CompSci 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. The simulator will support basic pipelining and a cache. Demonstrate the benefits of each of these enhancements separately and together by running benchmark programs and comparing the cycle counts against the unenhanced implementation. A graphical user interface will enable observation of all machine state and operations.

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 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.

What is a Simulator? An ISA simulator is a program that includes representations for all of the internal state of the architecture (memory, registers, status bits, pipeline dependences, 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, one clock cycle at a time. You can think of it as a form of virtual machine.

Memory Subsystem: Main 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 fewer cycles. The memory for the simulator will include a simulated cache, but will also have the ability to disable the cache, to evaluate its performance impact. 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 manually exercise the API by storing and loading memory values, seeing the cycle counts, and examining memory contents.** 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 instructions from memory and passes them through a pipeline consisting of at least five stages (fetch, decode, execute, memory, write back), with no forwarding or interrupt handling. It keeps and displays a count of

simulated clock cycles. It should have a user interface (UI) that enables the user to observe 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). At a later stage, you will implement a simple assembler/disassembler so you can write code and see it in memory in a more meaningful form. Note that the architecture has only one main memory memory, which contains both instructions and data, all represented in binary. 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 demonstration, you just need to have enough instructions working to show that all of the major operation types (load, store, ALU, branch) are working. The simulator should be able to: (1) load a binary program, (2) single-step execute it

The test program must at least demonstrate loading and storing values between memory and registers, register-to-register arithmetic, and a conditional branch. **A good demonstration is the equivalent of a for loop that reads a series of pairs of values from memory, adds them, stores the results back to memory, and exits or halts when the loop control counter reaches the termination condition.**

The simulator should also:

- keep and display a count of the execution cycles
- be able to run both with and without a cache
- be able to run both with and without the pipeline enabled

Having the pipeline disabled means each instruction goes all the way through the pipe before the next one starts. Thus, the simulator will have four modes of execution: (1) no cache or pipe, (2) cache only, (3) pipe only, (4) both cache and pipe.

The user interface at this stage should support viewing the registers (including PC, status, etc.) and memory (main and cache) in hexadecimal, loading a program from a file, and stepping through it to see how the state changes. At this point, you will have all of the major components of the simulator working.

Full ISA/GUI demo: 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, and managing the configurations of the simulator. For your own benefit, you will likely want to enable viewing values in different formats (e.g., decimal).

For extra credit, this is the stage where you can add more sophisticated caches, longer pipes, forwarding, interrupt handling, I/O devices, extra benchmarks, etc. For really ambitious teams, multiple pipes for superscalar issue can be implemented. Another ambitious enhancement would be to add a branch predictor.

Performance Evaluation: 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 for comparing performance. They must be sufficiently complex and use enough data to stress the limitations of the pipeline and cache. At least (1) an exchange sort, and (2) a matrix multiply should be implemented (on integer data). The data sets should be big enough to more than fill the cache, and every line in the cache should be accessed at least ten times. Additional benchmarks can be created for extra credit, and may be necessary to demonstrate special features.

Demo 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 Design Report: Early in the semester, a report will be submitted in which you provide 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), and a formal description of your architecture. 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.

Following the management plan, the report will have a section detailing overall design choices and parameters, and then a section that describes all of the architecture's instructions and their operation in sufficient detail that it would be possible for another team to take the report as a specification and build a simulator for your architecture. Some of the report will be derived from exercises done in class, but you will need to expand upon those to complete it.

Design choices and parameters to be addressed include: General purpose or special purpose? Clean sheet or based on an existing ISA (which one)? Distinguishing features. Word size, data types and supported operations on each type, number and types of all registers (including PC, status, etc.). Description of the execution model: number of addresses, instruction fetch paradigm, memory organization (Princeton/Harvard, word/byte addressed, address space, addressing modes), endianness, etc.

Be sure your instruction lists are complete, and make sense for your design choices. For example, if you say you will support 256 bit AES encryption, then you need registers to hold those values, and loads/stores for them, in addition to the AES operations. An integer-only ISA should at least have load, store, integer logical and arithmetic operations (including shifts), unconditional and conditional branching (with a mechanism for doing comparisons), subroutine call and return (some exotic designs might do some of these differently). Each instruction should have its full format and bit-level encoding specified, and a description of all the state that it will access and modify, and how it will operate.

Final Report: At the end of the semester, you will submit a final report that:

1. Describes the architecture (a revision of the earlier report that reflects changes and clarifications that arise later)
2. Describes the simulator (it's overall software design, and a user manual)
3. Specifies the software engineering methods used in its development
4. Lists the efforts by person: who ended up doing what on the project
5. Presents the performance results of running the benchmarks on the different simulator configurations, with analysis of the effects of the different modes
6. Concludes with a summary of what you have learned from the project and from experimenting with the simulator (it's helpful to keep a journal of all problems/decision points encountered and how they were resolved)

A draft of this report will be submitted with the final demo, for an interim grade with feedback. The final version is due at the time of 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:	Mon., 2/24
First report:	Mon., 3/3
Final report draft:	Wed., 4/30
Final report:	Final Exam Period

Presentations and demos:

ISA presentation:	Mon., 2/24
Memory, cache, and timing demo:	Wed., 3/12
Simulate basic instruction set, pipeline, cache, minimal UI, running code:	Wed., 4/2
Full ISA, UI, assembler:	Fri., 4/18
Final demo, benchmarks on four modes:	Wed., 4/30