Chapter Contents

4.1 Hardware Components of the Instruction Set Architecture
4.2 ARC, A RISC Computer
4.3 Pseudo-Ops
4.4 Examples of Assembly Language Programs
4.5 Accessing Data in Memory—Addressing Modes
4.6 Subroutine Linkage and Stacks
4.7 Input and Output in Assembly Language
4.8 Case Study: The Java Virtual Machine ISA
The Instruction Set Architecture

- The *Instruction Set Architecture (ISA)* view of a machine corresponds to the machine and assembly language levels.

- A *compiler* translates a high level language, which is architecture independent, into assembly language, which is architecture dependent.

- An *assembler* translates assembly language programs into executable binary codes.

- For fully compiled languages like C and Fortran, the binary codes are executed directly by the target machine. Java stops the translation at the byte code level. The *Java virtual machine*, which is at the assembly language level, interprets the byte codes (hardware implementations of the JVM also exist, in which Java byte codes are executed directly.)
The System Bus Model of a Computer System, Revisited

• A compiled program is copied from a hard disk to the memory. The CPU reads instructions and data from the memory, executes the instructions, and stores the results back into the memory.
Common Sizes for Data Types

- A byte is composed of 8 bits. Two nibbles make up a byte.
- Halfwords, words, doublewords, and quadwords are composed of bytes as shown below:

<table>
<thead>
<tr>
<th>Bit</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nibble</td>
<td>0110</td>
</tr>
<tr>
<td>Byte</td>
<td>10110000</td>
</tr>
<tr>
<td>16-bit word (halfword)</td>
<td>11001001 01000110</td>
</tr>
<tr>
<td>32-bit word</td>
<td>10110100 00110101 10011001 01011000</td>
</tr>
<tr>
<td>64-bit word (double)</td>
<td>01011000 01010101 10110000 11110011 11001110 11101110 01111000 00110101</td>
</tr>
<tr>
<td>128-bit word (quad)</td>
<td>01011000 01010101 10110000 11110011 11001110 11101110 01111000 00110101 00001011 10100110 11110010 11100110</td>
</tr>
<tr>
<td></td>
<td>10100100 01000100 10100101 01010001</td>
</tr>
</tbody>
</table>
Big-Endian and Little-Endian Formats

• In a byte-addressable machine, the smallest datum that can be referenced in memory is the byte. Multi-byte words are stored as a sequence of bytes, in which the address of the multi-byte word is the same as the byte of the word that has the lowest address.

• When multi-byte words are used, two choices for the order in which the bytes are stored in memory are: most significant byte at lowest address, referred to as big-endian, or least significant byte stored at lowest address, referred to as little-endian.

Word address is $x$ for both big-endian and little-endian formats.
Memory Map for the ARC

- Memory locations are arranged linearly in consecutive order. Each numbered location corresponds to an ARC word. The unique number that identifies each word is referred to as its address.
Abstract View of a CPU

- The CPU consists of a data section containing registers and an ALU, and a control section, which interprets instructions and effects register transfers. The data section is also known as the datapath.

![Diagram of CPU with datapath, control unit, registers, and ALU]
The Fetch-Execute Cycle

- The steps that the control unit carries out in executing a program are:
  1. Fetch the next instruction to be executed from memory.
  2. Decode the opcode.
  3. Read operand(s) from main memory, if any.
  4. Execute the instruction and store results.
  5. Go to step 1.

