CO Project 4 Report

R13942128 戴麒恩

I. Introduction

In this project, I explore the behavior of superscalar out-of-order processors using the gem5 simulator and the Konata visualization tool. Superscalar processors can issue and execute multiple instructions per cycle by exploiting instruction-level parallelism (ILP). To investigate how ILP and microarchitectural parameters affect pipeline performance, I designed two basic test programs: one with high ILP (independent instructions) and one with low ILP (dependent instruction chains). Each program was executed under eight different configurations, varying parameters such as fetch width, issue width, commit width, reorder buffer size, and the number of physical registers. The remainder of this report is organized as follows:

- **Experimental setup:** An explanation of my test programs and the configurations used.
- **Performance comparison:** A comparison of the number of ticks and Konata trace under different configurations.
- **Pipeline phenomenon identification:** Identification and explanation of each required pipeline behavior using Konata traces, in order to address the analysis tasks listed in the Project 4 guideline.
- **Discussion:** Insights and observations drawn from my experiments.
- **Conclusion:** A summary of key findings and what I have learned through this project.

II. Experimental setup

In this part, I divide the setup into two sections: the test programs and the microarchitectural configurations. I first introduce the design and purpose of my high ILP and low ILP test programs. Then, I describe the configuration parameters used in gem5, including how they were varied to evaluate different pipeline behaviors.

Test programs:

- High ILP version (low data dependencies):

```c
int main() {
    // Force input variables into specific registers
    register int a asm("x18") = 11;
    register int b asm("x19") = 22;

     // Declare result variables, mapped to different registers
    register int s asm("x20");
    register int t asm("x21");
    register int u asm("x22");
    register int v asm("x23");
    register int w asm("x24");
    register int x asm("x25");
    register int y asm("x26");
    register int z asm("x27");

    // Repeat the same set of independent arithmetic operations 1000 times
    for (int i = 0; i < 1000; i++) {
        asm volatile (
            // Perform addition: s = a + b
            "add x20, x18, x19\n"
            // Perform subtraction: t = a - b
            "sub x21, x18, x19\n"
            // Perform multiplication: u = a * b
            "mul x22, x18, x19\n"
            // Perform addition: v = a + b
            "add x23, x18, x19\n"
            // Perform multiplication: w = a * b
            "mul x24, x18, x19\n"
            // Perform addition: x = a + b
            "add x25, x18, x19\n"
            // Perform subtraction: y = a - b
            "sub x26, x18, x19\n"
            // Perform multiplication: z = a * b
            "mul x27, x18, x19\n"
        );
    }

    // Sum the results from the final iteration and print it
    int result = s + t + u + v;
    printf("Result = %d\n", result);
    return 0;
}
```

high_ILP.c within the .tgz archive

Description:

The high_ILP.c program consists of a loop that performs eight independent arithmetic operations in each iteration, including additions, subtractions, and multiplications. All operations use only the input constants a and b, and each result is written to a distinct physical register. This design ensures there are no data dependencies among the instructions, allowing them to be freely scheduled, issued, and executed in parallel. This program is ideal for evaluating superscalar processor features such as multiple instruction issue, out-of-order execution, and register renaming.

- Low ILP version (serial data dependencies)

```c
int main() {
    // Force input values into specific registers
    register int a asm("x18") = 11;
    register int b asm("x19") = 22;

    // Use fixed register x10 for chaining result
    register int r asm("x20") = 0;

    // Perform a chain of dependent operations 1000 times
    for (int i = 0; i < 1000; i++) {
        asm volatile (
            // Perform addition: r = r + a
            "add x20, x20, x18\n"
            // Perform subtraction: r = r - a
            "sub x20, x20, x18\n"
            // Perform multiplication: r = r * b
            "mul x20, x20, x19\n"
            // Perform addition: r = r + a
            "add x20, x20, x18\n"
            // Perform multiplication: r = r * b
            "mul x20, x20, x19\n"
            // Perform addition: r = r + b
            "add x20, x20, x19\n"
            // Perform subtraction: r = r - a
            "sub x20, x20, x18\n"
            // Perform multiplication: r = r * b
            "mul x20, x20, x19\n"

        );
    }

    // Print the final accumulated result
    printf("Result = %d\n", r);
    return 0;
}
```

low_ILP.c within the .tgz archive

Description:

The low_ILP.c program performs a long sequence of arithmetic operations on a single register r (mapped to x20) within each loop iteration. Every instruction reads from and writes to the same register, forming a continuous chain of read-after-write (RAW) dependencies. This strict data dependency forces the processor to execute each instruction in order, without the possibility of parallel issue or execution. As a result, the program is ideal for demonstrating low ILP, pipeline serialization, and the effects of resource stalls in out-of-order superscalar processors.

Microarchitectural configurations:

To evaluate how different microarchitectural parameters affect ILP and pipeline performance, I tested each program under eight distinct configurations. Each configuration isolates one specific parameter, such as fetch width, issue width, commit width, ROB size, IQ size, or the number of physical registers, while keeping the others constant. And also, I have tested with maximize the three widths. The table below lists all configurations used in this experiment.

These configurations are also written in my submitted .tgz archive as configs.txt. For easier description in the following sections of this report, I refer to each configuration using its corresponding config name listed in the table 1 below.

- **Default**: This is the baseline configuration, which uses the default values in gem5.

- **FW Reduce**: This configuration reduces the fetch width to 1, limiting how many instructions can be fetched per cycle.
- **IW Reduce**: This configuration reduces the issue width to 1, restricting the processor to issuing only one instruction per cycle.
- **CW Reduce**: This configuration reduces the commit width to 1, limiting the number of instructions that can retire per cycle.
- **ROBS Reduce**: This configuration sets the reorder buffer (ROB) size to 1, severely limiting out-of-order execution.
- **IQS Reduce**: This configuration reduces the issue queue size to 1, which constrains the number of instructions waiting to be issued.
- **Num PhyReg Reduce**: This configuration limits the number of physical registers to 1, affecting register renaming and exposing WAW/WAR hazards.
- **Half of default**: All six parameters are set to half of their default values to observe the impact of uniformly reduced resources.

