THE ESSENTIALS OF

Computer Organization and Architecture

THIRD EDITION

Linda Null
Julia Lobur

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:

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

2.2 Positional Numbering Systems

- The binary number 11001 in powers of 2 is:

$$1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

$$= 16 + 8 + 0 + 0 + 1 = 25$$

- 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

- 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

- **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:

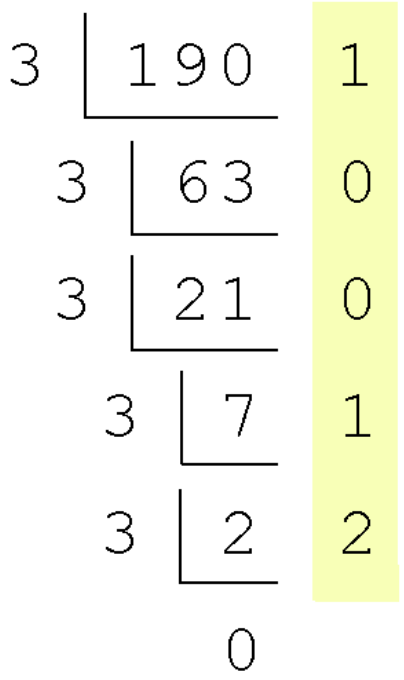
    $190_{10} = 21001_3$

190		
− 162	$= 3^4 \times$	2
28		
− 27	$= 3^3 \times$	1
1		
− 0	$= 3^2 \times$	0
1		
− 0	$= 3^1 \times$	0
1		
− 1	$= 3^0 \times$	1
0		

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:

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



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:

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

$$0.11_2 = 1 \times 2^{-1} + 1 \times 2^{-2}$$

$$= \frac{1}{2} + \frac{1}{4}$$

$$= 0.5 + 0.25 = 0.75$$

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:

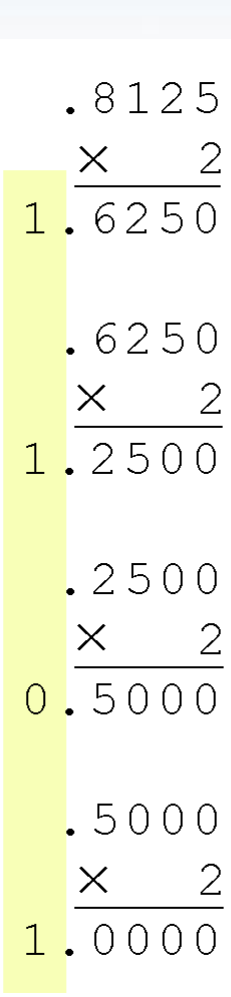
    $0.8125_{10} = 0.1101_2$
  - Of course, this method works with any base, not just binary.

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

2.3 Converting Between Bases

- **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.



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:

0011	0101	0001	1011
3	5	1	B

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 ($8 = 2^3$):

011	010	100	011	011
3	2	4	3	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.

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      0 + 1 = 1
1 + 0 = 1      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.

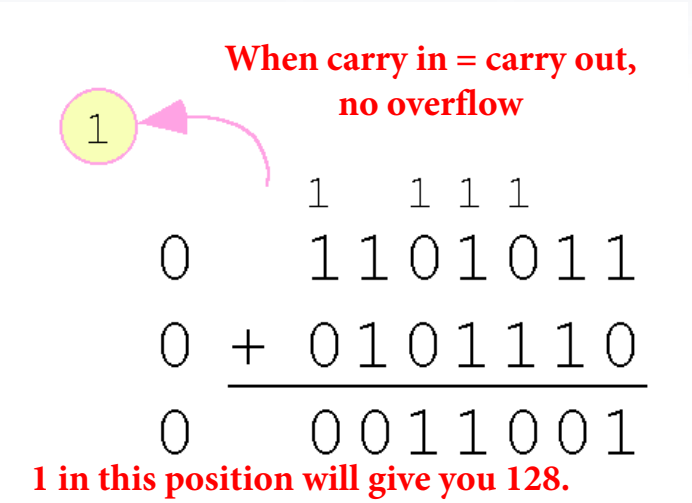
```
      1 1 1
0   1001011
0 + 0101110
-----------
0   1111001
```

- Once we have worked our way through all eight bits, we are done.

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.



