Project 1: Implementing MIPS Based Processor in Xilinx Vivado

Lei(Raymond) Chi
Albert Nerken School of Engineering
The Cooper Union for the Advancement of Science and Art
New York, USA
lei.chi@cooper.edu

Abstract

The project's objective was to implement a MIPS architecture processor within Xilinx Vivado. Specifically, the project aimed to design a 32-bit MIPS CPU incorporating essential functional components such as the Program Counter, Instruction Memory, Control Unit, Register File, ALU (Arithmetic Logic Unit), and Data Memory.The implemented CPU successfully includes core instructions such as ”add,” ”sub,” ”and,” ”or,” and ”slt.” To check the CPU's functionality, the testbench from the textbook ”Digital Design and Computer Architecture” by David Money Harris and Sarah L. Harris was used. This testbench testing was to ensure the CPU's correctness.Furthermore, the CPU underwent synthesis and implementation processes, ended in a successful post-implementation simulation, confirming the correctness of the CPU's implementation.

Index Terms

AMD Xillinx Vivado, 32-bit MIPS architecture, CPU, System Verilog, Processor, Hardware Design

I. Introduction

During the mid-20th century, The concept of the Von Neumann architecture was developed by the mathematician John von Neumann. This is a revolutionary development in the world of computing. The idea laid the basic groundwork for modern computer processors through introducing a shared memory space for the data and instructions. This development also helped the creation of a lot of different instruction set architectures(ISA) for modern computers to preform either more complex programs or faster run-time, such as recursion.

MIPS architecture's desgin philosophy follows RISC(Reduced Instruction Set) on the emphasize of its simplicity and reduced instruction set. MIPS was designed more so for teaching and learning purpose. The instruction set is fixed-length and follows a format. The MIPS instruction sets are consists of three different types: R-format, I-format, and J-format. The below table would break the 32-bit instruction into sections.

Name	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	Notes
R-formant	op	rs	rt	rd	shmt	funct	arithmetic, logic
I-format	op	rs	rt	address	/	immediate(16)	Load/store, branch, immediate
J-format	op	target		address		(26)	jump

The breakdown of the instruction set above is designed after the assembly code is translated into machine code (comprising zeros and ones). In MIPS, each instruction is represented as a 32-bit long number, and each type of instruction can be further broken down into different functionalities. The difference in instruction length is determine by whether the user cares more about the variety of functionality or the speed of the CPU. The longer bit length of the instruction means more functionality can be included, whereas, shorter means the CPU can process the instruction faster. In this project, all types of instructions are covered, and these instructions were thoroughly tested using a testbench to confirm their functionality.

There are different architectural systems, such as RISC-V and ARM. As mentioned earlier, RISC-V is designed to offer simplicity, flexibility, and adaptability, while ARM (Advanced RISC Machine) is renowned for its power efficiency. These architectural systems provide distinct features, and users should choose the one that best suits their specific needs.

Furthermore, this project involves a single-cycle MIPS machine, signifying that the processor processes each instruction in one cycle. An option to enhance the machine's performance is by implementing pipelining. Pipelining is a common technique that divides the execution of instructions into multiple stages, allowing these stages to be processed concurrently. Consequently, the CPU becomes capable of executing multiple instructions in parallel. However, it's essential to note that this technique has not been incorporated into this project; instead, it remains a standard single-cycle 32-bit MIPS machine.

