Binary Representations and Arithmetic

9-1-2016

Common number systems.

- Base 10: decimal
- Base 2: binary

- Base 16: hexadecimal (memory addresses)
- Base 8: octal (obsolete computer systems)
- Base 64 (email attachments, ssh keys)

Hexadecimal: Base 16

- Indicated by prefacing number with 0x

A number, written as the sequence of digits

$d_n d_{n-1} \ldots d_2 d_1 d_0$ where d is in {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}, represents the value

$$d_n * 16^n + d_{n-1} * 16^{n-1} + \ldots + d_2 * 16^2 + d_1 * 16^1 + d_0 * 16^0$$

What is the value of 0x1B7 in decimal?

$16^2 = 256$

A. 397

B. 409

C. 419

D. 437

E. 439

Each hex digit is a “nibble”

Hex digit:16 values, $2^4 = 16$ -> 4 bits / digit

0x1B7

256s digit: 1

16s digit: B (decimal 11)

1s digit: 7

In binary:	0001	1011	0111
	1	B	7

Converting hex and binary

- A group of four binary digits maps to one hex digit.

0x 48C1 = 0b 0100 1000 1100 0001

4 → 0100

8 → 1000

C → 1100 (12)

1 → 0001

What is 0b101100111011 in hex?

a) 0xb3b

b) 0x59d

c) 0xc5c

d) 0x37b

e) 0x5473

Converting among 2^x bases

Amounts to re-grouping digits.

- Binary → octal: group sets of three digits
- Octal → hex: group pairs of digits
- Hex → base64: group sets of four digits

- Split digits into groups to reverse the conversion.

Converting among arbitrary bases

- The division-and-mod method always works.
  - Requires division and mod in the start-base
- The subtract-base-powers method kind of works.
  - Must modify to subtract *multiples* of powers of the base.
  - Requires multiplication, powers and subtraction in the start-base.
- The sum-up-digits method always works.
  - Requires multiplication, powers and addition in the end-base.

- We're used to thinking in base 10, so it often helps to use base 10 as an intermediate step.

I will only make you do conversions among bases 2, 10, and 16.

Unsigned Addition (4-bit)

- Addition works like grade school addition:

```
    1
    0110       6
+   0100     + 4
--------    ----
    1010      10
```

Four bits give us range: 0 - 15

Unsigned Addition (4-bit)

- Addition works like grade school addition:

```
    1
    0110       6        1100     12
  + 0100     + 4      + 1010    +10
  ------     ---      ------    ---
    1010      10      1 0110      6
                      ^carry out
```

Overflow!

Four bits give us range: 0 - 15

What’s the sum?

```
 01100111
+10001110
```

a) 11110101, carry out = 1

b) 11110011, carry out = 0

c) 11110101, carry out = 0

d) 00110101, carry out = 1

e) 01100101, carry out = 0

So far: Unsigned Integers

- With N bits, can represent values: 0 to 2^n-1

- We can always add 0's to the front of a number without changing it:

10110= <u>0</u>10110 = <u>00</u>010110 = <u>0000</u>010110

- `1 byte:  char,` <u>`unsigned char`</u>
- `2 bytes: short,` <u>`unsigned short`</u>
- `4 bytes: int,` <u>`unsigned int`</u>`, float`
- `8 bytes: long long,` <u>`unsigned long long`</u>`, double`
- `4 or 8 bytes: long,` <u>`unsigned long`</u>

Representing Signed Values

- One option (used for floats, <u>NOT integers</u>)
  - Let the first bit represent the sign
  - 0 means positive
  - 1 means negative

- For example:
  - 0101 -> 5
  - 1101 -> -5

- Problem with this scheme?

Two's Complement

The Encoding comes from Definition of the 2's complement of a number:

2's complement of an N bit number, x, is its complement with respect to 2^N

Can use this to find the bit encoding, y, for the negation of x:

$$\text{For N bits, } y = 2^N - x$$

4 bit examples:

X	-X	2^4 - X
0000	0000	10000 – 0000 = 0000 (only 4 bits)
0001	1111	10000 – 0001 = 1111
0010	1110	10000 – 0010 = 1110
0011	1101	10000 – 0011 = 1101

Two's Complement

- Only one value for zero
- With N bits, can represent the range:
  - -2^{N-1} to $2^{N-1} - 1$
- First bit still designates positive (0) /negative (1)

- Negating a value is slightly more complicated:

  1 = 00000001, -1 = 11111111

From now on, unless we explicitly say otherwise, we'll assume all integers are stored using two's complement! This is the standard!

