Operating Systems and C
Fall 2022
2. Representing Data

Operating Systems and C
Fall 2022
2. Representing Data

why do we bother looking into this?

Motivation: Performance

Data Representation is crucial for data science, e.g.,
How to represent a binary matrix? (bits ⇒ ops bit-level)

```
#include <math.h>
#include <stdint.h>
#include <malloc.h>
#include <stdio.h>

#define N 8
#define M 8
#define n 0
#define m 1

int main(int argc, char *argv[])
{
        /* binary matrix allocation */
        uint64_t *matrix;
        matrix  = (uint64_t *) malloc((size_t) ceil((N*M)/sizeof(uint64_t)));
        *matrix = 0x1001b;
        /* reading the [n][m]-th element of an NxM binary matrix  */
        uint64_t mask = 1U << ((n*N+m)%64U);
        uint64_t result = mask & matrix[(size_t) ceil((n*N+m)/(64*1.0))];
        result = result >> ((n*N+m)%64U);
        printf("matrix[n][m]: %lu\n",result);
        return 0;
}
```


https://www.quora.com/What-is-the-best-way-to-declare-binary-matrices-in-C

Motivation: Security

l2ex1.c+

```c
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

expects type `size_t` (unsigned), given an `int`. what happens? problem?

- Similar to code found in FreeBSD’s implementation of getpeername
- There are legions of smart people trying to find vulnerabilities in programs

Outline

we need a deeper understand of how data is represented.

1. **Two's complement**
2. Integer Arithmetic
3. Bit Manipulation

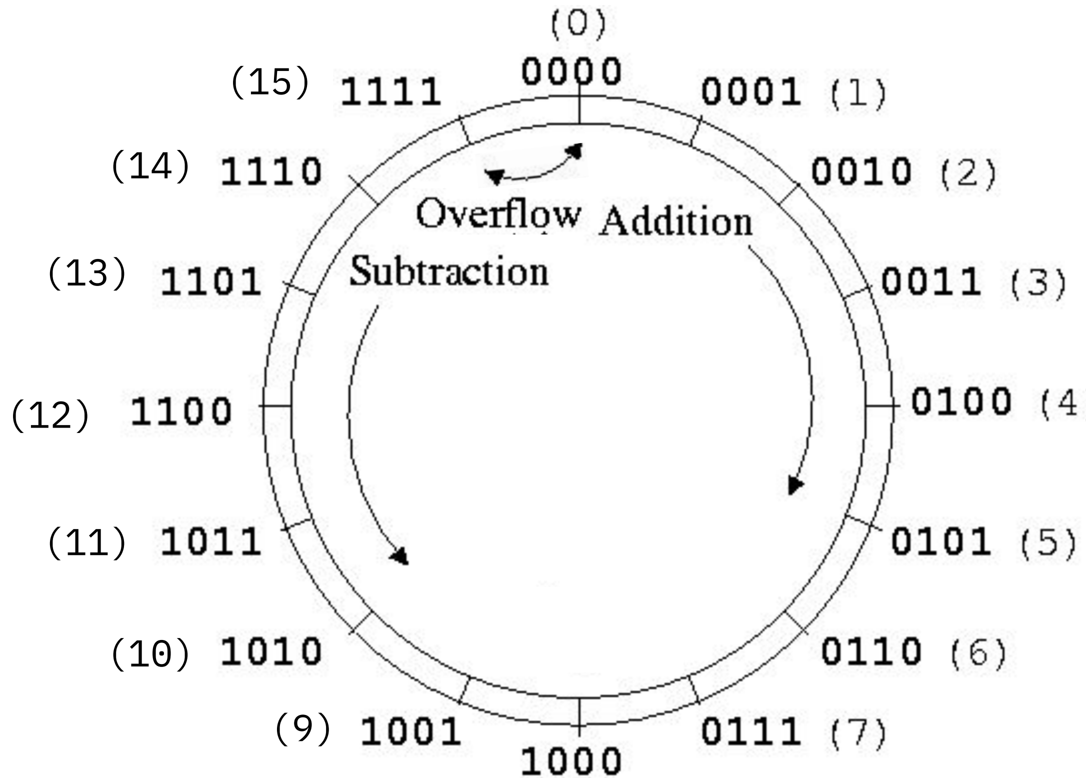
Two issues

1. Finite representation
   - There is a limit to the number of integers (put differently, a max value) that can be represented on a fixed number of bytes
2. Representing positive and negative integers
   - Sign and values must be represented

More than one way to skin a cat.
What's a **sensible** way?

one where *mathematical operations* make **sense**!

Spoiler: Algebra: Ring (Finite Field) - Unsigned



parentheses: how we *interpret* the bits.

Modular / Clock arithmetic.

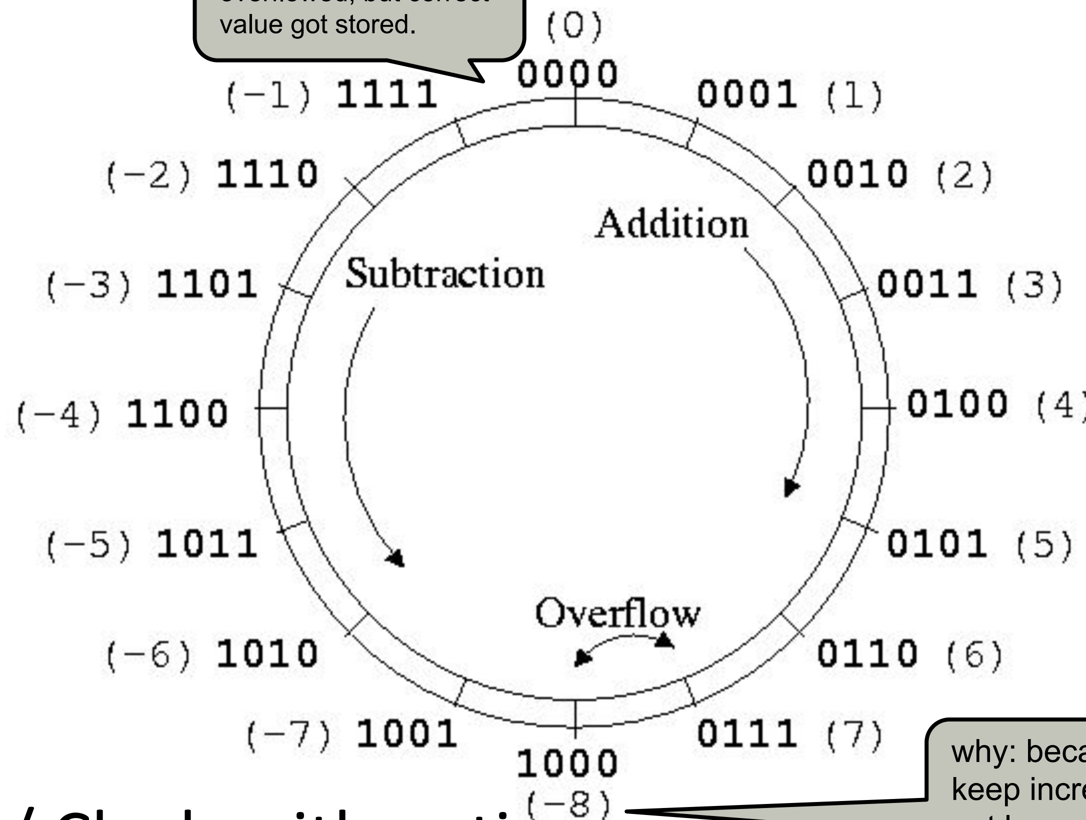
https://stackoverflow.com/questions/7221409/is-unsigned-integer-subtraction-defined-behavior

photo: https://dncsite.wordpress.com/tag/twos-complement/

Spoiler: Algebra: Ring (Finite Field) - Signed

