ELEC6234 – SystemVerilog Design of an Embedded Processor

Max Fuller
mf5g14
MEng Electronic Engineering
Dr Alex Weddell

ABSTRACT: *A SystemVerilog implementation of a picoMIPS architecture was designed and created to run an affine transformation program with fixed coefficients. Several modifications and optimisations were applied to the architecture to reduce the hardware cost. Simulations were conducted for each module using ModelSim. The synthesised design uses 127 logic units, 1 embedded multiplier and 240 bits of RAM. The final cost function was calculated to be 134. Multiple tests were conducted using the DE1-SoC FPGA development board. All test results were confirmed using a custom MATLAB script that reproduces the affine transform algorithm.*

1. Introduction

The objective of this piece of work was to design, model and test a synthesisable implementation of a picoMIPS-style processor. The design must be capable of performing an affine transform on a manually-entered set of co-ordinates and produce the correct output. It was required to design it in such a way as to use as few hardware resources as possible whilst still performing as a processor, as opposed to a block of dedicated hardware. This meant that there should be two separate discernible sections – a control path and a data path, with the affine transform implemented using generic instructions stored in program memory.

An initial overall system design was created before any HDL coding took place. This made it clear what components were required, how they would fit together and where there might be opportunities for space optimisation later in development. The design was started at the top-level (namely *picoMIPS*), with all the lower-level modules then being added before deciding on the data and control wire connections.

Once the design was complete, development of each module began using SystemVerilog. This was done from the bottom up - starting with the simplest of modules like the program counter. A testbench was created for each module so they could be simulated using ModelSim to test their functionality before moving on to the next one.

Unit testing meant that, once wired up, the overall system worked on the first attempt. Similarly, synthesising the design and running it on a DE1-SoC board also worked the first time around.

Several optimisations were done to the original design to reduce the hardware resources further, as was suggested in the specification:

> *"You may design your own instruction set and modify the instruction format in any way you wish. You may also modify the architecture if it helps to reduce the cost figure." [1]*

These included a minimisation of the instruction format and removal of branching in the program counter. The final design correctly runs the affine transform and is still a processor.

The remainder of this report provides an overview of the system design, followed by discussions about the design decisions made for each module, testing processes and synthesis outputs. It concludes by explaining how the complete system was synthesised for and tested on the DE1-SoC FPGA development board.

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

The overall design of the system can be seen in Figure 1. It is clearly built up of two sections – the *data path* and the *control path*. The control path - consisting of the program counter, program memory and decoder - handles reading instructions and deciding what to do with them, whilst the data path – made up of general purpose registers and an ALU – processes the data. Immediate values can be fed into the ALU from either literals in the program itself or from the development board switches, *SW[7:0]*.

