
DIGITAL SYSTEM DESIGN APPLICATIONS

Project-1 CONDITIONAL SUM ADDER

GROUP-2

Elif ÇATIKKAŞ 040190094

Berfin DUMAN 040190108

CONDITIONAL SUM ADDER

Conditional Sum Adder (CSA) represents a circuit design that performs addition based on specific conditions. This circuit embodies a structure where inputs and outputs are computed in parallel. The fundamental principle of CSA is to calculate sum and carry values separately for each condition based on whether each input is 1 or 0. For each digit, different calculations are made depending on whether the input value is 1 or 0.

For example, in a CSA circuit, distinct sum and carry values are calculated for each digit when the input is 1 or 0. These calculations take place under conditions determined by the inputs at that digit. The obtained values are then processed through components called multiplexers (mux) to achieve the actual sum and carry values.

In this manner, CSA operates as a summation circuit where inputs undergo different calculations based on specific conditions, and the results are concurrently obtained. This design is particularly employed to optimize and accelerate addition operations in digital circuits.

Examples to Understand the Algorithm

In order to learn the algorithm, an example is given in the decimal base, which is more suitable for the human mind.

0		2		9		5		9		9		0		9	
2		5		7		3		2		1		0		5	
3	2	8	7	17	16	9	8	12	11	11	10	1	0		14
28		27		169		168		121		120				14	
2869				2868								12014			
								28692014							

Figure – Example of a conditional-sum adder in the decimal system

In this example, the operation $2959909 + 25732105$ is performed. Conditions for carry (carry being 0 or 1) are written for each digit position. In the first layer, the addition is performed digit by digit, and the condition of carry (i.e., the +1 condition) is written in the adjacent box. The logic behind the creation of the second layer is as follows: for example, the result of the operation $9 + 5$ is 014, and when examined digit by digit, there is no carry. Therefore, the result from the adjacent digit (121/120) where there is no carry (120) is taken, and it is added in place of the carry digit. The next steps continue in this way.

$$014 \rightarrow 0 - 14 \rightarrow 12014$$

$$168 \rightarrow 1 - 68 \rightarrow 02868$$

$$x = 1\ 0\ 1\ 1\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ 0\ 1$$

$$y = 0\ 0\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ 0.$$

i	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	ASSUMED INITIAL CARRY	TIME INTERVAL
x_i	1	0	1	1	1	0	1	1	0	1	1	0	1	1	0	1	0	τ_0
y_i	0	0	0	1	1	0	0	1	1	0	1	1	0	1	1	0	0	
S	1	0	1	1	0	0	1	0	1	1	0	1	1	0	1	1	0	
C	0	0	0	1	1	0	0	1	0	0	1	0	0	1	0	0	0	
S	0	1	0	1	1	1	0	1	0	0	1	0	0	1	0	1	0	τ_1
C	1	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	
S	1	0	0	0	0	0	0	0	1	1	0	1	0	0	1	1	0	
C	0	1	1	1	1	1	1	1	0	1	1	0	0	1	0	0	1	
S	1	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	0	τ_2
C	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
S	1	1	0	1	0	1	0	1	0	0	1	0	1	0	1	0	1	
C	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
S	1	1	0	1	0	1	0	0	0	0	1	0	0	0	1	1	0	τ_3
C	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
S	1	1	0	1	0	1	0	1	0	0	1	0	0	0	1	1	0	
C	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
S_i	1	1	0	1	0	1	0	1	0	0	1	0	0	0	1	1	0	τ_4
C_{i+1}	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	