note: binary addition overflowed, but correct value got stored.

parentheses: how we *interpret* the bits.



why: because then, if you keep incrementing, you get larger numbers.

Modular / Clock arithmetic.

note: binary addition of 3 data-bits overflowed, thus overwriting the sign bit

https://stackoverflow.com/questions/72214 ... ubtraction-defined-behavior

https://dncsite.wordpress.com/tag/twos-complement/

Integer Types in C

- Signed vs. unsigned

- Number of bytes
  - 1B: char
  - 2B: short
  - 4B: int
  - 4B (32 bits) or 8B (64 bits): long
  - 8B: long long

Sequences of bits

hex: compact representation of bits.

Shorts in C are coded on 2B. Used for examples in these slides.

2B = 16 bits,

or 4 Hexadecimals (prefixed with Ox)

see book on
dec-bin, bin-dec,
dec-hex, hex-dec,
bin-hex, hex-bin

or:
`onlinetoolz.net`

Operations on sequences of bits:

- **bitwise** operations: and (&), or (|), not (~), xor (^), shift (<<, >>)
- (interpreted as Booleans) 0 is false; anything but 0 is true
  logical operations: and (&&), or (||), not (!)
- (interpreted as integers)
  arithmetic operations: +, -, *, /

Bitwise Operations: Examples

Bitwise on cos
https://github.com/mellowcandle/bitwise

- 0x0000+1 = 0x0001; 0x000E+1 = 0x000F; 0x000F+1 = 0x0010
- 1 << 15 = 0x8000; 1 << 8 = 0x0100; 1 << 3 = 0x0008
- 0xEFFF+1 = 0xF000; 0x24BC+1 = 0x24BD
- ~0x0000 = 0xFFFF; ~0x0001 = 0xFFFE
- X+~X = 0xFFFF ; X&~X = 0

Signed vs. Unsigned Interpretation

A **sequence of bits** is interpreted differently depending on whether the integer type is unsigned or signed.

how much each bit contributes to value

0xA000

-x is *not* just bits of x w/ a 1 in sign

Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

40960

Signed (**two's complement**)

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

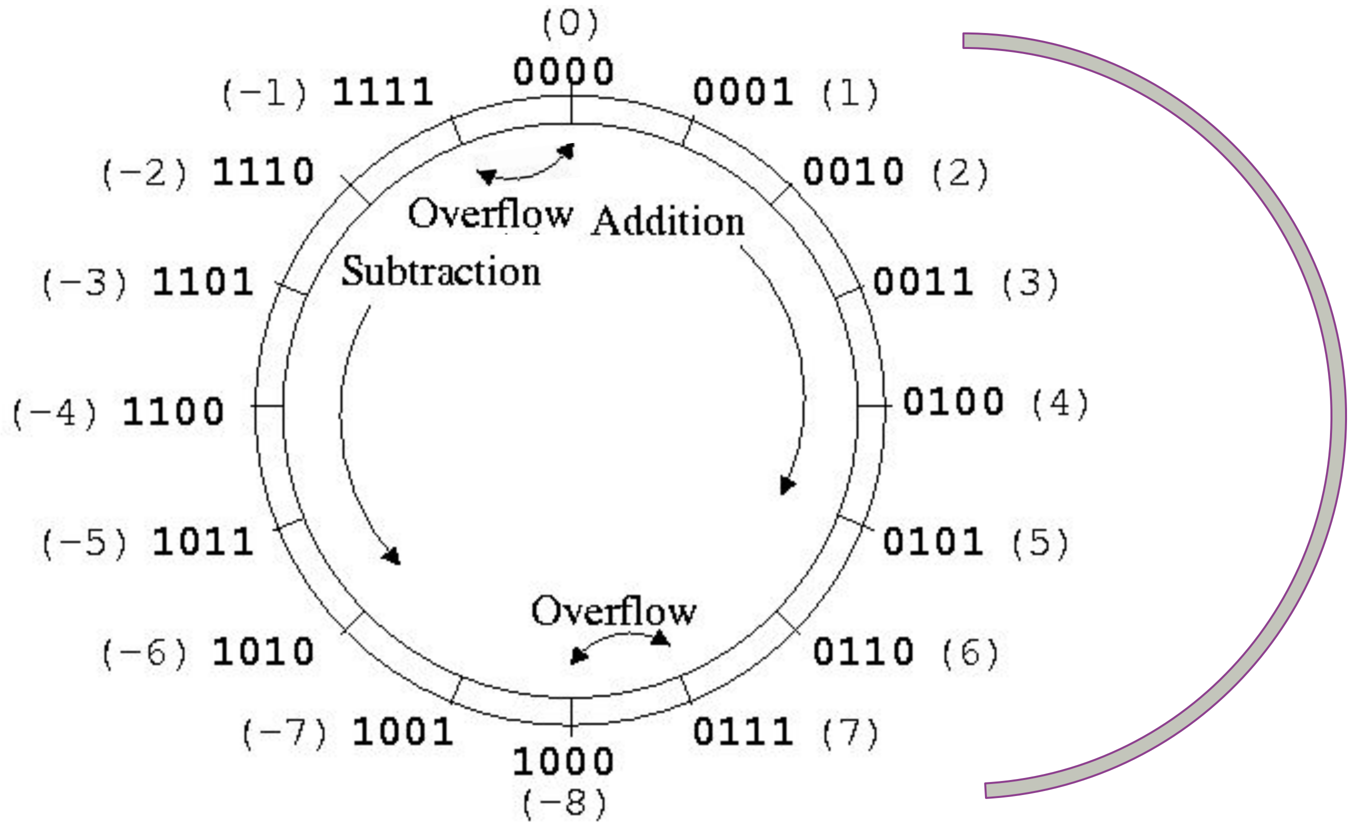
Sign Bit

-24576

same bits; different **interpretations** as numbers.

https://onlinetoolz.net/unsigned-signed

Identical Interpretation?



book has interesting results on
when bits have identical unsigned & signed interpretation.
(outside range, conversion either adds or subtracts 2^w)

Integer Types in C

$ vi /usr/include/limits.h

