Ray Chang
Tim Kim
ECE 154B

Project 1: In-Order MIPS Pipeline

Introduction

The goal of this project is to design a 32-bit 5-stage MIPS pipelined processor that is capable of executing simple programs and resolving any potential hazards. To achieve this, the necessary core modules of the processor were designed and tested, then connected together in an initial datapath. After confirming the performance of the initial circuit, more modules were added and the datapath was edited in order to account for additional assembly instructions. After significant debugging and testing, it was determined that our circuit met the design requirements of the project. This implementation is coded in Verilog and simulated in ModelSim.

Block-Level Top Schematic

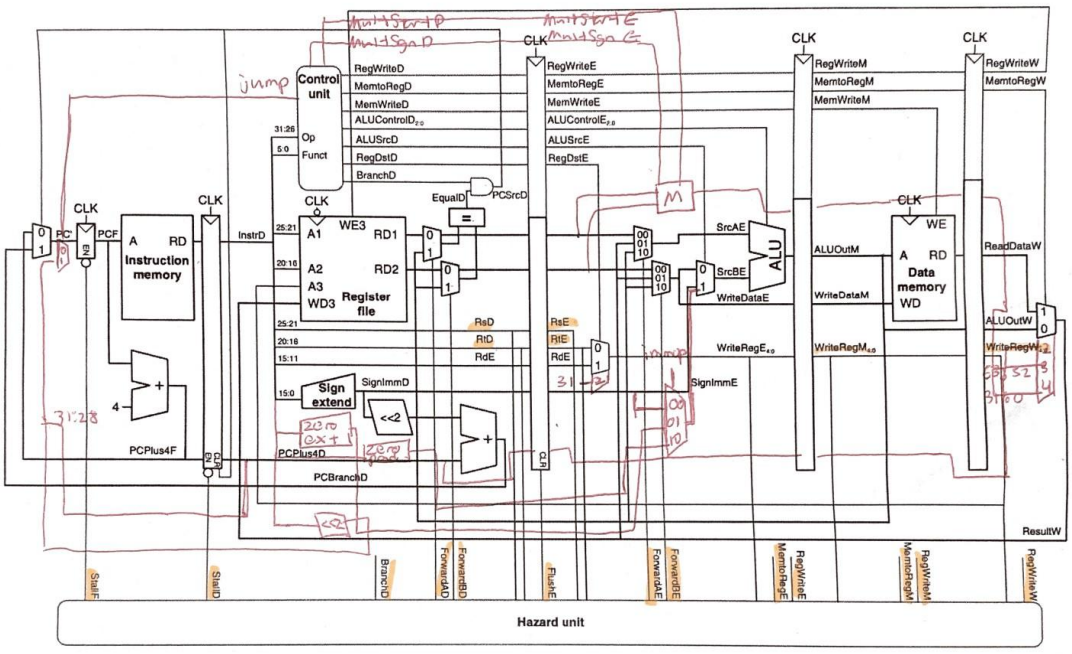


Figure 7.58 Pipelined processor with full hazard handling

Figure 1. Annotated Schematic of In-Order MIPS Pipeline

The schematic above is an annotated version of the in-order MIPS pipeline process that is implemented in this project. There are a few distinctions between the standard MIPS and the processor that is implemented in this project; the latter is capable of performing some additional instructions, such as XNOR, ANDI, BNE, and MULT. In order to make these instructions function properly, the controller module is designed to output additional signals that would give orders to certain multiplexers and the multiplier module. A new multiplexer is added to the

datapath, and it is responsible for choosing the correct immediate value, selecting a sign-extended, zero-extended, or zero-padded word to carry out certain instructions. A multiplier module is designed and implemented, and it is fully capable of performing either a signed or unsigned multiplication process. Finally, the writeback source multiplexer is extended to have three additional inputs, which are the pcplus4 data, the upper 32-bit array of the product, and the lower 32-bit array of the product.

