COMP 264 Fall 2015 Lecture Notes

R. I. Greenberg

September 09, 2015

1 Information Storage

1.1 Binary, Octal, and Hexadecimal

- Base 10 natural for many applications, but base 2 easier in hardware.
- Base 2 place values analogous to base 10.
- Binary notation is verbose, but decimal doesn't readily expose the bit pattern.
- Octal shorthand — group bits in threes: 000 = 0, 001 = 1, 010 = 2, 011 = 3, 100 = 4, 101 = 5, 110 = 6, 111 = 7.
- Hexadecimal — same idea but group bits in fours. But we need a one character code for each group; after 0–9, use A–F.
- Graphical examples in class.
- C notations: `0x...` for hex; `0...` without `x` for octal.
- Decimal conversions:
  - To decimal: use the place values.
  - From decimal: repeatedly divide by base (2 or 8 or 16) and collect the remainders.

1.2 Word Size

- n-bit word size allows virtual addresses 0 to $2^n - 1$.
- 32-bit word size: 4GB. But much larger disks are now routinely available. By going up to 64-bit word size, virtual address space expands to 16EB (exabytes).

1.3 Data Sizes, Pointers

- Likely sizes for C variables on 32-bit and 64-bit machines as per textbook Figure 2.3.
- Pointers. Like an ordinary variable but contains the address of a data item instead of the actual data item. Pointers still have type but will need to occupy full word size. Form of C declaration, e.g., `float *p`.
- A New "Y2K"-like "crisis"? Programs written with bad assumptions about data sizes will get into trouble on 64-bit machines.

Most 64-bit machines can run programs compiled for use on 32-bit machines, and there is a gcc flag `-m32` that can be used to generate a program that should run on a 32-bit or 64-bit machine, whereas the `-m64-` flag will yield a program that will only run on a 64-bit machine. (Default on `shannon` is 64-bit.)

1.4 Addressing, Byte Ordering, and C

- Multibyte objects generally addressed by specifying smallest addr. of bytes used. But two conventions for byte ordering: *big endian* (MSB first) and *little endian* (LSB first). Illustration in class.
- Endianness usually irrelevant for a programmer on a single machine but
  - Matters when transferring data from one machine to another.
  - One can manipulate individual bytes in C if desired. See `show-bytes.c` code in text Figure 2.4, and homework. (Can see also a little different verion in *02-bits-ints slides* 10–14.)
  - Need to understand byte ordering to read machine-level code representation produced by compiler.
- A very optional Intel white paper with more detail on endianness is as follows:
  Intel. Endianness white paper. `http://download.intel.com/design/intarch/papers/endian.pdf`, 2004.

Some C coding points:

- Formatted printing: read up on `printf`. Ex.:

```
int coursenum=264;
char *coursename="Intro. to Computing Systems";
printf("Course number %d is called %s.", coursenum, coursename);
```

- `&` and `*` are inverse operators for pointer creation and dereferencing. Ex.:

  ```
  int n=29;
  int *pointer_to_n;
  pointer_to_n = &n;
  printf("n directly: %d & through pointer: %d.\nAddress of n: %p.\n",
         n, *pointer_to_n, pointer_to_n);
  ```

- Type tricks.

  - `typedef` gives a shorthand name to a type, e.g.,

    ```
    typedef long unsigned int *ptrtoluint;
    ptrtoluint foobar;
    ptrtoluint barfoo;
    ```

    equivalent to

    ```
    long unsigned int *foobar;
    long unsigned int *barfoo;
    ```

  - A *cast* does type conversion, e.g.,:

    ```
    int x=5;
    int y=2;
    printf("Int quotient: %d, FP quotient: %f\n", 5/2, ((float) 5)/2);
    ```

  - Text's `show-bytes.c` code converts various kinds of pointers to the defined type `byte_pointer`, a pointer to an `unsigned char` so that array reference `start[i]` is interpreted correctly.

1.5 Representing Strings

Now we can better understand the output of `man ascii`.

Character with code 0 is called the null character. Strings in C always terminated by a NUL.

1.6 Representing Code

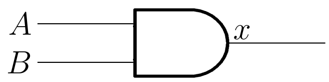
A sequence of bytes as illustrated in text sections 1.1 and 2.1.6. Representation generally differs when hardware or operating systems is different.

1.7 Boolean Algebra

- Boolean algebra and the underlying logical operations as in COMP 150/163, but C notations a little different: (Still using 1 for TRUE and 0 for FALSE.)

