MTH410E – RISC-V Architecture and Processor Design

Assignment 2: RiscV Core in SystemVerilog

Deniz Bashgoren 040180902

1. Introduction

This project is a RiscV core written in SystemVerilog, that implements RV32I subset of the ISA. All instructions except FENCE, EBREAK and ECALL are supported. The core is synchronous to an external clock signal, executes in-order, doesn't utilize a pipeline, doesn't support interrupts and memory virtualization techniques.

2. Source Code Organization

The main folder contains the following:

- **src/** : The folder with the source code. The top module is riscv_singlecycle.sv.
- **tb/** : The testbench modules. riscv_singlecycle_tb.sv is the module controlling the entire core. The other testbenches are there to test the various parts of the core. Running each of them should print "No errors" on the screen.
- **samples/** : The folder containing simple C programs to test the core. Each sample consists of a C source file (like prime.c), an elf executable file, compiled by GCC (prime.c.elf), an disassembler output file (prime.c.dump), and hex files ready to be imported by SystemVerilog, extracted from the elf file (prime.c.elf.imem and prime.c.elf.dmem).
- **utils/** : A script written in javascript that generates *.imem and *.dmem files from a given *.elf file. To parse the elf file it uses the library node-elf-file [1].
- **makefile** : The intended way to build the project.
- **compilation.ld** : A linker script used to reorder the memory mapping of the sections.
- **cpu.gtkw** : A GTK Wave configuration file. When it's run, it displays all the important wires by reading riscv_singlecycle.vcd file that is generated.
- **Report.pdf** : This file.

2. Required Software

Here it's assumed that the user runs a GNU/Linux machine that has access to basic tools like make. The software needed to run the core:

- **GNU Compiler Collection for RiscV** : This is used to compile the sample C files to RiscV binary files. Run the **`riscv64-linux-gnu-gcc`** command to make sure it's installed.
- **NodeJS** : This is a javascript runtime that is used to run the scripts to prepare the *.dmem and *.imem files. Run the **`node`** command to make sure it's installed.
- **Icarus Verilog** : The SystemVerilog simulator used for this project.
- **GTK Wave (optional)** : The program reading *.vcd files and showing the signal waveforms.

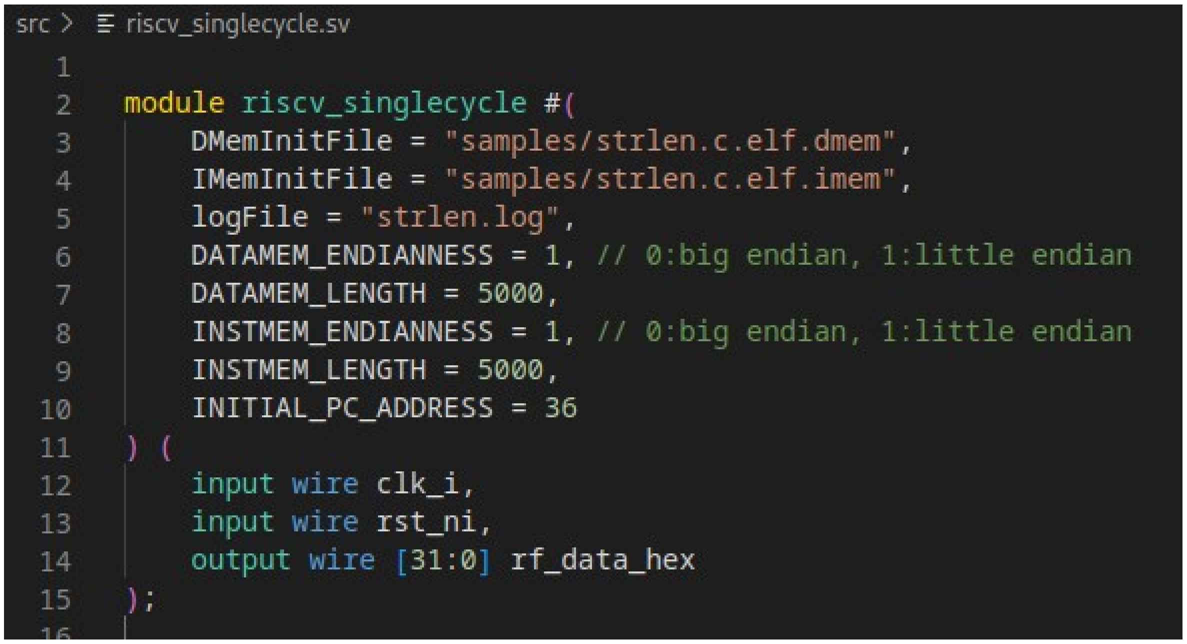
3. How to Run the Existing Tests

First let's test the various parts of the core to see if they contain errors. Make sure you are at the root of the project directory, and execute the following commands one by one:

```
make regfile
make datamem
make instmem
make immedgen
```

If everything goes well, you should see “No errors” text after each of these commands.

