PART III

Abstractions of State

In the previous part, we characterized each type of data by the collection of operations that could be performed on data of that type. However, there were only two fundamental kinds of operations: those that constructed new things and those that asked questions about existing things. In this part, we'll add a third, fundamentally different, kind of operation, one that modifies an existing object. This kind of operation also raises the possibility that two concurrently active computations will interact because one can modify an object the other is using. Therefore, at the conclusion of this part we'll examine concurrent, interacting computations.

We will lead into our discussion of changeable objects by looking at how computers are organized and how they carry out computations. The relevance of this discussion is that the storage locations in a computer's memory constitute the fundamental changeable object. We'll look next at how numerically indexed storage locations, like a computer's memory, can be used to make some computational processes dramatically more efficient by eliminating redundant recomputations of subproblem results. Then we'll look at other forms of modifiable objects, where the operations don't reflect the computer's numbered storage locations but rather reflect application-specific concerns. We'll then build this technique into object-oriented programming by blending in the idea that multiple concrete kinds of objects can share a common interface of generic operations. Finally, we will show how to transplant these same ideas into another programming language, Java, that we will use to introduce programs that have concurrent, interacting components.

CHAPTER ELEVEN

Computers with Memory

11.1 Introduction

In the first two parts of the book we looked at computational processes from the perspective of the procedures and the data on which those procedures describe operations, but we've not yet discussed the computer that does the processing. In this chapter, we'll look at the overall structure of a typical present-day computer and see how such a computer is actually able to carry out a procedurally specified computational process.

One of the most noteworthy components we'll see that computers have is *memory* (specifically, *Random Access Memory* or *RAM*). What makes memory so interesting is that it is unlike anything we've seen thus far—it is not a process or a procedure for carrying out a process, and it is also not simply a value or a collection of values. Rather, it is a collection of *locations* in which values can be stored; each location has a particular value at any one time, but the value can be changed so that the location contains a different value than it used to.

After seeing collections of memory locations as a component of computers, we'll see how they are also available for our use when programming in Scheme, as so-called *vectors*. In this chapter, we introduce vectors and use them to build a computer simulator in Scheme. In the following chapters we look at ways in which these locations can be used to improve the efficiency of computational processes and to construct software systems that are modular and naturally reflect the structure of noncomputational systems that the software models.

11.2 An Example Computer Architecture

In this section, we will attempt to "open the hood" of a computer like the one you have been using while working through this book. However, because so many different types of computers exist, and because actual computers are highly complex machines involving many engineering design decisions, we will make two simplifications. First, rather than choosing any one real computer to explain, we've made up our own simple, yet representative, computer, the Super-Lean Instruction Machine, also known as SLIM. Second, rather than presenting the design of SLIM in detail, we describe it at the *architectural* level. By *architecture* we mean the overall structure of the computer system to the extent it is relevant to the computer's ability to execute a program.

You might well wonder whether an actual SLIM computer exists that meets the specifications of our architectural design. To our knowledge, no such computer does exist, although in principle one could be fabricated. (Before construction could begin, the specifications would need to be made more complete than the version we present here.) Because you are unlikely to find a real SLIM, we provide a simulated SLIM computer on the web site for this book; we will say more about this simulated computer in the next section. In fact, this chapter's application section involves writing another simulator for SLIM.

Even at the architectural level, we have many options open to us as computer designers. We use the SLIM architecture to focus on a single representative set of choices rather than illustrating the entire range of options. These choices were made to be as simple as possible while still remaining broadly similar to what is typical of today's architectures. We point out a few specific areas where alternative choices are common, but you should keep in mind that the entire architecture consists of nothing but decisions, none of which is universal. A good successor course on computer organization and architecture will not only show you the options we're omitting but will also explain how a designer can choose among those options to rationally balance price and performance.

SLIM is a *stored program computer.* By this we mean that its behavior consists of performing a sequence of operations determined by a *program*, which is a list of *instructions*. The set of possible instructions, called the computer's *instruction set*, enumerates the computer's basic capabilities. Each instruction manipulates certain objects in the computer—for example, reading input from the keyboard, storing some value in a memory location, or adding the values in two memory locations and putting the result into a third. The way that an actual computer accomplishes these tasks is a very interesting story but not one we will pursue here. Viewing SLIM as a stored program computer allows us to focus on the computational core of computers.

You might well ask, "How does this information relate to my computer? I don't recall ever specifically telling my computer to run through a list of instructions." In fact, you probably have done so, regardless of how primitive or advanced your computer is. Turning on (or "booting up") the computer implicitly loads in and starts running an initial program known as an *operating system*, part of whose task is to make it easy to run other programs. The applications you use on your computer (such as your Scheme system) are programs stored in the computer's memory. When you

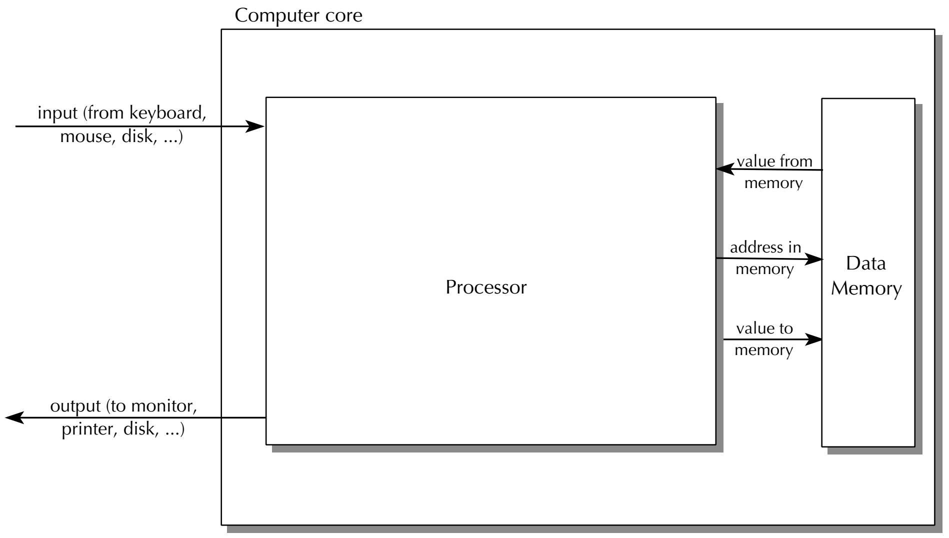

Figure 11.1 High-level view of SLIM

invoke one of these applications, perhaps by using a mouse to click on an icon, you are causing the operating system to load the application program into the computer's instruction memory and start execution at the program's first instruction.

We start with a *structural* description of SLIM: Figure 11.1 shows a high-level, coarse-grained view of its architecture. The boxes in this diagram represent SLIM's functional units, and the arrows show paths for the flow of data between these units. Our structural description will involve describing the tasks of the functional units, successively "opening them up" to reveal their functional subunits and internal data paths. We will stop at a level detailed enough to give an understanding of how a stored program works rather than continuing to open each unit until we get to the level of the electrical circuits that implement it. In the next section we will turn our attention to an *operational* understanding of the architecture, and will enumerate the instructions it can execute.

The *computer core* is an organizing concept referring to those parts of SLIM except its input and output devices—imagine it as your computer minus its keyboard, mouse, monitor, and disk drive. Because SLIM is a stored program computer, the task of the computer is to *run* (or *execute*) a program, which takes in input and produces output. Instead of considering all of the possible *input* and *output devices* enumerated in the diagram, we will make the simplifying assumption that input comes from the keyboard and output goes to the monitor screen.

The *processor* performs the operations that constitute the execution of a program, using the *data memory* to store values as needed for the program. When a processor operation requires that values be stored into or retrieved from memory, it sends to the memory unit the *address* (described below) of the memory location. The memory unit remembers what value was most recently stored by the processor into each location. When the processor then asks the memory to retrieve a value from a particular location, the memory returns to the processor that most recently stored value, leaving it in the location so that the same value can be retrieved again. The processor can also consume input and produce output.

Even at this very crude level, our architecture for SLIM already embodies important decisions. For example, we've decided not to have multiple independently operating processors, all storing to and retrieving from a single shared memory. Yet in practice, such *shared-memory multiprocessor* systems are becoming relatively common at the time of this writing. Similarly, for simplicity we've connected the input and output devices only to the processor in SLIM, yet real architectures today commonly include *Direct Memory Access* (*DMA*), in which input can flow into memory and output can be retrieved directly from memory without passing through the processor.

Now we need to examine each of the boxes in the computer core more closely. The memory component is the simpler one. Conceptually, it is a long sequence of "slots" (or "boxes") that are the individual memory locations. In order to allow the processor to uniquely specify each location, the slots are sequentially numbered starting at 0. The number corresponding to a given slot is called its *address*. When the processor asks the memory to store 7 at address 13, the memory unit throws away the value that is in the slot numbered 13 and puts a 7 into that slot, as shown in Figure 11.2. At any later time, as long as no other store into location 13 has been done in the meantime, the processor can ask the memory to retrieve the value from

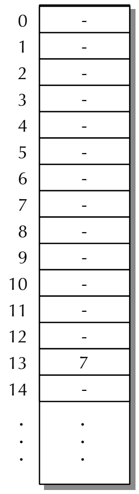

Figure 11.2 Memory, with 7 stored at address 13

address 13 and get the 7 back. (Note that the location numbered 13 is actually the fourteenth location because the first address is 0.)

The processor has considerably more internal structure than the memory because it needs to

- Keep track of what it is supposed to do next as it steps through the instructions that constitute the program
- Locally store a limited number of values that are actively being used so that it doesn't need to send store and retrieve requests to the memory so frequently
- Do the actual arithmetic operations, such as addition

The three subcomponents of the processor responsible for these three activities are called the *control unit*, the *registers*, and the *arithmetic logical unit* (or *ALU*), respectively. Figure 11.3 illustrates these three units and the main data paths between them. As you can see, in SLIM everything goes to or from the registers. (Registers are locations, like those in the memory: They can be stored into and retrieved from.) The ALU receives the operands for its arithmetic operations from registers and stores the result back in a register. If values stored in memory are to be operated on, they first have to be loaded into registers. Then the arithmetic operation can be performed, and the result will be stored in a register. Finally, the result can be stored in memory, if desired, by copying it from the register.

