

Memory

peek, poke: These functions provide direct access to the host memory. How can this low-level access be accomplished using the Jack high-level language? As it turns out, the Jack language includes a trapdoor that enables gaining complete control of the host computer's memory. This trapdoor can be exploited for implementing `Memory.peek` and `Memory.poke` using plain Jack programming.

The trick is based on an anomalous use of a reference variable (pointer). Jack is a weakly typed language; among other quirks, it does not prevent the programmer from assigning a constant to a reference variable. This constant can then be treated as an absolute memory address. When the reference variable happens to be an array, this scheme provides indexed access to every word in the host RAM. See [figure 12.11](#).

```
// Creates a Jack-level “proxy” of the RAM:
var Array memory;
let memory = 0; // No problem . . .
. . .
// Gets the value of the RAM at address i:
let x = memory[i];
. . .
// Sets the value of the RAM at address i:
let memory[i] = 17;
. . .
```

Figure 12.11 A trapdoor enabling complete control of the host RAM from Jack.

Following the first two lines of code, the base of the memory array points to the first address in the computer's RAM (address 0). To get or set the value of the RAM location whose physical address is *i*, all we have to do is

manipulate the array element `memory[i]`. This will cause the compiler to manipulate the RAM location whose address is $0 + i$, which is what we want.

Jack arrays are not allocated space on the heap at compile-time but rather at run-time, if and when the array's `new` function is called. Note that if `new` were a constructor and not a function, the compiler and the OS would have allocated the new array to some obscure address in the RAM that we cannot control. Like many classical hacks, this trick works because we use the array variable without initializing it properly, as is normally done when using arrays.

The memory array can be declared at the class level and initialized by the `Memory.init` function. Once this hack is done, the implementation of `Memory.peek` and `Memory.poke` becomes trivial.

alloc, deAlloc: These functions can be implemented by either one of the algorithms shown in [figures 12.5a](#) and [12.5b](#). Either *best-fit* or *first-fit* can be used for implementing `Memory.deAlloc`.

The standard VM mapping on the Hack platform (see section 7.4.1) specifies that the *stack* be mapped on RAM addresses 256 to 2047. Thus the *heap* can start at address 2048.

In order to realize the `freeList` linked list, the `Memory` class can declare and maintain a static variable, `freeList`, as seen in [figure 12.12](#). Although `freeList` is initialized to the value of `heapBase` (2048), it is possible that following several `alloc` and `deAlloc` operations `freeList` will become some other address in memory, as illustrated in the figure.

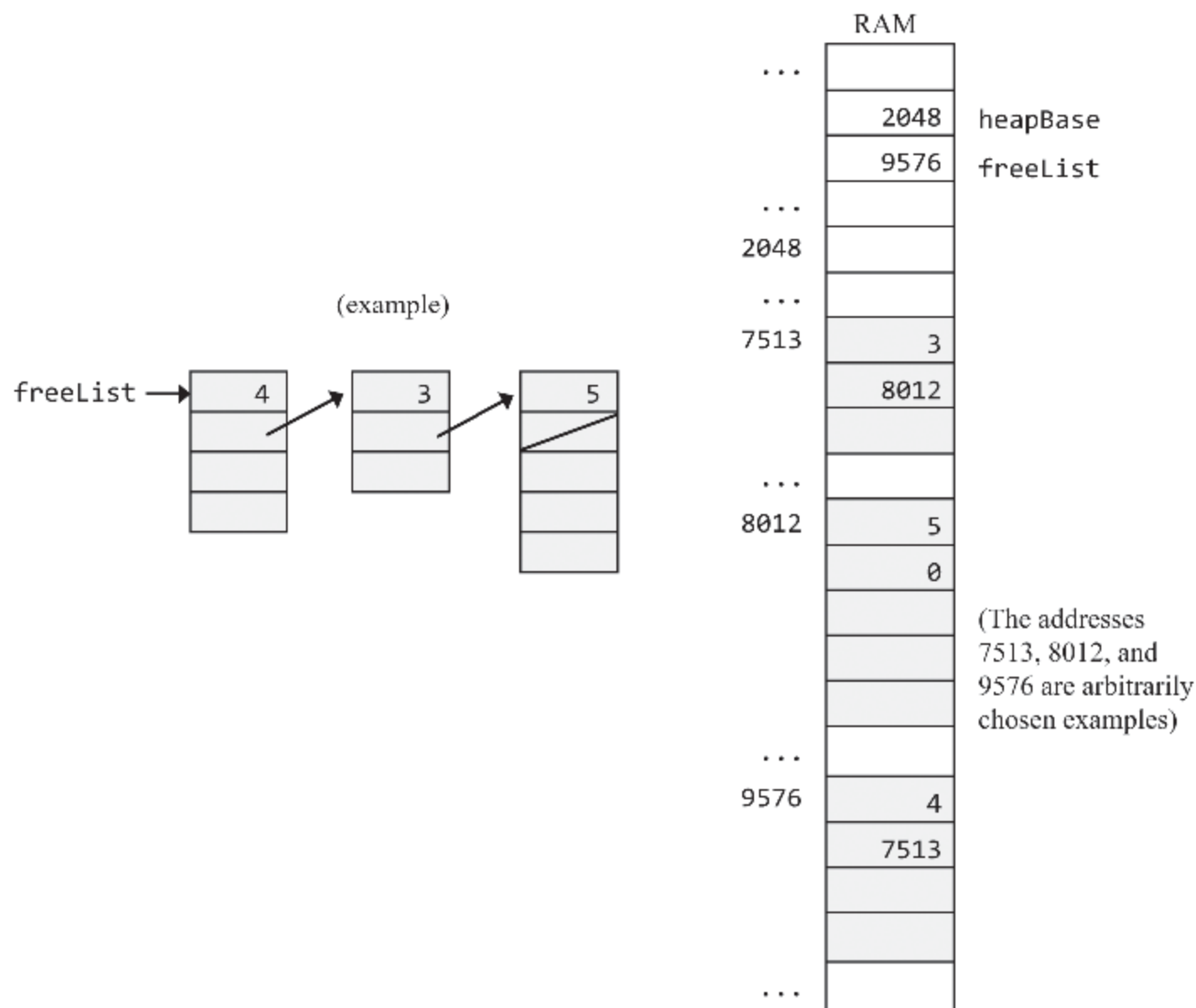


Figure 12.12 Logical view (left) and physical implementation (right) of a linked list that supports dynamic memory allocation.

For efficiency's sake, it is recommended to write Jack code that manages the freeList linked list directly in the RAM, as seen in [figure 12.12](#). The linked list can be initialized by the Memory.init function.

Screen

The Screen class maintains a *current color* that is used by all the drawing functions of the class. The current color can be represented by a static Boolean variable.

drawPixel: Drawing a pixel on the screen can be done using Memory.peek and Memory.poke. The screen memory map of the Hack platform specifies that the pixel at column *col* and row *row* ($0 \leq col \leq 511, 0 \leq row \leq 255$) is mapped to the $col \% 16$ bit of memory location $16384 + row \cdot 32 + col / 16$. Drawing a single pixel requires changing a single bit in the accessed word (and that bit only).

drawLine: The basic algorithm in [figure 12.7](#) can potentially lead to overflow. However, the algorithm's improved version eliminates the problem.

Some aspects of the algorithm should be generalized for drawing lines that extend to four possible directions. Be reminded that the screen origin (coordinates (0,0)) is at the top-left corner. Therefore, some of the directions and plus/minus operations specified in the algorithm should be modified by your drawLine implementation.

The special yet frequent cases of drawing straight lines, that is, when $dx = 0$ or $dy = 0$, should not be handled by this algorithm. Rather, they should benefit from a separate and optimized implementation.

drawCircle: The algorithm shown in [figure 12.8](#) can potentially lead to overflow. Limiting circle radii to be at most 181 is a reasonable solution.

Output

The Output class is a library of functions for displaying characters. The class assumes a character-oriented screen consisting of 23 rows (indexed 0 ... 22, top to bottom) of 64 characters each (indexed 0 ... 63, left to right). The top-left character location on the screen is indexed (0,0). A visible cursor, implemented as a small filled square, indicates where the next character will be displayed. Each character is displayed by rendering on the screen a rectangular image, 11 pixels high and 8 pixels wide (which includes margins for character spacing and line spacing). The collection of all the character images is called a *font*.

Font implementation: The design and implementation of a font for the Hack character set (appendix 5) is a drudgery, combining artistic judgment and rote implementation work. The resulting font is a collection of ninety-five rectangular bitmap images, each representing a printable character.

Fonts are normally stored in external files that are loaded and used by the character-drawing package, as needed. In Nand to Tetris, the font is embedded in the OS Output class. For each printable character, we define an array that holds the character's bitmap. The array consists of 11 elements, each corresponding to a row of 8 pixels. Specifically, we set the value of

each array entry j to an integer value whose binary representation (bits) codes the 8 pixels appearing in the j -th row of the character's bitmap. We also define a static array of size 127, whose index values 32 ... 126 correspond to the codes of the printable characters in the Hack character set (entries 0 ... 31 are not used). We then set each array entry i of that array to the 11-entry array that represents the bitmap image of the character whose character code is i (did we mention drudgery?).

The project 12 materials include a skeletal Output class containing Jack code that carries out all the implementation work described above. The given code implements the ninety-five-character font, except for one character, whose design and implementation is left as an exercise. This code can be activated by the `Output.init` function, which can also initialize the cursor.

printChar: Displays the character at the cursor location and advances the cursor one column forward. To display a character at location (row, col) , where $0 \leq row \leq 22$ and $0 \leq col \leq 63$, we write the character's bitmap onto the box of pixels ranging from $11 \cdot row$ to $11 \cdot row + 10$ and from $8 \cdot col$ to $8 \cdot col + 7$.

printString: Can be implemented using a sequence of `printChar` calls.

println: Can be implemented by converting the integer to a string and then printing the string.

Keyboard

The Hack computer memory organization (see section 5.2.6) specifies that the *keyboard memory map* is a single 16-bit memory register located at address 24576.

keyPressed: Can be implemented easily using `Memory.peek ()`.

readChar, readString: Can be implemented by following the algorithms in [figure 12.10](#).

readInt: Can be implemented by reading a string and converting it into an int value using a String method.

Sys

wait: This function is supposed to wait a given number of milliseconds and return. It can be implemented by writing a loop that runs approximately duration milliseconds before terminating. You will have to time your specific computer to obtain a one millisecond wait, as this constant varies from one CPU to another. As a result, your `Sys.wait()` function will not be portable. The function can be made portable by running yet another configuration function that sets various constants reflecting the hardware specifications of the host platform, but for Nand to Tetris this is not needed.

halt: Can be implemented by entering an infinite loop.

init: According to the Jack language specification (see section 9.2.2), a Jack program is a set of one or more classes. One class must be named `Main`, and this class must include a function named `main`. To start running a program, the `Main.main` function should be called.

The operating system is also a collection of compiled Jack classes. When the computer boots up, we want to start running the operating system and have *it* start running the main program. This chain of command is implemented as follows. According to the Standard VM Mapping on the Hack Platform (section 8.5.2), the VM translator writes bootstrap code (in machine language) that calls the OS function `Sys.init`. This bootstrap code is stored in the ROM, starting at address 0. When we reset the computer, the program counter is set to 0, the bootstrap code starts running, and the `Sys.init` function is called.

With that in mind, `Sys.init` should do two things: call all the `init` functions of the other OS classes, and then call `Main.main`.

From this point onward the user is at the mercy of the application program, and the Nand to Tetris journey has come to an end. We hope that you enjoyed the ride!

