CS2053 - Computer Architecture

Department of Computer Science and Engineering - University of Moratuwa

Lab 4: C Programming in RISC-V

Learning Outcomes:

After completing this lab, you will be able to:

- Compile and run a C program in Ripes simulator for RISC-V architecture.
- Identify RISC-V base instruction sets and extensions.
- Compare the implementation of multiplication (*) and division (/) operators with and without the Integer Multiplication and Division extension.

Introduction

C Programming

C programming language is a high-level programming language that converts the programs written in the C language directly into the machine code of the corresponding architecture. Therefore, an executable code, which is compiled for one architecture, cannot be used in another architecture. This means that if we want to compile a C program for RISC-V architecture, we cannot use a C compiler that targets x86, x64, or any other architecture. Therefore, we have to use a platform-specific compiler. For that, we are using the GCC compiler provided by the RISC-V toolchain of PlatformIO. Instructions to configure the Ripes simulator with the compiler were given in the pre-lab sheet.

When translating the C code into machine code, several steps are involved. Firstly, the C compiler will translate the code written in C language to the assembly language instructions of the targeted architecture. Then the assembler will assemble that into object code. Finally, the linker will link different object files and produce the output executable file. As an example, if you are compiling multiple C files into one executable code, firstly all the C files will be translated into object code separately by the compiler and the assembler. Then the linker will link all those object files and used libraries together to produce the executable file.

The general command to compile a C program using the GNU Compiler Collection (GCC) is,

For Windows: `gcc <file_name.c> -o <exe_name>.exe`

For Linux: `gcc <file_name.c> -o <exe_name>`

To stop the translation after generating the assembly instructions, you can use the `-S` flag as follows:

```
gcc -S <file_name>.c -o <asm_name>.s
```

RISC-V Base Instruction Sets and Extensions

Following are the base instruction sets of RISC-V architecture:

- **RV32I** - 32-bit wide integer registers.
- **RV32E** - Reduced version of RV32I for embedded systems.
- **RV64I** - 64-bit wide integer registers.
- **RV128I** - 128-bit wide integer registers.

Following are four of the main extensions to the base instruction sets:

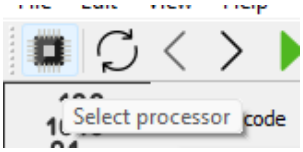
- **M** - Extension for integer multiplication and division.
- **F** - Extension for single-precision floating-point numbers.
- **D** - Extension for double-precision floating-point numbers.
- **A** - Extension for atomic instructions.

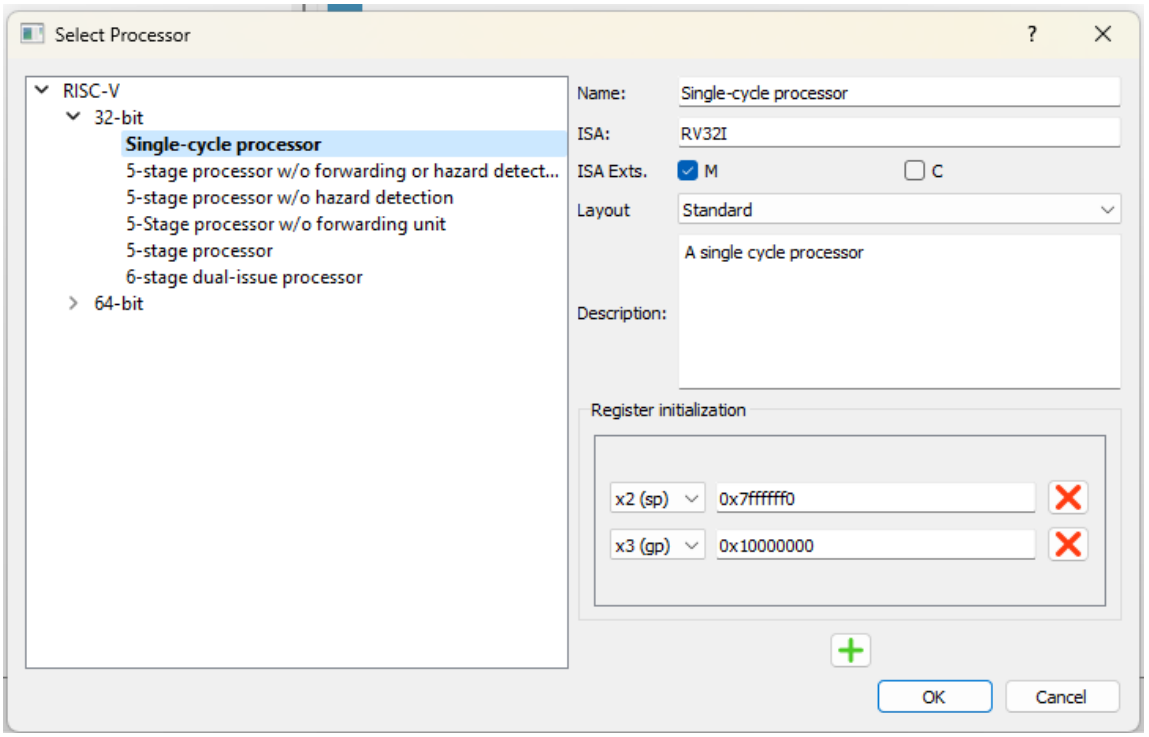
You can find more details about RISC-V base instruction sets and extensions in the RISC-V specification.

