Connor Hand

Zachary Scurlock

**CprE 381 Project Part 3**

**Introduction**

This document outlines the benchmarking, performance analysis, and possible optimizations that could be done to the single-cycle, software-scheduled pipelined, and hardware-scheduled pipelined processors that we developed in previous projects. This document also discusses some challenges we faced while making the processors.

**Benchmarking**

|  | **Single-Cycle** | **SW Pipelined** | **HW Pipelined** |
| --- | --- | --- | --- |

|  | **# Inst** | **Cycles** | **CPI** | **Max Cycle Time** | **Exec. Time** | **# Inst** | **Cycles** | **CPI** | **Max Cycle Time** | **Exec. Time** | **# Inst** | **Cycles** | **CPI** | **Max Cycle Time** | **Exec. Time** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Synthetic Benchmark** | 30 | 30 | 1 | 45.8 ns | 1,374 ns | 31 | 31 | 1 | 20.7 ns | 641.7 ns | 30 | 66 | 2.2 | 21.5 ns | 1,419 ns |
| **Grendel** | 2116 | 2116 | 1 | 45.8 ns | 96,912.8 ns | 5161 | 5895 | 1.14 | 20.7 ns | 122,026 ns | 2116 | 5461 | 2.58 | 21.5 ns | 117,411.5 ns |
| **Bubblesort** | 263 | 263 | 1 | 45.8 ns | 12,045.4 ns | 694 | 753 | 1.09 | 20.7 ns | 23,343 ns | 263 | 746 | 2.85 | 21.5 ns | 16,039 ns |

**Performance Analysis**

By looking at the above benchmarks table, you can see that the single-cycle processor has the best execution times for the grendel and bubblesort programs. This is because the single-cycle processor has a CPI of one, meaning there is exactly one cycle for each instruction. Because the single-cycle processor only executes one instruction per cycle, it does not have to use nops or insert stalls to make the programs runnable. Because the single-cycle processor does not have to use nops or stalls, it is faster than the other two processors in the case of these two programs.

Looking again at the benchmark table, you can see that the SW pipelined processor has the fastest time for the synthetic benchmark. This is because the synthetic benchmark program for the SW pipelined processor was specifically designed for it. It has no nops and because the SW pipelined processor has the smallest max cycle time, it is the fastest processor for this program. The SW pipelined processor has the smallest max cycle time because it is a single-cycle processor split into five stages (which allows for a smaller cycle time), and it has fewer functional units than the HW pipelined processor, giving it the smallest max cycle time.

Looking at the HW pipelined benchmarking results, you can see that it does not have the best execution time for any of the programs. This can be explained in the case of the synthetic benchmark because the synthetic benchmark for the SW pipelined is designed in such a way that it does not require nops, while it does require a lot more stalls in the HW pipelined version (using the proj1 base test). Looking at grendel and bubblesort, there are more stalls than the number of instructions executed, meaning there is at least one stalled cycle per instruction, which greatly increases the execution time, making the single-cycle processor faster for both programs.

We only had to use two equations for our benchmarking results because the other information was given when running the programs on our processors. We had to calculate CPI by doing CPI = # Cycles / # Inst. We also had to calculate the execution time using the following equation: Exec. Time = # Cycles \* Max Cycle Time. We can make our discussion With the information from running the programs on our processors and the information we found through these formulas.

Overall, the execution time of a program depends on the type of and order of instructions in the program (necessity to insert stalls/nops). Some programs will run better on the single-cycle, some will run better on the SW pipelined, and some will run better on the HW pipelined. In the real world, most programs will have many more instructions and will typically be best suited for a HW pipelined style processor due to the number of instructions and more sophisticated methods than we have used.

