# CprE 381 – Computer Organization and Assembly Level Programming

## Project Part 3

Team Members: Soma Szabo, Conner Spainhower, Kevin Nguyen

Project Teams Group #: 3_6

1. **Introduction**.

Our term project involved creating a single cycle and a pipelined MIPS processor. This required a thorough planning and design phase where we determined what each component should do and how to implement it using VHDL. Once the main modules such as the ALU, control unit, instructions and data memories were complete, we thoroughly tested each one and put them together to create the final processor. The implementations of the single cycle and software pipelined processors were tested using our own MIPS programs along with the given MIPS code to ensure the processors can successfully complete each one. Afterwards, we performed synthesis on both designs and analyzed the execution time of each program on the processors, comparing which one was faster.

2. **Benchmarking**. Now we are going to compare the performance of your three processor designs in terms of execution time. Please generate a table for each of your final single-cycle and software-scheduled pipeline designs. The rows should correspond to your synthetic benchmark (i.e., the benchmark in which you used all instructions), grendel (provided with the testing framework), and Bubblesort. The columns should be # instructions (count using MARS), total cycles to execute (count using your Modelsim simulations), CPI (using the previous two columns to calculate), maximum cycle time (from your synthesis results), and total execution time (using the appropriate previous columns). Note that the applications used to benchmark the software-scheduled pipeline programs should be modified to work on the software-scheduled processor and thus should have more instructions. **Count software-inserted NOPS as instructions**.

| Single Cycle Design | # of instructions | Total cycles to execute | CPI | Min cycle time | Total execution time |
|---|---|---|---|---|---|
| Synthetic benchmark | 63 | 63 | 1 | 41.771ns | 2.632ns |

| | | | | | |
|---|---|---|---|---|---|
| grendel | 2116 | 2116 | 1 | 41.771ns | 88.387us |
| Bubble Sort | 1626 | 1626 | 1 | 41.771ns | 67.920us |

Frequency: 23.94MHz
Cycle time: 1/23.94M = 41.771ns

| Software Pipeline Design | # of instructions | Total cycles to execute | CPI | Min cycle time | Total execution time |
|---|---|---|---|---|---|
| Synthetic benchmark | 109 | 123 | 1.128 | 19.616ns | 2.413us |
| grendel | 4053 | 4387 | 1.082 | 19.616ns | 86.055us |
| Bubble Sort | 2897 | 3235 | 1.117 | 19.616ns | 63.458us |

Frequency: 50.93MHz.
Cycle time: 1/50.93M = 19.616ns

The total cycles to execute factors in the time required to fill and empty the pipeline, meaning that the halt instruction is fetched 4 cycles earlier than the last instruction to execute. For instance, in grendel (testbench) this would mean halt is fetched in cycle 4383 and reaches the writeback stage in cycle 4387 so the processor officially stops executing in this cycle.
The CPI was calculated by dividing the total cycles to execute by the number of instructions.
The total execution time was calculated by multiplying the minimum cycle time by the total cycles to execute.

3. **Performance Analysis**. Analyze the performance of the three applications on the two processors. Explain in your own words why the performance was better on one processor versus another or why some applications may have had a smaller difference in performance between processors versus other applications. **Hypothesize and explain about where a hardware-scheduled pipeline would fit (you should at least be able to bound an estimate)**. This section should reference the above performance table and describe how it was generated including any formulas you used.

Based on the total execution time for the three applications, the software scheduled pipeline consistently performed better than the single cycle processor. The speedup for each application is shown below.
Synthetic benchmark: 2.632/2.413 = 1.0908 => 9.08% faster for pipeline
Grendel: 88.387/86.055 = 1.0271 => 2.71% faster for pipeline

Bubble Sort: 67.92/63.458 = 1.0703 => 7.03% faster for pipeline
This illustrates how optimizing both the software and hardware for a pipelined processor can increase performance. For each of the MIPS assembly codes we ensured that NOPs are only inserted when necessary and sometimes changed the order of execution to ensure data and control hazards are minimized.

The difference in speedup is likely due to the type of instructions and dependencies in each MIPS program. The pipelined processor requires a maximum of 2 NOPs after any of the supported instructions. Therefore, it will only be slower than the single cycle processor when 2 NOPs are required after an instruction. This is because the cycle time of the pipelined processor is less than half of the cycle time of the single cycle processor. Thus, 2 instructions can be completed in the pipeline in less time than 1 instruction completes on the single cycle processor. The only way for the pipelined processor to perform worse is if several instructions require 2 NOPs after them, meaning there are many data or control hazards one after another. Overall, to make the pipelined processor perform worse, there must be on average 2.13 or more NOPs per instruction.