12.4 Project

Objective: Implement the operating system described in the chapter.

Contract: Implement the operating system in Jack, and test it using the programs and testing scenarios described below. Each test program uses a subset of OS services. Each of the OS classes can be implemented and unit-tested in isolation, in any order.

Resources: The main required tool is Jack—the language in which you will develop the OS. You will also need the supplied Jack compiler for compiling your OS implementation, as well as the supplied test programs, also written in Jack. Finally, you’ll need the supplied VM emulator, which is the platform on which the tests will be executed.

Your projects/12 folder includes eight skeletal OS class files named Math.jack, String.jack, Array.jack, Memory.jack, Screen.jack, Output.jack, Keyboard.jack, and Sys.jack. Each file contains the signatures of all the class subroutines. Your task is completing the missing implementations.

The VM emulator: Operating system developers often face the following chicken-and-egg dilemma: How can we possibly test an OS class in isolation, if the class uses the services of other OS classes not yet developed? As it turns out, the VM emulator is perfectly positioned to support unit-testing the OS, one class at a time.

Specifically, the VM emulator features an executable version of the OS, written in Java. When a VM command call foo is found in the loaded VM code, the emulator proceeds as follows. If a VM function named foo exists in the loaded code base, the emulator executes its VM code. Otherwise, the emulator checks whether foo is one of the built-on OS functions. If so, it executes foo’s built-in implementation. This convention is ideally suited for supporting the testing strategy that we now turn to describe.

Testing Plan

Your projects/12 folder includes eight test folders, named MathTest, MemoryTest, ..., for testing each one the eight OS classes Math, Memory, Each folder contains a Jack program, designed to test (by using) the services of the corresponding OS class. Some folders contain test scripts and compare

files, and some contain only a .jack file or files. To test your implementation of the OS class *Xxx.jack*, you may proceed as follows:

- Inspect the supplied *XxxTest/*.jack* code of the test program. Understand which OS services are tested and how they are tested.
- Put the OS class *Xxx.jack* that you developed in the *XxxTest* folder.
- Compile the folder using the supplied Jack compiler. This will result in translating both your OS class file and the .jack file or files of the test program into corresponding .vm files, stored in the same folder.
- If the folder includes a .tst test script, load the script into the VM emulator; otherwise, load the folder into the VM emulator.
- Follow the specific testing guidelines given below for each OS class.

Memory, Array, Math: The three folders that test these classes include test scripts and compare files. Each test script begins with the command `load`. This command loads all the .vm files in the current folder into the VM emulator. The next two commands in each test script create an output file and load the supplied compare file. Next, the test script proceeds to execute several tests, comparing the test results to those listed in the compare file. Your job is making sure that these comparisons end successfully.

Note that the supplied test programs don't comprise a full test of `Memory.alloc` and `Memory.deAlloc`. A complete test of these memory management functions requires inspecting internal implementation details not visible in user-level testing. If you want to do so, you can test these functions by using step-by-step debugging and by inspecting the state of the host RAM.

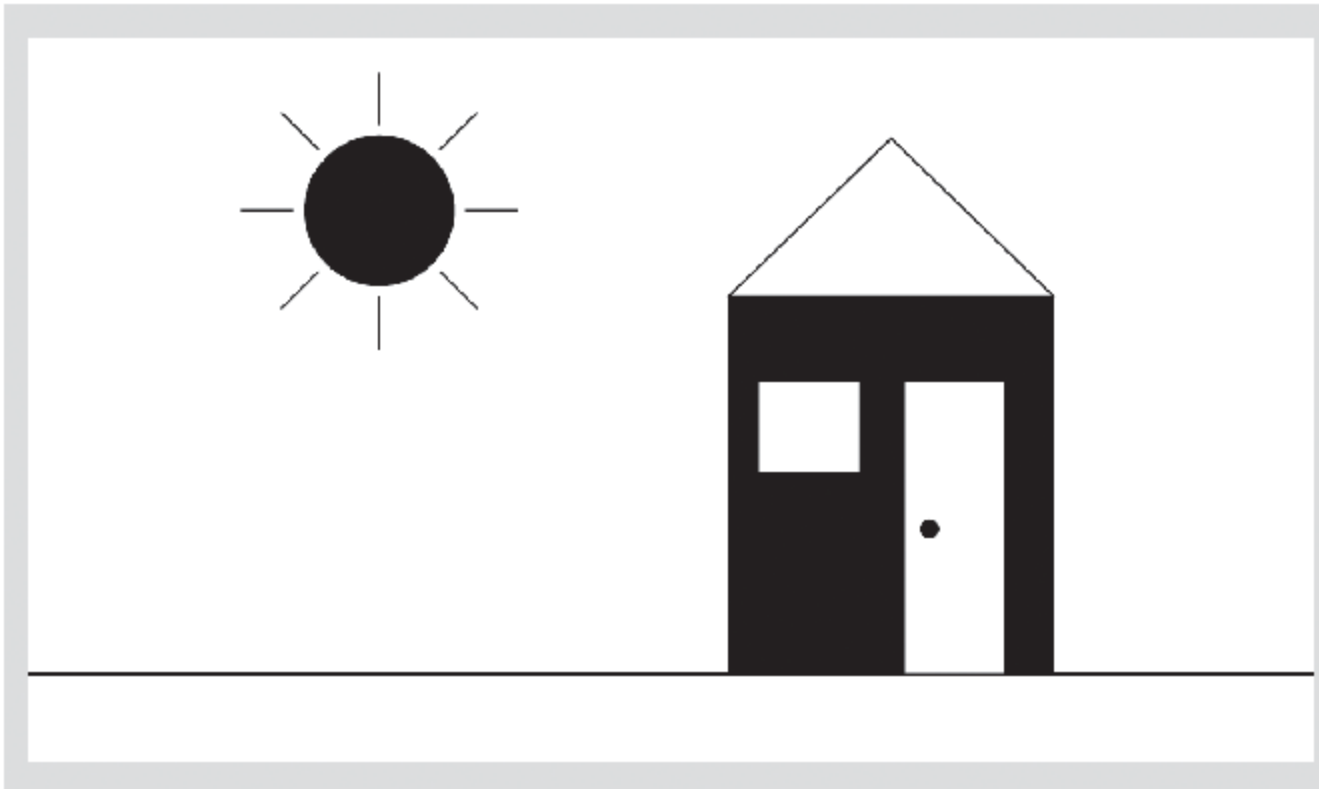
String: Execution of the supplied test program should yield the following output:


```
new,appendChar: abcde  
setInt: 12345  
setInt: -32767  
length: 5  
charAt[2]: 99  
setCharAt(2,'-'): ab-de  
eraseLastChar: ab-d  
intValue: 456  
intValue: -32123  
backSpace: 129  
doubleQuote: 34  
newLine: 128
```

Output: Execution of the supplied test program should yield the following output:

```
A B  
0123456789  
ABCDEFGHIJKLMN O PQRSTU VWXYZ abcdefghijklmnopqrstuvwxyz  
!#$%&'()*+,-./:;<=>?@[ ]^_`{|}~"  
-12346789  
C D
```

Screen: Execution of the supplied test program should yield the following output:



Keyboard: This OS class is tested by a test program that effects user-program interaction. For each function in the Keyboard class (keyPressed, readChar, readLine, readInt) the program prompts the user to press some keys. If the OS function is implemented correctly and the requested keys are pressed, the program prints ok and proceeds to test the next OS function. Otherwise, the program repeats the request. If all requests end successfully, the program prints Test ended successfully. At this point the screen should show the following output:

```
keyPressed test:
Please press the 'Page Down' key
ok
readChar test:
(Verify that the pressed character is echoed to the screen)
Please press the number '3': 3
ok
readLine test:
(Verify echo and usage of 'backspace')
Please type 'JACK' and press enter: JACK
ok
readInt test:
(Verify echo and usage of 'backspace')
Please type '-32123' and press enter: -32123
ok
Test completed successfully
```

Sys

The supplied `.jack` file tests the `Sys.wait` function. The program requests the user to press a key (any key) and waits two seconds, using a call to `Sys.wait`. It then prints a message on the screen. Make sure that the time that elapses between your release of the key and the appearance of the printed message is about two seconds.

The `Sys.init` function is not tested explicitly. However, recall that it performs all the necessary OS initializations and then calls the `Main.main` function of each test program. Therefore, we can assume that nothing will work properly unless `Sys.init` is implemented correctly.

Complete Test

After testing successfully each OS class in isolation, test your entire OS implementation using the Pong game introduced earlier in the book. The source code of Pong is available in `projects/11/Pong`. Put your eight OS `.jack` files in the Pong folder, and compile the folder using the supplied Jack compiler. Next, load the Pong folder into the VM emulator, execute the game, and ensure that it works as expected.

12.5 Perspective

This chapter presented a subset of basic services that can be found in most operating systems. For example, managing memory, driving I/O devices, supplying mathematical operations not implemented in hardware, and implementing abstract data types like the string abstraction. We have chosen to call this standard software library an *operating system* to reflect its two main functions: encapsulating the gory hardware details, omissions, and idiosyncrasies in transparent software services, and enabling compilers and application programs to use these services via clean interfaces. However, the gap between what we have called an OS and industrial-strength operating systems remains wide.

For starters, our OS lacks some of the basic services most closely associated with operating systems. For example, our OS supports neither multi-threading nor multiprocessing; in contrast, the kernel of most

operating systems is devoted to exactly that. Our OS supports no mass storage devices; in contrast, the main data store handled by operating systems is a file system abstraction. Our OS features neither a command-line interface (as in a Unix shell) nor a graphical interface consisting of windows and menus. In contrast, this is the operating system interface that users expect to see and interact with. Numerous other services commonly found in operating systems are not present in our OS, like security, communication, and more.

Another notable difference lies in the liberal way in which our OS operations are invoked. Some OS operations, for example, peek and poke, give the programmer complete access to the host computer resources. Clearly, inadvertent or malicious use of such functions can cause havoc. Therefore, many OS services are considered privileged, and accessing them requires a security mechanism that is more elaborate than a simple function call. In contrast, in the Hack platform, there is no difference between OS code and user code, and operating system services run in the same user mode as that of application programs.

In terms of efficiency, the algorithms that we presented for multiplication and division were standard. These algorithms, or variants thereof, are typically implemented in hardware rather than in software. The running time of these algorithms is $O(n)$ addition operations. Since adding two n -bit numbers requires $O(n)$ bit operations (gates in hardware), these algorithms end up requiring $O(n^2)$ bit operations. There exist multiplication and division algorithms whose running time is asymptotically significantly faster than $O(n^2)$, and for a large number of bits these algorithms are more efficient. In a similar fashion, optimized versions of the geometric operations that we presented, like line drawing and circle drawing, are often implemented in special graphics-acceleration hardware.

Like every hardware and software system developed in Nand to Tetris, our goal is not to provide a complete solution that addresses all wants and needs. Rather, we strive to build a working implementation and a solid understanding of the system's *foundation*, and then propose ways to extend it further. Some of these optional extension projects are mentioned in the next and final chapter in the book.

13 More Fun to Go

We shall not cease from exploration, and at the end we will arrive where we started, and know the place for the first time.

—T. S. Eliot (1888–1965)

Congratulations! We’ve finished the construction of a complete computing system, starting from first principles. We hope that you enjoyed this journey. Let us, the authors of this book, share a secret with you: We suspect that we enjoyed writing the book even more. After all, we got to design this computing system, and design is often the “funnest” part of every project. We are sure that some of you, adventurous learners, would like to get in on that design action. Maybe you would like to improve the architecture; maybe you have ideas for adding new features here and there; maybe you envision a wider system. And then, maybe, you want to be in the navigator’s seat and decide where to go, not just how to get there.

