high. In reality, there is no need to have a register there at all. All of its inputs can be fed directly through to the control store. As long as they are present at the control store at the falling edge of the clock when MIR is selected and read out, that is sufficient. There is no need to actually store them in MPC. For this reason, MPC might well be implemented as a **virtual register**, which is just a gathering place for signals, more like an electronic patch panel, than a real register. Making MPC a virtual register simplifies the timing: now events only happen on the falling and rising edges of the clock and nowhere else. But if it is easier for you to think of MPC as a real register, that is also a valid viewpoint.

4.2 AN EXAMPLE ISA: IJVM

Let us continue our example by introducing the ISA level of the machine to be interpreted by the microprogram running on the microarchitecture of Fig. 4-6 (IJVM). For convenience, we will sometimes refer to the Instruction Set Architecture as the **macroarchitecture**, to contrast it with the microarchitecture. Before we describe IJVM, however, we will digress slightly to motivate it.

4.2.1 Stacks

Virtually all programming languages support the concept of procedures (methods), which have local variables. These variables can be accessed from inside the procedure but cease to be accessible once the procedure has returned. The question thus arises: "Where should these variables be kept in memory?"

The simplest solution, to give each variable an absolute memory address, does not work. The problem is that a procedure may call itself. We will study these recursive procedures in Chap. 5. For the moment, suffice it to say that if a procedure is active (i.e., called) twice, it is impossible to store its variables in absolute memory locations because the second invocation will interfere with the first.

Instead, a different strategy is used. An area of memory, called the **stack**, is reserved for variables, but individual variables do not get absolute addresses in it. Instead, a register, say, LV, is set to point to the base of the local variables for the current procedure. In Fig. 4-8(a), a procedure *A*, which has local variables *a1*, *a2*, and *a3*, has been called, so storage for its local variables has been reserved starting at the memory location pointed to by LV. Another register, SP, points to the highest word of *A*'s local variables. If LV is 100 and words are 4 bytes, then SP will be 108. Variables are referred to by giving their offset (distance) from LV. The data structure between LV and SP (and including both words pointed to) is called *A*'s **local variable frame**.

Now let us consider what happens if *A* calls another procedure, *B*. Where should *B*'s four local variables (*b1, b2, b3, b4*) be stored? Answer: On the stack, on top of *A*'s, as shown in Fig. 4-8(b). Notice that LV has been adjusted by the

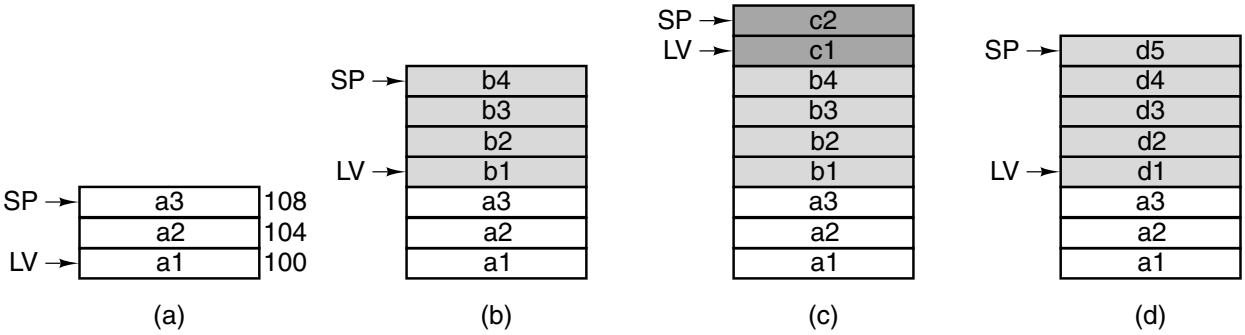

Figure 4-8. Use of a stack for storing local variables. (a) While *A* is active. (b) After *A* calls *B*. (c) After *B* calls *C*. (d) After *C* and *B* return and *A* calls *D*.

procedure call to point to *B*'s local variables instead of *A*'s. *B*'s local variables can be referred to by giving their offset from LV. Similarly, if *B* calls *C*, LV and SP are adjusted again to allocate space for *C*'s two variables, as shown in Fig. 4-8(c).

When *C* returns, *B* becomes active again, and the stack is adjusted back to Fig. 4-8(b) so that LV now points to *B*'s local variables again. Likewise, when *B* returns, we get back to the situation of Fig. 4-8(a). Under all conditions, LV points to the base of the stack frame for the currently active procedure, and SP points to the top of the stack frame.

Now suppose that *A* calls *D*, which has five local variables. We get the situation of Fig. 4-8(d), in which *D*'s local variables use the same memory that *B*'s did, as well as part of *C*'s. With this memory organization, memory is only allocated for procedures that are currently active. When a procedure returns, the memory used by its local variables is released.

Stacks have another use, in addition to holding local variables. They can be used for holding operands during the computation of an arithmetic expression. When used this way, the stack is referred to as the **operand stack**. Suppose, for example, that before calling *B*, *A* has to do the computation

```
a1 = a2 + a3;
```

One way of doing this sum is to push *a2* onto the stack, as shown in Fig. 4-9(a). Here SP has been incremented by the number of bytes in a word, say, 4, and the first operand stored at the address now pointed to by SP. Next, *a3* is pushed onto the stack, as shown in Fig. 4-9(b). As an aside on notation, we will typeset all program fragments in Helvetica, as above. We will also use this font for assembly language opcodes and machine registers, but in running text, program variables and procedures will be given in *italics*. The difference is that variables and procedure names are chosen by the user; opcodes and register names are built in.

The actual computation can be done by now executing an instruction that pops two words off the stack, adds them together, and pushes the result back onto the stack, as shown in Fig. 4-9(c). Finally, the top word can be popped off the stack and stored back in local variable *a1*, as illustrated in Fig. 4-9(d).

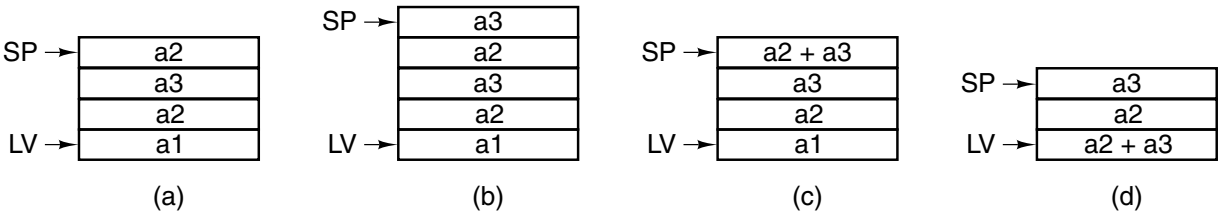

Figure 4-9. Use of an operand stack for doing an arithmetic computation.

The local variable frames and the operand stacks can be intermixed. For example, when computing an expression like $x^2 + f(x)$ part of the expression (e.g., x^2) may be on the operand stack when a function f is called. The result of the function is left on the stack, on top of x^2, so the next instruction can add them.

It is worth noting that while all machines use a stack for storing local variables, not all use an operand stack like this for doing arithmetic. In fact, most of them do not, but JVM and IJVM work like this, which is why we have introduced stack operations here. We will study them in more detail in Chap. 5.

4.2.2 The IJVM Memory Model

We are now ready to look at the IJVM's architecture. Basically, it consists of a memory that can be viewed in either of two ways: an array of 4,294,967,296 bytes (4 GB) or an array of 1,073,741,824 words, each consisting of 4 bytes. Unlike most ISAs, the Java Virtual Machine makes no absolute memory addresses directly visible at the ISA level, but there are several implicit addresses that provide the base for a pointer. IJVM instructions can only access memory by indexing from these pointers. At any time, the following areas of memory are defined:

1. *The Constant Pool.* This area cannot be written by an IJVM program and consists of constants, strings, and pointers to other areas of memory that can be referenced. It is loaded when the program is brought into memory and not changed afterward. There is an implicit register, CPP, that contains the address of the first word of the constant pool.

