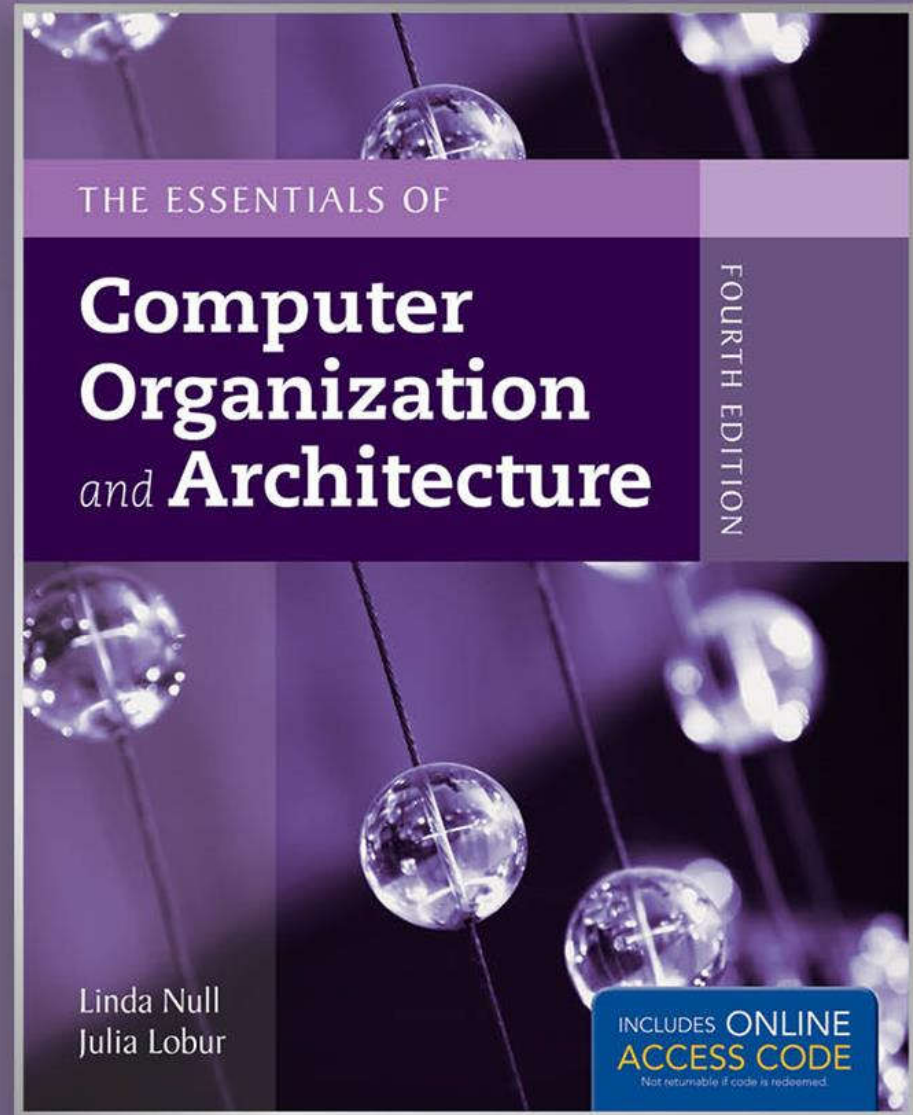


Chapter 2

Data Representation in Computer Systems



Chapter 2 Objectives

- Understand the fundamentals of numerical data representation and manipulation in digital computers.
- Master the skill of converting between various radix systems.
- Understand how errors can occur in computations because of overflow and truncation.

Chapter 2 Objectives

- Understand the fundamental concepts of floating-point representation.
- Gain familiarity with the most popular character codes.
- Understand the concepts of error detecting and correcting codes.

2.1 Introduction

- A *bit* is the most basic unit of information in a computer.
 - It is a state of “on” or “off” in a digital circuit.
 - Sometimes these states are “high” or “low” voltage instead of “on” or “off.”
- A *byte* is a group of eight bits.
 - A byte is the smallest possible *addressable* unit of computer storage.
 - The term, “addressable,” means that a particular byte can be retrieved according to its location in memory.