The complete microarchitectural configurations are summarized in Table1:

Config name	Fetch width	Issue width	Commit width	ROB size	IQ size	num_PhyReg
Default	8	8	8	192	64	256
FW reduce	1	8	8	192	64	256
IW reduce	8	1	8	192	64	256
CW reduce	8	8	1	192	64	256
ROBS reduce	8	8	8	1	64	256
IQS reduce	8	8	8	192	1	256
Num PhyReg reduce	8	8	8	192	64	34
Half of default	4	4	4	96	32	128

III. Performance comparison

To evaluate how different microarchitectural configurations impact pipeline performance, I compare the number of ticks for both high ILP and low ILP test programs under each configuration. I also analyze the pipeline behavior using Konata trace visualizations to identify how these configurations influence instruction throughput, parallelism, and pipeline efficiency.

Tick comparison:

The ticks of each configuration executed on both High ILP program and Low ILP program are listed in the table below.

Config name	high ILP program	low ILP program
Default	9433500	13622000
FW reduce	17930000	17514000
IW reduce	14562500	14416500
CW reduce	14263000	14082000
ROBS reduce	62809500	63278000
IQS reduce	34441000	34581500
Num PhyReg reduce	60756000	61190000
Half of default	11005500	14109000

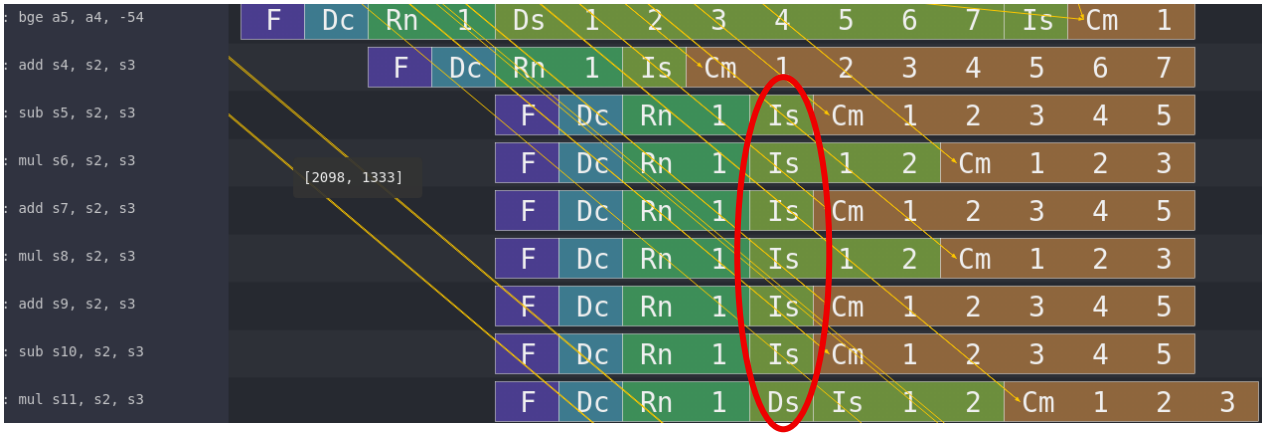
The performance results clearly demonstrate how ILP interacts with microarchitectural design and impacts execution efficiency. As shown in the tick data, the high ILP program consistently outperforms the low ILP program under wider or less constrained configurations, while the low ILP program shows limited improvement due to its inherent data dependencies.

In the **Default** configuration, the high ILP program completes in 9.43M ticks, while the low ILP program takes 13.62M ticks. When the fetch width is reduced to 1 (**FW Reduce**), both high ILP and low ILP program increase their tock counts to around 17M, but what we should observe is the degrade compare with the **Default**, high ILP program has a large gap indicate that the high ILP program are benefit from superscalar processor. And also indicates that fetch bandwidth is critical for high ILP workloads, which rely on the ability to feed multiple independent instructions into the pipeline per cycle. Similarly, reducing issue width and commit width to 1 significantly hurts the high ILP program, while having a smaller impact on the low ILP program, because low ILP program has less parallelism. In contrast, configurations like **ROB Size Reduced**, **IQ Size Reduced**, and **Physical Register Reduced** drastically degrade the performance of both programs, because they limit core mechanisms like out-of-order execution and register renaming. As for the **Half of default** configuration, it results in a moderate increase in tick counts, especially for the high ILP program. This reflects the reduced ability to exploit instruction-level parallelism due to narrower pipeline resources, including fetch, issue, and commit bandwidth, as well as limited ROB and IQ capacity.

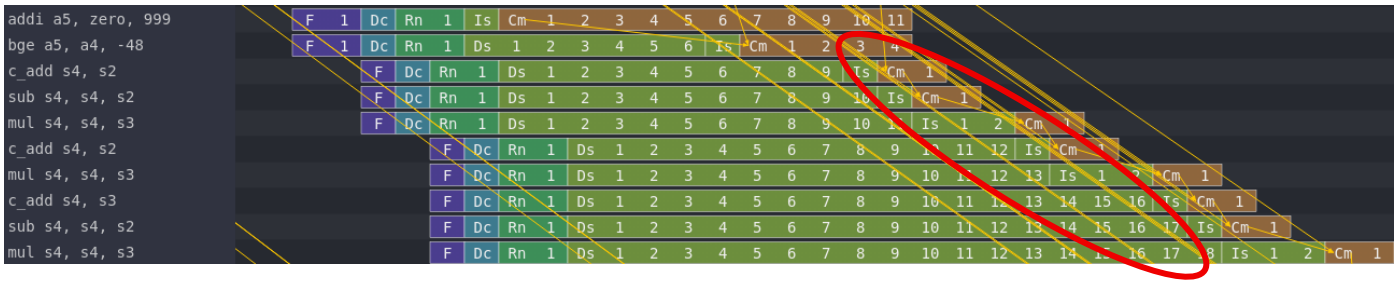
In conclusion, programs with high ILP benefit the most from superscalar processors, as these architectures can parallelize instruction execution when microarchitectural resources are sufficient.

