Topic: 1.4.4 Assembly language

Machine code - simple instructions that are executed directly by the CPU

As we already know from chapter 1.1.1, computers can only understand binary, 1s and 0s. We are now going to look at the simplest instructions that we can give a computer. This is called machine code.

Machine code allows computers to perform the most basic, but essential tasks. For this section we are going to use the Accumulator (you met this register earlier) to store the intermediate results of all our calculations. Amongst others, the following instructions are important for all processors:

- **LDD** - Loads the contents of the memory address or integer into the accumulator
- **ADD** - Adds the contents of the memory address or integer to the accumulator
- **STO** - Stores the contents of the accumulator into the addressed location

Assembly code is easy to read interpretation of machine code, there is a one to one matching; one line of assembly equals one line of machine code:

Machine code	Assembly code
000000110101 =	Store 53

Machine code and instruction sets

There is no set binary bit pattern for different opcodes in an instruction set. Different processors will use different patterns, but sometimes it might be the case that you are given certain bit patterns that represent different opcodes. You will then be asked to write machine code instructions using them.

Relationship between Assembly language and Low Level Language:

```
LOAD 253         Assembly Code
0000 11111101    Machine Code
```

For every assembly language command there is an equal machine language command. i.e. the relationship is 1:1.

Topic: 1.4.4 Assembly language

Below is an example of bit patterns that might represent certain instructions.

Machine code	Instruction	Addressing mode	Hexadecimal	Example
0000	STORE	Address	0	STO 12
0001	LOAD	Number	1	LDM #12
0010	LOAD	Address	2	LDD 12
0100	ADD	Number	4	ADD #12
1000	ADD	Address	8	ADD 12
1111	HALT	None	F	END

An assembler converts an assembly language program into machine language program. Machine code is binary. So an Assembler converts a program that looks like the program I wrote below into something like "10100110010101110101".

Single pass and two pass assembler:

One-pass assemblers go through the source code once and assume that all symbols will be defined before any instruction that references them. It has to create object code in single pass and it cannot refer any table further.

Two-pass assemblers does two passes as it creates a table with all symbols and their values in the first pass, then use the table in a second pass to generate code and the length of each instruction on the first pass must be determined so that the addresses of symbols can be calculated. It can be referring further and it does its actual translation in second pass and convert instruction into machine code. Let's look at this assembly instruction:

```
Op-Code operand
  ADD    A 20
```

Here Add is the operation and A and 20 are the operands. A programmer writes a bunch of these single line statements that make up the assembly language program. Each line in this program is converted into Binary format and loaded into consecutive memory locations when this program is called for execution. These instructions in consecutive memory locations are executed one after the other unless something called a Jump instruction is encountered. A jump instruction asks the system to go to a particular memory location (Marked by a Label in the source code) and start executing code from there.

The problem with this is that the current instruction under execution may ask you to jump to a particular memory location marked by label which the assembler has not yet encountered. See in the example below the first instruction is to jump to X, but X is not defined till line 5. To translate an assembly statement to machine code you need the exact value of the operand in that statement. If the operand is referring to a value in a register, you need the register number. If operand is referring to a value in a memory location, you need the address of that memory location. If the operand is a hardcoded value, you need that value. If you operand is referring to a particular line number in your source code that is marked by a label to jump to, you need that line number. Without this information you can't translate an assembly statement to machine code.

```
1   JMP x
2   y:
3   ADD A 30
4   JMP z
5   x:
6   MUV A 20
7   JMP y
8   z:
```

To solve this problem there are two approaches, hence leading to two types of assemblers.

(1) Two Pass Assembler: In older days when memory was a limitation people used to use a two pass assembler. A two pass assembler has a symbol table which is a table with two columns, symbol name, and symbol value. When it see the first sentence JMP x. It doesn't know what a x is, so it creates a new entry in the symbol table for x with value field un-initialized. When it gets to instruction 5. It sets the x value as 5.

So after reading the whole code(After first pass). The table will look something like this.

Symbol Name	Symbol Value
X	5
Y	2
Z	8

In the next pass it just runs through each sentence and converts the program sentence by sentence into machine code using the symbol table.

(2) Single Pass Assembler: In the Single Pass Assembler, we use a different kind of symbol table. Unlike the Two Pass Assembler where we stored the name of a symbol and its value, in one Pass Assembler we use a table where we have a symbol and all the locations where we have encountered that symbol. As soon as we encounter the initialization for a symbol we use the symbol table to jump to all the locations where the symbol was encountered, plug in the value we just read, convert those lines into machine code, and continue back from the line where we encountered the initialization.

As you can clearly see, the Single Pass Assembler needs a lot of memory at runtime. Much more than Two Pass. In olden days people could barely fit the assembly program in memory so people used to use the Two Pass Assembler.

Thanks to Sastry Aditya for this passage.

Topic: 1.4.4 Assembly language

Symbolic addressing:

So far, we have discussed mnemonics, opcodes and operands. But another key aspect of programming is to fetch or store data and instructions from memory.

The simplest way to do this is to refer directly to a memory location such as #3001. But this brings a number of problems

- it is difficult to see the meaning of the data in location #3001
- the data may not be able to be located at #3001 because another program is already using that location.

To overcome this issue, the idea of 'symbolic addressing' is used. Instead of referring to an absolute location, the assembly language allows you to define a 'symbol' for the data item or location. Like this:

`VarA DB` define a variable called VarA as a byte
`VarB DW` define a variable called VarB as a word
`MOV AL,[VarA]` Move data in VarA into register AL

The symbols being defined by this bit of code are VarA, VarB. The size of the variables VarA and VarB are defined, but notice that their location is not defined.

It is the job of the assembler to resolve the variables into locations in memory.

The advantages of using symbolic addressing over direct memory references are:

- The program is re-locatable in memory. It does not particularly care about its absolute location, it will still work
- Using symbols makes the software much more understandable

When the code is ready to be loaded and run, a 'symbol table' is created by the assembler for the linker and loader to use to place the software into memory.

Absolute addressing in assembly:

When a memory address is used in assembly language command it is absolute addressing. The numeric number that is used to represent the memory location is supposed to be numeric label. Like `LDD 253, ADD 56 etc.`

Assembler Directives:

Assembler directives are instructions to the assembler to perform various bookkeeping tasks, storage reservation, and other control functions. To distinguish them from other instructions, directive names begin with a period. Three common directives: .data, .text, and .word. The first two (.data and .text) are used to separate variable declarations and assembly language instructions. The .word directive is used to allocate and initialize space for a variable.

Topic: 1.4.4 Assembly language

Assembly Language Macros:
A macro is an extension to the basic ASSEMBLER language. They provide a means for generating a commonly used sequence of assembler instructions/statements. The sequence of instructions/statements will be coded ONE time within the macro definition. Whenever the sequence is needed within a program, the macro will be "called".

- Most assemblers include support for macros. The term macro refers to a word that stands for an entire group of instructions.
- Using macros in an assembly program involves two steps:
  1: Defining a macro:
  The definition of a macro consists of three parts: the header, body, and terminator:
  ```
  <label> MACRO The header
  . . . . The body: instructions to be executed
  ENDM The terminator
  ```

  2: Invoking a macro by using its given <label> on a separate line followed by the list of parameters used if any:
  ```
  <label> [parameter list]
  ```