ELEC6234 - SystemVerilog Design of a picoMIPS Embedded Processor

Lei Xun
Lx2u16
MSc System on Chip

ABSTRACT: PicoMIPS is a highly configurable, small and simple Reduced Instruction Set Computer (RISC) softcore microprocessor architecture. Its hardware can be easily tailored to the requirements of a specific application. This report introduces the detailed development of an embedded picoMIPS processor for the affine transformation of graphic pixels. The requirements in the design specification were fully achieved. The processor cost 180 logic elements on an Altera Cyclone IV FPGA.

1. Introduction

1.1 Overview of picoMIPS architecture

PicoMIPS is a Reduced Instruction Set Computer (RISC) softcore microprocessor architecture which inspired by MIPS architecture. It is highly configurable, the width of its data bus, the size of its program memory and the format of its instruction set can be specified by the user. It also has a much smaller size and simpler hardware than many popular RISC processor architecture such as ARM and MIPS. These two features allow the picoMIPS architecture can be tailored to the requirements of a specific application such as the affine transformation of graphic pixels and JPEG decompressor. This methodology allows picoMIPS variable-architecture processors to be high performance, low power consumption and using the minimum resources.

1.2 Affine transformation

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

$$\begin{matrix} X2 \\ Y2 \end{matrix} = A \times \begin{matrix} X1 \\ Y1 \end{matrix} + B$$

[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:

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

A and B are converted to 8-bit signed binary numbers and stored in the program memory as immediate values.

1.3 Objective of the design

The objective of this design is to develop a picoMIPS processor for the affine transformation of graphic pixels that using minimum logic resources on an Altera Cyclone IV FPGA. The size cost function of the design is defined as:

$$cost = number\ of\ logic\ elements\ used + \max(0, number\ of\ embedded\ multipliers\ used - 2) + 30 \times Kbits\ of\ RAM\ used$$

The objective was fully achieved. A machine-level program of the general affine transformation algorithm was developed first, then based on the instructions that used in the algorithm, the architecture of this processor was tailored to reduce the cost figure. The processor was implemented on an Altera FPGA, and the dedicated logic was also designed to control the timing of data input and output. Data was input through switches, and the results were shown on the LEDs on the FPGA board.

The design and verification of this processor are introduced in the following sections of this report: Section 2 introduces instruction format and control path. The data path is introduced in two sections. In section 3 the general purpose register file design is covered. In section 4 the design of Arithmetic Logic Unit (ALU) and the use of an embedded multiplier are discussed. Section 5 introduces how the FPGA is used to synthesis and accelerate the verification of the processor. Finally, section 6 gives the cost figure and suggests further improvement of the design.

2. Instruction format, program counter, program memory and instruction decoder design

The block diagram of the purposed picoMIPS embedded processor for affine transformation is shown Figure 1. It consists of two parts: control path, data path:

- Control path: program counter (PC), program memory and instruction decoder
- Data path: ALU, general purpose registers and input multiplexer (input mux)

In addition to these two parts, in order to implement and test this processor on an FPGA board, the pixel coordinates were input to the ALU from the switches through an input multiplexer, then stored into the general purpose registers. Moreover, the transformed pixel coordinates were output to the LEDs from the output of ALU directly. Furthermore, for the demonstration purpose, the data input and output should follow a handshaking protocol. Therefore, a handshaking switch is used to control the increment of the program counter. More details are covered in section 2.2.

