CHAPTER 2
Data Representation

Kinds Of Data

- Text
  - ASCII Characters
  - Strings
- Other
  - Graphics
  - Images
  - Video
  - Audio

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

Numbers Are Different!

- Computers use binary 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):

$$(S_1 S_0 . S_{-1} S_{-2})_R =$$

$$S_1 \times R^1 + S_0 \times R^0 + S_{-1} \times R^{-1} + S_{-2} \times R^{-2}$$

Converting Radix R *to Decimal*

$$36.72_8 = 3 \times 8^1 + 6 \times 8^0 + 7 \times 8^{-1} + 2 \times 8^{-2}$$

$$= \quad 24 \quad + \quad 6 \quad + \; 0.875 \; + \; 0.03125$$

$$= \; 30.90625_{10}$$

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

$$= \; 1\times2^7 + 0\times2^6 + 1\times2^5 + 1\times2^4 + 0\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$$

$$= \; 128 + 32 + 16 + 4 + 1$$

$$= \; 181_{10}$$

Variation on Polynomial Evaluation for converting fractional values

Example: Convert 0.437_8 to decimal:

$= 4 \times 8^{-1} + 3 \times 8^{-2} + 7 \times 8^{-3}$

~~Multiple divisions~~

$= 4 \times 0.125 + 3 \times 0.015625 + 7 \times 0.001953125$

~~Adding long decimal fractions~~

Alternative approach:

$= (4 \times 8^2 + 3 \times 8^1 + 7 \times 8^0) / 8^3$

$= (4 \times 64 + 3 \times 8 + 7 \times 1) / 512$

$= 287 / 512 = 0.56054687510$

Problems: $N_R \rightarrow N_{10}$

110101_2 / 432_5

$.110101_2$ / $.432_5$

$3B7A_{16}$ / 13.4_8

$F07D_{16}$ / $.130_8$

10_3

$.10_3$

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.

Decimal to Binary Conversion
(Whole Part: Repeated Division)

$97 \div 2 \rightarrow$	quotient = 48,	remainder = 1 (LSB)
$48 \div 2 \rightarrow$	quotient = 24,	remainder = 0.
$24 \div 2 \rightarrow$	quotient = 12,	remainder = 0.
$12 \div 2 \rightarrow$	quotient = 6,	remainder = 0.
$6 \div 2 \rightarrow$	quotient = 3,	remainder = 0.
$3 \div 2 \rightarrow$	quotient = 1,	remainder = 1.
$1 \div 2 \rightarrow$	quotient = 0 (Stop)	remainder = 1 (MSB)

Result = $1\,1\,0\,0\,0\,0\,1_2$

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).

Decimal to Binary Conversion
(Fractional Part: Repeated Multiplication)

$.1 \times 2 \rightarrow$ 0.2 (fractional part = .2, whole part = 0)

$.2 \times 2 \rightarrow$ 0.4 (fractional part = .4, whole part = 0)

$.4 \times 2 \rightarrow$ 0.8 (fractional part = .8, whole part = 0)

$.8 \times 2 \rightarrow$ 1.6 (fractional part = .6, whole part = 1)

$.6 \times 2 \rightarrow$ 1.2 (fractional part = .2, whole part = 1)

Result = $.00011001100110011_2$.....

(How much should we keep?)

$.1_{10} = .000110011001100110011\ldots{}_2$

How much should we keep?

Mathematician's Answer:

Use the proper notation: $\mathbf{.0\overline{0011}}$

Scientist's Answer:

Preserve significant digits and round:

.1 → 1 part out of 10

3 binary digits = 1 out of 8 → need 4 → .0001

Round: 5th digit = 1, thus **.0010**

Engineer's Answer:

Depends on #bits in the variable (**8, 16, 32, 64**)

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?

Problems: $N_{10} \rightarrow N_R$

$27_{10} \rightarrow N_8$

$.27_{10} \rightarrow N_8$

$27_{10} \rightarrow N_{16}$

$.27_{10} \rightarrow N_{16}$

$1/3_{10} \rightarrow N_3$

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

Dec	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

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….

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!

