5. The Assembler [1]

When I use a word, Humpty Dumpty said, in a rather scornful tone,
it means just what I choose it to mean – neither more, nor less.
(Lewis Carole, 1832-1898)

In this chapter we specify the Hack assembly language and describe how to write an assembler for it. The assembler is designed to translate programs written in the Hack assembly language to programs written in Hack machine language. The latter programs can run on the Hack computer platform built in chapters 1-4.

Note: Sections 5.2.2, 5.2.3, and 5.2.4 describe the Hack machine language are identical to Sections 4.2.1-4.2.2 of Chapter 4.

5.1 Background

The beginning of this section is work in progress.

The Assembler

Before an assembly program can be executed on the computer platform, it must be translated into the machine language of the target hardware. The translation task is done by a program called *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 then executed by the hardware. In the Hack platform, the correspondence between assembly commands and machine language instruction is one-to-one: each symbolic assembly command can be translated into a single 16-bit instruction.

We see that the assembler is essentially a text-processing program, designed to provide translation services. The programmer who is commissioned to write such an assembler must be given the full specifications of the target hardware’s instruction set, on the one hand, and the agreed-upon syntax of the symbolic assembly commands, on the other. Using this contract, it is not hard to write a program that, for each symbolic command, carries out the following operations (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 sub-tasks (parsing, code generation, and final assembly) are rather easy to implement. The fourth sub-task -- symbols handling -- is more challenging, and considered one of the main functions of the assembler.

[1] From *The Digital Core*, by Nisan & Schocken, 2003, www.idc.ac.il/csd, forthcoming by MIT Press.

The translation of symbols to numeric addresses is done it two conceptual stages. First, the assembler creates a *symbol table*, associating each symbol with a designated memory address. Next, the assembler uses the symbol table to translate each occurrence of each symbol in the program to its allocated address. This results with a program with no symbolic references, i.e. one that can be easily translated into binary code.

5.2 Hack Machine and Assembly Language Specification

We start with an example of a small program written in assembly language, together with the machine code equivalent. The assembler should convert the former into the latter. We then provide a specification of the machine language and the associated assembly language mnemonics. Finally, we specify syntactic issues relating only to the assembly language, such as file formats and symbols.

5.2.1 Example

Consider the following C code, designed to sum the numbers between 1 and 100:

```
 int i=1;
 int sum=0;
 while (i<=100){
    sum+=i;
    i++;
}
```

The Hack assembly language implementation of this algorithm is given in Prog. 5-1.

```
      @i     // i=1 (allocated at 0x0010)    // 0000 0000 0001 0000
      M=1                                    // 1110 1111 1100 1000
      @sum   // sum=0 (allocated at 0x0011)  // 0000 0000 0001 0001
      M=0                                    // 1110 1010 1000 1000
(loop)
      @i     // if i-100>0 then goto end     // 0000 0000 0001 0000
      D=M                                    // 1111 1100 0001 0000
      @100                                   // 0000 0000 0110 0100
      D=D-A                                  // 1110 0100 1101 0000
      @end                                   // 0000 0000 0001 0010
      D;jgt                                  // 1110 0011 0000 0001
      @i     // sum+=i                       // 0000 0000 0001 0000
      D=M                                    // 1111 1100 0001 0000
      @sum                                   // 0000 0000 0001 0001
      M=D+M                                  // 1111 0000 1000 1000
      @i     // i++                          // 0000 0000 0001 0000
      M=M+1                                  // 1111 1101 1100 1000
      @loop  // goto loop                    // 0000 0000 0000 0100
      0;jmp                                  // 1110 1010 1000 0111
 (end)
```

PROGRAM 5-1: Assembly and machine language representations of the same program. If the assembler is given a text file that contains the above program, it should generate the binary code documented on the right.

Instead of explaining the syntax and use of each command in the above program, let us make some general observations. First, note that the assembler gets rid of all the comments (text after "//") and white space in the source code. Second, note the recurring use of three built-in variables called D,A, and M. Third, note that some lines in the source code -- in particular (loop) and (end), generate no code. Finally, note that every manipulation of a variable as well as every jump command is preceded by some `@Xxx` command. We now turn to a detailed specification of these and other features of the Hack machine and assembly languages. **Note:** all the text from here up to the end of Section 5.2.4 is an exact copy of Sections 4.2.1-4.2.2 of Chapter 4.

5.2.2 Basic Constructs

The Hack platform is built after the von Newman paradigm. It is a 16-bit machine, consisting of a CPU, two separate memory modules serving as instruction memory and data memory, and two memory-mapped I/O devices: a screen and a keyboard.

Memory Address Spaces: The Hack programmer is aware of two distinct memory address spaces: an *instruction memory* and a *data memory*. Both memories are 16-bit wide and have a 15-bit address space, meaning that the maximum size of each memory is 32K 16-bit words.

The CPU can only execute programs that reside in the instruction memory. The instruction memory is a read-only device, and thus programs are loaded into it using some exogenous means. For example, the instruction memory can be implemented in a ROM chip which is pre-burned with the required program. Loading a new program can be done by replacing the entire ROM chip. In order to simulate this operation, hardware simulators of the Hack platform must provide means to load the instruction memory from a text file.

Registers: The Hack programmer is aware of three registers called `D`, `A`, and `PC`. `D` and `A` are general-purpose 16-bit registers that can be manipulated explicitly by arithmetic and logical instructions, e.g. `A=D-1` or `D=!A`. (where "`!`" means "`not`"). While `D` is used solely to store data values, `A` doubles as both a data register and an address register. That is to say, depending on the instruction context, the contents of `A` can be interpreted either as a data value, or as an address in the data memory, or as an address in the instruction memory, as explained below.

First, the `A` register can be used to facilitate direct access to the data memory (which, from now on, will be often referred to as "memory"). As the next section will describe, the syntax of the Hack language is such that memory access instructions do not specify an explicit address. Instead, they operate on an implicit memory location labeled "`M`", e.g. `D=M+1`. In order to resolve this address, the contract is such that any Hack instruction involving `M` should effect the memory word whose address equals the current value of `A`, i.e. `Memory[A]`. For example, if we set the `A` register to 516, the subsequent instruction `D=M-1` would imply `D=Memory[516]-1`.

Second, in addition to doubling as a general-purpose register and as an address register for the data memory, the hard working `A` register is also used to facilitate direct access to the instruction memory. As we will see shortly, the syntax of the Hack language is such that "jump" instructions do not specify a particular address. Instead, the contract is such that any "jump" operation affects a jump to the instruction memory word addressed by `A`. For example, if we set `A` to 35, a subsequent "goto" instruction would cause the computer to fetch the instruction located in InstructionMemory[35].

Jump instructions also affect the program counter, or `PC`. As a rule, the next instruction that the computer will fetch and execute is always InstructionMemory[`PC`]. Thus, if the current instruction specifies no jump, the `PC` is incremented by 1. If the current instruction implies a jump, the `PC` is set to the current value of `A`. Hence, the `PC` is also affected by the programmer, but only implicitly.

Instructions: The *Hack machine language* features two generic instructions represented by 16-bit codes, of which the MSB specifies which instruction it is. A 16-bit code beginning with a "0" represents an *address* instruction, also called *A*-instruction. A 16-bit code beginning with a "1" represents a *compute-store-jump* instruction, also called *C*-instruction. Each instruction has a binary representation, a symbolic representation, and an effect on the computer, as we now turn to specify.

