**1** Number Systems

Decimal

1 5 7

100 10 1 weights

1100 + 510 + 71 = 157

*base-10* number system

Binary

Binary is a *base-2* number system.

0 1 0 1 1

16 8 4 2 1 weights

016 + 18 + 04+ 12 + 11 = 11 dec

8 + 2 + 1 = 11 decimal.

We call a sequence of eight bits a *byte*.

To *complement* a bit means to flip it.

Hexadecimal

*Hexadecimal* (or hex for short) is the *base-16* number system.

16 symbols: 0 to 9 and A, B, C, D, E, and F

2 C 5

256 16 1 weights (in decimal)

2256 + C16 + 51

2256+1216+51 = 512+192+5 = 709

Decimal Binary Hex

0 0000 0

1 0001 1

2 0010 2

3 0011 3

4 0100 4

5 0101 5

6 0110 6

7 0111 7

8 1000 8

9 1001 9

10 1010 A (or a)

11 1011 B (or b)

12 1100 C (or c)

13 1101 D (or d)

14 1110 E (or e)

15 1111 F (or f)

*Rule*: Adding a 0 on the right side of a positional whole number multiplies its value by its base.

Numbering Bits

bit 7 bit 0

10101010

bit 3

*most significant bit*

*least significant bit*

Adding Positional Numbers

decimal addition:

1 carries

157

+ 238

395

binary addition:

11 carries

0011

+ 0011

0110

hex addition:

1 carries

1B

+ 37

52

Representing Negative Numbers

0011 = +3 Sign-magnitude

+ 1011 = -3

1110 = -6

Carry out of leftmost column

1 111 carries

1111

+ 0001

0000

*Rule*: Adding 1 to a binary number with a fixed number of bits all of which are 1 results in all zeros.

0011 = +3

+ 1100 = +3 with each bit flipped

1111

1100 = +3 with each bit flipped

+ 0001 add 1

1101 Is this -3?

Is 1101 the representation of -3 that we want? Let’s add it to +3 to see if it gives zero:

1 111 carries

0011 = +3

+ 1101 Is this -3?

0000

*Rule*: To negate a binary number in the two’s complement system, flip its bits and add 1.

*Rule*: In the two’s complement system, all 1’s represents -1.

Signed and Unsigned Numbers

Unsigned Value Signed Value

0000 0 1000 -8

0001 1 1001 -7

0010 2 1010 -6

0011 3 1011 -5

0100 4 1100 -4

0101 5 1101 -3

0110 6 1110 -2

0111 7 1111 -1

1000 8 0000 0

1001 9 0001 1

1010 10 0010 2

1011 11 0011 3

1100 12 0100 4

1101 13 0101 5

1110 14 0110 6

1111 15 0111 7

sign bit

Range of Numbers

*n* Unsigned Signed

3 0 to 7 -4 to 3

4 0 to 15 -8 to 7

5 0 to 31 -16 to 15

6 0 to 63 -32 to 31

7 0 to 127 -64 to 63

8 0 to 255 -128 to 127

9 0 to 511 -256 to 255

10 0 to 1023 -512 to 511

12 0 to 4095 -2048 to 2047

16 0 to 65535 -32768 to 32767

*k* 0 to 2*k*-1 -2*k*-1 to 2*k*-1 -1

Powers of 2

*n* 2*n*

1 2

2 4

3 8

4 16

5 32

6 64

7 128

8 256

9 512

10 1,024 (aka 1K)

11 2,048 (aka 2K)

12 4,096 (aka 4K)

15 32,768 (aka 32K)

16 65,536 (aka 64K)

20 1,048,576 (aka 1M)

30 1,073,741,824 (aka 1G)

Converting Between Binary and Hex

1 1010 1110 0000 1100

1 A E 0 C

Converting Decimal to Binary

remainders

0 1

10) 1 5

10) 15 7

10) 157 Start here and work up

0 1

2) 1 0

2) 2 0

2) 4 1

2) 9 1

2) 19 1

2) 39 0

2) 78 1

2) 157 Start here and work up

Converting Fractions

.153

x 10

1 .530

whole parts x 10 multiply fractional

are the digits 5 .300 part of each product

x 10

3. 000

Converting Binary Fraction

.375

x 2

0 .750

x 2

1 .500

x 2

1 .000

.375 decimal = .011 binary

Infinite Length Equivalents

.1

x 2

0 .2

x 2

0 .4 .2 repeats

x 2

0 .8

x 2

1 .6

x 2

1 .2

Zero and Sign Extension

Extend 11111111 to 16 bits

0000000011111111

1111111111111111

*Rule*: Always sign extend signed numbers.

*Rule*: Always zero extend unsigned numbers.

Exclusive OR Gate

**

x

z

y

Exclusive OR Gate

x y z

0 0 0

0 1 1

1 0 1

1 1 0

difference detecting gate

control

**

equal to data if control = 0

data equal to data flipped if control = 1

Control Data Output of exclusive OR

Data unaffected when control = 0

0 0 0

0 1 1

1 0 1

Data flipped when control = 1

1 1 0

*Rule*: If the control line of an exclusive OR is 0, the data passes through gate unchanged. If the control line is 1, the data is complemented.

