Nicholas Hunt

Independent Study Report

Elements of Computing Systems

Independent Study - Final Report

The Elements of Computing Systems is a thorough and easily digestible book on what it takes for a computer to recognize a set of instructions. Machine language, assembly, and high-level languages, are all covered within this book. By going through and completing the projects in each chapter, I have come to learn what it means to compile, translate, and assemble code. Other courses, such as Mechanics of Programming (MOP) and Concepts of Computing Systems (CCS), do a great job explaining the semantics of these operations. But this book forces you to complete these tasks through practice. There was no point where I felt lost in the book; it is thorough and explains every concept to its fullest. With that said, there are some things I would change before adapting this book to a graduate-level course. Now I am left with the big question that prof. Heliotis and Strout have also been asking all semester. Would this make for a suitable graduate-level course?

I will start with what I felt weird about/did not like. This book goes well into hardware, and you will learn how to build the several types of logic gates that come into play within the hardware. However, I think it could do more. The CPU we construct is minimalistic and cannot multiply or divide, which is a missed opportunity, especially when the CPU developed in most CCS courses includes this arithmetic in the ALU. Furthermore, the book does not touch on floating-point numbers within the hardware.

It gets around this by implementing a fraction library within the OS. It takes in a numerator and denominator within its constructor. Having multiplication, division, and decimal numbers represented this high up in a language is not optimal for performance and not practical in the real world. If I were to present this as a viable course, we would have to address these issues.

I suggest Intel's 8086 or 80186 CPUs, as they are old enough to be a bit tame and young enough to contain multiplication and division. Updating the CPU would also mean updating the book's Assembly instruction (It would just be a portion on the x86 language rather than the Hack language). This change would do the hardware portion of this class justice. I completed the hardware portion in six weeks, and these additions would take longer to go over. But students will have an extra week or two as we started this independent study a bit late. But of course, you could also add these instructions to the Hack language as it is already a made-up language!

Lastly, the provided VMEmulator and CPUEmilator simulators are very slow. It often took over ten minutes to run Jack programs with over 50 lines. If anything, we would need to rework the specified simulators. I tried finding the source code, but nothing was available. They are just simple single pane Swing GUIs, so it shouldn't take over a week.

As for what I found good, I feel confident with my understanding of compiler design and plan on taking more classes within the area. Elements from CS Theory, such as language recognition, came into play as I had to develop a syntax analyzer for the compiler portion of this course. This discovery was exciting when it all clicked. This book starts at a breathable pace. It also does not leave much up to interpretation. It thoroughly lays out each project. I understand why some CS courses stay away from this, but I would prefer a thorough write-up rather than being left with five or more questions I need to ask for clarification. Clarity is especially crucial when your professor did not write the write-up.

This book and its weekly projects always left me in awe as I saw everything connecting. Nothing should change within the first and third chapters and the software section (chapters 6-12). Other chapters contain information on the ALU, machine code, and assembly language specification (discussed above)! The software portion is easy to follow but very hard to implement. On average, I spent 11 hours writing, debugging, and commenting on the code for each software project. To put this in perspective, projects in other CS courses that are only assigned twice a semester take roughly the same time, if not a little more. These were projects that were due weekly. It is definitely up to standard - in my humble opinion - for what RIT expects from a graduate-level course.

The book also suggested using C or Java to develop the software portion of the projects, of which I chose Java. C would be interesting, but it would take way too long to complete each assignment. The option is nice, but I also think worrying about the semantics of C would add stress to the already stressful weekly assignments. In a sense it doesn't matter what language you use. As long as you generate the correct .hack, .jack, .asm, and .vm files, you are in the clear!

I also do not think this course needs a midterm or weekly tests as the book bases learning around the 12 projects. If you would like to incorporate a midterm, I suggest just making the Project 6 on CPU design the midterm. This project is meant to be one of the longer projects, occurs around the middle of the term, and incorporates the hardware designed throughout the entirety of the hardware portion. This midterm will be the real test of "Does this student understand the hardware portion of this course?"

