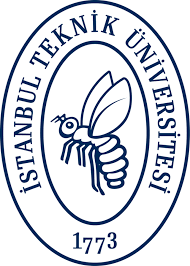
****

**DIGITAL SYSTEM DESIGN APPLICATION – EHB 436E**

**Project I**

**Muhammed Velihan Bağcı**

**040170093**

**Yiğit Bektaş Gürsoy**

**040180063**

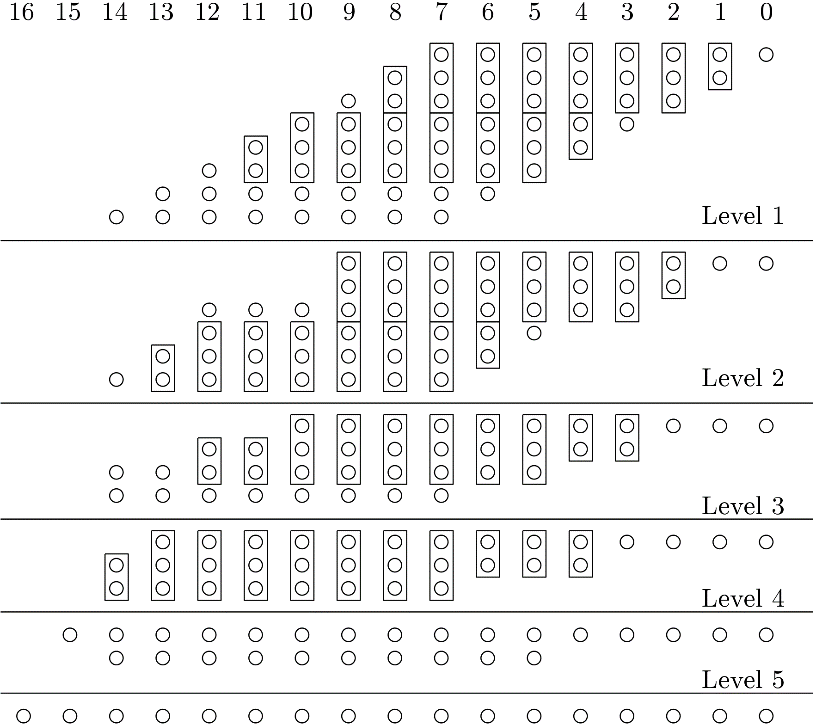
**Class Lecturer: Sıddıka Berna Örs Yalçın**

**Class Assistant:  
Serdar Duran  
Yasin Fırat Kula  
Mehmet Onur Demirtürk**

1. **Wallace Tree Multiplier**

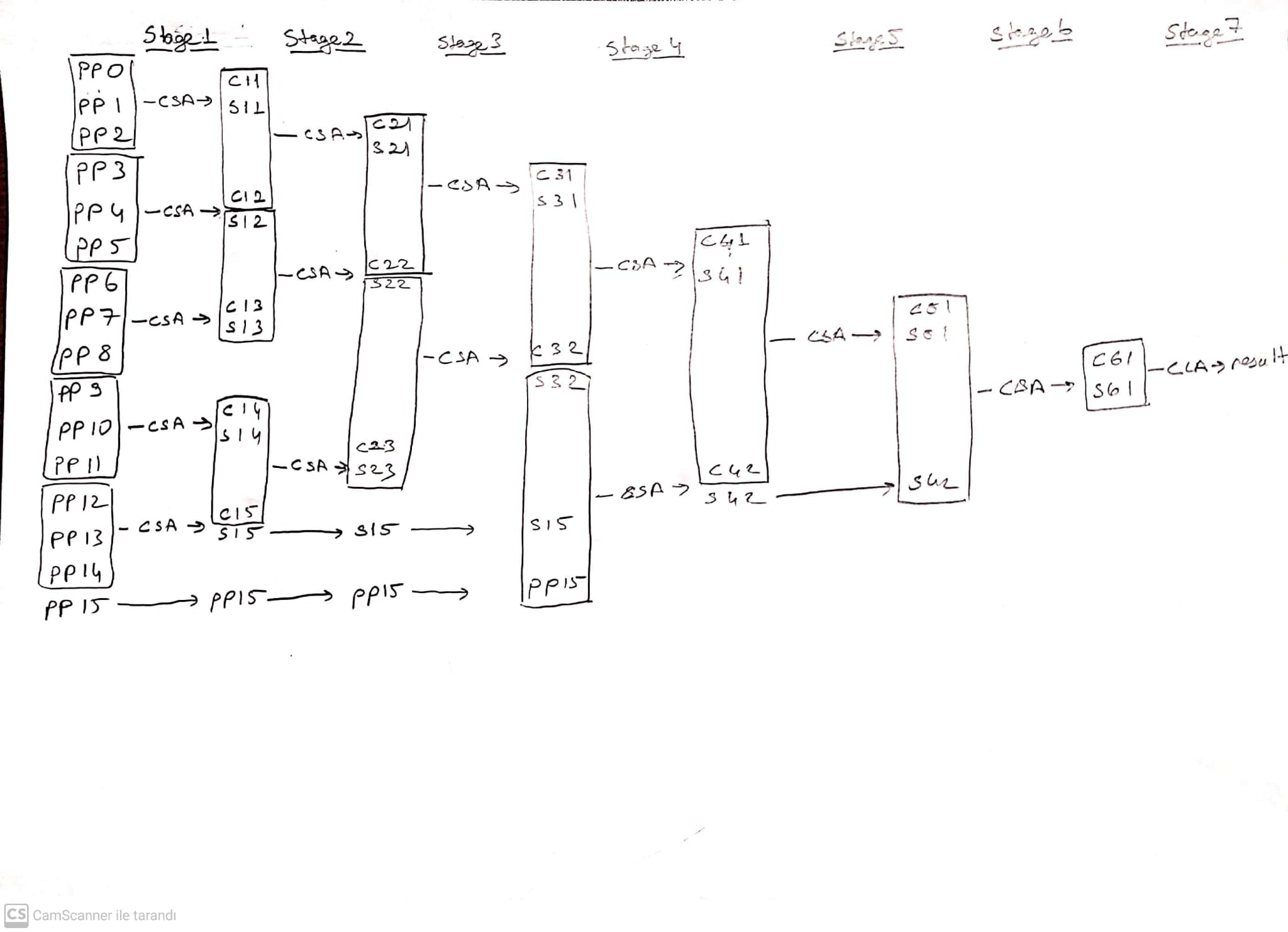
The wallace tree multiplier is an algorithm that performs multiplication for two binary numbers.