Hex	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Hex	Binary
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Binary/Hex Conversions

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

0011	1010	0101	0110	.	1110	0010	1111	1000
3	A	5	6	.	E	2	F	8

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

Problems: $N_{R1} \rightarrow N_{R2}$, where $R1=R2^k$

$FACE_{16} \rightarrow N_2$

$1101011_2 \rightarrow N_{16}$

$11.01011_2 \rightarrow N_{16}$

$BEEF_{16} \rightarrow N_4$

$1101011_2 \rightarrow N_8$

$FEED_{16} \rightarrow N_8$

$11010.11_2 \rightarrow N_8$

$1846_9 \rightarrow N_3$

$10220_3 \rightarrow N_9$

$10.220_3 \rightarrow N_9$

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.
  - <u>Exceeding the range</u> is known as *overflow*.

Surprise! Rollover is <u>not</u> synonymous with overflow!

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

Two Interpretations

	unsigned		signed	
167_{10}	←	10100111_2	→	-89_{10}

- Signed vs. unsigned is a matter of interpretation; thus <u>a single bit pattern can represent two different values.</u>
- Allowing both interpretations is useful:

  Some data (e.g., count, age) can never be negative, and having a greater range is useful.

Why Not Sign+Magnitude?

$$+2_{10} = \quad 0010$$
$$+(-7_{10}) = \underline{+1111}$$
????

$111 > 010$:

$$\begin{array}{r} 111 \\ \underline{-\ 010} \\ 101 \end{array}$$

Use sign of 111:

$\underline{1}101$

$= -5_{10}$

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!

Why 2's Complement?

$$
\begin{array}{rr}
+2_{10} = & 0010 \\
+(-7_{10}) = & \underline{+1001} \\
& 1011 \\
& = -5_{10}
\end{array}
$$

Unsigned:

$$
\begin{array}{rr}
2_{10} = & 0010 \\
+(9_{10}) = & \underline{+1001} \\
& 1011 \\
& =11_{10}
\end{array}
$$

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.

Changing the Sign

Sign+Magnitude:

+5 = 0101

Change 1 bit

-5 = 1101

2's Complement:

+5 = 0101

Invert

1010

+1 *Increment*

-5 = 1011

Easier Hand Method

Step 2: Copy the inverse of the remaining bits.

+4 = 0100

-4 = 1100

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

Representation Width

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

Apply algorithm

Expand to 8-bits

Wrong: +25 = 11001 → 00111 → 00000111 = +7

Right: +25 = 11001 → 00011001 → 11100111 = -25

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

Apply algorithm

Converting 2's complement numbers to decimal – Approach #1

If MSB is 0, the number is positive.

→ convert as if it were unsigned.

If MSB is 1, the number is negative.

1. Apply 2's comp. alg. to find bit pattern of the corresponding positive magnitude
2. Convert the bit pattern to decimal.
3. Put a minus sign in front.

Converting 2's complement numbers to decimal – Approach #1

Example: $10110010_2 = ?_{10}$

1. $10110010_2 \rightarrow -01001110_2$
2. $01001110_2 = 64 + 8 + 4 + 2 = 78_{10}$
3. Answer: -78_{10}

Converting 2's complement numbers to decimal – Approach #2

Use polynomial evaluation, but make the contribution of the MSB be negative:

Example: $10110010_2 = ?_{10}$

$= -128 + 32 + 16 + 2 = -78_{10}$

2’s Complement Anomaly!

-128 = 1000 0000 (8 bits)

+128?

Step 1: Invert all bits ➔ 0111 1111

Step 2: Increment ➔ 1000 0000

What does this imply about using method #1 for converting a negative 2’s comp. number to decimal?

Result is negative! Why?

(Note: The right-to-left method yields same result)

Range of Unsigned Integers

Each of 'n' bits can have one of two values.