This is known as the *fetch-execute cycle*. 
An Example Datapath

- The ARC datapath is made up of a collection of registers known as the register file and the arithmetic and logic unit (ALU).
The ARC ISA

- The ARC ISA is a subset of the SPARC ISA.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld</td>
<td>Load a register from memory</td>
</tr>
<tr>
<td>st</td>
<td>Store a register into memory</td>
</tr>
<tr>
<td>sethi</td>
<td>Load the 22 most significant bits of a register</td>
</tr>
<tr>
<td>andcc</td>
<td>Bitwise logical AND</td>
</tr>
<tr>
<td>orcc</td>
<td>Bitwise logical OR</td>
</tr>
<tr>
<td>orncc</td>
<td>Bitwise logical NOR</td>
</tr>
<tr>
<td>srl</td>
<td>Shift right (logical)</td>
</tr>
<tr>
<td>addcc</td>
<td>Add</td>
</tr>
<tr>
<td>call</td>
<td>Call subroutine</td>
</tr>
<tr>
<td>jmpl</td>
<td>Jump and link (return from subroutine call)</td>
</tr>
<tr>
<td>be</td>
<td>Branch if equal</td>
</tr>
<tr>
<td>bneg</td>
<td>Branch if negative</td>
</tr>
<tr>
<td>bcs</td>
<td>Branch on carry</td>
</tr>
<tr>
<td>bvs</td>
<td>Branch on overflow</td>
</tr>
<tr>
<td>ba</td>
<td>Branch always</td>
</tr>
</tbody>
</table>
ARC Assembly Language Format

- The ARC assembly language format is the same as the SPARC assembly language format.

```
lab_1: addcc %r1, %r2, %r3  ! Sample assembly code
```
### ARC User-Visible Registers

| Register 00 | %r0  [= 0] | Register 11 | %r11 |
| Register 01 | %r1     | Register 12 | %r12 |
| Register 02 | %r2     | Register 13 | %r13 |
| Register 03 | %r3     | Register 14 | %r14 [%sp] |
| Register 04 | %r4     | Register 15 | %r15 [link] |
| Register 05 | %r5     | Register 16 | %r16 |
| Register 06 | %r6     | Register 17 | %r17 |
| Register 07 | %r7     | Register 18 | %r18 |
| Register 08 | %r8     | Register 19 | %r19 |
| Register 09 | %r9     | Register 20 | %r20 |
| Register 10 | %r10    | Register 21 | %r21 |

| PSR | %psr |
| PC  | %pc  |

- 32 bits
## ARC Instruction and PSR Formats

### SETHI Format

<table>
<thead>
<tr>
<th>op</th>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00</th>
</tr>
</thead>
<tbody>
<tr>
<td>rd</td>
<td>op2</td>
</tr>
</tbody>
</table>

### Branch Format

<table>
<thead>
<tr>
<th>0 0 0</th>
<th>cond</th>
<th>op2</th>
<th>disp22</th>
</tr>
</thead>
</table>

### CALL format

| 0 1 | disp3 |                                                                                                                   |

### Arithmetic Formats

<table>
<thead>
<tr>
<th>1 0</th>
<th>rd</th>
<th>op3</th>
<th>rs1</th>
<th>0 0 0 0 0 0 0 0 0 0 0 0 rs2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0</td>
<td>rd</td>
<td>op3</td>
<td>rs1</td>
<td>1 simm13</td>
</tr>
</tbody>
</table>

### Memory Formats

<table>
<thead>
<tr>
<th>1 1</th>
<th>rd</th>
<th>op3</th>
<th>rs1</th>
<th>0 0 0 0 0 0 0 0 0 0 0 0 rs2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1</td>
<td>rd</td>
<td>op3</td>
<td>rs1</td>
<td>1 simm13</td>
</tr>
</tbody>
</table>

### PSR

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00</th>
</tr>
</thead>
<tbody>
<tr>
<td>n z v c</td>
</tr>
</tbody>
</table>

### Instructions

- **SETHI/Branch**: 00
  - 010: branch
  - 100: sethi

- **CALL**: 01
  - 10: CALL
  - 11: Arithmetic

- **Memory**: 11
  - 0: SETHI/Branch
  - 1: CALL
  - 10: Arithmetic
  - 11: Memory

