Appendix C

Data Representation

In order to understand how a computer is able to manipulate data and perform computations, you must first understand how data is represented by a computer.

At the lowest level, the indivisible unit of data in a computer is a *bit*. A bit represents a single binary value, which may be either 1 or 0. In different contexts, a bit value of 1 and 0 may also be referred to as "true" and "false", "yes" and "no", "high" and "low", "set" and "not set", or "on" and "off".

The decision to use binary values, rather than something larger (such as decimal values) was not purely arbitrary– it is due in a large part to the relative simplicity of building electronic devices that can manipulate binary values.

C.1 Representing Integers

C.1.1 Unsigned Binary Numbers

While the idea of a number system with only two values may seem odd, it is actually very similar to the decimal system we are all familiar with, except that each digit is a bit containing a 0 or 1 rather than a number from 0 to 9. (The word "bit" itself is a contraction of the words "binary digit") For example, figure C.1 shows several binary numbers, and the equivalent decimal numbers.

In general, the binary representation of 2^k has a 1 in binary digit k (counting from the *right*, starting at 0) and a 0 in every other digit. (For notational convenience, the ith bit of a binary number A will be denoted as A_i.)

The binary representation of a number that is not a power of 2 has the bits set corre-

Figure C.1: Binary and Decimal Numbers

Binary		**Decimal**
0	=	0
1	=	1
10	=	2
11	=	3
100	=	4
101	=	5
110	=	6
$\vdots$	$\vdots$	$\vdots$
11111111	=	255

sponding to the powers of two that sum to the number: for example, the decimal number 6 can be expressed in terms of powers of 2 as $1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$, so it is written in binary as 110.

An eight-digit binary number is commonly called a *byte*. In this text, binary numbers will usually be written as bytes (i.e. as strings of eight binary digits). For example, the binary number 101 would usually be written as 00000101– a 101 padded on the left with five zeros, for a total of eight digits.

Whenever there is any possibility of ambiguity between decimal and binary notation, the *base* of the number system (which is 2 for binary, and 10 for decimal) is appended to the number as a subscript. Therefore, 101_2 will always be interpreted as the binary representation for five, and never the decimal representation of one hundred and one (which would be written as 101_{10}).

Conversion of Binary to Decimal

To convert an unsigned binary number to a decimal number, add up the decimal values of the powers of 2 corresponding to bits which are set to 1 in the binary number. Algorithm C.1 shows a method to do this. Some examples of conversions from binary to decimal are given in figure C.2.

Since there are 2^n unique sequences of n bits, if all the possible bit sequences of length n are used, starting from zero, the largest number will be $2^n - 1$.

Algorithm C.1 (Binary to Decimal) *To convert a binary number to decimal.*

- Let X be a binary number, n digits in length, composed of bits $X_{n-1} \cdots X_0$.
- Let D be a decimal number.
- Let i be a counter.

1. Let $D = 0$.
2. Let $i = 0$.
3. While $i < n$ do:
   - If $X_i == 1$ (i.e. if bit i in X is 1), then set $D = (D + 2^i)$.
   - Set $i = (i + 1)$.

Figure C.2: Examples of Conversion from Binary to Decimal

Binary						**Decimal**
00000000	$=$	0	$=$	0	$=$	0
00000101	$=$	$2^2 + 2^0$	$=$	$4 + 1$	$=$	5
00000110	$=$	$2^2 + 2^1$	$=$	$4 + 2$	$=$	6
00101101	$=$	$2^5 + 2^3 + 2^2 + 2^0$	$=$	$32 + 8 + 4 + 1$	$=$	45
10110000	$=$	$2^7 + 2^5 + 2^4$	$=$	$128 + 32 + 16$	$=$	176

Conversion of Decimal to Binary

An algorithm for converting a decimal number to binary notation is given in algorithm C.2.

Algorithm C.2 (Decimal to Binary) *To convert a positive decimal number to binary.*

- Let X be an unsigned binary number, n digits in length.
- Let D be a positive decimal number, no larger than $2^n - 1$.
- Let i be a counter.