2.4 Signed Integer Representation

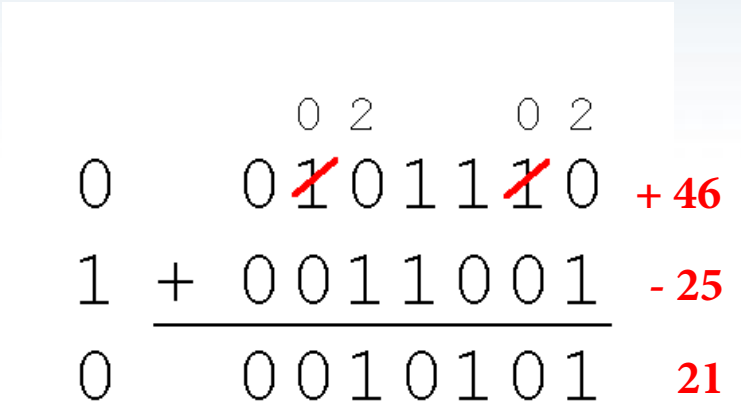
- The signs in signed magnitude representation work just like the signs in pencil and paper arithmetic.
  - Example: Using signed magnitude binary arithmetic, find the sum of - 46 and - 25.

```
    1 1
1   0101110   - 46
1 + 0011001   - 25
------------
1   1000111   - 71
```

- 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.



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

2.4 Signed Integer Representation

- For example, using 8-bit **one's complement** representation:

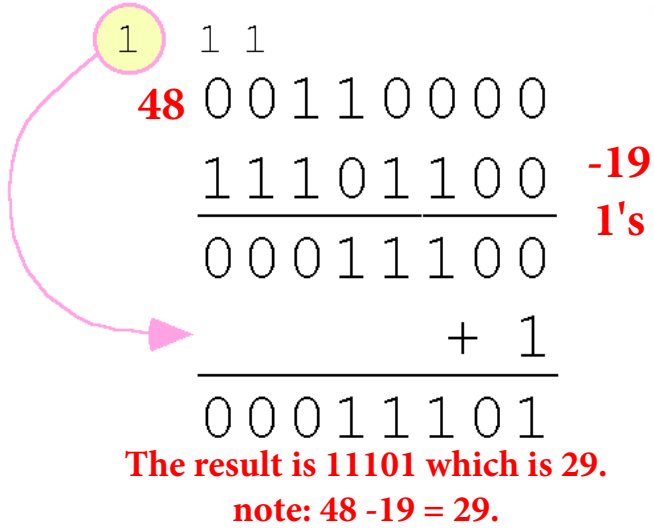
  + 3 is: `00000011`

  - 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



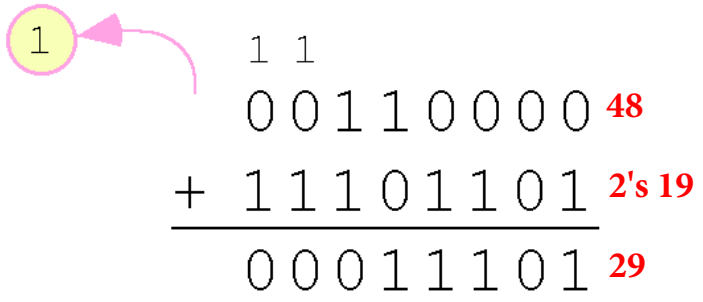
We note that 19 in binary is 00010011,
so -19 in one's complement is: 11101100.

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 two's complement binary arithmetic, find the sum of 48 and - 19.



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

2.4 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.

So when the numbers have the same sign bit, do not use any complement process. Just add them and put the sign in the sign bit.

```
  1 1   1 1 1
  01101011   107
+ 00101110   46
-----------
  10011001
```

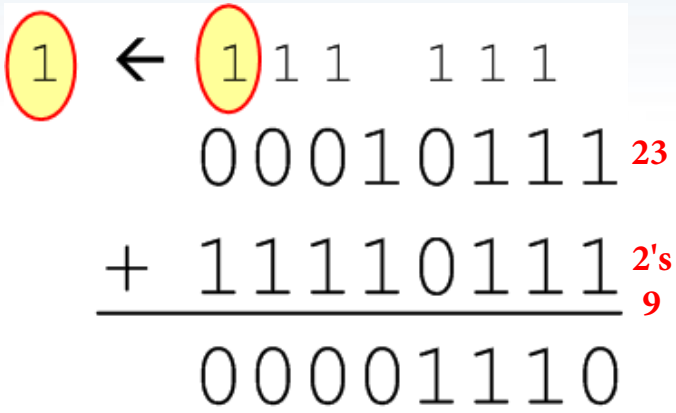
Output is 153

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.

Carry in and carry out are the same, so no overflow has occurred.



