Prof. Dr. Rolf Drechsler, drechsler@uni-bremen.de, MZH 4330
Dr. Muhammad Hassan, hassan@uni-bremen.de, MZH 4280
Milan Funck, mfunck@uni-bremen.de, MZH 4260 Jan Zielasko, zielasko@uni-bremen.de, MZH 4184

2. Exercise sheet of the lecture

Computer architecture and embedded systems

Rechnerarchitektur und Eingebettete Systeme

For all tasks, unless specified otherwise, holds:

- Non-compiling or undocumented code is considered as *not passed*. Adequate documentation is necessary for passing the exercise.
- Assembler programs shall conform to the (minimal) RISC-V calling convention of holding (32 bit) parameters in the registers **a0** ... **a7**, and returning a value in register **a0**. Caller shall use `JAL`, callee shall use `RET`. Usage of the stack is optional.

You should start from the provided `tiny-riscv-exercise` source tree. In the source tree, hints are given where to start with programming on the individual exercise tasks. For building *cross compiled* software, you will need *CMake*, patience, and other applicable buildtools to build the provided *GNU Compiler Toolchain*[1], depending on your host system.

Submission: This exercise will not be submitted. Instead you will demonstrate the runnig solutions in the last tutorial on **28. January 2025**. Still, all relevant files, including a PDF of the textual answers, are to be archived using `zip` or `gzip` formats and sent no later than **28. January 2025** to `mfunck@uni-bremen.de` with the subject "*Submission Exercise Sheet 2 Group X*", where X is your group number.

Exercise 1

a) Add all missing `I` Opcodes in `opcodes.h` and `iss.cpp`. Use the software-examples *asm-example* and *c-example* as tests. What do they calculate? Which one will be faster to program, which one is faster in runtime?

b) Write an assembler program that multiplies two numbers (located in **a0** and **a1**) using only `I`-instructions (except the `CSRs`). What is the (runtime) complexity of your program?

Exercise 2

Add the syscalls `printInt` (1), `printChar` (2) and `printString` (3). Write a C-program that uses these virtual syscalls to print something nice. You may use these syscalls or custom ones in the other exercises if it helps.

Exercise 3

Next up, we will look at the `M` instruction set:

a) Add all `M` instructions to your instruction set simulator.

b) Write the same program as in Task 1, but using the `M` instruction set. What is the runtime complexity of your program now? Would it be the same on a real-life processor?

c) Write a new assembler program that factorizes the registers **a0** and **a1** in the way $a_0 = {a_0}^{a_1}$. How many instructions does your ISS take to calculate 7^{10}?

[1] https://github.com/cirromulus/riscv-gnu-toolchain

Exercise 4

Using Hardware/Software Co-design concepts:

a) Add a peripheral device named *CharacterPrinter* that is accessed by memory-mapped IO in the range of $\texttt{0x90 0000}_{16}$-$\texttt{0x90 0008}_{16}$ with a register width of 4 byte, based on the following memory map (Table 1). It is ok if misaligned access is not supported.
b) Write a C-program that reads from this device and prints out 128 characters of every mode.

Address	Name	R/W?	Value
`0x0`	**A**	R	32 bit value based on register **B**
`0x4`	**B**	R/W	Selector based on Table 2

Table 1: Memory map of the *CharacterPrinter* peripheral.

Register B	Register A
0	Random 32 bit values
1	Random printable ASCII characters
2	The sequential characters of the first chorus of Rick Astley's *Never Gonna Give You Up*[2]
3	The constant 1337_{10}
else	The constant 0

Table 2: Outputs of register **A**, depending on value of register **B**.

[2] https://www.youtube.com/watch?v=dQw4w9WgXcQ *"XcQ, der Link bleibt zu"*