1. Let $X = 0$ (set all bits in X to 0).
2. Let $i = (n - 1)$.
3. While $i \geq 0$ do:
   (a) If $D \geq 2^i$, then
      - Set $X_i = 1$ (i.e. set bit i of X to 1).
      - Set $D = (D - 2^i)$.

   (b) Set $i = (i - 1)$.

Addition of Unsigned Binary Numbers

Addition of binary numbers can be done in exactly the same way as addition of decimal numbers, except that all of the operations are done in binary (base 2) rather than decimal (base 10). Algorithm C.3 gives a method which can be used to perform binary addition.

When algorithm C.3 terminates, if c is not 0, then an *overflow* has occurred– the resulting number is simply too large to be represented by an n-bit unsigned binary number.

C.1.2 Signed Binary Numbers

The major flaw with the representation that we've used for unsigned binary numbers is that it doesn't include a way to represent negative numbers.

Algorithm C.3 (Unsigned Binary Addition) *Addition of unsigned binary numbers.*

- Let A and B be a pair of n-bit binary numbers.
- Let X be a binary number which will hold the sum of A and B.
- Let c and $\hat{c}$ be carry bits.
- Let i be a counter.
- Let s be an integer.

1. Let $c = 0$.
2. Let $i = 0$.
3. While $i < n$ do:
   (a) Set $s = A_i + B_i + c$.
   (b) Set X_i and $\hat{c}$ according to the following rules:
      - If $s == 0$, then $X_i = 0$ and $\hat{c} = 0$.
      - If $s == 1$, then $X_i = 1$ and $\hat{c} = 0$.
      - If $s == 2$, then $X_i = 0$ and $\hat{c} = 1$.
      - If $s == 3$, then $X_i = 1$ and $\hat{c} = 1$.

   (c) Set $c = \hat{c}$.
   (d) Set $i = (i + 1)$.

There are a number of ways to extend the unsigned representation to include negative numbers. One of the easiest is to add an additional bit to each number that is used to represent the *sign* of the number– if this bit is 1, then the number is negative; otherwise the number is positive (or vice versa). This is analogous to the way that we write negative numbers in decimal– if the first symbol of the number is a negative sign, then the number is negative, otherwise the number is positive.

Unfortunately, when we try to adapt the algorithm for addition to work properly with this representation, this apparently simple method turns out to cause some trouble. Instead of simply adding the numbers together as we do with unsigned numbers, we now need to consider whether the numbers being added are positive or negative. If one number is positive and the other negative, then we actually need to do subtraction instead of addition, so we'll need to find an algorithm for subtraction. Furthermore, once we've done the subtraction, we need to compare the the unsigned magnitudes of the numbers to determine whether the result is positive or negative!

Luckily, there is a representation that allows us to represent negative numbers in such a way that addition (or subtraction) can be done easily, using algorithms very similar to the ones that we already have. The representation that we will use is called *two's complement* notation.

To introduce two's complement, we'll start by defining, in algorithm C.4, the algorithm that is used to compute the negation of a two's complement number.

Algorithm C.4 (Two's Complement Negation) *Negation of a two's complement number.*

1. Let $\bar{x}$ = the *logical complement* of x.

   The logical complement (also called the *one's complement*) is formed by flipping all the bits in the number, changing all of the 1 bits to 0, and vice versa.

2. Let $X = \bar{x} + 1$.

   If this addition *overflows*, then the overflow bit is discarded.

By the definition of two's complement, $X \equiv -x$.

Figure C.3 shows the process of negating several numbers. Note that the negation of zero is zero.

This representation has several important properties:

Figure C.3: Examples of Negation Using Two's Complement

	00000110	=	6
1's complement	11111001		
Add 1	11111010	=	-6
	11111010	=	-6
1's complement	00000101		
Add 1	00000110	=	6
	00000000	=	0
1's complement	11111111		
Add 1	00000000	=	0

- The leftmost (most significant) bit also serves as a sign bit; if 1, then the number is negative, if 0, then the number is positive or zero.
- The rightmost (least significant) bit of a number always determines whether or not the number is odd or even– if bit 0 is 0, then the number is even, otherwise the number is odd.
- The largest positive number that can be represented in two's complement notation in an n-bit binary number is $2^{n-1} - 1$. For example, if $n = 8$, then the largest positive number is $01111111 \quad = \quad 2^7 - 1 \quad = \quad 127$.
- Similarly, the "most negative" number is -2^{n-1}, so if $n = 8$, then it is 10000000, which is $-2^7 \quad = \quad -128$. Note that the negative of the most negative number (in this case, 128) cannot be represented in this notation.

