3. Computer Arithmetic and Representation of Data

(a) Unsigned Integers.
(b) Operations on Integers, Signed Integers.
(c) Other Data Structures.
(a) Unsigned Integers

Representation in ENIAC: One digit represented by 10 vacuum tubes, ie. by 10 bits.
(Bit = binary digit = digit 0 or 1)

Better approach: With 4 bits we can represent $2^4 = 16$ objects.
Represent each digit by 4 bits.

Appropriate for financial data (commercial rounding): Binary coded decimal (BCD):
135 coded as \[0001\underbrace{0011}_{1}\underbrace{0101}_{5}.\]

Otherwise: Waste of space (6 of 16 possibilities not used); further difficult to compute by a circuit.

Instead: Binary representation.
Decimal representation of numbers:

4053 stands for
\[ 4 \times 10^3 + 0 \times 10^2 + 5 \times 10^1 + 3 \times 10^0. \]

Other bases: Replace 10 by other numbers:
\[ (4053)_7 = 4 \times 7^3 + 5 \times 7^1 + 3 = \]
\[ 4 \times 343 + 5 \times 7 + 3 = 1410. \]

Binary representation means: basis is 2:
\[ (101101)_2 = 2^5 + 2^3 + 2^2 + 2^0 = 32 + 8 + 4 + 1 = 45. \]

Long binary numbers are difficult to read.

Hexadecimal representation better: Basis 16.
Lack of digits: One writes \( A, B, C, D, E, F \) for 10, 11, 12, 13, 14, 15.
Example \( (AF03)_{16} = 10 \times 16^3 + 15 \times 16^2 + 3 \)
Notation: $0xAF03$ for $AF03_{16}$.
$0b110110$ for $(110110)_2$.
In binary representation with fixed length, the bit corresponding to the highest power of 2 is called the most significant bit, MSB. In standard Western writing it is the bit most to the left, and on most machines (big-endian ones) it is the bit with the lowest number. On some machines (little-endian machines) this bit will be stored as the one with the highest number.
The bit corresponding to the lowest power of two is called the least significant bit, LSB.
Example:

```
0b 10101010
  ↑       ↑
MSB     LSB
bit No. 0 bit No. 7
on little-endian on little-endian
machines     machines
```
Plural of MSB and LSB

We use as well the plural of MSB and LSB:

- By **the three MSBs** (**three most significant bits**) the three bits with the heighest weight
- **LSBs** is used similarly.

Octal representation

Sometimes **octal representation** of numbers is used as well: this means that basis 8 is used. In this representation only **digits 0, \ldots, 7** occur.
Conversion from Binary to Hexadecimal

Trivial transformation:
A binary number is transformed into hexadecimal, by packing from the right four bits into one hexadecimal number:

Example: 0b1001110111111
= 0b \underbrace{1}_{1} \underbrace{0011}_{3} \underbrace{1011}_{B} \underbrace{1111}_{F}
= 0x13BF.

Conversion from Hexadecimal to Binary

Write every digit as a binary number with four bits.

Example: 0x13BF
= 0b1\underbrace{0001}_{1}3\underbrace{0011}_{B}\underbrace{1011}_{F}\underbrace{1111}_{F}
= 0b0001001110111111.
Leading 0’s can be omitted.
Conversion from Decimal to Binary

For small numbers practical method:
Find highest power of 2 you can subtract.
Continue with the rest, till you arrive at 0.
Form a binary number with 1 at the positions corresponding to the powers used:
Example: 15321

\[
\begin{array}{ccc|c|c}
 & & & \text{Position} \\
15321 - 2^{13} & = & 15321 - 8192 & = & 7129 & 13 \\
7129 - 2^{12} & = & 7129 - 4096 & = & 3033 & 12 \\
3033 - 2^{11} & = & 3033 - 2048 & = & 985 & 11 \\
985 - 2^9 & = & 985 - 512 & = & 473 & 9 \\
473 - 2^8 & = & 473 - 256 & = & 217 & 8 \\
217 - 2^7 & = & 217 - 128 & = & 89 & 7 \\
89 - 2^6 & = & 89 - 64 & = & 25 & 6 \\
25 - 2^4 & = & 25 - 16 & = & 9 & 4 \\
9 - 2^3 & = & 9 - 8 & = & 1 & 3 \\
1 - 2^0 & = & 1 - 1 & = & 0 & 0 \\
\end{array}
\]

If we describe positions by subscripts we have

\[
15321 = \\
0b \quad 1 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 0 \\
\uparrow \quad \uparrow \quad \uparrow \quad \uparrow \quad \uparrow \quad \uparrow \quad \uparrow \quad \uparrow \quad \uparrow \quad \uparrow \quad \uparrow \quad \uparrow \quad \uparrow \quad \uparrow \quad \uparrow \quad \uparrow \\
13 \quad 12 \quad 11 \quad 10 \quad 9 \quad 8 \quad 7 \quad 6 \quad 5 \quad 4 \quad 3 \quad 2 \quad 1 \quad 0 \\
= 0x3BD9
\]
Better algorithm:
If we divide a decimal number by 10 with remainder, we obtain the number shifted to the right by 1, and as remainder the least significant digit:

<table>
<thead>
<tr>
<th>Result</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>153 ÷ 10</td>
<td>15 3</td>
</tr>
<tr>
<td>15 ÷ 10</td>
<td>1 5</td>
</tr>
<tr>
<td>1 ÷ 10</td>
<td>0 1</td>
</tr>
</tbody>
</table>

The remainder are the digits of the decimal representation.
Do the same to convert to binary representation:
<table>
<thead>
<tr>
<th>Result</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>7660</td>
<td>1</td>
</tr>
<tr>
<td>3830</td>
<td>0</td>
</tr>
<tr>
<td>1915</td>
<td>0</td>
</tr>
<tr>
<td>957</td>
<td>1</td>
</tr>
<tr>
<td>478</td>
<td>1</td>
</tr>
<tr>
<td>239</td>
<td>0</td>
</tr>
<tr>
<td>119</td>
<td>1</td>
</tr>
<tr>
<td>59</td>
<td>1</td>
</tr>
<tr>
<td>29</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

$15321 = 0b1110111011001$.

**Conversion from decimal to hexadecimal:**
Probably easiest by hand: convert to binary, then to hexadecimal.
Otherwise: similar to the above method.
(b) Operations on Integers, Signed Integers

Addition of Unsigned Numbers

Addition of Decimals:

```
  1298
+  751
  Carry 11
  2049
```

Note: Carry is at most 1:
If we add two decimal digits, ie. two numbers in the interval 0...9 we get at most 18.
It might be that we have to add carry one, then we obtain at most 19.

