ELEC6233 – High-Level Synthesis

Ahmed Soliman
as8u22
EMECS – Embedded Computing Systems – Part 1
Tomasz Kazmierski

ABSTRACT: *A high level synthesis coursework done on a FFT butterfly algorithm from a given pseudocode converted to RTL level code in System Verilog. A PicoMIPS Architecture was built to implement the design. It was simulated on Modelsim, tested using a testbench, synthesized on Quartus, and uploaded on a FPGA for real life testing.*

1. 1. Introduction

The objective of this assignment is to implement Fast Fourier Transform butterfly algorithm using a pseudocode on am FPGA development board (Cyclone V).

This process starts by executing High level synthesis of converting pseudocode to flow chart then perform scheduling and binding down to our design implementation in Systemverilog which is synthesized on Quartus to produce RTL design of our hardware implementation.

The design was successfully simulated on Modelsim through a testbench and synthesized on Quartus then tested on FPGA board and showed correct behavior as Simulation predicted.

1. 2. Overall architecture of the design

In order to implement the FFT butterfly design from High level synthesis down to RTL design. First, we need to have a look at the equations needed to be implemented which can be seen in figure 1.

$$Re\ y + j\ Im\ y = (Re\ a + j\ Im\ a) + (Re\ b + j\ Im\ b) * (Re\ w + j\ Im\ w)$$
$$Re\ z + j\ Im\ z = (Re\ a + j\ Im\ a) - (Re\ b + j\ Im\ b) * (Re\ w + j\ Im\ w)$$

Figure 1. FFT Equations

Then after understanding the equations. We needed to translate the pseudocode given for the inputs and outputs sequence to a data flow chart (Figure 2) which will help us in the next step of scheduling the processes needed for inputs and bind those operations with our available resources (ALUs & Multipliers).