```
/* Minimum and maximum values a `signed short int' can hold.  */
#  define SHRT_MIN        (-32768)
#  define SHRT_MAX        32767

/* Maximum value an `unsigned short int' can hold.  (Minimum is 0.)  */
#  define USHRT_MAX       65535
```

0x0000 $B2U\_\min = 0$

0xFFFF $B2U_{max} = \sum_{i=0}^{15} 2^i$
$= 2^{16}-1$

0x8000 $B2Tmin = -2^{15}$

0x7FFF $B2T_{max} = \sum_{i=0}^{14} 2^i$
$= 2^{15}-1$

1 bit reserved for sign

there's 1 more neg than pos

Signed Integers

given 0s and 1s,
how do I interpret it?

Two's Complement. Representation on N bytes

- Positive numbers:
  - Sign bit is 0
  - Binary representation of value on (8*N)-1 bits
- Negative numbers:
  - Binary representation of corresponding positive value on 8*N bits
  - Invert all digits (0 becomes 1; 1 becomes 0)
  - Add one

o.O why?

```
short int x =  15213;
short int y = -15213;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
y	-15213	C4 93	11000100 10010011

yup, that checks out. but why?

Why this works?

EXAMPLE WITH 8 bits

Substracting from 00000000 or substracting from 100000000 is the same for all practical purpose (as the borrowing is carried forward beyond the 8th bit). So, in 2’s complement **-x** is represented by

2^8 **– x** = 100000000 - x = 11111111 + 1 - x = (11111111 - x) + 1

As we have seen (now with 8 bits):
~x + x = 11111111 => (11111111 - x) = ~x

recall: ~x is x with bits flipped

In two’s complement on w bits

-x is represented as $2^w – x$

=> ~x + 1

in other words: -x is
“what must I add to x to get 0?”
(answer: ~x + 1)

Two’s complement

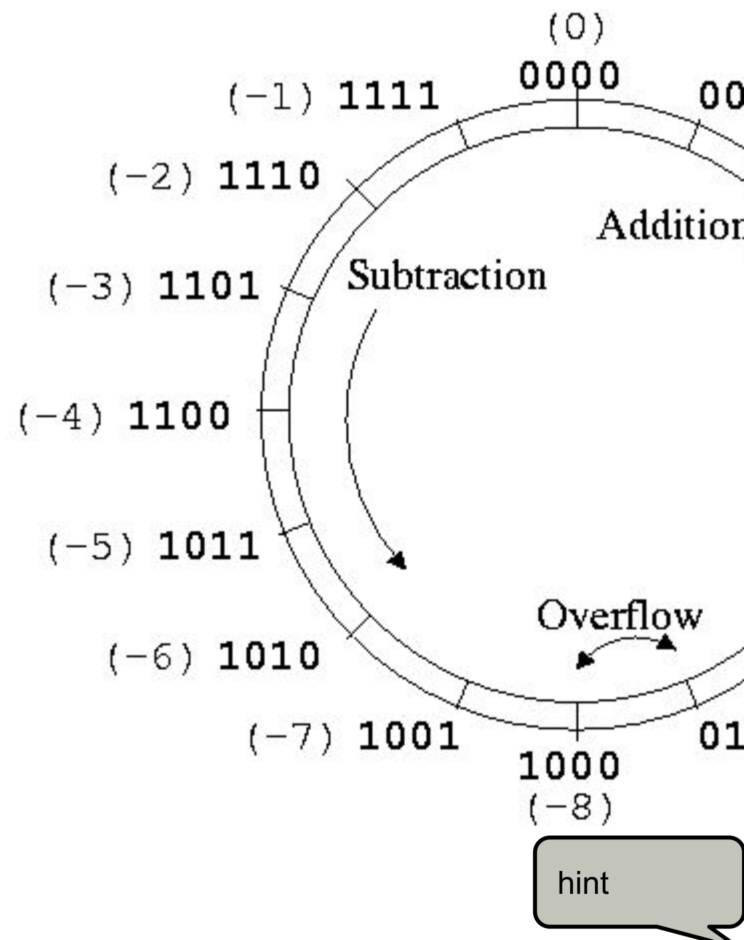
What does 0xFFFFFFFF represent?

How about 0x80000000?

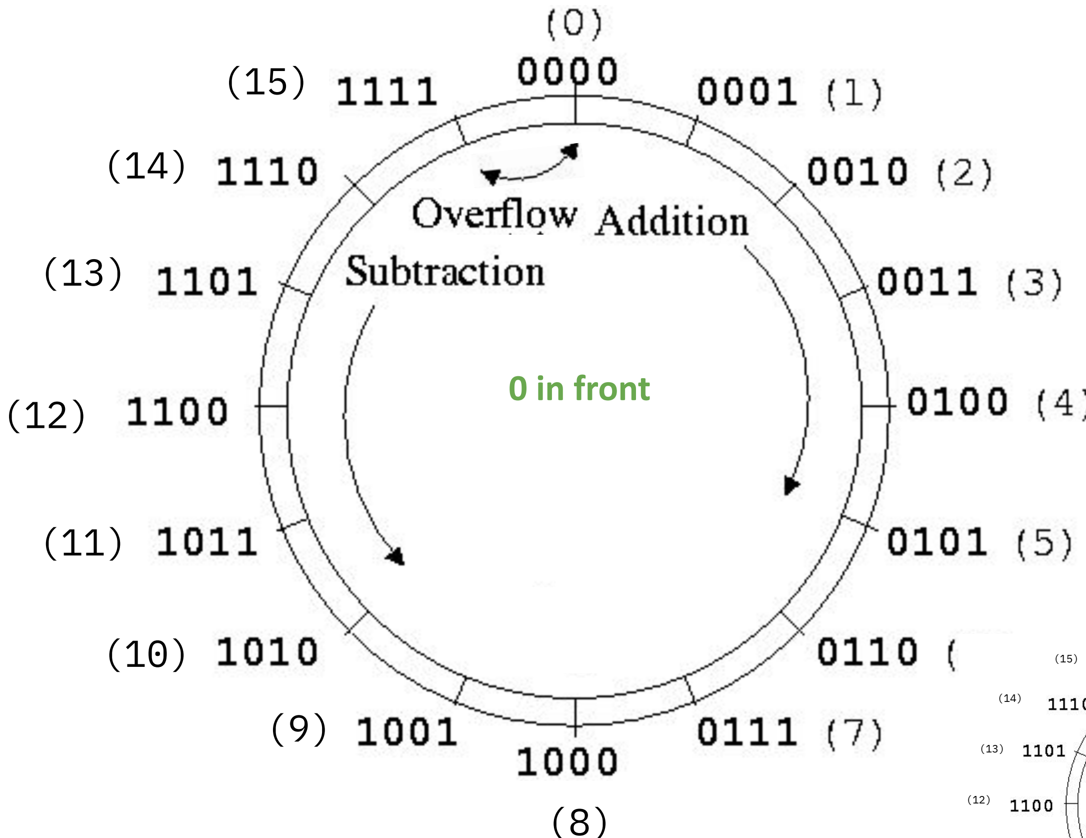
Two’s complement

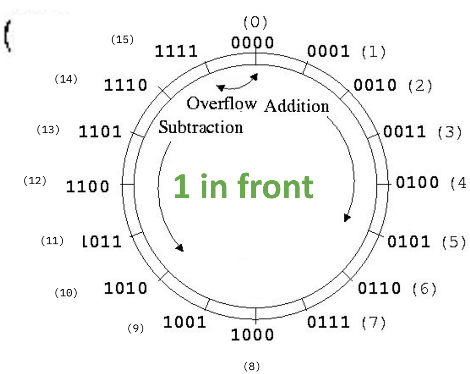
What does 0xFFFFFFFF represent?

How about 0x80000000?