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

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

```
      0011
    x 0110
    ------
    + 0000   (shift)
    - 0011   (subtract)
    + 0000   (shift)
    + 0011   (add)
    ------
    00010010
```

We see that 3 × 6 = 18!

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 right 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.

2.5 Floating-Point Representation

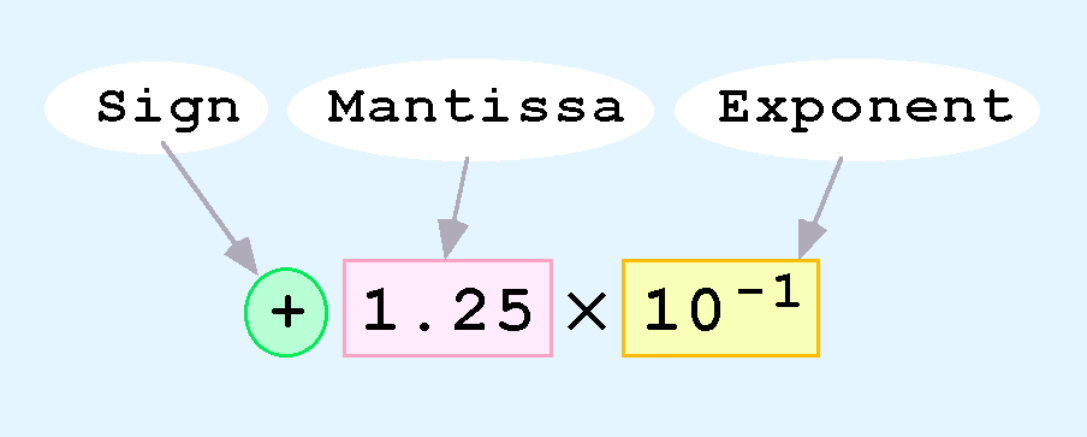
- Floating-point numbers allow an arbitrary number of decimal places to the right of the decimal point.
  - For example: $0.5 \times 0.25 = 0.125$
- They are often expressed in scientific notation.
  - For example:

    $0.125 = 1.25 \times 10^{-1}$

    $5{,}000{,}000 = 5.0 \times 10^{6}$

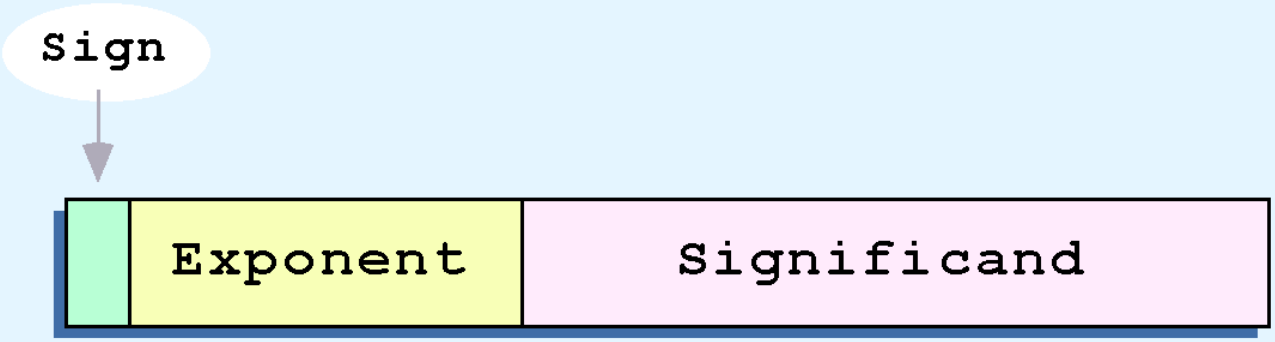
2.5 Floating-Point Representation

- Computers use a form of scientific notation for floating-point representation
- Numbers written in scientific notation have three components:



2.5 Floating-Point Representation

- Computer representation of a floating-point number consists of three fixed-size fields:

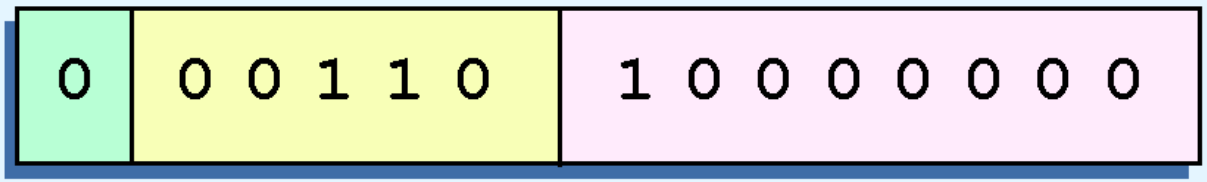


- This is the standard arrangement of these fields.

