System Software & Assembly Language (S&G, §§6.1–6.3)

Important Reminder!

• We hope you now have a clear understanding of what CS 100 is about and what is expected of you
• We hope that you have discovered that, by applying yourself, you can learn many interesting ideas from computer science
• However, if this is not the course for you, we remind you that you can still withdraw (until February 23)

Read S&G §6.4 (Operating Systems) for Thursday

Putting Clothes on the Naked Machine

• Imagine if we had to use the “naked” Von Neumann machine discussed last week
  – Must program purely in binary (1’s and 0’s)
  – Must not make mistakes
  – Must manually load instructions into memory
  – Must manually load address of first instruction into PC
• Bottom line: Von Neumann machine designed from perspective of machine, not humans!

Putting Clothes on the Naked Machine (2)

• The Von Neumann Machine needs a user interface
  – Hide messy details of hardware from user
  – Communicate with user using the user’s language, not binary
  – Provide easy access to computer’s resources
  – Prevent user from damaging hardware
• System software provides just such an interface; think of it as a well-designed dashboard

Example: Carry out the computation:

if ($a > b$) then
  set $c$ to $a + b$
else
  set $c$ to $a - b$
write value of $c$ to disk

Putting Clothes on the Naked Machine (3)
Putting Clothes on the Naked Machine (4)

- Imagine if you were responsible for
  - Fetching the values \(a, b\) from memory
  - Loading them into ALU registers
  - Putting the result into memory location \(c\)
  - Writing \(c\) to proper sector on a disk
- Should a programmer really have to worry about such things as registers and the ALU? No!
- System software provides a virtual machine that’s much friendlier than the Von Neumann machine

System Software

Types of system software:
- Language translators
  - Assemblers
    - Translate assembly language into machine language
  - Compilers
    - Translate high-level programming languages into assembly language or machine language
- Memory managers
- File systems
- Scheduler
- Utilities

Assemblers and Assembly Language

- As we have learned, it is difficult to write programs in machine language
  - We don’t deal well with 1’s and 0’s
  - We don’t deal well with numeric variable names
  - Machine language is difficult to change
- Assembly language is a first step toward a more user-friendly programming language
  - Supports English command set (e.g., LOAD, STORE)
  - Each command translates directly into a single machine language command
  - Supports mnemonic variable naming (e.g., “COUNT”)
  - Supports “pseudo-ops” for data generation
  - Supports comments

System Software (2)

Types of system software (cont.)
- Memory managers
  - Allocate and manage memory space for programs and data; load data and programs into memory
- File systems
  - Allow us to read from and write to mass storage devices like hard disks and CD-ROMs
- Scheduler
  - Schedules programs to be executed; maintains priority queue so that higher priority programs are executed first
- Utilities
  - Text editors
  - Graphics libraries
  - Graphical user interface libraries
  - Debuggers

System Software (3)

- We do not have time to cover all types of system software in a single week
- In order to get a taste of virtual machines and systems software, we will focus on two specific types of system software
  - Assemblers (this lecture)
  - Operating systems (next lecture)
Assembly Language (cont.)

Locating assembly language on the programming language continuum:

<table>
<thead>
<tr>
<th>Machine language</th>
<th>Assembly language</th>
</tr>
</thead>
<tbody>
<tr>
<td>(low level)</td>
<td>(high level)</td>
</tr>
</tbody>
</table>

Example: Output Sum of 1, ..., 10

```
Set sum to 0
Set count to 1
While count ≤ 10 do
  Set sum to sum + count
  Increment count
Output sum
```

Problems With Machine Language

- Binary op codes are difficult to remember
- Uses binary addresses for data, but humans are used to symbols (PI, LENGTH, etc.)
- Hard to tell the difference between data and instructions

Labeling Principle

- **Labeling Principle:**
  - Avoid arbitrary sequences more than a few items long
  - Do not require the user to know the absolute position of an item in a list
  - Instead, associate a meaningful label with each item and allow the items to occur in any order
- Example: symbolic labels for instructions & data

Tedious and Error-Prone

- Editing code is unbelievably difficult
  - Must shift instructions and data in memory
  - Must recalculate and edit addresses of operands
- Some errors can go unnoticed
  - The operand of a jump can be the address of a datum
  - The operand of a data manipulation can be the address of an instruction
  - The HALT instruction might not exist
Automation Principle

- **Automation Principle**: Automate mechanical, tedious, or error-prone activities.
- **Examples**:
  - Translation of decimal numbers and character data to binary
  - Translation of symbolic operation names to binary op codes

Assembly Language

- Provides built-in symbols for op codes (LOAD, STORE, etc.)
- Address fields must also be programmer-named symbols (PI, ENDF, etc.)
- Special syntax picks out data from instructions

Opcode Symbols

<table>
<thead>
<tr>
<th>Binary Opcode</th>
<th>Assembly Language Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>LOAD</td>
</tr>
<tr>
<td>0001</td>
<td>STORE</td>
</tr>
<tr>
<td>0010</td>
<td>CLEAR</td>
</tr>
<tr>
<td>0011</td>
<td>ADD</td>
</tr>
<tr>
<td>0100</td>
<td>INCREMENT</td>
</tr>
<tr>
<td>0101</td>
<td>SUBTRACT</td>
</tr>
<tr>
<td>0110</td>
<td>DECREMENT</td>
</tr>
<tr>
<td>0111</td>
<td>COMPARE</td>
</tr>
<tr>
<td>1000</td>
<td>JUMP</td>
</tr>
<tr>
<td>1001</td>
<td>JUMPGT</td>
</tr>
<tr>
<td>1010</td>
<td>JUMPEQ</td>
</tr>
<tr>
<td>1011</td>
<td>JUMPLT</td>
</tr>
<tr>
<td>1100</td>
<td>JUMPNEQ</td>
</tr>
<tr>
<td>1101</td>
<td>IN</td>
</tr>
<tr>
<td>1110</td>
<td>OUT</td>
</tr>
<tr>
<td>1111</td>
<td>HALT</td>
</tr>
</tbody>
</table>

