ELEC6016 - FPGA Synthesis of a picoMIPS

Henry Lovett
hl13g10
MEng Electronic Engineering
Tutor: Prof. Mark Zwolinski

26th April, 2014

Abstract

This report documents the design and test of a picoMIPS processor. The processor is able to calculate an affine transform with predefined constants. A small instruction set of ten instructions was implemented with an eight bit opcode format. The processor is a register-accumulator based architecture. It is written in System Verilog and is intended for the Altera Cyclone IV FPGA.

1 Introduction

This assignment is to design and test a processor capable of executing an affine transform of a vector. The processor should be optimised in terms of size, with no performance requirements. The cost function, seen in equation (1), should be minimised. The data is given in bytes and must be read from the switches and the result should be displayed on the LEDs. A handshaking protocol is also defined and must be followed. A processor is defined as having a distinct control, datapath and memory with an instruction set that is executed.

$$\text{Cost} = \text{ Number of Logic Elements} + 30 \times \text{Kbits of RAM} \tag{1}$$

A number of designs and approaches were investigated, including register-register and single cycle designs. The final design is a register-accumulator,

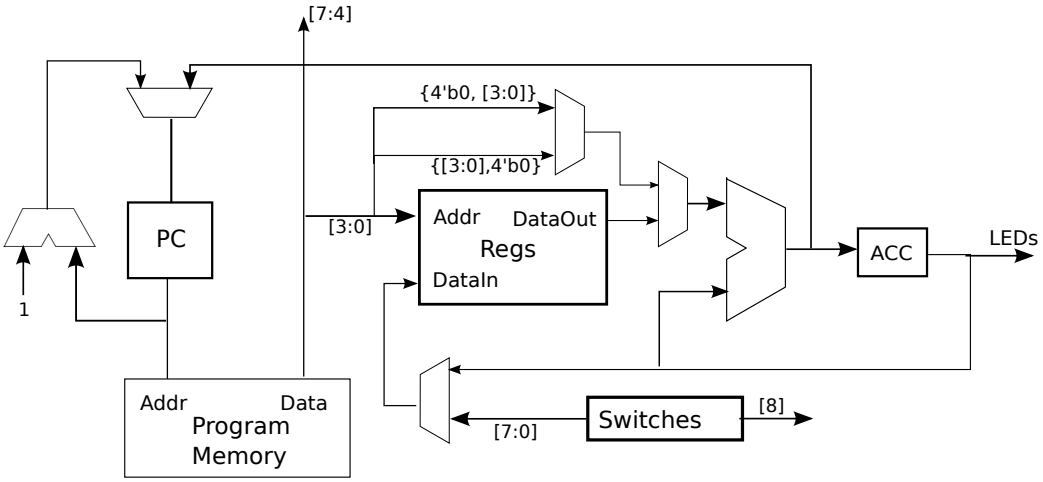

Figure 1: Block Diagram of the architecture. Control unit and signals have been omitted for clarity.

multi-cycle processor which utilises embedded multipliers and SRAM available on the target FPGA. It is able to conduct the affine transform using data set 2, shown in equation (2), and satisfies the handshaking protocols.

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} 0.5 & -0.875 \\ -0.875 & 0.75 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} + \begin{bmatrix} 5 \\ 12 \end{bmatrix} \tag{2}$$

Figure 1 shows the architecture of the processor, excluding the control and control signals.

Each module of the processor is documented in this report, with discussion on the design, test and synthesis of the code. Additional extras are also discussed, including a simple assembler (see Section 4 and extra logic added for debugging and the demonstration (see Section 9.

2 Instruction Set

In total, ten instructions are implemented in the design. They are fully documented in table 1.

Table 1: Instruction Set and their operations

Instruction	Acronym	Operation
Store Switches	STSW Rn	Rn ← Switches[7:0]
Store Accumulator	STACC Rn	Rn ← Acc
Wait while 0	WAIT0	if(Switches[8] == 1) Pc++
Wait while 1	WAIT1	if(Switches[8] == 0) Pc++
Absolute Jump	JMPA Imm	PC ← Acc + Imm
Load Upper Immediate	LUI Imm	Acc ← Imm, 4'b0000
Add Immediate	ADDI Imm	Acc ← Acc + Imm
Add Register	ADD Rn	Acc ← Acc + Rn
Multiply Register	MULT Rn	Acc ← Acc * Rn
Pass Register	PASSA Rn	Acc ← Rn

2.1 Arithmetic Instructions

Only three instructions implemented conduct arithmetic operations, `ADD`, `ADDI` and `MULT`. These are all signed arithmetic and are the only functions needed to compute the affine transform. Add immediate (`ADDI`) is the addition of a four bit unsigned immediate. No subtract immediate instruction was implemented to reduce the size of the ALU and no sign extension is needed for the immediate.

2.2 Accumulator and Register Instructions

The switches provide the main data input to the processor. A store switches instruction (`STSW`) is implemented to directly store to a register. Load upper immediate (`LUI`) is used to load constants from the program memory to the accumulator. Used in conjunction with the `ADDI` instruction, any 8 bit value can be loaded to the accumulator in two instructions. This can then be used to make a negative number to use for an arithmetic function.

The register-accumulator design, although allows for only one operand in the instruction, must also include extra instructions to move data. The instructions needed are a pass to accumulator (`PASSA`), to write a value from a register to the accumulator, with no modification. A store accumulator instruction (`STACC`) is also needed to write back the data to a register.

2.3 Control Instructions

The affine transform does not involve any decisions based on the data. The only required control flow is during the handshaking protocol. As this is reliant on the value of Switches[8], this is accomplished by two instructions, `WAIT0` and `WAIT1`. These will only allow the program counter to increment if the value of Switches[8] is correct. This means no flags need to be used in the design, and no conditional branches.

An absolute jump instruction is also implemented (`JMPA`). It is used once the affine transform is complete to return to the start of the loop. This replaces the value of the program counter with what is stored in the accumulator plus a four bit immediate. It means that the processor can jump to any eight bit memory location in two instructions.

2.4 Instruction Format

A total of ten instructions have been implemented. The ten instructions and the use of a four bit immediate, lends the instruction set to use an eight bit instruction format. This is summarised in table 2. A simple instruction format lends itself to simple decoding. A maximum of sixteen general purpose registers (assuming no dummy register) can be used.

3 Controller

The controller is the decoding sequential logic of a processor. It decodes the opcode into the correct signals for the multiplexers, ALU operations and enable signals.

