Bit Arithmetics

CS143: lecture 15

Konstantinos Ameranis, June 23

Unsigned Arithmetics

- unsigned means without sign (+/-). In C, `unsigned int x = 4`
- Why unsigned?
  - 32 bits means 2^32 choices
  - If we don't allow negative number, we have range [0, 2^32 - 1]
  - If we have negative numbers, the maximum has to be smaller.
- Most things that we care about are non-negative
  - counts, lengths, indices, dimensions...

Unsigned Arithmetics

Operators

- Arithmetic operators: `+`, `-`, `*`, `/`
- Bitwise operators: `&`, `|`, `^`, `~`
- Shifts: `<<`, `>>`

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
  1101 0011 1100 0001 0010 0100 1001 1111
& 0101 0101 0101 0101 0101 0101 0101 0101
-----------------------------------------
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
  1101 0011 1100 0001 0010 0100 1001 1111
& 0101 0101 0101 0101 0101 0101 0101 0101
-----------------------------------------
                                      0001
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
  1101 0011 1100 0001 0010 0100 1001 1111
& 0101 0101 0101 0101 0101 0101 0101 0101
-----------------------------------------
                                     0101
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
  1101 0011 1100 0001 0010 0100 1001 1111
& 0101 0101 0101 0101 0101 0101 0101 0101
-----------------------------------------
                                    0101
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
  1101 0011 1100 0001 0010 0100 1001 1111
& 0101 0101 0101 0101 0101 0101 0101 0101
-----------------------------------------
                                   0111
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
  1101 0011 1100 0001 0010 0100 1001 1111
& 0101 0101 0101 0101 0101 0101 0101 0101
-----------------------------------------
                                1  0111
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
  1101 0011 1100 0001 0010 0100 1001 1111
& 0101 0101 0101 0101 0101 0101 0101 0101
-----------------------------------------
                              01 0111
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
  1101 0011 1100 0001 0010 0100 1001 1111
& 0101 0101 0101 0101 0101 0101 0101 0101
-----------------------------------------
  0101 0001 0100 0001 0000 0100 0001 0101
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
  1101 0011 1100 0001 0010 0100 1001 1111
| 0101 0101 0101 0101 0101 0101 0101 0101
-----------------------------------------
                                      1
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
  1101 0011 1100 0001 0010 0100 1001 1111
| 0101 0101 0101 0101 0101 0101 0101 0101
-----------------------------------------
                                     11
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
  1101 0011 1100 0001 0010 0100 1001 1111
| 0101 0101 0101 0101 0101 0101 0101 0101
-----------------------------------------
                                     111
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

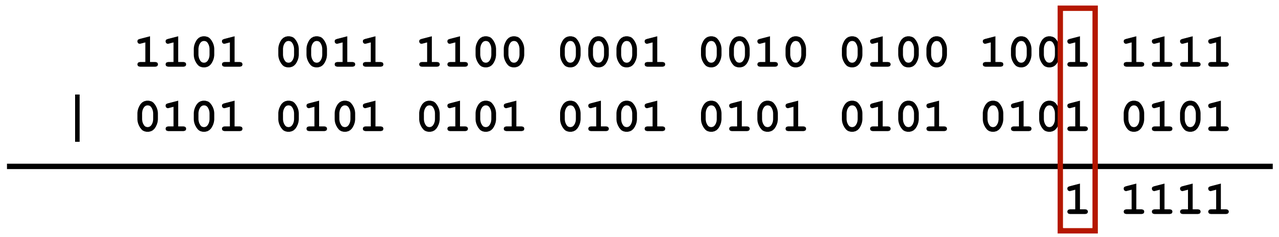
n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

```
  1101 0011 1100 0001 0010 0100 1001 1111
| 0101 0101 0101 0101 0101 0101 0101 0101
-----------------------------------------
                                   1111
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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



Unsigned Arithmetics

```
unsigned int n = 45;
```

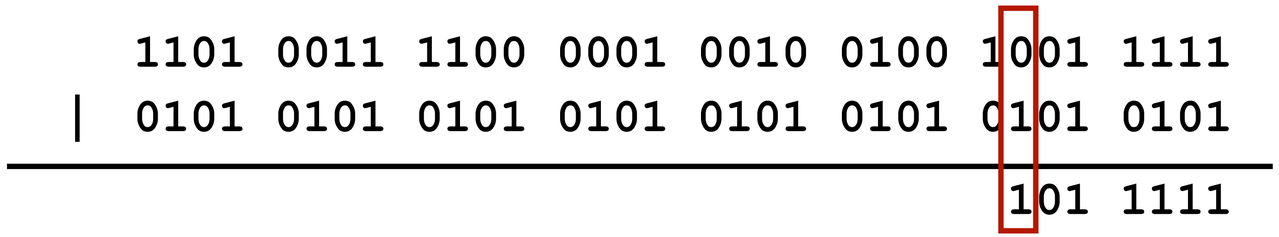
n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

```
  1101 0011 1100 0001 0010 0100 1001 1111
| 0101 0101 0101 0101 0101 0101 0101 0101
-----------------------------------------
                                01 1111
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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



Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
  1101 0011 1100 0001 0010 0100 1001 1111
| 0101 0101 0101 0101 0101 0101 0101 0101
-----------------------------------------
  1101 0111 1101 0101 0111 0101 1101 1111
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
~ 1101 0011 1100 0001 0010 0100 1001 1111
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
  ~ 1101 0011 1100 0001 0010 0100 1001 1111
-------------------------------------------
                                      0000
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
  ~ 1101 0011 1100 0001 0010 0100 1001 1111
-------------------------------------------
                                  0110 0000
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
  ~ 1101 0011 1100 0001 0010 0100 1001 1111
-------------------------------------------
                        1011 0110 0000
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
  ~ 1101 0011 1100 0001 0010 0100 1001 1111
-------------------------------------------
    0010 1100 0011 1110 1101 1011 0110 0000
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

Exclusive-Or (XOR): 1 if the bits are different 0 if the bits are the same.

```
  1101 0011 1100 0001 0010 0100 1001 1111
^ 0101 0101 0101 0101 0101 0101 0101 0101
-----------------------------------------
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
  1101 0011 1100 0001 0010 0100 1001 1111
^ 0101 0101 0101 0101 0101 0101 0101 0101
-----------------------------------------
                                       0
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
  1101 0011 1100 0001 0010 0100 1001 1111
^ 0101 0101 0101 0101 0101 0101 0101 0101
-----------------------------------------
                                      10
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
  1101 0011 1100 0001 0010 0100 1001 1111
^ 0101 0101 0101 0101 0101 0101 0101 0101
-----------------------------------------
                                     010
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
  1101 0011 1100 0001 0010 0100 1001 1111
^ 0101 0101 0101 0101 0101 0101 0101 0101
-----------------------------------------
                                   1010
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
  1101 0011 1100 0001 0010 0100 1001 1111
^ 0101 0101 0101 0101 0101 0101 0101 0101
-----------------------------------------
                                0  1010
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
  1101 0011 1100 0001 0010 0100 1001 1111
^ 0101 0101 0101 0101 0101 0101 0101 0101
-----------------------------------------
                                10 1010
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
  1101 0011 1100 0001 0010 0100 1001 1111
^ 0101 0101 0101 0101 0101 0101 0101 0101
-----------------------------------------
                               100 1010
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
  1101 0011 1100 0001 0010 0100 1001 1111
^ 0101 0101 0101 0101 0101 0101 0101 0101
-----------------------------------------
  1000 0110 1001 0100 0111 0001 1100 1010
```

Unsigned Arithmetics

- `~` is not the same as `!`, `&` is not the same as `&&`, `|` is not the same as `||`
  - `!`, `&&`, `||` are *Boolean* operators -- they treat zero as false and non-zero as true
  - `~`, `&`, `|` are *bitwise* operators -- they operate on each bit.

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
   0000 0000 0000 0000 0000 0000 0010 1101
<<                                      1
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
    0000 0000 0000 0000 0000 0000 0010 1101
<<                                       1
-----------------------------------------------
    0000 0000 0000 0000 0000 0000 0101 1010
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
    0000 0000 0000 0000 0000 0000 0010 1101
<<                                     1
-----------------------------------------
    0000 0000 0000 0000 0000 0000 0101 1010   = 0x5A
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
    0000 0000 0000 0000 0000 0000 0010 1101
<<                                     1
-----------------------------------------
    0000 0000 0000 0000 0000 0000 0101 1010   = 0x5A  = 90
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
   0000 0000 0000 0000 0000 0000 0010 1101
<<                                      1
-----------------------------------------
   0000 0000 0000 0000 0000 0000 0101 1010   = 0x5A  = 90
<<                                     16
-----------------------------------------
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
    0000 0000 0000 0000 0000 0000 0010 1101
<<                                       1
-----------------------------------------------
    0000 0000 0000 0000 0000 0000 0101 1010   = 0x5A  = 90
<<                                      16
-----------------------------------------------
    0000 0000 0101 1010 0000 0000 0000 0000
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
   0000 0000 0000 0000 0000 0000 0010 1101
<<                                      1
-----------------------------------------
   0000 0000 0000 0000 0000 0000 0101 1010   = 0x5A  = 90
<<                                     16
-----------------------------------------
   0000 0000 0101 1010 0000 0000 0000 0000
<<                                     16
-----------------------------------------
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
   0000 0000 0000 0000 0000 0000 0010 1101
<<                                      1
-----------------------------------------
   0000 0000 0000 0000 0000 0000 0101 1010   = 0x5A  = 90
<<                                     16
-----------------------------------------
   0000 0000 0101 1010 0000 0000 0000 0000
<<                                     16
-----------------------------------------
   0000 0000 0000 0000 0000 0000 0000 0000
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
   0000 0000 0000 0000 0000 0000 0010 1101
<<                                      1
-----------------------------------------
   0000 0000 0000 0000 0000 0000 0101 1010   = 0x5A  = 90
<<                                     16
-----------------------------------------
   0000 0000 0101 1010 0000 0000 0000 0000
<<                                     16
-----------------------------------------
   0000 0000 0000 0000 0000 0000 0000 0000
```

- When shift overflows, the bit is lost.

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
    0000 0000 0000 0000 0000 0000 0010 1101
>>                                       1
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
    0000 0000 0000 0000 0000 0000 0010 1101
>>                                       1
-----------------------------------------------
    0000 0000 0000 0000 0000 0000 0001 0110
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
    0000 0000 0000 0000 0000 0000 0010 1101
>>                                       1
-----------------------------------------------
    0000 0000 0000 0000 0000 0000 0001 0110   = 0x16
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
    0000 0000 0000 0000 0000 0000 0010 1101
>>                                       1
-----------------------------------------------
    0000 0000 0000 0000 0000 0000 0001 0110   = 0x16  = 22
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
    0000 0000 0000 0000 0000 0000 0010 1101
>>                                       1
-----------------------------------------------
    0000 0000 0000 0000 0000 0000 0001 0110   = 0x16  = 22
>>                                       8
-----------------------------------------------
    0000 0000 0000 0000 0000 0000 0000 0000
```

Unsigned Arithmetics

```
unsigned int n = 45;
```

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

```
    0000 0000 0000 0000 0000 0000 0010 1101
>>                                       1
-----------------------------------------
    0000 0000 0000 0000 0000 0000 0001 0110   = 0x16  = 22
>>                                       8
-----------------------------------------
    0000 0000 0000 0000 0000 0000 0000 0000
```

- When shift underflows, the bit is lost.

Bit-packing

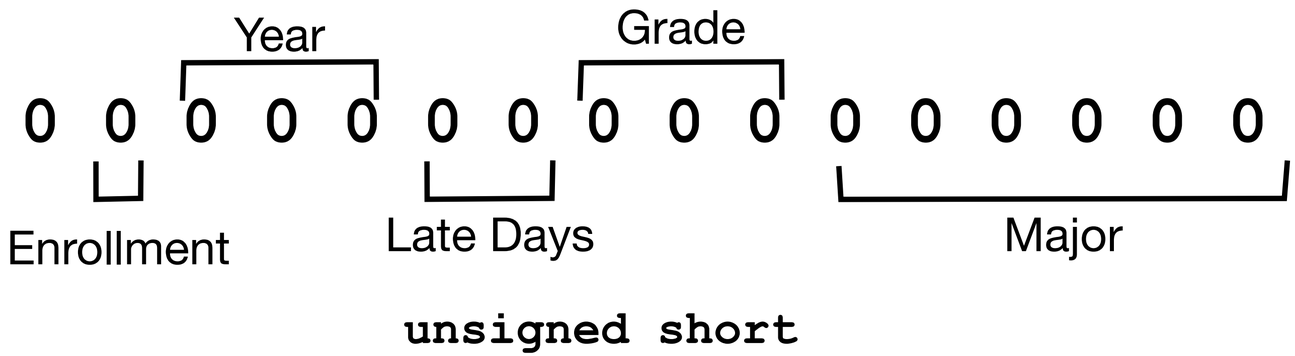
```
struct student {
        int is_enrolled;     /* is student formally enrolled in this class? */
        int year;
        int late_days_used;
        char letter_grade;
        char major[32];
};
```

Bit-packing

- Enrollment: 2 choices (1 bit)
- Year: 5 choices [1, 2, 3, 4, beyond-4] (3 bits)
- Late Days: 4 choices (2 bits)
- Letter grade: 5 choices (3 bits)
- Major: 53 majors (6 bits)
- None of these need 32 bit integers!
- 1 + 3 + 2 + 3 + 6 = 15, in fact we can fit the entire record inside a short.

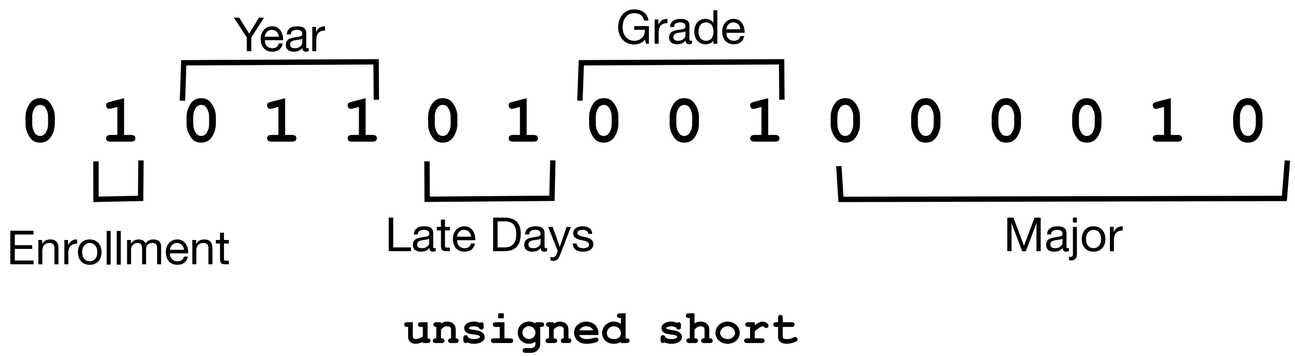
```
struct student {
        int is_enrolled;
        int year;
        int late_days_used;
        char letter_grade;
        char major[32];
};
```

Bit-packing



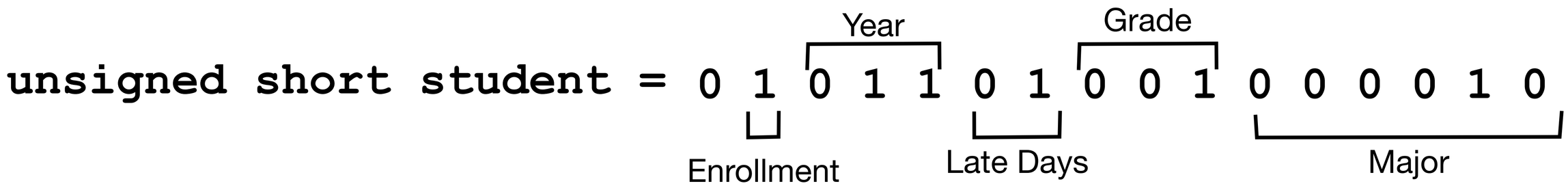
- Enrollment: 2 choices (1 bit)
- Year: 6 choices [0, 1, 2, 3, 4, beyond-4] (3 bits)
- Late Days: 4 choices (2 bits)
- Letter grade: 5 choices (3 bits)
- Major: 53 majors (6 bits)

Bit-packing



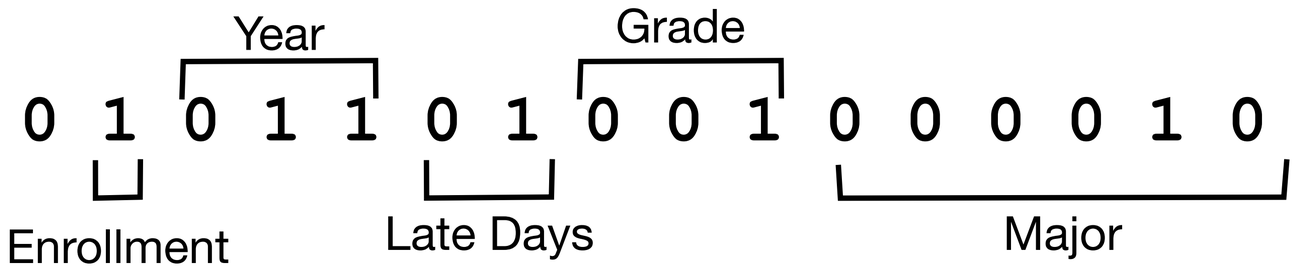
- Enrolled
- Year 3
- 1 late day
- 1st choice of grade
- 2nd choice of major