Addition of Binary Numbers: Do just the same:

```
  101101
+  111110
  Carry 1111
  1101011
```
Half One-Bit-Adder

We have the following table for addition

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Result</th>
<th>Carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

So we get

\[
\text{Result} = (A \lor B) \land \neg (A \land B) \ . \\
\text{Carry} = A \land B
\]

Implementation:
Full One-Bit-Adder

If we have a carry $C$ from the previous addition, we obtain the following table:

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$C$</th>
<th>Result</th>
<th>Carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

If we write $\overline{A}$ for $\neg A$ similarly $\overline{B}$, $\overline{C}$, we obtain

\[
\begin{align*}
\text{Result} & = (\overline{A} \land \overline{B} \land C) \\
              & \lor (\overline{A} \land B \land \overline{C}) \\
              & \lor (A \land \overline{B} \land \overline{C}) \\
              & \lor (A \land B \land C)
\end{align*}
\]

\[
\begin{align*}
\text{Carry} & = (A \land B) \lor (A \land C) \lor (B \land C)
\end{align*}
\]
Implementation

A | B | C

Sum

Carry
Implementation using two half one-bit-adders:

**Advantage:** Fewer gates needed.

**Disadvantage:** Delay, because the signal has to pass through more gates.
A Four Bit Adder

Possible carry from previous addition

1st bit input

2nd bit input

3rd bit input

4th bit input

Possible carry from previous addition

A B C

A B C

A B C

A B C

A B C

A B C

A B C

A B C

A B C

A B C

A B C

A B C

A B C

A B C

A B C

A B C

A B C

A B C

A B C

A B C

A B C

A B C

A B C

A B C

A B C

A B C

A B C

A B C

A B C

A B C

A B C

A B C

A B C

A B C
**Problem:** The carry-input for the $n$th bit has to pass through $n - 1$ half-adders, which causes delay.

**Solution:** Carry Lookahead: Compute it directly from the digits (passes through less gates - note that the $n$th carry is a function from the least significant $n$ bits of the addends). Becomes complicated with increasing $n$, several techniques for optimization.
Overflow

If we add two $k$ bit numbers, we might obtain a $k + 1$ bit number:
With $k$ bits we can represent numbers in the range $0 \ldots 2^k - 1$.
The sum of two such numbers is in the range $0 \ldots 2^{k+1} - 2$.
An adder usually has as result the $k$ least significant bits of the calculation, and additional one bit called overflow flag, representing the most significant bit.
Other operations have as well overflow or underflow (result too small to be represented).
Subtraction

In order to compute $a - b$, do the following:
Write $a$ and $b$ as $n$-bit binary numbers.
Take bitwise complement of $b$. Add one to this. Add $a$ to it as an $n$-bit number.

There will be a carry unless there is an underflow (the result is negative). Ignore this carry. (Especially don’t write the result as an $n+1$-bit number).

**Example:** Calculation of $7 - 6$:

\[
\begin{align*}
7 &= 0b0111. \\
6 &= 0b0110. \\
\text{Bitwise complement of 6} &= 0b1001. \\
1 \text{ added to bitwise complement } +7 &= 0b1010. \\
\text{Result of subtraction} &= 0b0001(\text{+Carry})
\end{align*}
\]
Remark 1: Note that in the above example, the addition of $0b1010$ and $0b0111$, without limiting the result of the addition to 4 bits plus carry would be $0b10001$, which is not the result of the subtraction. Instead the carry (the leading 1) has to be ignored.

Remark 2: In general carry in the resulting addition means: overflow of the subtraction. No carry in the addition means: underflow of the subtraction (negative value).

Remark 3 (for the mathematically interested): The above works as well in the special case $b = 0$. Then the bitwise complement of $b$ is a binary number consisting of $n$ ones, if we add 1 we obtain 0 plus carry, addition of $a$ has as result $a$ plus carry.
Why is the Algorithm for Subtraction Correct?

Bitwise complement of a binary number with \( k \) bits is the same as subtracting from a binary number with \( k \) ones the number:

\[
11111111 - 01101100 = 10010011
\]

The binary number, consisting of \( k \) ones is \( 2^k - 1 \).

The above calculation has as result

\[
a + ((2^k - 1) - b + 1) = 2^k + (a - b)
\]

If \( a \geq b \) an adder has as result \( a - b \) plus an overflow bit.

If \( a < b \), we get a number < \( 2^k \), no overflow, and \( a - b \) is negative, the subtraction has underflow.

Alternative Hardware Calculation of Subtraction: Compute it directly similar to an adder.
Multiplication

\[ 1101 \times 1011 = \]

\[ \begin{array}{c}
  1101 \\
  + 1101 \\
  + 1101 \\
  \hline
  10001111
\end{array} \]

The result of the multiplication of two \( k \) bit numbers requires up to \( 2k \) bits!

Multiplication can be achieved by:
Start with sum 0.
Shifting successively multiplicand to the left and multiplier to the right (\( k \) times if we have \( k \) bits).
Whenever the LSB of multiplier is 1, add multiplicand to the sum.
Result is sum.
First Algorithm for Multiplication of $m$ and $m'$

sum := 0
for $i = 1$ to $k$ do
    begin
        if LSB$(m) = 1$ then sum := sum + $m'$
        Shift $m$ right one bit
        Shift $m'$ left one bit
    end
sum is result
### Example (First Algorithm)

<table>
<thead>
<tr>
<th></th>
<th>$m$</th>
<th>$m'$</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initialization</strong></td>
<td>011</td>
<td>000 101</td>
<td>000 000</td>
</tr>
<tr>
<td><strong>Step 1a</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{LSB}(m) = 1$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add $m'$</td>
<td>011</td>
<td>000 101</td>
<td>000 101</td>
</tr>
<tr>
<td><strong>Step 1b</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shifts</td>
<td>001</td>
<td>001 010</td>
<td>000 101</td>
</tr>
<tr>
<td><strong>Step 2a</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{LSB}(m) = 1$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add $m'$</td>
<td>001</td>
<td>001 010</td>
<td>001 111</td>
</tr>
<tr>
<td><strong>Step 2b</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shifts</td>
<td>000</td>
<td>010 100</td>
<td>001 111</td>
</tr>
<tr>
<td><strong>Step 3a</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{LSB}(m) = 0$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No action</td>
<td>000</td>
<td>010 100</td>
<td>001 111</td>
</tr>
<tr>
<td><strong>Step 3b</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shifts</td>
<td>000</td>
<td>101 000</td>
<td>001 111</td>
</tr>
</tbody>
</table>

$0b011 \times 0b101 = 3 \times 5 = 15 = 0b1111$. 
Problem: We need a $2k$ bit adder for adding two $k$ bit numbers. However the number we add has always $k$ zeros (in a block left and right of the original multiplier).

Optimized version: Add the multiplier to the left most $k$ bits of the sum. Shift this sum then to the right.

