Lecture Notes on Assembly Language - J. Vaughan

11. Program Structure: Executable Code

The data definition sections are followed by executable code. This can be classified in three subsections: macros, subroutines and the main program.
Subroutines are sections of code that perform specific functions on data provided by the main program (or by another subroutine) as part of a subroutine call and store the results in such a way that the calling program can access them.

Macros are structured in a similar way to subroutines but, instead of being called, they are expanded in the line of the calling code and therefore they do not terminate with a RETURN instruction. Macro expansion can be considered as a preliminary phase in program translation during which the number of instructions in the program is increased at points where macro calls occur.

12. Instructions and data

The beginning of the executable code section is marked by

```
SECTION .text
```

There are two schools of thought on where to begin the main program: One takes the view that the main program contains the principal narrative and that the detail can be filled in later. In this approach, any subroutines are placed after the end of the main program. This results in forward references to not-yet-defined labels, which is not really a problem in a 2-pass assembler. The second approach is to place the subroutines before the main program, so that the addresses of subroutine calls are defined before the calls are encountered in the code. This is also the approach of structured programming, and was enforced by teaching languages such as Pascal. The nasm assembler does not enforce any particular design philosophy. However, like most assemblers, it recognises that the first instruction in the object file may not be the intended initial instruction. Therefore, a mechanism is provided for notifying the linker of the position of the first instruction, or entry point:

```
global main

main:
```

Most instructions operate on data. As a result, there are many ways of identifying where the data is located. These ways are referred to as addressing modes. Instructions can use different modes for addressing source and destination data.

The main modes are: implied, register, immediate, direct, register indirect and memory indirect. Indexing and based addressing are types of register indirection.

Implied addressing

The location of the operand is implied by the nature of the instruction, e.g. complement bits of the accumulator.

Register addressing

The operand source or destination is a register. Since there are several of these, several bits of the instruction must be allocated to distinguishing between them. At assembly language level, this translates as having particular names for individual registers, e.g.

```
mov eax, ebx
```

Immediate addressing

The data is located in memory within the instruction that operates on it. The most common example of this occurs when moving a constant into a register, for example in a loop count:

```
numchar     equ     26
            mov ecx, numchar
```

Direct addressing

The data is located in a memory location. The address of this location is located within the instruction that operates on the data, for example,

```
num1        db 1
            mov al, [num1]
```

In the interests of program legibility, it might be better to write the latter instruction as

```
            mov al, byte[num1]
```

in order to make it clear that it is the byte (rather than the word, doubleword, etc.) having address num1 that is being moved, although this can be deduced from the size of the destination register.

Register indirect addressing

The data to be operated on by the instruction is in memory. The address of the memory location is contained in a register, and the register is referenced in the instruction. This approach is useful, particularly in array processing, as it allows rapid modification of the operand address by changing register contents. For example:

```
numarray    db 1, 1, 2, 3, 5, 8, 13, 21, 34, 55
numlen      equ $-numarray
            mov ecx, numlen
            mov ebx, 0
add1        mov al, byte[numarray+ebx]
            inc al
            mov byte[numarray+ebx], al
            loop add1
```

The IA-32 architecture permits the combination of two registers and a constant in the manufacture of an operand address, a scheme traditionally named base + index + displacement.

Memory indirect addressing

The data to be operated on by the instruction is in memory. The address of the memory location is contained in a different memory location, and this latter memory location is addressed in the instruction. This addressing mode is not used for operand addressing in IA-32, but a limited form of it is available for indirect jumping.