Bit-packing


unsigned short student = 0 1 0 1 1 0 1 0 0 1 0 0 0 0 1 0
Year
Grade
Enrollment
Late Days
Major

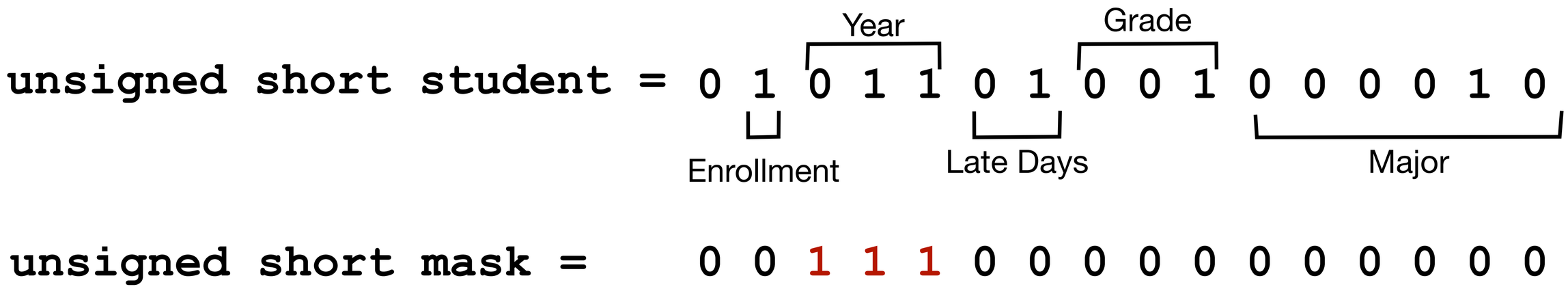
Bit-packing

```
unsigned short student = 0 1 0 1 1 0 1 0 0 1 0 0 0 0 1 0
```

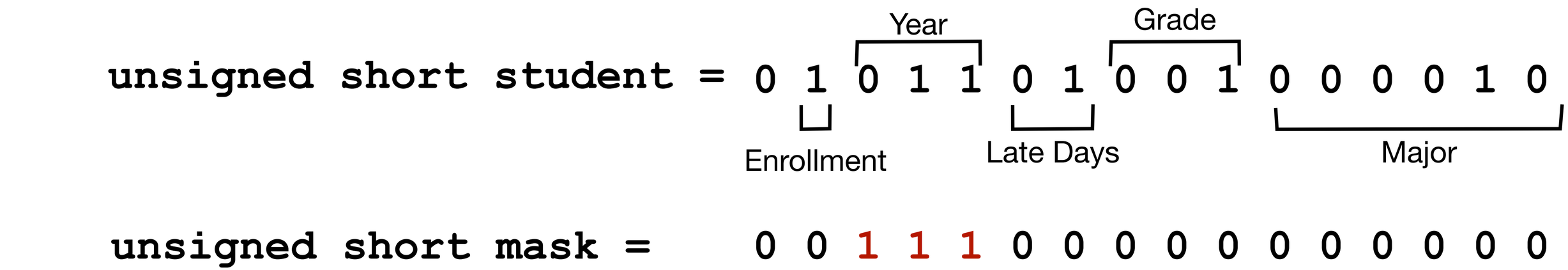



```
unsigned short mask =
```

Bit-packing

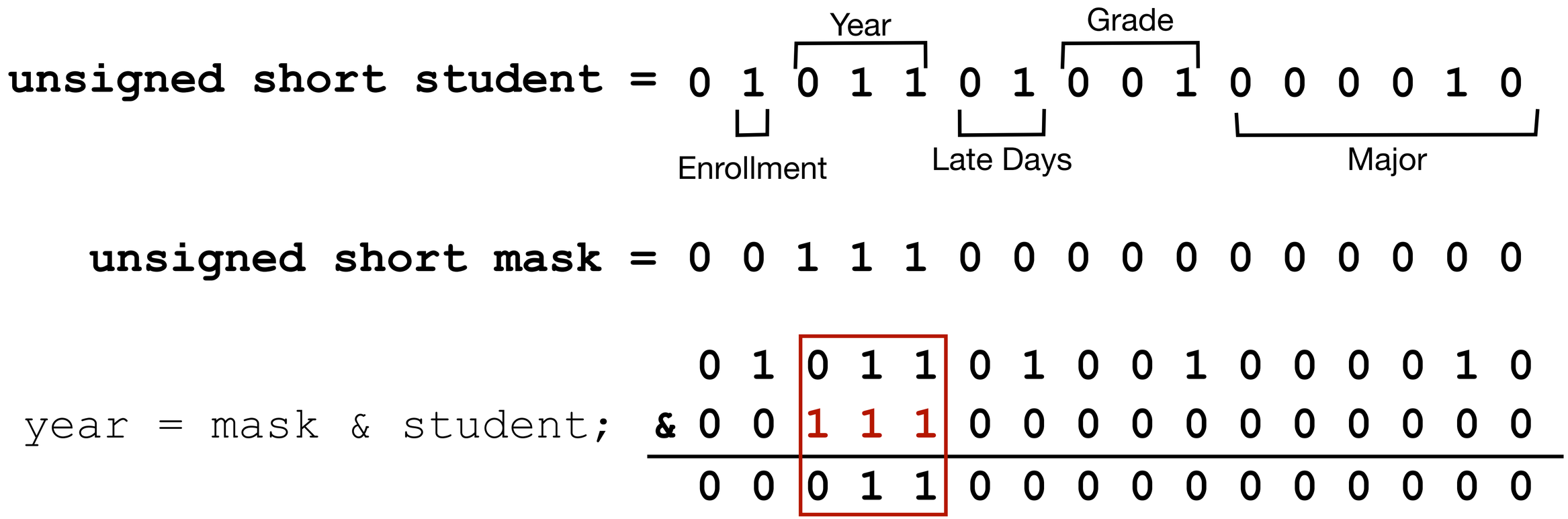

unsigned short student = 0 1 0 1 1 0 1 0 0 1 0 0 0 0 1 0
Year
Grade
Enrollment
Late Days
Major
unsigned short mask = 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0

Bit-packing

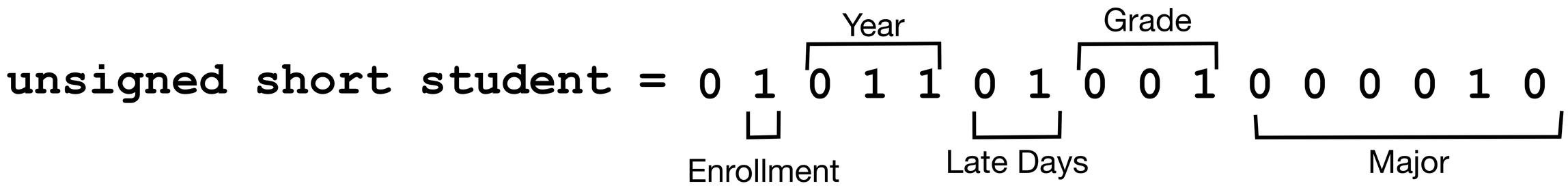



```
int year = mask & student;
```

Bit-packing


unsigned short student = 0 1 0 1 1 0 1 0 0 1 0 0 0 0 1 0
Year
Grade
Enrollment
Late Days
Major
unsigned short mask = 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0
int year = mask & student;
0 1 0 1 1 0 1 0 0 1 0 0 0 0 1 0
& 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0

Bit-packing

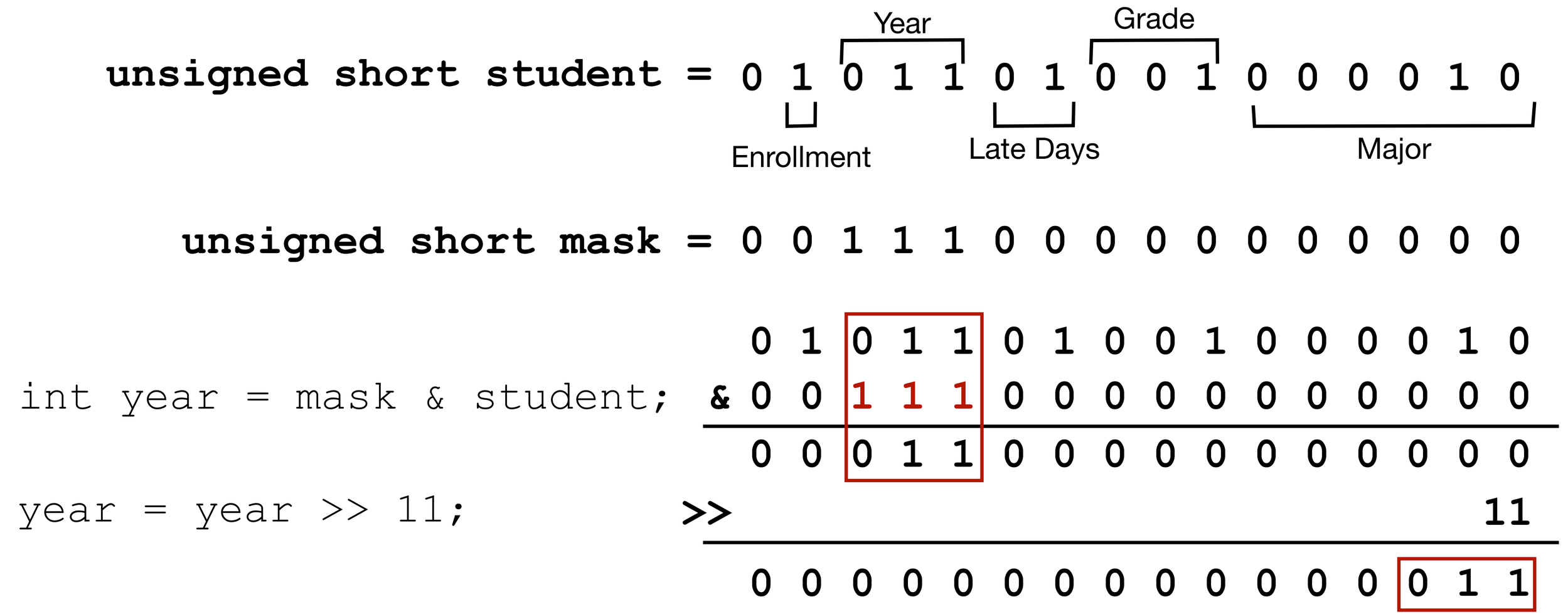



```
int year = mask & student;
```

```
  0 1 0 1 1 0 1 0 0 1 0 0 0 0 1 0
& 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0
-------------------------------
  0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0
```

```
year = year >> 11;
```

Bit-packing

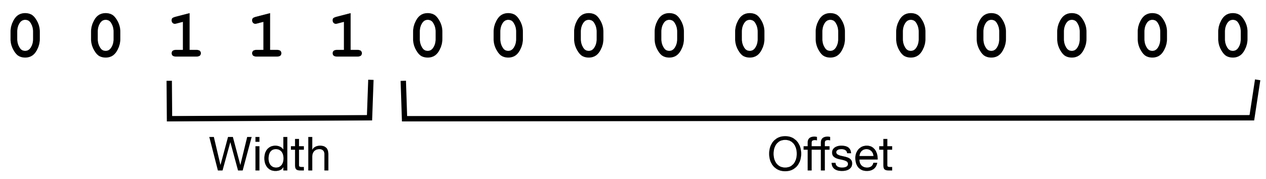

unsigned short student = 0 1 0 1 1 0 1 0 0 1 0 0 0 0 1 0
Year
Grade
Enrollment
Late Days
Major
unsigned short mask = 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 1 0 1 1 0 1 0 0 1 0 0 0 0 1 0
int year = mask & student; & 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0
year = year >> 11; >> 11
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1

Bit-packing

0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0

```
unsigned short mask = 14336;

unsigned short mask = 0x7 << 11;
```

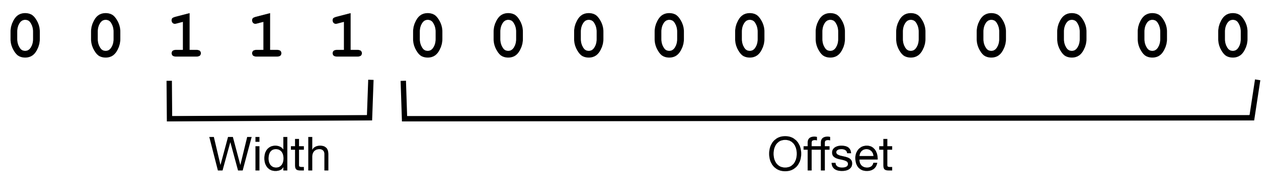
Bit-packing




```
unsigned short mask = 14336;

unsigned short mask = 0x7 << 11;


unsigned short mask = ~0u >> (16 - width) << offset;
```

Future Note: we need to write 0u instead of 0 because by default, 0 is signed, so >> is going to perform a signed right shift.

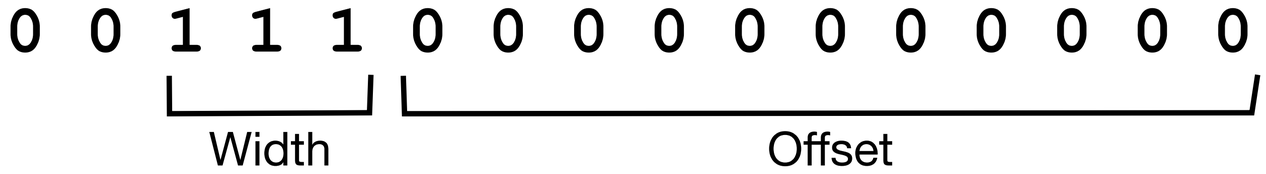
Bit-packing




```
unsigned short mask = ~0 >> (16 - width) << offset;
```

~ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

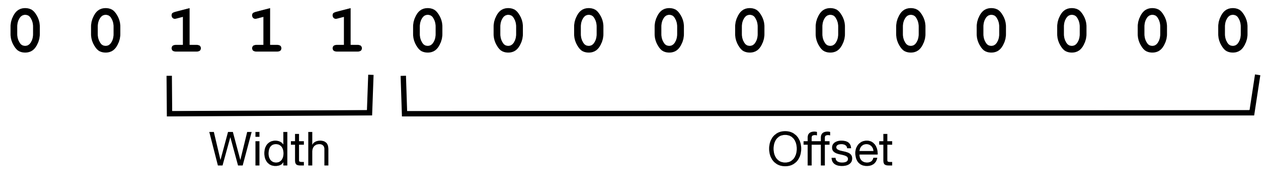
Bit-packing




```
unsigned short mask = ~0 >> (16 - width) << offset;
```

```
~ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
-------------------------------
  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

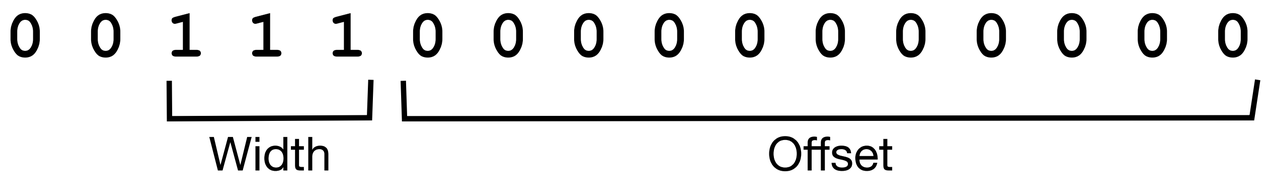
Bit-packing




```
unsigned short mask = ~0 >> (16 - width) << offset;
```

```
  ~ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  ---------------------------------
    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
>>                                13
  ---------------------------------
```

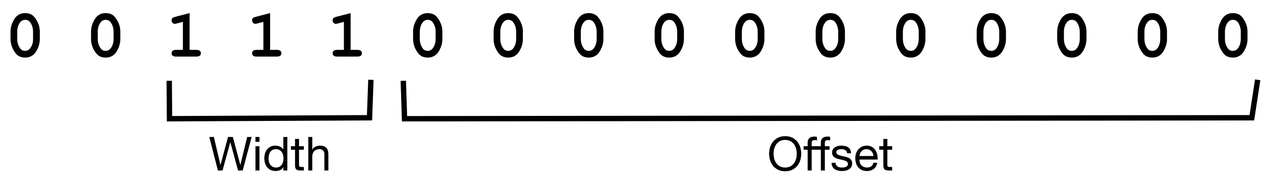
Bit-packing




```
unsigned short mask = ~0 >> (16 - width) << offset;
```

```
  ~ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  -------------------------------
    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
>>                             13
  -------------------------------
    0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1
```

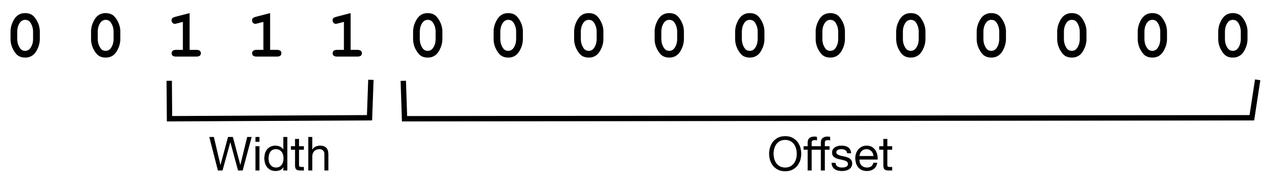
Bit-packing




