Verilog Implementation of MIPS CPU

I. Introduction

For our project, we designed and implemented the pipelined MIPS CPU outlined in Chapter 4 of the Patterson and Hennessey [1], but with a slightly expanded instruction set. Our language of choice was the Verilog hardware description language (HDL). Our development was split into three stages: first we implemented the single-cycle processor with the reduced instruction set, then we added pipelining to our design, and finally we expanded the instruction set to allow for running our (slightly modified) assignment 2 quicksort program. Our single-cycle design corresponded to the Figure 4.24 on page 271 of the textbook, while our pipelined design corresponds to Figure 4.65 on page 325, including the logic to handle data hazards and control hazards. In terms of development, the first step of our project was to familiarize ourselves with the various tools, specifically Icarus Verilog and the waveform viewer. More information on the tools are found in Appendix 1 (user guide). To ease development we split the program into Verilog modules, and distributed the modules based on experience. More information about the modules is in section II, and section III outlines how we were able to verify the correctness of our implementation during the various stages of development using the MARS simulator. Finally, Appendix 1 is a user guide including instructions on how to set up, run, and modify the program.

II. Architecture

Our Verilog design was broken up into modules in a way that roughly matched the pipeline architecture described by the textbook. In addition, we extended this architecture to support jumps and function calls (*j, jal, jr)*. Figure 1 provides a block-diagram view of the data and control paths implemented by our final pipelined design, including the hazard detection and forwarding units. Not shown in this diagram are the additional control signals and muxes that were added to support the instructions *j*, *jal*, *jr,* and *bne* which can cause control hazards.

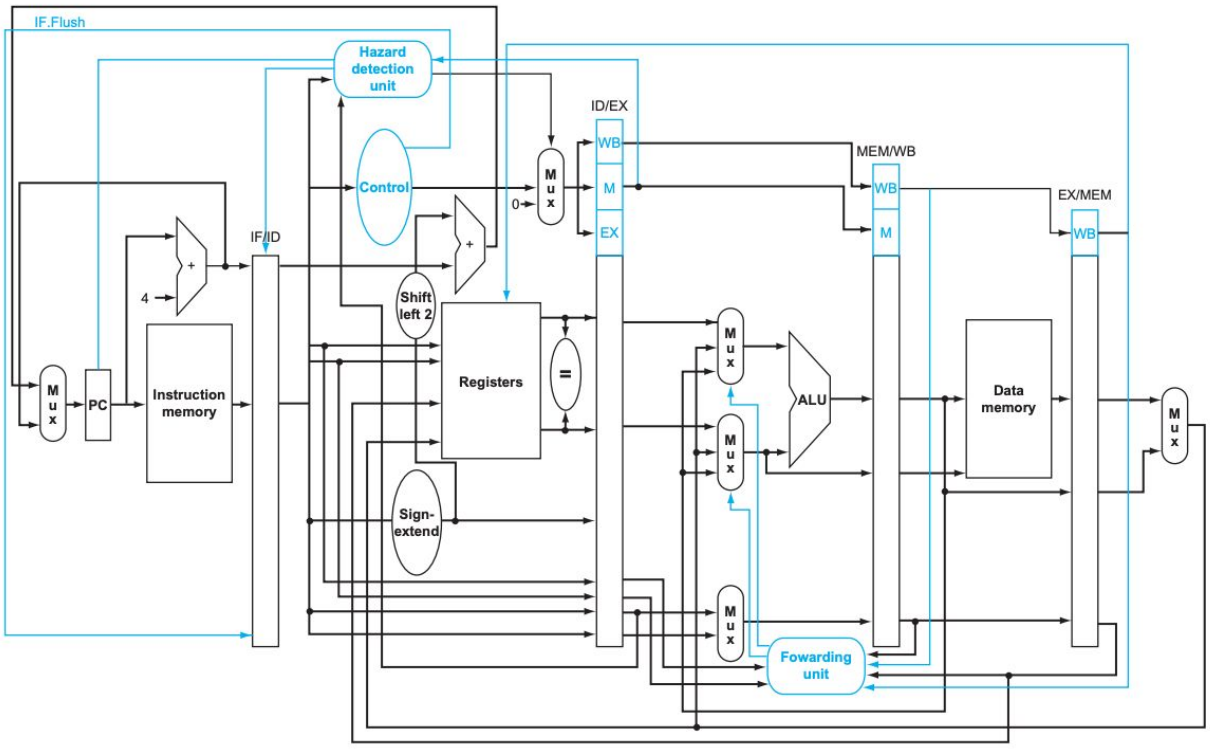

Figure 1: Overview of the pipelined data and control paths [1].

