Negative Numbers

CS143: lecture 6

Byron Zhong, July 20

What is this?

`1110 1001`

- 233
- Ascii é
- Yes, Yes, Yes, No, Yes, No, No Yes
- Sound wave sample with some magnitude
- A pixel intensity of 91.31%
- ...
- It's one of the 256 choices, whatever that is.

Negative Numbers

- If we have 32 bits, we have 2^{32} choices
  - unsigned: 2^{32} choices of non-negative numbers -- 0 through 2^{32} - 1
  - int: 2^{32} choices of both negative and positive numbers *about* -2^{31} through 2^{31}

Negative Numbers

Attempt 1

- The first bit is the sign, 0 -> positive, 1 -> negative

```
 3 = 0000 0000 0000 0000 0000 0000 0000 0011
-3 = 1000 0000 0000 0000 0000 0000 0000 0011
```

- Range?
  - $[-(2^{31} - 1), 2^{31} - 1]$

Negative Numbers

Attempt 1

- The first bit is the sign, 0 -> positive, 1 -> negative

```
 3 = 0000 0000 0000 0000 0000 0000 0000 0011
-3 = 1000 0000 0000 0000 0000 0000 0000 0011
```

- What's wrong with this?

```
0000 0000 0000 0000 0000 0000 0000 0000
1000 0000 0000 0000 0000 0000 0000 0000
```

Negative Numbers

Attempt 1

- The first bit is the sign, 0 -> positive, 1 -> negative

```
 3 = 0000 0000 0000 0000 0000 0000 0000 0011
-3 = 1000 0000 0000 0000 0000 0000 0000 0011
```

- What's wrong with this?

```
0000 0000 0000 0000 0000 0000 0000 0000
1000 0000 0000 0000 0000 0000 0000 0000
```

- We have +0 and -0 !?
  - Waste a combination, equality check is complicated +0 == -0

Negative Numbers

Attempt 1

- The first bit is the sign, 0 -> positive, 1 -> negative

```
 3 = 0000 0000 0000 0000 0000 0000 0000 0011
-3 = 1000 0000 0000 0000 0000 0000 0000 0011
```

- What's wrong with this?

```
   0000 0000 0000 0000 0000 0000 0000 0011
 + 1000 0000 0000 0000 0000 0000 0000 0011
```

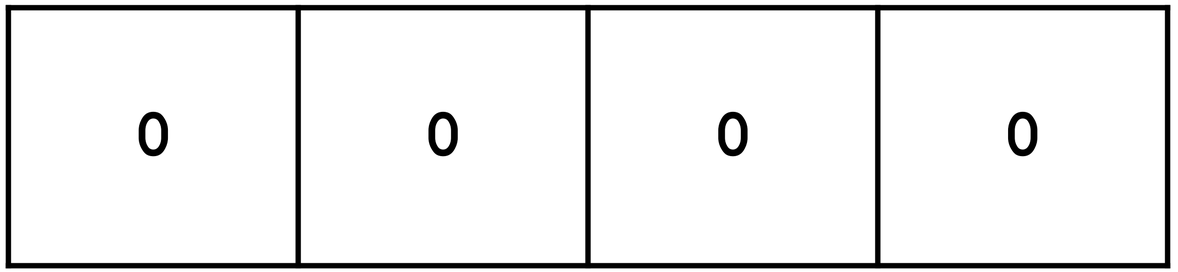
- 5 + (-5) is not 0 :(((
  - Need special circuit for adding signed numbers and unsigned numbers