### Instructions (op=10)

- addcc
- andcc
- orcc
- orncc
- srl
- jmpl

### Instructions (op=11)

- 000000: ld
- 000100: st

### Conditions

- 0001 be
- 0101 bcs
- 0110 bneg
- 0111 bvs
- 1000 ba
## ARC Data Formats

### Signed Formats
- **Signed Integer Byte**
  - 8 bits: 7 6 0
- **Signed Integer Halfword**
  - 16 bits: 15 14 0
- **Signed Integer Word**
  - 32 bits: 31 30 0
- **Signed Integer Double**
  - 64 bits: 63 62 32 31 0

### Unsigned Formats
- **Unsigned Integer Byte**
  - 8 bits: 7 0
- **Unsigned Integer Halfword**
  - 16 bits: 15 0
- **Unsigned Integer Word**
  - 32 bits: 31 0
- **Tagged Word**
  - 32 bits: 31 2 1 0
- **Unsigned Integer Double**
  - 64 bits: 63 32 31 0

### Floating Point Formats
- **Floating Point Single**
  - 32 bits: 31 30 23 22 0
- **Floating Point Double**
  - 64 bits: 63 62 52 51 32 31 0
- **Floating Point Quad**
  - 128 bits: 127 126 112 113 96 95 64 63 32 31 0
## ARC Pseudo-Ops

<table>
<thead>
<tr>
<th>Pseudo-Op</th>
<th>Usage</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>.equ</td>
<td>X .equ #10</td>
<td>Treat symbol (X) as ((10)_{16})</td>
</tr>
<tr>
<td>.begin</td>
<td>.begin</td>
<td>Start assembling</td>
</tr>
<tr>
<td>.end</td>
<td>.end</td>
<td>Stop assembling</td>
</tr>
<tr>
<td>.org</td>
<td>.org 2048</td>
<td>Change location counter to 2048</td>
</tr>
<tr>
<td>.dwb</td>
<td>.dwb 25</td>
<td>Reserve a block of 25 words</td>
</tr>
<tr>
<td>.global</td>
<td>.global Y</td>
<td>(Y) is used in another module</td>
</tr>
<tr>
<td>.extern</td>
<td>.extern Z</td>
<td>(Z) is defined in another module</td>
</tr>
<tr>
<td>.macro</td>
<td>.macro M a, b, ...</td>
<td>Define macro (M) with formal parameters (a, b, ...)</td>
</tr>
<tr>
<td>.endmacro</td>
<td>.endmacro</td>
<td>End of macro definition</td>
</tr>
<tr>
<td>.if</td>
<td>.if &lt;cond&gt;</td>
<td>Assemble if (&lt;\text{cond}) is true</td>
</tr>
<tr>
<td>.endif</td>
<td>.endif</td>
<td>End of .if construct</td>
</tr>
</tbody>
</table>

- Pseudo-ops are instructions to the assembler. They are not part of the ISA.
ARC Example Program

- An ARC assembly language program adds two integers:

```assembly
! This program adds two numbers

.org 2048
.progl: ld [x], %r1 ! Load x into %r1
       ld [y], %r2 ! Load y into %r2
       addcc %r1, %r2, %r3 ! %r3 ← %r1 + %r2
       st %r3, [z] ! Store %r3 into z
       jmpl %r15 + 4, %r0 ! Return

x: 15
y: 9
z: 0

.end
```

