Topics

“The difference between fiction and reality? Fiction has to make sense.”

– Tom Clancy

• Administrative stuff
  – Turn in Program 1 deliverables

• IJVM loops

• INVOKEVIRTUAL

• Mic-1 Microinstructions

IJVM Assembly Mnemonics

• IAND
  – Pops two values off the stack
  – Performs logical AND of the values
  – Pushes the result onto the stack

• IOR
  – Pops two values off the stack
  – Performs logical OR of the values
  – Pushes the result onto the stack

Programming Teams

• Weblink from class schedule page

• Team 1: Noor, Ted, Tamim
• Team 2: Michael, David F., Nikhil
• Team 3: Theodor, Brandon, Jon
• Team 4: Brett, Jeb, John W.
• Team 5: Dante’, Clay, William
• Team 6: John L., Lauren, David R.
IJVM Assembly Mnemonics

- **NOP**
  - No operation; used as a “spacer” or “delay”

- **POP**
  - Deletes the word on the top of the stack

- **SWAP**
  - Exchanges the top two words on the stack

Loops in Machine Language

<table>
<thead>
<tr>
<th>High-level language:</th>
<th>Address</th>
<th>Bytecode</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (int i = 3; i &gt;= 0; i--)</td>
<td>0</td>
<td>0x10</td>
</tr>
<tr>
<td>BIPUSH 3</td>
<td>1</td>
<td>0x03</td>
</tr>
<tr>
<td>ISTORE i</td>
<td>2</td>
<td>0x36</td>
</tr>
<tr>
<td>L1:</td>
<td>3</td>
<td>0x00</td>
</tr>
<tr>
<td>ILOAD i</td>
<td>4</td>
<td>0x15</td>
</tr>
<tr>
<td>IFEQ L2</td>
<td>5</td>
<td>0x00</td>
</tr>
<tr>
<td>IINC i -1</td>
<td>6</td>
<td>0x99</td>
</tr>
<tr>
<td>GOTO L1</td>
<td>7</td>
<td>0x00</td>
</tr>
<tr>
<td>L2:</td>
<td>8</td>
<td>0x09</td>
</tr>
<tr>
<td>HALT</td>
<td>9</td>
<td>0x84</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>0x00</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>0xFF</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>0xA7</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>0xFF</td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td>0xF8</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>0xFF</td>
</tr>
</tbody>
</table>

Code Segment using Standard Output

BIPUSH 0
ISTORE i
BIPUSH '1'
ISTORE z
L1: ILOAD i
BIPUSH 10
ILOAD z
OUT
IINC z 1
IINC i 1
GOTO L1
L2: HALT

What does this Code Segment do?

Displays ASCII characters ‘1’ through ‘9’ and ‘:’

Loops in Machine Language

<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>0x00</td>
</tr>
<tr>
<td>4</td>
<td>0x15</td>
</tr>
<tr>
<td>5</td>
<td>0x00</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>0x00</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>

Exercise – IJVM Code Size

BIPUSH 0
ISTORE x
BIPUSH 0
ISTORE i
L1: BIPUSH 19
ILOAD i
ISUB
ILOAD x
LOAD i
LOAD i
IADD
ISTORE x
IINC i 1
GOTO L1
L2: HALT

int x = 0;
for (int i = 0; i < 20; i++)
x += i;

How many bytes would this code fragment use?

30 bytes
**INVOKEVIRTUAL**

- **INVOKEVIRTUAL methodname**
  - Calls a method
  - Load the OBJREF constant on the stack before the call
  - Load any parameters that need to be passed to the method
  - Finally, execute the INVOKEVIRTUAL instruction

**OBJREF**

- Pointer to an object
- Java is an object-oriented language
  - All functions are methods
- Methods are associated with objects
- To simply the Mic-1 architecture we will always use the same “default” object pointer “0x40”

**INVOKEVIRTUAL (cont)**

- **INVOKEVIRTUAL methodname**
  - Remember that methodname is the disp operand (2 bytes)
  - Added to the CPP to retrieve the PC for the method being invoked

**INVOKEVIRTUAL (cont)**

- **Method code for methodname**
  - Starts at the PC retrieved from Constant Pool
  - First 4 bytes contain special data
    - 2 bytes form a 16-bit integer for the number of parameters including OBJREF (parameter 0)
    - 2 bytes form a 16-bit integer indicating the size of the local variable area for the invoked method
    - Fifth byte contains the first opcode to be executed
Executing INVOKEVIRTUAL

- OBJREF is overwritten to store the address of old PC
- Old LV is stored immediately above
- SP points to the top word on the "empty" stack

IRETURN

- Used to return from a method
- The value placed on the TOS is the value returned
- Only TOS is returned, nothing below
- Stack is in same state as before call except for return value on TOS