II. Building Blocks

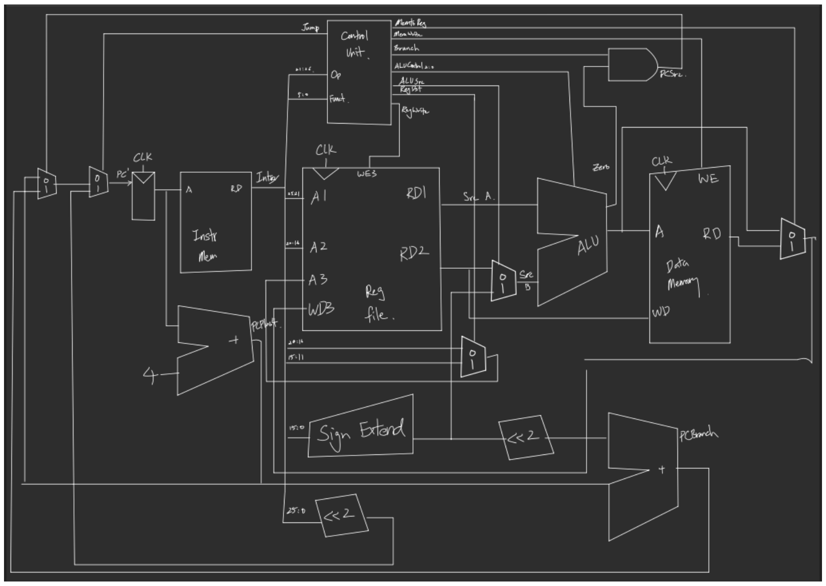


Fig. 1. MIPS single cycle processor schematic

Figure 1 is a diagram that displays all the basic functional blocks in a MIPS machine, such as the register file, sign extension, control unit, and ALU, among others. It also illustrates the inputs and outputs of each functional block. By following this schematic, developers of the CPU can navigate the I/O of each block and understand the data flow. The project proceeds by implementing each block as a module in SystemVerilog and connecting them all together.

As for Figure 2, the diagram provides a more detailed schematic of the functional components within the MIPS processor core. It consists of the controller, datapath, instruction memory, and data memory. The controller encompasses both the ALU decoder and the main decoder. It is responsible for controlling instruction execution, managing data flow, and processing subsequent operations. In the MIPS machine, the datapath is assigned to execute instructions, manipulate data operations, and process data. The instruction and data memories are considered external memory components: one stores and supplies instructions to the CPU, while the other stores and provides the data required by the CPU during execution.

