Binary Numbers

Counting the way computers do

topics…

- Positional numbering
- Fixed-size numbers and storage
- Addition
- Conversions between number bases
- binary → decimal
  - any base → decimal
- Binary and Hexadecimal
- Decimal to Binary
- Decimal to Hexadecimal
- Decimal to any other base
- Any base to any base

Positional numbering systems

- *Base-10* positional number system:
  - known as the *Decimal* number system
  - 10 possible digits ---
    { **0**, **1**, **2**, **3**, **4**, **5**, **6**, **7**, **8**, **9** }
  - Digit values depend on their positions in a number
    - Each position is the coefficient of a power of 10:

$$\mathbf{61078} = \mathbf{6}\bullet 10^{\mathbf{4}} + \mathbf{1}\bullet 10^{\mathbf{3}} + \mathbf{0}\bullet 10^{\mathbf{2}} + \mathbf{7}\bullet 10^{\mathbf{1}} + \mathbf{8}\bullet 10^{\mathbf{0}}$$

$$= 60000 + 1000 + 00 + 70 + 8$$

Positional numbering systems

- Any base *b* is possible
  - *b* digits are required --- { **0**, ..., ***b*-1** }
    - letters are often used for digits past "9"
  - *n*-digit numbers as coefficients in a polynomial:

$$d_{n-1}\, d_{n-2} \ldots d_1\, d_0 = d_{n-1}\bullet b^{n-1} + d_{n-2}\bullet b^{n-2} + \ldots + d_1\bullet b + d_0$$

(each digit d_k is in the range { 0, ..., b-1 })

Other bases: examples

- base: b = 8 ("Octal")
  - *8* digits are required --- { **0**,**1**,**2**,**3**,**4**,**5**,**6**,**7** }
  - *example:*

    $73042_8 = 7 \cdot 8^4 + 3 \cdot 8^3 + 0 \cdot 8^2 + 4 \cdot 8 + 2$

    $= 30242_{10}$
- base: b = 13
  - *13* digits --- { **0**,**1**,**2**,**3**,**4**,**5**,**6**,**7**,**8**,**9**,**a**,**b**,**c** }
  - *example:*

    $23a6_{13} = 2 \cdot 13^3 + 3 \cdot 13^2 + 10 \cdot 13 + 6$

    $= 5037_{10}$

Other bases: important bases

- base: b = 2 ("Binary")
  - *Only 2* digits are required --- { **0**, **1** }
  - *example:*

    $01101_2 = 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2 + 1$

    $= 13_{10}$
- base: b = 16 ("Hexadecimal")
  - *16* digits --- { **0**,**1**,**2**,**3**,**4**,**5**,**6**,**7**,**8**,**9**,**a**,**b**,**c**,**d**,**e**,**f** }
  - 1 digit ↔ 4 binary digits
  - *example:*

    $23a6_{16} = 2 \cdot 16^3 + 3 \cdot 16^2 + 10 \cdot 16 + 6$

    $= 9126_{10}$

C Notation

- C accepts integer values written in:
  - base 10 / decimal *(default)*
  - base 8 / octal
  - base 16 / hexadecimal
- Octal numbers: start with leading "0"
  - Examples: **`0734`** is an octal number
    `0835` is an *illegal* octal number
- Hexadecimal numbers: leading "0x"
  - Examples: **`0x734`** is in hexadecimal
    `0x23feed98` is in hexadecimal

GCC Extension to the C Standard

- Some versions of the "gcc" compiler accept "0b" as a prefix:
  - **`0b0101100`** equals **`0x2c`** equals **`054`** equals **`44`**
- NOT portable: other compilers may not recognize this

Storing values in memory

- Computer memories are collections of *switches*
  - Each switch has two positions, *Off* and *On*
  - Each memory word consists of up to 128 switches
  - One switch can represent 0 as *Off*, or 1 as *On*
- Binary digits zero and one are known as **bits**
  - "bit" = contraction of "binary digit"
  - Each switch stores 1 bit
- Eight bits taken together form a **byte**
  - Four bits are known as a "nybble"
    - (Wikipedia misspells this....)
- Memories are measured in multiples of bytes