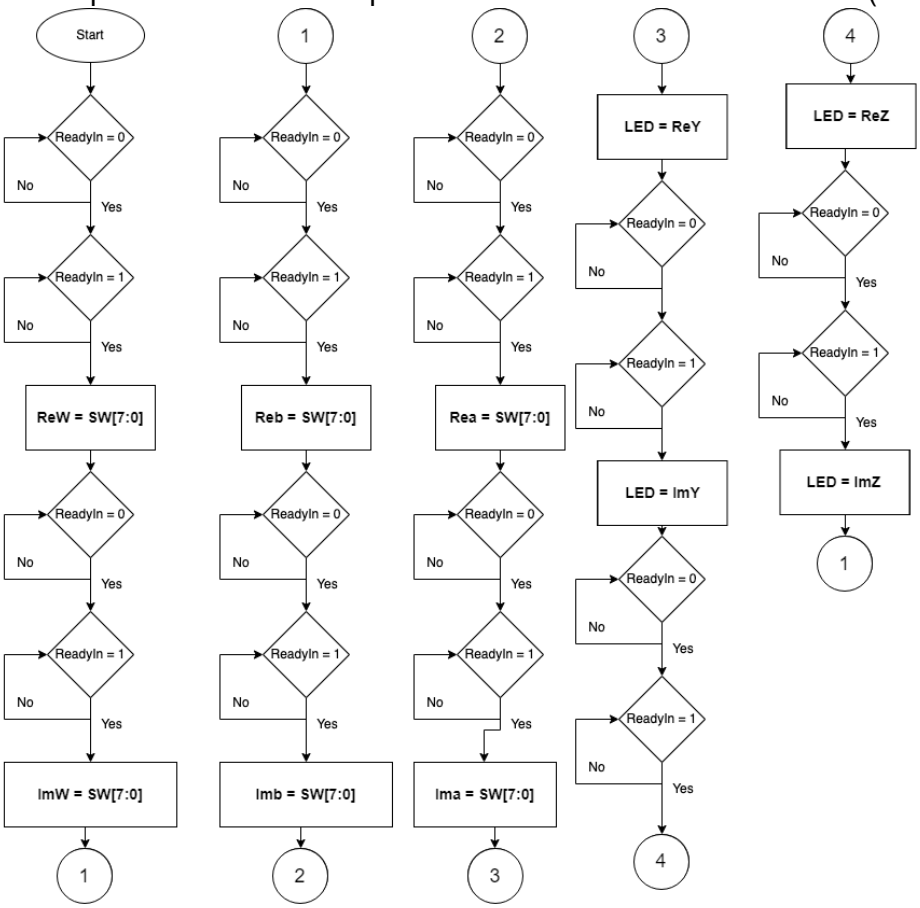

Figure 2. Flow Chart of Pseudocode

From the flow chart we now know the sequence needed to be implemented. And in order for it to be implemented a PicoMIPS embedded processor was used in which program memory carry the Sequence of instruction needed to implement the code then it uses the registers to access and store data, these data registers are given to an ALU and a multiplier to do the operations, finally the results are stored in specific registers and given to output LED to be displayed.
Choosing this implementation design meant that we have 1 ALU and 1 Multiplier as this is the conventional design structure for a PicoMIPS. Based on that, minimum latency-Resource constrained scheduling was made for our code which can be seen in figure 3. Diagram on the left shows every operation needed for our code while diagram on the right shows the operations after getting scheduled to specific cycles based on the table below. Binding can be seen as well for the ALU and MUL through the colors of the figure.

Candidates			
MUL	**ALU**	**Cycle**	**Chosen**
1,2,3,6		1	1
2,3,6	4,7	2	2,4
3,6	7,5,8	3	3,7
6	5,8,9,12	4	6,5
	8,9,12,10	5	8
	9,12,10,11	6	9
	12,10,11	7	11
	10,12	8	12
	10	9	10

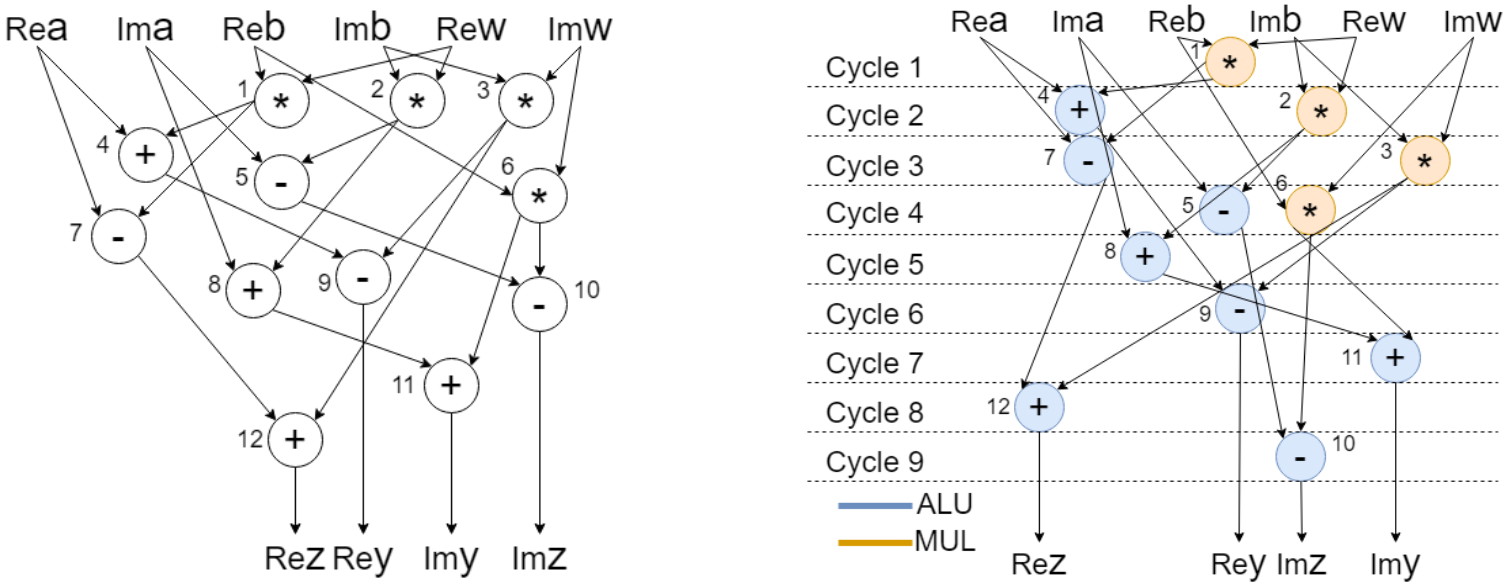

Figure 3. Scheduling of operation

Even though our design is based on PicoMIPS architecture, a simple binding diagram can be seen in Figure 4 were inputs to MUL and ALU are mapped and outputs will be saved in registers that will be used again in further calculations.

