DIGITAL SYSTEM DESIGN APPLICATION

EHB436E CRN: 11280

PROJECT1

Mehmet Yasir Bağcı

040200037

Salih Ömer Ongün

040220780

Hamming Weight

The Hamming weight is defined as the number of "1" bits in a binary array. This concept has extensive applications in various fields of computer science, including information theory, coding theory, and cryptography. It is also widely used in digital circuit design to analyze energy consumption in circuits.

In digital electronic circuit design, power consumption is often directly related to the number of active bits, as indicated by the Hamming weight. By analyzing the Hamming weight, the energy efficiency of a circuit can be evaluated, allowing for the optimization of power consumption. Furthermore, in chip design, reducing the Hamming weight of bit manipulation processes is a critical strategy for minimizing energy usage and creating energy-efficient hardware systems. The concept of Hamming weight is named after the American mathematician Richard Hamming, who made significant contributions to coding theory and information theory.

Hamming Weight Calculation Methods

The Hamming weight can be determined using several algorithms, including:

- The Iterative Method
- The Subtractive Method
- Parallel Counting

In the iterative method, each bit in the binary array is individually checked to count the number of "1" bits. The subtractive method, on the other hand, operates by repeatedly subtracting the least significant "1" bit and incrementing a counter until all bits are processed. Finally, parallel counting employs optimized techniques to calculate the Hamming weight more efficiently, particularly in systems requiring high-speed computations.

In our project, we utilize the parallel counting method due to its efficiency and suitability for modern hardware applications.

Algorithm

In our algorithm, we take 32-bit input to our main module. We divide 32-bit to 4-bit slices. In the submodule, we calculate the number of ones in 4-bit slices with half adders and full adders. Next, return the sum to the main module. Finally, we add the number of ones in all 4-bit slices.

Design Sources

```
(* DONT_TOUCH = "TRUE" *)
module calc_hamming
(
    input [31:0] DATA,
    output [5:0] RESULT
);
    wire [2:0] sum_ham [7:0];
    reg [5:0] res_reg;
    integer j;
    genvar i;
    generate
        for(i = 0; i<8; i = i+1) begin : gen_part
            slice_adder gen_slice
            (
                DATA[i*4 +:4],
                sum_ham[i]
            );
        end
    endgenerate
    always @(*) begin
        res_reg = 6'b0;
        for (j = 0; j < 8; j = j + 1) begin
            res_reg = res_reg + sum_ham[j];
        end
    end
    assign RESULT = res_reg;
endmodule
```

Şekil 1 Verilog Code

"calc_hamming" is our top module. We take a 32-bit binary input and create a 6-bit output. We divide the 32-bit input into eight 4-bit slices. We create eight "sum_ham" wires. Each wire has 3-bit length. These wires hold the ones in each 4-bit slice. We use always structure to calculate all slice results to the output of "calc_hamming" and we need reg type. Therefore, we create the "res_reg" wire that holds the number of ones in all slices

Next, we instantiate the "slice_adder" module and calculate the ones in each 4-bit slice with generate-for structure. Finally, we add all slice results to the output of "calc_hamming" with always structure.

```
module slice_adder
(
    input [3:0] slice,
    output [2:0] sum
);

    wire [1:0] half_sum1;
    wire [1:0] half_sum2;

    HA half1
    (
      .x(slice[0]),
      .y(slice[1]),
      .sum(half_sum1[0]),
      .cout(half_sum1[1])
    );

    HA half2
    (
      .x(slice[2]),
      .y(slice[3]),
      .sum(half_sum2[0]),
      .cout(half_sum2[1])
    );

    parametric_RCA para1
    (
        .x(half_sum1),
        .y(half_sum2),
        .cin(0),
        .cout(sum[2]),
        .sum(sum[1:0])
    );

endmodule
```

Şekil 2 Verilog Code of slice adder module

"slice_adder" is the module that calculates the number of ones in a slice. We take 4-bit input and 3-bit output. We use two half adder and a 2-bit ripple carry adder. We create two 2-bit wires. These wires hold number of ones in two bit of 4-bit slice. We add the numbers in the first two indexes with one half adder and the numbers in the last two bits with the other half adder and assign sum and cout of half adders to wires.

Finally, we add two wires with 2-bit ripple carry adder. The inputs to the ripple carry adder are two 2-bit wires, the cin is logical 0 because we calculate each slice independently. The sum and cout of ripple carry adder are connected to the total output of "slice_adder".

