Lecture 2
Ints

15-122: Principles of Imperative Computation (Spring 2020)
Frank Pfenning

Two fundamental types in almost any programming language are booleans and integers. Booleans are comparatively straightforward: they have two possible values (`true` and `false`) and conditionals to test boolean values. We will return to their properties in a later lecture.

Integers $\ldots, -2, -1, 0, 1, 2, \ldots$ are considerably more complex, because there are infinitely many of them. Because memory is finite, only a finite subrange of them can be represented in computers. In this lecture we discuss how integers are represented, how we can deal with the limited range in the representation, and how various operations are defined on these representations.

In terms of our learning goals, this lecture addresses:

Computational Thinking: Working with and around resource limitations.

Algorithms and Data Structures: Employing integer algorithms (binary addition)

Programming: Identifying, describing, and effectively using integers as signed modular arithmetic and as fixed-length bit vectors in C0.

1 Binary Representation of Natural Numbers

For the moment, we only consider the natural numbers $0, 1, 2, \ldots$ and we do not yet consider the problems of limited range. Number notations have a *base* b. To write down numbers in base b we need b distinct *digits*. Each digit is multiplied by an increasing power of b, starting with b^0 at the right end. For example, in base 10 we have the ten digits 0–9 and the string `9380` represents the number $9*10^3 + 3*10^2 + 8*10^1 + 0*10^0$. We call numbers in base 10 *decimal numbers*. Unless it is clear from context that we are talking about a certain base, we use a subscript$_{[b]}$ to indicate a number in base b.

In computer systems, two bases are of particular importance. *Binary numbers* use base 2, with digits 0 and 1, and *hexadecimal numbers* (explained more below) use base 16, with digits 0–9 and A–F. Binary numbers are so important because the basic digits, 0 and 1, can be modeled inside the computer by two different voltages, usually "off" for 0 and "on" for 1. To find the number represented by a sequence of binary digits we multiply each digit by the appropriate power of 2 and add up the results. In general, the value of an n-bit sequence

$$b_{n-1} \dots b_1 b_0\ _{[2]} = b_{n-1}2^{n-1} + \cdots + b_1 2^1 + b_0 2^0 = \sum_{i=0}^{n-1} b_i 2^i$$

For example, `10011`$_{[2]}$ represents $1 * 2^4 + 0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = 16 + 2 + 1 = 19$.

We can also calculate the value of a binary number in a nested way, exploiting Horner's rule for evaluating polynomials.

$$10011_{[2]} = (((1 * 2 + 0) * 2 + 0) * 2 + 1) * 2 + 1 = 19$$

In general, if we have an n-bit number with bits $b_{n-1} \dots b_0$, we can calculate

$$(\cdots((b_{n-1} * 2 + b_{n-2}) * 2 + b_{n-3}) * 2 + \cdots + b_1) * 2 + b_0$$

Example 1. *For example, taking the binary number* ***`10010110`***$_{[2]}$ *write the digits from most significant to least significant, calculating the cumulative value from left to right by writing it top to bottom.*

$$\begin{array}{rcccccr|l}
 & & & & \mathbf{1} & = & 1 & \uparrow\ \textit{leftmost bit} \\
1 & * & 2 & + & \mathbf{0} & = & 2 & \\
2 & * & 2 & + & \mathbf{0} & = & 4 & \\
4 & * & 2 & + & \mathbf{1} & = & 9 & \\
9 & * & 2 & + & \mathbf{0} & = & 18 & \\
18 & * & 2 & + & \mathbf{1} & = & 37 & \\
37 & * & 2 & + & \mathbf{1} & = & 75 & \\
75 & * & 2 & + & \mathbf{0} & = & 150 & \textit{rightmost bit}
\end{array}$$

Reversing this process allows us to convert a number into binary form. Here we start with the number and successively divide by two, calculating the remainder. At the end, the least significant bit is at the top.

