Assignment 5: Arithmetic and Testing

1DT056: Programming Embedded Systems
Uppsala University

March 18th, 2014

You can achieve a maximum number of 20 points in this assignment. At least 8 out of 20 points are required to pass the assignment. You will need to achieve a minimum of 60 points across all six assignments. Earning 80/90/100/110/120 points will earn you 1/2/3/4/5 points in the final exam. Assignments are to be solved by students *individually*.

Exercise 1 Fixed-point arithmetic (10p)

Below you can see the definitions for a basic version of signed fixed-point arithmetic, using a 32-bit significand and four digits after the comma:

```
typedef s32 FIA;          // s32: signed 32bit integers,
                          // defined in stm32f10x_type.h

#define P 4               // binary digits after comma

#define FROM_FLOAT(x)  (FIA)(((double)x) * (1 << P))
#define FADD(x, y)     ((x) + (y))
#define FSUB(x, y)     ((x) - (y))
#define FMUL(x, y)     (((x) * (y)) >> P)

void print(FIA x) {
  printf("%f\n", ((double)x) / (1 << P));
}
```

1. Determine the precise value range (the subset of the real numbers $\mathbb{R}$) that is represented by the fixed-point datatype.
2. As discussed in the lecture, the chosen definition of multiplication FMUL has the problem that the intermediate result (x) * (y) can overflow, even if the final result could correctly be represented using the 32-bit fixed-point datatype. E.g., the correct result of the computation

$$1\,000 \cdot 60\,000 \;=\; 60\,000\,000$$

can be represented, but our naive definition FMUL will produce a negative number.

Define two improved versions of FMUL ...

1. one using the signed 64-bit integer datatype "**long long**" for intermediate computations,
2. one using the decomposition

$$x \;=\; (\underbrace{(x \gg P)}_{=x_l} \ll P) + \underbrace{(x \;\&\; ((1 \ll P) - 1))}_{=x_r}$$

into the digits x_l before and the digits x_r after the comma. This decomposition is helpful, since the product

$$x \cdot y \;=\; ((x_l \ll P) + x_r) \cdot ((y_l \ll P) + y_r)$$

can be simplified using the standard laws of arithmetic (like distributivity) in such a way that problematic intermediate overflows can be avoided, even if all operations are carried out in 32-bit arithmetic. Justify the correctness of your implementation. It is sufficient if your implementation works for $P = 4$.

Because this implementation is likely to be more involved than the other fixed-point operations, it is a good idea to formulate it as an ordinary function

```
FIA FMUL_decomp(FIA x, FIA y) { ... }
```

rather than defining it as a macro.

Test your two implementations to ensure that they produce the same results, and that internal overflows have effectively been prevented.

3. A naive way to add a division operation to our fixed-point datatype is the following definition:

```
#define FDIV(x, y)      (((x) << P) / (y))
```

Recall that we require all fixed-point operations to round downwards: the result produced by a fixed-point operation $(x \tilde{\text{op}}\ y)$ shall be the greatest fixed-point number not larger than the precise mathematical result $(x\ \text{op}\ y)$. Investigate whether this is the case for FDIV, in particular also taking negative operands into account. Argue that FDIV indeed rounds correctly, or show cases where FDIV produces incorrect results and provide a corrected implementation of FDIV.
You do not have to take internal overflows into account in this question: in particular, you can assume that the expression (x) << P will not overflow.

4. Briefly summarize the differences betweeen fixed-point and floating-point arithmetic. Which is preferable for use in embedded systems and why?

Exercise 2 Unit testing a stateful module (10p)

Stateful modules are modules with internal state, which means that the behaviour of such components can depend on earlier interactions with the module. Testing a stateful module requires particular care, since the module first has to be brought in the right state for testing, and has to be reset after executing a test case and put back into its initial state. As an additional complication, after executing the test it is not easily possible to access the pre-state of the module anymore.

This means that an executable unit test case for stateful modules has the following general structure:

```
int testCase(void) {
  // set up module to be tested; prepare inputs (1)

  // call functions of module                   (2)

  // assess results of testing                  (3)

  // reset module                               (4)
  return testResult;
}
```

As an example, we consider a binary search tree datastructure given in file (`http://www.it.uu.se/edu/course/homepage/pins/vt15/search_tree_original.c`). This module is stateful, since the search tree is stored in form of global variables, and a side effect (in fact, the main effect) of the function "insertNode" is to modify the search tree.

1. First derive a general test oracle that checks whether the search tree is well-formed. A search tree is well-formed if, for any node n in the tree, the nodes reachable through $n.left$ carry data strictly smaller than $n.data$, and the nodes reachable through $n.right$ carry data strictly larger than $n.data$. Implement this oracle as a Boolean function

   ```
   int treeIsWellformed(void) { ... }
   ```

2. Implement a function

   ```
   int contains(int d) { ... }
   ```

   that checks whether the search tree currently contains the datum d.

3. Write one or multiple executable test cases that achieve full decision coverage for the function "insertNode", considering the decisions

   ∗currentPtr,
   n−>data < currentNode−>data,
   n−>data > currentNode−>data.

   We assume that the search tree is initially empty, i.e., root is NULL. At point (1) in the test case, the search tree can be filled with some initial amount of data. At (2), one or multiple invocations of "insertNode" are performed. At (3), the result of (2) is checked by invoking the function "treeIsWellFormed", and by checking that all data inserted at (2) is in fact present in the tree (using the function "contains"). At (4), the tree is reset and emptied by removing all nodes from it.
   Justify why your test suite achieves decision coverage.

Submission

Solution should be submitted via the Student Portal at latest on
Sunday, March 7th, 2015.