Project 1

Prof. Dr. Sebastian Hack
Pascal Lauer, M. Sc.
Marcel Ullrich, B. Sc.

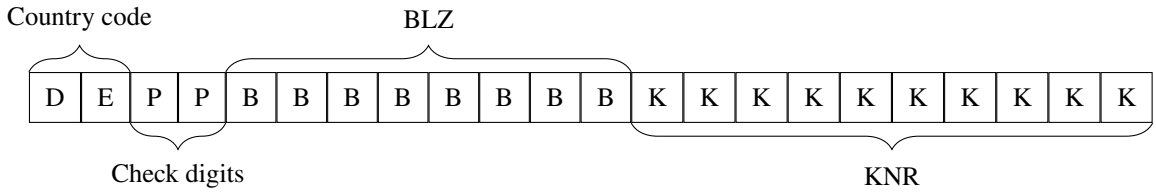
Preparation

To be able to edit the project in Mars, you first have to checkout the repository and import the project:

1. Clone the project in any folder:
   `git clone ssh://git@git.prog2.de:2222/project1/$NAME.git`
   where $NAME has to be replaced with your CMS username.
2. Open the cloned directory in Mars.

Check that the settings *Assemble all files in directory* and *Initialize Program Counter to global 'main' if defined* in the *Settings* menu in MARS are activated to compile the file `src/main.asm` and set the program entry point to the label `main`.
In the following document, we refer to the term character as a single byte encoding of a character according to the ASCII standard. The IBANs only contain ASCII upper case letters and digits. The KNRs and the BLZs consist of ASCII digits only.

IBAN-Calculator (10 Points)



In this project, you will implement an IBAN calculator that converts a bank account number (KNR) and a bank code (BLZ) to a German IBAN according to the format shown above, or extracts the KNR and the BLZ from a German IBAN. You can find a general introduction to IBANs in the Wikipedia (`https://en.wikipedia.org/wiki/International_Bank_Account_Number`).
You will write the program in MIPS assembler. To make the project easier to handle, it is divided into several tasks. Each task has its own file in which you will write the corresponding subroutine. Please also pay attention to the hints at the end of the project description.

1 Extract BLZ and KNR (2 Points)

Write a subroutine `iban2knr` (file `src/iban2knr.asm`), which extracts the BLZ and the KNR from a German IBAN. Your subroutine receives the following arguments:

1. A buffer with a German IBAN with 22 characters.
2. The target buffer for the BLZ with space for 8 characters.
3. The target buffer for the KNR with space for 10 characters.

2 Remainder calculation (4 Points)

Write a subroutine `modulo_str` (file `src/moduloStr.asm`), which calculates the remainder of a number given as a string. Your subroutine receives the following arguments:

1. A buffer with a number.

2. The length of the number in the buffer.
3. The divisor.

The result is returned in the return register. The number in the buffer is the remainder of the division of the number represented in the buffer by the divisor. The number in the buffer is a decimal number greater than zero represented as a string of digits and has no sign. The least significant digit of the number is at position $length - 1$ in the buffer. We advise you to calculate the result digit by digit according to the Hornerschema. Take care that the result is also correct if the number represented in the buffer exceeds the capacity of the machine registers. This can be achieved by clever use of the following equations (x mod y is the remainder of the division of x by y):

$$(a + b) \bmod c = (a + (b \bmod c)) \bmod c$$

$$(a \times b) \bmod c = (a \times (b \bmod c)) \bmod c$$

You can also reduce intermediate results modulo the divisor.

3 Validate IBAN check digits (2 Points)

Write a subroutine `validate_checksum` (file `src/validateChecksum.asm`), which validates the check digits of a German IBAN. Your subroutine receives the following argument:

1. The start address of a 22 character long buffer with a German IBAN.

The subroutine returns the result of the following calculation in the return register. The following steps show how the result is calculated.

1. Move the first four characters to the end of the IBAN.
2. Replace all letters with numbers, where A = 10, B = 11, … , Z = 35.
3. Calculate the remainder of the division by 97. This is the return value.

An example:

IBAN:	DE68	210501700012345678		
Reordering:		210501700012345678	DE	68
Digit substitution:		210501700012345678	1314	68
Remainder:		210501700012345678	1314	68 mod 97 = 1

You can use the subroutine `modulo_str` from the previous task to calculate the remainder.

The check digits are valid if the remainder is 1.

4 Generate IBAN (2 Points)

Write a subroutine `knr2iban` (file `src/knr2iban.asm`), which calculates a German IBAN with valid check digits from a KNR and a BLZ. Your subroutine receives the following arguments:

1. The target buffer for the IBAN with space for 22 characters.
2. The BLZ as an 8 character long buffer.
3. The KNR as a 10 character long buffer.