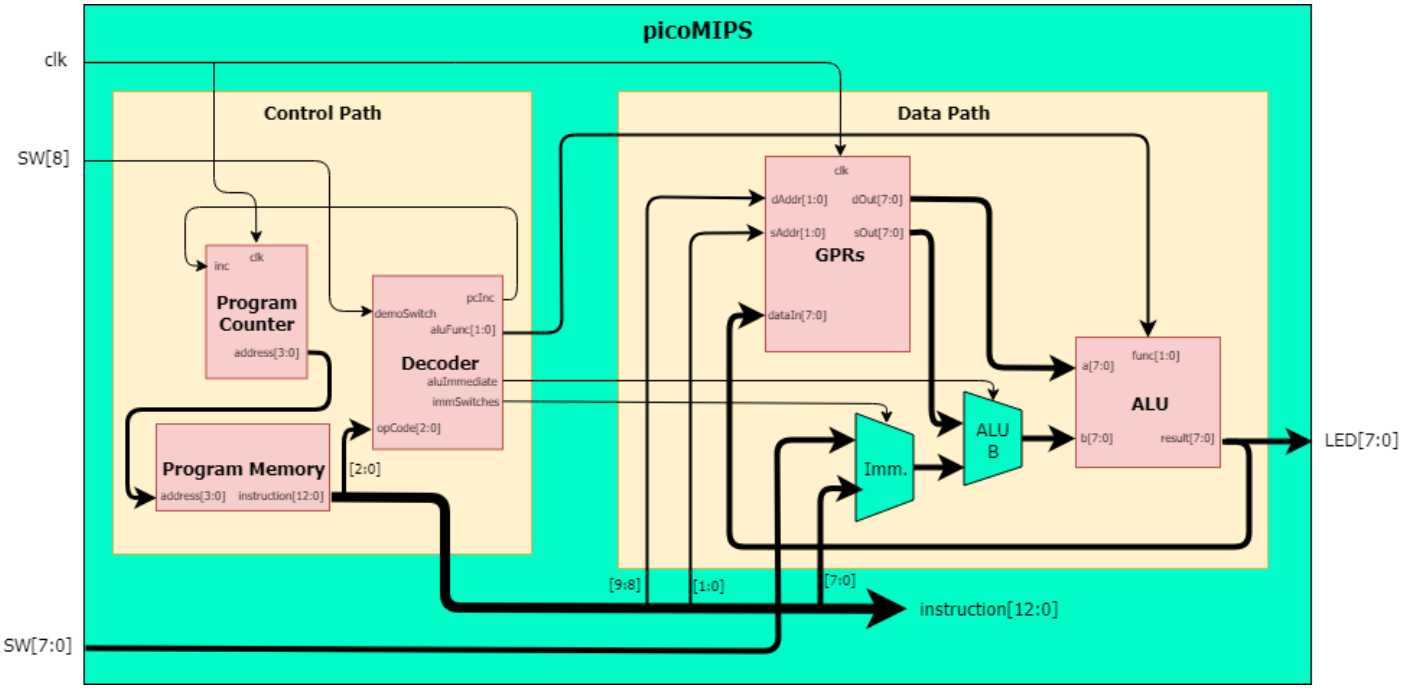

Figure 1 - Overall system block diagram with bus widths set according to the parameters that were used for the final synthesised design (e.g. N=8, O_SIZE=3)

A configuration file, *cpuConfig.sv (*Figure 2*)*, was created to define several parameters that determine the widths of the various data and address buses in the design, such as the size of the program memory (*P_SIZE*). Parameter values are passed down through each of the sub-modules. This enables for very quick and simple modifications to the entire design should the number of general purpose registers increase, for example.

In addition to the system parameters, the configuration file defines all the OpCodes and ALU functions as enumerated logic types. It also has an optional *DEMO_MODE* macro (discussed later) that results in the synthesis of extra logic to drive some seven-segment displays and enable push-button input from the FPGA development board.

The various elements of the configuration are used within other modules via the SystemVerilog *import* statement.

```
package cpuConfig;

    // Uncomment to enable 7-seg displays and clock LED
    `define DEMO_MODE


    // CPU parameters
    parameter N      = 8;   // Data bus width
    parameter A_SIZE = 2;   // ALU function width
    parameter O_SIZE = 3;   // OpCode width
    parameter P_SIZE = 4;   // Program memory address width
    parameter R_SIZE = 2;   // GPR address width


    // OpCodes
    typedef enum logic [(O_SIZE-1):0] {
        LDI,    // 000 Load immediate
        LDS,    // 001 Load from switches
        ADD,    // 010 Add
        ADDI,   // 011 Add immediate
        MUL,    // 100 Multiply
        MULI,   // 101 Multiply immediate
        WAIT0,  // 110 Wait for SW8 to be 0
        WAIT1   // 111 Wait for SW8 to be 1
    } opCode_t;


    // ALU functions
    typedef enum logic [(A_SIZE-1):0] {
        ALU_A,      // 00
        ALU_B,      // 01
        ALU_ADD,    // 10
        ALU_MUL     // 11
    } aluFunc_t;


endpackage
```

Figure 2 - Configuration file, cpuConfig.sv, that defines the system parameters as well as OpCodes and ALU functions

Instruction format

Unlike in a typical processor whose instructions may use two operands in addition to an immediate operand, this design was modified to use just one. The general format is as follows:

Instruction (13-bit) = OpCode (3-bit) + Destination register address (2-bit) + Immediate literal/Source register address (8-bit)

Using such a simple instruction format means that the hardware required to utilise it, as well as the program memory, can be a lot smaller. Its main limitation is that the result of ALU operations - such as an addition - overwrites the register that was supplying the first operand. It therefore requires very careful programming in order for data to not be lost.

Whilst this configuration is perhaps not ideal for a "generic" CPU, it is sufficient to perform the required affine transform and maintains the requirement that the design functions as a processor, whilst minimising hardware utilisation.

Decoder design and implementation

Deciding which instructions to implement into the design occurred alongside creating the affine transform program (below). That is, instructions were added to the decoder as required for each part of the program to work. The final instruction set is listed in Table 1.

OPCODE	VALUE	NAME	DESCRIPTION
LDI	000	Load Immediate	Load immediate value into destination register
LDS	001	Load Switches	Load value of switches into destination register when SW8 is 1
ADD	010	Add	Add source register to destination register
ADDI	011	Add Immediate	Add immediate value to destination register
MUL	100	Multiply	Multiply destination register by source register
MULI	101	Multiply Immediate	Multiply destination register by immediate value
WAIT0	110	Wait SW8 0	Wait for SW8 to be 0 – display destination register
WAIT1	111	Wait SW8 1	Wait for SW8 to be 1 – display destination register

Table 1 - Implemented instruction set showing literal OpCode values as well as descriptions for each instruction

The *OpCode* section of the instruction to be executed is fed into the decoder which then controls what the ALU and program counter do. The control wires are described in Table 2.