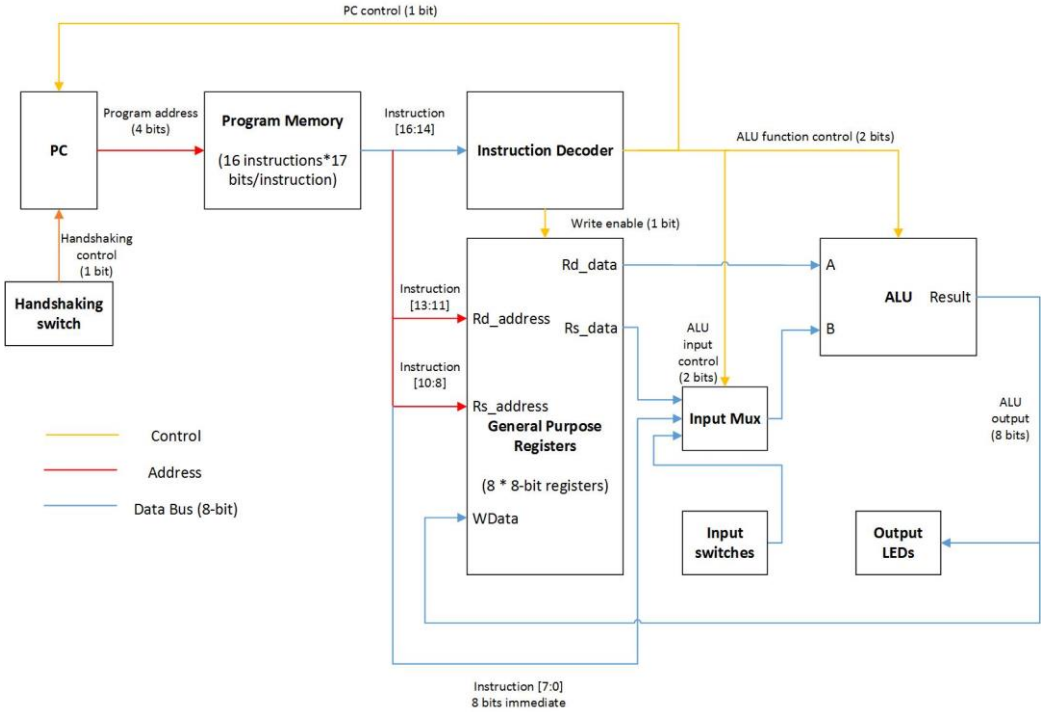


Figure 1: Block diagram of the picoMIPS embedded processor for affine transformation

2.1 Assembler program of the general affine transformation algorithm

```
0  NOP
1  IN %1, %0;          //Input X1 from SW to %1
2  NOP
3  IN %2, %0;          //Input Y1 from SW to %2
4  NOP
5  MULI %3, %1, 0.75; //%3 = X1*0.75
6  MULI %4, %2, 0.5;  //%4 = Y1*0.5
7  ADD  %3, %4;       //%3 = %3+%4
8  MULI %5, %1, -0.5; //%5 = X1*(-0.5)
9  MULI %6, %2, 0.75; //%6 = Y1*0.75
10 ADD  %5, %6;       //%5 = %5+%6
11 ADDI %3, %3, 20;   //%3 = %3+20 add coefficient B
12 ADDI %5, %5, -20;  //%5 = %5-20 add coefficient B
13 ADD  %3, %0;       //%3 = %3+%0 display the X2
14 ADD  %5, %0;       //%5 = %5+%0 display the Y2
15 NOP
```

Multiply the input data with coefficient A

IN is a user-defined instruction which stores the two input data from switches to register %1 and %2, respectively. NOP means no operation, and it is used as a halt state when using handshaking switch, more details are discussed in section 2.2.

As mentioned in section 1.2, general affine transform algorithm only involves multiplication and addition, and the coefficients A and B are stored in the program memory as immediate values. Therefore, instruction MULI was used to multiply the input data with coefficient A, and ADD was used to add the multiplied results. Instruction ADDI was used to add the sum of previous calculation with coefficient B.

2.2 Program counter(PC)

Program counter stores the address of instructions, and since there are 15 instructions in the assembler program, therefore the PC in this design is 4-bit. The PC increases by 1 at every positive clock edge by default. However, in order to agree with a handshaking protocol (Table 1) for the demonstration purpose, a state machine was designed inside the PC. The state machine detects the value of handshaking switch (SW8) and controls the increment of the PC. The state transition part of the state machine source code is shown with the Table 1. The PC was verified by running simulation in ModelSim, and the simulation results are presented in figure 2.

Table 1: Handshaking protocol

```
always_comb
case(state)
A:  next_state = (SW8)? X1:A;
X1: next_state = B;
B:  next_state = (!SW8)? C:B;
C:  next_state = (SW8)? Y1:C;
Y1: next_state = D;
D:  next_state = (!SW8)? X2;
X2: next_state = (count==4'b1000)?E:X2;
E:  next_state = (SW8)? Y2:E;
Y2: next_state = (!SW8)?A:Y2;
endcase
```

Switch value	Operations
0	Default
1	Read X1
0	
1	Read Y2
0	Calculation and display X2
1	Display Y2
0	Go back to initial address