Example

Download the *Lab04_Example.c* file from Moodle.

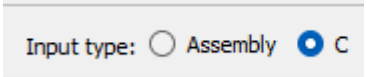
Firstly, make sure that you are using the RV32I base instruction set with the M extension. To do that, click the Select Processor icon.



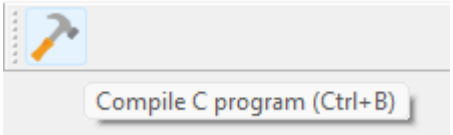


Make sure that you are using the 32-bit Single-cycle processor with the check box for **M** ISA Extension ticked as in the screenshot.

Change the input type to C.



Open the example in the Ripes editor and click the compile button.



It will compile the C code into machine code. You are able to see the disassembled code in the assembly language.

Observe that the multiplication and division operations are performed using the `mul` and `div` instructions defined in the M extension. You can find the instructions in the M extension from the RISC-V reference card.

```
10174:    02f707b3        mul x15 x14 x15
1018c:    02f747b3        div x15 x14 x15
```

Now remove the M extension by unchecking the checkbox in the select processor window.

Observe that the C compiler has defined and used two new assembly functions `__mulsi3` and `__divsi3` with some other helper functions. Since we do not have separate instructions for multiplication and division, instead, the compiler uses functions to calculate those in an iterative manner.

```
10174:    15c000ef        jal x1 348 <__mulsi3>
10190:    164000ef        jal x1 356 <__divsi3>
```

The `arr_sum` function calculates the sum of the array using recursion.

```
25 int arr_sum(int arr[], int index, int arr_size) {
26     if (index >= arr_size)
27         return 0;
28 
29     return arr[index] + arr_sum(arr, index + 1, arr_size);
30 }
```

We have defined the `print_int` function to print an integer to the console. In this function, we have directly instructed the C compiler using the assembly language to avoid the complexities in the machine code. Alternatively, you can use the `printf` function to print the integer and the new line as follows where `num` is the integer variable to be printed.

```
printf("%d\n", num);
```

Replace the calls to the `print_int` function with `printf` and observe the machine code. This will add the implementation of the `printf` function with the other required helper functions for the above task. Both the calls to `printf` and the custom `print_int` functions will perform the same task in this program. But `printf` is a general function whereas `print_int` is a task-specific function with a lesser number of steps.

When the performance of a program is critical, we can use assembly instructions within a C program to achieve higher performance. However, the same C program will not be able to be compiled into different architectures since the assembly language is architecture-specific. For example, if we use the `printf` function in this program, we can use a compiler targeted for x86 architecture to compile it but if we use the `print_int` function, we cannot.

★ As you may have noticed from the disassembled machine code, the program starts execution from the `_start` function rather than the `main` function.

```
00010090 <_start>:
   10090:       00002197        auipc x3 0x2
   10094:       f6018193        addi x3 x3 -160
   10098:       c3418513        addi x10 x3 -972
   1009c:       c5018613        addi x12 x3 -944
   100a0:       40a60633        sub x12 x12 x10
   100a4:       00000593        addi x11 x0 0
   100a8:       43c000ef        jal x1 1084 <memset>
   100ac:       00000517        auipc x10 0x0
   100b0:       34450513        addi x10 x10 836
   100b4:       2f4000ef        jal x1 756 <atexit>
   100b8:       398000ef        jal x1 920 <__libc_init_array>
   100bc:       00012503        lw x10 0 x2
   100c0:       00410593        addi x11 x2 4
   100c4:       00000613        addi x12 x0 0
   100c8:       07c000ef        jal x1 124 <main>
   100cc:       2f00006f        jal x0 752 <exit>
```

Even though the `main` function is considered the entry point to the program from the programmer's perspective, the execution starts from the `_start` function. The `_start` function will call the `main` function after performing the required initialisations. The implementation of the `_start` function is done in the assembly language and is linked during the compilation process. You can find the implementation of a sample `_start` function in the `<User folder>\.platformio\packages\framework-wd-riscv-sdk\board \nexys_a7_eh1\startup.S` assembly file.

