ELE6234 - Embedded Processors
Coursework Report – picoMIPS implementation

Longhao Zhu
lz6y23
MSc Internet of Things

I. INTRODUCTION

The main task of the assignment is to design an n-bit CPU with minimal resources for calculating the affine transformation algorithm based on the picoMIPS architecture, and it can run successfully on the FPGA.

In this task, I used the systemVerilog language to implement an 8-bit CPU of the picoMIPS architecture and successfully executed the affine transformation algorithm. At the same time, I ran the code on the FPGA development board. The following is the formula(1) for the affine transformation. Enter the parameter X1 and Y1 through the switch, complete the operation and output X2, Y2 to LED.

$$\begin{bmatrix} X2 \\ Y2 \end{bmatrix} = A * \begin{bmatrix} X1 \\ Y1 \end{bmatrix} + B \tag{1}$$

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

In this assignment, the calculation of affine transformation is completed by implementing the basic components of picoMIPS: program counter, program memory, decoder, ALU, register group, and designing NOP, ADD, ADDI, MULI, and LOAD instructions for encoding. The number of instructions is 15 bits: three-bit operands, two two-bit register addresses, and 8-bit immediate data. I input the X1 and Y1 inputs of switches 0-7 directly into the ALU through multiplexing and store them in registers, while switch 8 controls whether the program counter pauses in this line to wait for data input. I implemented the blocking function by modifying the PC component. I input the value of switch 8 directly into the PC and detected the falling edge of switch 8. When the switch is turned down, the program can continue to execute, but when the LOAD instruction is reached, the program will be suspended. In this way, blocking and input only require one instruction to save resources. For output, I directly read the values of registers 2 and 3, and used switch 8 to determine whether to output X2 or Y2 to the LED.

In order to easily observe the output results, I designed a module called number_display to display the output signal in a 7-bit digital tube. Because the output has 8 bits and the range is between -128–127, four digital tubes are needed for display. I perform a remainder operation on the output result to extract the ones, tens, hundreds digits and the highest sign bit, and display them on the corresponding digital tubes respectively.

Design Details Form			
Total Cost: 49	**ALMs: 49**	**Memory bits: 0**	**Multipliers: 1**
Instruction Set			
I implemented 5 instructions to ensure that the calculation can proceed smoothly			
Details:			
NOP	The NOP instruction is the default behavior of doing nothing. The default behavior of NOP is to block program execution, not write to registers, and not perform calculations.		
ADD	Add the two values extracted from the register, set the register to a writable state, and store the added result in register %1.		
ADDI	Read the data from the source register, add it to the immediate value, and then store the data in the destination register.		
MULI	Similar to ADDI, the value is read from the source register, multiplied by the immediate value and the result is stored in the destination register.		
LOAD	Take a reading from the switch and store the reading in the destination register.		
Instruction Format			

Instruction size : 15 bits, Data size : 8 bits

Format: 3 bits Operands, 2 bits address of destination register (%d), 2 bits address of source register (%s), 8 bits immediate. for example : NOP 000_00_00_00000000

