ELEC6234 – FPGA Synthesis of picoMIPS

Xing Jiang
xj1e21
Msc Electronic Engineering

ABSTRACT: *Design of a small picoMIPS architecture with a set of machine code for the affine transformation, including ALU, Register, Program Counter, Program Memory, Decoder and 2 Multiplexers. The design was proven to be able to run the algorithm and obtain correct results through ModelSim and Quartus simulations. And the processor is running properly on Altera DE0 development board.*

1.Introduction

The main design objective of this project is to design an 8-bit picoMIPS processor and run an affine transformation algorithm on the processor.

The algorithm can be expressed by the following equation:

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = A \times \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} + B$$

x_1 and y_1 represent the coordinates of a pixel before the transformation. And (x_2, y_2) represent the coordinates of a pixel after the transformation. A and B is the coefficient matrixes which seem like:

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad \mathrm{B} = \begin{bmatrix} \mathrm{b}_1 \\ \mathrm{b}_2 \end{bmatrix}$$

And the formula can be like:

$$\begin{cases} \mathrm{x}_2 = \mathrm{a}_{11} \cdot \mathrm{x}_1 + \mathrm{a}_{12} \cdot \mathrm{y}_1 + \mathrm{b}_1 \\ \mathrm{y}_2 = \mathrm{a}_{21} \cdot \mathrm{x}_1 + \mathrm{a}_{22} \cdot \mathrm{y}_1 + \mathrm{b}_2 \end{cases}$$

When the true coeffcients are into the formula ,it can be expressed like:

$$\begin{cases} \mathrm{x}_2 = 0.75 \cdot \mathrm{x}_1 + 0.5 \cdot \mathrm{y}_1 + 20 \\ \mathrm{y}_2 = -0.5 \cdot \mathrm{x}_1 + 0.75 \cdot \mathrm{y}_1 - 20 \end{cases}$$

According to the formula, the processor needs a multiplier and an adder and four 8-bit registers to complete all operations, so that the number of bits in the instruction set can be determined as 15 bits, where the first three specify the type of instruction, the next two specify the target register address and the sourse register address, and the next eight bits are used to store immediate numbers or other instructions.

2. Overall architecture of the design

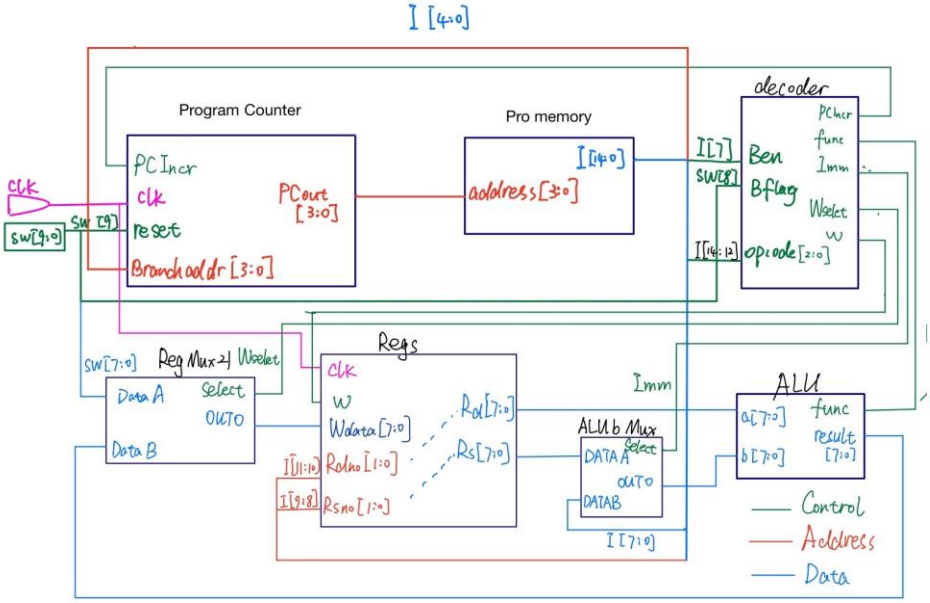

Fig1. The Architecture of my picoMIPS

2.1 Instruction set

I have designed 5 operation codes, which have their own functions as below:

Instruction	Bit 14-12	Bit 11-10	Bit 9-8	Bit 7-0 Data or Adress	Function
INPUT	3'b000	Rdno	Rsno	No means	Input num to %d
ADD	3'b001	Rdno	Rsno	No means	%d=%d+%s
ADDI	3'b010	Rdno	Rsno	8-bit operand	%d=%s+operand
MULI	3'b011	Rdno	Rsno	8-bit operand	%d= %s*operand
HOLD	3'b100	Rdno	Rsno	a, 2'b00,5-bits Porg address	If (a == SW[8]) Goto the address

Table1. Instruction set

From the table above, we can see that ADD, ADDI and MULI are used to calculate and INPUT is used to store the input numbers(x_1 and y_1), and HOLD is used to stay at the same instruction or go to the top of loop in order to input the number by user or restart the program to calculate the new pixel numbers.
Meanwhile, HOLD can set as 0 or 1 so that user can use different states of SW[8] to control the program process.

2.2 Decoder design

```
`include "decoder.sv"
module test_decoder;
logic [2:0] opcode ;
logic Switch8, HOLDen;
logic PCincr, func, writeSelect, imm, w;

decoder decoder1 (.opcode(opcode),
                  .Switch8(Switch8),
                  .HOLDen(HOLDen),
                  .PCincr(PCincr),
                  .func(func),
                  .writeSelect(writeSelect),
                  .imm(imm),
                  .w(w));

initial
    begin
        HOLDen = 0; Switch8 = 0;
        #10ns opcode = 3'b000; //INPUT
        #10ns opcode = 3'b001; //ADD
        #10ns opcode = 3'b010; //ADDI
        #10ns opcode = 3'b011; //MULI
        #10ns opcode = 3'b100; //HOLD
        #5ns HOLDen = 1; Switch8 = 0;
        #5ns HOLDen = 0; Switch8 = 1;
        #5ns HOLDen = 1; Switch8 = 1;
    end//initial

endmodule//module test_decoder
```

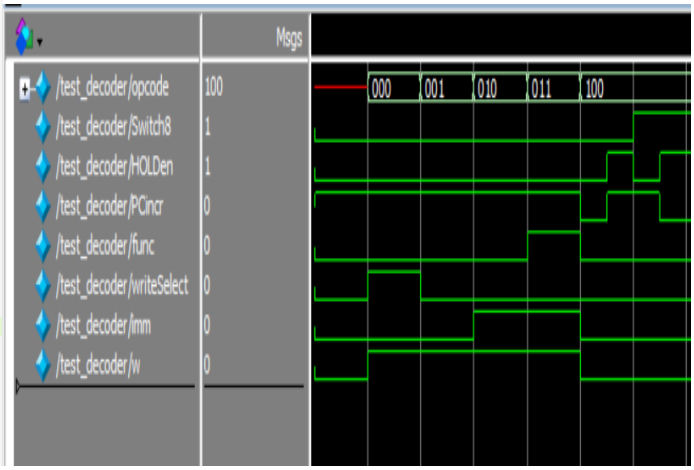

Fig2. Decoder testbench and waveform

From the Model Sim simulation result we can see that every Instrunction has the right output result, like INPUT can make writeSelect and w enable, so the input can store into register. And HOLD can judge the HOLDen and Switch8 as the same or not , and PCincr will change if these two value is the same. Above all Decoder can read the opcodes and put the output value to the right state very well.

2.3 Program Memory design

```
// HEX ////////// BINARY  ////////////////////// ASSEMBLER /////////
3000      //15'b011_00_00_00000000 MULI %0,%0,0; %0 = %0 * 0// clear %0
4001      //15'b100_00_00_00000001 HOLD 0 PC = 1
0000      //15'b000_00_00_00000000 INPUT %0; Store x1 to %0 register
4083      //15'b100_00_00_10000011 HOLD 1 PC = 3
4004      //15'b100_00_00_00000100 HOLD 0 PC = 4
0400      //15'b000_01_00_00000000 INPUT %1; Store y1 to %1 register
4086      //15'b100_00_00_10000110 HOLD 1 PC = 6
3860      //15'b011_10_00_01100000 MULI %2,%0,0.75; %2 = %0 * 0.75// 0.75x1
3D40      //15'b011_11_01_01000000 MULI %3,%1,0.5; %3 = %1 * 0.5// 0.5y1
1B00      //15'b001_10_11_00000000 ADD  %2,%3; %2 = %2 + %3// 0.75x1 +0.5y1
2814      //15'b010_10_00_00010100 ADDI %2,20; %2 = %2 + 20// 0.75x1 +0.5y1 + 20 = x2 display
400B      //15'b100_00_00_00001011 HOLD 0 PC = 11
38C0      //15'b011_10_00_11000000 MULI %2,%0,-0.5; %2 = %0 * -0.5// -0.5x1
3D60      //15'b011_11_01_01100000 MULI %3,%1,0.75; %3 = %1 * 0.75// 0.75y1
1B00      //15'b001_10_11_00000000 ADD  %2,%3; %2 = %2 + %3// -0.5x1 +0.75y1
28EC      //15'b010_10_00_11101100 ADDI %2,-20; %2 = %2 - 20// -0.5x1 +0.75y1 - 20 = y2 display
4090      //15'b100_00_00_10010000 HOLD 1 PC = 16
4000      //15'b100_00_00_00000000 HOLD 0 PC = 0
```

Fig 3. Machine code of the whole program.

The Machine code set is stored at Program Memory and it is the most important part of the picoMIPS. I use only 18 instructions to complete the program and I use the HOLD instruction to set stop points for handshake(SW[8] == I[7]) and other use.

During the calculation process, I use all 4 register to store the input numbers and Intermediate variables, %0 and %1 store two input numbers and %2, %3 store the Intermediate variables. LED read the ALU result directly so I can save a register from it.

2.4 Program counter design

```
// up to 32 instructions
module pc #(parameter Psize = 5) (input logic clk, reset, PCup,
                                  input logic [Psize-1:0] BranchAddress,
                                  output logic [Psize-1 : 0]PCout);

//------------- code starts here---------
    always_ff @ ( posedge clk or negedge reset) // async reset
        begin
            if (!reset) // sync reset
                PCout <= {Psize{1'b0}};

            else if (PCup) // increment
                PCout <= PCout + 1'b1;

            else if (!PCup) // absolute branch
                PCout <= BranchAddress;
        end

endmodule // module pc
```

```
`include "pc.sv"
module test_pc #(parameter Psize = 5);
logic clk ;
logic reset, PCup;
logic [Psize-1:0] BranchAddress;
logic [Psize-1:0] PCout;

pc #(.Psize(5)) pc1 (.clk(clk),
                     .reset(reset),
                     .PCup(PCup),
                     .BranchAddress(BranchAddress),
                     .PCout(PCout));
initial//clock period = 10ns
    begin
        clk = 1'b0;
        forever
            begin
                #5ns clk = ~clk;
            end
    end
initial
    begin
        reset = 0; PCup = 0; BranchAddress = 5'b0;
        #10ns reset = 0; PCup = 1;
        #10ns reset = 1;
        #30ns PCup = 0; BranchAddress = 5'b01100;
        #30ns reset = 1; PCup = 1;
        #30ns reset = 0;
        #10ns reset = 1;
    end//initial

endmodule//module test_pc
```

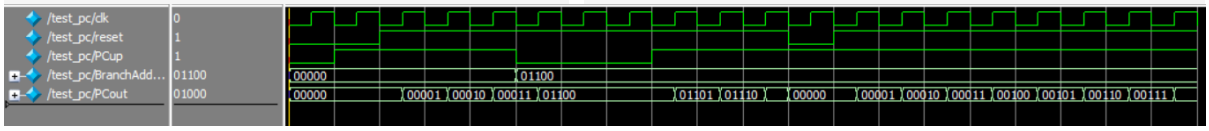

Fig 4. PC code and Model Sim waveform

As the picture can be seen, Program counter is controlled by clock and PCup. When the clock and PCup all are positive edge and 1'b1, PCout will add one so the instruction of the next address in Program Memory will be selected and decoded. While the PCup changes to 1'b0, the PCout will become the BranchAddress, and it is the way of HOLD to run its function.

2.5 Register and its MUX Design

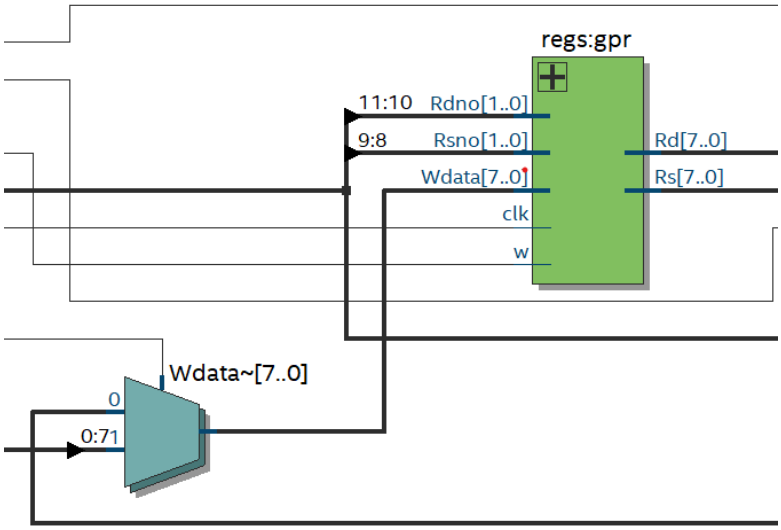

Fig 5. Register and its MUX Synthesis

I use a MUX for register to select data from different origins to write, when the writeSelect turns to 1'b0, MUX use the data from ALU to write into the register, or the SW[7:0](input numbers).
For register, I use the two input address (Rdno and Rsno) to supply the stored date to be read by ALU, and all date to be written are stored in Rdno Register.

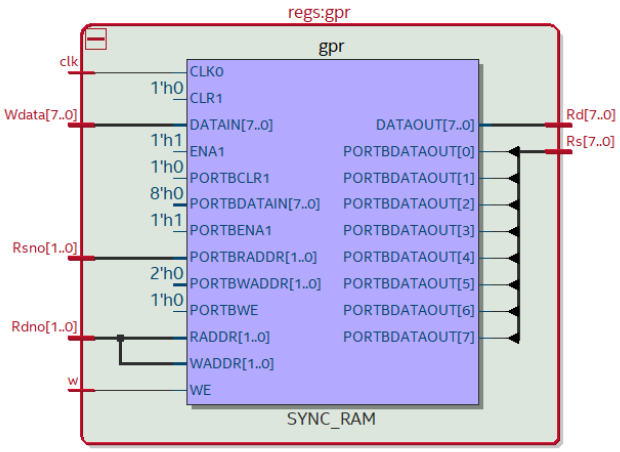

Fig 6. Register Synthesis

2.6 ALU and its MUX design

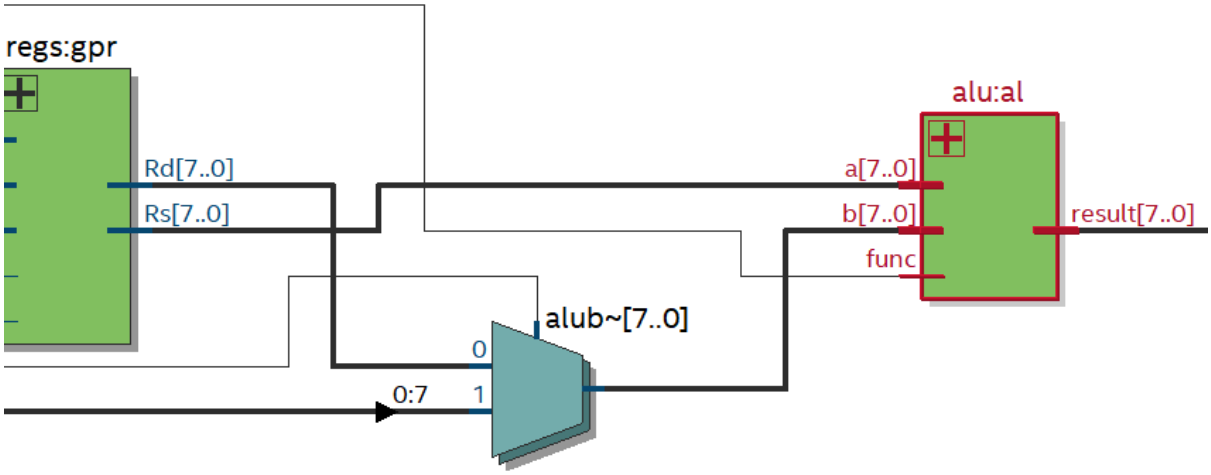

Fig 7. ALU and its MUX Synthesis

ALU has two input variable: a and b. And a is linked to Rs, because all calculation is a number stored in register with another number from register or immediate number(I[7:0]), so b is linked with a MUX, and the MUX can select data from register(Rd) or Machine code(I[7:0]).
ALU only has one output: result, which can be sent to LED or Register MUX.

3. FPGA implementation

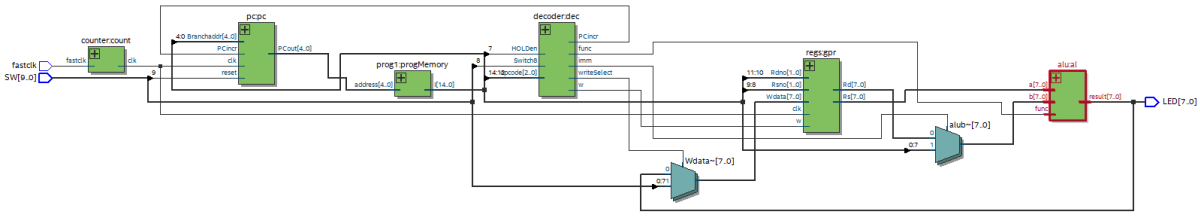

Fig 8. picoMIPS Synthesis

As we can see, the result of Quartus Synthesis is as the same as my design of picoMIPS architecture(Fig 1).And in order to run this project in the FPGA, I collect the counter output sign in to picoMIPS.sv, so the clock can enable the PC and Register.

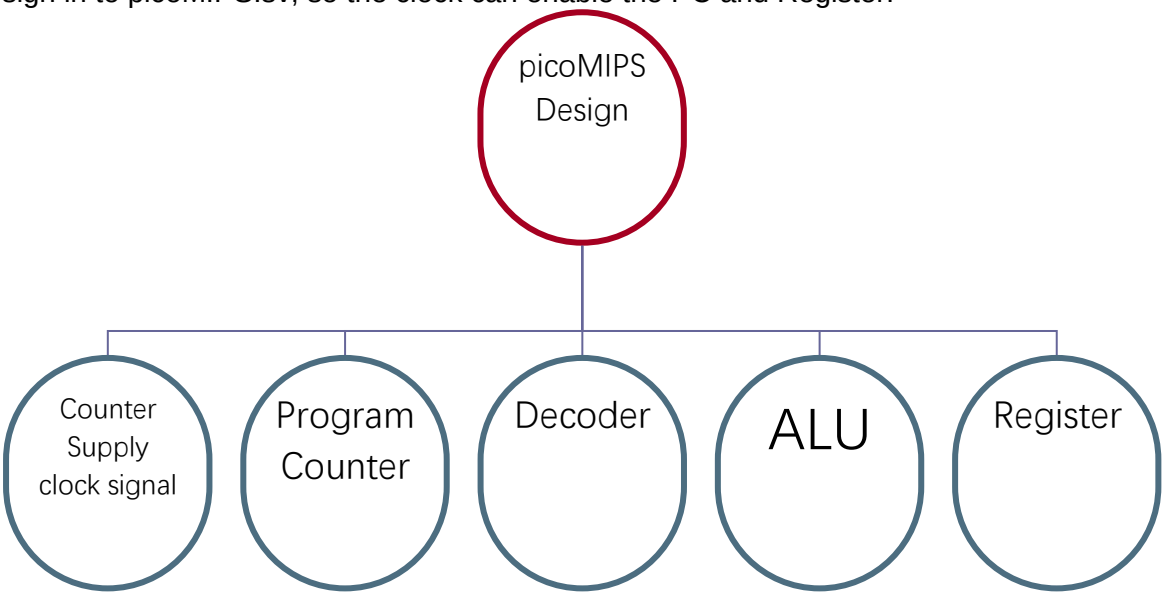

Fig 9. picoMIPS Test Architecture

I tried to input X1 = 40, Y1 = 40, Finally, the LED output the X2 =80 and Y2 = -10.
And the exact process will be shown in a separate verification video. But also I found some problems that because ALU result only show the integer part and it can make some different from the calculation results by hand.

4. Conclusion

Flow Status	Successful - Thu May 05 12:19:41 2022
Quartus Prime Version	16.1.0 Build 196 10/24/2016 SJ Lite Edition
Revision Name	picoMIPS
Top-level Entity Name	picoMIPS
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	126
Total combinational functions	102
Dedicated logic registers	37
Total registers	37
Total pins	19
Total virtual pins	0
Total memory bits	0
Embedded Multiplier 9-bit elements	1
Total PLLs	0

Fig 10. The synthesis report of picoMIPS

Overall, the picoMIPS I designed is able to perform all the tasks specified, each module runs

smoothly either individually or in combination, and the processor is able to run the affine transformation algorithm successfully and continuously without the need for power failure and restart.

According to the synthesis report above, it can be found that the cost of this design is as low as 126, and it only takes less than 1% of the FPGA computing resources, but there are many areas that can be improved, for example, for the instruction type, INPUT can be integrated into the ADD command, and at the same time, the register can only read the result of ALU. In addition, the design of the ALU can be further optimized, such as improving the multiplication operation for the affine transform algorithm by using the inverse operation instead of the current direct multiplication.
The successful completion of this project has helped me to be familiar with and master the design and debugging of complex projects, especially the analysis of program bugs using the waveform of ModelSim, which requires concentration and patience, as well as a deep understanding of the designed program.

5. Reference

[1] Dr Tom J Kazmierski, “ELEC6234 Embedded Processor Synthesis: Notes,” University of Southampton, [Online]. Available: https://secure.ecs.soton.ac.uk/notes/elec6234/ [Accessed 25/03/2017]
[2]Y. Tang, "GitHub - tangyeqiu/picoMIPS: Individual project in the University of Southampton.", GitHub, 2022. [Online]. Available: https://github.com/tangyeqiu/picoMIPS. [Accessed: 05- May- 2022].