Addition and Subtraction of Signed Binary Numbers

The same addition algorithm that was used for unsigned binary numbers also works properly for two's complement numbers.

$$
\begin{array}{rrcr}
 & 00000101 & = & 5 \\
+ & 11110101 & = & \text{-}11 \\
\hline
 & 11111010 & = & \text{-}6
\end{array}
$$

Subtraction is also done in a similar way: to subtract A from B, take the two's complement of A and then add this number to B.

The conditions for detecting overflow are different for signed and unsigned numbers, however. If we use algorithm C.3 to add two unsigned numbers, then if c is 1 when the addition terminates, this indicates that the result has an absolute value too large to fit the number of bits allowed. With signed numbers, however, c is not relevant, and an overflow occurs when the signs of both numbers being added are the same but the sign of the result is opposite. If the two numbers being added have opposite signs, however, then an overflow *cannot* occur.

For example, consider the sum of 1 and -1:

$$
\begin{array}{rrcrl}
 & 00000001 & = & 1 & \\
+ & 11111111 & = & \text{-}1 & \\
\hline
 & 00000000 & = & 0 & \textbf{Correct!}
\end{array}
$$

In this case, the addition will overflow, but it is not an error, since the result that we get (without considering the overflow) is exactly correct.

On the other hand, if we compute the sum of 127 and 1, then a serious error occurs:

$$
\begin{array}{rrcrl}
 & 01111111 & = & 127 & \\
+ & 00000001 & = & 1 & \\
\hline
 & 10000000 & = & \text{-}128 & \textbf{Uh-oh!}
\end{array}
$$

Therefore, we must be very careful when doing signed binary arithmetic that we take steps to detect bogus results. In general:

- If A and B are of the same sign, but $A + B$ is of the opposite sign, then an overflow or wraparound error has occurred.
- If A and B are of different signs, then $A + B$ will never overflow or wraparound.

Shifting Signed Binary Numbers

Another useful property of the two's complement notation is the ease with which numbers can be multiplied or divided by two. To multiply a number by two, simply shift the number "up" (to the left) by one bit, placing a 0 in the least significant bit. To divide a number in half, simply shift the the number "down" (to the right) by one bit (but do not change the sign bit).

Note that in the case of odd numbers, the effect of shifting to the right one bit is like dividing in half, rounded towards $-\infty$, so that 51 shifted to the right one bit becomes 25, while -51 shifted to the right one bit becomes -26.

	00000001	=	1
Double	00000010	=	2
Halve	00000000	=	0
	00110011	=	51
Double	01100110	=	102
Halve	00011001	=	25
	11001101	=	-51
Double	10011010	=	-102
Halve	11100110	=	-26

Hexadecimal Notation

Writing numbers in binary notation can soon get tedious, since even relatively small numbers require many binary digits to express. A more compact notation, called *hexadecimal* (base 16), is usually used to express large binary numbers. In hexadecimal, each digit represents four unsigned binary digits.

Another notation, which is not as common currently, is called *octal* and uses base eight to represent groups of three bits. Figure C.4 show examples of binary, decimal, octal, and hexadecimal numbers.

For example, the number $\mathtt{200}_{10}$ can be written as $\mathtt{11001000}_2$, $\mathtt{C8}_{16}$, or $\mathtt{310}_8$.

Figure C.4: Hexadecimal and Octal

Binary	0000	0001	0010	0011	0100	0101	0110	0111
Decimal	0	1	2	3	4	5	6	7
Hex	0	1	2	3	4	5	6	7
Octal	0	1	2	3	4	5	6	7

Binary	1000	1001	1010	1011	1100	1101	1110	1111
Decimal	8	9	10	11	12	13	14	15
Hex	8	9	A	B	C	D	E	F
Octal	10	11	12	13	14	15	16	17

C.2 Representing Characters

Just as sequences of bits can be used to represent numbers, they can also be used to represent the letters of the alphabet, as well as other characters.

