calculation. This form requires an offset field in the instruction large enough to hold an address, of course, so it is less efficient than doing it the other way; however, it is nevertheless frequently the best way.

5.4.7 Based-Indexed Addressing

Some machines have an addressing mode in which the memory address is computed by adding up two registers plus an (optional) offset. Sometimes this mode is called **based-indexed addressing**. One of the registers is the base and the other is the index. Such a mode would have been useful here. Outside the loop we could have put the address of *A* in R5 and the address of *B* in R6. Then we could have replaced the instruction at *LOOP* and its successor with

```
LOOP:   MOV R4,(R2+R5)
        AND R4,(R2+R6)
```

If there were an addressing mode for indirecting through the sum of two registers with no offset, that would be ideal. Alternatively, even an instruction with an 8-bit offset would have been an improvement over the original code since we could set both offsets to 0. If, however, the offsets are always 32 bits, then we have not gained anything by using this mode. In practice, however, machines that have this mode usually have a form with an 8-bit or 16-bit offset.

5.4.8 Stack Addressing

We have already noted that making machine instructions as short as possible is highly desirable. The ultimate limit in reducing address lengths is having no addresses at all. As we saw in Chap. 4, zero-address instructions, such as IADD, are possible in conjunction with a stack. In this section we will look at stack addressing more closely.

Reverse Polish Notation

It is an ancient tradition in mathematics to put the operator between the operands, as in $x + y$, rather than after the operands, as in $x\ y +$. The form with the operator "in" between the operands is called **infix** notation. The form with the operator after the operands is called **postfix** or **reverse Polish notation**, after the Polish logician J. Lukasiewicz (1958), who studied the properties of this notation.

Reverse Polish notation has a number of advantages over infix for expressing algebraic formulas. First, any formula can be expressed without parentheses. Second, it is convenient for evaluating formulas on computers with stacks. Third, infix operators have precedence, which is arbitrary and undesirable. For example, we

know that $a \times b + c$ means $(a \times b) + c$ and not $a \times (b + c)$ because multiplication has been arbitrarily defined to have precedence over addition. But does left shift have precedence over Boolean AND? Who knows? Reverse Polish notation eliminates this nuisance.

Several algorithms for converting infix formulas into reverse Polish notation exist. The one given below is an adaptation of an idea due to E. W. Dijkstra. Assume that a formula is composed of the following symbols: variables, the dyadic (two-operand) operators + − * /, and left and right parentheses. To mark the ends of a formula, we will insert the symbol ⊥ after the last symbol and before the first symbol.

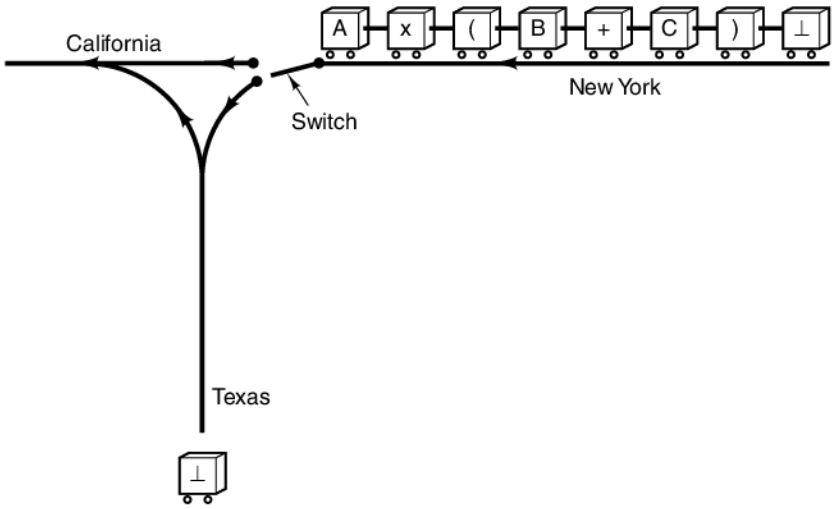

Figure 5-20. Each railroad car represents one symbol in the formula to be converted from infix to reverse Polish notation.

Figure 5-20 shows a railroad track from New York to California, with a spur in the middle that heads off in the direction of Texas. Think of the New York to California line as the main line with the branch down to Texas as a siding for temporary storage. The names and directions are not important. What matters is the distinction between the main line and the alternative place for temporarily storing things. Each symbol in the formula is represented by one railroad car. The train moves westward (to the left). When each car arrives at the switch, it must stop just before it and ask if it should go to California directly or take a side trip to Texas. Cars containing variables always go directly to California and never to Texas. Cars containing all other symbols must inquire about the contents of the nearest car on the Texas line before entering the switch.

The table of Fig. 5-21 shows what happens, depending on the contents of the nearest car on the Texas line and the car poised at the switch. The first ⊥ always goes to Texas. The numbers refer to the following situations:

1. The car at the switch heads toward Texas.
2. The most recent car on the Texas line turns and goes to California.
3. Both the car at the switch and the most recent car on the Texas line are diverted and disappear (i.e., both are deleted).
4. Stop. The symbols now in California represent the reverse Polish notation formula when read from left to right.
5. Stop. An error has occurred. The original formula was not correctly balanced.

Most recently arrived car on the Texas line \ Car at the switch	⊥	+	−	x	/	(	)
⊥	4	1	1	1	1	1	5
+	2	2	2	1	1	1	2
−	2	2	2	1	1	1	2
x	2	2	2	2	2	1	2
/	2	2	2	2	2	1	2
(	5	1	1	1	1	1	3

Figure 5-21. Decision table used by the infix-to-reverse Polish notation algorithm

After each action is taken, a new comparison is made between the car currently at the switch, which may be the same car as in the previous comparison or may be the next car, and the car that is now the last one on the Texas line. The process continues until step 4 is reached. Notice that the Texas line is being used as a stack, with routing a car to Texas being a push operation, and turning a car already on the Texas line around and sending it to California being a pop operation.

Infix	Reverse Polish notation
A + B × C	A B C × +
A × B + C	A B × C +
A × B + C × D	A B × C D × +
(A + B) / (C − D)	A B + C D − /
A × B / C	A B × C /
((A + B) × C + D)/(E + F + G)	A B + C × D + E F + G + /

Figure 5-22. Some examples of infix expressions and their reverse Polish notation equivalents.

The order of the variables is the same in infix and in reverse Polish notation. The order of the operators, however, is not always the same. Operators appear in

reverse Polish notation in the order they will actually be executed during the evaluation of the expression. Figure 5-22 gives several examples of infix formulas and their reverse Polish notation equivalents.

Evaluation of Reverse Polish Notation Formulas

Reverse Polish notation is the ideal notation for evaluating formulas on a computer with a stack. The formula consists of n symbols, each one either an operand or an operator. The algorithm for evaluating a reverse Polish notation formula using a stack is simple. Scan the reverse Polish notation string from left to right. When an operand is encountered, push it onto the stack. When an operator is encountered, execute the corresponding instruction.

Figure 5-23 shows the evaluation of

$$(8 + 2 \times 5) / (1 + 3 \times 2 - 4)$$

in IJVM. The corresponding reverse Polish notation formula is

$$8\ 2\ 5 \times +\ 1\ 3\ 2 \times +\ 4 - /$$

In the figure, we have introduced IMUL and IDIV as the multiplication and division instructions, respectively. The number on top of the stack is the right operand, not the left operand. This point is important for division (and subtraction) since the order of the operands is significant (unlike addition and multiplication). In other words, IDIV has been carefully defined so that first pushing the numerator, then pushing the denominator, and then doing the operation gives the correct result. Notice how easy code generation is from reverse Polish notation to IJVM: just scan the reverse Polish notation formula and output one instruction per symbol. If the symbol is a constant or variable, output an instruction to push it onto the stack. If the symbol is an operator, output an instruction to perform the operation.

5.4.9 Addressing Modes for Branch Instructions

So far we have been looking only at instructions that operate on data. Branch instructions (and procedure calls) also need addressing modes for specifying the target address. The modes we have examined so far also work for branches for the most part. Direct addressing is certainly a possibility, with the target address simply being included in the instruction in full.

However, other addressing modes also make sense. Register indirect addressing allows the program to compute the target address, put it in a register, and then go there. This mode gives the most flexibility since the target address is computed at run time. It also presents the greatest opportunity for creating bugs that are nearly impossible to find.

Another reasonable mode is indexed mode, which offsets a known distance from a register. It has the same properties as register indirect mode.

Step	Remaining string	Instruction	Stack
1	8 2 5 × + 1 3 2 × + 4 – /	BIPUSH 8	8
2	2 5 × + 1 3 2 × + 4 – /	BIPUSH 2	8, 2
3	5 × + 1 3 2 × + 4 – /	BIPUSH 5	8, 2, 5
4	× + 1 3 2 × + 4 – /	IMUL	8, 10
5	+ 1 3 2 × + 4 – /	IADD	18
6	1 3 2 × + 4 – /	BIPUSH 1	18, 1
7	3 2 × + 4 – /	BIPUSH 3	18, 1, 3
8	2 × + 4 – /	BIPUSH 2	18, 1, 3, 2
9	× + 4 – /	IMUL	18, 1, 6
10	+ 4 – /	IADD	18, 7
11	4 – /	BIPUSH 4	18, 7, 4
12	– /	ISUB	18, 3
13	/	IDIV	6

Figure 5-23. Use of a stack to evaluate a reverse Polish notation formula.

Another option is PC-relative addressing. In this mode, the (signed) offset in the instruction itself is added to the program counter to get the target address. In fact, this is simply indexed mode, using PC as the register.

5.4.10 Orthogonality of Opcodes and Addressing Modes

From a software point of view, instructions and addressing should have a regular structure, with a minimum number of instruction formats. Such a structure makes it much easier for a compiler to produce good code. All opcodes should permit all addressing modes wherever that makes sense. Furthermore, all registers should be available for all register modes [including the frame pointer (FP), stack pointer (SP), and program counter (PC)].

As an example of a clean design for a three-address machine, consider the 32-bit instruction formats of Fig. 5-24. Up to 256 opcodes are supported. In format 1, each instruction has two source registers and a destination register. All arithmetic and logical instructions use this format.

The unused 8-bit field at the end can be used for further instruction differentiation. For example, one opcode could be allocated for all the floating-point operations, with the extra field distinguishing among them. In addition, if bit 23 is set, format 2 is used and the second operand is no longer a register but a 13-bit signed immediate constant. LOAD and STORE instructions can also use this format to reference memory in indexed mode.

A small number of additional instructions are needed, such as conditional branches, but they could easily fit in format 3. For example, one opcode could be

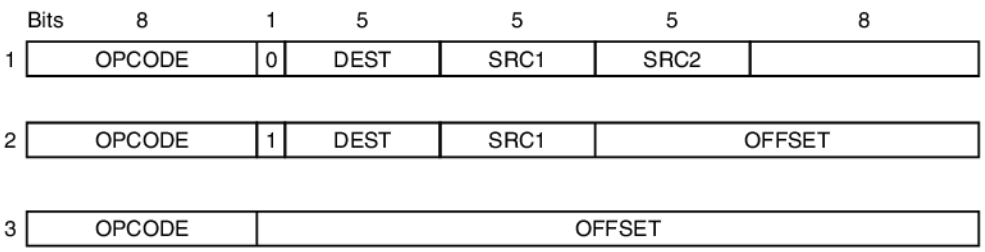

Figure 5-24. A simple design for the instruction formats of a three-address machine.

assigned to each (conditional) branch, procedure call, etc., leaving 24 bits for a PC relative offset. Assuming that this offset counted in words, the range would be ±32 MB. Also a few opcodes could be reserved for LOADs and STOREs that need the long offsets of format 3. These would not be fully general (e.g., only R0 could be loaded or stored), but they would rarely be used.

Now consider a design for a two-address machine that can use a memory word for either operand. It is shown in Fig. 5-25. Such a machine can add a memory word to a register, add a register to a memory word, add a register to a register, or add a memory word to a memory word. At present, memory accesses are relatively expensive, so this design is not currently popular, but if advances in cache or memory technology make memory accesses cheap in the future, it is a particularly easy and efficient design to compile to. The PDP-11 and VAX were highly successful machines that dominated the minicomputer world for two decades using designs similar to this one.

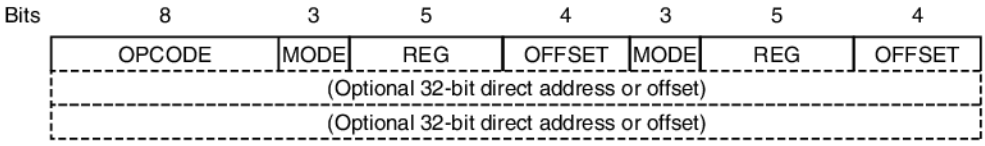

Figure 5-25. A simple design for the instruction formats of a two-address machine.

In this design, we again have an 8-bit opcode, but now we have 12 bits for specifying the source and an additional 12 bits for specifying the destination. For each operand, 3 bits give the mode, 5 bits tell the register, and 4 bits provide the offset. With 3 mode bits, we could support immediate, direct, register, register indirect, indexed, and stack modes, and have room left over for two more future modes. This is a clean and regular design that is easy to compile for and quite flexible, especially if the program counter, stack pointer, and local variable pointer are among the general registers that can be accessed the usual way.

The only problem here is that for direct addressing, we need more bits for the address. What the PDP-11 and VAX did was add an extra word to the instruction for the address of each directly addressed operand. We could also use one of the two available addressing modes for an indexed mode with a 32-bit offset following the instruction. Thus in the worst case, say, a memory-to-memory ADD with both operands directly addressed or using the long indexed form, the instruction would be 96 bits long and use three bus cycles (one for the instruction, two for its addresses). In addition, three more cycles would be needed to fetch the two operands and write the result. On the other hand, most RISC designs would require at least 96 bits, probably more, to add an arbitrary word in memory to another arbitrary word in memory and use at least four bus cycles, depending how the operands were addressed.

Many alternatives to Fig. 5-25 are possible. In this design, it is possible to execute the statement

```
i = j;
```

in one 32-bit instruction, provided that both *i* and *j* are among the first 16 local variables. On the other hand, for variables beyond 16, we have to go to 32-bit offsets. One option would be another format with a single 8-bit offset instead of two 4-bit offsets, plus a rule saying that either the source or the destination could use it, but not both. The possibilities and trade-offs are unlimited, and machine designers must juggle many factors to get a good result.

5.4.11 The Core i7 Addressing Modes

The Core i7's addressing modes are highly irregular and are different depending on whether a particular instruction is in 16-, 32-, or 64-bit mode. Below we will ignore the 16- and 64-bit modes; 32-bit mode is bad enough. The modes supported include immediate, direct, register, register indirect, indexed, and a special mode for addressing array elements. The problem is that not all modes apply to all instructions and not all registers can be used in all modes. This makes the compiler writer's job much more difficult and leads to worse code.

The MODE byte in Fig. 5-13 controls the addressing modes. One of the operands is specified by the combination of the MOD and R/M fields. The other is always a register and is given by the value of the REG field. The 32 combinations that can be specified by the 2-bit MOD field and the 3-bit R/M field are listed in Fig. 5-26. If both fields are zero, for example, the operand is read from the memory address contained in the EAX register.

