CSCI-UA.0201

Computer Systems Organization

Data Representation – Bits and Bytes

Thomas Wies

wies@cs.nyu.edu

https://cs.nyu.edu/wies

Byte Ordering Example

- Big Endian
  - Most significant byte has lowest address
- Little Endian
  - Most significant byte has highest address
- Example
  - Variable x has 4-byte representation 0x01234567
  - Address given by &x is 0x100

Most
Significant
Byte

Big Endian

		0x100	0x101	0x102	0x103		
		01	23	45	67		

Little Endian

		0x100	0x101	0x102	0x103		
		67	45	23	01		

Examining Data Representations

- Code to print Byte Representation of data

```
void show_bytes(unsigned char * start, int len){
  int i;
  for (i = 0; i < len; i++)
    printf(”%p\t%2x\n",start+i, start[i]);
  printf("\n");
}
```

`printf` directives:

%p: Print pointer

%x: Print integer in hexadecimal

show_bytes Execution Example

```
int a = 0x12345678;
printf("int a = 0x12345678;\n");
show_bytes((unsigned char *) &a, sizeof(int));
```

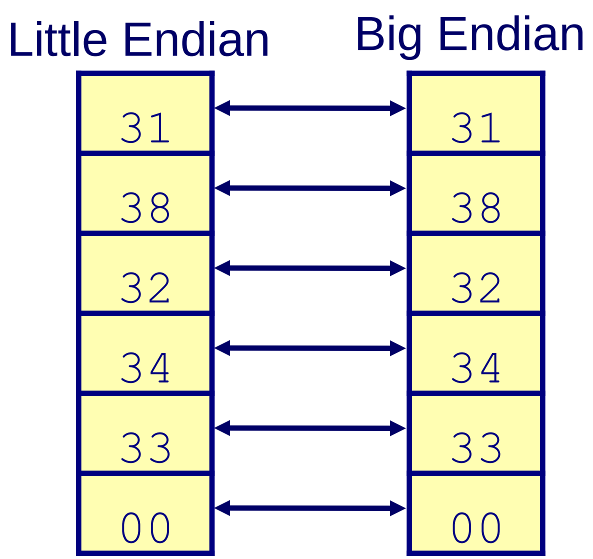
Result (Linux):

```
int a = 0x12345678;
0x11ffffcb8  0x78
0x11ffffcb9  0x56
0x11ffffcba  0x34
0x11ffffcbb  0x12
```

Representing Strings

```
char S[6] = "18243";
```

- Strings in C
  - Represented by array of characters
  - Each character encoded in ASCII format
    - Standard 7-bit encoding of character set
    - Character ‘0’ has code 0x30
      - Digit *i* has code 0x30+*i*
  - String should be null-terminated
- Byte ordering not an issue



Byte ordering is an issue for a single data item.
An array is a group of data items.

How to Manipulate Bits?

Boolean Algebra

- Applying Boolean operations, such as XOR, NAND, AND, …, to bits to generate new bit values.

And

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

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

Or

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

A	B	A | B
0	0	0
0	1	1
1	0	1
1	1	1

Not

- **~A = 1 when A=0**

A	~ A
0	1
1	0

Exclusive-Or (Xor)

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

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Boolean Algebra

- Applying Boolean operations, such as XOR, NAND, AND, …, to bits to generate new bit values.

NAND

- **The reverse of AND**

A	B	~(A&B)
0	0	1
0	1	1
1	0	1
1	1	0

NOR

- **The reverse of OR**

A	B	~(A|B)
0	0	1
0	1	0
1	0	0
1	1	0

Exclusive-NOR (Xor)

- **The reverse of XOR**

A	B	~(A^B)
0	0	1
0	1	0
1	0	0
1	1	1

Application of Boolean Algebra

- Applied to Digital Systems by Claude Shannon
  - 1937 MIT Master’s Thesis
  - Reason about networks of relay switches
    - Encode closed switch as 1, open switch as 0

Transistor

Lifting Operations to Bit Vectors

- Operate on Bit Vectors (e.g. an integer is a bit vector of 4 bytes = 32 bits)
  - Operations applied bitwise

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

Bit-Level Operations in C

- Operations `&`, `|`, `~`, `^` Available in C
  - Apply to any "integral" data type
    - `long`, `int`, `short`, `char`, `unsigned`
- Examples (Char data type)
  - `~0x41 = 0xBE`
    - $\sim 01000001_2 = 10111110_2$
  - `~0x00 = 0xFF`
    - $\sim 00000000_2 = 11111111_2$
  - `0x69 & 0x55 = 0x41`
    - $01101001_2 \,\&\, 01010101_2 = 01000001_2$
  - `0x69 | 0x55 = 0x7D`
    - $01101001_2 \mid 01010101_2 = 01111101_2$

Contrast: Logic Operations in C

- Contrast to Logical Operators

  `&&, ||, !`

  - View 0 as "false"
  - Anything nonzero as "true"
  - Always return 0 or 1
- Examples
  - `!0x41 = 0x00`
  - `!0x00 = 0x01`
  - `!!0x41 = 0x01`

