

- **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.

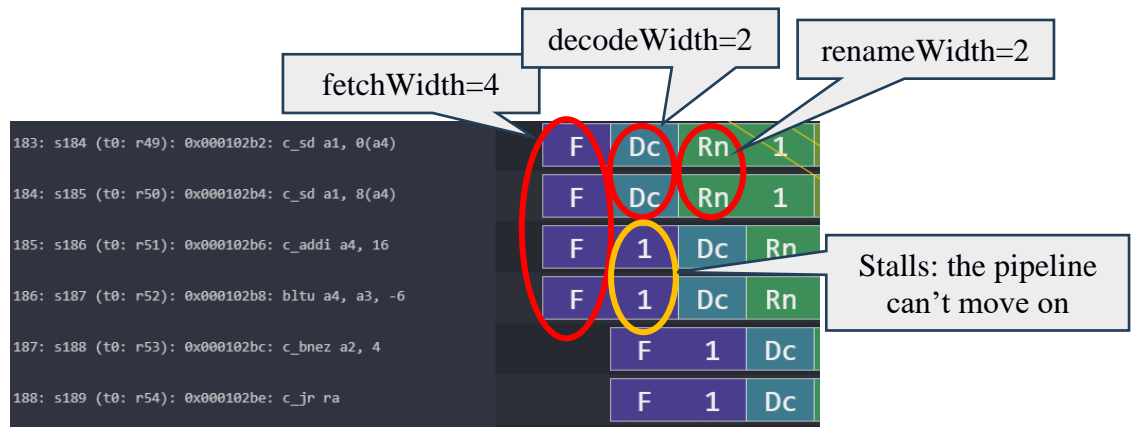


Figure 1: Understanding configurable OoO CPU parameters.

- **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.

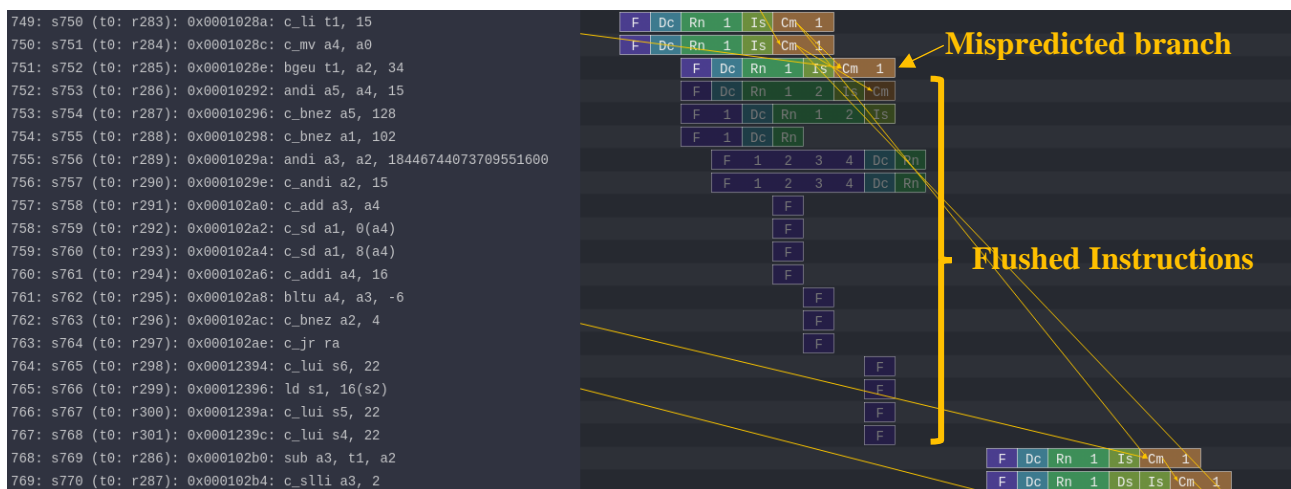


Figure 2: Example of a branch *misprediction* (transparent rows)

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, 0(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.

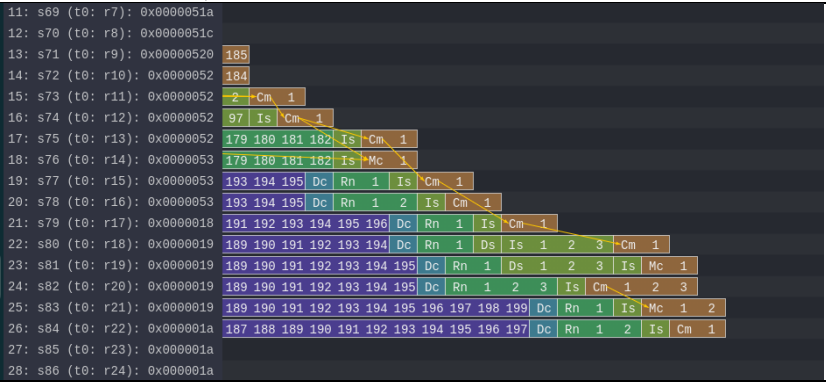
Exercise 1:

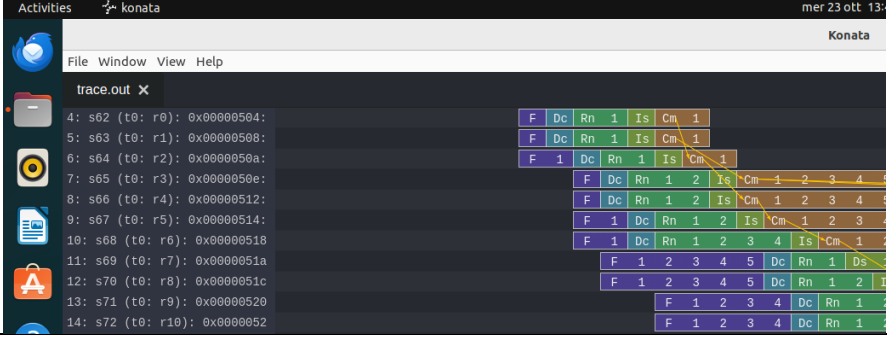
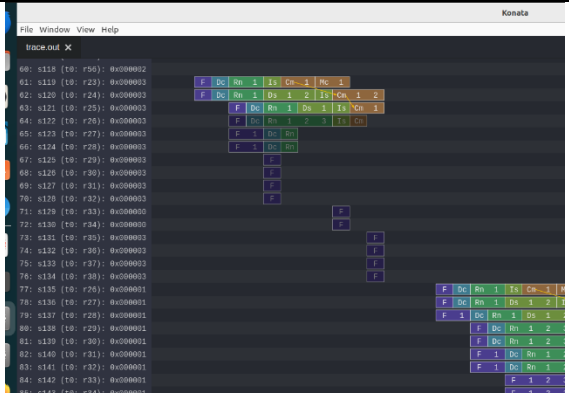
Simulate the benchmark *my_c_benchmark_2* (*main.c*) by using the gem5 simulator to obtain the *trace.out* file. Then, you can visualize the pipeline (i.e., load the *trace.out* file on Konata).

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	
Explain the reason behind the condition	L'esecuzione out-of-order permette a istruzioni indipendenti di essere completate in un ordine diverso rispetto a quello del programma. Tuttavia, grazie al Reorder Buffer (ROB) , il commit delle istruzioni avviene in ordine, mantenendo la consistenza del flusso di esecuzione.

<p>Briefly explain the advantages of the OoO execution in a CPU</p>	<p>Questa tecnica consente di sfruttare meglio le risorse della CPU, eseguendo parallelamente istruzioni indipendenti, e minimizza i tempi di inattività delle unità di calcolo. Così si ottimizza il throughput complessivo del processore senza sacrificare l'integrità dei dati.</p>
<p>Condition</p>	<p>Two or more commits in the same clock cycle</p>
<p>Screenshot from Konata</p>	
<p>Explain the reason behind the condition</p>	<p>Questa condizione si verifica perché il parametro commitWidth è impostato a 2, permettendo al processore di eseguire il commit di due istruzioni nello stesso ciclo. In questo modo, il ROB può liberare più slot per le nuove istruzioni, migliorando la capacità della pipeline di ricevere e processare istruzioni con maggiore efficienza.</p>
<p>Briefly explain the Commit functioning</p>	<p>La fase di commit finalizza i risultati delle istruzioni, scrivendoli nei registri o in memoria. Il passaggio dei dati dal ROB alla destinazione finale avviene in modo ordinato, garantendo che i risultati siano corretti anche con esecuzione out-of-order.</p>
<p>Condition</p>	<p>Flush of the pipeline</p>
<p>Screenshot from Konata</p>	
<p>Explain the reason behind the condition</p>	<p>Il flush della pipeline è una conseguenza di una mispredizione di branch. Se la previsione di un branch è errata, tutte le istruzioni speculative eseguite sulla base di quella previsione vengono rimosse dalla pipeline (le istruzioni successive al branch (anche se già eseguite)). Questo processo evita che i risultati di istruzioni eseguite in modo speculativo influenzino lo stato finale della CPU. La causa principale è la speculazione sulle istruzioni branch. Quando la previsione risulta sbagliata, la pipeline deve ripristinare lo stato corretto, eliminando tutte le istruzioni successive al branch errato. Anche se costoso in termini di prestazioni, questo processo è essenziale per mantenere la correttezza dei dati e l'affidabilità della CPU.</p>

