The execution of an arithmetic instruction may involve data transfer operations to position operands for input to the ALU, and to deliver the output of the ALU. Figure 3.5 illustrates the movements involved in both data transfer and arithmetic operations. In addition, of course, the ALU portion of the processor performs the desired operation.

Logical

Most machines also provide a variety of operations for manipulating individual bits of a word or other addressable units, often referred to as "bit twiddling." They are based upon Boolean operations (see Appendix B).

Some of the basic logical operations that can be performed on Boolean or binary data are shown in Table 10.6. The NOT operation inverts a bit. AND, OR, and Exclusive-OR (XOR) are the most common logical functions with two operands. EQUAL is a useful binary test.

These logical operations can be applied bitwise to *n*-bit logical data units. Thus, if two registers contain the data

$$(R1) = 10100101$$
$$(R2) = 00001111$$

then

$$(R1)\ \text{AND}\ (R2) = 00000101$$

where the notation (X) means the contents of location X. Thus, the AND operation can be used as a *mask* that selects certain bits in a word and zeros out the remaining bits. As another example, if two registers contain

$$(R1) = 10100101$$
$$(R2) = 11111111$$

then

$$(R1)\ \text{XOR}\ (R2) = 01011010$$

With one word set to all 1s, the XOR operation inverts all of the bits in the other word (ones complement).

In addition to bitwise logical operations, most machines provide a variety of shifting and rotating functions. The most basic operations are illustrated in Figure 10.5. With a **logical shift**, the bits of a word are shifted left or right. On one end, the bit shifted out

Table 10.6 Basic Logical Operations

P	Q	NOT P	P AND Q	P OR Q	P XOR Q	P=Q
0	0	1	0	0	0	1
0	1	1	0	1	1	0
1	0	0	0	1	1	0
1	1	0	1	1	0	1

code in the sequence of the binary representation of the EBCDIC code; that is, the IRA code is placed in the table at the relative location equal to the binary value of the EBCDIC code of the same character. Thus, locations 10F0 through 10F9 will contain the values 30 through 39, because F0 is the EBCDIC code for the digit 0, and 30 is the IRA code for the digit 0, and so on through digit 9. Now suppose we have the EBCDIC for the digits 1984 starting at location 2100 and we wish to translate to IRA. Assume the following:

- Locations 2100–2103 contain F1 F9 F8 F4.
- R1 contains 2100.
- R2 contains 1000.

Then, if we execute

TR R1, R2, 4

locations 2100–2103 will contain 31 39 38 34.

Input/Output

Input/output instructions were discussed in some detail in Chapter 7. As we saw, there are a variety of approaches taken, including isolated programmed I/O, memory-mapped programmed I/O, DMA, and the use of an I/O processor. Many implementations provide only a few I/O instructions, with the specific actions specified by parameters, codes, or command words.

System Control

System control instructions are those that can be executed only while the processor is in a certain privileged state or is executing a program in a special privileged area of memory. Typically, these instructions are reserved for the use of the operating system.

Some examples of system control operations are as follows. A system control instruction may read or alter a control register; we discuss control registers in Chapter 12. Another example is an instruction to read or modify a storage protection key, such as is used in the S/390 memory system. Another example is access to process control blocks in a multiprogramming system.

Transfer of Control

For all of the operation types discussed so far, the next instruction to be performed is the one that immediately follows, in memory, the current instruction. However, a significant fraction of the instructions in any program have as their function changing the sequence of instruction execution. For these instructions, the operation performed by the processor is to update the program counter to contain the address of some instruction in memory.

There are a number of reasons why transfer-of-control operations are required. Among the most important are the following:

1. In the practical use of computers, it is essential to be able to execute each instruction more than once and perhaps many thousands of times. It may require thousands or perhaps millions of instructions to implement an application. This

would be unthinkable if each instruction had to be written out separately. If a table or a list of items is to be processed, a program loop is needed. One sequence of instructions is executed repeatedly to process all the data.

2. Virtually all programs involve some decision making. We would like the computer to do one thing if one condition holds, and another thing if another condition holds. For example, a sequence of instructions computes the square root of a number. At the start of the sequence, the sign of the number is tested. If the number is negative, the computation is not performed, but an error condition is reported.
3. To compose correctly a large or even medium-size computer program is an exceedingly difficult task. It helps if there are mechanisms for breaking the task up into smaller pieces that can be worked on one at a time.

We now turn to a discussion of the most common transfer-of-control operations found in instruction sets: branch, skip, and procedure call.

Branch Instructions A branch instruction, also called a jump instruction, has as one of its operands the address of the next instruction to be executed. Most often, the instruction is a **conditional branch** instruction. That is, the branch is made (update program counter to equal address specified in operand) only if a certain condition is met. Otherwise, the next instruction in sequence is executed (increment program counter as usual). A branch instruction in which the branch is always taken is an **unconditional branch.**

There are two common ways of generating the condition to be tested in a conditional branch instruction. First, most machines provide a 1-bit or multiple-bit condition code that is set as the result of some operations. This code can be thought of as a short user-visible register. As an example, an arithmetic operation (ADD, SUBTRACT, and so on) could set a 2-bit condition code with one of the following four values: 0, positive, negative, overflow. On such a machine, there could be four different conditional branch instructions:

BRP X	Branch to location X if result is positive.
BRN X	Branch to location X if result is negative.
BRZ X	Branch to location X if result is zero.
BRO X	Branch to location X if overflow occurs.

In all of these cases, the result referred to is the result of the most recent operation that set the condition code.

Another approach that can be used with a three-address instruction format is to perform a comparison and specify a branch in the same instruction. For example,

BRE R1, R2, X	Branch to X if contents of R1 = contents of R2.

Figure 10.6 shows examples of these operations. Note that a branch can be either *forward* (an instruction with a higher address) or *backward* (lower address). The example shows how an unconditional and a conditional branch can be used to create a repeating loop of instructions. The instructions in locations 202 through 210 will be executed repeatedly until the result of subtracting Y from X is 0.

Because we would like to be able to call a procedure from a variety of points, the processor must somehow save the return address so that the return can take place appropriately. There are three common places for storing the return address:

- Register
- Start of called procedure
- Top of stack

Consider a machine-language instruction CALL X, which stands for *call procedure at location X.* If the register approach is used, CALL X causes the following actions:

$$RN \longleftarrow PC + \Delta$$
$$PC \longleftarrow X$$

where RN is a register that is always used for this purpose, PC is the program counter, and Δ is the instruction length. The called procedure can now save the contents of RN to be used for the later return.

A second possibility is to store the return address at the start of the procedure. In this case, CALL X causes

$$X \longleftarrow PC + \Delta$$
$$PC \longleftarrow X + 1$$

This is quite handy. The return address has been stored safely away.

Both of the preceding approaches work and have been used. The only limitation of these approaches is that they prevent the use of *reentrant* procedures. A reentrant procedure is one in which it is possible to have several calls open to it at the same time. A recursive procedure (one that calls itself) is an example of the use of this feature.

A more general and powerful approach is to use a stack (see Appendix 10A for a discussion of stacks). When the processor executes a call, it places the return address on the stack. When it executes a return, it uses the address on the stack. Figure 10.8 illustrates the use of the stack.

In addition to providing a return address, it is also often necessary to pass parameters with a procedure call. These can be passed in registers. Another possibility is to store the parameters in memory just after the CALL instruction. In this case, the return must be to the location following the parameters. Again, both of these approaches have drawbacks. If registers are used, the called program and the calling program must be written to assure that the registers are used properly. The storing of

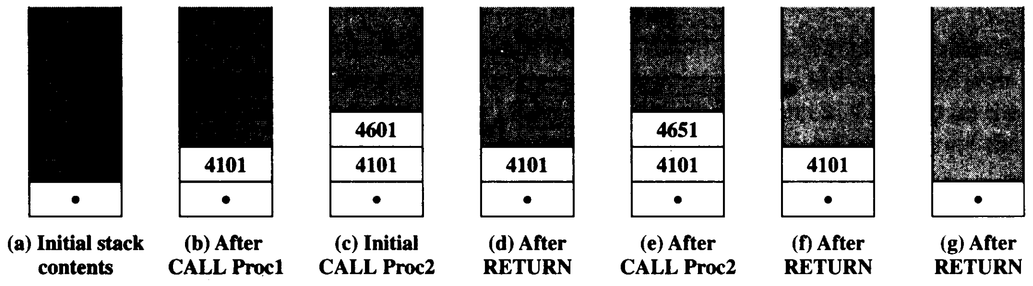


Figure 10.8 Use of Stock to Implement Nested Subroutines of Figure 10.7

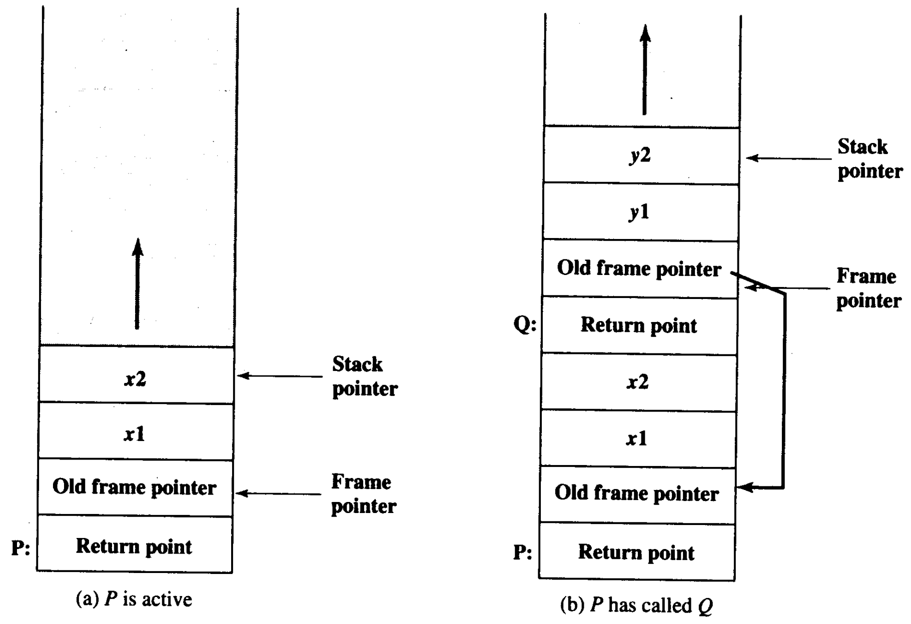


Figure 10.9 Stock Frame Growth Using Sample Procedures P and Q

parameters in memory makes it difficult to exchange a variable number of parameters. Both approaches prevent the use of reentrant procedures.

A more flexible approach to parameter passing is the stack. When the processor executes a call, it not only stacks the return address, it stacks parameters to be passed to the called procedure. The called procedure can access the parameters from the stack. Upon return, return parameters can also be placed on the stack. The entire set of parameters, including return address, that is stored for a procedure invocation is referred to as a *stack frame*.

An example is provided in Figure 10.9. The example refers to procedure P in which the local variables *x*1 and *x*2 are declared, and procedure Q, which P can call and in which the local variables *y*1 and *y*2 are declared. In this figure, the return point for each procedure is the first item stored in the corresponding stack frame. Next is stored a pointer to the beginning of the previous frame. This is needed if the number or length of parameters to be stacked is variable.

10.5 PENTIUM AND POWERPC OPERATION TYPES

Pentium Operation Types

The Pentium provides a complex array of operation types, including a number of specialized instructions. The intent was to provide tools for the compiler writer to produce optimized machine language translation of high-level language programs. Table 10.8 lists the types and gives examples of each. Most of these are the conventional instructions found in most machine instruction sets, but several types of instructions are tailored to the 80x86/Pentium architecture and are of particular interest.

Call/Return Instructions

The Pentium provides four instructions to support procedure call/return: CALL, ENTER, LEAVE, RETURN. It will be instructive to look at the support provided by these instructions. Recall from Figure 10.9 that a common means of implementing the procedure call/return mechanism is via the use of stack frames. When a new procedure is called, the following must be performed upon entry to the new procedure:

- Push the return point on the stack.
- Push the current frame pointer on the stack.
- Copy the stack pointer as the new value of the frame pointer.
- Adjust the stack pointer to allocate a frame.

The CALL instruction pushes the current instruction pointer value onto the stack and causes a jump to the entry point of the procedure by placing the address of the entry point in the instruction pointer. In the 8088 and 8086 machines, the typical procedure began with the sequence

```
PUSH    EBP
MOV     EBP, ESP
SUB     ESP, space_for_locals
```

where EBP is the frame pointer and ESP is the stack pointer. In the 80286 and later machines, the ENTER instruction performs all the aforementioned operations in a single instruction.

The ENTER instruction was added to the instruction set to provide direct support for the compiler. The instruction also includes a feature for support of what are called nested procedures in languages such as Pascal, COBOL, and Ada (not found in C or FORTRAN). It turns out that there are better ways of handling nested procedure calls for these languages. Furthermore, although the ENTER instruction saves a few bytes of memory compared with the PUSH, MOV, SUB sequence (4 bytes versus 6 bytes), it actually takes longer to execute (10 clock cycles versus 6 clock cycles). Thus, although it may have seemed a good idea to the instruction set designers to add this feature, it complicates the implementation of the processor while providing little or no benefit. We will see that, in contrast, a RISC approach to processor design would avoid complex instructions such as ENTER and might produce a more efficient implementation with a sequence of simpler instructions.

