

# **CPSC 352**

## *Computer Organization*

# **Chapter 4: The Instruction Set Architecture**

# Chapter Contents

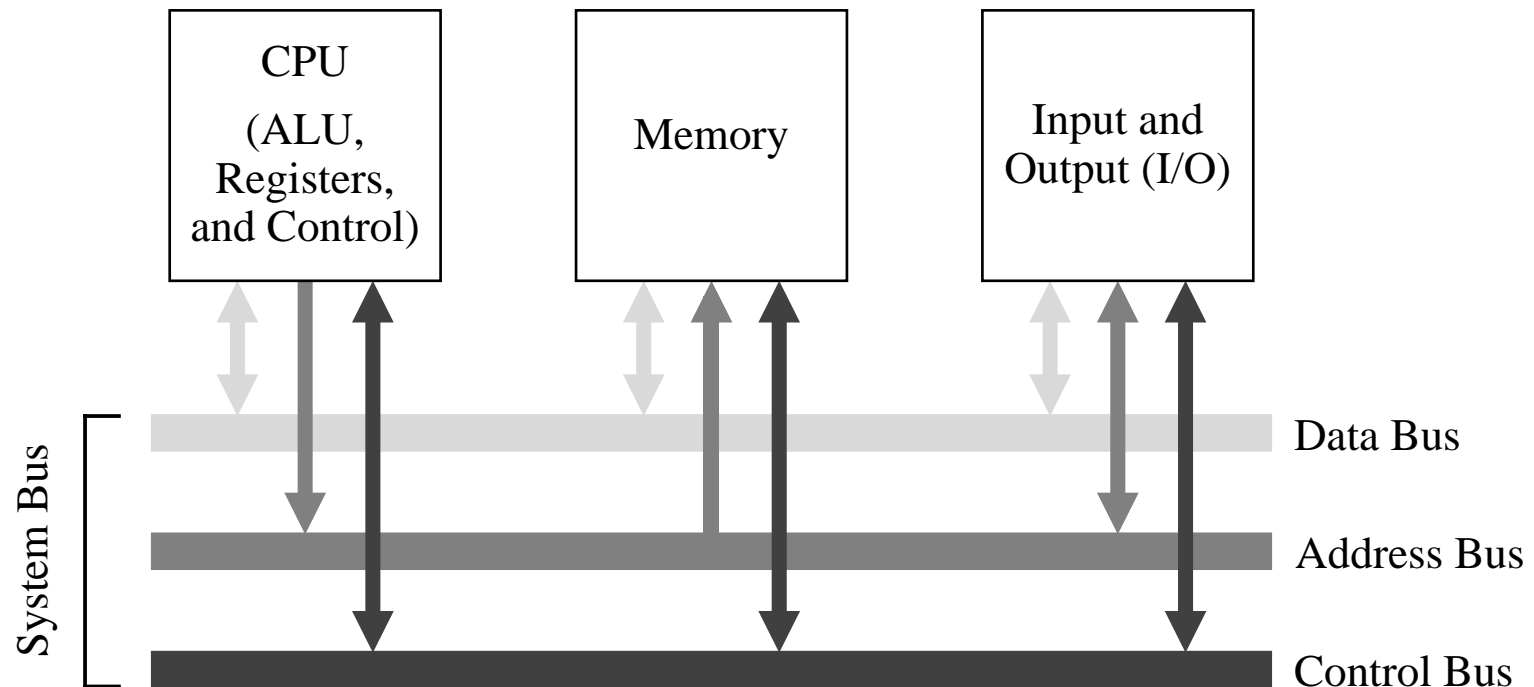
- 4.1 Hardware Components of the Instruction Set Architecture**
- 4.2 ARC, A RISC Computer**
- 4.3 Pseudo-Ops**
- 4.4 Examples of Assembly Language Programs**
- 4.5 Accessing Data in Memory—Addressing Modes**
- 4.6 Subroutine Linkage and Stacks**
- 4.7 Input and Output in Assembly Language**
- 4.8 Case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



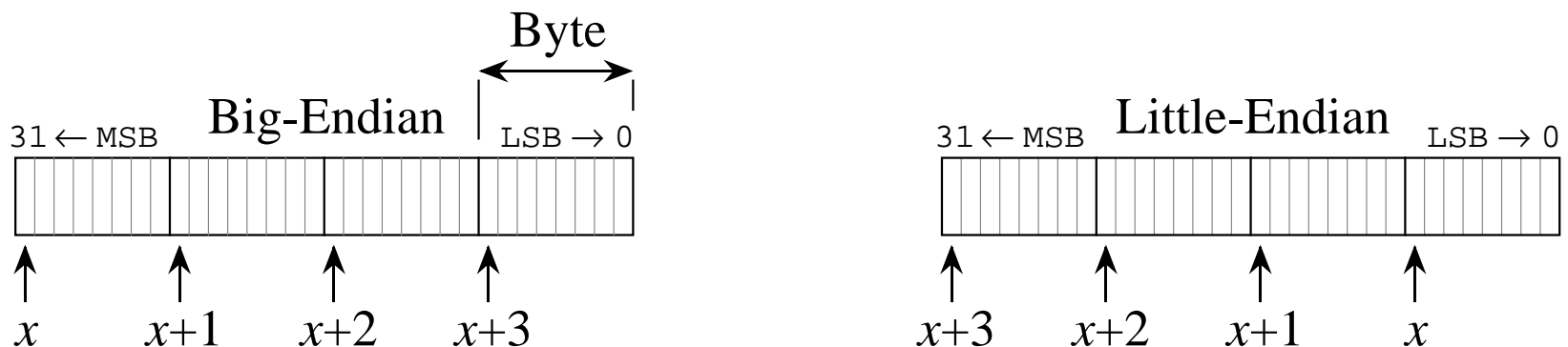
# Common Sizes for Data Types

- 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:

Bit	0
Nibble	0110
Byte	10110000
16-bit word (halfword)	11001001 01000110
32-bit word	10110100 00110101 10011001 01011000
64-bit word (double)	01011000 01010101 10110000 11110011 11001110 11101110 01111000 00110101
128-bit word (quad)	01011000 01010101 10110000 11110011 11001110 11101110 01111000 00110101 00001011 10100110 11110010 11100110 10100100 01000100 10100101 01010001