Note: Although "significand" and "mantissa" do not technically mean the same thing, many people use these terms interchangeably. We use the term "significand" to refer to the fractional part of a floating point number.

2.5 Floating-Point Representation

- Example:
  - Express 32_{10} in the simplified 14-bit floating-point model.
- We know that 32 is 2^5. So in (binary) scientific notation $32 = 1.0 \times 2^5 = 0.1 \times 2^6$.
  - In a moment, we'll explain why we prefer the second notation versus the first.
- Using this information, we put 110 (= 6_{10}) in the exponent field and 1 in the significand as shown.



2.5 Floating-Point Representation

- To resolve the problem of synonymous forms, we establish a rule that the first digit of the significand must be 1, with no ones to the left of the radix point.
- This process, called *normalization*, results in a unique pattern for each floating-point number.
  - In our simple model, all significands must have the form 0.1xxxxxxxx
  - For example, $4.5 = 100.1 \times 2^0 = 1.001 \times 2^2 = 0.1001 \times 2^3$. The last expression is correctly normalized.

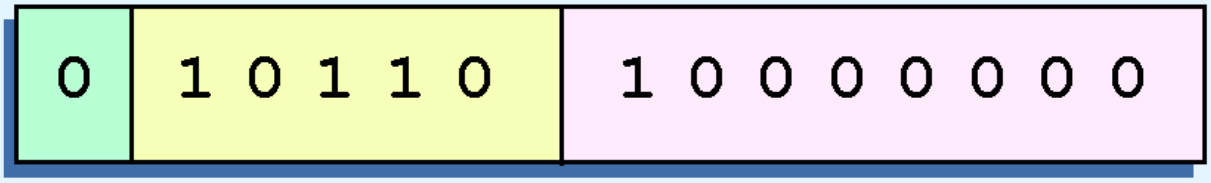
In our simple instructional model, we use no implied bits.

2.5 Floating-Point Representation

- To provide for negative exponents, we will use a *biased exponent*.
- A bias is a number that is approximately midway in the range of values expressible by the exponent. We subtract the bias from the value in the exponent to determine its true value.
  - In our case, we have a 5-bit exponent. **We will use 16 for our bias. This is called *excess-16* representation.**
- In our model, exponent values less than 16 are negative, representing fractional numbers.

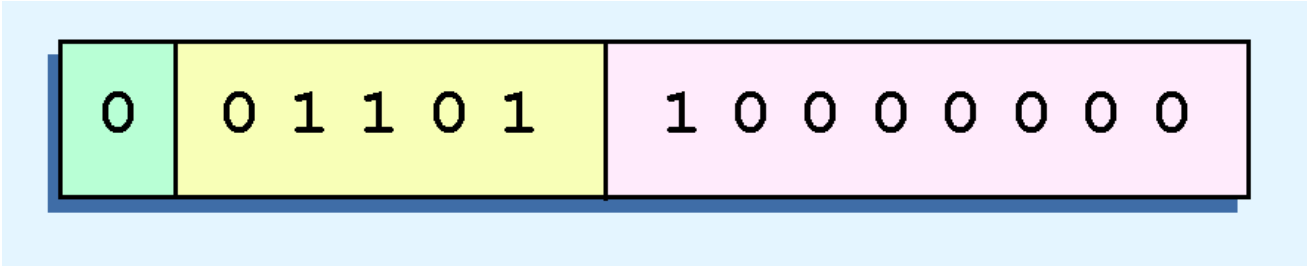
2.5 Floating-Point Representation

- Example:
  - Express 32_{10} in the revised 14-bit floating-point model.
- We know that $32 = 1.0 \times 2^5 = 0.1 \times 2^6$.
- To use our excess 16 biased exponent, we add 16 to 6, giving 22_{10} (=10110_2).
- So we have:



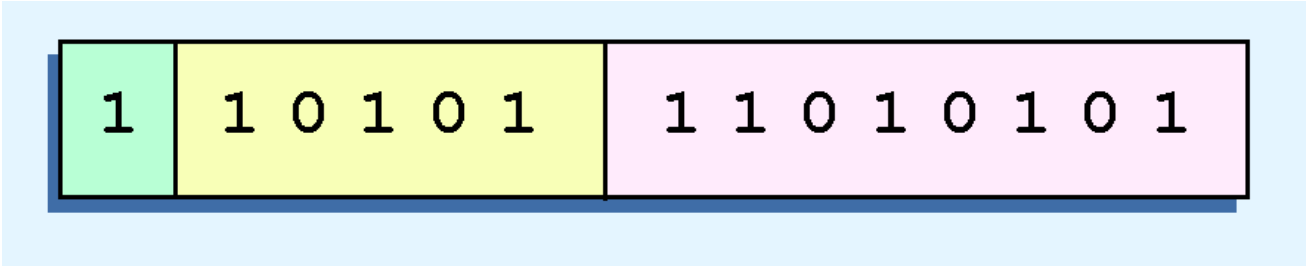
2.5 Floating-Point Representation

- Example:
  - Express 0.0625_{10} in the revised 14-bit floating-point model.
- We know that 0.0625 is 2^{-4}. So in (binary) scientific notation $0.0625 = 1.0 \times 2^{-4} = 0.1 \times 2^{-3}$.
- To use our excess 16 biased exponent, we add 16 to -3, giving 13_{10} (=01101_2).



2.5 Floating-Point Representation

- Example:
  - Express -26.625_{10} in the revised 14-bit floating-point model.
- We find $26.625_{10} = 11010.101_2$. Normalizing, we have: $26.625_{10} = 0.11010101 \times 2^5$.
- To use our excess 16 biased exponent, we add 16 to 5, giving 21_{10} (=10101_2). We also need a 1 in the sign bit.

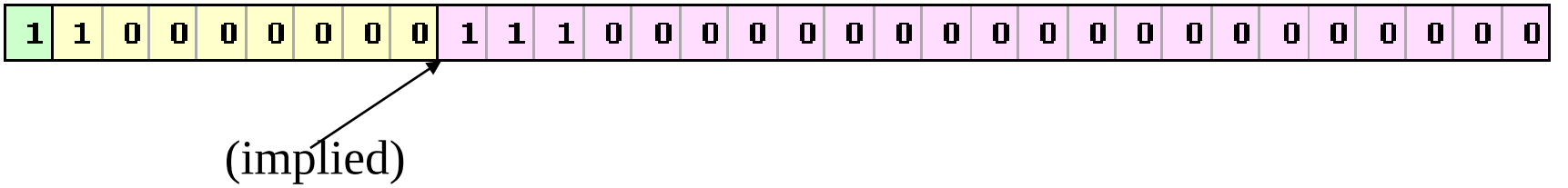


2.5 Floating-Point Representation

- The IEEE has established a standard for floating-point numbers
- The IEEE-754 *single precision* floating point standard uses an 8-bit exponent (with a bias of 127) and a 23-bit significand.
- The IEEE-754 *double precision* standard uses an 11-bit exponent (with a bias of 1023) and a 52-bit significand.

2.5 Floating-Point Representation

- Example: Express -3.75 as a floating point number using IEEE single precision.
- First, let's normalize according to IEEE rules:
  - 3.75 = -11.11_2 = -1.111×2^1
  - The bias is 127, so we add 127 + 1 = 128 (this is our exponent)



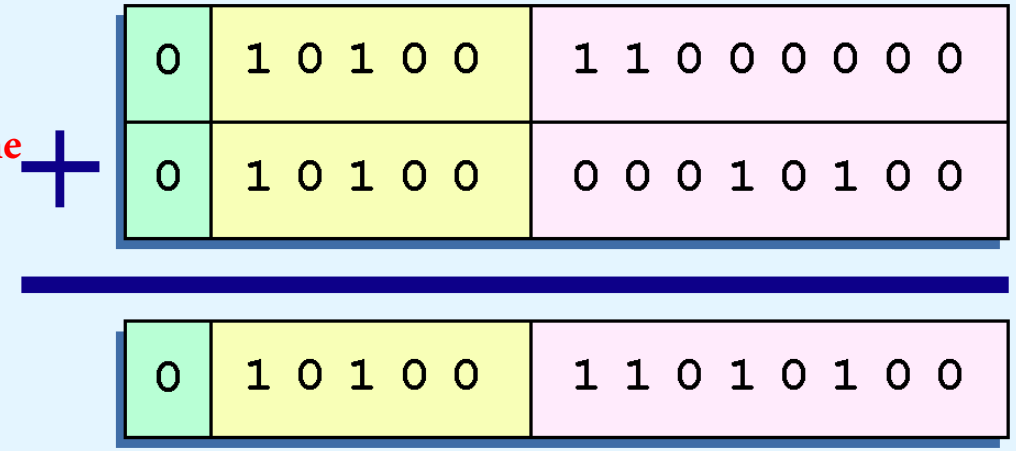
  - Since we have an implied 1 in the significand, this equates to

