SOC Design Report

Xuan Ban
xb1g10
Yang Huihuang
hy9g10
Dedic Charlie
cd2g10
Wang Chu
cw9g10

Electronics and Computer Science
University of Southampton

November 8, 2014

Abstract

Abstract goes here. Use this latex template as our report template.

Contents

1 Introduction

In this exercise, the final goal is to design, test and demonstrate an 8-bits affine transformation implementation of picoMIPS on an Altera FPGA development system. A simple affine transform[1] can be defined as such:

$$\begin{pmatrix} x2 \\ y2 \end{pmatrix} = A \times \begin{pmatrix} x1 \\ y1 \end{pmatrix} + B \tag{1}$$

The specific A and B datasets were chosen for the demonstration in this exercise, which are:

$$A = \begin{bmatrix} 0.75 & 0.5 \\ -0.5 & 0.75 \end{bmatrix} B = \begin{bmatrix} 20 \\ -20 \end{bmatrix} \tag{2}$$

In the picoMIPS program design, the six transformation constants must be included as immediate literals. The switches SW0-7 on the FPGA development system are required for Pixel coordinates input. The LEDs LED0-7 are assigned to display the transformation result. Switch SW8 controls the handshaking functionality to complete the transform sequence, as described below. The switch SW9 is used as an active low reset.

1	When SW8 is on read the coordinate x1 from SW[7:0], otherwise wait.
2	Wait until SW8 is off
3	When SW8 is on read the coordinate y1 from SW[7:0], otherwise wait
4	Wait until SW8 is off
5	Execute the affine transformation for x2 and display it on LED[7:0]
6	Wait until SW8 is on
7	Execute the affine transformation for y2 and display it on LED[7:0]
8	Wait until SW8 is off. If so go back to step 1

Table 1: Instruction Set

During the preparation, the picoMIPS architecture and SystemVerilog code examples of each picoMIPS block that are provided by ELEC6016 lecture slides were well studied and tested in the Modelsim. During the first stage of the design, a set of Systemverilog codes were developed based on the preparation and a successful working design was achieved to fulfil the basic requirement of the exercise. Separate testbenchs for the individual modules and the top-level design were also implemented and simulated in Modelsim to verify the correct functionality of the design. The input and output 8bit data were assigned to the designated ports. All modules were successfully synthesised and programed into the FPGA Development System.

At the second stage of the development, the goal was focused on optimizing the original design to reduce the cost figure as much as possible. The cost figure is defined as:

Cost = number of Logic Elements used + 30 x Kbits of RAM used

The optimization includes the usage of the synchronous RAM and simplification of the instruction set and other picoMIPS modules. As the result, the cost figure was successfully reduced from 132 to 76.

This report is split into two major sections. Section 2-4 will describe the original design and section 6 will discuss the AE0 synthesis and optimized design in detail. The cost figure discussion for each design will be included in conclusion section.

2 Introduction

In this exercise, the final goal is to design, test and demonstrate an 8-bits affine transformation implementation of picoMIPS on an Altera FPGA development system. A simple affine transform[1] can be defined as such:

$$\begin{pmatrix} x2 \\ y2 \end{pmatrix} = A \times \begin{pmatrix} x1 \\ y1 \end{pmatrix} + B \tag{3}$$

The specific A and B datasets were chosen for the demonstration in this exercise, which are:

$$A = \begin{bmatrix} 0.75 & 0.5 \\ -0.5 & 0.75 \end{bmatrix} B = \begin{bmatrix} 20 \\ -20 \end{bmatrix} \tag{4}$$

In the picoMIPS program design, the six transformation constants must be included as immediate literals. The switches SW0-7 on the FPGA development system are required for Pixel coordinates input. The LEDs LED0-7 are assigned to display the transformation result. Switch SW8 controls the handshaking functionality to complete the transform sequence, as described in table 1. The switch SW9 is used as an active low reset.

1	When SW8 is on read the coordinate x1 from SW[7:0], otherwise wait.
2	Wait until SW8 is off
3	When SW8 is on read the coordinate y1 from SW[7:0], otherwise wait
4	Wait until SW8 is off
5	Execute the affine transformation for x2 and display it on LED[7:0]
6	Wait until SW8 is on
7	Execute the affine transformation for y2 and display it on LED[7:0]
8	Wait until SW8 is off. If so go back to step 1

Table 2: handshaking protocol

During the preparation, the picoMIPS architecture and SystemVerilog code examples of each picoMIPS block that are provided by ELEC6016 lecture slides were well studied and tested in the Modelsim. During the first stage of the design, a set of Systemverilog codes were developed based on the preparation and a successful working design was achieved to fulfil the basic requirement of the exercise. Separate testbenchs for the individual modules and the top-level design were also implemented and simulated in Modelsim to verify the correct functionality of the design. The input and output 8bit data were assigned to the designated ports. All modules were successfully synthesised and programed into the FPGA Development System.

At the second stage of the development, the goal was focused on optimizing the original design to reduce the cost figure as much as possible. The cost figure is defined as:

Cost = number of Logic Elements used + 30 x Kbits of RAM used

The optimization includes the usage of the synchronous RAM and simplification of the instruction set and other picoMIPS modules. As the result, the cost figure was successfully reduced from 132 to 76.

This report is split into two major sections. Section 2-4 will describe the original design and section 6 will discuss the AE0 synthesis and optimized design in detail. The cost figure discussion for each design will be included in conclusion section.

3 Instruction set

3.1 Instruction format

During the project, there were seven types of instructions have been implemented. In the original design, each instruction was composed of 4 bits operand, 3 bits source and destination register address and 8 bits immediate (18 bits in total). The picoMIPS architecture design is illustrated in Diagram 1. The opcode bits were assigned to the decoder. Source and destination address bits were allocated to general purpose register. Immediate bits were connected to the ALU for immediate arithmetic operation. In this way each instruction could also be used to interact with register. All Instruction types are summarised in table 2.

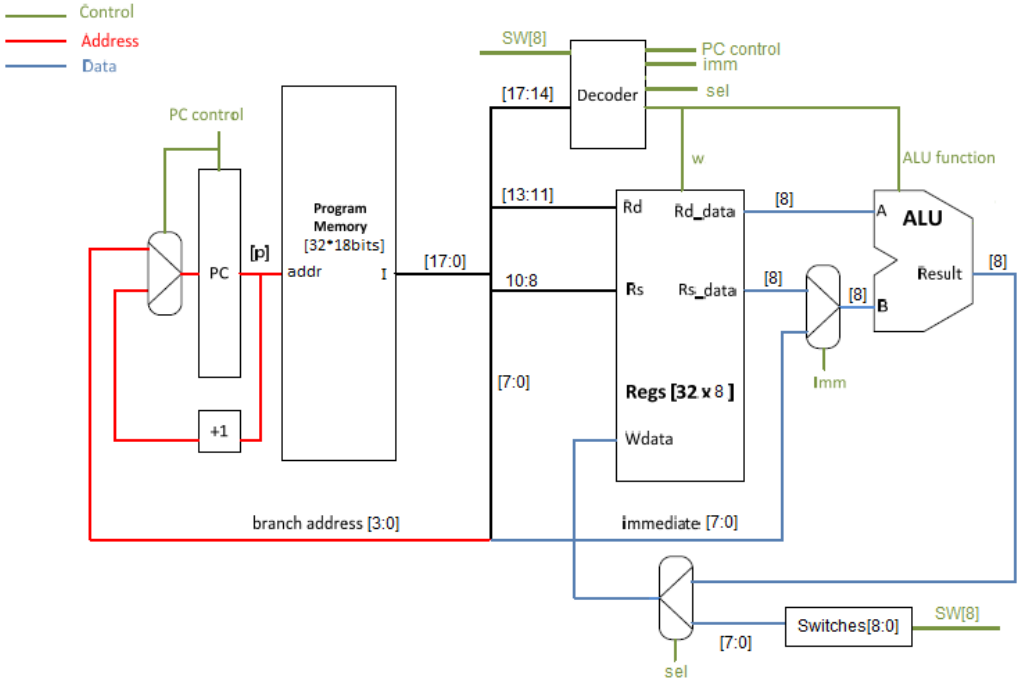


Figure 1: The origianl picoMIPS architecture design [2]

Instruction	Acronym	Description
Addition	ADD	$Rd \leftarrow Rd + Rs$
Addition immediate	ADDI	$Rd \leftarrow Rd + Imm$
Load switches SW[7:0]	LOAD	$Rd \leftarrow Switches[7:0]$
Multiply immediate	MUXI	$Rd \leftarrow Rd \times Imm$
Wait while SW[8]= 0	WAIT0	$if(SW[8] = 1)PC++$
Wait while SW[8]= 1	WAIT1	$if(SW[8] = 0)PC++$
Absolute jump	JUMP	$PC \leftarrow 0$

Table 3: Instruction Set

3.2 Control Instruction

The affine transform does not require any decisions based on the data. Therefore there is no flag needed, neither is the conditional branch. The only necessary control signal is for the handshaking procedure, and it comes from switch SW8. During the handshaking, the program counter will not increment until value of switch SW8 is correct. In this case, instruction *WAIT0* and *WAIT1* were implemented to accomplish this operation. At the end of the affine transform, an absolute jump is essential for returning the transform operation to the start. The instruction *JUMP* replaces the value of the program counter with its last four bits. So in fact, it can make the processor to jump to any instruction location. But in this project, it only resets the program counter.

3.3 Arithmetic and register instruction

In order to accomplish an affine transform, at least two kinds of arithmetic are needed: addition and multiplication. In the processor, the corresponding arithmetic instructions were *ADD* and *MUX*. They consist of the operand codes that will deliver operating commands to Arithmetic Logic Unit (ALU), designated source and destination register addresses. In addition, instruction *ADDI* and *MUXI* were developed for the arithmetic with signed immediate values. The immediate instructions select build-in immediate values rather than the ones from source register. It is very convenient when used in the calculations with matrix constants. No subtract instruction was implemented to reduce the size of the ALU.

At the beginning of the affine transform, x1 and y1 values are provided by the switches. The processor needs to read and store them in the certain memory locations. Therefore a store instruction *LOAD* is implemented for this purpose. Since the register will also be accessed in conjunction with other arithmetic instructions, the register addresses are included in those instructions.