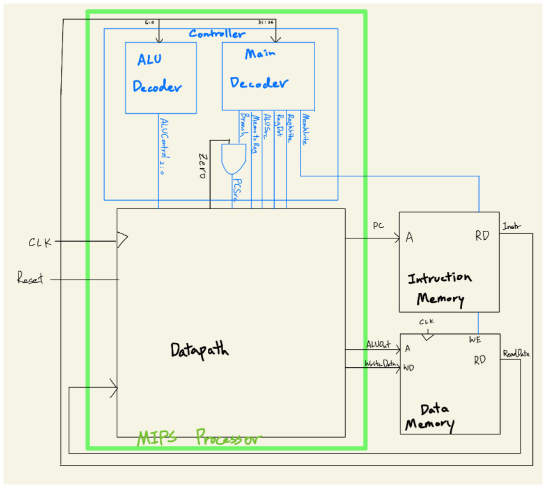


Fig. 2. MIPS single cycle processor with memory

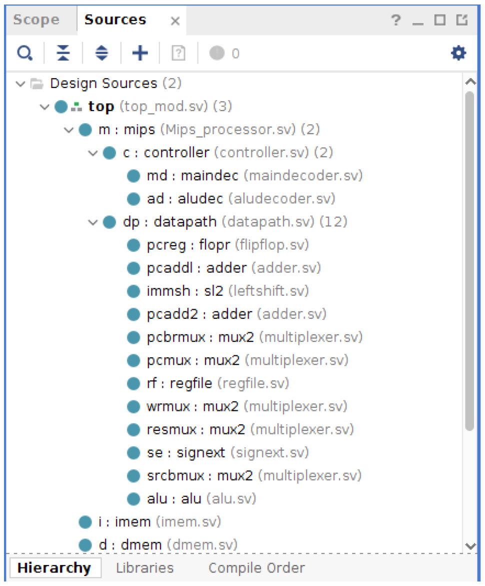
III. Methodology

As mentioned earlier, the project was developed by creating each block as a separate module in SystemVerilog. Each of the modules was coded behaviorally and using dataflow modeling. In SystemVerilog, there are three different types of coding models: behavioral modeling, structural modeling, and dataflow modeling. Behavioral modeling primarily focuses on describing how the module should behave, meaning it describes the desired output behavior, often utilizing constructs like ”always.” Structural modeling, on the other hand, deals with the internal structure of the module, including all the logic gates within it. Lastly, dataflow modeling involves handling input signal operations and assignments, which is why readers may notice assign statements in dataflow modeling.

Most of the SystemVerilog coded modules in this project were following the cookbook of ”Digital Design and Computer Architecture”, where most of the code can be found in Chapter 7. Figure 3 shows the module hierarchy after every module was connected together. Firstly, the CPU is all stored under one module called the top, which is the MIPS top-level module. This module fully represents Fig 2. where the single cycle MIPS processor is connected with the external memory interface which is the imem(instruction memory) and dmem(data memory). Then, inside the MIPS single-cycle processor are the controllers and the datapath, which are mentioned before. Then, the reason datapath is the core functional block that deals with all the data calculation is that it consists of all the modules: flipflop, adder, leftshift, multiplexer, register file, sign extension, and ALU. Most of the modules in Datapath are self-explanatory, such as adders are for address incrementation, muxs are for selecting signals, and register files consist of a set of registers for data storage, data manipulations, and temporary storage. However, ALU is not included in the cookbook, hence, we will further explain in depth.

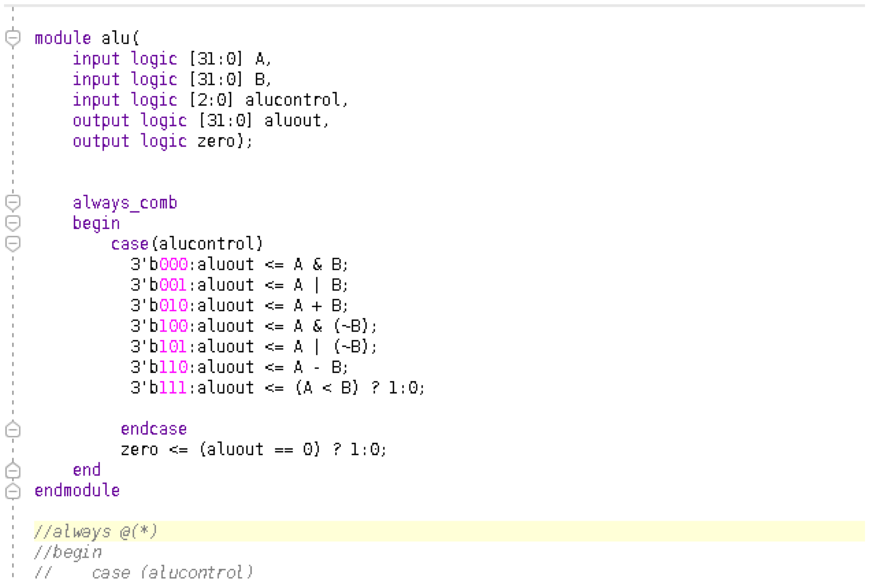
Figure 4 is the code that the aurthor coded for this project’s ALU(Arithmetic Logic Unit). The ALU is in charge of performing arithmetic and logic operation to the input signals to get the output of it, which can be view as the calculator.