Since all sequences of bits represent numbers, one way to think about representing characters by sequences of bits is to choose a number that corresponds to each character. The most popular correspondence currently is the ASCII character set. ASCII, which stands for the American Standard Code for Information Interchange, uses 7-bit integers to represent characters, using the correspondence shown in table C.5.

When the ASCII character set was chosen, some care was taken to organize the way that characters are represented in order to make them easy for a computer to manipulate. For example, all of the letters of the alphabet are arranged in order, so that sorting characters into alphabetical order is the same as sorting in numerical order. In addition, different classes of characters are arranged to have useful relations. For example, to convert the code for a lowercase letter to the code for the same letter in uppercase, simply set the 6th bit of the code to 0 (or subtract 32). ASCII is by no means the only character set to have similar useful properties, but it has emerged as the standard.

The ASCII character set does have some important limitations, however. One problem is that the character set only defines the representations of the characters used in written English. This causes problems with using ASCII to represent other written languages. In particular, there simply aren't enough bits to represent all the written characters of languages with a larger number of characters (such as Chinese or Japanese). Already new character sets which address these problems (and can be used to represent characters of many languages

Figure C.5: The ASCII Character Set

00 NUL	01 SOH	02 STX	03 ETX	04 EOT	05 ENQ	06 ACK	07 BEL
08 BS	09 HT	0A NL	0B VT	0C NP	0D CR	0E SO	0F SI
10 DLE	11 DC1	12 DC2	13 DC3	14 DC4	15 NAK	16 SYN	17 ETB
18 CAN	19 EM	1A SUB	1B ESC	1C FS	1D GS	1E RS	1F US
20 SP	21 !	22 "	23 #	24 $	25 %	26 &	27 ’
28 (	29)	2A *	2B +	2C ,	2D -	2E .	2F /
30 0	31 1	32 2	33 3	34 4	35 5	36 6	37 7
38 8	39 9	3A :	3B ;	3C <	3D =	3E >	3F ?
40 @	41 A	42 B	43 C	44 D	45 E	46 F	47 G
48 H	49 I	4A J	4B K	4C L	4D M	4E N	4F O
50 P	51 Q	52 R	53 S	54 T	55 U	56 V	57 W
58 X	59 Y	5A Z	5B [	5C	5D]	5E ^	5F _
60 `	61 a	62 b	63 c	64 d	65 e	66 f	67 g
68 h	69 i	6A j	6B k	6C l	6D m	6E n	6F o
70 p	71 q	72 r	73 s	74 t	75 u	76 v	77 w
78 x	79 y	7A z	7B {	7C |	7D }	7E ~	7F DEL

side by side) are being proposed, and eventually there will unquestionably be a shift away from ASCII to a new multilanguage standard[1].

C.3 Representing Programs

Just as groups of bits can be used to represent numbers, they can also be used to represent instructions for a computer to perform. Unlike the two's complement notation for integers, which is a standard representation used by nearly all computers, the representation of instructions, and even the set of instructions, varies widely from one type of computer to another.

The MIPS architecture, which is the focus of later chapters in this document, uses a relatively simple and straightforward representation. Each instruction is exactly 32 bits in length, and consists of several bit fields, as depicted in figure C.6.

[1]This shift will break many, many existing programs. Converting all of these programs will keep many, many programmers busy for some time.

Figure C.6: MIPS R2000 Instruction Formats

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
Register	op	reg1	reg2	des	shift	funct
Immediate	op	reg1	reg2	16-bit constant		
Jump	op	26-bit constant				

The first six bits (reading from the left, or high-order bits) of each instruction are called the `op` field. The op field determines whether the instruction is a *register*, *immediate*, or *jump* instruction, and how the rest of the instruction should be interpreted. Depending on what the op is, parts of the rest of the instruction may represent the names of registers, constant memory addresses, 16-bit integers, or other additional qualifiers for the op.