Storing numbers in memory

- Numbers occupy one or more bytes
  - Stored using binary notation

# of bits	# of bytes	C type	ARMv8 name
8	1	char	byte (*or* octet)
16	2	short	hword (*"half word"*)
32	4	int, float	word
64	8	long int, double	dword (*"double word"*)
128	16	(long long int, long double)	qword (*"quad word"*)

Sizes of C Types

```
/* Sizes of various types on 64-bit Linux w/ gcc v4.8.4
*/
#include<stdio.h>
int main(void)
{
   __int128 foo = -100;

   printf("sizeof  char: %lu\n", sizeof(char));
   printf("sizeof short: %lu\n", sizeof(short));
   printf("sizeof   int: %lu\n", sizeof(int));
   printf("sizeof  long: %lu\n", sizeof(long));
   printf("sizeof long long: %lu\n", sizeof(long long));

   printf("sizeof __int128: %lu\n", sizeof(__int128));
   printf("value of __int128: %Li\n", foo);
   printf("\n");

   printf("sizeof  float: %lu\n", sizeof(float));
   printf("sizeof double: %lu\n", sizeof(double));
   printf("sizeof long double: %lu\n", sizeof(long double));
   return 0;
}
```

sizeof() reports sizes in bytes – multiply by 8 for bits

```
611] ./typesizes
sizeof  char: 1
sizeof short: 2
sizeof   int: 4
sizeof  long: 8
sizeof long long: 8
sizeof __int128: 16
value of __int128: -100

sizeof  float: 4
sizeof double: 8
sizeof long double: 16
```

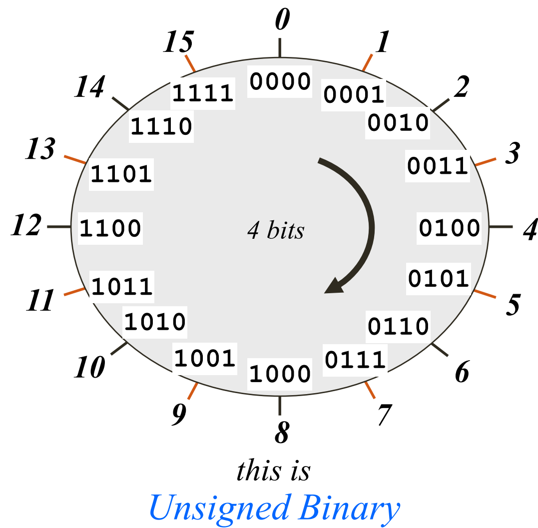
Range of possible numbers

- Fixed number of bits per number means a finite range of numbers
  - Smallest possible number –
    - unsigned: 0
    - signed: $-2^{(number_of_bits - 1)}$
  - Largest possible number –
    - unsigned: $2^{(number_of_bits)} - 1$
    - signed: $2^{(number_of_bits - 1)} - 1$
- What happens when you add 1 to the largest number?
  - Or subtract 1 from the smallest number?

Modulo counting — "Odometer numbers"

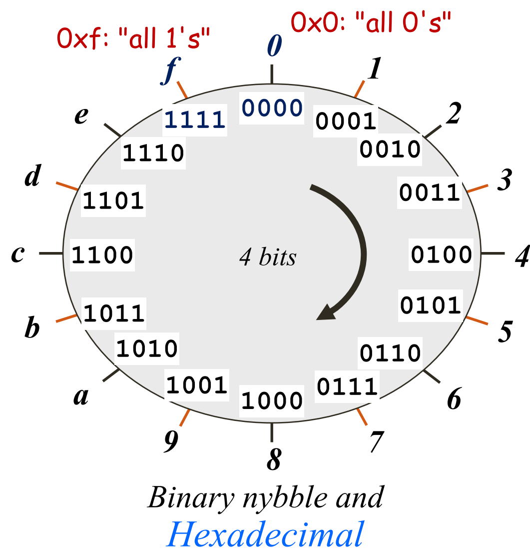
4-bit example

- Numbers start with `0000`*b* = ***0***
- Numbers "wrap around": after `1111`*b* = ***15***, back to `0000`*b* = ***0*** again
- 2^4 = 16 distinct values: {***0***, ***1***, …, ***15***}



