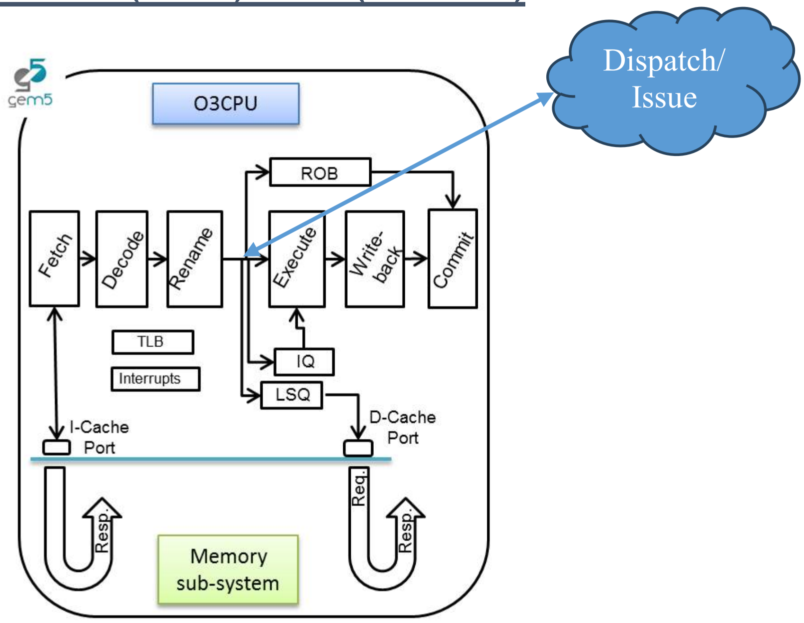
Architetture dei Sistemi di Elaborazione **O2GOLOV**	Delivery date: **16th November 2023**
Laboratory **4**	Expected delivery of **lab_4.zip** must include: • each configuration of the custom architecture (riscv_o3_custom.py) that you modify. • This document with all the field compiled and in PDF form.

Introduction and Background

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

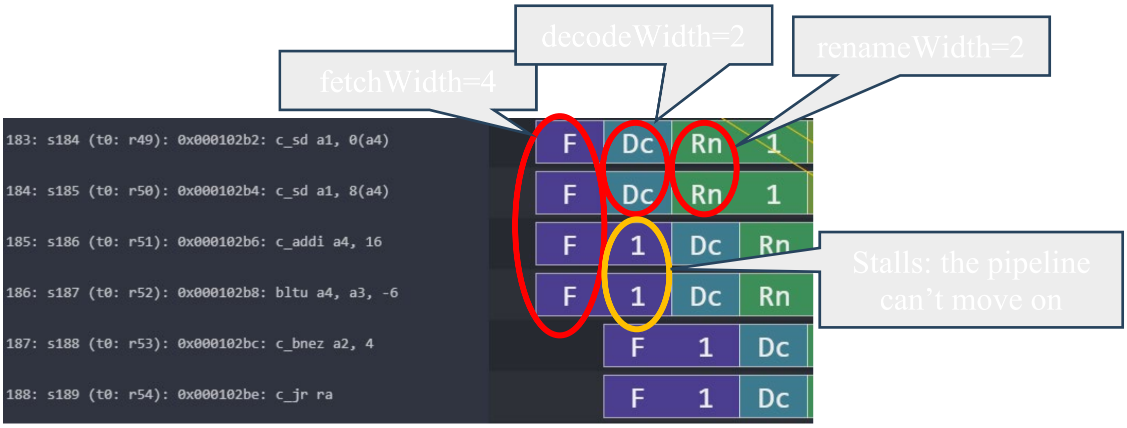


In this laboratory, you will be able to configure an OoO CPU by using a script called riscv_o3_custom.py. In a few words, the script configures an Out-of-Order (O3) processor based on the *DerivO3CPU,* a superscalar processor with a reduced number of features.

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.

