6. The Assembler[1]

1. Introduction

Work in progress.

2. Hack Assembly-to-Binary Translation Specification

This section gives a complete specification of the translation between the symbolic Hack assembly language to its equivalent binary representation. Unlike the language description given in chapter 4, this specification is both compact and formal. Therefore, it can be viewed as the contract that Hack assemblers must implement, in one way or another.

2.1 Syntax Conventions and Files Format

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 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 below.
- **`(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.

Constants and symbols in assembly programs: *Constants* must be non-negative and are always 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 in assembly programs: text beginning with two slashes ("//") and ending at the end of the line is considered a comment and is ignored.

White space in assembly programs: space characters are ignored. Empty lines are ignored.

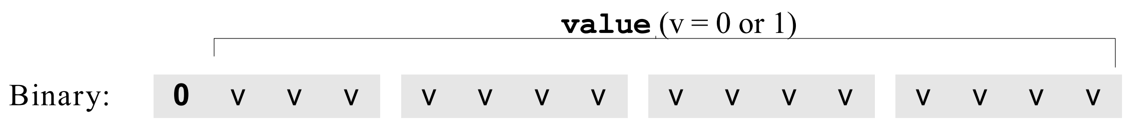
[1] From *The Elements of Computing Systems*, Nisan & Schocken, MIT Press, forthcoming in 2003, www.idc.ac.il/csd

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

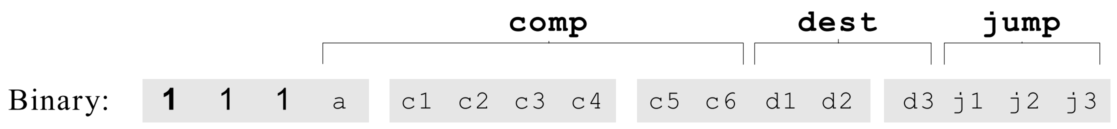
2.2 Instructions

The Hack machine language consists of two instruction types called *addressing instruction* (*A*-instruction) and *compute instruction* (*C*-instruction). The instructions 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 ommitted;
// If `jump` is empty, the ";" is omitted.



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

comp (a=0)	c1	c2	c3	c4	c5	c6	**comp** (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

2.3 Symbols

Hack assembly commands can refer to memory locations (addresses) using either constants or symbols. Symbols can be introduced into assembly programs in three ways:

Predefined symbols: Any Hack assembly program is allowed to use the following pre-defined 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 5 top RAM locations can be referred to using two pre-defined 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 user-defined 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).

2.4 Example

In chapter 4 we presented a program that sums up the numbers between 1 and 100. Program 1 repeats this example, showing both its assembly and binary version.

Assembly code

```
// sum the numbers 1...100
      @i     // i=1 (allocated at 0x0010)
      M=1
      @sum   // sum=0 (allocated at 0x0011)
      M=0
(loop)
      @i     // if i-100>0 then goto end
      D=M
      @100
      D=D-A
      @end
      D;jgt
      @i     // sum+=i
      D=M
      @sum
      M=D+M
      @i     // i++
      M=M+1
      @loop  // goto loop
      0;jmp
 (end)
```

Binary code

```
(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)
```

PROGRAM 1: Assembly and binary representations of the same program. If the assembler is given the text file on the left, it should generate the text file given on the right.

3. Implementation

The previous section gave a complete specification of the Hack language, in both its assembly and binary versions. The program that translates assembly programs into binary programs according to this contract is called the *Hack assembler*. This section describes a proposed design for this assembler.

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

```
prompt> Assembler Prog
```

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 command is translated into its corresponding bit-code according to the tables in section 2.2, and each symbol in the command is resolved to its numeric address as explained in section 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 for different mnemonics, a `SymbolTable` module that handles symbols, and a main program that drives the entire translation process.

3.1 The Parser

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 Module			
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 assembly language 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`	--	Enumeration: • `A_COMMAND` • `C_COMMAND` • `L_COMMAND`	Returns the type of the current command: • `C_COMMAND` for `dest=comp;jump` • `A_COMMAND` for `@Xxx` where `Xxx` is either a symbol or a decimal number • `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. The 8 possible mnemonics are given in section 2.2. Should be called only when `commandType()` is `C_COMMAND`.
`comp`	--	string	Returns the `comp` mnemonic in the current C-command. The 28 possible mnemonics are given in section 2.2. Should be called only when `commandType()` is `C_COMMAND`.
`jump`	--	string	Returns the `jump` mnemonic in the current C-command. The 8 possible mnemonics are given in section 2.2. Should be called only when `commandType()` is `C_COMMAND`.

