5-Stage Pipelined MIPS Simulator

Noah Patrick Kaplan

10-May-2022

An advanced project report submitted in partial fulfillment of the requirements for graduation with Honors in Computer Science.

Whitman College
2022

Certificate of Approval

This is to certify that the accompanying advanced project report by Noah Patrick Kaplan has been accepted in partial fulfillment of the requirements for graduation with Honors in Computer Science.

_____

John A. Stratton

Whitman College
May 11, 2022

Abstract

In this project, I develop a simulator for the MIPS32 Instruction Set Architecture (ISA). The designed processor uses a classical 5-stage pipeline, and logs results to a file. The programmer can use this file to inspect the processor's state at each clock cycle. While many other established simulation frameworks exist, this project was built from the ground up using common C++ libraries. This makes it portable and easy to understand or extend for any programmer with a background in C++.

The source code for this project can be found at `https://github.com/kaplannp/CCA-MIPS`

1 Introduction

Computer processor design involves complex decisions to balance many tradeoffs: power consumption with performance, cache latency with hit rate, single instruction latency vs throughput, etc.

Due to the inherent costs of designing and manufacturing a fully functional chip, the hardware industry has developed ways to research new designs and catch bugs earlier in the design process. For instance, Field Programmable Gate Arrays (FPGAs) can be used to prototype hardware systems before developing the expensive infrastructure necessary for large scale manufacturing [3]. Processor designs today also go through extensive simulation [1]. This is an invaluable tool for comparing performance, and is the prevalent method for researchers to experiment with new features. There are extensive libraries and simulation frameworks to aid the user in this process [1], but in my work, I developed a bare metal simulation framework from scratch in C++.

2 A Brief History Of MIPS

The MIPS (Microprocessor without Interlocked Pipelined Stages) ISA originated as a research project at Stanford lead by John Hennessy[4]. Stanford MIPS was radical in that it uses a pipelined architecture, but has no interlocks. This shifts the burden of decoding to the compiler, and allows instructions to run at higher speeds, but is often impractical because it precludes operand forwarding and speculative execution, which can lead to a performance hit.

The company MIPS improved upon the Stanford research project by adding interlocks, but maintained a simple instruction set, and is now a popular, energy efficient architecture for embedded systems. Throughout this paper, I shall use the term MIPS to refer to the corporate MIPS architecture[5].

3 Goals

The purpose of the project is to provide a simple, but extensible codebase for bare metal simulation of a pipelined MIPS32 architecture. This project is *not*

an attempt to replicate or improve upon research grade simulation frameworks. This code has little application as a research tool, but can be used to explain complicated processes, such as pipelining, without the added overhead and features found in other frameworks.

4 Implementation

The designed processor implements a subset of the MIPS I ISA used in an introductory architecture class at Gordon College, with a few extensions[2].

An operating system is beyond the scope of this project, and so a program loader is required to prepare an object file for execution. The code is written in C++ using the BOOST unit test framework and object-oriented software design. In this section I discuss some of the more important modules in the simulator.

4.1 Memory Module

In modern processors, the most costly operations tend to be memory accesses, so most simulators provide good support for modifying memory implementation with different cache levels, sizes, replacement policies, etc. The designed simulator implements an unrealistic, but simple, single clock cycle memory access, but it provides an interface for extension to more complicated memory systems, the `MemoryUnit`. The classes code to this interface so as to allow extension of the memory system without changing pipeline code.

I have designed two concrete types of memory, shown in their relationship to `MemoryUnit` in Figure 1. The `DRAM` class stores memory in an array of 32bit values, and memory accesses index directly into this array. The `VirtualMem` class obscures direct memory access behind a lookup table. It must be stressed that `VirtualMem` does not implement true virtual memory, as this requires operating system intervention. However, some form of virtual address translation is required to accommodate the 32bit address space of a MIPS32 program without allocating a 4GB array. By using a lookup table, `VirtualMem` services a wide address space with a much smaller block of allocated memory.