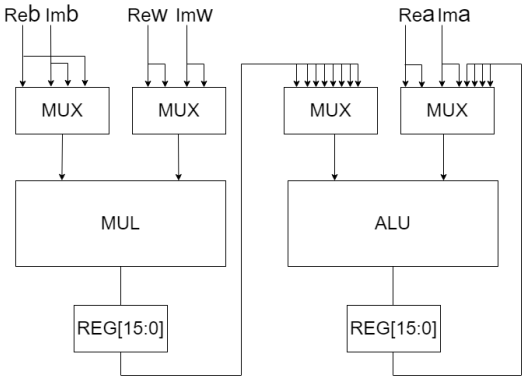

Figure 4. Binding diagram

After finishing those steps, the design was made on Modelsim and a testbench was written to test its functionality. As seen from the testbench inputs are given based on pseudocode sequence then outputs are displayed then code repeats from the inputs of Reb. For the set of inputs of the testbench, the expected outputs are shown in the comments which we will compare with the Modelsim waveform simulation.

D:/UoS/Digital System Synthesis/Coursework 2/Code Files/testbench.sv (/testbench) - Default

```
25    SW[8] = '0;
26    #50ns SW[7:0]= 8'b11110000; //ReW= -0.125
27    #70ns SW[8] = '1;
28    #100ns SW[8] = '0;
29    #10ns SW[7:0]=8'b01100000;  //ImW=  0.75
30    #70ns SW[8] = '1;
31    #100ns SW[8] = '0;
32    #10ns SW[7:0]= 8'b00010110; //Reb = 22
33    #70ns SW[8] = '1;
34    #100ns SW[8] = '0;
35    #10ns SW[7:0]= 8'b00000101; //Imb = 5
36    #70ns SW[8] = '1;
37    #100ns SW[8] = '0;
38    #10ns SW[7:0]= 8'b11100101;  //Rea = -27
39    #70ns SW[8] = '1;
40    #100ns SW[8] = '0;
41    #10ns SW[7:0]= 8'b00110100;  //Ima = 52
42    #300ns SW[8] = '1;
43    #60ns SW[8] = '0; //display Rey = -33.5
44    #300ns SW[8] = '1;
45    #60ns SW[8] = '0; //display Imy = 67.875
46    #300ns SW[8] = '1;
47    #60ns SW[8] = '0; //display Rez = -20.5
48    #300ns SW[8] = '1;
49    #60ns SW[8] = '0; //display Imz = 36.125
```

D:/UoS/Digital System Synthesis/Coursework 2/Code Files/testbench.sv (/testbench) - Default

```
46    #300ns SW[8] = '1;
47    #60ns SW[8] = '0; //display Rez = -20.5
48    #300ns SW[8] = '1;
49    #60ns SW[8] = '0; //display Imz = 36.125
50
51    #10ns SW[7:0]= 8'b00000001; //Reb = 1
52    #70ns SW[8] = '1;
53    #100ns SW[8] = '0;
54    #10ns SW[7:0]= 8'b00000010; //Imb = 2
55    #70ns SW[8] = '1;
56    #100ns SW[8] = '0;
57    #10ns SW[7:0]= 8'b00000011;  //Rea = 3
58    #70ns SW[8] = '1;
59    #100ns SW[8] = '0;
60    #10ns SW[7:0]= 8'b00000111;  //Ima = 7
61    #300ns SW[8] = '1;
62    #60ns SW[8] = '0; //display Rey = 1.375
63    #300ns SW[8] = '1;
64    #60ns SW[8] = '0; //display Imy = 7.5
65    #300ns SW[8] = '1;
66    #60ns SW[8] = '0; //display Rez = 4.625
67    #300ns SW[8] = '1;
68    #60ns SW[8] = '0; //display Imz = 6.5
69
70
```

Figure 5. testbench for simulation

From the waveform LED signal (Orange box) we can see the values we expected with a slight variation due to truncation of multiplication process which proves that our design worked as intended.