Executing IRETURN

- Deallocates space used by returning method
- Restores stack to former state except
  - OBJREF (Link ptr) and parameters are popped off the stack
  - Returned value placed on top of stack

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>Pop two words from stack and push onto stack</td>
</tr>
<tr>
<td>0x70</td>
<td>GOTO offset</td>
<td>Unconditional branch</td>
</tr>
<tr>
<td>0x74</td>
<td>ADD</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>0x89</td>
<td>IFGE offset</td>
<td>Pop two words from stack and branch if it is zero</td>
</tr>
<tr>
<td>0x88</td>
<td>IFEQ offset</td>
<td>Pop two words from stack and branch if it is equal</td>
</tr>
<tr>
<td>0x84</td>
<td>IINC</td>
<td>Add a constant to a local variable</td>
</tr>
<tr>
<td>0x15</td>
<td>ILOAD varnum</td>
<td>Push local variable onto stack</td>
</tr>
<tr>
<td>0x18</td>
<td>IRETURN</td>
<td>Invoke a method</td>
</tr>
<tr>
<td>0x59</td>
<td>IOR</td>
<td>Pop two words from stack; push Boolean OR</td>
</tr>
<tr>
<td>0x84</td>
<td>IRETURN</td>
<td>Return from method with integer value</td>
</tr>
<tr>
<td>0x36</td>
<td>ISTORE varnum</td>
<td>Pop two words from stack and store in local variable</td>
</tr>
<tr>
<td>0x15</td>
<td>ISUB</td>
<td>Pop two words from stack and push their difference</td>
</tr>
<tr>
<td>0x13</td>
<td>IPOP</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>0x17</td>
<td>POP</td>
<td>Deletion word on top of stack</td>
</tr>
<tr>
<td>0x05</td>
<td>SWAP</td>
<td>Swap the two top words on the stack</td>
</tr>
<tr>
<td>0x24</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, constant, and varnum are 1 byte. The operands dup, index, and others are 2 bytes.
**Block Diagram of Microarchitecture (Mic-1)**

**Datapath**
Part of CPU containing ALU, its inputs, and its outputs

**Control Section**
Part of CPU containing the HW necessary to direct the datapath

**Purpose**
Implement the ISA level above it (macro-architecture)

---

**Control Store (ROM)**

- Memory that holds the microprogram
- Contains 512 words, each a 36-bit microinstruction
- Each microinstruction specifies its successor
  - Not executed in order stored in control store
- Accessing the microprogram
  - MicroProgram Counter holds address for next microinstruction
  - MicroInstruction Register holds the current microinstruction

---

**Microinstruction Notation**

- Could use the 36-bit words
  - Cumbersome method
  - Easier to abstract to pseudocode

- Things to be familiar with
  - Register names & functions
  - MAR/MDR & PC/MBR functions
  - IJVM memory model

---

**Mic-1 Simulator Window**

- Microinstruction – current microinstruction being executed
- NextMicroinstruction – next microinstruction to be executed
- MPC – current location in the control store
- Registers – value stored in each microarchitecture register
Microcode for a “Mystery” IJVM Instruction

Each line is 1 execution cycle! Several tasks are combined.

H = LV
MAR = MBRU + H; rd
PC = PC + 1; fetch
H = MDR
PC = PC + 1; fetch
MDR = MBR + H; wr; goto Main1

Figure 4-17 contains the Mic-1 microprogram

Assignment and ALU Operations

Invalid Operations
• MDR = SP + MDR
• H = H - MDR
• SP = H + MAR
• Etc.

Multiple assignments
• SP = MDR = SP + 1
• MDR = TOS = MBR
• Etc.
Memory Access

- **rd, wr**
  - Uses MAR/MDR to access constant pool, current local variable frame and current operand stack

- **fetch**
  - Uses PC/MBR to access method area

- **Initiated at the end of the cycle**
  - After the C bus is valid

- **Data availability for rd and fetch**
  - At the end of the next cycle

Branching

- **Unconditional**
  - goto label
  - Can explicitly name a successor for unconditional branch

- **Conditional**
  - Use ALU flags Z and N
    - Program Status Word
    - Set according to the result of the ALU operation
    - Ex: Z = TOS
    - If (Z) goto L1; else L2
    - If (N) goto L1; else L2

- **Multiway branch for next opcode**
  - goto (MBR OR value)

Program Status Word

ISA register containing status flags (p. 310)

- **N** – Set when the result is Negative
- **Z** – Set when the result is Zero
- **V** – Set when the result caused an Overflow
- **C** – Set when the result caused a Carry out of MSB
- **P** – Set when the result had even Parity

Summary

- INVOKEVIRTUAL is used to execute code from IJVM methods

- IJVM instructions are executed by a series of microinstructions that each take one cycle

- Microinstruction notation references the structures within the Mic-1 microarchitecture