Chapter 7

Low Level Programming Languages
Layers of a Computing System

Communication
Application
Operating System
Programming
Hardware
Information
Chapter Goals

- List the operations that a computer can perform
- Describe the important features of a virtual computer
- Distinguish between immediate mode addressing and direct addressing
- Convert a simple algorithm into a machine-language program
- Distinguish between machine language and assembly language
- Describe the steps in creating and running an assembly-language program
Computer Operations

• A computer is a programmable electronic device that can store, retrieve, and process data.

• Data and instructions to manipulate the data are logically the same and can be stored in the same place.

• **Store, retrieve**, and **process** are actions that the computer can perform on data.
Machine Language

- **Machine language**  The instructions built into the **hardware** of a particular computer

- Initially, humans had no choice but to write programs in machine language because other programming languages had not yet been invented
Machine Language

- Every processor type has its own set of specific machine instructions
- The relationship between the processor and the instructions it can carry out is completely integrated
- Each machine-language instruction does only one very low-level task
Pep/7: A Virtual Computer

- **Virtual computer** A hypothetical machine designed to contain the important features of real computers that we want to illustrate

- Pep/7
  - designed by Stanley Warford
  - has 32 machine-language instructions

- We are only going to examine a few of these instructions
Features in Pep/7

- The memory unit is made up of 4,096 bytes
- Pep/7 Registers/Status Bits Covered
  - The program counter (PC) (contains the address of the next instruction to be executed)
  - The instruction register (IR) (contains a copy of the instruction being executed)
  - The accumulator (A register)
    - Status bit N (1 if A register is negative; 0 otherwise)
    - Status bit Z (1 if the A register is 0; and 0 otherwise)
Features in Pep/7

**Pep/7's CPU (as discussed in this chapter)**

<table>
<thead>
<tr>
<th>Status bits</th>
<th>N</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>A register (accumulator)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program counter (CP)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction Register (IR)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Pep/7's Memory**

<table>
<thead>
<tr>
<th>Address</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td></td>
</tr>
<tr>
<td>0002</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0FFE</td>
<td></td>
</tr>
<tr>
<td>0FFF</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 7.1 Pep/7’s architecture*
Instruction Format

- There are two parts to an instruction
  - The 8-bit instruction specifier
  - And optionally, the 16-bit operand specifier
Instruction Format

- The instruction specifier is made up of several sections
  - The operation code
  - The register specifier
  - The addressing-mode specifier
Instruction Format

• The *operation code* specifies which instruction is to be carried out.

• The 1-bit *register specifier* is 0 if register A (the accumulator) is involved, which is the case in this chapter.

• The 2-bit *addressing-mode specifier* says how to interpret the operand part of the instruction.
Instruction Format

(a) Immediate-mode addressing: Operand is shaded gray

(b) Direct-mode addressing: Operand is shaded gray

Figure 7.3 Difference between immediate-mode and direct-mode addressing
Some Sample Instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Meaning of Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>Stop execution</td>
</tr>
<tr>
<td>00001</td>
<td>Load operand into a register (either A of X)</td>
</tr>
<tr>
<td>00010</td>
<td>Store the contents of register (either A or X) into operand</td>
</tr>
<tr>
<td>00011</td>
<td>Add the operand to register (either A or X)</td>
</tr>
<tr>
<td>00100</td>
<td>Subtract the operand from register (either A or X)</td>
</tr>
<tr>
<td>11011</td>
<td>Character input to operand</td>
</tr>
<tr>
<td>11100</td>
<td>Character output from operand</td>
</tr>
</tbody>
</table>

*Figure 7.3  Subset of Pep/7 instructions*
### A Program Example

Let’s write "Hello" on the screen

<table>
<thead>
<tr>
<th>Module</th>
<th>Binary Instruction</th>
<th>Hex Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write &quot;H&quot;</td>
<td>11100000 0000000000100100</td>
<td>E0 0048</td>
</tr>
<tr>
<td>Write &quot;e&quot;</td>
<td>11100000 0000000001100101</td>
<td>E0 0065</td>
</tr>
<tr>
<td>Write &quot;l&quot;</td>
<td>11100000 0000000001101100</td>
<td>E0 006C</td>
</tr>
<tr>
<td>Write &quot;l&quot;</td>
<td>11100000 0000000001101100</td>
<td>E0 006C</td>
</tr>
<tr>
<td>Write &quot;o&quot;</td>
<td>11100000 0000000001101111</td>
<td>E0 006F</td>
</tr>
<tr>
<td>Stop</td>
<td>00000000</td>
<td>00</td>
</tr>
</tbody>
</table>
Pep/7 Simulator

