CHAPTER 1

Introduction

1.1 Purpose of This Book

This book is intended for use as a companion text for advanced undergraduate-level or introductory graduate-level courses in computer architecture. Its primary intended audience is students and faculty involved in computer architecture courses who are interested in additional explanations, practice problems, and examples to use in increasing their understanding of the material or in preparing assignments.

1.2 Background Assumed

This book assumes that the reader has a background similar to that of college sophomores or juniors in electrical engineering or computer science programs who have not yet had a course on computer organization or computer architecture. Basic familiarity with computer operation and terminology is assumed, as is some familiarity with programming in high-level languages.

1.3 Material Covered

This book covers a slightly wider range of topics than most one-term computer architecture courses in order to increase its utility. Readers may find the additional material useful as review or as an introduction to more advanced topics. The book begins with a discussion of data representation and computer arithmetic, followed by chapters on computer organization and programming models. Chapter 5 begins a three-chapter discussion of processor design, including pipelining and instruction-level parallelism. This is followed by three chapters on memory systems, including

coverage of virtual memory and caches. The final two chapters discuss I/O and provide an introduction to multiprocessors.

1.4 Chapter Objectives

The goal of this chapter is to prepare the reader for the material in later chapters by discussing the basic technologies that drive computer performance and the techniques used to measure and discuss performance. After reading this chapter and completing the exercises, a student should

1. Understand and be able to discuss the historical rates of improvement in transistor density, circuit performance, and overall system performance.
2. Understand common methods of evaluating computer performance.
3. Be able to calculate how changes to one part of a computer system affect overall performance.

1.5 Technological Trends

Since the early 1980s, computer performance has been driven by improvements in the capabilities of the integrated circuits used to implement microprocessors, memory chips, and other computer components. Over time, integrated circuits improve in *density* (how many transistors and wires can be placed in a fixed area on a silicon chip), *speed* (how quickly basic logic gates and memory devices operate), and *area* (the physical size of the largest integrated circuit that can be fabricated).

The tremendous growth in computer performance over the last two decades has been driven by the fact that chip speed and density improve geometrically rather than linearly, meaning that the increase in performance from one year to the next has been a relatively constant fraction of the previous year's performance, rather than a constant absolute value. On average, the number of transistors that can be fabricated on a silicon chip increases by about 50 percent per year, and transistor speed increases such that the delay of a basic logic gate (AND, OR, etc.) decreases by 13 percent per year. The observation that computer performance improves geometrically, not linearly, is often referred to as *Moore's Law*.

EXAMPLE

The amount of data that can be stored on a dynamic RAM (DRAM) memory chip has quadrupled every three years since the late 1970s, an annual growth rate of 60 percent.

From the late 1970s until the late 1980s, microprocessor performance was mainly driven by improvements in fabrication technology and improved at a rate of 35 percent per year. Since then, the rate of improvement has actually increased, to over 50 percent per year, although the rate of progress in semiconductor fabrication has

remained relatively constant. The increase in the rate of performance improvement has been due to improvements in computer architecture and organization—computer architects have been able to take advantage of the increasing density of integrated circuits to add features to microprocessors and memory systems that improve performance over and above the improvements in speed of the underlying transistors.

1.6 Measuring Performance

In this chapter, we have discussed how computer performance has improved over time, without giving a formal definition of what performance is. In part, this is because *performance* is a very vague term when used in the context of computer systems. Generally, performance describes how quickly a given system can execute a program or programs. Systems that execute programs in less time are said to have higher performance.

The best measure of computer performance is the execution time of the program or programs that the user wants to execute, but it is generally impractical to test all of the programs that will be run on a given system before deciding which computer to purchase or when making design decisions. Instead, computer architects have come up with a variety of metrics to describe computer performance, some of which will be discussed in this chapter. Architects have also devised a number of metrics for the performance of individual computer subsystems, which will be discussed in the chapters that cover those subsystems.

Keep in mind that many factors other than performance may influence design or purchase decisions. Ease of programming is an important consideration, because the time and expense required to develop needed programs may be more significant than the difference in execution times of the programs once they have been developed. Also important is the issue of compatibility; most programs are sold as binary images that will only run on a particular family of processors. If the program you need won't run on a given system, it doesn't matter how quickly the system executes other programs.

1.6.1 MIPS

An early measure of computer performance was the rate at which a given machine executed instructions. This is calculated by dividing the number of instructions executed in running a program by the time required to run the program and is typically expressed in *millions of instructions per second* (MIPS). MIPS has fallen out of use as a measure of performance, mainly because it does not account for the fact that different systems often require different numbers of instructions to implement a given program. A computer's MIPS rating does not tell you anything about how many instructions it requires to perform a given task, making it less useful than other metrics for comparing the performance of different systems.

1.6.2 CPI/IPC

Another metric used to describe computer performance is the number of clock cycles required to execute each instruction, known as *cycles per instruction*, or CPI. The CPI of a given program on a given system is calculated by dividing the number of clock cycles required to execute the program by the number of instructions executed in running the program. For systems that can execute more than one instruction per cycle, the number of *instructions executed per cycle*, or IPC, is often used instead of CPI. IPC is calculated by dividing the number of instructions executed in running a program by the number of clock cycles required to execute the program, and is the reciprocal of CPI. These two metrics give the same information, and the choice of which one to use is generally made based on which of the values is greater than the number 1. When using IPC and CPI to compare systems, it is important to remember that high IPC values indicate that the reference program took fewer cycles to execute than low IPC values, while high CPI values indicate that more cycles were required than low CPI values. Thus, a large IPC tends to indicate good performance, while a large CPI indicates poor performance.

EXAMPLE

A given program consists of a 100-instruction loop that is executed 42 times. If it takes 16,000 cycles to execute the program on a given system, what are that system's CPI and IPC values for the program?

Solution

The 100-instruction loop is executed 42 times, so the total number of instructions executed is $100 \times 42 = 4200$. It takes 16,000 cycles to execute the program, so the CPI is $16{,}000/4200 = 3.81$. To compute the IPC, we divide 4200 instructions by 16,000 cycles, getting an IPC of 0.26.

In general, IPC and CPI are even less useful measures of actual system performance than MIPS, because they do not contain any information about a system's clock rate or how many instructions the system requires to perform a task. If you know a system's MIPS rating on a given program, you can multiply it by the number of instructions executed in running the program to determine how long the program took to complete. If you know a system's CPI on a given program, you can multiply it by the number of instructions in the program to get the number of cycles it took to complete the program, but you have to know the number of cycles per second (the system's clock rate) to convert that into the amount of time required to execute the program.

As a result, CPI and IPC are rarely used to compare actual computer systems. However, they are very common metrics in computer architecture research, because most computer architecture research is done in simulation, using programs that simulate a particular architecture to estimate how many cycles a given program will take to execute on that architecture. These simulators are generally unable to predict

Scanned with CamScanner

the cycle time of the systems that they simulate, so CPI/IPC is often the best available estimate of performance.

1.6.3 BENCHMARK SUITES

Both MIPS and CPI/IPC have significant limitations as measures of computer performance, as we have discussed. ***Benchmark suites*** are a third measure of computer performance and were developed to address the limitations of MIPS and CPI/IPC.

A benchmark suite consists of a set of programs that are believed to be typical of the programs that will be run on the system. A system's score on the benchmark suite is based on how long it takes the system to execute all of the programs in the suite. Many different benchmark suites exist that generate estimates of a system's performance on different types of applications.

One of the best-known benchmark suites is the SPEC suite, produced by the Standard Performance Evaluation Corporation. The current version of the SPEC suite as of the publication of this book is the SPEC CPU2000 benchmark, the third major revision since the first SPEC benchmark suite was published in 1989.

Benchmark suites provide a number of advantages over MIPS and CPI/IPC. First, their performance results are based on total execution times, not rate of instruction execution. Second, they average a system's performance across multiple programs to generate an estimate of its average speed. This makes a system's overall rating on a benchmark suite a better indicator of its overall performance than its MIPS rating on any one program. Also, many benchmarks require manufacturers to publish their systems' results on the individual programs within the benchmark, as well as the system's overall score on the benchmark suite, making it possible to do a direct comparison of individual benchmark results if you know that a system will be used for a particular application.

1.6.4 GEOMETRIC VERSUS ARITHMETIC MEAN

Many benchmark suites use the *geometric* rather than the *arithmetic* mean to average the results of the programs contained in the benchmark suite, because a single extreme value has less of an impact on the geometric mean of a series than on the arithmetic mean. Using the geometric mean makes it harder for a system to achieve a high score on the benchmark suite by achieving good performance on just one of the programs in the suite, making the system's overall score a better indicator of its performance on most programs.

The geometric mean of n values is calculated by multiplying the n values together and taking the nth root of the product. The arithmetic mean, or average, of a set of values is calculated by adding all of the values together and dividing by the number of values.

EXAMPLE

What are the arithmetic and geometric means of the values 4, 2, 4, 82?

Solution

$$\frac{4+2+4+82}{4} = 23$$

$$\sqrt[4]{4 \times 2 \times 4 \times 82} = 7.36$$

Note that the one large extreme value in the series had a much greater effect on the arithmetic mean than on the geometric mean.

1.7 Speedup

Computer architects often use the term *speedup* to describe how the performance of an architecture changes as different improvements are made to the architecture. Speedup is simply the ratio of the execution times before and after a change is made, so

$$\text{Speedup} = \frac{\text{Execution time}_{\text{before}}}{\text{Execution time}_{\text{after}}}$$

For example, if a program takes 25 seconds to run on one version of an architecture and 15 seconds to run on a new version, the overall speedup is 25 seconds/15 seconds = 1.67.

1.8 Amdahl's Law

The most important rule for designing high-performance computer systems is *make the common case fast*. Qualitatively, this means that the impact of a given performance improvement on overall performance is dependent on both how much the improvement improves performance when it is in use and how often the improvement is in use. Quantitatively, this rule has been expressed as ***Amdahl's Law***, which states

$$\text{Execution Time}_{\text{new}} = \text{Execution Time}_{\text{old}} \times \left[\text{Frac}_{\text{unused}} + \frac{\text{Frac}_{\text{used}}}{\text{Speedup}_{\text{used}}}\right]$$

In this equation, $\text{Frac}_{\text{unused}}$ is the fraction of time (not instructions) that the improvement is not in use, $\text{Frac}_{\text{used}}$ is the fraction of time that the improvement is in use, and $\text{Speedup}_{\text{used}}$ is the speedup that occurs when the improvement is used (this would be the overall speedup if the improvement were in use at all times). Note that Frac_{old} and $\text{Frac}_{\text{unused}}$ are computed using the execution time *before* the modifica-

tion is applied. Computing these values using the execution time after the modification is applied will give incorrect results.

Amdahl's Law can be rewritten using the definition of speedup to give

$$\text{Speedup} = \frac{\text{Execution Time}_{old}}{\text{Execution Time}_{new}} = \frac{1}{\text{Frac}_{unused} + \frac{\text{Frac}_{used}}{\text{Speedup}_{used}}}$$

EXAMPLE

Suppose that a given architecture does not have hardware support for multiplication, so multiplications have to be done through repeated addition (this was the case on some early microprocessors). If it takes 200 cycles to perform a multiplication in software, and 4 cycles to perform a multiplication in hardware, what is the overall speedup from hardware support for multiplication if a program spends 10 percent of its time doing multiplications? What about a program that spends 40 percent of its time doing multiplications?

In both cases, the speedup when the multiplication hardware is used is 200/4 = 50 (ratio of time to do a multiplication without the hardware to time with the hardware). In the case where the program spends 10 percent of its time doing multiplications, $\text{Frac}_{unused} = 0.9$, and $\text{Frac}_{used} = 0.1$. Plugging these values into Amdhal's Law, we get Speedup = 1/[.9 + (.1/50)] = 1.11. If the program spends 40 percent of its time doing multiplications before the addition of hardware multiplication, then $\text{Frac}_{unused} = 0.6$, $\text{Frac}_{used} = 0.4$, and we get Speedup = 1/[.6 + (.4/50)] = 1.64.

This example illustrates the impact that the fraction of time an improvement is used has on overall performance. As Speedup_{used} goes to infinity, overall speedup converges to $1/\text{Frac}_{unused}$, because the improvement can't do anything about the execution time of the fraction of the program that does not use the improvement.

1.9 Summary

This chapter has been intended to provide a context for the rest of the book by explaining some of the technology forces that drive computer performance and providing a framework for discussing and evaluating system performance that will be used throughout the book.

The important concepts for the reader to understand after studying this chapter are as follows:

1. Computer technology is driven by improvements in semiconductor fabrication technology, and these improvements proceed at a geometric, rather than a linear, pace.
2. There are many ways to measure computer performance, and the most effective measures of overall performance are based on the performance of a system on a wide variety of applications.
3. It is important to understand how a given performance metric is generated

in order to understand how useful it is in predicting system performance on a given application.

4. The impact of a change to an architecture on overall performance is dependent not only on how much that change improves performance when it is used, but on how often the change is useful. A consequence of this is that the overall performance impact of an improvement is limited by the fraction of time that the improvement is not in use, regardless of how much speedup the improvement gives when it is useful.

Solved Problems

Technology Trends (I)

1.1. As an illustration of just how fast computer technology is improving, let's consider what would have happened if automobiles had improved equally quickly. Assume that an average car in 1977 had a top speed of 100 miles per hour (mi/h, an approximation) and an average fuel economy of 15 miles per gallon (mi/g). If both top speed and efficiency improved at 35 percent per year from 1977 to 1987, and by 50 percent per year from 1987 to 2000, tracking computer performance, what would the average top speed and fuel economy of a car be in 1987? In 2000?

Solution

In 1987:

The span 1977 to 1987 is 10 years, so both traits would have improved by a factor of $(1.35)^{10} = 20.1$, giving a top speed of 2010 mi/h and a fuel economy of 301.5 mi/g.

In 2000:

Thirteen more years elapse, this time at a 50 percent per year improvement rate, for a total factor of $(1.5)^{13} = 194.6$ over the 1987 values. This gives a top speed of 391,146 mi/h and a fuel economy of 58,672 mi/g. This is fast enough to cover the distance from the earth to the moon in under 40 min, and to make the round trip on less than 10 gal of gasoline.

Technology Trends (II)

1.2. Since 1987, computer performance has been increasing at about 50 percent per year, with improvements in fabrication technology accounting for about 35 percent per year and improvements in architecture accounting for about 15 percent per year.

1. If the performance of the best available computer on 1/01/1988 was defined to be 1, what would be the expected performance of the best available computer on 1/01/2001?
2. Suppose that there had been no improvements in computer architecture since 1987, making fabrication technology the only source of performance improvements. What would the expected performance of the best available computer on 1/01/2001 be?
3. Now suppose that there had been no improvements in fabrication technology, making improvements in architecture the only source of performance improvements. What would the expected performance of the fastest computer on 1/01/2001 be then?

Solution

1. Performance improves at 50 percent per year, and 1/01/1988 to 1/01/2001 is 13 years, so the expected performance of the 1/01/2001 machine is $1 \times (1.5)^{13} = 194.6$.
2. Here, performance only improves at 35 percent per year, so the expected performance is 49.5.
3. Performance improvement is 15 percent per year, giving an expected performance of 6.2.