The 01 and 10 columns involve modes in which a register is added to an 8- or 32-bit offset that follows the instruction. If an 8-bit offset is selected, it is first sign-extended to 32 bits before being added. For example, an ADD instruction with R/M = 011, MOD = 01, and an offset of 6 computes the sum of EBX and 6 and reads the memory word at that address for one of the operands. EBX is not modified.

	MOD			
R/M	00	01	10	11
000	M[EAX]	M[EAX + OFFSET8]	M[EAX + OFFSET32]	EAX or AL
001	M[ECX]	M[ECX + OFFSET8]	M[ECX + OFFSET32]	ECX or CL
010	M[EDX]	M[EDX + OFFSET8]	M[EDX + OFFSET32]	EDX or DL
011	M[EBX]	M[EBX + OFFSET8]	M[EBX + OFFSET32]	EBX or BL
100	SIB	SIB with OFFSET8	SIB with OFFSET32	ESP or AH
101	Direct	M[EBP + OFFSET8]	M[EBP + OFFSET32]	EBP or CH
110	M[ESI]	M[ESI + OFFSET8]	M[ESI + OFFSET32]	ESI or DH
111	M[EDI]	M[EDI + OFFSET8]	M[EDI + OFFSET32]	EDI or BH

Figure 5-26. The Core i7 32-bit addressing modes. M[x] is the memory word at x.

The MOD = 11 column gives a choice of two registers. For word instructions, the first choice is used; for byte instructions, the second choice. Observe that the table is not entirely regular. For example, there is no way to indirect through EBP and no way to offset from ESP.

In some modes, an additional byte, called **SIB** (**Scale, Index, Base**), follows the MODE byte (see Fig. 5-13). The SIB byte specifies a scale factor as well as two registers. When a SIB byte is present, the operand address is computed by multiplying the index register by 1, 2, 4, or 8 (depending on SCALE), adding it to the base register, and finally possibly adding an 8- or 32-bit displacement, depending on MOD. Almost all the registers can be used as either index or base.

The SIB modes are useful for accessing array elements. For example, consider the Java statement

```
for (i = 0; i < n; i++) a[i] = 0;
```

where a is an array of 4-byte integers local to the current procedure. Typically, EBP is used to point to the base of the stack frame containing the local variables and arrays, as shown in Fig. 5-27. The compiler might keep i in EAX. To access $a[i]$, it would use an SIB mode whose operand address was the sum of 4 × EAX, EBP, and 8. This instruction could store into $a[i]$ in a single instruction.

Is this mode worth the trouble? It is hard to say. Undoubtedly this instruction, when properly used, saves a few cycles. How often it is used depends on the compiler and the application. The problem is that this instruction occupies a certain amount of chip area that could have been used in a different way had this instruction not been present. For example, the level 1 cache could have been made bigger, or the chip could have been made smaller, possibly allowing a slightly higher clock speed.

These are the kinds of trade-offs that designers face constantly. Usually, extensive simulation runs are made before casting anything in silicon, but these simulations require having a good idea of what the workload is like. It is a safe bet that

the designers of the 8088 did not include a Web browser in their test set. Nevertheless, quite a few of that product's descendants are now used primarily for Web surfing, so the decisions made 20 years ago may be completely wrong for current applications. However, in the name of backward compatibility, once a feature gets in there, it is impossible to get it out.

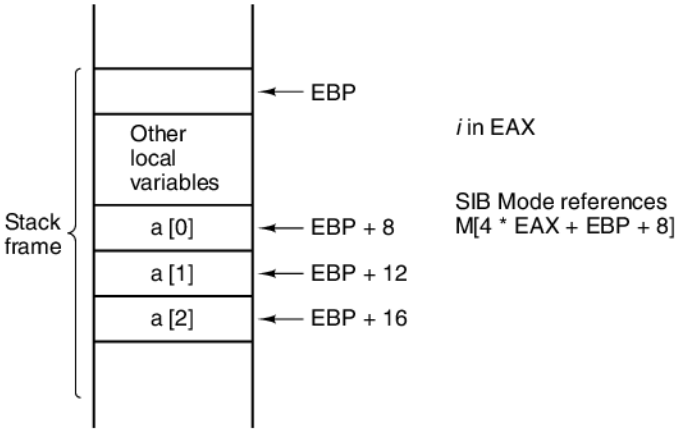

Figure 5-27. Access to $a[i]$.

5.4.12 The OMAP4440 ARM CPU Addressing Modes

In the OMAP4430, all instructions use immediate or register addressing except those that address memory. For register mode, the 5 bits simply tell which register to use. For immediate mode, an (unsigned) 12-bit constant provides the data. No other modes are present for the arithmetic, logical, and similar instructions.

Two kinds of instructions address memory: loads (LDR) and stores (STR). LDR and STR instructions have three modes for addressing memory. The first mode computes the sum of two registers and then indirects through it. The second mode computes the address as the sum of a base register and a 13-bit signed offset. The third addressing mode computes an address equal to the program counter (PC) plus a 13-bit signed offset. This third addressing mode, called PC-relative addressing, is useful for loading program constants which are stored with the program's code.

5.4.13 The ATmega168 AVR Addressing Modes

The ATmega168 has a fairly regular addressing structure. There are four basic modes. The first is register mode, in which the operand is in a register. Registers can be used as both sources and destinations. The second is immediate mode, where an 8-bit unsigned immediate value can be encoded into an instruction.

The remaining modes are usable only by load and store instructions. The third mode is direct addressing, where the operand is in memory at an address contained in the instruction itself. For 16-bit instructions, the direct address is limited to 7 bits (thus only addresses 0 to 127 can be loaded). The AVR architecture also defines a 32-bit instruction as well that accommodates a 16-bit direct address, which supports up to 64 KB of memory.

The fourth mode is register indirect, in which a register contains a pointer to the operand. Since the normal registers are 8 bits wide, the load and store instructions use register pairs to express a 16-bit address. A register pair can address up to 64 KB of memory. The architecture supports the use of three register pairs: X, Y, and Z, which are formed from the register pairs R26/R27, R28/R29, and R30/R31, respectively. To load an address into the X register for example, the program would have to load an 8-bit value into the R26 and R27 registers, using two load instructions.

5.4.14 Discussion of Addressing Modes

We have now studied quite a few addressing modes. The ones used by the Core i7, OMAP4430, and ATmega168 are summarized in Fig. 5-28. As we have pointed out, however, not every mode can be used in every instruction.

Addressing mode	Core i7	OMAP4430 ARM	ATmega168 AVR
Immediate	×	×	×
Direct	×		×
Register	×	×	×
Register indirect	×	×	×
Indexed	×	×	
Based-indexed		×	

Figure 5-28. A comparison of addressing modes.

In practice, not many addressing modes are needed for an effective ISA. Since most code written at this level these days will be generated by compilers (except possibly for the ATmega168), the most important aspect of an architecture's addressing modes is that the choices be few and clear, with costs (in terms of execution time and code size) that are readily computable. What that generally means is that a machine should take an extreme position: either it should offer every possible choice, or it should offer only one. Anything in between means that the compiler is faced with choices that it may not have the knowledge or sophistication to make.

Thus the cleanest architectures generally have only a very small number of addressing modes, with strict limits on their use. In practice, having immediate, direct, register, and indexed mode is generally enough for almost all applications.

Also, every register (including local variable pointer, stack pointer, and program counter) should be usable wherever a register is called for. More complicated addressing modes may reduce the number of instructions, but at the expense of introducing sequences of operations that cannot easily be parallelized with other sequential operations.

We have now completed our study of the various trade-offs possible between opcodes and addresses, and various forms of addressing. When approaching a new computer, you should examine the instructions and addressing modes not only to see which ones are available but also to understand why those choices were made and what the consequences of alternative choices would have been.

5.5 INSTRUCTION TYPES

ISA-level instructions can be approximately divided into a half-dozen groups that are relatively similar from machine to machine, even though they may differ in the details. In addition, every computer has a few unusual instructions, added for compatibility with previous models, or because the architect had a brilliant idea, or possibly because a government agency paid the manufacturer to include it. We will try to briefly cover all the common categories below, without any attempt at being exhaustive.

5.5.1 Data Movement Instructions

Copying data from one place to another is the most fundamental of all operations. By copying we mean the creating of a new object, with the identical bit pattern as the original. This use of the word "movement" is somewhat different from its normal usage in English. When we say that Marvin Mongoose has moved from New York to California, we do not mean that an identical copy of Mr. Mongoose was created in California and that the original is still in New York. When we say that the contents of memory location 2000 have been moved to some register, we always mean that an identical copy has been created there and that the original is still undisturbed in location 2000. Data movement instructions would better be called "data duplication" instructions, but the term "data movement" is already established.

There are two reasons for copying data from one location to another. One is fundamental: the assignment of values to variables. The assignment

```
A = B
```

is implemented by copying the value at memory address *B* to location *A* because the programmer has said to do this. The second reason is to stage the data for efficient access and use. As we have seen, many instructions can access variables only when they are available in registers. Since there are two possible sources for

a data item (memory or register), and there are two possible destinations for a data item (memory or register), four different kinds of copying are possible. Some computers have four instructions for the four cases. Others have one instruction for all four cases. Still others use LOAD to go from memory to a register, STORE to go from a register to memory, MOVE to go from one register to another register, and no instruction for a memory-to-memory copy.

Data movement instructions must indicate the amount of data to be moved. Instructions exist for some ISAs to move variable quantities of data ranging from 1 bit to the entire memory. On fixed-word-length machines, the amount to be moved is often exactly one word. Any more or less must be performed by a software routine using shifting and merging. Some ISAs provide additional capability both for copying less than a word (usually in increments of bytes) and for copying multiple words. Copying multiple words is tricky, particularly if the maximum number of words is large, because such an operation can take a long time, and may have to be interrupted in the middle. Some variable-word-length machines have instructions specifying only the source and destination addresses but not the amount. The move continues until an end-of-data field mark is found in the data.

5.5.2 Dyadic Operations

Dyadic operations combine two operands to produce a result. All ISAs have instructions to perform addition and subtraction on integers. Multiplication and division of integers are nearly standard as well. It is presumably unnecessary to explain why computers are equipped with arithmetic instructions.

Another group of dyadic operations includes the Boolean instructions. Although 16 Boolean functions of two variables exist, few, if any, machines have instructions for all 16. Usually, AND, OR, and NOT are present; sometimes EXCLUSIVE OR, NOR, and NAND are there as well.

An important use of AND is for extracting bits from words. Consider, for example, a 32-bit-word-length machine in which four 8-bit characters are stored per word. Suppose that it is necessary to separate the second character from the other three in order to print it; that is, it is necessary to create a word which contains that character in the rightmost 8 bits, referred to as **right justified**, with zeros in the leftmost 24 bits.

To extract the character, the word containing the character is ANDed with a constant, called a **mask**. The result of this operation is that the unwanted bits are all changed into zeros—that is, masked out—as shown below.

```
10110111 10111100 11011011 10001011  A
00000000 11111111 00000000 00000000  B (mask)
00000000 10111100 00000000 00000000  A AND B
```

The result would then be shifted 16 bits to the right to isolate the character to be extracted at the right end of the word.

An important use of OR is to pack bits into a word, packing being the inverse of extracting. To change the rightmost 8 bits of a 32-bit word without disturbing the other 24 bits, first the unwanted 8 bits are masked out and then the new character is ORed in, as shown below.

```
10110111 10111100 11011011 10001011  A
11111111 11111111 11111111 00000000  B (mask)
10110111 10111100 11011011 00000000  A AND B
00000000 00000000 00000000 01010111  C
10110111 10111100 11011011 01010111  (A AND B) OR C
```

The AND operation tends to remove 1s, because there are never more 1s in the result than in either of the operands. The OR operation tends to insert 1s, because there are always at least as many 1s in the result as in the operand with the most 1s. EXCLUSIVE OR, on the other hand, is symmetric, tending, on the average, neither to insert nor remove 1s. This symmetry with respect to 1s and 0s is occasionally useful, for example, in generating random numbers.

Most computers today also support a set of floating-point instructions, roughly corresponding to the integer arithmetic operations. Most machines provide at least two lengths of floating-point numbers, the shorter ones for speed and the longer ones for occasions when many digits of accuracy are needed. While there are lots of possible variations for floating-point formats, a single standard has now been almost universally adopted: IEEE 754. Floating-point numbers and IEEE 754 are discussed in Appendix B.

5.5.3 Monadic Operations

Monadic operations have one operand and produce one result. Because one fewer address has to be specified than with dyadic operations, the instructions are sometimes shorter, though often other information must be specified.

Instructions to shift or rotate the contents of a word or byte are quite useful and are often provided in several variations. Shifts are operations in which the bits are moved to the left or right, with bits shifted off the end of the word being lost. Rotates are shifts in which bits pushed off one end reappear on the other end. The difference between a shift and a rotate is illustrated below.

```
00000000 00000000 00000000 01110011  A
00000000 00000000 00000000 00011100  A shifted right 2 bits
11000000 00000000 00000000 00011100  A rotated right 2 bits
```

Both left and right shifts and rotates are useful. If an n-bit word is left rotated k bits, the result is the same as if it had been right rotated $n - k$ bits.

Right shifts are often performed with sign extension. This means that positions vacated on the left end of the word are filled up with the original sign bit, 0 or 1. It is as though the sign bit were dragged along to the right. Among other things,

it means that a negative number will remain negative. This situation is illustrated below for 2-bit right shifts.

```
11111111 11111111 11111111 11110000  A
00111111 11111111 11111111 11111100  A shifted without sign extension
11111111 11111111 11111111 11111100  A shifted with sign extension
```

An important use of shifting is multiplication and division by powers of 2. If a positive integer is left shifted k bits, the result, barring overflow, is the original number multiplied by 2^k. If a positive integer is right shifted k bits, the result is the original number divided by 2^k.

Shifting can be used to speed up certain arithmetic operations. Consider, for example, computing $18 \times n$ for some positive integer n. Because $18 \times n = 16 \times n + 2 \times n$, $16 \times n$ can be obtained by shifting a copy of n 4 bits to the left. $2 \times n$ can be obtained by shifting n 1 bit to the left. The sum of these two numbers is $18 \times n$. The multiplication has been accomplished by a move, two shifts, and an addition, which is often faster than a multiplication. Of course, the compiler can use this trick only when one factor is a constant.

Shifting negative numbers, even with sign extension, gives quite different results, however. Consider, for example, the ones' complement number, −1. Shifted 1 bit to the left it yields −3. Another 1-bit shift to the left yields −7:

```
11111111 11111111 11111111 11111110  −1 in ones' complement
11111111 11111111 11111111 11111100  −1 shifted left 1 bit = −3
11111111 11111111 11111111 11111000  −1 shifted left 2 bits = −7
```

Left shifting ones' complement negative numbers does not multiply by 2. Right shifting does simulate division correctly, however.

Now consider a two's complement representation of −1. When right shifted 6 bits with sign extension, it yields −1, which is incorrect because the integral part of −1/64 is 0:

```
11111111 11111111 11111111 11111111  −1 in two's complement
11111111 11111111 11111111 11111111  −1 shifted right 6 bits = −1
```

In general, right shifting introduces errors because it truncates down (toward the more negative integer), which is incorrect for integer arithmetic on negative numbers. Left shifting does, however, simulate multiplication by 2.