First, partial products are calculated. Partial products are the offset of the first number. The translation is done by looking at the second number. The first number is shifted to the left by the numeric value for each bit of the '1' value of the second number. In the '0' bits of the second number, the partial products get '0'.



The partial products obtained are collected in threes with the Carry save adder. As a result of this operation, 32-bit carry and sum numbers are obtained. The process is repeated in the same way for the subsequent results.

At the end of the transactions, the two results from a single CSA are combined with the CLA and the result is obtained.

****

1. **Verilog Code**

**(\*** DONT\_TOUCH **=** "TRUE" **\*)**

**module** Wallace\_Tree\_Mult**(**

**input** **[**15**:**0**]**A**,**

**input** **[**15**:**0**]**B**,**

**output** **[**31**:**0**]**C**,**

**output** carry

**);**

**wire** **[**31**:**0**]** pp **[**15**:**0**];**

PP partial\_products**(**

A**,**

B**,**

pp**[**0**][**31**:**0**],**

pp**[**1**][**31**:**0**],**

pp**[**2**][**31**:**0**],**

pp**[**3**][**31**:**0**],**

pp**[**4**][**31**:**0**],**

pp**[**5**][**31**:**0**],**

pp**[**6**][**31**:**0**],**

pp**[**7**][**31**:**0**],**

pp**[**8**][**31**:**0**],**

pp**[**9**][**31**:**0**],**

pp**[**10**][**31**:**0**],**

pp**[**11**][**31**:**0**],**

pp**[**12**][**31**:**0**],**

pp**[**13**][**31**:**0**],**

pp**[**14**][**31**:**0**],**

pp**[**15**][**31**:**0**]**

**);**

// STAGE1

**wire** **[**31**:**0**]** sum\_s11**,** carry\_s11**,** sum\_s12**,** carry\_s12**,** sum\_s13**,** carry\_s13**,** sum\_s14**,** carry\_s14**,** sum\_s15**,** carry\_s15**;**

CSA s11 **(**pp**[**0**][**31**:**0**],** pp**[**1**][**31**:**0**],** pp**[**2**][**31**:**0**],** sum\_s11**[**31**:**0**],** carry\_s11**[**31**:**0**]);**

CSA s12 **(**pp**[**3**][**31**:**0**],** pp**[**4**][**31**:**0**],** pp**[**5**][**31**:**0**],** sum\_s12**[**31**:**0**],** carry\_s12**[**31**:**0**]);**

CSA s13 **(**pp**[**6**][**31**:**0**],** pp**[**7**][**31**:**0**],** pp**[**8**][**31**:**0**],** sum\_s13**[**31**:**0**],** carry\_s13**[**31**:**0**]);**

CSA s14 **(**pp**[**9**][**31**:**0**],** pp**[**10**][**31**:**0**],** pp**[**11**][**31**:**0**],** sum\_s14**[**31**:**0**],** carry\_s14**[**31**:**0**]);**