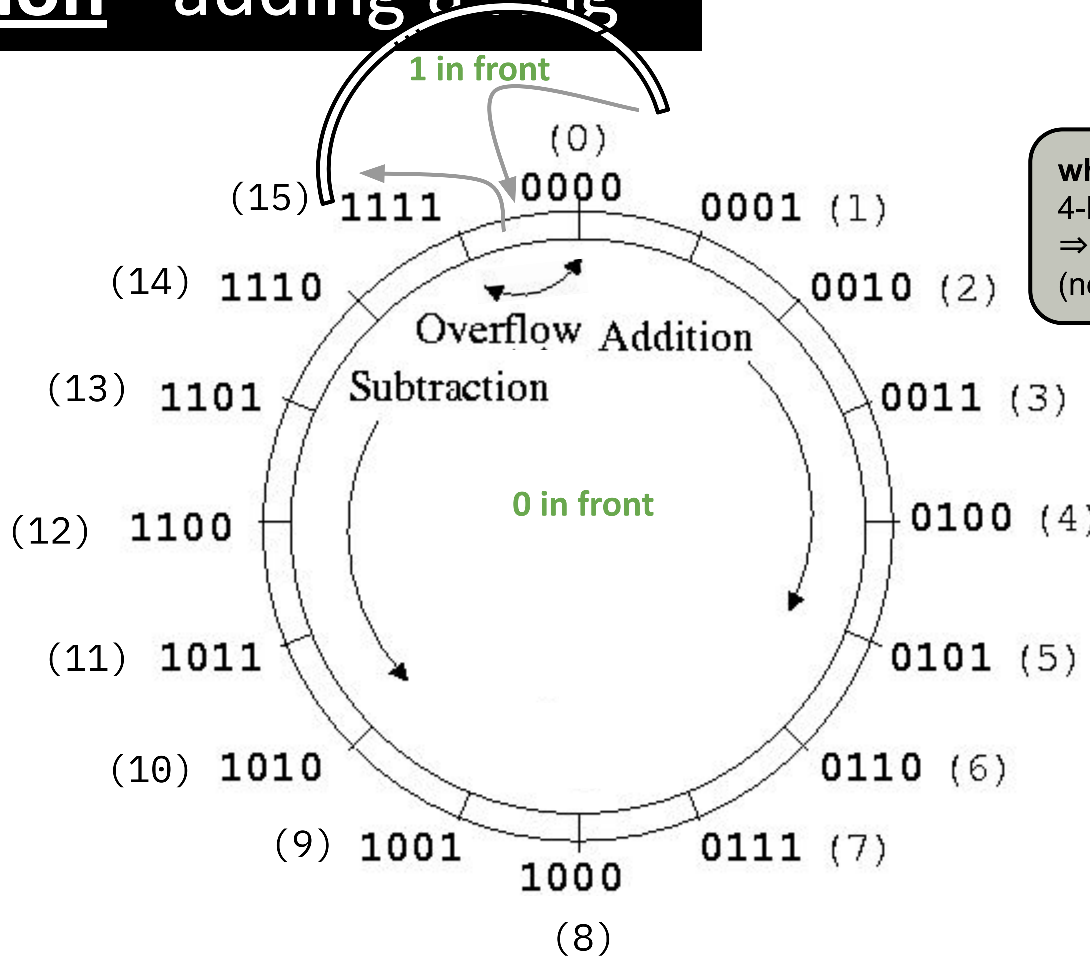


Bit **extension** - adding a ring



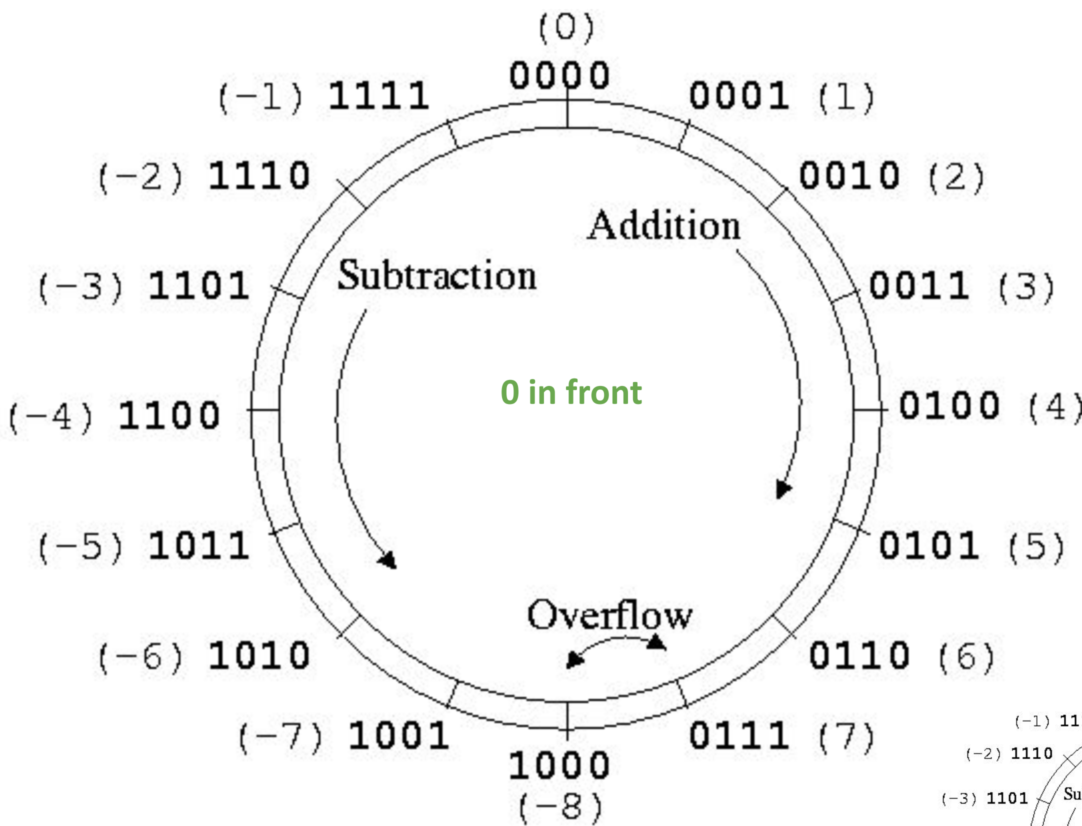


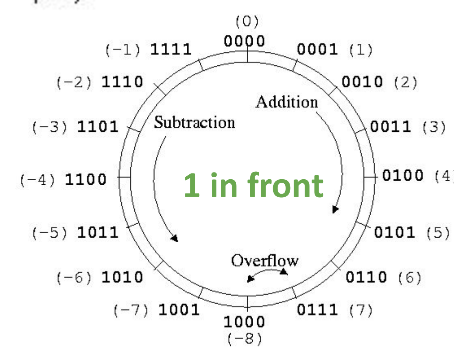
Bit **extension** - adding a ring



why this way: storing 4-bit value in 5-bit space ⇒ old bits are unchanged! (new 5th bit is just 0)

