Table 8.5 Pentium II Memory Management Parameters

Segment Descriptor (Segment Table Entry)

Base
Defines the starting address of the segment within the 4-GByte linear address space.

D/B bit
In a code segment, this is the D bit and indicates whether operands and addressing modes are 16 or 32 bits.

Descriptor Privilege Level (DPL)
Specifies the privilege level of the segment referred to by this segment descriptor.

Granularity bit (G)
Indicates whether the Limit field is to be interpreted in units by one byte or 4 KBytes.

Limit
Defines the size of the segment. The processor interprets the limit field in one of two ways, depending on the granularity bit: in units of one byte, up to a segment size limit of 1 MByte, or in units of 4 KBytes, up to a segment size limit of 4 GBytes.

S bit
Determines whether a given segment is a system segment or a code or data segment.

Segment Present bit (P)
Used for nonpaged systems. It indicates whether the segment is present in main memory. For paged systems, this bit is always set to 1.

Type
Distinguishes between various kinds of segments and indicates the access attributes.

Page Directory Entry and Page Table Entry

Accessed bit (A)
This bit is set to 1 by the processor in both levels of page tables when a read or write operation to the corresponding page occurs.

Dirty bit (D)
This bit is set to 1 by the processor when a write operation to the corresponding page occurs.

Page Frame Address
Provides the physical address of the page in memory if the present bit is set. Since page frames are aligned on 4K boundaries, the bottom 12 bits are 0, and only the top 20 bits are included in the entry. In a page directory, the address is that of a page table.

Page Cache Disable bit (PCD)
Indicates whether data from page may be cached.

Page Size bit (PS)
Indicates whether page size is 4 KByte or 4 MByte.

Page Write Through bit (PWT)
Indicates whether write-through or write-back caching policy will be used for data in the corresponding page.

Present bit (P)
Indicates whether the page table or page is in main memory.

Read/Write bit (RW)
For user-level pages, indicates whether the page is read-only access or read/write access for user-level programs.

User/Supervisor bit (US)
Indicates whether the page is available only to the operating system (supervisor level) or is available to both operating system and applications (user level).

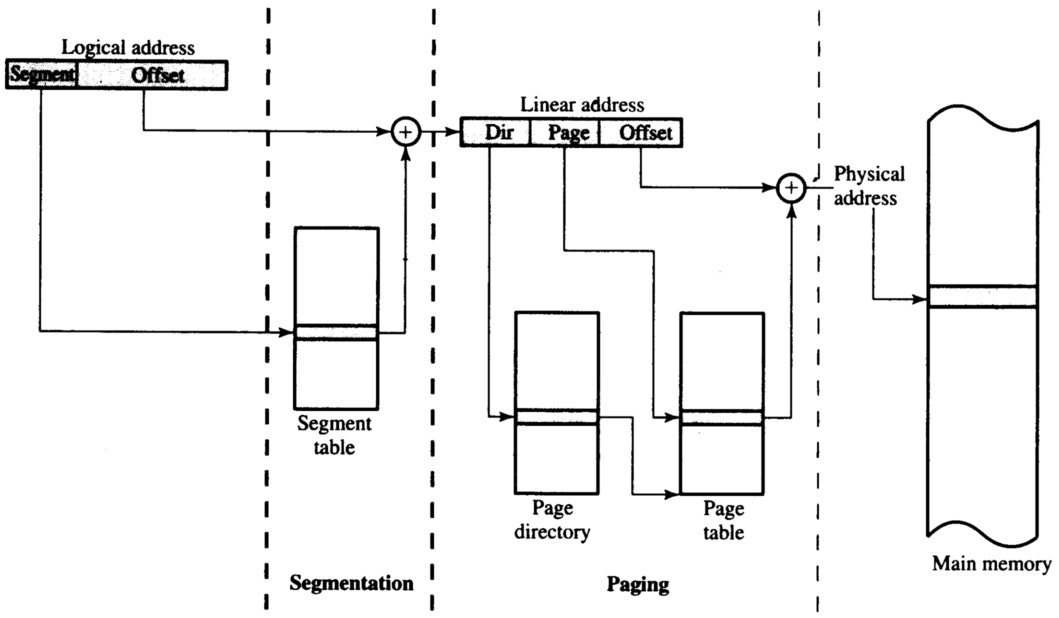


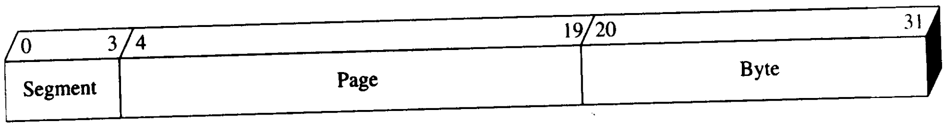
Figure 8.21 Pentium Memory Address Translation Mechanisms

PowerPC Memory-Management Hardware

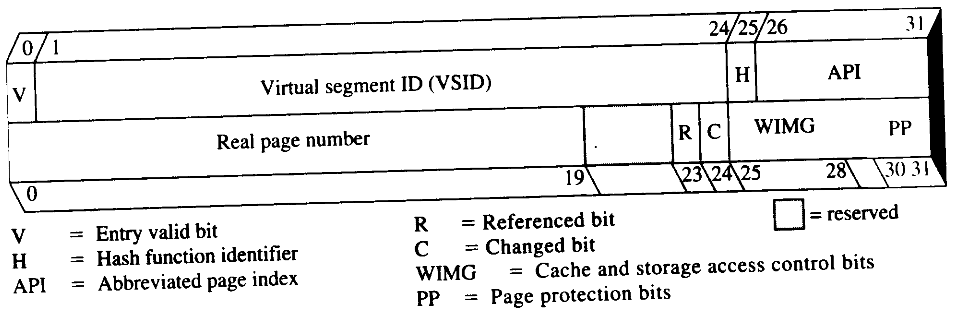
The PowerPC provides a comprehensive set of addressing mechanisms. For 32-bit implementations of the architecture, a paging scheme with a simple segmentation mechanism is implemented. For 64-bit implementations, paging and a more powerful segmentation mechanism are supported. In addition, for both 32-bit and 64-bit processors there is an alternative hardware mechanism, known as block address translation. Briefly, the block addressing scheme is designed to address one drawback of paging mechanisms. With paging, a large number of pages may be frequently referenced by a program. For example, programs that use OS tables or graphics frame buffers may exhibit this behavior. The result may be that frequently used pages are constantly paged in and out. Block addressing enables the processor to map four large blocks of instruction memory and four large blocks of data memory in a way that bypasses the paging mechanism.

