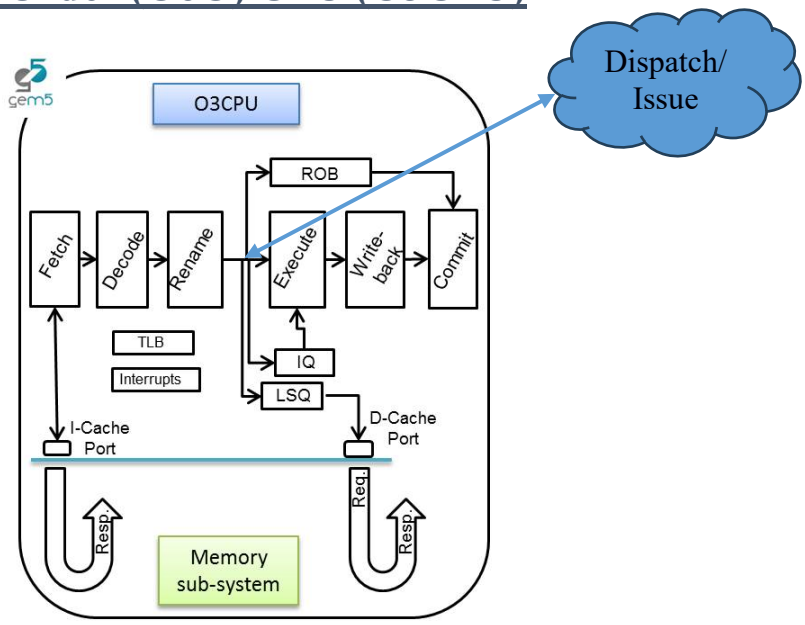
Architetture dei Sistemi di Elaborazione 02GOLOV	Computer Architectures 02LSEYG
Laboratory 0x05	Expected delivery of **lab_5.zip** must include: - each configuration of the custom architecture (riscv_o3_custom.py and create_predictor.py) that you modify. - This document with all the field compiled and in PDF form. Delivery date: November 13th 2025

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.

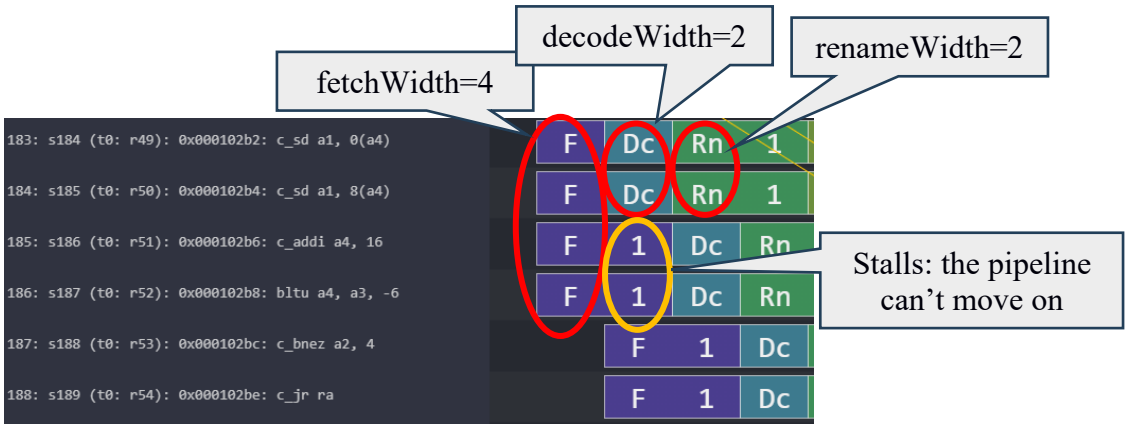


Figure 1: Understanding configurable OoO CPU parameters.

- **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

```
51  ##############################################
52  #########   PIPELINE VISUALIZER   #############
53  ##############################################
54  export PIPELINE_VISUALIZER="/mnt/d/uni/PoundtheHeadonDesk//konata-win32-x64/konata-win32-x64/konata.exe"
```

And change the path accordingly in your *setup_default* file (line 54).

