# CSE 220: Systems Programming

## Integers and Integer Representation

Ethan Blanton

Department of Computer Science and Engineering
University at Buffalo

# Integer Complications

It seems like integers should be simple.

However, there are complications.

- Different machines use different size integers
- There are multiple possible representations
- *etc.*

In this lecture, we will explore some of these issues in C.


©2020 Ethan Blanton / CSE 220: Systems Programming

# Memory as Bytes

Previously, I said “To the computer, memory is just bytes.”

While this isn’t precisely true, it’s close enough to get started.

The computer doesn’t “know” about data types.
A modern processor can probably directly manipulate:

- Integers (maybe only of a single bit length!)
- Maybe floating point numbers
- …often, that’s all!

Everything else we create in software.


©2020 Ethan Blanton / CSE 220: Systems Programming

# Memory as …Words?

It is probably more accurate to say memory is just words.

What is a word?

A word is the native integer size of a given platform.
For example, 64 bits on x86-64, or 32 bits on an ARM Cortex-A32.

A word can also (confusingly) be the width of the memory bus, if the processor's word size and its memory bus width are different.

- We will assume they are the same, at least for a while.

What is "native integer size"? What is the "width" of a memory bus?


©2020 Ethan Blanton / CSE 220: Systems Programming

# Imposing Structure on Memory

That said, programming languages expose things like:

- Booleans
- classes
- strings
- structures

How is that?

We impose meaning on words in memory by convention.

*E.g.*, as we saw before, a C string is a sequence of bytes that happen to be adjacent in memory.


©2020 Ethan Blanton / CSE 220: Systems Programming

# Hexadecimal

A brief aside: we will be using hexadecimal ("hex") a *lot*.

Hex is the base 16 numbering system.
One hex digit ranges from 0 to 15.

Contrast this to decimal, or base 10 —
one decimal digit ranges from 0 to 9.


©2020 Ethan Blanton / CSE 220: Systems Programming

# Hexadecimal

A brief aside: we will be using hexadecimal ("hex") a *lot*.

Hex is the base 16 numbering system.
One hex digit ranges from 0 to 15.

Contrast this to decimal, or base 10 —
one decimal digit ranges from 0 to 9.

In computing, hex digits are represented by 0-9 and then A-F.

| | |
|---|---|
| A = 10 | D = 13 |
| B = 11 | E = 14 |
| C = 12 | F = 15 |


©2020 Ethan Blanton / CSE 220: Systems Programming

# Why Hex?

Hexadecimal is used because one hex digit is four bits.

This means that two hex digits represents one 8-bit byte.

On machines with 8-bit-divisible words, this is *very convenient*.

| Hex | Bin | Hex | Bin |
|---|---|---|---|
| 0 | 0000 | 8 | 1000 |
| 1 | 0001 | 9 | 1001 |
| 2 | 0010 | A | 1010 |
| 3 | 0011 | B | 1011 |
| 4 | 0100 | C | 1100 |
| 5 | 0101 | D | 1101 |
| 6 | 0110 | E | 1110 |
| 7 | 0111 | F | 1111 |


©2020 Ethan Blanton / CSE 220: Systems Programming

# Integer Types

Platform-specific integer types you should know:

- `char`: One character.
- `short`: A short (small) integer
- `int`: An “optimally sized” integer
- `long`: A longer (bigger) integer
- `long long`: An *even longer* integer

Their sizes are: 8 bits $\leq$ `char` $\leq$ `short` $\leq$ `int` $\leq$ `long` $\leq$ `long long`

Furthermore:
`short`, `int` $\geq$ 16 bits, `long` $\geq$ 32 bits, `long long` $\geq$ 64 bits

*Whew!*


©2020 Ethan Blanton / CSE 220: Systems Programming

# Integer Modifiers

Every integer type may have modifiers.

Those modifiers include `signed` and `unsigned`.

All unmodified integer types *except* `char` are signed.
`char` may be signed or unsigned!

The keyword `int` may be elided for any type except `int`.
These two declarations are equivalent:

```
long long nanoseconds;
signed long long int nanoseconds;
```


©2020 Ethan Blanton / CSE 220: Systems Programming

# Integers of Explicit Size

The confusion of sizes has led to explicitly sized integers.

They live in `<stdint.h>`

Exact-width types are of the form `intN_t`.
They are exactly *N* bits wide; *e.g.*: `int32_t`.

