Integer arithmetic

Unsigned integers

- Computers are made out of Boolean gates
- But we want to represent numbers other than 0 and 1
- How do we proceed?

- Consider Booleans as **binary digits** (*bits*)
- Group them together to form numbers in base 2

Base-10 numbers

In base 10 (decimal), we have 10 distinct digits: { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }

Using one digit, we can count to 9:

```
0    1    2    3    4    5    6    7    8    9
```

Then we need more digits:

```
10   11   12   13   14   15   16   17   18   19
20   21   22   23   ...
```

If we wanted to count from 0 to 9999 (say, to represent a date), we may decide to use 4 digits:

```
0000 0001 0002 0003 0004 0005 0006 0007 0008 0009
0010 0011 0012 0013 ...
```

Base-10 numbers

1984 = ?

$$
\begin{array}{cccccccc}
 & 1 & & 9 & & 8 & & 4 \\
= & 1 \times 1000 & + & 9 \times 100 & + & 8 \times 10 & + & 4 \\
= & 1 \times 10^3 & + & 9 \times 10^2 & + & 8 \times 10^1 & + & 4 \times 10^0
\end{array}
$$

Base-2 numbers

In base 2 (binary), we have 2 distinct digits: { 0, 1 }

Using one digit, we can count to 1:

```
   0    1
```

Then we need more digits:

```
   10   11  100  101  110  111 1000 1001 ...
```

If we wanted to count from 0 to 15, we may decide to use 4 digits:

```
0000 0001 0010 0011 0100 0101 0110 0111
1000 1001 1010 1011 1100 1101 1110 1111
```

Base-2 numbers

1001b = ?

$$\begin{array}{ccccccc} & 1 & & 0 & & 0 & & 1 \\ = & 1 \times 8 & + & 0 \times 4 & + & 0 \times 2 & + & 1 \\ = & 1 \times 2^3 & + & 0 \times 2^2 & + & 0 \times 2^1 & + & 1 \times 2^0 \end{array}$$

$= 9$

Note:

- rightmost / least-significant bit is called bit 0
- leftmost / most-significant bit is called bit $n - 1$

Fixed bit width

- For any integer, we must always know how many digits (bits) it has.
- Typically, this number of bits is fixed in our code.

bits	a.k.a.	C type	other C type
8	byte†	`uint8_t`	`unsigned char`†
32		`uint32_t`	`unsigned int` (Windows, Linux, BSD, macOS)
64		`uint64_t`	`unsigned long` (Linux, BSD, macOS)
			`unsigned long long` (Windows)

† = on almost all contemporary platforms as of 2024

Integers in hardware and in programming languages

- Most computers† support 8, 16, 32 and 64-bit arithmetic natively (i.e., operations are fast)
- Arithmetic can be performed with arbitrary-sized integers by implementing the operations in software (hence much slower).
- In C, every integer type has a specific size.
- In C, arbitrary-sized integers are not supported by the language (they require using specific libraries).
- In Python, all integers can have arbitrary sizes (with a large performance penalty, especially when exceeding 32 bits)

bits	largest integer $= 2^{\text{bits}} - 1$	(approx.)
8	255	
16	65,535	
32	4,294,967,295	4 billions
64	18,446,744,073,709,551,615	2.10^{19}
128	340,282,366,920,938,463,463,374,607,431,768,211,455	3.10^{38}

$$1 \text{ decimal digit} = \log_2 10 \text{ bits} \simeq 3.3219 \text{ bits}$$

Operations with integers

Essentially the same a schoolbook operations:

$$
\begin{array}{ccccccccc}
 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\
+ & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
= & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0
\end{array}
$$

Just like in school:

- addition and subtraction are straightforward
- multiplication is more complex
- division is much more complex

