Topics

• Assembly language

• IJVM instruction set

• Mic-1 simulator programming
  – http://www.ontko.com/mic1/
  – Available in 2nd floor PC lab
  – S/W found in directory C:\mic1

Structured Computer Organization

Level 5  Problem-oriented language level
         Translation (compiler)
Level 4  Assembly language level
         Translation (assembler)
Level 3  Operating system machine level
         Partial interpretation (operating system)
Level 2  Instruction set architecture level
         Interpretation (microprogram) or direct execution
Level 1  Microarchitecture level
         Hardware
Level 0  Digital logic level
**IJVM Instruction Set**

- ISA level uses machine (binary) language
  - Not very helpful to programmers!

- Interpretation
  - Software-controlled datapath
  - Require stored program with microinstructions

How do we “write” machine language?

<table>
<thead>
<tr>
<th>Hex</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10</td>
<td></td>
</tr>
<tr>
<td>0x59</td>
<td></td>
</tr>
<tr>
<td>0xA7</td>
<td></td>
</tr>
<tr>
<td>0x60</td>
<td></td>
</tr>
<tr>
<td>0x7E</td>
<td></td>
</tr>
<tr>
<td>0x99</td>
<td></td>
</tr>
<tr>
<td>0x8B</td>
<td></td>
</tr>
<tr>
<td>0x8F</td>
<td></td>
</tr>
<tr>
<td>0xB4</td>
<td></td>
</tr>
<tr>
<td>0x15</td>
<td></td>
</tr>
<tr>
<td>0xB6</td>
<td></td>
</tr>
<tr>
<td>0x80</td>
<td></td>
</tr>
<tr>
<td>0xAC</td>
<td></td>
</tr>
<tr>
<td>0x36</td>
<td></td>
</tr>
<tr>
<td>0x64</td>
<td></td>
</tr>
<tr>
<td>0x13</td>
<td></td>
</tr>
<tr>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>0x57</td>
<td></td>
</tr>
<tr>
<td>0x5F</td>
<td></td>
</tr>
<tr>
<td>0xC4</td>
<td></td>
</tr>
</tbody>
</table>
Assembly Language

• Uses symbolic names (mnemonics) and symbolic addresses (variables) for the machine language

• An assembler converts the assembly language into machine language

• Each statement produces exactly one machine instruction (1:1 mapping)

IJVM Instruction Set

<table>
<thead>
<tr>
<th>Hex</th>
<th>Mnemonic</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10</td>
<td>BIPUSH byte</td>
<td>Push byte onto stack</td>
</tr>
<tr>
<td>0x59</td>
<td>DUP</td>
<td>Copy top word on stack and push onto stack</td>
</tr>
<tr>
<td>0xA7</td>
<td>GOTO offset</td>
<td>Unconditional branch</td>
</tr>
<tr>
<td>0x6D</td>
<td>IADD</td>
<td>Pop two words from stack; push their sum</td>
</tr>
<tr>
<td>0x7E</td>
<td>IAND</td>
<td>Pop two words from stack; push Boolean AND</td>
</tr>
<tr>
<td>0x99</td>
<td>IFZERO offset</td>
<td>Pop word from stack and branch if it is zero</td>
</tr>
<tr>
<td>0xB8</td>
<td>IFLT offset</td>
<td>Pop word from stack and branch if it is less than zero</td>
</tr>
<tr>
<td>0x9F</td>
<td>IF_ICMPEQ offset</td>
<td>Pop two words from stack; branch if equal</td>
</tr>
<tr>
<td>0xB4</td>
<td>IINC vvar, const</td>
<td>Add a constant to a local variable</td>
</tr>
<tr>
<td>0x15</td>
<td>ILOAD vvar</td>
<td>Push local variable onto stack</td>
</tr>
<tr>
<td>0x86</td>
<td>INVOKEVIRTUAL</td>
<td>Invoke a method</td>
</tr>
<tr>
<td>0x80</td>
<td>IOR</td>
<td>Pop two words from stack; push Boolean OR</td>
</tr>
<tr>
<td>0xAC</td>
<td>IRETURN</td>
<td>Return from method with integer value</td>
</tr>
<tr>
<td>0x36</td>
<td>ISTORE vvar</td>
<td>Pop word from stack and store in local variable</td>
</tr>
<tr>
<td>0xB4</td>
<td>ISUB</td>
<td>Pop two words from stack; push their difference</td>
</tr>
<tr>
<td>0x13</td>
<td>LDC_W index</td>
<td>Push constant from constant pool onto stack</td>
</tr>
<tr>
<td>0x00</td>
<td>NOP</td>
<td>Do nothing</td>
</tr>
<tr>
<td>0x57</td>
<td>POP</td>
<td>Delete word on top of stack</td>
</tr>
<tr>
<td>0x5F</td>
<td>SWAP</td>
<td>Swap the two top words on the stack</td>
</tr>
<tr>
<td>0xC4</td>
<td>WIDE</td>
<td>Prefix instruction; next instruction has a 16-bit index</td>
</tr>
</tbody>
</table>

Figure 4.11: The IJVM instruction set. The operands byte, const, and vvar are 1 byte. The operands disp, index, and offset are 2 bytes.
Why Use Assembly?