2.1 Introduction

- A *word* is a contiguous group of bytes.
 - Words can be any number of bits or bytes.
 - Word sizes of 16, 32, or 64 bits are most common.
 - In a word-addressable system, a word is the smallest addressable unit of storage.
- A group of four bits is called a *nibble*.
 - Bytes, therefore, consist of two nibbles: a “high-order nibble,” and a “low-order” nibble.

2.2 Positional Numbering Systems

- Bytes store numbers using the position of each bit to represent a power of 2.
 - The binary system is also called the base-2 system.
 - Our decimal system is the base-10 system. It uses powers of 10 for each position in a number.
 - Any integer quantity can be represented exactly using any base (or *radix*).

2.2 Positional Numbering Systems

- The decimal number 947 in powers of 10 is:

$$9 \times 10^2 + 4 \times 10^1 + 7 \times 10^0$$

- The decimal number 5836.47 in powers of 10 is:

$$\begin{aligned} &5 \times 10^3 + 8 \times 10^2 + 3 \times 10^1 + 6 \times 10^0 \\ &+ 4 \times 10^{-1} + 7 \times 10^{-2} \end{aligned}$$

2.2 Positional Numbering Systems

- The binary number 11001 in powers of 2 is:

$$\begin{aligned} & 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 16 + 8 + 0 + 0 + 1 = 25 \end{aligned}$$

- When the radix of a number is something other than 10, the base is denoted by a subscript.
 - Sometimes, the subscript 10 is added for emphasis:

$$11001_2 = 25_{10}$$

2.3 Converting Between Bases

- Because binary numbers are the basis for all data representation in digital computer systems, it is important that you become proficient with this radix system.
- Your knowledge of the binary numbering system will enable you to understand the operation of all computer components as well as the design of instruction set architectures.

2.3 Converting Between Bases

- In an earlier slide, we said that every integer value can be represented exactly using any radix system.
- There are two methods for radix conversion: the subtraction method and the division remainder method.
- The subtraction method is more intuitive, but cumbersome. It does, however reinforce the ideas behind radix mathematics.

2.3 Converting Between Bases

- **Suppose we want to convert the decimal number 190 to base 3.**
 - We know that $3^5 = 243$ so our result will be less than six digits wide. The largest power of 3 that we need is therefore $3^4 = 81$, and $81 \times 2 = 162$.
 - Write down the 2 and subtract 162 from 190, giving 28.

$$\begin{array}{r} 190 \\ - 162 \\ \hline 28 \end{array} = 3^4 \times 2$$

2.3 Converting Between Bases

- **Converting 190 to base 3...**
 - The next power of 3 is $3^3 = 27$. We'll need one of these, so we subtract 27 and write down the numeral 1 in our result.
 - The next power of 3, $3^2 = 9$, is too large, but we have to assign a placeholder of zero and carry down the 1.

$$\begin{array}{r} 190 \\ - 162 \\ \hline 28 \end{array} = 3^4 \times 2$$
$$\begin{array}{r} - 27 \\ \hline 1 \end{array} = 3^3 \times 1$$
$$\begin{array}{r} - 0 \\ \hline 1 \end{array} = 3^2 \times 0$$

2.3 Converting Between Bases

- **Converting 190 to base 3...**

- $3^1 = 3$ is again too large, so we assign a zero placeholder.
- The last power of 3, $3^0 = 1$, is our last choice, and it gives us a difference of zero.
- Our result, reading from top to bottom is:

$$190_{10} = 21001_3$$

190		
<u>- 162</u>	$= 3^4 \times$	2
28		
<u>- 27</u>	$= 3^3 \times$	1
1		
<u>- 0</u>	$= 3^2 \times$	0
1		
<u>- 0</u>	$= 3^1 \times$	0
1		
<u>- 1</u>	$= 3^0 \times$	1
0		

2.3 Converting Between Bases

- Another method of converting integers from decimal to some other radix uses division.
- This method is mechanical and easy.
- It employs the idea that successive division by a base is equivalent to successive subtraction by powers of the base.
- Let's use the division remainder method to again convert 190 in decimal to base 3.