Binary, hexadecimal, and nybbles

- Exactly 2^4 possible combinations of 4 bits…
- …exactly match the 16 hexadecimal digits
- Hex is used as "shorthand" for binary



Ranges of unsigned numbers

type / name	# of bits	decimal range	hexadecimal range
nybble	4	0 .. 15	0x0 .. 0xf
unsigned char	8	0 .. 255	0x0 .. 0xff
unsigned	16	0 .. 65,535	0x0 .. 0xffff
unsigned long	32	0 .. 4,294,967,296	0x0 .. 0xffff ffff
(unsigned long long)	64	0 .. 18,446,744,073, 709,551,616	0x0 .. 0xffff ffff ffff ffff

base conversions

Changing between decimal, binary, and hexadecimal representations

topics...

- binary → decimal
  - any base → decimal
- Binary and Hexadecimal

- Decimal → Binary
- Decimal → Hexadecimal
- Decimal → any other base

- Any base → any base

Conversion Possibilities

- Binary → Decimal
  - Straightforward
- *<any base>* → Decimal
  - Straightforward
- Hexadecimal → Binary, Binary → Hexadecimal
  - Should be trivial/automatic!
- Hexadecimal → Decimal
  - Straightforward, *or* use two steps: hex → binary → decimal

Binary to Decimal conversion

- Binary numbers use *positional notation*
- Each bit is another power of 2
  - $bit_{n-1}\ bit_{n-2} \ldots bit_1\ bit_0 =$

    $bit_{n-1} \bullet 2^{n-1} + bit_{n-2} \bullet 2^{n-2} + \ldots + bit_1 \bullet 2 + bit_0$

- Multiply each bit bit_k by 2^k, and add up the results
  - Choices:
    Start at the right (bit_0) and work to the left,
    or start at the left (bit_{n-1}) and work to the right?

Binary → Decimal

- right-to-left (least- to most-significant):
  - Set sum = 0
  - Set power = 1
  - For each successive bit:
    - » Multiply bit by current power, then add to the sum
    - » Multiply power by 2 to form next-higher power
- left-to-right (most- to least-significant):
  - Set sum = 0
  - For each successive bit:
    - » Multiply sum by 2
    - » Add the new bit's value

exercise: binary-to-decimal

- Write a function that accepts a **string** which represents a binary number
  - e.g., "0010110001011010"
  - How long could the string be? How many bits in the decimal value?
- The function converts this string to a decimal value, and returns it.
- Write a driver program to use the function.

Generalizing to other bases

- Using positional notation with any number base:
- Each digit is another power of the base
  - $d_{n\text{-}1}\, d_{n\text{-}2} \ldots d_1\, d_0 \;=$

    $d_{n\text{-}1} \bullet \text{base}^{n\text{-}1} + d_{n\text{-}2} \bullet \text{base}^{n\text{-}2} + \ldots + d_1 \bullet \text{base} + d_0$
- Multiply each digit d_k by base^k, and add up the results
  - Same choices:
    Start at the right (d_0) and work to the left,
    or start at the left ($d_{n\text{-}1}$) and work to the right?

