Number Systems

Number systems

- As humans, we prefer base 10 a.k.a. decimal.
- For reasons we will discuss, computers prefer different number systems…
  - Binary (base 2)
  - Hexadecimal (base 16)
- Its easy to understand these other number systems if we analyze how base 10 works.

Base 10

- For every base, the value of each digit depends position in the number.
- Decimal uses 10 digits, 0-9

7423

-is-

$(7 * 10^3) + (4 * 10^2) + (2 * 10^1) + (3 * 10^0)$

-or-

7000 + 400 + 20 + 3

Base 10

- For every base, the value of each digit depends position in the number.
- Decimal uses 10 digits, 0-9

  7423

  -is-

  $(7 * 10^3) + (4 * 10^2) + (2 * 10^1) + (3 * 10^0)$

  -or-

  7000 + 400 + 20 + 3

- Assuming 'w' represents the 'width' of the number and 'x' represents the number itself, we can generalize and convert any base to decimal as follows….

$$\sum_{i=0}^{w-1} x_i \cdot \boldsymbol{base}^i$$

Base 2

- Binary uses 2 digits, 0-1

1101

-is-

$(1 * 2^3) + (1 * 2^2) + (0 * 2^1) + (1 * 2^0)$

-or-

$8 + 4 + 0 + 1 = 13_{10}$

Base 2

- Binary uses 2 digits, 0-1

  1101

  -is-

  $(1 * 2^3) + (1 * 2^2) + (0 * 2^1) + (1 * 2^0)$

  -or-

  $8 + 4 + 0 + 1 = 13_{10}$

- You should be able to do this by hand! I have provided a way to check.
- See *decimal_to_binary.c*

Base 16

- Hexadecimal uses 16 digits 0-9, A-F *(A=10, B=11, etc.)*

  742A

  -is-

  $(7 * 16^3) + (4 * 16^2) + (2 * 16^1) + (10 * 16^0)$

  -or-

  $28672 + 1024 + 32 + 10 = 29738_{10}$

- You can see that the higher the base, the fewer digits it takes to express some value.
- Hexadecimal is used often notate memory addresses (among other things)

Relative expressiveness of systems

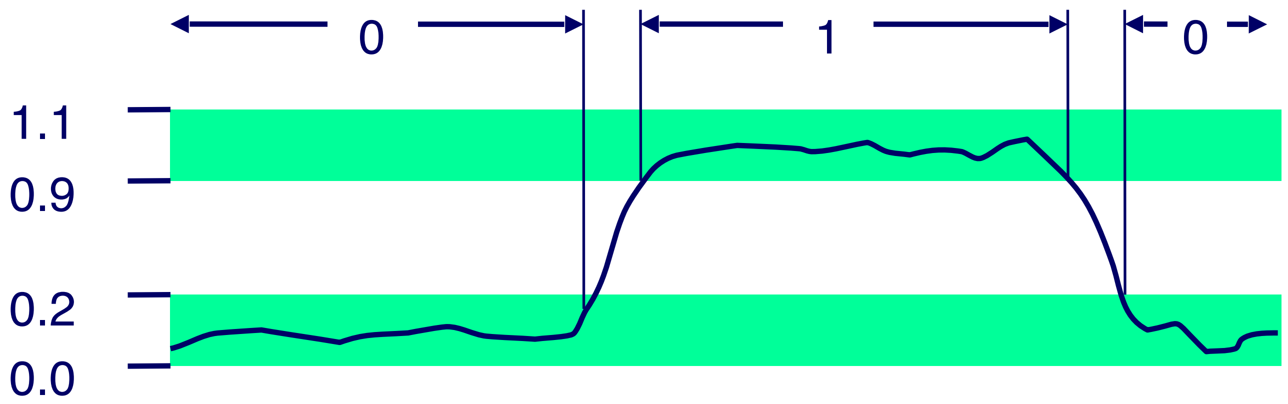
- Hexadecimal is useful because its often more convenient to write one digit as opposed to than four.
  - In other words, since a single digit in hexadecimal can represent 16 values, it can hold as much information as 4 bits.
- This chart here you should attempt to commit to memory, or at least be able to work out relatively quickly.

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Representing Information as Bits

Everything is bits

- Each bit is 0 or 1
- Everything on a computer is encoded as sets of binary digits, or *bits*
  - All programs running on disk and running in memory are represented as sets of bits
  - … and represent and manipulate numbers, sets, strings, etc…
- Why bits? Electronic implementation
  - Easy to store on bistable elements (an electronic circuit that has two stable states)
  - Reliably transmitted on noisy and inaccurate wires.



