ELEC6234 – FPGA Synthesis of a picoMIPS

Andrew Everitt
Department of Mechanical Engineering
University of Southampton
aje2g15@soton.ac.uk

Supervisor: Dr. Tomasz Kazmierski
Department of Electronics and Computer Science
University of Southampton
tjk@soton.ac.uk

***Abstract*— A No Instruction Set Processor, inspired off the picoMIPS architecture, has been designed, tested and optimised to use the fewest hardware components. The processor is capable of performing an affine transform from user inputted coordinates. The design uses 10 Arithmetic Logic Modules, 6 Digital Signal Processing units, and 16 memory bits on the Cyclone V FPGA.**

I. INTRODUCTION

The objective of this project was to create a processor, for a Cyclone V device, capable of performing an affine transform for a single input coordinate. The size of the processor should be minimised according to the function:

$$Cost = ALMs + \max(0, DSP - 2) + 30 * KbitsRAM$$

The picoMIPS processor was used as inspiration for the top-level design; however, there was lacking functionality, and an excess of unnecessary hardware. A new design was derived by first determining what operations would be required to perform the affine transform and output the results to match the program pseudocode. The Arithmetic Logic Unit (ALU) was entirely redesigned to act as an accumulator with signed addition and multiplication operations. Each module was redesigned, tested and optimised in turn to produce the final design.

A No Instruction Set Computing (NISC) processor, meeting all the requirements, has been designed and tested in Modelsim. The design has been optimised to use Digital Signal Processing (DSP) units over Arithmetic Logic Modules (ALMs) to minimise the cost function, **sections II.A** & **II.C**. The processor requires a minimum of 29 clock cycles to retrieve the necessary data and compute the affine transform, **section II.B**. Timing analysis was conducted, and a maximum allowable clock speed of 54.5 MHz was determined, **section III**.

II. DESIGN

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

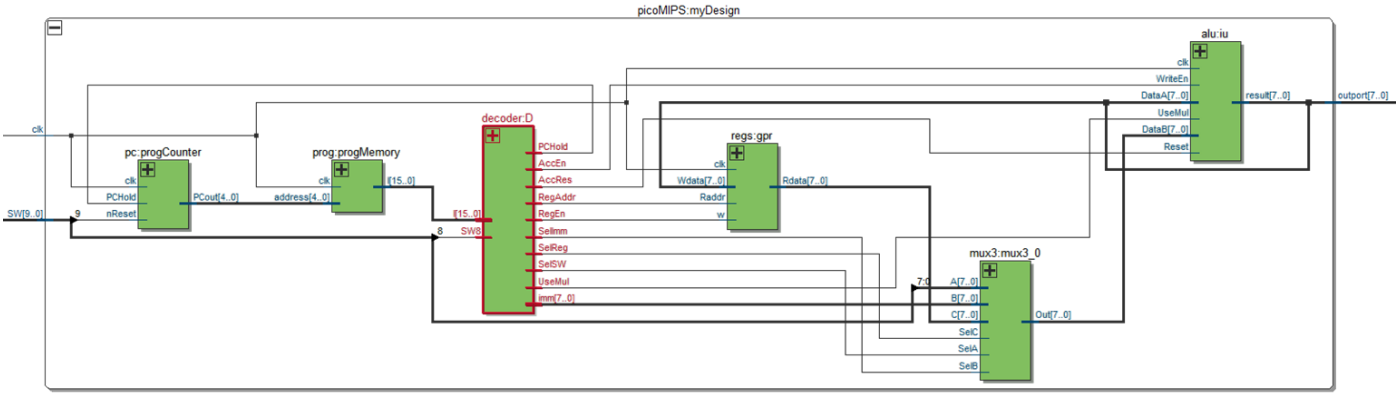

Figure 1: RTL block diagram illustrating the top-level component connections.

This processor utilises a NISC architecture. NISC provides two main advantages over a similar Reduced Instruction Set Computing (RISC) architecture; these are a simpler and more efficient hardware scheduler with better resource utilisation [1], [2]. A standard instruction decoder is not required, instead the processor uses 8 control bits with a n-bit immediate as the instruction, this results in an (8+n) bit instruction. The 8 control bits required by this processor to perform an affine transform are described in

TABLE I.

TABLE I: Instruction control bit descriptions.