Total # of patterns of n bits = $\underbrace{2\times 2\times 2\times \ldots 2}_{\text{'n' 2's}}$

$= 2^n$

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

Range: 0 to 2^n-1 (2^n different values)

Problems: Unsigned Range

- Base 2, 6 digits
- Base 3, 6 digits
- Base 8, 6 digits
- Base 16, 6 digits

Range of Signed Binary Integers

- Half of the 2^n patterns will be used for positive values, and half for negative.
- Half is 2^{n-1}.
- Positive Range: 0 to $2^{n-1}-1$ (2^{n-1} patterns)
- Negative Range: -2^{n-1} to -1 (2^{n-1} patterns)
- 8-Bits (n = 8): -2^7 (-128) to $+2^7-1$ (+127)

Problem: 2’s Comp Range

- 5 bits
- 10 bits
- 16 bits

Decimal Addition Table

Carry-In = 0

+	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	0
2	2	3	4	5	6	7	8	9	0	1
3	3	4	5	6	7	8	9	0	1	2
4	4	5	6	7	8	9	0	1	2	3
5	5	6	7	8	9	0	1	2	3	4
6	6	7	8	9	0	1	2	3	4	5
7	7	8	9	0	1	2	3	4	5	6
8	8	9	0	1	2	3	4	5	6	7
9	9	0	1	2	3	4	5	6	7	8

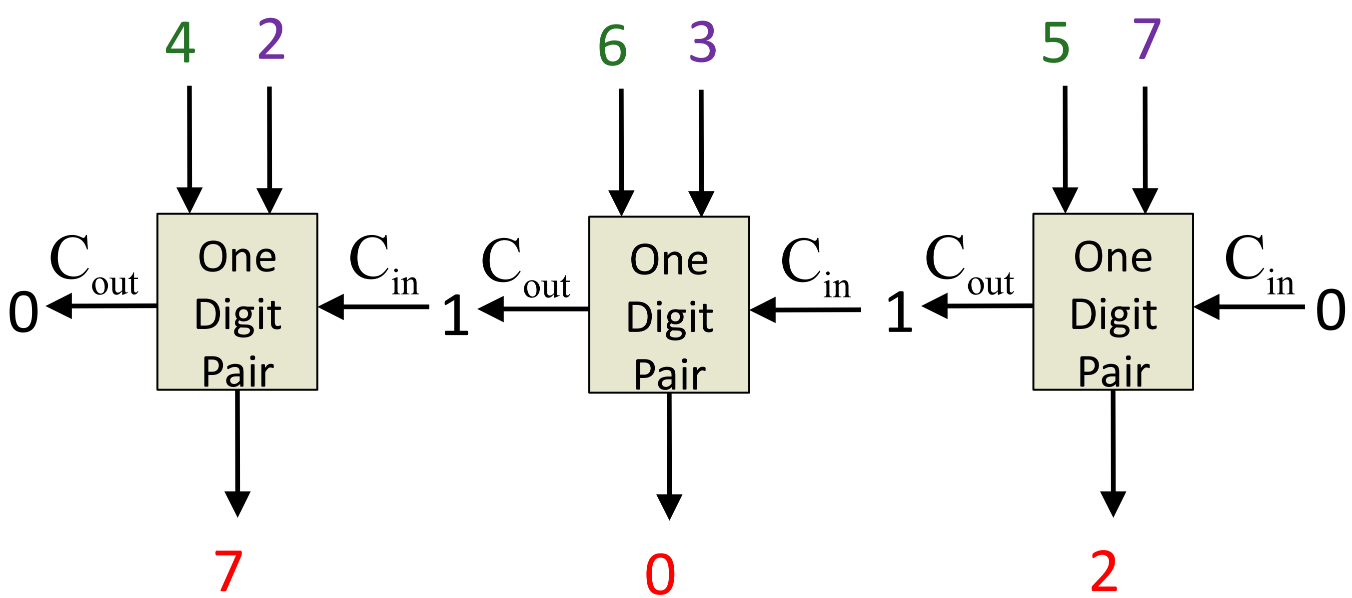
Carry-In = 1

+	0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9	0
1	2	3	4	5	6	7	8	9	0	1
2	3	4	5	6	7	8	9	0	1	2
3	4	5	6	7	8	9	0	1	2	3
4	5	6	7	8	9	0	1	2	3	4
5	6	7	8	9	0	1	2	3	4	5
6	7	8	9	0	1	2	3	4	5	6
7	8	9	0	1	2	3	4	5	6	7
8	9	0	1	2	3	4	5	6	7	8
9	0	1	2	3	4	5	6	7	8	9