For example, converting 198 to binary form would proceed as follows:

$$
\begin{array}{rcrcrcl|l}
198 & = & 99 & * & 2 & + & \mathbf{0} & \textit{rightmost bit} \\
99 & = & 49 & * & 2 & + & \mathbf{1} & \\
49 & = & 24 & * & 2 & + & \mathbf{1} & \\
24 & = & 12 & * & 2 & + & \mathbf{0} & \\
12 & = & 6 & * & 2 & + & \mathbf{0} & \\
6 & = & 3 & * & 2 & + & \mathbf{0} & \\
3 & = & 1 & * & 2 & + & \mathbf{1} & \\
1 & = & 0 & * & 2 & + & \mathbf{1} & \downarrow \; \textit{leftmost bit}
\end{array}
$$

We read off the answer, from bottom to top, arriving at `11000110`$_{[2]}$.

2 Modular Arithmetic

Within a computer, there is a natural size of words that can be processed by single instructions. In early computers, the word size was typically 8 bits; now it is 32 or 64. In programming languages that are relatively close to machine instructions like C or C0, this means that the native type **`int`** of integers is limited to the size of machine words. In C0, we decided that the values of type **`int`** occupy 32 bits.

This makes it very easy to deal with for small numbers, because the more significant digits can simply be 0. According to the formula that yields their number value, these bits do not contribute to the overall value. But we have to decide how to deal with large numbers, when operations such as addition or multiplication would yield numbers that are too big to fit into a fixed number of bits. One possibility would be to raise overflow exceptions. This is somewhat expensive (since the overflow condition must be explicitly detected), and has other negative consequences. For example, $(n+n)-n$ is no longer equal to $n+(n-n)$ because the former can overflow while the latter always yields n and does not overflow. Another possibility is to carry out arithmetic operations *modulo* the number of representable integers, which would be 2^{32} in the case of C0. We say that the machine implements *modular arithmetic*.

In higher-level languages, one would be more inclined to think of the type of **`int`** to be inhabited by integers of essentially unbounded size. This means that a value of this type would consist of a whole vector of machine words whose size may vary as computation proceeds. Basic operations such as addition no longer map directly onto machine instruction, but are implemented by small programs. Whether this overhead is acceptable depends on the application.

Returning to modular arithmetic, the idea is that any operation is carried out modulo 2^p for size p. Even when the modulus is not a power of two, many of the usual laws of arithmetic continue to hold, which makes it possible to write programs confidently without having to worry, for example, about whether to write $x + (y + z)$ or $(x + y) + z$. We have the following properties of the abstract algebraic class of *rings* which are shared between ordinary integers and integers modulo a fixed number n.

Commutativity of addition	$x + y = y + x$
Associativity of addition	$(x + y) + z = x + (y + z)$
Additive unit	$x + 0 = x$
Additive inverse	$x + (-x) = 0$
Cancellation	$-(-x) = x$
Commutativity of multiplication	$x * y = y * x$
Associativity of multiplication	$(x * y) * z = x * (y * z)$
Multiplicative unit	$x * 1 = x$
Distributivity	$x * (y + z) = x * y + x * z$
Annihilation	$x * 0 = 0$

Some of these laws, such as associativity and distributivity, do *not* hold for so-called *floating point* numbers that approximate real numbers. This significantly complicates the task of reasoning about programs with floating point numbers which we have therefore omitted from C0.

3 An Algorithm for Binary Addition

In the examples below, we use arithmetic modulo 2^4, with 4-bit numbers. Addition proceeds from right to left, adding binary digits modulo 2, and using a carry if the result is 2 or greater. For example,

$$\begin{array}{ccccccccc} & 1 & 0 & 1 & 1 & & = & 11 & \\ + & 1 & 0_1 & 0_1 & 1 & & = & 9 & \\ \hline (1) & 0 & 1 & 0 & 0 & & = & 20 & = 4 \ (\mathrm{mod}\ 16) \end{array}$$

where we used a subscript to indicate a carry from the right. The final carry, shown in parentheses, is ignored, yielding the answer of 4 which is correct modulo 16.

