Introduction and Background

Simulating an Out-of-Order (OoO) CPU (O3CPU)
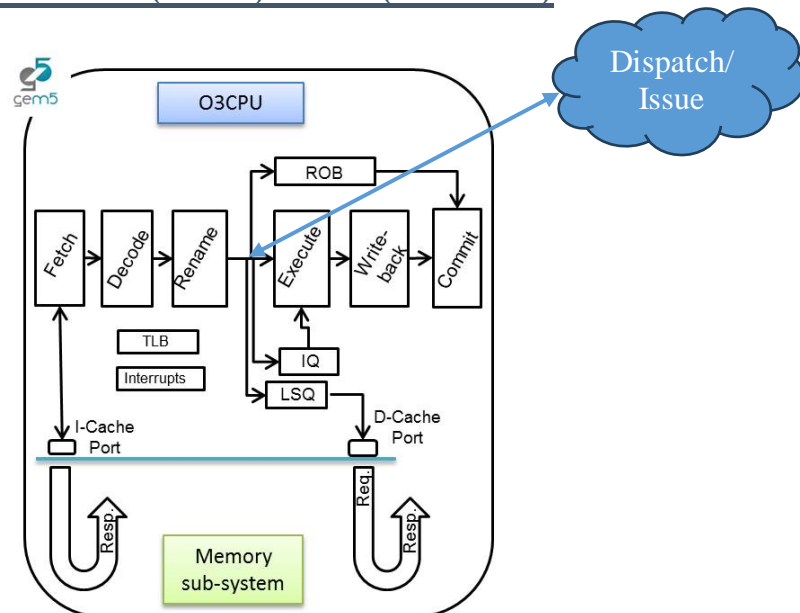
In this laboratory, you will be able to configure an OoO CPU by using a script called riscv_o3_custom.py. In a few words, the script configures an Out-of-Order (O3) processor based on the *DerivO3CPU,* a superscalar processor with a reduced number of features.

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.
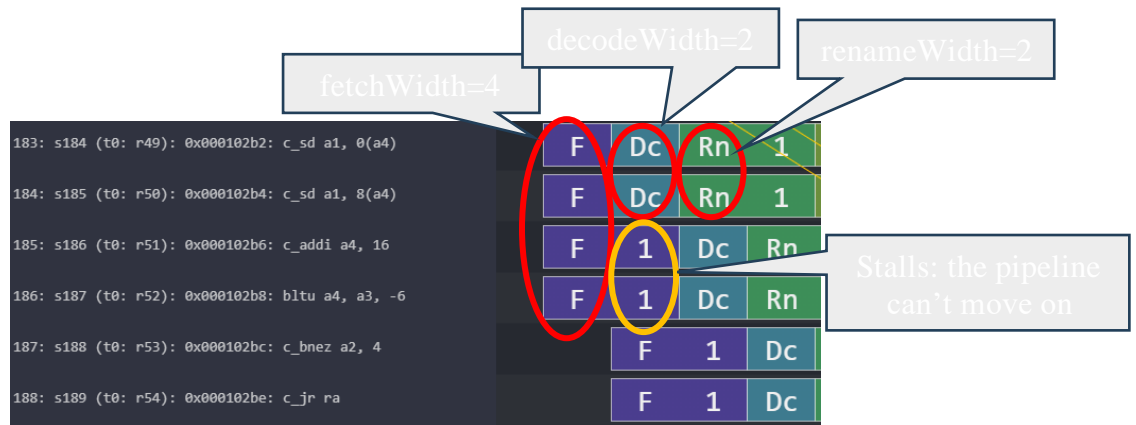
Figure 1: Understanding configurable OoO CPU parameters.