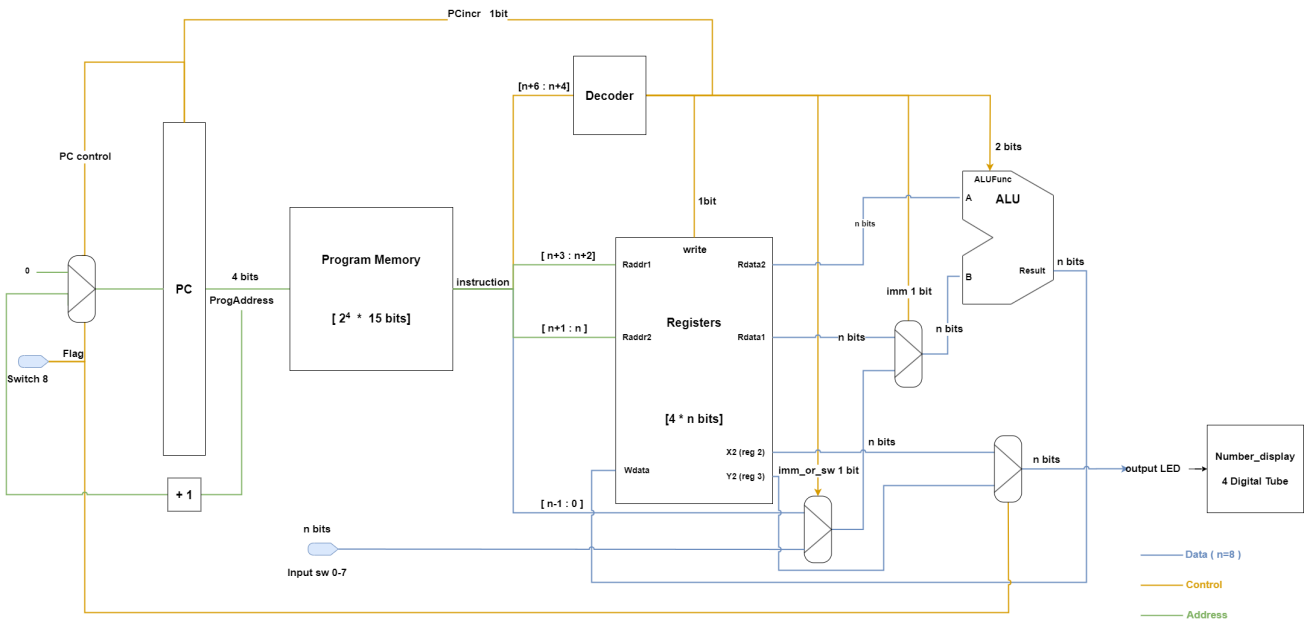
Details:

NOP	No operation	No operation
ADD	ADD %d , %s ;	%d = %d + %s
ADDI	ADDI %d , %s , imm;	%d = %s + immediate value
MULI	MULI %d , %s , imm;	%d = %s * immediate value
LOAD	LOAD %d;	%d = value of switch8

Affine transformation program

1	**0000000000000000**	**LOAD %0 %0 0;**	load X1 value from switch 8 to reg %0
2	**1000100000000000**	**LOAD %1 %0 0;**	load Y1 value from switch 8 to reg %1
3	**1111000011000000**	**MULI %2, %0, 0.75;**	reg %2 = reg %0 * 0.75
4	**1111101010000000**	**MULI %3, %1, 0.5;**	reg %3 = reg %1 * 0.5
5	**0101011000000000**	**ADD %2, %3;**	reg %2 = reg %2 + reg %3
6	**1111100110000000**	**MULI %3, %0, -0.5;**	reg %3 = reg %0 * (- 0.5)
7	**1110001011000000**	**MULI %0, %1, 0.75;**	reg %0 = reg %1 * (0.75)
8	**0101100000000000**	**ADD %3, %0;**	reg %3 = reg %3 + reg %0
9	**1101010000101000**	**ADDI %2, %2, 20;**	reg %2 = reg %2 + 20
10	**1101111111011000**	**ADDI %3, %3, -20;**	reg %3 = reg %3 - 20

Design Block Diagram



II. **Overall architecture of the design and simulations**

In this assignment, it is necessary to design an n-bit picoMIPS architecture CPU to calculate affine transformation. Because the large framework is under picoMIPS, my design basically includes the basic structure of picoMIPS: program counter (PC), program memory, decoder, arithmetic logic unit (ALU), registers. These components are connected to each other, variables are input through switches for calculation, and the results are finally displayed in LEDs.

A. Overall structure

In the overall architecture, according to formula (2), I use an 8-bit data bus width, a 3-bit operation, and 4 registers to implement the calculation of this formula. The overall input includes the clock signal, the reset signal represented by switch9, the input signal represented by switch8, and switch0-7 represent data. We need to toggle switch8 twice and read the values of switch0-7 twice to represent X1 and Y1. The on and off of the switch respectively represent whether the output is X2 or Y2.

$$\begin{aligned} X1 &= A_{11} * X1 + A_{12} * Y1 + B_1 \\ Y2 &= A_{21} * X1 + A_{22} * Y1 + B_2 \end{aligned} \tag{2}$$