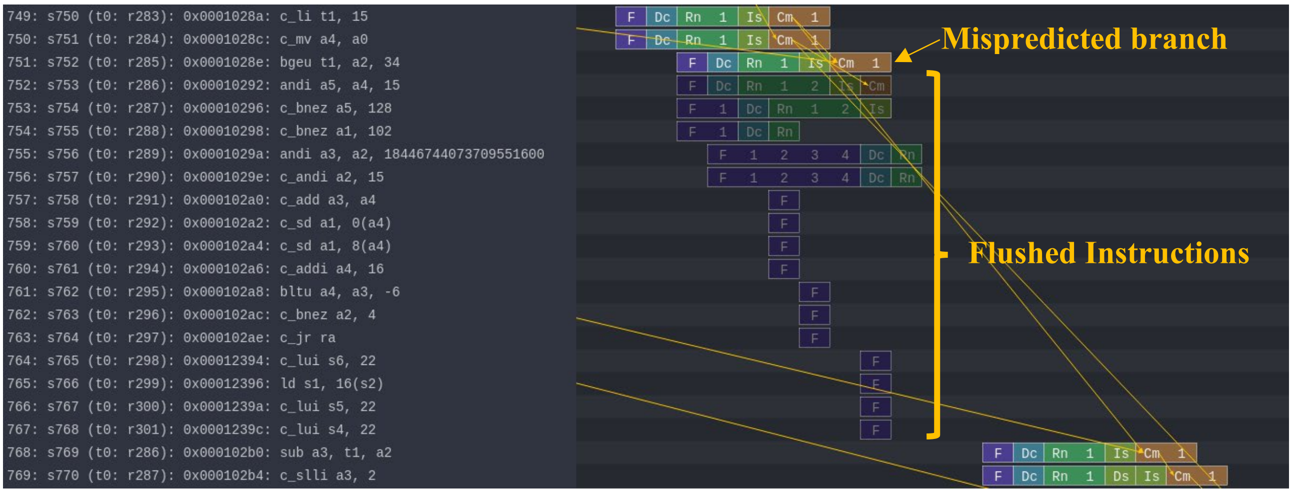


```
183: s184 (t0: r49): 0x000102b2: c_sd a1, 0(a4)
184: s185 (t0: r50): 0x000102b4: c_sd a1, 8(a4)
185: s186 (t0: r51): 0x000102b6: c_addi a4, 16
186: s187 (t0: r52): 0x000102b8: bltu a4, a3, -6
187: s188 (t0: r53): 0x000102bc: c_bnez a2, 4
188: s189 (t0: r54): 0x000102be: c_jr ra
```

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.



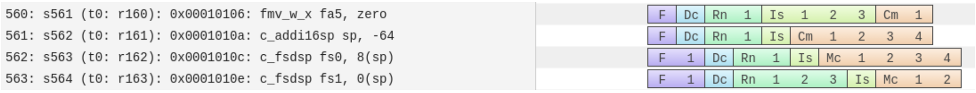
```
749: s750 (t0: r283): 0x0001028a: c_li t1, 15
750: s751 (t0: r284): 0x0001028c: c_mv a4, a0
751: s752 (t0: r285): 0x0001028e: bgeu t1, a2, 34
752: s753 (t0: r286): 0x00010292: andi a5, a4, 15
753: s754 (t0: r287): 0x00010296: c_bnez a5, 128
754: s755 (t0: r288): 0x00010298: c_bnez a1, 102
755: s756 (t0: r289): 0x0001029a: andi a3, a2, 18446744073709551600
756: s757 (t0: r290): 0x0001029e: c_andi a2, 15
757: s758 (t0: r291): 0x000102a0: c_add a3, a4
758: s759 (t0: r292): 0x000102a2: c_sd a1, 0(a4)
759: s760 (t0: r293): 0x000102a4: c_sd a1, 8(a4)
760: s761 (t0: r294): 0x000102a6: c_addi a4, 16
761: s762 (t0: r295): 0x000102a8: bltu a4, a3, -6
762: s763 (t0: r296): 0x000102ac: c_bnez a2, 4
763: s764 (t0: r297): 0x000102ae: c_jr ra
764: s765 (t0: r298): 0x00012394: c_lui s6, 22
765: s766 (t0: r299): 0x00012396: ld s1, 16(s2)
766: s767 (t0: r300): 0x0001239a: c_lui s5, 22
767: s768 (t0: r301): 0x0001239c: c_lui s4, 22
768: s769 (t0: r286): 0x000102b0: sub a3, t1, a2
769: s770 (t0: r287): 0x000102b4: c_slli a3, 2
```

*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.



```
560: s561 (t0: r160): 0x00010106: fmv_w_x fa5, zero      F Dc Rn 1 Is 1 2 3 Cm 1
561: s562 (t0: r161): 0x0001010a: c_addi16sp sp, -64     F Dc Rn 1 Is Cm 1 2 3 4
562: s563 (t0: r162): 0x0001010c: c_fsdsp fs0, 8(sp)     F 1 Dc Rn 1 Is Mc 1 2 3 4
563: s564 (t0: r163): 0x0001010e: c_fsdsp fs1, 0(sp)     F 1 Dc Rn 1 2 3 Is Mc 1 2
```

Figure 3: Pipeline example of FP instructions and FP registers

Laboratory: hands-on

All the needed resources are at a GitHub repository:

https://github.com/cad-polito-it/ase_riscv_gem5_sim

To create your simulation environment:
For HTTPS clone:

```
~/my_gem5Dir$ git clone https://github.com/cad-polito-it/ase_riscv_gem5_sim.git
```

For SSH:

```
~/my_gem5Dir$ git clone git@github.com:cad-polito-it/ase_riscv_gem5_sim.git
```

The environment is configured to be executed on the LABINF MACHINES.