<any base> → Decimal

- right-to-left (least- to most-significant):
  - Set sum = 0
  - Set power = 1
  - For each successive digit, starting on the right:
    - Multiply digit by current power, then add to the sum
    - Multiply power by the base, to form next-higher power
- left-to-right (most- to least-significant):
  - Set sum = 0
  - For each successive digit, starting on the left:
    - Multiply sum by the base
    - Add the digit's value

exercise: any base-to-decimal

- Write a function that accepts:
  - a (base-10) number which is the base,
  - a **string** which represents a number in that base
- The function converts this string to a decimal value and returns it.
  - Next slide shows a clumsy but effective way to handle each digit...

```c
unsigned base2dec(char *other, /*unsigned size,*/ unsigned base)
{
   unsigned sum = 0;
   char *p;
   p = other;

   while(*p != '\0') {
      sum *= base;
      switch(*(p++)) {
         case '0': break;
         case '1': sum += 1; break;
         case '2': sum += 2; break;
         case '3': sum += 3; break;
         case '4': sum += 4; break;
         case '5': sum += 5; break;
         case '6': sum += 6; break;
         case '7': sum += 7; break;
         case '8': sum += 8; break;
         case '9': sum += 9; break;
```

Example routine

This is straightforward, and not very elegant.

Exploit the power of ASCII…

Converting ASCII digits to numeric values

- In general, the "digits" are represented by ASCII characters.
  - '0' .. '9', 'a'..'f', etc.
- if-else or switch() statement: most general solution
  - if (ch == '3') value = 3; // etc.
  - cumbersome for larger bases
    - base-200 ain't gonna be pretty…
  - permits multi-character representations of remainders
    - necessary for base-257 and higher

eek

- Direct conversion: subtract 0x30 (ASCII '0') from the ASCII representation
  - only works for characters '0' .. '9' ?

```
/* in-class version of a base-to-decimal
   conversion routine.
   Implemented up to base-36.
   2013-10-03 -bob,mon.
*/

unsigned base2dec(char *other, /*unsigned size,*/ unsigned base)
{
   unsigned sum = 0;
   char *p;
   p = other;

   while(*p != '\0') {
      sum *= base;

      if (*p >= '0' && *p <= '9')
         sum += *p - '0';

      else if (*p >= 'A' && *p <= 'Z')
         sum += 10 + *p - 'A';
```

Example routine, reworked

- Six tests cover all 36 digits, instead of 36 separate cases.
- Code is both shorter and faster.

Conversion Possibilities

- Binary → Decimal
  - Straightforward
- *<any base>* → Decimal
  - Straightforward
- Hexadecimal → Binary, Binary → Hexadecimal
  - Should be trivial/automatic!
- Hexadecimal → Decimal
  - Straightforward, *or* use two steps: hex → binary → decimal

Binary and Hexadecimal

Converting between Binary and Hexadecimal

- This is just a "lookup"
- Binary→Hexadecimal
  – replace every four bits with the corresponding hex digit
  - `0110 1010 1001 0101` →
    `6 a 9 5`, or 0x6a95 (C notation)
- Hexadecimal→Binary
  – replace each hex digit with the corresponding four bits
  - `0xf83b`, or `f 8 3 b`
    → `1111 1000 0011 1011`

Memorize!

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

binary	hexadecimal
1000	8
1001	9
1010	a *or* A
1011	b *or* B
1100	c *or* C
1101	d *or* D
1110	e *or* E
1111	f *or* F

A historical note

binary	octal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

- Octal numbers are very similar to hexadecimal
  - for 3 bits instead of 4
- Some early computers used 12-bit "words", split into 4 octal digits
- Today's machines use power-of-16 "words"
  - Hexadecimal is a better match
- Octal is still seen sometimes

Decimal conversions

More Possibilities

- Decimal → Binary
- Decimal → Hexadecimal
- Decimal → *<any other base>*
- *<any base>* → *<any other base>*
  - Most straightforward is to use two steps:

    <any base> → Decimal → *<any other base>*
- Oh, yeah… Binary → Octal, Octal → Binary
  - (see hex→binary, binary→hex)

Decimal → Binary, for integers

- Generate answer left-to-right (most- to least-significant digits):
  - first step: divide by the largest possible power of 2…
  - keep the (integer) quotients
- Generate answer right-to-left – doesn't require knowing "where to start" :
  - Repeatedly divide by 2
  - keep the (integer) remainders
  - This just a particular case of decimal → *<any base>*

