6 Assembler

What's in a name? That which we call a rose by any other name would smell as sweet.
—Shakespeare, from *Romeo and Juliet*

The first half of the book (chapters 1–5) described and built a computer's *hardware platform*. The second half of the book (chapters 6–12) focuses on the computer's *software hierarchy*, culminating in the development of a compiler and a basic operating system for a simple, object-based programming language. The first and most basic module in this software hierarchy is the *assembler*. In particular, chapter 4 presented machine languages in both their *assembly* and *binary* representations. This chapter describes how assemblers can systematically translate programs written in the former into programs written in the latter. As the chapter unfolds, we explain how to develop a *Hack assembler*—a program that generates binary code that can run as is on the hardware platform built in chapter 5.

Since the relationship between symbolic assembly commands and their corresponding binary codes is straightforward, writing an assembler (using some high-level language) is not a difficult task. One complication arises from allowing assembly programs to use symbolic references to memory addresses. The assembler is expected to manage these user-defined symbols and resolve them to physical memory addresses. This task is normally done using a *symbol table*—a classical data structure that comes to play in many software translation projects.

As usual, the Hack assembler is not an end in itself. Rather, it provides a simple and concise demonstration of the key software engineering principles used in the construction of any assembler. Further, writing the assembler is the first in the series of seven software development projects that accompany the rest of the book. Unlike the hardware projects, which were implemented in HDL, the software projects that construct the translator programs (*assembler*, *virtual machine*, and *compiler*) may be implemented in any programming language. In each project, we provide a language-neutral API and a detailed step-by-step test plan, along with all the necessary test

programs and test scripts. Each one of these projects, beginning with the assembler, is a stand-alone module that can be developed and tested in isolation from all the other projects.

6.1 Background

Machine languages are typically specified in two flavors: *symbolic* and *binary*. The binary codes—for example, `1100001010000001100000000000000111`—represent actual machine instructions, as understood by the underlying hardware. For example, the instruction's leftmost 8 bits can represent an operation code, say `LOAD`, the next 8 bits a register, say `R3`, and the remaining 16 bits an address, say `7`. Depending on the hardware's logic design and the agreed-upon machine language, the overall 32-bit pattern can thus cause the hardware to effect the operation "load the contents of Memory[7] into register R3." Modern computer platforms support dozens if not hundreds of such elementary operations. Thus, machine languages can be rather complex, involving many operation codes, different memory addressing modes, and various instruction formats.

One way to cope with this complexity is to document machine instructions using an agreed-upon syntax, say `LOAD R3,7` rather than `110000101000000110 000000000000111`. And since the translation from symbolic notation to binary code is straightforward, it makes sense to allow low-level programs to be written in symbolic notation and to have a computer program translate them into binary code. The symbolic language is called *assembly*, and the translator program *assembler*. The assembler parses each assembly command into its underlying fields, translates each field into its equivalent binary code, and assembles the generated codes into a binary instruction that can be actually executed by the hardware.

Symbols Binary instructions are represented in binary code. By definition, they refer to memory addresses using actual numbers. For example, consider a program that uses a variable to represent the *weight* of various things, and suppose that this variable has been mapped on location 7 in the computer's memory. At the binary code level, instructions that manipulate the *weight* variable must refer to it using the explicit address 7. Yet once we step up to the assembly level, we can allow writing commands like `LOAD R3,weight` instead of `LOAD R3,7`. In both cases, the command will effect the same operation: "set R3 to the contents of Memory[7]." In a similar fashion, rather than using commands like `goto 250`, assembly languages allow commands like `goto loop`, assuming that somewhere in the program the symbol `loop` is

made to refer to address 250. In general then, symbols are introduced into assembly programs from two sources:

- *Variables:* The programmer can use symbolic variable names, and the translator will "automatically" assign them to memory addresses. Note that the actual values of these addresses are insignificant, so long as each symbol is resolved to the same address throughout the program's translation.
- *Labels:* The programmer can mark various locations in the program with symbols. For example, one can declare the label `loop` to refer to the beginning of a certain code segment. Other commands in the program can then `goto loop`, either conditionally or unconditionally.