Control Bit	Description
7	ALU accumulator reset
6	Hold program counter
5	Select switches as ALU input
4	Select immediate as ALU input
3	Select register as ALU input
2	Use multiplier
1	Enable ALU accumulator write
0	Enable register write

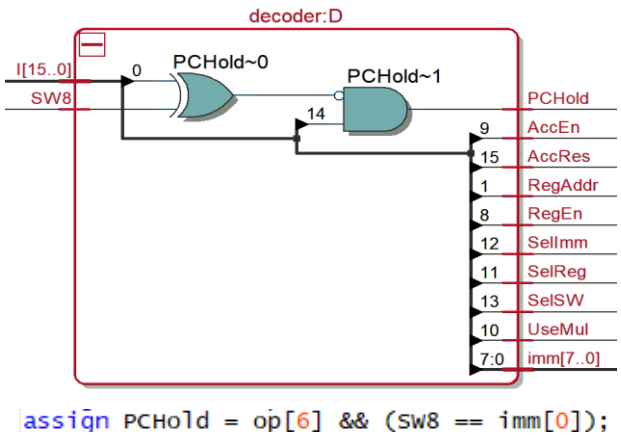

Figure 2: Instruction decoder module design.

The decoder module accepts an (8+n) bit instruction input, the 8 most significant bits (MSB) are assigned to their own output signal for easier use with other modules. The *n* least significant bits (LSB) are assigned to the immediate (*imm*) bus. Only control bit 6, *PCHold*, requires combinational logic. This is to compare the value of switch 8 to the *imm[0]* to either enable or disable the program counter, **Figure 2**. The decoder module is asynchronous, so the control bits are updated in the same clock cycle as the input instruction and *PCHold* can be disabled at any point.

The affine transform program control path is controlled using switch 8 to progress the program when new data is available to be input into the processor using switches [7:0]. No branching is required for the program counter, therefore only +1 and +0 increments have been implemented. If *PCHold* is high, then the program counter will not increase and the instruction will repeat until switch 8 matches *imm[0]* to push *PCHold* low. This increment control has been achieved using an embedded multiplier in a DSP on the Cyclone V FPGA to reduce the total cost function of the processor. The cost advantage of using an embedded multiplier to add is described in **section II.C**.

The program counter is synchronous with the active high clock edge. This ensures that only one set of instructions can occur per clock cycle.

The program is initially stored in a hex file, allowing for easy modification to the program. For an 8-bit processor, each instruction is 16 bits. The hex file is initialised in an asynchronous ROM block allowing the instruction to be retrieved in the same clock cycle as the program counter increments. Using asynchronous ROM means 15 ALMs are required to store the program; if the ROM is made synchronous then it would require an additional clock cycle to retrieve the instruction after the program counter increments, this prevents the implementation of the *PCHold* control signal from working however it also limits the maximum clock speed of the processor since there is a propagation delay from the input of the address signal to the output of the instruction. Using synchronous program memory only provided a very minimal cost advantage compared to asynchronous memory since it requires 480 memory bits.

By combining the instruction control bits, multiple operations can be carried out in a single clock cycle. An example of this is when waiting for switch 8 to be pushed low before displaying the coordinate *x2*, the ALU accumulator is reset and the appropriate register is loaded; three operations have been carried out using a single instruction.

B. General Purpose Register file design, simulation and synthesis

Two general purpose, synchronous registers are required to complete an affine transform. An additional register is incorporated into the ALU, the output of this register is always connected as first input to the ALU therefore acting as an accumulator, this is described further in **section II.C**.

TABLE II: Affine transform program showing the values stored in the ALU accumulator and the two general purpose registers at each point in the program.

Program Count	Accum	reg0	reg1
0	0	0	0
1	x_1	``	``
2	$A_{11} * x_1$	``	``
3	$A_{11} * x_1 + b_1$	``	``
4	$A_{11} * x_1 + b_1$	$A_{11} * x_1 + b_1$	``
5	0	``	``
6	x_1	``	``
7	$- A_{12} * x_1$	``	``
8	$- A_{12} * x_1 + b_2$	``	``
9	$- A_{12} * x_1 + b_2$	``	$- A_{12} * x_1 + b_2$
10-11	0	``	``
12	y_1	``	``
13	$A_{12} * y_1$	``	``
14-15	$A_{12} * y_1 + reg0$	``	``
16	$A_{12} * y_1 + A_{11} * x_1 + b_1$	$A_{12} * y_1 + A_{11} * x_1 + b_1$	
17	0	``	``
18	y_1	``	``
19	$A_{22} * y_1$	``	``
20-21	$A_{22} * y_1 + reg1$	``	``
22	$A_{22} * y_1 - A_{12} * x_1 + b_2$	``	$A_{22} * y_1 - A_{12} * x_1 + b_2$
23	0	``	``
24	reg0	``	``
25	0	``	``
26-31	reg1	``	``

