

**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**

Şekil 1 Verilog Code

**(\*** 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**

"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.

Şekil 2 Verilog Code of slice adder module

**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**

"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".

Şekil 3 OR Gate Module Code

**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**

**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

**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**

**Block Schema of “slice\_adder” Module Simulation Source**

slice[3]

slice[2]

slice[1]

slice[0]

**Half**

**Adder**

half\_sum2[1]

half\_sum1[1]

**Half**

**Adder**

half\_sum2[0]

half\_sum1[0]

sum[0]

sum[1]

sum[2]

0

**Full**

**Adder**

**Full**

**Adder**

**Ripple Carry Adder**

**Simulation Source**

Şekil 5 Slice adder testbench Code

**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**

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.

Şekil 6 Python Code for data

**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(**"----------------"**)**

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

32-bit data Number of ones in DATA

Şekil 7 Testbench code for project

**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**

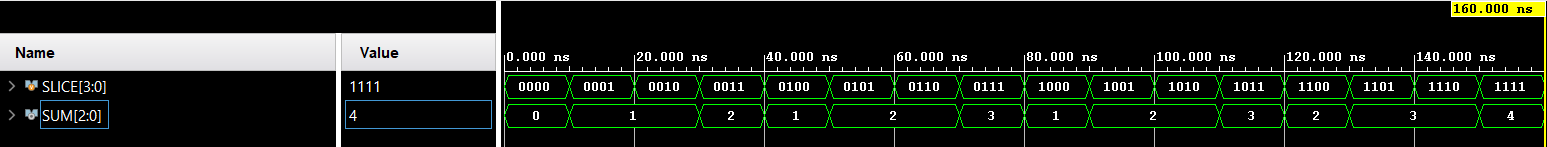
"calc\_hamming\_tb” is our main testbench module. We use the module for simulation. Firstly, we open the input file(stimulus\_file) and output file(outputs) with read and write modes respectively.  If it cannot open the files, it gives an error message. Next, while loop scans the line-by-line input file, 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

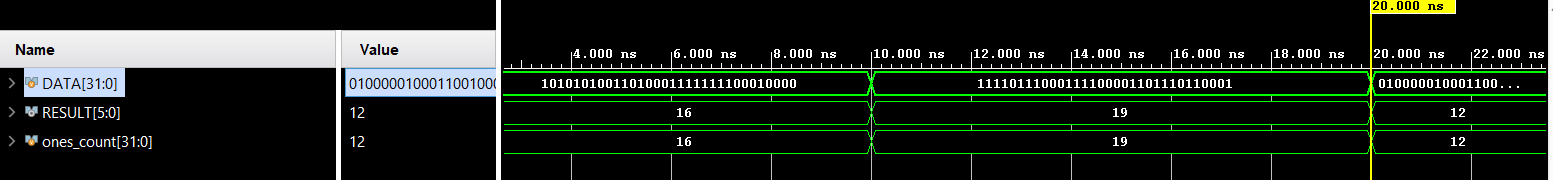
DATA: 10101010011010001111111100010000, Ones Count: 16, RESULT: 16, TRUE

**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**

metin, ekran görüntüsü, yazı tipi, sayı, numara içeren bir resim

Açıklama otomatik olarak oluşturuldumetin, ekran görüntüsü, yazı tipi, doküman, belge içeren bir resim

Açıklama otomatik olarak oluşturuldumetin, ekran görüntüsü, doküman, belge, yazı tipi içeren bir resim

