig. 2.9. Hence, all the carries are not required in CGB.

The SCB in general is implemented by using a carry select adder. As seen from the Fig. 2.9, appropriate sums in SCB are selected by the ‘carry’s generated in CGB using multiplexers (MUX). Carry-select adders are better suited for sparse adders with low sparseness of 2-bit and 4-bit. Tyagi [18] proposed a reduced area scheme carry-select adder which can be used in a SCB. An optimized implementation of a sparse adder with carry-select adder in SCB can be found in [12, 17].

![Figure 2.9 Example of a 8-bit Sparse adder with degree of sparseness as 4](image)

It is clear that sparse adders have simple carry generation block. But as the operand length increases, sparse adders also suffer from high fan-out and lateral wiring complexity in carry generation network as in a prefix network. For instance, when the CGB of a 8-bit Kogge-Stone Sparse adder shown in Fig. 2.9 is extended for 64-bit Kogge-Stone Sparse adder, shown in Fig. 2.10, it can be seen that the wiring complexity (i.e., congestion between wires) to generate carry signals is increased. Similarly when a 64-bit Sklansky based prefix network is used in CGB shown in Fig. 2.11 it can be noticed from the figure that the fan-out on carry signal ‘C31’ is high (signal ‘C31’ is an input to compute the higher bit ‘carry’s).
2.3 Design and Implementation of Efficient Sum Computation Block for Higher Bit Sparse Adders

While these drawbacks can be overcome by increasing the degree of sparseness, the SCB complexity however will increase. Moreover, the loading on Carry signal will increase further as the number of Sum bits in SCB increases. For example in an 8-bit SCB, the Carry signal has to drive eight MUXes to select the Sum which consume a large amount of area and power, thus limiting the usage of direct higher bit carry-select adder as SCBs. In this work a modified SCB is proposed and analyzed to address these problems.

**2.3 Design and Implementation of Efficient Sum Computation Block for Higher Bit Sparse Adders**

As discussed in section 2.2.4 earlier, the power, area and fan-out overheads limit the usage of carry-select adder in SCB as the degree of sparseness increases. The area overhead can be reduced by using prefix structure with late Carry-in concept proposed by Sklansky [4]. This late
2.3 Design and Implementation of Efficient Sum Computation Block for Higher Bit Sparse Adders

Carry-in concept or fastest input-carry processing is achieved by adding an extra row of node at the end of the prefix carry network as shown in Fig 2.12. This addition of extra node however increases the overall delay of the adder by one node stage. Any prefix structure can be preferred to implement the prefix carry network depending on the design requirement. However, the fan-out or loading on the Carry signals from the CGB is still a problem. In this work, the structure of prefix network and the late carry-in scheme are analyzed and a new structure is developed to address these problems.

Figure 2.12 Sklansky parallel prefix adder with late Carry-in

The proposed approach is to reduce the fan-out or loading on the late carry-in signals it is achieved by feeding the late Carry-in signal only for a few ‘carry’s. The remaining ‘carry’s are to be computed with these few ‘carry’s to generate all the ‘carry’s required for sum computation. The proposed technique is illustrated through an example by taking Han-Carlson prefix structure. This prefix structure is chosen because of its uniformity in fan-in and fan-out requirements as well as reduced number of nodes when compared to other prefix structures [9-11].

A 16-bit traditional Han-Carlson adder with late carry-in is shown in Figs 2.13(a). From the figure, it can be seen that the loading on the Cin signal is 16. To reduce this loading, the proposed technique is applied to this structure wherein the late carry-in signal (Cin) is fed only to odd ‘carry’s i.e., G1,0, G3,0, etc… Even ‘carry’s are than computed from the odd ‘carry’s. The modified Han-Carlson with late carry-in using the proposed technique is shown in Fig. 2.13 (b). It can be observed that a 16-bit modified adder has a fan-out requirement of only 9 compared to the traditional late carry-in prefix structure that has a fan-out of 16. Thus, the proposed technique results in the reduction of fan-out on the late carry-in signal.
2.3 Design and Implementation of Efficient Sum Computation Block for Higher Bit Sparse Adders

Figure 2.13 (a) 16-Bit Han-Carlson Adder with late Carry-in. (b) Modified 16-Bit Han-Carlson Adder with late Carry-in.

The gate-level realization of the nodes of the above structures is shown in Fig 2.14.
2.3 Design and Implementation of Efficient Sum Computation Block for Higher Bit Sparse Adders

The main limitation of the proposed technique is the uneven arrival of even and odd Sums. For example, as seen from the Fig. 2.15, Sum $S_2$ will be computed when the Generate term $G'_{1:0}$ arrives. Further, Sum $S_3$ has to wait for generate term $G'_{2:0}$ which depends on $G'_{1:0}$. This not only results in extra delay but also leads to different arrival times of the digits of final Sum. This issues is addressed in this work as described below.

From Fig. 2.15, the equation to compute the Sum digits $S_2$ and $S_3$ are as follows:

\[ S_2 = G'_{1:0} \oplus G_1 \]  
\[ S_3 = G'_{2:0} \oplus G_2 \]  

The generate signal $G'_{2:0}$ equation given in terms of $G'_{1:0}$ is as follows

\[ G'_{2:0} = G_2 + G'_{1:0} P_2 \]  

From equations 2.19 and 2.20, $S_3$ can be rewrite as

\[ S_3 = (G_2 + G'_{1:0} P_2) \oplus P_3 = (G_2 \oplus P_3) (G'_{1:0})' + ((G_2 + P_2) \oplus P_3)G'_{1:0} \]
2.3 Design and Implementation of Efficient Sum Computation Block for Higher Bit Sparse Adders

Equation 2.21 can be realized as shown in Fig. 2.16. Thus, the structure shown in Fig. 2.13 can be further modified using this block as shown in Fig. 2.17. It can be seen that the overall delay is reduced as well as the varied arrival times of different Sum digits is addressed. This technique can also be adopted for the design of higher bit sparse tree adders [19-21] as explained below.

![Figure 2.15 Modified 16-Bit Han-Carlson Adder with late Carry-in illustrating delay problem](image)

![Figure 2.16 Gate level implementation of Equation 2.21](image)
2.4 Design and Implementation of Higher Bit Sparse Adder

In Section 2.2.4, issues related to the implementations of higher bit sparse adders with degree of sparseness more than 4 such as increasing wiring complexity and loading on the carry-in signal have been explained. To address these problems a SCB has been proposed in Section 2.3. In this section, a 64-bit sparse adder is designed and implemented with a varying degree of sparseness of 8, 16 and 32-bit in order to verify the advantages of the proposed SCB structure stated earlier.

The CGB of 64-bit sparse adder with different degrees of sparseness mentioned above is shown in Fig 2.18 (a, b, c) [19-20]. From the figure, it can be observed that the CGB complexity has decreased as the degree of sparseness increases.

After the generation of ‘carry’s in CGB, the ‘sum’s are computed by using the SCB proposed in the previous section. The 16-bit SCB structure explained earlier can be used for a degree of sparseness 16. The same structure can also be extended for different bits to address different degrees of sparseness.

The SCB area can further be reduced by using some of the group ‘Generate’ and ‘Propagate’ terms that have already been computed in the CGB. If the intermediate Propagate and Generate terms generated at the end of the second stage of CGB, that is \((G_{[1:0]},P_{[1:0]})\),
2.4 Design and Implementation of Higher Bit Sparse Adder

\((G_{[3:2]}, P_{[3:2]})\) etc., are used for Sum computation in SCB, it will result in power and area reduction when compared to the existing sparse implementations. The proposed SCB with reduced cells is shown in Fig 2.19(a) and Fig 2.19(b) for 8-bit and 16-bit respectively [19-20]. The same can also be extended to 32-bit SCB.

Figure 2.18 Carry generation for 64-Bit (a) Sparse-8 (b) Sparse-16 and (c) Sparse-32 adders.
2.5 Simulation Results

From Fig 2.18 it can be seen that the SCBs have progressively lesser wiring and logic cell complexity with increasing sparseness while the corresponding CGBs have increasing complexity as seen from Fig 2.19. Thus, it can be generalized that for a sparse tree adder, the complexity of SCB is inversely proportional to the complexity of the CGB. Since the sparse adders provide the flexibility to control the Carry signal, these adders have application in the design of multi-precision adders [22-24].

2.5 Simulation Results

All adders have been described in Verilog HDL and simulated using Cadence Incisive Unified Simulator (IUS) v6.1 and are mapped on to the Synopsys 90nm generic Technology library, using Cadence RTL Compiler v7.1. The derived netlist was then passed to Cadence First Encounter XL v7.1 for floor-planning and routing.

The modified Han-Carlson Sum computation block for 64-bit Sparse-8, -16 and -32 has been compared with 64-bit Sparse-8, -16 and -32 Han-Carlson late Carry-in adder and 64-bit sparse-4 with conditional Sum adder [19-20]. Table 2.2 presents performance parameters such as area, power, delay and power-delay product for all the three designs. Also, Fig. 2.20 provides a graphical comparison of these parameters.
2.5 Simulation Results

Table 2.2 Area, power, delay and power-delay product for 64-bit adder with various sparseness