Difficulty: We will however get one overflow bit.
Modification needed: We have to use $2k + 1$ instead of $2k$ bits, and add the multiplier to the left most $k + 1$ bits of the sum.
Second Algorithm for the Multiplication of $m$ and $m'$

sum is a $2k + 1$ bit number.
sum := 0
for $i = 1$ to $k$ do
    begin
    if LSB$(m) = 1$ then add $m'$ to the $k+1$ most significant bits of sum
    Shift $m$ right one bit
    Shift sum right one bit
    end
sum is result
Example:

<table>
<thead>
<tr>
<th>Step</th>
<th>$m$</th>
<th>$m'$</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>011</td>
<td>101</td>
<td>0 000</td>
</tr>
<tr>
<td>Step 1a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSB($m$) =1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add $m'$</td>
<td>011</td>
<td>101</td>
<td>0 101</td>
</tr>
<tr>
<td>Step 1b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shifts</td>
<td>001</td>
<td>101</td>
<td>0 010 1</td>
</tr>
<tr>
<td>Step 2a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSB($m$) =1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add $m'$</td>
<td>001</td>
<td>101</td>
<td>0 111 1</td>
</tr>
<tr>
<td>Step 2b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shifts</td>
<td>000</td>
<td>101</td>
<td>0 011 11</td>
</tr>
<tr>
<td>Step 3a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSB($m$) =0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No action</td>
<td>000</td>
<td>101</td>
<td>0 011 11</td>
</tr>
<tr>
<td>Step 3b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shifts</td>
<td>000</td>
<td>101</td>
<td>0 001 111</td>
</tr>
</tbody>
</table>

$0b011 \times 0b101 = 3 \times 5 = 15 = 0b1111$. 

70
Observe that in the last example at the end of Step $l$, the $4 + l$ MSBs of sum (written in italic, in colour blue) are the same as the $4 + l$ LSBs of sum in the first algorithm.
Example, where one extra bit is needed:

<table>
<thead>
<tr>
<th></th>
<th>$m$</th>
<th>$m'$</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>111</td>
<td>111</td>
<td>0 000</td>
</tr>
<tr>
<td>Step 1a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSB($m$) =1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add $m'$</td>
<td>111</td>
<td>111</td>
<td>0 111</td>
</tr>
<tr>
<td>Step 1b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shifts</td>
<td>011</td>
<td>111</td>
<td>0 011</td>
</tr>
<tr>
<td>Step 2a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSB($m$) =1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add $m'$</td>
<td>011</td>
<td>111</td>
<td>1 010</td>
</tr>
<tr>
<td>Step 2b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shifts</td>
<td>001</td>
<td>111</td>
<td>0 101</td>
</tr>
<tr>
<td>Step 3a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSB($m$) =1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add $m'$</td>
<td>001</td>
<td>111</td>
<td>1 100</td>
</tr>
<tr>
<td>Step 3b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shifts</td>
<td>000</td>
<td>111</td>
<td>0 110</td>
</tr>
</tbody>
</table>

$0b111 \times 0b111 = 7 \times 7 = 49 = 0b110001.$
Third Algorithm for the Multiplication of $m$ and $m'$

We can store $m$ in least significant bits of the sum. Then we have only to shift sum to the right.

sum is a $2k + 1$ bit number.
sum := 0
Set the $k$ LSBs of sum to $m$
for $i = 1$ to $k$ do
  begin
    if LSB(sum) = 1 then add $m'$ to the $k+1$ MSBs of sum
    Shift sum right one bit
  end
sum is result
Example:

<table>
<thead>
<tr>
<th></th>
<th>( m )</th>
<th>( m' )</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>011</td>
<td>101</td>
<td>0 000</td>
</tr>
<tr>
<td>Step 1a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSB(sum) =1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add ( m' )</td>
<td>011</td>
<td>101</td>
<td>0 101</td>
</tr>
<tr>
<td>Step 1b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shift</td>
<td>011</td>
<td>101</td>
<td>0 010</td>
</tr>
<tr>
<td>Step 2a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSB(sum) =1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add ( m' )</td>
<td>011</td>
<td>101</td>
<td>0 111</td>
</tr>
<tr>
<td>Step 2b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shift</td>
<td>011</td>
<td>101</td>
<td>0 011</td>
</tr>
<tr>
<td>Step 3a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSB(sum) =0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No action</td>
<td>011</td>
<td>101</td>
<td>0 011</td>
</tr>
<tr>
<td>Step 3b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shift</td>
<td>011</td>
<td>101</td>
<td>0 001</td>
</tr>
</tbody>
</table>

\( 0b011 \times 0b101 = 3 \times 5 = 15 = 0b1111. \)
(In the last example, the bar $|$ separates the bits identical with the *sum* in the previous algorithm (written in *italic*, in colour *blue*) from $m'$ (written in *boldface*, in colour *green*) which are both shifted to the right simultaneously.)
Fourth Multiplication Algorithm: Booth’s Algorithm (Unsigned Numbers)

(A version of this algorithm for signed number will be presented later).

Use that a block of 1s can be obtained from subtracting from 1 followed by the same number of 0s 1:
0b1111 = 0b10000 − 0b1.

Therefore, subtract $m'$ from the $k$ MSBs of sum, if $m$ (or in the optimized version sum) has a 1 with 0 or blank to the right (beginning of a block of 1s).
Add $m'$ to the MSBs of sum, if $m$ (or in the optimized version sum) has a 0 with a 1 to the right (end of a block of 1s).
We need to save the LSB of sum after a shift. Let it be sum$_{-1}$.
**Note** that if the MSB of $m'$ is 1 then we have to subtract $2^{k+1} \cdot m$ from sum, i.e. the algorithm has to carry out the operation $k + 1$ times.

**Problem:** Negative results. If we subtract and obtain a negative number, this should be treated as if sum had infinitely many ones to the left:

$$0b000 - 0b101 = 0b \cdots 111111101$$

In this example, it suffices to keep $k + 1$ bits of the difference, all other bits to the left will be the same as the MSB of this $k + 1$-bit number. When shifting this number to right, we need to replace the MSB of the new number by the MSB of the old number. Arithmetic shift by one bit to the right means: shift a number to the right, replace the MSB of the new number by the MSB of the old number.

In the algorithm, we need to treat sum as a $2k + 2$-bit number.
Booth’s Algorithm for the Multiplication of $m$ and $m'$ (as Unsigned Numbers)

sum has in the following $2k + 2$ digits.

