CSC2/452 Computer Organization
Bitsets, Bitfields, Integer Arithmetic

Sreepathi Pai

URCS

September 9, 2019
Outline

Recap

Bitsets

Bitfields

Logical Operations

Integer Arithmetic
Stuff from previous lectures that we’ll need today

- **Bits**: 0 and 1
  - **Unary functions**: NOT
  - **Binary functions**: AND, OR, XOR, NAND, NOR, ...
  - Defined for inputs of 1 bit
  - Extensible to bit vectors and bit strings

- **Machine data types**
  - **Byte (8 bits)** smallest unit
  - Usually go up to 64 bits

- **Integers**
  - Sign-Magnitude, One’s complement, Two’s Complement, ...

- **C Mapping of Types**
  - machine-dependent sizes: char, int, ...
  - machine-independent sizes: uint8_t, uint16_t, ...
Outline

Recap

Bitsets

Bitfields

Logical Operations

Integer Arithmetic
Mathematical Sets

- $P = \{2, 3, 5, 7, 11, \ldots\}$
- $O = \{1, 3, 5, 7, 9, 11, 13, \ldots\}$
- Operations on mathematical sets:
  - Membership: $9 \in P$?
  - Union: $P \cup O$
  - Intersection: $P \cap O$
  - Complement: $\tilde{O} = O' = \{2, 4, 6, 8, \ldots\}$ (assuming $U = \mathbb{N}^+$)
  - Difference: $P \setminus \{2\}$
  - Symmetric Difference: $P \oplus O = \{1, 2, 9, \ldots\}$ (elements in either set but not both)
Suppose you want to store two pieces of information ... 
- On/Off
- Absent/Present
- Enable/Disable
- etc.

about \( N \) different items simultaneously

Example: Days of the week that I am available to meet
- information: available/not available
- 7 items: Sun, Mon, Tue, Wed, Thu, Fri, Sat

How should we encode this information?
- \( \text{not using} \ \lceil \log_2(7) \rceil \text{ bits!} \)
- That’s for storing 1 of \( N \) different values
Constructing Bitsets

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b₆</td>
<td>b₅</td>
<td>b₄</td>
<td>b₃</td>
<td>b₂</td>
<td>b₁</td>
<td>b₀</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>SAT</td>
<td>FRI</td>
<td>THU</td>
<td>WED</td>
<td>TUE</td>
<td>MON</td>
<td>SUN</td>
</tr>
</tbody>
</table>

- Use 1 bit per day, total 7 bits
- If bit $x$ is set to 1 then I’m available on day mapped to bit $x$
  - Interpretation matters: 1 could also be used to mean unavailable (and then 0 would be available)
  - We will use 1 for available, 0 for unavailable
enum DOW_BITS {
    SUN = 1, /* bit 0 */
    MON = 2,
    TUE = 4,
    WED = 8,
    THU = 16,
    FRI = 32,
    SAT = 64 /* bit 6 */
};

Another common option for defining constants in C:

▶ #define SUN 1 (avoid for new code)
C bitwise operations

- Bitwise $NOT(a)$ is $\sim a$
- Bitwise $AND(a, b)$ is $a \& b$
- Bitwise $OR(a, b)$ is $a \mid b$
- Bitwise $XOR(a, b)$ is $a \sim b$
Marking availability

Marking availability on Monday and Tuesday

<table>
<thead>
<tr>
<th></th>
<th>$b_6$</th>
<th>$b_5$</th>
<th>$b_4$</th>
<th>$b_3$</th>
<th>$b_2$</th>
<th>$b_1$</th>
<th>$b_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>availability (before)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mark</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>availability (after)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Marking availability (in code)

▶ Marking availability on Monday and Tuesday

<table>
<thead>
<tr>
<th>availability (before)</th>
<th>b6</th>
<th>b5</th>
<th>b4</th>
<th>b3</th>
<th>b2</th>
<th>b1</th>
<th>b0</th>
</tr>
</thead>
<tbody>
<tr>
<td>mark</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>availability (after)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

▶ In C code:

```c
uint8_t availability = 0;
availability = MON | TUE;
```
Adding availability

