CprE 381 – Computer Organization and Assembly Level Programming

Project Part 3

John Brittain
Joey Brekelmans
Jaida Arms

5/6/2025

Over the course of this CPRE 3810 semester, we designed, implemented, tested, and evaluated the performance of three distinct MIPS processor architectures, including:

- Single-Cycle MIPS Processor
- Software-Scheduled MIPS Pipelined Processor
- Hardware-Scheduled MIPS Pipelined Processor

The implementation of each processor was done using VHDL (VHSIC Hardware Description Language), and using tools such as Questa ModelSim, we were able to compile and test our designs through simulation. After implementation and testing, we synthesized the processors to analyze their theoretical performance. The following report will be an in-depth analysis and comparison between our three distinct processor designs.

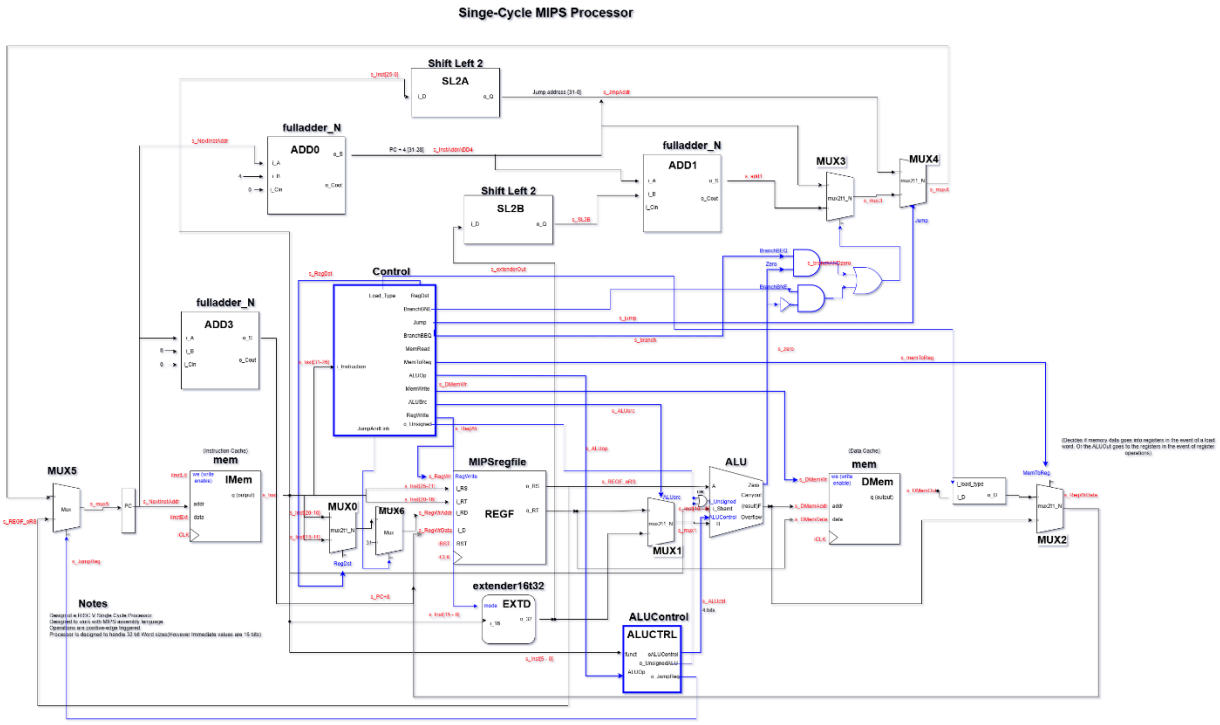


Figure 1High-level schematic of our Single-Cycle Processor Design

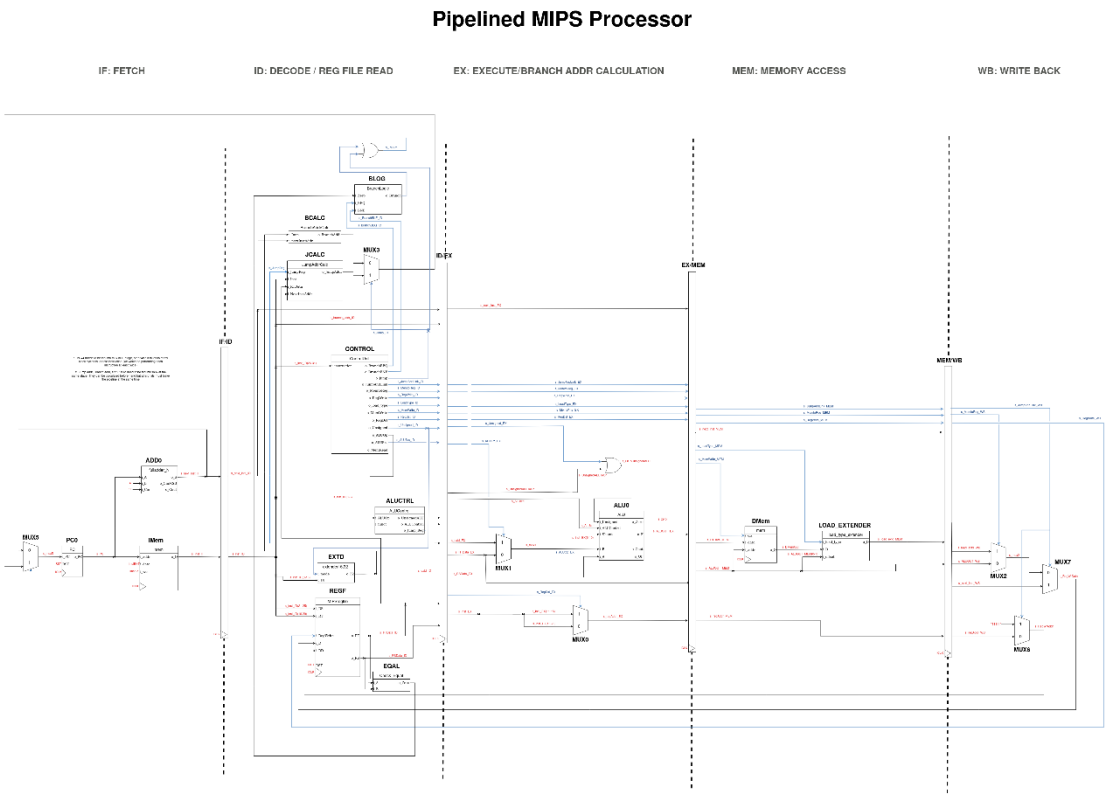


Figure 2High level schematic of our Software-Scheduled Pipeline Processor

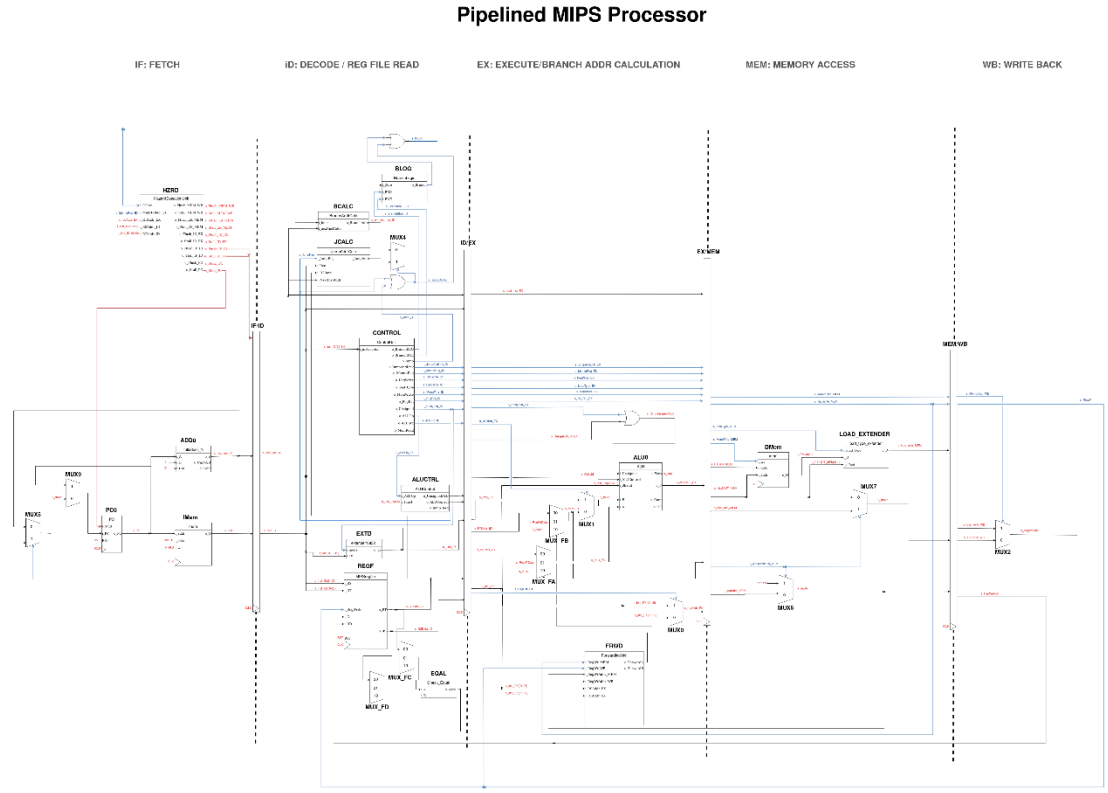


Figure 3High level schematic of our Hardware-Scheduled Pipeline Processor

Benchmarking

The following section contains information gathered from a series of benchmarking tests performed for each of our processor designs. The three Benchmarking tests are as follows;

Basetest–

A series of MIPS test which tested the functionality of all the supported individual instructions or operations. Such as add , or , nor , jal , lw etc.

