Area–Delay–Power Efficient Carry-Select Adder

Name :M Jaya Mukhesh

Roll Number: 22ECB0B23

Abstract:

In this project, we conduct a thorough analysis of the logic operations within conventional Carry Select Adders (CSLA) and Binary to Excess-1 Converter (BEC)-based CSLAs. Our focus is on understanding data dependencies and identifying redundant logic operations. Through this analysis, we successfully eliminate redundant logic operations present in conventional CSLAs and propose a novel logic formulation for CSLAs.

Our proposed approach prioritizes the Carry Select (CS) operation before computing the final sum, deviating from the conventional method. We leverage bit patterns from two anticipating carry words (corresponding to cin = 0 and 1) along with fixed cin bits for optimizing CS and generation units' logic. This optimization results in a highly efficient CSLA design with significantly reduced area and delay compared to recently proposed BEC-based CSLAs.

Moreover, our CSLA design exhibits a notably smaller carry-output delay, positioning it as an excellent candidate for Square-Root (SQRT) CSLAs. Theoretical estimates indicate that our proposed SQRT-CSLA offers nearly 35% less Area-Delay-Product (ADP) compared to BEC-based SQRT-CSLAs, making it the top-performing option across various bit-widths.

Application-specific integrated circuit (ASIC) synthesis validates our findings, revealing that the BEC-based SQRT-CSLA design consumes 48% more ADP and 50% more energy compared to our proposed SQRT-CSLA, on average, across different bit-widths. This abstraction captures the essence of the original text, focusing on the analysis, optimization, and superior performance of the proposed CSLA design, especially in the context of square-root CSLA implementations.

1. **Instructions:**

Low-power, area-efficient, and high-performance Very Large Scale Integration (VLSI) systems find widespread application in portable and mobile devices, multistandard wireless receivers, and biomedical instrumentation. The cornerstone of an arithmetic unit within these systems is the adder. While the ripple carry adder (RCA) offers a simple design, its main drawback lies in the carry propagation delay (CPD). To mitigate this delay, carry look-ahead and carry select (CS) methods have been proposed.

A conventional carry select adder (CSLA) typically consists of a configuration of two RCAs, generating a pair of sum words and output-carry bits corresponding to anticipated input-carry (cin = 0 and 1). It then selects one from each pair for the final sum and final output carry. While CSLA exhibits reduced CPD compared to RCA, its dual RCA design isn't optimal. Several attempts have been made to address this issue. Kim and Kim introduced a design using one RCA and one add-one circuit, replacing one RCA with a multiplexer-based add-one circuit. Similarly, He et al. proposed a square-root (SQRT)-CSLA to implement large bit-width adders with reduced delay by connecting CSLAs of increasing size in a cascading structure.

Ramkumar and Kittur proposed a binary to Excess-1 Converter (BEC)-based CSLA, which requires fewer logic resources but has slightly higher delay compared to conventional CSLAs. Additionally, designs based on Common Boolean Logic (CBL) have been explored, offering reduced logic resource utilization but with longer CPD. To address this, a SQRT-CSLA based on CBL was proposed, albeit at the cost of increased logic resources and delay compared to BEC-based implementations.

Through analysis, it was observed that logic optimization depends on identifying redundant operations, while adder delay is influenced by data dependence. However, existing designs often optimize logic without considering data dependence. In response, this project conducted an analysis of logic operations in conventional and BEC-based CSLAs to understand data dependence and identify redundant logic operations. Building upon these insights, a novel logic formulation for CSLA was proposed, emphasizing data dependence and optimized carry generator (CG) and CS design. These ideas were instrumental in shaping the contributions of this project.

2. RELATED WORKS:

The Carry Select Adder (CSLA) comprises two crucial units: 1) the Sum and Carry Generator (SCG) unit and 2) the Sum and Carry Selection unit. Among these, the SCG unit consumes a significant portion of logic resources and contributes substantially to the critical path. To optimize the implementation of the SCG unit, various logic designs have been proposed.

In our study, we delve into the logic designs proposed for the SCG unit of both conventional and BEC-based CSLAs, as outlined in [6], using appropriate logic expressions. Our primary goal is to pinpoint redundant logic operations and uncover data dependencies within these designs. Consequently, we systematically eliminate all redundant logic operations and organize the sequence of logic operations based on their data dependencies.