Design Methodology

As explained in the introduction, a processor consisting of the most essential components are created, and an initial datapath is developed to connect these components to create the standard MIPS pipeline. Before implementing additional functionality, the original processor is sufficiently tested using multiple testbenches in order to ensure that there are no issues with the core of the project. In doing so, it is able to be determined that the fundamental build is functioning properly, and any upcoming problems are caused by future implementations. Then, the datapath is revised to include new wiring and circuitry needed for newer instructions. After every single new instruction is implemented, an adequate testbench is created to execute the instruction, and if any problems arise, signals and waveforms are tracked down to pinpoint the source of the issue. Such an approach is essential for this project, especially due to the possibility of hazards in certain sets of instructions. This methodology is repeated for each module, minimizing the potential for errors in future implementations. Finally, the hazard unit is implemented. The hazard unit is essentially the most important part of the project as it is the component which ensures the parallel structure of the processor is compromised. Due to this reason, extensive tests were done on the hazard unit to make sure that the processor worked correctly.

Modules Designed

1. ALU
   The ALU takes as input 2 32-bit arrays and a 3-bit control signal and outputs a 32-bit array. The control signal determines what operation to perform on the input data. This implementation supports 7 operations, including AND, OR, ADD, XOR, XNOR, SUB, and SLT.

2. Instruction Memory
   To set the instruction memory, this module reads from a memory file which includes machine code for each instruction to be executed. A program counter is used to select the next instruction and output it.

3. Data Memory
   The data memory module takes in an address from the ALU module and reads data from its memory asynchronously. On the rising edge of a clock, the data memory writes data onto the memory if the enable signal is high. If signal reset is high, the data memory does not change states.

4. Register File
   The register file inputs 2 5-bit arrays that represent registers, and it outputs 2 32-bit arrays that represent the words that the registers were holding. On the falling edge of a clock, the register file writes data onto the specified register if the enable signal is high. If the reset signal is high, then all registers in the file are wiped.

5. Branch Comparator
   The branch comparator is a module that compares two source inputs and compares them in order to determine whether a program should branch to a specific address or not. The module takes a control signal which determines whether the comparator is used for a beq instruction or a bne instruction.

6. Controller
   The controller uses combinational logic to output control signals for all of the modules that are used in the datapath of the pipelined processor. The controller takes the decoded instruction and ensures that correct signals are outputted so that the requested instruction is correctly carried out.

7. Multiplier
   The multiplier takes two 32-bit arrays and performs binary multiplication using the shift and add algorithm, where the multiplicand is added to an intermediate product if the least significant bit of the multiplier is 1, then the multiplier is shifted to the right by 1 bit, and the cycle continues until the multiplier is 0. This algorithm is used for both signed and unsigned cases, with the difference being that in the signed case, the leading bit of the inputs are XORed to find the sign of the product, and if it is 1, the product is converted to its 2's complement form. After calculations are completed, a ProductValid signal is set high and outputted so that the processor knows when the product is ready and available for use.

8. Datapath
   The datapath is the most complex module of the project and instantiates most of the modules designed such as the register file, ALU, instruction and data memory, and multiplexers. The datapath takes input from the controller and hazard units in order to

pass the correct data to modules. The control signals are passed from one stage to the next through registers, and the hazard signals

9. Hazard
   The hazard unit is one of the major modules as it resolves data conflicts by performing stalling, flushing, and data forwarding techniques. Several datapath and control signals are inputted to determine the output of the hazard signals. Our implementation successfully performs data forwarding and branch stalling, however it is incapable of load stalling and stalling during multiplication. Attempts were made to resolve these hazards but they were unsuccessful.

10. Mips
    The mips module consists of the controller, datapath, and hazard unit. It takes the program counter, instruction, and read data as inputs in order to perform a specific instruction. The controller takes in the instruction code and outputs control signals to the datapath and hazard unit. The datapath and hazard unit work hand-in-hand to properly execute the instruction.

11. Mips Top
    The mips top module instantiates a mips module, instruction memory, and data memory. The module represents a working MIPS pipeline.

