Representation of Integers (lecture)

ICS312

Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

Integer Representation

- A computer needs to store integers
- Stored using different numbers of bytes (1 byte = 8 bits)
- Most high-level programming languages all define integer data types

#bytes	C/C++	Java	Rust	Go	Python
1	char unsigned char	byte	i8 u8	int8 uint8	-
2	short unsigned short	short	i16 u16	int16 uint16	-
4	int unsigned int	int	i32 u32	int32 uint32	-
8	long* unsigned long*	long	i64 u64	int64 uint64	-
∞	-	-	-	-	int

* Only 4 bytes on a 32-bit computer!

Math vs. Computer Arithmetic

- In math, we add two numbers and the result is whatever it is
- In computer arithmetic, we **must specify the size** of the numbers on which we perform the operation
- From now on, we will always state the size of the operands
- For example:
  - we'll never say "let's add 4 and 31"
  - Instead, we'll say "let's add 2-byte value 4 to 2-byte value 31"
- Also, in computer arithmetic, **the result has a specified size**
- For many operations, the size of the result must be the same as the size of the operand
  - For instance, the addition of two 2-byte values always produces a 2-byte value
- Let's see this on an example…

Math vs. Computer Arithmetic

- Let's consider the addition of two 16-bit values

Math	Computer Arithmetic
F111	F111
+ 6111	+ 6111
15222	5222

- In math, the result is a 17-bit number
- In computer arithmetic the leftover carry is dropped and the result is a 16-bit number

But then.... is it correct?

```
    F111
+   6111
  ------
    5222
```

- Is “dropping the leftover carry” even correct?
- Turns out, sometimes it is, sometimes it isn’t!
- More on this later this semester, but does anybody know how we call the situation in which the result is incorrect?

Unsigned and Signed Numbers

- So how do we encode integers in binary?

- Integers come in two flavors:
  - **Unsigned**: **ONLY positive**
  - **Signed**: **positive AND negative** values, with about the same number of negative values as the number of positive values

- This is why in the table on the first slide, there were “unsigned” / “u” versions of the data types for some of the languages

Unsigned / Signed Integers

- With the languages that support signed and unsigned integers you can pick what to use depending on what you need
  - If you know a variable will only be positive, then you have a higher maximum value when using unsigned
  - The compiler/IDE helps by throwing warnings when you, e.g., assign a negative value to an unsigned number

```
// C/C++
int x = -12;                // signed
signed int y = 40;          // signed
unsigned int z = 23;        // unsigned
```

```
// Rust
let x: i16 = -12;    // signed 16-bit
let y: u32 = 2_031;  // unsigned 32-bit
                     // (note the convenient underscore
                     //  that acts as a comma)
```

Unsigned / Signed Integers

```
// C/C++
int x = -12;             // signed
signed int y = 40;       // signed
unsigned int z = 40;     // unsigned
```

```
// Rust
let x: i16 = -12;    // signed
let y: u32 = 2_031;  // unsigned
```

- In Java, Python, JavaScript all integers are signed (there is no unsigned data type), which has raised A LOT of complaints
- But these languages have APIs to perform unsigned arithmetic

```
// Java
Integer.divideUnsigned(-100, -12) // divide as if numbers were unsigned

// Python
ctypes.c_uint32(-10).value        // interpret -10 as unsigned (32-bit)

// JavaScript
x = -10 >>> 0                     // interpret -10 as unsigned
```

- The code above is likely confusing right now because we don't know yet how we encode signed/unsigned integers in binary… stay tuned (we'll come back to these three examples!)

Encoding Unsigned Integers

- **Encoding unsigned integers is easy:** just use the bits of the integer's binary representation
- Example: 1-byte unsigned number 33_{10} is encoded as 00100001_2 (or 21_{16})

- **That's all!**

- Just note that we show exactly 8 bits, which may include the leading zeros
  - In the world of compute arithmetic we always show leading zeros
- So, if I ask "what's zero in binary as an 8-bit number?" the answer is 00000000, not just 0

Encoding Signed Integers

- **Encoding signed integer raises a question:** how to store the sign?
- One possible approach is called sign-magnitude: reserve the leftmost bit to represent the sign

  00100101 denotes $+\ 0100101_2$

  10100101 denotes $-\ 0100101_2$

- It's very easy to negate a number: just flip the leftmost bit
- Unfortunately, sign-magnitude complicates the logic in the CPU
  - There are two representations for zero: 10000000 and 00000000
  - Some operations are thus more complicated to implement in hardware
  - See a computer architecture / engineering course