For this example, we will test the samples/strlen.c program with the “Hello world!” string, which should output 12. All the sample programs already have *.dmem and *.imem files, so we will use those. For the cases where these files need to be generated, see section 4.
Before building the project, we need to use a text editor to edit the file src/riscv_singlecycle.sv and set the correct parameters.



src > riscv_singlecycle.sv

```

module riscv_singlecycle #(
    DMemInitFile = "samples/strlen.c.elf.dmem",
    IMemInitFile = "samples/strlen.c.elf.imem",
    logFile = "strlen.log",
    DATAMEM_ENDIANNESS = 1, // 0:big endian, 1:little endian
    DATAMEM_LENGTH = 5000,
    INSTMEM_ENDIANNESS = 1, // 0:big endian, 1:little endian
    INSTMEM_LENGTH = 5000,
    INITIAL_PC_ADDRESS = 36
) (
    input wire clk_i,
    input wire rst_ni,
    output wire [31:0] rf_data_hex
);
```

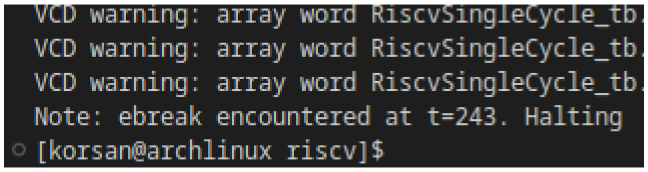
Here, the parameters are:

- **DMemInitFile** : The path to the dmem file, which is a human-readable file containing hex code, that will be uploaded to the data memory unit at t=0. Each hex number represents a byte and each byte will be uploaded to the data memory starting from address 0. This field can be left empty if you want all the bytes to be initialized to zero.
- **IMemInitFile** : The path to the imem file, which is a human-readable file containing hex code, that will be uploaded to the instruction memory unit at t=0. Each hex number represents a byte and each byte will be uploaded to the instruction memory starting from address 0. The bytes in this unit will be immutable during the simulation.
- **logFile** : The name of the file where the simulation will output the logs. It will be generated in the project directory.
- **DATAMEM_ENDIANNESS** : This will determine how bytes in the data memory will be transferred to and from the registers. This core fully supports both LE and BE.
- **DATAMEM_LENGTH** : This will determine the capacity of data memory. Measured in bytes. Make sure the length is enough to hold the stack, which grows downwards. For most samples, 5000 will be enough.
- **INSTMEM_ENDIANNESS** : This will determine how to interpret the *.imem. This core fully supports both LE and BE. Note that normally the instructions are LE as per

the spec[2], but the GCC objdump program reverses the order of the bytes in its output. Since all the instructions this processor supports are 4 bytes long, the order of the bytes doesn’t affect the readability. If you use GCC objdump, keep the endianness at 1 (LE).

- **INSTMEM_LENGTH** : This will determine the capacity of instruction memory. Measured in bytes. For most sample programs 5000 will be enough.
- **INITIAL_PC_ADDRESS** : This tells which byte holds the first instruction to be executed. For the sample C programs, this corresponds to the address of the _start function. You can find it in the *.dump file of each sample program.