Speedup (I)

1.3. If the 1998 version of a computer executes a program in 200 s and the version of the computer made in the year 2000 executes the same program in 150 s, what is the speedup that the manufacturer has achieved over the two-year period?

Solution

$$\text{Speedup} = \frac{\text{Execution time}_{\text{before}}}{\text{Execution time}_{\text{after}}}$$

Given this, the speedup is 200 s/150 s = 1.33. Clearly, this manufacturer is falling well short of the industrywide performance growth rate.

Speedup (II)

1.4. To achieve a speedup of 3 on a program that originally took 78 s to execute, what must the execution time of the program be reduced to?

Solution

Here, we have values for speedup and Execution $\text{time}_{\text{before}}$. Substituting these into the formula for speedup and solving for Execution $\text{time}_{\text{after}}$ tells us that the execution time must be reduced to 26 s to achieve a speedup of 3.

Measuring Performance (I)

1.5. 1. Why are benchmark programs and benchmark suites used to measure computer performance?
2. Why are there multiple benchmarks that are used by computer architects, instead of one "best" benchmark?

Solution

1. Computer systems are often used to run a wide range of programs, some of which may not exist at the time the system is purchased or built. Thus, it is generally not possible to measure a system's performance on the set of programs that will be run on the machine. Instead, benchmark programs and suites are used to measure the performance of a system on one or more applications that are believed to be representative of the set of programs that will be run on the machine.
2. Multiple benchmark programs suites exist because computers are used for a wide range of applications, the performance of which can depend on very different aspects of the computer system. For example, the performance of database and transaction-processing applications tends to depend strongly on the performance of a computer's input–output subsystem. In contrast, scientific computing applications depend mainly on the performance of a system's processor and memory system. Like applications, benchmark suites vary in terms of the amount of stress they place on each of the computer's subsystems, and it is important to use a benchmark that stresses the same subsystems as the intended applications; using a processor-intensive benchmark to evaluate computers intended for transaction processing would not give a good estimate of the rate at which these systems could process transactions.

Measuring Performance (II)

1.6. When running a particular program, computer A achieves 100 MIPS and computer B achieves 75 MIPS. However, computer A takes 60 s to execute the program, while computer B takes only 45 s. How is this possible?

Solution

MIPS measures the rate at which a processor executes instructions, but different processor architectures require different numbers of instructions to perform a given computation. If computer A had to execute significantly more instructions than computer B to complete the program, it would be possible for computer A to take longer to run the program than processor B despite the fact that computer A executed more instructions per second.

Measuring Performance (III)

1.7. Computer C achieves a score of 42 on a benchmark suite (higher scores are better), while computer D's score is 35 on the same benchmark. When running your program, you find that computer C takes 20 percent longer to run the program than computer D. How is this possible?

Solution

The most likely explanation is that your program is highly dependent on an aspect of the system that is not stressed by the benchmark suite. For example, your program might perform a large number of floating-point calculations, while the benchmark suite emphasized integer performance or vice versa.

CPI

1.8. When run on a given system, a program takes 1,000,000 cycles. If the system achieves a CPI of 40, how many instructions were executed in running the program

Solution

CPI = #Cycles/#Instructions. Therefore, #Instructions = #Cycles/CPI. 1,000,000 cycles/40 CPI = 25,000. So, 25,000 instructions were executed in running the program.

IPC

1.9. What is the IPC of a program that executes 35,000 instructions and requires 17,000 cycles to complete?

Solution

IPC = #Instructions/#Cycles, so the IPC of this program is 35,000 instructions/17,000 cycles = 2.06.

Geometric versus Arithmetic Mean

1.10. Given the following set of individual benchmark scores for each of the programs in the integer portion of the SPEC2000 benchmark, compute the arithmetic and geometric means of each set. Note that these scores do not represent an actual set of measurements taken on a machine. They were selected to illustrate the impact that using a different method to calculate the mean value has on benchmark scores.

Benchmark	Score Before Improvement	Score After Improvement
164.gzip	10	12
175.vpr	14	16
176.gcc	23	28
181.mcf	36	40
186.crafty	9	12
197.parser	12	120
252.eon	25	28
253.perlbmk	18	21
254.gap	30	28
255.vortex	17	21
256.bzip2	7	10
300.twolf	38	42

Solution

There are 12 benchmarks in the suite, so the arithmetic mean is computed by adding together all of the values in each set and dividing by 12, while the geometric mean is calculated by taking the 12th root of the product of all of the values in a set. This gives the following values:

Before improvement: Arithmetic Mean = 19.92. Geometric Mean = 17.39
After improvement: Arithmetic Mean = 31.5. Geometric Mean = 24.42

What we see from this is that the arithmetic mean is much more sensitive to large changes in one of the values in the set than the geometric mean. Most of the individual benchmarks see relatively small changes as we add the improvement to the architecture, but 197.parser improves by a factor of 10. This causes the arithmetic mean to increase by almost 60 percent, while the geometric mean increases by only 40 percent. This reduced sensitivity to individual values is why benchmarking

experts prefer the geometric mean for averaging the results of multiple benchmarks, since one very good or very bad result in the set of benchmarks has less of an impact on the overall score.

Amdahl's Law (I)

1.11. Suppose a computer spends 90 percent of its time handling a particular type of computation when running a given program, and its manufacturers make a change that improves its performance on that type of computation by a factor of 10.

1. If the program originally took 100 s to execute, what will its execution time be after the change?
2. What is the speedup from the old system to the new system?
3. What fraction of its execution time does the new system spend executing the type of computation that was improved?

Solution

1. This is a direct application of Amdahl's Law:

$$\text{Execution Time}_{new} = \text{Execution Time}_{old} \times \left[\text{Frac}_{unused} + \frac{\text{Frac}_{used}}{\text{Speedup}_{used}}\right]$$

   $\text{Execution Time}_{old} = 100$ s, $\text{Frac}_{used} = 0.9$, $\text{Frac}_{unused} = 0.1$, and $\text{Speedup}_{used} = 10$. This gives an Execution Time_{new} of 19 s.
2. Using the definition of speedup, we get a speedup of 5.3. Alternately, we could substitute the values from part 1 into the speedup version of Amdahl's Law to get the same result.
3. Amdahl's Law doesn't give us a direct way to answer this question. The original system spent 90 percent of its time executing the type of computation that was improved, so it spent 90 s of a 100-s program executing that type of computation. Since the computation was improved by a factor of 10, the improved system spends 90/10 = 9 s executing that type of computation. Because 9 s is 47 percent of 19 s, the new execution time, the new system spends 47 percent of its time executing the type of computation that was improved.

   Alternately, we could have calculated the time that the original system spent executing computations that weren't improved (10 s). Since these computations weren't changed when the improvement was made, the amount of time spent executing them in the new system is the same as the old system. This could then be used to compute the percent of time spent on computations that weren't improved, and the percent of time spent on computations that were improved generated by subtracting that from 100.

Amdahl's Law (II)

1.12. A computer spends 30 percent of its time accessing memory, 20 percent performing multiplications, and 50 percent executing other instructions. As a computer architect, you have to choose between improving either the memory, multiplication hardware, or execution of nonmultiplication instructions. There is only space on the chip for one improvement, and each of the improvements will improve its associated part of the computation by a factor of 2.

1. Without performing any calculations, which improvement would you expect to give the largest performance increase, and why?
2. What speedup would making each of the three changes give?

Solution

1. Improving the execution of nonmultiplication instructions should give the greatest benefit. Each benefit increases the performance of its affected area by the same amount, and the system spends more time executing nonmultiplication instructions than either of the other categories. Since Amdahl's Law says that the overall impact of an improvement goes up as the fraction of time the improvement is used goes up, improving nonmultiplication instructions should give the greatest improvement.
2. Substituting the percentage of time used and improvement when used values into the speedup form of Amdahl's Law shows that improving the memory system gives a speedup of 1.18, improving multiplication gives a speedup of 1.11, and improving the nonmultiplication instructions gives a speedup of 1.33, confirming the intuition from part 1.

Comparing Different Changes to an Architecture

1.13. Which improvement gives a greater reduction in execution time: one that is used 20 percent of the time but improves performance by a factor of 2 when used, or one that is used 70 percent of the time but only improves performance by a factor of 1.3 when used?

Solution

Applying Amdahl's Law, we get the following equation for the first improvement:

$$\text{Execution Time}_{\text{new}} = \text{Execution Time}_{\text{old}} \times \left[.8 \div \frac{.2}{2}\right]$$

So the execution time with the first improvement is 90 percent of the execution time without the improvement. Plugging the values for the second improvement into Amdahl's Law shows that the execution time with the second improvement is 84 percent of the execution time without the improvement. Thus, the second improvement will have a greater impact on overall execution time despite the fact that it gives less of an improvement when it is in use.

Converting Individual Improvements to Overall Performance Impact

1.14. A computer architect is designing the memory system for the next version of a processor. If the current version of the processor spends 40 percent of its time processing memory references, by how much must the architect speed up the memory system to achieve an overall speedup of 1.2? A speedup of 1.6?

Solution

To solve this, we apply Amdahl's Law for speedups, with $\text{Speedup}_{\text{used}}$ as the unknown, rather than overall Speedup. $\text{Frac}_{\text{used}}$ is 0.4, since the original system spends 40 percent of its time handling memory references, so $\text{Frac}_{\text{unused}}$ is 0.6. For the 20 percent increase in overall performance, this gives:

$$\text{Speedup} = 1.2 = \frac{1}{.6 + \frac{.4}{\text{Speedup}_{\text{used}}}}$$

Solving for $\text{Speedup}_{\text{used}}$, we get

$$\text{Speedup}_{\text{used}} = \frac{.4}{\frac{1}{1.2} - .6} = 1.71$$

To find the value of $\text{Speedup}_{\text{used}}$ required to give a speedup of 1.6, the only value that changes in the above equation is Speedup. Solving again, we get $\text{Speedup}_{\text{used}} = 16$. Here we again see the diminishing returns that come from repeatedly improving only one aspect of a system's performance. To increase the overall speedup from 1.2 to 1.6, we have to increase $\text{Speedup}_{\text{used}}$ by almost a factor of 10, because the 60 percent of the time that the memory system is not in use begins to dominate the overall performance as we improve the memory system performance.

Improving Instructions

1.15. Consider an architecture that has four types of instructions: additions, multiplications, memory operations, and branches. The following table gives the number of instructions that belong to each type in the program you care about, the number of cycles it takes to execute each instruction by type, and the speedup in execution of the instruction type from a proposed improvement (each improvement only affects one instruction type). Rank the improvements for each of the instruction types in terms of their impact on overall performance.

Instruction Type	Number	Execution Time	Speedup to Type
Addition	10 million	2 cycles	2.0
Multiplication	30 million	20 cycles	1.3
Memory	35 million	10 cycles	3.0
Branch	15 million	4 cycles	4.0

Solution

To solve this problem, we must first compute the number of cycles spent executing each instruction type before the improvements are applied and the fraction of the total cycles spent executing each instruction type ($\text{Frac}_{\text{used}}$ for each of the improvements). This will allow us to use Amdahl's Law to compute the overall speedup for each of the proposed improvements. Multiplying the number of instructions in each type by the execution time per instruction gives the number of cycles spent executing each instruction type, and adding these values together gives the total number of cycles to execute the program. This gives the values in the following table: (Total execution time is 1030 million cycles).

Instruction Type	Number	Execution Time	Speedup to Type	Number of Cycles	Fraction of Cycles
Addition	10 million	2 cycles	2.0	20 million	2%
Multiplication	30 million	20 cycles	1.3	600 million	58%
Memory	35 million	10 cycles	3.0	350 million	34%
Branch	15 million	4 cycles	4.0	60 million	6%

We can then plug these values into Amdahl's Law, using the fraction of cycles as $\text{Frac}_{\text{used}}$ to get the overall speedup from each of the improvements.

Thus, improving memory operations gives the best overall speedup, followed by improving multiplications, improving branches, and improving additions:

Instruction Type	Number	Execution Time	Speedup to Type	Number of Cycles	Fraction of Cycles	Overall Speedup
Addition	10 million	2 cycles	2.0	20 million	2%	1.01
Multiplication	30 million	20 cycles	1.3	600 million	58%	1.15
Memory	35 million	10 cycles	3.0	350 million	34%	1.29
Branch	15 million	4 cycles	4.0	60 million	6%	1.05

CHAPTER 2

Data Representations and Computer Arithmetic

2.1 Objectives

This chapter covers the most common methods that computer systems use to represent data and how arithmetic operations are performed on these representations. It begins with a discussion of how bits (binary digits) are represented by electrical signals and proceeds to discuss how integers and floating-point numbers are represented as sequences of bits.

After reading this chapter, you should

1. Have an understanding of how computers represent data internally, at both the bit-pattern and the electrical-signal level
2. Be able to translate integer and floating-point numbers to and from their binary representations
3. Be able to perform basic mathematical operations (addition, subtraction and multiplication) on integers and floating-point numbers

2.2 From Electrons to Bits

Modern computers are *digital* systems, meaning that they interpret electrical signals as having a set of discrete values, rather than as analog quantities. While this increases the number of signals required to convey a given amount of information

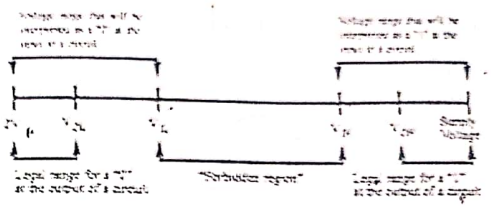

Fig. 2-1. Mapping voltages to bits.

makes storing information much easier and makes digital systems less subject to electrical noise than analog systems.

A digital system's *signaling convention* determines how analog electrical signals are interpreted as digital values. Figure 2-1 illustrates the most common signaling convention in modern computers. Each signal carries one of two values, depending on the voltage level of the signal. Low voltages are interpreted as a 0 and high voltages as a 1.

The digital signaling convention divides the range of possible voltages into several regions. The region from 0V to V_{IL} is the range of voltages that are guaranteed to be interpreted as a 0 at the input to a circuit, while the region from V_{IH} to the supply voltage is guaranteed to be interpreted as a 1 at a circuit's input. The region between V_{IL} and V_{IH} is known as the "forbidden region," because it is not possible to predict whether a circuit will interpret a voltage in that range as a 0 or as a 1.

V_{OL} is the highest voltage that a circuit is allowed to produce when generating a 0, and V_{OH} is the lowest voltage that a circuit is allowed to produce when generating a 1. It is important that V_{OH} and V_{OL} be closer to the extremes of the voltage range than V_{IH} and V_{IL}, because the gaps between V_{OL} and V_{IL} and between V_{IH} and V_{OH} determine the *noise margins* of the digital system. The noise margin of a digital system is the amount by which the output signal of a circuit can change before it is possible that it will be interpreted as the opposite value by another circuit. The wider the noise margin, the better the system is able to tolerate the effects of coupling between electrical signals, resistive losses in wires, and other effects that can cause signals to change between the point where they are generated and the point where they are used.

Systems that map each electrical signal onto two values are known as *binary systems*, and the information carried by each signal is called a *bit* (short for BInary digiT). Systems with more values per signal are possible, but the additional complexity of designing circuits to interpret these signaling conventions and the reduction in noise margins that occurs when the voltage range is divided into more than two values make such systems difficult to build. For this reason, almost all digital systems are binary.