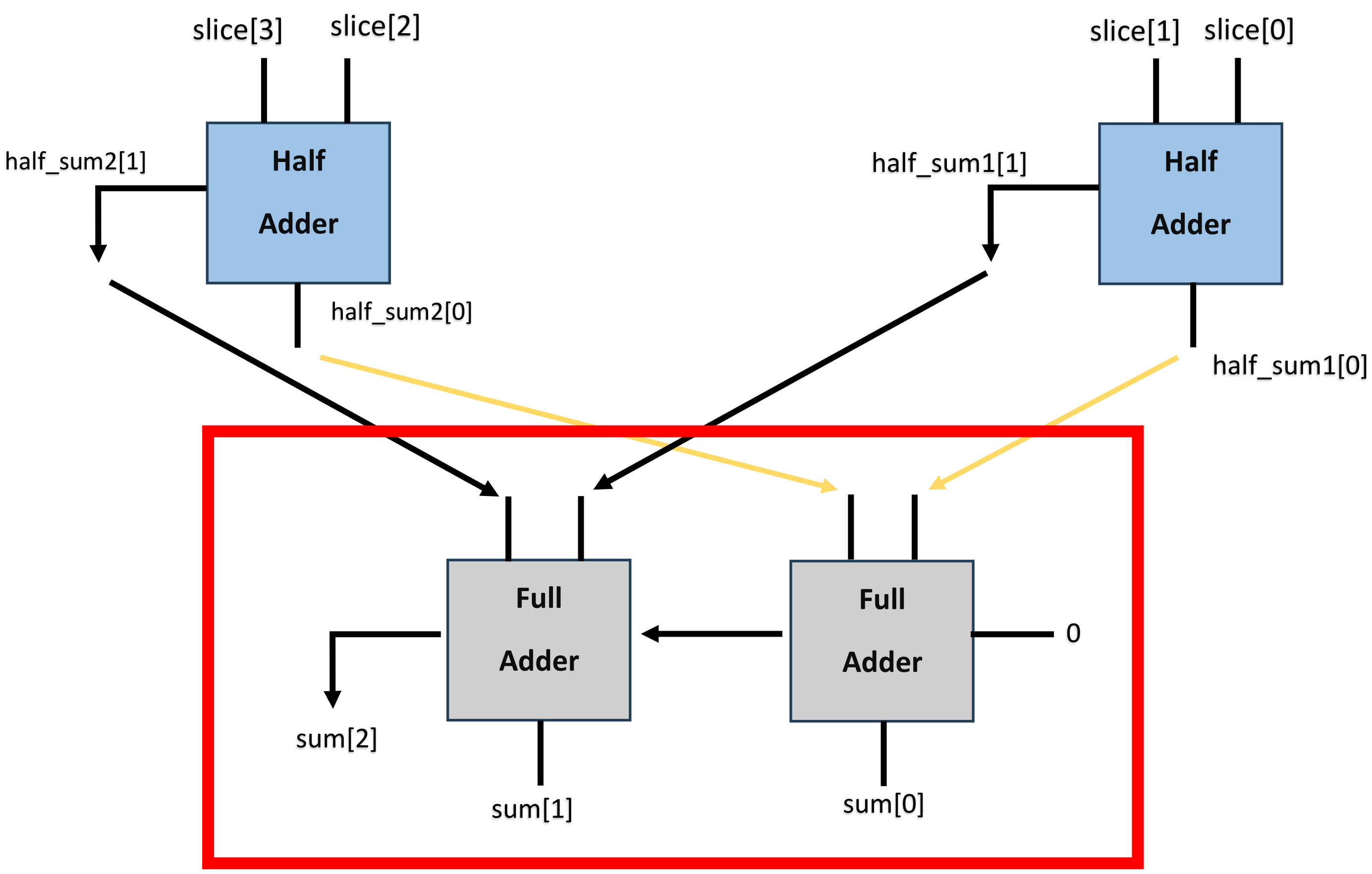
```verilog
module HA
(
    input x,
    input y,
    output cout,
    output sum
);
    assign sum = x ^ y ;
    assign cout = x & y;

endmodule


module FA
(
    input x,
    input y,
    input cin,
    output cout,
    output sum
);
    wire ha1_sum, ha1_cout,
ha2_cout;

    HA half1
    (
      .x(x),
      .y(y),
      .sum(ha1_sum),
      .cout(ha1_cout)
    );

    HA half2
    (
        .x(ha1_sum),
        .y(cin),
        .sum(sum),
        .cout(ha2_cout)
    );

    OR or1
    (
    .l1(ha1_cout),
    .l2(ha2_cout),
    .O(cout)
    );

endmodule
```

