JAM 8:
The Instruction Set
&
Sample Programs

Copyright Peter M. Kogge
CSE Dept. Univ. of Notre Dame
Jan. 8, 1999
Java Terms

• Java: “A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high performance, and dynamic language.” (Sun)

• Java Virtual Machine (JVM):
  - an Instruction Set Architecture into which Java programs are compiled
  - an interpreter which simulates this ISA (like XSPIM)

• JVM bytecodes: a sequence of bytes that define a JVM instruction

• Interpreter: software program that emulates at RTL level the execution of a program written in some ISA (usually different from that of the machine running the interpreted)
  - Java Interpreter: interprets programs constructed as JVM bytecodes

• Java Application: program written in Java that is executed like a program in any other language

• Java Applet:
  - (Small) program written in Java
  - Embedded in a web page
  - Loaded by, & interpreted in, web browser
**The Java Execution Model (non Applet)**

**.class file format:**
- Header information
- "Constant Pool"
  - Values of constants in the Java program
- "Interfaces"
  - Names of superclasses, interface methods, ...
- "Methods" (i.e. procedures)
  - One per method in original program
  - Storage requirements for execution
  - Compiled code in JVM "machine language"

JVM = “Java Virtual Machine”
- Instruction Set Architecture
- Easily interpreted by software
- Java chips under development

---

*java files (Java source) → Java Compiler → *.class files → Class Loader Parser Verifier → JVM bytecode interpreter*
The Java Virtual Machine

- Four 32 bit Programmer visible registers
  - `pc`: program counter = pointer to next bytecoded JVM instruction
  - `optop`: pointer to the top of a programmer-visible stack in memory
  - `frame`: pointer to memory area describing current method
  - `vars`: pointer to beginning of local variables for current method
- **Heap**: area in memory from which objects dynamically allocated
- **Stack**: area in memory referenced by optop, frame, vars registers
  - Separate stack allocated from heap at each method invocation
  - Holds arguments for, and results from, bytecode instructions
  - Holds arguments for, and results from, method invocations (local variables)
  - Holds state of each method invocation (its frame or environment)
  - Each stack word holds 32 bits (4 bytes)
- **Bytecoded Instructions**: one or more bytes that represent an instruction
- Support for operands of different sizes, 8, 16, 32 fixed, 32, 64 floating, strings, arrays, ...
JAM-8

• JAM-8 = “Java Abstract Machine - 8 bit version”
  - Too hard to say JVM-8
• Similarities to JVM
  - Same four registers
  - Subset of same bytecodes
  - Still stack oriented
• Differences from JVM
  - No heap
  - Only single stack which holds multiple method areas
  - All registers, objects only 8 bits wide
  - No floating point, extended precision operations
  - 16 bit operand JVM instructions work on 8 bit operands in JAM-8
  - All addresses only 8 bits; Memory space limited to 256 locations
  - Method call/return somewhat simplified
  - Only simple objects supported
JAM-8 Memory Model

- Memory
- Stack
  - $\text{vars} = \text{pointer to arguments and local variable}$
  - $\text{frame} = \text{pointer to state info for current method}$
  - $\text{optop} = \text{pointer to top of stack}$
    - "Pushing" involves decrementing optop
    - "Popping" involves incrementing optop
- Free Space
- Code
  - $\text{pc} = \text{pointer to next bytecoded instruction}$
JAM-8 Method Stack Environment

Caller's environment

Argument_0
Argument_1
...
Argument_n
old_pc
old_vars
old_frame
Working value
Topmost working

Stack grows down towards smaller addresses

d = pointer to start of current local variables (current method arguments and local storage)

A single environment for a method invocation

frame = pointer to info about caller

Intermediate values for this method's computation

optop = pointer to last used stack cell
JAM-8 Instruction Descriptions

- Each instruction includes:
  - Bytecode (in binary and hexadecimal)
  - Mnemonic
  - Description: English, stack, RTL
  - Instructions of form “<mnemonic> #” mean a bytecoded instruction followed by an 8 bit integer *used for address or data)

- Mnemonic is followed by footnotes:
  - “1” => same opcode and semantics as JVM
  - “2” => operand value is only 8 bits
  - “3” => “#” (2nd byte) is 1, not 2, bytes long (JVM has 2 bytes)
  - “4” => same opcode, but slightly different semantics from JVM
  - “5” => new instruction not present in JVM

