İSTANBUL TEKNİK ÜNİVERSİTESİ

ELEKTRİK ELEKTRONİK FAKÜLTESİ

DIGITAL SYSTEM DESIGN APPLICATIONS

(EHB 436E)

Carry Select Adder Report

Hasan Emre AYDEMİR

1. Introduction

Addition is one of the most fundamental operations in digital systems, and its performance directly affects the speed of many arithmetic units such as multipliers, ALUs, DSP blocks, and processors. In modern 32-bit and 64-bit architectures, wide adders must be designed to achieve low delay while keeping area and power consumption reasonable.

The Carry Select Adder (CSA) is a well-known high-performance adder architecture that improves speed by reducing the carry-propagation bottleneck. Instead of waiting for a single carry to ripple through all bit positions, the CSA computes two possible upper-word results in parallel and selects the correct one based on the actual carry-out of the lower block.

In this project, a 32-bit Carry Select Adder is designed to support signed integer addition and subtraction. The circuit includes two 32-bit inputs (A and B), a 32-bit output (SUM), a carry-out signal, and an overflow indicator. The goal is to design the architecture, implement it in Verilog HDL, verify its functionality using a testbench, and evaluate its performance in terms of logic utilization and timing.

2. Background and Theoretical Basis

Binary adders form the foundation of arithmetic logic design. The simplest form, the Ripple Carry Adder (RCA), has limited performance because each bit must wait for the previous carry to be resolved. This results in a delay proportional to the adder width. To overcome this limitation, several faster adder architectures have been proposed, including Carry Look-Ahead Adders (CLA), Carry Skip Adders, Parallel Prefix Adders, and Carry Select Adders.

The Carry Select Adder (CSA) improves addition speed by dividing the word into two or more blocks. The lower block is computed normally, while the upper blocks are calculated twice in parallel—once assuming the incoming carry is ‘0’ and once assuming it is ‘1’. After the actual carry from the lower block becomes available, a multiplexer selects the correct precomputed result. This parallel computation significantly shortens the critical path compared to ripple-carry structures.

Signed arithmetic uses the two’s complement representation, where negative values are encoded by inverting the bits and adding one. For subtraction, the operand B is transformed into its two’s complement form and added to A:

$$\mathbf{A - B = A + (\approx B + 1)}$$

Overflow in signed addition occurs when the computed result exceeds the representable range of 32-bit two's complement numbers. This condition is detected by checking whether the sign bits of the operands and the result satisfy:

$$\text{Overflow} = (A_{31} = B_{31}) \wedge (SUM_{31} \neq A_{31})$$

This theoretical foundation explains how the CSA reduces delay, how signed arithmetic is performed, and how overflow is detected in the final design.

3. Architecture of the 32-bit Carry Select Adder (CSA)

The 32-bit Carry Select Adder (CSA) is designed using a hierarchical and modular structure to improve speed while maintaining a low implementation cost. The architecture is divided into three main parts: the 4-bit Carry Look-Ahead (CLA) blocks, the 16-bit adder formed by cascading these CLA units, and the final 32-bit CSA structure, which uses parallel computation and multiplexer-based selection to reduce carry propagation delay. This section explains the functionality and interconnection of these components.

3.1. 4-bit CLA as the Basic Building Block

The smallest computational element in the adder is the 4-bit Carry Look-Ahead Adder (CLA). This block computes a 4-bit sum and generates a carry-out by using propagate and generate signals. These signals allow the carry to be computed without waiting for ripple propagation, giving the CLA block significantly higher speed than a traditional ripple-carry chain. Because of this advantage, the 4-bit CLA serves as an efficient building block for constructing wider adders.

In the design, four 4-bit CLAs are cascaded to form a complete 16-bit adder. Each CLA receives its corresponding slice of input bits A and B, and each block outputs a carry to the next stage. The structure ensures modularity, allowing easy extension to higher bit widths.

