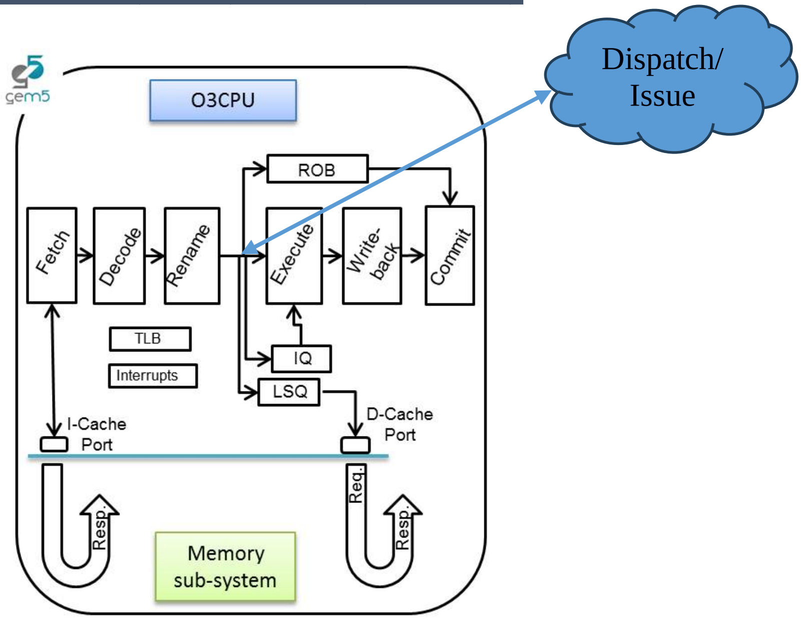
Architetture dei Sistemi di Elaborazione O2GOLOV	Delivery date: **October 30th 2024**
Laboratory 4	Expected delivery of **lab_4.zip** must include: • each configuration of the custom architecture (riscv_o3_custom.py) that you modify. • This document with all the field compiled and in PDF form.

Introduction and Background

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

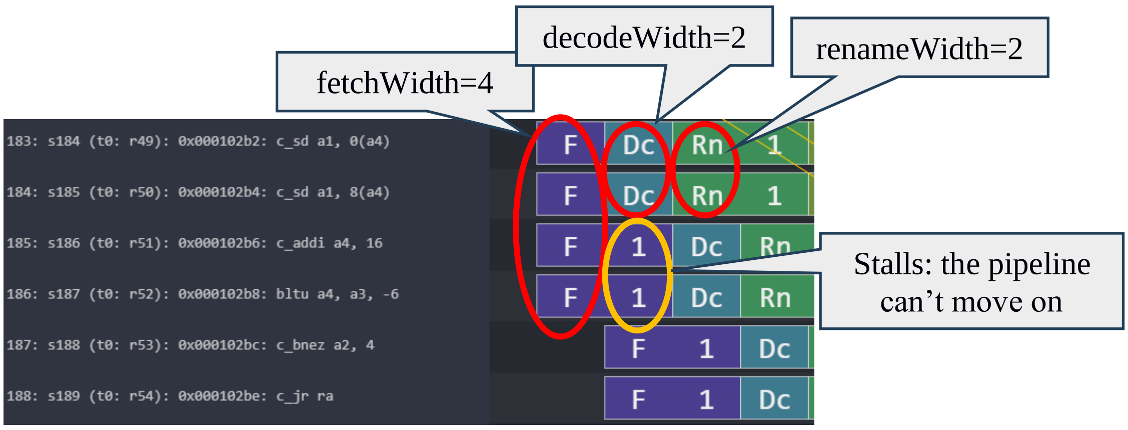


In this laboratory, you will be able to configure an OoO CPU by using a script called riscv_o3_custom.py. In a few words, the script configures an Out-of-Order (O3) processor based on the *DerivO3CPU*, a superscalar processor with a reduced number of features.

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.