  – AND

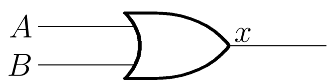
A	B	x
0	0	0
0	1	0
1	0	0
1	1	1

  $x = AB$ or
  $x = A \cdot B$ or
  $x = A \wedge B$
  C code: `x=A&&B`
  ("product")

  – OR

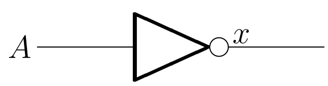
A	B	x
0	0	0
0	1	1
1	0	1
1	1	1

  $x = A + B$ or
  $x = A \vee B$
  C code: `x=A||B`
  ("sum")

  – NOT

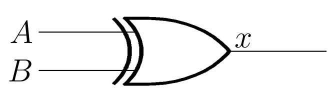
A	x
0	1
1	0

  $x = A'$ or
  $x = \overline{A}$ or
  $x = \neg A$
  C code: `x=!A`
  ("complement", "negation")

  – XOR
  (exclusive OR)

A	B	x
0	0	0
0	1	1
1	0	1
1	1	0

  $x = A \oplus B$
  = A'B+AB'
  C code: (bitwise below)

- Recall important identities: (Items 1 through 5 are defining laws for a boolean algebra Items 6 through 11 follow from the prior identities.)

  1. Associative laws:

$$(x + y) + z = x + (y + z) \qquad \text{and} \qquad (xy)z = x(yz) \qquad \forall x, y, z$$

2. Commutative laws:
$$x + y = y + x \qquad \text{and} \qquad xy = yx \qquad \forall x, y$$
3. Distributive laws:
$$x(y + z) = xy + xz \qquad \text{and} \qquad x + yz = (x + y)(x + z) \qquad \forall x, y, z$$
4. Identity laws:
$$x + 0 = x \qquad \text{and} \qquad x1 = x \qquad \forall x$$
5. Complement laws:
$$x + x' = 1 \qquad \text{and} \qquad xx' = 0 \qquad \forall x$$
6. Idempotent laws:
$$x + x = x \qquad \text{and} \qquad xx = x \qquad \forall x$$
7. Bound laws:
$$x + 1 = 1 \qquad \text{and} \qquad x0 = 0 \qquad \forall x$$
8. Absorption laws:
$$x + xy = x \qquad \text{and} \qquad x(x + y) = x \qquad \forall x, y$$
9. Involution law:
$$(x')' = x \qquad \forall x$$
10. 0 and 1 laws:
$$0' = 1 \qquad \text{and} \qquad 1' = 0$$
11. De Morgan's laws:
$$(x + y)' = x'y' \qquad \text{and} \qquad (xy)' = x' + y' \qquad \forall x, y$$

1.8 Bit-Level Operations in C

- In addition to the logical operators we've seen briefly, Cprovides operators that can apply to any integral data type bitwise. Specifically, `&&`, `||`, and `!` change to `&`, `|`, and `~`. We also now have XOR directly available bitwise as `^`. For example, if `x` and `y` are C variables with bit patterns as shown:

```
  x     01001110
  y     01100101
 ~x =>  10110001
x&y =>  01000100
x|y =>  01101111
x^y =>  00101011
```

- Cute exercise. Write code to swap contents of two variables without using a third variable for temporary storage.

```
x=x^y;
y=x^y;
x=x^y;
```

- Masking — select certain bits from a word. e.g., `y=x&~0xFF` makes y be x with rightmost byte zeroed. Independent of word size!

1.9 Logical Operations in C

- `&&`, `||`, `!` are *logical* operators for AND, OR, and NOT.
- Like `&`, `|`, and `~`, but *not* bitwise; instead treat any operand other than 0 as a 1 and give a result of 0 or 1.
- Also, these operators do not evaluate the second argument if not needed to determine result.

```
x=0;
y=0&++x;
printf("%d\n",x);
```

versus

```
x=0;
y=0&&++x;
printf("%d\n",x);
```

1.10 Shift Operations in C

- Left shift: `x<<k` shifts `x` left by k bits and pads with zeros at right. This is a *logical* left shift.
- Right shift: `x>>k` shifts `x` right by k bits. For unsigned data, pad with zeros at left (*logical* style). *Usually*, for signed data, pad with whatever the leftmost bit was originally (*arithmetic* style); will make more sense when we see usual way of representing signed integers. (But C standard does not require that right shifts be arithmetic.)

- (There is also a concept of arithmetic left shift, in which the leftmost bit is unmodified, but not directly available in C.)
- Be careful about operator precedence (lower for shifts than for addition/subtraction). Also, don't use a shift of more bits than the size of the data item being shifted.