CSCI 210: Computer Architecture
Lecture 7: Negative Numbers, Overflow

Stephen Checkoway
Oberlin College
Slides from Cynthia Taylor

Announcements

- Problem Set 2 due Friday
- Lab 1 available now

How We Store Numbers

- Binary numbers in memory are stored using a finite, fixed number of bits (typically 8, 16, 32, or 64)
  - 8 bits = byte (usually and always in this class)
- Pad extra digits with leading 0s
- A byte representing 4_{10} = 00000100

Java

- A `byte` is 8 bits
- A `char` is 16 bits
- A `short` is 16 bits
- An `int` is 32 bits
- A `long` is 64 bits

Rust

- bools are 1 byte, chars are 4 bytes
- Specify size in type for ints
  - i8, i16, i32, etc
- isize or usize will be the size of an address on the architecture it's compiled for
  - 32 bits on 32 bit systems, 64 bits on 64 bit systems

In C, an int is

A. 8 bits

B. 16 bits

C. 32 bits

D. It depends

E. None of the above

C specifies a *minimum size* for types

- chars are 1 byte and must be at least 8 bits
- shorts and ints must be at least 16 bits
- longs are at least 32 bits
- long longs are at least 64 bits
- sizeof(type) tells us how many bytes type is
- 1 = sizeof(char) ≤ sizeof(short) ≤ sizeof(int) ≤ sizeof(long) ≤ sizeof(long long)

So how do I know?

- Use sizeof(int) to check
- Or use C99 types like int16_t or int32_t

Questions So Far?

How do we indicate a negative number?

- Sign and magnitude (History)
- Ones’ Compliment (History)
- Two’s Compliment (Modern Systems)

Sign and Magnitude

- Have a separate bit for sign
- Set it to 0 for positive, and 1 for negative
- Can represent from -127 to 127 in 8 bits
- With n bits, can represent $-(2^{n-1}-1)$ to $2^{n-1}-1$

Addition and subtraction are a hassle

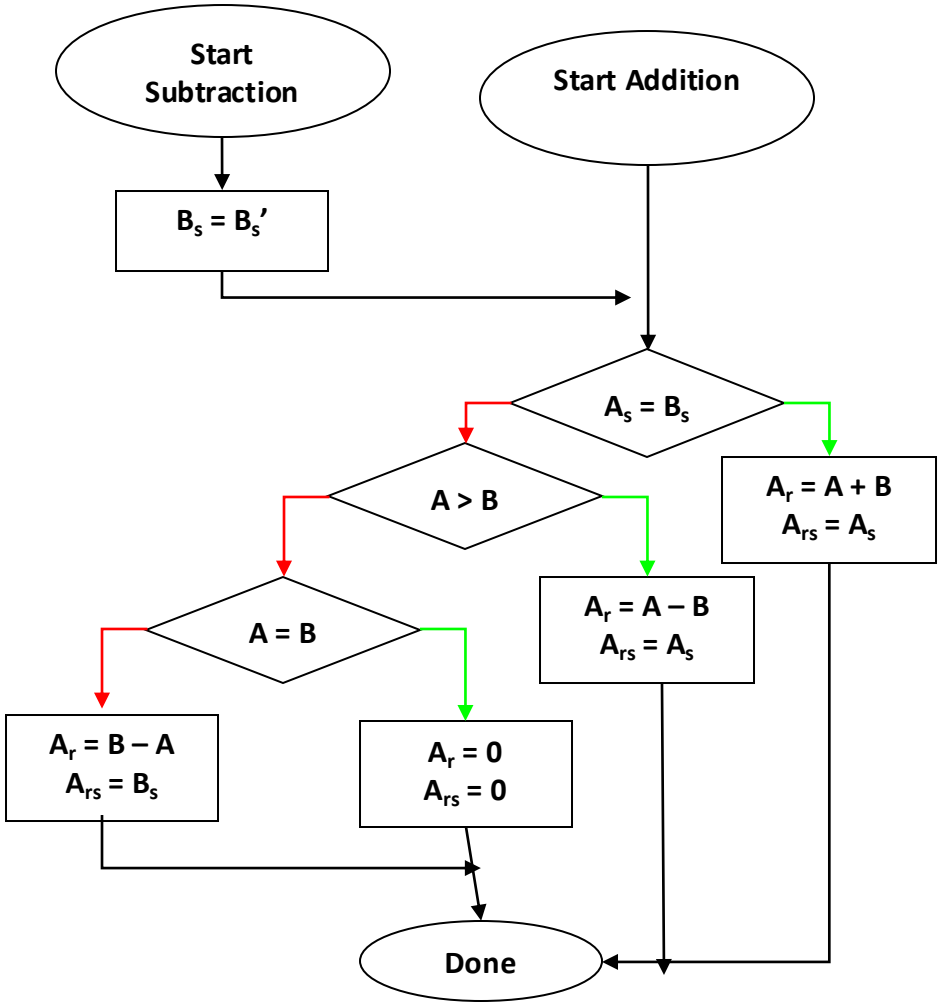

Diagram from Marek Andrzej Perkowski

A byte representing -6_{10} in Sign and Magnitude (with leftmost sign bit) is

A. 0000 0111

B. 1000 0110

C. 1000 0111

D. 1111 1110

E. None of the above

Which is NOT a drawback of Sign and Magnitude?