3.1 Design

The controller used is sequential. This is due to the program memory and registers being sequential memory (see sections 4 and 5). Three states are implemented, `Fetch`. `Read` and `Execute`. During the `Fetch` cycle, the program

Table 2: The instruction format used

Bit	7	6	5	4	3	2	1	0
Use	Opcode				Immediate / Register			

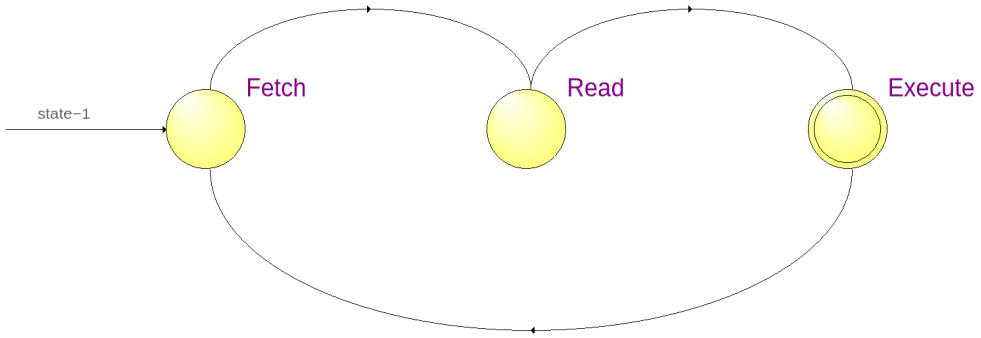

Figure 2: State transition diagram for controller

memory is accessed with a new address. This then stores the instruction at the output of the program memory. `Read` is needed to allow the read of the registers. By the `Execute` cycle, both the register value and the instruction are available. Any instruction can then be conducted at this point. Some instructions (`LUI`, `STSW`, `WAIT0`, `WAIT1`, `JMPA`, `STACC`) do not necessarily need three cycles to complete as the do not use a register value. However, keeping all instructions to execute in the same number of cycles reduces the complexity of the decoder. Figure 2 shows the state transition diagram generated by the Quartus synthesis tool.

By careful allocation of the opcode values, the controller can be greatly simplified. Each of the instructions in table 1 must have a unique four bit value. There are eight control signals for the ALU operation, multiplexer selection or write enable for a register. The combinational control signals do not rely on the state, and so can be reduced to a function between the bits of the opcode. The sequential control signals, however, must only be asserted during one cycle of an instruction and are therefore dependent on the state.

Table 3 shows all the control signals between the datapath and the controller. All signals default to 0. The instructions which assert the signals are listed in table 3. By grouping these instructions together, the control signals can then be reduced to simple logic operations. The assignment of opcodes to the instructions is seen in table 4. The most simple optimisation done is allowing the bottom two bits of the instruction to be the ALU operation. In this case, 0 will be a pass through instruction, 1 will be an add and 3 will be a multiply. As 2 is not used by any instruction, this can be set to anything in the ALU. This removes all need for any logic to decode the ALU function.

Table 3: Control signals and the instructions where they are asserted (are logic 1)

Type	Signal Name	Instruction(s)
Combinational	PcSel	JMPA
	ImmSel	LUI
	Op1Sel	JMPA, LUI, ADDI
	WDataSel	STSW
	AluOp	LUI, JMPA, PASSA, ADD, ADDI, MULT
Sequential	PcWe	WAIT0 WAIT1
	RegWe	STSW, STACC
	AccWe	PASSA, ADD, ADDI, MULT, LUI

Table 4: Opcode allocation to the instructions

		Opcode[3:2]			
		00	01	11	10
Opcode[1:0]	00	STSW	-	PASSA	LUI
	01	STACC	JMPA	ADD	ADDI
	11	WAIT0	WAIT1	MULT	-
	10	-	-	-	-

For the *ImmSel* signal, it requires more logic. Table 5 shows a K-Map for the *ImmSel* signal, replacing the instructions with values. A lot of instructions are not affected by this control signal as it only affects the instructions using the immediate. This results in a very simple logic expression of !(*Opcode*[0]|*Opcode*[1]), i.e. the `NOR` between the lowest two bits of the opcode. This can be done for all the signals, except *PcWe* as this is reliant on the value for of *Switches[8]*. The sequential signals can be minimised in a similar fashion, but are only asserted in the execute state. Listing 1 shows all the signals with their reduced decoding.

Listing 1: Reduced Control signal allocations

```
1 assign AluOp = alu_functions_t'({Op[1], Op[0]});
2 assign WDataSel = ~(Op[0] | Op[1]);
3 assign Op1Sel = (Op[2]) ^ (Op[3]);
4 assign ImmSel = ~(Op[0] | Op[1]);
5 assign PcSel = (OpCode == JMPA) ? PcJmp : PcInc;
```

Table 5: K-map of the *ImmSel* Control Signal - X is don't care

		Opcode[3:2]			
		00	01	11	10
Opcode[1:0]	00	X	X	X	1
	01	X	X	X	0
	11	X	0	X	X
	10	X	X	X	X

```
always_comb
begin
  AccWe = 1'b0;
  RegWe = 1'b0;
  if (state == Execute)
  begin
    AccWe = Op[3];
    RegWe = ~(Op[3] | Op[2] | Op[1]);
  end
end
```

3.2 Testbench

To test the controller, each instruction was passed in turn. During the execute cycle of the controller, the relevant outputs are checked. Assertions are used to verify the expected behaviour. If an assertion fails, an error counter is incremented. By the end of the simulation, the error counter should be zero. Figure 3 shows the waveform of the simulation. All inputs and outputs are shown, along with the internal state of the controller and the error counter. The error counter is at zero at the end of the simulation, showing that this testbench passes.