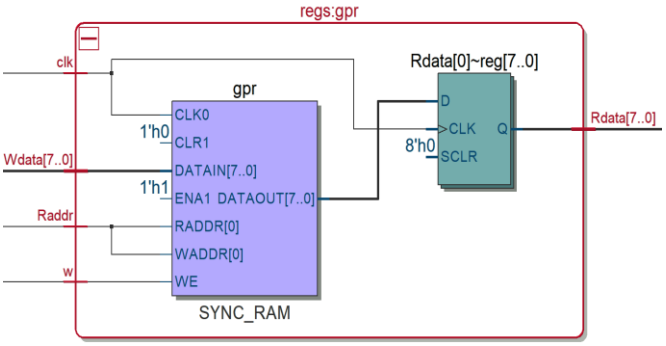

Figure 3: Register module.

```
module regs #(parameter n = 8)  // n - data bus width
    (input logic clk, w,        // clk and write control
    input logic [n-1:0] Wdata,  // write data
    input logic Raddr,          // read address
    output logic [n-1:0] Rdata  // read data
    );

    // Declare 2 n-bit registers
    logic [n-1:0] gpr [1:0];

    // initialise registers
    initial
    for (int i = 0; i < 2; ++i)
        gpr[i] = 0;

    always @ (posedge clk)
    begin
        // write process
        if (w)
            gpr[Raddr] <= Wdata;
        // read process
        Rdata <= gpr[Raddr];
    end

endmodule
```

The registers were made synchronous so that they utilised the memory bits available on the Cyclone V device. Two general purpose synchronous registers occupy 16 memory bits, whereas two asynchronous registers would require 13 ALMs, it is therefore lower cost to implement synchronous registers. A disadvantage is these registers require two clock cycles to load the correct value into the accumulator, as shown for program count 14-15 and 20-21 in **TABLE II**. If the previous instruction to when the register value is required does not modify the accumulator, other than resetting it, the appropriate register can be loaded into the second input to the ALU therefore saving one clock cycle. This method is used in program count 23 and 25, where the program counter is waiting for switch 8 to be in the correct position, the appropriate register is being loaded. The register to read from is determined from *imm[1]*, since there are only two registers. Using *imm[1]* allows loading to be done in parallel to *PCHold* which uses *imm[0]*.

A testbench was used to test the register module. **Figure 4** shows that input data *Wdata* changes at 100 ps, however it is written to the register on following positive clock edge at 105 ps, the data is then available to be read at the next positive clock edge at 115 ps. After the first positive clock edge, the output is being read from the correct register however the output data has not yet been updated due to having two flip flops in series, **Figure 3**.

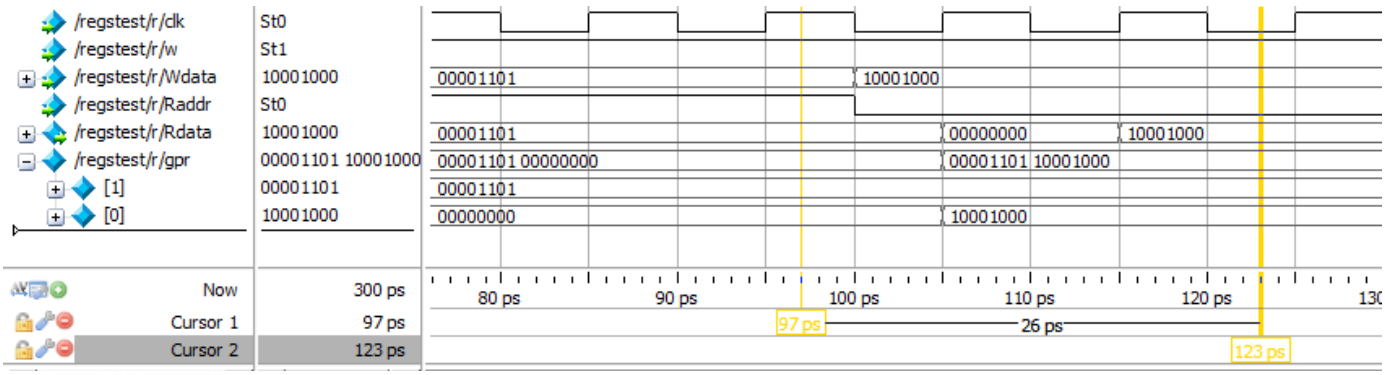

Figure 4: Register testbench simulation.

The *Rdata* output from the register module will update every clock cycle with the data stored in the register from the previous clock cycle regardless if writing is enabled. The data stored in the registers is only updated if writing is enabled.

