High Performance Programming and Systems

Assignment One

From logic to arithmetic

Set: *25th of November 2021*
Due: *3rd of December 2021 @ 23:59 CEST*

Synopsis:
Implement integer arithmetic using bitwise operations.

1 Introduction

This is the first of five assignments in the *High Performance Parallel Systems* course. In this assignment you will implement binary arithmetic in C. The purpose is twofold: partly to train you in basic C programming, and partly to show your understanding of binary number representations. In particular, to show you how arithmetic operations can be built on top of logical operations.

In this assignment you will be implementing a data type called `struct bits8` as well as functions that consume and produce values of type `struct bits8`. The definition of the data type (as a C `struct`) is given, as are the prototypes of the functions you are asked to implement. You may need to define helper functions along the way. The assignment builds on the `bits.h` header developed in the first exercise session.

The `struct bits8` type contains eight bits, with `b0` representing the least significant ("rightmost") bit:

```
struct bits8 {
  struct bit b0;
  struct bit b1;
  struct bit b2;
  struct bit b3;
  struct bit b4;
  struct bit b5;
  struct bit b6;
  struct bit b7;
};
```

As you you will learn later in the course, this is not a particularly efficient way to define this type in C, but it will suffice for this assignment. In particular, it will prevent you from mistakes that arise due to C's treatment of boolean values as numbers.

In the tasks below, we will interpret values of type `struct bits8` as both 8-bit signed integers and 8-bit unsigned integers, depending on the task. Always remember that at machine level, *data is just data*, and we assign it meaning by how we interpret it.

You will be working in a file called `numbers.h`. Use a program `test_numbers` to test your implementation, similarly to how it was done for `bits.h` at the exercise.

You will be asked to implement several specific functions. You are allowed and expected to define additional utility functions as needed, in order to make your code easier to read and understand.

Do not use C-based control flow such as `if` statements or loops in `numbers.h` (the ones in `bits.h` are fine). Feel free to use any language features you wish in `test_numbers.h`.

2 Converting to and from C integers

In this task you must implement these three functions:

```
// Convert C integer to a bits8.
struct bits8 bits8_from_int(unsigned int x);

// Convert a bits8 to a C integer.
unsigned int bits8_to_int(struct bits8 x);

// Print the bits of a bits8 in the conventional order,
// with no trailing newline.
void bits8_print(struct bits8 v);
```

2.1 Hints and specifics

- `bits8_from_int(2)` should produce a value with `b1` set to 1 and everything else to 0.

- `bits8_to_int(bits8_from_int(x)) == x` for values of `x` that fit in 8 bits.
- Consider implementing `bits8_print()` first to help with debugging.
- `bits8_print(bits8_from_int(123))` should print `01111011`.
- Consider writing a helper function

  ```
  unsigned int get_bit(unsigned int x, int i);
  ```

  that returns bit i from the integer x. E.g. `get_bit(2,1) == 1`, `get_bit(2,0) == 0`. This can be implemented with right-shifting (`>>`) and masking (`&1`).
- Consider writing a helper function

  ```
  unsigned int set_bit(unsigned int x, int i);
  ```

  that returns `x` but with the bit at position i set to 1. E.g. `set_bit(2,0) == 3`, `set_bit(2,1) == 2`. This can be implemented with left-shifting (`<<`) and bitwise-or (`|`).

3 Implementing addition

Implement this function:

```
struct bits8 bits8_add(struct bits8 x, struct bits8 y);
```

Interpret `x` and `y` to as unsigned 8-bit numbers and return their sum. Adding binary numbers is much like adding decimal numbers. Starting from the least significant (rightmost) bit, we add them elementwise, keeping a carry. Example for adding $x + y = s$ where $x = 01011_2, y = 01001_2$:

i	x_i	y_i	s_i	c_i
0	1	1	0	1
1	1	0	1	0
2	0	1	1	0
3	1	1	1	1
5	0	0	1	0

The result is $s = 11110_2$ with no carry. Precisely, if we denote exclusive-or by $\oplus$, then

$$s_i = x_i \oplus y_i \oplus c_{i-1} \tag{1}$$

$$c_i = (x \,\&\, y) \mid ((x \mid y) \,\&\, c) \tag{2}$$

where we let $c_{-1} = 0$ (i.e. assume no carry-in for the least significant bit). These two rules can be implemented as a digital circuit called a *full adder*, but we'll stick to implementing it as C code.