Negative Numbers

Better Idea: Two's Complement

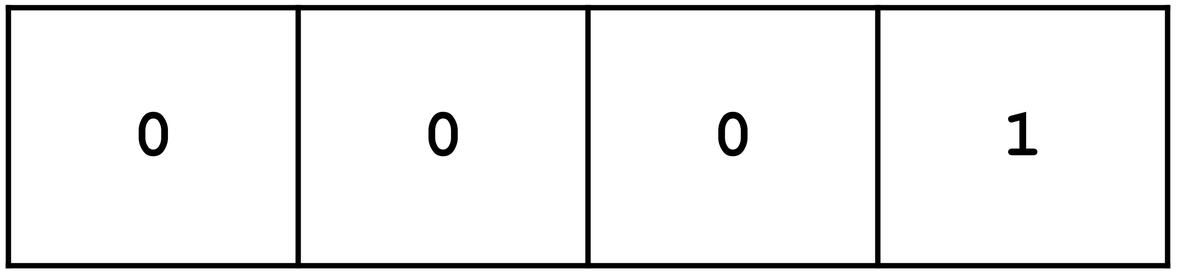
Negative Numbers

Better Idea: Two's Complement



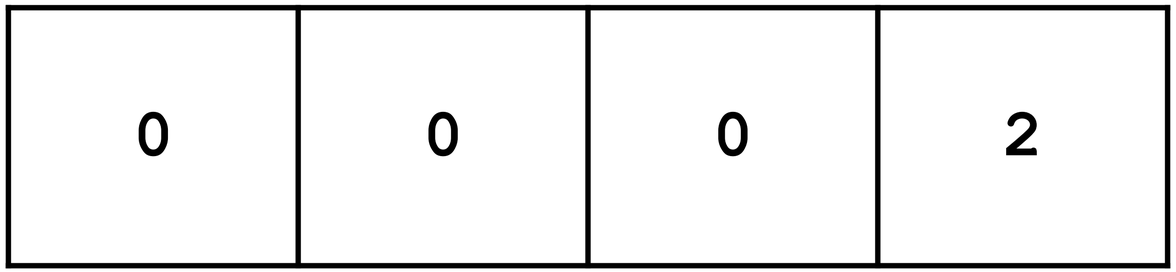
Negative Numbers

Better Idea: Two's Complement



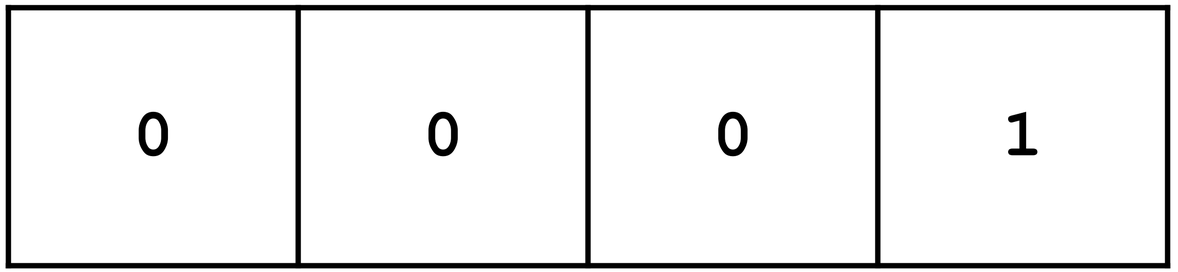
Negative Numbers

Better Idea: Two's Complement



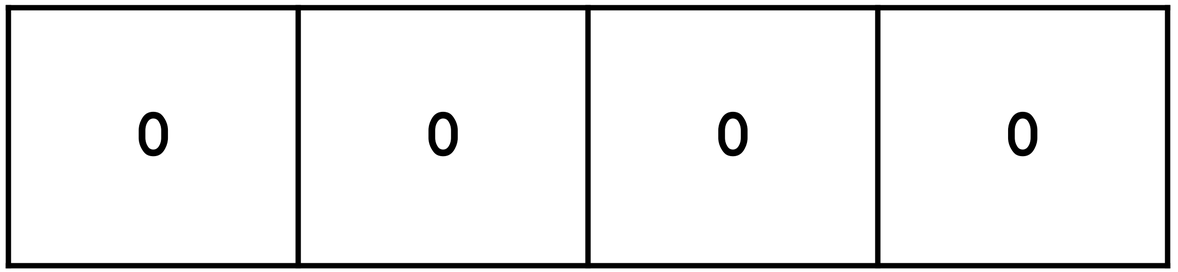
Negative Numbers

Better Idea: Two's Complement



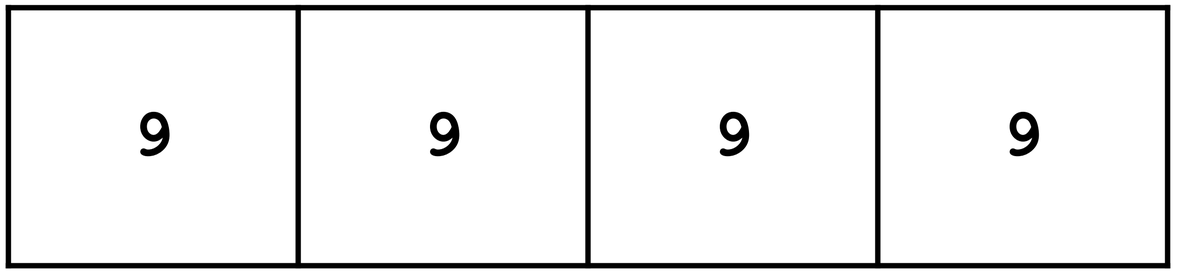
Negative Numbers

Better Idea: Two's Complement



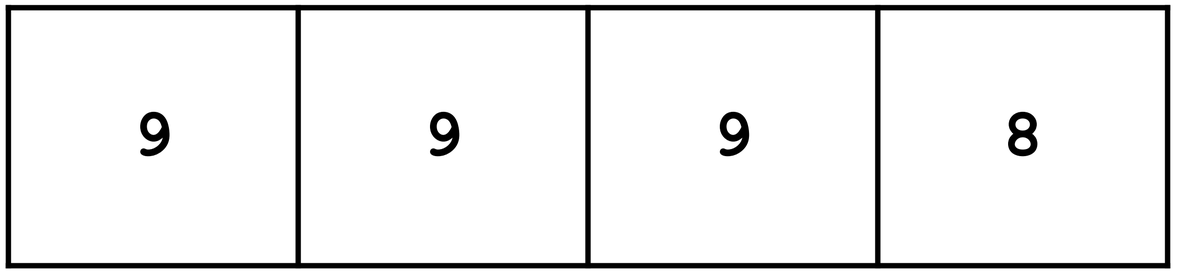
Negative Numbers

Better Idea: Two's Complement



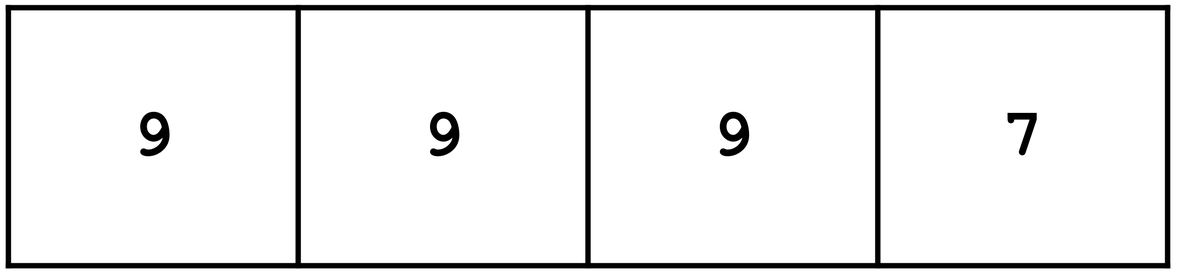
Negative Numbers

Better Idea: Two's Complement



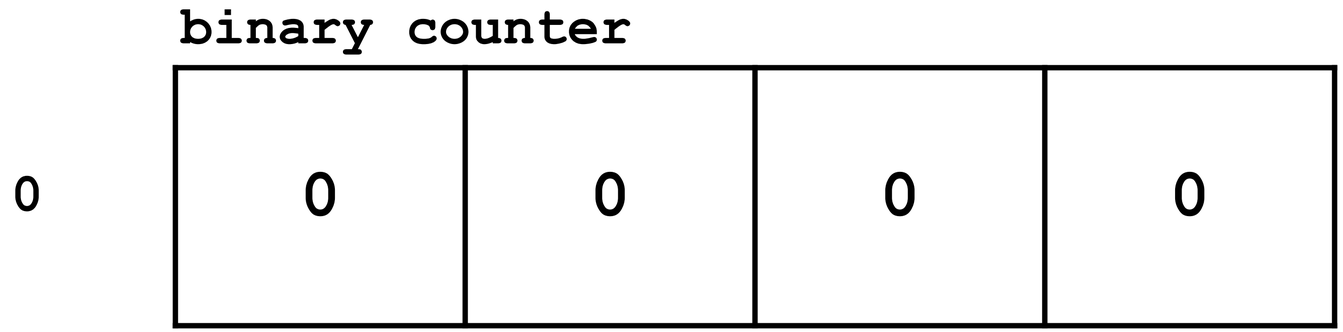
Negative Numbers

Better Idea: Two's Complement



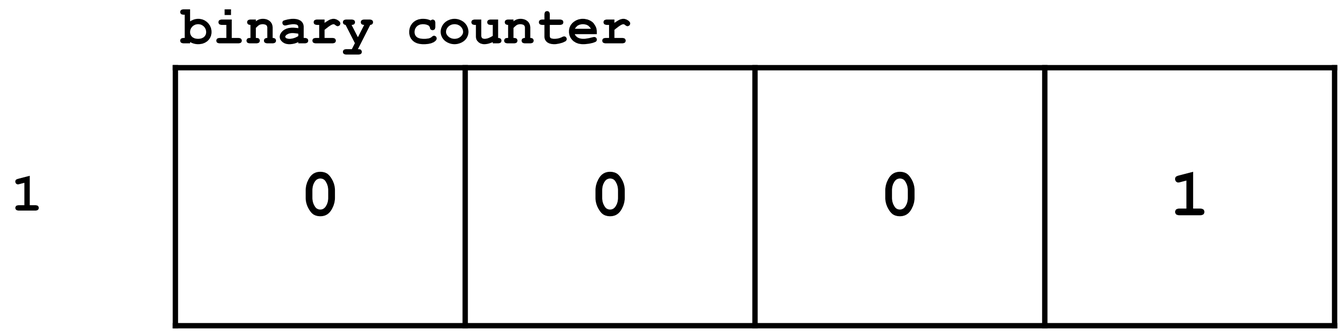
Negative Numbers

Better Idea: Two's Complement



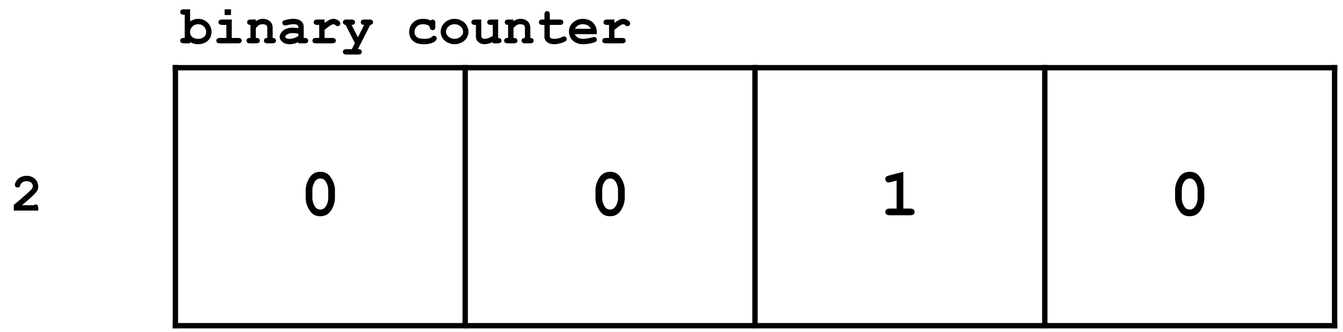
Negative Numbers

Better Idea: Two's Complement



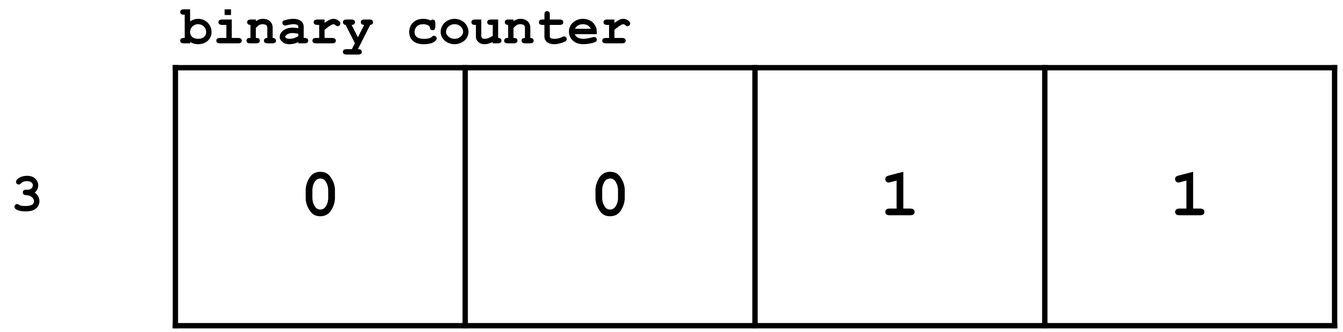
Negative Numbers

Better Idea: Two's Complement



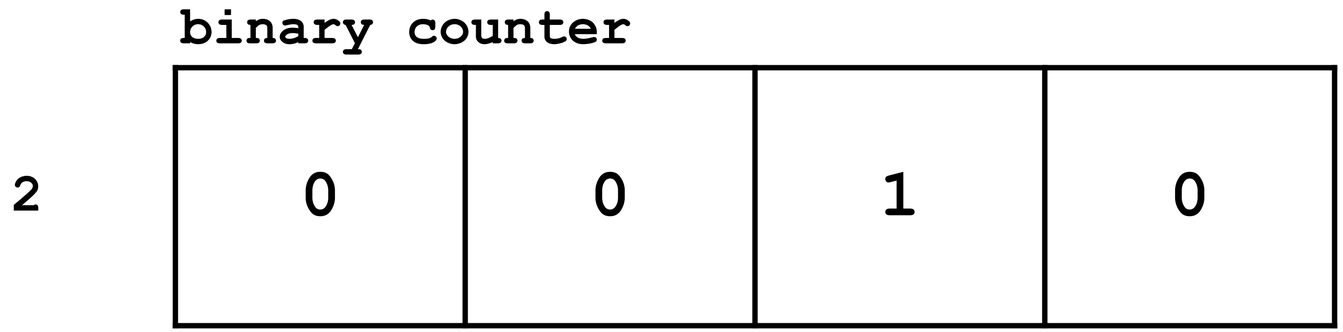
Negative Numbers

Better Idea: Two's Complement



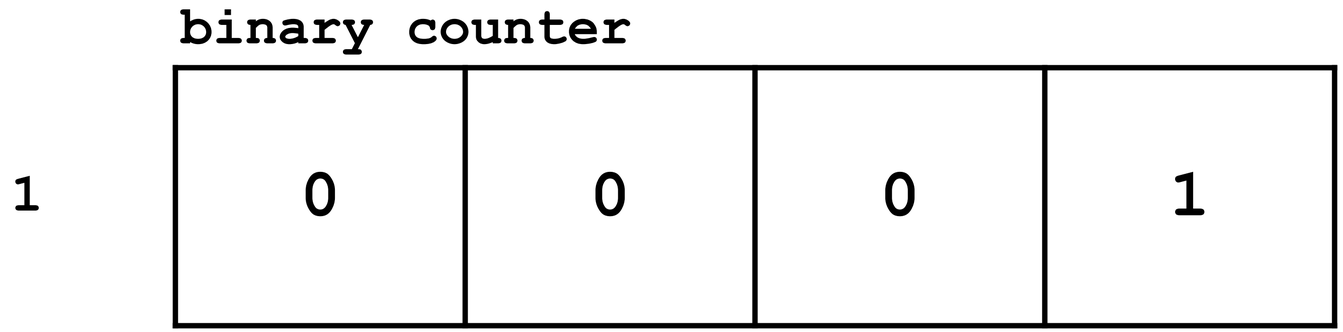
Negative Numbers

Better Idea: Two's Complement



