Integer Data Representation and Manipulation

198:231 Introduction to Computer Organization
Lecture 2

Instructor:
Nicole Hynes
nicole.hynes@rutgers.edu
Data Representation

- All data are represented as sequences of **bits** (0 or 1) but require different sizes.
- Sizes are multiples of 1 **byte** = 8 bits.
- Size of representation (in bytes) depends on machine and compiler:

<table>
<thead>
<tr>
<th>C data type</th>
<th>32-bit machine</th>
<th>64-bit machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short int</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>
Number Systems

- Positional base $b$ number system
  - Base (or radix) $b$ number is written as $\left( d_{n-1} \ldots d_2 d_1 d_0 \right)_b$
  - Each digit $d_i$ is from the set $\{0, 1, \ldots, b-1\}$
  - Decimal value is $\sum_{i=0}^{n-1} d_i \times b^i$
  - Leftmost digit $d_{n-1}$ is referred to as the most significant digit because it carries the largest weight $(b^{n-1})$
  - Similarly digit $d_0$ is the least significant digit
Decimal Numbers

• We’re familiar with decimal or base 10

• Each digit $d_i$ is from the set \{0,1,2,3,4,5,6,7,8,9\}

• Decimal value of $\left(d_{n-1} \ldots d_2 d_1 d_0\right)_{10}$ is $\sum_{i=0}^{n-1} d_i \times 10^i$

• Example:

$3745_{10} = (3\times10^3) + (7\times10^2) + (4\times10^1) + (5\times10^0)$
Binary Numbers

• Base 2
• Each binary digit (or *bit*) is from the set \{0,1\}
• Decimal value of \((d_{n-1} \ldots d_2 d_1 d_0)_2\) is \(\sum_{i=0}^{n-1} d_i \times 2^i\)
• Example:

\[10110_2 = (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (0 \times 2^0)\]

\[= 16 + 0 + 4 + 2 + 0\]

\[= 22_{10}\]
Octal Numbers

• Base 8
• Each octal digit is from the set \{0,1,2,3,4,5,6,7\}
• Decimal value of \( (d_{n-1} \ldots d_2 d_1 d_0)_8 \) is \( \sum_{i=0}^{n-1} d_i \times 8^i \)
• Example:
  \[ 7216_8 = (7 \times 8^3) + (2 \times 8^2) + (1 \times 8^1) + (6 \times 8^0) \]
  \[ = 3584 + 128 + 8 + 6 \]
  \[ = 3726_{10} \]
• Can write octal constants in C by prefixing a zero before the sequence of octal digits. E.g.,
  \text{int } a = 07216;
Hexadecimal Numbers

• Base 16
• Each hex digit can take on 16 different values: {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}
• First 10 symbols {0,..,9} same as in decimal; A=10, B=11, C=12, D=13, E=14, F=15

• Decimal value of \((d_{n-1} \ldots d_2 d_1 d_0)_{16}\) is \(\sum_{i=0}^{n-1} d_i \times 16^i\)

• Example:
\[
2F3A_{16} = (2 \times 16^3) + (15 \times 16^2) + (3 \times 16^1) + (10 \times 16^0)
= 8192 + 3840 + 48 + 10
= 12,090_{10}
\]

• Can write hexadecimal constants in C by prefixing ‘0x’ or ‘0X’ (“zero-x”) before sequence of hexadecimal digits. E.g.,
\[
\text{int } a = 0x2f3a;
\]
Converting from Decimal to Another Base

- **Recall**: Given a base-\(b\) number \((d_{n-1}...d_2d_1d_0)_b\), its decimal value is \(\sum_{i=0}^{n-1} d_i \times b^i\).

- What about the converse: Given a decimal number, what is its value in base \(b\)?

- **Example**: \(43_{10} = ?_2\)
Converting Decimal to Base $b$

- Given a decimal number $N$, do the following:
  1. Divide $N$ by $b$ producing a quotient $Q$ and a remainder $R$.
  2. If $Q \neq 0$, set $N$ to $Q$ and repeat 1.
  3. Read off sequence of remainders in reverse order; the result is the base $b$ equivalent of the decimal number.
Converting Decimal to Binary

- Example: $43_{10} = ?_2$
- Algorithm illustration:

<table>
<thead>
<tr>
<th>$quotient$</th>
<th>$rem$</th>
</tr>
</thead>
<tbody>
<tr>
<td>43 ÷ 2 = 21</td>
<td>1</td>
</tr>
<tr>
<td>21 ÷ 2 = 10</td>
<td>1</td>
</tr>
<tr>
<td>10 ÷ 2 = 5</td>
<td>0</td>
</tr>
<tr>
<td>5 ÷ 2 = 2</td>
<td>1</td>
</tr>
<tr>
<td>2 ÷ 2 = 1</td>
<td>0</td>
</tr>
<tr>
<td>1 ÷ 2 = 0</td>
<td>1</td>
</tr>
</tbody>
</table>

Read off remainders in reverse order

$43_{10} = 101011_2$

Stop when quotient = 0
Converting Decimal to Base $b$

- Another Example: $12,090_{10} = ?_{16}$

<table>
<thead>
<tr>
<th>Quotient</th>
<th>Rem</th>
</tr>
</thead>
<tbody>
<tr>
<td>12,090 ÷ 16 = 755</td>
<td>10 (= A&lt;sub&gt;16&lt;/sub&gt;)</td>
</tr>
<tr>
<td>755 ÷ 16 = 47</td>
<td>3</td>
</tr>
<tr>
<td>47 ÷ 16 = 2</td>
<td>15 (= F&lt;sub&gt;16&lt;/sub&gt;)</td>
</tr>
<tr>
<td>2 ÷ 16 = 0</td>
<td>2</td>
</tr>
</tbody>
</table>

Therefore, $12,090_{10} = 2F3A_{16}$
Converting Between Binary and Octal/Hexadecimal

- To convert from binary to octal/hex, can convert binary to decimal then convert decimal to octal/hex.
- Conversely, to convert from octal/hex to binary, can convert octal/hex to decimal then convert decimal to binary.
- However, a simpler algorithm can be obtained by exploiting the fact that binary/octet/hex are bases that are powers of two.
Converting Hex to Binary

• Convert each hex digit to its 4-bit binary equivalent:

\[
\begin{align*}
0_{16} &= 0000_2 \\
1_{16} &= 0001_2 \\
2_{16} &= 0010_2 \\
3_{16} &= 0011_2 \\
4_{16} &= 0100_2 \\
5_{16} &= 0101_2 \\
6_{16} &= 0110_2 \\
7_{16} &= 0111_2 \\
8_{16} &= 1000_2 \\
9_{16} &= 1001_2 \\
A_{16} &= 1010_2 \\
B_{16} &= 1011_2 \\
C_{16} &= 1100_2 \\
D_{16} &= 1101_2 \\
E_{16} &= 1110_2 \\
F_{16} &= 1111_2
\end{align*}
\]

• Concatenate resulting binary strings.

• Example:

\[
2AF8_{16} = 0010\ 1010\ 1111\ 1000_2 \\
= 10101011111000_2 \text{ (drop leading 0’s)}
\]

• What about octal to binary?
Converting Binary to Hex

• Group binary number into 4-bit sets from right to left (append 0’s to left if necessary).
• Convert each 4-bit set into its hex equivalent.
• Example: $10101011111000_2 = ?_{16}$

0010 1010 1111 1000

2 A F 8

• Therefore, $10101011111000_2 = 2AF8_{16}$
• What about octal to binary?
Integer Data Representation

• Will consider two types of integers:
  – unsigned integers
  – signed integers

• Unsigned integers
  – Set of non-negative integers \{0, 1, 2, \ldots\}
  – C supports unsigned integer type:

    ```c
    unsigned int a;
    a = 25; // valid
    a = -25; // invalid
    ```
Unsigned Integer Representation

- Let \( n \) be the data size in bits.
- Unsigned integer is represented in straight binary form, with 0’s appended to left if necessary.
- Example:
  - What is the 8-bit unsigned binary representation of \( 25_{10} \)?
  - \( 25_{10} = 11001_2 \)
  - Append 0’s to left to make 8 bits: 00011001
  - Therefore, unsigned 8-bit representation is 00011001
Unsigned Integer Representation

- Using $n$ bits, only unsigned integers in the range $[0, 2^n - 1]$ can be represented.
- Attempting to represent an integer outside this range results in overflow.

<table>
<thead>
<tr>
<th>Size in bits</th>
<th>Size in bytes</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1</td>
<td>[0, 255]</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>[0, 65,535]</td>
</tr>
<tr>
<td>32</td>
<td>4</td>
<td>[0, 4,294,967,295]</td>
</tr>
<tr>
<td>64</td>
<td>8</td>
<td>[0, 18,446,744,073,709,551,615]</td>
</tr>
</tbody>
</table>

- E.g., using $n = 8$ bits, $259_{10} = 100000011_2$ will result in overflow because it requires at least 9 bits.
Signed Integer Representation

• Signed integers: negative and positive integers, plus zero – i.e. {..., -2, -1, 0, 1, 2, ...}.

• In C, signed integers are of type int:

```c
int a;
a = 25;  // valid
a = -25; // valid
```

• Three commonly used binary representations:
  1. Sign-magnitude
  2. Ones’ complement
  3. Two’s complement
Sign-Magnitude Representation

• *n*-bit representation is divided into two parts:

  
  \[
  \begin{array}{c}
  \text{s} \\
  \text{magnitude}
  \end{array}
  \]

  
  \[
  \begin{array}{c}
  1 \\
  n-1
  \end{array}
  \]

• sign bit *s*:
  – 0 if positive integer
  – 1 if negative integer

• **magnitude**: unsigned binary representation of magnitude of number

• Example:
  – What is the 8-bit sign-magnitude representation of \(-25_{10}\)?
  – Since negative, sign bit *s* = 1
  – Magnitude is \(25_{10} = 11001_2\). Append 0’s to make 7 bits: 0011001
  – Therefore, sign-magnitude representation is 10011001
Sign-Magnitude Representation

- Negating a number: flipping the sign bit.
- Range representable in $n$ bits = $[-(2^{n-1}-1), (2^{n-1}-1)]$

<table>
<thead>
<tr>
<th></th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>-0</td>
<td>-1</td>
<td>-2</td>
<td>-3</td>
</tr>
</tbody>
</table>

- Drawbacks:
  - Zero has two distinct representations
  - Arithmetic algorithms are not easy to implement in hardware
One’s Complement Representation

• Ones’ complement bitwise operation:
  – Complement each bit: change 0 to 1; change 1 to 0
  – Example: ones’-complement(1011) = 0100

• Ones’ complement representation:
  – Positive integers: straight binary with a 0 in the most significant
    (leftmost) bit position
  – Negative integers: take the ones’ complement of the
    corresponding positive integer

• Example: \( n = 8 \)
  \[
  25_{10} \text{ is represented as } 00011001 \\
  -25_{10} \text{ is represented as } 11100110
  \]
Ones’ Complement Representation

- Negating a number: taking the ones’ complement of number
- Range representable in $n$ bits = $[-(2^{n-1}-1), (2^{n-1}-1)]$ (same range as sign-magnitude representation)

```
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>-3</td>
<td>-2</td>
<td>-1</td>
<td>-0</td>
</tr>
</tbody>
</table>
```

- Advantage:
  - Most significant bit still gives the “sign” of number: (msb = 0) → positive number, (msb = 1) → negative number
  - Arithmetic algorithms easier to implement in hardware (discussed later)
- Drawback: still has two representations for zero
Two’s Complement Representation

• Two’s complement bitwise operation:
  – Take the ones’ complement, then add 1
  – Example: two’s complement(1011) = 0100 + 1 = 0101

• Two’s complement representation:
  – Positive integers: straight binary with a 0 in the most significant (leftmost) bit position
  – Negative integers: take the two’s complement of the corresponding positive integer

• Example: $n = 8$
  - $25_{10}$ is represented as $00011001$
  - $-25_{10}$ is represented as $11100111$
Two’s Complement Representation

• Negating a number: taking the two’s complement of number
• Range representable in $n$ bits = $[-(2^{n-1}), (2^{n-1}-1)]$ (one extra negative number).

<table>
<thead>
<tr>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>-4</td>
<td>-3</td>
<td>-2</td>
<td>-1</td>
</tr>
</tbody>
</table>

• Advantages:
  – Most significant bit still gives the “sign” of number
  – Zero has a unique representation
  – Arithmetic algorithms easy to implement in hardware
• Predominantly used in today’s computers
Putting It All Together

• Consider the hex number $9E5A_{16}$. Assuming a 16-bit representation, what is the decimal value of this hex number if it encodes:
  1. an unsigned integer?
  2. a signed integer in sign-magnitude form?
  3. a signed integer in ones’ complement form?
  4. a signed integer in two’s complement form?
Putting It All Together

1. Decimal value as an unsigned integer:

\[ 9E5A_{16} = 9 \times 16^3 + 14 \times 16^2 + 5 \times 16^1 + 10 \times 16^0 \]
\[ = 36,864 + 3584 + 80 + 10 \]
\[ = 40,538_{10} \]
Putting It All Together

2. Decimal value as a signed integer in sign-magnitude form:

9E5A_{16} = 1001 1110 0101 1010_{2}
msb = sign bit = 1 → negative
magnitude = 001 1110 0101 1010_{2}

= 7770_{10}

Therefore, decimal value is \(-7770_{10}\)
Putting It All Together

3. Decimal value as a signed integer in ones’ complement form:

$$9E5A_{16} = 1001\ 1110\ 0101\ 1010_2$$

msb = 1 $\rightarrow$ negative

ones’ complement($1001\ 1110\ 0101\ 1010_2$)

= 0110\ 0001\ 1010\ 0101

= 24,997_{10}

Therefore, decimal value is $-24,997_{10}$
Putting It All Together

4. Decimal value as a signed integer in two’s complement form:

\[ 9\text{E5A}_{16} = 1001\ 1110\ 0101\ 1010_2 \]

msb = 1 → negative

two’s complement(1001\ 1110\ 0101\ 1010_2)

= 0110\ 0001\ 1010\ 0110

= 24,998_{10}

Therefore, decimal value is \(-24,998_{10}\)
Sign Extension

- It is sometimes necessary to convert the representation of an integer from one size to a larger size (e.g., 16 bits → 32 bits).
- E.g., multiplication of two $n$-bit integers generally produces a $2n$-bit product. Hardware multiplication algorithm requires each of the two $n$-bit operands to be expanded to $2n$ bits before the multiplication is performed.

Example:
$13_{10} \times 11_{10} = 143_{10}$

In binary ($n=5$):
$01101_2 \times 01011_2 = 0010001111_2$
Sign Extension

How should an integer be expanded to a larger size without changing its value?

• Unsigned integer
  - append 0’s to left

\[
\begin{array}{c}
11011 \\
0000011011
\end{array}
\]

= \(27_{10}\)

• Signed integer in sign-magnitude form
  - copy sign bit to msb position
  - fill remaining bits with 0’s

\[
\begin{array}{c}
11011 \\
1000001011
\end{array}
\]

= \(-11_{10}\)
Sign Extension

How should an integer be expanded to a larger size without changing its value?

• Signed integer in ones’ complement form
  - propagate sign bit (msb) to left

  \[
  \begin{array}{c}
  \text{11111} \\
  \text{11011}
  \end{array}
  \]
  \[
  = -4_{10}
  \]

• Signed integer in two’s complement form
  - same as ones’ complement
  - propagate sign bit (msb) to left

  \[
  \begin{array}{c}
  \text{11111} \\
  \text{11011}
  \end{array}
  \]
  \[
  = -5_{10}
  \]
Unsigned Integer Addition

- Recall unsigned addition in decimal:
  \[
  \begin{array}{cccc}
    & 1 & 1 & 1 \\
  \hline
  4 & 9 & 7 \\
  + & 9 & 2 & 8 \\
  \hline
  1 & 4 & 2 & 5
  \end{array}
  \]
  
  sum digits

- Similarly in binary:
  \[
  \begin{array}{cccc}
  \hline
  \text{carry bits} & 1 & 1 & 1 & 1 & 1 \\
  \hline
  0 & 1 & 1 & 0 & 1 & 1 \\
  + & 0 & 0 & 1 & 1 & 0 & 1 \\
  \hline
  1 & 0 & 1 & 0 & 0 & 0
  \end{array}
  \]
  
  (in decimal)

  \[
  \begin{array}{cccc}
  \hline
  \text{sum bits} \\
  \hline
  1 & 0 & 1 & 0 & 0 & 0
  \end{array}
  \]
  
  \[
  \begin{array}{cccc}
    = & 2 & 7 \\
  \hline
  \end{array}
  \]
  
  \[
  \begin{array}{cccc}
    = & + & 1 & 3 \\
  \hline
  \end{array}
  \]
  
  \[
  \begin{array}{cccc}
    \quad & 4 & 0
  \end{array}
  \]
Unsigned Integer Addition

• Recall that the range of unsigned integers that can be encoded using \( n \) bits is \([0, 2^n -1]\). E.g., for \( n = 6 \), range is \([0,63]\).

• Adding two \( n \)-bit unsigned integers may produce a result that requires more than \( n \) bits. This is called an **overflow**.

• Example:

\[
\begin{array}{c}
1111 \\
0110111 \\
+1010000 \\
\hline
10000111
\end{array}
\]

( in decimal)

\[
27
\]

( in decimal)

\[
+40
\]

\[
67
\]

• Detecting overflow in unsigned addition: *An overflow occurs when there is a carry out of the most significant bit position.*
Unsigned Integer Subtraction

• Recall subtraction in decimal:

\[
\begin{array}{c}
11 \\
\underline{8 \ 15}
\end{array}
\quad \text{borrow digits}
\]

\[
\begin{array}{c}
9 \ 2 \ 5
\end{array}
\quad \text{minuend}
\]

- \[
\begin{array}{c}
6 \ 4 \ 7
\end{array}
\quad \text{subtrahend}
\]

\[
\begin{array}{c}
2 \ 7 \ 8
\end{array}
\]

• Similarly in binary:

\[
\begin{array}{c}
10
\end{array}
\quad \text{borrow bits} \quad \rightarrow \quad \begin{array}{c}
0 \ \emptyset \ 10
\end{array}
\quad \begin{array}{c}
\text{(in decimal)}
\end{array}
\]

\[
\begin{array}{c}
0 \ \emptyset \ \emptyset \ 1 \ 1 \ 1
\end{array}
\quad = \quad \begin{array}{c}
2 \ 7
\end{array}
\]

- \[
\begin{array}{c}
0 \ 0 \ 1 \ 1 \ 0 \ 1
\end{array}
\quad = \quad \begin{array}{c}
- \ 1 \ 3
\end{array}
\]

\[
\begin{array}{c}
0 \ 0 \ 1 \ 1 \ 1 \ 0
\end{array}
\quad \begin{array}{c}
1 \ 4
\end{array}
\]
Unsigned Integer Subtraction

• For unsigned integers, subtraction is defined only if minuend $\geq$ subtrahend. (Why? Because there is no notion of a negative number.)

• Hence, overflow cannot occur since minuend $\geq (\text{minuend} - \text{subtrahend}) \geq 0$.

• Easy to determine which of two unsigned integers is larger:
  – Compare bitwise starting from left (msb).
  – Stop at bit position where the numbers differ.
  – The number with a ‘1’ bit is larger.

```
0 1 1 0 1 1 1
0 0 1 1 0 1
```

\[\uparrow\]
Signed Integer Arithmetic

- Recall that signed integers have three commonly used binary representations:
  1. Sign-magnitude
  2. Ones’ complement
  3. Two’s complement
- We will see that among these three representations, two’s complement affords the simplest and most efficient algorithms (and hardware implementations) for arithmetic operations.
- Consequently, two’s complement is the dominant signed integer encoding used in today’s computers.
Addition in Sign-Magnitude Form

1. If sign bits are same:
   (a) Add magnitudes.
   (b) Copy sign bit to result.

- If sign bits are same, overflow may occur when sum of magnitudes cannot fit in $n-1$ bits.

- How to detect overflow? Exercise
Addition in Sign-Magnitude Form

2. If sign bits are different:
   (a) Subtract smaller magnitude from larger magnitude.
   (b) Copy to result sign bit of larger-magnitude number.

\[
\begin{array}{cccc}
0 & 0 & 1 & 0 & 1 & 1 \\
+ & 1 & 1 & 1 & 0 & 1 & 0 \\
\hline
????
\end{array}
\quad
\begin{array}{cccc}
1 & 1 & 0 & 1 & 0 \\
- & 0 & 1 & 0 & 1 & 1 \\
\hline
0 & 1 & 1 & 1 & 1
\end{array}
\quad
\begin{array}{cccc}
0 & 0 & 1 & 0 & 1 & 1 \\
+ & 1 & 1 & 1 & 0 & 1 & 0 \\
\hline
1 & 0 & 1 & 1 & 1 & 1
\end{array}
\quad
\begin{array}{cccc}
0 & 0 & 1 & 0 & 1 & 1 \\
+ & 1 & 1 & 1 & 0 & 1 & 0 \\
\hline
1 & 0 & 1 & 1 & 1 & 1
\end{array}
\]

- Complicated algorithm: first need to compare signs to decide whether to add or subtract.
Addition in Ones’ Complement Form

• Simpler addition algorithm:

1. Add the two numbers as in unsigned addition.
2. If there is a carry out of the msb position (called end-around carry), increment the result by 1.

\[
\begin{align*}
\text{1111} & \text{110111} = 11 \\
+100101 & = +26 \\
\hline
110000 & = -15 \\
\end{align*}
\]

\[
\begin{align*}
\text{110100} & \text{= -11} \\
+011010 & = +26 \\
\hline
001110 & +1 \\
\hline
001111 & = 15
\end{align*}
\]
Addition in Ones’ Complement Form

- Facts: Let $n$ be the size of the representation
  - Ones’-complement($x$) = $(2^n - 1) - x$
  - End-around carry occurs only when result, treated as an unsigned integer, is $\geq 2^n$
  - Adding end-around carry is same as subtracting $(2^n - 1)$ from result
- Ones’ complement addition algorithm works!

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Addition result</th>
<th>Final result in 1’s compl. form</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>+</td>
<td>$A + B$</td>
<td>$A+B$</td>
</tr>
<tr>
<td>+</td>
<td>-</td>
<td>$A + (2^n - 1) - B = (2^n - 1) - (B-A)$</td>
<td>$-(B-A) = A-B$</td>
</tr>
<tr>
<td>-</td>
<td>+</td>
<td>$(2^n - 1) - A + B = (2^n - 1) - (A-B)$</td>
<td>$-(A-B) = B-A$</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>$(2^n - 1) - A + (2^n - 1) - B$ = $(2^n - 1) + (2^n - 1) - (A+B)$</td>
<td>$-(A+B)$</td>
</tr>
</tbody>
</table>

End-around carry occurs: subtract $(2^n - 1)$
Addition in Ones’ Complement Form

• Overflow Detection
  – Recall that range of numbers representable in \( n \) bits is \([- (2^{n-1} - 1), (2^{n-1} - 1)]\). E.g., if \( n = 6 \), range is \([-31,31]\)
  – Overflow cannot occur when two numbers have opposite sign. Why?
  – When two numbers have the same sign, overflow can occur:

\[
\begin{align*}
1 & 1 \\
0 & 0 \ 1 & 0 & 1 & 1 & = & 11 \\
+ & 0 & 1 & 1 & 0 & 1 & 0 & = & +2 6 \\
\hline
1 & 0 & 0 & 1 & 0 & 1 & \neq & 3 7
\end{align*}
\]

\[
\begin{align*}
& 1 \\
& 11 & 0 & 1 & 0 & 0 & = & -11 \\
+ & 1 & 0 & 0 & 1 & 0 & 1 & = & + -2 6 \\
\hline
& 0 & 1 & 1 & 0 & 0 & 1 \\
& +1 \\
\hline
& 0 & 1 & 1 & 0 & 1 & 0 & \neq & -3 7
\end{align*}
\]

– Detecting overflow: *Overflow occurs when the two numbers have the same sign and the result has the opposite sign (sign reversal).*
Addition in Two’s Complement Form

- Even simpler addition algorithm:
  1. Add the two numbers as in unsigned addition.
  2. Discard carry out of the msb position.

\[
\begin{align*}
1111 & \quad \text{discard} \quad \uparrow 11 \\
001011 & = 11 & 110101 & = -11 \\
+ 100110 & = +26 & + 011010 & = +26 \\
\hline
110001 & = -15 & 001111 & = 15
\end{align*}
\]

- Detecting overflow (same as ones’ complement addition): \textit{Overflow occurs when the two numbers have the same sign and the result has the opposite sign (sign reversal).}
Addition in Two’s Complement Form

• Facts: Let $n$ be the size of the representation
  – Two’s-complement($x$) = $2^n - x$
  – End-around carry occurs only when result, treated as an unsigned integer, is $\geq 2^n$
  – Discarding end-around carry is same as subtracting $2^n$ from result

• Two’s complement addition algorithm works!

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Addition result</th>
<th>Final result in 1’s compl. form</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>+</td>
<td>$A + B$</td>
<td>$A+B$</td>
</tr>
<tr>
<td>+</td>
<td>-</td>
<td>$A + 2^n - B = 2^n - (B-A)$</td>
<td>$-(B-A) = A-B$</td>
</tr>
<tr>
<td>-</td>
<td>+</td>
<td>$2^n - A + B = 2^n - (A-B)$</td>
<td>$-(A-B) = B-A$</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>$2^n - A + 2^n - B$</td>
<td>$-(A+B)$</td>
</tr>
</tbody>
</table>

End-around carry occurs: subtract $2^n$
Signed Integer Subtraction

- Subtraction reduces to addition because of the following identity: \( A - B \equiv A + (-B) \)
- Therefore, to compute \( A - B \):
  1. Compute \(-B\).
  2. Add \(-B\) to \(A\).
- Example: Compute \(11_{10} - 26_{10}\) assuming the numbers are represented in 6-bit two’s complement form.

  \( 11_{10} = 001011 \) in 6-bit two’s complement form
  \( 26_{10} = 011010 \) in 6-bit two’s complement form

  1. \(-26_{10} = \text{two’s complement(011010)} = 100110\)
  2. Therefore, \(11_{10} - 26_{10} = 001011 + 100101 = 110001 = -15_{10}\)
Unsigned Integer Multiplication

• Recall unsigned multiplication in decimal:

\[
\begin{array}{cccc}
1 & 3 & 5 & \text{multiplicand} \\
\times & 4 & 2 & 3 & \text{multiplier} \\
\hline
4 & 0 & 5 \\
2 & 7 & 0 \\
5 & 4 & 0 \\
\hline
5 & 7 & 1 & 0 & 5 & \text{product}
\end{array}
\]
Unsigned Integer Multiplication

- Similarly in binary:

\[
\begin{array}{c}
1 \ 1 \ 0 \ 1 \\
\times \ 0 \ 1 \ 0 \ 1 \\
\hline
1 \ 1 \ 0 \ 1 \\
0 \ 0 \ 0 \ 0 \\
1 \ 1 \ 0 \ 1 \\
0 \ 0 \ 0 \ 0 \\
\hline
1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1
\end{array}
\]

= \ 13_{10}
= \ 5_{10}

- The product of two \( n \)-bit numbers results can be as long as \( 2n \) bits, but not longer.

- Hence, no overflow can occur so long as the product is stored using \( 2n \) bits. Most hardware multipliers do this.
Unsigned Integer Multiplication

- Observe that each partial product is simply either:
  - multiplicand shifted left $i$ times (if $i$-th multiplier bit = 1), or
  - zero (if multiplier bit = 0)

```
  1 1 0 1
× 0 1 0 1
  1 1 0 1         (multiplicand × 1) shift left 0
  0 0 0 0         (multiplicand × 0) shift left 1
  1 1 0 1         (multiplicand × 1) shift left 2
  0 0 0 0         (multiplicand × 0) shift left 3
  1 0 0 0 0 0 0 1
```
Unsigned Integer Multiplication

• Unsigned integer multiplication algorithm

1. Initialize $2n$-bit product to 0

2. For $i=0$ to $n-1$ do the following:
   a. If $i$-th bit of multiplier = 1, add multiplicand to product
      else, add 0 to product
   b. Shift multiplicand left by 1 bit; fill LSB with 0

3. Product holds multiplicand $\times$ multiplier
Signed Integer Multiplication

- If binary numbers are treated as signed integers in two’s complement form, the unsigned multiplication algorithm does not work!

\[
\begin{array}{cccc}
  1 & 1 & 0 & 1 \\
\times & 0 & 1 & 0 & 1 \\
\hline
  1 & 1 & 0 & 1 \\
  0 & 0 & 0 & 0 \\
\hline
  1 & 1 & 0 & 1 \\
  0 & 0 & 0 & 0 \\
\hline
  1 & 0 & 0 & 0 & 0 & 0 & 1 \\
\end{array}
\]

= \(-3_{10}\)

= \(5_{10}\)

• In the example above, product is two’s complement representation of \(-63_{10}\) which is the wrong result.
Signed Integer Multiplication

• Why doesn’t the algorithm work?

\[
\begin{array}{c}
1 & 1 & 0 & 1 \\
\times & 0 & 1 & 0 & 1 \\
\hline
0 & 0 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\end{array}
\]

This is a positive number \((+13_{10})!\)

\[
1101_{10} = -3_{10} \\
0101_{10} = 5_{10} \\
1000001_{10} \neq -15_{10}
\]

• Because, implicitly, the partial products formed from the shifted multiplicand are positive numbers even though the multiplicand is a negative number.
Signed Integer Multiplication

• To correct the problem, the multiplicand should be **sign-extended** to $2n$ bits before forming partial products.

```
  111111101  = -3_{10}
× 0101
  111111101
  00000000
  1111101
  000000
  1111101
```

```
  111110001  = -15_{10}
```

• Note that carry out of the $(2n)$-th bit is discarded in each addition because we’re performing two’s complement addition. (That is, we keep only $2n$ least significant bits of product.)
Signed Integer Multiplication

• Unfortunately, the tweak does not work if the multiplier is negative:

\[
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
\times & 1 & 0 & 1 & 1 \\
\hline
1 & 1 & 1 & 1 & \quad 1 & 1 & 0 & 1 \\
1 & 1 & 1 & \quad 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 1 \\
\hline
1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & \neq & 15_{10}
\end{array}
\]

\[
1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
\hline
\]

• \textbf{Why?} Because the bits of the multiplier no longer specify appropriately shifted copies of the multiplicand.
Signed Integer Multiplication

Dealing with a negative multiplier:

- Let A = multiplicand and B = multiplier
- Rewrite the multiplier as $B = \pm B^*$ where $B^*$ is the magnitude of B
- If $B = +B^*$ (i.e., B is positive) then the multiplication algorithm gives the correct product because $A \times B = A \times B^*$
- If $B = -B^*$ (i.e., B is negative) the multiplication algorithm gives the incorrect product because it actually computes $A \times \text{two’s complement}(B^*) = A \times (2^n - B^*) = (A \times 2^n) - (A \times B^*)$
- If $B = -B^*$ the product should be $A \times B = A \times (-B^*) = -(A \times B^*)$
- Key Observations:
  1. Subtracting $(A \times 2^n)$ from $(A \times 2^n) - (A \times B^*)$ yields $-(A \times B^*)$ which is the correct product
  2. $(A \times 2^n)$ is equal to A shifted to the left $n$ times
  3. Subtracting $(A \times 2^n)$ is the same as adding the two’s complement of $(A \times 2^n)$
Signed Integer Multiplication

- Signed multiplication algorithm with numbers in two’s complement form

1. Set-up:
   a. Initialize 2n-bit product to 0
   b. Sign-extend multiplicand to 2n bits

2. For \( i=0 \) to \( n-1 \) do the following:
   a. If \( i \)-th bit of multiplier = 1, add multiplicand to product
      else, add 0 to product
   b. Shift multiplicand left by 1 bit; fill LSB with 0

3. If MSB of multiplier = 1, add two’s complement of multiplicand to product

4. Product holds multiplicand \( \times \) multiplier
Signed Integer Multiplication

• Algorithm Illustration:

```
1 1 1 1 1 1 0 1
× 1 0 1 1
---------------------
1 1 1 1 1 1 1 0 1
1 1 1 1 1 1 0 1 0
0 0 0 0 0 0 0 0 0
1 1 1 0 1 0 0 0
1 1 0 1 1 0 0 0
0 0 1 1 0 0 0 0
0 0 0 0 1 1 1 1 = 15_{10} = -3_{10} = -5_{10}
```

MSB=1:
- Take two’s complement of multiplicand
  11010000 → 00110000
- Add it to product
Unsigned Integer Division

• Recall unsigned division in decimal:

\[
\text{dividend} = (\text{quotient} \times \text{divisor}) + \text{remainder}
\]

• Satisfies the following property:
Unsigned Integer Division

• Similarly in binary:

\[
\begin{array}{c}
6_{10} = \quad 1 \quad 1 \quad 0 \\
\end{array}
\begin{array}{c|c}
\hline
0 & 1 & 1 & 1 \\
\hline
1 & 0 & 1 & 1 & 1 & 0 \\
- & 1 & 1 & 0 \\
\hline
1 & 0 & 1 & 1 \\
- & 1 & 1 & 0 \\
\hline
1 & 0 & 1 & 0 \\
- & 1 & 1 & 0 \\
\hline
1 & 0 & 0 \\
\end{array}
\]

= \quad 7_{10} = 46_{10}

Check:
46 = 7 \times 6 + 4

= 4_{10}
Unsigned Integer Division

• Unsigned division algorithm

1. Set-up:
   a. Q = 0; R= dividend
   b. D = divisor shifted left (LSB zero-filled) until same number of bits as dividend

2. If R < divisor, stop: Q contains quotient; R contains remainder

3. If R < D then shift Q left; fill LSB with 0

4. If R ≥ D then shift Q left; fill LSB with 1; R = R − D

5. Shift D right; fill MSB with 0

6. Go to 2
Unsigned Integer Division

• Algorithm Illustration:

```
  dividend = 101110
  divisor = 110

  R  1 0 1 1 1 0
  D  1 1 0 0 0 0
      R < D   Q  0 0 0 0 0

  R  1 0 1 1 1 0
  D  0 1 1 0 0 0
      R ≥ D   Q  0 0 0 1

  R  0 1 0 1 1 0
  D  0 0 1 1 0 0
      R ≥ D   Q  0 0 1 1

  R  0 0 1 0 1 0
  D  0 0 0 1 1 0
      R ≥ D   Q  0 1 1 1

  R  0 0 0 1 0 0
  D  0 0 0 1 1 0
      R < divisor Q  0 1 1 1
```

remainder

quotient
Signed Integer Division

• No simple direct way; reduce to unsigned division

• Algorithm:
  1. Take absolute values of dividend and divisor and apply unsigned division algorithm
  2. Attach the proper signs to quotient and remainder:

\[
\text{sign(quotient)} = \begin{cases} 
+ & \text{if } \text{sign(dividend)} = \text{sign(divisor)} \\
- & \text{if } \text{sign(dividend)} \neq \text{sign(divisor)} 
\end{cases}
\]

\[
\text{sign(remainder)} = \text{sign(dividend)}
\]

• Above sign rule ensures that:

\[
\text{dividend} = (\text{quotient} \times \text{divisor}) + \text{remainder}
\]
Signed Integer Division

• Illustration of sign rule:

1. \((+7) ÷ (+3) = ?\)
2. \((-7) ÷ (+3) = ?\)
3. \((+7) ÷ (-3) = ?\)
4. \((-7) ÷ (-3) = ?\)

Unsigned division yields: \(7 ÷ 3 = 2 \text{ rem. } 1\)

Therefore:

Check: dividend = quot. × divisor + rem.

1. \((+7) ÷ (+3) = (+2) \text{ rem. } (+1)\)
   \((+7) = (+2) × (+3) + (+1)\)
2. \((-7) ÷ (+3) = (-2) \text{ rem. } (-1)\)
   \((-7) = (-2) × (+3) + (-1)\)
3. \((+7) ÷ (-3) = (-2) \text{ rem. } (+1)\)
   \((+7) = (-2) × (-3) + (+1)\)
4. \((-7) ÷ (-3) = (+2) \text{ rem. } (-1)\)
   \((-7) = (+2) × (-3) + (-1)\)
Character Data

- In C, the `char` data type is considered an integer data type.

- **ASCII** (American Standard for Computer Information Exchange) is most commonly used standard for encoding characters.

- Original ASCII character set encodes 128 characters, each assigned a unique 7-bit binary value.
Character Data

• To meet the demand for more characters and symbols used in other languages, ASCII character set was subsequently extended to encode 256 characters, each assigned an 8-bit binary value.

• Thus each character fits in exactly one byte of storage.
Character Data

- Extended ASCII does not address the diversity of writing systems in the world (e.g., Hebrew, Russian, Chinese, ...). To address this problem, Unicode was developed.
- Current Unicode standard supports over 107,000 characters across over 90 writing systems.
- Unicode’s “Universal Character Set” encodes characters as 32-bit binary numbers. However, alternate encodings are supported. E.g., UTF-8 uses a variable-length encoding of 1 to 4 bytes for each character.
- The first 128 characters of UTF-8 corresponds one-to-one with ASCII, making valid ASCII text valid UTF-8-encoded Unicode as well. (However, not extended ASCII.)
- See http://www.unicode.org/ for further information.
- Java uses Unicode for its representation of strings. C uses ASCII but program libraries are available to support Unicode.