MIPS Architecture Implementation

The approach we took to the implementation of the single pipeline MIPS CPU in Verilog was fairly straightforward. We first examined the processor diagram for the single pipeline implementation and for every block in that diagram, we created a corresponding module in Verilog.

# Major Modules

* **reg\_file** – This was a register file consisting of 32 32-bit registers
* **CPU** – This is the main file for the whole project. It is the code file where all control wires, including RegDst, Jump, Branch, MemtoReg, ALUSrc, RegWrite, etc. are initialized and used in the modules that make up the CPU.
* **alu** – This is the arithmetic logic unit. It is capable of performing operations for logical and, logical or, add, subtract, slt, and nor.
* **controlunit** – This is the control unit responsible for reading in the opcode and generating values for RegDst, ALUSrc, MemToReg, RegWrite, MemRead, MemWrite, Branch, Jump, and ALUop
* **Memory** – This was a module provided by the professor that will reads values from a hexdump file and writes to a register file
* **signextender** – The sign extend module
* **alu\_control** – The ALU control unit that receives the ALUop signal from the control unit and then determines the operation that the ALU will perform in the CPU based on this signal and the corresponding instruction type.

We also included several basic modules needed for the CPU such as several different types of muxes, a left shift module, and a 32 bit register.

# Development Process

As mentioned previously, the design process for the CPU was simply to create corresponding modules for each block in the CPU block diagram. Many of these modules had already been implemented in homework 6, so this was a big help although we needed to make some minor changes. Among these changes was adding an asynchronous reset signal to reg\_file.v, reversing clock edges on certain files so that in all modules, the clock triggered changes on negative edges across the whole project, and altering operations on the ALU. The majority of control modules could be implemented with either a series of “if” and “else if” statements or a series of case statements. The datapath is laid out in the CPU module.

Because this was a partner project, we utilized some of the techniques we learned from previous group projects and working on co-op. The most important decision we made for collaboration was to use source control. We elected to use Github. With Github, we were able to create a remote repository to our code where we could regularly push and pull updates from so that we had matching code. This allowed us to work separately on the code and not trample over each other’s progress. Using source control also allowed us to revert to previous versions of our code in the case that we made a mistake. When we worked together, we used the technique of pair programming. In pair programming, one of us would write code while the other would review the code written while dictating what to do to the person writing code. This method allowed for maximum collaborative input from both team members as well as having the benefit of two pairs of eyes to catch simple mistakes that may have added a disproportionate amount of time later on in the process when trying to debug our code.

Once the code was all written, we were able to test our code. The tests we included in the final submission include both unit tests for individual modules as well as the functional test for the CPU unit that instantiates the CPU module as well as the memory module, which reads in from the hexdump file and assign the control signals to the CPU accordingly. This test is written in CPU\_tb.v. The output wave form of this test is included at the end of this report in three parts.

# Obstacles

We ran into several obstacles along the way while working on and debugging this project. The first obstacle we ran into consisted of the additions that we had to add to the modules from homework 6 that we mentioned in the development section. The other major obstacles occurred while debugging and consumed a good amount of time while we searched for the sources of error. The first was that we had some modules that were sensitive to the negative clock edge, as well as modules that were sensitive to the positive edge. Another issue we ran into was that our registers weren’t initialized. To solve this problem, we added a reset signal to the register files that when triggered in the testbench file would set them all to 0. Other difficulties included correctly configuring the project so that it would successfully read the hexdump file, as well as successfully making this project usable across our environments pulling from the source control, given that Model Sim projects use absolute paths.

# Final Results

The final register contents appear as follows:

R0: 0x00000000 R1: 0x00000000

R2: 0x00000000 R3: 0x00000000

R4: 0x00000000 R5: 0x00000000

R6: 0x00000000 R7: 0x00000000

R8: 0x00000028 R9: 0x00000000

R10: 0xfffffff0 R11: 0x00000015

R12: 0x00000015 R13: 0x00000001

R14: 0x00000000 R15: 0x00000000

R16: 0x00000000 R17: 0x00000009

R18: 0x00000005 R19: 0x00000009

R20: 0x00000000 R21: 0x00000000

R22: 0x00000000 R23: 0x00000000

R24: 0x00000000 R25: 0x00000000

R26: 0x00000000 R27: 0x00000000

R28: 0x00000000 R29: 0x00000000

R30: 0x00000000 R31: 0x00000000

The total number of instructions was 149 instructions, total number of cycles was 149 as the CPI is 1. Our cycle time was 20 ns, so the total run time is 2980 ns (this is the time when the halt instruction is executed).





