DOI:10.1145/3626513

CS course walks students through a step-by-step construction of a complete, general-purpose computer system—hardware and software—in one semester.

BY SHIMON SCHOCKEN

Nand to Tetris: Building a Modern Computer System from First Principles

SUPPOSE YOU WERE asked to design an abridged computer science (CS) program consisting of just three courses. How would you go about it? The first course would probably be an introduction to computer science, exposing students to computational thinking and equipping them with basic programming skills. The second course would most likely be algorithms and data structures. But what should the third course be?

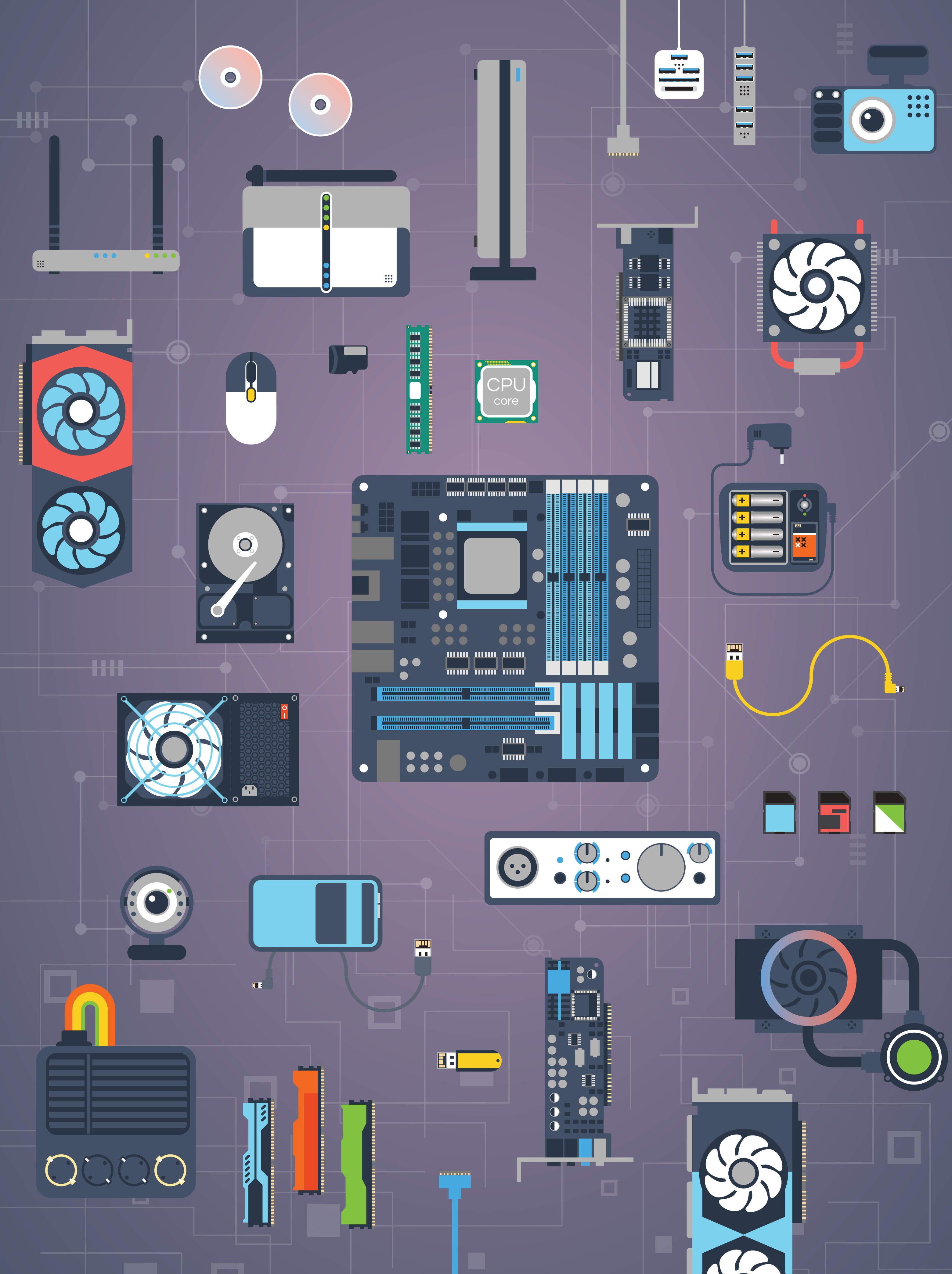
Several reasonable options come to mind. One is a hands-on overview of applied CS, building on the programming skills and theoretical knowledge acquired in the first two courses. Such a course could survey key topics in computer architecture, compilation, operating systems, and software engineering, presented in one cohesive framework. Ideally, the course would engage students in significant programming assignments, have them implement classical algorithms and widely used data structures, and expose them to a range of optimization and complexity issues. This hands-on synthesis could benefit students who seek an overarching understanding of computing systems, as well as self-learners and non-majors who cannot commit to more than a few core CS courses.

This article describes such a course, called *Nand to Tetris*, which walks students through a step-by-step construction of a complete, general-purpose computer system—hardware and software—in one semester. As it turns out, construction of the computer system built during the course requires exposure to, and application of, some of the most per-

» key insights

- **In the early days of computers, any curious person could gain a gestalt understanding of how the machine works. As digital technologies became increasingly more complex, this clarity is all but lost: The most fundamental ideas and techniques in applied computer science are now hidden under many layers of obscure interfaces and proprietary implementations.**
- **Starting from NAND gates only, students build a hardware platform comprising a CPU, RAM, datapath, and a software hierarchy consisting of an assembler, a virtual machine, a basic OS, and a compiler for a simple, Java-like object-based language.**
- **The result is a synthesis course that combines key topics from traditional systems courses in one hands-on framework. The course is self-contained, the only prerequisite being introduction to computer science.**

IMAGERY BY DEAD SAKURA


CPU
core

tinent and beautiful ideas and techniques in applied CS. The computer's hardware platform (CPU, RAM, datapath) is built using a simple hardware description language and a supplied hardware simulator. The computer's software hierarchy (assembler, virtual machine, compiler) can be built in any programming language, following supplied specifications. The resulting computer is equipped with a simple Java-like, object-based language that lends itself well to interactive applications using graphics and animation. Thousands of computer games have already been developed on this computer, and many of them are illustrated in YouTube.