WIRE NAME	USED FOR	DESCRIPTION
aluFunc[1:0]	All instructions	Selects the ALU function
aluImmediate	ADDI, MULI	Toggles ALU B between an immediate or register value
immSwitches	LDS	Toggles immediate value between a hard-coded literal and switch inputs
pcInc	LDS, WAIT0, WAIT1	Toggles between the program counter incrementing or stalling

Table 2 - Control wires emanating from the decoder, showing the instructions they are used for and a description of what they do

A *case* statement, operating on the inputted *opcode*, makes up the decoder. Default values for each control signal are assigned before the case statement such that each instruction need only alter the value of the control wire(s) it is affecting. For example, the program counter is typically required to increment each clock cycle, thus *pcInc* is set to 1 by default and only changed if a stall is required (*WAIT0*, *WAIT1* or *LDS*). In

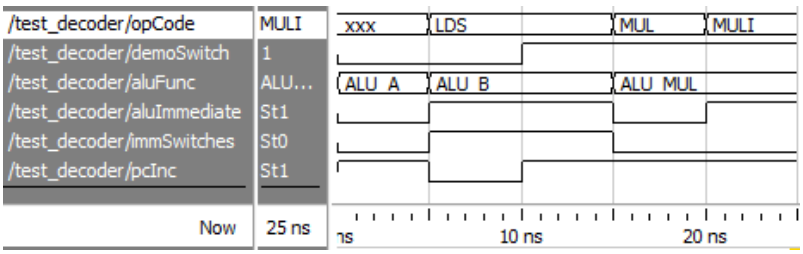

Figure 3 - Simulation of the decoder module, showing control outputs. The unknown OpCode demonstrates the default values for each control line, with the ALU passing through input A back into the register it comes from. The LDS instruction demonstrates how the pcInc signal is affected by SW8 (demoSwitch)

such a case, it is set depending on the state of SW8 (used for the demonstration).

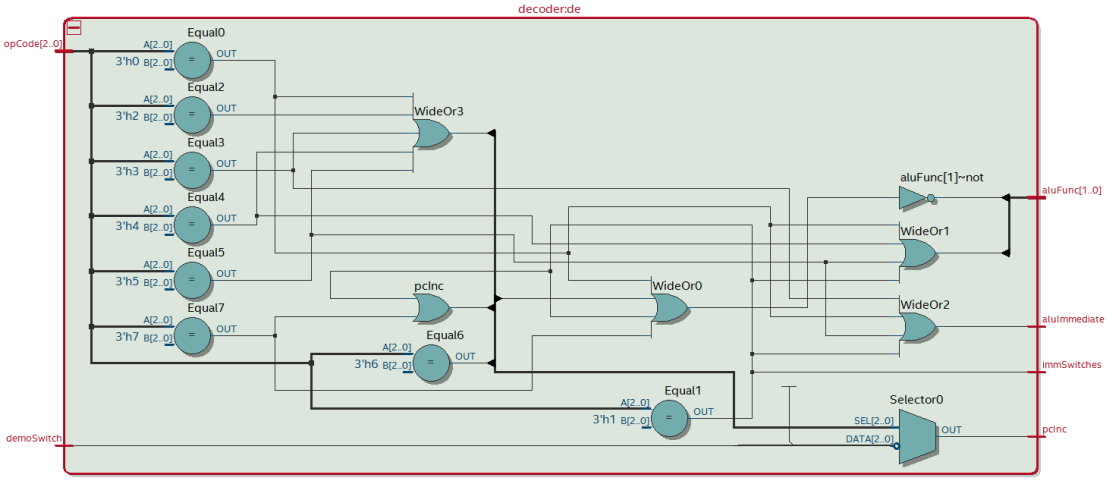

Figure 4 - Synthesised logic of the decoder module that is made up of basic gates, comparators and a multiplexer

ModelSim simulation results can be seen in Figure 3 for a testbench that tests both load instructions, both multiplication instructions and the default control wire values. Its synthesised layout is shown in Figure 4.

