CS1101: Lecture 37

Introduction to Assembly Language

Dr. Barry O'Sullivan
b.osullivan@cs.ucc.ie

Course Homepage
http://www.cs.ucc.ie/~osullb/cs1101

Lecture Outline

- Introduction
- What is Translation?
- Types of Translator
- What is an Assembly Language?
  - Assembly versus Machine Language
  - Why Use Assembly Language?
  - Performance & Machine Access
- Format of Assembly Language Statements
  - Register Lengths
  - Data Words
  - Operands Field
  - The Comments Field
- Pseudoinstructions
- **Reading**: Tanenbaum, Chapter 7, Section 1.

Introduction

- The assembly language level differs in a significant respect from the microarchitecture, ISA, and operating system machine levels – it is implemented by **translation** rather than by **interpretation**.
- Programs that convert a user's program written in some language to another language are called translators.
- The language in which the original program is written is called the **source language**
- The language to which it is converted is called the **target language**.

What is Translation?

- In translation, the original program in the source language is not directly executed.
- Instead, it is converted to an equivalent program called an **object program** or **executable binary program** whose execution is carried out only after the translation has been completed.
- In translation, there are two distinct steps: Generation of an equivalent program in the target language. Execution of the newly generated program.
- In translation, these two steps do not occur simultaneously.
- The second step does not begin until the first has been completed.
- In interpretation, there is only one step: executing the original source program.

Types of Translator

- Translators can be roughly divided into two groups, depending on the relation 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.
- 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-word machine language program.

Assembly versus Machine Language

- Assembly language is easier to use than machine language (hexadecimal)
- The use of symbolic names and symbolic addresses instead of binary or octal 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.

Assembly versus Machine Language

- 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 versus Machine Language

- The assembly programmer has access to all the features and instructions available on the target machine.
- The high-level language programmer does not.
- 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.
- One final difference that is worth making explicit is that an assembly language program can only run 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.
- 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.
- However, there are two reasons for using assembly language: performance and access to the machine

Performance & Machine Access

- Performance
  - An expert assembly language programmer can often produce code that is much smaller and much faster than a high-level language programmer can.
  - For some applications, speed and size are critical.
  - For example, smart cards, embedded applications, device drivers etc.
- Access to the machine:
  - 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.

Format of Assembly Language Statements

- See Figure 7-2
- Assembly language statements have four parts:
  - a label field,
  - an operation (opcode) field,
  - an operands field,
  - a comments field.
- Labels, which are used to provide symbolic names for memory addresses, are needed on executable statements so that the statements can be branched to.
- They are also 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.

Format of Assembly Language Statements

- Each of the three parts of Fig. 7-2 has four labels: FORMULA, I, J, and N.
- Note that sometimes colons are used and sometimes not.
- Each of the machines has some **registers**, but they have been given very different names.
- The Pentium II registers have names like EAX, EBX, ECX, and so on.
- The Motorola registers are called DO, D1, D2, among others.
- The SPARC registers have multiple names – here we have used

Format of Assembly Language Statements

- 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 Pentium family, 680x0, and SPARC all allow byte, word, and long operands.
- How does the assembler know which length to use?

Register Lengths

- On the Pentium II, different length registers have different names, so EAX is used to move 32-bit items, AX is 16-bit items, and AL and AH are used to move 8-bit items.
- The Motorola assembler uses a suffix .L for long, .W for word, or .B for byte to each opcode rather than giving subsets of DO, etc., different names.
- The SPARC uses different opcodes for the different lengths (e.g., LDSB, LDSH, and LDSW to load signed bytes, halfwords, and words into a 64-bit register, respectively).

Data Words

- The three assemblers also differ in how they reserve space for data.
- The Intel assembly language designers chose DW (Define Word), although WORD was added as an alternative later.
- The Motorola ones liked DC (Define Constant).
- The SPARC folks preferred WORD from the beginning.

Operands Field

- The operands field of an assembly language statement is used to specify the addresses and registers used as operands by the machine instruction.
- The operands field of an integer addition instruction tells what is to be added to what.
- The operands field of a branch instruction tells where to branch to.
- Operands can be registers, constants, memory locations, and so on.

The Comments Field

- The comments field provides a place for programmers to put helpful explanations of how the program works for the benefit of other programmers.
- An assembly language program without such documentation is nearly incomprehensible to all programmers.
- 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: DW.
- Some other from the Microsoft MASM assembler for the Intel family are shown on the next slide.

Examples: Pseudoinstructions

- 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.
- 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.

Examples: Pseudoinstructions

- See Figure 7-3
- 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.