4.2 Pipeline Module

Figure 1 shows the relationship of the pipeline classes to the memory module, as well as their inheritance hierarchy. Every pipeline stage inherits from the abstract `PipelinePhase` class. Although the current implementation adheres to a classical 5-stage pipeline, I anticipate that future work may implement more complicated pipelines. Much of the pipeline module is therefore designed for extensibility to allow for variable length pipelines. The `PipelinePhase` superclass provides an abstract interface so higher level objects can iterate through collections of pipeline stages using polymorphism to invoke `updateCycle`, and

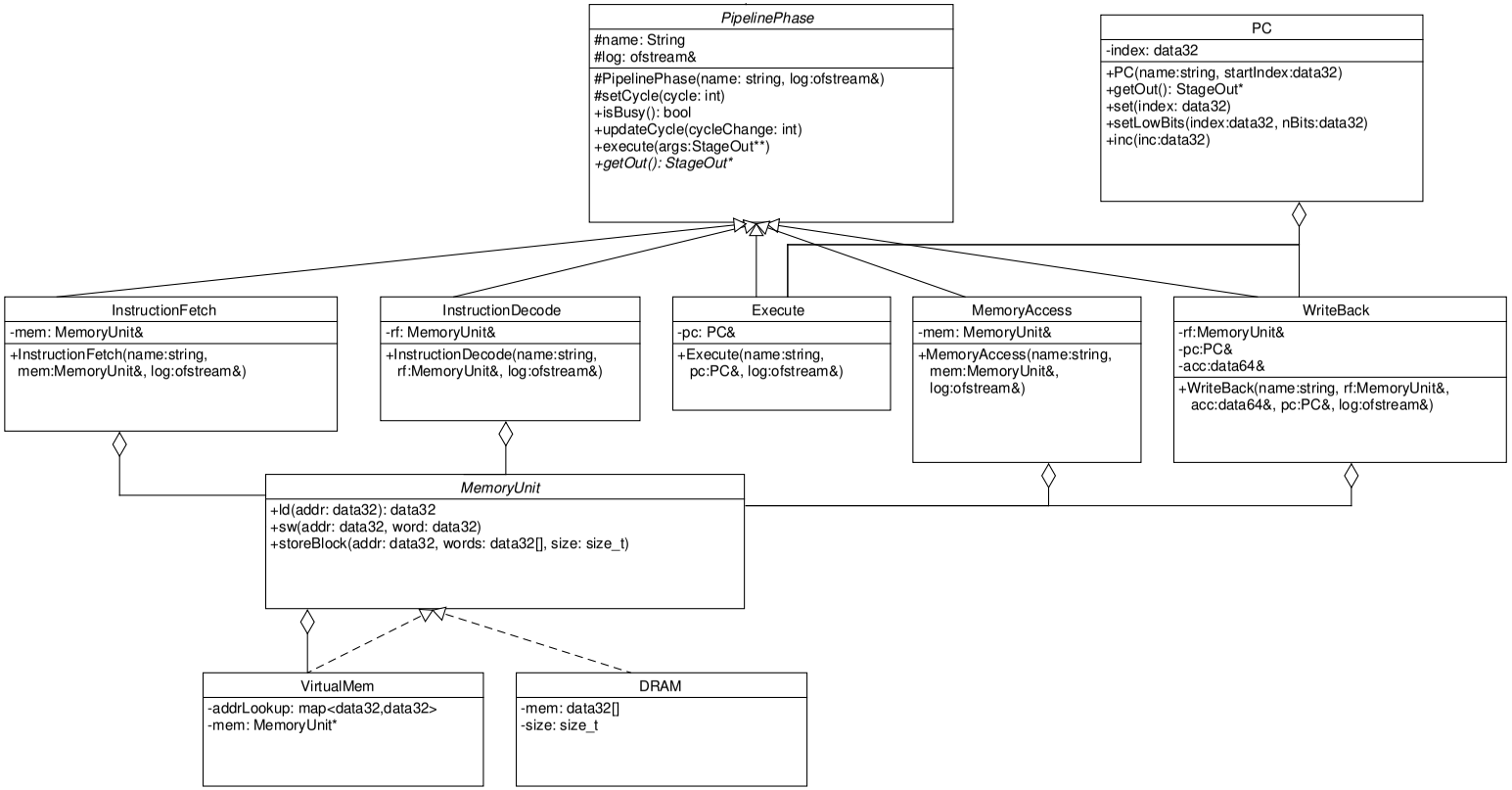

Figure 1: UML diagram of pipeline and memory modules.

