CHAPTER 1

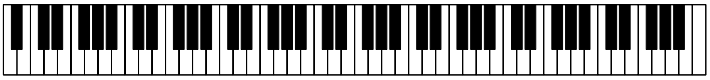
Architecture and Implementation

Computer science often distinguishes between abstraction and implementation—i.e., between the general and the particular. We may examine any computer system at two major levels: its *architecture* and its *organization*. Although numerous books convey both of these levels in their titles and contents, we are going to concentrate on architecture in this book. We first direct our readers toward an understanding of the distinction between these levels.

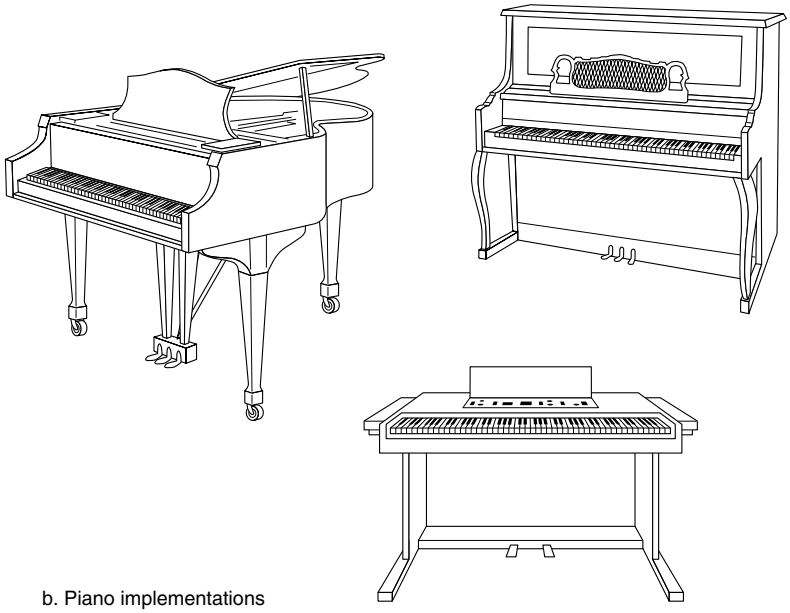
In the first decades of the history of computers, the sporadic emergence of new ideas and new companies resulted in a jumbled succession of disparate approaches to computer design. The design of the IBM® System/360™ series by Amdahl and his team, however, marked not only the trend-setting idea of a *family line* of computers but also a clear articulation of architecture as distinct from implementation:

- The *architecture* of a computer system is the abstraction equivalent to the user-visible interface: the structure and the operation of the system as viewed by the assembly language programmer and the compiler-writer. If an architecture is well-designed, well-engineered to adapt to future technologies, and well-liked by the market, it may persist for a decade or longer.
- An *implementation* is the realization and construction of that interface and structure out of specific hardware (and possibly software) components. Because of technological advances, any particular implementation (i.e., one model) may only be actively marketed for a relatively short period of time.

Several different implementations of an architecture may appear over a period of years. Each can offer different trade-offs among cost, performance, and convenience, but all will present the same interface to the assembly language programmer. Such consistency over time, despite technological change, has clearly helped computer system manufacturers to develop brand loyalty and facilitate the development of software as an allied industry.



a. Piano keyboard architecture



b. Piano implementations

Figure 1–1 Architecture and implementation of the piano

1.1 Analogy: Piano Architecture

Architecture applies to buildings, landscapes, computers, and even pianos. Let us briefly consider the architecture of pianos. Piano architecture can be defined by the specification of the keyboard, as shown in Figure 1–1a. The keyboard is the player (user) interface to this musical instrument. It consists of 88 keys: 36 black keys and 52 white keys. Striking a key causes a note of specified frequency to sound. As the size and the arrangement of the keys are identical for all modern piano keyboards, anyone who can play the piano can play *any* piano.

Many implementations of piano architecture are possible, as shown in Figure 1–1b. The implementation is concerned with the details of a particular maker's materials. The kinds of wood and metal used, the selection of ivory or plastic keys, and the size and shape of the instrument are all implementation decisions made by the piano builder. Regardless of the implementation decisions made, however, any piano player can play the final product.

Table 1–1 Generations of Computer Languages

Generation	Description	Attributes and Examples
1GL	Machine language	Each instruction speaks directly to the hardware level of a particular architecture. Instructions are numeric (i.e., binary patterns of 0s and 1s), but those can be made partially comprehensible by clustering adjacent bits together using an appropriate choice of base: • decimal (base 10), as in the IBM 1620 • octal (base 8), as in the PDP®-11 • hexadecimal (base 16), as in the Alpha™, Itanium®, and most other current architectures.
2GL	Assembly language	Each instruction is a mnemonic—e.g., `ADD`—but stands in near one-to-one correspondence with machine instructions. Additional directives to the *assembler* program help with storage allocation and program segmentation.
3GL	High-level languages	A *compiler* program translates statements in an arbitrarily defined artificial programming language into the appropriate sequences of machine-level primitives. Examples include COBOL, FORTRAN, PL/I, BASIC, C, Pascal, and Ada.
4GL	Newer languages	Newer types of computer languages include: • artificial intelligence languages (e.g., LISP) • data access languages (e.g., SQL) • natural-language query tools • object-oriented languages (e.g., C++, Smalltalk, or Java®).

In a computer system, the architecture consists of the programming interface: the instruction set, the structure and addressing of memory, the control of input and output (I/O), and so on. Several implementations of an architecture can be possible using different electronic design techniques that may have different size, cost, and performance characteristics. A program that runs on one machine should run on all machines conforming to the same architecture. Indeed, computer architects including Rau and Fisher of Hewlett-Packard have expressed that a contract exists between programs written for the architecture and the processor implementations of that architecture.