Almost every aspect of the Jack/Hack system can be improved, optimized, or extended. For example, the assembly language, the Jack language, and the operating system can be modified and extended by rewriting portions of your respective assembler, compiler, and OS implementations. Other changes would likely require modifying the supplied software tools as well. For example, if you change the hardware specification or the VM specification, then you would likely want to modify the respective emulators. Or, if you want to add more input or output devices to the Hack computer, you would probably need to model them by writing new built-in chips.

In order to allow complete flexibility of modification and extension, we made the source code of all the supplied tools publicly available. All our

code is 100 percent Java, except for some of the batch files used for starting the software on some platforms. The software and its documentation are available at www.nand2tetris.org. You are welcome to modify and extend all our tools as you deem desirable for your latest idea—and then share them with others, if you want. We hope that our code is written and documented well enough to make modifications a satisfying experience. In particular, we wish to mention that the supplied hardware simulator has a simple and well-documented interface for adding new built-in chips. This interface can be used for extending the simulated hardware platform with, say, mass storage or communications devices.

While we cannot even start to imagine what your design improvements may be, we can briefly sketch some of the ones we were thinking of.

Hardware Realizations

The hardware modules presented in the book were implemented either in HDL or as supplied executable software modules. This, in fact, is how hardware is actually designed. However, at some point, the HDL designs are typically committed to silicon, becoming “real” computers. Wouldn’t it be nice to make Hack or Jack run on a real hardware platform made of atoms rather than bits?

Several different approaches may be taken toward this goal. On one extreme, you can attempt to implement the Hack platform on an FPGA board. This would require rewriting all the chip definitions using a mainstream Hardware Description Language and then dealing with implementation issues related to realizing the RAM, the ROM, and the I/O devices on the host board. One such step-by-step optional project, developed by Michael Schröder, is referred to in the www.nand2tetris.org website. Another extreme approach may be to attempt emulating Hack, the VM, or even the Jack platform on an existing hardware device like a cell phone. It seems that any such project would want to reduce the size of the Hack screen to keep the cost of the hardware resources reasonable.

Hardware Improvements

Although Hack is a *stored program* computer, the program that it runs must be prestored in its ROM device. In the present Hack architecture, there is no way of loading another program into the computer, except for simulating the replacement of the entire physical ROM chip.

Adding a *load program* capability in a balanced way would likely involve changes at several levels of the hierarchy. The Hack hardware should be modified to allow loaded programs to reside in the writable RAM rather than in the existing ROM. Some type of permanent storage, like a built-in mass storage chip, should probably be added to the hardware to allow storage of programs. The operating system should be extended to handle this permanent storage device, as well as new logic for loading and running programs. At this point an OS user interface *shell* would come in handy, providing file and program management commands.

High-Level Languages

Like all professionals, programmers have strong feelings about their tools—programming languages—and like to personalize them. And indeed, the Jack language, which leaves much to be desired, can be significantly improved. Some changes are simple, some are more involved, and some—like adding inheritance—would likely require modifying the VM specification as well.

Another option is realizing more high-level languages over the Hack platform. For example, how about implementing Scheme?

Optimization

Our Nand to Tetris journey has almost completely sidestepped optimization issues (except for the operating system, which introduced some efficiency measures). Optimization is a great playfield for hackers. You can start with local optimizations in the hardware, or in the compiler, but the best bang for the buck will come from optimizing the VM translator. For example, you may want to reduce the size of the generated assembly code, and make it

more efficient. Ambitious optimizations on a more global scale will involve changing the specifications of the machine language or the VM language.

Communications

Wouldn't it be nice to connect the Hack computer to the Internet? This could be done by adding a built-in communication chip to the hardware and writing an OS class for dealing with it and for handling higher-level communication protocols. Some other programs would need to talk with the built-in communication chip, providing an interface to the Internet. For example, an HTTP-speaking web browser in Jack seems like a feasible and worthy project.

These are some of our design itches—what are yours?

Appendix 1: Boolean Function Synthesis

By logic we prove, by intuition we discover.

—Henri Poincaré (1854–1912)

In chapter 1 we made the following claims, without proof:

- Given a truth table representation of a Boolean function, we can synthesize from it a Boolean expression that realizes the function.
- Any Boolean function can be expressed using only And, Or, and Not operators.
- Any Boolean function can be expressed using only Nand operators.

This appendix provides proofs for these claims, and shows that they are interrelated. In addition, the appendix illustrates the process by which Boolean expressions can be simplified using Boolean algebra.

A1.1 Boolean Algebra

The Boolean operators And, Or, and Not have useful algebraic properties. We present some of these properties briefly, noting that their proofs can be easily derived from the relevant truth tables listed in [figure 1.1](#) of chapter 1.

Commutative laws: $x \text{ And } y = y \text{ And } x$

$$x \text{ Or } y = y \text{ Or } x$$

Associative laws: $x \text{ And } (y \text{ And } z) = (x \text{ And } y) \text{ And } z$

$$x \text{ Or } (y \text{ Or } z) = (x \text{ Or } y) \text{ Or } z$$

Distributive laws: $x \text{ And } (y \text{ Or } z) = (x \text{ And } y) \text{ Or } (x \text{ And } z)$

$$x \text{ Or } (y \text{ And } z) = (x \text{ Or } y) \text{ And } (x \text{ Or } z)$$

De Morgan laws: $\text{Not}(x \text{ And } y) = \text{Not}(x) \text{ Or } \text{Not}(y)$

$$\text{Not}(x \text{ Or } y) = \text{Not}(x) \text{ And } \text{Not}(y)$$

Idempotent laws: $x \text{ And } x = x$

$$x \text{ Or } x = x$$

These algebraic laws can be used to simplify Boolean functions. For example, consider the function $\text{Not}(\text{Not}(x) \text{ And } \text{Not}(x \text{ Or } y))$. Can we reduce it to a simpler form? Let's try and see what we can come up with:

```
Not(Not(x) And Not(x Or y)) =           // by De Morgan law:
Not(Not(x) And (Not(x) And Not(y))) =    // by the associative law:
Not((Not(x) And Not(x)) And Not(y)) =    // by the idempotent law:
Not(Not(x) And Not(y)) =                 // by De Morgan law:
Not(Not(x)) Or Not(Not(y)) =             // by double negation:
x Or y
```

Boolean simplifications like the one just illustrated have significant practical implications. For example, the original Boolean expression $\text{Not}(\text{Not}(x) \text{ And } \text{Not}(x \text{ Or } y))$ can be implemented in hardware using five logic gates, whereas the simplified expression $x \text{ Or } y$ can be implemented using a single logic gate. Both expressions deliver the same functionality, but the latter is five times more efficient in terms of cost, energy, and speed of computation.

Reducing a Boolean expression into a simpler one is an art requiring experience and insight. Various reduction tools and techniques are available, but the problem remains challenging. In general, reducing a Boolean expression into its simplest form is an *NP-hard* problem.

A1.2 Synthesizing Boolean Functions

Given a truth table of a Boolean function, how can we construct, or synthesize, a Boolean expression that represents this function? And, come to think of it, are we *guaranteed* that every Boolean function represented by a truth table can also be represented by a Boolean expression?

These questions have very satisfying answers. First, yes: every Boolean function can be represented by a Boolean expression. Moreover, there is a constructive algorithm for doing just that. To illustrate, refer to [figure A1.1](#), and focus on its leftmost four columns. These columns specify a truth table definition of some three-variable function $f(x,y,z)$. Our goal is to synthesize from these data a Boolean expression that represents this function.

x	y	z	$f(x,y,z)$	$f_3(x,y,z)$	$f_5(x,y,z)$	$f_7(x,y,z)$
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	1	1	0	0
0	1	1	0	0	0	0
1	0	0	1	0	1	0
1	0	1	0	0	0	0
1	1	0	1	0	0	1
1	1	1	0	0	0	0

$$f_3(x,y,z) = \text{Not}(x) \text{ And } y \text{ And Not}(z)$$

$$f_5(x,y,z) = x \text{ And Not}(y) \text{ And Not}(z)$$

$$f_7(x,y,z) = x \text{ And } y \text{ And Not}(z)$$

$$f(x,y,z) = f_3(x,y,z) \text{ Or } f_5(x,y,z) \text{ Or } f_7(x,y,z)$$

Figure A1.1 Synthesizing a Boolean function from a truth table (example).

We'll describe the synthesis algorithm by following its steps in this particular example. We start by focusing only on the truth table's rows in which the function's value is 1. In the function shown in [figure A1.1](#), this happens in rows 3, 5, and 7. For each such row i , we define a Boolean

function f_i that returns 0 for all the variable values except for the variable values in row i , for which the function returns 1. The truth table in [figure A1.1](#) yields three such functions, whose truth table definitions are listed in the three rightmost columns in the table. Each of these functions f_i can be represented by a conjunction (And-ing) of three terms, one term for each variable x , y , and z , each being either the variable or its negation, depending on whether the value of this variable is 1 or 0 in row i . This construction yields the three functions f_3 , f_5 , and f_7 , listed at the bottom of the table. Since these functions describe the only cases in which the Boolean function f evaluates to 1, we conclude that f can be represented by the Boolean expression $f(x, y, z) = f_3(x, y, z) \text{ Or } f_5(x, y, z) \text{ Or } f_7(x, y, z)$. Spelling it out: $f(x, y, z) = (\text{Not } (x) \text{ And } y \text{ And Not } (z)) \text{ Or } (x \text{ And Not } (y) \text{ And Not } (z)) \text{ Or } (x \text{ And } y \text{ And Not } (z))$.

Avoiding tedious formality, this example suggests that any Boolean function can be systematically represented by a Boolean expression that has a very specific structure: it is the disjunction (Or-ing) of all the conjunctive (And-ing) functions f_i whose construction was just described. This expression, which is the Boolean version of a sum of products, is sometimes referred to as the function's *disjunctive normal form* (DNF).

Note that if the function has many variables, and thus the truth table has exponentially many rows, the resulting DNF may be long and cumbersome. At this point, Boolean algebra and various reduction techniques can help transform the expression into a more efficient and workable representation.

A1.3 The Expressive Power of Nand

As our *Nand to Tetris* title suggests, every computer can be built using nothing more than Nand gates. There are two ways to support this claim. One is to actually build a computer from Nand gates only, which is exactly what we do in part I of the book. Another way is to provide a formal proof, which is what we'll do next.

Lemma 1: Any Boolean function can be represented by a Boolean expression containing only And, Or, and Not operators.

Proof: Any Boolean function can be used to generate a corresponding truth table. And, as we've just shown, any truth table can be used for synthesizing a DNF, which is an Or-ing of And-ings of variables and their negations. It follows that any Boolean function can be represented by a Boolean expression containing only And, Or, and And operators.

In order to appreciate the significance of this result, consider the infinite number of functions that can be defined over *integer numbers* (rather than binary numbers). It would have been nice if every such function could be represented by an algebraic expression involving only addition, multiplication, and negation. As it turns out, the vast majority of integer functions, for example, $f(x) = 2x$ for $x \neq 7$ and $f(7) = 312$, cannot be expressed using a close algebraic form. In the world of *binary numbers*, though, due to the finite number of values that each variable can assume (0 or 1), we do have this attractive property that every Boolean function *can* be expressed using nothing more than And, Or, and Not operators. The practical implication is immense: any computer can be built from nothing more than And, Or, and Not gates.

But, can we do better than this?

Lemma 2: Any Boolean function can be represented by a Boolean expression containing only Not and And operators.