▶ Add availability on Monday and Tuesday

<table>
<thead>
<tr>
<th>availability (before)</th>
<th>$b_6$</th>
<th>$b_5$</th>
<th>$b_4$</th>
<th>$b_3$</th>
<th>$b_2$</th>
<th>$b_1$</th>
<th>$b_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>mark</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

▶ What needs to change in C code?

```c
availability = MON | TUE;
```
Adding availability (answer)

- Add availability on Monday and Tuesday

<table>
<thead>
<tr>
<th>availability (before)</th>
<th>$b_6$</th>
<th>$b_5$</th>
<th>$b_4$</th>
<th>$b_3$</th>
<th>$b_2$</th>
<th>$b_1$</th>
<th>$b_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>mark</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>availability (after)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- C code

```c
availability = availability | MON | TUE;
```

- Which set operation does OR resemble?
OR

- Can be used to set bits to 1
- Does not change a bit if it is already 1
- Resembles the set union operation $\cup$
Handling “except” adds

▶ All days except Monday or Tuesday

<table>
<thead>
<tr>
<th></th>
<th>(b_6)</th>
<th>(b_5)</th>
<th>(b_4)</th>
<th>(b_3)</th>
<th>(b_2)</th>
<th>(b_1)</th>
<th>(b_0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>availability (before)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>except</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>availability (after)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

▶ C code

```c
uint8_t except;

except = MON | TUE;
availability = /* complete this */
```

▶ post-lecture answer: \(\sim\)except
Checking availability

▶ Am I available on Thursday?

<table>
<thead>
<tr>
<th></th>
<th>b6</th>
<th>b5</th>
<th>b4</th>
<th>b3</th>
<th>b2</th>
<th>b1</th>
<th>b0</th>
</tr>
</thead>
<tbody>
<tr>
<td>availability</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>check</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>result</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

▶ C code (what is OP?)

```c
uint8_t result;

result = availability OP THU;

/* value of result will be 0 or THU, depending on whether bit THU is zero or one respectively */
```

▶ answer: &
Comparing availability

Which days are both Prof. #1 and Prof. #2 available?

<table>
<thead>
<tr>
<th></th>
<th>b6</th>
<th>b5</th>
<th>b4</th>
<th>b3</th>
<th>b2</th>
<th>b1</th>
<th>b0</th>
</tr>
</thead>
<tbody>
<tr>
<td>availability (Prof. #1)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>availability (Prof. #2)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>result</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

C code (what is OP?)

```c
uint8_t avail_prof_1 = SAT | THU | TUE;
uint8_t avail_prof_2 = SAT | FRI | TUE;
uint8_t result_common;

result_common = avail_prof_1 OP avail_prof_2;
```

OP: &
Printing out the days both are available

```c
char *days[] = {"SUN", "MON", "TUE", "WED", "THU", "FRI", "SAT"};

int bitvalue = 1;
int bit;

for(bit = 0; bit < 7; bit++) {
    if((result_common & bitvalue) == bitvalue) {
        printf("both available on %s\n", days[bit]);
    }
    bitvalue = bitvalue * 2;
}
```
Mark unavailable on Tuesday and Thursday

- Change to unavailable for Tuesday and Thursday

<table>
<thead>
<tr>
<th></th>
<th>b₆</th>
<th>b₅</th>
<th>b₄</th>
<th>b₃</th>
<th>b₂</th>
<th>b₁</th>
<th>b₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>availability (before)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>remove</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mask</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>availability (after)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- C code (what are OP1 and OP2)?

```c
uint8_t remove = TUE | THU;

availability = availability OP1 OP2(remove);
```

- OP1: &
- OP2: ~
AND

- Can be used to set bits to 0 by masking them out
- Can test if bits are set to 1
- Resembles the membership operation $\in$ with single bits
- But more generally, the intersection operation $\cap$ on bitsets
## Bitset Difference

### Which days is Prof. #1 available, but Prof. #2 is not?

<table>
<thead>
<tr>
<th></th>
<th>$b_6$</th>
<th>$b_5$</th>
<th>$b_4$</th>
<th>$b_3$</th>
<th>$b_2$</th>
<th>$b_1$</th>
<th>$b_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>availability (Prof. #1)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>availability (Prof. #2)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>difference</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### C code (what are OP1 and OP2?)

```c
uint8_t result_diff;

result_diff = avail_prof_1 OP1 OP2(avail_prof_2);
```
### Bitset Difference Solution

Which days is Prof. #1 available, but Prof. #2 is not?