# Big-Endian and Little-Endian Formats

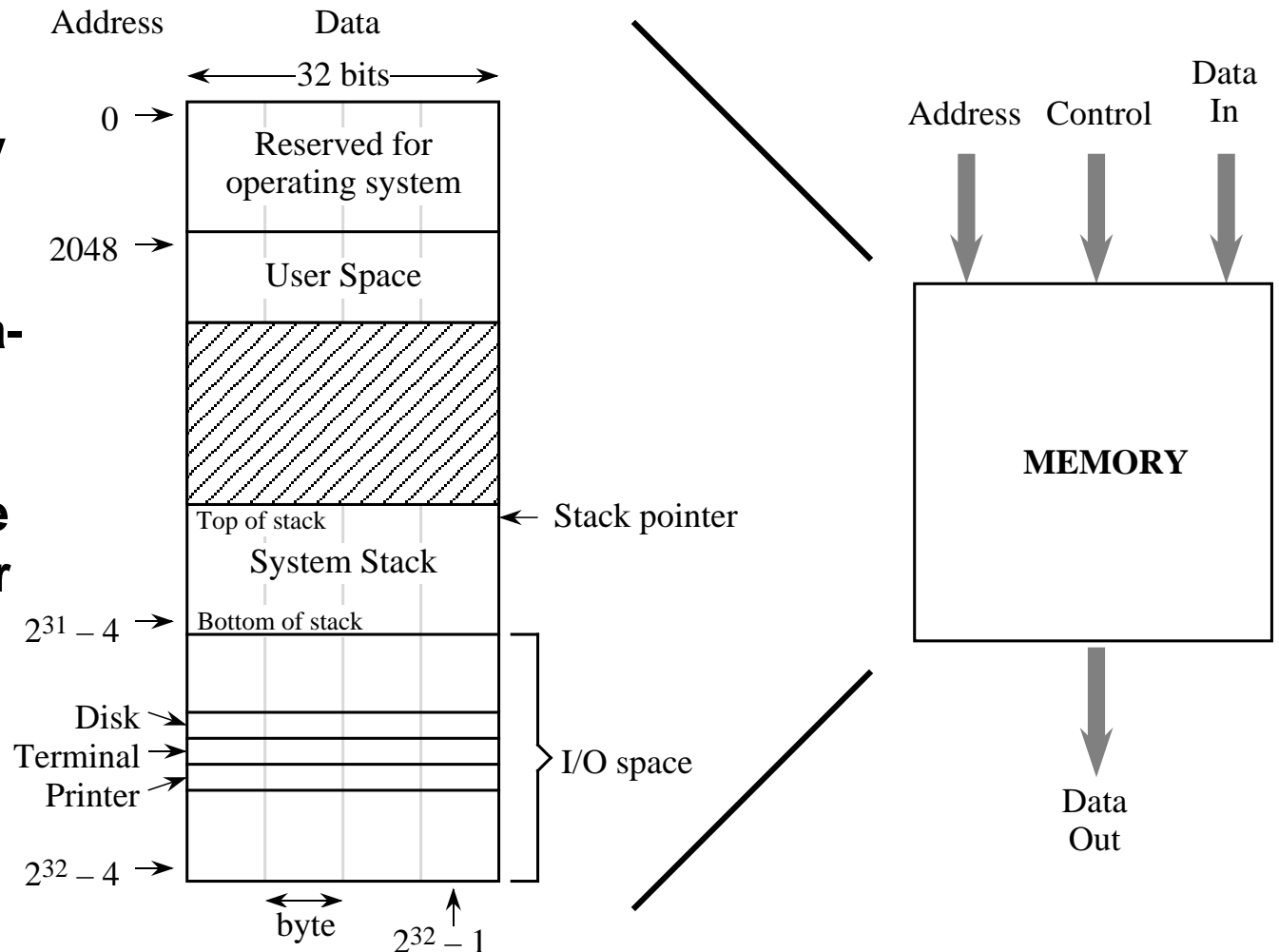
- In a byte-addressable machine, the smallest datum that can be referenced in memory is the byte. Multi-byte words are stored as a sequence of bytes, in which the address of the multi-byte word is the same as the byte of the word that has the lowest address.
- When multi-byte words are used, two choices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

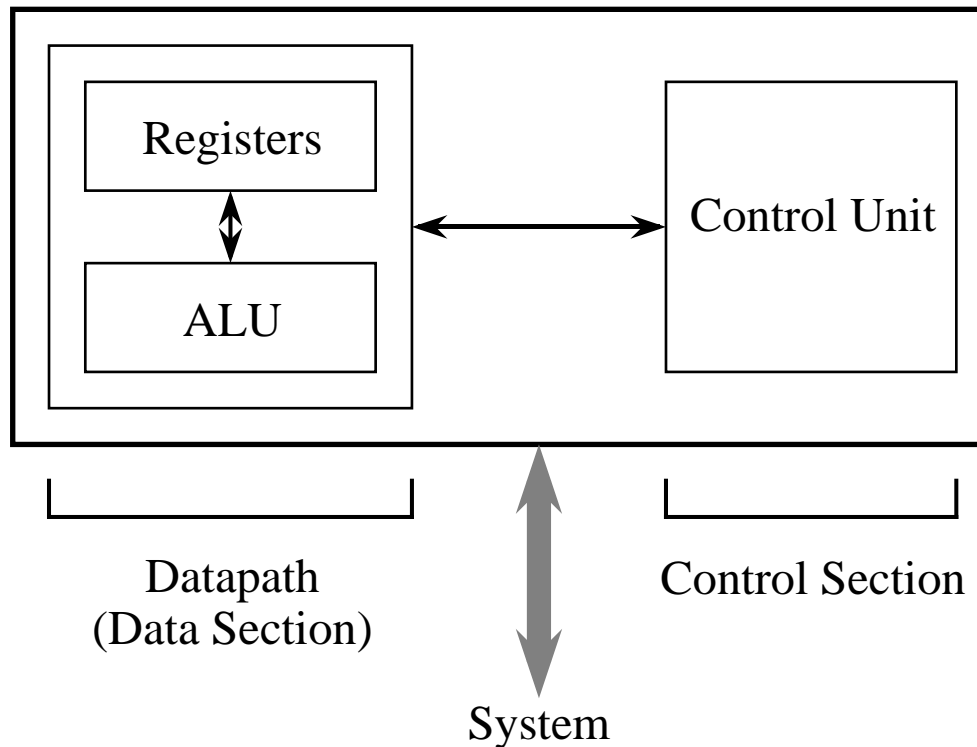
# Memory Map for the ARC

- **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*.



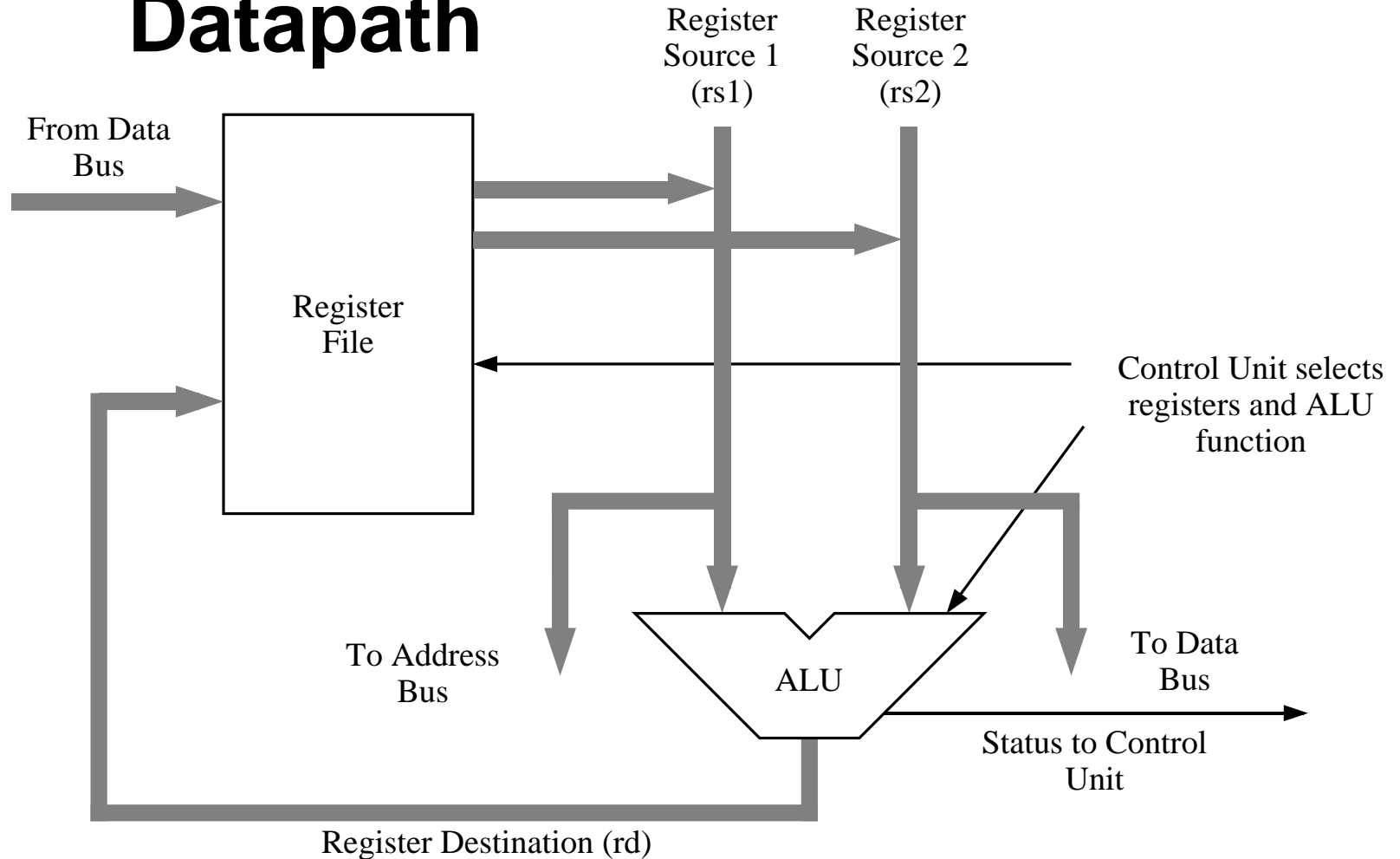


# The Fetch-Execute Cycle

- 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)*.