  - `0x69 && 0x55 = 0x01`
  - `0x69 || 0x55 = 0x01`
  - `p && *p` (avoids null pointer access – short circuiting)

Type `bool` in C

- Did not exist in standard C89/90
- It was introduced in C99 standard
- You may need to use the following switch with gcc:

  gcc –std=c99 …

```
#include <stdbool.h>
bool x;
x = false;   ← lower case
x = true;
```

Shift Operations

- Left Shift: `x << y`
  - Shift `x` left by `y` positions
    - Throw away extra bits on left
    - Fill with `0`'s on right
- Right Shift: `x >> y`
  - Shift `x` right `y` positions
    - Throw away extra bits on right
  - type1: Logical shift
    - Fill with `0`'s on left
  - type 2: Arithmetic shift (covered later)
    - Replicate most significant bit on right
- Undefined Behavior
  - Shift amount < 0 or ≥ size of x

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

How to present Integers in binary?

Two Types of Integers

- Unsigned
  - positive numbers and 0
- Signed numbers
  - negative numbers as well as positive numbers and 0

Unsigned Integers

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

1 0 1 1 1 0 1 1

128 64 32 16 8 4 2 1

187

Unsigned Integers

- An n-bit unsigned integer represents 2^n values:
  from 0 to 2^n-1.

2^2	2^1	2^0	
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

Unsigned Binary Arithmetic

- Base-2 addition – just like base-10
  - add from right to left, propagating carry

carry

```
  10010        10010       1111
+  1001      +  1011     +    1
  11011        11101      10000


               10111
             +   111
```

What About Negative Numbers?

People have tried several options:

Sign Magnitude:	One's Complement	Two's Complement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1

- Issues: balance, number of zeros, ease of operations
- Which one is best? Why?

Signed Integers

- With n bits, we have 2^n distinct values.
  - assign about half to positive integers and about half to negative
- **Positive integers**
  - just like unsigned: zero in *most significant* (MS) bit
    00101 = 5
- **Negative integers**
  - In two's complement form

In general: a 0 at the MS bit indicates positive and a 1 indicates negative.

Two's Complement

- *Two's complement* representation developed to make circuits easy for arithmetic.
  - for each positive number (X), assign value to its negative (-X), such that X + (-X) = 0 with "normal" addition, ignoring carry out

```
  00101   (5)          01001   (9)
+ 11011   (-5)       + 10111   (-9)
  -----                -----
  00000   (0)          00000   (0)
```

Two's Complement Signed Integers

- MS bit is sign bit.
- Range of an n-bit number: -2^{n-1} through $2^{n-1} - 1$.
  - The most negative number (-2^{n-1}) has no positive counterpart.

-2^3	2^2	2^1	2^0	
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7

-2^3	2^2	2^1	2^0	
1	0	0	0	-8
1	0	0	1	-7
1	0	1	0	-6
1	0	1	1	-5
1	1	0	0	-4
1	1	0	1	-3
1	1	1	0	-2
1	1	1	1	-1

Converting Binary (2's C) to Decimal

1. If MS bit is one (i.e. number is negative), take two's complement to get a positive number.
2. Get the decimal as if the number is unsigned (using power of 2s).
3. If original number was negative, add a minus sign.

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

Examples

$$X = 00100111_{two}$$
$$= 2^5+2^2+2^1+2^0 = 32+4+2+1$$
$$= 39_{ten}$$

$$X = 11100110_{two}$$
$$-X = 00011010$$
$$= 2^4+2^3+2^1 = 16+8+2$$
$$= 26_{ten}$$
$$X = -26_{ten}$$

n	**2^n**
0	**1**
1	**2**
2	**4**
3	**8**
4	**16**
5	**32**
6	**64**
7	**128**
8	**256**
9	**512**
10	**1024**

Shift Operations

- Left Shift: `x << y`
  - Shift `x` left by `y` positions
    - Throw away extra bits on left
    - Fill with `0`'s on right
- Right Shift: `x >> y`
  - Shift `x` right `y` positions
    - Throw away extra bits on right
  - type1: Logical shift
    - Fill with `0`'s on left
  - type 2: Arithmetic shift (covered later)
    - Replicate most significant bit on right
- Undefined Behavior
  - Shift amount < 0 or ≥ size of x

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

Numeric Ranges

Example: Assume 16-bit numbers

	Decimal	Hex	Binary
Unsigned Max	**65535**	**FF FF**	11111111 11111111
Signed Max	**32767**	7F FF	01111111 11111111
Signed Min	**-32768**	80 00	10000000 00000000
-1	**-1**	**FF FF**	11111111 11111111
0	**0**	00 00	00000000 00000000

Values for Different Sizes

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

- **C Programming**
  - #include <limits.h>
  - Declares constants, e.g.,
    - INT_MAX
    - LONG_MAX
    - INT_MIN
    - UINT_MIN
    - …