Signed integers

- How do we represent negative numbers?
- Impossible with previous approach.
- Solution 1:
    - “**sign-magnitude**”: sacrifice one bit, which we reserve to store the sign.
    - Drawback: zero has two representations (+0 and -0)
    - Drawback: Boolean logic for + and - must handle many cases
- Solution 2:
    - “**one’s complement**”: reserve top bit for the sign, must be zero for a positive number
    - when a number is negative, takes its (positive) opposite and flip all bits
    - Drawback: zero has two representations (+0 and -0)
    - Drawback: Boolean logic for + and - is simpler but still affected

- Solution 3 (all current computers†):
    - “**two’s complement**”: when a n-bit number x is negative, represent it the same as the unsigned number $2^n - |x|$.
    - The top bit is 1 for negative numbers.
    - Drawback: Flipping sign slightly more complex (flip all bits then add one).
    - Advantage: zero has a single representation
    - Advantage: Boolean logic for + and - is **the same** as for unsigned integers

Two's complement

- Given a single n-bit pattern,
  - let u be its unsigned value
  - let s be its signed value,
- If bit $(n-1) = $ 0, then:
  - $s := u$
- If bit $(n-1) = $ 1, then:
  - $s := u - 2^n$

4-bit example:

bit pattern	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
unsigned u	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
signed s	0	1	2	3	4	5	6	7	-8	-7	-6	-5	-4	-3	-2	-1

- bit $(n-1) = 0 \quad \Rightarrow \quad s = u$
- bit $(n-1) = 1 \quad \Rightarrow \quad s = u - 2^n$

In general:

bit pattern	00 … 0		01 … 1	10 … 0		11 … 1
unsigned u	0	$\ldots$	$(2^{n-1}) - 1$	(2^{n-1})	$\ldots$	$(2^n) - 1$
signed s	0	$\ldots$	$(2^{n-1}) - 1$	$-(2^{n-1})$	$\ldots$	-1

- Unsigned: $u \in \{0, \ldots, (2^n) - 1\}$
- Signed: $s \in \{-(2^{n-1}), \ldots, -1, 0, \ldots (2^{n-1}) - 1\}$

n **bits**	$-2^{\text{bits}-1}$ **(min)**	$2^{\text{bits}-1} - 1$ **(max)**
8	-128	127
16	-32768	32767
32	-2,147,483,648	2,147,483,647
64	$\simeq -9.10^{18}$	$\simeq 9.10^{18}$
128	$\simeq -2.10^{38}$	$\simeq 2.10^{38}$

Conversely:

- if $s \geq 0$ $\qquad s \in \{0, \ldots, (2^n) - 1\}$
  - represent with bit pattern of $u = s$.
- if $s < 0$ $\qquad s \in \{-(2^{n-1}), \ldots, (2^{n-1}) - 1\}$
  - represent with bit pattern of $u = 2^n - |s|$.
- if $s \notin \{-(2^{n-1}), \ldots, (2^{n-1}) - 1\}$
  - cannot represent, need larger n

Sign extension

Let us represent $s = -5$ in n-bit signed binary (two's complement):

$u = 2^n - |\, s \,| = 2^n - 5$

n	s	u	bit pattern
4	-5	11	1011
5	-5	27	11011
6	-5	59	111011
7	-5	123	1111011
8	-5	251	11111011
9	-5	507	111111011
10	-5	1019	1111111011
11	-5	2043	11111111011
12	-5	4091	111111111011

Increasing the number of bits

To convert an n-bit number to an $(n+k)$-bit number $(k \geq 0)$**:**

- Unsigned:
  - Additional high-order (leftmost) bits are set to zero
- Signed ("sign extension"):
  - Additional high-order (leftmost) bits are set to the value of bit $(n-1)$

Overflow

Q: What happens if we run this?

```c
unsigned char a = 255;
unsigned char b = 1;
unsigned char x = a + b;
```

```c
signed char a = 127;
signed char b = 1;
signed char x = a + b;
```

```c
unsigned char a = 1;
unsigned char b = 2;
unsigned char x = a - b;
```

```c
signed char a = -128;
signed char b = 1;
signed char x = a - b;
```

A: It’s complicated!