Memory Management Another set of specialized instructions deals with memory segmentation. These are privileged instructions that can only be executed from the operating system. They allow local and global segment tables (called descriptor tables) to be loaded and read, and for the privilege level of a segment to be checked and altered.

The special instructions for dealing with the on-chip cache were discussed in Chapter 4.

Condition Codes We have mentioned that condition codes are bits in special registers that may be set by certain operations and used in conditional branch instructions. These conditions are set by arithmetic and compare operations. The compare

Table 10.9 Pentium Condition Codes

Status Bit	Name	Description
C	Carry	Indicates carrying or borrowing into the left-most bit position following an arithmetic operation. Also modified by some of the shift and rotate operations.
P	Parity	Parity of the result of an arithmetic or logic operation. 1 indicates even parity; 0 indicates odd parity.
A	Auxiliary Carry	Represents carrying or borrowing between half-bytes of an 8-bit arithmetic or logic operation using the AL register.
Z	Zero	Indicates that the result of an arithmetic or logic operation is 0.
S	Sign	Indicates the sign of the result of an arithmetic or logic operation.
O	Overflow	Indicates an arithmetic overflow after an addition or subtraction.

operation in most languages subtracts two operands, as does a subtract operation. The difference is that a compare operation only sets condition codes, whereas a subtract operation also stores the result of the subtraction in the destination operand.

Table 10.9 lists the condition codes used on the Pentium. Each condition, or combinations of these conditions, can be tested for a conditional jump. Table 10.10

Table 10.10 Pentium Conditions for Conditional Jump and SETcc Instructions

Symbol	Condition Tested	Comment
A, NBE	C=0 AND Z=0	Above; Not below or equal (greater than, unsigned)
AE, NB, NC	C=0	Above or equal; Not below (greater than or equal, unsigned); Not carry
B, NAE, C	C=1	Below; Not above or equal (less than, unsigned); Carry set
BE, NA	C=1 OR Z=1	Below or equal; Not above (less than or equal, unsigned)
E, Z	Z=1	Equal; Zero (signed or unsigned)
G, NLE	[(S=1 AND O=1) OR (S=0 and O=0)] AND [Z=0]	Greater than; Not less than or equal (signed)
GE, NL	(S=1 AND O=1) OR (S=0 AND O=0)	Greater than or equal; Not less than (signed)
L, NGE	(S=1 AND O=0) OR (S=0 AND O=1)	Less than; Not greater than or equal (signed)
LE, NG	(S=1 AND O=0) OR (S=0 AND O=1) OR (Z=1)	Less than or equal; Not greater than (signed)
NE, NZ	Z=0	Not equal; Not zero (signed or unsigned)
NO	O=0	No overflow
NS	S=0	Not sign (not negative)
NP, PO	P=0	Not parity; Parity odd
O	O=1	Overflow
P	P=1	Parity; Parity even
S	S=1	Sign (negative)

shows the combinations of conditions for which conditional jump opcodes have been defined.

Several interesting observations can be made about this list. First, we may wish to test two operands to determine if one number is bigger than another. But this will depend on whether the numbers are signed or unsigned. For example, the 8-bit number 11111111 is bigger than 00000000 if the two numbers are interpreted as unsigned integers ($255 > 0$) but is less if they are considered as 8-bit twos complement numbers ($-1 < 0$). Many assembly languages therefore introduce two sets of terms to distinguish the two cases: If we are comparing two numbers as signed integers, we use the terms *less than* and *greater than;* if we are comparing them as unsigned integers, we use the terms *below* and *above.*

A second observation concerns the complexity of comparing signed integers. A signed result is greater than or equal to zero if (1) the sign bit is zero and there is no overflow ($S = 0$ AND $O = 0$), or (2) the sign bit is one and there is an overflow. A study of Figure 9.4 should convince you that the conditions tested for the various signed operations are appropriate (see Problem 10.13).

Pentium MMX Instructions In 1996, Intel introduced MMX technology into its Pentium product line. MMX is set of highly optimized instructions for multimedia tasks. There are 57 new instructions that treat data in a SIMD (single-instruction, multiple-data) fashion, which makes it possible to perform the same operation, such as addition or multiplication, on multiple data elements at once. Each instruction typically takes a single clock cycle to execute. For the proper application, these fast parallel operations can yield a speedup of two to eight times over comparable algorithms that do not use the MMX instructions [ATKI96].

The focus of MMX is multimedia programming. Video and audio data are typically composed of large arrays of small data types, such as 8 or 16 bits, whereas conventional instructions are tailored to operate on 32- or 64-bit data. Here are some examples: In graphics and video, a single scene consists of an array of pixels,[2] and there are 8 bits for each pixel or 8 bits for each pixel color component (red, green, blue). Typical audio samples are quantized using 16 bits. For some 3D graphics algorithms, 32 bits are common for basic data types. To provide for parallel operation on these data lengths, three new data types are defined in MMX. Each data type is 64 bits in length and consists of multiple smaller data fields, each of which holds a fixed-point integer. The types are as follows:

- **Packet byte:** Eight bytes packed into one 64-bit quantity
- **Packed word:** Four 16-bit words packed into 64 bits
- **Packed doubleword:** Two 32-bit doublewords packed into 64 bits

Table 10.11 lists the MMX instruction set. Most of the instructions involve parallel operation on bytes, words, or doublewords. For example, the PSLLW

[2]A pixel, or picture element, is the smallest element of a digital image that can be assigned a gray level. Equivalently, a pixel is an individual dot in a dot-matrix representation of a picture.

Table 10.11 MMX Instruction Set

Category	Instruction	Description
Arithmetic	PADD [B, W, D]	Parallel add of packed eight bytes, four 16-bit words, or two 32-bit doublewords, with wraparound.
	PADDS [B, W]	Add with saturation.
	PADDUS [B, W]	Add unsigned with saturation.
	PSUB [B, W, D]	Subtract with wraparound.
	PSUBS [B, W]	Subtract with saturation.
	PSUBUS [B, W]	Subtract unsigned with saturation.
	PMULHW	Parallel multiply of four signed 16-bit words, with high-order 16 bits of 32-bit result chosen.
	PMULLW	Parallel multiply of four signed 16-bit words, with low-order 16 bits of 32-bit result chosen.
	PMADDWD	Parallel multiply of four signed 16-bit words; add together adjacent pairs of 32-bit results.
Comparison	PCMPEQ [B, W, D]	Parallel compare for equality; result is mask of 1s if true or 0s if false.
	PCMPGT [B, W, D]	Parallel compare for greater than; result is mask of 1s if true or 0s if false.
Conversion	PACKUSWB	Pack words into bytes with unsigned saturation.
	PACKSS [WB, DW]	Pack words into bytes, or doublewords into words, with signed saturation.
	PUNPCKH [BW, WD, DQ]	Parallel unpack (interleaved merge) high-order bytes, words, or doublewords from MMX register.
	PUNPCKL [BW, WD, DQ]	Parallel unpack (interleaved merge) low-order bytes, words, or doublewords from MMX register.
Logical	PAND	64-bit bitwise logical AND
	PNDN	64-bit bitwise logical AND NOT
	POR	64-bit bitwise logical OR
	PXOR	64-bit bitwise logical XOR
Shift	PSLL [W, D, Q]	Parallel logical left shift of packed words, doublewords, or quadword by amount specified in MMX register or immediate value.
	PSRL [W, D, Q]	Parallel logical right shift of packed words, doublewords, or quadword.
	PSRA [W, D]	Parallel arithmetic right shift of packed words, doublewords, or quadword.
Data Transfer	MOV [D, Q]	Move doubleword or quadword to/from MMX register.
State Mgt	EMMS	Empty MMX state (empty FP registers tag bits).

Note: If an instruction supports multiple data types [byte (B), word (W), doubleword (D), quadword (Q)], the data types are indicated in brackets.

instruction performs a left logical shift separately on each of the four words in the packed word operand; the PADDB instruction takes packed byte operands as input and performs parallel additions on each byte position independently to produce a packed byte output.

One unusual feature of the new instruction set is the introduction of **saturation arithmetic**. With ordinary unsigned arithmetic, when an operation overflows (i.e., a carry out of the most significant bit), the extra bit is truncated. This is referred to as wraparound, because the effect of the truncation can be, for example, to produce an addition result that is smaller than the two input operands. Consider the addition of the two words, in hexadecimal, F000h and 3000h. The sum would be expressed as

$$\begin{array}{rl} \text{F000h} = & 1111\ 0000\ 0000\ 0000 \\ +3000h = & \underline{0011\ 0000\ 0000\ 0000} \\ & 10010\ 0000\ 0000\ 0000 = 2000\text{h} \end{array}$$

If the two numbers represented image intensity, then the result of the addition is to make the combination of two dark shades turn out to be lighter. This is typically not what is intended. With saturation arithmetic, if addition results in overflow or subtraction results in underflow, the result is set to the largest or smallest value representable. For the preceding example, with saturation arithmetic, we have

$$\begin{array}{rl} \text{F000h} = & 1111\ 0000\ 0000\ 0000 \\ +3000h = & \underline{0011\ 0000\ 0000\ 0000} \\ & 10010\ 0000\ 0000\ 0000 \\ & \quad 1111\ 1111\ 1111\ 1111 = \text{FFFFh} \end{array}$$

To provide a feel for the use of MMX instructions, we look at an example, taken from [PELE97]. A common video application is the fade-out, fade-in effect, in which one scene gradually dissolves into another. Two images are combined with a weighted average:

$$\text{Result\_pixel} = \text{A\_pixel} \times \text{fade} + \text{B\_pixel} \times (1 - \text{fade})$$

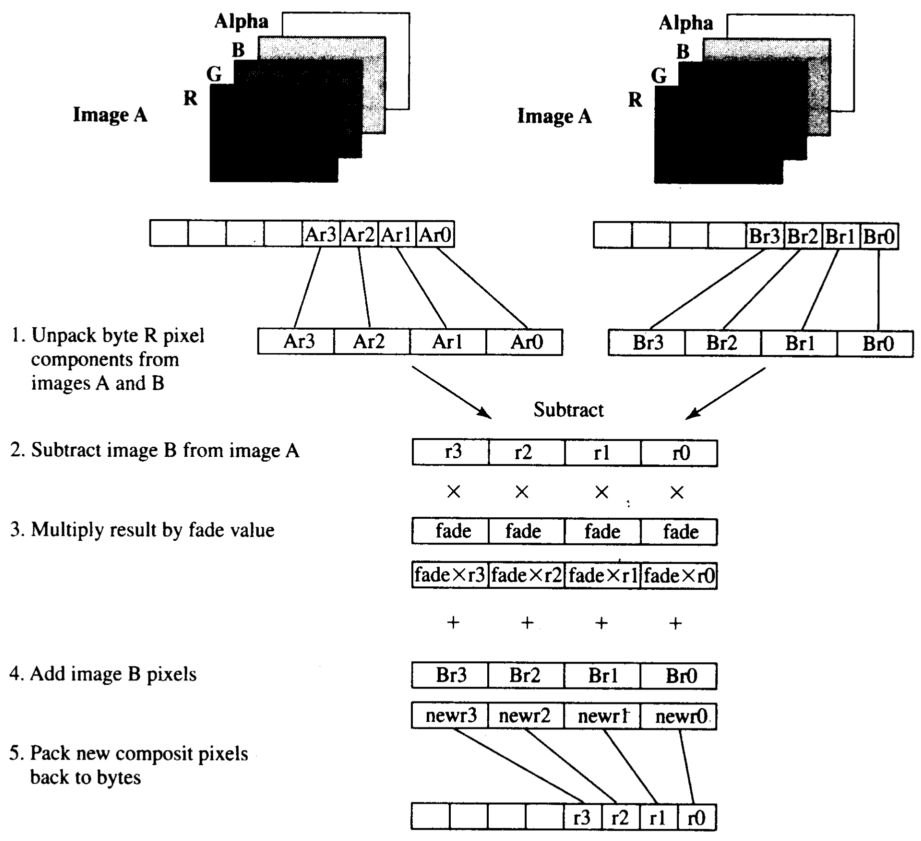
This calculation is performed on each pixel position in A and B. If a series of video frames is produced while gradually changing the fade value from 1 to 0 (scaled appropriately for an 8-bit integer), the result is to fade from image A to image B.

Figure 10.10 shows the sequence of steps required for one set of pixels. The 8-bit pixel components are converted to 16-bit elements to accommodate the MMX 16-bit multiply capability. If these images use 640 × 480 resolution, and the dissolve technique uses all 255 possible values of the fade value, then the total number of instructions executed using MMX is 535 million. The same calculation, performed without the MMX instructions, requires 1.4 billion instructions [INTE98].

