JAM 16:
The Instruction Set
&
Sample Programs

Copyright Peter M. Kogge
CSE Dept. Univ. of Notre Dame
Jan. 8, 1999, modified 4/4/01
Revised to 16 bits: Dec. 5, 2007
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
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-16**

- JAM-16 = “Java Abstract Machine - 16 bit version”
  - Too hard to say JVM-16
- 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 16 bits wide
  - No floating point, extended precision operations
  - 16 bit integer subset selected
  - All addresses only 16 bits; Memory space limited to 65K locations
  - Method call/return somewhat simplified
  - Only simple objects supported
JAM-16 Memory Model

- Memory
- Stack
- Free Space
- Code

- 65K
- pc = pointer to next bytecoded instruction
- vars = pointer to arguments and local variables
- frame = pointer to state info for current method
- optop = pointer to top of stack
  - “Pushing” involves decrementing optop
  - “Popping” involves incrementing optop

Revised to 16 bits: Dec. 5, 2007
**JAM-16 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**

- **vars** = 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-16 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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>... =&gt; ..., #</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>..., value =&gt; ..., value, value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>... =&gt; ..., Local_variable_#</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>..., adr =&gt; ..., M(adr)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>... =&gt; ..., vars-#</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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>
</table>
| 00110110 0x36 | istore # 1,2 | Pop stack to local variable # ...
|           |          |           | ..., value => ...
|           |          |           | M(vars-#)<-M(optop++); pc<-pc+2 |
| 11111110x FF | rstore 5 | Pop stack top to memory ...
|           |          |           | ..., adr, value => ...
|           |          |           | M(M(optop+1))<-M(optop); optop<-optop+2; pc++ |
| 01010111 0x57 | pop 1 | Pop top value off of stack and discard ...
|           |          |           | ..., value => ...
|           |          |           | optop++ || pc++ |
### Basic Computational Operations

<table>
<thead>
<tr>
<th>Bytecode</th>
<th>Mnemonic</th>
<th>Stack/RTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>01100000</td>
<td>iadd</td>
<td>Add top two elements on stack</td>
</tr>
<tr>
<td>0x60</td>
<td></td>
<td>..., value1, value2 =&gt; result</td>
</tr>
<tr>
<td></td>
<td></td>
<td>M(optop+1)&lt;-M(optop+1)+M(optop); optop++; pc++</td>
</tr>
<tr>
<td>01100100</td>
<td>isub</td>
<td>Subtract top two elements on stack</td>
</tr>
<tr>
<td>0x64</td>
<td></td>
<td>..., value1, value2 =&gt; result</td>
</tr>
<tr>
<td></td>
<td></td>
<td>M(optop+1)&lt;-M(optop+1)-M(optop); optop++; pc++</td>
</tr>
<tr>
<td>01111111</td>
<td>iand</td>
<td>And top two elements on stack</td>
</tr>
<tr>
<td>0x7E</td>
<td></td>
<td>..., value1, value2 =&gt; result</td>
</tr>
<tr>
<td></td>
<td></td>
<td>M(optop+1)&lt;-M(optop+1) and M(optop); optop++; pc++</td>
</tr>
<tr>
<td>10000000</td>
<td>ior</td>
<td>Or top two elements on stack</td>
</tr>
<tr>
<td>0x80</td>
<td></td>
<td>..., value1, value2 =&gt; result</td>
</tr>
<tr>
<td></td>
<td></td>
<td>M(optop+1)&lt;-M(optop+1) or M(optop); optop++; pc++</td>
</tr>
<tr>
<td>10000100</td>
<td>iinc #1,#2</td>
<td>Increment local variable #1 by #2</td>
</tr>
<tr>
<td>0x84</td>
<td></td>
<td>... =&gt; ...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>M(vars-#1)&lt;-M(vars-#1)+#2; pc&lt;-pc+3</td>
</tr>
<tr>
<td>01011111</td>
<td>swap</td>
<td>Swap top two values</td>
</tr>
<tr>
<td>0x5F</td>
<td></td>
<td>..., value1, value2 =&gt; ..., value 2, value1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>temp&lt;-M(optop); M(optop)&lt;-M(optop+1); M(optop+1)&lt;-temp; pc++</td>
</tr>
</tbody>
</table>
### Basic Transfer of Control

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

<table>
<thead>
<tr>
<th>Bytecode</th>
<th>Mnemonic</th>
<th>Stack/RTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101000 0xAA</td>
<td>jsr #</td>
<td>Jump to subroutine at # ... =&gt; ..., return_address M(optop-1)&lt;-PC+2; optop--</td>
</tr>
<tr>
<td>10101001 0xAA</td>
<td>ret</td>
<td>return from subroutine ... , return_address =&gt; ... pc&lt;-M(optop++)</td>
</tr>
<tr>
<td>10111000 0xBB</td>
<td>invoke #</td>
<td>call new method ... =&gt; ..., “space,” old_pc, old_vars, old_frame optop&lt;-optop-M(#); M(optop)&lt;-pc+2; M(optop-1)&lt;-vars; vars&lt;-optop+M(#+1); M(optop-2)&lt;-frame; frame&lt;-optop-2; pc&lt;-#+2; optop&lt;-optop-2;</td>
</tr>
<tr>
<td>10101100 0xAC</td>
<td>ireturn</td>
<td>return value to calling method ... , value =&gt; “stack as of last invoke,” value M(vars)&lt;-M(optop); optop&lt;-vars; pc&lt;-M(frame+2); vars&lt;-M(frame+1); frame&lt;-M(frame)</td>
</tr>
</tbody>
</table>
JAM-16 at Reset

- pc<-0; vars<-0; frame<-0; optop<-0
- Program execution starts at location 0
- First push of any value goes to location 65,535 (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

int n, m;
int 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: 24 //space for n,m, and 10 array elts
   // note that n is local 0, m is local 2, temp starts at local 4
4: 24 // 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
Sample Compilation (page 2)

```c
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)

```c
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

Basic computing enhancements:
- jsr
- ret

“Java” enhancements
- iload
- istore

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

JAM-8