## LECTURE 17

## TOPIC. INTRODUCTION TO ASSEMBLY LANGUAGE

Translators can be roughly divided into two groups, depending on the relationship between the source language and the target language. When the source language is essentially a symbolic representation for a numerical machine language, the translator is called an assembler and the source language is called an assembly language. When the source language is a high-level language such as Java or C and the target language is either a numerical machine language or a symbolic representation for one, the translator is called a compiler.

### What Is an Assembly Language?

A pure assembly language is a language in which each statement produces exactly one machine instruction. In other words, there is a one-to-one correspondence between machine instructions and statements in the assembly program. If each line in the assembly language program contains exactly one statement and each machine word contains exactly one machine instruction, then an n-line assembly program will produce an n-instruction machine language program. The reason that people use assembly language, as opposed to programming in machine language (in binary or hexadecimal), is that it is much easier to program in assembly language. The use of symbolic names and symbolic addresses instead of binary or hexadecimal ones makes an enormous difference. Most people can remember that the abbreviations for add, subtract, multiply, and divide are ADD, SUB, MUL, and DIV, but few can remember the corresponding numerical values the machine uses. The assembly language programmer need only remember the symbolic names because the assembler translates them to the machine instructions. The same remarks apply to addresses. The assembly language programmer can give symbolic names to memory locations and have the assembler worry about supplying the correct numerical values. The machine language programmer must always work with the numerical values of the addresses. As a consequence, no one programs in machine language today, although people did so decades ago, before assemblers had been invented. Assembly languages have another property, besides the one-to-one mapping of assembly language statements onto machine instructions, that distinguishes them from high-level languages. The assembly programmer has access to all the features and instructions available on the target machine. The high-level language programmer does not. For example, if the target machine has an overflow bit, an assembly language program can test it, but a Java program cannot test it. An assembly language program can execute every instruction in the instruction set of the target machine, but the high-level language program cannot. In short, everything that can be done in machine language can be done in assembly language, but many instructions, registers, and similar features are not available for the high-level language programmer to use. Languages for system programming, like C, are a cross between these types, with the syntax of a high-level language but with some of the access to the machine of an assembly language. One final difference that is worth making explicit is that an assembly language program can run only on one family of machines, whereas a program written

in a high-level language can potentially run on many machines. For many applications, this ability to move software from one machine to another is of great practical importance.

**Why Use Assembly Language?**

Assembly language programming is difficult. Make no mistake about that. It is not for wimps and weaklings. Furthermore, writing a program in assembly language takes much longer than writing the same program in a high-level language. It also takes much longer to debug and is much harder to maintain. Under these conditions, why would anyone ever program in assembly language? There are two reasons: performance and access to the machine. First of all, an expert assembly language programmer working very hard can sometimes produce code that is much smaller and much faster than a high-level language programmer can. For some applications, speed and size are critical. Many embedded applications, such as the code on a smart card or RFID card, device drivers, stringmanipulation libraries, BIOS routines, and the inner loops of performance-critical real-time applications fall in this category. Second, some procedures need complete access to the hardware, something usually impossible in high-level languages. For example, the low-level interrupt and trap handlers in an operating system and the device controllers in many embedded real-time systems fall into this category. Besides these reasons for programming in assembly language, there are also two additional reasons for studying it. First, a compiler must either produce output used by an assembler or perform the assembly process itself. Thus understanding assembly language is essential to understanding how compilers work. Someone has to write the compiler (and its assembler) after all. Second, studying assembly language exposes the real machine to view. For students of computer architecture, writing some assembly code is the only way to get a feel for what a machine is really like at the architectural level.

