Assemblers

An assembler is a piece of software that takes a program written in the assembly language of a particular processor (source program), and translates it into the equivalent binary machine code program (object program). It will also offer additional facilities to assist in the development of low level language programs.

Tasks Performed by an Assembler

Analysis of the Structure of the Assembly Language Program

Each assembly language has a set of syntax rules that determine the structure of the program. They are usually a lot simpler than the syntax rules of a high level language like C or Pascal, but they are still important. A typical set of rules might be that the source code is divided up into columns, which contain the following items:

Labels	Instruction code	Operand	Comment
num:	**LDA**	**$0330**	**; This is a comment**

The assembler may enforce strict layout rules so that it can easily distinguish between labels, instructions and so on.

Decoding Mnemonic Instruction Codes

This is straightforward. There is a one-to-one correspondence between mnemonic instruction codes and binary machine operation codes. Somewhere in the assembler program there will be a lookup table containing both sets of codes. Each mnemonic instruction is looked up in the table and the corresponding machine operation code is placed into the object program. If the mnemonic instruction is not in the table, an error message is displayed.

Dealing with Symbolic Addresses

The objective here is to replace each label with the actual memory address that it refers to.

This is often dealt with by making two passes through the source code.

First Pass
Each time a label is encountered in the first column, enter it into a table (often called the symbol table) and also enter the actual memory address that the label refers to. This can be worked out because the assembler knows where the program is to start in memory, and it knows how long each instruction is going to be.

Second Pass
Each time a label is encountered in the third (Operand) column, look it up in the table and place the actual memory address that it refers to into the Object program.

Automatic Data Conversion

If the assembler allows you to use different number bases and character codes, then conversion algorithms and tables will be needed to convert them into their binary equivalents so that they can be placed in the object code program.

Most assemblers allow you to use characters in quotes, instead of having to look up the ASCII code. Some allow you to specify numbers in Hex, or Decimal, or even Octal. This means that the assembler will have to convert them for you, using one of the standard algorithms that exist.

Interpreting Directives

Directives are assembly language instructions that are not converted into binary machine code instructions. They are actually instructions to the translation program itself.

Examples

A directive that tells the assembler where it should place the assembled code in memory.

```
      .ORG  $0200       ; This is a 6502 assembler directive.
```

A directive that allows you to reserve a word of memory and give it a label.

```
      num:  .DW   $00   ; This is a 6502 assembler directive.
```

These can vary from assembler to assembler, even if both assemblers are for the same model of processor.

Macro Expansion

Some assembly languages allow the use of macros. In this context, a macro instruction is a single instruction which is used to represent a set of instructions.

When the macro definition is encountered during assembly, the assembler stores the name of the macro in a table, along with the machine code instructions for the assembler code in that macro.

When the macro instruction is encountered during assembly, the assembler inserts the set of machine code instructions that it represents, into the object program.

It is common for programmers to develop libraries of useful macro routines which can be used by any program that they are working on. It saves them having to re-invent the wheel all the time.

For example, on the 6502 simulator, you can define macros by having something like this at the top of your program:

```
Put:  .MACRO chr
      LDA #chr          ; load value of parameter 'chr'
      STA io_putc       ; Output it
      .ENDM
```

This defines a small macro instruction called 'Put' which expects one parameter called chr. The macro simply loads the value of the parameter into the accumulator and outputs it.

You would use it from within the main body of your program by doing something like this:

```
      LDA #$0120        ; These instructions have no significance
      STA num           ; They are just here to show how you would
      Put 'A'
      LDA #$0111        ; Include a macro in your main program
```

A macro is not technically a subroutine – there is no jump, no return. The stack is not used. It is just that when the assembler sees the macro, it will go to a table and pull out the machine code instructions for the macro, and insert them into the assembled machine code at that point.

Language of Assemblers

The very first assembler must have been written directly in machine code.

Nowadays, they will often be written in assembly language themselves – although you will want to develop them on a different, tried and tested assembler, until you are sure that you have gotten rid of the bugs.

There is nothing to stop you writing one in a high level language like C or Pascal – it would certainly be easier. Some assemblers can produce machine code for a processor different to the one that the assembler is running on. They are called **Cross Assemblers**.