

CS 241: Systems Programming

Lecture 13. Bits and Bytes 2

Spring 2020

Prof. Stephen Checkoway

Internal data representation

Data are stored in binary

32 bit unsigned integer values are:

Internal data representation

Data are stored in binary

32 bit unsigned integer values are:

00000000 00000000 00000000 00000000 = 0

Internal data representation

Data are stored in binary

32 bit unsigned integer values are:

00000000 00000000 00000000 00000000 = 0

00000000 00000000 00000000 00000001 = 1

Internal data representation

Data are stored in binary

32 bit unsigned integer values are:

00000000 00000000 00000000 00000000 = 0

00000000 00000000 00000000 00000001 = 1

00000000 00000000 00000000 00000010 = 2

Internal data representation

Data are stored in binary

32 bit unsigned integer values are:

00000000 00000000 00000000 00000000 = 0

00000000 00000000 00000000 00000001 = 1

00000000 00000000 00000000 00000010 = 2

00000000 00000000 00000000 00000011 = 3

Internal data representation

Data are stored in binary

32 bit unsigned integer values are:

00000000 00000000 00000000 00000000 = 0

00000000 00000000 00000000 00000001 = 1

00000000 00000000 00000000 00000010 = 2

00000000 00000000 00000000 00000011 = 3

...

11111111 11111111 11111111 11111111 = $2^{32}-1$

Bitwise operators

Binary operators apply the operation to the corresponding bits of the operands

- ▶ $x \ \& \ y$ — bitwise AND
- ▶ $x \ | \ y$ — bitwise OR
- ▶ $x \ ^ \ y$ — bitwise XOR

Unary operator applies the operation to each bit

- ▶ $\sim x$ — one's complement (flip each bit)

Review of Boolean logic

A	B	$\sim A$	$A \& B$	$A \mid B$	$A \wedge B$
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

What is the value of $0x4E$ & $0x1F$?

- A. $0xE$
- B. $0x51$
- C. $0x5F$
- D. $0xB1$
- E. $0xE0$

Hex Binary		Hex Binary	
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Bit shifting

Bit shifting

Manipulates the position of bits

Bit shifting

Manipulates the position of bits

- Left shift fills with 0 bits

Bit shifting

Manipulates the position of bits

- Left shift fills with 0 bits
- Right shift of **unsigned** variable fills with 0 bits

Bit shifting

Manipulates the position of bits

- Left shift fills with 0 bits
- Right shift of **unsigned** variable fills with 0 bits
- Right shift of **signed** variable fills with sign bit (Actually implementation defined if negative!)

Bit shifting

Manipulates the position of bits

- Left shift fills with 0 bits
- Right shift of **unsigned** variable fills with 0 bits
- Right shift of **signed** variable fills with sign bit (Actually implementation defined if negative!)

x << 2; // shifts bits of **x** two positions left

Bit shifting

Manipulates the position of bits

- Left shift fills with 0 bits
- Right shift of **unsigned** variable fills with 0 bits
- Right shift of **signed** variable fills with sign bit (Actually implementation defined if negative!)

x << **2**; // shifts bits of **x** two positions left

- Same as multiplying by 4

Bit shifting

Manipulates the position of bits

- Left shift fills with 0 bits
- Right shift of **unsigned** variable fills with 0 bits
- Right shift of **signed** variable fills with sign bit (Actually implementation defined if negative!)

x << 2; // shifts bits of **x** two positions left

- Same as multiplying by 4

x >> 3; // shifts bits of **x** three positions right

Bit shifting

Manipulates the position of bits

- Left shift fills with 0 bits
- Right shift of **unsigned** variable fills with 0 bits
- Right shift of **signed** variable fills with sign bit (Actually implementation defined if negative!)

x << 2; // shifts bits of **x** two positions left

- Same as multiplying by 4

x >> 3; // shifts bits of **x** three positions right

- Same as dividing by 8 (if x is unsigned)

What does the following do?

```
x = ( (x >> 2) << 2 );
```

- A. Changes x to be positive
- B. Sets the least significant two bits to 0
- C. Sets the most significant two bits to 0
- D. Gives an integer overflow error
- E. Implementation-defined behavior