Following early versions of the course[6] that underwent many improvements and extensions, the complete Nand to Tetris approach was described in the book *The Elements of Computing Systems*, by Noam Nisan and Shimon Schocken.[3] By choosing a book title that nods to Strunk and White's masterpiece,[9] we sought to allude to the concise and principled nature of our approach. All course materials—lectures, projects, specifications, and software tools—are freely available in open source.[5] Versions of the course are now offered in many educational settings, including academic departments, high schools, bootcamps, and online platforms. A typical course syllabus is available.[2] About half of the course's online learners are developers who wish to acquire a deep, hands-on understanding of the hardware and software infrastructures that enable their work. And the best way to understand something deeply is to build it from the ground up.

From Nand to Tetris

The explicit goal of Nand to Tetris courses is to build a general-purpose computer system from elementary logic gates. The implicit goals are to offer a hands-on exposition of key concepts and techniques in applied computer science, and a compelling synthesis of core topics from digital architectures, compilation, operating systems, and software engineering. We make this synthesis concrete by walking students through 12 hands-on projects. Each project presents and motivates an important hardware or software abstraction, and then it provides guidelines for implementing the abstraction using executable modules developed in previous, lower-level projects. The computer system that emerges from this effort is built gradually and from the bottom up (see Figure 1).

The first five projects in the course focus on constructing the chipset and architecture of a simple von Neumann computer. The remaining seven projects revolve around the design and implementation of a typical software hierarchy. In particular, we motivate and build an assembler; a virtual machine; a two-tier compiler for a high-level, object-based programming language; a basic operating system (OS), and an application—typically a simple computer game involving animation and interaction. The overall course consists of two parts, as we now turn to describe.

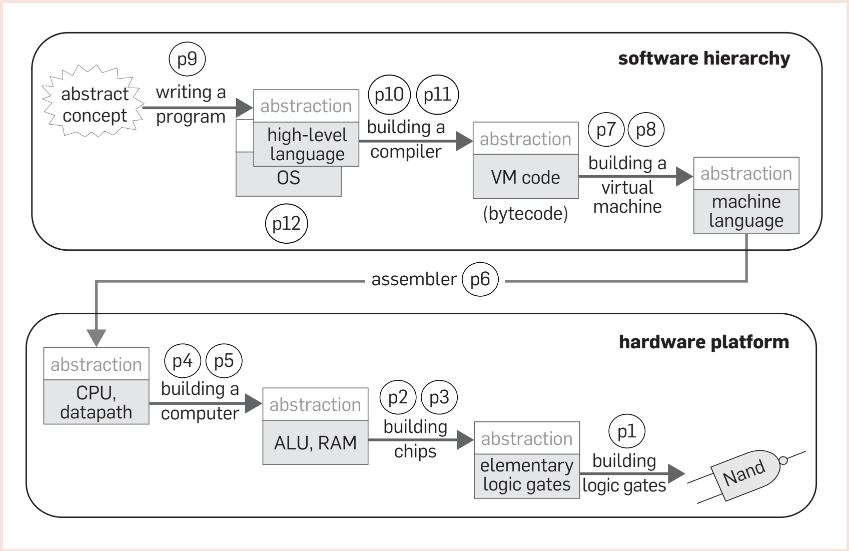
Part I: Hardware. The course starts with a focused overview of Boolean algebra. We use disjunctive normal forms and reductive reasoning to show that every Boolean function can be realized using no more than NAND operators. This provides a theoretical yet impractical demonstration that the course goal—building a general-purpose computer system from just NAND gates—is indeed feasible. We then discuss gate logic and chip specification, and present a simple hardware description language (HDL) that can be learned in a few hours.

In project 1, students use this HDL and a supplied hardware simulator to build and unit-test elementary logic gates such as AND, OR, NOT, multiplexors, and their 16-bit extensions. We then discuss Boolean arithmetic, two's complement, and arithmetic-logic operations. This background sets the stage for project 2, in which students use the elementary gates built in project 1 to implement a family of combinational chips, leading up to an ALU. We then discuss how sequential logic and finite-state automata can be used to implement chips that maintain state. In project 3, students apply this knowledge to gradually build and unit-test 1bit and 16bit registers, as well as a family of direct-access memory units (in which addressing and storage are realized using combinational and sequential logic, respectively), leading up to a RAM.

At this stage, we have all the basic building blocks necessary for synthesizing a simple 16bit von Neumann machine, which we call "Hack." Before doing so, we present the instruction set of this target computer (viewed abstractly), in both its symbolic and binary versions. In project 4, students use the symbolic Hack machine language to write assembly

Figure 1. Overall course plan.

Each project p1, p2, ..., p12 lasts one to two course-weeks. Project numbers indicate the normal sequence, although they can be done in any desired order.



programs that perform basic algebraic, graphical, and user-interaction tasks (the Hack computer specification includes input and output drivers that use memory bitmaps for rendering pixels on a connected screen and for reading 16bit character codes from a connected keyboard). The students test and execute their assembly programs on a supplied emulator that simulates the Hack computer along with its screen and keyboard devices.