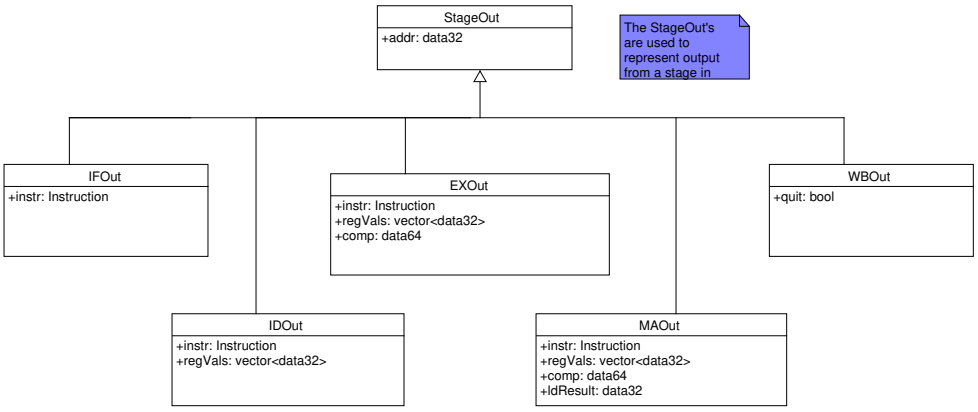

Figure 2: UML diagram of StageOut and subclasses

`getOut`. Each call to `getOut()` returns a `StageOut` object specific to the subclass of `PipelinePhase` as shown in figure 2. This allows for the natural data flow of each output to the next stage in the pipe.

Given that this is a 32bit system, it may be surprising that the output computation of the Execute unit is a 64bit value. This reflects the MIPS architecture for multiplies and divides. A 32bit multiply has the potential to produce a 64bit value, and a divide computation produces both a 32bit remainder, and a 32bit quotient. It is therefore useful to store the results of these operations in one 64bit register, the accumulator register, and access the 32 high or low bits with subsequent instructions.

The private variables associated with each pipeline stage reflect the buses that would connect the components of that stage to other components of the processor. For example, the only phases with access to main memory are `InstructionFetch` and `MemoryAccess`. Stages have access only to the components they need to perform their function, but this diagram would change somewhat if more of the ISA were implemented. For instance, `MemoryAccess` would likely be used as a computational unit for certain operations, such as a multiply and accumulate instruction (MADD).

4.3 Processor

The processor module incorporates classes from the pipeline and memory modules to produce a logical simulation flow. The core of this module is the `Processor5S`, shown in Figure 3. It has a 5 stage pipeline of `PipelinePhases` corresponding to our simple processor design, as well as a register file, main memory, accumulator and program counter (PC). It's interface is simple. The `updateCycle` method iterates through the pipe to simulate the passing of a

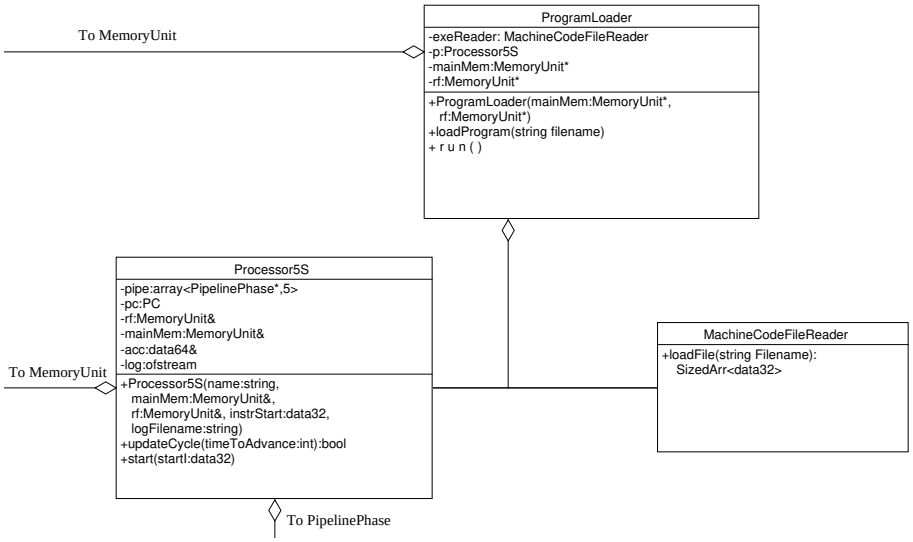

Figure 3: UML diagram of the processor module

given number of clock cycles. The `start` method begins a simulation at the given instruction address, and continues until a termination instruction is executed.

