Data Representation I

Integers

Connor Baker

Compiled on September 14, 2019 at 10:55am

Reading Track

This Lecture covers Chapters 2.2 and 2.3 (Integers and Integer Arithmetic)

Reading Homework:

1. Read through Chapter 2.4 (Floating Point)
    - Also continue to review Chapter 2.1-2.3 in its entirety
    - We'll spend the next several classes finishing up through Chapter 2.4
2. Continue to Review C. This semester, you'll be doing the following operations in Assembly Language. If you know these well in C, it will be a lot easier over the next two months:
    - Passing by reference and working with pointers in functions
    - Dereferencing values: `*ptr = 42;`
    - Pointer arithmetic with arrays: `*(ary + 3)= 42;`
    - Working with `struct`s and dynamic memory
    - Working with linked lists using dynamically allocated `struct`s

Lecture Overview from CS262

Bits and Ints	Describe the Limits on Integers in C	Describe how Integers are Represented in Memory	2.2
		Describe the concept of Overflow for Signed and Unsigned Integers	
		Describe the Range for Signed and Insigned Integers given Size in Bytes	
		Describe how to Access the Range of Integers in C using limits.h	
		Describe Converting Unsigned to Signed Integers of Varying Sizes	
		Describe Implicit Conversion on Expressions involving FP Values in C	(+ 2.4.6)
	Perform Arithmetic Operations on Integers by Constants	Perform Left and Right Shifts on an Integer	2.1.9
		Multiple an Integer by Shifts and Arithmetic	2.3
		Divide an Integer Properly using Shifts and Arithmetic	
		Describe the Dangers of Right Shifting in C	

Figure 1: Lecture overview.

Note on Endian-ness and Operations

For most applications that we will write in this class, the Endian-ness of a machine isn't something that we'll need to concern ourselves with.

Operations, even when done in C or Assembly, operate on a multi-byte integers as we would expect them to, if they were just large binary values.

Knowing about Endian-ness is only needed when we want to examine how the CPU stores integers in memory, looking at it byte by byte.

Integer Representation

Unsigned Integers

With unsigned integers, all bits have the same magnitude that they do in binary. We can use notation to explain this:

$$\sum_{i=0}^{\omega-1} x_2 \times 2^i$$

where ω is the number of bits, and x_i is the bit at position i (zero-indexed from the Least Significant Bit (LSB)).

Signed Integers

Most machines use two's complement when they encode bits.

With the two's complement, all bits have the same magnitude as they would in binary. However, the Most Significant Bit (MSB) of the data type has a negative value. Mathematically,

$$-x_{\omega-1} \times 2^{\omega-1} + \sum_{i=0}^{\omega-2} x_2 \times 2^i$$

where ω is the number of bits, and x_i is the bit at position i (zero-indexed from the LSB).

Question 1

Unsigned 8-bit binary integers to decimal:

Note	Binary	Decimal
UMIN	00000000	0
None	10000000	128
None	01111111	127
UMAX	11111111	255

Question 2

Unsigned 8-bit binary integers to decimal:

Note	Binary	Decimal
None	`00000000`	0
`TMIN`	`10000000`	-128
`TMAX`	`01111111`	127
None	`11111111`	-1

Adding one repeatedly

Let's consider what happens when we repeatedly add one to an 8-bit number, looking at it as both a signed and unsigned integer.

Unsigned	Binary	Signed
0	`00000000`	0
1	`00000001`	1
⋮	⋮	⋮
127	`01111111`	127
128	`10000000`	-128
⋮	⋮	⋮
255	`11111111`	-1

Integer Ranges

In general, with an n-bit data type, you have an unsigned range of

$$[0, 2^n - 1]$$

and a signed range of

$$[2^{n-1}, 2^{n-1} - 1]$$

with two's complement.

The limits for different data types are in `limits.h` and `stdint.h`.

Negating a Number

With signed integers in two's complement, we can negate values using the following process:

- Flip all of the bits
- Add one

Take for example the following snippet:

```
char x = 5; // x = 00000101
x = ~x + 1; // x = 11111011
x = ~x + 1; // x = 00000101
```

Performing the operation twice returns us to our original value.

Unsigned and Signed Values

Omitted slide 15.

