ELEC6234 FPGA Synthesis of a picoMIPS Processor

Joshua Tyler
jlt1e16
MSc Embedded Systems
Dr Jeff Reeve

Abstract

This project covers the design, and implementation using SystemVerilog, of a picoMIPS embedded processor. The processor designed is Turing complete (if infinite memory were available), but is tailored to the execution of an affine pixel transform.

The architecture created is minimal in size, however a modular and parametric design allows for simple extension and modification for other purposes.

In addition to the processor design, a feature-rich assembler was written using Python, capable of translating high level assembly instructions into a series of machine code instructions.

A signed binary number to binary coded decimal converter was also created in order to allow decimal display of the binary output word using 7-segment displays.

1 Introduction

This project relates to the design and implementation of a picoMIPS processor. The objectives are:

1. Design of a picoMIPS processor, capable of executing an affine transform.
2. Implementation of the designed processor in SystemVerilog.
3. Verification of the SystemVerilog model, by simulation.
4. Validation of the synthesised SystemVerilog model.
5. Minimisation the resources used by the synthesised processor.

In preparation for the assignment, research was conducted into minimal instruction set computers, and the system was designed on paper. The design of the system is discussed in section 2 as a whole. It should be noted that whilst example code was provided to implement the processor, this was not used, and the processor has been designed from scratch.

Part of this preparation involved investigation into the structure of Altera logic blocks, and small test modules were synthesised in order to see their resource utilisation.

The processor design was successful and the processor is fully functional. As an extension exercise a SystemVerilog module was designed to convert the signed 8-bit words on the switches and light emitting diodes (LEDs) to a binary coded decimal (BCD) representation. This is then displayed using the seven segment displays present on the development board. This is discussed briefly in Section 4.

The simulation timing diagrams reproduced in the report have been exported from ModelSim, and converted to a Ti*k*Z timing waveform using modelsim2latex [1]. Trivial modifications were made to make the script compatible with the files exported from ModelSim. In each case, the waveform rendered by Ti*k*Z has been manually validated against the displayed result in ModelSim, to ensure correct operation of the tool.

2 System Design and Verification

2.1 Overview

A block diagram of the picoMIPS implementation is shown in Figure 1. Each block in this diagram represents a SystemVerilog module in the design.

The cycle counter defines the current stage of instruction execution. The processor is not pipelined, so the cycle counter controls which parts of the processor are active on any given clock cycle.

The program memory stores the instructions which make up the program, and on each instruction cycle it presents the next to the processor.

The register block contains a small amount of storage necessary for program execution. On each instruction cycle, two registers are read, and the result of the arithmetic logic unit (ALU) operation is written back to the second register.

The ALU performs the requested operations on the input registers, and presents the result to the rest of the processor so that the appropriate action can be carried out.

The final block is the program address multiplexer. This block is responsible for presenting the program memory with either the next address, or a branch address depending on the ALU result. In a traditional processor, the next address would be calculated and stored by a program counter. This processor uses a slightly different approach and stores the next address is stored in the program code instead. The reason for this choice is that it contributes less to the cost function. The processor uses 5-bit addresses, and therefore a 5-bit counter would be needed. This would require five logic elements (a cost of 5). There are 31 instructions in the main program, and so the cost of adding 5 bits to each one is $\frac{31\times5}{1024} \times 30 = 4.54$, therefore it is cheaper to not have a dedicated program counter.