- A program that behaves just like the Pep/7 virtual machine behaves
- To run a program, we enter the hexadecimal code, byte by byte with blanks between each
Assembly Language

• **Assembly languages** A language that uses mnemonic codes to represent machine-language instructions
  • The programmer uses these alphanumerical codes in place of binary digits
  • A program called an assembler reads each of the instructions in mnemonic form and translates it into the machine-language equivalent
## Pep/7 Assembly Language

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operand, Mode Specifier</th>
<th>Meaning of Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>STOP</td>
<td></td>
<td>Stop execution</td>
</tr>
<tr>
<td>LOADA</td>
<td>h#008B,i</td>
<td>Load 008B into register A</td>
</tr>
<tr>
<td>LOADA</td>
<td>h#008B,d</td>
<td>Load the contents of location 8B into register A</td>
</tr>
<tr>
<td>STOREA</td>
<td>h#008B,d</td>
<td>Store the contents of register A into location 8B</td>
</tr>
<tr>
<td>ADDA</td>
<td>h#008B,i</td>
<td>Add 008B to register A</td>
</tr>
<tr>
<td>ADDA</td>
<td>h#008B,d</td>
<td>Add the contents of location 8B to register A</td>
</tr>
<tr>
<td>SUBA</td>
<td>h#008B,i</td>
<td>Subtract 008B from register A</td>
</tr>
<tr>
<td>SUBA</td>
<td>h#008B,d</td>
<td>Subtract the contents of location 8B from register A</td>
</tr>
<tr>
<td>CHARI</td>
<td>h#008B,d</td>
<td>Read a character and store it into byte 8B</td>
</tr>
<tr>
<td>CHAR0</td>
<td>c#/B/,i</td>
<td>Write the character B</td>
</tr>
<tr>
<td></td>
<td>h#008B,d</td>
<td>Write the character stored in byte 8B</td>
</tr>
<tr>
<td>DECI</td>
<td>h#008B,d</td>
<td>Read a decimal number and store it into location 8B</td>
</tr>
<tr>
<td>DECO</td>
<td>h#008B,i</td>
<td>Write the decimal number 139 (8B in hex)</td>
</tr>
<tr>
<td>DECO</td>
<td>h#008B,d</td>
<td>Write the decimal number stored in 8B</td>
</tr>
</tbody>
</table>
Assembly Process

Program in Assembly Language → Input → Assembler → Output → Program in Machine Code
A Simple Program

Set sum to 0
Read num1
Add num1 to sum
Read num2
Add num2 to sum
Read num3
Add num3 to sum
Write sum
Our Completed Program

```
BR Main ; branch to location Main
sum: .WORD d#0 ; set up word with zero as the contents
num1: .BLOCK d#2 ; set up a two byte block for num1
num2: .BLOCK d#2 ; set up a two byte block for num2
num3: .BLOCK d#2 ; set up a two byte block for num3
Main: LOADA sum,d ; load a copy of sum into accumulator
DECI num1,d ; read and store a decimal number in num1
ADDA num1,d ; add the contents of num1 to accumulator
DECI num2,d ; read and store a decimal number in num2
ADDA num2,d ; add the contents of num2 to accumulator
DECI num3,d ; read and store a decimal number in num3
ADDA num3,d ; add the contents of num3 to accumulator
STOREA sum,d ; store contents of the accumulator into sum
DECO sum,d ; output the contents of sum
STOP ; stop the processing
.END ; end of the program
```
**Status Bits**

Status bits allow a program to make a choice.