Exercise 1

Write a program in C that computes the factorial of a given non-negative number `n`, using **iterative multiplications**. While you should test your program for multiple values of `n`, your final submission should be for `n = 7`. The program should print out the value of `factorial(n)` at the end of the program. `n` should be a variable that is statically defined within the program. Name the program **factorial.c**.

Exercise 2

Write a program in C that computes the factorial of a given non-negative number `n`, by means of a **recursive function**. While you should test your program for multiple values of `n`, your final submissions should be for `n = 7`. The program should print out the value of factorial(n) at the end of the program. n should be a variable that is statically defined within the program. Name the program **factorial_rec.c**.

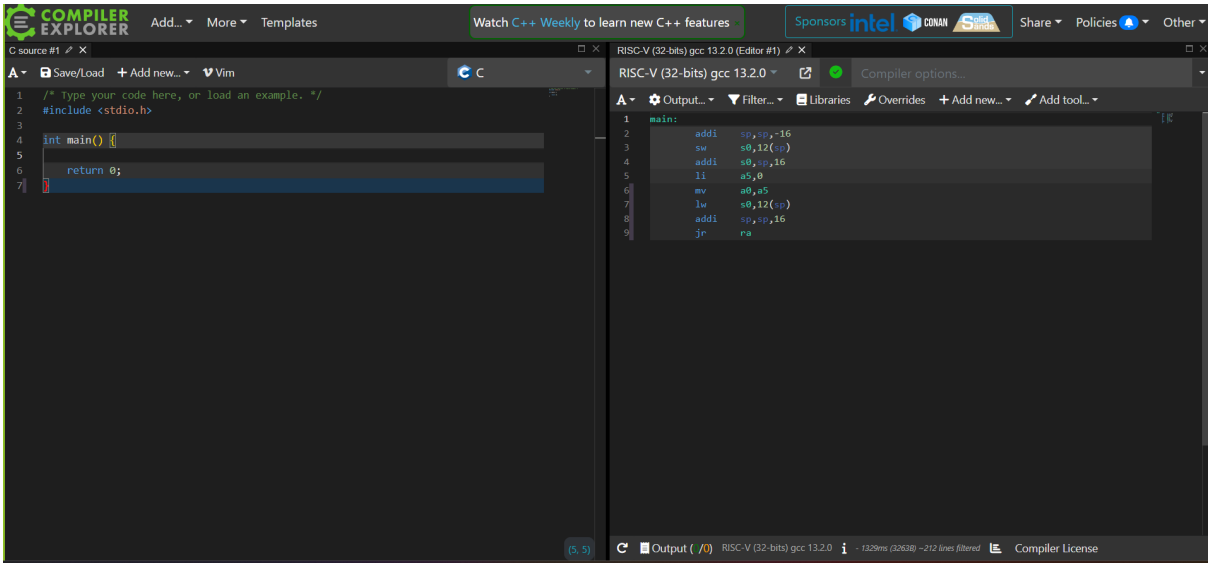
Additional Exercise: Exploring RISC-V Compiler Performance with Godbolt

Introduction to Godbolt Compiler Explorer

Godbolt is a web-based tool that allows you to compare and analyse the assembly output of different compilers and compiler options. It is a valuable resource for understanding how your code is translated into machine instructions and how different optimizations impact performance.

Basic Setup

1. Open your web browser and navigate to https://godbolt.org/.
2. You'll see a split-screen interface. On the left side, you can write and edit your C code. On the right side, you'll see the corresponding assembly code generated by different compilers and options.



Task

Let's start by writing a simple C program to test how different compiler options affect the assembly output. Copy and paste the following C code into the left pane:

```
#include <stdio.h>

int main() {
    int a = 5;
    int b = 7;
    int result = a * b;
    printf("Result: %d\n", result);
    return 0;
}
```

1. In the Godbolt interface, notice the "Compiler" dropdown menu at the top of the right pane. By default, it will show the compiler being used for your code.
2. Type `-march=rv32imd` in the 'Compiler Options' text box. This option tells the compiler to generate machine code for the RISC-V 32-bit base instruction set with multiplication and floating point extensions.

Observations

- Observe the changes in the assembly code on the right pane as you switch between different compiler options.
- Recompile the code without the -march flag and observe the differences in the generated assembly code.
- Change the compiler to 'x86 gcc 1.27' and observe the difference in code length. Compare the output with the assembly code generated from the RISC-V (32 bits) gcc compiler.

Through this exercise, you will be able to explore the Godbolt tool and witness how compiler options can impact the assembly output. This understanding is crucial for writing efficient code.

Additional Reading

[1] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[2] https://www.sifive.com/blog/all-aboard-part-1-compiler-args