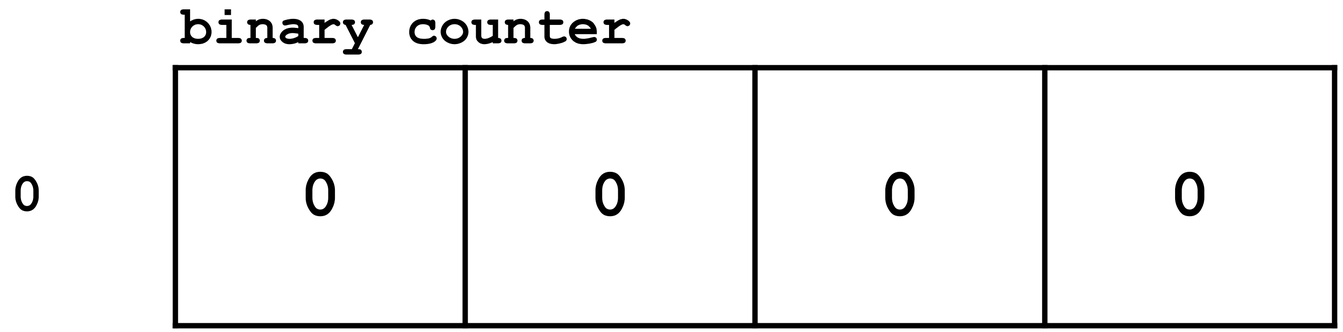
Negative Numbers

Better Idea: Two's Complement



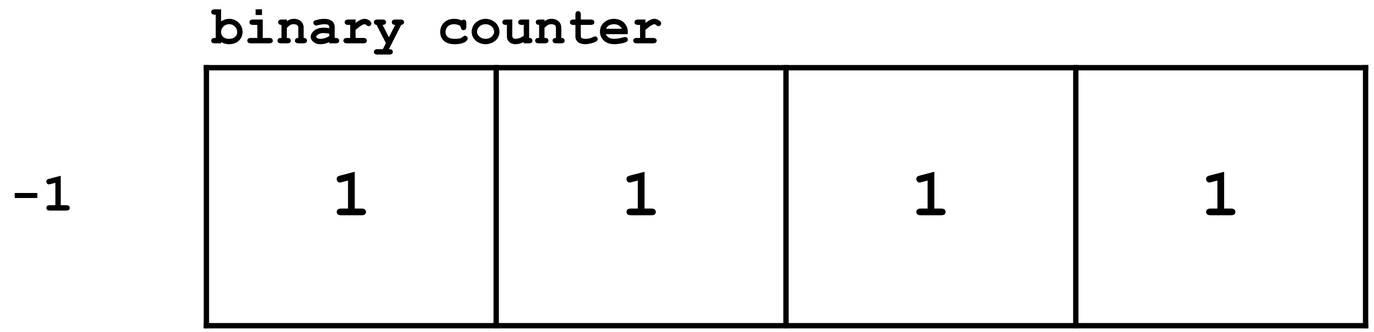
Negative Numbers

Better Idea: Two's Complement



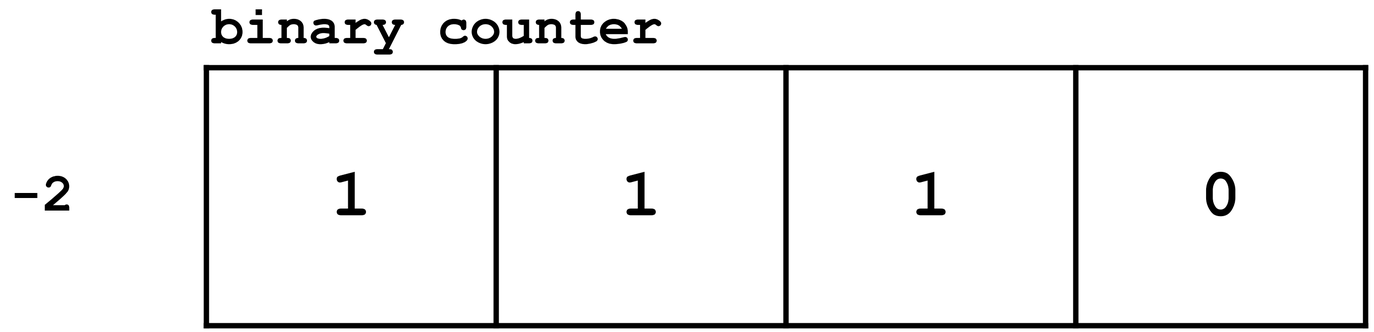
Negative Numbers

Better Idea: Two's Complement



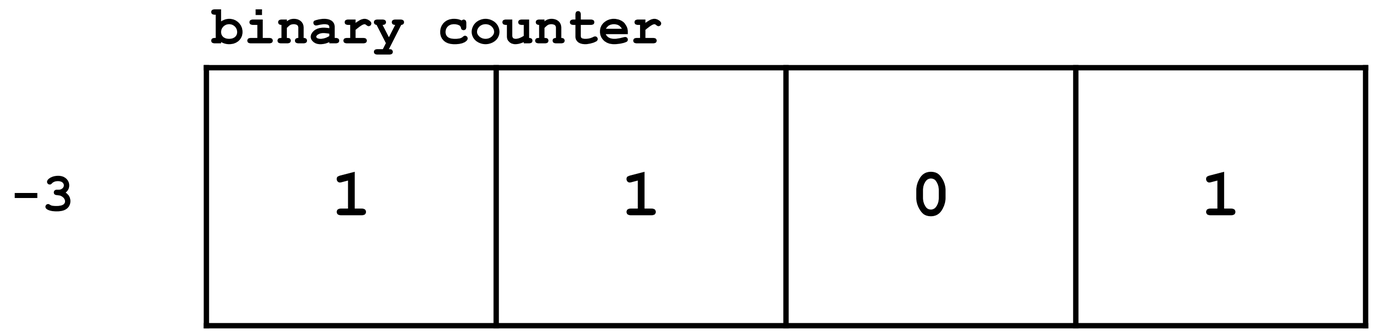
Negative Numbers

Better Idea: Two's Complement



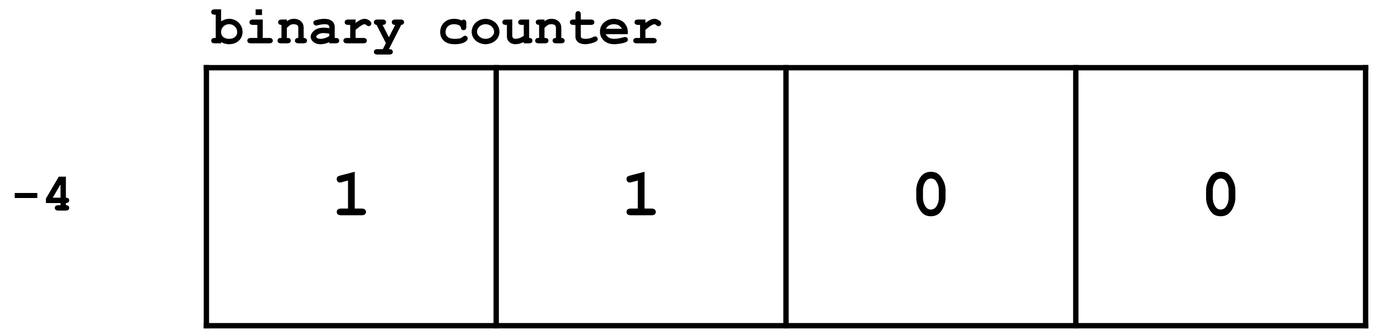
Negative Numbers

Better Idea: Two's Complement



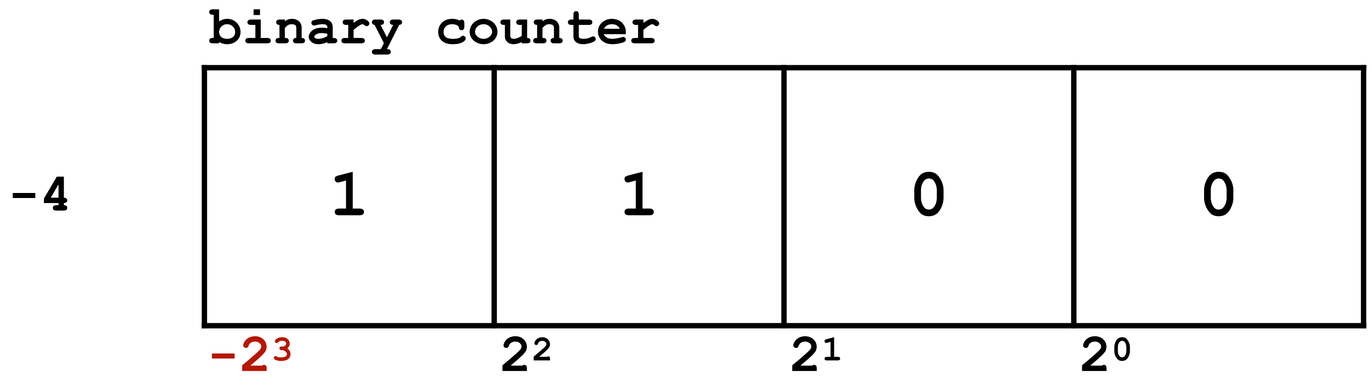
Negative Numbers

Better Idea: Two's Complement



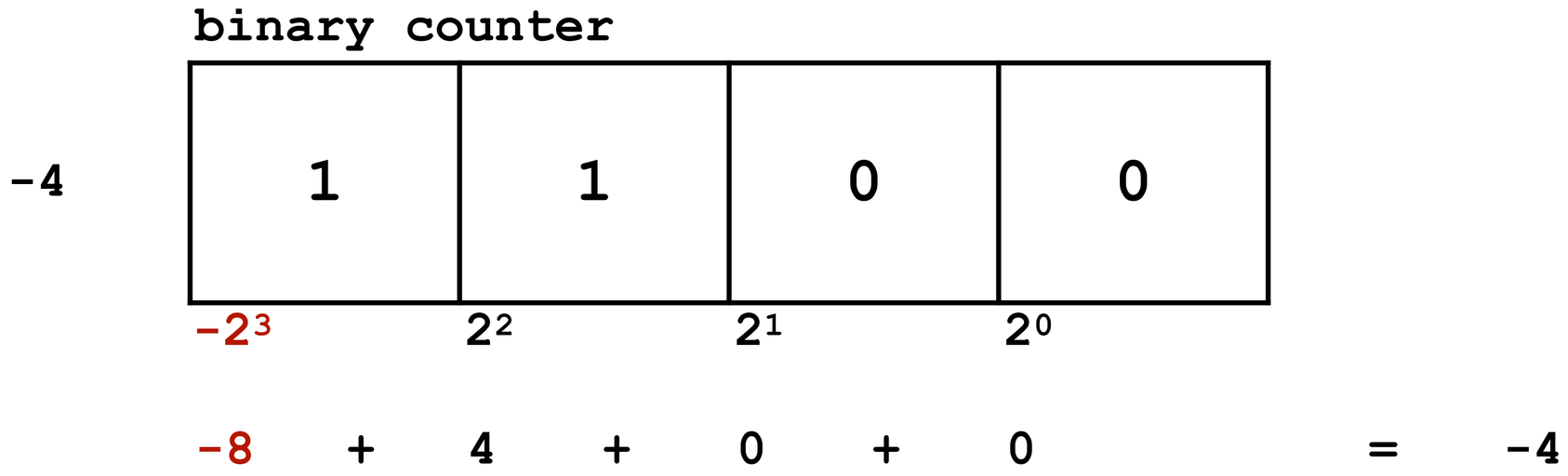
Negative Numbers

Two's complement: two ways



Negative Numbers

Two's complement: two ways



- The most significant bit is negative.

Negative Numbers

Two's complement: two ways

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

```
1110 1001
```

- Unsigned: 233
- What does this represent if it is a signed integer?
- -128 + 64 + 32 + 8 + 1 = -23

Negative Numbers

Two's complement: two ways

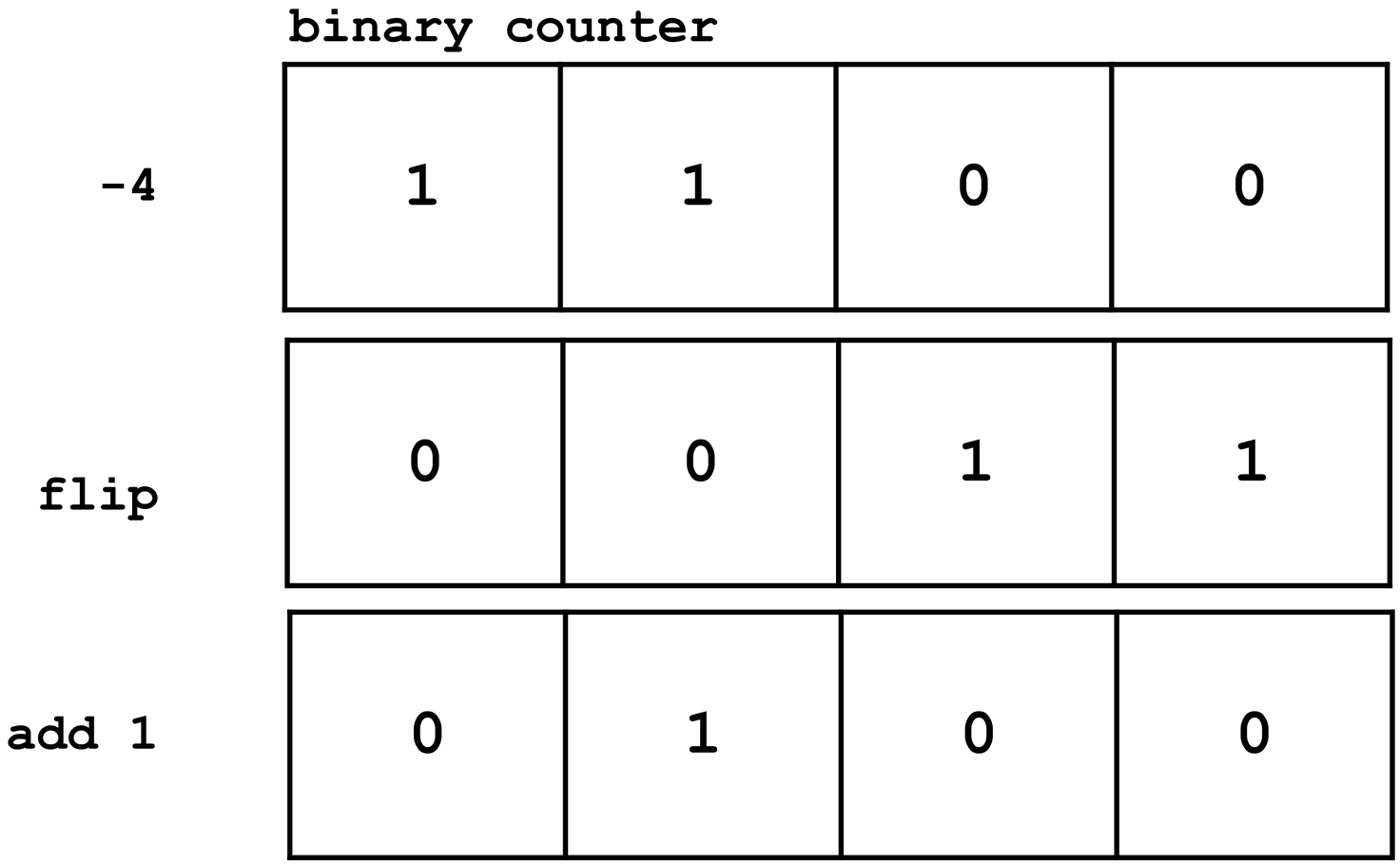
n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

```
1110 1001
```

- How do hardwares negate the sign?
  - Flip all the bits and add one
- Flip all the bits: 0001 0110
- add one: 0001 0111
- : 16 + 4 + 2 + 1 = 23