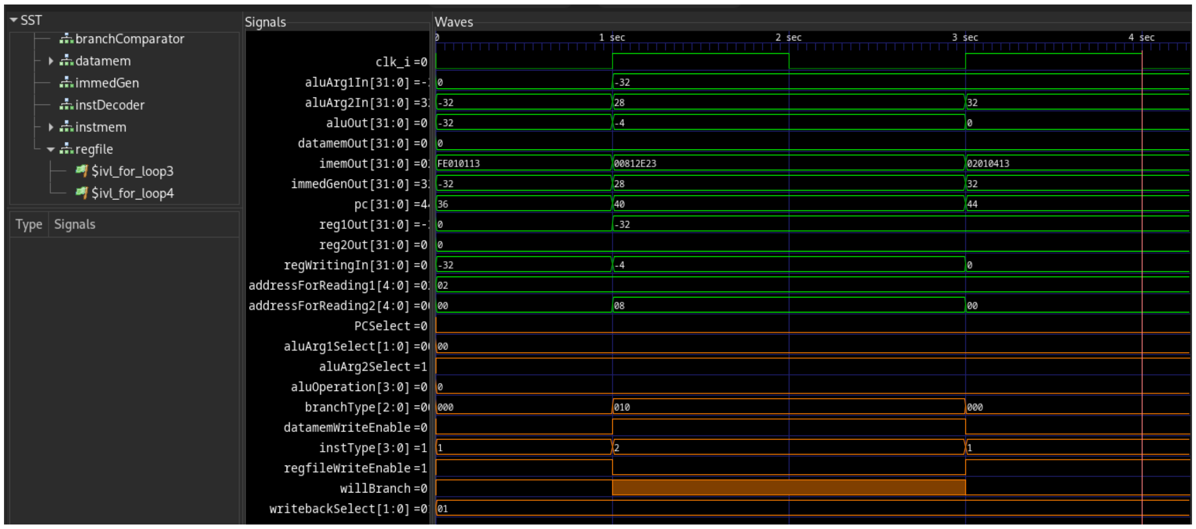
After setting the parameters correctly, after making sure we are in the project root directory, we can execute **`make cpu`**. This command will first compile the verilog source, then remove existing *.log files if any, and then run the executable. The reason it removes the log files is that the log file is opened in Append mode. This allows multiple modules to append logs to the log file concurrently.



```
 VCD warning: array word RiscvSingleCycle_tb.
 VCD warning: array word RiscvSingleCycle_tb.
 VCD warning: array word RiscvSingleCycle_tb.
 Note: ebreak encountered at t=243. Halting
○ [korsan@archlinux riscv]$
```

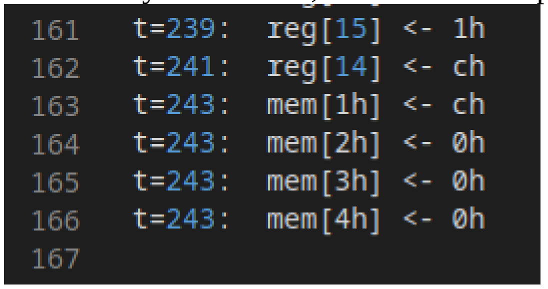
When the program reaches the EBREAK instruction, the simulator treats it as an exit function, and halts. When the program runs, 3 files will be generated in the root directory:

- **a.out** : The executable.
- **riscv_singlecycle.vcd** : The waveform of the simulation. We can run the cpu.gtkw file using GTK Wave and see the important signals:



Here the green signals are the wire interconnecting the big modules, and the orange signals are parts of the control word. The core is synchronous, so the values change only at posedge clock.

- **strlen.log** : The human-readable log file containing all the moments when registers or the memory is modified, as well as when pc jumps somewhere. If we look inside,