A discussion of block addressing is beyond the scope of this chapter. In this subsection, we concentrate on the paging and segmentation mechanisms of the 32-bit PowerPC. The 64-bit scheme is similar.

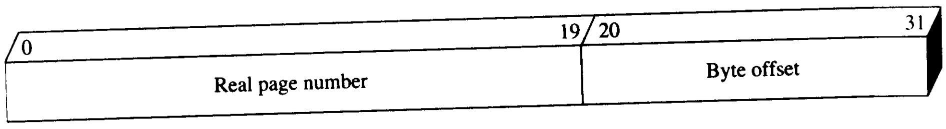
The 32-bit PowerPC makes use of a 32-bit effective address (Figure 8.22a). The address includes a 12-bit byte selector and a 16-bit page identifier. Thus, 2^{12} = 4 KByte pages are used. Up to 2^{16} = 64 K pages per segment are allowed. Four bits of the address are used to designate one of 16 segment registers. The contents of these registers are controlled by the operating system. Each segment register includes access control bits and a 24-bit identifier, so that the 32-bit effective address maps into a 52-bit virtual address (Figure 8.23).



(a) Effective address


0
1
24
25
26
31
V
Virtual segment ID (VSID)
H
API
Real page number
R
C
WIMG
PP
0
19
23
24
25
28
30
31
V = Entry valid bit
H = Hash function identifier
API = Abbreviated page index
R = Referenced bit
C = Changed bit
WIMG = Cache and storage access control bits
PP = Page protection bits
□ = reserved

(b) Page table entry



(c) Real address

Figure 8.22 PowerPC 32-Bit Memory-Management Formats

The PowerPC makes use of a single inverted page table. The virtual address is used to index into the page table in the following manner. First, a hash code is computed as follows:

$$H(0\ldots18) = SID(5\ldots23) \oplus VPN(0\ldots18)$$

The virtual page number in the virtual address is padded on the left (most significant end) with three binary zeros to form a 19-bit number. Then a bit-by-bit exclusive-or is calculated of that number and the 19 right-most bits of the virtual segment ID to form a 19-bit hash code. The table is organized as *n* groups of 8 entries. From 10 to 19 bits of the hash code (depending on the size of the page table) are used to select one of the groups in the table. The memory-management hardware then scans the eight entries of the group to test for a match with the virtual address.

To do the match, each page table entry includes the virtual segment ID and the leftmost 6 bits of the virtual page number, called the abbreviated page index (because at least 10 bits of the 16-bit virtual page number always participate in the hash to select a page table entry group, only an abbreviated form of the virtual page number need be carried in the page table entry to match the virtual address). If there is a match, then the 20-bit real page number from the address is concatenated with the lower 12 bits of the effective address to form the 32-bit physical address to be accessed.

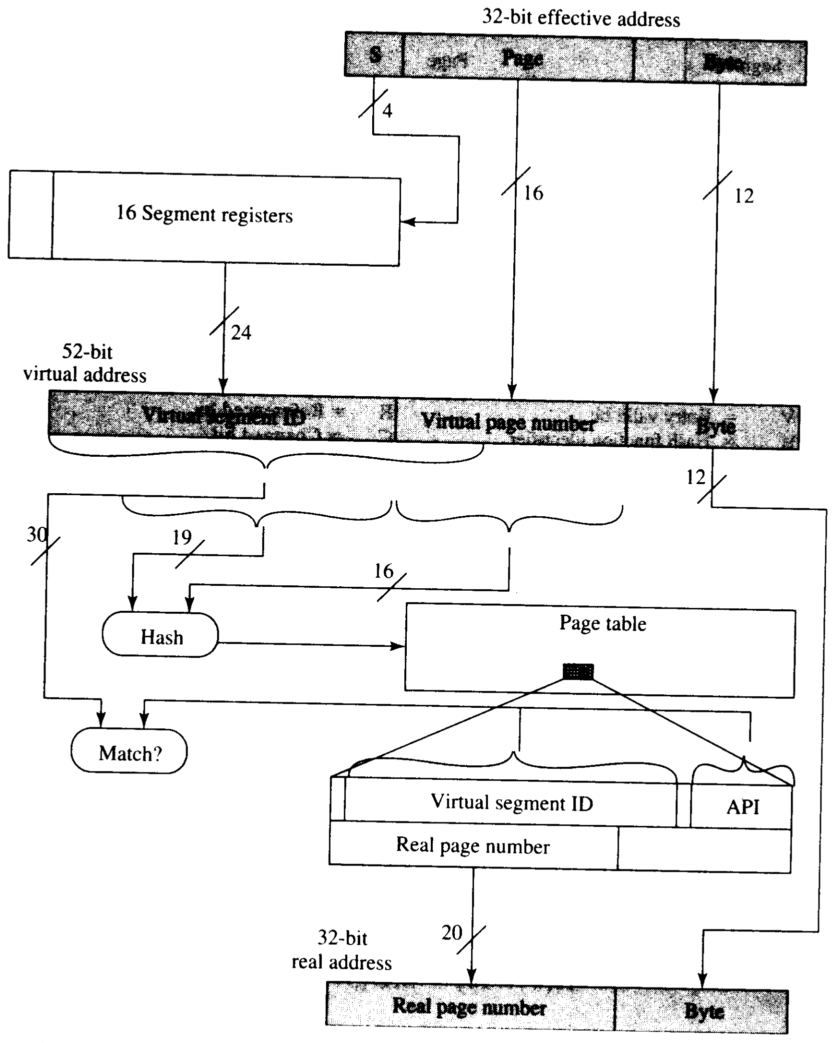


Figure 8.23 PowerPC 32-Bit Address Translation

If there is no match, then the hash code is complemented to produce a new page table index that is in the same relative position at the opposite end of the table. This group is then scanned for a match. If no match is found, a page fault interrupt occurs.