CSA s15 **(**pp**[**12**][**31**:**0**],** pp**[**13**][**31**:**0**],** pp**[**14**][**31**:**0**],** sum\_s15**[**31**:**0**],** carry\_s15**[**31**:**0**]);**

//STAGE2

**wire** **[**31**:**0**]** sum\_s21**,** carry\_s21**,** sum\_s22**,** carry\_s22**,** sum\_s23**,** carry\_s23**;**

CSA s21 **(**sum\_s11**[**31**:**0**],** carry\_s11**[**31**:**0**],** sum\_s12**[**31**:**0**],** sum\_s21**[**31**:**0**],** carry\_s21**[**31**:**0**]);**

CSA s22 **(**carry\_s12**[**31**:**0**],** sum\_s13**[**31**:**0**],** carry\_s13**[**31**:**0**],** sum\_s22**[**31**:**0**],** carry\_s22**[**31**:**0**]);**

CSA s23 **(**sum\_s14**[**31**:**0**],** carry\_s14**[**31**:**0**],** sum\_s15**[**31**:**0**],** sum\_s23**[**31**:**0**],** carry\_s23**[**31**:**0**]);**

//STAGE3

**wire** **[**31**:**0**]** sum\_s31**,** carry\_s31**,** sum\_s32**,** carry\_s32**;**

CSA s31 **(**sum\_s21**[**31**:**0**],** carry\_s21**[**31**:**0**],** sum\_s22**[**31**:**0**],** sum\_s31**[**31**:**0**],** carry\_s31**[**31**:**0**]);**

CSA s32 **(**carry\_s22**[**31**:**0**],** sum\_s23**[**31**:**0**],** carry\_s23**[**31**:**0**],** sum\_s32**[**31**:**0**],** carry\_s32**[**31**:**0**]);**

//STAGE4

**wire** **[**31**:**0**]** sum\_s41**,** carry\_s41**,** sum\_s42**,** carry\_s42**;**

CSA s41 **(**sum\_s31**[**31**:**0**],** carry\_s31**[**31**:**0**],** sum\_s32**[**31**:**0**],** sum\_s41**[**31**:**0**],** carry\_s41**[**31**:**0**]);**

CSA s42 **(**carry\_s32**[**31**:**0**],** carry\_s15**[**31**:**0**],** pp**[**15**][**31**:**0**],** sum\_s42**[**31**:**0**],** carry\_s42**[**31**:**0**]);**

//STAGE5

**wire** **[**31**:**0**]** sum\_s51**,** carry\_s51**;**

CSA s51 **(**sum\_s41**[**31**:**0**],** carry\_s41**[**31**:**0**],** sum\_s42**[**31**:**0**],** sum\_s51**[**31**:**0**],** carry\_s51**[**31**:**0**]);**

//STAGE6

**wire** **[**31**:**0**]** sum\_s61**,** carry\_s61**;**

CSA s61 **(**sum\_s51**[**31**:**0**],** carry\_s51**[**31**:**0**],** carry\_s42**[**31**:**0**],** sum\_s61**[**31**:**0**],** carry\_s61**[**31**:**0**]);**

CLA Carry\_Lookahead\_Adder**(**

**.**A**(**sum\_s61**),**

**.**B**(**carry\_s61**),**

**.**CIN**(**1'b0**),**

**.**COUT**(**carry**),**

**.**SUM**(**C**)**

**);**

**endmodule**

1. **Code Algorithm Explanation**

In the above code, first partial products are calculated and PP module is used for this process. The obtained partial product values ​​are collected using CSA modules and the two 32-bit outputs are sent to the next stage. Stages may have unprocessed inputs and these inputs are transferred directly to the next stage. At each stage, the outputs of the previous stages and the values ​​that have not been processed before are added to the CSA process. In this way, at the end of 6 stages, 2 32-bit numbers are obtained. The obtained numbers are added to the addition process with the help of CLA.

Although carry output is added in this circuit, it always gives '0' output. This is because the product of two 16-bit numbers cannot exceed 32-bit. However, the carry output is connected to the carry output of the CLA and it has been observed that it always gives '0' as a result of the simulations.

1. **Random Number Generated with Python**

With the python code specified below, 2 random integers are generated, each of which is 100. Since the circuit in the project is a multiplication circuit, the numbers produced separately are multiplied with each other and written as result, then all of these results are written to the random\_numbers.txt file. These generated numbers were generated for use in simulation.

