ELEC6234 – Embedded Processors

Coursework Report – picoMIPS implementation

Guga Kupradze
Gk1u22
European Masters In Embedded Computing Systems (EMECS)
Tomasz Kazmierski

1. Introduction

Through the mentioned coursework, I have successfully designed and implemented a custom picoMIPS processor architecture tailored to perform affine transformations. The preparation work involved understanding the requirements of the project, and modifying existing architecture accordingly.

The experimental work comprised of running simulations using ModelSim to verify that the modified components functioned correctly. The FPGA implementation was tested using the test cases in the picoMIPS module testbench, ensuring that the simulation results matched the real implementation. Also, the area has been optimized by adjusting the optimization settings to minimize the number of ALMs used to 45.

As a result, the custom picoMIPS processor architecture presented in this report is capable of efficiently performing affine transformations, as demonstrated through the simulations and FPGA. The specific details of the architecture modifications and extensions, as well as the experimental work results are provided in the following sections.

Design Details Form			
Total Cost: 45	**ALMs: 45**	**Memory bits: 0**	**Multipliers: 1**

Instruction Set

In total I have implemented 8 instructions:

1) IMM (11001) → Store immediate value in the register
2) SWI (11101) → Store value from switches (0-7) to the register and jump to the next instruction when SW8 becomes 1
3) ADD (01010) → Add value from the destination register to the value from the source and store it in the destination register
4) ADDI (01110) → Add an immediate value to the value stored in the destination regis and save the sum into the destination register
5) MUL (10011) → Multiply the value stored in the source register by the value stored i the destination register. Update the destination register
6) MULI (10111) → Multiply the value stored in the destination register by the immediat value and update the destination register
7) ST_0 (00000) → Wait for the SW8 to become 0
8) ST_1 (00100) → Wait for the SW8 to become 1

Instruction Format

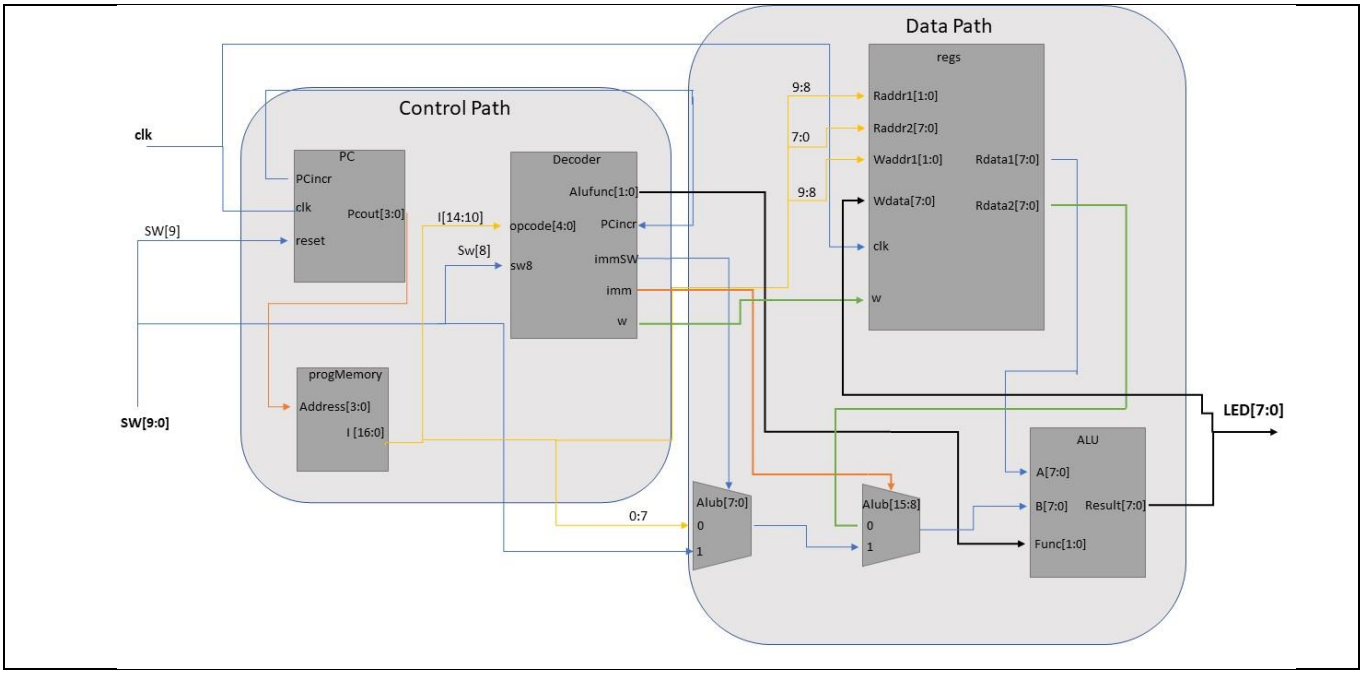
Implemented instruction format consists of 15 bits.