In addition to the data paths shown in the diagram, additional paths lead out of the control unit to the other units that allow the control unit to tell the ALU which arithmetic operation to do (addition, subtraction, multiplication, . . .), to tell the register set which specific registers' values are to be retrieved or stored, and to

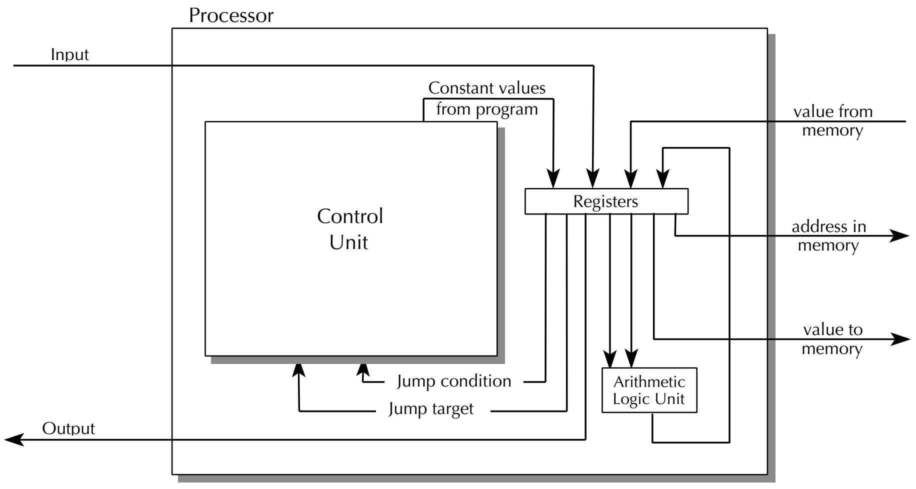

Figure 11.3 SLIM's processor

tell the memory whether a value is to be stored into the specified address or retrieved from it. We don't show these control paths because they complicate the diagram. Keep in mind, however, that whenever we describe some capability of one of the units, we are implicitly telling you that there is a connection from the control unit to the unit in question that allows the control unit to cause that capability to be used. For example, when we tell you that the ALU can add, we are telling you that there is a path leading from the control unit to the ALU that carries the information about whether the control unit wishes the ALU to add. From the operational viewpoint of the next section, therefore, an *add* instruction is in SLIM's instruction set.

Zooming in another level of detail, we examine each of the boxes shown in the processor diagram more closely, starting with the registers. The registers unit is just like the memory, a numbered collection of locations, except that it is much smaller and more intimately connected with the rest of the processor. SLIM has 32 registers, a typical number currently. (In contrast, the data memory has at least tens of thousands of locations and often even millions.) The registers are numbered from 0 to 31, and these register numbers play an important role in nearly all of the computer's instructions. For example, an instruction to perform an addition might specify that the numbers contained in registers 2 and 5 should be added together, with the sum to be placed into register 17. Thus, the addition instruction contains three register numbers: two *source registers* and one *destination register*. An instruction to store a value into memory contains two register numbers: the source register, which holds the value to store, and an *address register*, which holds the memory address for storing that value.

The ALU can perform any of the arithmetic operations that SLIM has instructions for: addition, subtraction, multiplication, division, quotient, remainder, and numeric comparison operations. The numeric comparison operations compare two numbers and produce a numeric result, which is either 1 for true or 0 for false. The ALU can do six kinds of comparison: $=$, $\neq$, $<$, $>$, $\leq$, and $\geq$. This is a quite complete set of arithmetic and comparison operations by contemporary standards; some real architectures don't provide the full set of comparison operations, for example, or they require multiplication and division to be done with a sequence of instructions rather than a single instruction.

The control unit, shown in Figure 11.4, contains the program to execute in an *instruction memory*. Like the main (data) memory, the instruction memory has numbered locations, and we call the location numbers addresses. The difference is that instead of containing values to operate on, these locations contain the instructions for doing the operating. For example, the instruction at address 0 might say to load a 7 into register number 3. (Many architectures share a single memory for both instructions and data; the kind of architecture we've chosen, with separate memories, is known as a *Harvard architecture*.)

At any time, the computer is executing one particular instruction from the instruction memory, which we call the *current instruction*. The address in the instruction

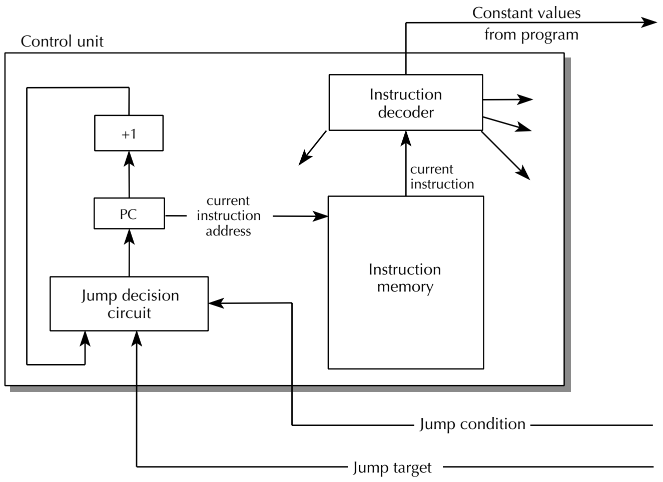

Figure 11.4 SLIM's control unit

memory at which this current instruction appears is the *current instruction address*. A special storage location, called the *program counter*, or *PC*, is used to hold the current instruction address. When the computer is first started, a 0 is placed into the PC, so the first instruction executed will be the one at address 0 (i.e., the first one in the instruction memory). Thereafter, the computer normally executes consecutive instructions (i.e., after executing the instruction at address 0, the computer would normally execute the instruction at address 1, then 2, etc.). This is achieved by having a unit that can add 1 to the value in the PC and having the output from this adder feed back into the PC. However, there are special *jump* instructions that say to *not* load the PC with the output of the adder but instead to load it with a new instruction address (the *jump target address*) taken from a register. Thus, there is a jump decision circuit that controls whether the PC is loaded from the adder (which always adds 1), for continuing on to the next instruction, or from the jump target address (which comes from the registers unit), for shifting to a different place in the instruction sequence when a jump instruction is executed. This jump decision circuit can decide whether to jump based on the jump condition value, which also comes from the registers unit.

The PC provides the current instruction address to the instruction memory, which in turn provides the current instruction to the *instruction decoder*. The instruction decoder sends the appropriate control signals to the various units to make the instruction actually happen. For example, if the instruction says to add the contents of registers 2 and 5 and put the sum in register 17, the control unit would send *control signals* to the registers to retrieve the values from registers 2 and 5 and pass those values to the ALU. Further control signals would tell the ALU to add the values it received. And finally, control signals would tell the registers to load the value

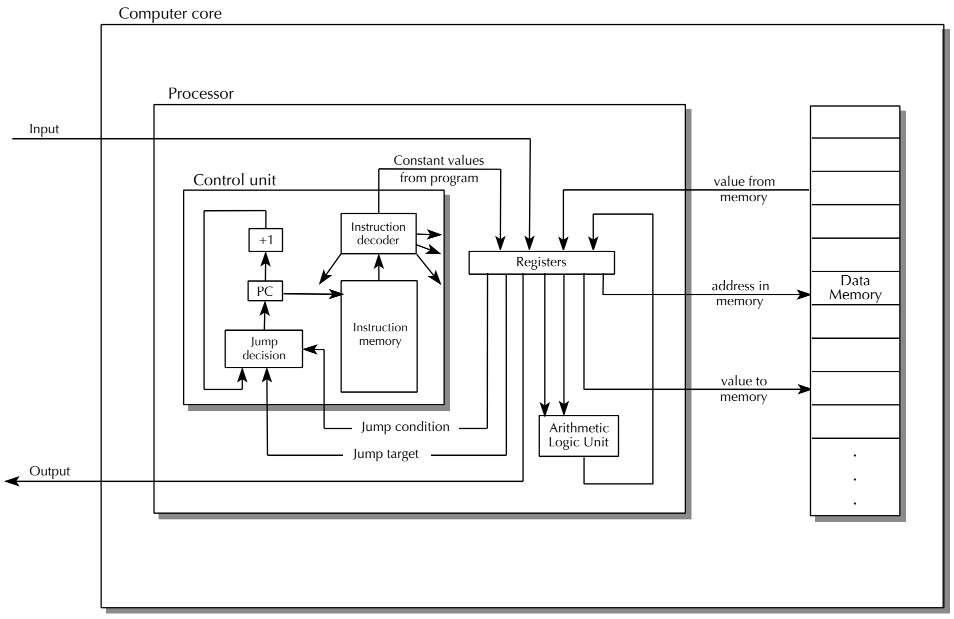

Figure 11.5 The entire SLIM architecture

received from the ALU into register 17. The other connection from the control unit to the rest of the processor allows instructions to include constant values to load into a register; for example, if an instruction said to load a 7 into register 3, the control unit would send the 7 out along with control signals saying to load this constant value into register 3.

We could in principle continue "opening up" the various boxes in our diagrams and elucidating their internal structure in terms of more specialized boxes until ultimately we arrived at the level of individual transistors making up the computer's circuitry. However, at this point we'll declare ourselves satisfied with our *structural* knowledge of the computer architecture. In the next section we will turn our attention to an *operational* understanding of the architecture and will enumerate the instructions it can execute. Our structural knowledge of the SLIM architecture is summarized in Figure 11.5, which combines into a single figure the different levels of detail that were previously shown in separate figures.

11.3 Programming the SLIM

In this section, we will examine the instructions that SLIM can execute. Each instruction can be written in two notations. Within the computer's instruction memory, each location contains an instruction that is encoded in *machine language*, which is a notation for instructions that is designed to be easy for the computer to decode and execute rather than to be easy for humans to read and write. Therefore, for human convenience we also have an *assembly language* form for each instruction, which is the form that we use in this book. A program known as an *assembler* can mechanically translate the instructions constituting a program from assembly language to machine language; the result can then be loaded into the instruction memory for running. (The machine language form of an instruction in instruction memory is a pattern of *bits*, that is, of 0s and 1s. The sidebar What Can Be Stored in a Location? explains that memory locations fundamentally hold bit patterns; in instruction memory, those bit patterns represent instructions.)