<table>
<thead>
<tr>
<th></th>
<th>$b_6$</th>
<th>$b_5$</th>
<th>$b_4$</th>
<th>$b_3$</th>
<th>$b_2$</th>
<th>$b_1$</th>
<th>$b_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>availability (Prof. #1)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>availability (Prof. #2)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mask</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>difference</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

▶ C code (OP1=& and OP2=˜)

```c
uint8_t result_diff;

result_diff = avail_prof_1 OP1 OP2(avail_prof_2);
```
Symmetric Difference

On which days is only one Prof. available?

<table>
<thead>
<tr>
<th>availability (Prof. #1)</th>
<th>b_6</th>
<th>b_5</th>
<th>b_4</th>
<th>b_3</th>
<th>b_2</th>
<th>b_1</th>
<th>b_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>availability (Prof. #2)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>symmetric difference</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

C code (what is OP?)

```c
result_diff = avail_prof_1 ^ avail_prof_2;
```

OP: ^
XOR

- Note that $A \oplus B \oplus A = B$
- XOR is a reversible operation
- XOR is symmetric difference on bitsets, hence also represented by $\oplus$
Summary of bitsets

- Bitsets equivalent to sets
  - But can only store 0/1 about $N$ items
  - Requires $N$ bits
- Very compact and easily to manipulate if $N < 64$ (i.e. machine word size)
  - But it is easy to build bitsets of any size (just use arrays of uint8_t or a larger type)
- Operation mapping
  - Union: OR (|)
  - Intersection: AND (&)
  - Complement: NOT (~)
  - Difference: AND NOT (& ~)
  - Symmetric Difference: XOR (^)
Outline

Recap

Bitsets

Bitfields

Logical Operations

Integer Arithmetic
Bitpacking

- Sometimes space is at a premium
- Want to use as few bits as possible
- Bitfields:
  - Partition a machine word into distinct fields
Example

- Suppose we want to store day and day of the week using as little space as possible
  
  ```c
  uint8_t day = 9;
  uint8_t dow = MON;
  ```

- Day takes value 1–31
  - Bits required: ?

- Day of week takes 0 (SUN)– 6(SAT)
  - Note here we want to store 1 of 7 values
  - No need for a bitset
  - Bits required: ?

- Total storage used by two `uint8_t` variables: 16 bits

- Bits wasted: ?
enum days_of_week {
    SUN = 0,
    MON = 1,
    TUE = 2,
    WED = 3,
    THU = 4,
    FRI = 5,
    SAT = 6
};
Constructing a bitfield

```c
uint8_t daydow;
daydow = (9 << 3) | MON;
```

- Here, 9 is the day
- It is *left-shifted* by 3 bits using the left-shift (`<<`) operator
  - Empty positions at right are filled with zeroes
  - Bits at left are *discarded*
  - Like multiplying by $10^3$ in the metric system, except here we’re multiplying by $2^3$
    - 0x9 (binary 1001) becomes 0x48 (binary 0100 1000)
- Then we OR the day of the week into the freshly created lower 3 zero bits
Getting the Day of the Week

<table>
<thead>
<tr>
<th>Day of Month</th>
<th>b7</th>
<th>b6</th>
<th>b5</th>
<th>b4</th>
<th>b3</th>
<th>b2</th>
<th>b1</th>
<th>b0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon 9</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Day of Week

<table>
<thead>
<tr>
<th>Day of Month</th>
<th>b7</th>
<th>b6</th>
<th>b5</th>
<th>b4</th>
<th>b3</th>
<th>b2</th>
<th>b1</th>
<th>b0</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- We want to force the day field to zero.
- What are OP and MASK?

\[
dow = \text{daydow \ OP \ MASK};
\]
Getting the Day of the Week (Solution)

<table>
<thead>
<tr>
<th></th>
<th>$b_7$</th>
<th>$b_6$</th>
<th>$b_5$</th>
<th>$b_4$</th>
<th>$b_3$</th>
<th>$b_2$</th>
<th>$b_1$</th>
<th>$b_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon 9 Day of Month</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Day of Week</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mask</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>result</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