2.3 Binary Representation of Positive Integers

Positive integers are represented using a place-value binary (base-2) system, similar to the place-value system used in decimal (base-10) arithmetic. In base-10 arithmetic, numbers are represented as the sum of multiples of each power of 10, so the number $1543 = (1 \times 10^3) + (5 \times 10^2) + (4 \times 10^1) + (3 \times 10^0)$. For binary numbers, the base of the number is 2, so each position in the number represents an increasing power of 2, rather than an increasing power of 10. For example, the number $0b100111 = (1 \times 2^5) + (0 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) = 39$. Binary quantities are usually preceded by the prefix "0b" to identify them as binary, rather than decimal numbers. Just as an n-digit decimal number can represent values from 0 to $10^n - 1$, an n-bit unsigned binary number can represent values from 0 to $2^n - 1$.

A disadvantage of binary numbers as compared to decimal numbers is that they require significantly more digits to represent a given integer, which can make them cumbersome to work with. To address this, *hexadecimal* notation, in which each digit has 16 possible values, is often used to represent binary numbers. In hexadecimal notation, the numbers 0 through 9 have the same value that they do in decimal notation, and the letters A through F (or a through f—case is irrelevant in hexadecimal) are used to represent the numbers 10 through 15, as shown in Fig. 2-2. To differentiate hexadecimal numbers from binary or decimal numbers, the prefix

Decimal Number	Binary Representation	Hexadecimal Representation
0	0b0000	0x0
1	0b0001	0x1
2	0b0010	0x2
3	0b0011	0x3
4	0b0100	0x4
5	0b0101	0x5
6	0b0110	0x6
7	0b0111	0x7
8	0b1000	0x8
9	0b1001	0x9
10	0b1010	0xA
11	0b1011	0xB
12	0b1100	0xC
13	0b1101	0xD
14	0b1110	0xE
15	0b1111	0xF

Fig. 2-2. Hexadecimal notation.

"0x" is usually placed to the left of the number. Place-value notation with a base of 16 is used when representing values larger than 15 in hexadecimal notation.

EXAMPLE
What are the binary and hexadecimal representations of the decimal number 47?

Solution

To convert decimal numbers to binary, we express them as a sum of values that are powers of two:

$$47 = 32 + 8 + 4 + 2 + 1 = 2^5 + 2^3 + 2^2 + 2^1 + 2^0$$

Therefore, the binary representations of 47 is 0b101111.

To convert decimal numbers to hexadecimal, we can either express the number as the sum of powers of 16 or group the bits in the binary representation into sets of 4 bits and look each set up in Fig. 2-2. Converting directly, $47 = 2 \times 16 + 15 = 0x2F$. Grouping bits, we get $47 = 0b101111 = 0b0010\ 1111$. $0b0010 = 0x2$, $0b1111 = 0xF$, so $47 = 0x2F$.

2.4 Arithmetic Operations on Positive Integers

Arithmetic in base-2 (binary) can be done using the same techniques that humans use for base-10 (decimal) arithmetic, except for the restricted set of values that can be represented by each digit. This is often the easiest way for humans to solve math problems involving binary numbers, but in some cases these techniques cannot be implemented easily in circuits, leading computer designers to choose other implementations of these operations. As we will see in the following sections, addition and multiplication are implemented using circuits that are analogous to the techniques used by humans in doing arithmetic. Division is implemented using computer-specific methods, and subtraction is implemented differently on different systems depending on the representation they use for negative integers.

EXAMPLE
Compute the sum of 9 and 5 using binary numbers in 4-bit binary format.

Solution

The 4-bit binary representations of 9 and 5 are 0b1001 and 0b0101, respectively. Adding the low bits, we get 0b1 + 0b1 = 0b10, which is a 0 in the low bit of the result, and a carry of 1 into the next bit position. Computing the next bit of the sum, we get 0b1 (carry) + 0b0 + 0b0 = 0b1. Repeating for all bits, we get the final result of 0b1110. Figure 2-3 illustrates this process.

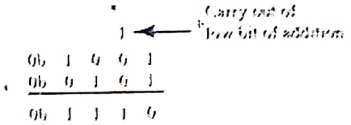

Fig. 2-3. Binary addition example.

2.4.1 ADDITION/SUBTRACTION

The hardware that computers use to implement addition is very similar to the method outlined above. Modules, known as *full adders*, compute each bit of the output based on the corresponding bits of the input and the carry generated by the next-lower bit of the computation. Figure 2-4 shows an 8-bit adder circuit.

For the type of adder described above, the speed of the circuit is determined by the time it takes for the carry signals to propagate through all of the full adders. Essentially, each full adder can't perform its part of the computation until all the full adders to the right of it have completed their parts, so the computation time grows linearly with the number of bits in the inputs. Designers have developed circuits that speed this up somewhat by doing as much of the work of the full adder as possible before the carry input is available to reduce the delay once the carry becomes available, or by taking several input bits into account when generating carries, but the basic technique remains the same.

Subtraction can be handled by similar methods, using modules that compute 1 bit of the difference between two numbers. However, the most common format for negative integers, two's-complement notation, allows subtraction to be performed by negating the second input and adding, making it possible to use the same hardware for addition and subtraction. Two's-complement notation is discussed in Section 2.5.2.

2.4.2 MULTIPLICATION

Unsigned integer multiplication is handled in a similar manner to the way humans multiply multidigit decimal numbers. The first input to the multiplication is multiplied by each bit of the second input separately, and the results added. In binary multiplication, this is simplified by the fact that the result of multiplying

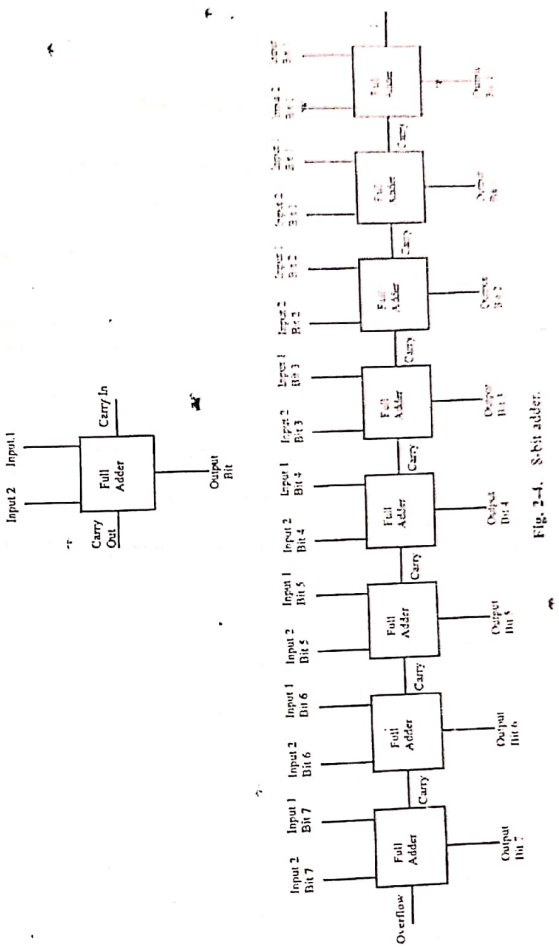

Fig. 2-4. 8-bit adder.

```
  0b1011
x 0b0101
------
  1011
  0000
 1011
+0000
------
0b110111
```

Fig. 2-5. Multiplication example.

number by a bit is either the original number or 0, making the hardware less complex.

Figure 2-5 shows an example of multiplying 11 (0b1011) by 5 (0b0101). First, 0b1011 is multiplied by each bit of 0b0101 to get the partial products shown in the figure. Then, the partial products are added to get the final result. Note that each successive partial product is shifted to the left one position to account for the differing place values of the bits in the second input.

One problem with integer multiplication is that the product of two n-bit numbers can require as many as $2n$ bits to represent. For example, the product of the two 4-bit numbers in Fig. 2-5 requires 6 bits to represent. Many arithmetic operations can generate results that cannot be represented in the same number of bits as their inputs. This is known as *overflow* or *underflow*, and is discussed in Section 2.4.4. In the case of multiplication, the number of bits of overflow is so large that hardware designers take special measures to deal with it. In some cases, designers provide separate operations to compute the high and low n bits of the result of an n-bit by n-bit multiplication. In others, the system discards the high n bits, or places them in a special output register where the programmer can access them if necessary.

2.4.3 DIVISION

Division can be implemented on computer systems by repeatedly subtracting the divisor from the dividend and counting the number of times that the divisor can be subtracted from the dividend before the dividend becomes smaller than the divisor. For example, 15 can be divided by 5 by subtracting 5 repeatedly from 15, getting 10, 5, and 0 as intermediate results. The quotient, 3, is the number of subtractions that had to be performed before the intermediate result became less than the dividend.

While it would be possible to build hardware to implement division through repeated subtraction, it would be impractical because of the number of subtractions required. For example, 2^{31} (one of the larger numbers representable in 32-bit unsigned integers) divided by 2 is 2^{30}, meaning that 2^{30} subtractions would have to be done to perform this division by repeated subtraction. On a system operating at 1 GHz, this would take approximately 1 s, far longer than any other arithmetic operation.

Instead, designers use table lookup-based methods to implement division. Using pregenerated tables, these techniques generate 2 to 4 bits of the quotient in each cycle. This allows 32-bit or 64-bit integer divisions to be done in a reasonal number

of cycles, although division is typically the slowest of the basic mathematical operations on a computer.

2.4.4 OVERFLOW/UNDERFLOW

The bit width of a computer limits the maximum and minimum numbers it can represent as integers. For unsigned integers, an n-bit number can represent values from 0 to $2^n - 1$. However, arithmetic operations on numbers that can be represented in a given number of bits can generate results that cannot be represented in the same format. For example, adding two n-bit integers can generate a result of up to $2(2^n - 1)$, which cannot be represented in n bits, and it is possible to generate negative results by subtracting two positive integers, which also cannot be represented by an n-bit unsigned number.

When an operation generates a result that cannot be expressed in the format of its input operands, an *overflow* or *underflow* is said to have occurred. Overflows occur when the result of an operation is too large to represent in the input format, and underflows occur when the result is too small to represent in the format. Different systems handle overflows and underflows in different ways. Some signal an error when they occur. Others replace the result with the closest value that can be represented in the format. For floating-point numbers, the IEEE standard specifies a set of special representations that indicate that overflow or underflow has occurred. These representations are called not-a-numbers (NaNs) and are discussed in Section 2.6.1.

2.5 Negative Integers

To represent negative integers as sequences of bits, the place-value notation used for positive integers must be extended to indicate whether a number is positive or negative. We will cover two schemes for doing this: sign-magnitude representations and two's-complement notation.

2.5.1 SIGN-MAGNITUDE REPRESENTATION

In sign-magnitude representations, the high bit (also known as the sign bit) of a binary number indicates whether the number is positive or negative, and the remainder of the number indicates the absolute value (or magnitude) of the number, using the same format as unsigned binary representation. N-bit sign-magnitude numbers can represent quantities from $-(2^{(N-1)} - 1)$ to $+(2^{(N-1)} - 1)$. Note that there are two representations of 0 in sign-magnitude notation: +0 and −0. +0 has a value of 0 in the magnitude field and a positive sign bit. −0 has a value of 0 in the magnitude field and a negative sign bit.

EXAMPLE
The 16-bit unsigned binary representation of 152 is 0b0000 0000 1001 1000. In a 16-bit sign-magnitude system, −152 would be represented as 0b1000 0000 1001 1000. Here, the leftmost bit of the number is the sign bit, and the rest of the number gives the magnitude.

Sign-magnitude representations have the advantage that taking the negative of a number is very easy—just invert the sign bit. Determining whether a number is positive or negative is also easy, as it only requires examining the sign bit. Sign-magnitude representation makes it easy to perform multiplication and division on signed numbers, but hard to perform addition and subtraction. For multiplication and division, the hardware can simply perform unsigned operations on the magnitude portion of the inputs and examine the sign bits of the inputs to determine the sign bit of the result.

EXAMPLE
Multiply the numbers +7 and −5, using 6-bit signed-magnitude integers.

Solution

The binary representation of +7 is 000111, and −5 is 100101. To multiply them, we multiply their magnitude portions as unsigned integers, giving 0100011 (35). We then examine the sign bits of the numbers being multiplied and determine that one of them is negative. Therefore, the result of the multiplication must be negative, giving 1100011 (−35).

Addition and subtraction of sign-magnitude numbers requires relatively complex hardware because adding (or subtracting) the binary representation of a positive number and the binary representation of a negative number does not give the correct result. The hardware must take the value of the sign bit into account when computing each bit of the output, and different hardware is required to perform addition and subtraction. This hardware complexity is the reason why very few current systems use sign-magnitude notation for integers.

EXAMPLE
What is the result if you try to directly add the 8-bit sign-magnitude representations of +10 and −4?

Solution

The 8-bit sign-magnitude representations of +10 and −4 are 0b00001010 and 0b10000100. Adding these binary numbers together gives 0b10001110, which sign-magnitude systems interpret as −14, not 6 (the correct result of the computation).

2.5.2 TWO'S-COMPLEMENT NOTATION

In two's-complement notation, a negative number is represented by inverting each bit of the unsigned representation of the number and adding 1 to the result (discarding any overflow bits that do not fit in the width of the representation). The name "two's complement" comes from the fact that the unsigned sum of an n-bit two's-complement number and its negative is 2^n.

EXAMPLE
What is the 8-bit two's-complement representation of −12, and what is the unsigned result of adding the representations of +12 and −12?

Solution

The 8-bit representation of +12 is 0b00001100, so the 8-bit two's-complement representation of −12 is 0b11110100. (Negating each bit in the positive representation gives 0b11110011, and adding 1 gives the final result of 0b11110100.) This process is illustrated in Fig. 2-6.

Original value:	0b00001100	(12)
Negate each bit:	0b11110011	
Add 1:	0b11110100	(Two's-complement representation of -12)

Fig. 2-6. Two-complement negation.

Adding the representations of +12 and −12 gives 0b00001100 + 0b11110100, which is 0b100000000. Treating this as a 9-bit unsigned number, we interpret it as 256. $256 = 2^8$. Treating the result as an 8-bit two's-complement number, we drop the 1 that overflowed out of the 8-bit computation to get the 8-bit result $0b00000000 = 0$, which is the result we'd expect from adding +12 and −12.

Two's-complement numbers have a number of useful properties, which explain why they are used in almost all modern computers:

1. The sign of a number can be determined by examining the high bit of the representation. Negative numbers have 1s in their high bit; positive numbers have 0s.
2. Negating a number twice gives the original number, so no special hardware is required to negate negative numbers.
3. Two's-complement notation has only one representation for zero, eliminating the need for hardware to detect +0 and −0.
4. Most importantly, adding the representations of a positive and a negative two's-complement number (discarding overflows) gives the two's-complement representation of the correct result. In addition to eliminating the need for special hardware to handle addition of negative numbers, subtraction can be recast as addition by computing the two's-complement

negation of the subtrahend and adding that to the two's-complement representation of the minuend (e.g., $14 - 7$ becomes $14 + (-7)$), further reducing hardware costs.

While sign-magnitude representations share the first two advantages of two's-complement numbers, the other two give two's-complement notation a significant advantage over sign-magnitude notation. One somewhat unusual characteristic of two's-complement notation is that an n-bit two's-complement number can represent values from $-(2^{(n-1)})$ to $+(2^{(n-1)} - 1)$. This asymmetry comes from the fact that there is only one representation for zero, allowing an odd number of nonzero quantities to be represented.