1.2 Types of Computer Languages

The earliest computers had to be programmed by people who knew the detailed capabilities and limitations of the hardware. Memory was an especially precious resource, and great care and ingenuity were required to squeeze algorithms into the extremely limited space available. Subsequently, as the overall capabilities of computers improved and use became widespread, successive generations of computer languages were devised in order to improve programmer productivity and accuracy at the expense of performance and resources. Table 1–1 shows the progression of programming languages, where *n* GL means *n*th generation language.

Assembly language lies between machine language and higher-level languages, such as C or Pascal. Assembly language more precisely expresses the constraints of an architecture than do high-level programming languages, but the latter are more amenable to reuse of code segments and global optimization. Accordingly, learning at least one assembly language can lead to an appreciation of what high-level languages actually do behind the scenes.

1.3 Why Study Assembly Language?

The convenience and greater portability of high-level languages raise the very real question of why anyone should study assembly language. Is it some arcane rite of initiation for the truly computer-savvy? Or is the primary motivation to see how a computer really works?

In a purely intellectual sense, an in-depth appreciation of at least one computer architecture is incredibly helpful in trying to comprehend its most basic strengths and weaknesses. Moreover, in a pragmatic sense, assembly language may make it possible to accomplish the following:

- the fastest attainable execution speed;
- the least memory usage;
- very specialized data manipulation, thereby compensating for features lacking in a particular high-level programming language; and
- specialized device control, such as a device driver not regularly furnished with the operating system.

Hardware improvements over time certainly bring ever-faster program execution speeds. While the ever-greater densities of memory technology would seem to reduce most concerns about program size, occasions do still arise where the tightest, fastest code has significant value. Good software engineering teams know when to utilize assembly language, even when they primarily use 3GL and 4GL tools. Many compilers for the C programming language explicitly provide a means to embed brief sequences of assembly language into a program using the `asm` keyword or `_Asm` intrinsic functions.

To be sure, assembly language is relatively difficult to code, debug, and maintain. Most severely, assembly language lacks portability from one architecture to another and thus, in long-term value, suffers significantly in comparison to high-level languages. Hyde has discussed these and other objections to assembly language, but has offered further reasons why learning assembly language deserves the attention of aspiring and practicing computer scientists. In particular, good assembly language programmers make better high-level language programmers "because they understand the limitations of the compiler and they know what it's doing with their code." We will return to this concept in later chapters.

1.4 Prefixes for Binary Multiples

For certain topics, especially architecture, computer science is a quantitative science. We shall introduce appropriate discussions of number systems and related matters as they are needed. The first such discussion here treats the matter of scale factors and abbreviations for "big" numbers.

Treating 2^{10} (1024) and 10^3 (1000) as synonymous introduced a sloppy situation for nomenclature in the fields of computer science, data processing, and data transmission. People were using the prefix *kilo*, from the International System of Units (SI), not only for its proper meaning of precisely 1000, but also to denote the binary multiple 1024. Doing so introduces an uncertainty of 2.4% in magnitude. The uncertainty compounds to more than 4.8% for the SI prefix mega, and the discrepancy just gets worse and worse for larger quantities. Moreover, kilo is typically abbreviated as a capital K in computer-related contexts, even though the proper SI abbreviation is a small k for kilo (which is an exception to the capitalization of most other abbreviated SI prefixes that denote multiples).

In 1998, the International Electrotechnical Commission (IEC) approved a new standard of names and symbols for the prefixes for binary multiples. The new names have mnemonic analogies to corresponding decimal multiples, as shown in Table 1–2.

Table 1–2 Prefixes for Binary Multiples

Factor	Name	Symbol	Origin	Analogy
2^{10}	kibi	Ki	kilobinary: $(2^{10})^1$	kilo: $(10^3)^1$
2^{20}	mebi	Mi	megabinary: $(2^{10})^2$	mega: $(10^3)^2$
2^{30}	gibi	Gi	gigabinary: $(2^{10})^3$	giga: $(10^3)^3$
2^{40}	tebi	Ti	terabinary: $(2^{10})^4$	tera: $(10^3)^4$
2^{50}	pebi	Pi	petabinary: $(2^{10})^5$	peta: $(10^3)^5$
2^{60}	exbi	Ei	exabinary: $(2^{10})^6$	exa: $(10^3)^6$

The IEC-approved prefixes are not part of the International System of Units (SI), but they have been supported by the International Committee for Weights and Measures (CIPM) and the Institute of Electrical and Electronics Engineers (IEEE®).

We use these new prefixes in this book. For instance, in the next three tables of this chapter, you will see KiB and MiB denoting kibibytes and mebibytes, where you might have expected KB and MB denoting kilobytes and megabytes, respectively.

1.5 Instruction Set Architectures

An *instruction set architecture* (ISA) abstracts the interface between a computer's hardware and the lowest-level software for the programmer or compiler-writer. Thorough knowledge of an ISA involves not only the appropriate ways to operate the programmer-accessible registers used for calculations and storage of intermediate results, but also how to move data between those

registers and memory, storage, or other attached devices. With knowledge of a particular ISA, one knows in principle what the computer can do and—as we shall sketch out in this book—what simple studies to conduct when seeking improved performance with particular implementations.

A well-conceived ISA leaves considerable latitude for implementations that realize the architectural concept in numerous actual products that serve different purposes with different cost implications. Ideally, the ISA will not be rendered prematurely obsolete by any unforeseen technological developments, because the effort and cost to design a new ISA and gain its adoption in the marketplace are very high.

Certain interrelated facets of the design of an ISA usually cannot be changed through new implementations; instead, they require extension or replacement of the architecture. Those intrinsic components of design include the following: the bit width(s) of data to be easily manipulated; the types of data to be represented and manipulated; the number and nature of registers; the amount of memory to be readily accessible, and in how many ways it can be accessed; the way that external devices are to be accessed; and the numbers and classes of machine instructions.