2.3 Converting Between Bases

- **Converting 190 to base 3...**
 - First we take the number that we wish to convert and divide it by the radix in which we want to express our result.
 - In this case, 3 divides 190 63 times, with a remainder of 1.
 - Record the quotient and the remainder.

$$\begin{array}{r|l} 3 & 190 \\ \hline & 63 \end{array} \quad 1$$

2.3 Converting Between Bases

- **Converting 190 to base 3...**
 - 63 is evenly divisible by 3.
 - Our remainder is zero, and the quotient is 21.

$$\begin{array}{r} 3 \overline{) 190} \quad 1 \\ 3 \overline{) 63} \quad 0 \\ \quad 21 \end{array}$$

2.3 Converting Between Bases

- **Converting 190 to base 3...**
 - Continue in this way until the quotient is zero.
 - In the final calculation, we note that 3 divides 2 zero times with a remainder of 2.
 - Our result, reading from bottom to top is:

$$190_{10} = 21001_3$$

3		190	1
3		63	0
3		21	0
3		7	1
3		2	2
		0	

2.3 Converting Between Bases

- Fractional values can be approximated in all base systems.
- Unlike integer values, fractions do not necessarily have exact representations under all radices.
- The quantity $\frac{1}{2}$ is exactly representable in the binary and decimal systems, but is not in the ternary (base 3) numbering system.

2.3 Converting Between Bases

- Fractional decimal values have nonzero digits to the right of the decimal point.
- Fractional values of other radix systems have nonzero digits to the right of the *radix point*.
- Numerals to the right of a radix point represent negative powers of the radix:

$$0.47_{10} = 4 \times 10^{-1} + 7 \times 10^{-2}$$

$$\begin{aligned} 0.11_2 &= 1 \times 2^{-1} + 1 \times 2^{-2} \\ &= \frac{1}{2} + \frac{1}{4} \\ &= 0.5 + 0.25 = 0.75 \end{aligned}$$

2.3 Converting Between Bases

- As with whole-number conversions, you can use either of two methods: a subtraction method or an easy multiplication method.
- The subtraction method for fractions is identical to the subtraction method for whole numbers. Instead of subtracting positive powers of the target radix, we subtract negative powers of the radix.
- We always start with the largest value first, n^{-1} , where n is our radix, and work our way along using larger negative exponents.

2.3 Converting Between Bases

- The calculation to the right is an example of using the subtraction method to convert the decimal 0.8125 to binary.
 - Our result, reading from top to bottom is:
$$0.8125_{10} = 0.1101_2$$
 - Of course, this method works with any base, not just binary.

$$\begin{array}{rcl} 0.8125 & & \\ - 0.5000 & = 2^{-1} \times 1 & \\ \hline 0.3125 & & \\ - 0.2500 & = 2^{-2} \times 1 & \\ \hline 0.0625 & & \\ - 0 & = 2^{-3} \times 0 & \\ \hline 0.0625 & & \\ - 0.0625 & = 2^{-4} \times 1 & \\ \hline 0 & & \end{array}$$

2.3 Converting Between Bases

- **Using the multiplication method to convert the decimal 0.8125 to binary, we multiply by the radix 2.**
 - The first product carries into the units place.

$$\begin{array}{r} .8125 \\ \times \quad 2 \\ \hline 1.6250 \end{array}$$

2.3 Converting Between Bases

- **Converting 0.8125 to binary . . .**
 - Ignoring the value in the units place at each step, continue multiplying each fractional part by the radix.

$$\begin{array}{r} .8125 \\ \times \quad 2 \\ \hline 1.6250 \end{array}$$

$$\begin{array}{r} .6250 \\ \times \quad 2 \\ \hline 1.2500 \end{array}$$

$$\begin{array}{r} .2500 \\ \times \quad 2 \\ \hline 0.5000 \end{array}$$

2.3 Converting Between Bases

- **Converting 0.8125 to binary . . .**

- You are finished when the product is zero, or until you have reached the desired number of binary places.
- Our result, reading from top to bottom is:

$$0.8125_{10} = 0.1101_2$$

- This method also works with any base. Just use the target radix as the multiplier.

$$\begin{array}{r} .8125 \\ \times \quad 2 \\ \hline 1.6250 \\ \\ .6250 \\ \times \quad 2 \\ \hline 1.2500 \\ \\ .2500 \\ \times \quad 2 \\ \hline 0.5000 \\ \\ .5000 \\ \times \quad 2 \\ \hline 1.0000 \end{array}$$

2.3 Converting Between Bases

- The binary numbering system is the most important radix system for digital computers.
- However, it is difficult to read long strings of binary numbers -- and even a modestly-sized decimal number becomes a very long binary number.
 - For example: $11010100011011_2 = 13595_{10}$
- For compactness and ease of reading, binary values are usually expressed using the hexadecimal, or base-16, numbering system.