This grade-school algorithm is quite easy to implement in software, but it is not suitable for a hardware implementation because it is too sequential. On 32 bit numbers the algorithm would go through 32 stages, for an operation which, ideally, we should be able to perform in one machine cycle. Modern hardware accomplishes this by using an algorithm where more of the work can be done in parallel.

4 Two's Complement Representation

So far, we have concentrated on the representation of natural numbers $0, 1, 2, \ldots$. In practice, of course, we would like to program with negative numbers. How do we define negative numbers? We define negative numbers as additive inverses: *$-x$ is the number y such that $x + y = 0$.* A crucial observation is that in modular arithmetic, additive inverses already exist! For example, $-1 = 15 \pmod{16}$ because $-1 + 16 = 15$. And $1 + 15 = 16 = 0 \pmod{16}$, so, indeed, 15 is the additive inverse of 1 modulo 16.

Example 2. *Similarly, $-2 = 14 \pmod{16}$, $-3 = 13 \pmod{16}$, etc. Writing out the equivalence classes of numbers modulo 16 together with their binary representation, we have*

…	−16	0	16	…	0000
…	−15	1	17	…	0001
…	−14	2	18	…	0010
…	−13	3	19	…	0011
…	−12	4	20	…	0100
…	−11	5	21	…	0101
…	−10	6	22	…	0110
…	−9	7	23	…	0111
…	−8	8	24	…	1000
…	−7	9	25	…	1001
…	−6	10	26	…	1010
…	−5	11	27	…	1011
…	−4	12	28	…	1100
…	−3	13	29	…	1101
…	−2	14	30	…	1110
…	−1	15	31	…	1111

At this point we just have to decide which numbers we interpret as positive and which as negative. We would like to have an equal number of positive and negative *numbers, where we include 0 among the positive ones. From this consideration we can see that $0, \ldots, 7$ should be positive and $-8, \ldots, -1$ should be negative and that the highest bit of the 4-bit binary representation tells us if the number is positive or negative.*

Just for verification, let's check that $7 + (-7) = 0 \pmod{16}$:

$$\begin{array}{rcccc} & 0 & 1 & 1 & 1 \\ + & 1_1 & 0_1 & 0_1 & 1 \\ \hline (1) & 0 & 0 & 0 & 0 \end{array}$$

We can obtain $-x$ from x in the bit representation by first complementing all the bits and then adding 1. In fact, the addition of x with its bitwise complement (written $\sim x$) always consists of all 1's, because in each position we have a 0 and a 1, and no carries at all. Adding one to the number $11\ldots11$ will always result in $00\ldots00$, with a final carry of 1 that is ignored. We can write this as a handy formula to compute $\sim x$ given x:

$$-x = \sim x + 1$$

These considerations also show that, regardless of the number of bits, -1 is always represented as a string of 1's.

In 4-bit numbers, the maximal positive number is 7 and the minimal negative number is -8, thus spanning a range of $16 = 2^4$ numbers. In general, in a representation with p bits, the positive numbers go from 0 to $2^{p-1} - 1$ and the negative numbers from -2^{p-1} to -1. It is remarkable that because of the origin of this representation in modular arithmetic, the "usual" bit-level algorithms for addition and multiplication can ignore that some numbers are interpreted as positive and others as negative and still yield the correct answer modulo 2^p.

However, for comparisons, division, and modulus operations the sign does matter. We discuss division below in Section 7. For comparisons, we just have to properly take into account the highest bit because, say, $-1 = 15 \pmod{16}$, but $-1 < 0$ and $0 < 15$.