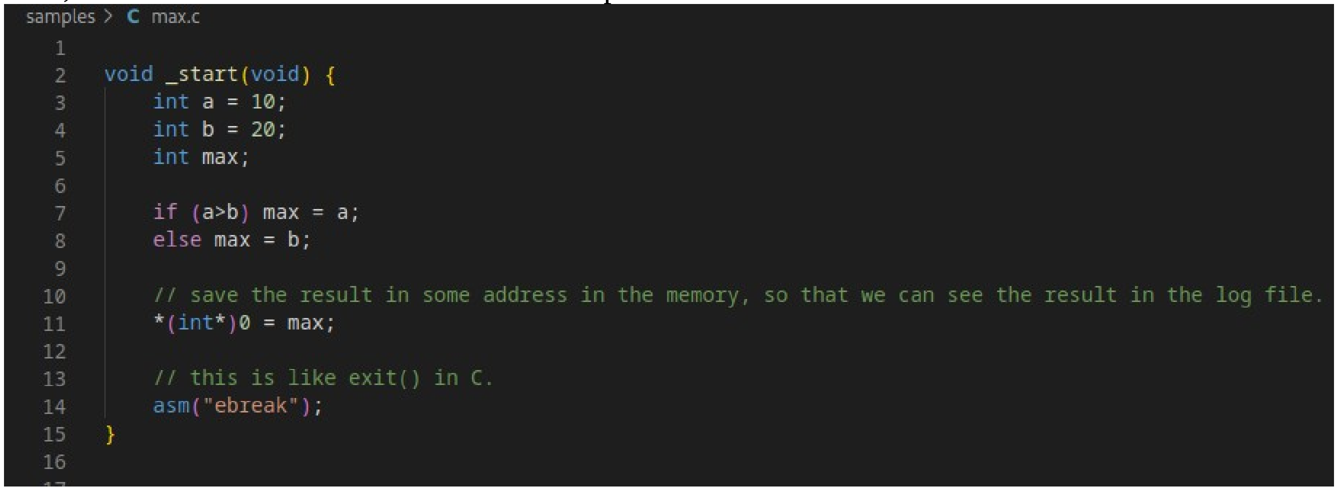


```
t=239:  reg[15] <- 1h
t=241:  reg[14] <- ch
t=243:  mem[1h] <- ch
t=243:  mem[2h] <- 0h
t=243:  mem[3h] <- 0h
t=243:  mem[4h] <- 0h
```

  we can see that at t=243 a 4-byte value of c (12 in decimal) is saved to the address 1, in little endian format. The logs show that the length is calculated correctly.

4. How to Write New Tests

First, let's create a new file "max.c" in the samples/ folder. The contents of the file:



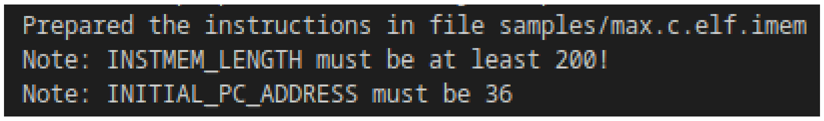
samples > C max.c

```c

void _start(void) {
    int a = 10;
    int b = 20;
    int max;

    if (a>b) max = a;
    else max = b;

    // save the result in some address in the memory, so that we can see the result in the log file.
    *(int*)0 = max;

    // this is like exit() in C.
    asm("ebreak");
}

```

Save the file and run the following command in the root directory of the project:

```
make prepare SRC=samples/max.c
```

Make sure the = doesn't have spaces on either side. After executing the command, we should see

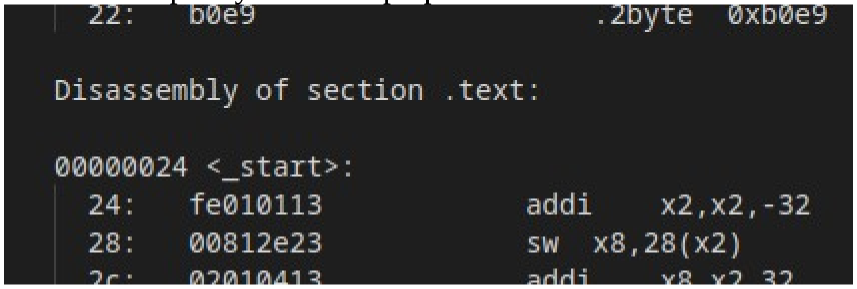


```
Prepared the instructions in file samples/max.c.elf.imem
Note: INSTMEM_LENGTH must be at least 200!
Note: INITIAL_PC_ADDRESS must be 36
```

In the samples/ folder, the following files are now created:

- **max.c.elf** : The binary. We don't use this file directly.