- Input:
  ’A’ and ’B’ are 32 bits inputs and ’alucontrol’ is a 3 bits control input that determines the operation.
- Ouput:
  ’aluout’ is a 32 bits output that is the results after operation between A and B. ’zero’ is a 1 bit output that indicates if the result is zero.



```
Scope   Sources
Design Sources (2)
  top (top_mod.sv) (3)
    m : mips (Mips_processor.sv) (2)
      c : controller (controller.sv) (2)
        md : maindec (maindecoder.sv)
        ad : aludec (aludecoder.sv)
      dp : datapath (datapath.sv) (12)
        pcreg : flopr (flipflop.sv)
        pcaddl : adder (adder.sv)
        immsh : sl2 (leftshift.sv)
        pcadd2 : adder (adder.sv)
        pcbrmux : mux2 (multiplexer.sv)
        pcmux : mux2 (multiplexer.sv)
        rf : regfile (regfile.sv)
        wrmux : mux2 (multiplexer.sv)
        resmux : mux2 (multiplexer.sv)
        se : signext (signext.sv)
        srcbmux : mux2 (multiplexer.sv)
        alu : alu (alu.sv)
    i : imem (imem.sv)
    d : dmem (dmem.sv)
Hierarchy   Libraries   Compile Order
```

Fig. 3. CPU Module Hierarchy



```
module alu(
    input logic [31:0] A,
    input logic [31:0] B,
    input logic [2:0] alucontrol,
    output logic [31:0] aluout,
    output logic zero);


    always_comb
    begin
        case(alucontrol)
          3'b000:aluout <= A & B;
          3'b001:aluout <= A | B;
          3'b010:aluout <= A + B;
          3'b100:aluout <= A & (~B);
          3'b101:aluout <= A | (~B);
          3'b110:aluout <= A - B;
          3'b111:aluout <= (A < B) ? 1:0;

         endcase
         zero <= (aluout == 0) ? 1:0;
    end
endmodule

//always @(*)
//begin
//    case (alucontrol)
```

Fig. 4. SystemVerilog code for ALU

- Behavior:
  The uses behavioral code ’always-comb’ and ’case’ statement to switch operation, such as 000 from alucontrol would be ’and’ing the two 32 bits inputs.

Results and Discussion

The cookbook provided a testbench to check to functionality of the CPU. The testbench that the book provided is fairly simple to understand. The testbench initializes an oscillating clock and then checks if the CPU stored a number 7 on the address of 84. The assembly code is also provide, which the author only has to load the assembly code into the instruction memory. Assembly code basically means a set of instructions, which the result of these instructions output should output a 7 on address 84. In order to check if the CPU is working running the behavioral simulation would show if the code is running under behavioral simulation. However, the project was aimed to be able to implement, therefore, successful implementation is also important.