format: 5b opcode, 2b %d, 8b immediate or %Source --> 15 Bits Total

Your affine transformation program

```
// HEX //////////// BINARY  ////////////////////// ASSEMBLER /////////
//Store num1 to %0
7400    // 11101 00 00000000          SWI          %0      num1
//Wait for SW8 to become 0
0000    // 00000 00 00000000          ST_0   %0
//Store num2 to %1
7500    // 11101 01 00000000          SWI          %1      num2
//Wait for SW8 to become 0
0100    // 00000 01 00000000          ST_0   %1

//Store 0.75 to %2
6660    // 11001 10 01100000          IMM          %2      0.75
//%2 = 0.75 * num1
4e00    // 10011 10 00000000          MUL          %2      %0
Store 0.5 to %3
6740    // 11001 11 01000000          IMM          %3      0.5
//%3 = 0.5*num2
4f01    // 10011 11 00000001          MUL          %3      %1
//%2 = %2+%3
2a03    // 01010 10 00000011          ADD          %2      %3

//%0 = num1 * (-0.5)
5cc0    // 10111 00 11000000          MULI   %0      -0.5
//%1 = num2 * 0.75
5d60    // 10111 01 01100000          MULI   %1      0.75
//%0 = %0+%1
2801    // 01010 00 00000001          ADD          %0      %1

3a14    // 01110 10 00010100          ADDI   %2      20
1200    // 00100 10 00000000          ST_1   %2

38ec    // 01110 00 11101100          ADDI   %0      -20
0000    // 00000 00 00000000          ST_0   %0
```

Design a Block Diagram showing your modules, data signals and control signals (do not use Quartus generated RTL diagrams)



2. Overall architecture of the design and simulations

As shown in the design block diagram mentioned the implementation of picoMIPS can be divided into two main parts: the Data path, consisting of ALU and General purpose registers GPR), and the Control path, consisting of a program counter and a program memory.

ALU

After having a draft of the general architecture of the processor, and after making decisions in the scope of instruction format/set and memory content, firstly, the original ALU module has been modified. Because affine transformation requires only addition and multiplication, extra ALU function codes have been deleted and the RMUL function has been added to the module. It should be noted that input variables (a and b) were modified to be signed variables and at the same time signed multiplication has been implemented as a separate module.