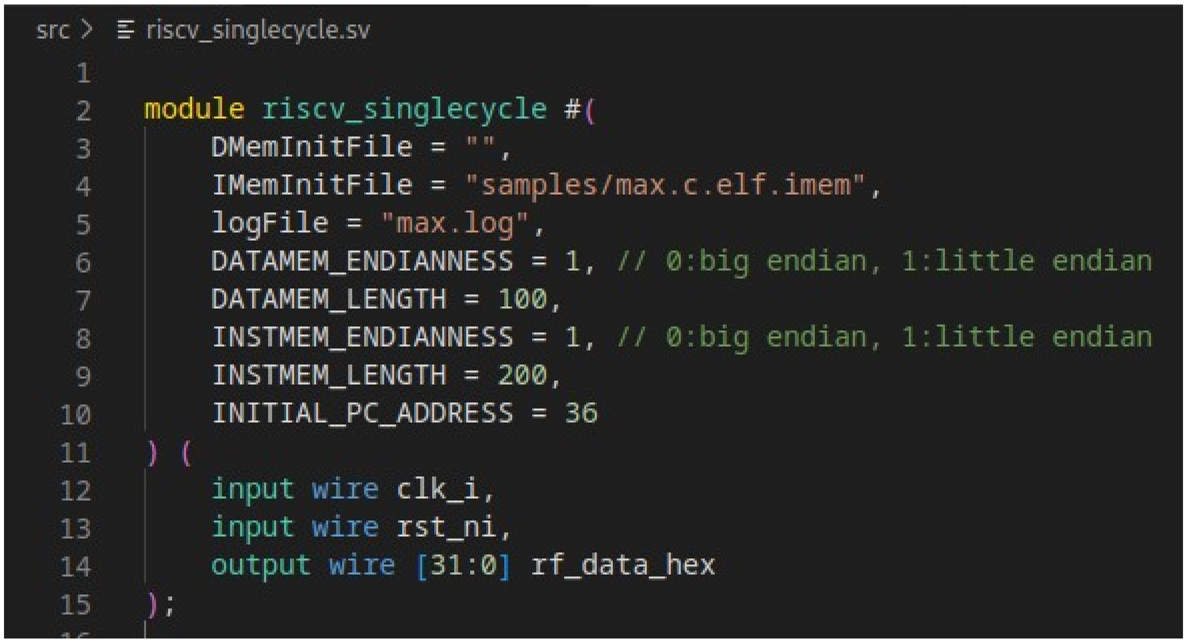
- **max.c.dump** : The output of the disassembler when the elf file is input. By analyzing it we can see that the address of _start is 24 in hex. This is equal to 36 in decimal, and it's the value output by the make prepare command.



```
  22:   b0e9                    .2byte  0xb0e9

Disassembly of section .text:

00000024 <_start>:
  24:   fe010113                addi    x2,x2,-32
  28:   00812e23                sw  x8,28(x2)
  2c:   02010413                addi    x8,x2,32
```

- **max.c.dump.imem** : The bytes of the .text section are placed here. This file can be input to the core.
- **max.c.dump.dmem** : Since we didn't use global variables or strings, this file is not generated. If we used those, we would see .data, .bss and .rodata sections would be generated and the bytes would be placed to the dmem file.

Next, we need to edit the src/riscv_singlecycle.sv file and set the parameters:

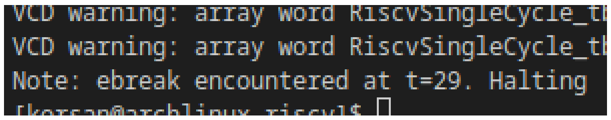


src > riscv_singlecycle.sv

```

module riscv_singlecycle #(
    DMemInitFile = "",
    IMemInitFile = "samples/max.c.elf.imem",
    logFile = "max.log",
    DATAMEM_ENDIANNESS = 1, // 0:big endian, 1:little endian
    DATAMEM_LENGTH = 100,
    INSTMEM_ENDIANNESS = 1, // 0:big endian, 1:little endian
    INSTMEM_LENGTH = 200,
    INITIAL_PC_ADDRESS = 36
) (
    input wire clk_i,
    input wire rst_ni,
    output wire [31:0] rf_data_hex
);
```

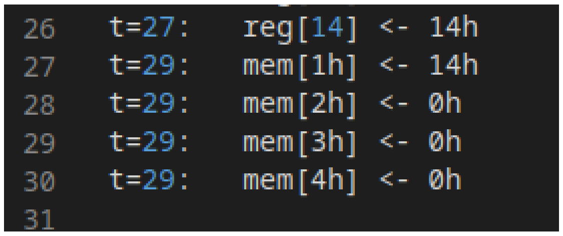
Here we left out the DMemInitFile field, because we don't have global variables. The values 128 and 36 need to agree with the values output by make prepare command above. 100 is used for Datamem length, because the main function will use the stack for its local variables.

