1.0 Stack Architectures - An Introduction

Computer architectures can be loosely divided into register architectures and stack architectures depending where the intermediate results of calculations are kept. Register architectures use special high-speed storage locations (registers) to hold the intermediate results of calculations. Registers are different from main memory in that they are faster, they are physically distinct from the main memory (registers are usually located on the processor chip), and they are addressed differently from main memory (although some architectures, e.g. the PDP-10, assign memory addresses to registers.) Registers allow for shorter (hence faster) instructions since the source or target operand of a calculation can be a register. Operations which only access registers are faster since referencing memory is comparatively slow. One extreme example of register architecture is the so-called load and store architecture where computations require that all operands be pre-loaded into registers eliminating the need to access memory except to load and store registers.

For example to execute the instruction \( x = a * b + c \) (where \( %r0 - %r3 \) are registers):

\[
\begin{align*}
\text{load } %r1, & \ a  \\
\text{load } %r2, & \ b  \\
\text{load } %r3, & \ c  \\
\text{mul } %r0, & \ %r1, %r2  \\
\text{add } %r0, & \ %r0, %r3  \\
\text{store x, } & \ %r0 \\
\end{align*}
\]

On the other hand stack architectures use a stack (not registers) to hold the intermediate results of calculations where all arithmetic and logical operations assume that operands are on the stack. For example, to add two integers, both values must first be pushed (loaded) onto the stack. The add operation then pops the stack twice, adds the two values then pushes the result back onto the top of the stack. The result on top of the stack can be popped and written to memory (store) or used as an operand for a subsequent operation. Since operands for arithmetic operations are on the stack, instruction formats for arithmetic instructions do not contain operand fields (contrast with the \texttt{mul} and \texttt{add} instructions above). In fact, the only operations that need to reference memory are the push (load) and pop (store) operations.

For example, to execute the instruction \( x = a * b + c \), a stack architecture would use the instruction sequence:

\[
\begin{align*}
\text{load c}  \\
\text{load b}  \\
\text{load a}  \\
\text{mul}  \\
\text{add}  \\
\text{store x} \\
\end{align*}
\]

Stack architectures have a number of advantages over register-based architectures; instructions are shorter since they do not need to explicitly reference operands and high-level arithmetic and logical expressions are easier to translate into machine code.

2.0 A Stack Architecture Implementation: The Java Virtual Machine Engine

An interesting example of a stack architecture is provided by Java Byte code. Java language programs are compiled to byte code which is the machine code for a stack-based virtual architecture; virtual meaning that in reality there is no such physical machine. Java byte code is then interpreted by the host architecture. This gives Java language programs two advantages. Since writing interpreters for Java byte code is fairly easy to do (the semantic gap between byte code and any computer's ISA is small) many architectures support Java interpreters. This makes it easy for many different computers to run Java programs. Second, such interpreters can be and are written to prevent Java programs from compromising the security of the host machine. Thus Java programs can be shared without the threat of compromising the security of another computer.
The stack architecture presented below executes a subset of the Java byte code.

**The JVM Engine Memory Model:** Since Java is an object-oriented language, the underlying Java virtual architecture is structured to support objects which bundle together data structures and the subroutines (called methods) that operate on them. (Of course the programs that we will write will not make use of objects!). This accounts for the rather complicated memory model presented below.

Most programs require two “areas” of memory - a code area and a data area. A stack requires a third area of memory (Think of the code, data and stack segments for the Intel 80x86.) However, to implement the Java language, there are four memory areas each pointed to by special *virtual* registers which are not directly accessible by Java language programs: a code area, a constant pool area for global constants, a local variables area for variables and of course a stack area. Again these hidden registers are not accessible by the Java byte code; they are only mentioned for completeness and to provide some background on how addressing is implemented.

<table>
<thead>
<tr>
<th>Variables</th>
<th>Constants</th>
<th>Code</th>
<th>Operand Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;- LV Reg.</td>
<td>&lt;- CPP</td>
<td>&lt;- PC</td>
<td></td>
</tr>
</tbody>
</table>

The **LV register** points to the base (i.e. first word) of the Local Variable Frame. Addresses of local variables are computed offsets from the base of the Local Variable Frame contained in the LV register. There is a separate Local Variable Frame for each Java object; hence all variables are local.

The **CPP** (Constant Pool Pointer) points to the base of the first word of the constant pool. Like local variables, the addresses of constants are computed offsets from the base of the Constant Pool Area which is contained in the CPP register. All constants are global.

The **SP** (Stack Pointer) points to the word on the top of the stack (this is the “stack” used for calculations). As items are pushed onto the stack, the stack grows downward in memory.