5 Hexadecimal Notation

In C0, we use 32 bit integers. Writing these numbers out in decimal notation is certainly feasible, but sometimes awkward since the bit pattern of the representation is not easy to discern. Binary notation is rather expansive (using 32 bits for one number) and therefore difficult to work with. A good compromise is found in *hexadecimal notation*, which is a representation in base 16 with the sixteen digits 0–9 and A–F. "Hexadecimal" is often abbreviated as "hex". In the concrete syntax of C0 and C, hexadecimal numbers are preceded by `0x` in order to distinguish them from decimal numbers.

binary	hex	decimal
0000	`0x0`	0
0001	`0x1`	1
0010	`0x2`	2
0011	`0x3`	3
0100	`0x4`	4
0101	`0x5`	5
0110	`0x6`	6
0111	`0x7`	7
1000	`0x8`	8
1001	`0x9`	9
1010	`0xA`	10
1011	`0xB`	11
1100	`0xC`	12
1101	`0xD`	13
1110	`0xE`	14
1111	`0xF`	15

Hexadecimal notation is convenient because most common word sizes (8 bits, 16 bits, 32 bits, and 64 bits) are multiples of 4. For example, a 32 bit number can be represented by eight hexadecimal digits. We can even do a limited amount of arithmetic on them, once we get used to calculating modulo 16. Mostly, though, we use hexadecimal notation when we use bitwise operations rather than arithmetic operations.

6 Useful Powers of 2

The drive to expand the native word size of machines by making circuits smaller was influenced by two different considerations. For one, since the bits of a machine word (like 32 or 64) are essentially treated in parallel in the circuitry, operations on larger numbers are much more efficient. For another, we can address more memory directly by using a machine word as an address.

A useful way to relate this to common measurements of memory and storage capacity is to use

$$2^{10} = 1024 = 1K$$

Note that this use of "$1K$" in computer science is slightly different from its use in other sciences where it would indicate one thousand ($1,000$). If we want to see how much memory we can address with a 16 bit word we calculate

$$2^{16} = 2^6 * 2^{10} = 64K$$

so roughly 64K cells of memory each usually holding a byte which is 8 bits wide). We also have

$$2^{20} = 2^{10} * 2^{10} = 1,048,576 = 1M$$

(pronounced "1 Meg") which is roughly 1 million and

$$2^{30} = 2^{10} * 2^{10} * 2^{10} = 1,073,741,824 = 1G$$

(pronounced "1 Gig") which is roughly 1 billion.

In a more recent processor with a word size of 32 we can therefore address

$$2^{32} = 2^2 * 2^{10} * 2^{10} * 2^{10} = 4GB$$

of memory where "GB" stands for Gigabyte.

The next significant number would be $1024GB$ which would be $1TB$ (Terabyte).

7 Integer Division and Modulus

The division and modulus operators on integers are somewhat special. As a multiplicative inverse, division is not always defined, so we adopt a different definition. We write x/y for *integer division* of x by y and $x\%y$ for *integer modulus*. The two operations must satisfy the property

$$(x/y) * y + (x\%y) = x$$

so that $x\%y$ is like the remainder of division. The above is not yet sufficient to define the two operations. In fact, $x/y = 0$ and $x\%y = x$ for all x and y would satisfy this property, but this is clearly not what we want. In addition we say that $0 \leq |x\%y| < |y|$. Still, this leaves open the possibility that the modulus is positive or negative when y does not divide x evenly. We fix this by stipulating that integer division truncates its result towards zero. This means that the modulus must be negative if x is negative and there is a remainder, and it must be positive if x is positive. Thus, $7\%5 = 2$ and $-7\%5 = -2$.

Another way to satisfy the above property under the constraint that $0 \leq |x\%y| < |y|$ is for x/y to always truncate down (towards $-\infty$), which means that the *remainder* $x\%y$ is positive exactly when y is positive. With such definition, these two operations are called quotient and remainder, respectively. In C0, `x/y` and `x%y` implement integer division and integer modulus. There are no primitive operators in C0 for quotient and remainder, but they can be implemented with the ones at hand.

Of course, the above constraints are impossible to satisfy when $y = 0$, because $0 \leq |x\%0| < |0|$ is impossible. But division by zero is defined to raise an error, and so is the modulus.

