Solution developed in collaboration with:
• Riva Governanda s359182
• Rizzo s361451
• Levo s354939

<Architetture dei Sistemi di Elaborazione
02GOLOV

Computer Architectures
02LSEYG

Laboratory 0x05

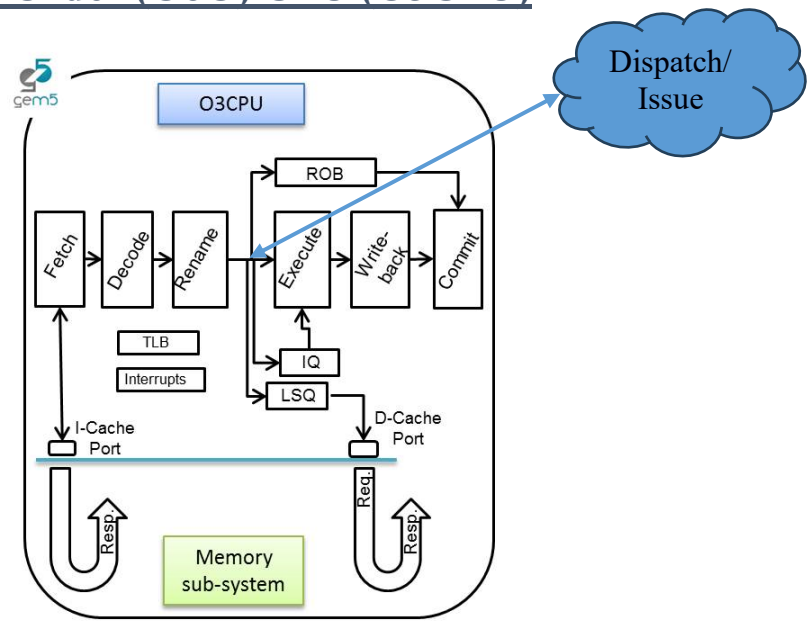
Expected delivery of **lab_05.zip** must include:
- each configuration of the custom architecture (riscv_o3_custom.py and create_predictor.py) that you modify.
- This document with all the field compiled and in PDF form.

Delivery deadlines:
GROUP1: 15/11/2025
GROUP2: 16/11/2025
GROUP3: 20/11/2025
(you can check your group on the new schedule on the portal)
THE NAME OF THE UPLOADED ZIP MUST BE "lab_05.zip"

If you completed the laboratory in collaboration with other students (maximum of three per group), include at the beginning of your report the following statement:
"Solution developed in collaboration with
[Collaborator's Last Name] [Collaborator's Student ID]"
If there are two collaborators, list both names and IDs.
Every member must upload the report.

Introduction and Background

Simulating an Out-of-Order (OoO) CPU (O3CPU)



In this laboratory, you will be able to configure an OoO CPU by using a script called riscv_o3_custom.py (in the *gem5* folder). In a few words, the script configures an Out-of-Order (O3) processor based on the *DerivO3CPU,* a superscalar processor with a reduced number of features.

Pipeline

The processor pipeline stages can be summarized as:

- **Fetch stage:** instructions are fetched from the instruction cache. The `fetchWidth` parameter sets the number of fetched instructions. This stage does branch prediction and branch target prediction.
- **Decode stage:** This stage decodes instructions and handles the execution of unconditional branches. The `decodeWidth` parameter sets the maximum number of instructions processed per clock cycle.
- **Rename stage:** As suggested by the name, registers are renamed, and the instruction is pushed to the IEW (Issue/Execute/Write Back) stage. It checks that the *Instruction Queue* (**IQ**)/*Load and Store Queue* (**LSQ**) can hold the new instruction. The maximum number of instructions processed per clock cycle is set by the `renameWidth` parameter.