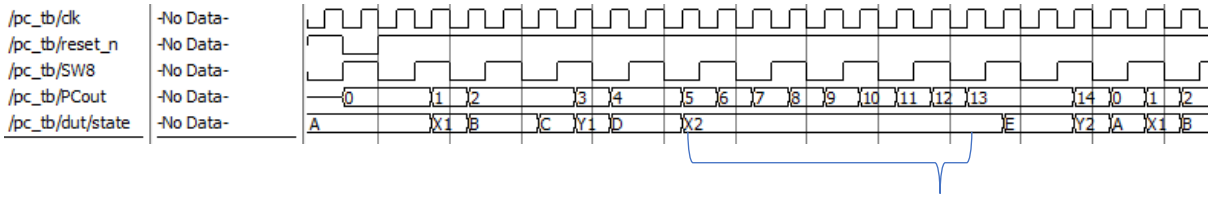


Figure 2: ModelSim simulation of PC

Figure 2 shows that:

- The output of the PC is reset to address 0 at the negative edge of the active low "reset_n" signal, wait at state A and no longer increases at every positive clock edge due to the control of state machine. At address 0, the processor executes NOP instruction repeatedly.
- When the state machine detects "SW8" becomes 1, the PC moves to address 1 at next positive edge of the clock to input X1 from switches to register %1.
- Then the PC moves to address 2 to let processor execute NOP instruction again. Meanwhile, the state moves to B to detect the "SW8" becomes 0, when this happen, move to state C to detects "SW8" becomes 1.
- When state C detects the "SW8" become 1, PC moves to address 3 to input Y1 from switches to register %2.
- Then the PC moves to address 4 to let processor execute NOP instruction again. Meanwhile, the state moves to D to detect "SW8" becomes 0.
- When state D detects the "SW8" becomes 0, the state moves to X2, and in this state, the PC increases 9 times to perform the affine transformation algorithm and stop at address 13 to output the result X2.
- Then the state moves to E to detect the "SW8" becomes 1 again, and if a 1 is detected, the PC moves to address 14 to output the result Y2. Meanwhile, state Y2 detect "SW8" becomes 0 to reset the PC address to 0.

2.3 Instruction format

The instruction format is defined based on the width of the data bus, the number of registers and instructions used.

Table 2: Instruction format (17-bit)

3-bit opcodes	3-bit address for destination register (Rd)	3-bit address for souce register (Rs)	8-bit immediate

First of all, the lowest 8 bits of the instruction are preserved for coefficients A and B. Secondly, since only 7 registers were used in assembler program for affine transformation. Therefore the address of general purpose registers is 3 bits. The upper 6 bits are used for Rd and Rs. Finally, since 5 instructions were used in the assembler program, therefore the top 3 bits is used for opcodes.

2.4 Program memory

Program memory is used to store the binary representation of the assembler program in section 2.1. It outputs the instructions to the instruction decoder according to the address from PC. The ModelSim simulation result of the program memory is shown in Figure 3. Moreover, the hexadecimal representation of the affine transformation algorithm is shown in Table 3.

Table 3: Hexadecimal representation of affine transformation algorithm

Address	Hex
0	00000
1	10800
2	00000
3	11000
4	00000
5	19960
6	1A240
7	05C00
8	19AC0
9	1B260
10	06E00
11	15B14
12	16DEC
13	05800
14	06800
15	00000

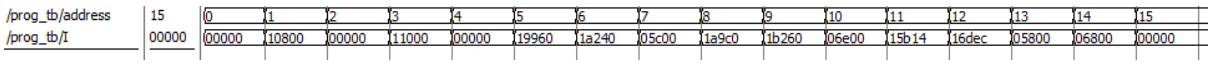


Figure 3: ModelSim simulation of program memory

The results in Figure 3 matches the expected results in Table 3. Therefore the program memory is functionally correct.

2.5 Instruction decoder

As shown in Figure 1, instruction decoder translates the opcodes in the instructions into four control signals:

- Control the increment of PC (PCincr): Pcincr=1 by default.
- Write enable of general purpose registers (w): w=1 for instructions IN, ADD, MULI and ADDI in this design, since all of them needs to store the result from ALU for further operations.
- Control the ALU input multiplexer (imm): 00 – input from the register, 01 – input from switches on the FPGA board, 11 – input immediate values from instructions.
- ALU functions: output input B directly (RB), addition (RADD) and multiplication (RMUL).