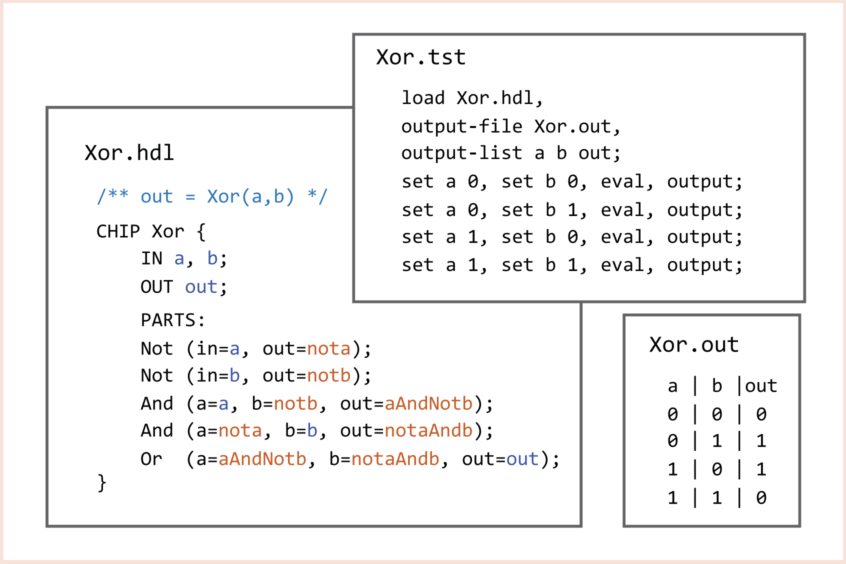
Next, we present possible skeletal architectures of the Hack CPU and datapath. This is done abstractly, by discussing how an architecture can be functionally planned to fetch, decode, and execute binary instructions written in the Hack instruction set. We then discuss how the ALU, registers, and RAM chips built in projects 1–3 can be integrated into a hardware platform that realizes the Hack computer specification and machine language. Construction of this topmost computer-on-a-chip is completed in project 5.

Altogether, in Part I of the course, students build 35 combinational and sequential chips, which are developed in an HDL and tested on a supplied hardware simulator. For each chip, we provide a skeletal HDL program (listing the chip name, I/O pin names, and functional documentation comprising the chip API); a test script, which is a sequence of set/eval/compare steps that walk the chip simulation through representative test cases; and a compare file, listing the outputs that a correctly implemented chip should generate when tested on the supplied test script (see Figure 2). For each chip developed in the course, the contract is identical: Complete the given HDL skeletal program and test it on the hardware simulator using the supplied test script. If the outputs generated by your chip implementation are not identical to the supplied compare file, keep working; otherwise, your chip behaves to specification, but perhaps you want to optimize it for efficiency. The chip logic is evaluated and tested on our hardware simulator (see Figure 3).

Part II: Software. The barebones computer that emerges from Part I of the course can be viewed as an abstraction that has a well-defined interface: the Hack instruction set. Using this machine language as a point of departure, in Part II of the course we construct a software hierarchy that empowers the Hack computer to execute code written in high-level programming languages. This effort entails six projects that build a compiler and a basic operating system on top of the hardware platform built in Part I. Specifically, we implement a simple object-based, Java-like language called "Jack." We start this journey by introducing the Jack language and the OS (abstractly) and discussing the trade-offs of one-tier and

Figure 2. The specification of each chip consists of a stub HDL file (containing the chip signature and an empty PARTS section), a test script, and a compare file.

When evaluated by the hardware simulator, the output file produced by a correctly implemented HDL program should be identical to the given compare file.



Xor.hdl

```
/** out = Xor(a,b) */
CHIP Xor {
    IN a, b;
    OUT out;
    PARTS:
    Not (in=a, out=nota);
    Not (in=b, out=notb);
    And (a=a, b=notb, out=aAndNotb);
    And (a=nota, b=b, out=notaAndb);
    Or  (a=aAndNotb, b=notaAndb, out=out);
}
```

Xor.tst

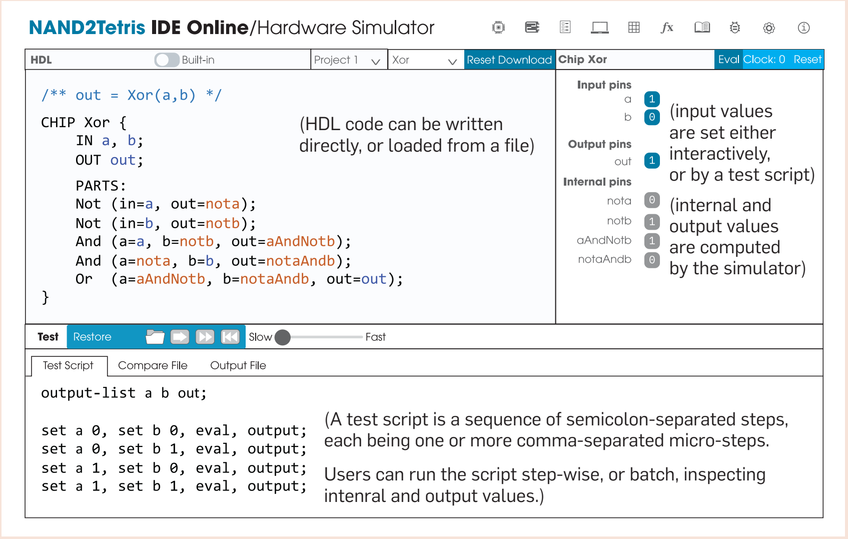
```
load Xor.hdl,
output-file Xor.out,
output-list a b out;
set a 0, set b 0, eval, output;
set a 0, set b 1, eval, output;
set a 1, set b 0, eval, output;
set a 1, set b 1, eval, output;
```

Xor.out

```
a | b |out
0 | 0 | 0
0 | 1 | 1
1 | 0 | 1
1 | 1 | 0
```

Figure 3. The Hardware Simulator, running/evaluating the HDL program shown in Figure 2 (the roles of the various panels are explained in the text in parentheses).

This particular XOR implementation, which is readable but not necessarily efficient, is based on two NOT, two AND, and one OR chip parts, each implemented as a standalone HDL program. When evaluating a chip, the simulator evaluates recursively all its chip parts, all the way down to evaluating NAND gates, which have a primitive (built-in) implementation.



```
/** out = Xor(a,b) */

CHIP Xor {
    IN a, b;
    OUT out;

    PARTS:
    Not (in=a, out=nota);
    Not (in=b, out=notb);
    And (a=a, b=notb, out=aAndNotb);
    And (a=nota, b=b, out=notaAndb);
    Or  (a=aAndNotb, b=notaAndb, out=out);
}
```

(HDL code can be written directly, or loaded from a file)

Input pins	
a	1
b	0
Output pins	
out	1
Internal pins	
nota	0
notb	1
aAndNotb	1
notaAndb	0

(input values are set either interactively, or by a test script)

(internal and output values are computed by the simulator)

