Assignment 4: Exploring Instruction-Level Parallelism (ILP) in Modern Processors

Milan Bista

University of Cumberlands
MSCS-531-M50: Computer Architecture and Design
Instructors: Machica Mcclain / Charles Lively
Github: https://github.com/mbista25742/MSCS-531-M50-Assignment4

Abstract

Instruction-Level Parallelism (ILP) is a crucial measure of a processor's capability to execute multiple instructions simultaneously by overlapping their execution. This paper explores the evolution of ILP, its implementation techniques, and the challenges faced in modern processor designs. It also discusses performance metrics related to ILP, current challenges, and future directions for research, highlighting the importance of ILP in enhancing computational throughput without increasing clock frequency.

Introduction

Instruction-Level Parallelism (ILP) is defined as the ability of a processor to execute multiple instructions concurrently, capitalizing on the inherent parallelism found in instruction sequences. By exploiting the independence of instructions, ILP maximizes computational throughput within a single processor core. Increasing the number of instructions processed per clock cycle enhances performance without necessitating an increase in clock frequency, which is vital due to physical and thermal constraints on clock speeds.

Techniques for Implementing ILP

ILP is realized through various techniques, including:

1. **Pipelining**: This technique allows different stages of instruction execution to occur simultaneously, thereby increasing efficiency.
2. **Superscalar Execution**: In this approach, multiple instructions are dispatched to parallel pipelines, enabling multiple instructions to be processed per clock cycle.
3. **Out-of-Order Execution**: This allows instructions to be executed as soon as their required data becomes available, rather than strictly following program order, thereby improving throughput.
4. **Speculative Execution**: Processors preemptively execute instructions based on predicted paths of conditional branches, maintaining a steady flow of operations.

Each of these techniques addresses independent instructions that can be processed concurrently, thus reducing idle cycles and enhancing overall efficiency.

Dependencies in Instruction Streams

ILP is fundamentally reliant on the identification and management of dependencies within instruction streams, which include:

- **Data Dependencies**: Occur when one instruction requires the output of a previous instruction (e.g., read-after-write).
- **Control Dependencies**: Arise from branches or conditional instructions affecting the flow of execution.
- **Resource Conflicts**: Occur when hardware resources are insufficient for parallel execution.

By addressing these dependencies, ILP has become essential in modern processor design, enhancing performance across various applications.

Historical Context of ILP

The evolution of ILP has significantly advanced modern processor design, beginning with basic pipelining techniques in the 1960s. Early pipeline architectures, such as the IBM Stretch and CDC 6600, laid the groundwork by allowing multiple stages of instruction execution to overlap.

In the 1980s, the introduction of superscalar architectures, exemplified by the IBM System/360 Model 91, allowed processors to execute multiple instructions per clock cycle. This marked a significant shift in how processors utilized parallelism within a single thread of instructions.

The 1990s saw the introduction of out-of-order execution in high-performance processors like the Intel Pentium Pro, enabling instructions to be processed as their data became available. This adaptive management of instruction sequences further increased ILP.

Techniques for Exploiting ILP

Modern processors leverage various techniques to maximize ILP:

- **Dynamic Scheduling and Out-of-Order Execution**: Allow instructions to be executed as soon as dependencies are resolved.
- **Register Renaming**: Helps resolve false dependencies by assigning unique hardware registers to instructions.
- **Branch Prediction and Speculative Execution**: Predicts the outcomes of branches and executes instructions along predicted paths.
- **Superscalar Execution**: Dispatches multiple instructions per clock cycle across functional units.
- **Vectorization**: Supports parallel execution within instructions, ideal for data-heavy operations.

Challenges in ILP

Despite its benefits, ILP faces several challenges:

1. **Increasing Complexity**: Modern processors' architectures have become complex, making effective ILP implementation challenging.
2. **Diminishing Returns**: Traditional ILP techniques have shown diminishing returns as transistor scaling slows down.
3. **Power Constraints**: High ILP designs can lead to increased power usage and thermal issues.
4. **Memory Bottlenecks**: High demand for memory bandwidth can hinder the full exploitation of ILP.