Grendel –

A very large and hard to execute program which performs a Topological sort using an adjacency matrix.

Bubblesort –

Standard bubblesort algorithm implemented in MIPS which sorts through a predefined array of 11 word-sized elements

Single-Cycle Processor

	# instructions	*Clock Cycles*	*CPI*	*Cycle Time*	*Execution Time*
Basetest.s	40	40	1	48.112ns	1,924.48 ns
Grendel.s	2116	2116	1	48.112ns	101,757.79 ns
Bubblesort.s	524	524	1	48.112ns	25,220.49 ns

Software-Scheduled Pipelined Processor

	# instructions	*Clock Cycles*	*CPI*	*Cycle Time*	*Execution Time*
Basetest.s	53	59	1.11	20.380ns	1,202.42 ns
Grendel.s	5248	5509	1.05	20.380ns	112,297.42 ns
Bubblesort.s	1026	1117	1.09	20.380ns	22,763.46 ns

The number of instructions are increased because the pipeline is Software-Scheduled

Hardware-Scheduled Pipelined Processor

	# instructions	*Clock Cycles*	*CPI*	*Cycle Time*	*Execution Time*
Basetest.s	40	51	1.27	22.500ns	1,147.50 ns
Grendel.s	2116	2775	1.31	22.500ns	62,437.50 ns
Bubblesort.s	524	616	1.18	22.500ns	13,860.00 ns

Performance Analysis

This section will attempt to analyze the performance of each processor design using data from the test cases above.

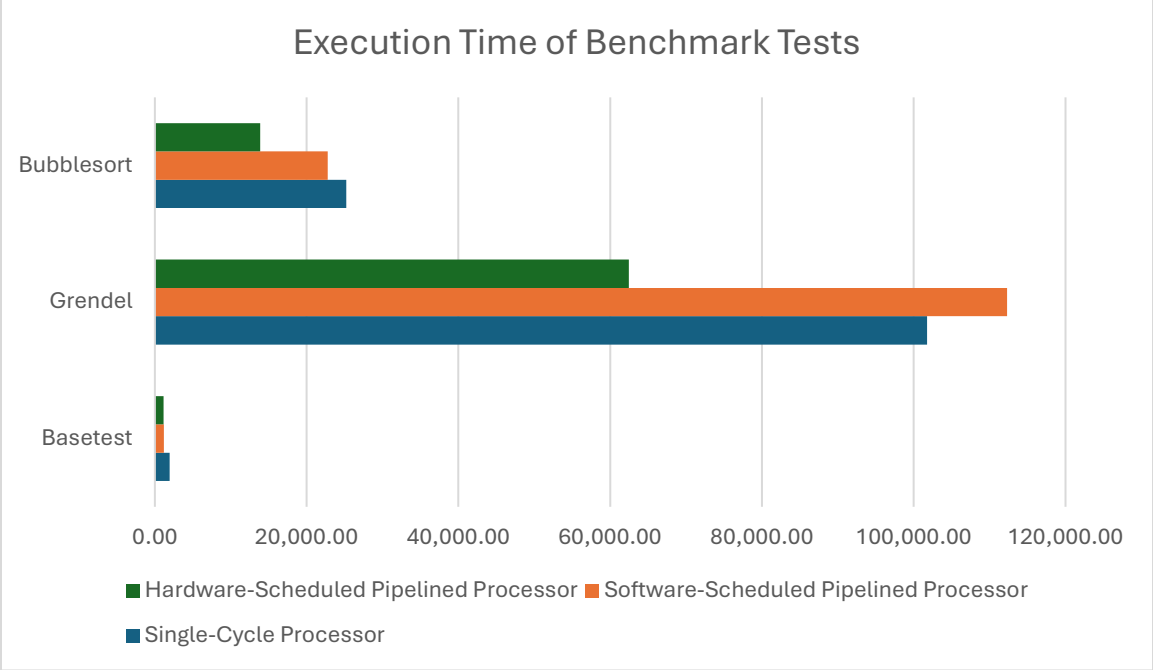


Figure 4 Column chart representing the different execution times each processor has for the benchmark tests

The column graph up above shows the execution time each processor has for the benchmark, at a quick glance it seems as though the Hardware-Schedule processor outperforms both the Single-Cycle and Software-Scheduled processor in all benchmarking cases by a wide margin.