In the next section, we show some of the very different choices the computer industry has made for those and other architectural facets. We give more details in Chapter 2 and throughout the book.

1.6 The Life Cycle of Computer Architectures

Successive implementations of a computer architecture are routinely brought forth by a hardware innovator at fairly frequent intervals. A completely new architecture, or a whole new class of architectures, appears much less frequently.

Contemporary architectures fall into three classes. *Complex Instruction Set Computers* (CISC) typically include large numbers of machine instructions of many different styles. That complexity poses difficulties of implementation, because each style of instruction may require substantial real estate on the computer chip. *Reduced Instruction Set Computers* (RISC) are defined by smaller numbers of machine instructions of very few styles. The savings in space on a computer chip can, in favorable situations, make possible intrinsically faster circuitry. RISC programs can thus potentially execute faster than CISC programs, even though they usually contain more machine instructions. The third and newest class of contemporary architecture, *Explicitly Parallel Instruction Computers* (EPIC), includes the Itanium architecture that this book uses as its central example.

Computer architectures may also be classified according to the width of the *datapath,* the internal components through which information flows—e.g., 64 bits for Itanium architecture.

When new architectures emerge, they may appear to be *evolutionary* because they evince strong family resemblances to earlier architectures from the same vendor. On the other hand, they may appear *revolutionary* because they offer a clean break with the past. We now illustrate these concepts through the history of three families of computer architectures.

1.6.1 The 32-Bit Intel® Architecture and Its Predecessors

In 1971, Intel® Corporation integrated all of the traditional functionality of a central processing unit (CPU) into a single microcomputer chip, given the marketing number 4004. This pioneering chip could handle data 4 bits at a time and could access 640 bytes of memory. A year later, the 8008 microprocessor chip appeared with the ability to handle data 8 bits at a time and access 16,384 bytes of memory.

Table 1–3 looks at the characteristics of successive members of the family of processors starting with the Intel 8080, which was the basis for some of the earliest inexpensive general-purpose personal computers. As memory technologies improved, the push toward convenient addressing of larger amounts of memory drove Intel and other manufacturers to redesign their products with successively greater widths for the internal registers and pathways where addresses (pointers) as well as data are manipulated. By 1990 this trend settled on 32 bits as the prevailing standard register width for even the smallest computers, which had come to be called *microcomputers* because of their physical size, and not as a measure of their computing power.

An important feature of these Intel processors is that each generation can execute most programs prepared for the previous generation because of similarities of integer registers. The right half of a 16-bit register can be used to manipulate 8-bit integers, the right half of a 32-bit register can be used to manipulate 16-bit integers, and the right quarter of a 32-bit register can be used to manipulate 8-bit integers. Assembly language programs written for the 8080 could automatically be translated into a format suitable for the 8086 processor, and programs written for the 16-bit processors can run directly on the 32-bit processors. We shall discuss some of the other information in Table 1–3 in later sections of this book.

1.6.2 The Alpha™ Architecture and Its Predecessors

The design of the IBM System/360 strongly influenced many subsequent computer architectures, including 16-bit minicomputers brought forth by numerous manufacturers during the 1970s. Quite possibly the most successful among those designs, the PDP®-11 by Digital® Equipment Corporation, not only persisted through about a dozen implementations over more than two decades, but also came to be seen as the progenitor of families of 32-bit VAX® and 64-bit Alpha™ computers.

Some of the attributes of the PDP-11, VAX, and Alpha product lines are summarized in Table 1–4. The VAX (Virtual Addressing eXtension) is frequently cited as the exemplar of a Complex Instruction Set Computer (CISC). The Alpha processor was the first 64-bit Reduced Instruction Set Computer (RISC) to attain wide commercial deployment. Several other manufacturers had already marketed successful 32-bit RISC designs, but Digital Equipment Corporation opted to make its move to 64 bits simultaneously with its move from CISC to RISC. We shall discuss some of the other information in Table 1–4 later in this book.

Table 1–3 Comparisons Among Computer Architectures by Intel Corporation

Datapath width	8 bits (1 byte)	16 bits (2 bytes)	32 bits (4 bytes)
Marketing names	8080, 8085	8086, 8088, 80286	Intel386™, Intel486™, Pentium®
Physical formats	single chip	single chip	single chip
Complexity classification	classic micro	CISC	CISC
Number of integer registers	8	14	16
Interchangeability of registers	almost none	some	moderate
Instruction size	1, 2, 3 bytes	1–4 bytes	1–17 bytes
Number of instruction styles	7	byte stream	byte stream
Number of opcodes	74	133	154
Number of operands	0, 1, 2	0, 1, 2, 3	0, 1, 2, 3
Allowed memory access	few instructions	many instructions	many instructions
Number of addressing modes	6	8	8+
Number of integer data types	2	3	4
Number of floating data types	0	3*	3*
Byte ordering	little-endian	little-endian	little-endian
Unidirectional branch range	full-range 64 KiB COND and JUMP	127 bytes	32 KiB
Logical address space	64 KiB	1 MiB†	4 GiB†
Input/output strategy	IN, OUT	memory-mapped; IN, OUT	memory-mapped; IN, OUT
Date of introduction	1974	1978	1985

* Floating-point operations required a second chip for all 16-bit processors and some 32-bit processors.

† Initially; subsequently extended.