Each instruction contains an *operation code*, or *opcode*, that specifies what operation should be done, for example, an addition or store. In the assembly language we will use, the opcode always will be a symbol at the beginning of the instruction; for example, in the instruction `add 17, 2, 5`, the opcode is `add` and indicates that an addition should be done. After the opcode, the remainder of the instruction consists of the *operand specifiers*. In the preceding example, the operand specifiers are `17`, `2`, and `5`. In the SLIM instruction set, most operand specifiers need to be register numbers. For example, this instruction tells the computer to add the contents of registers 2 and 5 and store the sum into register 17. (Note that the first operand specifies where the result should go.) To summarize, we say that the form of an addition instruction is `add` *destreg*`,` *sourcereg*$_1$`,` *sourcereg*$_2$. We'll use this same notation for describing the other kinds of operations as well, with *destreg* for operand specifiers that are destination register numbers, *sourcereg* for operand specifiers that are source register numbers, *addressreg* for operand specifiers that are address register numbers (i.e., register numbers for registers holding memory addresses), and *const* for operand specifiers that are constant values.

Each of the 12 arithmetic operations the ALU can perform has a corresponding instruction opcode. We've already seen `add` for addition; the others are `sub` for subtraction, `mul` for multiplication, `div` for division, `quo` for quotient, `rem` for remainder, `seq` for $=$, `sne` for $\neq$, `slt` for $<$, `sgt` for $>$, `sle` for $\leq$, and `sge` for $\geq$. The overall form of all these instructions is the same; for example, for multiplication it would be `mul` *destreg*`,` *sourcereg*$_1$`,` *sourcereg*$_2$. Recall that the comparison operations all yield a 0 for false or a 1 for true. So, if register 7 contains a smaller number than register 3 does, after executing the instruction `slt 5, 7, 3`, register number 5 will contain a 1. The `s` on the front of the comparison instructions is for "set," because they set the destination register to an indication of whether the specified relationship holds between the source registers' values.

There are two instructions for moving values between registers and memory. To *load* a value into a register from a memory location, the `ld` opcode is used: `ld` *destreg*`,` *addressreg*. To *store* a value from a register into a memory location, the `st` opcode is used: `st` *sourcereg*`,` *addressreg*. As an example, if register 7 contains 15, and memory location 15 contains 23, after executing the instruction `ld 3, 7`,

What Can Be Stored in a Location?

One of the many issues we gloss over in our brief presentation of computer architecture is the question of what values can be stored in a memory location or register. Until now we have acted as though any number could be stored into a location. The purpose of this sidebar is to confess that locations in fact have a limited size and as such can only hold a limited range of numbers.

Each storage location has room for some fixed number of units of information called *bits*. Each bit-sized piece of storage is so small that it can only accommodate two values, conventionally written as 0 and 1. Because each bit can have two values, 2 bits can have any of four bit patterns: $(0, 0)$, $(0, 1)$, $(1, 0)$, and $(1, 1)$; similarly 3 bits worth of storage can hold eight different bit patterns: $(0, 0, 0)$, $(0, 0, 1)$, $(0, 1, 0)$, $(0, 1, 1)$, $(1, 0, 0)$, $(1, 0, 1)$, $(1, 1, 0)$, and $(1, 1, 1)$. In general, with n bits it is possible to store 2^n different patterns. The number of bits in a storage location is called the *word size* of the computer. Typical present-day computers have word sizes of 32 or 64 bits. With a 64-bit word size, for example, each storage location can hold 2^{64}, or 18,446,744,073,709,551,616 distinct patterns.

It is up to the computer's designer to decide what values the 2^n bit patterns that can be stored in an n-bit *word* represent. For example, the 2^{64} bit patterns that can be stored in a 64-bit word could be used to represent either the integers in the range from 0 to $2^{64} - 1$ or the integers in the range from -2^{63} to $2^{63} - 1$ because both ranges contain 2^{64} values. It would also be possible to take the bit patterns as representing fractions with numerator and denominator both in the range -2^{31} to $2^{31} - 1$ because there are 2^{64} of these as well. Another option, more popular than fractions for representing nonintegral values, is so-called *floating point* numerals of the form $m \times 2^e$, where the *mantissa*, m, and the *exponent*, e, are integers chosen from sufficiently restricted ranges that both integers' representations can be packed into the word size. In our example of a 64-bit word, it would be typical to devote 53 bits to the the mantissa and 11 bits to the exponent. If each of these subwords were used to encode a signed integer in the conventional way, this would allow m to range from -2^{52} to $2^{52} - 1$ and e to range from -2^{10} to $2^{10} - 1$. Again, this results in a total of 2^{64} numerals.

Of course, the circuitry of the ALU will have to reflect the computer designer's decision regarding which number representation is in use. This is because the ALU is in the business of producing the bit pattern that represents the result of an arithmetic operation on the numbers represented by two given bit patterns. Many computers actually have ALUs that can perform arithmetic operations on several different representations; on such a machine, instructions to carry out arithmetic operations specify not only the operation but also the representation. For example,

(Continued)

What Can Be Stored in a Location? (Continued)

one instruction might say to add two registers' bit patterns interpreted as integers, whereas a different instruction might say to add the bit patterns interpreted as floating point numerals.

Whatever word size and number representation the computer's designer chooses will impose some limitation on the values that can be stored in a location. However, the computer's programmer can represent other kinds of values in terms of these. For example, numbers too large to fit in a single location can be stored in multiple locations, and nonnumeric values can be encoded as numbers.

register 3 will also contain 23. If register 4 contains 17 and register 6 contains 2, executing `st 4, 6` will result in memory location 2 containing a 17.

To read a value into a register from the keyboard, the instruction `read` *destreg* can be used, whereas to write a value from a register onto the display, the instruction would be `write` *sourcereg*. We've included these instructions in the SLIM architecture in order to make it easier to write simple numeric programs in assembly language, but a real machine would only have instructions for reading in or writing out individual characters. For example, to write out 314, it would have to write out a `3` character, then a `1`, and finally a `4`. The programming techniques shown later in this chapter would allow you to write assembly language subprograms to perform numeric input and output on a variant of the SLIM architecture that only had the read-a-character and write-a-character operations. Thus, by assuming that we can input or output an entire number with one instruction, all we are doing is avoiding some rather tedious programming.

The one other source for a value to place into a register is a constant value appearing in the program. For this the so-called *load immediate* opcode, `li`, is used: `li` *destreg*`,` *const*. For example, a program that consisted of the two instructions:

```
li 1, 314
write 1
```

would display a 314 because it loads that value into register 1 and then writes out the contents of register 1 to the display.

Actually, the preceding two-instruction program above isn't quite complete because nothing stops the computer from going on to the third location in its instruction memory and executing the instruction stored there as well; we want to have some way to make the computer stop after the program is done. This action can be arranged by having a special `halt` instruction, which stops the computer. So, our first real program is as follows:

```
li 1, 314
write 1
halt
```

Exercise 11.1

Suppose you want to store the constant value 7 into location 13 in memory. Let's see what is involved in making this store happen. You'll need a copy of Figure 11.5 for this exercise (that figure is the diagram of the entire SLIM architecture).

a. Use color or shading to highlight the lines in the diagram that the value 7 will travel along on its way from the instruction memory to the main memory.

b. Similarly, use another color or shading style to highlight the lines in the diagram that the address 13 will travel along on its way from the instruction memory to the main memory.

c. Finally, write a sequence of SLIM instructions that would make this data movement take place.

As mentioned above, a simulated SLIM computer is on the web site for this book. It is called SLIME, which stands for SLIM Emulator. SLIME has the functionality of an assembler built into it, so you can directly load in an assembly language program such as the preceding one, without needing to explicitly convert it from assembly language to machine language first. Once you have loaded in your program, you can run it in SLIME either by using the Start button to start it running full speed ahead or by using the Step button to step through the execution one instruction at a time. Either way, SLIME shows you what is going on inside the simulated computer by showing you the contents of the registers, memories, and program counter.

In designing SLIME, we needed to pin down what range of numbers the storage locations can hold, as described in the preceding sidebar. Our decision was to allow only integers in the range from -2^{31} through $2^{31} - 1$. Because we are only allowing integers, we made the `div` instruction (division) completely equivalent to `quo` (quotient). The two opcodes are different because other versions of SLIM might allow fractions or floating point numerals. Also, any arithmetic operation that would normally produce a result bigger than $2^{31} - 1$ or smaller than -2^{31} gets mapped into that range by adding or subtracting the necessary multiple of 2^{32}. This produces a result that is congruent to the real answer, modulo 2^{32}. For example, if you use SLIME to compute factorials, it will correctly report that $5! = 120$ but will falsely claim that 14! is 1,278,945,280; the real answer is larger than that by 20×2^{32}.

For another example of assembly language programming, suppose you want to write a program that reads in two numbers and then displays their product. This is accomplished by reading the input into two registers, putting their product into a third, and then writing it out:

```
read 1
read 2
mul 3, 1, 2
write 3
halt
```

Note that we didn't actually need to use the third register: we could have instead written `mul 2, 1, 2`, storing the result instead in register 2; we would then have to also change `write 3` to `write 2`. In a larger program, this might help us stay within 32 registers; it would only be possible, however, if the program didn't need to make any further use of the input value after the product has been calculated.

Exercise 11.2

Write a program that reads in two numbers and then displays the sum of their squares.

Now we only have one more kind of instruction, the instructions for jumping, or causing some instruction other than the one in the immediately following location in instruction memory to be executed next. SLIM follows tradition by having two kinds of jump instructions, *conditional jumps*, which under some conditions jump and other conditions *fall through* to the next instruction, and *unconditional jumps*, which jump under all circumstances. For simplicity, we've designed SLIM to only have a single conditional jump opcode: jump if equal to zero, or `jeqz`. The way this code is used is that `jeqz` *sourcereg*, *addressreg* will cause the computer to check to see if the *sourcereg* register contains zero or not. If it doesn't contain zero, execution falls through to the next instruction, but if it does contain zero, the contents of the *addressreg* register is used as the address in instruction memory at which the execution should continue. Because the comparison instructions, such as `slt`, use 0 for false and 1 for true, you can also think of `jeqz` as being a "jump when false" instruction. The unconditional jump, `j` *addressreg*, will always use the number stored in the *addressreg* register as the next instruction address.