4 Program Counter, Program memory and Decoder design

4.1 Program Counter

Program counter is the simplest sequential logic module in the system. The counter will increment every clock cycle if the increment control signal *PCincr* is high. If the *JUMP* instruction is called, the absolute branch control signal *PCabsbranch* will force the counter to jump to the address given by the instruction. In this case, it would be the start position of the operation. The program counter size was set to 5 bits, so it can count up to 32 instructions.

```
1  module pctest;
2   parameter Psize = 5;
3   logic clk, reset, PCincr,PCabsbranch;
4   logic [Psize-1:0] Branchaddr;
5   logic [Psize-1 : 0]PCout;
6
7  pc  #(.Psize(Psize)) P(.*);
8
9  //-------------- code starts here ----------
10 //clock
11 initial
12 begin
13   clk =  0;
```

```
14   #5ns  forever #5ns clk = ~clk;
15 end
16
17 initial
18 begin
19   //default
20  reset = 1;
21  PCincr = 0;
22  Branchaddr = 4'b0000;
23  PCabsbranch = 0;
24
25   // test all functions
26   #10 reset = 0;
27   #10 PCincr = 1; //start counting
28   #10 PCincr = 0; //stop counting
29   #10 PCincr = 1;
30   #50 PCabsbranch = 1;PCincr = 0; //JUMP
31   #10 PCabsbranch = 0;
32   #10 PCincr = 1;
33
34 end
35
36 endmodule //end of pctest
```

Listing 1: Testbench for program counter

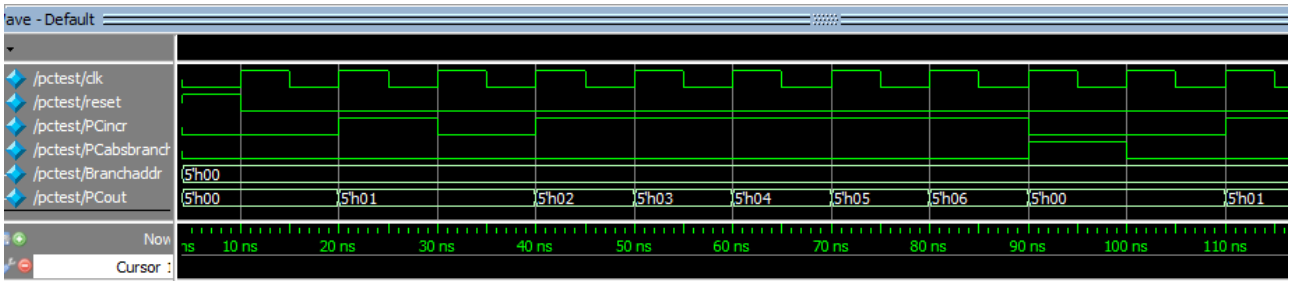


Figure 2: Testbench result of Program Counter

The testbench and simulation result of program counter are shown in listing 1 and Figure 2. At 20ns, the increment control signal PCincr went high so the counter started to count up. At 30ns, PCincr went low so counter stopped counting at the current clock cycle. At 90ns, the testbench simulated a *JUMP* instruction: *PCincr* became low and absoluate branch control signal *PCabsbranch* went high. In this case, the counter loads the address from Branch address *Brachaddr* (0000) and restarts the counting.

The RTL level synthesis of program counter is shown in Figure 3.

4.2 Program Memory

In general, the program memory is a ROM with asynchronous read. It has one input port to receive the address sent by program counter. The address extracts the corresponding instruction stored in the memory to the output port. All the instructions were decoded in hexadecimal format and stored in the separate hex file *prog.hex*. During the initiation of the program memory, this file was loaded into the program RAM. The program address width and instruction length were acquired for the declaration of the program memory.

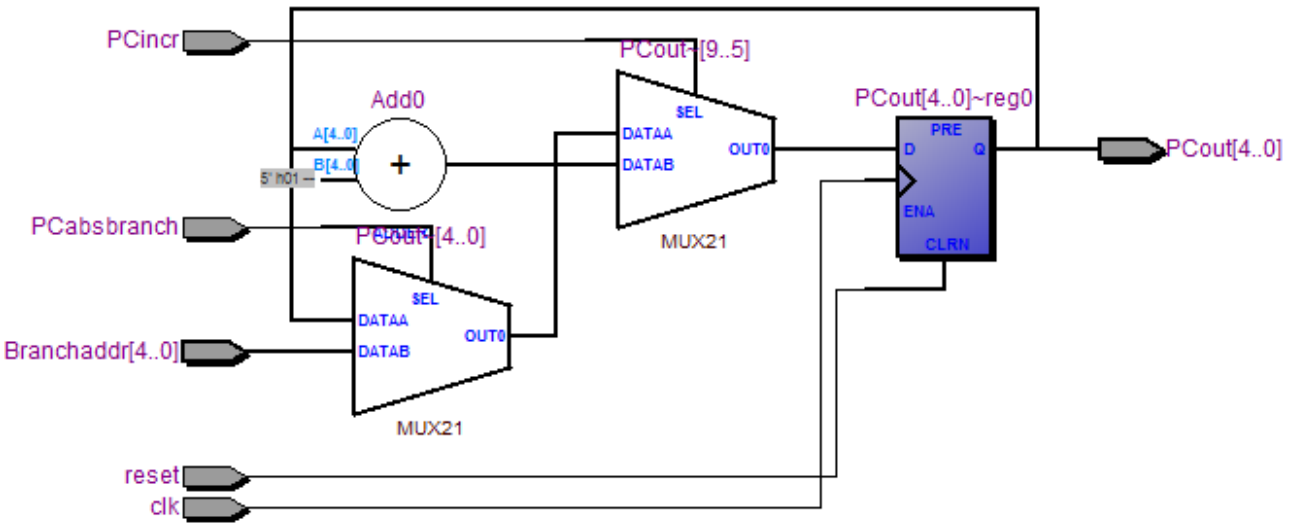


Figure 3: RTL synthesis of Program Counter

The testbench codes are demonstrated in listing 2.

```
1
2  module progtest;
3  parameter Psize = 5, Isize = 18;
4  logic [Psize-1:0] address;
5  logic [Isize-1:0] I; // I - instruction code
6
7  // program memory declaration, note: 1<<n is same as 2^n
8  logic [Isize:0] progMem[ (1<<Psize)-1:0];
9
10 prog #(.Psize(Psize),.Isize(Isize)) r(.*);
11
12 //------------- code starts here ----------
13 initial
14 begin
15   //try all the memory address
16   address [Psize-1:0] = 5'h00;
17   #10ns  address [Psize-1:0] = 5'h01;
18   #10ns  address [Psize-1:0] = 5'h02;
19   #10ns  address [Psize-1:0] = 5'h03;
20   #10ns  address [Psize-1:0] = 5'h04;
21   #10ns  address [Psize-1:0] = 5'h05;
22   #10ns  address [Psize-1:0] = 5'h06;
23   #10ns  address [Psize-1:0] = 5'h07;
24   #10ns  address [Psize-1:0] = 5'h08;
25   #10ns  address [Psize-1:0] = 5'h09;
26   #10ns  address [Psize-1:0] = 5'h0A;
27   #10ns  address [Psize-1:0] = 5'h0B;
28   #10ns  address [Psize-1:0] = 5'h0C;
29   #10ns  address [Psize-1:0] = 5'h0D;
30   #10ns  address [Psize-1:0] = 5'h0E;
31   #10ns  address [Psize-1:0] = 5'h0F;
32   #10ns  address [Psize-1:0] = 5'h10;
33   #10ns  address [Psize-1:0] = 5'h11;
34   #10ns  address [Psize-1:0] = 5'h12;
35 end
36
37 endmodule // module regstest
```

Listing 2: Testbench for program Memory

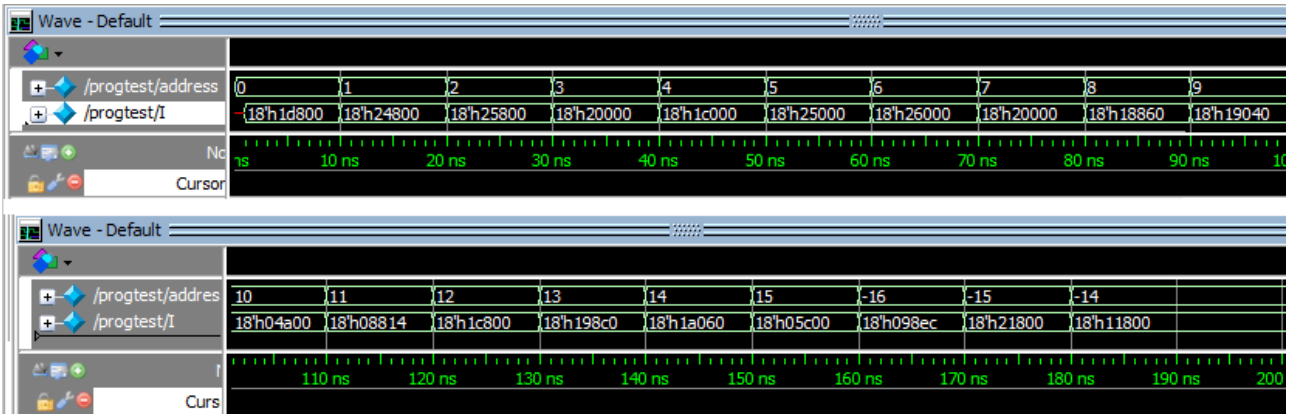


Figure 4: Testbench result of Program Memory

As shown in Figure 4, when the testbench feed an address to the program memory, the corresponding instruction was extracted at the output. The testbench attempted all the memory address and outputted instructions were verified correct according to the hex file *prog.hex*.

The RTL synthesis of program memory is shown in Figure 5. Note that the write-enable of the memory is always zero, which makes the program memory a READ-ONLY memory.

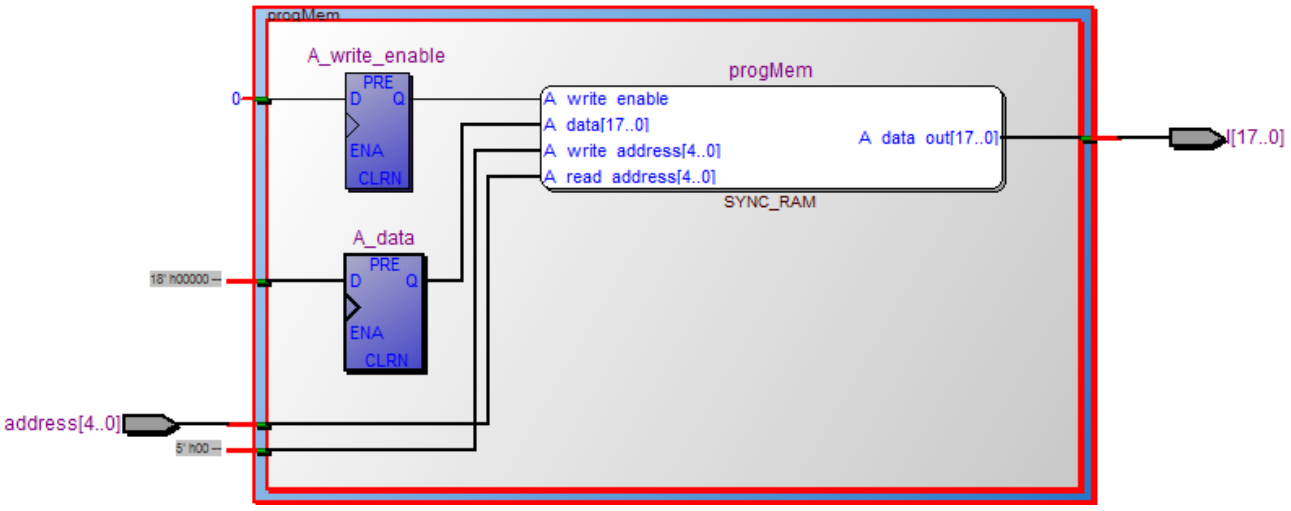


Figure 5: RTL synthesis of Program Memory

4.3 Decoder

The decoder was implemented as a combinational logic. It decodes the opcodes, which is the first four bits of the instruction. Eight types of instructions mentioned in the instruction chapter will be read from the program memory and translate into the control signals for ALU, program counter and general purpose register. The decoder also reads the control signal from switch SW[8] on the top level so the instruction *WAIT0* and *WAIT1* can be executed correctly. If an unexpected opcode is

provided in the decoder, an error message will display for the debugging. Each opcode was defined in file *opcodes.sv*. So it is easier to recognize and edit during the programming. ALU function control signals were defined in file *alucodes.sv* for the same purpose. This file is also used in Arithmetic Logic Unit (ALU).

