

Your team will design an instruction set architecture for a computer system, write a software emulation of the system, and evaluate its performance. The initial emulation will be for an atomic fetch-execute, flat memory version of the architecture. You will then add pipelining and caching. Each of these additions will be evaluated for its effect on performance, separately and together. To evaluate the performance of the system, you will need to write several benchmark applications to run on the emulator. These must exercise the pipeline as well as the cache, so they should be computationally intensive and use a large memory footprint. I recommend an insertion sort and a matrix multiply as relatively easy to program yet able to operate on a large data set.

The architecture should support at least 1 M (2^{20}) words of storage. It should provide basic arithmetic and logic operations, memory access operations, and branch/jump operations. Integer and floating-point arithmetic should be provided. The architecture does not need to support supervisor state or specific I/O capabilities. I/O can be simulated through the emulator's user interface's capabilities for examining and changing registers and memory. The emulator must keep an accurate count of the clock cycles required to execute each instruction so that you can measure performance.

You are free to choose the programming language for development of your emulator. I do, however, recommend the use of an object-oriented language such as Java or C++ to reduce the software engineering effort. Although teams have built projects using Shockwave in the past, they have run into major problems with performance, which have not usually surfaced until trying to make full benchmarking runs late in the semester. The choice of language probably depends on what both team members are most comfortable with. Be certain from the start that you and your teammate have access to compatible tools. Teams in the past have run into serious trouble when they assigned development work by phases, and found at hand-off time that their environments were incompatible.

Design and Development Phases

Instruction Set Architecture

The first step in the project is to define the architecture that you will emulate. I recommend that you focus on the Reduced Instruction Set Computer (RISC) approach as it produces simple and regular instruction sets that are easier to emulate. A typical RISC instruction set has all instructions fitting in a single memory word. The instructions all have a common "format" field that identifies the type of the instruction. Typical RISC sets have a small set of formats: memory, jump, branch, arithmetic/logic unit (ALU). They have a large set of registers (32 integer, 32 floating-point, 32 control), and all memory references are through explicit load and store instructions. RISC architectures usually support a small number of addressing modes, and these are often specific to the instruction type. For example, memory references are register-indirect, branches are immediate offset, ALU operations are all register direct, and so on.

An alternative approach is an accumulator architecture, which has just one main register. The accumulator approach is somewhat simpler to emulate, but is more difficult to program with benchmarks. We will cover instruction set options in our class discussions. One instruction that is not usually present in modern processors that you will want to include is a "halt" operation. If you prefer, this can be encoded as, for example, a jump to a specific address – effectively a trap instruction, but because there is no operating system, the effect is to turn control over to the user interface and stop the processor.

A preliminary ISA design will be due as part of a homework on **Sept. 25**. After receiving feedback on the design, a polished ISA design will be presented **Oct. 2**.

Basic Data Path

The data path is the route that data follows in ALU operations: From registers, through functional units, and back to registers. Having defined the ISA, you now have encodings for the arithmetic and logical operations that your architecture supports. Your development of the emulator begins with the simulation of the registers and functional units.

The inputs for activating the data path are: Source register numbers, destination register number, operation, and clock. Most ALU operations execute in a single clock cycle, but multiply and divide, for example, take multiple cycles. The simulation for the data path should allow the user to enter the register number and operation inputs and then indicate (for example, with a button press) that the simulation should advance one cycle. A count of the clock cycles should be displayed, and the user should be able to reset the count.

Your emulation should show the data in all of the registers, and allow the user to change the register values. For a multi-cycle integer operation such as multiply, divide, and shift, any internal registers should be updated as they would be in the actual hardware, and their contents should be displayed after each clock cycle. You do not need to simulate the actual hardware implementation (that is, the gate-level operation) of the single-cycle operations, or the floating-point operations.

You can use the native operations of the programming language to do the actual computation, as long as you correctly account for the time. For example, a floating-point multiply operation includes an integer multiply (but with fewer bits, because the mantissa is smaller than a word), an integer add, a normalize (shift) operation, and rounding that take some number of cycles, N . You can have the simulation for this instruction simply perform no-ops for the first $N-1$ clock cycles, and then on cycle N , it computes the product and stores it in the result register.

The data path will be demonstrated in class on **October 18**. Be sure that you have tested your demo on the hardware that you'll use in class. All demos will be shown on an LCD projector for the entire class to see, so if you plan to use your own laptop, you should arrange with me for a pretest of connecting it to a projector if you haven't done so previously.