The following simple program reads in two numbers and then uses conditional jumping to display the larger of the two. We include comments, written with a semicolon just like in Scheme:

```
read 1      ; read input into registers 1 and 2
read 2
sge 3, 1, 2 ; set reg 3 to 1 if reg 1 >= reg 2, otherwise 0
li 4, 7     ; 7 is address of the "write 2" instruction, for jump
jeqz 3, 4   ; if reg 1 < reg 2, jump to instruction 7 (write 2)
write 1     ; reg 1 >= reg 2, so write reg 1 and halt
halt
write 2     ; reg 1 < reg 2, so write reg 2 and halt
halt
```

Notice that we must figure out the instruction number (i.e., the address in instruction memory) of the jump target for the `jeqz` instruction, which is 7 (not 8) because we start at instruction number 0. We also need to load the 7 into a register (in this case register 4) because jump instructions take their jump target from a register (the *address register*) rather than as an immediate, constant value. The need to determine the address of some instructions is one factor that contributes to the difficulty of writing, and even more of understanding, assembly language programs. It is even worse if you need to modify a program because if your change involves adding or deleting instructions, you might well have to recalculate the addresses of all potential jump targets and change all references to those addresses. As you can imagine, this problem would make program modification very difficult indeed.

Another factor contributes to the difficulty of programming in assembly language, which also relates to numbers within a program. We reference the value in a register by its register number; thus, we write the instruction `sge 3, 1, 2` knowing that registers 1 and 2 contain the input values and register 3 will contain the value indicating which is larger. In a simple program, this is not much of a problem (especially if the comments are adequate), but you can probably imagine that this can make larger programs very hard to understand and nearly impossible to modify.

Both of these difficulties would be reduced if we had a way to use names to make our programs more understandable. Assemblers typically have such a capability; the one we have built into SLIME is no exception. Our assembler allows names to be assigned to registers and allows us to embed symbolic *labels* at points within our program; both types of names can be used within assembly language instructions. Thus, we could rewrite the program as follows:

```
   allocate-registers input-1, input-2
   allocate-registers comparison, jump-target

   read input-1
   read input-2
   sge comparison, input-1, input-2
   li jump-target, input-2-larger
   jeqz comparison, jump-target

   write input-1
   halt

input-2-larger:    ; an instruction label, referring to the
   write input-2   ; write input-2 instruction
   halt
```

We use blank lines to indicate the basic logical blocks within the program and indent all the lines except labels to make the labels more apparent.

The designation of register names is done using the `allocate-registers` lines, which instruct the assembler to choose a register number (between 0 and 31) for each of the names. The division into two separate `allocate-registers` lines is simply to avoid having one very long line. Either way, each name is assigned a different register number. The register names can be used exactly as register numbers would be, to specify the operands in assembly language instructions. Note that there is no guarantee as to which register number is assigned to a given name, and there is a limit of 32 names. In fact, if you use register names, *do not* refer to registers by number because you may be using the same register as a symbolically named one.

In addition to these names for register numbers, our assembler (like most) allows names to be given to instruction numbers by using labels within the program, such as `input-2-larger:`. The labels end with a colon to distinguish them from instructions. A label can be used as a constant would be, in an `li` instruction, as illustrated previously. Notice that the colon doesn't appear in the `li` instruction, just where the label is actually labeling the next instruction.

The key point to keep in mind about register names and instruction labels is that they are simply a convenient shorthand notation, designed to let the assembler do the counting for you. The two versions of the preceding program will be completely identical by the time they have been translated into machine language and are being executed by the machine. For example, the instruction label `input-2-larger` in the `li` instruction will have been replaced by the constant `7` in the course of the assembly process.

Exercise 11.3

The quadratic formula states that the roots of the quadratic equation $ax^2+bx+c=0$ (where $a \neq 0$) are given by the formula

$$\frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

Therefore, the equation will have 0, 1, or 2 real roots depending on whether b^2-4ac is < 0, $= 0$, or > 0.

Write an assembly language program that reads in three values (corresponding to a, b, and c) and writes out whether the equation $ax^2+bx+c=0$ has 0, 1, or 2 real roots.

Even with our ability to use names, assembly language programming is still excruciatingly detail-oriented, which is why we normally program in a language like Scheme instead. Even though SLIM (like real computers) can only execute instructions in its own machine language, we can still use Scheme to program it in either of two ways:

1. We can write a single assembly language program, namely, for a Scheme read-eval-print-loop, like the one we programmed in Scheme in the previous chapter. From then on, the computer can just run the result of assembling that one program, but we can type in whatever Scheme definitions and expressions we want. This is called using an *interpreter*.
2. We can write a program (in Scheme) that translates a Scheme program into a corresponding assembly language or machine language program. This is known as a *compiler*. Then we can use the compiler (and the assembler, if the compiler's

SLIM's Instruction Set

add *destreg*, $sourcereg_1$, $sourcereg_2$
sub *destreg*, $sourcereg_1$, $sourcereg_2$
mul *destreg*, $sourcereg_1$, $sourcereg_2$
div *destreg*, $sourcereg_1$, $sourcereg_2$
quo *destreg*, $sourcereg_1$, $sourcereg_2$
rem *destreg*, $sourcereg_1$, $sourcereg_2$

seq *destreg*, $sourcereg_1$, $sourcereg_2$
sne *destreg*, $sourcereg_1$, $sourcereg_2$
slt *destreg*, $sourcereg_1$, $sourcereg_2$
sgt *destreg*, $sourcereg_1$, $sourcereg_2$
sle *destreg*, $sourcereg_1$, $sourcereg_2$
sge *destreg*, $sourcereg_1$, $sourcereg_2$

ld *destreg*, *addressreg*
st *sourcereg*, *addressreg*

li *destreg*, *const*

read *destreg*
write *sourcereg*

jeqz *sourcereg*, *addressreg*
j *addressreg*

halt

output is assembly language) to translate our Scheme programs for execution by the computer.

For your convenience, a complete list of SLIM's instructions is given in a sidebar.

11.4 Iteration in Assembly Language

The previous sections described the capabilities of a computer by showing the structure of SLIM and enumerating the instructions it can carry out. We also wrote some simple programs in assembly language that used the naming and labeling capabilities of the assembler. In this section, we turn our attention to extending our programming skills by writing programs in SLIM's assembly language for carrying out iterative processes. We extend this skill to recursive processes in the next section.

You may recall that in Part I of this book we introduced recursion before iteration. This order was because in our experience students find many problems easier to solve using the recursion strategy rather than the iteration strategy. However, by now you should be experienced at solving problems both ways, and iterative solutions can be more naturally expressed in assembly language than can recursive solutions. Therefore, we've reversed the order of presentation here, starting with iteration and then moving on to recursion in the next section. The reason why it is straightforward to write assembly language programs that generate iterative processes is that iterative behavior is fairly easy to achieve through programming *loops* caused by jumps in the code.

Consider the simple problem of printing out the numbers from 1 to 10. One solution is described in the *flow chart* in Figure 11.6. The loop is visually apparent in the flow chart and is accomplished in assembly language as follows:

```
   allocate-registers count, one, ten, loop-start, done

   li count, 1
   li one, 1
   li ten, 10
   li loop-start, the-loop-start

the-loop-start:
   write count
   add count, count, one
   sgt done, count, ten
   jeqz done, loop-start

   halt
```

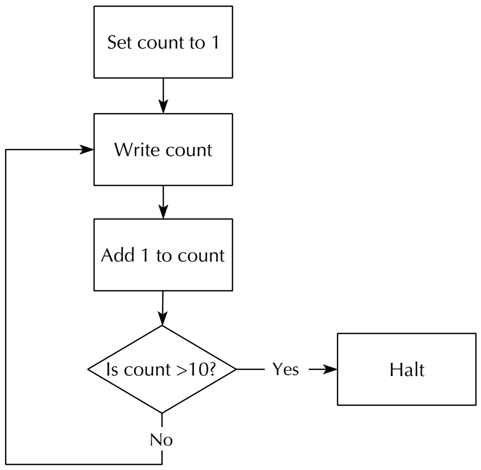

Figure 11.6 Flow chart for printing the numbers from 1 to 10

Before going on, be sure to compare the flow chart with the program and see how completely they parallel each other.

Having now written a simple iterative program, we can write more interesting programs, perhaps drawing from iterative procedures we have written in Scheme. For example, we can write a program to read in a number, iteratively calculate its factorial, and print out the result, similar to the following Scheme program:

```
(define factorial-product
  (lambda (a b) ; computes a * b!, provided b is a nonnegative integer
    (if (= b 0)
        a
        (factorial-product (* a b) (- b 1)))))

(define factorial
  (lambda (n)
    (factorial-product 1 n)))
```

Just as this Scheme program has a comment explaining what the `factorial-product` procedure does, so too our assembly language version has a comment saying what we can expect to happen when execution reaches the instruction labeled `factorial-product-label`:

```
   allocate-registers a, b, one, factorial-product, end

   li a, 1
   read b
   li one, 1
   li factorial-product, factorial-product-label
   li end, end-label

factorial-product-label:
   ;; computes a * b! into a and then jumps to end
   ;; provided that b is a nonnegative integer;
   ;; assumes that the register named one contains 1 and
   ;; the factorial-product register contains this address;
   ;; may also change the b register's contents
   jeqz b, end  ; if b = 0, a * b! is already in a

   mul a, a, b   ; otherwise, we can put a * b into a
   sub b, b, one ; and b - 1 into b, and start the
   j factorial-product            ; iteration over

end-label:
   write a
   halt
```

Exercise 11.4

Translate into SLIM assembly language the procedure for raising a base to a power given in Section 3.2.

Exercise 11.5

SLIME has a counter that shows how many instructions have been executed. This counter can be used to carefully compare the efficiency of different algorithms. Translate into SLIM assembly language the following alternative `power-product` procedure and compare its efficiency with that of your program from the preceding exercise, with increasingly large exponents. (*Hint:* You'll need an extra register in which to store the remainder of *e* divided by 2. You'll also need one more label because the `cond` has three cases; another register to hold the numeric value of that label will also come in handy.) You should be able to predict the instruction counts by carefully analyzing your programs; that way the simulator's instruction counts can serve as empirical verification of your prediction, showing that you have correctly understood the programs.

```
(define power-product
  (lambda (a b e)   ; returns a times b to the e power
    (cond ((= e 0) a)
          ((= (remainder e 2) 0)
           (power-product a (* b b) (/ e 2)))
          (else (power-product (* a b) b (- e 1))))))
```

Exercise 11.6

Translate into SLIM assembly language your procedure for finding the exponent of 2 in a positive integer, from Exercise 3.2.

Exercise 11.7

Translate into SLIM assembly language the procedure for finding a Fermat number by repeated squaring given in Section 3.2.

