Topics

• Spring 2004 course announcement
  – Digital Systems Architecture
  – EECE 292-02

• Assembly review

• Java bytecode

Have you seen this before???

Additional Sample Code
Input and Output

• IN
  – Reads a character from the key buffer
  – Pushes its **ASCII** value onto the stack
  – If no key has been pressed, zero will be pushed onto the stack

• OUT
  – Pops a word off the stack
  – Prints it to the standard output text area
  – Can only output **ASCII** values

The ASCII character set is found in Fig. 2-41 on p. 110 of your text

Basic Program Format

```
 .constant
  // all constants declared within .constant and .end-constant
 .end-constant

 .main
 .var
  // all variables declared within .var and .end-var
 .end-var

  // Main program goes here
  // Program execution is terminated with a HALT statement
 .end-main

 .method methodname (parameter_a, parameter_b, etc.)
  // all methods declared within .method and .end-method
 .var
  // local variables for method
 .end-var

  // method code goes here
  // use IRETURN to go back to the main program
 .end-method
```
Sample Code for Using Constants

```assembly
.const
LargeNo 0x3E8 // 1000
.end-const
.main
.var
x
y
.end-var
LDC_W LargeNo
ISTORE x
```

```
Constant Pool
CPP
```

“Feature” in Mic-1 Simulator

```assembly
.const
largePos 0x7FFFFFFF
largeNeg 0x80000000
.end-const
.main
    LDC_W largeNeg
    HALT
.end-main
```

- The constant only holds 31 bits
- In 2’s complement, a negative number must have a ‘1’ in MSB
- Workaround solutions?
  - LDC_W and then shift left
    - Add 1 if needed
  - LDC_W, DUP, and IADD

The assembler will **not** generate a .IJVM file with a 32-bit constant
Structured Computer Organization

**Level 5**
- Problems-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

- Assembly language allows us to use ISA
- ISA is in machine language (bytecode)
- Microarchitecture implements the ISA

From Assembly to Bytecode

- An **assembler** converts the assembly language into machine language (ISA)
- Each assembly statement produces exactly one machine (ISA) instruction (1:1 mapping)
- The microarchitecture interprets the machine language to execute the program (bytecode)
Datapath Registers (Memory Control)

- **PC**
  - Program Counter
  - Contains address of next IJVM instruction to be executed

- **MBR**
  - Memory Buffer Register
  - 8-bit register used to read a single byte from memory

  *8-bit read-only port that reads from Method Area (byte addressable)*

Parts of IJVM Memory

- CPP, LV, and SP registers are pointers to **words**
- PC contains a **byte address**
IJVM Instruction Set

- **Mnemonics**
  - 1 byte

- **The operands** byte, const, varnum
  - 1 byte

- **The operands** disp, index, offset
  - 2 bytes

<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>0x60</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>0x69</td>
<td>IFEQ offset</td>
<td>Pop word from stack and branch if it is zero</td>
</tr>
<tr>
<td>0x9B</td>
<td>IFLT offset</td>
<td>Pop word from stack and branch if 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>0x84</td>
<td>INC varnum const</td>
<td>Add a constant to a local variable</td>
</tr>
<tr>
<td>0x15</td>
<td>ILLOAD varnum</td>
<td>Push local variable onto stack</td>
</tr>
<tr>
<td>0x66</td>
<td>INVOKEVIRTUAL disp</td>
<td>Invoke a method</td>
</tr>
<tr>
<td>0x8D</td>
<td>IOR</td>
<td>Pop two words from stack; push Boolean OR</td>
</tr>
<tr>
<td>0x9C</td>
<td>IRETURN</td>
<td>Return from method with integer value</td>
</tr>
<tr>
<td>0x35</td>
<td>ISTORE varnum</td>
<td>Pop word from stack and store in local variable</td>
</tr>
<tr>
<td>0x64</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 varnum are 1 byte. The operands disp, index, and offset are 2 bytes.
Java Bytecode

```
i = j + k;
if (i == 3)
k = 0;
else
j = j - 1;
BIPUSH 3
ISTORE i
BIPUSH 4
ISTORE j
ILOAD i
ILOAD j
IADD
ISTORE n
HALT
```

```
0x10 0x03
0x36 0x01
0x10 0x04
0x36 0x02
0x15 0x01
0x15 0x02
0x15 0x01
0x60
0x36 0x03
0xFF
```

Figure 4.14. (a) A Java fragment. (b) The corresponding Java assembly language. (c) The JVM program in hexadecimal.