A. There are two zeros

B. Unclear where to put the sign bit

C. Complicated arithmetic

D. Difficult to convert numbers to negative representation

E. None of the above

Ones' Complement

- To make a number negative, just flip all its bits!

- Need to know how many bits: -5 in
  - 4 bits: -0101 = 1010
  - 8 bits: -00000101 = 11111010

A byte representing -6_{10} in Ones' Complement is

A. 00000110

B. 10000110

C. 11111001

D. 11110110

E. None of the above

Ones' complement

- Two zeros: 00000000 and 11111111 (in 8 bits)

- Addition:
  - Perform normal n-bit addition
  - Add the carryout bit back to the result

Two's Complement

- To compute –x, flip all the bits of x and add 1
- For n bits, the unsigned version of $-x = 2^n - x$
- Can represent –128 to 127 in 8 bits
  - In n bits, can represent -2^{n-1} to $2^{n-1} - 1$
- Only one zero (00000000 in 8 bits)
- Used in modern computers

Short aside

- ones' complement involves taking each bit and taking the complement with respect to 1; there are many bits so many complements with respect to 1 hence "ones' complement"
- two's complement involves taking a complement with respect to a single power of 2, not bit-by-bit, hence "two's complement"
- Yes. It *is* confusing. No. No one remembers this.

-6 in Two’s Complement

A. 11110110

B. 11111001

C. 11111010

D. 11111110

E. None of the above

Two's Complement: $11111101_2 = ?_{10}$

A. -2

B. -3

C. -4

D. -5

E. None of the above

If we multiply 11110001_2 by -1, we get $\_\_\_\_\_{}_2$

A. 00001110

B. 00001111

C. 00011110

D. 01110001

E. None of the above

Addition and Subtraction

- Positive and negative numbers are handled in the same way.
- The carry out from the most significant bit is ignored.
- To perform the subtraction A − B, compute A + (two's complement of B)

For n bits, the sum of a number and its negation will be

A. $0_{n-1}\ldots0_0$

B. $1_{n-1}0_{n-2}\ldots0_0$

C. $1_{n-1}\ldots1_0$

D. It will vary

E. None of the above

$$11110110_2 + 00001100_2 = ?_2$$

A. 00000010

B. 00001100

C. 11110010

D. 11111110

E. None of the above

$$1001_2 + 1011_2 = ?_2$$

A. 0010

B. 0100

C. 1000

D. 1111

E. None of the above

Overflow

- Overflow occurs when an addition or subtraction results in a value which cannot be represented using the number of bits available.

- In that case, the algorithms we have been using produce incorrect results.

What will this java code print?

A. -2147483648

B. 0

C. 2147483647

D. 2147483648

```
public static void main(String args[]) {
        int x = 2147483647;
        x = x + 1;
        System.out.println(x);
}
```

Handling Overflow

- Hardware can detect when overflow occurs

- Software may or may not check for overflow
  - Java guarantees two's complement behavior!
  - In C, overflow is "undefined behavior" meaning, it can do anything
  - In Rust, overflow is checked in debug builds but not optimized builds!

How To Detect Overflow

- On an addition, an overflow occurs if and only if the carry into the sign bit differs from the carry out from the sign bit.

- Overflow occurs if adding two negative numbers produces a positive result or if adding two positive numbers produces a negative result.

Will $01111111_2 + 00000101_2$ result in overflow when treated as 8-bit signed integers?

A. Yes

B. No

C. It depends

Unsigned Numbers

- Some types of numbers, such as memory addresses, will never be negative
- Some programming languages reflect this with types such as “unsigned int”, which only hold positive numbers
  - uint32_t in C99
  - u32 in Rust
  - Java only has signed types (except for char which is unsigned 16-bit)
- In an unsigned byte, values will range from 0 to 255

In MIPS

- add, sub, addi instructions cause exceptions on (signed) overflow
- addu, subu, addiu instructions do not

- Rationale: In C, unsigned types never cause overflow, they're defined to wrap (produce a value modulo 2^n)
- In practice: Since overflow is undefined behavior, it is assumed to never happen so compilers always use addu/subu/addiu

Reading

- Next lecture: How Instructions Are Represented
  - Section 2.5

- Problem Set 2 due Friday

- Lab 1 due next Monday