**import** random

# Open the file for writing

**with** **open(**'random\_numbers.txt'**,** 'w'**)** **as** f**:**

# Generate 100 random pairs of integers

**for** i **in** **range(**100**):**

a **=** random**.**randint**(**0**,** 65535**)**

b **=** random**.**randint**(**0**,** 65535**)**

result **=** a **\*** b

# Write the result to the file

f**.**write**(str(**a**)** **+** " " **+** **str(**b**)+** " " **+str(**result**)** **+** '\n'**)**

1. **Test Bench Code**

`timescale 1ns **/** 1ps

**module** Wallace\_tb**();**

**reg** **[**15**:**0**]** A**;**

**reg** **[**15**:**0**]** B**;**

**wire** **[**31**:**0**]** C**;**

**wire** carry**;**

**integer** status**;**

**wire** **[**31**:**0**]**Expected**;**

**integer** fin\_ptr**,** fout\_ptr**;**

Wallace\_Tree\_Mult UUT **(**

**.**A**(**A**),**

**.**B**(**B**),**

**.**C**(**C**),**

**.**carry**(**carry**)**

**);**

**initial** **begin**

fin\_ptr **=** $fopen**(**"C:/Users/yigit/Desktop/random\_numbers.txt"**,**"r"**);**

fout\_ptr **=** $fopen**(**"C:/Users/yigit/Desktop/output.txt"**,**"w"**);**

**while(!**$feof**(**fin\_ptr**))**

**begin**

$fscanf**(**fin\_ptr**,**"%d %d %d\n"**,** A**,** B**,** Expected**);** **#**10**;**

$display**(**"A=%b, %d / B=%b, %d / Expected=%b, %d"**,** A**,** A**,** B**,** B**,** Expected**,** Expected**);**

$display**(**"Result=%b, %d / carry=%b || Expected Result=%b, %d || Status = %d \n "**,** C**,** C**,** carry**,** Expected**,** Expected**,** status **);**

**assign** status **=** C **===** Expected **?** 1'b1 **:** 1'b0**;**

$fwrite**(**fout\_ptr**,**"Result=%b, %d / carry=%b || Expected Result=%b, %d || Status= %d \n"**,** C**,** C**,** carry**,** Expected**,** Expected**,** status**);**

**end**

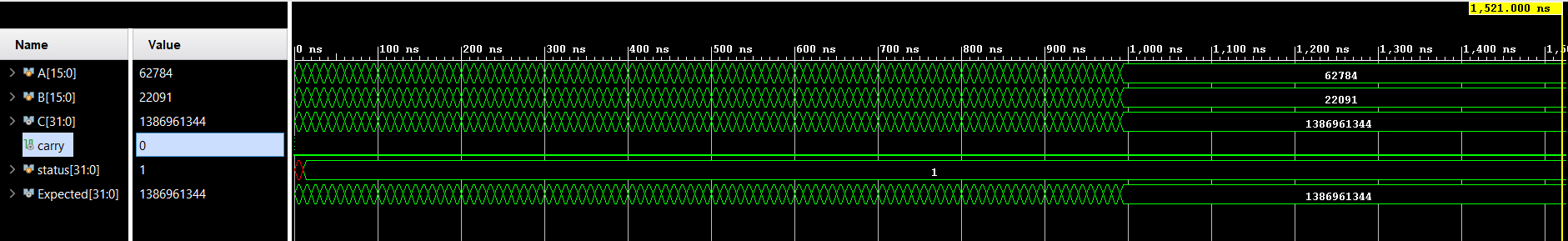
$finish**();**

**end**

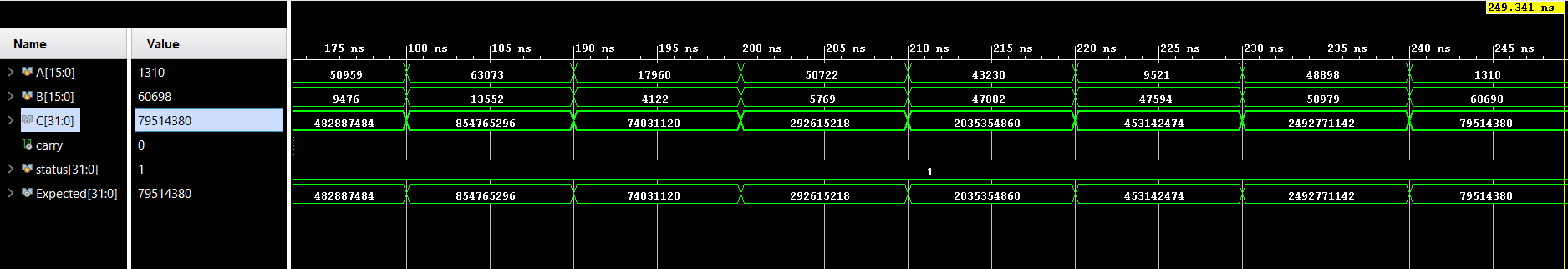
**endmodule**

1. **Behavioral Simulation Results**

Two simulation results are shown below. A and B values are numbers that can take a maximum of 216-1 values drawn in the random\_numbers.txt file and randomly generated on python. The C value is the value that represents the A multiplied by the B value. The value called Expected was written to a .txt file to be verified from Python, and then transferred to the simulation via Vivado for accuracy comparison. It is written in the testbench code to give the value "TRUE" when the status value is 1 and "FALSE" when it is 0. This can be seen in the testbench code above. The carry value represents the COUT value from the CLA.