Şekil 4 Half Adder Module Code

```verilog
module OR
(
    input l1,
    input l2,
    output O
);

    assign O = l1 | l2;

endmodule

module parametric_RCA
(
    input [1:0] x,
    input [1:0] y,
    input cin,
    output cout,
    output [1:0] sum
);

    wire [2:0] cout_gen;

    assign cout_gen[0] = cin;

    genvar i;
    generate
        for(i = 0; i<2; i = i+1)
begin : gen_full_adder
            FA gen_full
            (
                x[i],
                y[i],
                cout_gen[i],
                cout_gen[i+1],
                sum[i]
            );
        end
    endgenerate
    assign cout = cout_gen[2];

endmodule
```

Şekil 3 OR Gate Module Code

Block Schema of "slice_adder" Module Simulation Source



Ripple Carry Adder

Simulation Source

```
module slice_adder_tb();

    reg [3:0] SLICE = 4'b000;
    wire [2:0] SUM;

    slice_adder uut
    (
        .slice(SLICE),
        .sum(SUM)
    );

    initial
    begin

        SLICE = 4'b0000;
        #10;
        SLICE = 4'b0001;
        #10;
        SLICE = 4'b0010;
        #10;
        SLICE = 4'b0011;
        #10;
        SLICE = 4'b0100;
        #10;
        SLICE = 4'b0101;
        #10;
        SLICE = 4'b0110;
        #10;
        SLICE = 4'b0111;
        #10;
        SLICE = 4'b1000;
        #10;
        SLICE = 4'b1001;
        #10;
        SLICE = 4'b1010;
        #10;
        SLICE = 4'b1011;
        #10;
        SLICE = 4'b1100;
        #10;
        SLICE = 4'b1101;
        #10;
        SLICE = 4'b1110;
        #10;
        SLICE = 4'b1111;
        #10;
        $finish;
    end
endmodule
```

Şekil 5 Slice adder testbench Code

We created all 4-bit numbers and assigned them to the input of the “slice_adder” module. I controlled “slice_adder” module whether the system is working.

```python
import random

def gen_binary():
    with open("stimulus_file.txt","w") as f:
        for k in range(0,100):
            number = random.getrandbits(32)
            bin_num = format(number, "0b")
            print(len(bin_num))
            if len(bin_num)<32:
                for i in range(0,32-len(bin_num)):
                    bin_num = "0" + str(bin_num)
            ones = bin_num.count('1')
            print("number of 1s")
            print(ones)
            f.write(bin_num + " " + str(ones) +"\n")
            print(len(bin_num))
            print("----------------")
```

Şekil 6 Python Code for data

We created a Python code to generate 32-bit random data. We used random library. "random.getrandbits(32)" produces numbers in the range 0 to 2^(32) -1. "format function" converts decimal type to binary. If the numbers are less than 32 bits long, we do zero extension. For example, if the number is 5, it has a binary representation of 101. After zero extension, it has a 32-bit representation. "ones" function calculates number of ones in the string. To compare the ones count with our Verilog results, we added the 32-bit data and the ones count of the data to the same line in the file.

Example input file line

10101010011010001111111100010000 16

$\underbrace{\hspace{6cm}}_{\text{32-bit data}}\ \underbrace{\hspace{4cm}}_{\text{Number of ones in DATA}}$

```
module calc_hamming_tb();

    reg [31:0] DATA = 32'b0;
    wire [5:0] RESULT;
    integer file, out_file,ones_count;

    calc_hamming uut
    (
        .DATA(DATA),
        .RESULT(RESULT)
    );


    initial begin

        file = $fopen("stimulus_file.txt", "r");
        if (file == 0) begin
            $display("Cannot open stimulus file.");
            $finish;
        end

        out_file = $fopen("outputs.txt", "w");
        if (out_file == 0) begin
            $display("Cannot open outputs file.");
            $finish;
        end


        while (!$feof(file)) begin
            $fscanf(file, "%b %d\n", DATA, ones_count);
            #10;
            if( ones_count == RESULT) begin
                $display("DATA: %b, Ones Count:%d ,RESULT: %d, TRUE",DATA,
ones_count,RESULT);
                $fdisplay(out_file, "DATA: %b, Ones Count: %d, RESULT: %d, TRUE", DATA,
ones_count, RESULT);
            end
            else begin
                $display("DATA: %b, Ones Count:%d ,RESULT: %d, FALSE",DATA,
ones_count,RESULT);
                $fdisplay(out_file, "DATA: %b, Ones Count: %d, RESULT: %d, TRUE", DATA,
ones_count, RESULT);
            end
        end
        $fclose(file);
        $fclose(out_file);
        $finish;
    end

endmodule
```