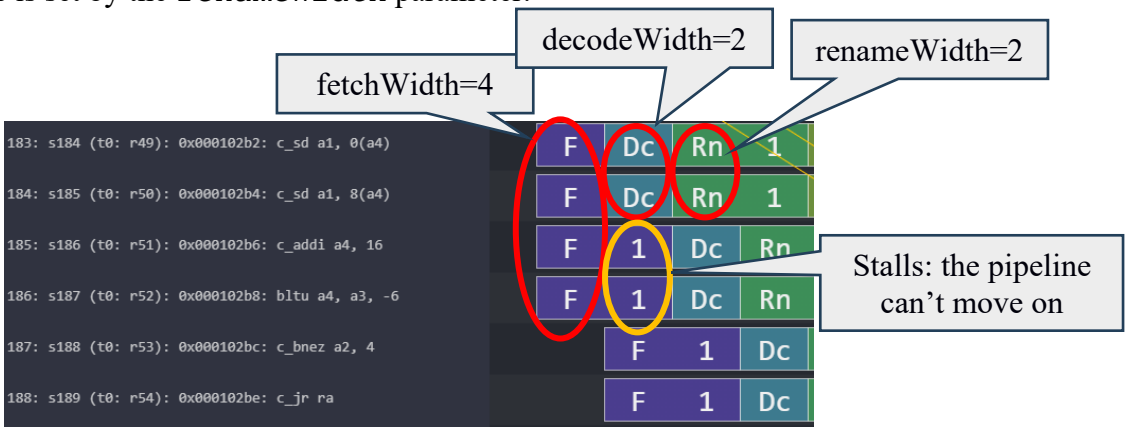


Figure 1: Understanding configurable OoO CPU parameters.

- **Dispatch stage**: instructions whose renamed operands are available are dispatched to functional units (**FU**). For loads and stores, they are dispatched to the Load/Store Queue (**LSQ**). The maximum number of instructions processed per clock cycle is set by the `dispatchWidth` parameter.
- **Issue stage**: The simulated processor has a single instruction queue from which all instructions are issued. Ordinarily, instructions are taken in-order from this queue. An instruction is issued if it does not have any dependency.
- **Execute stage:** the functional unit (**FU**) processes their instruction. Each functional unit can be configured with a different latency. Conditional branch mispredictions are identified here. The maximum number of instructions processed per clock cycle depends on the different functional units configured and their latencies.
- **Writeback stage**: it sends the result of the instruction to the reorder buffer (**ROB**). The maximum number of instructions processed per clock cycle is set by the `wbWidth` parameter.
- **Commit stage**: it processes the reorder buffer, freeing up reorder buffer entries. The maximum number of instructions processed per clock cycle is set by the `commitWidth` parameter. Commit is done in order.

In the event of a **branch misprediction**, trap, or other speculative execution event, "squashing" can occur at all stages of this pipeline. When a pending instruction is squashed, it is removed from the instruction queues, reorder buffers, requests to the instruction cache, etc.

*Figure 2: Example of a branch **misprediction** (transparent rows)*

Pipeline Resources

Additionally, it has the following structures:

- Branch predictor (BP)
  - Allows for selection between several branch predictors, including a local predictor, a global predictor, and a tournament predictor. Also has a branch target buffer (BTB) and a return address stack (RAS).
- Reorder buffer (ROB)
  - Holds instructions that have reached the back end. Handles squashing instructions and keep instructions in program order.
- Instruction queue (IQ)
  - Handles dependencies between instructions and scheduling ready instructions. Uses the **memory dependence predictor** to tell when memory operations are ready.
- Load-store queue (LSQ)
  - Holds loads and stores that have reached the back end. It hooks up to the d-cache and initiates accesses to the memory system once memory operations have been issued and executed. Also handles forwarding from stores to loads, replaying memory operations if the memory system is blocked, and detecting memory ordering violations.
- Functional units (FU)
  - Provides timing for instruction execution. Used to determine the latency of an instruction executing, as well as what instructions can issue each cycle.
  - **Floating point units, floating point registers,** and respective instructions are supported.

