Answer

11001000110001011101001111010111

CHAPTER FOUR

4.0 MODES OF DATA REPRESENTATION

Most data structures are abstract structures and are implemented by the programmer with a series of assembly language instructions. Many cardinal data types (bits, bit strings, bit slices, binary integers, binary floating point numbers, binary encoded decimals, binary addresses, characters, etc.) are implemented directly in hardware for at least parts of the instruction set. Some processors also implement some data structures in hardware for some instructions — for example, most processors have a few instructions for directly manipulating character strings.

An assembly language programmer has to know how the hardware implements these cardinal data types. Some examples: Two basic issues are bit ordering (big endian or little endian) and number of bits (or bytes). The assembly language programmer must also pay attention to word length and optimum (or required) addressing boundaries. Composite data types will also include details of hardware implementation, such as how many bits of mantissa, characteristic, and sign, as well as their order.

4.1 INTEGER REPRESENTATION

Sign-magnitude is the simplest method for representing signed binary numbers. One bit (by universal convention, the highest order or leftmost bit) is the sign bit, indicating positive or

negative, and the remaining bits are the absolute value of the binary integer. Sign-magnitude is simple for representing binary numbers, but has the drawbacks of two different zeros and much more complicates (and therefore, slower) hardware for performing addition, subtraction, and any binary integer operations other than complement (which only requires a sign bit change).

In sign magnitude, the sign bit for positive is represented by 0 and the sign bit for negative is represented by 1.

Examples:

1. Convert +52 to binary using an 8 bits machine

Answer: The binary equivalence of 52 is 110100 but 0 is used to represent positive magnitude, hence 0 is added to the front of this binary equivalence. This makes a total of 7bits, since we are working on an eight bit machine, we have to pad the numbers with 0 so as to make it a total of 8bits. Thus the binary equivalence 0f 52 is 00110100.

2. Convert -52 to binary using an 8 bits machine

Answer: The binary equivalence of 52 is 110100 but 1 is used to represent positive magnitude, hence 1 is added to the front of this binary equivalence. This makes a total of 7bits, since we are working on an eight bit machine, we have to pad the numbers with 0 so as to make it a total of 8bits. In this case, the sign bit has to come first and the padded 0 follows. Thus the binary equivalence 0f -52 is 10110100.

Exercise:

a. Convert +47 to binary on an 8 bits machine
b. Convert -17 to binary on an 8 bits machine
c. Convert -567 on a 16 bits machine

In **one's complement** representation, positive numbers are represented in the "normal" manner (same as unsigned integers with a zero sign bit), while negative numbers are represented by complementing all of the bits of the absolute value of the number. Numbers are negated by complementing all bits. Addition of two integers is peformed by treating the numbers as unsigned integers (ignoring sign bit), with a carry out of the leftmost bit position being added to the least significant bit (technically, the carry bit is always added to the least significant bit, but when it is zero, the add has no effect). The ripple effect of adding the carry bit can almost double the time to do an addition. And there are still two zeros, a positive zero (all zero bits) and a negative zero (all one bits).

The I's complement form of any binary number is simply by changing each 0 in the number to a 1 and vice versa.

Examples

1. Find the 1's complement of -7

Answer: -7 in the actual representation without considering the machine bit is 1111. To change this to 1's complement, the sign bit has to be retained and other bits have to be inverted. Thus, the answer is: 1000. 1 denotes the sign bit.

2. Find the1's complement of -7.25

The actual magnitude representation of -7.25 is 1111.01 but retaining the sign bits and inverting the other bits gives: 1000.10

Exercises

1. Find the one's complement of -47
2. Find the one's complement of -467 and convert the answer to hexcadecimal.

In **two's complement** representation, positive numbers are represented in the "normal" manner (same as unsigned integers with a zero sign bit), while negative numbers are represented by complementing all of the bits of the absolute value of the number and adding one. Negation of

a negative number in two's complement representation is accomplished by complementing all of the bits and adding one. Addition is performed by adding the two numbers as unsigned integers and ignoring the carry. Two's complement has the further advantage that there is only one zero (all zero bits). Two's complement representation does result in one more negative number (all one bits) than positive numbers.