As previously mentioned, the speedup greatly depends on the order of execution for each instruction, the dependencies, and the number of branches/jumps since those always require 1 NOP. Another expensive instruction is the jump and link since that always requires 2 NOPs after it. Grendel had lots of branches/jumps and load/store operations that required at least 1 NOP which is why the speedup was the lowest for that benchmark. Bubble sort also had several branches/jumps and maybe a little more dependencies than the synthetic benchmark which likely had the lowest NOP to actual instruction ratio. Thus, the execution time matches our expectations.

A hardware scheduled pipeline would likely improve performance because the critical path in the decode stage would probably not get longer; however, other paths like the execute or memory stage will have an increased latency due to the forwarding logic. This may create a new critical path and increase cycle time. Even if the cycle time increases by ~1ns the frequency would still be twice as fast as the single cycle and fewer instructions would require stalls. The maximum number of stalls an instruction would need is one (if load/store hazards are forwarded). Since most applications do not require stalls that often, I think the hardware scheduled pipeline with forwarding would perform about 1.5 to 2 times better assuming the cycle time is around 20ns and less than 50% of instructions require a stall after them.

4. **Software Optimization**. Identify and describe one software optimization (i.e., assembly-level software refactoring) that would improve the performance of software on the software-scheduled pipeline relative to the others. Provide an estimate of the performance benefit this

change could have given your specific benchmarks.

One change could be reordering the instruction to reduce the number of NOPs required. This may be done by checking for dependencies and figuring out an optimal instruction order that minimizes hazards and therefore NOPs. This would include executing branches and jumps 1 instruction earlier in the code if possible since they always require 1 NOP after them. This way, that cycle could be used by a useful instruction instead of a NOP.
Another option may be changing the instructions if possible to ones that are not pseudo instructions such as a subi instruction to an addi with a negative immediate.

Below is an example of jump reordering.

```
13  Main:
14    #la    $a0, array #array base address
15    lui $1, 0x00001001
16
17    nop
18    nop
19
20    ori $4, $1, 0x00000000
21
22    jal BubbleSort
23    addi  $a1, $zero, 10  #array size
24    #nop
25
26    #exit program
27    halt  #halt for testingframework
28
29    nop
30    nop
31
32    li $v0, 10
33    syscall
34
35  BubbleSort: #swap (a[j], a[j+1])
36    nop
37
38    addi $t0, $t0, 0 #int i
39    addi $t1, $t1, 0 #int j
40    addi $t2, $a1, 0 #size of array
41    addi $t3, $t3, 0 #comparison variable
42    addi $s1, $s1, 0 #array address
43    addi $s0, $a0, 0 #load base address for the array passed from function
```

Reorder the jal and addi so instead of the NOP after jal there is a useful addi which does not have any dependencies with the code.

These changes would increase performance by the percentage of jumps/branches multiplied by the cycle time (19.616ns) assuming 1 NOP is removed from each. Furthermore any other restructuring of the MIPS program could further reduce cycle time as NOPs are reduced. The synthetic benchmark could have about 8 NOPs removed, decreasing execution time by about 5% to 7%. Grendel had 10 beq instructions and about 30 jumps when only looking at the code, assuming at least half of those NOPs could be removed, it may decrease execution time by about 2% to 4%. Finally, bubble sort had 4 jumps and 4 branches and considering there are several loops this may also reduce execution time by about 5% or 6%.

5. **Hardware Optimization**. Identify and describe at least one different hardware optimization for each design that would improve its performance. The optimization cannot be turning it into one of the other designs (e.g., adding hazard detection and forwarding logic to the software scheduled pipeline). Certain optimizations can be beneficial to more than one design – chose one design on which you would apply the optimization. Briefly list the specific set of changes you would have to make to your design to accommodate each optimization (a figure would be helpful). Provide an estimate of the performance benefit each optimization could have given your specific benchmarks.

For the single cycle design upgrading to a faster ALU with the shifter integrated in it like in the pipelined processor would decrease the cycle time. This is because it is in the critical path and the shifter outside the ALU adds a significant amount of latency. This may improve cycle time by a couple of nanoseconds, resulting in a 3% to 6% reduction in cycle time.

For the software scheduled pipeline the critical path is when a branch instruction occurs because 2 register values must be read, compared, and the jump address needs to be sent back to the PC. Speeding up this process would reduce the cycle time and allow for a faster frequency since that is the slowest pipeline stage. This may be done by combining some muxes or making the register file faster and would be very marginal like 1% to 2% faster since it would be difficult to further optimize this path. However, if we manage to speed that up a lot, another pipeline stage may become the slowest and have to improve that to further decrease the cycle time.