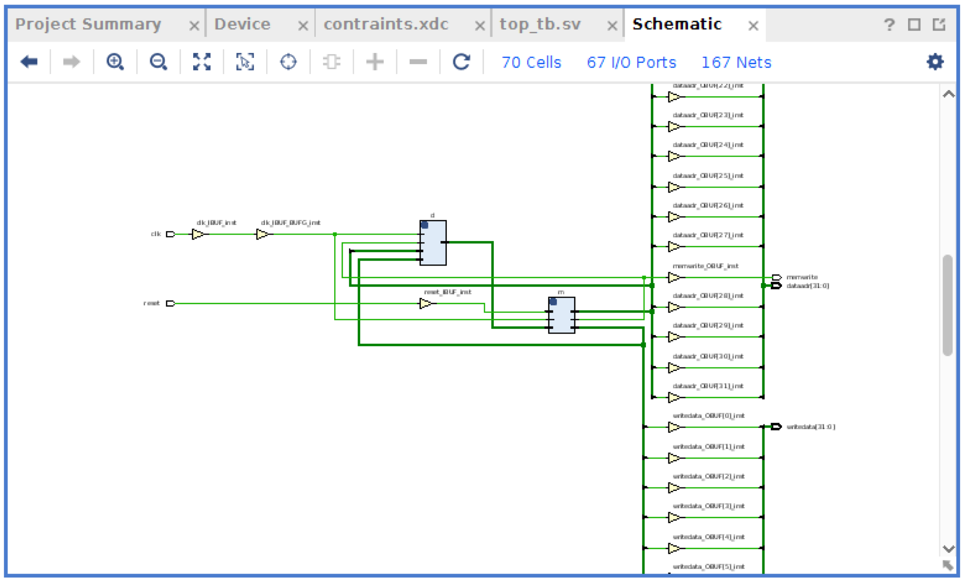


Fig. 5. RTL analysis - open elaborate design schmetic

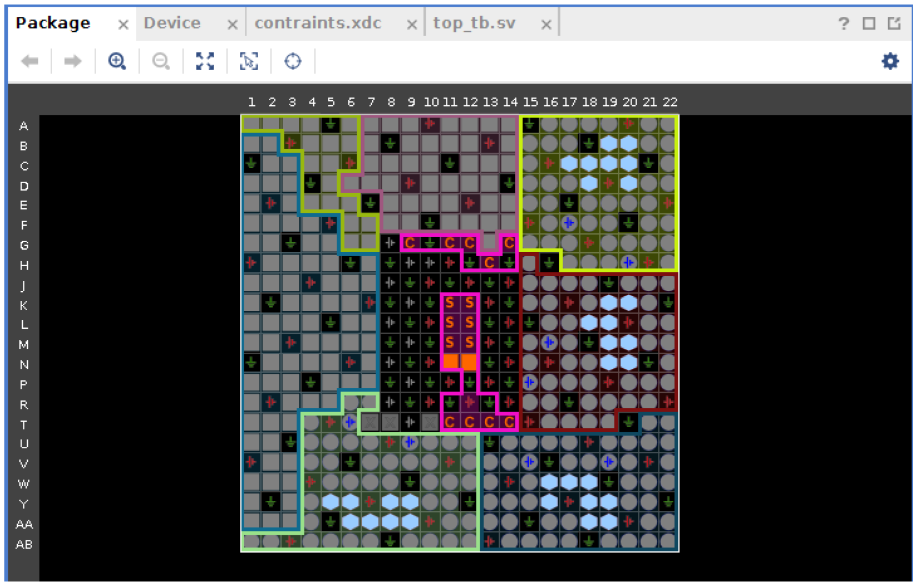


Fig. 6. implementation schmetic

Figure 5 and 6 shows that the RTL analysis and implementation was successfully operated. These two figures also shows that the behavior simulation is good and if implementation schematic shows up which means that both implementation and synthesis had operated smoothly.

The only thing left to do is to run the simulation over post implementation timing simulation, which in Fig 7, it is clearly shown that the results are 84 and 7. Furthermore, in Figure 8, the readers can clear identify the Worst Negative Slack(WNS) is positive, meaning the design is fully routed. If it is a negative number then it means that at least one time the design is not meeting the timing, but that is not the case for this CPU design.