Rotate operations are useful for packing and unpacking bit sequences from words. If it is desired to test all the bits in a word, rotating the word 1 bit at a time either way successively puts each bit in the sign bit, where it can be easily tested, and also restores the word to its original value when all bits have been tested. Rotate operations are more pure than shift operations because no information is lost: an arbitrary rotate operation can be negated with another rotate operation.

Certain dyadic operations occur so frequently with particular operands that ISAs sometimes have monadic instructions to accomplish them quickly. Moving

zero to a memory word or register is extremely common when initializing a calculation. Moving zero is, of course, a special case of the general move data instructions. For efficiency, a CLR operation, with only one address, the location to be cleared (i.e., set to zero), is often provided.

Adding 1 to a word is also commonly used for counting. A monadic form of the ADD instruction is the INC operation, which adds 1. The NEG operation is another example. Negating X is really computing $0 - X$, a dyadic subtraction, but again, because of its frequent use, a separate NEG instruction is sometimes provided. It is important to note here the difference between the arithmetic operation NEG and the logical operation NOT. The NEG operation produces the **additive inverse** of a number (the number which when added to the original gives 0). The NOT operation simply inverts all the individual bits in the word. The operations are very similar, and in fact, for a system using ones' complement representation, they are identical. (In twos' complement arithmetic, the NEG instruction is carried out by first inverting all the individual bits, then adding 1.)

Dyadic and monadic instructions are often grouped together by their use, rather than by the number of operands they require. One group includes the arithmetic operations, including negation. The other group includes logical operations and shifting, since these two categories are most often used together to accomplish data extraction.

5.5.4 Comparisons and Conditional Branches

Nearly all programs need the ability to test their data and alter the sequence of instructions to be executed based on the results. A simple example is the square-root function, $\sqrt{x}$. If x is negative, the procedure gives an error message; otherwise it computes the square root. A function *sqrt* has to test x and then branch, depending on whether it is negative or not.

A common method for doing so is to provide conditional branch instructions that test some condition and branch to a particular memory address if the condition is met. Sometimes a bit in the instruction indicates whether the branch is to occur if the condition is met or not met, respectively. Often the target address is not absolute, but relative to the current instruction.

The most common condition to be tested is whether a particular bit in the machine is 0 or not. If an instruction tests the sign bit of a number and branches to *LABEL* if it is 1, the statements beginning at *LABEL* will be executed if the number was negative, and the statements following the conditional branch will be executed if it was 0 or positive.

Many machines have condition code bits that are used to indicate specific conditions. For example, there may be an overflow bit that is set to 1 whenever an arithmetic operation gives an incorrect result. By testing this bit, one checks for overflow on the previous arithmetic operation, so that if an overflow occurred, a branch can be made to an error routine and corrective action taken.

Similarly, some processors have a carry bit that is set when a carry spills over from the leftmost bit, for example, if two negative numbers are added. A carry from the leftmost bit is quite normal and should not be confused with an overflow. Testing of the carry bit is needed for multiple-precision arithmetic (i.e., where an integer is represented in two or more words).

Testing for zero is important for loops and many other purposes. If all the conditional branch instructions tested only 1 bit, then to test a particular word for 0, one would need a separate test for each bit, to ensure that none was a 1. To avoid this situation, many machines have an instruction to test a word and branch if it is zero. Of course, this solution merely passes the buck to the microarchitecture. In practice, the hardware usually contains a register all of whose bits are ORed together to give a single bit telling whether the register contains any 1 bits. The *Z* bit in Fig. 4-1 would normally be computed by ORing all the ALU output bits together and then inverting the result.

Comparing two words or characters to see if they are equal or, if not, which one is greater is also important, in sorting for example. To perform this test, three addresses are needed: two for the data items, and one for the address to branch to if the condition is true. Computers whose instruction format allows three addresses per instruction have no trouble, but those that do not must do something to get around this problem.

One common solution is to provide an instruction that performs a comparison and sets one or more condition bits to record the result. A subsequent instruction can test the condition bits and branch if the two compared values were equal, or unequal, or if the first was greater, and so on. The Core i7, OMAP4430 ARM CPU, and ATmega168 AVR all use this approach.

Some subtle points are involved in comparing two numbers. For example, comparison is not quite as simple as subtraction. If a very large positive number is compared to a very large negative number, the subtraction will result in overflow, since the result of the subtraction cannot be represented. Nevertheless, the comparison instruction must determine whether the specified test is met and return the correct answer—there is no overflow on comparisons.

Another subtle point relating to comparing numbers is deciding whether or not the numbers should be considered signed or not. Three-bit binary numbers can be ordered in one of two ways. From smallest to largest:

Unsigned	Signed	
000	100	(smallest)
001	101	
010	110	
011	111	
100	000	
101	001	
110	010	
111	011	(largest)

The column on the left shows the positive integers 0 to 7 in increasing order. The column on the right shows the two's complement signed integers −4 to +3. The answer to the question "Is 011 greater than 100?" depends on whether or not the numbers are regarded as being signed. Most ISAs have instructions to handle both orderings.

5.5.5 Procedure Call Instructions

A **procedure** is a group of instructions that performs some task and that can be invoked (called) from several places in the program. The term **subroutine** is often used instead of procedure, especially when referring to assembly-language programs. In C, procedures are called **functions**, even though they are not necessarily functions in the mathematical sense. In Java, the term used is **method**. When the procedure has finished its task, it must return to the statement after the call. Therefore, the return address must be transmitted to the procedure or saved somewhere so that it can be located when it is time to return.

The return address may be placed in any of three places: memory, a register, or the stack. Far and away the worst solution is putting it in a single, fixed memory location. In this scheme, if the procedure called another procedure, the second call would cause the return address from the first one to be lost.

A slight improvement is having the procedure-call instruction store the return address in the first word of the procedure, with the first executable instruction being in the second word. The procedure can then return by branching indirectly to the first word or, if the hardware puts the opcode for branch in the first word along with the return address, branching directly to it. The procedure may call other procedures, because each procedure has space for one return address. If the procedure calls itself, this scheme fails, because the first return address will be destroyed by the second call. The ability for a procedure to call itself, called **recursion**, is exceedingly important for both theorists and practical programmers. Furthermore, if procedure *A* calls procedure *B*, and procedure *B* calls procedure *C*, and procedure *C* calls procedure *A* (indirect or daisy-chain recursion), this scheme also fails. This scheme for storing the return address in the first word of a procedure was used on the CDC 6600, the fastest computer in the world during much of the 1960s. The main language used on the 6600 was FORTRAN, which forbade recursion, so it worked then. But it was, and still is, a terrible idea.

A bigger improvement is to have the procedure-call instruction put the return address in a register, leaving the responsibility for storing it in a safe place to the procedure. If the procedure is recursive, it will have to put the return address in a different place each time it is called.

The best thing for the procedure-call instruction to do with the return address is to push it onto a stack. When the procedure has finished, it pops the return address off the stack and stuffs it into the program counter. If this form of procedure call is available, recursion does not cause any special problems; the return address will

automatically be saved in such a way as to avoid destroying previous return addresses. Recursion works just fine under these conditions. We saw this form of saving the return address in IJVM in Fig. 4-12.

5.5.6 Loop Control

The need to execute a group of instructions a fixed number of times occurs frequently and thus some machines have instructions to facilitate doing this. All the schemes involve a counter that is increased or decreased by some constant once each time through the loop. The counter is also tested once each time through the loop. If a certain condition holds, the loop is terminated.

One method initializes a counter outside the loop and then immediately begins executing the loop code. The last instruction of the loop updates the counter and, if the termination condition has not yet been satisfied, branches back to the first instruction of the loop. Otherwise, the loop is finished and it falls through, executing the first instruction beyond the loop. This form of looping is characterized as test-at-the-end (or post-test) type looping, and is illustrated in C in Fig. 5-29(a). (We could not use Java here because it does not have a goto statement.)

```
        i = 1;
L1:     first-statement;
        .
        .
        .
        last-statement;
        i = i + 1;
        if (i < n) goto L1;
```

(a)

```
        i = 1;
L1:     if (i > n) goto L2;
        first-statement;
        .
        .
        .
        last-statement
        i = i + 1;
        goto L1;
L2:
```

(b)

Figure 5-29. (a) Test-at-the-end loop. (b) Test-at-the-beginning loop.

Test-at-the-end looping has the property that the loop will always be executed at least once, even if n is less than or equal to 0. Consider, as an example, a program that maintains personnel records for a company. At a certain point in the program, it is reading information about a particular employee. It reads in n, the number of children the employee has, and executes a loop n times, once per child, reading the child's name, sex, and birthday, so that the company can send him or her a birthday present, one of the company's fringe benefits. If the employee does not have any children, n will be 0 but the loop will still be executed once sending presents and giving erroneous results.

Figure 5-29(b) shows another way of performing the test (pretest) that works properly even for n less than or equal to 0. Notice that the testing is different in the two cases, so that if a single ISA instruction does both the increment and the test, the designers are forced to choose one method or the other.

Consider the code that should be produced for the statement

```
for (i = 0; i < n; i++) { statements }
```

If the compiler does not have any information about n, it must use the approach of Fig. 5-29(b) to correctly handle the case of $n \leq 0$. If, however, it can determine that $n > 0$, for example, by seeing where n is assigned, it may use the better code of Fig. 5-29(a). The FORTRAN standard formerly stated that all loops were to be executed once, to allow the more efficient code of Fig. 5-29(a) to be generated all the time. In 1977, that defect was corrected when even the FORTRAN community began to realize that having a loop statement with outlandish semantics that sometimes gave the wrong answer was not a good idea, even if it did save one branch instruction per loop. C and Java have always done it right.

5.5.7 Input/Output

No other group of instructions exhibits as much variety from machine to machine as the I/O instructions. Three different I/O schemes are in current use in personal computers. These are

1. Programmed I/O with busy waiting.
2. Interrupt-driven I/O.
3. DMA I/O.

We now discuss each of these in turn.

The simplest possible I/O method is **programmed I/O**, which is commonly used in low-end microprocessors, for example, in embedded systems or in systems that must respond quickly to external changes (real-time systems). These CPUs usually have a single input instruction and a single output instruction. Each of these instructions selects one of the I/O devices. A single character is transferred between a fixed register in the processor and the selected I/O device. The processor must execute an explicit sequence of instructions for each and every character read or written.

As a simple example of this method, consider a terminal with four 1-byte registers, as shown in Fig. 5-30. Two registers are used for input, status and data, and two are used for output, also status and data. Each one has a unique address. If memory-mapped I/O is being used, all four registers are part of the computer's memory address space and can be read and written using ordinary instructions. Otherwise, special I/O instructions, say, IN and OUT, are provided to read and write them. In both cases, I/O is performed by transferring data and status information between the CPU and these registers.

The keyboard status register has 2 bits that are used and 6 that are not. The leftmost bit (7) is set to 1 by the hardware whenever a new character arrives. If the software has previously set bit 6, an interrupt is generated, otherwise it is not

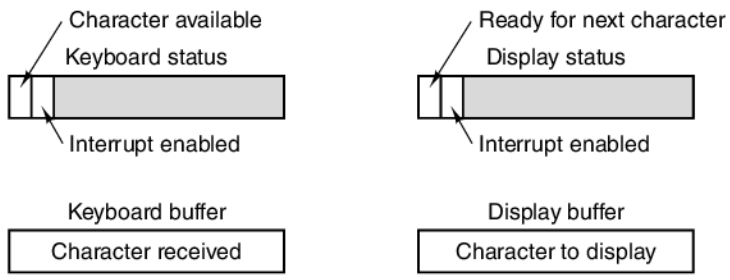

Figure 5-30. Device registers for a simple terminal.

(interrupts will be discussed shortly). When using programmed I/O, to get input, the CPU normally sits in a tight loop, repeatedly reading the keyboard status register, waiting for bit 7 to go on. When this happens, the software reads in the keyboard buffer register to get the character. Reading the keyboard data register causes the CHARACTER AVAILABLE bit to be reset to 0.

Output works in a similar way. To write a character to the screen, the software first reads the display status register to see if the READY bit is 1. If not, it loops until the bit goes to 1, indicating that the device is ready to accept a character. As soon as the terminal is ready, the software writes a character to the display buffer register, which causes it to be transmitted to the screen, and also causes the device to clear the READY bit in the display status register. When the character has been displayed and the terminal is prepared to handle the next character, the READY bit is automatically set to 1 again by the controller.

As an example of programmed I/O, consider the Java procedure of Fig. 5-31. It is called with two parameters: a character array to be output, and the count of characters present in the array, up to 1K. The body of the procedure is a loop that outputs characters one at a time. For each character, first the CPU must wait until the device is ready, then the character is output. The procedures *in* and *out* would typically be assembly-language routines to read and write the device registers specified by the first parameter from or to the variable specified as the second parameter. The implicit division by 128 by shifting gets rid of the low-order 7 bits, leaving the READY bit in bit 0.

The primary disadvantage of programmed I/O is that the CPU spends most of its time in a tight loop waiting for the device to become ready. This approach is called **busy waiting**. If the CPU has nothing else to do (e.g., the CPU in a washing machine), busy waiting may be OK (though even a simple controller often needs to monitor multiple, concurrent events). However, if there is other work to do, such as running other programs, busy waiting is wasteful, so a different I/O method is needed.

The way to get rid of busy waiting is to have the CPU start the I/O device and tell it to generate an interrupt when it is done. Looking at Fig. 5-30, we show how

```
public static void output_buffer(char buf[ ], int count) {
   // Output a block of data to the device
   int status, i, ready;

   for (i = 0; i < count; i++) {
      do {
         status = in(display_status_reg);        // get status
         ready = (status >> 7) & 0x01;           // isolate ready bit
      } while (ready != 1);
      out(display_buffer_reg, buf[i]);
   }
}
```

Figure 5-31. An example of programmed I/O.

this is done. By setting the INTERRUPT ENABLE bit in a device register, the software can request that the hardware give it a signal when the I/O is completed. We will study interrupts in detail later in this chapter when we come to flow of control.

It is worth mentioning that in many computers, the interrupt signal is generated by ANDing the INTERRUPT ENABLE bit with the READY bit. If the software first enables interrupts (before starting I/O), an interrupt will happen immediately, because the READY bit will be 1. Thus it may be necessary to first start the device, then immediately afterward enable interrupts. Writing a byte to the status register does not change the READY bit, which is read only.

Although interrupt-driven I/O is a big step forward compared to programmed I/O, it is far from perfect. The problem is that an interrupt is required for every character transmitted. Processing an interrupt is expensive. A way is needed to get rid of most of the interrupts.

The solution lies in going back to programmed I/O, but having somebody else do it. (The solution to many problems lies in having somebody else do the work.) Figure 5-32 shows how this is arranged. Here we have added a new chip, a **DMA** (**Direct Memory Access**) controller to the system, with direct access to the bus.

The DMA chip has (at least) four registers inside it, all of which can be loaded by software running on the CPU. The first contains the memory address to be read or written. The second contains the count of how many bytes (or words) are to be transferred. The third specifies the device number or I/O space address to use, thus specifying which I/O device is desired. The fourth tells whether data are to be read from or written to the I/O device.