sum := 0
Set the $k$ LSBs of sum to $m$
sum_{-1} := 0
for $i = 1$ to $k + 1$ do
    begin
        if LSB(sum) = 1 and sum_{-1} = 0 then
            subtract $m'$ as a $k + 1$-bit number
            from the $k + 1$ MSBs of sum
            and ignore a borrow
        if LSB(sum) = 0 and sum_{-1} = 1 then
            add $m'$ as a $k + 1$-bit number
            to the $k + 1$ MSBs of sum
            and ignore a carry
        sum_{-1} := LSB(sum)
        Arithmetical shift right of sum by one bit
    end
sum is result
<table>
<thead>
<tr>
<th></th>
<th>( m )</th>
<th>( m' )</th>
<th>sum</th>
<th>sum_1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initialization</strong></td>
<td>011</td>
<td>101</td>
<td><strong>0000</strong></td>
<td><strong>0011</strong></td>
</tr>
<tr>
<td><strong>Step 1a</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSB(sum) =1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sum_1=0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subtract ( m' )</td>
<td>011</td>
<td>101</td>
<td><strong>1011</strong></td>
<td><strong>0011</strong></td>
</tr>
<tr>
<td><strong>Step 1b</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arithmetic shift</td>
<td>011</td>
<td>101</td>
<td><strong>1101 11</strong></td>
<td><strong>001</strong></td>
</tr>
<tr>
<td><strong>Step 2a</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSB(sum) =</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sum_1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No action</td>
<td>011</td>
<td>101</td>
<td><strong>1101 11</strong></td>
<td><strong>001</strong></td>
</tr>
<tr>
<td><strong>Step 2b</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arithmetic shift</td>
<td>011</td>
<td>101</td>
<td><strong>1110 11</strong></td>
<td><strong>00</strong></td>
</tr>
<tr>
<td><strong>Step 3a</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSB(sum) =0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sum_1=1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add ( m' )</td>
<td>011</td>
<td>101</td>
<td><strong>0011 11</strong></td>
<td><strong>00</strong></td>
</tr>
<tr>
<td><strong>Step 3b</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arithmetic shift</td>
<td>011</td>
<td>101</td>
<td><strong>0001 111</strong></td>
<td><strong>0</strong></td>
</tr>
<tr>
<td><strong>Step 4a</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSB(sum) =</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=sum_1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No action</td>
<td>011</td>
<td>101</td>
<td><strong>0001 111</strong></td>
<td><strong>0</strong></td>
</tr>
<tr>
<td><strong>Step 3b</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arithmetic shift</td>
<td>011</td>
<td>101</td>
<td><strong>0000 1111</strong></td>
<td><strong>0</strong></td>
</tr>
</tbody>
</table>
Note the following points in the last example:

- Subtracting \( m' \) has to be done as 4 bit numbers.
  If we carry out the bitwise complement of \( m' \) (as 4 bit numbers) and add one, we obtain 1011. Subtraction of \( m' \) as 4 bit number is the same as addition of 1011.
  (If we directly carry out the subtraction, one obtains a borrow (the equivalent of a carry in case of subtraction). This borrow can be ignored.)

- When adding \( m' \), one usually obtains a carry, i.e. an overflow bit. This bit can be again ignored.
The above can be explained as follows: If we subtracted \( m' \) by hand without ignoring the borrow, we obtained infinitely many ones to the left; in the example above in Step 1a we obtain 
\[ \cdots 11111110110011. \]

We regard now for the sum written with finitely many digits:

- 1 as MSB as an indication that there are **infinitely many ones** to the left,
- 0 as MSB as an indication that there are **infinitely many zeros** to the left.

When carrying out an arithmetic shift to the right, this corresponds in our representation to an ordinary shift of a number with infinitely many bits to the right.

When we add now \( m' \) later (in step 3a), and are working with infinitely many digits, all the leading 1s are swallowed.
On the next slide we look at the same example, if we treated sum as a number having infinitely many digits.
On slide 76e, another (ordinary) example of Booth’s algorithm is carried out. There subtraction of \( m' \) is the same as addition of 1101.
<table>
<thead>
<tr>
<th>Step</th>
<th>sum</th>
<th>sum_1</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>... 000 0000</td>
<td>0011</td>
<td>0</td>
</tr>
<tr>
<td>Step 1a</td>
<td>LSB(sum) = 1</td>
<td>sum_1 = 0</td>
<td>Subtract m'</td>
</tr>
<tr>
<td>Step 1b</td>
<td>Shift</td>
<td>... 111 1101</td>
<td>1</td>
</tr>
<tr>
<td>Step 2a</td>
<td>LSB(sum) = sum_1</td>
<td>No action</td>
<td>... 111 1101</td>
</tr>
<tr>
<td>Step 2b</td>
<td>Shift</td>
<td>... 111 1110</td>
<td>11</td>
</tr>
<tr>
<td>Step 3a</td>
<td>LSB(sum) = 0</td>
<td>sum_1 = 1</td>
<td>add m'</td>
</tr>
<tr>
<td>Step 3b</td>
<td>Shift</td>
<td>... 000 0001</td>
<td>111</td>
</tr>
<tr>
<td>Step 4a</td>
<td>LSB(sum) = sum_1</td>
<td>No action</td>
<td>... 000 0001</td>
</tr>
<tr>
<td>Step 3b</td>
<td>Shift</td>
<td>... 000 0000</td>
<td>1111</td>
</tr>
</tbody>
</table>

76d
<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
<th>(m)</th>
<th>(m')</th>
<th>sum</th>
<th>(\text{sum}_1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>(\text{sum}_1=0)</td>
<td>111</td>
<td>011</td>
<td>0000</td>
<td>0111</td>
</tr>
<tr>
<td>Step 1a</td>
<td>LSB(sum) = 1</td>
<td>111</td>
<td>011</td>
<td>1101</td>
<td>0111</td>
</tr>
<tr>
<td></td>
<td>subtract (m')</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 1b</td>
<td>Arithmetic shift</td>
<td>111</td>
<td>011</td>
<td>11101</td>
<td>011</td>
</tr>
<tr>
<td>Step 2a</td>
<td>LSB(sum) = (\text{sum}_1)</td>
<td>111</td>
<td>011</td>
<td>11101</td>
<td>011</td>
</tr>
<tr>
<td></td>
<td>No action</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 2b</td>
<td>Arithmetic shift</td>
<td>111</td>
<td>011</td>
<td>111101</td>
<td>0</td>
</tr>
<tr>
<td>Step 3a</td>
<td>LSB(sum) = (\text{sum}_1)</td>
<td>111</td>
<td>011</td>
<td>111101</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>No action</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 3b</td>
<td>Arithmetic shift</td>
<td>111</td>
<td>011</td>
<td>1111101</td>
<td>0</td>
</tr>
<tr>
<td>Step 4a</td>
<td>LSB(sum) = 0 (\text{sum}_1=1)</td>
<td>111</td>
<td>011</td>
<td>0010101</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Add (m')</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 3b</td>
<td>Arithmetic shift</td>
<td>111</td>
<td>011</td>
<td>00010101</td>
<td>0</td>
</tr>
</tbody>
</table>
(b) Signed Integers

**Oldest Method:** One more bit for sign: The MSB is the sign and the remaining bits form the value.
If the MSB is 0, the number is positive, if it is 1, the number is just the negative value of the number with MSB 0.
Eg. 0011 represents 3, 1011 represents -3.

**Disadvantage:** Two representations of 0 (±0, −0).
Arithmetic complicated. Especially in case of addition one needs 4 different algorithms depending on the sign of the first and second argument.
Second method (mainly used)

Twos complement representation:
If we represent numbers by k bits, the MSB gets value $-2^{k-1}$ instead of $2^{k-1}$.

Example:
With $k = 8$ bits ($2^{k-1} = 128$),
01101010 represents $2 + 8 + 32 + 64 = 106$.
11101010 represents $2 + 8 + 32 + 64 - 128 = -22$.