5.2.3 The *A*-Instruction

The *A*-instruction is used to set the `A` register to a 15-bit value:

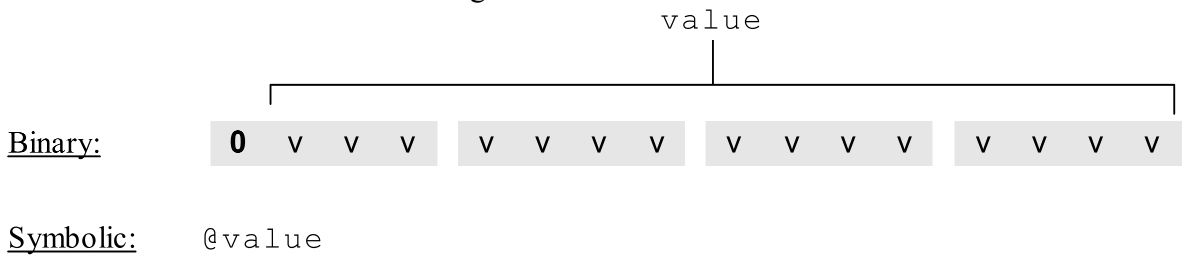

FIGURE 5-2: The *A*-Instruction syntax.

The symbolic syntax of the *A*-instruction is such that `value` is the decimal representation of the 15-bit constant `vvvvvvvvvvvvvvv` (each `v` being 0 or 1). This instruction causes the computer to store the constant in the `A` register. For example, the instruction `@5`, which is equivalent to `0000000000000101`, causes the computer to store the 2's complement binary representation of 5 in the `A` register.

The *A*-instruction is used for three different purposes. First, it provides the only way to enter a constant into the computer under program control. Second, it sets the stage for a *C*-instruction designed to manipulate a certain data memory location, by first setting `A` to the address of that location. Third, it sets the stage for a *C*-instruction that involves a jump, by first loading the address of the jump destination to the `A` register. These different use patterns are illustrated below.

5.2.4 The *C*-Instruction

The *C*-instruction is the programming workhorse of the Hack platform -- the instruction that gets almost everything done. The instruction code is a specification that answers three questions: (a) what to compute? (b) where to store the computed value? and (c) what to do next? Along with the *A*-instruction, these specifications determine all the possible operations of the computer.

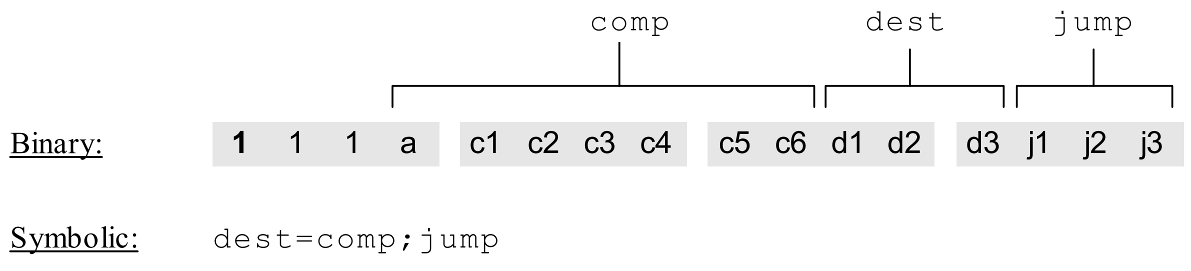

FIGURE 5-3: The *C*-Instruction syntax.

The MSB is the *C*-instruction code, which is 1. The next two bits are not used. The remaining bits form three fields that correspond to the three parts of the instruction's symbolic representation. Taken together, the semantics of the symbolic instruction `dest=comp;jump` is as follow. The `comp` field instructs the CPU what to compute. The `dest` field instructs where to store the computed value. The `jump` field specifies a jump condition. Either the `dest` field or the `jump` field or both may be empty. If the `dest` field is empty then the "=" sign may be omitted. If the `jump` field is empty then the ";" symbol may be omitted. We now turn to describe the format and semantics of each of the three fields.