Pipeline behavior analysis:

- **Default:**



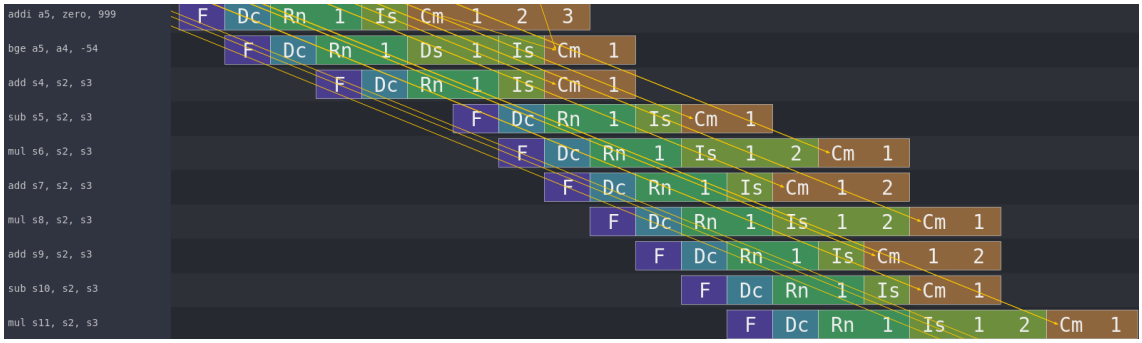
Screenshot from the Konata trace of high_ILP.c



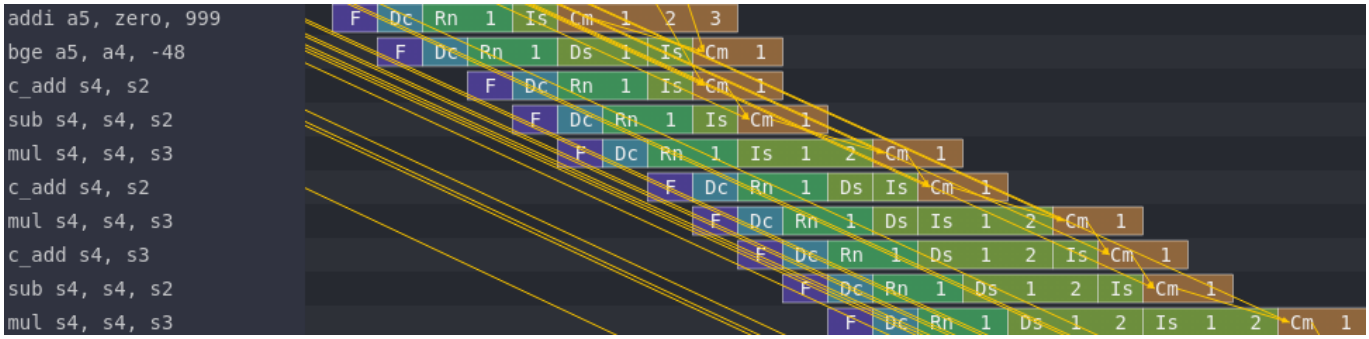
Screenshot from the Konata trace of low_ILP.c

Under the **Default** configuration, the high ILP trace shows multiple instructions being issued and executed in parallel. The pipeline stages are overlapped, and instructions are committed out-of-order, indicating good utilization of superscalar and out-of-order execution capabilities. In contrast, the low ILP trace demonstrates serialized execution, where each instruction waits for the previous one to complete due to data dependencies. This results in a stair-step pattern with minimal overlap, highlighting the limited ILP and increased stalls in the pipeline.

- **FW reduce:**



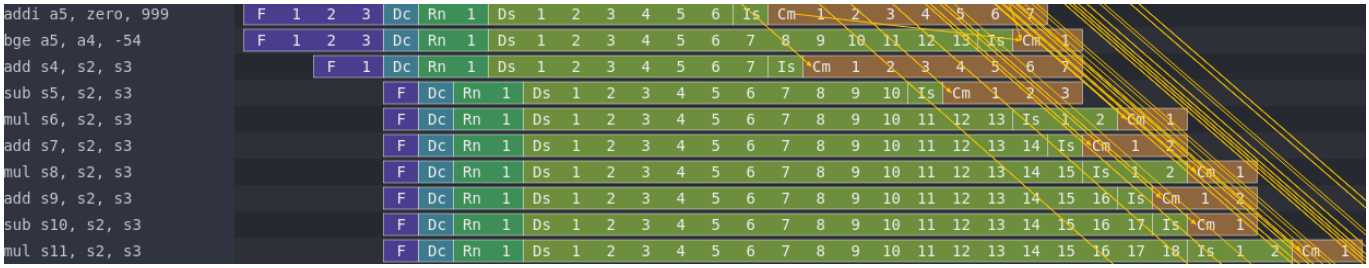
Screenshot from the Konata trace of high_ILP.c



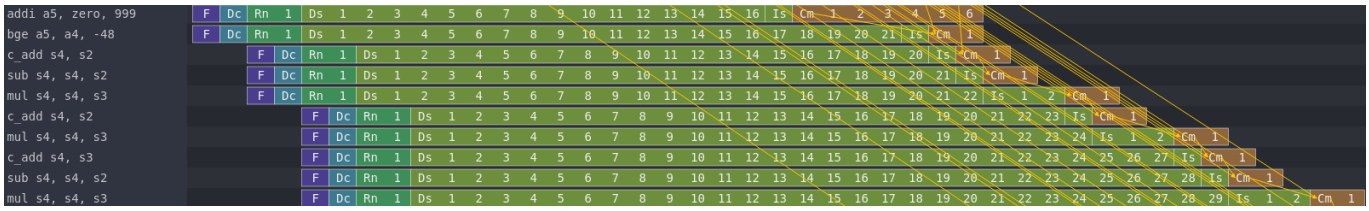
Screenshot from the Konata trace of low_ILP.c