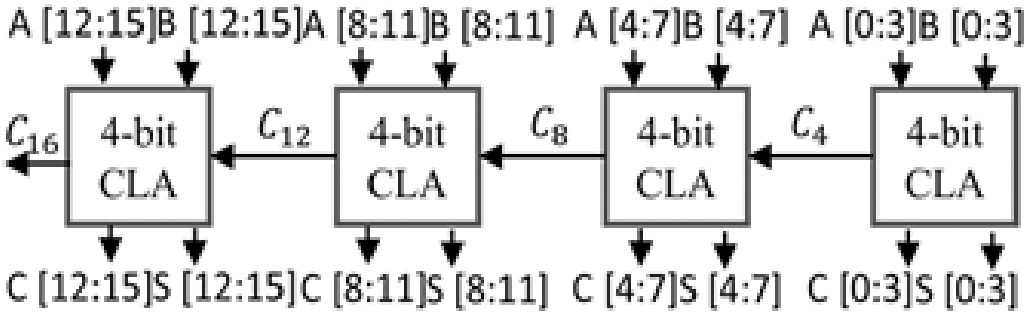

Figure 1. Illustrates this 16-bit structure built from four 4-bit CLA blocks

Figure 1. 16-bit adder constructed from four 4-bit CLA units. Each unit generates a 4-bit partial sum and forwards its carry to the next stage.

3.2. Overall Structure of the 32-bit Carry Select Adder

The full 32-bit CSA is formed by combining one 16-bit adder and two parallel 16-bit adders. The lower 16 bits are computed using a standard CLA-based 16-bit adder. This block produces:

- a lower 16-bit sum, and
- a carry-out signal C_{16}, which becomes the select signal for the upper section.

The key idea of the CSA architecture is to pre-compute the upper 16 bits twice:

1. once assuming carry-in = 0
2. once assuming carry-in = 1

Both upper adders operate in parallel. Since these two computations run simultaneously, the architecture avoids the long delay normally caused by waiting for the actual carry from the lower bits.

After the lower carry C_{16} is determined, a 2:1 multiplexer selects the correct upper 16-bit result. This greatly reduces the critical path delay, making the CSA faster than ripple-carry or single-path CLA adders of equal bit width.

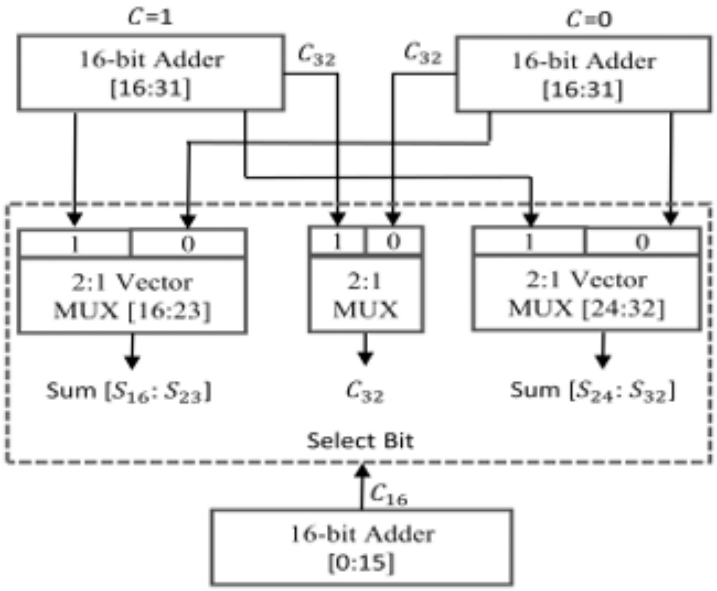

Figure 2. Shows the complete 32-bit CSA structure with its lower and upper 16-bit sections, and the multiplexer stage that performs the final selection.

Figure 2. Block diagram of the 32-bit Carry Select Adder (CSA). The lower 16-bit adder generates C_{16}, while two parallel 16-bit adders compute the upper word for carry-in values 0 and 1. A 2:1 multiplexer selects the correct upper result.

3.3. Multiplexer Logic for Upper-Word Selection

The final selection between the two precomputed upper results is performed by a transmission-gate based 2:1 multiplexer, which chooses the correct output according to the value of C_{16}. Transmission-gate multiplexers offer low delay and full-swing output, making them suitable for high-speed arithmetic circuits.

If $C_{16} = 0$, the upper sum generated with carry-in = 0 is selected. If $C_{16} = 1$, the upper sum generated with carry-in = 1 is selected.

This selection occurs instantly after C_{16} is known, completing the 32-bit addition operation.