```
unsigned short mask = ~0 >> (16 - width) << offset;
```

```
  ~ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  ---------------------------------
    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
>>                                13
  ---------------------------------
    0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1
<<                               11
  ---------------------------------
```

Bit-packing



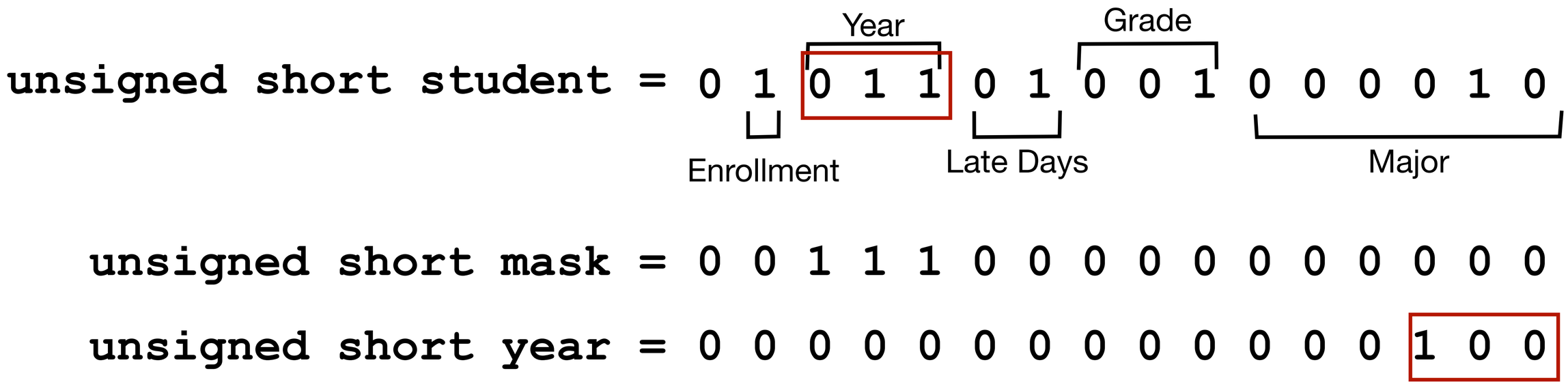

```
unsigned short mask = ~0 >> (16 - width) << offset;
```

```
  ~ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  -------------------------------
    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
>>                             13
  -------------------------------
    0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1
<<                             11
  -------------------------------
    0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0
```

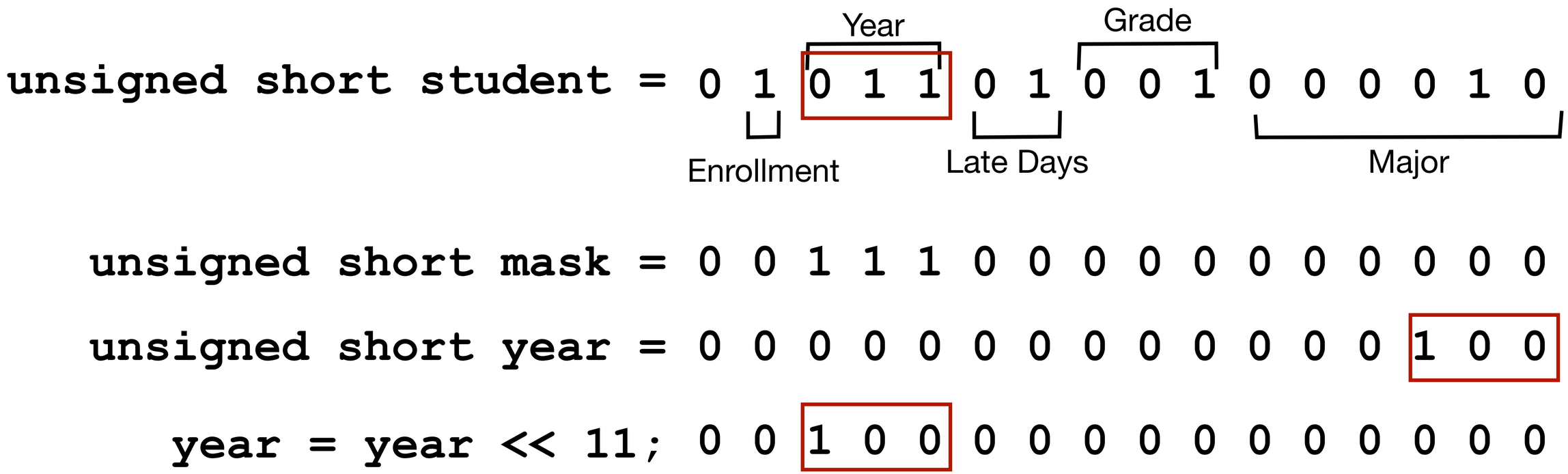
Bit-packing

```
unsigned short student = ...;
unsigned short year_mask = ~0 >> (16 - 3) << 11;

unsigned int year = (student & year_mask) >> 11;
```

Bit-packing


unsigned short student = 0 1 0 1 1 0 1 0 0 1 0 0 0 0 1 0
Year
Grade
Enrollment
Late Days
Major
unsigned short mask = 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0
unsigned short year = 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0

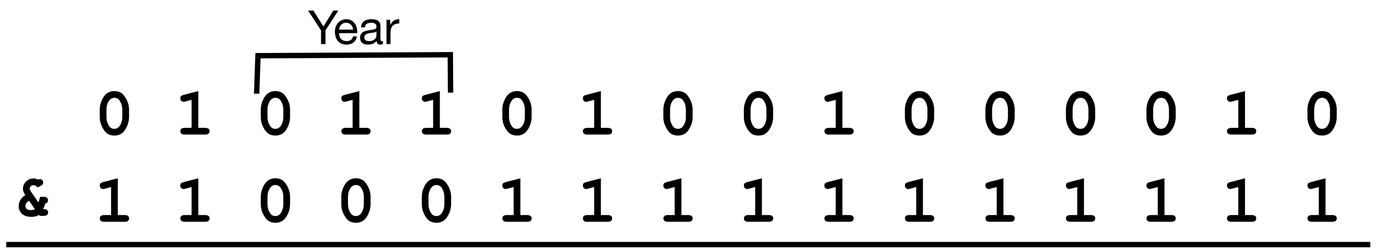
Bit-packing


unsigned short student = 0 1 0 1 1 0 1 0 0 1 0 0 0 0 1 0
Year
Grade
Enrollment
Late Days
Major
unsigned short mask = 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0
unsigned short year = 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
year = year << 11; 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0

??

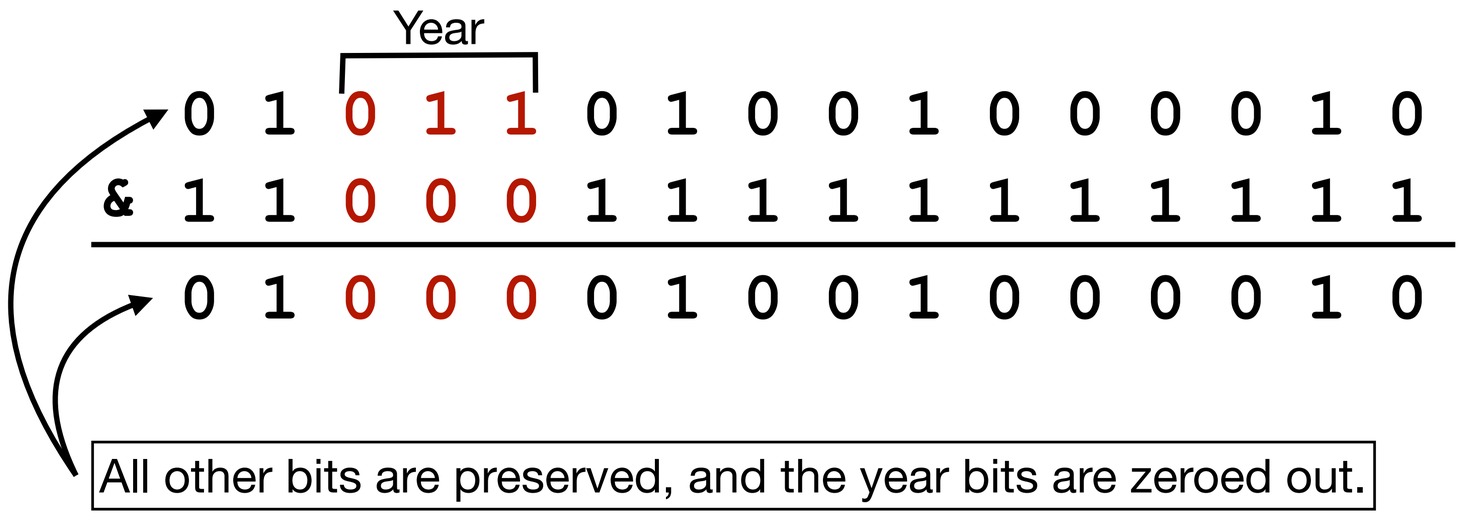
Bit-packing

```
student = student & ~year_mask;
```



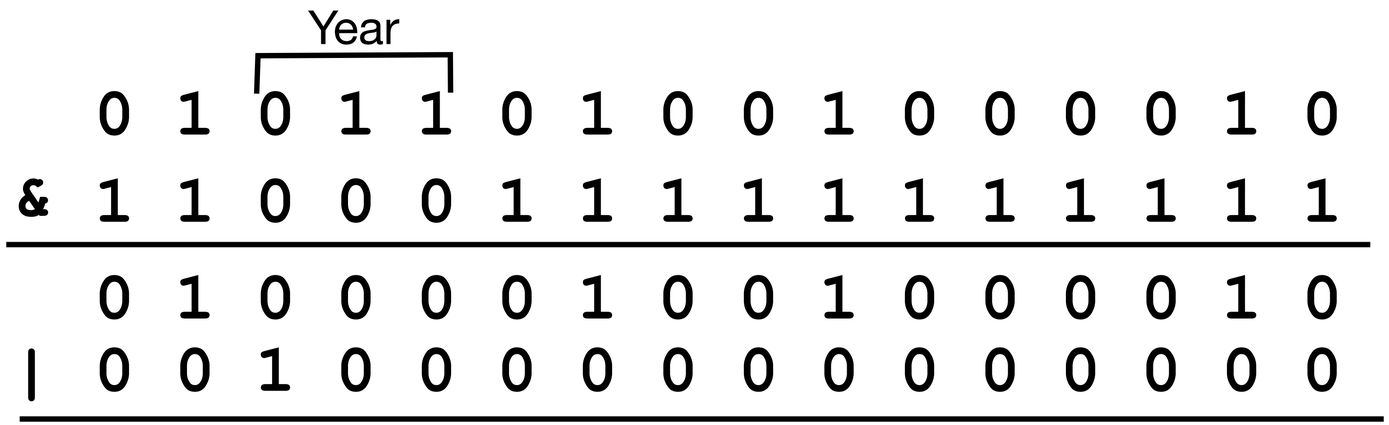
Bit-packing

```
student = student & ~year_mask;
```



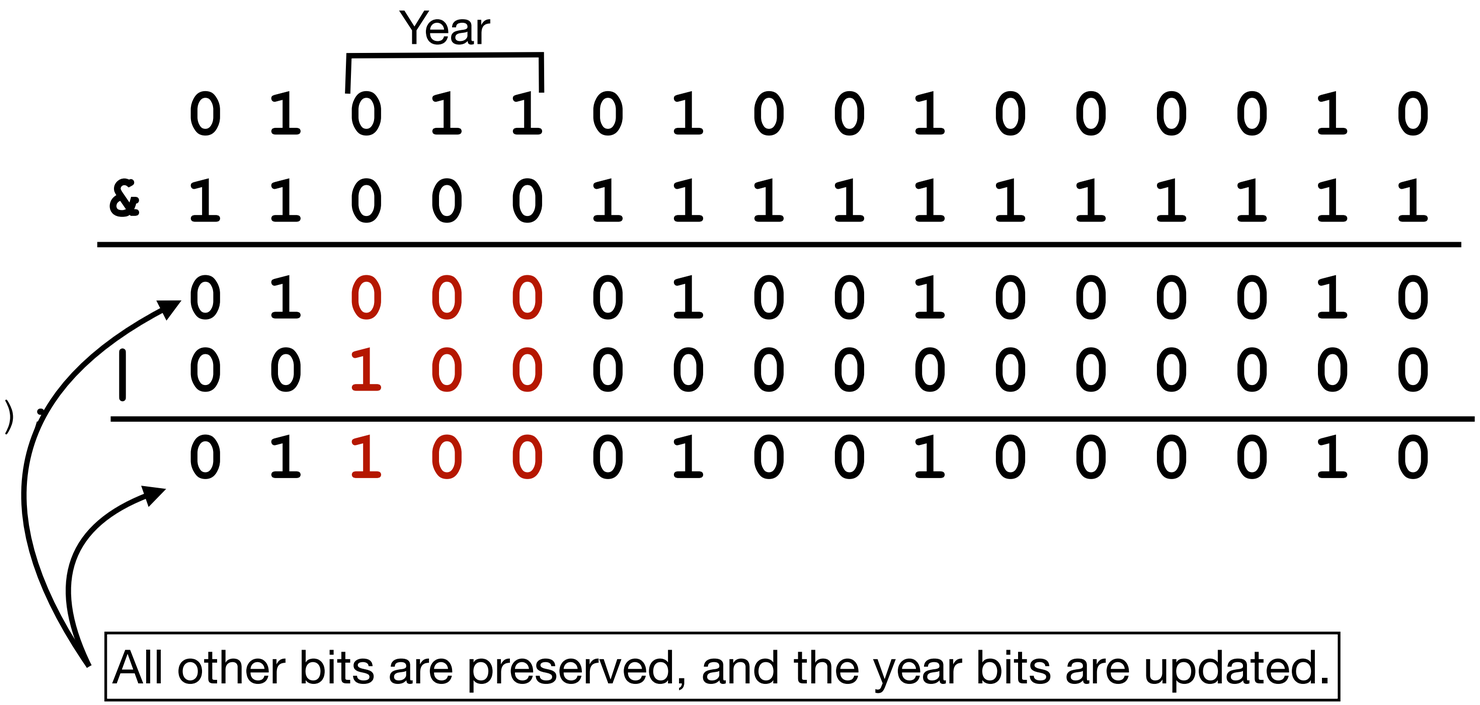
Bit-packing

```
student = student & ~year_mask;

student = student | (year << 11);
```



Bit-packing

```
student = student & ~year_mask;

student = student | (year << 11);
```



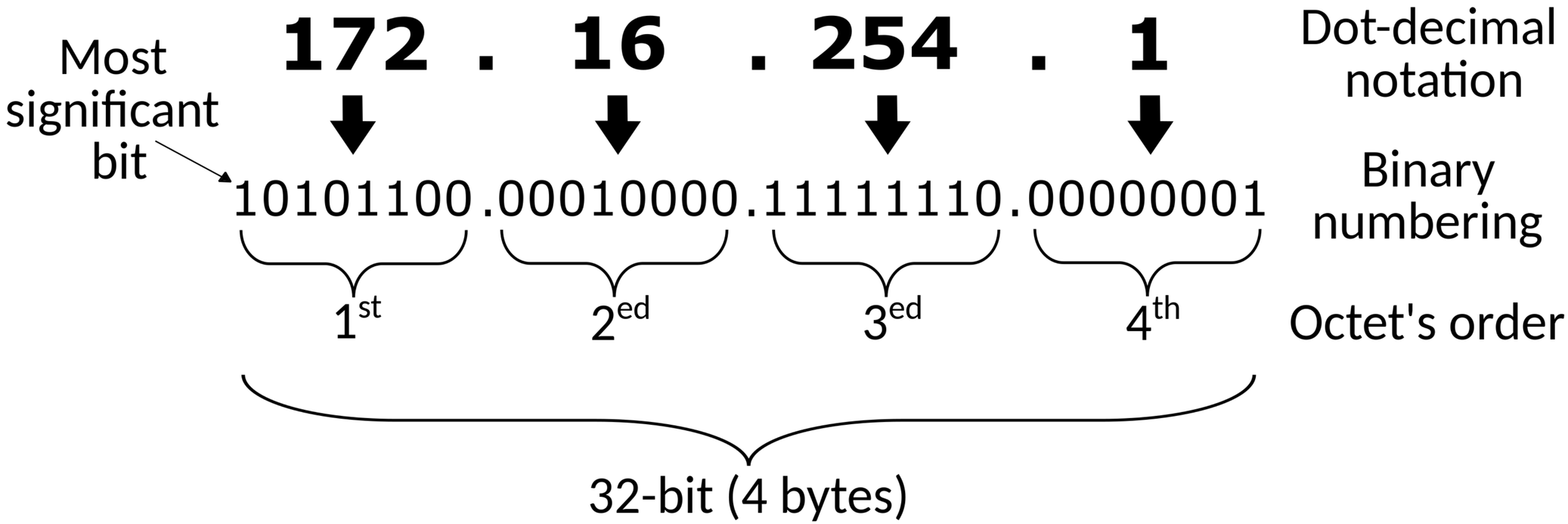
Bit-packing

Summary

- A mask has the bit fields of interest set 1 and all others 0.
- `& mask` gets the bits
- `& ~mask` clears out the bits
- `~0u >> (16 - width) << offset` creates a 16-bit mask with `width` and `offset`.
- After the bits are zeroed out, `|` is used to write new bits in the field.
- Does anyone use this irl?

Bit-packing

Example: IP Address



Bit-packing

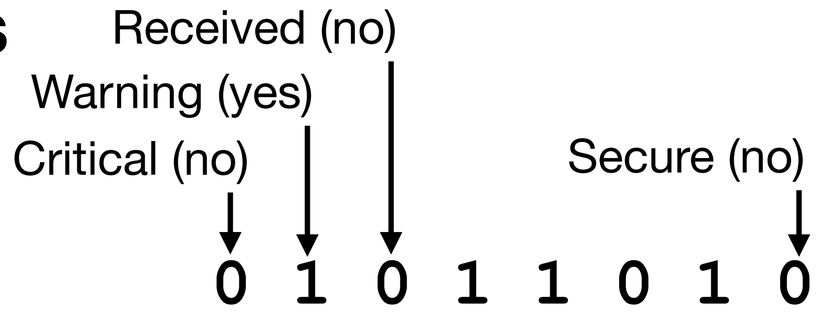
Example: IP Address

IP address	192.168.1.2
Subnet mask	255.255.255.0
Router	192.168.1.1

What is this mask trying to get?

Bit-packing

Example: Flag Bits

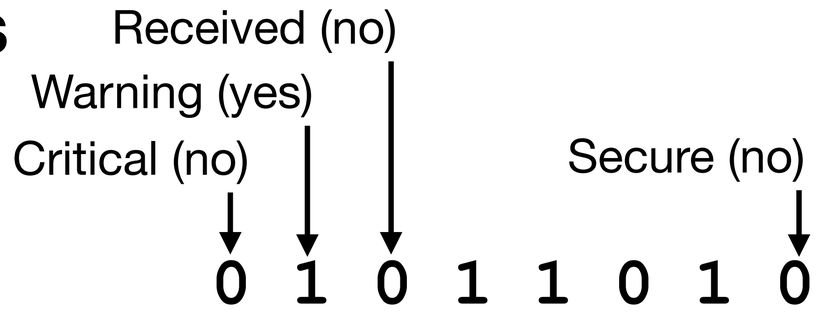


Each bit answers one yes/no question -- 8 Booleans in a byte!