Two’s Compliment

- Each two’s compliment number is now:

$-2^{n-1}*d_{n-1} + 2^{n-2}*d_{n-2} + \ldots + 2^{1}*d_{1} + 2^{0}*d_{0}$

Note the negative sign on just the first digit. This is why first digit tells us negative vs. positive.

2's Complement to Decimal

High order bit is the sign bit, otherwise just like unsigned conversion. 4-bit examples:

$$0110:\ 0*-2^3 + 1*2^2 + 1*2^1 + 0*2^0$$

$$0 + 4 + 2 + 0 = 6$$

$$1110:\ 1*-2^3 + 1*2^2 + 1*2^1 + 0*2^0$$

$$-8 + 4 + 2 + 0 = -2$$

Try: 1010

1111

What is 11001 in decimal?

- Each two's compliment number is now:

$-2^{n-1}*d_{n-1} + 2^{n-2}*d_{n-2} + \ldots + 2^{1}*d_{1} + 2^{0}*d_{0}$

A. -2

B. -7

C. -9

D. -25

Range of binary values

Smallest unsigned value:

$00000000 = 0$

Largest unsigned value:

$11111111 = 2^N – 1$

Smallest 2's complement value:

$10000000 = -2^{N-1}$

Largest 2's complement value:

$01111111 = 2^{N-1} - 1$

Addition & Subtraction

- Addition is the same as for unsigned
  - One exception: different rules for overflow

  \+ Can use the same hardware for both

- Subtraction is the same operation as addition
  - Just need to negate the second operand…

- 6 - 7 = 6 + (-7)

How to negate in 2’s complement

1. Flip the bits (0’s become 1’s, 1’s become 0’s)
2. Add 1

Example: -1 * 00101110

1. Flip bits: 11010001
2. Add 1: + 00000001 = 11010010

Subtraction Hardware

<u>Negate and add 1 to second operand:</u>

Can use the same circuit for add and subtract:

6 - 7 == 6 + ~7 + 1

```
input 1 -------------------------------->  ADD CIRCUIT
input 2 --> possible bit flipper -->       ADD CIRCUIT ---> result
            possible +1 input-------->
```

Let's call this possible +1 input: "Carry in"

(0: on add, 1: on subtract)

Examples:

4 bit signed values (a-b is a + ~b + 1):

subtraction: flip bits and add 1

```
3 -  6 =   0011
            1001      (6: 0110  ~6: 1001)
        +  ____1
```

Addition: don’t flip bits or add 1

```
3 + -6 =   0011
        +  1010
           ----
```

Convert and subtract

Perform the subtraction 12 – 19 in 6-bit binary.

```
      12 = 001100
      19 = 010011
     ~19 = 101100
     -19 = 101101
12 - 19 = 111001 = -7
```

Bits and Bytes

- Bit: a 0 or 1 value (binary)
  - HW represents as two different voltages
    - 1: the presence of voltage (high voltage)
    - 0: the absence of voltage (low voltage)

- Byte: 8 bits, the smallest addressable unit

  Memory: 01010101 10101010 00001111 …

  (address) [0] [1] [2] …

- Other names:
  - 4 bits: Nibble
  - "Word": Depends on system, often 4 bytes

How many unique values can we represent with 9 bits?

- One bit: two values (0 or 1)
- Two bits: four values (00, 01, 10, or 11)
- Three bits: eight values (000, 001, …, 110, 111)

A. 18
B. 81
C. 256
D. 512
E. Some other number of values.

How many values?

1 bit: 0 1

2 bits: 0 0 0 1 1 0 1 1

3 bits: 0 0 0 0 0 1 0 1 0 0 1 1 1 0 0 1 0 1 1 1 0 1 1 1

4 bits:

0 0 0 0 0 0 0 1 0 0 1 0 0 0 1 1 16 values
0 1 0 0 0 1 0 1 0 1 1 0 0 1 1 1

1 0 0 0 1 0 0 1 1 0 1 0 1 0 1 1
1 1 0 0 1 1 0 1 1 1 1 0 1 1 1 1

N bits: 2^N values

Determining sizes of C types (on my laptop)

```
#include <stdio.h>

int main(){
  char c;
  short s;
  int i;
  long l;
  long long ll;
  float f;
  double d;

  printf("size of char:      %lu\n", sizeof(c));
  printf("size of short:     %lu\n", sizeof(s));
  printf("size of int:       %lu\n", sizeof(i));
  printf("size of long:      %lu\n", sizeof(l));
  printf("size of long long: %lu\n", sizeof(ll));
  printf("size of float:     %lu\n", sizeof(f));
  printf("size of double:    %lu\n", sizeof(d));
}
```

Written homework #1

- Will be released tomorrow.
- Will be due by 4:00pm next Friday.
- Topics:
    - binary/hex conversions
    - binary arithmetic