3.3 Synthesis

The synthesis of the controller is seen in figure 4. The largest part of the controller is the state machine. All of the discussed simplifications to the decoding can be seen in this diagram. Interestingly, the dependence of the write enable signals is realised by a multiplexer. A simpler solution could be an `AND` gate. If the state is encoded using one hot coding, then no extra logic would be required to use an `AND` gate. However, the synthesis tool probably

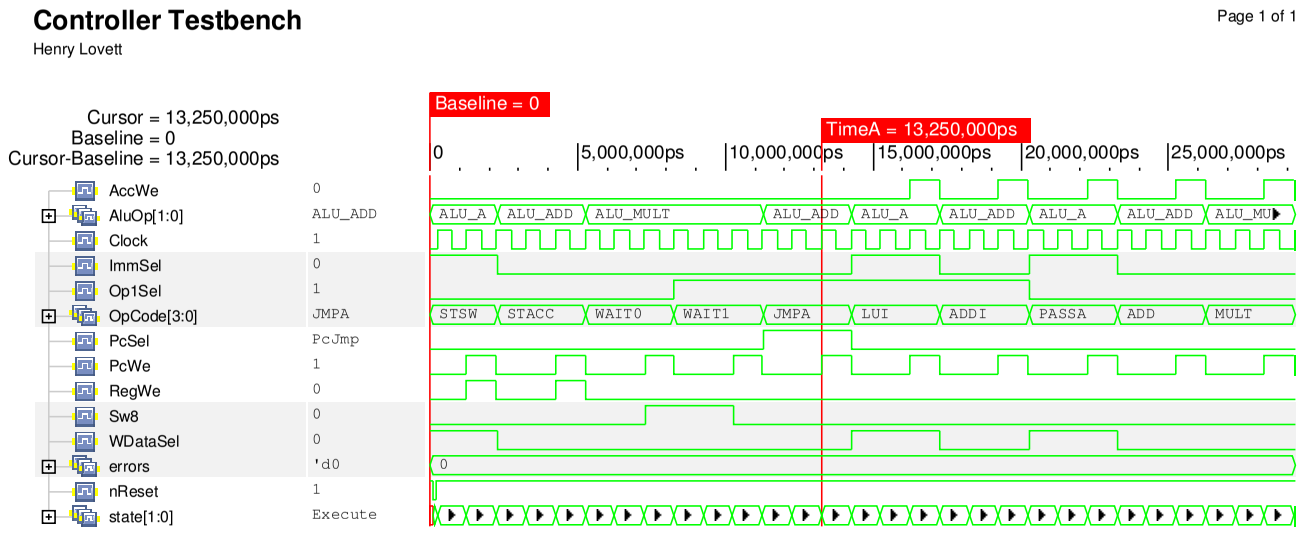

Figure 3: Controller simulation waveform

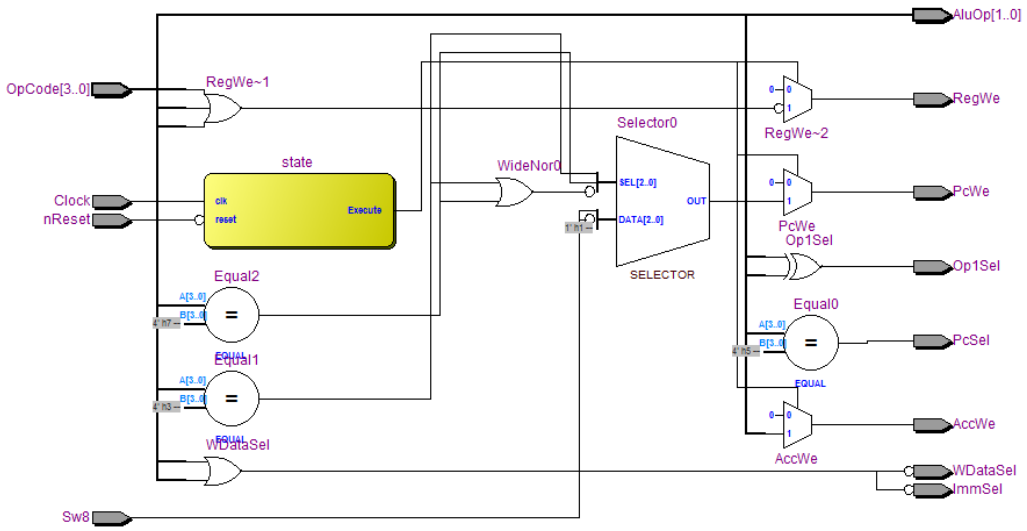

Figure 4: Synthesis of the controller

takes into account the layout of the device and utilises the logic blocks as much as possible.

4 Program

4.1 Program Counter

The program counter is a register storing the address of the current instruction. It is used to access the program memory to retrieve instructions. The implementation involves a six bit write enable register with a multiplexed input. The inputs to the multiplexer are the output of the ALU and an incremented value of itself. The increment is used to progress the program in normal operation, where as the ALU output is used to jump to any location by use of the `JMPA` instruction. Only six bits are used as the program implemented is 46 instructions long.

4.2 Program Memory

The program memory is implemented as sequential read only memory (ROM). It takes a clock cycle to output the data at the given address. This is advan-

tageous as the Cyclone IV has a large amount of synchronous RAM, which is far more efficient that using logic elements. However, this then requires a multi-cycle control module.

4.3 Program

The program is in two sections, the set up and the loop. The set up loads all the constants into the registers. The loop then conducts the handshaking and the affine transform.

4.3.1 Set Up

The initial part of the program loads in the matrix coefficients into the registers. The coefficients of the **A** matrix are represented by fixed point notation. This means that the values must be 128 times bigger than the decimal equivalent. Listing 2 shows the code to load all the constants to memory.

Listing 2: Loading initial constants

```
LUI 4; load constants
ADDI 0
STACC 0 ; a11 = 0.5 = x40
LUI 9;
ADDI 0
STACC 1 ; a12 = -0.875 = x90
LUI 9;
ADDI 0
STACC 2 ; a21 = -0.875 = x90
LUI 6
ADDI 0
STACC 3 ; a22 = 0.75 = x60
LUI 0;
ADDI 5
STACC 4 ; b1
LUI 0;
ADDI 12
STACC 5 ; b2
```

4.3.2 Loop

An initial handshake is done to load the values of x_1 and y_1 into memory. This is achieved by using the `WAIT`n instructions to control the program flow,

and the `STSW` to write the switches to a memory location. The whole process is seen in listing 3.

Listing 3: Handshaking protocol for switch reading

```
WAIT0 ;wait while sw8 == 0 ; LOOP
STSW 6 ; store switches to R6 ; X
WAIT1 ; Wait while sw8 == 1
WAIT0 ; Wait while sw8 == 0
STSW 7 ; store switches to R7 ; Y
WAIT1
```

The affine transform is shown in equation (3). By multiplying the matrix out, x_2 and y_2 can be calculated by equations (4) and (5) respectively. Listing 4 shows the instructions used to calculated x_2. The y_2 value is calculated in the same way, using different source registers.