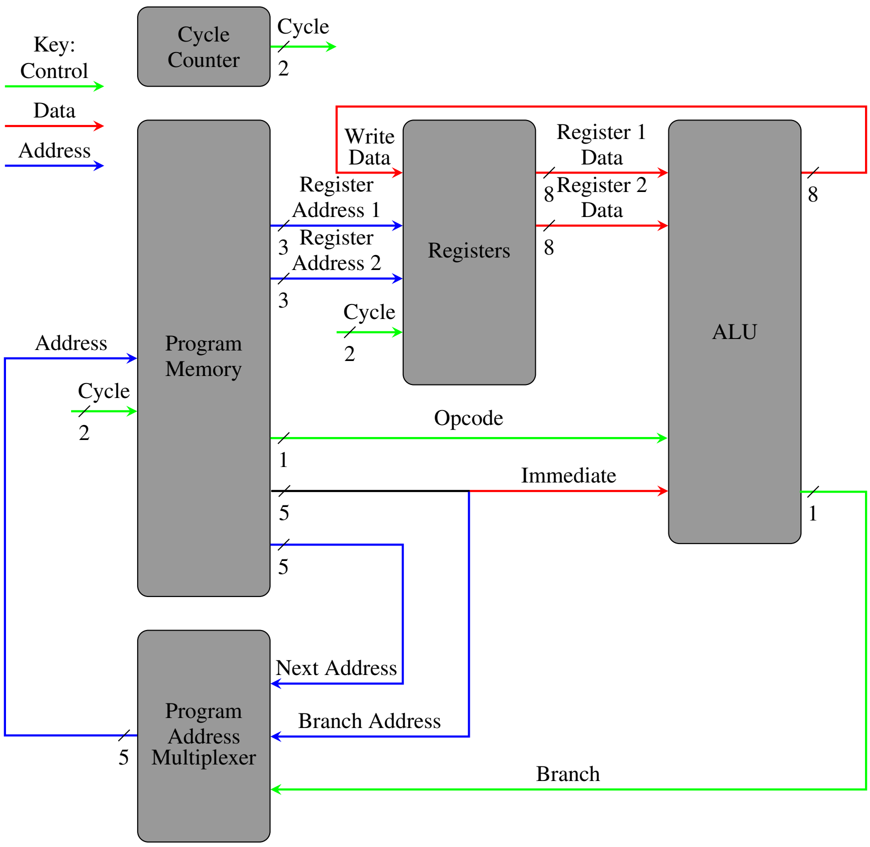

Figure 1: Final processor architecture

2.2 Cycle Counter Design

The counter has three states, decode, execute, and write, which make up one instruction cycle. This is represented using a one hot encoding, with the zero state also valid. This arrangement means that the counter uses the same number of bits as the equivalent binary counter. Careful assignment of state encodings means that no decoder is necessary for decoding the cycle anywhere in the design. This is possible because no signals need to be asserted in the decode state, and so this state can be encoded as zero.

A timing diagram showing the execution of the processor is shown in Figure 2. During the decode stage, the new instruction is valid, and so the register values can be fetched from random access memory (RAM). During the execute stage, the ALU generates the result and branch flag. During the write cycle this result is written back to RAM. A cycle is necessary for this because the RAM blocks inside the Cyclone IV field programmable gate array (FPGA) do not support read during write with new data, so a cycle is necessary to ensure that the register value is valid for the next decode cycle [2, p.3-16].

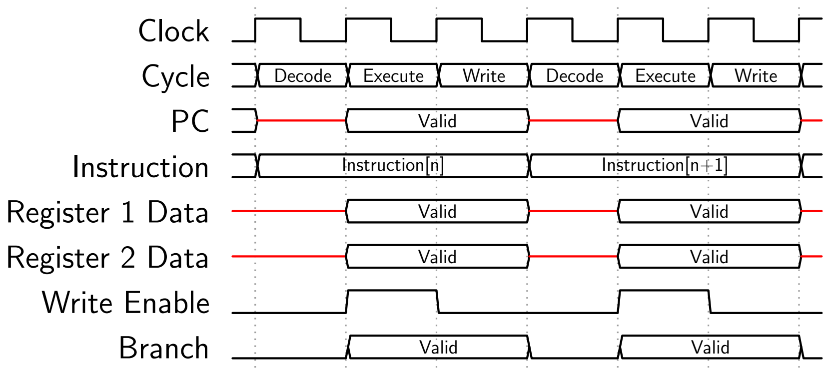

Figure 2: Processor timing diagram

2.3 Instruction and Program Design

The processor is capable of performing the two instructions tabulated in Table 1. `SUBLEQ` was chosen as the primary instruction because it is Turing complete by itself, so can execute any program. Using solely `SUBLEQ` instructions does have a downside however. Executing a multiply operation would require many lines of code, and so would take a large amount of time, as well as a lot of program and register memory. For this reason a second instruction, `MULTI`, is also used. This instruction also gives a method of loading immediates, which would require an additional multiplexer in a `SUBLEQ` only system.

