**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.

+ REPORT ORGANISATION

1. **Instruction format, decoder design, program memory and program counter**

The overall design of the system can be seen in FIG. *N*, *A\_SIZE* etc. are parameters, defined in a configuration file, *cpuConfig.sv*, that determine the widths of the various data and address buses. Parameter values are passed down through each of the sub-modules of the system. This enables for very quick and simple modifications to the entire design should the number of general purpose registers increase, for example.

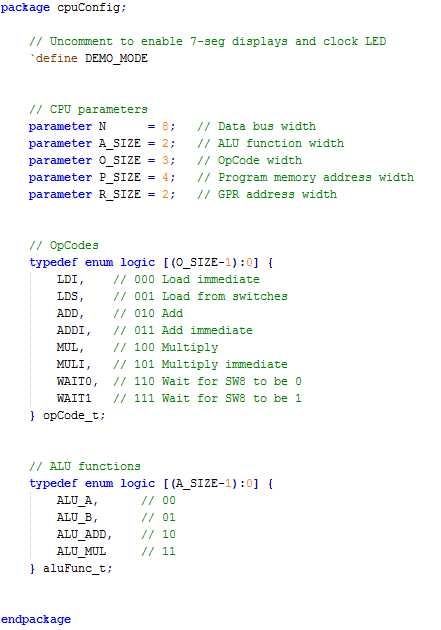


Figure 1 - Configuration file, cpuConfig.sv, that defines the system parameters as well as OpCodes and ALU functions

+ OVERALL DIAGRAM

In addition to the system parameters, the configuration file (Figure 1) 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.

**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 dest. register |
| LDS | 001 | Load Switches | Load value of switches into dest. register when SW8 is 1 |
| ADD | 010 | Add | Add source register to dest. register |
| ADDI | 011 | Add Immediate | Add immediate value to dest. register |
| MUL | 100 | Multiply | Multiply dest. register by source register |
| MULI | 101 | Multiply Immediate | Multiply dest. register by immediate value |
| WAIT0 | 110 | Wait SW8 0 | Wait for SW8 to be 0 – display dest. register |
| WAIT1 | 111 | Wait SW8 1 | Wait for SW8 to be 1 – display dest. 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 such a case, it is set depending on the state of SW8 (used for the demonstration).