Under the **FW Reduce** configuration, both high ILP and low ILP traces show similar serialized pipeline behavior. Since the fetch width is limited to one instruction per cycle, only one instruction can enter the pipeline at a time, regardless of ILP potential. In the high_ILP trace, although instructions are independent, the processor cannot issue them in parallel due to the fetch bottleneck. As a result, the trace shows a one-by-one progression through the pipeline stages, similar to the low_ILP trace where instruction dependencies naturally enforce serialization. This demonstrates that front-end bandwidth is a critical factor in enabling superscalar execution.

- **IW reduce:**



Screenshot from the Konata trace of high_ILP.c

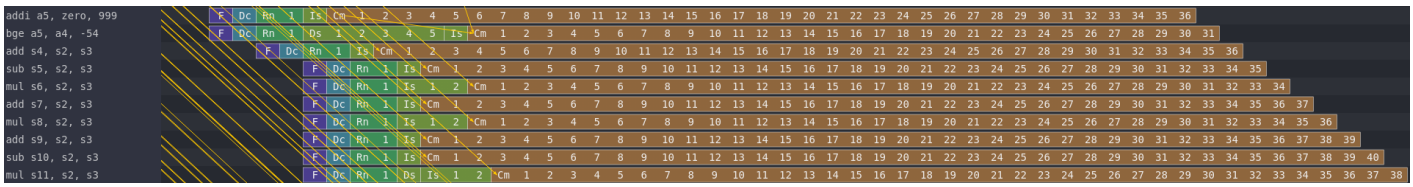


Screenshot from the Konata trace of low_ILP.c

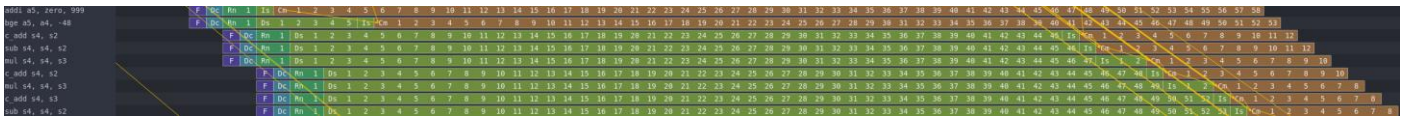
Under the **IW Reduce** configuration, the issue width is limited to one instruction per cycle. As shown in the trace, even though the high_ILP program has multiple independent instructions and fetches them in parallel, only one instruction can be issued at a time. This creates a serialized pipeline flow similar to that of the low_ILP program. Both traces show instructions entering the issue stage sequentially, with minimal overlap in execution. This confirms that the issue stage bandwidth is a critical

factor in realizing ILP, and limiting it can neutralize the advantage of instruction independence.

- **CW reduce:**



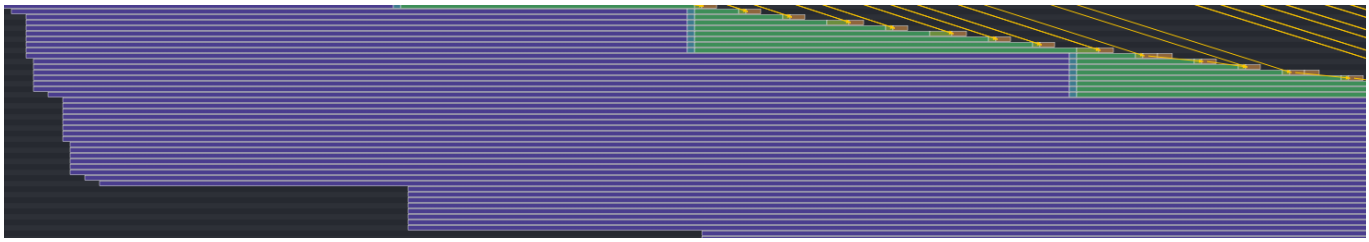
Screenshot from the Konata trace of high_ILP.c



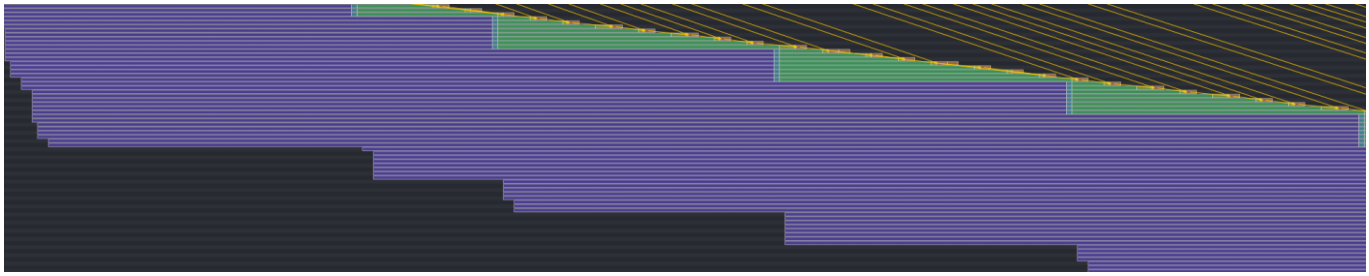
Screenshot from the Konata trace of low_ILP.c

Under the **CW Reduce** configuration, the issue width remains wide, allowing multiple instructions to be issued in parallel. However, the commit width is restricted to one, which becomes the performance bottleneck. As shown in the trace, many instructions are issued around the same time, but the commit stage progresses one instruction per cycle. This serialization at the end of the pipeline leads to backpressure, extending the overall execution time even though the front-end and execution units are underutilized.