The introduction of symbols into assembly languages suggests that assemblers must be more sophisticated than dumb text processing programs. Granted, translating agreed-upon symbols into agreed-upon binary codes is not a complicated task. At the same time, the mapping of user-defined variable names and symbolic labels on actual memory addresses is not trivial. In fact, this symbol resolution task is the first nontrivial translation challenge in our ascent up the software hierarchy from the hardware level. The following example illustrates the challenge and the common way to address it.

Symbol Resolution Consider figure 6.1, showing a program written in some self-explanatory low-level language. The program contains four user-defined symbols: two variable names (`i` and `sum`) and two labels (`loop` and `end`). How can we systematically convert this program into a symbol-less code?

We start by making two arbitrary game rules: The translated code will be stored in the computer's memory starting at address 0, and variables will be allocated to memory locations starting at address 1024 (these rules depend on the specific target hardware platform). Next, we build a *symbol table*, as follows. For each new symbol *xxx* encountered in the source code, we add a line (xxx, n) to the symbol table, where n is the memory address associated with the symbol according to the game rules. After completing the construction of the symbol table, we use it to translate the program into its symbol-less version.

Note that according to the assumed game rules, variables `i` and `sum` are allocated to addresses 1024 and 1025, respectively. Of course any other two addresses will be just as good, so long as *all* references to `i` and `sum` in the program resolve to the same physical addresses, as indeed is the case. The remaining code is self-explanatory, except perhaps for instruction 6. This instruction terminates the program's execution by putting the computer in an infinite loop.

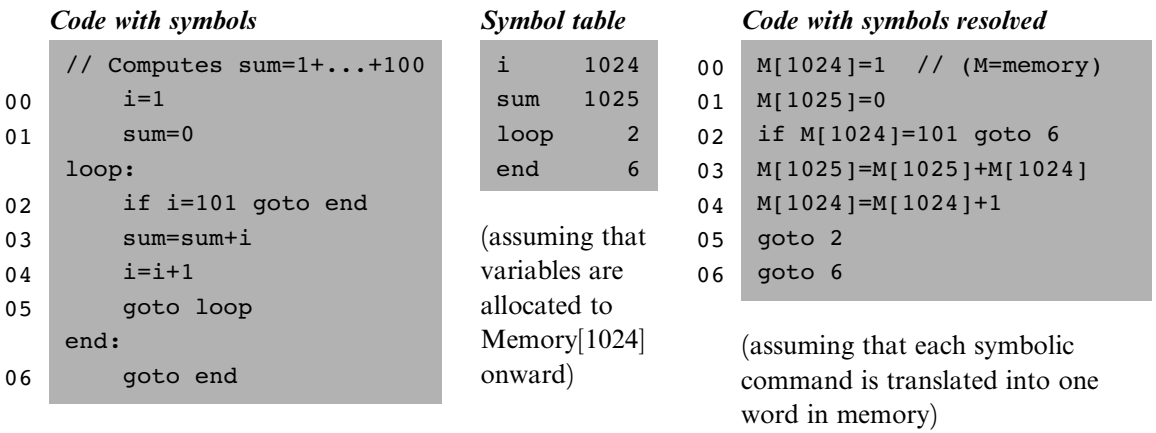


Figure 6.1 Symbol resolution using a symbol table. The line numbers are not part of the program—they simply count all the lines in the program that represent real instructions, namely, neither comments nor label declarations. Note that once we have the symbol table in place, the symbol resolution task is straightforward.

Three comments are in order here. First, note that the variable allocation assumption implies that the largest program that we can run is 1,024 instructions long. Since realistic programs (like the operating system) are obviously much larger, the base address for storing variables will normally be much farther. Second, the assumption that each source command is mapped on one word may be naïve. Typically, some assembly commands (e.g., `if i=101 goto end`) may translate into several machine instructions and thus will end up occupying several memory locations. The translator can deal with this variance by keeping track of how many words each source command generates, then updating its ''instruction memory counter'' accordingly.

Finally, the assumption that each variable is represented by a single memory location is also naïve. Programming languages feature variables of different types, and these occupy different memory spaces on the target computer. For example, the C language data types `short` and `double` represent 16-bit and 64-bit numbers, respectively. When a C program is run on a 16-bit machine, these variables will occupy a single memory address and a block of four consecutive addresses, respectively. Thus, when allocating memory space for variables, the translator must take into account both their data types and the word width of the target hardware.

The Assembler Before an assembly program can be executed on a computer, it must be translated into the computer's binary machine language. The translation task is

done by a program called the assembler. The assembler takes as input a stream of assembly commands and generates as output a stream of equivalent binary instructions. The resulting code can be loaded as is into the computer's memory and executed by the hardware.

We see that the assembler is essentially a text-processing program, designed to provide translation services. The programmer who is commissioned to write the assembler must be given the full documentation of the assembly syntax, on the one hand, and the respective binary codes, on the other. Following this contract—typically called *machine language specification*—it is not difficult to write a program that, for each symbolic command, carries out the following tasks (not necessarily in that order):

- Parse the symbolic command into its underlying fields.
- For each field, generate the corresponding bits in the machine language.
- Replace all symbolic references (if any) with numeric addresses of memory locations.
- Assemble the binary codes into a complete machine instruction.

Three of the above tasks (parsing, code generation, and final assembly) are rather easy to implement. The fourth task—symbols handling—is more challenging, and considered one of the main functions of the assembler. This function was described in the previous section. The next two sections specify the Hack assembly language and propose an assembler implementation for it, respectively.

6.2 Hack Assembly-to-Binary Translation Specification

The Hack assembly language and its equivalent binary representation were specified in chapter 4. A compact and formal version of this language specification is repeated here, for ease of reference. This specification can be viewed as the contract that Hack assemblers must implement, one way or another.

6.2.1 Syntax Conventions and File Formats

File Names By convention, programs in binary machine code and in assembly code are stored in text files with "`hack`" and "`asm`" extensions, respectively. Thus, a `Prog.asm` file is translated by the assembler into a `Prog.hack` file.

Binary Code (`.hack`) Files A binary code file is composed of text lines. Each line is a sequence of 16 "`0`" and "`1`" ASCII characters, coding a single 16-bit machine language instruction. Taken together, all the lines in the file represent a machine language program. When a machine language program is loaded into the computer's instruction memory, the binary code represented by the file's *n*th line is stored in address *n* of the instruction memory (the count of both program lines and memory addresses starts at 0).

Assembly Language (`.asm`) Files An assembly language file is composed of text lines, each representing either an *instruction* or a *symbol declaration*:

- *Instruction*: an *A*-instruction or a *C*-instruction, described in section 6.2.2.
- `(Symbol)`: This pseudo-command binds the `Symbol` to the memory location into which the next command in the program will be stored. It is called "pseudo-command" since it generates no machine code.

(The remaining conventions in this section pertain to assembly programs only.)

Constants and Symbols *Constants* must be non-negative and are written in decimal notation. A user-defined *symbol* can be any sequence of letters, digits, underscore (`_`), dot (`.`), dollar sign (`$`), and colon (`:`) that does not begin with a digit.

Comments Text beginning with two slashes (`//`) and ending at the end of the line is considered a comment and is ignored.

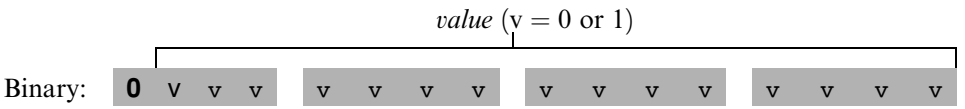
White Space Space characters are ignored. Empty lines are ignored.

Case Conventions All the assembly mnemonics must be written in uppercase. The rest (user-defined labels and variable names) is case sensitive. The convention is to use uppercase for labels and lowercase for variable names.

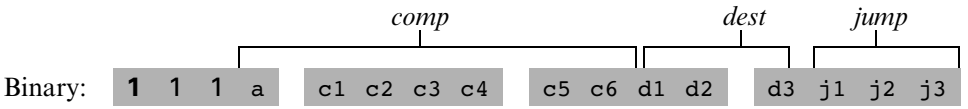
6.2.2 Instructions

The Hack machine language consists of two instruction types called *addressing instruction* (*A*-instruction) and *compute instruction* (*C*-instruction). The instruction format is as follows.

A-instruction: *@value* // Where *value* is either a non-negative decimal number
// or a symbol referring to such number.



C-instruction: *dest=comp;jump* // Either the *dest* or *jump* fields may be empty.
// If *dest* is empty, the "=" is omitted;
// If *jump* is empty, the ";" is omitted.



The translation of each of the three fields *comp, dest, jump* to their binary forms is specified in the following three tables.

comp (when a=0)	c1	c2	c3	c4	c5	c6	***comp*** (when a=1)
0	1	0	1	0	1	0	
1	1	1	1	1	1	1	
-1	1	1	1	0	1	0	
D	0	0	1	1	0	0	
A	1	1	0	0	0	0	M
!D	0	0	1	1	0	1	
!A	1	1	0	0	0	1	!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0	0	1	0	D+M
D-A	0	1	0	0	1	1	D-M
A-D	0	0	0	1	1	1	M-D
D&A	0	0	0	0	0	0	D&M
D|A	0	1	0	1	0	1	D|M

dest	d1	d2	d3
null	0	0	0
M	0	0	1
D	0	1	0
MD	0	1	1
A	1	0	0
AM	1	0	1
AD	1	1	0
AMD	1	1	1

jump	j1	j2	j3
null	0	0	0
JGT	0	0	1
JEQ	0	1	0
JGE	0	1	1
JLT	1	0	0
JNE	1	0	1
JLE	1	1	0
JMP	1	1	1

6.2.3 Symbols

Hack assembly commands can refer to memory locations (addresses) using either constants or symbols. Symbols in assembly programs arise from three sources.

Predefined Symbols Any Hack assembly program is allowed to use the following predefined symbols.

Label	*RAM address*	*(hexa)*
SP	0	0x0000
LCL	1	0x0001
ARG	2	0x0002
THIS	3	0x0003
THAT	4	0x0004
R0–R15	0-15	0x0000-f
SCREEN	16384	0x4000
KBD	24576	0x6000

Note that each one of the top five RAM locations can be referred to using two predefined symbols. For example, either `R2` or `ARG` can be used to refer to RAM[2].

Label Symbols The pseudo-command `(Xxx)` defines the symbol `Xxx` to refer to the instruction memory location holding the next command in the program. A label can be defined only once and can be used anywhere in the assembly program, even before the line in which it is defined.

Variable Symbols Any symbol `Xxx` appearing in an assembly program that is not predefined and is not defined elsewhere using the `(Xxx)` command is treated as a variable. Variables are mapped to consecutive memory locations as they are first encountered, starting at RAM address 16 (0x0010).

6.2.4 Example

Chapter 4 presented a program that sums up the integers 1 to 100. Figure 6.2 repeats this example, showing both its assembly and binary versions.

Assembly code (`Prog.asm`)

```
// Adds 1 + ... + 100
      @i
      M=1    // i=1
      @sum
      M=0    // sum=0
(LOOP)
      @i
      D=M    // D=i
      @100
      D=D-A  // D=i-100
      @END
      D;JGT  // if (i-100)>0 goto END
      @i
      D=M    // D=i
      @sum
      M=D+M  // sum=sum+i
      @i
      M=M+1  // i=i+1
      @LOOP
      0;JMP  // goto LOOP
(END)
      @END
      0;JMP  // infinite loop
```

Assembler

Binary code (`Prog.hack`)

```
(this line should be erased)
0000 0000 0001 0000
1110 1111 1100 1000
0000 0000 0001 0001
1110 1010 1000 1000
(this line should be erased)
0000 0000 0001 0000
1111 1100 0001 0000
0000 0000 0110 0100
1110 0100 1101 0000
0000 0000 0001 0010
1110 0011 0000 0001
0000 0000 0001 0000
1111 1100 0001 0000
0000 0000 0001 0001
1111 0000 1000 1000
0000 0000 0001 0000
1111 1101 1100 1000
0000 0000 0000 0100
1110 1010 1000 0111
(this line should be erased)
0000 0000 0001 0010
1110 1010 1000 0111
```

Figure 6.2 Assembly and binary representations of the same program.

6.3 Implementation

The Hack assembler reads as input a text file named `Prog.asm`, containing a Hack assembly program, and produces as output a text file named `Prog.hack`, containing the translated Hack machine code. The name of the input file is supplied to the assembler as a command line argument:

```
prompt> Assembler Prog.asm
```

The translation of each individual assembly command to its equivalent binary instruction is direct and one-to-one. Each command is translated separately. In particular, each mnemonic component (field) of the assembly command is translated into its corresponding bit code according to the tables in section 6.2.2, and each symbol in the command is resolved to its numeric address as specified in section 6.2.3.

We propose an assembler implementation based on four modules: a *Parser* module that parses the input, a *Code* module that provides the binary codes of all the assembly mnemonics, a *SymbolTable* module that handles symbols, and a main program that drives the entire translation process.

A Note about API Notation The assembler development is the first in a series of five software construction projects that build our hierarchy of translators (*assembler*, *virtual machine*, and *compiler*). Since readers can develop these projects in the programming language of their choice, we base our proposed implementation guidelines on language independent APIs. A typical project API describes several *modules*, each containing one or more *routines*. In object-oriented languages like Java, C++, and C#, a module usually corresponds to a *class*, and a routine usually corresponds to a *method*. In procedural languages, routines correspond to *functions*, *subroutines*, or *procedures*, and modules correspond to collections of routines that handle related data. In some languages (e.g., Modula-2) a module may be expressed explicitly, in others implicitly (e.g., a *file* in the C language), and in others (e.g., Pascal) it will have no corresponding language construct, and will just be a conceptual grouping of routines.

6.3.1 The *Parser* Module

The main function of the parser is to break each assembly command into its underlying components (fields and symbols). The API is as follows.

Parser: Encapsulates access to the input code. Reads an assembly language command, parses it, and provides convenient access to the command's components (fields and symbols). In addition, removes all white space and comments.

Routine	Arguments	Returns	Function
Constructor/ initializer	Input file/ stream	—	Opens the input file/stream and gets ready to parse it.
`hasMoreCommands`	—	Boolean	Are there more commands in the input?
`advance`	—	—	Reads the next command from the input and makes it the current command. Should be called only if `hasMoreCommands()` is true. Initially there is no current command.
`commandType`	—	`A_COMMAND`, `C_COMMAND`, `L_COMMAND`	Returns the type of the current command: ■ `A_COMMAND` for `@Xxx` where `Xxx` is either a symbol or a decimal number ■ `C_COMMAND` for `dest=comp;jump` ■ `L_COMMAND` (actually, pseudo-command) for `(Xxx)` where `Xxx` is a symbol.
`symbol`	—	string	Returns the symbol or decimal `Xxx` of the current command `@Xxx` or `(Xxx)`. Should be called only when `commandType()` is `A_COMMAND` or `L_COMMAND`.
`dest`	—	string	Returns the `dest` mnemonic in the current *C*-command (8 possibilities). Should be called only when `commandType()` is `C_COMMAND`.

Routine	Arguments	Returns	Function
`comp`	—	string	Returns the `comp` mnemonic in the current *C*-command (28 possibilities). Should be called only when `commandType()` is `C_COMMAND`.
`jump`	—	string	Returns the `jump` mnemonic in the current *C*-command (8 possibilities). Should be called only when `commandType()` is `C_COMMAND`.

6.3.2 The *Code* Module

Code: Translates Hack assembly language mnemonics into binary codes.

Routine	Arguments	Returns	Function
`dest`	mnemonic (string)	3 bits	Returns the binary code of the `dest` mnemonic.
`comp`	mnemonic (string)	7 bits	Returns the binary code of the `comp` mnemonic.
`jump`	mnemonic (string)	3 bits	Returns the binary code of the `jump` mnemonic.