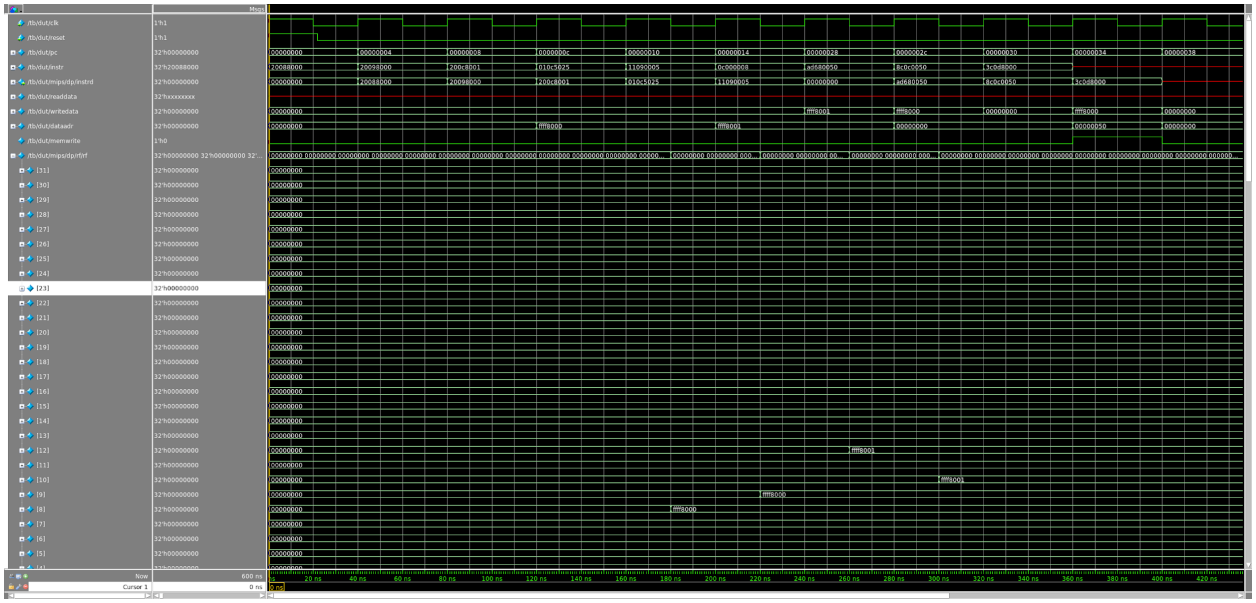
12. Memory File
    The memory files are used to load data into either the instruction or/and data memory for specific purposes. The data memory is preemptively loaded with data to test a certain functionality, but then is wiped after the test is complete. The instruction memory is always loaded with instructions to be carried out.

13. Testbench
    The testbench module is the main code to examine if the MIPS pipeline works correctly. A clock signal is generated in this module. The module is used to generate necessary waveforms which are used to test certain instructions and pinpoint the source of a problem.

14. Miscellaneous Modules
    The miscellaneous modules consist of n-bit multiplexers, n-bit adders, shifters, n-bit registers, zero-extenders, zero-padders, sign-extenders, branch comparators, n-input NOT gates, and n-input AND gates.

Questions

4.1: A short program was executed on our processor to check the performance of some common instructions. This program is located in memfile3.dat