Table 1–4 Comparisons Among Computer Architectures by Digital Equipment Corporation

Datapath width	16 bits (2 bytes)	32 bits (4 bytes)	64 bits (8 bytes)
Marketing names	PDP-11, LSI-11	VAX, MicroVAX	Alpha
Physical formats	circuit board(s), single chip	circuit board(s), single chip	single chip
Complexity classification	classic mini	CISC	RISC
Number of integer registers	8	16	32
Interchangeability of registers	extensive	extensive	extensive
Instruction size	2, 4, 6 bytes	1 – 37 bytes	4 bytes
Number of instruction styles	6	byte stream	7
Number of opcodes	> 100	> 256	> 100
Number of operands	0, 1, 2	0–6	0–3
Allowed memory access	many instructions	many instructions	only load/store
Number of addressing modes	8	12	2
Number of integer data types	2	5	2
Number of floating data types	2	4	5
Byte ordering	little-endian	little-endian	little-endian*
Unidirectional branch range	255 bytes	127 bytes	4 MiB
Logical address space	64 KiB	4 GiB	16 EiB
Input/output strategy	memory-mapped	memory-mapped	memory-mapped
Lifetime as a marketed product	1971–1995	1978–2000	1992–

*Initially; subsequently extended.

1.6.3 The Itanium® Architecture and Its Predecessors

Both the VAX architecture and Intel's 32-bit architecture are cited as exemplars of Complex Instruction Set Computers (CISC). At the implementation level, the circuitry required on a chip to recognize and carry out many different styles of instructions can be elaborate and difficult to optimize for fast execution times. By the 1980s, research in computer science had led to serious proposals for a different approach to high performance: Select fewer instructions and design ways to execute them exceedingly quickly, rather than introduce more instructions of increasing specialization.

Although Intel engineers continually found ways to bring out further implementations of a successful 32-bit architecture without compromising performance, several other computer makers redirected their attention to the design of Reduced Instruction Set Computers (RISC). Hewlett-Packard® Company developed a 32-bit architecture that was at first called Precision Architecture, and later PA-RISC®. The salient features of RISC processors include fixed-size instructions, fewer instructions overall, larger numbers of integer registers to be used in similar ways, and restrictions on the number of instruction types that can directly manipulate data in the computer's memory. The 64-bit Alpha architecture (Table 1–4) and the 32- and 64-bit PA-RISC architectures (Table 1–5) follow these RISC principles.

Even as CISC and RISC designs coexisted amongst marketed computer systems in the 1990s, Intel collaborated with Hewlett-Packard in developing another fundamental class of computer architecture, Explicitly Parallel Instruction Computers (EPIC). In essence, an EPIC computer can simultaneously pursue more than one course of action, thus gaining a throughput advantage. An EPIC design can avoid certain time penalties that plague other architectures when the flow of instructions in a program has to change abruptly, as must occur for loop control. The earliest commercial product using EPIC design principles is the 64-bit Itanium processor, whose characteristics are compared with PA-RISC in Table 1–5.

1.6.4 The Naming of Architectures and Implementations

Sometimes the same or similar names have been given both to an architecture and to its implementations. In the case of VAX architecture, "VAX" was incorporated into the name of every implementation; moreover, the ISA remained very stable during the long prevalence of this architecture in the marketplace. There is no such continuing thread connecting the names of implementations within the various generations of products from Intel or several other corporations.

Intel Corporation has at various times designated its 16-, 32-, and 64-bit instruction sets as IA-16, IA-32, and IA-64. Any popular ISA, such as IA-32, accrues additional instructions over time. This leads to an ambiguity: whether the original designation defines the basic instruction set or the currently augmented set. Neither "IA-16" nor "IA-32" has appeared prominently in commercial product descriptions.

Intel used the phrase "IA-64 architecture" while the new 64-bit architecture was under development, but changed to "Itanium architecture" when marketing the first implementation, which was called "the Itanium processor." As the Itanium 2 processor (code name McKinley) reached production, Intel established the primacy of "Itanium Architecture" over "IA-64" or the lesser-known "Itanium Processor Family" (IPF). Thus, just as with VAX architecture and VAX processor implementations, we must accept some blurring of the distinction between the naming of an architecture and any implementation.

Table 1–5 Comparisons Among Computer Architectures by Hewlett-Packard and Intel

Datapath width	32 bits (4 bytes)	64 bits (8 bytes)	64 bits (8 bytes)
Marketing names	PA-RISC® 7xxx	PA-RISC® 8xxx	Itanium®
Physical formats	circuit board(s), single chip	single chip	single chip
Complexity classification	RISC	RISC	EPIC
Number of integer registers	32	32	128
Instruction size	4 bytes	4 bytes	$(3 \times 41) + 5$ bits
Number of instruction styles	~13	> 20	6
Number of principal opcodes	45	59	41
Number of operands	0–3	0–3	0–5
Allowed memory access	only load/store	only load/store	only load/store
Number of addressing modes	4	4	5
Number of integer data types	5	5	4
Number of floating data types	3	3	3
Byte ordering	big-endian*	big-endian, little-endian	little-endian, big-endian
Unidirectional branch range	256 KiB	256 KiB	16 MiB
Logical address space	4 GiB*	15 EiB	16 EiB
Input/output strategy	memory-mapped	memory-mapped	memory-mapped
Date of introduction	1986	1996	2001

* Initially; subsequently extended.

The code-name issue deserves further discussion: When a company has something new under development, the company (or the trade press) may use a code name for it. The trade press wrote for several years about “Merced” as something forthcoming from Intel Corporation. It was not always clear whether Merced meant an implementation or an architecture. Furthermore, some future author writing retrospectively about Itanium architecture and/or the original Itanium processor might miss a lot of historical source material without using the Merced, IA-64, IA64, and IPF as additional search keywords.

1.7 SQUARES: A First Programming Example

Beginnings are hard. Just think back to the very first program that you wrote in any computer language. It was probably oversimplified, and the hardest task may have been getting data in or out. In this book, each illustrative example in assembly language is kept as simple as possible in order to help you focus your reading and study on the matter at hand, though you must also attend to the details.

