CS 33

Data Representation, Part 1

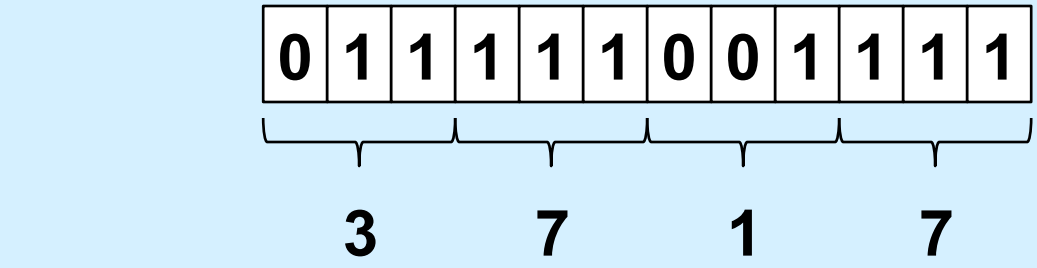
Number Representation

- Hindu-Arabic numerals
  - developed by Hindus starting in 5th century
    - » positional notation
    - » symbol for 0
  - adopted and modified somewhat later by Arabs
    - » known by them as “Rakam Al-Hind” (Hindu numeral system)
  - 1999 rather than MCMXCIX
    - » (try doing long division with Roman numerals!)

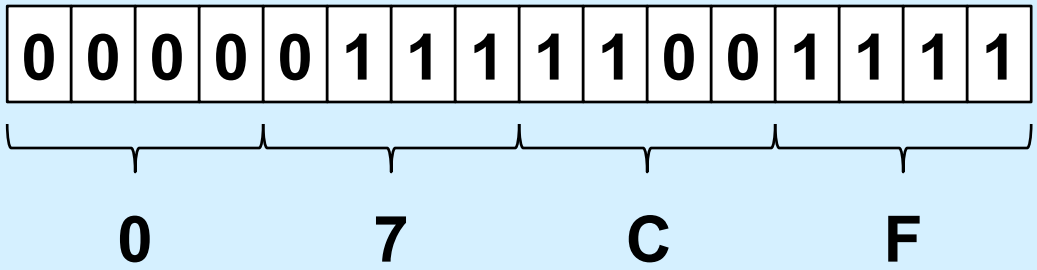
Which Base?

- 1999
  - base 10
    - » $9 \cdot 10^0 + 9 \cdot 10^1 + 9 \cdot 10^2 + 1 \cdot 10^3$
  - base 2
    - » 11111001111
      - $1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 + 0 \cdot 2^4 + 0 \cdot 2^5 + 1 \cdot 2^6 + 1 \cdot 2^7 + 1 \cdot 2^8 + 1 \cdot 2^9 + 1 \cdot 2^{10}$
  - base 8
    - » 3717
      - $7 \cdot 8^0 + 1 \cdot 8^1 + 7 \cdot 8^2 + 3 \cdot 8^3$
    - » why are we interested?
  - base 16
    - » 7CF
      - $15 \cdot 16^0 + 12 \cdot 16^1 + 7 \cdot 16^2$
    - » why are we interested?

Words …



12-bit computer word



16-bit computer word

Algorithm …

```
void baseX(unsigned int num, unsigned int base) {
   char digits[] = {'0', '1', '2', '3', '4', '5', '6', … };
   char buf[8*sizeof(unsigned int)+1];
   int i;

   for (i = sizeof(buf) - 2; i >= 0; i--) {
      buf[i] = digits[num%base];
      num /= base;
      if (num == 0)
         break;
   }

   buf[sizeof(buf) - 1] = '\0';
   printf("%s\n", &buf[i]);
}
```

Or …

```
$ bc
obase=16
1999
7CF
$
```

Quiz 1

- **What's the decimal (base 10) equivalent of 23_{16}?**
  - **a) 19**
  - **b) 33**
  - **c) 35**
  - **d) 37**

Encoding Byte Values

- **Byte = 8 bits**
  - **binary 00000000_2 to 11111111_2**
  - **decimal: 0_{10} to 255_{10}**
  - **hexadecimal 00_{16} to FF_{16}**
    - » **base 16 number representation**
    - » **use characters '0' to '9' and 'A' to 'F'**
    - » **write $FA1D37B_{16}$ in C as**
      - **0xFA1D37B**
      - **0xfa1d37b**

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

Boolean Algebra

- **Developed by George Boole in 19th Century**
  - **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

General Boolean Algebras

- **Operate on bit vectors**
  - **operations applied bitwise**

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

- **All of the properties of boolean algebra apply**

Example: Representing & Manipulating Sets

- **Representation**
  - width-w bit vector represents subsets of {0, …, w–1}
  - $a_j = 1$ iff $j \in A$

    01101001 { 0, 3, 5, 6 }
    76543210

    01010101 { 0, 2, 4, 6 }
    76543210

- **Operations**

&	intersection	01000001	{ 0, 6 }
|	union	01111101	{ 0, 2, 3, 4, 5, 6 }
^	symmetric difference	00111100	{ 2, 3, 4, 5 }
~	complement	10101010	{ 1, 3, 5, 7 }

Bit-Level Operations in C

- **Operations &, |, ~, ^ available in C**
  - **apply to any "integral" data type**
    - » `long, int, short, char`
  - **view arguments as bit vectors**
  - **arguments applied bit-wise**
- **Examples (char datatype)**