Figure – Example of a conditional-sum adder

Logical Design of Conditional Sum Adder

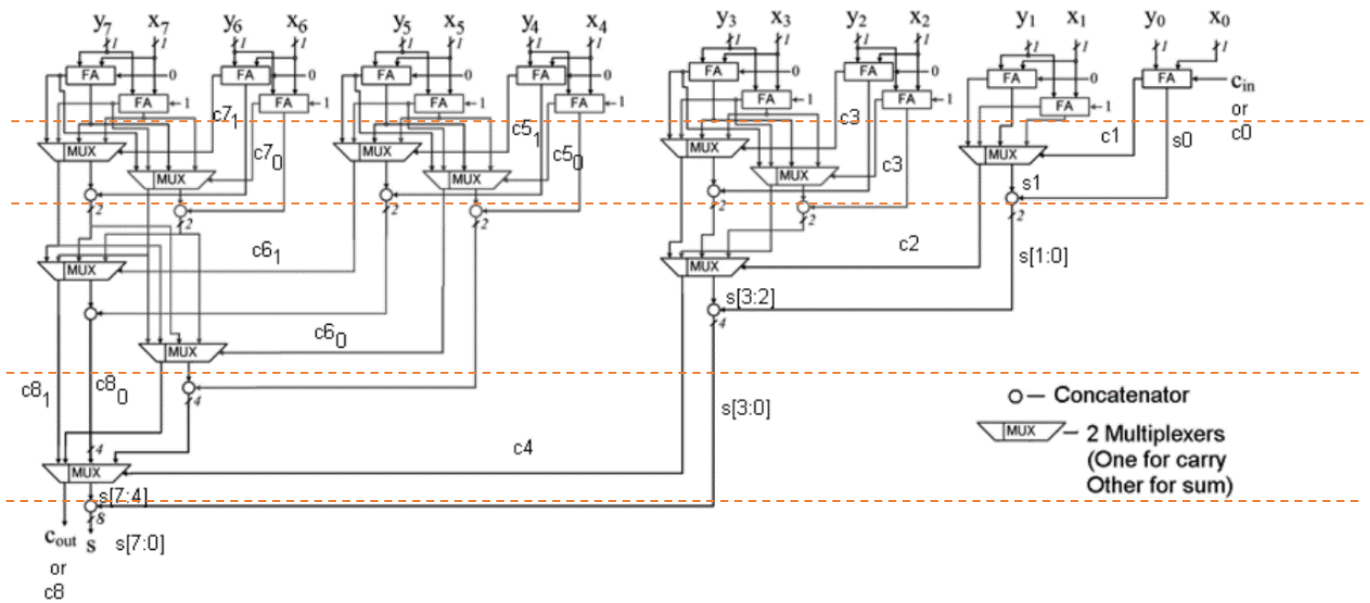


Figure – 8-bit conditional-sum adder

If the advantages of design are mentioned; in multi-bit addition operations, the processes occur in parallel, allowing the calculation of the carry for each bit to be much faster. However, waiting for the previous carry values is still necessary for obtaining the correct result. On the other hand, the design drawback lies in the significant resource usage, as for $N=2^n$ -bit addition, it requires $2N-1$ full-adders and $2^{n+1} - n - 2$ multiplexers, which constitutes an excessive utilization of resources.

In that case, for 32-bit conditional sum adder,

$$32 = 2^5; n \rightarrow 5$$

$$2 * 5 - 1 = 11 \rightarrow \text{full-adder}$$

$$2^6 - 5 - 2 = 57 \rightarrow \text{mux}$$

is required.

In addition, the design is divided into layers so that the code of the design can be written more easily.

- Adder Layer
- Mux Layer
- Mux Layer
- Mux Layer
- Output Layer

Verilog Code

Designing in smaller modules makes more sense since 32-bit inputs (x, y) and outputs (sum) will be used. Therefore, excluding the first 4 bits, the subsequent every four bits will be considered modularly.

```

module fourBcond(
    input [3:0] x,
    input [3:0] y,
    input cout_in,
    output [3:0] mux_3_sum ,
    output mux_3_cout

);
    wire [7:0] fa_cout;
    wire [7:0] fa_sum;
    wire [3:0] mux_1_sum ;
    wire [3:0] mux_1_cout;
    wire [1:0] mux_2_sum [1:0];
    wire [1:0] mux_2_cout;

    //x4 y4
    FA FA0(.x(x[0]), .y(y[0]), .cin(1'b0), .cout(fa_cout[0]),
.s(fa_sum[0]));
    FA FA1(.x(x[0]), .y(y[0]), .cin(1'b1), .cout(fa_cout[1]),
.s(fa_sum[1]));

    //x5 y5
    FA FA2(.x(x[1]), .y(y[1]), .cin(1'b0), .cout(fa_cout[2]),
.s(fa_sum[2]));
    FA FA3(.x(x[1]), .y(y[1]), .cin(1'b1), .cout(fa_cout[3]),
.s(fa_sum[3]));

    //x6 y6
    FA FA4(.x(x[2]), .y(y[2]), .cin(1'b0), .cout(fa_cout[4]),
.s(fa_sum[4]));
    FA FA5(.x(x[2]), .y(y[2]), .cin(1'b1), .cout(fa_cout[5]),
.s(fa_sum[5]));

    //x7 y7
    FA FA6(.x(x[3]), .y(y[3]), .cin(1'b0), .cout(fa_cout[6]),
.s(fa_sum[6]));
    FA FA7(.x(x[3]), .y(y[3]), .cin(1'b1), .cout(fa_cout[7]),
.s(fa_sum[7]));

```

fourBcond module-Part 1

```

//FIRST LAY
//MUX
assign mux_1_sum[0]=(fa_cout[0])? fa_sum[3]:fa_sum[2];
assign mux_1_cout[0]=(fa_cout[0])? fa_cout[3]:fa_cout[2];
//MUX
assign mux_1_sum[1]=(fa_cout[1])? fa_sum[3]:fa_sum[2];
assign mux_1_cout[1]=(fa_cout[1])? fa_cout[3]:fa_cout[2];
//MUX
assign mux_1_sum[2]=(fa_cout[4])? fa_sum[7]:fa_sum[6];
assign mux_1_cout[2]=(fa_cout[4])? fa_cout[7]:fa_cout[6];
//MUX
assign mux_1_sum[3]=(fa_cout[5])? fa_sum[7]:fa_sum[6];
assign mux_1_cout[3]=(fa_cout[5])? fa_cout[7]:fa_cout[6];

//SECOND LAY
//MUX
assign mux_2_sum[0]=(mux_1_cout[0])?
{mux_1_sum[3],fa_sum[5]}:{mux_1_sum[2],fa_sum[4]};
assign mux_2_cout[0]=(mux_1_cout[0])? mux_1_cout[2]:mux_1_cout[3];

//MUX
assign mux_2_sum[1]=(mux_1_cout[1])?
{mux_1_sum[3],fa_sum[5]}:{mux_1_sum[2],fa_sum[4]};
assign mux_2_cout[1]=(mux_1_cout[1])? mux_1_cout[3]:mux_1_cout[2];

assign mux_3_sum= (cout_in) ?
{mux_2_sum[1][1],mux_2_sum[1][0],mux_1_sum[1],fa_sum[1]} :
{mux_2_sum[0][1],mux_2_sum[0][0], mux_1_sum[0], fa_sum[0]};
assign mux_3_cout= (cout_in) ? mux_2_cout[1]: mux_2_cout[0];

endmodule