The computation specification: The ALU is designed to compute a fixed set of functions on the `D`, `A`, and `M` registers (where `M`=Memory[`A`]). The computed function is specified by the `a`-bit and the six `c`-bits comprising the instruction's `comp` field. This 7-bit pattern can potentially code 128 different functions, of which only the 28 listed in Table 5-4 are documented in the computer specification.

a=0							
mnemonic	c1	c2	c3	c4	c5	c6	
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
	c1	c2	c3	c4	c5	c6	**mnemonic**
							a=1

TABLE 5-4: The "compute" specification of the *C*-instruction. D and A are names of registers. M refers to the memory location addressed by A, i.e. to Memory[A]. The symbols "+" and "–" denote 16-bit 2's complement addition and subtraction, respectively, while "!", "|", and "&" denote the 16-bit bit-wise Boolean operators Not, Or, And, respectively. Note the similarity between this instruction set and the ALU specification given in Table 2-7.

Recall that the format of the *C*-instruction is "111a cccc ccdd djjj". Suppose we want to compute D-1, i.e. "the current value of the D register minus 1". According to Table 5-4, this can be done by issuing the instruction "1110 0011 10xx xxxx" (we use "x" to label bits that are irrelevant to the given example). To compute the value of D|M, we issue the instruction "1111 0101 01xx xxxx". To compute the constant -1, we issue the instruction "1110 1110 10xx xxxx", and so on. In short, the instruction set supports a selection of 28 different functions of interest involving unary, binary, and null operations on A, D, and M.

The destination specification: The value computed by the comp part of the *C*-instruction can be simultaneously stored in several destinations, as specified by the instruction's dest part. The first and second d-bits code whether to store the computed value in the A register and in the D register, respectively. The third d-bit codes whether to store the computed value in M (i.e. in Memory[A]). One, more than one, or none of these bits may be asserted, as follows:

d1	**d2**	**d3**	***mnemonic***	***destination (where to store the computed value)***
0	0	0	null	the value is not stored anywhere
0	0	1	M	Memory[A] (memory register addressed by A)
0	1	0	D	D register
0	1	1	MD	Memory[A] and D register
1	0	0	A	A register
1	0	1	AM	A register and Memory[A]
1	1	0	AD	A register and D register
1	1	1	AMD	A register, Memory[A], and D register

TABLE 5-5: The "destination" specification of the *C*-instruction.

Recall that the format of the *C*-instruction is "`111a cccc ccdd djjj`". Suppose we want the computer to increment the value of Memory[7] by 1, and also store the result in the `D` register. According to tables 5-4 and 5-5, this can be accomplished by the instructions:

```
0000 0000 0000 0111    // @7
1111 1101 1101 1xxx    // DM=M+1  (x=irrelevant bits)
```

The first *A*-instruction causes the computer to select the memory register whose address is 7 (the so called "`M` register"). The subsequent *C*-instruction computes the value of `M+1` and stores the result in both `D` and `M`. The role of the 3 LSB bits of the second instruction is explained next.

The jump specification: The `jump` field of the *C*-instruction tells the computer what to do next. There are two possibilities: the next instruction to execute may either be the next instruction in the instruction memory (implying a no-jump: `PC=PC+1`, where `PC` is the *program counter*) or it may be the instruction whose address is held in the `A` register (implying a jump: `PC=A`). In the latter case, we assume that the `A` register has been previously set to the address to which we want to jump.

A jump is performed conditionally according to the sign of the value computed by the ALU. The first `j`-bit specifies whether to jump in case the ALU output is negative, the second `j`-bit in case the ALU output is zero, and the third `j`-bit in case it is positive. This gives 8 possible jump conditions, listed in Table 5-6.