To calculate the check digits, the function from the previous task can be used. For this, the 2nd and 3rd character of the IBAN are set to 0. The check digits are then the difference of 98 and the result of the validation function. A single digit check digit is prefixed with a 0.

5 Validate IBAN check digits without intermediate memory (2 bonus points)

Write a variant of `validate_checksum` (file `src/bonus/validateChecksum.asm`), which validates the IBAN check digits without writing intermediate results to memory. The subroutine must return the same result as `validate_checksum` from task 3. In particular, any syscall or memory write disqualifies the solution.
Do not change the data segment, define macros, or define multiple text segments in the file `src/bonus/validateChecksum.asm`. To qualify for the bonus points, all other tests must pass as well.

6 Evaluation

For every task there are *public*, *daily* as well as *eval* tests. The *public* tests are available in your local repository and can be executed at any time. The next section will detail how to run those tests. You have to pass all *public* tests for a task to get any points for that task. Note that a task can contain multiple subtasks, as seen with task **??**. Every public test of every subtask must be passed.
The *daily* tests are not available locally. These are scheduled to run after you upload your project to our server (`git push`). After running the tests you will be notified via e-mail and can lookup the results in the CMS. Based on the limited server capacity, it might take a while before the tests are run. We only guarantee that you get the tests results once per day.
Finally, the *eval* tests will be used to rate the projects upon completion. These tests are unavailable to you.

7 Executing Tests and Debugging

Execute the command `./run_tests.py` in the root directory of the project to test your implementation. By default all public tests in the directory `tests/pub` are run. To run only specific tests, use the option `-t test_name_1 ...test_name_n` to run only the tests with names `test_name_1` through `test_name_n`. For example, execute `./run_tests.py -t test_check_win1` to run the public test `test_check_win1` only. To list all test names, use the `-l` option: `./run_tests.py -l`.
The output of the tests is colored to make the results more clear. This should display correctly in the majority of terminals, including the integrated terminal of Visual Studio Code. If the colored output poses a problem for the terminal you use, you can disable it with the option `-nc`.
If a test fails, you can debug it in MARS. The command `./build_testbox tests/pub/test_X.asm` copies the test as well as your implementation into the folder `testbox/`. From there you can start the test using MARS. Note, that only the files given in the task descriptions are included in the evaluation and the automated tests. Therefore do not commit additional files, especially not in the `testbox/` folder.

Self-written Tests

We recommend writing your own tests to supplement the given *public* tests. You can put your own test files into the folder `tests/student`.
A test comprises two files, `test_name.asm` contains the test file while `test_name.ref` contains the expected console output of the test. The printed output of the test is checked against the content of the reference file, *including newlines and spaces*. If the content matches, the test is passed. Please use the public tests as a reference.
With the option `-d dir_1 ... dir_n`, you can run all tests contained in the subdirectories `dir_1` through `dir_n` of `tests/`. For example, with `./run_tests.py -d pub student`, you can run all public tests and all of your own tests in sequence. The option can be combined with the `-t` option. Use `./run_test.py -d student -t test_X` to selectively run the self-written test `test_X`.

Remember that when consecutively printing two integers with a `syscall`, no space is produced between them automatically, which for example leads to the ambiguous output “11” if the value “1” is printed twice. Make sure you format the printed output appropriately when writing your tests.

Scripts for the bonus task

Analog to the regular test programs, there is the script `build_bonusbox` with which you can debug your solution to the bonus task. The script `build_bonusbox` also copies the project with the implementation of the bonus task to the folder `bonusbox/`, where you can start it with MARS. You can invoke `run_tests.py` with the argument `-b` to run the tests with your bonus task implementation.

Submission of Solutions

git push until 09. May 2023, 23:59.

Hints

1. The file `src/util.asm` contains useful subroutines that you can use for this project. Some of the subroutines implemented there should be familiar to you from the MIPS exercise sheet. You should not change this file as we will use the original version for the automatic tests.
2. Remember that your subroutines also have to implement the calling convention.
3. For your submission, only the files in the folder `src/` whose names are given in the task description will be considered.
4. Please note that only the version of MARS installed on the virtual machine is supported by us. This differs from the official versions in some details.
5. We point out that no guarantees can be given for the generated IBANs, BLZs and KNRs.
6. The description of the procedure for calculating the check digits is largely based on the proposed method (`http://de.wikipedia.org/wiki/IBAN#Validierung_der_Pr.C3.BCfsumme`) for verifying the check digits described on Wikipedia.
7. Document your program with code comments. This makes the code more readable and helps you to debug your program in MARS.

Problem Debugging