2. *The Local Variable Frame.* For each invocation of a method, an area is allocated for storing variables during the lifetime of the invocation. It is called the **local variable frame**. At the beginning of this frame reside the parameters (also called arguments) with which the method was invoked. The local variable frame does not include the operand stack, which is separate. However, for efficiency reasons, our implementation chooses to implement the operand stack immediately above the local variable frame. There is an implicit register that contains the address of the first location in the local variable frame. We will call this register LV. The parameters passed at the invocation of the method are stored at the beginning of the local variable frame.

3. *The Operand Stack.* The stack frame is guaranteed not to exceed a certain size, computed in advance by the Java compiler. The operand stack space is allocated directly above the local variable frame, as illustrated in Fig. 4-10. In our implementation, it is convenient to think of the operand stack as part of the local variable frame. In any case, there is an implicit register that contains the address of the top word of the stack. Notice that, unlike CPP and LV, this pointer, SP, changes during the execution of the method as operands are pushed onto the stack or popped from it.

4. *The Method Area.* Finally, there is a region of memory containing the program, referred to as the "text" area in a UNIX process. There is an implicit register that contains the address of the instruction to be fetched next. This pointer is referred to as the Program Counter, or PC. Unlike the other regions of memory, the Method Area is treated as a byte array.

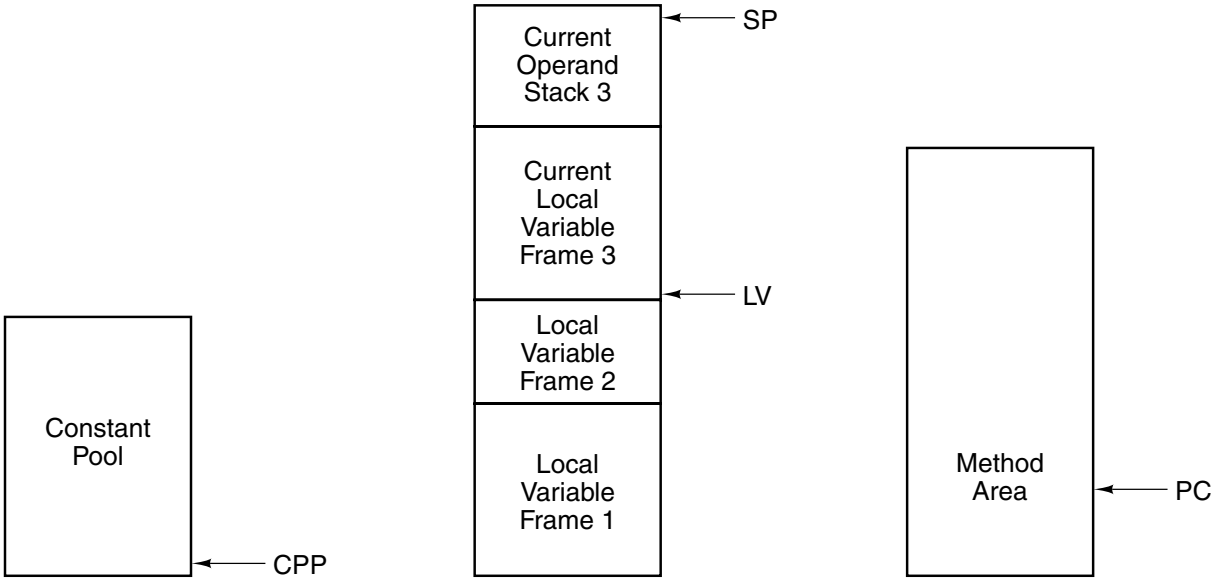

Figure 4-10. The various parts of the IJVM memory.