+ SIM RESULTS

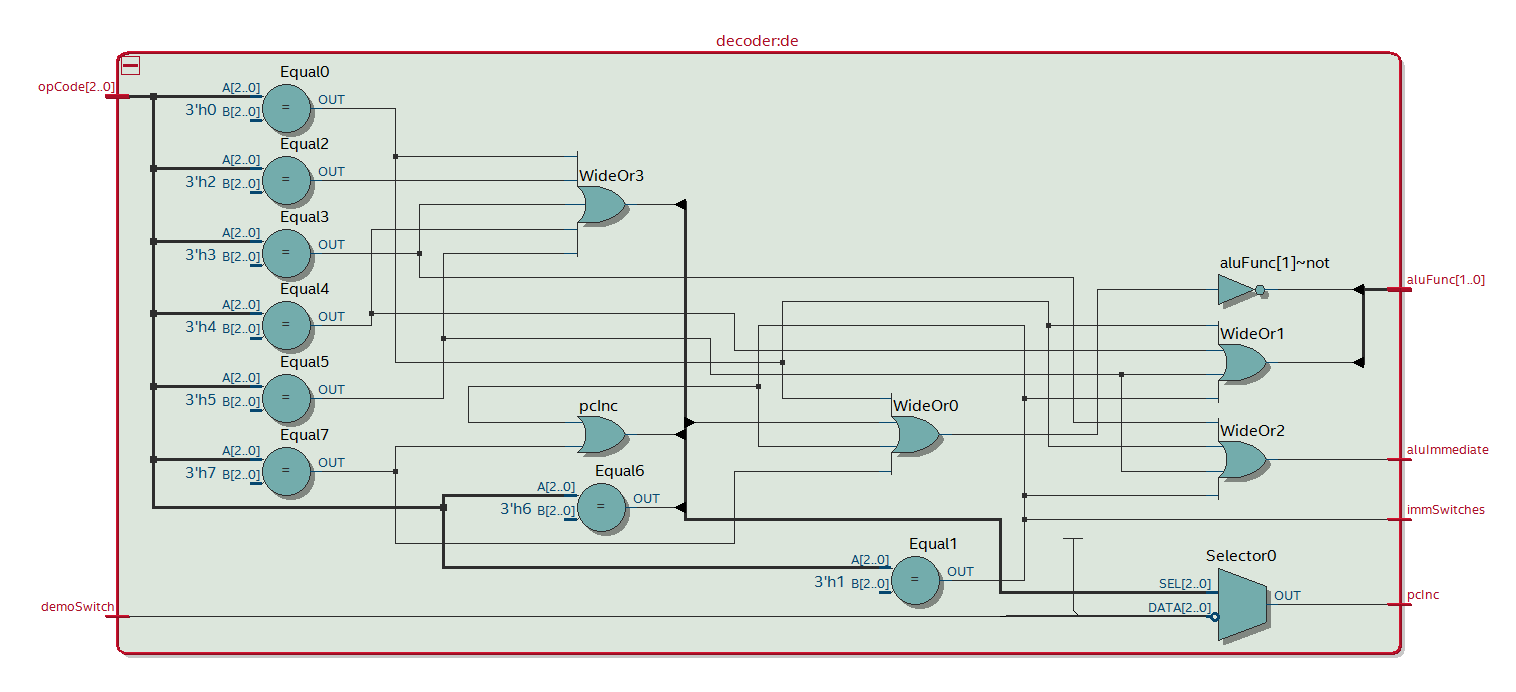


Figure 2 - Synthesised logic of the decoder module that is made up of basic gates, comparators and a multiplexer

**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 a number of constant values arranged in a matrix. The implementation required for this project used two matrices, **A** (2x2) and **B** (2x1) and follows the equation:

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

Complete equations for the resulting values are:

Realising that each matrix coefficient is used only once is what enabled the instruction format to be so simple, since a register holding its value can be overwritten with the result of an ALU operation. Following on from that, the only operations required by the ALU are addition and multiplication (as well as operand pass-through), thus making its design very simple, as discussed in a later section.

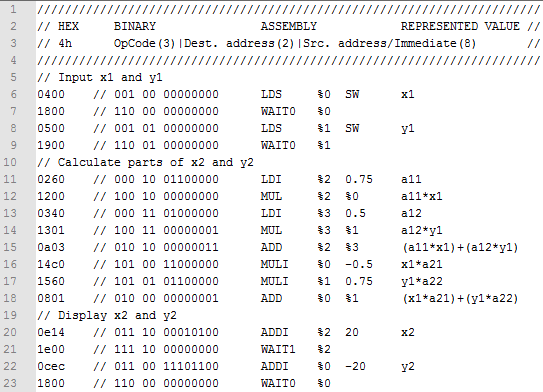


Figure 3 - Commented affine transform program showing hexadecimal machine code followed by its binary representation, assembly code equivalent and the resulting value of the destination register

The working affine transform algorithm can be seen in Figure 2. 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 (24) 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.

**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 4 shows the result of the synthesis of the *programCounter* module. Simulation results in Figure 3 demonstrate how it overflows to restart the program.

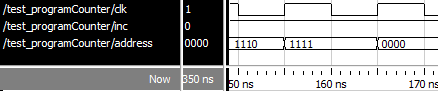


Figure 4 - ModelSim results demonstrating the wrap-around behaviour of an overflowing program counter

