# CmpSci 535
# Semester Team Project

**Executive Summary:** Design an instruction set architecture (ISA) and develop a simulator for it that runs binary code that has been translated from assembly language, and that keeps a count of the clock cycles required to execute that code. To the basic simulator, add pipelining and a cache. Demonstrate the benefits of these enhancements by running benchmark programs and comparing the cycle counts against the baseline implementation.

**Design an Instruction Set Architecture:** This is the description of a computer architecture at the assembly language level, but without system calls or libraries -- just the raw instructions that the hardware can run directly. The goal of this design exercise is for you to experience the tradeoffs and design decisions involved in developing a clean-sheet architecture, and to gain a better appreciation for why architectures have the kinds of instruction sets that they do. The ISA should support at least the basic arithmetic and logical operations on integers, memory access operations (with some basic addressing modes), plus control flow instructions (branches, jumps, subroutine jump and return). A simple RISC style architecture will satisfy the requirements, but taking a more sophisticated or unusual architectural approach is one way of earning extra credit (but consult with me before you leap into the deep end of this pool). You will present your design in class and submit a draft report that describes it. Based on feedback for the draft, you will submit a revised final version.

**Develop a Simulator:** An ISA simulator is a program that includes representations for all of the internal state of the architecture (memory, registers, status bits, minor cycle state machine, etc.). It can load a machine language program from a file and interpret the individual instructions in the proper order, updating the simulated state as it does so, mimicking a hardware implementation of the architecture. You can think of it as a form of virtual machine.

**Memory Subsystem:** Memory (DRAM) takes roughly 100 cycles to respond to a random request for loading or storing data. A cache is smaller, faster, memory that holds recently used instructions and data so the processor can access them in one or a few cycles. We will see how cache is designed. The memory for the simulator will include a simulated cache, but will also have the ability to disable the cache and adjust the clock cycle accounting correspondingly. At a minimum, the cache will be unified, direct mapped, write-through, no-allocate, with lines of four words each. For extra credit, teams may implement more advanced cache variations, such as associative caches, alternate policies, different line lengths, multiple levels, etc. The basic memory with cache and timing is the first part of the simulator you will implement. It will be demonstrated with a simple driver program that enables the user to exercise the API. If designed in an object-oriented manner, the same class can be used to instantiate and link together all of the layers of the memory subsystem.

**Partial Simulator:** This first version of the simulation fetches each instruction from memory and executes it to completion before fetching the next one. It keeps and displays a count of simulated clock cycles for each instruction. It should have a user interface (UI) that enables the user to observe and modify the state of the architecture as a program executes. Thus, somewhat like a debugger, it will allow single stepping, execution to a breakpoint, and/or for a specified number of cycles. It will need commands for loading and saving programs (or the entire state), and resetting the state.

Instructions will be encoded in binary (do not use strings for this). You will implement a simple assembler/disassembler so that you can write code and see it in memory in a more meaningful form. Note that the architecture has only one memory, which contains both instructions and data, all represented in binary. So you will want to be able to switch the simulator's display of memory contents between instruction and data modes. You will also want to be able to see the data in multiple formats (binary, hex, decimal, and perhaps floating point, ASCII and/or Unicode). Because the memory will be too large to be displayed on the screen all at once, you will need a UI that can selectively display sections of it.

For the first demonstration, you just need to have enough instructions working to show that all of the major operation types are working. The simulator should be able to load a binary program, and then single-step execute it. The program will demonstrate loading and storing values between memory and registers, register-to-register arithmetic, and a conditional branch. The equivalent of a for loop that reads a few pairs of values from memory, adds them, stores the results back to memory, and exits when the loop control counter reaches the termination condition. The simulator should keep a count of the execution cycles, and it should be possible to run both with and without a cache.

The user interface at this stage should support viewing the registers (including PC, status, etc.) and memory (main and cache) in at least one format (e.g., hexadecimal), loading a program from a file, and stepping through it to see how the state changes. It should also display a count of simulated clock cycles.