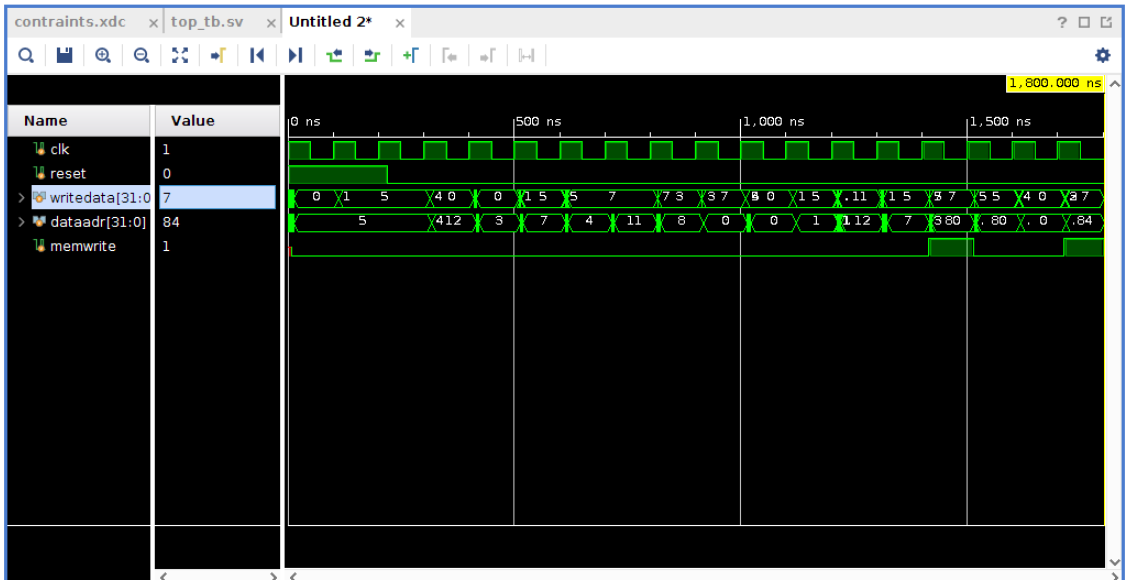
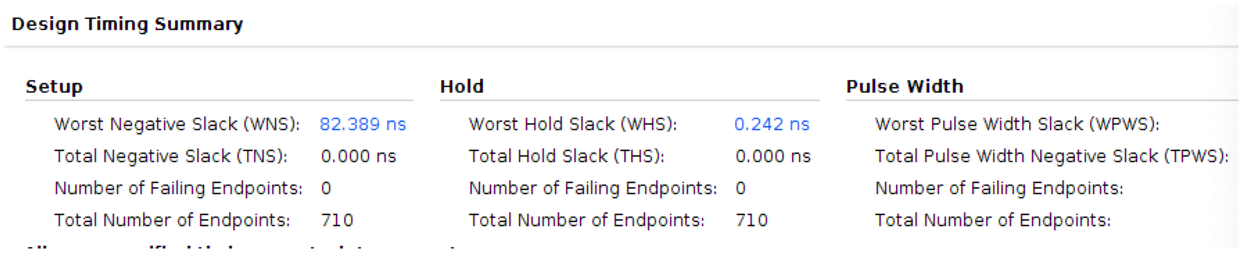


Fig. 7. post implementation timing simulation waveform



Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	82.389 ns	Worst Hold Slack (WHS):	0.242 ns	Worst Pulse Width Slack (WPWS):	
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	
Total Number of Endpoints:	710	Total Number of Endpoints:	710	Total Number of Endpoints:	

Fig. 8. slack report

Throughout the process of designing and building the CPU, the author enjoyed integrating the knowledge into a new technology(vivado) and also taking the time to debug. This project took a long time for debugging but the lessons that learnt from it is far more satisfying and fruitful than building the CPU. The documentation is something that the author hates the most, since writing is not one of his strength and having to write a report with professional tone is the most challenging part. The author also notice that the hardest part about this project is to figure out why post-implementation simulation would have a different result than behavioral. It turns out that the default simulation delay values are too insignificant the implementation cycle could not catch up, which cause identify signals. In order to fix the problem, the readers are suggested to increase the default delay by a power of 10. After this project, the author is fully comfortable with Xilinx Vivado and has a thorough understanding of how a MIPS CPU works. Hopefully, pipelining can be integrated in the future.

Conclusion

In conclusion, the author had built a fully functional MIPS CPU with implementation working. The above report shows that the author has a in-depth understand of the material and the technology. The author would like to try implementing either pipelining or cache in the future.

Acknowledgment

Firstly, R.C. would like to express his thanks to Professor Hoerning for his guidance and supervision throughout the entire project. He would also like to express his appreciation to ECE 311 Hardware Design for providing him with the opportunity to work on Xilinx Vivado and the ZedBoard.

References

[1] Harris, D. R., Harris, S. (2007). Digital design and computer architecture. In Elsevier eBooks. https://doi.org/10.1016/b978-0-12-370497-9.x5000-8