To write a block of 32 bytes from memory address 100 to a terminal (say, device 4), the CPU writes the numbers 32, 100, and 4 into the first three DMA registers, and then the code for WRITE (say, 1) in the fourth one, as illustrated in Fig. 5-32. Once initialized like this, the DMA controller makes a bus request to read byte 100 from the memory, the same way the CPU would read from the memory. Having gotten this byte, the DMA controller then makes an I/O request to device 4 to write the byte to it. After both of these operations have been completed,

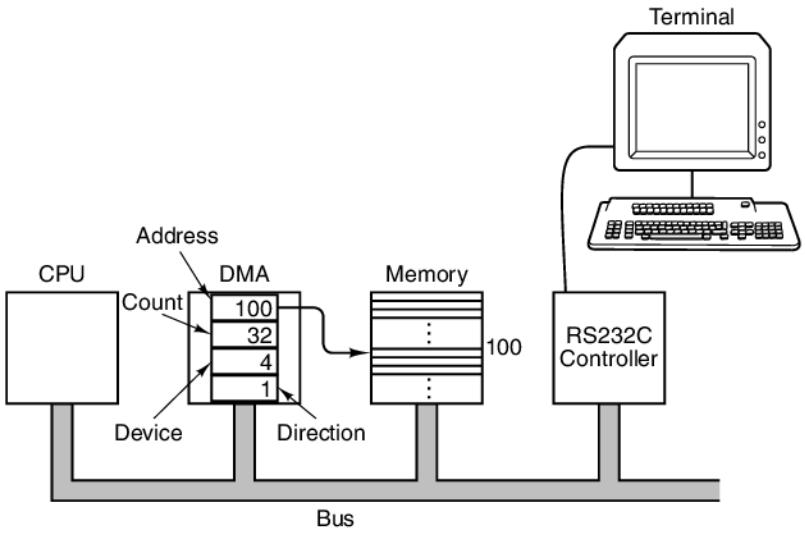

Figure 5-32. A system with a DMA controller.

the DMA controller increments its address register by 1 and decrements its count register by 1. If the count register is still greater than 0, another byte is read from memory and then written to the device.

When the count finally goes to 0, the DMA controller stops transferring data and asserts the interrupt line on the CPU chip. With DMA, the CPU only has to initialize a few registers. After that, it is free to do something else until the complete transfer is finished, at which time it gets an interrupt from the DMA controller. Some DMA controllers have two, or three, or more sets of registers, so they can control multiple simultaneous transfers.

While DMA greatly relieves the CPU from the burden of I/O, the process is not totally free. If a high-speed device, such as a disk, is being run by DMA, many bus cycles will be needed, both for memory references and device references. During these cycles the CPU will have to wait (DMA always has a higher bus priority than the CPU because I/O devices frequently cannot tolerate delays). The process of having a DMA controller take bus cycles away from the CPU is called **cycle stealing**. Nevertheless, the gain in not having to handle one interrupt per byte (or word) transferred far outweighs the loss due to cycle stealing.

5.5.8 The Core i7 Instructions

In this section and the next two, we will look at the instruction sets of our three example machines: the Core i7, the OMAP4430 ARM CPU, and the ATmega168 AVR. Each has a core of instructions that compilers normally generate, plus a set

Moves

MOV DST,SRC	Move SRC to DST
PUSH SRC	Push SRC onto the stack
POP DST	Pop a word from the stack to DST
XCHG DS1,DS2	Exchange DS1 and DS2
LEA DST,SRC	Load effective addr of SRC into DST
CMOVcc DST,SRC	Conditional move

Arithmetic

ADD DST,SRC	Add SRC to DST
SUB DST,SRC	Subtract SRC from DST
MUL SRC	Multiply EAX by SRC (unsigned)
IMUL SRC	Multiply EAX by SRC (signed)
DIV SRC	Divide EDX:EAX by SRC (unsigned)
IDIV SRC	Divide EDX:EAX by SRC (signed)
ADC DST,SRC	Add SRC to DST, then add carry bit
SBB DST,SRC	Subtract SRC & carry from DST
INC DST	Add 1 to DST
DEC DST	Subtract 1 from DST
NEG DST	Negate DST (subtract it from 0)

Binary coded decimal

DAA	Decimal adjust
DAS	Decimal adjust for subtraction
AAA	ASCII adjust for addition
AAS	ASCII adjust for subtraction
AAM	ASCII adjust for multiplication
AAD	ASCII adjust for division

Boolean

AND DST,SRC	Boolean AND SRC into DST
OR DST,SRC	Boolean OR SRC into DST
XOR DST,SRC	Boolean Exclusive OR SRC to DST
NOT DST	Replace DST with 1's complement

Shift/rotate

SAL/SAR DST,#	Shift DST left/right # bits
SHL/SHR DST,#	Logical shift DST left/right # bits
ROL/ROR DST,#	Rotate DST left/right # bits
RCL/RCR DST,#	Rotate DST through carry # bits

Test/compare

TEST SRC1,SRC2	Boolean AND operands, set flags
CMP SRC1,SRC2	Set flags based on SRC1 - SRC2

Transfer of control

JMP ADDR	Jump to ADDR
Jxx ADDR	Conditional jumps based on flags
CALL ADDR	Call procedure at ADDR
RET	Return from procedure
IRET	Return from interrupt
LOOPxx	Loop until condition met
INT n	Initiate a software interrupt
INTO	Interrupt if overflow bit is set

Strings

LODS	Load string
STOS	Store string
MOVS	Move string
CMPS	Compare two strings
SCAS	Scan Strings

Condition codes

STC	Set carry bit in EFLAGS register
CLC	Clear carry bit in EFLAGS register
CMC	Complement carry bit in EFLAGS
STD	Set direction bit in EFLAGS register
CLD	Clear direction bit in EFLAGS reg
STI	Set interrupt bit in EFLAGS register
CLI	Clear interrupt bit in EFLAGS reg
PUSHFD	Push EFLAGS register onto stack
POPFD	Pop EFLAGS register from stack
LAHF	Load AH from EFLAGS register
SAHF	Store AH in EFLAGS register

Miscellaneous

SWAP DST	Change endianness of DST
CWQ	Extend EAX to EDX:EAX for division
CWDE	Extend 16-bit number in AX to EAX
ENTER SIZE,LV	Create stack frame with SIZE bytes
LEAVE	Undo stack frame built by ENTER
NOP	No operation
HLT	Halt
IN AL,PORT	Input a byte from PORT to AL
OUT PORT,AL	Output a byte from AL to PORT
WAIT	Wait for an interrupt

SRC = source
DST = destination
# = shift/rotate count
LV = # locals

Figure 5-33. A selection of the Core i7 integer instructions.

of instructions that are rarely used, or are used only by the operating system. In our discussion, we will focus on the common instructions. Let us start with the Core i7. It is the most complicated. Then it is all downhill from there.

The Core i7 instruction set is a mixture of instructions that make sense in 32-bit mode and those that hark back to its former life as an 8088. In Fig. 5-33 we show a small selection of the more common integer instructions that compilers and programmers are likely to use these days. This list is far from complete, as it does not include any floating-point instructions, control instructions, or even some of the more exotic integer instructions (such as using an 8-bit byte in AL to perform table lookup). Nevertheless, it does give a good feel for what the Core i7 can do.

Many of the Core i7 instructions reference one or two operands, either in registers or in memory. For example, the two-operand ADD instruction adds the source to the destination and the one operand INC instruction increments (adds 1 to) its operand. Some of the instructions have several closely related variants. For example, the shift instructions can shift either left or right and can treat the sign bit specially or not. Most of the instructions have a variety of different encodings, depending on the nature of the operands.

In Fig. 5-33, the SRC fields are sources of information and are not changed. In contrast, the DST fields are destinations and are normally modified by the instruction. There are some rules about what is allowed as a source or a destination, varying somewhat erratically from instruction to instruction, but we will not go into them here. Many instructions have three variants, for 8-, 16-, and 32-bit operands, respectively. These are distinguished by different opcodes and/or a bit in the instruction. The list of Fig. 5-33 emphasizes the 32-bit instructions.

For convenience, we have divided the instructions into several groups. The first group contains instructions that move data around the machine, among registers, memory, and the stack. The second group does arithmetic, both signed and unsigned. For multiplication and division, the 64-bit product or dividend is stored in EAX (low-order part) and EDX (high-order part).

The third group does Binary Coded Decimal (BCD) arithmetic, treating each byte as two 4-bit **nibbles**. Each nibble holds one decimal digit (0 to 9). Bit combinations 1010 to 1111 are not used. Thus a 16-bit integer can hold a decimal number from 0 to 9999. While this form of storage is inefficient, it eliminates the need to convert decimal input to binary and then back to decimal for output. These instructions are used for doing arithmetic on the BCD numbers. They are heavily used by COBOL programs.

The Boolean and shift/rotate instructions manipulate the bits in a word or byte in various ways. Several combinations are provided.

The next two groups deal with testing and comparing, and then jumping based on the results. The results of test and compare instructions are stored in various bits of the EFLAGS register. Jxx stands for a set of instructions that conditionally jump, depending on the results of the previous comparison (i.e., bits in EFLAGS).

The Core i7 has several instructions for loading, storing, moving, comparing, and scanning strings of characters or words. These instructions can be prefixed by a special byte called REP, which cause them to be repeated until a certain condition is met, such as ECX, which is decremented after each iteration, reaching 0. In this

way, arbitrary blocks of data can be moved, compared, and so on. The next group manages the condition codes.

The last group is a hodge-podge of instructions that do not fit in anywhere else. These include conversions, stack frame management, stopping the CPU, and I/O.

The Core i7 has a number of **prefixes**, of which we have already mentioned one (REP). Each of these prefixes is a special byte that can precede most instructions, analogous to WIDE in IJVM. REP causes the instruction following it to be repeated until ECX hits 0, as mentioned above. REPZ and REPNZ repeatedly execute the following instruction until the *Z* condition code is set, or not set, respectively. LOCK reserves the bus for the entire instruction, to permit multiprocessor synchronization. Other prefixes are used to force an instruction to operate in 16-bit mode, or in 32-bit mode, which not only changes the length of the operands but also completely redefines the addressing modes. Finally, the Core i7 has a complex segmentation scheme with code, data, stack, and extra segments, a holdover from the 8088. Prefixes are provided to force memory references to use specific segments, but these will not be of concern to us (fortunately).

5.5.9 The OMAP4430 ARM CPU Instructions

Nearly all of the user-mode integer ARM instructions that a compiler might generate are listed in Fig. 5-34. Floating-point instructions are not given here, nor are control instructions (e.g., cache management, system reset), instructions involving address spaces other than the user's, or instruction extensions such as Thumb. The set is surprisingly small: the OMAP4430 ARM ISA really is a reduced instruction set computer.

The LDR and STR instructions are straightforward, with versions for 1, 2, and 4 bytes. When a number less than 32 bits is loaded into a (32-bit) register, the number can be either sign extended or zero extended. Instructions for both exist.

The next group is for arithmetic, which can optionally set the processor status register's condition code bits. On CISC machines, most instructions set the condition codes, but on a RISC machine that is undesirable because it restricts the compiler's freedom to move instructions around when trying to schedule them to tolerate instruction delays. If the original instruction order is A ... B ... C with A setting the condition codes and B testing them, the compiler cannot insert C between A and B if C sets the condition codes. For this reason, two versions of many instructions are provided, with the compiler normally using the one that does not set the condition codes, unless it is planning to test them later. The programmer specifies the setting of the condition codes by adding an "S" to the end of the instruction opcode name, for example, ADDS. A bit in the instruction indicates to the processor that the condition codes should be set. Multiplication and multiply-accumulate are also supported.

The shift group contains one left shift and two right shifts, each of which operate on 32-bit registers. The rotate right instruction does a circular rotation of bits

Loads

LDRSB DST,ADDR	Load signed byte (8 bits)
LDRB DST,ADDR	Load unsigned byte (8 bits)
LDRSH DST,ADDR	Load signed halfwords (16 bits)
LDRH DST,ADDR	Load unsigned halfwords (16 bits)
LDR DST,ADDR	Load word (32 bits)
LDM S1,REGLIST	Load multiple words

Stores

STRB DST,ADDR	Store byte (8 bits)
STRH DST,ADDR	Store halfword (16 bits)
STR DST,ADDR	Store word (32 bits)
STM SRC,REGLIST	Store multiple words

Arithmetic

ADD DST,S1,S2IMM	Add
ADD DST,S1,S2IMM	Add with carry
SUB DST,S1,S2IMM	Subtract
SUB DST,S1,S2IMM	Subtract with carry
RSB DST,S1,S2IMM	Reverse subtract
RSC DST,S1,S2IMM	Reverse subtract with carry
MUL DST,S1,S2	Multiply
MLA DST,S1,S2,S3	Multiple and accumulate
UMULL D1,D2,S1,S2	Unsigned long multiple
SMULL D1,D2,S1,S2	Signed long multiple
UMLAL D1,D2,S1,S2	Unsigned long MLA
SMLAL D1,D2,S1,S2	Signed long MLA
CMP S1,S2IMM	Compare and set PSR

Shifts/rotates

LSL DST,S1,S2IMM	Logical shift left
LSR DST,S1,S2IMM	Logical shift right
ASR DST,S1,S2IMM	Arithmetic shift right
ROR DSR,S1,S2IMM	Rotate right

Boolean

TST DST,S1,S2IMM	Test bits
TEQ DST,S1,S2IMM	Test equivalence
AND DST,S1,S2IMM	Boolean AND
EOR DST,S1,S2IMM	Boolean Exclusive-OR
ORR DST,S1,S2IMM	Boolean OR
BIC DST,S1,S2IMM	Bit clear

Transfer of control

Bcc IMM	Branch to PC+IMM
BLcc IMM	Branch with link to PC+IMM
BLcc S1	Branch with link to reg add

Miscellaneous

MOV DST,S1	Move register
MOVT DST,IMM	Move imm to upper bits
MVN DST,S1	NOT register
MRS DST,PSR	Read PSR
MSR PSR,S1	Write PSR
SWP DST,S1,ADDR	Swap reg/mem word
SWPB DST,S1,ADDR	Swap reg/mem byte
SWI IMM	Software interrupt

S1 = source register
S2IMM = source register or immediate
S3 = source register (when 3 are used)
DST = destination register
D1 = destination register (1 of 2)
D2 = destination register (2 of 2)

ADDR = memory address
IMM = immediate value
REGLIST = list of registers
PSR = processor status register
cc = branch condition

Figure 5-34. The primary OMAP4430 ARM CPU integer instructions.

within the register, such that bits that rotate off the least significant bit reappear as the most significant bit. The shifts are mostly used for bit manipulation. Rotates are useful for cryptographic and image-processing operations. Most CISC machines have a vast number of shift and rotate instructions, nearly all of them totally useless. Few compiler writers will spend restless nights mourning their absence.

The Boolean instruction group is analogous to the arithmetic one. It includes AND, EOR, ORR, TST, TEQ, and BIC. The latter three are of questionable value, but they can be done in one cycle and require almost no additional hardware so they got thrown in. Even RISC machine designers sometimes succumb to temptation.

