Translators can be divided into two groups.

- 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, the translator is called a **compiler**.

A pure assembly language is a language in which each statement produces exactly one machine instruction.
The Assembly Language Level

- The use of symbolic names and symbolic addresses (rather than binary or hexadecimal ones) makes it easier to program in assembly language than in machine language.
- The assembly programmer has access to all the features and instructions on the target machine.
  - The high-level language programmer does not.
  - Languages for system programming, such as C, provide much of the access to the machine of an assembly language.
- Assembly programs are not portable.
Why Use Assembly Language?

- There are several reasons to program in assembly, rather than a high-level language:
  1. An expert assembly language programmer can often produce code that is much smaller and much faster than a high-level language programmer can.
  2. Some procedures need complete access to the hardware, something usually impossible in high-level languages.
  3. A compiler must either produce output used by an assembler or perform the assembly process itself - and someone has to program the compiler.
  4. Studying assembly language exposes the real machine to view.
Why Use Assembly Language?

<table>
<thead>
<tr>
<th></th>
<th>Programmer-years to produce the program</th>
<th>Program execution time in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly language</td>
<td>50</td>
<td>33</td>
</tr>
<tr>
<td>High-level language</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>Mixed approach before tuning</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Critical 10%</td>
<td>1</td>
<td>90</td>
</tr>
<tr>
<td>Other 90%</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>Total</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>Mixed approach after tuning</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Critical 10%</td>
<td>6</td>
<td>30</td>
</tr>
<tr>
<td>Other 90%</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>Total</td>
<td>15</td>
<td>40</td>
</tr>
</tbody>
</table>

Comparison of assembly language and high-level language programming, with and without tuning.
Assembly language statements have four parts:

- a label field
- an operation (opcode) field
- an operands field
- a comments field

Labels are used for branches and to give a symbolic name to some memory address.

- Some assemblers restrict labels to six or eight characters.
Assembly Language Statements

Computation of $N = I + J$. (a) Pentium 4.

<table>
<thead>
<tr>
<th>Label</th>
<th>Opcode</th>
<th>Operands</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORMULA:</td>
<td>MOV</td>
<td>EAX, I</td>
<td>; register EAX = I</td>
</tr>
<tr>
<td></td>
<td>ADD</td>
<td>EAX, J</td>
<td>; register EAX = I + J</td>
</tr>
<tr>
<td></td>
<td>MOV</td>
<td>N, EAX</td>
<td>; $N = I + J$</td>
</tr>
<tr>
<td>I</td>
<td>DD</td>
<td>3</td>
<td>; reserve 4 bytes initialized to 3</td>
</tr>
<tr>
<td>J</td>
<td>DD</td>
<td>4</td>
<td>; reserve 4 bytes initialized to 4</td>
</tr>
<tr>
<td>N</td>
<td>DD</td>
<td>0</td>
<td>; reserve 4 bytes initialized to 0</td>
</tr>
</tbody>
</table>

(a)
In addition to specifying which machine instructions to execute, an assembly language program can also contain commands to the assembler itself.

- For example, allocate some storage, or eject to a new page in the listing.
- Commands to the assembler itself are called **pseudoinstructions** or **assembler directives**.
- Some typical pseudoinstructions are shown on the following slide. These are from the Microsoft MASM assembler for the Intel family.
### Assembler Pseudoinstructions

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

Some of the pseudoinstructions available in the Pentium 4 assembler (MASM).
Macros

- Assembly language programmers frequently need to repeat sequences of instructions several times within a program.
  - One way is to make the sequence a procedure and call it several times.
    - This requires a procedure call instruction and return instruction every time.
    - This could significantly slow down the program if the sequence is short but repeated frequently.
  - Macros provide an easy and efficient solution.
  - A macro definition is a way to give a name to a piece of text.
• After a macro has been defined, the programmer can write the macro name rather than the piece of program.
• The following slide shows an assembly language program for the Pentium II that exchanges the contents of the variables p and q twice.
• These sequences could be defined as macros.