The **PC** (Program Counter) points to the byte address of the next instruction.

Java uses 32-bit integers but Java byte-code consists of variable length byte instructions. Hence the JVM Engine memory is both 32-bit word addressable and byte addressable. This means that the JVM Engine memory can be thought of as consisting of 4 gigabytes (32 bit addresses) or though of as consisting of 1 giga-word (30 bit addresses). Word addresses are converted to byte addresses by shifting the word address 2 bits to the left. For example word address 0x02F (binary 00101111) is byte address 0x0BC (binary 10111100). Words are aligned to begin at byte addresses whose right-most bits are 00. Data is word length (LV, CPP and SP hold word addresses); code is byte length (PC holds a byte address).
3.0 - A Sample Java Byte Code Program

To introduce Java Byte code we will examine a simple program. The following Java Byte code program evaluates the polynomial \( y = 2x^2 + 3x + 5 \) for \( x = 7 \). The code should be fairly straight-forward to follow.

The code itself is divided into three segments: a constant segment, a variables (data) segment and a code segment. The directive pairs .const, .end-const, .var, .end-var, .main and .end-main delimit each segment. Note that local variable segment is nested within the code segment.

The values for the coefficients \( a, b, \) and \( c \), and the initial value for \( x \) (\( x_0 \)) are stored as constants. The value \( x_0 \) is transferred to the variable \( x \) by pushing it onto the stack then popping it to variable \( x \). Data moves are done via the stack. The code then pushes the constants \( a, b, \) and \( c \) along with \( x \) onto the stack in such a way that a series of multiplications and additions results in the answer ending up on top of the stack. The result is popped to variable \( y \).

; A program to compute \( y = 2x^2 + 3x + 5 \) for \( x = 7 \)
.
.const        ; constant pool
  a 2
  b 3
  c 5
  x0 7
.end-const
.
.main         ; main method
.
.var          ; variable segment for main
  x 0
  y 0
.end-var

begin:     ; push initial x value
  ldc x0
  istore x

      ; push x
  iload x
  dup    ; duplicate top of stack
  imul   ; \( x^2 \)
  ldc a  ; push constant a
  imul   ; ax^2
  iload x ; push x
  ldc b  ; push constant b
  imul   ; bx
  ldc c  ; push constant c
  iadd   ; bx + c
  iadd   ; ax^2 + bx + c
  istore y ; pop to y

halt
.end-main
4.0 Java Byte Code Instructions (A Subset)

Java Byte Code instructions can be grouped according to function. The following is a minimal subset of Java Virtual Machine Engine byte code instructions. Included in the brief descriptions are the codes and how each affects the stack.

<table>
<thead>
<tr>
<th>assembler mnemonic</th>
<th>opcode</th>
<th>description</th>
<th>sample use</th>
<th>stack before</th>
<th>stack after</th>
<th>notes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stack operations</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iload</td>
<td>0x15</td>
<td>integer load (push)</td>
<td>iload loc</td>
<td>x</td>
<td>a</td>
<td>loc is location in LV area</td>
</tr>
<tr>
<td>istore</td>
<td>0x36</td>
<td>integer store (pop)</td>
<td>istore loc</td>
<td>a</td>
<td>x</td>
<td>loc is location in LV area</td>
</tr>
<tr>
<td>ldc</td>
<td>0x12</td>
<td>load constant</td>
<td>ldc c_loc</td>
<td>x</td>
<td>a</td>
<td>c_loc is location in CP area</td>
</tr>
<tr>
<td>dup</td>
<td>0x59</td>
<td>duplicate top of stack</td>
<td>dup</td>
<td>a</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>swap</td>
<td>0x5f</td>
<td>swap top items on stack</td>
<td>swap</td>
<td>a</td>
<td>b</td>
<td>sometimes operands are out of order</td>
</tr>
<tr>
<td>pop</td>
<td>0x57</td>
<td>pop and discard TOS</td>
<td>pop</td>
<td>a</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>bipush</td>
<td>0x10</td>
<td>push byte</td>
<td>bipush 0x13</td>
<td>x</td>
<td>0x13</td>
<td>operand is any byte value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>arithmetic operations</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iadd</td>
<td>0x60</td>
<td>add</td>
<td>iadd</td>
<td>a</td>
<td>b</td>
<td>a + b</td>
</tr>
<tr>
<td>isub</td>
<td>0x64</td>
<td>subtract</td>
<td>isub</td>
<td>a</td>
<td>b</td>
<td>b - a</td>
</tr>
<tr>
<td>imul</td>
<td>0x68</td>
<td>multiply</td>
<td>imul</td>
<td>a</td>
<td>b</td>
<td>a × b</td>
</tr>
<tr>
<td>idiv</td>
<td>0x6c</td>
<td>integer divide</td>
<td>idiv</td>
<td>a</td>
<td>b</td>
<td>b / a</td>
</tr>
<tr>
<td>irem</td>
<td>0x70</td>
<td>remainder</td>
<td>irem</td>
<td>a</td>
<td>b</td>
<td>b % a</td>
</tr>
<tr>
<td>ineg</td>
<td>0x74</td>
<td>integer negate</td>
<td>ineg</td>
<td>a</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>branching</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>goto</td>
<td>0x7a</td>
<td>goto (branch always)</td>
<td>goto offset</td>
<td>-</td>
<td>-</td>
<td>offset, a 16 bit value, is added to the PC</td>
</tr>
<tr>
<td>ifeq</td>
<td>0x99</td>
<td>branch on equal to 0</td>
<td>ifeq offset</td>
<td>a</td>
<td>-</td>
<td>if TOS == 0 then offset is added to PC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>also ifne (not equal), iflt (less than), ifle (less than or equal to), ifgt (greater than), ifge (greater than or equal to)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if_icmpeq</td>
<td>0x9f</td>
<td>integer compare equal</td>
<td>if_icmpeq</td>
<td>a</td>
<td>b</td>
<td>if a == b then offset is added to PC</td>
</tr>
</tbody>
</table>

