ALE Exercises Book v 0.1

1. Introduction

This book contains a set of exercises to support professors and students in the assembly language learning process. The book is organized in chapters. Each chapter assumes the student has already learned a given set of concepts covered by the textbook (An Introduction to Assembly Programming with RISC-V) and by the ALE Manual.

2. Getting started with code generation tools

This chapter contains a set of exercises to help students getting started with code generation tools, including the compiler, the assembler, the linker and the disassembler tools.

Ex. 2.1: Step-by-step compilation process

This exercise is a tutorial that guides the student to generate executable files from source files. It requires the students to perform, step-by-step, all the code generation steps, including compilation, assembling, and linking.

Prerequisites

- Read and get familiar with the concepts and commands discussed in sections 3.1 to 3.6 of the ALE Manual. These sections cover the basics of generating and inspecting object and executable code.

Instructions

Perform the following program compilation process step-by-step, *i.e.*, generating the assembly language code and then the object code for each source file and finally calling the linker to put all the object files together and produce the executable file named `prog.x`. You must use the clang-15 compiler and generate code for the RISC-V R32 architecture, as in sections 3.1 to 3.6 of the ALE Manual.

The program is composed of two source files: file1.c and file2.c. The following listings show the file contents.

```
extern void write(int __fd, const void *__buf, int __n);

int main(void) {
  const char str[] = "Hello World!\n";
  write(1, str, 13);
  return 0;
}
```

file1.c

```
extern int main(void);

void exit(int code) {
  __asm__ __volatile__(
```

```
    "mv a0, %0           # return code\n"
    "li a7, 93           # syscall exit (64) \n"
    "ecall"
    :              // Output list
    :"r"(code)     // Input list
    : "a0", "a7"
  );
}

void write(int __fd, const void *__buf, int __n){
  __asm__ __volatile__(
    "mv a0, %0           # file descriptor\n"
    "mv a1, %1           # buffer \n"
    "mv a2, %2           # size \n"
    "li a7, 64           # syscall write (64) \n"
    "ecall"
    :   // Output list
    :"r"(__fd), "r"(__buf), "r"(__n)    // Input list
    : "a0", "a1", "a2", "a7"
  );
}

void _start() {
  int ret_code = main();
  exit(ret_code);
}
```

file2.c

To perform this task, you must, first, create these two files (with their respective contents) on your computer.

Once the compilation is complete, you must:

- Use the disassembler tool to inspect the contents of the object and executable files. You'll notice that it contains the sequence of instructions listed in the function write from file2.c and the names of functions main, _start, and write.
- Compare the contents of file1.s assembly file with the output generated by the disassembling tool for the file1.o file.

TIPS:

- Remember to use the RISC-V 32 compilation tools, otherwise you might not be able to correctly compile file2.c (Notice that this file contains assembly instructions for RISCV processors).

Ex. 2.2: Automating the compilation process with Makefiles

This exercise requires students to reason about the compilation process and automate this process with Makefiles.

Prerequisites

- Read and get familiar with the concepts and commands discussed in sections 3.1 to 3.5 of the ALE Manual. These sections cover the basics of generating and inspecting object and executable code.

- Read section 3.6 of the ALE Manual to learn more about Makefiles

Instructions

In this exercise, you must produce a Makefile script that automates the compilation process carried out manually in the previous exercise (Ex. 2.1), i.e., for a program that contains two source files (`file1.c` and `file2.c`). The final file to be produced must be named `prog.x`. For this, you must create a rule for each intermediate file, until you reach the final file.

You might test your script executing the following commands:

```
make file1.s
make file2.s
make file1.o
make file2.o
make prog.x
```

These commands must generate, respectively: the assembly code for file1.c and file2.c; the object files `file1.o` and `file2.o`; and, finally, the executable file named `prog.x`.

3. Getting started with the simulator

This chapter contains a set of exercises to help students getting started with the ALE simulator.

All exercises in this chapter assume the student is already familiar with compiling, assembling, and linking code for RISCV processors. These concepts are covered by Section 3 of the ALE manual and the exercises in the previous chapter.

Ex. 3.1: Running the hello world program

This exercise is a tutorial that guides the student to run executable files on the simulator.

Prerequisites

- Complete exercise 2.1 to generate the Hello World program (prog.x).
- Get familiar with the concepts and commands discussed in sections 4.1.1 and 4.1.2 to load and execute programs on the ALE simulator.

Instructions

Load and execute the prog.x file on the ALE simulator and inspect the output produced by the program on the simulator terminal. You should see the string "Hello world!".

Ex. 3.2: Debugging a program

This exercise requires students to use the ALE simulator interactive commands to inspect the contents of registers and memory locations of a running program.

Prerequisites

- Get familiar with the concepts of instructions, registers and labels.
- Get familiar with the concepts and commands discussed in Section 4.2 of the ALE simulator.

Instructions

In this exercise, you must modify the code below with your academic record (RA) number, assemble and link the code, and run it step by step in the simulator's interactive mode, as explained in the previous sections.

```
.globl _start

_start:
  li a0, 134985  #<<<=== Academic Record number (RA)
  li a1, 0
  li a2, 0
  li a3, -1
loop:
  andi t0, a0, 1
  add  a1, a1, t0
  xor  a2, a2, t0
  addi a3, a3, 1
  srli a0, a0, 1
  bnez a0, loop

end:
  la a0, result
  sw a1, 0(a0)
  li a0, 0
  li a7, 93
  ecall

result:
  .word 0
```

This program receives as input the value of its RA in register **a0** and produces as output the values in registers **a1**, **a2,** and **a3**. Assume the program has received your RA as input and answer the following questions:

1. What are the values of the following registers in hexadecimal representation when the execution reaches the "end" label?
    a. a0:
    b. a1:
    c. a2:
    d. a3:

2. What are the values of the following registers in hexadecimal representation when the execution reaches the “loop” label for the fifth time?
    a. a0:
    b. a1:
    c. a2:
    d. a3:

3. What are the values of the following registers in hexadecimal representation after the simulator executes the first **25** instructions?
    a. a0:
    b. a1:
    c. a2:
    d. a3:

4. What are the values of the following registers in hexadecimal representation when the execution reaches the “loop” label for the eighth time?
    a. a0:
    b. a1:
    c. a2:
    d. a3:

5. What are the values of the following registers in hexadecimal representation after the simulator executes the first **30** instructions?
    a. a0:
    b. a1:
    c. a2:
    d. a3:

6. What are the values of the following registers in hexadecimal representation when the contents of a1 and a2 are different from 0 and have the same value?
    a. a0:
    b. a3:

7. What value (in hexadecimal representation) is stored at the "result" memory location after the execution of instruction `sw a1, 0(a0)` (located after the "end" label)?

TIP: To see the complete list of commands available in the interactive mode you can execute the "help" command. For this particular exercise, we suggest using the commands listed in Table 4.1 of the ALE Manual.

Ex. 3.2: Single digit calculator

Prerequisites

- Get familiar with the concepts and commands discussed in Section 4.2 of the ALE simulator.

Instructions
In this exercise, you must modify the code below with your academic record (RA) number, assemble and link the code, and run it step by step in the simulator's interactive mode, as explained in the previous sections.

4. Data representation on modern computers

This chapter contains a set of exercises to help students learn how data is represented on modern computers, including how numbers and text are represented, how data is organized on the main memory, and how computer instructions are encoded.
Students are expected to read chapter 2 of the text book (An Introduction to Assembly Programming with RISC-V) and by the ALE Manual.

Ex. 4.1: Number Base Conversion in C

Instructions:
Write a C program that reads a number from the user input in either hexadecimal or decimal format and converts it to binary, decimal, and hexadecimal representations. The program should handle both positive and negative numbers.

Your program must read a series of characters encoded in ASCII format from the standard input (STDIN). These characters will represent a number in either decimal or hexadecimal format. Below, you will find the necessary functions for reading/writing information from/to the standard input/output., along with a concise illustration of how they can be utilized:

```
int read(int __fd, const void *__buf, int __n){
    int ret_val;
  __asm__ __volatile__(
    "mv a0, %1           # file descriptor\n"
    "mv a1, %2           # buffer \n"
    "mv a2, %3           # size \n"
    "li a7, 63           # syscall write code (63) \n"
    "ecall               # invoke syscall \n"
    "mv %0, a0           # move return value to ret_val\n"
    : "=r"(ret_val)  // Output list
    : "r"(__fd), "r"(__buf), "r"(__n)    // Input list
    : "a0", "a1", "a2", "a7"
  );
  return ret_val;
}

void write(int __fd, const void *__buf, int __n)
{
  __asm__ __volatile__(
    "mv a0, %0           # file descriptor\n"
    "mv a1, %1           # buffer \n"
    "mv a2, %2           # size \n"
    "li a7, 64           # syscall write (64) \n"
    "ecall"
    :   // Output list
    :"r"(__fd), "r"(__buf), "r"(__n)    // Input list
```

```
    : "a0", "a1", "a2", "a7"
  );
}

void exit(int code)
{
  __asm__ __volatile__(
    "mv a0, %0           # return code\n"
    "li a7, 93           # syscall exit (64) \n"
    "ecall"
    :   // Output list
    :"r"(code)    // Input list
    : "a0", "a7"
  );
}

void _start()
{
  int ret_code = main();
  exit(ret_code);
}

#define STDIN_FD  0
#define STDOUT_FD 1

int main()
{
  char str[20];
  /* Read up to 20 bytes from the standard input into the str buffer */
  int n = read(STDIN_FD, str, 20);
  /* Write n bytes from the str buffer to the standard output */
  write(STDOUT_FD, str, n);
  return 0;
}
```

Input:

- A 32-bit number represented by a string of up to 10 characters, followed by a newline character ("\n").
    - If the string represents a number in the hexadecimal base, it will start with characters "0x".
    - Otherwise, it will start with a number from 1 to 9 or with the minus sign (-), indicating that the number to be read is negative.
    - Note: The minus sign (-) will not be used in inputs in hexadecimal representation (e.g., -0x12 is not a valid input).