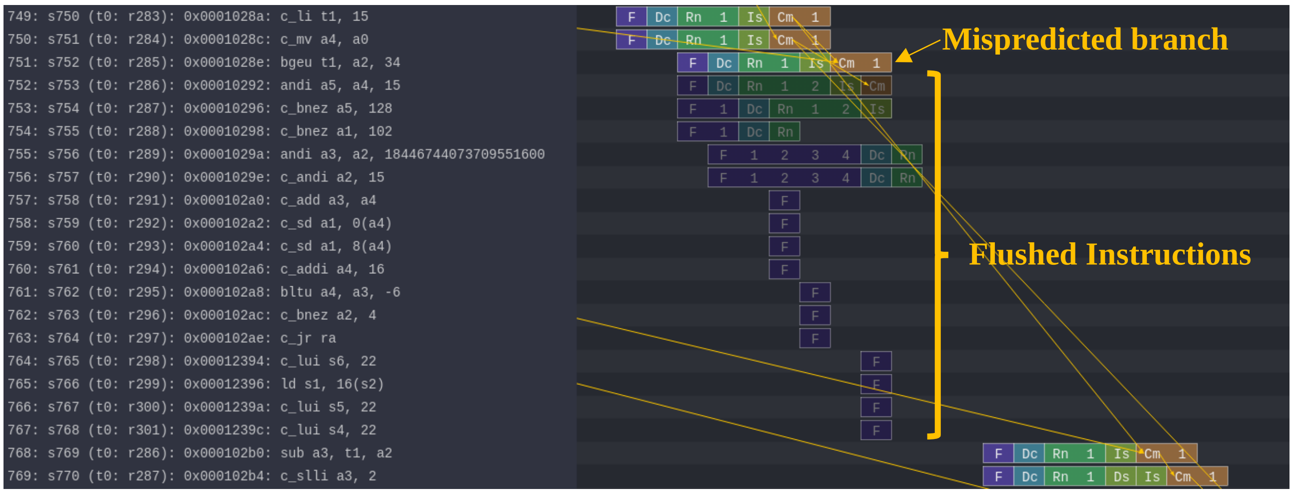


```
183: s184 (t0: r49): 0x000102b2: c_sd a1, 0(a4)
184: s185 (t0: r50): 0x000102b4: c_sd a1, 8(a4)
185: s186 (t0: r51): 0x000102b6: c_addi a4, 16
186: s187 (t0: r52): 0x000102b8: bltu a4, a3, -6
187: s188 (t0: r53): 0x000102bc: c_bnez a2, 4
188: s189 (t0: r54): 0x000102be: c_jr ra
```

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.