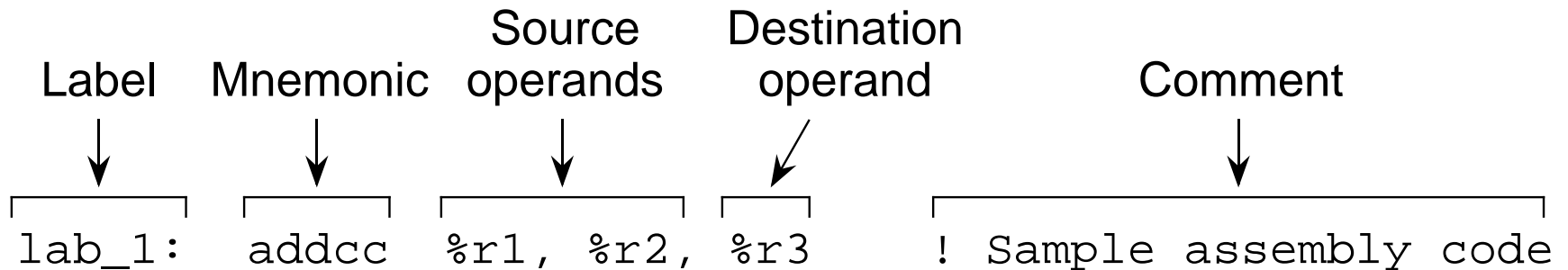
# The ARC ISA

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

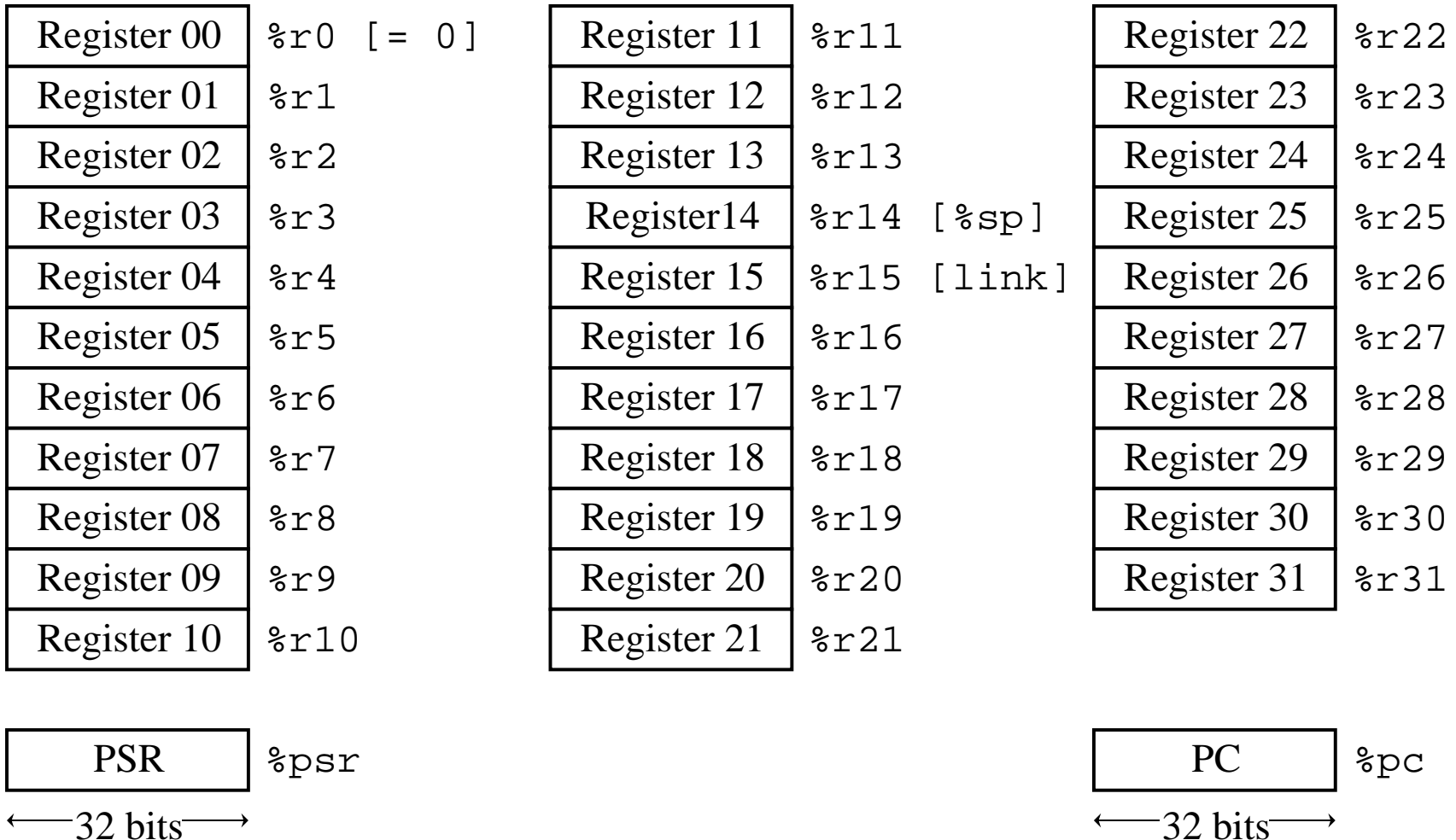
	Mnemonic	Meaning
Memory	<b>ld</b>	Load a register from memory
	<b>st</b>	Store a register into memory
Logic	<b>sethi</b>	Load the 22 most significant bits of a register
	<b>andcc</b>	Bitwise logical AND
	<b>orcc</b>	Bitwise logical OR
	<b>orncc</b>	Bitwise logical NOR
	<b>srl</b>	Shift right (logical)
Arithmetic	<b>addcc</b>	Add
Control	<b>call</b>	Call subroutine
	<b>jmp1</b>	Jump and link (return from subroutine call)
	<b>be</b>	Branch if equal
	<b>bneg</b>	Branch if negative
	<b>bcs</b>	Branch on carry
	<b>bvs</b>	Branch on overflow
	<b>ba</b>	Branch always