- “Stack notation” used in bytecode descriptions:
  - ..., a1, a2, ... an => ..., b1, b2, ... bk
  - Left of => is stack before instruction is executed
  - Right of => is stack after instruction is executed
  - top n values are replaced by k results
## JAM-8: Basic Stack “Push” Operations

<table>
<thead>
<tr>
<th>Bytecode</th>
<th>Mnemonic</th>
<th>English/Stack/RTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>00010000 0x10</td>
<td>bipush # 1</td>
<td>Push a one byte constant to stack &lt;br&gt;... =&gt; ..., #&lt;br&gt;optop--; M(optop)&lt;-M(pc+1); pc&lt;-pc+2</td>
</tr>
<tr>
<td>01011001 0x59</td>
<td>dup 1</td>
<td>Duplicate top element of stack &lt;br&gt;..., value =&gt; ..., value, value&lt;br&gt;M(optop-1)&lt;-M(optop); optop--</td>
</tr>
<tr>
<td>00010101 0x15</td>
<td>iload # 1,2</td>
<td>Push local variable # to stack &lt;br&gt;... =&gt; ..., Local_variable_#&lt;br&gt;M(optop-1)&lt;-M(vars-#); optop--; pc&lt;-pc+2</td>
</tr>
<tr>
<td>11111110 0xFE</td>
<td>rload 5</td>
<td>Load using top of stack as address &lt;br&gt;..., adr =&gt; ..., M(adr)&lt;br&gt;M(optop)&lt;-M(M(optop)); pc++</td>
</tr>
<tr>
<td>00011001 0x19</td>
<td>aload # 4</td>
<td>Push address of local var # to stack &lt;br&gt;... =&gt; ..., vars-#&lt;br&gt;optop--; M(optop)&lt;-vars-#; pc&lt;-pc+2</td>
</tr>
</tbody>
</table>
# Basic Stack “Pop” Operations

<table>
<thead>
<tr>
<th>Bytecode</th>
<th>Mnemonic</th>
<th>Stack/RTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>00110110</td>
<td>istore #1,2</td>
<td>Pop stack to local variable # ..., value =&gt; ... M(vars-#)&lt;-M(optop++); pc&lt;-pc+2</td>
</tr>
<tr>
<td>0x36</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11111111</td>
<td>rstore 5</td>
<td>Pop stack top to memory ..., adr, value =&gt; ... M(M(optop+1))&lt;-M(optop); optop&lt;-optop+2; pc++</td>
</tr>
<tr>
<td>0xFF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01010111</td>
<td>pop 1</td>
<td>Pop top value off of stack and discard ..., value =&gt; ... optop++</td>
</tr>
<tr>
<td>0x57</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Basic Computational Operations

<table>
<thead>
<tr>
<th>Bytecode</th>
<th>Mnemonic</th>
<th>Stack/RTL</th>
</tr>
</thead>
</table>
| 01100000 0x60 | iadd | Add top two elements on stack  
... value1, value2 => result  
M(optop+1)<-M(optop+1)+M(optop); optop++; pc++ |
| 01100100 0x64 | isub | Subtract top two elements on stack  
... value1, value2 => result  
M(optop+1)<-M(optop+1)-M(optop); optop++; pc++ |
| 01111110 0x7E | iand | And top two elements on stack  
... value1, value2 => result  
M(optop+1)<-M(optop+1) and M(optop); optop++; pc++ |
| 10000000 0x80 | ior | Or top two elements on stack  
... value1, value2 => result  
M(optop+1)<-M(optop+1) or M(optop); optop++; pc++ |
| 10000100 0x84 | iinc #1,#2 | Increment local variable #1 by #2  
... => ...  
M(vars-#1)<-M(vars-#1)+#2; pc<-pc+3 |
| 01011111 0x5F | swap | Swap top two values  
... value1, value2 => ... value2, value1  
temp<-M(optop); M(optop)<-M(optop+1);  
M(optop+1)<-temp; pc++ |
# Basic Transfer of Control