Everything is bits *con't*

- Again, the basic unit of information in computing is the bit.
- A single bit denotes two states "on or off"
- Note that values with more than two states require multiple bits.
  - A collection of two bits has four possible states
    - Ex. 00, 01, 10, 11
  - A collection of three bits has eight possible states
    - Ex. 000, 001, 010, 011, 100, 101, 110, 111
  - A collection of *w* bits has 2^w possible states.
- We call a collection of bits a **'bit vector'**

Bytes

- Byte = 8 bits
  - So how many different values can it represent?

- It is the smallest addressable unit of memory in most computer architectures.

- Range of representation (non-negative integers)
  - Binary: 00000000_2 to 11111111_2
  - Decimal: 0_{10} to 255_{10}
  - Hexadecimal: 00_{16} to FF_{16}
    - By the way, we can write hexadecimal numbers in C as
      - 0xFF or 0xff
      - See *hex.c*

Data types in C in bytes

C Data Type	Typical 32-bit	Typical 64-bit
`char`	1	1
`short`	2	2
`int`	4	4
`long`	4	8
`float`	4	4
`double`	8	8
pointer	4	8

MB, KB, GB….

- In some contexts, what is meant by a KB, MB, or GB differ.
  - Depends on number base, 2 or 10
- The true names for these things are mebibyte, kebibyte, gibibyte, etc..
  - You can read in detail here https://en.wikipedia.org/wiki/Kibibyte
- In computers we love base 2 so we will be using the binary semantics of the 'mega', 'kilo', etc. prefixes.

Binary	Decimal
1 KB (1KiB) = 2^10 bytes = 1,024 bytes	1 KB = 10^3 bytes = 1,000 bytes
1MB (1MiB) = 2^20 bytes = 1,048,576 bytes	1 MB = 10^6 bytes = 1,000,000 bytes
1GB (1GiB) = 2^30 bytes = 1,073,741,824 bytes	1 GB = 10^9 bytes = 1,000,000,000 bytes
1TB (1TiB) = 2^40 bytes = 1,099,511,627,776 bytes	1 TB = 10^12 bytes = 1,000,000,000,000 bytes

Bit-level manipulations

Boolean Algebra

- Algebraic representation of logic
  - Encode "True" as 1 and "False" as 0

And

A&B = 1 when both A=1 and B=1

&	0	1
0	0	0
1	0	1

Or

A|B = 1 when either A=1 or B=1

|	0	1
0	0	1
1	1	1

Not

~A = 1 when A = 0

~	
0	1
1	0

Exclusive-Or (Xor)

A^B = 1 when either A = 1 or B = 1, but not both

^	0	1
0	0	1
1	1	0

Boolean Algebra *con't*

- We can use this algebra to operate on bit vectors
  - Operations applied bitwise

```
   01101001      01101001      01101001
 & 01010101    | 01010101    ^ 01010101    ~ 01010101
 __________    __________    __________    __________
   01000001      01111101      00111100      10101010
```

  - All the properties of Boolean Algebra apply.
  - We have these operators in C, they are called *bitwise* operators.

Bit-level operations in C

- Operations &, |, ~, ^ available in C
  - Apply to any "integral" data type (long, int, short, char, unsigned)
  - View arguments as bit vectors, arguments applied bit-wise
- Examples:

```
     ~ 1100 = 0011

0110 & 1010 = 0010

0110 | 1010 = 1110

0110 ^ 1010 = 1100
```

- See *bit_flipping.c*

Example: representing & manipulating sets

- "Bit sets", very useful in practice
  - Width w bit vector represents subsets of {0, …, w–1}
  - $a_j = 1$ if $j \in A$

01101001 represents set { 0, 3, 5, 6 } — Set A
*7**65**4**3**21**0***

01010101 represents set { 0, 2, 4, 6 } — Set B
7**6**5**4**3**2**1**0**

- Operations

&	Intersection (A & B)	01000001	{ 0, 6 }
|	Union (A | B)	01111101	{ 0, 2, 3, 4, 5, 6 }
^	Symmetric difference (A ^ B)	00111100	{ 2, 3, 4, 5 }
~	Complement (~B)	10101010	{ 1, 3, 5, 7 }

Contrast: logical operators

- These operators in some cases looks the same but have very different effects.