<table>
<thead>
<tr>
<th>64-Bit adder</th>
<th>Sparse-4 with conditional sum</th>
<th>Sparse-8 with Han-Carlson late carry-in</th>
<th>Sparse-8 with Modified Han-Carlson late carry-in</th>
<th>Sparse-16 with Han-Carlson late carry-in</th>
<th>Sparse-32 with Han-Carlson late carry-in</th>
<th>Sparse-32 with Modified Han-Carlson late carry-in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Area (um²)</td>
<td>4316 (100%)</td>
<td>4739 (109.8%)</td>
<td>3998 (92.63%)</td>
<td>5257 (121.80%)</td>
<td>4383 (101.55%)</td>
<td>5652 (130.95%)</td>
</tr>
<tr>
<td>Power (mW)</td>
<td>0.286 (100%)</td>
<td>0.281 (98.25%)</td>
<td>0.2471 (86.36%)</td>
<td>0.309 (108.04%)</td>
<td>0.2627 (91.60%)</td>
<td>0.3278 (114.62%)</td>
</tr>
<tr>
<td>Delay (ns)</td>
<td>1.101 (100%)</td>
<td>0.956 (86.83%)</td>
<td>0.969 (88.01%)</td>
<td>1.05 (95.37%)</td>
<td>1.03 (93.55)</td>
<td>0.89 (80.84%)</td>
</tr>
<tr>
<td>Power-Delay product (pJ)</td>
<td>0.3149 (100%)</td>
<td>0.2686 (85.29%)</td>
<td>0.2394 (75.87%)</td>
<td>0.3245 (103.05%)</td>
<td>0.2706 (86.03%)</td>
<td>0.2917 (92.63%)</td>
</tr>
</tbody>
</table>

Figure 2.20 (a)
2.5 Simulation Results

Figure 2.20 (b)

Figure 2.20 (c)
Table 2.2 and Fig. 2.20 provide a comparison of various design parameters for 64-bit adder with different degrees of sparseness. As can be seen, the proposed 64-bit adder with a sparseness of 8 involving the modified Han-Carlson adder performs better than other designs in terms of power (a reduction of 14%) and delay (a reduction of 12%) resulting in an overall reduction of 25% in power-delay product. Further, there is also a reduction of 8% in area. But, if delay is the only parameter important, then the design with a degree of sparseness 32 with modified Han-Carlson late Carry-in adder that results in a 20% reduction in delay can be used.

It can be observed from the above table and figure that the 64-bit adder with a sparseness of 16 and 32, while performing better than that with a sparseness of 4, do not perform as well as that with a sparseness of 8. Also the adder with sparseness 32 performs better than that with sparseness 16. This is because the 64-bit adder with a sparseness of 8 and 16, using either existing compound adder or late Carry-in adder, needs 6 carry merge stages to compute Carry and has a fan-out of 8 and 16 respectively in the critical path. However, the same adder with a sparseness of 32 needs 5 carry merge stages to compute carry and has a fan-out of 32 in the critical path. Hence, there is an increase in delay for sparse-16 adder when compared to sparse-8...
2.5 Simulation Results

or sparse-32. This is also applicable to the sparse adders with modified sum computation block [19-20].

An extended analysis has also been done for a 128-bit adder using the proposed technique. Table 2.3 and Fig 2.21 present data related to area, power, delay and power-delay product for sparse-16 and sparse-32 adders with modified Han-Carlson sum computation block.

Table 2.3 Area, power, delay and power-delay product for 128-bit adder with sparseness of 16 and 32

<table>
<thead>
<tr>
<th>128-Bit adder</th>
<th>Sparse-16 with Modified Han-Carlson late Carry-in</th>
<th>Sparse-32 with Modified Han-Carlson late Carry-in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Area(um²)</td>
<td>8775</td>
<td>9846</td>
</tr>
<tr>
<td>Power(mW)</td>
<td>0.5354</td>
<td>0.603</td>
</tr>
<tr>
<td>Delay(ns)</td>
<td>1.367</td>
<td>1.349</td>
</tr>
<tr>
<td>Power-Delay Product(pJ)</td>
<td>0.7319</td>
<td>0.813</td>
</tr>
</tbody>
</table>

Figure 2.21 (a)
2.5 Simulation Results

Figure 2.21 (b)

Power (mW)

Figure 2.21 (c)

Delay (ns)
2.6 Conclusions

Figure 2.21 (d)

Figure 2.21 Extended analysis of 128-bit adder using the proposed technique in terms of
(a) Area, (b) Delay (c) Power and Power-Delay Product

It can be seen from the above table and figure that the 128-bit adder with a sparseness of 16 performs better than the one with sparseness of 32.

2.6 Conclusions

In this chapter, novel designs for higher bit (64 & 128) sparse adders have been proposed. The increased complexity of the sum computation block at larger bit lengths has been compensated with alternate designs of carry generation block that results in reduced complexity. A detailed analysis of the 64 & 128-bit sparse adders with different degree of sparseness indicates that they perform better than the designs reported in literature.