

Lecture Notes on Assembly Language - J. Vaughan

1. Introduction

Assembly Language is a generic term for low-level programming languages in which the instructions, expressed in an alphabetic mnemonic form, have a 1-to-1 correspondence with equivalent register-level machine codes. Assembly languages have an architectural dependency: there is at least one assembly language for each processor. These notes assume an underlying Von Neumann architecture, in which instructions are stored in a memory unit which is separate to the Central Processing Unit (CPU). The CPU's principal mode of operation is one in which an Instruction Cycle is repeated. The cycle consists of fetching an instruction from the memory unit and executing it within the CPU. The Instruction Cycle is generally regarded as being composed of two phases, called the Fetch Phase and the Execute Phase.

The Fetch Phase may be described as follows:

$IR \leftarrow (m \langle PC \rangle)$

This is a rough Register Transfer Language notation in which IR is the Instruction Register, PC is the Program Counter, the left arrow stands for "gets" or "is assigned", the parentheses () mean "the contents of" and the less than and greater than signs $\langle \rangle$ stand for "the address of".

Therefore, the above notation may be read as;

"The Instruction Register is assigned the contents of the memory location whose address is given by the contents of the Program Counter."

A group of instructions, intended to perform some function when executed in combination, is referred to as a program.

2. Why do Assembly Language?

- Greater visibility of computer components: registers, ALU, memory, input-output subsystem, interrupt subsystem.
- Close to hardware, so the programmer can take advantage of his/her awareness of the physical layout of the computer and controlled system.
- Very suitable for small-scale, no-frills programming, e.g. in embedded systems.
- The programmer can optimise code for speed of execution or size (amount of memory used).

3. Machine Code

An instruction in machine code is composed of a pattern of bits. This pattern is held in the Instruction Register (IR) and interpreted and sequenced by the Instruction Decoder. Note that the decoder may itself be programmed at a lower level, but this is not of interest to assembly language programmers.

The length (in bytes) of an instruction can be fixed or variable.

Most processors use a variable-length instruction format, but fixed-length instructions may still be encountered.

4. Assembly Language

A 1-to-1 correspondence between assembly language instructions and Machine Code. The instructions are written as patterns of letters that suggest the operation that the instruction will perform. Each pattern helps the programmer to remember the general function of the instruction, so they are quite often referred to as *mnemonics*, meaning memory aids.

5. Instruction types

The entire range of instructions available is called the Instruction Set.

The programmer sees registers (by name) and memory (usually by byte address).

The programmer sees the ALU indirectly by virtue of the arithmetic and logical operations that are available in the instruction set.

The programmer is aware of other issues, such as the way in which the stack is implemented and the different ways in which a memory address can be formed.

The instructions can be grouped according to general function.

There are usually 1-byte instructions to do nothing (NOP) or to halt the instruction cycle (HLT).

The **Data Movement** group contains instructions that move data from 1 register to another or between registers and memory. These are used to set up the inputs to the ALU while ensuring that other necessary data is preserved.

The **Test and Branch** instructions are used to implement control structures such as if statements and loops. The Intel processor approach is to set condition flags as a result of an ALU operation. An older approach was to set a condition flag based on the contents of a register.

The **Arithmetic and Logic** group contains instructions that specify ALU operations, such as addition, subtraction, multiplication, division, shifting and rotating.

Other instructions are often bunched together in a miscellaneous collection called the **Stack, I/O and Machine Control Group**.

Stack instructions are those that manipulate the stack or the Stack Pointer explicitly. They include PUSH and POP, but CALL and RET are usually included with the Test and Branch group.

The input-output subsystem can have a separate I/O address space (INTEL) or may be memory-mapped (Motorola). Even in the former case, memory-mapped I/O can be provided, but this is unusual. You can recognise the presence of a separate I/O address space by the presence of specific **I/O instructions** such as IN and OUT.

The **Machine Control** subgroup contains instructions that can influence processor operation by setting or clearing control flags. Examples are STD, CLD (set and clear the Direction Flag), STI/EI, CLI/DI (enable and disable interrupts). The NOP and HLT instructions are usually included in this subgroup.

Note: CLD causes edi and esi to be incremented on a string move, while STD causes them to be decremented. STI and CLI affect the Interrupt Enable (IF) flag. When set/cleared, this flag enables/disables maskable external interrupts.

6. Instruction Format

At Assembly Language level, "instruction format" means that the instruction opcode (i.e. the mnemonic) is followed by the operands on which the instruction acts. If there are two operands, and one of them is the destination for the result of the operation, then the format of "Intel syntax" assemblers such as **MASM** and **nasm** is

opcode *destination, source*

There are other assemblers, such as **as** or **gas**, which use "AT&T syntax", which is

opcode *source, destination*

6a. Register Architecture

We will be working with Intel x86 assembly language. We will assume a 386 configuration of 8 x 32-bit registers:

eax, **ebx**, **ecx**, **edx** are general-purpose registers.

esi and **edi** are source index and destination index registers.

esp and **ebp** are the Stack Pointer and Base Pointer respectively.

Bits 0 – 15 of **eax** can be treated as 16-bit register **ax**.

Bits 0 – 15 of **ebx** can be treated as 16-bit register **bx**.

Bits 0 – 15 of **ecx** can be treated as 16-bit register **cx**.

Bits 0 – 15 of **edx** can be treated as 16-bit register **dx**.

Bits 0 – 7 and 8-15 of **ax** can be treated as 8-bit registers **al** and **ah** respectively.

Bits 0 – 7 and 8-15 of **bx** can be treated as 8-bit registers **bl** and **bh** respectively.

Bits 0 – 7 and 8-15 of **cx** can be treated as 8-bit registers **cl** and **ch** respectively.

Bits 0 – 7 and 8-15 of **dx** can be treated as 8-bit registers **dl** and **dh** respectively.

One of the reasons for divisibility of **eax**, **ebx**, **ecx** and **edx** is for backward compatibility through the 8086 16-bit processor to the 8080 8-bit processor.

Originally, **a**, **b**, **c** and **d** were held to stand for accumulator, base, count and data registers, due to some special functions of these registers.