Figure 8.22 shows the logic of the address translation mechanism, and Figure 8.23 shows the formats of the effective address, page table entry, and real address. Finally, Table 8.6 defines the parameters in the page table entry.

The 64-bit memory management scheme is designed to be upwardly compatible with the 32-bit implementation. In essence, all effective addresses, general registers, and branch address registers are extended on the left to 64 bits.

Table 8.6 PowerPC Memory Management Parameters

Segment Table Entry

Effective Segment ID
Indicates one of 64G effective segments; used to determine entry in segment table.

Entry Valid (V) bit
Indicates whether this is a memory or I/O segment

Segment Type (T) bit
Indicates whether this is a memory or I/O segment

Supervisor Key (Ks)
Used with the virtual page number to determine entry in page table.

Page Table Entry

Entry Valid (V) bit
Indicates whether there is valid data in this entry.

Hash Function Identifier (H)
Indicates whether this is a primary or secondary hash entry.

Abbreviated Page Index (API)
Used to match a virtual address uniquely.

Referenced (R) bit
This bit is set to 1 by the processor when a read or write operation to the corresponding page occurs.

Changed (C) bit
This bit is set to 1 by the processor when a write operation to the corresponding page occurs.

WIMG bits
W = 0: use write-back policy; W = 1: use write-through policy.
I = 0: caching not inhibited; I = 1: caching inhibited.
M = 0: not shared memory; M = 1: shared memory.
G = 0: not guarded memory; G = 1: guarded memory.

Page Protection (PP) bits
Access control bits used with K bits from segment register or segment table entry to define access rights.

8.5 RECOMMENDED READING AND WEB SITES

[STAL05] covers the topics of this chapter in detail.

STAL05 Stallings, W. *Operating Systems, Internals and Design Principles, Fifth Edition.* Upper Saddle River, NJ: Prentice Hall, 2005.

Recommended Web Sites:

- **Operating System Resource Center**: A useful collection of documents and papers on a wide range of OS topics

- **ACM Special Interest Group on Operating Systems**: Information on SIGOPS publications and conferences
- **IEEE Technical Committee on Operating Systems and Applications**: Includes an online newsletter and links to other sites
- **Review of Operating Systems**: Comprehensive review of commercial, free, research, and hobby OSs

8.6 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

batch system	**multitasking**	**resident monitor**
demand paging	**nucleus**	**segmentation**
interactive operating system	**operating system (OS)**	**short-term scheduling**
interrupt	**paging**	**swapping**
job control language (JCL)	**page table**	**thrashing**
kernel	**partitioning**	**time-sharing system**
logical address	**physical address**	**translation lookaside buffer (TLB)**
long-term scheduling	**privileged instruction**	**utility**
medium-term scheduling	**process**	**virtual memory**
memory management	**process control block**	
memory protection	**process state**	
multiprogramming	**real memory**	

Review Questions

8.1 What is an operating system?

8.2 List and briefly define the key services provided by an operating system.

8.3 List and briefly define the major types of OS scheduling.

8.4 What is the difference between a process and a program?

8.5 What is the purpose of swapping?

8.6 If a process may be dynamically assigned to different locations in main memory, what is the implication for the addressing mechanism?

8.7 Is it necessary for all of the pages of a process to be in main memory while the process is executing?

8.8 Must the pages of a process in main memory be contiguous?

8.9 Is it necessary for the pages of a process in main memory to be in sequential order?

8.10 What is the purpose of a translation lookaside buffer?

Problems

8.1 Suppose that we have a multiprogrammed computer in which each job has identical characteristics. In one computation period, T, for a job, half the time is spent in I/O and the other half in processor activity. Each job runs for a total of N periods. Assume that a simple round-robin priority is used, and that I/O operations can overlap with processor operation. Define the following quantities:

- Turnaround time = actual time to complete a job
- Throughput = average number of jobs completed per time period T.
- Processor utilization = percentage of time that the processor is active (not waiting)

Compute these quantities for one, two, and four simultaneous jobs, assuming that the period T is distributed in each of the following ways:

a. I/O first half, processor second half

b. I/O first and fourth quarters, processor second and third quarters

8.2 An I/O-bound program is one that, if run alone, would spend more time waiting for I/O than using the processor. A processor-bound program is the opposite. Suppose a short-term scheduling algorithm favors those programs that have used little processor time in the recent past. Explain why this algorithm favors I/O-bound programs and yet does not permanently deny processor time to processor-bound programs.

8.3 A program computes the row sums

$$C_i = \sum_{j=1}^{n} a_{ij}$$

of an array A that is 100 by 100. Assume that the computer uses demand paging with a page size of 1000 words, and that the amount of main memory allotted for data is five page frames. Is there any difference in the page fault rate if A were stored in virtual memory by rows or columns? Explain.

8.4 Consider a fixed partitioning scheme with equal-size partitions of 2^{16} bytes and a total main memory size of 2^{24} bytes. A process table is maintained that includes a pointer to a partition for each resident process. How many bits are required for the pointer?

8.5 Consider a dynamic partitioning scheme. Show that, on average, the memory contains half as many holes as segments.

8.6 Suppose the page table for the process currently executing on the processor looks like the following. All numbers are decimal, everything is numbered starting from zero, and all addresses are memory byte addresses. The page size is 1024 bytes.

Virtual page number	Valid bit	Reference bit	Modify bit	Page frame number
0	1	1	0	4
1	1	1	1	7
2	0	0	0	—
3	1	0	0	2
4	0	0	0	—
5	1	0	1	0

a. Describe exactly how, in general, a virtual address generated by the CPU is translated into a physical main memory address.

b. What physical address, if any, would each of the following virtual addresses correspond to? (Do not try to handle any page faults, if any.)

(i) 1052

(ii) 2221

(iii) 5499

8.7 Give reasons that the page size in a virtual memory system should be neither very small nor very large.

8.8 A process references five pages, A, B, C, D, and E, in the following order:

A; B; C; D; A; B; E; A; B; C; D; E

Assume that the replacement algorithm is first-in-first-out and find the number of page transfers during this sequence of references starting with an empty main memory with three page frames. Repeat for four page frames.