8 Bitwise Operations on Ints

Ints are also used to represent other kinds of data. An example is colors (discussed further in Section 10). The so-called *ARGB* color model divides an **`int`** into four 8-bit quantities. The highest 8 bits represent the opaqueness of the color against its background, while the lower 24 bits represent the intensity of the red, green and blue components of a color. Manipulating this representation with addition and multiplication is quite unnatural; instead we usually use bitwise operations.

The bitwise operations are defined by their action on a single bit and then applied in parallel to a whole word. The tables below define the meaning of *bitwise and* `&`, *bitwise exclusive or* `^` and *bitwise or* `|`. We also have *bitwise negation* `~` as a unary operation.

And

&	0	1
0	0	0
1	0	1

Exclusive Or

^	0	1
0	0	1
1	1	0

Or

|	0	1
0	0	1
1	1	1

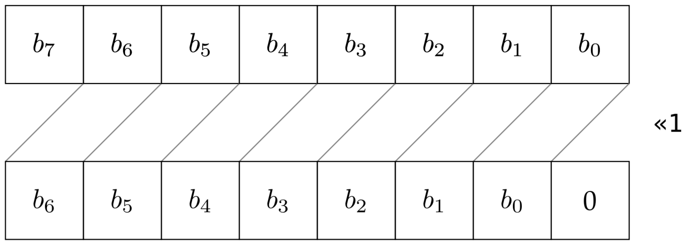
Negation

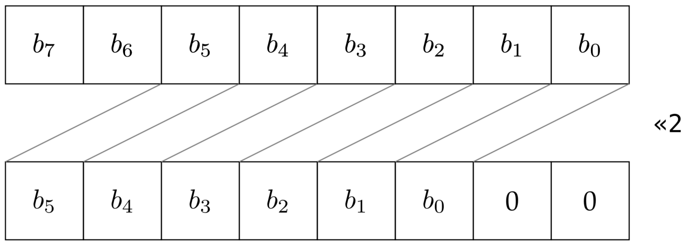
~	0	1
	1	0

9 Shifts

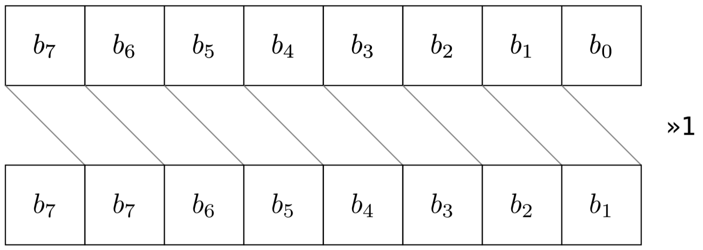
We also have some hybrid operators on ints, somewhere between bit-level and arithmetic. These are the *shift* operators. We write `x << k` for the result of shifting x by k bits to the left, and `x >> k` for the result of shifting x by k bits to the right. In both cases, the value of k must be between 0 (inclusive) and 32 (exclusive) — any other value is an arithmetic error like division by zero. We assume below that k is in that range.

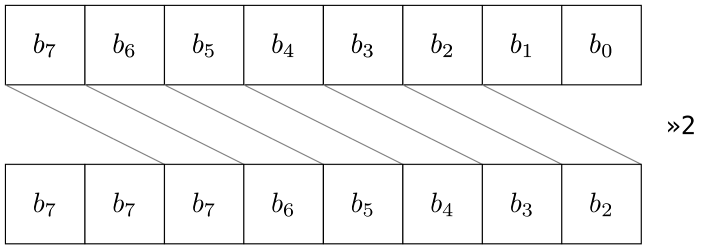
The left shift, `x << k` (for $0 \leq k < 32$), fills the result with zeroes on the right, so that bits $0, \ldots, k-1$ will be 0. Every left shift corresponds to a multiplication by 2 so `x << k` returns $x * 2^k$ (modulo 2^{32}). We illustrate this with 8-bit numbers.


b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
«1
b_6 b_5 b_4 b_3 b_2 b_1 b_0 0


b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
«2
b_5 b_4 b_3 b_2 b_1 b_0 0 0