The main detail to be considered in the scope of the multiplication was the bit extraction process. The temp variable shown in Figure 1 represents a 16-bit result gotten from the multiplication, while in the case of the RMUL function call, needed 8 bits are extracted and stored as a final result. To test the ALU functionality separate testbench has been implemented, where all 4 ALU mods have been tested as shown in Figure 1.

```
// create the ALU, use signal ar in arithmetic operations
always_comb
begin
  //default output values; prevent latches

  result = a; // default

  case(func)
    `RA    : result = a;
    `RB    : result = b;
     `RADD  : begin
         result = a+b; // arithmetic addition

        end
    `RMUL: begin

        result = temp[14:7];
    end

   endcase


 end //always_comb

endmodule //end of module ALU
```

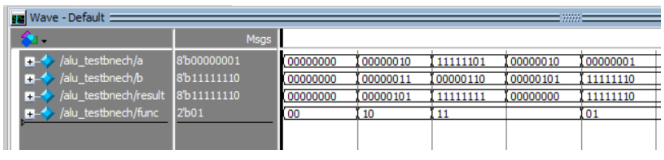
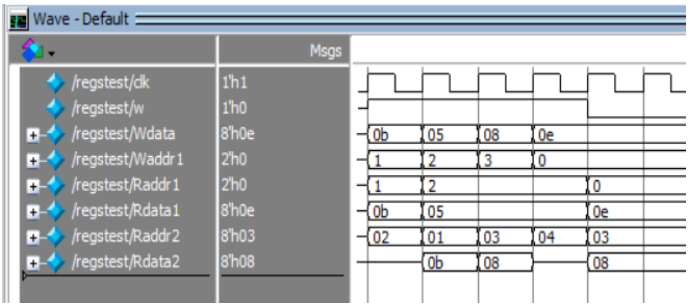


Figure 1: ALU module and ModelSim Simulation

GPR

As a next step, the general purpose register module was modified in different ways – Because in the mentioned architecture register %0 is used to store data, extra code has been deleted from the always_comb block, and the write address (Waddr1) variable has been introduced in the scope of the write process. To test GPR's functionality, I have created a separate testbench module and verified that both, read and write processes function as expected. If the write enable variable, w is 1, at a positive edge of the clock Wdata content should get stored in gpr's Waddr1 address. At the same time, it should continuously read from two addresses, which also has been verified in Figure 3.




```
// write process
always_ff @ (posedge clk)
begin
    if(w)
     gpr[Waddr1] <= Wdata;

end

always_comb
begin

    Rdata1 = gpr[Raddr1];

    Rdata2 = gpr[Raddr2];
end
```

Figure 3: GPR Modelsim Simulation and GPR module's main part

Program Counter (PC)

Because program memory consists of 16 instructions, the Psize variable was reduced to 4. Through the mentioned way it became possible to avoid the use of branches as the PC always jumps back to 0, after counting to 15. Thus, the main objective of the program counter in the mentioned architecture is to return to 0 after resetting and counting to 15, incrementing by 1 when the PCincr flag is 1, and to remain at the same value if PCincr is 0.

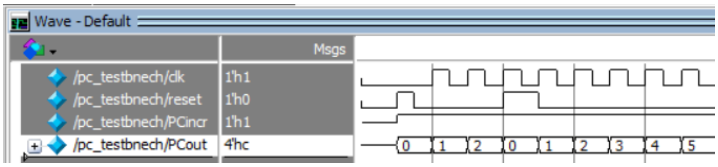


Figure 4 demonstrates that the PC is counting as expected and after reset its value is returned to 0.

Figure 4: PC Modelsim Simulation

Program Memory

Considering the address width and instruction width, program memory has been implemented. As shown in Figure 5, the main objective of the main module was to declare memory correctly considering the parameters, and get memory content from the modified prog.hex file and read instructions according to the address that acts as an input in the mentioned scope. After synthesis, the mentioned module was synthesized as 480 bits RAM block (figure 6).

```
// program memory declaration, note: 1<<n is same as 2^n
logic [Isize:0] progMem[ (1<<Psize)-1:0];

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

// program memory read
always_comb
  I = progMem[address];

endmodule // end of module prog
```