Şekil 7 Testbench code for project

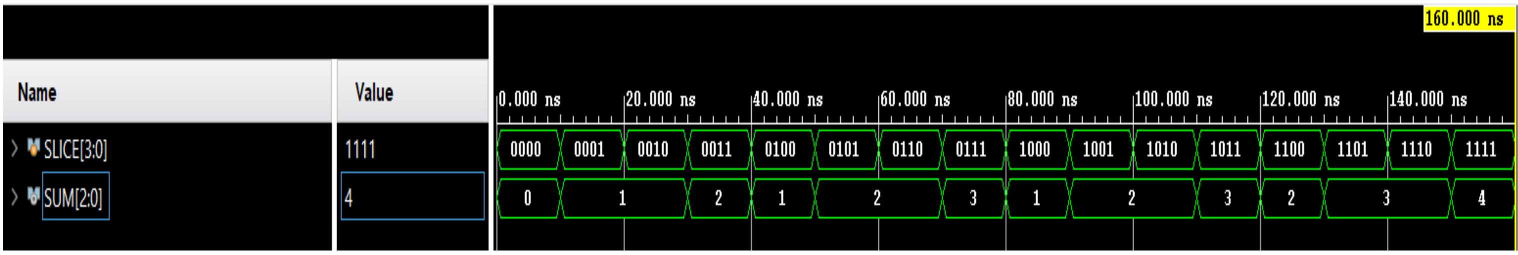
"calc_hamming_tb" is our main testbench module. We use the module for simulation. Firstly, we open the input file(stimulus_file) and output file(outputs) with read and write modes respectively. If it cannot open the files, it gives an error message. Next, while loop scans the line-by-line input file, it assigns the first value in the file (32-bit input) to “DATA” and, the second to “ones_count”(number of ones in data). After calculating “RESULT,” it compares “RESULT” with “ones_count”. If it has the same results, it gives the true message. Otherwise,

it provides a false message. It provides message TCL Console and out output file. Finally, it closes the input and output files and completes the module.

Example output file line

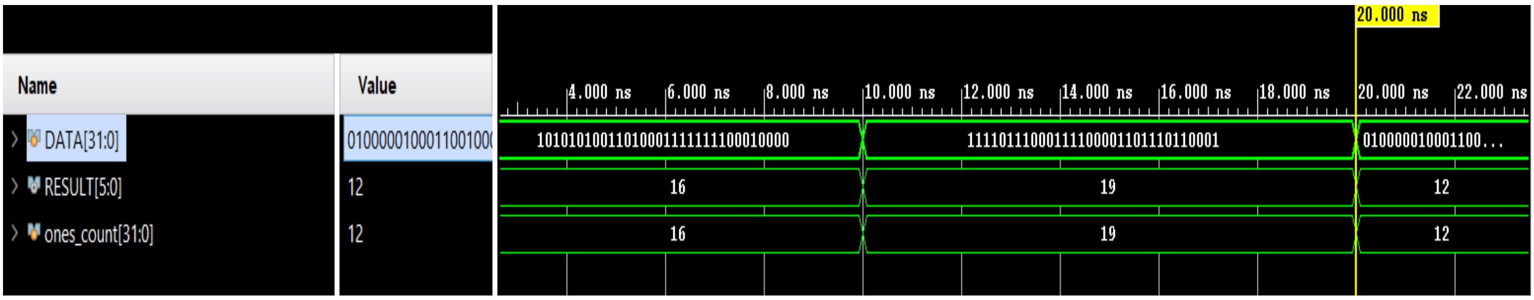
DATA: 10101010011010001111111100010000, Ones Count: 16, RESULT: 16, TRUE

Simulation Wave



Şekil 8 Slice adder waveform

To test if “slice_adder” module works, we generate all 4-bit values and calculate the number of ones in the values. The design calculates the number of ones accurately.