8.9 The following sequence of virtual page numbers is encountered in the course of execution on a computer with virtual memory:

3 4 2 6 4 7 1 3 2 6 3 5 1 2 3

Assume that a least recently used page replacement policy is adopted. Plot a graph of page hit ratio (fraction of page references in which the page is in main memory) as a function of main-memory page capacity n for $1 \leq n \leq 8$. Assume that main memory is initially empty.

8.10 In the VAX computer, user page tables are located at virtual addresses in the system space. What is the advantage of having user page tables in virtual rather than main memory? What is the disadvantage?

8.11 Suppose the program statement

```
for (i = 1; i <= n; I++)
    a[i] = b[i] + c[i];
```

is executed in a memory with page size of 1000 words. Let $n = 1000$. Using a machine that has a full range of register-to-register instructions and employs index registers, write a hypothetical program to implement the foregoing statement. Then show the sequence of page references during execution.

8.12 The IBM System/370 architecture uses a two-level memory structure and refers to the two levels as segments and pages, although the segmentation approach lacks many of the features described earlier in this chapter. For the basic 370 architecture, the page size may be either 2 Kbytes or 4 Kbytes, and the segment size is fixed at either 64 Kbytes or 1 Mbyte. For the 370/XA and 370/ESA architectures, the page size is 4 Kbytes and the segment size is 1 Mbyte. Which advantages of segmentation does this scheme lack? What is the benefit of segmentation for the 370?

8.13 Consider a computer system with both segmentation and paging. When a segment is in memory, some words are wasted on the last page. In addition, for a segment size s and a page size p, there are s/p page table entries. The smaller the page size, the less waste in the last page of the segment, but the larger the page table. What page size minimizes the total overhead?

8.14 A computer has a cache, main memory, and a disk used for virtual memory. If a referenced word is in the cache, 20 ns are required to access it. If it is in main memory but not in the cache, 60 ns are needed to load it into the cache, and then the reference is started again. If the word is not in main memory, 12 ms are required to fetch the word from disk, followed by 60 ns to copy it to the cache, and then the reference is started again. The cache hit ratio is 0.9 and the main-memory hit ratio is 0.6. What is the average time in ns required to access a referenced word on this system?

8.15 Assume a task is divided into four equal-sized segments and that the system builds an eight-entry page descriptor table for each segment. Thus, the system has a combination of segmentation and paging. Assume also that the page size is 2 KBytes.

a. What is the maximum size of each segment?

b. What is the maximum logical address space for the task?

c. Assume that an element in physical location 00021ABC is accessed by this task. What is the format of the logical address that the task generates for it? What is the maximum physical address space for the system?

8.16 Assume a microprocessor capable of accessing up to 2^{32} bytes of physical main memory. It implements one segmented logical address space of maximum size 2^{31} bytes. Each instruction contains the whole two-part address. External memory management units (MMUs) are used, whose management scheme assigns contiguous blocks of physical memory of fixed size 2^{22} bytes to segments. The starting physical address of a segment is always divisible by 1024. Show the detailed interconnection of the external mapping mechanism that converts logical addresses to physical addresses using the appropriate number of MMUs, and show the detailed internal structure of an MMU (assuming that each MMU contains a 128-entry directly mapped segment descriptor cache) and how each MMU is selected.

8.17 Consider a paged logical address space (composed of 32 pages of 2 KBytes each) mapped into a 1-MByte physical memory space.

a. What is the format of the processor's logical address?

b. What is the length and width of the page table (disregarding the "access rights" bits)?

c. What is the effect on the page table if the physical memory space is reduced by half?

8.18 In IBM's mainframe operating system, OS/390, one of the major modules in the kernel is the System Resource Manager (SRM). This module is responsible for the allocation of resources among address spaces (processes). The SRM gives OS/390 a degree of sophistication unique among operating systems. No other mainframe operating system, and certainly no other type of operating system, can match the functions performed by SRM. The concept of resource includes processor, real memory, and I/O channels. SRM accumulates statistics pertaining to utilization of processor, channel, and various key data structures. Its purpose is to provide optimum performance based on performance monitoring and analysis. The installation sets forth various performance objectives, and these serve as guidance to the SRM, which dynamically modifies installation and job performance characteristics based on system utilization. In turn, the SRM provides reports that enable the trained operator to refine the configuration and parameter settings to improve user service.

This problem concerns one example of SRM activity. Real memory is divided into equal-sized blocks called frames, of which there may be many thousands. Each frame can hold a block of virtual memory referred to as a page. SRM receives control approximately 20 times per second and inspects each and every page frame. If the page has not been referenced or changed, a counter is incremented by 1. Over time, SRM averages these numbers to determine the average number of seconds that a page frame in the system goes untouched. What might be the purpose of this and what action might SRM take?

PART THREE

The Central Processing Unit

ISSUES FOR PART THREE

Up to this point, we have viewed the CPU essentially as a "black box" and have considered its interaction with I/O and memory. Part Three examines the internal structure and function of the CPU. The CPU consists of registers, the arithmetic and logic unit, the instruction execution unit, a control unit, and the interconnections among these components. Architectural issues, such as instruction set design and data types, are covered. The part also looks at organizational issues, such as pipelining.

ROAD MAP FOR PART THREE

Chapter 9 Computer Arithmetic

Chapter 9 examines the functionality of the ALU and focuses on the representation of numbers and techniques for implementing arithmetic operations. Processors typically support two types of arithmetic: integer, or fixed point, and floating point. For both cases, the chapter first examines the representation of numbers and then discusses arithmetic operations. The important IEEE 754 floating-point standard is examined in detail.

Chapter 10 Instruction Sets: Characteristics and Functions

From a programmer's point of view, the best way to understand the operation of a processor is to learn the machine instruction set that it executes. The complex topic of instruction set design occupies Chapters 10 and 11. Chapter 10 focuses on the functional aspects of instruction set design. The chapter examines the types of functions that are specified by computer instructions and then looks specifically at the types of operands (which specify the data to be operated on) and the types of operations (which specify the operations to be performed) commonly found in instruction sets. Then the relationship of processor instructions to assembly language is briefly explained.