Proof: According to De Morgan law, the Or operator can be expressed using the Not and And operators. Combining this result with Lemma 1, we get the proof.

Pushing our luck further, can we do better than this?

Theorem: Any Boolean function can be represented by a Boolean expression containing only Nand operators.

Proof: An inspection of the Nand truth table (the second-to-last row in [figure 1.2](#) in chapter 1) reveals the following two properties:

- $\text{Not}(x) = \text{Nand}(x, x)$

In words: If you set both the x and y variables of the Nand function to the same value (0 or 1), the function evaluates to the negation of that

value.

- $\text{And}(x, y) = \text{Not}(\text{Nand}(x, y))$

It is easy to show that the truth tables of both sides of the equation are identical. And, we've just shown that Not can be expressed using Nand.

Combining these two results with Lemma 2, we get that any Boolean function can be represented by a Boolean expression containing only Nand operators.

This remarkable result, which may well be called the fundamental theorem of logic design, stipulates that computers can be built from one atom only: a logic gate that realizes the Nand function. In other words, if we have as many Nand gates as we want, we can wire them in patterns of activation that implement any given Boolean function: all we have to do is figure out the right wiring.

Indeed, most computers today are based on hardware infrastructures consisting of billions of Nand gates (or Nor gates, which have similar generative properties). In practice, though, we don't have to limit ourselves to Nand gates only. If electrical engineers and physicists can come up with efficient and low-cost physical implementations of other elementary logic gates, we will happily use them directly as primitive building blocks. This pragmatic observation does not take away anything from the theorem's importance.

Appendix 2: Hardware Description Language

Intelligence is the faculty of making artificial objects, especially tools to make tools.

—Henry Bergson (1859–1941)

This appendix has two main parts. Sections A2.1–A2.5 describe the HDL language used in the book and in the projects. Section A2.6, named HDL Survival Guide, provides a set of essential tips for completing the hardware projects successfully.

A Hardware Description Language (HDL) is a formalism for defining *chips*: objects whose *interfaces* consist of input and output *pins* that carry binary signals, and whose *implementations* are connected arrangements of other, lower-level chips. This appendix describes the HDL that we use in Nand to Tetris. Chapter 1 (in particular, section 1.3) provides essential background that is a prerequisite to this appendix.

A2.1 HDL Basics

The HDL used in Nand to Tetris is a simple language, and the best way to learn it is to play with HDL programs using the supplied hardware simulator. We recommend starting to experiment as soon as you can, beginning with the following example.

Example: Suppose we have to check whether three 1-bit variables a , b , c have the same value. One way to check this three-way equality is to evaluate the Boolean function $\neg((a \neq b) \vee (b \neq c))$. Noting that the binary

operator *not-equal* can be realized using a Xor gate, we can implement this function using the HDL program shown in [figure A2.1](#).

interface	{	<pre> /** If the three given bits are equal, sets out to 1; else sets out to 0. */ CHIP Eq3 { IN a, b, c; OUT out; PARTS: Xor(a=a, b=b, out=neq1); // Xor(a,b) → neq1 Xor(a=b, b=c, out=neq2); // Xor(b,c) → neq2 Or (a=neq1, b=neq2, out=outOr); // Or(neq1,neq2) → outOr Not(in=outOr, out=out); // Not(outOr) → out } </pre>
implementation	{	

Figure A2.1 HDL program example.

The Eq3.hdl implementation uses four *chip-parts*: two Xor gates, one Or gate, and one Not gate. To realize the logic expressed by $\neg((a \neq b) \vee (b \neq c))$, the HDL programmer connects the chip-parts by creating, and naming, three *internal pins*: neq1, neq2, and outOr.

Unlike internal pins, which can be created and named at will, the HDL programmer has no control over the names of the input and output pins. These are normally supplied by the chips' architects and documented in given APIs. For example, in Nand to Tetris, we provide *stub files* for all the chips that you have to implement. Each stub file contains the chip interface, with a missing implementation. The contract is as follows: You are allowed to do whatever you want *under* the PARTS statement; you are not allowed to change anything *above* the PARTS statement.

In the Eq3 example, it so happens that the first two inputs of the Eq3 chip and the two inputs of the Xor and Or chip-parts have the same names (a and b). Likewise, the output of the Eq3 chip and that of the Not chip-part happen to have the same name (out). This leads to bindings like a=a, b=b, and out=out. Such bindings may look peculiar, but they occur frequently in HDL programs, and one gets used to them. Later in the appendix we'll give a simple rule that clarifies the meaning of these bindings.

Importantly, the programmer need not worry about how chip-parts are implemented. The chip-parts are used like black box abstractions, allowing the programmer to focus only on how to arrange them judiciously in order

to realize the chip function. Thanks to this modularity, HDL programs can be kept short, readable, and amenable to unit testing.

HDL-based chips like `Eq3.hdl` can be tested by a computer program called *hardware simulator*. When we instruct the simulator to evaluate a given chip, the simulator evaluates all the chip-parts specified in its PARTS section. This, in turn, requires evaluating *their* lower-level chip-parts, and so on. This recursive descent can result in a huge hierarchy of downward-expanding chip-parts, all the way down to the terminal Nand gates from which all chips are made. This expensive drill-down can be averted using *built-in chips*, as we'll explain shortly.

HDL is a declarative language: HDL programs can be viewed as textual specifications of chip diagrams. For each chip *chipName* that appears in the diagram, the programmer writes a *chipName* (...) statement in the HDL program's PARTS section. Since the language is designed to describe *connections* rather than *processes*, the order of the PARTS statements is insignificant: as long as the chip-parts are connected correctly, the chip will function as stated. The fact that HDL statements can be reordered without affecting the chip's behavior may look odd to readers who are used to conventional programming. Remember: HDL is not a programming language; it's a specification language.

White space, comments, case conventions: HDL is case-sensitive: `foo` and `Foo` represent two different things. HDL keywords are written in uppercase letters. Space characters, newline characters, and comments are ignored. The following comment formats are supported:

```
// Comment to end of line
/* Comment until closing */
/** API documentation comment */
```

Pins: HDL programs feature three types of *pins*: input pins, output pins, and internal pins. The latter pins serve to connect outputs of chip-parts to inputs of other chip-parts. Pins are assumed by default to be single-bit, carrying 0 or 1 values. Multi-bit *bus* pins can also be declared and used, as described later in this appendix.

Names of chips and pins may be any sequence of letters and digits not starting with a digit (some hardware simulators disallow using hyphens). By convention, chip and pin names start with a capital letter and a lowercase letter, respectively. For readability, names can include uppercase letters, for example, `xorResult`. HDL programs are stored in `.hdl` files. The name of the chip declared in the HDL statement `CHIP Xxx` must be identical to the prefix of the file name `Xxx.hdl`.

Program structure: An HDL program consists of an *interface* and an *implementation*. The interface consists of the chip's API documentation, chip name, and names of its input and output pins. The implementation consists of the statements below the `PARTS` keyword. The overall program structure is as follows:

```
/** API documentation: what the chip does. */
CHIP ChipName {
    IN inputPin1, inputPin2, ... ;
    OUT outputPin1, outputPin2, ... ;
PARTS:
    // Here comes the implementation.
}
```

Parts: The chip implementation is an unordered sequence of chip-part statements, as follows:

```
PARTS:
    chipPart(connection, ... , connection);
    chipPart(connection, ... , connection);
    ...
```

Each *connection* is specified using the binding `pin1 = pin2`, where *pin1* and *pin2* are input, output, or internal pin names. These connections can be visualized as “wires” that the HDL programmer creates and names, as needed. For each “wire” connecting *chipPart1* and *chipPart2* there is an internal pin that appears twice in the HDL program: once as a *sink* in some

chipPart1(...) statement, and once as a *source* in some other *chipPart2 (...)* statement. For example, consider the following statements:

```
chipPart1(..., out = v,...);           // out of chipPart1 feeds the internal pin v
chipPart2(..., in = v, ...);           // in of chipPart2 is fed from v
chipPart3(..., in1 = v, ..., in2 = v,...); // in1 and in2 of chipPart3 are also fed from v
```

Pins have fan-in 1 and unlimited fan-out. This means that a pin can be fed from a single source only, yet it can feed (through multiple connections) one or more pins in one or more chip-parts. In the above example, the internal pin *v* simultaneously feeds three inputs. This is the HDL equivalent of *forks* in chip diagrams.

The meaning of *a = a*: Many chips in the Hack platform use the same pin names. As shown in [figure A2.1](#), this leads to statements like *Xor (a=a, b=b, out=neq1)*. The first two connections feed the *a* and *b* inputs of the implemented chip (*Eq3*) into the *a* and *b* inputs of the *Xor* chip-part. The third connection feeds the *out* output of the *Xor* chip-part to the internal pin *neq1*. Here is a simple rule that helps sort things out: In every chip-part statement, the left side of each *=* binding always denotes an input or output pin *of the chip-part*, and the right side always denotes an input, output, or internal pin *of the implemented chip*.

A2.2 Multi-Bit Buses

Each input, output, or internal pin in an HDL program may be either a single-bit value, which is the default, or a multi-bit value, referred to as a *bus*.

Bit numbering and bus syntax: Bits are numbered from right to left, starting with 0. For example, *sel=110*, implies that *sel[2]=1*, *sel[1]=1*, and *sel[0]=0*.

Input and output bus pins: The bit widths of these pins are specified when they are declared in the chip's IN and OUT statements. The syntax is $x[n]$, where x and n declare the pin's name and bit width, respectively.

Internal bus pins: The bit widths of internal pins are deduced implicitly from the bindings in which they are declared, as follows,

```
chipPart1(..., x[i] = u, ...);  
chipPart2(..., x[i..j] = v, ...);
```

where x is an input or output pin of the chip-part. The first binding defines u to be a single-bit internal pin and sets its value to $x[i]$. The second binding defines v to be an internal bus-pin of width $j - i + 1$ bits and sets its value to the bits indexed i to j (inclusive) of bus-pin x .

Unlike input and output pins, internal pins (like u and v) may not be subscripted. For example, $u[i]$ is not allowed.

True/false buses: The constants true (1) and false (0) may also be used to define buses. For example, suppose that x is an 8-bit bus-pin, and consider this statement:

```
chipPart(..., x[0..2] = true, ..., x[6..7] = true, ...);
```

This statement sets x to the value 11000111. Note that unaffected bits are set by default to false (0). [Figure A2.2](#) gives another example.

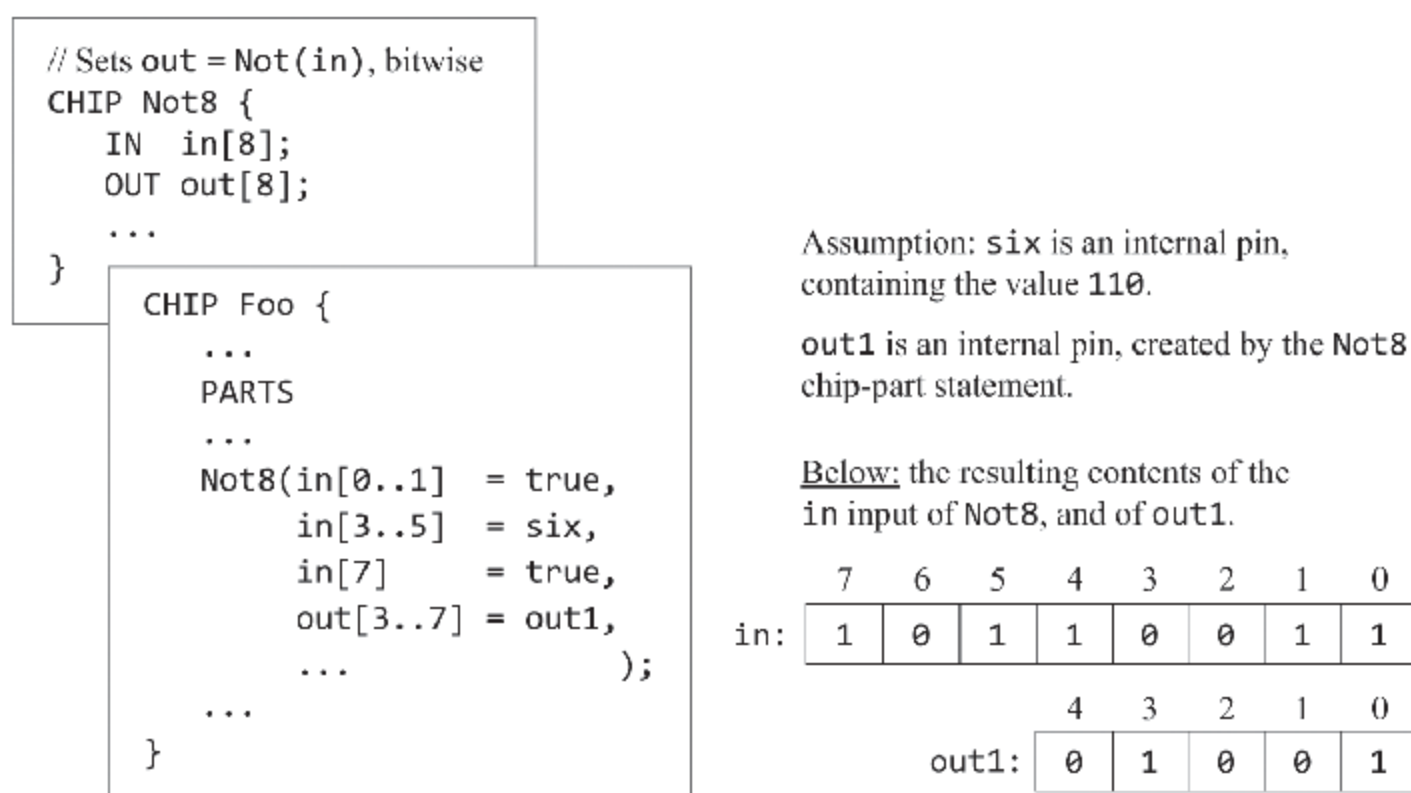


Figure A2.2 Buses in action (example).

A2.3 Built-In Chips

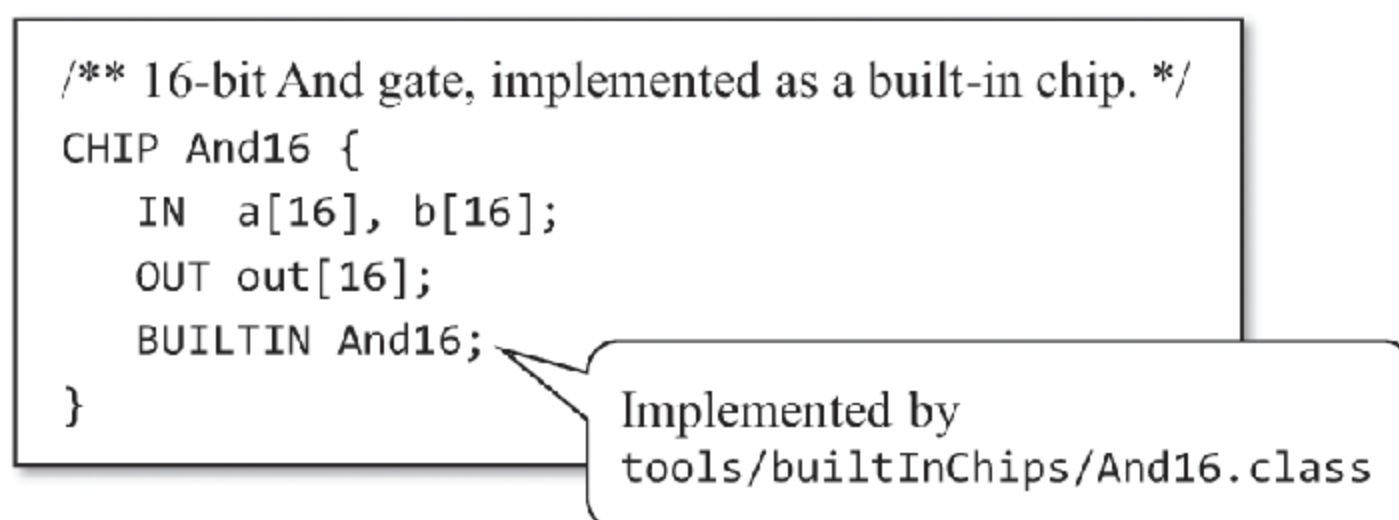
Chips can have either a *native* implementation, written in HDL, or a *built-in* implementation, supplied by an executable module written in a high-level programming language. Since the Nand to Tetris hardware simulator was written in Java, it was convenient to realize the built-in chips as Java classes. Thus, before building, say, a Mux chip in HDL, the user can load a built-in Mux chip into the hardware simulator and experiment with it. The behavior of the built-in Mux chip is supplied by a Java class file named `Mux.class`, which is part of the simulator's software.

The Hack computer is made from about thirty generic chips, listed in appendix 4. Two of these chips, Nand and DFF, are considered *given*, or *primitive*, akin to axioms in logic. The hardware simulator realizes given chips by invoking their built-in implementations. Therefore, in Nand to Tetris, Nand and DFF can be used without building them in HDL.

Projects 1, 2, 3, and 5 evolve around building HDL implementations of the remaining chips listed in appendix 4. All these chips, except for the CPU and Computer chips, also have built-in implementations. This was done in order to facilitate behavioral simulation, as explained in chapter 1.

The built-in chips—a library of about thirty `chipName.class` files—are supplied in the `nand2tetris/tools/builtInChips` folder in your computer. Built-in chips have HDL interfaces identical to those of regular HDL chips.

Therefore, each .class file is accompanied by a corresponding .hdl file that provides the built-in chip interface. Figure A2.3 shows a typical HDL definition of a built-in chip.



```
/** 16-bit And gate, implemented as a built-in chip. */
CHIP And16 {
    IN  a[16], b[16];
    OUT out[16];
    BUILTIN And16;
}
```

Implemented by
tools/builtInChips/And16.class

Figure A2.3 Built-in chip definition example.

It's important to remember that the supplied hardware simulator is a general-purpose tool, whereas the Hack computer built in Nand to Tetris is a specific hardware platform. The hardware simulator can be used for building gates, chips, and platforms that have nothing to do with Hack. Therefore, when discussing the notion of built-in chips, it helps to broaden our perspective and describe their general utility for supporting any possible hardware construction project. In general, then, built-in chips provide the following services:

Foundation: Built-in chips can provide supplied implementations of chips that are considered *given*, or *primitive*. For example, in the Hack computer, Nand and DFF are given.

Efficiency: Some chips, like RAM units, consist of numerous lower-level chips. When we use such chips as chip-parts, the hardware simulator has to evaluate them. This is done by evaluating, recursively, all the lower-level chips from which they are made. This results in slow and inefficient simulation. The use of built-in chip-parts instead of regular, HDL-based chips speeds up the simulation considerably.

Unit testing: HDL programs use chip-parts abstractly, without paying attention to their implementation. Therefore, when building a new chip, it is

always recommended to use built-in chip-parts. This practice improves efficiency and minimizes errors.

Visualization: If the designer wants to allow users to “see” how chips work, and perhaps change the internal state of the simulated chip interactively, he or she can supply a built-in chip implementation that features a graphical user interface. This GUI will be displayed whenever the built-in chip is loaded into the simulator or invoked as a chip-part. Except for these visual side effects, GUI-empowered chips behave, and can be used, just like any other chip. Section A2.5 provides more details about GUI-empowered chips.

Extension: If you wish to implement a new input/output device or create a new hardware platform altogether (other than Hack), you can support these constructions with built-in chips. For more information about developing additional or new functionality, see chapter 13.

A2.4 Sequential Chips

Chips can be either *combinational* or *sequential*. Combinational chips are time independent: they respond to changes in their inputs instantaneously. Sequential chips are time dependent, also called *clocked*: when a user or a test script changes the inputs of a sequential chip, the chip outputs may change only at the beginning of the next *time unit*, also called a *cycle*. The hardware simulator effects the progression of time using a simulated clock.

The clock: The simulator’s two-phase clock emits an infinite sequence of values denoted $0, 0+, 1, 1+, 2, 2+, 3, 3+$, and so on. The progression of this discrete time series is controlled by two simulator commands called *tick* and *tock*. A *tick* moves the clock value from t to $t+$, and a *tock* from $t+$ to $t+1$, bringing upon the next time unit. The *real time* that elapsed during this period is irrelevant for simulation purposes, since the simulated time is controlled by the user, or by a test script, as follows.

First, whenever a sequential chip is loaded into the simulator, the GUI enables a clock-shaped button (dimmed when simulating combinational

chips). One click on this button (a tick) ends the first phase of the clock cycle, and a subsequent click (a tock) ends the second phase of the cycle, bringing on the first phase of the next cycle, and so on.

Alternatively, one can run the clock from a test script. For example, the sequence of scripting commands `repeat n {tick, tock, output}` instructs the simulator to advance the clock n time units and to print some values in the process. Appendix 3 documents the *Test Description Language* (TDL) that features these commands.

The two-phased time units generated by the clock regulate the operations of all the sequential chip-parts in the implemented chip. During the first phase of the time unit (tick), the inputs of each sequential chip-part affect the chip’s internal state, according to the chip logic. During the second phase of the time unit (tock), the chip outputs are set to the new values. Hence, if we look at a sequential chip “from the outside,” we see that its output pins stabilize to new values only at tock—at the point of transition between two consecutive time units.

We reiterate that combinational chips are completely oblivious to the clock. In Nand to Tetris, all the logic gates and chips built in chapters 1–2, up to and including the ALU, are combinational. All the registers and memory units built in chapter 3 are sequential. By default, chips are combinational; a chip can become *sequential* explicitly or implicitly, as follows.

Sequential, built-in chips: A *built-in chip* can declare its dependence on the clock explicitly, using the statement,

```
CLOCKED pin, pin, ..., pin;
```

where each *pin* is one of the chip’s input or output pins. The inclusion of an input pin x in the CLOCKED list stipulates that changes to x should affect the chip’s outputs only at the beginning of the next time unit. The inclusion of an output pin x in the CLOCKED list stipulates that changes in any of the chip’s inputs should affect x only at the beginning of the next time unit. [Figure A2.4](#) presents the definition of the most basic, built-in, sequential chip in the Hack platform—the DFF.