Green: Carry-Out = 0
Yellow: Carry-Out = 1

Decimal Addition Using Table

```
0110  ← Carries
 465
+237
----
 702
```



Binary Addition & Carries

C_{in}	X	Y	n	C_{out}	S
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	1	0	1
0	1	1	2	1	0
1	0	0	1	0	1
1	0	1	2	1	0
1	1	0	2	1	0
1	1	1	3	1	1

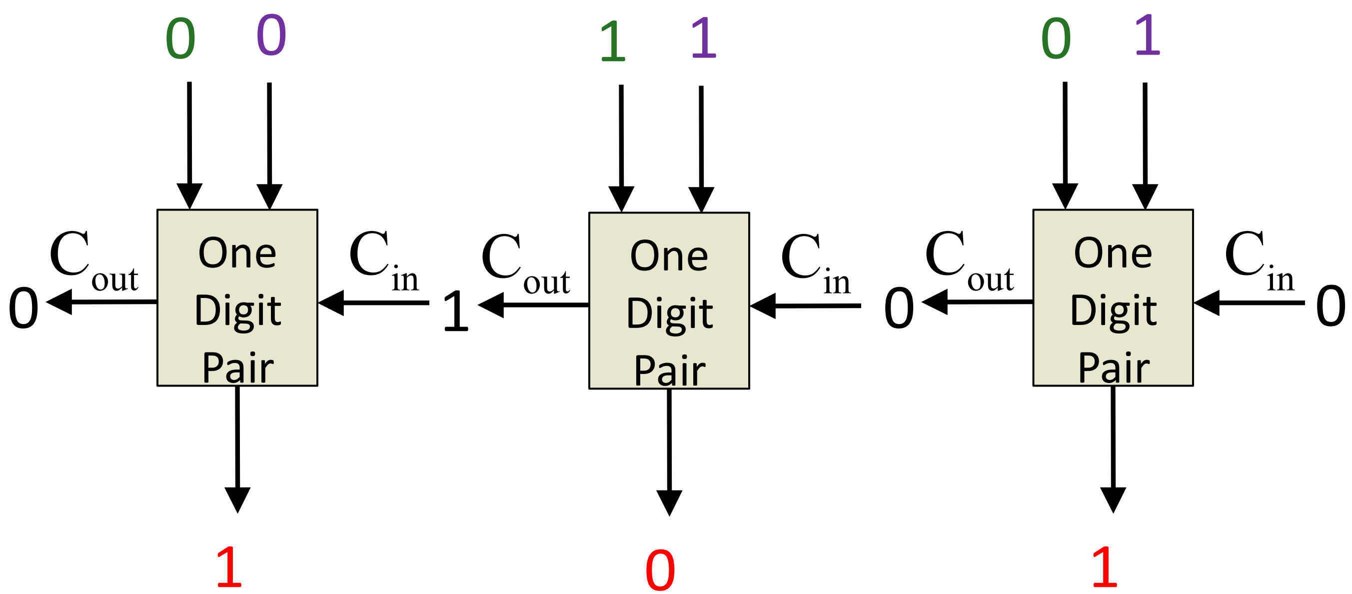
C_{in} = Carry-in
C_{out} = Carry-out
S = Sum digit

Column "n" is simply the sum of C_{in}, X and Y.

Columns C_{out} & S are simply the binary representation of n.

Binary Addition Using Table

```
0100  ← Carries
  010
 +011
 ----
  101
```



Binary Subtraction & Borrows

B_{in}	X	-Y	n	B_{out}	D
0	0	0	0	0	0
0	0	1	1	1	1
0	1	0	1	0	1
0	1	1	0	0	0
1	0	0	1	1	1
1	0	1	2	1	0
1	1	0	0	0	0
1	1	1	1	1	1

Bin = Borrow-In
Bout = Borrow-Out
D = Difference digit

Most people find this table to be less intuitive than the one for addition, and thus difficult to memorize.