Testing if a bit is set (i.e., is 1)

```
#include <stdbool.h>
```

```
// Returns true if the nth bit of x is 1.
```

```
bool is_bit_set(unsigned int x, unsigned int n) {  
    return x & (1u << n); // 1u is an unsigned int with value 1.  
}
```

Testing if a bit is set (i.e., is 1)

```
#include <stdbool.h>
```

```
// Returns true if the nth bit of x is 1.
```

```
bool is_bit_set(unsigned int x, unsigned int n) {  
    return x & (1u << n); // 1u is an unsigned int with value 1.  
}
```

`1u << n` gives an integer with only the nth bit set

Testing if a bit is set (i.e., is 1)

```
#include <stdbool.h>
```

```
// Returns true if the nth bit of x is 1.
```

```
bool is_bit_set(unsigned int x, unsigned int n) {  
    return x & (1u << n); // 1u is an unsigned int with value 1.  
}
```

`1u << n` gives an integer with only the n th bit set

If the n th bit is 1, then `x & (1u << n)` is `1u << n` which is nonzero.

If the n th bit is 0, then `x & (1u << n)` is 0

Testing if a bit is set (i.e., is 1)

```
#include <stdbool.h>
```

```
// Returns true if the nth bit of x is 1.
```

```
bool is_bit_set(unsigned int x, unsigned int n) {  
    return x & (1u << n); // 1u is an unsigned int with value 1.  
}
```

`1u << n` gives an integer with only the n th bit set

If the n th bit is 1, then `x & (1u << n)` is `1u << n` which is nonzero.

If the n th bit is 0, then `x & (1u << n)` is 0

What happens if n is too large?

Testing if a bit is set (i.e., is 1)

```
#include <stdbool.h>
```

```
// Returns true if the nth bit of x is 1.
```

```
bool is_bit_set(unsigned int x, unsigned int n) {  
    return x & (1u << n); // 1u is an unsigned int with value 1.  
}
```

`1u << n` gives an integer with only the n th bit set

If the n th bit is 1, then `x & (1u << n)` is `1u << n` which is nonzero.

If the n th bit is 0, then `x & (1u << n)` is 0

What happens if n is too large?

- Undefined behavior!

UB

```
#include <err.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

// Returns true if the nth bit of x is 1.
bool is_bit_set(unsigned int x, unsigned int n) {
    return x & (1u << n);
}

int main(int argc, char **argv) {
    if (argc != 3)
        errx(1, "Usage: %s integer bit", argv[0]);
    unsigned int x = atoi(argv[1]);
    unsigned int n = atoi(argv[2]);
    if (is_bit_set(x, n))
        printf("Bit %u of %u is 1\n", n, x);
    else
        printf("Bit %u of %u is 0\n", n, x);
    return 0;
}
```

UB

```
$ ./bad_shift 3 0
```

```
#include <err.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

// Returns true if the nth bit of x is 1.
bool is_bit_set(unsigned int x, unsigned int n) {
    return x & (1u << n);
}

int main(int argc, char **argv) {
    if (argc != 3)
        errx(1, "Usage: %s integer bit", argv[0]);
    unsigned int x = atoi(argv[1]);
    unsigned int n = atoi(argv[2]);
    if (is_bit_set(x, n))
        printf("Bit %u of %u is 1\n", n, x);
    else
        printf("Bit %u of %u is 0\n", n, x);
    return 0;
}
```

UB

```
$ ./bad_shift 3 0  
Bit 0 of 3 is 1
```

```
#include <err.h>  
#include <stdbool.h>  
#include <stdio.h>  
#include <stdlib.h>  
  
// Returns true if the nth bit of x is 1.  
bool is_bit_set(unsigned int x, unsigned int n) {  
    return x & (1u << n);  
}  
  
int main(int argc, char **argv) {  
    if (argc != 3)  
        errx(1, "Usage: %s integer bit", argv[0]);  
    unsigned int x = atoi(argv[1]);  
    unsigned int n = atoi(argv[2]);  
    if (is_bit_set(x, n))  
        printf("Bit %u of %u is 1\n", n, x);  
    else  
        printf("Bit %u of %u is 0\n", n, x);  
    return 0;  
}
```