Şekil 9 calc_hamming waveform

We generate 100 random 32-bit numbers. “ones_count” calculated in Phyton code, “RESULT” calculated in Verilog. There are 3 values in the photo but we tested 100 numbers.

TCL Console

```
# run 1000ns
DATA: 10101010011010001111111100010000, Ones Count:          16 ,RESULT: 16, TRUE
DATA: 11110111000111100001101110110001, Ones Count:          19 ,RESULT: 19, TRUE
DATA: 01000001000110010000111000011101, Ones Count:          12 ,RESULT: 12, TRUE
DATA: 00000001010111011101001101111011, Ones Count:          17 ,RESULT: 17, TRUE
DATA: 00001111110001101111110101001011, Ones Count:          19 ,RESULT: 19, TRUE
DATA: 01100111101001111000110011110001, Ones Count:          18 ,RESULT: 18, TRUE
DATA: 00000110111000000000011000100010, Ones Count:           9 ,RESULT:  9, TRUE
DATA: 11110100111110010101010000011100, Ones Count:          17 ,RESULT: 17, TRUE
DATA: 01110111011111011110101111001100, Ones Count:          22 ,RESULT: 22, TRUE
DATA: 11001001010011001100100011001101, Ones Count:          15 ,RESULT: 15, TRUE
DATA: 01101111100001100011001011011100, Ones Count:          17 ,RESULT: 17, TRUE
DATA: 11100111000000110110001100110111, Ones Count:          17 ,RESULT: 17, TRUE
DATA: 01110101000110010110000011100101, Ones Count:          15 ,RESULT: 15, TRUE
DATA: 10011111011010111011101001001101, Ones Count:          20 ,RESULT: 20, TRUE
DATA: 10001111111110001110100110011111, Ones Count:          21 ,RESULT: 21, TRUE
DATA: 10011011000101011111101010011011, Ones Count:          19 ,RESULT: 19, TRUE
DATA: 11100011100101101111001110001101, Ones Count:          19 ,RESULT: 19, TRUE
DATA: 11101010000010011000000000001000, Ones Count:           9 ,RESULT:  9, TRUE
DATA: 10100011001000010011111101000000, Ones Count:          13 ,RESULT: 13, TRUE
DATA: 01010111010001110000100100001111, Ones Count:          15 ,RESULT: 15, TRUE
DATA: 01011010101101110111111101000100, Ones Count:          19 ,RESULT: 19, TRUE
DATA: 00110010110001000001001011001010, Ones Count:          12 ,RESULT: 12, TRUE
DATA: 00011011000101111111110000110011, Ones Count:          18 ,RESULT: 18, TRUE
DATA: 11000000110110100100011110001001, Ones Count:          14 ,RESULT: 14, TRUE
DATA: 00000001011110101000001001100011, Ones Count:          12 ,RESULT: 12, TRUE
DATA: 01001101100001100000011011100011, Ones Count:          14 ,RESULT: 14, TRUE
DATA: 00010110110000000110110011000011, Ones Count:          13 ,RESULT: 13, TRUE
DATA: 01110100101100011000111100011110, Ones Count:          17 ,RESULT: 17, TRUE
DATA: 00100111110111100010011100001101, Ones Count:          17 ,RESULT: 17, TRUE
DATA: 00111111010101001011111100111100, Ones Count:          20 ,RESULT: 20, TRUE
DATA: 01000000001000001100101110010010, Ones Count:          10 ,RESULT: 10, TRUE
```