The next instruction group contains the control transfers. *Bcc* represents a set of instructions that branch on the various conditions. *BLcc* is similar in that it

branches on various conditions, but it also deposits the address of the next instruction in the link register (R14). This instruction is useful to implement procedure calls. Unlike all other RISC architectures, there is no explicit branch to register address instruction. This instruction can be easily synthesized by using a MOV instruction with the destination set to the program counter (R15).

Two ways are provided for calling procedures. The first *BLcc* instruction uses the "Branch" format of Fig. 5-14 with a 24-bit PC-relative *word* offset. This value is enough to reach any instruction within 32 megabytes of the called in either direction. The second *BLcc* instruction jumps to the address in the specified register. This can be used to implement dynamically bound procedure calls (e.g., C++ virtual functions) or calls beyond the reach of 32 megabytes.

The last group contains some leftovers. MOVT is needed because there is no way to get a 32-bit immediate operand into a register. The way it is done is to use MOVT to set bits 16 through 31 and then have the next instruction supply the remaining bits using the immediate format. The MRS and MSR instructions allow reading and writing of the processor status word (PSR). The SWP instructions perform atomic swaps between a register and a memory location. These instruction implement the multiprocessor synchronization primitives that we will learn about in Chap. 8. Finally, the SWI instruction initiates a software interrupt, which is an overly fancy way of saying that it initiates a system call.

5.5.10 The ATmega168 AVR Instructions

The ATmega168 has a simple instruction set, shown in Fig. 5-35. Each line gives the mnemonic, a brief description, and a snippet of pseudocode that details the operation of the instruction. As is to be expected, there are a variety of MOV instructions for moving data between the registers. There are instructions for pushing and popping from a stack, which is pointed to by the 16-bit stack pointer (SP) in memory. Memory can be accessed with either an immediate address, register indirect, or register indirect plus a displacement. To allow up to 64 KB of addressing, the load with an immediate address is a 32-bit instruction. The indirect addressing modes utilize the X, Y, and Z register pairs, which combine two 8-bit registers to form a single 16-bit pointer.

The ATmega168 has simple arithmetic instructions for adding, subtracting, and multiplying, the latter of which use two registers. Incrementing and decrementing are also possible and commonly used. Boolean, shift, and rotate instructions are also present. The branch and call instruction can target an immediate address, a PC-relative address, or an address contained in the Z register pair.

5.5.11 Comparison of Instruction Sets

The three example instruction sets are very different. The Core i7 is a classic two-address 32-bit CISC machine, with a long history, peculiar and highly irregular addressing modes, and many instructions that reference memory.

Instruction	Description	Semantics
ADD DST,SRC	Add	DST ← DST + SRC
ADC DST,SRC	Add with Carry	DST ← DST + SRC + C
ADIW DST,IMM	Add Immediate to Word	DST+1:DST ← DST+1:DST + IMM
SUB DST,SRC	Subtract	DST ← DST - SRC
SUBI DST,IMM	Subtract Immediate	DST ← DST - IMM
SBC DST,SRC	Subtract with Carry	DST ← DST - SRC - C
SBCI DST,IMM	Subtract Immediate with Carry	DST ← DST - IMM – C
SBIW DST,IMM	Subtract Immediate from Word	DST+1:DST ← DST+1:DST - IMM
AND DST,SRC	Logical AND	DST ← DST AND SRC
ANDI DST,IMM	Logical AND with Immediate	DST ← DST AND IMM
OR DST,SRC	Logical OR	DST ← DST OR SRC
ORI DST,IMM	Logical OR with Immediate	DST ← DST OR IMM
EOR DST,SRC	Exclusive OR	DST ← DST XOR SRC
COM DST	One's Complement	DST ← 0xFF - DST
NEG DST	Two's Complement	DST ← 0x00 - DST
SBR DST,IMM	Set Bit(s) in Register	DST ← DST OR IMM
CBR DST,IMM	Clear Bit(s) in Register	DST ← DST AND (0xFF - IMM)
ING DST	Increment	DST ← DST + 1
DEC DST	Decrement	DST ← DST - 1
TST DST	Test for Zero or Minus	DST ← DST AND DST
CLR DST	Clear Register	DST ← DST XOR DST
SER DST	Set Register	DST ← 0xFF
MUL DST,SRC	Multiply Unsigned	R1:R0 ← DST * SRC
MULS DST,SRC	Multiply Signed	R1:R0 ← DST * SRC
MULSU DST,SRC	Multiply Signed with Unsigned	R1:R0 ← DST * SRC
RJMP IMM	PC-relative Jump	PC ← PC + IMM + 1
IJMP	Indirect Jump to Z	PC ← Z (R30:R31)
JMP IMM	Jump	PC ← IMM
RCALL IMM	Relative Call	STACK ← PC+2, PC ← PC + IMM + 1
ICALL	Indirect Call to (Z)	STACK ← PC+2, PC ← Z (R30:R31)
CALL	Call	STACK ← PC+2, PC ← IMM
RET	Return	PC ← STACK
CP DST,SRC	Compare	DST – SRC
CPC DST,SRC	Compare with Carry	DST – SRC – C
CPI DST,IMM	Compare with Immediate	DST – IMM
BRcc IMM	Branch on Condition	if cc(true) PC ← PC + IMM + 1
MOV DST,SRC	Copy Register	DST ← SRC
MOVW DST,SRC	Copy Register Pair	DST+1:DST ← SRC+1:SRC
LDI DST,IMM	Load Immediate	DST ← IMM
LDS DST,IMM	Load Direct	DST ← MEM[IMM]
LD DST,XYZ	Load Indirect	DST ← MEM[XYZ]
LDD DST,XYZ+IMM	Load Indirect with Displacement	DST ← MEM[XYZ+IMM]
STS IMM,SRC	Store Direct	MEM[IMM] ← SRC
ST XYZ,SRC	Store Indirect	MEM[XYZ] ← SRC
STD XYZ+IMM,SRC	Store Indirect with Displacement	MEM[XYZ+IMM] ← SRC
PUSH REGLIST	Push Register on Stack	STACK ← REGLIST
POP REGLIST	Pop Register from Stack	REGLIST ← STACK
LSL DST	Logical Shift Left by One	DST ← DST LSL 1
LSR DST	Logical Shift Right by One	DST ← DST LSR 1
ROL DST	Rotate Left by One	DST ← DST ROL 1
ROR DST	Rotate Right by One	DST ← DST ROR 1
ASR DST	Arithmetic Shift Right by One	DST ← DST ASR 1

SRS = source register
DST = destination register
IMM = immediate value

XYZ = X, Y, or Z register pair
MEM[A] = access memory at address A

Figure 5-35. The ATmega168 AVR instruction set.

The OMAP4430 ARM CPU is a modern three-address 32-bit RISC, with a load/store architecture, hardly any addressing modes, and a compact and efficient instruction set. The ATmega168 AVR architecture is a tiny embedded processor designed to fit on a single chip.

Each machine is the way it is for a good reason. The Core i7's design was determined by three major factors:

1. Backward compatibility.
2. Backward compatibility.
3. Backward compatibility.

Given the current state of the art, no one would now design such an irregular machine with so few registers, all of them different. This makes compilers hard to write. The lack of registers also forces compilers to constantly spill variables into memory and then reload them, an expensive business, even with two or three levels of caching. It is a testimonial to the quality of Intel's engineers that the Core i7 is so fast, even with the constraints of this ISA. But as we saw in Chap. 4, the implementation is exceedingly complex.

The OMAP4430 ARM represents a state-of-the-art ISA design. It has a full 32-bit ISA. It has many registers, an instruction set that emphasizes three-register operations, plus a small group of LOAD and STORE instructions. All instructions are the same size, although the number of formats has gotten a bit out of hand. Still, it lends itself to a straightforward and efficient implementation. Most new designs tend to look like the OMAP4430 ARM architecture, but with fewer instruction formats.

The ATmega168 AVR has a simple and fairly regular instruction set with relatively few instructions and few addressing modes. It is distinguished by having 32 8-bit registers, rapid access to data, a way to access registers in the memory space, and surprisingly powerful bit-manipulation instructions. Its main claim to fame is that it can be implemented with a very small number of transistors, thus making it possible to put a large number on a die, which keeps the cost per CPU very low.

5.6 FLOW OF CONTROL

Flow of control refers to the sequence in which instructions are executed dynamically, that is, during program execution. In general, in the absence of branches and procedure calls, successively executed instructions are fetched from consecutive memory locations. Procedure calls cause the flow of control to be altered, stopping the procedure currently executing and starting the called procedure. Coroutines are related to procedures and cause similar alterations in the flow of control. They are useful for simulating parallel processes. Traps and interrupts

also cause the flow of control to be altered when special conditions occur. All these topics will be discussed in the following sections.

5.6.1 Sequential Flow of Control and Branches

Most instructions do not alter the flow of control. After an instruction is executed, the one following it in memory is fetched and executed. After each instruction, the program counter is increased by the instruction length. If observed over an interval of time that is long compared to the average instruction time, the program counter is approximately a linear function of time, increasing by the average instruction length per average instruction time. Stated another way, the dynamic order in which the processor actually executes the instructions is the same as the order in which they appear on the program listing, as shown in Fig. 5-36(a). If a program contains branches, this simple relation between the order in which instructions appear in memory and the order in which they are executed is no longer true. When branches are present, the program counter is no longer a monotonically increasing function of time, as shown in Fig. 5-36(b). As a result, it becomes difficult to visualize the instruction execution sequence from the program listing.

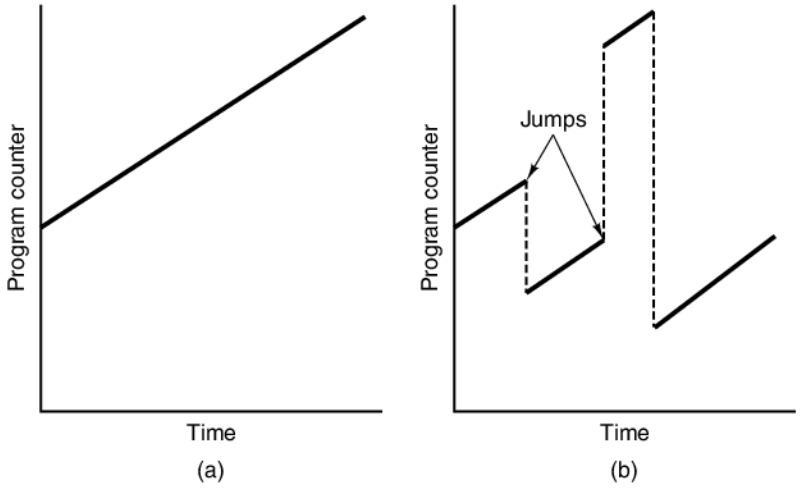

Figure 5-36. Program counter as a function of time (smoothed). (a) Without branches. (b) With branches.

When programmers have trouble keeping track of the sequence in which the processor will execute the instructions, they are prone to make errors. This observation led Dijkstra (1968a) to write a then-controversial letter entitled "GO TO Statement Considered Harmful," in which he suggested avoiding goto statements. That letter gave birth to the structured programming revolution, one of whose tenets is the replacement of goto statements with more structured forms of flow

control, such as while loops. Of course, these programs compile down to level 2 programs that may contain many branches, because the implementation of if, while, and other high-level control structures requires branching around.

5.6.2 Procedures

The most important technique for structuring programs is the procedure. From one point of view, a procedure call alters the flow of control just as a branch does, but unlike the branch, when finished performing its task, it returns control to the statement or instruction following the call.

However, from another point of view, a procedure body can be regarded as defining a new instruction on a higher level. From this standpoint, a procedure call can be thought of as a single instruction, even though the procedure may be quite complicated. To understand a piece of code containing a procedure call, it is only necessary to know *what* it does, not *how* it does it.

One particularly interesting kind of procedure is the **recursive procedure**, that is, a procedure that calls itself, either directly or indirectly via a chain of other procedures. Studying recursive procedures gives considerable insight into how procedure calls are implemented, and what local variables really are. Now we will give an example of a recursive procedure.

The "Towers of Hanoi" is an ancient problem that has a simple solution involving recursion. In a certain monastery in Hanoi, there are three gold pegs. Around the first one were a series of 64 concentric gold disks, each with a hole in the middle for the peg. Each disk is slightly smaller in diameter than the disk directly below it. The second and third pegs were initially empty. The monks there are busily transferring all the disks to peg 3, one disk at a time, but at no time may a larger disk rest on a smaller one. When they finish, it is said the world will come to an end. If you wish to get hands-on experience, it is all right to use plastic disks and fewer of them, but when you solve the problem, nothing will happen. To get the end-of-world effect, you need 64 of them and in gold. Figure 5-37 shows the initial configuration for $n = 5$ disks.

The solution of moving n disks from peg 1 to peg 3 consists first of moving $n - 1$ disks from peg 1 to peg 2, then moving 1 disk from peg 1 to peg 3, then moving $n - 1$ disks from peg 2 to peg 3. This solution is illustrated in Fig. 5-38.

To solve the problem we need a procedure to move n disks from peg i to peg j. When this procedure is called, by

```
towers(n, i, j)
```

the solution is printed out. The procedure first makes a test to see if $n = 1$. If so, the solution is trivial, just move the one disk from i to j. If $n \neq 1$, the solution consists of three parts as discussed above, each being a recursive procedure call.

The complete solution is shown in Fig. 5-39. The call

```
towers(3, 1, 3)
```

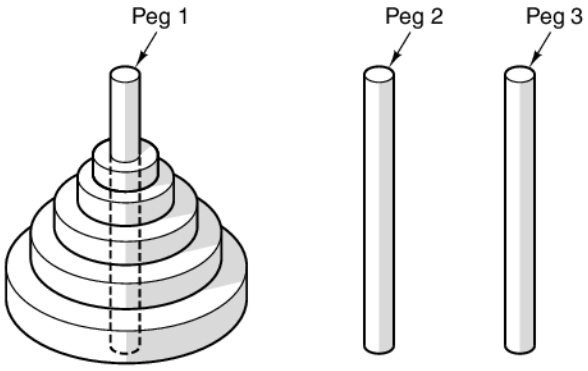

Figure 5-37. Initial configuration for the Towers of Hanoi problem for five disks.

to solve the problem of Fig. 5-38 generates three more calls. Specifically, it makes the calls

```
towers(2, 1, 2)
towers(1, 1, 3)
towers(2, 2, 3)
```

The first and third will generate three calls each, for a total of seven.

In order to have recursive procedures, we need a stack to store the parameters and local variables for each invocation, the same as we had in IJVM. Each time a procedure is called, a new stack frame is allocated for the procedure on top of the stack. The frame most recently created is the current frame. In our examples, the stack grows upward, from low memory addresses to high ones, just like in IJVM. So the most recent frame has higher addresses than all the others.

In addition to the stack pointer, which points to the top of the stack, it is often convenient to have a frame pointer, FP, which points to a fixed location within the frame. It could point to the link pointer, as in IJVM, or to the first local variable. Figure 5-39 shows the stack frame for a machine with a 32-bit word. The original call to *towers* pushes *n*, *i*, and *j* onto the stack and then executes a CALL instruction that pushes the return address onto the stack, at address 1012. On entry, the called procedure stores the old value of FP on the stack at 1016 and then advances the stack pointer to allocate storage for the local variables. With only one 32-bit local variable (*k*), SP is incremented by 4 to 1020. The situation, after all these things have been done, is shown in Fig. 5-39(a).