NOR Gate



1 when all inputs = 0

x y z

x 0 0 1

z

y 0 1 0

1 0 0

1 1 0

zero detecting gate

Addition and Subtraction

sum out

carry out

carry in

Full

Adder

top bit in bottom bit in

Three-bit Adder/Subtractor Circuit

0: addition

1: subtraction

Full

Adder

Full

Adder

Full

Adder

** top number 0: bottom number passes through

1: bottom number is flipped

bottom number

Signed Overflow

+32767

.

P .

P+N must be in .

this range 0 Range of 16-bit signed numbers

.

N .

.

-32768

*Rule*: If two signed numbers *with different signs* are added or two signed numbers *with the same sign* are subtracted, overflow cannot occur.

Overflow scenarios:

Adding two non-negative numbers Adding two negative numbers

(a) (b)

0 1

+ 0 + 1

1 0

Sign bit should be 1

Sign bit should be 0

Here are all the possible scenarios for no overflow:

(c) (d) (e) (f)

0 1 0 0

+ 0 + 1 + 1 + 1

0 1 0 1

Leftmost full adder

Carry-out line Carry-in line

** Exclusive OR gate

0: no signed overflow

1: signed overflow

v

*Rule*: In an addition or subtraction of signed numbers, overflow has occurred if the carry into the leftmost position does not match the carry out.

*Rule*: If overflow occurs during the addition of two signed numbers, the sign bit of the computed result is wrong.

Unsigned Overflow

1 111111111111111 carries

Carry out of leftmost position

1111111111111111 = 65535

+ 0000000000000001 = 1

0000000000000000 = 0

*Rule*: In an addition of unsigned numbers, a carry out of the leftmost position indicates overflow. In a subtraction, no carry out of the leftmost position (or a borrow in if the borrow technique is used) indicates overflow.

Three-bit Adder/Subtractor Circuit with n, z, c, and v Flags

 0: sum/diff 0

0: sum/diff 0 1: sum/diff = 0

1: sum/diff < 0

z

n

0: addition

1: subtraction

Full

Adder

Full

Adder

Full

Adder

** top number 0: bottom number passes through

1: bottom number is flipped

bottom number

*a b*

**

0: no unsigned overflow 0: no signed overflow

1: unsigned overflow 1: signed overflow

v

c

Floating Point

hidden bit exponent

1.10101 x 23 (which equals 13.25 decimal)

fractional

part

* One bit: the sign bit (0 for zero or positive, 1 for negative)
* Eight bits: the exponent *to which 127 is added*
* 23 bits: the fractional part, zero extended on the right

10101000000000000000000.

Thus, the 32-bit floating-point representation of 13.25 decimal is

sign

0 10000010 10101000000000000000000

adjusted fractional part

exponent

Special Representations

Exponent field Fractional part Value

all 0’s all 0’s 0.0

all 0’s nonzero denormal number (exponent = -126, hidden bit 0)

all 1’s all 0’s infinity

all 1’s nonzero NaN (Not a Number)

Denormal Numbers

Smallest positive magnitude that can be represented by a regular floating-point number is

0 00000001 00000000000000000000000

which equals

1.00000000000000000000000 x 2-126

hidden bit

0 00000000 11111111111111111111111 =

0.11111111111111111111111 x 2-126

which is just under 2-126, down to

0 00000000 00000000000000000000001 =

0.00000000000000000000001 x 2-126

negative denormal numbers positive denormal numbers

| | |

0

negative regular number -2-149 +2-149 positive regular number

with smallest magnitude (-2-126) with smallest magnitude (+2-126)

Floating-point numbers (float and double in C) are *only an approximation of the set of real numbers in mathematics*

1 // ex0101.c Finite precision of floating point numbers

2 #include <stdio.h>

3 int main()

4 {

5 double sum = 0.0;

6 int i;

7 for (i = 1; i <= 10; i++)

8 sum = sum + 0.1;

9 printf("%.17f\n", sum); // sum is not 1.0

10 return 0;

11 }

1 // ex0102.c Floating-point errors

2 #include <stdio.h>

3 int main()

4 {

5 double x = 1E70; // 1070

6 double y = 1E5; // 105

7 if (x == x + y)  
 8 printf("y must be zero\n"); // displayed but y is not zero  
 9 else  
10 printf("y must be nonzero\n");

11 return 0;

12 }

*Observation*: Numbers can act like zero when added to numbers with significantly larger magnitudes.

1 // ex0103.c Lack of associativity in floating-point computations

2 #include <stdio.h>

3 int main(void)

4 {

5 double sum = 0.0;

6 int i;

7 for (i = 1; i <= 100000000; i++)

8 sum = sum + 1.0/i;

9 printf("%.17f\n", sum);

10

11 sum = 0.0;

12 for (i = 100000000; i >= 1; i--) // better for loop

13 sum = sum + 1.0/i;

14 printf("%.17f\n", sum);

15 return 0;

16 }

*Observation*: Addition of floating-point numbers is not associative (i.e., order of evaluation matters).