One aspect of the iterative factorial program to note carefully is the order of the multiplication and subtraction instructions. Because the multiplication is done first, the old value of *b* is multiplied into *a*; only afterward is *b* reduced by 1. If the order of these two instructions were reversed, the program would no longer compute the correct answer. In Scheme terms, the correct version of the SLIM program is like the following Scheme procedure:

```
(define factorial-product
  (lambda (a b) ; computes a * b!, given b is a nonnegative integer
    (if (= b 0)
        a
        (let ((a (* a b)))
          (let ((b (- b 1)))
            (factorial-product a b))))))
```

If the multiplication and subtraction were reversed, it would be like this (incorrect) Scheme procedure:

```
(define factorial-product ; this version doesn't work
  (lambda (a b) ; computes a * b!, given b is a nonnegative integer
    ;;(continued)
    (if (= b 0)
        a
        (let ((b (- b 1)))
          (let ((a (* a b))) ; note that this uses the new b
            (factorial-product a b))))))
```

In the `factorial-product` procedure, the new value for `b` is calculated without making use of the (old) value of `a`, so we can safely "clobber" `a` and then compute `b`. Other procedures may not be so lucky in this regard; there may be two arguments where each needs to have its new value computed from the old value of the other one. In these cases, it is necessary to use an extra register to temporarily hold one of the values. For example, consider translating into the SLIM assembly language the following Scheme procedure for computing a greatest common divisor:

```
(define gcd
  (lambda (x y)
    (if (= y 0)
        x
        (gcd y (remainder x y)))))
```

Here the new value of `x` is computed using the old value of `y` (in fact, it is simply the same as the old value of `y`), and the new value of `y` is computed using the old value of `x`; thus, it appears neither register can receive its new value first because the old value of that register is still needed for computing the new value of the other register. The solution is to use an extra register; we can model this solution in Scheme using `let`s as follows:

```
(define gcd
  (lambda (x y)
    (if (= y 0)
        x
        (let ((old-x x))
          (let ((x y))                      ; x changes here
            (let ((y (remainder old-x y))) ; but isn't used here
              (gcd x y)))))))
```

Exercise 11.8

Translate `gcd` into a SLIM assembly language program for reading in two numbers and then computing and writing out their greatest common divisor. (*Hint:* To copy a value from one register to another, you can add it to zero.)

Returning to the iterative factorial program, another more subtle point to note is that the comment at the `factorial-product-label` specifies the behavior that will result in terms of the values in three kinds of registers:

1. The `a` and `b` registers correspond to the arguments in the Scheme procedure. These control which specific computation the loop will carry out, within the range of computations it is capable of.
2. The `one` register, which is assumed to have a 1 in it whenever execution reaches the label, has no direct analog in the Scheme procedure. This register isn't intended to convey information into the loop that can be varied to produce varying effects, the way the `a` and `b` registers can. Instead, it is part of the specification for efficiency reasons only. Instead of requiring that a 1 be in the `one` register whenever execution reaches the label, it would be possible to load the 1 in after the label. However, that would slow the program down because the loading would be needlessly done each time around the loop. The same considerations apply to the `factorial-product` register, which also holds a constant value, the starting address of the loop.
3. The `end` register is perhaps the most interesting of all. It is what we call a *continuation register* because it holds the *continuation address* for the `factorial-product` procedure. That is, this register holds the address that execution should continue at after the `factorial-product` computation is completed. Once the computer has finished computing $a \times b!$, it jumps to this continuation address, providing another opportunity to control the behavior of this looping procedure, as we'll see shortly. Namely, in addition to varying what numbers are multiplied, we can also vary where execution continues afterward.

To see how we would make more interesting use of a continuation register, consider writing a procedure for computing $n! + (2n)!$ as follows:

```
(define factorial-product ; unchanged from the above
  (lambda (a b) ; computes a * b!, given b is a nonnegative integer
    (if (= b 0)
        a
        (factorial-product (* a b) (- b 1)))))

(define two-factorials
  (lambda (n)
    (+ (factorial-product 1 n)
       (factorial-product 1 (* 2 n)))))
```

Clearly something different should happen after `factorial-product` is done with its first computation than after it is done with its second computation. After the first computation, it is still necessary to do the second computation, whereas after the second computation, it is only necessary to do the addition and write the result out. So, not only will different values be passed in the `b` register, but different values will be passed in the `end` continuation register as well:

```
   allocate-registers a, b, one, factorial-product
   allocate-registers end, n, result, zero

   li one, 1
   li zero, 0
   li factorial-product, factorial-product-label
   read n
   li a, 1
   add b, zero, n ; copy n into b by adding zero
   li end, after-first ; note continuation is after-first

factorial-product-label:
   ;; computes a * b! into a and then jumps to end
   ;; provided that b is a nonnegative integer;
   ;; assumes that the register named one contains 1 and
   ;; the factorial-product register contains this address;
   ;; may also change the b register's contents
   jeqz b, end  ; if b = 0, a * b! is already in a

   mul a, a, b  ; otherwise, we can put a * b into a
   sub b, b, one ; and b - 1 into b, and start the
   j factorial-product      ; iteration over

after-first:
   add result, zero, a   ; save n! away in result
   li a, 1
   add b, n, n           ; and set up to do (2n)!,
   li end, after-second  ; continuing differently after
   j factorial-product   ; this 2nd factorial-product,

after-second:            ; namely, by
   add result, result, a ; adding (2n!) in with n!
   write result          ; and displaying the sum
   halt
```

To understand the role that the `n` and `result` registers play in the two-factorials program, it is helpful to contrast it with the following double-factorial program. If we refer to the value the double-factorial program reads in as n, what it is computing and displaying is $(n!)!$:

```
   allocate-registers a, b, one, factorial-product, end, zero

   li one, 1
   li zero, 0
   li factorial-product, factorial-product-label
   li a, 1
   read b ; the first time, the read-in value is b
   li end, after-first ; and the continuation is after-first

factorial-product-label:
   ;; computes a * b! into a and then jumps to end
   ;; provided that b is a nonnegative integer;
   ;; assumes that the register named one contains 1 and
   ;; the factorial-product register contains this address;
   ;; may also change the b register's contents
   jeqz b, end  ; if b = 0, a * b! is already in a

   mul a, a, b  ; otherwise, we can put a * b into a
   sub b, b, one ; and b - 1 into b, and start the
   j factorial-product     ; iteration over

after-first:
   add b, zero, a ; move the factorial into b by adding zero
   li a, 1        ; so that we can get the factorial's factorial
   li end, after-second ; continuing differently after
   j factorial-product  ; this second factorial-product,

after-second:          ; namely, by
   write a             ; displaying the result
   halt
```

This latter program reads the n value directly into the `b` register, ready for computing the first factorial. The earlier two-factorials program, in contrast, read n into a separate `n` register and then copied that register into `b` before doing the first factorial. The reason why the two-factorials program can't read the input value directly into `b` the way double-factorial does is that it will need the n value again, after $n!$ has been computed, to compute $(2n)!$. Therefore, this n value needs to be stored somewhere "safe" while the first factorial is being computed. The `b` register isn't a safe place because the factorial-product loop changes that register (as its comment warns). Thus, a separate `n` register is needed.

The `result` register is needed for a similar reason, to be a safe holding place for $n!$ while $(2n)!$ is being computed; clearly the result of $n!$ can't be left in the `a` register while the second factorial is being computed. In double-factorial, on the other hand, the result of $n!$ isn't needed after $(n!)!$ is computed, so it doesn't need to be saved anywhere.

Exercise 11.9

Write a SLIM program for reading in four numbers, x, y, n, and m, and computing $x^n + y^m$ and displaying the result. Your program should reuse a common set of instructions for both exponentiations.

To review, we've learned two lessons from the two-factorials program:

1. If a procedure within the program is invoked more than once, a continuation register can be used to make the procedure continue differently when it is done with one invocation than when it is done with another.
2. If a value needs to be preserved across a procedure invocation, it shouldn't be stored in a register that will be clobbered (i.e., stored into) by the procedure. Instead, the value should be moved somewhere "safe," a location *not* stored into by the procedure.

11.5 Recursion in Assembly Language

In the previous section, we wrote assembly language procedures that generated iterative processes. Along the way, we learned two important lessons: the use of a continuation register and the importance of choosing a safe location for values that must be preserved across a procedure invocation. With these two lessons in mind, it is time to consider recursive processes. Sticking with factorials, we'll use the following Scheme procedure as our starting point:

```
(define factorial
  (lambda (n)
    (if (= n 0)
        1
        (* (factorial (- n 1))
           n))))
```

Consider using a SLIM program based on this procedure to compute 5! by computing 4! and then multiplying the result by 5. What needs to be done after the factorial procedure has finished computing 4!? What needs to be done after the factorial procedure has finished computing 5!? Are these the same? This should bring to mind the first of the two lessons from two-factorials: A continuation register can be used to make the factorial procedure continue differently after computing 4! than after computing 5!. After computing 4!, the computer needs to multiply the result by 5, whereas after computing 5!, the computer needs to display the final result. Tentatively, then, we'll assume we are going to use three registers: an `n` register for the argument to the factorial procedure, a `cont` register for the procedure's continuation address, and a `val` register to hold the resulting value of *n*!.

Next question: Do any values need preserving while the computation of 4! is underway? Yes, the fact that *n* is 5 needs to be remembered so that when 4! has been computed as 24, the computer knows to use 5 as the number to multiply 24 by. The computer also needs to save the 5! computation's continuation address across the 4! computation so that once it has multiplied 24 by 5 and gotten 120, it knows what to do next. It isn't obvious that the continuation will be to display the result—the computation of 5! *might* have been as a step in computing 6!. So, the continuation address needs to be preserved as the source of this information on how to continue. Thus, two values must be preserved across the recursive factorial subproblem: the main factorial problem's value of *n* and the main factorial problem's continuation address.

The second of the two lessons we learned from two-factorials leads us to ask: What are safe locations to hold these values so they aren't overwritten? Clearly the `n` register is not a safe place to leave the value 5 while computing 4!, because in order to compute 4!, we'll store 4 into `n`. Similarly, the `cont` register is not a safe place to leave the continuation address for the computation of 5! while the computation of 4! is underway, because the continuation address for the 4! computation will be stored there. Should we introduce two more registers to hold the main problem's *n* value and continuation address while the subproblem uses the `n` and `cont` registers?

If there were only two levels of procedure invocation—the main problem and the subproblem—the proposed solution of using two more registers would be reasonable. Unfortunately, the subproblem of computing 4! itself involves the sub-subproblem of computing 3!, which involves the sub-sub-subproblem of computing 2!, and so forth down to the base case. Each level will have two values to preserve, but we can't use two registers per level; among other things we only have 32 registers total, so we'd never be able to compute 52! if we used two registers per level.