```

fourBcond module-Part 2

This module, written in the Verilog language, generates a four-bit output sum (`mux_3_sum`) and a carry-out (`mux_3_cout`) based on four-bit input numbers (`x` and `y`) and a carry-in input (`cout_in`). The module handles every four bits excluding the first 4 bits in a modular fashion. Here is an explanation of the code in paragraphs: The module processes two different conditions for each pair of bits using a full-adder (FA). Each FA takes two input bits and a carry-in input, producing a sum (`s`) and a carry-out (`cout`). This operation is realized using a total of eight FAs. The first four FAs handle two different conditions by changing the carry-in input to 0 and 1.

Next, for each pair of bits, the first-layer MUXes are used to select the sums and carry-outs obtained from two different conditions. Each MUX determines which sum to select based on a carry-out condition. The second-layer MUXes use the outputs of the first-layer MUXes to process four different conditions. Each MUX determines which sum to select based on a carry-out condition. Finally, the top-layer MUX selects the outputs of the second-layer MUXes based on the carry-in input, determining the final output sum and carry-out.

Afterwards, the main Verilog code based on the fourBcond module was written .

```
`timescale 1ns / 1ps

(*DONT_TOUCH="TRUE"*)
module Cond_Sum_Adder(
    input cin,
    input [31:0] x,
    input [31:0] y,
    output overflow,
    output cout,
    output [31:0] sum
);

    wire [6:0] fa_cout;
    wire [6:0] fa_sum;
    wire [2:0] mux_1_sum ;
    wire [2:0] mux_1_cout;
    wire [1:0] mux_2_sum ;
    wire mux_2_cout;
    wire [3:0] mux_3_sum ;
    wire mux_3_cout;
    wire [3:0]
mux_3_sum_2,mux_3_sum_3,mux_3_sum_4,mux_3_sum_5,mux_3_sum_6,mux_3_sum_7;
    wire mux_3_cout_2, mux_3_cout_3, mux_3_cout_4, mux_3_cout_5, mux_3_cout_6,
mux_3_cout_7;

    //x0,y0
    FA FA0(.x(x[0]), .y(y[0]), .cin(cin), .cout(fa_cout[0]), .s(fa_sum[0]));

    //x1 y1
    FA FA1(.x(x[1]), .y(y[1]), .cin(1'b0), .cout(fa_cout[1]), .s(fa_sum[1]));
    FA FA2(.x(x[1]), .y(y[1]), .cin(1'b1), .cout(fa_cout[2]), .s(fa_sum[2]));

    //x2 y2
    FA FA3(.x(x[2]), .y(y[2]), .cin(1'b0), .cout(fa_cout[3]), .s(fa_sum[3]));
    FA FA4(.x(x[2]), .y(y[2]), .cin(1'b1), .cout(fa_cout[4]), .s(fa_sum[4]));

    //x3 y3
    FA FA5(.x(x[3]), .y(y[3]), .cin(1'b0), .cout(fa_cout[5]), .s(fa_sum[5]));
    FA FA6(.x(x[3]), .v(v[3]), .cin(1'b1), .cout(fa cout[6]), .s(fa sum[6]));
```

```

//First Layer
//MUX
assign mux_1_sum[0]=(fa_cout[0])? fa_sum[2]:fa_sum[1];
assign mux_1_cout[0]=(fa_cout[0])? fa_cout[2]:fa_cout[1];
//MUX
assign mux_1_sum[1]=(fa_cout[3])? fa_sum[6]:fa_sum[5];
assign mux_1_cout[1]=(fa_cout[3])? fa_cout[6]:fa_cout[5];
//MUX
assign mux_1_sum[2]=(fa_cout[4])? fa_sum[6]:fa_sum[5];
assign mux_1_cout[2]=(fa_cout[4])? fa_cout[6]:fa_cout[5];

//Second Layer
//MUX
assign mux_2_sum=(mux_1_cout[0])?
{mux_1_sum[2],fa_sum[4]}:{mux_1_sum[1],fa_sum[3]};
assign mux_2_cout=(mux_1_cout[0])? mux_1_cout[2]:mux_1_cout[1];
assign cout3=mux_2_cout;
fourBcond
fourb(.x(x[7:4]),.y(y[7:4]),.cout_in(cout3),.mux_3_sum(mux_3_sum),.mux_3_cout(mu
x_3_cout)); //4-7
fourBcond
fourb2(.x(x[11:8]),.y(y[11:8]),.cout_in(mux_3_cout),.mux_3_sum(mux_3_sum_2),.mux
_3_cout(mux_3_cout_2)); //8-11
fourBcond
fourb3(.x(x[15:12]),.y(y[15:12]),.cout_in(mux_3_cout_2),.mux_3_sum(mux_3_sum_3),
.mux_3_cout(mux_3_cout_3)); //12-15
fourBcond
fourb4(.x(x[19:16]),.y(y[19:16]),.cout_in(mux_3_cout_3),.mux_3_sum(mux_3_sum_4),
.mux_3_cout(mux_3_cout_4)); //16-19
fourBcond
fourb5(.x(x[23:20]),.y(y[23:20]),.cout_in(mux_3_cout_4),.mux_3_sum(mux_3_sum_5),
.mux_3_cout(mux_3_cout_5)); //20-23
fourBcond
fourb6(.x(x[27:24]),.y(y[27:24]),.cout_in(mux_3_cout_5),.mux_3_sum(mux_3_sum_6),
.mux_3_cout(mux_3_cout_6)); //24-27
fourBcond
fourb7(.x(x[31:28]),.y(y[31:28]),.cout_in(mux_3_cout_6),.mux_3_sum(mux_3_sum_7),
.mux_3_cout(mux_3_cout_7)); //28-31

//OUTPUT Layer
assign cout=mux_3_cout_7;
assign sum =
{mux_3_sum_7,mux_3_sum_6,mux_3_sum_5,mux_3_sum_4,mux_3_sum_3,mux_3_sum_2,
mux_3_sum,mux_2_sum[1],mux_2_sum[0], mux_1_sum[0], fa_sum[0]};
assign overflow = (x[31] & y[31] & ~sum[31]) | (~x[31] & ~y[31] & sum[31]);