Minimum-width types are of the form `int_leastN_t`.
They are at least *N* bits wide.

There are also unsigned equivalent types, which start with `u`:
`uint32_t`, `uint_least8_t`

*N* may be: 8, 16, 32, 64.

# `dump_mem()`

In the following slides, we will use the function `dump_mem()`.

We will examine it in detail at some point, but for now:

- `dump_mem()` receives a memory address and number of bytes
- It then prints the hex values of the bytes at that address

Don't worry too much about its details for now.


©2020 Ethan Blanton / CSE 220: Systems Programming

# A Simple Integer

First, a simple integer:

```
int x = 98303; // 0x17fff
dump_mem(&x, sizeof(x));
```


©2020 Ethan Blanton / CSE 220: Systems Programming

# A Simple Integer

First, a simple integer:

```
int x = 98303; // 0x17fff
dump_mem(&x, sizeof(x));
```

Output:

```
ff 7f 01 00
```

Let's pull this apart.


©2020 Ethan Blanton / CSE 220: Systems Programming

# Byte Ordering

Why is 98303, which is `0x17fff`, represented by `ff 7f 01 00`?

# Byte Ordering

Why is 98303, which is `0x17fff`, represented by `ff 7f 01 00`?

The answer is endianness.

Words are organized into bytes in memory — but in what order?

- Big Endian: The "big end" comes first.
  This is how we write numbers.
- Little Endian: The "little end" comes first.
  This is how x86 processors (and others) represent integers.

You cannot assume anything about byte order in C!


© 2020 Ethan Blanton / CSE 220: Systems Programming

# Sign Extension

```
char c = 0x80;
int  i = c;

dump_mem(&i, sizeof(i));
```

# Sign Extension

```
char c = 0x80;
int  i = c;

dump_mem(&i, sizeof(i));
```

Output:

```
80 ff ff ff
```

`0xffffff80`? Where did all those one bits come from?!

# Positive Integers

A formal definition of a positive integer on a modern machine is:

Consider an integer of width $w$ as a vector of bits, $\vec{x}$:

$$\vec{x} = x_{w-1}, x_{w-2}, \ldots, x_0$$

This vector $\vec{x}$ has the decimal value:

$$\vec{x} \doteq \sum_{i=0}^{w-1} x_i 2^i$$


©2020 Ethan Blanton / CSE 220: Systems Programming

# Calculating Integer Values

Consider the 8-bit binary integer `0010 1011`:

$$
\begin{aligned}
0010\,1011\text{b} &= 0 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\
&= 0 \cdot 128 + 0 \cdot 64 + 1 \cdot 32 + 0 \cdot 16 + 1 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 1 \cdot 1 \\
&= 32 + 8 + 2 + 1 \\
&= 43
\end{aligned}
$$


©2020 Ethan Blanton / CSE 220: Systems Programming

# Negative Integers

Previously, the variable `c` was sign extended into `i`.

As previously discussed, integers may be signed or unsigned.

Since integers are just bits, the negative numbers must have different bits set than their positive counterparts.

There are several typical ways to represent this, the most common being:

- One’s complement
- Two’s complement


© 2020 Ethan Blanton / CSE 220: Systems Programming

# One's Complement

One's complement integers represent a negative by inverting the bit pattern.

Thus, a 32-bit 1:
`00000000 00000000 00000000 00000001`

And a 32-bit -1:
`11111111 11111111 11111111 11111110`

Formally, this is like a positive integer, except:

$$x_{w-1} \doteq -2^{w-1} + 1$$


©2020 Ethan Blanton / CSE 220: Systems Programming

# Decoding Negative One's Complement

Therefore, 4-bit −1: `1110`

$$\begin{aligned} 1110b &= 1 \cdot (-2^3 + 1) + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \\ &= 1 \cdot -7 + 1 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 \\ &= -7 + 4 + 2 \\ &= -1 \end{aligned}$$

# Decoding Negative One's Complement

Therefore, 4-bit −1: `1110`

$$\begin{aligned}
1110b &= 1 \cdot (-2^3 + 1) + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \\
&= 1 \cdot -7 + 1 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 \\
&= -7 + 4 + 2 \\
&= -1
\end{aligned}$$

This is fine, except there are two zeroes!:

$$\begin{aligned}
0000b &= 0 \cdot (-2^3 + 1) + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 \\
1111b &= 1 \cdot -2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\
&= -7 + 4 + 2 + 1
\end{aligned}$$


©2020 Ethan Blanton / CSE 220: Systems Programming

# Two's Complement

Most (modern) machines use two's complement.