Figure 5: Main content of mem module

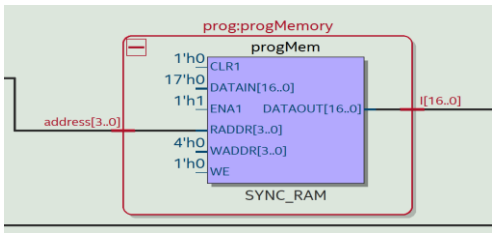


Figure 6: Synthesised progmem block

Decoder

Decoder stands as an essential module in the scope of processor architecture that is mostly responsible for executing instructions in the right way and provide triggers to external components based on input variables. Considering the fact that generally program counter's value should get incremented by 1 after every instruction execution, the default value for PCincr has been set to 1. SWI, ST_0, and ST_1 are instructions, where their progress depends on a user's unput (If one switches SW8 to 0/1).

The last 2 bits of opcode have been used as an ALUfunc indicator. Mentioned logic has been kept from the original decoder example. At the same time, flags called imm and immSW, help to implement the logic of reading the last 8 bits of instruction as an immediate value and storing values from switches.

A separate testbench module has been created to test the decoder, where assertions have been implemented to achieve efficiency during testing.

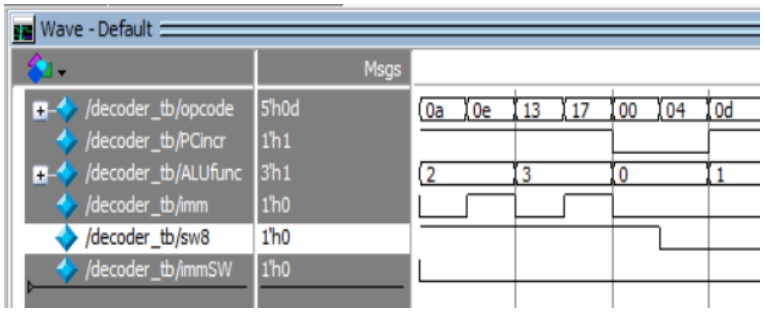


Figure 7: Decoder Simulation

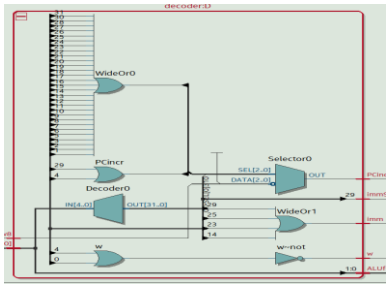


Figure 8: Decoder block

After compiling all the project files, ModelSim waveforms have been observed (figure 7) so to make sure that by switching between the opcodes, the right ALUfuncs are called. After synthesis decoder module has been built just by using simple logic gates and multiplexers Figure (8).

picoMIPStest and CPU

After all five important modules have been developed, the CPU encapsulating module has also been modified according to the used parameters and used variables to allow integration and simple program execution. Because both original modules covered most of the details, not that many modifications have been made.

Finally, a testbench module has been developed for picoMIPS (Using the fast clock), where several examples have been tested. Through the ModelSim simulation, it was possible to verify that reset, SW8, and display functionalities work properly and one could proceed with the FPGA implementation. As shown in Figure 9, for inputs 3 and 6 correct results have been obtained and what is the most important program has automatically returned to the first instruction.

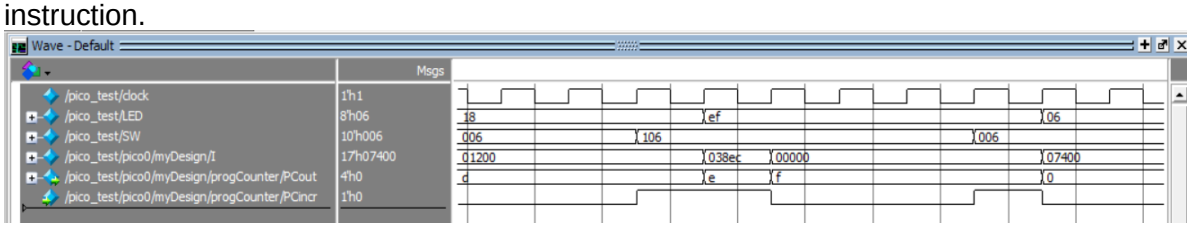


Figure 9: picoMIPStest module's simulation results. As a result of triggering SW8 results diplaying on LEDs change accordingly and instructions get executed as expected.