**Format of an Assembly Language Statement**

Although the structure of an assembly language statement closely mirrors the structure of the machine instruction that it represents, assembly languages for different machines sufficiently resemble one another to allow a discussion of assembly language in general. Figure 7-1 shows fragments of assembly language programs for the x86 which performs the computation N = I + J. The statements below the blank line are commands to the assembler to reserve memory for the variables I, J, and N and are not symbolic representations of machine instructions.

```
Label      Opcode    Operands    Comments
FORMULA:   MOV       EAX,I       ; register EAX = I
           ADD       EAX,J       ; register EAX = I + J
           MOV       N,EAX       ; N = I + J

I          DD        3           ; reserve 4 bytes initialized to 3
J          DD        4           ; reserve 4 bytes initialized to 4
N          DD        0           ; reserve 4 bytes initialized to 0
```

Figure 7-1. Computation of N = I + J on the x86.

Several assemblers exist for the Intel family (i.e., x86), each with a different syntax. In this chapter we will use the Microsoft MASM assembly language for our example. There are many assemblers for the ARM, but the syntax is comparable to the x86 assembler, so one example should siffice. Assembly language statements have up to four parts: first, a label field, second, an operation (opcode) field, third, an operand field, and fourth, a comments field. None of these is mandatory. Labels, which are used to provide symbolic names for memory addresses, are needed on executable statements so that the statements can be branched to. Additionally, they are needed for data words to permit the data stored there to be accessible by symbolic name. If a statement is labeled, the label (usually) begins in column 1. The example of Fig. 7-1 has four labels: FORMULA, I, J, and N. The MASM requires colons on code labels but not on data labels. There is nothing fundamental about this. Other assemblers may have other requirements. Nothing in the underlying architecture suggests one choice or the other. One advantage of the colon notation is that with it a label can appear by itself on a line, with the opcode in column 1 of the next line. This style is convenient for compilers to generate. Without the colon, there would be no way to tell a label on a line all by itself from an opcode on a line all by itself. The colon eliminates this potential ambiguity. It is an unfortunate characteristic of some assemblers that labels are restricted to six or eight characters. In contrast, most high-level languages allow the use of arbitrarily long names. Long, well-chosen names make programs much more readable and understandable by other people. Each machine has some registers, so they need names. The x86 registers have names like EAX, EBX, ECX, and so on. The opcode field contains either a symbolic abbreviation for the opcode—if the statement is a symbolic representation for a machine instruction—or a command to the assembler itself. The choice of an appropriate name is just a matter of taste, and different assembly language designers often make different choices. The designers of the MASM assembler decided to use MOV for both loading a register from memory and storing a register into memory but they could have chosen MOVE or LOAD and STORE. Assembly progams often need to reserve space for variables. The MASM assembly language designers chose DD (Define Double), since a word on the 8088 was 16 bits. The operand field of an assembly language statement is used to specify the addresses and registers used as operands by the machine instruction. The operand field of an integer addition instruction

tells what is to be added to what. The operand field of a branch instruction tells where to branch to. Operands can be registers, constants, memory locations, and so on. The comments field provides a place for programmers to put helpful explanations of how the program works for the benefit of other programmers who may subsequently use or modify the program (or for the benefit of the original programmer a year later). An assembly language program without such documentation is nearly incomprehensible to all programmers, frequently including the author as well. The comments field is solely for human consumption; it has no effect on the assembly process or on the generated program.

**Pseudoinstructions**

In addition to specifying which machine instructions to execute, an assembly language program can also contain commands to the assembler itself, for example, asking it to allocate some storage or to eject to a new page on the listing. Commands to the assembler itself are called pseudoinstructions or sometimes assembler directives. We have already seen a typical pseudoinstruction in Fig. 7-1(a): DD. Some other pseudoinstructions are listed in Fig. 7-2. These are taken from the Microsoft MASM assembler for the x86.