One point needs to be made regarding the pointers. The CPP, LV, and SP registers are all pointers to *words*, not *bytes*, and are offset by the number of words. For the integer subset we have chosen, all references to items in the constant pool, the local variables frame, and the stack are words, and all offsets used to index into these frames are word offsets. For example, LV, LV + 1, and LV + 2 refer to the first three words of the local variables frame. In contrast, LV, LV + 4, and LV + 8 refer to words at intervals of four words (16 bytes).

In contrast, PC contains a byte address, and an addition or subtraction to PC changes the address by a number of bytes, not a number of words. Addressing for

PC is different from the others, and this fact is apparent in the special memory port provided for PC on the Mic-1. Remember that it is only 1 byte wide. Incrementing PC by one and initiating a read results in a fetch of the next *byte*. Incrementing SP by one and initiating a read results in a fetch of the next *word*.

4.2.3 The IJVM Instruction Set

The IJVM instruction set is shown in Fig. 4-11. Each instruction consists of an opcode and sometimes an operand, such as a memory offset or a constant. The first column gives the hexadecimal encoding of the instruction. The second gives its assembly language mnemonic. The third gives a brief description of its effect.

Hex	Mnemonic	Meaning
0x10	BIPUSH *byte*	Push byte onto stack
0x59	DUP	Copy top word on stack and push onto stack
0xA7	GOTO *offset*	Unconditional branch
0x60	IADD	Pop two words from stack; push their sum
0x7E	IAND	Pop two words from stack; push Boolean AND
0x99	IFEQ *offset*	Pop word from stack and branch if it is zero
0x9B	IFLT *offset*	Pop word from stack and branch if it is less than zero
0x9F	IF_ICMPEQ *offset*	Pop two words from stack; branch if equal
0x84	IINC *varnum const*	Add a constant to a local variable
0x15	ILOAD *varnum*	Push local variable onto stack
0xB6	INVOKEVIRTUAL *disp*	Invoke a method
0x80	IOR	Pop two words from stack; push Boolean OR
0xAC	IRETURN	Return from method with integer value
0x36	ISTORE *varnum*	Pop word from stack and store in local variable
0x64	ISUB	Pop two words from stack; push their difference
0x13	LDC_W *index*	Push constant from constant pool onto stack
0x00	NOP	Do nothing
0x57	POP	Delete word on top of stack
0x5F	SWAP	Swap the two top words on the stack
0xC4	WIDE	Prefix instruction; next instruction has a 16-bit index

Figure 4-11. The IJVM instruction set. The operands *byte*, *const*, and *varnum* are 1 byte. The operands *disp*, *index*, and *offset* are 2 bytes.

Instructions are provided to push a word from various sources onto the stack. These sources include the constant pool (LDC_W), the local variable frame (ILOAD), and the instruction itself (BIPUSH). A variable can also be popped from

the stack and stored into the local variable frame (ISTORE). Two arithmetic operations (IADD and ISUB) as well as two logical (Boolean) operations (IAND and IOR) can be performed using the two top words on the stack as operands. In all the arithmetic and logical operations, two words are popped from the stack and the result pushed back onto it. Four branch instructions are provided, one unconditional (GOTO) and three conditional ones (IFEQ, IFLT, and IF_ICMPEQ). All the branch instructions, if taken, adjust the value of PC by the size of their (16-bit signed) offset, which follows the opcode in the instruction. This offset is added to the address of the opcode. There are also IJVM instructions for swapping the top two words on the stack (SWAP), duplicating the top word (DUP), and removing it (POP).

Some instructions have multiple formats, allowing a short form for commonly-used versions. In IJVM we have included two of the various mechanisms JVM uses to accomplish this. In one case we have skipped the short form in favor of the more general one. In another case we show how the prefix instruction WIDE can be used to modify the ensuing instruction.