Chapter 11 Instruction Sets: Addressing Modes and Formats

Whereas Chapter 10 can be viewed as dealing with the semantics of instruction sets, Chapter 11 is more concerned with the syntax of instruction sets. Specifically, Chapter 11 looks at the way in which memory addresses are specified and at the overall format of computer instructions.

Chapter 12 CPU Structure and Function

Chapter 12 is devoted to a discussion of the internal structure and function of the processor. The chapter describes the use of registers as the CPU's internal memory and then pulls together all of the material covered so far to provide an overview of CPU structure and function. The overall organization (ALU, register file, control unit) is reviewed. Then the organization of the register file is discussed. The remainder of the chapter describes the functioning of the processor in executing machine instructions. The instruction cycle is examined to show the function and interrelationship of fetch, indirect, execute, and interrupt cycles. Finally, the use of pipelining to improve performance is explored in depth.

Chapter 13 Reduced Instruction Set Computers

The remainder of Part Three looks in more detail at the key trends in CPU design. Chapter 13 describes the approach associated with the concept of a reduced instruction set computer (RISC), which is one of the most significant innovations in computer organization and architecture in recent years. RISC architecture is a dramatic departure from the historical trend in processor architecture. An analysis of this approach brings into focus many of the important issues in computer organization and architecture. The chapter examines the motivation for the use of RISC design and then looks at the details of RISC instruction set design and RISC CPU architecture and compares RISC with the complex instruction set computer (CISC) approach.

Chapter 14 Instruction-Level Parallelism and Superscalar Processors

Chapter 14 examines an even more recent and equally important design innovation: the superscalar processor. Although superscalar technology can be used on any processor, it is especially well suited to a RISC architecture. The chapter also looks at the general issue of instruction-level parallelism.

KEY POINTS

- ◆ The two principal concerns for computer arithmetic are the way in which numbers are represented (the binary format) and the algorithms used for the basic arithmetic operations (add, subtract, multiply, divide). These two considerations apply both to integer and floating-point arithmetic.
- ◆ Floating-point numbers are expressed as a number (significand) multiplied by a constant (base) raised to some integer power (exponent). Floating-point numbers can be used to represent very large and very small numbers.
- ◆ Most processors implement the IEEE 754 standard for floating-point representation and floating-point arithmetic. IEEE 754 defines both a 32-bit and a 64-bit format.

We begin our examination of the processor with an overview of the arithmetic and logic unit (ALU). The chapter then focuses on the most complex aspect of the ALU, computer arithmetic. The logic functions that are part of the ALU are described in Chapter 10, and implementations of simple logic and arithmetic functions in digital logic are described in Appendix B of this book.

Computer arithmetic is commonly performed on two very different types of numbers: integer and floating point. In both cases, the representation chosen is a crucial design issue and is treated first, followed by a discussion of arithmetic operations.

This chapter includes a number of examples, each of which is highlighted in a shaded box.

9.1 THE ARITHMETIC AND LOGIC UNIT

The ALU is that part of the computer that actually performs arithmetic and logical operations on data. All of the other elements of the computer system—control unit, registers, memory, I/O—are there mainly to bring data into the ALU for it to process and then to take the results back out. We have, in a sense, reached the core or essence of a computer when we consider the ALU.

An ALU and, indeed, all electronic components in the computer are based on the use of simple digital logic devices that can store binary digits and perform simple Boolean logic operations. For the interested reader, Appendix B explores digital logic implementation.

Figure 9.1 indicates, in general terms, how the ALU is interconnected with the rest of the processor. Data are presented to the ALU in registers, and the results of an operation are stored in registers. These registers are temporary storage locations within the processor that are connected by signal paths to the ALU (e.g., see Figure 2.3). The ALU may also set flags as the result of an operation.

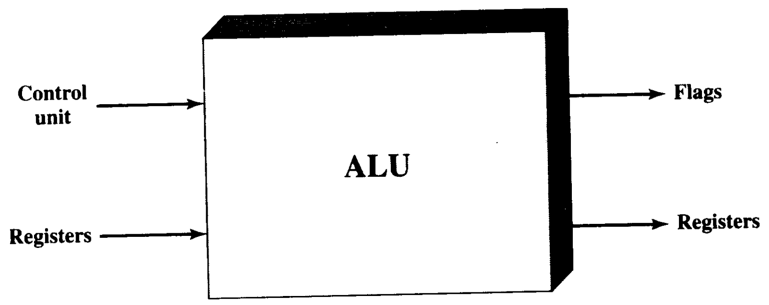


Figure 9.1 ALU Inputs and Outputs

For example, an overflow flag is set to 1 if the result of a computation exceeds the length of the register into which it is to be stored. The flag values are also stored in registers within the processor. The control unit provides signals that control the operation of the ALU and the movement of the data into and out of the ALU.

9.2 INTEGER REPRESENTATION

In the binary number system,[1] arbitrary numbers can be represented with just the digits zero and one, the minus sign, and the period, or **radix point**.

$$-1101.0101_2 = -13.3125_{10}$$

For purposes of computer storage and processing, however, we do not have the benefit of minus signs and periods. Only binary digits (0 and 1) may be used to represent numbers. If we are limited to nonnegative integers, the representation is straightforward.

An 8-bit word can represent the numbers from 0 to 255, including

$$\begin{aligned} 00000000 &= 0 \\ 00000001 &= 1 \\ 00101001 &= 41 \\ 10000000 &= 128 \\ 11111111 &= 255 \end{aligned}$$

In general, if an n-bit sequence of binary digits $a_{n-1}a_{n-2}\dots a_1a_0$ is interpreted as an unsigned integer A, its value is

$$A = \sum_{i=0}^{n-1} 2^i a_i$$

[1]See Appendix A for a basic refresher on number systems (decimal, binary, hexadecimal).

Sign-Magnitude Representation

There are several alternative conventions used to represent negative as well as positive integers, all of which involve treating the most significant (leftmost) bit in the word as a sign bit. If the sign bit is 0, the number is positive; if the sign bit is 1, the number is negative.