also if_icmpne, if_icmplt, if_icmple, if_icmpgt, if_icmpge versions
The JVM Engine as a Stack Architecture

With the JVM Engine stack architecture, everything goes through the stack. Obviously the stack is used for arithmetic (and Boolean) operations but the top of the stack is what is tested by conditional branches. I/O operations transfer the top of the stack to/from the I/O buffers. Note that I/O is character oriented.

Operands when they are used are offsets into the Local Variables Frame (e.g. iload, istore), the Constant Pool (ldc) or the Code area (goto, etc). The memory area is determined by the op-code (iload loads a variable, ldc loads a constant). The eight-bit offsets for iload and istore instructions limit the number of variables to 256. Branching is all PC relative.
5.0 Writing JVM Engine Assembler Language Programs

Consider the following JVM Engine assembler language programming to compute 2+3:

```
.const          // constant pool
  a  2
  b  3
.end-const
.main
.var
  sum 0    // sum of 2 + 3
.end-var
.start: ldc a
        ldc b
        iadd
        istore sum
        halt
.end-main
```

The format for each statement is an optional label followed by an operation followed by an optional operand (depending on the operation) followed by an optional comment.

```
[label:] operation [operand] [// comment]
```

All labels terminate with a colon and all comments begin with a slash (or a semi-colon).

Constants (defined in the constant pool) are a symbolic name followed by a value followed by an optional comment.

```
const_name value [// comment]
```

Variables are defined in the local variables segment as a symbolic name followed by an initial value or 0 followed by an optional comment. All constants and variables are 32 bit signed integers.

```
var_name 0 [// comment]
```

JVM Engine assembly language programs have three segments: a constant pool segment, a (local) variables segment, and a (main) program code segment. Segments are marked by paired directives. The constant pool segment is marked by the paired directives .const .end-const (note both begin with a period). The main method code segment is marked by .main .end-main and the local variable segment is marked by the directives .var .end-var. Since variables are local to the main method, the .var .end-var directives go inside the .main .end-main directives.
The program as defined above will not display the answer. To display the character ‘5’, push the sum onto the stack, push a 48 onto the stack, add to convert the integer 5 to ASCII ‘5’ and use the OUT instruction to display it. The modification to the main method is given below.

```
.const // constant pool
  a 2
  b 3
.end-const
.main
.var
  sum 0 // sum of 2 + 3
.end-var
.start: ldc a
     ldc b
     iadd
     istore sum // get the sum
     iload sum // get the sum
     bipush 48 // add 48
     iadd // to convert to ASCII
     out // and display it
     bipush 0x0a // newline
     out
     halt
.end-main
```

### 6.0 Executing JVM Engine Assembly Language Programs

```
+----------------+    +-------------------+
pgm.txt -> | JVM Assembler | -> pgm.jve -> | Java Engine |
+----------------+    +-------------------+
```

**Step 1:** Using any text editor (e.g. NotePad) create a JVM Engine assembly language source code file with a .txt file extension (e.g. pgm.txt).