Output:

After reading the 32-bit number, you should print the following information followed by newline characters:

- The value in binary base preceded by "0b". If the number is negative, you must display the value in two's complement representation (as illustrated in the third example below);
- The value in decimal base assuming that the 32-bit number is encoded in two's complement representation (In this case, if the most significant bit is 1, the number is negative);

- The value in hexadecimal base preceded by "0x". If the number is negative, you must show the value in two's complement representation (as illustrated in the third example below);
- The value in decimal base assuming that the 32-bit number is encoded in unsigned representation and its endianness has been swapped;
    - For example, assuming the 32-bit number 0x00545648 was entered as input, after the endian swap, the number becomes 0x48565400, and its decimal value is 1213617152.

Examples:

`Input:`	`545648`

`Output:`	`0b10000101001101110000`
	`545648`
	`0x85370`
	`1884489728`

`Input:`	`0x545648`

`Output:`	`0b10101000101011001001000`
	`5527112`
	`0x545648`
	`1213617152`

`Input:`	`-545648`

`Output:`	`0b11111111111101111010110010010000`
	`-545648`
	`0xfff7ac90`
	`2427254783`

`Input:`	`0x80000000`

`Output:`	`0b10000000000000000000000000000000`
	`-2147483648`
	`0x80000000`
	`128`

Observations and Tips:

- This exercise challenges you to apply your understanding of number representation and conversion in C language. It's essential to consider edge cases, such as handling negative numbers, when designing and implementing your program.
- Note that the code for the write and read functions contains assembly instructions for the RISC-V architecture; as such, it is not possible to compile this code (as it is) for other architectures.
- You may assume that all input values will be small enough to be encoded within 32 bits.
- For printing the signed number in decimal base, we recommend:
    - Check if the number is negative by examining the most significant bit (MSB).
    - If it is negative:
        - Print the "-" sign;
        - Print the negated value of the number (consider the two's complement representation as discussed in the theory class).
- The RISC-V simulator stores data written with the write syscall in internal buffers and only displays the content in the terminal after encountering a newline character ("\n").
- In debugging mode, the terminal only accepts interactive debug commands. To provide input data to your program, you must use the "Standard IO" feature in the "Operating System" tab or the "write-stdin" command in the terminal.
- You can test your code using the simulator's assistant from this link.
    - Simply select your program, click on the assistant icon (small eye in the top-right corner of the screen), and click "Run Tests."
- We have identified two minor issues in the simulator. For now, we have updated the example in the prompt and suggest considering:
    - The input "-1" will not be provided to your program.
    - When using the read function, request a larger number of bytes than necessary (for this exercise, we suggest using the read function with n = 20, as shown in the example).

5. Assembly, object and executable files

This chapter contains a set of exercises to help students learn about how assembly, object and executable files are encoded.
Students are expected to read chapter 3 of the textbook (An Introduction to Assembly Programming with RISC-V).

Ex. 5.1: Organization of an ELF file

In this activity, you will manually read and interpret an ELF executable file from its hexadecimal byte-by-byte representation (hexdump), as exemplified below.

```
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 f3 00 01 00 00 00  b4 10 01 00 34 00 00 00  |............4...|
00000020  a4 01 00 00 04 00 00 00  34 00 20 00 04 00 28 00  |........4. ...(.|
```

```
00000030  06 00 04 00 06 00 00 00  34 00 00 00 34 00 01 00  |........4...4...|
00000040  34 00 01 00 80 00 00 00  80 00 00 00 04 00 00 00  |4...............|
00000050  04 00 00 00 01 00 00 00  00 00 00 00 00 00 01 00  |................|
00000060  00 00 01 00 b4 00 00 00  b4 00 00 00 04 00 00 00  |................|
00000070  00 10 00 00 01 00 00 00  b4 00 00 00 b4 10 01 00  |................|
00000080  b4 10 01 00 48 00 00 00  48 00 00 00 05 00 00 00  |....H...H.......|
00000090  00 10 00 00 51 e5 74 64  00 00 00 00 00 00 00 00  |....Q.td........|
000000a0  00 00 00 00 00 00 00 00  00 00 00 00 06 00 00 00  |................|
000000b0  00 00 00 00 37 15 02 00  13 05 95 f4 93 05 00 00  |....7...........|
000000c0  13 06 00 00 93 06 f0 ff  93 72 15 00 b3 85 55 00  |.........r....U.|
000000d0  33 46 56 00 93 86 16 00  13 55 15 00 e3 16 05 fe  |3FV......U......|
000000e0  17 05 00 00 13 05 85 01  23 20 b5 00 13 05 00 00  |........# ......|
000000f0  93 08 d0 05 73 00 00 00  00 00 00 00 4c 69 6e 6b  |....s.......Link|
00000100  65 72 3a 20 4c 4c 44 20  31 30 2e 30 2e 30 00 00  |er: LLD 10.0.0..|
00000110  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000120  01 00 00 00 e0 10 01 00  00 00 00 00 00 00 01 00  |................|
00000130  05 00 00 00 c8 10 01 00  00 00 00 00 00 00 01 00  |................|
00000140  0a 00 00 00 f8 10 01 00  00 00 00 00 00 00 01 00  |................|
00000150  11 00 00 00 b4 10 01 00  00 00 00 00 10 00 01 00  |................|
00000160  00 2e 74 65 78 74 00 2e  63 6f 6d 6d 65 6e 74 00  |..text..comment.|
00000170  2e 73 79 6d 74 61 62 00  2e 73 68 73 74 72 74 61  |.symtab..shstrta|
00000180  62 00 2e 73 74 72 74 61  62 00 00 65 6e 64 00 6c  |b..strtab..end.l|
00000190  6f 6f 70 00 72 65 73 75  6c 74 00 5f 73 74 61 72  |oop.result._star|
000001a0  74 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |t...............|
000001b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000001c0  00 00 00 00 00 00 00 00  00 00 00 00 01 00 00 00  |................|
000001d0  01 00 00 00 06 00 00 00  b4 10 01 00 b4 00 00 00  |................|
000001e0  48 00 00 00 00 00 00 00  00 00 00 00 04 00 00 00  |H...............|
000001f0  00 00 00 00 07 00 00 00  01 00 00 00 30 00 00 00  |............0...|
00000200  00 00 00 00 fc 00 00 00  13 00 00 00 00 00 00 00  |................|
00000210  00 00 00 00 01 00 00 00  01 00 00 00 10 00 00 00  |................|
00000220  02 00 00 00 00 00 00 00  00 00 00 00 10 01 00 00  |................|
00000230  50 00 00 00 05 00 00 00  04 00 00 00 04 00 00 00  |P...............|
00000240  10 00 00 00 18 00 00 00  03 00 00 00 00 00 00 00  |................|
00000250  00 00 00 00 60 01 00 00  2a 00 00 00 00 00 00 00  |....`...*.......|
00000260  00 00 00 00 01 00 00 00  00 00 00 00 22 00 00 00  |............"...|
00000270  03 00 00 00 00 00 00 00  00 00 00 00 8a 01 00 00  |................|
00000280  18 00 00 00 00 00 00 00  00 00 00 00 01 00 00 00  |................|
00000290  00 00 00 00                                       |....|
00000294
```

This is an example of the Hexdump for the executable of exercise Ex. 3.2. In the example, each line contains the following information, separated by spaces:

1. Offset of the first byte in the line (in hexadecimal). The offset is a number indicating the distance of the information (in this case, the first byte of the line) from a reference point (in this case, the beginning of the file).
2. 16 hexadecimal numbers, each representing a byte of the file.
3. 16 ASCII characters enclosed in "|". This is an attempt to decode the bytes into ASCII. The character "." may indicate a value that doesn't represent valid or printable ASCII characters.

As a first step, you should carefully read the description of the ELF format (particularly the tables for File Header, Program Header, and Section Header).

Notes and Tips:

- All numbers are represented in hexadecimal.
- Memory words store numbers in little-endian representation. Therefore, the bytes "34 00 01 00" represent the value 0x10034 in 4 bytes.