Addressing ILP Challenges

To address these challenges, researchers are exploring various techniques:

- **Heterogeneous Architectures**: Integrating specialized cores alongside traditional CPU cores for optimized execution.
- **Machine Learning-Based Optimizations**: Utilizing machine learning to enhance scheduling, resource allocation, and branch prediction.
- **Dynamic Voltage and Frequency Scaling (DVFS)**: Adjusting power consumption dynamically based on workload requirements.
- **Novel Instruction Set Architectures (ISAs)**: Developing ISAs that support efficient parallel execution.

Future Directions for ILP Research

Emerging trends suggest promising directions for future ILP research, including:

1. **Specialized Accelerators**: The rise of domain-specific accelerators for workloads like AI is likely to redefine ILP considerations.
2. **Quantum Computing**: The implications of quantum algorithms on ILP could yield significant insights.
3. **Enhanced Memory Architectures**: Innovations in memory technologies could alleviate bottlenecks and enable better ILP exploitation.
4. **Software-Driven Approaches**: Sophisticated software tools and compilers can optimize instruction scheduling and resource management.
5. **Energy-Efficient Designs**: Research into energy-efficient designs that maintain high performance will be crucial as power constraints become increasingly pressing.

Conclusion

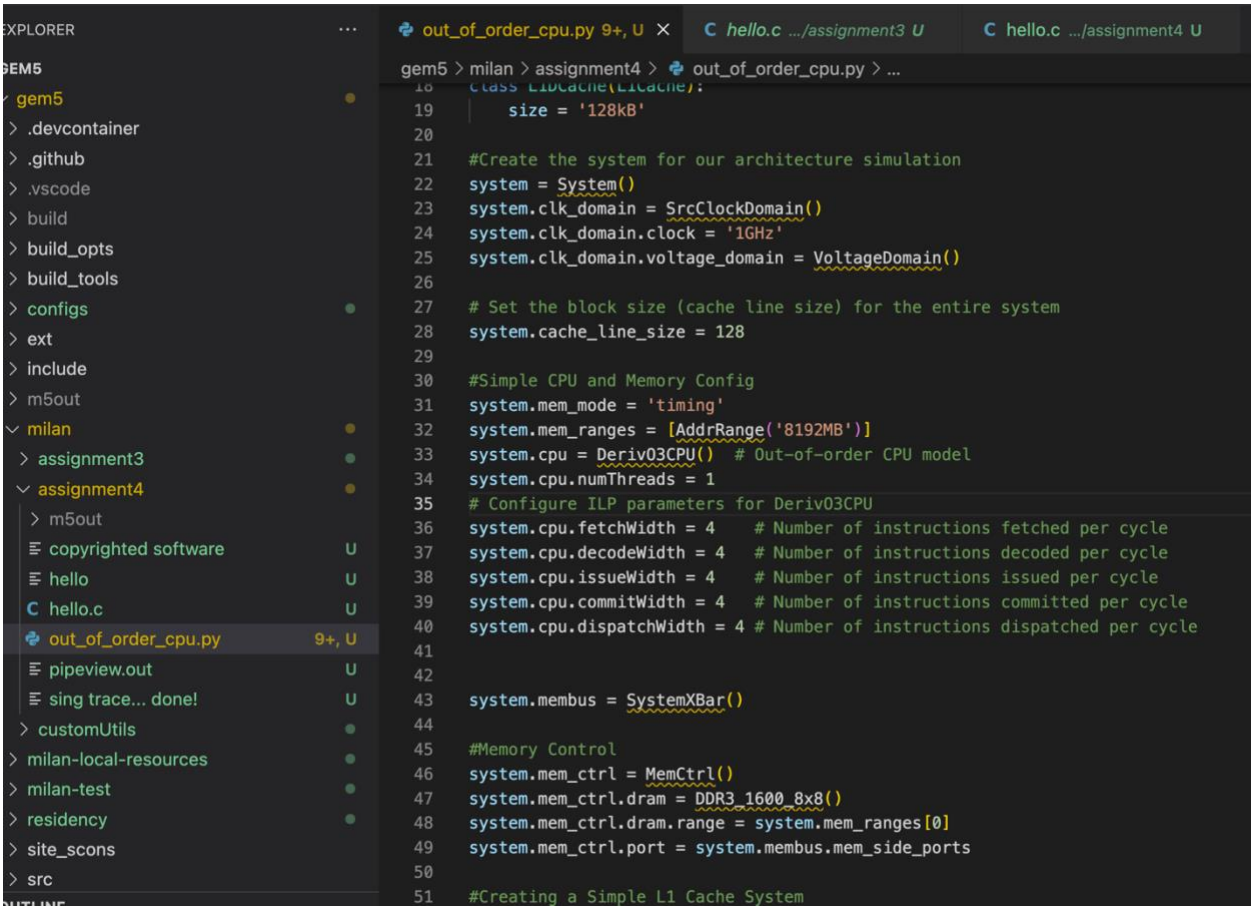
The evolution of Instruction-Level Parallelism (ILP) reflects a continual effort to maximize computational efficiency in processors, adapting to the evolving demands of hardware and software. This journey has driven innovations that have not only improved processor performance but also laid the groundwork for the complex architectures seen in today's computing landscape. As challenges persist, ongoing research will play a vital role in harnessing ILP to meet future computing needs.