We will dedicate an entire chapter to this.

Base 16

Hexadecimal digits: `0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f`

binary	hexadecimal	decimal
0000 0000	00	0
0000 0001	01	1
0000 0010	02	2
0000 0011	03	3
0000 0100	04	4
0000 0101	05	5
0000 0110	06	6
0000 0111	07	7
0000 0000	08	8
0000 1001	09	9
0000 1010	0a	10
0000 1011	0b	11
0000 1100	0c	12
0000 1101	0d	13
0000 1110	0e	14
0000 1111	0f	15

binary	hexadecimal	decimal
0001 0000	10	16
0001 0001	11	17
0001 0010	12	18
0001 0011	13	19
0001 0100	14	20
0001 0101	15	21
0001 0110	16	22
0001 0111	17	23
0001 0000	18	24
0001 1001	19	25
0001 1010	1a	26
0001 1011	1b	27
0001 1100	1c	28
0001 1101	1d	29
0001 1110	1e	30
0001 1111	1f	31

binary	hexadecimal	decimal
0010 0000	20	32
0010 0001	21	33
0010 0010	22	34
0010 0011	23	35
0010 0100	24	36
0010 0101	25	37
0010 0110	26	38
0010 0111	27	39
0010 0000	28	40
0010 1001	29	41
0010 1010	2a	42
0010 1011	2b	43
0010 1100	2c	44
0010 1101	2d	45
0010 1110	2e	46
0010 1111	2f	47

- Pros:
  - Directly maps to binary numbers:

    hex `12f3` = binary `0001 0010 1111 0011`
  - More compact than binary
  - Directly maps to bytes:

    two hex digits = one byte
- Cons:
  - Not human-friendly (esp. for arithmetic)

Characters and text

Q: How do we map bit patterns to characters in order to form text?

- Many standards
- Some similaritites
- Some incompatibilities

ASCII (1963-)

- American Standard Code for Information Interchange
- Each character stored stored in 1 byte (8 bits, 256 possible characters)
- 128 standardized characters
- Many derivatives specify the remaining 128

```
                  |                 2nd hex digit
                  |      0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
              ----|-----------------------------------------------------
                0 |                                  \t \n       \r
                1 |
                2 |         !  "  #  $  %  &  '  (  )  *  +  ,  -  .  /
                3 |      0  1  2  3  4  5  6  7  8  9  :  ;  <  =  >  ?
1st hex digit:  4 |      @  A  B  C  D  E  F  G  H  I  J  K  L  M  N  O
                5 |      P  Q  R  S  T  U  V  W  X  Y  Z  [  \  ]  ^  _
                6 |      `  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o
                7 |      p  q  r  s  t  u  v  w  x  y  z  {  |  }  ~
```

Unicode (1988-)

- Associates “code points” (roughly, characters) to integers
- Up to 1,112,064 code points (currently 149,813 assigned)
- First 128 code points coincide with ASCII
- Multiple possible encodings into bytes (“transmission formats”):
  - UTF-8
    - First 128 code points encoded into a single byte (backward compatible with ASCII)
    - Sets most significant bit (bit 7) to 1 to signify “more bytes needed”
    - Up to 4 bytes per code point
    - Default on BSD, iOS/MacOS, Android/Linux and for most internet communications
  - UTF-16
    - Code points are encoded by either two or four bytes
    - Default on Windows, for Java code, and for SMS

Unicode (1988-)

- Aims at encoding all languages:
  - including extinct ones
  - left-to-right, right-to-left or vertical
  - and more (emojis 🤷)
- Some “characters” require multiple code points (flag emojis, skin tone modifiers)
- What is even a “character”? (code point, glpyh, grapheme, cluster)
- Unicode is extremely complicated
- Latest version (v15.1.0, 2023-09-12) specification is 1,060 pages