In this section, we present a simple but complete program that has served well as a teaching example. We are going to express the algorithm in three high-level languages and in assembly language for the Itanium architecture

Statement of the problem: Write a program that will produce in memory a table of the squares of the first three integers without using multiplication instructions.

Presentation of the algorithm: We begin by writing down the first several integers, N, their squares, N^2, and finally the first and second tabular differences in Figure 1–2.

N	*N²*	*1st tabular difference*	*2nd tabular difference*
1	1		
		3	
2	4		2
		5	
3	9		2
		7	
4	16		2
		9	
5	25		

Figure 1–2 Computation of squares by tabular differences

Successive values of the first tabular difference are computed by adding the constant second tabular difference each time. Then successive values of the squares can be computed by adding the appropriate value of the first difference to the already known previous square.

1.7.1 C, FORTRAN, and COBOL

The enumeration for computing successive squares is readily expressed in a standard 3GL programming language such as C:

```
#include <stdio.h>
main()
{
    long long sq1, sq2, sq3;
    long long temp, diff1, diff2;

    diff1 = 1;
    diff2 = 2;
    temp = 1;
    sq1 = temp;
```

```
        diff1 = diff2 + diff1;
        temp = diff1 + temp;
        sq2 = temp;
        diff1 = diff2 + diff1;
        temp = diff1 + temp;
        sq3 = temp;
        printf("%lld\t%lld\t%lld\n", sq1, sq2, sq3);
        return 0;
    }
```

or FORTRAN:

```
      PROGRAM SQUARES

      INTEGER*8 SQ1, SQ2, SQ3
      INTEGER*8 TEMP, DIFF1, DIFF2

      DIFF1 = 1
      DIFF2 = 2
      TEMP = 1
      SQ1 = TEMP
      DIFF1 = DIFF2 + DIFF1
      TEMP = DIFF1 + TEMP
      SQ2 = TEMP
      DIFF1 = DIFF2 + DIFF1
      TEMP = DIFF1 + TEMP
      SQ3 = TEMP
      PRINT *, SQ1, SQ2, SQ3
      END
```

or COBOL:

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. SQUARES.
      *
       DATA DIVISION.
       WORKING-STORAGE SECTION.
        01 OUTPUT-FIELD.
      * Note that COMP for 18 digits equates to a quad word.
      * CR1,2,3 are 0x0d = 13 = CR, for on-screen display
           05  SQ1     PIC  9(18)        VALUE 0.
           05  CR1     PIC  X(1)         VALUE X"0D".
           05  SQ2     PIC  9(18)        VALUE 0.
           05  CR2     PIC  X(1)         VALUE X"0D".
           05  SQ3     PIC  9(18)        VALUE 0.
           05  CR3     PIC  X(1)         VALUE X"0D".
        01 CALCULATION-FIELD.
           05  DIFF1   PIC  9(18)        VALUE 1.
           05  DIFF2   PIC  9(18)        VALUE 2.
           05  TEMP    PIC  9(18)        VALUE 1.
```

```
*
 PROCEDURE DIVISION.
 CALCULATE-SQUARES SECTION.
      MOVE TEMP TO SQ1.
      ADD DIFF2 TO DIFF1 GIVING DIFF1.
      ADD DIFF1 TO TEMP GIVING TEMP.
      MOVE TEMP TO SQ2.
      ADD DIFF2 TO DIFF1 GIVING DIFF1.
      ADD DIFF1 TO TEMP GIVING TEMP.
      MOVE TEMP TO SQ3.
 DISPLAY-RESULTS SECTION.
      DISPLAY OUTPUT-FIELD.
*
      EXIT PROGRAM.
 END PROGRAM SQUARES.
```

It should be evident that the pattern of first adjusting `diff1`, then using `diff1` to adjust `temp`, and finally storing `temp` as the next square could be iterated any desired number of times to compute `sq4`, etc.

1.7.2 Assembly Language for Itanium Architecture

Now let us transform the expression of this algorithm from a 3GL implementation into a 2GL equivalent in Itanium assembly language. This listing will appear quite new to you even if you are familiar with the IA-32 or PA-RISC architectures. Stark differences are very common when attempting to move from one assembly language to another.

The algorithm for the SQUARES program (Figure 1–2), as written in Itanium assembly language, is shown in Figure 1–3. We will not fully explain the language elements used here. For the present, it is enough to appreciate that the three columns at the left make up the actual program instructions. The phrases in mixed case to the right of two slash characters (`//`) are explanatory comments that annotate the programmer's intended relationship between those instructions and the algorithm.

We shall show how to run SQUARES in Chapter 3, after we have introduced the symbolic debugger. For now, you may focus your attention on the substance of the algorithm from `first` to `done`. In later chapters we shall also explain the purpose of the lines preceding `main` and following `done`.

The Itanium `add` instruction reads left to right like an algebraic expression in a high-level language, but with a comma instead of a plus sign. An Itanium processor has 128 integer registers, Gr_0 … Gr_{127}, that are addressed as `r0` … `r127` in assembly language.

