COL 215 - Hardware Assignment 3 : Matrix Multiplication

Suren (2021CS10072) and Pravar Kataria (2021CS10075)

November 13, 2022

§1 Objective

The aim of the assignment is to program the FPGA board to perform matrix multiplication. We get the input matrices as coe files and store the data in the ROM, and the final matrix product is stored in the RAM. Registers are to be used as intermediate storage devices.

§2 Design Procedure

1. We first create two distributed block ROMs using the vivado IP catalog which store the input matrices
2. In the same way, we create a RAM using the distributed memory generator in the vivado IP catalog.
3. A MAC unit is designed which accumulates a product of two terms of the matrices into an already stored sum.
4. A finite state machine enables the components in a desired sequence. This means that the finite state machine controls the path, i.e., the numbers are fetched from the memory, after that they are multiplied and added to the stored product. After the required number of computations, the output is stored in the corresponding address in the ram.
5. Once all the computation for the 128^2 locations in the RAM have been done, we can start using the switches on the board to display the output on the second segment display in hexadecimal format (since the ram has data width 16 bits).
6. To display, we use the same circuit as designed in assignment 1, i.e. use a clock divider circuit to display the four digits only using the four anodes and seven cathodes on the FPGA board.

§3 Design Decisions

The matrix multiplier is essentially controlled using a finite state machine which declares the sequence of operations that occur. For example, we must write the output on the RAM only after the entire sum has been computed.

§3.1 Register

A register is a storage device which stores the input data into an output signal but only at the rising edges of the clock. Thus, the register is an edge triggered device. We also have a write enable signal which tells whether any new data should be stored in the register. A signal for reseting the register data is also fed as input to the register.

§3.2 Finite State Machine

The finite state machine is a component which controls the states which the machine is in. It makes the machine consistent by telling the other components to function in the correct order. In our case, the finite state machine instructs the register to take out a value from the rom first, then instructs the mac to multiply and accumulate the product in an intermediate signal. After 128 iterations of this process, when one of the outputs is ready to be written as the output, the intermediate signal is sent to the ram, and the write enable signal for the ram is set to 1.

§3.3 RAM

A RAM component is generated using the ip catalog of vivado. This has data width 16 bits and depth 16384.

§3.4 ROM

Two ROM components are generated using the ip catalog of vivado. These have data width 8 bits and depth 16384.

§3.5 MAC

A Multiplier Accumulator Component takes two numbers as inputs and adds it to an already stored partial sum.

§4 Block Diagram

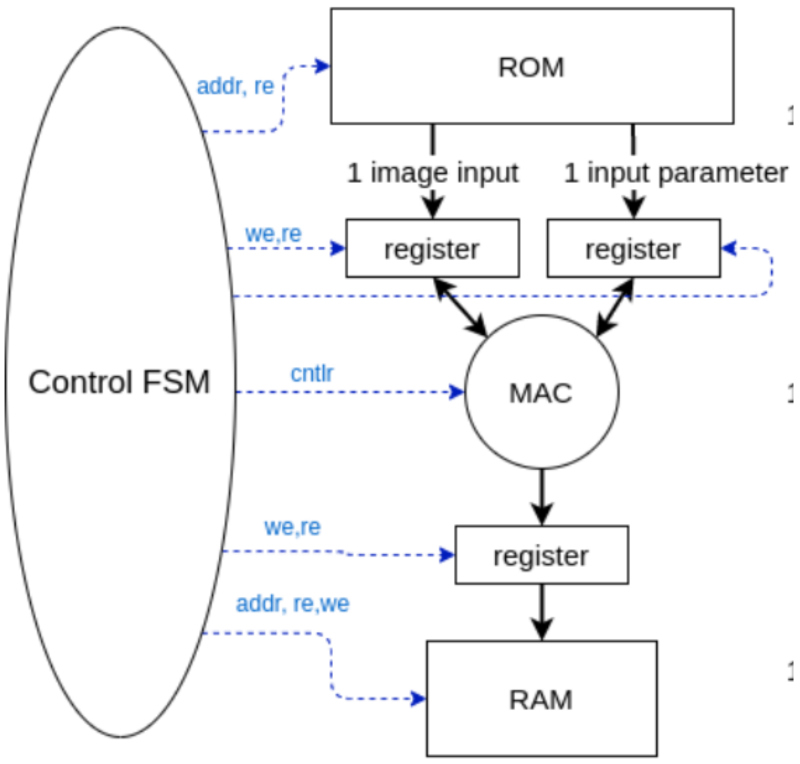


Figure 1: This figure shows the block diagram for the matrix multiplication module.

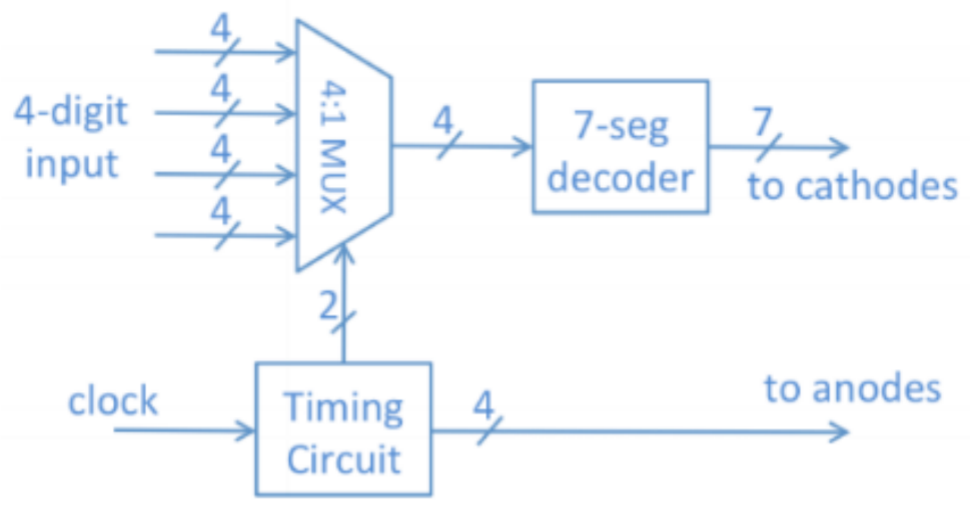


Figure 2: This figure shows the block diagram for the matrix multiplication module.

§5 Simulation Snapshots

§5.1 Registers

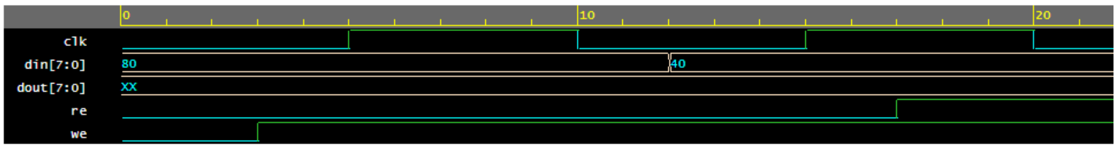


Figure 3: This figure shows the simulation waveform for the register component. This waveform has been generated using EDA Playground.

§5.2 MAC

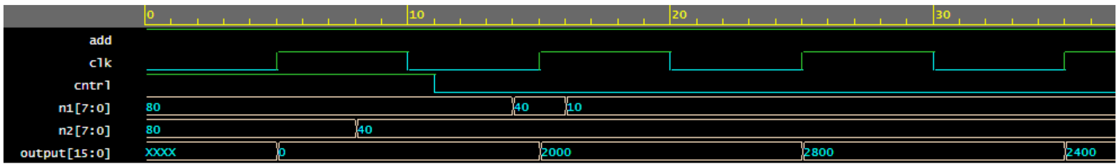


Figure 4: This figure shows the simulation waveform for the mac component. The waveform is generated using EDA Playground.

§5.3 RAM

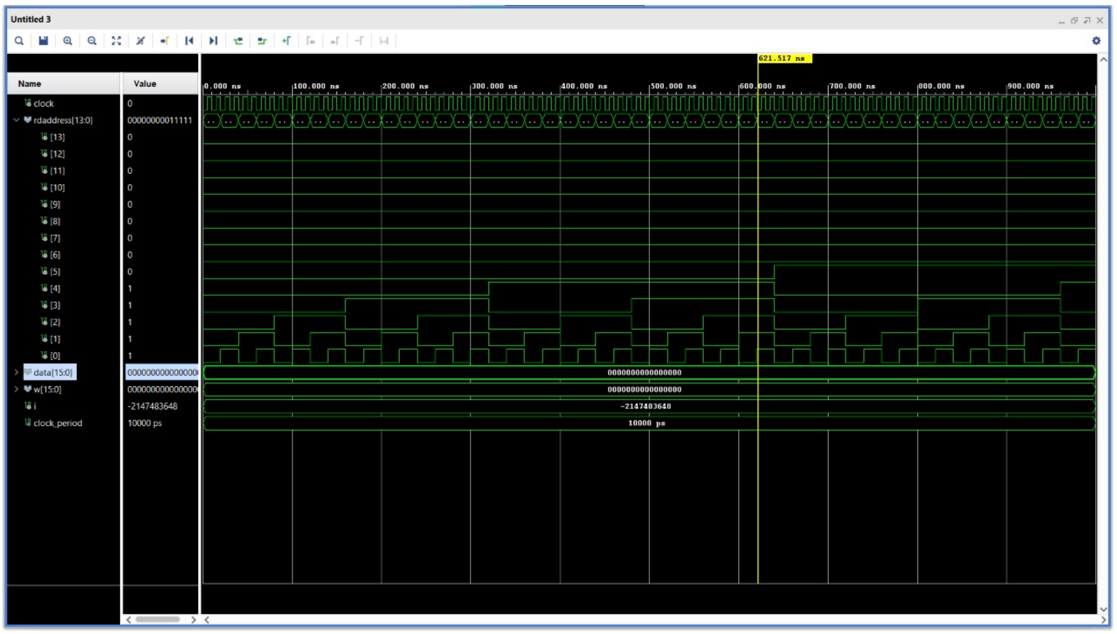


Figure 5: This figure shows the simulation waveforms for the RAM component. This was not required, strictly speaking, since the RAM was created using the vivado ip catalog. We created this simulation waveform just to be sure that the RAM component is working correctly. This has been included in the report for the sake of completeness.

§5.4 ROM

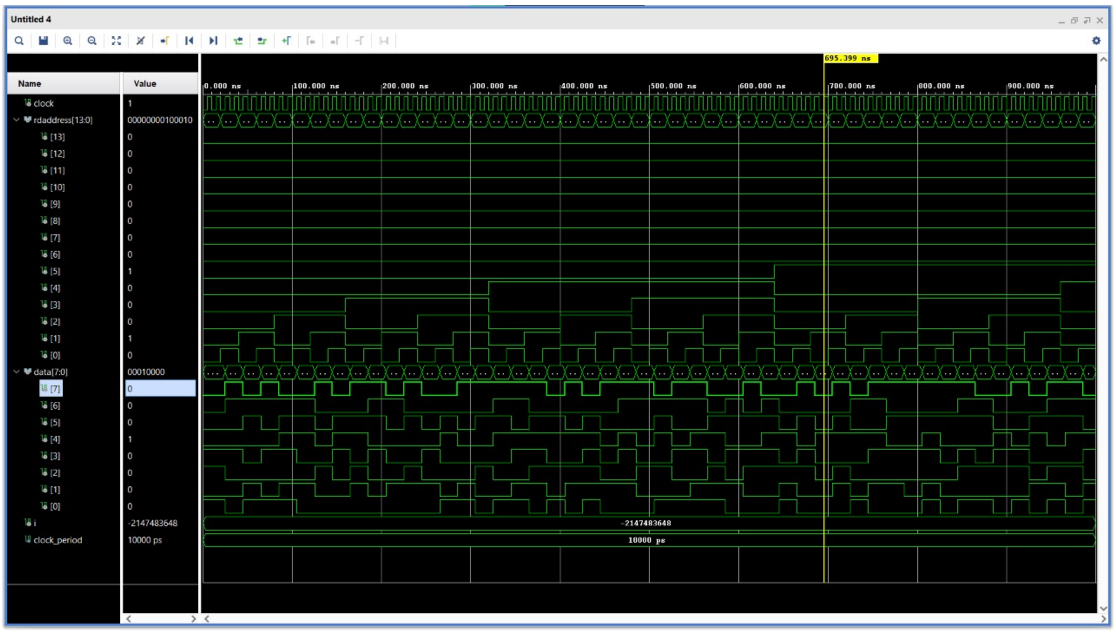
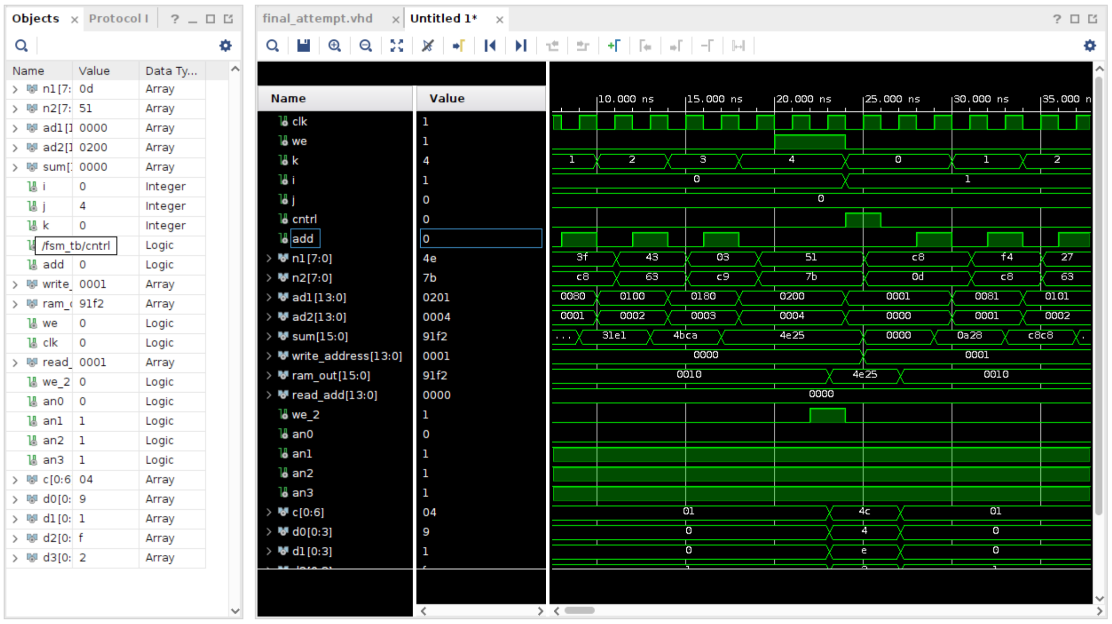


Figure 6: This figure shows the simulation waveforms for the ROM component. This was not required, strictly speaking, since the ROM was created using the vivado ip catalog. We created this simulation waveform just to be sure that the ROM component is working correctly. This has been included in the report for the sake of completeness.

§5.5 Finite State Machine

To create these simulations, we have taken 4x4 matrices instead of 128x128 for the sake of simplicity of reading the simulation.

The control signal turns to 1 only after one whole computation is done. The write enable signal for the ram also turns to 1 only after one whole computation is done. The integer signal k is effectively the point at which we are in the present computation (for computing the entry (i, j) of the output matrix, we need to sum the product of the (i, k) entry and the (k, j) entry of the corresponding matrices. We can clearly see that the changes in the signals i,j and k in the simulation follow the expected pattern.



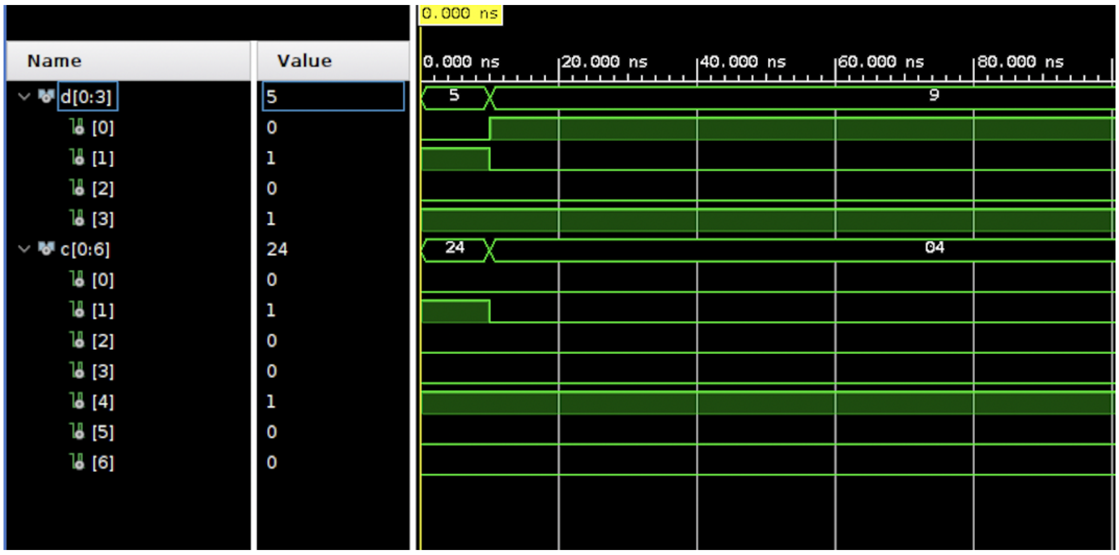
Name	Value	Data Ty...
n1[7:	0d	Array
n2[7:	51	Array
ad1[1	0000	Array
ad2[1	0200	Array
sum[:	0000	Array
i	0	Integer
j	4	Integer
k	0	Integer
/fsm_tb/cntrl		Logic
add	0	Logic
write_	0001	Array
ram_(	91f2	Array
we	0	Logic
clk	0	Logic
read_	0001	Array
we_2	0	Logic
an0	0	Logic
an1	1	Logic
an2	1	Logic
an3	1	Logic
c[0:6	04	Array
d0[0:	9	Array
d1[0:	1	Array
d2[0:	f	Array
d3[0:	2	Array

Name	Value
clk	1
we	1
k	4
i	1
j	0
cntrl	0
add	0
n1[7:0]	4e
n2[7:0]	7b
ad1[13:0]	0201
ad2[13:0]	0004
sum[15:0]	91f2
write_address[13:0]	0001
ram_out[15:0]	91f2
read_add[13:0]	0000
we_2	1
an0	0
an1	1
an2	1
an3	1
c[0:6]	04
d0[0:3]	9
d1[0:3]	1

Figure 7: This figure shows the results (zoomed in for clarity) for the testbench of the finite state machine component.

§5.6 Display Component

§5.6.1 Decoder snapshots



Name	Value
d[0:3]	5
[0]	0
[1]	1
[2]	0
[3]	1
c[0:6]	24
[0]	0
[1]	1
[2]	0
[3]	0
[4]	1
[5]	0
[6]	0

Figure 8: This figure shows the results (zoomed in for clarity) for the testbench of the decoder component.

§5.6.2 Timing circuit testbench snapshots

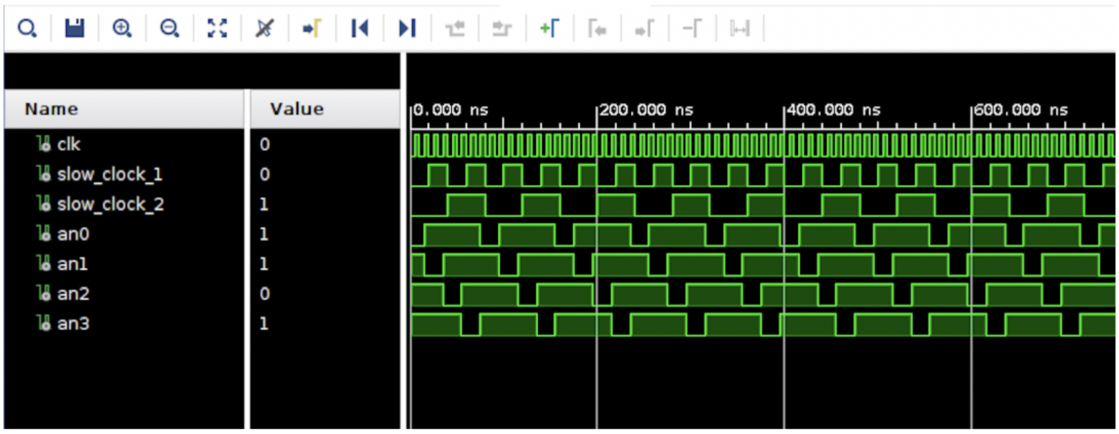


Figure 9: This figure shows the results (zoomed in for clarity) for the testbench of the timing circuit component.

To create these simulation outputs we have taken the time of the slower clocks to be 20 ns and 40 ns just for better visibility in the simulation outputs.
We can clearly see that only one of the anode outputs is zero at a given point of time. Since the anodes and cathodes are active low pins, only one of the seven segment displays is on at a given moment.

§5.6.3 Mux testbench snapshots

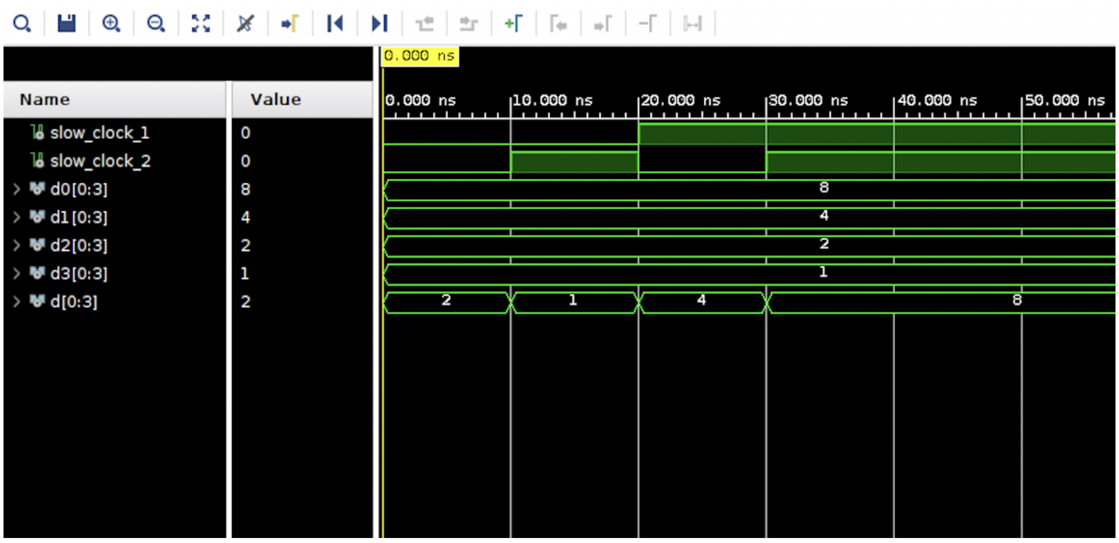


Figure 10: This figure shows the results (zoomed in for clarity) for the testbench of the mux component.

We can see that we have inputs `d0,d1,d2,d3` and an output `d` which is one out of the 4 inputs and is selected according to the two selection lines which are named `slow_clock_1` and `slow_clock_2` (they are actually outputs from the timing circuit in the final design).

§6 Netlist Representation

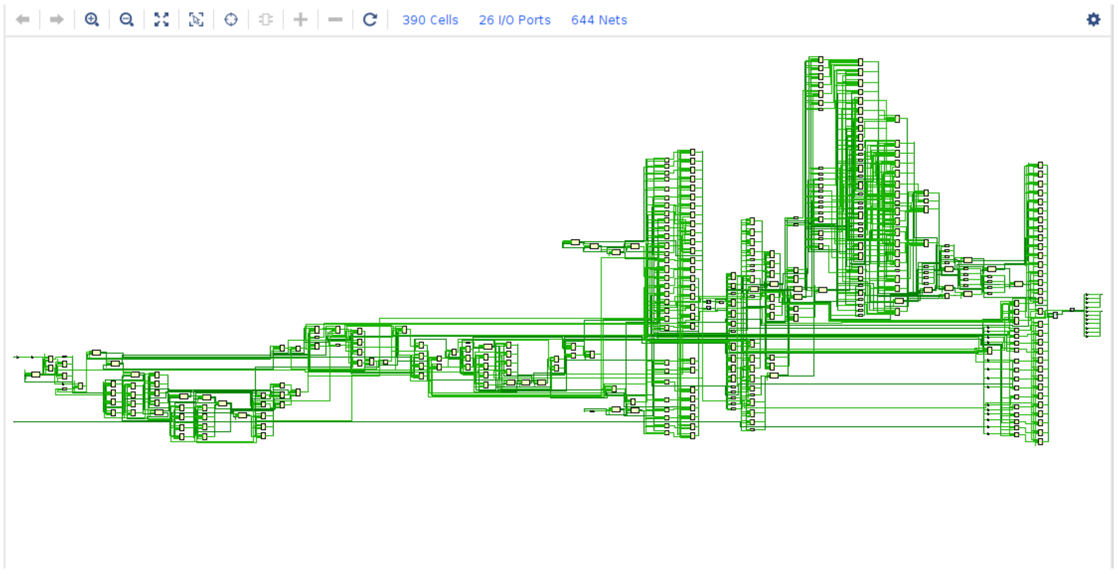


Figure 11: This figure shows the netlist representation of the final design.

§7 Implemented Design

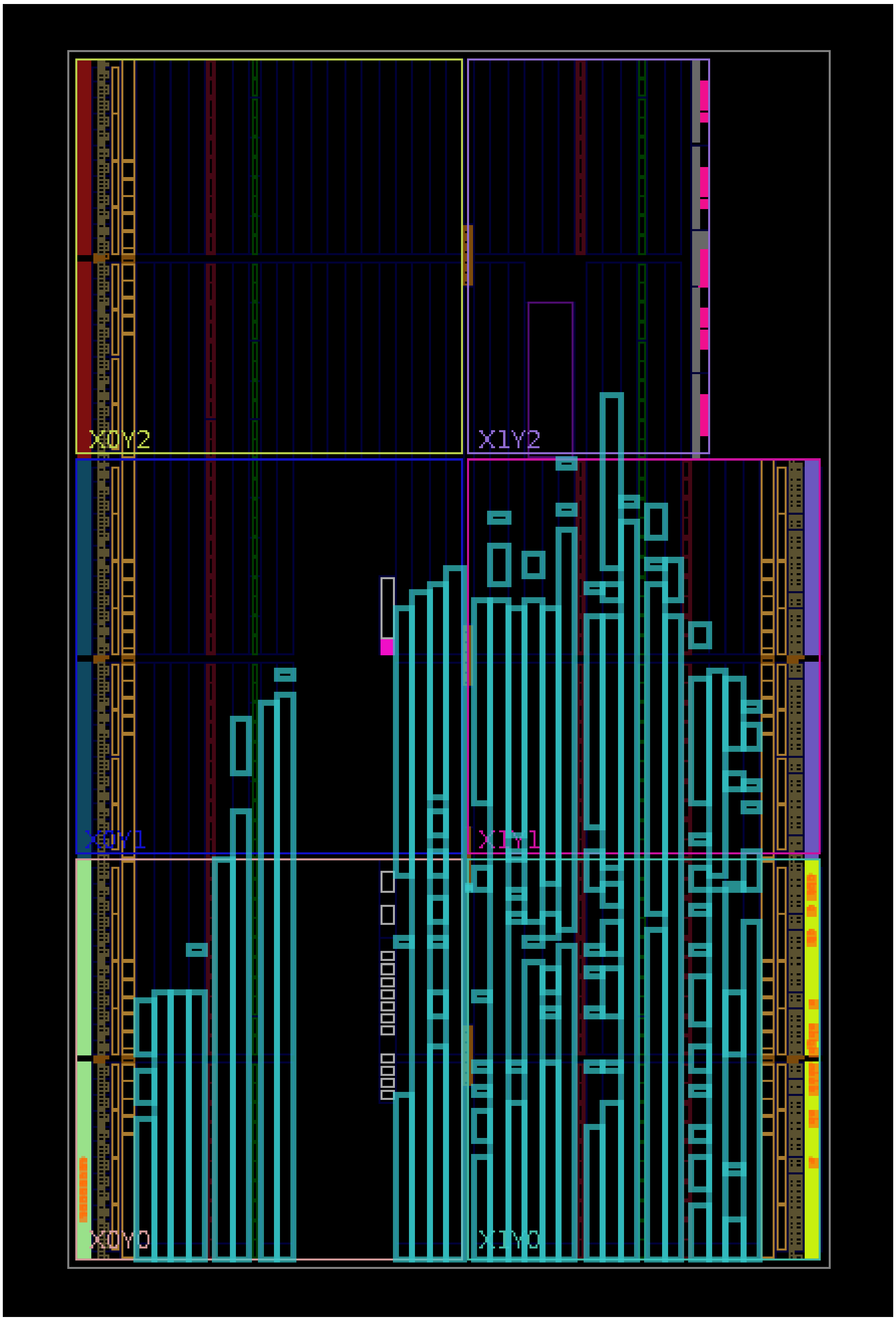
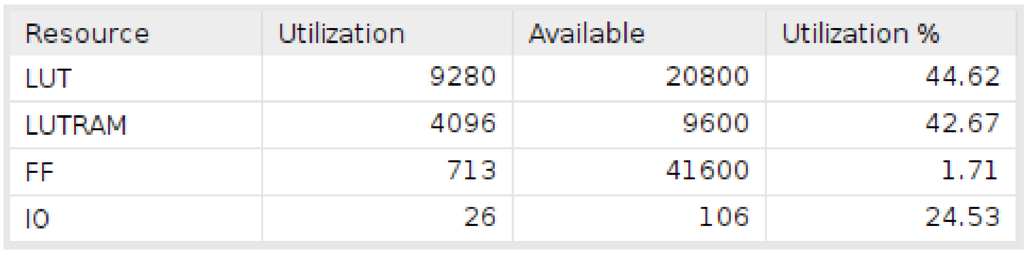


Figure 12: This figure shows the implementation of the final design.

§8 Hardware Utilization



Resource	Utilization	Available	Utilization %
LUT	9280	20800	44.62
LUTRAM	4096	9600	42.67
FF	713	41600	1.71
IO	26	106	24.53

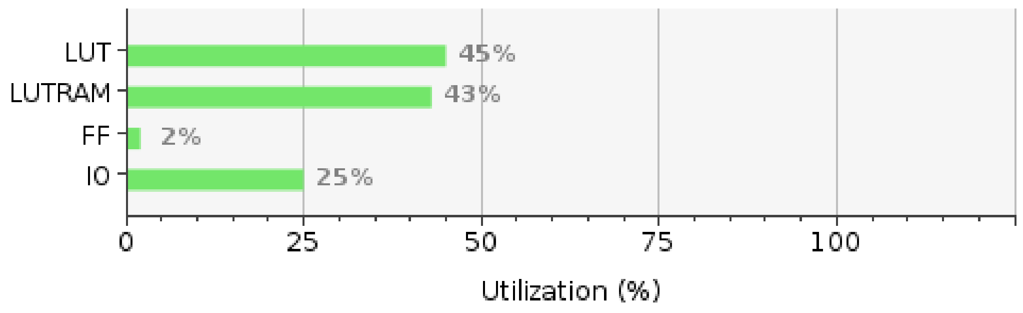
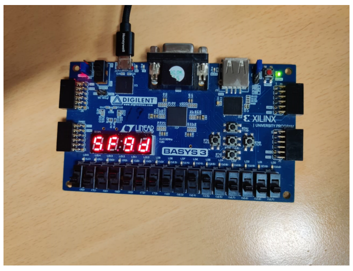
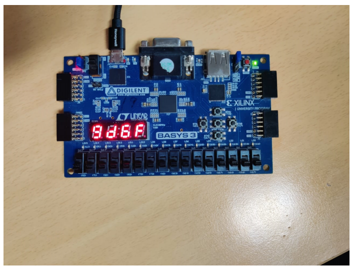


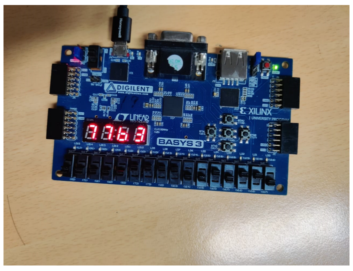
Figure 13: This figure shows the hardware utilization of the final design.

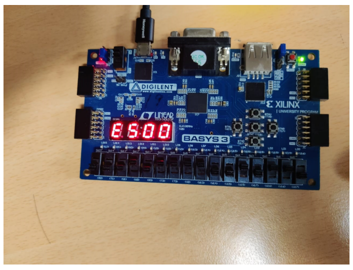
Component	Count
Flip Flops	713
LUT	9280
IO	26
LUTRAM	4096

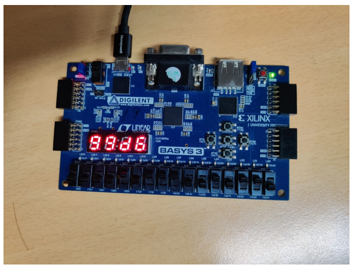
§9 Lab Work


(a) 5F9D


(b) 9D6F


(c) 77B3


(d) E500


(e) 99D6

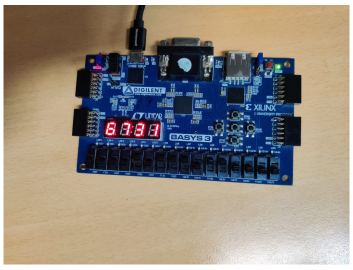

(f) 6731

Figure 14: Some examples of outputs displayed on the board