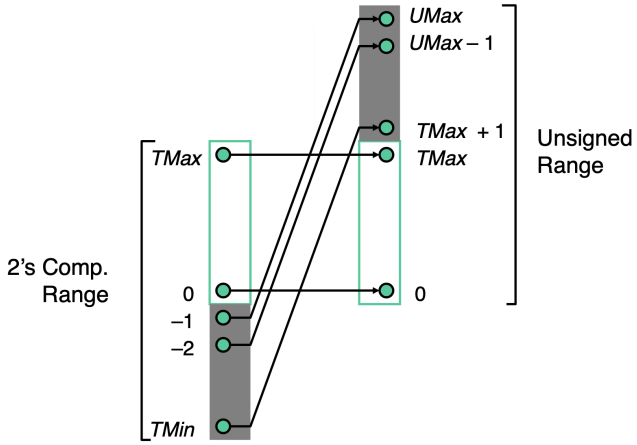

Figure 2: Mapping between the two types.

Two's complement to unsigned conversion

Note: Converting between signed and unsigned does not change the bits – it's really just telling the program how to interpret that region of memory.

$$T2U(x) = \begin{cases} x, & x \geq 0 \\ x + w^{\omega}, & x < 0 \end{cases}$$

Example: Convert 8-bit value `-1` to an unsigned number.

Since $x < 0$, we use $T2U(x) = x + 2^{\omega}$. Substituting, $T2U(-1) = -1 + 256 = 255$.

Unsigned to two's complement conversion

$$U2T(u) = -u_{\omega-1} \times 2^{\omega} + u$$

Example: Convert 8-bit value `255` to an signed number.

Evaluating $U2T(255)$ yields $-256 + 255$ which is equivalent to -1.

Signed and Unsigned in C

Constants

- Constant values in code are treated as signed integers.
  - To treat them as unsigned, add the `U` suffix:
    * `x += 429496725;` becomes `x += 429496725U;`

Casting

- You can explicitly cast between signed and unsigned data types
- This is is the same thing as using $U2T$ or $T2U$: the bits are unchanged
  - `uy = (unsigned int)ty;`
- C will also implicitly cast if needed
  - `int tx = uy; //may throw compiler warning`

Casting Surprises

File: `casting.c`

Question 3

Consider the following table (the constants are all 32-bit integers, so `TMAX = 2147483647` and `TMIN = -2147483648`).

Constant 1	Constant 2	Signed or Unsigned Evaluation?	Relation? ==, <, or >
0	0U	Unsigned	Constant 1 == Constant 2
-1	0	Signed	Constant 1 < Constant 2
-1	0U	Unsigned	Constant 1 > Constant 2
2147483647	-2147483648	Signed	Constant 1 > Constant 2
2147483647U	-2147483648	Unsigned	Constant 1 < Constant 2
-1	-2	Signed	Constant 1 > Constant 2
(unsigned) -1	-2	Unsigned	Constant 1 > Constant 2
2147483647	2147483648U	Unsigned	Constant 1 < Constant 2
2147483647	(int) 2147483648U	Signed	Constant 1 > Constant 2

Zero Extension

When we cast an unsigned integral data type to a larger unsigned integral data type it is padded with leading zeros.

This does not change the value.

Signed Extension

When we cast a signed integral data type to a larger signed integral data type it is padded with whatever the MSB was.

This does not change the value.

Omitted slide 23.

File: `sign.c`

As an example, consider the following C snippet:

```c
short x = 15213; // 0x3b6d
int ix = (int) x;
short y = -15213; // 0xc493
int iy = (int) y;
```

Type	Decimal	Hex	Binary
`short x`	15213	3B 6D	00111011 01101101
`int ix`	15213	00 00 3B 6D	00000000 00000000 00111011 01101101

Type	Decimal	Hex	Binary
`short y`	-15213	C4 93	11000100 10010011
`int iy`	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

Running it yields

```
$ ./sign 6d 3b
6d 3b 00 00
93 c4
93 c4 ff ff
```

Truncation

- The opposite of extension
- Reduces an integer to a smaller data type
- Chops off the higher-order bits
- This process can change the value if the new size is too small

When to use Unsigned

File: `mod.c`

- Essentially only use it if you're
  - Performing modular arithmetic
  - When you absolutely need an extra bit worth of range

Unsigned Integer Addition

Modular arithmetic

The sum of two unsigned value may overflow. It is in effect a modular sum: $u + v = (u + v) \mod 2^{\omega}$. It will wrap at most once. It follow that the true sum will require at most $\omega + 1$ bits to represent. The discarded bit is called the *carry*.

$$UAdd_{\omega}(u, v) = \begin{cases} u + v, & u + v < 2^{\omega} \\ u + v - 2^{\omega}, & u + v \geq 2^{\omega} \end{cases}$$

The result of the $UAdd$ function looks like modulo-ω applied to $z = x + y$ in the first octant of $\mathbb{R}^3$.

