Number representation: binary arithmetic; overflow

COSC 208, Introduction to Computer Systems, 2022-02-10

Announcements

- Project 1 Part A due Thursday at 11pm

Outline

- Warm-up
- Binary arithmetic
- Overflow

Warm-up

- *Express these decimal numbers using 8-bit two's complement:*
  - Q1: -49 = `0b11001111`
  - Q2: -11 = `0b11110101`
- *What is the easy way to negate a number?*
  - Flip all bits and add 1
  - Example:
    - 11 = `0b00001011`
    - Flip bits: `0b11110100`
    - Add 1: `0b11110101`

Binary arithmetic

Addition

Same as decimal, except you carry a one instead of a ten Example: 5 + 5

```
  0b0101
+ 0b0101
--------
```

```
  0b0101
+ 0b0101
--------
```

```
      1
  0b0101
+ 0b0101
--------
       0
```

```
     01
  0b0101
+ 0b0101
--------
      10
```

```
   101
  0b0101
+ 0b0101
--------
     010
```

```
   101
  0b0101
+ 0b0101
--------
  0b1010
```

Check our work:

```
1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0 = 8 + 2 = 10 = 5+5
```

Another example: 5 + -5

```
  0b0101
+ 0b1011
--------
```

```
     1
  0b0101
+ 0b1011
--------
       0
```

```
    11
  0b0101
+ 0b1011
--------
      00
```

```
   111
  0b0101
+ 0b1011
--------
     000
```

```
    111
  0b0101
+ 0b1011
--------
    0000
(Carry-out => 1)
```

Subtraction

Simply add the negation

Practice using 8-bit signed integers

- Q3: 10 + 5 = `0b00001010` + `0b00000101` = `0b00001111`
- Q4:7 + 15 = `0b00000111` + `0b00001111` = `0b00010110`
- Q5: -10 + 5 = (`0b11110101` + `0b1`) + `0b00000101` = `0b11110110` + `0b00000101` = `0b11111011`
- Q6: 10 - 5 = `0b00001010` + (`0b11111010` + `0b1`) = `0b00001010` + `0b11111011` = `0b00000101`
- Q7: 64 + 64 = `0b01000000` + `0b1000000` = `0b10000000`

Overflow

- *Convert the 8-bit signed integer `0b10000000` to decimal:* -128
- 64 + 64 = -128!? What!?
- Computation overflowed — i.e., result is too large to be represented
    - Computation wraps around to negative numbers
    - Can only occur when you add two positive numbers
- Computation can also underflow — i.e., result is too small to be represented
    - Computation wraps around to positive numbers
    - E.g., -64 + -65 = `0b11000000` + `0b10111111` = `0b01111111` = 127
    - Can only occur when you add two negative numbers
- Overflow and underflow are impossible when adding a positive number and a negative number
    - Assume you add the largest magnitude positive number and the smallest magnitude negative number (-1); the result will be slightly less magnitude than the positive operand, and thus cannot overflow
    - Assume you add the smallest magnitude positive number (1) and the largest magnitude negative number; the result will be slightly less magnitude than the negative operand, and thus cannot underflow
- *What happens if you overflow with unsigned integers?*
  — you wrap around to zero, and get a smaller positive integer
- *What happens if you underflow with unsigned integers?* — you wrap around to the maximum value, and get a larger positive integer

Practice with overflow

For each of the following computations, determine whether the computation overflows, underflows, or neither. Assume we are using 8-bit signed integers.

- Q8: `0b10000000` + `0b01111111` — neither
- Q9: `0b10000001` + `0b01111111` — neither
- Q10: `0b10000000` + `0b10000001` — underflow
- Q11: `0b11000000` + `0b11000000` — neither
- Q12: `0b01111111` + `0b00000001` — overflow

Extra practice

- QA: *Convert `512` to unsigned binary.* — `0b1000000000`
- QB: *Convert `-42` to 8-bit signed binary.* — `0b11010110`
- QC: *Convert `0xFAB` to unsigned binary.* – `0b111110101011`
- QD: *Write a function called `valid_hex` that takes a string and returns 1 if it is a valid hexadecimal number; otherwise return 0. A valid hexadecimal number must start with `0x` and only contain the digits `0`-`9` and letters `A`-`F` (in upper or lower case).*

```
int valid_hex(char str[]) {
    if (str[0] != '0' || str[1] != 'x') {
        return 0;
    }
    for (int i = 2; i < strlen(str); i++) {
        if (!((str[i] >= '0 && str[i] <= '9')
            || (str[i] >= 'A' && str[i] <= 'F')
            || (str[i] >= 'a' && str[i] <= 'f'))) {
            return 0;
        }
    }
    return 1;
}
```

- QE: *Write a function called `bits_required` that takes an `unsigned long` decimal (i.e., base 10) number and returns the minimum number of bits required to represent the number.*

```
int bits_required(unsigned long number) {
    int bits = 0;
    while (number > 0) {
        bits++;
        number = number / 2;
    }
    return bits;
}
```