# ARC Assembly Language Format

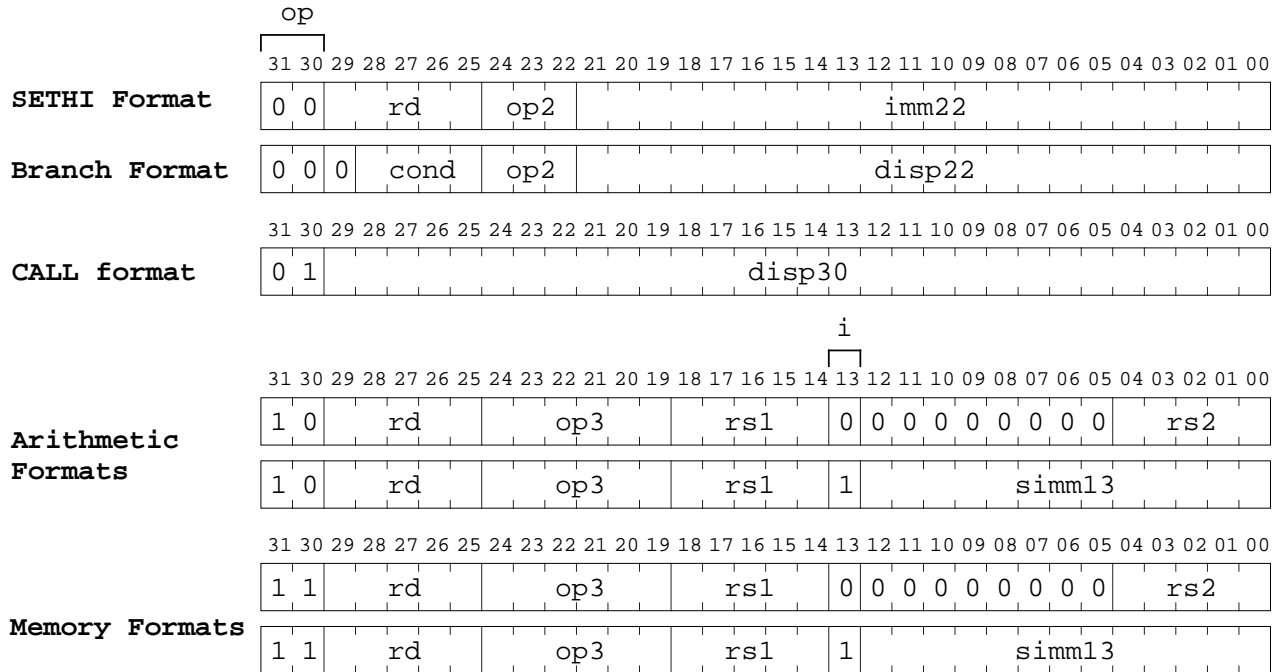
- The ARC assembly language format is the same as the SPARC assembly language format.



# ARC User-Visible Registers



# ARC Instruction and PSR Formats



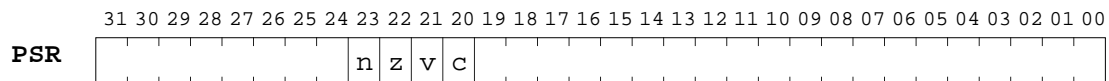
op	Format
00	SETHI/Branch
01	CALL
10	Arithmetic
11	Memory

op2	Inst.
010	branch
100	sethi

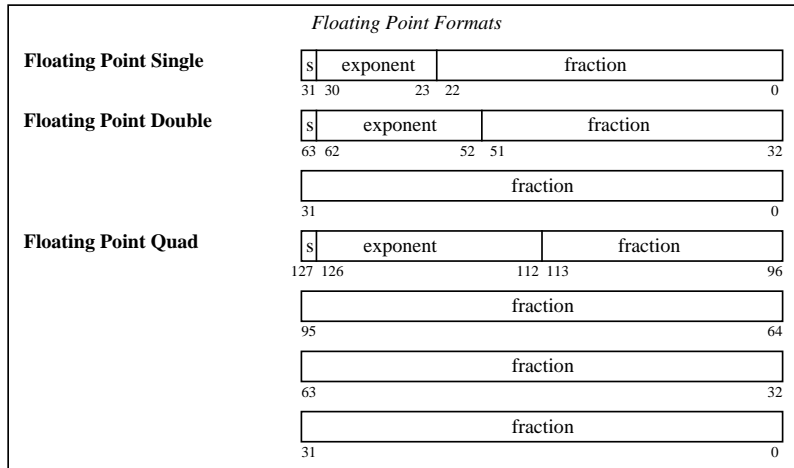
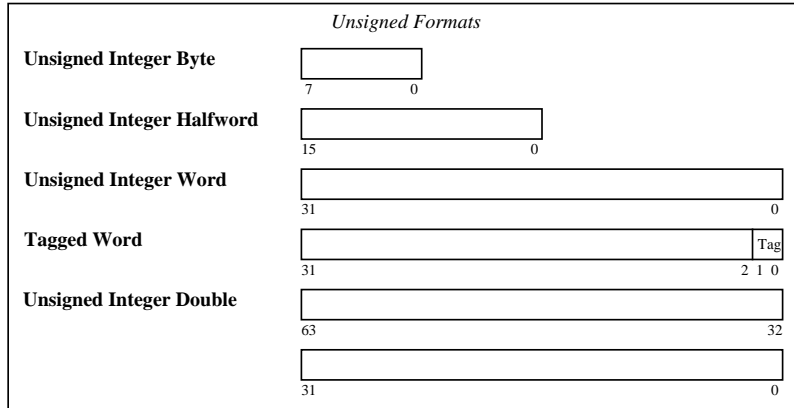
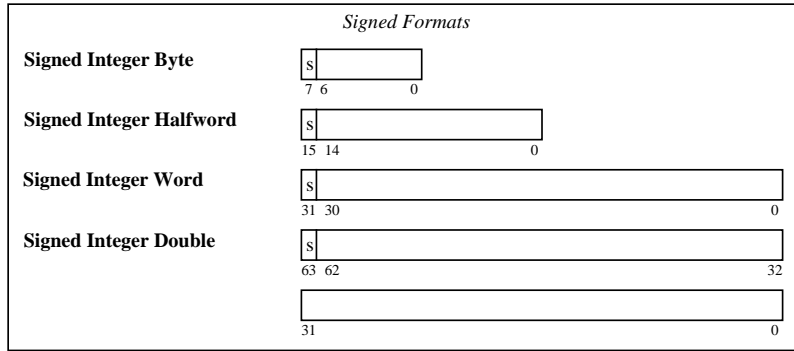
op3 (op=10)	
010000	addcc
010001	andcc
010010	orcc
010110	orncc
100110	srl
111000	jmp1

op3 (op=11)	
000000	ld
000100	st

cond	branch
0001	be
0101	bcs
0110	bneg
0111	bvs
1000	ba



# ARC Data Formats



# ARC Pseudo-Ops

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

- **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:

```
! This programs adds two numbers
    .begin
    .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
      jmp1  %r15 + 4, %r0    ! Return
x:    15
y:    9
z:    0
      .end
```

# A More Complex Example Program

- An ARC program sums five integers.

```

! 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  $A = B * C + D$  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  $A = B * C + D$  might be coded as:

```
mult  B, C, A
add   D, A, A
```

which means multiply B by C and store the result at A. (The `mult` and `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  $A = B * C + D$  is:

```
load  B, A
mult  C, A
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  $A = B * C + D$  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

Addressing Mode	Syntax	Meaning
Immediate	#K	K
Direct	K	M[K]
Indirect	(K)	M[M[K]]
Register	(R <sub>n</sub> )	M[R <sub>n</sub> ]
Register Indexed	(R <sub>m</sub> + R <sub>n</sub> )	M[R <sub>m</sub> + R <sub>n</sub> ]
Register Based	(R <sub>m</sub> + X)	M[R <sub>m</sub> + X]
Register Based Indexed	(R <sub>m</sub> + R <sub>n</sub> + X)	M[R <sub>m</sub> + R <sub>n</sub> + X]

- **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.