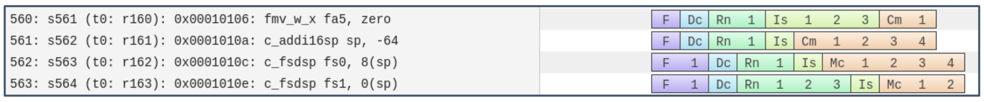
```
749: s750 (t0: r283): 0x0001028a: c_li t1, 15
750: s751 (t0: r284): 0x0001028c: c_mv a4, a0
751: s752 (t0: r285): 0x0001028e: bgeu t1, a2, 34
752: s753 (t0: r286): 0x00010292: andi a5, a4, 15
753: s754 (t0: r287): 0x00010296: c_bnez a5, 128
754: s755 (t0: r288): 0x00010298: c_bnez a1, 102
755: s756 (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.

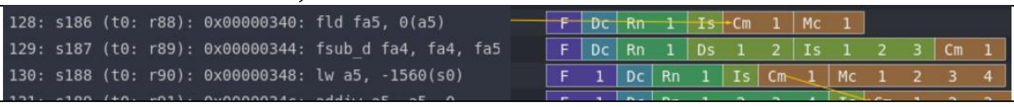
Exercise 1:

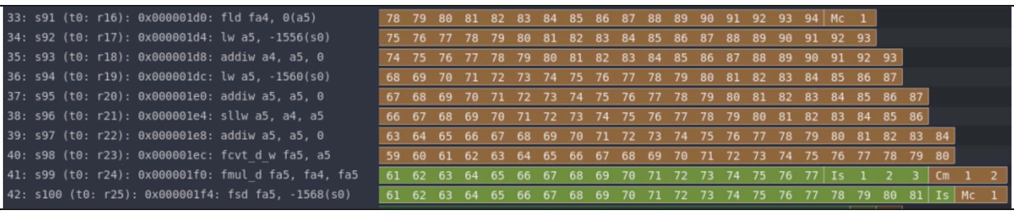
Simulate the benchmark `my_c_benchmark_2` (*main.c*) by using the gem5 simulator to obtain the `trace.out` file. Then, you can visualize the pipeline (i.e., load the `trace.out` file on Konata).

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

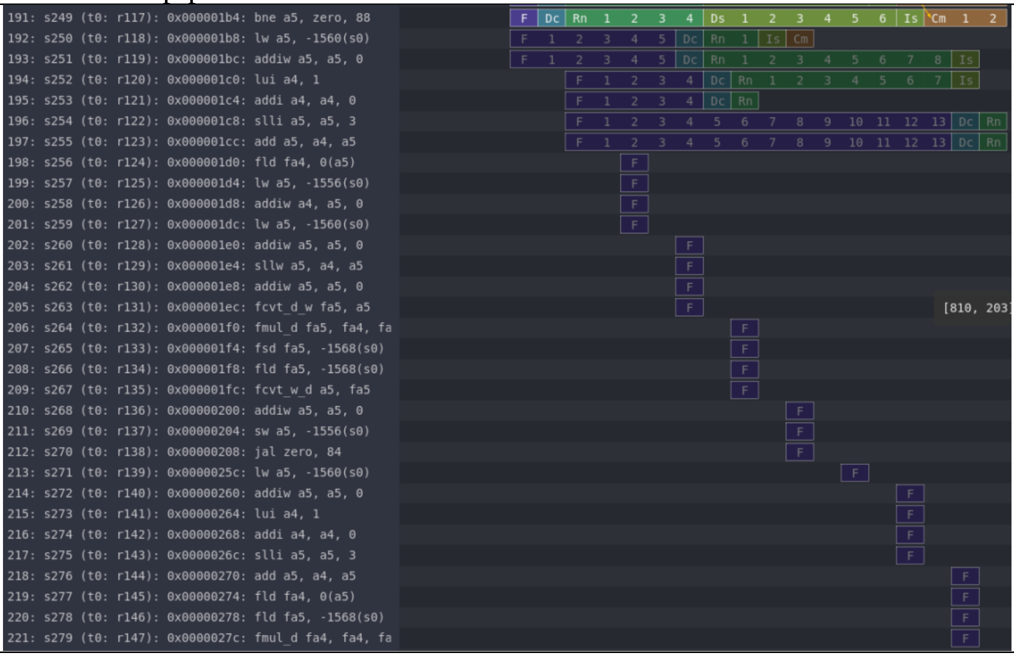
1. Out-of-order execution (issue), in-order commit (commit)
2. Two commits in the same clock cycle
3. Flush of the pipeline.

For every condition, fill the following tables.

Condition	Out-of-order execution, in-order commit
Screenshot from Konata	 128: s186 (t0: r88): 0x00000340: fld fa5, 0(a5) F Dc Rn 1 Is Cm 1 Mc 1 129: s187 (t0: r89): 0x00000344: fsub_d fa4, fa4, fa5 F Dc Rn 1 Ds 1 2 Is 1 2 3 Cm 1 130: s188 (t0: r90): 0x00000348: lw a5, -1560(s0) F 1 Dc Rn 1 Is Cm 1 Mc 1 2 3 4
Explain the reason behind the condition	In this scenario, the instruction at line 130 executes out-of-order because the previous instruction at line 129 is stalled, waiting for the value in register "fa5". Although line 130 is ready to execute and does so out of order, both instructions (line 129 and line 130) will commit in program order. This is because the instruction at line 130 cannot commit before line 129, ensuring that the program's sequential integrity is preserved despite the out-of-order execution.
Briefly explain the advantages of the OoO execution in a CPU	• **Performance Optimization**: OoO execution allows the CPU to execute instructions that are ready, even if earlier instructions are waiting on resources. This maximizes resource usage and reduces idle time, leading to higher performance. • **Reduced Pipeline Stalls**: Since instructions can be processed as soon as resources are available, OoO execution minimizes pipeline stalls, keeping the pipeline running smoothly and enhancing instruction throughput.
Condition	Two or more commits in the same clock cycle

Screenshot from Konata	