```

Main code – Part 2

Despite appearing complex at first glance, this design performs separate operations for each four-bit group due to its modular structure, enhancing the understandability and manageability of the design. When the main code is examined, the expression `(* DONT_TOUCH = "TRUE" *)` is used. The statement `(* DONT_TOUCH = "TRUE" *)` is used in Verilog design to instruct synthesis tools not to modify or optimize a specific element. This expression directs synthesis tools to disable any automatic optimization or alteration on the associated element.

This Verilog module is designed to compute the sum of two 32-bit numbers (x and y) along with a carry input (cin). Additionally, it produces a 32-bit sum (`sum`) and a carry output (`cout`) as output. The module is structured using smaller components. Within the module, full adder units (FA) are employed to handle two different conditions for each pair of bits. The first four FAs process two different conditions by altering the carry input to 0 and 1. This step involves a total of 8 FAs to perform the 32-bit addition. For each pair of bits, the first-layer MUXes are utilized to select the sums and carry outputs obtained from two different conditions. These conditions are determined based on the carry outputs of the preceding FAs.

Subsequently, the second-layer MUXes process the outputs of the first-layer MUXes. This layer handles four different conditions, forming a total of four pairs of bits. The carry output (`cout3`) is obtained from this layer. Each four-bit group is processed modularly using a submodule named `fourBcond`. Each submodule takes the previous carry output as input and produces a sum (`mux_3_sum`) and a carry output (`mux_3_cout`) for the respective group. The four-bit groups are processed in sequence for the bit ranges 4-7, 8-11, 12-15, 16-19, 20-23, 24-27, and 28-31. These submodules reduce the complexity of the design, allowing separate operations for each four-bit group. The output layer combines the sums from the four-bit groups to form the overall sum (`sum`). Additionally, it determines the final carry output (`mux_3_cout_7`). The design employs a modular approach, breaking down the 32-bit addition into smaller, more manageable components, enhancing the understandability and maintainability of the design.

"Overflow" refers to an excess overflow, typically indicating a situation where the result of a calculation is too large or too small to be represented. This can be interpreted in the context of both signed and unsigned numbers. The given code snippet includes a control statement for detecting overflow. The code uses a ternary expression to determine whether overflow occurs during a summation operation. The expression checks the 31st index bits of three bit arrays, x, y, and sum, to control overflow. If the bits `x[31]` and `y[31]` are set (1), and the `sum[31]` bit is not set, or if `x[31]` and `y[31]` bits are clear (0) and the `sum[31]` bit is set, the "assign overflow" variable is set to 1; otherwise, the overflow variable takes the value 0. This statement aims to identify overflow during a 32-bit addition operation.

Testing Phase

In order for the testing phase to be unbiased, Python code that generates random numbers was written.

Python Code

```
import random

def twos_complement(value, bit_length):
    # 2's complement dönüşümü
    if value >= 0:
        return format(value, f"0{bit_length}b")
    else:
        return format((1 << bit_length) + value, f"0{bit_length}b")

dosya_adı = "sayilar.txt"

with open(dosya_adı, "w") as dosya:
    for _ in range(100):
        x = int(random.uniform(-(2*7 - 1), 2*7 - 1))
        y = int(random.uniform(-(2*7 - 1), 2*7 - 1))
        z = x + y
        dosya.write(f"{twos_complement(x, 8)}\n{twos_complement(y, 8)}\n{twos_complement(z, 8)}\n")

print(f"{dosya_adı} adlı dosya olusturuldu.")
```

This Verilog code represents a testbench (`cond_sum_tb`) designed to evaluate the functionality of a module named `Cond_Sum_Adder`. The testbench includes various registers and wires to interface with the adder module, such as inputs (`cin`, `x`, `y`), outputs (`sum`, `cout`, `overflow`), and an expected sum value (`sum_exp`). The testbench initializes the carry-in (`cin`) to zero, reads a file (`random_numbers.txt`) containing 300 sets of input data, and opens an output file (`output.txt`) for result logging. It then iterates through the input data, assigning values to `x`, `y`, and `sum_exp`, and simulates the adder operation with a delay of 5 time units. The results are displayed using `\$display` and logged to the output file using `\$fdisplay`. The displayed information includes binary and decimal representations of inputs and outputs, along with a comparison between the computed sum and the expected sum, as well as an overflow indication. The testbench concludes with a message indicating the end of the test, and the simulation is terminated with `\$finish`. The purpose of this testbench is to verify the correctness of the `Cond_Sum_Adder` module by comparing its output against expected results for a set of input data.