Note that 11101010 does not represent $-106$!!!.
Observe further, that this interpretation is highly dependent on the number of bits one uses:
11101010 represents $-22$ as a 8 bit number, but 011101010 represents $+234$.
The correct expansion of the signed number 11101010 to 9 bits is obtained by adding one more 1 in front of it: it is 111101010.
**Easy algorithm** for computing binary representation of negative numbers:
In order to represent $-n$, compute the binary representation of $n$.

**Compute the bit-wise complement, add one**

**Example** with 8 bits:
$22 = 0b00010110$
Complement is $11101001$.
Add one $11101010$. 
Why is this Algorithm Correct?

Use again: bitwise complement of a number is the same as subtracting it from a binary number consisting of \( k \) ones, i.e. from \( 2^k - 1 \). So our algorithm yields for \( -n \) as result

\[
((2^k - 1) - n) + 1 = 2^k - n.
\]

If we take \( 2^k - n \) and decode it as a two’s complement, we have to interpret the MSB as \( -2^{k-1} \) instead of \( 2^{k-1} \). So we have to subtract twice \( 2^{k-1} \) for it.

The result is \( (2^k - n) - 2^{k-1} - 2^{k-1} = -n \).

So the number obtained in the algorithm represents \( -n \).
## Arithmetical Operations for Signed Numbers

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Addition</strong></td>
<td>Use algorithm for addition as for unsigned numbers. Ignore overflow from unsigned addition. Overflow of signed addition iff both addends have same sign, and the result a different sign.</td>
</tr>
<tr>
<td><strong>Subtraction</strong></td>
<td>Add negative of the subtrahend number</td>
</tr>
<tr>
<td><strong>Multiplication</strong></td>
<td>Use Booth’s algorithm as for unsigned numbers. However omit $k + 1$th step and treat sum as a $2k$-ary number.</td>
</tr>
</tbody>
</table>
Booth’s Algorithm for the Multiplication of Signed $k$-bit Numbers $m$ and $m'$

sum has in the following $2k$ digits.

sum := 0  
Set the $k$ LSBs of sum to $m$  
sum$_{-1}$ := 0  
for $i = 1$ to $k$ do  
begin  
  if LSB(sum) = 1 and sum$_{-1}$ = 0 then  
    subtract $m'$ as a $k$-bit signed number  
    from the $k$ MSBs of sum  
    and ignore any borrow  
  if LSB(sum) = 0 and sum$_{-1}$ = 1 then  
    add $m'$ as a $k$-bit signed number  
    to the $k$ MSBs of sum  
    and ignore any carry  
  sum$_{-1}$ := LSB(sum)  
  Arithmetical shift right of sum by one bit  
end  
sum is result
### Example for Multiplication of Signed Numbers with Booth’s algorithm

<table>
<thead>
<tr>
<th></th>
<th>( m )</th>
<th>( m' )</th>
<th>sum</th>
<th>sum(_{-1})</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initialization</strong></td>
<td>101</td>
<td>110</td>
<td>000</td>
<td>101</td>
</tr>
<tr>
<td><strong>Step 1a</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{LSB}(\text{sum}) = 1 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{sum}_{-1} = 0 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subtract ( m' )</td>
<td>101</td>
<td>110</td>
<td>010</td>
<td>101</td>
</tr>
<tr>
<td><strong>Step 1b</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{Arithmetic shift} )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>101</td>
<td>110</td>
<td>001</td>
<td>010</td>
</tr>
<tr>
<td><strong>Step 2a</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{LSB}(\text{sum}) = 0 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{sum}_{-1} = 1 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add ( m' )</td>
<td>101</td>
<td>110</td>
<td>111</td>
<td>010</td>
</tr>
<tr>
<td><strong>Step 2b</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{Arithmetic shift} )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>101</td>
<td>110</td>
<td>111</td>
<td>101</td>
</tr>
<tr>
<td><strong>Step 3a</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{LSB}(\text{sum}) = 1 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{sum}_{-1} = 0 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subtract ( m' )</td>
<td>101</td>
<td>110</td>
<td>001</td>
<td>101</td>
</tr>
<tr>
<td><strong>Step 3b</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{Arithmetic shift} )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>101</td>
<td>110</td>
<td>000</td>
<td>110</td>
</tr>
</tbody>
</table>
101 represents 

\[-4 + 1 = -3.\]

110 represents 

\[-4 + 2 = -2.\]

\((-3) \times (-2) = 6\) is represented by 000110 (with 6 bits!)

Note that subtraction of 110 means addition of 010.
(c) Other Data structures

Fixed Point Numbers

In decimal representation, real numbers can be written as numbers with potentially infinite digits after the point:

\[ \pi = 3.1415926 \ldots \]

**Analysis:** All digits get a weight in terms of powers of 10. Left to the point, this is a positive power of 10, right of it a negative one:

\[
\sqrt{111} = \begin{array}{ccccccc}
1 & 0 & . & 5 & 3 & 5 & 5 \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
10^1 & 10^0 & 10^{-1} & 10^{-2} & 10^{-3} & 10^{-4} \\
= & = & = & = & = & = \\
10 & 1 & \frac{1}{10} & \frac{1}{100} & \frac{1}{1000} & \frac{1}{10000} \\
\end{array}
\]

\[
= 1 \times 10^1 + 0 \times 10^0 + 5 \times 10^{-1} \\
+ 3 \times 10^{-2} + 5 \times 10^{-3} + 5 \times 10^{-4} + \ldots
\]

**Fixed point numbers:** Round or cut off after a fixed number of positions after the point.
**Binary fixed point numbers:** With basis 2 we have a similar representation:

\[
\begin{array}{cccccc}
0b & 1 & 0 & . & 1 & 0 & 1 & 0 \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
2^1 & 2^0 & 2^{-1} & 2^{-2} & 2^{-3} & 2^{-4} \\
= & = & = & = & = & = \\
2 & 1 & \frac{1}{2} & \frac{1}{4} & \frac{1}{8} & \frac{1}{16} \\
\end{array}
\]

\[
= 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} \\
+ 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + \ldots
\]

**Conversion from decimal to binary:**
Convert the **whole-numbered** part (the part before the point) into a binary number.
Convert the **fractional part** (the part after the point) into the fractional part of a binary number as follows:
Multiply the number by 2
If the result is $\geq 1$, the next digit is 1, subtract 1 from the result and continue.
If the result is $< 1$, the next digit is 0, continue.
Example

We write 10.5355 as a fixed point binary number.
(This is not $\sqrt{111}$, since we have rounded this number already).
$10 = 0b1010$.