Scalability

Scalability refers to how well a processor handles increasing workload or program complexity. An important factor when analyzing the performance of our processor designs is how the execution time increases relative to the size(instruction count) of an application. This reveals the scalability of the processor and how it handles increasingly complex work.

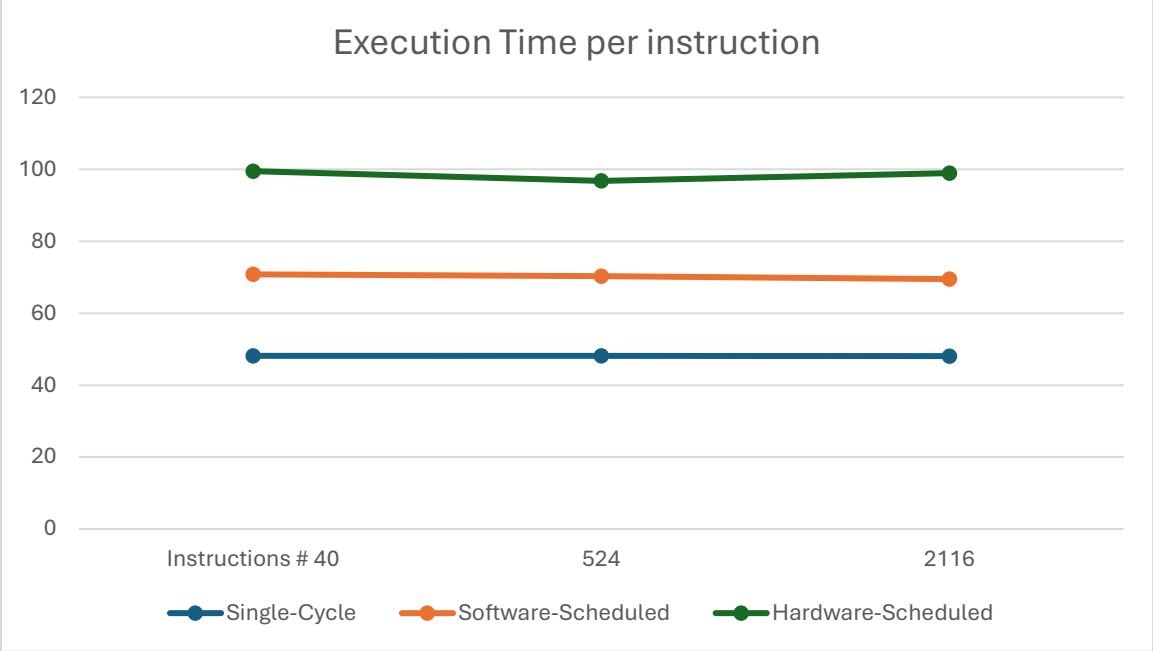


Figure 5 Line graph showing the execution time per instruction of each processor in the benchmarks

The Single-Cycle processor maintains a constant execution time per instruction, which is expected given that each instruction completes in exactly one clock cycle. While this suggests stable behavior, it also highlights a limitation: no performance gain can be achieved from instruction-level parallelism.

The Software-Scheduled pipeline shows a slight improvement in execution time per instruction as program size increases. This suggests some level of amortized efficiency gains, though this benefit depends heavily on how well the software is scheduled to minimize hazards and stalls.

The Hardware-Scheduled processor initially shows a drop in execution time per instruction and then stabilizes. This trend indicates strong scalability for larger programs, with consistent high performance, even as control flow complexity and data hazards increase. The ability to dynamically resolve hazards makes it a more reliable and scalable architecture overall.

Calculating Average Relative Speed of each processor

A useful metric in comparing CPU's performance with one-another is the average relative speeds of them. This is partially because the performance data acquired is based off of an idealized model, and is not exactly representative of real-world behavior. This way we can get a more holistic sense of not only what processor is faster, but also when given other factors can two CPU;s have equal performance but unequal efficiency.