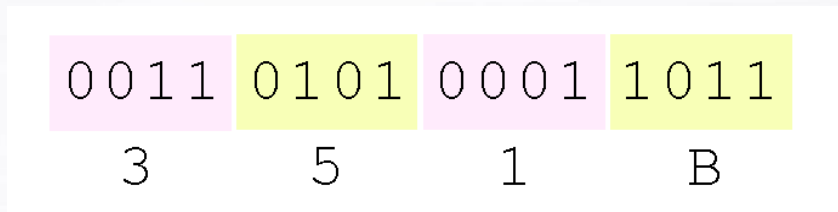
2.3 Converting Between Bases

- The hexadecimal numbering system uses the numerals 0 through 9 and the letters A through F.
 - The decimal number 12 is C_{16} .
 - The decimal number 26 is $1A_{16}$.
- It is easy to convert between base 16 and base 2, because $16 = 2^4$.
- Thus, to convert from binary to hexadecimal, all we need to do is group the binary digits into groups of four.

A group of four binary digits is called a hextet

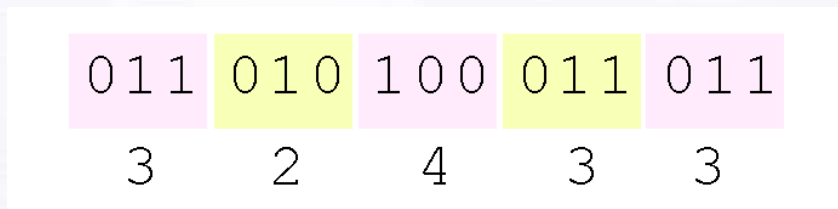
2.3 Converting Between Bases

- Using groups of hextets, the binary number 11010100011011_2 ($= 13595_{10}$) in hexadecimal is:



If the number of bits is not a multiple of 4, pad on the left with zeros.

- Octal (base 8) values are derived from binary by using groups of three bits ($8 = 2^3$):



Octal was very useful when computers used six-bit words.

2.4 Signed Integer Representation

- The conversions we have so far presented have involved only unsigned numbers.
- To represent signed integers, computer systems allocate the high-order bit to indicate the sign of a number.
 - The high-order bit is the leftmost bit. It is also called the most significant bit.
 - 0 is used to indicate a positive number; 1 indicates a negative number.
- The remaining bits contain the value of the number (but this can be interpreted different ways)

2.4 Signed Integer Representation

- There are three ways in which signed binary integers may be expressed:
 - Signed magnitude
 - One's complement
 - Two's complement
- In an 8-bit word, *signed magnitude* representation places the absolute value of the number in the 7 bits to the right of the sign bit.

$L(10)$

+4

-4

00000100

10000100

↳ 0 for plus

↳ 1 for minus

2.4 Signed Integer Representation

- For example, in 8-bit signed magnitude representation:
 - +3 is: 00000011
 - 3 is: 10000011
- Computers perform arithmetic operations on signed magnitude numbers in much the same way as humans carry out pencil and paper arithmetic.
 - Humans often ignore the signs of the operands while performing a calculation, applying the appropriate sign after the calculation is complete.

2.4 Signed Integer Representation

- Binary addition is as easy as it gets. You need to know only four rules:

$$0 + 0 = 0 \qquad 0 + 1 = 1$$

$$1 + 0 = 1 \qquad 1 + 1 = 10$$

- The simplicity of this system makes it possible for digital circuits to carry out arithmetic operations.
 - We will describe these circuits in Chapter 3.

Let's see how the addition rules work with signed magnitude numbers . . .