<table>
<thead>
<tr>
<th>Fraction</th>
<th>Digit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5355</td>
<td></td>
</tr>
<tr>
<td>x 2 = 1.0710</td>
<td>1</td>
</tr>
<tr>
<td>–1 = 0.0710</td>
<td></td>
</tr>
<tr>
<td>x 2 = 0.1420</td>
<td>0</td>
</tr>
<tr>
<td>x 2 = 0.2840</td>
<td>0</td>
</tr>
<tr>
<td>x 2 = 0.5680</td>
<td>0</td>
</tr>
<tr>
<td>x 2 = 1.1360</td>
<td>1</td>
</tr>
<tr>
<td>–1 = 0.1360</td>
<td></td>
</tr>
<tr>
<td>x 2 = 0.2320</td>
<td>0</td>
</tr>
</tbody>
</table>

$10.5350 = 0b1010.100010 \ldots$
Remark: The full (not-fixed-point) binary representation of the decimal number $0.2$ is $0.00110011001100110011\ldots$.

So numbers might have exact representation as a decimal fixed point number, but not as a binary fixed point number.

Consequence: Rounding errors occur. A calculation with decimal fixed-point representation and with binary fixed-point representation might have different results.

Important when working with financial data.
Signed fixed point numbers

As fixed point numbers, the MSB gets weight $-2^k$, if $k + 1$ bits to the left of the point. Conversion: Convert whole-numbered part as a signed binary number and the fractional part as before:

Example ($k = 3$):  
1001.01 represents $-8 + 1 + 0.25 = -6.75$.  
0001.01 represents $1 + 0.25 = 1.25$. 
Operations on (Signed) Fixed Point Numbers

**General Method:** Write the operands as fixed point numbers with the same number of bits to the left and to the right of the point. Perform the operation as for (signed) integers. Now reintroduce the point in the result:

- **Case Addition and subtraction:**
  If it initially was after the \( n \)th bit from the right, it should be in the result after the \( n \)th bit from the right.

- **Case Multiplication:**
  If it initially was after the \( n \)th bit from the right, it should be in the result after the \( 2n \)-th bit from the right.
At the end one might omit
- leading zeros (only in case of unsigned fixed point numbers)
  - eg. convert 010.01 into 10.01;
- zeros as least significand bits after the point
  - eg. convert 0.010 into 0.01.
Examples (all unsigned and in binary)

Fixed Point Numbers  
1.01 + 0.01  
= 1.10

Corresponding Integers  
101 + 001  
= 110

Example with overflow in the addition

10.01 + 1.11  
= 10.01 + 01.11  
= 100.00  
= 100

1001 + 0111  
= 10000

10 - 1.11  
= 10.00 - 01.11  
= 00.01  
= 0.01

1000 - 0111  
= 0001

1.00 - 0.01  
= 0.11

100 - 001  
= 011

1.01 x 1.11  
= 10.0011

101 x 111  
= 100011.
Floating Point Numbers

In decimal representation: Numbers represented like 
0.23578 \times 10^k, \ -0.99998 \times 10^{-k}, \ 0.0 \times 10^0. In general we obtain the form \ s \times a \times 10^k, \ where

- \ s = +1 \ or \ s = -1; \ s \ is \ called \ the \ sign.
- \ a \ is \ a \ fixed \ point \ number \ (in \ decimal \ representation) \ s.t. \ 0.1 \leq a < 1 \ or \ a = 0; \ a \ is \ called \ significand \ or \ mantissa;
- \ k \ is \ a \ signed \ integer \ called \ exponent.

Note that

0.349 \times 10^8 = \ \begin{array}{c} 3\ 4900000 \ \uparrow \ 10^7 \end{array}

So the exponent is the \textbf{10 times the weight} of the digit in decimal representation most to the left that is not equal to 0, and the significand is the sequence of \textbf{digits} following it.
Binary Floating Point Numbers

Similarly, only basis 2:
We write numbers as \( s \times a \times 2^e \) where \( s \) is the
sign as before, \( \frac{1}{2} \leq a < 1 \) or \( a = 0 \).

(If \( a \) is chosen in this range, we call this a 
\textbf{normalized floating point representation};
there are several non-normalized representations of the same number, eg. \( 0\bar{b}0.1 \times 2^5 \) and
\( 0\bar{b}0.01 \times 2^6 \) and \( 0\bar{b}1 \times 2^4 \) represent the same
number, of which the first one is the normalized representation).
Optimization:

- If $a \neq 0$ then $a$ will always be of the form $0b0.1xyz\cdots$, with $x, y, z \in \{0, 1\}$. So the first 1 has not to be stored, except for $a = 0$.

- Instead of having positive and negative exponents one uses biased one: Assume we have $k$ bits to represent the exponents, which allows to represent positive numbers in the interval $[0, 2^k - 1]$. Fix a bias $b$, i.e. a positive integer, for all representations of floating point numbers. Then we represent an exponent $e$ in the interval $[-b, -b + 2^k - 1]$ by the positive integer $e + b$, (which is then in the interval $[0, 2^k - 1]$). $e + b$ is called the biased exponent.
Example

If $k = 8$, $2^k = 256$. If we fix the bias 126, then we have the following representations:

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Biased Exponent</th>
<th>Binary Representation (with 8 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-100</td>
<td>26</td>
<td>00011010</td>
</tr>
<tr>
<td>0</td>
<td>126</td>
<td>01111110</td>
</tr>
<tr>
<td>10</td>
<td>136</td>
<td>10001000</td>
</tr>
<tr>
<td>-126</td>
<td>0</td>
<td>00000000</td>
</tr>
<tr>
<td>129</td>
<td>255</td>
<td>11111111</td>
</tr>
</tbody>
</table>

We can represent exponents in the range $[-126, 129]$. 
IEEE 754 Standard for Binary Floating-Point Representation

We consider only the single format with 32 bits, there exists as well a double format with 64 bits.

Floating points are written in the form

<table>
<thead>
<tr>
<th>Bit position</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>23 − 30</td>
</tr>
<tr>
<td></td>
<td>0− 22</td>
</tr>
<tr>
<td></td>
<td>sign</td>
</tr>
<tr>
<td></td>
<td>biased exponent</td>
</tr>
<tr>
<td></td>
<td>significand</td>
</tr>
</tbody>
</table>

- Sign 0 means a positive, 1 a negative number (so \( s \in \{0, 1\} \) represents sign \((-1)^s\)).
- The exponent is biased with bias 126, and represented by 8 bits. So we can represent exponents in the range \([-126, 129]\). However exponent 129 will be used for infinity and NaN, see below, giving an effective range \([-126, 128]\) for the exponent.
- If the exponent is > −126, the significand is interpreted as having an additional leading 1, so the effective significand has 24 bits, not 23 bits.
• If the exponent is \(-126\) (so the biased exponent is 0), a leading one is not implied. Especially significand 0 and biased exponent 0 means value 0. Because of the sign bit, two representations of zero exist.
• Exponent of all ones (0b11111111) with significand 0 expresses positive or negative infinity, depending on the sign; used when overflow occurs.
• Exponent of all ones and non-zero significand represents NaN, not a number, for undefined (error, which cannot be associated to positive or negative infinity, for instance 0 divided by 0).
Remarks on the Bias in the 
IEEE Floating Point Standard