To contrive the average relative speed to one another we used this formula:

$$Average\ Relative\ Speed = \frac{ExecutionTimeReference}{ExecutionTimeOther}$$

Benchmark	Single-Cycle Time (ns)	Software-Scheduled Time (ns)	Rel. Speed (SW)	Hardware-Scheduled Time (ns)	Rel. Speed (HW)
Basetest	1924.48	1202.42	**1.60**	1147.50	**1.68**
Grendel	101757.79	112297.42	**0.91**	62437.50	**1.63**

Benchmark	Single-Cycle Time (ns)	Software-Scheduled Time (ns)	Rel. Speed (SW)	Hardware-Scheduled Time (ns)	Rel. Speed (HW)
Bubblesort	25220.49	22763.46	**1.11**	13860.00	**1.82**

The Software-Scheduled pipelined processor outperforms the Single-Cycle design in the Basetest and Bubblesort benchmarks, but performs worse on Grendel. This discrepancy is due to poor scheduling within the Grendel benchmark: we inserted a significant number of NOP instructions to resolve data hazards manually. While this was intentional to demonstrate the cost of poor scheduling, it highlights a key limitation of software-scheduled pipelines—they are highly sensitive to how well the software is optimized. In real-world scenarios, well-optimized programs would likely show better performance across the board.

In contrast, the Hardware-Scheduled pipelined processor outperforms both other designs across all benchmarks. On average, it completes the benchmarks about 1.71× faster than the Single-Cycle processor. This is because hardware-based hazard detection and forwarding allow it to avoid the performance penalties of added NOPs and manual scheduling. Additionally, pipelining enables higher clock frequencies compared to a single-cycle design, further contributing to better performance.

Because it combines pipelining and automatic hazard handling, the Hardware-Scheduled design is not only faster but also more robust and consistent across varying workloads. It's less reliant on the programmer's ability to manually schedule instructions and generally provides superior scalability as programs become larger and more complex.

Analyzing CPU performance isn't so cut and dry when it comes to identifying clear winners. There are many factors at play when deciding whether or not a single design is the best choice. Performance isn't just about raw speed, it involves a balance between several critical aspects, each affecting the outcome depending on the context and specific requirements of the application.

- CPI *(Cycles Per Instruction)*
- Maximum frequency
- Hardware Complexity
- Scalability
- Power Consumption

Single-Cycle Processor Performance:

Average CPI: 1 (Every instruction executes in a single-cycle in a single-cycle design)
Cycle Time: 48.112ns (Critical path includes the entire processor)
Clock max frequency: 20.78Mhz
Speed: 1 (Baseline)
Scalability Rating: Does not change
Hardware Complexity: Simplest – Easiest to debug/implement
Usability: Flexible, doesn't require any software-scheduling and will correctly execute any program without the worry of control hazards

Software-Scheduled Processor Performance:

Average CPI: 1.08 (Including NOPS added in software scheduling)
Cycle Time: 20.380ns
Clock max frequency: 57.26Mhz
Speed: 1.20 (Heavily relies on how optimized the program is)

Scalability Rating: Good (If Optimized)
Hardware Complexity: Moderate – Most basic pipelined processor
Usability: Difficult (To make the processor functional and enjoy the benefits from a pipelined processor the program must be optimized, which makes it more difficult on the programmer)

Hardware-Scheduled Processor Performance:

Average CPI: 1.25
Cycle Time: 22.500ns
Clock max frequency: 51.35Mhz
Speed: 1.71
Scalability Rating: Excellent
Hardware Complexity: Complex – Includes extra hardware for hazards
Usability: Flexible, doesn't require any software-scheduling and will correctly execute any program without the worry of control hazards

Conclusion

Overall, each processor design offers distinct advantages and trade-offs to be considered. The Single-Cycle processor is a simple, predictable, and easy to implement design which is ideal when performance is not critical or where a high-complexity processor is not possible, manufacturing, smaller chips, embedded systems etc.

The Software-Scheduled Pipeline introduces significant speedups via pipelining but only if the code is optimized to avoid hazards. However, the hardware is smaller and less complex to implement than a Hardware-Scheduled pipeline. This makes this design suitable in cases where the processor runs specialized software programs and isn't required for general use.

The Hardware-Scheduled Pipeline is objectively the fastest in all situations and outperforms the others. It is the most complex to design/implement and requires the most amount of additional hardware, forwarding unit, Hazard detection, Branch predictor etc. but the payoff is superior efficiency, flexibility and performance for general use. That being said, in more specialized and controlled situations a Hardware-Scheduled pipeline might not be needed.

Optimization and improvements(Software)

From the performance analysis one of the most influential factors on how well the processors performed was how optimized the programs were, especially for the Software-Scheduled design.