The opcode part of the decoder code is show in the listing 3.

```
1
2  module decoder;
3  .....
4   case(opcode)
5       //do nothing
6       `NOP: PCincr = 1'b1;
7
8       //register-register add
9        `ADD:begin
10            w = 1'b1; // write result to dest register
11            PCincr = 1'b1;
12            ALUfunc = `RADD;
13          end
14
15      //register-immediate add
16        `ADDI:begin
17              w = 1'b1;
18              imm = 1'b1; // select immediate operand
19              PCincr = 1'b1;
20              ALUfunc = `RADD;
21          end
22
23      //register-register multiply
24        `MUX: begin
25              w = 1'b1; // write result to dest register
26              ALUfunc = `RMUX;
27              PCincr = 1'b1;
28          end
29
30      //register-immediate multiply
31        `MUXI: begin
32               w = 1'b1;
33               imm = 1'b1;
34               ALUfunc = `RMUX;
35               PCincr = 1'b1;
36          end
37
38      //wait while 1
39        `WAIT1: begin
40              if(SW == 1'b1) //if SW[8] == 0 PC++
41               begin
42               PCincr = 1'b0;
43               sel = 1'b0;
44              end
45              else
46               begin
47               PCincr = 1'b1;
48               sel = 1'b0;
49              end
50          end
51
52      //wait while 0
53        `WAIT0: begin
54              if(SW == 1'b1) //if SW[8] == 1 PC++
```

```
55              begin
56              PCincr = 1'b1;
57              sel = 1'b0;
58              end
59             else
60              begin
61              PCincr = 1'b0;
62              sel = 1'b0;
63              end
64         end
65
66      //Read Switches[7:0] and load into register
67        `LOAD: begin
68             w = 1'b1;
69             PCincr = 1'b1;
70             sel = 1'b1;
71        end
72
73      //JUMP (unconditional absolute branch)
74        `J:  begin
75             PCincr = 1'b0;
76             PCabsbranch = 1'b1;
77        end
78
79      default:
80      error("unimplemented opcode");
81   endcase // opcode
82 endmodule
```

Listing 3: Opcode part of Decoder code

Decoder Opcode is shown in listing 4.

```
1 `define NOP    4'b0000
2 `define ADD    4'b0001
3 `define ADDI   4'b0010
4 `define J      4'b0100
5 `define MUX    4'b0101
6 `define MUXI   4'b0110
7 `define WAIT0  4'b0111
8 `define WAIT1  4'b1000
9 `define LOAD   4'b1001
```

Listing 4: opcode.sv

Similar to program memory, the testbench examines all control signals can be decoded correctly for all types of opcode, as shown in listing 5 .

```
1 `include "alucodes.sv"
2 `include "opcodes.sv"
3
4 module decodertest;
5
6 logic [3:0] opcode; // top 6 bits of instruction
7 logic [2:0] ALUfunc;
8 logic SW;
9 logic PCincr,imm,w,PCabsbranch,sel;
10
11 decoder c(.*);
12 //--------------- code starts here ----------
13
```

```
14 initial
15 begin
16   SW = 0;
17   #10ns opcode =4'b0111; //WAIT0
18   #10ns SW=1;
19   #10ns opcode =4'b1000; //WAIT1
20   #10ns SW=0;
21   #10ns opcode =4'b1001; //LOAD
22   #10ns opcode =4'b0110; //MUXI
23   #10ns opcode =4'b0001; //ADD
24   #10ns opcode =4'b0010; //ADDI
25   #10ns opcode =4'b0100; //JUMP
26   end
27 endmodule
```

Listing 5: Decoder testbench

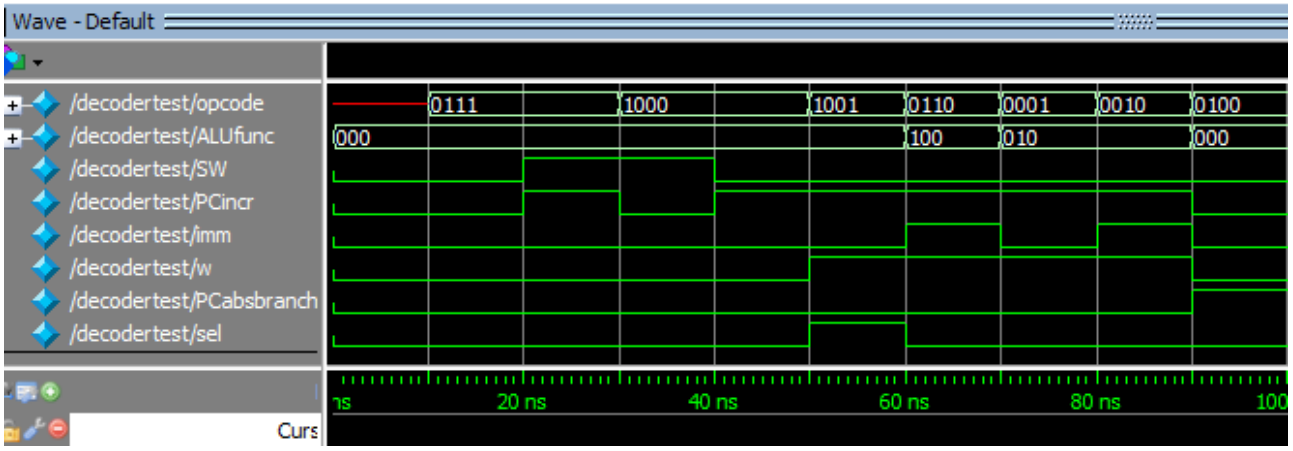


Figure 6: Simulation results of Decoder

From the simulation results in Figure 6, at 0ns, the program counter does not increment (PCincr = 0) in default state (no opcode input). When the opcode is *WAIT0* (at 10ns) or *WAIT1* (at 30ns), the program counter signal *PCincr* depends on switch signal *SW*.

At 50ns, because the opcode is *LOAD*, the "write-to-register" signal w and "read-from-switches" signal *sel* rise to high. The arithmetic opcode decoding test starts from 60ns. The decoder outputs the ALU control signal *ALUfunc* accordingly, along with the write signal w. If the arithmetic opcodes involve the immediate value (at 60ns and 80ns, the control signal *imm* will rise to high. The last opcode in the simulation is *JUMP*, therefore the absolute branch signal *PCabsbranch* becomes high.

The RTL synthesis is shown in Figure 7.

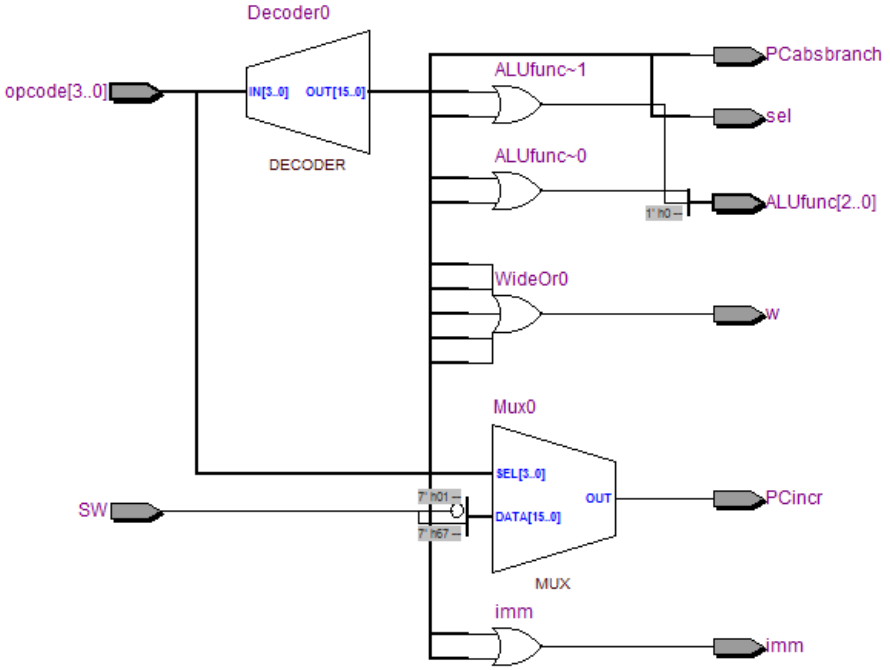


Figure 7: RTL synthesis of Decoder

5 General Purpose Register file design

The general purpose register was implemented as the multiple-port RAM with asynchronous read. It has two register address input ports and two register value output ports. The register-to-register calculation is greatly benefited of the dual-port design. In total there were five registers declared and each register could store one 8bits data. Different from the program memory, the general purpose register must also be able to restore the data to the designated address. However the register will not record the data only if the write-to-register signal w from the decoder rises to high. The fifth to seventh bits of the instruction were used as the destination register address. Eighth and tenth bits were the source register address.

The testbench code is shown in listing 6.

```
1  module regstest;
2
3  parameter n = 8;
4  logic clk, w, reset;
5  logic [n-1:0] Wdata;
6  logic [2:0] Rdno, Rsno;
7  logic [n-1:0] Rd, Rs;
8
9  regs  #(.n(n)) r(.*);
10
11 //--------------- code starts here ----------
12 //clock
13 initial
14 begin
15   clk =  0;
16   #5ns  forever #5ns clk = ~clk;
17 end
18
```

```
19 initial
20 begin
21     w = 0;
22     Wdata = 8'b00000000;
23     reset = 1;
24     Rdno = 0; Rsno =0;
25   #10ns Wdata = 8'b00011111; reset =0;
26   #10ns w =1; Rdno = 3'b001; //write data in register 1
27   #10ns w = 0;
28   #10ns Wdata = 8'b00001111;
29   #10ns w =1; Rdno = 3'b010; //write data in register 2
30   #10ns w=0;
31   #10ns Rdno = 3'b001;//test Rd
32   #10ns Rsno = 3'b010;//test Rs
33 end
34 endmodule // module regstest
```

Listing 6: General Purpose Register testbench

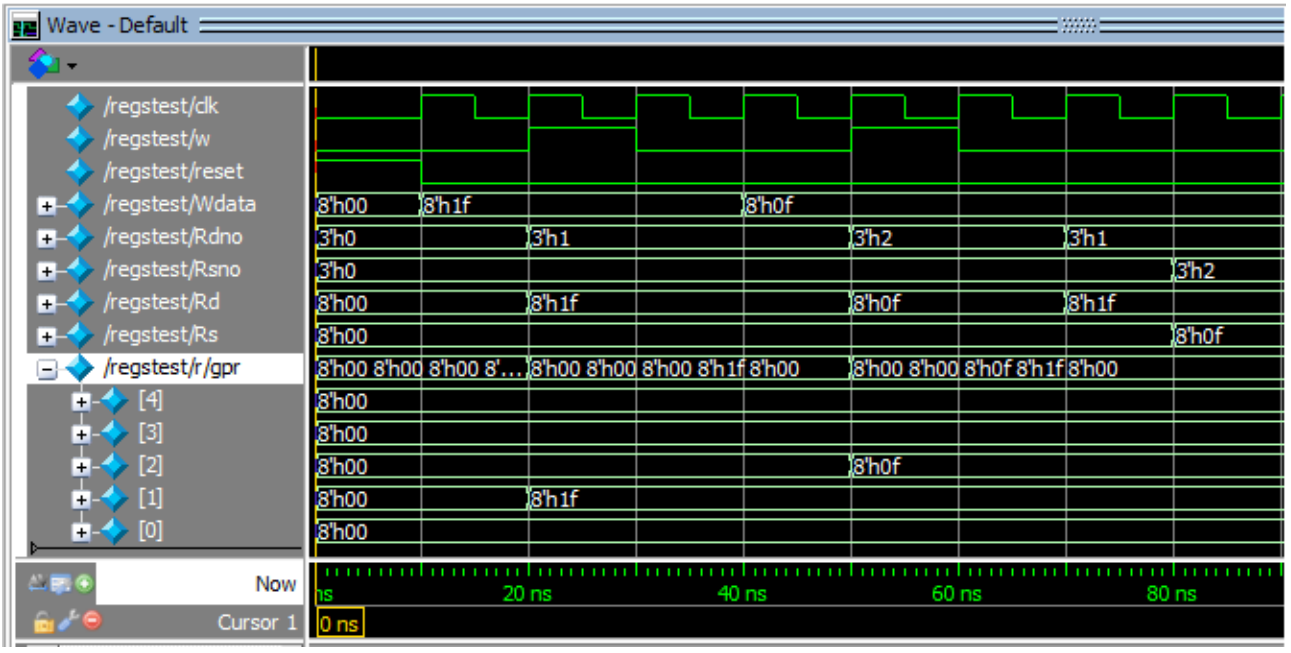


Figure 8: Simulation results of General Purpose Register

In the testbench simulation there were two data written in the register 1 and 2 at different time. Because the register has asynchronous read, so the data could be written in and read out during the same clock cycle. At 70ns and 80ns, destination and source addresses were changed, and so were the data output at the destination and source register. This behaviour indicates the correct operation of the general purpose register.