- **Macro definitions generally require the following parts:**
  • A macro header giving the name of the macro.
  • The text comprising the body of the macro
  • A pseudoinstruction marking the end of the macro
Macros

Assembly language code for interchanging P and Q twice.
(a) Without a macro.  (b) With a macro.
When the assembler encounters a macro definition, it saves it in a table for subsequent use.

- From that point on, whenever the name of the macro appears as an opcode, the assembler replaces it by the macro body.
- The use of a macro name as an opcode is known as a **macro call**.
- Its replacement by the macro body is known as a **macro expansion**.
  - Macro expansion occurs during the assembly process, not during execution of the program.
## Macros

Comparison of macro calls with procedure calls.

<table>
<thead>
<tr>
<th>Item</th>
<th>Macro call</th>
<th>Procedure call</th>
</tr>
</thead>
<tbody>
<tr>
<td>When is the call made?</td>
<td>During assembly</td>
<td>During program execution</td>
</tr>
<tr>
<td>Is the body inserted into the object program every place the call is made?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Is a procedure call instruction inserted into the object program and later executed?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Must a return instruction be used after the call is done?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>How many copies of the body appear in the object program?</td>
<td>One per macro call</td>
<td>One</td>
</tr>
</tbody>
</table>
The Assembly Process

- The assembler cannot directly read a one-line statement and convert it into machine language.
  - The difficulty is caused by the **forward reference problem** where a symbol $L$ has been used before it is declared (i.e. in a branch statement).

- We can deal with this problem in two ways.
  - The assembler may in fact read the source program twice.
    - Each reading of the source is called a **pass**.
    - This kind of translator is called a **two-pass translator**.
  - On pass one the definitions of symbols including labels are collected and stored in a table.
The Assembly Process

• By the time the second pass begins, the values of all symbols are known, thus there are no forward references.

▲ The second approach consists of reading the assembly program once, converting it to an intermediate form, and storing it in a table.
  • Then a second pass is made over the table instead of over the source program.
  • If the table fits in main memory, this approach saves I/O.

▲ Defining the symbols and expanding the macros are generally combined into one pass.
The principal function of the first pass is to build up a table called the symbol table, containing the values of all symbols.

- A symbol is either a label or a value that is assigned a symbolic name by means of a pseudoinstruction.
- In assigning a value to a symbol in the label field of an instruction, the assembler must know what address that instruction will have during program execution.
- To keep track of the execution-time address of the instruction being assembled, the assembler maintains a variable known as the ILC (Instruction Location Counter).
### The Assembly Process

<table>
<thead>
<tr>
<th>Label</th>
<th>Opcode</th>
<th>Operands</th>
<th>Comments</th>
<th>Length</th>
<th>ILC</th>
</tr>
</thead>
<tbody>
<tr>
<td>MARIA:</td>
<td>MOV</td>
<td>EAX, I</td>
<td>EAX = I</td>
<td>5</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>MOV</td>
<td>EBX, J</td>
<td>EBX = J</td>
<td>6</td>
<td>105</td>
</tr>
<tr>
<td>ROBERTA:</td>
<td>MOV</td>
<td>ECX, K</td>
<td>ECX = K</td>
<td>6</td>
<td>111</td>
</tr>
<tr>
<td></td>
<td>IMUL</td>
<td>EAX, EAX</td>
<td>EAX = I * I</td>
<td>2</td>
<td>117</td>
</tr>
<tr>
<td></td>
<td>IMUL</td>
<td>EBX, EBX</td>
<td>EBX = J * J</td>
<td>3</td>
<td>119</td>
</tr>
<tr>
<td></td>
<td>IMUL</td>
<td>ECX, ECX</td>
<td>ECX = K * K</td>
<td>3</td>
<td>122</td>
</tr>
<tr>
<td>MARILYN:</td>
<td>ADD</td>
<td>EAX, EBX</td>
<td>EAX = I * I + J * J</td>
<td>2</td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>ADD</td>
<td>EAX, ECX</td>
<td>EAX = I * I + J * J + K * K</td>
<td>2</td>
<td>127</td>
</tr>
<tr>
<td>STEPHANY:</td>
<td>JMP</td>
<td>DONE</td>
<td>branch to DONE</td>
<td>5</td>
<td>129</td>
</tr>
</tbody>
</table>

