From Nand to Tetris[1]

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

The elementary logic gate *Nand* (or its close relative *Nor*) is the fundamental building block from which all hardware platforms are made. In this course we start from the humble Nand gate and work our way through the construction of a modern computer system – hardware and software – capable of running Tetris and any other program. In the process, we will learn how computers work, how they are constructed, and how to plan and execute large-scale systems building projects.

Prerequisite: The course is open to undergraduate and graduate students from all schools, the only prerequisite being an Introduction to CS (Computer Science) or equivalent. This is a self-contained course: All the CS knowledge and tools needed for building the computer and completing the course are given in the course itself.

Overview: Nand to Tetris is a journey of discovery, synthesizing key topics in applied CS in one unified framework. This is done constructively, by building a general-purpose computer system from the ground up. We'll explore key ideas and techniques used in the design of modern hardware and software systems, and discuss major trade-offs and future trends. As we go through this journey, you will gain many cross-section views of the computing field, from the details of bare bone switching circuits to the compilation of high-level software abstractions. We will also provide a historical perspective, narrating the key people, circumstances and innovations that paved the way to the digital computer.

At which stage in the program it is best to take this course? If you've already taken some CS courses, Nand to Tetris will help integrate them into a coherent picture, showing the forest for the trees. If you've just taken introduction to CS, Nand to Tetris will provide an organizing roadmap that will accommodate subsequent courses in applied CS.

Methodology: This is mostly a hands-on journey. Each hardware and software module of the emerging computer will be introduced by an abstract specification and an executable solution, illustrating *what* the module is designed to do. This will set the stage for a detailed implementation guideline, proposing *how* to build the module, and a test script, specifying how to *test* its evolving implementation.

Programming: The hardware modules will be built using a simple Hardware Description Language, learned in the course. You will simulate and test your HDL-based chips on a supplied hardware simulator running on your PC – exactly how chips are designed in industry. The software projects (an assembler, a virtual machine, and a compiler) can be done in Java, Python, Perl, or other languages approved by your instructor.

Testing: The hardware and software modules built in the course can be tested in three ways. First, you will test them on your PC, using supplied tools and test scripts. Second, you'll be able to upload them to a server that tests correctness and provides useful feedback. Finally, the course staff will review your work, focusing on readability, elegance, and efficiency.

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

Textbook: Nisan and Schocken, *The Elements of Computing Systems*, MIT Press (recommended, not required).

[1] This syllabus describes a one-semester course that is normally taught in thirteen three-hour lectures.

Course Plan (by week)

Part I: Hardware

We'll build a general-purpose computer equipped with a symbolic machine language and an assembler. To give motivation and context, we'll start the journey by demonstrating some of the numerous video games that were already developed for this computer: *Pong*, *Snake*, *Space Invaders*, *Life*, *Google's Dyno*, and more.

1. **Boolean logic**: We will start with basic concepts in gate logic and Boolean algebra, and discuss the importance of separating abstraction from implementation. We'll provide a theoretical proof that any computer can be built from Nand gates only, and present a simple Hardware Description Language that allows doing it in practice. In project 1 you will use this HDL to build a set of elementary logic gates (And, Or, Not, Mux, …) from primitive Nand gates.

2. **Boolean arithmetic:** We'll learn how to use bits (0's and 1's) for representing signed numbers, and how to use logic gates for realizing arithmetic operations on such bitwise representations. We will then show how elementary logic gates can be used to build a family of *adder* chips, culminating in the construction of an *Arithmetic-Logic Unit*. In project 2 you will build this ALU, using the logic gates built in project 1.

3. **Memory:** We'll explain how sequential logic, clocks, and elementary time-dependent gates called *flip-flops* can be used to maintain state and realize memory units. We'll also discuss the crucial invention of the *transistor*. In project 3 you will build a memory hierarchy, from single-bit cells to registers to RAM units of arbitrary sizes.