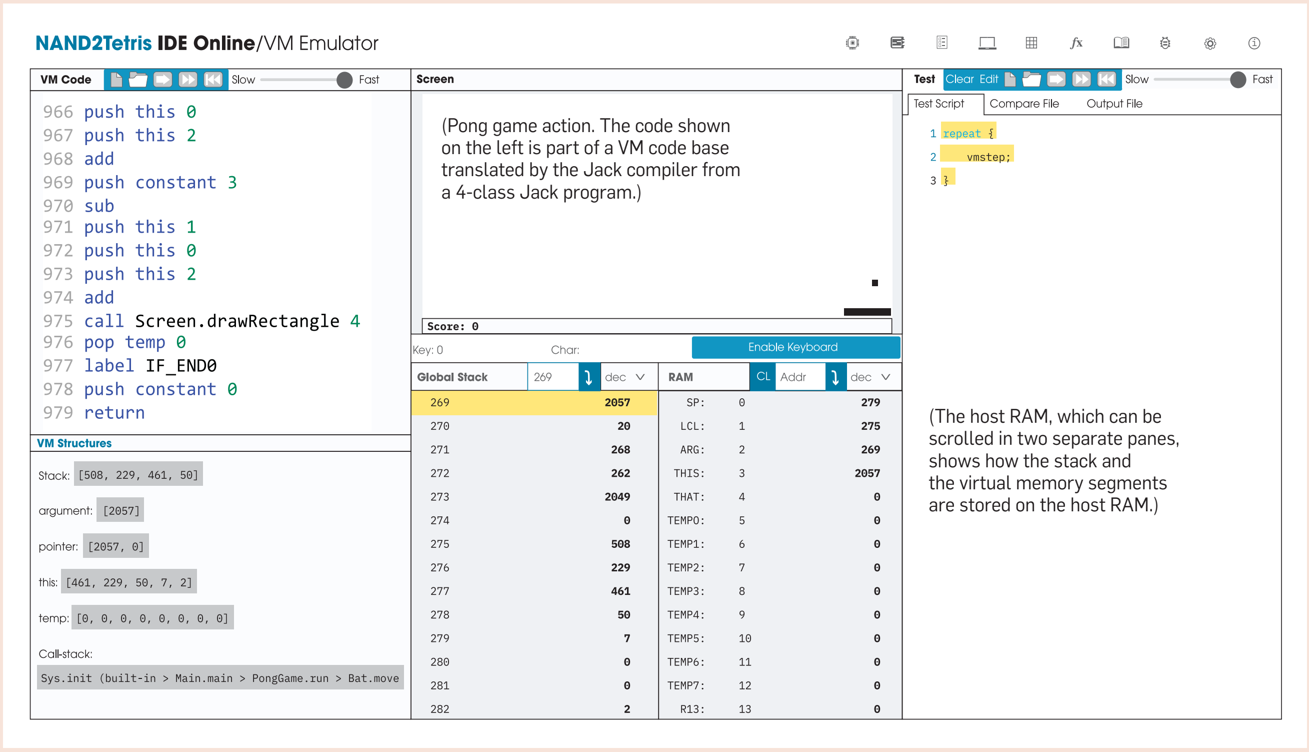
```
output-list a b out;

set a 0, set b 0, eval, output;
set a 0, set b 1, eval, output;
set a 1, set b 0, eval, output;
set a 1, set b 1, eval, output;
```

(A test script is a sequence of semicolon-separated steps, each being one or more comma-separated micro-steps.

Users can run the script step-wise, or batch, inspecting intenral and output values.)

Figure 4. A typical computer game, developed in Jack.

The compiled VM code is loaded into, and executed by, the VM Simulator shown here. The simulator displays the VM code, the simulated computer screen, the VM stack and the virtual memory segments, and the host RAM in which they are realized. For example, RAM[0] stores the stack pointer, RAM[1] stores the base address of the local variables segment, and so on.



```
push this 0
push this 2
add
push constant 3
sub
push this 1
push this 0
push this 2
add
call Screen.drawRectangle 4
pop temp 0
label IF_END0
push constant 0
return
```

(Pong game action. The code shown on the left is part of a VM code base translated by the Jack compiler from a 4-class Jack program.)

```
Score: 0
```

```
repeat {
    vmstep;
}
```

```
Stack: [508, 229, 461, 50]
argument: [2057]
pointer: [2057, 0]
this: [461, 229, 50, 7, 2]
temp: [0, 0, 0, 0, 0, 0, 0, 0]
Call-stack:
Sys.init (built-in > Main.main > PongGame.run > Bat.move
```

```
Global Stack 269          RAM
269    2057               SP:     0    279
270      20               LCL:    1    275
271     268               ARG:    2    269
272     262               THIS:   3   2057
273    2049               THAT:   4      0
274       0               TEMP0:  5      0
275     508               TEMP1:  6      0
276     229               TEMP2:  7      0
277     461               TEMP3:  8      0
278      50               TEMP4:  9      0
279       7               TEMP5: 10      0
280       0               TEMP6: 11      0
281       0               TEMP7: 12      0
282       2               R13:   13      0
```

(The host RAM, which can be scrolled in two separate panes, shows how the stack and the virtual memory segments are stored on the host RAM.)

two-tier compilation models. We also emphasize the role that intermediate bytecode plays in modern programming frameworks.

Following this general overview, we introduce a stack-based virtual machine and a VM language that features push/pop, stack arithmetic, branching, and function call-and-return primitives. This abstraction is realized in projects 7 and 8, in which students write a program that translates each VM command into a sequence of Hack instructions. This translator serves two purposes. First, it implements our virtual machine abstraction. Second, it functions as the back-end module of two-tier compilers. For example, instead of writing a monolithic compiler that translates Jack programs into the target machine language, one can write a simple and elegant front-end translator that parses Jack programs and generates intermediate VM code, just like Java and C# compilers do. Before developing such a compiler, we offer a complete specification of the Jack language, and illustrate Jack programs involving arrays, objects, list processing, recursion, graphics, and animation (the basic Jack language is augmented by a standard class library that extends it with string operations, I/O support, graphics rendering, memory management, and more). These OS services are used by Jack programs abstractly and implemented in the last project in the course. In project 9, students use Jack to build a simple computer game of their choosing. The purpose of this project is not learning Jack, but rather setting the stage for writing a Jack compiler and a Jack-based OS.