The right shift, `x >> k` (for $0 \leq k < 32$), copies the highest bit while shifting to the right, so that bits $31, \ldots, 32-k$ of the result will be equal to the highest bit of x. If viewing x as an integer, this means that the sign of the result is equal to the sign of x, and shifting x right by k bits corresponds to integer division by 2^k except that it truncates towards $-\infty$. For example, `-1 >> 1 == -1`.


b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
»1
b_7 b_7 b_6 b_5 b_4 b_3 b_2 b_1


b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
»2
b_7 b_7 b_7 b_6 b_5 b_4 b_3 b_2

10 Representing Colors

As a small example of using the bitwise interpretation of **`int`**s, we consider colors. Colors are decomposed into their primary components red, green, and blue; the intensity of each uses 8 bits and therefore varies between 0 and 255 (or `0x00` and `0xFF`). We also have the so-called *α-channel* which indicates how opaque the color is when superimposed over its background. Here, `0xFF` indicates completely opaque, and `0x00` completely transparent.

Example 3. *For example, to extract the intensity of the red color in a given pixel p, we could compute `(p >> 16) & 0xFF`. The first shift moves the red color value into the bits 0–7; the bitwise and masks out all the other bits by setting them to 0. The result will always be in the desired range, from 0–255.*

Conversely, if we want to set the intensity of green of the pixel p to the value of g (assuming we already have $0 \leq g \leq 255$), we can compute `(p & 0xFFFF00FF) | (g << 8)`. This works by first setting the green intensity to 0, while keep everything else the same, and then combining it with the value of g, shifted to the right position in the word.

For more on color values and some examples, see Assignment 1.

11 Exercises

Exercise 1. *Write functions* `quot` *and* `rem` *that calculate quotient and remainder as explained in Section 7. Your functions should have the property that*

```
quot(x,y)*y + rem(x,y) == x;
```

for all **`int`***s x and y unless* `quot` *overflows. How is that possible?*

Exercise 2. *Write a function* `int2hex` *that returns a string containing the hexadecimal representation of a given integer as a string. Your function should have prototype*

```
string int2hex(int x);
```

(The prototype *of a function is the function without its body: the prototype gives the function name and the type of its arguments.)*

Exercise 3. *Write a function* `lsr` *(logical shift right), which is like right shift (*`>>`*) except that it fills the most significant bits with zeroes instead of copying the sign bit. Explain what* `lsr(x,1)` *means on integers in two's complement representation.*

Exercise 4 (sample solution on page 13)**.** *Rewrite the functions* `POW` *and* `f` *(the mystery function) from Chapter* **??** *so that they signal an error in case of an overflow rather than silently working in modular arithmetic. You can use the statement* `error("Overflow");` *to signal an overflow. Don't worry about catching overflow with a negative base — that's a little more complicated.*

Sample Solutions

Solution of exercise 4

```
int POW (int x, int y)
//@requires y >= 0;
{
  if (y == 0)
    return 1;
  int p = x * POW(x, y-1);
  if (x > 0 && p <= 0)  error("Overflow");
  return p;
}
```

If what gets returned from a recursive call is ever negative, we know for sure we have had an overflow. This holds also if this value is 0, but only if `x` did not start at 0.

```
int f(int x, int y)
//@requires y >= 0;
//@ensures \result == POW(x,y);
{
  int r = 1;
  int b = x;             /* base     */
  int e = y;             /* exponent */
  while (e > 0)
  //@loop_invariant e >= 0;
  //@loop_invariant r * POW(b,e) == POW(x,y);
  {
    if (e % 2 == 1) {
      r = b * r;
      if (x > 0 && r <= 0) error("Overflow");
    }
    b = b * b;
    e = e / 2;
  }
  //@assert e == 0;
  return r;
}
```

Putting a similar check after `b = b*b` would be incorrect: performing this computation on the last iteration (when `e==1`) does not influence the result of the function and therefore it is unimportant whether it overflows.

Note that this isn't how we'd normally want to detect overflow. We'll talk about this later in the course, but we'd rather notice that overflow will happen before actually causing it, rather than noticing after it happens.