Architetture dei Sistemi di Elaborazione O2GOLOV	Delivery date: Until 2 AM, November 6th 2024
Laboratory 5	Expected delivery of lab_5.zip must include: This file in pdf format.

Exercise 1:

Software Optimizations

Starting from Exercise 2 of Lab 4, you are required to further speedup the benchmark (*my_c_benchmark*).

For readability, provide the previously used configurations (Cut & Paste).

Parameters	**Configuration 1**	**Configuration 2**	**Configuration 3**	**Configuration 4**
First changed paramenter	the_cpu.fetchWidth = 0xc1a0	the_cpu.issueWidth = 2	the_cpu.fetchWidth = 8	the_cpu.dispatchWidth = 3
Second changed paramenter	the_cpu.dispatchWidth =1	the_cpu.numPhysIntRegs = 70	tag_latency = 1	the_cpu.smtROBThreshold = 90
…		The_cpu.fetchQueueSize = 16	data_latency = 1	The_cpu.numPhysVecPredRegs = 64
				The_cpu.fetcQueueSize = 64

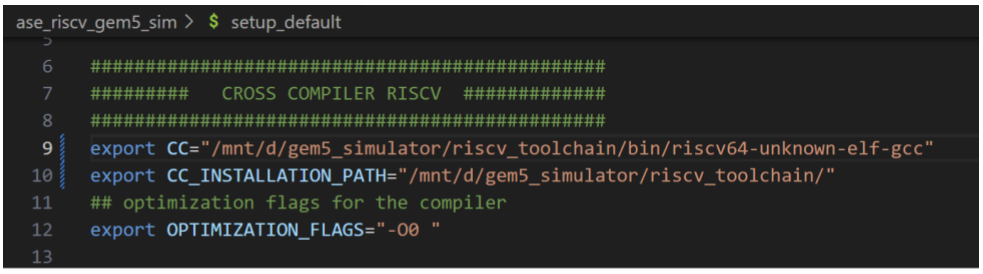
Original CPI (no hardware optimization):

	Configuration 1	**Configuration 2**	**Configuration 3**	**Configuration 4**
CPI	2.083105	2.180444	2.197422	2.186703
Speedup (w.r.t. the Original CPI)	1	0.9934	0.9967	1.0016

Despite the hardware enhancements for increasing the CPU performance, remember that optimizing compilers for programs in high-level code also exist. The aim of optimizing compilers is to minimize or maximize some attributes of an executable computer program (code size, performance, etc.). They are also aware of hardware enhancements to perform very accurate optimizations.

Compilers can be your best friend (or worst enemy!). The more information you provide in your program, the better the optimized program will be.

You can compile your programs with different SW optimization strategies and/or additional features. In the *setup_default* file:



```
ase_riscv_gem5_sim > $ setup_default
 6  ################################################
 7  #########   CROSS COMPILER RISCV   #############
 8  ################################################
 9  export CC="/mnt/d/gem5_simulator/riscv_toolchain/bin/riscv64-unknown-elf-gcc"
10  export CC_INSTALLATION_PATH="/mnt/d/gem5_simulator/riscv_toolchain/"
11  ## optimization flags for the compiler
12  export OPTIMIZATION_FLAGS="-O0 "
13
```

You can change the line 12.

Simulate the program for different optimization levels and collect statistics. You are required to change the OPTIMIZATION_FLAGS variable in the *setup_default*. O0 is the default value, you need to change the optimization value accordingly to the values in parenthesis in the following Table.

<u>DO NOT CONFUSE -O3 WITH O3 PROCESSOR.</u>

TABLE1: IPC for different compiler optimization levels and configurations

Optimization / Configuration	Opt lvl 0 (-O0)	Opt lvl 1 (-O1)	Opt lvl 2 (-O2)	Opt size (-Os)	Opt lvl 3 (-O3)	Opt lvl 2 (-O2 --fast-math)
Original Configuration	0.480053	0.396446	0.443626	0.415027	0.443626	0.458622
Configuration 1	0.480053	0.396446	0.443626	0.415027	0.443626	0.458622
Configuration 2	0.479800	0.450377	0.442308	0.415413	0.442308	0.457604
Configuration 3						
Configuration 4	0.525302	0.441512	0.495103	0.448781	0.495103	0.519635
Configuration 5	0.479966	0.396991	0.442308	0.415510	0.442308	0.457721
Program Size [Bytes]	3228	3044	3032	3016	3032	3032

Regarding the Program Size (Code and Data!!), you can retrieve the size from:

```
~/ase_riscv_gem5_sim$ /opt/riscv-2023.10.18/bin/riscv64-unknown-elf-size -format=gnu
-radix=10 ./programs/my_c_benchmark/my_c_benchmark.elf
```

For brave and curious guys:
For visualize the enabled optimizations from the compiler perspective, you can run:

```
~/my_gem5Dir$ /opt/riscv-2023.10.18/bin/riscv64-unknown-elf-gcc -Q -O2 --help=optimizers
```