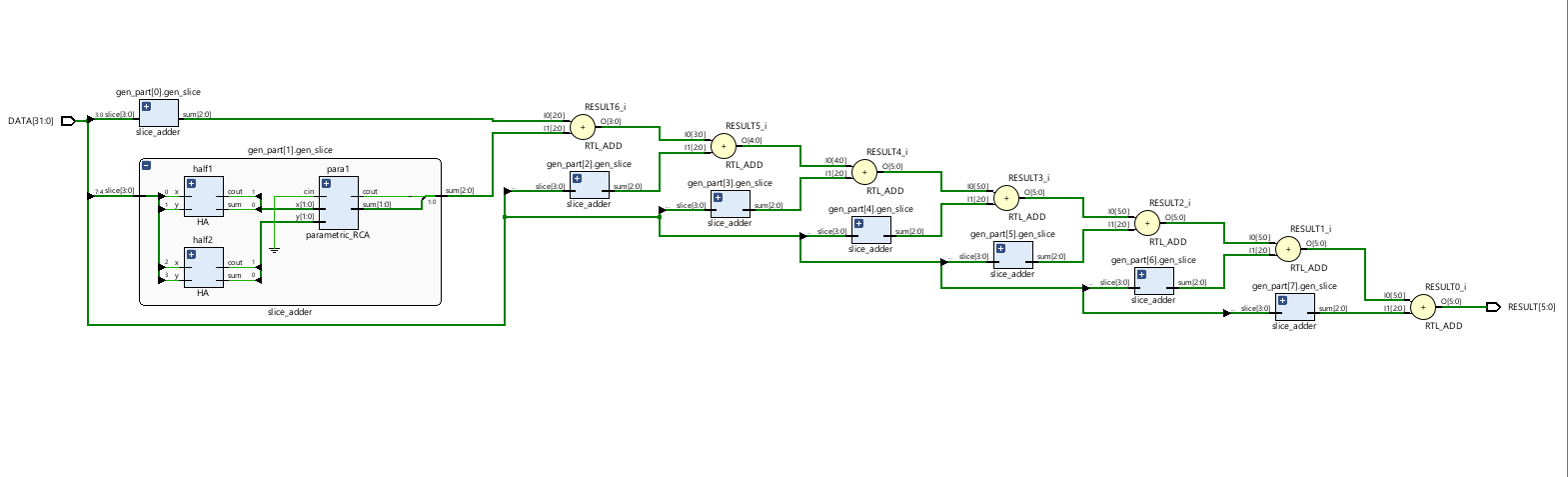
Açıklama otomatik olarak oluşturuldu

metin, ekran görüntüsü, yazı tipi, çizgi içeren bir resim

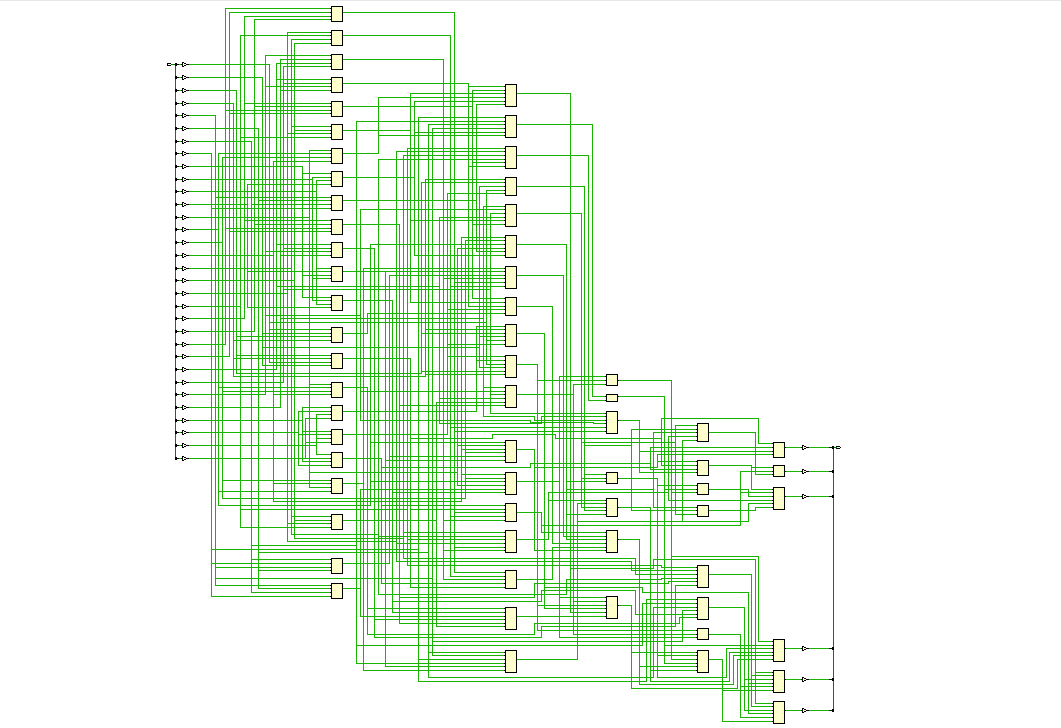
Açıklama otomatik olarak oluşturuldu

Şekil 10 Console Outputs

**RTL AND TECHNOLOGY SCHEMATIC**

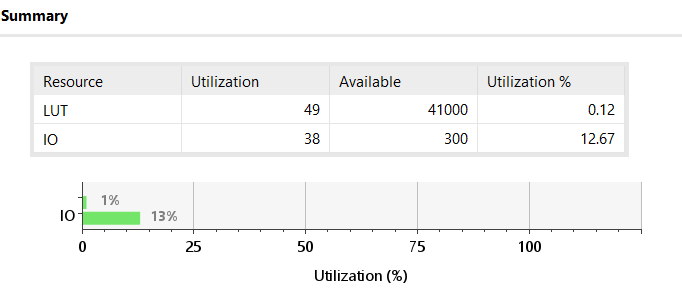


Şekil 11 RTL Schematic



Şekil 12 Technology Schematic

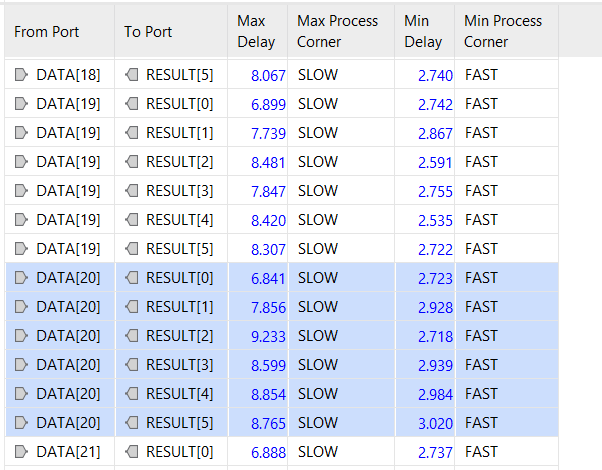
**UTILIZATION AND DELAYS**



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:

*Tclock  = Tcp fclock = 1/Tclock*

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

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](https://en.wikipedia.org/wiki/Hamming_weight%20)

**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 |