ECSE 425 – Winter 2016
Computer Organization and Architecture

Final Report - Group 1

Joud Chataoui – joud.chataoui@mail.mcgill.ca
Sajad Darabi – sajad.darabi@mail.mcgill.ca
Sebastien Melki – sebastien.melki@mail.mcgill.ca
Hoai Phuoc Truong – phuoc.truong2@mail.mcgill.ca

***Abstract* — This document discusses the design of our assembler and simplified pipelined processor for ECSE 425. We will describe our design approach, justify our decisions and explain the evaluation of our system enhancement (cache) performance.**

I. Problem Statement

This project requires us to implement a pipelined processor that can execute a subset of the MIPS instruction set. Pipelined processors were designed to increase the overall CPU throughput by using instruction level parallelism. Each instruction is divided into smaller steps, performed by different functional units in the CPU that are pipelined together. We can overlap the execution of these smaller steps on several instructions, allowing for better use of resources and greater throughput. Different types of hazards limit the improvement in performance gained with pipelining. We were required to mitigate some of these hazards by using different techniques like forwarding and instruction caching. We were also asked to implement an assembler that assembles a subset of the MIPS instruction set into machine code for our processor to execute. Finally, we had to test our system and evaluate changes in performance for different design configurations.

II. System Implementation

In this section, we will give an overview of our system implemenation starting with our initial pipelined processor and assembler before describing our improved design that relies on instruction caching to eliminate some structural hazards and reduce memory access delay.

A. Pipelined processor

We implemented each unit of our pipelined processor individually and made sure the behavior was correct through unit tests. After that, we wired the components together and wrote integration tests to verify the correctness of our design. As one would expect, we had to revise our design multiple times along the way.

1. Instruction fetch stage

   This stage fetches an instruction from memory based on the current value of the program counter register. An interesting point to note here is that all memory accesses in our system take at least 2 clock cycles (without memory cache) because we first need to send signals to our memory arbiter which then does memory operations on behalf of our CPU. Because of this, instructions are fetched every 2 clock cycles and these instructions are interleaved with no-op (add $0 $0 $0) in the pipeline.
2. Decoder stage

   The decoder receives the instruction from the instruction fetch unit and sends appropriate signals to the different units in our pipeline, making it one of the most important units in our pipeline. The decoder parses the instruction and sends the following signals:
   - **Operation**: This is used by the ALU to perform the appropriate operation.
   - **Mem_writeback_register**: Used by the write-back stage to update the correct register.
   - **Branch_signal**: Sent to the instruction fetch unit to change PC value.
   - **Branch_address**: This signal complements the previous one, it represents the address of the next instruction when we branch. This address becomes the new PC if the branch is taken.
   - **Data1 & Data2**: These signals represent the register values at the current clock cycle. However, they are not directly sent to the execution stage. Instead, we have an intermediate forwarding block that will multiplex between these values and the ALU output in case of data dependency.
   - **Data1_register & Data2_register**: These represent the registers on which the ALU should operate (as opposed to the actual value inside those registers). This is used by the forwarding unit and possibly passed to the ALU.
   - **Previous_forwarding_destinations_output:** Passed to the forwarding unit. This signal keeps track of the destination register of the previous instruction (i.e. which register is being modified

by the last instruction). With this information, the forwarding unit can decide whether the ALU should operate on the current register value (Data1 & Data2) or the ALU output.

- **Previous_forwarding_sources_output:** This signal tells the forwarding unit whether the ALU should receive the ALU output (from the previous instruction) or whether the operand should be forwarded from memory stage (when the current instruction uses the result of a load from the previous instruction).
- **Previous_stall_destinations_output**: This signal is sent to the stall unit which decides whether the pipeline has to stall because of data dependency that cannot be resolved by forwarding.
- **Previous_stall_sources_output:** This signal is also forwarded to the stall unit and it is used with the previous_stall_destinations_output to decide whether the source of an instruction is the destination of a previous instruction and a stall is required for branch resolution.