Furthermore, if you need a final exam, the software components (Compiler, VMtranslator, and Assembler) can all be put together to translate Jack Code down to 16-bit instructions to run on the CPU from the midterm. This exam is the final test of whether the student has completed everything correctly. Whether or not you would like to use the student's CPU or one guaranteed to work for testing the machine code is up to the professor and administration. Maybe students can earn extra credit if they use their CPU for testing the code. The options are plentiful.

After the revisions I proposed - I know this would be a reputable course for the graduate level. I think it would best fit into the category of single system design. It covers everything from computer architecture, language analysis, compilation and assembly of code, and a simple OS design. It could be a course students take before CSCI 742 Compiler Construction. Or maybe an architecture and operating systems (AOS) cluster; there is currently only one cluster course for AOS at the graduate level! I will leave it up to the admins of CS to finalize where it should go, but I figured those were some suitable suggestions.

Over the course of this semester, I took around three hours a week reading through each chapter, taking detailed notes. On top of that there were weekly projects. Hardware projects ranged from three to seven hours and 100 to 300 lines, and the software projects ranged from six to ten hours and 500 to 1500 lines. Notes range from four to ten pages. I estimate I spent around seven to fourteen hours a week working towards completing these assignments.

If this independent does develop into a graduate course, I would be happy to assist in any way possible! This independent study has been an impactful experience, and I would love to be a part of its evolution. I will be on Co-op in Ohio during the summer and fall, so I should have some free time if my assistance is needed.

If I had to create a Mock Weekly schedule for the 15-week course it would be something like this:

Week #	Topics	Assignments Given	Events and Assignments Due
Week 1	Boolean Logic: Composition of Logic gates	Project 1: Logic Gates from Nand Gate	
Week 2	Boolean Arithmetic I: Composition of ALU without multiplication/division	Project 2: ALU design within HDL (without mult and div).	Project 1: Due
Week 3	Boolean Arithmetic II: Composition of Full ALU	Project 3: ALU design within HDL (complete)	Project 2: Due
Week 4	Memory: Memory devices and combination and sequential logic	Project 4: Memory and logic device design within HDL.	Project 3: Due
Week 5	Machine Learning: Going over the Assembly language of the processor chosen.	Project 5: Complete the selected assembly programs.	Project 4: Due
Week 6	Computer Architecture: Data busses	Project 6: Build your CPU within HDL language using your created ALU and memory devices.	Project 5: Due
Week 7	Computer Architecture II: Finish the processor described.		
Week 8	Assembler: How to implement an assembler	Project 7: Construct an assembler using Java or C++.	Project 6: Due
Week 9	Virtual Machine I: How to implement a virtual machine for the sake of easier compilation.	Project 8: Construct Push, Pop, and arithmetic VM translator instructions.	Project 7: Due
Week 10	Virtual Machine II: VM instructions such as function, goto, if-goto, call, etc.	Project 9: Complete VM Translator construction.	Project 8: Due
Week 11	High-Level Language: The Jack language.	Project 10: Complete the selected Jack language programs.	Project 9: Due
Week 12	Compiler I: Syntax Analysis: building a syntax Analyzer to recognize or reject a language.	Project 11: Complete the syntax analyzer within Java or C++.	Project 10: Due
Week 13	Compiler II: Code Generation: Code Compilation, writing VM instruction.	Project 12: Complete the Compiler within Java or C++.	Project 11: Due
Week 14	Operating System: Learn what an OS library needs to operate.	Project 13: Build the OS library within the Jack language, compile it into VM code.	Project 12: Due
Week 15	Finals Week		Project 13: Due

Bibliography

Nisan, Noam, and Shimon Schocken. The Elements of Computing Systems, Second Edition. Second, MIT Press, 2021.