Development of the compiler spans two projects. We start with a general discussion of lexicons, grammars, parse trees, and recursive-descent parsing algorithms. We then present an XML mark-up representation designed to capture the syntax of Jack programs. In project 10, students build a program that parses Jack programs as input and generates their XML mark-up representations as output. An inspection of the resulting XML code allows verifying that the parser's logic can correctly tokenize and decode programs. Next, we discuss algorithms for translating parsed statements, expressions, objects, arrays, methods, and constructors into VM commands that realize the program's semantics on the virtual machine built in projects 7–8. In project 11, students apply these algorithms to morph the parser built in project 10 into a full-scale compiler. Specifically, we replace the logic that generated passive XML code with logic that generates executable VM code. The resulting code can be executed on the supplied VM emulator (see Figure 4) or translated further into machine language and executed on the hardware simulator.

The software hierarchy is summarized in Figure 5. The final task in the course is developing a basic operating system. The OS is minimal, lacking many typical services, such as process and file management. Rather, our OS serves two purposes. First, it extends the basic Jack language with added functionality, like mathematical and string operations. Second, the OS is designed to close gaps between the software hierarchy built in Part II and the hardware platform built in Part I. Examples include a heap-management system for storing and disposing arrays and objects, an input driver for reading characters and strings from the keyboard, and output drivers for rendering text and graphics on the screen. For each such OS service, we discuss its abstraction and API, as well as relevant algorithms and data structures for realizing them. For example, we use bitwise algorithms for efficient implementation of algebraic operations, first-fit/best-fit and linked list algorithms for memory management, and Bresenham's algorithm for drawing lines and circles. In project 12, students use these CS gems to develop the OS, using Jack and supplied API's. And with that, the Nand to Tetris journey comes to an end.

Discussion: Engineering

Abstraction-implementation. A hallmark of sound system engineering is separating the abstract specification of *what* a system does from the implementation details of *how* it does it. In Patterson and Hennessy's "Seven Great Ideas in Computer Architecture," abstraction is at the top of the list.[4] Likewise, Dijkstra describes abstraction as an essential mental tool in programming.[1] In Nand to Tetris, the discussion of every hardware or software module begins with an abstract specification of its intended functionality. This is followed by a proposed implementation plan that hints, in outline form, how the abstraction can be realized using abstract building blocks from the level below (see Figure 1). Here we mean "abstract" in a very concrete way: Before tasking students to develop a hardware or software module—any module—we guide them to experiment with a supplied executable solution that entails precisely what the module seeks to do.

These experiments are facilitated by the Nand to Tetris online IDE,[8] developed by David Souther and Neta London. This set of tools includes a hardware simulator, a CPU emulator, a Hack assembler, a Jack compiler, and a VM emulator/runtime system that implements our virtual machine and OS. Before implementing a chip, or, when teaching or learning its intended behavior, one can load a built-in chip implementation into the hardware simulator and experiment with it (we elaborate on this "behavioral simulation" practice later in this article). Before implementing the assembler, one can load assembly programs into the supplied assembler and visually inspect how symbolic instructions are translated into binary codes. Prior to implementing the Jack compiler, one can use the supplied compiler to translate representative Jack programs, inspect the compiled VM code, and observe its execution on the supplied VM emulator. And before implementing any OS function, one can call the function from a compiled Jack test program and investigate its input-output behavior.

The central role of abstraction is also inherent in all the project materials: One cannot start implementing a module before carefully studying its intended functionality. Every chip is specified abstractly by a stub HDL file containing the chip signature and documentation, a test script, and a compare file. Every software module—for example, the assembler's symbol table or the compiler's parser—is specified by an API that documents the module along with staged test programs and compare files. These specifications leave no room for design uncertainty: Before setting out to implement a module, students have an exact, hands-on understanding of its intended functionality.

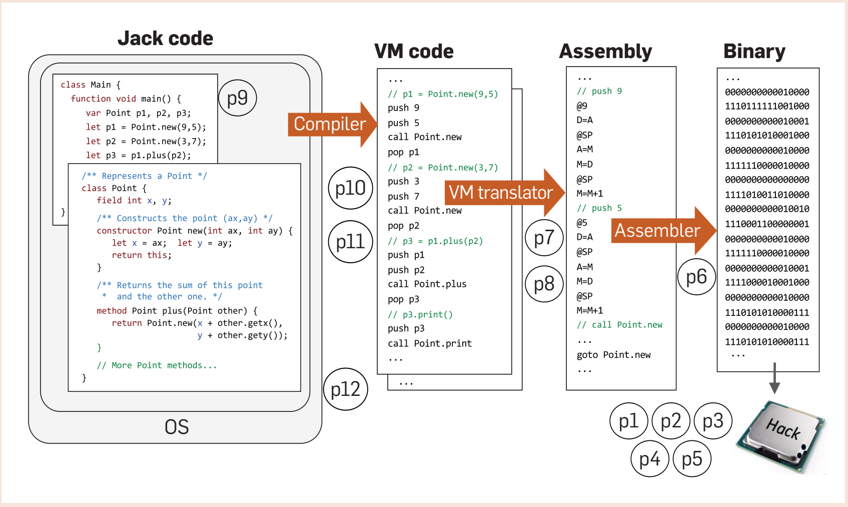
The ability to experiment with executable solutions has subtle educational virtues. In addition to actively understanding the abstraction—a rich world in itself—students are encouraged to discuss and question the merits and limitations of the abstraction's design. We describe these explorations in the last section of this article, where we discuss the course's pedagogy.

Modularity. A system architecture is said to be modular when it consists of (recursively) a set of relatively small and standalone modules, so that each module can be independently developed and unit tested. Like abstraction, modularity is a key element of sound system engineering: The ability to work on each module in isolation, and often in parallel, allows developers to compartmentalize and manage complexity.

The computer system built in Nand to Tetris courses comprises many hardware and software modules. Each module is accompanied by an abstract specification and a proposed architecture that outlines how it can be built from lower-level modules. In-

Figure 5. The software hierarchy built in Part II of the course (projects 7–12).

