CprE 381 – Computer Organization and Assembly Level Programming

Comparative Analysis of Processor Designs & Implementations

1. **Introduction**

Our project consists of three major designs of a MIPS based processor. The first is a single cycle processor which executes a single instruction per cycle. The second is a multi-cycle processor which is like the single cycle processor in many ways, but is divided into five stages with register files and relies on software for hazard avoidance. The third design is also a multicycle processor but contains hardware logic that can avoid data and control hazards.

2. **Benchmarking.** Now we are going to compare the performance of our three processor designs in terms of execution time. Below is a table for each of our processor designs. The rows correspond to our synthetic benchmark (i.e., the one with all instructions), grendel (provided with the testing framework), and Bubble Sort. The columns should be the number of instructions (counted using MARS), total cycles to execute (counted using our Modelsim simulations), CPI (using the previous two columns to calculate), maximum cycle time (from our synthesis results), and total execution time (using the appropriate previous columns). Note that the applications used to benchmark the single-cycle and hardware-scheduled pipeline applications are identical and thus the same number of instructions, while the software-scheduled pipeline programs should be modified to work on the software-scheduled processor and thus should have more instructions. We counted software-inserted NOPS as instructions.

Single-Cycle	**Number of Instructions**	**Total Cycles to Execute**	**CPI**	**Maximum Cycle Time**	**Total Execution Time**
Synthetic Benchmark	52 Instructions	52	1.0	25.07 MHz	2.0742µs
Grendel	2116 Instructions	Failed on Cycle 665	1.0*	25.07 MHz	84.404µs
Bubblesort	52 Instructions	52	1.0	25.07 MHz	2.0742µs

*Grendel failed, but CPI should be 1.0 because it is a single cycle processor

Multi-Cycle Software	Number of Instructions	Total Cycles to Execute	CPI	Maximum Cycle Time	Total Execution Time
Synthetic Benchmark	125 Instructions	145	1.16	59.86 MHz	2.4223µs
Grendel	5455 Instructions	6487	1.19	59.86 MHz	108.44µs
Bubblesort	759 Instructions	959	1.26	59.86 MHz	15.9763µs

*Estimated CPI based on: total Cycles / total instruction

Multi-Cycle Hardware	Number of Instructions	Total Cycles to Execute	CPI	Maximum Cycle Time	Total Execution Time
Synthetic Benchmark	52 Instructions	125	2.4	54.3 MHz	2.2983 µs
Grendel	2116 Instructions	5363	2.53	54.3 MHz	98.591 µs
Bubblesort	324 Instructions	785	2.42	54.3 MHz	14.4398 µs

Total Execution time calculated by: ET = # of Instructions * CPI * (1/Maximum Cycle Time)

3. **Performance Analysis.** Analysis of the performance of the three applications on the three processors. This section explains why the performance was better on specific processors compared to others and why some applications may have had a smaller difference in performance between processors versus other applications.

   The hardware and software processors are inherently better than the single cycle processors because they are multi cycle processors. With Multicycle processors, we are able to optimize certain instructions more than others. For example, a normal add instruction would take all five cycles to execute completely, but with the software and hardware multicycle processors, it can make certain optimizations like making branch instructions only take 3 cycles (writing back in the Execute stage) instead of the full 5.

   Additionally, hardware processor was generally better than the software processor because it was able to handle hazards by using forwarding rather than having to simply wait for instruction signals to propogate through the entire processor. Because with the software processor we had to simply add in nops in order to handle hazards, that inreased the total number of instructions which negatively impacts the total execution time, ET = CPI * Total Cycles. So, by eliminating the need for nops in the hardware multicycle processor, you reduce the Total Cycles required, thereby reducing total execution time.