The need for two safe storage locations per level (i.e., two locations that won't be stored into by the other levels) is real. So, having seen that registers won't suffice, we turn to our other, more plentiful, source of storage locations, the data memory. The top-level problem can store its *n* value and continuation address into the first two memory locations for safekeeping (i.e., it can store them at addresses 0 and 1). The subproblem would then similarly use the next two memory locations, at addresses 2 and 3, the sub-subproblem would use addresses 4 and 5, etc. Because each level uses different locations, we cannot clobber values, and because the memory is large, the maximum recursion depth attainable, although limited, will be sufficient for most purposes. (You may have noticed that this example is our first use of memory. In the following section and subsequent chapters we'll see other uses, so recursion isn't the only reason for having memory. It is one important reason, however.)

To keep track of how much of the memory is already occupied by saved values, we'll use a register to hold the number of locations that are in use. When a procedure starts executing, this register's value tells how much of memory should be left untouched and also tells which memory locations are still available for use. If the register holds 4, that means that the first four locations should be left alone, but it also means that 4 is the first address that is up for grabs, because the four locations in use are at addresses 0 through 3.

The procedure can therefore store its own values that need safekeeping into locations 4 and 5; it should increase the memory-in-use register's value by 2 to reflect this fact. When the procedure later retrieves the values from locations 4 and 5, it can decrease the count of how many memory locations are in use by 2. Thus, when the procedure exits, the memory-in-use register is back to 4, the value it had on entry.

This very simple idea of having procedures "clean up after themselves," by deallocating the memory locations they've allocated for their own use, is known as *stack discipline*. (When we speak of allocating and deallocating memory locations, we're referring to increasing and decreasing the count of how many locations are in use.) The reason for the name *stack discipline* is that the pattern of growth and shrinkage in the memory's use is like piling things up on a stack and then taking them off. The most recently piled item is on top of the stack, and that is the one that needs to be taken off first. So too with the *stack* in the computer's memory; locations 0 and 1 were allocated first, then 2 and 3 "on top" of those, and then 4 and 5 "on top" of those. Now, the first locations to be deallocated are 5 and 4—the stack shrinks from the top. Computer scientists traditionally refer to putting items on a stack as *pushing* onto the stack and removing items from the stack as *popping*. The register that records how much of the memory is currently occupied by the stack is known as the *stack pointer*, or *SP*. We'll use the register name `sp` in the program below. The stack pointer is a procedure's indication of what locations in memory to use for its saved values, as in the following recursive factorial program:

```
allocate-registers n, cont ; the argument, continuation,
allocate-registers val     ; and result of factorial procedure
allocate-registers factorial, base-case ; hold labels' values
allocate-registers sp ; the "stack pointer", it records how many
                      ; memory locations are occupied by saved
                      ; values (starting at location 0)
allocate-registers one ; the constant 1, used in several places
```

```
   ;; set up the constants
   li one, 1
   li factorial, factorial-label
   li base-case, base-case-label
   ;; initialize the stack pointer (nothing saved yet)
   li sp, 0
   ;; set up for the top level call to factorial
   read n  ; the argument, n, is read in
   li cont, after-top-level ; the continuation is set
   ;; and then we can fall right into the procedure

factorial-label:
   ;; computes the factorial of n into val and jumps to cont;
   ;;  doesn't touch the first sp locations of memory and
   ;;  restores sp back to its entry value when cont is jumped to;
   ;;  assumes the factorial, base-case, and one registers hold the
   ;;  constant values established at the beginning of the program
   jeqz n, base-case

   ;; if n isn't zero, we save n and cont into memory for
   ;; safe keeping while computing (n-1)!; sp tells us where in
   ;; memory to save them (so as not to clobber other, previously
   ;; saved values), and we adjust sp to reflect the new saves
   st n, sp
   add sp, sp, one
   st cont, sp
   add sp, sp, one
   ;; now that we're done saving, we can set up for (n-1)!
   sub n, n, one  ; using n-1 as the new n argument
   li cont, after-recursive-invocation ; the continuation
   j factorial                         ;  after this call

after-recursive-invocation:            ;  is down here
   ;; having made it through the recursive call, the saved
   ;; values of cont and n can be restored to their registers
   ;; from memory; note that they are "popped" from the stack
   ;; in the opposite order they were "pushed" onto the stack,
   ;; since the second one pushed wound up "on top" (i.e., later
   ;; in memory), so should be retrieved first
   sub sp, sp, one
   ld cont, sp
   sub sp, sp, one
   ld n, sp
   ;; having retrieved n and cont and set sp back to the way it
   ;; was on entry (since it went up by two and back down by two)
   ;; we are ready to compute n! as (n-1)! * n, i.e. val * n,
   ;; putting the result into val, and jump to the continuation
   mul val, val, n
   j cont

base-case-label:
   ;; this is the n = 0 case, which is trivial
   li val, 1
   j cont

after-top-level:
   ;; when the top level factorial has put n! in val, it jumps here
   write val ; to display that result
   halt
```

Exercise 11.10

Write a SLIM program based on the recursive `power` procedure you wrote in Exercise 2.1 on page 28. Try not to save any more registers to the stack than are needed. You should use SLIME to compare the efficiency of this version with the two iterative versions you wrote in Exercises 11.4 and 11.5. (As before, it should be possible for you to predict the instruction counts in advance by analyzing the programs; the simulator can then serve to verify your prediction.)

Exercise 11.11

Write a SLIM program based on your procedure for recursively computing the sum of the digits in a number; you wrote this procedure in Exercise 2.11 on page 39.

Exercise 11.12

Write a SLIM program based on your procedure for recursively computing the exponent of 2 in a number; you wrote this procedure in Exercise 2.12 on page 40.

11.6 Memory in Scheme: Vectors

In the previous sections of this chapter we've looked "under the hood" at computer architecture and assembly language programming. This introduction is valuable in its own right because it provided you with a clearer understanding of how computation actually happens. However, it had an important side benefit as well: We encountered a new kind of programming, based on sequentially retrieving values from locations, computing new values from them, and storing the new values back into the locations. This style is known as the *imperative style* or *imperative paradigm* of programming because each instruction is issuing a command to the computer to carry out a particular action. The imperative style is closely tied to the concept of *state* (i.e., that actions can change something about the computer that affects future actions). It is what ties together the disjointed sequence of commands into a purposeful program—the fact that each changes something that future instructions examine. In SLIM, as in most computers today, the primary form of state is the storage locations.

In the following chapters we'll see some interesting applications of state and imperative programming. However, before we do so, it is worth recapturing what we lost in moving from Scheme to assembly language. As an example of what we've lost, consider storing $(x + y) \times (z + w)$ into some memory location. In Scheme, we could express the product of sums as exactly that, a product of sums: `(* (+ x y) (+ z w))`. However, we haven't yet seen any way to store the resulting value into a location. (That's about to change.) In assembly language, on the other hand, we'd have no problem storing the result into a location, but we'd also be forced to store the component sums into locations, whether we wanted to or not. We'd need to compute the first sum and store it somewhere—and we'd have to pick the location to store it into. Then we could do the second sum and again store it somewhere we'd have to choose. Finally, we could compute the product we wanted and store it as desired. The result is that what was a natural nesting of computations got painstakingly *linearized* into a sequence of steps, and the new ability to store values into locations spread like a cancer into even those parts of the computation where it didn't naturally belong.

The ability to nest computations (like the product of sums) and avoid storing intermediate results is a large part of what makes a higher-level programming language like Scheme so much more convenient than assembly language. Nesting of computations works naturally when the computations correspond to mathematical functions, which compute result values from argument values. The programming we've done in Scheme up until now has all had this *functional* character—we say that we were programming in the *functional style* or *functional paradigm*. In functional programming the natural way to combine computations is through structured nesting, whereas in imperative programming, the natural means of combination is through linear sequencing. An important open research topic in computer programming language design is to find natural ways to use richer structures of combination with state. For now, we'll tackle a simpler goal: integrating the two styles of programming so that the *value-oriented* portions of a program can use the full power of functional programming, whereas the *state-oriented* portions will have an imperative flavor. To do this, we introduce memory into Scheme.

Chunks of memory in Scheme are called *vectors*. Each vector has a size and contains that many memory locations. Vectors are made by the `make-vector` procedure, which must be told how big a vector to make. For example `(make-vector 17)` will make a vector with 17 locations in it. The locations are numbered starting from 0 in each vector, so the locations in the example vector would be numbered from 0 to 16. A simple example follows of creating and using a vector; it shows how values can be stored into and retrieved from the vector's locations and how the size of a vector can be determined:

```
(define v (make-vector 17))

(vector-length v)    ; find out how many locations
17

(vector-set! v 13 7) ; store a 7 into location 13

(vector-ref v 13)    ; retrieve what's in location 13
7

(vector-set! v 0 3)  ; put a 3 into the first location (location 0)

(vector-ref v 13)    ; see if location 13 still intact
7

(vector-set! v 13 0) ; now clobber it

(vector-ref v 13)    ; see that location 13 did change
0
```

Notice that the procedure for changing the contents of one of a vector's locations is called `vector-set!`, with an exclamation point at the end of its name. This is an example of a convention that we generally follow, namely, that procedures for changing an object have names that end with an exclamation point. Just as with the question mark at the end of a predicate's name, this naming convention is purely for better communication among humans; the Scheme language regards the exclamation point or question mark as no different than a letter. Another point to notice is that we didn't show any value for the evaluations that applied `vector-set!`. This is because the definition of the Scheme programming language leaves this value unspecified, so it can vary depending on the particular Scheme system you are using. Your particular system might well return something useful (like the old value that was previously stored in the location), but you shouldn't make use of that because doing so would render your programs nonportable.

As an example of how a vector could be used, consider making a *histogram* of the grades the students in a class received on an exam. That is, we'd like to make a bar chart with one bar for each grade range, where the length of the bar corresponds to how many students received a grade within that range. A simple histogram might look as follows:

```
90-99: XXXXXXXX
80-89: XXXXXXXXXX
70-79: XXXXXXX
60-69: XXX
50-59: XXXXX
40-49: XXX
30-39: XX
20-29:
10-19: X
00-09:
```

This histogram shows, for example, that two students received grades in the 30s; one X appears for each student in the grade range.