EXAMPLE
What is the result of negating the 4-bit two's-complement representation of +5 twice?

Solution

$+5 = 0b0101$. The two's-complement negation of this is 0b1011 (−5). Negating that quantity gives 0b0101, the original number.

EXAMPLE
Add the quantities +3 and −4 in 4-bit two's-complement notation.

Solution

The 4-bit two's-complement representations of +3 and −4 are 0b0011 and 0b1100. Adding these together gives 0b1111, which is the two's-complement representation of −1.

EXAMPLE
Compute −3 − 4 in 4-bit two's-complement notation.

Solution

To perform subtraction, we negate the second operand and add. Thus, the actual computation we want to perform is $-3 + (-4)$. The two's-complement representations of −3 and −4 are 0b1101 and 0b1100. Adding these quantities, we get 0b11001 (a 5-bit result, counting the overflow). Discarding the fifth bit, which doesn't fit in the representation, gives 0b1001, the two's-complement representation of −7.

Multiplication of two's-complement numbers is more complicated, because performing a straightforward unsigned multiplication of the two's-complement

representations of the inputs does not give the correct result. Multipliers could be designed to convert both of their inputs to positive quantities and use the sign bits of the original inputs to determine the sign of the result, but this increases the time required to perform a multiplication. Instead, a technique called *booth encoding*, which is beyond the scope of this book, is used to quickly convert two's-complement numbers into a format that is easily multiplied.

As we have seen, both sign-magnitude and two's-complement numbers have their pros and cons. Two's-complement numbers allow simple implementations of addition and subtraction, while sign-magnitude numbers facilitate multiplication and division. Because addition and subtraction are much more common in computer programs than multiplication and division, virtually all computer manufacturers have chosen two's-complement representations for integers, allowing them to "make the common case fast."

2.5.3 SIGN EXTENSION

In computer arithmetic, it is sometimes necessary to convert numbers represented in a given number of bits to a representation that uses a larger number of bits. For example, a program might need to add an 8-bit input to a 32-bit quantity. To get the correct result, the 8-bit input must be converted to a 32-bit quantity before it can be added to the 32-bit integer, which is known as *sign extension*.

Converting unsigned numbers to wider representations simply requires filling in the bits to the left of those in the original representation with zeroes. For example, the 8-bit unsigned quantity 0b10110110 becomes the 16-bit unsigned quantity 0b0000000010110110. To sign-extend a sign-magnitude number, move the sign bit (the most significant bit) of the old representation into the sign bit of the new representation, and fill in all of the additional bits in the new representation (including the bit position of the old sign bit) with zeroes.

EXAMPLE
What is the 16-bit sign-magnitude representation of the 8-bit sign-magnitude quantity 0b10000111 (−7)?

Solution

To sign-extend the number, we move the old sign bit to the most significant bit of the new representation and fill in all other bit positions with zeroes. This gives 0b1000000000000111 as the 16-bit sign-magnitude representation of −7.

Sign-extending two's-complement numbers is slightly more complicated. To sign-extend a two's-complement number, *copy* the high bit of the old representation into each additional bit of the new representation. Thus, sign-extended positive numbers will have zeroes in all of the bits added in going to the wider representation, and sign-extended negative numbers will have ones in all of these bit positions.

EXAMPLE

What is the 16-bit sign extension of the 8-bit two's-complement number 0b10010010 (−110)?

Solution

To sign extend this number, we copy the high bit into all of the new bit positions introduced by widening the representation. This gives 0b1111111110010010. Negating this, we get 0b0000000001101110 (+110), confirming that sign extending two's complement numbers gives the correct result.

2.6 Floating-Point Numbers

Floating-point numbers are used to represent quantities that cannot be represented by integers, either because they contain fractional values or because they lie outside the range representable within the system's bit width. Virtually all modern computers use the floating-point representation specified in IEEE standard 754, in which numbers are represented by a mantissa and an exponent. Similar to scientific notation, the value of a floating-point number is $mantissa \times 2^{exponent}$.

This representation allows a wide range of values to be represented in a relatively small number of bits, including both fractional values and values whose magnitude is much too large to represent in an integer with the same number of bits. However, it creates the problem that many of the values in the range of the floating-point representation cannot be represented exactly, just as many of the real numbers cannot be represented by a decimal number with a fixed number of significant digits. When a computation creates a value that cannot be represented exactly by the floating-point format, the hardware must round the result to a value that can be represented exactly. In the IEEE 754 standard, the default way of rounding (called the *rounding mode*) is round-to-nearest. In round-to-nearest, values are rounded to the closest representable number, and results that lie exactly halfway between two representable numbers are rounded such that the least-significant digit of their result is even. The standard specifies several other rounding modes that can be selected by programs, including round toward 0, round toward +infinity, and round toward −infinity.

EXAMPLE

The rounding modes in the IEEE standard can be applied to decimal numbers as well as floating-point binary representations. How would the following decimal numbers be rounded to two significant digits, using round-to-nearest mode?

a. 1.345
b. 78.953
c. 12.5
d. 13.5

Solution

a. 1.345 is closer to 1.3 than 1.4, so it will be rounded to 1.3. Another way to look at this is that the third-most-significant digit is less than 5, so it rounds to 0 when we round to two significant digits.
b. In 78.953, the third-most-significant digit is 9, which rounds up to 10, so 78.953 will round to 79.
c. In 12.5, the third-most-significant digit is 5, so we round in the direction that makes the least-significant digit of the result even. In this case, that means rounding down to a result of 12.
d. Here, we have to round up to 14 because the third-most-significant digit is 5, and we have to round up to make the result even.

The IEEE 754 standard specifies a number of bit widths for floating-point numbers. The two most commonly used widths are single-precision and double-precision, which are illustrated in Fig. 2-7. Single-precision numbers are 32 bits long and contain 8 bits of exponent, 23 bits of fraction, and 1 sign bit, which contains the sign of the fraction field. Double-precision numbers have 11 bits of exponent, 52 bits of fraction, and 1 sign bit.

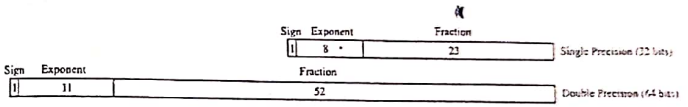

Fig. 2-7. IEEE 754 Floating-point formats.

Both the exponent and the fraction field of an IEEE 754 floating-point number are encoded differently than the integer representations we have discussed in this chapter. The fraction field is a sign-magnitude number that represents the fractional portion of a binary number whose integer portion is assumed to be 1. Thus, the mantissa of an IEEE 754 floating number is $+/-1.fraction$, depending on the value of the sign bit. Using an assumed "leading 1" in this way increases the number of significant digits that can be represented by a floating-point number of a given width.

EXAMPLE

What is the fraction field of the single-precision floating point representation of 6.25?

Solution

Fractional binary numbers use the same place-value representation as decimal numbers, with a base of 2, so the binary number $0b11.111 = 2^1 + 2^0 + 2^{-1} + 2^{-2} + 2^{-3} = 3.875$. Using this format, a decimal fraction

EXAMPLE
What is the 16-bit sign-extension of the 8-bit two's-complement quantity 0b10010010 (−110)?

Solution

To sign-extend this number, we copy the high bit into all of the new bit positions introduced by widening the representation. This gives 0b1111111110010010. Negating this, we get 0b0000000001101110 (+110), confirming that sign-extending two's-complement numbers gives the correct result.

2.6 Floating-Point Numbers

Floating-point numbers are used to represent quantities that cannot be represented by integers, either because they contain fractional values or because they lie outside the range representable within the system's bit width. Virtually all modern computers use the floating-point representation specified in IEEE standard 754, in which numbers are represented by a mantissa and an exponent. Similar to scientific notation, the value of a floating-point number is $mantissa \times 2^{exponent}$.

This representation allows a wide range of values to be represented in a relatively small number of bits, including both fractional values and values whose magnitude is much too large to represent in an integer with the same number of bits. However, it creates the problem that many of the values in the range of the floating-point representation cannot be represented exactly, just as many of the real numbers cannot be represented by a decimal number with a fixed number of significant digits. When a computation creates a value that cannot be represented exactly by the floating-point format, the hardware must round the result to a value that can be represented exactly. In the IEEE 754 standard, the default way of rounding (called the *rounding mode*) is round-to-nearest. In round-to-nearest, values are rounded to the closest representable number, and results that lie exactly halfway between two representable numbers are rounded such that the least-significant digit of their result is even. The standard specifies several other rounding modes that can be selected by programs, including round toward 0, round toward +infinity, and round toward −infinity.

EXAMPLE
The rounding modes in the IEEE standard can be applied to decimal numbers as well as floating-point binary representations. How would the following decimal numbers be rounded to two significant digits, using round-to-nearest mode?

a. 1.345
b. 78.953
c. 12.5
d. 13.5

Solution

a. 1.345 is closer to 1.3 than 1.4, so it will be rounded to 1.3. Another way to look at this is that the third-most-significant digit is less than 5, so it rounds to 0 when we round to two significant digits.
b. In 78.953, the third-most-significant digit is 9, which rounds up to 10, so 78.953 will round to 79.
c. In 12.5, the third-most-significant digit is 5, so we round in the direction that makes the least-significant digit of the result even. In this case, that means rounding down to a result of 12.
d. Here, we have to round up to 14 because the third-most-significant digit is 5, and we have to round up to make the result even.

The IEEE 754 standard specifies a number of bit widths for floating-point numbers. The two most commonly used widths are single-precision and double-precision, which are illustrated in Fig. 2-7. Single-precision numbers are 32 bits long and contain 8 bits of exponent, 23 bits of fraction, and 1 sign bit, which contains the sign of the fraction field. Double-precision numbers have 11 bits of exponent, 52 bits of fraction, and 1 sign bit.

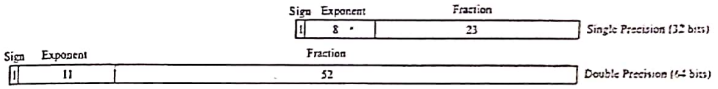

Fig. 2-7. IEEE 754 Floating-point formats.

Both the exponent and the fraction field of an IEEE 754 floating-point number are encoded differently than the integer representations we have discussed in this chapter. The fraction field is a sign-magnitude number that represents the fractional portion of a binary number whose integer portion is assumed to be 1. Thus, the mantissa of an IEEE 754 floating number is $+/-\ 1.\ fraction$, depending on the value of the sign bit. Using an assumed "leading 1" in this way increases the number of significant digits that can be represented by a floating-point number of a given width.

EXAMPLE
What is the fraction field of the single-precision floating point representation of 6.25?

Solution

Fractional binary numbers use the same place-value representation as decimal numbers, with a base of 2, so the binary number $0b11.111 = 2^1 + 2^0 + 2^{-1} + 2^{-2} + 2^{-3} = 3.875$. Using this format, a decimal fraction

can be converted to a binary fraction directly, so $6.25 = 2^2 + 2^1 + 2^{-2} = 0b110.01$.

To find the fraction field, we shift the binary representation of the number down so that the value to the left of the binary[1] point is 1, so 0b110.01 becomes $0b1.1001 \times 2^2$. In the normalized fraction representation used in floating-point numbers, the leading 1 is assumed, and only the values to the right of the binary point (1001 in this case) are represented. Extending the value to the 23-bit fraction format of single-precision floating-point, we get 1001 0000 0000 0000 0000 000 as the fraction field. Note that, when we extend fractional values to a wider representation, we add zeroes to the *right* of the last one, as opposed to sign-extending unsigned integers, where the zeroes are added to the left of the most-significant one.

The exponent field of a floating-point number uses a biased integer representation, in which a fixed bias is added to a value to determine its representation. For single-precision floating-point numbers, the bias is 127 (bias = 1023 for double-precision numbers), so the value of the exponent field can be found by subtracting 127 from the unsigned binary number contained in the field.

EXAMPLE
How would the numbers −45 and 123 be represented in the 8-bit biased notation used in the exponents of single-precision numbers?

Solution

The bias value for this format is 127, so we add 127 to each number to get the biased representation. $-45 + 127 = 82 = 0b01010010$. $123 + 127 = 250 = 0b11111010$.

EXAMPLE
What is the value of the exponent represented by an exponent field of 0b11100010 in a single-precision floating-point number?

Solution

$0b11100010 = 226$. $226 - 127 = 99$, so the exponent field has a value of 99.

Biased representations are somewhat unusual, but they have one significant advantage: They allow floating-point comparisons to be done using the same comparison hardware as unsigned integer comparisons, since larger values of a biased encoding correspond to larger values of the encoded number. Given the

[1] *The binary point* is the equivalent of the decimal point in a binary representation.

formats for the fraction and exponent fields, the value of a floating-point number is [illegible] if sign bit is 1, 1 if sign bit is 0) × (1.fraction) × 2^(exponent − bias).

EXAMPLE
What is the value of the single-precision floating-point number represented by the hexadecimal value 0x3FC00000?

Solution

Dividing this number into the fields specified in Fig. 2-7, we get a sign bit of 0, an exponent field of $0b01111111 = 127$, and a fraction field of 0b10000000000000000000000. Subtracting the bias of 127 from the exponent field gives an exponent of 0. The mantissa is $1.1_2 = 1.5$ once we include the implied 1 to the left of the binary point in the fraction field, so the value of the floating-point number is $1 \times 1.5 \times 2^0 = 1.5$.

2.6.1 NaNs AND DENORMALIZED NUMBERS

The IEEE floating-point standard specifies several bit patterns that represent values that it is not possible to represent exactly in the base floating-point format: zero, denormalized numbers, and NaNs (not-a-numbers). The assumed 1 in the mantissa of floating-point numbers allows an additional bit of precision in the representation that prevents the value 0 from being represented exactly, since a fraction field of 0 represents the mantissa 1.0. Since representing 0 exactly is very important for numerical computations, the IEEE standard specifies that, when the exponent field of a floating-point number is 0, the leading bit of the mantissa is assumed to be 0. Thus, a floating-point number with a fraction field of 0 and an exponent field of 0 represents 0 exactly. This convention also allows numbers that lie closer to 0 than $1.0 \times 2^{(1-bias)}$ to be represented, although they have fewer bits of precision than numbers that can be represented with an assumed 1 before the fraction field.

Floating-point numbers (except for 0) that have an exponent field of 0 are known as *denormalized* numbers because of the assumed 0 in the integer portion of their mantissa. This is in contrast to numbers with other values of the exponent field, which have an assumed 1 in the integer portion of their mantissa and are known as *normalized* numbers. All denormalized numbers are assumed to have an exponent field of (1-bias), instead of the (0-bias) that would be generated by just subtracting the bias from the value of their exponent. This provides a smaller gap between the smallest-magnitude normalized number and the largest-magnitude denormalized number that can be represented by a format.