```
// SQUARES      Table of Squares
        .data                   // Declare storage
        .align  8               // Desired alignment
sq1:    .skip   8               // To store 1 squared
sq2:    .skip   8               // To store 2 squared
sq3:    .skip   8               // To store 3 squared
                                // etc.
        .text                   // Section for code
        .align  32              // Desired alignment
        .global main            // These three lines
        .proc   main            //  mark the mandatory
main:                           //   'main' program entry
        .body                   // Now we really begin...
first:  mov     r21 = 1;;       // Gr21 = first difference
        mov     r22 = 2;;       // Gr22 = 2nd difference
        mov     r20 = 1;;       // Gr20 = first square
        addl    r14 = @gprel(sq1),gp;;  // Point to storage
        st8     [r14] = r20;;   //         for sq1
        add     r21 = r22,r21;; // Adjust first difference
        add     r20 = r21,r20;; // Gr20 = second square
        addl    r14 = @gprel(sq2),gp;;  // Point to storage
        st8     [r14] = r20;;   //         for sq2
        add     r21 = r22,r21;; // Adjust first difference
        add     r20 = r21,r20;; // Gr20 = third square
        addl    r14 = @gprel(sq3),gp;;  // Point to storage
        st8     [r14] = r20;;   //         for sq3
                                // etc.
done:   mov     r8 = 0;;        // Signal all is normal
        br.ret.sptk.many b0;;   // Back to command line
        .endp   main            // Mark end of procedure
```

Figure 1–3 SQUARES program for Itanium architecture

Unlike many older assembly languages, Itanium assembly language does not support direct symbolic addressing of a data location, such as `sq1` where we want to store the first computed square. It instead requires two steps. First, we calculate the address of `sq1` in register `r14` by adding an offset `@gprel(sq1)` computed by the assembler onto the address contained in register `gp`, the global pointer. This pointer gets a value when the system loads the program. We then store the computed 8-byte value in register `r20` into memory at the address given by register `r14`, using the assembler syntax `[r14]` with a store (`st8`) instruction. Similarly, we copy the value of each successive square computed in register `r20` into the appropriate memory location.

The double semicolons shown in Figure 1–3 mark *stops*, which inform an Itanium assembler that we have *not* analyzed potential timing interdependencies among the machine instructions. The assembler can produce from this format a valid program free from such complications, as we shall show later in this book. This simple SQUARES program does not illustrate the parallelism or predication features of the EPIC architecture.

1.8 Review of Number Systems

Throughout this book we use the decimal (base 10), binary (base 2), octal (base 8), and hexadecimal (base 16) number systems. A concise review of number systems and representations for integer data will conclude this introductory chapter. You should skip this material only if you are already adept with conversions among these representations, including expressions of negative integer values.

All data stored and manipulated in contemporary computers exist in binary form. Integers, floating-point numbers, characters, and instructions exist as sequences of zeros and ones. This binary representation is base 2.

When we display binary data, we usually use base 8 (octal), base 10 (decimal), or base 16 (hexadecimal), instead of base 2 (pure binary). These larger bases are easier for humans to comprehend than long strings of binary digits, but the *value* of a stored chunk of numeric information is the *same*, regardless of the base used to represent it.

Different bases are suitable for different applications. Bases 8 and 16 are particularly useful for emphasizing patterns of bits within a stored unit, while base 10 is useful for understanding the everyday value of a stored number, since it is the base used for counting in natural human languages.

1.8.1 Positional Coefficients and Weights

The most frequently encountered numbering system involves *positional coefficients* and *weights*. The digit value at each position in a number is its positional coefficient. The weight of each digit is a successively larger power of the base of the number system, from right to left. We can express a value Q in the number system with base r (also called the radix), as follows, allowing for a fractional portion {in braces}:

$$Q = x_n w_n + x_{n-1} w_{n-1} + \ldots + x_1 w_1 + x_0 w_0 + \{x_{-1} w_{-1} + \ldots + x_{-m} w_{-m}\}$$

where

$$w_i = r^{\,i}$$

that is, weight = $r^{\,i}$ and r = radix or base, and

$$0 \leq x_i \leq r - 1$$

that is, x_i = positional coefficient.

This formalism ensures that the largest value of a positional coefficient is always one less than the base value—i.e., 1 for base 2 (binary), 9 for base 10 (decimal), and so forth. When the base is greater than 10, the letters A, B, … can be used to convey the positional coefficients with values of 10 and beyond. Base 16 (hexadecimal) uses the letters A through F to represent numerical values 10 through 15, respectively.

We will illustrate these concepts by expressing the value 154 in the bases 2, 8, 10, and 16, as follows:

$$
\begin{aligned}
154 \quad &= 1 \times 10^2 + 5 \times 10^1 + 4 \times 10^0 && \text{(base 10)} \\
&= 1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 && \text{(base 2)} \\
&= 2 \times 8^2 + 3 \times 8^1 + 2 \times 8^0 && \text{(base 8)} \\
&= 9 \times 16^1 + \mathrm{A} \times 16^0 && \text{(base 16)}
\end{aligned}
$$

That is, the value 154 results from summing the non-zero terms:

$$
\begin{aligned}
154 \quad &= 100 + 50 + 4 && \text{(base 10)} \\
&= 128 + 16 + 8 + 2 && \text{(base 2)} \\
&= 128 + 24 + 2 && \text{(base 8)} \\
&= 144 + 10 && \text{(base 16)}
\end{aligned}
$$

In summary: $154_{10} = 10011010_2 = 232_8 = 9\mathrm{A}_{16}$.

Up to this point, we have used a subscript of 16 when denoting representations of base 16. Notations commonly used in computer science are 0X or 0x (e.g., compilers for the C language) or a suffix of H or h (e.g., some Intel documentation) when expressing a hexadecimal value: $154_{10} = 9\mathrm{A}_{16} = 0\mathrm{x}9\mathrm{A} = 9\mathrm{AH}$.

1.8.2 Binary and Hexadecimal Representations