`~0x41 → 0xBE`

$\sim 01000001_2 \rightarrow 10111110_2$

`~0x00 → 0xFF`

$\sim 00000000_2 \rightarrow 11111111_2$

`0x69 & 0x55 → 0x41`

$01101001_2 \;\&\; 01010101_2 \rightarrow 01000001_2$

`0x69 | 0x55 → 0x7D`

$01101001_2 \mid 01010101_2 \rightarrow 01111101_2$

Contrast: Logic Operations in C

- **Contrast to Logical Operators**
  - `&&, ||, !`
    - » **view 0 as "false"**
    - » **anything nonzero as "true"**
    - » **always return 0 or 1**
    - » **early termination/short-circuited execution**
- **Examples (char datatype)**

```
!0x41 → 0x00
!0x00 → 0x01
!!0x41 → 0x01

0x69 && 0x55 → 0x01
0x69 || 0x55 → 0x01
p && *p
```

(avoids null pointer access)

Contrast: Logic Operations in C

- **Contrast to Logical Operators**
  - **&&, ||, !**
    - **» vie… “false”**

Watch out for && vs. & (and || vs. |)… One of the more common oopsies in C programming

- **E…**

```
!0x41 → 0x00
!0x00 → 0x01
!!0x41 → 0x01

0x69 && 0x55 → 0x01
0x69 || 0x55 → 0x01
p && *p     (avoids null pointer access)
```

Shift Operations

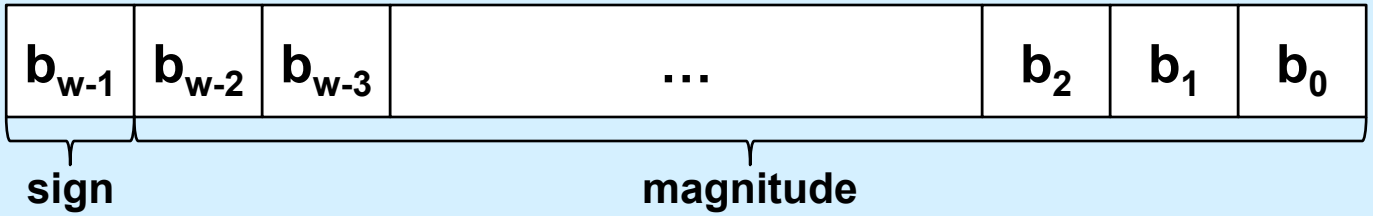
- **Left Shift: `x << y`**
  - **shift bit-vector `x` left `y` positions**
    - **throw away extra bits on left**
  - » **fill with `0`'s on right**
- **Right Shift: `x >> y`**
  - **shift bit-vector `x` right `y` positions**
    - » **throw away extra bits on right**
  - **logical shift**
    - » **fill with `0`'s on left**
  - **arithmetic shift**
    - » **replicate most significant bit on left**
- **Undefined Behavior**
  - **shift amount < 0 or ≥ word size**

Argument `x`	`01100010`
`<< 3`	`00010000`
Log. `>> 2`	`00011000`
Arith. `>> 2`	`00011000`

Argument `x`	`10100010`
`<< 3`	`00010000`
Log. `>> 2`	`00101000`
Arith. `>> 2`	`11101000`

Signed Integers

- **Sign-magnitude**



$$\text{value} = (-1)^{b_{w-1}} \cdot \sum_{i=0}^{w-2} b_i \cdot 2^i$$

- two representations of zero!
  - computer must have two sets of instructions
    - one for signed arithmetic, one for unsigned

Signed Integers

- **Ones' complement**
  - **negate a number by forming its bitwise complement**
    - » **e.g., (-1)·01101011 = 10010100**

value = $-b_{w-1}(2^{w-1}-1)+\sum_{i=0}^{w-2} b_i \cdot 2^i$

$$= \sum_{i=0}^{w-2} b_i \cdot 2^i \quad \text{if } b_{w-1}=0$$

$$= \sum_{i=0}^{w-2} (b_i - 1) \cdot 2^i \quad \text{if } b_{w-1}=1$$

two zeroes!

Signed Integers

- **Two's complement**

  $b_{w-1} = 0 \Rightarrow$ non-negative number

$$\text{value} = \sum_{i=0}^{w-2} b_i \cdot 2^i$$

  $b_{w-1} = 1 \Rightarrow$ negative number

$$\text{value} = (-1) \cdot 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i$$

one zero!

Signed Integers

- **Negating two's complement**

$$value = -b_{w-1}2^{w-1} + \sum_{i=0}^{w-2} b_i 2^i$$

  - **how to compute –*value*?**

    (~value)+1

Signed Integers

- **Negating two's complement (continued)**

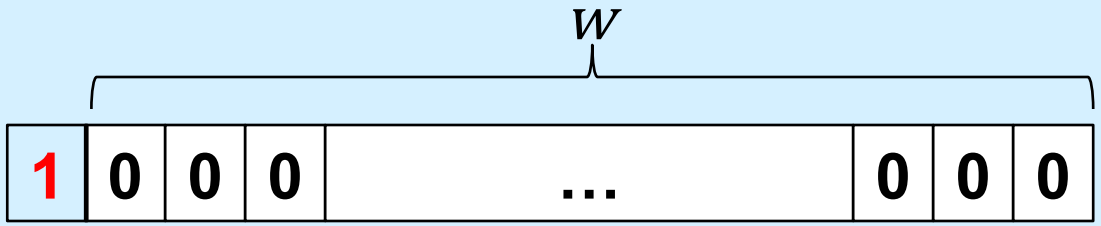
$$value + (\sim value + 1)$$

$$= (value + \sim value) + 1$$

$$= (2^w - 1) + 1$$

$$= 2^w$$

$$=$$



Quiz 2

- **We have a computer with 4-bit words that uses two's complement to represent negative numbers. What is the result of subtracting 0010 (2) from 0001 (1)?**
  a) **0111**
  b) **1001**
  c) **1110**
  d) **1111**