The other type of special value in the floating-point standard is NaNs. NaN stands for "not a number," and NaNs are used to signal error conditions such as overflows, underflows, division by 0, and so on. When one of these error conditions occur in an operation, the hardware generates a NaN as its result rather than signaling an exception. Subsequent operations that receive a NaN as one of their inputs copy that NaN to their outputs rather than performing their normal

computation. NaNs are indicated by all 1s in the exponent field of a floating-point number, unless the fraction field of the number is 0, in which case the number represents infinity. The existence of NaNs makes it easier to write programs that run on multiple different computers, because programmers can check the results of each computation for errors within the program, rather than relying on the system's exception-handling functions, which vary significantly between different computers. Figure 2-8 summarizes the interpretation of different values of the exponent and fraction fields of a floating-point number.

Exponent Field	Fraction Field	Represents
0	0	0
0	not 0	+/- (0.fraction)$\times 2^{(1\text{-bias})}$ [depending on sign bit]
Not 0, not all 1s	any	+/- (1.fraction)$\times 2^{(\text{exponent-bias})}$ [depending on sign bit]
All 1s	0	+/- infinity [depending on sign bit]
All 1s	not 0	NaN

Fig. 2-8. Interpretation of IEEE floating-point numbers.

2.6.2 ARITHMETIC ON FLOATING-POINT NUMBERS

Given the similarities between the IEEE floating-point representation and scientific notation, it is not surprising that the techniques used for floating-point arithmetic on computers are very similar to the techniques used to do arithmetic on decimal numbers that are expressed in scientific notation. A good example of this is floating-point multiplication.

To multiply two numbers using scientific notation, the mantissas of the numbers are multiplied and the exponents added. If the result of multiplying the mantissas is greater than 10, the product of the mantissas is shifted so that there is exactly one nonzero digit to the left of the decimal point, and the sum of the exponents is incremented as necessary to keep the value of the product the same. For example, to multiply 5×10^3 by 2×10^6, we multiply the mantissas ($5 \times 2 = 10$) and add the exponents ($3 + 6 = 9$) to get an initial result of 10×10^9. Since the mantissa of this number is greater than 10, we shift it down one position and add 1 to the exponent to get the final result of 1×10^{10}.

Computers multiply floating-point numbers using a very similar process, as illustrated in Fig. 2-9. The first step is to multiply the mantissas of the two numbers, using techniques analogous to those used to multiply decimal numbers, and to add their exponents. IEEE floating-point numbers use a biased representation for exponents, so adding the exponent fields of two floating-point numbers is slightly more complicated than adding two integers. To compute the sum of the exponents, the exponent fields of the two floating-point numbers are treated as integers and

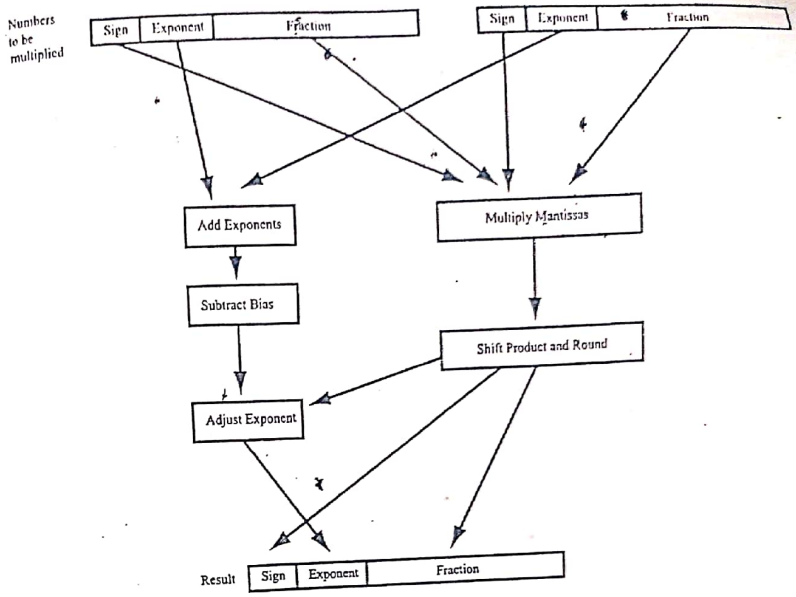

Fig. 2-9. Floating-point multiplication.

added, and the bias value is subtracted from the result. This gives the correct biased representation for the sum of the two exponents.

Once the mantissas have been multiplied, the result may need to be shifted so that only 1 bit remains to the left of the binary point (i.e., so that it fits the form $1.xxxx_2$), and the sum of the exponents incremented so that the value of the mantissa $\times 2^{\text{exponent}}$ remains the same. The product of the mantissas may also have to be rounded to fit within the number of bits allocated to the fraction field, since the product of two n-bit mantissas may require up to $2n$ bits to represent exactly. Once the mantissa has been shifted and rounded, the final product is assembled out of the product of the mantissas and the sum of the exponents.

EXAMPLE
Using single-precision floating-point numbers, multiply 2.5 by 0.75.

Solution

$2.5 =$ 0b0100 0000 0010 0000 0000 0000 0000 0000 (exponent field of 0b10000000, fraction field of 0b010 0000 0000 0000 0000 0000, mantissa of

$1.010\ 0000\ 0000\ 0000\ 0000\ 0000_2$). $0.75 = 0b0011\ 1111\ 0100\ 0000\ 0000\ 0000\ 0000\ 0000$ (exponent field of 0b01111110, fraction field of 0b100 0000 0000 0000 0000 0000, mantissa of $1.100\ 0000\ 0000\ 0000\ 0000\ 0000_2$). Adding the exponent fields directly and subtracting the bias gives a result of 0b01111111, the biased representation of 0. Multiplying the mantissas gives a result of $1.111\ 0000\ 0000\ 0000\ 0000\ 0000_2$, which converts into a fraction field of 0b111 0000 0000 0000 0000 0000, so the result is 0b0011 1111 1111 0000 0000 0000 0000 0000 $= 1.111_2 \times 2^0 = 1.875$.

Floating-point division is very similar to multiplication. The hardware computes the quotient of the mantissas and the difference between the exponents of the numbers being divided, adding the bias value to the difference between the exponent fields of the two numbers to get the correct biased representation of the result. The quotient of the mantissas is then shifted and rounded to fit within the fraction field of the result.

Floating-point addition requires a different set of computations, which are illustrated in Fig. 2-10. As with adding numbers in scientific notation, the first step is to shift one of the inputs until both inputs have the same exponent. In adding floating-point numbers, the number with the smaller exponent is right-shifted. For example, in adding $1.01_2 \times 2^3$ and $1.001_2 \times 2^0$, the smaller value is shifted to become $0.001001_2 \times 2^3$. Shifting the number with the smaller exponent allows the use of techniques that retain just enough information about the less-significant bits of the smaller number to perform rounding, reducing the number of bits that actually have to be added.

Once the inputs have been shifted, their mantissas are added, and the result shifted if necessary. Finally, the result is rounded to fit in the fraction field, and the computation is complete. Floating-point subtraction uses the same process, except that the difference between, rather than the sum of, the shifted mantissas is computed.

EXAMPLE

Using single-precision floating-point numbers, compute the sum of 0.25 and 1.5.

Solution

$0.25 = 0b0011\ 1110\ 1000\ 0000\ 0000\ 0000\ 0000\ 0000$ (1.0×2^{-2}).
$1.5 = 0b0011\ 1111\ 1100\ 0000\ 0000\ 0000\ 0000\ 0000$ (1.5×2^0).

To add these numbers, we shift the one with the smaller exponent (0.25) to the right until both exponents are the same (two places in this case). This gives mantissas of 1.100 0000 0000 0000 0000 0000 and 0.010 000 0000 0000 0000 0000 for the two numbers (including the assumed 1s in the values to be shifted.) Adding these two mantissas gives a result of 1.110 0000 0000 0000 0000 0000 $\times 2^0$ (the exponent of the input with the larger exponent) = 1.75. The single-precision representation of the full result is 0b0011 1111 1110 0000 0000 0000 0000 0000.

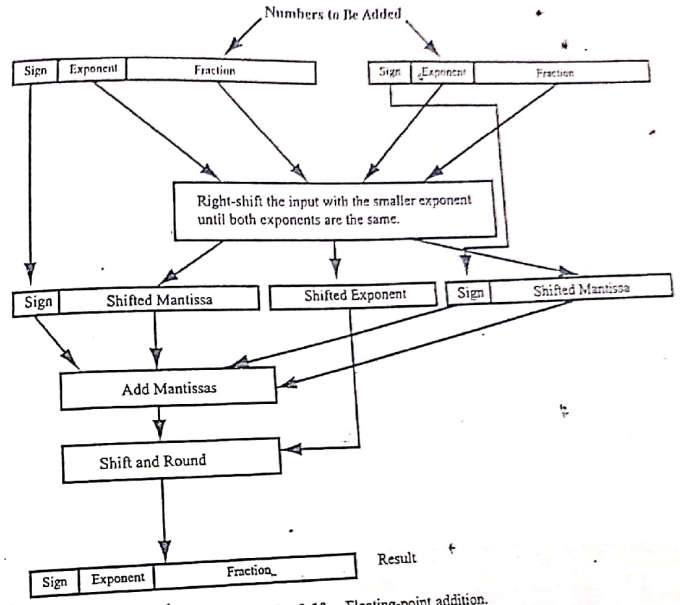

Fig. 2-10. Floating-point addition.

2.7 Summary

This chapter has described techniques that computer systems use to represent and manipulate data. In general, computers use two levels of abstraction in representing data, one layered on top of the other. The lower-level abstraction is the digital signaling convention that maps the numbers 0 and 1 onto an analog electrical signal. A variety of signaling conventions are used, but the most common one defines a logical 0 as a range of voltages near the ground voltage of the system, and a logical 1 as a range of voltages near the supply voltage of the system (V_{dd}). Different ranges are defined for the voltage levels that may be generated by a circuit and the input values that are guaranteed to be interpreted as a 0 or 1 by another circuit, to protect the circuit against electrical noise.

The second-level abstraction defines how groups of bits are used to represent integer and noninteger numbers. Positive integer numbers are represented using a

place-value system analogous to the decimal system. Sign-magnitude or two's-complement representations are used to represent negative integers, with two's-complement being the more common representation because it allows simple implementations of both addition and subtraction.

Noninteger values are represented using floating-point numbers. Floating-point numbers are similar to scientific notation, representing numbers as a mantissa and an exponent. This allows a very wide range of values to be represented in a small number of bits, although not all of the values in the range can be represented exactly. Multiplication and division can be implemented simply on floating-point numbers, while addition and subtraction are more complicated, since the mantissa of one of the numbers must be shifted to make the exponents of the two numbers equal.

Using a combination of integer and floating-point numbers, programs can perform a wide variety of arithmetic operations. However, all of these representations have their limitations. The range of integers that a computer can represent is limited by its bit width, and attempts to perform computations that generate results outside of this range will generate incorrect results. Floating-point numbers also have a limited range, although the mantissa-exponent representation makes this range much larger. A more significant limitation of floating-point numbers comes from the fact that they can only represent numbers to a limited number of significant digits, because of the limited number of bits used to represent the mantissa of each number. Computations that require more accuracy than the floating-point representation allows will not operate correctly.

Solved Problems

Signaling Conventions (I)

2.1. Suppose a digital system has $V_{DD} = 3.3\,V$, $V_{IL} = 1.2\,V$, $V_{OL} = 0.7\,V$, $V_{IH} = 2.1\,V$, $V_{OH} = 3.0\,V$.
What is the noise margin of this signaling convention?

Solution

The noise margin is the lesser of the differences between the valid output and input levels for a 0 or 1. For this signaling convention $|V_{OL} - V_{IL}| = 0.5\,V$, and $|V_{OH} - V_{IH}| = 0.9\,V$. Therefore, the noise margin for this signaling convention is 0.5 V, indicating that the value of any valid output signal from a logic gate can change by up to 0.5 V due to noise in the system without becoming an invalid value.

Signaling Conventions (II)

2.2. Suppose you were told that a given signaling convention had $V_{DD} = 3.3\,V$, $V_{IL} = 1.0\,V$, $V_{OL} = 1.2\,V$, $V_{IH} = 2.1\,V$, $V_{OH} = 3.0\,V$.
Why would this signaling convention be a bad idea?

Solution

In this signaling convention, $V_{IH} > V_{OL} > V_{IL}$. This means that a logic gate is allowed to generate an output value that lies in the forbidden region between V_{IH} and V_{IL}. Such an output value is not guaranteed to be interpreted as either a 0 or a 1 by any gate that receives it as an input. Another way of expressing this is to say that this signaling convention allows a gate that is trying to output a 0 to generate an output voltage that will not be interpreted as a 0 by the input of another gate, even if there is no noise in the system.

Binary Representation of Positive Integers (I)

2.3. Show how the following integers would be represented by a system that uses 8-bit unsigned integers.

a. 37
b. 89
c. 4
d. 126
e. 298

Solution

a. $37 = 32 + 4 + 1 = 2^5 + 2^2 + 2^0$. Therefore, the 8-bit unsigned binary representation of 37 is 0b00100101.
b. $89 = 64 + 16 + 8 + 1 = 2^6 + 2^4 + 2^3 + 2^0 = 0b01011001$
c. $4 = 2^2 = 0b00000100$
d. $126 = 64 + 32 + 16 + 8 + 4 + 2 = 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 = 0b01111110$
e. This is a trick question. The maximum value that can be represented by an 8-bit unsigned number is $2^8 - 1 = 255$. 298 is greater than 255, so it cannot be represented by an 8-bit unsigned binary number.

Binary Representation of Positive Integers (II)

2.4. What is the decimal value of the following unsigned binary integers?

a. 0b1100
b. 0b100100
c. 0b11111111

Solution

a. $0b1100 = 2^3 + 2^2 = 8 + 4 = 12$
b. $0b100100 = 2^5 + 2^2 = 32 + 4 = 36$
c. $0b11111111 = 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$

Hexadecimal Notation (I)

2.5. What are the hexadecimal representations of the following integers?

a. 67
b. 142
c. 1348

Solution

a. $67 = (4 \times 16) + 3 = 0x43$
b. $142 = (8 \times 16) + 14 = 0x8e$
c. $1348 = (5 \times 16 \times 16) + (4 \times 16) + 4 = 0x544$

Hexadecimal Notation (II)

2.6. What are the decimal values of the following hexadecimal numbers?
a. 0x1b
b. 0xa7
c. 0x8ce

Solution

a. $0x1b = (1 \times 16) + 11 = 27$
b. $0xa7 = (10 \times 16) + 7 = 167$
c. $0x8ce = (8 \times 16 \times 16) + (12 \times 16) + 14 = 2254$

Addition of Unsigned Integers

2.7. Compute the sums of the following pairs of unsigned integers:
a. 0b11000100 + 0b00110110
b. 0b00001110 + 0b10101010
c. 0b11001100 + 0b00110011
d. 0b01111111 + 0b00000001

Solution

(Note that all of these problems can be checked by converting the inputs and outputs into decimal.)
a. 0b11111010
b. 0b10111000
c. 0b11111111
d. 0b10000000

Multiplication of Unsigned Integers

2.8. Compute the product of the following pairs of unsigned integers. Generate the full 8-bit result.
a. 0b1001 × 0b0110
b. 0b1111 × 0b1111
c. 0b0101 × 0b1010

Solution

a. $0b1001 \times 0b0110 = (0b1001 \times 0b100) + (0b1001 \times 0b10) = 0b100100 + 0b10010 = 0b00110110$
b. 0b11100001
c. 0b00110010

Number of Bits Required

2.9. How many bits are required to represent the following decimal numbers as unsigned binary integers?
a. 12
b. 147
c. 354
d. 1497

Solution

a. 12 is greater than $2^3 - 1$ and less than $2^4 - 1$, so 12 cannot be represented in a 3-bit unsigned integer, but can be represented in a 4-bit binary integer. Therefore 4 bits are required.
b. $2^7 - 1 < 147 < 2^8 - 1$, so 8 bits are required.
c. $2^8 - 1 < 354 < 2^9 - 1$, so 9 bits are required.
d. $2^{10} - 1 < 1497 < 2^{11} - 1$, so 11 bits are required.