C. *Arithmetic Logic Unit and Multiplier design*

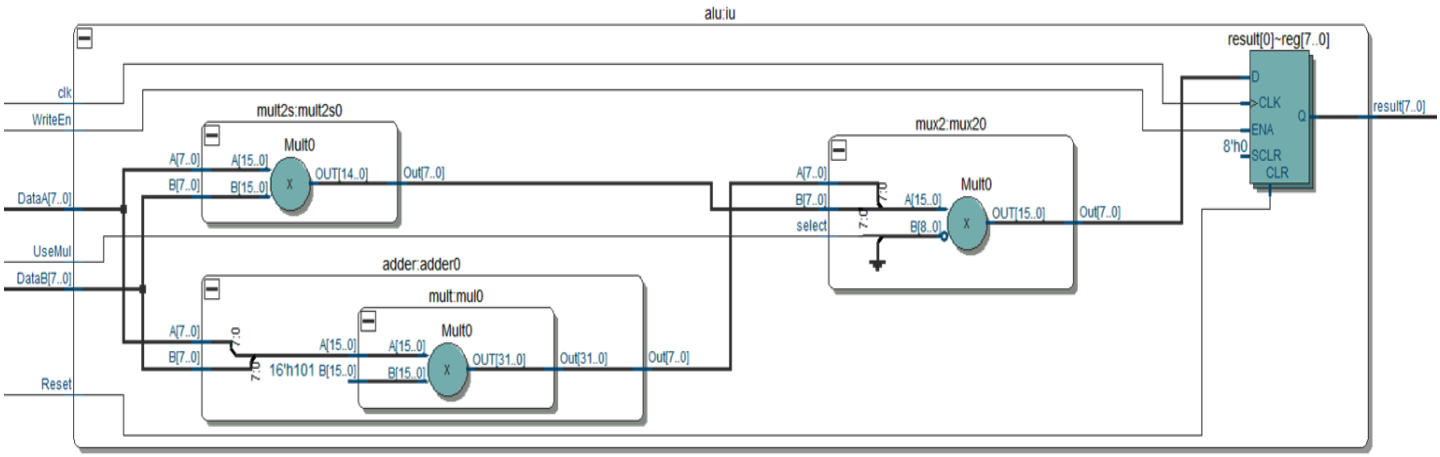

Figure 5: ALU top-level design.

To compute an affine transform, signed addition and signed multiplication are required. By taking 2's complement inputs for both operations, subtraction and negative multiplication can also accomplished. It is possible to add binary number through multiplication, a comparison between the effective costs of an adder and a multiplicative adder using the cost function in **section I** is required to determine the optimal design of the ALU.

To add by multiplication, the two n-bit binary numbers, *A* & *B*, are concatenated into a single 2n-bit number, *{A, B}*. This is multiplied by *{{(n-1){1'b0}}, 1'b1, {(n-1){1'b0}}, 1'b1}*, the output of the multiplication is 4n bits with the format *{{n{1'b0}}, A, A+B, B}*. The addition output is easily segmented from this result. When synthesised, this method of addition uses a single DSP. Alternatively, if an adder is used then 5 ALMs are required. Using the cost function provided, it is therefore significantly more efficient to use the multiplicative approach to addition; however, there are significantly fewer DSPs (87) available in the Cyclone V FPGA compared to ALMs (32,070) so this approach would not scale to a larger project.

Traditional ALU flags have been removed from this design since they are not required to accomplish an affine transform. If the operand stored in the ALU accumulator overflows during an operation, then it is assumed to wrap around as if the edges of the image wrap top-bottom & left-right, hence the *overflow* flag has been removed. The *carry* flag is not required since the ALU output is directly fed back into the first ALU input and as mention prior, overflow is acceptable. The *negative* result flag could be implemented without any additional logic since 2's complement number are used; however, it is not required to perform an affine transform and has therefore been removed. The final *zero* flag could be used when comparing the value of switch 8 to an *imm* while holding the program counter, implementing this functionality into the ALU would mean the accumulator would need to be cleared, the *imm* loaded into the accumulator and then the current switch value compared to it. To reduce the total program instructions required to assess the state of switch 8, the required functionality of the *zero* flag has instead been moved into the decoder where it is computed asynchronously for every new instruction. The total logic used is equal, requiring 1 ALM, however the number of program instructions is

```
// Out = A * B
module mult #(parameter n = 8) (
    input signed [n-1:0] A,
    input signed [n-1:0] B,
    output signed [2*n-1:0] Out
    );

    assign Out = A * B;

endmodule

// Out = A + B
module adder #(parameter n = 8) (
    input signed [n-1:0] A,
    input signed [n-1:0] B,
    output logic [n-1:0] Out
    );

    logic [2*n-1:0] addOut;

    //-------------------------------------------------
    // Uses adder
    assign addOut = ({A, B}) * ({7'b0, 1'b1, 7'b0, 1'b1});
    //-------------------------------------------------

    //-------------------------------------------------
    // Uses multiplier
    mult #(2*n) mul0(
        .A      ({A, B}),
        .B      ({7'b0, 1'b1, 7'b0, 1'b1}),
        .Out    (addOut)
    );
    //-------------------------------------------------

    assign Out = addOut[2*n-1:n];
endmodule
```