Follow the HOWTO instructions available on the GitHub Repository for simulating a program.

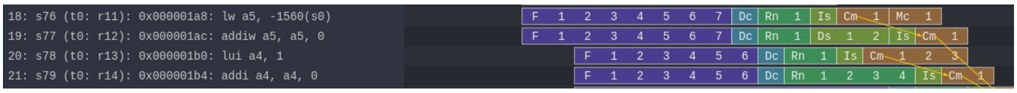
Exercise 1:

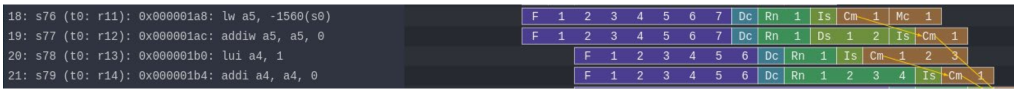
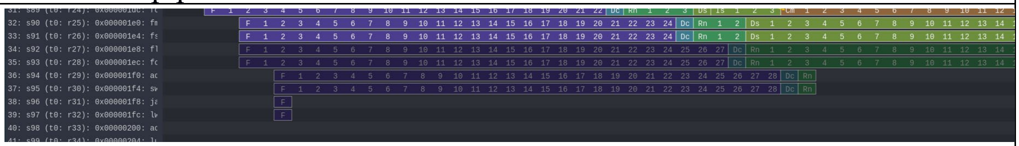
Simulate the benchmark `my_c_benchmark` (*main.c*) by using the gem5 simulator to obtain the `trace.out` file. Then, you can visualize the pipeline (i.e., load the `trace.out` file on Konata).

Based on the CPU architecture described in `riscv_o3_custom.py`, visualize the Konata's pipeline to find out the conditions:

1. Out-of-order execution (issue), in-order commit (commit)
2. Two commits in the same clock cycle
3. Flush of the pipeline.

For every condition, fill the following tables.

Condition	Out-of-order execution, in-order commit
Screenshot from Konata	 18: s76 (t0: r11): 0x000001a8: lw a5, -1560(s0) F 1 2 3 4 5 6 7 Dc Rn 1 Is Cm 1 Mc 1 19: s77 (t0: r12): 0x000001ac: addiw a5, a5, 0 F 1 2 3 4 5 6 7 Dc Rn 1 Ds 1 2 Is Cm 1 20: s78 (t0: r13): 0x000001b0: lui a4, 1 F 1 2 3 4 5 6 Dc Rn 1 Is Cm 1 2 3 21: s79 (t0: r14): 0x000001b4: addi a4, a4, 0 F 1 2 3 4 5 6 Dc Rn 1 2 3 4 Is Cm 1
Explain the reason behind the condition	In this case the lui instruction in line 20 is OoO because the previous instruction, the addiw in line 77, has RAW hazard, so it is executed, but it waits for commit.
Briefly explain the advantages of the OoO execution in a CPU	OoO execution is important because the instruction can be excuted despite the previous instructions are stalled because of any hazard. This improves our program and reduces number o CPI.
Condition	Two or more commits in the same clock cycle

Screenshot from Konata	 18: s76 (t0: r11): 0x000001a8: lw a5, -1560(s0) 19: s77 (t0: r12): 0x000001ac: addiw a5, a5, 0 20: s78 (t0: r13): 0x000001b0: lui a4, 1 21: s79 (t0: r14): 0x000001b4: addi a4, a4, 0
Explain the reason behind the condition	The processor executes instructions OoO and it's possible for some instructions to be completed ahead of others. However, the "commit" process must ensure that the results are made visible in the correct order. So, even if an operation finishes earlier, it must wait to be committed.
Briefly explain the Commit functioning	"Commit" is the process where completed instructions become final and their results are made visible to the outside world in the correct order.
Condition	Flush of the pipeline
Screenshot from Konata	 31: s89 (t0: r24): 0x000001dc: fc 32: s90 (t0: r25): 0x000001e0: fm 33: s91 (t0: r26): 0x000001e4: fs 34: s92 (t0: r27): 0x000001e8: fl 35: s93 (t0: r28): 0x000001ec: fc 36: s94 (t0: r29): 0x000001f0: ac 37: s95 (t0: r30): 0x000001f4: sw 38: s96 (t0: r31): 0x000001f8: ja 39: s97 (t0: r32): 0x000001fc: lw 40: s98 (t0: r33): 0x00000200: ac 41: s99 (t0: r34): 0x00000204: lw
Explain the reason behind the condition	When a pipeline flush due to a branch misprediction, the goal is to eliminate all instructions in the pipeline that follow the incorrect branch prediction. When a branch prediction is incorrect, the pipeline is "flushed" to refill it with the correct instructions from the actual execution path, ensuring that execution continues from the correct position.