Testbench Code

```

module cond_sum_tb;

    reg cin;
    reg [31:0] all_data[300:0]; reg [31:0] x,y,sum_exp;
    wire [31:0] sum; wire cout; wire overflow;
    integer output_file;
    integer i;

    // Module instantiation
    Cond_Sum_Adder uut (
        .cin(cin),
        .x(x),
        .y(y),
        .overflow(overflow),
        .cout(cout),
        .sum(sum)
    );
    initial begin

        cin=0;
        $readmemb("random_numbers.txt", all_data);
        output_file = $fopen("output.txt", "w");

        for (i=0; i<300; i=i+3)
            begin
                x = all_data[i];
                y = all_data[i+1];
                sum_exp=all_data[i+2];

                #5; // Add appropriate delay if needed
                $display("A=\"bin=%b, dec=%d\"; B=\"bin=%b, dec=%d\"; C_circ=\"bin=%b, dec=%d\";
C_exp=\"bin=%b, dec=%d\"; status=%s; overflow=%b\"; ",
                    x, x, y, y, sum, sum, sum_exp, sum_exp, (sum == sum_exp) ? "TRUE" : "FALSE",
overflow);
                $fdisplay(output_file, "A=\"bin=%b, dec=%d\"; B=\"bin=%b, dec=%d\"; C_circ=\"bin=%b,
dec=%d\"; C_exp=\"bin=%b, dec=%d\"; status=%s; overflow=%b\"; ",
                    x, x, y, y, sum, sum, sum_exp, sum_exp, (sum == sum_exp) ? "TRUE" : "FALSE",
overflow);
            end
        $display("end of test");
        $finish;
    end

endmodule