Finally, there is an instruction (INVOKEVIRTUAL) for invoking another method, and another instruction (IRETURN) for exiting the method and returning control to the method that invoked it. Due to the complexity of the mechanism we have slightly simplified the definition, making it possible to produce a straightforward mechanism for invoking a call and return. The restriction is that, unlike Java, we only allow a method to invoke a method existing within its own object. This restriction severely cripples the object orientation but allows us to present a much simpler mechanism, by avoiding the requirement to locate the method dynamically. (If you are not familiar with object-oriented programming, you can safely ignore this remark. What we have done is turn Java back into a nonobject-oriented language, such as C or Pascal.) On all computers except JVM, the address of the procedure to call is determined directly by the CALL instruction, so our approach is actually the normal case, not the exception.

The mechanism for invoking a method is as follows. First, the caller pushes onto the stack a reference (pointer) to the object to be called. (This reference is not needed in IJVM since no other object may be specified, but it is retained for consistency with JVM.) In Fig. 4-12(a) this reference is indicated by OBJREF. Then the caller pushes the method's parameters onto the stack, in this example, *Parameter 1*, *Parameter 2*, and *Parameter 3*. Finally, INVOKEVIRTUAL is executed.

The INVOKEVIRTUAL instruction includes a displacement which indicates the position in the constant pool that contains the start address within the Method Area for the method being invoked. However, while the method code resides at the location pointed to by this pointer, the first 4 bytes in the method area contain special data. The first 2 bytes are interpreted as a 16-bit integer indicating the number of parameters for the method (the parameters themselves have previously been pushed onto the stack). For this count, OBJREF is counted as a parameter:

parameter 0. This 16-bit integer, together with the value of SP, provides the location of OBJREF. Note that LV points to OBJREF rather than the first real parameter. The choice where LV points is somewhat arbitrary.

The second 2 bytes in the method area are interpreted as another 16-bit integer indicating the size of the local variable area for the method being invoked. This is necessary because a new stack will be established for the method, beginning immediately above the local variable frame. Finally, the fifth byte in the method area contains the first opcode to be executed.