```
const uint8_t CRITICAL = 0x80;   ->10000000
const uint8_t WARNING  = 0x40;   ->01000000
const uint8_t RECEIVED = 0x20;   ->00100000
...
const uint8_t SECURE   = 0x01;   ->00000001
```

Bit-packing

Example: Flag Bits

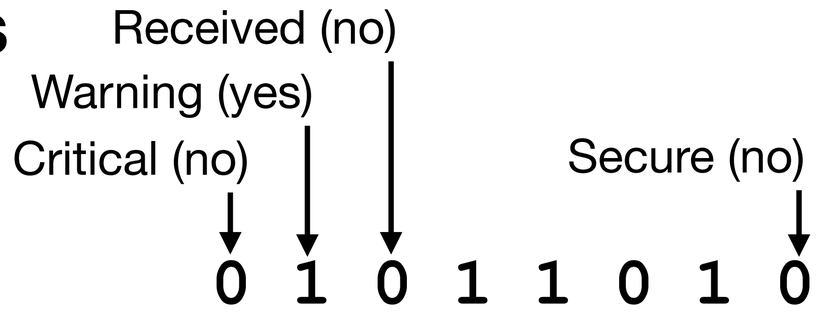



```
const uint8_t CRITICAL = 0x80;   ->10000000
const uint8_t WARNING  = 0x40;   ->01000000
const uint8_t RECEIVED = 0x20;   ->00100000
...
const uint8_t SECURE   = 0x01;   ->00000001

if (flags & CRITICAL) { ... }
```

This is non-zero if and only if the first bit is set.

Bit-packing

Example: Flag Bits



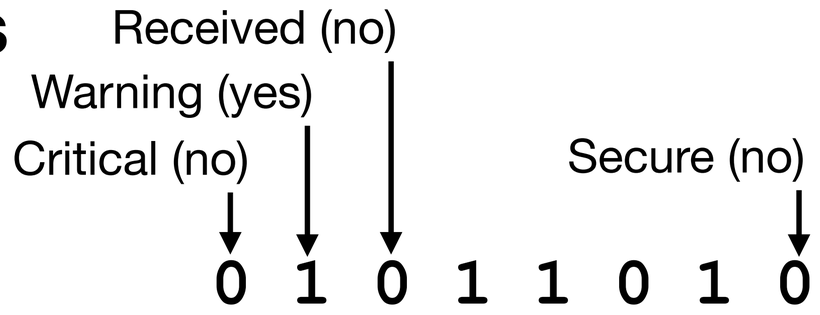

```
const uint8_t CRITICAL = 0x80;   ->10000000
const uint8_t WARNING  = 0x40;   ->01000000
const uint8_t RECEIVED = 0x20;   ->00100000
...
const uint8_t SECURE   = 0x01;   ->00000001

if (flags & (CRITICAL | SECURE)) { ... }
```

10000001

This is non-zero if either the first bit or the last bit is set.

Bit-packing

Example: Flag Bits

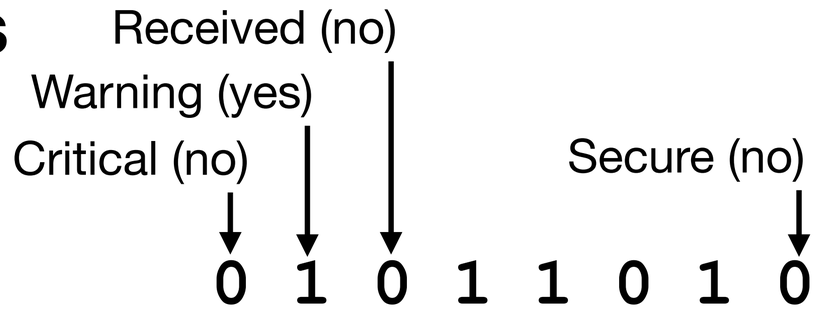



```
const uint8_t CRITICAL = 0x80;   ->10000000
const uint8_t WARNING  = 0x40;   ->01000000
const uint8_t RECEIVED = 0x20;   ->00100000
...
const uint8_t SECURE   = 0x01;   ->00000001


flags |= RECEIVED;
```

This sets the RECEIVED bit.

Bit-packing

Example: Flag Bits




```
const uint8_t CRITICAL = 0x80;   ->10000000
const uint8_t WARNING  = 0x40;   ->01000000
const uint8_t RECEIVED = 0x20;   ->00100000
...
const uint8_t SECURE   = 0x01;   ->00000001


flags &= ~SECURE;
```

This unsets the SECURE bit.

Interlude: Command-Line Arguments *and* ++, --

Negative Numbers

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
  - `signed 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

```
 5 = 0000 0000 0000 0000 0000 0000 0000 0011
-5 = 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

```
 5 = 0000 0000 0000 0000 0000 0000 0000 0011
-5 = 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

```
 5 = 0000 0000 0000 0000 0000 0000 0000 0011
-5 = 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

```
 5 = 0000 0000 0000 0000 0000 0000 0000 0011
-5 = 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
```

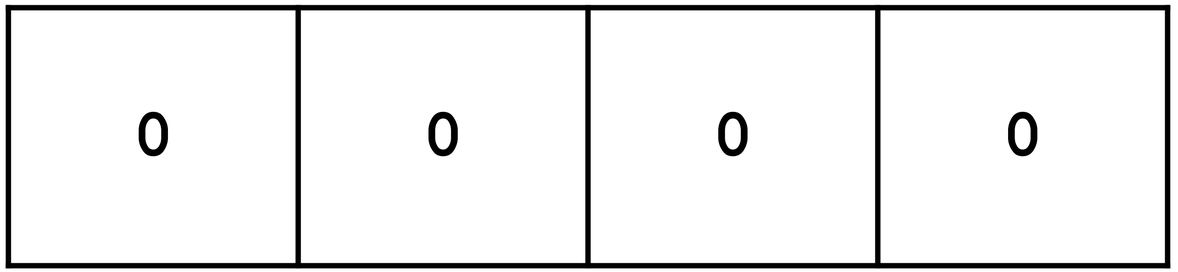
- 5 + (-5) is not 0 :(((
  - Need special circuit for adding signed numbers and unsigned numbers