In the literature, one often sees the bias 127 instead of 126 as on the last slides. 
The reason for this difference is as follows: 
When working with bias 127, the calculation of the number represented by a sequence of bits is as follows:
Assume that
• the sign is $s$,
• the sequence of bits corresponding to the significand in the represented number is $\tilde{b}$ and
• the biased exponent is $e$
Then the represented number is calculated as

$$s \times 1.\tilde{b} \times 2^{e-127}$$

where $1.\tilde{b}$ is a binary fixed point number. 
Now this formula yields **not** a normalized number, which we defined as numbers of the form

$$+0.1bbb \ldots \times 2^{-e}.$$
In our slides we therefore calculated as follows: the normalized version of the above number is

\[ s \times 0.1\overline{b} \times 2^{e-126} \]

So if we

- take as effective significand the significand extended by 1,
- use a bias of 126
- and treat the result as a normalized number,

we get the above formula and are consistent with our previous development of the topic.
Examples

\[-0.75 \, = \, (-1) \times 0b0.11 \times 2^0 \]
\[= \, (-1) \times 0b0.11 \times 2^{-126+126} \]
\[= \, (-1) \times 0b0.11 \times 2^{-126+0b1111110} \]

is represented by
(first 1 of the significand omitted!)

\[\begin{array}{c}
1 \\
01111110 \\
\hline
10000000000000000000000000000000
\end{array}\]

sign exponent  

significand

\[0.4375 \, = \, 0b0.111 \times 2^{-1} \]
\[= \, 0b0.111 \times 2^{-126+125} \]
\[= \, 0b0.111 \times 2^{-126+0b1111101} \]

is represented by
(first 1 of the significand omitted!)

\[\begin{array}{c}
0 \\
01111101 \\
\hline
11000000000000000000000000000000
\end{array}\]

sign exponent  

significand
\[ 0.5 \times 2^{-126} = 0.1 \times 2^{-126+0} \]

is represented by
(first 1 of the significand not omitted!)

\[
\begin{array}{c}
0 \overbrace{00000000}^{\text{sign}} \overbrace{100000000000000000000000000000}^{\text{exponent}} \overbrace{.}^{\text{significand}}
\end{array}
\]

0 is represented by

\[
\begin{array}{c}
0 \overbrace{00000000}^{\text{sign}} \overbrace{000000000000000000000000000000}^{\text{exponent}} \overbrace{.}^{\text{significand}}
\end{array}
\]

and

\[
\begin{array}{c}
1 \overbrace{00000000}^{\text{sign}} \overbrace{000000000000000000000000000000}^{\text{exponent}} \overbrace{.}^{\text{significand}}
\end{array}
\]

Infinity is represented by

\[
\begin{array}{c}
0 \overbrace{11111111}^{\text{sign}} \overbrace{000000000000000000000000000000}^{\text{exponent}} \overbrace{.}^{\text{significand}}
\end{array}
\]

is one (of many) representation of NaN.
**Arithmetical Operations** on floating point numbers:

**Simplification here:** We don’t treat the specialties with omitting leading ones, infinity, NaN.

**Addition:** Represent both numbers in the form $s \times a \times 2^k$ where $k$ is the maximum of the exponents of both numbers.
Add the new significands. Adapt the representation, s.t. one obtains a binary floating point number in normal form:

**Example**

\[
\begin{align*}
0b0.1011 \times 2^{0b110} &+ 0b0.101 \times 2^{0b011} \\
= 0b0.1011 \times 2^{0b110} &+ 0b0.000101 \times 2^{0b110} \\
= 0b0.110001 \times 2^{0b110}
\end{align*}
\]

\[
\begin{align*}
0b0.10 \times 2^{0b000} &+ 0b0.10 \times 2^{0b000} \\
= 0b1.0 \times 2^{0b000} \\
= 0b0.1 \times 2^{0b001}
\end{align*}
\]
Subtraction: Similar.

Multiplication

\[(s_0 \times a_0 \times 2^{k_0}) \times (s_1 \times a_1 \times 2^{k_1}) = (s_0 \times s_1) \times (a_0 \times a_1) \times 2^{k_0+k_1}\]

Therefore the sign of the result is the negation of the exclusive or of \(s_0\) and \(s_1\).
(Exclusive or of \(a, b\) is \((a \land \neg b) \lor (\neg a \land b)\)).
The significand of the result is the product of the significands.
The exponent is the sum of the exponents (if exponents are biased, the bias has then to be subtracted).
If one does not obtain a number in normal form, adapt the result.

Examples:

\((0b0.1 \times 2^{0b01}) \times (0b0.1 \times 2^{0b01}) = 0b0.01 \times 2^{0b10} = 0b0.1 \times 2^{0b01}\)

\((-1) \times 0b0.11 \times 2^{0b10}) \times ((-1) \times 0b0.11 \times 2^{0b10}) = 0b0.1001 \times 2^{0b100}.\)
**Characters, Texts**

**Standard Representation of Characters:** ASCII, uses 7 bits which represent numbers between 0 and 127 to represent on character. Table:

<table>
<thead>
<tr>
<th>ASCII</th>
<th>Char</th>
<th>ASCII</th>
<th>Char</th>
<th>ASCII</th>
<th>Char</th>
<th>ASCII</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NUL</td>
<td>16</td>
<td>DLE</td>
<td>32</td>
<td>SP</td>
<td>48</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>SOH</td>
<td>17</td>
<td>DC1</td>
<td>33</td>
<td>!</td>
<td>49</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>STX</td>
<td>18</td>
<td>DC2</td>
<td>34</td>
<td>&quot;</td>
<td>50</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>ETX</td>
<td>19</td>
<td>DC3</td>
<td>35</td>
<td>#</td>
<td>51</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>EOT</td>
<td>20</td>
<td>DC4</td>
<td>36</td>
<td>$</td>
<td>52</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>ENQ</td>
<td>21</td>
<td>NAK</td>
<td>37</td>
<td>%</td>
<td>53</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>ACK</td>
<td>22</td>
<td>SYN</td>
<td>38</td>
<td>&amp;</td>
<td>54</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>BEL</td>
<td>23</td>
<td>ETB</td>
<td>39</td>
<td>'</td>
<td>55</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>BS</td>
<td>24</td>
<td>CAN</td>
<td>40</td>
<td>(</td>
<td>56</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>HT</td>
<td>25</td>
<td>EM</td>
<td>41</td>
<td>)</td>
<td>57</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>LF</td>
<td>26</td>
<td>SUB</td>
<td>42</td>
<td>*</td>
<td>58</td>
<td>:</td>
</tr>
<tr>
<td>11</td>
<td>VT</td>
<td>27</td>
<td>ESC</td>
<td>43</td>
<td>+</td>
<td>59</td>
<td>;</td>
</tr>
<tr>
<td>12</td>
<td>FF</td>
<td>28</td>
<td>FS</td>
<td>44</td>
<td>,</td>
<td>60</td>
<td>&lt;</td>
</tr>
<tr>
<td>13</td>
<td>CR</td>
<td>29</td>
<td>GS</td>
<td>45</td>
<td>-</td>
<td>61</td>
<td>=</td>
</tr>
<tr>
<td>14</td>
<td>SO</td>
<td>30</td>
<td>RS</td>
<td>46</td>
<td>.</td>
<td>62</td>
<td>&gt;</td>
</tr>
<tr>
<td>15</td>
<td>SI</td>
<td>31</td>
<td>US</td>
<td>47</td>
<td>/</td>
<td>63</td>
<td>?</td>
</tr>
<tr>
<td>ASCII</td>
<td>Char</td>
<td>ASCII</td>
<td>Char</td>
<td>ASCII</td>
<td>Char</td>
<td>ASCII</td>
<td>Char</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------</td>
<td>------</td>
<td>-------</td>
<td>------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>64</td>
<td>@</td>
<td>80</td>
<td>P</td>
<td>96</td>
<td>'</td>
<td>112</td>
<td>p</td>
</tr>
<tr>
<td>65</td>
<td>A</td>
<td>81</td>
<td>Q</td>
<td>97</td>
<td>a</td>
<td>113</td>
<td>q</td>
</tr>
<tr>
<td>66</td>
<td>B</td>
<td>82</td>
<td>R</td>
<td>98</td>
<td>b</td>
<td>114</td>
<td>r</td>
</tr>
<tr>
<td>67</td>
<td>C</td>
<td>83</td>
<td>S</td>
<td>99</td>
<td>c</td>
<td>115</td>
<td>s</td>
</tr>
<tr>
<td>68</td>
<td>D</td>
<td>84</td>
<td>T</td>
<td>100</td>
<td>d</td>
<td>116</td>
<td>t</td>
</tr>
<tr>
<td>69</td>
<td>E</td>
<td>85</td>
<td>U</td>
<td>101</td>
<td>e</td>
<td>117</td>
<td>u</td>
</tr>
<tr>
<td>70</td>
<td>F</td>
<td>86</td>
<td>V</td>
<td>102</td>
<td>f</td>
<td>118</td>
<td>v</td>
</tr>
<tr>
<td>71</td>
<td>G</td>
<td>87</td>
<td>W</td>
<td>103</td>
<td>g</td>
<td>119</td>
<td>w</td>
</tr>
<tr>
<td>72</td>
<td>H</td>
<td>88</td>
<td>X</td>
<td>104</td>
<td>h</td>
<td>120</td>
<td>x</td>
</tr>
<tr>
<td>73</td>
<td>I</td>
<td>89</td>
<td>Y</td>
<td>105</td>
<td>i</td>
<td>121</td>
<td>y</td>
</tr>
<tr>
<td>74</td>
<td>J</td>
<td>90</td>
<td>Z</td>
<td>106</td>
<td>j</td>
<td>122</td>
<td>z</td>
</tr>
<tr>
<td>75</td>
<td>K</td>
<td>91</td>
<td>[</td>
<td>107</td>
<td>k</td>
<td>123</td>
<td>{</td>
</tr>
<tr>
<td>76</td>
<td>L</td>
<td>92</td>
<td>\</td>
<td>108</td>
<td>l</td>
<td>124</td>
<td></td>
</tr>
<tr>
<td>77</td>
<td>M</td>
<td>93</td>
<td>]</td>
<td>109</td>
<td>m</td>
<td>125</td>
<td>}</td>
</tr>
<tr>
<td>78</td>
<td>N</td>
<td>94</td>
<td>^</td>
<td>110</td>
<td>n</td>
<td>126</td>
<td>~</td>
</tr>
<tr>
<td>79</td>
<td>O</td>
<td>95</td>
<td>_</td>
<td>111</td>
<td>o</td>
<td>127</td>
<td>DEL</td>
</tr>
</tbody>
</table>
IBM has extended ASCII (extended ASCII) by adding 128 characters, especially symbols occurring in other European languages, as well some mathematical symbols. In total therefore 8 bits (≡ 1 byte) are needed to encode them. Characters 0 - 127 are identically with ASCII and 128 - 255 are new. Unfortunately variants of this set of characters exist, partially due to the fact that IBM’s set is not allowed to be used on non-IBM-licensed PCs. Characters 128 - 255 are not standardized. If one prints out non-text-files as ASCII files, control characters result often in unintended print commands, often page breaks. If one edits non-text-files, these control character lead as well to unintended behaviour, for instance ringing of the bell (modern editors avoid this).
Other character encodings

- **Unicode** (allows to represent international characters including Chinese, Japanese and Korean, with 16 bits. Used in Java).
- **EBCDIC** (used by IBM and IBM-compatible mainframe computers).
Text Represented as sequences of characters. Use of end-of-line, TAB, to save symbols.

Advanced formats: Programming languages for storing text with graphics (ps, pdf, TeX, Word). Various compressing algorithms in order to save space.

Other data: Graphics: as bit maps. Main formats for graphics: graphics interchange format, gif), with some compression. Music, video data in various formats.

Main Problem: multi-media data require a lot of space. Good compression algorithms needed. Use of the fact that small changes between consecutive images, sounds.
Some compression algorithms lead to loss of data (lossy algorithms). For instance, omission of invisible colour changes in areas where texture is particularly vivid. **Examples:** JPEG (images), MPEG (video, audio; computationally complex to code and decode), Quicktime (video). MPEG-2 achieves compression rates of more than 100:1.
Storage of Programs

High level program: As text (usually ASCII).
Machine language: Machine specific.
Examples of Machine language commands (MIPS R2000):
(MIPS used by NEC, Nintendo, Silicon Graphics, Sony; typical RISC language).

lb $R1, a1  Load byte at memory location $a1 to register $R1.
add $R1, $R2, $R3  Add contents of registers $R2, $R3 and store result in $R1
beq $R1, $R2, l1  Branch, if contents of register $R1 and $R2 are equal, to location labeled by l1.

(Labels do not occur themselves in machine code, instead the argument will be an appropriate offset).

High level programming languages are translated into such machine code.
Example

for $i = 0$ to 5 do $a := a - i$

is translated into

```
li R1, 0  %Load immediate value 0
          %to register R1
lw R2, a  %Load content of memory
          %location a into register R2
li R3, 6  %Load immediate value 6
          %into register R3
l1 sub R2, R2, R1 %Subtract R1 from R2,
                  %store result in R2
addi R1, R1, 1 %Add immediate value 1
                %to register R1,
                %store result in R1
bne R1, R3, l1 %Branch if R1 is not equal
               %to R3 to label l1
sw R2, a    %Store register R2 at
            %memory location a
```
Encoding of Assembler Programs

**Definition:** *Word* is an architecture-dependent natural unit of organization of memory. Typically the number of bits used to represent a number and the smallest instruction length. Not clearly defined.

**Storage of Machine Instructions:** Encode the instruction code and the arguments (register numbers, addresses) usually into one or several words. The length varies usually. So the first word contains information about how many words are needed.
**Goal:** Principal design of an architecture, which is able to execute such instructions.
Correction

Please draw AND-gates as follows:

and OR-gates as follows: