Computer Organization & Assembly Language Programming

CSE 2312
Lecture 20 Assembly Process

Junzhou Huang, Ph.D.
Department of Computer Science and Engineering
Intuitive Assembler Solution

- **Intuitive solution**
  - An assembly language program consists of a series of one-line statements.
  - The intuitive solution is to have an assembler that read one statement, then translated it to machine language, and finally output the generated machine language onto a file, along with the corresponding piece of the listing, if any, onto another file.
  - This process would then be repeated until the whole program had been translated.

- **Does not work**
  - Consider the situation where the first statement is a branch to L.
  - The assembler cannot assemble this statement until it knows the address of statement L.
  - Statement L may be near the end of the program, making it impossible for the assembler to find the address without first reading almost the entire program.
  - This difficulty is called the *forward reference problem*, because a symbol, L, has been used before it has been defined; that is, a reference has been made to a symbol whose definition will only occur later.
How to Handle Forward Reference

- **First Way**
  - The assembler may read the source program twice.
  - Each reading of the source program is called a pass; Any translator that reads the input program twice is called a two-pass translator.
  - On pass one, the definitions of symbols, including statement labels, are collected and stored in a table.
  - By the time the second pass begins, the values of all symbols are known; thus no forward reference remains and each statement can be read, assembled, and output.
  - Although this approach requires an extra pass, it is conceptually simple.

- **Second Way**
  - Reading the assembly program once, converting it to an intermediate form, and storing this intermediate form in a table in memory.
  - Then a second pass is made over the table instead of over the source program.
  - If there is enough memory (or virtual memory), this approach saves I/O time.
  - If a listing is to be produced, then the entire source statement, including all the comments, has to be saved.
  - If no listing is needed, then the intermediate form can be reduced to the bare essentials.
Pass One

- **Principle Function**
  - Build up a table called the symbol table, containing the values of all symbols

- **A Symbol**
  - A symbol is either a label or a value that is assigned a symbolic name by means of a pseudoinstruction such as BUFSIZE EQU 8192
  - To assign a value to a symbol in the label field of an instruction, the assembler must know the address of the instruction during execution.
  - To keep track of the execution-time, address of the instruction being assembled, the assembler maintains a variable during assembly, known as the ILC (Instruction Location Counter).

- **ILC (Instruction Location Counter)**
  - It is set to 0 at the beginning of pass one and incremented by the instruction length for each instruction processed.
Instruction Location Counter

<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.
Pass One: Three Internal Tables

• Tables
  – Pass one of most assemblers uses at least three internal tables
  – If needed, a literal table is also kept.

• The symbol table
  – The symbol table has one entry for each symbol.
  – Symbols are defined either by using them as labels or by explicit definition (e.g., EQU).
  – Each symbol table entry contains the symbol itself (or a pointer to it), its numerical value, and sometimes other information.
  – This additional information may include:
    1) The length of data field associated with symbol.
    2) The relocation bits. (Does the symbol change value if the program is loaded at a different address than the assembler assumed?)
    3) Whether or not the symbol is to be accessible outside the procedure.
The Symbol Table

• The symbol table
  – The symbol table has one entry for each symbol.
  – Symbols are defined either by using them as labels or by explicit definition
  – Each symbol table entry contains the symbol itself (or a pointer to it), its numerical value, and sometimes other information.

• The additional information
  – The length of data field associated with symbol.
  – The relocation bits. (Does the symbol change value if the program is loaded at a different address than the assembler assumed?);
  – Whether or not the symbol is to be accessible outside the procedure.

<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>

A symbol table
The Opcode Table

- **The Opcode table**
  - The opcode table contains at least one entry for each symbolic opcode (mnemonic) in the assembly language.
  - The Figure shows part of an opcode table.
  - Each entry contains the symbolic opcode, two operands, the opcode’s numerical value, the instruction length, and a type number that separates the opcodes into groups depending on the number and kind of operands.

<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>

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

• **Pseudoinstruction**
  - Some assemblers allow programmers to write instructions using immediate addressing even though no corresponding target language instruction exists.
  - Pseudo-immediate instructions: The assembler allocates memory for the immediate operand at the end of the program and generates an instruction that references it.
  - In this way, the programmer avoids explicitly writing a pseudoinstruction to allocate a word initialized to one value.