CPU

The CPU module was the top-level module that instantiated the other sub-modules described below. It also contained the control logic, program counter and pipeline registers, as well as the necessary bubbling/flushing logic to resolve control hazards and jumps. The cpu module coordinates the timing of all the sub-modules, ensuring that the register file and memories are accessed at the appropriate clock edge and moves the data through the pipeline.

Our design took a simple approach for resolving control hazards with branches; we assume that branches are never taken. That is, instructions immediately after the branch instruction are always loaded into the pipeline. This was the simplest strategy to implement and sufficient for our purposes. Data hazards were resolved by the forwarding and hazard detection units which are detailed below.

Register File

The register file contains 32 32-bit general-purpose registers, two read ports, and one write port. For each data word to be read from the registers, we had an input to specify the register number to be read and an output to carry the value that had been read from the register. To write a data word, we had four inputs: a register number, the data to write, a write signal and a clock that controls the writing into the register. The write of the register file occurs on the positive clock edge. In addition, the reset signal at the positive edge would reset all of the registers.

Memory

MIPS is a Harvard architecture where there are separate hardware blocks for instruction and data memory. To reflect this we implemented separate modules: instr_mem, and data_mem respectively. For simplicity, we limited instruction memory to 4 KB and data memory to 12 KB. Both were simple to implement and required little code. Additionally, neither module required modifying to extend the single-cycle implementation to include pipelining.

Data Memory

Data memory was relatively simple to implement. Upon a reset signal, data memory is completely zeroed out. Additionally, the memory write control signal is checked synchronously at every positive edge of the clock cycle. If this signal is high we fill the input address with the input data. This is done through an if statement inside an always block. Initially, we had data memory also checking the memory read control signal synchronously, however, this produced a bug when we ran our test_3 program. In fact, we needed the memory read to be happening asynchronously and modified the memory read implementation to an assign statement outside of the always block.

Instruction Memory

This module was perhaps the simplest to implement, taking one input and giving one output. Its task is straightforward, taking the address of the next instruction delivered by the program counter (PC) and outputting the instruction at that address. To match the MARS

simulator memory layout we used an internal PC for this module to offset the input PC to the section of memory addresses MARS uses for instructions. To implement the reading of the instruction address we used an assign statement to asynchronously read the address and output its contents. We also added reset functionality that synchronously checks for a reset signal at every positive edge of the clock. If instruction memory receives this signal it wipes instruction memory by filling it with zeroes.

Arithmetic Logic Unit

The Arithmetic Logic Unit (ALU) is a straightforward implementation of the ALU presented in the textbook (pp. 259 - 261). It is comprised of two verilog modules, the first is the alu_control module which is responsible for parsing the instruction. It takes as its inputs the funct from the instruction and the alu_op mux, and outputs a 4 bit control pattern to specify the operation to be performed. The second module is the alu itself. As inputs it takes two data arguments and the aforementioned alu control bits, and outputs a zero bit and the alu result. Initially the ALU only supported the add, subtract, and, or, and nor operations, but it was expanded after completing the pipeline to support the shift left, shift right, set less than, and set less than unsigned operations as well.

Forwarding Unit

The forwarding unit was added in order to handle data hazards in pipelining. The implementation was relatively simple and involved asynchronously checking conditions. Depending on which pipeline register the data is being forwarded from, the forwarding unit outputs zero, one or two. Signals are sent to two multiplexores that then select the correct data source for the ALU unit. This is implemented with a few if statements within an always block. The always block is set up asynchronously and so any change to the values of the inputs into the forwarding unit cause the conditions involving that input to be evaluated immediately.

Hazard Detection Unit

The Hazard Detection Unit is a special unit to detect load-use hazards within the pipeline. It specifically checks for the case where a register is used immediately after a value has been

loaded to it from memory. As inputs it takes the idex mem read control bit, the instruction in the ifid register, and the rt register in idex. It uses the exact logic presented in the textbook (p. 314), and it outputs a single bit which signals to the CPU whether or not it needs to insert a bubble after the lw instruction.