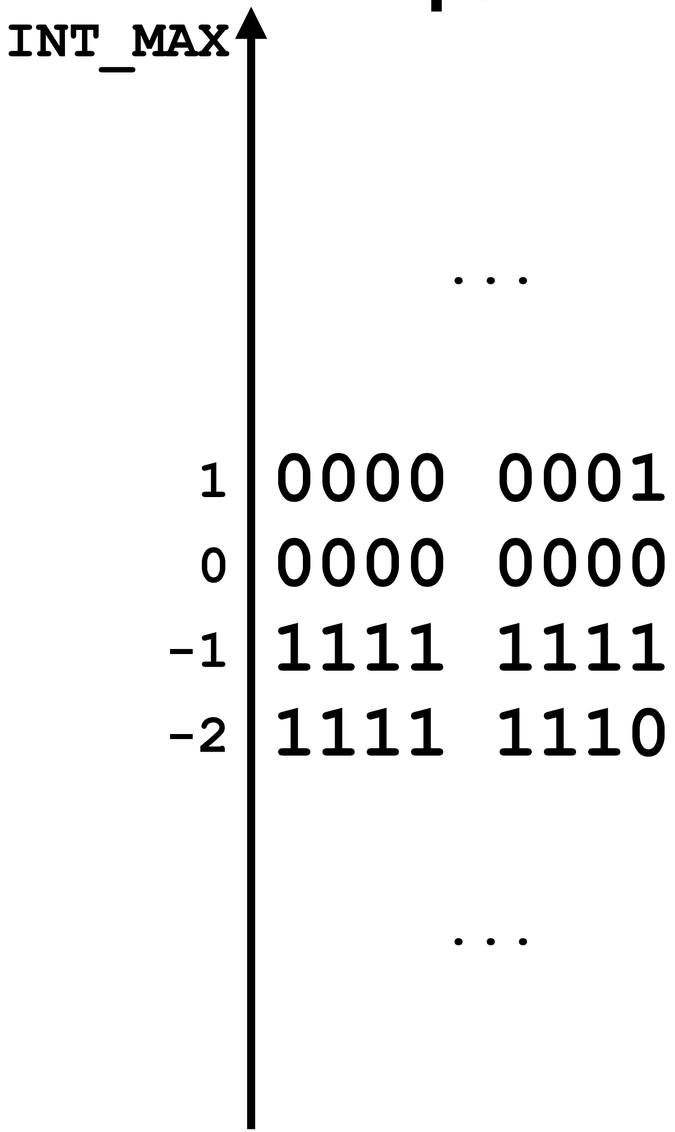
Negative Numbers

Two's complement: two ways



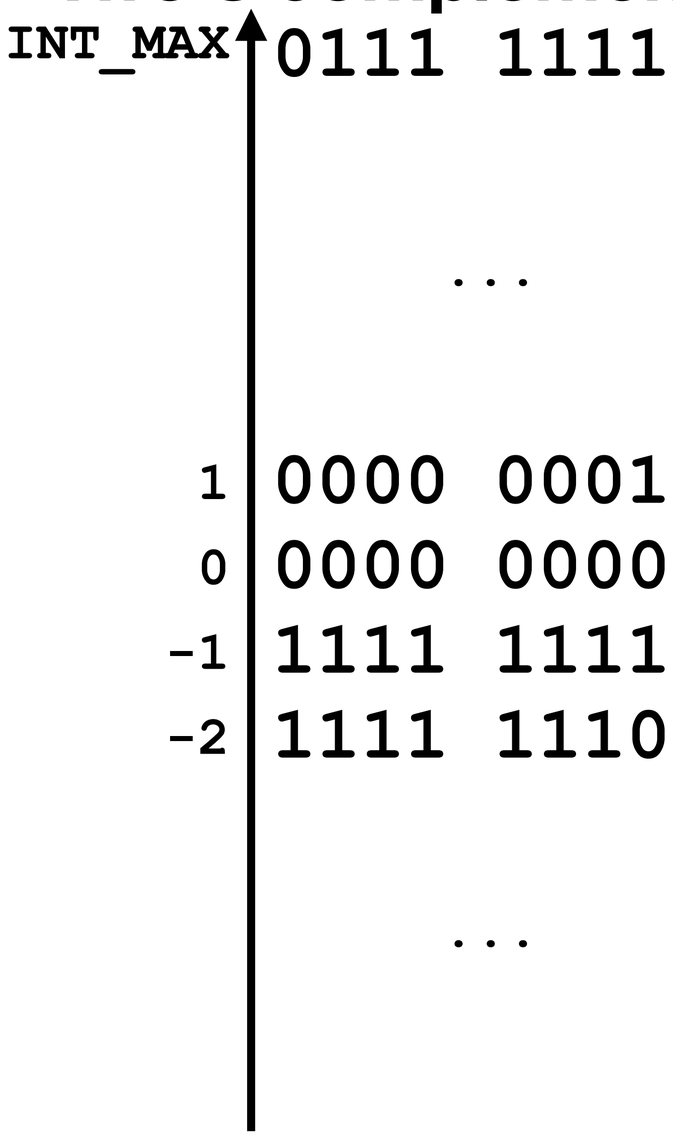
Negative Numbers

Two's complement: range



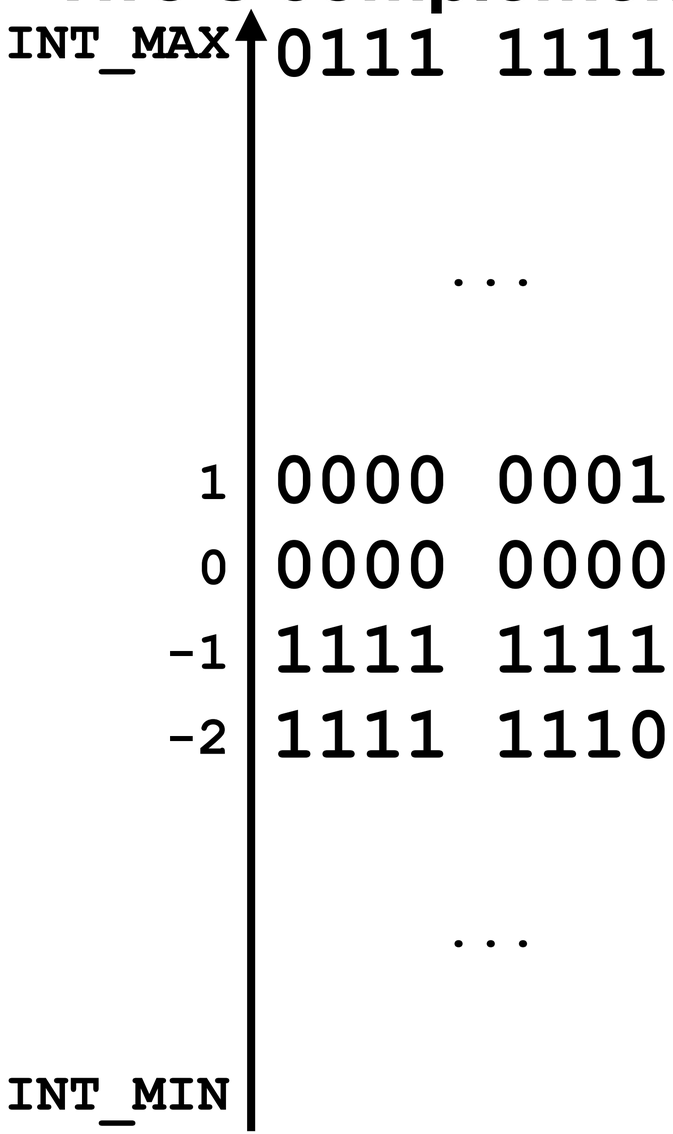
Negative Numbers

Two's complement: range



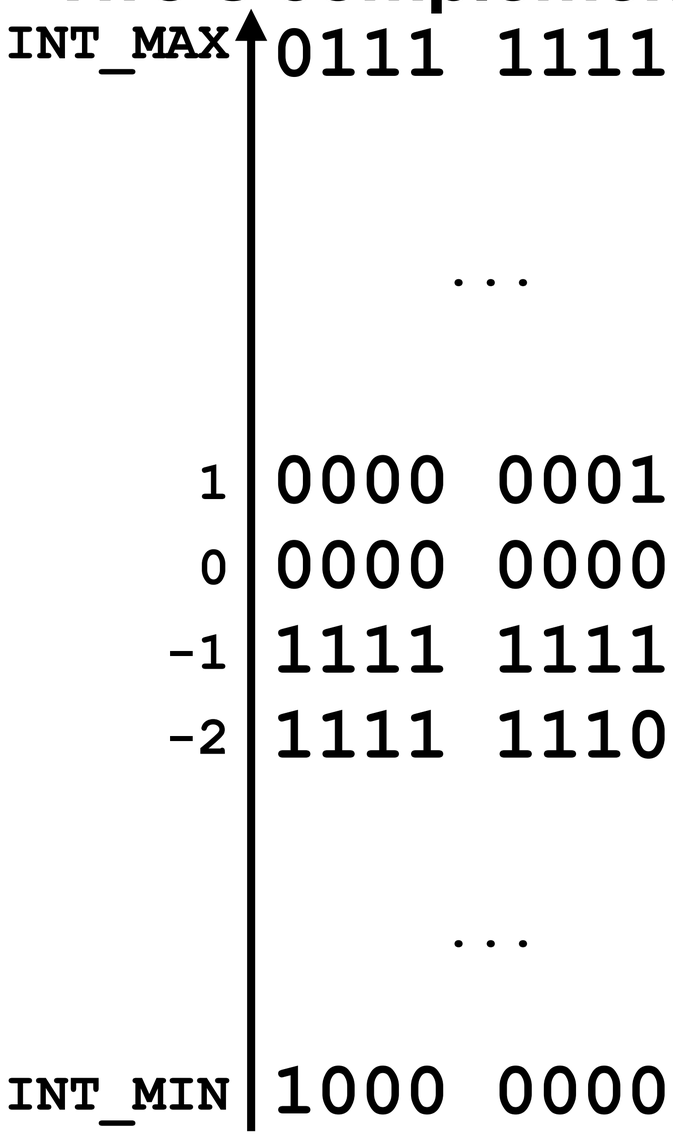
Negative Numbers

Two's complement: range



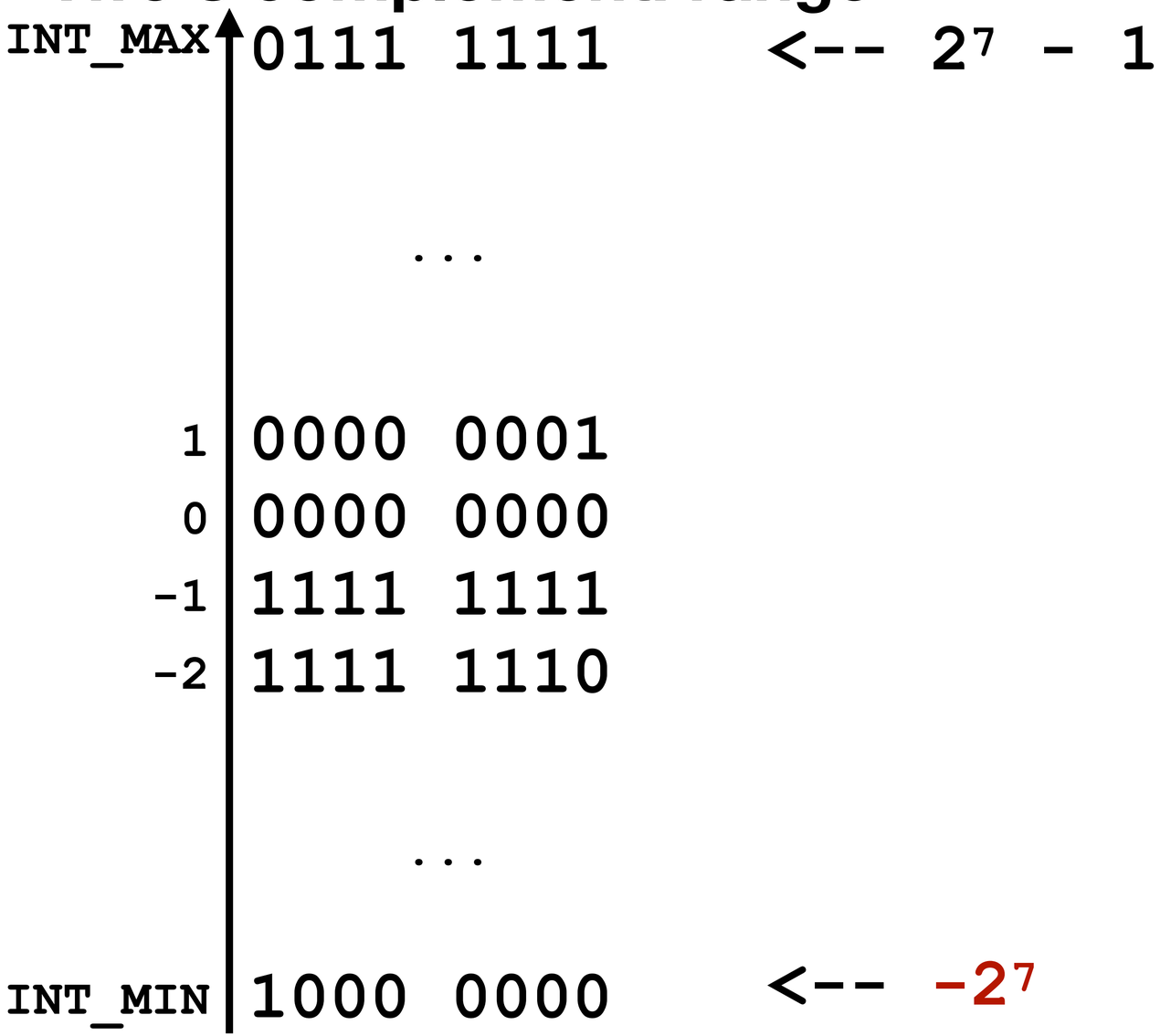
Negative Numbers

Two's complement: range



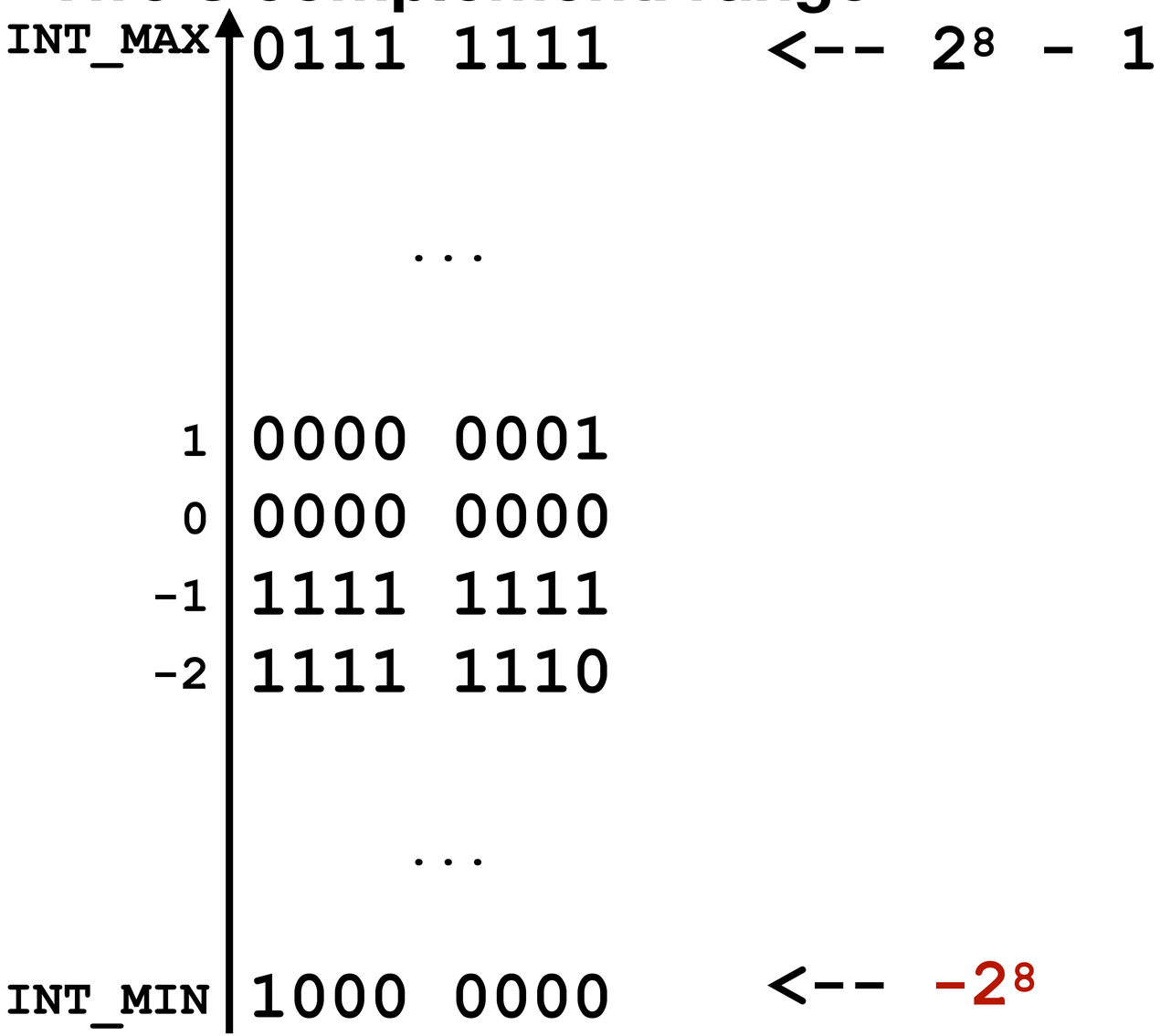
Negative Numbers

Two's complement: range

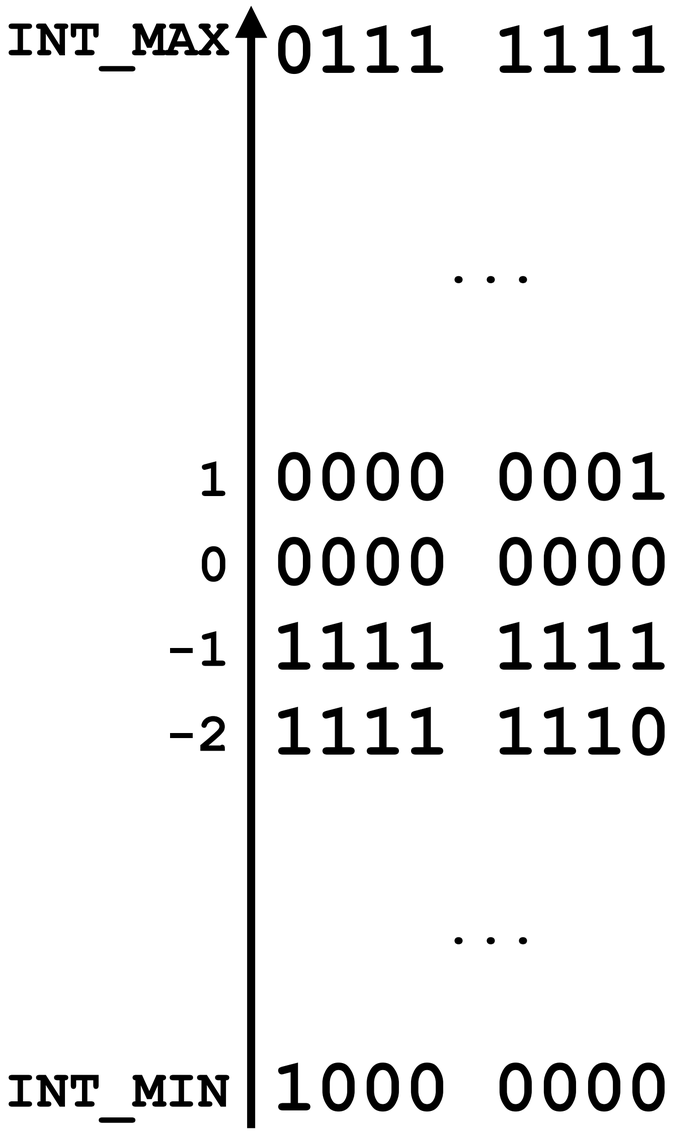


Negative Numbers

Two's complement: range

