Assembler

Friday, February 25, 2022 3:15 PM

"What's in the name? That which we call by any other name would smell as sweet."

- *Shakespeare, Romeo and Juliet*

6.1 Background

Binary Machines

Binary machines take binary code that is an agreed upon package of micro-codes designed to be decoded and executed by some target hardware platform.

What we went over in Chapter 4 - Machine Language

Symbol Machines

Lets look at the instruction goto 312. Assume goto is some fetch cycle that will fetch 312 and bring the process to that address. Now lets say that address is actual the start of a loop or any arbitrary important start of code. Then we can now say goto LOOP where LOOP is the address 312.

All we have to do is store somewhere that LOOP = 312. Now when we assemble the code. Everywhere we have LOOP it will be replaced with 312.

We now know what symbols are

Symbols are used for three purposes:

- Labels
- Variables
- Predefined symbols (i.e. SCREEN KBD)

Symbol handling is one of the most important functions of the assembler.

6.2 The Hack Machine Language Specification

See Chapter 4 - Machine Language for basics on the Hack MLS

6.3 Assembly-to-Binary Translation

An assembler takes in a stream of assembly instructions and generates its output as a stream of translated binary instructions.

6.3.1 Handling Instructions

For each assembly instruction, the assembler:

- Parses the instruction into its underlying fields
- For each field, generates the corresponding bit-code, as specified in figure 6.2
- If the instruction contains a symbol reference, resolves the symbol into its numeric value
- Assembles the resulting binary codes into a string of sixteen 0 and 1 characters
- Writes the assembled string to the output file

6.3.1 Handling Symbols

Assembly code will sometimes contain symbols. Symbols can be mentioned before they're defined. This makes the job of the assembly code writers easier and that of the assembler developers harder. To solve this issue assemblers will take two passes of code. One to initialize all of the symbols into a symbol table and the next to initialize their values to the corresponding symbol.

Initialization:

The assembler creates a symbol table and initializes it with all the pre-defined symbols and the pre-allocated values (R0…R15, KBD, etc.).

First pass:

The assembler will go line by line making sure to keep track of the line numbers.

It will also increment a number starting at 0 every time it encounters an A or C-instruction. It will not increment when it encounters a comment.

Every time the assembler encounters a (XXX) label it will add it to the symbol table associating it with the current line number + 1.

No Code is generated in the first pass.

Second pass:

Every time an A instruction is encountered namely @XXX, where XXX is a symbol and not a number the assembler looks up XXX in the symbol table. If the symbol is found the assembler replaces it with its numeric value and completes the instructions translation.

If it is not found then it must represent a new variable. The assembler will:
(I) add the entry <XXX, value> where value is the next available address in RAM space designated for variables.
(II) complete the instruction translation by then replacing it with the numeric value

6.4 Implementation

Cmd line argument:

Prompt>HackAssembler prog.asm

The output is a file called prog.hack

6.4.1 Developing a Basic Assembler

The basic assembler (start with this) will translate basic assembly with NO SYMBOLS. So it is either processing white space, A-instructions, C-instruction, or comments.

A recommendation of Parser -> Code -> Hack approach to developing this assembler.

API Documentation

Each project concerning software will give an API specification for the software we will build.

The rest of the chapter is specific to developing the HACK assembler. I left it out of notes because I spent around 7+ hours coding the assembler itself and the API is just that so I do not need to include it in the notes.