**Step 2:** To assemble the JVM source code file into a bytecode object file that can be run using the Java Engine simulator, execute the program JVMAssembling.exe. This program will prompt you for the name of the source code file and the object code file. (A convention is to use the file extension .jve for the object code file). If successful the program will display the object file to the screen;

```
*******************************
*                         *
*     JVM Assembler       *
*    (08/03/07)           *
*                         *
*******************************

Enter source file name example01.txt
Enter object file name example01.jve
```
Step 3: To run the current version of the JVM Engine simulator, execute the program jvm02b.exe

6.1 Description of Java Virtual Machine Engine

There are three areas to the display screen

Java Engine Memory Areas (upper right) which has four areas

- Constant Pool area
- Local Variables area
- Stack
- Code

I/O Text Area (lower left)

Buttons (Upper Left)

- Execute – with trace/display
- Step – execute next operation
- Reset – reset PC to top of code
- Load Pgm – load object code file obtained from previous assembly
- Hex/Dec – switch between hexadecimal/signed decimal display
- Quit – exit simulator
7.0 Advanced JVM Assembly Language Programs

7.1 JVM Engine Assembler I/O: The two I/O operations \texttt{in} and \texttt{out} are non-standard Java Byte code ops supported by the JVM engine assembler and Java Engine simulator program. The \texttt{in} operation grabs the next character from the input buffer and pushes it onto the stack. If the input buffer is empty, it pushes a zero (ASCII null). The top of the stack can be checked using an \texttt{ifeq} instruction to see if a character was read and if so the program can continue. If a character was not read (the top of the stack is zero) the program loops back to the \texttt{in} instruction (a busy waiting loop). The following six instruction sequence is used to read a character.

\begin{verbatim}
getch:  in ; get next input character & push on stack
dup ; duplicate top of stack
ifeq reread ; check if top of stack == 0. If so check again
goto done ; else goto to done
reread: pop ; pop stack
goto getch ; loop to get character
done:
\end{verbatim}

The \texttt{dup} instruction is used to duplicate the top of the stack so the following \texttt{ifeq} instruction which pops the stack can be used to test if the character read was a 0. If true control jumps to the \texttt{reread: pop} which gets rid of the duplicated 0 and branches back to the \texttt{getch: in} instruction. However, if the top of the stack is not zero, control breaks out of the loop (\texttt{goto done}) leaving the character read by the \texttt{in} instruction on top of the stack.

The \texttt{out} instruction is simpler to use. It pops the top of the stack to the output buffer. Character strings are displayed by first using the \texttt{bipush} operation to push in reverse order the ASCII codes of the characters onto the stack then using \texttt{out} to display each character.

The following JVM Engine assembler program reads a character, adds one to its ASCII code, displays it and halts. Note the three segments. The Constant Pool Area holds the constant value 1, the Local Variable Frame segment stores the character read in, and the Code segment contains the main method to read a character, add one to its ASCII code, display it and halt.

\begin{verbatim}
; Author : B. Shelburne
; File   : Sample04.txt
; Date   : August 3, 2007
; Desc   : Reads a character and output its ASCII successor

.const ; constant pool area
    one  1
.end-const
.main ; main method
.var ; local variable frame
    char 0
.end-var
start:  bipush 63 ; push ? on stack
        out ; and display
        bipush 32 ; push blank on stack
        out ; and blank
getch:  in ; get next input character & push on stack
dup ; duplicate top of stack
ifeq reread ; if top of stack == 0 skip
goto done ; else go to done
reread: pop ; pop stack
goto getch ; loop to get character
\end{verbatim}
done: dup ; duplicate top of stack
   out ; and display it
   istore char ; store char
   bipush 0x0a ; push new line character on stack
   out ; and display it
   iload char ; load char
   ldc one ; load 1
   iadd ; add 1 to character code
   out ; display it
   halt

.end-main

7.2 Subroutines

The jump to subroutine instruction jsr pushes a return address on to the stack then branches to the subroutine at a signed 16 bit offset from the address of the jsr instruction (hence the value of pc+3 is pushed onto the stack). The first instruction of the subroutine is an astore instruction which stores the address in the local variables area of memory. The return address must be stored in memory since the return instruction ret effects a return by fetching the return address from memory and placing it in the PC; the stack is not used for the return.