<pre>! Calling routine : ld    [x], %r1 ld    [y], %r2 call  add_1 st    %r3, [z] : x:    53 y:    10 z:    0</pre>	<pre>! Called routine ! %r3 ← %r1 + %r2  add_1: addcc  %r1, %r2, %r3       jmp1   %r15 + 4, %r0</pre>
---	---

# 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
        jmp1  %r15 + 4, %r0
```



# Subroutine Linkage – Stack

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

<pre>! Calling routine : %sp .equ %r14 addcc %sp, -4, %sp st    %r1, %sp addcc %sp, -4, %sp st    %r2, %sp call  add_3 ld    %sp, %r3 addcc %sp, 4, %sp :</pre>	<pre>! Called routine ! Arguments are on stack. ! %sp[0] ← %sp[0] + %sp[4] %sp .equ %r14 add_3: ld    %sp, %r8       addcc %sp, 4, %sp       ld    %sp, %r9       addcc %r8, %r9, %r10       st    %r10, %sp       jmp1  %r15 + 4, %r0</pre>
---	--

# Stack Linkage Example

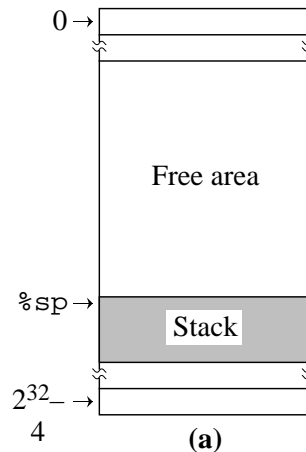
- A C program illustrates nested function calls.

```
Line No. /* C program showing nested subroutine calls */
00 main()
01 {
02     int w, z;          /* Local variables */
03     w = func_1(1,2);  /* Call subroutine func_1 */
04     z = func_2(10);  /* Call subroutine func_2 */
05 }                    /* End of main routine */

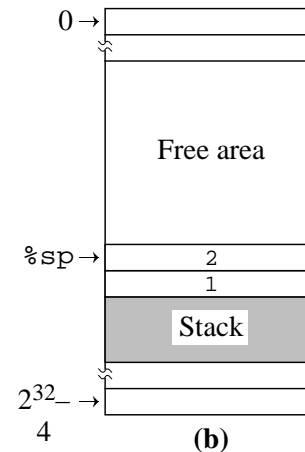
06 int func_1(x,y)      /* Compute x * x + y */
07 int x, y;           /* Parameters passed to func_1 */
08 {
09     int i, j;        /* Local variables */
10     i = x * x;
11     j = i + y;
12     return(j);      /* Return j to calling routine */
13 }

