Term Project Project Part 3

James Gaul

Andy Eslick

Introduction

During this semester, we created three processors that implement the RISC-V ISA using three different implementation strategies: Single Cycle, Software Scheduled Pipeline, and Hardware Scheduled Pipeline. For each processor, we sketched a high level design, implemented the processor, wrote tests, debugged, and did performance analysis after the processor was confirmed to be working. This report discusses the results, our interpretations, and explores potential for improvement.

Benchmarking

Single Cycle	**Instruction Count**	**Total Cycles to execute**	**CPI**	**Maximum Cycle Time**	**Total Execution Time**
Testbench	137	137	1.0	36.27 ns	4 968.99 ns
Grendel	2 129	2 129	1.0	36.27 ns	77 218.83 ns
Mergesort	549	549	1.0	36.27 ns	19 912.23 ns

Software Scheduled Pipeline	**Instruction Count**	**Total Cycles to execute**	**CPI**	**Maximum Cycle Time**	**Total Execution Time**
Testbench	216	239	1.11	17.72 ns	4 248.54 ns
Grendel	6 013	6275	1.04	17.72 ns	110 812.37 ns
Mergesort	1425	1515	1.06	17.72 ns	26 750.95 ns

Hardware Scheduled Pipeline	**Instruction Count**	**Total Cycles to execute**	**CPI**	**Maximum Cycle Time**	**Total Execution Time**
Testbench	137	261	1.91	19.42 ns	5 081.63 ns
Grendel	2 129	7073	3.32	19.42 ns	137 265.99 ns
Mergesort	549	1551	2.83	19.42 ns	30 172.27 ns

Performance Analysis

$\text{Instruction Count} \times \text{CPI} \times \text{Maximum Cycle Time}$ gives us our total execution time in nanoseconds. For each processor, we synthesized it to get our Maximum Cycle Time, and ran each test program (Testbench, Grendel, and Mergesort) using the provided toolflow to get the instruction count from RARS, the cycle count, and CPI.

Each processor has its own pros and cons, some of which are listed below.

	Single Cycle	**Software Scheduled**	**Hardware Scheduled**
Pros	• No hazards • Small instruction count	• No hazard detection overhead • Small cycle time • Instruction reordering	• Small Instruction Count • Register Forwarding • Small cycle time
Cons	• Large cycle time • Module Downtime[1]	• Large instruction count • No forwarding	• Hazard detection overhead

Because single cycle processors don't need to work around hazards, they do not need to insert NOPs, so each cycle an instruction gets executed. This results in having the smallest possible CPI of 1.0.

Our software scheduled pipeline can reorder instructions, but cannot forward.

Our hardware scheduled pipeline can forward, but cannot reorder instructions.

Software Optimization

One major source of downtime for the software-scheduled pipeline was jump and branch commands. It was often necessary to add nops inside the structure of a loop. This decreased throughput during the entirety of the looping function, even when the data hazards they prevented were only possible when the program entered or exited this loop. Restructuring the programs to decrease loops and branches would prevent dozens if not hundreds of unnecessary stalls.

Hardware Optimization

TODO: Performance Benefits of carry look-ahead adder.

For our Software Scheduled pipeline, the slowest stage was Execute, with the ALU Control taking around 3 ns, and the ALU's AddSub module taking around 13 ns to complete. ALU Control only needs the current instruction and Program Counter value to set the signals for the ALU. If we moved the ALU Control to the Instruction Decode stage, we would bring the Execute cycle time down by 3 ns for free, since ALU Control would work in parallel with the Register File and Control units since it doesn't rely on any of the registers or an immediate value. We would also be able to remove the Program Counter field from our ID/EX buffer, making it more space efficient.

1 For each cycle and instruction, not all modules are used. For example, addi does not use DMem or the DMem Sign Extender. In a pipelined processor, more modules can be used at a time, resulting in less potential downtime for a given module.

On our hardware scheduled pipeline, the Hazard Detection Unit contributed about 4 ns to each cycle, making up ~20% of the overall cycle time to *each cycle*. The time is split between decoding the instruction in IF, and actually sending the signals for avoiding hazards. Within the Hazard Detection Unit, 2 ns was spent comprehensively extracting fields (RS1, RS2, RD, Instruction Encoding, etc.) from each instruction stage in parallel. If we cut down on the number of fields extracted from each instruction, we would make the Hazard Detection Unit faster, and lower the maximum cycle time.

It Depends

There aren't many major programs that perform better since our hardware scheduled pipeline cannot forward. This analysis will assume that our processor has full forwarding implemented.

When data can be forwarded in a program, the hardware scheduled pipeline shines. Data forwarding takes place when instructions rely on results in close proximity. The following script highlights this.

```
main:
    addi a1, zero, 1    # Line 1
    addi a2, a1, 2      # Line 2
    addi a3, a2, 3      # Line 3
```

Assuming full forwarding, data can be forwarded from MEM to EX, and no stalling is necessary. However, in our software scheduled pipeline, nops must be inserted to allow each dependency to pass through the entire pipeline, increasing the number of cycles it takes to execute. Since Line 3 depends on Line 2, and Line 2 depends on line 1, we cannot optimize by reordering instructions either.

Challenges

We had an issue during part 1 of the project where our system diagram didn't match up with what we had in our VHDL implementation file. When it was time to add jumping and branching, the solution wasn't immediately obvious because we couldn't just look at the diagram and see what was going on, and took quite a bit of time to iron out. In response, we invested time into modeling all of our individual components in Microsoft Visio, and reconstructing the single cycle processor design so it was 1:1 with the VHDL implementation, and we reused the components in the software scheduled and hardware scheduled pipelines. The reason we didn't keep our schematic up to date was because Visio was a pain to use with our initial workflow. We had to spend time making sure arrows were lined up right, everything was on the correct layer, and we had to keep resizing components to make sure implementation details fit inside. In the future, when we decide on a modeling software, we should take some time to practice with it before we jump straight into designing our processor, so we have a good foundation from the start.

When we were designing our hardware scheduled pipeline, we had issues with branching and jumping, mostly with getting the Program Counter updated right. We had decided earlier that one person was going to do the Hazard Detection Unit, and the other was going to do the Forwarding Unit. We spent a few days working on our own thing, but we weren't really making any progress on anything. The next day, James, who was in charge of the Forwarding Unit, had thought up a simple solution for the Hazard Detection Unit, and it worked. I don’t think that I would have seen the solution since I was so deep in the debugging trenches, we needed a change in perspective. In the future, when we are in a position where we haven’t come up with a solution for a while, we should change what we’re working on so we can bring a fresh perspective to other road blocks in the project.

While James was working on the Forwarding Unit for our hardware scheduled pipeline, he was struggling with the forwarding conditions. For example, the Forwarding Unit wouldn’t forward when it was supposed to, and other times it would forward when it wasn’t supposed to. We both looked at the problem, but were unable to solve it ourselves, so James reached out to his older brother who works with computer processors as a job to get some help. Had we reached out to more experienced people earlier, we could have saved ourselves a lot of stress and deadline troubles. In the future, we should reach out for help sooner.

Demo