Ranges of Binary Representations

2.10. What are the largest and smallest integers representable in 4-, 8-, and 16-bit values using
a. Unsigned binary representation
b. Sign-magnitude binary representation
c. Two's-complement representation

Also, why are your answers to b and c different?

Solution

a. In any unsigned representation 0 is the smallest representable value. The largest value representable in an n-bit unsigned binary integer is $2^n - 1$, giving maximum representable values of 15, 255, and 65,535 for 4-, 8-, and 16-bit unsigned integers.
b. Sign-magnitude representations use 1 bit to record the sign of a number, allowing them to represent values from $-(2^{n-1} - 1)$ to $2^{n-1} - 1$. This gives a range of -7 to $+7$ for 4-bit quantities, -127 to $+127$ for 8-bit quantities, and $-32,767$ to $+32,767$ for 16-bit quantities.
c. n-bit two's-complement integers can represent values from -2^{n-1} to $2^{n-1} - 1$. Therefore, 4-bit two's-complement integers can represent values from -8 to $+7$, 8-bit numbers can represent values from -128 to $+127$, and 16-bit numbers can represent values from $-32,768$ to $+32,767$.

Sign-magnitude representations have two representations for zero, while two's-complement representations have only one. This gives two's-complement representations the ability to represent one more value than sign-magnitude representations with the same number of bits.

Sign-Magnitude Representation

2.11. Convert the following decimal numbers to 8-bit sign-magnitude representation:
a. 23
b. −23
c. −43
d. −65

Solution

a. In sign-magnitude representation, positive integers are represented in the same way as they are in unsigned binary representation, except that the high bit of the representation is reserved for the sign bit. Therefore, the 8-bit sign-magnitude representation of 23 is 0b00010111.
b. To get the sign-magnitude representation of −23, we simply set the sign bit of the representation of +23 to 1, giving 0b10010111.
c. 0b10110000
d. 0b11000001

Two's-Complement Notation

2.12. Give the 8-bit two's-complement representation of the quantities from Problem 2.11.

Solution

a. Like sign-magnitude representation, the two's-complement representation of a positive number is the same as the unsigned representation of that number, giving 0b00010111 as the 8-bit two's-complement representation of 23.
b. To negate a number in two's-complement representation, we invert all of the bits of its representation and add 1 to the result, giving 0b11101001 as the 8-bit two's-complement representation of −23.
c. 0b11010000
d. 0b10111111

Sign-Extension

2.13. Give the 8-bit representation of the numbers 12 and −18 in sign-magnitude and two's-complement notation, and show how these representations are sign-extended to give 16-bit representations in each notation.

Solution

The 8-bit sign-magnitude representations of 12 and −18 are 0b00001100 and 0b10010010, respectively. To sign-extend a sign-magnitude number, the sign bit is copied into the most significant bit of the new representation and the sign bit of the old representation is cleared, giving 16-bit sign-magnitude representations of 0b0000000000001100 for 12 and 0b1000000000010010 for −18.

The 8-bit two's-complement representations of 12 and −18 are 0b00001100 and 0b11101110. Two's-complement numbers are sign-extended by copying the high bit of the number into the additional bits of the new representation, giving 16-bit representations of 0b0000000000001100 and 0b1111111111101110, respectively.

Math on Two's-Complement Integers

2.14. Using 8-bit two's-complement integers, perform the following computations:
a. $-34 + (-12)$
b. $17 - 15$
c. $-22 - 7$
d. $18 - (-5)$

Solution

a. In two's-complement notation, −34 = 0b11011110 and −12 = 0b11110100. Adding these, we get 0b11010010 (remember that the 9th bit is dropped when two 8-bit two's-complement numbers are added). This is −46, the correct answer.
b. Here, we can take advantage of the fact that we're using two's-complement notation to transform 17 − 15 into 17 + (−15), or 0b00010001 + 0b11110001 = 0b00000010 = 2.
c. Again, transform this to −22 + (−7) to get the result of 0b11100011 = −29.
d. This transforms into 18 + 5 = 0b00010111 = 23.

Comparing Integer Representations

2.15. Which of the two integer representations described in this chapter (sign-magnitude and two's-complement) would be better suited to the following situations:
a. When it is critical that the hardware to negate a number be as simple as possible.
b. When most of the mathematical operations performed will be additions and subtractions.
c. When most of the mathematical operations performed will be multiplications and divisions.
d. When it is essential that it be as easy as possible to detect whether a number is positive or negative.

Solution

a. In this case, sign-magnitude representation would be better, because negating a number simply requires inverting the sign bit.
b. Two's-complement numbers allow simpler hardware for addition and subtraction than sign-magnitude numbers. With two's-complement numbers, no additional hardware is required to add positive and negative numbers—treating the numbers as unsigned quantities and adding them gives the correct two's-complement result. Subtraction can be implemented by negating the second operand and then adding.

   In contrast, sign-magnitude representations require different hardware to perform subtractions or to add positive and negative numbers, making this representation more expensive if most of the computations to be performed are additions and/or subtractions.
c. Sign-magnitude representation is better in this case, because multiplication and division can be implemented by treating the magnitude portions of the numbers as unsigned integers and then determining the sign of the result by examining the sign bits of the two values.
d. In this case, the two representations are very close. In general, the sign of a number in either representation can be determined by examining the high bit of the number—if the high bit is 1, the number is negative. The exception to this is when the value of the number is 0. Sign-magnitude representations have two representations of zero, one with the sign bit 1, and one with it equal to 0, while there is only one representation of zero in two's-complement notation.

   In summary, the two representations are equivalent if it is unimportant whether zero values detect as positive or negative. If it is important to determine whether the number is positive, negative, or zero, two's-complement numbers are slightly better.

Rounding

2.16. Using round-to-nearest, round the following decimal quantities to three significant digits:

a. 1.234
b. 8940.999
c. 179.5
d. 178.5

Solution

a. 1.23
b. 8940 (significant digits don't have to be to the right of the decimal point)
c. 180 (round to even)
d. 178 (round to even)

Floating-Point Representations (I)

2.17. Convert the following quantities to IEEE single-precision floating-point:
a. 128
b. −32.75
c. 18.125
d. 0.0625

Solution

a. $128 = 2^7$, giving an exponent field of 134, a sign bit of 0, and a fraction field of 0 (because of the assumed 1). Therefore, 128 = 0b0100 0011 0000 0000 0000 0000 0000 0000 in single-precision floating-point format.
b. $-32.75 = -100000.11_2$ or $-1.0000011_2 \times 2^5$. The single-precision floating-point representation of −32.75 is 0b1100 0010 0000 0011 0000 0000 0000 0000.
c. $18.125 = 10010.001_2$ or $1.0010001_2 \times 2^4$, giving a single-precision floating-point representation of 0b0100 0001 1001 0001 0000 0000 0000 0000.
d. $0.0625 = 0.0001_2$ or 1×2^{-4}, giving a single-precision floating-point representation of 0b0011 1101 1000 0000 0000 0000 0000 0000.

Floating-Point Representations (II)

2.18. What values are represented by the following IEEE single-precision floating-point numbers?
a. 0b1011 1101 0100 0000 0000 0000 0000 0000
b. 0b0101 0101 0110 0000 0000 0000 0000 0000
c. 0b1100 0001 1111 0000 0000 0000 0000 0000
d. 0b0011 1010 1000 0000 0000 0000 0000 0000

Solution

a. For this number, the sign bit = 1, exponent field = 122, so exponent = −5. Fraction field = 100 0000 0000 0000 0000 0000, so this binary string represents $-1.1_2 \times 2^{-5} = -0.046875$
b. Here, we have a sign bit of 0, exponent field of 170, so the exponent is 43, and a fraction field of 110 0000 0000 0000 0000 0000, so the value of this number is $1.11_2 \times 2^{43} = 1.539 \times 10^{13}$ (to 4 significant digits)
c. $-1.111_2 \times 2^4 = -30$
d. $1.0_2 \times 2^{-10} = 0.0009766$ (to 4 significant digits)

NaNs and Denormalized Numbers

2.19. For each of the IEEE single-precision values below, explain what type of number (normalized, denormalized, infinity, 0, or NaN) they represent. If the quantity has a numeric value, give it.
a. 0b0111 1111 1000 1111 0000 1111 0000 0000
b. 0b0000 0000 0000 0000 0000 0000 0000 0000
c. 0b0100 0010 0100 0000 0000 0000 0000 0000
d. 0b1000 0000 0100 0000 0000 0000 0000 0000
e. 0b1111 1111 1000 0000 0000 0000 0000 0000

Solution

a. The exponent field of this number is all 1s, and its fraction field is not 0, so it's a NaN.
b. This number has an exponent field of 0, a sign bit of 0, and a fraction field of 0, which is the IEEE floating-point representation for +0.
c. This number has an exponent field of 132 and a fraction field of 100 0000 0000 0000 0000 0000. Since its exponent field is neither all 0s or all 1s, it represents a normalized number, with a value of $1.1_2 \times 2^5 = 48$.
d. This number has an exponent field of all 0s, and its fraction field is nonzero, so it's a denormalized number. Its value is $-0.1_2 \times 2^{-126} = -2^{-127} = -5.877 \times 10^{-39}$ (to four significant digits).
e. The exponent field of this number is all 1s, its fraction field is 0, and its sign bit is 1, so it represents −infinity.

Arithmetic on Floating-Point Numbers

2.20. Use IEEE single-precision floating-point numbers to compute the following quantities:
a. 32 × 16
b. 147.5 + 0.25
c. 0.125 × 8
d. 13.25 + 4.5

Solution

a. $32 = 2^5$ and $16 = 2^4$, so the floating-point representations of these numbers are 0b0100 0010 0000 0000 0000 0000 0000 0000 and 0b0100 0001 1000 0000 0000 0000 0000 0000. To multiply them, we convert their fraction fields into a mantissa and multiply, while adding the exponent fields and subtracting the bias from the sum. This gives a resulting exponent field of 10001000 and a fraction field of 000 0000 0000 0000 0000 0000 once we remove the assumed 1 from the product of the mantissas. The resulting floating-point number is 0b0100 0100 0000 0000 0000 0000 0000 0000 $= 2^9 = 512$.
b. $147.5 = 1.00100111_2 \times 2^7$ = 0b0100 0011 0001 0011 1000 0000 0000 0000. $0.25 = 1.0_2 \times 2^{-2}$ = 0b0011 1110 1000 0000 0000 0000 0000 0000. Shifting the number with the smaller exponent (.25) to the right to make the exponents of both numbers equal gives $0.25 = 0.000000001_2 \times 2^7$. Adding the mantissas gives a sum of $1.001001111_2 \times 2^7$ = 0b0100 0011 0001 0011 1100 0000 0000 0000.
c. Converting these numbers to floating-point gives representations of 0b0011 1110 0000 0000 0000 0000 0000 0000 for 0.125, and 0b0100 0001 0000 0000 0000 0000 0000 0000 for 8. Multiplying the mantissas and adding the exponents gives a result of 0b0011 1111 1000 0000 0000 0000 0000 0000 = 1.

d. $13.25 = 1.10101_2 \times 2^3 = 0b0100\ 0001\ 0101\ 0100\ 0000\ 0000\ 0000\ 0000$. $4.5 = 1.001_2 \times 2^2 = 0b0100\ 0000\ 1001\ 0000\ 0000\ 0000\ 0000\ 0000$. Shifting 4.5 to the right one place to make the exponents the same gives $4.5 = 0.1001_2 \times 2^3$. Adding mantissas gives a result of $10.00111_2 \times 2^3$, so we have to shift this down one to get $1.000111_2 \times 2^4$. The single-precision floating-point representation of this is 0b0100 0001 1000 1110 0000 0000 0000 0000.

CHAPTER 3

Computer Organization

3.1 Objectives

The last two chapters laid the groundwork for our discussion of computer architecture by explaining how computer architects describe and analyze performance, and how computers represent and manipulate real-world values. In this chapter, we begin covering computer architecture *per se* by describing the basic building blocks that make up conventional computer systems: processors, memory, and I/O. We will also briefly describe how programs are represented internally by computer systems and how operating systems schedule programs and control the physical devices that make up a computer.

After completing this chapter, you should

1. Understand the basic concepts behind processors, memory, and I/O devices, and be able to describe their functions
2. Be familiar with stored-program computer architecture
3. Understand the basic functions of operating systems

3.2 Introduction

As shown in Fig. 3-1, most computer systems can be divided into three subsystems: the processor, the memory, and the input/output (I/O) subsystem. The processor is responsible for executing programs, the memory provides storage space for programs and the data they reference, and the I/O subsystem allows the processor and memory to control devices that interact with the outside world or store data, such as the CD-ROM, hard disk, and video card/monitor shown in the figure.

In most systems, the processor has a single data bus that connects to a switch module, such as the PCI bridge found in many PC systems, although some processors integrate the switch module onto the same integrated circuit as the processor to reduce the number of chips required to build a system and thus the

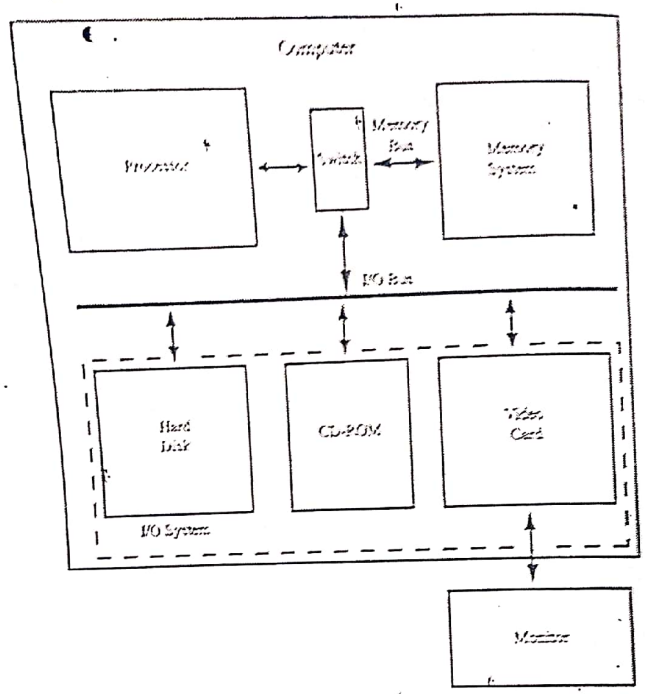

Fig. 3-1. Computer organization.

system cost. The switch communicates with the memory through a *memory bus*, a dedicated set of wires that transfer data between these two systems. A separate *I/O bus* connects the switch to the I/O devices. Separate memory and I/O buses are used because the I/O system is generally designed for maximum flexibility, to allow as many different I/O devices as possible to interface to the computer, while the memory bus is designed to provide the maximum-possible bandwidth between the processor and the memory system.