Three ALU operations were used in this design. Therefore the opcodes[1:0] is assigned to ALU functions. The ModelSim simulation result is shown in Figure 4, and the translation between opcodes and control signal is summarised in Table 4:

Table 4: opcodes

opcodes	ALU operations	w	imm
000 (NOP)	00 (RB)	0	00
100 (IN)	00 (RB)	1	00
001 (ADD)	01 (RADD)	1	01
110 (MULI)	10 (RMUL)	1	11
101 (ADDI)	01 (RADD)	1	11

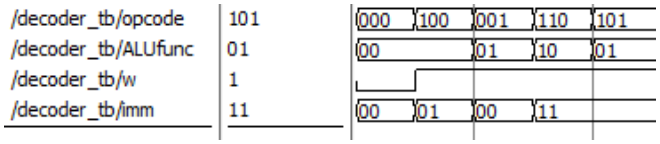


Figure 4: ModelSim simulation of Instruction decoder

The results in Figure 4 matches the expected results in Table 4. Therefore the instruction decoder is functionally correct.

3. General Purpose Register file design, simulation and synthesis

Seven general purpose registers were used in this design as mentioned in section 2.1. Their purposes are summarised in Table 5:

Table 5: General Purpose Registers

Register number	Purpose
%0	%0 = 0 is preserved
%1	Dedicated register for input X1
%2	Dedicated register for input Y1
%3	Store intermediate values and final result X2
%4	Store intermediate values
%5	Store intermediate values and final result Y2
%6	Store intermediate values

The write enable signal of the general purpose register is controlled by the instruction decoder, and the write process is synchronised with the clk signal, but the read process is not. To verify this module, it is simulated in ModelSim. The result is shown in Figure 5:

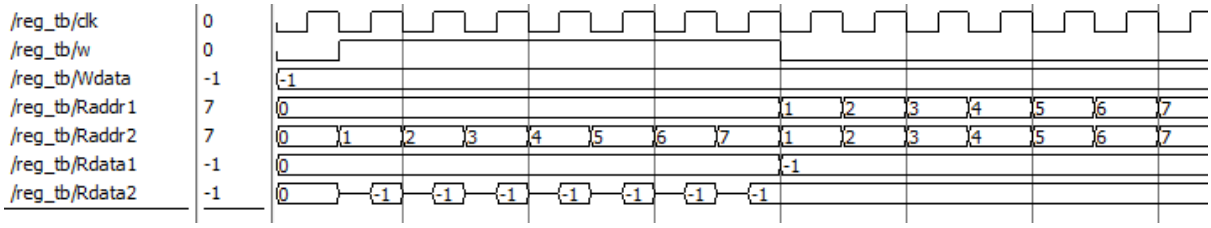


Figure 5: ModelSim simulation of General Purpose Registers

Figure 5 shows that:

- First of all, the write enable is set to 0, and the addresses were set to 0 to test register %0, the data outputs (Rdata1 and Rdata2) indicates register %0 = 0
- Secondly, the write enable is set to 1 for 7 clock cycles. Meanwhile, the address of destination register (Raddr2) increases at every positive clock edge to allow the input data (Wdata = -1) to be written into the registers. Because the read process is asynchronous, therefore once a value is written into the register, it appears at the data output (Rdata2)
- Finally, the write enable is set to 0 again, and address of registers increases every clock cycle. The data outputs (Rdata1 and Rdata2) indicates that the registers can be written and read correctly.

4. ALU design and the use of embedded multiplier

Arithmetic Logic Unit is used for arithmetic calculations such as addition, subtraction, multiplication and division, and logic operations such as AND, OR, XOR, NOT. In this design, the affine transformation algorithm only consists of addition and multiplication. Therefore the ALU is tailored to reduce the hardware cost. Moreover, in order to input data from switches, an RB operation is used to output the input B of ALU directly. The block diagram of the ALU is shown in Figure 6. Two data inputs a and b are 8-bit signed binary number, the “func” signal is controlled by the instruction decoder and used for output selection, and multiplier enable signal as shown in Table 6:

Table 6: ALU output selection

Func value	Result
00 (RB)	b
01 (RADD)	ar
10 (RMUL)	m[14:7] (fraction part is discarded)