    $-(1).111_2 \times 2^{(128-127)} = -1.111_2 \times 2^1 = -11.11_2 = -3.75.$

2.5 Floating-Point Representation

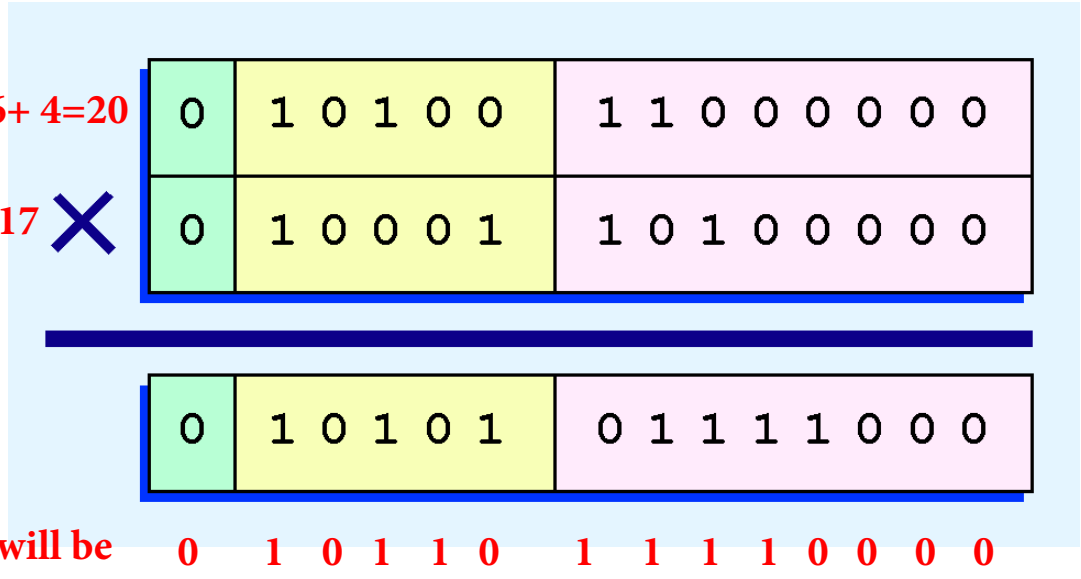
- Example:
  - Find the sum of 12_{10} and 1.25_{10} using the 14-bit "simple" floating-point model. with a bias of 16.
- We find $12_{10} = 0.1100 \times 2^4$. And $1.25_{10} = 0.101 \times 2^1 = 0.000101 \times 2^4$.
- Thus, our sum is 0.110101×2^4.

Bias is 16 + 4 which is 20 so the exponet for both will be 20



2.5 Floating-Point Representation

- Example:
  - Find the product of 12_{10} and 1.25_{10} using the 14-bit floating-point model. with bias 16.
- We find $12_{10} = 0.1100 \times 2^{4}$. And $1.25_{10} = 0.101 \times 2^{1}$.
- Thus, our product is $0.0111100 \times 2^{5} = 0.1111 \times 2^{4}$.
- The **normalized** product requires an exponent of $22_{10} = 10110_{2}$.



2.8 Error Detection and Correction

- Modulo 2 arithmetic works like clock arithmetic.
- In clock arithmetic, if we add 2 hours to 11:00, we get 1:00.
- In modulo 2 arithmetic if we add 1 to 1, we get 0. The addition rules couldn't be simpler:

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

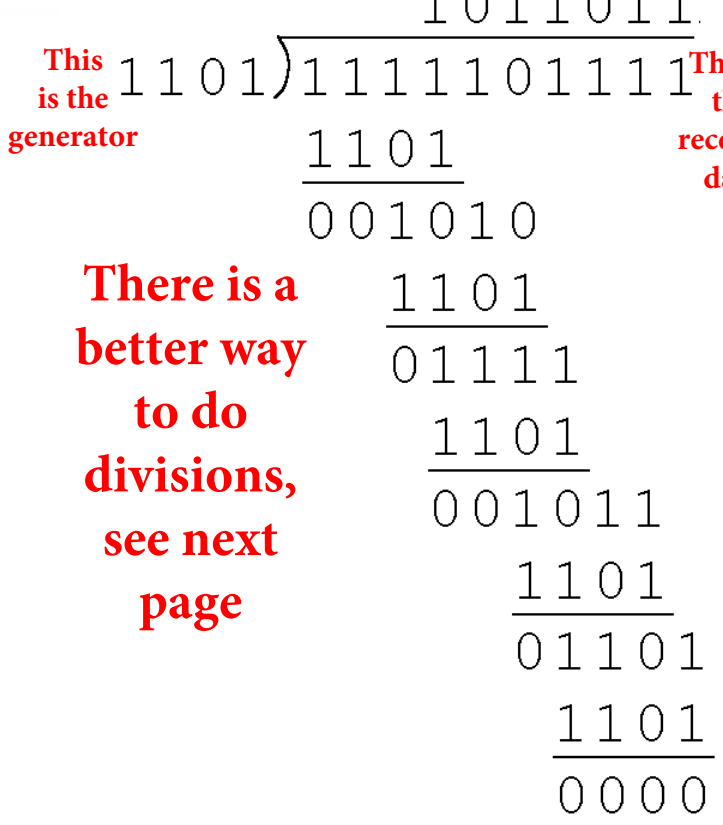
You will fully understand why modulo 2 arithmetic is so handy after you study digital circuits in Chapter 3.