Save the file and run the command **`make cpu`**. This will compile the source and start the simulation.



```
VCD warning: array word RiscvSingleCycle_tb
VCD warning: array word RiscvSingleCycle_tb
Note: ebreak encountered at t=29. Halting
[korsan@archlinux riscv]$ 
```

Here we see that the results are ready at t=29. Let's look inside the max.log file:



```
t=27:   reg[14] <- 14h
t=29:   mem[1h] <- 14h
t=29:   mem[2h] <- 0h
t=29:   mem[3h] <- 0h
t=29:   mem[4h] <- 0h
```

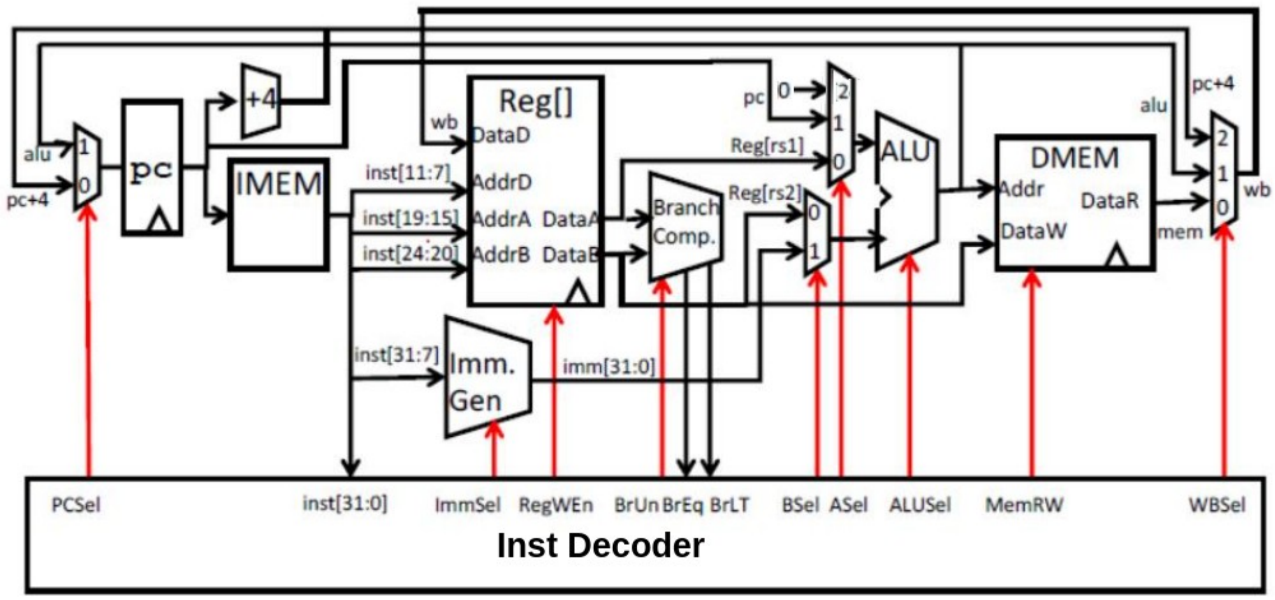
As expected, the value of 14h or 20 in dec is saved to the memory at address 1.

5. Internals

The register file (regfile.sv) holds 32 registers. The first one (memory[0]) is always kept at zero, as specified in RiscV specification[2]. There are two reading ports; two corresponding output ports; a separate write port (address and value); an enable; a clock, and a reset input. Reading works asynchronously. Writing works at the rising edge of the clock, and only if the enable bit is set. Reset is low-active and works asynchronously. At t=0 all registers are set to zero.

The working memory (datamem.sv) is parameterized. It holds LENGTH number of bytes. There's a single input port (address is used both for reading and writing); an output port (value); a write enable bit; flags for reading and writing, which specify how many bytes to read/write; a clock, and a reset bit. Reading works asynchronously, while writing works synchronously, on positive edge of the clock. Reset is asynchronous, low-active, and sets all the bytes to zero (not to the initial values!).

Both instmem and datamem wrap around their length values. That means, if LENGTH=10, reading a 4-byte word from address 8 will give a value at addresses 8, 9, 0, 1. The core fully supports both LE and BE. The core fully supports accesses and writes at misaligned offsets. This means that writing a 4-byte word at address 1 won't raise any exceptions. This implementation doesn't raise exceptions on misaligned accesses to the instruction memory, even though the RiscV specification orders so.



6. References

[1] https://github.com/k13-engineering/node-elf-file
[2] https://riscv.org/technical/specifications/