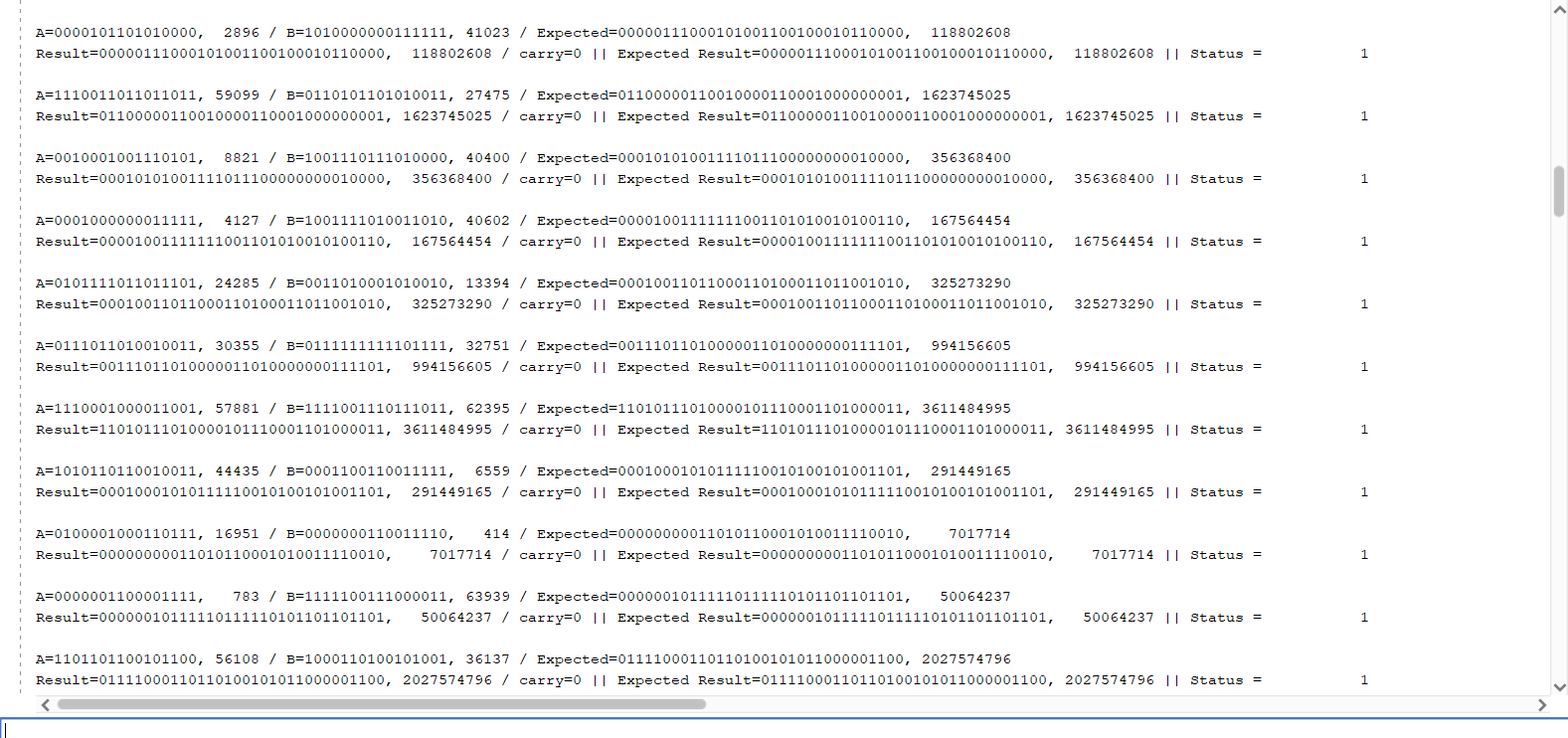
****

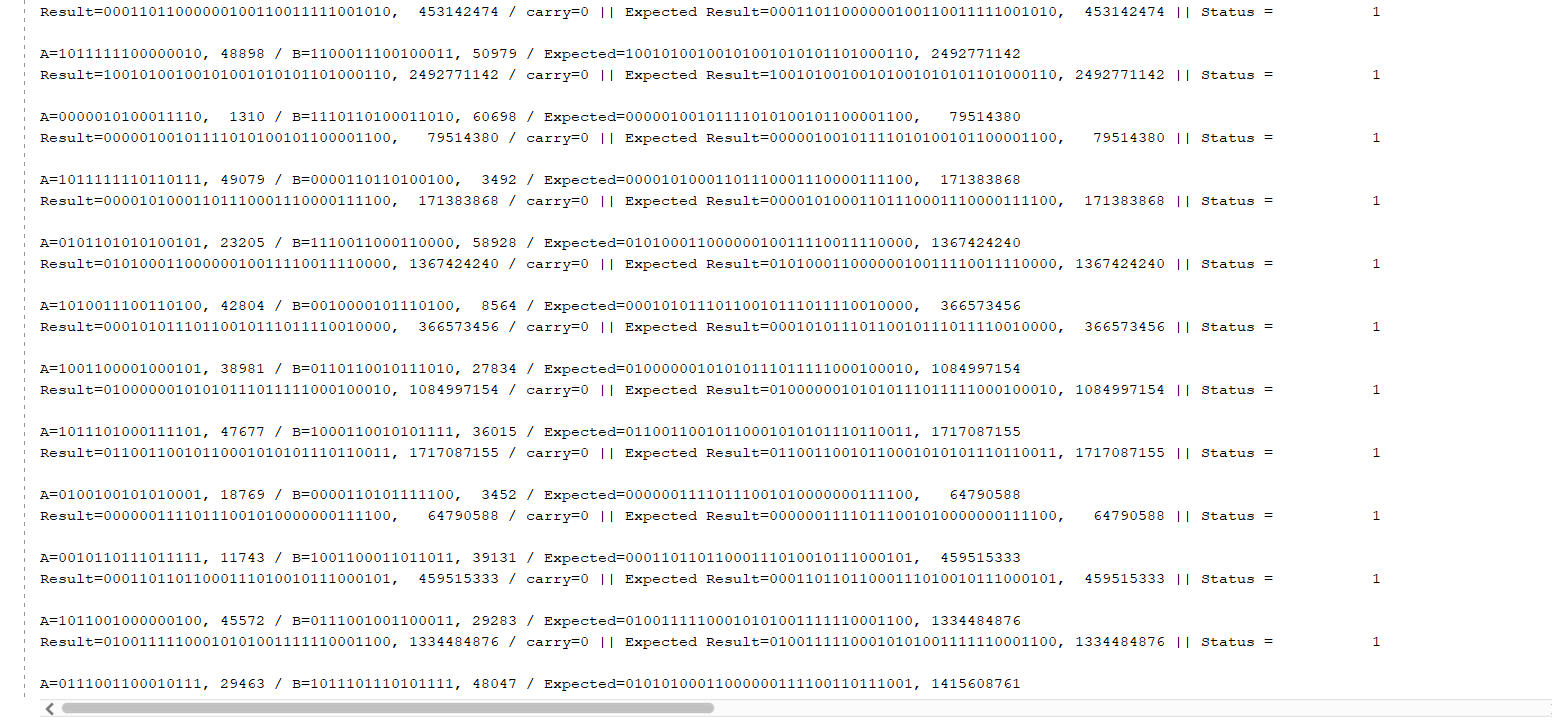
Şekil

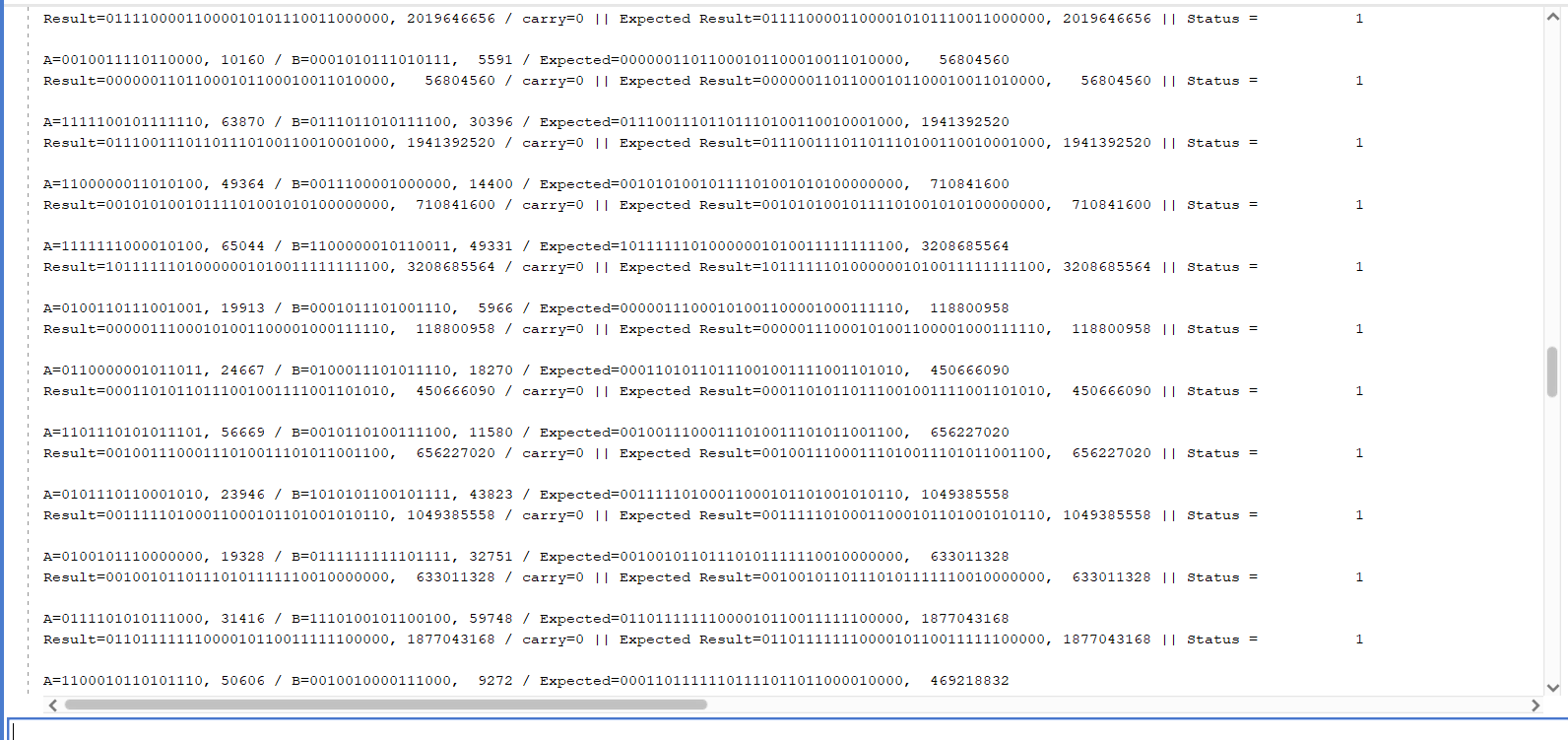
****

Şekil

1. **TCL Console**

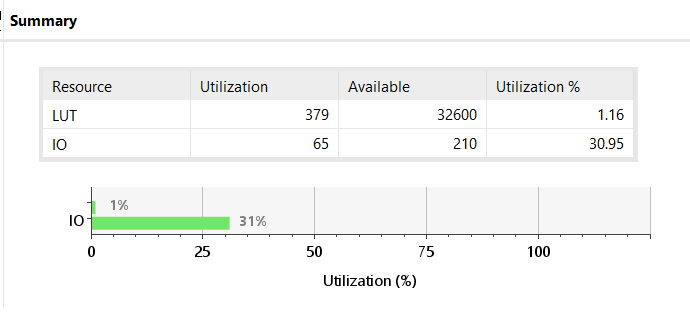