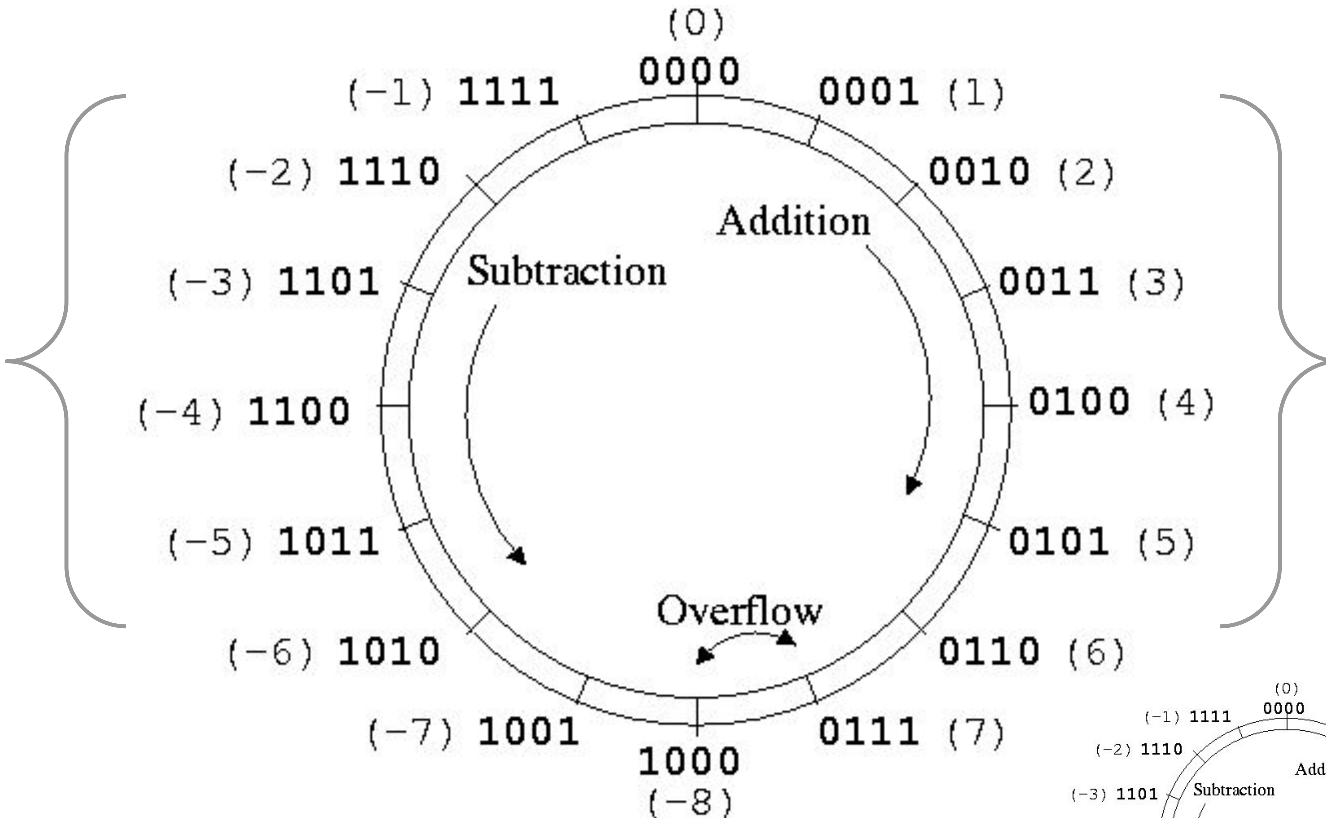
Sign extension - adding a ring (bit extension, signed)





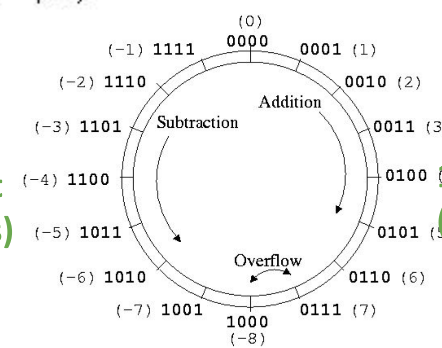
Sign extension - adding a ring (bit extension, signed)

1 in front
(old neg)



0 in front
(old pos)

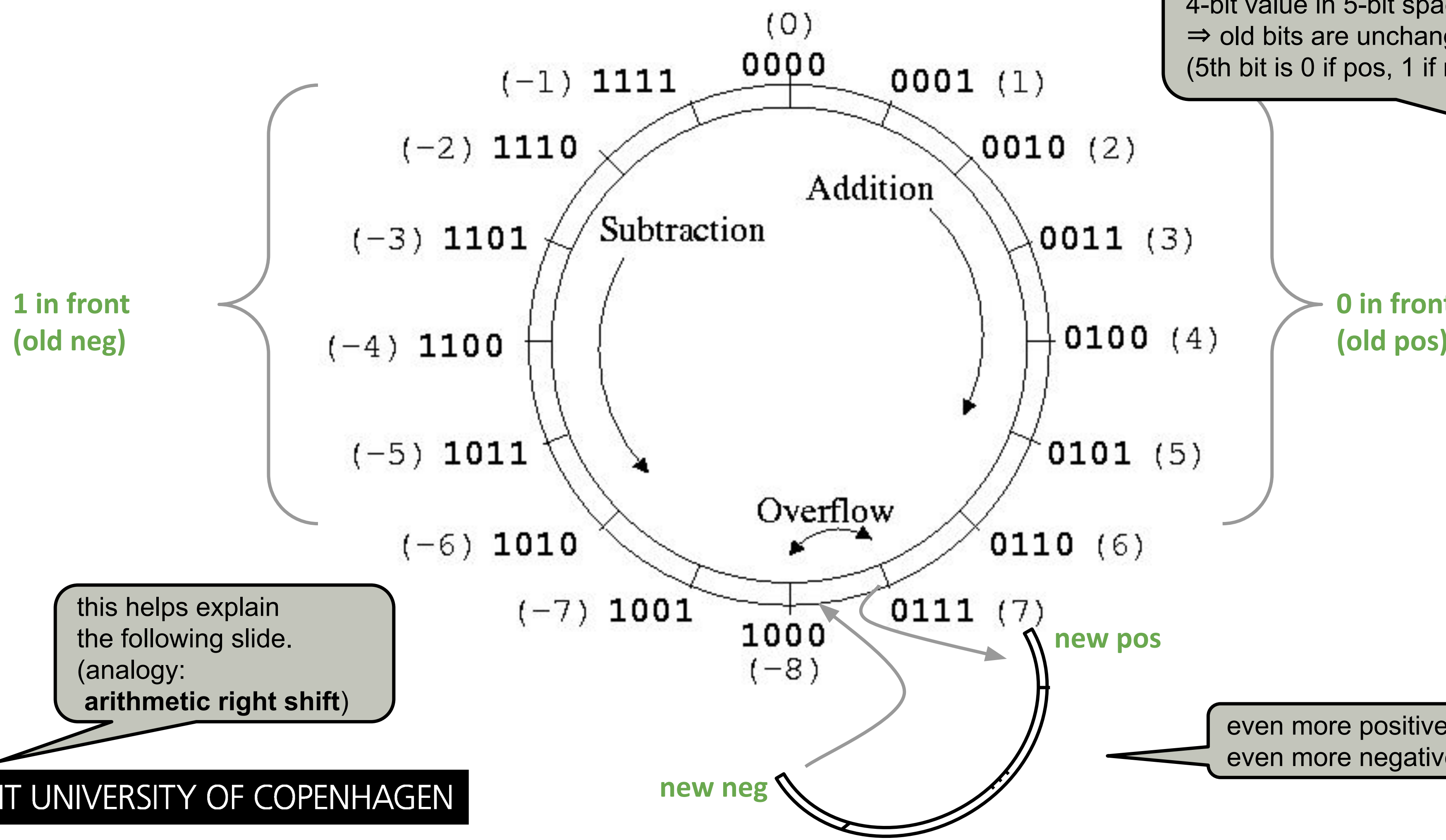
0 in front
(new pos)



1 in front
(new neg)