3.1 Hints and specifics

- Do not use loops in your implementation.
- Do not convert to C integers and use C's `+` operator.
- Use the bit operations from `bits.h` (`bit_and`, `bit_or` etc).
- It may be useful (but not required) to define a helper function that implements a full adder. Since a full adder produces two values (s_i, c_i), and C functions only can return a single value, we have to define a new type to contain two bits:

  ```
  struct add_result {
    struct bit s;
    struct bit c;
  };

  struct add_result
  bit_add(struct bit x, struct bit y, struct bit c) {
    ...
  }
  ```

- Feel free to test your implementation against C's builtin `+`, which you may assume is correct. For example, given two values `x` and `y` of type `unsigned int`, a good test may be to check whether

  ```
  bits8_to_int(bits8_add(bits8_from_int(x),
                         bits8_from_int(y)))
  ```

and

```
(x+y) & 0xFF
```

produce the same result.

4 Implementing negation

Implement this function:

```
struct bits8 bits8_negate(struct bits8 x);
```

Interpret `x` as an 8-bit two's complement number and return the arithmetic negation. A two's complement number is negated by not-ing each bit individually and then incrementing by one.

4.1 Hints and specifics

- *Negation* is sometimes used to denote flipping each bit—what we refer to in this text as not-ing. In this assigment, "negation" means the equivalent to multiplying by -1.
- Consider negating -1_{10}. The two's complement representation of -1_{10} in 8 bits is 11111111_2. Not-ing each bit gives us 00000000_2, and incrementing then gives $00000001_2 = 1_{10}$. If we not each bit again, we get 11111110_2, and incrementing then gives $11111111_2 = -1_{10}$.
- Incrementing can be done with `bits8_add()`.

5 Implementing multiplication

Implement this function:

```
struct bits8 bits8_mul(struct bits8 x, struct bits8 y);
```

Interpret `x` and `y` as unsigned numbers and return their product.

Multiplying binary numbers can be done using essentially the same algorithm you learned in elementary school, but with bits instead of digits. More efficient algorithms exist, but the naive one is enough for this assignment.

To compute the product z of two k-bit numbers x and y, use the formula

$$z = \sum_{i=0}^{k-1} x \cdot y_i \cdot 2^i$$

where y_i is the ith digit of y.

5.1 Hints and specifics

- The summation can be done using the addition function you implemented before.
- The product $x \cdot y_i$ is multiplying a binary number with a single bit. That is, $x \cdot y_i = x$ if $y_i = 1$, and otherwise $x \cdot y_i = 0$. Is this similar to a bitwise operation you have seen before?
- The multiplication with 2^i can be implemented with a left-shift.

6 Code handout

`Makefile`: How to build the source files. Already containts rules for `test_numbers`, so you do not have to modify it—but you may if you wish. Use `make test_numbers` to compile the test program.

`bits.h`: An implementation of single bits, from the Thursday exercises. Do not modify this file.

`test_bits.h`: Tests for `bits.h`, from the Thursday exercises. Do not modify this file.

`numbers.h`: The implementation of binary numbers. You will have to modify this file.

`test_numbers.c`: An initially empty test program. You will have to modify this file.

7 Your Report

You are expected to comment on the *interesting* details of your implementation. You are *not* expected to give a line-by-line walkthrough of your code. Most importantly, you are expected to reflect on the *quality* of your code:

- Do you think it is functionally correct? Why or why not?
- Is there some improvement you'd have liked to make, but didn't have the time?

It is more important to be aware of the strengths or shortcomings of your solution, than it is to have a complete solution.

7.1 The structure of your report

Your report should be structured exactly as follows:

Introduction: Briefly mention very general concerns, your own estimation of the quality of your solution, and possibly how to run your tests.

A section for each of the four tasks: Mention whether your solution is functional, which cases it fails for, and what you think might be wrong.

A section answering the following specific questions:

1. Does `bits8_add()` provide "correct" results if you pass in negative numbers in two's complement representation? Why or why not?
2. Does `bits8_mul()` provide "correct" results if you pass in negative numbers in two's complement representation?
3. How would you implement a function `bits8_sub()` for subtracting 8-bit numbers?

All else being equal, **a short report is a good report**.

8 Deliverables for This Assignment

You should submit the following items:

- A single PDF file, A4 size, no more than 5 pages, describing each item from report section above.
- A single zip/tar.gz file with all code relevant to the implementation, including at least all the files from the handout.

9 Assessment

You will get written qualitative feedback, and points from zero to 4. There are no resubmissions, so please hand in what you managed to develop, even if you have not solved the assignment completely.