The first thing a procedure must do when called is save the previous FP (so it can be restored at procedure exit), copy SP into FP, and possibly increment by one word, depending on where in the new frame FP points. In this example, FP points to the first local variable, but in IJVM, LV pointed to the link pointer. Different

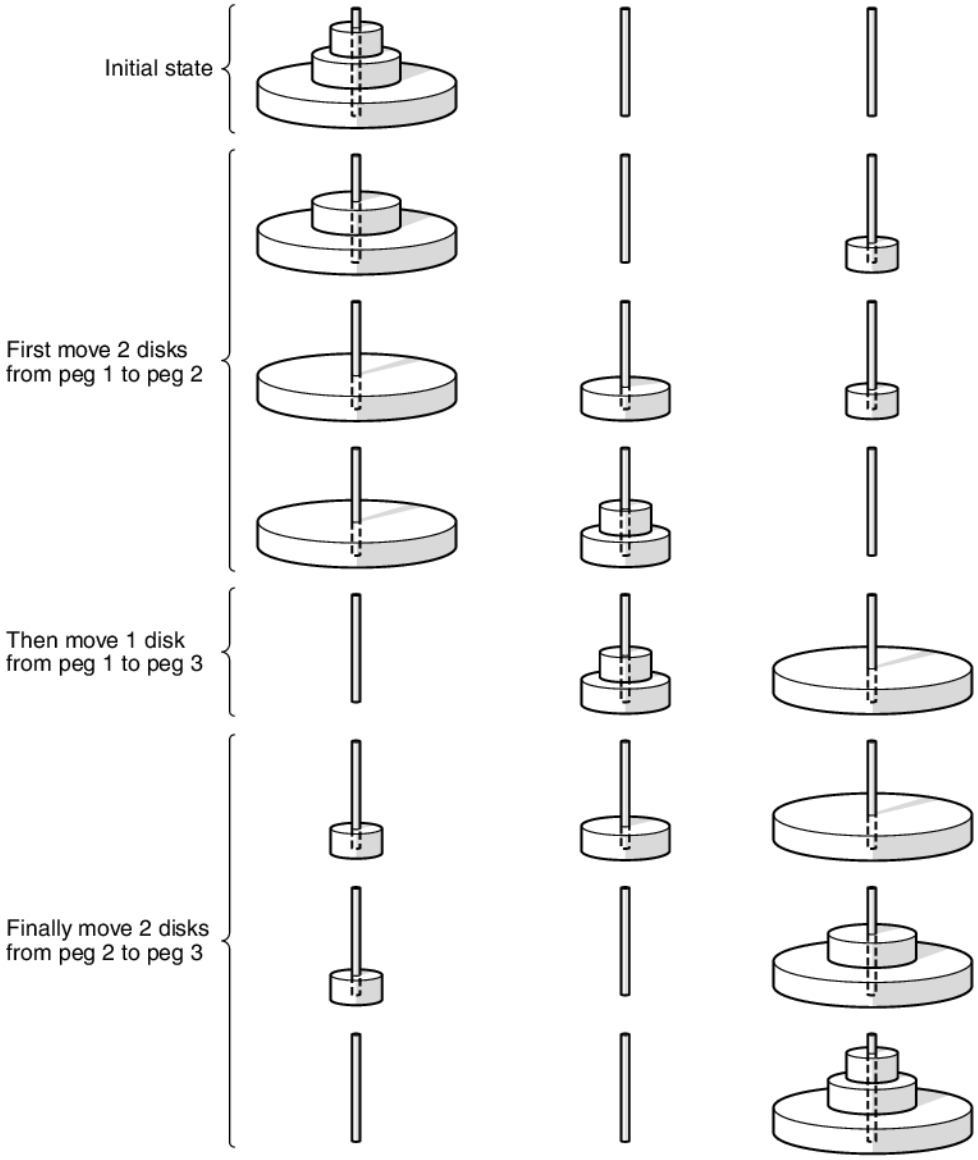

Figure 5-38. The steps required to solve the Towers of Hanoi for three disks.

machines handle the frame pointer slightly differently, sometimes putting it at the bottom of the stack frame, sometimes at the top, and sometimes in the middle as in Fig. 5-40. In this respect, it is worth comparing Fig. 5-40 with Fig. 4-12 to see two different ways to manage the link pointer. Other ways are also possible. In all cases, the key is the ability to later be able to do a procedure return and restore the

```
public void towers(int n, int i, int j) {
   int k;

   if (n == 1)
      System.out.println("Move a disk from " + i + " to " + j);
   else {
      k = 6 – i – j;
      towers(n – 1, i, k);
      towers(1, i, j);
      towers(n – 1, k, j);
 }
}
```

Figure 5-39. A procedure for solving the Towers of Hanoi.

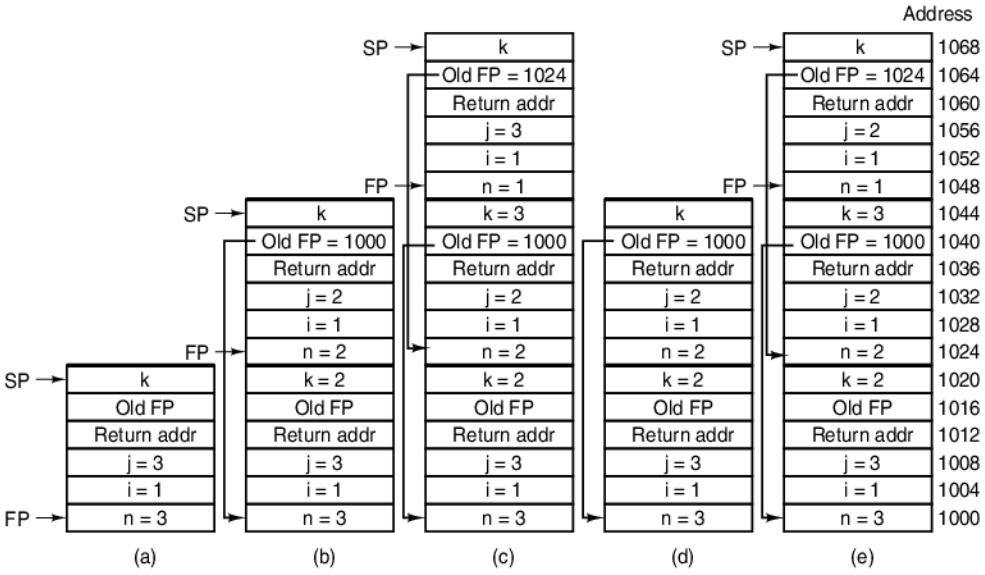

Figure 5-40. The stack at several points during the execution of Fig. 5-39.

state of the stack to what it was just prior to the current procedure invocation.

The code that saves the old frame pointer, sets up the new one, and advances the stack pointer to reserve space for local variables is called the **procedure prolog**. Upon procedure exit, the stack must be cleaned up again, something called the **procedure epilog**. One of the most important characteristics of any computer is how short and fast it can make the prolog and epilog. If they are long and slow, procedure calls will be expensive. Programmers who worship at the altar of efficiency will learn to avoid writing many short procedures and write large, monolithic, unstructured programs instead. The Core i7 ENTER and LEAVE instructions

have been designed to do most of the procedure prolog and epilog work efficiently. Of course, they have a particular model of how the frame pointer should be managed, and if the compiler has a different model, they cannot be used.

Now let us get back to the Towers of Hanoi problem. Each procedure call adds a new frame to the stack and each procedure return removes a frame from the stack. In order to illustrate the use of a stack in implementing recursive procedures, we will trace the calls starting with

```
towers(3, 1, 3)
```

Figure 5-40(a) shows the stack just after this call has been made. The procedure first tests to see if $n = 1$, and on discovering that $n = 3$, fills in k and makes the call

```
towers(2, 1, 2)
```

After this call is completed the stack is as shown in Fig. 5-40(b), and the procedure starts again at the beginning (a called procedure always starts at the beginning). This time the test for $n = 1$ fails again, so it fills in k again and makes the call

```
towers(1, 1, 3)
```

The stack then is as shown in Fig. 5-40(c) and the program counter points to the start of the procedure. This time the test succeeds and a line is printed. Next, the procedure returns by removing one stack frame, resetting FP and SP to Fig. 5-40(d). It then continues executing at the return address, which is the second call:

```
towers(1, 1, 2)
```

This adds a new frame to the stack as shown in Fig. 5-40(e). Another line is printed; after the return a frame is removed from the stack. The procedure calls continue in this way until the original call completes execution and the frame of Fig. 5-40(a) is removed from the stack. To best understand how recursion works, it is recommended that you simulate the complete execution of

```
towers(3, 1, 3)
```

using pencil and paper.

5.6.3 Coroutines

In the usual calling sequence, there is a clear distinction between the calling procedure and the called procedure. Consider a procedure *A*, on the left which calls a procedure *B* on the right in Fig. 5-41.

Procedure *B* computes for a while and then afterwards returns to *A*. At first sight you might consider this situation symmetric, because neither *A* nor *B* is a main program, both being procedures. (Procedure *A* may have been called by the

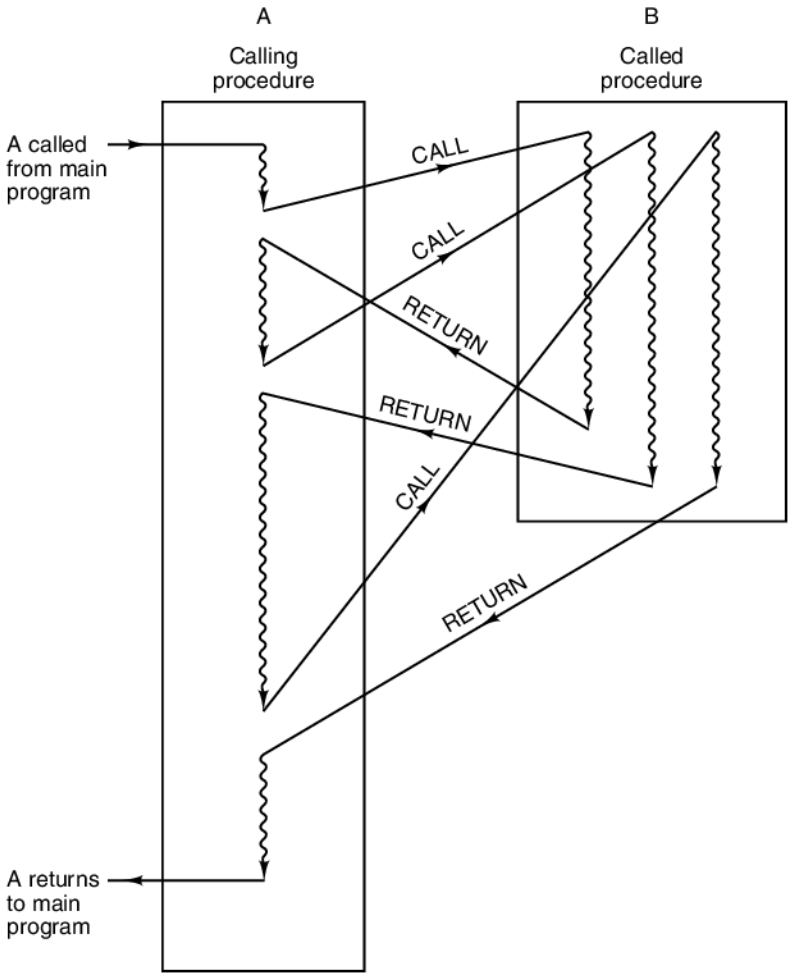

Figure 5-41. When a procedure is called, execution of the procedure always begins at the first statement of the procedure.

main program but that is irrelevant.) Furthermore, first control is transferred from *A* to *B*—the call—and later control is transferred from *B* to *A*—the return.

The asymmetry arises from the fact that when control passes from *A* to *B*, procedure *B* begins executing at the beginning; when *B* returns to *A*, execution starts not at the beginning of *A* but at the statement following the call. If *A* runs for a while and calls *B* again, execution starts at the beginning of *B* again, not at the statement following the previous return. If, in the course of running, *A* calls *B* many times, *B* starts at the beginning all over again each and every time, whereas *A* never starts over again. It just keeps going forward.

This difference is reflected in the method by which control is passed between *A* and *B*. When *A* calls *B*, it uses the procedure call instruction, which puts the return address (i.e., the address of the statement following the call) somewhere useful, for example, on top of the stack. It then puts the address of *B* into the program counter to complete the call. When *B* returns, it does not use the call instruction but instead it uses the return instruction, which simply pops the return address from the stack and puts it into the program counter.

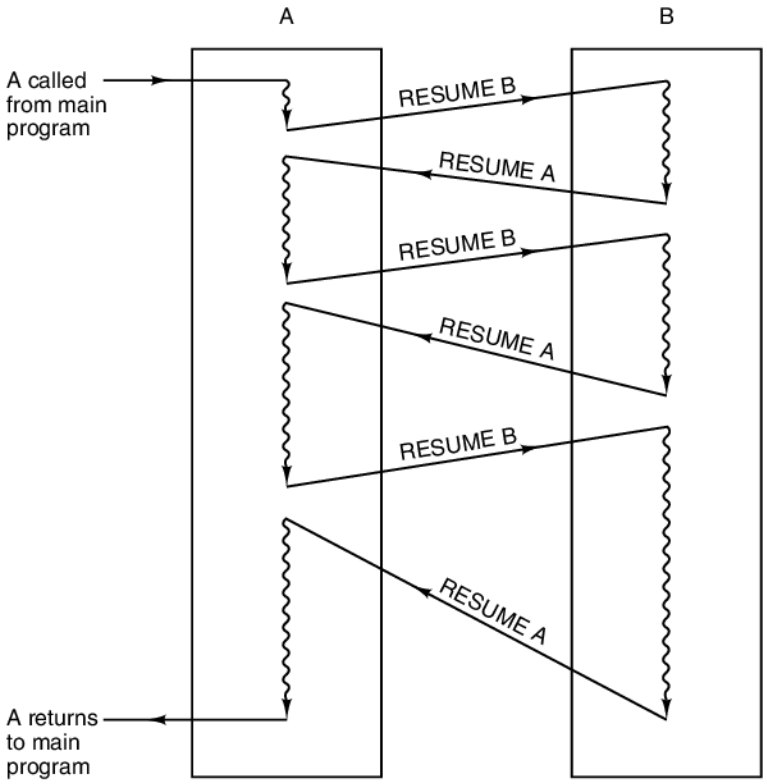

Figure 5-42. When a coroutine is resumed, execution begins at the statement where it left off the previous time, not at the beginning.

Sometimes it is useful to have two procedures, *A* and *B*, each of which calls the other as a procedure, as shown in Fig. 5-42. When *B* returns to *A*, it branches to the statement following the call to *B*, as above. When *A* transfers control to *B*, it does not go to the beginning (except the first time) but to the statement following the most recent "return," that is, the most recent call of *A*. Two procedures that work this way are called **coroutines**.

