Review: Data Representation and Boolean operators in C

Based on slides © O'Hallaron
Additional material © 2020 Narahari

1

Binary Representation: Summary

- Every storage locations stores a finite sequence of bits
  - 8-bit, 16-bit, 32-bit etc.
- The same bit string can mean different things depending on how the program wants to look at it.

Address	7	6	5	4	3	2	1	0
35	1	0	0	0	0	0	0	1
36	1	0	0	0	0	1	1	1
37	1	1	1	0	0	0	0	1
38	0	1	1	0	1	1	0	1

Unsigned: +129

2C: -127

2C: 109

ASCII: 'm'

2

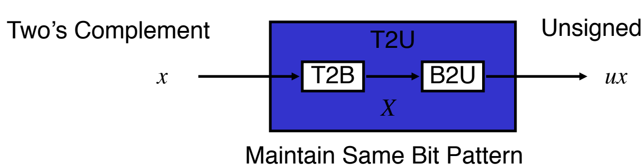
Unsigned & Signed Numeric Values

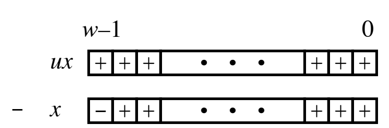
X	B2U(*X*)	B2T(*X*)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

- Equivalence
  - Same encodings for nonnegative values
- Uniqueness
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding
- ⇒ Can Invert Mappings
  - U2B(x) = B2U^{-1}(x)
    - Bit pattern for unsigned integer
  - T2B(x) = B2T^{-1}(x)
    - Bit pattern for two's comp integer

3

Relation between Signed & Unsigned



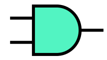
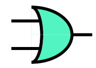
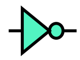
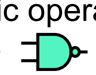
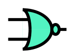


$$+2^{w-1} - -2^{w-1} = 2*2^{w-1} = 2^{w}$$

$$ux = \begin{cases} x & x \geq 0 \\ x + 2^{w} & x < 0 \end{cases}$$

4

Basic Logic Operations/Gates

- AND:
  - Equivalent notations: A AND B = A.B = $A \wedge B$ 
- OR
  - Equivalent notations: A or B = A+B = $A \vee B$ NOT 
- NOT
  - Equivalent notations: not A = A' = $\overline{A}$ 
- XOR
  - Equivalent Notations: A XOR B = A ^ B
- Other common logic operations:
  - NAND = NOT AND 
  - NOR = NOT OR 

5

Next…a little bit of "reality"

- look at how some of the concepts we have studied take shape in 'real life'
  - C programming language

6

Data Representations

- Sizes of C Objects (in Bytes)
  - C Data Type Compaq Alpha Typical Intel IA32/IA64

C Data Type	Compaq Alpha	Typical Intel IA32/IA64
int	4	4
long int	8	8
char	1	1
short	2	2
float	4	4
double	8	8
long double	8	10
char *	8	4/8 (64 bit needs 8)

- – Or any other pointer

7

Signed vs Unsigned in C

- C allows int to be defined as unsigned or signed (!!)
- Constants
  - By default are considered to be signed integers
  - Unsigned if have "U" as suffix

```
0U, 4294967259U
```

- Casting – nasty stuff!!! Or is it fun ??
  - Explicit casting between signed & unsigned same as U2T and T2U

```
int tx, ty;
unsigned ux, uy;
tx = (int) ux;
uy = (unsigned) ty;
```

  - Implicit casting also occurs via assignments and procedure calls

```
tx = ux;
uy = ty;
```

8

Casting Signed to Unsigned

- C Allows Conversions from Signed to Unsigned

```
short int          x =  15213;
unsigned short int ux = (unsigned short) x;
short int          y  = -15213;
unsigned short int uy = (unsigned short) y;
```

- Resulting Value
  - No change in bit representation
  - Nonnegative values unchanged
    - *ux* = 15213
  - Negative values change into (large) positive values
    - *uy* = 50323
- Casting Surprises in expression evaluation
  - If you mix signed and unsigned then signed cast to unsigned….and unexpected results in comparisons (>, < etc.)

9

Why Should I Use Unsigned?

- *Don't* Use Just Because Number Nonzero
  - Easy to make mistakes

```
for (i = cnt-2; i >= 0; i--)
  a[i] += a[i+1];
```

- *Do* Use When Performing Modular Arithmetic
  - Multiprecision arithmetic
  - Other esoteric stuff
- *Do* Use When Need Extra Bit's Worth of Range
  - Working right up to limit of word size

10

Logical Operations in C

- C supports both bitwise and boolean logic operations
  - x & y   bitwise logic operation
  - x && y   boolean operation: output is boolean value
- What's going on here?
  - In boolean operation the result has to be TRUE (1) or FALSE (0)
  - Treats any non-zero argument as TRUE and returns only TRUE (1) or FALSE (0)
- In C: logical operators do not evaluate their second argument if result can be obtained from first
  - a && 5/a   can we get divide by zero error?