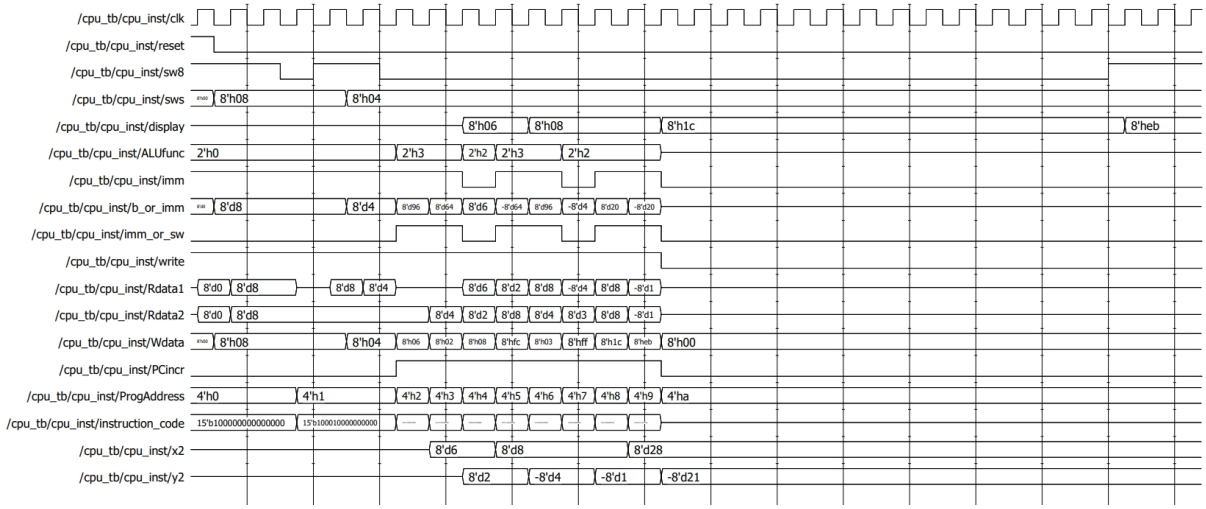


Fig. 1. overall structure wave

According to Figure 1, there are many variables in the overall CPU architecture. We mainly focus on the effects we need to achieve.

- First we focus on the output to see if the calculation is correct. I entered X1 as 8 and Y1 as 4 in the overall CPU testbentch. According to formula 2, I calculated that X2 is 28 and Y2 is -21.In Figure 1, the display first shows 8'h1c and then pulls up sw8 to display it as 8'heb. Judging from the results, the display is correct.At the same time, the two variables X2 and Y2 at the bottom also display 28 and -21.
- Second, we look at the blocking effect. Ideally, when sw8 is at a high level, the program needs to block and wait for the input of numbers. When sw8 is pulled down, the program continues to execute until it encounters LOAD for the second time and blocks waiting for input. In Figure 1, observe the value of PCincr. It is always in the state of 0 in the LOAD instruction, that is, blocked. When the sw8 pull-down program is executed downwards until it is no longer blocked.

The above two points can prove that the CPU I wrote has completed the function from data input to calculation of affine transformation to output display.

B. Program counter

The PC module is a key role in the entire CPU. Its function is to allow the program to continue executing and decide whether to continue executing based on the instructions read out.

My implementation is to set an add signal and sign signal. When PCincr is at 0, flag (sw8) triggers add to 1 on each rising edge. The program adds 1 and changes the sign signal to 1. And I detect the sign rising edge. Each time the sign signal changes to 1, add changes to 0 again, blocking and waiting for the next instruction.