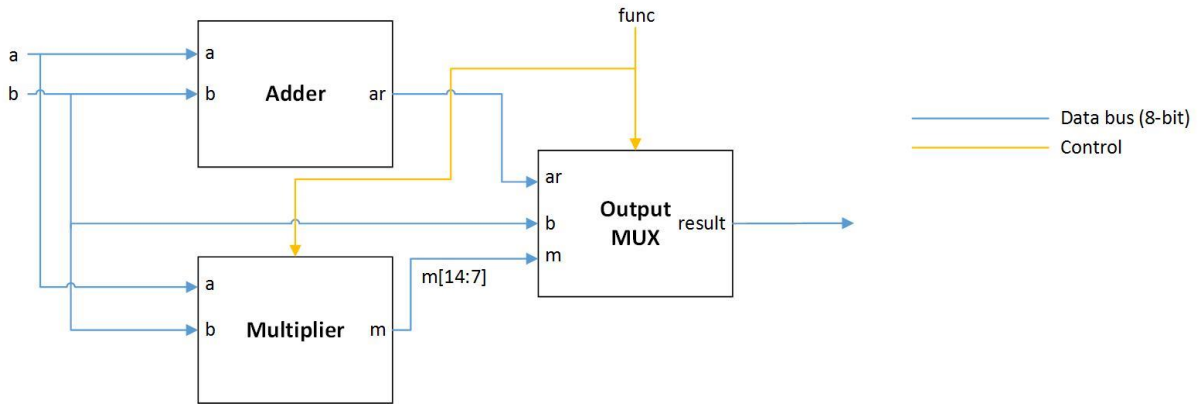


Figure 6: Block diagram of the ALU module

4.1 Adder design and the use of embedded multiplier

```
//adder
logic[n-1:0] ar;
always_comb
begin
    ar = a+b;
end
```

```
//Multiplier
logic [15:0] m;
always_comb
begin
    if(func==`RMUL)
     m = a*b;
else
     m = '0;
end
```

The source code of the adder module and the multiplier module is given above. After the processor was synthesised on the Altera Cyclone IV FPGA. The adder module was mapped into standard logic. However, the multiplier was mapped into the embedded multiplier.

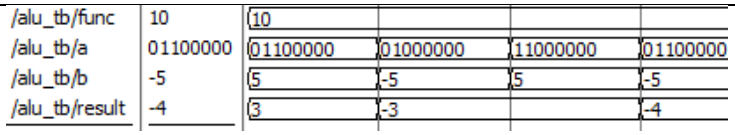
4.2 Verification of the ALU

To verify the ALU, it was simulated in the ModelSim. The results in Table 7 indicate the ALU module is functionally correct:

Table 7: ModelSim simulation of ALU

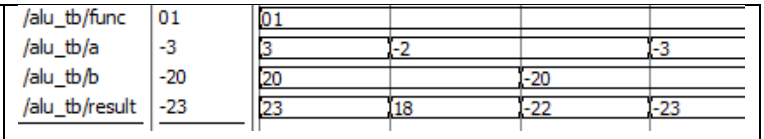
```
       func =2'b10;//MUL
       a=8'b01100000;//0.75
       b=8'b00000101;//5
#10ns  a=8'b01000000;//0.5
       b=8'b11111011;//-5
#10ns  a=8'b11000000;//-0.5
       b=8'b00000101;//5
#10ns  a=8'b01100000;//0.75
       b=8'b11111011;//-5
```

Expected results: 3, -3, -3, -4



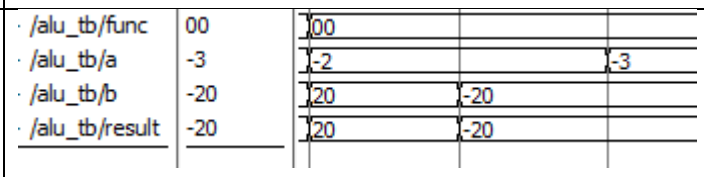
```
#10ns  func=3'b01;//ADD
       a=8'b00000011;//3
       b=8'b00010100;//20
#10ns  a=8'b11111110;//-2
       b=8'b00010100;//20
#10ns  a=8'b11111110;//-2
       b=8'b11101100;//-20
#10ns  a=8'b11111101;//-3
       b=8'b11101100;//-20

Expected results: 23, 18, -22, -23
```




```
#10ns  func=2'b00;//RB
       a=8'b11111110;//-2
       b=8'b00010100;//20
#10ns  a=8'b11111110;//-2
       b=8'b11101100;//-20
#10ns  a=8'b11111101;//-3
       b=8'b11101100;//-20

Expected results: 20, -20, -20
```



4.3 Verification of the entire CPU core

After all sub-modules had been verified, the whole CPU core was simulated in ModelSim as shown in Figure 7.

