ELEC6234 – Embedded Processors

Coursework Report – picoMIPS implementation

Ye Lin
yl12y23
MSc Internet of Things
Tomasz Kazmierski

1. Introduction

1.1 Overview of picoMIPS design

The assignment aimed to create a picoMIPS architecture processor with n bits specifically for executing affine transformations on graphic pixels. This processor was successfully implemented on an Altera FPGA development board, utilizing minimal logic resources. Achieving this objective, a machine-level program has been developed to execute a customized instruction set tailored for handling generic affine transformation algorithms. After inputting X1 and Y1 in the range -128 to 127, X2 and Y2 are output through the affine transformation algorithm.

In this machine-level program, the 6 constants given in the assessment requirements (the data sets 1) are used. The instruction design for this task was inspired by the MIPS instruction set, specifically the BEQ and BNE branch instructions. I specified the register numbers (%6) and the SW8 interface to implement the handshake function. Additionally, I allocated a specific register (%5) connected to switches SW0-7 to read the X1 and Y1 coordinate values and assigned another register (%4) connected to LEDs to display the transformed pixel coordinates, thus reducing hardware resources. Furthermore, I optimized resource usage by effectively utilizing the %1, %2, and %3 registers to store intermediate calculation results. Ultimately, only four general-purpose registers (GPR) were used.

Design Details Form

Total Cost: 57	ALMs: 57	Memory bits: 0	Multipliers:1

Flow Summary

<<Filter>>

Flow Status	Successful - Mon Apr 15 14:32:45 2024
Quartus Prime Version	16.1.2 Build 203 01/18/2017 SJ Standard Edition
Revision Name	picoMIPS
Top-level Entity Name	cpu
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	57 / 32,070 (< 1 %)
Total registers	36
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 %)

Instruction Set

A total of 6 instrucions have been implemented to ensure that the basic operations are satisfied. BEQ and BNE is designed for handshaking functionality.

LD	Put value stored in the source register to the destination register
ADD	Adds the values stored in the source and destination registers, and then stores the result in the destination register
ADDI	Adds the immediate value and the value stored in the source register, and then store the result in the destination register
MUL	Multiplies the values stored in the source and destination registers, then stores the result in the destination register
BEQ	Branch if registers are equal
BNE	Branch if registers are not equal

Instruction Format

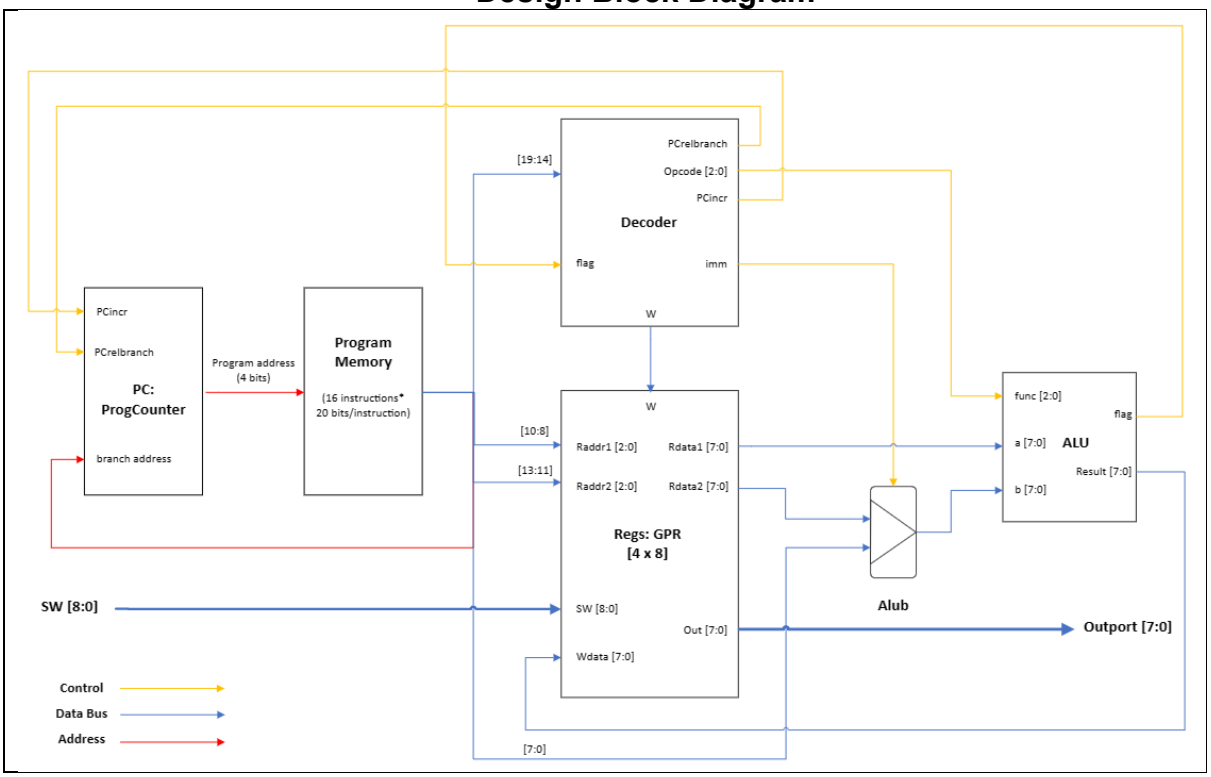
Databus size = 8 bits, Instructions size = 20 bits,
format: 3bits reserve, 3bits opcode, 3bits destination register (%d), 3bits source register (%s), 8bits immediate or address
%s corresponds to Raddr1 in the regs.sv file
%d corresponds to Raddr2 in the regs.sv file