- Refer to the [Practical Guide to RISC-V: Atlas of an Open Architecture to decode the instructions](). Especially, consult the beginning of Chapter 2 and Figure 2.3.

Example Solution:
To make your job easier, we have discussed how to read the ELF file given above. The following listing contains the same file contents, but with color marks to simplify the discussion.

```
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 f3 00 01 00 00 00  b4 10 01 00 34 00 00 00  |............4...|
00000020  a4 01 00 00 04 00 00 00  34 00 20 00 04 00 28 00  |........4. ...(.|
00000030  06 00 04 00 06 00 00 00  34 00 00 00 34 00 01 00  |........4...4...|
00000040  34 00 01 00 80 00 00 00  80 00 00 00 04 00 00 00  |4...............|
00000050  04 00 00 00 01 00 00 00  00 00 00 00 00 00 01 00  |................|
00000060  00 00 01 00 b4 00 00 00  b4 00 00 00 04 00 00 00  |................|
00000070  00 10 00 00 01 00 00 00  b4 00 00 00 b4 10 01 00  |................|
00000080  b4 10 01 00 48 00 00 00  48 00 00 00 05 00 00 00  |....H...H.......|
00000090  00 10 00 00 51 e5 74 64  00 00 00 00 00 00 00 00  |....Q.td........|
000000a0  00 00 00 00 00 00 00 00  00 00 00 00 06 00 00 00  |................|
000000b0  00 00 00 00 37 15 02 00  13 05 95 f4 93 05 00 00  |....7...........|
000000c0  13 06 00 00 93 06 f0 ff  93 72 15 00 b3 85 55 00  |.........r....U.|
000000d0  33 46 56 00 93 86 16 00  13 55 15 00 e3 16 05 fe  |3FV......U......|
000000e0  17 05 00 00 13 05 85 01  23 20 b5 00 13 05 00 00  |........# ......|
000000f0  93 08 d0 05 73 00 00 00  00 00 00 00 4c 69 6e 6b  |....s.......Link|
00000100  65 72 3a 20 4c 4c 44 20  31 30 2e 30 2e 30 00 00  |er: LLD 10.0.0..|
00000110  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000120  01 00 00 00 e0 10 01 00  00 00 00 00 00 00 01 00  |................|
00000130  05 00 00 00 c8 10 01 00  00 00 00 00 00 00 01 00  |................|
00000140  0a 00 00 00 f8 10 01 00  00 00 00 00 00 00 01 00  |................|
00000150  11 00 00 00 b4 10 01 00  00 00 00 00 10 00 01 00  |................|
00000160  00 2e 74 65 78 74 00 2e  63 6f 6d 6d 65 6e 74 00  |..text..comment.|
00000170  2e 73 79 6d 74 61 62 00  2e 73 68 73 74 72 74 61  |.symtab..shstrta|
00000180  62 00 2e 73 74 72 74 61  62 00 00 65 6e 64 00 6c  |b..strtab..end.l|
00000190  6f 6f 70 00 72 65 73 75  6c 74 00 5f 73 74 61 72  |oop.result._star|
000001a0  74 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |t...............|
000001b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000001c0  00 00 00 00 00 00 00 00  00 00 00 00 01 00 00 00  |................|
000001d0  01 00 00 00 06 00 00 00  b4 10 01 00 b4 00 00 00  |................|
000001e0  48 00 00 00 00 00 00 00  00 00 00 00 04 00 00 00  |H...............|
000001f0  00 00 00 00 07 00 00 00  01 00 00 00 30 00 00 00  |............0...|
00000200  00 00 00 00 fc 00 00 00  13 00 00 00 00 00 00 00  |................|
00000210  00 00 00 00 01 00 00 00  01 00 00 00 10 00 00 00  |................|
00000220  02 00 00 00 00 00 00 00  00 00 00 00 10 01 00 00  |................|
00000230  50 00 00 00 05 00 00 00  04 00 00 00 04 00 00 00  |P...............|
00000240  10 00 00 00 18 00 00 00  03 00 00 00 00 00 00 00  |................|
00000250  00 00 00 00 60 01 00 00  2a 00 00 00 00 00 00 00  |....`...*.......|
00000260  00 00 00 00 01 00 00 00  00 00 00 00 22 00 00 00  |............"...|
00000270  03 00 00 00 00 00 00 00  00 00 00 00 8a 01 00 00  |................|
00000280  18 00 00 00 00 00 00 00  00 00 00 00 01 00 00 00  |................|
00000290  00 00 00 00                                       |....|
00000294
```

1. First, we must identify the values of fields `e_shoff`, `e_shnum`, and `e_shstrndx` (see their description on the Wikipedia page related to the ELF format), which are colored in purple, blue, and red, respectively.
    a. The value of the `e_shoff` field is `0x000001a4` (recall the little-endian representation), indicating that the Section Headers start at this offset.
2. According to the ELF format, we know that each Section Header contains 0x28 bytes. Thus, we marked the beginning of each of them in green.

3. `e_shstrndx` contains the value 4, indicating that Section Header number 4 (counting from 0) is the header of `shstrtab`, which stores the information about the names of the sections. We colored its content in orange.
4. The `sh_offset` field of Section Header 4 (orange number with gray background) stores the address of `shstrtab`: `0x00000160`.
5. The `sh_name` field of Section Header 4 is the offset (from the beginning of the `shstrtab` section) to the string representing the section's name. For example, Section Header number 1 has an offset of `0x1` (highlighted in cyan), so its name is the string at position `0x00000160` + `0x1`. In other words, this is the Header of the "`.text`" section, which stores the executable's instructions.
6. Our objective is to find the "`.symtab`" and "`.strtab`" sections, which respectively store the addresses and names of the symbols.
7. By checking the name of each section, we identified that Section Headers number 3 and 5 represent the "`.symtab`" and "`.strtab`" sections.
8. Evaluating the `sh_offset` field of each of them, we know that the "`.symtab`" and "`.strtab`" sections are located at addresses `0x00000110` and `0x0000018a`. Evaluating the sh_size field, we know that their sizes are `0x50` and `0x18` bytes, respectively.
9. In the `.symtab` section, for each symbol:
    a. The first 4 bytes represent the offset of the symbol's name in the "`.strtab`" section.
    b. The next 4 bytes represent the symbol's address in the program's memory.
    c. The last 8 bytes represent other information about the symbol (not useful to us at this moment).
10. Copying the sections here:
    a. `.symtab`: (offsets to `.strtab` are highlighted in blue, addresses in red)

```
00000110  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000120  01 00 00 00 e0 10 01 00  00 00 00 00 00 00 01 00  |................|
00000130  05 00 00 00 c8 10 01 00  00 00 00 00 00 00 01 00  |................|
00000140  0a 00 00 00 f8 10 01 00  00 00 00 00 00 00 01 00  |................|
00000150  11 00 00 00 b4 10 01 00  00 00 00 00 10 00 01 00  |................|
```

    b. `.strtab`:

```
00000180  62 00 2e 73 74 72 74 61  62 00 00 65 6e 64 00 6c  |b..strtab..end.l|
00000190  6f 6f 70 00 72 65 73 75  6c 74 00 5f 73 74 61 72  |oop.result._star|
000001a0  74 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |t...............|
```

11. Now we can easily identify the names and addresses of the symbols:
    a. The first line of the `symtab` is null, and we can ignore it.
    b. The second line tells us that the symbol with the name at offset `0x1` is at address `0x000110e0`. Consulting the `strtab`, we see that offset `0x1` is occupied by the name "end" (String defined by the bytes `65 6e 64 00`. Notice that the string is terminated with the NULL character, with a value of 00).
    c. The third line tells us that the symbol with the name at offset `0x5` is at address `0x000110c8`. Consulting the `strtab`, we see that offset `0x5` is occupied by the name "`loop`".

d. Following this same logic, we can identify the name and address of all symbols.

6. Bit Manipulation and Instruction Encoding

This chapter contains a set of exercises to help students learn about bit manipulation and how 32 bit RISC-V instructions are encoded.

Ex. 6.1: Bit Masking and Shift operations

Instructions:
Write a C program that reads 5 numbers from the user input in decimal format and packs its bits to a single 32 bit value that will be written to the standard output as a hexadecimal value. The program should handle both positive and negative numbers.

A function with the signature ***void* pack(*int input*, *int start_bit*, *int end_bit*, *int *val*)** might be a good start for this exercise.