Sign extension - adding a ring (bit extension, signed)

why this way: storing 4-bit value in 5-bit space
⇒ old bits are unchanged!
(5th bit is 0 if pos, 1 if neg)



this helps explain the following slide. (analogy: **arithmetic right shift**)

even more positive, even more negative

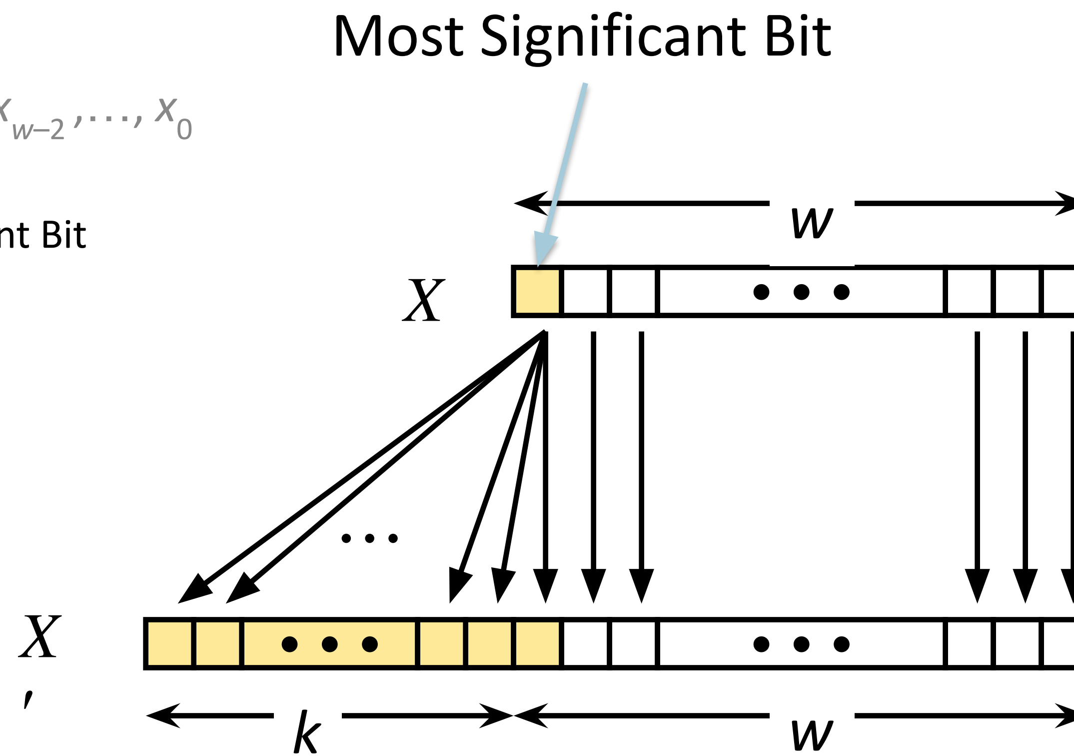
Sign Extension

Expanding: Converting from smaller to larger integer data type

Make k copies of sign bit:

$X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0$

k copies of Most Significant Bit



the overall value of negative numbers does not change.

Sign Extension

```
short int x =  15213;
int       ix = (int) x;
short int y = -15213;
int       iy = (int) y;
```

easy in C!
(one of the nice properties of two's complement)

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

C automatically performs sign extension

Sign Extension

Expanding (e.g., short int to int)

- Unsigned: zeros added (on left)
- Signed: sign extension (as shown on previous slides)
- Both yield expected result

Truncating (e.g., unsigned to unsigned short)

- Unsigned/signed: bits are truncated
- Result reinterpreted
- Unsigned: mod operation
- Signed: depends on bit pattern (large negative number might be truncated to positive number)

Why do we care?

In C type system,

If there is a mix of signed and unsigned values in an expression, then signed values are **implicitly cast to unsigned**:

- The bit pattern is maintained
- But re-interpreted!!
- Can have unexpected effects => adding or subtracting 2^N

Security, Revisited

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    . . .
}
```

l2ex1.c+

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

memcpy

```
MEMCPY(3)                    Linux Programmer's Manual                    MEMCPY(3)

NAME
       memcpy - copy memory area

SYNOPSIS
       #include <string.h>

       void *memcpy(void *dest, const void *src, size_t n);

DESCRIPTION
       The  memcpy()  function  copies  n  bytes from memory area src to memory area dest.  The memory areas must not
       overlap.  Use memmove(3) if the memory areas do overlap.

RETURN VALUE
       The memcpy() function returns a pointer to dest.
```

From GNU glibc manual /Appendix A – Language Features /Important Data Types:

Data Type: **size_t**

> This is an unsigned integer type used to represent the sizes of objects. The result of the `sizeof` operator is of this type, and functions such as `malloc` (see Unconstrained Allocation) and `memcpy` (see Copying Strings and Arrays) accept arguments of this type to specify object sizes. On systems using the GNU C Library, this will be `unsigned int` or `unsigned long int`.

Usage Note: `size_t` is the preferred way to declare any arguments or variables that hold the size of an object.

Security - Woops

```c
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    . . .
}
```

l2ex1.c+

```c
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

-528 in two's complement:

0xFFFFFDF0 → Reinterpreted as unsigned within memcpy: 4294966768 (decimal)

Why do we care?

Always be mindful/careful of unsigned integers!

Outline

1. Two’s complement
2. **Integer Arithmetic**
3. Bit Manipulation

Integer Arithmetic

How to deal with finite representation?

Adding two integers encoded on w bytes
Should take w+1 bytes
How to encode the sum on w bytes?

No magic!! The sum will **overflow**

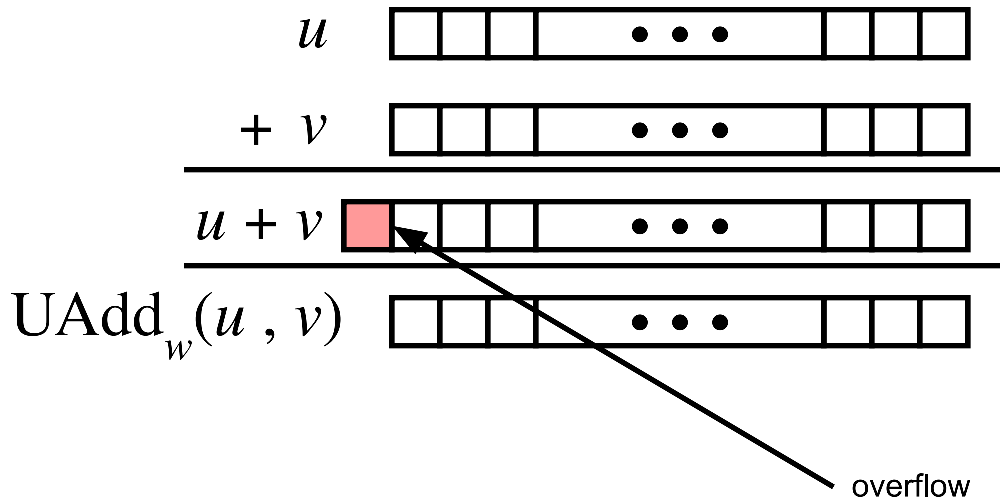
story time (Java standard library)

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



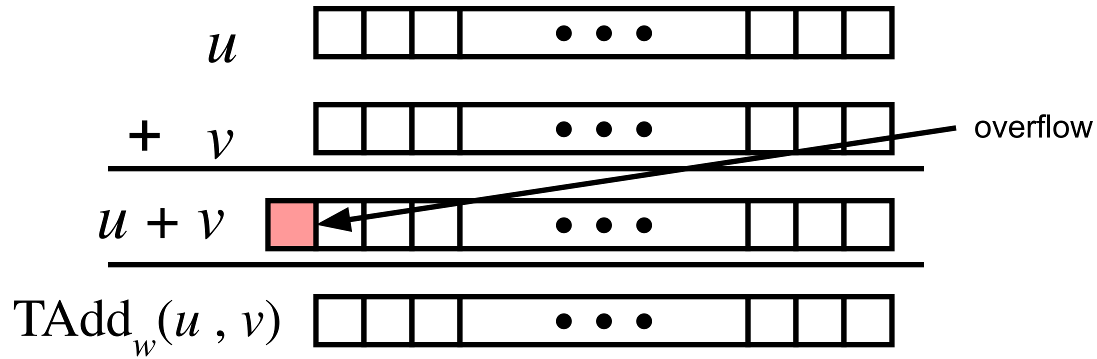
$UAdd_w(u , v)$ is u+v mod 2^w

Signed Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



TAdd and UAdd have Identical **Bit-Level** Behavior

Signed vs. unsigned addition in C:

```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v
```