© 1999 M. Murdocca and V. Heuring
A More Complex Example Program

- An ARC program sums five integers.

```assembly
; This program sums LENGTH numbers
; Register usage:
; %r1 - Length of array a
; %r2 - Starting address of array a
; %r3 - The partial sum
; %r4 - Pointer into array a
; %r5 - Holds an element of a

.begin    ; Start assembling
.org 2048 ; Start program at 2048
.a_start  .equ 3000  ; Address of array a

ld [length], %r1 ! %r1 ← length of array a
ld [address],%r2 ! %r2 ← address of a
andcc %r3, %r0, %r3 ! %r3 ← 0

loop:    andcc %r1, %r1, %r0 ! Test # remaining elements
be done  ! Finished when length=0
addcc %r1, -4, %r1 ! Decrement array length
addcc %r1, %r2, %r4 ! Address of next element
ld %r4, %r5 ! %r5 ← Memory[%r4]
addcc %r3, %r5, %r3 ! Sum new element into r3
ba loop  ! Repeat loop.

done:    jmpl %r15 + 4, %r0 ! Return to calling routine

length:  20 ! 5 numbers (20 bytes) in a
address: a_start
.org a_start ! Start of array a

a:       25 ! length/4 values follow
         -10
         33
         -5
         7

.end      ! Stop assembling
```
One, Two, Three-Address Machines

• Consider how the C expression \( A = B \times C + D \) might be evaluated by each of the one, two, and three-address instruction types.

• Assumptions: Addresses and data words are two bytes in size. Opcodes are 1 byte in size. Operands are moved to and from memory one word (two bytes) at a time.

• Three-Address Instructions: In a three-address instruction, the expression \( A = B \times C + D \) might be coded as:

\[
\text{mult } B, C, A \\
\text{add } D, A, A
\]

which means multiply \( B \) by \( C \) and store the result at \( A \). (The \text{mult} and \text{add} operations are generic; they are not ARC instructions.) Then, add \( D \) to \( A \) and store the result at address \( A \). The program size is \( 7 \times 2 = 14 \) bytes. Memory traffic is \( 16 + 2 \times (2 \times 3) = 28 \) bytes.
One, Two, Three-Address Machines

• Two Address Instructions: In a two-address instruction, one of the operands is overwritten by the result. Here, the code for the expression $A = B \times C + D$ is:

\[
\text{load B, A} \\
\text{mult C, A} \\
\text{add D, A}
\]

The program size is now $3 \times (1 + 2 \times 2)$ or 15 bytes. Memory traffic is $15 + 2 \times 2 + 2 \times 2 \times 3$ or 31 bytes.
One, Two, Three-Address Machines

- **One Address (Accumulator) Instructions**: A one-address instruction employs a single arithmetic register in the CPU, known as the *accumulator*. The code for the expression $A = B \times C + D$ is now:

```
load   B
mult   C
add    D
store  A
```

The `load` instruction loads $B$ into the accumulator, `mult` multiplies $C$ by the accumulator and stores the result in the accumulator, and `add` does the corresponding addition. The `store` instruction stores the accumulator in $A$. The program size is now $2 \times 2 \times 4$ or 16 bytes, and memory traffic is $16 + 4 \times 2$ or 24 bytes.
### Addressing Modes

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Syntax</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>#K</td>
<td>K</td>
</tr>
<tr>
<td>Direct</td>
<td>K</td>
<td>M[K]</td>
</tr>
<tr>
<td>Indirect</td>
<td>(K)</td>
<td>M[M[K]]</td>
</tr>
<tr>
<td>Register</td>
<td>(Rn)</td>
<td>M[Rn]</td>
</tr>
<tr>
<td>Register Indexed</td>
<td>(Rm + Rn)</td>
<td>M[Rm + Rn]</td>
</tr>
<tr>
<td>Register Based</td>
<td>(Rm + X)</td>
<td>M[Rm + X]</td>
</tr>
<tr>
<td>Register Based Indexed</td>
<td>(Rm + Rn + X)</td>
<td>M[Rm + Rn + X]</td>
</tr>
</tbody>
</table>

- Four ways of computing the address of a value in memory: (1) a constant value known at assembly time, (2) the contents of a register, (3) the sum of two registers, (4) the sum of a register and a constant. The table gives names to these and other addressing modes.
Subroutine Linkage – Registers