Affine transform program

"An affine transform is a linear mapping method that preserves points, straight lines, and planes. Sets of parallel lines remain parallel after an affine transformation." [2]

Often used to correct for geometric distortions in camera images, an affine transform essentially moves a set of points to a new location based upon several constant values arranged in a matrix. The implementation required for this project used two matrices, **A** (2x2) and **B** (2x1) and follows the equation:

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

Coefficients for both **A** and **B** are hard-coded into the program as immediate literals, and use the following values:

$$\boldsymbol{A} = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = \begin{vmatrix} 0.75 & 0.5 \\ -0.5 & 0.75 \end{vmatrix} \qquad \boldsymbol{B} = \begin{vmatrix} b_1 \\ b_2 \end{vmatrix} = \begin{vmatrix} 20 \\ -20 \end{vmatrix}$$

Complete equations for the resulting values are:

$$x_2 = (a_{11}x_1) + (a_{12}y_1) + b_1$$

$$y_2 = (a_{21}x_1) + (a_{22}y_1) + b_2$$

The working affine transform algorithm can be seen in Figure 5. It was designed in a right-to-left order as it is seen in the image – assembly code was written first, followed by each component of the instructions in clearly-spaced out binary and finally the equivalent hexadecimal machine code that is to be loaded into the program memory of the processor.

One requirement of the design is that, once complete, the program immediately restarts. This would typically be achieved using a branch instruction that directly sets the program counter back to zero. However, after writing the above program, it was noted that it consists of exactly 16 instructions. This means that it fits exactly into a program memory with 16 (2^4) locations. Therefore, it will automatically wrap around to the first instruction and as such means that the design does not require branch functionality in the program counter.

```
1  ////////////////////////////////////////////////////////////////////////
2  // HEX     BINARY               ASSEMBLY           REPRESENTED VALUE //
3  // 4h      OpCode(3)|Dest. address(2)|Src. address/Immediate(8)       //
4  ////////////////////////////////////////////////////////////////////////
5  // Input x1 and y1
6  0400    // 001 00 00000000      LDS     %0  SW      x1
7  1800    // 110 00 00000000      WAIT0   %0
8  0500    // 001 01 00000000      LDS     %1  SW      y1
9  1900    // 110 01 00000000      WAIT0   %1
10 // Calculate parts of x2 and y2
11 0260    // 000 10 01100000      LDI     %2  0.75    a11
12 1200    // 100 10 00000000      MUL     %2  %0      a11*x1
13 0340    // 000 11 01000000      LDI     %3  0.5     a12
14 1301    // 100 11 00000001      MUL     %3  %1      a12*y1
15 0a03    // 010 10 00000011      ADD     %2  %3      (a11*x1)+(a12*y1)
16 14c0    // 101 00 11000000      MULI    %0  -0.5    x1*a21
17 1560    // 101 01 01100000      MULI    %1  0.75    y1*a22
18 0801    // 010 00 00000001      ADD     %0  %1      (x1*a21)+(y1*a22)
19 // Display x2 and y2
20 0e14    // 011 10 00010100      ADDI    %2  20      x2
21 1e00    // 111 10 00000000      WAIT1   %2
22 0cec    // 011 00 11101100      ADDI    %0  -20     y2
23 1800    // 110 00 00000000      WAIT0   %0
```

Figure 5 - Commented affine transform program showing hexadecimal machine code followed by its binary representation, assembly code equivalent and the resulting value of the destination register

Program counter

Although initially designed and tested with absolute and relative branch functionality included, this was later removed to reduce the hardware cost. The *programCounter* module in its final state can either increment the address of the next instruction or "do nothing" – essentially stalling the processor whilst it waits for user input. Its function is determined by the *pcInc* control signal from the decoder – a value of 1 causes normal (incremental) operation.

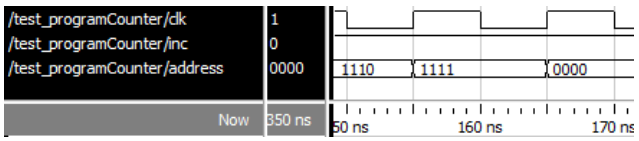

Figure 7 - ModelSim results demonstrating the wrap-around behaviour of an overflowing program counter

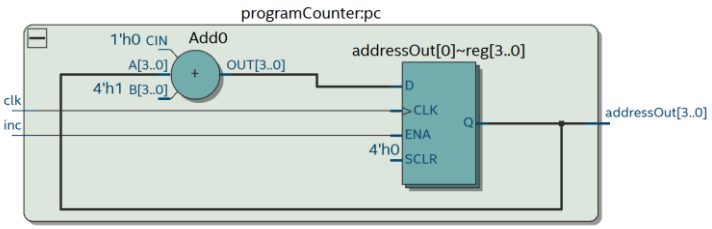

Figure 6 - Synthesised programCounter module consisting of a 4-bit adder and 4 registers

Figure 6 shows the result of the synthesis of the *programCounter* module. Simulation results in Figure 7 demonstrate how it overflows to restart the program.

Program memory

Program memory consists of a block of single-port RAM that is read from to obtain the next executable instruction. The contents of the RAM are loaded at compile-time from a file containing the affine transform algorithm in machine code, *prog.hex*, and do not change during runtime. This is achieved via the *$readmemh()* SystemVerilog function within an *initial* block.