Table 1: Instructions

Mnemonic	Mathematical operation	English Description
`SUBLEQ`	$\text{regs}[b] = \text{regs}[b] - \text{regs}[a];$ $if(\text{regs}[b] \leq 0)$ branch;	Subtract and branch if less than, or equal to, zero.
`MULTI`	$\text{regs}[b] = \text{regs}[a] \times \text{Immediate};$	Multiply immediate.

The processor uses 17 bit instructions. This breaks down as shown in Figure 3.

The immediate / branch address bits can be perform 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 3.

16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op.	Reg. 1 Addr.			Reg. 2 Addr.			Imm. / Branch Address					Next Address				

Figure 3: Instruction Format and Assembler

The affine transform which the processor executes can be represented by Equation 1 [3].

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

A listing of the program written to implement this is given in Appendiz A. This is a custom dialect 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 in 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 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 read only memory (ROM), initialised with the data from Listing 7.

Initially the ROM was inferred from SystemVerilog code. However later the ROM was instantiated using a dedicated Altera library element. The reason for this choice is that the design requires access to the asynchronous clear input on the address register in order to reset the design. This is something which was difficult to infer using standard SystemVerilog code, since Quartus would fail typically add an external register created using logic blocks to the design.

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 3.1.

2.5 Program Address Multiplexer Design

The multiplexer used to multiplex between the next address and branch address is not implemented using a traditional structure, instead a multiplier is used.

The reason for this choice is that an n bit multiplexer requires n logic blocks for a traditional implementation, however using hardware multipliers, one can implement up to a 9 bit multiplexer using a single 18×18 hardware multiplexer. This would represent a cost of 2 (because an 18×18 multiplexer is formed of two 9×9 multipliers).

The operation of the multiplexer is illustrated by Figure 4. If the two words to be multiplexed a and b are concatenated to a single multiplier input, and a single bit is set in the second word, then the input can be shifted by a set amount. Setting the second input to either 1 or $(1 << (\text{Word Width}))$, therefore acts as a multiplexer if we observe the output bits from $[(\text{Word Width}) \times 2 : (\text{Word width}) + 1]$.

Input	$b[3]$	$b[2]$	$b[1]$	$b[0]$	$a[3]$	$a[2]$	$a[1]$	$a[0]$
Sel	0	0	0	0	0	0	0	1
Result	$b[3]$	$b[2]$	$b[1]$	$b[0]$	$a[3]$	$a[2]$	$a[1]$	$a[0]$
Input	$b[3]$	$b[2]$	$b[1]$	$b[0]$	$a[3]$	$a[2]$	$a[1]$	$a[0]$
Sel	0	0	0	1	0	0	0	0
Result	$a[3]$	$a[2]$	$a[1]$	$a[0]$	0	0	0	0

Figure 4: 4-bit Multiplexer Operation

This multiplexer was designed as a parametric model, so that it can be used as the program address multiplexer, as well as elsewhere in the design.

2.5.1 Testing

In order to test the multiplexer a testbench was written which instantiates an 8 bit multiplexer, asserts two numbers on it's inputs, and switches between them using `sel`. The value of the output using assertions.

The stimulus portion of the testbench is listed in Listing 1, and the resultant waveform is presented in Figure 5. Inspection of the waveform demonstrates that the output is equal to `a` when `sel` is low, and equal to `b` when `sel` is high. This is the expected behaviour, and the assertions in the testbench pass.

Listing 1: `test_multiplexer.sv` Stimulus

```
initial
begin
    a = 34;
    b = 95;
    sel = 0;
    # 10ns;
    assert(out == a);
    sel = 1;
    # 10ns;
    assert(out == b);
    sel = 0;
    # 10ns;
    assert(out == a);
    $stop;
end
```

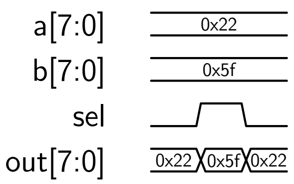

Figure 5: `test_multiplexer.sv` Output

2.6 Register Design

2.6.1 Layout

The register block is the most complex block of the design. At its heart it uses a dual port RAM block (with one dedicated read port, and one read/write port) to access data.

The memory map of the registers is shown in Table 2. Registers R1–R4 are general purpose computation registers. These are entirely application specific registers, and both reading and writing is legal for any of them. The contents of R4, however, does also map to the LEDs on the FPGAs development board.