The obvious way to make a histogram like this one is to go through the students' grades one by one, keeping 10 counts, one for each grade range, showing how many students have been discovered to be in that range. Initially each count is 0, but as a grade of 37 is encountered, for example, the counter for the 30s range would be increased by 1. At the end, the final counts are used in printing out the histogram to determine how many Xs to print in each row. The fact that the counts need to change as the grades are being processed sounds like storage locations; the fact that we need 10 of them sounds like we need a vector of length 10. So, we have a plan for our program:

1. Make a vector of length 10.
2. Put a 0 into each location in the vector.
3. Read in the grades one by one. For each, increment the appropriate location or "bin" within the vector.
4. Display the vector as a histogram.

Writing this in Scheme, we get the following:

```
(define do-grade-histogram
  (lambda ()
    (let ((histogram (make-vector 10)))
      (define read-in-grades-loop
        (lambda ()
          (let ((input (read)))
            ;;(continued)
            (if (equal? input 'done)
                'done-reading
                (let ((bin (quotient input 10)))
                  (vector-set! histogram bin
                               (+ 1 (vector-ref histogram bin)))
                  (read-in-grades-loop)))))) ;end of loop
      (zero-out-vector! histogram) ;start of main procedure
      (newline)
      (display
       "Enter grades in the range 0 - 99; enter done when done.")
      (newline)
      (read-in-grades-loop)
      (display-histogram histogram))))
```

This relies on two other procedures: `zero-out-vector!` puts the initial 0 into each location, and `display-histogram` displays the vector as a histogram. They can be written as follows:

```
(define zero-out-vector!
  (lambda (v)
    (define do-locations-less-than
      (lambda (limit)
        (if (= limit 0)
            'done
            (let ((location (- limit 1)))
              (vector-set! v location 0)
              (do-locations-less-than location)))))
    (do-locations-less-than (vector-length v))))

(define display-histogram
  (lambda (histogram)
    (define display-row
      (lambda (number)
        (display number)
        (display "0-")
        (display number)
        (display "9: ")
        ;; display-times from page 313 useful here
        (display-times "X" (vector-ref histogram number))
        (newline)))
    ;;(continued)
    (define loop
      (lambda (counter)
        (if (< counter 0)
            'done
            (begin (display-row counter)
                   (loop (- counter 1))))))
    (newline)
    (loop 9)
    (newline)))
```

Exercise 11.13

Some students earn grades of 100 for their exams, rather than just 0 to 99. There is no one clearly right way to modify the histograms to accommodate this. Consider some of the options, choose one, justify your choice, and implement it.

The previous program used one X per student. This works for those of us fortunate enough to have small classes, but in a large course at a large university, some of the bars of Xs would no doubt run off the edge of the computer's screen. This problem can be resolved by scaling the bars down so that each X represents 10 students instead of 1, for example. The scaling factor can be chosen automatically to make the longest bar fit on the screen. For example, we could choose as the number of students per X the smallest positive integer that makes the longest bar no longer than 70 Xs. Here is a version of `display-histogram` that does this:

```
(define maximum-bar-size 70)

(define display-histogram
  (lambda (hist)
    (let ((scale (ceiling  ; i.e., round up to an integer
                  (/ (largest-element-of-vector hist)
                     maximum-bar-size))))
      (define display-row
        (lambda (number)
          (display number)
          (display "0-")
          (display number)
          (display "9: ")
          (display-times "X" (quotient
                              (vector-ref hist number)
                              scale))
          (newline)))
      ;;(continued)
      (define loop
        (lambda (counter)
          (if (< counter 0)
              'done
              (begin (display-row counter)
                     (loop (- counter 1))))))
      (newline)
      (display "Each X represents ")
      (display scale)
      (newline)
      (loop 9)
      (newline))))
```

Exercise 11.14

Write the `largest-element-of-vector` procedure that it takes to make this work.

11.7 An Application: A Simulator for SLIM

Because vectors in Scheme are so similar to SLIM's memory and registers, we can use vectors to build a *model* of SLIM as the core of a Scheme program that simulates the execution of SLIM programs. In this section, we'll build such a simulator. It won't be as fancy as SLIME, but it will suffice to execute any SLIM program. Our ultimate goal is to write a procedure called `load-and-run` that executes a given machine language program, but of course we'll write a lot of other procedures along the way. The `load-and-run` procedure will receive the program to run as a vector of machine language instructions; later we'll see how those instructions can be constructed.

In attacking a project as large as the SLIM simulator, it is helpful to divide it up into separate *modules* and understand the interfaces between the modules first, before actually doing any of the programming. For our simulator, we'll use three modules (one of which, the instructions module, is shared with the assembler):

1. The first module provides an abstract data type called the *machine model*. A machine model keeps track of the state of the simulated machine, which is where the contents of the simulated machine's registers, memory, and program counter are stored. Whether the machine is in the special halted state is also stored here.

   More specifically, this module provides the rest of the program with a `make-machine-model` procedure, which can be used to make a new machine model, with all the locations holding 0 and with the machine not halted. This module then allows the rest of the program to inspect and modify the state of that machine model by using procedures such as `get-pc` for getting the current contents of the program counter and `set-pc!` for changing the program counter's contents. Other interface procedures are `get-reg` and `set-reg!` for registers, `get-mem` and `set-mem!` for memory, and `halted?` and `halt!` for haltedness. All the procedures except `make-machine-model` take a machine model as their first argument. The procedures that concern registers and memory take a register number or memory address as the second argument. All the `set-`... `!` procedures take the new value as the final argument. For example, `set-reg!` takes a machine model, a register number, and a new value for the register as its arguments.

2. The instructions module provides the assembler with constructors for each kind of machine language instruction. It also provides the simulator with a `do-instruction-in-model` procedure, which takes one of the machine language instructions and a machine model and carries out the effects of that instruction in the model. Notice that nothing outside of this module needs to care what the encoding of a machine language instruction is. Also, this module's `do-instruction-in-model` procedure doesn't need to care about the representation for machine models because it can use the access and updating procedures previously described. Part of the effect of every instruction on a model is to update that model's program counter, using `set-pc!`. This effect exists even for nonjumping instructions, which set the PC to 1 more than its previous value.

   The instruction constructors are `make-load-inst`, `make-store-inst`, `make-load-immediate-inst`, `make-add-inst`, `make-sub-inst`, `make-mul-inst`, `make-div-inst`, `make-quo-inst`, `make-rem-inst`, `make-seq-inst`, `make-sne-inst`, `make-slt-inst`, `make-sgt-inst`, `make-sle-inst`, `make-sge-inst`, `make-jeqz-inst`, `make-jump-inst`, `make-read-inst`, `make-write-inst`, and `make-halt-inst`. Each of these takes one argument per operand specifier in the same order as the operand specifiers appear in the assembly language instruction. For example, `make-load-inst` takes two arguments because load instructions have two operand specifiers. The two arguments in this case are the register numbers. (They must be numbers, not names.) For `make-load-immediate-inst`, the second argument must be the actual numeric constant value.

3. Finally, there is the main module that provides the `load-and-run` procedure. It makes heavy use of the services provided by the other two modules, concerning itself primarily with the overall orchestration of the execution of the simulated program. Its argument is a vector of instructions; at each step (so long as the machine isn't halted), it retrieves from this vector the instruction addressed by the model's program counter and does that instruction in the model. Once the model indicates that the machine has halted, the `load-and-run` procedure returns a count of how many instructions were executed.

The main module is the simplest, so let's start there. It needs only to provide the `load-and-run` procedure. Having identified that the machine model module makes the `make-machine-model`, `halted?` and `get-pc` procedures available, and that the instructions module makes the `do-instruction-in-model` procedure available, we can write `load-and-run` as follows:

```
(define load-and-run
  (lambda (instructions)
    (let ((model (make-machine-model))
          (num-instructions (vector-length instructions)))
      (define loop
        (lambda (instructions-executed-count)
          (if (halted? model)
              instructions-executed-count
              (let ((current-instruction-address (get-pc model)))
                (cond
                 ((= current-instruction-address num-instructions)
                  (error "Program counter ran (or jumped) off end"))
                 ((> current-instruction-address num-instructions)
                  (error
                   "Jump landed off the end of program at address"
                   current-instruction-address))
                 (else
                  (do-instruction-in-model
                   (vector-ref instructions
                               current-instruction-address)
                   model)
                  (loop (+ instructions-executed-count 1))))))))
      (loop 0))))
```

Either of the other two modules could be done next or they could even be done simultaneously by two different programmers. In this textbook, we choose to focus on the machine model module first. Recall that it provides a constructor, `make-machine-model`, and various procedures for examining and updating machine models (we call these latter procedures *selector* and *mutator* procedures, respectively). A machine model needs to contain models of the machine's memory, registers, and the two miscellaneous items of state, the program counter and the haltedness indicator. The obvious representation for the memory is as a vector that is as large as the simulated memory. Similarly, it seems natural to use a vector of length 32 to model the machine's bank of 32 registers. We'll lump the other two pieces of state into a third vector, of length 2; this leads to the following constructor:

```
(define mem-size 10000)
(define reg-bank-size 32)
```

```
(define make-machine-model
  (lambda ()
    (let ((memory (make-vector mem-size))
          (registers (make-vector reg-bank-size))
          (misc-state (make-vector 2)))
      (zero-out-vector! memory)
      (zero-out-vector! registers)
      (vector-set! misc-state 0 0) ; PC = 0
      (vector-set! misc-state 1 #f) ; not halted
      (list memory registers misc-state))))
```

This constructor produces machine models that are three element lists, consisting of the memory vector, the registers vector, and the miscellaneous state vector. This latter vector has the PC in location 0 and the haltedness is location 1. Using this information, we can now write the four selectors and four mutators. Here, for example, are the selector and mutator for the registers:

```
(define get-reg
  (lambda (model reg-num)
    (vector-ref (cadr model) reg-num)))

(define set-reg!
  (lambda (model reg-num new-value)
    (vector-set! (cadr model) reg-num new-value)))
```

Exercise 11.15

Write the remaining selectors and mutators: `get-pc`, `set-pc!`, `halted?`, `halt!`, `get-mem`, and `set-mem!`.

Moving to the instructions module, we could choose from many possible representations for machine language instructions. Some would be more realistic if the machine language were actually to be loaded into a hardware SLIM, built from silicon and copper. Here we'll cop out and use a representation that makes the simulator easy to develop. If the representation used by this module were changed, both the assembler (which uses this module's instruction constructors) and the main part of the simulator would remain unchanged; therefore, we can afford to cop out now, knowing that the decision is reversible if we ever get serious about building hardware.