The simplest form of representation that employs a sign bit is the sign-magnitude representation. In an n-bit word, the rightmost $n - 1$ bits hold the magnitude of the integer.

$$\begin{aligned} +18 &= 00010010 \\ -18 &= 10010010 \quad \text{(sign magnitude)} \end{aligned}$$

The general case can be expressed as follows:

Sign Magnitude

$$A = \begin{cases} \sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 0 \\ -\sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 1 \end{cases} \qquad \textbf{(9.1)}$$

There are several drawbacks to sign-magnitude representation. One is that addition and subtraction require a consideration of both the signs of the numbers and their relative magnitudes to carry out the required operation. This should become clear in the discussion in Section 9.3. Another drawback is that there are two representations of 0:

$$\begin{aligned} +0_{10} &= 00000000 \\ -0_{10} &= 10000000 \quad \text{(sign magnitude)} \end{aligned}$$

This is inconvenient because it is slightly more difficult to test for 0 (an operation performed frequently on computers) than if there were a single representation.

Because of these drawbacks, sign-magnitude representation is rarely used in implementing the integer portion of the ALU. Instead, the most common scheme is twos complement representation.[2]

Twos Complement Representation

Like sign magnitude, twos complement representation uses the most significant bit as a sign bit, making it easy to test whether an integer is positive or negative. It differs from the use of the sign-magnitude representation in the way that the other bits are interpreted. Table 9.1 highlights key characteristics of twos complement representation and arithmetic, which are elaborated in this section and the next.

Most treatments of twos complement representation focus on the rules for producing negative numbers, with no formal proof that the scheme "works." Instead, our presentation of twos complement integers in this section and in Section 9.3 is based on [DATT93], which suggests that twos complement representation is best understood

[2]In the literature, the terms *two's complement* or *2's complement* are often used. Here we follow the practice used in standards documents and omit the apostrophe (e.g., IEEE Std 100-1992, *The New IEEE Standard Dictionary of Electrical and Electronics Terms*).

Table 9.1 Characteristics of Twos Complement Representation and Arithmetic

Range	-2^{n-1} **through** $2^{n-1} - 1$
Number of Representations of Zero	**One**
Negation	**Take the Boolean complement of each bit of the corresponding positive number, then add 1 to the resulting bit pattern viewed as an unsigned integer.**
Expansion of Bit Length	**Add additional bit positions to the left and fill in with the value of the original sign bit.**
Overflow Rule	**If two numbers with the same sign (both positive or both negative) are added, then overflow occurs if and only if the result has the opposite sign.**
Subtraction Rule	**To subtract *B* from *A*, take the twos complement of *B* and add it to *A*.**

by defining it in terms of a weighted sum of bits, as we did previously for unsigned and sign-magnitude representations. The advantage of this treatment is that it does not leave any lingering doubt that the rules for arithmetic operations in twos complement notation may not work for some special cases.

Consider an n-bit integer, A, in twos complement representation. If A is positive, then the sign bit, a_{n-1}, is zero. The remaining bits represent the magnitude of the number in the same fashion as for sign magnitude:

$$A = \sum_{i=0}^{n-2} 2^i a_i \qquad \text{for } A \geq 0$$

The number zero is identified as positive and therefore has a 0 sign bit and a magnitude of all 0s. We can see that the range of positive integers that may be represented is from 0 (all of the magnitude bits are 0) through $2^{n-1} - 1$ (all of the magnitude bits are 1). Any larger number would require more bits.

Now, for a negative number A ($A < 0$), the sign bit, a_{n-1}, is one. The remaining $n - 1$ bits can take on any one of 2^{n-1} values. Therefore, the range of negative integers that can be represented is from -1 to -2^{n-1}. We would like to assign the bit values to negative integers in such a way that arithmetic can be handled in a straightforward fashion, similar to unsigned integer arithmetic. In unsigned integer representation, to compute the value of an integer from the bit representation, the weight of the most significant bit is $+2^{n-1}$. For a representation with a sign bit, it turns out that the desired arithmetic properties are achieved, as we will see in Section 9.3, if the weight of the most significant bit is -2^{n-1}. This is the convention used in twos complement representation, yielding the following expression for negative numbers:

$$\textbf{Twos Complement} \quad A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i \qquad \textbf{(9.2)}$$

In the case of positive integers, $a_{n-1} = 0$, so the term $-2^{n-1}a_{n-1} = 0$. Therefore, Equation (9.2) defines the twos complement representation for both positive and negative numbers.

Table 9.2 Alternative Representations for 4-Bit Integers

Decimal Representation	Sign-Magnitude Representation	Twos Complement Representation	Biased Representation
+8	—	—	1111
+7	0111	0111	1110
+6	0110	0110	1101
+5	0101	0101	1100
+4	0100	0100	1011
+3	0011	0011	1010
+2	0010	0010	1001
+1	0001	0001	1000
+0	0000	0000	0111
−0	1000	—	—
−1	1001	1111	0110
−2	1010	1110	0101
−3	1011	1101	0100
−4	1100	1100	0011
−5	1101	1011	0010
−6	1110	1010	0001
−7	1111	1001	0000
−8	—	1000	—

Table 9.2 compares the sign-magnitude and twos complement representations for 4-bit integers. Although twos complement is an awkward representation from the human point of view, we will see that it facilitates the most important arithmetic operations, addition and subtraction. For this reason, it is almost universally used as the processor representation for integers.

A useful illustration of the nature of twos complement representation is a value box, in which the value on the far right in the box is 1 (2^0) and each succeeding position to the left is double in value, until the leftmost position, which is negated. As you can see in Figure 9.2a, the most negative twos complement number that can be represented is -2^{n-1}; if any of the bits other than the sign bit is one, it adds a positive amount to the number. Also, it is clear that a negative number must have a 1 at its leftmost position and a positive number must have a 0 in that position. Thus, the largest positive number is a 0 followed by all 1s, which equals $2^{n-1} - 1$.

The rest of Figure 9.2 illustrates the use of the value box to convert from twos complement to decimal and from decimal to twos complement.

Converting between Different Bit Lengths

It is sometimes desirable to take an n-bit integer and store it in m bits, where $m > n$. In sign-magnitude notation, this is easily accomplished: simply move the sign bit to the new leftmost position and fill in with zeros.