One’s complement

- Another idea to encode a negative number is to take the complement (i.e., flip all bits) of its positive counterpart
- Example: I want to encode integer -87
  - $87_{10} = 01010111_2$
  - $-87_{10} = 10101000_2$
- Simple, but still two representations for zero: 00000000 and 11111111
- It turns out that computer logic to deal with 1’s complement arithmetic is complicated
- **Important:** It’s easy to compute the 1’s complement of a number represented in hexadecimal
  - example: 57_{16}
  - subtract each hex digit from F: F-5=A F-7=8
  - 1’s complement of 57_{16} is $A8_{16}$

2’s complement

- Nowadays all computers use 2’s complement to encode signed integers
- The most significant bit indicates the sign of a number
- The 2’s complement representation of a **positive** number is simply the number’s binary representation
  - But the most significant bit must be a zero
  - Which is why we have half as many positive signed numbers as we have unsigned numbers
- The 2’s complement representation of a **negative** number is done in **two steps** (**“flip and add one”**)
  - Step 1: Compute the 1’s complement of the positive version of the number
  - Step 2: Add 1 to the result
- In fact, the “flip add one” transformation negates a signed number

2's Complement: example

- Let's encode -87_{10}
  - First, start with the positive version of the number: $87_{10} = 57_{16}$
  - "Flip" the bits or hex digits to compute the one's complement: $A8_{16}$
  - Add one: $A9_{16}$
- Let's invert again to check we get back to the positive number
  - We start with: $A9_{16}$
  - Flip the digits (one's complement): 56_{16}
  - Add one: 57_{16}, which represents 87_{10}
- So, when I write, say in C++, `char x = -87;` somewhere the value A9 (or 10101001) is stored

2’s Complement: 0 and -1

- **There is a single representation for zero!**
  - Assuming 8-bit signed numbers, zero is 0000 0000
  - Let’s compute -0:
    - Flip: 1111 1111
    - Add one: 1 0000 0000 (9 bits!!)
  - BUT, when adding two X-bit quantities in a computer one **always** obtains another X-bit quantity
  - Again, the computer DROPS the extra carry because it doesn’t “fit”
  - Final result: 0000 0000
  - And so, there is a single representation for 0 (unlike for 1’s complement)

- **-1’s representation is all bits set to 1**
  - +1 is represented as 0000 0001
  - Flip: 1111 1110
  - Add one: 1111 1111

How to tell the sign of a signed integer?

- All programming languages support signed integers (as far as I know)
- A very common need is to determine whether a signed value is positive or negative
  - Whenever I write code like: `if (x > 0) {…}` then the compiler has to generate code that does the test

- **Remember that the most significant bit (the leftmost bit) indicates the sign of the number (0: positive, 1: negative)**
  - In hex, if the left-most “digit” is 8, 9, A, B, C, D, E, or F, then the number is negative, otherwise it is positive
  - Those are the hex digit of the form 1xxx in binary

- Let’s look at ALL 3-bit unsigned and signed numbers….

All 3-bit unsigned/signed numbers

UNSIGNED	
Decimal	Representation
0	`000`
1	`001`
2	`010`
3	`011`
4	`100`
5	`101`
6	`110`
7	`111`

SIGNED	
Decimal	Representation
0	`000`
1	`001`
2	`010`
3	`011`
-4	`100`
-3	`101`
-2	`110`
-1	`111`

All 3-bit unsigned/signed numbers

UNSIGNED	
Decimal	Representation
0	`000`
1	`001`
2	`010`
	`011`
4	`100`
5	`101`
6	`110`
7	`111`

SIGNED	
Decimal	Representation
0	`000`
	`001`
	`010`
3	`011`
-4	`100`
	`101`
	`110`
-1	`111`