Instructions are read out to the processor via a simple *assign* statement that selects the RAM entry to read from based on the program counter-supplied address. The complete functional code can be seen in Figure 8.

Synthesis of the program memory (Figure 9) shows a RAM block of 208 bits; that is 16 instructions of a 13-bit length.

```
// Storage for program memory
logic [(I_SIZE-1):0] memory [((1 << P_SIZE)-1):0];


// Load memory contents from .hex file
initial $readmemh("hex/prog.hex", memory);


// Asynchronously read program memory
assign instructionOut = memory[addressIn];
```

Figure 8 - Fully-functional program memory code that loads the affine transform program as machine code from prog.hex

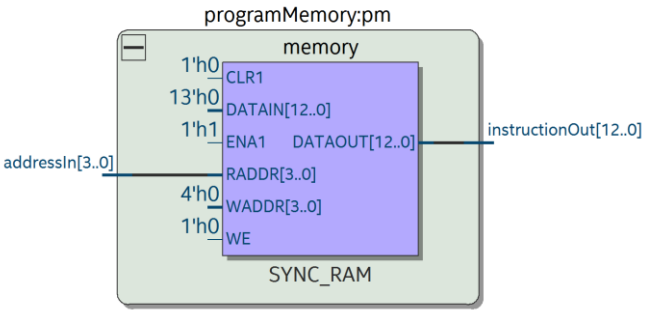

Figure 9 - Synthesised programMemory module showing a 208-bit single-port RAM block

3. General Purpose Register file design, simulation and synthesis

General purpose registers – the *registers* module – consist of a block of dual-port synchronous RAM. The addresses for each port come directly from the currently-executing instruction, in the form of the destination register operand and the lowest two bits of the 8-bit source register/immediate value operand.

```
// Register memory
logic [(N-1):0] regs [((1 << R_SIZE)-1):0];


// Synchronous write
always_ff @ (
    posedge clk
) begin
    regs[dAddressIn] <= dataIn;
end


// Asynchronous read
assign dOut = regs[dAddressIn];
assign sOut = regs[sAddressIn];
```

Figure 10 - Functional part of the registers module showing how synchronous write and simultaneous asynchronous reading is achieved

Only port A is used for writing data to the RAM. Its *writeEnable* signal is tied high such that data is always being written into it; this is not how most processors function, but it is a suitable set-up for this design considering the program required. The data that is to be written into port A comes from the output of the ALU. Therefore, for a selected register to maintain its value across instructions, the ALU must be configured to pass through input A.

Data can be read from both ports simultaneously and asynchronously; that is, a rising clock edge is not required for data to be read out of the RAM. This is achieved via simple assign statements (see Figure 10) in the HDL, much like for the program memory (described previously).

After writing the affine transform program, it was noted that just 4 general purpose registers were required for full operation. This means that the address selection buses are just 2 bits wide. No reset circuitry was included into the module since the provided top-level module, *picoMIPS4test*, does not provide a reset signal. This means that registers must be assigned a value during program execution and cannot be used to provide a value of zero by default.

The testbench for the *registers* module demonstrates both the synchronous writing and simultaneous, asynchronous reading functionalities of the model. Its simulation output can be seen in Figure 12.

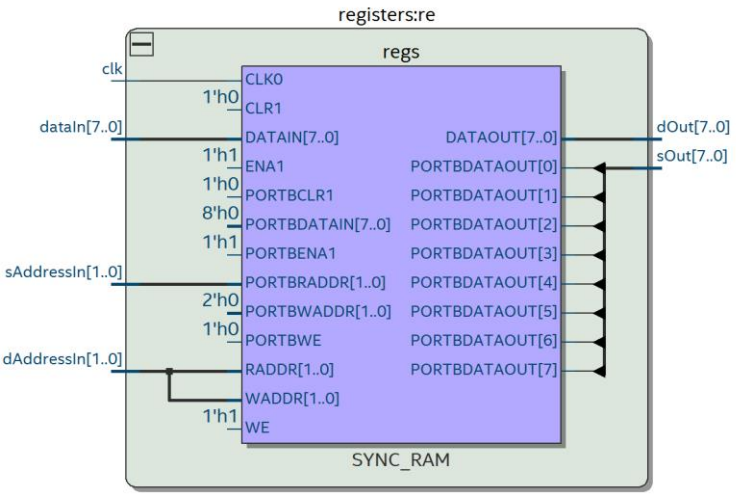

Figure 11 - Synthesised registers module, showing the general purpose registers as a 32-bit block of dual-port RAM

The synthesised model of the *registers* module is shown in Figure 11. Quartus results, as well as its HDL specification, suggest that it uses a 32-bit RAM block. This is 4 registers of 8-bit length.