• **Literal**
  - Constants for which the assembler automatically reserves memory are called **literals**.
  - In addition to saving the programmer a little writing, literals improve the readability of a program by making the value of the constant apparent in the source statement.
  - Most current computers have immediate instructions, so their assemblers do not provide literals. Immediate instructions are quite common nowadays, but formerly they were unusual.
  - If literals are needed, a literal table is maintained during assembly, with a new entry made each time a literal is encountered. After the first pass, this table is sorted and duplicates removed.
Pass One (1)

```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; // misc. variables
    final int END_STATEMENT = -2; // signals end of input

    location_counter = 0; // assemble first instruction at 0
    initialize_tables(); // general initialization

    while (more_input) { // more_input set to false by END
        line = read_next_line(); // get a line of input
        length = 0; // # bytes in the instruction
        type = 0; // which type (format) is the instruction
    }
}
```

Pass one of a simple assembler.
Pass One (2)

```c
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) {
        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.
Pass One (3)

... 

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
}
}

Pass one of a simple assembler.
Pass One: Summary

• **Pass One**
  – As it reads the program, pass one has to parse each line to find the opcode (e.g., ADD), look up its type (basically, the pattern of operands), and compute the instruction’s length.
  – This information is also needed on the second pass, so it is possible to write it out explicitly to eliminate the need to do them again.
  – Write out a temporary file containing the type, opcode, length, and actual input line. Pass two reads them instead of the raw input file.
  – When the END pseudoinstruction is read, pass one is over. The symbol table and literal tables can be sorted at this point if needed. The sorted literal table can be checked for duplicate entries, which can be removed.

• **Structured Programming**
  – Structuring programs in this way has other advantages in addition to ease of programming.
  – If the assembler is being written by a group of people, the various procedures can be parceled out among the programmers. All the (nasty) details of getting the input are hidden away in read next_line.
  – If they should change (due to OS change), only one subsidiary procedure is affected, and no changes are needed to the pass one procedure itself.
Pass Two

• Function
  – The function of pass two is to generate the object program and possibly print the assembly listing.
  – In addition, pass two must output certain information needed by the linker for linking up procedures assembled at different times into a single executable file.

• Operation
  – The operation of pass two is more-or-less similar to that of pass one: it reads the lines one at a time and processes them one at a time.
  – Since we have written the type, opcode, and length at the start of each line (on the temporary file), all of these are read in to save some parsing.
  – The main work of the code generation is done by the procedures eval_type1, eval_type2, and so on. It generates the binary code for the instruction and returns it in code. Then it is written out.
  – The original source statement and the object code generated from it (in hexadecimal) can then be printed or put into a buffer for later printing.
  – After the ILC has been adjusted, the next statement is fetched.
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;        // fields of the instruction
    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
    }
}

Pass two of a simple assembler
A Simple Assembler (2)

... if (type != 0) {
    switch(type) {
        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) {
    more_input = false; // if so, perform housekeeping tasks
    finish_up(); // odds and ends
}

Pass two of a simple assembler
Common Errors

• **Assembly Language Programming**
  – Common errors often appear in the programs
  – The best thing for the assembler to do with an errant statement is to print an error message and try to continue assembly.