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:

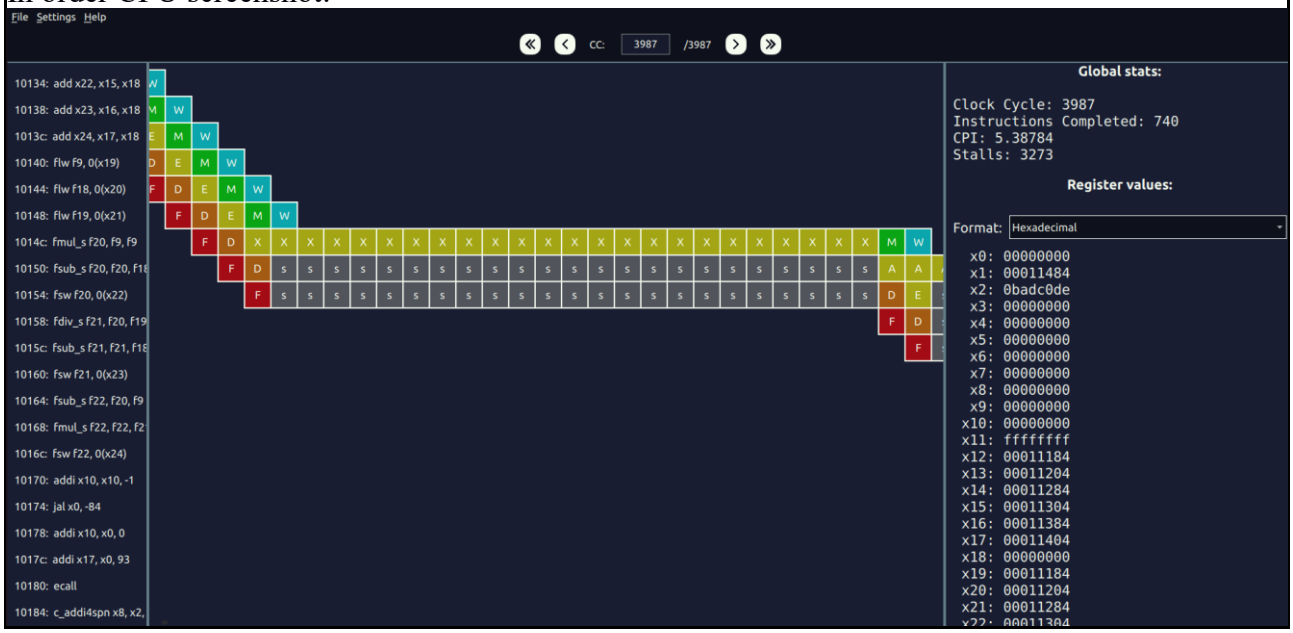
Simulate the programs written in the previous lab with the OoO CPU based on the CPU architecture described in `riscv_o3_custom.py`,
You can visualize the pipeline (i.e., load the `trace.out` file on Konata).