Negative Numbers

Better Idea: Two's Complement

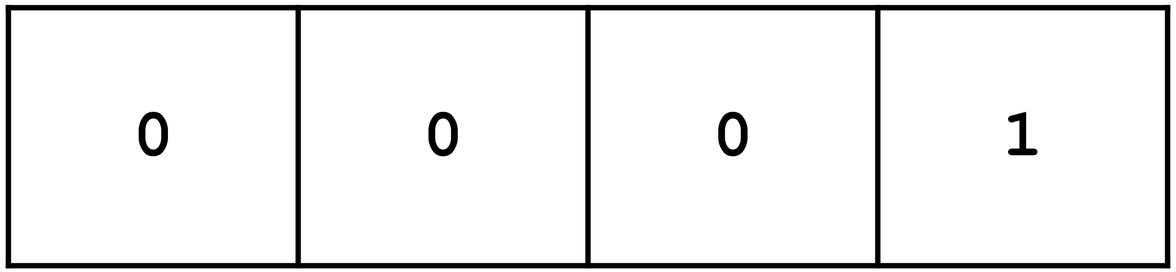
Negative Numbers

Better Idea: Two's Complement



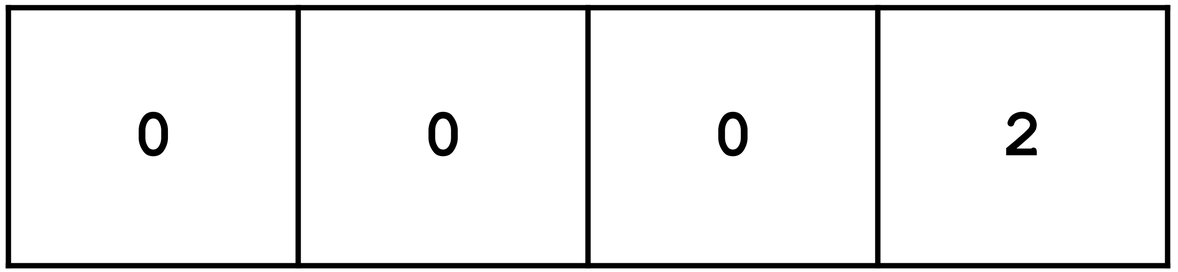
Negative Numbers

Better Idea: Two's Complement



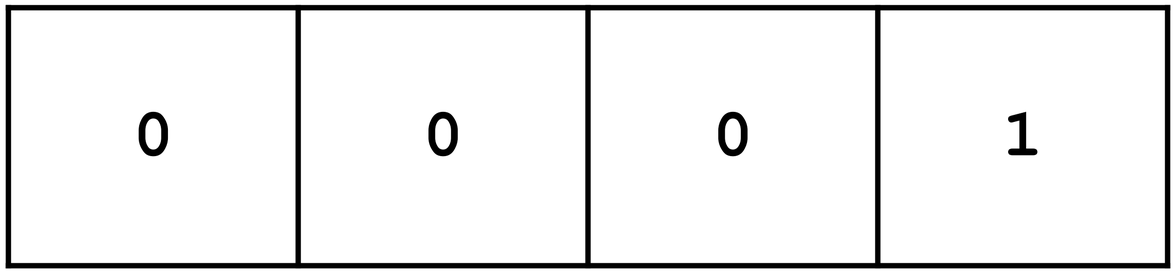
Negative Numbers

Better Idea: Two's Complement



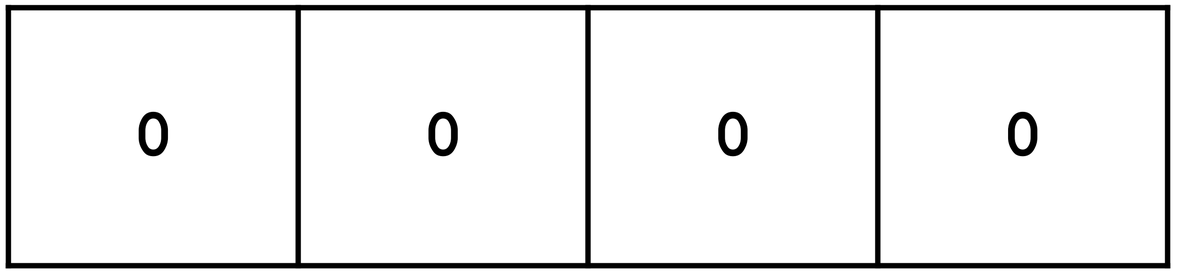
Negative Numbers

Better Idea: Two's Complement



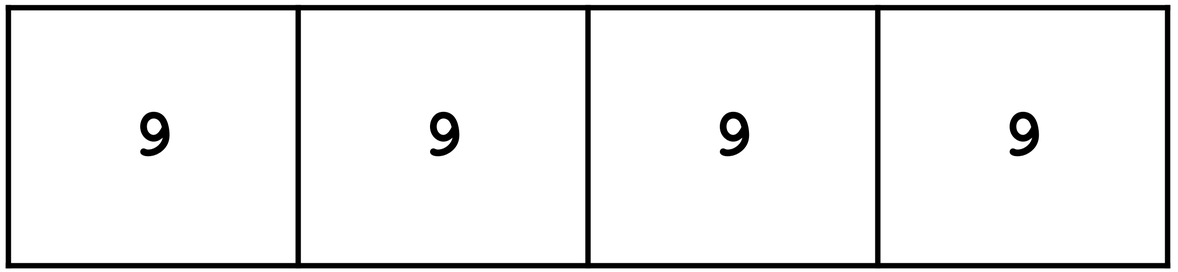
Negative Numbers

Better Idea: Two's Complement



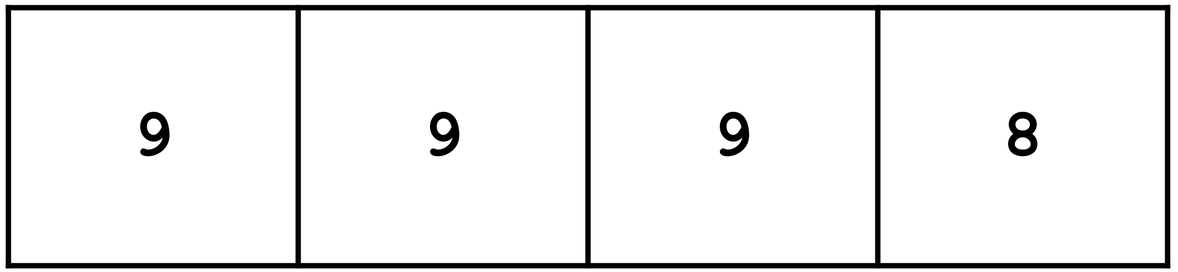
Negative Numbers

Better Idea: Two's Complement



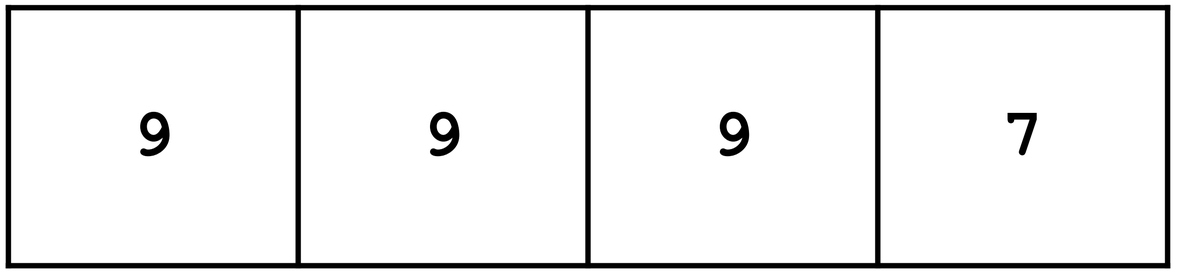
Negative Numbers

Better Idea: Two's Complement



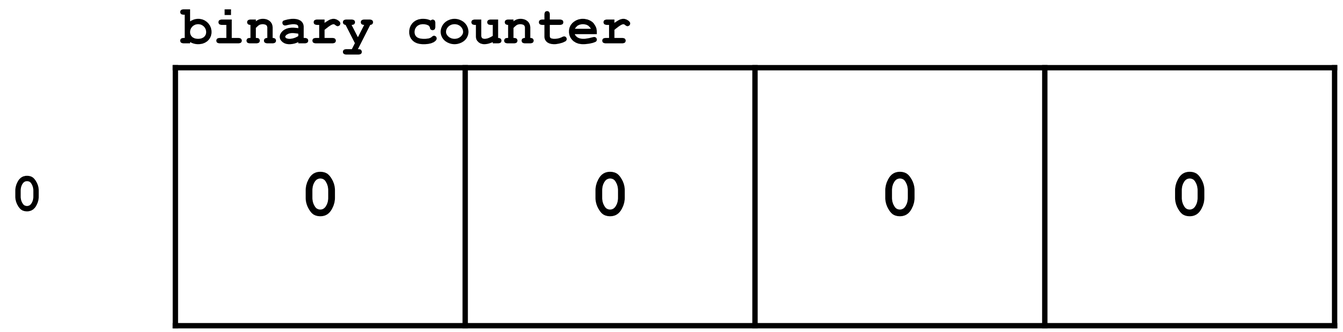
Negative Numbers

Better Idea: Two's Complement



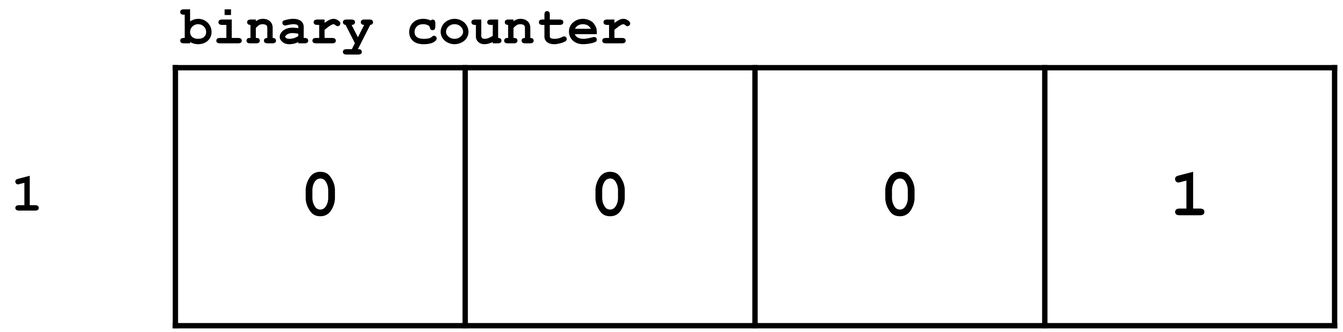
Negative Numbers

Better Idea: Two's Complement



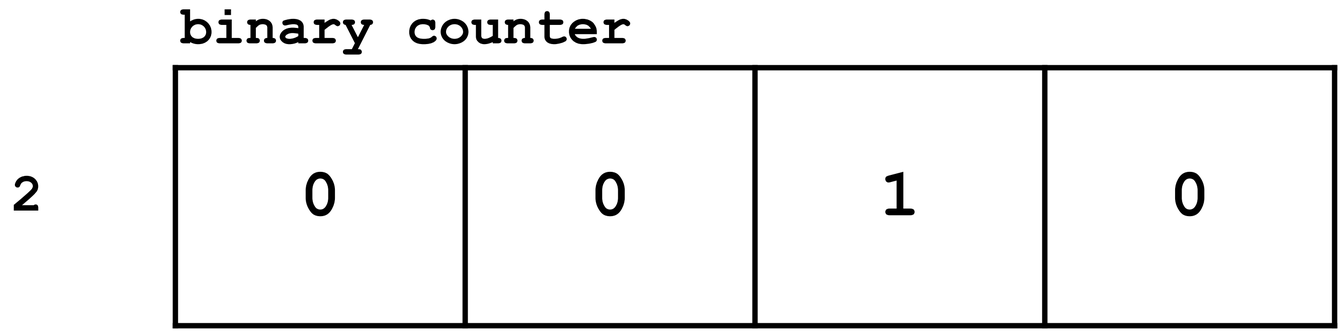
Negative Numbers

Better Idea: Two's Complement



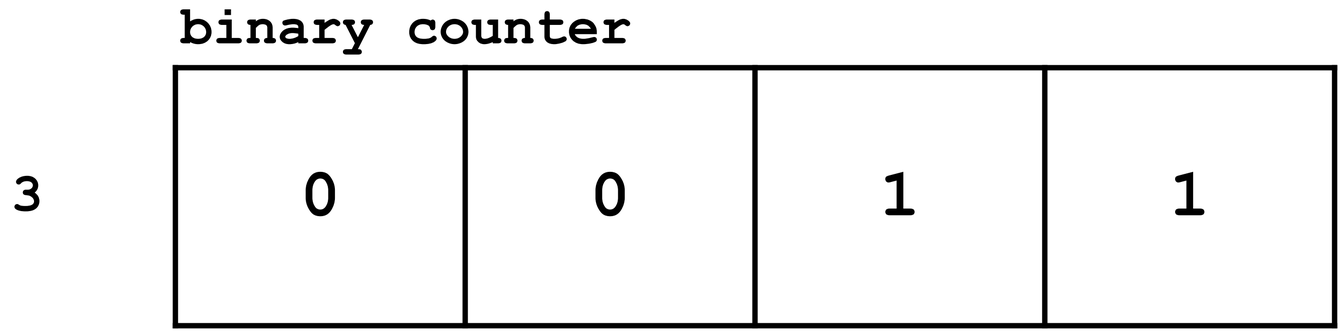
Negative Numbers

Better Idea: Two's Complement



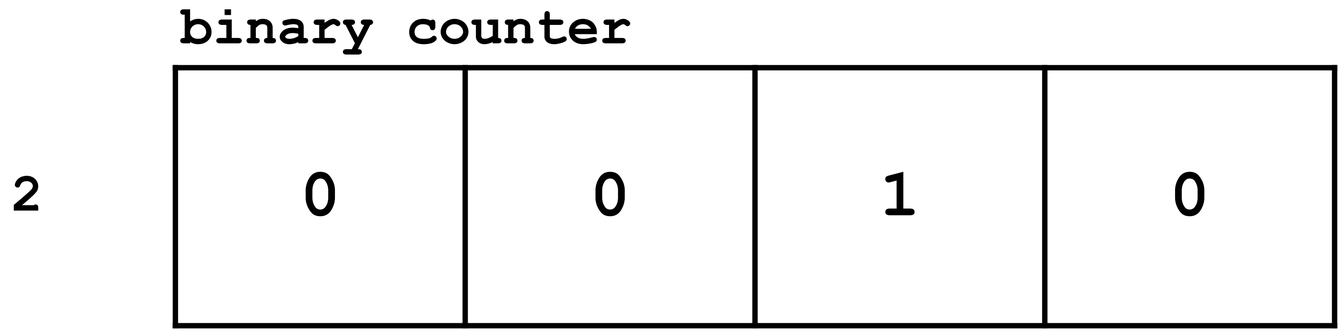
Negative Numbers

Better Idea: Two's Complement



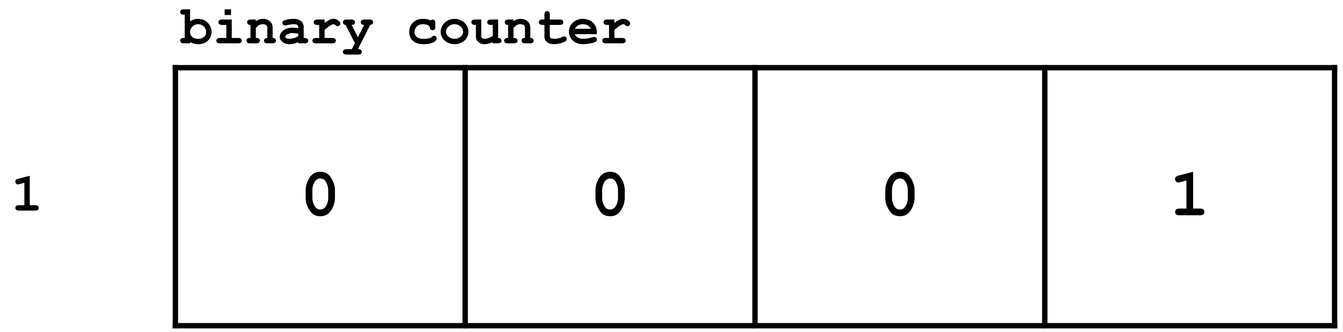
Negative Numbers

Better Idea: Two's Complement



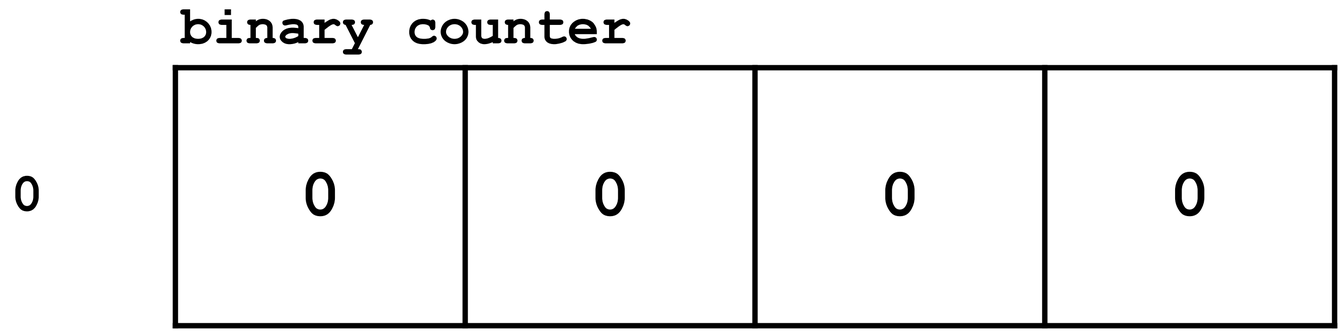
Negative Numbers

Better Idea: Two's Complement