```

Behavioral Simulation

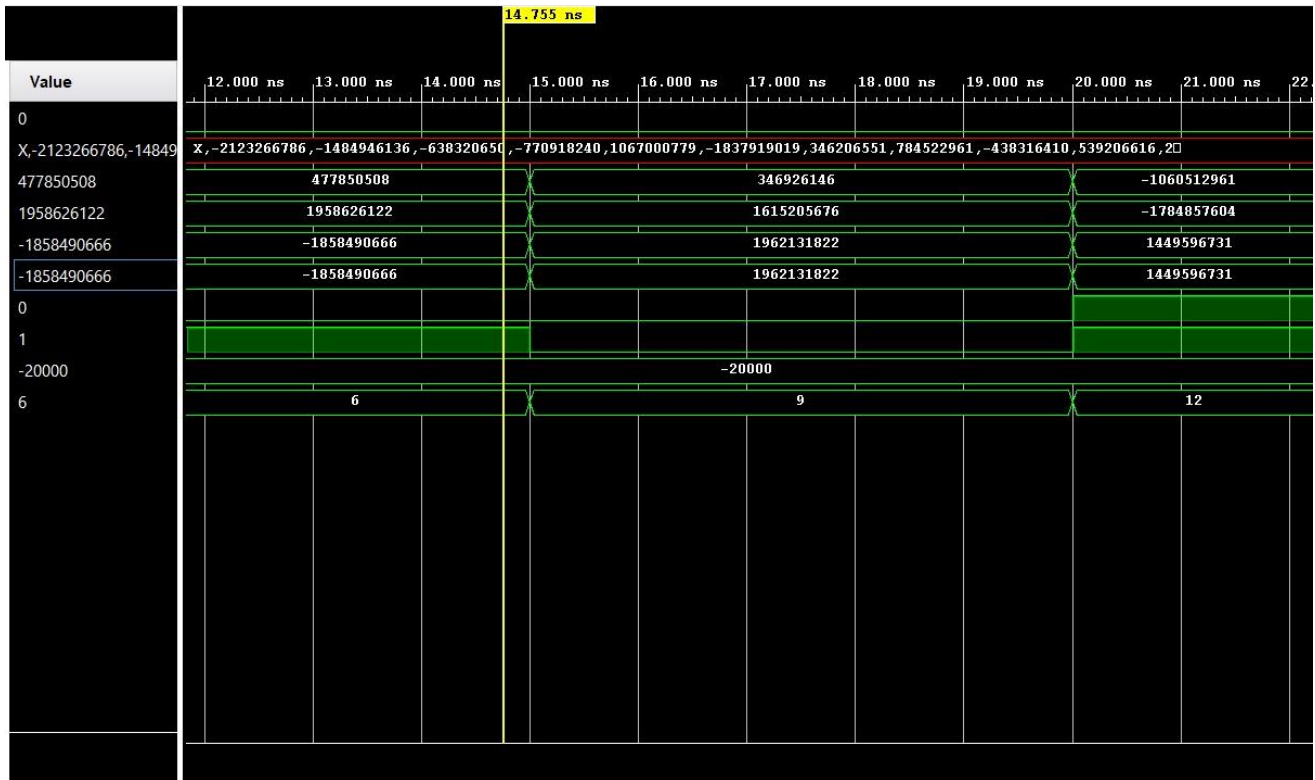


Figure – Screenshot showing correct addition result in Simulation

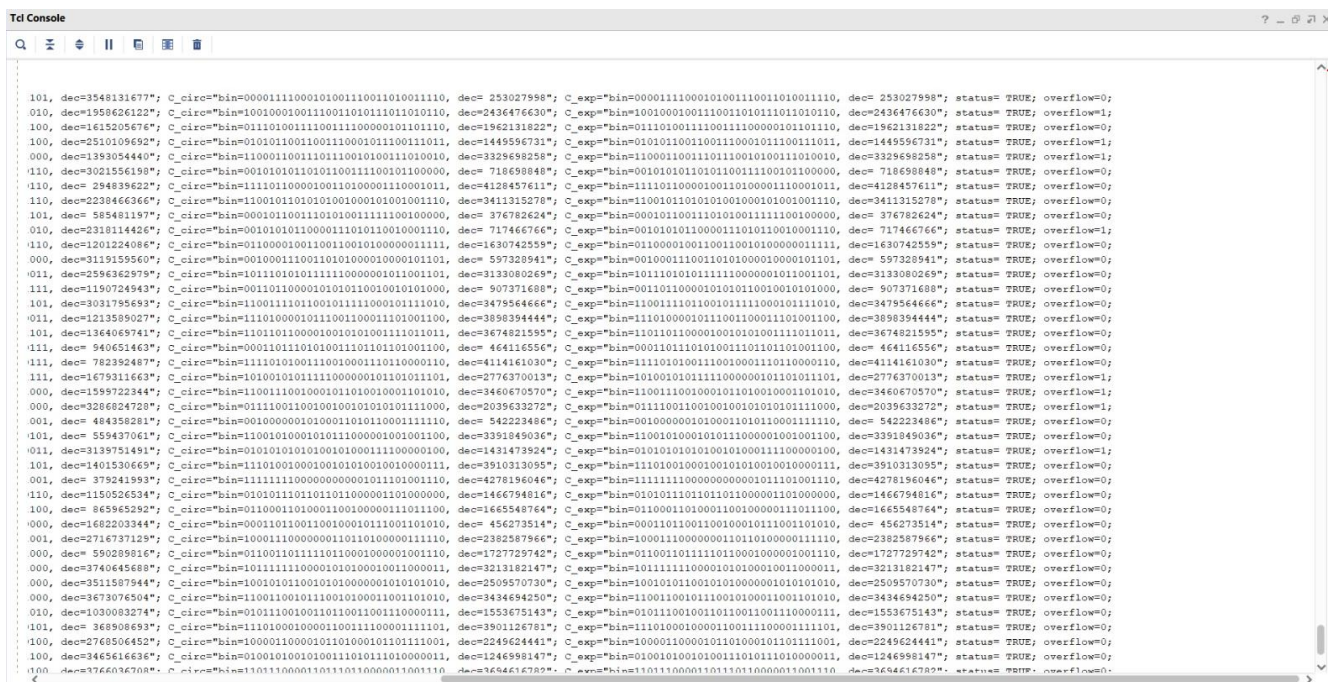