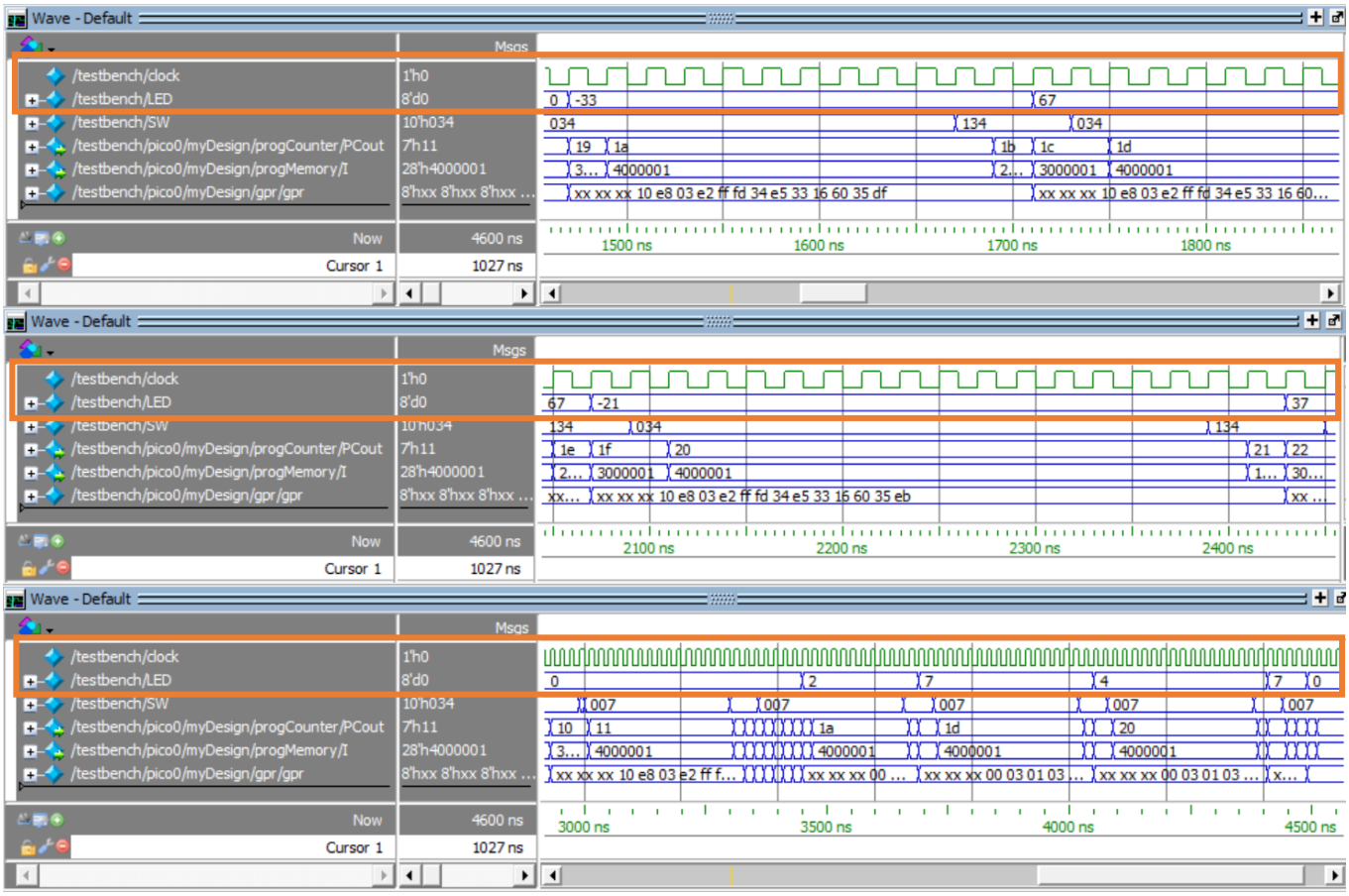

Figure 6. Modelsim waveform test results

1. 3. Details of hardware blocks

As explained in the previous section in order to implement the design from the high level synthesize process that was done, a PicoMIPS embedded processor architecture was used. From the block diagram below we can see memory carrying the code instructions, PC giving the address, decoder translating instructions functions and giving out flags, ALU & MUL to do calculations.