- **BRLT** Set the PC to the operand, if N is 1
  (A register is *less than* zero)

- **BREQ** Set the PC to the operand, if Z is 1
  (A register is *equal to* zero)
Testing

- **Test plan**  A document that specifies how many times and with what data the program must be run in order to thoroughly test the program.

- A **code-coverage approach** designs test cases to ensure that each statement in the program is executed.

- A **data-coverage approach** designs test cases to ensure that the limits of the allowable data are covered.
Chapter 8

High Level Programming Languages
Chapter Goals

- Describe the translation process and distinguish between assembly, compilation, interpretation, and execution.
- Name four distinct programming paradigms and name a language characteristic of each.
- Describe the following constructs: stream input and output, selection, looping, and subprograms.
- Construct Boolean expressions and describe how they are used to alter the flow of control of an algorithm.
- . . . Some Hands-On
• **Compiler** A program that translates a high-level language program into machine code

• High-level languages provide a richer set of instructions that makes the programmer’s life even easier
Compilers

Figure 8.1 Compilation process
Interpreters

• **Interpreter** A translating program that translates and executes the statements in sequence

  • Unlike an assembler or compiler which produce machine code as output, which is then executed in a separate step

  • An interpreter translates a statement and then immediately executes the statement

  • Interpreters can be viewed as *simulators*
Java

• Introduced in 1996 and swept the computing community by storm

• **Portability** was of primary importance

• Java is compiled into a standard machine language called **Bytecode**

• A software interpreter called the **JVM (Java Virtual Machine)** takes the Bytecode program and executes it
What is a paradigm?

A set of assumptions, concepts, values, and practices that constitute a way of viewing reality
Programming Language Paradigms

Figure 8.2
Portability provided by standardized languages versus interpretation by Bytecode
Programming Language Paradigms

(b) Java program compiled into Bytecode and run on different systems

Figure 8.2
Portability provided by standardized languages versus interpretation by Bytecode
Programming Language Paradigms

- **Imperative or procedural model**
  - FORTRAN, COBOL, BASIC, C, Pascal, Ada, and C++

- **Functional model**
  - LISP, Scheme (a derivative of LISP), and ML
Programming Language Paradigms

- Logic programming
  - PROLOG

- Object-oriented paradigm
  - SIMULA and Smalltalk
  - C++ is as an imperative language with some object-oriented features
  - Java is an object-oriented language with some imperative features
Functionality of Imperative Languages

- **Sequence** Executing statements in sequence until an instruction is encountered that changes this sequencing
- **Selection** Deciding which action to take
- **Iteration** (looping) Repeating an action

Both selection and iteration require the use of a Boolean expression
Boolean Expressions

- **Boolean expression**  A sequence of identifiers, separated by compatible operators, that evaluates to *true* or *false*

- Boolean expression can be
  - A **Boolean variable**
  - An arithmetic expression followed by a **relational operator** followed by an arithmetic expression
  - A Boolean expression followed by a **Boolean operator** followed by a Boolean expression
Boolean Expressions

• **Variable**  A location in memory that is referenced by an identifier that contains a data value

  Thus, a Boolean variable is a location in memory that can contain either *true* or *false*
Boolean Expressions

• A relational operator between two arithmetic expressions is asking if the relationship exists between the two expressions.

• For example, $xValue < yValue$

<table>
<thead>
<tr>
<th>Relationship</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>equal to</td>
<td>= or ==</td>
</tr>
<tr>
<td>not equal to</td>
<td>&lt;&gt; or != or /=</td>
</tr>
<tr>
<td>less than or equal to</td>
<td>&lt;=</td>
</tr>
<tr>
<td>greater than or equal to</td>
<td>&gt;=</td>
</tr>
<tr>
<td>less than</td>
<td>&lt;</td>
</tr>
<tr>
<td>greater than</td>
<td>&gt;</td>
</tr>
</tbody>
</table>
Strong Typing

- **Strong typing** The requirement that only a value of the proper type can be stored into a variable

- **Data type** A description of the set of values and the basic set of operations that can be applied to values of the type
Data Types

- **Integer** numbers
- **Real** numbers
- **Characters**
- **Boolean** values
- **Strings**
Integers