PowerPC Operation Types

The PowerPC provides a large collection of operation types. Table 10.12 lists the types and gives examples of each. Several features are worth noting.

Branch-Oriented Instructions The PowerPC supports the usual unconditional and conditional branch capabilities. Conditional branch instructions test a single bit of the condition register for true, false, or don't care and the contents of the count register for zero, nonzero, or don't care. Thus, there are nine separate conditions that can be



MMX code sequence performing this operation:

```
pxor        mm7, mm7       ;zero out mm7
movq        mm3, fad_val   ;load fade value replicated 4 times
movd        mm0, imageA    ;load 4 red pixel components from image A
movd        mm1, imageB    ;load 4 red pixel components from image B
punpckblw   mm0, mm7       ;unpack 4 pixels to 16 bits
punpckblw   mm1, mm7       ;unpack 4 pixels to 16 bits
psubw       mm0, mm1       ;subtract image B from image A
pmulhw      mm0, mm3       ;multiply the subtract result by fade values
padddw      mm0, mm1       ;add result to image B
packuswb    mm0, mm7       ;pack 16-bit results back to bytes
```

Figure 10.10 Image Compositing on Color Plane Representation [PELE97]

defined for the conditional branch instruction. If the count register is tested for zero or nonzero, then it is decremented by 1 prior to the test. This is convenient for setting up iteration loops.

Branch instructions can also indicate that the address of the location following the branch is to be placed in the link register, described in Chapter 14. This facilitates call/return processing.

Load/Store Instructions In the PowerPC architecture, only load and store instructions access memory locations; arithmetic and logical instructions are performed

Table 10.12 PowerPC Operation Types (with Examples of Typical Operations)

Instruction	Description
	Branch-Oriented
b	Unconditional branch
bl	Branch to target address and place effective address of instruction following the branch into the Link Register.
bc	Branch conditional on Count Register and/or on bit in Condition Register.
sc	System call to invoke an operating system service
trap	Compare two operands and invoke system trap handler if specified conditions are met.
	Load/Store
lwzu	Load word and zero extend to left; update source register.
ld	Load doubleword.
lmw	Load multiple word; load consecutive words into contiguous registers from the target register through general-purpose register 31.
lswx	Load a string of bytes into registers beginning with target register; 4 bytes per register; wrap around from register 31 to register 0.
	Integer Arithmetic
add	Add contents of two registers and place in third register.
subf	Subtract contents of two registers and place in third register.
mullw	Multiply low-order 32-bit contents of two registers and place 64-bit product in third register.
divd	Divide 64-bit contents of two registers and place in quotient in third register.
	Logical and Shift
cmp	Compare two operands and set four condition bits in the specified condition register field.
crand	Condition register AND: two bits of the Condition Register are ANDed and the result placed in one of the two bit positions.
and	AND contents of two registers and place in third register.
cntlzd	Count number of consecutive 0 bits starting at bit zero in source register and place count in destination register.
rldic	Rotate left doubleword register, AND with mask, and store in destination register.
sld	Shift left bits in source register and store in destination register.
	Floating-Point
lfs	Load 32-bit floating-point number from memory, convert to 64-bit format, and store in floating-point register.
fadd	Add contents of two registers and place in third register.
fmadd	Multiply contents of two registers, add the contents of a third, and place result in fourth register.
fcmpu	Compare two floating-point operands and set condition bits.
	Cache Management
dcbf	Data cache block flush; perform lookup in cache on specified target address and perform flushing operation.
icbi	Instruction cache block invalidate

only on registers. This is characteristic of RISC design, and it is explored further in Chapter 13.

There are two features that characterize the different load/store instructions:

- **Data size:** Data can be transferred in units of byte, halfword, word, or doubleword. Instructions are also available for loading or storing a string of bytes into or from multiple registers.
- **Sign extension:** For halfword and word loads, the unused bits to the left in the 64-bit destination register are either filled with zeros or with the sign bit of the loaded quantity.

10.6 ASSEMBLY LANGUAGE

A processor can understand and execute machine instructions. Such instructions are simply binary numbers stored in the computer. If a programmer wished to program directly in machine language, then it would be necessary to enter the program as binary data.

Consider the simple BASIC statement

$$N = I + J + K$$

Suppose we wished to program this statement in machine language and to initialize I, J, and K to 2, 3, and 4, respectively. This is shown in Figure 10.11a. The program starts

Address	Contents			
101	0010	0010	101	2201
102	0001	0010	102	1202
103	0001	0010	103	1203
104	0011	0010	104	3204
201	0000	0000	201	0002
202	0000	0000	202	0003
203	0000	0000	203	0004
204	0000	0000	204	0000

(a) Binary program

Address	Contents
101	2201
102	1202
103	1203
104	3204
201	0002
202	0003
203	0004
204	0000

(b) Hexadecimal program

Address	Instruction	
101	LDA	201
102	ADD	202
103	ADD	203
104	STA	204
201	DAT	2
202	DAT	3
203	DAT	4
204	DAT	0

(c) Symbolic program

Label	Operation	Operand
FORMUL	LDA	I
	ADD	J
	ADD	K
	STA	N
I	DATA	2
J	DATA	3
K	DATA	4
N	DATA	0

(d) Assembly program

Figure 10.11 Computation of the Formula N = I + J + K

in location 101 (hexadecimal). Memory is reserved for the four variables starting at location 201. The program consists of four instructions:

1. Load the contents of location 201 into the AC.
2. Add the contents of location 202 to the AC.
3. Add the contents of location 203 to the AC.
4. Store the contents of the AC in location 204.

This is clearly a tedious and very error-prone process.

A slight improvement is to write the program in hexadecimal rather than binary notation (Figure 10.11b). We could write the program as a series of lines. Each line contains the address of a memory location and the hexadecimal code of the binary value to be stored in that location. Then we need a program that will accept this input, translate each line into a binary number, and store it in the specified location.

For more improvement, we can make use of the symbolic name or mnemonic of each instruction. This results in the *symbolic program* shown in Figure 10.11c. Each line of input still represents one memory location. Each line consists of three fields, separated by spaces. The first field contains the address of a location. For an instruction, the second field contains the three-letter symbol for the opcode. If it is a memory-referencing instruction, then a third field contains the address. To store arbitrary data in a location, we invent a *pseudoinstruction* with the symbol DAT. This is merely an indication that the third field on the line contains a hexadecimal number to be stored in the location specified in the first field.

For this type of input we need a slightly more complex program. The program accepts each line of input, generates a binary number based on the second and third (if present) fields, and stores it in the location specified by the first field.

The use of a symbolic program makes life much easier but is still awkward. In particular, we must give an absolute address for each word. This means that the program and data can be loaded into only one place in memory, and we must know that place ahead of time. Worse, suppose we wish to change the program some day by adding or deleting a line. This will change the addresses of all subsequent words.

A much better system, and one commonly used, is to use symbolic addresses. This is illustrated in Figure 10.11d. Each line still consists of three fields. The first field is still for the address, but a symbol is used instead of an absolute numerical address. Some lines have no address, implying that the address of that line is one more than the address of the previous line. For memory-reference instructions, the third field also contains a symbolic address.

With this last refinement, we have an *assembly language*. Programs written in assembly language (assembly programs) are translated into machine language by an *assembler*. This program must not only do the symbolic translation discussed earlier but also assign some form of memory addresses to symbolic addresses.

The development of assembly language was a major milestone in the evolution of computer technology. It was the first step to the high-level languages in use today. Although few programmers use assembly language, virtually all machines provide one. They are used, if at all, for systems programs such as compilers and I/O routines.

10.7 RECOMMENDED READING

The Pentium instruction set is well covered by [BREY03]. The PowerPC instruction set is covered in [IBM94] and [WEIS94].

BREY03 Brey, B. *The Intel Microprocessors: 8086/8066, 80186/80188, 80286, 80386, 80486, Pentium, Pentium Pro Processor, Pentium II, Pentium III, Pentium 4.* Upper Saddle River, NJ: Prentice Hall, 2000.

IBM94 International Business Machines, Inc. *The PowerPC Architecture: A Specification for a New Family of RISC Processors.* San Francisco, CA: Morgan Kaufmann, 1994.

WEIS94 Weiss, S., and Smith, J. *POWER and PowerPC.* San Francisco: Morgan Kaufmann, 1994.

10.8 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

accumulator	jump	procedure call
address	little endian	procedure return
arithmetic shift	logical shift	push
bi-endian	machine instruction	reentrant procedure
big endian	operand	reverse Polish notation
branch	operation	rotate
conditional branch	packed decimal	skip
instruction set	pop	stack

Review Questions

10.1 What are the typical elements of a machine instruction?

10.2 What types of locations can hold source and destination operands?

10.3 If an instruction contains four addresses, what might be the purpose of each address?

10.4 List and briefly explain five important instruction set design issues.

10.5 What types of operands are typical in machine instruction sets?

10.6 What is the relationship between the IRA character code and the packed decimal representation?

10.7 What is the difference between an arithmetic shift and a logical shift?

10.8 Why are transfer of control instructions needed?

10.9 List and briefly explain two common ways of generating the condition to be tested in a conditional branch instruction.

10.10 What is meant by the term *nesting of procedures*?

10.11 List three possible places for storing the return address for a procedure return.

10.12 What is a reentrant procedure?

10.13 What is the difference between assembly language and machine language?

10.14 What is reverse Polish notation?

10.15 What is the difference between big endian and little endian?

Problems

10.1 Show in hex notation:
a. The packed decimal format for 23
b. The ASCII characters 23

10.2 For each of the following packed decimal numbers, show the decimal value:
a. 0111 0011 0000 1001 **b.** 0101 1000 0010 **c.** 0100 1010 0110

10.3 A given microprocessor has words of one byte. What is the smallest and largest integer that can be represented in the following representations?
a. Unsigned
b. Sign-magnitude
c. Ones complement
d. Twos complement
e. Unsigned packed decimal
f. Signed packed decimal

10.4 Many processors provide logic for performing arithmetic on packed decimal numbers. Although the rules for decimal arithmetic are similar to those for binary operations, the decimal results may require some corrections to the individual digits if binary logic is used.

Consider the decimal addition of two unsigned numbers. If each number consists of N digits, then there are $4N$ bits in each number. The two numbers are to be added using a binary adder. Suggest a simple rule for correcting the result. Perform addition in this fashion on the numbers 1698 and 1786.

10.5 The tens complement of the decimal number X is defined to be $10^N - X$, where N is the number of decimal digits in the number. Describe the use of tens complement representation to perform decimal subtraction. Illustrate the procedure by subtracting $(0326)_{10}$ from $(0736)_{10}$.

10.6 Compare zero-, one-, two-, and three-address machines by writing programs to compute

$$X = (A + B \times C)/(D - E \times F)$$

for each of the four machines. The instructions available for use are as follows:

0 Address	1 Address	2 Address	3 Address
PUSH M	LOAD M	MOVE (X ← Y)	MOVE (X ← Y)
POP M	STORE M	ADD (X ← X + Y)	ADD (X ← Y + Z)
ADD	ADD M	SUB (X ← X − Y)	SUB (X ← Y − Z)
SUB	SUB M	MUL (X ← X × Y)	MUL (X ← Y × Z)
MUL	MUL M	DIV (X ← X/Y)	DIV (X ← Y/Z)
DIV	DIV M		

10.7 Consider a hypothetical computer with an instruction set of only two n-bit instructions. The first bit specifies the opcode, and the remaining bits specify one of the $2^n - 1$ n-bit words of main memory. The two instructions are

SUBS X	Subtract the contents of location X from the accumulator, and store the result in location X and the accumulator.
JUMP X	Place address X in the program counter.

A word in main memory may contain either an instruction or a binary number in twos complement notation. Demonstrate that this instruction repertoire is reasonably complete by specifying how the following operations can be programmed:
a. Data transfer: Location X to accumulator, accumulator to location X
b. Addition: Add contents of location X to accumulator
c. Conditional branch

d. Logical OR
e. I/O Operations

10.8 Many instruction sets contain the instruction NOOP, meaning no operation, which has no effect on the processor state other than incrementing the program counter. Suggest some uses of this instruction.

10.9 In Section 10.4, it was stated that both an arithmetic left shift and a logical left shift correspond to a multiplication by 2 when there is no overflow, and if overflow occurs, arithmetic and logical left shift operations produce different results, but the arithmetic left shift retains the sign of the number. Demonstrate that these statements are true for 5-bit twos complement integers.

10.10 In what way are numbers rounded using arithmetic right shift (e.g., round toward $+\infty$, round toward $-\infty$, toward zero, away from 0)?

10.11 Suppose a stack is to be used by the processor to manage procedure calls and returns. Can the program counter be eliminated by using the top of the stack as a program counter?

10.12 The Pentium architecture includes an instruction called Decimal Adjust after Addition (DAA). DAA performs the following sequence of instructions:

```
if ((AL AND 0FH) > 9) OR (AF = 1) then
    AL ← AL + 6;
    AF ← 1;
else
    AF ← 0;
endif;
if (AL > 9FH) OR (CF = 1) then
    AL ← AL + 60H;
    CF ← 1;
else
    AF ← 0;
endif.
```

"H" indicates hexadecimal. AL is an 8-bit register that holds the result of addition of two unsigned 8-bit integers. AF is a flag set if there is a carry from bit 3 to bit 4 in the result of an addition. CF is a flag set if there is a carry from bit 7 to bit 8. Explain the function performed by the DAA instruction.