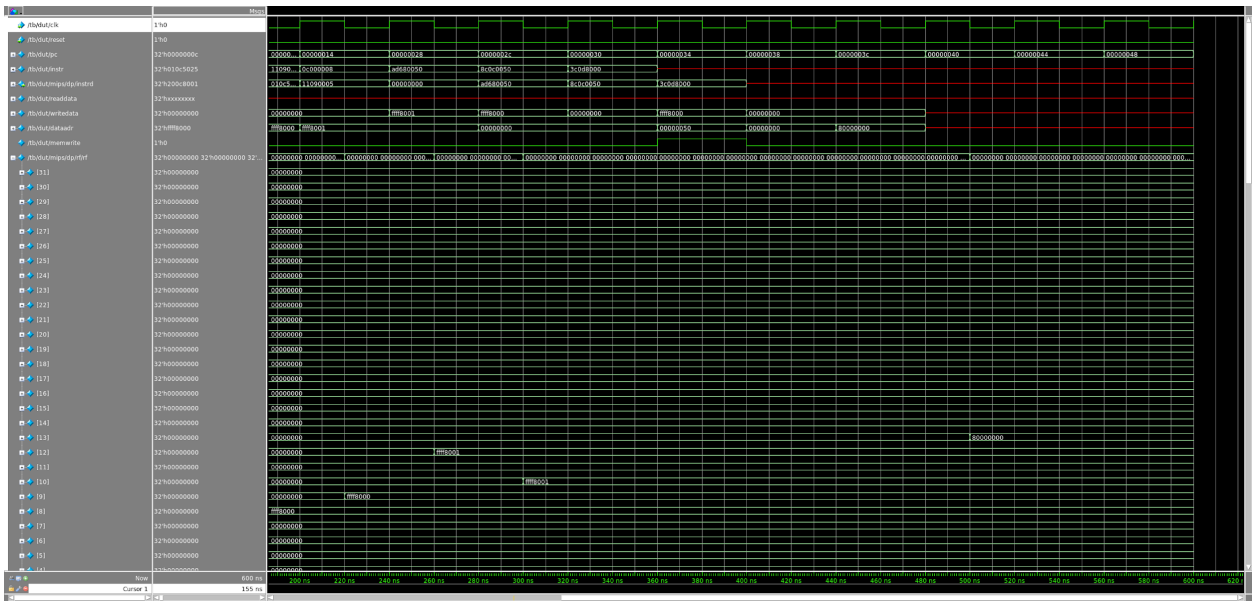


Figure 2. Waveform Results of 4.1

As shown above, the andi instruction is correct as the immediate is sign extended. Also, data forwarding is successful as $t2 holds the correct value even though the or instruction uses a register that has not been written to the register file yet. Also beq is successful as the program counter jumps to the store instruction. The correct value is stored to the correct location in memory and it is loaded to $t4. Finally the lui instruction sets $t5 to 0x8000000 which is the expected result. This program shows the correct output for many of the processor instructions.

4.2: A program is written in assembly code to calculate the dot product of two vectors consisting of five signed decimals each. The program stores both arrays into data memory, finds the product of each element of the vectors, then adds them up together to output the final product. This program is located in memfile4.dat. The waveforms of the final result are shown below.