Binary Subtraction & Borrows

$-B_{in}$	X	-Y	n	B_{out}	D
0	0	0	0	0	0
0	0	-1	-1	1	1
0	+1	0	+1	0	1
0	+1	-1	0	0	0
-1	0	0	-1	1	1
-1	0	-1	-2	1	0
-1	+1	0	0	0	0
-1	+1	-1	-1	1	1

Think of the contribution of X to the difference as positive, and that of Y and B_{in} as negative.

Column "n" is simply the sum of their contributions to the result.

Columns B_{out} & D are simply the 2's compl. representation of n.

Unsigned Overflow

1100	(12)
+0111	(7)
10011	

Lost

(Result limited by word size)

0011 (3) **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 ➔ (0)10000000

This is not rollover!

Signed Overflow

-120_{10}	➔	10001000_2
$\underline{-17}_{10}$		$+\underline{11101111}_2$
sum: -137_{10}		101110111_2
		01110111_2 (keep 8 bits)
		$(+119_{10})$ **wrong**

Note: $119 - 2^8 = 119 - 256 = -137$

Detecting Overflow

Unsigned:

Carry-out of MSB when incrementing or adding.

Borrow-out of MSB when decrementing or subtracting.

Signed (2’s complement):

Impossible when adding numbers of different signs.

Impossible when subtracting numbers of same sign.

Human Method: Sign of result seems incorrect.

Computer Method: Carries/Borrows in/out of MSB differ.

Problems: Overflow

Unsigned (4 bits)		*2's comp. (4 bits)*		
01010	11100	01010	11100	Addition Problems
0101	1011	0101	1011	
+0101	+0110	+0101	+0110	
1010	0001	1010	0001	

Unsigned (4 bits)		*2's comp. (4 bits)*		
11100	01000	11100	01000	Subtraction Problems
0100	1011	0100	1011	
−0110	−0101	−0110	−0101	
1110	0110	1110	0110	

Comparing Integers

Which is Greater: 1001 or 0011?

Unsigned:

Borrows	0 1 1 0 0	Borrow Out = 0
X	1 0 0 1	
Y	- 0 0 1 1	**1001 ≥ 0011**
X-Y	0 1 1 0	

Borrow Out	Relationship
0	**X ≥ Y**
1	X < Y

Signed (2's Complement):

Borrows	0 1 1 0 0	Overflow!
X	1 0 0 1	**1001 < 0011**
Y	- 0 0 1 1	
X-Y	0 1 1 0	Positive

Sign of X-Y	Overflow	True Sign	Relationship
Positive	No	Positive	X ≥ Y
Negative	Yes	Positive	X ≥ Y
Positive	Yes	Negative	**X < Y**
Negative	No	Negative	**X < Y**

Floating-Point Reals

31	30 23	22 0
$\pm$	Exponent	Fractional Bits of Significand

Three components:

Base is implied

$$\pm\ \text{significand} \times 2^{\text{exponent}}$$

Sign

A biased integer value

An <u>unsigned</u> fractional value

Single-precision Floating-point Representation

```
      S  Exp+127            Significand

 2.000 0 10000000 (1).00000000000000000000000
 1.000 0 01111111 (1).00000000000000000000000
 0.750 0 01111110 (1).10000000000000000000000
 0.500 0 01111110 (1).00000000000000000000000
 0.000 0 00000000 (0).00000000000000000000000
-0.500 1 01111110 (1).00000000000000000000000
-0.750 1 01111110 (1).10000000000000000000000
-1.000 1 01111111 (1).00000000000000000000000
-2.000 1 10000000 (1).00000000000000000000000
```

Representation of Characters

Representation

00100100

Interpretation

$

ASCII
Code

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.