A Jack program, consisting of one or more class files, and the OS (implemented as a library of Jack classes) are compiled into a set of VM files. The VM files are compiled further into assembly code, which is translated by the assembler into binary code. The target code can be executed by the computer built in Part I of the course (projects 1–5).



```
class Main {
  function void main() {
     var Point p1, p2, p3;
     let p1 = Point.new(9,5);
     let p2 = Point.new(3,7);
     let p3 = p1.plus(p2);
}
```

```
/** Represents a Point */
class Point {
   field int x, y;

   /** Constructs the point (ax,ay) */
   constructor Point new(int ax, int ay) {
      let x = ax;  let y = ay;
      return this;
   }

   /** Returns the sum of this point
    *  and the other one. */
   method Point plus(Point other) {
      return Point.new(x + other.getx(),
                       y + other.gety());
   }

   // More Point methods...
}
```

```
...
// p1 = Point.new(9,5)
push 9
push 5
call Point.new
pop p1
// p2 = Point.new(3,7)
push 3
push 7
call Point.new
pop p2
// p3 = p1.plus(p2)
push p1
push p2
call Point.plus
pop p3
// p3.print()
push p3
call Point.print
...
...
```

```
...
// push 9
@9
D=A
@SP
A=M
M=D
@SP
M=M+1
// push 5
@5
D=A
@SP
A=M
M=D
@SP
M=M+1
// call Point.new
...
goto Point.new
...
```

```
...
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
1111000010001000
0000000000010000
1110101010000111
0000000000010000
1110101010000111
 ...
```

dividual modules are relatively small, so developing each one is a manageable and self-contained activity. Specifically, the HDL construction of a typical chip in Part I of the course includes an average of seven lower-level chip-parts, and the proposed API of a typical software module in Part II of the course consists of an average of ten methods.

This modularity impacts the project work as well as the learning experience. For example, in project 2, students build several chips that carry out Boolean arithmetic, including a "Half Adder." Given two input bits *x* and *y*, the half adder computes a two-bit output consisting of the "sum bit" and the "carry bit" of $x + y$. As it turns out, these bits can be computed, respectively, by AND-ing and XOR-ing *x* and *y*. But what if, for some reason, the student did not implement the requisite AND or XOR chip-parts in the previous project? Or, for that matter, the instructor has chosen to skip this part of the course? Blissfully, it does not matter, as we now turn to explain.

Behavioral simulation. When our hardware simulator evaluates a program like HalfAdder.hdl that uses lower-level chip-parts, the simulator proceeds as follows: If the *chipPart*.hdl file (like And.hdl and Xor.hdl) exists in the project directory, the simulator recurses to parse and evaluate these lower-level HDL programs, all the way down to the terminal Nand.hdl leaves, which have a primitive/built-in implementation. If, however, a *chipPart*.hdl file is missing in the project directory, the simulator invokes and evaluates a built-in chip implementation instead. This contract implies that all the chips in the course can be implemented in any desired order, and failure to implement a chip does not prevent the implementation of other chips that depend on it.

Using another example, a 1bit register can be realized using a data flip-flop and a multiplexor. Implementing a flip-flop gate is an intricate art, and instructors may wish to use it abstractly. With that in mind, HDL programs that use DFF chip-parts can be implemented as is, without requiring students to implement a DFF.hdl program first. The built-in chip library, which is part of our open source hardware simulator, includes Java implementations of all the chips built in the course. Instructors who wish to modify or extend the Hack computer or build other hardware platforms can edit the existing built-in chips library or create new libraries.

Behavioral simulation plays a prominent role in the software projects as well. For example, when developing the Jack compiler, there is no need to worry about how the resulting VM code is executed: The supplied VM emulator can be used to test the code's correctness. And when writing the native VM implementation, there is no need to worry about the execution of the resulting assembly code, since the latter can be loaded into, and executed, on the supplied CPU emulator. In general, although we recommend building the projects from the bottom up in their natural order (see Figure 1), any project in the course represents a standalone building block that can be developed independently of all the other projects, in any desired order. The only requisite is the API of the level below—that is, its abstract interface.

Discussion: Pedagogy

A modular architecture and a system specification are static artifacts, not plans of action. To turn them into a working system, we provide staged implementation plans. The general staging strategy is based on sequential decomposition: Instead of realizing a complex abstraction in one sweep, the system architect can specify a basic version, which is implemented first. Once the basic version is developed and tested, one proceeds to extend it to a complete solution. Ideally, the API of the basic version should be a subset of the complete API, and the basic version should be morphed into, rather than replaced by, the complete version. Such staged implementations must be carefully articulated and supported by staged scaffolding.

Staging is informed by, but is not identical to, modularity. In some cases, the architect simply recommends the order in which modules should be developed and tested. In other cases, the development of the module itself is staged. For example, the hardware platform developed in Part I of the course consists of 35 modules (standalone chips) that are developed and unit-tested separately, according to staged plans given in each project. In complex chips such as the ALU, CPU, and the RAM, the implementation of the module itself is explicitly staged. For example, the Hack ALU is designed to compute a family of arithmetic/logic functions $f(x,y)$ on two 16bit inputs *x* and *y*. In addition, the ALU computes two 1bit outputs, indicating that its output is zero or negative. The computations of these flag bits are orthogonal to the ALU's main logic and can be realized separately, by independent blocks of HDL statements. With that in mind, our project 2 guidelines recommend building and testing a basic ALU that computes the $f(x,y)$ output only, and then extending the basic implementation to handle the two flag bits as well. The staged implementation is supported by two separate sets of ALU stub files, test scripts, and compare files.

Staged implementations are also inherent in the software projects in Part II of the course. For example, consider the assembler's development: In stage I, students are guided to develop a basic assembler that handles assembly programs containing no symbolic addresses. This is a fairly straightforward task: One writes a program that translates symbolic mnemonics into their binary codes, following the Hack machine-language specification. In stage II, students are guided to implement and unit-test a symbol table, following a proposed API. Finally, and using this added functionality, in stage III students morph the basic assembler into a final translator capable of handling assembly code with or without symbolic addresses. Here, too, the separation to stages is supported by customized and separate test files: assembly programs in which all variables and jump destinations are physical memory addresses for stage I, and assembly programs with symbolic labels for stages II and III.