UB

```
$ ./bad_shift 3 0
Bit 0 of 3 is 1
$ ./bad_shift 3 1
```

```
#include <err.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

// Returns true if the nth bit of x is 1.
bool is_bit_set(unsigned int x, unsigned int n) {
    return x & (1u << n);
}

int main(int argc, char **argv) {
    if (argc != 3)
        errx(1, "Usage: %s integer bit", argv[0]);
    unsigned int x = atoi(argv[1]);
    unsigned int n = atoi(argv[2]);
    if (is_bit_set(x, n))
        printf("Bit %u of %u is 1\n", n, x);
    else
        printf("Bit %u of %u is 0\n", n, x);
    return 0;
}
```

UB

```
$ ./bad_shift 3 0
Bit 0 of 3 is 1
$ ./bad_shift 3 1
Bit 1 of 3 is 1
```

```
#include <err.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

// Returns true if the nth bit of x is 1.
bool is_bit_set(unsigned int x, unsigned int n) {
    return x & (1u << n);
}

int main(int argc, char **argv) {
    if (argc != 3)
        errx(1, "Usage: %s integer bit", argv[0]);
    unsigned int x = atoi(argv[1]);
    unsigned int n = atoi(argv[2]);
    if (is_bit_set(x, n))
        printf("Bit %u of %u is 1\n", n, x);
    else
        printf("Bit %u of %u is 0\n", n, x);
    return 0;
}
```

UB

```
$ ./bad_shift 3 0
Bit 0 of 3 is 1
$ ./bad_shift 3 1
Bit 1 of 3 is 1
$ ./bad_shift 3 2
```

```
#include <err.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

// Returns true if the nth bit of x is 1.
bool is_bit_set(unsigned int x, unsigned int n) {
    return x & (1u << n);
}

int main(int argc, char **argv) {
    if (argc != 3)
        errx(1, "Usage: %s integer bit", argv[0]);
    unsigned int x = atoi(argv[1]);
    unsigned int n = atoi(argv[2]);
    if (is_bit_set(x, n))
        printf("Bit %u of %u is 1\n", n, x);
    else
        printf("Bit %u of %u is 0\n", n, x);
    return 0;
}
```


UB

```
$ ./bad_shift 3 0
Bit 0 of 3 is 1
$ ./bad_shift 3 1
Bit 1 of 3 is 1
$ ./bad_shift 3 2
Bit 2 of 3 is 0
```

```
#include <err.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

// Returns true if the nth bit of x is 1.
bool is_bit_set(unsigned int x, unsigned int n) {
    return x & (1u << n);
}

int main(int argc, char **argv) {
    if (argc != 3)
        errx(1, "Usage: %s integer bit", argv[0]);
    unsigned int x = atoi(argv[1]);
    unsigned int n = atoi(argv[2]);
    if (is_bit_set(x, n))
        printf("Bit %u of %u is 1\n", n, x);
    else
        printf("Bit %u of %u is 0\n", n, x);
    return 0;
}
```


UB

```
$ ./bad_shift 3 0
Bit 0 of 3 is 1
$ ./bad_shift 3 1
Bit 1 of 3 is 1
$ ./bad_shift 3 2
Bit 2 of 3 is 0
$ ./bad_shift 3 32
```

```
#include <err.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

// Returns true if the nth bit of x is 1.
bool is_bit_set(unsigned int x, unsigned int n) {
    return x & (1u << n);
}

int main(int argc, char **argv) {
    if (argc != 3)
        errx(1, "Usage: %s integer bit", argv[0]);
    unsigned int x = atoi(argv[1]);
    unsigned int n = atoi(argv[2]);
    if (is_bit_set(x, n))
        printf("Bit %u of %u is 1\n", n, x);
    else
        printf("Bit %u of %u is 0\n", n, x);
    return 0;
}
```