Exercise 2:

Given your benchmark (`main.c` in `my_c_benchmark`), optimize the CPU architecture (i.e., modify the `riscv_o3_custom.py` file) and write down the improvements in terms of CPI and speedup.

- To optimize the CPU architecture, open the configuration file of the CPU (i.e., the `riscv_o3_custom.py),` and tune specific hardware-related parameters.

  You have to change specific values in **one or more** stages of the pipeline:

  - # - FETCH STAGE
    - Tune parameters such as the `fetchWidht`, `fetchBuffersize` and so on, and see the effects on your system.
  - # - DECODE STAGE
  - # - RENAME STAGE
    - Try changing some values, but don't touch the "Phys" ones.
  - # - DISPATCH/ISSUE STAGE
  - # - EXECUTE STAGE

    - Here you can optimize the Functional units of your CPU like the INT ALU, the FP ALU, the FP Multiplier/Divider and so on.
    - Tune the number of units (`count`) that you have in the system, as well as their latency (`opLat`) to see how this affects the execution of your program.

- You can create a different branch predictor. They are defined in `create_predictor.py)`

- You can also try to change the parameters of the L1 Cache. Look for the "class L1Cache" in the `riscv_o3_custom.py` file. The L1 cache, also referred to as the primary cache, is the smallest and fastest level of memory. It is located directly on the processor, and it is used to store frequently accessed data by the CPU. In this way, the CPU saves time with respect to the normal access to the main memory.

> **HINT:** To implement the best hardware optimization, and understand how to change the parameters, the best option consists in analysing the *stats.txt* file (in **ase_riscv_gem5_sim/results/my_c_benchmark**).
> Find information regarding the workload profiling. In other words, look for lines such as "system.cpu.commitStats0.committedInstType::**IntAlu**", and the following ones to understand which kind of instructions are executed the most. In this way, you can target a specific functional unit and modify its specifications.

Fill the following Tables with the CPI that you obtain with the old and the new architectures. Compute also the equivalent speedup that you obtain.
HINT: You can get the CPI and other useful information from the `stats.txt` file.

Parameters	**Configuration 1**	**Configuration 2**	**Configuration 3**	**Configuration 4**
The_cpu.fetchWidth	12	8	none	12
The_cpu.fetchBufferSize	none	32	none	16
The_cpu.fetchQueueSize	none	64	none	256
The_cpu.decodeWidth	8	8	none	12
The_cpu.renameWidth	none	4	none	12
The_cpu.dispatchWidth	none	8	none	12
The_cpu.issueWidth	none	none	none	12
The_cpu.CPU_IntALU	6	6	6	6
The_cpu.numIQEntries	none	none	32	64
CPU_FP_ALU FloatAdd optLAt	none	none	none	1
CPU_FP_ALU FloatCvt optLAt	none	none	none	1
CPU_FP_MultDiv FloatMult optLat	none	none	none	1
CPU_FP_MultDiv FloatDiv optLat	none	none	none	1

Original CPI (no hardware optimization): 2.08310

	Configuration 2	**Configuration 2**	**Configuration 3**	**Configuration 4**
CPI	1.983529	1.946718	0.945976	0.859995
Speedup (wrt Original CPI)	1.0625	1.0625	2.125	2.428

Which is the best optimization in terms of CPI and speedup, why?

Your answer:

By exclusively modifying the parameters of fetch, decode, and adding three units of intALU, a significant improvement is not observed, and this could be attributed to various factors. It is possible that a bottleneck occurs somewhere in the system, causing the stall of numerous instructions. Additionally, the presence of data dependencies, known as data hazards, could lead to persistent stalls. Even by enhancing other parameters such as fetchbuffersize, fetchQueueSize, renamewidth, and dispatchwidth, substantial progress does not emerge, most likely due to the same previously mentioned issues. However, by intervening in the number of instructions in the processor's input queue (The_cpu.numIQEntries), a significant improvement is observed. This result could be attributed to the fact that the increase in the number of instructions in the queue allows the processor to achieve a higher degree of parallelism, contributing to a drastic reduction in the number of cycles per instruction (CPI). Moreover, a larger instruction queue can decrease the probability of stalls due to data dependencies or situations where the processor has to wait for the arrival of new instructions. Additionally, with a larger instruction queue, the processor gains greater flexibility in scheduling and executing instructions, adapting to resource availability and reducing the risk of underutilization of execution units or other processor resources.

The optimal configuration is achieved by improving all parameters of the three mentioned configurations and reducing the latency times of the ALUs that are used more frequently, resulting in a speedup of 2.428.