- **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.
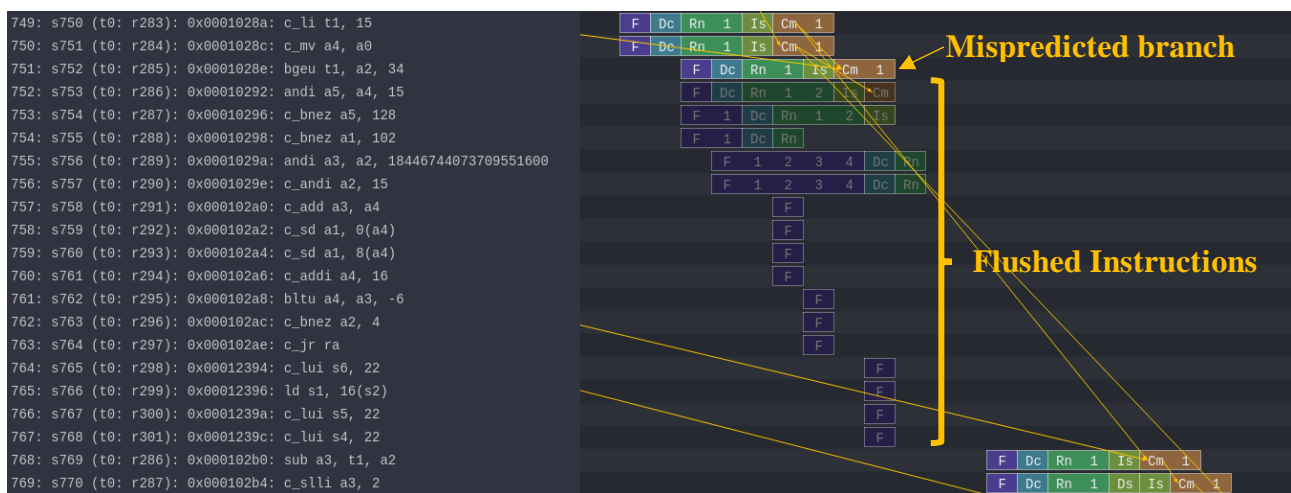

*Figure 2: Example of a branch **misprediction** (transparent rows)*

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.


```
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

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.

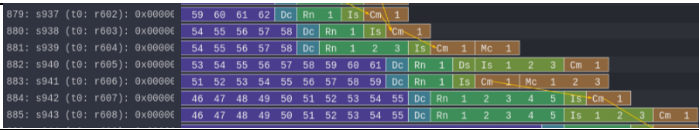Follow the HOWTO instructions available on the GitHub Repository for simulating a program.

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	
Explain the reason behind the condition	Il ReOrder Buffer e l'architettura intorno permettono di eseguire le operazioni anche in modo non sequenziale se queste non sono dipendenti l'una con l'altra. In questo caso, quindi, la seconda operazione era pronta all'esecuzione precedentemente alla prima ed ha dunque potuto iniziare l'esecuzione senza stallare, mentre per evitare hazard il commit avviene sempre in ordine sequenziale all'ingresso dell'istruzione del ROB
Briefly explain the advantages of the OoO execution in a CPU	Questa dinamica è utilissima nei superscalari, poiché permette una riorganizzazione del codice rendendo quindi il programma molto più efficiente. Ad esempio, in questo caso è stato possibile iniziare l'esecuzione di un'istruzione successiva già pronta che altrimenti avrebbe dovuto stallare, e con lei anche le sue successive.

Condition	Two or more commits in the same clock cycle
Screenshot from Konata	
Explain the reason behind the condition	In questo caso l'istruzione successiva aveva terminato l'esecuzione prima della precedente, ma non è permesso che esegua anche il commit prima della precedente.
Briefly explain the Commit functioning	Il commit ha la funzione di confermare in memoria i cambiamenti avvenuti durante l'istruzione. Se però un'istruzione avesse il permesso di fare non solo l'execute ma anche il commit prima di un'altra, si potrebbero verificare degli hazard di tipo RAW e WAW che quindi comprometterebbero l'intera esecuzione del programma. Per evitare ciò, è permessa l'esecuzione fuori ordine ma è obbligatorio che il commit avvenga in ordine
Condition	Flush of the pipeline
Screenshot from Konata	
Explain the reason behind the condition	Questo tipo di comportamento avviene quando la predizione al salto è sbagliata; in questo caso il compilatore stava eseguendo istruzioni che non dovevano essere eseguite, indi per cui è costretto a svuotare interamente la pipeline ed a ricominciare da capo.

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.

- o 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:

  - o # - FETCH STAGE

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

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

- o 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 4	Configuration 5
First changed paramenter	the_cpu.fetch Width = 8	the_cpu.fetchWi dth = 2	tag_latency = 1	tag_latency = 1
Second changed paramenter	the_cpu.fetchB ufferSize = 16		data_latency = 1	tgts_per_mshr = 5
Third changed paramenter	the_cpu.fetchQ ueueSize = 32		response_latenc y = 1	
	the_cpu.decod eWidth = 4			

Original CPI (no hardware optimization): 2.083

	Configuration 1	Configuration 2	Configuration 4	Configuration 5
CPI	2.081	2.045	1.905	2.038
Speedup (wrt Original CPI)	1.001	1.019	1.094	1.022

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

La miglior ottimizzazione in termini di CPI è quella della configurazione 4. Dalla CPI infatti si vorrebbe il più basso valore possibile, che in questo caso è dato da 1.905. Ovviamente, visto che lo speedup di ognuno di questi programmi si confronta con lo stesso, quello base, quello che avrà alla fine una resa maggiore sarà anche quello ad essere migliorato di più; non sorprende che sia sempre la configurazione 4 ad aver avuto lo speedup maggiore.