2.4 Signed Integer Representation

- Example:
 - Using signed magnitude binary arithmetic, find the sum of 75 and 46.
- First, convert 75 and 46 to binary, and arrange as a sum, but separate the (positive) sign bits from the magnitude bits.

$$\begin{array}{r} 0 \quad 1001011 \\ 0 + \underline{0101110} \end{array}$$

2.4 Signed Integer Representation

- Example:
 - Using signed magnitude binary arithmetic, find the sum of 75 and 46.
- Just as in decimal arithmetic, we find the sum starting with the rightmost bit and work left.

$$\begin{array}{r} 0 \quad 1001011 \\ 0 + 0101110 \\ \hline 1 \end{array}$$

2.4 Signed Integer Representation

- Example:
 - Using signed magnitude binary arithmetic, find the sum of 75 and 46.
- In the second bit, we have a carry, so we note it above the third bit.

$$\begin{array}{r} 0 \quad 1001011 \\ 0 + 0101110 \\ \hline \quad \quad \quad 01 \end{array}$$

2.4 Signed Integer Representation

- Example:
 - Using signed magnitude binary arithmetic, find the sum of 75 and 46.
- The third and fourth bits also give us carries.

$$\begin{array}{r} 10111 \\ 0 + 0101110 \\ \hline 101 \end{array}$$

2.4 Signed Integer Representation


- Example:
 - Using signed magnitude binary arithmetic, find the sum of 75 and 46.
- Once we have worked our way through all eight bits, we are done.

$$\begin{array}{r} \\ \\ 0 \\ 0 + 0101110 \\ \hline 0 \end{array}$$

In this example, we were careful to pick two values whose sum would fit into seven bits. If that is not the case, we have a problem.

2.4 Signed Integer Representation

- Example:
 - Using signed magnitude binary arithmetic, find the sum of 107 and 46.
- We see that the carry from the seventh bit *overflows* and is discarded, giving us the erroneous result: $107 + 46 = 25$.



The diagram illustrates a binary addition in signed magnitude format. A yellow circle containing the number '1' is positioned to the left of the first column of the addition. A pink arrow points from this circle to the seventh column of the addition, indicating a carry that has overflowed from the seventh bit. The addition is shown as follows:

$$\begin{array}{r} 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1 \\ 0 + 0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 1 \quad 0 \\ \hline 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \end{array}$$

2.4 Signed Integer Representation

- The signs in signed magnitude representation work just like the signs in pencil and paper arithmetic.

- **Example:** Using signed magnitude binary arithmetic, find the sum of - 46 and - 25.

$$\begin{array}{r} \\ 1 \\ 1 + \\ \hline 1 \end{array}$$

- Because the signs are the same, all we do is add the numbers and supply the negative sign when we are done.

2.4 Signed Integer Representation

- Mixed sign addition (or subtraction) is done the same way.
 - Example: Using signed magnitude binary arithmetic, find the sum of 46 and - 25.

$$\begin{array}{r}
 \begin{array}{ccccccc}
 & & 0 & 2 & & 0 & 2 \\
 0 & & 0 & \cancel{1} & 0 & 1 & 1 & \cancel{1} & 0
 \end{array} \\
 1 + \begin{array}{ccccccc}
 0 & 0 & 1 & 1 & 0 & 0 & 1
 \end{array} \\
 \hline
 0 \quad \begin{array}{ccccccc}
 0 & 0 & 1 & 0 & 1 & 0 & 1
 \end{array}
 \end{array}$$

- The sign of the result gets the sign of the number that is larger.
 - Note the “borrows” from the second and sixth bits.

get higher magnitude

Max + Min

39 take Max Sign

$$\underline{-19} + 13$$

Max mag

$$\begin{array}{r} ^1 ^2 \\ 1 \ 0010011 \\ - 0 \ 0001101 \\ \hline 1 \ 0000110 \end{array} \quad \begin{array}{l} -(19) \\ +(13) \\ -(6) \end{array}$$

$$\begin{array}{r} ^0 ^2 ^2 \\ 0 \ 0101110 \\ 1 \ 0011001 \\ \hline 0 \ 0010101 \end{array}$$

one's complement

$$\begin{array}{l} 3 - 4 \\ \hookrightarrow 3 + (-4) \\ 3 = 0011 \\ -4 = 1011 \end{array}$$

$$\begin{array}{r} ^1 ^1 \\ 3 \quad 0011 \\ -4 \quad 1011 \\ \hline -1 \quad 1110 \end{array}$$

$$\begin{array}{l} 7 - 6 \\ 7 + (-6) \\ 7 = 0111 \\ -6 = 1001 \end{array}$$

$$\begin{array}{r} \textcircled{1} ^1 ^1 \\ 7 \quad 0111 \\ -6 \quad 1001 \\ \hline 0000 \\ + \quad \quad \quad 1 \\ \hline 0001 \end{array}$$