The U register stands for unity. This register is guaranteed to hold the constant necessary for an immediate to be loaded directly into a register using the `MULTI` command. Despite the name, this register actually holds the constant 4,because immediates are treated treated as a fractional constant by the multiplier, as discussed in Section 3. It is forbidden for any program to write to this value, as doing so will break the `LDI` (load immediate) command. The value of this register is initialised by the bitstream only, and resetting the processor using the reset switch will not reset its value. It should be noted that writing to this register is not prevented in hardware, but it is forbidden for any program to do so.

The Z register stands for zero. The value of this register is kept at zero, and writing to it should be done with extreme caution. Many of the higher level assembly commands internally rely on this register being zero. Writing to Z, however, is not forbidden entirely as many of these higher level commands use it as a general purpose computation register, but they all guarantee to clear Z back to zero before completion. Z is not initialised to zero by the bitstream, but instead the first command of the program must be `SUBLEQ Z Z` in order to clear it. This approach is taken so that the processor still functions correctly if it is reset using the reset switch whilst Z is non-zero.

The SW07 and SW8 registers are different from the others in that they do not map to internal storage in the FPGA. When the program attempts to read their value the value of switches 0–7, or switch 8 is returned instead, this is achieved by multiplexing the data outputs of the register bank. There do exist, however, registers inside the register memory at the addresses of the switch registers, this is because writing to the SW07 and SW8 registers is legal, but has no effect, physical registers need to exist therefore in order to avoid an out of range write.

Table 2: Register map

Address	Mnemonic
0x0	R1
0x1	R2
0x2	R3
0x3	R4 / LED
0x4	U
0x5	Z
0x6	SW07
0x7	SW8

2.6.2 Implementation

The registers are implemented as shown in Figure 6. Data 1 is set to the value of the data stored at address 1 on each rising clock edge, and data 2 is set to the value of the data stored at address 2 if the write enable is false, otherwise the value stored at address 2 is replaced with the write data.

The LED register mirrors register R4 in the main memory bank, but allows the LEDs to be constantly driven.

The switch multiplexers are three way multiplexers that multiplex between the register data, switches[0:7], and switches[8], dependant on the register address selected. The control signals for these multiplexers are derived from two register addresses. Internally the multiplexers are formed of two cascaded two input multiplexers, one to choose between the two registers of switches, and one to select between the switch values and the data register.

2.6.3 Testing

In order to test the registers. A testbench was written which instantiates the register block, and presents its inputs with the signals which would be present if if were part of the overall system.

The stimulus portion of the testbench is listed in Listing 2, and the resultant waveform is presented in Figure 7. Inspection of the stimulus portion shows that the registers are initially loaded with random values for testing, this allows the registers to be tested more rigorously since each register will have

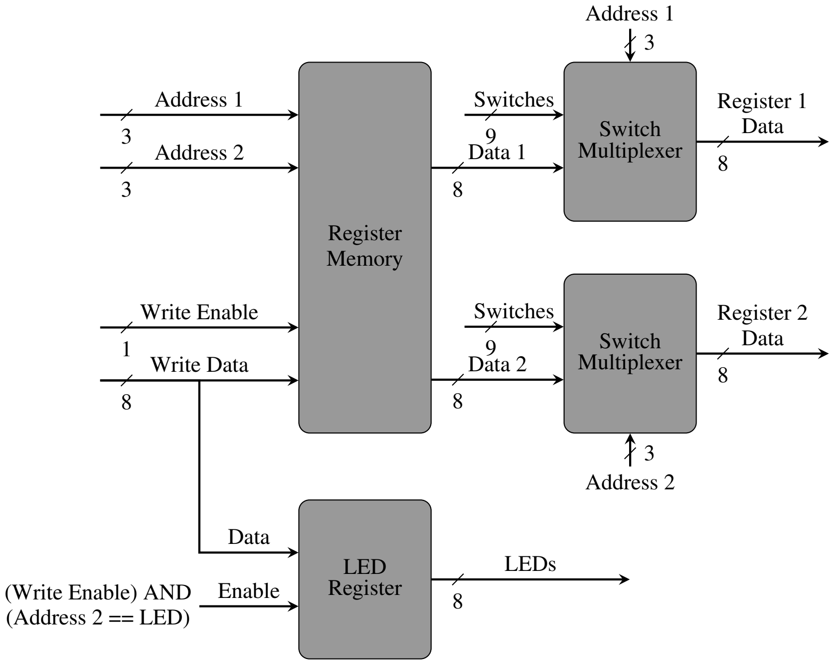

Figure 6: Register memory architecture

a unique value, rather than the initialisation value. A SystemVerilog task was written to perform one read/write cycle, and this uses assertions to test the output, and internal value of the register memory, at each state.