**Baseline Simulator:** At this point, you will have all of the major components of the simulator working. The next step is to fully populate the instruction set, and add an assembler to generate binary code. The user interface will need to be extended to support running to completion, breakpoints, viewing memory in different formats, and managing the configurations of the simulator.

**Pipelined Simulator:** Pipelining is a basic mechanism for adding parallelism to an architecture. It enables a short sequence of instructions to execute simultaneously, by breaking their actions into a series of steps that can be done in stages. Each instruction in a sequence occupies one stage at a time, where it has the corresponding step performed for it. Because there are multiple stages, multiple instructions are able to have their independent steps executed simultaneously.

Some instructions have dependences on others, so their smooth flow through the stages can be disrupted. The baseline simulator will be augmented to execute instructions in a pipelined manner, which will require more complex control and clock cycle accounting. The minimum pipeline will consist of the standard five stages: fetch, decode, execute, memory, and write back, with no forwarding or interrupt handling. For extra credit, longer pipes, forwarding, and interrupts may be added, or for really ambitious teams, multiple pipes for superscalar issue can be implemented. Another optional enhancement would be to add a branch predictor.

**Performance Evaluation:** By the final demo the simulator should support four modes of operation: no cache or pipeline, cache only, pipeline only, cached and pipelined. Your final report should show the results of running at least two benchmark programs through the simulator in all four modes.

**Benchmarks:** These are programs that are run on all versions of the architecture, and used for the purposes of comparing performance. They must be sufficiently complex and use enough data to actually stress the limitations of the pipeline and cache. At least an exchange sort, and a matrix multiply should be implemented (both on integer data). Additional benchmarks can be created for extra credit, and may be necessary to demonstrate special features.

**Grading:** The project will have a series of milestones with demonstrations in class on those days. Each demonstration will be graded on the basis of how well it meets the requirements for that milestone. Bugs, missing features, clumsy interfaces, poor preparation, etc. will result in lower grades. Meeting the minimal requirements is equivalent to a B-level grade. Going beyond the requirements, with additional features, getting ahead of schedule, a clean interface, good preparation for the demo, etc. will result in a higher grade. Part of the demonstration will include a description of software engineering methodology and tools used to help manage the project (both design and code), and validate the code. Although this is extra work to set up at first, experience has shown that it produces better results with less work as the pieces come together later in the semester.

**Project Reports:** Early in the semester, a report will be submitted in which you provide a formal description of your architecture, and a management plan for implementing the project (including who will contribute what to the team effort at each stage, and what software engineering tools and approaches will be employed). The report will be submitted once as a draft that is given an interim grade, with comments that indicate how it can be improved. It may then be updated and submitted a second time for a reevaluation, which will result in a grade that replaces the first one.

At the end of the semester, you will submit a final report that describes the architecture (a revision of the earlier report that reflects changes and clarifications that arise later), the simulator (it's overall design, and a user manual), the software engineering methods used, the performance results of running the benchmarks on the different simulator configurations, and a summary of what you have learned from the project and from

experimenting with the simulator. You will submit a draft of this report one week before the end of classes, for an interim grade with feedback. The final version is due at the final exam and its grade replaces the interim grade. If you get a perfect score on either draft report, you don't have to submit the final version.

As with the demos, a basic report that just fulfills the requirements will earn a B-level score. A higher score will result from a report that goes into more detail, shows results from additional configurations or benchmarks and is able to draw broader conclusions.

**Report Due Dates:**

| | |
|---|---|
| First report draft: | Wed., 2/7 |
| First report: | Wed., 2/21 |
| Final report draft: | Wed., 4/25 |
| Final report: | Tue., 5/8 (Final exam) |

**Presentations and demos:**

| | |
|---|---|
| Memory, cache, and timing demo: | Mon., 3/5 |
| Simulate basic instruction set, assembler, minimal UI, running code: | Mon., 3/26 |
| Full ISA, UI, benchmarks, with/without, cache: | Mon., 4/2 |
| Final demo, baseline, cache, pipeline: | Mon., 4/23 |