Operand Symbols

- Can be any symbol except those in the op code set
- Two types:
  - Labels of data (serve as variables and constants)
  - Labels of instructions (destinations of jumps)

Data Labels

Data labels name variables and are declared with the `.data` directive

Format of data label:

```
<any but jump opcode> <a label>
<a label>: .data <a decimal integer>
```

Example:

```
load n
n: .data 50
```

Instruction Labels

Instruction labels name instructions which are the destinations of jumps

Format of instruction label:

```
<jump opcode> <a label>
<a label>: <another instruction>
```

Example:

```
jumpgt okay
okay: out answer
```
Example: The Simplest Assembly Language Program

```
.begin
halt
.end
```

This program executes one instruction and halts.

*begin* and *end* are not instructions. They are directives that tell the computer where the program begins and ends.

Example: Declaring Data Labels

```
.begin
out eleven
halt
zero: .data 0
one: .data 1
eleven: .data 11
.end
```

This program reserves three cells of memory for data. The computer initializes these cells with the given numbers before any instructions are executed.

Format:

```
<symbol>: .data <decimal integer>
```

Example: Output the Sum of Two Input Numbers

```
.begin
in first
load first
add second
store sum
halt
first: .data 0
second: .data 0
sum: .data 0
.end
```

Example: Output Absolute Value

```
.begin
in x
load zero
compare x
jumpgt endif
subtract x
store x
endif: output x
halt
x: .data 0
zero: .data 0
.end
```

Full Example: Output Sum of 1 .. 10

```
.begin
clear sum
load one
store count
while: load count
compare eleven
jump while: sum: .data 0
count: .data 0
.endwhile: output sum
halt
one: .data 1
eleven: .data 11
sum: .data 0
count: .data 0
.end
```

Translation to Machine Code

- A program called an assembler translates an assembly language program to an equivalent program in machine code
- The assembler keeps track of all the binary op codes and addresses
- The assembler also catches some errors
Program Development

Editor
Create assembly language program

Assembler
Check for syntax errors and translate to machine code

Run-time system
Execute the machine language program

Not OK (error messages)

OK

Example Syntax Errors

1. \texttt{width} has not been declared as a data label
2. The \texttt{halt} instruction occurs below the data declaration section

The assembler prevents this incorrect program from running

Security Principle

- \textit{Security Principle:} No program that violates the definition of the language, or its own intended structure, should escape detection.
  - C. A. R. Hoare

- Examples:
  - Detect undefined symbolic labels
  - Detect multiple definition of symbolic labels

Benefits of Assembly Language

- Maintenance is much easier – just edit and assemble
- Code is much easier to understand
- Some errors are caught automatically before execution

Problems With Assembly Language

Still reflects underlying machine architecture and operations
- Uses \texttt{load, add, store} instead of algebraic notation for arithmetic expressions ($x = y + z$)
- No structured control statements like \texttt{while} and \texttt{if/else}
- Data are typeless (no \texttt{int, char, String})

Problems With Assembly Language (2)

- Each machine has a different architecture and instruction set, so it needs a different dialect of assembly language – Tower of Babel!
- Must rewrite programs for each new machine that comes along
Portability Principle
Avoid feature or facilities that are dependent on a particular computer or a small class of computers

A Better Programming Language
• We would like one standard language with translators for different machines
• This language should express the logic of problem solving more than the logic of a machine’s architecture
• This language should be more like pseudocode

High-Level Languages
• Code looks more like pseudocode, allows logic of problem solving to be expressed in syntax
• One standard for each language (“write once, run anywhere”)
• Compilers translate code to machine language of various target machines
• Error handling is much more extensive

Examples
• BASIC (1960s)
• Pascal (1970s)
• C++ (1980s)
• Java (1990s)
• ?

Implementing an Assembler
• An Assembler is a program. What is it written in?
• You can write it in machine language for your computer, but that’s very difficult
• You can write it in a language implemented by an assembler or compiler on another computer, but:
  – You have to another computer with this software
  – You end up with an assembler that runs on the other computer, not yours

Bootstrapping
• This sort of problem can be solved by “bootstrapping”
• Program the first version of the assembler in itself
• Hand translate this program to machine language
• Run and debug it, but correct errors in both the machine language and assembly language versions
• Once it is debugged, it ought to be able to assemble itself
• You can throw away the ML version, and from now on edit the AL version
• New assembler version is always assembled by the previous version
Translating Assembly Language Program into Machine Language

• Four basic tasks performed by assembler:
  – Convert symbolic op codes and addresses to binary
  – Do pseudo-ops
  – Write machine language program to file

• Assembler does this in two passes

• Pass 1: Build symbol table that maps labels to addresses; this will enable us to translate those labels to actual addresses in the second pass

Translating Assembly Language (2)

• Pass 2: Create machine language object file

Open new object file
Get first Assembly language instruction
While the instruction is not .END
  If instruction is .DATA
    Create binary value corresponding to value in address field
    Store the value in object file with address where to load it
  Else
    Look up the op code of the instruction
    Get binary value of op code
    Look up address field in symbol table
    Build machine language instruction and write to object file
EndIf
Get next instruction
EndWhile