Modern digital computers use the binary number system internally because the most practical physical components are intrinsically binary in nature. Since long strings of 0s and 1s are cumbersome for human beings, most computer professionals routinely use base 16, or occasionally base 8. The various system software components—assemblers, compilers, linkers, and so forth—readily convert such numbers to their binary equivalents. Table 1–6 shows the binary, octal, and hexadecimal equivalents for the decimal values 0 through 16.

Base 8 and 16 representations are not only convenient but are easily derived from binary representations. Converting from binary to base 8 or 16 simply requires separating the binary number into 3-bit (for octal) or 4-bit (for hexadecimal) groups, from right to left, and then replacing each binary group with the appropriate digit for the new base (from Table 1–6). Consider the following illustration of decimal value 1249:

$$
\begin{aligned}
&010011100001_2 && && \text{(base 2)} \\
&010\ 011\ 100\ 001_2 &&= \quad 2341_8 && \text{(base 8)} \\
&0100\ 1110\ 0001_2 &&= \quad 4\mathrm{E}1_{16} && \text{(base 16)}
\end{aligned}
$$

Although this process is relatively intuitive, you may already appreciate that some pocket calculators have the capability to convert numbers amongst these common bases.

Table 1–6 Conversion Table for the First Few Integers

Decimal	Binary	Octal	Hexadecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	1 0000	20	10

The system software components for the assembly language programmer may express their outputs—program listings, linker maps, debugging aids, and dumps of memory contents—in octal or hexadecimal by default; decimal conversion may also be offered. Sometimes manual interpretation is required, as when several data elements of different bit widths have been packed for storage as one composite binary number.

1.8.3 Signed Integers

In the history of computer development, three methods have been considered for binary representations of ranges that include negative as well as positive integers. These methods are: *sign and magnitude*, *one's complement*, and *two's complement*. Sign and magnitude requires greater complexity at the physical implementation level, while one's complement introduces the complication of having two bit patterns that both represent the number zero. Since testing for zero is

a common operation, the need to consider two cases would require greater complexity at the physical implementation level. Accordingly, contemporary computers use two's complement representation for signed integers.

All methods for representing signed integers within *N* bits have the net effect of allocating one bit to represent positive/negative and the remaining $N - 1$ bits to represent the number's magnitude. For two's complement, zero is considered a positive number. Two's complement representation offers the advantage of making successive additions or subtractions of one work smoothly right through zero. Table 1–7 shows how small signed numbers can be represented by three bits. All of the representations agree for positive values, but differ in the handling of zero and negative values.

Table 1–7 Representations for Small Integer Values

Value	Two's Complement	One's Complement	Sign and Magnitude
+3	011	011	011
+2	010	010	010
+1	001	001	001
0	000	000 and 111	000 and 100
–1	111	110	101
–2	110	101	110
–3	101	100	111
–4	100	too big to represent	too big to represent

Notice that the two's complement representation looks like a complete binary counting sequence (0 to 7) that has been cut in half (0 to 3, and 4 to 7) and restacked. Also appreciate that the range of integers that can be represented within *N* bits extends from the value -2^{N-1} through 0 to $+2^{N-1} - 1$.

No special action is required to form the two's complement representation for a positive integer, except to realize that the most significant bit *must* be an explicit zero. Forming the two's complement of a negative integer can be accomplished by subtracting the magnitude from zero, for example:

$$\begin{array}{rr} 0 & 000 \\ -\underline{(+3)} & -\underline{(011)} \\ -3 & 101 \end{array}$$

(subtraction, with due attention to "borrowings")

Another method is to perform a bit-by-bit complementation of the positive number (including its zero sign bit) and then to add 1 to that intermediate result:

+3	011	becomes	100	(intermediate result)
			+(001)	(addition, with due attention to "carries")
–3			101	

In either of these methods, any borrowing or carrying outside of the *N*-bit field is not to be written as part of the final result.

Finding the hexadecimal representation of a negative integer proceeds similarly by mentally subtracting each digit from "F" and then adding 1 to that intermediate result:

+15502	3C8E	becomes	C371	(intermediate result)
			+(0001)	(addition, with due consideration for "carries")
–15502			C372	

Fortunately, as with all such techniques, familiarity comes with practice.

You will find that an understanding of hexadecimal and binary number systems is helpful to your study of architecture and assembly language programming, as well as other computer concepts.

Summary

We had several goals in this introductory chapter. First we drew a clear philosophical distinction between computer architectures and their particular implementations. In this view, an architecture should last longer than any one implementation. The marketing names chosen by the computer industry sometimes do not maintain a clear distinction between architecture and implementation.

We discussed the pros and cons of assembly language as a means of programming a computer, explaining that it provides the programmer with the closest view of intrinsic architectural features. Learning assembly language is a good way to learn about computer architecture. We showed what a simple algorithm looks like expressed in assembly language for Itanium architecture, and contrasted that to high-level languages.

This chapter included two mathematical topics. First, the recently standardized prefixes for naming certain powers of two, versus certain powers of ten, for describing computer storage were introduced. We then discussed how computers typically represent signed and unsigned integers, in anticipation of a continuing need for that understanding.

REFERENCES

Alpha Architecture Committee, *Alpha Architecture Reference Manual*, 3rd ed. Woburn, Mass.: Butterworth-Heinemann (Digital Press), 1998.