Two's complement differs *slightly* from one's complement.
Its $w - 1$th bit is defined as:

$$x_{w-1} \doteq -2^{w-1}$$

(Recall that one's complement added 1 to this!)

This means there is only one zero — all 1s is -1!

# Decoding Two's Complement

Consider 1110 in two's complement:

$$\begin{aligned} 1110\text{b} &= 1 \cdot -2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \\ &= -8 + 4 + 2 + 0 \\ &= -2 \end{aligned}$$


©2020 Ethan Blanton / CSE 220: Systems Programming

# Decoding Two's Complement

Consider 1110 in two's complement:

$$\begin{aligned}
1110\text{b} &= 1 \cdot -2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \\
&= -8 + 4 + 2 + 0 \\
&= -2
\end{aligned}$$

$w$-bit Two's complement integers run from $-2^{w-1}$ to $2^{w-1} - 1$.


©2020 Ethan Blanton / CSE 220: Systems Programming

# Negative Integer Bit Patterns

*In general*, the high-order bit of a negative integer is 1.

In our previous example:

```
char c = 0x80;
int  i = c;
```

c is signed, and thus equivalent to -128.


©2020 Ethan Blanton / CSE 220: Systems Programming

# Negative Integer Bit Patterns

*In general*, the high-order bit of a negative integer is 1.

In our previous example:

```
char c = 0x80;
int  i = c;
```

`c` is signed, and thus equivalent to -128.

It is then sign extended into `i` by duplicating the high bit to the left.

This results in an `i` that also equals -128.

Why?


©2020 Ethan Blanton / CSE 220: Systems Programming

# Computing `c` and `i`

```
char c = 0x80;
```

Here, `c` is -128 plus no other bits set.

```
int i = c;
```

What is `i` if we sign extend?


©2020 Ethan Blanton / CSE 220: Systems Programming

# Computing `c` and `i`

```
char c = 0x80;
```

Here, `c` is -128 plus no other bits set.

```
int i = c;
```

What is `i` if we sign extend?

```
11111111 11111111 11111111 10000000
```

What is the value of that two's complement integer?


©2020 Ethan Blanton / CSE 220: Systems Programming

# Computing Sign Extension

```
11111111 11111111 11111111 10000000
```

Remember that the high 1 bit indicates $-2^{w-1}$, or $-2^{31}$, here.


©2020 Ethan Blanton / CSE 220: Systems Programming

# Computing Sign Extension

`11111111 11111111 11111111 10000000`

Remember that the high 1 bit indicates $-2^{w-1}$, or $-2^{31}$, here.

We then add in each of the other bits as positive values.

Every bit from $2^7$ through $2^{30}$ is set, and $2^0$ through $2^6$ are unset:

$$-2^{31} + 2^{30} + 2^{29} + \ldots + 2^8 + 2^7$$

# Computing Sign Extension

```
11111111 11111111 11111111 10000000
```

Remember that the high 1 bit indicates $-2^{w-1}$, or $-2^{31}$, here.

We then add in each of the other bits as positive values.

Every bit from $2^7$ through $2^{30}$ is set, and $2^0$ through $2^6$ are unset:

$$-2^{31} + 2^{30} + 2^{29} + \ldots + 2^8 + 2^7$$

…this sums to -128!

# Summary

- The CPU and memory deal only in words
- Buses and registers have native word widths
- Integers have different:
  - Bit widths
  - Endianness
  - Sign representation
- One’s and two’s complement representation


©2020 Ethan Blanton / CSE 220: Systems Programming

# References I

## Required Readings

[1] Randal E. Bryant and David R. O'Hallaron. *Computer Science: A Programmer's Perspective*. Third Edition. Chapter 2: Intro, 2.1 through 2.1.3, 2.2. Pearson, 2016.


©2020 Ethan Blanton / CSE 220: Systems Programming

# License

Copyright 2019, 2020 Ethan Blanton, All Rights Reserved.
Copyright 2019 Karthik Dantu, All Rights Reserved.

Reproduction of this material without written consent of the author is prohibited.

To retrieve a copy of this material, or related materials, see `https://www.cse.buffalo.edu/~eblanton/`.