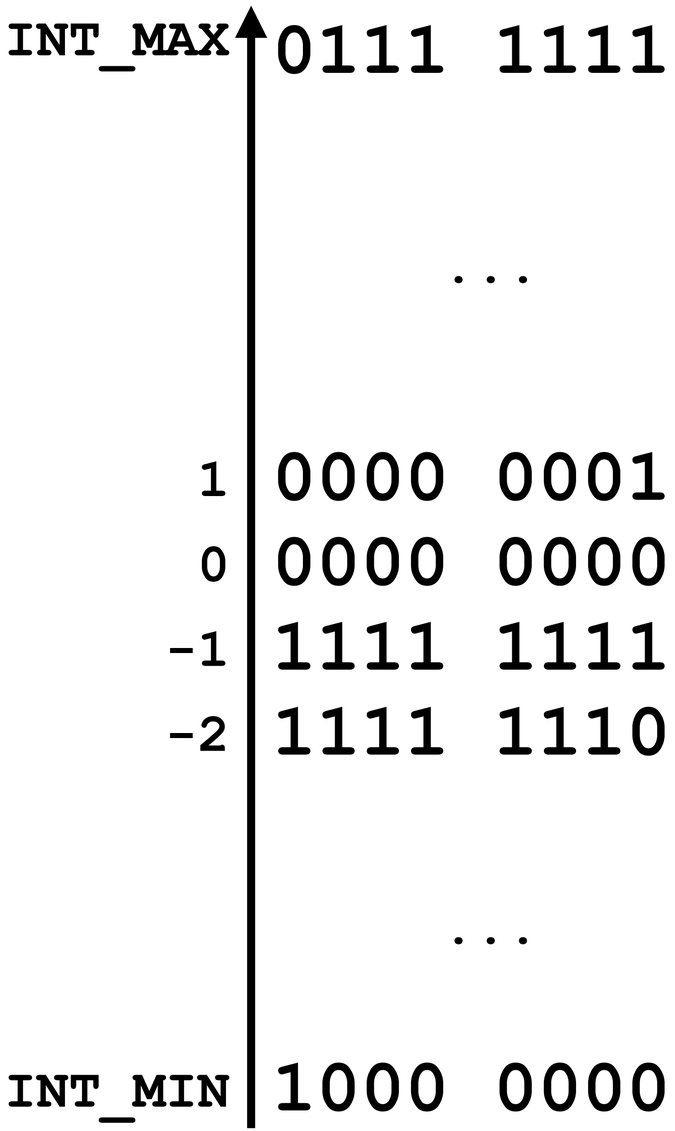


Signed Arithmetics



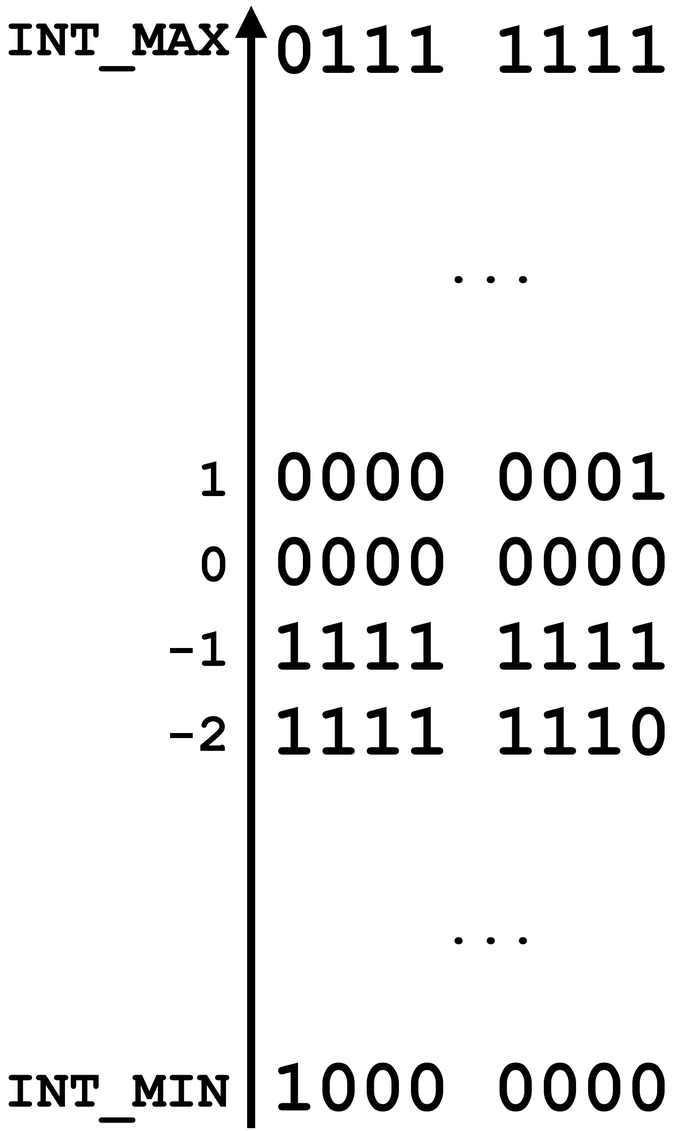

```
  1111 1110
+ 0000 0011
-----------
```

Signed Arithmetics




```
  1111 1110          -2
+ 0000 0011           3
-------------
```

Signed Arithmetics



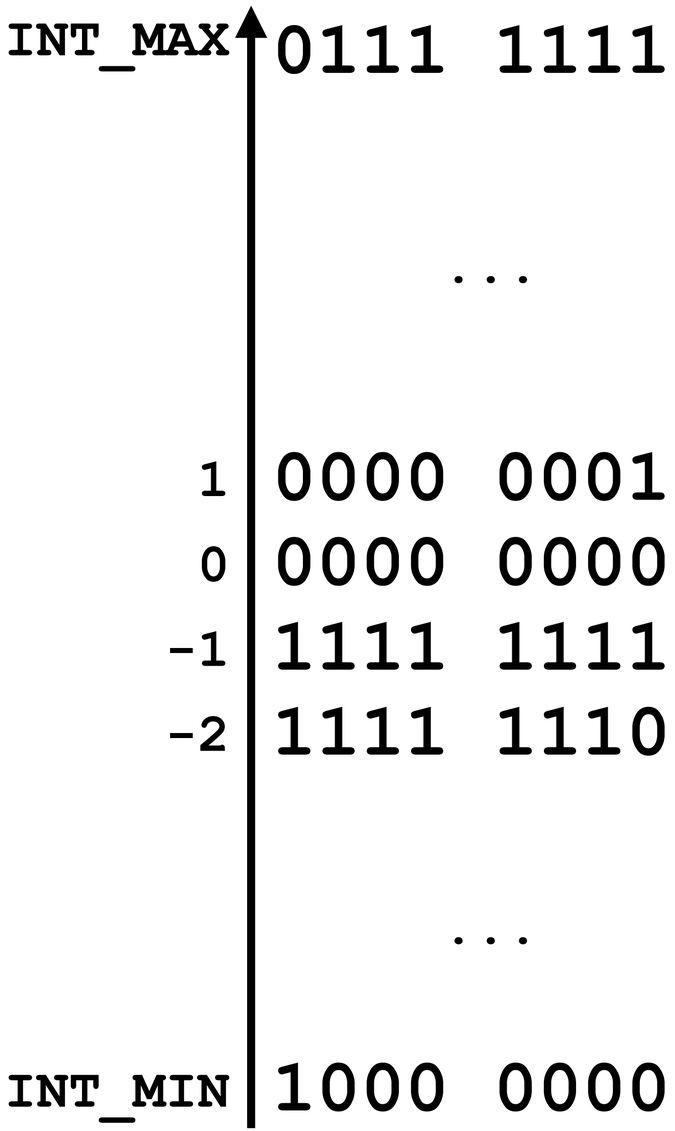

```
  1111 1110          -2
+ 0000 0011           3
-------------------------
             1
```

Signed Arithmetics

```
INT_MAX  0111 1111

            ...

      1  0000 0001
      0  0000 0000
     -1  1111 1111
     -2  1111 1110

            ...

INT_MIN  1000 0000
```