**Software Optimization**

One software optimization that could be made to improve the performance of the software-scheduled pipeline would be to rearrange instructions to reduce data hazards and minimize the number of nops found within the program. By doing this overall CPI could be brought closer to 1. Assuming that CPI is equal to 1, we estimate that there would be a 38.5% decrease in execution time. We concluded this by looking at the number of instructions in grendel for the software-pipelined processor. We also compared its instruction count to the instruction count of the single-cycle processor version of grendel. From this, we concluded that we can estimate the number of nops by rearranging instructions to reduce data hazards and reduce the total number of nops.

**Hardware Optimization**

To improve the hardware design of our single-cycle processor, we would have to improve one of the functional units themselves. Keeping the same design and rearranging our parts to improve it is impossible. If we were to improve the access time of the data memory by five ns, we would have a max cycle time of 40.8 ns. This would make our single-cycle processor 10.9% faster.

To improve the hardware design of our SW-scheduled pipelined processor, we could detect if the instruction is a load or store instruction, then skip the data memory stage if it is not. Doing this would require another functional unit to determine if it would be safe to skip the memory stage (the writeback stage isn’t already busy) and if it is not a load or store instruction. You would also need to place a MUX at the end of the EX stage that loads the data into the EX/DM register or the DM/WB register. Let's use the grendel program benchmark numbers for the estimate calculation. If 5% of a program contained instructions that could skip the memory stage, then 5% of the instructions would only need four cycles while the rest need 5. With this, we can eliminate about 106 cycles needed to complete the program, resulting in the processor running about 2% faster.

To improve the hardware design of our HW-scheduled pipelined processor, we could add data forwarding from the execute and memory stages. Doing this would decrease the number of stalls it has to input by 40% (⅕ stages had forwarding, now ⅗ do, ⅖ = 40%). Looking at grendel, with the number of instructions, there should be 2120 cycles, meaning 3341 of the cycles are stalls. Decreasing this number by 40% you get 2004 cycles. Adding this back to 2120, we have 4124 cycles total, giving us an execution time of 88,666 ns. This optimization made our HW-scheduled pipelined processor 24% faster.

**It Depends**

To make a program that performs better on the single-cycle processor than the HW-scheduled pipelined processor, use no data memory access instructions (load and store) and make sure there are a lot of data dependencies. This works because the single-cycle processor does not have to wait for a cycle in the memory stage, and data dependencies do not make the single-cycle processor wait any longer (like they do in the HW-scheduled pipelined processor).

EX:

add $1, $2, $3

add $4, $1, $2

add $5, $2, $4 # No load or store instructions

To make a program that performs better on the HW-scheduled pipelined processor than the SW-scheduled pipelined processor, ensure there are many immediate data dependencies. The HW-scheduled pipelined processor can forward data, while the SW-scheduled pipelined processor cannot. This program would run way faster on the HW.

EX:

add $1, $2, $3

add $4, $1, $2

add $5, $2, $4 # Every instruction has immediate data dependencies

**Challenges**

Throughout the course of this project, we faced many challenges, including debugging, an issue involving the instruction memory, and an issue related to stalls on the hardware scheduled pipelined processor.

While debugging, we spent many hours trying to find the bugs preventing the three versions of the processor from functioning. Some of these bugs were seemingly minor things that turned out to be catastrophic, such as assigning the wrong signal to an output. Others were slightly larger issues regarding our logic.

During the creation of the single-cycled processor, we were unaware that the instruction memory needed to start at the memory address 0x00400000, so our IM was starting at 0x00000000, therefore causing our tests to fail. This issue was fixed by adding a multiplexor at the beginning of the design, with one of the lines being hard-wired with the proper memory address.

One of the final notable challenges we faced came about when testing our design for the hardware-scheduled pipelined processor. All of the tests were initially failing during our testing for this stage. After some investigation, we realized the same instructions were being fetched multiple times. This was because we tried inserting stalls by setting the PC write enable to 0, which didn’t work. This problem was fixed by adding a multiplexor to the instruction output filling the pipeline with zeros when a stall was executed.