- **ROBS reduce:**



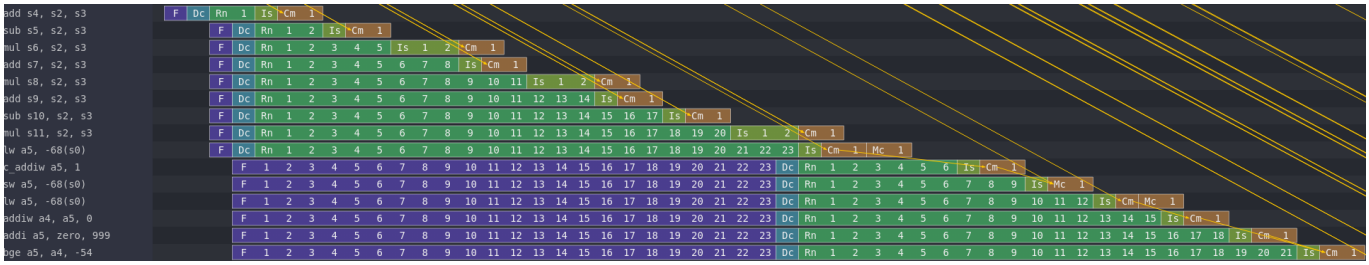
Screenshot from the Konata trace of high_ILP.c (too long so no details)



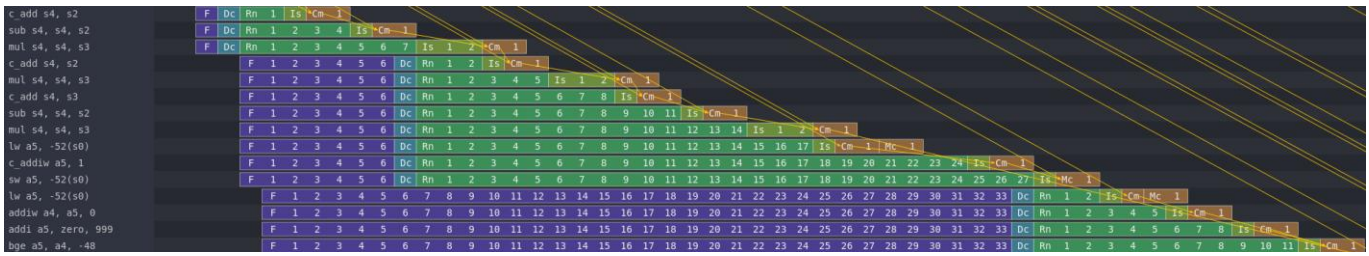
Screenshot from the Konata trace of low_ILP.c (too long so no details)

Under the **ROBS Reduce** configuration (ROB size = 1), both traces show severe pipeline serialization. Since only one instruction can reside in the reorder buffer at a time, all subsequent instructions must wait for the current one to retire before proceeding. In the high_ILP trace, even though the instructions are independent, they are forced to execute one at a time, eliminating any benefit from out-of-order or superscalar execution. The low_ILP trace exhibits similar behavior due to its intrinsic dependencies, but the performance gap between the two programs nearly disappears under this constraint. Overall, reducing ROB size to one effectively disables the processor's ability to exploit ILP, resulting in heavy front-end stalls and significant performance degradation for both programs.

- **IQS reduce:**



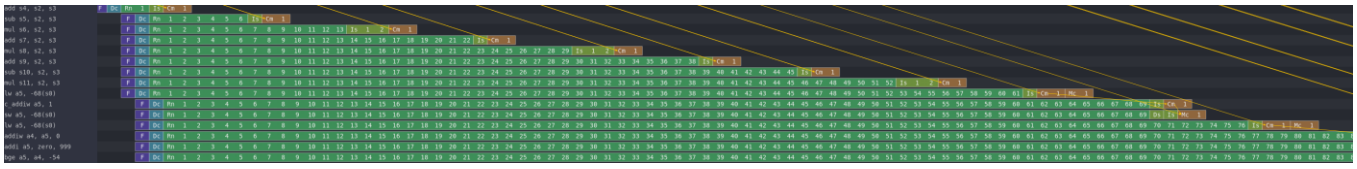
Screenshot from the Konata trace of high_ILP.c



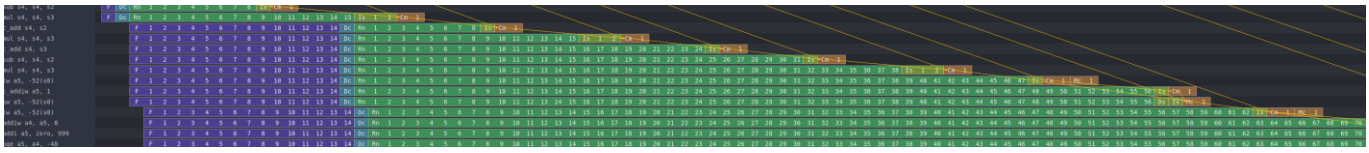
Screenshot from the Konata trace of low_ILP.c

Under the **IQS Reduce** configuration, the issue queue is limited to a single entry. As shown in the trace, both high_ILP and low_ILP programs experience instruction dispatch stalls, since only one instruction can occupy the queue at a time. In the high_ILP trace, although multiple independent instructions are fetched and renamed, they are blocked from entering the issue stage simultaneously. This results in serialized execution and the loss of multiple-issue capability. In the low_ILP trace, the behavior remains similar to its default case, as its intrinsic dependencies already prevent parallel issue. Overall, a constrained issue queue neutralizes the benefit of ILP by restricting instruction flow before execution can even begin.