Consider the following example below which calls a subroutine to read a character from the keyboard

; Author: B. Shelburne
; File:   Sample05.txt
; Date:   August 1, 2007
;
; Desc:  Calls a subroutine to read a character
;        then outputs its ASCII successor
;
.const ; constant pool area
   one 1
.end-const

.main ; main method

.var ; local variable frame
   char 0
   link 0 ; holds return address
.end-var

start: bipush 63 ; push ? on stack
   out ; and display
   bipush 32 ; push blank on stack
   out ; and blank
   jsr getch ; call getch – return char on stack
   dup ; duplicate top of stack
   out ; and display it
   istore char ; store char
   bipush 0x0a ; push new line character on stack
   out ; and display it
   iload char ; load char
   ldc one ; load 1
   iadd ; add 1 to character code
   out ; display it
   halt ; end of main code

; subroutine

;
getch:      astore link ; store return address
loop:       in      ; get next input character & push on stack
dup         ; duplicate top of stack
ifeq reread ; if top of stack == 0 skip
goto done  ; else go to done
reread:     pop      ; pop stack
goto loop  ; loop to get character
done:       ret link ; return
.end-main

7.3 Programming with Dynamic Arrays

In Java Byte code, integer arrays are dynamically allocated using the newarray int instruction. The capacity of the array is pushed onto the stack before the newarray int instruction is executed which returns the address of the array on the stack. The astore instruction can be used to save the address (pointer to the array).

```
iload size       ; push size onto stack
newarray int     ; allocate array
astore a         ; store address
```

The iastore instruction is used to write to an array; the address of the array, the index and the value to be stored are pushed onto the stack (in that order).

```
aload a          ; push address of array
iload index      ; push index
iload x          ; push value
iastore          ; a[index] <- x
```

To access the elements in an array push the array address and the index and execute the ilaload instruction pops the stack for the index and array address and pushes the array item onto the stack.

```
aload a          ; push address of array
iload index      ; push index
iaload           ; TOS <- a[index]
istore x         ; x <- a[index]
```

A countdown loop is used to cycle through the items in the array. Just before the loop, a loop counter count is initialized to the number of times the loop is to be executed (in this case, the size of the array). Control branches to the bottom of the loop where count is tested to see if it’s positive and if it is control branches to the top of the loop; otherwise control drops through the loop to the next statement. Each pass through the loop decrements the loop counter using the iinc count -1 instruction. This occurs just before the test for zero.

```
iload size       ; initialize loop counter
istore count     ; to the size of the array
goto end1        ; goto to bottom of loop
loop1:           ;
                  ; body of loop goes here
iinc count -1    ; decrement loop counter
end1: iload count ; test if
ifgt loop1       ; loop counter is positive
```

All of the above is used in the following program which allocates an integer array of capacity 10, stores the powers of 2 from 1 to 512 in the array, then reads and sums the value in the array.
; Author: B. Shelburne
; File: Sample06.txt
; Date: August 6, 2007
;
; Desc: Demonstrates (dynamic) arrays. Creates an integer
; array of capacity 10, inserts the values 1, 2, 4 ... 512
; into the array then reads the array summing the values.
;
.const
    size 10 ; array capacity
.end-const
.main
    a 0 ; holds address of array
    index 0 ; index for a
    count 0 ; loop counter
    n 0
    sum 0
.begin:
    ldc size
    newarray int ; allocate int a[size]
    astore a ; store address
    bipush 0
    istore index ; initialize index to 0
    bipush 1
    istore n ; initialize n to 1
    ldc size
    istore count ; initialize loop counter to 10
    goto end1
.loop1:
    aload a ; push address of a
    iload index ; push index
    iload n ; push n
    lastore ; a[index] <- n
    iload n ; get n
    bipush 1
    ishl ; double n
    istore n ; & store result
    iinc index 1 ; index++
    iinc count -1 ; decrement loop counter
.end1:
    iload count ; loop control
    ifgt loop1
    bipush 0
    istore index ; reset index to 0
    bipush 0
    istore sum ; initialize sum to 0
    ldc size
    istore count ; initialize loop counter to 10
    goto end2
.loop2:
    aload a ; push address of a
    iload index ; push index
    iaload ; TOS <- a[index]
    iload sum
    iadd
    istore sum ; sum = sum + a[index]
iinc index 1 ; index++
iinc count -1 ; decrement loop counter
end2:  iload count ; loop control
ifgt loop2
halt
.end-main

7.4 Java Byte Code

What does Java Byte code look like? For example, here is a short Java Byte code program to compute x = 2+ 3 and y = x * x. First the program uses ldc to load the constants 2 and 3 onto the stack, add them, then store the result at x. Then it loads x back onto the stack, duplicates the top of the stack, multiplies and stores the product at y.

.const
   a  2
   b  3
.end-const
.main
.var
   x  0
   y  0
.end-var
main: ldc a
   ldc b
   iadd ; x <- a + b
   iload x
   istore x
   iload x
   dup
   imul ; y <- x * x
   istore y
   halt
.end-main

When assembled, the following output is generated by the JVMAssembler program which displays the constant pool, the local variables, then the byte code itself.