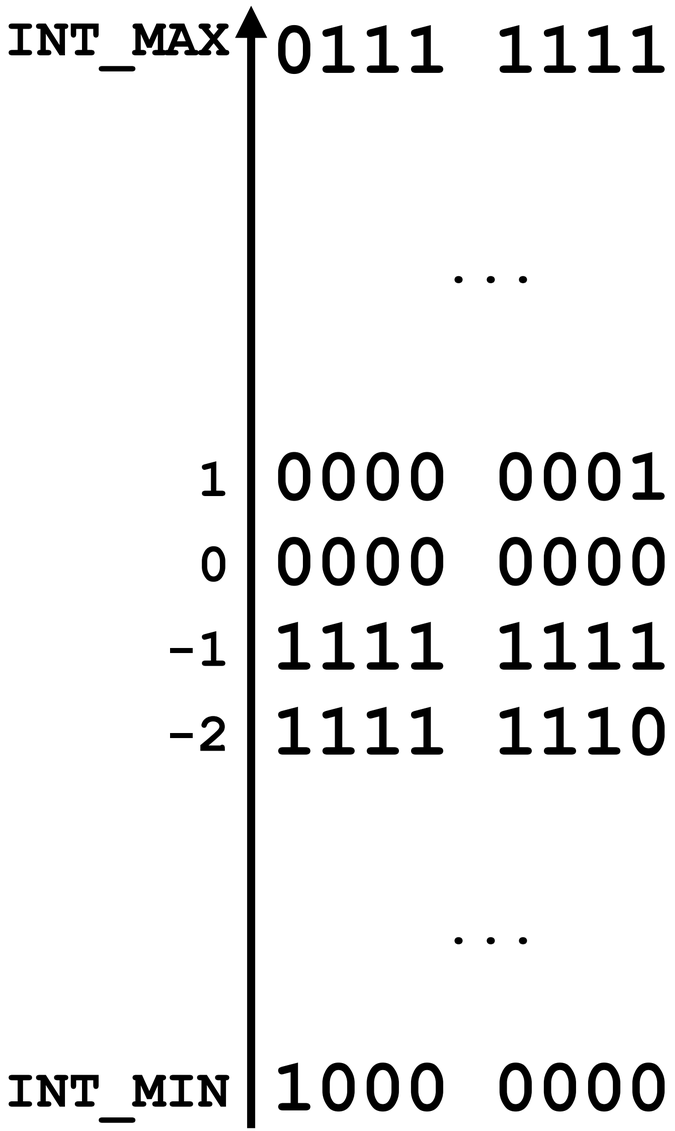
```
    1111 1110        -2
  + 0000 0011         3
          1
  -----------
           01
```

Signed Arithmetics



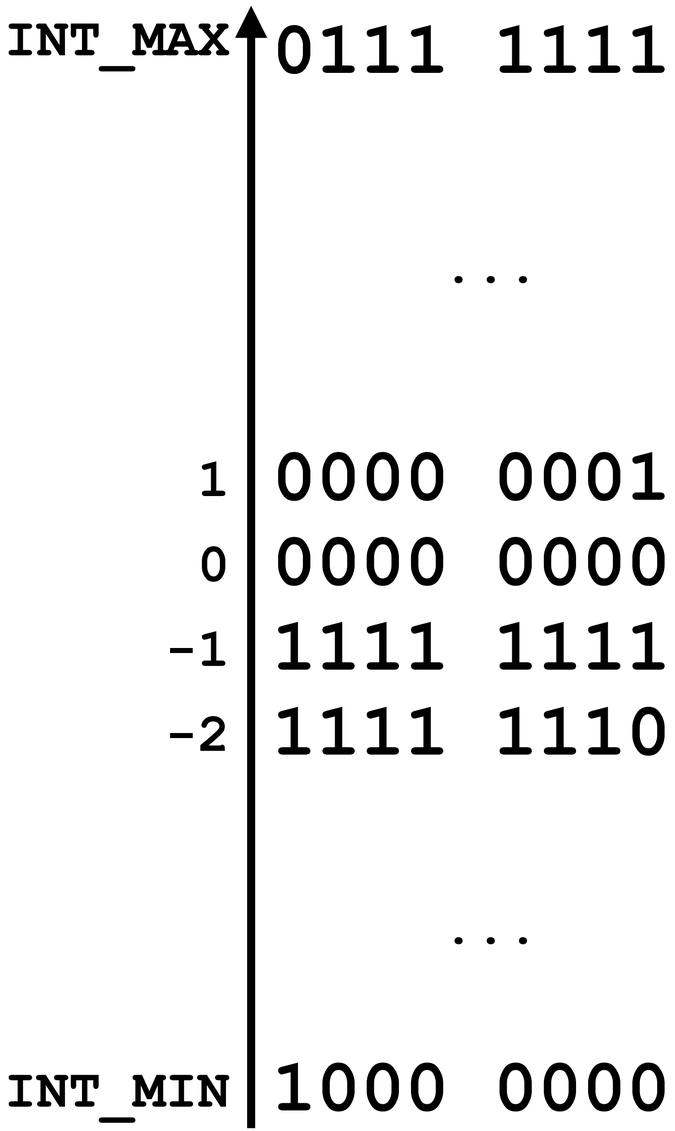

```
  1111 1110          -2
+ 0000 0011           3
-----------------------
         001
```

Signed Arithmetics



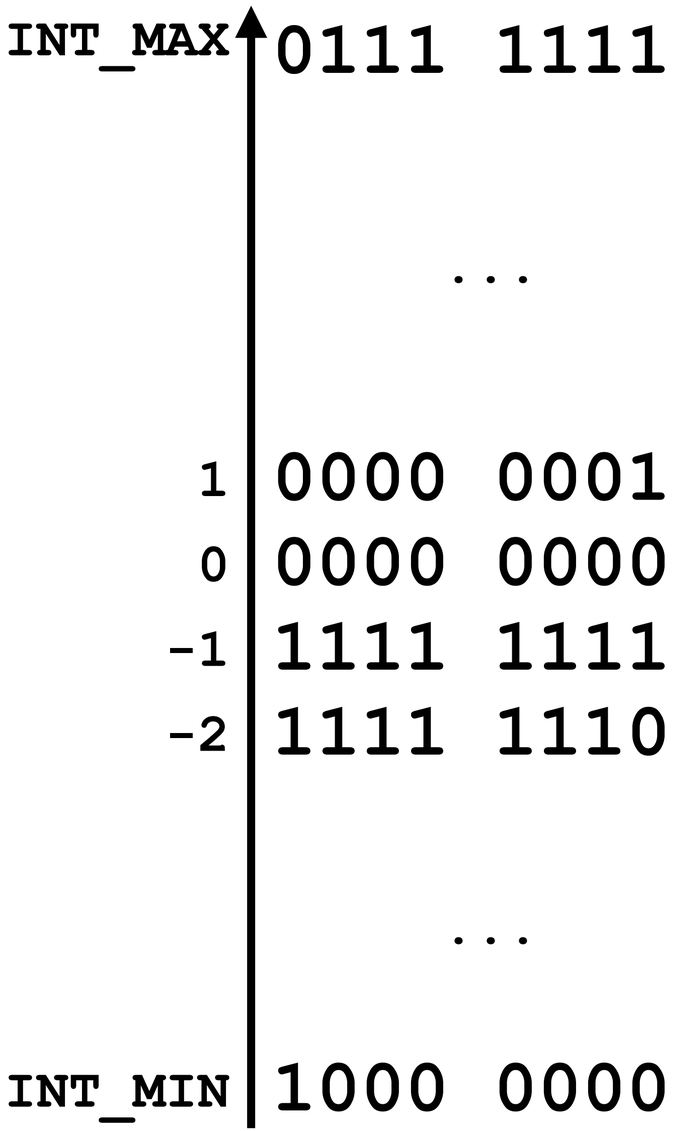

```
  1111 1110          -2
+ 0000 0011           3
  ---------------------
  1    1 1
  ---------------------
         0001
```

Signed Arithmetics



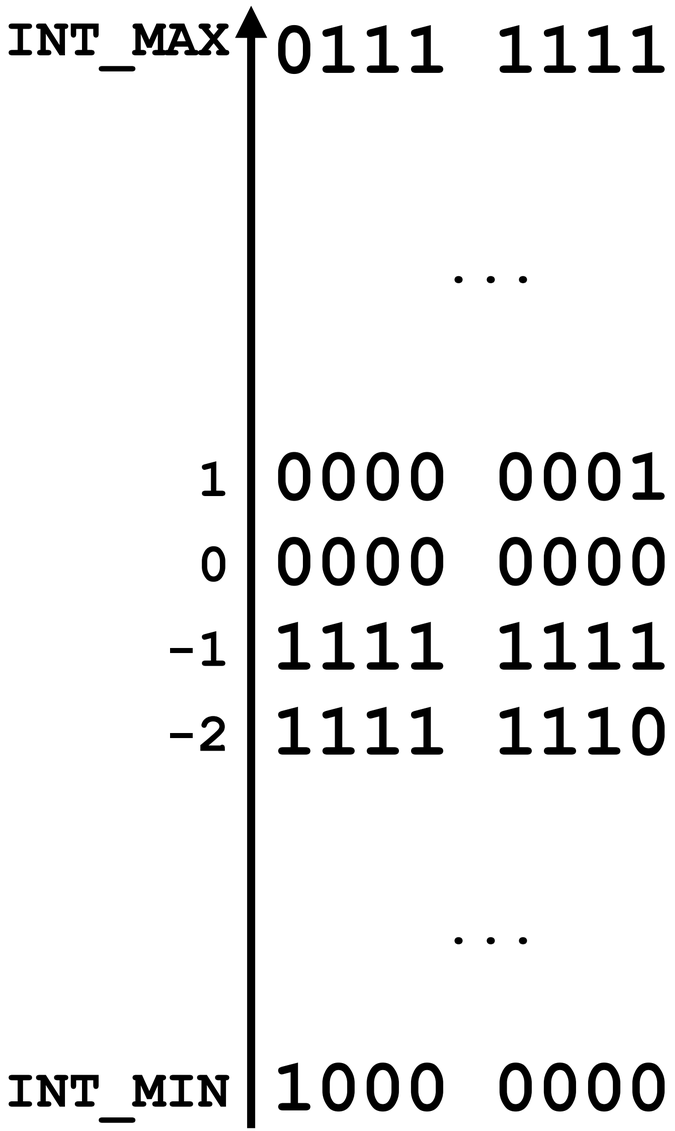

```
  1111 1110          -2
+ 0000 0011           3
-----------------------
     0 0001
```

Signed Arithmetics



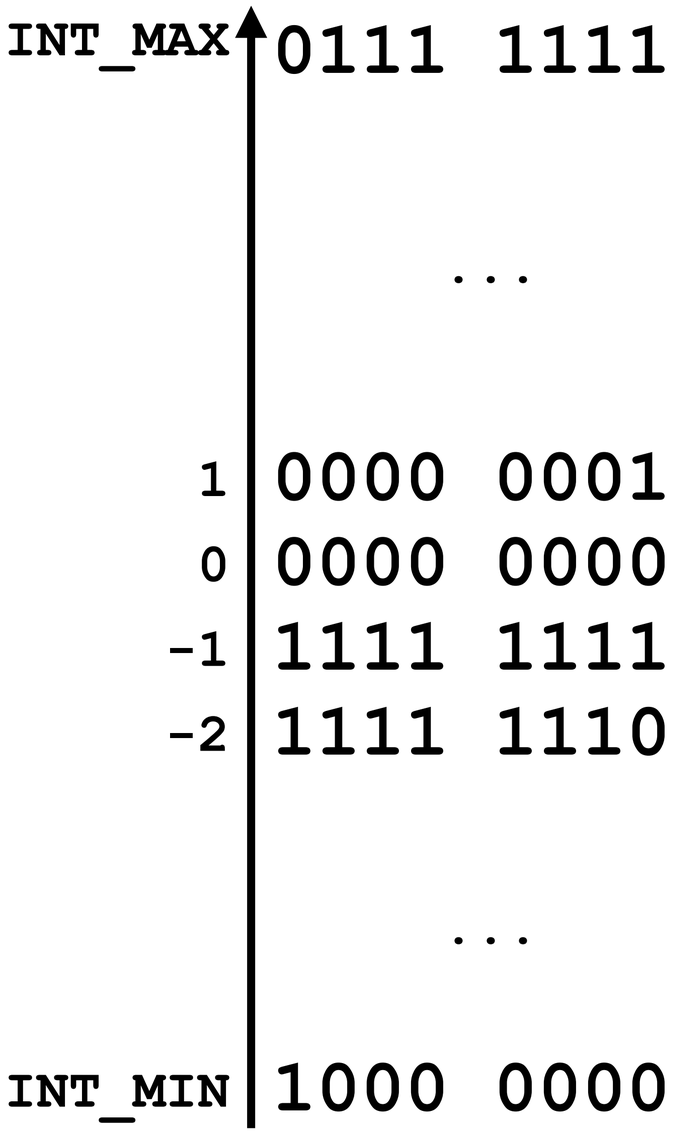

```
  1111 1110          -2
+ 0000 0011           3
-------------
  1 1 1  1 1
     00 0001
```

Signed Arithmetics




```
  1111 1110          -2
+ 0000 0011           3
  1111 1111
-------------
  1 1 1 1  1 1
    000 0001
```

Signed Arithmetics




```
  1111 1110          -2
+ 0000 0011           3
  1 1 1 1 1   1 1
-------------------------
  0000 0001
```

- The hardware does not need to know/ care whether the arithmetics is signed or unsigned
- **`INT_MAX + 1 == INT_MIN`**

Signed Arithmetics

Unsigned >>

```
  1111 1111 1111 1111 1111 1111 1111  1100          -4
>>                                       1
```

Signed Arithmetics

Unsigned >>

```
  1111 1111 1111 1111 1111 1111 1111  1100          -4
>>                                       1
-------------------------------------------------------
  0111 1111 1111 1111 1111 1111 1111  1110   2147483646
```

- If we just add 0 in right shift, shifting a negative number results in a positive number.
- Ideally, we would like to keep the sign / shifting is dividing by 2.
- We repeat the most significant bit when doing a *signed* right shift.