Two's complement is used in just about every binary computer ever made. Most processors have one more negative number than positive numbers. Some processors use the "extra" neagtive number (all one bits) as a special indicator, depicting invalid results, not a number (NaN), or other special codes.

2's complement is used to represent negative numbers because it allows us to perform the operation of subtraction by actually performing addition. The 2's complement of a binary number is the addition of 1 to the rightmost bit of its 1's complement equivalence.

Examples

1. Convert -52 to its 2's complement

The 1's complement of -52 is 11001011

To convert this to 2's complement we have

11001011

+ 1

11001100

2. Convert -419 to 2's complement and hence convert the result to hexadecimal

The sign magnitude representation of -419 on a 16 bit machine is 1000000110100011

The I's complement is 1111111001011100

To convert this to 2's complement, we have:

1111111001011100

+ 1

1111111001011101

Dividing the resulting bits into four gives an hexadecimal equivalence of $FE5D_{16}$

In general, if a number $a_1, a_2 \ldots\ldots a_n$ is in base b, then we by an form its b's complement by subtracting each digit of the number from b-1 and adding 1 to the result.

Example: Find the 8's complement of 7245_8

Answer:

7777

-7245

0532

+ 1

0533

Thus the 8's complement of 7245_8 is 0533

ADDITION OF NUMBERS USING 2'S COMPLEMENT

1. Add +9 and +4 for 5 bits machine

= 01001

00100

01101

01101 is equivalent to +13

2. Add +9 and -4 on a 5 bits machine

+9 in its sign magnitude form is 01001

-4 in the sign magnitude form is10100

Its 1's complement equivalence is 11011

Its 2's complement equivalence is 11100 (by adding 1 to its 1's complement)

Thus addition +9 and -4 which is also +9 + (-4)

This gives

01001

+ 11100

100101

Since we are working on a 5bits machine, the last leftmost bit is an off-bit, thus it is neglected. The resulting answer is thus 00101. This is equivalent to +5

3. Add -9 and +4

The 2's complement of -9 is 10111

The sign magnitude of +4 is 00100

This gives;

10111

+ 00100

11011

The sum has a sign bit of 1 indicating a negative number, since the sum is negative, it is in its 2's complement, thus the last 4 bits 1.e 1011 actually represent the 2's complement of the sum. To find the true magnitude of the sum, we 2's complement the sum

11011 to 1's complement is 10100

2's complement is 1

10101

This is equivalent to -5

Exercise:

1. Using the last example's concept add -9 and -4.

The expected answer is 11101

2. Add -9 and +9

The expected answer is 100000, the last leftmost bit is an off bit thus, it is truncated.

In **unsigned** representation, only positive numbers are represented. Instead of the high order bit being interpreted as the sign of the integer, the high order bit is part of the number. An unsigned number has one power of two greater range than a signed number (any representation) of the same number of bits. A comparison of the integer arithmetic forms is shown below;

bit pattern	*sign-mag.*	*one's comp.*	*two's comp*	*unsigned*
000	0	0	0	0
001	1	1	1	1
010	2	2	2	2
011	3	3	3	3
100	-0	-3	-4	4
101	-1	-2	-3	5
110	-2	-1	-2	6
111	-3	-0	-1	7

4.2 FLOATING POINT REPRESENTATIONS

Floating point numbers are the computer equivalent of "scientific notation" or "engineering notation". A floating point number consists of a fraction (binary or decimal) and an exponent (bianry or decimal). Both the fraction and the exponent each have a sign (positive or negative).

In the past, processors tended to have proprietary floating point formats, although with the development of an IEEE standard, most modern processors use the same format. Floating point numbers are almost always binary representations, although a few early processors had (binary coded) decimal representations. Many processors (especially early mainframes and early

microprocessors) did not have any hardware support for floating point numbers. Even when commonly available, it was often in an optional processing unit (such as in the IBM 360/370 series) or coprocessor (such as in the Motorola 680x0 and pre-Pentium Intel 80x86 series).