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

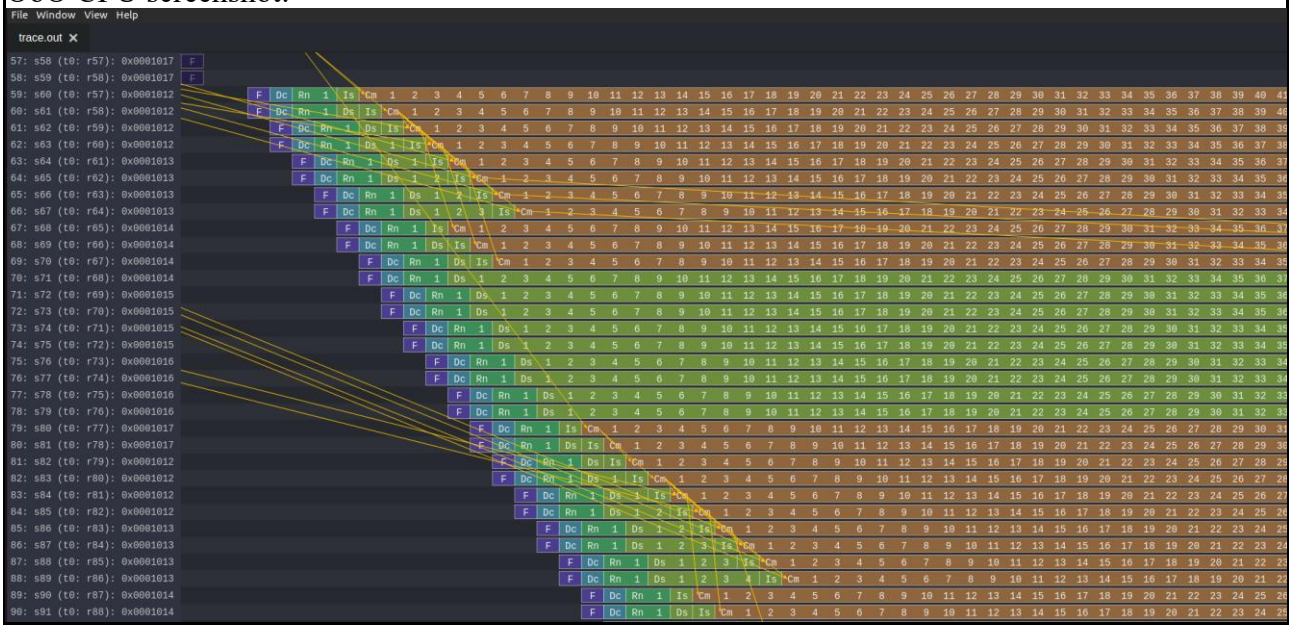
Laboratory	Program	Clock Per Instructions	
		In Order CPU	OoO CPU
0x01	program3.s	793	1282
0x02	program1.s	3987	1501
0x03	program1.s	2302	1776
	program1_a.s	2239	1784
	program1_b.s	1678	1913
0x04	program1.s	342	479
	program1_a.s	237	474
	program1_b.s	201	451

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

Situation: Lab 0x02, program_1.s: OoO più veloce di In Order
Explanation: In questo programma è presente un'elevata quantità di parallelismo a livello di istruzione all'interno del loop principale (più operazioni ALU e accessi in memoria non tutte mutualmente dipendenti). Sulla CPU In Order ogni dipendenza load-use e ogni branch costringe la pipeline a fermarsi finché il risultato non è disponibile, generando molte bolle e lasciando spesso inutilizzate le unità funzionali. La CPU OoO può invece rinominare i registri e riordinare dinamicamente le istruzioni indipendenti, emettendole mentre i load o i branch precedenti sono ancora in attesa, sovrapponendo anche iterazioni diverse del loop. In questo modo nasconde gran parte delle latenze di memoria e dei data hazard, riducendo i cicli totali (da 3987 a 1501) e risultando significativamente più veloce dell'architettura In Order per questo workload.
In order CPU screenshot:



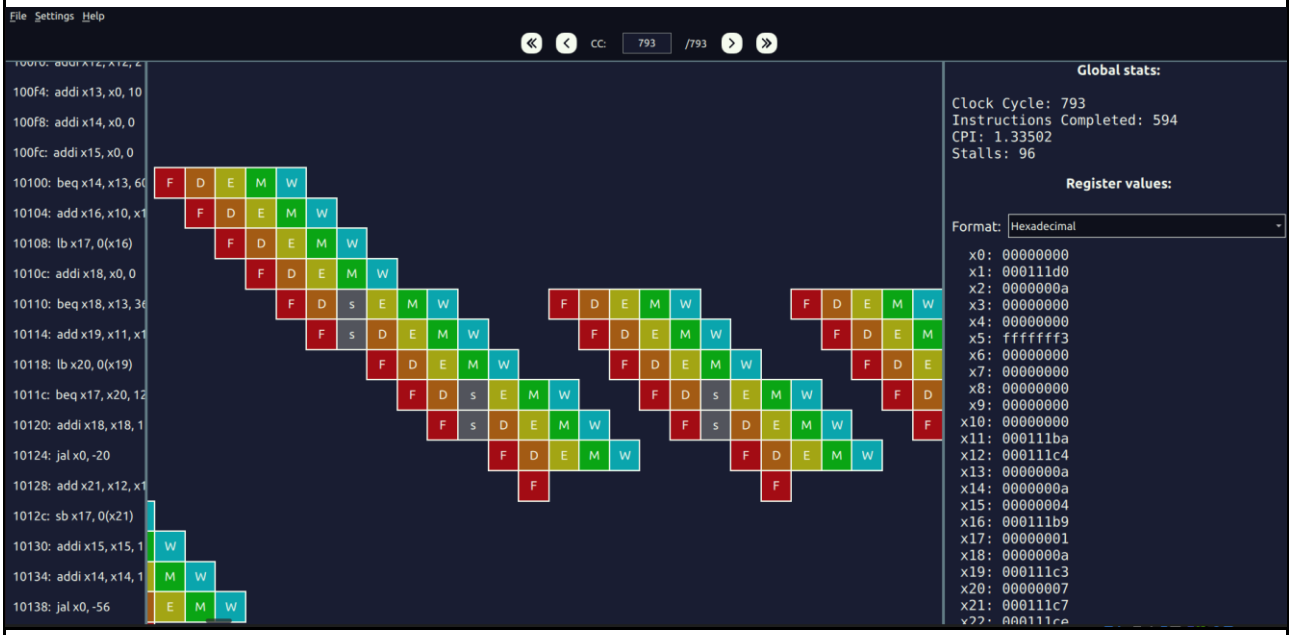
OoO CPU screenshot:



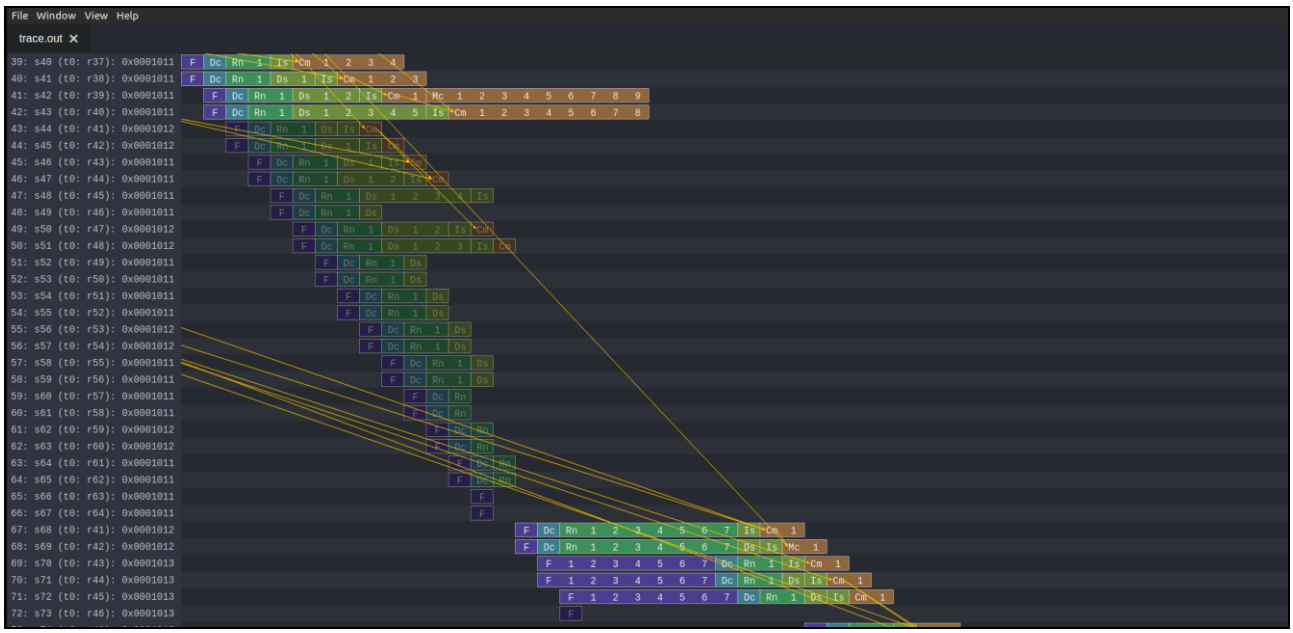
Situation: Lab 0x01, program 3.s: OoO ha prestazioni inferiori rispetto a In Order