```
33: s91 (t0: r16): 0x000001d0: fld fa4, 0(a5)          78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 Mc 1
34: s92 (t0: r17): 0x000001d4: lw a5, -1556(s0)        75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93
35: s93 (t0: r18): 0x000001d8: addiw a4, a5, 0         74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93
36: s94 (t0: r19): 0x000001dc: lw a5, -1560(s0)        68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87
37: s95 (t0: r20): 0x000001e0: addiw a5, a5, 0         67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87
38: s96 (t0: r21): 0x000001e4: sllw a5, a4, a5         66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86
39: s97 (t0: r22): 0x000001e8: addiw a5, a5, 0         63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84
40: s98 (t0: r23): 0x000001ec: fcvt_d_w fa5, a5        59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80
41: s99 (t0: r24): 0x000001f0: fmul_d fa5, fa4, fa5    61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 Is 1 2 3 Cm 1 2
42: s100 (t0: r25): 0x000001f4: fsd fa5, -1568(s0)     61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 Is Mc 1
```

Explain the reason behind the condition	In this simulation, the commitWidth is set to 2, meaning the processor can commit a maximum of two instructions per clock cycle. This setting ensures that while multiple instructions can be finalized in one cycle, it is not feasible to see more than two commits within the same cycle. Consequently, seeing two commits frequently is expected and aligns with the pipeline's designed throughput. This behavior highlights the efficiency of Tomasulo's algorithm with speculation in managing dependencies. With speculation, the processor can predict branches and execute instructions out of order, effectively bypassing output dependencies (write-after-write hazards) by using register renaming. This mechanism prevents one instruction's output from blocking subsequent instructions, enabling higher throughput and efficient resource use in the pipeline, especially in the presence of speculative execution and branch prediction.
Briefly explain the Commit functioning	The commit stage in a CPU pipeline is responsible for finalizing the results of instructions. Although instructions may be executed out of order, their results are finalized in program order. This maintains the appearance of sequential execution for the program, ensuring that the CPU's state reflects the correct sequence of operations.
Condition	Flush of the pipeline
Screenshot from Konata	