Amdahl, G. M., G. A. Blaauw, and F. P. Brooks, Jr., "Architecture of the IBM System/360," *IBM Journal of Research and Development* **8** (2), 87–101 (1967).

Blaauw, Gerrit A. and Frederick P. Brooks, Jr., *Computer Architecture: Concepts and Evolution*. Reading, Mass.: Addison-Wesley, 1997.

Brunner, Richard A., *VAX Architecture Reference Manual,* 2nd ed. Bedford, Mass.: Digital Press, 1991.

Eckhouse, Richard H. and L. Robert Morris, *Minicomputer Systems: Organization, Programming, and Applications (PDP-11).* Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1979.

Evans, James S. and Richard H. Eckhouse, *Alpha RISC Architecture for Programmers.* Upper Saddle River, N.J.: Prentice Hall PTR, 1999.

Hyde, Randall, *The Art of Assembly Language Programming*, 1996 [cited 25 September 2001]. Available from http://webster.cs.ucr.edu/.

Intel Corporation, "Basic Architecture," *IA-32 Intel Architecture Software Developer's Manual*, Vol. 1, 2001.

Intel Corporation, *History of the Microprocessor* [cited 23 May 2002]. Available from http://www.intel.com/intel/intelis/museum/exhibit/hist_micro/.

Kane, Gerry, *PA-RISC 2.0 Instruction Set Architecture,* Upper Saddle River, N.J.: Prentice Hall PTR, 1996.

Kernighan, Brian W. and Dennis M. Ritchie, *The C Programming Language*, 2nd ed. Englewood Cliffs, N.J.: Prentice Hall PTR, 1988.

Levy, Henry M. and Richard H. Eckhouse, *Computer Programming and Architecture: The VAX,* 2nd ed. Bedford, Mass.: Digital Press, 1989.

PA-RISC 1.1 Architecture and Instruction Set Reference Manual, 3rd ed. Hewlett-Packard Company, 1994 [cited 2 December 2001]. Available from http://h21007.www2.hp.com/dspp/tech/tech_TechDocumentDetailPage_IDX/1,1701,958,00.html.

PDP-11 Architecture Handbook. Maynard, Mass.: Digital Equipment Corporation, 1982.

"Prefixes for Binary Multiples," *NIST Reference on Constants, Units, and Uncertainty,* Physics Laboratory, National Institute of Standards and Technology, 2000 [cited 25 September 2001]. Available at http://physics.nist.gov/cuu/Units/binary.html.

Rau, B. Ramakrishna, and Joseph A. Fisher, "Instruction-Level Parallel Processing: History, Overview, and Perspective," *HP Technical Report HPL-92-132* (October 1992) [cited 9 March 2002]. Available from http://www.hpl.hp.com/techreports/.

Schlansker, Michael S., and B. Ramakrishna Rau, "EPIC: An Architecture for Instruction-Level Parallel Processors," *HP Technical Report HPL-1999-111* (2000) [cited 27 January 2002]. Available from http://www.hpl.hp.com/techreports/.

Triebel, Walter, *Itanium Architecture for Software Developers,* Intel Press, 2000.

The Virtual Museum of Computing, 2002 [cited 26 September 2002]. Available from http://vmoc.museophile.com/.

We also list here several contemporary books that include a greater emphasis on computer hardware than we provide in our book:

Carpinelli, John D., *Computer Systems: Organization & Architecture.* Boston, Mass.: Addison Wesley Longman, Inc., 2001.

Clements, Alan, *The Principles of Computer Hardware,* 2nd ed. Oxford, UK: Oxford University Press, 2000.

Hamacher, Carl, Zvonko Vranesic, and Safwat Zaky, *Computer Organization,* 5th ed. Boston, Mass.: McGraw-Hill, 2002.

Hennessy, John L. and David A. Patterson, *Computer Organization and Design: The Hardware/Software Interface,* 2nd ed. San Francisco, Cal.: Morgan Kaufmann Publishers, Inc., 1998.

Murdocca, Miles J. and Vincent P. Heuring, *Principles of Computer Architecture.* Upper Saddle River, N.J.: Prentice-Hall, Inc., 2000.

Stallings, William, *Computer Organization and Architecture: Designing for Performance,* 6th ed. Upper Saddle River, N.J.: Prentice Hall, Inc., 2003.

Tanenbaum, Andrew S., *Structured Computer Organization,* 4th ed. Upper Saddle River, N.J.: Prentice Hall, Inc., 1999.

EXERCISES

1. Describe the architecture of a bicycle. What parts of the bicycle are not parts of the architecture? What are some implementation differences among different bicycles?
2. Consider the car rental industry. What aspects of automobile architecture are essential to the ability of any driver to operate a randomly allocated rental car? What are some implementation differences among different automobiles that are not especially relevant to either the driver or the rental agency?
3. Explain why harpsichords, organs, and accordions are *not* exemplars of "piano architecture."
4. Why do you think that one computer manufacturer builds computers with a different architecture from those of another manufacturer? What does this imply about the importance of well-standardized high-level languages?
5. What effect do you think the standardization of operating systems or command languages might have on the future development of computer architectures?
6. Compute how much larger 1 EiB is than 1 EB, according to the IEC conventions for binary multiples.

7. PA-RISC Architecture 1.1 defines extension registers that permit virtual addresses to be 16, 24, or 32 bits wider than the base level 32 bits. Express the three resulting maximum virtual address sizes using appropriate binary prefixes from Table 1–2.

8. Adapt the program SQUARES in a high-level language to compute the cubes of the first five integers without using any explicit multiplication. Hint: An algorithm for N^3 can be discovered by writing down the series 1, 8, 27, 64, … and then inspecting the pattern of first, second, and third tabular differences. This is only a pencil-and-paper exercise, but save your work for future adaptation.

9. Digital's first highly successful minicomputer, the PDP-8, was a 12-bit machine. What range of integers can be represented in a 12-bit unsigned binary number? In a 12-bit two's complement signed number?

10. Convert 101010101_2 into hexadecimal and octal. Convert $10A34_{16}$ into octal and decimal. Negate both values using 32-bit two's-complement hexadecimal form.

11. Complete the following table by converting the given number in each row into the other bases.

	Decimal	Binary	Octal	Hexadecimal
a.	100			
b.		100		
c.			100	
d.				100

12. Perform the following hexadecimal arithmetic:

a. 205 – 6
b. AF9 + 9
c. 1A × B2 (use "long" multiplication, and think carefully about the "carries")
d. 1CFF + F2FF

13. Write a high-level language program that inputs a decimal number and a radix and reformats that decimal number in the specified radix. For example, if the inputs are 10 and 16, the output should be A. Limit yourself to bases less than or equal to 16.