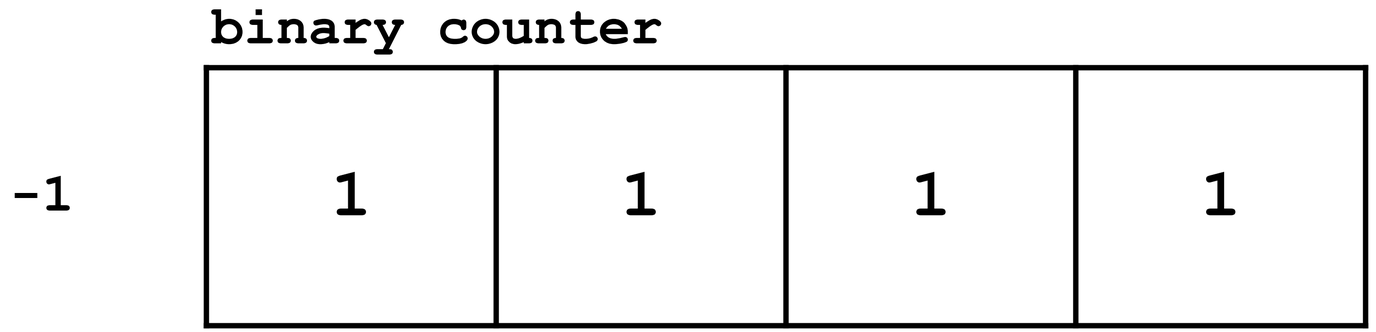
Negative Numbers

Better Idea: Two's Complement



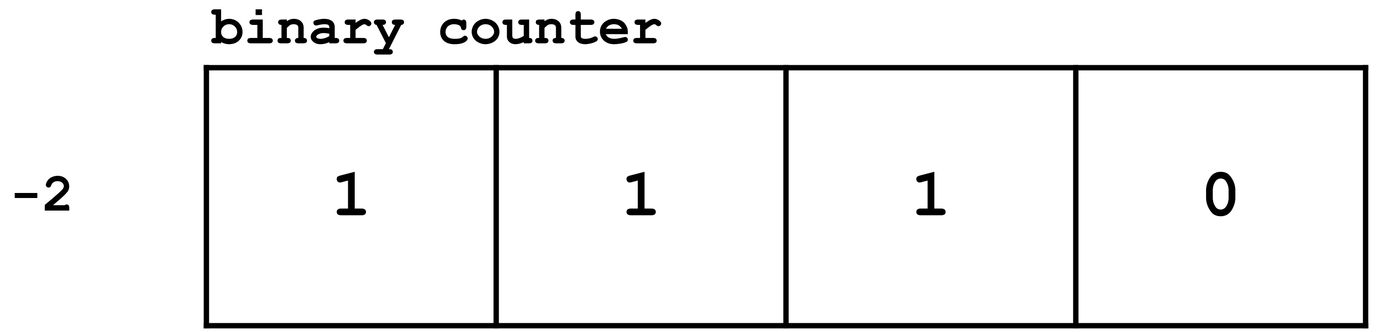
Negative Numbers

Better Idea: Two's Complement



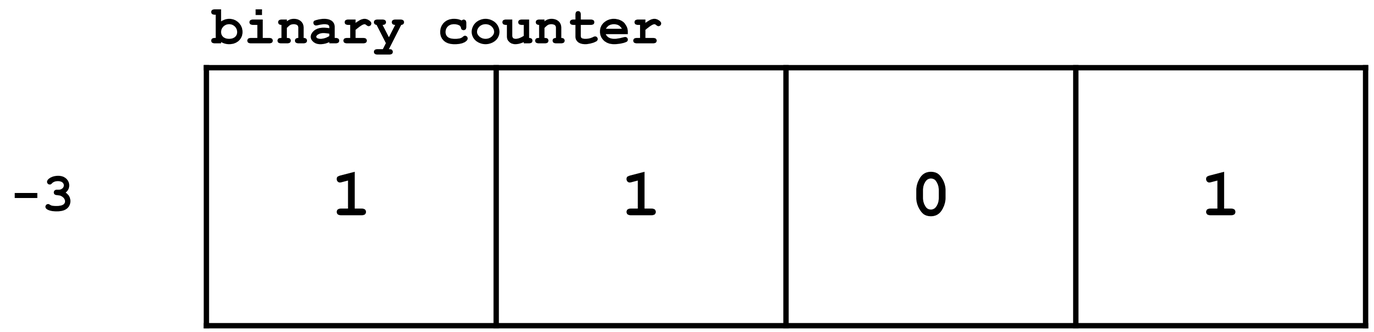
Negative Numbers

Better Idea: Two's Complement



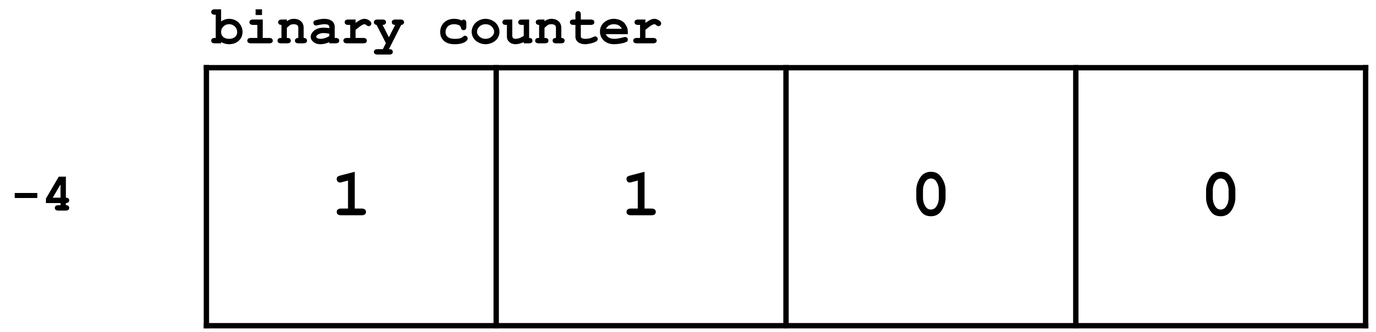
Negative Numbers

Better Idea: Two's Complement



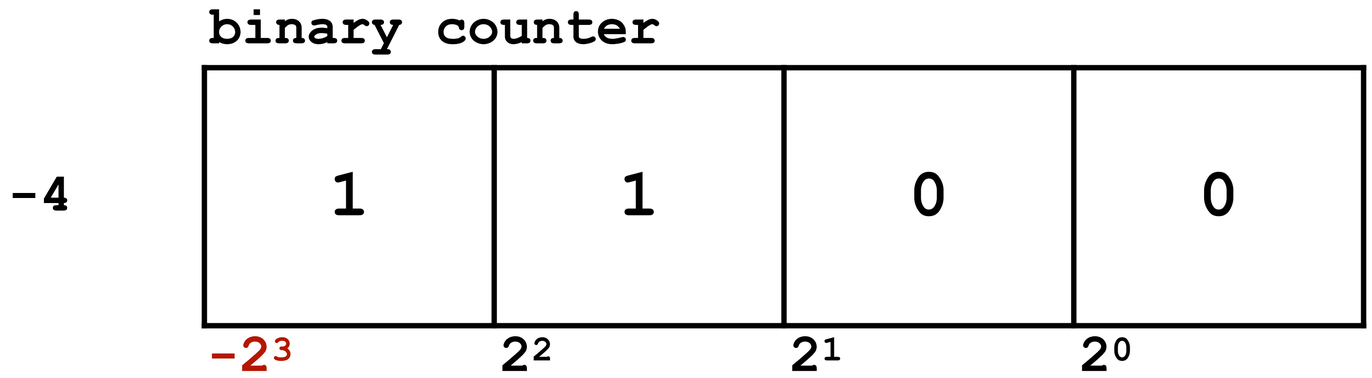
Negative Numbers

Better Idea: Two's Complement



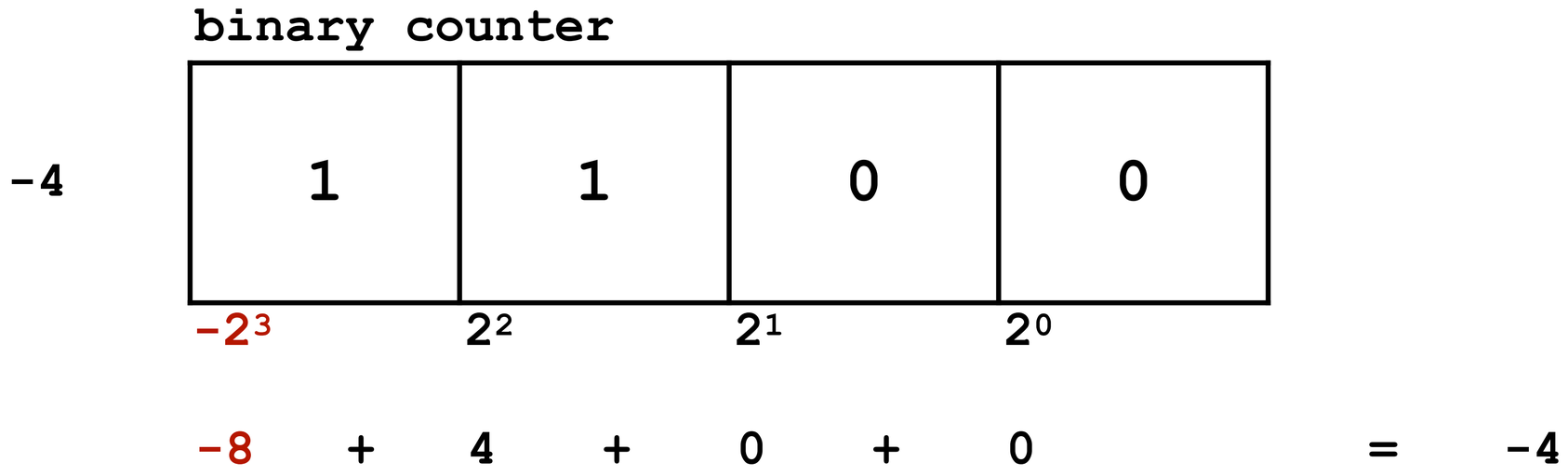
Negative Numbers

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

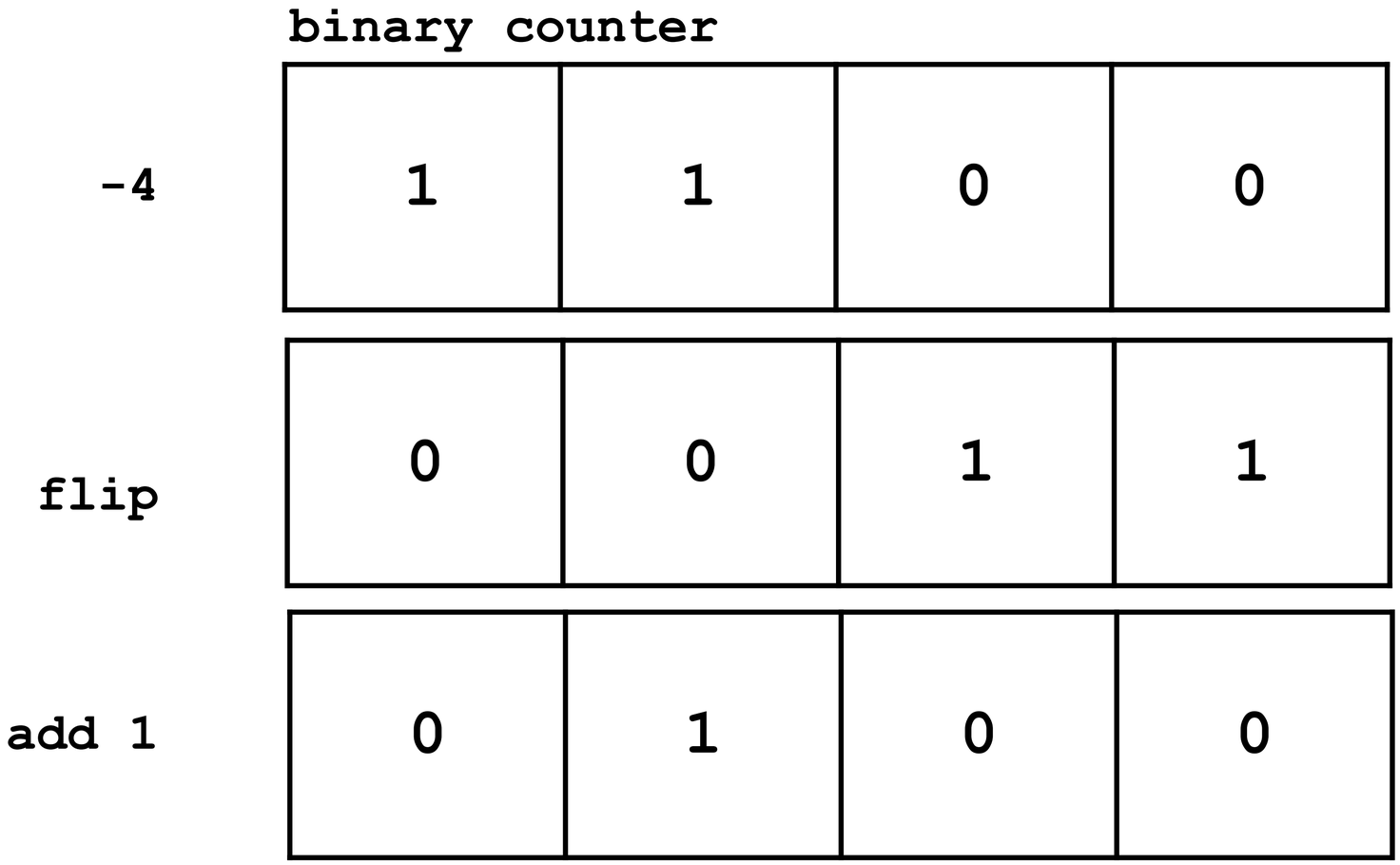
n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

```
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

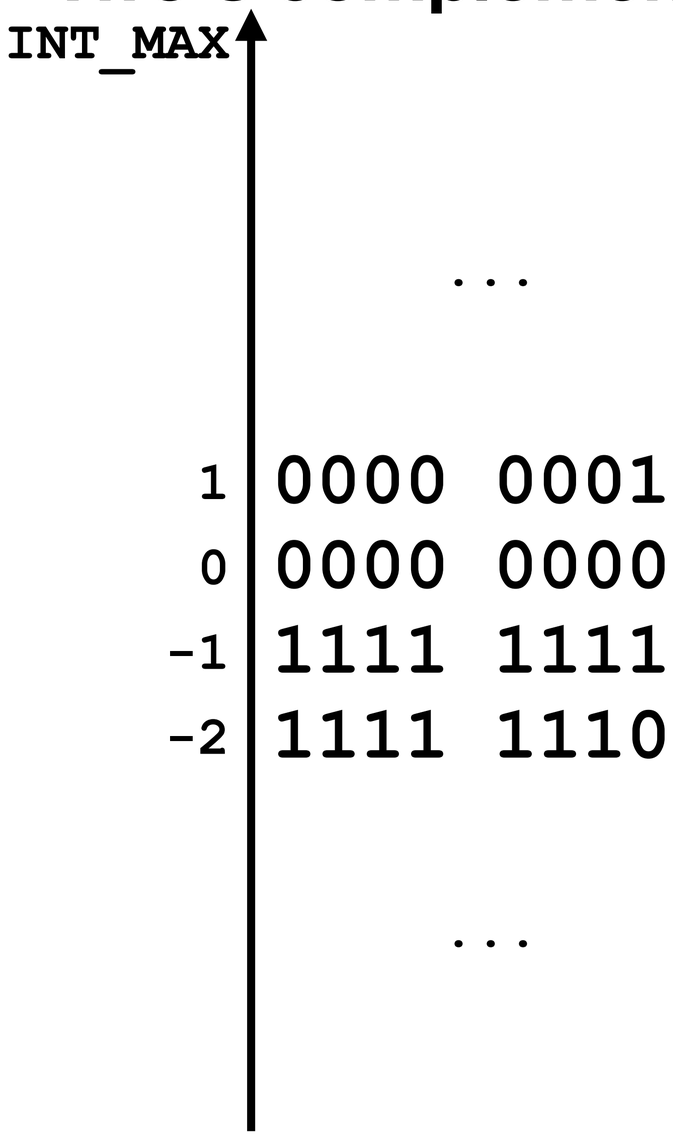
Negative Numbers

Two's complement: two ways



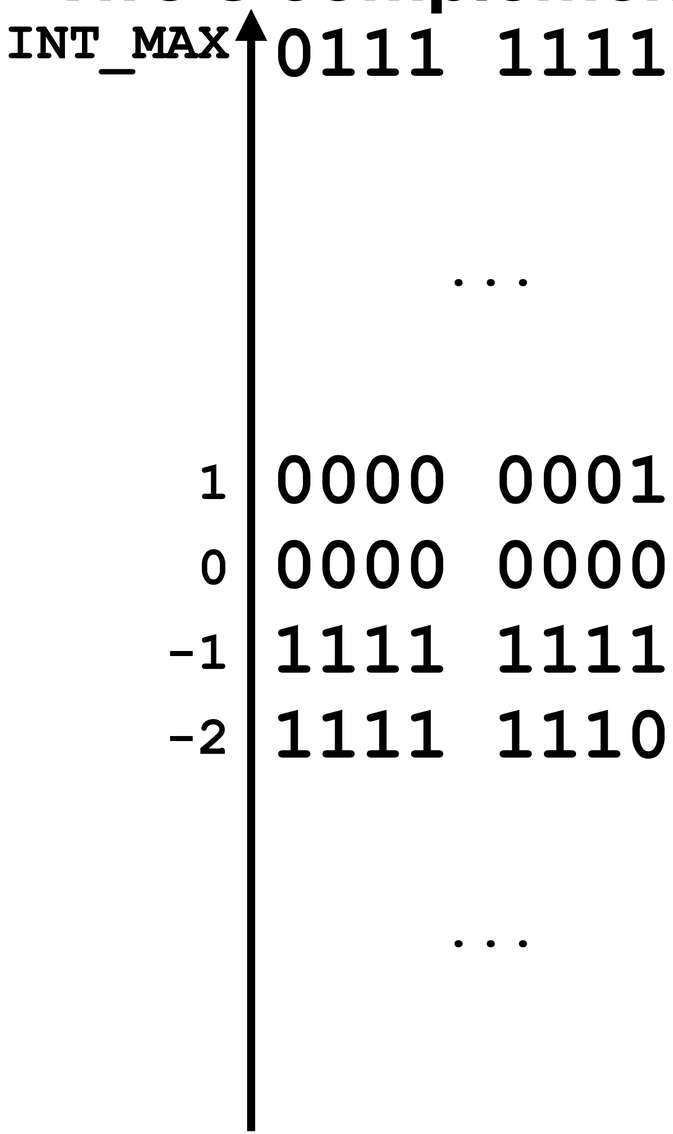
Negative Numbers

Two's complement: range



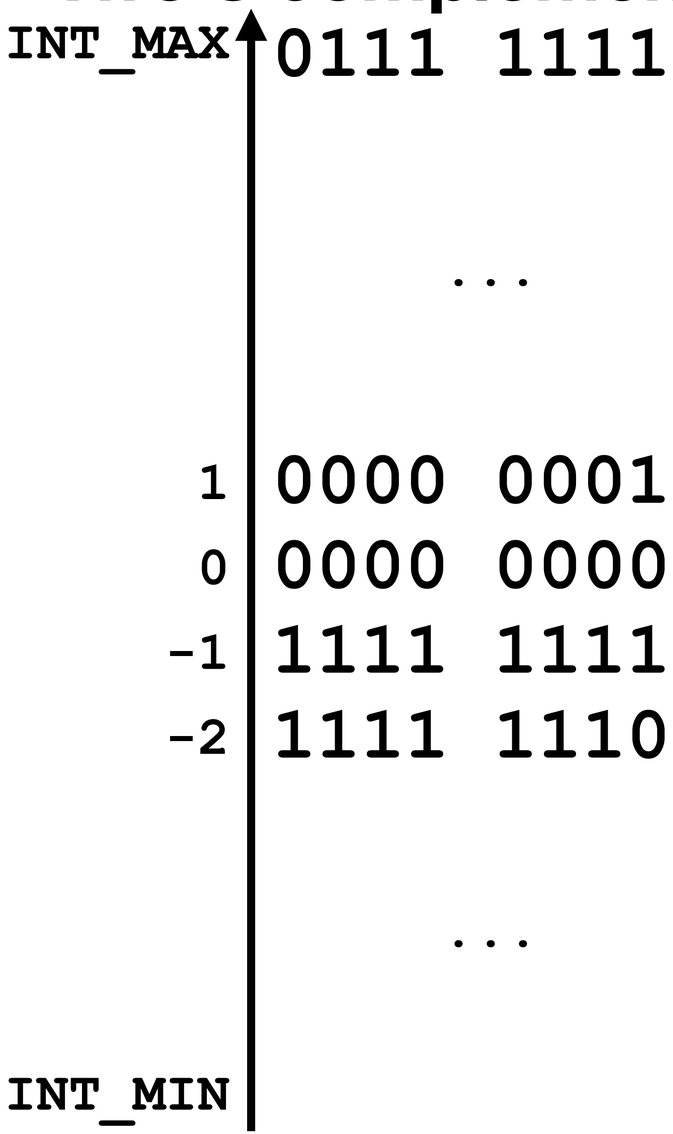
Negative Numbers

Two's complement: range



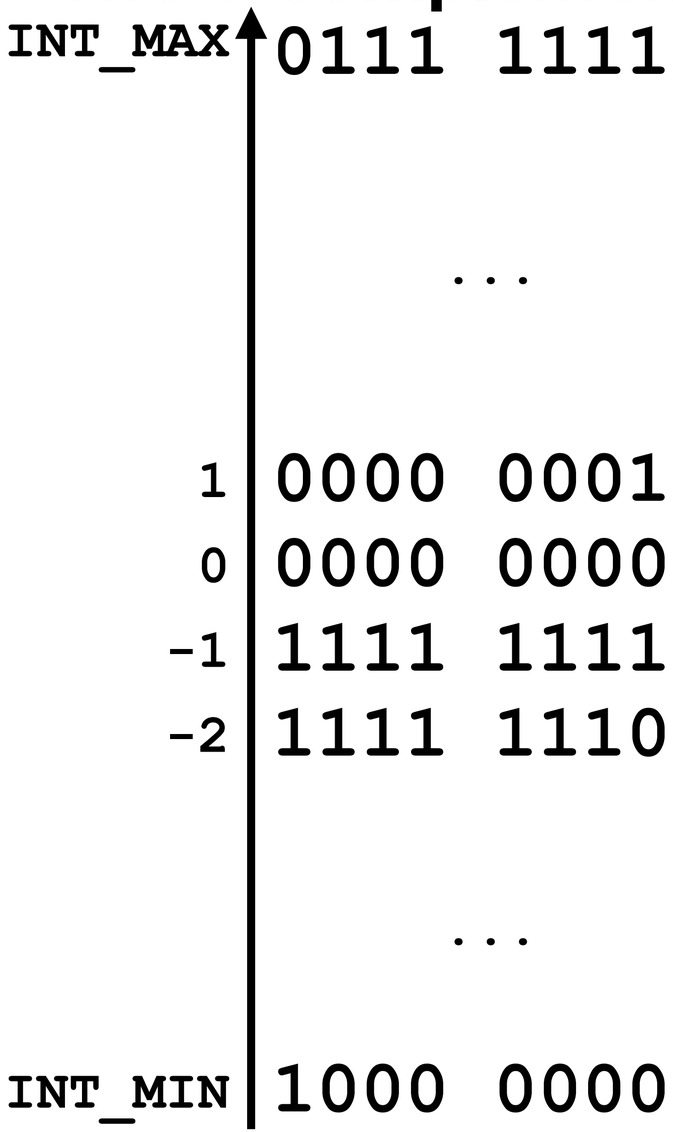
Negative Numbers

Two's complement: range



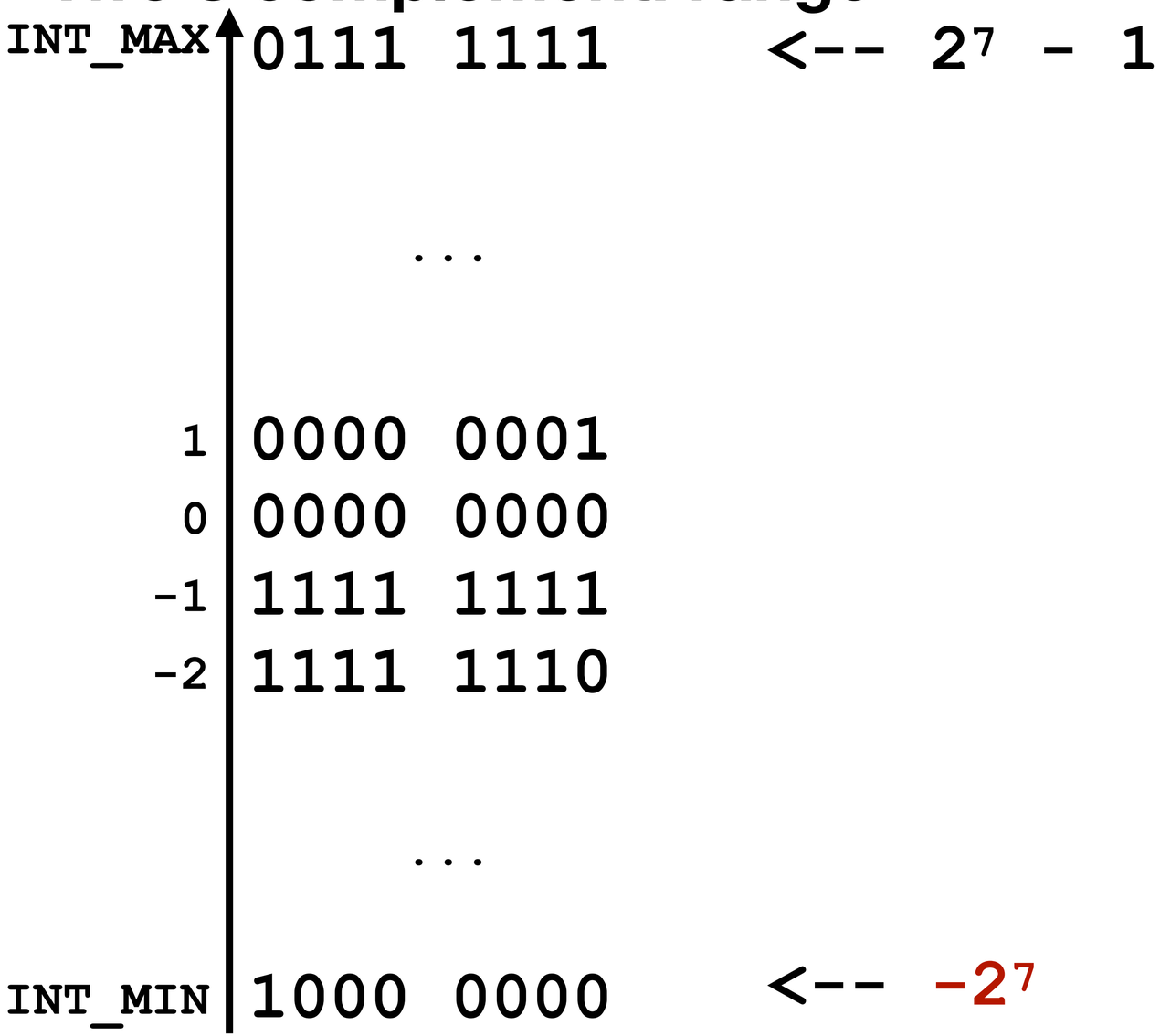
Negative Numbers

Two's complement: range

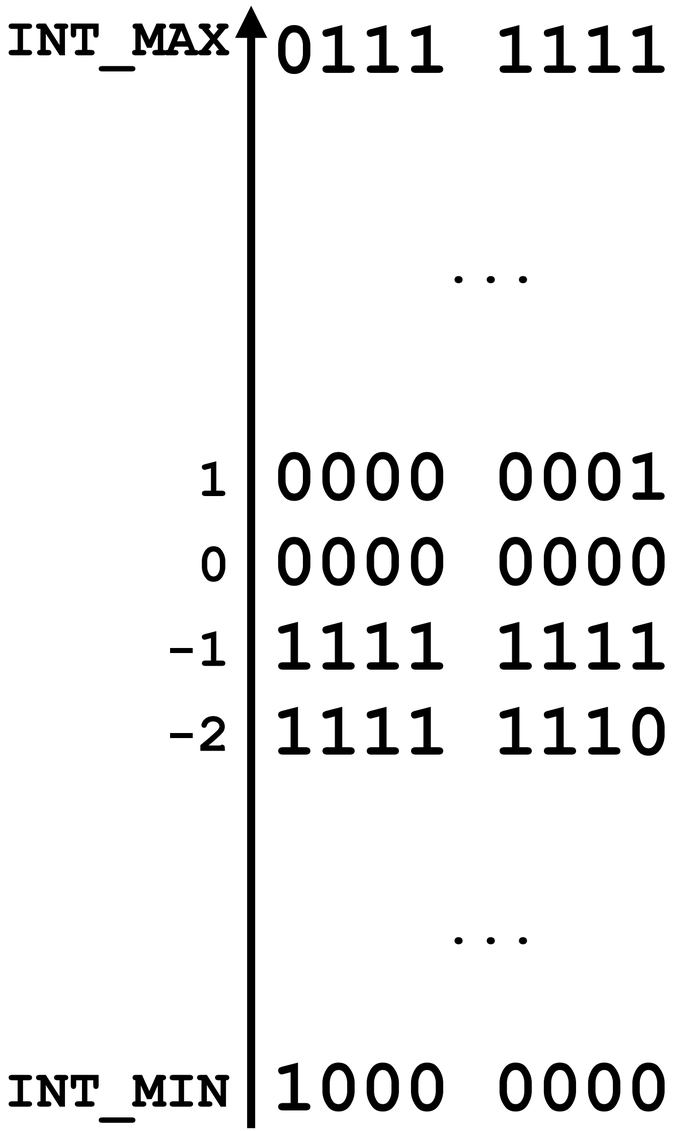


Negative Numbers

Two's complement: range

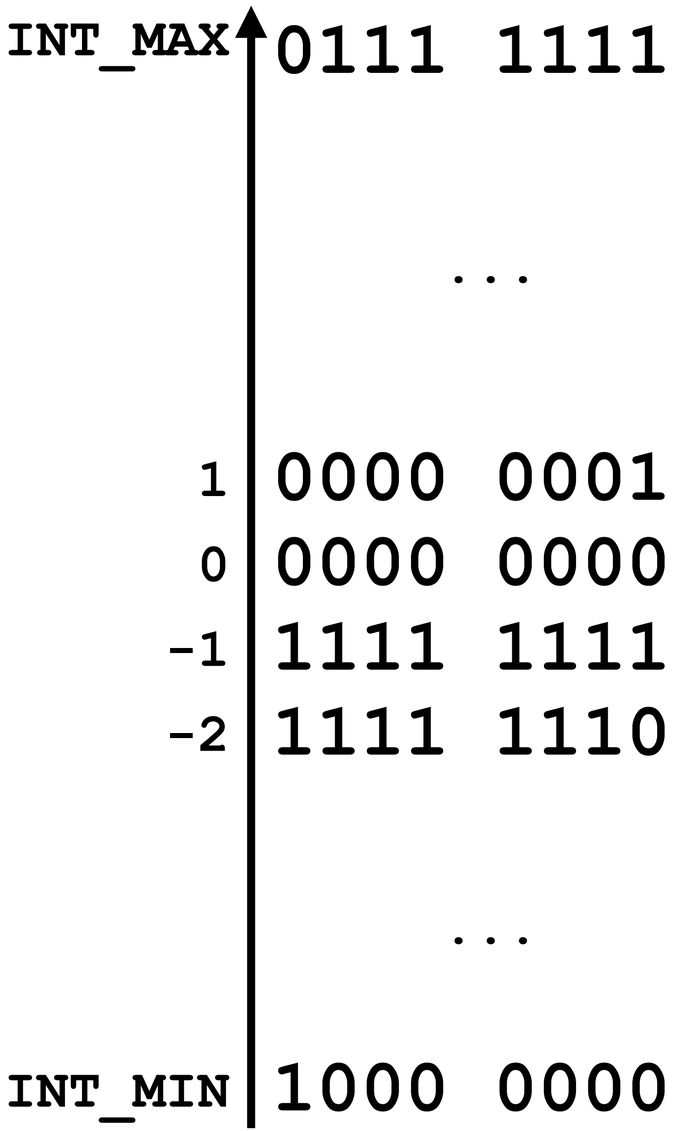


Signed Arithmetics



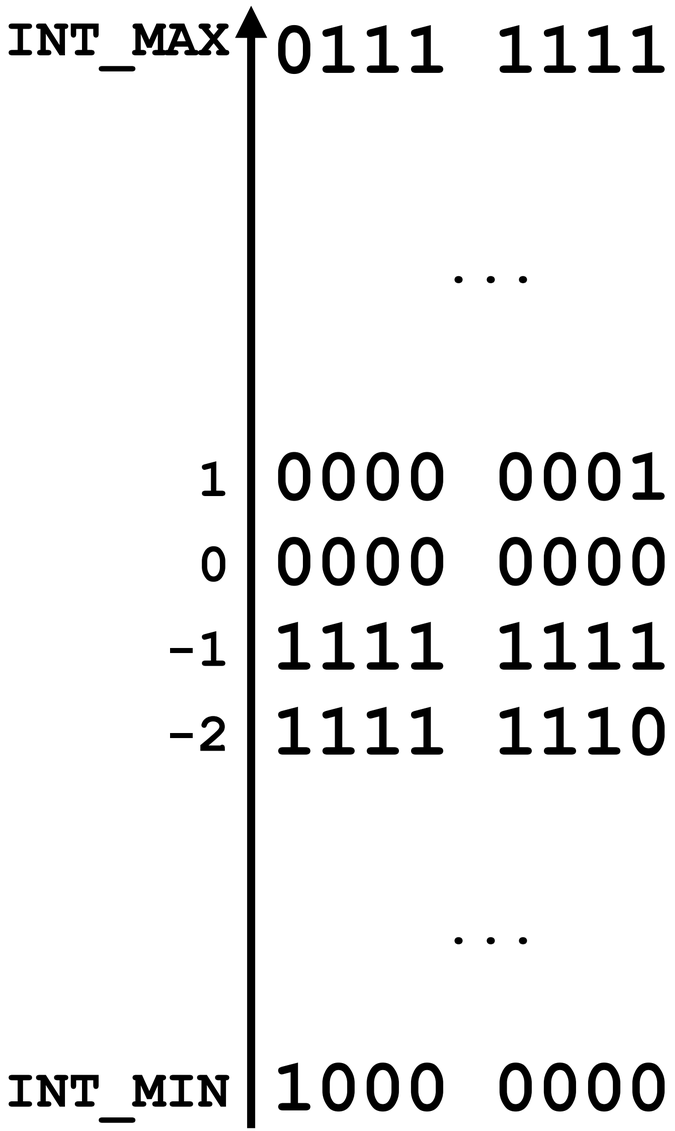