3.3 Programs

Programs are sequences of instructions that tell the computer what to do, although the computer's view of the instructions that make up a given program is often very

different from the program writer's view. To the computer, a program is made up of a sequence of numbers that represent individual operations. These operations are known as *machine instructions* or just *instructions*, and the set of operations that a given processor can execute is known as its *instruction set*.

Almost all computers in use today are *stored-program computers* that represent programs as numbers that are stored in the same address space as data.[1] The stored-program abstraction (representing instructions as numbers stored in memory) was one of the major breakthroughs in early computer architecture. Prior to this breakthrough, many computers were programmed by setting switches or rewiring circuit boards to define the new program, which required a great deal of time and was prone to errors.

The stored-program abstraction provides two major advantages over previous approaches. First, it allows programs to be easily stored and loaded into the machine. Once a program has been developed and debugged, the numbers that represent its instructions can be written out onto a storage device, allowing the program to be loaded back into memory at some point in the future. On early systems, the most common storage device was punched cards or paper tape. Modern systems generally use magnetic media, such as hard disks. Being able to store programs like data eliminates errors in reloading the program (assuming that the device the program is stored on is error-free), while requiring a human to reenter the program each time it is used generally introduces errors that have to be corrected before the program runs correctly—imagine having to debug your word processor each time you started it up!

Second, and perhaps even more significant, the stored-program abstraction allows programs to treat themselves or other programs as data. Programs that treat themselves as data are called *self-modifying programs*. In a self-modifying program, some of the instructions in a program compute other instructions in the program. Self-modifying programs were common on early computers, because they were often faster than non-self-modifying programs, and because early computers implemented a small number of instructions, making some operations hard to do without self-modifying code. In fact, self-modifying code was the only way to implement a conditional branch on at least one early computer—the instruction set did not provide a conditional branch operation, so programmers implemented conditional branches by writing self-modifying code that computed the destination addresses of unconditional branch instructions as the program executed.

Self-modifying code has become less common on more-modern machines because changing the program during execution makes it harder to debug. As computers have become faster, ease of program implementation and debugging has become more important than the performance improvements achievable through self-modifying code in most cases. Also, memory systems with caches (discussed in Chapter 9) make self-modifying code less efficient, reducing the performance improvements that can be gained by using this technique.

[1] Stored-program computers are also sometimes called von Neumann computers, after John von Neumann, one of the developers of this concept.

3.3.1 PROGRAM DEVELOPMENT TOOLS

Programs that treat other programs as data, however, are very common and most program development tools fall into this category. These include *compilers* that convert programs from high-level languages such as C or FORTRAN into assembly language, *assemblers* that convert assembly-language instructions into the numeric representation used by the processor, and *linkers* that join multiple machine language programs into a single executable file. Also included in this category are *debuggers*, programs that display the state of another program as it executes to allow programmers to track the progress of a program and find errors.

The first stored-program computers were programmed directly in *machine language*, the numeric instruction representation used internally by the processor. To write a program, a programmer would determine the sequence of machine instructions needed to generate the correct result and would then enter the numbers that represented these instructions into the computer. This was a very time-consuming process, and it resulted in large numbers of programming errors.

The first step in simplifying program development came when assemblers were developed, allowing programmers to program in *assembly language*. In assembly language, each machine instruction has a text representation (such as ADD, SUB, or LOAD) that represents what it does, and programs are written using these assembly-language instructions. Once the program has been written, the programmer runs the assembler to convert the assembly-language program into a machine language program that can be executed on the computer. Figure 3-2 shows an example of an assembly-language instruction and the machine-language instruction that might be generated from it.

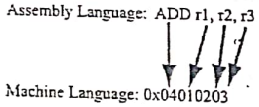

Fig. 3-2. Assembly language.

Using assembly language made programming much easier by allowing programmers to use an instruction format that is more easily understood by humans. Programming was still extremely tedious because of the small amount of work done by each instruction and because the instructions available to the programmer differed from machine to machine. If a programmer wanted to run a program on a different type of computer, the program had to be completely rewritten in the new computer's assembly language.

High-level languages, such as FORTRAN, COBOL, and C, were developed to address these issues. A high-level language instruction can specify much more work than an assembly-language instruction. Studies have shown that the average number of instructions written and debugged per day by a programmer is relatively independent of the language used. Since high-level languages allow programs to be written in many fewer instructions than assembly language, the time to

implement a program in a high-level language is typically much less than the time to implement the program in assembly language.

Another advantage of writing programs in high-level languages is that they are more portable than programs written in assembly language or machine language. Programs written in high-level languages can be converted for use on different types of computer by recompiling the program using a compiler for the new computer. In contrast, assembly-language programs must be completely rewritten for the new system, which takes much more time.

The problem with high-level languages is that computers cannot execute high-level language instructions directly. Instead, a program called a *compiler* is used to convert the program into assembly language, which is then converted into machine language by an assembler. Figure 3-3 illustrates the process of developing and executing a program in a high-level language.

An alternative to compiling a program is to use an *interpreter* to execute the high-level language version of the program. Interpreters are programs that take high-level language programs as inputs and perform the steps defined by each instruction in the

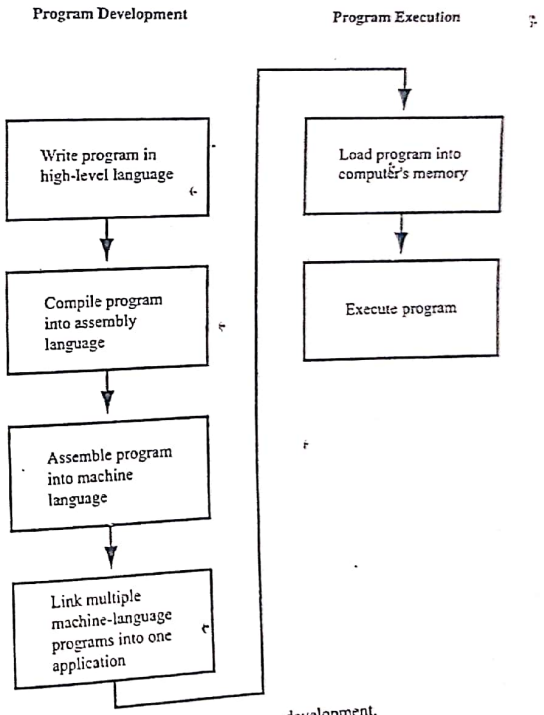

Fig. 3-3. Program development.

high-level language program, generating the same result as compiling the program and then executing the compiled version. Interpreted programs tend to be much slower than compiled programs, because the interpreter has to examine each instruction in the source program as it occurs and then jump to a routine that performs the instruction. In many ways, this is similar to the compiler's task of determining the assembly-language instruction sequence that implements a given high-level language instruction, except that the interpreter must reinterpret each high-level language instruction each time it executes. If a program contains a loop that executes 10,000 times, the interpreter must interpret the loop 10,000 times, but the compiler only needs to compile it once.

Given their speed disadvantages, interpreters are much less common than compiled programs. They are mainly used in cases where it is important to be able to run a program on multiple different types of computer without recompiling it. In this case, using an interpreter allows each type of computer to execute the high-level language version of the program directly.

Compilers and assemblers have very different tasks. In general, there is a one-to-one mapping between assembly-language instructions and machine-language instructions, so all the assembler has to do is convert each instruction from one format to the other. A compiler, on the other hand, has to determine a sequence of assembly-language instructions that implement a high-level language program as efficiently as possible. Because of this, the execution time of a program written in a high-level language depends a great deal on how good the compiler is, while the execution time of a program originally written in assembly language depends completely on the set of instructions written by the programmer.

3.4 Operating Systems

On workstations, PCs, and mainframe computers, the *operating system* is responsible for managing the physical resources of the system, loading and executing programs, and interfacing with users. *Embedded systems*—computers designed for one specific task, such as controlling an appliance—often do not have operating systems because they only execute one program. The operating system is simply another program, one that knows about all the hardware in the computer, with one exception—it runs in *privileged* (or supervisor) mode, which allows it access to physical resources that user programs cannot control and gives it the ability to start or stop the execution of user programs.

3.4.1 MULTIPROGRAMMING

Most computer systems support *multiprogramming* (also called multitasking), a technique that allows the system to present the illusion that multiple programs are running on the computer simultaneously, even though the system may only have one processor. In a multiprogrammed system, user programs do not need to know which

other programs are running on the system at the same time they are, or even how many other programs there are. The operating system and the hardware provide *protection* for programs, preventing any program from accessing another program's data unless the two programs have specifically arranged to access each other's data. Many multiprogrammed computers are also *multiuser* and allow more than one person to be logged in to the computer at once. Multiuser systems require that the operating system not only protect programs from accessing each other's data but prevent users from accessing data that is private to other users.

A multiprogrammed operating system presents the illusion that multiple programs are running simultaneously by switching between programs very rapidly, as illustrated in Fig. 3-4. Each program is allowed to execute for a fixed amount of time, known as a *timeslice*. When a program's timeslice ends, the operating system stops it, removes it from the processor, and loads another program into the processor. This process is known as a *context switch*. To do a context switch, the operating system copies the contents of the currently running program's register file (sometimes called the program's *context*) into memory, and then copies the contents of the next program's register file out of memory and into the register file. Programs cannot tell that a context switch has been performed—to them, it looks like they have been continuously running on the processor.

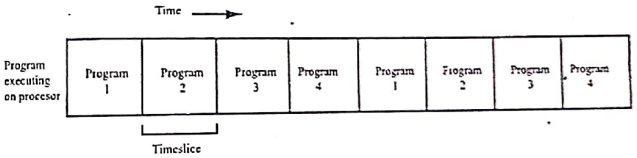

Fig. 3–4. Multiprogrammed system.

Many computers context-switch 60 times per second, making timeslices 1/60th of a second, although some recent systems have started to context-switch more frequently. This has caused problems with programs that expect timeslices to be 1/60th of a second and use that information to time events or determine performance.

By context-switching 60 or more times per second, a computer can give each program an opportunity to execute sufficiently frequently that the system can prevent the illusion that a moderate number of programs are executing simultaneously on the system. Obviously, as the number of programs on the system increases, this illusion breaks down—if the system is executing 120 programs, each program may only get a timeslice once every 2 seconds, which is enough of a delay for humans to notice. Multiprogramming can also increase the running time of applications, because the resources of the system are shared among all of the programs running on it.

3.4.2 PROTECTION

One of the main requirements of a multiprogrammed operating system is that it provide *protection* between programs running on the computer. This essentially means that the result of any program running on a multiprogrammed computer must be the same as if the program was the only program running on the computer. Programs must not be able to access other programs' data and must be confident that their data will not be modified by other programs. Similarly, programs must not be able to interfere with each other's use of the I/O subsystem.

Providing protection in a multiprogrammed or multiuser system requires that the operating system control the physical resources of the computer, including the processor, the memory, and the I/O devices. Otherwise, user programs could access any of the memory or other storage on the computer, gaining access to data that belongs to other programs or other users. This also allows the operating system to prevent more than one program from accessing an I/O device, such as a printer, at one time.

One technique that operating systems use to protect each program's data from other programs is *virtual memory*, which is described in detail in Chapter 10. Essentially, virtual memory allows each program to operate as if it were the only program running on the computer by *translating* memory addresses that the program references into the addresses used by the memory system. As long as the virtual memory system ensures that two programs' addresses don't translate into the same address, programs can be written as if they were the only program running on the machine, since no program's memory references will access data from another program.

3.4.3 PRIVILEGED MODE

To ensure that the operating system is the only program that can control the system's physical resources, it executes in *privileged mode*, while user programs execute in *user mode* (sometimes called unprivileged mode). Certain tasks, such as accessing an I/O device, performing context switches, or performing memory allocation, require that a program be in privileged mode. If a user-mode program tries to perform one of these tasks, the hardware prevents it from doing so and signals an error. When user-mode programs want to do something that requires privileged mode, they send a request to the operating system, known as a *system call*, that asks the operating system to do the operation for them. If the operation is something that the user program is allowed to do, the operating system performs the operation and returns the result to the user. Otherwise, it signals an error.

Because it controls the physical resources of the computer, the operating system is also responsible for the low-level user interface. When a user presses a key or otherwise sends input to the computer, the operating system is responsible for determining which program should receive the input and sending the input value to that program. Also, when a program wants to display some information for the user, such as printing a character on the monitor, it executes a system call to request that the operating system display the data.

3.5 Computer Organization

Figure 3-1 presented a high-level block diagram of a typical computer system. In this section, we present a brief introduction to each of the main subsystems: processor, memory, and I/O. The goal of this discussion is to give the reader enough high-level understanding of each subsystem to prepare them for later chapters that discuss each of these subsystems in more detail.

3.5.1 THE PROCESSOR

The processor is responsible for actually executing the instructions that make up programs and the operating system. As shown in Fig. 3-5, processors are made up of several building blocks: execution units, register files, and control logic. The execution units contain the hardware that executes instructions. This includes the hardware that fetches and decodes instructions, as well as the arithmetic logic units (ALUs) that perform actual computation. Many processors contain separate execution units for integer and floating-point computations because very different

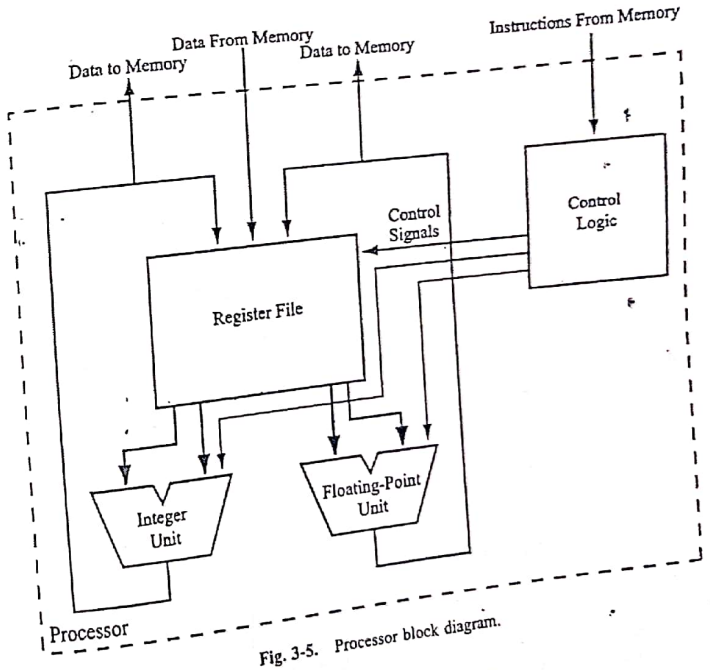

Fig. 3-5. Processor block diagram.

hardware is required to handle these two data types. In addition, as we will see in Chapter 7, modern processors often use multiple execution units to execute instructions in parallel to improve performance.

The *register file* is a small storage area for data that the processor is using. Values stored in the register file can be accessed more quickly than data stored in the memory system, and register files generally support multiple simultaneous accesses. This allows an operation, such as an addition, to read all of its inputs from the register file at the same time, rather than having to read them one at a time. As we will see in Chapter 4, different processors access and use their register files in very different ways, but virtually all processors have a register file of some sort.