- Subroutine linkage with registers passes parameters in registers.

```
! Calling routine
  ...
  ld [x], %r1
  ld [y], %r2
  call add_1
  st %r3, [z]
  ...
  x:  53
  y:  10
  z:  0

! Called routine
  ! %r3 ← %r1 + %r2
  add_1: addcc %r1, %r2, %r3
          jmpl %r15 + 4, %r0
```

Subroutine Linkage – Data Link Area

- Subroutine linkage with a data link area passes parameters in a separate area in memory. The address of the memory area is passed in a register (%r5 here).

```
Calling routine
.
.
st  %r1, [x]
st  %r2, [x+4]
sethi x, %r5
srl  %r5, 10, %r5
call  add_2
ld  [x+8], %r3
.
.
! Data link area
x: .dwb 3

Called routine
!
x[2] ← x[0] + x[1]
!
add_2: ld  %r5, %r8
         ld  %r5 + 4, %r9
         addcc %r8, %r9, %r10
         st  %r10, %r5 + 8
         jmpl %r15 + 4, %r0
```
### Subroutine Linkage – Stack

- Subroutine linkage with a stack passes parameters on a stack.

---

<table>
<thead>
<tr>
<th>Calling routine</th>
<th>Called routine</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>! Arguments are on stack.</td>
</tr>
<tr>
<td>.</td>
<td>! %sp[0] ← %sp[0] + %sp[4]</td>
</tr>
<tr>
<td>.</td>
<td>%sp .equ %r14</td>
</tr>
<tr>
<td>%sp .equ %r14</td>
<td>%sp .equ %r14</td>
</tr>
<tr>
<td>addcc %sp, -4, %sp</td>
<td>add_3: ld %sp, %r8</td>
</tr>
<tr>
<td>st %r1, %sp</td>
<td>addcc %sp, 4, %sp</td>
</tr>
<tr>
<td>addcc %sp, -4, %sp</td>
<td>ld %sp, %r9</td>
</tr>
<tr>
<td>st %r2, %sp</td>
<td>addcc %r8, %r9, %r10</td>
</tr>
<tr>
<td>call add_3</td>
<td>st %r10, %sp</td>
</tr>
<tr>
<td>ld %sp, %r3</td>
<td>jmpl %r15 + 4, %r0</td>
</tr>
<tr>
<td>addcc %sp, 4, %sp</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
</tbody>
</table>
Stack Linkage Example

- A C program illustrates nested function calls.

```c
/* C program showing nested subroutine calls */

main()
{
    int w, z; /* Local variables */
    w = func_1(1,2); /* Call subroutine func_1 */
    z = func_2(10); /* Call subroutine func_2 */
}

/* End of main routine */

int func_1(x,y) /* Compute x * x + y */
    int x, y; /* Parameters passed to func_1 */
{
    int i, j; /* Local variables */
    i = x * x;
    j = i + y;
    return(j); /* Return j to calling routine */
}

int func_2(a) /* Compute a * a + a + 5 */
    int a; /* Parameter passed to func_2 */
{
    int m, n; /* Local variables */
    n = a + 5;
    m = func_1(a,n);
    return(m); /* Return m to calling routine */
}
```
Stack Linkage Example (cont')

- (a-f) Stack behavior during execution of the program shown in previous slide.

  (a) Initial configuration. \( w \) and \( z \) are already on the stack. (Line 00 of program.)

  (b) Calling routine pushes arguments onto stack, prior to \texttt{func_1} call. (Line 03 of program.)

  (c) After the call, called routine saves PC of calling routine (\%r15) onto stack. (Line 06 of program.)

  (d) Stack space is reserved for \texttt{func_1} local variables \( i \) and \( j \). (Line 09 of program.)

  (e) Return value from \texttt{func_1} is placed on stack, just prior to return. (Line 12 of program.)

  (f) Calling routine pops \texttt{func_1} return value from stack. (Line 03 of program.)
Stack Linkage Example (cont’)

- (g-k) Stack behavior during execution of the C program shown previously.

- (g) A stack frame is created for `func_2` as a result of function call at line 04 of program.