The previously presented functions **read** and **write** can be used for reading/writing information from/to the standard input/output. The code snippet below can be used to write the resulting hexadecimal value to STDOUT (note that it uses the **write** function).

```
void hex_code(int val){
    char hex[11];
    unsigned int uval = (unsigned int) val, aux;

    hex[0] = '0';
    hex[1] = 'x';
    hex[10] = '\n';

    for (int i = 9; i > 1; i--){
        aux = uval % 16;
        if (aux >= 10)
            hex[i] = aux - 10 + 'A';
        else
            hex[i] = aux + '0';
        uval = uval / 16;
    }
    write(1, hex, 11);
}
```

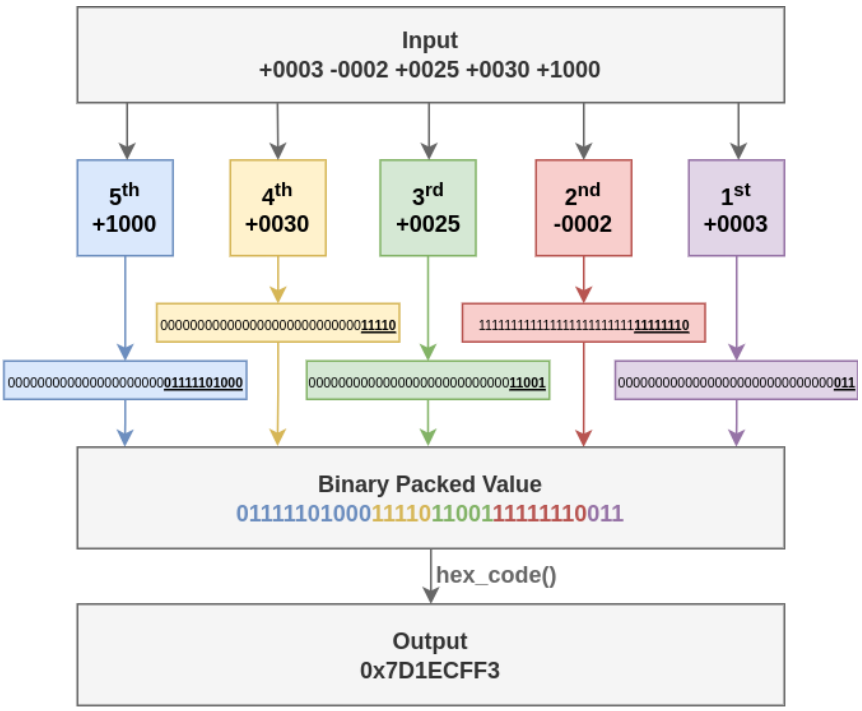
Input

- 5 signed 4-digit decimal numbers separated by spaces (' '), followed by a newline character ('\n'). The whole input takes up 30 bytes.
    - String Format - **"SDDDD SDDDD SDDDD SDDDD SDDDD\n"**
    - **S**: sign, can be either '+' for positive numbers and '-' for negative.
    - **D**: a decimal digit, (0-9)

Output:
After reading all 5 numbers, you must pack their least significant bits (LSB) following the rules listed below:

- 1st number: 3 LSB => Bits 0 - 2

- 2nd number: 8 LSB => Bits 3 - 10
- 3rd number: 5 LSB => Bits 11 - 15
- 4th number: 5 LSB => Bits 16 - 20
- 5th number: 11 LSB => Bits 21 - 31



Examples:

Input:	-0001 -0001 -0001 -0001 -0001
Output:	0xFFFFFFFF

Input:	+0001 +0001 -0001 -0001 -0001
Output:	0xFFFFF809

Input:	+0003 -0002 +0025 +0030 +1000
Output:	0x7D1ECFF3

Input:	+9999 +9999 +9999 +9999 +9999
Output:	0xE1EF787F

Notes and Tips:

- You may use the C operator for bit manipulation (<<, >>, |, &, ^, …) to implement your function. Suggested readings:
  - Bit Operations
  - Bitwise Operators
- You can test your code using the simulator's assistant from this link.

Ex. 6.2: RISC-V Instruction Encoding

Instructions:
Write a C program that reads a string with a RISC-V instruction from STDIN, parses its content in a way of obtaining its fields, packs the instruction's fields in a 32 bit value and writes the hexadecimal representation of the instruction to STDOUT.

The code snippet below can be used to compare strings as standard C libraries such as **string.h** are not available in the simulator. It is similar to **string.h**'s *strcmp* but it has the number of characters to be compared as a parameter.

```
int strcmp_custom(char *str1, char *str2, int n_char){
    for (int i = 0; i < n_char; i++){
        if (str1[i] < str2 [i])
            return -1;
        else if (str1[i] > str2 [i])
            return 1;
    }
    return 0;
}
```

The set of instructions that need to be encoded by your program is presented in the table below, alongside its opcode, instruction type and other fields (e.g. funct3 and funct7) if applicable.

Instruction	Inst Syntax	Inst Type	OPCODE	FUNCT3	FUNCT7
lui	lui rd, imm	U	0110111	N/A	N/A
auipc	auipc rd, imm	U	0010111	N/A	N/A
jal	jal rd, imm	J	1101111	N/A	N/A
jalr	jalr rd, imm(rs1)	I	1100111	000	N/A
beq	beq rs1, rs2, imm	B	1100011	000	N/A
bne	bne rs1, rs2, imm	B	1100011	001	N/A
blt	blt rs1, rs2, imm	B	1100011	100	N/A
bge	bge rs1, rs2, imm	B	1100011	101	N/A
bltu	bltu rs1, rs2, imm	B	1100011	110	N/A

bgeu	bgeu rs1, rs2, imm	B	1100011	111	N/A
lb	lb rd, imm(rs1)	I	0000011	000	N/A
lh	lh rd, imm(rs1)	I	0000011	001	N/A
lw	lw rd, imm(rs1)	I	0000011	010	N/A
lbu	lbu rd, imm(rs1)	I	0000011	100	N/A
lhu	lhu rd, imm(rs1)	I	0000011	101	N/A
sb	sb rs2, imm(rs1)	S	0100011	000	N/A
sh	sh rs2, imm(rs1)	S	0100011	001	N/A
sw	sw rs2, imm(rs1)	S	0100011	010	N/A
addi	addi rd, rs1, imm	I	0010011	000	N/A
slti	slti rd, rs1, imm	I	0010011	010	N/A
sltiu	sltiu rd, rs1, imm	I	0010011	011	N/A
xori	xori rd, rs1, imm	I	0010011	100	N/A
ori	ori rd, rs1, imm	I	0010011	110	N/A
andi	andi rd, rs1, imm	I	0010011	111	N/A
slli	slli rd, rs1, imm**	I	0010011	001	0000000*
srli	srli rd, rs1, imm	I	0010011	101	0000000*
srai	srai rd, rs1, imm	I	0010011	101	0100000*
add	add rd, rs1, rs2	R	0110011	000	0000000
sub	sub rd, rs1, rs2	R	0110011	000	0100000
sll	sll rd, rs1, rs2	R	0110011	001	0000000
slt	slt rd, rs1, rs2	R	0110011	010	0000000
sltu	sltu rd, rs1, rs2	R	0110011	011	0000000
xor	xor rd, rs1, rs2	R	0110011	100	0000000
srl	srl rd, rs1, rs2	R	0110011	101	0000000
sra	sra rd, rs1, rs2	R	0110011	101	0100000
or	or rd, rs1, rs2	R	0110011	110	0000000
and	and rd, rs1, rs2	R	0110011	111	0000000

* *slli*, *srli* and *srai* are type I instructions but its immediate takes up only 5 bits, the remaining 7 bits are filled with a *funct7* value.

** The `imm` field may also appear as `shamt`, which stands for shift amount.

Input

- RV32I assembly instruction string with at most 40 bytes. There will be no pseudo-instructions and the registers will be referenced with their x-name (e.g. x2, x10)

Output:

- The 32 bit encoded instruction in its Big Endian hexadecimal representation (**hex_code()** from the previous exercise can be used).

Examples:

`Input:`	`lb x10, 4(x9)`
`Output:`	`0x00448503`

`Input:`	`and x31, x20, x25`
`Output:`	`0x019A7FB3`

`Input:`	`slti x12, x13, -1`
`Output:`	`0xFFF6A613`

`Input:`	`bge x7, x0, 256`
`Output:`	`0x1003D063`

`Input:`	`jalr x1, -32(x9)`
`Output:`	`0xFE0480E7`

Notes and Tips:

- This exercise depends on some things used in Exercise 6.1, such as **hex_code()** and **pack()** functions, so it is recommended to do exercise 6.1 first.
- Refer to the Practical Guide to RISC-V: Atlas of an Open Architecture to check how each instruction is encoded. Especially, consult the beginning of Chapter 2 and Figure 2.3.
- You can test your code using the simulator's assistant from this link.

7. Assembly User-level Programming

Ex. 7.1: Square Root

Instructions:
Write a program in RISC-V assembly language that computes the approximated square root of integers.

To perform read and write of data from/to the terminal, you must use the *read* and *write* syscalls (similarly to exercise 4.1, but now in assembly language)

read syscall example:

```
    li a0, 0  # file descriptor = 0 (stdin)
    la a1, input_address #  buffer to write the data
    li a2, 1  # size (reads only 1 byte)
    li a7, 63 # syscall read (63)
    ecall

input_address: .skip 0x10  # buffer
```

write syscall example:

```
    li a0, 1            # file descriptor = 1 (stdout)
    la a1, string       # buffer
    li a2, 19           # size
    li a7, 64           # syscall write (64)
    ecall
string:  .asciz "Hello! It works!!!\n"
```

Input

- Four 4-digit decimal numbers separated by spaces (' '), followed by a newline character ('\n'). The whole input takes up 20 bytes.
    - String Format - **"DDDD DDDD DDDD DDDD\n"**
    - **D**: a decimal digit, (0-9)

Output:
For each 4-digit number read, you must compute its approximate square root and write its value to STDOUT using 4-digits and each square root must be separated by a space (' ') and the last one is followed by a newline character ('\n'), so the output will also take up 20 bytes.

