ELEC6234 – Embedded Processor Synthesis

Karthik Sathyanarayanan
Ks6n19
MSc Embedded systems
Tom Kasmierski

ABSTRACT: *The project aims at making a picoMIPS architecture based embedded processor on FPGA along with a program to achieve affine transform. The individual modules of the processor are tested in ModelSim and the hardware is then synthesised in Quartus prime Lite and it successfully synthesises as imagined.*

1. 1. Introduction

The objective of the assignment was to produce a design that is to produce a picoMIPS architecture n bit implementation on FPGA DE-1 SoC.. The input is 8 bit 2s compliments form of data. The data should go through an affine transform by the picoMIPS design. The affine transformation is a transformation that preserves co linearity. This means that the points will still lie on the same line as it was before after undergoing transformation.
It can be expressed by the equation shown below.

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

Here [x1, y1] are the coordinates of the pixel before the transform and [x2, y2] are the coordinates after the transformation We use the following dataset values to implement the transform.

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

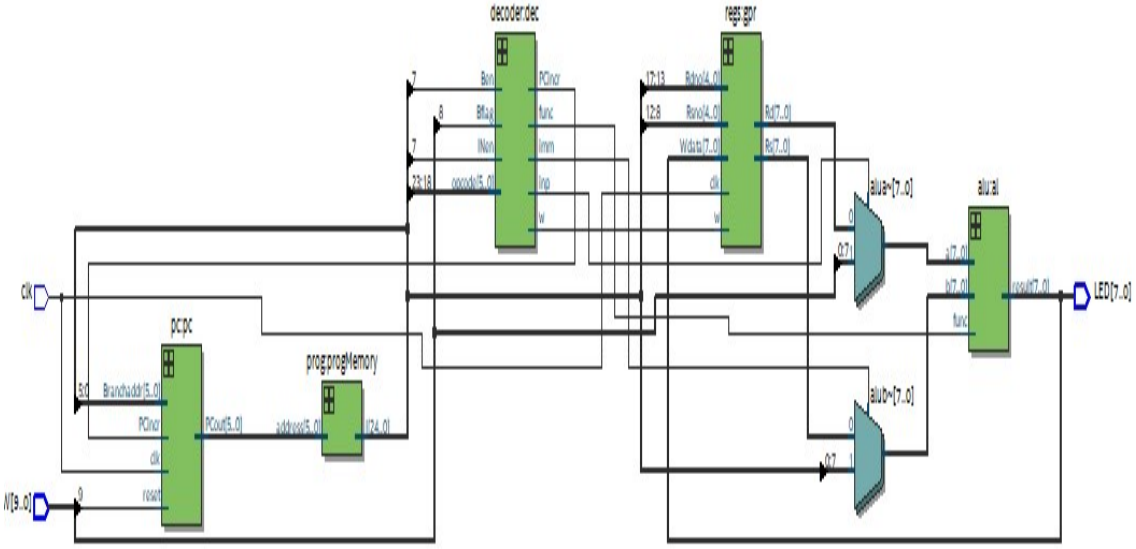
When we open the equation, we can can arrive at the following equation:

$$\{x2 = a_{11} * x_1 + a_{12} * y_1 + b_1\}$$
$$\{y2 = a_{21} * x_1 + a_{22} * y_1 + b_2\}$$

As we can see from the above equation, we need 4 multiplication operations and 4 addition operations. Thus, we use only add and multiply operation in the ALU. Thus, we use only 2-bit alucodes for ADD and MUL. We use the embedded multipliers in the dsp block to implement the multiplication operation. The picoMIPS architecture is more than enough to implement the affine transformation. We use a 8 bit data width for the picoMIPS and 24 bit instruction.

1. 2. Overall architecture of the design

The architecture uses a program counter, a general-purpose register, a 24 bit instruction decoder and a