   As for why certain files didn't have as big of a performance difference, that is because those files use different types of instructions. As I mentioned above, in the multicycle processor, our branch instructions and jump instructions write back to PC in the execute stage. This means that whenever a branch instruction or jump instruction is being executed on the multicycle designs, it will be faster than the single cycle designs

because it only has to take up a fraction of the "full cycle" (all 1 cycle for all of the 5 stages). So, files with more branch and or jump instructions would generally perform better on the multicycle processors than on the single cyclep processors. Additionally, for the hardware processor, certain instructions required more nops than others. This means that if a file had a lot more nops because the instructions being used within it required more nops than, say, a branch instruction, it would perform worse. So, by removing nops entirely, this means that on files that originally had a large number of nops we are saving a lot of time.

4. **Software Optimization.** Brief discussion describing one software optimization (i.e., assembly-level software refactoring) that would improve the performance of the software on the software-scheduled pipeline relative to the others. An estimate on the performance benefit this change would have given our specific benchmarks is also given.

   For our specific processor, one software optimization we could make is to forward some signals so that it does not require as many nops for all the instructions. We know this would work because that is exactly what we do for our branch and jump instructions, and those are the fastest executing instructions for our software multicyle processor. For those instructions, we forward the branch and jump result signals back to the muxes that feed the next instruction to the PC. If we were to do this for more instructions than just the branch and jump instructions, we would have a similar result. The exact performanc benefit is somewhat difficult to determine because it would depend on what files we are executing and what instructions they have the most of. But, for the instructions we would forward signals for, it would improve the execution time of those instructions roughly by a factor of 20% * # of stages being skipped

5. H**ardware Optimization.** Brief discussion describing one hardware optimization for each design that would improve the performance of the software on the hardware-scheduled pipeline relative to the others. An estimate on the performance benefit this change would have given our specific benchmarks is also given.

6. **It Depends.** Identify or write a program that performs better on a single-cycle processor versus a hardware-scheduled pipeline and another one that performs better on the hardware-scheduled pipeline versus the software-scheduled pipeline.

   An example of a program that would be better on a single cycle processor than on a hardware multicycle processor would be one that contains a large number of if statements. This is because, if a program has lots of conditional branches with only a small number of other instructions between them, the latency associated with deciding which way each branch instruction will go will make it impossible to keep the pipeline filled with useful instruction, which is the main benefit of a multicyle processor.

7. **Challenges.** Brief description of the three most critical challenges our project faced during its development and how they could be avoided in the future.

   The biggest challenge we faced was trying to understand how fetching and writing back instructions works. We were stuck on that topic for a large amount of time and that really slowed us down because we couldn't start wiring everything together for our top level vhdl files until we understood how to get instructions *into* the processors. We eventually figured it out by just going through step by step what

each instruction did and where it went in the processors. We referenced the green sheet a lot to make sure we understood the purpose of each bit of each instruction and we just traced those bits through the processor until it got to the point
Where it needed to be written back to the PC.

Another big issue we faced was not understanding how to debug our processors. We hadn't realized that we were able to pull up the waveform files that were generated when we ran the mips tests within the toolflow, so we really struggled debugging until finally we asked our TAs and they helped walk us through the debugging process. That was very beneficial for us because it allowed us to trace specific signals through the processors and realize things like "Oh! This mux is taking the normal PC + 4 address instead of the jump address! That means there's an issue with our control unit signal".

Finally, another issue we experienced was figuring out how to make the hardware and software pipelined processors not fail on all the tests. First of all, we didn't understand that for the software processor we were supposed to literally insert nops into the mips code. We spend a long time attempting to forward different signals around or pause the processor, etc just to get it to work. Eventually we tried inserting a nop into the actual mips code and it solved our issue.

And then when we worked on the hardware processor, we struggled to get the hazard detection unit to work. The biggest help for figuring out the hazard detection was going into questasim with the hazard detection unit and experimenting with different forced values to see what it outputs. By doing that we were able to figure out why certain signals were not being set to the correct values and when those issues were occurring. That said, even though we got it to work, we both suspected that it was very inefficient.