<table>
<thead>
<tr>
<th>Bytecode</th>
<th>Mnemonic</th>
<th>Stack/RTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>10100111</td>
<td>goto #</td>
<td>go to #</td>
</tr>
<tr>
<td>0xA7</td>
<td>1,3</td>
<td>... =&gt; ...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC&lt;-#</td>
</tr>
<tr>
<td>10011001</td>
<td>ifeq #</td>
<td>if top of stack=0 then go to #</td>
</tr>
<tr>
<td>0x99</td>
<td>1,3</td>
<td>... , value =&gt; ...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if M(optop++)=0 then PC&lt;-# else PC&lt;-PC+2</td>
</tr>
<tr>
<td>10011010</td>
<td>ifne #</td>
<td>if top of stack&lt;&gt;0 then go to #</td>
</tr>
<tr>
<td>0x9A</td>
<td>1,3</td>
<td>... , value =&gt; ...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if M(optop++)&lt;&gt;0 then PC&lt;-# else PC&lt;-PC+2;</td>
</tr>
<tr>
<td>10011011</td>
<td>iflt #</td>
<td>if top of stack&lt;0 then go to #</td>
</tr>
<tr>
<td>0x9B</td>
<td>1,3</td>
<td>... , value =&gt; ...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if M(optop++)&lt;0 then PC&lt;-# else PC&lt;-PC+2;</td>
</tr>
<tr>
<td>10011100</td>
<td>ifge #</td>
<td>if top of stack&gt;=0 then go to #</td>
</tr>
<tr>
<td>0x9C</td>
<td>1,3</td>
<td>... , value =&gt; ...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if M(optop++)&gt;=0 then PC&lt;-# else PC&lt;-PC+2;</td>
</tr>
<tr>
<td>11001010</td>
<td>breakpoint</td>
<td>wait until external signal</td>
</tr>
<tr>
<td>0xC A</td>
<td>4</td>
<td>... =&gt; ...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>when “go” goes from 0 to 1 then continue</td>
</tr>
</tbody>
</table>
# Basic Method (Procedure) Invocation

<table>
<thead>
<tr>
<th>Bytecode</th>
<th>Mnemonic</th>
<th>Stack/RTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101000</td>
<td>jsr #</td>
<td>Jump to subroutine at #</td>
</tr>
<tr>
<td>0xA8</td>
<td></td>
<td>... =&gt; ..., return_address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>M(optop-1)&lt;-PC+2; optop--</td>
</tr>
<tr>
<td>10101001</td>
<td>ret</td>
<td>return from subroutine</td>
</tr>
<tr>
<td>0xA9</td>
<td></td>
<td>... , return_address =&gt; ...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pc&lt;-M(optop++)</td>
</tr>
<tr>
<td>10111000</td>
<td>invoke #</td>
<td>call new method</td>
</tr>
<tr>
<td>0xB8</td>
<td></td>
<td>... =&gt; ..., “space,” old_pc, old_vars, old_frame</td>
</tr>
<tr>
<td></td>
<td></td>
<td>optop&lt;-optop-M(#); M(optop)&lt;-pc+2;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>M(optop-1)&lt;-vars; vars&lt;-optop+M(#+1);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>M(optop-2)&lt;-frame; frame&lt;-optop-2;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pc&lt;-#+2; optop&lt;-optop-2</td>
</tr>
<tr>
<td>10101100</td>
<td>ireturn</td>
<td>return value to calling method</td>
</tr>
<tr>
<td>0xAC</td>
<td></td>
<td>... , value =&gt; “stack as of last invoke,” value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>M(vars)&lt;-M(optop); optop&lt;-vars;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pc&lt;-M(frame+2); vars&lt;-M(frame+1);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>frame&lt;-M(frame)</td>
</tr>
</tbody>
</table>
JAM-8 at Reset

• pc<-0; vars<-0; frame<-0; optop<-0
• Program execution starts at location 0
• First push of any value goes to location 255 (0xFF) - top of memory
• To allocate space for “global” variables, need an early “invoke”

0: invoke 3 (Note: this uses the two numbers at 3 as constants to set up stack space)

2: breakpoint

3: n+1 (a magic constant) where n=amount of storage for global variables; see “invoke”

4: n

5: start of main program - can now reference local vars
Compilation

- Constant value accessing: “bipush #” to place on stack
- Variable value accessing:
  - Local variable: “iload #” or “istore #” to move value to/from stack and “#” argument in current stack frame
  - Static/Global variable: “rload” or “rstore” use value on the stack as an address
- Expression evaluation: convert expression into “Reverse Polish”
  - Eg. A + (B and (C - D)) => A B C D - and +
  - Generate code from left to right: vars -> iloads, ops -> bytecodes
  - Eg. A B C D - and + => iload A; iload B; iload C; iload D; sub; and; add;
- “If <expr1> <comparator> <expr2> then <then_code> else <else_code>:”
  - Assume <comparator> is = or !=
  - Compile <expr1> to leave value on stack
  - Compile <expr2> to leave value on stack
  - “sub”
  - “ifeq” or “ifne” # where # is address of start of else code
  - Compile <then_code>, followed by “goto #” where # is after <else>
  - Compile <else_code>
Compilation (Continued)