- (h) A stack frame is created for `func_1` as a result of function call at line 19 of program.

- (i) `func_1` places return value on stack. (Line 12 of program.)

- (j) `func_2` places return value on stack. (Line 20 of program.)

- (k) Program finishes. Stack is restored to its initial configuration. (Lines 04 and 05 of program.)
Input and Output for the ISA

- Memory map for the ARC, showing memory mapped I/O.

- Stack pointer
- I/O space
- Add-in video memory #1
- Add-in video memory #2
- Unused
- Working Memory
- Top of stack
  - System Stack
  - Bottom of stack
- Screen Flash
  - FFFFFEC<sub>16</sub>
- Touchscreen x
  - FFFFF0<sub>16</sub>
- Touchscreen y
  - FFFFF4<sub>16</sub>

Address and Data:
- 32 bits
- Reserved for built-in bootstrap and graphics routines
- 2<sup>16</sup> → Add-in video memory #1
- 2<sup>17</sup> → Add-in video memory #2
- 2<sup>19</sup> → Unused
- 2<sup>22</sup> → Working Memory
- 2<sup>23</sup> – 4 → Screen Flash
- 2<sup>24</sup> – 4 → Touchscreen y
- 2<sup>24</sup> – 4 → Touchscreen x
- byte
Touchscreen I/O Device

- A user selecting an object on a touchscreen:
Flowchart for I/O Device

- Flowchart illustrating the control structure of a program that tracks a touchscreen.
Java Virtual Machine Architecture

Java Stack
- Stack frame
  - Java Stack
  - Operand stack
  - Local variables
  - Constant pool
  - State variables

Byte Codes

Stack top index
Thread state
Current method pointer
Current method’s class pointer
Current method’s constant pool pointer
Stack frame pointer
Program counter

Java Execution Engine

Control
// This is file add.java

public class add {
    public static void main(String args[]) {
        int x=15, y=9, z=0;
        z = x + y;
    }
}

// This is file compiled Class File

0000 cafe babe 0003 002d 0012 0700 0e07 0010
0010 0a00 0200 040c 0007 0005 0100 0328 2956
0020 0100 1628 5b4c 6a61 7661 2f6c 616e 672f
0030 5374 7269 6e67 3b29 5601 0006 3c69 6e69
0040 0000 0443 6f64 6501 000d 436f 6e73 7461 6e74 5661 6c75 6501 000a 4578 6365 7074 696f 6e73 0100 0f4c 696e 654e 756d 6265 7462 6c65 7301 000a 536f 7572 6365 4669 6c65 0100 0361 6464 0100 081c 0008 0006 000c 0002 0001 0007 0005 0001 0110 0008 0000 01d0 0001 0001 0000 0005 2ab7 0120 0003 b100 0000 0100 0b00 0000 0600 0100 0130 0000 0100 0100 0d00 0000 0200 0f00

// This is file add.java
public class add {
    public static void main(String args[]) {
        int x=15, y=9, z=0;
        z = x + y;
    }
}
A Java Class File

18 items in constant pool

Tag = 7 (Class)
Name index = 14
Tag = 7 (Class)
Name index = 16

Tag = 10 (Methodref)
Class index = 2
Name and type index = 4

Tag = 1 (Utf)
Length = 22 bytes
"([Ljava/lang/
String;)V"

Tag = 1 (Utf)
Length = 6 bytes
"<ini"

Tag = 1 (Utf)
Length = 4 bytes
"t>"

Tag = 1 (Utf)
Length = 13 bytes
"Cons"

Tag = 1 (Utf)
Length = 10 bytes
"Exce"

Tag = 1 (Utf)
Length = 15 bytes
"ptions"

Tag = 1 (Utf)
Length = 14 bytes
"berTable"

Tag = 1 (Utf)
Length = 10 bytes
"Variables"

Tag = 1 (Utf)
Length = 3 bytes
"ceFile"

