Course program and reading list

Semester 1 Year 2024

School: Efi Arazi School of Computer Science B.Sc

Digital Systems

Lecturer:

Prof. Shimon Schocken schocken@runi.ac.il

Teaching Assistant:

Mr. Yitzchak Vaknin yitzchak.vaknin@post.runi.ac.il

Course No.:	**Course Type :**	**Weekly Hours :**	**Credit:**
3955	Elective	3	4

Course Requirements :	**Group Code :**	**Language:**
Final Paper	241395501	English

Prerequisites

Prerequisite:

56 - Discrete Mathematics
417 - Introduction To Computer Science

Students who took one of the courses listed below will not be allowed to register to the course Digital Systems (3955):

79 - Digital Architectures
287 - Digital Systems Construction

Course Description

Digital Systems / Nand to Tetris:

Building a Modern Computer System from First Principles

Instructor: Shimon Schocken

Fall 2023

The game of the name: The elementary logic gate *Nand* (or its close relative *Nor*) is the basic building block from which all computers are made. In this course we start from the humble Nand gate and work our way upward 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 carry out system building projects using supplied API's and guided unit-testing.

Prerequisites: Introduction to Computer Science. All the knowledge required for building the computer and completing the assignments will be gained in the course.

Description: Nand to Tetris is a journey of discovery, synthesizing key topics in applied computer science in one unified, hands-on framework. This is done constructively, by building a general-purpose computer system from the ground up. We will utilize key ideas and techniques used in the design of modern hardware and software systems, and discuss major trade-offs and future trends. As the course progresses, you will gain many cross-section views of the computing field, from gate logic to chip design to the compilation of high-level software abstractions. We will also provide a historical perspective, focusing on key people, circumstances and innovations that paved the way to the digital computer.

Lectures: One class meeting, each week. Regular track: Sundays, 09:45 to 12:15. International track: Mondays, 18:30 to 21:00. In both tracks, the language of instruction is English.

Methodology: This is mostly a hands-on course. Each hardware and software module will be introduced by an abstract specification and an executable solution, illustrating *what* the module is designed to do. This will be followed by 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 (HDL), learned in the course. You will simulate and test your HDL-based chips on a supplied hardware simulator running on your PC – just like chips are designed in practice. The software hierarchy (assembler, virtual machine, compiler) will be developed in Java. Students who want to do so can use other languages like Python or Perl.

Testing: The hardware and software modules built in the course will be tested twice. First, you will test your solutions on your PC, using supplied simulators and test scripts. Second, the course staff will test your work, using test scripts that may add additional testing scenarios.

Resources: All course materials – lecture notes, simulators, software tools, tutorials and test programs – will be available for download / access from the course web site.

Workload: All the assignments can be done by pairs of students. We provide extensive support, including supplied API's, guided unit-testing, and partial implementations. As a result, the course workload is quite average.

Course Grade: 60% homework assignment grades, 40% final examination grade.

Books (recommended / optional reading):

Nisan and Schocken, *The Elements of Computing Systems*, MIT Press, 2021, 2nd Edition

Isaacson, *The Innovators*, Simon & Schuster, 2015

Patterson and Hennessy, Computer Organization and Design RISC-V Edition, Morgan Kauffman, 2020 (or latest edition).

Course Plan (by week)

Part I: Hardware

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

October 22 / Boolean logic: We'll present 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 assignment 1 you will use this HDL to build a set of elementary logic gates (And, Or, Not, Mux,...) from primitive Nand gates.

October 22 / 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 assignment 2 you will build this ALU, using the logic gates built in assignment 1.

October 29 / 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 assignment 3 you will build a memory hierarchy, from single-bit cells to registers and RAM units of arbitrary sizes.

November 5 / Machine language: This is the critical interface where high-level software is ultimately reduced to binary codes committed to silicon. We'll introduce an instruction set and an abstract computer architecture, and learn how to write low-level programs on this platform. We'll also learn how to handle input / output devices like screen and keyboard, using memory-resident bitmaps. In assignment 4 you will write low-level interactive programs in assembly language and execute them on a supplied CPU emulator, running on your PC.

November 12 / Computer architecture: We'll describe the *stored program* concept, the *fetch-execute* cycle, and a ubiquitous computing framework that informs the design of all computer architectures. We will also discuss the microchip revolution. We will then show how the chips built in weeks 1-3 can be integrated into a computer platform capable of executing programs written in the instruction set introduced in week 4. In assignment 5 you will build this computer, and use it to run the programs you wrote in assignment 4.

November 19 / Assembler: In 1843, Ada Lovelace showed how symbolic instructions can liberate programming from the obscure tyranny of physical instructions. 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 basic techniques for parsing, code generation, and symbol resolution. This will set the stage for assignment 6, in which you will write this assembler in Java.

Part II: Software

Using the computer built in Part I as a point of departure, we will build a tiered 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 all the necessary API's, implementation guidelines, and test programs.

November 26 / Virtual machine I: Modern compilers typically translate high-level programs into an intermediate "bytecode", 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 JVM-like language that provides push/pop and stack arithmetic services. In assignment 7 you'll develop a basic VM translator that translates commands written in this language into assembly code, designed to run on the computer built in part I of the course.

December 3 / Virtual machine II: We'll discuss two fundamental programming artifacts – *branching* and *subroutines* – and show how they can be realized on our virtual machine. In particular, we'll discuss algorithms for stack-based implementation of *function-call-and-return*, and *recursion*. In assignment 8 you will extend the basic translator built in assignment 7 into a complete VM translator. This translator will serve as the backend of the compiler that will be built later in the course.

December 17 / High Level Language: We'll present the top-most layer of our software hierarchy: A simple, object-based, java-like language. In assignment 9 you will use this language to implement a simple computer game of your choice. 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.

December 24 / Compiler I: We'll discuss context-free grammars and recursive parsing algorithms, and show how they can be used for 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. In assignment 10 you will implement this analyzer, using a proposed software architecture and API.

December 31 / Compiler II: We'll discuss how to realize high-level programming abstractions (classes, methods, statements, expressions, objects, etc.) by generating VM code for our virtual machine. In assignment 11 you will morph the syntax analyzer built in week 10 into a full-scale compiler. This will be done by replacing the routines that wrote passive XML with routines that generate executable VM code for the stack machine developed in weeks 7-8.

January 7 / Operating system: The OS is designed to close gaps between the software hierarchy built in Part II and the computer platform built in Part I. We'll discuss space/time trade-offs, and present classical algorithms for managing memory, realizing mathematical operations, rendering graphics, handling strings, and more – efficiently and elegantly. In assignment 12 you will implement these algorithms, leading up to a basic operating system.

January 14 / More fun to go: We'll discuss ideas for optimizing and extending the hardware and software systems built in the course. For example, how to speed up the CPU operations, how to connect the computer to the Internet, how to implement a file system and an OS shell, and so on. We'll also illustrate how to realize the computer built in this course in silicon, using an important hardware technology known as FPGA. These are some of our design itches; What are yours?

Course Goals

See "course description".

Grading

See "course description".

Reading List

See "course description".