The instruction location counter (ILC) keeps track of the address where the instructions will be loaded in memory. In this example, the statements prior to MARIA occupy 100 bytes.
**The Assembly Process**

- Pass one of most assemblers uses at least three tables:
  - the symbol table
  - the pseudoinstruction table
  - the opcode table
  - if needed, a literal table is also kept.

- Each symbol table entry contains the symbol itself, its numerical value, and other information:
  - length of the data field associated with the symbol
  - relocation bits
  - whether or not the symbol is accessible outside the procedure
The Assembly Process

A symbol table for the program of Fig. 7-7.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
<th>Other information</th>
</tr>
</thead>
<tbody>
<tr>
<td>MARIA</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>ROBERTA</td>
<td>111</td>
<td></td>
</tr>
<tr>
<td>MARILYN</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>STEPHANY</td>
<td>129</td>
<td></td>
</tr>
</tbody>
</table>
### The Assembly Process

A few excerpts from the opcode table for a Pentium 4 assembler.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>First operand</th>
<th>Second operand</th>
<th>Hexadecimal opcode</th>
<th>Instruction length</th>
<th>Instruction class</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAA</td>
<td>—</td>
<td>—</td>
<td>37</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>ADD</td>
<td>EAX</td>
<td>immed32</td>
<td>05</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>ADD</td>
<td>reg</td>
<td>reg</td>
<td>01</td>
<td>2</td>
<td>19</td>
</tr>
<tr>
<td>AND</td>
<td>EAX</td>
<td>immed32</td>
<td>25</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>AND</td>
<td>reg</td>
<td>reg</td>
<td>21</td>
<td>2</td>
<td>19</td>
</tr>
</tbody>
</table>
The Assembly Process

- Some assemblers allow programmers to write instructions using immediate addressing even though no corresponding target language instruction exists.
  - The assembler allocates memory for the immediate operand at the end of the program and generates an instruction that references it.
  - Constants for which the assembler automatically reserves memory are called **literals**.
    - Literals improve the readability of a program.
  - Immediate instructions are common today, but previously they were unusual.
Pass One (1)

Pass one of a simple assembler.

```java
public static void pass_one() {
    // This procedure is an outline of pass one of a simple assembler.
    boolean more_input = true;  // flag that stops pass one
    String line, symbol, literal, opcode;
    int location_counter, length, value, type;  // fields of the instruction
    final int END_STATEMENT = -2;  // misc. variables
    // signals end of input

    location_counter = 0;
    initialize_tables();

    while (more_input) {
        line = read_next_line();  // assemble first instruction at 0
        length = 0;  // general initialization
        type = 0;

        // more_input set to false by END
        // get a line of input
        // # bytes in the instruction
        // which type (format) is the instruction
    }
}
```
if (line_is_not_comment(line)) {
    symbol = check_for_symbol(line);  // is this line labeled?
    if (symbol != null)                // if it is, record symbol and value
        enter_new_symbol(symbol, location_counter);
    literal = check_for_literal(line);  // does line contain a literal?
    if (literal != null)               // if it does, enter it in table
        enter_new_literal(literal);

