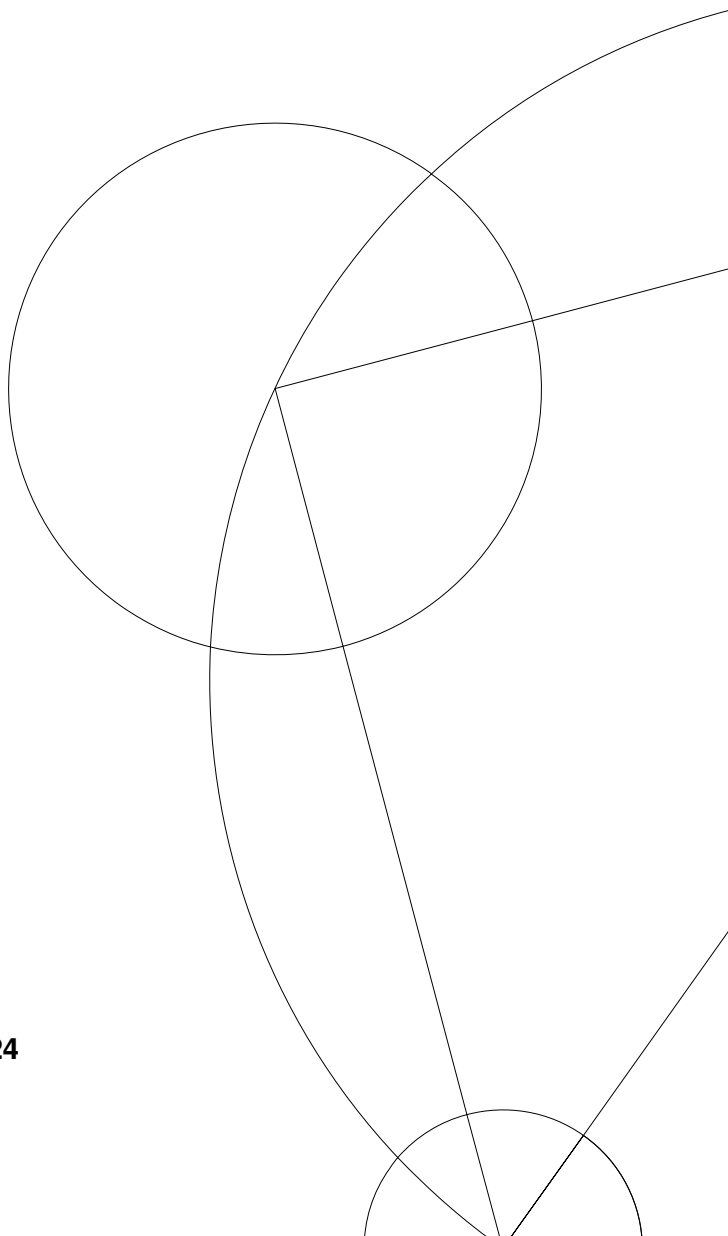
KØBENHAVNS UNIVERSITET

Assignment 1

Group members: zlp616, fmt568, dzb502

High Performance Programming and Systems (HPPS)
Course code » NDAB20001U



Course supervisor: Troels Henriksen

Department of Computer Science November 21, 2024

Table of contents

1 Introduction

A common concern with these tasks is that we are working with only 8 bits, which is not a very large range; for signed numbers, we effectively have only 7 bits to work with. This limitation means that, even with thorough testing to ensure good quality, overflow can still occur when handling numbers that are too large, potentially leading to undefined behavior. Overflow can occur, such as in the case of $127 \times -2 = -254$. This exceeds the range of $[-128, 127]$ for `int8_t`. When cast to 8 bits, (-254) becomes 2, resulting in undefined behavior due to overflow. Additionally, we have used a naive approach for multiplication, which is sufficient for this assignment.