```
  1111 1110
+ 0000 0011
-----------
```

Signed Arithmetics



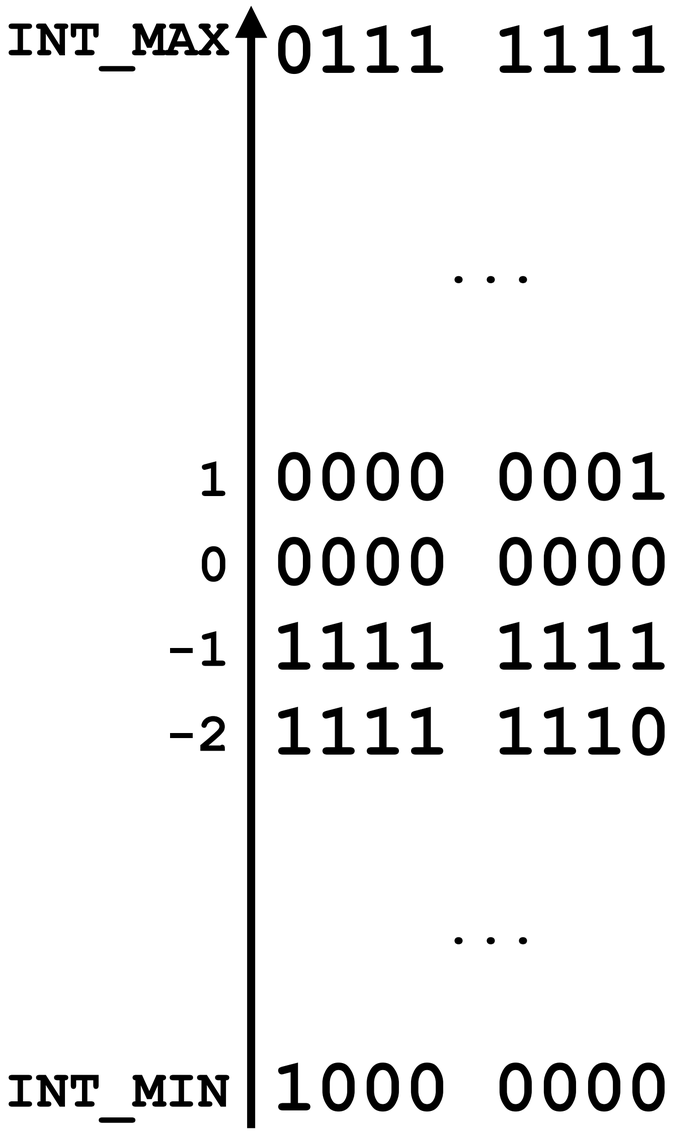

```
  1111 1110          -2
+ 0000 0011           3
-------------
```

Signed Arithmetics



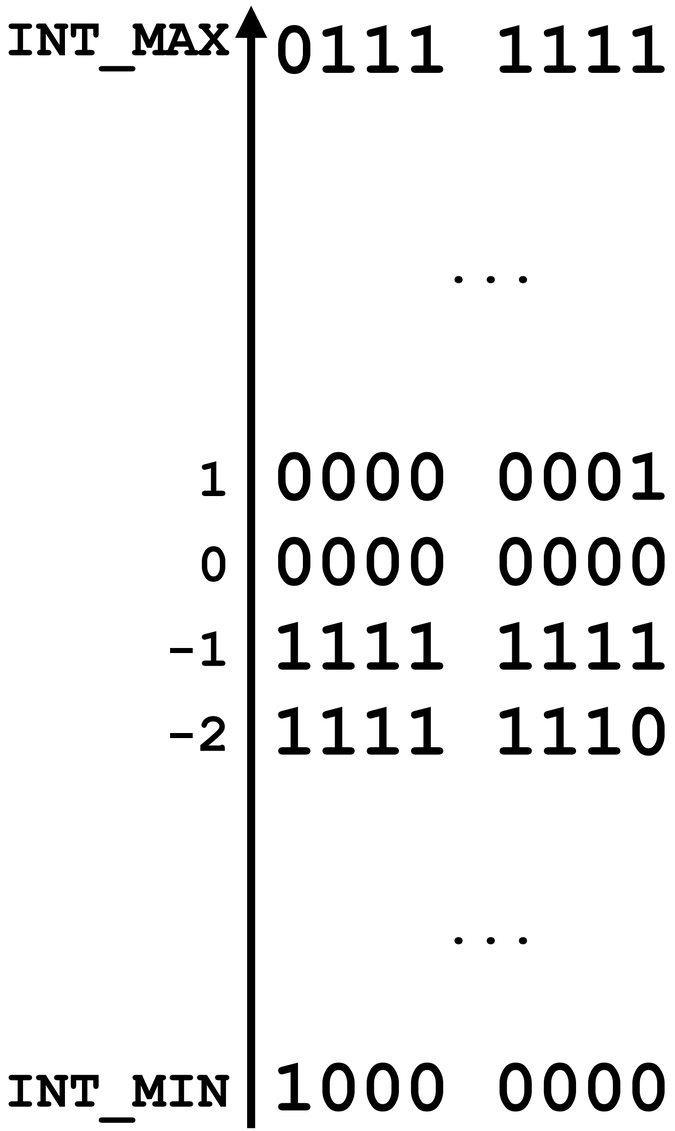

```
  1111 1110          -2
+ 0000 0011           3
-------------------------
             1
```

Signed Arithmetics




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

Signed Arithmetics



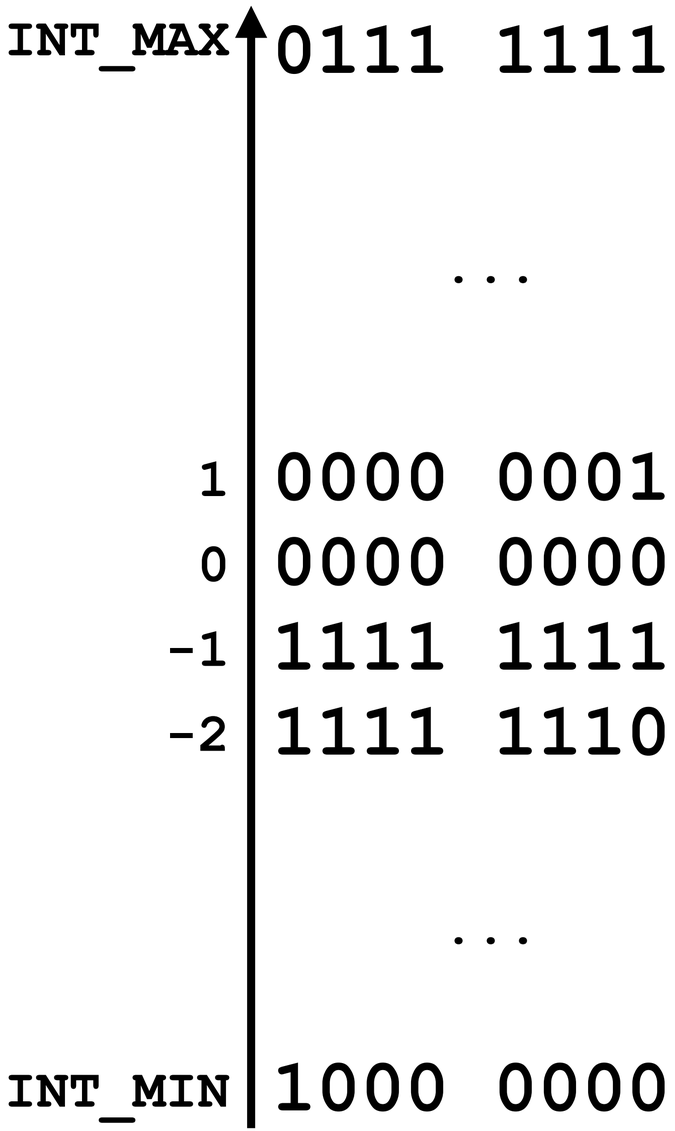

