CS356: Discussion #2

Integer Operations

Marco Paolieri (paolieri@usc.edu)

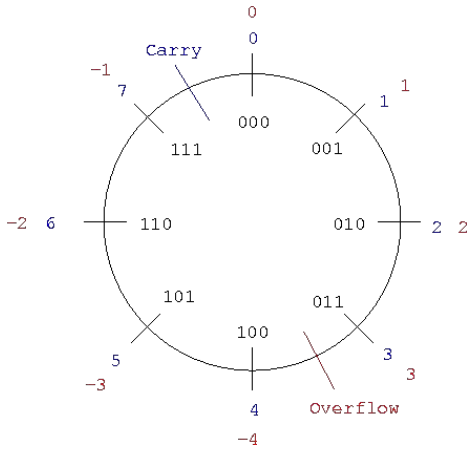
USC University of Southern California

Integers in C (64-bit architecture)

Type	Size (bytes)	Unsigned Range	Signed Range
`char`	1	0 to 255	-128 to 127
`short`	2	0 to 65535	-32,768 to 32,767
`int`	4	0 to 4G	-2G to 2G
`long`	8	0 to 18×10^{18}	-9×10^{18} to 9×10^{18}

- Rule: **0 to 2^n-1** (unsigned) and **-2^{n-1} to 2^{n-1}-1** (signed) using *n* bits
- Signed integers are represented using **2's complement**:
  `0x80 == -128, 0xFF == -1, 0x00 == 0, 0x01 == 1, 0x7F == 127`

8				0			
1	0	0	0	0	0	0	0
-128	64	32	16	8	4	2	1



Integer Operations

- **Addition** / **Subtraction** (reduces to addition using 2's complement): `+` `-`
  - Unsigned addition overflow: result smaller than inputs
  - Unsigned subtraction overflow: result larger than minuend
  - Signed addition overflow: pos + pos = neg or neg + neg = pos
- **Multiplication** / **Division**: `*` `/`
  - Product of *n*-bit numbers is at most *(2n)*-bit
  - Division of *n*-bit numbers gives *n*-bit quotient and *n*-bit remainder
- **Bitwise operations**
  - Bitwise AND (`x & mask`): clear bits that are 0 in the mask
  - Bitwise OR (`x | mask`): set bits that are 1 in the mask
  - Bitwise XOR (`x ^ mask`): flip bits that are 1 in the mask
  - Bitwise NOT (~x): flip all bits
- **Shift operations**
  - Left shift (`x << n`): fill in zeros
  - Right shift (`x >> n`): fill in zeros (unsigned) or repeat MSB (signed)

The Assignment

- **Deadline**: Next Wednesday (Sep. 5, 2018 at 11:59pm PDT)
- **Steps**
  - Read the instructions at http://bytes.usc.edu/cs356/assignments/datalab-1.pdf
  - Read the hints at ee.usc.edu/~redekopp/cs356/slides/CS356Proj1_DataLabFa18.pdf
  - Clone your class repository inside the VM
    `$ git clone git@github.com:usc-csci356-fall2018/hw-username.git`
  - Inside the file **`bits.c`**, complete the body of the functions **`bitAnd`**, **`allEvenBits`**, **`bitMask`**, **`isTmax`**, **`addOk`**, **`isGreater`**, **`satMul2`**
  - Check violations (**`./dlc bits.c`**), correctness (**`make; ./btest`**) and your final score (**`./driver.pl`**)
  - Commit, push, submit full commit hash at http://bytes.usc.edu/cs356/assignments

The Assignment: What to implement

- Only 1-byte constants (**`0xFA`**), no loops (**`for`**, **`while`**), no conditionals (**`if`**), no macros (**`INT_MAX`**), no comparisons (**`x==y`**, **`x>y`**), no **`unsigned int`**, no operators **`- && ||`**, only the operators **`! ~ & | ^ + << >>`**
- **`int bitAnd(int x, int y)`**: return `x&y` using only the operators `~ |`
- **`int allEvenBits(int x)`**: return 1 if bits 0, 2, .., 30 are 1, 0 otherwise
  - Cannot use a cycle, cannot use a 4-byte constant `0x55555555`
- **`int bitMask(int highbit, int lowbit)`**: return a 32-bit sequence with ones from **`highbit`** to **`lowbit`** (included), zeros everywhere else
- **`int isTmax(int x)`**: return 1 if x is `INT_MAX`, 0 otherwise (no `if` / macro)
- **`int addOK(int x, int y)`**: return 1 if `x+y` does not cause overflow
  - Cannot use if and comparisons like **`x >= 0 && y >= 0 && sum < 0`**
- **`int isGreater(int x, int y)`**: return `x>y` but cannot use `>`
- **`int satMul2(int x)`**: return `2*x` (no overflow), `MAX_INT` (overflow), `MIN_INT` (underflow)... but without `if` or 4-byte constants

Read the hints!