To execute the tests, use the following commands, which produce the tests in appendix 1:

```
$./make
$./test_numbers
```

Listing 1: Commands for running the tests

2 Tasks

2.1 Converting to bits8 from int

The purpose is to convert an unsigned integer `x` to an 8-bit representation using the struct `bits8`. Firstly, we ensure, that `x` is within the range of an 8-bit unsigned integer (0 to 255). If a signed integer is passed instead, its value will first be implicitly converted to an unsigned integer. In such cases, negative signed values are interpreted as their unsigned equivalents using two's complement representation. For example, `-1` (signed) becomes `255` (unsigned), and `-128` (signed) becomes `128` (unsigned). We know, that the struct stores each bit-value as `b0.v`, `b1.v` ... `b7.v`, thus we need to extract the individual bits from the given unsigned int `x` using bitwise operations:

```
struct bits8 bits8_from_int(unsigned int x) {
  ASSERT(x <= 255, "Value is too large for 8 bits");
  result.b0.v = x & 1;
   ⋮
  result.b7.v = x >> 7 & 1;
  return result;
}
```

Listing 2: Converting to bits8 from int

Each bit is extracted by performing a bitwise `AND`-operation with 1, represented as `0b1`, after right-shifting `x` by the appropriate number of bits, which ensures that we are looking at the specific bit placed as the least significant bit followed by only 0's.

2.2 Converting to int from bits8

The purpose is to convert a struct `bits8` to an unsigned integer. We know, that the struct stores each bit-value as `b0.v`, `b1.v`, ..., `b7.v`, thus we need to combine these bits into an unsigned integer using bitwise operations:

```
unsigned int bits8_to_int(struct bits8 x) {
  result |= x.b0.v;
   ⋮
  result |= x.b7.v << 7;
  return result;
}
```

Listing 3: Converting to int from bits8

Each bit is combined by performing a bitwise `OR`-operation with the result, after left-shifting the bit-value by the appropriate number of bits, which ensures that we are placing the specific bit in the correct position. So, for `x.b0.v`, we are placing the bit in the least significant bit, for `x.b1.v` we are shifting the bit one position to the left, so it is in the second least significant bit, and so forth.

2.3 Printing bits8

The purpose is to print the binary representation of a `struct bits8`. It converts the `bits8` to its integer equivalent using `bits8_to_int` and will then iterate over each bit, starting from the most significant bit to the least significant bit and extracts each bit by right-shifting the integer with the same method as above.

3 Implementing addition

The purpose is to add two bits8 values (`xv` and `yv`). For this, we have implemented a helper function `add_helper` that performs the bit-wise addition for a single bit, considering the carry from the previous operation (can be 0 or 1 respectively). It has three cases:

- If both `xv` $= 1 =$ `yv`, then set result-bit (`bit->v`) to the value of the carry and update the carry to 1 if not already set.
- If either `xv` or `yv` is 1, then set the result-bit to the negated value of `carry`:
  $$\texttt{bit->v} = \begin{cases} 1 & \text{if } \texttt{carry} = 0 \\ 0 & \text{if } \texttt{carry} = 1 \end{cases}$$
  The reason for this is, that if `carry` is 1, we have $1 + 1$, meaning that the resulting bit will be 0 and should carry over a 1. If `carry` is 0, we have $0 + 1$, so the resulting bit will be 1 and should not carry over any value.

- If both `xv` = 0 = `yv`, then set the result-bit to the value of the `carry` and reset `carry` to 0.