Hardware floating point support usually consists of two sizes, called **single precision** (for the smaller) and **double precision** (for the larger). Usually the double precision format had twice as many bits as the single precision format (hence, the names single and double). Double precision floating point format offers greater range and precision, while single precision floating point format offers better space compaction and faster processing.

F_floating format (single precision floating), DEC VAX, 32 bits, the first bit (high order bit in a register, first bit in memory) is the sign magnitude bit (one=negative, zero=positive or zero), followed by 15 bits of an excess 128 binary exponent, followed by a normalized 24-bit fraction with the redundant most significant fraction bit not represented. Zero is represented by all bits being zero (allowing the use of a longword CLR to set a F_floating number to zero). Exponent values of 1 through 255 indicate true binary exponents of -127 through 127. An exponent value of zero together with a sign of zero indicate a zero value. An exponent value of zero together with a sign bit of one is taken as reserved (which produces a reserved operand fault if used as an operand for a floating point instruction). The magnitude is an approximate range of $.29*10^{-38}$ through $1.7*10^{38}$. The precision of an F_floating datum is approximately one part in 2^{23}, or approximately seven (7) decimal digits).

32 bit floating format (single precision floating), AT&T DSP32C, 32 bits, the first bit (high order bit in a register, first bit in memory) is the sign magnitude bit (one=negative, zero=positive or zero), followed by 23 bits of a normalized two's complement fractional part of the mantissa, followed by an eight bit exponent. The magnitude of the mantissa is always normalized to lie between 1 and 2. The floating point value with exponent equal to zero is reserved to represent the number zero (the sign and mantissa bits must also be zero; a zero exponent with a nonzero sign and/or mantissa is called a "dirty zero" and is never generated by hardware; if a dirty zero is an operand, it is treated as a zero). The range of nonzero positive floating point numbers is $N = [1 * 2^{-127}, [2-2^{-23}] * 2^{127}]$ inclusive. The range of nonzero negative floating point numbers is $N = [-[1 + 2^{-23}] * 2^{-127}, -2 * 2^{127}]$ inclusive.

40 bit floating format (extended single precision floating), AT&T DSP32C, 40 bits, the first bit (high order bit in a register, first bit in memory) is the sign magnitude bit (one=negative, zero=positive or zero), followed by 31 bits of a normalized two's complement fractional part of the mantissa, followed by an eight bit exponent. This is an internal format used by the floating point adder, accumulators, and certain DAU units. This format includes an additional eight guard bits to increase accuracy of intermediate results.

D_floating format (double precision floating), DEC VAX, 64 bits, the first bit (high order bit in a register, first bit in memory) is the sign magnitude bit (one=negative, zero=positive or zero), followed by 15 bits of an excess 128 binary exponent, followed by a normalized 48-bit fraction with the redundant most significant fraction bit not represented. Zero is represented by all bits being zero (allowing the use of a quadword CLR to set a D_floating number to zero). Exponent values of 1 through 255 indicate true binary exponents of -127 through 127. An exponent value of zero together with a sign of zero indicate a zero value. An exponent value of zero together with a sign bit of one is taken as reserved (which produces a reserved operand fault if used as an operand for a floating point instruction). The magnitude is an approximate range of $.29*10^{-38}$ through $1.7*10^{38}$. The precision of an D_floating datum is approximately one part in 2^{55}, or approximately 16 decimal digits).

CHAPTER FIVE

COMPUTER INSTRUCTION SET

5.0 An **instruction set** is a list of all the instructions, and all their variations, that a processor (or in the case of a virtual machine, an interpreter) can execute.

- Arithmetic such as **add** and **subtract**
- Logic instructions such as **and**, **or**, and **not**
- Data instructions such as **move**, **input**, **output**, **load**, and **store**
- Control flow instructions such as **goto**, **if ... goto**, **call**, and **return**.

An **instruction set**, or **instruction set architecture** (ISA), is the part of the computer architecture related to programming, including the native data types, instructions, registers,