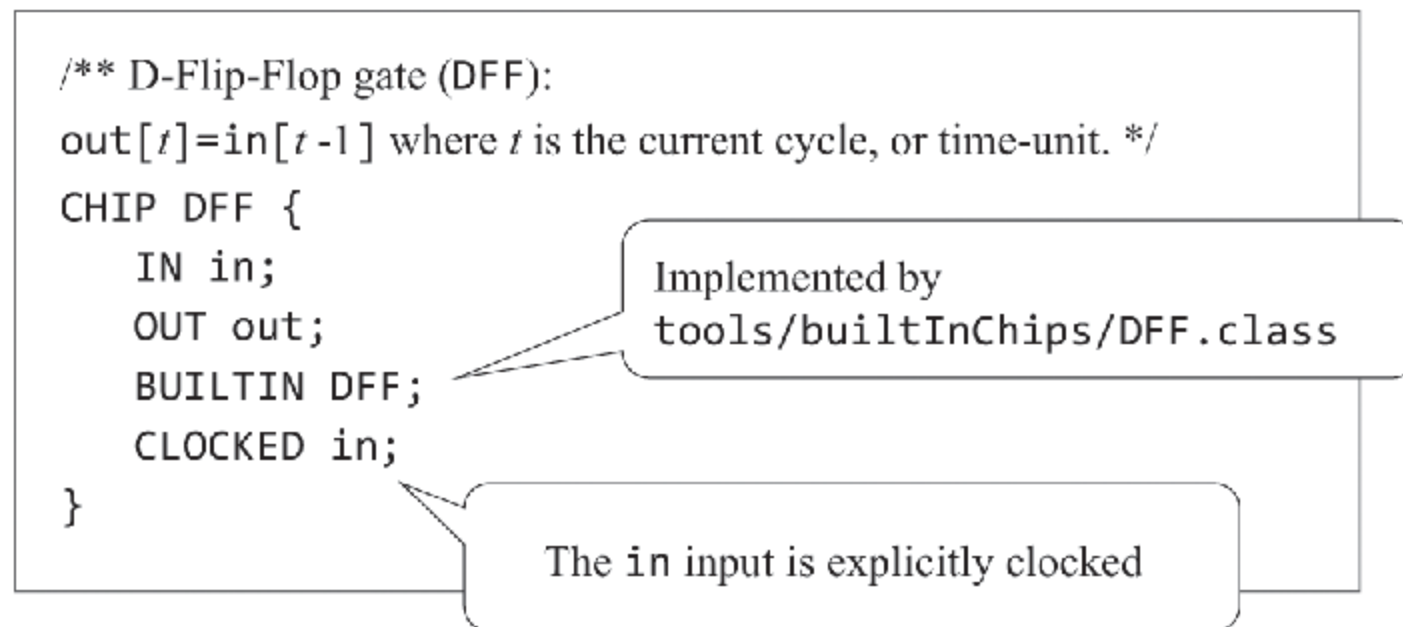


Figure A2.4 DFF definition.

It is possible that only some of the input or output pins of a chip are declared as clocked. In that case, changes in the non-clocked input pins affect the non-clocked output pins instantaneously. That's how the address pins are implemented in RAM units: the addressing logic is combinational and independent of the clock.

It is also possible to declare the CLOCKED keyword with an empty list of pins. This statement stipulates that the chip may change its internal state depending on the clock, but its input-output behavior will be combinational, independent of the clock.

Sequential, composite chips: The CLOCKED property can be defined explicitly only in built-in chips. How, then, does the simulator know that a given chip-part is sequential? If the chip is not built-in, then it is said to be clocked when one or more of its chip-parts is clocked. The clocked property is checked recursively, all the way down the chip hierarchy, where a built-in chip may be explicitly clocked. If such a chip is found, it renders every chip that depends on it (up the hierarchy) “clocked.” Therefore, in the Hack computer, all the chips that include one or more DFF chip-parts, either directly or indirectly, are clocked.

We see that if a chip is not built-in, there is no way to tell from its HDL code whether it is sequential or not. *Best-practice advice:* The chip architect should provide this information in the chip API documentation.

Feedback loops: If the input of a chip feeds from one of the chip's outputs, either directly or through a (possibly long) path of dependencies, we say

that the chip contains a *feedback loop*. For example, consider the following two chip-part statements:

```
Not (in=loop1, out=loop1) // Invalid feedback loop
DFF (in=loop2, out=loop2) // Valid feedback loop
```

In both examples, an internal pin (loop1 or loop2) attempts to feed the chip's input from its output, creating a feedback loop. The difference between the two examples is that Not is a combinational chip, whereas DFF is sequential. In the Not example, loop1 creates an instantaneous and uncontrolled dependency between in and out, sometimes called a *data race*. In contrast, in the DFF case, the in-out dependency created by loop2 is delayed by the clock, since the in input of the DFF is declared clocked. Therefore, $out(t)$ is not a function of $in(t)$ but rather of $in(t - 1)$.

When the simulator evaluates a chip, it checks recursively whether its various connections entail feedback loops. For each loop, the simulator checks whether the loop goes through a clocked pin somewhere along the way. If so, the loop is allowed. Otherwise, the simulator stops processing and issues an error message. This is done to prevent uncontrolled data races.

A2.5 Visualizing Chips

Built-in chips may be *GUI empowered*. These chips feature visual side effects designed to animate some of the chip operations. When the simulator evaluates a GUI-empowered chip-part, it displays a graphical image on the screen. Using this image, which may include interactive elements, the user can inspect the chip's current state or change it. The permissible GUI-empowered actions are determined, and made possible, by the developer of the built-in chip implementation.

The present version of the hardware simulator features the following GUI-empowered, built-in chips:

ALU: Displays the Hack ALU's inputs, output, and the presently computed function.

Registers (ARegister, DRegister, PC): Displays the register's contents, which may be modified by the user.

RAM chips: Displays a scrollable, array-like image that shows the contents of all the memory locations, which may be modified by the user. If the contents of a memory location change during the simulation, the respective entry in the GUI changes as well.

ROM chip (ROM32K): Same array-like image as that of RAM chips, plus an icon that enables loading a machine language program from an external text file. (The ROM32K chip serves as the instruction memory of the Hack computer.)

Screen chip: Displays a 256-rows-by-512-columns window that simulates the physical screen. If, during a simulation, one or more bits in the RAM-resident *screen memory map* change, the respective pixels in the screen GUI change as well. This continuous refresh loop is embedded in the simulator implementation.

Keyboard chip: Displays a keyboard icon. Clicking this icon connects the real keyboard of your computer to the simulated chip. From this point on, every key pressed on the real keyboard is intercepted by the simulated chip, and its binary code appears in the RAM-resident *keyboard memory map*. If the user moves the mouse focus to another area in the simulator GUI, the control of the keyboard is restored to the real computer.

[Figure A2.5](#) presents a chip that uses three GUI empowered chip-parts. [Figure A2.6](#) shows how the simulator handles this chip. The GUIDemo chip logic feeds its in input into two destinations: register number address in the RAM16K chip-part, and register number address in the Screen chip-part. In addition, the chip logic feeds the out values of its three chip-parts to the “dead-end” internal pins a, b, and c. These meaningless connections are designed for one purpose only: illustrating how the simulator deals with built-in, GUI-empowered chip-parts.

```
// Demo of GUI-empowered chips.
// The logic of this chip is meaningless, and is used merely to force
// the simulator to display the GUI effects of its built-in chip-parts.

CHIP GUIDemo {
    IN in[16], load, address[15];
    OUT out[16];
    PARTS:
    RAM16K(in=in, load=load, address=address[0..13], out=a);
    Screen(in=in, load=load, address=address[0..12], out=b);
    Keyboard(out=c);
}
```

Figure A2.5 A chip that activates GUI-empowered chip-parts.

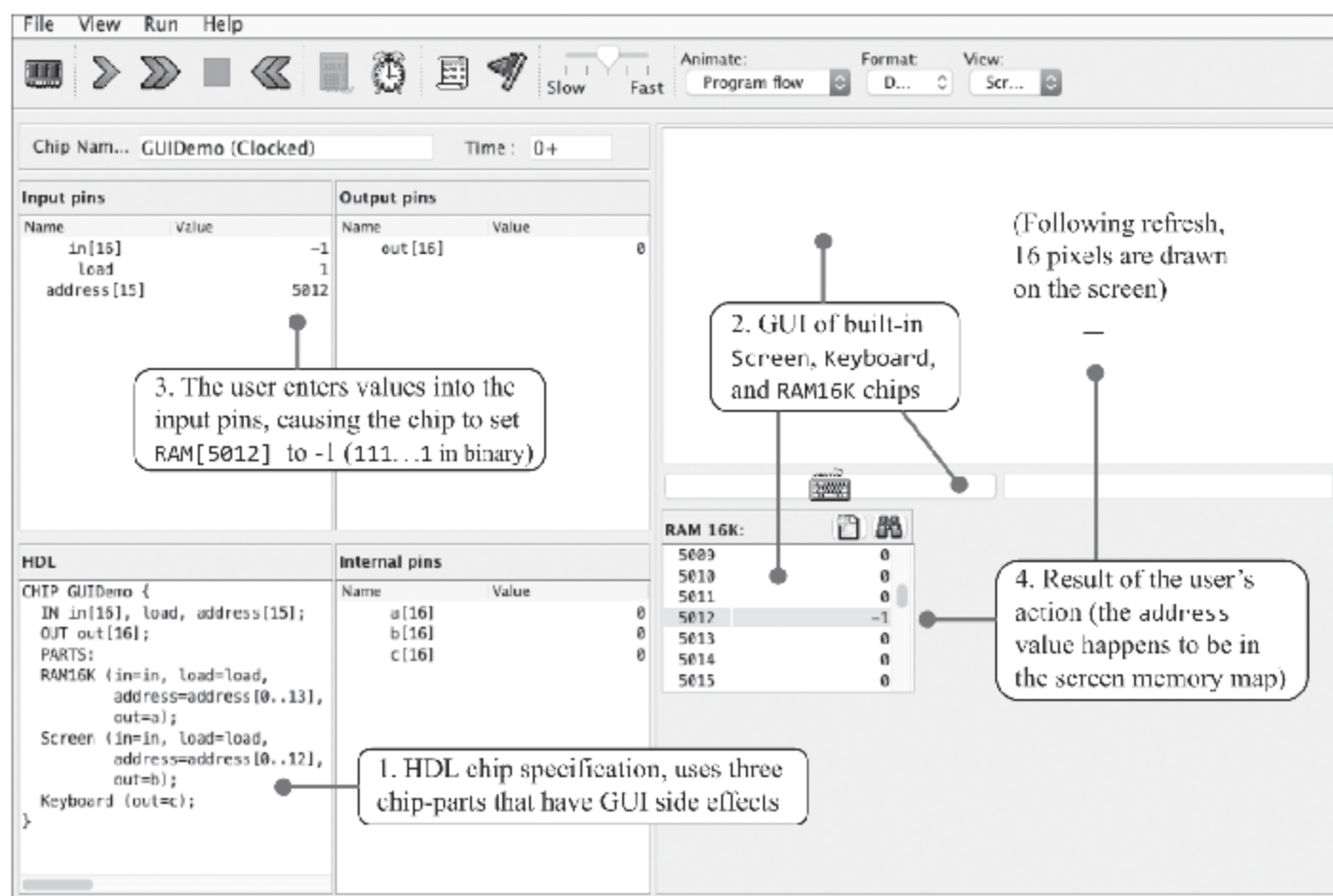


Figure A2.6 GUI-empowered chips demo. Since the loaded HDL program uses GUI-empowered chip-parts (step 1), the simulator renders their respective GUI images (step 2). When the user changes the values of the chip input pins (step 3), the simulator reflects these changes in the respective GUIs (step 4).

Note how the changes effected by the user (step 3) impact the screen (step 4). The circled horizontal line shown on the screen is the visual side effect of storing -1 in memory location 5012. Since the 16-bit two's complement binary code of -1 is 1111111111111111, the computer draws 16 pixels starting at column 320 of row 156, which happen to be the screen

coordinates associated with RAM address 5012. The mapping of memory *addresses* on (*row*, *column*) screen coordinates is specified in chapter 4 (section 4.2.5).