PC gives address of size 7 bits which determines which instruction is executed. Its value is incremented each clock cycle or jumps to a relative branch given by the instruction. The choice is made by the decoder depending on the operation and flags inputs.

```
always_comb
  if (PCincr)
       Rbranch = { {(Psize-1){1'b0}}, 1'b1};
  else Rbranch =  Branchaddr;
```

```
always_ff @ ( posedge clk, posedge reset) // async reset
  if (reset) // sync reset
     PCout <= {Psize{1'b0}};
  else if (PCincr | PCrelbranch) // increment or relative branch
     PCout <= PCout + Rbranch; // 1 adder does both
```

Figure 7. PC increment or Branch jump

In the decoder, it takes opcode function and flags to give output flags for REG and ALU. For example (ADDMUL) is an instruction to add and multiply which results in setting the output flags (W1, W2) 1 which means that we store the values coming from ALU & MUL

```
case (opcode)

  `ADD, `LDI, `SUB : begin // register-register
            w1 = 1'b1; // write result to dest register
          end
  `MUL: begin // register-immediate
            w2 = 1'b1; // write result to dest register
          end
  `ADDMUL, `SUBMUL : begin // register-immediate
            w1 = 1'b1; // write result to dest register
            w2 = 1'b1; // write result to dest register
          end

 // branches
     `BEQ: takeBranch = flag; // branch if Z==1
     `BNE: takeBranch = ~flag; // branch if Z==0

     default:
```

Figure 8. Decoder Implementation

The Register module takes 6 addresses from the program memory (2 ALU source registers, 1 ALU destination register, 2 MUL source registers, 1 MUL destination register)

```
always_ff @ (posedge clk)
begin
        if (w1)
    gpr[R1addr3-3] <= Wdata1;

        if (w2)
    gpr[R2addr3-3] <= Wdata2;
end
```

```
  if (R1addr1==4'd0)
        R1data1 =  {n{1'b0}};
else if (R1addr1==4'b0001)
        R1data1={7'b0000000,SW[8]};
else if (R1addr1==4'b0010)
        R1data1=SW[7:0];
else  R1data1 = gpr[R1addr1-3];
```

Figure 9. Reading and writing in Registers

ALU and MUL is the calculations modules where they take the inputs perform the calculation and give output to register to be stored. In ALU process can be addition, subtraction, comparing. While MUL only perform multiplication operation.