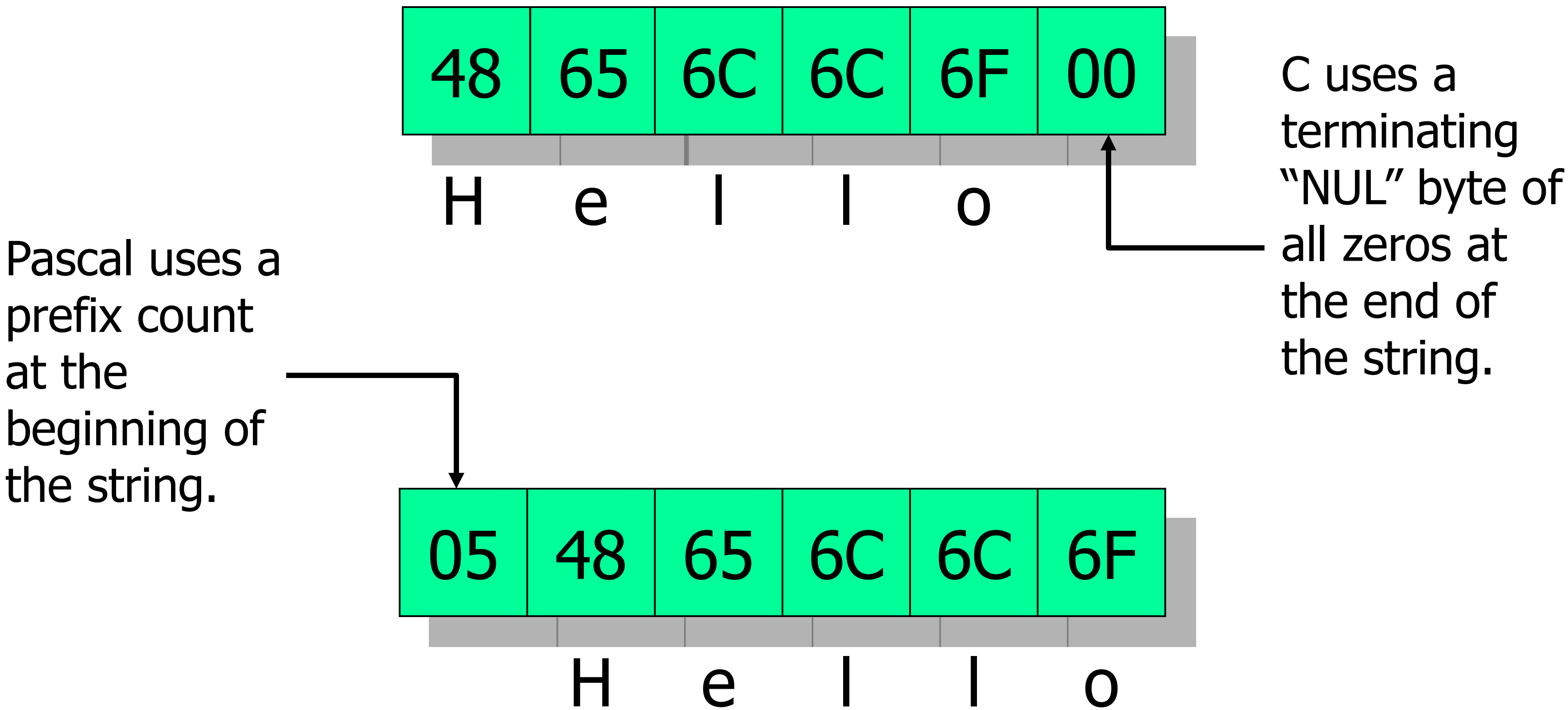
Character Escapes

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

Escape Character		Character Constant
\b	Backspace	'\b'
\t	Horizontal Tab	'\t'
\n	Linefeed	'\n'
\r	Carriage return	'\r'

See Table 2-9 in the text for others.

Representation of Strings



String Constants in C

Character string	C string constant
COEN 20 is “fun”!	“COEN 20 is \“fun\”!”

43	4F	45	4E	20	32	30	20	69	73	20	22	66	75	6E	22	21	00
C	O	E	N		2	0		i	s		“	f	u	n	”	!	\0

Binary Coded Decimal (BCD)

Packed (2 digits per byte):

0111	0011
7	3

Unpacked (1 digit per byte):

0000	0111	0000	0011
	7		3