`j1` ($out \prec 0$)	`j2` ($out = 0$)	`j3` ($out \succ 0$)	**Mnemonic**	**Effect**
0	0	0	`null`	no jump
0	0	1	`JGT`	if $out \succ 0$ jump
0	1	0	`JEQ`	if $out = 0$ jump
0	1	1	`JGE`	if $out \geq 0$ jump
1	0	0	`JLT`	if $out \prec 0$ jump
1	0	1	`JNE`	if $out \neq 0$ jump
1	1	0	`JLE`	if $out \leq 0$ jump
1	1	1	`JMP`	jump

TABLE 5-6: The "jump" specification of the *C*-instruction. In the right column, the "jump" directive implies "fetch the instruction addressed by the `A` register" (i.e. set `PC=A`), while *out* refers to the ALU output value. This value results from the instruction's `comp` part.

For example, suppose we want to effect the following logic: `if Memory[3]=5 goto 100 else goto 200`. According to the language specification, this can be accomplished by the following code:

Desired logic

```
if Memory[j]=5 then
   goto 100
else goto 200
```

Hack implementation

```
@3    // D=Memory[3]
D=M
@5    // D=D-5
D=D-A
@100  // if D=0 goto 100
D;JEQ
@200  // goto 200
0;JMP
```

The last instruction ("`0;JMP`") effects an unconditional jump. Since the *C*-instruction syntax requires that we always effect *some* computation, we instruct to compute 0 (an arbitrary choice).

Restrictions on the *C*-Instruction: As was illustrated in the code above, the programmer can use the `A` register in order to select a *data memory* location for a subsequent *C*-instruction involving `M`, or else `A` can be used to select an *instruction memory* location in order to set the stage for a subsequent *C*-instruction involving a jump. To prevent conflicting use of the `A` register, we require that *a C-instruction that may cause a jump (i.e. with some non-zero j bits) cannot contain a reference to* `M`. Violations of this rule will cause the computer to do something unexpected. Problematic instructions like this one can be intercepted by the Assembler program, as we will see later in the chapter.

5.2.5 Assembly Language Extensions

Now that we have specified the Hack machine language, we are ready to describe an extended version of it, called the *Hack assembly language*. Compared to machine language, assembly language has two main features that make low-level programming considerably easier. First, instead of writing numeric instructions like `1110010011010000`, the programmer is allowed to use equivalent symbolic commands like `D=D-A`. Second, the programmer is allowed to create and use symbolic variables and labels, thus freeing the code from any references to physical memory addresses. The first assembly feature -- using mnemonics instead of binary codes -- is documented in the machine language specification. The second feature -- using symbolic references -- is the subject of this section.

Files Format

Machine language files: A machine language file is composed of text lines. Each line is a sequence of "0" and "1" characters, coding a single machine language instruction. Taken together, all the lines in the file represent a machine language program. The contract is such that when a machine language file `Prog.bin` 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).

By convention, machine language programs are stored in text files with a "`bin`" extension, e.g. `Prog.bin`. The name of the program (e.g. `Prog`) is not part of the language specification, but it is customary to refer to the program by the name of the text file in which it is stored.

Assembly language files: By convention, assembly language programs are stored in text files with an "asm" extension, e.g. `Prog.asm`. An assembly language file is composed of text lines, each representing either an *instruction* or a *symbol*:

- **Instruction:** one of the assembly language commands described in the previous section.
- **(Symbol):** This pseudo-command causes the assembler to assign the label 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. Each symbol can be defined only once in a program.

White space: Space characters, newline characters, and comments in the assembly program are ignored.

Comments: The following comment formats can be used in assembly programs, and are ignored:

```
// comment to end of line
/* comment until closing */
```

The assembly language is not case sensitive.

Symbolic Variables and Labels

Assembly commands can refer to memory locations (addresses) using either constants or symbols. Constants must be non-negative and are always written in decimal notation. A symbol can be any sequence of letters, digits, underscore ("_"), dot ("."), dollar sign ("$"), and colon (":"), that does not begin with a digit. Symbols are not case sensitive.

Symbols may occur in assembly commands in three different ways:

- **Symbolic references to addresses in the instruction memory:** These references, which serve as "labels" (destinations of *goto* commands), are defined using the "`(Xxx)`" syntax. This psuedo-command defines the symbol Xxx to refer to the instruction memory location that will eventually store the next translated command of the assembly program. Each symbol can be defined only once and can be used anywhere in the assembly program, even before the line in which it is defined.

- **Predefined symbols of special data memory addresses**: A special subset of RAM addresses can be referred to (by any assembly program) using pre-defined symbols. In particular, the following symbols are predefined:

  - **Virtual registers**: the symbols `R0` to `R15` are pre-defined to refer to addresses 0 to 15, respectively;

  - **VM pointers**: the symbols `SP`, `LCL`, `ARG`, `THIS`, `THAT`, and `TEMP` are pre-defined to refer to addresses 0 to 5, respectively. Note that each of these memory locations has two labels, e.g. address 2 can be referred to using either `R3` or `ARG`;

  - **Pointers to memory maps**: the symbols `SCREEN` and `KBD` are pre-defined to refer to addresses 16384 (0x4000) and 24576 (0x6000), respectively.

  The virtual registers are designed to simplify assembly programming. The VM pointers will come to play in the virtual machine implementation, discussed in Chapters 6 and 7. The memory map pointers point to the base addresses of the screen and keyboard memory maps.

- **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" and mapped to a memory location. Variables are allocated running addresses in the RAM after `R15`, i.e. starting from address 16 (0x0010). Hence, the first variable that appears in a program will be allocated address 16, the next one address 17, and so on.

5.3 Implementation

The *Hack assembler* is a program that translates programs written in the Hack assembly language to programs written in the Hack machine language. Both languages were specified in the previous section. This section gives a proposed design for the assembler, intended for a programmers.

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

```
prompt> Assembler Prog
```

The translation of each individual assembly command to its equivalent machine language command is direct and one-to-one. Each command is translated separately. In particular, each mnemonic component (field) of the command is translated to its corresponding binary bits according to tables 4-4/5/6, and each symbol in the command is resolved to its numeric address.

We propose an implementation that handles most of the assembly logic in a single object called Parser. This object parses the assembly commands and translates their mnemonic components to binary codes. Using this object, the main program is rather straightforward and may be written as a single class called Assembler.

5.3.1 The Parser

The Parser reads an assembly language command, parses it, and gives the main program convenient access to the command's components (fields and symbols). In addition, the parser removes all white space and comments from the code.

We propose implementing the parser as a **Parser** object with the following API:

`Parser(Reader input)`: Constructs a `Parser` object for the given input stream (Reader).

`boolean hasMoreCommands()`: Are there more commands in the input?

`void advance()`: Reads the next command from the input and makes it the *current command*. This method should be called only if `hasMoreCommands()` is true. Initially there is no *current command*.

`int commandType()`: Detects and returns the type of the current command. The assembly language has three command types: *C*-command, i.e. dest=comp;jump, *A*-command, i.e. @Xxx where Xxx is either a symbol or a decimal number, and label command, i.e. (Xxx) where Xxx is a symbol. This method detects the type of the current command and returns accordingly one of the following class constants: `C_COMMAND`, `A_COMMAND`, or `L_COMMAND`.

`String data()`: Returns the constant Xxx associated with the assembly commands @Xxx and (Xxx). Should be called only when `commandType()` returns `A_COMMAND` or `L_COMMAND`.

`byte dest()`: Returns the 3-bit binary code of the dest mnemonic of the command dest=comp;jump. The 8 possible codes are given in table 4-5. Should be called only when `commandType()` returns `C_COMMAND`.

`byte comp()`:Returns the 7-bit binary code of the comp mnemonic of the command dest=comp;jump. The 28 possible codes are given in table 4-4. Should be called only when `commandType()` returns `C_COMMAND`.

`byte jump()`: Returns the 3-bit binary code of the jump mnemonic of the command `dest=comp;jump`. The 8 possible codes are given in table 4-6. Should be called only when `commandType()` returns `C_COMMAND`.

5.3.2 Assembler for programs with no symbols

We suggest that the rest of the assembler be built in two stages. In the first stage, symbols are not handled, i.e. the assembler is designed to translate assembly language programs that only use constants and not symbols. In the second stage, the assembler is extended with symbol handling capabilities.

The contract for the first symbol-less stage is such 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 (b) the file contains no label commands, i.e. no commands of type `(Xxx)`.

The symbol-less assembler program can be implemented as one class called `Assembler`. First, the program creates am empty file named `Prog.bin`. Next, the program marches through the assembly commands in the supplied `Prog.asm` file. For each *C*-command, the program concatenates the translated binary codes of the command components, returned by the Parser, into a single 16-bit word. Next, the program writes this word into the `Prog.bin` file. For each *A*-command of type @Xxx, the program simply translates the decimal constant returned by the Parser into its binary representation and writes it into the `Prog.bin` file.

5.3.3 Assembler for programs with symbols

In order to create and maintain the correspondence between symbols and their meaning, the assembler program must manage a *symbol table*. The symbol table is a data structure that contains pairs of symbols and their corresponding semantics, which in our case are `RAM` and `ROM` addresses.

One complication that must be handled by the assembler is that symbols are often used in an assembly program before they are defined. A common solution is to write a 2-pass assembler that reads the code twice, from start to end. In the first pass, the symbol table is built and no code is generated. In the second pass, all the symbols mentioned in the program have already been bound to memory locations. Thus, for each symbol encountered in the assembly code in the second pass, the parser can replace the symbol with its corresponding meaning (number) in order to generate the final binary code.

Recall that the language specification specifies three types of symbols: *pre-defined symbols*, *labels*, and *variables*. The symbol table should contain all these symbols, as follows:

Initialization: Initialize the symbol table with all the pre-defined symbols and their pre-allocated `RAM` addresses, according to the language specification. In other words, add the pairs `(R0,0)`, `(R1,1)`, … to the table, up to (R15,15).

First pass: Go through the entire assembly program, line by line, and build the symbol table without generating any code. As you go through the program lines, keep a running number anticipating the ROM address that will eventually be allocated to the current command. This number starts at 0 and is increased by 1 whenever a *C*-command and an *A*-command is encountered, but does not change when a label command is encountered. Each time a label command "(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.

Second pass: Now go again through the entire program, and parse each line. Each time a symbolic *A*-command is encountered, i.e. the command is "@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 means that it represents a new variable. Hence, allocate a RAM address to it, say *n*, and add the pair (Xxx, *n*) to the symbol table. The allocated RAM addresses are running, starting at 16 (just after the 15 virtual registers).

5.4 Perspective

Work in progress.

5.5 Build it

The project consists of two parts: (a) writing assembly programs, and (b) writing the assembler itself. These parts are described in sections 5.5.1 and 5.5.2, respectively. The two parts are independent of each other, but we recommend doing the first before the second.

5.5.1 Assembly Programming

Objective: To give a taste of low-level programming in assembly language, and to introduce the overall Hack computer platform. In the process of working on this project, you will get a hands-on understanding of the assembly process, and you will appreciate visually how the target platform executes the translated binary code on the hardware.

Contract: Write and test the two assembly programs described below. The first program carries out a simple processing task. The second program illustrates basic handling of keyboard input and screen output. To get full credit, *your* programs must pass *our* tests, and they must adhere to the *Best Programming Practice* guidelines given in Appendix C.

How to write and test assembly programs: You are welcome to write and test your program in any way you see fit. At some point though, your program must pass some "official tests", as specified in a test script and a compare file supplied by us. Let us assume that the program name is `Prog`.

1. Create a new directory on your computer, e.g. `...hack/myPrograms/prog`. Put the supplied `Prog.tst` and `Prog.cmp` files in this directory. All the files mentioned below should be stored in this directory as well.

2. Use a text editor to write the required assembly program, and save it in `Prog.asm`.

3. Use our `Assembler` program (in either batch or interactive mode) to debug and translate your program. The result will be a binary file called `Prog.bin`.