However, Altera Cyclone III in FPGA only supports the synchronous memory block. Therefore the RTL synthesis tool have to use plenty of combinational logic units instead of the build-in RAM to create the required register. This results in a large size sequential register with no use of any memory bits, as shown in Figure 9.

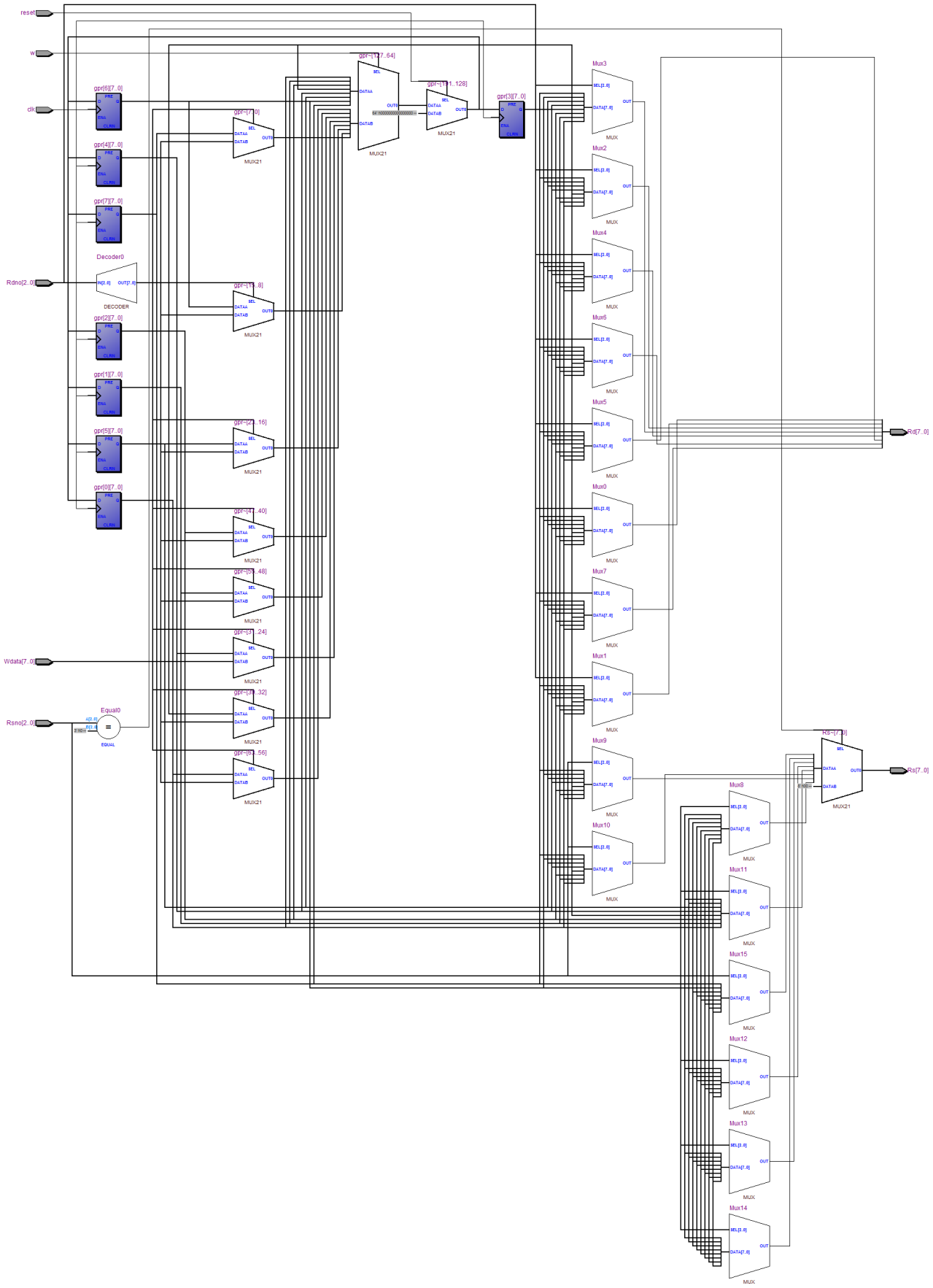


Figure 9: RTL synthesis of General Purpose Register

6 Arithmetic Logic Unit (ALU) and Multiplier design

ALU is a complex combination logic used to execute all the arithmetic operation. It reads ALU operands from general purpose register or immediate bits in the instructions. The ALU function depends on the control opcode *ALUfunc* given by the decoder. In this project, only addition and multiplication were developed in the ALU, and no sign extension was used. The corresponding ALU opcodes were defined as *RADD* and *RMUX*. When ALU is not used, the input operands can pass it by using ALU opcode *RA* and *RB*.

6.1 Multiplier

A signed 8×8 combinational multiplier was implemented in the ALU to perform the calcuation. The basic operation of the multiplier can be explained with Figure 10. In the example, multiplicand X = $x_7x_6x_5x_4x_3x_2x_1x_0$ and multiplier Y = $y_7y_6y_5y_4y_3y_2y_1y_0$. In each row Each bit of the multiplier is multiplied against the shifted multiplicand. Each small box represents the partial product y_ix_i, which is the logical *AND* of multiplier bit y_i and multiplicand bit x_j. And it is aligned according to the position of the bit within the multiplier. The final product P = $p_{15}p_{14}...p_2p_1p_0$ is obtained at the bottom by adding all the partial products. The 8bit number multiplication yields a 16bit result.

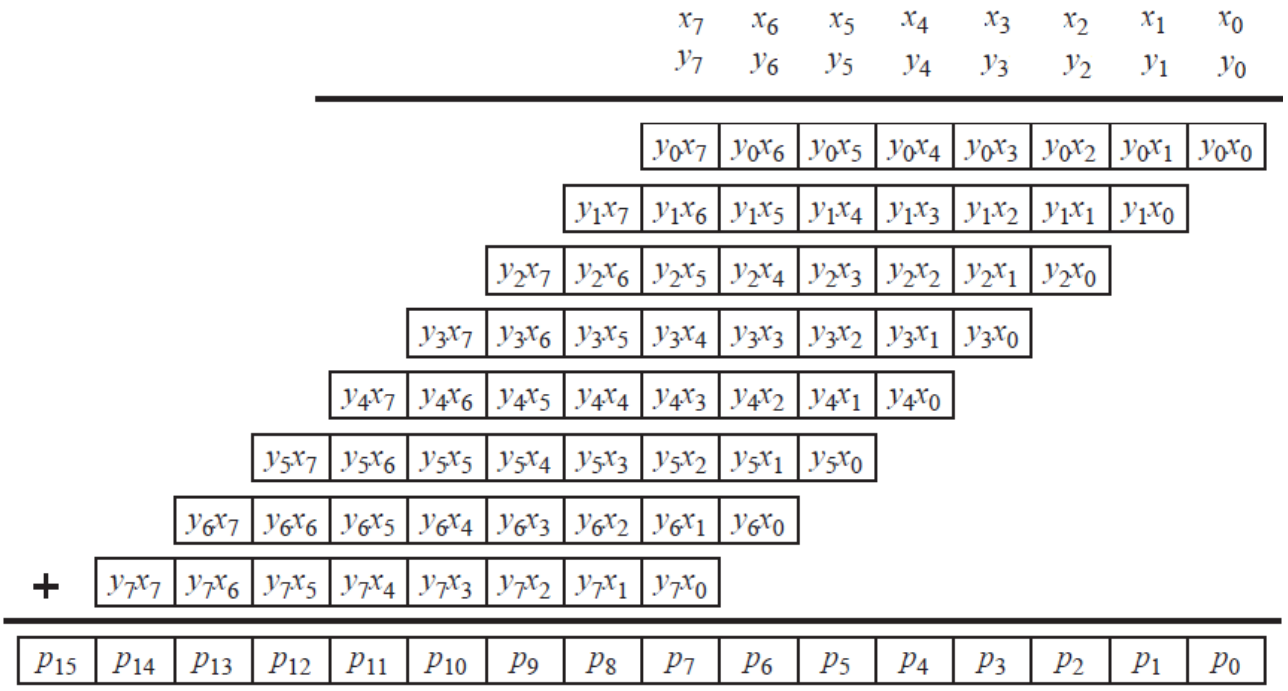


Figure 10: Explaination of combinational multiplier [3]

In this project, only the integers are involved in the addition. But the matrix coefficients for the multiplication in the affine transform are 2s complement signed fixedpoint fractions, which have the radix point positioned after the most significant bits. Therefore the fifteenth and eighth bits of the multiplication result were extracted in order to represent the correct integer of the calculation, and the fraction part was discarded entirely. The weight of the leading bit of the truncated representation is considered as -2^7 so both positive and negative number can be correctly represented.

The testbench of ALU is shown in listing 8. ALU opcode is shown in listing 7.

```
// alu function bits
`define Fsize  2

`define RA   2'b00
`define RB   2'b01
`define RADD 2'b10
`define RMUX 2'b11
```

Listing 7: alucode.sv

```
`include "alucodes.sv"
module alutest;
  parameter n =8; // data bus width
  logic signed[n-1:0] a, b; // ALU input operands
  logic [`Fsize-1:0] func; // ALU function code
  logic [n-1:0] result; // ALU result

alu  #(.n(n)) r(.*);

//------------- code starts here ----------
initial
begin
  a= 8'b11100000;
  b= 8'b00010100;

  // test all functions
  #10 func = `RA;      // result = a
  #10 func = `RB;      // result = b
  #10 func = `RADD;    // result = a+b
  #10 func = `RMUX;    // result = a*b

end
endmodule
```

Listing 8: Arithmetic Logic Unit testbench

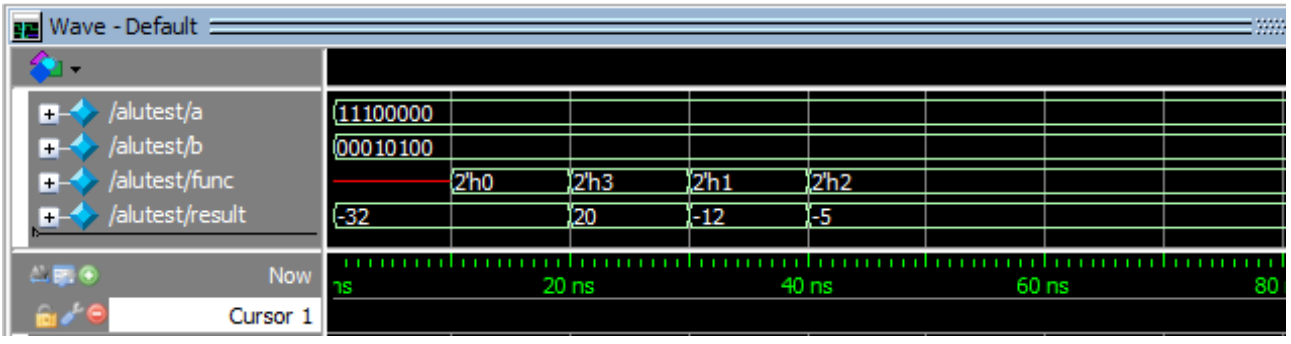


Figure 11: Simulation results of Arithmetic Logic Unit

The testbench simulated Pass-the-ALU, addition and multiplication function with two numbers. Number A was considered as -0.25 for a fractional number or -32 for an integer and B was treated only as 20 for an integer. At the 10ns and 20ns, the ALU was commanded to pass -0.25 and 20 without any calculation. At the 30ns, an addition was executed and -12 was the result of -32 plus 20. At the 40ns, -5 was shown as the result of a multiplication between -0.25 and 20. The manual calculation confirmed the ALU test results were all correct.