```
  1111 1110          -2
+ 0000 0011           3
-------------------------
           001
```

Signed Arithmetics

INT_MAX	0111 1111
	...
1	0000 0001
0	0000 0000
-1	1111 1111
-2	1111 1110
	...
INT_MIN	1000 0000

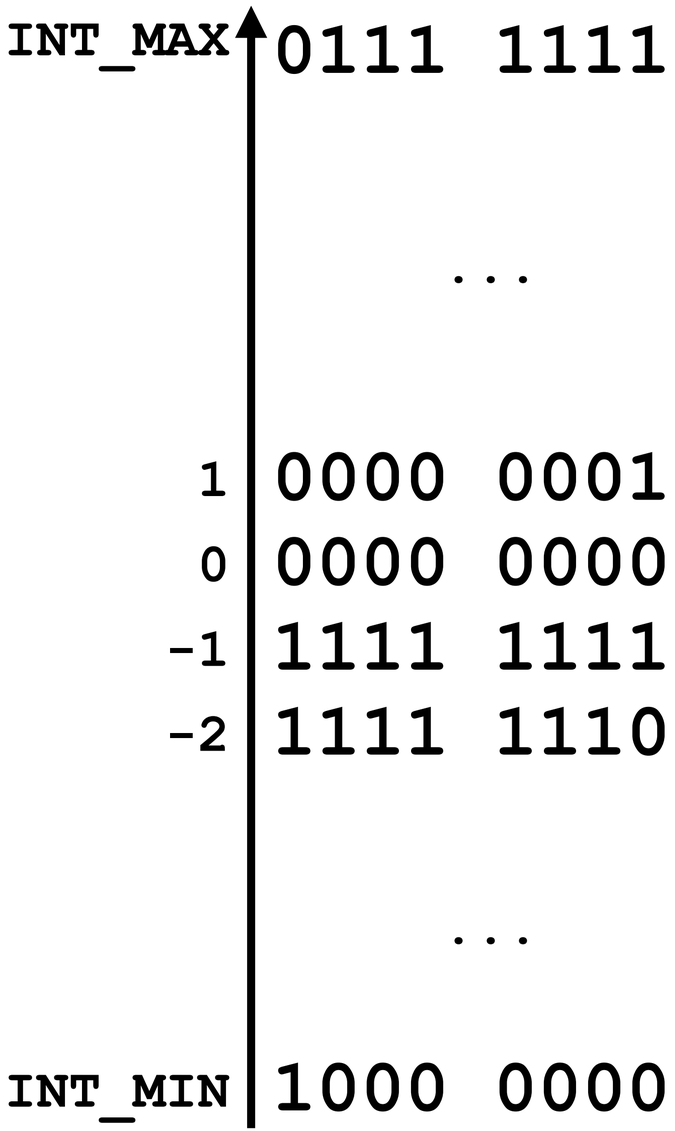
```
    1111 1110          -2
  + 0000 0011           3
        1   1 1
  -------------
         0001
```

Signed Arithmetics



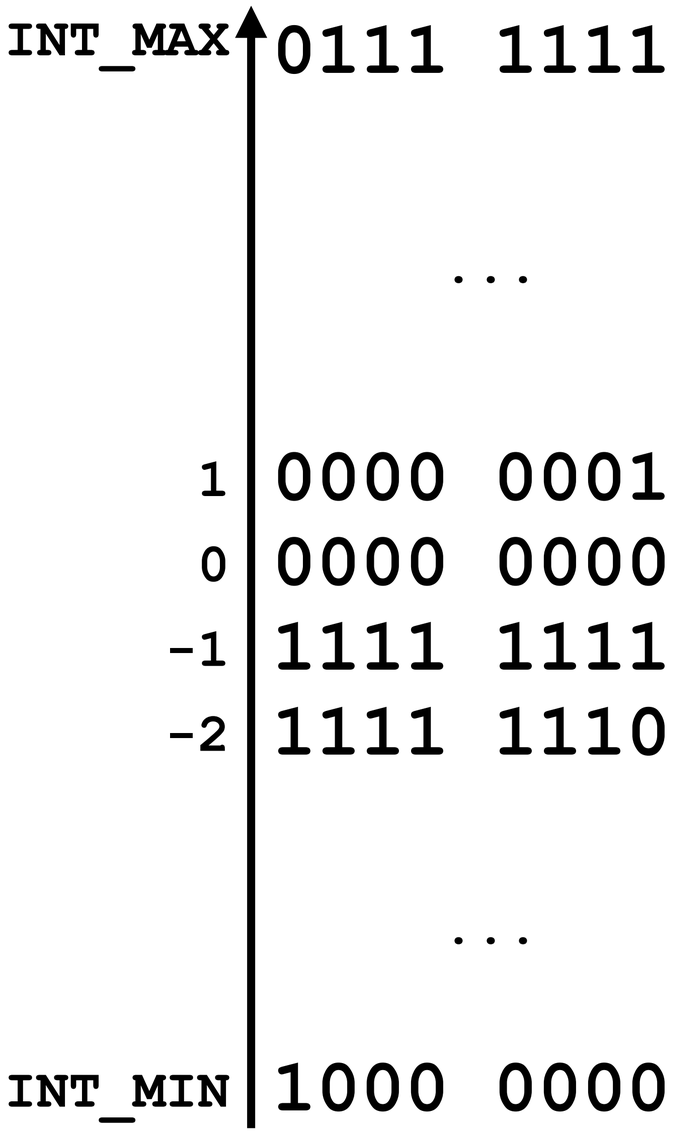

```
  1111 1110          -2
+ 0000 0011           3
  ¹¹  ¹¹
-------------------------
       0 0001
```

Signed Arithmetics



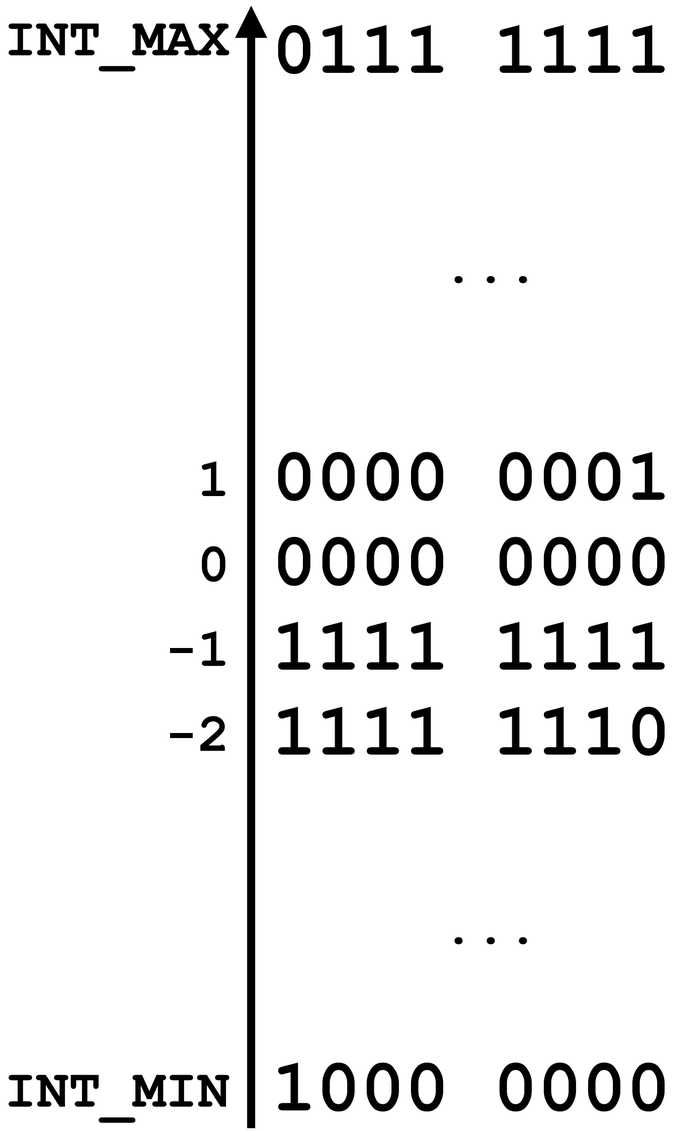

```
  1111 1110          -2
+ 0000 0011           3
  ^^^^ ^^^^
  0000 0001
```

Signed Arithmetics




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

Signed Arithmetics




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

Machine Arithmetics

Summary

- 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.

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

Bits

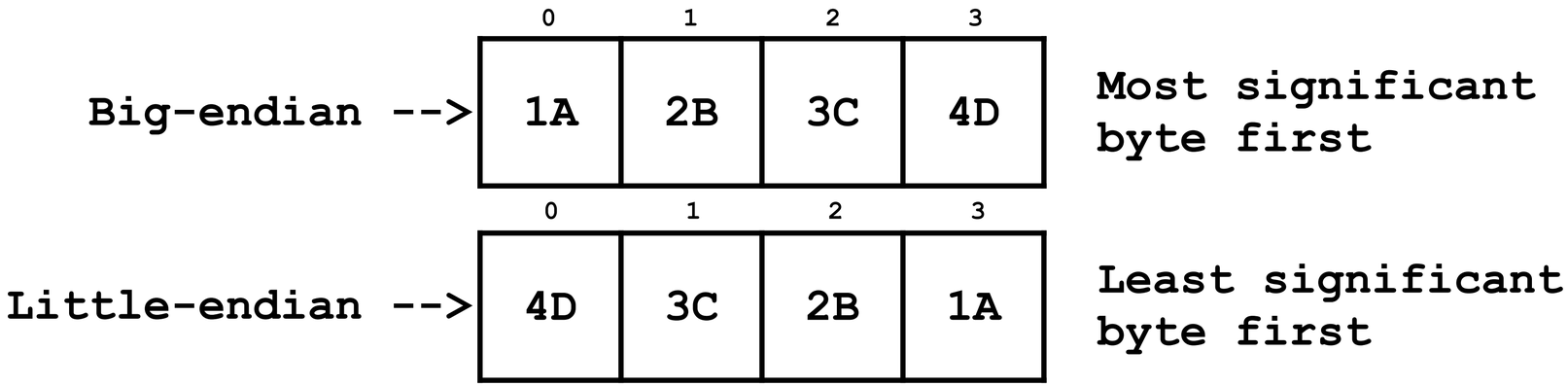
- A bit answers a yes/no question
  - Bits have no inherent meanings; to read encoding, need to know the intended interpretation
  - Any device that can be one of two states can encode a bit.
- Binary and hexadecimal numbers
  - $xyz_m = x \cdot m^2 + y \cdot m^1 + z \cdot m^0$
  - One hex digit corresponds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

Enum

Demo

Endian

- We think of an integer as one atomic value:
  - `int x = 0x1A2B3C4D;`
- But if an integer has 4 bytes and each byte is addressable, which of the 4 bytes is stored first?



Endian

- Is my machine little-endian or big-endian?
- Let's find out!

Endian

- Our machine is little-endian?????
- We usually write numbers in big-endian: 345 is three hundred and forty-five
- But there are some advantages for little-endian:
  - comparing two numbers of different length (long and int e.g.)
    - `4E3C2B1A`
    - `4E3C2B1A00000000`
  - addition, subtraction circuits work from low to high
  - etc.

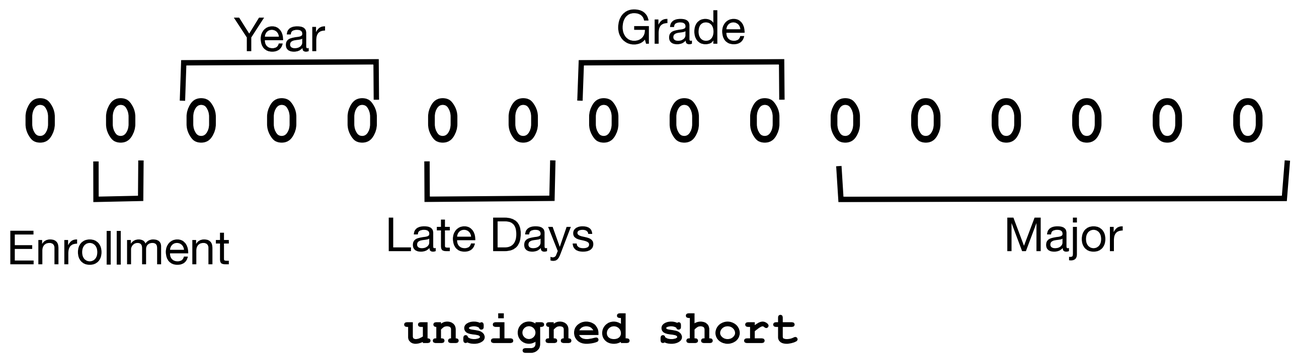
Endian

Does it matter?

- Mostly we don't care. Unless you do memory trickery, variables work as you would expect
- However, when we serialize data into byte sequences, you need to pay extra attention:
  - Writing a number to a file
  - Sending a number over a network
- You and the reader must agree on byte order
  - For this purpose, *network byte order* is defined for TCP/IP

Choices

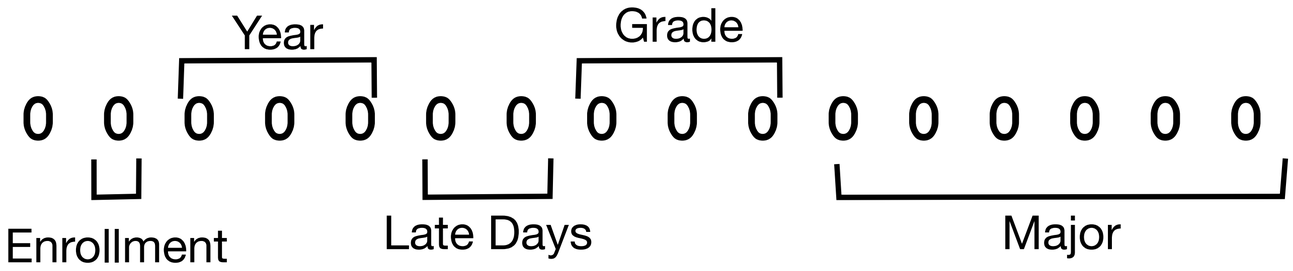
Interlude



- How do we assign bit patterns to majors?
- Anthropology - 0, Architectural Studies - 1, Art History - 2, ...
- One has to look up on a table to figure out which major this is :(

Choices

Interlude: `enum`

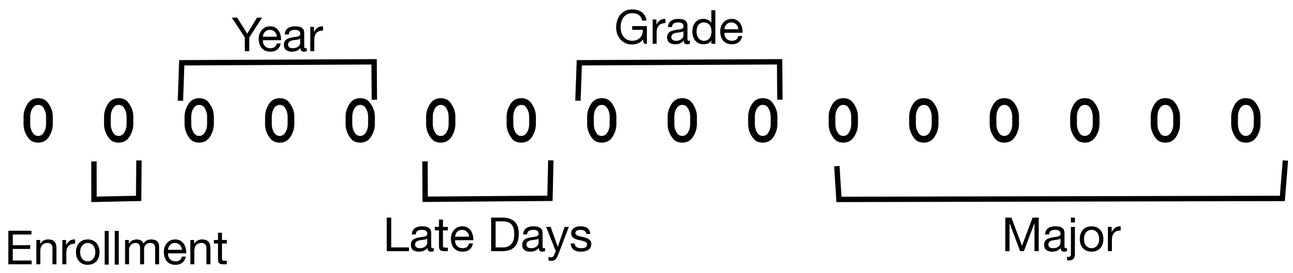



```
enum major {
        ANTHROPOLOGY,
        ARCHITECTURAL_STUDIES,
        ART_HISTORY,
        ASTRONOMY_ASTROPHYSICS,
        BIG_PROBLEMS,
        BIOLOGICAL_CHEMISTRY,
        ...
};
```

- Nothing fancy here: C just assigns an integer sequentially for each choice.
- Can use them as global constants
- `if (major == 2) { ... }`
- `if (major == ART_HISTORY) { ... }`

Choices

Interlude: `enum`



`unsigned short`

```
enum major {
        ANTHROPOLOGY,
        ARCHITECTURAL_STUDIES,
        ART_HISTORY,
        ASTRONOMY_ASTROPHYSICS,
        BIG_PROBLEMS,
        BIOLOGICAL_CHEMISTRY,
        ...
};
```

```
enum major student_major = STATISTICS;
```

```
enum major student_major = 3;
```

`clang` will *not* complaint about this. But this is bad style.

Choices

Interlude: `switch`

```
if (major == ANTHROPOLOGY) {
        ...
} else if (major == ARCHITECTURAL_STUDIES) {
        ...
} else if (major == ART_HISTORY) {
        ...
} else if (major == ASTRONOMY_ASTROPHYSICS) {
        ...
} else if (major == BIG_PROBLEMS) {
        ...
} ...
```

```
switch (major) {
case ANTHROPOLOGY:
        ...;
        break;
case ARCHITECTURAL_STUDIES:
        ...;
        break;
case ART_HISTORY:
        ...;
        break;
case BIOLOGICAL_CHEMISTRY:
        ...;
        break;
case BIG_PROBLEMS:
        ...;
        break;
default:
        break;
}
```

`break` signals the end of a case. Without break, C will execute the next case, *falling through* another case.

The default branch is run when the major matches none of the above cases.

Choices

Interlude: `switch`

- Why switch?
- Cleaner code
- More efficient than if ... else if ... chain:
  - C stores the branches in a table, and switch will jump to the branch instead of comparing one by one
  - `switch(x)`, `x` has to be an integer.
- `break` is critical! Forgetting the break is really difficult to debug.

Choices

Interlude: `switch`

- Demo! if we have time