Chapter Goals

• List the operations that a computer can perform

• Discuss the relationship between levels of abstraction and the determination of concrete algorithm steps

• Describe the important features of the Pep/7 virtual machine

• Distinguish between immediate mode addressing and direct addressing
Chapter Goals

• 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
• Convert a simple algorithm into an assembly-language program
Chapter Goals

• Distinguish between instructions to the assembler and instructions to be translated
• Describe two approaches to testing
• Design and implement a test plan for a simple assembly-language program
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 Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>A register (accumulator)</td>
<td></td>
</tr>
<tr>
<td>Program counter (CP)</td>
<td></td>
</tr>
<tr>
<td>Instruction Register (IR)</td>
<td></td>
</tr>
</tbody>
</table>

Pep/7's Memory

0000
0001
0002
...
0FFE
0FFF

Figure 7.1 Pep/7’s architecture
There are two parts to an instruction

- The 8-bit instruction specifier
- And optionally, the 16-bit operand specifier

(a) The two parts of an instruction

(b) The instruction specifier part of an instruction
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

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 or 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 0000000001001000</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 languages**  A language that uses mnemonic codes to represent machine-language instructions
  
  – The programmer uses these alphanumeric 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>
Figure 7.5  Assembly Process
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 num2 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 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.