A common use for coroutines is to simulate parallel processing on a single CPU. Each coroutine runs in pseudoparallel with the others, as though it had its

own CPU. This style of programming makes programming some applications easier. It also is useful for testing software that will later actually run on a multiprocessor.

Neither the usual CALL nor the usual RETURN instruction works for calling coroutines, because the address to branch to comes from the stack like a return, but, unlike a return, the coroutine call itself puts a return address somewhere for the subsequent return to it. It would be nice if there were an instruction to exchange the top of the stack with the program counter. In detail, this instruction would first pop the old return address off the stack into an internal register, then push the program counter onto the stack, and finally, copy the internal register into the program counter. Because one word is popped off the stack and one word is pushed onto the stack, the stack pointer does not change. This instruction rarely exists, so in most cases it has to be simulated as several instructions.

5.6.4 Traps

A **trap** is a kind of automatic procedure call initiated by some condition caused by the program, usually an important but rarely occurring condition. A good example is overflow. On many computers, if the result of an arithmetic operation exceeds the largest number that can be represented, a trap occurs, meaning that the flow of control is switched to some fixed memory location instead of continuing in sequence. At that fixed location is a branch to a procedure called the **trap handler**, which performs some appropriate action, such as printing an error message. If the result of an operation is within range, no trap occurs.

The essential point about a trap is that it is initiated by some exceptional condition caused by the program itself and detected by the hardware or microprogram. An alternative method of handling overflow is to have a 1-bit register that is set to 1 whenever an overflow occurs. A programmer who wants to check for overflow must include an explicit "branch if overflow bit is set" instruction after every arithmetic instruction. Doing so is both slow and wasteful of space. Traps save both time and memory compared with explicit programmer-controlled checking.

The trap may be implemented by an explicit test performed by the microprogram (or hardware). If an overflow is detected, the trap address is loaded into the program counter. What is a trap at one level may be under program control at a lower level. Having the microprogram make the test still saves time compared to a programmer test, because it can be easily overlapped with something else. It also saves memory, because it need occur only in one place, for example, the main loop of the microprogram, independent of how many arithmetic instructions occur in the main program.

A few of the common conditions that can cause traps are floating-point overflow, floating-point underflow, integer overflow, protection violation, undefined opcode, stack overflow, attempt to start a nonexistent I/O device, attempt to fetch a word from an odd-numbered address, and division by zero.

5.6.5 Interrupts

Interrupts are changes in the flow of control caused not by the running program, but by something else, usually related to I/O. For example, a program may instruct the disk to start transferring information, and set the disk up to provide an interrupt as soon as the transfer is finished. Like the trap, the interrupt stops the running program and transfers control to an interrupt handler, which performs some appropriate action. When finished, the interrupt handler returns control to the interrupted program. It must restart the interrupted process in exactly the same state that it was in when the interrupt occurred, which means restoring all the internal registers to their preinterrupt state.

The essential difference between traps and interrupts is this: *traps* are synchronous with the program and *interrupts* are asynchronous. If the program is rerun a million times with the same input, the traps will reoccur in the same place each time but the interrupts may vary, depending, for example, on precisely when a person at a terminal hits carriage return. The reason for the reproducibility of traps and irreproducibility of interrupts is that traps are caused directly by the program and interrupts are, at best, indirectly caused by the program.

To see how interrupts really work, let us consider a common example: a computer wants to output a line of characters to a terminal. The system software first collects all the characters to be written to the terminal together in a buffer, initializes a global variable *ptr*, to point to the start of the buffer, and sets a second global variable *count* equal to the number of characters to be output. Then it checks to see if the terminal is ready and if so, outputs the first character (e.g., using registers like those of Fig. 5-30). Having started the I/O, the CPU is then free to run another program or do something else.

In due course of time, the character is displayed on the screen. The interrupt can now begin. In simplified form, the steps are as follows.

HARDWARE ACTIONS

1. The device controller asserts an interrupt line on the system bus to start the interrupt sequence.
2. As soon as the CPU is prepared to handle the interrupt, it asserts an interrupt acknowledge signal on the bus.
3. When the device controller sees that its interrupt signal has been acknowledged, it puts a small integer on the data lines to identify itself. This number is called the **interrupt vector**.
4. The CPU removes the interrupt vector from the bus and saves it temporarily.
5. Then the CPU pushes the program counter and PSW onto the stack.

6. The CPU then locates a new program counter by using the interrupt vector as an index into a table in low memory. If the program counter is 4 bytes, for example, then interrupt vector n corresponds to address $4n$. This new program counter points to the start of the interrupt service routine for the device causing the interrupt. Often the PSW is loaded or modified as well (e.g., to disable further interrupts).

SOFTWARE ACTIONS

7. The first thing the interrupt service routine does is save all the registers it uses so they can be restored later. They can be saved on the stack or in a system table.
8. Each interrupt vector is generally shared by all the devices of a given type, so it is not yet known which terminal caused the interrupt. The terminal number can be found by reading some device register.
9. Any other information about the interrupt, such as status codes, can now be read in.
10. If an I/O error occurred, it can be handled here.
11. The global variables, *ptr* and *count*, are updated. The former is incremented, to point to the next byte, and the latter is decremented, to indicate that 1 byte fewer remains to be output. If *count* is still greater than 0, there are more characters to output. Copy the one now pointed to by *ptr* to the output buffer register.
12. If required, a special code is output to tell the device or the interrupt controller that the interrupt has been processed.
13. Restore all the saved registers.
14. Execute the RETURN FROM INTERRUPT instruction, putting the CPU back into the mode and state it had just before the interrupt happened. The computer then continues from where it was.

A key concept related to interrupts is **transparency**. When an interrupt happens, some actions are taken and some code runs, but when everything is finished, the computer should be returned to exactly the same state it had before the interrupt. An interrupt routine with this property is said to be transparent. Having all interrupts be transparent makes interrupts much easier to understand.

If a computer has only one I/O device, then interrupts always work as we have just described, and there is nothing more to say about them. However, a large computer may have many I/O devices, and several may be running at the same time, possibly on behalf of different users. A nonzero probability exists that while an interrupt routine is running, a second I/O device wants to generate *its* interrupt.

Two approaches can be taken to this problem. The first one is for all interrupt routines to disable subsequent interrupts as the very first thing they do, even before saving the registers. This approach keeps things simple, as interrupts are then taken strictly sequentially, but it can lead to problems for devices that cannot tolerate much delay. If the first one has not yet been processed when the second one arrives, data may be lost.

When a computer has time-critical I/O devices, a better design approach is to assign each I/O device a priority, high for very critical devices and low for less critical ones. Similarly, the CPU should also have priorities, typically determined by a field in the PSW. When a priority n device interrupts, the interrupt routine should also run at priority n.

While a priority n interrupt routine is running, any attempt by a device with a lower priority to cause an interrupt is ignored until the interrupt routine is finished and the CPU goes back to running lower-priority code. On the other hand, interrupts from higher-priority devices should be allowed to happen with no delay.

With interrupt routines themselves subject to interrupt, the best way to keep the administration straight is to make sure that all interrupts are transparent. Let us consider a simple example of multiple interrupts. A computer has three I/O devices, a printer, a disk, and an RS232 (serial) line, with priorities 2, 4, and 5, respectively. Initially ($t = 0$), a user program is running, when suddenly at $t = 10$ a printer interrupt occurs. The printer Interrupt Service Routine (ISR) is started up, as shown in Fig. 5-43.

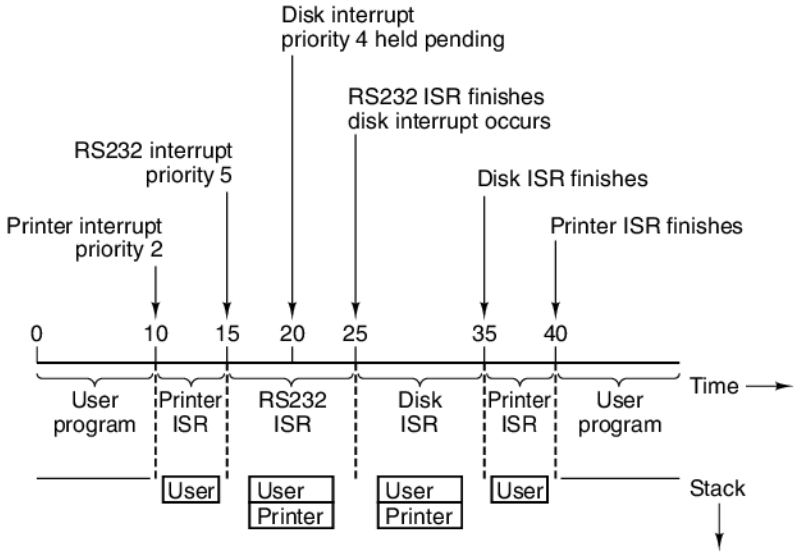

Figure 5-43. Time sequence of multiple-interrupt example.

At $t = 15$, the RS232 line wants attention and generates an interrupt. Since the RS232 line has a higher priority (5) than the printer (2), the interrupt happens. The

state of the machine, which is now running the printer interrupt service routine, is pushed onto the stack, and the RS232 interrupt service routine is started.

A little later, at $t = 20$, the disk is finished and wants service. However, its priority (4) is lower than that of the interrupt routine currently running (5), so the CPU hardware does not acknowledge the interrupt, and it is held pending. At $t = 25$, the RS232 routine is finished, so it returns to the state it was in just before the RS232 interrupt happened, namely, running the printer interrupt service routine at priority 2. As soon as the CPU switches to priority 2, before even one instruction can be executed, the disk interrupt at priority 4 now is allowed in, and the disk service routine runs. When it finishes, the printer routine gets to continue. Finally, at $t = 40$, all the interrupt service routines have completed and the user program continues from where it left off.

Since the 8088, the Intel CPU chips have had two interrupt levels (priorities): maskable and nonmaskable. Nonmaskable-interrupts are generally used only for signaling near-catastrophes, such as memory parity errors. All the I/O devices use the one maskable interrupt.

When an I/O device issues an interrupt, the CPU uses the interrupt vector to index into a 256-entry table to find the address of the interrupt service routine. The table entries are 8-byte segment descriptors and the table can begin anywhere in memory. A global register points to its start.

With only one usable interrupt level, there is no way for the CPU to let a high-priority device interrupt a medium-priority interrupt service routine while prohibiting a low-priority device from doing so. To solve this problem, the Intel CPUs are normally used with an external interrupt controller (e.g., an 8259A). When the first interrupt comes in, say at priority n, the CPU is interrupted. If a subsequent interrupt comes in at a higher priority, the interrupt controller interrupts a second time. If the second interrupt is at a lower priority, it is held until the first one is finished. To make this scheme work, the interrupt controller must know when the current interrupt service routine is finished, so the CPU must send it a command when the current interrupt has been fully processed.

5.7 A DETAILED EXAMPLE: THE TOWERS OF HANOI

Now that we have studied the ISA of three machines, let us put all the pieces together by taking a close look at the same example program for the two larger machines. Our example is the Towers of Hanoi program. We gave a Java version of this program in Fig. 5-39. In the following sections we will give assembly-code programs for the Towers of Hanoi.

However, we will cheat a tiny bit. Rather than give the translation of the Java version, for the Core i7 and OMAP4430 ARM CPU we will give the translation of a C version to avoid some problems with Java I/O. The only difference is the replacement of the Java call to *println* with the standard C statement

```
printf("Move a disk from %d to %d\n", i, j)
```

For our purposes, the syntax of *printf* format strings is unimportant (basically, the string is printed literally except that *%d* means print the next integer in decimal). The only thing that is relevant here is that the procedure is called with three parameters: a format string and two integers.

The reason for using the C version for the Core i7 and OMAP4430 ARM CPU is that the Java I/O library is not available in native form for these machines; the C library is. The difference is minimal, affecting only the print statement.

5.7.1 The Towers of Hanoi in Core i7 Assembly Language

Figure 5-44 gives a possible translation of the C version of the Towers of Hanoi for the Core i7. For the most part, it is fairly straightforward. The EBP register is used as the frame pointer. The first two words are used for linkage, so the first actual parameter, n (or N here, as MASM is case insensitive), is at EBP + 8, followed by i and j at EBP +12 and EBP + 16, respectively. The local variable, k, is in EBP + 20.

The procedure begins by establishing the new frame at the end of the old one. It does this by copying ESP to the frame pointer, EBP. Then it compares n to 1, branching off to the else clause if $n > 1$. The then code pushes three values on the stack: the address of the format string, i, and j, and calls itself.

The parameters are pushed in reverse order, which is required for C programs. This is necessary to put the pointer to the format string on top of the stack. Since *printf* has a variable number of parameters, if the parameters were pushed in forward order, *printf* would not know how deep in the stack the format string was.

After the call, 12 is added to ESP to remove the parameters from the stack. Of course, they are not really erased from memory, but the adjustment of ESP makes them inaccessible via the normal stack operations.

The else clause, which starts at *L1*, is straightforward. It first computes $6 - i - j$ and stores this value in k. No matter what values i and j have, the third peg is always $6 - i - j$. Saving it in k saves the trouble of recomputing it the second time.

Next, the procedure calls itself three times, with different parameters each time. After each call, the stack is cleaned up. That is all there is to it.

Recursive procedures sometimes confuse people at first, but when viewed at this level, they are straightforward. All that happens is that the parameters are pushed onto the stack and the procedure itself is called.

5.7.2 The Towers of Hanoi in OMAP4430 ARM Assembly Language

Now let us try again, only this time for the OMAP4430 ARM. The code is listed in Fig. 5-45. Because the OMAP4430 ARM code is especially unreadable, even for assembly code and even after a lot of practice, we have taken the liberty to

```
        .686                            ; compile for Core i7 class processor
.MODEL FLAT
PUBLIC _towers                          ; export 'towers'
EXTERN _printf:NEAR                     ; import printf
.CODE
_towers: PUSH EBP                       ; save EBP (frame pointer) and decrement ESP
        MOV EBP, ESP                    ; set new frame pointer above ESP
        CMP [EBP+8], 1                  ; if (n == 1)
        JNE L1                          ; branch if n is not 1
        MOV EAX, [EBP+16]               ; printf(" ...", i, j);
        PUSH EAX                        ; note that parameters i, j and the format
        MOV EAX, [EBP+12]               ; string are pushed onto the stack
        PUSH EAX                        ; in reverse order. This is the C calling convention
        PUSH OFFSET FLAT:format         ; offset flat means the address of format
        CALL _printf                    ; call printf
        ADD ESP, 12                     ; remove params from the stack
        JMP Done                        ; we are finished
L1:     MOV EAX, 6                      ; start k = 6 – i – j
        SUB EAX, [EBP+12]               ; EAX = 6 – i
        SUB EAX, [EBP+16]               ; EAX = 6 – i – j
        MOV [EBP+20], EAX               ; k = EAX
        PUSH EAX                        ; start towers(n – 1, i, k)
        MOV EAX, [EBP+12]               ; EAX = i
        PUSH EAX                        ; push i
        MOV EAX, [EBP+8]                ; EAX = n
        DEC EAX                         ; EAX = n – 1
        PUSH EAX                        ; push n – 1
        CALL _towers                    ; call towers(n – 1, i, 6 – i – j)
        ADD ESP, 12                     ; remove params from the stack
        MOV EAX, [EBP+16]               ; start towers(1, i, j)
        PUSH EAX                        ; push j
        MOV EAX, [EBP+12]               ; EAX = i
        PUSH EAX                        ; push i
        PUSH 1                          ; push 1
        CALL _towers                    ; call towers(1, i, j)
        ADD ESP, 12                     ; remove params from the stack
        MOV EAX, [EBP+12]               ; start towers(n – 1, 6 – i – j, i)
        PUSH EAX                        ; push i
        MOV EAX, [EBP+20]               ; EAX = k
        PUSH EAX                        ; push k
        MOV EAX, [EBP+8]                ; EAX = n
        DEC EAX                         ; EAX = n–1
        PUSH EAX                        ; push n – 1
        CALL _towers                    ; call towers(n – 1, 6 – i – j, i)
        ADD ESP, 12                     ; adjust stack pointer
Done:   LEAVE                           ; prepare to exit
        RET 0                           ; return to the called
.DATA
format  DB "Move disk from %d to %d\n"  ; format string
END
```

Figure 5-44. The Towers of Hanoi for the Core i7.

define a few symbols in the beginning to clean it up. To make this work, the program has to be run through a program called *cpp*, the C preprocessor, before assembling it. Also we have used lowercase letters here because the OMAP4430 ARM assembler insists on them (in case any readers wish to type the program in).