From: ee.usc.edu/~redekopp/cs356/slides/CS356Proj1_DataLabFa18.pdf

USC Viterbi School of Engineering P1.4

Hint: DeMorgan's Theorem

- DeMorgan's Theorem
  - $\neg(x \wedge y) = \neg x \vee \neg y$
  - $\neg(x \vee y) = \neg x \wedge \neg y$

Exercise: Build large constants

Write a function `int abcd()` that returns the constant `0xABCD0000`.
Use: bitwise/shift ops (`&|^~ << >>`), negation (`!`), **1-byte** const (`0x00` to `0xFF`).

```c
#include <stdio.h>
static int abcd() {
    return ((0xAB << 8) | 0xCD) << 16;
}

int main() {
    printf("%X [ABCD0000]\n", abcd());
}
/* 0x000000AB    0xAB
   0x0000AB00    0xAB << 8
   0x0000ABCD   (0xAB << 8) | 0xCD
   0xABCD0000 ((0xAB << 8) | 0xCD) << 16 */
```

Exercise: Create bit masks

Write a function `int bitMask(int highbit)` that returns a word with

- bits `[31, highbit-1]` set to `0`
- bits `[highbit, 0]` set to `1`

Use only `~ + <<` (**Hint**: `11111111 + 00000100 == 00000011`)

```c
#include <stdio.h>
static int bitMask(int highbit) {
    return ~0 + (1 << (highbit + 1));
}

int main() {
    printf("%08X [0000FFFF]\n", bitMask(15));
    printf("%08X [0001FFFF]\n", bitMask(16));
    printf("%08X [0003FFFF]\n", bitMask(17));
}
```

Exercise: Check if variable is zero

Write a function `int isZero(int x)` that returns 1 if `x==0` and 0 otherwise. Use only `!`

```c
#include <stdio.h>
static int isZero(int x) {
    return !x;
}

int main() {
    printf("%d [1]\n", isZero(0));
    printf("%d [0]\n", isZero(1));
    printf("%d [0]\n", isZero(-1));
    printf("%d [0]\n", isZero(42));
}
```

Exercise: Check if variable is non-zero

Write a function `int isNonZero(int x)` that returns 1 if `x!=0`, 0 otherwise. Use only `!`

```c
#include <stdio.h>
static int isNonZero(int x) {
    return !!x;
}

int main() {
    printf("%d [0]\n", isNonZero(0));
    printf("%d [1]\n", isNonZero(1));
    printf("%d [1]\n", isNonZero(-1));
    printf("%d [1]\n", isNonZero(42));
}
```

Exercise: Compare without ==

Write a function `int equals(int x, int y)` that returns 1 if x==y and 0 otherwise. Use only `^` and `!`.

```c
#include <stdio.h>
static int equals(int x, int y) {
    return !(x ^ y);  /* The difference is zero */
}

int main() {
    printf("%x [1]\n", equals(0, 0));
    printf("%x [1]\n", equals(1, 1));
    printf("%x [0]\n", equals(1, -1));
    printf("%x [0]\n", equals(0x7FFFFFFF, 0x80000000));
    printf("%x [0]\n", equals(0x7FFFFFFF, 0x7FFFFFFE));
}
```

Hint: Properties of `MAX_INT` (aka Tmax)

- `MAX_INT == 0x7FFFFFFF`
- Largest positive integer
  - What happens if you add 1?
  - What happens if you add it to itself?

```c
#include <stdio.h>
int main() {
    printf("%08X [80000000]\n", 0x7FFFFFFF + 1);
    printf("%08X [FFFFFFFE]\n", 0x7FFFFFFF + 0x7FFFFFFF);
}
```

- Other numbers with these properties?
- Hint
  - Use these properties to detect `MAX_INT`
  - Exclude other numbers with these properties

Exercise: Extract the last byte

Write a function `int leastSignificantByte(int x)` that returns the least significant byte of the input `x`.

Use: bitwise/shift ops (`&|^~ << >>`), negation (`!`), 1-byte const (`0x00` to `0xFF`).

```c
#include <stdio.h>

static int leastSignificantByte(int x) {
    return x & 0xFF;
}

int main() {
    printf("%08X [00000078]\n",
           leastSignificantByte(0x12345678));
}
```

Exercise: Extract the last three bits

Write a function `int lastThreeBits(int x)` that returns the last three bits of the input `x`.