Figure 6: ALU adder module showing two different implementations using multiplication that synthesis different hardware. (To synthesis only one implementation should be used).

reduced from 4 to 1. As mentioned in **section II.A** the instruction used to hold the program counter can be combined with other operations to prepare the accumulator for the next instruction, reset & load.

```
// 2's complement multiplier
module mult2s #(parameter n = 8) (
    input signed [n-1:0] A,
    input signed [n-1:0] B,
    output signed [n-1:0] Out
    );

    logic [2*n-1:0] TmpA, TmpB;
    logic [4*n-1:0] mulOut;

    assign TmpA = {{(n){A[n-1]}}, A};
    assign TmpB = {{(n){B[n-1]}}, B};

    assign mulOut = TmpA * TmpB;

    assign Out = mulOut[2*n-2:n-1];
endmodule
```

Figure 7: ALU 2's complement multiplier.

For an 8-bit processor, signed addition utilises a 16x16 bit embedded multiplier as described earlier. This allows for both addition and subtraction using 2's complement inputs. Quartus prime 16.0 will not synthesis an embedded multiplier if an *assign* statement is used directly with the multiplication when adding numbers. However, if a submodule is used to multiply the numbers then a DSP embedded multiplier is inferred, both methods are shown in **Figure 6**. This is independent of the *AUTO_DSP_RECOGNITION* global assignment, which must be set to *ON* to use DSP embedded multipliers in this design.

Signed multiplication uses a second 16x16 bit embedded multiplier. To multiply 2's complement numbers the binary inputs must first be sign extended to twice their original length [3]. Only the 16 LSBs of the 32-bit multiplication result are important. From these 16 bits, the radix point is between bit 6 and 7, meaning only bits 14 to 7 give the integer result, **Figure 7**. Since both inputs are variable, Quartus will synthesis an embedded multiplier if *AUTO_DSP_RECOGNITION* is enabled. If this global assignment is disabled then 102 ALMs will be used to achieve the same output.

A structural ALU format has been used, the addition and multiplication results are therefore both computed in parallel asynchronously. To select which output goes to the accumulator, a multiplexer is used with a single control bit from the program instruction *UseMul*. To reduce the total logic used, a third embedded multiplier is used to simulate a 2-input multiplexer using the same logic as the adder however input B has the value *({7'b0, select, 7'b0, ~select})*.

To perform an affine transform, given the desired switch input sequence where each part of the initial coordinate is given sequentially. The arithmetic done on each input can be done sequentially without needing to store and load various a large amount of values into registers. It is therefore more efficient to temporarily store the output of the ALU in a dedicated register. Since multiple operations are done to the ALU output, it also requires fewer program instructions to connect the ALU output as the first input. Since the ALU operations are asynchronous, the synchronous register must have a write enable. It would be possible to use this accumulator without a write enable signal since the default operation if no control signals are high is *accum += 0*, however having a dedicated control signal adds additional security that the register will not change when it is not desired.

```
// 3 input multiplexer
// SelA | SelB | SelC | Out
//  1   |  0   |  0   |  A
//  0   |  1   |  0   |  B
//  X   |  X   |  1   |  C
//  1   |  1   |  0   |  A+B
module mux3 #(parameter n = 8) (
    input signed [n-1:0] A,
    input signed [n-1:0] B,
    input signed [n-1:0] C,
    input SelA, SelB, SelC,
    output logic [n-1:0] Out
    );

    logic [2*n-1:0] mul0Out, mul1Out;

    assign mul0Out = ({A              , B}) * ({7'b0, SelB, 7'b0,  SelA});
    assign mul1Out = ({mul0Out[2*n-1:n], C}) * ({7'b0, SelC, 7'b0, ~SelC});

    assign Out = mul1Out[2*n-1:n];
endmodule
```

Figure 8: 3-input multiplexer. SelA, SelB, SelC are the control signals that select which input will be propagated.

The second input to the ALU can be 1 of 3 signals: the current switch values, the immediate, or a register value. A 3-input multiplexer selects which signal to propagate using 3 control bits. Embedded multipliers have once again been used to optimise the cost function, **Figure 8**.

