Translators

"*Things didn't become simpler without reason. It's the mind effort we made made it happen."*

Assembler

Into

Congratulations! We really appreciate your comprehension which enables you to implement PC and learn that complicated hack machine language. However, one must be aware of the fact that we create machine language because our PC is, at this moment, foolish and cannot comprehend our thoughts.

So what can we do? We cannot just leave it like this.

Let us not **forget**, my friends, that we were once the same foolish as our machine is, when we were kids. But now we are not anymore, not just because we *learnt things*, but because we were *guided to learn* as well.

That's why in this chapter we will ***focus*** on ***how a machine can learn***, and ***by what means we can guide it***.

How can a machine learn?

To answer this question we must first try to figure out the answer to the other one. That is: **"How can we learn?"**

Unfortunately we are not able to give you a perfect answer because of the complicated mechanism of learning inside our brain. But it's okey for us to suppose that when you learn something, there must be a **physical evidence** in your organism. For us, it might be a curve in our brain; for machine, it can be a new chip added or a new function implemented.

~~(One critical advantage we have over machine now is that this physcial improvement can be made by ourselves. A machine cannot optimize its hardware by itself, at least for now(。・・)ノ.)~~

By what means can we guide it?

Assembly

You might have noticed by now that when we were trying to teach you machine language, we've given a great deal of names to those "1"s and "0"s such as"A-instructions", "C-instructions", "comp field", "jump field", etc.

-"WHY? "

-"Because it helps us understand? "

-"Correct! But HOW? "

Now we are talking.

Names, or to be more accurate, symbols, provide abstraction. Abstraction stands for connection. A symbol that refers to a group of certain elements creates sequence and order. Instead of memorizing ***all*** the details, we just need to understand ***each*** one of them, and build a symbol table to refer to when necessary. Moreover, if the symbol table can be read and carried out by our machine, we will have to manufacture those hard works no more.

This *table* is exactly what we are going to achieve in this chapter: The assembly.

Now, before we move on, we will have to point out that from this project on, you'll need at least one programming language. If you know no programming language at all, you might as well start learning one to finish the subsequent project.

Specification for Assembly

Assembly, is somehow like a bijection. (Almost)Every line hack machine language you've learnt before can find its counter part in our assembly. So, the main purpose of creating assembly is to make machine language more easy to write.

Eg: instead of writing 0000000000000010 (if you learnt to write a program in machine language you will know how hard it is), we first understand that we are writing an A-instruction, or a "noun", literally. And for noun, we give it a symbol **@**. And since we know exactly how to deal with decimalism and the binary system, we simply use the decimal number. So in assembly, this line can be written in "***@2***".

Specificaion for A-Instruction

We all know that there are two types of A-instruction: "@142" and "@qwe".***(Really?)*** The former gives us a specific number to refer to and is of more practical use(that is, when you are calculating). The latter is more like a symbol of address, telling people that "I'll store a number in a memory that is called ***qwe.***", and the assembler automatically assign the "qwe" a number(starting from 16). And then the line "@qwe" will always be translated into "@13".

Please, stop for one second and see what we've just done. Our thoughts, though only a small part of it, will be able to be understood by our machine through this new program. ***How did it happen? Why can't our other thoughts?*** We will leave that question to you for now.

Because we decided and started to make a difference. The programming process will seem to be quite simple, and that is because **we did the difficult part for you**. (Which part?) You will find it not so easy when you are facing the problems that follow with bare hands. We are happy to remind you that they seem to be hard because you are making efforts, which means ***you care***, and ***are trying*** to find a solution.

Just some more details in this section:

1. When meeting a line starting with "@", it goes into "A-Instruction Class".

2. When you see any non-digital symbol in the following, you assign a memory address to that symbol, otherwise you translate the decimal number into binary number, making sure it is sixteen bit.

Now, why not get your hands dirty and **begin** achieving some of our functions?

(Go to our subproject 1 and find your task~)

Specification for C-Instruction

Unlike A-Instructions, C-Instructions(verbs) are more complicated. **Like** A-Instructions, C-Instructions can also be classified. Remember, there in nothing but symbols, and combinition of symbols. The ability you must aquire to build all translators is figuring out the proper connection between one group of elements and another. In this particular section, it is how the symbol table is generated and how it works.

Take a look at the following example:

Machine Codes	Assembly
0000 0000 0001 0000	@counter
1110 1111 1100 1000	M=1
0000 0000 0001 0001	@sum
1110 1010 1000 1000	M=0
	(LOOP)
0000 0000 0001 0000	@counter
1111 1100 0001 0000	D=M
0000 0000 0110 0100	@100
1110 0100 1101 0000	D=D-A
0000 0000 0001 0010	@END
1110 0011 0000 0001	D;JGT
0000 0000 0001 0000	@counter
1111 1100 0001 0000	D=M
0000 0000 0001 0001	@sum
1111 0000 1000 1000	M=D+M
0000 0000 0001 0000	@counter
1111 1101 1100 1000	M=M+1
0000 0000 0000 0100	@LOOP
1110 1010 1000 0111	0;JMP
	(END)
0000 0000 0001 0010	@END
1110 1010 1000 0111	0;JMP

This is a powerful example. Here are two A-class codes with labels. Look how their address are assigned. And also there are loops, too, making it easy to do the following clarifications.

- Codes like (LOOP) will not be translated. Nor will it be considered one line.(***So what are such lines for?***)
- When an A-Instruction has the same symbol existing inside a parenthesis, say "@LOOP" the rules changed. That line should be translated into the address which the label (LOOP) are **pointing at**(that is, the address of the line that follows the label). In our example you can see it clearly.
- Go to the previous page with a "(really?)". Now you know that there in fact are three types of A-Instructions. That thought should go directly into your translator.
- Where is our *"Table"*? Try to figure that out.