UB

```
$ ./bad_shift 3 0
Bit 0 of 3 is 1
$ ./bad_shift 3 1
Bit 1 of 3 is 1
$ ./bad_shift 3 2
Bit 2 of 3 is 0
$ ./bad_shift 3 32
Bit 32 of 3 is 1
```

```
#include <err.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

// Returns true if the nth bit of x is 1.
bool is_bit_set(unsigned int x, unsigned int n) {
    return x & (1u << n);
}

int main(int argc, char **argv) {
    if (argc != 3)
        errx(1, "Usage: %s integer bit", argv[0]);
    unsigned int x = atoi(argv[1]);
    unsigned int n = atoi(argv[2]);
    if (is_bit_set(x, n))
        printf("Bit %u of %u is 1\n", n, x);
    else
        printf("Bit %u of %u is 0\n", n, x);
    return 0;
}
```

UB

```
$ ./bad_shift 3 0
Bit 0 of 3 is 1
$ ./bad_shift 3 1
Bit 1 of 3 is 1
$ ./bad_shift 3 2
Bit 2 of 3 is 0
$ ./bad_shift 3 32
Bit 32 of 3 is 1
$ ./bad_shift 3 33
```

```
#include <err.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

// Returns true if the nth bit of x is 1.
bool is_bit_set(unsigned int x, unsigned int n) {
    return x & (1u << n);
}

int main(int argc, char **argv) {
    if (argc != 3)
        errx(1, "Usage: %s integer bit", argv[0]);
    unsigned int x = atoi(argv[1]);
    unsigned int n = atoi(argv[2]);
    if (is_bit_set(x, n))
        printf("Bit %u of %u is 1\n", n, x);
    else
        printf("Bit %u of %u is 0\n", n, x);
    return 0;
}
```

UB

```
$ ./bad_shift 3 0
Bit 0 of 3 is 1
$ ./bad_shift 3 1
Bit 1 of 3 is 1
$ ./bad_shift 3 2
Bit 2 of 3 is 0
$ ./bad_shift 3 32
Bit 32 of 3 is 1
$ ./bad_shift 3 33
Bit 33 of 3 is 1
```

```
#include <err.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

// Returns true if the nth bit of x is 1.
bool is_bit_set(unsigned int x, unsigned int n) {
    return x & (1u << n);
}

int main(int argc, char **argv) {
    if (argc != 3)
        errx(1, "Usage: %s integer bit", argv[0]);
    unsigned int x = atoi(argv[1]);
    unsigned int n = atoi(argv[2]);
    if (is_bit_set(x, n))
        printf("Bit %u of %u is 1\n", n, x);
    else
        printf("Bit %u of %u is 0\n", n, x);
    return 0;
}
```

UB

```
$ ./bad_shift 3 0
Bit 0 of 3 is 1
$ ./bad_shift 3 1
Bit 1 of 3 is 1
$ ./bad_shift 3 2
Bit 2 of 3 is 0
$ ./bad_shift 3 32
Bit 32 of 3 is 1
$ ./bad_shift 3 33
Bit 33 of 3 is 1
$ ./bad_shift 3 34
```

```
#include <err.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

// Returns true if the nth bit of x is 1.
bool is_bit_set(unsigned int x, unsigned int n) {
    return x & (1u << n);
}

int main(int argc, char **argv) {
    if (argc != 3)
        errx(1, "Usage: %s integer bit", argv[0]);
    unsigned int x = atoi(argv[1]);
    unsigned int n = atoi(argv[2]);
    if (is_bit_set(x, n))
        printf("Bit %u of %u is 1\n", n, x);
    else
        printf("Bit %u of %u is 0\n", n, x);
    return 0;
}
```