1. Carry Select Adder Using Common Boolean Logic:

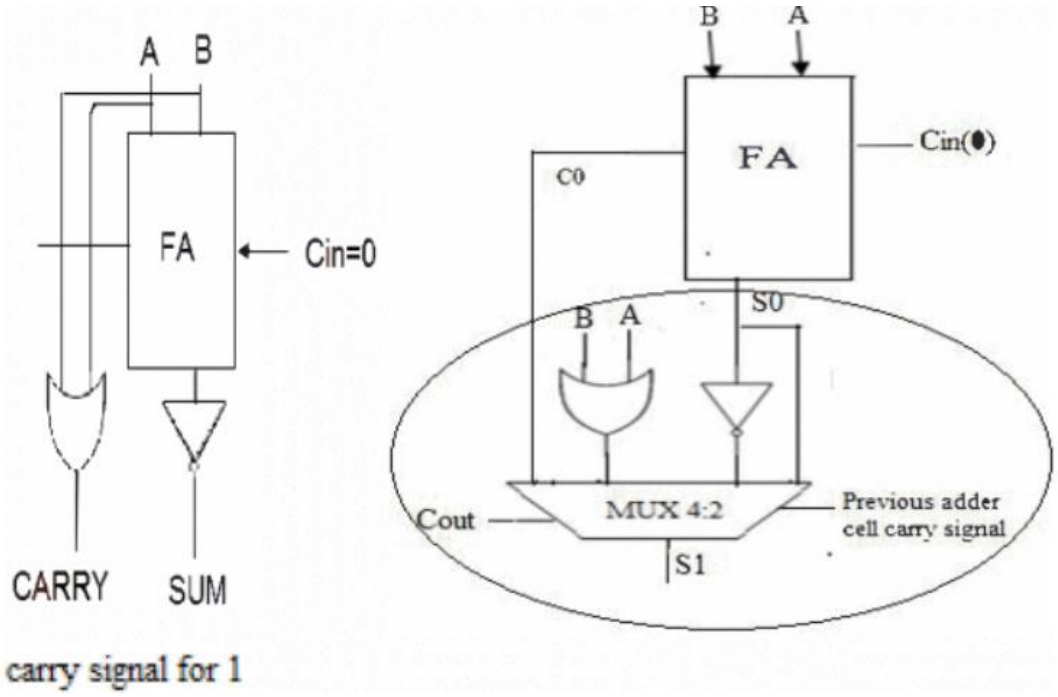
In this approach, we introduce an area-efficient carry select adder (CSLA) by leveraging common Boolean logic terms to eliminate duplicated adder cells found in conventional CSLAs. This optimization not only reduces transistor counts but also contributes to lower power consumption.

Through a detailed analysis of the truth table of a single-bit full adder, we discover that the output of the summation signal when the carry-in signal is logic "0" is the inverse of itself when the carry-in signal is logic "1". This insight is depicted by two dotted circles in the truth table (Fig. 7).

Utilizing this understanding, our proposed CSLA design, illustrated in Fig. 8, incorporates shared common Boolean logic terms in summation generation. By implementing just one OR gate with one INV gate, we efficiently generate the Carry signal and summation signal pair. Once the carry-in signal is determined, the appropriate carry-out output is selected based on its logic state.

Comparing our proposed CSLA with the Modified Carry Select Adder, we observe that our design exhibits slightly higher speed, albeit nearly equivalent to that of the Regular CSLA. The delay time in our proposed design is directly proportional to the bit number N, similar to other CSLAs. However, the delay time of the multiplexer in our design is shorter than that of a full adder.

C_{in}	A	B	S0	C0
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



Overall, our proposed CSLA stands out for its area efficiency and low power consumption, while maintaining speed comparable to the Regular CSLA.

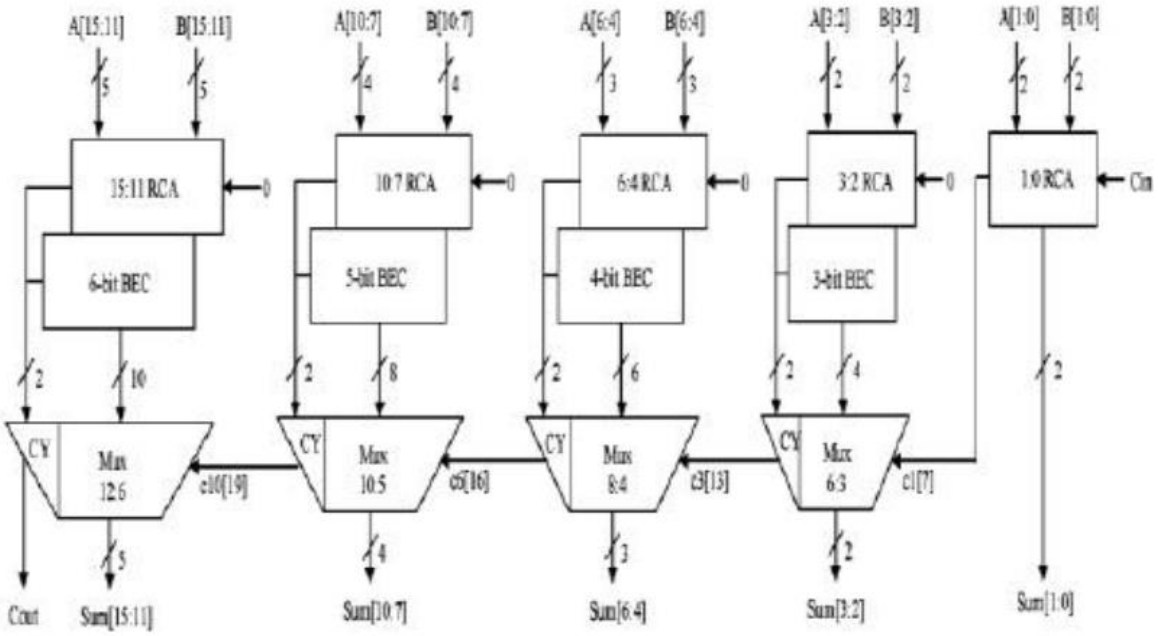
2. Carry select adder using BEC and RCA:

The Carry Select Adder (CSLA) is a crucial component in computational systems, aimed at mitigating the issue of carry propagation delay by generating multiple carries independently and then selecting the appropriate carry to generate the sum. However, conventional CSLAs suffer from inefficiency in terms of area utilization, as they employ multiple pairs of Ripple Carry Adders (RCAs) to generate partial sums and carries, which are then selected using multiplexers (mux).

The core concept of our work is to enhance the efficiency of CSLA by replacing RCAs with Binary to Excess-1 Converters (BECs), aiming for improved speed and reduced power consumption. Our paper presents qualitative evaluations of CSLA with and without BEC architectures. We specifically focus on writing VERILOG (Hardware Description Language) code for Carry Skip and Carry Select adders to highlight the performance characteristics common to these classes of adders.

Our findings indicate that implementing CSLA with BEC leads to improved efficiency in terms of delay time and power consumption. The primary

advantage of employing BEC logic lies in its ability to achieve comparable functionality with fewer logic gates compared to the traditional n-bit Full Adder (FA) structure.



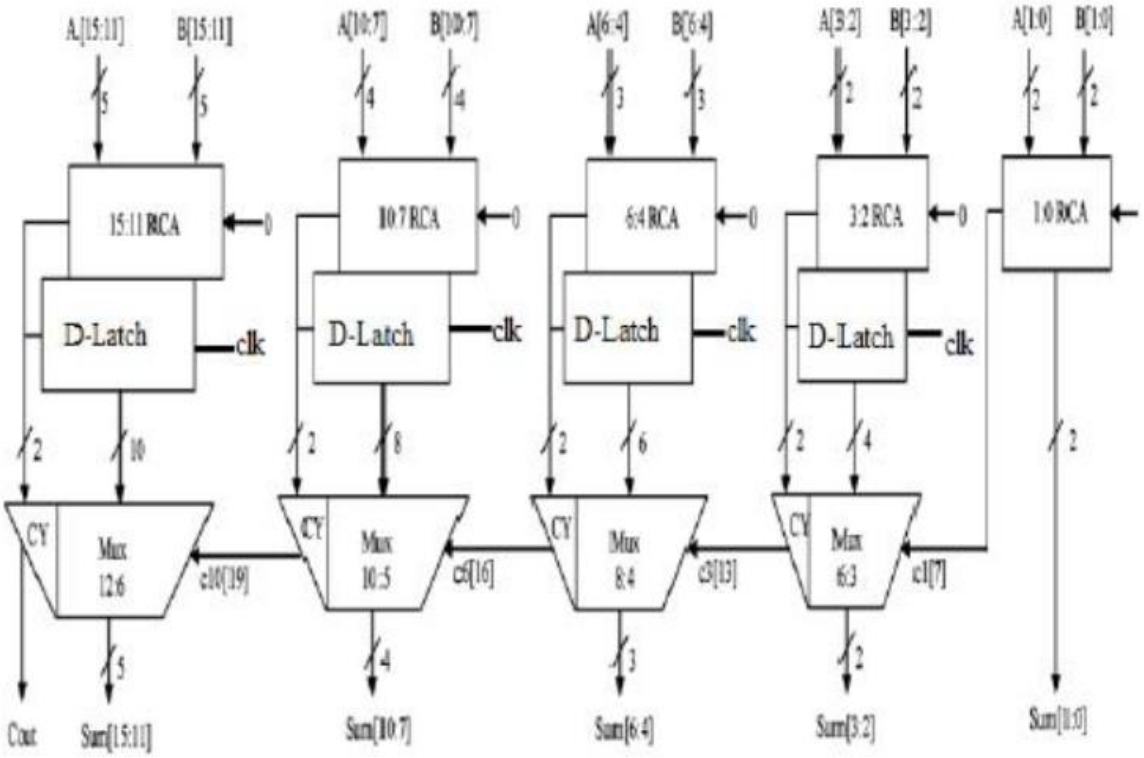
3. Lower Power High Performance Carry Select Adder:

In the proposed structure of the Carry Select Adder (CSA), an additional D-latch is introduced to reduce delay at the expense of a slight increase in area. However, the trade-off results in enhanced speed performance.

This 16-bit CSA architecture, incorporating D-latches, is organized into five clusters of bit word sizes, ranging from n-bit RCA [5] to D-latches of 3b, 4b, 5b, and 6b word sizes concurrently. Unlike the traditional CSA employing two separate adders, our method utilizes only one adder, effectively reducing area, power consumption, and delay. Additionally, each of the two additions is executed within a single clock cycle.

In this 16-bit adder configuration, the Least Significant Bit (LSB) operates as a ripple carry adder, spanning 2 bits. The Upper Half Adder, constituting the Most Significant Bit (MSB) and being 14-b wide, functions based on clock principles. Upon the clock signal going high, the addition for carry input zero is performed, while during a low clock phase, the carry is assumed to be zero, and the addition

is stored within the adder itself. The carry-out from the preceding stage, i.e., the least significant bit adder, serves as the control signal for the multiplexer to select the final output carry and sum of the 16-bit adder [6] [7]. If the actual carry input is one, then the computed sum and carry latch are accessed; otherwise, the MSB adder is accessed. The output carry, Cout, is produced accordingly.



4. An Efficient 64-Bit Carry Select Adder With Less Delay And Reduced Area Application:

In this paper, we present an innovative approach aimed at enhancing the efficiency of Square-Root Carry Select Adder (SQRT CSLA) architecture by strategically addressing area and delay considerations. Our approach centers on the substitution of Ripple Carry Adders (RCAs) with Binary to Excess-1

Converters (BECs) within the structure, thereby reducing the overall number of gates. The comparative analysis of results illustrates that the modified SQRT CSLA exhibits a marginally larger area footprint for lower-order bits, which progressively diminishes for higher-order bits. However, the significant reduction in delay achieved with the modified SQRT CSLA stands out as a notable advantage. This outcome underscores the potential of our approach as a compelling alternative for adder implementations across diverse processor designs. It is crucial to note that while our approach proves highly effective for lower-order bits, its feasibility diminishes for higher-order bits such as 128 or 256. This limitation arises due to the inherent complexities associated with larger bit widths, wherein the structural modifications may not yield the same level of efficiency improvement. Nonetheless, for processors and applications where lower-order bit processing is predominant, our modified SQRT CSLA emerges as a promising solution, offering a balance between area optimization and delay reduction.

III. **METHODOLOGY**:

Conventional Carry Select Adder:

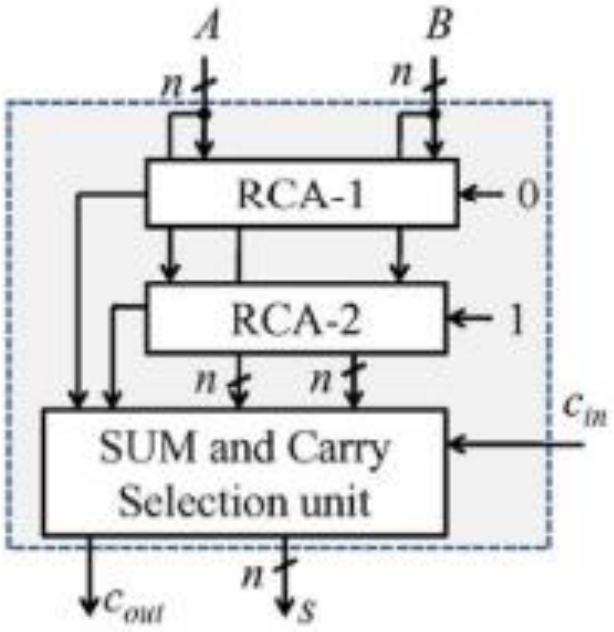
The Carry Select Adder (CSLA) leverages Ripple Carry Adders (RCAs) to compute sum and carry values using initial carry assumptions of 0 and 1, respectively, ahead of the actual carry's arrival. Specifically, the upper RCA operates with an initial carry value of logic "0," while the lower RCA begins with a carry value of logic "1." A multiplexer selects the output from the "carry 0" path if the preceding carry is logic '0,' or the output from the "carry 1" path if the preceding carry is logic '1.' This mechanism enables the use of the actual carry to determine the sum and carry outputs through multiplexing.

The carry-select adder, a vital component in digital arithmetic circuits, combines ripple-carry adders and multiplexers for efficient addition of two n-bit numbers. This method involves two stages of addition: one assuming the carry-in as zero and the other assuming it as one. The results are then evaluated using a multiplexer to select the correct sum and carry-out based on the actual carry-in value.

In the uniform block size scenario, the optimal delay occurs when each carry-select block has a size of $\lfloor\sqrt{n}\rfloor$, minimizing the overall delay. For variable block sizes, the delay from addition inputs A and B to the carry-out should match that of the multiplexer chain leading into it, ensuring precise calculation timing. This approach, with an $O(\sqrt{n})$ delay, is derived from uniform sizing, where the number of full-adder elements per block equals the square root of the number of bits being added, ensuring balanced performance.

In CSLA, each RCA pair can independently compute the sum value in parallel before the arrival of the previous stage's carry. Consequently, the critical path of an N-bit adder is significantly reduced. Compared to conventional adders, where the critical path consists of an N-bit carry propagation path and one sum generation stage, CSLA offers a critical path of (N/L)-bit carry propagation path and L stages of multiplexers, along with one sum generation stage in the N-bit CSLA. Here, L represents the number of stages in CSLA. Given that L is substantially smaller than N and the delay introduced by the multiplexer is less

than that of a full adder, CSLA demonstrates significantly reduced delay compared to RCA.



However, it's important to note that CSLA incurs increased power consumption and cost due to the presence of replicated hardware in each stage. This duplication of hardware across stages contributes to a higher power consumption and overall cost.

$$s_0^0(i) = A(i) \oplus B(i) \quad c_0^0(i) = A(i) \cdot B(i)$$
$$s_1^0(i) = s_0^0(i) \oplus c_1^0(i-1)$$
$$c_1^0(i) = c_0^0(i) + s_0^0(i) \cdot c_1^0(i-1) \quad c_{\text{out}}^0 = c_1^0(n-1)$$
$$s_0^1(i) = A(i) \oplus B(i) \quad c_0^1(i) = A(i) \cdot B(i)$$
$$s_1^1(i) = s_0^1(i) \oplus c_1^1(i-1)$$
$$c_1^1(i) = c_0^1(i) + s_0^1(i) \cdot c_1^1(i-1) \quad c_{\text{out}}^1 = c_1^1(n-1)$$

Binary To Excess-1 Convertor Carry Select Adder:

The BEC-based CSLA utilizes Ripple Carry Adders (RCAs) to compute sum and carry values, assuming an initial carry of 0. Subsequently, the BEC unit processes the carry output (c0) from the preceding RCA and generates an (n + 1)-bit excess-1 code. The

BEC operates by employing XOR and AND gates in a configuration where each AND gate serves as an internal carry generator circuit, producing the carry for consecutive sum bits.

In this process, the first sum bit (S0) from the initial stage of the RCA is inverted, and the inverted bit is treated as the first sum bit for the subsequent add-one operation. The first sum bit of the RCA's initial stage is then ANDed with the second sum bit to generate the first internal carry bit. This carry bit is combined with the third sum bit, continuing the process of carry generation from the sum bits of the initial stage and the summation of consecutive internally generated carry bits with previous sum bits.

Consequently, we obtain the final sum bits for both scenarios: when the carry is 0, calculated by the initial stage RCA, and when the carry is 1, computed using the BEC logic block. These two sequences are fed into a final stage MUX array, which selects one of the sequences based on the previous stage's carry value (0 or 1).

This method offers several design advantages:

(i) The need for calculating s01 in the Sum and Carry Generator (SCG) unit is eliminated.

(ii) Only an n-bit select unit is required instead of (n + 1) bits.

(iii) The output-carry delay is minimized, resulting in improved overall performance.

$$s_1^1(0) = \overline{s_1^0}(0) \quad c_1^1(0) = s_1^0(0)$$
$$s_1^1(i) = s_1^0(i) \oplus c_1^1(i-1)$$
$$c_1^1(i) = s_1^0(i) \cdot c_1^1(i-1)$$
$$c_{\text{out}}^1 = c_1^0(n-1) \oplus c_1^1(n-1)$$

This method offers several design advantages:

1) Elimination of the need to calculate s0

1 in the Sum and Carry Generator (SCG) unit.

2) Requirement for an n-bit select unit instead of (n + 1) bits.

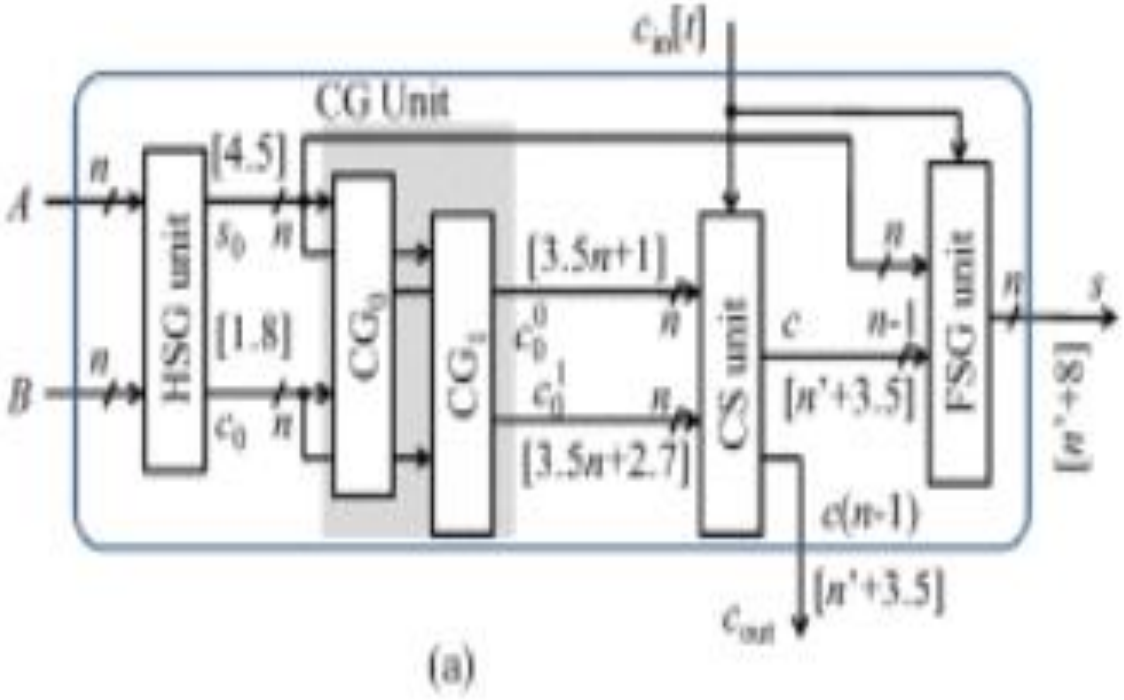
3) Reduced output-carry delay.

By removing redundant logic operations and reorganizing logic expressions based on their dependencies, we propose the following logic formulation for the CSLA:

s0(i) = A(i) ⨁ B.

$$
\begin{aligned}
&s_0(i) = A(i) \oplus B(i) \quad c_0(i) = A(i) \cdot B(i) \\
&c_1^0(i) = c_1^0(i-1) \cdot s_0(i) + c_0(i) \quad \text{for } \left(c_1^0(0) = 0\right) \\
&c_1^1(i) = c_1^1(i-1) \cdot s_0(i) + c_0(i) \quad \text{for } \left(c_1^1(0) = 1\right) \\
&c(i) = c_1^0(i) \quad \text{if } (c_{\text{in}} = 0) \\
&c(i) = c_1^1(i) \quad \text{if } (c_{\text{in}} = 1)
\end{aligned}
$$

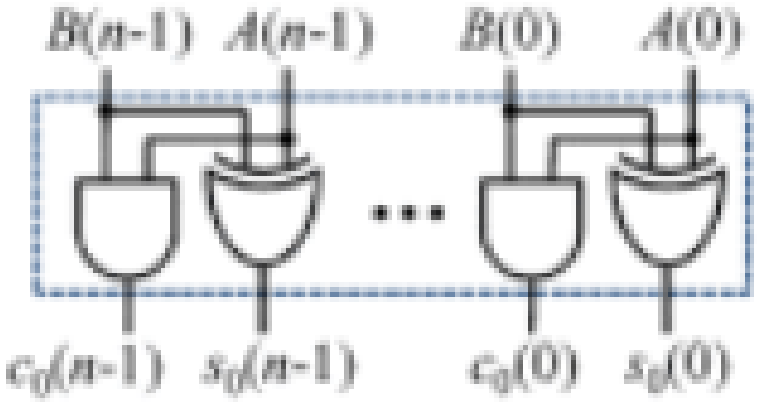
PROPOSED ADDER DESIGN:



As shown in the RCA calculates n-bit *sum* S_{01} and C_{0out} corresponding to $C_{in}=0$. The BEC unit receives S_{01} and C_{0out} from the RCA and generates (n+1)-bit excess-1 code. The most significant bit (MSB) of BEC represents C_{1out}, in which n least significant bits (LSBs) represent S_{11}. The logic expressions of the RCA are the same as those given in given equations. The logic expressions of the BEC unit of the n-bit BEC-based CSLA are given as

$$s_1^1(0) = \overline{s_1^0(0)} \quad c_1^1(0) = s_1^0(0)$$
$$s_1^1(i) = s_1^0(i) \oplus c_1^1(i-1)$$
$$c_1^1(i) = s_1^0(i) \cdot c_1^1(i-1)$$
$$c_{\text{out}}^1 = c_1^0(n-1) \oplus c_1^1(n-1)$$

HSG Module:



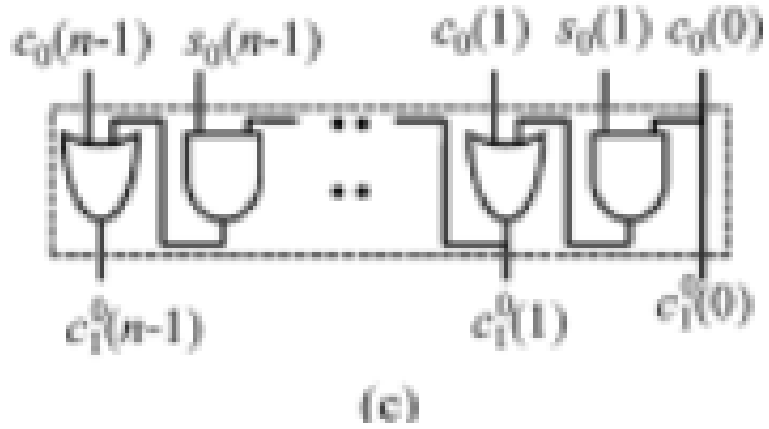
It gives the half sum of the adder.

Code:

```
module FSG_UNIT(
input [15:0]A,
input [15:0]B,
input Cin,
output [15:0]S,
output Cout
);
wire [15:0] S0;
wire [15:0] C;
HSG uut(.A(A),.B(B),.S0(S0));
CS_UNIT uut1(.A(A),.B(B),.C(C),.Cin(Cin));
assign S[0] = S0[0] ^ Cin;
assign S[1] = S0[1] ^ C[0];
assign S[2] = S0[2] ^ C[1];
assign S[3] = S0[3] ^ C[2];
assign S[4] = S0[4] ^ C[3];
```

```
assign S[5] = S0[5] ^ C[4];
assign S[6] = S0[6] ^ C[5];
assign S[7] = S0[7] ^ C[6];
assign S[8] = S0[8] ^ C[7];
assign S[9] = S0[9] ^ C[8];
assign S[10] = S0[10] ^ C[9];
assign S[11] = S0[11] ^ C[10];
assign S[12] = S0[12] ^ C[11];
assign S[13] = S0[13] ^ C[12];
assign S[14] = S0[14] ^ C[13];
assign S[15] = S0[15] ^ C[14];
assign Cout=C[15];
endmodule
```

CG0 MODULE:

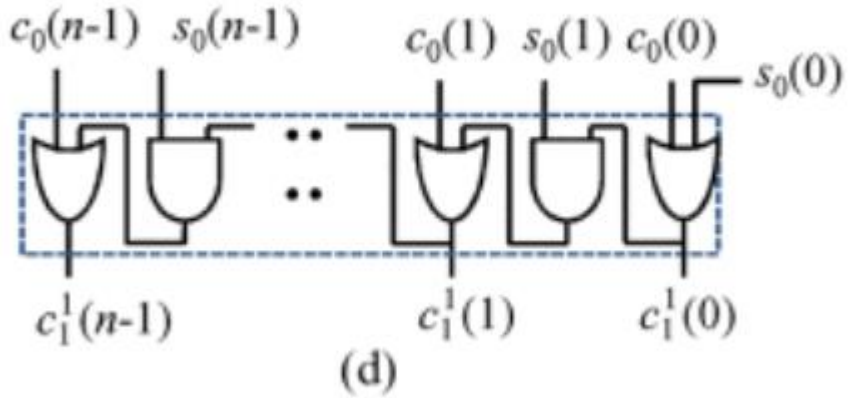


(c)

```
module CG0_UNIT(A,B,C01);
input [15:0]A;
input [15:0]B;
output [15:0] C01;
wire [15:0] S0,C0;
HSG UUT(.A(A),.B(B),.C0(C0),.S0(S0));
assign C01[0] = C0[0];
assign C01[1] = C01[0] & S0[1] | C0[1];
assign C01[2] = C01[1] & S0[2] | C0[2];
assign C01[3] = C01[2] & S0[3] | C0[3];
assign C01[4] = C01[3] & S0[4] | C0[4];
assign C01[5] = C01[4] & S0[5] | C0[5];
assign C01[6] = C01[5] & S0[6] | C0[6];
assign C01[7] = C01[6] & S0[7] | C0[7];
assign C01[8] = C01[7] & S0[8] | C0[8];
assign C01[9] = C01[8] & S0[9] | C0[9];
assign C01[10] = C01[9] & S0[10] | C0[10];
```

```
assign C01[11] = C01[10] & S0[11] | C0[11];
assign C01[12] = C01[11] & S0[12] | C0[12];
assign C01[13] = C01[12] & S0[13] | C0[13];
assign C01[14] = C01[13] & S0[14] | C0[14];
assign C01[15] = C01[14] & S0[15] | C0[15];
endmodule
```

CG1 MODULE:



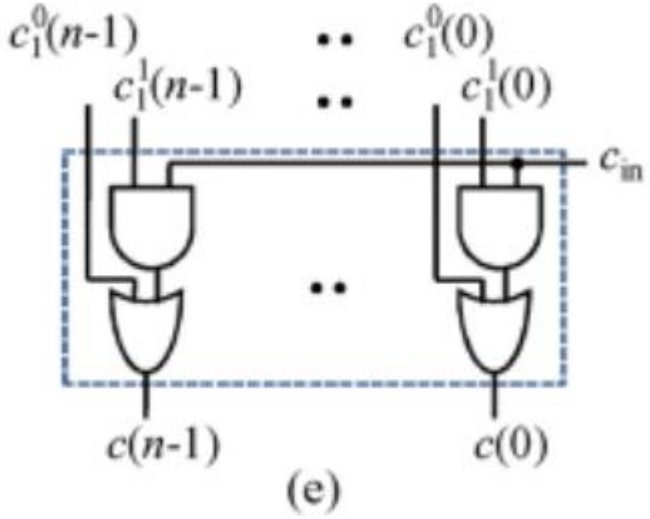
(d)

Code:

```
module CG1_UNIT(
input [15:0]A,
input [15:0]B,
wire [15:0]S0,
wire[15:0]C0,
output [15:0]C11
 );
HSG uut(.A(A),.B(B),.C0(C0),.S0(S0));
assign C11[0] = C0[0]|S0[0];
assign C11[1] = C11[0] & S0[1] | C0[1];
assign C11[2] = C11[1] & S0[2] | C0[2];
assign C11[3] = C11[2] & S0[3] | C0[3];
assign C11[4] = C11[3] & S0[4] | C0[4];
```

```
assign C11[5] = C11[4] & S0[5] | C0[5];
assign C11[6] = C11[5] & S0[6] | C0[6];
assign C11[7] = C11[6] & S0[7] | C0[7];
assign C11[8] = C11[7] & S0[8] | C0[8];
assign C11[9] = C11[8] & S0[9] | C0[9];
assign C11[10] = C11[9] & S0[10] | C0[10];
assign C11[11] = C11[10] & S0[11] | C0[11];
assign C11[12] = C11[11] & S0[12] | C0[12];
assign C11[13] = C11[12] & S0[13] | C0[13];
assign C11[14] = C11[13] & S0[14] | C0[14];
assign C11[15] = C11[14] & S0[15] | C0[15];
endmodule
```

CS Module:



(e)

Code

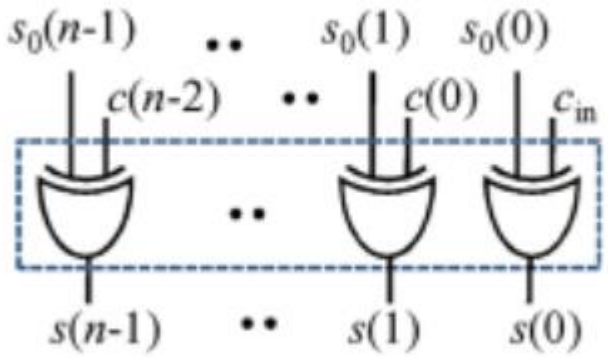
```
module CS_UNIT(
input [15:0]A,
input [15:0]B,
input Cin,
wire [15:0] C01, C11,
output [15:0] C
);
CG0_UNIT uut(.A(A),.B(B),.C01(C01));
CG1_UNIT uut1(.A(A),.B(B),.C11(C11));
assign C[0] = (Cin == 1'b0) ? C01[0] : C11[0];
assign C[1] = (Cin == 1'b0) ? C01[1] : C11[1];
assign C[2] = (Cin == 1'b0) ? C01[2] : C11[2];
assign C[3] = (Cin == 1'b0) ? C01[3] : C11[3];
assign C[4] = (Cin == 1'b0) ? C01[4] : C11[4];
```

```
assign C[5] = (Cin == 1'b0) ? C01[5] : C11[5];
assign C[6] = (Cin == 1'b0) ? C01[6] : C11[6];
assign C[7] = (Cin == 1'b0) ? C01[7] : C11[7];
assign C[8] = (Cin == 1'b0) ? C01[8] : C11[8];
assign C[9] = (Cin == 1'b0) ? C01[9] : C11[9];
assign C[10] = (Cin == 1'b0) ? C01[10] : C11[10];
assign C[11] = (Cin == 1'b0) ? C01[11] : C11[11];
assign C[12] = (Cin == 1'b0) ? C01[12] : C11[12];
assign C[13] = (Cin == 1'b0) ? C01[13] : C11[13];
assign C[14] = (Cin == 1'b0) ? C01[14] : C11[14];
assign C[15] = (Cin == 1'b0) ? C01[15] : C11[15];
endmodule
```