Figure 3: Pipeline example of FP instructions and FP registers

All the needed resources are at a GitHub repository:
https://github.com/cad-polito-it/ase_riscv_gem5_sim

To update your simulation environment:

```
~/ase_riscv_gem5_sim$ git pull
```

ATTENTION: You may have modified files that would prevent you to directly pull the workspace. Please save them and then pull the repository.

In order to simulate an OoO CPU you MUST set the following variables in your *setup_default* file:

```
22  # Select the in order CPU
23  #export GEM5_SIMULATION_SCRIPT="./gem5/riscv_in_order_hen_patt.py"
24  export GEM5_CPU_IN_ORDER=false
25  #Select one of this OoO CPU
26  export GEM5_SIMULATION_SCRIPT="./gem5/riscv_o3_custom.py"
27  #export GEM5_SIMULATION_SCRIPT="./gem5/riscv_o3_simple.py"
28  #export GEM5_SIMULATION_SCRIPT="./gem5/riscv_o3_two_level_caches.py"
29
```

This sets the python scripts modelling the OoO CPU (line 23-24 and 25).

Moreover, you must install an additional pipeline visualizer called Konata

And change the path accordingly in your *setup_default* file (bottom of the file).

```
################################################
########  PIPELINE VISUALIZER  #############
################################################
export PIPELINE_VISUALIZER=/opt/konata-v0.39/konata-linux-x64/konata # FOR LABINF
```

In the repository's README there are the instructions for installing Konata (it is necessary to download a precompiled binary for your system).

Exercise 1:

Simulate the programs written in the previous lab with the OoO CPU based on the CPU architecture described in `riscv_o3_custom.py`,
You can visualize the pipeline (i.e., load the `trace.out` file on Konata).

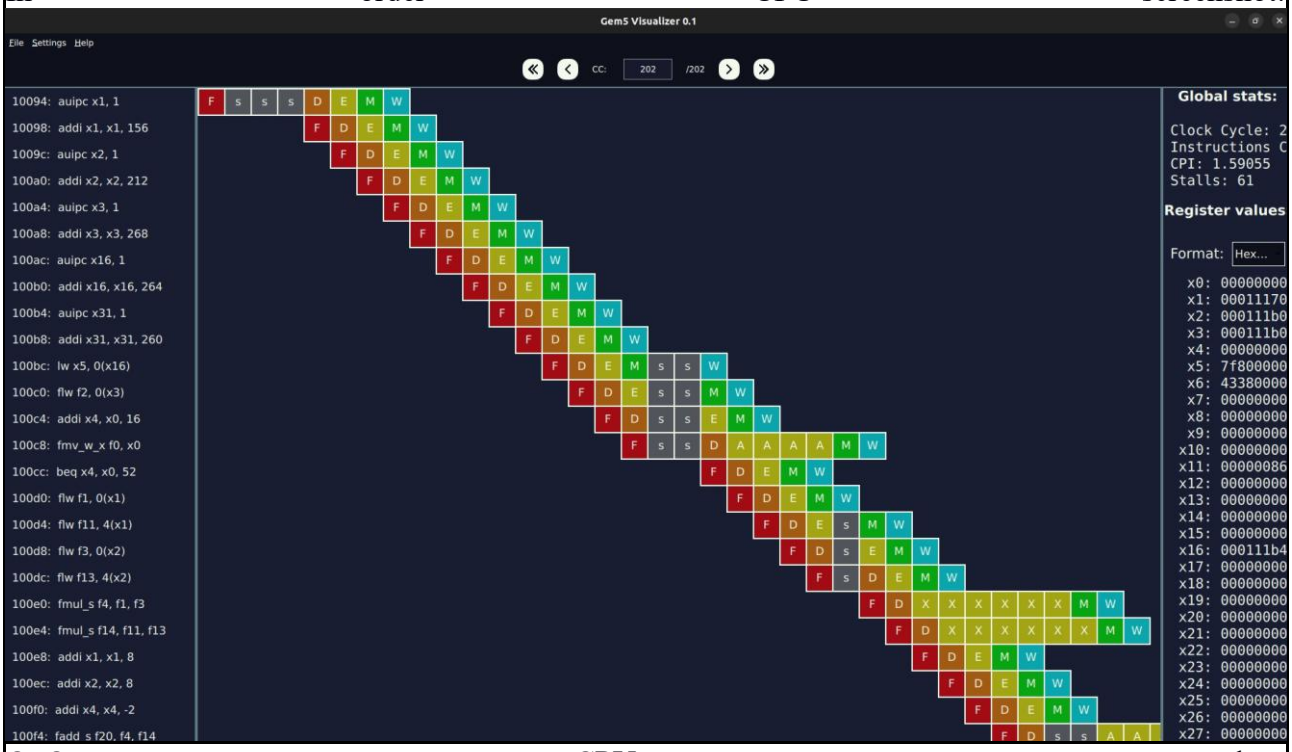
Collects statistics about the execution of your programs in the following table:

Laboratory	Program	Clock Per Instructions	
		In Order CPU	OoO CPU
0x01	program3.s	1.4774	3.1891
0x02	program1.s	2.2769	1.9677
0x03	program1.s	2.3943	1.4378
	program1_a.s	2.1111	1.3840
	program1_b.s	2.5172	1.6463
0x04	program1.s	1.8809	1.8935
	program1_a.s	1.5928	2.0964
	program1_b.s	1.5905	2.2114

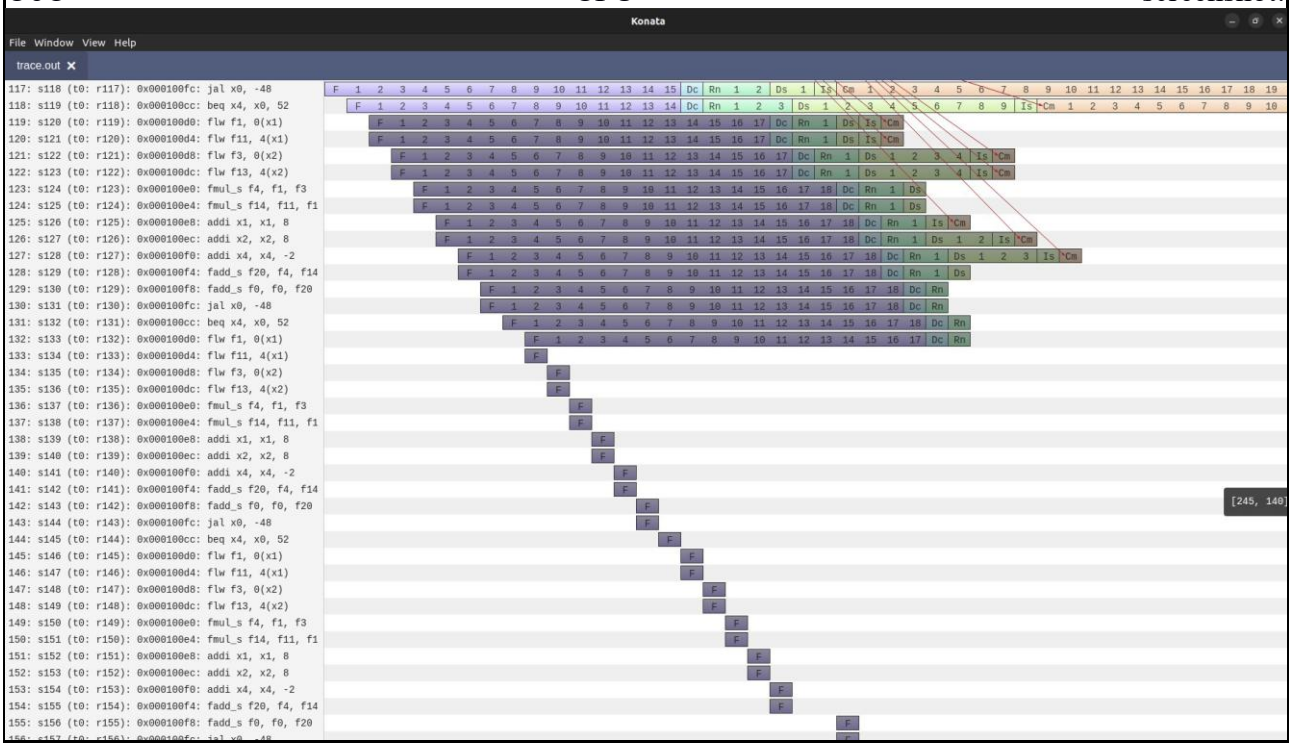
Can you identify situations (**at least three**) in which the OoO architecture perform better/worse than the In order architecture and viceversa?

