

# ELEC6234 FPGA Synthesis of a picoMIPS Processor

Joshua Tyler  
jlt1e16  
MSc Embedded Systems  
Dr Jeff Reeve

## Abstract

**Summarise your work in less than 100 words stating briefly what was achieved.**

## 1 Introduction

**State the objectives of the assignment. Summarise briefly your preparation work, your experimental work,, and results achieved. Specifically, state which parts of the assignment were delivered according to the requirements and summarise any extensions to the basic specification you have carried out with references to the sections. ( approx. 0.5 page).**

The objective of this project was to design and build a picoMIPS processor capable of performing an affine transform of a two dimensional coordinate. This is equivalent to the matrix transformation of Equation 1

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} + \begin{bmatrix} B_1 \\ B_2 \end{bmatrix} \quad (1)$$

The goal of the exercise is to create a design as minimal as possible, but capable of implementing this affine transform.

In preparation for the assignment I conducted research into minimal instruction set computers, and designed my system on paper. The design of the instruction set is Summarised in section 2, and the design of the remainder of the system is discussed therein as well as in subsequent sections.

As an extension exercise I wrote SystemVerilog code to convert the signed 8-bit words on the switches and LEDs to binary coded decimal and then display this using the seven segment displays present on the development board. This is discussed briefly in Section 5.

**Mention size tests and investigation into logic element usage**

## 2 Instruction format, decoder design, program memory, and program counter

### 2.1 System overview

A block diagram showing my picoMIPS implementation is shown in Figure 2. In this diagram, blue lines show address signals, red shows data signals, and green shows control signals. The cycle counter is a four bit, one hot encoded counter, and controls the flow of data across four clock cycles, which make up one instruction cycle. The four cycles are fetch, decode 1, decode 2, and execute. A timing diagram showing the execution of the processor across these cycles is shown in Figure 1. During the fetch stage, the program counter is valid and so the new instruction will be fetched from the program memory on the next clock edge. During the two decode cycles, the processor retrieves the values of the two data registers from memory. Two cycles are necessary for this because the Cyclone IV

memory architecture only support dual port random access memories (RAMs) and read only memories (ROMs). Therefore two clock cycles must be used in order to read from two data addresses. During the execute cycle, the data values are valid, and so the ALU can calculate the result. The ALU is combinational, and so the branch flag and write data are valid on the next clock edge. This means that the program counter can be set to the new value, and the data can be written to the relevant register.