As might be guessed from its name, the control logic controls the rest of the processor, determining when instructions can be executed and what operations are required to execute each instruction. In early processors, the control logic was a very small fraction of the processor hardware compared to the ALUs and the register file, but the amount of control logic required has grown dramatically as processors have become more complex, making this one of the more difficult parts of a processor to design.

3.5.2 THE MEMORY SYSTEM

The memory system acts as a storage receptacle for the data and programs used by the computer. Most computers have two types of memory: *read-only memory* (ROM) and *random-access memory* (RAM). As its name suggests, the contents of the read-only memory cannot be modified by the computer but may be read. In general, the ROM is used to hold a program that is executed automatically by the computer every time it is turned on or reset. This program is called the bootstrap, or "boot" loader, and instructs the computer to load its operating system off of its hard disk or other I/O device. The name of this program comes from the idea that the computer is "pulling itself up by its own bootstraps" by executing a program that tells it how to load its own operating system.

The random-access memory, on the other hand, can be both read and written, and is used to hold the programs, operating system, and data required by the computer. RAM is generally volatile, meaning that it does not retain the data stored in it when the computer's power is turned off. Any data that needs to be stored while the computer is off must be written to a permanent storage device, such as a hard disk.

Memory (both RAM and ROM) is divided into a set of storage locations, each of which can hold 1 byte (8 bits) of data. The storage locations are numbered, and the number of a storage location (called its *address*) is used to tell the memory system which location the processor wants to reference. One of the important characteristics of a computer system is the width of the addresses it uses, which limits the amount of memory that the computer can address. Most current computers use either 32-bit or 64-bit addresses, allowing them to access either 2^{32} or 2^{64} bytes of memory.

Until Chapter 9, we will be using a simple random-access model of memory, in which all memory operations take the same amount of time. Our memory system will support two operations: load and store. Store operations take two operands, a value to be stored and the address that the value should be stored in. They place the

specified value in the memory location specified by the address. Load operations take an operand that specifies the address containing the value to be loaded and return the contents of that memory location into their destination.

Using this model, the memory can be thought of as functioning similar to a large sheet of lined paper, where each line on the page represents a 1-byte storage location. To write (store) a value into the memory, you count down from the top of the page until you reach the line specified by the address, erase the value written on the line, and write in the new value. To read (load) a value, you count down from the top of the page until you reach the line specified by the address, and read the value written on that line.

Most computers allow more than 1 byte of memory to be loaded or stored at one time. Generally, a load or store operation operates on a quantity of data equal to the system's bit width, and the address sent to the memory system specifies the location of the lowest-addressed byte of data to be loaded or stored. For example, a 32-bit system loads or stores 32 bits (4 bytes) of data with each operation into the 4 bytes that start with the operation's address, so a load from location 424 would return a 32-bit quantity containing the bytes in location 424, 425, 426, and 427. To simplify the design of the memory system, some computers require loads and stores to be "aligned," meaning that the address of a memory reference must be a multiple of the size of the data being loaded or stored, so a 4-byte load must have an address that is a multiple of 4, an 8-byte store must have an address that is a multiple of 8, and so on. Other systems allow unaligned loads and stores, but take significantly longer to complete such operations than aligned loads.

An additional issue with multibyte loads and stores is the order in which the bytes are written to memory. There are two different ordering schemes that are used in modern computers: *little endian* and *big endian*. In a little-endian system, the least-significant (smallest value) byte of a word is written into the lowest-addressed byte, and the other bytes are written in increasing order of significance. In a big-endian system, the byte order is reversed, with the most significant byte being written into the byte of memory with the lowest address. The other bytes are written in decreasing order of significance. Figure 3-6 shows an example of how a little-endian system and a big-endian system would write a 32-bit (4-byte) data word to address 0x1000.

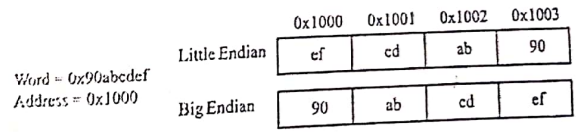

Fig. 3-6. Little endian versus big endian.

In general, programmers do not need to know the endianness of the system they are working on, except when the same memory location is accessed using loads and stores of different lengths. For example, if a 1-byte store of 0 into location 0x1000 was performed on the systems shown in Fig. 3-6, a subsequent 32-bit load from

0x1000 would return 0x90abcd00 on the little-endian system and 0x00abcdef on the big-endian system. Endianness is often an issue when transmitting data between different computer systems. However, as big-endian and little-endian computer systems will interpret the same sequence of bytes as different words of data. To get around this problem, the data must be processed to convert it to the endianness of the computer that will read it.

Memory system design has a tremendous impact on computer system performance and is often the limiting factor on how quickly an application executes. Both bandwidth (how much data can be loaded or stored in a given amount of time) and latency (how long a particular memory operation takes to complete) are critical to application performance. Other important issues in memory system design include protection (preventing different programs from accessing each other's data) and how the memory system interacts with the I/O system.

3.5.3 THE I/O SUBSYSTEM

The I/O subsystem contains the devices that the computer uses to communicate with the outside world and to store data, including hard disks, video displays, printers, and tape drives. I/O systems are covered in detail in Chapter 11. As shown in Fig. 3-1, I/O devices communicate with the processor through an I/O bus, which is separate from the memory bus that the processor uses to communicate with the memory system.

Using an I/O bus allows a computer to interface with a wide range of I/O devices without having to implement a specific interface for each I/O device. An I/O bus can also support a variable number of devices, allowing users to add devices to a computer after it has been purchased. Devices can be designed to interface with the bus, allowing them to be compatible with any computer that uses the same type of I/O bus. For example, almost all PCs and many workstations use the PCI bus standard for their I/O bus. All of these systems can interface to devices designed to meet the PCI standard. All that is required is a *device driver* for each operating system—a program that allows the operating system to control the I/O device. The downsides of using an I/O bus to interface to I/O devices is that all of the I/O devices on a computer must share the I/O bus and I/O buses are slower than dedicated connections between the processor and an I/O device because I/O buses are designed for maximum compatibility and flexibility.

I/O systems have been one of the least-studied aspects of computer architecture, even though their performance is critical to many applications. In recent years, the performance of I/O systems has become even more critical with the rising importance of database and transaction-processing systems, both of which depend heavily on the I/O subsystems of the computers they run on. This has made I/O systems an area of active research, particularly given the high prices that many companies are willing to pay for improvements in the performance of database and transaction-processing systems.

3.6 Summary

The purpose of this chapter has been to lay the groundwork for upcoming chapters by presenting an introduction to the major hardware building blocks of computer systems and the software components that interact with them. We discussed how computer systems are broken down into processors, memory systems, and I/O, and provided an introduction to each of these topics. This chapter also covered the different levels at which programs are implemented, ranging from the machine languages that processors execute to the high-level languages that humans typically use to program computers.

The next several chapters will concentrate on processor architecture, from programming models to techniques for improving performance like pipelining and instruction-level parallelism. After that, we will examine the memory system, discussing virtual memory, memory hierarchies, and cache memories. Finally, we will conclude with a discussion of I/O systems and an introduction to multiprocessing.

Solved Problems

Stored-Program Computers

3.1. The program for emacs (a UNIX text editor) is 2,878,448 bytes long on the computer being used to write this book. If a human could enter 1 byte of the program per second into the switches used to program a non-stored-program computer (which seems optimistic), how long would it take to start up the emacs program? If the human had an error rate of 0.001 percent in entering data, how many errors would be made in entering the program?

Solution

At 1 byte/s, the program would take 2,878,448 s, which is approximately 47,974 min, or 800 h, or 33.3 days. Obviously, a text editor would not be much use if it took over a month to start up.

An error rate of 0.001 percent is one error per 100,000 bytes, which would be extremely good for a human. Even so, the human would be expected to make approximately 29 errors in entering the program, each of which would have to be debugged and corrected before the program would run correctly.

Machine Language versus Assembly Language

3.2. a. What is the difference between machine language and assembly language?
b. Why is assembly language considered easier for humans to program in than machine language?

Solution

a. Machine-language instructions are the patterns of bits used to represent operations within the computer. Assembly language is a more human-readable version of machine language, in which each instruction is represented by a text string that describes what the instruction does.

b. In assembly-language programming, the assembler is responsible for converting assembly-language instructions into machine language, not the human. Humans generally find it easier to understand the text strings that represent assembly-language instructions than the numbers that encode machine language instructions. Also, relying on the assembler to translate assembly-language instructions into machine-language instructions eliminates the possibility of errors in generating the machine-language representation of each instruction.

Self-Modifying Programs

3.3. Why are self-modifying programs less common today than they were on early computers?

Solution

There are two main reasons. First, self-modifying code is harder to debug than non-self-modifying code, because the program that is executed is different from the one that was written. As computers have gotten faster, the performance advantages of self-modifying code have become less significant than the increased debugging difficulty.

Second, improvements in memory system design have reduced the performance improvement that can be gained through self-modifying code.

Compilers versus Assemblers

3.4. Explain briefly why the quality of a compiler has more impact on the execution time of a program developed using the compiler than the quality of an assembler has on programs developed using the assembler.

Solution

In general, there is a one-to-one mapping between assembly-language and machine-language instructions. An assembler's job is to translate each assembly-language instruction into its machine-language representation. Assuming the assembler does this translation correctly, the instructions in the resulting machine-language program are exactly the same as those in the source assembly-language program, just in a different encoding. Because the assembler does not change the set of instructions in a program, it has no impact on the execution time of the program.

In contrast, a compiler's job is to determine a sequence of assembly-language instructions that perform the computation specified by a high-level language program. Since the compiler is creating the sequence of assembly-language instructions for the program itself, the quality of the compiler has a great deal of impact on how long the resulting program takes to execute. Bad compilers create programs that do a great deal of unnecessary work and therefore run slowly, while good compilers eliminate this unnecessary work to deliver better performance.

ltiprogramming (I)

. How does a multiprogrammed system present the illusion that multiple programs are running on the machine simultaneously? What factors can cause this illusion to break down?

Solution

Multiprogrammed systems rotate through the programs running on them very frequently (60 or more times per second. As long as the number of programs running on the system is relatively small, each program will get a chance to execute often enough that the system looks like it is executing all of the programs at the same time, in that they all appear to be making progress simultaneously.

If the number of programs running on the system gets too large—for example, approaching the number of context switches performed per second—users will be able to notice the gaps in time when a given program isn't making progress, and the illusion will break down. Even with a small number of programs running on the machine, it is often possible to tell that the machine is sharing its processor among the programs, because each program's rate of progress will be slower than if it had the machine to itself.

ultiprogramming (II)

. If an 800-MHz computer context-switches 60 times per second, how many cycles are there in each time slice?

Solution

800 MHz = 800,000,000 cycles/s. 800,000,000/60 = 13,333,333 cycles/timeslice.

ultiprogramming (III)

7. Suppose a given computer context-switches 60 times per second. If the human interacting with the computer notices any time a given operation takes more than 0.5 s to respond to an input, how many programs can be running on the computer and have the system guarantee that the human never notices a delay? (Assume that the system switches between programs in a round-robin fashion, that a program can always respond to an input during the first timeslice that it executes in after the input occurs, and that programs always execute for a full timeslice when they are selected to execute.) How many programs can be running before the human notices a delay at least half the time?

Solution

If the computer context-switches 60 times per second, there are 30 context switches in 0.5 s. Therefore, the computer can execute up to 30 programs and guarantee that each program gets a timeslice within 0.5 s of any user input. Since each program is guaranteed to respond to user inputs during its first timeslice after the input occurs, this will guarantee that the human never notices a delay.

For the human to only see a delay half of the time, there has to be a 50 percent chance that the program that an input is directed to gets a timeslice within 0.5 s after the input occurs. Since programs execute in round-robin fashion, this means that half of the programs have to

execute within 0.5 s after the input. Since 30 programs execute in 0.5 s, there can be up to 60 programs running on the computer before the human notices a delay. More than half the time

Operating Systems (I)

3.8. Give two examples of problems that could occur if a computer allowed user programs to access I/O devices directly, rather than requiring them to go through the operating system.

Solution

The two examples described in this chapter are

1. Protection violations—If user programs can access data storage devices directly, then they can read or write data that belongs to other programs that they should not have access to.
2. Sequential access violation—A program might not be done with a device when its timeslice ends. If another program tries to use the device before the program gets another timeslice, the I/O operations of the two programs could become interleaved. This could result in unusual errors, such as the output of two programs being interleaved on the computer's monitor.

Another problem that wasn't discussed has to do with the fact that user programs are untrusted—they may have errors or may be actively malicious. If user programs can directly access I/O devices, they can send illegal operations or otherwise interfere with the ability of other programs to use the device. Having the operating system, which is a trusted program, control the devices eliminates this problem because the operating system can check to make sure any requested operation is legal before performing it.

Operating Systems (II)

3.9. Why is it necessary for a computer system to provide a privileged mode and a user mode for programs?

Solution

Privileged mode is the mechanism that computers use to prevent user programs from performing tasks that are limited to the operating system. Without a privileged execution mode, the hardware would be unable to tell whether the program attempting an operation were a user program or the operating system, preventing it from knowing whether the program should be allowed to perform the operation. Some systems provide more than two execution levels to allow different programs to be allowed access to different resources, but two levels are sufficient for most operating systems.

Register Files

3.10. Why does increasing the amount of data that can be stored in a processor's register file generally increase the performance of the processor?

Solution

Data in the register files can be accessed more quickly than data in the memory system. Therefore, being able to keep more data in the register file allows more data to be accessed at this faster speed, improving performance.

Memory Systems (I)

3.11. What is the difference between RAM and ROM?

Solution

Read-only memory (ROM) holds data that the processor can only read, not modify. It is used for things like the boot program that the system runs when the power is turned on. Random-access memory (RAM) can be both read and written. It is used to hold programs and data during operation of the computer.

Memory Systems (II)

3.12. Suppose that a computer's memory system did not have the random-access property—that is, that memory references took different amounts of time to complete depending on which address they referenced. How would this complicate the process of program development?

Solution

Non-random-access memory would require that programmers keep track of where a program's data was stored in memory in order to maximize performance. Programmers would want to store frequently accessed variables in locations that had short access times to reduce the total amount of time spent accessing memory. With random-access memory, the location of a variable in memory is irrelevant to its access time, so it doesn't matter where data is placed.

Big-Endian versus Little-Endian Addressing

3.13. The 32-bit value 0x30a79847 is stored to the location 0x1000. What is the value of the byte in address 0x1002 if the system is big endian? Little endian?

Solution

In a big-endian system, the most-significant byte in the word is stored in the address specified by the store operation, with successively lower-order bytes being stored in later locations. Therefore, the byte 0x98 will be stored in location 0x1002.

In a little-endian system, the lowest-order byte is stored in the location of the operation, and higher-order bytes are stored in successive locations. In this system, the value in location 0x1002 will be 0xa7 after the operation completes.

I/O Systems

3.14. What are the trade-offs involved in using a single I/O bus for all of the devices connected to a given system?

Solution

The trade-off is mainly one of bandwidth versus versatility. Using a single bus means that all of the devices attached to the processor have to share the bandwidth of the bus, limiting performance. However, using a single bus allows many devices to interface to a single system without requiring that the system designers provide separate interfaces for each possible device. Devices can be designed to match the interface of the bus, allowing the computer to interface to a wide range of devices, even ones that did not exist when the computer was being designed.