Situation: OoO performs worse than In Order in lab4 program1_b.s
Explanation: in this case the parallelism of the out of order cpu cannot be exploited well because of dependencies on the multiple operations inside the loops and then we can notice a failure in the branch prediction resulting in loss of performances.

In order CPU screenshot:

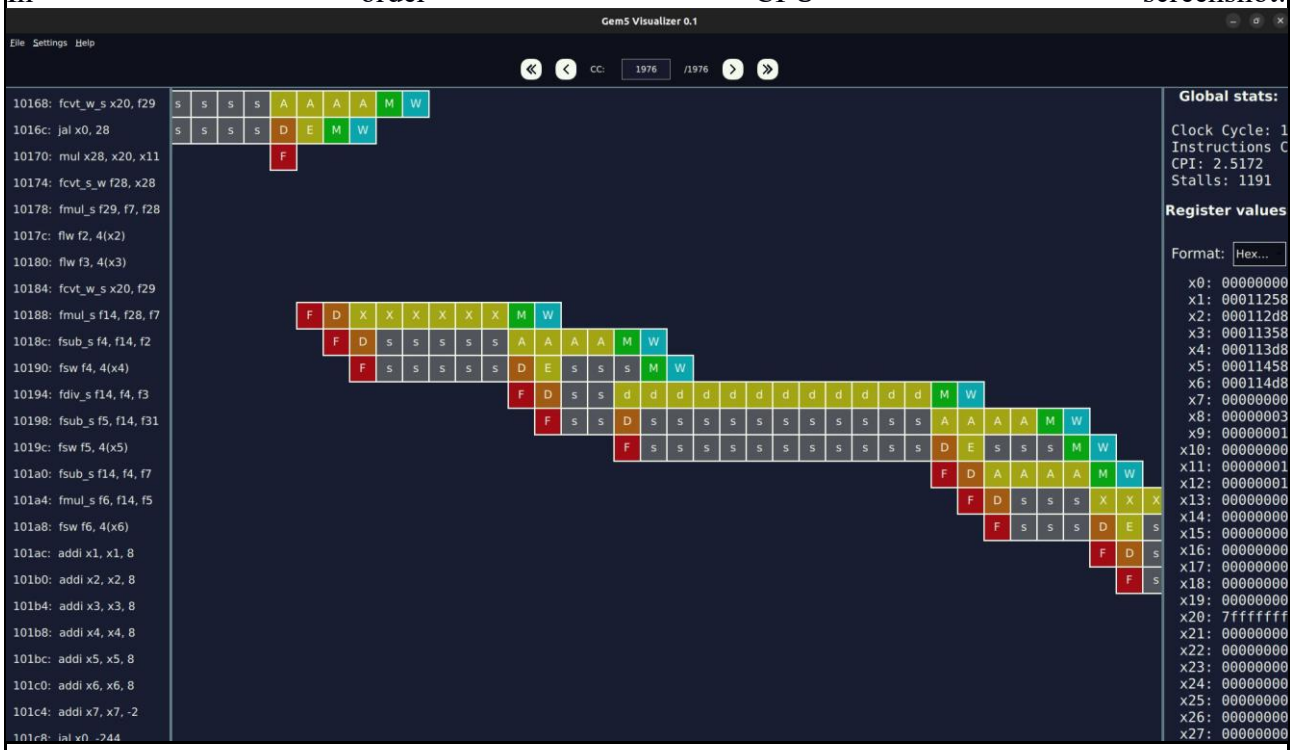


OoO CPU screenshot:

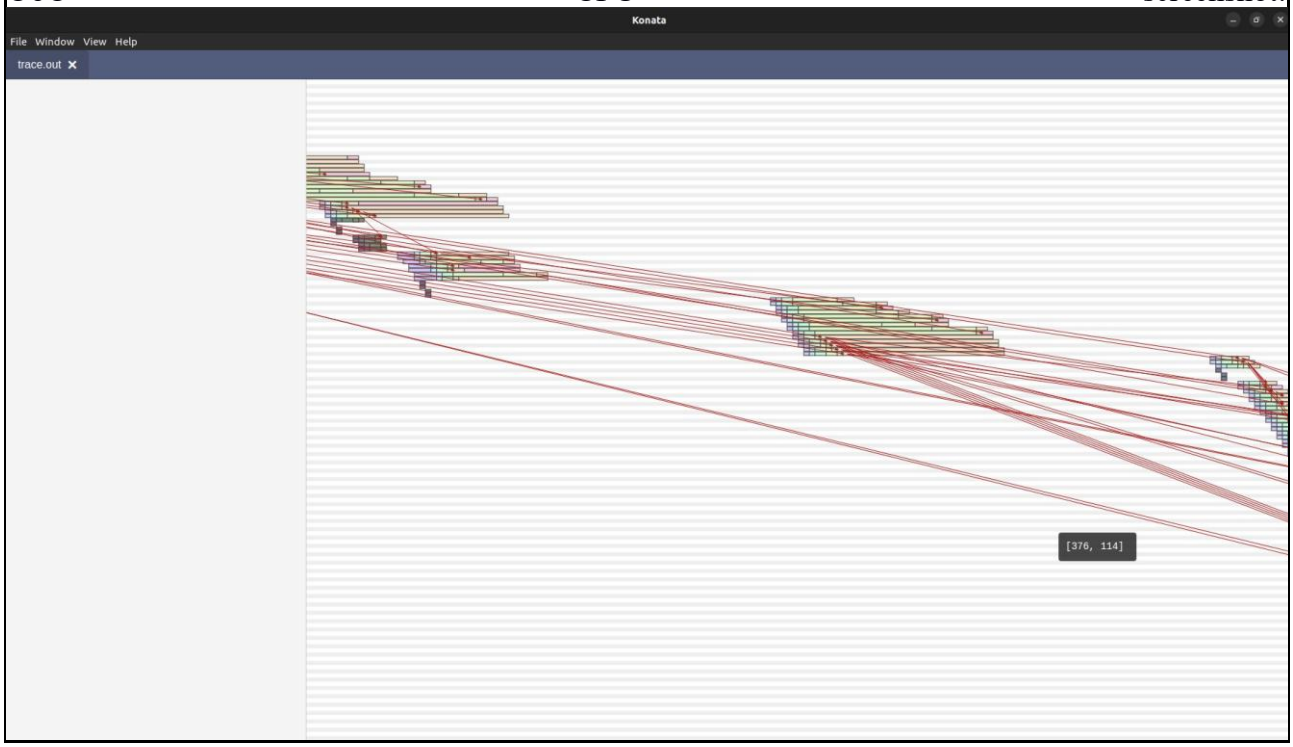


Situation: In Order performs worse than OoO in lab3 program1_b.s
Explanation: the out of order cpu here performs better and we can notice that the branch predicts are working better but we have some chache misses but somehow the in order configuration is working worse because of the stalls.