Since the simulator lacks an operating system, some class needs to load the code into memory, point the PC to the first address, and start execution. This is the purpose of the `MachineCodeFileReader` and the `ProgramLoader`. The `MachineCodeFileReader` takes as input a newline separated value file of 32bit hex strings, and returns an array with the contents of the file. The array is then loaded into memory by the `ProgramLoader`. The `ProgramLoader` adds a terminate instruction at the end of the instruction sequence, and loads the address of that instruction into the return address (RA) register, $31. This allows the program to exit when the main function returns.

4.4 Compilation

C programs are compiled to object code using the GCC MIPS cross compiler. It is easier to compile to object code rather than an executable because executables contain directives to the operating system that are not relevant for a bare metal processor. I use the binutils package to extract the .text section from the Executable and Linkable File (ELF), and parse these instructions to output a newline separated value file of 32bit hex instructions in the format expected by the program loader. This pipeline is wrapped in a simple python script to automate the generation of machine code from simple C programs.

5 Evaluation

Much of the software was written under the Test Driven Development (TDD) model. As a result, most every class has undergone rigorous testing, and an extensive test suite has been accumulated. Embedded in that suite are a few human coded assembly snippets that have been run on the processor with successful results.

In addition to the testing suite, the code has been run on a simple C program (an empty main function). The program terminated successfully and output a log of processor state information that matches expected behaviour.

6 Limitations

Because the simulator does not implement the complete MIPS I ISA, certain compiled programs will not run. Implementing further instructions would improve the versatility of the processor, but the implemented subset is sufficient to run basic C programs. If the processor encounters an unimplemented instruction, it will raise an exception, at which point the programmer may implement it themselves, or translate their instruction into a sequence of accepted instructions.

The simulator does not support byte addressable memory. This is inconvenient as many compiler generated branch instructions add offsets to the PC in multiples of 4. Such an offset will jump to 4 times the expected location because the PC will read the immediate as words instead of bytes. This could be remedied by refactoring the current code to calculate offsets in terms of bytes, but divide any load or store by 4 to get the proper address.

As in Stanford MIPS, the processor has no interlocks or forwarding[4]. As a result, it is the responsibility of the compiler to produce code without data or control hazards. As I have alluded, this strategy can lead to performance hits, and furthermore, the compiler used assumes a target commercial MIPS architecture, so interlocks are required to run more complex programs correctly.

As previously mentioned, memory access is unrealistic. For simplicity, a memory access is completed in a single clock cycle, but infrastructure, such as the `MemoryUnit` abstraction, is in place to extend to more complicated memory systems with different latencies.

7 Conclusion

I designed and implemented a bare metal MIPS simulator that can execute a large subset of the MIPS I ISA. The simulator passes rigorous test cases, and has successfully executed a simple C program on a parallel pipeline.

8 Acknowledgements

I thank my project supervisor, Professor John Stratton, for his invaluable feedback on this project. I'd also like to acknowledge Bucknell University for providing a free online MIPS disassembler.

References

[1] Ayaz Akram and Lina Sawalha. 2019. A survey of computer architecture simulation techniques and tools. *Ieee Access*, 7, 78120–78145.

[2] [n. d.] Cps311 - gordon college department of mathematics and computer science. (). `http://www.cs.gordon.edu/courses/cs311/handouts-2015/MIPS%5C%20ISA.pdf`.

[3] Michael Gschwind, Valentina Salapura, and Dietmar Maurer. 2001. Fpga prototyping of a risc processor core for embedded applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9, 2, 241–250.

[4] John Hennessy, Norman Jouppi, Steven Przybylski, Christopher Rowen, Thomas Gross, Forest Baskett, and John Gill. 1982. Mips: a microprocessor architecture. *ACM SIGMICRO Newsletter*, 13, 4, 17–22.

[5] [n. d.] Mips32 architecture. (). `https://www.mips.com/products/architectures/mips32-2/`.