As shown in Figure 2, when the program blocks, the flag rises, causing add to change to 1, the program adds 1, and sign changes to 1. Sign changes to 1, which causes add to change to 0 again until the next time PCincr changes to 1 or flag changes to 1

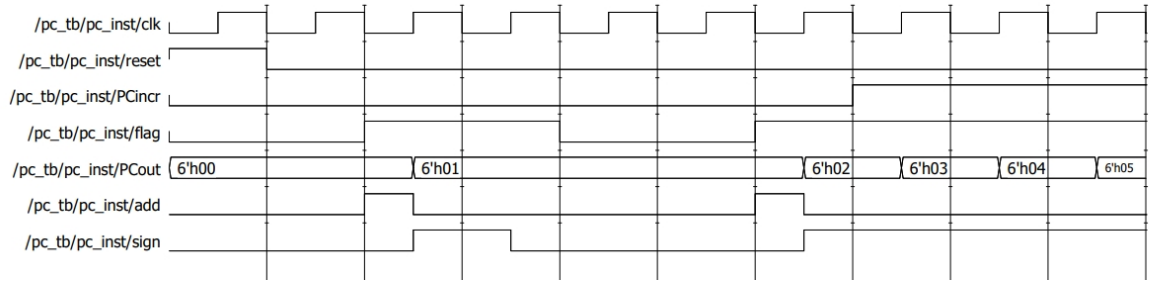


Fig. 2. pc wave

C. Decoder

The main responsibility of the decoder is to parse the 3-bit operands in the instruction and output the imm, imm_or_sw, ALUFunc, write, and PCincr variables. The functions of these variables are to control whether the immediate value or the value of the register is input to the ALU, whether the immediate value represents a number in the instruction or a switch input, the instruction code of the ALU, whether to write to the register, and whether the program blocks.

The waveforms in Figure 3 correspond to the ones in the table.

opcode	opcode	ALUFunc	Pcincr	write	imm	imm_or_sw
000	NOP	00	0	0	0	0
010	ADD	10	1	1	0	0
110	ADDI	10	1	1	1	1
111	MULI	11	1	1	1	1
100	LOAD	00	0	1	1	0

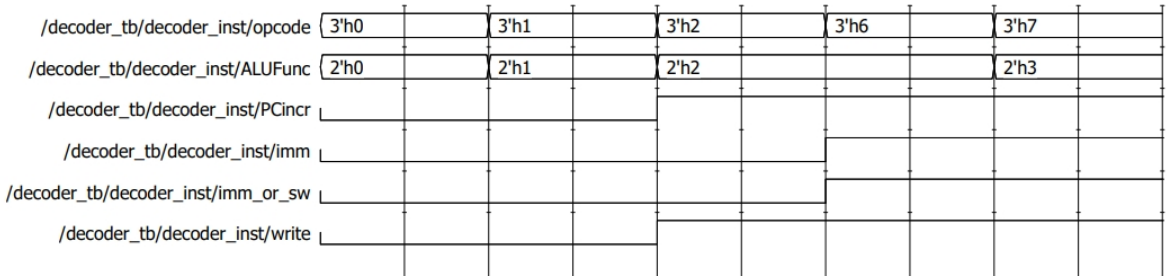


Fig. 3. decoder wave

D. ALU

In my architecture, the most useful ALU is actually the simplest. It only needs to be responsible for addition and multiplication.

It is very intuitive to see in Figure 4 that 3+20 = 23 and 0.57 * 5 = 3. m here is an intermediate value, used to calculate multiplication only takes 14 to 7 bits

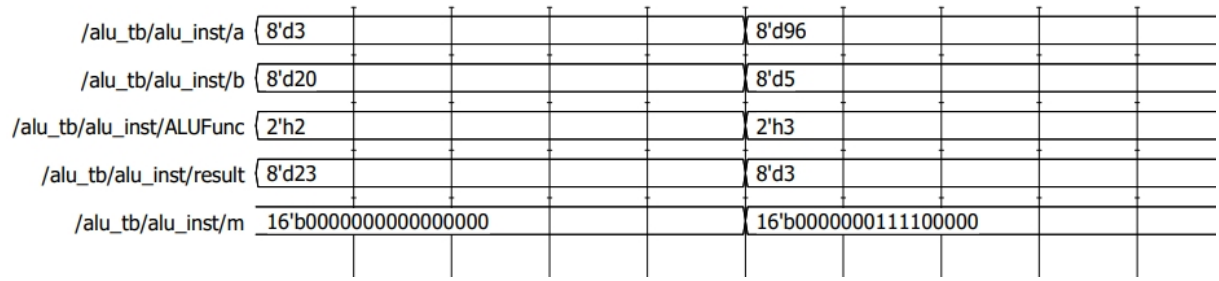


Fig. 4. alu wave

E. Registers

The main function of the register is to store and retrieve data. Storing data depends on the write flag in the decoder. Fetching data depends on the address of the register in the instruction. As shown in Figure 1, the register will also output the values of X2 and Y2. for final presentation.