Specifically, we'll represent each machine language instruction as a procedure for suitably updating the machine model, when passed that model as its argument. In other words, we'll have a trivial `do-instruction-in-model` :

```
(define do-instruction-in-model
  (lambda (instruction model)
    (instruction model)))
```

Even though this pushes off all the real work to the instruction constructors, they aren't especially hard to write either, thanks to the support provided by the machine model module. Here is an example:

```
(define make-load-inst
  (lambda (destreg addressreg)
    (lambda (model)
      (set-reg! model
                destreg
                (get-mem model
                         (get-reg model addressreg)))
      (set-pc! model (+ (get-pc model) 1)))))
```

Exercise 11.16

Write the other instruction constructors: `make-store-inst`, `make-load-immediate-inst`, `make-add-inst`, `make-sub-inst`, `make-mul-inst`, `make-div-inst`, `make-quo-inst`, `make-rem-inst`, `make-seq-inst`, `make-sne-inst`, `make-slt-inst`, `make-sgt-inst`, `make-sle-inst`, `make-sge-inst`, `make-jeqz-inst`, `make-jump-inst`, `make-read-inst`, `make-write-inst`, and `make-halt-inst`.

At this point, you should be able to try out the simulator. Here is an example that avoids using the assembler; it is the program that we presented in Section 11.3 as our first complete program, the one for displaying 314 and then halting:

```
(let ((instructions (make-vector 3)))
  (vector-set! instructions 0
               (make-load-immediate-inst 1 314))
  (vector-set! instructions 1
               (make-write-inst 1))
  (vector-set! instructions 2
               (make-halt-inst))
  (load-and-run instructions))
```

Of course, there is no need to make all your instructions by hand this way, when an assembler can do the work for you. Writing an assembler in Scheme that could read in the exact assembly notation we showed earlier would distract us with various

```
(define write-larger
  (assemble
   '((allocate-registers input-1 input-2 comparison jump-target)

     (read input-1)
     (read input-2)
     (sge comparison input-1 input-2)
     (li jump-target input-2-larger)
     (jeqz comparison jump-target)

     (write input-1)
     (halt)

     input-2-larger
     (write input-2)
     (halt))))
```

Figure 11.7 Assembling a program using our variant notation. This definition would result in `write-larger` being a vector of machine language instructions; you could then do `(load-and-run write-larger)`. Note in particular that labels don't end with colons.

messy details, like how to strip the colon off the end of a label. On the other hand, if we are willing to accept the variant notation illustrated in Figure 11.7, the assembler becomes a straightforward application of techniques from earlier chapters, such as the pattern/action list. An assembler written in this way is included in the software on the web site for this book. We won't include a printed version of it here, but you might be interested in looking at it on your computer.

Review Problems

Exercise 11.17

Suppose you wanted to make SLIM cheaper to build by eliminating some of the six comparison instructions.

a. If you were only willing to modify your programs in ways that didn't make them any longer, how many of the comparison operations could you do without? Explain.

b. Suppose you were willing to lengthen your programs. Now how many of the comparison operations do you really need? Explain.

Exercise 11.18

Write a SLIM program to read in one or more single-digit numbers, followed by a negative number to indicate that the digits are over. The program should then write out the number those digits represent, treated as a decimal integer. The first digit read in should be the most significant digit. So, for example, if the program reads in 3, 1, 4, and then -1, it should output the number 314. If you had a machine similar to SLIM but (more realistically) only able to input a single character at a time, this is how you would have to input numbers.

Exercise 11.19

Write a SLIM program to read a nonnegative integer in and then display its digits one by one, starting with the leftmost digit. If you had a machine similar to SLIM but it was (more realistically) only able to output a single character at a time, this method is how you would have to output numbers. (*Hint:* Your solution will probably be similar to the one for Exercise 11.11.)

Exercise 11.20

Suppose you bought a second-hand SLIM dirt cheap, only to find that the `j` instruction on it didn't work. Explain how you could rewrite all your programs to not use this instruction.

Exercise 11.21

Write a procedure in Scheme that when given a vector and two integers specifying locations within the vector, it swaps the contents of the specified locations.

Exercise 11.22

Write a procedure in Scheme that when given a vector, it stores 0 into location 0 of that vector, 1 into location 1, 2 into location 2, etc.

Exercise 11.23

We can represent a deck of cards as a 52-element vector, initialized to hold the values 0 through 51 using the procedure from Exercise 11.22. If we wish to randomize the order of the "cards" (i.e., values) prior to using the deck in a game, we can use the following plan:

1. Randomly pick an integer in the range from 0 to 51; we'll call this integer *i*.
2. Swap the contents of locations *i* and 51 of the vector, using the swapping procedure from Exercise 11.21.
3. Now similarly pick a random number in the range from 0 to 50, and swap that location with location 50.
4. Continue in this way with progressively smaller prefixes of the vector, swapping a randomly chosen location within the range with the last element of the range.
5. Once a random choice of either location 0 or 1 has been swapped with location 1, the vector has been totally randomized.

Implement this randomization procedure in Scheme; the procedure should take the vector to randomize as its argument and should work for vectors of sizes other than 52.

▷ Exercise 11.24

Write a procedure, `shift!`, which takes a vector as its one argument and modifies that vector by shifting its contents down one position as follows. If the length of the vector is n, for k in the range $0 \leq k < n - 1$, position k of the vector should be modified to hold the value that was originally in position $k + 1$. The last element of the vector should be left unchanged. Warning: It matters whether you loop upward from the beginning of the vector to the end or downward from the end of the vector to the beginning.

▷ Exercise 11.25

The following SLIM assembly language program reads in two integers, x and y, computes some function of them, and writes out the answer. You may assume that neither x nor y is negative and that the arithmetic operations done by the program never overflow (i.e., never produce a result too large or small to represent). Express in a simple mathematical formula what function of x and y the program computes. Explain the reasoning behind your answer.

```
allocate-registers x, y, z, one, loop-reg, end-reg

read x
read y
li one, 1
li loop-reg, loop
li end-reg, end
```

```
loop:
   jeqz y, end-reg
   add z, x, x
   add z, z, z
   sub x, x, z
   sub y, y, one
   j loop-reg

end:
   write x
   halt
```

Exercise 11.26

Consider the following iterative procedure that computes the sum of digits of a nonnegative number:

```
(define sum-of-digits
  (lambda (n) ; assume n >= 0
    (define iter
      (lambda (n acc)
        (if (= n 0)
            acc
            (let ((last-digit (remainder n 10)))
              (iter (quotient n 10)
                    (+ acc last-digit))))))
    (iter n 0)))
```

Write a SLIM assembly language program that reads in an integer *n* (which you may assume is nonnegative) and writes out the sum of its digits, using the algorithm described in the previous Scheme procedure.

Exercise 11.27

Write a procedure `multiply-by!` that takes a vector and a number and changes the vector by multiplying each value in the vector by that number. You may assume that the vector itself is filled with numbers. As an example, you should have the following interaction:

```
(define v (vector 2 1 4 5))

v
#(2 1 4 5) ; <-- this is the way Scheme displays vectors

(multiply-by! v 3)
done

v
#(6 3 12 15)

(multiply-by! v 2)
done

v
#(12 6 24 30)
```

Exercise 11.28

Write a SLIM assembly language program that reads in an integer *n* (which you may assume is positive) and writes out its largest odd divisor, using the algorithm described in the Scheme procedure of Exercise 3.13 on page 69.

Hint: You can test whether the contents of a given register is odd by using the SLIM instruction `rem` to check whether its remainder upon division by 2 is 1.

Chapter Inventory

Vocabulary

memory
Random Access Memory (RAM)
locations
vectors
Super-Lean Instruction Machine (SLIM)
architecture
stored program computer
program
instructions
instruction set
operating system
computer core
run a program
execute a program
input device
output device
processor
data memory
address
shared-memory multiprocessor
Direct Memory Access (DMA)
control unit
registers
arithmetic logical unit (ALU)
source registers
destination register
address register

instruction memory
Harvard architecture
current instruction
current instruction address
program counter (PC)
jump
jump target address
instruction decoder
control signals
bits
word size
word
floating point
mantissa
exponent
machine language
assembly language
assembler
operation code
opcode
operand specifiers
load
store
conditional jumps
fall through
unconditional jumps
labels
interpreter
compiler
loops
flow chart
continuation register
continuation address
stack discipline
stack
push
pop
stack pointer (SP)
imperative style or paradigm
state
linearization
functional style or paradigm
value oriented
state oriented
vectors
module
selector
mutator

Abstract Data Types

machine models
machine language instructions

New Predefined Scheme Names

```
make-vector
vector-length
vector-set!
vector-ref
```

Scheme Names Defined in This Chapter

```
power-product
gcd
two-factorials
do-grade-histogram
zero-out-vector!
display-histogram
maximum-bar-size
largest-element-of-vector
load-and-run
make-machine-model
get-pc
set-pc!
get-reg
set-reg!
```

```
get-mem
set-mem!
halted?
halt!
do-instruction-in-model
make-load-inst
make-store-inst
make-load-immediate-inst
make-add-inst
make-sub-inst
make-mul-inst
make-div-inst
make-quo-inst
make-rem-inst
make-seq-inst
make-sne-inst
make-slt-inst
make-sgt-inst
make-sle-inst
make-sge-inst
make-jeqz-inst
make-jump-inst
make-read-inst
make-write-inst
make-halt-inst
mem-size
reg-bank-size
write-larger
assemble
shift!
sum-of-digits
multiply-by!
```

Sidebars

What Can Be Stored in a Location?
SLIM Instruction Set

Notes

Our SLIM architecture is similar to that of many modern RISC instruction set architectures, such as the MIPS architecture described by Kane and Heinrich [30]. A good next step if you are interested in computer organization and assembly language programming is Patterson and Hennessy's book [39]. (That book uses the MIPS architecture for its examples, so the transition from SLIM should be relatively smooth.)

One key difference to be aware of if you compare our treatment of assembly language programming with that of other authors is that what we refer to as a *continuation address* is called a *return address* by most other authors; similarly, they refer to a *return address register* rather than a *continuation register*. This alternate name reflects the fact that the continuation of a procedure often is directly after the corresponding call, so the procedure continues by returning whence it came. Our choice of name reflects the fact that although this returning behavior is common, it isn't universal, so we'd rather use the more neutral name *continuation address*.