These cases ensure, that we correctly add the bits with respect to the `carry` gotten from the previous operation, which can be seen in the code for `add_helper` below:

```
void add_helper(int* carry, struct bit* bit, bool xv, bool yv) {
  if (xv && yv) {
    bit->v |= *carry;
    *carry |= !(*carry);
  } else if (xv || yv) {
    bit->v |= !(*carry);
  } else {
    bit->v |= *carry;
    *carry = 0;
  }
}
```

Listing 4: Helper-function for addition

Furthermore, we have implemented another helper function `populate_bit_queue` to prepare an array of type `bit_operation_queue_t`, which is used to hold the parameters for each bit's addition. The struct holds a pointer to the `carry` and each individual bit in our result bits8-struct, and also holds the addition-bits `xv` and `yv` as booleans, which are the values to be used for the addition.
Lastly, the `bits8_add` function initializes both the `carry` and the result-bit to 0. It declares a `bit_operation_queue_t` structure to store the parameters for each bit operation. The function populates the queue with the addition parameters for all 8 bits and iterates over it, performing addition for each bit using the helper function `add_helper`, as shown in the code below:

```
struct bits8 bits8_add(struct bits8 x, struct bits8 y) {
  // ...initializing values
  bit_operations_queue_t queue[8];
  populate_bit_queue(queue, &carry, &result, x, y);
  for (int i = 0; i < 8; i++) {
    add_helper(queue[i].c, queue[i].b, queue[i].xv, queue[i].yv);
  }
  return result;
}
```

Listing 5: The `bits8_add` function.

3.1 Implementing negation

The objective is to compute the two's complement negation of an 8-bit binary number. The function initializes an 8-bit variable, `result`, to 0, which will store the negated version of

the input `bits8 x`. The function flips each bit of the input `x` and updates the `result` variable accordingly. Finally, it adds 1 to the `result` to complete the two's complement operation. The implementation is shown below:

```
struct bits8 bits8_negate(struct bits8 x) {
  result.b0.v = !x.b0.v;
   ⋮
  result.b7.v = !x.b7.v;
  return bits8_add(result, bits8_from_int(1));
}
```

Listing 6: The `bits8_negate` function.

3.2 Implementing multiplication

The purpose of this function is to multiply two 8-bit numbers using bitwise arithmetic. The function begins by initializing an unsigned integer accumulator, `z`, to 0, which will hold the result of the multiplication. The two 8-bit inputs (`x` and `y`) are then converted to integers using `bits8_to_int`, simplifying the bitwise manipulations. We're given the naive approach for multiplication as:

$$z = \sum_{i=0}^{k-1} x \cdot y_i \cdot 2^i \quad \text{where } k = \text{ number of bits.}$$

When y_i-th bit is equal to 0, we have $x \cdot 0 \cdot 2^i = 0$, and when y_i-th bit is equal to 1, we have $x \cdot 1 \cdot 2^i = x \cdot 2^i$, this is why we shift x by i positions, because its equivalent to multiplying by 2^i. The shifted value is then converted back to a `bits8` structure and added to the accumulator, `z`, which is also seen below in the loop within `bits8_mul`:

```
for (int i = 0; i < 8; i++) {
    if (y_bits_as_int >> i & 1) {
      x = bits8_from_int(x_bits_as_int << i);
      z += bits8_to_int(x);
    }
}
```

Listing 7: Code Snippet from the `bits8_mul` function.

After processing all bits of `y`, the accumulator `z` contains the final product in integer form. This value is converted back to a `bits8` structure before being returned as the result.

4 Test-framework

This project introduces a custom modular test framework for validating the functionality of operations on a custom `bits8` data type. It is designed such that it can be easily ex-

tended to test other data types and operations, and is thus very flexible. The `assert_test` function evaluates test cases by comparing expected and actual values using function pointers for comparison (`Comparator`) and formatting (`Printer`). This abstraction enables the framework to be adaptable to different data types while ensuring concise test logic. The framework tracks results using the `TestResult` structure, which records passed and failed tests. Test outputs are formatted into a table-like structure for readability, displaying descriptions, values, and outcomes. The individual test functions e.g., `test_bits8_from_int` and `test_bits8_addition`, tests for the specific operations like conversions and arithmetic w.r.t. the `bits8` data type. The `main` function coordinates test execution, iterating through an array of test cases and summarizing results by printing to the output stream.