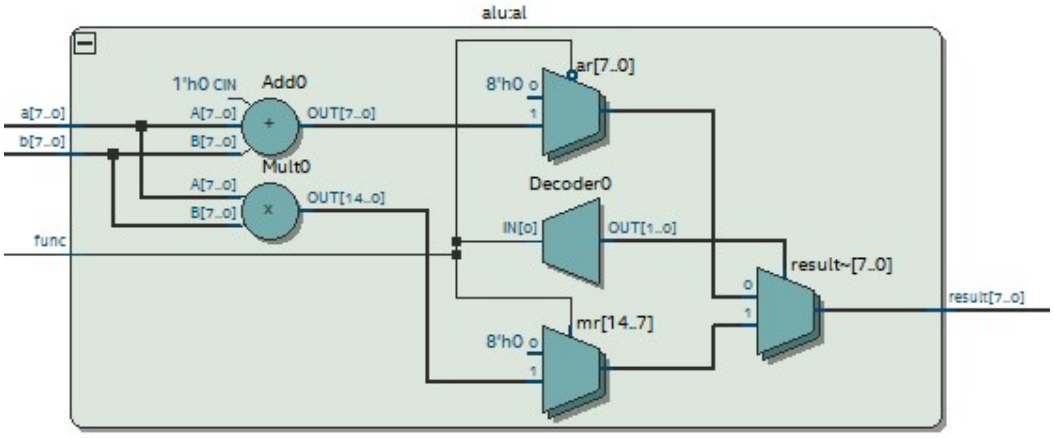


program memory. These blocks are connected with the other blocks as shown in the RTL diagram shown above. The input switches carrying the coordinate data are connected to the ALU block via multiplexers.

3. Details of hardware blocks

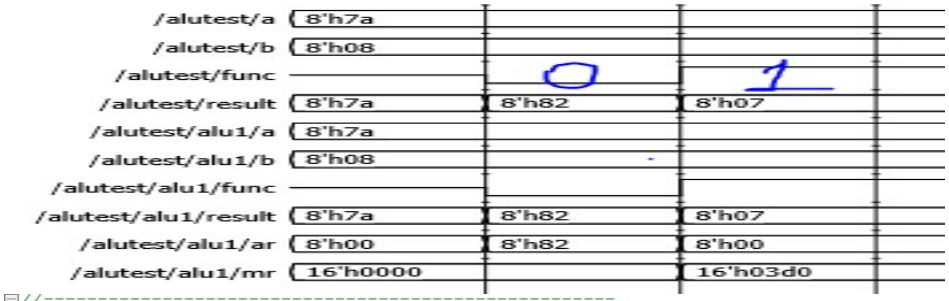
ALU:

The arithmetic logical unit uses one adder and one multiplier to implement the ALU instructions. The input func inputs the alucode to decide which ALU operation to implement. The ALU is also used to input in the values of the data from the switches and store it in the general-purpose registers.



The design uses a 9x9 signed multiplier to implement the multiplication and uses an 8-bit adder to implement the addition. The multiplicand is 2s compliment integer and we truncate the 16-bit output and extract the bits only between 14 to 7. This is because one of the multiplicand is a fraction and another is integer and we do not want the fraction part when displaying the result. A better mechanism could be rounding off the value to avoid corner

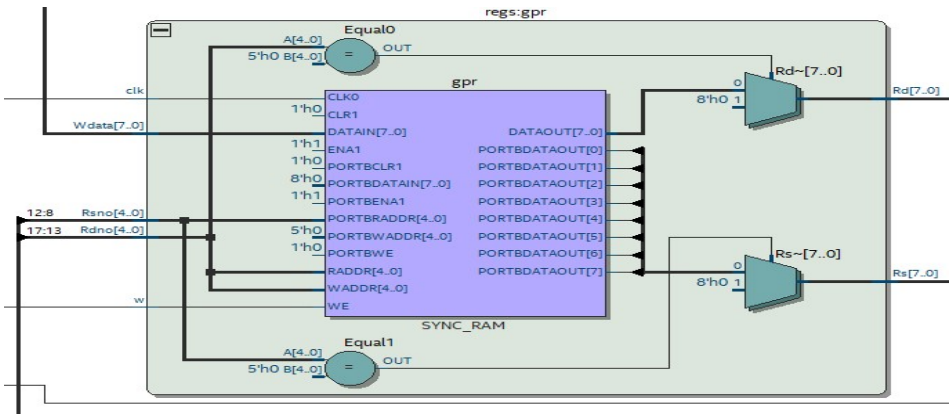
and edge cases during the affine transformation. The signal func comes from the decoder and assigns which part of the ALU is to be used.
ModelSim Simulations:




```
//-----------------------------------------------------
// File Name   : alutest.sv
// Function    : ALUtest module for picoMIPS
// Version: 1,  only 2 funcs
// Author:  ks6nl9
// Last rev. 11-05-20
//-----------------------------------------------------

`include "alucodes.sv"
module alutest;
parameter n =8; // data bus width
logic [n-1:0] a, b; // ALU input operands
logic  func; // ALU func code
logic [n-1:0] result; // ALU result
alu #(.n(n)) alu1 (.a(a), .b(b), .func(func), .result(result) );  // create alu object
//------------- code starts here ---------
    initial
        begin  a= 8'h7a;  b= 8'h08;        // test all funcRons
        #10 func = `RADD; // result = a+b
        #10 func = `RMUL ; // result = a*b

        end //initial
endmodule //end of alutest
```