Figure – Screenshot showing correct addition result in TCL console

Implementation

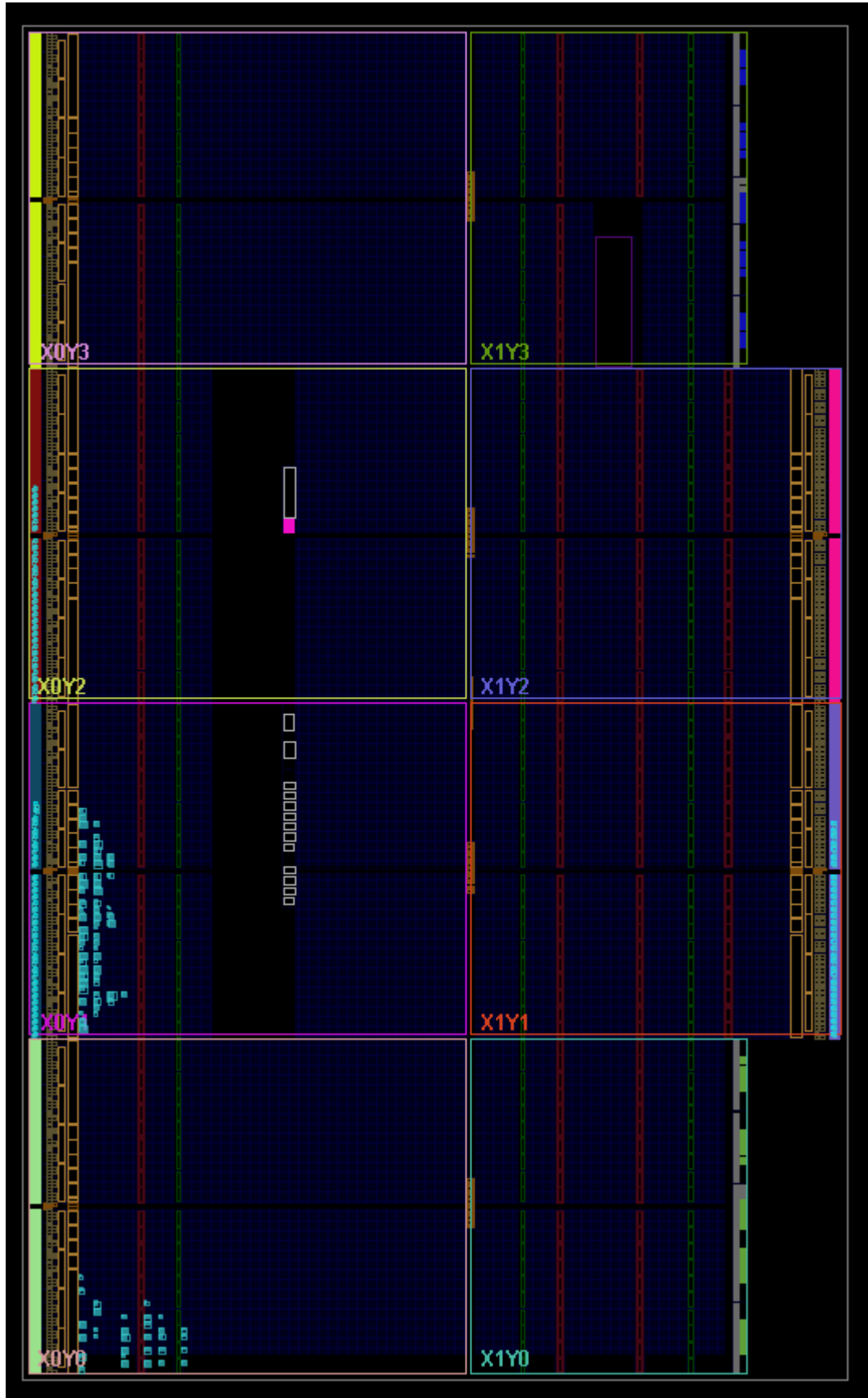
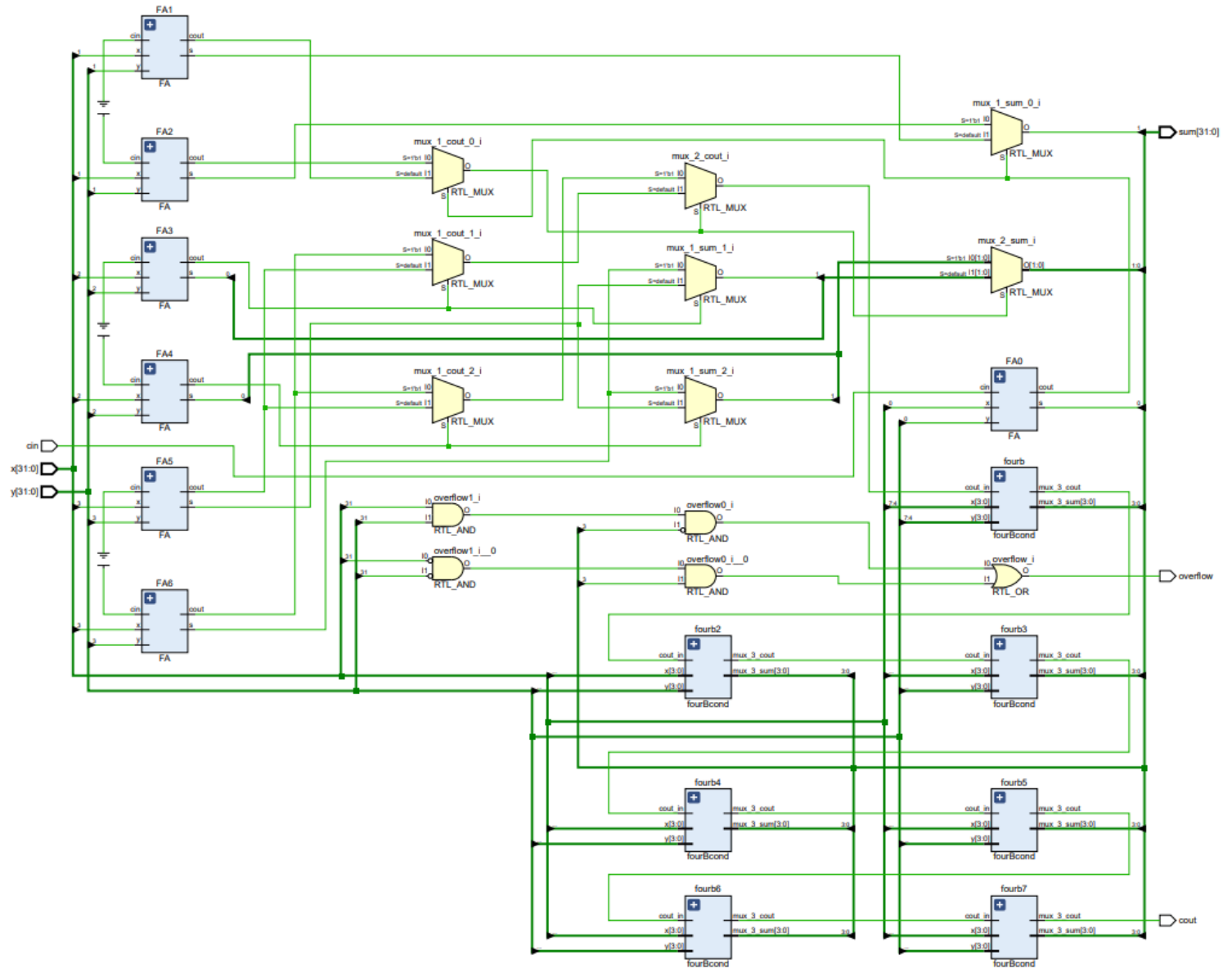