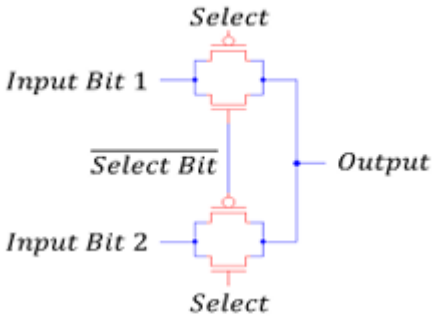

Fig. 9. TG 2:1 MUX used in proposed and conventional CSA.

TABLE I. TRUTH TABLE OF G_i AND P_i

Input		Output	
A_i	B_i	G_i	P_i
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Figure 3. Shows the TG-based 2:1 multiplexer used in the CSA.

Figure 3. Transmission-gate 2:1 multiplexer used to select between the two precomputed upper sums in the CSA.

4. Top Module Code

```
(* DONT_TOUCH = "TRUE" *)
module CSA32_top (
    input  wire        SUB,
    input  wire [31:0] A,
    input  wire [31:0] B,
    output wire [31:0] SUM,
    output wire        COUT,
    output wire        OVERFLOW
);


    wire [31:0] B_mod = SUB ? (~B + 32'd1) : B;


    wire [32:0] FULL = {1'b0, A} + {1'b0, B_mod};

    assign SUM  = FULL[31:0];
    assign COUT = FULL[32];


    assign OVERFLOW = (A[31] == B_mod[31]) && (SUM[31] != A[31]);

endmodule
```

5. 100 Random Input

```
module generate_stimulus;

    integer fd;
    integer i;

    reg  signed [31:0] A;
    reg  signed [31:0] B;
    reg  signed [31:0] A_adj;
    reg  signed [31:0] B_adj;
    reg  signed [31:0] SUM_exp;
    reg                COUT_exp;
    reg                OVERFLOW_exp;

    reg [32:0] FULL;

    initial begin
        fd = $fopen("stimulus_bin.txt", "w");
        if (fd == 0) begin
            $display("ERROR: Cannot open stimulus file.");
            $finish;
        end

        for (i = 0; i < 100; i = i + 1) begin

            A = $urandom;
            B = $urandom;


            B_adj = B;


            FULL     = {1'b0, A} + {1'b0, B_adj};
            SUM_exp  = FULL[31:0];
            COUT_exp = FULL[32];


            OVERFLOW_exp = (A[31] == B_adj[31]) && (SUM_exp[31] !=
A[31]);


            $fwrite(fd, "%b %b %b %b %b\n",
                    A, B, SUM_exp, COUT_exp, OVERFLOW_exp);
        end

        $fclose(fd);
        $display("stimulus_bin.txt created successfully.");
        $finish;
    end

endmodule
```

6. Testbench Code

```
`timescale 1ns/1ps

module tb_CSA32_top;


    reg         SUB;
    reg  [31:0] A;
    reg  [31:0] B;
    wire [31:0] SUM;
    wire        COUT;
    wire        OVERFLOW;


    reg  [31:0] SUM_exp;
    reg         COUT_exp;
    reg         OF_exp;

    integer fin;
    integer fout;
    integer r;
    integer vec;
    reg     status;

    CSA32_top dut (
        .SUB      (SUB),
        .A        (A),
        .B        (B),
        .SUM      (SUM),
        .COUT     (COUT),
        .OVERFLOW (OVERFLOW)
    );

    initial begin
        SUB = 1'b0;
        vec = 0;


        fin = $fopen("stimulus_bin.txt", "r");
        if (fin == 0) begin
            $display("ERROR: Cannot open stimulus_bin.txt");
            $finish;
        end


        fout = $fopen("results.txt", "w");
        if (fout == 0) begin
            $display("ERROR: Cannot open results.txt");
            $finish;
        end

        $display("Starting simulation and reading stimulus_bin.txt ...");


        begin : loop_block
            while (!$feof(fin)) begin

                r = $fscanf(fin, "%b %b %b %b %b\n",
                            A, B, SUM_exp, COUT_exp, OF_exp);

                if (r != 5) begin
                    $display("WARNING: fscanf read %0d items, expected 5. Exiting loop.", r);
                    disable loop_block;
                end

                #1;

                status = (SUM === SUM_exp) &&
                         (COUT === COUT_exp) &&
                         (OVERFLOW === OF_exp);

                vec = vec + 1;


                $fwrite(fout,
                    "A=\"bin=%032b,dec=%0d\"; B=\"bin=%032b,dec=%0d\"; ",
                    A, $signed(A),
                    B, $signed(B)
                );
                $fwrite(fout,
                    "SUM=\"bin=%032b,dec=%0d\"; SUM_exp=\"bin=%032b,dec=%0d\"; ",
                    SUM,     $signed(SUM),
                    SUM_exp, $signed(SUM_exp)
                );
                $fwrite(fout,
                    "COUT=%0b; COUT_exp=%0b; OF=%0b; OF_exp=%0b; status=%s\n",
                    COUT, COUT_exp,
                    OVERFLOW, OF_exp,
                    status ? "TRUE" : "FALSE"
                );


                $display(
                    "A=\"bin=%032b,dec=%0d\"; B=\"bin=%032b,dec=%0d\"; SUM=\"bin=%032b,dec=%0d\"; SUM_exp=\"bin=%032b,dec=%0d\"; COUT=%0b;
COUT_exp=%0b; OF=%0b; OF_exp=%0b; status=%s",
                    A, $signed(A),
                    B, $signed(B),
                    SUM,     $signed(SUM),
                    SUM_exp, $signed(SUM_exp),
                    COUT, COUT_exp,
                    OVERFLOW, OF_exp,
                    status ? "TRUE" : "FALSE"
                );
            end
        end


        $display("Simulation finished. Total vectors: %0d", vec);

        $fclose(fin);
        $fclose(fout);
        $finish;
    end

endmodule
```