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{3}$$

$$x_2 = a_{11} \times x_1 + a_{12} \times y_1 + b_1 \tag{4}$$

$$y_2 = a_{21} \times x_1 + a_{22} \times y_1 + b_2 \tag{5}$$

Listing 4: Affine transform for calculating x_2

```
PASSA 6 ;transform - load x1 to acc
MULT 0 ; a11 * x1
STACC 8 ; store to r8
PASSA 7 ; y1 to acc
MULT 1 ; acc * a12
ADD 8 ; (y1 * a12) + (x1 + a11)
ADD 4 ; acc + b1
STACC 9 ; store to r9
PASSA 6 ;transform - load x1 to acc
MULT 2 ; a21 * x1
STACC 8 ; store to r8
```

Once the transform is complete, a second handshake is done to read back the result. The program then jumps to the start of the loop, ready for the next calculation. This code is seen in listing 5.

Listing 5: End of loop code to display results and return to the beginning of the loop

```
PASSA 9 ; display x2
WAIT0
PASSA 10 ; display y2
WAIT1
LUI 1 ;return to loop - load address to acc
JMPA 2 ; jump to address 19
```

A basic assembler is implemented to aid development. It is capable of reading the SystemVerilog package containing the definition of the opcodes. It then assembles the program into the hex values. Concurrent development of programs and processor is then greatly simplified as all definitions are obtained from the same package. It is written in python and accepts command line arguments to compile a asm file. A help prompt is also given when `-h` is given as an argument.

The testing of the program counter is done as a part of the datapath test and is discussed in section 7.

4.4 Synthesis

The synthesised logic of the ROM can be seen in figure 5. It shows that there is some redundancy in the logic as the standard cells contain write circuitry. Here, constants are used to disable the functionality, as these are fabricated, rather than built out of logic elements.

5 Registers

The allocation in the instruction set allows for up to sixteen general purpose registers. In the program, discussed in section 4, only eleven registers are used. Six are used to store the constants, two for the initial vector, two for the result and a temporary register. To save RAM, only the required registers are implemented. This still requires a four bit address and attempting to address the registers above the valid range would result in undefined behaviour.

The registers were implemented using the Synchronous RAM. This, at the expense of performance, utilised the on chip SRAM blocks. The design was parameterized to allow the data and address width and number of registers to be easily changed.

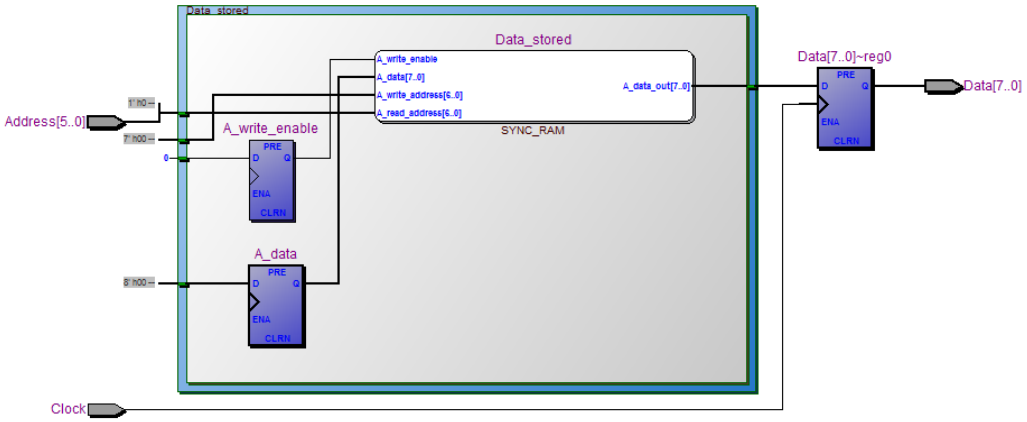

Figure 5: Synthesis of the Program ROM module

5.1 Testbench

A task is used to do a basic test. The code for this task is seen in listing 6. A random byte of data, and a random register are chosen. The data is then written to the register. The input data is changed to check that the data on the output is stored in memory and assertions are used to verify. The data is checked to persist after the write enable signal is inactive. If an assertion fails, a global error counter is incremented.

This task is then done a large amount of times to check all registers. Figure 6 shows the output waveform of the simulation. The error count is zero at the end of the simulation showing that the module is functioning correctly. The test is done with a complete set of registers. This is so the behaviour can be verified as address redundancy cannot be predicted.

Listing 6: Register task for writing and reading to the register.

```
task WriteandRead;
  Data =$random();
  temp = Data;
  Rs1 = $random();
  WE = 1;
  @(posedge Clock) //write clock edge
  #20 WE = 0;
  Data = $random(); //change the input
  @(posedge Clock) //need to wait a cycle to get the data back
  #20 assert(Rd1 == temp) else begin
    errors++;
```

```
12      $display("Write/Read Error");
13    end
14    @(posedge Clock) //check that the data persists with new
       input, WE = 0;
15    #20 assert(Rd1 == temp) else begin
16      errors++;
17      $display("Persist Error");
18    end
19  endtask
```

5.2 Synthesis

The synthesised logic of the register module is shown in 7. This is virtually identical to the Program ROM module, seen in figure 5 apart from the write address and data are inputs, rather than hardwired constants. This utilises the internal SRAM of the FPGA.

6 ALU

6.1 Design

The Arithmetic Logic Unit (ALU) is a key part of a processor. It is a combinational block, responsible for the arithmetic and logic operations on the data. The ALU in this processor has three operations on two operands. No flags, such as carry or zero, are implemented either as no conditional branches are needed.

The ALU supports the operations in table 6. The operation encoding is discussed in section 3. Due to the use of fixed point notation, the integer result of the multiplication is located at bits [14:7] of the 16 bit result. To correctly synthesise combinational logic, all inputs must have a defined output. The ALU was smallest in size when the 'A' operation was repeated in the redundant state.

The Cyclone IV FPGA has a number of embedded multipliers. One of these is utilised within the ALU to conduct signed multiplication. This is done by defining the inputs as signed, and using a combinational assign statement. The Quartus tool then recognises this and synthesises the design using the embedded multipliers.