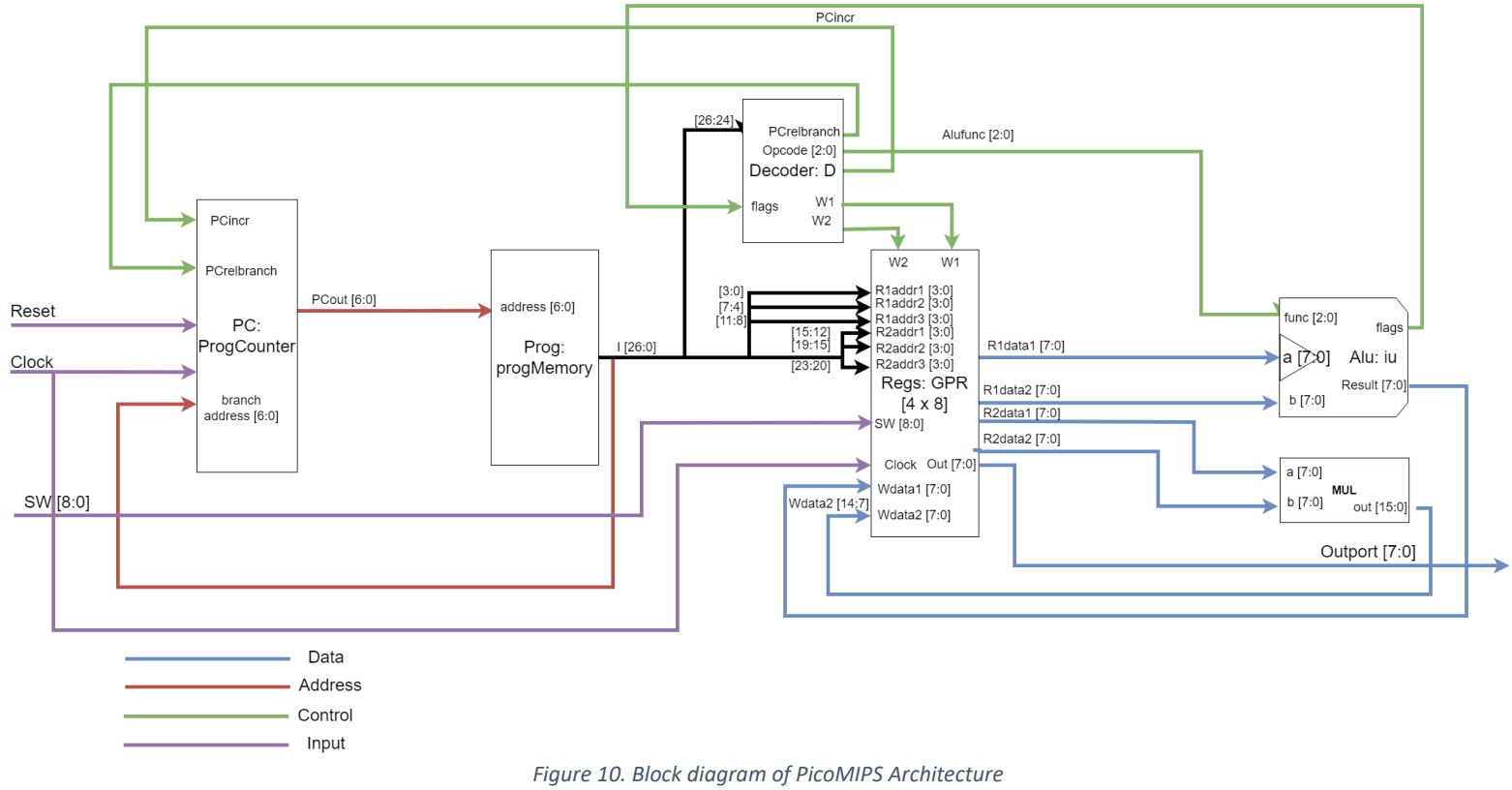

Figure 10. Block diagram of PicoMIPS Architecture

Modifications were made to standard PicoMIPS to accommodate our design. First, the Size of the Instruction increased to include 6 register addresses (4 bits) and 3-bit opcode, 8 bits of immediate address were removed and for branching the value is stored in the place of source registers of multiplier as seen in the figure.

```
1
2   // Isize = 27 bits
3   // format: 3b opcode, 4b %dM, 4b %sM, 4b %sM, 4b %dM, 4b %sM, 4b %sM
4   //
5   // HEX ////////// BINARY ////////////////////// ASSEMBLER /////////
6   5000330 // 16'b101_0000_0000_0000 0011_0011_0000    LOAD  %3, %3, %0;  %3 = %0, load reg 0 in reg 3
7   4000001 // 16'b100_0000_0000_0000 0000_0000_0001    BEQ %0, %1, -1; branch if %1=0
```

MUL Registers

ALU Registers

Relative Branch address

Figure 11. Instruction code structure

1. 5. FPGA implementation

To synthesize our design, we used Quartus to upload our code to the FPGA board and test it in real life. A module called counter is used with our design to speed down the clock cycles which helps in solving problem of debouncing of switches when it runs on 50 MHz speed. Pin allocation was set in the qsf file to connect the switches and LEDs to our code variables. From the flow summary we can see hat the code was synthesized successfully.