Will give `s == t`

Signed Addition

Beware:
This is undefined behavior in C!

l2ex2.c+

```c
#include <stdio.h>
#include <limits.h>

int main() {
  // max short +1
  short int i = 0xFFFF + 1;
  printf("signed: 0xFFFF + 1 = %d\n", i);

  // adding to large negative numbers
  short int j = 0x8000;
  short int k = j + j;
  printf("signed: 0x8000 + 0x8000 = %d\n", k);

  return 0;
}
```

```
cc     l2ex2.c   -o l2ex2
l2ex2.c:6:24: warning: implicit conversion from 'int' to 'short' changes value from 65536 to 0 [-Wconstant-conversion]
  short int i = 0xFFFF + 1;
            ~   ~~~~~~~^~~
1 warning generated.
```

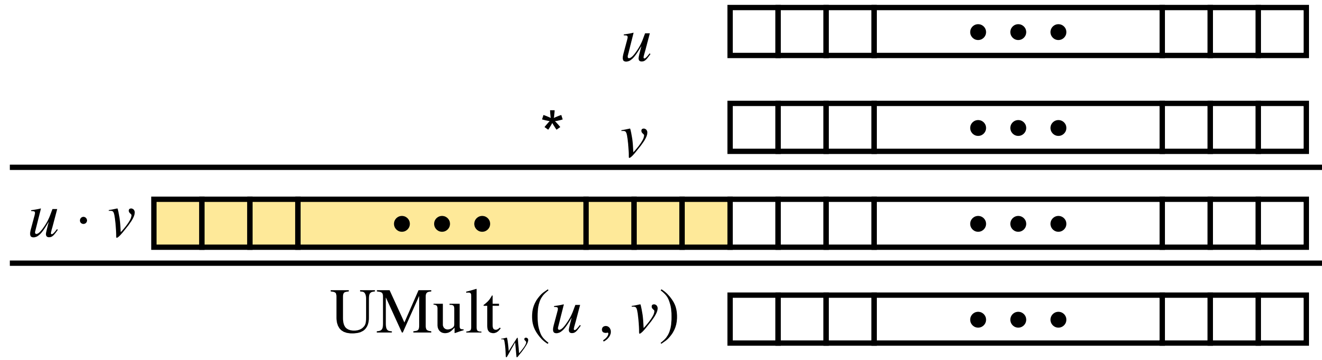
```
~/Documents/Tmp ❯ ./l2ex2
signed: 0xFFFF + 1 = 0
signed: 0x8000 + 0x8000 = 0
```

Unsigned Multiplication in C

Operands: *w* bits

True Product: 2**w* bits

Discard *w* bits: *w* bits



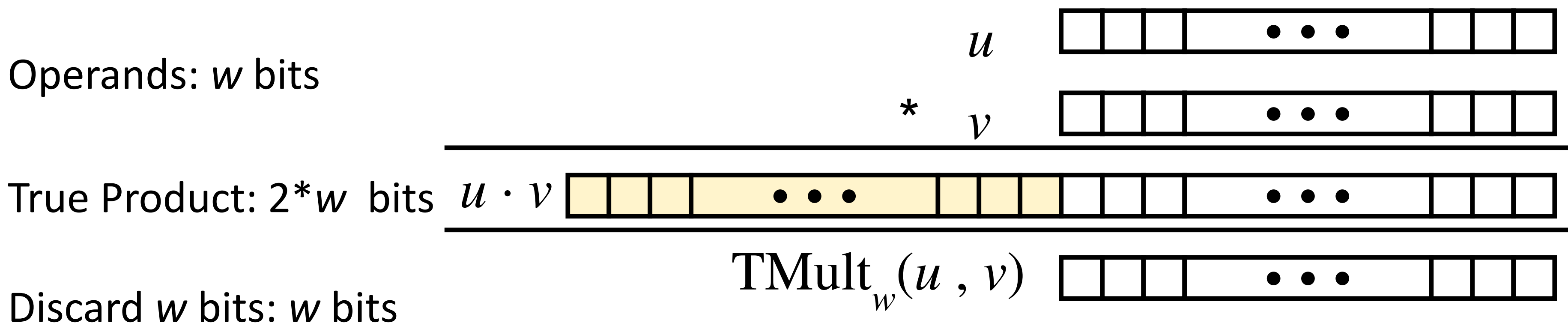
Standard Multiplication Function

Ignores high order *w* bits

Implements Modular Arithmetic

$UMult_w(u, v) = u \cdot v \bmod 2^w$

Signed Multiplication in C



Standard Multiplication Function

- Ignores high order *w* bits
- Some of which are different for signed vs. unsigned multiplication
- Lower bits are the same