- &&, ||, !
  - View 0 as "False"
  - Anything nonzero as "True"
  - All expressions with these operators always return 0 or 1
  - Early termination a.k.a. "short circuiting"

- Example mistake:
  - 1010 & 0101 ➙ 0000 (false)

    -vs-
  - 1010 && 0101 ➙ 0001 (true)

- See *bitwise_vs_boolean.c*

Shift operations

- **Left Shift: x << y**
  - Shift bit-vector x left y positions
    - Throw away extra bits on left
    - Fill with 0's on right
- **Right Shift: x >> y**
  - Shift bit-vector x right y positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0's on left
  - Arithmetic shift
    - Replicate most significant bit on left
- **Undefined Behavior**
  - Shift amount < 0 or ≥ width of type
- See *bit_shifting.c*

Argument x	01100010
<< 3	00010*000*
Log. >> 2	*00*011000
Arith. >> 2	*00*011000

Argument x	10100010
<< 3	00010*000*
Log. >> 2	*00*101000
Arith. >> 2	*11*101000

Swapping values using XOR

- Swapping values of two variables normally requires a temporary storage
- Using the bitwise exclusive or operator we can actually do this using only the storage of the two bit-vectors
- See *xor_swap.c*

Integer Encoding

Two Types of Integers

- **Unsigned**
  - positive numbers and 0
  - *unsigned char* has a range of 0-255
- **Signed**
  - negative numbers as well as positive numbers and 0
  - *signed char* has a range of -128-127
- Signed and unsigned have the same cardinality, but different ranges of values!
- If **unsigned** keyword used type is unsigned, if not defaults to signed.

```
int signed_int = -1;                /* positive & negative allowed */
unsigned int unsigned_int = 1;      /* only non-negative allowed */
```

Unsigned Integers

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

- B2U stands for binary-to-unsigned
- X is a binary number, a bit pattern
- w is the ‘width’ of the binary number (i.e. number of bits)
- Take the sum of every i’th position of X multiplied by 2^i

Unsigned Integers *con't*

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

1 0 1 1 1 0 1 1

128 0 32 16 8 0 2 1 $= 187_{10}$

Signed integers

$$B2T(X) \;=\; \underbrace{-x_{w-1} \cdot 2^{w-1}}_{\textbf{sign bit}} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

- B2T stands for binary-to-twos-complement
- Same as equation for binary-to-unsigned, with one modification.
- For 2's complement *most significant bit* indicates *sign*, gets special treatment
  - 0 indicates a nonnegative number
  - 1 indicates a negative number

Signed integers *con't*

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

sign bit

1 0 1 1 1 0 1 1

-128 0 32 16 8 0 2 1 = -69_{10}

Signed integers *con't*

- Again….
  - With n bits, we have 2^n distinct values.
    - Assign about half to positive integers and about half to negative
  - non-negative integers
    - if 0 in most significant bit, behave like unsigned:
      0101 = 5
  - Negative integers
    - If 1 in most significant bit, use two's complement form:
      1101 = -3

Numeric ranges

- **Unsigned**
  - Umin = 0
  - Umax = 2^w-1
- Example
  - Assume w = 5
    - Smallest unsigned $00000_2 = 0_{10}$
    - Largest unsigned $11111_2 = 31_{10}$

- **Signed** (Two's Complement)
  - Tmin = -2^{w-1}
  - Tmin = $2^{w-1} - 1$
- Example
  - Assume w = 5
    - Smallest signed $10000_2 = -16_{10}$
    - Largest signed $01111_2 = 15_{10}$

Umax, Tmin, Tmax for standard widths

	W			
	8	**16**	**32**	**64**
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

- Observations:
  - For a given value of w

    Umax = 2 * Tmax + 1
  - Range of two's complement not symmetric

    |Tmin| = |Tmax| + 1

- In C…
  - These ranges are system specific. Therefore, to reference them we must *#include<limits.h>*
  - Declares constants, e.g.,
    - UINT_MAX
    - INT_MAX
    - INT_MIN
  - See *limits.c*

+

Comparison of unsigned & signed

X	B2U(*X*)	B2T(*X*)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

- **Equivalence**
  - Same encodings for non-negative values.
  - +/- 16 for negative two's complement and positive unsigned 4-bit values.
- **Uniqueness**
  - Every bit pattern represents a unique integer value.
  - Ever integer has a unique bit pattern.