FSG UNIT:

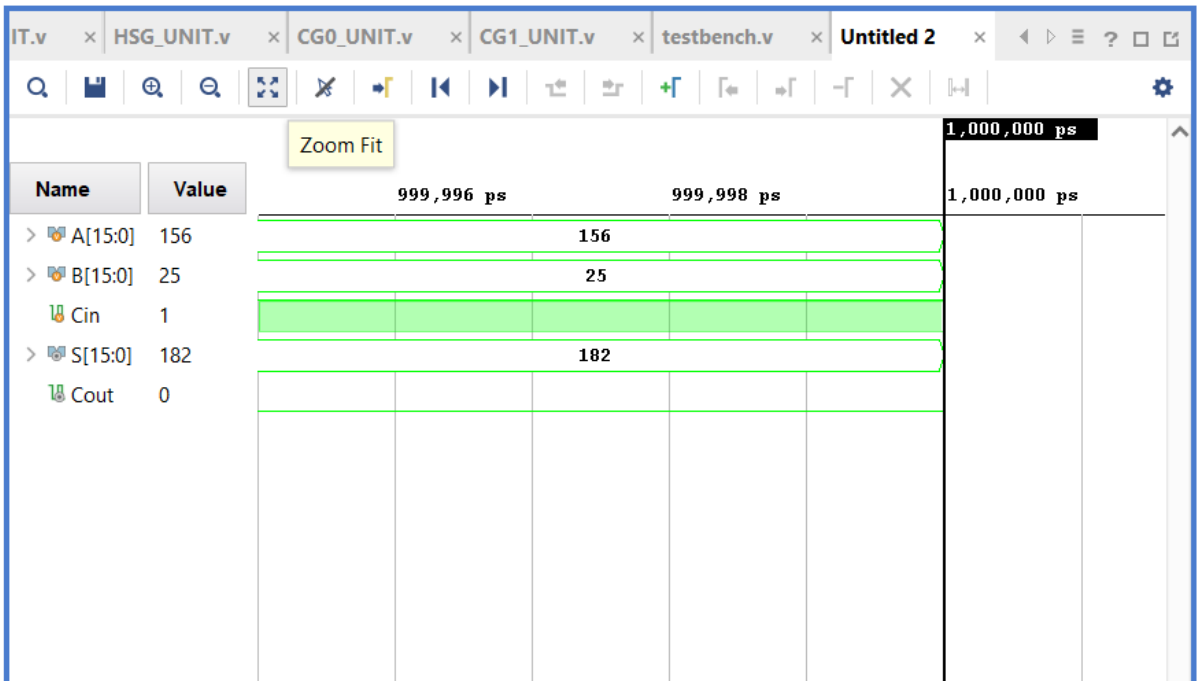
```
module FSG_UNIT(
input [15:0]A,
input [15:0]B,
input Cin,
output [15:0]S,
output Cout
);
wire [15:0] S0;
wire [15:0] C;
HSG uut(.A(A),.B(B),.S0(S0));
CS_UNIT uut1(.A(A),.B(B),.C(C),.Cin(Cin));
assign S[0] = S0[0] ^ Cin;
assign S[1] = S0[1] ^ C[0];
assign S[2] = S0[2] ^ C[1];
assign S[3] = S0[3] ^ C[2];
assign S[4] = S0[4] ^ C[3];
assign S[5] = S0[5] ^ C[4];
assign S[6] = S0[6] ^ C[5];
assign S[7] = S0[7] ^ C[6];
assign S[8] = S0[8] ^ C[7];
assign S[9] = S0[9] ^ C[8];
assign S[10] = S0[10] ^ C[9];
assign S[11] = S0[11] ^ C[10];
assign S[12] = S0[12] ^ C[11];
```

```
assign S[13] = S0[13] ^ C[12];
assign S[14] = S0[14] ^ C[13];
assign S[15] = S0[15] ^ C[14];
assign Cout=C[15];
endmodule
```



(f)

OUTPUT:


IT.v
HSG_UNIT.v
CG0_UNIT.v
CG1_UNIT.v
testbench.v
Untitled 2
Zoom Fit
1,000,000 ps
Name
Value
999,996 ps
999,998 ps
1,000,000 ps
A[15:0]
156
156
B[15:0]
25
25
Cin
1
S[15:0]
182
182
Cout
0