• **Common Errors**
  – A symbol has been used but not defined.
  – A symbol has been defined more than once.
  – The name in the opcode field is not a legal opcode.
  – An opcode is not supplied with enough operands.
  – An opcode is supplied with too many operands.
  – An octal number contains an 8 or a 9.
  – Illegal register use (e.g., a branch to a register).
  – The END statement is missing.
Symbol Table (1)

• **Symbol Table**
  - During pass one of the assembly process, the assembler accumulates information about symbols and their values that must be stored in the symbol table for lookup during pass two.
  - Several different ways are available for organizing the symbol table. All of them attempt to simulate an associative memory, which conceptually is a set of (symbol, value) pairs. Given the symbol, the associative memory must produce the value.

• **The Simplest Implementation**
  - The simplest implementation technique is indeed to implement the symbol table as an array of pairs, the first element of which is (or points to) the symbol and the second of which is (or points to) the value.
  - Given a symbol to look up, the symbol table routine just searches the table linearly until it finds a match.
  - This method is easy to program but is slow, because, on the average, half the table will have to be searched on each lookup.
Symbol Table (2)

• Another Way
  – Another way to organize the symbol table is to sort it on the symbols and use the binary search algorithm to look up a symbol.
  – This algorithm works by comparing the middle entry in the table to the symbol.
  – If the symbol comes before the middle entry alphabetically, the symbol must be located in the first half of the table.
  – If the symbol comes after the middle entry, it must be in the second half of the table.
  – If the symbol is equal to the middle entry, the search terminates.

• Complexity
  – Assuming that the middle entry is not equal to the symbol sought, we at least know which half of the table to look for it in. Binary search can now be applied to the correct half, which yields either a match, or the correct quarter of the table. Applying the algorithm recursively, a table of size $n$ entries can be searched in about $\log_2 n$ attempts. Obviously, this way is much faster than searching linearly, but it requires maintaining the table in sorted order.
Symbol Table (3)

- **Hashing**
  - A completely different way of simulating an associative memory is a technique known as hash coding or hashing.
  - This approach requires having a “hash” function that maps symbols onto integers in the range 0 to \( k - 1 \).
  - One possible function is to multiply the ASCII codes of the characters in the symbols together, ignoring overflow, and taking the result modulo \( k \) or dividing it by a prime number.
  - In fact, almost any function of the input that gives a uniform distribution of the hash values will do. Symbols can be stored by having a table consisting of \( k \) buckets numbered 0 to \( k - 1 \).

- **Complexity**
  - All the (symbol, value) pairs whose symbol hashes to \( i \) are stored on a linked list pointed to by slot \( i \) in the hash table. With \( n \) symbols and \( k \) slots in the hash table, the average list will have length \( n/k \).
  - By choosing \( k \) approximately equal to \( n \), symbols can be located with only about one lookup on the average. By adjusting \( k \) we can reduce table size at the expense of slower lookups.
Example

• Compute hash code:
  – By adding up the letters and taking the result modulo the hash table size
  – The letter A=1, B=2, ……
  – The hash table has 19 slots, numbered 0 to 18

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Values</th>
<th>Hashing Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>els</td>
<td>(5 + 12 + 19)</td>
<td>17</td>
</tr>
<tr>
<td>jan</td>
<td>(10 + 1 + 14)</td>
<td>6</td>
</tr>
<tr>
<td>jelle</td>
<td>(10 + 5 + 12 + 12 + 5)</td>
<td>6</td>
</tr>
<tr>
<td>maaike</td>
<td>(13 + 1 + 1 + 9 + 11 + 5)</td>
<td>2</td>
</tr>
</tbody>
</table>

mod 19
## Exercise

- **Compute hash code:**
  - By adding up the letters and taking the result modulo the hash table size
  - The letter A=1, B=2, ……
  - The hash table has 20 slots, numbered 0 to 19