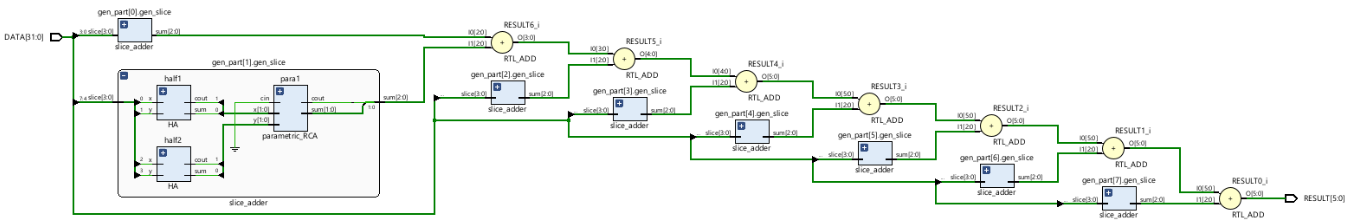
```
DATA: 11100010000010111001101000001001, Ones Count:          13 ,RESULT: 13, TRUE
DATA: 11110100100111000001111010110001, Ones Count:          17 ,RESULT: 17, TRUE
DATA: 00000111101001101000010000101001, Ones Count:          12 ,RESULT: 12, TRUE
DATA: 10001111110010000011100010010110, Ones Count:          15 ,RESULT: 15, TRUE
DATA: 01010011011111010011001011001000, Ones Count:          16 ,RESULT: 16, TRUE
DATA: 01101111001101110100111100100101, Ones Count:          19 ,RESULT: 19, TRUE
DATA: 01100101100001111001010001101101, Ones Count:          16 ,RESULT: 16, TRUE
DATA: 10001000111111000010001110100000, Ones Count:          13 ,RESULT: 13, TRUE
DATA: 01111110110101011000010001010001, Ones Count:          16 ,RESULT: 16, TRUE
DATA: 01101001000000101111111011101100, Ones Count:          17 ,RESULT: 17, TRUE
DATA: 00110001011101001001101001011010, Ones Count:          15 ,RESULT: 15, TRUE
DATA: 00101001010000000001110001011111, Ones Count:          13 ,RESULT: 13, TRUE
DATA: 00010100010100010111001111011111, Ones Count:          17 ,RESULT: 17, TRUE
DATA: 10011111111010010000110010110000, Ones Count:          15 ,RESULT: 15, TRUE
DATA: 10010000101001010000100000111101, Ones Count:          12 ,RESULT: 12, TRUE
DATA: 01111011011001101111010101011101, Ones Count:          22 ,RESULT: 22, TRUE
DATA: 01001010111110100110111110001110, Ones Count:          19 ,RESULT: 19, TRUE
DATA: 10000111011010100101000000111101, Ones Count:          15 ,RESULT: 15, TRUE
DATA: 01000000100011011001011011101010, Ones Count:          14 ,RESULT: 14, TRUE
DATA: 11100111000011111011011101111010, Ones Count:          21 ,RESULT: 21, TRUE
DATA: 11110110000010110110000011010111, Ones Count:          17 ,RESULT: 17, TRUE
DATA: 01100101000001001001001001000011, Ones Count:          11 ,RESULT: 11, TRUE
DATA: 11101110000110011101110100110011, Ones Count:          19 ,RESULT: 19, TRUE
DATA: 00100000101011110001100100011100, Ones Count:          13 ,RESULT: 13, TRUE
DATA: 01111111001010111100001011000001, Ones Count:          17 ,RESULT: 17, TRUE
DATA: 11111010000101101100001001111101, Ones Count:          18 ,RESULT: 18, TRUE
DATA: 10111100110101010101000001011010, Ones Count:          16 ,RESULT: 16, TRUE
DATA: 11111011110101100101111101100000, Ones Count:          20 ,RESULT: 20, TRUE
DATA: 00010110100000101111001110010111, Ones Count:          16 ,RESULT: 16, TRUE
DATA: 10011001011110011101011010010011, Ones Count:          18 ,RESULT: 18, TRUE
DATA: 10111010101110110100000100010000, Ones Count:          14 ,RESULT: 14, TRUE
```

```
DATA: 10001101000101000110010101100011, Ones Count:          14 ,RESULT: 14, TRUE
DATA: 01100110011110111011110011010001, Ones Count:          19 ,RESULT: 19, TRUE
DATA: 11110001100100100111101110101110, Ones Count:          19 ,RESULT: 19, TRUE
DATA: 10011011100001100010010111010010, Ones Count:          15 ,RESULT: 15, TRUE
DATA: 00011111011101000010001100110001, Ones Count:          15 ,RESULT: 15, TRUE
DATA: 10000111101000100010010101000001, Ones Count:          12 ,RESULT: 12, TRUE
DATA: 11011010001100111111011011110111, Ones Count:          22 ,RESULT: 22, TRUE
DATA: 01000110110111001000011011011001, Ones Count:          16 ,RESULT: 16, TRUE
DATA: 11111100010010011110111011000111, Ones Count:          20 ,RESULT: 20, TRUE
DATA: 11010110000010100001100110101111, Ones Count:          16 ,RESULT: 16, TRUE
DATA: 01000001011001010100100001001010, Ones Count:          11 ,RESULT: 11, TRUE
DATA: 10100100001011000111010110001100, Ones Count:          14 ,RESULT: 14, TRUE
DATA: 00100000100000000101011001111101, Ones Count:          12 ,RESULT: 12, TRUE
DATA: 11001001011110101011010111011011, Ones Count:          20 ,RESULT: 20, TRUE
DATA: 00111101111000001111011101100111, Ones Count:          20 ,RESULT: 20, TRUE
DATA: 01111101010001000001101111110101, Ones Count:          18 ,RESULT: 18, TRUE
DATA: 10111010111010000011110011111110, Ones Count:          20 ,RESULT: 20, TRUE
DATA: 01000000011101110011011100010010, Ones Count:          14 ,RESULT: 14, TRUE
DATA: 00101110000100010001110101100011, Ones Count:          14 ,RESULT: 14, TRUE
DATA: 11100010110011010111010000011101, Ones Count:          17 ,RESULT: 17, TRUE
DATA: 11000111011101000111111100110110, Ones Count:          20 ,RESULT: 20, TRUE
DATA: 00000101011110000011101001011010, Ones Count:          14 ,RESULT: 14, TRUE
DATA: 10101001010010101100010000000000, Ones Count:          10 ,RESULT: 10, TRUE
DATA: 10001011110000011100101000111111, Ones Count:          17 ,RESULT: 17, TRUE
DATA: 10111100010101011101100111010100, Ones Count:          18 ,RESULT: 18, TRUE
DATA: 00000100110000101111001110101011, Ones Count:          15 ,RESULT: 15, TRUE
DATA: 11100001010100010001010111111000, Ones Count:          15 ,RESULT: 15, TRUE
DATA: 10110110000001010110010010001111, Ones Count:          15 ,RESULT: 15, TRUE
DATA: 01001110101101010101111001001111, Ones Count:          19 ,RESULT: 19, TRUE
DATA: 10100101000100011111000000011001, Ones Count:          13 ,RESULT: 13, TRUE
DATA: 11001010001110101000110001110110, Ones Count:          16 ,RESULT: 16, TRUE
DATA: 00110110011101101010111001110111, Ones Count:          20 ,RESULT: 20, TRUE
DATA: 00110010101000101010010001011100, Ones Count:          13 ,RESULT: 13, TRUE
```