- String Format - **"DDDD DDDD DDDD DDDD\n"**
- **D**: a decimal digit, (0-9)

Examples:

Input:	0400 5337 2240 9166
Output:	0020 0073 0047 0095

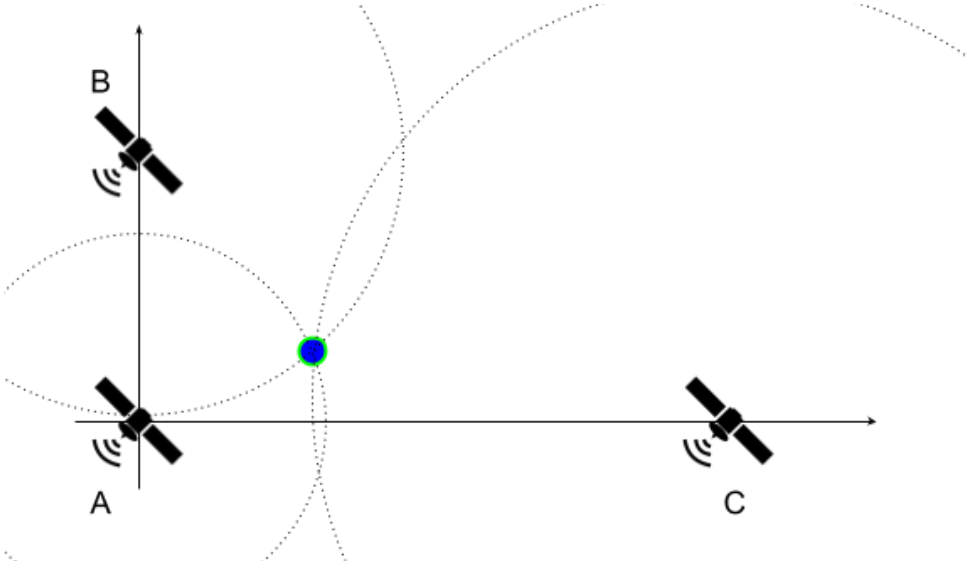
Notes and Tips:

- The usage of Babylonian method with 10 iterations is recommended. Considering that we want to compute the square root of a number y, the basic idea of this method is:
  - Compute an initial guess for the square root:
    $$k = \frac{y}{2}.$$
  - Approximate your estimative, k, to the real value of the square root by applying the following equation:
    $$k' = \frac{k + \frac{y}{k}}{2}$$
  - Each time the above equation is applied is considered "one iteration". For this exercise, use 10 iterations.
- For this exercise, approximate solutions are accepted.
  - Solutions with an absolute error smaller than 10 will be considered correct.
- Other methods to square root approximation can be used, as long as:
  - It used only integers. Floating point numbers or the RISC-V square root instruction cannot be used.
  - The approximation is as or more precise than the suggested method.
- You can test your code using the simulator's assistant from this link.

Ex. 7.2: GPS

Instructions:

In this activity, you must write a program in RISC-V assembly language that computes your geographical coordinates in a bidimensional plane, based on the current time and messages received from 3 satellites.



To simplify the exercise, it is assumed that satellite A is placed at the origin of the cartesian plane (0, 0), while B and C are positioned at (0, Y_B) and (X_C, 0), respectively. The satellites continuously send messages with a *timestamp* via waves that propagate in all directions at a speed of 3 x 10^8 m/s. At a given time T_R, you receive a message from each satellite

containing the timestamps T_A, T_B and T_C. Assuming that all clocks are perfectly synchronized, print your coordinates (x, y) in the cartesian plane. Note that the formulation used in this exercise is not realistic.

Input:

- **Line 1** - Coordinates Y_B and X_c. Values are in meters, represented by 4-digit integers in decimal base and preceded by a sign ('+' or '-').
- **Line 2** - Times T_A, T_B, T_c and T_R. Values are in nanoseconds, represented by 4-digit integers in decimal base

Output:

- **Your coordinate** - (x, y). Values are in meters, approximated, represented by 4-digit integers in decimal base and preceded by a sign ('+' or '-').

Examples:

Input:	+0700 -0100 2000 0000 2240 2300
Output:	-0088 +0016

Input:	+1042 -2042 6823 4756 6047 9913
Output:	-0902 -0215

Input:	-2168 +0280 3207 5791 3638 9550
Output:	+0989 -1626

Input:	-2491 +0965 2884 7511 2033 9357
Output:	-0065 -1941

Input:	-0656 +1337 0162 2023 1192 9133
Output:	+1255 -2381

Notes and Tips:

- Multiple values written or read on/from the same line will be separated by a single space.
- Each line ends with a newline character '\n'.
- For this exercise, approximate solutions are accepted.
    - Solutions with an absolute error smaller than 10 will be considered correct.

- **The usage of the same method used in Exercise 7.1 with more iterations (e.g. 21 iterations) is recommended**. Other methods to square root approximation can be used, as long as:
    - It used only integers. Floating point numbers or the RISC-V square root instruction cannot be used.
    - The approximation is as or more precise than the suggested method.
- It is best to work with distances in meters and time in nanoseconds, so that the provided input values do not cause *overflow* when using the proposed method and a good precision might be achieved.
- Problem Geometry:
    - There are many ways to solve this exercise. Here, we propose an approach that uses the equation of a circle. Given that d_A, d_B and d_C are the distances between your position and the satellites A, B and C, respectively:
        - $x^2 + y^2 = d_A^2$ (Eq. 1)
        - $x^2 + (y - Y_B)^2 = d_B^2$ (Eq. 2)
        - $(x - X_C)^2 + y^2 = d_C^2$ (Eq. 3)
    - Using Equations 1 and 2:
        - $y = (d_A^2 + Y_B^2 - d_B^2) / 2Y_B$ (Eq. 4)
        - $x = + \text{sqrt}(d_A^2 - y^2)$ **OU** $- \text{sqrt}(d_A^2 - y^2)$ (Eq. 5)
    - To find the correct x, you can try both possible values in Equation 3 and check which one is closer to satisfying the equation.

- You can test your code using the simulator's assistant from this link.

Ex. 7.3: Hamming Code

Instructions:
In this activity, you must write a program in RISC-V assembly language that performs the encoding and decoding of a Hamming(7, 4) code.

Encoding
For the first part of this exercise, you will receive a sequence of 4 bits, and you have to encode these data bits using the Hamming Code. Assuming that the 4 bit input is given as:

$d_1d_2d_3d_4$

The output will be

$p_1p_2d_1p_3d_2d_3d_4$

The new inserted bits with radical **p** are **parity** bits. Each one of the 3 parity bits is responsible for reflecting the parity of a given subset of bits (subset of 3 elements from the 4 available input bits). A parity bit is **1** if the evaluated set of bits has an **odd** number of **1**s, or **0** otherwise. The following table can be used as reference:

Parity bit	Subset of tested bits
p_1	$d_1d_2d_4$
p_2	$d_1d_3d_4$
p_3	$d_2d_3d_4$

Decoding

On the second part of this exercise, you will receive a sequence of 7 bits that has been encoded. You have to extract the data field from this sequence, and also check if the data contains an error caused by a bit flip (there is no need for correcting the data if an error is detected). For this error checking, you have to verify the parity of each one of the 3 subsets.

The XOR operator can be used for a given subset of bits. For instance, to check the parity for which $\mathbf{p_1}$ is responsible, p1 XOR d1 XOR d2 XOR d4 must be equal to 0. Otherwise, there is an error on the encoded data. Do this for the 3 subsets of bits in order to check if you can trust the data encoded with Hamming(7, 4).

Input:

- **Line 1** - a sequence of 4 bits that must be encoded in a Hamming code using 3 parity bits.
- **Line 2** - a sequence of 7 bits that is Hamming encoded, and must be decoded and checked.

Output:

- **Line 1** - sequence of 7 bits that has been encoded using Hamming code
- **Line 2** - sequence of 4 bits that has been decoded from the Hamming code.
- **Line 3 -** 1 if an error was detected when decoding the Hamming code, 0 otherwise.

Examples:

`Input:`	`1001` `0011001`

`Output:`	`0011001`
	`1001`
	`0`

`Input:`	`0000` `0000000`

`Output:`	`0000000`
	`0000`
	`0`

`Input:`	`0001` `0010001`

`Output:`	`1101001`
	`1001`
	`1`

`Input:`	`1111` `1001001`

`Output:`	`1111111`
	`0001`
	`1`

`Input:`	`1010` `1011010`

`Output:`	`1011010`
	`1010`
	`0`

Notes and Tips:

- Exclusive OR (XOR) is a logic operator that facilitates the computation of parity bits
- AND instruction is useful to leave only a given group of bits set (masking).
- The decoded data doesn't need to be corrected, in case an error is detected.
- You can test your code using the simulator's assistant from this link.

