

CS101: Digital Systems Construction

Prof. Shimon Schocken

“Nothing is more important than seeing the sources of invention which are, in my opinion, more interesting than the inventions themselves.” Leibnitz (1646-1716)

Fall 2005 semester, Wednesday and Friday, 3:30-5:00

Prerequisite: Open to computer science and engineering majors from both undergraduate and graduate levels, the only prerequisite being a programming experience (CS50 or equivalent). All the computer science knowledge needed for completing this course is given in the course lectures and textbook.

Course overview: The course objective is to integrate key topics from *algorithms*, *computer architecture*, *operating systems*, *compilers*, and *software engineering*, in one unified framework. This will be done constructively, by building a general-purpose computer system from the ground up. In the process, we will explore many ideas and techniques used in the design of modern hardware and software systems, and discuss major trade-offs and future trends. Throughout this journey, you will gain many cross-section views of the computing field, from the bare bone details of switching circuits to the high level abstraction of object-based software design.

At which stage in the program it is best to take this course? Well, it doesn't really matter. If you've already taken some CS courses, this course will help integrate them into a coherent picture. If you've just taken introduction to programming, then this course will provide a solid basis for subsequent courses in the program.

Methodology: This is mostly a hands-on course, which evolves around implementing a series of hardware and software modules. Each module development task will be accompanied by a design document and an executable solution (illustrating *what* the module is supposed to do), a detailed implementation document (proposing *how* to build it), and a test script (specifying how to *test* it). The homework assignments will be spread out evenly, so there will be no special “crunch” toward the semester's end. Each lecture will start by reviewing the work that was done thus far, and giving instructions on what has to be done next. The homework assignments can be done in pairs.

Programming: The hardware projects will be done in a simple Hardware Description Language (HDL) that can be learned in a few hours. The resulting chips (as well as the topmost computer-on-a-chip system) will be tested and simulated on a supplied hardware simulator that can run on your home PC. The software projects can be done in any language of your choice (but if it's not Java, C++, C#, or Pearl you have to get the instructor's approval first).

Resources: All the course materials – lecture notes, simulators, software tools, tutorials and test programs -- can be downloaded freely from the course web site. All the supplied software can run as is on either Windows or Linux.

Course Grade: 60% homework grades and 40% final examination.

Textbook: Nisan and Schocken, *The Elements of Computing Systems*, MIT Press, 2005.

Course Plan (by week)

1. **Getting started:** Demonstration of some games (like *Pong* and *Tetris*) running on the final computer built in the course; The abstraction / implementation paradigm and its role in systems design; Overview of the Hardware Description Language (HDL) used in the course; Designing a set of elementary logic gates from primitive Nand gates; Implementing the gates in HDL.
2. **Boolean arithmetic:** Using the previously built logic gates to design and implement a family of binary adders, culminating in the construction of a simple ALU (Arithmetic-Logic Unit).
3. **Sequential logic:** Design and implementation of a memory hierarchy, from elementary flip-flop gates to single-bit cells to registers to RAM units of arbitrary sizes.
4. **Machine language:** Introducing an instruction set, in both binary and symbolic versions; Discussing some trade-offs related to its design; Writing some low-level programs in its assembly language.
5. **Computer architecture:** Integrating all the chips built in weeks 1-3 into a computer platform capable of running programs written in the instruction set introduced in week 4.
6. **Assembler:** Basic language translation techniques (parsing, symbol table, macro-assembly); Building an assembler for the assembly language presented in week 4.
7. **Virtual machine I:** Pushdown automata, stack machines, the role of virtual machines in modern software architectures like Java and .NET; Introduction of a typical VM language, focusing on its arithmetic commands. Building a program that translates from this VM language into the assembly language presented in week 4.
8. **Virtual machine II:** Completing the design of the stack machine and its associated VM language, focusing on flow-of-control and subroutine call-and-return commands; Completing the implementation of the VM translator.
9. **High Level Language:** Introducing a simple high-level object-based language with a Java-like syntax; Discussing various trade-offs related to the language's design and implementation; Writing a simple interactive game and running it on the computer built in weeks 1-8 (using the compiler and OS that will be built in weeks 10-13).
10. **Compiler I:** Context-free grammars and recursive ascent parsing algorithms; Building a syntax analyzer (tokenizer and parser) for the high-level language presented in week 9; The syntax analyzer will generate XML code reflecting the structure of the translated program.
11. **Compiler II:** Code generation, low-level handling of arrays and objects; Morphing the design of the syntax analyzer into a full-scale compiler; This is done by replacing the routines that write passive XML with routines that generate executable VM code for the stack machine presented in weeks 7-8.
12. **Operating system I:** Discussion of OS/hardware and OS/software design trade-offs, and time/space efficiency considerations. Design and implementation of classic arithmetic and geometric algorithms (needed for implementing the OS's *Math* and *Graphics* libraries).
13. **Operating system II:** More classic mathematical, memory management, string processing, and I/O handling algorithms, needed for completing the OS implementation.
14. **More fun to go:** Discussing how the computer system (hardware and software) built in the course can be improved along two dimensions: optimization, and functional extensions; Proposed directions for further explorations.