The RTL synthesis of ALU is shown in Figure 12. The ALU is quite compact without other redundant arithmetic logics. The multiplier is synthesised as an embedded hardware unit of Altera Cyclone III.

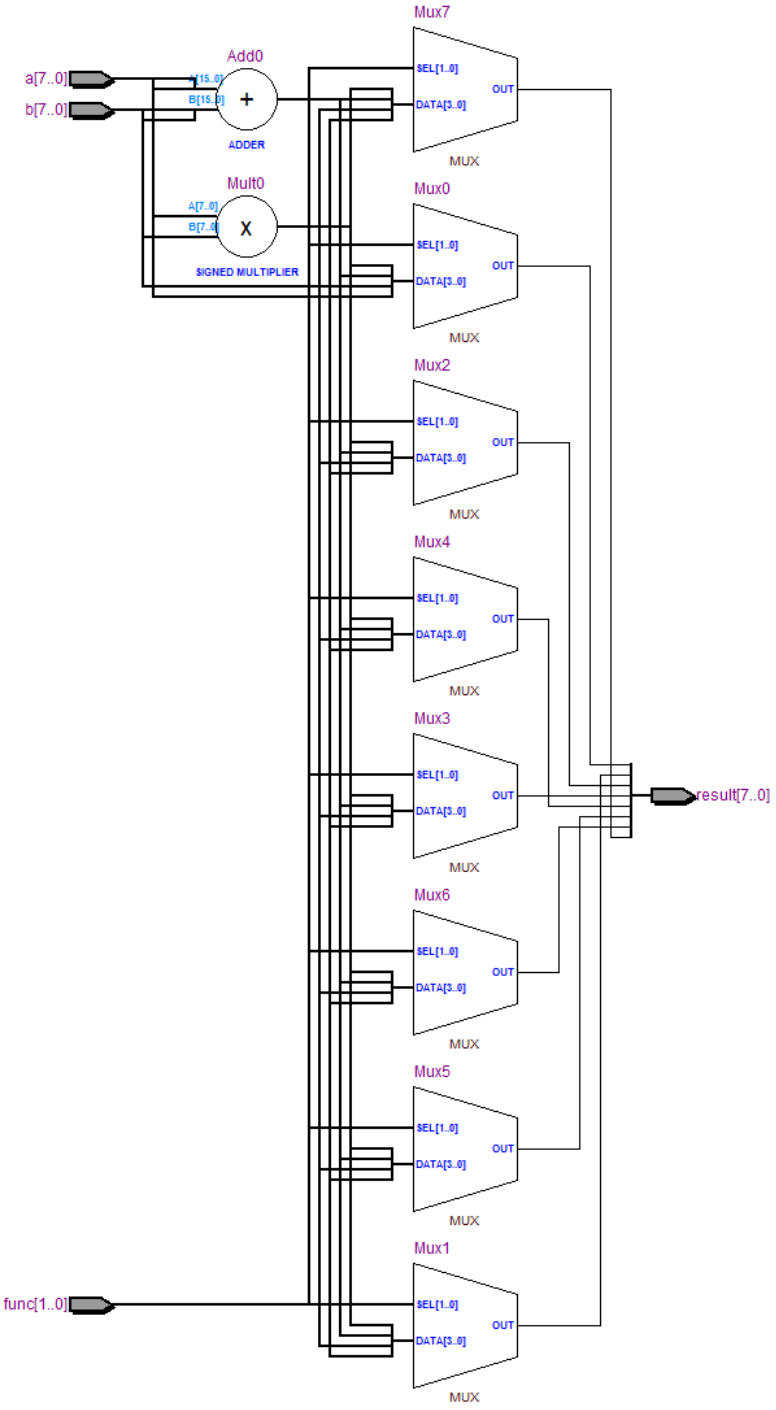


Figure 12: RTL synthesis of Arithmetic Logic Unit

7 Altera DE0 implementation and optimization

7.1 Original design DE0 implementaion

A top level picoMIPS CPU module was implemented to encapsulate all sub-level individual modules. The CPU reads ten switches SW [9:0] from FPGA device, and outputs the results directly to the

LEDs to save some hardware. Additional multiplexer *imm* selects the data between immediate bits and source register output, and the multiplexer *sel* selects switches [7:0] or ALU output. A simple counter [4] was added to slow down the Altera DE0 embedded 50MHz clock to around 10Hz for the demonstration of the design.

The testbench for the complete CPU is shown in listing 9. A pixel coordinates [16 20] were used as the example in the simulation.

```
1  module cputest;
2  parameter n = 8;
3  logic clk;                                  //counter clock
4  logic [9:0] SW;                             //input SW
5  logic[n-1:0] outport;                       //output port (ALU output)
6  logic reset;                                //master reset
7
8  // ALU
9  logic [1:0] ALUfunc;                        // ALU function
10 logic imm;                                  // immediate operand signal
11 logic [n-1:0] Alub;                         // output from imm MUX
12
13 // registers
14 logic [n-1:0] Rd, Rs, WdataALU, Wdata;      // Register data
15 logic w;                                    // register write control
16
17 // Program Counter
18 parameter Psize = 5;                        // up to 32 instructions
19 logic PCincr,PCabsbranch;                   // program counter control
20 logic [Psize-1 : 0]ProgAddress;             // program memory address
21
22 // Program Memory
23 parameter Isize = n+10;                     // Isize - instruction width
24 logic [Isize-1:0] I;                        // I - instruction code (hex)
25
26 cpu2  #(.n(n)) mydesign(.*);
27
28 //------------- code starts here ---------
29 initial
30 begin
31   clk =  0;
32   #5ns  forever #5ns clk = ~clk;
33 end
34
35 initial
36 begin
37   SW[9:0] = 10'b1000000000;
38   #50ns SW[9] = 0;
39   #50ns SW[7:0] = 8'b00010000;              //x1
40   #50ns SW[8] = 1;
41   #50ns SW[8] = 0;
42   #50ns SW[7:0] = 8'b00010100;              //y1
43   #50ns SW[8] = 1;
44   #50ns SW[8] = 0;
45   #50ns SW[8] = 1;
46   #50ns SW[8] = 0;                          //stop the processor to start a new
       calculation
47 end
48 endmodule
```

Listing 9: Testbench for CPU

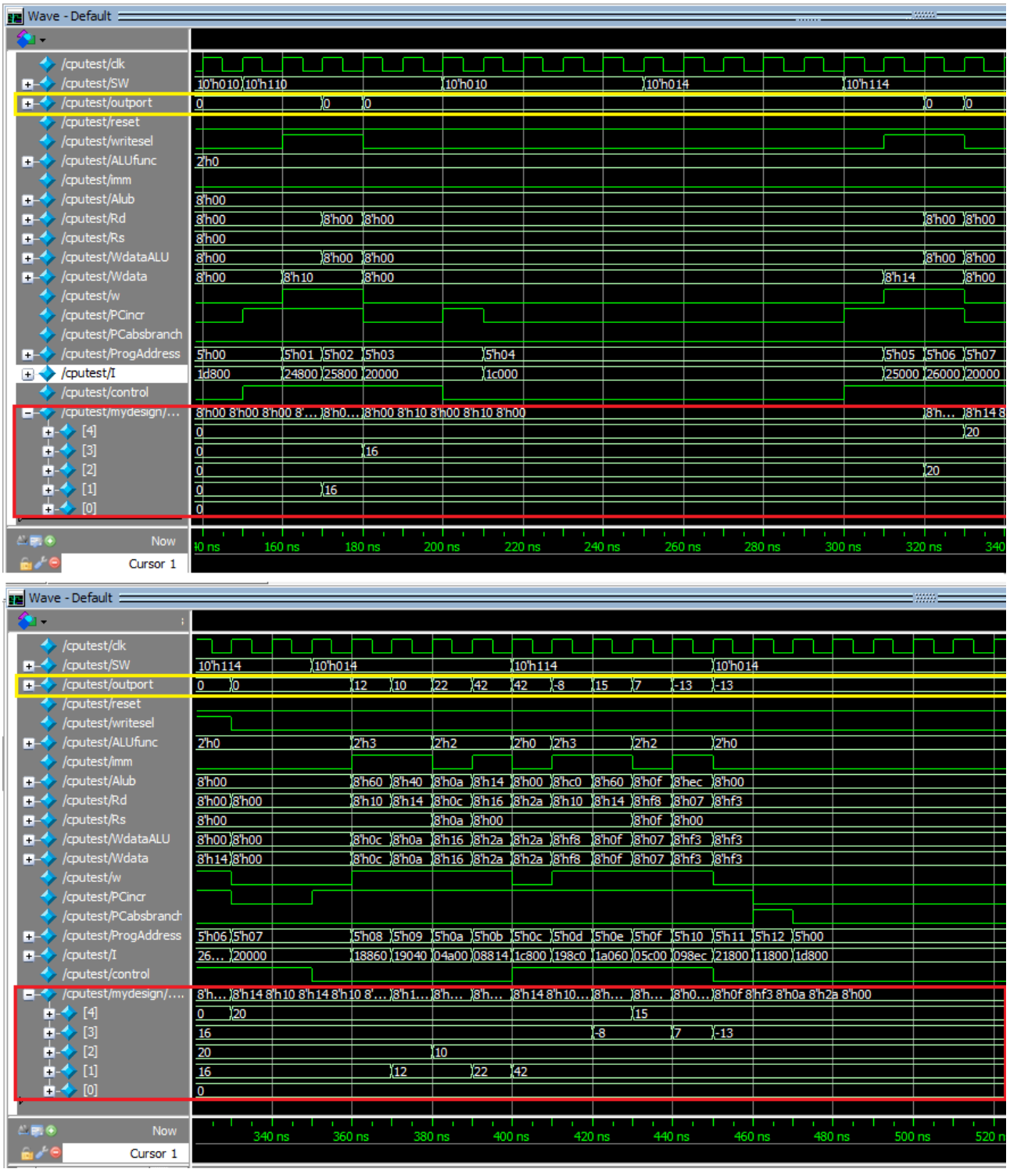


Figure 13: Simulation result of original CPU. (register values in the red box, output in the yellow box)

As indicated in Figure 13, the CPU successfully completed the affine transform. In details, at 150ns and 300ns, the WAIT instructions paused the operation until the required switch signal *control (SW[8])* has appeared. The coordinates were correctly loaded into the register with LOAD instruction at 160ns and 310ns. The transform started from 360ns. It was executed by a series of addition and multiplication and output result was checked correct with manual calculation (at 390ns and 440ns).

The CPU module was synthesised by software *Quartus II*. It was then programmed into chip EP3C16F484C6 (Cyclone III) on FPGA through the Quartus II programmer. The CPU modules was also re-synthesised using Cyclone IV E as the target to obtain the cost figure. For the FPGA test, the test coordinates were exactly the same as in the testbench. After the algorithm, the LEDs outputted the same results from the testbench, as shown in Figure 14.