The figure above shows the modelsim simulation of the ALU module as well as the code for the testbench. In the second clock cycle, we put the function code as 1'b0 which triggers the ALU code for 'RADD and we see that the result is available in the result port which is the summation of operands a and b.
In the second clock cycle, we put the function code as 1'b1 which triggers the Alu code for 'RMUL which does the multiplication operation on the operands 'a' and 'b'. The variable ar and mr are temporary variables which aid in the ALU operations for addition and multiplication respectively.

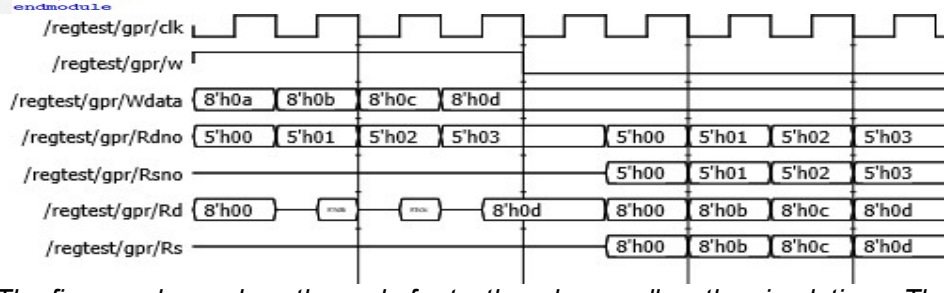
General purpose Regs:
The general purpose registers are used to store the intermediate values from the ALU during the affine transformation operation. The registers addresses are 5 bit wide and store data from the ALU which are 8 bit wide. We initialise 32 gpr registers having 5 bit address width. A register control signal 'w' is used to control which register is being used at a instance. The gpr registers are synthesised as synchronous RAM in quartus with some multiplexers and buffers to output the data to the ALU.



The figure above shows the RTL diagram of the registers showing the various blocks such

as multiplexers, buffers and port connections for the synchronous RAM.
ModelSim Simulations:

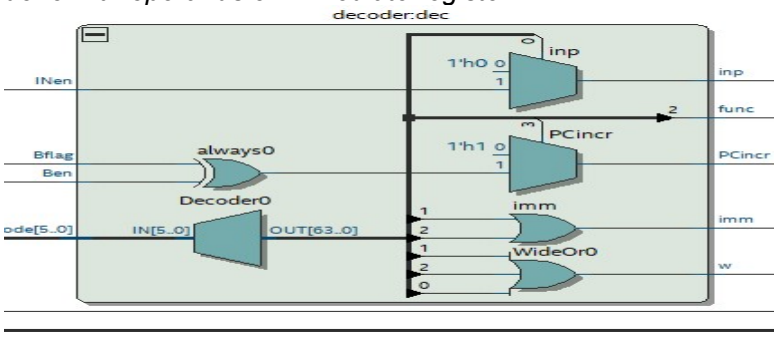
```
//-----------------------------------------------------
// File Name    : regtest.sv
// Function     : gpr register testbench for picoMIPS
// Version: 1,   values from %0, %1, %2, %3,
// Author:  ks6n19
// Last rev. 11-05-20
//-----------------------------------------------------
module regtest ;
parameter n = 8 ;
logic clk, w ;
logic [n-1:0] Wdata ;
logic [4:0] Rsno, Rdno ;
logic [n-1:0] Rs , Rd ;

regs #(.n(8)) gpr (.clk(clk),.w(w),
                   .Wdata(Wdata), // write ALU result to Rd
                   .Rdno(Rdno),  // reg %d number
                   .Rsno(Rsno),  // reg %s number
                   .Rd(Rd),.Rs(Rs));


    initial
        begin
            clk = 0 ;
            forever #5ns clk = ~clk ;
            end

    initial
        begin
            w = 1 ;
            Rdno = 5'b00000; Wdata = 8'd10 ;
        #10 Rdno = 5'b00001; Wdata = 8'd11 ;
        #10 Rdno = 5'b00010; Wdata = 8'd12 ;
        #10 Rdno = 5'b00011; Wdata = 8'd13 ;
        #10 w = 0 ;
        #10 Rdno = 5'b00000; Rsno = 5'b00000;
        #10 Rdno = 5'b00001; Rsno = 5'b00001;
        #10 Rdno = 5'b00010; Rsno = 5'b00010;
        #10 Rdno = 5'b00011; Rsno = 5'b00011;
        end
endmodule
```