Memory Data Path

The next step is to add the memory into the data path. A memory operation requires the inputs: Mode (read or write), Source/Destination register, Address (which may be either a value or a register number, depending on the ISA), and clock. The memory is just a large array of words (at least a megaword). The address is used to index into this array (and must be at least 20 bits in size to access the megaword). The value in the Source/Destination register is either replaced by the array value at this index (read), or replaces the value at the address index (write).

Main memory, however, is much slower than the CPU. For each memory access, your emulation should take 100 cycles. It's likely that you'll want to add a "Clock X100" operation to your user interface to avoid having to single-step through all 100 cycles of every memory access. You should not update the display of the registers until the end of the 100 cycles. The user should be able to view at least one section of memory by entering a start address for a memory display area. (With OOD, it should actually be easy to open more than one memory view at a time, which you may find helpful in debugging your benchmarks, but this is purely optional – consider using the JPanel class in Java).

It is also advisable to provide operations that save memory to a file and load it back in. Once you begin to program the system, you'll want to save code and data for reuse.

Note that this memory will eventually contain both code and data. At this stage, it holds only data, and the instructions are provided manually, one at a time. The next stage of the project adds a control unit that fetches instructions from this same memory, and executes them automatically. For speed and compactness, I recommend that you don't use strings to represent values in memory. Using int or Java's BitSet are better choices.

The memory data path will be demonstrated in class on **November 1** Note that this is effectively just over a week after the basic data path demo because our midterm exam falls between the two demos. This is an easy addition if you've thought ahead in the design of the basic data path.

Instruction Path and Control Unit

At this point you have all of the operational components of a computer except for the automatic fetch and execution of instructions. You must add a program counter (PC), an instruction register (IR), and program logic to decode the instruction and drive the different units. Part of this control unit is also responsible for implementing branch and jump operations, whose basic function is to set the PC to a different value.

The instructions should be stored in the same memory as the data. The control unit is a finite state machine that fetches a value from memory (using the PC as the address) to the IR, then (based on the instruction type), it calls upon the different units to perform the specified operations. In effect, it automates what you've been doing by hand as you

entered the inputs for the data and memory paths. Of course, it also updates the PC value to the next word for non-branch/jump instructions.

The control unit must correctly handle the case of an operation that takes multiple cycles. For example, in a memory access, the control unit must issue 100 clocks, waiting for the memory to signal that it is done. (You may need to add such a signal to the memory if you haven't provided it in the initial design.)

You will probably find it useful to add some support for breakpoints into the emulator at this time to help you with debugging benchmark code. For example, stop on fetching an address, or stop on loading an address. You should also supply an instruction-step operation in addition to the "Run", "Clock", and "Clock X100" operations, so that we can observe the emulation running instruction-by-instruction. Your user interface should show the PC and IR whenever the emulator halts. It is not necessary to update the display for each instruction when the emulator is in "Run" mode. At this point, you may also choose to disable the display of the intermediate steps of multi-cycle operations, except when single clock operation is used.

The demo for the instruction path and control unit (the baseline processor), together with one benchmark (see below), is on **November 15**. For this demo, you need only show a simple program that demonstrates fetching and execution of an instruction of each type.

Benchmarks and Performance

The next step does not require you to change the emulator, and should be started in parallel with its development. You'll need some significant applications to use in evaluating the performance enhancements that are to be added in the next two phases. Suggested applications are given above. You may choose others with prior approval from me. These must be coded in the instruction set of your machine. You can either hand-code them or develop a simple assembler for your instruction set (some groups have also developed a disassembler to enable viewing memory as instructions).

To obtain a large array of values on which to operate, you can use a random number generator. However, to be sure that your performance runs are repeatable, you must have some way of saving this array and loading it back into your emulator's memory. The data array should be 90K (1K = 1024 words) in size so that we can see some benefit from use of cache and pipeline in the next phases of the project.

On **November 15** you will also demonstrate the emulator executing one of the benchmarks (integer insertion sort). Because execution of the benchmark is usually time consuming, we will have everyone run it at the same time and compare execution times (both cycle counts and emulator run time). The floating-point matrix multiply benchmark will be demonstrated together with the demo of the pipeline.