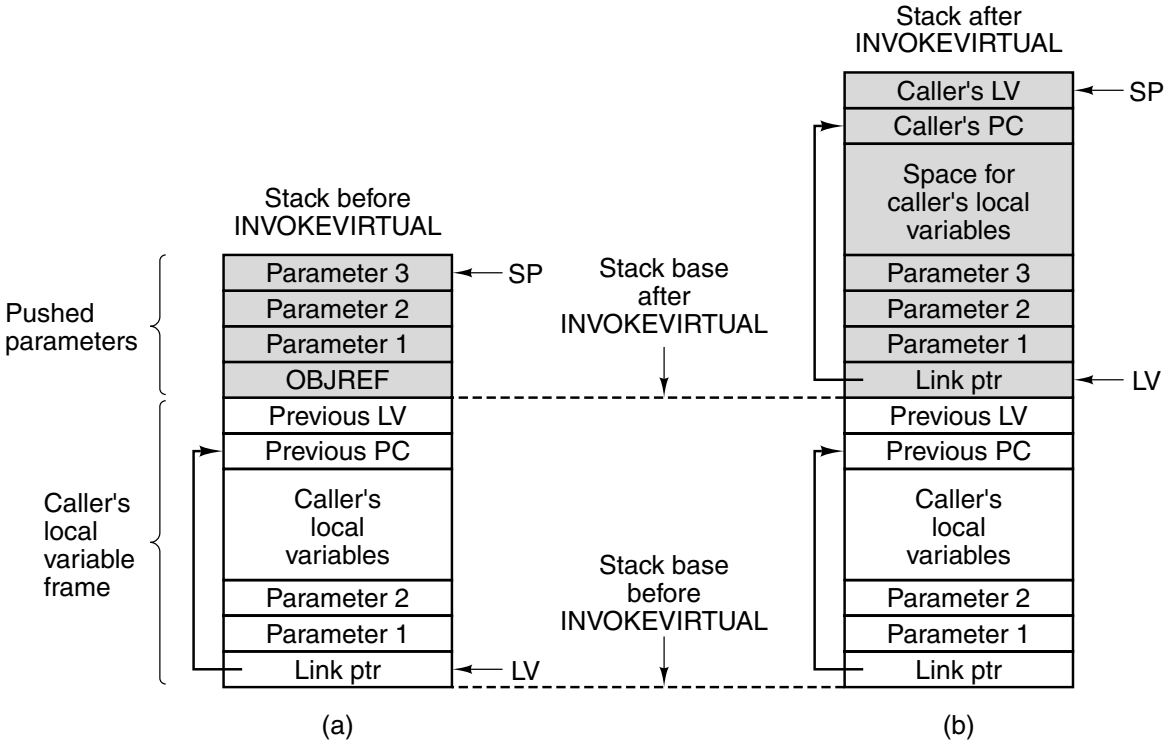

Figure 4-12. (a) Memory before executing INVOKEVIRTUAL. (b) After executing it.

The actual sequence that occurs for INVOKEVIRTUAL is as follows and is depicted in Fig. 4-12. The two unsigned index bytes that follow the opcode are used to construct an index into the constant pool table (the first byte is the high-order byte). The instruction computes the base address of the new local variable frame by subtracting off the number of parameters from the stack pointer and setting LV to point to OBJREF. At this location, overwriting OBJREF, the implementation stores the address of the location where the old PC is to be stored. This address is computed by adding the size of the local variable frame (parameters + local variables) to the address contained in LV. Immediately above the address where the old PC is to be stored is the address where the old LV is to be stored. Immediately above that address is the beginning of the stack for the newly-called procedure. SP is set to point to the old LV, which is the address immediately below the first empty location on the stack. Remember that SP always points to the top word on

the stack. If the stack is empty, it points to the first location below the end of the stack because our stacks grow upward, toward higher addresses. In our figures, stacks always grow upward, toward the higher address at the top of the page.

The last operation needed to carry out INVOKEVIRTUAL is to set PC to point to the fifth byte in the method code space.