Modularity and staging play a key role in the compiler's implementation, beginning with the separation into a back-end module (the bytecode-to-assembly translator developed in projects 7–8) and a front-end module (the Jack-to-bytecode compiler developed in projects 10–11). The implementation of each module is staged further into two separate projects. In project 7, students implement and test a basic virtual machine that features push/pop and arithmetic commands only. In project 8, they extend the machine to also handle branching and function calling. In project 10, students implement a basic compilation engine that uses a tokenizer and a parser to analyze the source code's syntax. In project 11, the compilation engine is extended to generate code. In each of these projects, students are guided to first handle source code that contains constants only, then variables, then expressions, and finally arrays and objects, each accompanied with customized test programs and compare files. For example, when writing the tokenizer and the parser, students use test programs that process the entire source code and print token lists and parse trees. These test programs are unsuitable for later stages, since the fully developed compiler gets the next token on the fly and builds the parse tree dynamically. However, the staged scaffolding is essential for turning the compiler's development from a daunting assignment into a sequence of relatively small tasks that can be localized, tested, and graded separately. In general, staged development is one of the key enablers of the accelerated pace of Nand to Tetris courses.

Design. In Nand to Tetris courses, instructors and students play the respective roles of system architects and junior developers. It is unsettling to see how, in many non-trivial programming assignments, computer science students are often left to their own devices, expected to figure out three very different things: how to design a system, how to implement it, and how to test it. As system architects, we eliminate two-thirds of this uncertainty: For each hardware and software module, we supply detailed design specifications, staged implementation plans, and test programs. Students are allowed to deviate from our proposed implementation and develop their own tests, but they are not permitted to modify the given specifications.

In Nand to Tetris, the discussion of every hardware or software module begins with an abstract specification of its intended functionality, and a tool that realizes the abstraction, hands-on.

Clearly, students must learn how to architect and specify systems. We believe, though, that a crucial element of mastering the art of design is seeing many good examples, as done consciously in architecture, law, medicine, and many other professional disciplines. In writing workshops, for example, significant learning time is spent reading the works of great masters and critically evaluating those of other workshop participants. Why not do the same when teaching systems building? In Nand to Tetris courses, students engage in dozens of meticulously planned architectures, specifications, and staged implementation plans. For many students, this may well be the most well-designed and well-managed development experience in their careers. Another reason for factoring out design and specification requirements to other courses is pragmatic: It allows completing the Nand to Tetris journey in one course, giving students a unique sense of closure and accomplishment.

Focus. Even when detailed designs and specifications are given, developing a general-purpose computer system in one academic course is a tall order. To render it feasible, we make two major concessions. First, we require that the constructed computer system will be fast enough, but no faster. By "fast enough" we mean that the computer must deliver a satisfying user experience. For example, if the computer's graphics are sufficiently smooth to support the animation required by simple computer games, then there is no need to optimize relevant hardware or software modules. In general, the performance of each built module is viewed pragmatically: As long as the module passes a set of operational tests supplied by us, there is no need to optimize it further. One exception is the OS, which is based on highly efficient and elegant algorithms.

In any hardware or software implementation project, much work is spent on handling exceptions such as edge cases and erroneous inputs. Our second concession is downplaying the former and ignoring the latter. For example, when students implement a chip that computes an n-bit arithmetic operation, they are allowed to ignore overflow and settle for computed values that are correct up to n bits. And, when they develop the assembler and the compiler, they are allowed to assume the source programs contain no syntax errors. Although learning to handle exceptions is an important educational objective, we believe it is equally important to assume, at least provisionally, an error-free world. This allows focusing on fundamental ideas and core concepts, rather than spending much time on handling exceptions, as required by industrial-strength applications.

The rationale for these concessions is pragmatic. First, without them, there would be no way to complete the computer's construction in one semester. Second, Nand to Tetris is a synthesis course that leaves many details to other, more specific CS courses. Third, any one of the limitations inherent in our computer system (and there are many, to be sure) provides a rich and well-motivated opportunity for extension projects, as described in this article's final section.

Exploration. Before implementing a hardware or software abstraction, we encourage playing with executable solutions. As students engage in these experiments, questions abound. We use these questions to motivate and explain our design decisions. For example: How can we rely on the ALU's calculations if it takes a while before they produce correct answers? Answer: When we will introduce sequential logic in the next project, we will set the clock cycle sufficiently long to allow time for the ALU circuits to stabilize on correct results. How can we use goto label instructions in assembly programs before the labels are declared? Answer: When we will write the assembler later in the course, we will present a two-pass translation algorithm that addresses this very issue. When a class *Foo* method creates a new object of class *Bar*, and given that each class is a separate compilation unit, how does *Foo*'s code know how much memory to allocate for the *Bar* object without having access to its field declarations? Answer: It does not know, but as you will see when we write the compiler, the compiled code of the *Bar* class constructor includes a call to an OS routine that allocates the required memory. Why are assignment statements in the Jack language preceded by a let prefix, as in `let x = 1`? Answer: This is one of the grammatical features that turns Jack into an LL(1) language, which is easier to compile using recursive descent algorithms. And why does Jack not have a `switch` statement? Answer: Indeed, this could be a nice touch; why not extend the language specification and implement switch in your compiler? And so it goes: Students are invited to question every design aspect of the architectures and languages presented in the course, and instructors are invited to discuss them critically and propose possible extensions.

The ability to work on each module in isolation, and often in parallel, allows developers to compartmentalize and manage complexity.

