Lab 5: Compiler Optimization

Learning Outcomes:

After completing this lab, you will be able to:
- Define what compiler optimization entails and its significance in computer architecture.
- Compare and contrast the output assembly code of optimised versus non-optimized program versions.
- Explain the need for and the methods of instruction compression in RISC-V & recognize the patterns in machine code that are suitable for compression.
- Enumerate the pros and cons of higher code density when optimising programs.

Introduction

Compiler Optimization

A compiler's primary task is to translate high-level programming language code , which is human readable, into machine code suitable for execution on a specific architecture. But given the vast array of possible translations, there's often room for improvement in the translated code. This is where optimization steps in.

In this lab we will be exploring 3 such optimization techniques.

01. Compiler Optimization flags

Compiler optimization flags are command-line options provided by compilers to control the optimization process of transforming source code into machine code. These flags instruct the compiler on how aggressively it should optimise, and in what manner, to potentially achieve better runtime performance, reduced code size, or a balance between the two.

When using a compiler, such as gcc, a developer can specify different levels and types of optimizations using these flags. For example:

-O0: No optimization. This level compiles the fastest and generates the most straightforward, albeit often least performant, machine code.(default)

-O1: Basic optimization. This level reduces code size and execution time without taking an excessive amount of compilation time.

-O2: Further optimization. It includes almost all recommended optimizations that do not involve a space-speed trade-off.

-O3: Full optimization. This applies even more optimizations, including those that might increase the size of the generated code in pursuit of faster execution.

-Os: Optimise for size. This prioritises reducing the code size over execution speed.

02. <u>Instruction Compression</u>

In many modern architectures, including RISC (Reduced Instruction Set Computing) designs, there's often a fixed instruction length, ensuring simplicity in fetching and decoding operations. However, not all instructions need the full width provided, leading to potential inefficiencies in memory usage.

Instruction compression aims to address this by:

- Identifying Common Patterns: By analysing frequently used instruction sequences or patterns, these can be represented in a compressed form.
- Variable-length Encoding: Instead of having a fixed length for every instruction, compressed instructions might use variable-length encoding, where frequent instructions are represented using fewer bits.
- Decompression Mechanism: For execution, compressed instructions need to be decompressed. This decompression happens either in hardware (before the instruction is executed) or via specialised software routines.

03. <u>Instruction level optimization</u>

Instruction-level optimization refers to the process of enhancing the efficiency and performance of individual instructions in a program, often within the context of a particular instruction set architecture (ISA). These optimizations are crucial because they directly impact the speed, power consumption, and overall efficiency of code execution on a hardware platform.

There are multiple techniques we can use to improve performance using instruction level optimization. Below are a few such techniques.

- Loop Unrolling
  This optimization increases the loop body's size by replicating its content multiple times, reducing the overhead of loop control.
  Ex: For a simple loop iterating four times to add elements of two arrays, instead of checking the loop condition four times, loop unrolling can make it so the condition is checked once, and the addition operation is executed four times consecutively.
- Function Inlining
  Function inlining involves replacing a function call with the actual body of the function, thereby eliminating the overhead associated with the call. In other words, instead of jumping to a different location in memory each time the function is called, the instructions are directly present at the call site.
- Precompute values
  Precomputing involves evaluating expressions during the compilation process and replacing them with their computed results in the generated code. This is especially beneficial when dealing with invariant values inside loops or frequently called functions.

For more details on compiler optimization, refer to the following documents.
https://caiorss.github.io/C-Cpp-Notes/compiler-flags-options.html
https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Exercise 1

Copy and paste the following code (which is similar to the program you wrote in lab 4) into the Ripes source code editor. **Make the input type as C** programming language.

```c
#include <stdio.h>

int factorial(int n) {
    if (n == 0 || n == 1) {
        return 1;
    }
    return n * factorial(n - 1);
}

int main() {
    int n = 10; // Try changing this value and compile
    with different optimization flags
    int result = factorial(n);
    printf("%d\n", result);
    return 0;
}
```
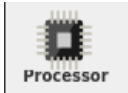
Open the Settings dialog box at **Edit -> Settings**.
In the **Compiler tab**, you can observe that the **Compiler path** is set to the RISC-V gcc compiler from PlatformIO. (Step done in Lab 4).
Note the **-O** flag given as a **Compiler argument**. It refers to the level of optimization to be done by the gcc compiler at compile time. There are several levels of optimization such as -O0, -O1, -O2, -O3.
Set the optimization flag as -O0 and click OK.

Now **build your C program** by clicking on the icon. The code will be compiled with the optimization level of -O0.

Open the **processor view** by clicking on the icon on the left sidebar.

Execute your compiled code by clicking on the icon from the toolbar on top.
Observe the **number of cycles** utilized by the program which is given under the **Execution info** section at the bottom right of the window.
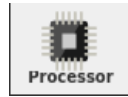
Do the same steps with the optimization level compiler argument changed to -O1, -O2, and -O3. Observe the difference in CPU cycles utilization.

What can you conclude from your observations?

Exercise 2

Load the **matrixmul example** from Ripes examples(File->Load Example->C->matrixmul.c)

Examine the code to see how matrix multiplication is implemented using three loops in C.

Open the **processor view** by clicking on the [Processor] icon on the left sidebar.

First **deselect** the ISA extension flag **'C'** which stands for Compressed instructions. Then the compressed instructions are not used.

Then compile your C program by clicking on the [hammer] icon. Examine the assembly instructions generated.

Then save the binary file (File->Save file as->(Select assembled/compiled binary and a put a name). A .bin file should be saved.

Again do the same, but **select** the 'C' flag from processor view to use compressed instructions.

Examine the assembly instructions carefully from the 'disassembled' view mode and you should see compressed instructions marked 'c. <instruction>'. Switch to 'binary' view mode and you'll see that the size of compressed instructions are smaller.

Save the compiled binary again and compare the file sizes.

What are your observations and what might be the reasons for them?

Additional Exercise

Optimise for loop at code level

Let's consider the following for loop.

$$for\ (i=0;\ i<1000;\ i++)$$
$$x[i] = x[i] + s$$

s - fixed integer

1. First write a basic RISC-V code for the above task.
2. Next try to reduce branches.
3. As the next stage try to eliminate induction variables (i).
   a. Hint - Use memory addresses instead of induction variables.
4. Use loop unrolling to optimise more.