UB

```
$ ./bad_shift 3 0
Bit 0 of 3 is 1
$ ./bad_shift 3 1
Bit 1 of 3 is 1
$ ./bad_shift 3 2
Bit 2 of 3 is 0
$ ./bad_shift 3 32
Bit 32 of 3 is 1
$ ./bad_shift 3 33
Bit 33 of 3 is 1
$ ./bad_shift 3 34
Bit 34 of 3 is 0
```

```
#include <err.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

// Returns true if the nth bit of x is 1.
bool is_bit_set(unsigned int x, unsigned int n) {
    return x & (1u << n);
}

int main(int argc, char **argv) {
    if (argc != 3)
        errx(1, "Usage: %s integer bit", argv[0]);
    unsigned int x = atoi(argv[1]);
    unsigned int n = atoi(argv[2]);
    if (is_bit_set(x, n))
        printf("Bit %u of %u is 1\n", n, x);
    else
        printf("Bit %u of %u is 0\n", n, x);
    return 0;
}
```

Testing if a bit is set (i.e., is 1)

```
#include <assert.h>
#include <limits.h>
#include <stdbool.h>
```

```
// Returns true if the nth bit of x is 1.
```

```
bool is_bit_set(unsigned int x, unsigned int n) {
    // assert(cond) will abort at runtime if cond is false.
    assert(n < CHAR_BIT * sizeof x);
    return x & (1u << n); // 1u is an unsigned int with value 1.
}
```

Testing if a bit is set (i.e., is 1)

```
#include <assert.h>
#include <limits.h>
#include <stdbool.h>
```

```
// Returns true if the nth bit of x is 1.
```

```
bool is_bit_set(unsigned int x, unsigned int n) {
    // assert(cond) will abort at runtime if cond is false.
    assert(n < CHAR_BIT * sizeof x);
    return x & (1u << n); // 1u is an unsigned int with value 1.
}
```

E.g., if **CHAR_BIT** is 8 and **sizeof** x is 4, then n must be less than 32 or the program aborts

Setting a bit (to 1)

```
// Returns the value of x with the nth bit set to 1.
unsigned int set_bit(unsigned int x, unsigned int n) {
    assert(n < CHAR_BIT * sizeof x);
    return x | (1u << n);
}
```

Clearing a bit (setting it to 0)

```
// Returns the value of x with the nth bit set to 0.  
unsigned int set_bit(unsigned int x, unsigned int n) {  
    assert(n < CHAR_BIT * sizeof x);  
    return x & ~(1u << n);  
}
```

Clearing a bit (setting it to 0)

```
// Returns the value of x with the nth bit set to 0.  
unsigned int set_bit(unsigned int x, unsigned int n) {  
    assert(n < CHAR_BIT * sizeof x);  
    return x & ~(1u << n);  
}
```

1u << n gives an integer with just the nth bit set

Clearing a bit (setting it to 0)

```
// Returns the value of x with the nth bit set to 0.  
unsigned int set_bit(unsigned int x, unsigned int n) {  
    assert(n < CHAR_BIT * sizeof x);  
    return x & ~(1u << n);  
}
```

1u << n gives an integer with just the nth bit set

~(**1u** << n) gives an integer with all bits set *except* the nth bit

Given an unsigned integer `x` with some value, what value should we use for `mask` to clear all of the bits of `x` except for the least significant 5 bits?

```
unsigned int x = /* ... */;    // Given some value here,  
unsigned int mask = /* ... */; // what value goes here  
x = x & mask;                  // to clear the required bits?
```

A. `0x5u`

B. `~0x5u`

C. `0x1Fu`

D. `~0x1Fu`

E. `sizeof x - 5`

Given an unsigned integer `x` with some value, what value should we use for `mask` to clear the 5 least significant bits of `x`?

```
unsigned int x = /* ... */;    // Given some value here,  
unsigned int mask = /* ... */; // what value goes here  
x = x & mask;                  // to clear the required bits?
```

A. `0x5u`

B. `~0x5u`

C. `0x1Fu`

D. `~0x1Fu`

E. `sizeof x - 5`

Combining flags via |

Specify flags via individual bits

Combine flags with |

E.g., set file system permissions via the flags

`S_I{R,W,X}{USR,GRP,OTH}`

```
#define S_IRWXU 0000700 /* RWX mask for owner */
#define S_IRUSR 0000400 /* R for owner */
#define S_IWUSR 0000200 /* W for owner */
#define S_IXUSR 0000100 /* X for owner */
```

```
#define S_IRWXG 0000070 /* RWX mask for group */
#define S_IRGRP 0000040 /* R for group */
#define S_IWGRP 0000020 /* W for group */
#define S_IXGRP 0000010 /* X for group */
```

```
#define S_IRWXO 0000007 /* RWX mask for other */
#define S_IROTH 0000004 /* R for other */
#define S_IWOTH 0000002 /* W for other */
#define S_IXOTH 0000001 /* X for other */
```

```
int chmod(char const *path, mode_t mode);
```