Pipelined Instruction/Data Path

For this part of the project, the fetch-execute cycle is divided-up and pipelined. The pipe should at least be divided into Fetch, Decode, Execute, Memory, and Write-back stages (the more stages, the shorter the clock period, and the greater the parallelism). We will cover in class how these operate, and we'll see how data and control hazards must be accounted for by inserting stall slots into the pipe. The benchmarks will be rerun to compare their execution clock cycles with the baseline processor.

For the demo on **December 4**, you should run a short program that illustrates the operation of the pipeline, in addition to the matrix multiply. The user interface should show the progress of the different instructions through the pipe stages when single-clocked, and should update the pipe status after execution of an instruction sequence. By that stage you should also have run both of your benchmarks and be able to report the performance figures.

Cache

The last phase of the project is to add a simple direct-mapped 1K unified cache to the memory system. Because this is an addition that is isolated within the memory unit, you should be able to add it to both the baseline processor and the pipelined processor by relinking to the different version. Also, because it is independent of the pipeline development, you should work on the cache implementation in parallel with developing the pipelined version of the processor.

The cache is just a second block of 1K ($= 2^{10} = 1024$) words of memory that responds to a memory access in a single cycle, if the requested value is present. When a memory read occurs, the address is mapped to a location in the cache using a subset of its bits. The remainder of the address is compared to the "tag" field of that cache location and if there is a match, the value is taken from the cache and given to the data path in one cycle. If the tags do not match, then the value is read from main memory (with a 100 cycle penalty), and replaces the value in the cache. If the value in the cache was marked as "dirty," meaning it has been changed by the processor, then it is written back to memory with another 100 cycle penalty (in a more advanced system, the processor would continue while the memory is writing, until it encounters another operation that requires a memory access).

When a value is written to memory, the tag is compared to the address and if there is a match, the write goes into the cache. If there is a mismatch and the value is dirty, then the current value is first written to memory and then replaced by the newly written value. If it was a "clean" value, then it is simply overwritten.

This is far from a complete description of the cache, and we'll cover it in detail in class. However, you should see that the cache can greatly improve performance as long as values are used repeatedly. You may think of ways to restructure the loops in your benchmarks to enhance this reuse.

The last demo will be on **December 13**, and you should show a small program that demonstrates the operation of the cache, in addition to the two benchmarks. The cache should be shown in the user interface (or at least a selectable portion of it). You should also report the benchmark performance with the cache at that demo. Just for fun, we may also run a race to see whose simulation completes most quickly.

General Notes About Demos

Each demo day, we have many demos to view. This means that each team gets just a few minutes to show their demo. Plan your demo in advance so that it goes smoothly and quickly. Write down what you want to demonstrate. Make sure that the demo is ready to run and has been fully tested. But also be prepared for me to ask you to do something that you have not anticipated – I may want to check some feature or clarify some operation that seems obvious to you.

Be sure that you know how to connect your laptop to the projector and switch it to display via its video port. If you have a computer that has a separate converter for video output, remember to bring it. If you plan to use my laptop for displaying your demo via the net, be sure to get me the URL so I can check it prior to the demo day (and don't count on my getting your email the morning of the demo – or even getting it the night before). My laptop does not read floppy disks – only CDs. It is a Macintosh G4 running OSX, and has Netscape, Explorer, and Safari with wireless network access.

If you can't get a demo to run in class, I will cut you off after a few minutes and arrange for a separate demo during office hours. It is better to present a partial demo than to not present at all. I will deduct points for late demos. If you run a demo and something doesn't work, I will ask you to get it working by the next demo. If it works at the next demo and all of the new elements for that demo are also working (in other words, if you've caught up), then it won't count against you. What I want to avoid is having a group fall behind and stay behind, or slip even further. Under no circumstances can you finish the project late. The final evaluation of the project is on the last demo day.

The final report for the project – a summary of the ISA, a discussion of the structure of the emulator, instructions for operating the emulator, and performance figures for the benchmarks running on the baseline, baseline+pipeline, baseline+cache, and baseline+pipeline+cache, is due on the day of the final exam.

Honors Projects

If you are enrolled in the honors section of this class, you may elect to do either a semester reading and report, or an enhanced version of the project. Enhancements could include: Emulating an actual processor, adding branch prediction, implementing more advanced cache configurations, providing superscalar issue to multiple pipelines, and so on.