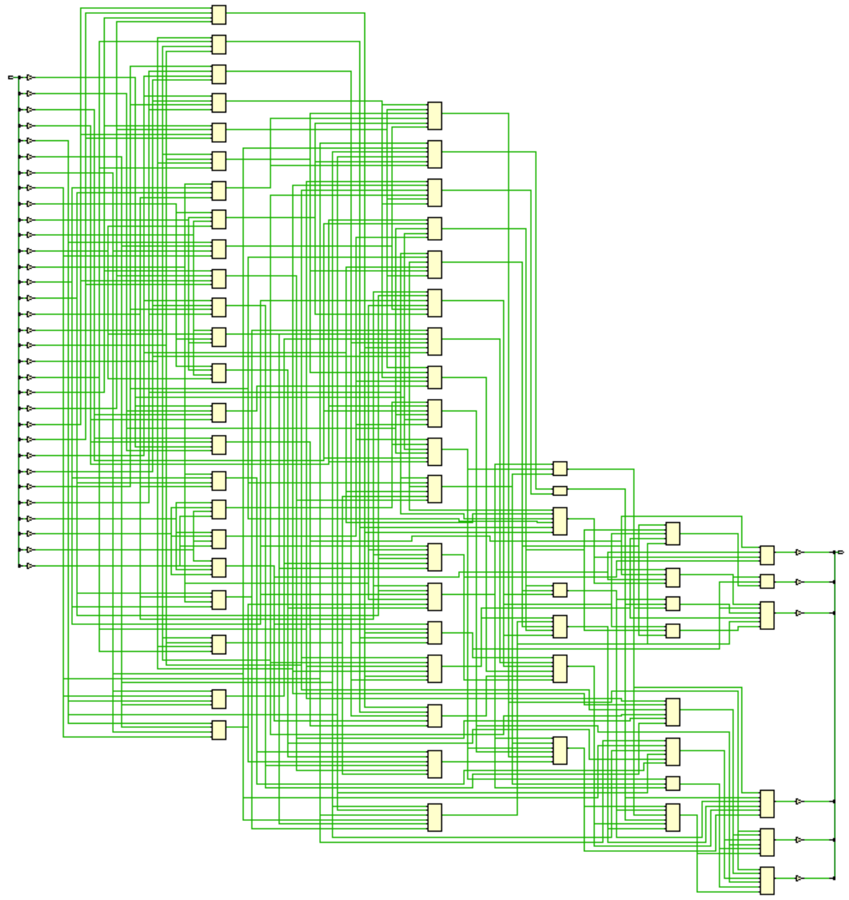
```
DATA: 00110100000000111110000111100000, Ones Count:          13 ,RESULT: 13, TRUE
DATA: 00000001001010010110111100101110, Ones Count:          14 ,RESULT: 14, TRUE
DATA: 01000011110100001010111011101101, Ones Count:          17 ,RESULT: 17, TRUE
DATA: 10100010110101010000000101101000, Ones Count:          12 ,RESULT: 12, TRUE
DATA: 01101010011010110010110100110001, Ones Count:          16 ,RESULT: 16, TRUE
```