10.13 The Pentium Compare instruction (CMP) subtracts the source operand from the destination operand; it updates the status flags (C, P, A, Z, S, O) but does not alter either of the operands. The CMP instruction can be used to determine if the destination operand is greater than, equal to, or less than the source operand.

a. Suppose the two operands are treated as unsigned integers. Show which status flags are relevant to determine the relative size of the two integer and what values of the flags correspond to greater than, equal to, or less than.
b. Suppose the two operands are treated as twos complement signed integers. Show which status flags are relevant to determine the relative size of the two integer and what values of the flags correspond to greater than, equal to, or less than.
c. The CMP instruction may be followed by a conditional Jump (Jcc) or Set Condition (SETcc) instruction, where cc refers to one of the 16 conditions listed in Table 10.11. Demonstrate that the conditions tested for a signed number comparison are correct.

10.14 Suppose we wished to apply the Pentium CMP instruction to 32-bit operands that contained numbers in a floating-point format. For correct results, what requirements have to be met in the following areas?

a. The relative position of the significand, sign, and exponent fields.
b. The representation of the value zero.
c. The representation of the exponent.
d. Does the IEEE format meet these requirements? Explain.

10.15 Most microprocessor instruction sets include an instruction that tests a condition and sets a destination operand if the condition is true. Examples include the SETcc on the Pentium, the Scc on the Motorola MC68000, and the Scond on the National NS32000.

a. There are a few differences among these instructions:
- SETcc and Scc operate only on a byte, whereas Scond operates on byte, word, and doubleword operands.
- SETcc and Scond set the operand to integer one if true and to zero if false. Scc sets the byte to all binary ones if true and all zeros if false.

What are the relative advantages and disadvantages of these differences?

b. None of these instructions set any of the condition code flags, and thus an explicit test of the result of the instruction is required to determine its value. Discuss whether condition codes should be set as a result of this instruction.

c. A simple IF statement such as IF a > b THEN can be implemented using a numerical representation method, that is, making the Boolean value manifest, as opposed to a *flow of control* method, which represents the value of a Boolean expression by a point reached in the program. A compiler might implement IF a > b THEN with the following 80 × 86 code:

```
          SUB    CX, CX    ;set register CX to 0
          MOV    AX, B     ;move contents of location B to register AX
          CMP    AX, A     ;compare contents of register AX and location A
          JLE    TEST      ;jump if A = B
          INC    CX        ;add 1 to contents of register CX
TEST             JCXZ      OUT          ;jump if contents of CX equal 0
THEN

OUT
```

The result of (A > B) is a Boolean value held in a register and available later on, outside the context of the flow of code just shown. It is convenient to use register CX for this, because many of the branch and loop opcodes have a built-in test for CX.

Show an alternative implementation using the SETcc instruction that saves memory and execution time. (*Hint:* No additional new 80 × 86 instructions are needed, other than the SETcc.)

d. Now consider the high-level language statement:

A: = (B > C) OR (D = F)

A compiler might generate the following code:

```
          MOV   EAX, B    ;move contents of location B to register EAX
          CMP   EAX, C    ;compare contents of register EAX and location C
          MOV   BL, 0     ;0 represents false
          JLE   N1        ;jump if B = C
          MOV   BL, 1     ;1 represents false
N1              MOV       EAX, D
          CMP   EAX, F
          MOV   BH, 0
          JNE   N2
          MOV   BH, 1
N2              OR        BL, BH
```

Show an alternative implementation using the SETcc instruction that saves memory and execution time.

10.16 Suppose that two registers contain the following hexadecimal values: AB0890C2, 4598EE50. What is the result of adding them using MMX instructions:

a. for packed byte
b. for packed word
Assume saturation arithmetic is not used.

10.17 Appendix 10A points out that there are no stack-oriented instructions in an instruction set if the stack is to be used only by the processor for such purposes as procedure handling. How can the processor use a stack for any purpose without stack-oriented instructions?

10.18 Convert the following formulas from reverse Polish to infix:
a. AB + C + D ×
b. AB/CD/ +
c. ABCDE+ × ×/
d. ABCDE + F/ + G − H/× +

10.19 Convert the following formulas from infix to reverse Polish:
a. A + B + C + D + E
b. (A + B) × (C + D) + E
c. (A × B) + (C × D) + E
d. (A − B) × (((C − D × E)/F)/G) × H

10.20 Convert the expression A + B − C to postfix notation using Dijkstra's algorithm. Show the steps involved. Is the result equivalent to (A + B) − C or A + (B − C)? Does it matter?

10.21 Using the algorithm for converting infix to postfix defined in Appendix 10A, show the steps involved in converting the expression of Figure 10.15 into postfix. Use a presentation similar to Figure 10.17.

10.22 Show the calculation of the expression in Figure 10.17, using a presentation similar to Figure 10.16.

10.23 Redraw the little-endian layout in Figure 10.18 so that the bytes appear as numbered in the big-endian layout. That is, show memory in 64-bit rows, with the bytes listed left to right, top to bottom.

10.24 For the following data structures, draw the big-endian and little-endian layouts, using the format of Figure 10.18, and comment on the results.

```
a.  struct {
        double i;    //0x1112131415161718
    } s1;
b.  struct {
        int i;       //0x11121314
        int j;       //0x15161718
    } s2;
c.  struct {
        short i;     //0x1112
        short j;     //0x1314
        short k;     //0x1516
        short l;     //0x1718
    } s3;
```

10.25 The PowerPC architecture specification does not dictate how a processor should implement little-endian mode. It specifies only the view of memory a processor must have when operating in little-endian mode. When converting a data structure from big endian to little endian, processors are free to implement a true byte-swapping mechanism or to use some sort of an address modification mechanism. Current PowerPC processors are all default big-endian machines and use address modification to treat data as little-endian.

Consider the structure s defined in Figure 10.18. The layout in the lower-right portion of the figure shows the structure s as seen by the processor. In fact, if structure s is compiled in little-endian mode, its layout in memory is shown in

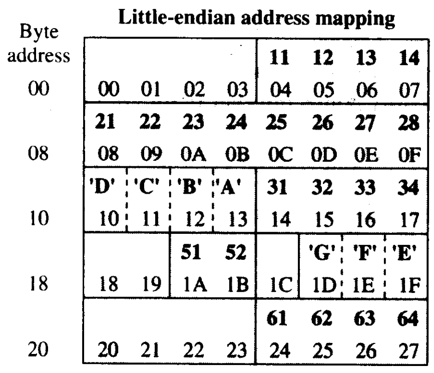


Figure 10.12 PowerPC Little-Endian Structure s in Memory

Figure 10.12. Explain the mapping that is involved, describe an easy way to implement the mapping, and discuss the effectiveness of this approach.

10.26 Write a small program to determine the endianness of machine and report the results. Run the program on a computer available to you and turn in the output.

10.27 The MIPS processor can be set to operate in either big-endian or little-endian mode. Consider the Load Byte Unsigned (LBU) instruction, which loads a byte from memory into the low-order 8 bits of a register and fills the high-order 24 bits of the register with zeros. The description of LBU is given in the MIPS reference manual using a register-transfer language as

```
mem ← LoadMemory(...)
byte ← $\text{VirtualAddress}_{1..0}$
**if** CONDITION **then**
    $\text{GPR}[rt] \leftarrow 0^{24} \,\|\, \text{mem}_{31 - 8 \times \text{byte} .. 24 - 8 \times \text{byte}}$
**else**
    $\text{GPR}[rt] \leftarrow 0^{24} \,\|\, \text{mem}_{7 + 8 \times \text{byte} .. 8 \times \text{byte}}$
**endif**
```

where *byte* refers to the two low-order bits of the effective address and *mem* refers to the value loaded from memory. In the manual, instead of the word CONDITION, one of the following two words is used: BigEndian, LittleEndian. Which word is used?

10.28 Most, but no all, processors use big- or little-endian bit ordering within a byte that is consistent with big- or little-endian ordering of bytes within a multibyte scalar. Let us consider the Motorola 68030, which uses big-endian byte ordering. The documentation of the 68030 concerning formats is confusing. The user's manual explains that the bit ordering of bit fields is the opposite of bit ordering of integers. Most bit field operations operate with one endian ordering, but a few bit field operations require the opposite ordering. The following description from the user's manual describes most of the bit field operations:

> A bit operand is specified by a base address that selects one byte in memory (the base byte), and a bit number that selects the one bit in this byte. The most significant bit is bit seven. A bit field operand is specified by **(1)** a base address that selects one byte in memory; **(2)** a bit field offset that indicates the leftmost (base) bit of the bit field in relation to the most significant bit of the base byte; and **(3)** a bit field width that determines how many bits to the right of the base byte are in the bit field. The most significant bit of the base byte is bit field offset 0, the least significant bit of the base byte is bit field offset 7.

Do these instructions use big-endian or little-endian bit ordering?

APPENDIX 10A STACKS

Stacks

A *stack* is an ordered set of elements, only one of which can be accessed at a time. The point of access is called the *top* of the stack. The number of elements in the stack, or *length* of the stack, is variable. The last element in the stack is the *base* of the stack. Items may only be added to or deleted from the top of the stack. For this reason, a stack is also known as a *pushdown list*[3] or a *last-in-first-out (LIFO) list.*

Figure 10.13 shows the basic stack operations. We begin at some point in time when the stack contains some number of elements. A PUSH operation appends one new item to the top of the stack. A POP operation removes the top item from the stack. In both cases, the top of the stack moves accordingly. Binary operations, which require two operands (e.g., multiply, divide, add, subtract), use the top two stack items as operands, pop both items, and push the result back onto the stack. Unary operations, which require only one operand (e.g., logical NOT), use the item on the top of the stack. All of these operations are summarized in Table 10.13.

Stack Implementation

The stack is a useful structure to provide as part of a processor implementation. One use, discussed in Section 10.4, is to manage procedure calls and returns. Stacks may also be useful to the programmer. An example of this is expression evaluation, discussed later in this section.

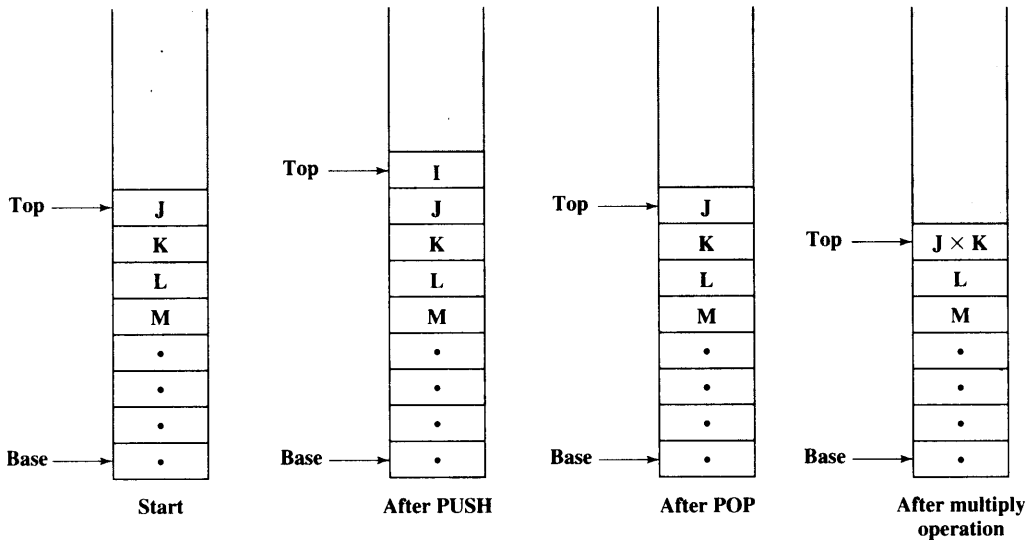


Figure 10.13 Basic Stack Operation

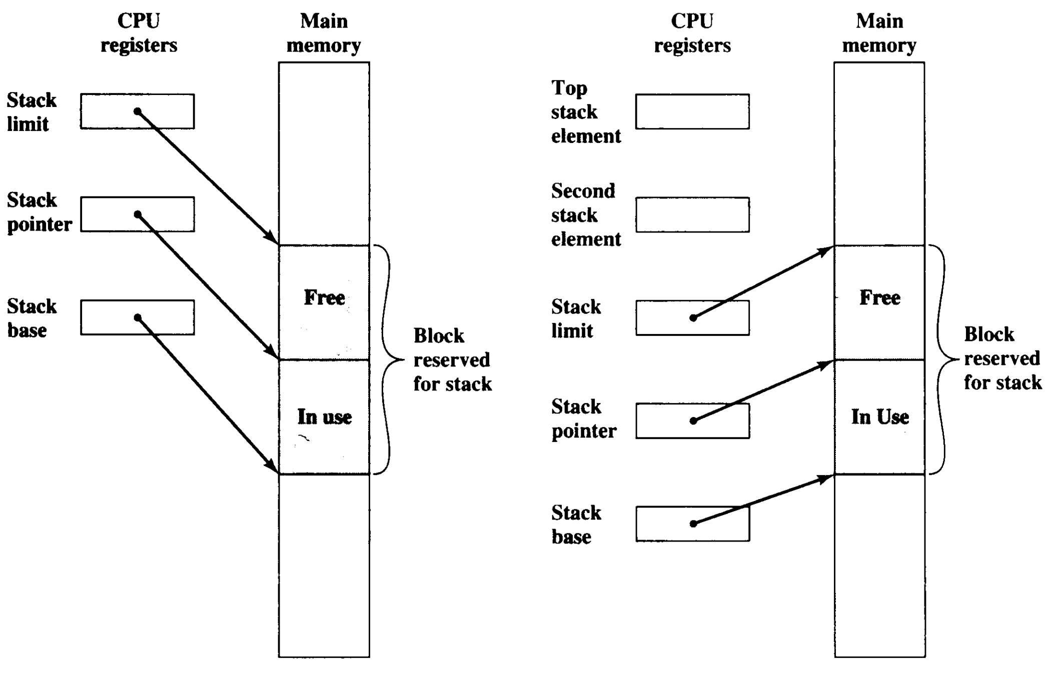
[3]A better term would be *place-on-top-of list* because the existing elements of the list are not moved in memory, but a new element is added at the next available memory address.