- **Num PhyReg reduce:**



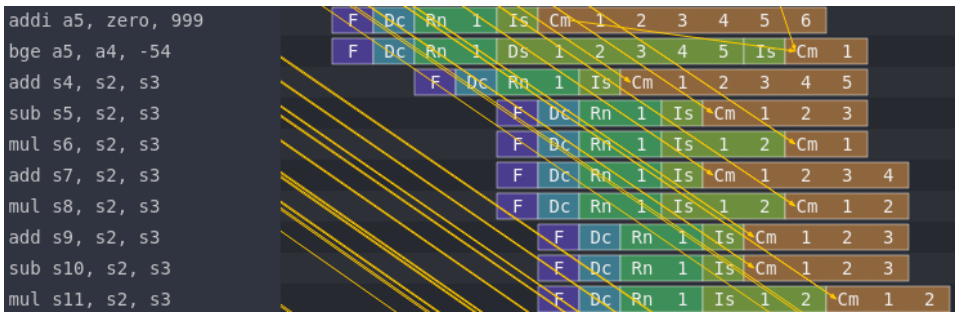
Screenshot from the Konata trace of high_ILP.c



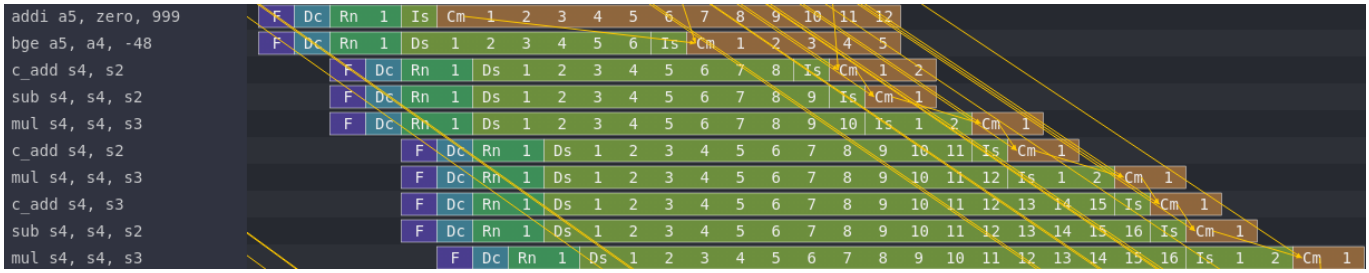
Screenshot from the Konata trace of low_ILP.c

Under the **Num PhyReg Reduce** configuration, the number of physical registers is reduced to the minimum required for execution, leaving almost no room for register renaming. In the high_ILP trace, we observe severe stalls at the rename stage, where instructions cannot proceed due to the lack of free physical registers. This eliminates the processor's ability to resolve write-after-write (WAW) and write-after-read (WAR) hazards, causing the pipeline to behave almost in-order despite having independent instructions. The low_ILP trace is similarly affected, though its performance was already limited by data dependencies. Overall, the lack of rename resources results in significant pipeline backpressure and poor ILP exploitation.

- **Half of default:**



Screenshot from the Konata trace of high_ILP.c

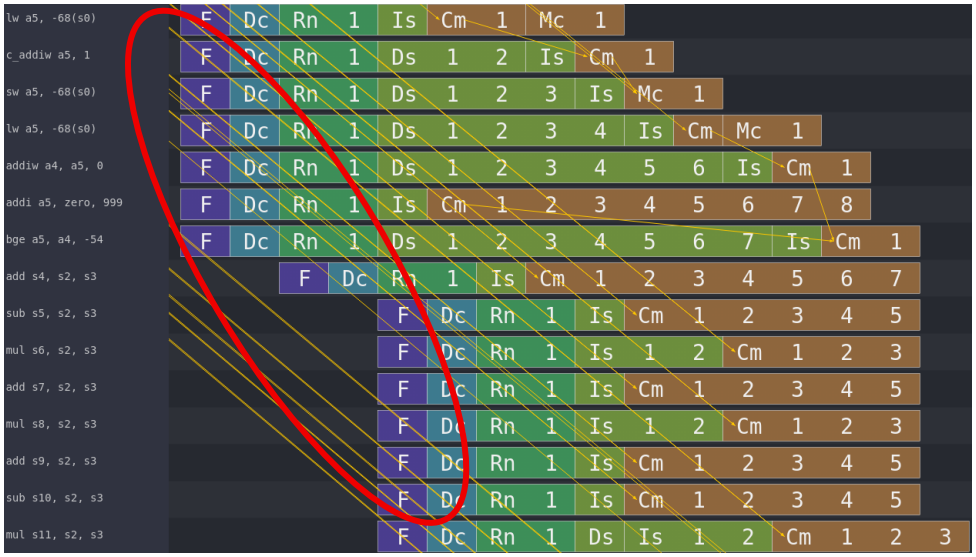


Screenshot from the Konata trace of low_ILP.c

Under the **Half of Default** configuration, the processor operates with half the default widths and buffer sizes, leading to visibly reduced pipeline parallelism. In the high_ILP trace, fewer instructions are issued and executed concurrently compared to the Default case, resulting in longer and more serialized commit stages. The low_ILP trace remains largely unchanged, as its inherent dependencies dominate execution behavior. This comparison highlights how reducing core pipeline resources limits the processor's ability to exploit ILP, particularly in programs with high instruction independence.