Şekil 10 Console Outputs

RTL AND TECHNOLOGY SCHEMATIC



Şekil 11 RTL Schematic

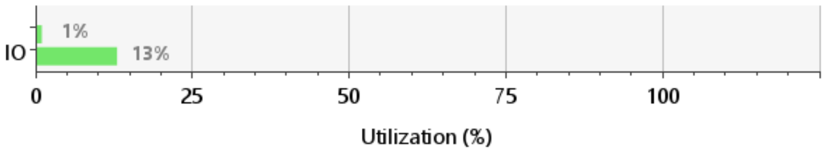


Şekil 12 Technology Schematic

UTILIZATION AND DELAYS

Summary

Resource	Utilization	Available	Utilization %
LUT	49	41000	0.12
IO	38	300	12.67



Şekil 13 Utilization Summary

According to utilization report, 49 LUT block have been used in our hamming weigth desisgn as the technology design shows above. The table below shows us the some of the delay time of our design. Between DATA[20] and RESULT[2] ports, delay time is the highest time of the circuit which means this path is considered as critical path. If we want to calculate the maximum clock frequency; first step is assess the minimum period of the clock by finding critical path. This means:

$$T_{clock} = T_{cp} \qquad f_{clock} = 1/T_{clock}$$

By utilizing with this equation we obtain 1/9,233 = 108,3 MHz for maximum clock frequency.

From Port	To Port	Max Delay	Max Process Corner	Min Delay	Min Process Corner
DATA[18]	RESULT[5]	8.067	SLOW	2.740	FAST
DATA[19]	RESULT[0]	6.899	SLOW	2.742	FAST
DATA[19]	RESULT[1]	7.739	SLOW	2.867	FAST
DATA[19]	RESULT[2]	8.481	SLOW	2.591	FAST
DATA[19]	RESULT[3]	7.847	SLOW	2.755	FAST
DATA[19]	RESULT[4]	8.420	SLOW	2.535	FAST
DATA[19]	RESULT[5]	8.307	SLOW	2.722	FAST
DATA[20]	RESULT[0]	6.841	SLOW	2.723	FAST
DATA[20]	RESULT[1]	7.856	SLOW	2.928	FAST
DATA[20]	RESULT[2]	9.233	SLOW	2.718	FAST
DATA[20]	RESULT[3]	8.599	SLOW	2.939	FAST
DATA[20]	RESULT[4]	8.854	SLOW	2.984	FAST
DATA[20]	RESULT[5]	8.765	SLOW	3.020	FAST
DATA[21]	RESULT[0]	6.888	SLOW	2.737	FAST

Şekil 14 Delay Times

REFERENCES

[1] Behrooz Parhami, Computer Arithmetic Algorithms and Hardware Designs, 2nd ed. 2010, pp. 164– 167.

[2] "Hamming weight," Wikipedia. https://en.wikipedia.org/wiki/Hamming_weight

Work Package

Name	Code	Report
Mehmet Yasir Bağcı	calc_hamming module slice_adder_tb module	• Hamming Weight • Hamming Weight Calculation Methods • RTL AND TECHNOLOGY SCHEMATIC • UTILIZATION AND DELAYS
Salih Ömer Ongün	slice_adder module calc_hamming_tb module phyton code	• Algorithm • Design Sources • Block Schema • Simulation Source • Simulation Wave • TCL Console