A Modelsim testbench was used to assess whether the ALU logic was correct, **Figure 9**. For this test the *result* was not connected to *DataA*. For addition, both *DataA* and *DataB* are 2's complement integers, while for multiplication *DataB* is considered to be a 2's complement floating point number with a radix point after the MSB. For each set of inputs, the correct temporary outputs are produced, *TmpAdd* and *TmpMul*. The correct temporary output, *TmpOut*, is then

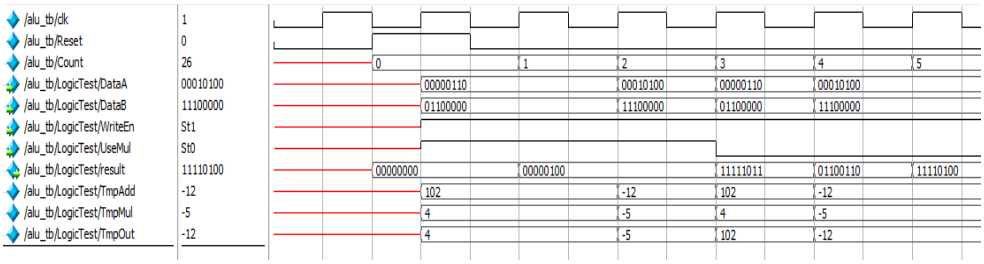

Figure 9: ALU adder and multiplier modelsim logic test.

selected using the multiplexer with *UseMul*, *TmpOut* is stored in the output register, *result*, on the next positive clock edge.

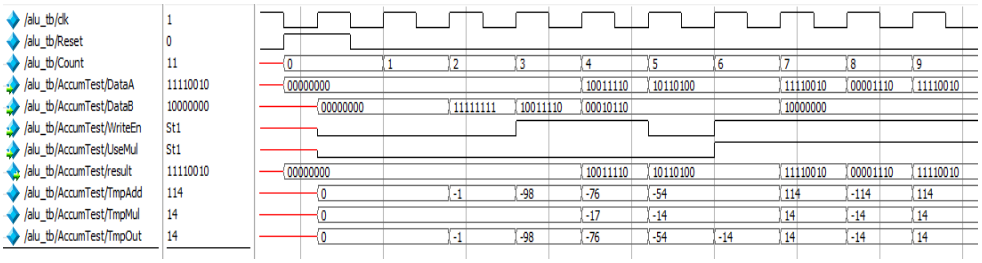

Figure 11: ALU adder and multiplier modelsim accumulator test.

A second testbench was used to test the accumulator functionality, **Figure 11** & **Figure 10**. This further successfully tested the logic results, it can also be seen that *DataA* updates simultaneously with the output register *result*, it is then correctly used in the next operation.

```
// testing ALU with feedback
always_ff @ (posedge clk) begin
    case(Count)
        0 : begin // result should not change
            WriteEn1 = 0;
            UseMul1 = 0;
            DataB1 = 8'b00000000;
            end
        1 : begin // result should not change
            WriteEn1 = 0;
            UseMul1 = 0;
            DataB1 = 8'b11111111;   // -1
            end
        2 : begin // result += DataB1
            WriteEn1 = 1;
            UseMul1 = 0;
            DataB1 = 8'b10011110;   // -98
            end
        3 : begin // result += DataB1
            WriteEn1 = 1;
            UseMul1 = 0;
            DataB1 = 8'b00010110;   // 22
            end
        4 : begin // result = result
            WriteEn1 = 0;
            UseMul1 = 0;
            DataB1 = 8'b00010110;   // 22
            end
        5 : begin // result *= DataB1
            WriteEn1 = 1;
            UseMul1 = 1;
            DataB1 = 8'b00010110;   // 0.171875
            end
        6 : begin // result *= DataB1
            WriteEn1 = 1;
            UseMul1 = 1;
            DataB1 = 8'b10000000;   // -1
            end
    endcase
end

// ALU with closed loop feedback to test accumulator
alu AccumTest (
    .clk(clk),
    .Reset(Reset),
    .DataA(result1),
    .DataB(DataB1),
    .WriteEn(WriteEn1),
    .UseMul(UseMul1),
    .result(result1)
);
```

Figure 10: Extract from ALU accumulator testbench code.

III. FPGA IMPLEMENTATION

Before the processor design was implemented on the Cyclone V FPGA, each module was tested with a testbench in Modelsim to debug modules independently. Once each module was confirmed to work as intended, a top-level testbench was designed to simulate the entire processor with switch inputs. The switch changes were offset from the clock allowing successful debugging of synchronous and asynchronous updates in each module.