We want to force the day field to zero.

dow = daydow & 0x7;
Getting Day of Month

<table>
<thead>
<tr>
<th></th>
<th>b7</th>
<th>b6</th>
<th>b5</th>
<th>b4</th>
<th>b3</th>
<th>b2</th>
<th>b1</th>
<th>b0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon 9</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Day of Month</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mask</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>result</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

▶ C code, all bits except lower 3

```c
    day = daydow & (0x1f << 3);
```

▶ Is the result what we want?
Undoing the left shift

day = (daydow & (0x1f << 3)) >> 3;

▶ We got \(0x48\) because we had shifted it left
▶ We can undo it by doing a right shift by 3 bits using the right-shift >>) operator
  ▶ Bits at right are *discarded*
  ▶ Like dividing by \(2^3\), and throwing away the remainder/fractional part
  ▶ \(0x48\) (binary 0100 1000) becomes \(0x9\) (binary 0000 1001)
Even shorter ...

day = (daydow >> 3) & 0x1f;

▶ Since we’re using uint8_t, the masking is superfluous
▶ For unsigned integers, right shifting will fill in bits at left with 0
▶ Since all bits to the left of the Day of Month field are zero, we can eliminate the mask and the AND
  ▶ But recommend always using a mask, as good programming practice
Real-life bitfields and bitsets: Unix file permissions

- Basic File permissions (can be simultaneously enabled)
  - READ
  - WRITE
  - EXECUTE

- Permissions for
  - User/Owner
  - User’s Group
  - Others

<table>
<thead>
<tr>
<th>rwxr-x---</th>
<th>(b_8)</th>
<th>(b_7)</th>
<th>(b_6)</th>
<th>(b_5)</th>
<th>(b_4)</th>
<th>(b_3)</th>
<th>(b_2)</th>
<th>(b_1)</th>
<th>(b_0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>User/Owner</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Group</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Others</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Outline

Recap

Bitsets

Bitfields

Logical Operations

Integer Arithmetic
Bitwise Operations

- Bitwise operations operate on integers and produce integers
- Logical operations operate on booleans/integers and produce booleans
- Logical operators:
  - AND: `&&`
  - OR: `||`
  - NOT: `!`
- Boolean values are TRUE and FALSE
  - In C, 0 is FALSE
  - Everything else is TRUE
Comparing Bitwise and Boolean: Results

- **AND**
  - Bitwise: $3 \& 7 = 3$
  - Boolean: $3 \&\& 7 = 1$

- **OR**
  - Bitwise: $3 \mid 7 = 7$
  - Boolean: $3 \mid\mid 7 = 1$

- **NOT**
  - Bitwise: $\sim 0 = 0xff...$
  - Boolean: $!0 = 1$
  - Boolean: $!5 = 0$
Boolean Operators: Short-circuit Behaviour

- **AND**
  - Boolean: $0 \&\& \left( \frac{5}{0} \right) = 0$

- **OR**
  - Boolean: $3 \mid\mid \left( \frac{5}{0} \right) = 1$

None of the divisions by zero in the expressions above will be executed

- $\&\&$ stops evaluating as soon as a subexpression evaluates to FALSE
- $\mid\mid$ stops evaluating as soon as a subexpression evaluates to TRUE
For int32_t type:
  - 1 bit sign
  - 31 bits value
  - Essentially a bitfield!
Adding two integers of different widths

```c
int8_t x = -5;
int16_t y = 10;

y = y + x;
```
Shifting a signed integer to the right

```c
int8_t x = -72;
x = x >> 3;
```
Unsigned integer overflow and underflow

/* overflow */
uint8_t x = 255;
x = x + 1;

/* underflow */
uint8_t x = 0;
x = x - 1;
Signed integer overflow

/* overflow */
int8_t x = INT8_MAX;
x = x + 1;

/* underflow */
int8_t x = INT8_MIN;
x = x - 1;
Summary

- Bitsets
  - and set-like operations using bitwise operators
- Bitfields
  - shifts and masks
- Logical operators
  - differences from bitwise and short-circuit behaviour
- Integer arithmetic
  - Some things to ponder about