Table 10.13 Stack-Oriented Operations

PUSH	Append a new element on the top of the stack.
POP	Delete the top element of the stack.
Unary operation	Perform operation on top element of stack. Replace top element with result.
Binary operation	Perform operation on top two elements of stack. Delete top two elements of stack. Place result of operation on top of stack.

The implementation of a stack depends in part on its potential uses. If it is desired to make stack operations available to the programmer, then the instruction set will include stack-oriented operations, including PUSH, POP, and operations that use the top one or two stack elements as operands. Because all of these operations refer to a unique location, namely the top of the stack, the address of the operand or operands is implicit and need not be included in the instruction. These are the zero-address instructions referred to in Section 10.1.

If the stack mechanism is to be used only by the processor, for such purposes as procedure handling, then there will not be explicit stack-oriented instructions in the instruction set. In either case, the implementation of a stack requires that there be some set of locations used to store the stack elements. A typical approach is illustrated in Figure 10.14a. A contiguous block of locations is reserved in main memory (or virtual memory) for the stack. Most of the time, the block is partially filled with stack elements and the remainder is available for stack growth.



(a) All of stack in memory (b) Two top elements in registers

Figure 10.14 Typical Stack Organizations

Three addresses are needed for proper operation, and these are often stored in processor registers:

- **Stack pointer:** Contains the address of the top of the stack. If an item is appended to or deleted from the stack, the pointer is incremented or decremented to contain the address of the new top of the stack.
- **Stack base:** Contains the address of the bottom location in the reserved block. If an attempt is made to POP when the stack is empty, an error is reported.
- **Stack limit:** Contains the address of the other end of the reserved block. If an attempt is made to PUSH when the block is fully utilized for the stack, an error is reported.

Traditionally, and on most machines today, the base of the stack is at the high-address end of the reserved stack block, and the limit is at the low-address end. Thus, the stack grows from higher addresses to lower addresses.

To speed up stack operations, the top two stack elements are often stored in registers, as shown in Figure 10.14b. In this case, the stack pointer contains the address of the third element of the stack.

Expression Evaluation

Mathematical formulas are usually expressed in what is known as **infix** notation. In this form, a binary operation appears between the operands (e.g., $a + b$). For complex expressions, parentheses are used to determine the order of evaluation of expressions. For example, $a + (b \times c)$ will yield a different result than $(a + b) \times c$. To minimize the use of parentheses, operations have an implied precedence. Generally, multiplication takes precedence over addition, so that $a + b \times c$ is equivalent to $a + (b \times c)$.

An alternative technique is known as **reverse Polish**, or **postfix**, notation. In this notation, the operator follows its two operands. For example,

$a + b$	becomes $a\ b\ +$
$a + (b \times c)$	becomes $a\ b\ c \times +$
$(a + b) \times c$	becomes $a\ b + c \times$

Note that, regardless of the complexity of an expression, no parentheses are required when using reverse Polish.

The advantage of postfix notation is that an expression in this form is easily evaluated using a stack. An expression in postfix notation is scanned from left to right. For each element of the expression, the following rules are applied:

1. If the element is a variable or constant, push it onto the stack.
2. If the element is an operator, pop the top two items of the stack, perform the operation, and push the result.

After the entire expression has been scanned, the result is on the top of the stack.

The simplicity of this algorithm makes it a convenient one for evaluating expressions. Accordingly, many compilers will take an expression in a high-level language, convert it to postfix notation, and then generate the machine instructions from that notation. Figure 10.15 shows the sequence of machine instructions for

	Stack	General Registers	Single Register
	Push a	Load R1, a	Load d
	Push b	Subtract R1, b	Multiply e
	Subtract	Load R2, d	Add c
	Push c	Multiply R2, e	Store f
	Push d	Add R2, c	Load a
	Push e	Divide R1, R2	Subtract b
	Multiply	Store R1, f	Divide f
	Add		Store f
	Divide		
	Pop f		
Number of instructions	10	7	8
Memory access	10 op + 6 d	7 op + 6 d	8 op + 8 d

Figure 10.15 Comparison of Three Programs to Calculate $f = \dfrac{a - b}{c + (d \times e)}$

evaluating f = (a − b)/(c + d × e) using stack-oriented instructions. The figure also shows the use of one-address and two-address instructions. Note that, even though the stack-oriented rules were not used in the last two cases, the postfix notation served as a guide for generating the machine instructions. The sequence of events for the stack program is shown in Figure 10.16.

The process of converting an infix expression to a postfix expression is itself most easily accomplished using a stack. The following algorithm is due to Dijkstra [DIJK63]. The infix expression is scanned from left to right, and the postfix expression is developed and output during the scan. The steps are as follows:

1. Examine the next element in the input.
2. If it is an operand, output it.

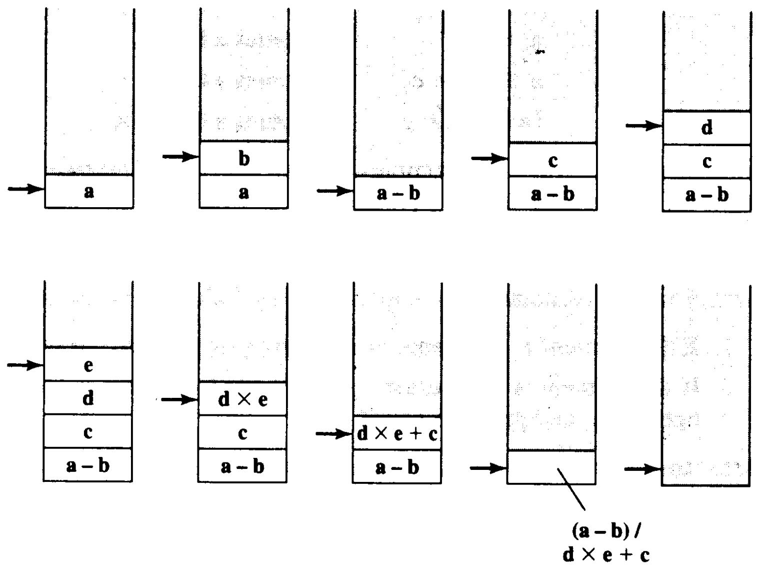


Figure 10.16 Use of Stock to Compute f = (a − b)/(d × e + c)

Input	Output	Stack (top on right)
A + B × C + (D + E) × F	empty	empty
+ B × C + (D + E) × F	A	empty
B × C + (D + E) × F	A	+
× C + (D + E) × F	A B	+
C + (D + E) × F	A B	+ ×
+ (D + E) × F	A B C	+ ×
(D + E) × F	A B C × +	+
D + E) × F	A B C × +	+ (
+ E) × F	A B C × + D	+ (
E) × F	A B C × + D	+ (+
) × F	A B C × + D E	+ (+
× F	A B C × + D E +	+
F	A B C × + D E +	+ ×
empty	A B C × + D E + F	+ ×
empty	A B C × + D E + F × +	empty

Figure 10.17 Conversion of an Expression from Infix to Postfix Notation

3. If it is an opening parenthesis, push it onto the stack.
4. If it is an operator, then
   - If the top of the stack is an opening parenthesis, then push the operator.
   - If it has higher priority than the top of the stack (multiply and divide have higher priority than add and subtract), then push the operator.
   - Else, pop operation from stack to output, and repeat step 4.
5. If it is a closing parenthesis, pop operators to the output until an opening parenthesis is encountered. Pop and discard the opening parenthesis.
6. If there is more input, go to step 1.
7. If there is no more input, unstack the remaining operands.

Figure 10.17 illustrates the use of this algorithm. This example should give the reader some feel for the power of stack-based algorithms.

APPENDIX 10B LITTLE-, BIG- AND BI-ENDIAN

An annoying and curious phenomenon relates to how the bytes within a word and the bits within a byte are both referenced and represented. We look first at the problem of byte ordering, and then consider that of bits.

Byte Ordering

The concept of endianness was first discussed in the literature by Cohen [COHE81]. With respect to bytes, endianness has to do with the byte ordering of multibyte scalar values. The issue is best introduced with an example. Suppose we have the 32-bit hexadecimal value 12345678 and that it is stored in a 32-bit word in byte-addressable memory at byte location 184. The value consists of four bytes, with the least significant byte containing the value 78 and the most significant byte containing the value 12. There are two ways to store this value:

Address	Value
184	12
185	34
186	56
187	78

Address	Value
184	78
185	56
186	34
187	12

The mapping on the left stores the most significant byte in the lowest numerical byte address; this is known as big endian and is equivalent to the left-to-right order of writing in Western culture languages. The mapping on the right stores the least significant byte in the lowest numerical byte address; this is known as little endian and is reminiscent of the right-to-left order of arithmetic operations in arithmetic units.[4] For a given multibyte scalar value, big endian and little endian are byte-reversed mappings of each other.

The concept of endianness arises when it is necessary to treat a multiple-byte entity as a single data item with a single address, even though it is composed of smaller addressable units. Some machines, such as the Intel 80x86, Pentium, VAX, and Alpha, are little-endian machines, whereas others, such as the IBM System 370/390, the Motorola 680x0, Sun SPARC, and most RISC machines, are big endian. This presents problems when data are transferred from a machine of one endian type to the other and when a programmer attempts to manipulate individual bytes or bits within a multibyte scalar.

The property of endianness does not extend beyond an individual data unit. In any machine, aggregates such as files, data structures, and arrays are composed of multiple data units, each with endianness. Thus, conversion of a block of memory from one style of endianness to the other requires knowledge of the data structure.

Figure 10.18 illustrates how endianness determines addressing and byte order. The C structure at the top contains a number of data types. The memory layout in the lower left results from compilation of that structure for a big-endian machine, and that in the lower right for a little-endian machine. In each case, memory is depicted as a series of 64-bit rows. For the big-endian case, memory typically is laid out left to right, top to bottom, whereas for the little-endian case, memory typically is laid out right to left, top to bottom. Note that these layouts are arbitrary. Either scheme could use either left to right or right to left within a row; this is a matter of depiction, not memory assignment. In fact, in looking at programmer manuals for a

[4]The terms *big endian* and *little endian* come from Part I, Chapter 4 of Jonathan Swift's *Gulliver's Travels*. They refer to a religious war between two groups, one that breaks eggs at the big end and the other that breaks eggs at the little end.

```
struct{
  int      a;      //0x1112_1314                         word
  int      pad;    //
  double   b;      //0x2122_2324_2526_2728               doubleword
  char*    c;      //0x3132_3334                         word
  char     d[7];   //'A'.'B','C','D','E','F','G'         byte array
  short    e;      //0x5152                              halfword
  int      f;      //0x6162_6364                         word
} s;
```

Big-endian address mapping

Byte address								
	11	**12**	**13**	**14**				
00	00	01	02	03	04	05	06	07
	21	**22**	**23**	**24**	**25**	**26**	**27**	**28**
08	08	09	0A	0B	0C	0D	0E	0F
	31	**32**	**33**	**34**	**'A'**	**'B'**	**'C'**	**'D'**
10	10	11	12	13	14	15	16	17
	'E'	**'F'**	**'G'**		**51**	**52**		
18	18	19	1A	1B	1C	1D	1E	1F
	61	**62**	**63**	**64**				
20	20	21	22	23				

Little-endian address mapping

								Byte address
				11	**12**	**13**	**14**	
07	06	05	04	03	02	01	00	00
21	**22**	**23**	**24**	**25**	**26**	**27**	**28**	
0F	0E	0D	0C	0B	0A	09	08	08
'D'	**'C'**	**'B'**	**'A'**	**31**	**32**	**33**	**34**	
17	16	15	14	13	12	11	10	10
		51	**52**		**'G'**	**'F'**	**'E'**	
1F	1E	1D	1C	1B	1A	19	18	18
				61	**62**	**63**	**64**	
				23	22	21	20	20

Figure 10.18 Example C Data Structure and its Endian Maps

variety of machines, a bewildering collection of depictions is to be found, even within the same manual.