2.4 Signed Integer Representation

- Signed magnitude representation is easy for people to understand, but it requires complicated computer hardware.
- Another disadvantage of signed magnitude is that it allows two different representations for zero: positive zero and negative zero.
- For these reasons (among others) computers systems employ *complement systems* for numeric value representation.

2.4 Signed Integer Representation

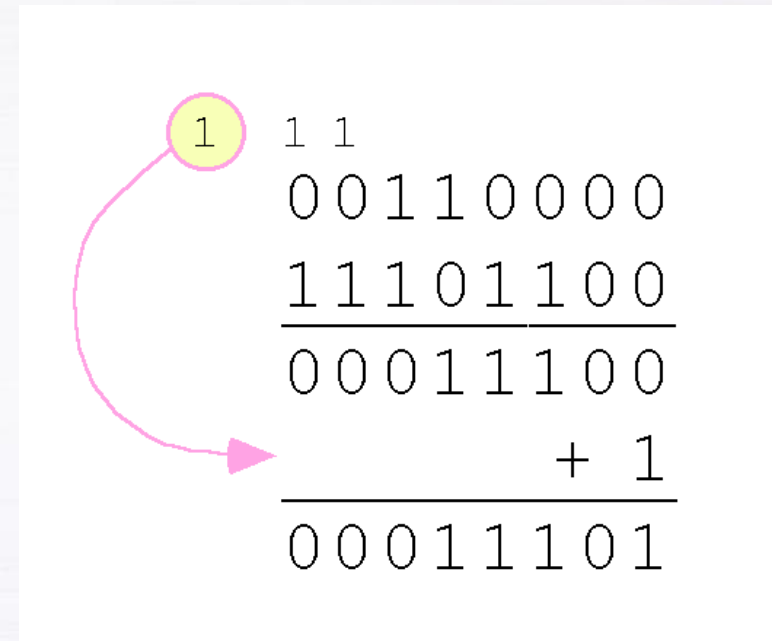
- In complement systems, negative values are represented by some difference between a number and its base.
- The *diminished radix complement* of a non-zero number N in base r with d digits is $(r^d - 1) - N$
- In the binary system, this gives us *one's complement*. It amounts to little more than flipping the bits of a binary number.

2.4 Signed Integer Representation

- For example, using 8-bit one's complement representation:
 - + 3 is: 00000011
 - 3 is: 11111100
- In one's complement representation, as with signed magnitude, negative values are indicated by a 1 in the high order bit.
- Complement systems are useful because they eliminate the need for subtraction. The difference of two values is found by adding the minuend to the complement of the subtrahend.

2.4 Signed Integer Representation

- With one's complement addition, the carry bit is “carried around” and added to the sum.
 - Example: Using one's complement binary arithmetic, find the sum of 48 and - 19



The diagram illustrates the one's complement addition of 48 and -19. The numbers are represented in 8-bit binary: 48 is 00110000 and -19 is 11101100. Their sum is 00011100. A carry bit of 1 is shown above the first two bits. A pink arrow indicates this carry bit being added to the least significant bit of the sum (00011100 + 1 = 00011101).

$$\begin{array}{r} 1\ 1 \\ 00110000 \\ 11101100 \\ \hline 00011100 \\ \quad + 1 \\ \hline 00011101 \end{array}$$

We note that 19 in binary is
so -19 in one's complement is:

00010011,
11101100.

2.4 Signed Integer Representation


- Although the “end carry around” adds some complexity, one’s complement is simpler to implement than signed magnitude.
- But it still has the disadvantage of having two different representations for zero: **positive zero** and **negative zero**.
- Two’s complement solves this problem.
- Two’s complement is the radix complement of the binary numbering system; the *radix complement* of a non-zero number N in base r with d digits is $r^d - N$.