- The **range varies** depending upon how many bytes are assigned to represent an integer value.
- Some high-level languages provide several integer types of different sizes.
- Operations that can be applied to integers are the standard arithmetic and relational operations.
Reals

- Like the integer data type, the range varies depending on the number of bytes assigned to represent a real number.
- Many high-level languages have two sizes of real numbers.
- The operations that can be applied to real numbers are the same as those that can be applied to integer numbers.
Characters

- It takes **one byte** to represent characters in the **ASCII character set**
- **Two bytes** to represent characters in the **Unicode character set**
- Our English alphabet is represented in ASCII, which is a subset of Unicode
Characters

• Applying arithmetic operations to characters doesn’t make much sense

• Comparing characters does make sense, so the relational operators can be applied to characters

• The meaning of “less than” and “greater than” when applied to characters is “comes before” and “comes after” in the character set
Boolean

• The **Boolean data type** consists of two values: *true* and *false*

• Not all high-level languages support the Boolean data type

• If a language does not, then you can simulate Boolean values by saying that the Boolean value *true* is represented by 1 and *false* is represented by 0
Strings

- **A string is a sequence of characters** considered as one data value

- For example: **“This is a string.”**
  - Containing 17 characters: one uppercase letter, 12 lowercase letters, three blanks, and a period

- The operations defined on strings vary from language to language
  - They include concatenation of strings and comparison of strings in terms of lexicographic order
Declarations

• **Declaration** A statement that associates an identifier with a variable, an action, or some other entity within the language that can be given a name so that the programmer can refer to that item by name
## Declarations

<table>
<thead>
<tr>
<th>Language</th>
<th>Variable Declaration</th>
</tr>
</thead>
</table>
| Ada      | `sum : Float := 0; -- set up word with 0 as contents`  
           | `num1: Integer; -- set up a two-byte block for num1`  
           | `num2: Integer; -- set up a two-byte block for num2`  
           | `num3: INTEGER; -- set up a two-byte block for num3`  
           | `...`  
           | `num1 := 1;` |
| VB.NET   | `Dim sum As Single = 0.0F ' set up word with 0 as contents`  
           | `Dim num1 As Integer ' set up a two-byte block for num1`  
           | `Dim num2 As Integer ' set up a two-byte block for num2`  
           | `Dim num3 As Integer ' set up a two-byte block for num3`  
           | `...`  
           | `num1 = 1` |
| C++/Java | `float sum = 0.0; // set up word with 0 as contents`  
           | `int num1; // set up a block for num1`  
           | `int num2; // set up a block for num2`  
           | `int num3; // set up a block for num3`  
           | `...`  
           | `num1 = 1;` |
Declarations

- **Reserved word** A word in a language that has special meaning

- **Case-sensitive** Uppercase and lowercase letters are considered the same
Assignment statement

- **Assignment statement** An action statement (not a declaration) that says to evaluate the expression on the right-hand side of the symbol and store that value into the place named on the left-hand side.

- **Named constant** A location in memory, referenced by an identifier, that contains a data value that cannot be changed.
## Assignment Statement

<table>
<thead>
<tr>
<th>Language</th>
<th>Constant Declaration</th>
</tr>
</thead>
</table>
| Ada      | Comma : constant Character := ',',;  
            | Message : constant String := "Hello";  
            | Tax_Rate : constant Float := 8.5; |
| VB.NET   | Const WORD1 As Char = ","c  
            | Const MESSAGE As String = "Hello"  
            | Const TaxRate As Double = 8.5 |
| C++      | const char COMMA = ",'';  
            | const string MESSAGE = "Hello";  
            | const double TAX_RATE = 8.5; |
| Java     | final char COMMA = ",'';  
            | final String MESSAGE = "Hello";  
            | final double TAX_RATE = 8.5; |
Homework

• Read Chapter Seven, Concentrate on Slides
• Read Chapter Eight, Secs 8.1 & 8.2
Mid-Term

- **Take Home** Exam – Non-Trivial (think!)
- **Cover** Chapters 1-5 & 16 & Anything Covered In Class
- **Given Out:** Oct 11
- **Due Back:** Oct 18
- **No Lateness!!!**
- **You can email, if you like :)**
Have A Nice Weekend!