“Simple” Addition Bytecode

```
// i = 3; j = 4; n = i + j;
BIPUSH 3
0x10 0x03
ISTORE i
0x36 0x01
BIPUSH 4
0x10 0x04
ISTORE j
0x36 0x02
ILOAD i
0x15 0x01
ILOAD j
0x15 0x02
IADD
0x60
ISTORE n
0x36 0x03
HALT
0xFF
```
RPN Exercise – Convert to Bytecode

// (3 + 5) - (7 + 2)

Using RPN notation:
// 3 5 + 7 2 + -

• Push 3 onto stack
• Push 5 onto the stack
  – The stack now contains (3, 5)
• Use “+” operator
  – Pop two numbers from stack and push result (8)
• Push 7 onto the stack
• Push 2 onto the stack
  – The stack now contains (8, 7, 2)
• Use “+” operator
  – Pop two numbers from stack and push result (9)
• Use “-” operator
  – Pop two numbers from stack and push result (-1)
RPN Exercise – Convert to Bytecode

Using RPN notation:
// 3 5 + 7 2 + - (read from left to right)
• BIPUSH 3
• BIPUSH 5
  – The stack now contains (3, 5)
• IADD
  – Pop two numbers from stack and push result (8)
• BIPUSH 7
• BIPUSH 2
  – The stack now contains (8, 7, 2)
• IADD
  – Pop two numbers from stack and push result (9)
• ISUB
  – Pop two numbers from stack and push result (-1)

RPN Exercise – Convert to Bytecode

Using RPN notation:
// (3 + 5) - (7 + 2)

<table>
<thead>
<tr>
<th>Address</th>
<th>Bytecode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x10</td>
</tr>
<tr>
<td>1</td>
<td>0x03</td>
</tr>
<tr>
<td>2</td>
<td>0x10</td>
</tr>
<tr>
<td>3</td>
<td>0x05</td>
</tr>
<tr>
<td>4</td>
<td>0x60</td>
</tr>
<tr>
<td>5</td>
<td>0x10</td>
</tr>
<tr>
<td>6</td>
<td>0x07</td>
</tr>
<tr>
<td>7</td>
<td>0x10</td>
</tr>
<tr>
<td>8</td>
<td>0x02</td>
</tr>
<tr>
<td>9</td>
<td>0x60</td>
</tr>
<tr>
<td>A</td>
<td>0x64</td>
</tr>
<tr>
<td>B</td>
<td>0xFF</td>
</tr>
</tbody>
</table>
## Loops in Machine Language

### High-level language:

for (int i = 3; i != 0; i--)

- BIPUSH 3
- ISTORE i

**L1:**
- ILOAD i
- IFEQ L2
- IINC i -1
- GOTO L1

**L2:** HALT

### Address Bytecode

<table>
<thead>
<tr>
<th>Address</th>
<th>Bytecode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x10</td>
</tr>
<tr>
<td>1</td>
<td>0x03</td>
</tr>
<tr>
<td>2</td>
<td>0x36</td>
</tr>
<tr>
<td>3</td>
<td>0x01</td>
</tr>
<tr>
<td>4</td>
<td>0x15</td>
</tr>
<tr>
<td>5</td>
<td>0x01</td>
</tr>
<tr>
<td>6</td>
<td>0x99</td>
</tr>
<tr>
<td>7</td>
<td>0x00</td>
</tr>
<tr>
<td>8</td>
<td>0x09</td>
</tr>
<tr>
<td>9</td>
<td>0x84</td>
</tr>
<tr>
<td>A</td>
<td>0x01</td>
</tr>
<tr>
<td>B</td>
<td>0xFF</td>
</tr>
<tr>
<td>C</td>
<td>0xA7</td>
</tr>
<tr>
<td>D</td>
<td>0xFF</td>
</tr>
<tr>
<td>E</td>
<td>0xF8</td>
</tr>
<tr>
<td>F</td>
<td>0xFF</td>
</tr>
</tbody>
</table>