Ex. 7.4: Image on Canvas

Instructions:
In this activity, you must write a program in RISC-V assembly language that reads an image in PGM format from a file and shows it on screen using the canvas peripheral.

Input:
Your program must read a file called "image.pgm" that will be in the same directory as the executable file. The *syscall* open can be used to open the file (an example is shown at the Notes section)

Some examples of PGM images can be found on this site:
PGMB Files - Binary Portable Gray Map Graphics Files

Output:
Your program must show the image on the screen using *syscalls* to the Canvas peripheral. The Canvas must be adjusted to have the image size. The available *syscalls* are:

Syscall	Input	Description
setPixel	a0: pixel's x coordinate a1: pixel's y coordinate a2: concatenated pixel's colors: R|G|B|A • A2[31..24]: Red • A2[23..16]: Green • A2[15..8]: Blue • A2[7..0]: Alpha a7: 2200 (*syscall* number)	Defines the color of a given canvas' pixel. For gray scale, use the same values for the colors (R = G = B) and alpha = 255.
setCanvasSize	a0: canvas width (value between 0 and 512) a1: canvas height (value between 0 and 512)	Resets and defines canvas' size.

	a7: 2201 (*syscall* number)	
setScaling	a0: horizontal scaling a1: vertical scaling a7: 2202 (*syscall* number)	Updates canvas' scaling

Notes and Tips:

- To test on the simulator, you have to load your program (.s file) and the image file (name "image.pgm") simultaneously.
- When new files are loaded to the simulator, older ones are erased, so you have to load the program and image files together every time.
- To use the Canvas, you must enable it on the simulator. To do so, go to the tab “Hardware” -> “External Devices” table -> “+” Icon on the Canvas row. A new tab will appear where the canvas can be seen.
- This exercise uses multiple *syscall* numbers. These values will always be stored on the register a7, and the *ecall* function has a different behavior for each value. To check the *syscall* for a specific functionality, the simulator table can be checked (note that the syscalls related to external devices, like Canvas, are not shown in this table if the device is not enabled).
- You will not receive images larger than 512x512 (that typically takes up 262159 bytes).
- In all images, Maxval will be 255.
- The canvas is indexed starting on 0.
- **You need to resize the canvas (setCanvasSize *syscall*) according to the file header.**
- You can test your code using the simulator's assistant from this link.

***setPixel* example:**

```
    li a0, 100 # x coordinate = 100
    li a1, 200 # y coordinate = 200
    li a2, 0xFFFFFFFF # white pixel
    li a7, 2200 # syscall setPixel (2200)
    ecall
```

***open* example:**

The open *syscall* returns the file descriptor (fd) for the file on a0. This file descriptor must be used on the read syscall to indicate the file from which the operating system must read from *syscall* to get the contents of the file.

```
    la a0, input_file     # address for the file path
    li a1, 0              # flags (0: rdonly, 1: wronly, 2: rdwr)
    li a2, 0              # mode
    li a7, 1024           # syscall open
    ecall

input_file: .asciz "image.pgm"
```

Ex. 7.5: Applying a Filter to an Image

Instructions:
In this activity, you must write a program in RISC-V assembly language that reads an image in PGM format from a file, applies an edge detection filter and shows the result on screen using the canvas peripheral.

The first step of this exercise is to read an image in the PGM format and store its content in a matrix (exactly as done in exercise 7.4). After that, you must apply the following filter on the image:

w =

-1	-1	-1
-1	8	-1
-1	-1	-1

Assuming *w* the filter matrix above, M_{in} the matrix representing the input image and M_{out} the matrix representing the output image. The basic idea of applying the filter is that each M_{out} *pixel*[i, j] is defined as:

$$M_{out}[i][j] = \sum_{k=0}^{2} \sum_{q=0}^{2} w[k][q] * W_{in}[i+k-1][j+q-1]$$

Note that this can lead to the M_{in} matrix to be indexed with invalid indices (negative or out of bounds). In order to avoid these cases, the border *pixels* of the image M_{out} must be initialized with black and it is not necessary to compute the filter value for them. You can visualize how this filter works in Image Kernels explained visually (select the "outline" filter). This filter is also known as the Laplacian operator for edge detection.

Also note that the image *pixels* must have values between 0 (black) and 255 (white). If the result of the equation presented above is not in this interval, you must use the closest value in the interval (i.e., values lower than 0 become 0, and values greater than 255 become 255).

Input:
Your program must read a file called "image.pgm" that will be in the same directory as the executable file, as explained in exercise 7.4.

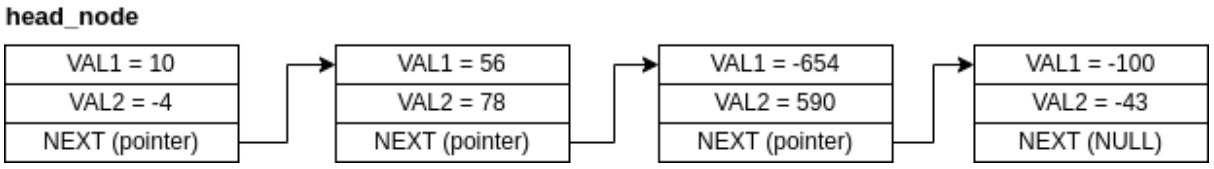
Output:
Your program must show the result image on the screen, using the Canvas peripheral, as explained in exercise 7.4.

Notes and Tips:

- **You need to resize the canvas (setCanvasSize *syscall*) according to the file header.**
- You can test your code using the simulator's assistant from this link.

Ex. 7.6: Custom Search on a Linked List

Instructions:
In this activity, you must write a program in RISC-V assembly language that receives a number via STDIN, goes through a linked list and prints the index of the node where the sum of the two stored values is equal to the number read from STDIN. In case none of the nodes in the linked list fill this requirement, the number -1 must be printed.



For the linked list above, in case the input was 134, the number 1 (index of the second node on the list) should be printed as 56 + 78 = 134. If the input was a number that is not equal to the sum of any of the nodes (e.g. 8), -1 should be printed.

Input:
An ASCII encoded decimal number on the interval -10000 and 10000.

Output:
Index of the node where the sum of the two stored values is equal to the received value, if it exists on the linked list, -1 if it does not exist.

Examples:
Example of *data.s* file that will be linked to your program:

```
.globl head_node

.data
head_node:
    .word 10
    .word -4
    .word node_1
.skip 10
node_1:
    .word 56
    .word 78
    .word node_2
.skip 5
node_3:
    .word -100
    .word -43
    .word 0
node_2:
    .word -654
    .word 590
    .word node_3
```

Input:	6

Output:	0

Input:	45
Output:	-1

Input:	-64
Output:	2

Notes and Tips:

- The head node of the linked list is stored on the address marked by the label *head_node* (DO NOT use this label on your code).
- The fields of the linked list node struct are VAL1, VAL2, and NEXT, in this order. VAL1 and VAL2 are 32-bit **signed** integer values stored on the node and NEXT is the pointer to the next node on the linked list. If there isn't a next node, it will be a NULL pointer.
- To check if the received value is on the current node, the comparison VAL1 + VAL2 = received value must be made.
- A NULL pointer is represented by the value 0.
- The indexing of the list nodes starts at 0 (*i.e.*, the head_node has index 0).
- All nodes will have different sum values.
- You can test your code using the simulator's assistant from this link.

Ex. 7.7: ABI-compliant linked list custom search

Instructions:

In this activity, you must write a program in RISC-V assembly language that has a ABI-compliant function **int linked_list_search(Node *head_node, int val),** similar to the one implemented in exercise 7.6**.** Your code will be linked to a code in C that calls the **linked_list_search** function and expects the return value to be the index of the node where the sum of the values is equal to **val** or -1 in case there isn't such a node on the linked list.

In addition to **linked_list_search**, you will need to implement a few utility functions that will be required for this exercise and future ones.

`void puts ( const char * str );`		
Description	Parameters	Return Value
puts - C++ Reference	String terminated by \0	-

`char * gets ( char * str );`

Description	Parameters	Return Value
gets - C++ Reference	Buffer to be filled	Check function documentation

`int atoi (const char * str);`		
Description	Parameters	Return Value
atoi - C++ Reference	String terminated by \0	Integer represented by the string

`char *  itoa ( int value, char * str, int base );`		
Description	Parameters	Return Value
itoa - C++ Reference	Check function documentation. Only bases 10 and 16 will be tested. The tests won't be *case-sensitive*.	

`void exit(int code);`		
Description	Parameters	Return Value
Calls *exit* syscall to finish execution.	Return code (usually used as error code).	-

***lib.h*:**

```
typedef struct Node {
    int val1, val2;
    struct Node *next;
} Node;

int linked_list_search(Node *head_node, int val);
void puts ( const char *str );
char *gets ( char *str );
int atoi (const char *str);
char *itoa ( int value, char *str, int base );
void exit(int code);
```

Input:
Test Case index and specific input depending on the test case (check example.c).