****

****

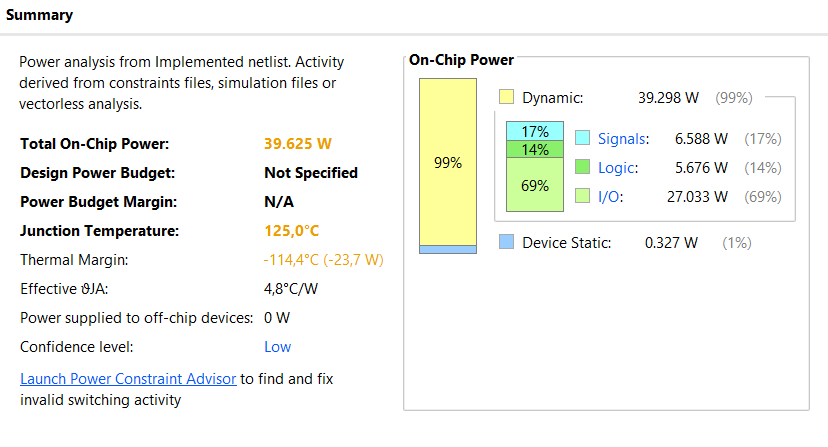
****

1. **Circuit Analysis Summary**

As seen in the circuit, 379 LUTs are used. It is seen that the I/O value is too high. In the partial product process, we saw that due to the version we use, Vivado does not allow the codes made in the experiments and we cannot use it in the top module. As a solution to this, we tried to find a solution by defining these values separately as output.