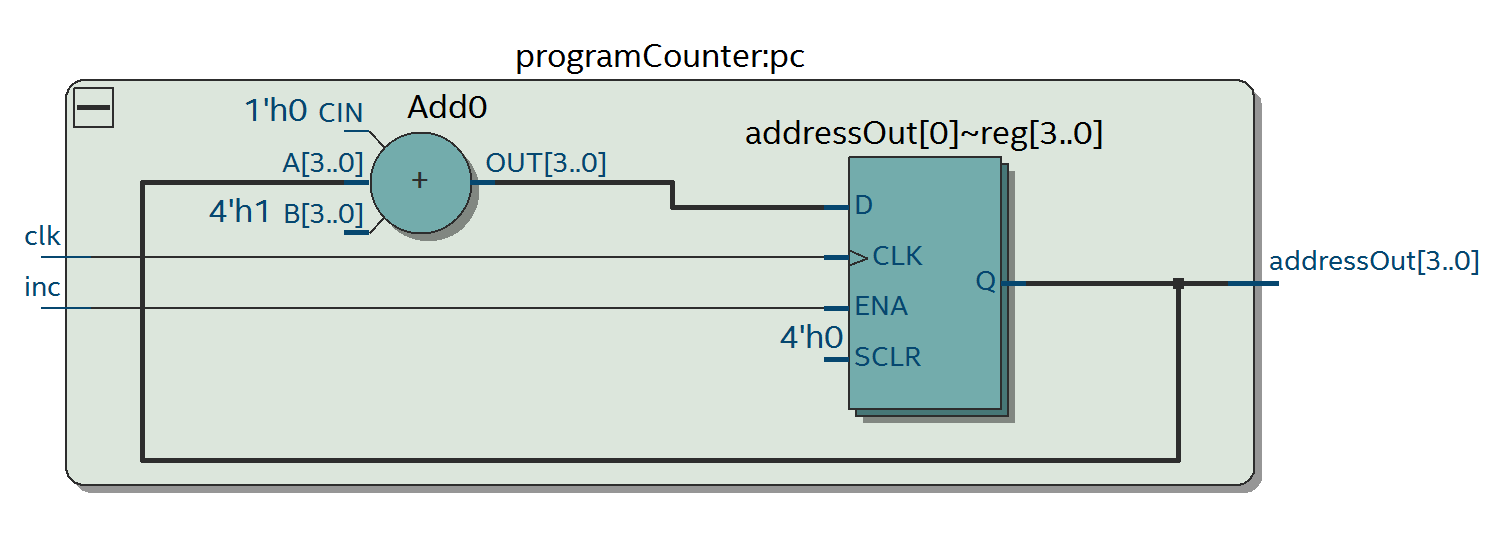


Figure 5 - Synthesised programCounter module consisting of a 4-bit adder and 4 registers

**Program memory**

Program memory consists of a block of single-port RAM that is read from in order 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. By default, the *writeEnable* signal for the RAM is kept low.

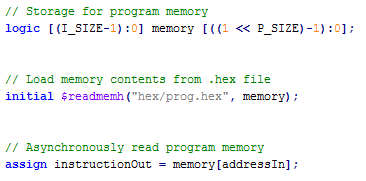


Figure 6 - Fully-functional program memory code that loads the affine transform program as machine code from prog.hex

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 passed into the *programMemory* module. The complete functional code can be seen in Figure 5.

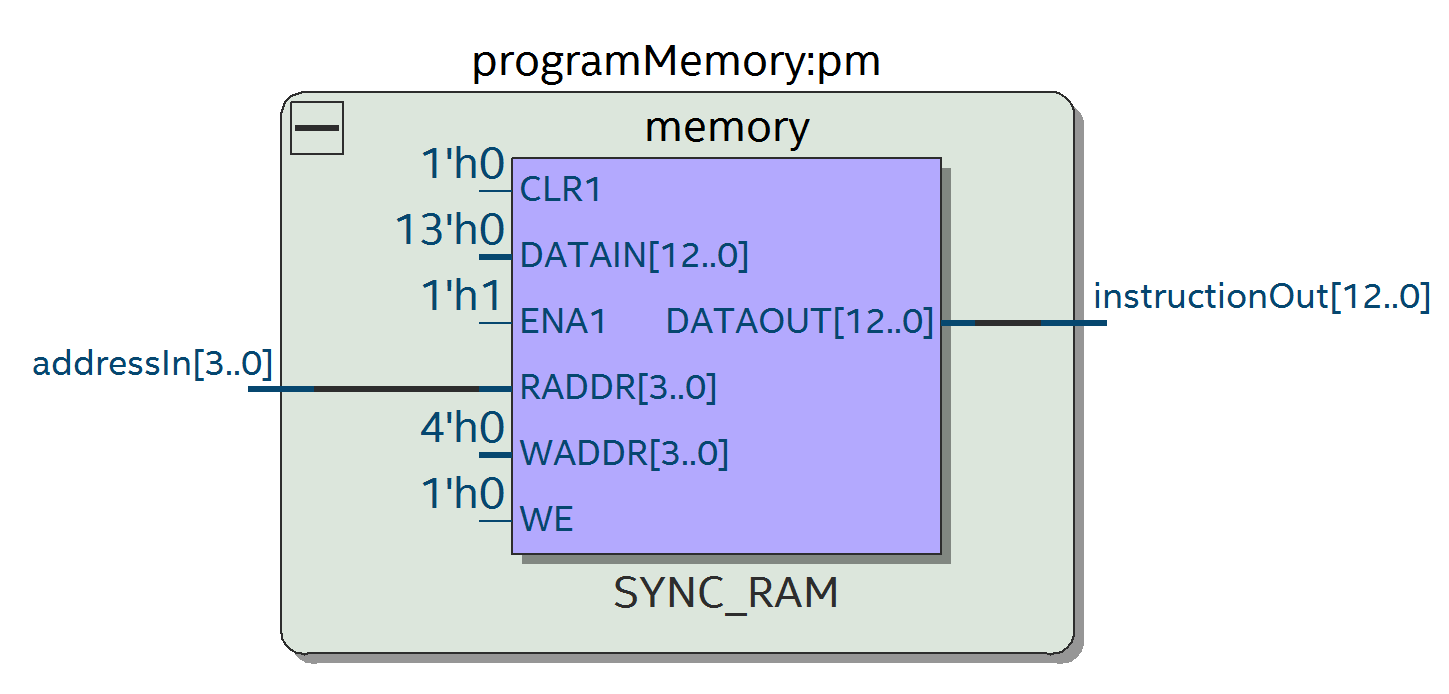


Figure 7 - Synthesised programMemory module showing a 208-bit single-port RAM block

Synthesis of the program memory (Figure 6) shows a RAM block of 208 bits; that is 16 instructions of a 13-bit length.

1. **General Purpose Register file design, simulation and synthesis**

As this and this did this

+ 4x8 = 32-bit RAM for GPR

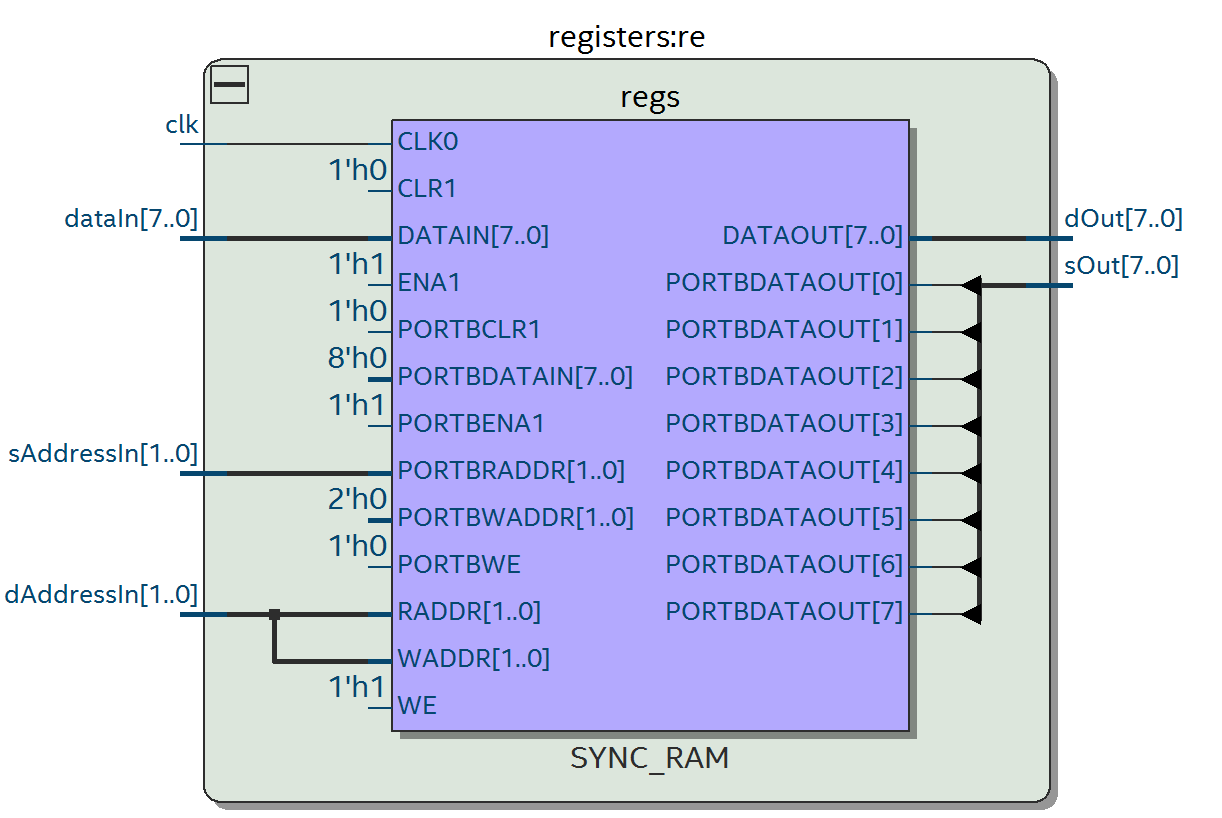


Figure 8 - Synthesised registers module, showing the general purpose registers as a 32-bit block of dual-port RAM

1. **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 8.

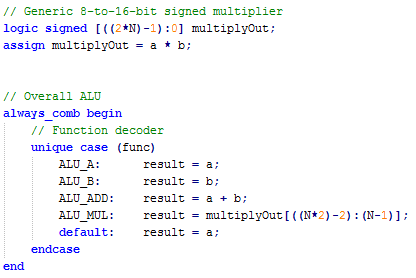


Figure 9 - The combinational code that makes up the ALU module. Parameter N is 8 for this implementation

|  |  |
| --- | --- |
| 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.

A picture containing sky

Description generated with high confidenceMultiplication 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 8).

Figure 10 - 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 9). This method also retains the sign of the resultant value; thus, the multiplier is able to work with negative numbers.

Testing of each ALU function was conducted in ModelSim, with the results being shown in Figure 10. 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.



Figure 11 - Results of an ALU simulation demonstrating passthrough, addition and multiplication involving both positive and negative numbers

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

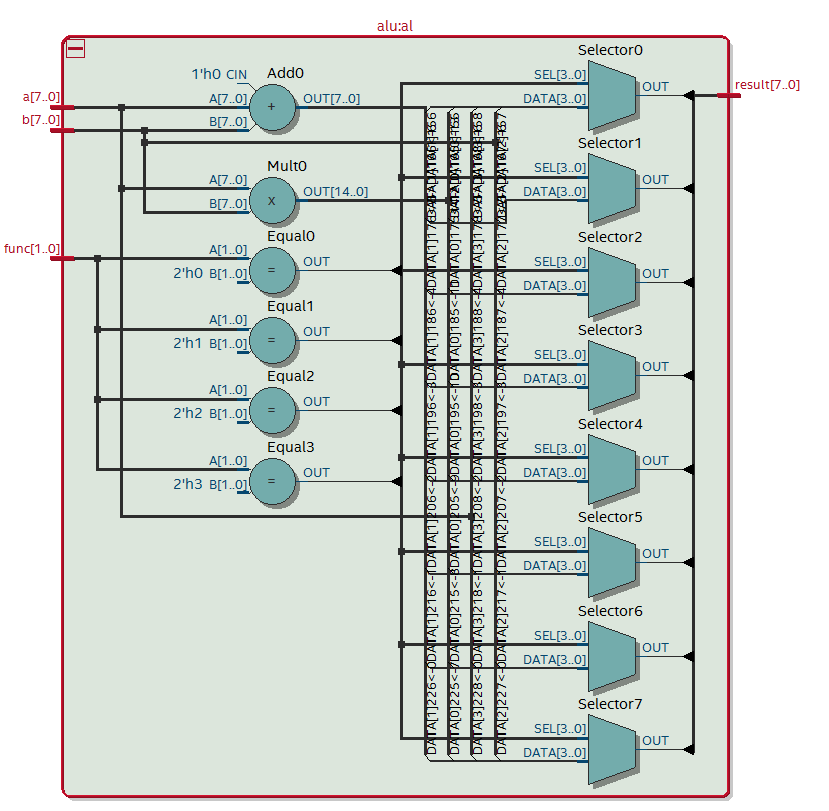


Figure 12 - Synthesised logic for the ALU that uses an embedded multiplier

1. **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. Each board required its own Quartus project as well.

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 12) 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.

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 13 shows the testing in action for the set of inputs present in the sample script usage in Figure 12. 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.

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.

Figure 13 - 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

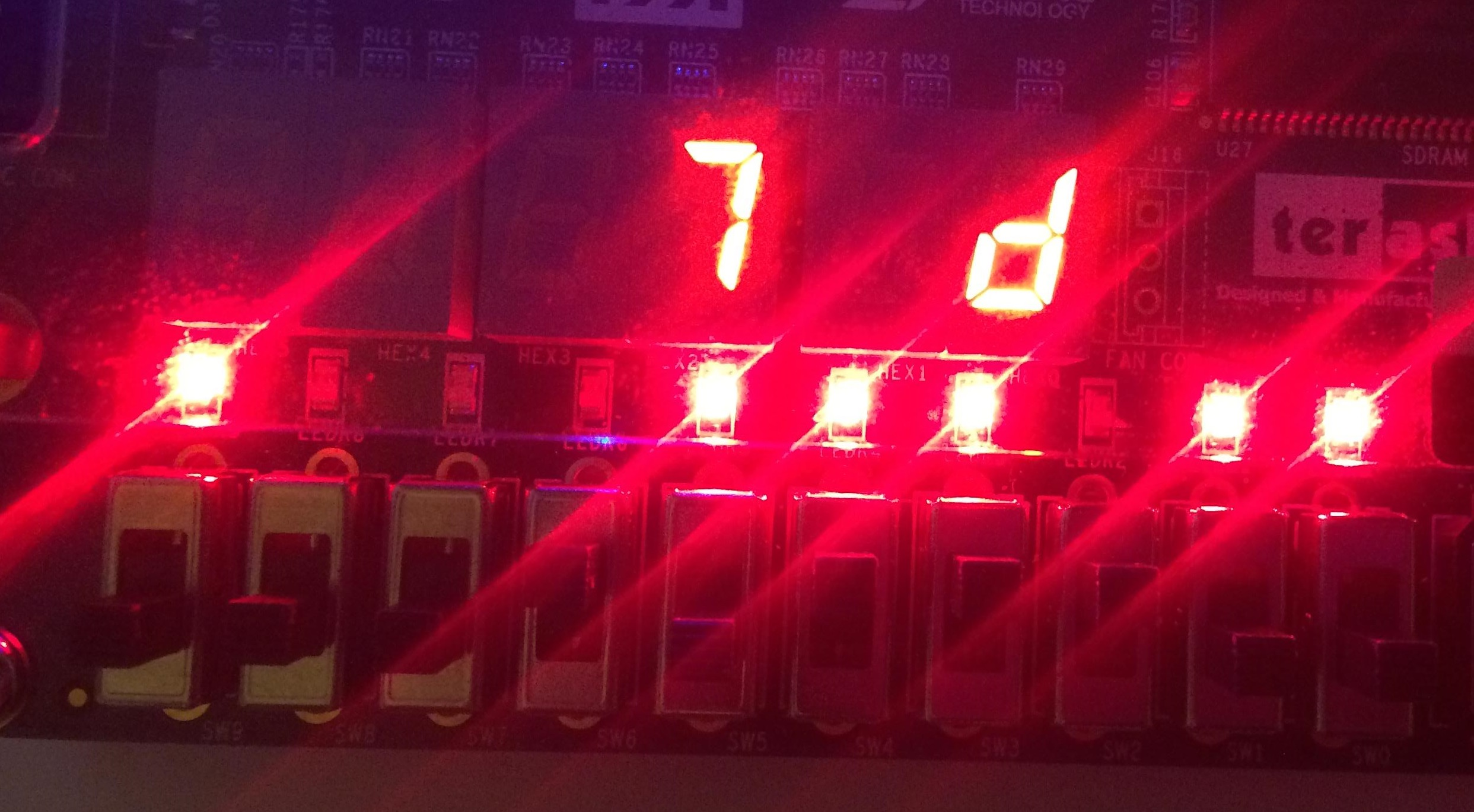
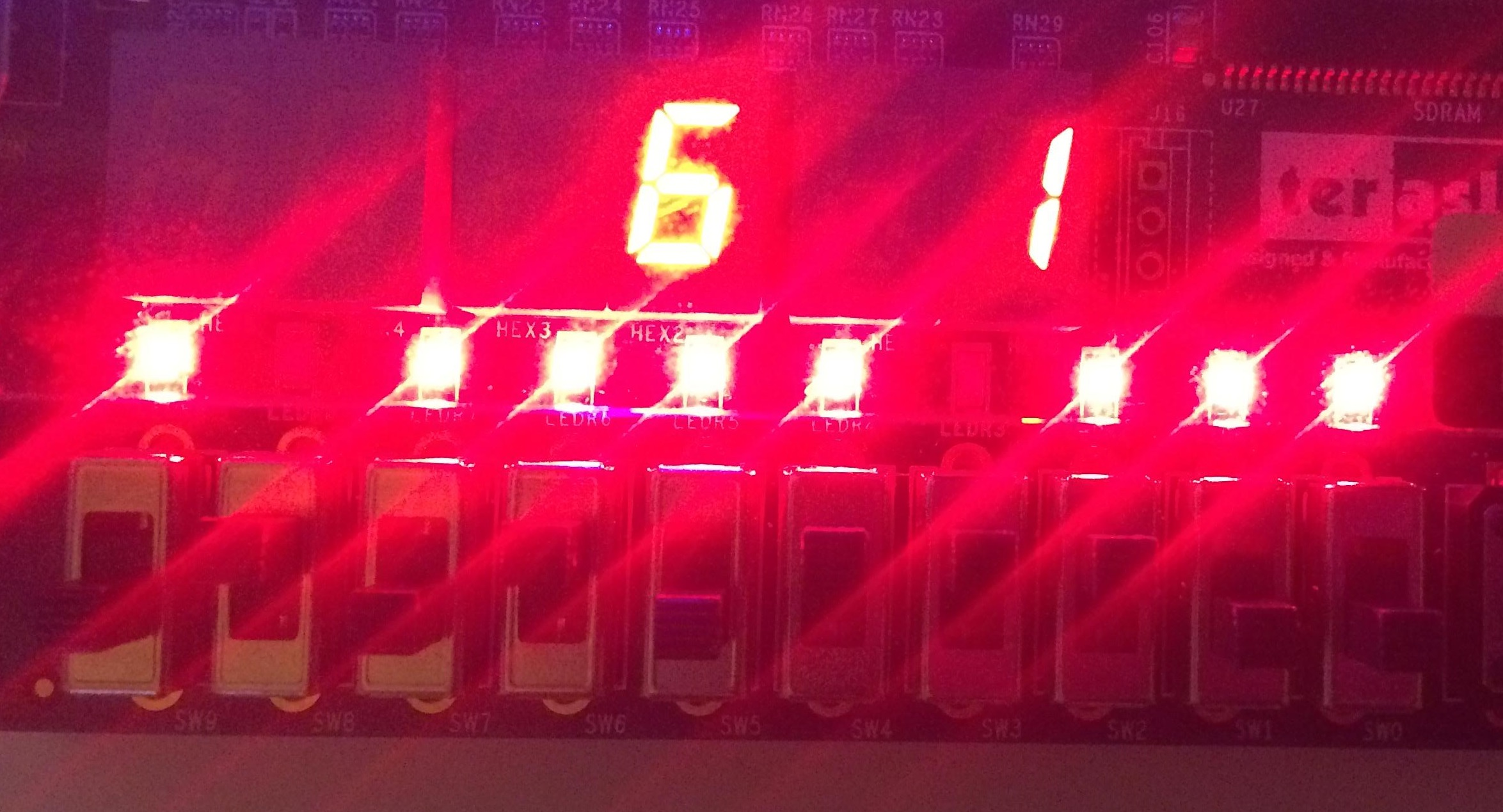
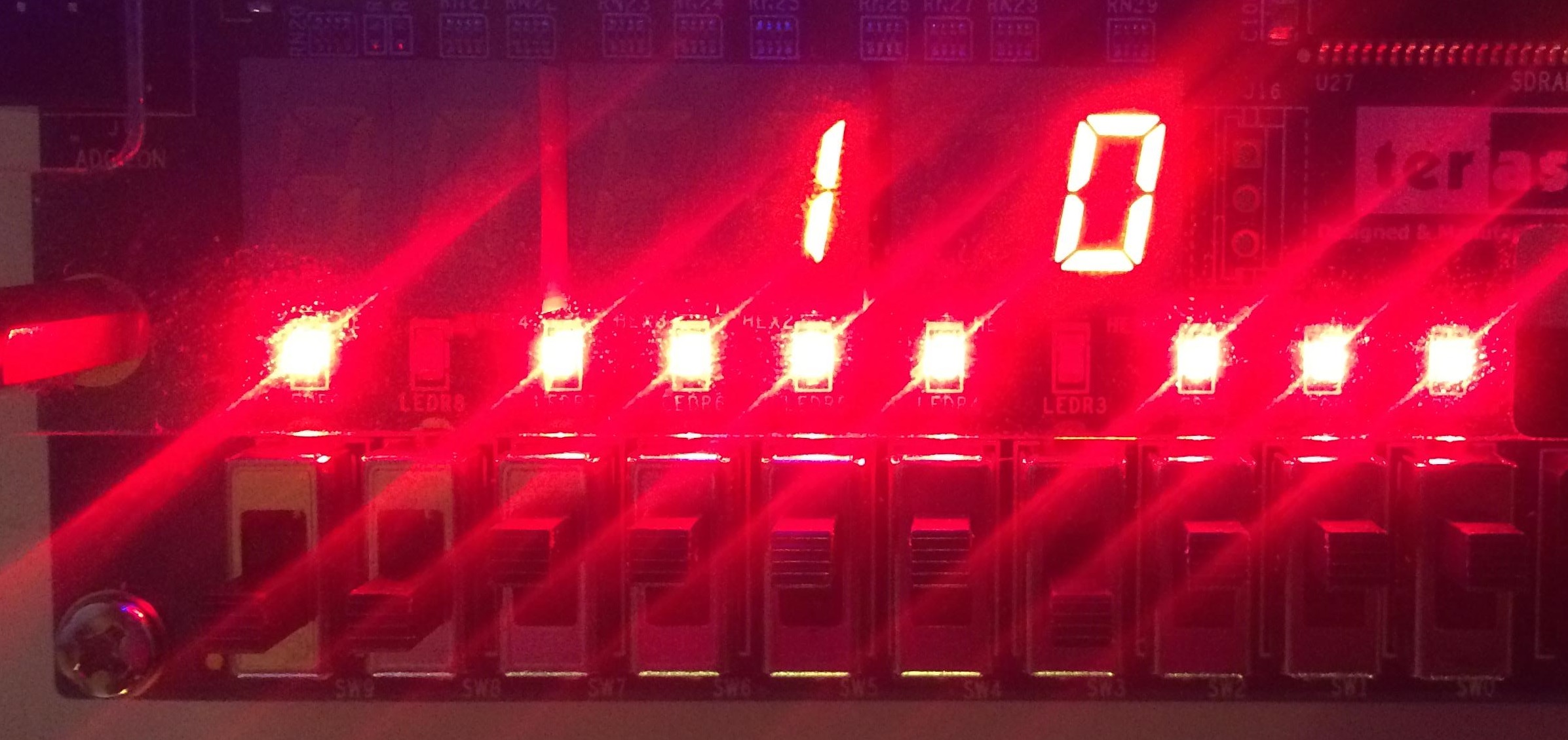
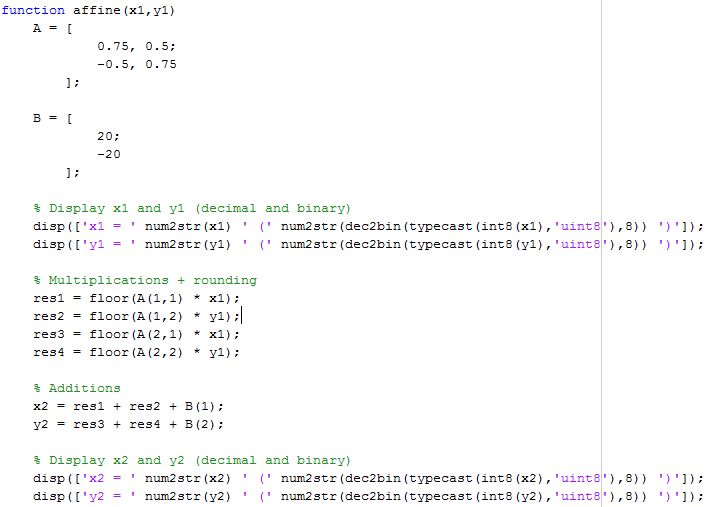


Figure 14 - Photos of the working design. The left and right seven-segment displays show the OpCode and program counter respectively, with the leftmost LED connected to the clock. Top-left shows input x1, bottom-left is y1. Outputs x2 and y2 are on the right.

1. **Conclusion**

Concluding thoughts

+ cost function = 127 + 0 + (30x(1024/240)) = 134

# **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. |