3. Execution stage
   The execution stage is a very simple unit that simply receives signals from the decoder and the forwarding unit and acts on the passed operands based on the op-code.
4. Memory stage
   Memory stage is responsible for all data memory accesses. It sends signals to the memory arbiter and the latter deals with memory. The memory stage gets a signal from the decoder to determine whether the current instruction actually has a memory access. The memory stage also sends a signal back to the decoder to make the pipeline stall in case memory is busy.
5. Register write-back stage
   This stage takes signals from the decoder to determine whether it should wait on memory (in case of a load) or it can directly take the value to be written back from the EX stage. It also takes the correct destination register from the decoder.
6. Forwarding unit
   The forwarding unit receives the aforementioned signals from the decoder and chooses which values should be sent to the EX stage. When the latter instruction depends on a non-memory instruction that preceded it, the forwarding unit will feedback the output of the EX stage back to that stage instead of sending the value register value as read by the decoder. When the latter instruction depends on a memory instruction that preceded it by either one clock cycle or two clock cycles, the forwarding unit will pass the output of the memory stage to the EX stage. For this reason, the forwarding unit keeps track of the destination register of the two previous instructions.
7. Stall unit
   The stall unit gets the aforementioned signals from the decoder and decides whether the pipeline should stall. For example, when a latter instruction depends on a memory instruction that came right before it, the pipeline will have to stall for a clock cycle and then the result from memory will be forwarded to the EX stage. Another case where a stall is required is when we have a branch instruction that is dependent on a previous instruction. This should be handled as a separate case because our pipeline resolves branches in the decode stage (while other instructions process data in the EX stage), making forwarding impossible. As a result, we need to stall when the branch condition depends on a destination register from the immediately preceding instruction. When the pipeline should stall, the stall unit sends a signal to the decoder and the decoder will stall the appropriate units in our pipeline.

B. Assembler

The assembler was implemented with the assumption that labels will be placed on the same line as the assembly code. Since a jump instruction may refer to a label at a higher address, the assembler first scans through the entire code and produce a map from label address. The assembler will then step through the assembly code and translate each instruction into the according machine code. The translation from instruction to machine code is delegated into a group of instruction type handlers, which will recognize the type of instruction and handle the amount of parameters accordingly. If a label related instruction is encountered, the assembler will use the mapping produced in the first scan to translate that label into the correct address.

C. *Instruction Cache*

Our second deliverable consisted of improving the performance of our pipeline by adding an instruction cache. This allows us to save clock cycles when fetching instructions from memory. In fact, as mentioned before, our memory assembler sits between the fetch stage and main memory. Therefore, any memory access would take at least 2 clock cycles because of propagation delay, meaning that we had a no-op interleaved between every two actual instructions. By adding a cache between the memory arbiter and the pipeline, we were able to reduce the instruction fetch to a single clock cycle in case of a cache hit (cache tag comparison takes one clock cycle). We implemented our cache in a modular way, allowing us to change cache parameters easily. These parameters include cache size (any power of two), associativity (direct-mapped, two-way associative, four-way associative and fully associative) and replacement algorithm (random and pseudo-LRU). Our cache block size is one instruction since this is compatible with default main memory and block size was not a parameter that we are interested in.

III. TESTING AND PERFORMANCE MEASUREMENT

In this section, we will discuss how we tested our system and evaluated its performance after improving it through instruction caching. We will first describe how we tested our system from a functional perspective. After that, we will give a thorough description of our performance evaluation using different caching parameters.

A. Functional testing

As mentioned before, we tested each unit of our pipeline individually after implementation. This allowed us to easily divide the workload among team members. We usually worked in subgroups of two on each unit. For each unit, we wrote thorough test beds that covered corner cases and different functionalities. In order to perform these tests

in an isolated fashion, we drove the inputs to each unit (instead of having other units control them) and we checked the outputs to make sure the behavior was as expected.
Once behavior was validated, we wired the units together and performed integration testing of the whole pipeline. Unsurprisingly, we noticed numerous flaws in our design and had to modify several components and add intermediary signals. One important instance of that is the fact that we noticed that oftentimes, memory and write-back stages would get the wrong signal. For example, the write-back stage would write the result to the wrong register. After some investigation, we discovered that the register the data was being written to was the destination register of the instruction that came 4 clock cycles later (two instructions later). This behavior was due to the fact that the signal coming from the decoder was not delayed (it was directly wired to the write-back stage). To fix this, we added delay

B. Cache Performance Testing

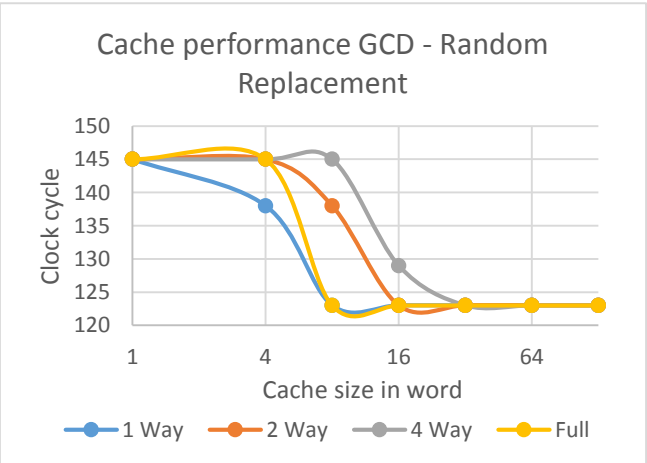
The instruction cache was tested by running our pipeline on 5 different programs. However, due to the limited presentation space, we only present analysis result on 4. To see the detail runtime for all tests, please refer to Appendix. For each program the associativity (1, 2, 4, full), the cache size as well as the replacement strategy (random, pseudo least recently used (PLRU)) are variables that could be changed to determine cache performance. For each program we measured the runtime in clock cycles without cache and with different cache configurations. We believe a discussion of the miss rate is unnecessary because this is accounted for by the runtime. In this section, we will look at the runtime of each program and the performance of each cache configuration in each case.