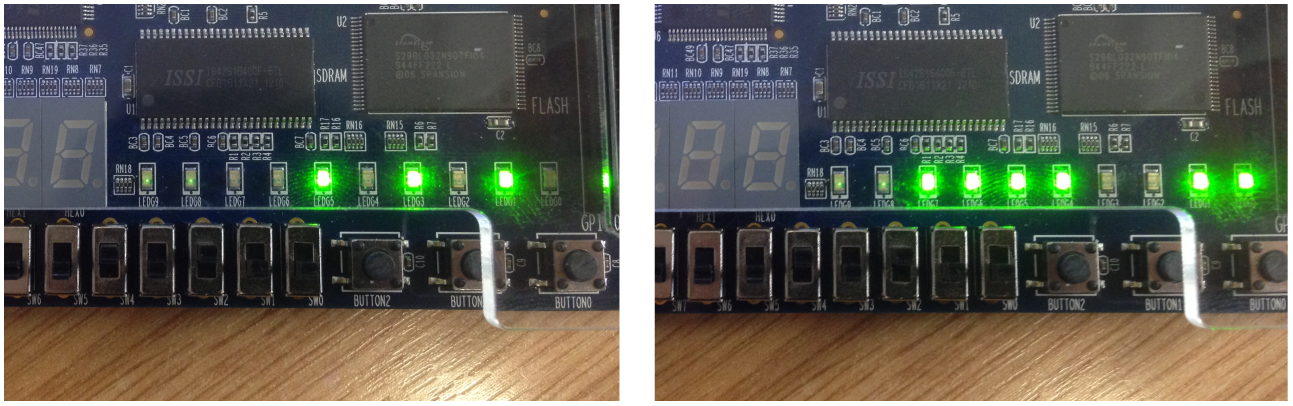


Figure 14: *Left*: x_2 = 42 (00101010) *Right*: y_2 = -13 (11110011)

Although the original picoMIPS processor successfully performed the affine transform, the cost figure was unsatisfying high (Figure 15). The general purpose register occupied the most logic units in the design, and long instruction length also enlarge the program counter and memory. Thus they were the major targets at the optimization stage.

Analysis & Synthesis Resource Utilization by Entity

	Compilation Hierarchy Node	LC Combinationals	LC Registers	Memory Bits	DSP Elements
1	◢ |picoMIPS4test	156 (0)	69 (0)	0	1
1	|counter:c|	24 (24)	24 (24)	0	0
2	◢ |cpu:mydesign|	132 (26)	45 (0)	0	1
1	◢ |alu:iu|	24 (24)	0 (0)	0	1
1	◢ |lpm_mult:Mult0|	0 (0)	0 (0)	0	1
1	|mult_73t:auto_generated|	0 (0)	0 (0)	0	1
2	|decoder:D|	9 (9)	0 (0)	0	0
3	|pc:progCounter|	15 (15)	5 (5)	0	0
4	|prog:progMemory|	28 (28)	0 (0)	0	0
5	|regs:gpr|	30 (30)	40 (40)	0	0

Figure 15: synthesis statistics of the original CPU

7.2 General purpose register optimization

Using the embedded memory in the FPGA could significantly reduce the usage of combinational logic in the register. However, it is synchronous. Therefore the asynchronous read was replaced by synchronous one in the systemverliog file to describe new register. In total there were ten registers declared, although only four registers were used in the affine transform. Otherwise the synthesis tool would prefer to use combinational logics rather than the embedded memory to build the desgin.

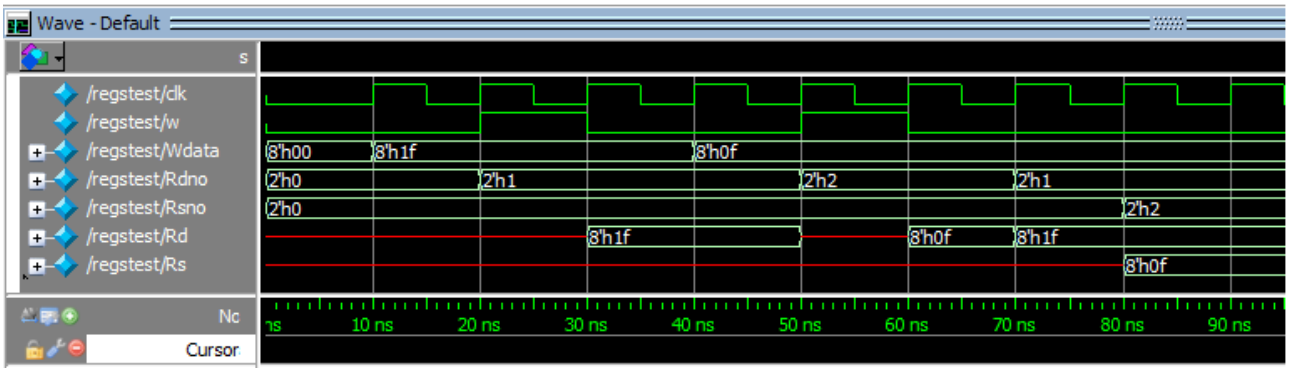


Figure 16: Simulation results of the new General Purpose Register

The testbench for the new register is identical to the original one. However the simulation result from Figure 16 indicates that the register requires two clock cycles to output the newly written data due to the synchronous read (at 20nns and 50ns). During the affine transform, this behaviour will cause correct calculation result from ALU to appear only after the second clock cycle. As the result, the new register was synthesised as one embedded 32 bits memory block with no extra logic units used, as shown in Diagram 17.

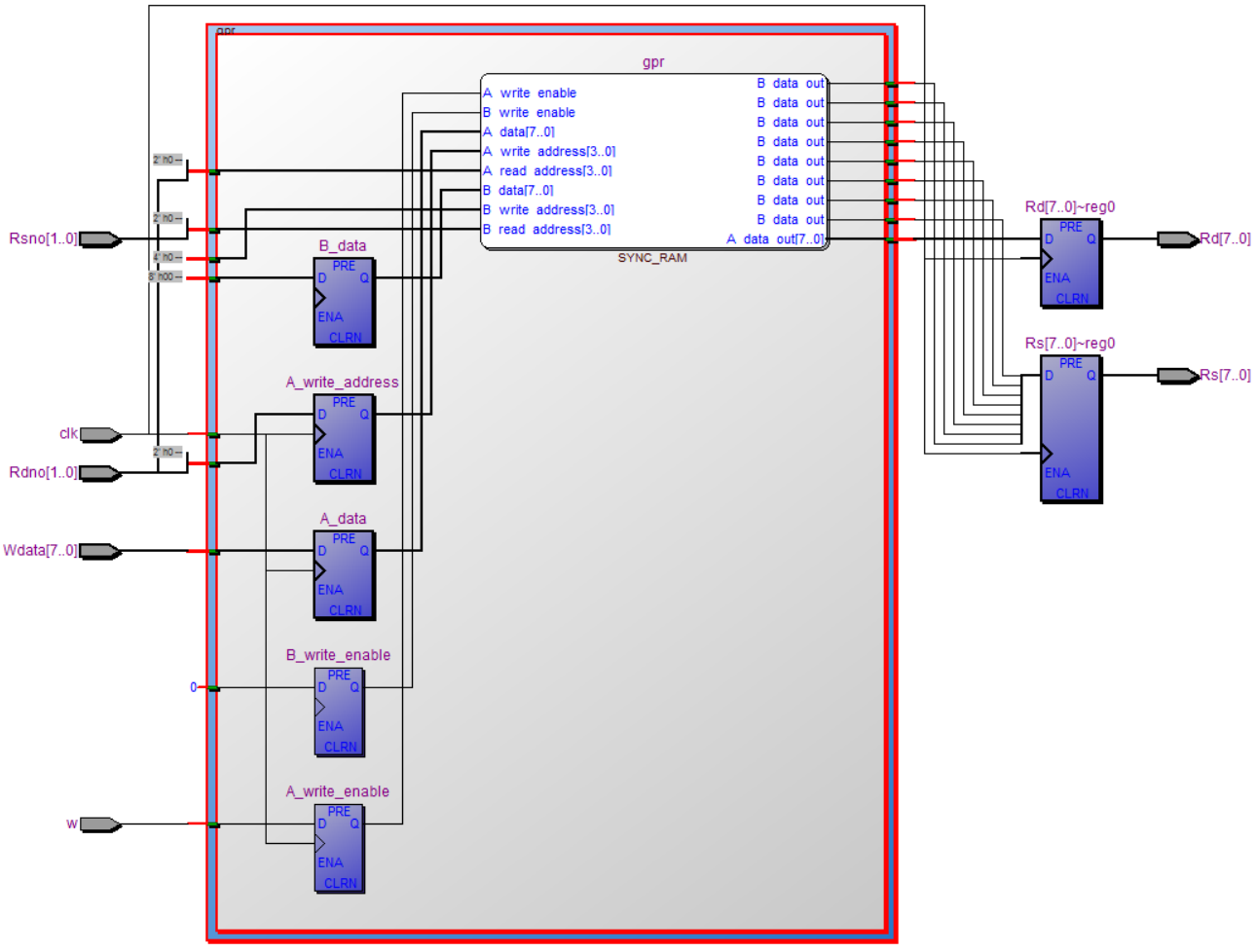


Figure 17: RTL synthesis of the new General Purpose Register

7.3 Decoder optimization

A two-stage state machine (listing 10) was added to the original decoder to address the problem caused by the new register.

```
module decoder
.....
always_ff @ (posedge clk or posedge reset)
begin
  if (reset)
      state = `FETCH;
  else
  begin
    case(state)
      `FETCH: state <= `EXECUTE;
      `EXECUTE: state <= `FETCH;
    endcase
  end
end
.....

endmodule
```

Listing 10: state machine part of the new decoder code

The write-register signal will rise only at *EXECUTE* state to ensure the register could store the correct data, as shown in the Figure 18 at 150ns. The PC increment signal will also rise at *EXECUTE* state to avoid the clock cycle mismatch when a *WAIT* instruction moves to the next one, which may happens at 180ns. Additional signal *selfirst* was implemented to select 4bits instruction immediate as the upper part to form the complete 8bits immediate value because of the new instruction format. Otherwise the 4bits immediate will be used as the lower part. It only rises to high in instruction *ADDIF* and *MUXI* (at 220ns and 260ns).