2.4 Signed Integer Representation

- To express a value in two's complement representation:
 - If the number is positive, just convert it to binary and you're done.
 - If the number is negative, find the one's complement of the number and then add 1.
- Example:
 - In 8-bit binary, 3 is:
00000011
 - -3 using one's complement representation is:
11111100
 - Adding 1 gives us -3 in two's complement form:
11111101.

2.4 Signed Integer Representation

- With two's complement arithmetic, all we do is add our two binary numbers. Just discard any carries emitting from the high order bit.
- Example: Using one's complement binary arithmetic, find the sum of 48 and - 19.


$$\begin{array}{r} 11 \\ 00110000 \\ + 11101101 \\ \hline 00011101 \end{array}$$

We note that 19 in binary is: **00010011**,
so -19 using one's complement is: **11101100**,
and -19 using two's complement is: **11101101**.

2's Comp

$$7 - 6$$

$$7 + (-6)$$

$$7 = 0111$$

$$-6 = 1010$$

$$\begin{array}{r} \text{Discard} \\ 1 \\ 0111 \end{array}$$

$$\begin{array}{r} + \quad 1010 \\ \hline 0001 \end{array}$$

2.4 Signed Integer Representation

- Excess- M representation (also called offset binary representation) is another way for unsigned binary values to represent signed integers.
 - Excess- M representation is intuitive because the binary string with all 0s represents the smallest number, whereas the binary string with all 1s represents the largest value.
- An unsigned binary integer M (called the *bias*) represents the value 0, whereas all zeroes in the bit pattern represents the integer $-M$.
- The integer is interpreted as positive or negative depending on where it falls in the range.

2.4 Signed Integer Representation

- If n bits are used for the binary representation, we select the bias in such a manner that we split the range equally.
- Typically we choose a bias of $2^{n-1} - 1$.
 - For example, if we were using 4-bit representation, the bias should be $2^{4-1} - 1 = 7$.
- Just as with signed magnitude, one's complement, and two's complement, there is a specific range of values that can be expressed in n bits.

2.4 Signed Integer Representation

- The unsigned binary value for a signed integer using excess- M representation is determined simply by adding M to that integer.
 - For example, assuming that we are using excess-7 representation, the integer 0_{10} is represented as $0 + 7 = 7_{10} = 0111_2$.
 - The integer 3_{10} is represented as $3 + 7 = 10_{10} = 1010_2$.
 - The integer -7 is represented as $-7 + 7 = 0_{10} = 0000_2$.
 - To find the decimal value of the excess-7 binary number 1111_2 subtract 7: $1111_2 = 15_{10}$ and $15 - 7 = 8$; thus 1111_2 , in excess-7 is $+8_{10}$.

2.4 Signed Integer Representation

- Lets compare our representations:

Decimal	Binary (for absolute value)	Signed Magnitude	One's Complement
2	00000010	00000010	00000010
-2	00000010	10000010	11111101
100	01100100	01100100	01100100
-100	01100100	11100100	10011011

Decimal	Binary (for absolute value)	Two's Complement	Excess-127
2	00000010	00000010	10000001
-2	00000010	11111110	01111101
100	01100100	01100100	11100011
-100	01100100	10011100	00011011

2.4 Signed Integer Representation

- When we use any finite number of bits to represent a number, we always run the risk of the result of our calculations becoming too large or too small to be stored in the computer.
- While we can't always prevent overflow, we can always *detect* overflow.
- In complement arithmetic, an overflow condition is easy to detect.

2.4 Signed Integer Representation

- Example:
 - Using two's complement binary arithmetic, find the sum of 107 and 46.
- We see that the nonzero carry from the seventh bit *overflows* into the sign bit, giving us the erroneous result: $107 + 46 = -103$.

$$\begin{array}{r} \text{1} \text{ } 1 \quad 1 \text{ } 1 \text{ } 1 \\ 01101011 \\ + 00101110 \\ \hline 10011001 \end{array}$$

But overflow into the sign bit does not always mean that we have an error.