    // Now determine the opcode type. -1 means illegal opcode.
    opcode = extract_opcode(line);    // locate opcode mnemonic
    type = search_opcode_table(opcode); // find format, e.g. OP REG1,REG2
    if (type < 0)                      // if not an opcode, is it a pseudoinstruction?
        type = search_pseudo_table(opcode);
    switch(type) {
        // determine the length of this instruction
        case 1: length = get_length_of_type1(line); break;
        case 2: length = get_length_of_type2(line); break;
        // other cases here
    }
}
Pass one of a simple assembler.

```c
write_temp_file(type, opcode, length, line);  // useful info for pass two
location_counter = location_counter + length;  // update loc_ctr
if (type == END_STATEMENT) {
    more_input = false;  // are we done with input?
    rewind_temp_for_pass_two();  // if so, perform housekeeping tasks
    sort_literal_table();  // like rewinding the temp file
    remove_redundant_literals();  // and sorting the literal table
    remove_redundant_literals();  // and removing duplicates from it
}
}
```
public static void pass_two() {
    // This procedure is an outline of pass two of a simple assembler.
    boolean more_input = true; // flag that stops pass two
    String line, opcode;
    int location_counter, length, type; // misc. variables
    final int END_STATEMENT = -2; // signals end of input
    final int MAX_CODE = 16; // max bytes of code per instruction
    byte code[] = new byte[MAX_CODE]; // holds generated code per instruction

    location_counter = 0; // assemble first instruction at 0
    while (more_input) { // more_input set to false by END
        type = read_type(); // get type field of next line
        opcode = read_opcode(); // get opcode field of next line
        length = read_length(); // get length field of next line
        line = read_line(); // get the actual line of input

        if (type != 0) { // type 0 is for comment lines
            switch(type) { // generate the output code
                case 1: eval_type1(opcode, length, line, code); break;
                case 2: eval_type2(opcode, length, line, code); break;
                // other cases here
            }
        }
    }

    write_output(code); // write the binary code
    write_listing(code, line); // print one line on the listing
    location_counter = location_counter + length; // update loc_ctr
    if (type == END_STATEMENT) { // are we done with input?
        more_input = false; // if so, perform housekeeping tasks
        finish_up(); // odds and ends
    }
}
The Symbol Table

- During pass one of the assembler, the symbol table is built up.
- Several different approaches to organizing the symbol table exist.
  - All of them attempt to simulate an associative memory, which conceptually is a set of (symbol, value) pairs.
  - The simplest approach just does a linear search of an array of pairs.
    - On average, half of the symbol table must be searched.
  - Another approach does a binary search of the symbol table.
Hash Coding

- Another approach to simulating associative memory is a technique called **hash coding**.
  - A hash function is chosen which maps symbols onto integers in the range 0 to $k - 1$.
  - For example, multiply the ASCII codes of the characters of the symbol together and take the result modulo $k$.
  - Symbols can be stored in a table consisting of $k$ **buckets** numbered 0 to $k - 1$.
  - Symbols whose hash functions are equal are stored on a linked list pointed to by a slot in the hash table.
The Symbol Table: Hash coding

(a) Symbols, values, and the hash codes derived from the symbols.
(b) Eight-entry hash table with linked lists of symbols and values.
Generation of an executable binary program from a collection of independently translated source procedures requires using a linker.
Linking and Loading - Tasks Performed by the Linker

Each module has its own address space, starting at 0.

Object module A

- 400: CALL B
- 300: MOVE P TO X
- 100: BRANCH TO 200

Object module B

- 600: CALL C
- 500: MOVE Q TO X
- 00: BRANCH TO 300

Object module C

- 500: CALL D
- 400: MOVE R TO X
- 200: BRANCH TO 200

Object module D

- 300: MOVE S TO X
- 200: BRANCH TO 200
Linking and Loading

The object modules before relocation and linking

The same object modules after linking and relocation.

Together they form an executable binary program, ready to run
The internal structure of an object module produced by a translator.
The relocated binary program of Fig. 7-15(b) moved up 300 addresses. Many instructions now refer to an incorrect memory address.
Before EARTH is called.
After EARTH has been called and linked.
Dynamic Linking in Windows

Use of a DLL file by two processes.