3. FPGA implementation

The FPGA implementation process involved several stages, including synthesis, programming, testing, and optimization. After synthesizing the picoMIPS architecture, FPGA has been programmed using the generated bitstream file. During the initial testing stage, picoMIPS module testbench cases have been used to verify the functionality of the FPGA implementation. This allowed us to ensure that the simulation results matched the real implementation.

Throughout the testing process, several modifications were made to the hardware configuration, such as adjusting the clock source from fast to slow and correcting the switch/LED pin assignmets. At the same time, to optimize the FPGA implementation, Quartus optimization settings have been modified to prioritize area minimization. By changing the settings (Figure 9) from “Balanced” to “Area” the number of ALMs used in the design was reduced by 5, which improved the overall area usage. Although mentioned optimization reduced the processor’s performance, it provided a more compact implementation that meets the design requirements.

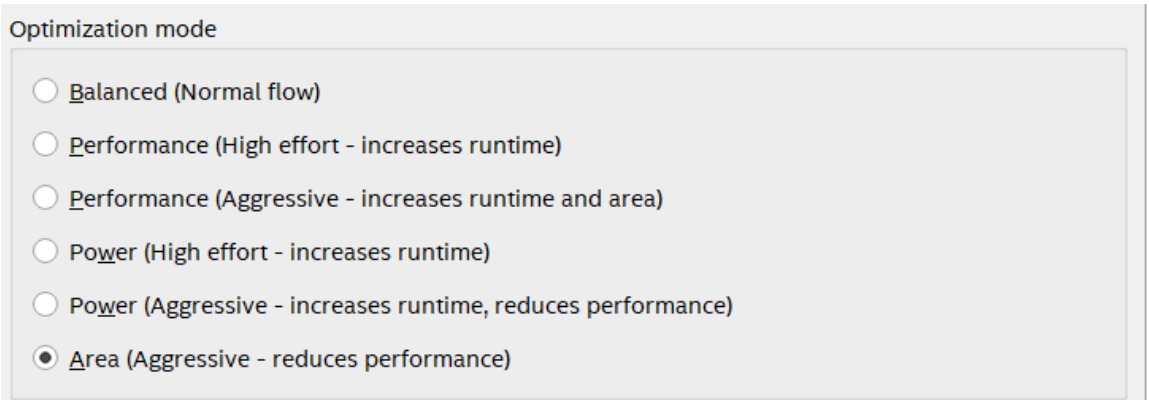


Figure 10: Quartus Optimization Modes

4. Conclusion

In conclusion, the successful implementation and design of a custom picoMIPS architecture for affine transformations demonstrate the adaptability of processor designs. The mentioned assignment provided valuable insights in terms of modifying a processor architecture, component interactions, simulation, and FPGA implementations.

While the current design works efficiently, there are opportunities for improvement. Advanced optimization techniques could be explored to reduce area usage or improve power efficiency. On top of that, the instruction set can be researched more to make it shorter and more compact. Additionally, the processor’s capabilities could be expanded by adding more specialized instructions or integrating the processor into a larger system.

Overall, this assignment has been a valuable learning experience in custom processor design, testing, and optimization, with experience gained that can be applied to future engineering challenges.

5. References

[1] ELEC6234 Embedded Processor Lecture Notes
[2] Intel Corporation. (n.d.). Intel® Quartus® Prime Pro and Standard Software User Guides. Retrieved from https://www.intel.co.uk/content/www/uk/en/support/programmable/support-resources/design-software/user-guides.html
[3] Terasic Technologies. (n.d.). DE1-SoC Development Kit. Retrieved from https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=836