For brevity, Figure 7 shows a single iteration of the `rwTest` task, and is the result of executing the algorithm up to the commented out `$stop` command on line 96. Inspection of the Figure shows that the values of registers at addresses (0x0), and (0x1) is requested, these are (0x72), and (0xb2) respectively. These values are correctly asserted on the output after one clock cycle. In addition, the value of the write data, (0x14), is written to register (0x2) on the third clock cycle as expected.

Listing 2: `test_regs.sv` Stimulus

```
82  //Stimulus
83  initial
84  begin
85      //Initialise memory with dummy values
86      dut.mem0.mem[`REG_R1_ADDR] = $urandom_range(255,0);
87      dut.mem0.mem[`REG_R2_ADDR] = $urandom_range(255,0);
88      dut.mem0.mem[`REG_R3_ADDR] = $urandom_range(255,0);
89      dut.mem0.mem[`REG_R4_ADDR] = $urandom_range(255,0);
90      dut.mem0.mem[`REG_Z_ADDR] = 0; //The program code normally does this
91      dut.mem0.mem[`REG_SW07_ADDR] = $urandom_range(255,0);
92      dut.mem0.mem[`REG_SW8_ADDR] = $urandom_range(255,0);
93      # 1ns;
94      $display($time, ": begin test 1");
95      rwTest(`REG_R1_ADDR,`REG_R2_ADDR, $urandom_range(255,0),$urandom_range
            (255,0));
96  //  $stop;
97      $display($time, ": begin test 2");
98      rwTest(`REG_R3_ADDR,`REG_R4_ADDR, $urandom_range(255,0),$urandom_range
            (255,0));
99      $display($time, ": begin test 3");
100     rwTest(`REG_SW07_ADDR,`REG_R1_ADDR, $urandom_range(255,0),
            $urandom_range(255,0));
```

```
    $display($time, ": begin test 4");
    rwTest(`REG_SW8_ADDR,`REG_R2_ADDR, $urandom_range(255,0),
        $urandom_range(255,0));
    $display($time, ": begin test 5");
    rwTest(`REG_Z_ADDR,`REG_U_ADDR, $urandom_range(255,0),$urandom_range
        (255,0));
    $display("Testing complete");
    $stop;
end
```

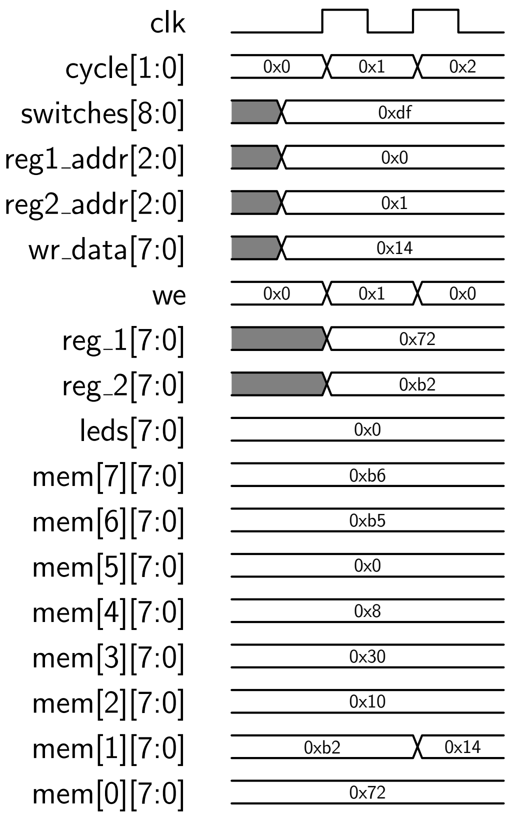

Figure 7: `test_regs.sv` Output

3 ALU Design

The ALU design is relatively simple owing to the fact that the processor only contains two instructions. This means that the ALU only contains the implementation of the instructions, and a multiplexer to assert the correct output based upon the opcode. This is illustrated graphically in Figure 8. The multiplexer implementation uses a multiplier as discussed in Section 2.5

`SUBLEQ` is implemented using logic elements to form a subtracter, then the output of the subtracter is tested for the branch condition using a multiplier.

The condition to branch is if the result is less than or equal to zero, we can test this using multiplication by -1. $-1 \times 0 = 0$, and $-1 \times$ [Negative number] $=$ [Positive number], however $-1 \times$ [Positive number] $=$ [Negative number], therefore the branch condition is the logical negation of the most significant bit (MSB) of the multiplier output. This saves the use of several logic elements.

`MULTI` requires minimal overhead (only a single hardware multiplier – a cost of 1 in the cost function). Its implementation is straightforward, simply inferring a single hardware multiplier. The only slight complication is setting the significance of the input words. Whilst the data stored in the register has normal significance $[-2^7 : 2^0]$, the immediate has significance $[-2^1 : 2^{-3}]$.

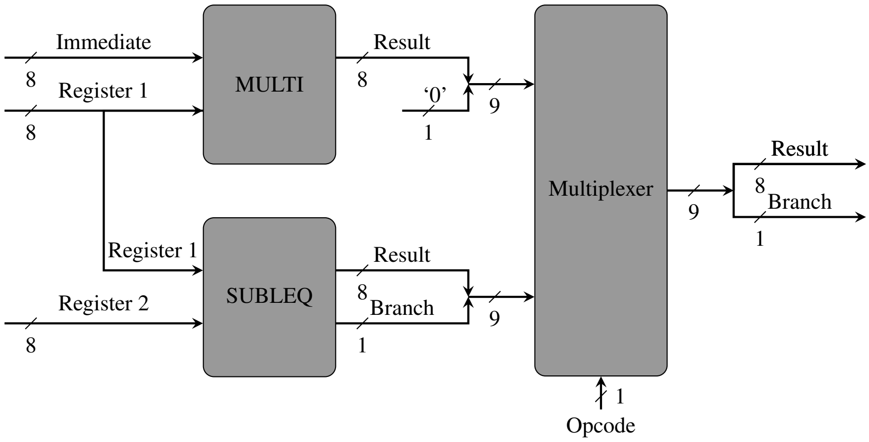

Figure 8: ALU architecture

3.0.1 Testing

Due to the simplicity of both overall the overall ALU, and the `SUBLEQ` module, I chose to only test the `MULTI` operation separately testing of the other modules is addressed in system level verification. I wrote a testbench which instantiates `MULTI`, presents stimulus inputs and ensures that the result produced is equal to manually calculated results.

The stimulus portion of the testbench is listed in Listing 3, and the resultant waveform is presented in Figure 9. Inspection of the stimulus shows that for each case the multiplier correctly calculates the value of the of `result`, such that it is equal to the manually calculated value in each case. This is confirmed by the assertions.

Listing 3: `test_multi.sv` Stimulus

```
initial
begin
    register = 8'b00000110; //6
    immediate = 5'b00110; //0.75
    #1ns
    assert(result == 8'b00000100); //4 (truncated from 4.5)

    #10ns
    register = 8'b00001000; //8
    immediate = 5'b01100; //12 (but divided down for immediate)
    #1ns
    assert(result == 8'b00001100); //12 (truncated from 4.5)

    #10ns
    register = 8'b10000000; //-128
    immediate = 5'b00100; //0.5
    #1ns
    assert(result == 8'b11000000); //12 (truncated from 4.5)
    #10ns;
    $stop;
end
```

3.1 System Level testing

To test the overall functionality of the system, a testbench was created to simulate switch input, and confirm the output result against a model of the affine transform calculated by the testbench. Using

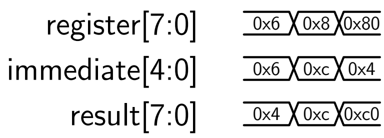

Figure 9: `test_multi.sv` Output

this method, the system was able to be tested for all $2^{16} = 65536$ possible inputs. Listing 4 shows the stimulus for the testing, which loops over all the possible input combinations. The stimulus in each case, along with expected values, is logged to a text file so that it can be used during the validation process as well. The key part of this stimulus is the SystemVerilog task `testAffineTransform`. This function calculates the expected value, mimics the switch input which the user would perform, and ensures that the outputs match using assertions.

Figure 10 shows the stimulus and output for one transform $x_1 = -1$ (0xff), $y_1 = 2$ (0x2). The expected outputs for this operation are $x_2 = 2$ (0x2), $y_2 = 13$ (0xd), so inspection of the `LED` signal in the waveform, along with `SW[7:0]`, show that the result is correct. Note that clock is shown as a solid bus in the waveform because it has too many cycles to be printed at such a scale.

Listing 4: `test_picoMIPS.sv` Stimulus

```
//Stimulus
initial
begin
    logFile = $fopen("log.txt");
    $fdisplay(logFile, "xi\tyi\txo\tyo");

    //Initialise
    SW17 = 0;
    SW8 = 0;

    //Reset
    SW9 = 0;
    #100ns;
    SW9 = 1;

    //testAffineTransform(-1,2);
    //$stop;

    //Test all possible values
    for(int i = -128; i < 128; i++)
    begin
        $display(i);
        for(int j = -128; j < 128; j++)
        begin
            testAffineTransform(i,j);
        end
    end

    $fclose(logFile);
    $stop;

end
```

Figure 10: `test_picoMIPS.sv` Output

4 Altera DE2-115 Implementation and Validation

Due to the extensive system level verification discussed in Section 3.1, few problems were encountered during implementation. Upon initial testing, it was noted that the system did not respond to any input. Investigation of the project configuration in Quartus revealed that the pin mapping `.qsf` file had not been loaded properly, and so the pin assignments were not correct. Upon fixing this issue, the design worked as expected without further modification.

The validation methodology was:

1. Select an x and y co-ordinate to test.
2. Look up the expected result in the output log produced by the system level testbench.
3. Input the data to the system under test.
4. Check the result produced.

In particular, attention was paid to ensuring that a range of random values, as well as edge cases (co-ordinates close to 0 and the limits of the integer size) were validated. In all cases the tested value matched the expected result.

4.1 Decimal decoder

In order to improve the ease of validation, a small SystemVerilog module was written to decode the signed 8-bit values on the switches and LEDs, and display them in decimal on four seven segment displays. This module does not contribute to the cost figure of the design as it is instantiated outside of the picoMIPS module. The design of the module will not be covered in detail, as it is largely outside of the scope of the project.

4.1.1 Testing

Testing of the decimal decoder is shown in Listing 5 and Figure 11. In the testbench stimulus, it can be seen that all $2^8 = 256$ possible integers are tested, however the shown waveform presented in this report, only covers the first two -128 and -127. Inspection of the waveform confirms that for -128 (0x80), the sign signal is true, and the tens, hundreds and units are correctly decoded to 1, 2, and 8 respectively. The same is true for the second test, -127 (0x81).

Listing 5: `test_bin_to_bcd.sv` Stimulus

```
initial
begin
    for(int i = -128; i < 128; i++)
    begin
        in = i;
        # 10ns;
    end
    $stop;
end
```

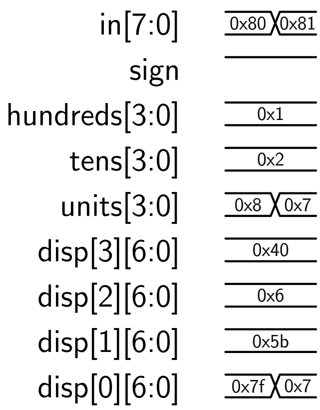

Figure 11: `test_bin_to_bcd.sv` Output

5 Conclusion

All of the objectives noted in Section 1 have been achieved. Objectives 1 and 2 are realised through the design, and subsequent SystemVerilog implementation, of the processor discussed in Section 2. Objectives 3 and 4 have been realised through the verification and validation of the design discussed in Sections 3.1 and 4 respectively.

Objective 5, minimalisation of the design has been at the heart of the design philosophy throughout the project, and this culminates in the tiny cost Figure calculated by Equation 2 [3].

$$\begin{aligned}
\text{Cost} &= [\text{No. Logic Elements}] + \max([\text{No. 9-bit Multipliers used}] - 2, 0) + \frac{[\text{kBits of RAM}]}{1024} \times 30 \\
&= 13 + \max((13-2), 0) + \frac{607}{1024} \times 30 \\
&= 13 + 11 + 17.78 \\
&= 41.78 \qquad (2)
\end{aligned}$$

The created processor performs well and can easily be tailored to new applications due to the parametrised and modular design. The fact that the processor is accompanied by a powerful assembler aids in its adaptability to new applications.