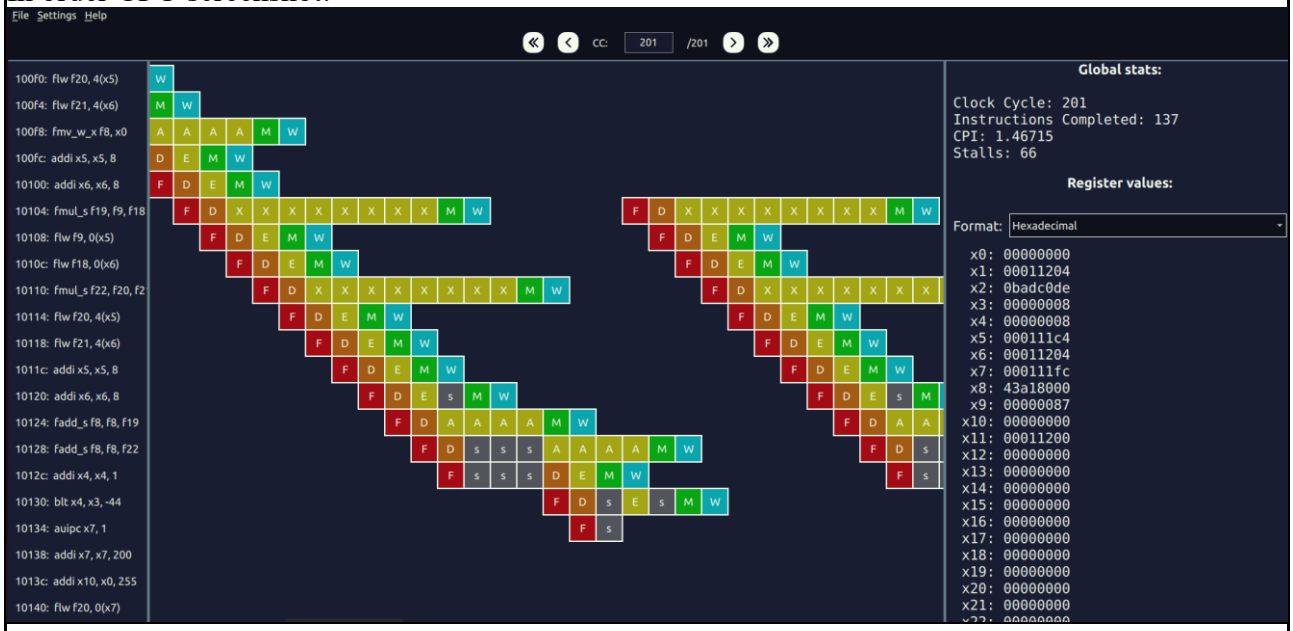
Explanation: In questo caso il programma è relativamente corto e offre poca ILP (poche istruzioni indipendenti eseguibili in parallelo). L'architettura OoO introduce stadi e strutture aggiuntive (rename, ROB, code di issue/LSQ) e una gestione più costosa dei salti: le eventuali mispredizioni comportano flush più lunghi e cicli per far avanzare ogni istruzione fino al commit. Poiché il core OoO non riesce a sfruttare parallelismo sufficiente per compensare questi overhead, il numero totale di cicli aumenta (1282 contro 793). Al contrario, la CPU In Order ha una pipeline più semplice e compatta, con pochi hazard effettivi da gestire, e riesce quindi a eseguire il programma quasi alla "velocità ideale", risultando più efficiente dell'architettura OoO per questo workload semplice e poco parallelo.