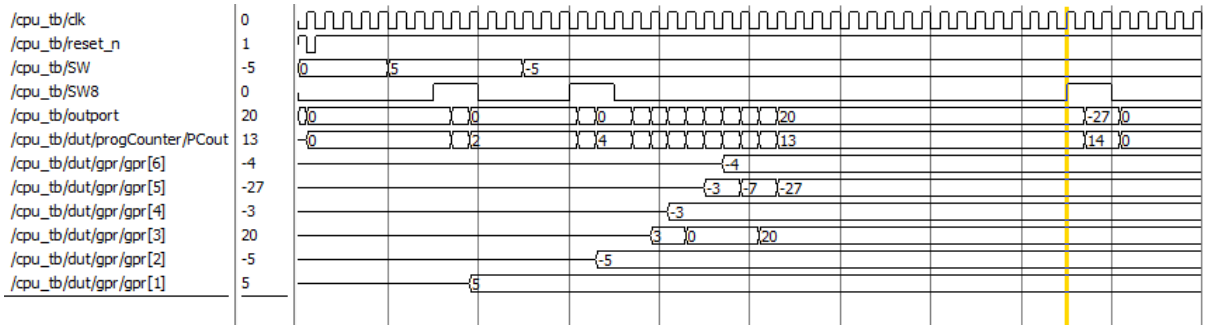


Figure 7: ModelSim simulation of the CPU

The inputs used in the testbench were 5 and -5, and the expected results should be 20 and -27. The Figure 7 shows that the input data was written into the general purpose register %1 and %2, respectively. Then they were multiplied with coefficient A, and the results were stored in %3 and %5. Then the multiplied results were added with coefficient B and stored back to %3 and %5, the values in the figure matches the expected results. "Outport" signal shows the final result -27 was delayed, this is due to waiting for the handshaking switch.

5. Altera DE2-115 implementation

After the behaviour model of the picoMIPS processor was verified, it was synthesised on an Altera Cyclone IV FPGA (EP4CE115F29C7) on the DE2-115 development board. Then the CPU core was instantiated in a top-level module with a clock divider. The purpose of using clock divider was to slow down the main clock to eliminate the bouncing effects of the mechanical switches on the board. Then test vectors were input through the switches, and the results were checked in compliance with the handshaking protocol.

5.1 Write synthesisable RTL codes

To synthesise the design on an FPGA, the RTL codes of the design must be synthesisable. There were errors indicate some combinational logic in the design was not purely combinational. To fix this issues, attention needs to be given to the incomplete conditional statements.

5.2 Synthesis optimisation

The specification of this design requires the design uses as fewer logic resources as possible. Therefore, when synthesising the design, the optimisation technique in the synthesis setting needs to be changed from "Balanced (default)" to "Area". The results are shown in Table 8:

Table 8: Effect of using synthesis optimisation technique

Technique	Logic resources
Balanced	193
Area	190

5.3 Altera Quartus II 13.0 web edition vs. 15.0 full edition

The synthesis results from the previous section were obtained from Quartus II 13.0 web edition. To optimise this design area further, Quartus II 15.0 full edition was used. Moreover, the results are shown in Table 9, and they indicate that higher version and edition of the design tool use better synthesis algorithm.

Table 9: Effect of using Quartus II full edition

Edition	Logic resources
13.0 Web	190
15.0 Full	180

5.4 Reduce the length of instruction

The first version of this design using 24-bit instruction. The second version reduced this length to 17 bits. However, the cost figure did not change.

6. Conclusion

This report introduces the detailed development of an embedded picoMIPS processor for the affine transformation of graphic pixels. The functional requirements in the design specification were fully achieved. Moreover, by using the equation in section 1.3. The cost figure of this design is 180 as shown in Figure 8. Through this coursework, my knowledge of computer architecture has been boosted, and my ability to do digital hardware design has been reinforced. In the future, the cost figure can be reduced further by utilising the RAM to store the program, and the handshaking protocol could be implemented by adding branch instructions and modify the program, instead of using a state machine to control handshaking protocol which cost hardware resources.

Quartus II 64-Bit Version	15.0.0 Build 145 04/22/2015 SJ Full Version
Revision Name	cpu
Top-level Entity Name	cpu
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	180 / 114,480 (< 1 %)
Total combinational functions	174 / 114,480 (< 1 %)
Dedicated logic registers	68 / 114,480 (< 1 %)
Total registers	68
Total pins	19 / 529 (4 %)
Total virtual pins	0
Total memory bits	0 / 3,981,312 (0 %)
Embedded Multiplier 9-bit elements	1 / 532 (< 1 %)
Total PLLs	0 / 4 (0 %)

Figure 8: Cost figure of the final design