6.3.3 Assembler for Programs with No Symbols

We suggest building the assembler in two stages. In the first stage, write an assembler that translates assembly programs without symbols. This can be done using the Parser and Code modules just described. In the second stage, extend the assembler with symbol handling capabilities, as we explain in the next section.

The contract for the first symbol-less stage is that the input `Prog.asm` program contains no symbols. This means that (a) in all address commands of type `@Xxx` the `Xxx` constants are decimal numbers and not symbols, and (b) the input file contains no label commands, namely, no commands of type `(Xxx)`.

The overall symbol-less assembler program can now be implemented as follows. First, the program opens an output file named `Prog.hack`. Next, the program marches through the lines (assembly instructions) in the supplied `Prog.asm` file. For each *C*-instruction, the program concatenates the translated binary codes of the instruction fields into a single 16-bit word. Next, the program writes this word into the `Prog.hack` file. For each *A*-instruction of type `@Xxx`, the program translates the decimal constant returned by the parser into its binary representation and writes the resulting 16-bit word into the `Prog.hack` file.

6.3.4 The *SymbolTable* Module

Since Hack instructions can contain symbols, the symbols must be resolved into actual addresses as part of the translation process. The assembler deals with this task using a *symbol table*, designed to create and maintain the correspondence between symbols and their meaning (in Hack's case, RAM and ROM addresses). A natural data structure for representing such a relationship is the classical *hash table*. In most programming languages, such a data structure is available as part of a standard library, and thus there is no need to develop it from scratch. We propose the following API.

SymbolTable: Keeps a correspondence between symbolic labels and numeric addresses.

Routine	Arguments	Returns	Function
`Constructor`	—	—	Creates a new empty symbol table.
`addEntry`	`symbol` (string), `address` (int)	—	Adds the pair (`symbol`, `address`) to the table.
`contains`	`symbol` (string)	Boolean	Does the symbol table contain the given `symbol`?
`GetAddress`	`symbol` (string)	int	Returns the address associated with the `symbol`.

6.3.5 Assembler for Programs with Symbols

Assembly programs are allowed to use symbolic labels (destinations of *goto* commands) before the symbols are defined. This convention makes the life of assembly

programmers easier and that of assembler developers harder. A common solution to this complication is to write a two-pass assembler that reads the code twice, from start to end. In the first pass, the assembler builds the symbol table and generates no code. In the second pass, all the label symbols encountered in the program have already been bound to memory locations and recorded in the symbol table. Thus, the assembler can replace each symbol with its corresponding meaning (numeric address) and generate the final binary code.

Recall that there are three types of symbols in the Hack language: *predefined symbols*, *labels*, and *variables*. The symbol table should contain and handle all these symbols, as follows.

Initialization Initialize the symbol table with all the predefined symbols and their pre-allocated RAM addresses, according to section 6.2.3.

First Pass Go through the entire assembly program, line by line, and build the symbol table without generating any code. As you march through the program lines, keep a running number recording the ROM address into which the current command will be eventually loaded. This number starts at 0 and is incremented by 1 whenever a *C*-instruction or an *A*-instruction is encountered, but does not change when a label pseudocommand or a comment is encountered. Each time a pseudocommand `(Xxx)` is encountered, add a new entry to the symbol table, associating `Xxx` with the ROM address that will eventually store the next command in the program. This pass results in entering all the program's *labels* along with their ROM addresses into the symbol table. The program's variables are handled in the second pass.

Second Pass Now go again through the entire program, and parse each line. Each time a symbolic *A*-instruction is encountered, namely, `@Xxx` where `Xxx` is a symbol and not a number, look up `Xxx` in the symbol table. If the symbol is found in the table, replace it with its numeric meaning and complete the command's translation. If the symbol is not found in the table, then it must represent a new variable. To handle it, add the pair (`Xxx`, *n*) to the symbol table, where *n* is the next available RAM address, and complete the command's translation. The allocated RAM addresses are consecutive numbers, starting at address 16 (just after the addresses allocated to the predefined symbols).