****

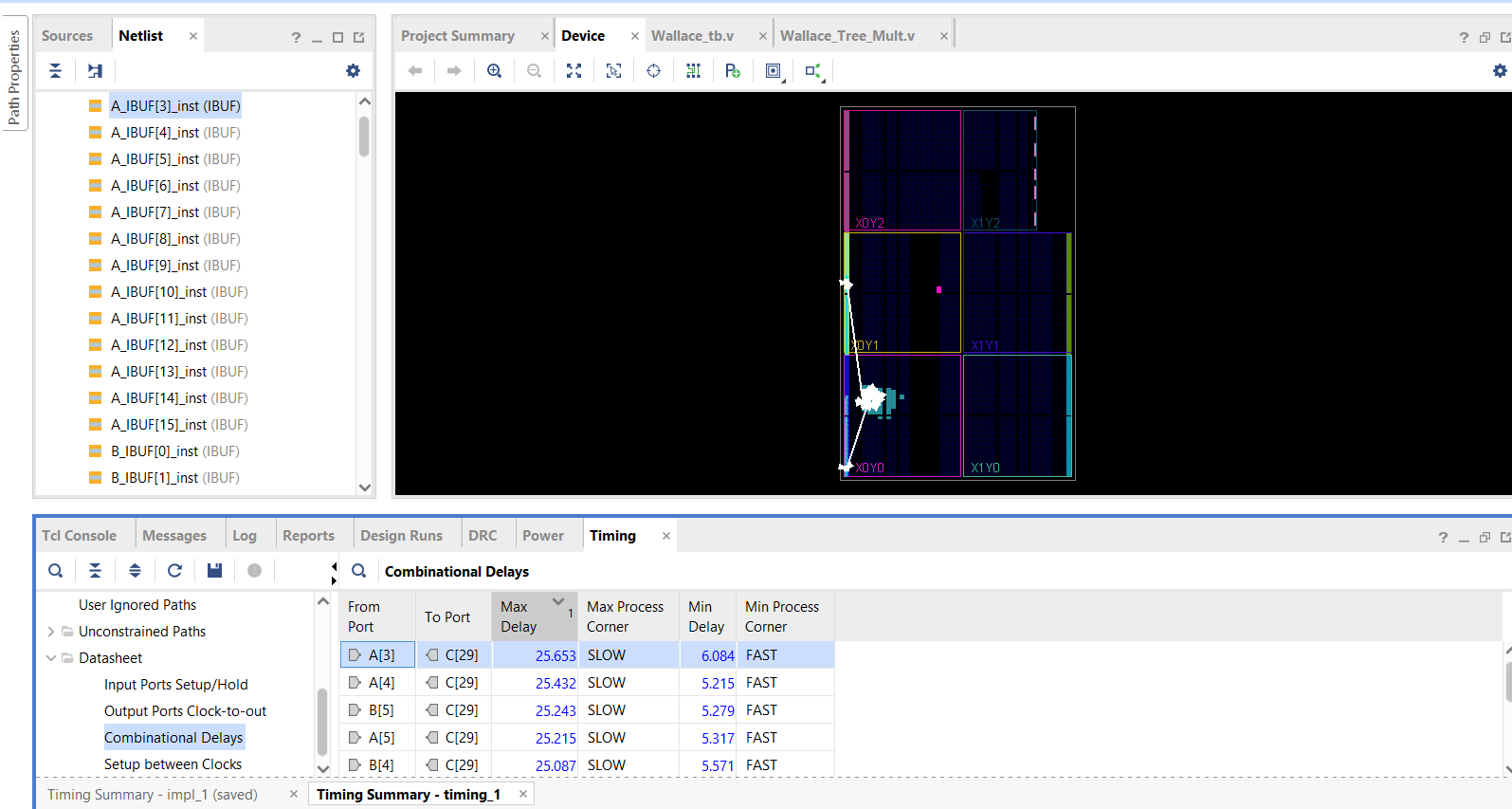
As seen below, the total chip on power value is 39,625W. The individual power consumptions are indicated on the right. Most of the power consumption is due to the high I/O rating. Although signal and logic consume approximately 20% power, I/O value accounts for most of the power consumption with 69% value.

****

We have listed the combinational delays data values after the implementation of the circuit. After sorting the delay values from the largest to the smallest, we saw that our maximum delay value is 26.563ns. If we convert this to frequency with the formula f=1/T, we can see that the maximum clock frequency is 37.646 Mhz. This value means that we cannot go above this value while we are simulating and tells us that we should simulate accordingly.



1. **Critical Path**

The critical path is the path between A[3] and C[29] with the longest delay value. As seen in the photo above, the max delay is 25.563ns. The critical path is specified in the device below.

1. **References**

B. Parhami, Computer arithmetic - algorithms and hardware designs, Oxford University Press, (2010)

1. **Work Package**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **NAME SURNAME** | **Assignment Research** | **Verilog Code** | **Python Code** | **Report** | **Vivado Outputs** |
| **Yiğit Bektaş GÜRSOY** | x | Wallace\_Tree.v  CSA.v  PP.v  Wallace\_tb.v | Random\_number.py | x | x |
| **Muhammed Velihan BAĞCI** | x | Wallace\_Tree.v  CSA.v  CLA.v  Wallace\_tb.v |  | x |  |