14 int func_2(a)        /* Compute a * a + a + 5 */
15 int a;              /* Parameter passed to func_2 */
16 {
17     int m, n;        /* Local variables */
18     n = a + 5;
19     m = func_1(a,n);
20     return(m);      /* Return m to calling routine */
21 }
```

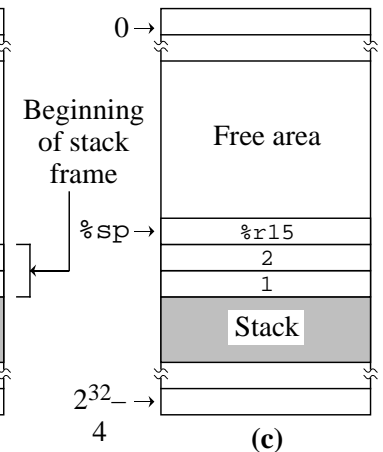
# Stack Linkage Example (cont')



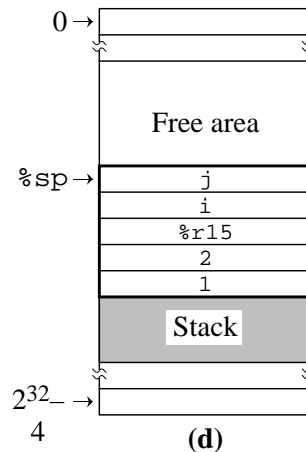
(a) Initial configuration.  $w$  and  $z$  are already on the stack. (Line 00 of program.)



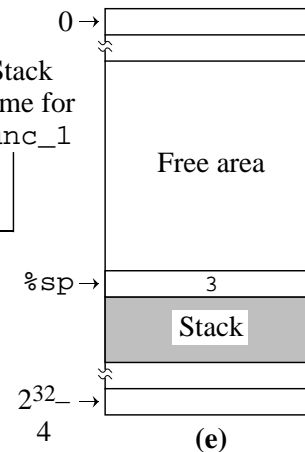
(b) Calling routine pushes arguments onto stack, prior to `func_1` call. (Line 03 of program.)



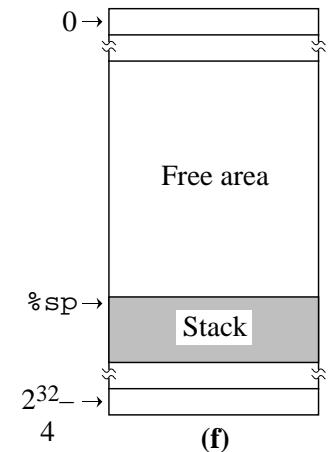
(c) After the call, called routine saves PC of calling routine ( $\%r15$ ) onto stack. (Line 06 of program.)



(d) Stack space is reserved for `func_1` local variables  $i$  and  $j$ . (Line 09 of program.)



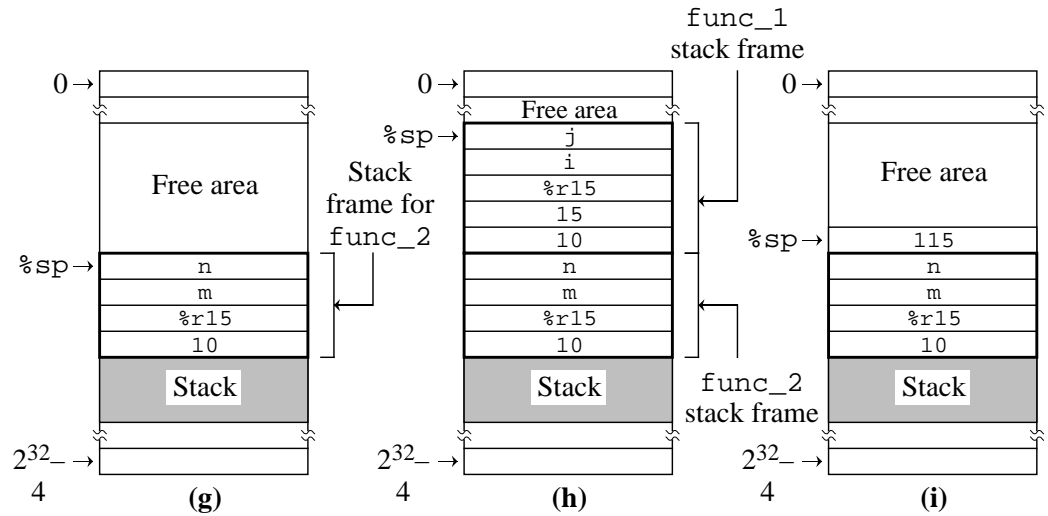
(e) Return value from `func_1` is placed on stack, just prior to return. (Line 12 of program.)



(f) Calling routine pops `func_1` return value from stack. (Line 03 of program.)

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