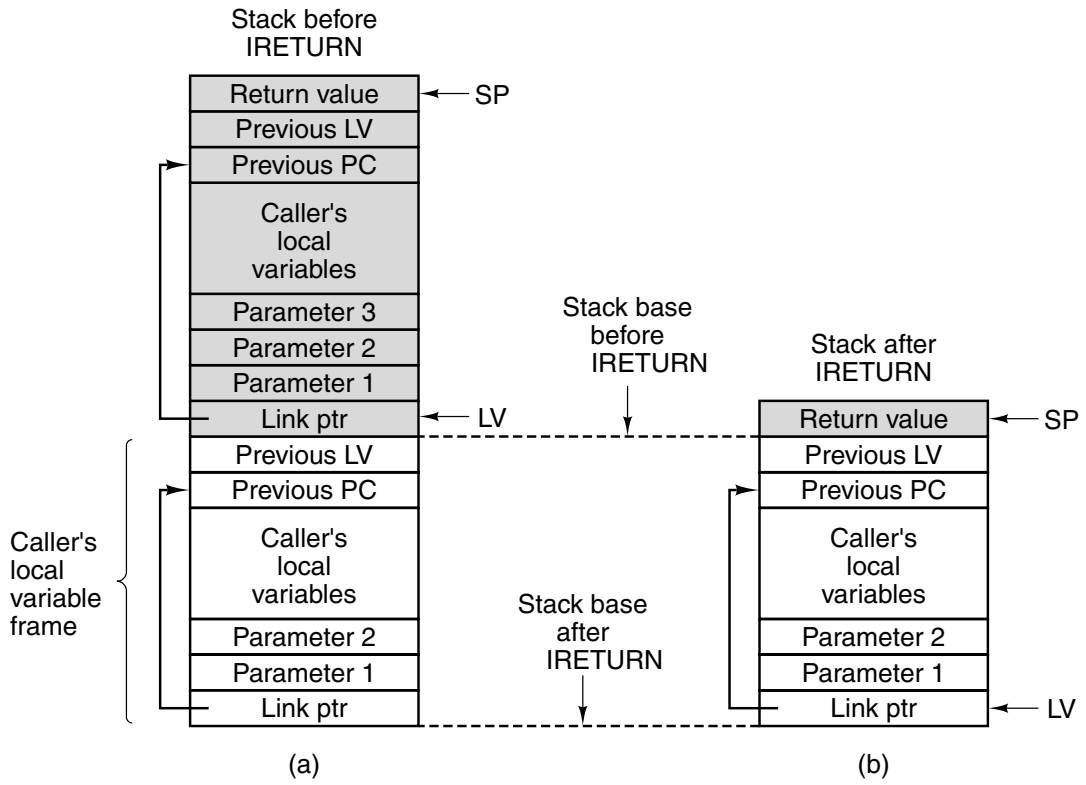

Figure 4-13. (a) Memory before executing IRETURN. (b) After executing it.

The IRETURN instruction reverses the operations of the INVOKEVIRTUAL instruction, as shown in Fig. 4-13. It deallocates the space used by the returning method. It also restores the stack to its former state, except that (1) the (now overwritten) OBJREF word and all the parameters have been popped from the stack, and (2) the returned value has been placed at the top of the stack, at the location formerly occupied by OBJREF. To restore the old state, the IRETURN instruction must be able to restore the PC and LV pointers to their old values. It does this by accessing the link pointer (which is the word identified by the current LV pointer). In this location, remember, where the OBJREF was originally stored, the INVOKEVIRTUAL instruction stored the address containing the old PC. This word and the word above it are retrieved to restore PC and LV, respectively, to their old values. The return value, which is stored at the top of the stack of the terminating method, is copied to the location where the OBJREF was originally stored, and SP is restored to point to this location. Control is therefore returned to the instruction immediately following the INVOKEVIRTUAL instruction.

So far, our machine does not have any input/output instructions. Nor are we going to add any. It does not need them any more than the Java Virtual Machine needs them, and the official specification for JVM never even mentions I/O. The theory is that a machine that does no input or output is "safe." (Reading and writing are performed in JVM by means of calls to special I/O methods.

4.2.4 Compiling Java to IJVM

Let us now see how Java and IJVM relate to one another. In Fig. 4-14(a) we show a simple fragment of Java code. When fed to a Java compiler, the compiler would probably produce the IJVM assembly language shown in Fig. 4-14(b). The line numbers from 1 to 15 at the left of the assembly language program are not part of the compiler output. Nor are the comments (starting with //). They are there to help explain a subsequent figure. The Java assembler would then translate the assembly program into the binary program shown in Fig. 4-14(c). (Actually, the Java compiler does its own assembly and produces the binary program directly.) For this example, we have assumed that *i* is local variable 1, *j* is local variable 2, and *k* is local variable 3.

```
i = j + k;                1       ILOAD j        // i = j + k        0x15 0x02
if (i == 3)               2       ILOAD k                         0x15 0x03
    k = 0;                3       IADD                            0x60
else                      4       ISTORE i                        0x36 0x01
   j = j – 1;             5       ILOAD i        // if (i == 3)     0x15 0x01
                          6       BIPUSH 3                        0x10 0x03
                          7       IF_ICMPEQ L1                    0x9F 0x00 0x0D
                          8       ILOAD j        // j = j – 1       0x15 0x02
                          9       BIPUSH 1                        0x10 0x01
                         10       ISUB                            0x64
                         11       ISTORE j                        0x36 0x02
                         12       GOTO L2                         0xA7 0x00 0x07
                         13  L1:  BIPUSH 0       // k = 0          0x10 0x00
                         14       ISTORE k                        0x36 0x03
                         15  L2:
   (a)                              (b)                             (c)
```

Figure 4-14. (a) A Java fragment. (b) The corresponding Java assembly language. (c) The IJVM program in hexadecimal.

The compiled code is straightforward. First *j* and *k* are pushed onto the stack, added, and the result stored in *i*. Then *i* and the constant 3 are pushed onto the stack and compared. If they are equal, a branch is taken to *L1*, where *k* is set to 0. If they are unequal, the compare fails and code following IF_ICMPEQ is executed. When it is done, it branches to *L2*, where the then and else parts merge.