2.8 Error Detection and Correction

CRC

If there is a remainder in the final division then the received bit pattern is in error, therefore the receiver will request a retransmission of the original pattern from the transmitter

- If no bits are lost or corrupted, dividing the received information string by the agreed upon pattern will give a remainder of zero.
- We see this is so in the calculation at the right.
- Real applications use longer polynomials to cover larger information strings.
  - Some of the standard poly-nomials are listed in the text.



There is a better way to do divisions, see next page

We have

$$111110\,1111 = X^9 + X^8 + X^7 + X^6 + X^5 + \underbrace{X^4}_{\text{off}} + X^3 + X^2 + X + 1$$

and

$$1101 = X^3 + X^2 + \underbrace{X}_{\text{ff}} + 1$$

Therefore

$$\begin{array}{r|l}
X^3 + X^2 + 1 & X^6 + X^4 + X^3 + X + 1 \\
\hline
X^9 + X^8 + X^7 + X^6 + X^5 + X^3 + X^2 + X + 1 & \\
X^9 + X^8 + X^6 & \\
\hline
X^7 + X^5 + X^3 + X^2 + X + 1 & \\
X^7 + X^6 + X^4 & \\
\hline
X^6 + X^5 + X^4 + X^3 + X^2 + X + 1 & \\
X^6 + X^5 + X^3 & \\
\hline
X^4 + X^2 + X + 1 & \\
X^4 + X^3 + X & \\
\hline
X^3 + X^2 + 1 & \\
X^3 + X^2 + 1 & \\
\hline
0\quad 0\quad 0 &
\end{array}$$

where:

$$X^6 + X^4 + X^3 + X + 1$$

is

101101

Ans.

Using Polynomials is Better

Hamming Codes

Error Detection and Correction

The telephone system has 3 parts; the switches, the interoffice trunks, and the local loops. The first two are becoming digital in most countries. The local loops are still analogue twisted copper pairs everywhere. Errors are rare in the digital part but are common in the analogue part. Also wireless comm. has errors too.

Errors are either come in BURSTS OR ISOLATED SINGLE-BIT ERRORS, with the former being much harder to detect and correct, see chapter 7 Later.

A. Error Correcting codes: These codes include enough redundant info. along each block of data sent to enable the receiver to deduce what the transmitted character must have been. One good method is called the HAMMING CODE for single error correction. This technique is quite common for memory addressing and transferring bits from registers to RAM and Back.

An (11,7) Hamming code example will be given, where 11 is the total number of transmitted codeword bits and 7 is the number of message bits.

At the encoder (Transmitter), the 7 bits will be transformed into 11 bit after adding 4 parity bits.

Example: Given the message 1010011 find the codeword using Hamming codes.

Solution: The parity bits will be inserted in the following positions (1, 2, 4, 8, 16, etc) i.e. power of 2's ($2^0, 2^1, 2^2, 2^3, 2^4$, etc).

∴ The code word should be

11	10	9	8	7	6	5	4	3	2	1
1	0	1	P_4	0	0	1	P_3	1	P_2	P_1

Now, Positions 11, 9, 5, and 3 are all having a binary 1 and are choosen to be added together using XOR (modulo-2 arithmetic) which is

$0 \oplus 0 = 1 \oplus 1 = 0$, $1 \oplus 0 = 0 \oplus 1 = 1$

$\therefore$ 11 = 1011
9 = 1001
5 = 0101
3 = 0011
} (modulo-2 or XOR) Encoding at the Transmitter

0100 $\therefore P_1 = 0$, $P_2 = 0$, $P_3 = 1$, $P_4 = 0$

$\therefore$ The transmitted codeworde is

1 0 1 0 0 0 1 1 0 0 ← The Answer.
11 9 5 4 3

Now, at the Rx, it will XOR all positions with a binary 1

11 = 1 0 1 1
9 = 1 0 0 1
5 = 0 1 0 1
4 = 0 1 0 0
3 = 0 0 1 1
} (XOR)

0 0 0 0

Hence, the Rx codeword is correct.