2.4 Signed Integer Representation

- Example:
 - Using two's complement binary arithmetic, find the sum of 23 and -9.
 - We see that there is carry into the sign bit and carry out. The final result is correct: $23 + (-9) = 14$.

$$\begin{array}{r} \textcircled{1} \leftarrow \textcircled{1} 1 1 \quad 1 1 1 \\ \quad 0 0 0 1 0 1 1 1 \\ + \quad 1 1 1 1 0 1 1 1 \\ \hline 0 0 0 0 1 1 1 0 \end{array}$$

Rule for detecting signed two's complement overflow: When the “carry in” and the “carry out” of the sign bit differ, overflow has occurred. If the carry into the sign bit equals the carry out of the sign bit, no overflow has occurred.

2.4 Signed Integer Representation

- Signed and unsigned numbers are both useful.
 - For example, memory addresses are always unsigned.
- Using the same number of bits, unsigned integers can express twice as many “positive” values as signed numbers.
- Trouble arises if an unsigned value “wraps around.”
 - In four bits: $1111 + 1 = 0000$.
- Good programmers stay alert for this kind of problem.

2.4 Signed Integer Representation

- Research into finding better arithmetic algorithms has continued for over 50 years.
- One of the many interesting products of this work is Booth's algorithm.
- In most cases, Booth's algorithm carries out multiplication faster and more accurately than pencil-and-paper methods.
- The general idea is to replace arithmetic operations with bit shifting to the extent possible.

2.4 Signed Integer Representation

In Booth's algorithm:

- If the current multiplier bit is 1 and the preceding bit was 0, subtract the multiplicand from the product
- If the current multiplier bit is 0 and the preceding bit was 1, we add the multiplicand to the product
- If we have a 00 or 11 pair, we simply shift.
- Assume a mythical "0" starting bit
- Shift after each step

$$\begin{array}{r}
 0011 \quad (\text{multiplicand}) \\
 \times 0110 \quad (\text{multiplier}) \\
 \hline
 + 0000 \quad (\text{shift}) \\
 - 0011 \quad (\text{subtract}) \\
 + 0000 \quad (\text{shift}) \\
 + 0011 \quad (\text{add}) \\
 \hline
 00010010
 \end{array}$$

We see that $3 \times 6 = 18$!

2.4 Signed Integer Representation

- Here is a larger example.

Ignore all bits over $2n$.

```

      00110101
    x 01111110
    -----
+ 000000000000000000
+ 1111111111001011
+ 0000000000000000
+ 0000000000000000
+ 0000000000000000
+ 00000000000000
+ 000000000000
+ 000110101
    -----
10001101000010110

```

$53 \times 126 = 6678!$

2.4 Signed Integer Representation

- Overflow and carry are tricky ideas.
- Signed number overflow means nothing in the context of unsigned numbers, which set a carry flag instead of an overflow flag.
- If a carry out of the leftmost bit occurs with an unsigned number, overflow has occurred.
- Carry and overflow occur independently of each other.

The table on the next slide summarizes these ideas.

2.4 Signed Integer Representation

Expression	Result	Carry?	Overflow?	Correct Result?
0100 + 0010	0110	No	No	Yes
0100 + 0110	1010	No	Yes	No
1100 + 1110	1010	Yes	No	Yes
1100 + 1010	0110	Yes	Yes	No

2.4 Signed Integer Representation

- We can do binary multiplication and division by 2 very easily using an *arithmetic shift* operation
- A *left arithmetic shift* inserts a 0 in for the rightmost bit and shifts everything else left one bit; in effect, it multiplies by 2
- A *right arithmetic shift* shifts everything one bit to the right, but copies the sign bit; it divides by 2
- Let's look at some examples.

2.4 Signed Integer Representation

Example:

Multiply the value 11 (expressed using 8-bit signed two's complement representation) by 2.

We start with the binary value for 11:

00001011 (+11)

We shift left one place, resulting in:

00010110 (+22)

The sign bit has not changed, so the value is valid.

To multiply 11 by 4, we simply perform a left shift twice.

2.4 Signed Integer Representation

Example:

Divide the value 12 (expressed using 8-bit signed two's complement representation) by 2.

We start with the binary value for 12:

00001100 (+12)

We shift left one place, resulting in:

00000110 (+6)

(Remember, we carry the sign bit to the left as we shift.)

To divide 12 by 4, we right shift twice.