| Pseudoinstruction | Meaning |
|---|---|
| SEGMENT | Start a new segment (text, data, etc.) with certain attributes |
| ENDS | End the current segment |
| ALIGN | Control the alignment of the next instruction or data |
| EQU | Define a new symbol equal to a given expression |
| DB | Allocate storage for one or more (initialized) bytes |
| DW | Allocate storage for one or more (initialized) 16-bit (word) data items |
| DD | Allocate storage for one or more (initialized) 32-bit (double) data items |
| DQ | Allocate storage for one or more (initialized) 64-bit (quad) data items |
| PROC | Start a procedure |
| ENDP | End a procedure |
| MACRO | Start a macro definition |
| ENDM | End a macro definition |
| PUBLIC | Export a name defined in this module |
| EXTERN | Import a name from another module |
| INCLUDE | Fetch and include another file |
| IF | Start conditional assembly based on a given expression |
| ELSE | Start conditional assembly if the IF condition above was false |
| ENDIF | End conditional assembly |
| COMMENT | Define a new start-of-comment character |
| PAGE | Generate a page break in the listing |
| END | Terminate the assembly program |

Figure 7-2. Some of the pseudoinstructions available in the MASM assembler (MASM).

The SEGMENT pseudoinstruction starts a new segment, and ENDS terminates one. It is allowed to start a text segment, with code, then start a data segment, then go back to the text segment, and so on. ALIGN forces the next line, usually data, to an address that is a multiple of its argument. For example, if the current segment has 61 bytes of data already, then after ALIGN 4 the next address allocated will be 64. EQU is used to give a symbolic name to an expression. For example, after the pseudoinstruction BASE EQU 1000 the symbol BASE can be used everywhere instead of 1000. The expression that follows the EQU can involve multiple defined symbols combined with arithmetic and other operators, as in LIMIT EQU 4 * BASE + 2000 Most assemblers, including MASM, require that a symbol be defined before it is used in an expression like this. The next four pseudoinstructions, DB, DW, DD, and DQ, allocate storage for one or more variables of size 1, 2, 4, or 8 bytes, respectively. For example, TABLE DB 11, 23, 49 allocates space for 3 bytes and initializes them to 11, 23, and 49, respectively. It also defines the symbol TABLE and sets it equal to the address where 11 is stored. The PROC and ENDP pseudoinstructions define the beginning and end of assembly language procedures, respectively. Procedures in assembly language have the same function as procedures in other programming languages. Similarly, MACRO and ENDM delimit the scope of a macro definition. We will study macros later in this chapter. The next two pseudoinstructions, PUBLIC and EXTERN, control the visibility of symbols. It is common to write programs as a collection of files. Frequently, a procedure in one file needs to call a procedure or access a data word defined in another file. To make this cross-file referencing possible, a symbol that is to be made available to other files is exported using PUBLIC. Similarly, to prevent the assembler from complaining about the use of a symbol that is not defined in the current file, the symbol can be declared as EXTERN, which tells the assembler that it will be defined in some other file. Symbols that are not declared in either of these pseudoinstructions have a scope of the local file. This default means that using, say, FOO in multiple files does not generate a conflict because each definition is local to its own file. The INCLUDE pseudoinstruction causes the assembler to fetch another file and include it bodily into the current one. Such included files often contain definitions, macros, and other items needed in multiple files. Many assemblers, support conditional assembly. For example, WORDSIZE EQU 32 IF WORDSIZE GT 32 WSIZE: DD 64 ELSE WSIZE: DD 32 ENDIF allocates a single 32-bit word and calls its address WSIZE. The word is initialized to either 64 or 32, depending on the value of WORDSIZE, in this case, 32. Typically this construction would be used to write a program that could be assembled for either 32-bit mode or 64-bit mode. IF and ENDIF, then by changing a single definition, WORDSIZE, the program can automatically be set to assemble for either size. Using this approach, it is possible to maintain one source program for multiple (different) target machines, which makes software development and maintenance easier. In many cases, all the machine-dependent definitions, like WORDSIZE, are collected into a single file, with different versions for different machines. By including the right definitions file, the program can be easily recompiled for different machines. The COMMENT pseudoinstruction allows the user to change the comment delimiter to

something other than semicolon. PAGE is used to control the listing the assembler can produce, if requested. END marks the end of the program. Many other pseudoinstructions exist in MASM. Other x86 assemblers have a different collection of pseudoinstructions available because they are dictated not by the underlying architecture, but by the taste of the assembler writer.