In conclusion there are few ways in which the design could be improved without vastly extending the scope of the design. One key element that would make it more versatile would be adding support for a higher level programming language. This would allow the system to be tailored to new applications with ease. A C compiler would be the obvious choice, however this would encompass a large design effort, so porting a Forth runtime to the processor would likely be a more realistic goal.

References

[1] sh-ow. (2016, jan) modelsim2latex. [Online]. Available: https://github.com/sh-ow/modelsim2latex/

[2] Altera, *Cyclone IV Device Handbook*. Altera, mar 2016, vol. 1.

[3] T. J. Kazmierski, “Systemverilog design of an embedded processor,” University of Southampton, Tech. Rep., 2017.

Appendix A Program Code

This appendix contains the program which the picoMIPS processor runs, both in assembly code and machine code form.

Listing 6: Main Program

```
//Assembly for Affine Transform

//Define constants - data set 2
        CONST   A11     4               // 00100 = 0.5
        CONST   A12     25              // 11001 = -0.875
        CONST   A21     25              // 11001 = -0.875
        CONST   A22     6               // 00110 = 0.75

        CONST   B1      5               // 00101 = 5
        CONST   B2      12              // 01100 = 12

//Ensure that zero register is zero
SUBLEQ Z Z

//Load pixels
start:  JLEZ    SW8     start                   // Wait for SW8 = 0
        MOV     SW17    R1                      // Store X1 in R1
poll2:  JGZ     SW8     poll2
poll3:  JLEZ    SW8     poll3
        MOV     SW17    R2                      // Store Y1 in R2
poll4:  JGZ     SW8     poll4

//Begin Affine algorithm execution part 1
//Note this could be optimised if some coefficients are repeated
        MULTI   R1      R3      A11             // R3 = A11 * X1
        MULTI   R2      R4      A12             // R4 = A12 * Y1
        ADD     R3      R4                      // R4 = R3 + R4
        LDI     R3      B1                      // Store B2 in R3
        ADD     R3      R4                      // R4 = Y2 = B2+(A21*X1)+(A22*Y1)

//Begin output stage
//No need to move R4 to LED as it is already connected
poll5:  JLEZ      SW8     poll5

//Begin Affine algorithm execution part 2
//Note this could be optimised if some coefficients are repeated
        MULTI   R1      R3      A21             // R3 = A21 * X1
        MULTI   R2      R4      A22             // R4 = A22 * Y1
        ADD     R3      R4                      // R4 = R3 + R4
        LDI     R3      B2                      // Store B1 in R3
        ADD     R3      R4                      // R4 = X2 = B1+(A11*X1)+(A12*Y1)

//Begin output stage
//No need to move R4 to LED as it is already connected
poll6:  JGZ     SW8     poll6
        JP      start
```

Listing 7: Main Program (compiled)

```
-- Automatically generated memory map by python
-- 03:00AM on April 28 2017

DEPTH = 31;
WIDTH = 17;
ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;
CONTENT
BEGIN

00 : 01011010000100001;
01 : 01011110000100010;
02 : 11100000100000011;
03 : 01011110010100100;
04 : 01011010001100101;
05 : 01011110010100110;
06 : 11100010100000111;
07 : 01011110100101000;
08 : 01011010011101001;
09 : 10000100010001010;
0a : 10010111100101011;
0b : 00101010110001100;
0c : 01010110110101101;
0d : 01011010111001110;
0e : 11000100010101111;
0f : 00101011000010000;
10 : 01010111000110001;
11 : 01011011001010010;
12 : 01011111001010011;
13 : 10000101100110100;
14 : 10010110011010101;
15 : 00101011011010110;
16 : 01010111011110111;
17 : 01011011100011000;
18 : 11000100110011001;
19 : 00101011101011010;
1a : 01010111101111011;
1b : 01011011110011100;
1c : 01011111111011101;
1d : 01011011110011110;
1e : 01011010000111111;

END;
```