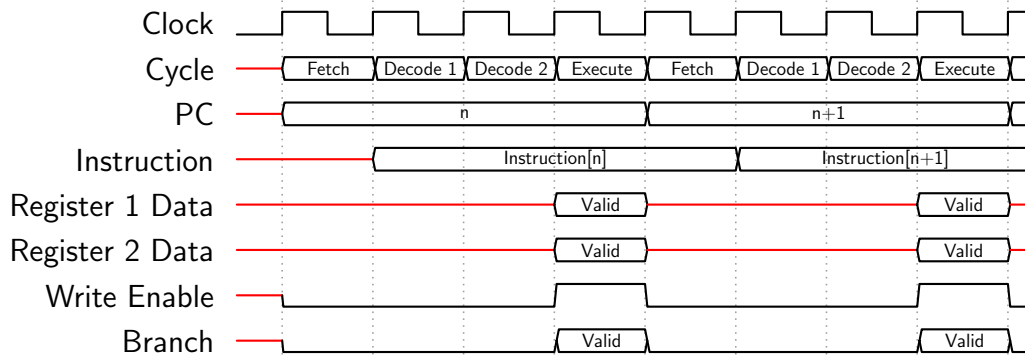


Figure 1: Processor timing diagram

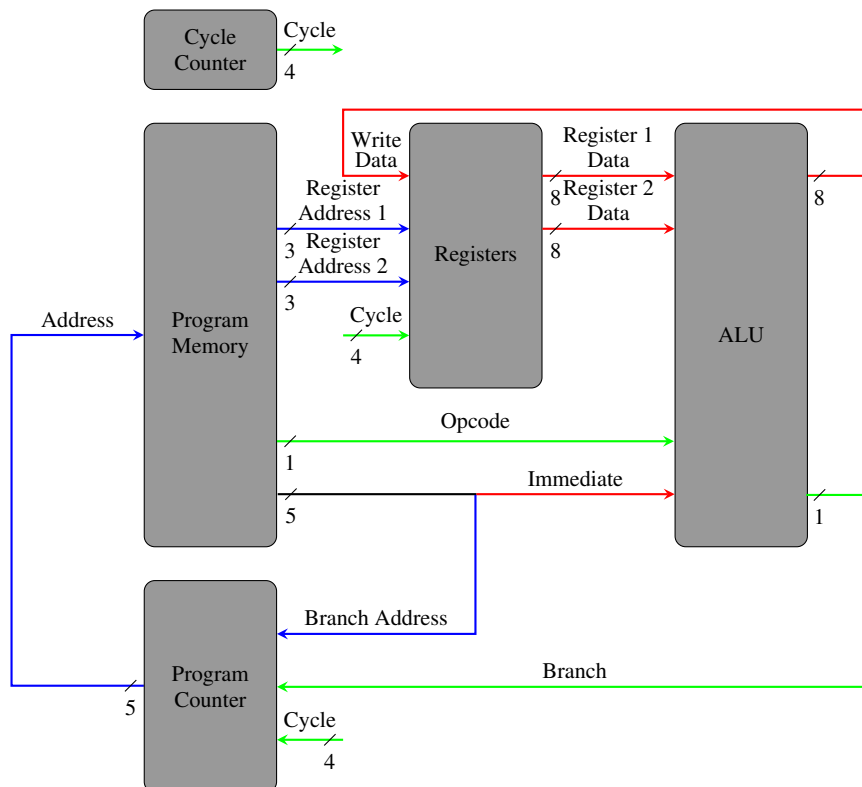


Figure 2: Final processor architecture

## 2.2 System Level testing

. fill in

## 2.3 Instruction Format

The processor uses 12 bit instructions. This breaks down into a one bit opcode, two 3 bit register addresses, and a 5 bit immediate / branch address, as shown in Figure 3. The opcode can be made one

bit long because there are only two instructions in the processor, and therefore one bit is enough to differentiate them. The immediate / branch address bits can be shared across the two functions because the MULTI instruction uses an immediate but does not branch, whilst the SUBLEQ instruction does not use an immediate, but can branch. The details of why these two instructions were chosen, and the details of their implementation is given in Section 4.

Bit	11	10	9	8	7	6	5	4	3	2	1	0
Purpose	op-code	Reg. 1 Address			Reg. 2 Address			Immediate / Branch Address				

Figure 3: Instruction Format and Assembler

A listing of the program used is given in Listing 1. This is a custom form of assembly code created for this project. Upon reading the source code, the reader will notice that the majority of assembly instructions used are not present as machine instructions on the processor architecture. This is because writing assembly code solely using SUBLEQ and MULTI instructions can be very confusing. For this reason a two-stage compilation toolchain was created using the Python programming language. The first stage takes each assembly instruction that doesn't map into a machine instruction, and re-writes it so that only SUBLEQ and MULTI instructions are used. This is performed by `optimiser.py`, and the second stage is to compile this assembly code into machine instructions. This is performed by `assembler.py`. The machine code output by the compilation toolchain is listed in Listing 2.

The majority of instructions such as MOV and ADD have their conventional definitions, however there are some slightly more esoteric instructions, namely JLEZ, and JGZ. These are represent 'jump if less than or equal to zero', and 'jump if greater than or equal to zero' respectively. These instructions are used to poll switch 8. Initially the more conventional JZ, and JNZ ('jump if zero', and 'jump if not zero') were used. The system worked with these instructions, however they require more SUBLEQ instructions to implement, and since the switch 8 register is guaranteed to be either 0x00, or 0x01 the simpler instructions are functionally equivalent.

## 2.4 Program Memory Design

The program memory block is very simple, it consists solely of a block of synchronous RAM initialised with the data from Listing 2. The RAM was inferred by creating an array of words, and then transferring the addressed word to the output on each rising clock edge, using a non blocking assignment within a rising clock edge always block. This inference approach was taken as opposed to using a dedicated Altera RAM library element because it allows the code to be portable between field programmable gate array (FPGA) vendors, and an explicit instantiation was not required because compilation with Quartus reveals that the synthesis engine has correctly inferred RAM from this block.

Due to its simplicity this block was not tested with an individual testbench, and the functionality of the block was verified during system level testing, as described in Section 2.2.

## 2.5 Program Counter Design

The program counter consists of three elements: a register to hold the current value of the program counter, a combinational adder to calculate the next value of the program counter if the program does not branch, and a multiplexer to select between the branch address and next instruction. The selected address is clocked into the register on the clock edge after the execute cycle. This in turn allow the program memory to fetch the correct instruction.

**This is wrong. We cannot use the RAM register because we cannot read it back, so we need a store of the current PC anyway Could mention consideration of storing the program counter**

**in the program code itself** The register is not strictly necessary in this design in the sense that one could make use of the read enable and asynchronous clear features of the program counter's memory block. This would also remove the need for the 'Fetch' cycle in the processor's execution. However the registers do not add to the cost figure of the design, since the logic cells are already being instantiated for the adder. In addition to this, the timing constraints of an asynchronous clear signal to the register input of the memory blocks are complex, and so likely to be violated by the use of a simple switch clear input.

The program counter has a total utilisation of five logic elements for the logic cells containing the adder and counter, and a further multiplier for the multiplexer. Due to it's simplicity this block was tested using the system level testing, as described in Section 2.2, rather than a dedicated testbench.