III. Testing

In order to verify the correctness of our implementation, it was necessary to test it. Initially, we tested individual modules using Verilog testbenches. However, once the CPU module and CPU testbench were operational, we dropped these testbenches in favour of assembly programs. The most straightforward testing method we came up with was to run our assembly programs in the MARS simulator, dump its memory and registers in plain text format, and compare the final memory and register values of our implementation to the MARS results. We automated this process by having our own implementation also dump its memory, and using the diff command within a makefile. This was crucial in helping us identify when we had problems in our implementation, and often reading the memory dump gave us clues on how to locate the problem.

When the memory dump wasn't enough, we would use a waveform viewer to view the output of our implementation cycle by cycle. This was especially helpful as it allowed us to see the values of each individual control bit and register and compare them to our expected values. We could then trace these incorrect values backwards in time and find the errors in our Verilog code.

While our test cases were mostly general in nature, we did make an effort to create test cases that targeted specific aspects of the architecture. For example, test_0 is a test that specifically tackles the load-use hazard and was used to debug our bubble-insertion. Test_1 contains a standard data hazard, and was used to test the data forwarding module. And so on, each test targeted a specific aspect of the single-cycle and then the pipelined architecture. Many tests evolved over the course of development. Data hazards, for example, were not a problem in the single-cycle implementation and therefore appeared frequently in our simple tests. These tests were no longer simple in the pipeline implementation.

Our testing culminated with ‘test_a2c’ which is a slightly modified version of the quicksort code from assignment 2, that runs identically on our design and the MARS simulator. This complicated program that uses the vast majority of the instructions we implemented gives us confidence that our design is functionally correct.

IV. Conclusion

Despite varying levels of HDL background, the single-cycle CPU proved not to be a challenge. This was thanks in part to the good division of tasks, but also to the in-depth diagrams present in the textbook. Completing the single-cycle implementation early gave us ample time to implement the pipelined architecture, which was just as successful. The clever use of the MARS simulator allowed us to quickly verify the correctness of our implementation at various stages of development. Given more time, we would have liked to implement more instructions, exception handling, and some micro optimizations noted in the textbook. Overall, this was a very fulfilling project which provided a deep dive into the specifics of the MIPS pipelined architecture.

References

[1] D. A Patterson and J. L Hennessy, *Computer Organization and Design*. Waltham, MA: Morgan Kaufmann, 2014.

Appendix A -- User Guide

Requirements

- Icarus Verilog simulator: http://iverilog.icarus.com/
- Java 8 for running the waveform viewer and MARS MIPS simulator
- Make
- Unix command line tools (diff)

Development

- To elaborate the design, run `make`
- To run the top level module listed in the Makefile (TOP variable), use `make run`. This can be changed to run a different testbench file.
- To run a test program, run `make <test_name>` where `test_name` is the name of the MIPS assembly file. For example, run `make test_1` to run test_1.asm.
- This will assemble the test file using MARS, and run the assembled executable on both the MARS simulator and our design. A text-formatted dump of the registers and data memory is then compared using diff.
- To run the full suite of tests, run `make regression`
- To view a VCD waveform produced by the simulator, run `make waves`.

Supported Instructions

- lw, sw
- lbu, sb
- beq, bne
- j, jal, jr
- addi, addiu
- slt, sltu
- and, or, nor
- add, addu
- sub, subu
- srl, sll
- syscall (only for terminating program, no exception handling)

Memory Layout

Our CPU uses roughly the same memory map as MARS run in "CompactDataAtZero" mode.

Note that only the first 12 KB of space are currently used.

The address space is as follows:

- 0x0000-0x0FFF .data segment
- 0x1000-0x1FFF .extern segment (for global pointer)
- 0x2000-0x2FFF stack segment
- 0x3000-0x3FFF .text segment (in reality, separate instruction memory is used by our CPU)
- 0x4000-0x7FFF kernel segment (not modeled by our CPU)

Credits

Waveform viewer from https://github.com/jbush001/WaveView

MARS MIPS simulator from http://courses.missouristate.edu/KenVollmar/mars/