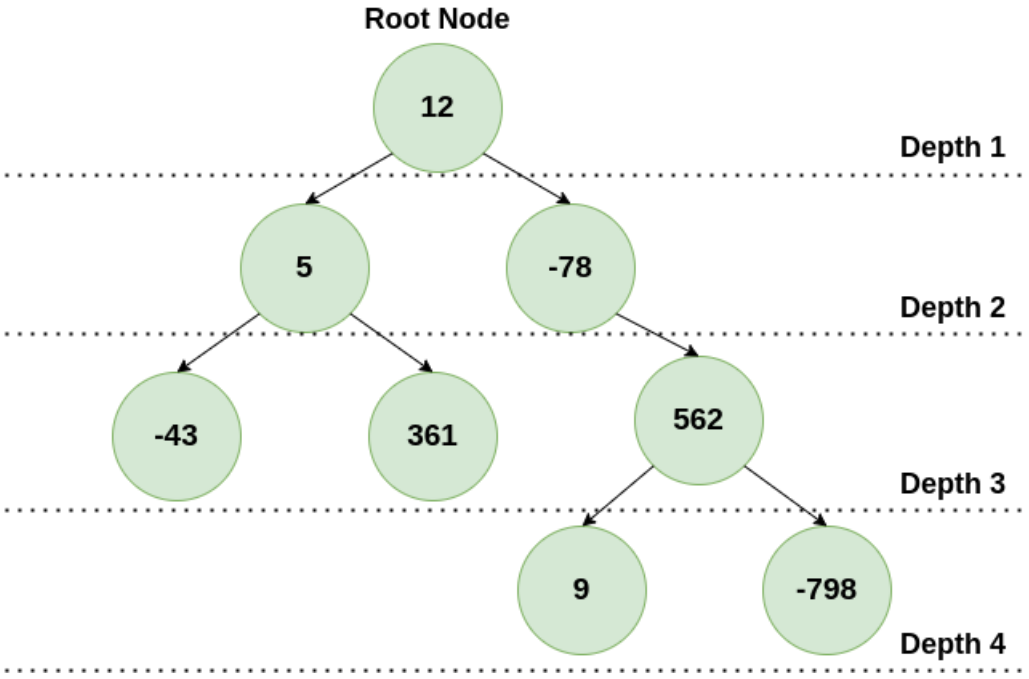
Output:
Output according to the test case.

Notes and Tips:

- The **linked_list_search** function will receive the address of the head node on register a0 and the value being searched on register a1, and must return on the register a0 the index of the node, if found, or -1 otherwise.
- The fields of the linked list node struct are VAL1, VAL2 and NEXT, in this order. VAL1 and VAL2 are the **signed** int values stored on the node and NEXT is the pointer to the next node on the linked list. If there isn't a next node, it will be a NULL pointer.
- To check if the received value is on the current node, the comparison VAL1 + VAL2 = received value must be made.
- A NULL pointer is represented by the value 0.
- The indexing of the list nodes starts at 0 (i.e. the head_node has index 0).
- All nodes will have different sum values.
- You can test your code using the simulator's assistant from this link.

Ex. 7.8: ABI-compliant recursive binary tree search

Instructions:
In this activity, you must write a program in RISC-V assembly language that has a ABI-compliant recursive function **int recursive_tree_search(Node *root, int val).** Your code will be linked to a code in C that calls the **recursive_tree_search** function and expects the return value to be the depth of the value if the searched value is present on the tree, 0 if not.



The image above shows an example of a binary tree. If the value 361 was being searched, 3 would be the return value of the function, as it is the depth of the value in the tree.

Each tree node is a struct that contains a value on its first position, pointers to the left child and right child on the second and third positions, respectively. In case one of the children does not exist, the pointer will be NULL.
***lib.h*:**

```
typedef struct Node {
    int val;
    struct Node *left, *right;
} Node;

int recursive_tree_search(Node *root_node, int val);
void puts ( const char *str );
char *gets ( char *str );
int atoi (const char *str);
char *itoa ( int value, char *str, int base );
void exit(int code);
```

Input:
Your function will receive the address of the root of the tree on register a0 and the value being searched on register a1.

Output:
Your function must return on the register a0 the depth of the value if the searched value is present on the tree, 0 otherwise.

Examples:

Input:	12
Output:	1

Input:	562
Output:	3

Input:	-40
Output:	0

Notes and Tips:

- The root node has depth 1.
- The fields of the binary tree node struct are VAL, LEFT and RIGHT, in this order. VAL is a **signed** int value stored on the node and LEFT and RIGHT are pointers to the node's children. If a node doesn't have one of these children, the respective pointer will be NULL.
- To check if the received value is on the current node, the comparison VAL = received value must be made.
- A NULL pointer is represented by the value 0.
- All nodes will have different values.
- The utility functions implemented in exercise 7.7 are necessary in this exercise.
- You can test your code using the simulator's assistant from this link.

8. Assembly System-level Programming

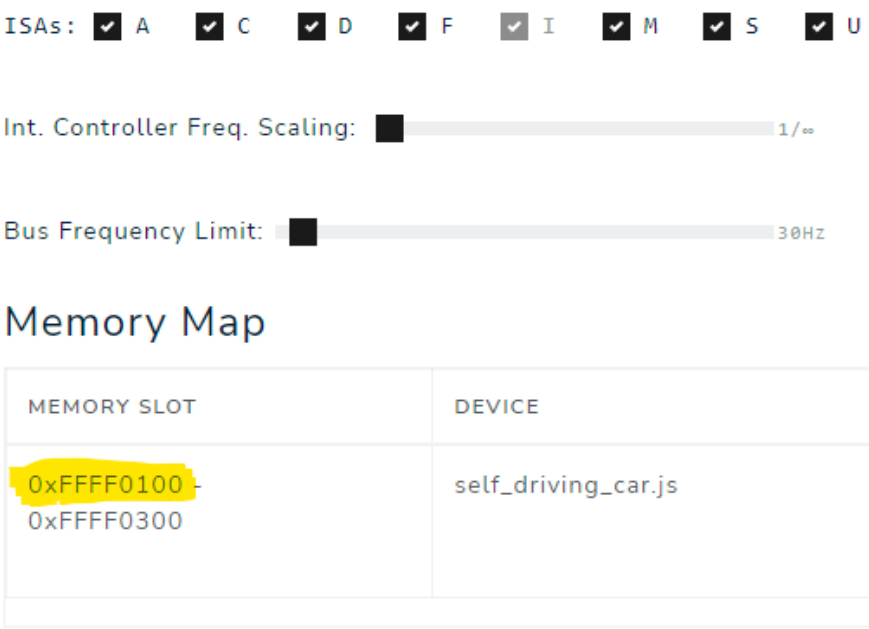
Ex. 8.1: Accessing Peripherals - Controlling the Car

Instructions:
In this activity, you must control a car to move it from a parking lot to the entrance of the Test Track (shown in the picture below) in, at most, 180 seconds. The car is an external device connected to the RISC-V processor and can be controlled by *syscalls* or MMIO.

- In this exercise, you must use only MMIO to control the car. The MMIO specifications for the peripherals of the simulator can be seen in this document.
    - You only need to use MMIO to control the steering wheel, engine, brakes and **get coordinates**.
    - The entrance to the Test Track is placed at:
        - x: 73, y: 1, z: -19
    - The base address is shown at the table "Memory Map" (see example below)
- The car will be considered to have reached its destination when it reaches a point inside a radius of 15 meters from the Test Track entrance. Your code must call the exit *syscall* to finalize the execution.

Infrastructure:
In order to use the car, first you must enable the device '*Self-Driving Car*' on the RISC-V simulator. This can be done on the tab '*Hardware*' and selecting the device *'Self-Driving Car'* as shown in the picture below.

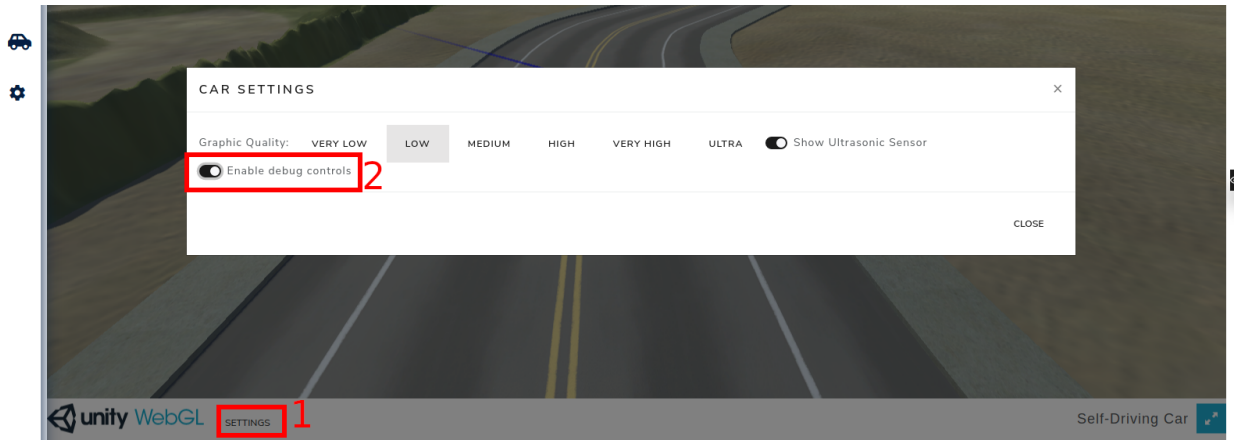
After adding the device, its **base address** will be listed on the table:



The car can be accessed through *'Self-driving car'*, on the left bar, as shown in the picture below. The car icon can take a while to appear for the first time.