Negative numbers

Negative numbers

Usually stored using two's complement

- Take the magnitude of the number
- Invert all of the bits
- Add 1

Negative numbers

Usually stored using two's complement

- Take the magnitude of the number
- Invert all of the bits
- Add 1

representation of -5 in 8 bits

Negative numbers

Usually stored using two's complement

- Take the magnitude of the number
- Invert all of the bits
- Add 1

representation of -5 in 8 bits
magnitude: 0000_0101

Negative numbers

Usually stored using two's complement

- Take the magnitude of the number
- Invert all of the bits
- Add 1

representation of -5 in 8 bits

magnitude: 0000_0101

invert bits: 1111_1010

Negative numbers

Usually stored using two's complement

- Take the magnitude of the number
- Invert all of the bits
- Add 1

representation of -5 in 8 bits

magnitude: 0000_0101

invert bits: 1111_1010

Add 1: 1111_1011

Negative numbers

Usually stored using two's complement

- Take the magnitude of the number
- Invert all of the bits
- Add 1

Computing $-x$ from x (regardless of sign)

- Invert all of the bits
- Add 1

representation of -5 in 8 bits

magnitude: 0000_0101

invert bits: 1111_1010

Add 1: 1111_1011

Negative numbers

Usually stored using two's complement

- Take the magnitude of the number
- Invert all of the bits
- Add 1

Computing $-x$ from x (regardless of sign)

- Invert all of the bits
- Add 1

representation of -5 in 8 bits

magnitude: 0000_0101

invert bits: 1111_1010

Add 1: 1111_1011

$-(-5)$ in 8 bits

Negative numbers

Usually stored using two's complement

- Take the magnitude of the number
- Invert all of the bits
- Add 1

Computing $-x$ from x (regardless of sign)

- Invert all of the bits
- Add 1

representation of -5 in 8 bits

magnitude: 0000_0101

invert bits: 1111_1010

Add 1: 1111_1011

$-(-5)$ in 8 bits

-5: 1111_1011

Negative numbers

Usually stored using two's complement

- Take the magnitude of the number
- Invert all of the bits
- Add 1

Computing $-x$ from x (regardless of sign)

- Invert all of the bits
- Add 1

representation of -5 in 8 bits

magnitude: 0000_0101

invert bits: 1111_1010

Add 1: 1111_1011

$-(-5)$ in 8 bits

-5: 1111_1011

invert bits: 0000_0100

Negative numbers

Usually stored using two's complement

- Take the magnitude of the number
- Invert all of the bits
- Add 1

Computing $-x$ from x (regardless of sign)

- Invert all of the bits
- Add 1

representation of -5 in 8 bits

magnitude: 0000_0101

invert bits: 1111_1010

Add 1: 1111_1011

$-(-5)$ in 8 bits

-5: 1111_1011

invert bits: 0000_0100

Add 1: 0000_0101

Negative numbers

Usually stored using two's complement

- Take the magnitude of the number
- Invert all of the bits
- Add 1

Computing $-x$ from x (regardless of sign)

- Invert all of the bits
- Add 1

representation of -5 in 8 bits

magnitude: 0000_0101

invert bits: 1111_1010

Add 1: 1111_1011

$-(-5)$ in 8 bits

-5: 1111_1011

invert bits: 0000_0100

Add 1: 0000_0101

0: 0000_0000

Negative numbers

Usually stored using two's complement

- Take the magnitude of the number
- Invert all of the bits
- Add 1

Computing $-x$ from x (regardless of sign)