Use: bitwise/shift ops (`&|^~ << >>`), negation (`!`), 1-byte const (`0x00` to `0xFF`).

```c
#include <stdio.h>

static int lastThreeBits(int x) {
    return x & 7;
}

int main() {
    printf("%x [0]\n", lastThreeBits(0x12345678));
    printf("%x [7]\n", lastThreeBits(0x1234567F));
}
```

Exercise: Extract the first bit (sign bit)

Write a function `int getFirstBit(int x)` that returns the MSB of `x`.
Use: bitwise/shift ops (`&|^~ << >>`), negation (`!`), 1-byte const (`0x00` to `0xFF`).

```c
#include <stdio.h>

static int getFirstBit(int x) {
    return (x >> 31) & 1;
}

int main() {
    printf("%x [1]\n", getFirstBit(0xFFFFFFFF));
    printf("%x [1]\n", getFirstBit(0x8FFABCDE));
    printf("%x [0]\n", getFirstBit(0x7FFFFFFF));
}
```

Exercise: Check if numbers have same sign

Write a function `int sameSign(int x, int y)` that returns 1 if `x` and `y` have the same sign.
Use: bitwise/shift ops (`&|^~ << >>`), negation (`!`), 1-byte const (`0x00` to `0xFF`).

```c
#include <stdio.h>
static int sameSign(int x, int y) {
    return ((x >> 31) & 1) ^ ((y >> 31) & 1) ^ 1;
}
int main() {
    printf("%x [1]\n", sameSign(-1, -3));
    printf("%x [1]\n", sameSign(0, 1));
    printf("%x [1]\n", sameSign(0x80000000, 0x80000000));
    printf("%x [1]\n", sameSign(0x7FFFFFFF, 0x7FFFFFFF));
    printf("%x [0]\n", sameSign(0x7FFFFFFF, 0x80000000));
    printf("%x [0]\n", sameSign(-1, 0));
}
```

Variation

- Can we reduce the number of operations?
- The solution

```
((x >> 31) & 1) ^ ((y >> 31) & 1) ^ 1
```

is equivalent to

```
(((x ^ y) >> 31) & 1) ^ 1
```

Exercise: Extract the byte after the bit sign

Write a function `int getBits23to30(int x)` that returns the byte starting after the first bit of `x`.

Use: bitwise/shift ops (`&|^~ << >>`), negation (`!`), 1-byte const (`0x00` to `0xFF`).

```c
#include <stdio.h>

static int getBits23to30(int x) {
    return (x >> 23) & 0xFF;
}

int main() {
    printf("%X [E0]\n", getBits23to30(0xF00ABCDE));
    printf("%X [1F]\n", getBits23to30(0x0FABCDEF));
}
```

Exercise: Negate without -

Write a function `int negate(int x)` that returns the two's complement of `x`. Use only ~ and +.

```c
#include <stdio.h>
static int negate(int x) {
    return ~x + 1;
}

int main() {
    printf("%d [0]\n", negate(0));
    printf("%d [-1]\n", negate(1));
    printf("%d [1]\n", negate(-1));
    printf("%x [80000001]\n", negate(0x7FFFFFFF));
    printf("%x [80000000]\n", negate(0x80000000));  // overflow
}
```

Hint: Is `(x > y)` the same as `(x-y > 0)`?

- Can you compute `-y` without the `-` operator?
- Can you check if `x-y` is positive?
- Is `(x > y)` the same as `(x-y > 0)`?
  - No! There can be overflow in `x-y` so that the sign becomes wrong...
- Can you detect the overflow?
  - `pos + pos = neg`
  - `neg + neg = pos`

Exercise: Conditionals without `if`

Write a function `int negOrElse(int x, int y)` that returns

- x if `(x < 0)`
- y if `(x >= 0)`

Use only `>>` `~` `&` `|`

```
#include <stdio.h>
static int negOrElse(int x, int y) {
    int isNeg = x >> 31;  /* 0xFFFFFFFF or 0x00000000 */
    return  (isNeg & x) | (~isNeg & y);
}
int main() {
    printf("%d [-1]\n", negOrElse(-1, 42));
    printf("%d [42]\n", negOrElse(1, 42));
    printf("%d [42]\n", negOrElse(0, 42));
    printf("%d [42]\n", negOrElse(0x7FFFFFFF, 42));
    printf("%x [80000000]\n", negOrElse(0x80000000, 42));
}
```

Exercise: Multiply using shifts

Write a function `void mult(int x)` that multiplies x

- by 6, using 2 shifts and 1 add/sub;
- by 31, using 1 shifts and 1 add/sub;
- by -6, using 2 shifts and 1 add/sub;
- by 55, using 2 shifts and 2 add/sub.

```
#include <stdio.h>
static void mult(int x) {  printf("\nx = %d\n", x);
    printf(" 6 * x = (8-2) * x = %d\n", (x << 3) - (x << 1));
    printf("31 * x = (32-1) * x = %d\n", (x << 5) - x);
    printf("-6 * x = (2-8) * x = %d\n", (x << 1) - (x << 3));
    printf("55 * x = (64-8-1) * x = %d\n", (x << 6)-(x << 3)-x);
}
int main() {
    mult(0); mult(1); mult(-1); mult(10); mult(-100); mult(7);
}
```