• **Versus machine language**
  – Easier to remember mnemonics for instructions instead of corresponding machine language
  – Easier to work with the symbolic addresses instead of numerical values of address

• **Versus high-level language**
  – **Access**: has access to all features and instructions of the ISA
  – **Performance**: Code produced can be much smaller (for low-memory devices) and faster (for speed-critical functions)

---

### Assembly vs. High-Level Programming

<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>1</td>
<td>90</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>6</td>
<td>30</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>

**5x effort for 3x speedup**

**10% of program responsible for 90% of execution time**

**Tune critical 10% for improved performance**

*Figure 7.1. Comparison of assembly language and high-level language programming, with and without tuning.*

**We will use assembly for the Mic-1 simulator**
Declaring Variables

```
.main
// all variables declared within .var and .end-var
.var
j
i
n
.end-var
// Main program goes here
// Program execution is terminated with a HALT statement
.end-main
```

IJVM Assembly Mnemonics

- **BIPUSH value**
  - Pushes the value onto the stack
  - Octal, decimal, hexadecimal, character (ASCII)

- **ISTORE variable**
  - Pops a value from the stack and places it in the variable

- **ILOAD variable**
  - Pushes the value in the variable onto the stack
BIPUSH Value Formats

- BIPUSH *value*
  - Value is one byte
  - Value is decimal by default
    - BIPUSH 32
  - Leading 0 for octal number
    - BIPUSH 032
  - Leading 0x for hexadecimal number
    - BIPUSH 0x32
  - Leading ‘ for ASCII characters
    - BIPUSH ‘M

IJVM Assembly Mnemonics

- IADD
  - Pops two values off the stack
  - Adds them together
  - Pushes the result onto the stack

- ISUB
  - Pops two values off the stack
  - Subtracts the first value popped from the second value popped off the stack
  - Pushes the result onto the stack
ISUB in Detail

- First value popped from stack passes through ALU to H register (A input)
- Inverting A and adding 1 gives the 2’s complement
- Second value popped from stack is added to 2’s complement

Storing a Value into a Variable

- Use BIPUSH to push the value onto the stack and then use ISTORE to pop it off and place it in the variable.
- Use IN to get it from the keyboard and push it onto the stack and then use ISTORE to pop it off and place it in the variable.
“Simple” Addition

// i = 3; j = 4; n = i + j;
BIPUSH 3
ISTORE i
BIPUSH 4
ISTORE j
ILOAD i
ILOAD j
IADD
ISTORE n
HALT

“Simple” Subtraction

// i = 3; j = 4; n = j - i;
BIPUSH 3
ISTORE i
BIPUSH 4
ISTORE j
ILOAD j
ILOAD i
ISUB
ISTORE n
HALT
IJVM Assembly Mnemonics

- **IINC variable value**
  - Adds the value to the variable and stores the result back in the variable

- **DUP**
  - Copies the value on the top on the stack and pushes the same value onto the stack
  - The value at the top of the stack and the value one below the top of the stack will be the same (duplicated).

Labels and Conditional Statements

- **Label name**: A reference to a place in the program
  - L1: BIPUSH 8

- **IFEQ label name**
  - IFEQ L1
  - Pops a value off the stack if it is equal to zero than the program jumps to the label.

- **GOTO label name**
  - GOTO L1
  - Unconditional jump to the label
Loops in Assembly Language

BIPUSH 3
ISTORE i
L1: ILOAD i
IFEQ L2
IINC i -1
GOTO L1
L2: HALT

High-level language:
for (int i = 3; i == 0; i--)

Other Conditional Statements

• IFLT label name
  – Pop a value off the stack
  – If the value is less than zero jump to the label

• IF_ICMPEQ label name
  – Pop 2 values off the stack
  – If they are equal jump to the label.
Exercise
What does “z” contain?

BIPUSH 4
ISTORE i
BIPUSH 1
ISTORE z
L1:  ILOAD i
     IFLT L2
     ILOAD z
     DUP
     IADD
     ISTORE z
     IINC i -1
     GOTO L1
L2:  HALT

z = 32
(the loop executes 5x)