7. Total LUT Utilization, Maximum Delays and Maximum Clock Frequency

Primitives

Ref Name	Used	Functional Category
IBUF	65	IO
OBUF	34	IO
LUT1	32	LUT
LUT4	31	LUT
CARRY4	17	CarryLogic
LUT5	1	LUT
LUT2	1	LUT

Number of LUTs from the Primitives table:

- LUT1: 32
- LUT2: 1
- LUT4: 31
- LUT5: 1

Total **LUT = 32 + 1 + 31 + 1 = 65** LUT.

Combinational Delays

From Port	To Port	Max Delay	Max Process Corner	Min Delay	Min Process Corner
B[0]	OVERFLOW	12.234	SLOW	3.415	FAST
B[23]	OVERFLOW	12.096	SLOW	3.438	FAST
B[14]	OVERFLOW	12.067	SLOW	3.324	FAST
B[6]	OVERFLOW	12.067	SLOW	3.462	FAST
B[10]	OVERFLOW	12.066	SLOW	3.393	FAST
B[22]	OVERFLOW	12.057	SLOW	3.345	FAST
B[7]	OVERFLOW	12.046	SLOW	3.408	FAST
B[1]	OVERFLOW	12.043	SLOW	3.392	FAST
B[18]	OVERFLOW	12.017	SLOW	3.337	FAST
B[4]	OVERFLOW	11.991	SLOW	3.377	FAST
B[3]	OVERFLOW	11.980	SLOW	3.519	FAST
B[16]	OVERFLOW	11.971	SLOW	3.381	FAST
B[15]	OVERFLOW	11.962	SLOW	3.391	FAST
B[24]	OVERFLOW	11.950	SLOW	3.403	FAST
B[12]	OVERFLOW	11.907	SLOW	3.366	FAST
B[11]	OVERFLOW	11.903	SLOW	3.359	FAST
B[5]	OVERFLOW	11.885	SLOW	3.292	FAST
B[20]	OVERFLOW	11.873	SLOW	3.349	FAST
B[19]	OVERFLOW	11.871	SLOW	3.357	FAST
B[8]	OVERFLOW	11.868	SLOW	3.336	FAST

The 32-bit Carry Select Adder was synthesized and implemented on the target FPGA using Vivado. According to the post-implementation utilization report, the design uses **65 LUTs** for combinational logic, in addition to **17 CARRY4 blocks** for fast carry propagation and the required I/O buffers.

The post-implementation timing summary reports the worst-case combinational delay from the inputs to the **OVERFLOW** output as **12.234 ns** under the slow process corner. Assuming that the adder is placed between two register stages in a synchronous system, the maximum operating clock frequency can be estimated as:

$$f_{\max} = \frac{1}{T_{\text{delay}}} = \frac{1}{12.234 \times 10^{-9}} \approx 81.7 \text{ MHz}$$

Therefore, under the given implementation and timing constraints, the designed 32-bit CSA can operate correctly at a clock frequency of approximately **81.7 MHz** (in practice, a slightly lower value such as **80 MHz** can be chosen as a safe design margin).