HPPS 2025

Assignment One

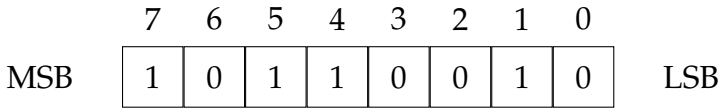
From logic to arithmetic

Due: 2025-11-27

Synopsis: Implement integer arithmetic using bitwise operations without traditional control flow.

1 Introduction

In this assignment you will implement binary arithmetic in C. The purpose is twofold: (1) to test your skill in basic C programming, and (2) to test your understanding of binary numbers and bitwise operations. More specifically, you will build arithmetic operations on top of bitwise operations, **without using traditional control flow (i.e., no ifs, no loops)**.



This assignment relies `bitvec8`: an eight-bit vector, where bit `0` is the least significant ("rightmost") bit, as shown above. In the tasks below, we will interpret values of type `bitvec8` as 8-bit signed integers or 8-bit unsigned integers, depending on the task. Always remember that at machine level, *bits are just bits*, and we assign them meaning by how we interpret them.

The definition of the data type is given, as are the prototypes of the functions you are asked to implement. You will be working in files called `bitvec8.c` and `test_bitvec8.c` to test your implementation, similarly to how it was done in the week 1 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 `bitvec8.c`

2 Implementing addition

Implement this function:

```
bitvec8 bitivec8_add(bitvec8 x, bitvec8 y);
```

It returns the sum of `x` and `y` considered as unsigned 8-bit numbers.

Here is a description of the approach you should implement.

Let us start our explanation with the addition of two bits b_i and b_j in binary:

```
0 + 0 = 0 (sum: 0, carry: 0)
1 + 0 = 1 (sum: 1, carry: 0)
0 + 1 = 0 (sum: 1, carry: 0)
1 + 1 = 10 (sum: 0, carry: 1)
```

This is called a half-adder (half because the carry is generated but not included in the addition). The key here is that the sum is obtained with `b_i ^ b_j` (bitwise xor), while the carry is obtained with `b_i & b_j` (bitwise and). This works for a single bit. How should we proceed with 8 bits?

The key observation is that we need to handle the carry.

The approach you will implement is called **Ripple-Carry Adder**. The idea is to (a) compute a partial sum of two bit vectors (ignoring carries, as in the half adder above), and (b) compute the carry bits. The good news is that we can use `bitwise xor` and `bitwise and` as in the half adder above to compute sum and carry. The bad news is that we need to apply the carry to the partial sum: Worse, the carry should be added to the `next position` in the bit vector. To do this we must left shift the carry by one position. The final sum is obtained by iterating over the computation of the partial sum and carry, first on the initial bit vectors, then on the resulting partial sum and carry, until there is no `next position` to shift the carry into.

2.1 Hints and specifics

- Do not convert to C integers and do not use the + operator.

- When you find yourself tempted to write a `for` loop or a `while` loop with a fixed number of iterations (which is not allowed), consider *unrolling* the loop - that is, replicating the loop body for every desired iteration

- It is natural to define a helper function that implements a modified half adder issueing partial sum and repositioned carry as explained above. Since such a function 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
sum_and_carry(bitvec8 x, bitvec8 y) {
  ...
}
```

- 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.

3 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 logically negating each bit individually and then incrementing by one.

3.1 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" is an arithmetic operation multiplying by -1.
- Consider negating -1_{10}. The two's complement representation of -1_{10} in 8 bits is $\langle 11111111 \rangle$. Not-ing each bit gives us $\langle 00000000 \rangle$, and incrementing then gives $\langle 00000001 \rangle = 1_{10}$. If we not each bit again, we get $\langle 11111110 \rangle$, and incrementing then gives $\langle 11111111 \rangle = -1_{10}$.
- Incrementing can be done with `bits8_add()`.

4 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. The product z of two k-bit numbers x and y is given by

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

where y_i is the ith digit of y.

4.1 Hints and specifics

- The key issue is to find a way to iterate over each digit of the bit vector. Note that the combining some of the functions from the exercise will get you there.

- 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 $x \cdot y_i = 0$.
- The summation can be done using the addition function you implemented before.
- The multiplication with 2^i can be implemented with a left-shift.

5 Code handout

`Makefile`: How to build the source files. Already contains rules for building `test_bitvec8`, so you do not have to modify it—but you may if you wish. Use `make test` to compile the test program.

`bitvec8.h`: The definition of the `bitvec8` and the prototype for the functions that are tested. You do not have to modify this file.

`bitvec8.c`: The implementation of the functions defined in `bitvec8.h` and additional helper functions. You will have to modify this file.

`test_bitvec8.c`: A unit test program with one test function for each function defined in `bitvec8.h`. You will have to modify this file.

6 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.

6.1 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 three tasks: Mention whether your solution is functional, how you define your tests, 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**. Please keep it under 5 pages.

7 Deliverables for This Assignment

You must 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. For this assignment it is not necessary to add additional files.

Remember to follow the general assignment rules listed on the course homepage.