1. GCD of 3 numbers:

This program finds the greatest common divisor of 3 numbers using the Euclidean algorithm.

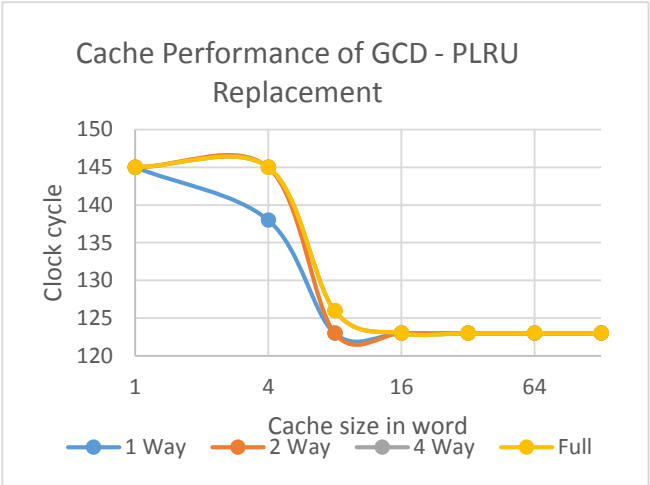
a. Random Replacement

Direct-mapped cache has the best performance in a random replacement scheme. As mentioned before, this can be intuitively explained by the fact that a direct mapped scheme is less affected by randomness in replacement since each instruction has only one slot it can be sent to.



b. PLRU

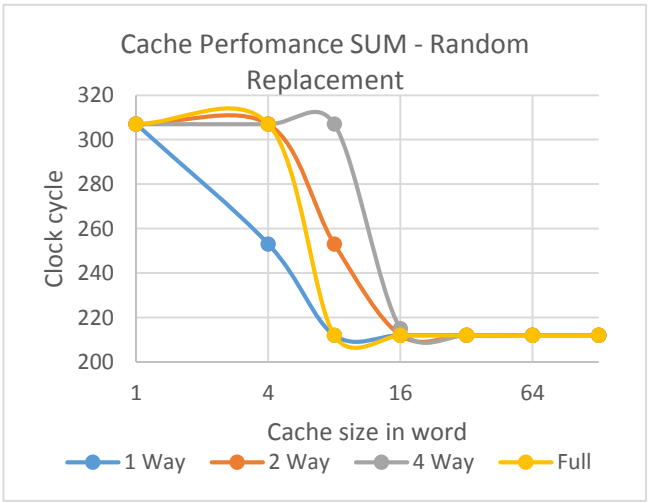
The PLRU replacement strategy provides more consistency in runtime among different associativity schemes since this replacement strategy relies on the principle of temporal locality to ensure that the frequently accessed instructions remain in the cache regardless of the set size we are dealing with.



2. Sum from 1 to n:

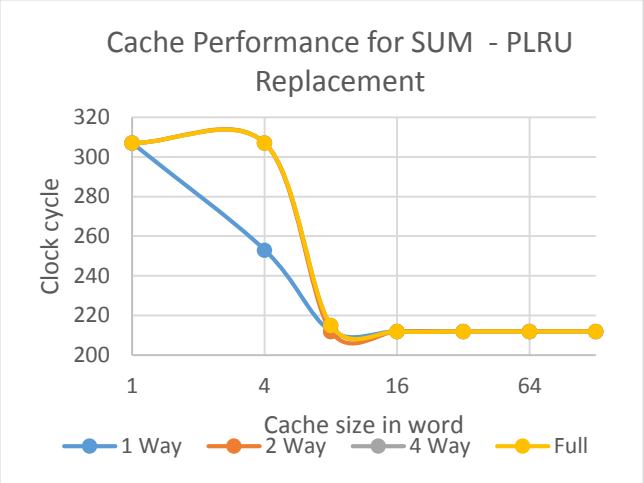
This program calculates the sum of the first n integers. The test was carried out with n = 10.