We can make several observations about this data structure:

- Each data item has the same address in both schemes. For example, the address of the doubleword with hexadecimal value 2122232425262728 is 08.
- Within any given multibyte scalar value, the ordering of bytes in the little-endian structure is the reverse of that for the big-endian structure.
- Endianness does not affect the ordering of data items within a structure. Thus, the four-character word c exhibits byte reversal, but the seven-character byte array d does not. Hence, the address of each individual element of d is the same in both structures.

The effect of endianness is perhaps more clearly demonstrated when we view memory as a vertical array of bytes, as shown in Figure 10.19.

There is no general consensus as to which is the superior style of endianness.[5] The following points favor the big-endian style:

- **Character-string sorting:** A big-endian processor is faster in comparing integer-aligned character strings; the integer ALU can compare multiple bytes in parallel.

[5]The prophet revered by both groups in the Endian Wars of *Gulliver's Travels* had this to say. "All true Believers shall break their Eggs at the convenient End." Not much help!

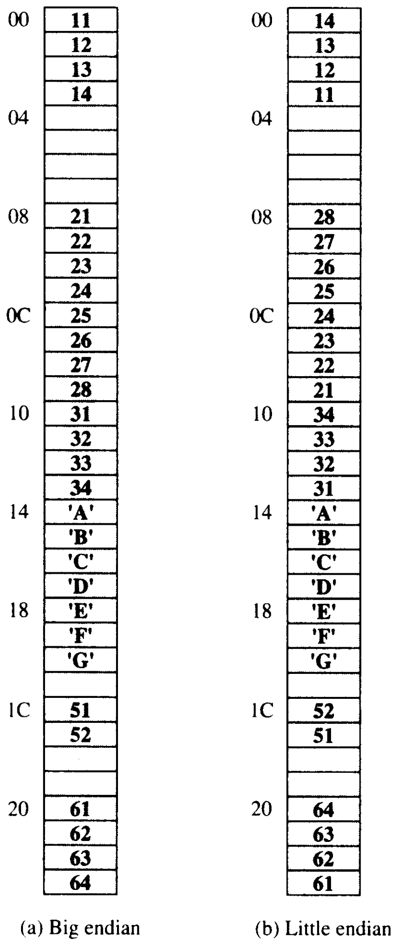


Figure 10.19 Another View of Figure 10.18

- **Decimal/IRA dumps:** All values can be printed left to right without causing confusion.
- **Consistent order:** Big-endian processors store their integers and character strings in the same order (most significant byte comes first).

The following points favor the little-endian style:

- A big-endian processor has to perform addition when it converts a 32-bit integer address to a 16-bit integer address, to use the least significant bytes.
- It is easier to perform higher-precision arithmetic with the little-endian style; you don't have to find the least-significant byte and move backward.

The differences are minor and the choice of endian style is often more a matter of accommodating previous machines than anything else.

The PowerPC is a bi-endian processor that supports both big-endian and little-endian modes. The bi-endian architecture enables software developers to choose either mode when migrating operating systems and applications from other machines. The operating system establishes the endian mode in which processes execute. Once a mode is selected, all subsequent memory loads and stores are determined by the memory-addressing model of that mode. To support this hardware feature, 2 bits are maintained in the machine state register (MSR) maintained by the operating system as part of the process state. One bit specifies the endian mode in which the kernel runs; the other specifies the processor's current operating mode. Thus, mode can be changed on a per-process basis.

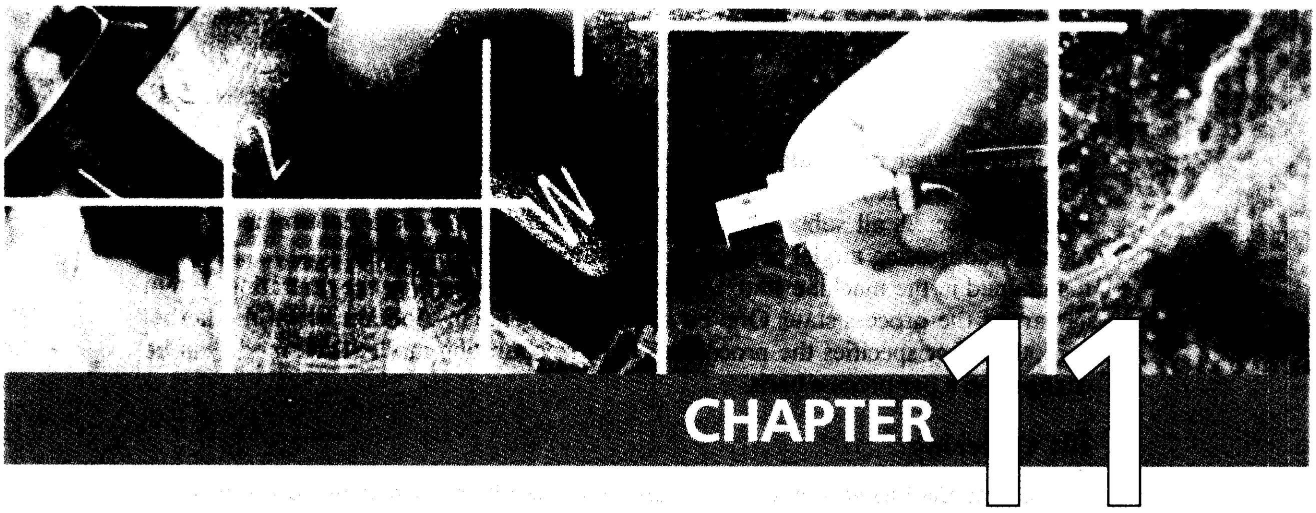
Bit Ordering

In ordering the bits within a byte, we are immediately faced with two questions:

1. Do you count the first bit as bit zero or as bit one?
2. Do you assign the lowest bit number to the byte's least significant bit (little endian) or to the bytes most significant bit (big endian)?

These questions are not answered in the same way on all machines. Indeed, on some machines, the answers are different in different circumstances. Furthermore, the choice of big- or little-endian bit ordering within a byte is not always consistent with big- or little-endian ordering of bytes within a multibyte scalar. The programmer needs to be concerned with these issues when manipulating individual bits.

Another area of concern is when data are transmitted over a bit-serial line. When an individual byte is transmitted, does the system transmit the most significant bit first or the least significant bit first? The designer must make certain that incoming bits are handled properly. For a discussion of this issue, see [JAME90].



INSTRUCTION SETS: ADDRESSING MODES AND FORMATS

KEY POINTS

- **An operand reference in an instruction either contains the actual value of the operand (immediate) or a reference to the address of the operand. A wide variety of addressing modes is used in various instruction sets. These include direct (operand address is in address field), indirect (address field points to a location that contains the operand address), register, register indirect, and various forms of displacement, in which a register value is added to an address value to produce the operand address.**
- **The instruction format defines the layout fields in the instruction. Instruction format design is a complex undertaking, including such consideration as instruction length, fixed or variable length, number of bits assigned to opcode and each operand reference, and how addressing mode is determined.**

In Chapter 10, we focused on *what* an instruction set does. Specifically, we examined the types of operands and operations that may be specified by machine instructions. This chapter turns to the question of *how* to specify the operands and operations of instructions. Two issues arise: First, how is the address of an operand specified, and second, how are the bits of an instruction organized to define the operand addresses and operation of that instruction?

11.1 ADDRESSING

The address field or fields in a typical instruction format are relatively small. We would like to be able to reference a large range of locations in main memory or, for some systems, virtual memory. To achieve this objective, a variety of addressing techniques has been employed. They all involve some trade-off between address range and/or addressing flexibility, on the one hand, and the number of memory references in the instruction and/or the complexity of address calculation, on the other. In this section, we examine the most common addressing techniques:

- Immediate
- Direct
- Indirect
- Register
- Register indirect
- Displacement
- Stack

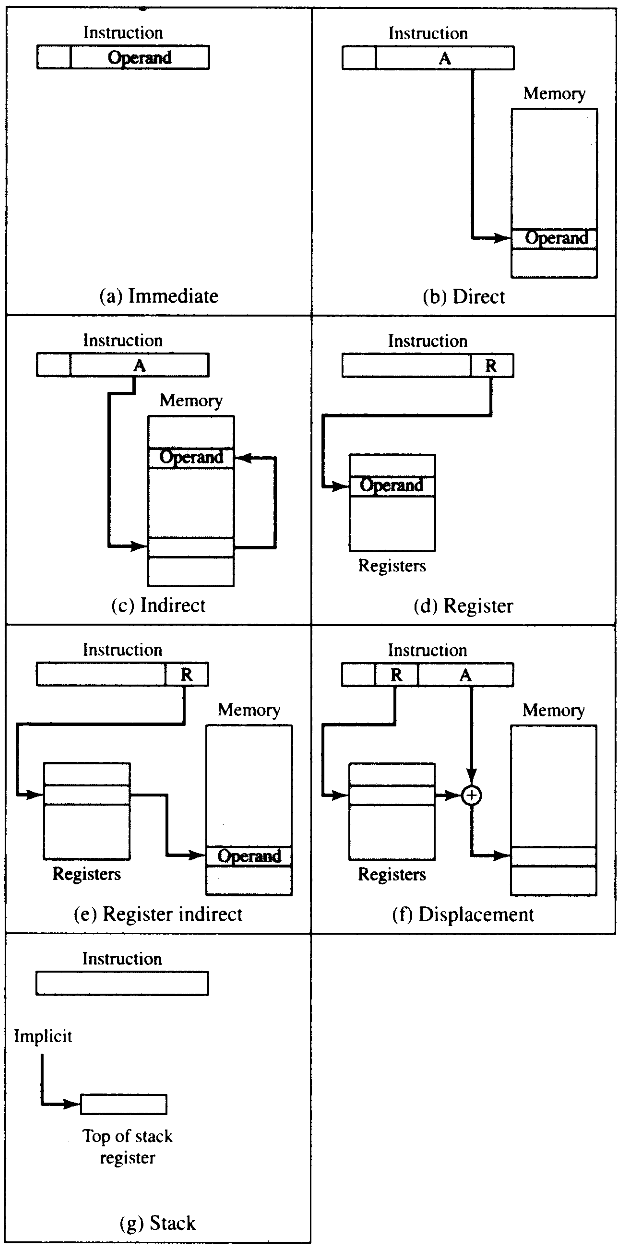


Figure 11.1 Addressing Modes

These modes are illustrated in Figure 11.1. In this section, we use the following notation:

A	= contents of an address field in the instruction
R	= contents of an address field in the instruction that refers to a register
EA	= actual (effective) address of the location containing the referenced operand
(X)	= contents of memory location X or register X

Table 11.1 Basic Addressing Modes

Mode	Algorithm	Principal Advantage	Principal Disadvantage
Immediate	Operand = A	No memory reference	Limited operand magnitude
Direct	EA = A	Simple	Limited address space
Indirect	EA = (A)	Large address space	Multiple memory references
Register	EA = R	No memory reference	Limited address space
Register indirect	EA = (R)	Large address space	Extra memory reference
Displacement	EA = A + (R)	Flexibility	Complexity
Stack	EA = top of stack	No memory reference	Limited applicability

Table 11.1 indicates the address calculation performed for each addressing mode.

Before beginning this discussion, two comments need to be made. First, virtually all computer architectures provide more than one of these addressing modes. The question arises as to how the processor can determine which address mode is being used in a particular instruction. Several approaches are taken. Often, different opcodes will use different addressing modes. Also, one or more bits in the instruction format can be used as a *mode field*. The value of the mode field determines which addressing mode is to be used.

The second comment concerns the interpretation of the effective address (EA). In a system without virtual memory, the *effective address* will be either a main memory address or a register. In a virtual memory system, the effective address is a virtual address or a register. The actual mapping to a physical address is a function of the paging mechanism and is invisible to the programmer.

Immediate Addressing

The simplest form of addressing is immediate addressing, in which the operand value is present in the instruction

$$\text{Operand} = A$$

This mode can be used to define and use constants or set initial values of variables. Typically, the number will be stored in twos complement form; the leftmost bit of the operand field is used as a sign bit. When the operand is loaded into a data register, the sign bit is extended to the left to the full data word size.

The advantage of immediate addressing is that no memory reference other than the instruction fetch is required to obtain the operand, thus saving one memory or cache cycle in the instruction cycle. The disadvantage is that the size of the number is restricted to the size of the address field, which, in most instruction sets, is small compared with the word length.

Direct Addressing

A very simple form of addressing is direct addressing, in which the address field contains the effective address of the operand:

$$EA = A$$

The technique was common in earlier generations of computers but is not common on contemporary architectures. It requires only one memory reference and no special calculation. The obvious limitation is that it provides only a limited address space.