This completes the assembler's implementation.

6.4 Perspective

Like most assemblers, the Hack assembler is a relatively simple program, dealing mainly with text processing. Naturally, assemblers for richer machine languages are more complex. Also, some assemblers feature more sophisticated symbol handling capabilities not found in Hack. For example, the assembler may allow programmers to explicitly associate symbols with particular data addresses, to perform "constant arithmetic" on symbols (e.g., to use `table+5` to refer to the fifth memory location after the address referred to by `table`), and so on. Additionally, many assemblers are capable of handling *macro commands*. A macro command is simply a sequence of machine instructions that has a name. For example, our assembler can be extended to translate an agreed-upon macro-command, say `D=M[`*xxx*`]`, into the two instructions *@xxx* followed immediately by `D=M` (*xxx* being an address). Clearly, such macro commands can considerably simplify the programming of commonly occurring operations, at a low translation cost.

We note in closing that stand-alone assemblers are rarely used in practice. First, assembly programs are rarely written by humans, but rather by compilers. And a compiler—being an automaton—does not have to bother to generate symbolic commands, since it may be more convenient to directly produce binary machine code. On the other hand, many high-level language compilers allow programmers to embed segments of assembly language code within high-level programs. This capability, which is rather common in C language compilers, gives the programmer direct control of the underlying hardware, for optimization.

6.5 Project

Objective Develop an assembler that translates programs written in Hack assembly language into the binary code understood by the Hack hardware platform. The assembler must implement the translation specification described in section 6.2.

Resources The only tool needed for completing this project is the programming language in which you will implement your assembler. You may also find the following two tools useful: the assembler and CPU emulator supplied with the book. These tools allow you to experiment with a working assembler before you set out to build one yourself. In addition, the supplied assembler provides a visual

line-by-line translation GUI and allows online code comparisons with the outputs that *your* assembler will generate. For more information about these capabilities, refer to the assembler tutorial (part of the book's software suite).

Contract When loaded into your assembler, a `Prog.asm` file containing a valid Hack assembly language program should be translated into the correct Hack binary code and stored in a `Prog.hack` file. The output produced by your assembler must be identical to the output produced by the assembler supplied with the book.

Building Plan We suggest building the assembler in two stages. First write a symbol-less assembler, namely, an assembler that can only translate programs that contain no symbols. Then extend your assembler with symbol handling capabilities. The test programs that we supply here come in two such versions (without and with symbols), to help you test your assembler incrementally.

Test Programs Each test program except the first one comes in two versions: `ProgL.asm` is symbol-less, and `Prog.asm` is with symbols.

Add: Adds the constants 2 and 3 and puts the result in R0.

Max: Computes max(R0, R1) and puts the result in R2.

Rect: Draws a rectangle at the top left corner of the screen. The rectangle is 16 pixels wide and R0 pixels high.

Pong: A single-player Ping-Pong game. A ball bounces constantly off the screen's "walls." The player attempts to hit the ball with a bat by pressing the left and right arrow keys. For every successful hit, the player gains one point and the bat shrinks a little to make the game harder. If the player misses the ball, the game is over. To quit the game, press `ESC`.

The *Pong* program was written in the *Jack* programming language (chapter 9) and translated into the supplied assembly program by the *Jack compiler* (chapters 10–11). Although the original Jack program is only about 300 lines of code, the executable Pong application is about 20,000 lines of binary code, most of which being the Jack operating system (chapter 12). Running this interactive program in the CPU emulator is a slow affair, so don't expect a high-powered Pong game. This slowness is actually a virtue, since it enables your eye to track the graphical behavior of the program. In future projects in the book, this game will run much faster.

Steps Write and test your assembler program in the two stages described previously. You may use the assembler supplied with the book to compare the output of your assembler to the correct output. This testing procedure is described next. For more information about the supplied assembler, refer to the assembler tutorial.

The Supplied Assembler The practice of using the supplied assembler (which produces correct binary code) to test another assembler (which is not necessarily correct) is illustrated in figure 6.3. Let `Prog.asm` be some program written in Hack assembly. Suppose that we translate this program using the supplied assembler, producing

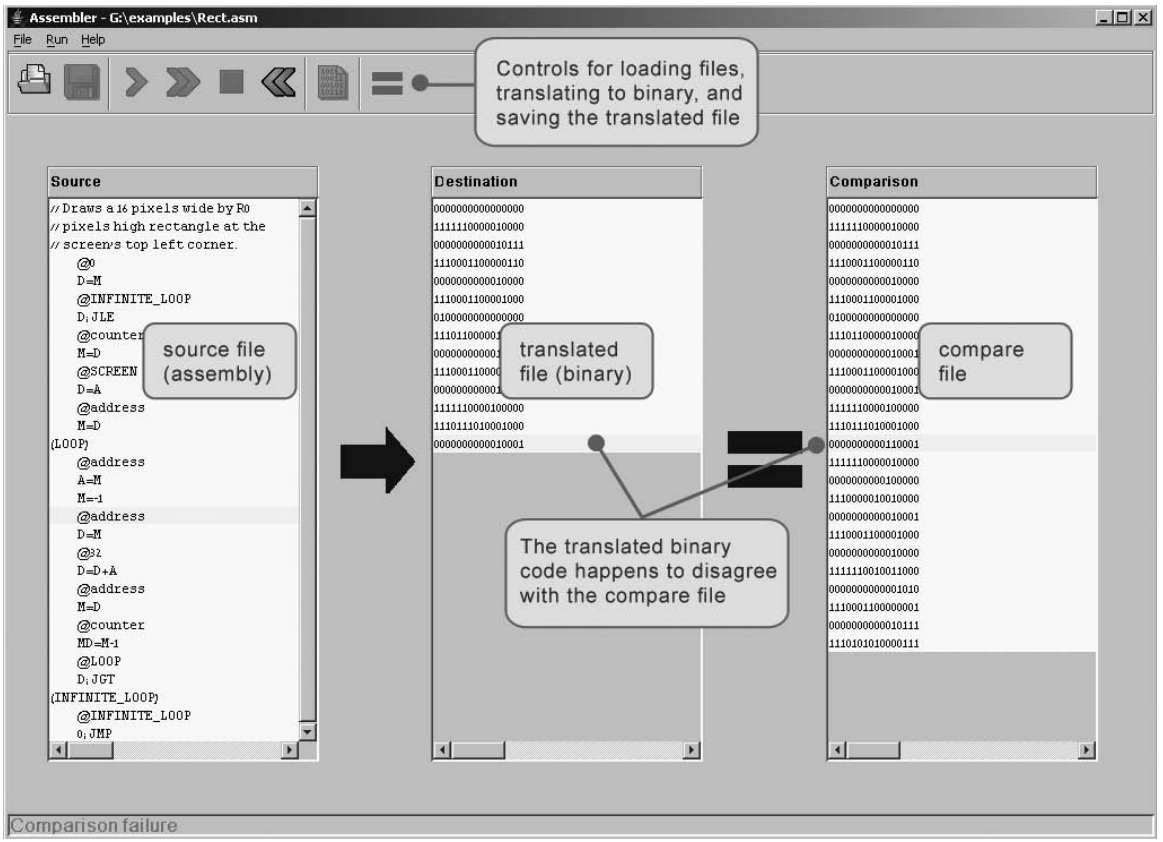


Figure 6.3 Using the supplied assembler to test the code generated by another assembler.

a binary file called `Prog.hack`. Next, we use another assembler (e.g., the one that you wrote) to translate the same program into another file, say `Prog1.hack`. Now, if the latter assembler is working correctly, it follows that `Prog.hack` = `Prog1.hack`. Thus, one way to test a newly written assembler is to load `Prog.asm` into the supplied assembler program, load `Prog1.hack` as a compare file, then translate and compare the two binary files (see figure 6.3). If the comparison fails, the assembler that produced `Prog1.hack` must be buggy; otherwise, it may be error-free.