Appendix 1

Hardware:

Project 01: Boolean Logic

This project had me implement all of the logic gates required, such as and, or, xor, not, mux, demux, and their 16-bit equivalents. Due to starting this independent study a little later than anticipated. Some of the gates, such as Mux8Way16, were left unfinished as they were similar to the 1-bit multiplexer. We did so using a hardware description language (HDL), which is a language used by hardware designers to test hardware through software.

Project 02: Boolean Arithmetic

This project has me implement an incrementor, adder, 16-bit adder, and the ALU, which we would implement into our CPU in later projects. We did so using HDL.

Project 03: Memory

This project had us construct our memory devices and our program counter, all to be used in implementing our CPU in later projects. We did so using HDL.

Project 04: Machine Language

This project had us write a couple of assembly programs, multiply and fill. I completed multiply.asm program via iterative adding. I finished the fill.asm program through multiple labels and jump statements. The function of the fill.asm program is to turn the pixels of a screen black when you press a key on the keyboard. After releasing the key, the screen would restore itself to white. We did so using the Hack language described in the chapter.

Project 05: Computer Architecture

This project was to implement the hardware from previous chapters and complete the CPU design. We did so using HDL.

Project 06: Assembly

This project was to build an assembler for the Hack language and then run our assembled code through the CPU we created in the previous project. Project 06 was the first project I had to use Java for, and the rest of the software projects are either in Java or the language the book creators made, Jack.

Appendix 2

Software:

Project 07: Virtual Machine I: Processing

This project was an introduction to a stack machine. We used this abstract machine to ease the compilation process. Without this we would have to translate Jack code straight into Assembly code. Which is both a difficult task and not as portable, as you would have to write a whole compiler for every assembly language, we compile Jack code. We only implemented the push, pop, and arithmetic operations in project 07.

Project 08: Virtual Machine II: Control

This project was a continuation of the previous. We furthered the design of our abstract stack machine to handle function declaration, function calls, variable labels, goto and if-goto (unconditional and conditional) jumps, return statements.

Project 09: High-Level Language

This project had us complete the set of programs they gave us or come up with one ourselves. I chose the ladder as I felt it would be interesting. I decided to implement a queue abstract data type (queueADT) as the Jack language is very bare bones and only has character, integer, and boolean primitive types. This queueADT uses two arrays. One that keeps track of the data value, and the other keeps track of the data types at each index. This is needed as the jack language doesn't differentiate between char, integer, and boolean very easily. You have to keep track of what each type a variable is. For example, you can print a boolean out as an integer, a character as a boolean, or an integer as a character. It can do this because there is no type checking within the Jack language. I thought it was a creative way to get around the issue. It ended up working well.

Project 10: Compiler I: Syntax Analysis

This project had us construct a way of recognizing the Jack language. It did so by have us implement a push down automata (PDA). A PDA is a machine that recognizes if a String, or list of tokens, is in a language. The string/list in question is a jack program. We first broke up the Jack program into a list of tokens. Then we had to process each token and verify that the syntax of the Jack file was correct. If the syntax is wrong, the analyzer would stop and throw an error. The error message denoted what was wrong to help debugging. I wrote this program once again in Java. The output of this would be an XML file of the syntax.

Project 11: Compiler II: Code Generation

This project had us finish out compiler by generating VM instructions for the written jack code. We used the code from the previous section and replaced all of the XML output for VM output. Project 11 took me the longest at twelve hours to complete and get all the vm instructions working. Errors were particularly hard to debug as I didn't have any vm instructions to compare it. I only knew the program wasn't working, and roughly the position I was mistranslating code.

Project 12: Operating System

This project had us write jack code for Array.jack, Keyboards.jack, Math.jack, Memory.jack, Output.jack, Screen.jack, String.jack, and Sys.jack OS library programs. These are the general classes that the Jack OS. Project 12 was used to showcase the basic libraries you will need to develop to construct an OS.