Figure – conditional-sum adder device

Technology Schematic

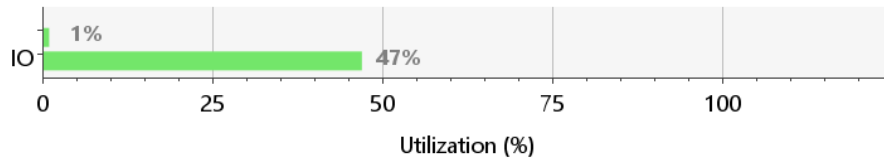


RTL Schematic



Utilization Report

Resource	Utilization	Available	Utilization %
LUT	515	63400	0.81
IO	99	210	47.14



Primitives Report

Primitives		
Ref Name	Used	Functional Category
LUT1	341	LUT
LUT2	151	LUT
IBUF	65	IO
LUT6	46	LUT
OBUF	34	IO
LUT5	22	LUT
LUT4	15	LUT
LUT3	8	LUT

Power Consumption

Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power: **24.594 W (Junction temp exceeded!)**

Design Power Budget: **Not Specified**

Power Budget Margin: **N/A**

Junction Temperature: **125.0°C**

Thermal Margin: **-52.2°C (-10.9 W)**

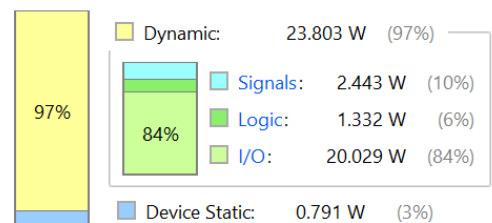
Effective θ_{JA} : 4.6°C/W

Power supplied to off-chip devices: 0 W



































Confidence level: **Low**

[Launch Power Constraint Advisor](#) to find and fix invalid switching activity

On-Chip Power



Timing Report

From Port	To Port	Max Process Corner	Min Delay	Min Process Corner	
 x[5]	 sum[31]	19.520	SLOW	6.840	FAST
 x[5]	 sum[30]	19.464	SLOW	6.799	FAST
 x[5]	 sum[29]	19.171	SLOW	6.679	FAST
 x[5]	 sum[28]	19.063	SLOW	6.611	FAST
 x[5]	 sum[26]	19.049	SLOW	6.640	FAST
 x[5]	 sum[27]	19.047	SLOW	6.622	FAST
 x[1]	 sum[31]	18.965	SLOW	6.719	FAST
 x[1]	 sum[30]	18.909	SLOW	6.677	FAST
 x[5]	 sum[25]	18.881	SLOW	6.558	FAST
 x[7]	 sum[31]	18.808	SLOW	6.697	FAST
 x[7]	 sum[30]	18.752	SLOW	6.656	FAST
 x[5]	 sum[24]	18.694	SLOW	6.458	FAST
 x[1]	 sum[29]	18.616	SLOW	6.558	FAST
 x[1]	 sum[28]	18.508	SLOW	6.490	FAST
 x[1]	 sum[26]	18.494	SLOW	6.518	FAST
 x[1]	 sum[27]	18.492	SLOW	6.501	FAST
 x[4]	 sum[31]	18.487	SLOW	6.840	FAST

The max delay was observed in the path between x[5] - sum[31] and was 19.520 ns. max frequency is $1 / 19.520\text{ns} = 51.229\text{Mhz}$

Critical path represents the longest duration within a project or process, typically the most complex route. This path determines the project completion time, and any delay on it can extend the overall completion time of the project. In the context of circuit design, this concept can refer to the path from the first bits of the input to the last bits of the output.

Timing reports showcase the maximum delay times in different sections of the circuit. When examining these reports, it can be observed that the path from the first bits of the input to the last bits of the output is indeed a critical path. Max delay values, supporting these predictions, align with the expectations when arranged from largest to smallest, indicating that the paths with the most significant delays are consistent with the predictions. This situation demonstrates that the critical paths of the design are accurately predicted and should be optimized.

Work Package Table

	Berfin DUMAN	Elif ÇATIKKAŞ
Literature Review	x	x
Research	x	x
Preparation	x	x
Design/Verilog	x	x
Report	x	x

References

1. J. Sklansky, "Conditional-Sum Addition Logic," in *IRE Transactions on Electronic Computers*, vol. EC-9, no. 2, pp. 226-231, June 1960, doi: 10.1109/TEC.1960.5219822.
2. B. Parhami, *Computer arithmetic - algorithms and hardware designs*, Oxford University Press, (2010)