As can be seen in Figure 5, the data in sws is transferred to wdata and written to regs according to wirte. wdate displays all intermediate process calculation results. Finally, X2 and Y2 are assigned to the display and displayed in the LED.

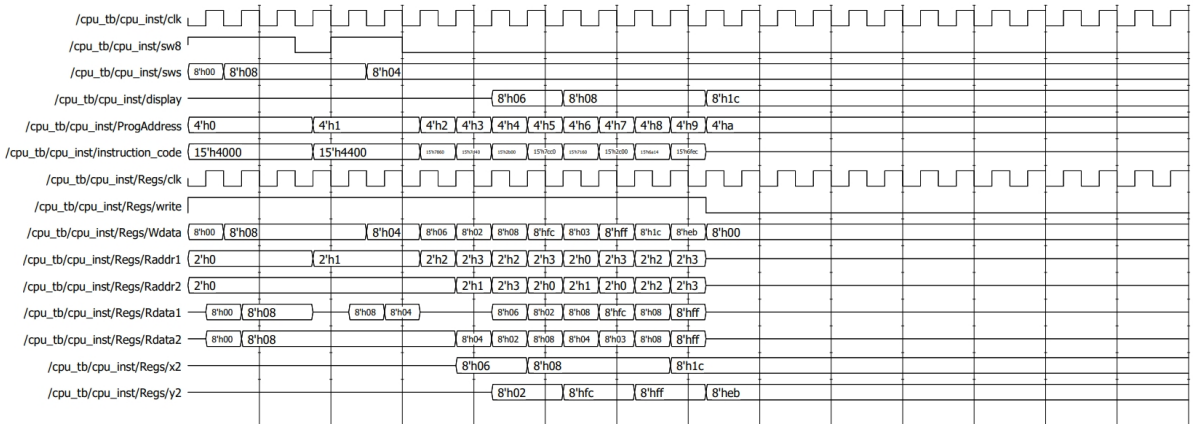


Fig. 5. regs wave

III. FPGA IMPLEMENTATION

The code simulated in modelsim is partially different from the code in FPGA. Among them, we need to add the counter module to reduce the clock frequency to prevent exceptions when switching on and off. In order to run successfully on FPGA, I still need to do a lot of work. First, you need to map all input and output pins to ensure that the FPGA switches, LEDs, and digital tubes work properly. Secondly, find the specific model of the corresponding FPGA, otherwise the code cannot be burned.

In the test FPGA, because the LED displays 8 bits which are very difficult to read, it is impossible to know whether the calculation is correct at the first time. So I added the digital tube display module I wrote myself to the CPU's display output to quickly display the results.Next, select a set of test data for testing.

input data	$X1 : 00001000_2/8_{10}$	$Y1 : 00000100_2/4_{10}$
output data	$X2 : 00001000_2/28_{10}$	$Y2 : 00000100_2/-21_{10}$

The picture below is the output of X2:

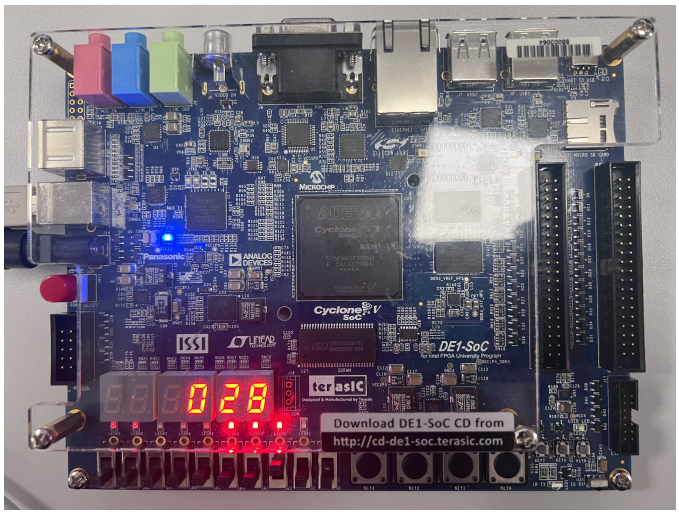


Fig. 6. X2

The picture below is the output of Y2:

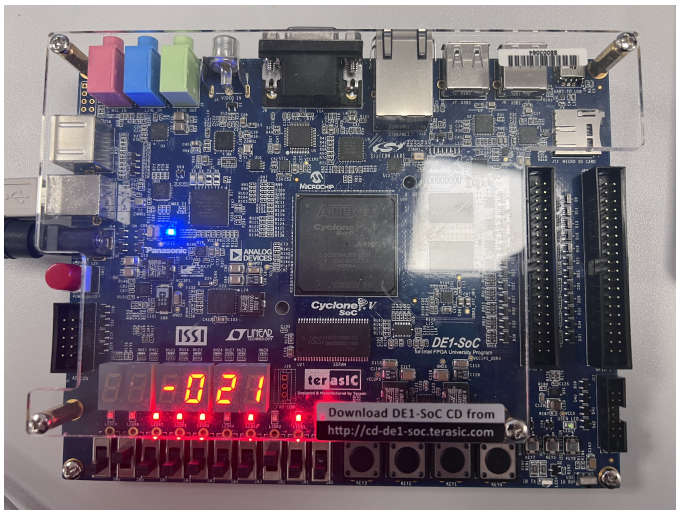


Fig. 7. Y2

IV. Conclusion

In this assignment, I completed the most basic structure of a CPU, implemented the affine transformation algorithm through programming machine code, and successfully ran it on the FPG development board. At the same time, I also learned the systemVerilog language and was able to use it to write hardware code. I also have a deeper understanding of the principles of computer composition. I know the basic structures that the CPU is composed of, and I also understand the process from assembly language to machine instructions, and how machine instructions are executed in hardware.

If I have more time, I would like to understand which of my codes are causing excessive waste of resources. I would like to change the instructions from 3 bits to only 2 bits, because in this assignment my ALU instructions only require 1 bit to determine Is it ADD or MUL. So in theory, my instruction only needs 2 bits, which can save more resources.

The input of this job is through switch, and the output is through LED and digital tube. For subsequent extensions, I want to write systemverilog code to read data from the keyboard and display the results and process on the screen. Second, I want to implement the JUMP command on this basis to loop through multiple inputs and calculate the output.

The following is the result obtained by quatus. There are 47 pins because I added digital tubes. Not included counter and number display.

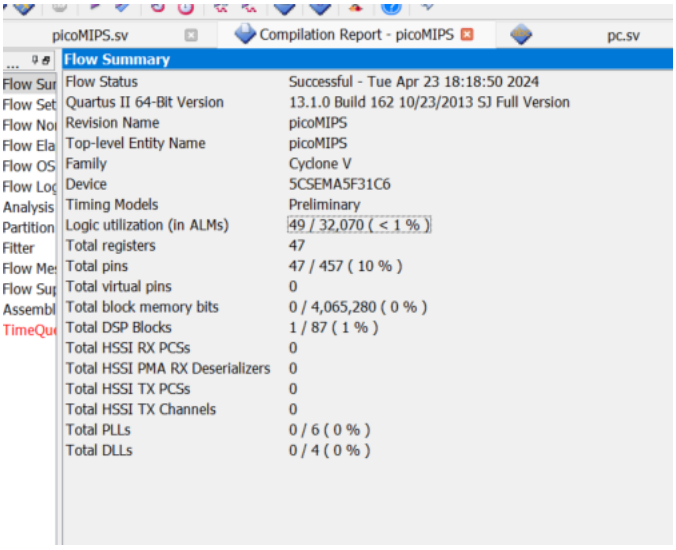


Fig. 8. quartus FPGA cost

V. References

[1] https://github.com/HaosenYu/ELEC6234_Coursework_picoMIPS_processor
[2] https://github.com/lx2u16/ELEC6234_picoMIPS_processor_Coursework