Indirect Addressing

With direct addressing, the length of the address field is usually less than the word length, thus limiting the address range. One solution is to have the address field refer to the address of a word in memory, which in turn contains a full-length address of the operand. This is known as *indirect addressing:*

$$\text{EA} = (\text{A})$$

As defined earlier, the parentheses are to be interpreted as meaning *contents of.* The obvious advantage of this approach is that for a word length of N, an address space of 2^N is now available. The disadvantage is that instruction execution requires two memory references to fetch the operand: one to get its address and a second to get its value.

Although the number of words that can be addressed is now equal to 2^N, the number of different effective addresses that may be referenced at any one time is limited to 2^K, where K is the length of the address field. Typically, this is not a burdensome restriction, and it can be an asset. In a virtual memory environment, all the effective address locations can be confined to page 0 of any process. Because the address field of an instruction is small, it will naturally produce low-numbered direct addresses, which would appear in page 0. (The only restriction is that the page size must be greater than or equal to 2^K.) When a process is active, there will be repeated references to page 0, causing it to remain in real memory. Thus, an indirect memory reference will involve, at most, one page fault rather than two.

A rarely used variant of indirect addressing is multilevel or cascaded indirect addressing:

$$\text{EA} = (\ldots(\text{A})\ldots)$$

In this case, one bit of a full-word address is an indirect flag (I). If the I bit is 0, then the word contains the EA. If the I bit is 1, then another level of indirection is invoked. There does not appear to be any particular advantage to this approach, and its disadvantage is that three or more memory references could be required to fetch an operand.

Register Addressing

Register addressing is similar to direct addressing. The only difference is that the address field refers to a register rather than a main memory address:

$$\text{EA} = \text{R}$$

To clarify, if the contents of a register address field in an instruction is 5, then register R5 is the intended address, and the operand value is contained in R5. Typically, an address field that references registers will have from 3 to 5 bits, so that a total of from 8 to 32 general-purpose registers can be referenced.

The advantages of register addressing are that (1) only a small address field is needed in the instruction, and (2) no time-consuming memory references are required. As was discussed in Chapter 4, the memory access time for a register internal to the processor is much less than that for a main memory address. The disadvantage of register addressing is that the address space is very limited.

If register addressing is heavily used in an instruction set, this implies that the processor registers will be heavily used. Because of the severely limited number of registers (compared with main memory locations), their use in this fashion makes sense only if they are employed efficiently. If every operand is brought into a register from main memory, operated on once, and then returned to main memory, then a wasteful intermediate step has been added. If, instead, the operand in a register remains in use for multiple operations, then a real savings is achieved. An example is the intermediate result in a calculation. In particular, suppose that the algorithm for twos complement multiplication were to be implemented in software. The location labeled A in the flowchart (Figure 9.12) is referenced many times and should be implemented in a register rather than a main memory location.

It is up to the programmer to decide which values should remain in registers and which should be stored in main memory. Most modern processors employ multiple general-purpose registers, placing a burden for efficient execution on the assembly-language programmer (e.g., compiler writer).

Register Indirect Addressing

Just as register addressing is analogous to direct addressing, register indirect addressing is analogous to indirect addressing. In both cases, the only difference is whether the address field refers to a memory location or a register. Thus, for register indirect address,

$$\text{EA} = (\text{R})$$

The advantages and limitations of register indirect addressing are basically the same as for indirect addressing. In both cases, the address space limitation (limited range of addresses) of the address field is overcome by having that field refer to a word-length location containing an address. In addition, register indirect addressing uses one less memory reference than indirect addressing.

Displacement Addressing

A very powerful mode of addressing combines the capabilities of direct addressing and register indirect addressing. It is known by a variety of names depending on the context of its use, but the basic mechanism is the same. We will refer to this as *displacement addressing:*

$$\text{EA} = \text{A} + (\text{R})$$

Displacement addressing requires that the instruction have two address fields, at least one of which is explicit. The value contained in one address field (value = A) is used directly. The other address field, or an implicit reference based on opcode, refers to a register whose contents are added to A to produce the effective address.

We will describe three of the most common uses of displacement addressing:

- Relative addressing
- Base-register addressing
- Indexing

Relative Addressing For relative addressing, also called PC-relative addressing, the implicitly referenced register is the program counter (PC). That is, the next instruction address is added to the address field to produce the EA. Typically, the address field is treated as a twos complement number for this operation. Thus, the effective address is a displacement relative to the address of the instruction.

Relative addressing exploits the concept of locality that was discussed in Chapters 4 and 8. If most memory references are relatively near to the instruction being executed, then the use of relative addressing saves address bits in the instruction.

Base-Register Addressing For base-register addressing, the interpretation is the following: The referenced register contains a main memory address, and the address field contains a displacement (usually an unsigned integer representation) from that address. The register reference may be explicit or implicit.

Base-register addressing also exploits the locality of memory references. It is a convenient means of implementing segmentation, which was discussed in Chapter 8. In some implementations, a single segment-base register is employed and is used implicitly. In others, the programmer may choose a register to hold the base address of a segment, and the instruction must reference it explicitly. In this latter case, if the length of the address field is K and the number of possible registers is N, then one instruction can reference any one of N areas of 2^K words.

Indexing For indexing, the interpretation is typically the following: The address field references a main memory address, and the referenced register contains a positive displacement from that address. Note that this usage is just the opposite of the interpretation for base-register addressing. Of course, it is more than just a matter of user interpretation. Because the address field is considered to be a memory address in indexing, it generally contains more bits than an address field in a comparable base-register instruction. Also, we shall see that there are some refinements to indexing that would not be as useful in the base-register context. Nevertheless, the method of calculating the EA is the same for both base-register addressing and indexing, and in both cases the register reference is sometimes explicit and sometimes implicit (for different processor types).

An important use of indexing is to provide an efficient mechanism for performing iterative operations. Consider, for example, a list of numbers stored starting at location A. Suppose that we would like to add 1 to each element on the list. We need to fetch each value, add 1 to it, and store it back. The sequence of effective addresses that we need is A, A + 1, A + 2, . . . , up to the last location on the list. With indexing, this is easily done. The value A is stored in the instruction's address field, and the chosen register, called an *index register*, is initialized to 0. After each operation, the index register is incremented by 1.

Because index registers are commonly used for such iterative tasks, it is typical that there is a need to increment or decrement the index register after each reference to it. Because this is such a common operation, some systems will automatically do this as part of the same instruction cycle. This is known as *autoindexing*. If certain registers are devoted exclusively to indexing, then autoindexing can be invoked implicitly and automatically. If general-purpose registers are used, the autoindex operation may need to be signaled by a bit in the instruction. Autoindexing using increment can be depicted as follows:

$$\begin{aligned} \text{EA} &= \text{A} + (\text{R}) \\ (\text{R}) &\leftarrow (\text{R}) + 1 \end{aligned}$$

In some machines, both indirect addressing and indexing are provided, and it is possible to employ both in the same instruction. There are two possibilities: The indexing is performed either before or after the indirection.

If indexing is performed after the indirection, it is termed *postindexing:*

$$\text{EA} = (\text{A}) + (\text{R})$$

First, the contents of the address field are used to access a memory location containing a direct address. This address is then indexed by the register value. This technique is useful for accessing one of a number of blocks of data of a fixed format. For example, it was described in Chapter 8 that the operating system needs to employ a process control block for each process. The operations performed are the same regardless of which block is being manipulated. Thus, the addresses in the instructions that reference the block could point to a location (value = A) containing a variable pointer to the start of a process control block. The index register contains the displacement within the block.

With *preindexing,* the indexing is performed before the indirection:

$$\text{EA} = (\text{A} + (\text{R}))$$

An address is calculated as with simple indexing. In this case, however, the calculated address contains not the operand, but the address of the operand. An example of the use of this technique is to construct a multiway branch table. At a particular point in a program, there may be a branch to one of a number of locations depending on conditions. A table of addresses can be set up starting at location A. By indexing into this table, the required location can be found.

Typically, an instruction set will not include both preindexing and postindexing.

Stack Addressing

The final addressing mode that we consider is stack addressing. As defined in Appendix 9A, a stack is a linear array of locations. It is sometimes referred to as a *pushdown list* or *last-in-first-out queue.* The stack is a reserved block of locations. Items are appended to the top of the stack so that, at any given time, the block is partially filled. Associated with the stack is a pointer whose value is the address of the top of the stack. Alternatively, the top two elements of the stack may be in processor registers, in which case the stack pointer references the third element of the stack (Figure 10.14b). The stack pointer is maintained in a register. Thus, references to stack locations in memory are in fact register indirect addresses.

The stack mode of addressing is a form of implied addressing. The machine instructions need not include a memory reference but implicitly operate on the top of the stack.

11.2 PENTIUM AND POWERPC ADDRESSING MODES

Pentium Addressing Modes

Recall from Figure 8.21 that the Pentium address translation mechanism produces an address, called a virtual or effective address, that is an offset into a segment. The sum of the starting address of the segment and the effective address produces a linear address. If paging is being used, this linear address must pass through a page-translation mechanism to produce a physical address. In what follows, we ignore this last step because it is transparent to the instruction set and to the programmer.

The Pentium is equipped with a variety of addressing modes intended to allow the efficient execution of high-level languages. Figure 11.2 indicates the logic involved. The segment register determines the segment that is the subject of the reference. There are six segment registers; the one being used for a particular reference depends on the

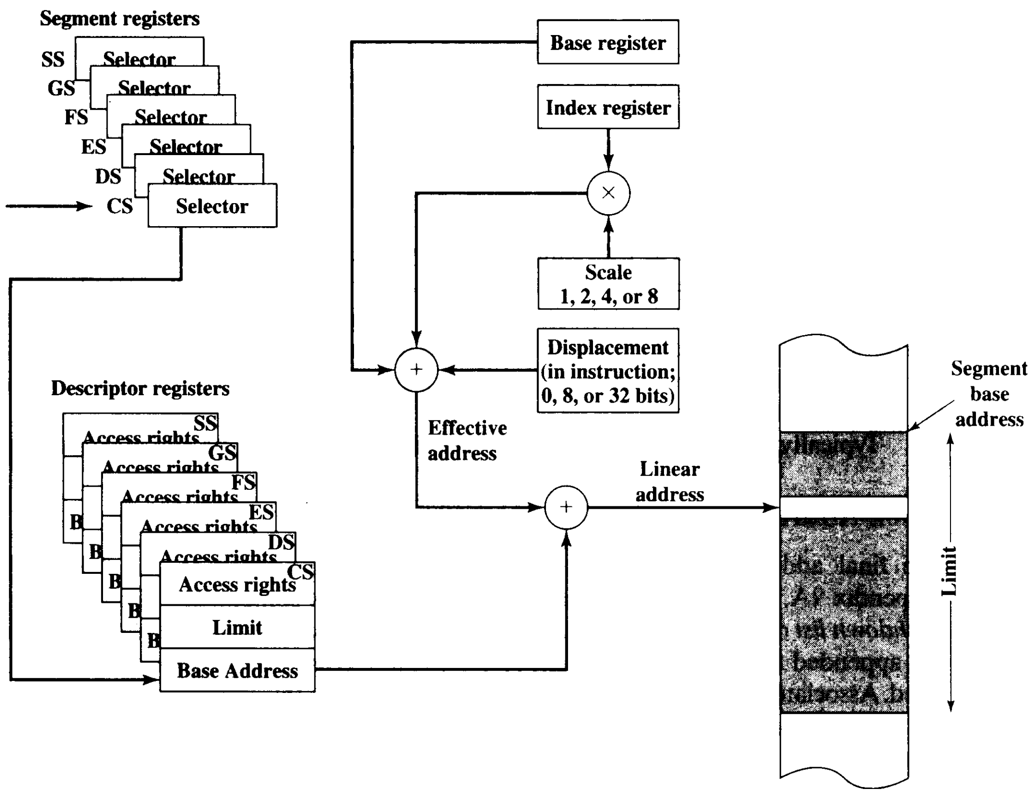


Figure 11.2 Pentium Addressing Mode Calculation

Table 11.2 Pentium Addressing Modes

Mode	Algorithm
Immediate	Operand = A
Register Operand	LA = R
Displacement	LA = (SR) + A
Base	LA = (SR) + (B)
Base with Displacement	LA = (SR) + (B) + A
Scaled Index with Displacement	LA = (SR) + (I) × S + A
Base with Index and Displacement	LA = (SR) + (B) + (I) + A
Base with Scaled Index and Displacement	LA = (SR) + (I) × S + (B) + A
Relative	LA = (PC) + A

LA	=	linear address
(X)	=	contents of X
SR	=	segment register
PC	=	program counter
A	=	contents of an address field in the instruction
R	=	register
B	=	base register
I	=	index register
S	=	scaling factor

context of execution and the instruction. Each segment register holds the starting address of the corresponding segment. Associated with each user-visible segment register is a segment descriptor register (not programmer visible), which records the access rights for the segment as well as the starting address and limit (length) of the segment. In addition, there are two registers that may be used in constructing an address: the base register and the index register.

Table 11.2 lists the 12 Pentium addressing modes. Let us consider each of these in turn.

For the **immediate mode**, the operand is included in the instruction. The operand can be a byte, word, or doubleword of data.