11

Bitwise Logical Operators

- View *n*-bit number as a collection of *n* logical values
  - operation applied to each bit independently
- Number operated on is an n-bit number
- Operation being performed is logical operation on each bit
- Masking operations
  - If we are only interested in last 8 bits of a 32 bit number X, how to extract this?
  - X & 0xFF   (0xFF is notation for hex number FF)
    - Zero out the most significant 24 bits; value of least significant 8 bits is same as the value of these in X
    - xABCD27A4 & 0xFF = xA4 (in 32 bits: 0x000000A4)
  - X & 0x1 = 0 if X is even and =1 if X is odd

12

Bitwise Logical Operations

- View n-bit field as a collection of n logical values
  - Apply operation to each bit independently
- Bitwise AND: useful for clearing bits
  - AND with zero = 0
  - AND with one = no change

```
     11000101
AND  00001111
     --------
     00000101
```

- Bitwise OR: useful for setting bits
  - OR with zero = no change
  - OR with one = 1
- Computers don't support individual bits as a data type
  - Just use least significant of n-bit integer
  - Integers are generally more useful

```
     11000101
OR   00001111
     --------
     11001111
```

13

Bitwise Operators in C

- Can only be applied to integral operands
- *that is,* `char`, `short`, `int` *and* `long`
- `(signed` *or* `unsigned)`

`&`	Bitwise AND
`|`	Bitwise OR
`^`	Bitwise XOR
`<<`	Shift Left
`>>`	Shift Right
`~`	1's Complement (Inversion)

14

Bitwise AND

- Bitwise AND: 0101 AND 0110 *in C:(5 & 6)*
  - 0100
- Bitwise OR: 0101 OR 0110 *in C:(5 | 6)*
  - 0111
- Bitwise NOT: NOT 0101 in C: ~5
  - 1010
- Bitwise XOR: 0101 XOR 0110 *in C: 5^6*
  - 0011
- Bitwise NAND – no C operator, therefore
  - 0101 NAND 0110 in C: *~(5 & 6)*
- Bitwise NOR – no C operator, therefore
  - 0101 NOR 0110 in C: *~(5 | 6)*

15

Shift Operations

- x = 01100001 and y=2 (using 8-bit numbers)
- z = 10100001
- x >> y
  - x right shifted y bit positions, sign extended/arithmetic shift
    - Sign bit shifted into positions vacated by shifted bits
  - x= 01100001 y=2 (using 8-bit numbers)
  - x >> y = 00011000
  - z >> y =11101000
- x << y
  - x left shifted y bit positions, zero placed in positions vacated by shifted bits
  - x << y = 10000100
  - z << y = 10000100
- In C, x,y are 32 bit numbers:
  - What is F= (x >> 31) & 0x1

16

Boolean Relational Operators

- What is the semantics of:
  - If (x==0) then ……
  - how many outcomes for (x==0) ?
- Concept of boolean operators
  - Apply logic operators, but treat input and output as boolean variables
    - Only 1 or 0 (True or False) values for entire variable
  - But input strings can be n-bits long?
    - Treat entire string as ONE boolean variable
    - How ?

17

Logical Operations in C

- *!* Logical NOT
  - *!x*
    - !x=0 if x is non-zero, !x=1 if value of x is zero
- *&&* Logical AND
  - *x && y*
    - x && y = 1 if value of x is not zero and value of y is not zero
    - x && y = 0 if both x and y are zero
- || logical OR
  - *x* || *y*
    - x || y =1 if at least one of x,y are not zero
    - x || y = 0 if both x,y are zero

18

Examples

- `8 bit numbers, f=7, g=8`
  - `f= 00000111    g = 00001000`
- `h= (f & g) (bitwise AND)….`
  - `h= 00000000`
- `h = (f && g) (logical AND)…`
  - `h = 1`
- `!h = 0 since h is non-zero`
- `h= (f | g)  (bitwise OR)… h= ?`
- `h= (f || g) (logical OR)…. h= ?`
- `h= (~f | ~g)…h=?`
- `h= (!f && !g)…h=?`