The RTL schematic, and flow summary available in Quartus were used to ensure the desired hardware was synthesised. Adjustments to the design were often needed to infer specific hardware, such as implementing a submodule in the ALU adder to infer an embedded multiplier, or making the register read synchronous to use the memory bits available on the Cyclone V. The RTL schematic was a useful tool to ensure connection were being properly made.

Initial testing of the multiplication operation in the ALU confirmed that it was not working for all combinations of positive and negative numbers, it was later found that the inputs needed to be sign extended to compute 2's complement multiplication. Once the processor was functional, significant effort went into reducing the logic usage and minimising the number of clock cycles required to accomplish the program objective. The initial design required 47 instructions

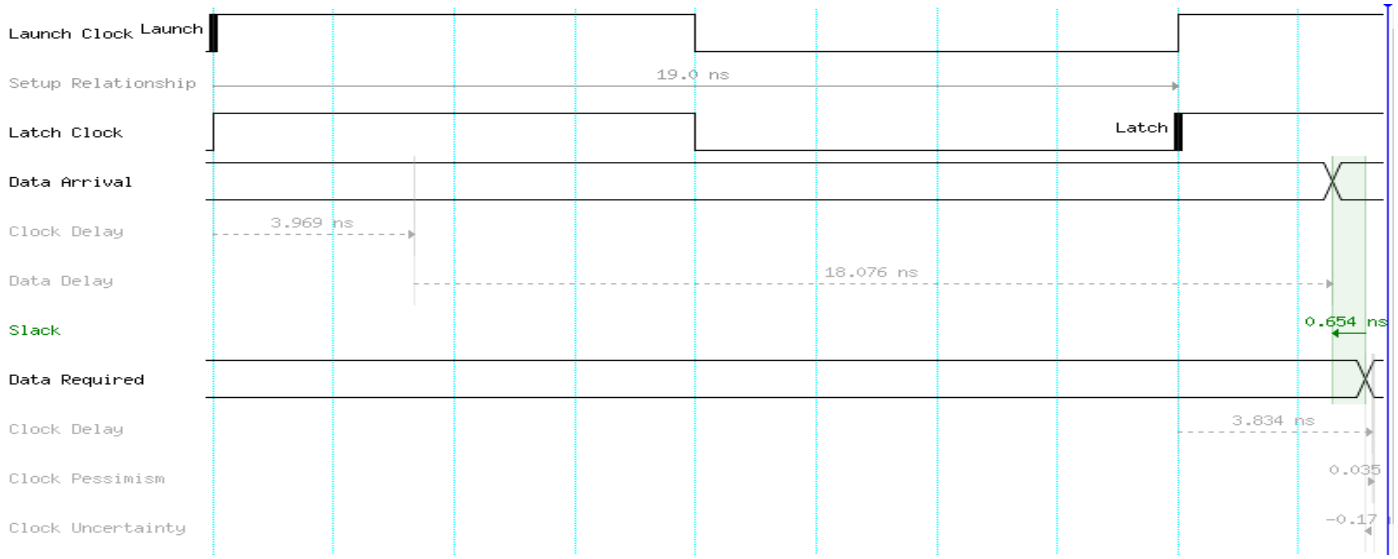

Figure 12: Timing analysis of the picoMIPS module. Maximum allowable clock frequency of 54.5 MHz.

meaning a 6-bit program counter, however the final design reduced the required instructions to 29. A 5-bit program counter could therefore be used.

Some marginal resource improvements could have been made by using synchronous RAM however it also caused issues with the current implementation of *PCHold* as mentioned in **section II.A**.

Timing analysis was conducted on the processor design to determine the maximum clock speed before the propagation delay of the logic units and DSPs caused data to be lost as flip flops update before the next flip flop has received the old data. A minimum clock period of 18.346 ns is required to have positive slack, shown in **Figure 12**, this relates to a maximum clock speed of 54.5 MHz [4]. Since the maximum clock speed available of the Cyclone V FPGA is 50 MHz, this processor design is within the hardware limitations of the FPGA.

Pin assignments to connect to the peripherals were found in the DE1-SoC User Manual [5].

