CHAPTER 3
DATA REPRESENTATION

Kinds Of Data

- Numbers
  - Integers
    - Unsigned
    - Signed
  - Reals
    - Fixed-Point
    - Floating-Point
    - Binary-Coded Decimal

- Text
  - ASCII Characters
  - Strings

- Other
  - Graphics
  - Images
  - Video
  - Audio

Numbers Are Different!

- Computers use binary (not decimal) numbers (0's and 1's).
  - Requires more digits to represent the same magnitude.
- Computers store and process numbers using a fixed number of digits ("fixed-precision").
- Computers represent signed numbers using 2's complement instead of the more natural (for humans) "sign-plus-magnitude" representation.

Positional Number Systems

- Numeric values are represented by a sequence of digit symbols.
- Symbols represent numeric values.
  - Symbols are not limited to ‘0’-'9'!
- Each symbol’s contribution to the total value of the number is weighted according to its position in the sequence.
Polynomial Evaluation

Whole Numbers (Radix = 10):

\[1234_{10} = 1 \times 10^3 + 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0\]

With Fractional Part (Radix = 10):

\[36.72_{10} = 3 \times 10^1 + 6 \times 10^0 + 7 \times 10^{-1} + 2 \times 10^{-2}\]

General Case (Radix = R):

\[(S_1S_0.S_{-1}S_{-2})_R = S_1 \times R^1 + S_0 \times R^0 + S_{-1} \times R^{-1} + S_{-2} \times R^{-2}\]

Converting Radix R to Decimal

\[36.72_8 = 3 \times 8^1 + 6 \times 8^0 + 7 \times 8^{-1} + 2 \times 8^{-2}\]
\[= 24 + 6 + 0.875 + 0.03125\]
\[= 30.90625_{10}\]

**Important:** Polynomial evaluation doesn't work if you try to convert in the other direction – I.e., from decimal to something else! Why?

Binary to Decimal Conversion

Converting to decimal, so we can use polynomial evaluation:

\[10110101_2\]
\[= 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0\]
\[= 128 + 32 + 16 + 4 + 1\]
\[= 181_{10}\]

Decimal to Binary Conversion

- Converting to binary – can’t use polynomial evaluation!
- Whole part and fractional parts must be handled separately!
  - Whole part: Use repeated division.
  - Fractional part: Use repeated multiplication.
  - Combine results when finished.
Decimal to Binary Conversion (Whole Part: Repeated Division)

• Divide by target radix (2 in this case)
• Remainders become digits in the new representation (0 <= digit < R)
• Digits produced in right to left order.
• Quotient is used as next dividend.
• Stop when the quotient becomes zero, but use the corresponding remainder.

97 ÷ 2 → quotient = 48, remainder = 1 (LSB)
48 ÷ 2 → quotient = 24, remainder = 0.
24 ÷ 2 → quotient = 12, remainder = 0.
12 ÷ 2 → quotient = 6, remainder = 0.
 6 ÷ 2 → quotient = 3, remainder = 0.
 3 ÷ 2 → quotient = 1, remainder = 1.
 1 ÷ 2 → quotient = 0 (Stop) remainder = 1 (MSB)

Result = 1 1 0 0 0 0 1₂

Decimal to Binary Conversion (Fractional Part: Repeated Multiplication)

• Multiply by target radix (2 in this case)
• Whole part of product becomes digit in the new representation (0 <= digit < R)
• Digits produced in left to right order.
• Fractional part of product is used as next multiplicand.
• Stop when the fractional part becomes zero (sometimes it won’t).

.1 × 2 → 0.2 (fractional part = .2, whole part = 0)
  .2 × 2 → 0.4 (fractional part = .4, whole part = 0)
  .4 × 2 → 0.8 (fractional part = .8, whole part = 0)
  .8 × 2 → 1.6 (fractional part = .6, whole part = 1)
  .6 × 2 → 1.2 (fractional part = .2, whole part = 1)

Result = .00011001100110011₂…..

(How much should we keep?)
Moral

• Some fractional numbers have an exact representation in one number system, but not in another! E.g., 1/3rd has no exact representation in decimal, but does in base 3!
• What about 1/10th when represented in binary?
• Can these representation errors accumulate?
• What does this imply about equality comparisons of real numbers?

Counting

• Principle is the same regardless of radix.
  – Add 1 to the least significant digit.
  – If the result is less than R, write it down and copy all the remaining digits on the left.
  – Otherwise, write down zero and add 1 to the next digit position, etc.

Counting in Binary

<table>
<thead>
<tr>
<th>Dec</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
</tr>
</tbody>
</table>

Note the pattern!
• LSB (bit 0) toggles on every count.
• Bit 1 toggles on every other count.
• Bit 2 toggles on every fourth count.
• Etc....

Question:

• Do you trust the used car salesman that tells you that the 1966 Mustang he wants to sell you has only the 13,000 miles that it’s odometer shows?
• If not, what has happened?
• Why?
Representation Rollover

- Consequence of *fixed precision*.
- Computers use fixed precision!
- Digits are lost on the left-hand end.
- Remaining digits are still correct.
- Rollover while counting . . .
  Up: “999999” ➔ “000000” (R\(^n\)-1 ➔ 0)
  Down: “000000” ➔ “999999” (0 ➔ R\(^n\)-1)

Rollover in Unsigned Binary

- Consider an 8-bit byte used to represent an unsigned integer:
  - Range: 00000000 ➔ 11111111 (0 ➔ 255\(_{10}\))
  - Incrementing a value of 255 should yield 256, but this exceeds the range.
  - Decrementing a value of 0 should yield –1, but this exceeds the range.
  - *Exceeding the range* is known as overflow.

Surprise! Rollover is **not** synonymous with overflow!

- Rollover describes a pattern sequence behavior.
- Overflow describes an arithmetic behavior.
- Whether or not rollover causes overflow depends on how the patterns are interpreted as numeric values!
  - E.g., In signed two’s complement representation, 11111111 ➔ 00000000 corresponds to counting from minus one to zero.

Hexadecimal Numbers (Radix = 16)

- The *number* of digit symbols is determined by the radix (e.g., 16)
- The *value* of the digit symbols range from 0 to 15 (0 to R-1).
- The *symbols* are 0-9 followed by A-F.
- Conversion between binary and hex is trivial!
- Use as a shorthand for binary (significantly fewer digits are required for same magnitude).
### Memorize This!

<table>
<thead>
<tr>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
</tr>
</tbody>
</table>

### Binary/Hex Conversions

Hex digits are in one-to-one correspondence with groups of four binary digits:

<table>
<thead>
<tr>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>A</td>
<td>1010</td>
</tr>
<tr>
<td>B</td>
<td>1011</td>
</tr>
<tr>
<td>C</td>
<td>1100</td>
</tr>
<tr>
<td>D</td>
<td>1101</td>
</tr>
<tr>
<td>E</td>
<td>1110</td>
</tr>
<tr>
<td>F</td>
<td>1111</td>
</tr>
</tbody>
</table>

- Conversion is a simple table lookup!
- Zero-fill on left and right ends to complete the groups!
- Works because $16 = 2^4$ (power relationship)

### Two Interpretations

Unsigned: 167<sub>10</sub>  
Signed: 10100111<sub>2</sub>  
-89<sub>10</sub>

- Signed vs. unsigned is a matter of interpretation; thus a single bit pattern can represent two different values.
- Allowing both interpretations is useful:
  - Some data (e.g., count, age) can never be negative, and having a greater range is useful.

### One Hardware Adder Handles Both! (or subtractor)

- Manipulates bit patterns, not numbers!
Which is Greater: 1001 or 0011?

Answer: It depends!

So how does the computer decide:
“if (x > y) ..” /* Is this true or false? */

It’s a matter of interpretation, and depends on how x and y were declared: signed? Or unsigned?

Why Not Sign+Magnitude?

• Complicates addition:
  – To add, first check the signs. If they agree, then add the magnitudes and use the same sign; else subtract the smaller from the larger and use the sign of the larger.
  – How do you determine which is smaller/larger?

• Complicates comparators:
  – Two zeroes!

Copyright © 2000, Daniel W. Lewis. All Rights Reserved.
Why 2’s Complement?

1. Just as easy to determine sign as in sign-magnitude.
2. Almost as easy to change the sign of a number.
3. Addition can proceed w/out worrying about which operand is larger.
4. A single zero!
5. One hardware adder works for both signed and unsigned operands.

<table>
<thead>
<tr>
<th>Value</th>
<th>2’s Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td>1100</td>
</tr>
<tr>
<td>-2</td>
<td>1110</td>
</tr>
<tr>
<td>-1</td>
<td>1111</td>
</tr>
<tr>
<td>+1</td>
<td>0001</td>
</tr>
<tr>
<td>+2</td>
<td>0010</td>
</tr>
<tr>
<td>+3</td>
<td>0011</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
</tr>
</tbody>
</table>

Changing the Sign

Sign+Magnitude: 2’s Complement:

+5 = 0101
-5 = 1101

Invert Increment

+5 = 0101
1010

+1

-5 = 1011

Easier Hand Method

Step 1: Copy the bits from right to left, through and including the first 1.
Step 2: Copy the inverse of the remaining bits.

+4 = 0100
-4 = 1100

Representation Width

Be Careful! You must be sure to pad the original value out to the full representation width before applying the algorithm!

Wrong: +25 = 11001 \rightarrow 00111 \rightarrow 00000111 = +7

Right: +25 = 11001 \rightarrow 00011001 \rightarrow 11100111 = -25

If positive: Add leading 0’s
If negative: Add leading 1’s

Apply algorithm
Expand to 8-bits
Apply algorithm
**Subtraction Is Easy!**

![Diagram](image)

- **A**
- **B**
- **Controlled Inverter**
- **Adder**

Result

\[ 0 = \text{add}, \quad 1 = \text{sub (A-B)} \]

Just a bunch of exclusive-OR gates!

---

**2’s Complement Anomaly!**

-128 = \[ 1000\ 0000 \] (8 bits)

Step 1: Invert all bits \[ \rightarrow 0111\ 1111 \]

Step 2: Increment \[ \rightarrow 1000\ 0000 \]

Same result with either method! **Why?**

---

**Range of Unsigned Integers**

Each of ‘n’ bits can have one of two values.

Total # of patterns of n bits = \[ 2 \times 2 \times 2 \times \ldots \times 2 \]

\[ = 2^n \]

If n-bits are used to represent an unsigned integer value:

**Range:** \[ 0 \text{ to } 2^n-1 \] (\[ 2^n \text{ different values} \])

---

**Range of Signed Integers**

- **Half** of the \[ 2^n \text{ patterns} \] will be used for positive values, and half for negative.

- **Half is** \[ 2^{n-1} \].

- Positive Range: \[ 0 \text{ to } 2^{n-1}-1 \] (\[ 2^{n-1} \text{ patterns} \])

- Negative Range: \[ -2^{n-1} \text{ to } -1 \] (\[ 2^{n-1} \text{ patterns} \])

- 8-Bits (\[ n = 8 \]): \[ -2^7 (-128) \text{ to } +2^7-1 (+127) \]
Unsigned Overflow

1100 (12)
+0111 ( 7)
10011

Lost

(Result limited by word size)

0011 ( 3) \textit{wrong}

Value of lost bit is \(2^n (16)\).

\[
16 + 3 = 19
\]

(The right answer!)

Signed Overflow

• Overflow is impossible ☺ when adding (subtracting) numbers that have different (same) signs.

• Overflow occurs when the magnitude of the result extends into the sign bit position:

\[
01111111 ightarrow (0)10000000
\]

This is not rollover!

Signed Overflow

\(-120_{10} \rightarrow 10001000_2\)
\(-17_{10} \rightarrow +11101111_2\)

\(-137_{10}\)

\[
\begin{array}{c}
101110111_2
01110111_2 \text{ (keep 8 bits)}
\end{array}
\]

\(+119_{10}\) \textit{wrong}

Note: \(119 - 2^8 = 119 - 256 = -137\)

Floating-Point Reals

Three components:

\[
\begin{array}{l}
\pm \text{significand} \times 2^{\text{exponent}}
\end{array}
\]

\[
\begin{array}{l}
\pm \text{significand} \times 2^{\text{exponent}}
\end{array}
\]

Base is implied

A biased integer value

An unsigned fractional value
**Single-precision Floating-point Representation**

<table>
<thead>
<tr>
<th>S</th>
<th>Exp+127</th>
<th>Significand</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.000</td>
<td>0 10000000</td>
<td>(1).00000000000000000000000</td>
</tr>
<tr>
<td>1.000</td>
<td>0 01111111</td>
<td>(1).00000000000000000000000</td>
</tr>
<tr>
<td>0.750</td>
<td>0 01111110</td>
<td>(1).10000000000000000000000</td>
</tr>
<tr>
<td>0.500</td>
<td>0 01111110</td>
<td>(1).00000000000000000000000</td>
</tr>
<tr>
<td>0.000</td>
<td>0 00000000</td>
<td>(0).00000000000000000000000</td>
</tr>
<tr>
<td>-0.501</td>
<td>1 01111110</td>
<td>(1).00000000000000000000000</td>
</tr>
<tr>
<td>-0.751</td>
<td>1 01111110</td>
<td>(1).10000000000000000000000</td>
</tr>
<tr>
<td>-1.001</td>
<td>1 01111111</td>
<td>(1).00000000000000000000000</td>
</tr>
<tr>
<td>-2.001</td>
<td>1 10000000</td>
<td>(1).00000000000000000000000</td>
</tr>
</tbody>
</table>

**Fixed-Point Reals**

Three components:

- **Implied binary point**
- **Whole part**
- **Fractional part**

**Fixed vs. Floating**

- **Floating-Point**:
  - Pro: Large dynamic range determined by exponent; resolution determined by significand.
  - Con: Implementation of arithmetic in hardware is complex (slow).

- **Fixed-Point**:
  - Pro: Arithmetic is implemented using regular integer operations of processor (fast).
  - Con: Limited range and resolution.

**Fixed-Point & Scale Factors**

- The position of the binary point is determined by a *scale factor*.
- Different variables can have a different scale factors.
- Determine scale factor by expected range and required resolution.
- Programmer must keep track of scale factors! (Tedious)
### Fixed-Point Add/Subtract Using Operands w/Same Scale Factors

<table>
<thead>
<tr>
<th>Operand/Result</th>
<th>Bit Pattern</th>
<th>Integer</th>
<th>Scale Factor</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>00011.110</td>
<td>+30</td>
<td>2^-3 = 1/8</td>
<td>+3.750</td>
</tr>
<tr>
<td>B</td>
<td>00110.011</td>
<td>+51</td>
<td>2^-3 = 1/8</td>
<td>+6.375</td>
</tr>
<tr>
<td>A + B</td>
<td>01010.001</td>
<td>+81</td>
<td>2^-3 = 1/8</td>
<td>+10.125</td>
</tr>
<tr>
<td>A - B</td>
<td>11101.011</td>
<td>-21</td>
<td>2^-3 = 1/8</td>
<td>-2.625</td>
</tr>
</tbody>
</table>

### Fixed-Point Add/Subtract Using Operands w/Different Scale Factors

- Must align binary points before adding or subtracting; this makes scale factors the same.
- Two possibilities:
  - If you shift the operand with fewer fractional bits left, be careful that it doesn't cause an overflow.
  - If you shift the operand with more fractional bits right, be careful that it doesn't cause a loss of precision.
- Either approach may be used, but the scale factor of the resulting sum or difference will be quite different.

### Fixed-Point Multiplication/Division

- No need to pre-align binary points!
- Number of fractional bits in result (determines the scale factor):
  - Multiplication: The number of fractional bits in the multiplicand plus the number in the multiplier.
  - Division: The number of fractional bits in the dividend less the number in the divisor.

### Multiplying Fixed-Point Real Numbers

<table>
<thead>
<tr>
<th>Operand/Result</th>
<th>Bit Pattern</th>
<th>Int</th>
<th>Scale Factor</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>000000000011.110</td>
<td>30</td>
<td>2^-3 = 1/8</td>
<td>+3.7500</td>
</tr>
<tr>
<td>B</td>
<td>0000000101100.111</td>
<td>×51</td>
<td>2^-2 = 1/4</td>
<td>+12.7500</td>
</tr>
<tr>
<td>A × B</td>
<td>00000101111.11010</td>
<td>=1530</td>
<td>2^-3 = 1/32</td>
<td>+47.8125</td>
</tr>
</tbody>
</table>
Dividing Fixed-Point Real Numbers.

<table>
<thead>
<tr>
<th>Operand/Result</th>
<th>Bit Pattern</th>
<th>Int</th>
<th>Scale Factor</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0000101111.11010</td>
<td>1530</td>
<td>$2^{-5} = 1/32$</td>
<td>+47.8125</td>
</tr>
<tr>
<td>B</td>
<td>0000000001110.011</td>
<td>+115</td>
<td>$2^{-3} = 1/8$</td>
<td>+14.3750</td>
</tr>
<tr>
<td>A ÷ B</td>
<td>00000000000011.011</td>
<td>+13</td>
<td>$2^{-5+3} = 1/4$</td>
<td>+3.2500</td>
</tr>
</tbody>
</table>

Shifting Before Dividing Fixed-Point Real Numbers.

<table>
<thead>
<tr>
<th>Operand/Result</th>
<th>Bit Pattern</th>
<th>Int</th>
<th>Scale Factor</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^3 A$</td>
<td>00101111.11010000</td>
<td>12240</td>
<td>$2^{-8} = 1/256$</td>
<td>+47.8125</td>
</tr>
<tr>
<td>B</td>
<td>0000000001110.011</td>
<td>+115</td>
<td>$2^{-3} = 1/8$</td>
<td>+14.3750</td>
</tr>
<tr>
<td>$2^3 A ÷ B$</td>
<td>00000000000011.010</td>
<td>+106</td>
<td>$2^{-8+3} = 1/32$</td>
<td>+3.3125</td>
</tr>
</tbody>
</table>

16.16 Fixed-Point Format

Implied binary point

16.16 Fixed-Point Multiplication

Note: On a 32-bit CPU, you can simply use the regular integer multiply instruction, which produces a 64-bit product stored in a pair of 32-bit registers.
16.16 Fixed-Point Division

<table>
<thead>
<tr>
<th>Sign Extension</th>
<th>Whole Part</th>
<th>Fractional Part</th>
<th>Fill with 0's</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>63</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Dividend**

<table>
<thead>
<tr>
<th>Whole Part</th>
<th>Fractional Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>

**Divisor**

<table>
<thead>
<tr>
<th>Whole Part</th>
<th>Fractional Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>

**Quotient**

<table>
<thead>
<tr>
<th>Whole Part</th>
<th>Fractional Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>

"Brute-Force" 32.32 Format

Implied binary point

\[ 0 \cdots 00.00 \cdots 0 \]

32-bits 32-bits

This format uses lots of bits, but memory is relatively cheap and it supports both very large and very small numbers. If all variables use this same format (i.e., a common scale factor), programming is simplified. This is the strategy used in the Sony PlayStation.

32.32 Fixed-Point Multiplication

Problem: How do you compute the product of two 64-bit numbers using a 32-bit CPU?

<table>
<thead>
<tr>
<th>Whole Part</th>
<th>Fractional Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>0</td>
</tr>
</tbody>
</table>

**Multiplicand**

<table>
<thead>
<tr>
<th>Whole Part</th>
<th>Fractional Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>0</td>
</tr>
</tbody>
</table>

**Multiplier**

<table>
<thead>
<tr>
<th>Whole Part</th>
<th>Fractional Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>63</td>
</tr>
</tbody>
</table>

**Product**

<table>
<thead>
<tr>
<th>Discard</th>
<th>Whole Part</th>
<th>Fractional Part</th>
<th>Discard</th>
</tr>
</thead>
<tbody>
<tr>
<td>127</td>
<td>64</td>
<td>63</td>
<td>0</td>
</tr>
</tbody>
</table>

32.32 Fixed-Point Multiplication

Strategy:

1. Consider how to compute the 128-bit product of two 64-bit unsignede integers.
2. Modify that result to handle signed integers.
3. Note how discarding the 64 unused bits of the 128-bit product simplifies the computation.
32.32 Fixed-Point Multiplication

First consider a 64-bit unsigned number:

\[ A_u = 2^{63}A_{63} + 2^{62}A_{62} + \ldots + 2^0A_0 \]
\[ = 2^{63}A_{63} + (2^{62}A_{62} + \ldots + 2^0A_0) \]
\[ = 2^{63}A_{63} + A_{62..0} \]

where \( A_{62..0} = 2^{62}A_{62} + \ldots + 2^0A_0 \)

Thus the 128-bit product of two 64-bit unsigned operands would be:

\[ A_u B_u = (2^{63}A_{63} + A_{62..0})(2^{63}B_{63} + B_{62..0}) \]
\[ = 2^{126}A_{63}B_{63} + 2^{63}(A_{63}B_{62..0} + B_{63}A_{62..0}) + A_{62..0}B_{62..0} \]

32.32 Fixed-Point Multiplication

Now consider a 64-bit signed number:

\[ A_s = -2^{63}A_{63} + 2^{62}A_{62} + \ldots + 2^0A_0 \]
\[ = -2^{63}A_{63} + (2^{62}A_{62} + \ldots + 2^0A_0) \]
\[ = -2^{63}A_{63} + A_{62..0} \]

Thus the 128-bit product of two 64-bit signed operands would be:

\[ A_u B_s = (-2^{63}A_{63} + A_{62..0})(-2^{63}B_{63} + B_{62..0}) \]
\[ = 2^{126}A_{63}B_{63} - 2^{63}(A_{63}B_{62..0} + B_{63}A_{62..0}) + A_{62..0}B_{62..0} \]
\[ = A_u B_u - 2^{64}A_{63}B_{62..0} - 2^{64}B_{63}A_{62..0} \]
32.32 Fixed-Point Multiplication

What does this result mean?

\[ A_S B_S = A_u B_u - 2^{64} A_{63} B_{62}...0 - 2^{64} B_{63} A_{62}...0 \]

If \( A \) is negative, subtract \( B_{62}...0 \) from the most-significant half of \( A_u B_u \).

If \( B \) is negative, subtract \( A_{62}...0 \) from the most-significant half of \( A_u B_u \).

---

32.32 Fixed-Point Multiplication

\[ A_u B_u = (2^{32} A_{hi} + A_{lo})(2^{32} B_{hi} + B_{lo}) \]
\[ = 2^{64} A_{hi} B_{hi} + 2^{32}(A_{hi} B_{lo} + A_{lo} B_{hi}) + A_{lo} B_{lo} \]

---

Representation of Characters

<table>
<thead>
<tr>
<th>Representation</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>00100100</td>
<td>$</td>
</tr>
</tbody>
</table>
Character Constants in C

- To distinguish a character that is used as data from an identifier that consists of only one character long:
  - `x` is an identifier.
  - ‘x’ is a character constant.
- The value of ‘x’ is the ASCII code of the character x.

Character Escapes

- A way to represent characters that do not have a corresponding graphic symbol.

<table>
<thead>
<tr>
<th>Escape</th>
<th>Character</th>
<th>Character Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>\b</td>
<td>Backspace</td>
<td>‘\b’</td>
</tr>
<tr>
<td>\t</td>
<td>Horizontal Tab</td>
<td>‘\t’</td>
</tr>
<tr>
<td>\n</td>
<td>Linefeed</td>
<td>‘\n’</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage return</td>
<td>‘\r’</td>
</tr>
</tbody>
</table>

See Table 2-9 in the text for others.

Representation of Strings

C uses a terminating “NUL” byte of all zeros at the end of the string.
Pascal uses a prefix count at the beginning of the string.

48 65 6C 6C 6F 00
Hello

05 48 65 6C 6C 6F
Hello

String Constants in C

Character string

COEN 20 is “fun”!

C string constant

“COEN 20 is \"fun\"!”
Binary Coded Decimal (BCD)

Packed (2 digits per byte):

\[
\begin{array}{c|c}
0111 & 0011 \\
\hline
7 & 3 \\
\end{array}
\]

Unpacked (1 digit per byte):

\[
\begin{array}{c|c|c}
0000 & 0111 & 0000 & 0011 \\
\hline
7 & 3 \\
\end{array}
\]