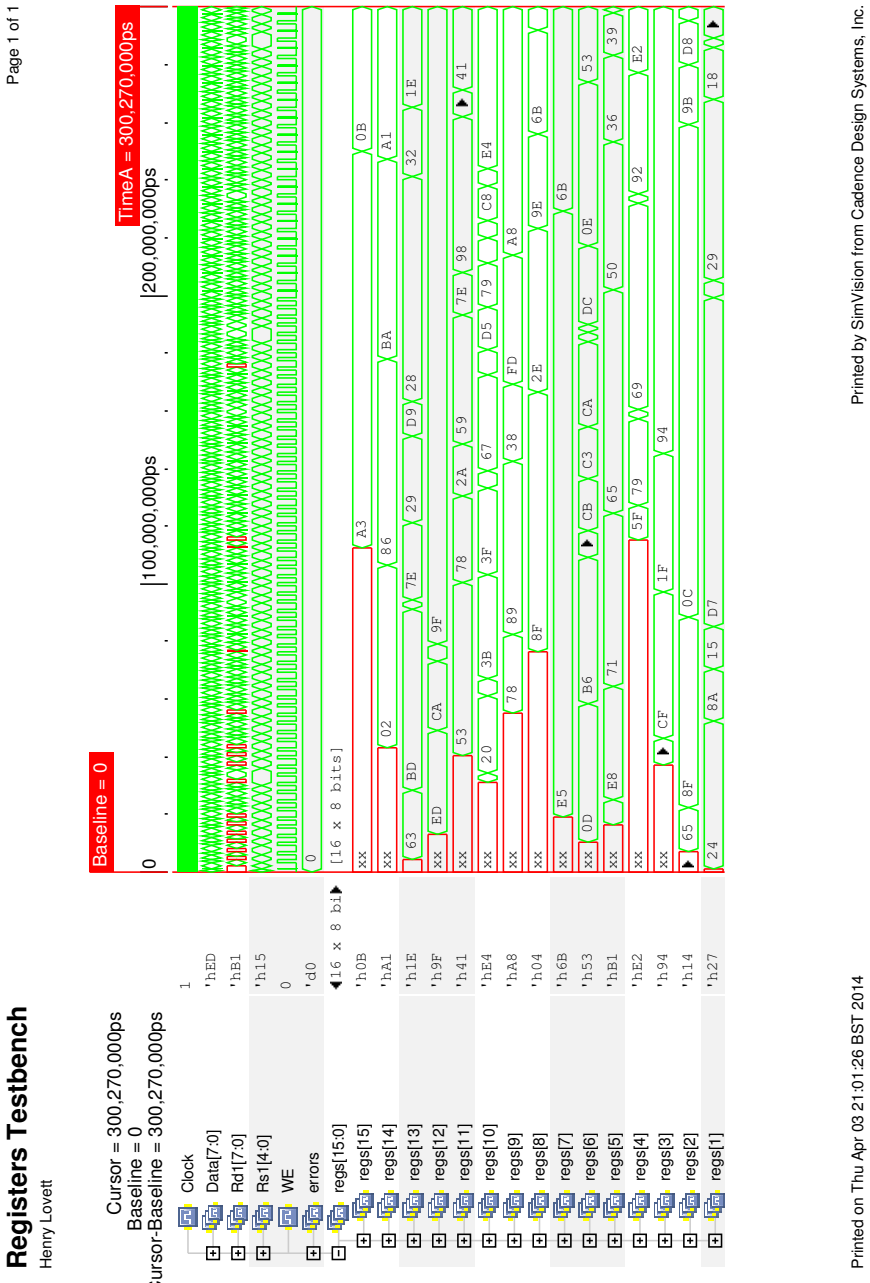

Figure 6: Waveform simulation for register testbench

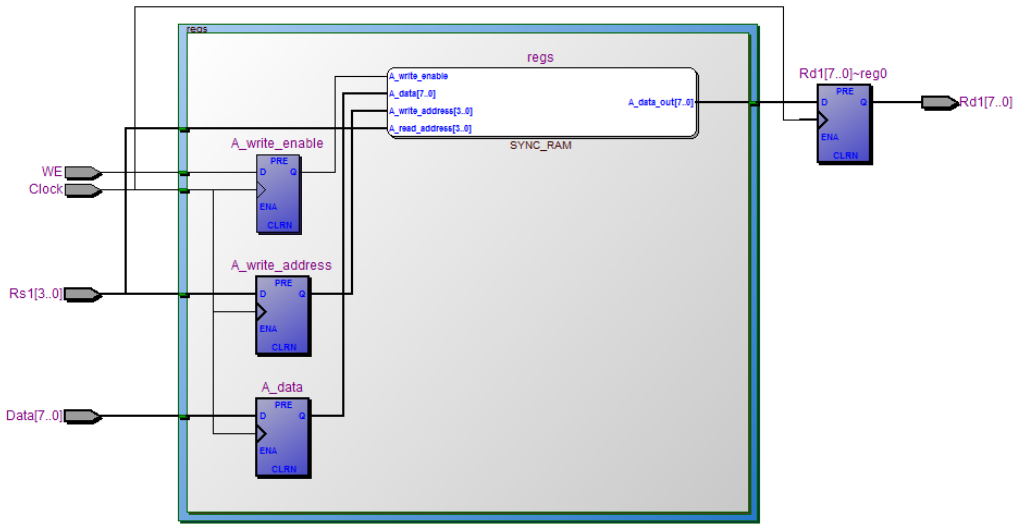

Figure 7: Synthesis of the register module

Table 6: ALU Operations supported

Operation	Explanation	Encoding (binary)
A	Result = Operand1	00 (10)
Add	Result = Operand1 + Operand2	01
Multiply	Result = (Operand1 × Operand2)[14:7]	11

6.2 Testbench

The testbench tests each operation with ten different values of the operands. An assertion then checks each result is as expected. An error count is kept and the simulation is successful if this is zero at the end of the simulation. Figure 8 shows the waveform for the simulation. It can be seen that all three operations are tested and the error count is zero at the end. The fourth, undefined operation is not tested as the result does not matter.

6.3 Synthesis

Synthesis of the ALU largely consists of multiplexers. As three operations are implemented, this requires 8×3-to-1 multiplexers. There is also an adder and a multiplier. The multiplier implemented utilised the embedded multipliers available on the device.

7 Datapath

The datapath is an encapsulating module to connect the registers and ALU. It also implements the Program Counter and multiplexors needed.

7.1 Testbench

The testbench verifies the following:

1. Program counter increment and write enable
2. Immediate loading into accumulator
3. Addition with an immediate
4. Write back to register
5. Read from register
6. Store switches to registers
7. Read switches from register
8. Program counter jump to accumulator value

Figure 8: Simulation waveform for the ALU testbench

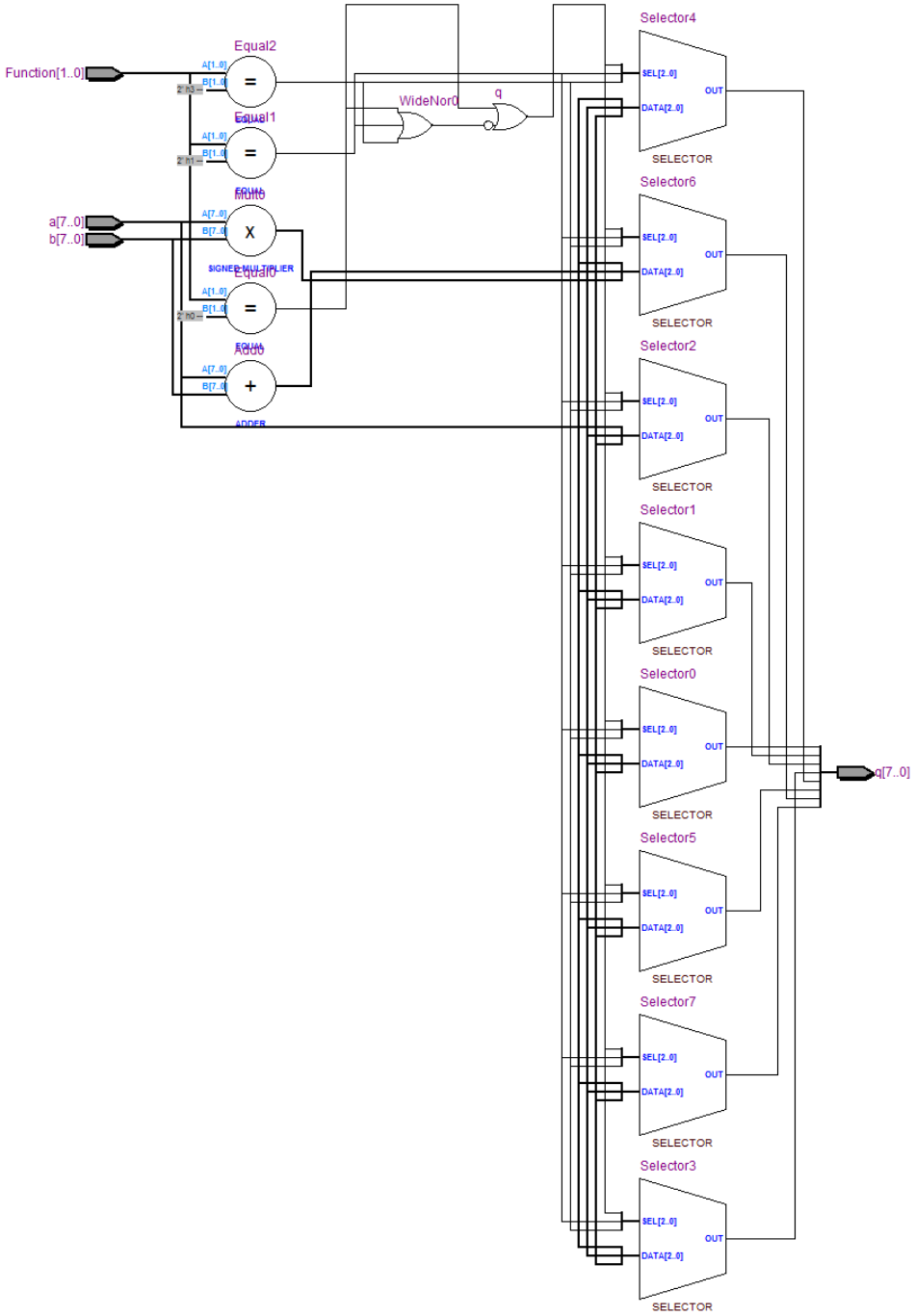

Figure 9: Synthesis of the Arithmetic Logic Unit

These operations are verified by viewing the LED output (the accumulator) and the program counter. Assertions are used to check the outputs and keep an error count. This testbench is to check the internal connections of the datapath, and also the implementation of the extra registers. Figure 10 shows the waveform of the simulation. The error count remains zero through out showing no errors exist in the datapath.

7.2 Synthesis

The synthesis of the datapath creates an instance of the register and ALU. It also implements the accumulator and program counter. A large number of multiplexers are used to direct the data flow.These take up a significant amount of the datapath. The whole synthesis is shown in figure 11.

8 CPU

8.1 Testbench

The testbench has two main parts; a task to commence the handshaking protocols, and a function to calculate the expected result. The handshaking task also contains assertions to verify the output. The function contains the constants and calculates the transform, with correct rounding. The task can be seen in listing 7 and the function is shown in listing 8. These are used in a loop to test random sets of inputs. A few basic tests are also done with simple data.

Listing 7: Verification task to conduct the handshaking protocols. Assertions are used in this task to verify the output.

```
//reset
initial
begin
        nReset = 1;
        #100 nReset = 0;
        #1000 nReset = 1;
end
//begin listing
task CheckTransform;// (logic[7:0] x, y);
  input logic [7:0] x1,y1,x2,y2;
  $display("x1 = %d\ny1 = %d", x1,y1);
  SW[7:0] = x1;
```

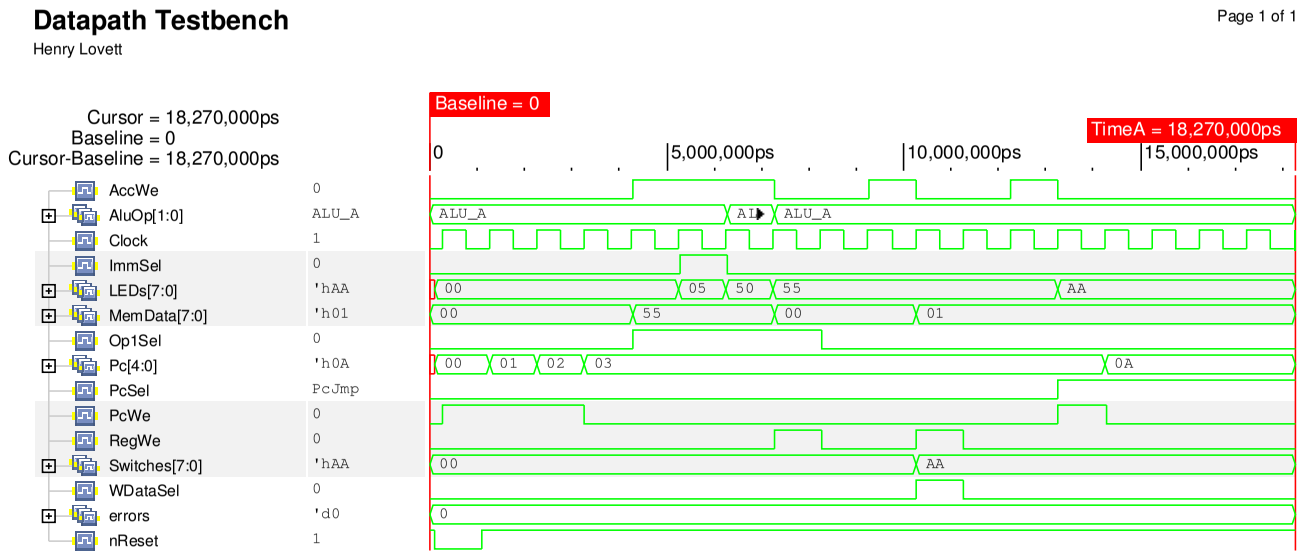

Figure 10: Datapath simulation waveform

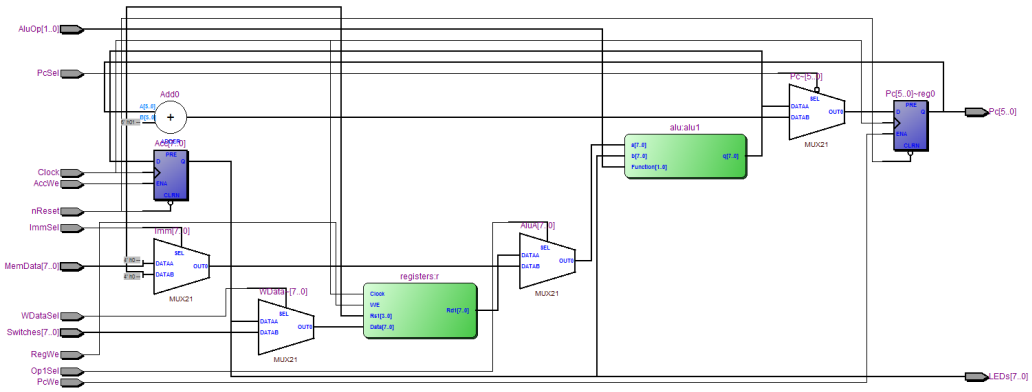

Figure 11: Synthesis of the Datapath

```
13    #60000 SW[8] = 1; //load x1
14    #10000 SW[8:0] = y1;
15    #10000 SW[8] = 1; //load y1
16    #10000 SW[8] = 0; //Go!
17    #60000   assert(LED[7:0] == x2)
18      $display("x2 = %d", x2);
19    else begin
20      $display("X Fail.\nx2 = %d. (Should be %d)", LED[7:0], x2);
21      errors++;
22    end
23
24   SW[8] = 1; //show y
25    #5000 assert(LED[7:0] == y2)
26      $display("y2 = %d", y2);
27    else begin
28      $display("Y Fail.\ny2 = %d. (Should be %d)", LED[7:0], y2);
29      errors++;
30    end
31    #3000 SW = 0;
32    #10000 ;
33  endtask
```

Listing 8: Function used to calculate the expected output of the affine transform.

```
1  function logic [15:0] CalculateTransform(logic signed [7:0] x1,
       y1);
2    automatic logic signed [7:0] x2c, y2c, a11, a12,a21,a22,b1,b2;
3    automatic logic signed [15:0] temp1,temp2,temp3,temp4;
```

```
4    a11 = 8'h40;
5    a12 = 8'h90;
6    a21 = 8'h90;
7    a22 = 8'h60;
8    b1  = 8'h05;
9    b2  = 8'h0C;
10   temp1 = a11 * x1;
11   temp2 = a12 * y1;
12   x2c = temp1[14:7] + temp2[14:7] + b1;
13   temp3 = a21 * x1;
14   temp4 = a22 * y1;
15   y2c = temp3[14:7] + temp4[14:7] + b2;
16   return {x2c, y2c};
17 endfunction
```

8.2 Simulation Results

Figure 12 shows the output waveform of the test. Registers 0-5 contain the constants used. Registers 6 and 7 are used to store the initial x_1 and y_1 vector. Registers 9 and 10 are used to store the result and register 8 is used as a temporary register. The global error counter is shown, indicating that the whole system passes the test. Each iteration consists of a function call to calculate the expected outcome, and a task call to input the data. This process is then looped to test different data sets.

8.3 Synthesis

The top level module creates a instance of the controller, datapath and ROM. The interconnects are made here and the inputs and outputs to the controller are also made. This top level synthesis is shown in figure 13.

9 DE0 Implementation

A counter is used to slow down the $50MHz$ clock supplied.This slowed the processor down to a human usable speed. The added logic from this is not included in the cost function.

To aid the demonstration, a small amount of extra logic is used to show if the opcode is a `WAIT`n instruction. This can be turned on or off by used of a *'define* statement declared in the *options.sv* file. It provides an insight in

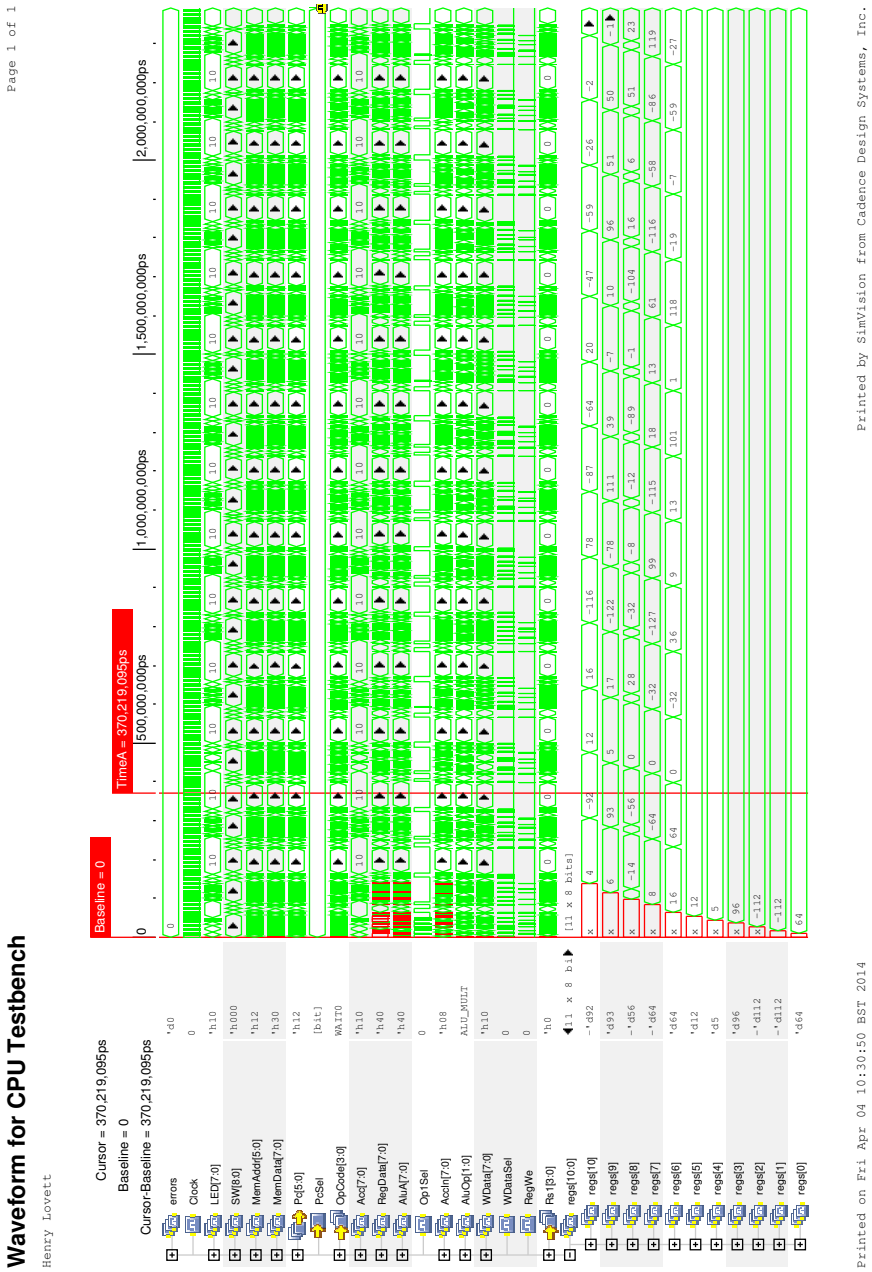

Figure 12: CPU simulation waveform

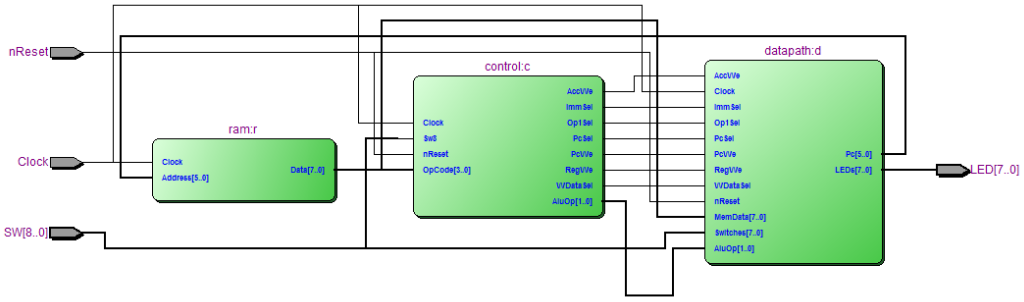

Figure 13: Top level synthesis of the processor.

to the current state of the processor and if it is waiting for an input. This is accomplished by the code seen in listing 9. The logic added by here is not included in the overall cost function and is added to aid the use.

Listing 9: Extra LED logic to show when the processor is executing a `WAIT` instruction.

```
`ifdef demo
  always_comb
  begin
    case (opcodes_t'(MemData[7:4]))
      WAIT0: LED[9:8] = 2'b01;
      WAIT1: LED[9:8] = 2'b10;
       default: LED[9:8] = 2'b00;
    endcase
  end
`endif
```

9.1 Issues encountered

The ROM was found to be difficult to minimise in size. A threshold was found at 512 bits where, if any lower, the module was synthesised into logic elements. This could be due to the cost figure for the Altera Synthesis tool to minimise is very different to the cost figure minimised here. The use of logic elements would keep the same functionality, but utilises the look up tables in a logic cell. The overall cost was still lower using the SRAM, even though a large proportion of the memory is inaccessible and remains unused.

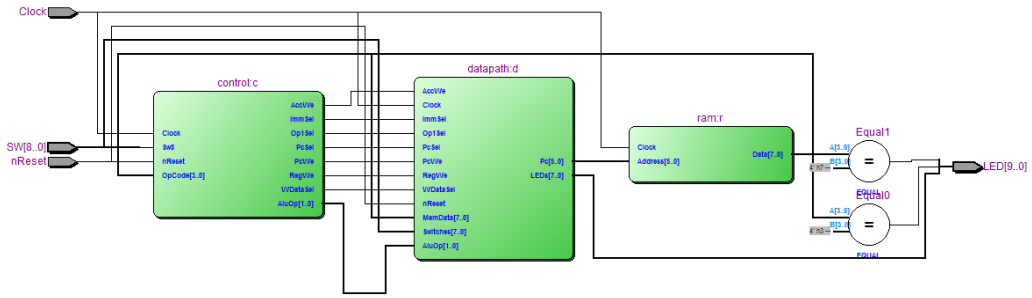

Figure 14: Synthesised top level module showing the logic added for the demonstration

9.2 Synthesis

The extra logic created by the demo definition is shown in figure 14. It consists of two equality checks between the opcode and a constant (the `WAIT`n opcodes).

10 Conclusion

The final processor satisfies the specification and the objectives highlighted in section 1. The processor is a register-accumulator based, MIPS-esque architecture. It implements ten instructions in total to calculate the affine transform of 8 bit data.

The total cost of the processor is 72. A full breakdown of the costs is seen in table 7. A large amount of the cost is located in the datapath. This is due to the multiplexers needed to direct the data around. A more compact design could be achieved by aiming to minimise these elements. The two registers in the datapath also contribute a fair amount to the cost. These are a compromise and by using a register-register architecture, the decoding logic is increased and a larger instruction is needed. Overall, the design is a success, passing thorough testing and achieves the initial goals.

Table 7: Break down of the costs by module. Costs do not include any instances made by the module. Clock reduction and debugging logic is not included in this cost.

Module	Logic Elements	Memory (bits)	Embedded Multipliers
Control	9	0	0
Datapath	24	0	0
Registers	0	88	0
ALU	21	0	1
ROM	0	512	0
Total	54	600	1