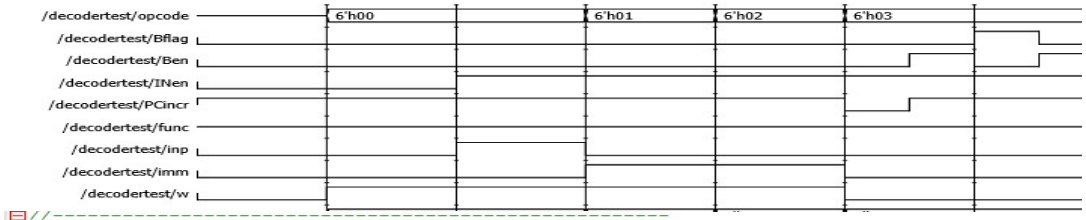


The figures above show the code for testbench as well as the simulations. The write control is intitally set to 1 and the Wdata which comes in from the ALU output is assigned values of hexadecimal 0a, 0b,0c and 0d. to the registers %0, %1, %2, %3. The register %0 is permanently given the value of 0 and is reflected in the output Rd and Rs. When w = 1, The data from Wdata gets stored in the registers %0, %1, %2, %3 as given by Rdno. When the w is set to 0 , the data that is stored in the gpr is available at the ouput Rd. We can also see he working of the zero register in the diagram where instead of the inputted value 0a, it gives out 0.

Decoder:

Decoder are used to decode the 6 bit opcode instructions from the 24 bit data coming from the program counter. The Bflag is connected to the 8th switch while the Ben and INen is connected to the 7th switch. When the 7th bit is 1 , the inp is 1 and INen is 1 signalling the multiplexer at ALU to read from the input port. When BFlag is equal to Ben, the Pc increment is reset to 0 or else the PCincr increments by default. When the opcode is ADDI, imm is set to 1 and the operand is read from immediate register. Similarly at MULI , the multiplication is done with operands of immediate register.





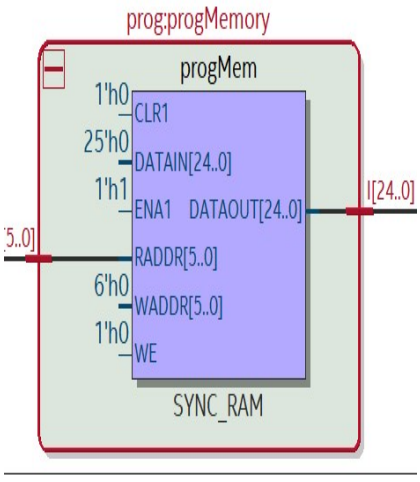

```
//-------------------------------------------------------
// File Name    : decodertest.sv
// Function     : decoder testbench for picoMIPS
// Version: 1,  'ADD, 'ADDI , 'MULI, 'B funcs and Bflags.
// Author:  ks6n19
// Last rev. 11-05-20
//-------------------------------------------------------
module decodertest ;
logic [5:0] opcode ;
logic Bflag, Ben, INen ;
logic PCincr, func, inp , imm, w ;
decoder D(.opcode(opcode),
          .Bflag(Bflag),
          .Ben(Ben),
          .INen(INen),
          .PCincr(PCincr),
          .inp(inp),
          .imm(imm),
          .w(w));

initial
    begin
        Bflag = '0; Ben = '0; INen = '0 ;
        #10 opcode = 6'b000000 ; //add
        #10 INen = '1 ;
        #10 opcode = 6'b000001 ; //addi
        #10 opcode = 6'b000010 ; //muli
        #10 opcode = 6'b000011 ; //B
        #5 Ben = '1 ; Bflag = '0 ;
        #5 Ben = '0 ; Bflag = '1 ;
        #5 Ben = '1 ; Bflag = '0 ;
    end
endmodule
```

The figures above show the modelsim simulation of decoder. The opcode for ADD, ADDI, MULI and B are tested in consecutive clock cycles. At opcode 00, the INen and inp are set to 1 and the imm is set to 0 doing the default case of addition. At opode 01, inp is set to 0 and the imm is set to 1 signalling the addition from the immediate register. Similarly for MULI, the operand is read from the immefdiate register and not from the input port of the mux. The branch operation is not shown properly as the testbench makes the cases such that Bflag is never equal to Ben is the same clock cycle, therefore the PCincr is never set to 0 and keeps on incrementing.