e.g. 18d60 // 20'b 000110_001_101_01100000 // MUL %1, %5, 0.75;

Your affine transformation program

```
// HEX ////////// BINARY  ////////////////////// ASSEMBLER /////////
02000 // 000000 100 000 00000000 // LD     %4, %4, %0;
08600 // 000010 000 110 00000000 // BEQ  %0, %6, -1;
18d60 // 000110 001 101 01100000 // MUL  %1, %5, 0.75;
195c0 // 000110 010 101 11000000 // MUL  %2, %5, -0.5;
04600 // 000001 000 110 00000000 // BNE  %0, %6, -1;
08600 // 000010 000 110 00000000 // BEQ  %0, %6, -1;
19d40 // 000110 011 101 01000000 // MUL  %3, %5, 0.5;
10b00 // 000100 001 011 00000000 // ADD  %1, %1, %3;
19d60 // 000110 011 101 01100000 // MUL  %3, %5, 0.75;
04600 // 000001 000 110 00000000 // BNE  %0, %6, -1;
16114 // 000101 100 001 00010100 // ADDI %4, %1, 20;
11300 // 000100 010 011 00000000 // ADD  %2, %2, %3;
08600 // 000010 000 110 00000000 // BEQ  %0, %6, -1;
162ec // 000101 100 010 11101100 // ADDI %4, %2, -20;
04600 // 000001 000 110 00000000 // BNE  %0, %6, -1;
```

Design Block Diagram



2. Overall architecture of the design and simulations

2.1 Affine transformation

The general affine transformation algorithm can be represented as multiplication and addition of matrix:

$$\begin{bmatrix} X_2 \\ Y_2 \end{bmatrix} = A \times \begin{bmatrix} X_1 \\ Y_1 \end{bmatrix} + B$$

[X1, Y1] and [X2, Y2] are 8-bit signed binary numbers which represent input and output pixel coordinates respectively. A and B are transformation coefficients, and their value is set in this design:

$$A = \begin{bmatrix} 0.75 & 0.5 \\ -0.5 & 0.75 \end{bmatrix} \quad B = \begin{bmatrix} 20 \\ -20 \end{bmatrix}$$

A and B are converted to 8-bit signed binary numbers and stored in the program memory as immediate values. Based on the above mathematical equations, we can get the following equation:

$$X_2 = A_{11} \times X_1 + A_{12} \times Y_1 + B_1$$
$$Y_2 = A_{21} \times X_1 + A_{22} \times Y_1 + B_2$$

2.2 ALU design

In this design, the affine transformation algorithm only consists of addition and multiplication. Also, since this design needs to make use of two judgement instructions, BEQ and BNE, the ALU needs to perform subtraction calculations. The testbench and validation results for ALU module are shown is Figure 3.