4. **Machine language:** This is the delicate software-hardware interface where high-level programs are ultimately reduced to concrete binary codes committed to silicon. We'll introduce an instruction set and an abstract computer architecture, and learn how to write low-level programs in this framework. We'll also learn how to handle input / output devices (a screen and a keyboard) using memory-resident bitmaps. In project 4 you will write some low-level interactive programs in assembly language and execute them on a supplied CPU emulator, running on your PC.

5. **Computer architecture:** We'll describe the stored program concept, the fetch-execute cycle, a general computing framework that informs the design of all computer architectures, and the microchip revolution. We will then show how the chips built in weeks 1-3 can be integrated into a computer platform capable of running programs written in the instruction set introduced in week 4. In project 5 you will build this computer, known as *Hack*, and use it to run the programs you wrote in project 4.

6. **Assembler:** In 1843, Ada Lovelace showed how symbolic instructions can liberate programming from the obscure tyranny of binary codes. We will learn how to translate the symbolic instructions introduced in week 4 into the binary micro-codes understood by the computer built in week 5. In order to develop this translator, known as *assembler*, we will learn how to perform parsing, code generation, and symbol resolution. This will set the stage for project 6, in which you will build this assembler, using a high-level language of your choice.

Part II: Software

Using the computer built in Part I as a point of departure, we will build a multi-tier software hierarchy consisting of a virtual machine, a compiler for a simple java-like programming language, and a basic operating system. Acting as the systems architects, we'll provide elaborate implementation guidelines, scaffolding, and APIs.

7. **Virtual machine I:** Modern compilers typically translate high-level programs into an intermediate code designed to run on an abstract virtual machine. Following a discussion of pushdown automata and stack processing, we'll discuss the role of virtual machines in software architectures like Java, Python, and .NET. We will then present a simple VM language, modeled after Java's JVM. In project 7 you will write a JRE-like program that translates the push/pop and arithmetic VM commands into assembly code, designed to run on the Hack computer built in part I of the course.

8. **Virtual machine II:** We'll discuss two critically important programming abstractions – *branching* and *subroutines* – and show how they can be realized on our virtual machine. In particular, we'll discuss algorithms for realizing *goto* and *function-call-and-return* commands on a global stack. In project 8 you will extend the basic translator built in project 7 into a full-blown VM translator. This translator will become the backend of compilers that translate high-level programs into VM code (similar to Bytecode).

9. **High Level Language:** We'll introduce *Jack* – a simple, high-level, object-based, java-like language. We'll discuss language design issues and trade-offs, and how Jack can be realized over the Hack platform. In project 9 you will select a computer game idea, and implement it in Jack. It will be thrilling to see this game running on the computer that you've built in Part I of the course. You will develop this program using executable versions of the compiler and OS that we now turn to describe.

10. **Compiler I:** We'll discuss context-free grammars and recursive parsing algorithms, and guide how they can be used for building a syntax analyzer (tokenizer and parser) for the Jack language presented in week 9. In project 10 you will implement this analyzer, using a proposed software architecture and API.

11. **Compiler II:** We'll discuss how to realize high-level programming abstractions (classes, methods, statements, expressions, objects, etc.) by generating VM code for the virtual machine built in weeks 7-8. In project 11 you will morph the syntax analyzer built in week 10 into a full-scale compiler that translates Jack programs into VM code.

12. **Operating system:** The OS is designed to close gaps between Jack and Hack. We'll discuss classical OS algorithms for managing memory, realizing mathematical operations, rendering graphics, handling strings, and more – and a bootstrapping technique for realizing these services efficiently on the Hack platform. In project 12 you will implement this OS in Jack, in the spirit of how Unix was implemented in C.

13. **More fun to go:** We'll discuss ideas for extending and optimizing the hardware and software systems built in the course. For example, how to implement a file system and an OS shell, and how to connect the Hack computer to the Internet. We'll also describe how to build the computer in silicon, using FPGA. These are some of our design itches; What are yours?