19

Byte-Oriented Memory Organization

- Programs Refer to Virtual Addresses
  - Conceptually very large array of bytes
  - Actually implemented with hierarchy of different memory types
    - SRAM, DRAM, disk
    - Only allocate for regions actually used by program
  - In Unix and Windows, address space private to particular "process"
    - Program being executed
    - Program can clobber its own data, but not that of others
- Compiler + Run-Time System Control Allocation
  - Where different program objects should be stored
  - Multiple mechanisms: static, stack, and heap
  - In any case, all allocation within single virtual address space

20

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`
      - Or `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

21

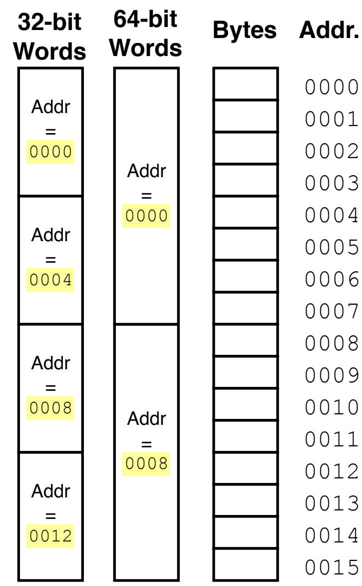
Machine Words

- Machine Has "Word Size"
  - Nominal size of integer-valued data
    - Including addresses
  - Some current machines are 32 bits (4 bytes)
    - Limits addresses to 4GB
    - Becoming too small for memory-intensive applications
  - Most (and all Higher-end) systems are 64 bits (8 bytes)
    - Potentially address $\approx 1.8 \times 10^{19}$ bytes
  - Machines support multiple data formats
    - Fractions or multiples of word size
    - Always integral number of bytes

22

Word-Oriented Memory Organization

- Addresses Specify Byte Locations
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)



23

Byte Ordering

- How should bytes within multi-byte word be ordered in memory?
- Conventions
  - PowerPC (old Mac's) are "Big Endian" machines
    - Least significant byte has highest address
    - Big end first
  - Intel x86, PC's are "Little Endian" machines
    - Least significant byte has lowest address
    - Little end first
  - Most network protocols use Big Endian
- The terms big-endian and little-endian come from Jonathan Swift's eighteenth-century satire Gulliver's Travels. The subjects of the empire of Blefuscu were divided into two factions: those who ate eggs starting from the big end and those who ate eggs starting from the little end.

24

Byte Ordering Example

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

Big Endian

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

Little Endian

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

25

Representing Integers

- `int A = 15213;`
- `int B = -15213;`
- `long int C = 15213;`

Decimal: `15213`

Binary: `0011 1011 0110 1101`

Hex: `0000 3 B 6 D`

Decimal: `-15213`

Hex: `FFFF C 4 9 3`

•Little endian layout for A:

•For B

•For C

•Big endian layout for A:

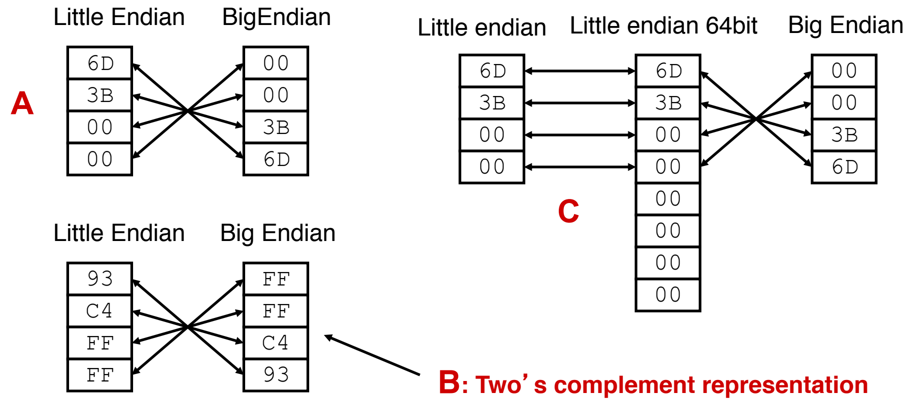
•For B:

•For C:

26

Representing Integers

- `int A = 15213;`
- `int B = -15213;`
- `long int C = 15213;`



27

Why this discussion of Bit manipulation operations in C…..Project 2!

- Project 2: Given a set of functions, each of which does not use conditional statements and implements some bit manipulation function, determine the function being implemented.
  - Rewrite the code to provide an equivalent more readable code using any C operators including conditional statements.
  - Why is this useful – some functions can be executed much quicker if they can re-written using bit manipulation operations

28