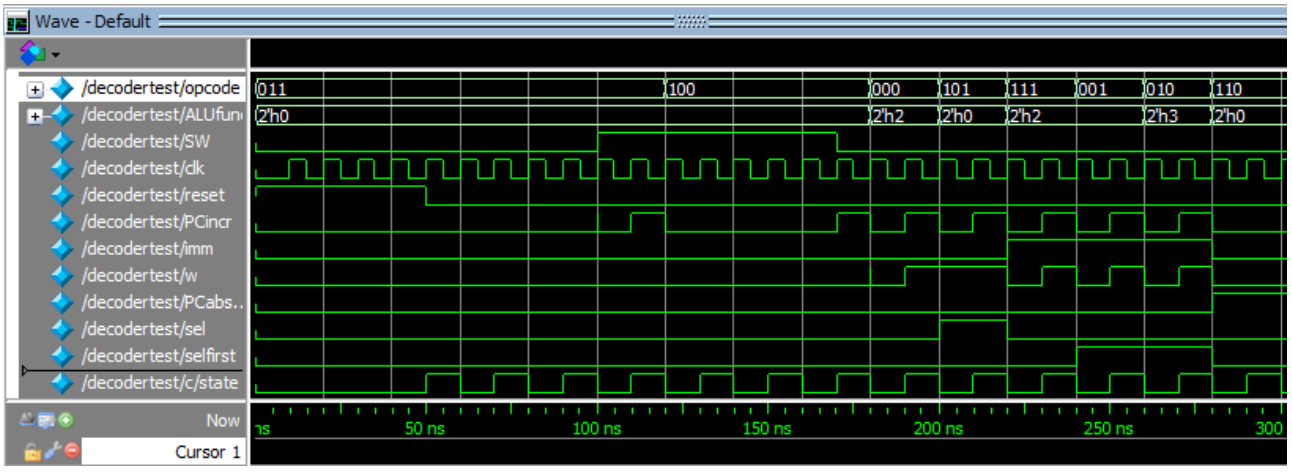


Figure 18: Simulation results of the new Decoder

The testbench for the new decoder is shown in listing 11.

```
`include "alucodes1.sv"
`include "opcodes.sv"
`include "states.sv"

```

```
5  module decodertest;
6
7  logic [2:0] opcode;
8  logic [1:0] ALUfunc;
9  logic SW, clk, reset;
10 logic PCincr, imm, w, PCabsbranch, sel, selfirst;
11
12 decoder c(.*);
13 //---------- code starts here ----------
14 initial
15 begin
16   clk =  0;
17   #5ns  forever #5ns clk = ~clk;
18 end
19 initial
20 begin
21   SW = 0;
22   reset=1;
23   opcode =3'b011;//WAIT0
24   #50ns reset =0;
25   #50ns SW=1;
26   #20ns opcode =3'b100; //WAIT1
27   #50ns SW=0;
28   #10ns opcode =3'b000; //ADD
29   #20ns opcode =3'b101; //LOAD
30   #20ns opcode =3'b111; //ADDIL
31   #20ns opcode =3'b001; //ADDIF
32   #20ns opcode =3'b010; //MUXI
33   #20ns opcode =3'b110; //JUMP
34   end
35 endmodule
```

Listing 11: New Decoder testbench

The RTL synthesis of the new decoder is shown in Figure 19. It has an extra state machine.

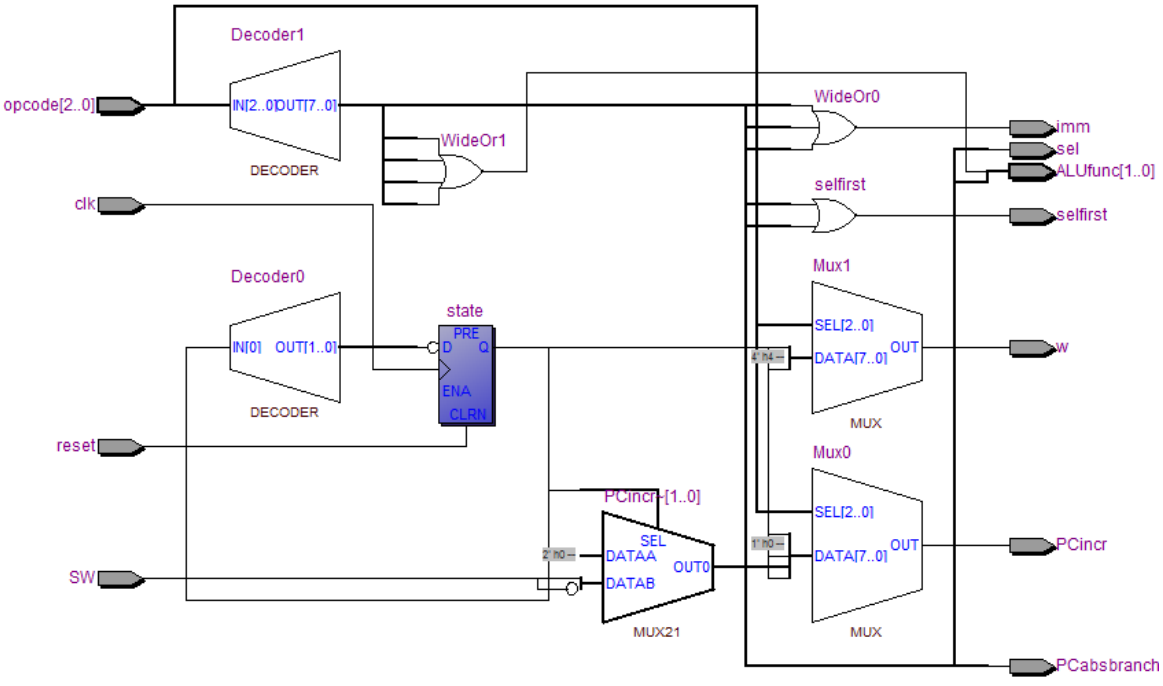


Figure 19: RTL synthesis of the new Decoder

7.4 Instruction optimization

Non-used instructions such as NOP and MUX were removed so the opcode could be reduced to 3 bits. The register address bits were reduced to 2 bits since there were only four registers needed. The source register address bits was also shared as the part of the immediate bits when the register was not in use. The immediate was reduced to 4 bits and a new signal *selfirst* was added in the decoder to decide it as the upper or lower 4 bits of the whole immediate value. Accordingly, the instruction *ADDI* was divided into two. Instruction *ADDIL* loads 4bits as lower part of immediate and instruction *ADDIF* uses them as upper part of immediate. The improved instruction now has only 9 bits in total. The modified picoMIPS architecture is shown in Figure 20.

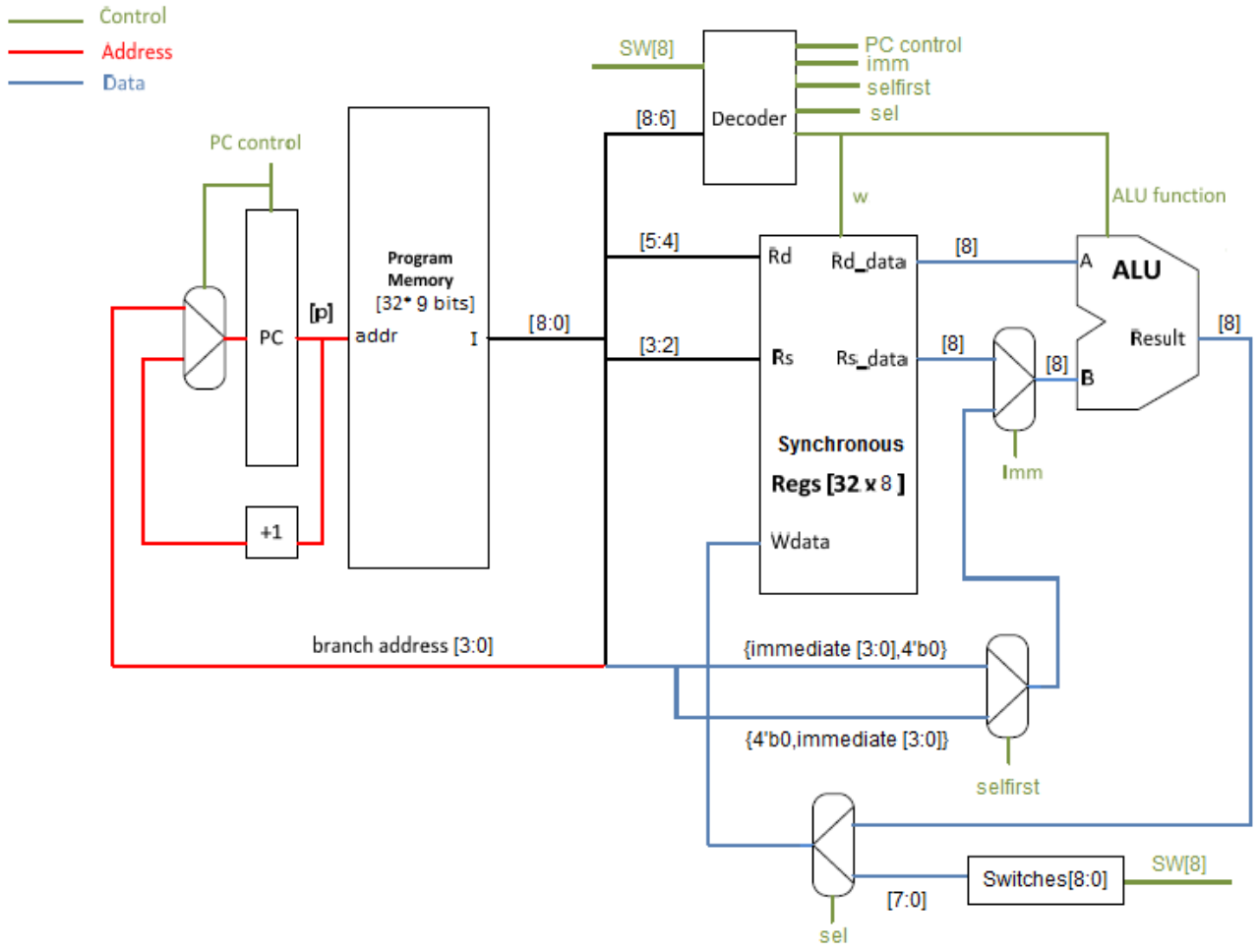


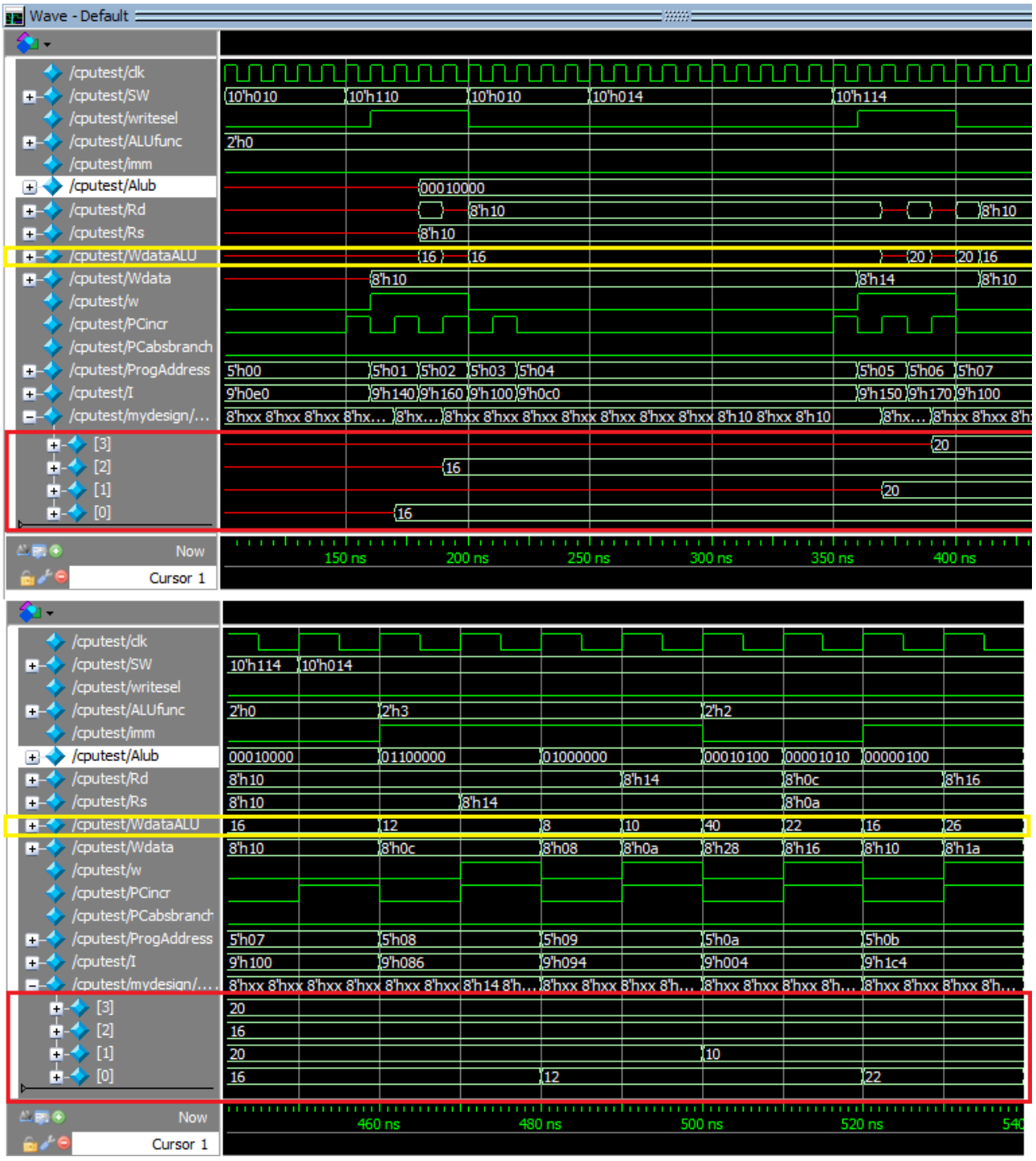
Figure 20: The optimized picoMIPS architecture [2]

7.5 Improved design DE0 implementaion

The simulation of new CPU is shown in Figure 21. The testbench is almost identical to the original one, expect it uses shorter instruction length.

The simulation results illustrates the same transform result as the original CPU. The immediate operand signal Alub shows that immediate value was correctly reformed despite the short immediate bits. The new behaviour of register was also well sorted by the new decoder as designed. The writing

and read procedure were working properly during the affine transform from 450ns to 730ns. Because of the hexadecimal format, the instructions were extended to 12 bits with 0s to avoid the lenght truncation warning by the synthesis tool.



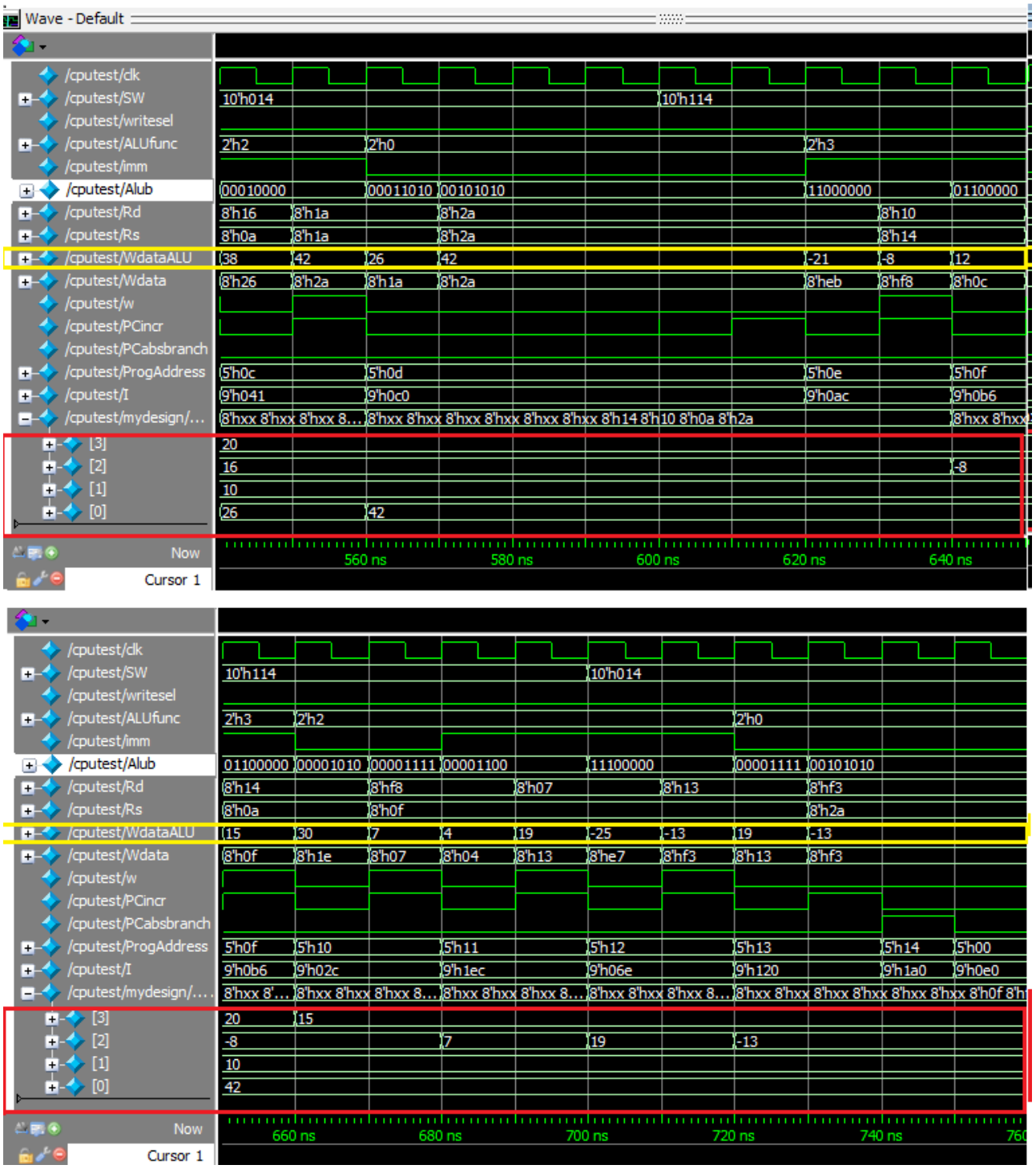


Figure 21: Simulation result of New CPU. (register values in the red box, output in the yellow box, immediate signal Alub is highlighted in white box)

The improved picoMIPS processor generated the same results as the original one on the FPGA, by using the identical pixel coordinates, but the cost figure was significantly reduced (Figure 22).

Analysis & Synthesis Resource Utilization by Entity

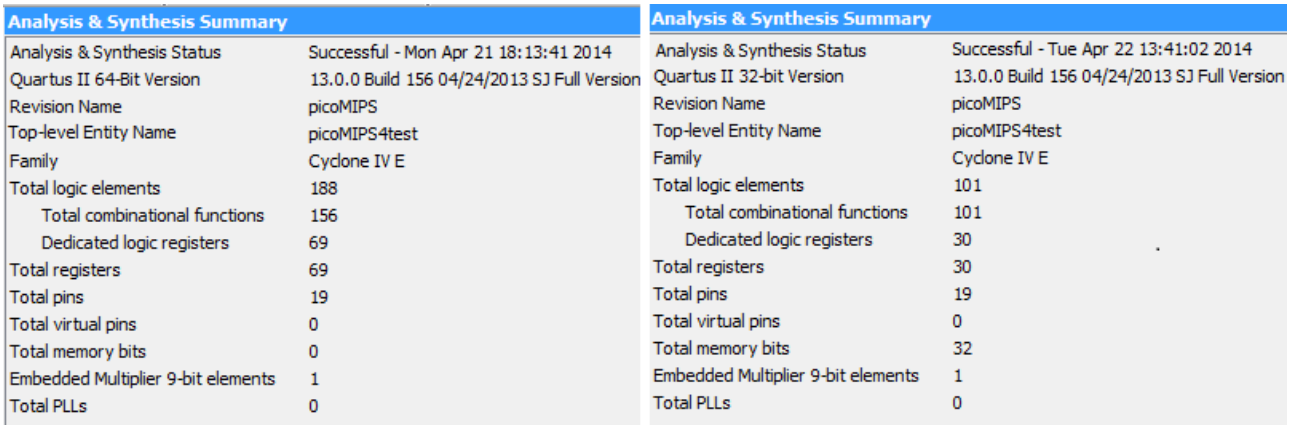
	Compilation Hierarchy Node	LC Combinationals	LC Registers	Memory Bits	DSP Elements	DSP 9x9
1	◢ |picoMIPS4test|	101 (1)	30 (0)	32	1	1
1	|counter:c|	24 (24)	24 (24)	0	0	0
2	◢ |cpu2:mydesign|	76 (16)	6 (0)	32	1	1
1	◢ |alu:iu|	24 (24)	0 (0)	0	1	1
1	◢ |lpm_mult:Mult0|	0 (0)	0 (0)	0	1	1
1	|mult_u1t:auto_generated|	0 (0)	0 (0)	0	1	1
2	|decoder:D|	9 (9)	1 (1)	0	0	0
3	|pc:progCounter|	7 (7)	5 (5)	0	0	0
4	|prog:progMemory|	20 (20)	0 (0)	0	0	0
5	◢ |regs:gpr|	0 (0)	0 (0)	32	0	0
1	◢ |altsyncram:gpr_rtl_0|	0 (0)	0 (0)	32	0	0
1	|altsyncram_58u1:auto_generated|	0 (0)	0 (0)	32	0	0

Figure 22: synthesis statistics of the improved CPU

8 Conclusions

During this project a systemverliog program was successfully implemented on FPGA device to perform the affine transform with designated matrix constants. During the process the picoMIPS architecture was well studied, including the understanding of the function of each module, the usage of the synchronous and asynchronous RAM, top level CPU design, instructions design and so on. Meanwhile, experience was gained in using synthesis tool and programmer Quarter II.

At the second stage of the development, the processor was optimized with usage of synchronous RAM and the cost figure of the system was significantly reduced. The number of logic units used in the original design and improved one are summarized in Figure fig:cpucost.



Analysis & Synthesis Summary	
Analysis & Synthesis Status	Successful - Mon Apr 21 18:13:41 2014
Quartus II 64-Bit Version	13.0.0 Build 156 04/24/2013 SJ Full Version
Revision Name	picoMIPS
Top-level Entity Name	picoMIPS4test
Family	Cyclone IV E
Total logic elements	188
Total combinational functions	156
Dedicated logic registers	69
Total registers	69
Total pins	19
Total virtual pins	0
Total memory bits	0
Embedded Multiplier 9-bit elements	1
Total PLLs	0

Analysis & Synthesis Summary	
Analysis & Synthesis Status	Successful - Tue Apr 22 13:41:02 2014
Quartus II 32-bit Version	13.0.0 Build 156 04/24/2013 SJ Full Version
Revision Name	picoMIPS
Top-level Entity Name	picoMIPS4test
Family	Cyclone IV E
Total logic elements	101
Total combinational functions	101
Dedicated logic registers	30
Total registers	30
Total pins	19
Total virtual pins	0
Total memory bits	32
Embedded Multiplier 9-bit elements	1
Total PLLs	0

Figure 23: Logic elements usage of original and improved CPU, including slow clock counter. (*Left*: Origianl CPU *Right*: Improved CPU)

The cost figure of original design is $132 + 30 \times 0 = 132$.
The improved one is $76 + 30 \times 0.03125 \approx 76.94$.
The percentage of reduction is 42%.

The future work could focus on reducing the instructions in the program memory in the case that the affine transform will be combined within other operations, even though this improvement may increase the complexity of other modules. To reduce the size of ALU, the addition operation could be divided out of it to save a 8bits multiplexer. Same operation could be done by an accumulator added after the ALU, at the cost that adding one more clock cycle delay to the already slow system and changing the whole architecture dramatically. An alternative approach would be introducing the pipeline to improve the calculation efficiency.

References

[1] e. Hazewinkel, Michiel. (2001) Affine transformation. http://www.encyclopediaofmath.org/index.php/Affine_transformation.

[2] T. Kazmierski. (2014) Simple decoder and cpu test. https://secure.ecs.soton.ac.uk/notes/elec6016/tjk1314/5cpu1.pdf.

[3] J. Tidd and J. Bessant, *Digital Design Principles and Practices,The fourth edition.* Prentice Hall division of Pearson Education, ISBN 0-13-186389-4, 2013.

[4] T. Kazmierski. Clock divider: counter.sv. https://secure.ecs.soton.ac.uk/notes/elec6016/tjk1314/assignment/counter.sv. Accessed: 2014-04-28.