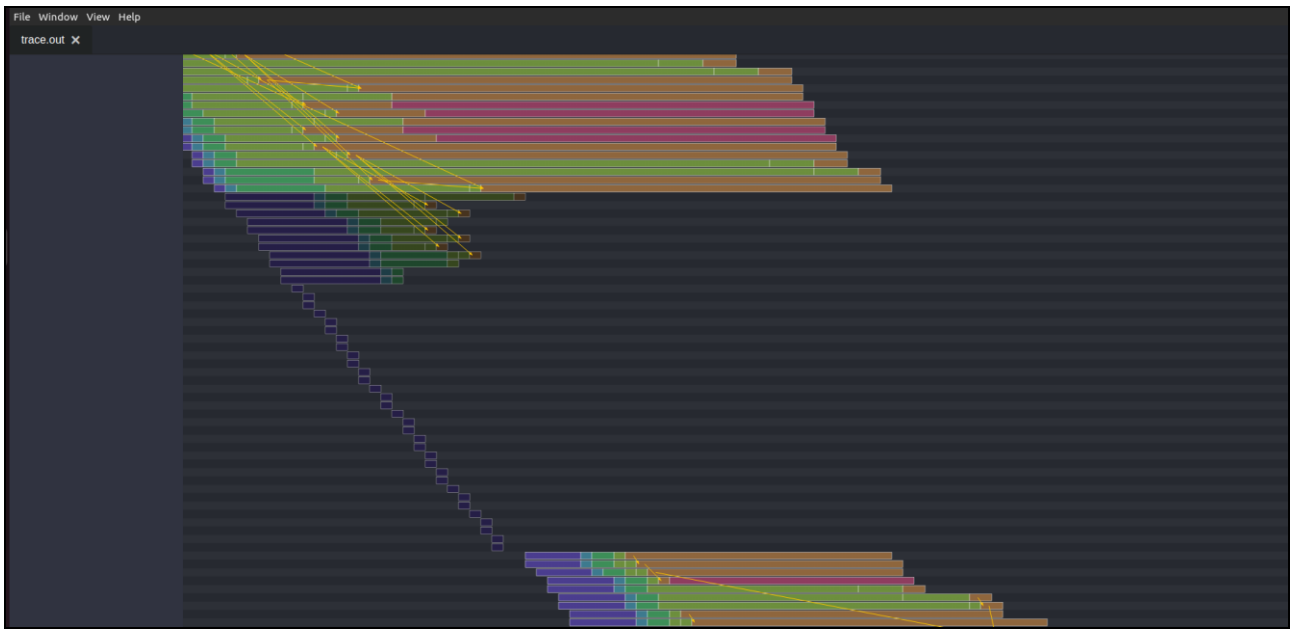
In order CPU screenshot:



OoO CPU screenshot:



Situation: Lab 0x04, program_1_b.s: OoO ha prestazioni inferiori rispetto a In Order
Explanation: In questo caso il sorgente program_1_b.s è stato esplicitamente ottimizzato per la pipeline In-Order del lab 0x04: le istruzioni sono rischedulate in modo da distanziare le dipendenze, riempire i “buchi” dopo i load e sfruttare al massimo l’issue statica (quasi nessuno stall, pipeline sempre piena). Di fatto il core In Order riesce a eseguire il loop vicino al throughput ideale, chiudendo il programma in pochi cicli (201). L’architettura OoO, invece, non trova praticamente ulteriore ILP oltre a quello già esposto dal codice ottimizzato, ma paga comunque il costo di rename, ROB, code di issue/LSQ e della speculazione sui salti: i flush dopo i branch e il commit in ordine introducono cicli extra che non vengono compensati da maggiore parallelismo. Il risultato è che la CPU OoO richiede molti più cicli (451 contro 201) e quindi performa peggio della In Order su questo workload già “cucito addosso” alla pipeline semplice.
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	2302	1776	1779	1753
0x04	program1.s	342	479	584	590