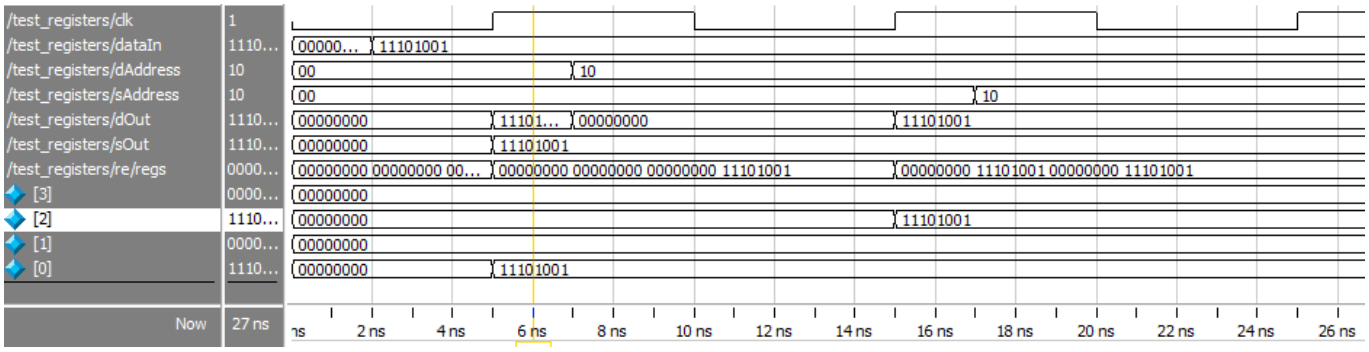

Figure 12 - Simulation results of the register module. A synchronous, always-enabled, memory write is displayed first by writing -23 into address 0. Asynchronous data reading is then shown as dOut and sOut do not wait for a clock edge to update

4. Arithmetic Logic Unit and Multiplier design

Since the affine transform algorithm requires just addition and multiplication, the design of the ALU and multiplier is very simple. The ALU can perform four operations: addition, multiplication and passthrough of either input A or input B. These are shown in Table 3, with the corresponding code in Figure 16.

ALU FUNCTION	VALUE
ALU_A	00
ALU_B	01
ALU_ADD	10
ALU_MUL	11

Table 3 - ALU function codes with their binary values

Passthrough of input A is used when displaying the value of a register or when stalling the program whilst waiting for user input. Passthrough of B is used for loading immediate values into a register during *LDI* or *LDS* instructions. When an *LDI* instruction is being executed, the immediate value is taken from the program as a literal. In the case of an *LDS* instruction, the decoder sets the *immSwitches* control line high enabling the immediate value to come from the current state of the development board switches SW 0-7. This is used when loading *x1* and *y1* in the affine transform program.

The *ALU_ADD* function is a simple one-line section of HDL that synthesises to an N-bit adder in hardware.

Multiplication is slightly more complex. This is because the operations required involve multiplying a fixed-point number – in the range -1.0 to $+(1.0 - 2^{-8})$ – by an integer – in the range -256 to +255. In addition, the result of multiplying two *N*-bit binary numbers is a *2N*-bit binary number, which is twice the size of the processor's data bus. The multiplication itself is therefore executed using an 8-to-16-bit multiplier (see *multiplyOut* in Figure 16).

-2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}
0	0	0	0	0	0	1	0	0.	1	0	0	0	0	0	0

Figure 13 - Table showing the bits to retain (shaded) and bits to discard (white) of the result of an 8-bit fixed point by 8-bit integer multiplication. Reproduced from [1]

According to the specification, the design should only retain the integer part of the fixed-point 16-bit result for further calculations, thus discarding any fractional component. Whilst reducing the accuracy of the system when compared to one that retains all of the data, it means that extracting the result of a multiplication simply requires keeping bits

N-1 to *2N-2* (bits 7 to 14 for the implemented 8-bit system, see Figure 13). This method also retains the sign of the resultant value; thus, the multiplier is able to work with negative numbers.

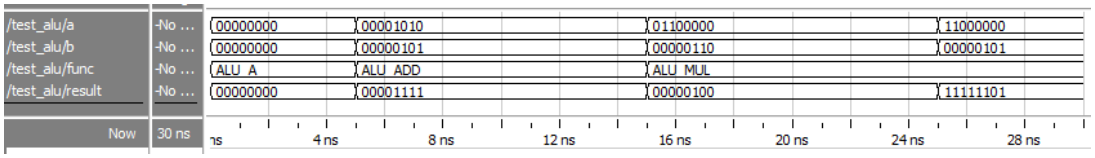

Figure 14 - Results of an ALU simulation demonstrating passthrough, addition and multiplication involving both positive and negative numbers

Testing of each ALU function was conducted in ModelSim, with the results being shown in Figure 14. The first of the multiplication tests multiplied 0.75 by 6. This equals 4.5, which the ALU correctly truncates to 4. The second test is -0.5x5. -2.5 is correctly truncated to -3.