Example: 35_{10} → $?_2$

- 35 ÷ 2 → 17, r 1
- 17 ÷ 2 → 8, r 1
- 8 ÷ 2 → 4, r 0
- 4 ÷ 2 → 2, r 0
- 2 ÷ 2 → 1, r 0
- 1 ÷ 2 → **0**, r 1

…and we're done

- 35_{10} → $10\ 0011_2$

a Coding Issue

- The "right-to-left" algorithm generates bits starting with the *least significant* (rightmost)
- If a software routine stores these bits in an array, they will look "backward" when printed
- Get around this with a routine that reverses strings
  - C doesn't have a library function for this

A coding solution

- This routine reverses a string *in place*
  - That is, it changes its argument
  - This is called a *side effect*, although it's the whole point of this routine

```
#include <string.h>     // for strlen()

void reverse_in_place(char *str)
{
   unsigned length = strlen(str);
   char *end = str + length - 1; // Don't move the terminal NULL !!!

   do {
      char t;
      t = *str, *str = *end, *end = t;
      str++; end--;
   } while ( str < end);
}
```

exercise: decimal-to-binary

- Write a function that accepts:
  - an empty string which will receive the result,
  - the string's maximum length,
  - a decimal number to be converted to binary
- The function uses "right-to-left" to fill the string with the binary form of the decimal number
  - At the end it should either reverse the bits itself, or call another function to do so
- Write a driver program to use the function(s)

Decimal → Hexadecimal

- Just like Decimal → Binary, but the new base is Hexadecimal instead of binary
- right-to-left:
  - repeat:
    - divide value by base
    - convert *remainder* to digit in the base (e.g. A,B,C...)
    - save digit in next-more-significant position of converted number
    - set value equal to value divided by base (w/o remainder)
  - This again produces a right-to left "backwards" result, which *must be reversed*.

Example: $999999_{10} \rightarrow ?_{16}$

- 999999 ÷ 16 → 62499, r 15 = f
- 62499 ÷ 16 → 3906, r 3
- 3906 ÷ 16 → 244, r 2
- 244 ÷ 16 → 15, r 4
- 15 ÷ 16 → **0**, r 15 = f

...and we're done

- $999999_{10} \rightarrow$ f 423f$_{16}$

Decimal → <any other base>

- Just like Decimal → Binary or Hexadecimal , but the base is something else ...
- right-to-left:
  - repeat:
    - divide value by base
    - convert *remainder* to digit in the base (e.g. A,B,C...)
    - save digit in next-more-significant position of converted number
    - set value equal to value divided by base (w/o remainder)
  - Remember that appending each remainder to a list produces a right-to left "backwards" result, which *must be reversed*.

Example: $54321_{10} \rightarrow ?_7$

- 54321 ÷ 7 → 7760, r 4
- 7760 ÷ 7 → 1108, r 4
- 1108 ÷ 7 → 158, r 2
- 158 ÷ 7 → 22, r 4
- 22 ÷ 7 → 3, r 1
- 3 ÷ 7 → **0**, r 3

...and we're done

- $54321_{10} \rightarrow 31\ 4244_7$

exercise: decimal-to-any base

- Modify the decimal-to-binary function so that it works for any target base
  - Needs another argument to specify the target base
  - Only bases up to 16 have to be covered
- Converting numeric remainders to ASCII characters:
  - ...see next slide...

Converting numeric remainders to digits

- In general, the "digits" must be represented by ASCII characters.
- Most general solution:
  if-else or switch() statement
  - still cumbersome for larger bases
    - base-30 ain't gonna be pretty…
  - permits multi-character representations of remainders
    - necessary for base-257 and higher (eek)
- Direct conversion: add 0x30 to the remainder
  - only works for digits ≤ 9
  - modifications allow for digits a..z, A..Z

Lookup Tables

- Table lookup consists of setting up a table of *values* matched to *keys* or *indexes*, and looking for the entry corresponding to the desired key.
  - Also called "LUT", "Content-Addressable Memory", "dictionary"
- For digit conversions:
  - the "table" is an array of ASCII characters
    - in C, works for single-character representations
  - the remainders are the "keys", or indexes into the array