Flow Summary

<<Filter>>

Flow Status	Successful - Sun May 7 22:33:58 2023
Quartus Prime Version	22.1std.1 Build 917 02/14/2023 SC Lite Edition
Revision Name	picoMIPS4test
Top-level Entity Name	picoMIPS4test
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	217 / 32,070 (< 1 %)
Total registers	111
Total pins	19 / 457 (4 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total DSP Blocks	1 / 87 (1 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 12. Flow summary of Quartus synthesize

Code was then uploaded on FPGA and inputs were given as the testbench values and LED outputs showed same values as expected. Also, after synthesizing the code, we can see the produced RTL diagram of Quartus which shows exact blocks as the one made above.

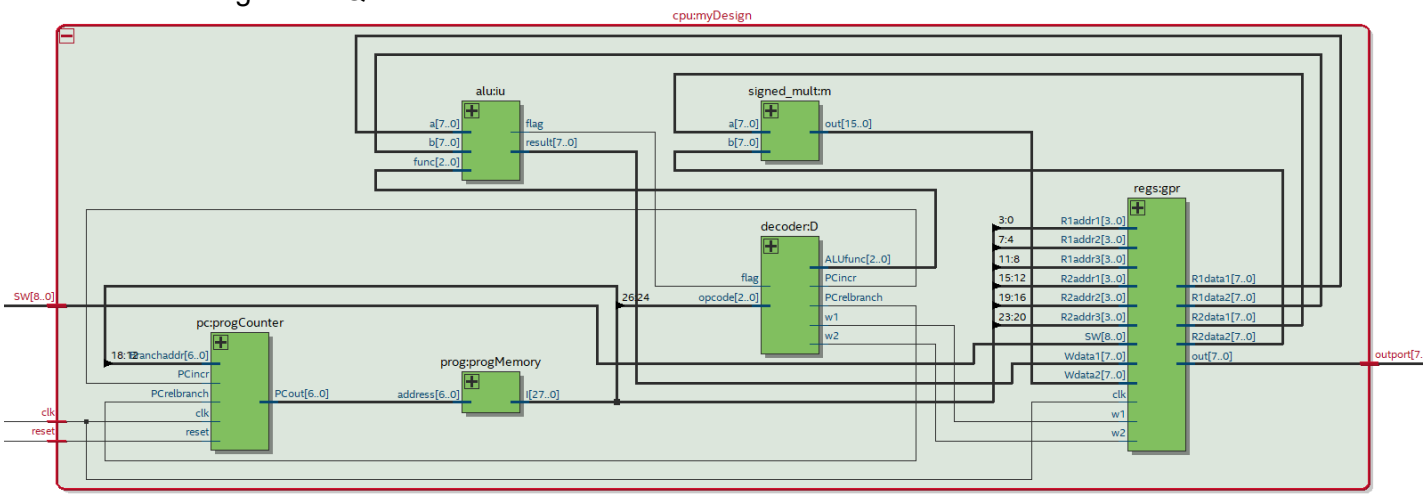

Figure 13. RTL diagram of PicoMIPS

Finally, we can see pictures of results on the FPGA's LEDs that shows same values as the testbench for the same input values.