Power-of-2 Multiply with Shift

Operation

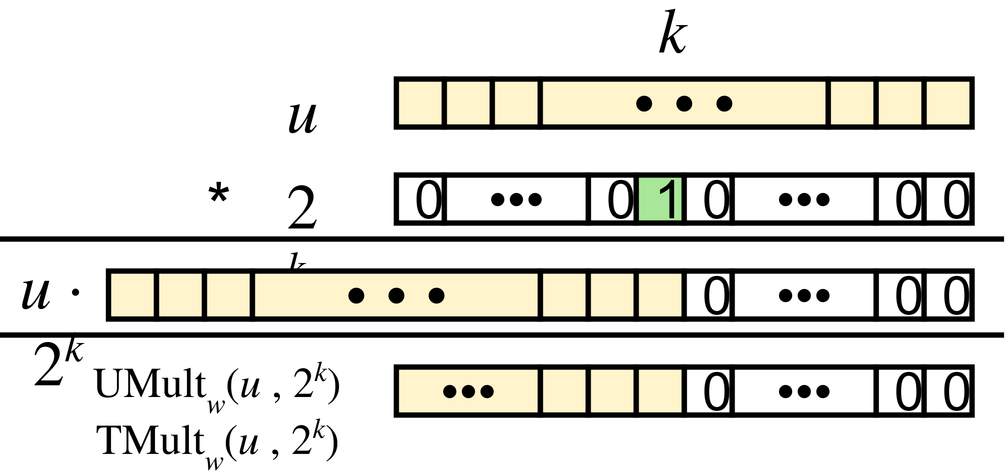
`u << k` gives **`u * 2`**^{k}

Both signed and unsigned

Operands: *w* bits

True Product: *w*+*k* bits

Discard *k* bits: *w* bits



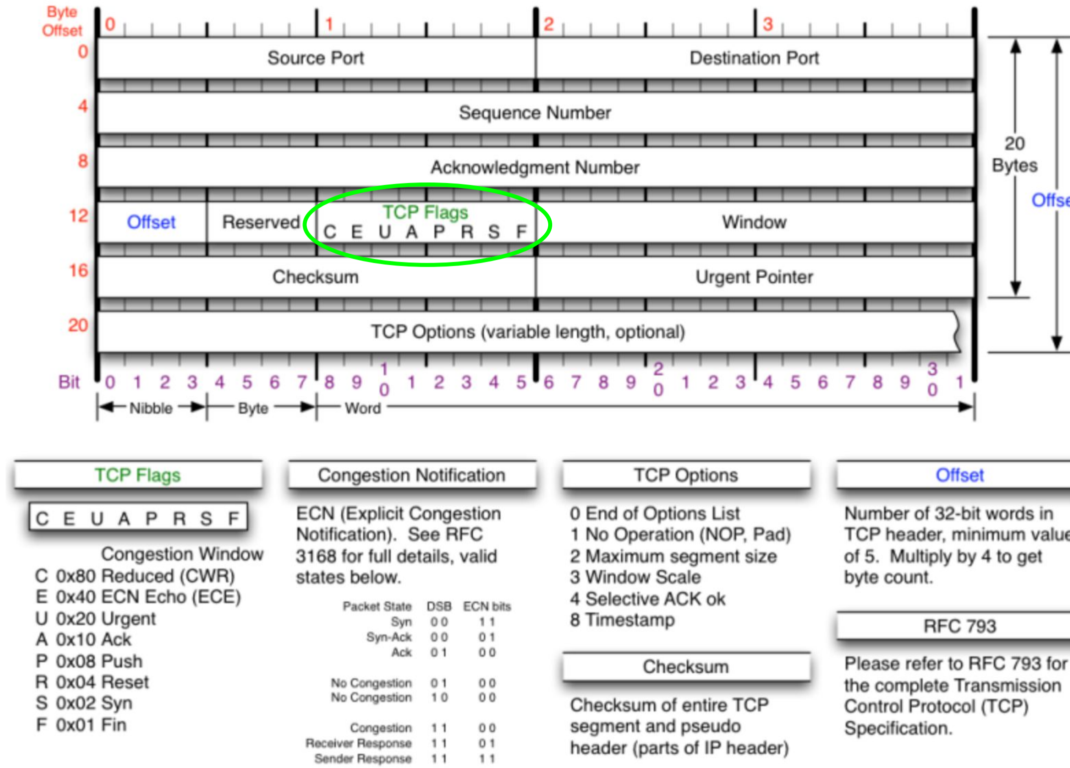
Examples

```
u << 3== u * 8
u << 5 - u << 3== u * 24
```

Outline

1. Two’s complement
2. Integer Arithmetic
3. **Bit Manipulation**

Bit Representation



TCP Header

Is Ack flag set?
Is any TCP flag set?
Is a single flag set?
How many TCP flags are set?

small puzzles;
what the assignment is about

to answer these,
you need to work with bitwise representations.

Sequence of bits and boolean

EXAMPLE WITH 8 bits

- 00000000 interpreted as **False**
- Byte containing at least a 1, e.g., 00100010 interpreted as **True**
- X is a sequence of bit
  - X interpreted as Boolean expression (X != 0)
  - !X interpreted as Boolean expression (X == 0)

Beware difference between:

- **Bitwise operations** applied on sequence of bits
  &, |, ~, ^, >>, <<
- **Logical operations** applied on Booleans
  &&, ||, !

TCP Problem

Consider the TCP flags encoded on 8 bits as X.

How do you test whether the Ack flag (0x10) is set?

How do you test whether any flag is set?

How do you test if a single flag is set?

How do you count the number of flags set?

Answers

- How do you test whether the Ack flag (0x10) is set?

  (X & 0x10)

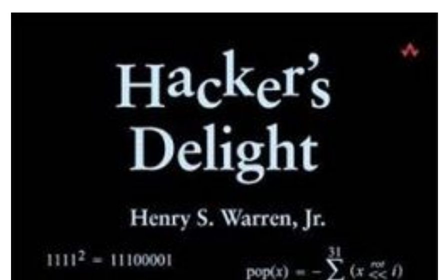
  => interpreted as false if flag not set, true otherwise
- How do you test whether any flag is set?

  (X)

  => interpreted as false if no flag is set , true otherwise
- How do you test if a single flag is set?

  ((X & (X-1)) == 0) && (X != 0)

  => (X & (X-1)) transforms the rightmost 1-bit into a 0-bit



small puzzles!
some hate it; we like it (beauty)

Answers

How do you count the number of flags set?

Divide and conquer

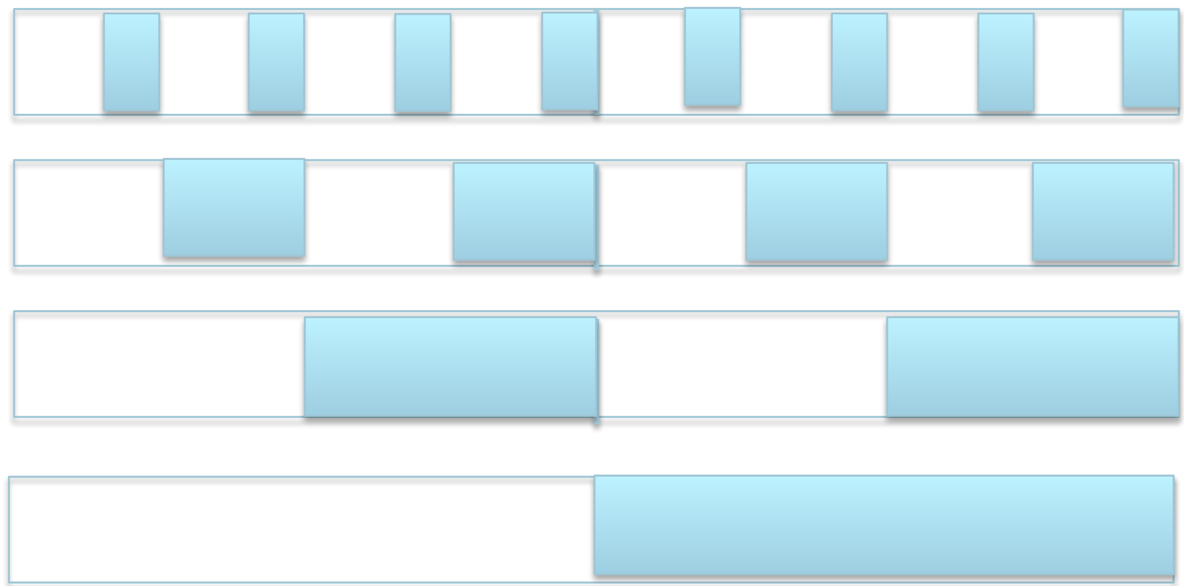
Trivial for 2 bits

- 0+0 -> 00
- 0+1 -> 01
- 1+0 -> 01
- 1+1 -> 10

Answers

x is a short: 2B

Masks



& 0x5555

& 0x3333

& 0x0F0F

& 0x00FF

Bitwise operations in parallel

x = (x & 0x5555) + ((x >> 1) & 0x5555)
x = (x & 0x3333) + ((x >> 2) & 0x3333)
x = (x & 0x0F0F) + ((x >> 4) & 0x0F0F)
x = (x & 0x00FF) + ((x >> 8) & 0x00FF)

Motivation: Performance

Fast binary matrix operations

```
#include <math.h>
#include <stdint.h>
#include <malloc.h>
#include <stdio.h>

#define N 8
#define M 8
#define n 0
#define m 1

int main(int argc, char *argv[])
{
        /* binary matrix allocation */
        uint64_t *matrix;
        matrix  = (uint64_t *) malloc((size_t) ceil((N*M)/sizeof(uint64_t)));
        *matrix = 0x1001b;
        /* reading the [n][m]-th element of an NxM binary matrix  */
        uint64_t mask = 1U << ((n*N+m)%64U);
        uint64_t result = mask & matrix[(size_t) ceil((n*N+m)/(64*1.0))];
        result = result >> ((n*N+m)%64U);
        printf("matrix[n][m]: %lu\n",result);
        return 0;
}
```


https://www.quora.com/What-is-the-best-way-to-declare-binary-matrices-in-C

Take Aways

Two complements used to represent signed integers

Two complements procedure is:

1. Write positive value in binary
2. Invert all digits
3. Add 1

Beware **implicit cast** to unsigned in C!!

security

Bit manipulation enables **efficient operations**.
It is a rich algorithmic playground.

performance