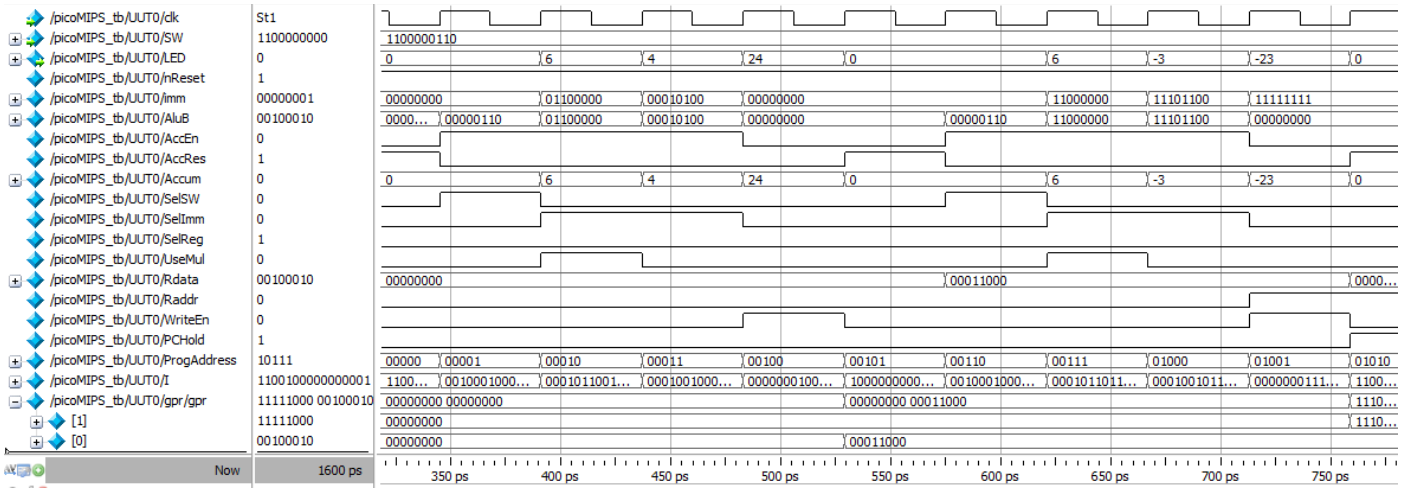

Figure 13: An extract of the affine transform program being simulated on the top-level processor testbench.

Flow Status	Successful - Sun Apr 14 17:15:10 2019
Quartus Prime Version	16.0.0 Build 211 04/27/2016 SJ Lite Edition
Revision Name	picoMIPS4test
Top-level Entity Name	picoMIPS
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	10 / 32,070 (< 1 %)
Total registers	5
Total pins	19 / 457 (4 %)
Total virtual pins	0
Total block memory bits	16 / 4,065,280 (< 1 %)
Total DSP Blocks	6 / 87 (7 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 14: Quartus flow summary of the processor design.

IV. Conclusion

All the design objectives have been met by this processor design. The required hardware of the processor is: 10 ALMs, 6 DSPs, and 16 memory bits. The total cost is therefore 14.48.

I think that this approach to designing a processor has limited room for further optimisation. It might be possible to further reduce the number of DSPs used as multiplexers by computing all possible results (addition and multiplication) for all inputs (switches, immediate, and registers) and storing them in additional registers. If all the inputs were concatenated in a specific format, then the multiplication and addition could be done with the current ALU design. An additional general-purpose register would be required. One further improvement could be to add button debouncing, since when using a full 50 MHz clock, some stages appear to be skipped due to bounce. This is not an issue when the clock speed is reduced.

I have learnt how to program in Systemverilog, design and synthesis a medium sized device and test its operation both in simulation and on an FPGA.

Compilation Hierarchy Node	LC Combinationals	LC Registers	Block Memory Bits	DSP Blocks	Pins
|picoMIPS	16 (0)	13 (0)	16	6	19
|alu:iu|	0 (0)	8 (8)	0	3	0
|decoder:D|	1 (1)	0 (0)	0	0	0
|mux3:mux3_0|	0 (0)	0 (0)	0	2	0
|pc:progCounter|	0 (0)	5 (5)	0	1	0
|prog:progMemory|	15 (15)	0 (0)	0	0	0
|regs:gpr|	0 (0)	0 (0)	16	0	0

Figure 15: Resource usage by entity.

REFERENCES

[1] R. Grubiši, 'The Use of the No-Instruction-Set Computer (NISC) for Acceleration in WISHBONE-Based Systems', p. 22.

[2] 'A No Instruction Set processor'. [Online]. Available: http://cowlark.com/nisc/index.html. [Accessed: 27-Mar-2019].

[3] 'Beyond 354 - Two's Complement Multiplication'. [Online]. Available: http://pages.cs.wisc.edu/~smoler/cs354/beyond354/int.mult.html. [Accessed: 01-Apr-2019].

[4] 'Using TimeQuest Timing Analyzer', *Altera Corp. - Univ. Program*, p. 16, 2014.

[5] *DE1-SoC User Manual*. Terasic Technologies Inc., 2013.