Term Project Project Part 3

James Gaul

Andy Eslick

Disclaimer: We will be submitting our processor with a non functioning forwarding unit. At some points, this report assumes our processor has forwarding.

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

Updated after demo, see Testbench and Software Scheduled Mergesort.

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	1 425	1 515	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	• Perfect CPI • No hazards • Small instruction count	• No hazard detection overhead • Small cycle time • Instruction reordering	• Register Forwarding • Small cycle time
Cons	• Large cycle time • Module Downtime[1]	• Large instruction count • No forwarding	• Hazard detection overhead

From a hardware perspective, our processors are very different. Single cycle processors don't need to work around hazards, they do not need to insert NOPs, so for every 1 cycle, 1 instruction gets executed. This results in having an ideal CPI of 1.0. However, since signals need to travel through the all used components in a single clock cycle, it has the largest critical path. Our software scheduled processor enjoys the benefits of pipelining without the overhead of hazard detection and forwarding. This results in the smallest cycle time of our processors. The overhead from the hazard detection unit and a forwarding unit would increase our cycle time, but forwarding would be worth it, reducing the number of number of cycles needed to complete a typical program.

Since we had freedom with when instructions were executed in our custom testbench, we were able to make various optimizations for the pipeline, such as reordering instructions. This let us bring the execution time of the test bench lower than the single cycle and hardware scheduled. The testbench doesn't have too many places where instructions depend on each other, so the hardware scheduled mostly keeps up with the single cycle, with it only really falling behind because of the jumps and branches in the program.

I don't think nops should have been counted as instructions for our software scheduled pipeline. Since we're missing forwarding, our current hardware scheduled pipeline is just a fancy automatic nop-inserter. By including nops as instructions, the software scheduled pipeline's CPI is artificially reduced, making it's calculated total execution time much less than the hardware scheduled pipeline. When we remove nops from the calculation, our software pipeline's instruction count becomes the same as hardware and single cycle. Using mergesort as an example, we get 549 instructions not counting nops, then recalculating the CPI with the same number of total cycles to execute gives us ~2.75, much closer to the CPI of our hardware scheduled pipeline.

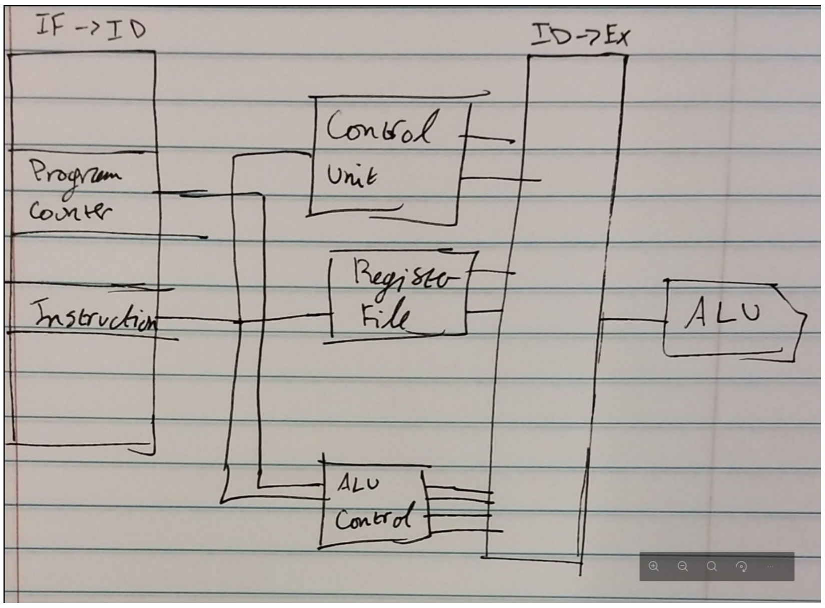
1 For each cycle and instruction, not all modules are used. For example, addi does not use DMem or the DMem Sign Extender. In a pipelined processor, more modules can be used at a time, resulting in less potential downtime for a given module.

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

In our single cycle processor, our AddSub module in the ALU takes about 13 ns to finish. The AddSub implementation uses ripple carry adders to compute the 32 bit result. Switching to a Carry Look-Ahead implementation would speed up the ALU, reducing our overall cycle time. Since all adders in the processor use the same AddSub component, this implementation change would also make the branching paths finish faster. A ripple adder has a time complexity of O(*n*), while a carry look-ahead adder has a time complexity of O(log(*n*)). While I cannot get an exact performance increase, it would be not be trivial because of the difference in time complexity. This change would also positively impact the software scheduled pipeline and hardware scheduled pipeline.



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. Here is a diagram with the change. This would also positively impact the hardware scheduled pipeline.

On our hardware scheduled pipeline, the Hazard Detection Unit contributed about 4 ns to each cycle, making up ~20% of the overall 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 and computing 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 are cases where our hardware scheduled pipeline performs worse than our single cycle design. In our hardware scheduled pipeline, when the program counter is updated, we need to flush the instruction in ID, wasting a cycle. Here is an example program.

```
main:
     j sub1

sub1:
     j sub2

sub2:
     j sub3

sub3:
     j end

end:
     wfi
```

Jumps take two cycles, and a single cycle is faster on our single cycle processor. This program executes faster on our single cycle design rather than our hardware scheduled pipeline.

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.

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