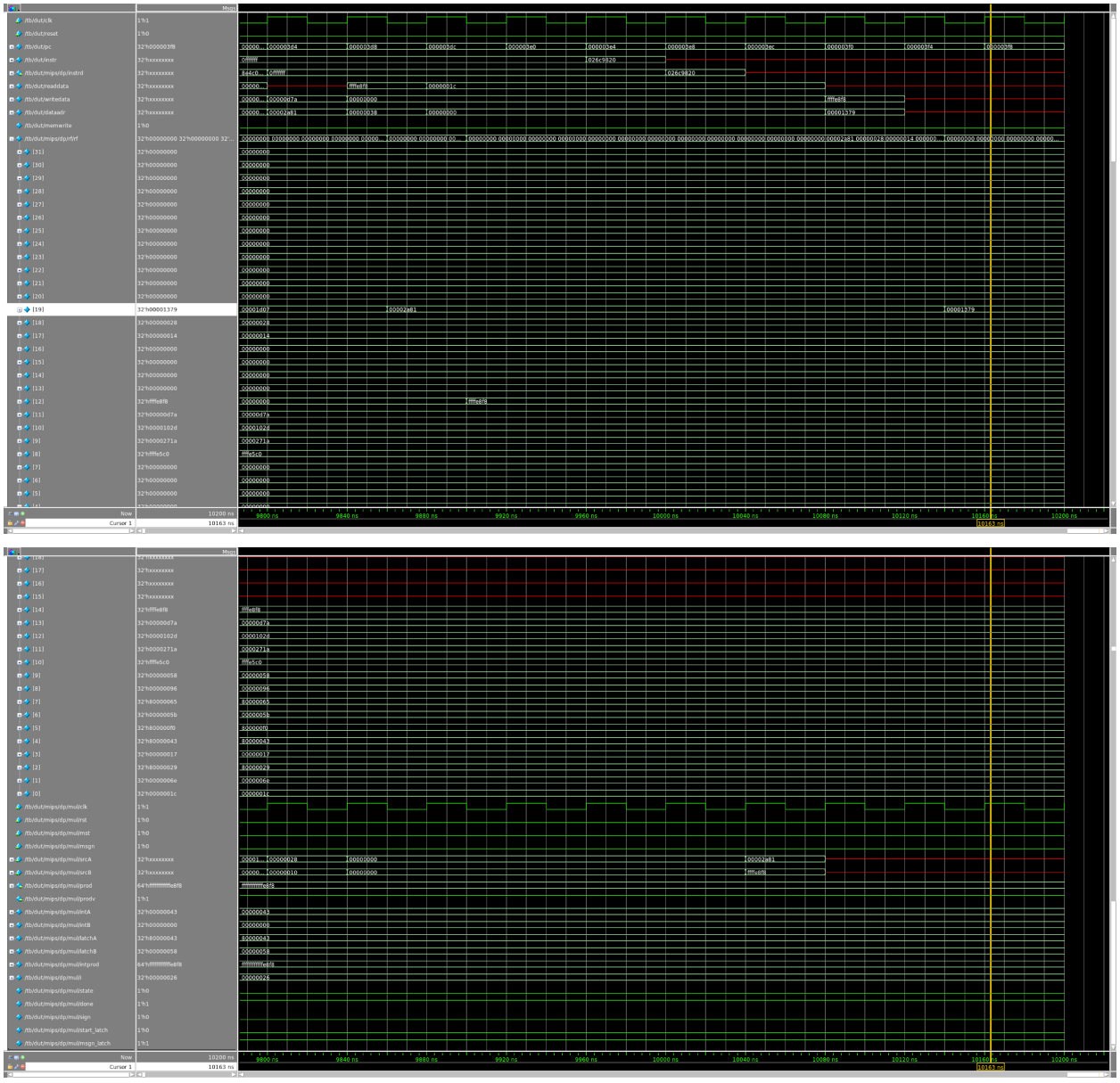


Figure 3. Waveform Results of 4.2

As shown above, register 19 which correlates to $s3, holds 0x1379, which is 4985 in decimal and the correct answer for the dot product.

The number of cycles it takes to complete the dot product operation is 253. The program is 250 instructions long including nops. Therefore, the CPI is 1.012.

$$number\ of\ cycles = \frac{time\ elapsed}{period} = \frac{10{,}140\ ns}{40\ ns} = 253\ cycles$$

$$CPI = \frac{number\ of\ cycles}{number\ of\ instructions} = \frac{253\ cycles}{250\ instructions} = 1.012$$

Conclusion

The approach we took to complete this project is what allowed us to be so successful in our implementation. We ensured that we had implemented a fundamental MIPS pipeline that was working correctly in order to eliminate the possibility of any core issues. We then implemented any additional modules and control signals, and rewired the datapath to make a complete MIPS pipeline. Finally, we developed a hazard unit and attempted to resolve any problems that were introduced by the parallelism of the processor structure. Through our simulations we proved that our implementation is capable of producing the correct values, however simply lacks support from the hazard unit to resolve all potential conflicts. In total, the project took about 30 hours to complete. Throughout the process, we experienced the arduous process of debugging and troubleshooting, but it helped us to learn from common mistakes in Verilog programming, such as incorrectly wiring signals, assigning wrong bit values to inputs and outputs, and failing to understand what is happening in the memory of the processor. All relevant code and waveform results are uploaded in the zip folder.