```
191: s249 (t0: r117): 0x000001b4: bne a5, zero, 88          F Dc Rn 1 2 3 4 Ds 1 2 3 4 5 6 Is Cm 1 2
192: s250 (t0: r118): 0x000001b8: lw a5, -1560(s0)          F 1 2 3 4 5 Dc Rn 1 Is Cm
193: s251 (t0: r119): 0x000001bc: addiw a5, a5, 0           F 1 2 3 4 5 Dc Rn 1 2 3 4 5 6 7 8 Is
194: s252 (t0: r120): 0x000001c0: lui a4, 1                 F 1 2 3 4 Dc Rn 1 2 3 4 5 6 7 Is
195: s253 (t0: r121): 0x000001c4: addi a4, a4, 0            F 1 2 3 4 Dc Rn
196: s254 (t0: r122): 0x000001c8: slli a5, a5, 3            F 1 2 3 4 5 6 7 8 9 10 11 12 13 Dc Rn
197: s255 (t0: r123): 0x000001cc: add a5, a4, a5            F 1 2 3 4 5 6 7 8 9 10 11 12 13 Dc Rn
198: s256 (t0: r124): 0x000001d0: fld fa4, 0(a5)            F
199: s257 (t0: r125): 0x000001d4: lw a5, -1556(s0)          F
200: s258 (t0: r126): 0x000001d8: addiw a4, a5, 0           F
201: s259 (t0: r127): 0x000001dc: lw a5, -1560(s0)          F
202: s260 (t0: r128): 0x000001e0: addiw a5, a5, 0           F
203: s261 (t0: r129): 0x000001e4: sllw a5, a4, a5           F
204: s262 (t0: r130): 0x000001e8: addiw a5, a5, 0           F
205: s263 (t0: r131): 0x000001ec: fcvt_d_w fa5, a5          F                 [810, 203]
206: s264 (t0: r132): 0x000001f0: fmul_d fa5, fa4, fa       F
207: s265 (t0: r133): 0x000001f4: fsd fa5, -1568(s0)        F
208: s266 (t0: r134): 0x000001f8: fld fa5, -1568(s0)        F
209: s267 (t0: r135): 0x000001fc: fcvt_w_d a5, fa5          F
210: s268 (t0: r136): 0x00000200: addiw a5, a5, 0           F
211: s269 (t0: r137): 0x00000204: sw a5, -1556(s0)          F
212: s270 (t0: r138): 0x00000208: jal zero, 84              F
213: s271 (t0: r139): 0x0000025c: lw a5, -1560(s0)          F
214: s272 (t0: r140): 0x00000260: addiw a5, a5, 0           F
215: s273 (t0: r141): 0x00000264: lui a4, 1                 F
216: s274 (t0: r142): 0x00000268: addi a4, a4, 0            F
217: s275 (t0: r143): 0x0000026c: slli a5, a5, 3            F
218: s276 (t0: r144): 0x00000270: add a5, a4, a5            F
219: s277 (t0: r145): 0x00000274: fld fa4, 0(a5)            F
220: s278 (t0: r146): 0x00000278: fld fa5, -1568(s0)        F
221: s279 (t0: r147): 0x0000027c: fmul_d fa4, fa4, fa       F
```

Explain the reason behind the condition	In this architecture, branch prediction and branch target prediction are used to allow speculative execution of instructions following a branch. This speculative execution improves performance by keeping the pipeline busy, even when the outcome of a branch is not yet known. However, if a branch misprediction occurs, like in this case, (or if an exception or trap arises), the speculatively executed instructions that followed the branch are invalid. To correct this, the pipeline must be flushed, or "squashed," which involves discarding these instructions. This means they are removed from instruction queues, reorder buffers, and cache requests, and the pipeline restarts from the correct instruction path. This process ensures that only valid instructions are committed, maintaining program correctness despite speculation.

Exercise 2:

Given your benchmark (`main.c` in `my_c_benchmark_2`), 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, <u>but don't touch the "Phys" ones</u>.
  - # - 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.

<u>HINT:</u> 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_2**).
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

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**	**Configuration 5**
First changed paramenter	the_cpu.decodeWidth = 4	the_cpu.fetchWidth = 8	All conf 2 params	All conf 2 params	All conf 4 params
Second changed paramenter		the_cpu.fetchBufferSize = 32	l1i_size="64kB"	All FP opLat = 2	OpDesc(opClass="IntMult", opLat=3, pipelined=True),
Third changed paramenter		the_cpu.decodeWidth = 8	l1i_assoc=16		OpDesc(opClass="IntDiv", opLat=4, pipelined=True)
Fourth changed paramenter			l1d_size="64kB"		
Fifth changed paramenter			l1d_assoc=16		

Original CPI (no hardware optimization):

	Configuration 1	**Configuration 2**	**Configuration 3**	**Configuration 4**	**Configuration 5**
CPI	2.159332	2.102977	2.102977	2.077996	2.062745
Speedup (wrt Original CPI)	2.20 / 2.159332 = 1.01883	2.20 / 2.102977 = 1.04614	2.20 / 2.102977 = 1.04614	2.20 / 2.077996 = 1.05871	2.20 / 2.062745 = 1.06654

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

Your answer:

The best optimization in terms of CPI and speedup is indeed the fifth, as it builds upon all previous optimizations, resulting in the most comprehensive improvement. By incrementally applying each optimization, the fifth configuration achieves the highest overall performance, benefiting from cumulative enhancements across various components of the pipeline.

However, when considering each optimization individually, the fourth optimization stands out as the most effective. This is because it specifically targets the floating-point (FP) unit, which plays a significant role in the benchmark program. Optimizing the FP unit leads to a notable improvement in CPI and speedup, as the benchmark relies heavily on floating-point operations.

On the other hand, the third optimization, which involves increasing cache size, shows no speedup over the second optimization. This is likely because the benchmark program doesn’t experience significant cache misses or does not benefit from a larger cache, making this change less impactful for optimization.