# Stack Linkage Example (cont')

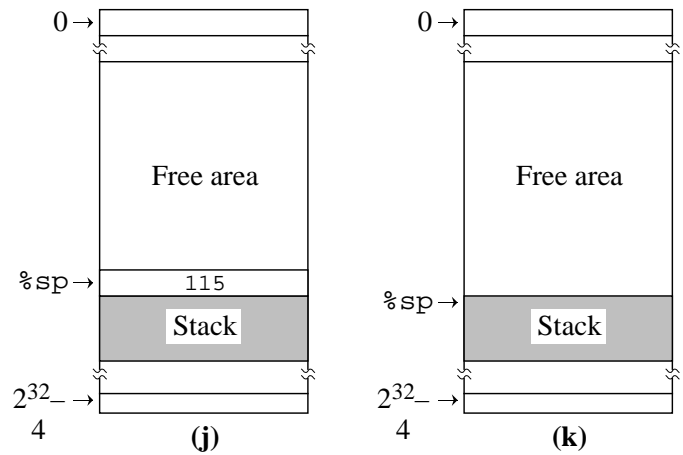


A stack frame is created for func\_2 as a result of function call at line 04 of program.

A stack frame is created for func\_1 as a result of function call at line 19 of program.

func\_1 places return value on stack. (Line 12 of program.)

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

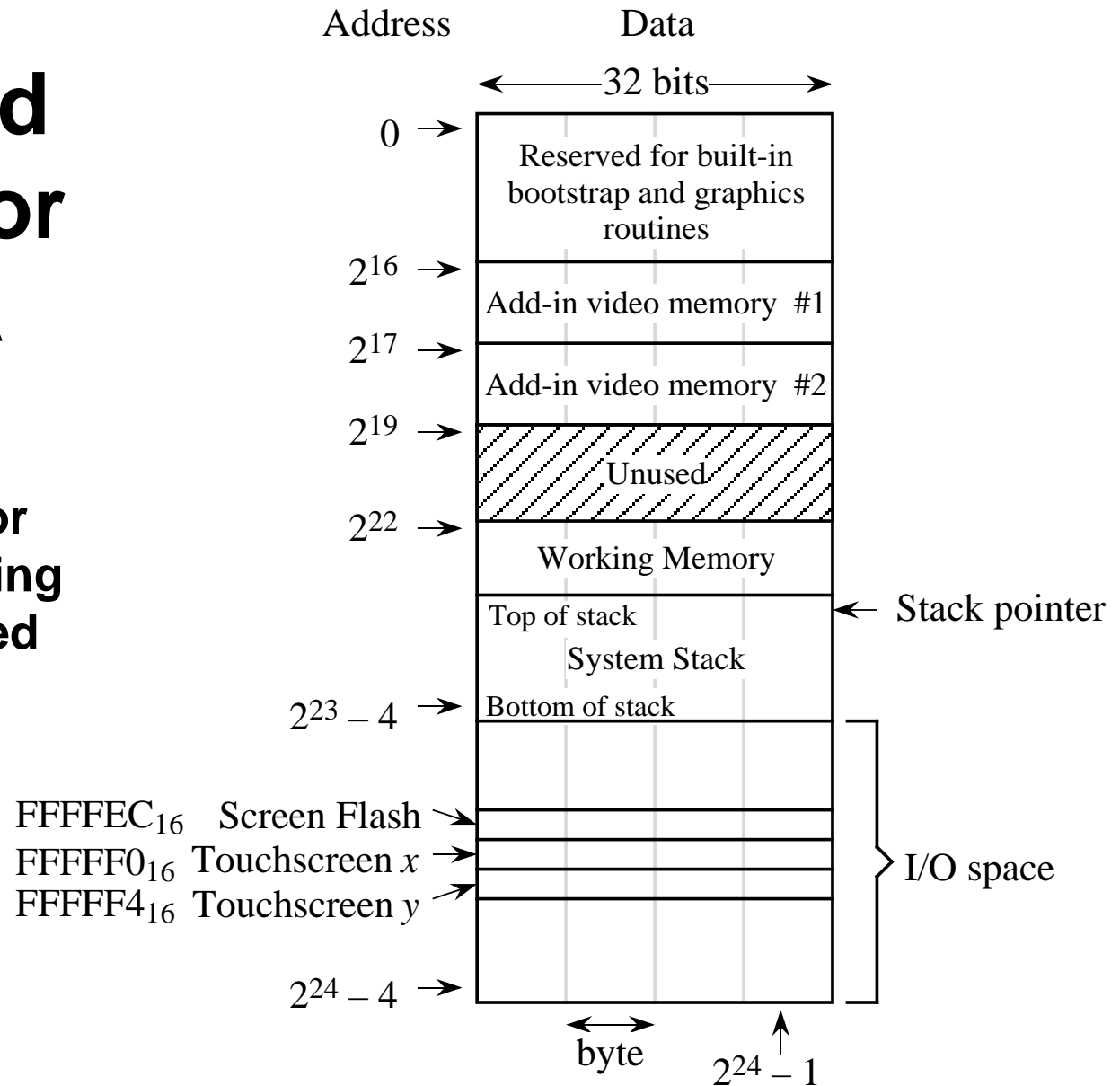


func\_2 places return value on stack. (Line 20 of program.)

Program finishes. Stack is restored to its initial configuration. (Lines 04 and 05 of program.)

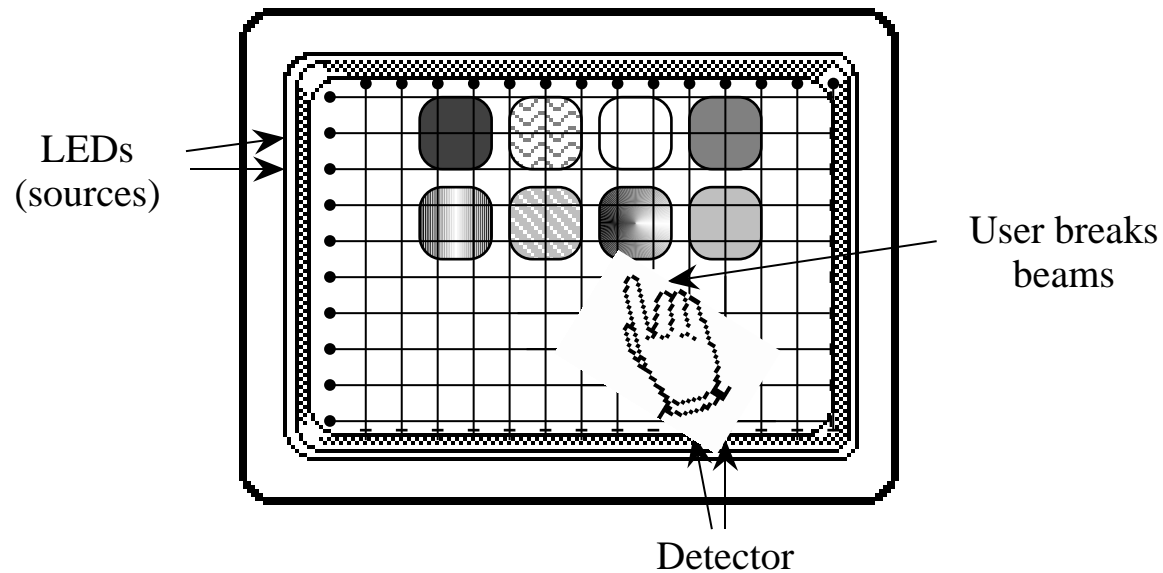
# Input and Output for the ISA

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



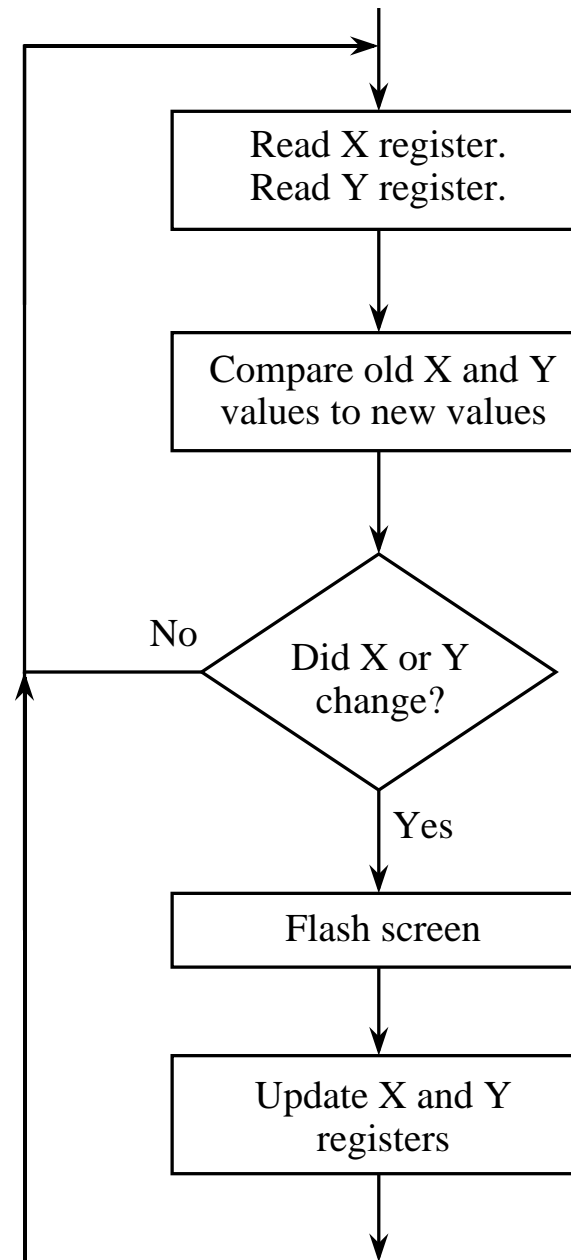
# Touchscreen I/O Device

- A user selecting an object on a touchscreen:

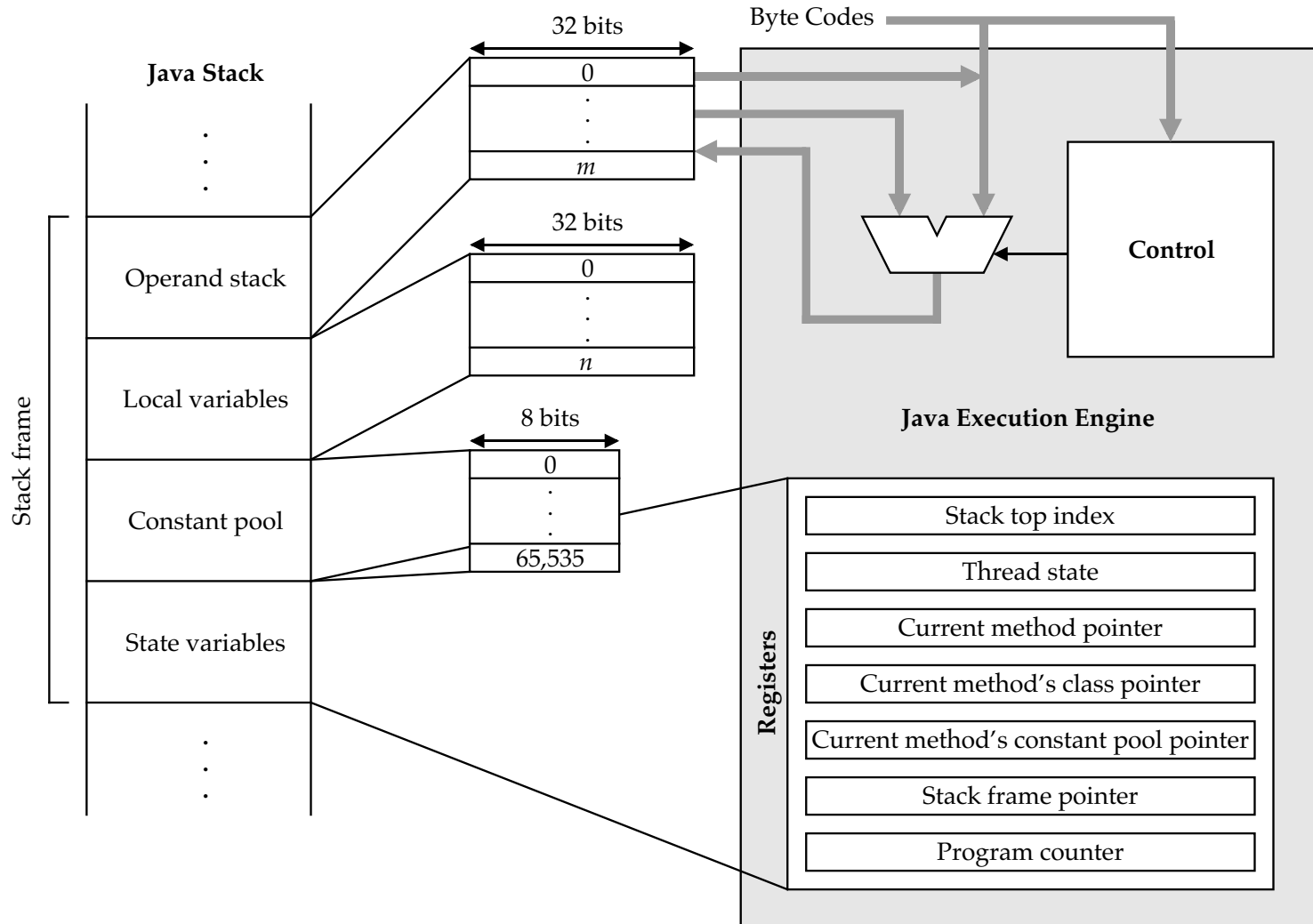


# Flowchart for I/O Device

- Flowchart illustrating the control structure of a program that tracks a touchscreen.



# Java Virtual Machine Architecture





# Java Program and Compiled Class File

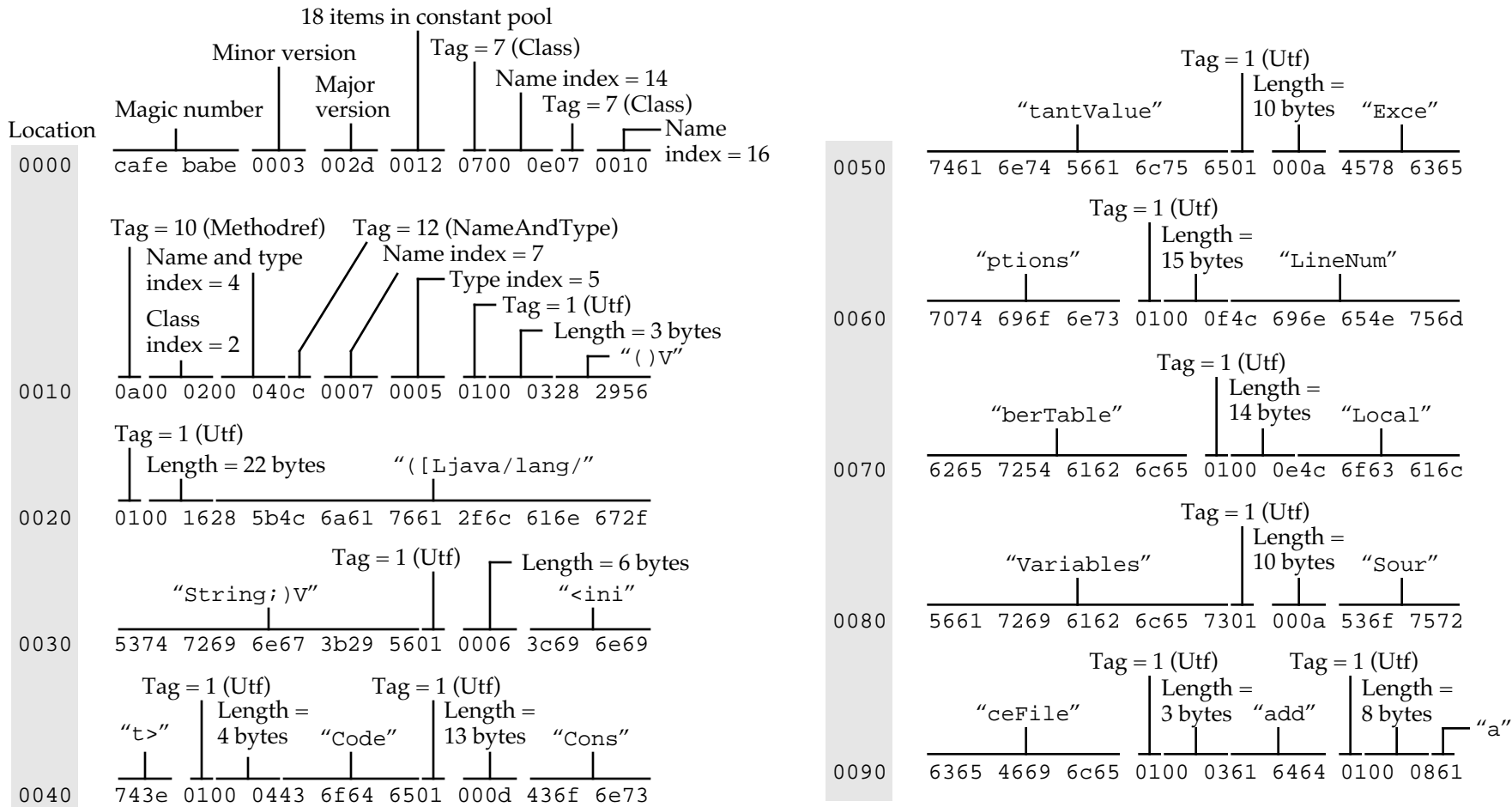
```