−128	64	32	16	8	4	2	1

(a) An eight-position twos complement value box

−128	64	32	16	8	4	2	1
1	0	0	0	0	0	1	1

−128 +2 +1 = −125

(b) Convert binary 10000011 to decimal

−128	64	32	16	8	4	2	1
1	0	0	0	1	0	0	0

−120 = −128 +8

(c) Convert decimal −120 to binary

Figure 9.2 Use of a Value Box for Conversion between Twos Complement Binary and Decimal

+18	=	00010010	**(sign magnitude, 8 bits)**
+18	=	0000000000010010	**(sign magnitude, 16 bits)**
−18	=	10010010	**(sign magnitude, 8 bits)**
−18	=	1000000000010010	**(sign magnitude, 16 bits)**

This procedure will not work for twos complement negative integers. Using the same example,

+18	=	00010010	**(twos complement, 8 bits)**
+18	=	0000000000010010	**(twos complement, 16 bits)**
−18	=	11101110	**(twos complement, 8 bits)**
−32,658	=	1000000001101110	**(twos complement, 16 bits)**

The next to last line is easily seen using the value box of Figure 9.2. The last line can be verified using Equation (9.2) or a 16-bit value box.

Instead, the rule for twos complement integers is to move the sign bit to the new leftmost position and fill in with copies of the sign bit. For positive numbers, fill in with zeros, and for negative numbers, fill in with ones. This is called sign extension.

−18	=	11101110	**(twos complement, 8 bits)**
−18	=	1111111111101110	**(twos complement, 16 bits)**

To see why this rule works, let us again consider an n-bit sequence of binary digits $a_{n-1}a_{n-2} \cdots a_1a_0$ interpreted as a twos complement integer A, so that its value is

$$A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

If A is a positive number, the rule clearly works. Now, if A is negative and we want to construct an m-bit representation, with $m > n$. Then

$$A = -2^{m-1}a_{m-1} + \sum_{i=0}^{m-2} 2^i a_i$$

The two values must be equal:

$$\begin{aligned}
-2^{m-1} + \sum_{i=0}^{m-2} 2^i a_i &= -2^{n-1} + \sum_{i=0}^{n-2} 2^i a_i \\
-2^{m-1} + \sum_{i=n-1}^{m-2} 2^i a_i &= -2^{n-1} \\
2^{n-1} + \sum_{i=n-1}^{m-2} 2^i a_i &= 2^{m-1} \\
1 + \sum_{i=0}^{n-2} 2^i + \sum_{i=n-1}^{m-2} 2^i a_i &= 1 + \sum_{i=0}^{m-2} 2^i \\
\sum_{i=n-1}^{m-2} 2^i a_i &= \sum_{i=n-1}^{m-2} 2^i \\
\Rightarrow \quad a_{m-2} = \cdots = a_{n-2} &= a_{n-1} = 1
\end{aligned}$$

In going from the first to the second equation, we require that the least significant $n - 1$ bits do not change between the two representations. Then we get to the next to last equation, which is only true if all of the bits in positions $n - 1$ through $m - 2$ are 1. Therefore, the sign-extension rule works.

Fixed-Point Representation

Finally, we mention that the representations discussed in this section are sometimes referred to as fixed point. This is because the radix point (binary point) is fixed and assumed to be to the right of the rightmost digit. The programmer can use the same representation for binary fractions by scaling the numbers so that the binary point is implicitly positioned at some other location.

9.3 INTEGER ARITHMETIC

This section examines common arithmetic functions on numbers in twos complement representation.

Negation

In sign-magnitude representation, the rule for forming the negation of an integer is simple: Invert the sign bit. In twos complement notation, the negation of an integer can be formed with the following rules:

1. Take the Boolean complement of each bit of the integer (including the sign bit). That is, set each 1 to 0 and each 0 to 1.
2. Treating the result as an unsigned binary integer, add 1.

This two-step process is referred to as the **twos complement operation**, or the taking of the twos complement of an integer.

```
                 +18  =  00010010 (twos complement)
bitwise complement    =  11101101
                         +      1
                         11101110 = -18
```

As expected, the negative of the negative of that number is itself:

```
                 -18  =  11101110 (twos complement)
bitwise complement    =  00010001
                         +      1
                         00010010 = +18
```

We can demonstrate the validity of the operation just described using the definition of the twos complement representation in Equation (9.2). Again, interpret an n-bit sequence of binary digits $a_{n-1}a_{n-2}\dots a_1a_0$ as a twos complement integer A, so that its value is

$$A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

Now form the bitwise complement, $\overline{a_{n-1}}\,\overline{a_{n-2}}\dots\overline{a_0}$, and, treating this is an unsigned integer, add 1. Finally, interpret the resulting n-bit sequence of binary digits as a twos complement integer B, so that its value is

$$B = -2^{n-1}\overline{a_{n-1}} + 1 + \sum_{i=0}^{n-2} 2^i \overline{a_i}$$

Now, we want $A = -B$, which means $A + B = 0$. This is easily shown to be true:

$$\begin{aligned}
A + B &= -(a_{n-1} + \overline{a_{n-1}})2^{n-1} + 1 + \left(\sum_{i=0}^{n-2} 2^i(a_i + \overline{a_i})\right) \\
&= -2^{n-1} + 1 + \left(\sum_{i=0}^{n-2} 2^i\right) \\
&= -2^{n-1} + 1 + (2^{n-1} - 1) \\
&= -2^{n-1} + 2^{n-1} = 0
\end{aligned}$$

The preceding derivation assumes that we can first treat the bitwise complement of A as an unsigned integer for the purpose of adding 1, and then treat the result as a twos complement integer. There are two special cases to consider. First, consider $A = 0$. In that case, for an 8-bit representation:

```
                  0  =  00000000 (twos complement)
bitwise complement   =  11111111
                        +      1
                        100000000 = 0
```

There is *carry* out of the most significant bit position, which is ignored. The result is that the negation of 0 is 0, as it should be.

The second special case is more of a problem. If we take the negation of the bit pattern of 1 followed by $n - 1$ zeros, we get back the same number. For example, for 8-bit words,

```
               -128  =  10000000 (twos complement)
bitwise complement   =  01111111
                        +      1
                        10000000 = -128
```

Some such anomaly is unavoidable. The number of different bit patterns in an n-bit word is 2^n, which is an even number. We wish to represent positive and negative integers and 0. If an equal number of positive and negative integers are represented (sign magnitude), then there are two representations for 0. If there is only one representation of 0 (twos complement), then there must be an unequal number of negative and positive numbers represented. In the case of twos complement, for an n-bit length, there is a representation for -2^{n-1} but not for $+2^{n-1}$.