Smallest number: 000
Largest number: 111

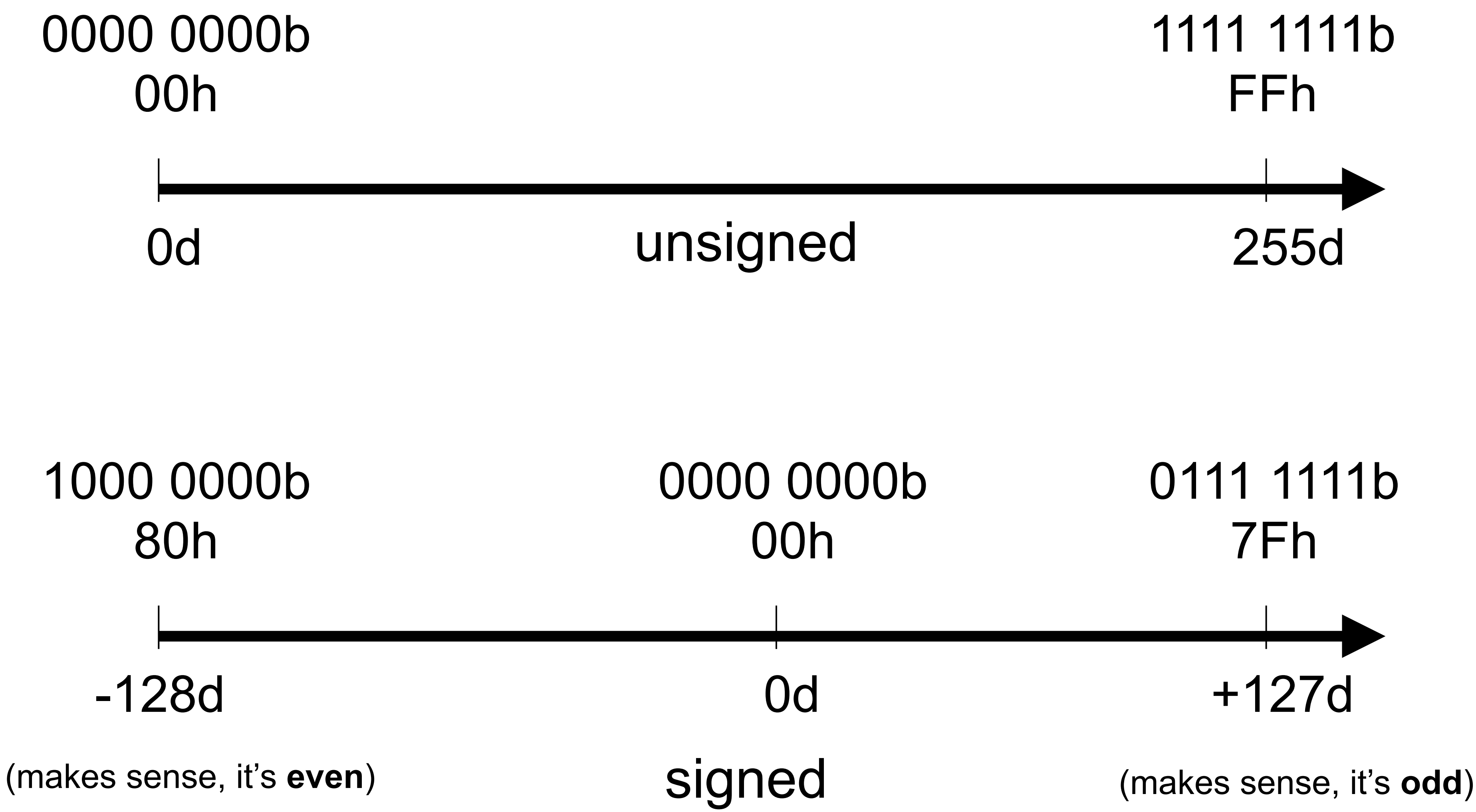
Smallest >= 0 number: 000
Largest >= 0 number: 011

Smallest < 0 number: 100
Largest < 0 number: 111

Ranges of Numbers

- For 1-byte values
  - Unsigned
    - Smallest value: 00_{16} or $0000\ 0000_2$ (0_{10})
    - Largest value: FF_{16} or $1111\ 1111_2$ (255_{10})
  - Signed
    - Smallest value: 80_{16} or $1000\ 0000_2$ (-128_{10})
    - Largest value: $7F_{16}$ or $0111\ 1111_2$ ($+127_{10}$)
- For 2-byte values
  - Unsigned
    - Smallest value: 0000_{16} (0_{10})
    - Largest value: $FFFF_{16}$ ($65{,}535_{10}$)
  - Signed
    - Smallest value: 8000_{16} ($-32{,}768_{10}$)
    - Largest value: $7FFF_{16}$ ($+32{,}767_{10}$)
- etc.

1-byte Ranges



The magic of 2's complement (for addition)

- Say I have two 1-byte values, A3 and 17, and I add them together:

  $A3_{16} + 17_{16} = BA_{16}$ ("blind" hex addition)

- If my interpretation of the numbers is unsigned:
  - $A3_{16} = 163_{10}$
  - $17_{16} = 23_{10}$
  - $BA_{16} = 186_{10}$
  - and indeed, $163_{10} + 23_{10} = 186_{10}$
- If my interpretation of the numbers is signed:
  - $A3_{16} = -93_{10}$
  - $17_{16} = 23_{10}$
  - $BA_{16} = -70_{10}$
  - and indeed, $-93_{10} + 23_{10} = -70_{10}$
- So, as long as I stick to my interpretation, the binary addition does the right thing assuming 2's complement representation!!!
  - Same thing for the subtraction

Dropping the Carry?