IV. Pipeline phenomenon identification

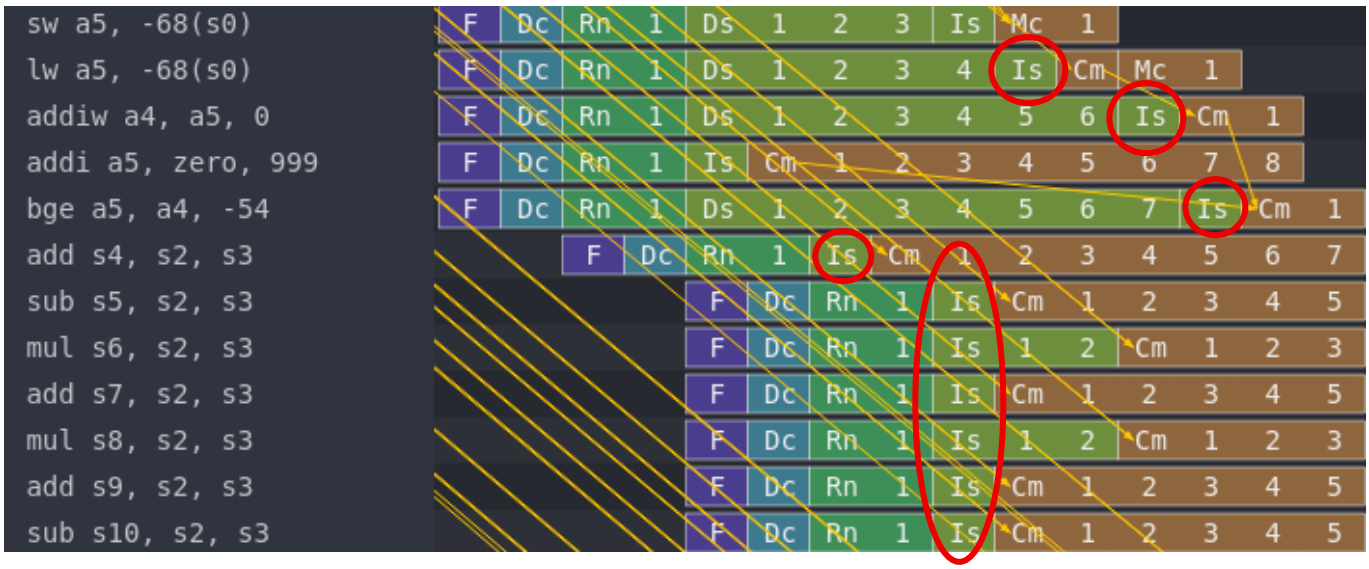
- In-order fetch and decode



Screenshot from the Konata trace of high_ILP.c with Default configuration

As shown in the trace, multiple instructions can enter the fetch (F) stage in parallel, resulting in horizontally aligned F stages across different instructions. However, despite the parallelism, the fetch unit maintains strict program order, instructions are always fetched sequentially from memory without skipping ahead or reordering. This confirms that the processor performs widened in-order fetch, where parallelism is achieved by fetching multiple consecutive instructions per cycle, not by fetching future instructions out of order.

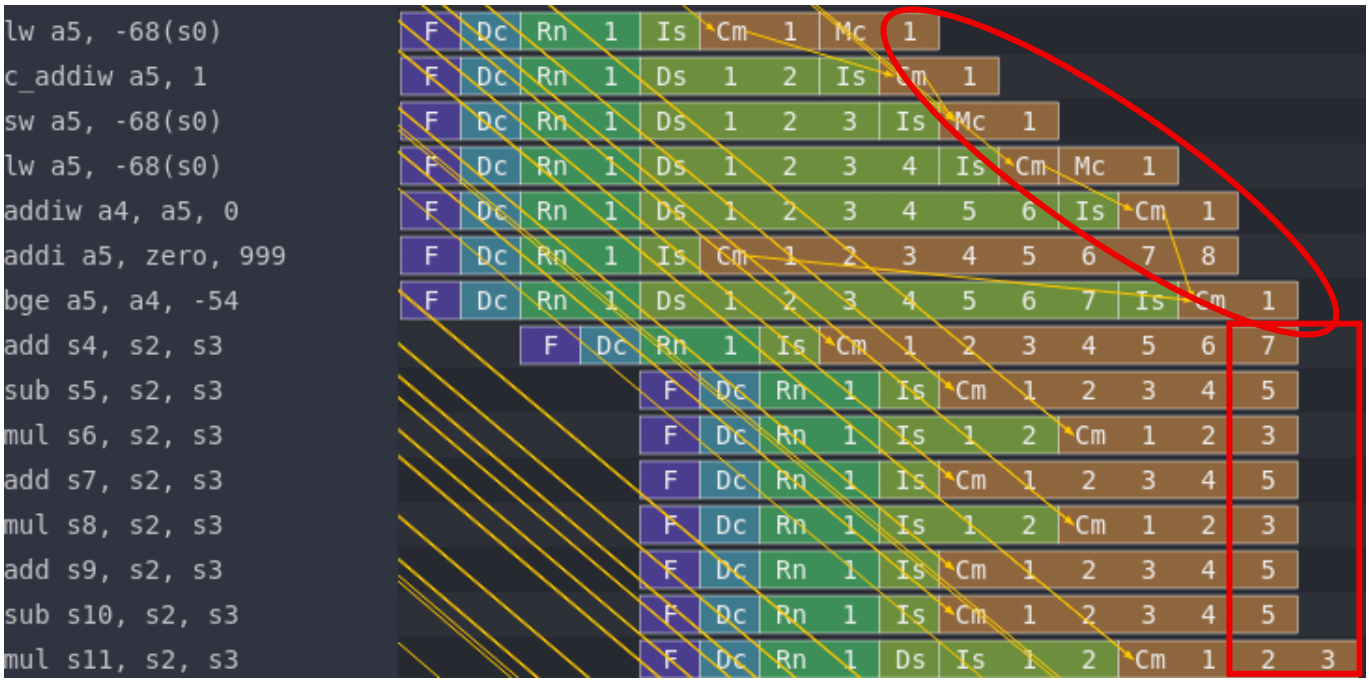
- Out-of-order issue and execution



Screenshot from the Konata trace of high_ILP.c with Default configuration

The trace demonstrates clear out-of-order issue and execution behavior. Although instructions are fetched and decoded in program order, the issue stage (Is) shows instructions being dispatched based on operand readiness rather than their original sequence. For example, earlier instructions like lw or bge may be delayed due to data dependencies or functional unit contention, the processor then allow younger independent instructions to issue first. This behavior confirms the processor's ability to exploit ILP through dynamic scheduling and out-of-order execution.