5 Answers to the specific questions

5.1 Handling Negative Numbers in `bits_add()` and `bits_mul()`

The test results confirm that both `bits_add()` and `bits_mul()` produce correct results for negative numbers in two's complement representation, as shown in Table 1 (test cases 25, 26, 49, 50, 51, 52, and 53 for addition and multiplication). The functions work correctly because `struct bits8` is a simple 8-bit structure, performing arithmetic without interpreting the number's sign. The sign of a number becomes relevant only during interpretation, handled by our `printers.c` functions, and not during arithmetic. `bits_add()` and `bits_mul()` perform arithmetic on raw bits without considering the MSB as a sign. Proper interpretation of the MSB in `printers.c` is crucial to correctly display signed values.

5.2 Implementing the `bits8_sub()` function

Implementing `bits8_sub()` would be done in a very similar way as `bits8_add()`, but in this case, we have a `borrow` instead of a `carry`. For each bit in `x` and `y` starting from the least significant bit, the operations can be categorized into these cases for $\texttt{xv} - \texttt{yv}$:

1. If $\texttt{xv} = \texttt{yv}$ and `borrow` is not set, no `borrow` is needed, and the `result-bit` is simply 0. However, if $\texttt{borrow} = 1$, then the `result-bit` must be set to 1, and the `borrow` remains 1, as we need to borrow.

2. If $\texttt{xv} > \texttt{yv}$ and `borrow` is not set, no `borrow` is needed, and the `result-bit` is simply `xv`. If $\texttt{borrow} = 1$, then the `borrow` is neutralized, setting both the `result-bit` and `borrow` to 0.

3. If $\texttt{xv} < \texttt{yv}$ and `borrow` is not set, a `borrow` is required, and both the `result-bit` and `borrow` are set to 1, as we are in 'debt' for the next operation. If $\texttt{borrow} = 1$, the `borrow` stays 1, and the `result-bit` must be 0, as the `borrow` propagates further.

6 Appendix

```
No  Description              | Expected      Got           Result
bits8_from_int:
1   (Convert 81)             | 01010001      01010001      ✓
2   (Convert 255)            | 11111111      11111111      ✓
3   (Convert 0)              | 00000000      00000000      ✓
4   (Convert 127)            | 01111111      01111111      ✓
5   (Convert -1)             | 11111111      11111111      ✓
6   (Convert -128)           | 10000000      10000000      ✓
bits8_from_int PASSED:


No  Description              | Expected      Got           Result
bits8_to_int:
7   (Convert 0x10100010)     | 162           162           ✓
8   (Convert 0x10101010)     | 170           170           ✓
9   (Convert 0x01010101)     | 85            85            ✓
10  (Convert 0x00000001)     | 1             1             ✓
11  (Convert 0x10000000)     | 128           128           ✓
12  (Convert 0x11111111)     | 255           255           ✓
13  (Convert 0x00000000)     | 0             0             ✓
bits8_to_int PASSED:


No  Description              | Expected      Got           Result
bits8_addition:
14  (1 + 1)                  | 00000010      00000010      ✓
15  (2 + 2)                  | 00000100      00000100      ✓
16  (4 + 4)                  | 00001000      00001000      ✓
17  (8 + 8)                  | 00010000      00010000      ✓
18  (16 + 16)                | 00100000      00100000      ✓
19  (32 + 32)                | 01000000      01000000      ✓
20  (64 + 64)                | 10000000      10000000      ✓
21  (100 + 100)              | 200           200           ✓
22  (101 + 101)              | 202           202           ✓
23  (102 + 102)              | 204           204           ✓
24  (103 + 103)              | 206           206           ✓
25  (100 + -50)              | 50            50            ✓
26  (-5 + -3)                | -8            -8            ✓
bits8_addition PASSED:


No  Description              | Expected      Got           Result
bits8_negate:
27  (Negate -1)              | 1             1             ✓
28  (Negate 1)               | -1            -1            ✓
29  (Negate 0)               | 0             0             ✓
30  (Double negate -1)       | -1            -1            ✓
31  (Negate -20)             | 20            20            ✓
32  (Negate 20)              | -20           -20           ✓
33  (Negate -60)             | 60            60            ✓
34  (Negate 40)              | -40           -40           ✓
35  (Negate -100)            | 100           100           ✓
36  (Negate 60)              | -60           -60           ✓
37  (Negate 116)             | -116          -116          ✓
38  (Negate 80)              | -80           -80           ✓
39  (Negate 76)              | -76           -76           ✓
40  (Negate 100)             | -100          -100          ✓
bits8_negate PASSED:


No  Description              | Expected      Got           Result
bits8_mul:
41  (5 * 20)                 | 100           100           ✓
42  (1 * 2)                  | 2             2             ✓
43  (2 * 2)                  | 4             4             ✓
44  (4 * 2)                  | 8             8             ✓
45  (8 * 2)                  | 16            16            ✓
46  (16 * 2)                 | 32            32            ✓
47  (32 * 2)                 | 64            64            ✓
48  (64 * 2)                 | 128           128           ✓
49  (-5 * 20)                | -100          -100          ✓
50  (5 * -20)                | -100          -100          ✓
51  (-5 * -20)               | 100           100           ✓
52  (5 * -1)                 | -5            -5            ✓
53  (-5 * -1)                | 5             5             ✓
bits8_mul PASSED:

Summary: 53 tests passed, 0 tests failed
```


