CS101: From Nand to Tetris

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)

Prerequisite: Open to both undergraduate and graduate students, the only prerequisite being a programming experience. All the computer science knowledge necessary for completing this course is given in the course lectures, projects, and textbook.

Course overview: The course objective is to integrate key notions 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 should I take this course? 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, the course will provide a solid framework for subsequent courses in the program.

Methodology: This is mostly a hands-on course, evolving around building a series of hardware and software modules. Each module development task is accompanied by a design document, an API, an executable solution, a test script (illustrating *what* the module is supposed to do), and a detailed implementation plan (proposing *how* to build it). The projects are spread out evenly, so there will be no special pressure towards the semester's end. Each lecture will start by reviewing the work that was done thus far, and giving guidelines on what to do next. The projects 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, running on the student's computer. The software projects can be done in either Java or Python.

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

Course Grade: 70% homework grades and 30% final examination.

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

Course Plan (by week)

1. Hello, World Below: Demonstration of some interactive games (like *Pong*, *Tetris, Sokoban*) running on the computer built in the course, tracing their execution from the object-oriented language level down to the Nand level; 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. Combinational logic: using the logic gates built in week 1 to design and implement a family of binary adders, culminating in the construction of a simple ALU (Arithmetic-Logic Unit).

3. **Sequential logic:** using the logic gates built in week 1 to design and implement a memory hierarchy, from elementary flip-flop gates to registers and RAM units of arbitrary sizes.

4. Machine language: introducing an instruction set, in both binary and assembly (symbolic) versions; Writing some low-level assembly programs and running them on a supplied CPU emulator.

5. Computer architecture: Integrating the chip-sets built in weeks 1-3 into a computer platform capable of running programs written in the machine language presented 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: The role of virtual machines in modern software architectures like Java and .NET; Introduction of a typical VM language, focusing on stack-based arithmetic, logical, and memory access operations; Implementing part I of a VM translator that translates from the VM language into the assembly language presented in week 4.

8. Virtual machine II: Continued discussion of the VM abstraction and implementation, focusing on stack-based flow-of-control and subroutine call-and-return techniques; Extending the VM translator from week 7 into a complete VM implementation that serves as the back-end component of the compiler built later in the course.

9. High Level Language: Introducing *Jack*, a simple high-level object-based language with a Java-like syntax; Discussing various trade-offs related to the language design and implementation; Using Jack to write a simple interactive game and running it on the computer built in weeks 1-5. This project makes use of the compiler and operating systems built in the remainder of the course.

10. Compiler I: Context-free grammars and recursive parsing algorithms; Building a syntax analyzer (tokenizer and parser) for the jack language; 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 syntax analyzer built in week 10 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.

12-13. Operating system: Discussion of OS/hardware and OS/software design trade-offs, and time/space efficiency considerations; Design and implementation of some classical arithmetic and geometric algorithms that come to play in the implementation of our OS, as well as classical 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; Exploring possible software extensions (e.g. building an HTTP server), and hardware extensions (e.g. FPGA implementations).