Tag = 1 (Utf)
Length = 8 bytes
"a"
A Java Class File (Cont')

```
00a0 6464 2e6a 6176 6101 0010 6a61 7661 2f6c
00b0 616e 672f 4f62 6a65 6374 0100 046d 6169
00c0 6e00 2100 0010 0200 0000 0000 0200 0900
00d0 1100 0600 0100 0800 0000 2d00 0200 0400
00e0 0000 0d10 0f3c 1009 3d03 3e1b 1c60 3eb1
```

```
00f0 0000 0001 000b 0000 000e 0000 0004
0100 0008 0006 000c 0002 0001 0007 0005 0001
0110 0000 0100 0000 0200 0f00
0120 0003 b100 0000 0100 0b00 0000 6000 0100
0130 0000 0100 0100 0d00 0000 0200 0f00
```

```
"dd.java" Tag = 1 (Utf) Length = 16 bytes "java/l"

"ang/Object" Tag = 1 (Utf) Length = 4 bytes "mai"

Access flags: ACC_PUBLIC | ACC_SUPER
Superclass: java/lang/Object
This class: add
Interface count
Fields count
Methods count

Access flags: ACC_PUBLIC | ACC_SUPER | ACC_STATIC

Access flags: ACC_PUBLIC
Superclass: java/lang/Object
This class: add
Interface count
Fields count
Methods count

Name index "<init>">
Type index "[Ljava/lang/String;V"

Attribute name index: "Code"
Bytes count = 45 Max stack = 2 Max locals = 4

Attribute name index: "Code"
Max stack = 2 Code count = 5

Attribute name index: "SourceFile"
Source file index: "add.java"

```
Principles of Computer Architecture by M. Murdocca and V. Heuring © 1999 M. Murdocca and V. Heuring
Byte Code for Java Program
• Disassembled byte code for previous Java program.

<table>
<thead>
<tr>
<th>Location</th>
<th>Code</th>
<th>Mnemonic</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00e3</td>
<td>0x10</td>
<td>bipush</td>
<td>Push next byte onto stack</td>
</tr>
<tr>
<td>0x00e4</td>
<td>0x0f</td>
<td>15</td>
<td>Argument to bipush</td>
</tr>
<tr>
<td>0x00e5</td>
<td>0x3c</td>
<td>istore_1</td>
<td>Pop stack to local variable 1</td>
</tr>
<tr>
<td>0x00e6</td>
<td>0x10</td>
<td>bipush</td>
<td>Push next byte onto stack</td>
</tr>
<tr>
<td>0x00e7</td>
<td>0x09</td>
<td>9</td>
<td>Argument to bipush</td>
</tr>
<tr>
<td>0x00e8</td>
<td>0x3d</td>
<td>istore_2</td>
<td>Pop stack to local variable 2</td>
</tr>
<tr>
<td>0x00e9</td>
<td>0x03</td>
<td>iconst_0</td>
<td>Push 0 onto stack</td>
</tr>
<tr>
<td>0x00ea</td>
<td>0x3e</td>
<td>istore_3</td>
<td>Pop stack to local variable 3</td>
</tr>
<tr>
<td>0x00eb</td>
<td>0x1b</td>
<td>iload_1</td>
<td>Push local variable 1 onto stack</td>
</tr>
<tr>
<td>0x00ec</td>
<td>0x1c</td>
<td>iload_2</td>
<td>Push local variable 2 onto stack</td>
</tr>
<tr>
<td>0x00ed</td>
<td>0x60</td>
<td>iadd</td>
<td>Add top two stack elements</td>
</tr>
<tr>
<td>0x00ee</td>
<td>0x3e</td>
<td>istore_3</td>
<td>Pop stack to local variable 3</td>
</tr>
<tr>
<td>0x00ef</td>
<td>0xb1</td>
<td>return</td>
<td>Return</td>
</tr>
</tbody>
</table>