The multiplier within the ALU synthesises to use an embedded multiplier (Figure 19) and so does not add to the hardware resource cost for this project.

```
// Generic 8-to-16-bit signed multiplier
logic signed [((2*N)-1):0] multiplyOut;
assign multiplyOut = a * b;


// Overall ALU
always_comb begin
    // Function decoder
    unique case (func)
        ALU_A:      result = a;
        ALU_B:      result = b;
        ALU_ADD:    result = a + b;
        ALU_MUL:    result = multiplyOut[((N*2)-2):(N-1)];
        default:    result = a;
    endcase
end
```

Figure 16 - The combinational code that makes up the ALU module. Parameter N is 8 for this implementation

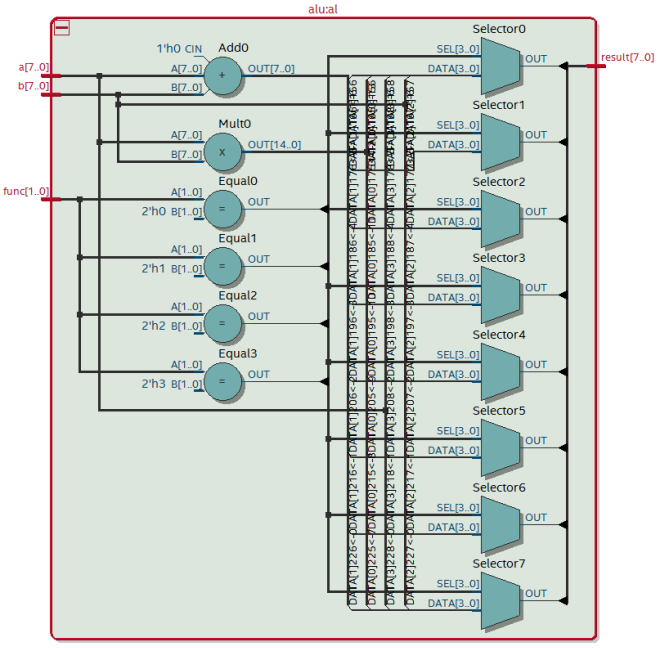

Figure 15 - Synthesised logic for the ALU that uses an embedded multiplier

5. Altera DE1-SoC implementation

As previously mentioned, the design worked upon the first attempt to program it. This was largely due to the extensive simulation phase of development, including the unit tests.

Mapping between the physical pins of the device and the ports in the topmost design module is achieved using a Quartus *.qsf* file. This file allows for simple pin mapping as well as voltage level configurations for each pin, among other settings.

The pin mappings for the clock, LEDs, switches, buttons, and seven-segment displays were found in the relevant product datasheets [3] [4] and were configured before attempting to compile or synthesise the design. One *.qsf* was created for the DE1-SoC – the board that was loaned out for the project – and a separate one for the DE2-115 – the board that the design was to be synthesised for, for resource cost calculations.

In order to test the functionality of the synthesised design in a clear and simple manner, an additional two modules were created that allowed the use of the seven-segment displays and push buttons. The first of these modules was a seven-segment display driver, which maps a 4-bit binary number to its corresponding hexadecimal representation on a single seven-segment display. The second was a modified version of the *picoMIPS4test* module, named

picoMIPS4demo, that provides connections to the seven-segment displays and push buttons. It also connects the clock to an unused LED for clarity during a demo. When using this module, the clock was initially remapped to one of the push buttons to allow for manual clocking during testing. With no bugs found, this turned out to be irrelevant, but it would have been particularly useful had there been an issue with the design.

It should be noted that the original picoMIPS4test module was retained in order to keep to the specification should the new one not be suitable during a demonstration. These can be switched between by commenting out a `define DEMO_MODE *pre-processor macro in the* cpuConfig *file and switching the top-level module in Quartus. Output (.sof) files were backed up for both the test and demo modes for rapid switching during the demonstration.*