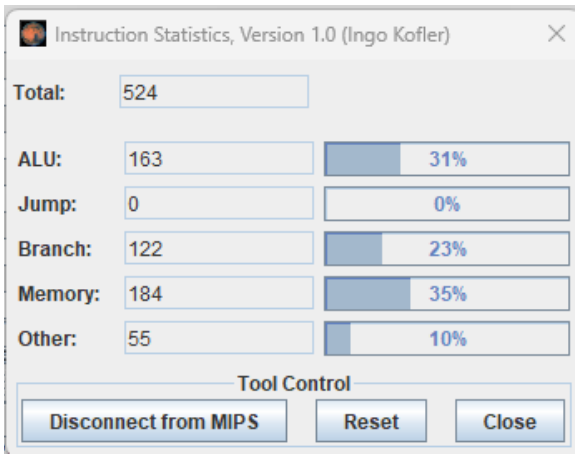
Base Test

The base test was intentionally designed to verify general processor functionality with minimal complexity. It executes a small number of arithmetic, memory, and control instructions in sequence and does not feature any tight loops, deep call stacks, or dependency hazards. As such, software optimization is neither necessary nor beneficial for this program. All designs performed similarly on this test, validating baseline correctness.

Bubblesort

The Bubblesort benchmark includes nested loops and many branches, making it particularly sensitive to control hazards and pipeline stalls. For the software-scheduled pipeline, these branches can cause multiple cycles of delay due to:

- The need for NOPs to handle data hazards between loads and uses.
- Delay slots or conservative branch handling.



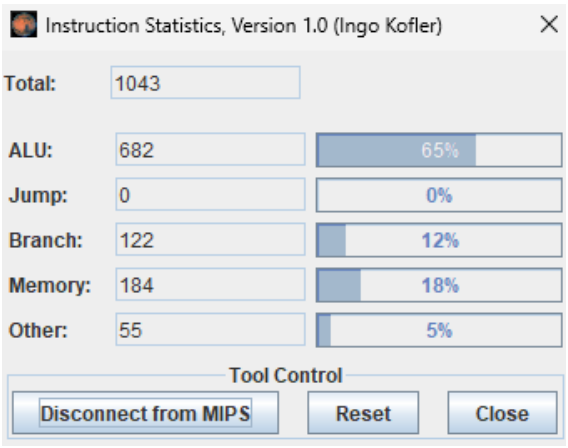


Figure 6 Mars Instruction Stats Bubble Sort Software Schedule

Grendel

Grendel is a more complex control-heavy benchmark that was deliberately programmed poorly including nested loops and many branches, data hazards and redundant/duplicate code. In this case the Software-Scheduled pipeline performed worse than the Single-Cycle design.

The most impactful optimization in this case would be instruction reordering, where instructions are reordered or removed to reduce the amount of stalls needed for data hazards. This is done by restructuring code to fit as many independent instructions in between dependent instructions as possible.

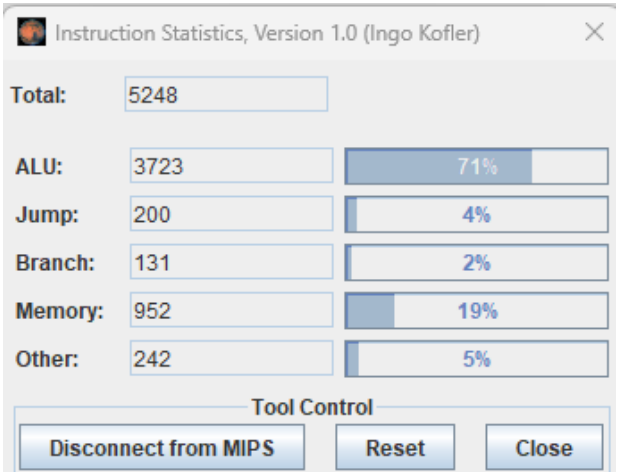


Figure 8Grendel Stats software scheduled

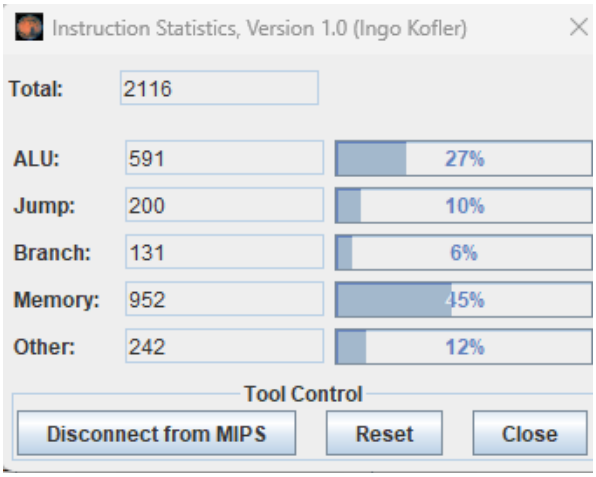


Figure 7 Grendel Stats

By comparing the instruction counts from the software-scheduled to non-software scheduled Grendel we can assume that the additional instructions are NOPS, and are a high priority target

when trying to optimize. 5248 – 2116 = 3132 NOP instructions added. Assuming you can remove %50 percent of these the performance benefit estimate is:

3698(total instructions) * 20.38ns(Cycle time) = 75,365.24ns new total execute time

112,297.42 ns (total execution time before) / 75,365.24 ns = ~1.5 x faster with optimization and brings it comparable to the Hardware-Scheduled pipelines performance.