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 *fetchWidth*, *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_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 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	Configuration 5
First changed parameter	CPU_IntALU->count=6	L1Cache-tag_latency=1	the_cpu.numROB Entries = 128	L1Cache-tag_latency=1	L1Cache-tag_latency=1
Second changed parameter		L1Cache.response_latency=1	the_cpu.numIQEntries = 6	L1Cache.response_latency=1	L1Cache.response_latency=1
Third changed parameter		L1Cache.data_latency=1		L1Cache.data_latency=1	L1Cache.data_latency=1
Fourth changed parameter		L1Cache.assoc=4		L1Cache.assoc=4	L1Cache.assoc=4
Fifth changed parameter				the_cpu.numROB Entries = 128	the_cpu.numROB Entries = 128
6 th changed parameter				the_cpu.numIQEntries = 6	the_cpu.numIQEntries = 12

Original CPI (no hardware optimization): 2.190180

	Configuration 1	Configuration 2	Configuration 3	Configuration 4	Configuration 5
CPI	2.190180	2.047219	1.605301	1.447422	1.202202
Speedup (wrt Original CPI)	1	1.069831806	1.364342264	1.513159258	1.821806984

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

Your answer:

Conf1: Nonostante il fatto che le istruzioni intere (int ALU) rappresentino circa il 58,88% delle operazioni totali (system.cpu.statIssuedInstType_0::IntAlu 4096 58.88% 58.88% # Number of instructions issued per FU type, per thread (Count)), l'incremento del numero di unità ALU (come fatto nella Configurazione 1) non ha prodotto miglioramenti nel CPI, che è rimasto invariato, e quindi non ha portato alcun guadagno in termini di speedup (speedup pari a 1). Questo è dovuto al fatto che molte delle istruzioni che richiedono l'uso dell'ALU intera sono vincolate da dipendenze con altre istruzioni. Di conseguenza, anche se sono disponibili più ALU, queste istruzioni devono comunque attendere che le istruzioni precedenti completino il commit per poter essere eseguite. In questo scenario, avere 3 ALU o 6 ALU non cambia il risultato, poiché il limite principale è rappresentato dalle dipendenze tra le istruzioni, non dalla disponibilità delle unità di esecuzione.

Conf2: Qui si agisce sulla cache di livello 1 (L1), riducendo i tempi di latenza per le operazioni di ricerca e accesso (tag_latency, data_latency, response_latency). La riduzione della latenza permette alla CPU di accedere più rapidamente ai dati richiesti, diminuendo il tempo di attesa tra il processore e la cache L1. L'aumento dell'associatività (assoc = 4) migliora la probabilità di trovare i dati richiesti in cache, riducendo i miss. Questo tipo di ottimizzazione è particolarmente efficace per migliorare il CPI, poiché abbassa i tempi di accesso ai dati frequentemente usati.

Benefici dell'aumento dell'associatività a 4 vie:

- Riduzione dei miss da conflitto: Con una cache a 4 vie, è più probabile che blocchi con lo stesso indice di set possano coesistere senza sovrapporsi, riducendo i miss causati da conflitti e aumentando la probabilità di trovare i dati necessari.
- Aumento della probabilità di hit: Maggiori blocchi per set permettono di mantenere più dati frequentemente richiesti, riducendo i tempi di accesso e migliorando l'efficienza complessiva della CPU, con conseguente abbassamento del CPI.

- Compromesso tra efficienza e complessità: Aumentare l'associatività richiede più hardware, ma una cache a 4 vie resta più gestibile e veloce rispetto a una cache fully associative.

Conf3: In questa configurazione, viene incrementata la dimensione del Reorder Buffer (ROB) e della Instruction Queue (IQ). Il ROB è responsabile di mantenere l'ordine delle istruzioni eseguite out-of-order per garantire che i risultati siano scritti in ordine. Aumentando numROBEntries, è possibile gestire un numero maggiore di istruzioni simultaneamente, migliorando la parallelizzazione. Allo stesso modo, aumentando numIQEntries, si incrementa il numero di istruzioni pronte per essere emesse dalla coda, riducendo la probabilità di stalli. Questa configurazione riduce ulteriormente il CPI, migliorando lo speedup.

Conf4: Questa configurazione combina i miglioramenti della cache L1 (come nella configurazione 2) e dell'instruction queue e del reorder buffer (come nella configurazione 3). La combinazione di queste ottimizzazioni permette di ridurre ancora di più il CPI rispetto alle singole configurazioni, poiché si riducono sia i tempi di accesso ai dati che la capacità di gestione delle istruzioni out-of-order. Questa è una configurazione molto equilibrata, che porta a un CPI notevolmente più basso e a un buon speedup.

Conf5: In questa configurazione finale, viene ulteriormente aumentato il numero di IQ entries a 12. Questo cambiamento permette di mantenere in coda un numero ancora maggiore di istruzioni pronte per l'esecuzione, riducendo i tempi di attesa per la CPU e aumentando il throughput del processore. Con più IQ entries, la pipeline è meno soggetta a rallentamenti causati dalla mancanza di istruzioni pronte, riducendo ulteriormente il CPI. Questa configurazione porta al miglior CPI tra tutte, con uno speedup significativo, poiché ottimizza la capacità della CPU di processare istruzioni in parallelo.