In order CPU screenshot:

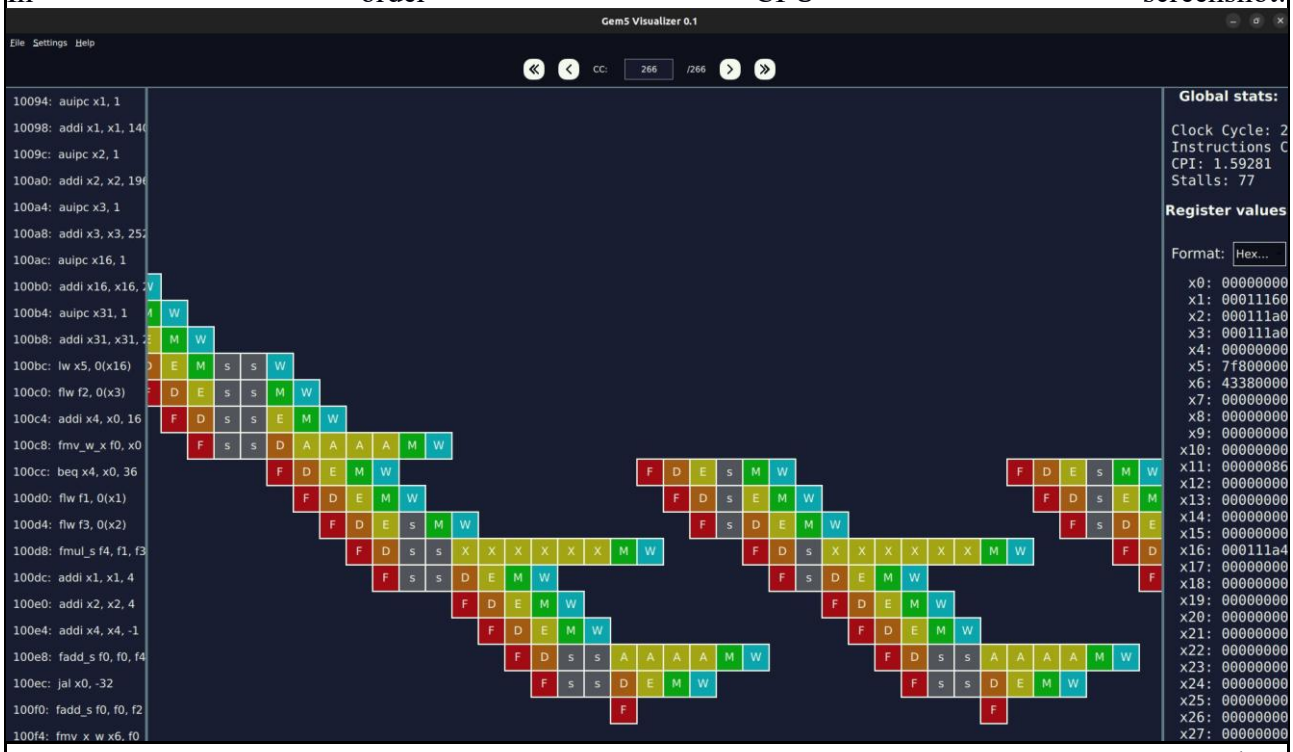


OoO CPU screenshot:

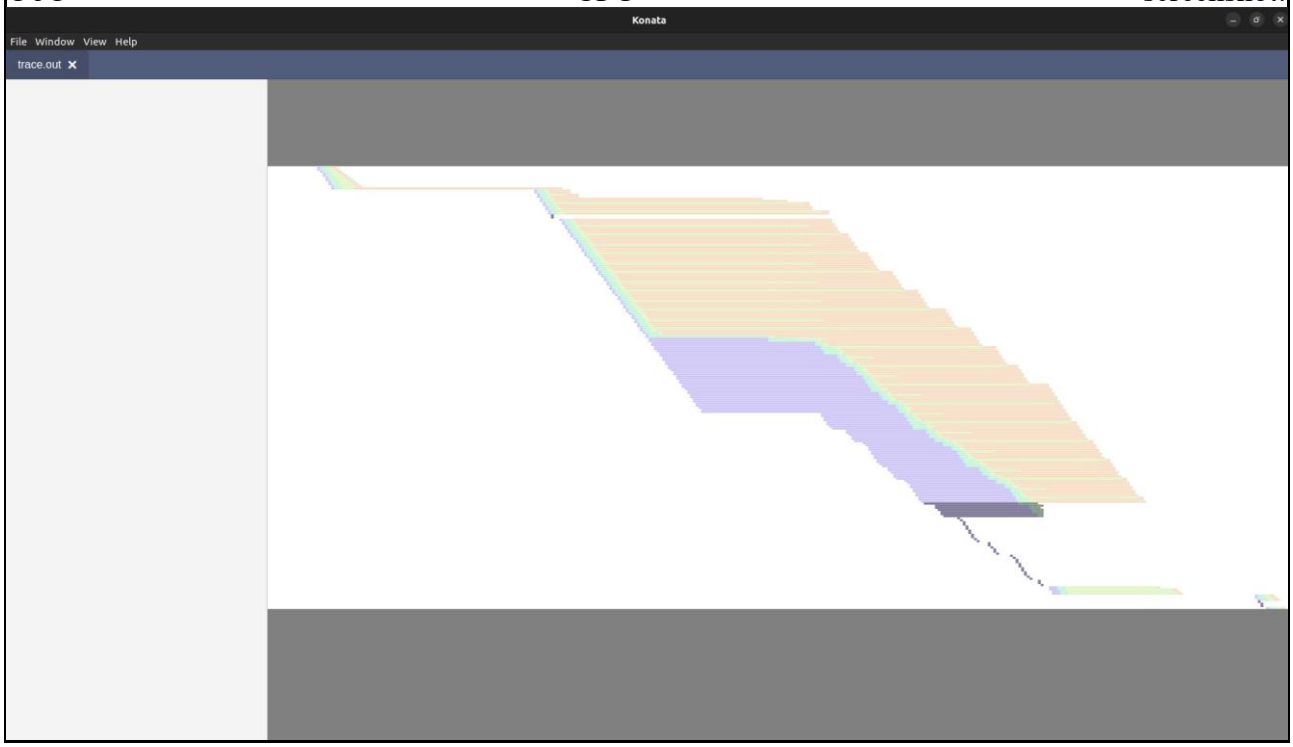


Situation: OoO performs worse than In Order in lab4 program1_a.s
Explanation:The in order cpu works better than the out of order one in this case because the program is pretty “linear” so the in order one can use the parallelism better than what the o3 do in exploiting its branch predictions and other optimization features

In order CPU screenshot:



OoO CPU screenshot:



Exercise 2:

In this exercise you will experiment the usage of different Branch Prediction Unit (BPU).

You are asked to avoid as much as possible mis-predicted branches by resorting to different BPUs. Use as benchmarks the two programs developed in lab 0x03 and 0x04 (**program1.s**).

To modify the CPU BPU, open the configuration file of the CPU (i.e., the *riscv_o3_custom.py):*

```
230     # *************** ************
231     # -- BPU SELECTION
232     # ****************************
233     # predictors from src/cpu/pred/BranchPredictor.py
234     # see create_predictors for choosing a predictor
235     the_cpu.branchPred = predictor.create_LocalBP()
```

You can create/use a different branch predictors. They are defined in *create_predictor.py*
You are encouraged to change their internal values.

Collects statistics about the execution of your programs in the following table:

Laboratory	**Program**	**Clock Per Instructions**			
		In Order CPU	**OoO CPU BPU = A**	**OoO CPU BPU = B**	**OoO CPU BPU = C**
0x03	program1.s	2.3943	1.3588	1.3468	1.4376
0x04	program1.s	1.8809	1.8934	1.8934	1.8934

A = TournamentBP
B = BiModeBP
C = LocalBP with:
pred.localPredictorSize = 64
pred.localCtrBits = 2
pred.BTBEntries = 512