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

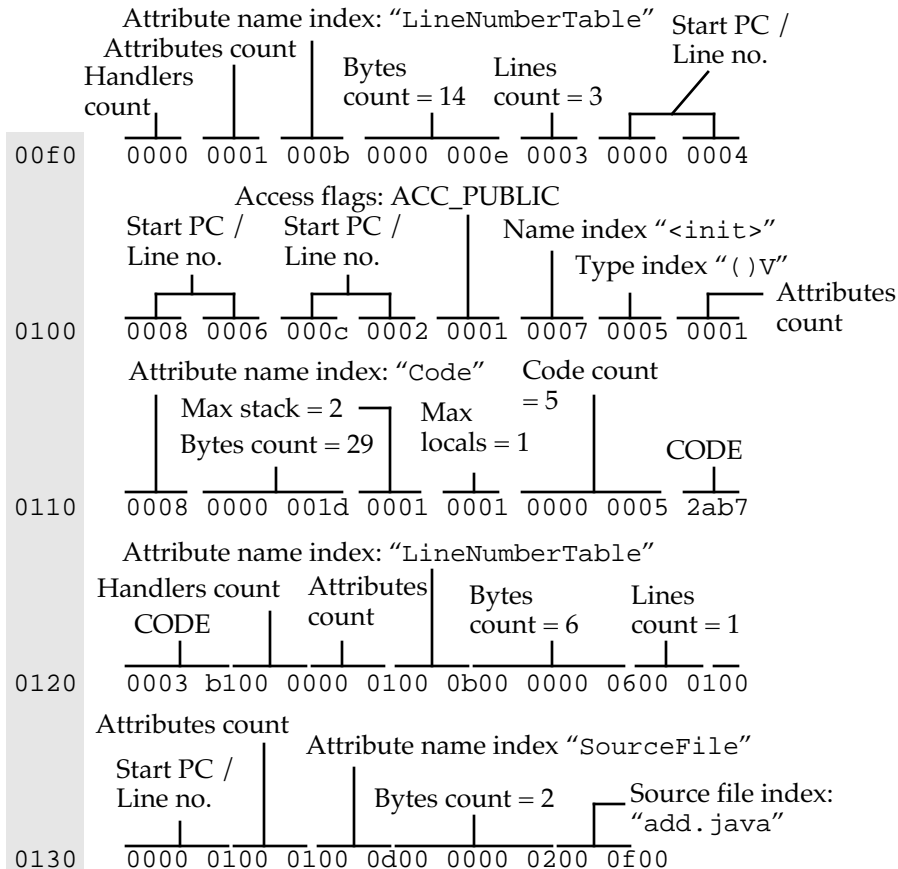
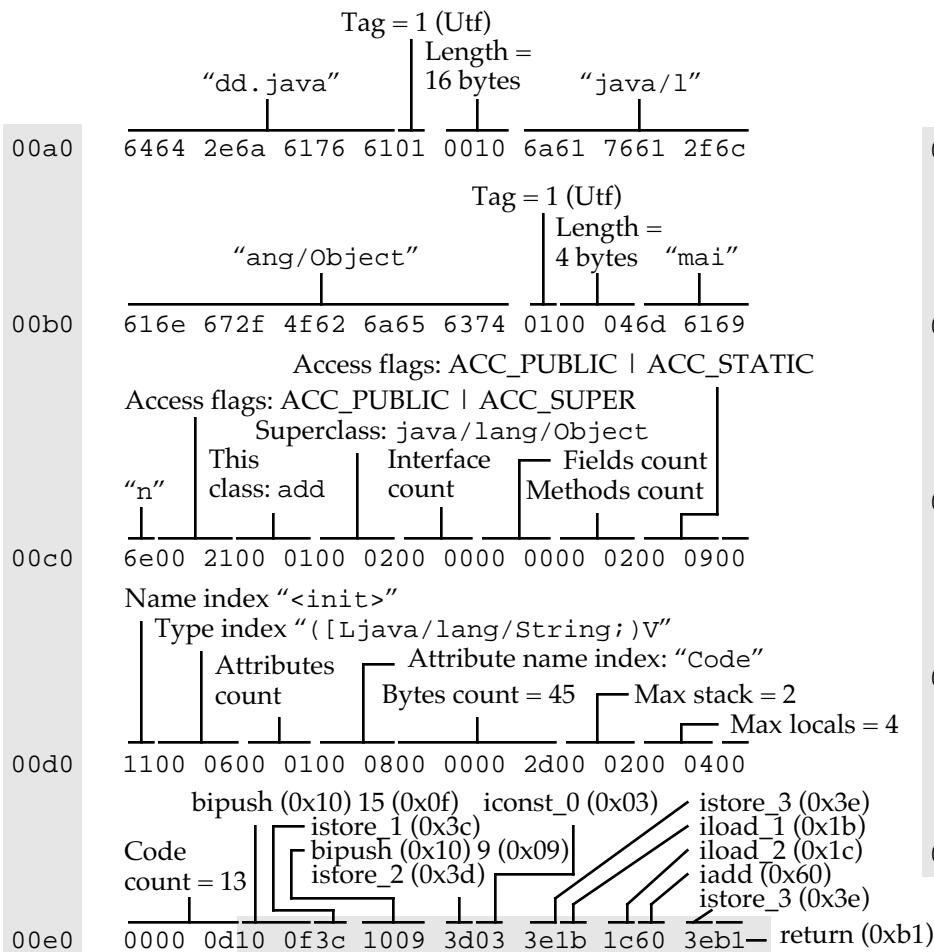
```

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	...([Ljava/lang/
0030	5374	7269	6e67	3b29	5601	0006	3c69	6e69	String;)V...<ini
0040	743e	0100	0443	6f64	6501	000d	436f	6e73	t>...Code...Cons
0050	7461	6e74	5661	6c75	6501	000a	4578	6365	tantValue...Exce
0060	7074	696f	6e73	0100	0f4c	696e	654e	756d	ptions...LineNum
0070	6265	7254	6162	6c65	0100	0e4c	6f63	616c	berTable...Local
0080	5661	7269	6162	6c65	7301	000a	536f	7572	Variables...Sour
0090	6365	4669	6c65	0100	0361	6464	0100	0861	ceFile...add...a
00a0	6464	2e6a	6176	6101	0010	6a61	7661	2f6c	dd.java...java/l
00b0	616e	672f	4f62	6a65	6374	0100	046d	6169	ang/Object...mai
00c0	6e00	2100	0100	0200	0000	0000	0200	0900	n.....
00d0	1100	0600	0100	0800	0000	2d00	0200	0400	.....
00e0	0000	0d10	0f3c	1009	3d03	3e1b	1c60	3eb1	.....
00f0	0000	0001	000b	0000	000e	0003	0000	0004	.....
0100	0008	0006	000c	0002	0001	0007	0005	0001	.....
0110	0008	0000	001d	0001	0001	0000	0005	2ab7	.....
0120	0003	b100	0000	0100	0b00	0000	0600	0100	.....
0130	0000	0100	0100	0d00	0000	0200	0f00		.....

# A Java Class File



# A Java Class File (Cont')



# Byte Code for Java Program

- Disassembled byte code for previous Java program.

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