4. Use our `CPUemulator` program to test your `Prog.bin` code. You can begin by loading `Prog.bin` into the simulator and testing it in interactive fashion. At some point, though, you must load our `Prog.tst` script into the simulator, and execute it. This script is programmed to load your `Prog.bin` and test it using a set of pre-planned tests. We suggest reading the contents of `Prog.tst` to familiarize you with theses tests.

The Programs

Multiplication program (`Mult`): The inputs of this program are the current values stored in `R0` and `R1`. The program computes the product `R0*R1` and stores the result in `R2`. The algorithm can be iterative addition.

Recall that the Hack platform maps `R0`,`R1`, and `R2` on RAM addresses 1,2, and 3, respectively. The contract (in this program) is that `R0`$\geq$`0`, `R1`$\geq$`0`, and `R0*R1<32768`. Your program need not test these conditions, but rather assume that they hold.

IO-Handling Program (`Fill`): The program runs an infinite loop that "listens" to the keyboard input. When a key is pressed (any key), the program blackens the screen, i.e. writes "black" in every pixel. When no key is pressed, the screen should be cleared. Note: this program has a test script (`Fill.tst`), but no compare file.

Steps: We recommend to proceed in the following order:

1. If you haven't done that already, download the software suite that accompanies the book. The `simulators` directory of the software suite includes the `Assembler` and `CPUEmulator` programs needed for this project.
2. Download the `project5a.zip` file and extract its contents to a directory called `project5a` on your computer.
3. Download and go through the *Assembler Tutorial*;
4. Write and debug the first program, using the supplied Assembler program;
5. Download and go through the *CPUEmulator Tutorial*;
6. Test your program using the supplied CPU Emulator;
7. Write and test the second program.

5.5.2 Writing the Assembler

Objective: To develop an assembler that translates programs written in Hack assembly language into Hack machine code. The assembler must implement the language specified in Section 5.2 of this chapter (*Hack Machine and Assembly Language Specification*).

Contract: When loaded into your assembler, a `Prog.asm` file containing a Hack assembly language program should be translated into the correct Hack machine code and stored in an Prog.bin file. When loaded into the CPU Emulator, the translated `Prog.bin` program should effect the behavior described in the program's documentation.

Implementation tips: We suggest building the assembler in two stages. First write a symbol-less assembler, i.e. an assembler that can only translate programs that use no symbols. Then extend your assembler with symbols handling capabilities. The test programs that we supply for this project come in two versions (without and with symbols), and thus you can test your assembler incrementally.

Although the API of the `Parser` object given in Section 5.3.1 is Java-oriented, it can be easily implemented in other languages as well. In languages other than Java the parser will not be initialized with a "`Reader`" object described in the API, but rather with the language's standard text stream type or directly with a file name.

If you write the assembler in Java, you may find it helpful to implement the symbol table using the `java.util.Map` interface (and its implementing classes) from the Java API collections library. If you use a Java version older than 1.2, you may use `java.util.Hashtable` instead.

Test Programs

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

Add: This program adds the constants 2 and 3 and puts the result in `R0`.

Max: This program performs the operation `R2=max(R0,R1)`.

Rect: This program 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`.

This program was written in the Jack programming language and translated by the Jack compiler into the supplied assembly program (about 31,000 lines of code). Running this game 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 this game will run much faster.

Steps: We recommend to proceed in the following order:

1. Download the `project5b.zip` file and extract its contents to a directory called `project5a` on your computer.
2. It's important to get a feeling of assembly programming before you write the assembler itself. Therefore we recommend doing the first part of the project (Section 5.5.1) first.
3. Another way to get acquainted with assembly programming and with the supplied test programs is to use the supplied Assembler and CPUEmulator to translate and run the supplied test programs (`Add`, `Rect`, `Max`, `Pong`).
4. Write and test your assembler program in two stages, as described above.