Signed Arithmetics

Signed >>

```
    1111 1111 1111 1111 1111 1111 1111 1100          -4
>>                                        1
-----------------------------------------------------
    1111 1111 1111 1111 1111 1111 1111 1110
```

Signed Arithmetics

Signed >>

```
    1111 1111 1111 1111 1111 1111 1111 1100        -4
>>                                      1
-------------------------------------------------------
    1111 1111 1111 1111 1111 1111 1111 1110        -2

    0000 0000 0000 0000 0000 0000 0000 0100         4
>>                                      1
-------------------------------------------------------
```

Signed Arithmetics

Signed >>

```
    1111 1111 1111 1111 1111 1111 1111 1100          -4
>>                                      1
-------------------------------------------------------
    1111 1111 1111 1111 1111 1111 1111 1110          -2

    0000 0000 0000 0000 0000 0000 0000 0100           4
>>                                      1
-------------------------------------------------------
    0000 0000 0000 0000 0000 0000 0000 0010           2
```

- If the first bit is 0, right shifting fills with 0.
- If the first bit is 1, right shifting fills with 1.

Signed Arithmetics

Signed >>

- In C, *signed* right shift and *unsigned* right shift looks exactly the same
- C will use *signed* shift if the variable being shifted is a *signed* integer

```
int x = ~0;
x = x >> 4;
printf("0x%x\n", x);
```

```
0xffffffff
```

```
unsigned int x = ~0;
x = x >> 4;
printf("0x%x\n", x);
```

```
0x0fffffff
```

Signed Arithmetics

- Signed extension works the same way:

```
short x = ~0;
int y = x;
printf("0x%x\n", y);
```

```
0xffffffff
```

```
unsigned short x = ~0;
unsigned int y = x;
printf("0x%x\n", y);
```

```
0x0000ffff
```

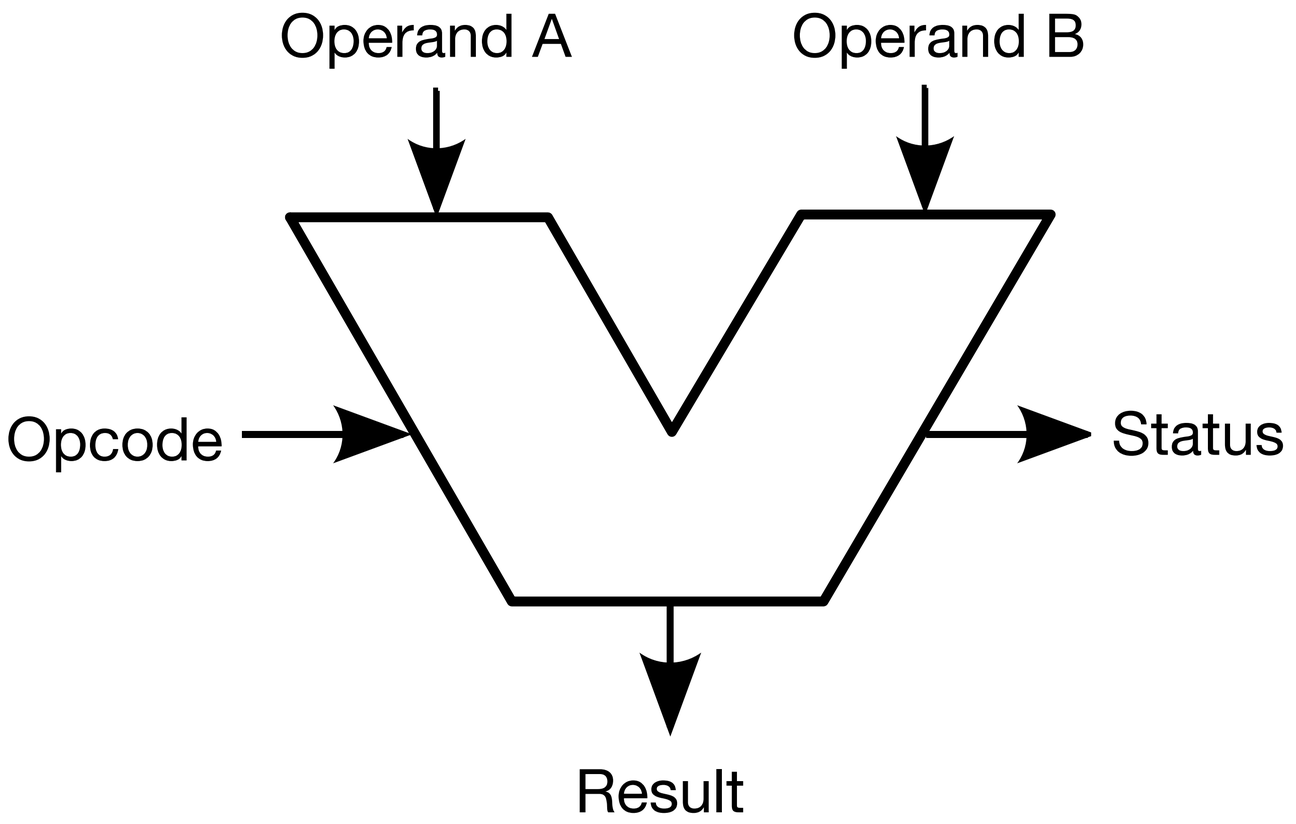
Octal Number

Octal Number

- Base-8 numbers, 0-7
- Corresponds to 3 bits
- In C, prefix by 0, e.g. `0123`, is 83
- Not very common

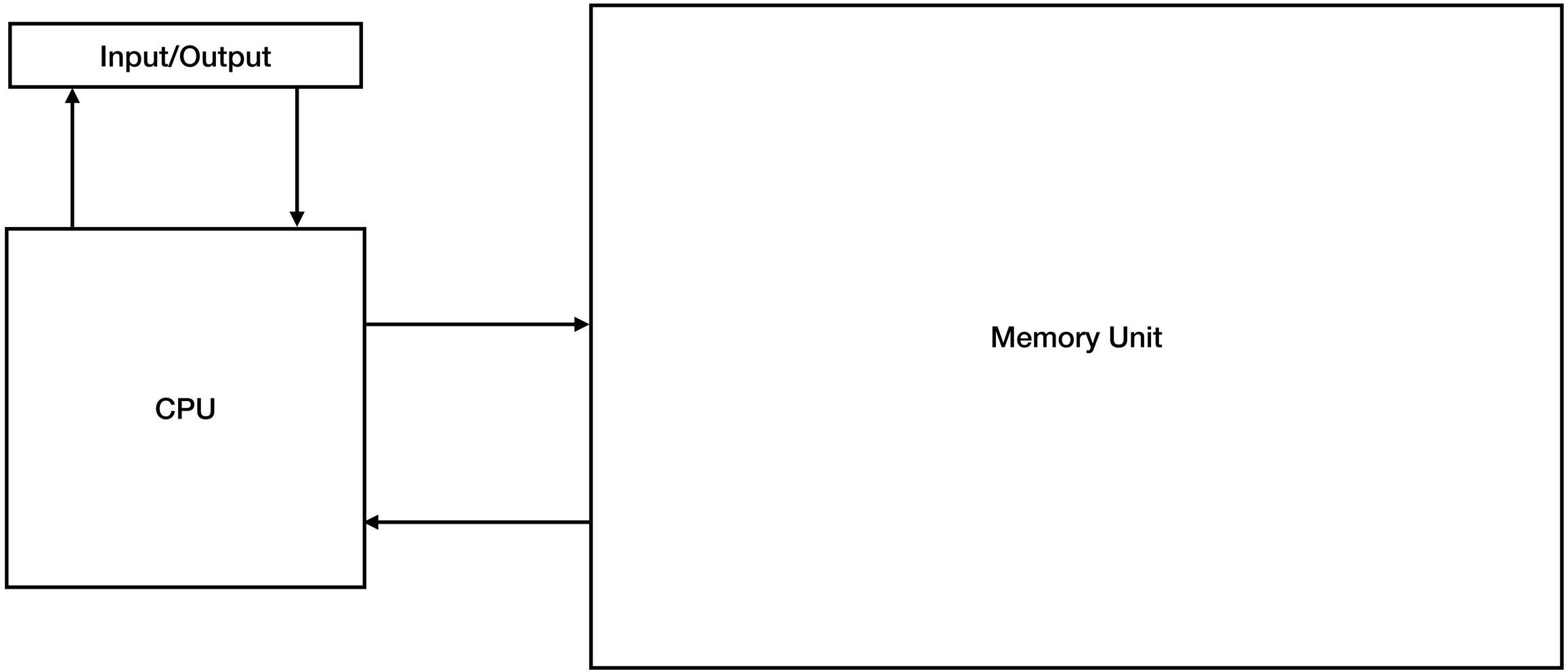
Arithmetic Logic Unit

```
0000 and
0001  or
0010 sub
0100 add
1010 mov
     ...
```

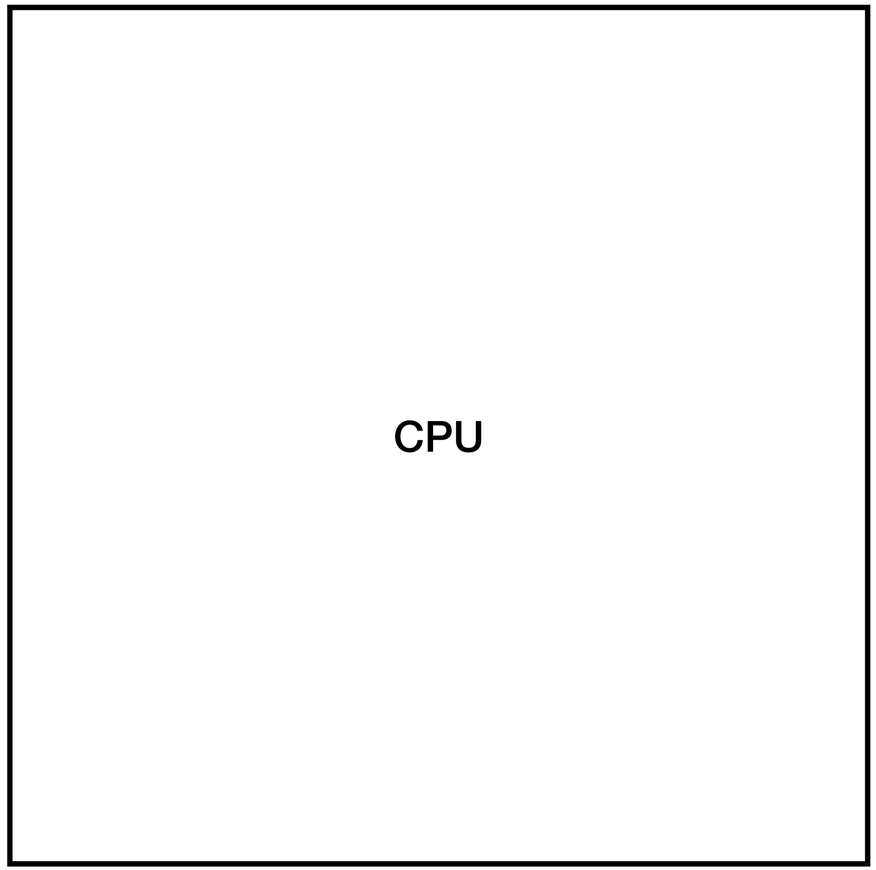



```
overflow?
iszero?
isgreater?
...
```