3.2 The Code Module

The `Code` module translates Hack mnemonics into their respective binary codes. The details are given in the following API.

Code Module			
Translates Hack assembly language mnemonics into binary codes.			
Routine	**Arguments**	**Returns**	**Function**
`dest`	string mnemonic	3 bits	Returns the 3-bit binary code of the `dest` mnemonic, as listed in section 2.2.
`comp`	string mnemonic	7 bits	Returns the 7-bit binary code of the `comp` mnemonic, as listed in section 2.2.
`jump`	string mnemonic	3 bits	Returns the 3-bit binary code of the `jump` mnemonic, as listed in section 2.2.

3.3 Assembler for programs with no symbols

We suggest building the rest of 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 file contains no label commands, i.e. 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 it into the `Prog.hack` file.

3.3 The SymbolTable Module

Since Hack instructions are allowed to use symbols, the symbols must be resolved 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.

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. In general, the most appropriate data structure for representing such a relationship is the classical *hash table*. In many programming environments, 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 Module			
A symbol table that keeps a correspondence between symbolic labels and numeric addresses.			
Routine	**Arguments**	**Returns**	**Function**
`Constructor`	--	--	Creates a new empty symbol table
`addEntry`	`string symbol,` `int address`	--	Adds the pair (`symbol`, `address`) to the table.
`contains`	`string symbol`	`boolean`	Does the symbol table contain the given symbol?
`addressOf`	`string symbol`	`int`	Returns the address associated with the symbol.

3.5 Assembler for programs with symbols

Once we have a symbol table in place, the handling of symbols in the translation process is straightforward. Whenever we encounter a new symbol in the program, we allocate a numeric address to it, and add the pair (*symbol*, *address*) to the table. To translate an instruction that includes a symbol into binary code, we simply look-up the symbol in the symbol table, retrieve its numeric address, and plant it in the translated instruction. This symbol handling capability is all we need to complete the assembler's implementation

There's one complication though: in assembly programs, label symbols (used in *goto* commands) are often used before they are defined. One 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 label symbols encountered in the program have already been bound to memory locations. Thus, in the second pass the assembler can replace them with their corresponding meanings (numbers) in order to generate the final binary code.

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

Initialization: Initialize the symbol table with all the pre-defined symbols and their pre-allocated RAM addresses, according to Section 2.2.

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 anticipating the ROM address that will eventually be allocated to the current command. 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 pseudo-command or a comment is encountered. Each time a pseudo 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.

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, i.e. "`@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 the next available RAM address to it, say *n*, add the pair (`Xxx`, *n*) to the symbol table, and complete the command's translation. The allocated RAM addresses are running, starting at address 16 (just after the addresses allocated to the pre-defined symbols).

This completes the assembler's implementation.

4. Perspective

Work in Progress.

5. Build it

Objective: To 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 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 supplied *Assembler Tutorial*.

Contract: When loaded into your assembler, a `Prog.asm` file containing a 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.

Testing: We suggest building the assembler in two stages. First write a symbol-less assembler, i.e. 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 for this project 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.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`.

The *Pong* program was written in the *Jack* programming language (described in Chapter 9) and translated by the *Jack compiler* (described in Chapters 10-11) into the supplied assembly program. The resulting machine-level program is about 20,000 lines of code, which also include the Sack operating system (described in Chapter 12). 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 in the book this game will run much faster.

Steps: We recommend proceeding in the following order:

1. Download `project6.zip` and extract its contents into a directory called `project6` on your computer, without changing the directories structure embedded in the zip file.
2. Write and test your assembler program in the two stages described above. You may use the assembler supplied with the book to compare the output of your assembler to the correct output. This can be done by treating the `.hack` file generated by your assembler as the compare file used by the supplied assembler. For more information about the supplied assembler, go through the *Assembler Tutorial*.