2 constant area
0 2 A
1 3 B
2 variables area
0 0 X
1 0 Y
14 code area
0x12 ldc  0   0x12 0x00
0x00 a
0x12 ldc  2   0x12 0x01
0x00 b
0x60 iadd  4   0x60
0x36 istore  5   0x36 0x00
0x00 x
0x15 iload  7   0x15 0x00
0x00 x
0x59 dup  9   0x59
0x68 imul 10   0x68
0x36 istore 11   0x36 0x01
0x01 y
0xFF halt 13 0xff

*** Constant Pool Dump ***
0 : A : 2
1 : B : 3

*** Local Variables Dump ***
0 : X : 0
1 : Y : 0

*** Labels Table Dump ***
0 : MAIN : 0
### 8.0 The JVM Engine Instruction Set

The instruction set for the JVM Engine is a subset of Java Byte code instructions. Instructions are limited to integer-type operations with the instructions being variable length byte sequences anywhere from 1 to 4 bytes long (the Java virtual machine bytecode). Below we group the instructions by function giving their op-code and assembler language mnemonics.

#### 8.1 Stack Operations

<table>
<thead>
<tr>
<th>assembler mnemonic</th>
<th>opcode</th>
<th>description</th>
<th>sample use</th>
<th>stack before</th>
<th>stack after</th>
<th>notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>iload</td>
<td>0x15</td>
<td>integer load (push)</td>
<td>iload loc</td>
<td>x</td>
<td>a</td>
<td>loc is location in LV area ([loc] = a)</td>
</tr>
<tr>
<td>istore</td>
<td>0x36</td>
<td>integer store (pop)</td>
<td>istore loc</td>
<td>a x</td>
<td>x</td>
<td>loc is location in LV area ([loc] &lt; - a)</td>
</tr>
<tr>
<td>pop</td>
<td>0x57</td>
<td>pop and discard TOS</td>
<td>pop</td>
<td>a</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>bipush</td>
<td>0x10</td>
<td>push byte</td>
<td>bipush 0x13</td>
<td>x</td>
<td>a x</td>
<td>operand is any byte value</td>
</tr>
<tr>
<td>dup</td>
<td>0x59</td>
<td>duplicate top of stack</td>
<td>dup</td>
<td>a</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>dup2</td>
<td>0x5c</td>
<td>duplicate twice</td>
<td>dup2</td>
<td>a b</td>
<td>a</td>
<td>duplicate top two items on stack</td>
</tr>
<tr>
<td>swap</td>
<td>0x5f</td>
<td>swap top items on stack</td>
<td>swap</td>
<td>a b</td>
<td>a</td>
<td>used if operands are out of order</td>
</tr>
<tr>
<td>ldc</td>
<td>0x12</td>
<td>load constant</td>
<td>ldc c_loc</td>
<td>x</td>
<td>a x</td>
<td>c_loc is location in CP area ([c_loc] = a)</td>
</tr>
<tr>
<td>pop</td>
<td>0x57</td>
<td>pop and discard TOS</td>
<td>pop</td>
<td>a</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>aload</td>
<td>0x19</td>
<td>address load (push)</td>
<td>aload loc</td>
<td>-</td>
<td>addr</td>
<td>push the address stored at loc in LV onto stack</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>See notes below</td>
</tr>
<tr>
<td>iaload</td>
<td>0x2e</td>
<td>integer array load (push)</td>
<td>iaload</td>
<td>i addr</td>
<td>a[i]</td>
<td>push a[i] onto the stack where addr = [a]</td>
</tr>
<tr>
<td>astore</td>
<td>0x3a</td>
<td>address store (pop)</td>
<td>astore loc</td>
<td>addr</td>
<td>-</td>
<td>pop address on TOS to loc in LV area</td>
</tr>
<tr>
<td>iastore</td>
<td>0x4f</td>
<td>integer array store (pop)</td>
<td>iastore</td>
<td>b i [a]</td>
<td>-</td>
<td>assign b to a[i]</td>
</tr>
<tr>
<td>newarray</td>
<td>0xbc</td>
<td>dynamic allocation of 1-D integer array</td>
<td>newarray int</td>
<td>n</td>
<td>addr</td>
<td>int is a built in symbol; newarray dynamically allocates in integer array of capacity n and returns its address onto the stack</td>
</tr>
</tbody>
</table>
Arrays are dynamically allocated using the `newarray` instruction which returns on the top of the stack the address of the first component in the array (an address from the heap). The `astore` instruction is used to store this address at a location in the LV area.