By changing the "-O2" parameter with the desired one, you will find the enabled/disabled optimizations.
Here are some possible types of optimizations:

- https://en.wikipedia.org/wiki/Optimizing_compiler
- https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

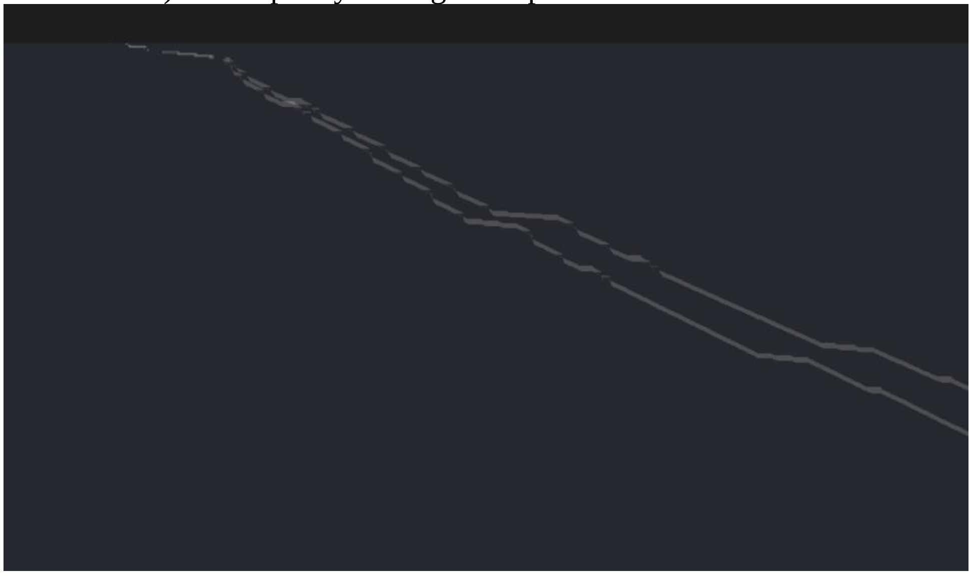
Exercise 2:

Given your benchmark (*`my_c_benchmark.c`*), select the best optimization to obtain **your best angle of optimization,** compared to the baseline configuration **(***`riscv_o3_custom.py; -O0`***).**

1. Based on Table 1 (from Exercise 1), select the best optimization (for example, the green box corresponding to Configuration 1 with -O2).

Optimization / Configuration	Opt lvl 0 (-O0)	Opt lvl 1 (-O1)	Opt lvl 2 (-O2)	Opt size (-Os)	Opt lvl 3 (-O3)	Opt lvl 2 (-O2 --fast-math)
Original Configuration						
Configuration 1		Worst IPC				
Configuration 2						
Configuration 3						
Configuration 4	Best IPC					
Configuration 5						
Program Size [Bytes]						

2. By using **Konata**, overlap the two pipelines (the original obtained with *`riscv_o3_custom.py`* and the optimized corresponding to the best SW-HW combination) to compute your angle of optimization.

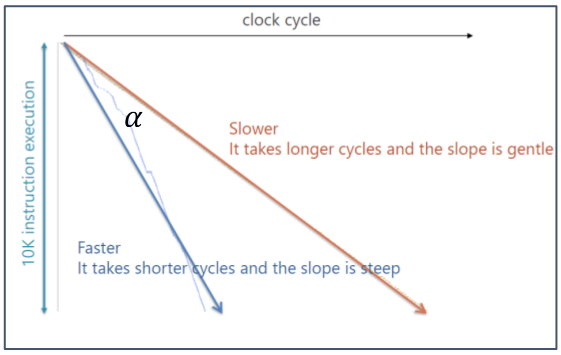


Compute the angle α (named optimization angle) existing between the traces.

Hint: To load different traces in **Konata, load them separately.** Afterward, **righ-click in the pipeline visualizer and select "transparent mode". You need to adjust the scale!**

3. To compute the **angle of optimization $\boldsymbol{\alpha}$**:

$$\alpha = \arctan\left(\frac{ClockCycles_{baseline}}{Instructions_{baseline}}\right) - \arctan\left(\frac{ClockCycles_{optimized}}{Instructions_{optimized}}\right)$$



clock cycle
10K instruction execution
α
Slower
It takes longer cycles and the slope is gentle
Faster
It takes shorter cycles and the slope is steep

The angle of optimization is equal to:
In the best performing program: the clock cycles count is 15264 with 8055 instructions
In the original program: the clock cycles count is 16694 with 8055 instructions

The angle alpha = 2.063447546

4. Do you see any visual improvements (for example, a less discontinued pipeline)? Yes, why? No, why? What is happening? How they could be improved?

No, there don’t appear to be visual improvements in pipeline continuity. The optimization focused on reducing the number of clock cycles rather than achieving a smoother flow in the pipeline. This means the program runs faster, but the pipeline still shows gaps, indicating stages where execution stalls. To improve continuity, techniques like out-of-order execution or better instruction scheduling could help reduce these gaps, allowing for a more consistent instruction flow.