For **register operand mode**, the operand is located in a register. For general instructions, such as data transfer, arithmetic, and logical instructions, the operand can be one of the 32-bit general registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP), one of the 16-bit general registers (AX, BX, CX, DX, SI, DI, SP, BP), or one of the 8-bit general registers (AH, BH, CH, DH, AL, BL, CL, DL). For floating-point operations, 64-bit operands are formed by using two 32-bit registers as a pair. There are also some instructions that reference the segment registers (CS, DS, ES, SS, FS, GS).

The remaining addressing modes reference locations in memory. The memory location must be specified in terms of the segment containing the location and the offset from the beginning of the segment. In some cases, a segment is specified explicitly; in others, the segment is specified by simple rules that assign a segment by default.

In the **displacement mode**, the operand's offset (the effective address of Figure 11.2) is contained as part of the instruction as an 8-, 16-, or 32-bit displacement.

With segmentation, all addresses in instructions refer merely to an offset in a segment. The displacement addressing mode is found on few machines because, as mentioned earlier, it leads to long instructions. In the case of the Pentium, the displacement value can be as long as 32 bits, making for a 6-byte instruction. Displacement addressing can be useful for referencing global variables.

The remaining addressing modes are indirect, in the sense that the address portion of the instruction tells the processor where to look to find the address. The **base mode** specifies that one of the 8-, 16-, or 32-bit registers contains the effective address. This is equivalent to what we have referred to as register indirect addressing.

In the **base with displacement mode**, the instruction includes a displacement to be added to a base register, which may be any of the general-purpose registers. Examples of uses of this mode are as follows:

- Used by a compiler to point to the start of a local variable area. For example, the base register could point to the beginning of a stack frame, which contains the local variables for the corresponding procedure.
- Used to index into an array when the element size is not 1, 2, 4, or 8 bytes and which therefore cannot be indexed using an index register. In this case, the displacement points to the beginning of the array, and the base register holds the results of a calculation to determine the offset to a specific element within the array.
- Used to access a field of a record. The base register points to the beginning of the record, while the displacement is an offset to the field.

In the **scaled index with displacement mode**, the instruction includes a displacement to be added to a register, in this case called an index register. The index register may be any of the general-purpose registers except the one called ESP, which is generally used for stack processing. In calculating the effective address, the contents of the index register are multiplied by a scaling factor of 1, 2, 4, or 8, and then added to a displacement. This mode is very convenient for indexing arrays. A scaling factor of 2 can be used for an array of 16-bit integers. A scaling factor of 4 can be used for 32-bit integers or floating-point numbers. Finally, a scaling factor of 8 can be used for an array of double-precision floating-point numbers.

The **base with index and displacement mode** sums the contents of the base register, the index register, and a displacement to form the effective address. Again, the base register can be any general-purpose register and the index register can be any general-purpose register except ESP. As an example, this addressing mode could be used for accessing a local array on a stack frame. This mode can also be used to support a two-dimensional array; in this case, the displacement points to the beginning of the array, and each register handles one dimension of the array.

The **based scaled index with displacement mode** sums the contents of the index register multiplied by a scaling factor, the contents of the base register, and the displacement. This is useful if an array is stored in a stack frame; in this case, the array elements would be 2, 4, or 8 bytes each in length. This mode also provides efficient indexing of a two-dimensional array when the array elements are 2, 4, or 8 bytes in length.

Finally, **relative addressing** can be used in transfer-of-control instructions. A displacement is added to the value of the program counter, which points to the next

Table 11.3 PowerPC Addressing Modes

Mode	Algorithm
	Load/Store Addressing
Indirect	EA = (BR) + D
Indirect Indexed	EA = (BR) + (IR)
	Branch Addressing
Absolute	EA = I
Relative	EA = (PC) + I
Indirect	EA = (L/CR)
	Fixed-Point Computation
Register	EA = GPR
Immediate	Operand = I
	Floating-Point Computation
Register	EA = FPR

EA	= effective address
(X)	= contents of X
BR	= base register
IR	= index register
L/CR	= link or count register
GPR	= general-purpose register
FPR	= floating-point register
D	= displacement
I	= immediate value
PC	= program counter

instruction. In this case, the displacement is treated as a signed byte, word, or doubleword value, and that value either increases or decreases the address in the program counter.

PowerPC Addressing Modes

In common with most RISC machines, and unlike the Pentium and most CISC machines, the PowerPC uses a simple and relatively straightforward set of addressing modes. As Table 11.3 indicates, these modes are conveniently classified with respect to the type of instruction.

Load/Store Architecture The PowerPC provides two alternative addressing modes for load/store instructions (Figure 11.3). With indirect addressing, the instruction includes a 16-bit displacement to be added to a base register, which may be any of the general-purpose registers. In addition, the instruction may specify that the newly computed effective address is to be fed back to the base register, updating the current contents. The update option is useful for progressive indexing of arrays in loops.

The other addressing technique for load/store instructions is **indirect indexed addressing**. In this case, the instruction references a base register and an index register,

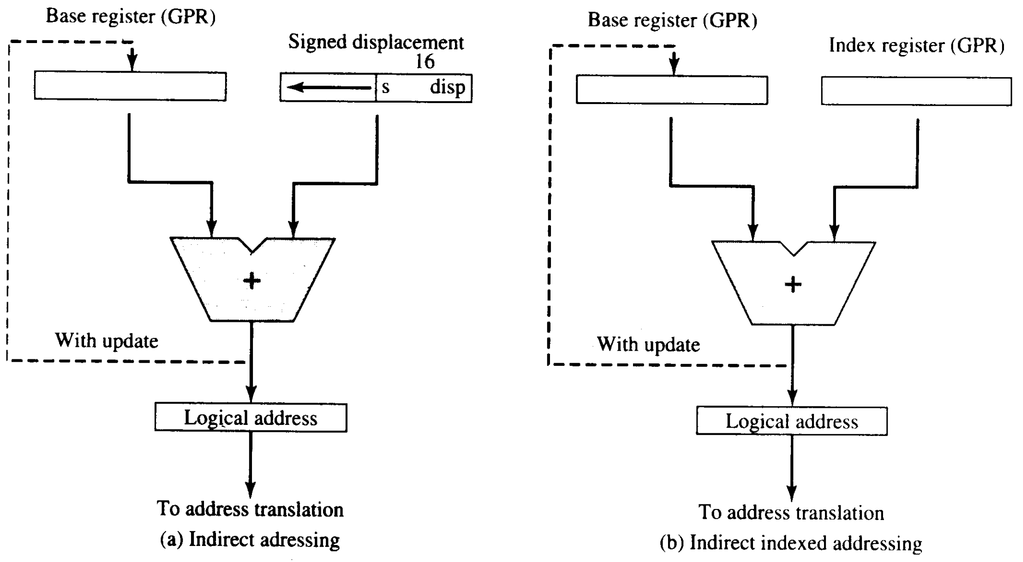


Figure 11.3 PowerPC Memory Operand Addressing Modes

both of which may be any of the general-purpose registers. The effective address is the sum of the contents of these two registers. Again, the update option causes the base register to be updated to the new effective address.

Branch Addressing Three branch addressing modes are provided. When **absolute addressing** is used with unconditional branch instructions, the effective address of the next instruction is derived from a 24-bit immediate value within the instruction. The 24-bit value is extended to a 32-bit value by adding two zeros to its least significant end (this is permissible because all instructions must occur on 32-bit boundaries) and sign extending. For conditional branch instructions, the effective address of the next instruction is derived from a 16-bit immediate value within the instruction. The 16-bit value is extended to a 32-bit value by adding two zeros to its least significant end and sign extending.

With **relative addressing**, the 24-bit immediate value (unconditional branch instructions) or 14-bit immediate value (conditional branch instructions) is extended as before. The resulting value is then added to the program counter to define a location relative to the current instruction. The other conditional branch addressing mode is **indirect addressing**. This mode obtains the effective address of the next instruction from either the link register or the count register. Note that in this case the count register is used to hold the address for a branch instruction. This register may also be used to hold a count for looping, as explained earlier.

Arithmetic Instructions For integer arithmetic, all operands must be contained either in registers or as part of the instruction. With register addressing, a source or destination operand is specified as one of the general-purpose registers. With immediate addressing, a source operand appears as a 16-bit signed quantity in the instruction.

For floating-point arithmetic, all operands are in floating-point registers; that is, only register addressing is used.

11.3 INSTRUCTION FORMATS

An instruction format defines the layout of the bits of an instruction, in terms of its constituent fields. An instruction format must include an opcode and, implicitly or explicitly, zero or more operands. Each explicit operand is referenced using one of the addressing modes described in Section 11.1. The format must, implicitly or explicitly, indicate the addressing mode for each operand. For most instruction sets, more than one instruction format is used.

The design of an instruction format is a complex art, and an amazing variety of designs have been implemented. We examine the key design issues, looking briefly at some designs to illustrate points, and then we examine the Pentium and PowerPC solutions in detail.

Instruction Length

The most basic design issue to be faced is the instruction format length. This decision affects, and is affected by, memory size, memory organization, bus structure, processor complexity, and processor speed. This decision determines the richness and flexibility of the machine as seen by the assembly-language programmer.

The most obvious trade-off here is between the desire for a powerful instruction repertoire and a need to save space. Programmers want more opcodes, more operands, more addressing modes, and greater address range. More opcodes and more operands make life easier for the programmer, because shorter programs can be written to accomplish given tasks. Similarly, more addressing modes give the programmer greater flexibility in implementing certain functions, such as table manipulations and multiple-way branching. And, of course, with the increase in main memory size and the increasing use of virtual memory, programmers want to be able to address larger memory ranges. All of these things (opcodes, operands, addressing modes, address range) require bits and push in the direction of longer instruction lengths. But longer instruction length may be wasteful. A 64-bit instruction occupies twice the space of a 32-bit instruction but is probably less than twice as useful.

Beyond this basic trade-off, there are other considerations. Either the instruction length should be equal to the memory-transfer length (in a bus system, data-bus length) or one should be a multiple of the other. Otherwise, we will not get an integral number of instructions during a fetch cycle. A related consideration is the memory transfer rate. This rate has not kept up with increases in processor speed. Accordingly, memory can become a bottleneck if the processor can execute instructions faster than it can fetch them. One solution to this problem is to use cache memory (see Section 4.3); another is to use shorter instructions. Thus, 16-bit instructions can be fetched at twice the rate of 32-bit instructions but probably can be executed less than twice as fast.

A seemingly mundane but nevertheless important feature is that the instruction length should be a multiple of the character length, which is usually 8 bits, and of the length of fixed-point numbers. To see this, we need to make use of that unfortunately ill-defined word, *word* [FRAI83]. The word length of memory is, in some sense, the "natural" unit of organization. The size of a word usually determines the size of fixed-point numbers (usually the two are equal). Word size is also typically equal to, or at least integrally related to, the memory transfer size. Because a common form of data is character data, we would like a word to store an integral number of characters. Otherwise, there are wasted bits in each word when storing multiple characters, or a character will have to straddle a word boundary. The importance of this point is such that IBM, when it introduced the System/360 and wanted to employ 8-bit characters, made the wrenching decision to move from the 36-bit architecture of the scientific members of the 700/7000 series to a 32-bit architecture.

Allocation of Bits

We've looked at some of the factors that go into deciding the length of the instruction format. An equally difficult issue is how to allocate the bits in that format. The trade-offs here are complex.

For a given instruction length, there is clearly a trade-off between the number of opcodes and the power of the addressing capability. More opcodes obviously mean more bits in the opcode field. For an instruction format of a given length, this reduces the number of bits available for addressing. There is one interesting refinement to this trade-off, and that is the use of variable-length opcodes. In this approach, there is a minimum opcode length but, for some opcodes, additional operations may be specified by using additional bits in the instruction. For a fixed-length instruction, this leaves fewer bits for addressing. Thus, this feature is used for those instructions that require fewer operands and/or less powerful addressing.

The following interrelated factors go into determining the use of the addressing bits.

- **Number of addressing modes:** Sometimes an addressing mode can be indicated implicitly. For example, certain opcodes might always call for indexing. In other cases, the addressing modes must be explicit, and one or more mode bits will be needed.
- **Number of operands:** We have seen that fewer addresses can make for longer, more awkward programs (e.g., Figure 10.3). Typical instructions on today's machines provide for two operands. Each operand address in the instruction might require its own mode indicator, or the use of a mode indicator could be limited to just one of the address fields.
- **Register versus memory:** A machine must have registers so that data can be brought into the processor for processing. With a single user-visible register (usually called the accumulator), one operand address is implicit and consumes no instruction bits. However, single-register programming is awkward and requires many instructions. Even with multiple registers, only a few bits are needed to specify the register. The more that registers can be used for operand references, the fewer bits are needed. A number of studies indicate that a total of 8 to 32 user-visible registers is desirable [LUND77, HUCK83]. Most contemporary architectures have at least 32 registers.
- **Number of register sets:** Most contemporary machines have one set of general-purpose registers, with typically 32 or more registers in the set. These registers can be used to store data and can be used to store addresses for displacement