`newarray int` evaluates to 0xbc 0x0a – this takes the integer n on the top of the stack, allocates from (dynamic) memory an integer array with capacity n, and pushes on the stack the address of the array; note `int` is a literal

; This is like the Java code:
;    int a[] = new int[20];

bipush 20     ; push 20 onto the stack
newarray int  ; call newarray to allocate a 20-element int array
astore a      ; store the reference to the array in LV area

To access elements in an array (using `iaload` to push an array item onto the stack or `iastore` to pop an item from the stack to the array, the address in the LV area where the address of the array is stored and the index value (0 indexing) must be pushed onto the stack (everything goes thru the stack).

Example: push `a[2]`, where a contains the address of an int array

aload a        ; push offset where address of integer array is stored
bipush 2       ; push the index 2 onto the stack
iaload         ; TOS <- a[2]

### 8.2 Arithmetic Operations

<table>
<thead>
<tr>
<th>assembler mnemonic</th>
<th>opcode</th>
<th>description</th>
<th>sample use</th>
<th>stack before</th>
<th>stack after</th>
<th>notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>iadd</td>
<td>0x60</td>
<td>add</td>
<td>iadd</td>
<td>a b</td>
<td>a + b</td>
<td>no operands needed</td>
</tr>
<tr>
<td>isub</td>
<td>0x64</td>
<td>subtract</td>
<td>isub</td>
<td>a b</td>
<td>b - a</td>
<td>note order! TOS is subtracted</td>
</tr>
<tr>
<td>imul</td>
<td>0x68</td>
<td>multiply</td>
<td>imul</td>
<td>a b</td>
<td>a x b</td>
<td></td>
</tr>
<tr>
<td>idiv</td>
<td>0x6c</td>
<td>integer divide</td>
<td>idiv</td>
<td>a b</td>
<td>b / a</td>
<td>note order! TOS is divided</td>
</tr>
<tr>
<td>irem</td>
<td>0x70</td>
<td>remainder</td>
<td>irem</td>
<td>a b</td>
<td>b % a</td>
<td>note order!</td>
</tr>
<tr>
<td>ineg</td>
<td>0x74</td>
<td>integer negate</td>
<td>ineg</td>
<td>a</td>
<td>-a</td>
<td></td>
</tr>
<tr>
<td>iinc</td>
<td>0x84</td>
<td>integer increment</td>
<td>iinc a i</td>
<td>-</td>
<td>-</td>
<td>increment a in LV area by i does not affect stack a and i are 8-bit values</td>
</tr>
</tbody>
</table>
### 8.3 Logical Operations

<table>
<thead>
<tr>
<th>assembler mnemonic</th>
<th>opcode</th>
<th>description</th>
<th>sample use</th>
<th>stack use</th>
<th>stack before</th>
<th>stack after</th>
<th>notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>logical operations</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iand</td>
<td>0x7e</td>
<td>logical and</td>
<td>iand</td>
<td>a</td>
<td>b</td>
<td>a &amp; b</td>
<td>no operands needed</td>
</tr>
<tr>
<td>ior</td>
<td>0xb0</td>
<td>logical or</td>
<td>ior</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>ixor</td>
<td>0x82</td>
<td>exclusive or</td>
<td>ixor</td>
<td>a</td>
<td>b</td>
<td>a xor b</td>
<td></td>
</tr>
<tr>
<td>ishl</td>
<td>0x78</td>
<td>logical shift left</td>
<td>ishl</td>
<td>n</td>
<td>a</td>
<td>a*2^n</td>
<td>logical left shift a by n bits</td>
</tr>
<tr>
<td>ishr</td>
<td>0x7a</td>
<td>arithmetic shift right</td>
<td>ishr</td>
<td>n</td>
<td>a</td>
<td>a / 2^n</td>
<td>arithmetic right shift a by n bits</td>
</tr>
<tr>
<td>iushl</td>
<td>0x7c</td>
<td>logical right shift</td>
<td>iushl</td>
<td>n</td>
<td>a</td>
<td>a &gt;&gt; n</td>
<td>logical right shift a by n bits</td>
</tr>
</tbody>
</table>

### 8.4 I/O Operations (Non Standard)

<table>
<thead>
<tr>
<th>assembler mnemonic</th>
<th>opcode</th>
<th>description</th>
<th>sample use</th>
<th>stack use</th>
<th>stack before</th>
<th>stack after</th>
<th>notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>in</td>
<td>0xfc</td>
<td>input to TOS</td>
<td>in</td>
<td>-</td>
<td>ch</td>
<td>-</td>
<td>push byte character from input buffer to TOS; if buffer is empty push 0x00</td>
</tr>
<tr>
<td>out</td>
<td>0xfd</td>
<td>pop TOS to output</td>
<td>out</td>
<td>ch</td>
<td>-</td>
<td>-</td>
<td>pop byte character from TOS to output buffer</td>
</tr>
</tbody>
</table>
## 8.5 Branching Instructions