Now, Suppose there is an error in one of the message bits, say bit 5

then the Rx codeword would have been

1 0 1 0 0 0 0 1 1 0 0
↑

then, also XOR only positions with bit 1

$$\left.\begin{array}{l} 11 = 1\ 0\ 11 \\ 9 = 1\ 0\ 0\ 1 \\ 4 = 0\ 1\ 0\ 0 \\ 3 = 0\ 0\ 1\ 1 \end{array}\right\} \text{XOR}$$

Decoding at the Receiver.

$0\ 1\ 0\ 1$ ← Non-Zero ∴ There is ERROR.

ALSO, 0101 ⇒ is equivalent to decimal 5.

∴ change this bit in position 5 from old state (0) to new state (1).

Hamming codes are a single-bit error correcting codes. It also can detect two-bit errors BUT other mutiple-bit errors cannot be detected.

The number of Parity bits can be estimated using the following equation.

If m = Number of message bits

r = Number of Parity bits

$$\therefore \quad (m + r + 1) \leq 2^r$$

and in our example $m = 7$

$$\therefore \quad 7 + 1 \leq 2^r - r$$

$$\text{or} \quad 2^3 \leq 2^r - r$$

let $r = 4$

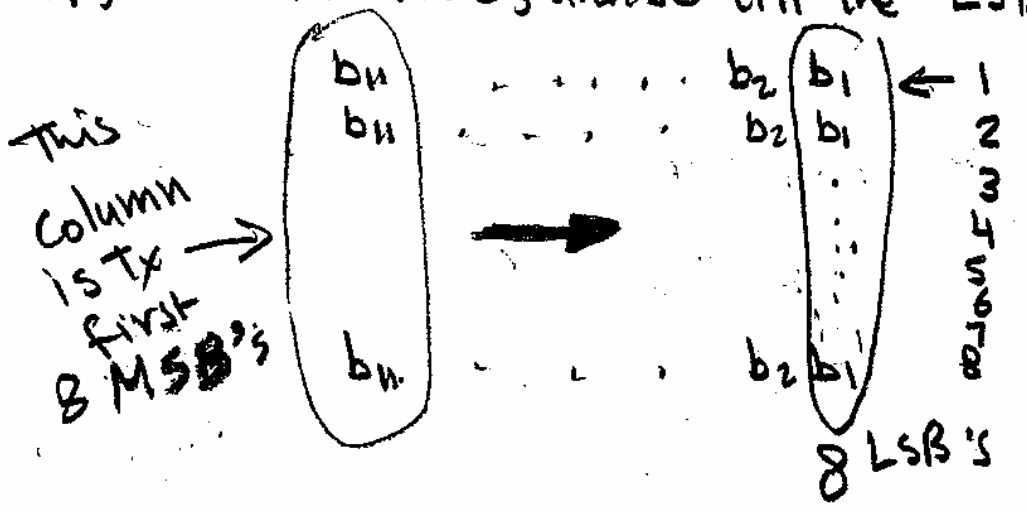
$$\therefore \quad 8 \leq 2^4 - 4 = 12$$

$$\therefore \quad 8 < 12 \quad \therefore \text{ choose } r = 4.$$

Since $r = 3$ gives $2^3 - 3 = 5$

and $8 \not\leq 5$.

As mentioned before, the main types of error occurring in many data Comm. N.W.s are error bursts rather than isolated single or double-bit errors.
The Hamming coding scheme may be extended by a simple technique to deal with burst errors. Now, if for example we are required to transmit a block of data, comprising a string of, say, eight ASCII characters, over a simplex channel that has a high Probability of an error burst occuring. The apprach in such a case would be for the controlling device to first convert each ASCII character into 11-bit codeword form, to give a block of eight 11-bit codewords. Then, instead of Tx. each codeword separately, the controlling device would transmit the contents of the block of codewords a column at a time. Thus the eight, say, MSB's would be Tx. first, then next MSB's, and so till the LSB's.



The controlling device at the Rx. then performs the reverse operation, reassembling the Tx. block in a memory, prior to performing the detection and if necessary, correction operation on each codeword.