Testing the design first involved preparing several sample inputs and calculating their corresponding outputs. This was simplified by creating a MATLAB script (Figure 17) that takes in *x1* and *y1* as signed decimal inputs and prints their 8-bit signed binary values, then calculates *x2* and *y2* and then prints both in decimal and binary formats. This massively decreased the set-up time for each test since no manual number conversion or calculation was required.

```
function affine(x1,y1)
    A = [
            0.75, 0.5;
            -0.5, 0.75
        ];

    B = [
            20;
            -20
        ];

    % Display x1 and y1 (decimal and binary)
    disp(['x1 = ' num2str(x1) ' (' num2str(dec2bin(typecast(int8(x1),'uint8'),8)) ')']);
    disp(['y1 = ' num2str(y1) ' (' num2str(dec2bin(typecast(int8(y1),'uint8'),8)) ')']);

    % Multiplications + rounding
    res1 = floor(A(1,1) * x1);
    res2 = floor(A(1,2) * y1);
    res3 = floor(A(2,1) * x1);
    res4 = floor(A(2,2) * y1);

    % Additions
    x2 = res1 + res2 + B(1);
    y2 = res3 + res4 + B(2);

    % Display x2 and y2 (decimal and binary)
    disp(['x2 = ' num2str(x2) ' (' num2str(dec2bin(typecast(int8(x2),'uint8'),8)) ')']);
    disp(['y2 = ' num2str(y2) ' (' num2str(dec2bin(typecast(int8(y2),'uint8'),8)) ')']);
```

```
>> affine(-9,92);
x1 = -9 (11110111)
y1 = 92 (01011100)
x2 = 59 (00111011)
y2 = 53 (00110101)
```

Figure 17 - A MATLAB script that performs the affine transform on a set of inputs and prints both the inputs and outputs in 8-bit signed binary format, ready for physical testing. Next to it is a sample output from the script for x1=-9 and y1=92

Each set of inputs was entered into the FPGA on the DE1-SoC board and the outputs checked against those from the MATLAB script. The process required to do so matches the pseudocode provided in the project specification, with handshaking provided by SW8. Figure 18 shows the testing in action for the set of inputs present in the sample script usage in Figure 17. Whilst it is difficult to see the status of the switches in the photos, it is possible to match up the program address (rightmost seven-segment display) with the OpCode (leftmost display), the ALU output (LEDs 0-7) and the relevant test inputs and outputs. The synthesised design performed as required for every test input entered into it.

Figure 18 - Photos of the working design. The left and right seven-segment displays show the OpCode and program counter respectively, with the leftmost LED connected to the clock. Top-left shows input x1, bottom-left is y1. Outputs x2 and y2 are on the right.

For further confidence in the design, the coefficients in the program (and the MATLAB script) were changed to the

alternative set provided in the specification and similar testing conducted. The system continued to produce the correct outputs.

6. Conclusion

At the end of the project, testing showed that the design works exactly as it is specified to. The entire process of design, implementation and testing was particularly seamless because of the use of unit testing for each module. It meant that the final implementation worked first time and as such more time could be spent on optimising various elements of the design, such as the instruction set and program.

Inspecting the compilation report from Quartus (Figure 19), after synthesising for the Altera Cyclone IV 4CE115 device, showed that the final design used 127 logic elements. The sizes of the generated RAM blocks within the *programMemory* and *registers* module – 208 bits and 32 bits respectively - were obtained from the RTL viewer window. Calculation of the "cost function" for the design resulted in a value of 134.

Cost function = 127 logic blocks + 0 multipliers + ((30*(1000/240)) bits of RAM = 134

This is approximately double that of the same design synthesised onto the DE1-SoC board used during testing but is still relatively low when compared to other, similar, designs.

Top-level Entity Name	picoMIPS4test
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	127 / 114,480 (< 1 %)
Total registers	60
Total pins	19 / 529 (4 %)
Total virtual pins	0
Total memory bits	0 / 3,981,312 (0 %)
Embedded Multiplier 9-bit elements	1 / 532 (< 1 %)
Total PLLs	0 / 4 (0 %)

Figure 19 - Quartus synthesis report for the design on the Cyclone IV E board, showing the use of 127 logic elements and 1 embedded multiplier

A further possible optimisation to the design could be to reduce the size of the instruction set. This would simplify the decoder and so reduce its resource usage. However, in order to do so, additional logic may have to be implemented to initialise the general purpose registers.

References

[1] D. T. J. Kazmierski, "SystemVerilog Design of an Embedded Processor," 7 February 2017. [Online]. Available: https://secure.ecs.soton.ac.uk/notes/elec6234/cswk/instructionspm.pdf.

[2] MathWorks, "Affine Transformation," [Online]. Available: https://uk.mathworks.com/discovery/affine-transformation.html.

[3] Terasic, "DE1-SoC User Manual," 8 April 2015. [Online]. Available: https://www.terasic.com.tw/cgi-bin/page/archive_download.pl?Language=China&No=836&FID=ae336c1d5103cac046279ed1568a8bc3.

[4] Terasic, "DE2-115 User Manual," 2012. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/portal/dsn/42/doc-us-dsnbk-42-1404062209-de2-115-user-manual.pdf.