- while <expr1> <comparator> <expr2> do <while_body>
  - “goto” # where # is address of start of condition
  - Compile <while_body>
  - Compile <expr1>
  - Compile <expr2>
  - “ifeq” or “ifne” with #, where # is address of start of <while_body>

- Simple procedure call
  - “jsr #” to save pc on stack and go to #. frame, vars unchanged
  - In body of procedure, for return use “ret”

- Method/Function call with arguments/local definitions: foo(a0, ... an)
  - Compile each argument ai to leave value on stack, in order 0, ... n
  - “invoke #” where # is address of start of procedure

- Function/method body
  - At entry, first byte has value = # of extra local storage needed +1
  - Next is a byte with value = # arguments + # local storage  
  - Following this is start of code for body of method
  - At completion, top of stack has return value, do “ireturn”
Sample Program

cchar n, m;
cchar temp[10];

main()
{
    initarray();
    m=total(1, 5, &temp[0]);
}

void initarray(void); // initialize global array temp
{
    n=0;
    while (n != 10)
    {
        temp[n]=n;
        n++
    }

byte total(char n, char m, char *ptr);
char sum=0; char i; // Sum up elements of input array from n to m
{
    for (i=n; i != m; i++)
    {
        sum=sum+* (ptr+i);
        return sum
}
Sample Compilation

char n, m;
char temp[10];

0: invoke 3 // initial call to establish storage for globals
2: breakpoint // this is here to stop the machine if it runs amok
3: 13 // space for n, m, and 10 array elts
    // note that n is local 0, m is local 1, temp starts at local 2
4: 12 // magic constant needed by invoke

main (void)
initarray;

5: jsr 18 // call initarray without building new context
m=total(1, 5, &temp[0]);

7: bipush 1 // first argument
9: bipush 5 // second argument
11: aload 2 // third argument — address of temp[0]
13: invoke 43 // call total with new frame
15: istore 1 // save returned value in m
17: breakpoint // our instruction to stop after main program is done

}
void initarray(void); // start initarray next
{
    n=0;
    18: bipush 0 // write 0 into n
    20: istore 0

while (n != 10)
    22: goto 35 // jump to the loop test
{
    temp[n]=n;
    24: aload 2 // compute address of temp[n]
    26: iload 0
    28: isub
    29: iload 0 // get value of n
    31: rstore // store n into temp(n)
    n++}

    32: iinc 0,1 // increment n by 1
    35: iload 0 // start of the loop test
    37: bipush 10
    39: isub
    40: ifne 24 // if loop continues, branch backwards to loop body
    42: ret // return from initarray
Sample Compilation (page 3)

byte total(char n, char m, char *ptr);

char sum=0; char i;

43: 3 // 2 local variables
44: 5 // 3 arguments + 2 local vars
45: bipush 0 // initial value for sum
47: istore 3 // store it in sum

for (i=n; i != m; i++) sum=sum+&(ptr+i);

49: iload 0 // get the local n
51: istore 4 // store in local i
53: goto 69 // enter the test code
55: iload 3 // get sum
57: iload 2 // get pointer to array
59: iload 4 // get i
61: isub // compute address of array
62: rload // get value of array location
63: iadd // compute new sum
64: istore 3 // and update sum
66: iinc 4,1 // increment i by 1
69: iload 4 // get i. start of loop test
71: iload 1 // get m
73: isub
75: ifne 55 // return to loop start

return sum}

77: iload 3
79: ireturn
Layers in the ISA

“Simple 12” equivalent ISA
- bipush
- rload
- rstore
- iadd
- isub
- iand
- ior
- goto
- ifeq
- iflt

Minimal “Java” extensions
- invoke
- ireturn
- aload

“Java” enhancements
- iload
- istore

Basic computing enhancements:
- jsr
- ret

Additional enhancements
- ifne
- ifgt
- swap
- dup
- pop
- iinc

JAM-8