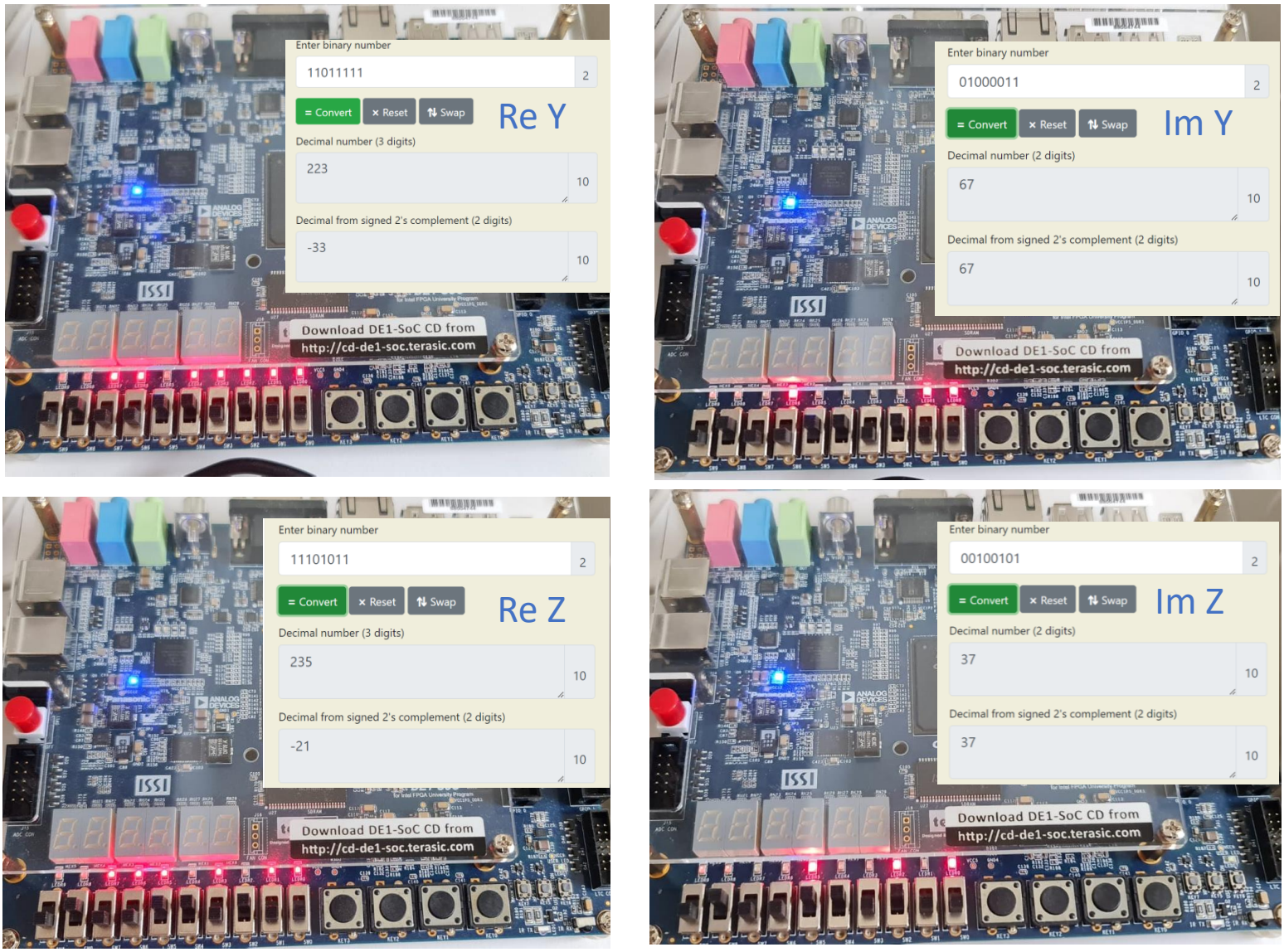

Figure 14. Output Result on FPGA Board

1. 6. Conclusion

In conclusion, this report has demonstrated that it is possible to implement the FFT butterfly algorithm using high-level synthesis and a PicoMIPS embedded processor architecture. The design was tested on an FPGA, and its behavior was found to be consistent with the simulation results. The report has shown that the chosen implementation design is a viable solution for this problem. However, there are areas for improvement or further extension of the design. For example, using a different processor architecture or optimizing the scheduling and binding process to reduce latency. Also using state machine might help in improving the design.

Overall, this project has provided valuable insight into the high-level synthesis process and the design of embedded processor architectures. The experience gained from this project will undoubtedly prove useful in future projects in the field.

1. 7. References

Zwolinski, M. (2017). Synthesis Walk-Through – Quartus and SystemVerilog [PDF file]. Retrieved from https://secure.ecs.soton.ac.uk/notes/adv_hdl/2017/QuartusWalkthrough.pdf

(2014). DE1-SoC User Manual. Imperial College London. Retrieved from http://www.ee.ic.ac.uk/pcheung/teaching/ee2_digital/de1-soc_user_manual.pdf