A2.6 HDL Survival Guide

This section provides practical tips about how to develop chips in HDL using the supplied hardware simulator. The tips are listed in no particular order. We recommend reading this section once, beginning to end, and then consulting it as needed.

Chip: Your `nand2tetris/projects` folder includes thirteen subfolders, named 01, 02, ..., 13 (corresponding to the relevant chapter numbers). The hardware project folders are 01, 02, 03, and 05. Each hardware project folder contains a set of supplied HDL *stub files*, one for each chip that you have to build. The supplied HDL files contain no implementations; building these implementations is what the project is all about. If you do not build these chips in the order in which they are described in the book, you may run into difficulties. For example, suppose that you start project 1 by building the Xor chip. If your `Xor.hdl` implementation includes, say, `And` and `Or` chip-parts, and you have not yet implemented `And.hdl` and `Or.hdl`, your `Xor.hdl` program will not work even if its implementation is perfectly correct.

Note, however, that if the project folder included no `And.hdl` and `Or.hdl` files, your `Xor.hdl` program will work properly. The hardware simulator, which is a Java program, features built-in implementations of all the chips necessary to build the Hack computer (with the exception of the CPU and Computer chips). When the simulator evaluates a chip-part, say `And`, it looks for an `And.hdl` file in the current folder. At this point there are three possibilities:

- No HDL file is found. In this case, the built-in implementation of the chip kicks in, covering for the missing HDL implementation.
- A stub HDL file is found. The simulator tries to execute it. Failing to find an implementation, the execution fails.

- An HDL file is found, with an HDL implementation. The simulator executes it, reporting errors, if any, to the best of its ability.

Best-practice advice: You can do one of two things. Try to implement the chips in the order presented in the book and in the project descriptions. Since the chips are discussed bottom-up, from basic chips to more complex ones, you will encounter no chip order implementation troubles—provided, of course, that you complete each chip implementation correctly before moving on to implement the next one.

A recommended alternative is to create a subfolder named, say, stubs, and move all the supplied .hdl stub files into it. You can then move each stub file that you want to work on into your working folder, one by one. When you are done implementing a chip successfully, move it into, say, a completed subfolder. This practice forces the simulator to always use built-in chips, since the working folder includes only the .hdl file that you are working on (as well as the supplied .tst and .cmp files).

HDL files and test scripts: The .hdl file that you are working on and its associated .tst test script file must be located in the same folder. Each supplied test script starts with a load command that loads the .hdl file that it is supposed to test. The simulator always looks for this file in the current folder.

In principle, the simulator's File menu allows the user to load, interactively, both an .hdl file and a .tst script file. This can create potential problems. For example, you can load the .hdl file that you are working on into the simulator, and then load a test script from another folder. When you execute the test script, it may well load a different version of the HDL program into the simulator (possibly, a stub file). When in doubt, inspect the pane named HDL in the simulator GUI to check which HDL code is presently loaded. *Best-practice advice:* Use the simulator's File menu to load either an .hdl file or a .tst file, but not both.

Testing chips in isolation: At some point you may become convinced that your chip is correct, even though it is still failing the test. Indeed, it is possible that the chip is perfectly implemented, but one of its chip-parts is not. Also, a chip that passed its test successfully may fail when used as a

chip-part by another chip. One of the biggest inherent limitations of hardware design is that test scripts—especially those that test complex chips—cannot guarantee that the tested chip will operate perfectly in all circumstances.

The good news is that you can always diagnose which chip-part is causing the problem. Create a test subfolder and copy into it only the three .hdl, .tst, and .out files related to the chip that you are presently building. If your chip implementation passes its test in this subfolder as is (letting the simulator use the default built-in chip-parts), there must be a problem with one of your chip-part implementations, that is, with one of the chips that you built earlier in this project. Copy the other chips into this test folder, one by one, and repeat the test until you find the problematic chip.

HDL syntax errors: The hardware simulator displays errors on the bottom status bar. On computers with small screens, these messages are sometimes off the bottom of the screen, not visible. If you load an HDL program and nothing shows up in the HDL pane, but no error message is seen, this may be the problem. Your computer should have a way to move the window, using the keyboard. For example, on Windows use Alt+Space, M, and the arrow keys.

Unconnected pins: The hardware simulator does not consider unconnected pins to be errors. By default, it sets any unconnected input or output pin to false (binary value 0). This can cause mysterious errors in your chip implementations.

If an output pin of your chip is always 0, make sure that it is properly connected to another pin in your program. In particular, double-check the names of the internal pins (“wires”) that feed this pin, either directly or indirectly. Typographic errors are particularly hazardous here, since the simulator doesn’t throw errors on disconnected wires. For example, consider the statement `Foo(..., sum=sum)`, where the sum output of Foo is supposed to pipe its value to an internal pin. Indeed, the simulator will happily create an internal pin named sum. Now, if sum’s value was supposed to feed the output pin of the implemented chip, or the input pin of another chip-part, this pin will in fact be 0, *always*, since nothing will be piped from Foo onward.

To recap, if an output pin is always 0, or if one of the chip-parts does not appear to be working correctly, check the spelling of all the relevant pin names, and verify that all the input pins of the chip-part are connected.

Customized testing: For every *chip.hdl* file that you have to complete your project folder also includes a supplied test script, named *chip.tst*, and a compare file, named *chip.cmp*. Once your chip starts generating outputs, your folder will also include an output file named *chip.out*. If your chip fails the test script, don't forget to consult the *.out* file. Inspect the listed output values, and seek clues to the failure. If for some reason you can't see the output file in the simulator GUI, you can always inspect it using a text editor.

If you want, you can run tests of your own. Copy the supplied test script to, say, *MyTestChip.tst*, and modify the script commands to gain more insight into your chip's behavior. Start by changing the name of the output file in the output-file line and deleting the compare-to line. This will cause the test to always run to completion (by default, the simulation stops when an output line disagrees with the corresponding line in the compare file). Consider modifying the output-list line to show the outputs of your internal pins.

Appendix 3 documents the Test Description Language (TDL) that features all these commands.

Sub-busing (indexing) internal pins: This is not permitted. The only bus-pins that can be indexed are the input and output pins of the implemented chip or the input and output pins of its chip-parts. However, there is a workaround for sub-busing internal bus-pins. To motivate the work-around, here is an example that doesn't work:

```
CHIP Foo {
  IN  in[16];
  OUT out;
  PARTS:
  Not16 (in=in, out=notIn);
  Or8Way (in=notIn[4..11], out=out); // Error: internal bus cannot be indexed.
}
```


Possible fix, using the work-around:

```
Not16 (in=in, out[4..11]=notIn);  
Or8Way (in=notIn, out=out); // Works!
```

Multiple outputs: Sometimes you need to split the multi-bit value of a bus-pin into two buses. This can be done by using multiple out= bindings.

For example:

```
CHIP Foo {  
    IN  in[16];  
    OUT out[8];  
    PARTS:  
    Not16 (in=in, out[0..7]=low8, out[8..15]=high8); // Splitting the out value  
    Bar8Bit (a=low8, b=high8, out=out);  
}
```

Sometimes you may want to output a value and also use it for further computations. This can be done as follows:

```
CHIP Foo {  
    IN  a, b, c;  
    OUT out1, out2;  
    PARTS:  
    Bar (a=a, b=b, out=x, out=out1); // Bar's output feeds the out1 output of Foo  
    Baz (a=x, b=c, out=out2);         // A copy of Bar's output also feeds Baz's a input  
}
```

Chip-parts “auto complete” (sort of): The signatures of all the chips mentioned in this book are listed in appendix 4, which also has a web-based version (at www.nand2tetris.org). To use a chip-part in a chip implementation, copy-paste the chip signature from the online document into your HDL program, then fill in the missing bindings. This practice saves time and minimizes typing errors.

Appendix 3: Test Description Language

Mistakes are the portals of discovery.

—James Joyce (1882–1941)

Testing is a critically important element of systems development, and one that typically gets insufficient attention in computer science education. In Nand to Tetris we take testing very seriously. We believe that before one sets out to develop a new hardware or software module P , one must first develop a module T designed to test it. Further, T should be part of P 's development's contract. Therefore, for every chip or software system specified in this book, we supply official test programs, written by us. Although you are welcome to test your work in any way you see fit, the contract is such that, ultimately, your implementation must pass *our* tests.

In order to streamline the definition and execution of the numerous tests scattered all over the book projects, we designed a uniform test description language. This language works almost the same across all the relevant tools supplied in Nand to Tetris: the *hardware simulator* used for simulating and testing chips written in HDL, the *CPU emulator* used for simulating and testing machine language programs, and the *VM emulator* used for simulating and testing programs written in the VM language, which are typically compiled Jack programs.

Every one of these simulators features a GUI that enables testing the loaded chip or program interactively, or batch-style, using a test script. A test script is a sequence of commands that load a hardware or software module into the relevant simulator and subject the module to a series of preplanned testing scenarios. In addition, the test scripts feature commands

for printing the test results and comparing them to desired results, as defined in supplied compare files. In sum, a test script enables a systematic, replicable, and documented testing of the underlying code—an invaluable requirement in any hardware or software development project.

In Nand to Tetris, we don't expect learners to write test scripts. All the test scripts necessary to test all the hardware and software modules mentioned in the book are supplied with the project materials. Thus, the purpose of this appendix is not to teach you how to write test scripts but rather to help you understand the syntax and logic of the supplied test scripts. Of course, you are welcome to customize the supplied scripts and create new ones, as you please.

A3.1 General Guidelines

The following usage guidelines are applicable to all the software tools and test scripts.

File format and usage: The act of testing a hardware or software module involves four types of files. Although not required, we recommend that all four files have the same prefix (file name):

Xxx.yyy: where *Xxx* is the name of the tested module and *yyy* is either *hdl*, *hack*, *asm*, or *vm*, standing, respectively, for a chip definition written in HDL, a program written in the Hack machine language, a program written in the Hack assembly language, or a program written in the VM virtual machine language

Xxx.tst: a test script that walks the simulator through a series of steps designed to test the code stored in *Xxx*

Xxx.out: an optional output file to which the script commands can write current values of selected variables during the simulation

Xxx.cmp: an optional compare file containing the *desired* values of selected variables, that is, the values that the simulation *should* generate if the module is implemented correctly

All these files should be kept in the same folder, which can be conveniently named *Xxx*. In the documentation and descriptions of all the simulators, the term “current folder” refers to the folder from which the last file has been opened in the simulator environment.

White space: Space characters, newline characters, and comments in test scripts (*Xxx.tst* files) are ignored. The following comment formats can appear in test scripts:

```
// Comment to end of line
/* Comment until closing */
/** API documentation comment */
```

Test scripts are not case-sensitive, except for file and folder names.

Usage: For each hardware or software module *Xxx* in Nand to Tetris we supply a script file *Xxx.tst* and a compare file *Xxx.cmp*. These files are designed to test your implementation of *Xxx*. In some cases, we also supply a skeletal version of *Xxx*, for example, an HDL interface with a missing implementation. All the files in all the projects are plain text files that should be viewed and edited using plain text editors.

Typically, you will start a simulation session by loading the supplied *Xxx.tst* script file into the relevant simulator. The first command in the script typically loads the code stored in the tested module *Xxx*. Next, optionally, come commands that initialize an output file and specify a compare file. The remaining commands in the script run the actual tests.

Simulation controls: Each one of the supplied simulators features a set of menus and icons for controlling the simulation.

File menu: Allows loading into the simulator either a relevant program (.hdl file, .hack file, .asm file, .vm file, or a folder name) or a test script (.tst file). If the user does not load a test script, the simulator loads a default test script (described below).