Optimization and improvements(Hardware)

Our current hardware optimizations for each design as follows

Single-Cycle Processor:

No optimizations beyond baseline.

Software-Scheduled Pipeline:

Branch Comparator in ID Stage: Moves branch decision earlier in the pipeline, reducing branch resolution delay from EX to ID stage. This shortens control hazard penalties from three cycles to just one or two, depending on forwarding.

Positive-edge Register Reads, Negative-edge Writeback: Allows a register to be written and read in the same cycle without conflict, eliminating many common data hazards—especially load/use dependencies that would otherwise require NOP

Hardware-Scheduled Pipeline:

Branch Comparator in ID Stage: Moves branch decision earlier in the pipeline, reducing branch resolution delay from EX to ID stage. This shortens control hazard penalties from three cycles to just one or two, depending on forwarding.

Positive-edge Register Reads, Negative-edge Writeback: Allows a register to be written and read in the same cycle without conflict, eliminating many common data hazards—especially load/use dependencies that would otherwise require NOP

Forwarding Paths with Hazard Detection, Full data forwarding from EX/MEM, MEM/WB, EX/ID stages which prevents most data hazards except for load-use hazards. This enables near-optimal throughput throughout without NOPS.

Potential Optimization Improvements

Single-Cycle Processor:

Faster memory, the most limiting factor in our Single-Cycle design is the memory access time by implementing a faster memory the Critical path time would be shortened and the processors maximum clock frequency would be increased improving execution time.

Software-Scheduled Pipeline:

Faster ALU, the limiting factory in our Software-Scheduled Pipeline was the use of a ripple-carry adder which are not the best adder/subtractor design when it comes to performance. A faster adder/subtractor design could be a Carry-lookahead adder (CLA). During testing we attempted to implement this into our Software-Scheduled pipeline with various results:

Timing with CLA adder:

Data Required Path:

```
 Total (ns) Incr (ns) Type Element
 ========== ========== == ===== ===================================
 20.000 20.000 latch edge time
 23.394 3.394 R clock network delay
 23.374 -0.020 clock uncertainty
 23.392 0.018 uTsu EX_MEM:EX_MEM0|reg_N:reg_N_ALUOut|dffg:\gen_dffs:30:u_dffg|s_Q
 Data Arrival Time : 21.307
 Data Required Time : 23.392
 Slack : 2.085
```

Max Frequency: 55.8MHz (Still higher than 50Mhz constraint)

However the CLA adder reduced the overall frequency minimaly, I believe this is because the synthesis architecutre is mimicing FPGA's which excel at ripple carry addition, another style is in order.

Timing with 4-Bit CSA

Data Required Path:

```
 Total (ns) Incr (ns) Type Element
 ========== ========== == ===== ===================================
 20.000 20.000 latch edge time
 22.981 2.981 R clock network delay
 23.013 0.032 clock pessimism removed
 22.993 -0.020 clock uncertainty
 23.011 0.018 uTsu PC:PC0|RegPC:PC_reg_inst|oQ[5]
 Data Arrival Time : 20.087
 Data Required Time : 23.011
 Slack : 2.924
 ============================================================================
```

Max Frequency: 58.68 MHz

This implementation increases our max clock frequency from 57.26Mhz to 58.68Mhz (1.42 change)

Hardware-Scheduled Pipeline:

Split WriteBack/PC update stages, by adding a dedicated register stage just before PC writing to the PC we can shorten our critical path detailed by synthesis, this would lead to an increase in clock frequency however add complexity to our hazard avoidance logic. We would implement this by adding an extra stage which would Compute the next PC early then pass that onto the PC update stage.

Singe-Cycle VS Hardware VS Software Scheduling

To demonstrate architectural and functional differences between Single-Cycle vs Software and Hardware scheduling we designed two programs which will show off strengths and weaknesses.

Single-Cycle Favored Program:

```
addi $t0, $zero, 0
loop:
  addi $t1, $t1, 1
  addi $t0, $t0, 1
```

```
  bne  $t0, $t2, loop
```

The MIPS assembly code above contains a lot of RAW and Loop based data hazards which will require stalls and forwarding in the pipeline. On a single-cycle processor, this program performs comparably because of (1) the code is short and during the start up period of a pipelined processor. (2) The control hazards from the branch and data dependencies will require a stall on each iteration as well as data forwarding. The Hardware-Schedules processor will function just fine with this program but comparable to the Single-Cycle design.