6. **It Depends**. Given the above discussion, you should now understand the interaction between the programs and your hardware designs in terms of performance. Identify or write a program that performs better on a single-cycle processor versus a software-scheduled pipeline (when reasonably optimized for the design). Describe your approach to building this program. If it is impossible given your designs, argue quantitatively why that is the case.

```
#simple program where single cycle maybe better due
# to having only two nops in each instruction.
addi $t1, $zero, 4 # $t1 = size = 8
nop
nop

addi $t2, $t1, 8 # $t2 = size = 12
nop
nop

addi $t3, $t2, 12 # $t3 = size = 24
nop
nop

addi $t4, $t3, 16 # $t4 = size = 40

nop
nop

add $t1, $zero, $t4 # t1 = size = 40

nop
nop

# Exit program
#halt
nop
nop

li $v0, 10
syscall
```

| | | |
|---|---|---|
| $zero | 0 | 0 |
| $at | 1 | 0 |
| $v0 | 2 | 10 |
| $v1 | 3 | 0 |
| $a0 | 4 | 0 |
| $a1 | 5 | 0 |
| $a2 | 6 | 0 |
| $a3 | 7 | 0 |
| $t0 | 8 | 0 |
| $t1 | 9 | 40 |
| $t2 | 10 | 12 |
| $t3 | 11 | 24 |
| $t4 | 12 | 40 |
| $t5 | 13 | 0 |
| $t6 | 14 | 0 |
| $t7 | 15 | 0 |
| $s0 | 16 | 0 |
| $s1 | 17 | 0 |
| $s2 | 18 | 0 |
| $s3 | 19 | 0 |
| $s4 | 20 | 0 |
| $s5 | 21 | 0 |
| $s6 | 22 | 0 |
| $s7 | 23 | 0 |
| $t8 | 24 | 0 |
| $t9 | 25 | 0 |
| $k0 | 26 | 0 |
| $k1 | 27 | 0 |
| $gp | 28 | 268468224 |
| $sp | 29 | 2147479548 |
| $fp | 30 | 0 |
| $ra | 31 | 0 |
| pc | | 4194380 |

Added halt so I can run the program

The single cycle processor executes an instruction every 41.771ns.
The software schedule pipeline design executes an instruction every 19.616ns.
Therefore, if there is only 1 NOP after an instruction it would still perform better than the single cycle design because 19.616*2=39.232ns. However, if there are 2 NOPs after an instruction, it would perform worse because the processor would only complete the instruction after 19.616 *3 = 58.848ns.
The maximum number of NOPs needed after any supported instruction is 2.

With the above in mind, we simply created a program that has dependencies after every instruction which forces 2 NOPs to be inserted after an instruction for the software scheduled pipelined to avoid hazards. The 5 add instructions would complete in 5 cycles for the single cycle processor whereas the pipelined processor would need 15 cycles. This requires 5*41.771ns = 208.855ns to execute on the single cycle compared to 15*19.616ns=294.24ns on the pipelined which is a 208.855/294.24 = 0.71 speedup. Therefore, the pipelined processor is about 29% slower for this benchmark.

7. **Challenges**. This term project was challenging for every group. In at least three detailed paragraphs, describe the three most critical challenges your group faced, how you resolved them, and how you could avoid them in the future.

I believe our group faced challenges with scheduling, virtual learning, and the differences in understanding the content as well as taking responsibility for working on tasks. The first challenge of scheduling was especially challenging as we had to try and find a time that worked for everyone to meet. Some weeks we wouldn't be able to meet at all due to our own busy schedules, so that hindered our work on the project. Eventually, we were able to get a time that works for everyone, and successfully complete the two processors to the best of our abilities. In the future, we could do some more planning in advance to make sure specific meet times work out.

The next challenge, virtual learning, was also quite a hindrance on the completion of these projects. When the content is all online, it is easy to get distracted with other things since the entire internet is just a click away. Also, with not being able to have face-to-face conversations, it can be hard to recognize who is talking, making it harder to know who to talk to about a certain problem or line of code. Once we learned each other's voice, it was easy to associate them with a name and ask the questions we needed. In the future we could do introductions or ask who is talking if someone is confused.

The last challenge of not having everyone on the same page with the content was probably the most time-consuming. It was also frustrating how everyone's expectations and quality of work varied a lot and therefore our contributions greatly differed. If a problem was brought up, it would require everyone to be on the same page about the content we had, which could take some time. We were able to eventually combat this with lots of patience, explanations, and some time devoted to the issue. To combat this, members should have watched the pre-recorded lecture videos more consistently, done the readings, and tried to understand the concepts better. It would have helped a lot if everyone contributed evenly and spent more time on their assigned work to ensure it is high quality.

8. **Demo**. You will be expected to demo your benchmarking process to the Professor and TAs on the last day of the lab. Each member of the project group will be required to be present for the demo, which will take place during regular lab hours. During this time, you will describe the various design tradeoffs of your project parts, describe how they compare to each other, demonstrate simulations of your benchmarked applications, and discuss potential optimizations.

Done during the lab section on 4/30/2021.