Play icon: Instructs the simulator to execute the next simulation step, as specified in the currently loaded test script.

Pause icon: Instructs the simulator to pause the execution of the currently loaded test script. Useful for inspecting various elements of the simulated environment.

Fast-forward icon: Instructs the simulator to execute all the commands in the currently loaded test script.

Stop icon: Instructs the simulator to stop the execution of the currently loaded test script.

Rewind icon: Instructs the simulator to reset the execution of the currently loaded test script, that is, be ready to start executing the test script from its first command onward.

Note that the simulator's icons listed above don't "run the code." Rather, they run the test script, which runs the code.

A3.2 Testing Chips on the Hardware Simulator

The supplied hardware simulator is designed for testing and simulating chip definitions written in the Hardware Description Language (HDL) described in appendix 2. Chapter 1 provides essential background on chip development and testing; thus, we recommend reading it first.

Example: [Figure A2.1](#) in appendix 2 presents an Eq3 chip, designed to check whether three 1-bit inputs are equal. [Figure A3.1](#) presents Eq3.tst, a script designed to test the chip, and Eq3.cmp, a compare file containing the expected output of this test.

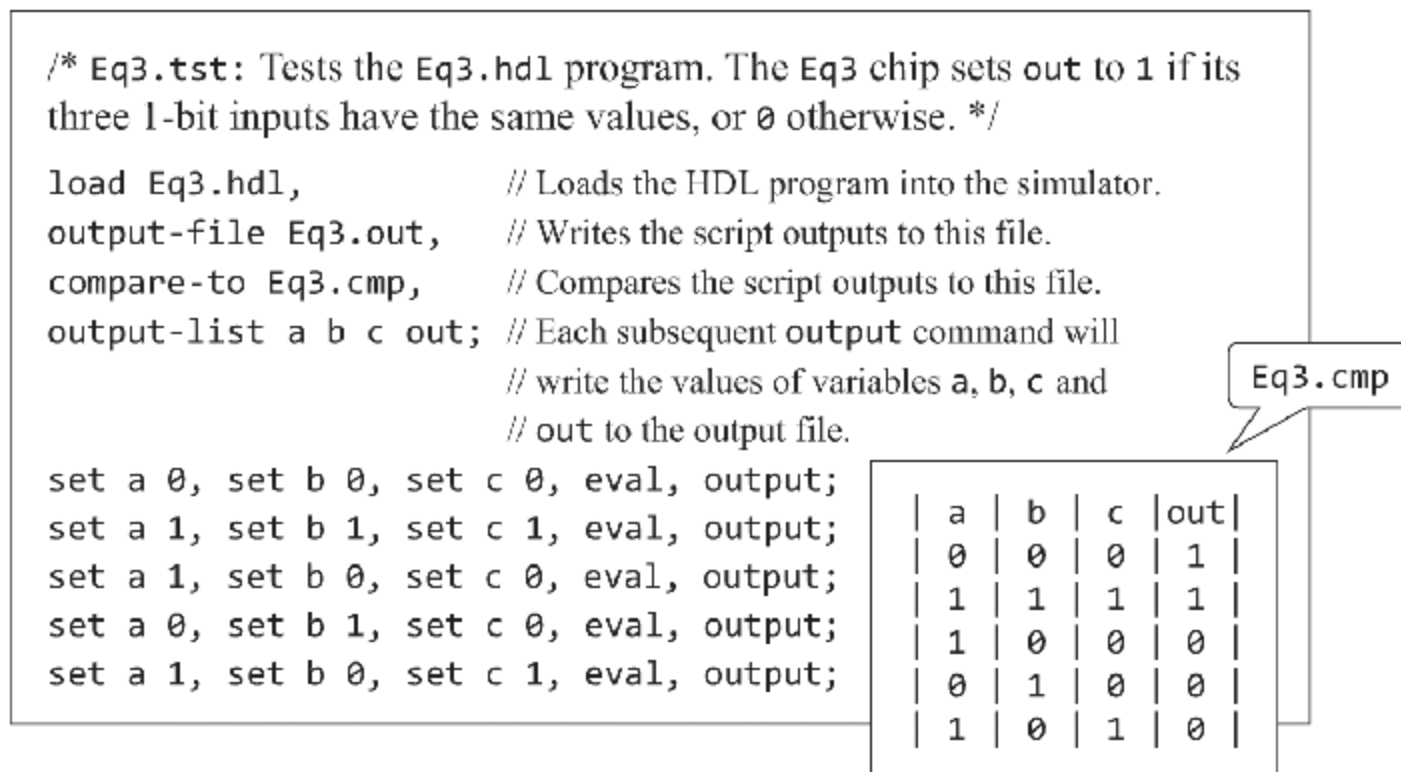


Figure A3.1 Test script and compare file (example).

A test script normally starts with some setup commands, followed by a series of simulation steps, each ending with a semicolon. A simulation step typically instructs the simulator to bind the chip's input pins to test values, evaluate the chip logic, and write selected variable values into a designated output file.

The Eq3 chip has three 1-bit inputs; thus, an exhaustive test would require eight testing scenarios. The size of an exhaustive test grows exponentially with the input size. Therefore, most test scripts test only a subset of representative input values, as shown in the figure.

Data types and variables: Test scripts support two data types: *integers* and *strings*. Integer constants can be expressed in decimal (%D prefix) format, which is the default, binary (%B prefix) format, or hexadecimal (%X prefix) format. These values are always translated into their equivalent two's complement binary values. For example, consider the following commands:

```
set a1 %B1111111111111111
set a2 %XFFFF
set a3 %D-1
set a4 -1
```

All four variables are set to the same value: 1111111111111111 in binary, which happens to be the binary, two's complement representation of -1 in decimal.

String values are specified using a %S prefix and must be enclosed by double quotation marks. Strings are used strictly for printing purposes and cannot be assigned to variables.

The hardware simulator's two-phase clock (used only in testing sequential chips) emits a series of values denoted 0, 0+, 1, 1+, 2, 2+, 3, 3+, and so on. The progression of these *clock cycles* (also called *time units*) can be controlled by two script commands named tick and tock. A tick moves the clock value from t to $t+$, and a tock from $t+$ to $t+1$, bringing upon the next time unit. The current time unit is stored in a system variable named time, which is read-only.

Script commands can access three types of variables: pins, variables of built-in chips, and the system variable time.

Pins: input, output, and internal pins of the simulated chip (for example, the command set in 0 sets the value of the pin whose name is in to 0)

Variables of built-in chips: exposed by the chip's external implementation (see [figure A3.2](#).)

Chip name	Exposed variables	Data type / range	Methods
Register	Register[]	16-bit (-32768...32767)	load Xxx.hack / Xxx.asm
ARegister	ARegister[]	16-bit	
DRegister	DRegister[]	16-bit	
PC (prog. counter)	PC[]	15-bit (0...32767)	
RAM8	RAM8[0...7]	each entry is 16-bit	
RAM64	RAM64[0...63]	each entry is 16-bit	
RAM512	RAM512[0...511]	each entry is 16-bit	
RAM4K	RAM4K[0...4095]	each entry is 16-bit	
RAM16K	RAM16K[0...16383]	each entry is 16-bit	
ROM32K	ROM32K[0...32767]	each entry is 16-bit	
Screen	Screen[0...16383]	each entry is 16-bit	
Keyboard	Keyboard[]	16-bit, read-only	

Figure A3.2 Variables and methods of key built-in chips in Nand to Tetris.

time: the number of time-units that elapsed since the simulation started (a read-only variable)

Script commands: A script is a sequence of commands. Each command is terminated by a comma, a semicolon, or an exclamation mark. These terminators have the following semantics:

Comma (,): Terminates a script command.

Semicolon (;): Terminates a script command and a simulation step. A *simulation step* consists of one or more script commands. When the user instructs the simulator to *single-step* using the simulator's menu or play icon, the simulator executes the script from the current command until a semicolon is reached, at which point the simulation is paused.

Exclamation mark (!): Terminates a script command and stops the script execution. The user can later resume the script execution from that point onward. Typically used for debugging purposes.

Below we group the script commands in two conceptual sections: *setup commands*, used for loading files and initializing settings, and *simulation commands*, used for walking the simulator through the actual tests.

Setup Commands

load *Xxx*.hdl: Loads the HDL program stored in *Xxx*.hdl into the simulator. The file name must include the .hdl extension and must not include a path specification. The simulator will try to load the file from the current folder and, failing that, from the tools/builtInChips folder.

output-file *Xxx*.out: Instructs the simulator to write the results of the output commands in the named file, which must include an .out extension. The output file will be created in the current folder.

output-list *v1*, *v2*, ...: Specifies what to write to the output file when the output command is encountered in the script (until the next output-list command, if any). Each value in the list is a variable name followed by a formatting specification. The command also produces a single header line, consisting of the variable names, which is written to the output file. Each item *v* in the output-list has the syntax *varName format padL.len.padR* (without any spaces). This directive instructs the simulator to write *padL* space characters, then the current value of the variable *varName*, using the

specified *format* and *len* columns, then *padR* spaces, and finally the divider symbol |. The *format* can be either %B (binary), %X (hexa), %D (decimal), or %S (string). The default format is %B1.1.1.

For example, the CPU.hdl chip of the Hack platform has an input pin named reset, an output pin named pc (among others), and a chip-part named DRegister (among others). If we want to track the values of these entities during the simulation, we can use something like the following command:

```
Output-list time%S1.5.1    // The system variable time
              reset%B2.1.2  // One of the chip's input pins
              pc%D2.3.1     // One of the chip's output pins
              DRegister[]%X3.4.4 // The internal state of this chip-part
```

(State variables of built-in chips are explained below). This output-list command may end up producing the following output, after two subsequent output commands:

	time		reset		pc		DRegister[]	
	20+		0		21		FFFF	
	21		0		22		FFFF	

compare-to Xxx.cmp: Specifies that the output line generated by each subsequent output command should be compared to its corresponding line in the specified compare file (which must include the .cmp extension). If any two lines are not the same, the simulator displays an error message and halts the script execution. The compare file is assumed to be present in the current folder.

Simulation Commands

set varName value: Assigns the value to the variable. The variable is either a pin or an internal variable of the simulated chip or one of its chip-parts. The bit widths of the value and the variable must be compatible.

eval: Instructs the simulator to apply the chip logic to the current values of the input pins and compute the resulting output values.

output: Causes the simulator to go through the following logic:

1. Get the current values of all the variables listed in the last output-list command.
2. Create an output line using the format specified in the last output-list command.
3. Write the output line to the output file.
4. (If a compare file has been previously declared using a compare-to command): If the output line differs from the compare file's current line, display an error message and stop the script's execution.
5. Advance the line cursors of the output file and the compare file.

tick: Ends the first phase of the current time unit (clock cycle).

tock: Ends the second phase of the current time unit and embarks on the first phase of the next time unit.

repeat *n* {*commands*}: Instructs the simulator to repeat the commands enclosed in the curly brackets *n* times. If *n* is omitted, the simulator repeats the commands until the simulation has been stopped for some reason (for example, when the user clicks the Stop icon).

while *booleanCondition* {*commands*}: Instructs the simulator to repeat the commands enclosed in the curly brackets as long as the *booleanCondition* is true. The condition is of the form *x op y* where *x* and *y* are either constants or variable names and *op* is =, >, <, >=, <=, or <>. If *x* and *y* are strings, *op* can be either = or <>.

echo *text*: Displays the *text* in the simulator status line. The text must be enclosed in double quotation marks.

clear-echo: Clears the simulator's status line.