Hardware Favored Program:

```
la   $t0  array #psuedo code – must be expanded upon for software
lw   $t1, 0($t0)
add  $t2, $t1, $s1
sw   $t2, 0($a1)
```

The MIPS assembly code above is designed to favor Hardware scheduling, it is a very short code but captures the fundamental difference between both designs. The problem here is that each sequential instruction is dependent on the last, and it's so short that it would be difficult to optimize with instruction reordering. The Software Scheduled pipeline would require NOPS to be inserted in between each instruction, including la which is a pseudo instruction. This pseudo instruction would need to be broken down into its base operations and injected with NOPS. The Software-Scheduled program would be as follows:

```
lui  $t0, 0x1001 #whatever the top of the array address is
nop
nop
nop
ori  $t0, 0x0000 #whatever the bottom of the array address is
nop
nop
nop
lw   $t1, 0($t0)
nop
nop
nop
add  $t2, $t1, $s1
nop
nop
nop
sw   $t2, 0($t1)
```

The additional nops injected add 12 more instructions for the Software-Scheduled processor, 12 more cycles doing nothing. For the Hardware-Scheduled pipelines all of the data hazards would be mitigated with the forwarding unit, and it would be able to execute the original code without any stalls.

Software Favored Program?

It is not possible to write a program that is Software Favored over Hardware given our implementation. Both stages use the same base stages and hardware, ALU, MEM etc. except the Hardware-Scheduled processor adds logic for detecting and resolving hazards during runtime, mitigating any stalls etc. In an ideal case, where the program has no hazards, jumps etc. then the Software-Scheduled processor would win based on the fact it operates at a higher

max speed but not by much. The difference in clock speeds is around 5Mhz the different execution times would be noticeable in longer programs, but that program would have to contain zero hazards and we can't think of many reasons a longer program wouldn't contain at least a couple. That being said, in most circumstances the Hardware-Scheduled processor will always outperform the Software-Scheduled design because of it's native hazard detection and avoidance.

Challenges

Our team was faced with a large and heavy workload for this project, and there were several major roadblocks along the way in each stage.

Single-Cycle Challenges

The single-cycle processor seems simple to us now but at this point designing and implementing processors was very new to us. Conceptually we had to learn how to build the basic components of a processor (ALU, Control unit, Register File) and then how to implement them on VHDL None of us particularly had much experience coding in VHDL at this point nevertheless building a processor. The ALU we designed had to be able to execute a wide variety of base instructions, each of those instructions required additional architecture to be implemented into the design. Distinctly we had issues with detecting overflow logic based on the type of operation and if it was unsigned or not. Another hurdle came when we implemented jumps and branches and how to properly update the PC register. Another when we implemented loading partial data from the processor, lb, lh, etc. especially considering our memory is word organized.

Software-Scheduled Challenges

The Software-Scheduled design was difficult enough, but we had our base processor working so all we needed to do was split it up into 5 stages? It seemed simple but it was not, as we begun to implement and tests this design more and more issues with how data propagated throughout the pipeline began to popup, particularly when it came to executing branch/jump logic. Another distinct challenge was debugging, because it is Software-Scheduled when a tests didn't work it was difficult to figure out if it was a hardware issue, or a software issue sometimes both. Tediously inserting NOPs and reordering programs was very labor intensive and mentally challenging.

Hardware-Scheduled Challenges

Designing and implementing a Hardware-Scheduled processor was definitely the hardest tasks in the project particularly when it came to implementing stalls and flushes, there was confusion of how to correctly stall/flush instructions would be lost in the pipeline and never executed on stalls, and flushes would repeat the same instruction. When it came to Control hazards a branch or jump could result in an infinite loop. After hours of staring at signals and debugging we were able to solve these issues, our first main misunderstanding was which side to flush the pipelined processors. Originally, we were resetting the registers themselves which ended up setting data on the opposite side to zero and not clearing the actual stage we were hoping. Another difficult challenge was how to stall the PC to prevent it from going to a new instruction, to solve this we developed quite a sophisticated set of muxes which properly choose the next instruction or reuse the old one depending.

Conclusion

This project pushed us far beyond what we initially thought we were capable of. We not only learned how to design increasingly complex CPU architectures in VHDL, but also how to think critically about instruction timing, hazards, and performance. We developed a deep

appreciation for the challenges involved in real-world processor design and validation. Despite the obstacles, our team grew tremendously in technical skill, patience, and collaboration. If we were to do it again, we would place a stronger emphasis on testbench automation and early modular testing, as this would have significantly reduced our debugging time and improved development velocity. This project was a true culmination of the knowledge gained throughout the course—and a rewarding challenge that we're proud to have overcome.