Figure 1: Terminal output showcasing test case results, including expected values, actual outcomes, and pass/fail statuses.

6.1 Test-framework code

6.1.1 Printers.c

```
void to_int8_base10(void* value, char* buffer, size_t buffer_size) {
  snprintf(buffer, buffer_size, "%d", *(int8_t*)value);
}
```

Listing 8: A preview of a printer that populates a buffer with the string representation of a uint8.

6.1.2 Comparators.c

```
int uint8_bits8_comparator(void* expected, void* got) {
  uint8_t* exp = (uint8_t*)expected;
  struct bits8* actual = (struct bits8*)got;
  uint8_t actual_value = bits8_to_int(*actual);
  return (*exp == actual_value);
}
```

Listing 9: An preview of a comparator that compares a uint with a bits8 struct.

6.1.3 Struct definitions

```
typedef int (*Comparator)(void* expected, void* got);
typedef void (*Printer)(void* value, char* buffer, size_t buffer_size);

typedef struct {
  int passed;
  int failed;
} TestResult;
```

Listing 10: Definitions for `Comparator` and `Printer` function pointers, which are used for comparing expected and actual values, and for formatting these values as strings, respectively.

6.1.4 Assert test function

```c
int assert_test(/*...params omitted for clarity*/) {
  // ...initializing variables
  int succeeded = compare(expected, got);

  if (succeeded) {
    result_icon = CHECKMARK;
    result_color = GREEN;
    result->passed++;
  } else {
    result_icon = CROSSMARK;
    result_color = RED;
    result->failed++;
  }

  char expected_str[32];
  char got_str[32];

  print_expected(expected, expected_str, sizeof(expected_str));
  print_got(got, got_str, sizeof(got_str));

  char parenthesized_description[40];
  snprintf(parenthesized_description, sizeof(parenthesized_description), "(%s)",
           description);

  printf("%-3u %-25s | %-14s %-14s %s%s%s\n", test_case,
         parenthesized_description, expected_str, got_str, result_color,
         result_icon, RESET);
  return !succeeded;
}
```

Listing 11: Implementation of the `assert_test` function, which evaluates a test case by comparing expected and actual values using a `Comparator`, formats the results using `Printer` functions, and tracks the outcome in a `TestResult` structure.