- Invert all of the bits
- Add 1

representation of -5 in 8 bits

```
magnitude:    0000_0101
invert bits:   1111_1010
Add 1:         1111_1011
```

$-(-5)$ in 8 bits

```
-5:           1111_1011
invert bits:   0000_0100
Add 1:         0000_0101
```

```
0:            0000_0000
invert bits:   1111_1111
```

Negative numbers

Usually stored using two's complement

- Take the magnitude of the number
- Invert all of the bits
- Add 1

Computing $-x$ from x (regardless of sign)

- Invert all of the bits
- Add 1

representation of -5 in 8 bits

```
magnitude:    0000_0101
invert bits:   1111_1010
Add 1:         1111_1011
```

$-(-5)$ in 8 bits

```
-5:           1111_1011
invert bits:   0000_0100
Add 1:         0000_0101
```

```
0:            0000_0000
invert bits:   1111_1111
Add 1:         1_0000_0000
```

Negative numbers

Usually stored using two's complement

- Take the magnitude of the number
- Invert all of the bits
- Add 1

Computing $-x$ from x (regardless of sign)

- Invert all of the bits
- Add 1

Most significant bit indicates the sign

- 1 indicates a negative number

representation of -5 in 8 bits

```
magnitude:    0000_0101
invert bits:   1111_1010
Add 1:         1111_1011
```

$-(-5)$ in 8 bits

```
-5:           1111_1011
invert bits:   0000_0100
Add 1:         0000_0101
```

```
0:            0000_0000
invert bits:   1111_1111
Add 1:         1_0000_0000
```

Signed numbers in two's complement

10000000	00000000	00000000	00000000	=	-2^{31}
10000000	00000000	00000000	00000001	=	$-2^{31}+1$
...					
11111111	11111111	11111111	11111110	=	-2
11111111	11111111	11111111	11111111	=	-1
00000000	00000000	00000000	00000000	=	0
00000000	00000000	00000000	00000001	=	1
00000000	00000000	00000000	00000010	=	2
00000000	00000000	00000000	00000011	=	3
...					
01111111	11111111	11111111	11111110	=	$2^{31}-2$
01111111	11111111	11111111	11111111	=	$2^{31}-1$

Not the only choice

Not the only choice

Sign and magnitude

- ▶ Most significant bit represents the sign, remaining bits are the magnitude
- ▶ Range $-(2^{n-1} - 1)$ to $2^{n-1} - 1$
- ▶ Two different bit patterns for zero: 0 and 0x80000000 (assuming 32-bits)

Not the only choice

Sign and magnitude

- ▶ Most significant bit represents the sign, remaining bits are the magnitude
- ▶ Range $-(2^{n-1} - 1)$ to $2^{n-1} - 1$
- ▶ Two different bit patterns for zero: 0 and 0x80000000 (assuming 32-bits)

Ones' complement

- ▶ Negative numbers are the bitwise inverse of positive numbers ($-x = \sim x$)
- ▶ Range $-(2^{n-1} - 1)$ to $2^{n-1} - 1$
- ▶ Two different bit patterns for zero: 0 and 0xFFFFFFFF (assuming 32-bits)

Not the only choice

Sign and magnitude

- ▶ Most significant bit represents the sign, remaining bits are the magnitude
- ▶ Range $-(2^{n-1} - 1)$ to $2^{n-1} - 1$
- ▶ Two different bit patterns for zero: 0 and 0x80000000 (assuming 32-bits)

Ones' complement

- ▶ Negative numbers are the bitwise inverse of positive numbers ($-x = \sim x$)
- ▶ Range $-(2^{n-1} - 1)$ to $2^{n-1} - 1$
- ▶ Two different bit patterns for zero: 0 and 0xFFFFFFFF (assuming 32-bits)

Two's complement

- ▶ Negative numbers are ones' complement plus one ($-x = \sim x + 1$)
- ▶ Range -2^{n-1} to $2^{n-1} - 1$
- ▶ Only one zero

In-class exercise

<https://checkoway.net/teaching/cs241/2020-spring/exercises/Lecture-13.html>

Grab a laptop and a partner and try to get as much of that done as you can!