- Remember earlier when I said that dropping the carry can be numerically correct?
  - That should have felt wrong, because we are dropping information
- We’ll come back to this but just consider 1-byte signed hex addition: FF + FF
- In math, we’d get: 1FE
- In computer arithmetic we get: FE (carry is dropped)
- So in computer arithmetic: FF + FF = FE
- If those numbers are **signed**: FF is -1d, and FE is -2d
  - And yes, -1 + -1 = -2 :)
- So dropping the carry is ok in this case
- Stay tuned for more on this later….

No Data Types in Assembly???

- The computer simply stores data as bits based on what a program does
- **But if you say "In memory somewhere we have the one-byte value FF" we have no clue whether that means 255 or -1 (or if it's even used as an integer in the program!)**
- When using a high-level language we can indicate the interpretation of an integer using data types: "I declare x as an `int` and y as an `unsigned char`"

- **In assembly there is no data type (because there are no data types in the CPU!)**
- **But** we have many instructions that operate on different types of data, and we'll just have to be careful to use the right instruction
  - We just saw that for adding integers, a single instruction can be used though, which is amazing!
- The point of using assembly as a teaching vehicle is to demystify everything, including data types :)

Signed does not mean negative!

- It means “a number that is encoded in binary using 2’s complement so that it can take either positive or negative values”
- The encoding of a positive value is the “normal” one (just the binary representation but only if it starts with a 0 bit - otherwise it’s out of range)
- The encoding of a strictly negative value is the “flip-and-add-one” transformation of the strictly positive counterpart value (and the representation will thus NECESSARILY starts with a 1 bit)

Back to the PL examples

```
// C/C++
int x = -12;                 // signed
signed int y = 40;           // signed
unsigned int z = 40;         // unsigned
```

- x is encoded as FF FF FF F4 in hex
- y is encoded as 00 00 00 28 in hex
- z is encoded as 00 00 00 28 in hex

```
// Rust
let x: i16 = -12; // signed
let y: u32 = 2_031; // unsigned
```

- x is encoded as FF F4 in hex
- y is encoded as 00 00 04 07 in hex

Back to the PL examples

```
// Java
Integer.divideUnsigned(-100, -12) // divide as if numbers were unsigned
```

- -100 is a signed number encoded as FF FF FF 9B in hex
- -12 is a signed numbers encoded as FF FF FF F4 in hex
- The above will:
  - Interpret FF FF FF 9B as an unsigned number (which is $4{,}294{,}967{,}196_{10}$)
  - Interpret FF FF FF F4 as an unsigned number (which is $4{,}294{,}967{,}284_{10}$)
  - Perform the division of $4{,}294{,}967{,}196_{10}$ by $4{,}294{,}967{,}284_{10}$
  - The quotient will be zero!

Back to the PL examples

```
// Python
ctypes.c_uint32(-10).value          // interpret -10 as unsigned (32-bit)
```

- -10 is a signed number encoded as FF FF FF F6 in hex
- The above will:
  - Interpret FF FF FF F6 as an unsigned number (which is $4{,}294{,}967{,}286_{10}$)
  - Return the (positive) integer 4,294,967,286

Back to the PL examples

```
// JavaScript
x = -10 >>> 0                              // interpret -10 as unsigned
```

- -10 is a signed number encoded as FF FF FF F6 in hex
- The above will:
  - Interpret FF FF FF F6 as an unsigned number (which is $4,294,967,286_{10}$)
  - Perform a logical right shift by 0 bits, which has the side effect of setting x to $4,294,967,286_{10}$
  - This is very strange (but commonly done in JavaScript to use unsigned numbers)
  - We'll understand it fully when we cover bitwise operations and shifts

Why?

- Why bother with unsigned data types at all?
- Pros:
  - Doubling the range of possible positive values can be critical in some cases
  - Modeling values that cannot be negative is useful
  - Bitwise operations (see later in the semester)
- Cons:
  - Mixing them with signed data types causes all kinds of bugs / design decisions (which is why you need to know about it)
- In the end, even languages that don’t have unsigned integers as a base type provides ways to deal with unsigned values

Conclusion

- We’ll come back to numbers and arithmetic when we use arithmetic assembly instructions
- But for now you must make sure you have solid mastery of the material in this module

- We’ll do some of the posted practice problems in-class

- We will then have **a quiz on this module**