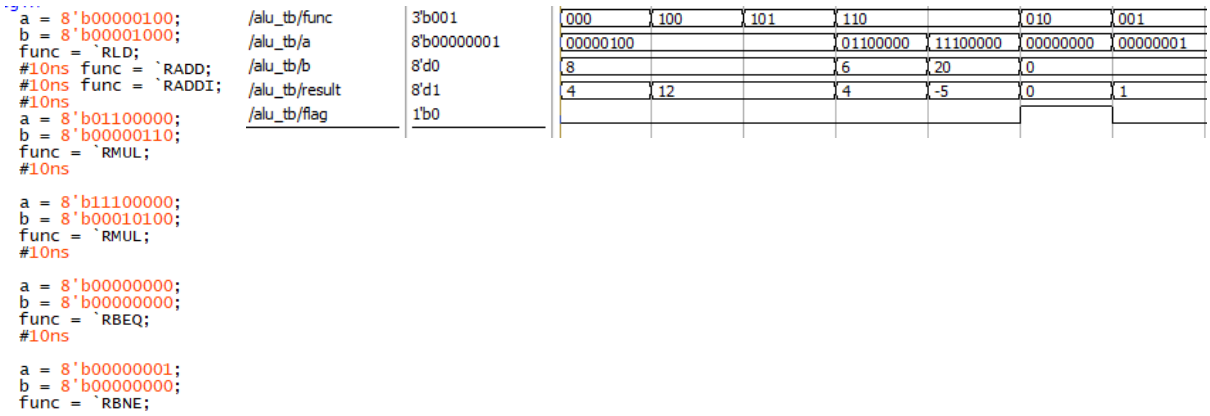


Figure 3. validation for ALU module

The simulation tested that when the inputs are decimal 4 and 8, the corresponding addition result is 12; when the inputs are 0.75 and 6, the corresponding multiplication result is 4 (rounding off the decimal part); and when the inputs are -0.25 and 20, the corresponding multiplication result is -5. Up to this point, when the input result is either two positive integers, positive fractions, or negative numbers, the calculation result is correct.

In addition to this, the instructions BEQ and BNE are tested to see if they are correct when judging SW8. From Rdata1={7'b0000000, SW[8]} in the regs.sv file, it follows that Rdata1 should be 0, i.e., a is 0, when SW8=0, and Rdata2 is %0, i.e., b is always the immediate number 0 to judge the value of SW8. At this time, ALU performs subtraction calculation. b1=~b+1'b1=0, ar=a+b1, so result is 0,flag is 1. Otherwise, when SW8=1, ar=a+b1=1+0=1. So result is 1,flag is 0.

2.3 Program counter design

The PC module has inputs of clock, reset, PCincr, PCrelbranch, and branchaddress, and has an output of PCout. The value for PC size is 4 bits which can carry up to 16 instructions, which is enough for this design. The Figure 4 shows the validation result of PC module. The PC operation is by incrementing its value each clock cycle to be able to move to next instruction of the code. If PCrelbranch is set, the relative branch will be added to the current PC value. For example, if Branchaddr is set to zero, PCout will maintain the previous value, which is shown in the Figure 4.

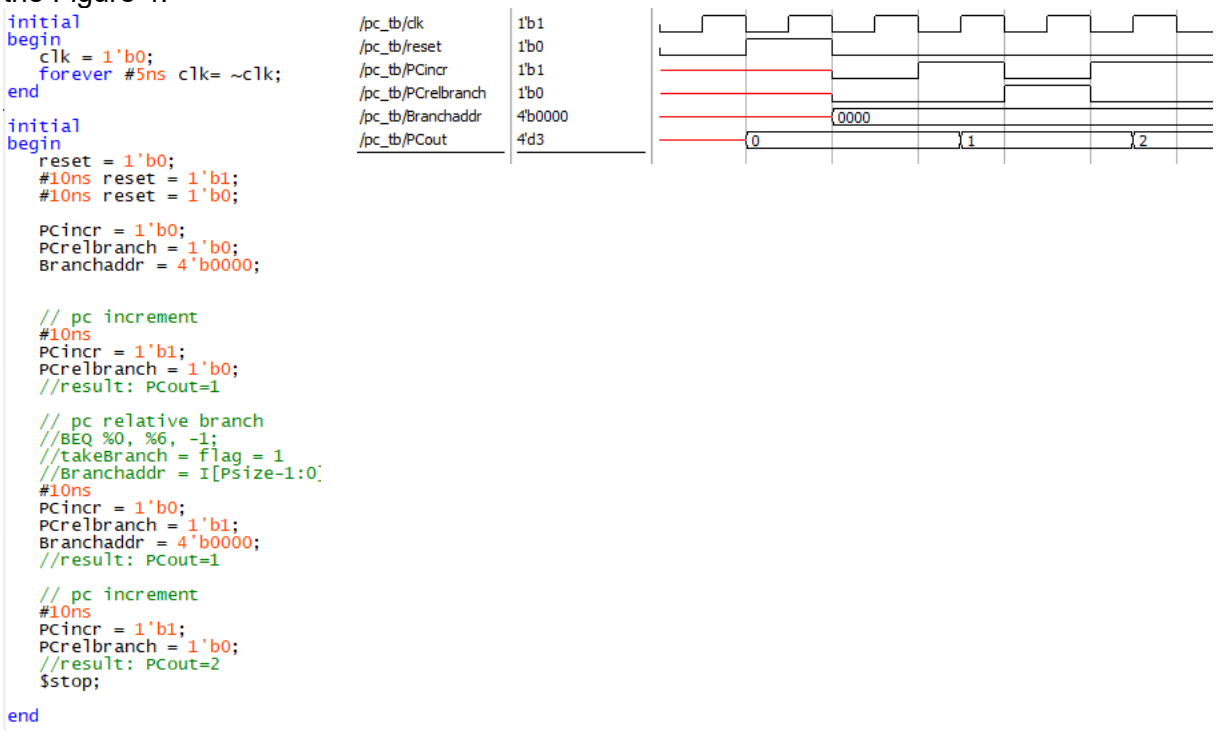


Figure 4. validation for PC module

2.4 Regs module design

For the Regs module, it takes input of clock, w, Raddr1, Raddr2, SW. As Figure 5 shown, Reg %0 is reserved for zero value; Reg %5 is reserved for input data of the switch0-7; Reg %6 is reserved for the handshaking switch8. The remaining registers are applied to perform internal calculations %1-%4. So in reality we only need 4 gpr: gpr[0] →%1, gpr[1] →%2, gpr[2] →%3, gpr[3] →%4.

```
// Declare 32 n-bit registers
logic [n-1:0] gpr [3:0];
assign out = gpr [3];   //%4 for LED display


// write process, dest_reg is Raddr2
always_ff @ (posedge clk)
begin
   if (w)
         gpr[Raddr2-1] <= Wdata;

end
```

```
// read process, output 0 if %0 is selected
always_comb
begin
   if (Raddr1 == 3'b000)            //%0
         Rdata1 = {n{1'b0}};
   else if (Raddr1 == 3'b101)       //%5
         Rdata1 = SW[7:0];
   else if (Raddr1 == 3'b110)       //%6
         Rdata1 = {7'b0000000, SW[8]};
   else  Rdata1 = gpr[Raddr1-1];    //%1-%4

   if (Raddr2 == 3'b000)            //%0
         Rdata2 = {n{1'b0}};
//   else if (Raddr2 == 3'b101)       //%5
//         Rdata2 = SW[7:0];
   else  Rdata2 = gpr[Raddr2-1];    //%1-%4
end
```

Figure 5. Regs module

As Figure 6 shown, I simulated three types of instructions that occur during programme execution. Firstly, if the BEQ instruction executes, source register is %6, destination register is %0, so Rdata1 should be 7{1'b0}+SW8. Secondly, if the MUL instruction executes, assuming that source register is %5, destination register is %1, gpr[0] should be same with Wdata, and Rdata2 will be connected to gpr[0], because %1→gpr[0]. Thirdly, if the ADDI instruction executes, it should be the display LED state, so destination register is %4, and it results in: Rdata2=gpr[3]→%4.

```
initial
begin
   w = 1'b0;
   wdata = 7'b0;

   //BEQ %0, %6, -1;
   SW = 9'b100000100;    //SW8 = 1, x1 = 4
   Raddr1 = 3'b110;      //%d = %0, %s = %6
   Raddr2 = 3'b000;
   //result: Rdata1=8'b00000001, Rdata2=n{1'b0}

   //MUL %1, %5, 0.75
   #30ns
   w = 1'b1;
   wdata = 8'b00000011; //4 * 0.75 = 3
   Raddr1 = 3'b101;      //%s = %5
   Raddr2 = 3'b001;      //%d = %1
   //result: Rdata1=8'b00000100, Rdata2=gpr[0]
   //        gpr[0]=8'b00000011

   //ADDI %4, %1, 20
   #30ns
   w = 1'b1;
   wdata = 8'b00010111; //3 + 20 = 23
   Raddr1 = 3'b001;      //%s = %1
   Raddr2 = 3'b100;      //%d = %4
   //result: Rdata1=gpr[0]=8'b00000011
   //        Rdata2=gpr[3]
   //        gpr[3]=8'b00010111 --> out
end
```

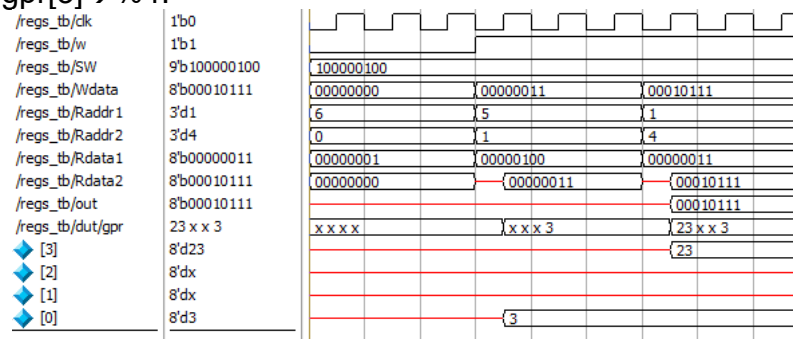
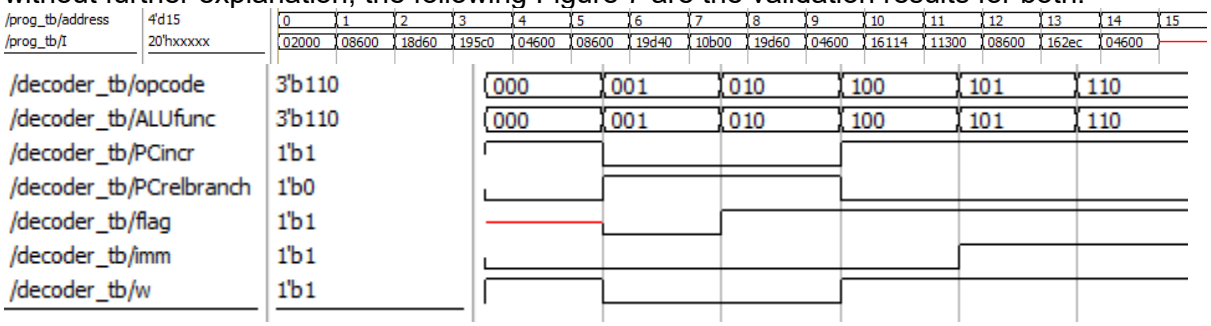


Figure 6. validation for regs module

2.4 other modules design

The design of the program memory module and the decoder module is relatively simple, so without further explanation, the following Figure 7 are the validation results for both.



```
initial
begin
        address = 4'b0000;    //PCout++
    #10ns address = 4'b0001;
    #10ns address = 4'b0010;
    #10ns address = 4'b0011;
    #10ns address = 4'b0100;
    #10ns address = 4'b0101;
    #10ns address = 4'b0110;
    #10ns address = 4'b0111;
    #10ns address = 4'b1000;
    #10ns address = 4'b1001;
    #10ns address = 4'b1010;
    #10ns address = 4'b1011;
    #10ns address = 4'b1100;
    #10ns address = 4'b1101;
    #10ns address = 4'b1110;
    #10ns address = 4'b1111;
    $stop;
end
```

```
initial
begin
      opcode=3'b000; //LD

#10ns opcode=3'b001; //BNE
      flag = 1'b0;

#10ns opcode=3'b010; //BEQ
      flag = 1'b1;

#10ns opcode=3'b100; //ADD

#10ns opcode=3'b101; //ADDI

#10ns opcode=3'b110; //MUL

end
```

Figure 7. validation for prog and decoder module

3. FPGA implementation

During the FPGA implementation, firstly I assigned the pins for this design as per the relevant documents of DE1-SOC as shown in Figure 8.Secondly during the first implementation, I tested the input X1=4, Y1=8 and came up with the result of X2=27, Y2=-8.This is a deviation from the actual calculation of X2=27,Y2=-16. Careful testing of the alu module revealed that it was a miscalculation when the inputs were -0.5 and 4 for multiplication. The result of the calculation should be -2, however the initial alu module calculated 6.Analysing the multiplier module written, it was found that when the input is a negative number, all the positions before the sign bit should be placed at the same value, for example, if the sign bit of a negative number is 1, then when multiplying the numbers, the higher positions are all 1 to ensure that the final result is calculated correctly in terms of both positive and negative. Finally, I modified the multiplier in the alu module, and as Figure11 shown, the result is correct.

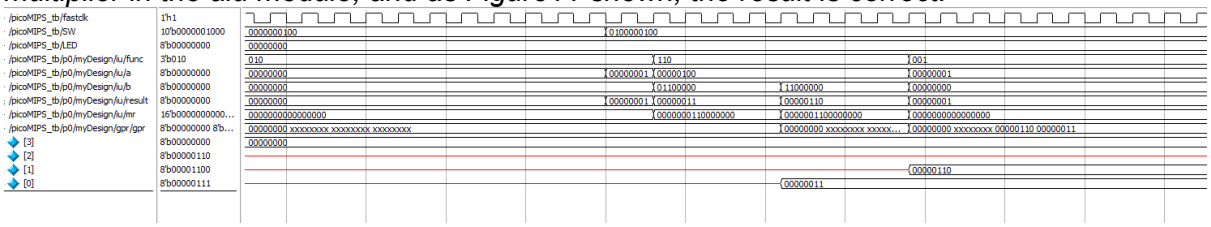


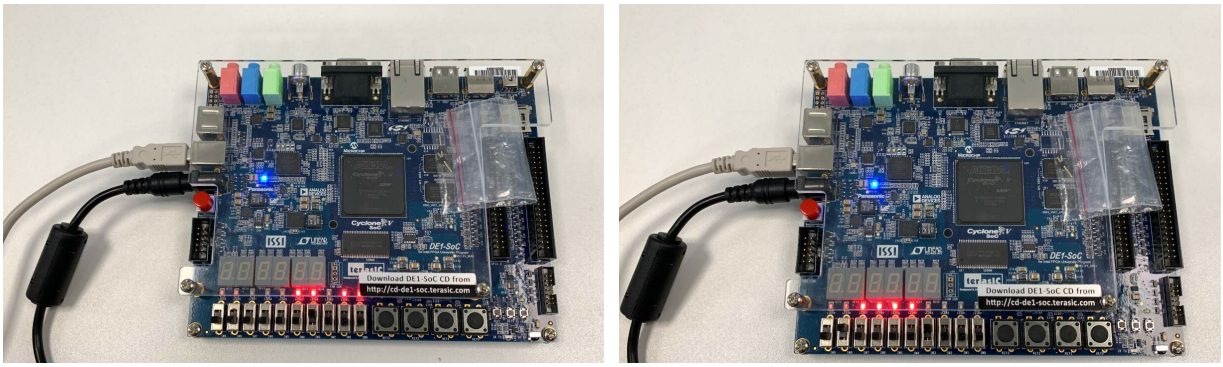
Figure 8. wrong results

To, Location
SW[0], PIN_AB12
SW[1], PIN_AC12
SW[2], PIN_AF9
SW[3], PIN_AF10
SW[4], PIN_AD11
SW[5], PIN_AD12
SW[6], PIN_AE11
SW[7], PIN_AC9
SW[8], PIN_AD10
SW[9], PIN_AE12
fastclk, PIN_AF14
LED[0], PIN_V16
LED[1], PIN_W16
LED[2], PIN_V17
LED[3], PIN_V18
LED[4], PIN_W17
LED[5], PIN_W19
LED[6], PIN_Y19
LED[7], PIN_W20

Figure 9. pin assignment file

```
//multiplier
always_comb
   begin
      mr = {{n{a[7]}}, a[7:0]} * {{n{b[7]}}, b[7:0]};
   end
```

Figure 10. Modified multiplier module in alu



X1: 00000100→ 4
X2: 00011011→ 27

Y1: 00001000→ 8
Y2: 11110000→ -16

Figure 11. final results

4. Conclusion

In this project, all the functional requirements and objectives have been fully achieved. A smallest picoMIPS architecture processor for the affine transformation of graphic pixels was successfully designed, and a machine-level program for the general affine transformation was developed at the same time. The processor is implemented and verified on an Altera FPGA development board with minimal logic resources.

To improve the design, we could optimize the design further by reducing the number of instructions needed and using more efficient algorithms for the operations. For example, designing a special instruction to replace the functions of both the BNE and BEQ instructions, since essentially they perform the same function. Or we could drastically change the idea of the design and not follow the step-by-step waiting for SW8 to perform the handshake protocol. Instead, a special instruction is designed so that when SW8 has no feedback, it waits in a constant loop; when SW8 is triggered, the appropriate instruction is executed. Overall, this project provided a valuable learning experience in embedded processor design and implementation.

5. References

[1] Dr Tom Kazmierski, "ELEC6234 Embedded Processor: Notes," University of Southampton [Online]. Available: https://secure.ecs.soton.ac.uk/notes/elec6234/