Progmem:

The program memory has input giving the opcodes and ouput as the 24 bit instruction for the decoder to extract the 6 bit opcode form it. It uses a prog.hex file to read the address from and is synthesised as a synchronous RAM.

```
// sample picoMIPS program 1
// n = 8 bits, Isize = 16+n = 24 bits
// format: 6b opcode, 5b %d, 5b %s, 8b immediate or address
//
// HEX ////////// BINARY  ////////////////////// ASSEMBLER /////////
080000    // 24'b000010_00000_00000_00000000; //MULI 0 //CLEAR %0
0C0001    // 24'b000011_00000_00000_00000001; //B 0 PC=1
000080    // 24'b000000_00000_00000_10000000; //ADD  %0; %0 = inport x1
002080    // 24'b000000_00001_00000_10000000; //ADD  %1; %1 = inport x1
0C0084    // 24'b000011_00000_00000_10000100; //B 1 PC=4
0C0005    // 24'b000011_00000_00000_00000101; //B 0 PC=5
004080    // 24'b000000_00010_00000_10000000; //ADD  %2; %2 = inport y1
006080    // 24'b000000_00011_00000_10000000; //ADD  %3; %3 = inport y1
0C0088    // 24'b000011_00000_00000_10001000; //B 1 PC=8
080060    // 24'b000010_00000_00000_01100000; //MULI %0, %0, 0.75; %0 = %0 * 0.75 // 0.75x1
0821C0    // 24'b000010_00001_00001_11000000; //MULI %1, %1, -0.5; %1 = %1 * -0.5 // -0.5x1
084240    // 24'b000010_00010_00010_01000000; //MULI %2, %2, 0.5; %2 = %2 * 0.5 // 0.5y1
086360    // 24'b000010_00011_00011_01100000; //MULI %3, %3, 0.5; %3 = %3 * 0.75 // 0.75y1
000200    // 24'b000000_00000_00010_00000000; //ADD %0, %2; %0 = %0 + %2 // 0.75x1 + 0.5y1
006100    // 24'b000000_00011_00001_00000000; //ADD %3, %1; %3 = %3 + %1 // 0.75y1 - 0.5x1
040014    // 24'b000001_00000_00000_00010100; //ADDI %0, 20; //%0 = %0 + 20 //x2 = 0.75x1 + 0.5y1 + 20
0C0010    // 24'b000011_00000_00000_00010000; //B 0 PC = 16
080000    // 24'b000010_00000_00000_00000000; //MULI 0 //CLEAR %0
0460EC    // 24'b000001_00011_00000_11101100; //ADDI %3, -20; //%3 = %3 + -20 //y2 = 0.75x1 + 0.5y1 + 20
000300    // 24'b000000_00000_00011_00000000; //ADD %0, %3; %0 = %0 + %3 //y2 = 0.75x1 + 0.5y1 + 20
0400EC    // 24'b000001_00000_00000_11101100; //ADDI %0, -20; //%0 = %0 - 20 //DISP
0C0094    // 24'b000011_00000_00000_10010100; //B 1 PC = 20
```

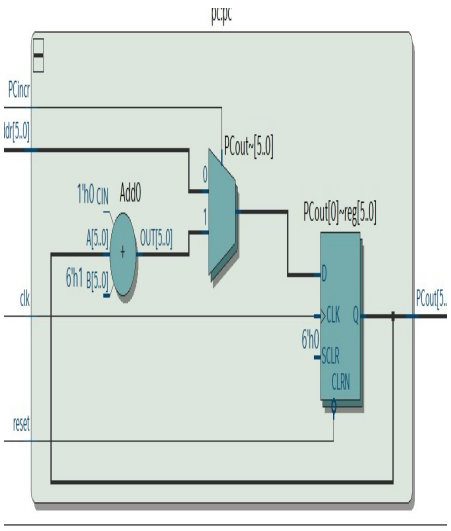


The figures above show the RTL diagram as well as the prog.hex file. The program first clears the data in %0 register, then starts the PCincrement. The program then stores the inputs from the ALU by adding the input from switches to 0. The operands are then multiplied and stored in registers 0, 1 ,2 and 3. The B opcode acts as a handshaking signal between operations. The values of B(20, -20) are then added and the final values are shown in the last few cycles after an absolute branch operation.