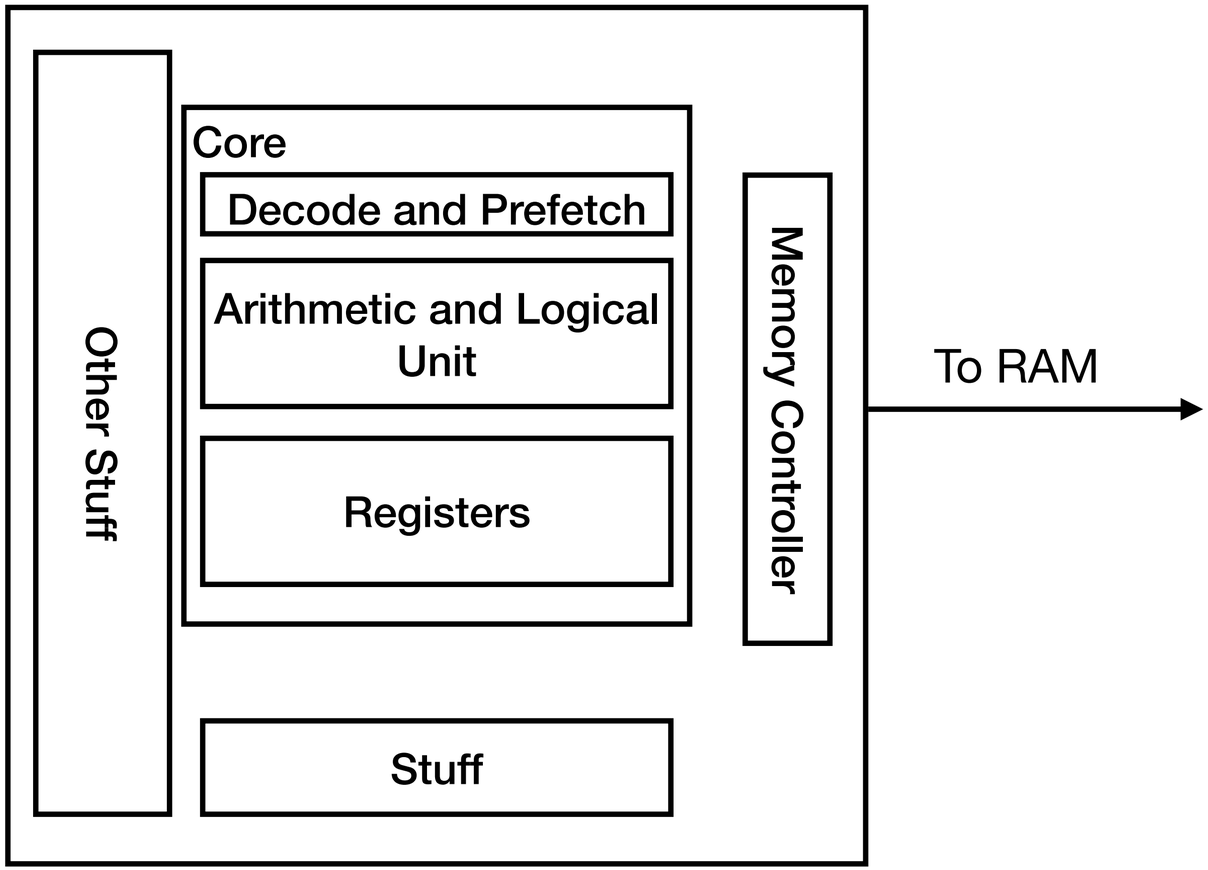
A Von Neumann Machine



A Von Neumann Machine

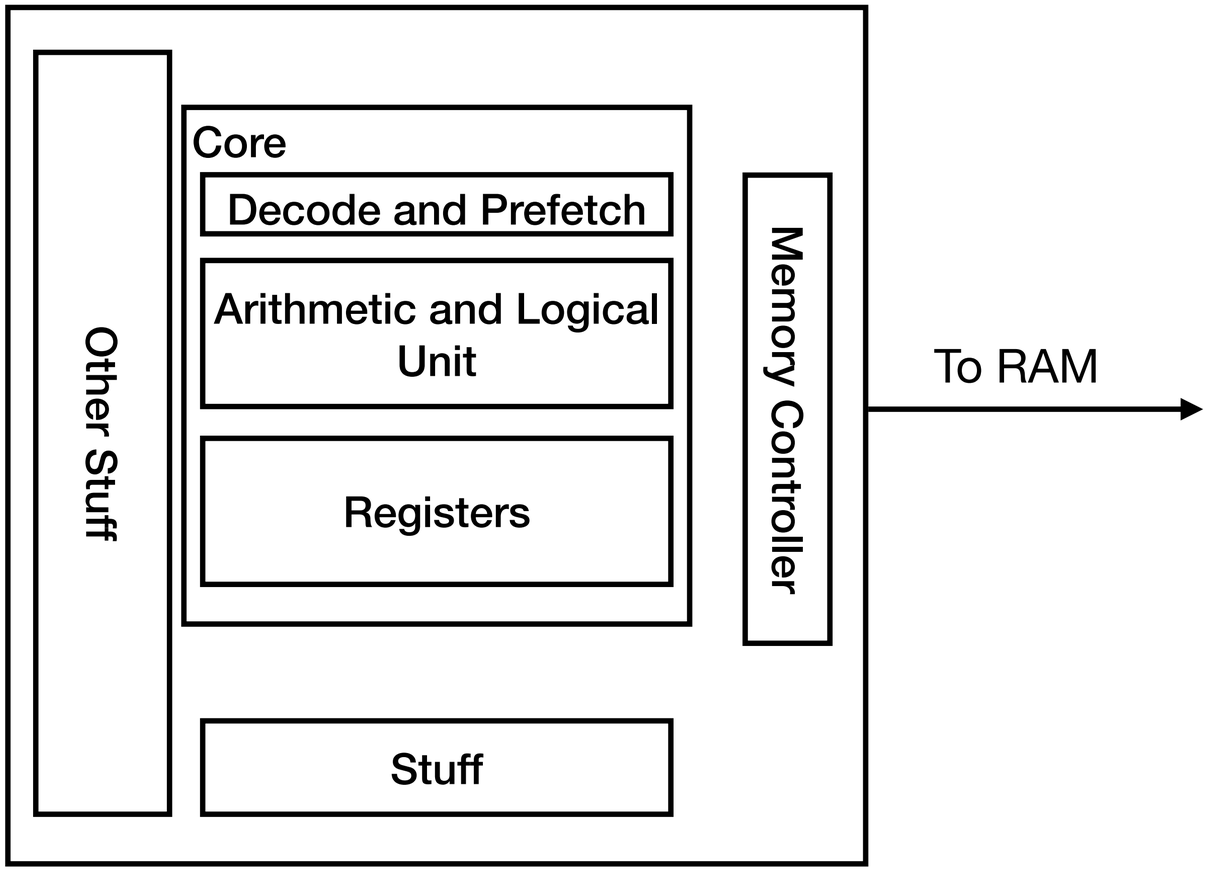


CPU



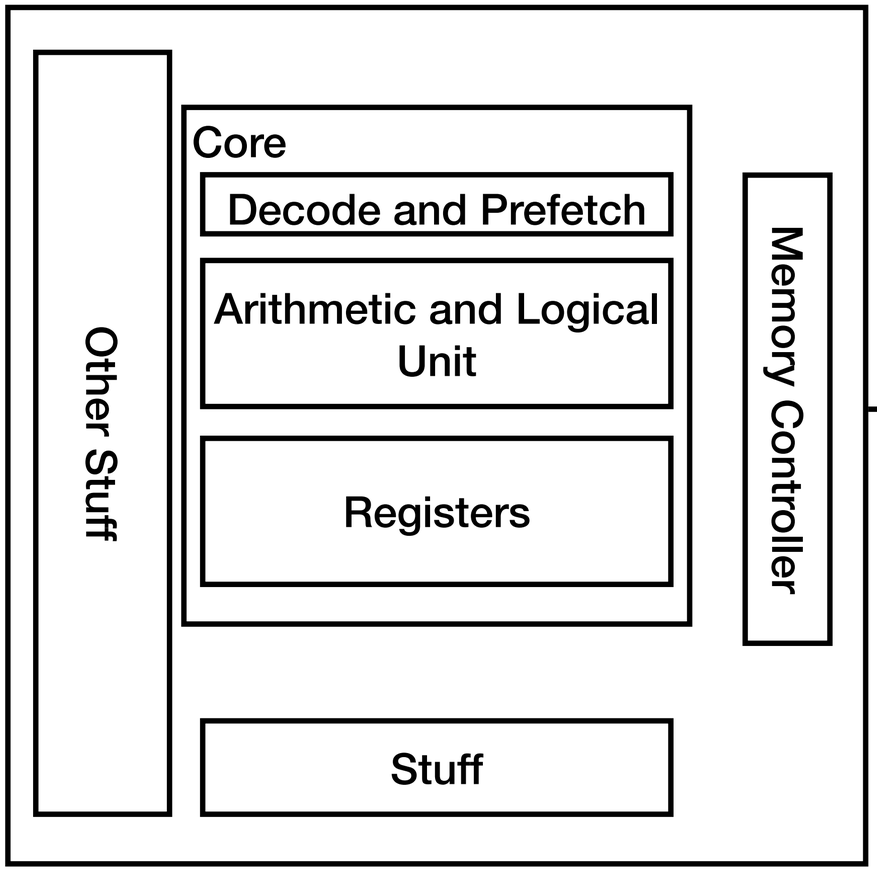
- *Registers*: named locations storing 64-bit values. These registers can hold integers or addresses (pointers).
  - Some registers keep track of program states; others hold temporary data, such as local variables
- *ALU*: reads from registers and perform calculations.

CPU



- *Cache*: stores recently read memory to improve performance.
  - 144!
- *Program counter* (pc), or *instruction pointer* (ip), points at some instruction in memory.

CPU



- From the time power is applied to the system, until the power is shut off, CPU performs the same basic tasks repeatedly:
- Reads the instruction pointed by *pc*
- Interprets the bits in the instruction
- Performs some operations as instructed
- Updates the *pc* to point to the *next* instruction

Instructions

- Load: copy some bytes from memory to a register
- Store: copy some bytes from a register to memory
- Update: copy the contents of two registers to the ALU, which does some calculation and stores the result in a register
- I/O operations: read/write from an I/O device into a register
- Jump: Set the *pc* to be some arbitrary value

Instructions

```c
int accum = 0;

int sum(int x, int y)
{
        int t = x + y;
        accum += t;
        return t;
}
```

main.c

```c
int sum(int x, int y);

int main(void)
{
        return sum(1, 3);
}
```

```
clang -O2 -o prog main.c
```

Instructions

```
0000000000401110 <sum>:
  401110:89 f8                    mov    %edi,%eax
  401112:01 f0                    add    %esi,%eax
  401114:01 05 12 2f 00 00        add    %eax,0x2f12(%rip)        # 40402c <accum>
  40111a:c3                       retq
  40111b:0f 1f 44 00 00           nopl   0x0(%rax,%rax,1)

0000000000401120 <main>:
  401120:bf 01 00 00 00           mov    $0x1,%edi
  401125:be 03 00 00 00           mov    $0x3,%esi
  40112a:e9 e1 ff ff ff           jmpq   401110 <sum>
  40112f:90                       nop
```

Instructions

```
0000000000401110 <sum>:
  401110:89 f8                    mov    %edi,%eax
  401112:01 f0                    add    %esi,%eax
  401114:01 05 12 2f 00 00        add    %eax,0x2f12(%rip)        # 40402c <accum>
  40111a:c3                       retq
  40111b:0f 1f 44 00 00           nopl   0x0(%rax,%rax,1)

0000000000401120 <main>:
  401120:bf 01 00 00 00           mov    $0x1,%edi
  401125:be 03 00 00 00           mov    $0x3,%esi
  40112a:e9 e1 ff ff ff           jmpq   401110 <sum>
  40112f:90                       nop
```

Instructions

```
0000000000401110 <sum>:
  401110:89 f8                    mov    %edi,%eax
  401112:01 f0                    add    %esi,%eax
  401114:01 05 12 2f 00 00        add    %eax,0x2f12(%rip)        # 40402c <accum>
  40111a:c3                       retq
  40111b:0f 1f 44 00 00           nopl   0x0(%rax,%rax,1)

0000000000401120 <main>:
  401120:bf 01 00 00 00           mov    $0x1,%edi
  401125:be 03 00 00 00           mov    $0x3,%esi
  40112a:e9 e1 ff ff ff           jmpq   401110 <sum>
  40112f:90                       nop
```

jump to location 401110

Instructions

```
0000000000401110 <sum>:
  401110:89 f8                    mov    %edi,%eax
  401112:01 f0                    add    %esi,%eax
  401114:01 05 12 2f 00 00        add    %eax,0x2f12(%rip)        # 40402c <accum>
  40111a:c3                       retq
  40111b:0f 1f 44 00 00           nopl   0x0(%rax,%rax,1)

0000000000401120 <main>:
  401120:bf 01 00 00 00           mov    $0x1,%edi
  401125:be 03 00 00 00           mov    $0x3,%esi
  40112a:e9 e1 ff ff ff           jmpq   401110 <sum>
  40112f:90                       nop
```

The global variable has been assigned a location 40402c

Bits

- A bit answers a yes/no question
  - Bits have no inherent meanings; to read encoding, need to know the intended interpretation
  - Any device that can be one of two states can encode a bit.
- Binary and hexadecimal numbers
  - $xyz_m = x \cdot m^2 + y \cdot m^1 + z \cdot m^0$
  - One hex digit corresponds two 4 binary digits (bits) -- easy conversion between the two
- Positive and negative numbers
- Arithmetics
  - `+`, `-`, `<<`, `&`, `|`, `^`, `~`, two `>>`, negate
  - Bit-packing

Bits

- For a 32-bit integer:
  - Range: unsigned [0, 2^{32} - 1]; signed [-2^{31}, 2^{31} - 1].
  - Meaning of the highest bit: unsigned 2^{31}, signed -2^{31}.
- Negate a number: flips all bits and add 1 (-n == ~n + 1).
- Most arithmetic operators work regardless of the signedness of the integer except right shift:
  - Signed right shift (arithmetic right shift): fills with the highest bit.
  - Unsigned right shift (logical right shift): fills with 0.
  - Left shift is the same.