Extensions

Optimization. With the exception of the OS, the computer system built in the course is largely unoptimized, and improving its efficiency is a fertile playground for aspiring hardware and software engineers. We provide two examples, focusing on hardware and software optimizations. The n-bit ripple array adder chip built in Part I of the course (n = 16) is based on n lower-level full-adder chip-parts, each adding up two input bits and a carry bit. In the worst case, carry bits propagate from the least- to the most-significant full-adders, resulting in a computation delay that is proportional to n. To boost performance, we can augment the basic adder logic with Carry Look Ahead (CLA) logic. The CLA logic uses AND/OR operations to compute carry bits up the carry chain, enabling various degrees of parallel addition, depending on how far we are willing to look ahead. Alas, for large n values, the CLA logic becomes complex, and the efficiency gain of parallel addition may not justify the cost of the supplementary look-ahead logic. Cost-benefit analyses of various CLA schemes can help yield an optimized adder which is demonstratively faster than the basic one. This optimization is "nice to have," since the basic design of the adder is sufficiently fast for the course purposes. That said, every hardware module built in the course offers improvement opportunities that can be turned into follow-up, bonus assignments that go beyond the basic project requirements. Other examples include instructions requiring different clock times (IMUL / IDIV), pipelining, cache hierarchy, and more. Built-in versions of these extensions can be implemented in our open-ended hardware simulator, and then realized by students in HDL.

One of the software modules built in Part II of the course is a virtual machine. In projects 7–8, we guide students to realize this abstraction by writing a program that translates each VM command into several machine-language instructions. For example, consider the VM code sequence `push` *a*, `push` *b*, `add`. The semantics of the latter `add` primitive is "pop the two topmost values from the stack, add them up, and push the result onto the stack." In the standard VM implementation, the translation of each such VM command yields a separate chunk of binary instructions. Yet, an optimized translator could infer from the VM code that the first two push operations are superfluous, replacing the whole sequence with binary code that implements the single semantic operation `push` (*a* + *b*). Similar optimizations were made by Robert Woodhead, at the Hack assembly language level.[10] Such optimizations yield dramatic efficiency gains as well as valuable hands-on system-building lessons.

These are just two examples of the numerous opportunities to improve the efficiency of the hardware and software platforms built in Nand to Tetris courses. The simplicity of the platforms and the ubiquity of the software tools that support the coursework make such analyses and improvements a natural sequel of every lecture and project. Quite simply, once an improvement has been articulated algorithmically or technically, learners have what it takes to realize the extension and appreciate the resulting gains by experimenting with the optimized design in the relevant simulator.

FPGA. In typical Nand to Tetris courses, students build chips by writing HDL programs and executing them on the supplied hardware simulator. Committing the Hack computer to silicon requires two additional steps. First, one must rewrite the HDL programs of the main Hack chips using an industrial-strength language, such as Verilog or VHDL. This is not a difficult task, but one must learn the language's basics, which may well be one of the goals of this extension project. Next, using a low-cost FPGA board and open source FPGA synthesis tools, one can translate the HDL programs into an optimized configuration file that can then be loaded into the board, which becomes a physical implementation of the Hack computer. Examples of such extension projects, including step-by-step guidelines, are publicly available.[8]

Input/output. The Hack computer built in the course uses two memory bitmaps to connect to a black-and-white screen and to a standard keyboard. It would be nice to extend the basic Hack platform to accommodate a flexible and open-ended set of sensors, motors, relays, and displays, like those found on Arduino and Raspberry Pi platforms. This extension can be done as follows. First, allocate additional maps in the Hack memory for representing the various peripheral devices. Second, specify and implement an interrupt controller chip that stores the states of the individual interrupts triggered by the various I/O devices. Third, extend the Hack CPU to probe and handle the output of the interrupt controller. Finally, extend the operating system to mask, clean, and handle interrupts. We have started working on such extensions, but readers may well come up with better implementations.

Conclusion

This article described Nand to Tetris, an infrastructure for courses that teach applied computer science by building a general-purpose computer system—hardware and software—from the ground up. Nand to Tetris demystifies how computers work and how they are built, engaging students in 12 hands-on projects. Different courses can use different subsets of these projects and implement them in any desired order. Nand to Tetris courses are offered in academic settings that seek to combine key lessons from computer architecture and compilation in one course, and as popular MOOCS taken by many self-learners and developers. Part I of Nand to Tetris (hardware) is also suitable for high school CS programs. All Nand to Tetris course materials (lectures, projects, software tools) are available freely in open source[5,8] and instructors are welcome to use and extend them.

Acknowledgments

The chief contributors to the software suite that preceded the online Nand to Tetris IDE were Yaron Ukrainitz, Nir Rozen, and Yannai Gonczarowski. Mark Armbrust, William Bahn, Ran Navok, Yong Bakos, Tali Gutman, Rudolf Adamkovič, and Eytan Lifshitz made other significant contributions. Most of the key ideas and techniques underlying Nand to Tetris came from the brilliant mind of my friend and colleague, Noam Nisan.

References

1. Dijkstra, E.W. The humble programmer. *Commun. ACM 15*, 3 (Oct. 1972), 859–866.
2. *Nand to Tetris Course Syllabus. Computer Science Dept.*, Princeton University; https://bit.ly/3raALBk.
3. Nisan, N. and Schocken, S. *The Elements of Computing Systems*. 2nd ed., MIT Press (2021).
4. Patterson, D.A. and Hennessy, J.L. *Computer Organization and Design RISC-V Edition*. 2nd ed., Morgan Kauffman, Cambridge, MA (2021), 11–13.
5. Schocken, S. and Nisan, N. *Nand to Tetris website*; https://bit.ly/3XD0Rt4.
6. Schocken, S., Nisan, N., and Armoni, M. A synthesis course in hardware architecture, compilers, and software engineering. In *Proceedings of the ACM SIGCSE*. ACM (Mar. 2009), 443–447.
7. Schröder, M. *FPGA implementations of the Hack Computer*; https://bit.ly/3puLCpp, https://bit.ly/3NZIiMu.
8. Souther, D. and London, N. *Nand to Tetris IDE Online*; bit.ly/3wNjeSu.
9. Strunk, Jr., W. and White, E.B. *The Elements of Style*, Macmillan (1959).
10. Woodhead, R.J. Optimizing Nand2Tetris assembly code. *Medium* (Dec. 2023); bit.ly/4acMJfc

Shimon Schocken (schocken@runi.ac.il) is a professor at the Efi Arazi School of Computer Science, Reichman University, Israel.

© 2024 Copyright held by owner(s)/author(s).

Watch the author discuss this work in the exclusive *Communications* video. https://cacm.acm.org/videos/nand-to-tetris