Addition and Subtraction

Addition in twos complement is illustrated in Figure 9.3. Addition proceeds as if the two numbers were unsigned integers. The first four examples illustrate successful operations. If the result of the operation is positive, we get a positive number in twos complement

```
  1001 = -7                 1100 = -4
 +0101 =  5                +0100 =  4
  1110 = -2                10000 =  0
 (a) (-7) + (+5)           (b) (-4) + (+4)

  0011 = 3                  1100 = -4
 +0100 = 4                 +1111 = -1
  0111 = 7                 11011 = -5
 (c) (+3) + (+4)           (d) (-4) + (-1)

  0101 = 5                  1001 = -7
 +0100 = 4                 +1010 = -6
  1001 = Overflow          10011 = Overflow
 (e) (+5) + (+4)           (f) (-7) + (-6)
```

Figure 9.3 Addition of Numbers in Twos Complement Respresentation

form, which is the same as in unsigned-integer form. If the result of the operation is negative, we get a negative number in twos complement form. Note that, in some instances, there is a carry bit beyond the end of the word (indicated by shading), which is ignored.

On any addition, the result may be larger than can be held in the word size being used. This condition is called **overflow**. When overflow occurs, the ALU must signal this fact so that no attempt is made to use the result. To detect overflow, the following rule is observed:

OVERFLOW RULE: If two numbers are added, and they are both positive or both negative, then overflow occurs if and only if the result has the opposite sign.

Figures 9.3e and f show examples of overflow. Note that overflow can occur whether or not there is a carry.

Subtraction is easily handled with the following rule:

SUBTRACTION RULE: To subtract one number (subtrahend) from another (minuend), take the twos complement (negation) of the subtrahend and add it to the minuend.

Thus, subtraction is achieved using addition, as illustrated in Figure 9.4. The last two examples demonstrate that the overflow rule still applies.

Some insight into twos complement addition and subtraction can be gained by looking at a geometric depiction [BENH92], as shown in Figure 9.5. The circle

0010 = 2 +1001 = -7 1011 = -5 (a) M = 2 = 0010 S = 7 = 0111 -S = 1001	0101 = 5 +1110 = -2 ■0011 = 3 (b) M = 5 = 0101 S = 2 = 0010 -S = 1110
1011 = -5 +1110 = -2 ■1001 = -7 (c) M =-5 = 1011 S = 2 = 0010 -S = 1110	0101 = 5 +0010 = 2 0111 = 7 (d) M = 5 = 0101 S =-2 = 1110 -S = 0010
0111 = 7 +0111 = 7 1110 = Overflow (e) M = 7 = 0111 S = -7 = 1001 -S = 0111	1010 = -6 +1100 = -4 ■0110 = Overflow (f) M = -6 = 1010 S = 4 = 0100 -S = 1100

Figure 9.4 Subtraction of Numbers in Twos Complement Representation (M − S)

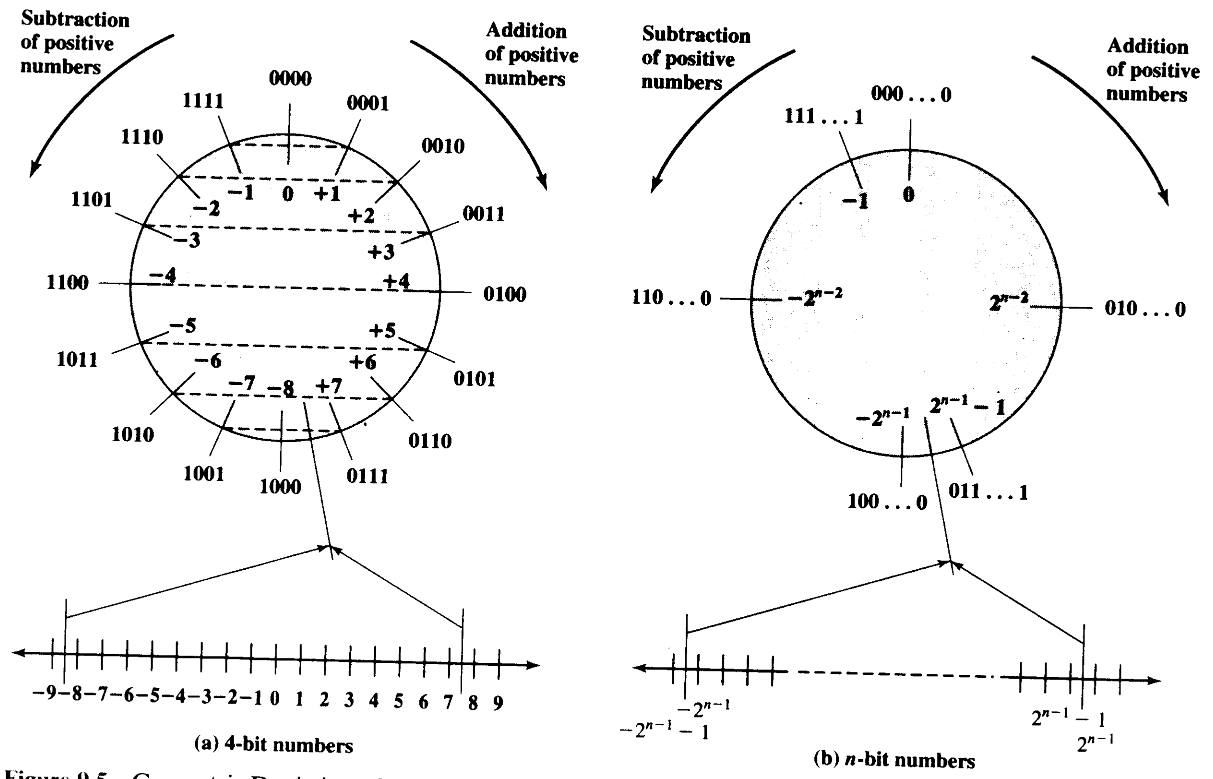


Figure 9.5 Geometric Depiction of Twos Complement Integers