Algorithmically, the OMAP4430 ARM version is identical to the Core i7 version. Both test n to start with, branching to the else clause if $n > 1$. The main complexity of the ARM version is due to some properties of the ISA.

To start with, the OMAP4430 ARM has to pass the address of the format string to *printf*, but the machine cannot just move the address to the register that holds the outgoing parameter because there is no way to put a 32-bit constant in a register in one instruction. It takes two instructions to do this, MOVW and MOVT.

The next thing to notice is that the stack adjustments are automatically handled by the PUSH and POP instructions at the beginning and end of functions. These instructions also handle the saving and restoring of the return address, by saving the *LR* register on entry and restoring the PC on function exit.

5.8 THE IA-64 ARCHITECTURE AND THE ITANIUM 2

Around year 2000, some engineers inside Intel felt the company was getting to the point where it had just about squeezed every last drop of juice out of the IA-32 line of processors. New models were still seeing benefits from advances in manufacturing technology, which means smaller transistors (hence faster clock speeds). However, finding new tricks to speed up the implementation even more was getting harder and harder as the constraints imposed by the IA-32 ISA were looming larger all the time.

Some engineers felt that the only real solution was to abandon the IA-32 as the main line of development and go to a completely new ISA. This is, in fact, what Intel started working toward. In fact, it had plans for two new architectures. The EMT-64 is a wider version of the traditional Pentium ISA, with 64-bit registers and a 64-bit address space. This new ISA solves the address-space problem but still has all the implementation complexities of its predecessors. It can best be thought of as a wider Pentium.

The other new architecture, which was developed jointly by Intel and Hewlett Packard, was named the **IA-64**. It is a full 64-bit machine from beginning to end, not an extension of an existing 32-bit machine. Furthermore, it is a radical departure from the IA-32 architecture in many ways. The initial market was for high-end servers, but Intel had hoped it would catch on in the desktop world eventually. That did not happen. Bad as it was, the customers refused to abandon the IA-32. Nevertheless, the architecture is so radically different from anything we have studied so far that it is worth examining just for that reason. The first implementation of the IA-64 architecture is the Itanium series. In the remainder of this section we will study the IA-64 architecture and the Itanium 2 CPU that implements it.

```
#define Param0                          r0
#define Param1                          r1
#define Param2                          r2
#define FormatPtr                       r0
#define k                               r7
#define n_minus_1                       r5

        .text
towers: push {r3, r4, r5, r6, r7, lr}   @ save return addr and touched regs
        mov r4, Param1
        mov r6, Param2
        cmp Param0, #1                  @ is (n == 1)?
        bne else                        @ if not, jump to else code sequence
        movw FormatPtr, #:lower16:format @ load format string pointer
        movt FormatPtr, #:upper16:format
        bl printf                       @ print move
        pop {r3, r4, r5, r6, r7, pc}

else:   rsb k, r1, #6                   @ k = 6 – i – j
        subs k, k, r2
        add n_minus_1, r0, #-1          @ compute (n-1) for recursive call
        mov r0, n_minus_1
        mov r2, k
        bl towers                       @ call towers(n–1, i, k)
        mov r0, #1
        mov r1, r4
        mov r2, r6
        bl towers                       @ call towers(1, k, j)
        mov r0, n_minus_1
        mov r1, k
        mov r2, r6
        bl towers                       @ call towers(n–1, k, j)
        pop {r3, r4, r5, r6, r7, pc}    @ restore touched registers and return to called

        .global main
main:   push {lr}                       @ save called's return address
        mov Param0, #3
        mov Param1, #1
        mov Param2, Param0
        bl towers                       @ call towers(3, 1, 3)
        pop {pc}                        @ pop return address, return to called

format: .ascii "Move a disk from %d to %d\n\0"
```

Figure 5-45. The Towers of Hanoi for the OMAP4430 ARM CPU.

5.8.1 The Problem with the IA-32 ISA

Before getting into the details of the IA-64 and Itanium 2, it is useful to review what is wrong with the IA-32 ISA to see what problems Intel was trying to solve with the new architecture. The main fact of life that causes all the trouble is that

IA-32 is an ancient ISA with all the wrong properties for current technology. It is a CISC ISA with variable-length instructions and a myriad of different formats that are hard to decode quickly on the fly. Current technology works best with RISC ISAs that have one instruction length and a fixed-length opcode that is easy to decode. The IA-32 instructions can be broken up into RISC-like micro-operations at execution time, but doing so requires hardware (chip area), takes time, and adds complexity to the design. That is strike one.

The IA-32 is also a two-address memory-oriented ISA. Most instructions reference memory, and most programmers and compilers think nothing of referencing memory all the time. Current technology favors load/store ISAs that reference memory only to get the operands into registers but otherwise perform all their calculations using three-address memory register instructions. And with CPU clock speeds going up much faster than memory speeds, the problem will get worse with time. That is strike two.

The IA-32 also has a small and irregular register set. Not only does this tie compilers in knots, but the small number of general-purpose registers (four or six, depending on how you count ESI and EDI) requires intermediate results to be spilled into memory all the time, generating extra memory references even when they are not logically needed. That is strike three. The IA-32 is out.

Now let us start the second inning. The small number of registers causes many dependences, especially unnecessary WAR dependences, because results have to go somewhere and no extra registers are available. Getting around the lack of registers requires the implementation to do renaming internally—a terrible hack if ever there was one—to secret registers inside the reorder buffer. To avoid blocking on cache misses too often, instructions have to be executed out of order. However, the IA-32's semantics specify precise interrupts, so the out-of-order instructions have to be retired in order. All of these things require a lot of very complex hardware. Strike four.

Doing all this work quickly requires a deep pipeline. In turn, the deep pipeline means that instructions entered into it take many cycles before they are finished. Consequently, very accurate branch prediction is essential to make sure the right instructions are being entered into the pipeline. Because a misprediction requires the pipeline to be flushed, at great cost, even a fairly low misprediction rate can cause a substantial performance degradation. Strike five.

To alleviate the problems with mispredictions, the processor has to do speculative execution, with all the problems it entails, especially when memory references on the wrong path cause an exception. Strike six.

We are not going to play the whole baseball game here, but it should be clear by now that there is a problem. And we have not even mentioned that IA-32's 32-bit addresses limit individual programs to 4 GB of memory, which is a big problem on servers. The EMT-64 solves this problem but not all the others.

All in all, the situation with IA-32 can be favorably compared to the state of celestial mechanics just prior to Copernicus. The then-current theory dominating

astronomy was that the earth was fixed and motionless in space and that the planets moved in circles with epicycles around it. However, as observations got better and more deviations from this model could be clearly observed, epicycles were added to the epicycles until the whole model just collapsed from its internal complexity.

Intel is in the same pickle now. A huge fraction of all the transistors on the Core i7 are devoted to decomposing CISC instructions, figuring out what can be done in parallel, resolving conflicts, making predictions, repairing the consequences of incorrect predictions, and other bookkeeping, leaving surprisingly few for doing the real work the user asked for. The conclusion that Intel is being inexorably driven to is the only sane conclusion: junk the whole thing (IA-32) and start all over with a clean slate (IA-64). The EMT-64 provides some breathing room, but it really papers over the complexity issue.

5.8.2 The IA-64 Model: Explicitly Parallel Instruction Computing

The key idea behind the IA-64 is moving work from run time to compile time. On the Core i7, during execution the CPU reorders instructions, renames registers, schedules functional units, and does a lot of other work to determine how to keep all the hardware resources fully occupied. In the IA-64 model, the compiler figures out all these things in advance and produces a program that can be run as is, without the hardware having to juggle everything during execution. For example, rather than tell the compiler that the machine has eight registers when it actually has 128 and then try to figure out at run time how to avoid dependences, in the IA-64 model, the compiler is told how many registers the machine really has so it can produce a program that does not have any register conflicts to start with. Similarly, in this model, the compiler keeps track of which functional units are busy and does not issue instructions that use functional units that are not available. The model of making the underlying parallelism in the hardware visible to the compiler is called **EPIC** (**Explicitly Parallel Instruction Computing**). To some extent, EPIC can be thought of as the successor to RISC.

The IA-64 model has a number of features that speed up performance. These include reducing memory references, instruction scheduling, reducing conditional branches, and speculation. We will now examine each of these in turn and discuss how they are implemented in the Itanium 2.

5.8.3 Reducing Memory References

The Itanium 2 has a simple memory model. Memory consists of up to 2^{64} bytes of linear memory. Instructions are available to access memory in units of 1, 2, 4, 8, 16, and 10 bytes, the latter for 80-bit IEEE 745 floating-point numbers. Memory references need not be aligned on their natural boundaries, but a performance penalty is incurred if they are not. Memory can be either big endian or little endian, determined by a bit in a register loadable by the operating system.

Memory access is a huge bottleneck in all modern computers because CPUs are so much faster than memory. One way to reduce memory references is to have a large level 1 cache on chip and an even larger level 2 cache close to the chip. All modern designs have these two caches. But one can go beyond caching to look for other ways to reduce memory references, and the IA-64 uses some of these ways.

The best way to speed up memory references is to avoid having them in the first place. The Itanium 2 implementation of the IA-64 model has 128 general-purpose 64-bit registers. The first 32 of these are static, but the remaining 96 are used as a register stack, very similar to the register window scheme in other RISC processors, such as the UltraSPARC. However, unlike the UltraSPARC, the number of registers visible to the program is variable and can change from procedure to procedure. Thus each procedure has access to 32 static registers and some (variable) number of dynamically allocated registers.

When a procedure is called, the register stack pointer is advanced so the input parameters are visible in registers, but no registers are allocated for local variables. The procedure itself decides how many registers it needs and advances the register stack pointer to allocate them. These registers need not be saved on entry or restored on exit, although if the procedure needs to modify a static register it must take care to explicitly save it first and restore it later. By making the number of registers available variable and tailored to what each procedure needs, scarce registers are not wasted and procedure calls can go deeper before registers have to be spilled to memory.

The Itanium 2 also has 128 floating-point registers in IEEE 745 format. They do not operate as a register stack. This very large number of registers means that many floating-point computations can keep all their intermediate results in registers and avoid having to store temporary results in memory.

There are also 64 1-bit predicate registers, eight branch registers, and 128 special-purpose application registers used for various purposes, such as passing parameters between application programs and the operating system. An overview of the Itanium 2's registers is given in Fig. 5-46.

5.8.4 Instruction Scheduling

One of the main problems in the Core i7 is the difficulty of scheduling the various instructions over the various functional units and avoiding dependences. Exceedingly complex mechanisms are needed to handle these issues at run time, and a large fraction of the chip area is devoted to managing them. The IA-64 and Itanium 2 avoid all these problems by having the compiler do the work. The key idea is that a program consists of a sequence of **instruction groups**. Within certain boundaries, all the instructions within a group do not conflict with one another, do not use more functional units and resources than the machine has, do not contain RAW and WAW dependences, and have only certain restricted WAR dependences. Consecutive instruction groups give the appearance of being executed

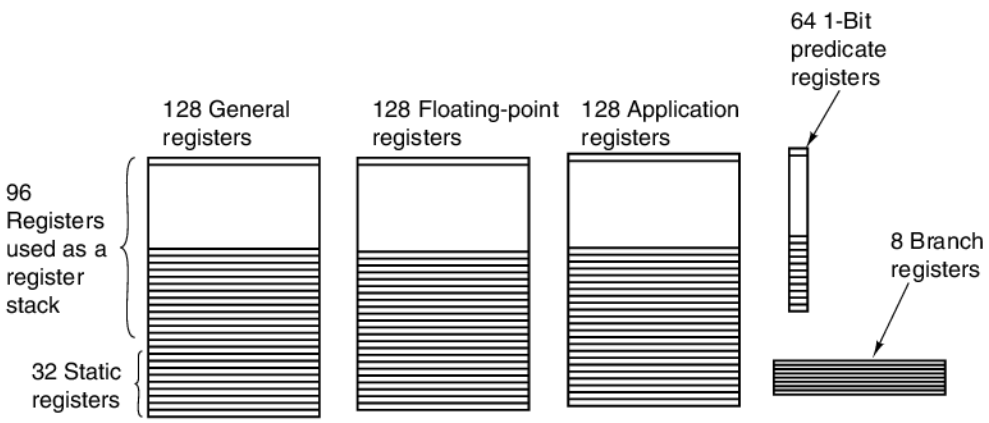

Figure 5-46. The Itanium 2's registers.

strictly sequentially with the second group not starting until the first one has completed. The CPU may, however, start the second group (in part) as soon as it feels it is safe to.

As a consequence of these rules, the CPU is free to schedule the instructions within a group in any order it chooses, possibly in parallel if it can, without having to worry about conflicts. If the instruction group violates the rules, program behavior is undefined. It is up to the compiler to reorder the assembly code generated from the source program to meet all these requirements. For rapid compilation while a program is being debugged, the compiler can put every instruction in a different group, which is easy to do but gives poor performance. When it is time to produce production code, the compiler can spend a long time optimizing it.

Instructions are organized into 128-bit **bundles** as shown at the top of Fig. 5-47. Each bundle contains three 41-bit instructions and a 5-bit template. An instruction group need not be an integral number of bundles; it can start and end in the middle of a bundle.

Over 100 instruction formats exist. A typical one, in this case for ALU operations such as ADD, which sums two registers into a third one, is shown in Fig. 5-47. The first field, the OPERATION GROUP, is the major group and mostly tells the broad class of the instruction, such as an integer ALU operation. The next field, the OPERATION TYPE, gives the specific operation required, such as ADD or SUB. Then come the three register fields. Finally, we have the PREDICATE REGISTER, to be described shortly.

The bundle template essentially tells which functional units the bundle needs and also the position of an instruction-group boundary present, if any. The major functional units are the integer ALU, non-ALU integer instructions, memory operations, floating-point operations, branching, and other. Of course, with six units and three instructions, complete orthogonality would require 216 combinations,