

# Assignment 1: Assembly

---

**Due** 1 May 2022 by 20:00      **Points** 12      **Submitting** a file upload  
**Available** until 1 May 2022 at 21:00

---


This assignment was locked 1 May 2022 at 21:00.


This week's assignment consists of three programming tasks. Each given equal weight.

In the the first task you must write a small GDB script and try to make sense of a given assembly program. In the two last tasks you must write assembly code for an x86-64 Linux system. Your solution for all tasks must work in the course VM.

You should write your assembly programs using the Intel syntax. (If you really, really, really want to use the AT&T syntax, then you need to get an OK from your TA.)

## What to hand in

You should hand in a `handin.zip` file that contains your (well commented) assembly code, `Makefile` and other support files; and a file with your report (`week1.txt`, `week1.md` or `week1.pdf`). Use the `week1-assembly-assignment.zip` (<https://absalon.ku.dk/courses/56922/files/5925212?wrap=1>)  ([https://absalon.ku.dk/courses/56922/files/5925212/download?download\\_frd=1](https://absalon.ku.dk/courses/56922/files/5925212/download?download_frd=1)) file as a starting point.

Make sure to check your `handin.zip` file with **OnlineTA**  (<https://pcs.incorrectness.dk>) before handing in; otherwise you might risk to get your hand-in auto-failed.

## Task 1: Understand a strange program

You are given the assembly program `strange.asm`, your task is to to make sense of this program (and to get some experience with GDB).

Make a GDB script in the file `strange_debug.gdb`. The script should do the following:

- Set a break-point at the label `outer`
- Use the `commands` command to make sure that `rdi` is examined as a string each time the program is at `outer`.
- Set a break-point at the label `update`
- Use the `commands` command to find out what the `rax` and `rbx` registers contain each time program is at `update`.
- Log the result of running the script to a file called `strange_gdb_debug.output`

It should be possible to run your script with the command:

```
$ gdb --command=strange_debug.gdb ./strange
```

This should result in the file `strange_gdb_debug.output`.

In your report you should give a *high-level* explanation of what `strange` does and how it works, based on your reading of `strange.asm` and the debug output.

*Hints:* Try to experiment with different formats when printing `rax` and `rbx`.

## Task 2: Quaternary numbers

Convert a quaternary number, represented as an ASCII string, with `'A'`, `'B'`, `'C'` and `'D'` as base numbers (e.g., `"BADCAB"`, `NUL` terminated), to an unsigned 64-bit integer.

Your code must not use any library functions. Likewise, your code should only make one pass over the input.

To get full points your code should be able to handle the following special cases:

- strings specifying an invalid quaternary number should be converted to the value 0.
- you should allow prefixed `'A'`s (zeros), that is `"AAAAB"` denote the same value as `"B"`.

Quaternary numbers use base-4. That is, we use four symbols: `'A'`, `'B'`, `'C'` and `'D'` for the base values 0, 1, 2 and 3. The last place in a quaternary number is the 1's place ( $4^0$ ). The second to last is the 4's place ( $4^1$ ), the third to last is the 16's place ( $4^2$ ), and so on.

```
; "BADCAB"
      B      A      D      C      A      B      ; the number
1*4^5 + 0*4^4 + 3*4^3 + 2*4^2 + 0*4^1 + 1*4^0      ; the value
1024 +      0 +      192 +      32 +      0 +      1 = 1249
```

Make a file `quaternary.asm` that defines a global symbol `quaternary_convert` for a function that can convert quaternary number. The function should follow the standard calling convention, that is:

- input address of input string is in `rdi`
- the return address is at `[rsp]`
- the result should be in `rax`
- the callee-save registers should be preserved. That is, `rbp`, `rbx`, `r12`, `r13`, `r14`, `r15` should have the same value when the function returns as when it was called.

The file should not define any other global symbols than `quaternary_convert`, as that may break our testing framework. But you may of course use local functions/labels if you want.

## Task 3: Get a line

Make function `getaline` that reads a line of text from a C file stream and save it (including the newline and NUL terminated) to a given memory buffer.

Your function should ensure that a line of text is at most 96 characters long. That is, if a `'\n'` is not encountered after 96 characters the line is considered ended and no further bytes are read. Likewise, you should always ensure that you return with a string that is terminated with `'\n'` and NUL (`'\0'`) in the buffer. You can assume that the buffer is large enough to hold 98 bytes (to handle the special case that after 96 non-newline bytes you have to insert both a newline and a NUL byte). Likewise, we are only concerned with ASCII files.

The only external function you may use is `fgetc(3)`. Note that `fgetc(3)` returns `-1` (on Linux) when it encounters the end of file.

Your function should return how many bytes it *read* from the file. That is, if you read a `'\n'` in the first 96 bytes it should be counted, otherwise it shouldn't. In other words your function should always return an integer between 0 and 96 (both inclusive).



Make a file `getaline.asm` that defines a global symbol `getaline` for a function that can read a line of text from a C file stream. The function should follow the convention that:

- A pointer to an open file stream (`FILE *`) is given in `rdi`, and a pointer to the buffer is given in `rsi`.
- the return address is at `[rsp]`
- the result should be in `rax`.
- the callee-save registers should be preserved. That is, `rbp`, `rbx`, `r12`, `r13`, `r14`, `r15` should have the same value when the function returns as when it was called.

The file should not define any other global symbols, as that may break our testing framework. But you may of course use local functions/labels if you want.

## Reporting

For each task you should clearly state how much of the task you have solved. Shortly outline the structure of your code to make it comprehensible for your TA.

Test your code with [OnlineTA](https://pcs.incorrectness.dk/)  (<https://pcs.incorrectness.dk/>) before handing in. If [OnlineTA](https://pcs.incorrectness.dk/)  (<https://pcs.incorrectness.dk/>) has complaints make sure to address these in your report.

For each of your functions explain which registers you use and for what, and likewise if you use any memory (like the stack). Explain which challenges you encountered and what you did to overcome them.


Explain how you have tested your code (or why you haven't tested it).

## Hints

1. Solve the lab exercises.
2. In a terminal type each of the following and press enter:

```
man man  
man ascii  
man 3 fgetc  
man 3 puts
```

### 3. To keep your TA happy:

- Keep you reporting to one or two pages,
- Don't hand in a Word document, if you need fancy formatting and pictures then use PDF; otherwise hand in a text file.
- Use the handed out skeleton code.
- Pretty please test your code with [OnlineTA](https://pcs.incorrectness.dk/)  (<https://pcs.incorrectness.dk/>) before handing in.