<table>
<thead>
<tr>
<th>assembler mnemonic</th>
<th>opcode</th>
<th>description</th>
<th>sample use</th>
<th>stack before</th>
<th>stack after</th>
<th>notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>goto</td>
<td>0x7a</td>
<td>goto (branch always)</td>
<td>goto offset</td>
<td>-</td>
<td>-</td>
<td>offset, a 16 bit value, is added to the PC</td>
</tr>
<tr>
<td>ifeq</td>
<td>0x99</td>
<td>branch on equal to 0</td>
<td>ifeq offset</td>
<td>a</td>
<td>-</td>
<td>if TOS == 0 then offset is added to PC</td>
</tr>
<tr>
<td>ifne</td>
<td>0x9a</td>
<td>branch on not equal to 0</td>
<td>ifne offset</td>
<td>a</td>
<td>-</td>
<td>if TOS != 0 then offset is added to PC</td>
</tr>
<tr>
<td>iflt</td>
<td>0x9b</td>
<td>branch on less than 0</td>
<td>iflt offset</td>
<td>a</td>
<td>-</td>
<td>if TOS &lt; 0 then offset is added to PC</td>
</tr>
<tr>
<td>ifle</td>
<td>0x9c</td>
<td>branch on less than or equal to 0</td>
<td>ifle offset</td>
<td>a</td>
<td>-</td>
<td>if TOS &lt;= 0 then offset is added to PC</td>
</tr>
<tr>
<td>ifgt</td>
<td>0x9d</td>
<td>branch on greater than 0</td>
<td>ifgt offset</td>
<td>a</td>
<td>-</td>
<td>if TOS &gt; 0 then offset is added to PC</td>
</tr>
<tr>
<td>ifge</td>
<td>0x9e</td>
<td>branch on greater than or equal to 0</td>
<td>ifge offset</td>
<td>a</td>
<td>-</td>
<td>if TOS &gt;= 0 then offset is added to PC</td>
</tr>
<tr>
<td>if_icmpeq</td>
<td>0x9f</td>
<td>integer compare equal</td>
<td>if_icmpeq offset</td>
<td>a</td>
<td>b</td>
<td>-</td>
</tr>
<tr>
<td>if_icmpne</td>
<td>0xa0</td>
<td>integer compare not equal</td>
<td>if_icmpne offset</td>
<td>a</td>
<td>b</td>
<td>-</td>
</tr>
<tr>
<td>if_icmplt</td>
<td>0xa1</td>
<td>integer compare less than</td>
<td>if_icmplt offset</td>
<td>a</td>
<td>b</td>
<td>-</td>
</tr>
<tr>
<td>if_icmple</td>
<td>0xa4</td>
<td>integer compare less than or equal to</td>
<td>if_icmple offset</td>
<td>a</td>
<td>b</td>
<td>-</td>
</tr>
<tr>
<td>if_icmpgt</td>
<td>0xa3</td>
<td>integer compare greater than</td>
<td>if_icmpgt offset</td>
<td>a</td>
<td>b</td>
<td>-</td>
</tr>
<tr>
<td>if_icmpge</td>
<td>0xa2</td>
<td>integer compare greater than or equal to</td>
<td>if_icmpge offset</td>
<td>a</td>
<td>b</td>
<td>-</td>
</tr>
</tbody>
</table>

## 8.6 Subroutines

<table>
<thead>
<tr>
<th>assembler mnemonic</th>
<th>opcode</th>
<th>description</th>
<th>sample use</th>
<th>stack before</th>
<th>stack after</th>
<th>notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>jsr</td>
<td>0xa8</td>
<td>jump to subroutine</td>
<td>jsr offset</td>
<td>-</td>
<td>PC+3</td>
<td>push PC+3 onto stack offset added to PC</td>
</tr>
<tr>
<td>ret</td>
<td>0xa9</td>
<td>return</td>
<td>ret loc</td>
<td>-</td>
<td>-</td>
<td>move contents of loc in LV area to PC</td>
</tr>
<tr>
<td>Halt</td>
<td>0xff</td>
<td>halt</td>
<td>halt</td>
<td>-</td>
<td>-</td>
<td>halt – non-standard</td>
</tr>
</tbody>
</table>
jsr offset – Jump to subroutine at signed 16-bit offset from jsr byte; push return address (pc+3) onto stack. Note – first instruction in subroutine should be astore which stores return address in LV area.

ret loc – return from subroutine; loc is a 8-bit offset in LV that holds the return address; does not affect stack.

Example

    jsr subr ; jump to subroutine subr
     . . .

    subr: astore link ; pop return offset to link
     . . .
    ret link ; return

9.0 Addenda