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Values</th>
<th>Hashing Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>els</td>
<td>(5 + 12 + 19)</td>
<td></td>
</tr>
<tr>
<td>jan</td>
<td>(10 + 1 + 14)</td>
<td></td>
</tr>
<tr>
<td>jelle</td>
<td>(10 + 5 + 12 + 12 + 5)</td>
<td></td>
</tr>
<tr>
<td>maaike</td>
<td>(13 + 1 + 1 + 9 + 11 + 5)</td>
<td></td>
</tr>
</tbody>
</table>
Exercise

• **Compute hash code:**
  – By adding up the letters and taking the result modulo the hash table size
  – The letter A=1, B=2, ……
  – The hash table has 20 slots, numbered 0 to 19

\[
\text{mod } 20
\]

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Values</th>
<th>Hashing Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>els</td>
<td>(5 + 12 + 19)</td>
<td>16</td>
</tr>
<tr>
<td>jan</td>
<td>(10 + 1 + 14)</td>
<td>5</td>
</tr>
<tr>
<td>jelle</td>
<td>(10 + 5 + 12 + 12 + 5)</td>
<td>4</td>
</tr>
<tr>
<td>maaike</td>
<td>(13 + 1 + 1 + 9 + 11 + 5)</td>
<td>0</td>
</tr>
</tbody>
</table>
HW2

• Problem 1:
  – Consider the operation of a machine with the data path of figure 2-2. Suppose that loading the ALU input registers takes 5 nsec, running the ALU takes 10 nsec, and storing the result back in the register scratchpad takes 5 nsec. What is the maximum number of MIPS this machine is capable of in the absence of pipelining?

  – Solution:

    The cycle time = 5+10+5 = 20 nsec. Since there is no pipelining, the best the machine can do is 1/20nsec = 50 * 10^6 instructions. So the maximum number of instructions is 50 MIPS.
Problem 23:

To burn a CD-R, the laser must pulse on and off at a high speed. When running at 10x speed in mode 1, what is the pulse length, in nanoseconds?

Solution: (Hints: Lecture 8, page 18)

The mode does not matter, since the laser has to pulse for preamble bits, data bits, ECC bits, and all the overhead bits as well.

The gross data rate at 1x is 75 sectors/sec. Each sector consists of $98 \times 588 = 57,624$ bits. Thus $75 \times 57,624 = 4,321,800$ bits/sec fly by the head at 1x.

At 10x, this is $43,218,000$ bits/sec. Thus each pulse must last no more than $(1/43,218,000) = 23.14$ nsec (actually slightly less, since there is a blank interval between pulses).
HW2

• **Problem 33:**
  – A high end digital camera has a sensor with 16 million pixels, each with 3 bytes/pixel. How many pictures can be stored on a 1 GB flash memory card if the compression factor is 5x? Assume that 1 GB means $2^{30}$ bytes.

  – **Solution:**

  1 photo has $16 \times 10^6$ pixels = $16 \times 3 \times 10^6 = 48$ million bytes.

  When compressed the size will be $48 / 5 = 9.6$ million bytes.

  Number of such images that can be stored = $(2^{30}) / (9.6 \times 10^6) = 111$ with a few megabytes left.
HW3

- **Problem 6:**
  
  - Given the memory values below and a one address machine with an accumulator, what values do the following instructions load into the accumulator?

  Word 20 contains 40
  Word 30 contains 50
  Word 40 contains 60
  Word 50 contains 70

  a. LOAD IMMEDIATE 20
  b. LOAD DIRECT 20
  c. LOAD INDIRECT 20
  d. LOAD IMMEDIATE 30
  e. LOAD DIRECT 30
  f. LOAD INDIRECT 30

- **Solution:**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>20</td>
</tr>
<tr>
<td>B</td>
<td>40</td>
</tr>
<tr>
<td>C</td>
<td>60</td>
</tr>
<tr>
<td>D</td>
<td>30</td>
</tr>
<tr>
<td>E</td>
<td>50</td>
</tr>
<tr>
<td>F</td>
<td>70</td>
</tr>
</tbody>
</table>
HW3

• Problem 32:
  - A computer uses DMA to read from its disk. The disk has 64 512 byte sectors per track. The disk rotation time is 16 msec. The bus is 16 bits wide, and bus transfers take 500 nsec each. The average CPU instruction requires two bus cycles. How much is the CPU slowed down by DMA?

  - Solution:

    Bus width = 16 bits = 2 bytes. Bus transfer rate = 500 nsec.
    The disk rotation time is 16 msec. That means, to transfer entire data of a track, it requires 16 msec.

    The amount of data in a track = 64 * 512 byte = 32,768 bytes.
    The time spent to transfer 32,768 bytes of data from disk = 16 * 10^-3 s.
    So, time spent to transfer 2 bytes of data from disk = 976 nsec.

    Therefore, time spent by the CPU for 2 bytes = (required time – bus transfer time) = (976-500) = 476 nsec

    So the CPU takes= (476/976)*100 = 48.8%
    Therefore due to DMA, the CPU is slowed down by (100 – 48.8) = 51.2%