If the op field is 0, then the instruction is a register instruction, which generally perform an arithmetic or logical operations. The *funct* field specifies the operation to perform, while the *reg1* and *reg2* represent the registers to use as operands, and the *des* field represents the register in which to store the result. For example, the 32-bit hexadecimal number `0x02918020` represents, in the MIPS instruction set, the operation of adding the contents of registers 20 and 17 and placing the result in register 16.

Field	op	reg1	reg2	des	shift	funct
Width	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
Values	0	20	17	16	0	`add`
Binary	000000	10100	10001	10000	00000	100000

If the op field is not 0, then the instruction may be either an *immediate* or *jump* instruction, depending on the value of the op field.

C.4 Memory Organization

We've seen how sequences of binary digits can be used to represent numbers, characters, and instructions. In a computer, these binary digits are organized and manipulated in discrete groups, and these groups are said to be the *memory* of the computer.

C.4.1 Units of Memory

The smallest of these groups, on most computers, is called a *byte*. On nearly all currently popular computers a byte is composed of 8 bits.

The next largest unit of memory is usually composed of 16 bits. What this unit is called varies from computer to computer– on smaller machines, this is often called a *word*, while on newer architectures that can handle larger chunks of data, this is called a *halfword*.

The next largest unit of memory is usually composed of 32 bits. Once again, the name of this unit varies– on smaller machines, it is referred to as a *long*, while on newer and larger machines it is called a *word*.

Finally, on the newest machines, the computer also can handle data in groups of 64 bits. On a smaller machine, this is known as a *quadword*, while on a larger machine this is known as a *long*.

Historical Perspective

There have been architectures that have used nearly every imaginable word size– from 6-bit bytes to 9-bit bytes, and word sizes ranging from 12 bits to 48 bits. There are even a few architectures that have no fixed word size at all (such as the CM-2) or word sizes that can be specified by the operating system at runtime.

Over the years, however, most architectures have converged on 8-bit bytes and 32-bit longwords. An 8-bit byte is a good match for the ASCII character set (which has some popular extensions that require 8 bits), and a 32-bit word has been, at least until recently, large enough for most practical purposes.

C.4.2 Addresses and Pointers

Each unique byte[2] of the computer's memory is given a unique identifier, known as its *address*. The *address of* a piece of memory is often refered to as a *pointer to* that piece of memory– the two terms are synonymous, although there are many contexts where one is commonly used and the other is not.

The memory of the computer itself can often be thought of as a large array (or group of arrays) of bytes of memory. In this model, the address of each byte of memory is simply the index of the memory array location where that byte is stored.

[2]In some computers, the smallest distinct unit of memory is not a byte. For the sake of simplicity, however, this section assumes that the smallest distinct unit of memory on the computer in question is a byte.

C.4.3 Summary

In this chapter, we've seen how computers represent integers using groups of bits, and how basic arithmetic and other operations can be performed using this representation.

We've also seen how the integers or groups of bits can be used to represent several different kinds of data, including written characters (using the ASCII character codes), instructions for the computer to execute, and addresses or pointers, which can be used to reference other data.

There are also many other ways that information can be represented using groups of bits, including representations for rational numbers (usually by a representation called *floating point*), irrational numbers, graphics, arbitrary character sets, and so on. These topics, unfortunately, are beyond the scope of this chapter.

C.5 Exercises

1. Complete the following table:

Decimal	123				
Binary		01101100			
Octal			143		
Hex				3D	
ASCII					Z

2. Invent an algorithm for multiplying two unsigned binary numbers. You may find it easiest to start by thinking about multiplication of decimal numbers (there are other ways as well, but you should start on familiar ground).

3. (a) Invent an algorithm for dividing two unsigned binary numbers. You may find it easiest to start by thinking about long division of decimal numbers.

   (b) A user complains that the division algorithm you invented to solve the previous part of this problem is too slow. For her application, she would prefer an algorithm that gets an answer that is "reasonably close" to the right answer, but which may take considerably less time to compute. Invent an algorithm that has this property. Find the relationship between "reasonably close" and the speed of your algorithm.