- In-order commit (Retirement)



Screenshot from the Konata trace of high_ILP.c with Default configuration

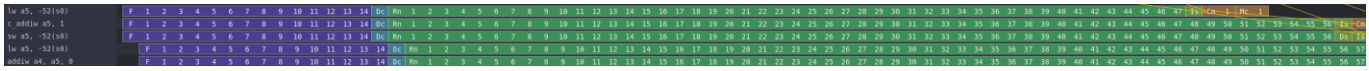
In the screenshot above, we can see that although the instructions are issued out-of-order, they are still committed in program order. This is shown in the Cm stage where the instructions retire sequentially from top to bottom, without any instruction jumping ahead. Even when some younger instructions finish execution earlier, they have to wait until all older instructions ahead of them have retired, it won't happen the phenomenon of the instruction younger retire earlier than the older one. This confirms that the commit stage maintains strict in-order behavior, as expected in out-of-order CPU designs.

- Multiple instructions issued or committed in the same cycle

Screenshot from the Konata trace of high_ILP.c with Default configuration

In this trace segment, we can observe that multiple instructions are issued in the same cycle, as shown in the Is stage where several instructions are aligned vertically. For instance, sub s5, s2, s3 and mul s6, s2, s3… are issued together. These instructions have no data dependencies between them, they read the same source registers but write to different destination registers , so there's no WAR or WAW hazard. Since the processor has enough issue bandwidth and functional units available, it can issue multiple instructions in parallel. This illustrates the processor's ability to exploit ILP by issuing independent instructions simultaneously.

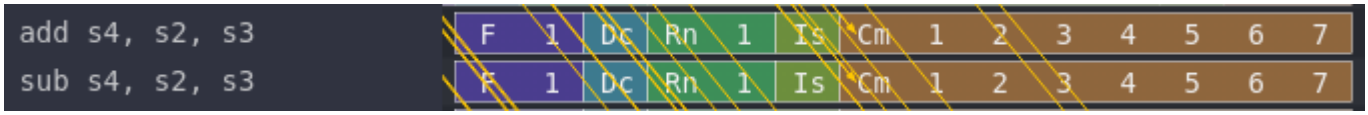
- Pipeline bubbles due to data hazards, control hazards, resource stalls, etc.



Screenshot from the Konata trace of low_ILP.c with Default configuration

In this trace, we observe a classic load-use data hazard. The instruction lw a5, -52(s0) loads a value into register a5, and the following instruction addiw a4, a5, 0 immediately uses that same register. Since the load result is not yet available, the dependent instruction must wait, causing a visible stall in the pipeline. This delay shows up as a bubble between the rename and issue stages (Rn → Is) for the second instruction. This kind of hazard is common when a load is followed closely by an ALU operation that uses the loaded value, and requires the processor to stall to preserve correctness.

- [Bonus] Evidence of renaming eliminating write-after-write or write-after-read hazards



Screenshot from the Konata trace of WAW_example.c with Default configuration

The trace below shows two independent instructions writing to the same architectural register s4. In a traditional in-order pipeline without renaming, the second instruction (sub) would have to wait for issue until the first instruction (add) completes and commits. However, both instructions proceed through the rename and issue stages concurrently, indicating that the processor allocated separate physical registers to resolve the write-after-write hazard.

- [Bonus] Additional phenomena

Screenshot from the Konata trace of WAW_example.c with “Num PhyReg reduce” configuration

Compare with the previous phenomena, here I want to show the opposite of WAW example. In this trace, although two independent instructions write to the same register s4, the second instruction (sub) is delayed in the issue stage. This suggests that the WAW hazard was not completely eliminated by register renaming, possibly due to limited rename register resources. Unlike other cases where such hazards are removed by allocating different physical registers, here the processor enforces sequential execution.

V. Discussion

This project gave me a clearer view of how each microarchitectural component plays a role in actual pipeline behavior. One of the most striking observations was how dramatically high-ILP programs suffer under fetch or issue width constraints, even though their instructions are independent, they effectively become serialized when only one instruction can enter or be issued per cycle. This limitation completely neutralized the advantage of instruction-level parallelism.

On the other hand, configurations like ROB size, issue queue size, or physical register count impacted both high-ILP and low-ILP programs, though in different ways. For example, with a tiny ROB, both traces showed nearly in-order behavior regardless of ILP. Similarly, with only 34 physical registers, I observed instructions getting stuck at the rename stage, a clear sign that resource pressure in the backend can throttle the entire pipeline. Interestingly, even when issue and execution are out-of-order, the

processor strictly maintains in-order commit. This makes sense from a correctness standpoint but also causes backpressure if commit becomes a bottleneck. These patterns really helped clarify which pipeline stages offer flexibility and which act as strict barriers.

Finally, watching how register renaming eliminated WAW hazards until resources became tight was a very intuitive way to internalize the benefits of renaming and how it enables out-of-order execution in the first place.

VI. Conclusion

Through this project, I gained hands-on insight into how superscalar out-of-order processors actually exploit instruction-level parallelism, and what gets in their way. I now understand that ILP isn't just about writing independent code; it also depends heavily on the microarchitecture design, and whether there are enough internal resources like rename registers and issue queue slots to support concurrent execution. The Konata tool was particularly helpful in making abstract concepts visual and concrete. Seeing instructions flow (or get stuck) stage by stage gave me a much better intuition for pipeline dynamics. It also taught me to think about tradeoffs: wider fetch and issue paths help ILP, but only if there are enough backend resources to sustain that throughput.

Overall, this lab made the internals of modern CPUs much more tangible, and gave me a new appreciation for how architecture design directly shapes program performance at the cycle level.