**Provide a block diagram of your picoMIPS design showing the sizes of all the busses and modules.**

**Describe your picoMIPS instruction format and the instructions you have implemented in your decoder. Give a listing of your program implemented in the Program Memory. You can show snippets of your source code. There is no need to show the full source code for all your modules in the report as the full source code must be submitted separately. Do not copy any code or diagrams from the lectures and picoMIPS SystemVerilog files provided on the ELEC6016 notes site. Give your Modelsim testbenches and Modelsim results. DO NOT make statements such as: Figure 2 shows the simulation results of the module functioning correctly. Instead, explain the results shown in the figures to demonstrate that you understand how the tested modules work. You can show RTL level diagrams from Quartus if you wish. (max 2.5 pages).**

### **3 General Purpose Register file design, simulation and synthesis**

The register block is the most complex block of the design. At its heart it uses a dual port RAM block (with one read port, and one write port) to access data. Reading data therefore uses two clock cycles, because the first register has to be read, and then the second has to be read subsequent to that. This limitation could be overcome by use of a triple port RAM. However this type of RAM would require Tri-port RAM would require doubling of the

**Could use tri-port RAM for this. Fill in once decision made.  
As above (max 1.5 page)**

### **4 Arithmetic Logic Unit and Multiplier Design**

**Explain the functions implemented in your ALU and explain your testbench. Show Modelsim test results. If you have implemented a hardware multiplier (or multipliers), explain your multiplier design and give Modelsim test results. State if your multiplier module synthesised as an embedded hardware multiplier. (approx. 1.5- 2 pages)**

### **5 Altera DE0 implementation**

**Explain how you tested your design after programming the FPGA. In case you had to edit your original code and resynthesize explain what you did. ( approx. 1-2 pages)**

### **6 Conclusion**

**State which objectives listed in your Introduction have been achieved. Calculate the cost figure of your design for synthesis on a Cyclone IV E.. Give your general conclusion, comment on what you learnt. Comment on ways to improve the design or extend it further. ( approx.0.25 0.5 of a page)**

## References

# Appendix A Program Code

Listing 1: Main Program

```
1 //Assembly for Affine Transform
2
3 //Define constants – data set 2
4     CONST    A11    4           // 00100 = 0.5
5     CONST    A12    25          // 11001 = -0.875
6     CONST    A21    25          // 11001 = -0.875
7     CONST    A22    6           // 00110 = 0.75
8
9     CONST    B1     5           // 00101 = 5
10    CONST    B2     12          // 01100 = 12
11
12 //Ensure that zero register is zero
13 SUBLEQ Z Z
14
15 //Load pixels
16 start: JLEZ    SW8    start      // Wait for SW8 = 0
17        MOV     SW17   R1         // Store X1 in R1
18 poll2: JGZ     SW8    poll2
19 poll3: JLEZ    SW8    poll3
20        MOV     SW17   R2         // Store Y1 in R2
21 poll4: JGZ     SW8    poll4
22
23 //Begin Affine algorithm execution part 1
24 //Note this could be optimised if some coefficients are repeated
25        MULTI   R1     R3    A11   // Multiply A11, and X1, store in
26        R3
27        MULTI   R2     R4    A12   // Multiply A12, and Y1, store in
28        R4
29        ADD     R3     R4         // Add R3 and R4, store in R4
30        LDI     R3     B1         // Store B2 in R3
31        ADD     R3     R4         // R4 = Y2 = B2 + (A21*X1) + (A22*
32        Y1)
33
34 //Begin output stage
35 //No need to move R4 to LED as it is already connected
36 poll5: JLEZ    SW8    poll5
37
38 //Begin Affine algorithm execution part 2
39 //Note this could be optimised if some coefficients are repeated
40        MULTI   R1     R3    A21   // Multiply A21, and X1, store in
41        R3
42        MULTI   R2     R4    A22   // Multiply A22, and Y1, store in
43        R4
44        ADD     R3     R4         // Add R3 and R4, store in R4
45        LDI     R3     B2         // Store B1 in R3
46        ADD     R3     R4         // R4 = X2 = B1 + (A11*X1) + (A12*
47        Y1)
48
49 //Begin output stage
50 //No need to move R4 to LED as it is already connected
```

45 poll6: JGZ SW8 poll6  
46 JP start

---

Listing 2: Main Program (compiled)

---

```
1 // Automatically generated memory map by python
2 // 06:58PM on April 26 2017
3 010110100001
4 010111100001
5 111000001000
6 010111100101
7 010110100011
8 010111100101
9 111000101000
10 010111101001
11 010110100111
12 100001000100
13 100101111001
14 001010101100
15 010101101101
16 010110101110
17 110001000101
18 001010110000
19 010101110001
20 010110110010
21 010111110010
22 100001011001
23 100101100110
24 001010110110
25 010101110111
26 010110111000
27 110001001100
28 001010111010
29 010101111011
30 010110111100
31 010111111110
32 010110111100
33 010110100001
```

---