Note that, after adding the device, the car will be placed in an arbitrary position. **Use the Assistant** to place the car inside the parking lot.

In addition to that, you can test moving the car manually using the arrow keys or WASD. To do so, enable the option ‘Enable debug controls’ on the simulator, as demonstrated below.



Notes and Tips:

- The starting point and destination are **fixed**. You have to use the assistant to place the car at the starting point (there is a button on the assistant that can be used for that).
- The steering wheel direction values range from -127 to 127, negative values steer to the left and positive values to the right.
- For debugging purposes, you can control the car when the rest fails, using the keys WASD or arrow keys, with the option "Enable debug controls" enabled. However, this option can't be enabled while your code is running on the simulator.
- You can test your code using the simulator's assistant from this link.

Ex. 8.2: Accessing Peripherals - Using Serial Port

Instructions:

In this activity, you will communicate with the Serial Port device via MMIO. You will need to read a stream of bytes from the Serial Port and then perform an operation on top of the data read.

Interactions with Serial Port can be done through two memory addresses:

- `base+0x00`: storing the value 1 triggers a write, writing the byte that is stored at `base+0x01`. The byte at `base+0x00` is set to 0 when write is complete.
- `base+0x02`: storing the value 1 triggers a read, reading one byte and storing it at `base+0x03`. The byte at `base+0x02` is set to 0 when read is complete.

There will be 4 different sets of operations to perform:

- **Operation 1:** read a string and write it back to Serial Port
  - **Input:** 1\n{string with variable size}\n
  - **Output:** {string with variable size}\n
- **Operation 2**: read a string and write it back to Serial Port reversed
  - **Input:** 2\n{string with variable size}\n
  - **Output:** {string with variable size reversed}\n
- **Operation 3**: read a number in decimal representation and write it back in hexadecimal representation.
  - **Input:** 3\n{decimal number with variable size}\n

    - **Output:** {number in hexadecimal}\n
- **Operation 4**: read a string that represents an algebraic expression, compute the expression and write the result to Serial Port.
    - **Input:** 5\n{number with variable size} {operator} {number with variable size}\n
    - **Output:** {operation result in decimal representation}\n
    - Operator can be **+** (add) , **-** (sub), ***** (mul) or **/** (div)

Examples:

Input:	1 Random String
Output:	Random String

Input:	2 Reversed String
Output:	gnirtS desreveR

Input:	3 1876
Output:	754

Input:	4 244 + 67
Output:	311

Input:	4 2340 / 50
Output:	46

Notes and Tips:

- You can debug your program reading from STDIN and writing to STDOUT. Before testing with the Assistant, the functions responsible for IO need to be changed in order to interact with the Serial Port.
- The final version of your program can't use *syscalls* except for the exit *syscall*.
- You can use the program stack to store strings with variable sizes.
- You can test your code using the simulator's assistant from this link.

Ex. 8.3: External Interrupts - MIDI Player

Instructions:

In this activity, you must implement a library that will be used by a MIDI audio player. The peripherals used in this activity are the MIDI Synthesizer and General Purpose Timer (GPT), and their description can be found in the ALE Peripheral MMIO Manual.
Your library will have three main functionalities:

1. Implement a *_start* function, that initializes the program's stack (see Notes and Tips for more information on how to do that), sets interrupts, and calls the *main* function (using the `jal` instruction), available in the provided application.
    - Note that the `jal` instruction will be used, as it won't be necessary to change execution mode. In future exercises, this may be necessary (and `mret` will be used instead of `jal`).
2. Program and handle external interrupts of a timer device (GPT).
    - The GPT interrupt handler must simply increment a global time counter (*_system_time*), that needs to be initialized as zero and must store the system time in milliseconds. (System time is the time that has elapsed since the system has been booted).
    - We suggest programming GPT to generate interrupts every 100 ms.
3. Provide a function *"play_note"* to access the MIDI Synthesizer peripheral through MMIO.
    - The function prototype is:
      ```
      void play_note(int ch, int inst, int note, int vel, int dur);
      ```
    - The parameters are:
        i. ch: channel;
        ii. inst: instrument ID;
        iii. note: musical note;
        iv. vel: note velocity;
        v. dur: note duration.

Notes and Tips:

- To allocate the stacks, you can declare two arrays in your program. When initializing the stack pointer (SP), remember that RISC-V stacks are full-descending.
- You must implement all the functions in a single file `lib.s`, in RISC-V assembly language.
- We provide our application in the file `midi-player.c`, that uses your library and mustn't be changed. This file must be uploaded with the .s library file.
- All functions must follow the ABI.
- All functions must be *thread-safe*. Right now, you **don't** need to understand this concept. Just ensure that your code **doesn't** use global variables (the only exceptions are the variables *_system_time*, the program and ISR stacks). Use only local variables, allocated on the stack or on registers.
- **Simulator configuration:**
    - To receive external interrupts, set the "Bus Controller Frequency Scaling" on the "Hardware" tab to $1/2^7$.
    - Also on the "Hardware" tab, add the GPT, and after that, the Sound Synthesizer (MIDI), **in this order**.
- Before beginning to test the MIDI Synthesizer, check the volume of your browser and computer.
- You must access the peripherals through MMIO, just like Exercises 8.1 and 8.2.

- Each device's `base_addr` can be seen on the table “Memory Map” in the "Hardware" tab.
- You can test your code using this simulator setup, but there isn't an assistant for this exercise.

Ex. 8.4: Software Interrupts - Controlling the Car

Instructions:
In this exercise, you will do the **same** activity as Exercise 8.1: you must control a car to move it from a parking lot to the entrance of the Test Track. However, you can no longer access the MMIO addresses directly on user code, *i.e.*, the high-level code with the control logic. You must implement a set of syscalls to control the car and use them in your control logic, that has to be executed in user mode.

Separating the control code
The following code is suggested as a starting point for your solution. You can change it however you want.

```
.text
.align 4

int_handler:
  ###### Syscall and Interrupts handler ######

  # <= Implement your syscall handler here

  csrr t0, mepc  # load return address (address of
                 # the instruction that invoked the syscall)
  addi t0, t0, 4 # adds 4 to the return address (to return after ecall)
  csrw mepc, t0  # stores the return address back on mepc
  mret           # Recover remaining context (pc <- mepc)


.globl _start
_start:

  la t0, int_handler  # Load the address of the routine that will handle interrupts
  csrw mtvec, t0      # (and syscalls) on the register MTVEC to set
                      # the interrupt array.

# Write here the code to change to user mode and call the function
# user_main (defined in another file). Remember to initialize
# the user stack so that your program can use it.

.globl control_logic
control_logic:
  # implement your control logic here, using only the defined syscalls
```

You must create a second file, **called main.s**, that contains only the code below. This code can't be changed and will be used by the assistant to check your solution.

```
.text
.globl user_main
user_main:
  jal control_logic
infinite_loop:
```

```
j infinite_loop
```

Syscalls

You **can** implement the system calls listed below. in case your code doesn't use some of them, it is not necessary to implement it. For this exercise, you mustn't implement any additional system calls.

Syscall_set_engine_and_steering Code: 10	a0: Movement direction a1: Steering wheel angle	0 if successful and -1 if failed (invalid parameters)	Start the engine to move the car. a0's value can be -1 (go backward), 0 (off) or 1 (go forward). a1's value can range from -127 to +127, where negative values turn the steering wheel to the left and positive values to the right.
Syscall_set_handbreak Code: 11	a0: value stating if the hand brakes must be used.	-	a0 must be 1 to use hand brakes.
Sycall_read_sensors Code: 12	a0: address of an array with 256 elements that will store the values read by the luminosity sensor.	-	Read values from the luminosity sensor
Syscall_get_position Code: 15	a0: address of the variable that will store the value of x position. a1: address of the variable that will store the value of y position. a2: address of the variable that will store the value of z position.	-	Read the car's approximate position using the GPS device.

Notes and Tips:

- If you are not using the assistant, you must configure the simulator in a similar way as done for Exercise 8.1. However, with a small difference: the option "*Enable Operating System"* must stay disabled.
- You mustn't call the exit *syscall* (as there isn't an operating system, it doesn't exist).
- The *syscall* operation code must be passed via register a7, as done in previous exercises.
- You can test your code using the simulator's assistant from this link.