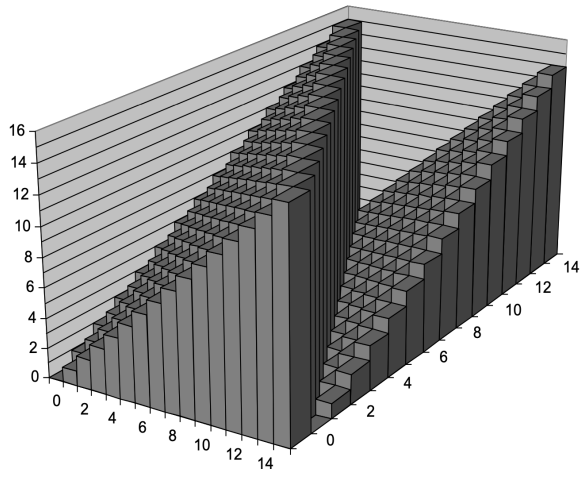

Figure 3: $UAdd_4(u, v)$.

Detecting Overflow in Unsigned Addition

If $u + v < u$ or $u + v < v$ then you had an overflow.

Two's Complement Integer Addition

Essentially, we see the same behavior as we did with unsigned integer addition. We still have the carry, and the true sum still requires at most $\omega + 1$ bits. This is largely due to the fact that the Arithmetic Logic Unit (ALU) only sees bits and does not interpret them.

Note: If you add a positive and a negative number, you'll never have an overflow.

Characterizing Signed Addition

Functionality

- The true sum requires up to $\omega + 1$ bits
- This means that MSB may be truncated

- The remaining bits are treated as signed bits

$$TAdd_\omega = \begin{cases} u + v - 2^\omega, & TMax_\omega < u + v \\ u + v, & TMin_\omega \leq u + v \geq TMax_\omega \\ u + v + 2^\omega, & u + v < TMin_\omega \end{cases}$$

- In the first case, we handle positive overflow (positive plus positive yields negative)
- In the last case, we handle negative overflow (negative plus negative yields positive)

Visualizing Two's Complement Addition

Consider what happens with a 4-bit data type.

- Range of [-8,7]
- If $u + v \geq 2^{\omega-1}$ (greater than `TMax`) it becomes negative
- If $u + v < -2^{\omega-1}$ (less than `TMin`) it becomes positive (or zero)

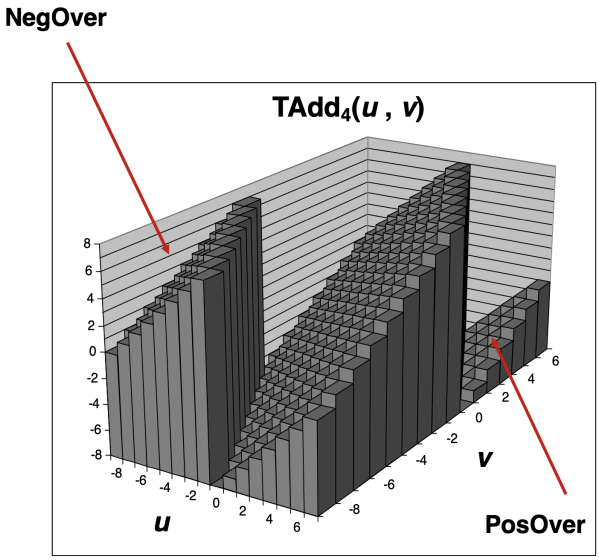

Figure 4: $TAdd_4(u, v)$.

Detecting Signed Overflow

Type of Overflow	Criteria
Negative Overflow	$u < 0, v < 0, u + v \geq 0$
Positive Overflow	$u \geq 0, v \geq 0, u + v <> 0$
Overflow	Operands have the same sign, but the sum is a different sign

We can express the last one as a boolean statement in C:

```
(u < 0 == v < 0) && (u < 0 != sum < 0)
```