Architetture dei Sistemi di Elaborazione **O2GOLOV**	Delivery date: 23 November 2023
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.branchPred = predictor.create_BiModeBP()	FloatCmp.opLat = 1	the_cpu.fetchWidth = 8	L1Cache.tag_latency = 1
Second changed paramenter		FloatCvt.opLat = 1	the_cpu.decodeWidth = 4	L1Cache.data_latency = 1
…		FloatMult.opLat = 3	the_cpu.renameWidth = 4	L1Cache.response_latency = 1
		FloatMultAcc.opLat = 3	the_cpu.dispatchWidth = 4	

Original CPI (no hardware optimization): 2.083105

	Configuration 1	**Configuration 2**	**Configuration 3**	**Configuration 4**
CPI	2.020090	2.026204	1.967806	1.889194
Speedup (wrt Original CPI)	1.03119	1.028082	1.058593	1.102642

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	0.480	0.396	0.444	0.415	0.444	0.459
Configuration 1	0.495	0.419	0.447	0.418	0.447	0.460
Configuration 2	0.494	0.417	0.446	0.421	0.446	0.466
Configuration 3	0.508	0.421	0.450	0.430	0.450	0.467
Configuration 4	0.529	0.450	0.500	0.454	0.500	0.525
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

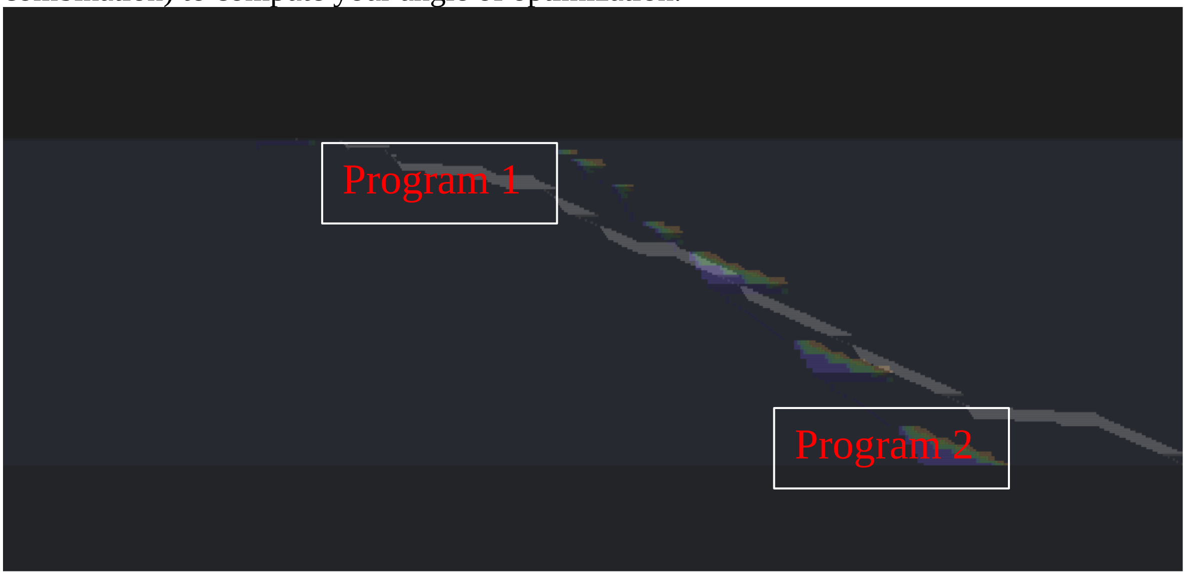
Exercise 2:

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	0.480	0.396	0.444	0.415	0.444	0.459
Configuration 1	0.495	0.419	0.447	0.418	0.447	0.460
Configuration 2	0.494	0.417	0.446	0.421	0.446	0.466
Configuration 3	0.508	0.421	0.450	0.430	0.450	0.467
Configuration 4	0.529	0.450	0.500	0.454	0.500	0.525
Program Size [Bytes]	3228	3044	3032	3016	3032	3032

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.

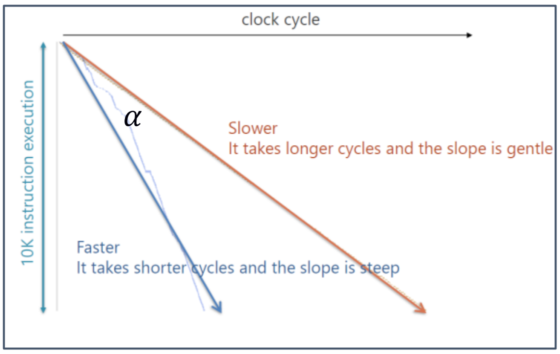


Compute the angle α (named optimization angle) existing between the traces.

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 α**:

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



The angle of optimization is equal to: $$\alpha = 2.257$$

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?

Si nota che la pipeline ottimizzata ha un angolo più "incidente"; poiché l'ottimizzazione coinvolge la latenza della cache, tutte le operazioni di load/store che coinvolgono la cache sono più brevi e impiegano meno cicli di clock per essere eseguite. Non si notano meno "discontinuità" nella pipeline: ciò concorda con il fatto che il predittore di salto è lo stesso nelle due configurazioni messe a confronto, quindi effettua le stesse predizioni.