a. Random replacement



b. PLRU

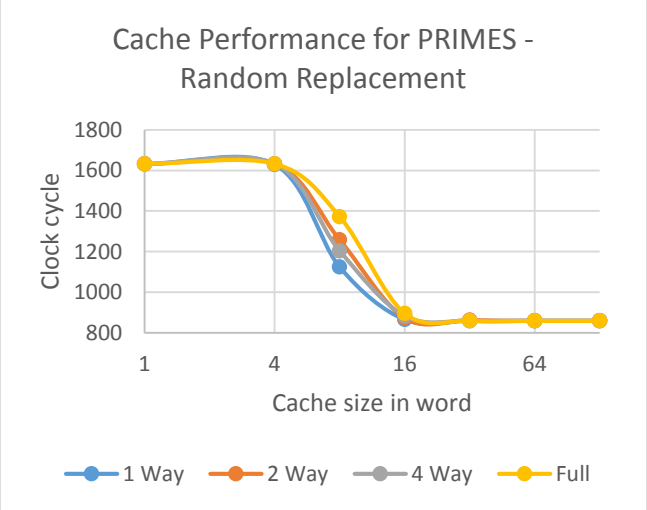
Since the main loop of the program is very small (5 instructions), the runtime improvements manifest more significant at a smaller cache size (4 words) compared to the GCD program.



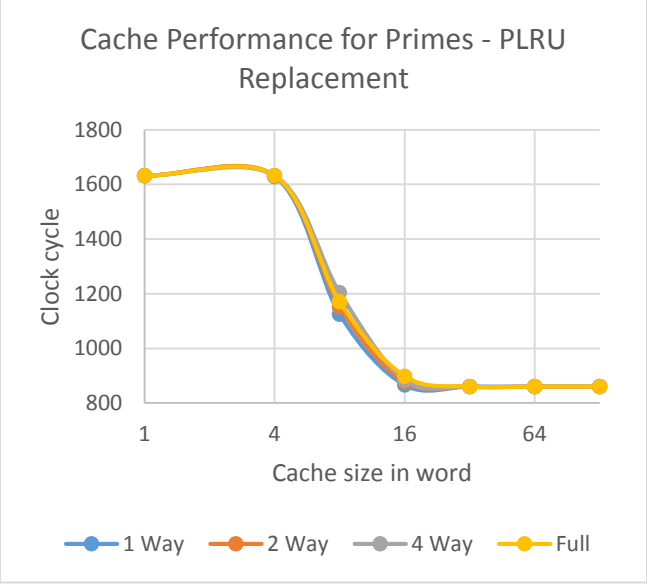
3. Prime divisors:

This programs finds all the prime divisor of a number. Our input was n = 15 for testing. It contains one large loop of 16 instructions. The large loop contains a smaller loop of 7 instructions. Consequently, the cache is only effective when cache size reaches 8 and above, which agrees with the result presented in the plots below.

a. Random



b. PLRU



4. Bitwise program:

This is the sample bitwise program provided on MyCourses. It contains no loop and a few branch command. Consequently, caching does not have any effect on the program performance. This is reflected as a **constant** runtime of 105 clock cycle regardless of cache configuration.

5. Common characteristics:

To observe the common characteristics among the tested programs, one needs to closely study all plots previously presented in this section. The **first** common characteristic of all programs is that the runtime decreases as cache size increases most likely because of a decrease in miss rate. The **second** common characteristic is that the runtimes plateau as cache size reaches 16 words because of the small loop size (less than 16 instructions). The **third** common characteristic is that direct-mapped cache offers best performance among the associativity schemes because the limited size of our test program results in very small competition for the same cache block. The **fourth** common characteristic is that PLRU replacement strategy, as previously observed with GCD program, provides a very consistent cache performance regardless of cache associativity. This manifests as the plots for different associativity schemes almost overlap in most cases in the presence of PLRU policy.

IV. Conclusion

In conclusion, this project allowed us to gain insights into how modern pipelined processors are implemented. We started by designing a simplified pipeline with basic forwarding capabilities and then improved our implementation by using an instruction cache, allowing us to reduce the instruction fetch time. While the implementation of the pipeline is somehow fixed, the cache has many design parameters that can be changed like cache size, associativity and replacement strategy.

We measured the performance of each different combinations of these parameters by running sample programs that we wrote. The cache evaluation shed light into the effect of cache size, associativity and replacement policy. The one feature we found interesting was that PLRU policy provides much more consistency in cache behavior compared to random. For better cache evaluation in the future, it would be beneficial to evaluate the cache over programs of medium (50,000 – 100,000 lines) to large size (500,000 – 1,000,000+ lines). This will truly reveals the nuances between 2 – way and 4 – way associativity as well as the power of PLRU over random replacement policy.

Other fairly simple pipeline improvements that could be implemented in the future are better branch prediction and register renaming to eliminate name dependencies.

V. APPENDIX

For detailed of the runtime obtained from our testing, please refer to Evaluation.xlsx file attached together with this document. It contains heat maps with detail runtime for each program.

For source code of the programs used, please refer to deliverable 5 submission under directory evaluations.