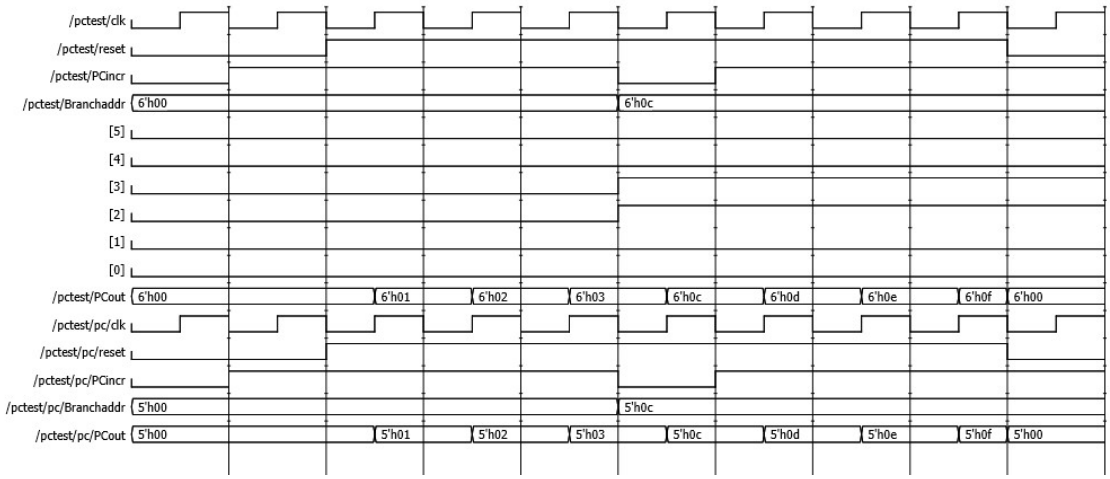
Program Counter:

The program counter is designed to increment the PCincr by default. The figures below show the RTL diagram and the testbench code.

```
//-------------------------------------------------------
// File Name   : pctest.sv
// Function    : program counter testbench for picoMIPS
// Version: 1,  increment or branch
// Author:  ks6n19
// Last rev. 11-05-20
//-------------------------------------------------------
module pctest ;
logic clk ;
logic reset ;
logic PCincr ;
logic [5:0] Branchaddr ;
logic [5:0] PCout ;
    pc #(.Psize(5)) pc (.clk(clk),
                        .reset(reset),
                        .PCincr(PCincr),
                        .Branchaddr(Branchaddr),
                        .PCout(PCout));
initial
    begin
        clk = 1'b0 ;
        forever #5ns clk = ~clk ;
    end
initial
        begin
            reset = '0; PCincr = '0; Branchaddr = '0 ;
            #10 reset = '0 ; PCincr = '1 ;
            #10 reset = '1 ;
            #30 PCincr = '0 ; Branchaddr = 5'b01100 ;
            #10 reset = '1 ; PCincr = '1;
            #30 reset = '0 ;
            #10 reset = '1 ;

        end
endmodule
```



The program counter increments by default unless and until the branchaddr is set high where in it resets the program counter and starts incrementing from the branch address specified. The simulations below show that initially , the Branch address is 0 and the program counter increments from 0 . Then the Branch address is set to 0c , the program counter resets and the counter starts incrementing from 0c , 0d0e and so on.



1. 6. Conclusion

The picoMIPS architecture is successfully synthesised and uses 65 ALMs, 42 registers and 2 synchronous RAMS for prog memory and gpr regs.

Flow Status	Successful - Thu May 14 09:06:04 2020
Quartus Prime Version	19.1.0 Build 670 09/22/2019 SJ Lite Edition
Revision Name	picoMIPS
Top-level Entity Name	picoMIPS
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	65 / 32,070 (< 1 %)
Total registers	42
Total pins	19 / 457 (4 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total DSP Blocks	1 / 87 (1 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0

There are 32 registers that are using synchronous load and 6 using asynchronous load. Overall the cost is pretty low and the utilisation is less than 1% of the resources. A proper picoMIPS program simulation could not be done to test the validity of the program to implement affine transformation. However, each individual blocks were simulated and the port connections were done properly to synthesis the correct architecture. The assignment did give a strong understanding of embedded processors and the various challenges faced when implementing a efficient design and a program to implement the various intricacies for affine transform.

1. 7. References

[1] https://github.com/tangyeqiu/picoMIPS
[2] https://secure.ecs.soton.ac.uk/notes/elec6234/picomips/