Part 2: Practical Exploration of ILP Techniques

In this part, I am going to simulate Instruction Level Parallelism in gem5

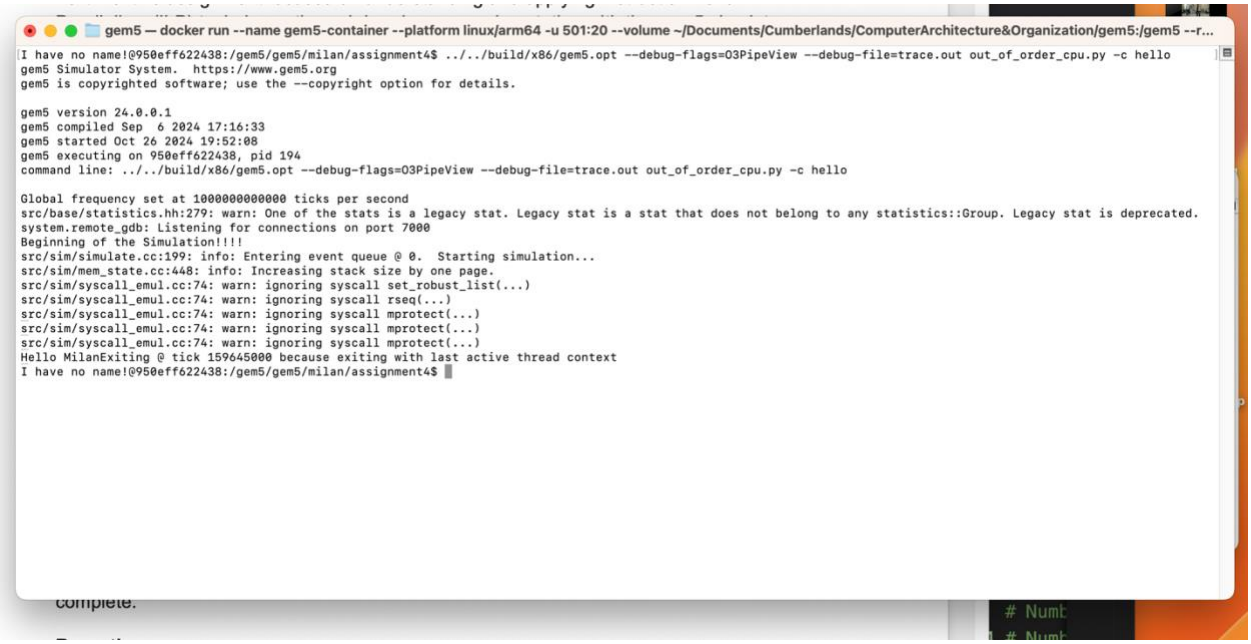
gem5 Configuration:

I have created a simple out of order cpu model that will help us simulate and visualize fetch decode and execute instructions



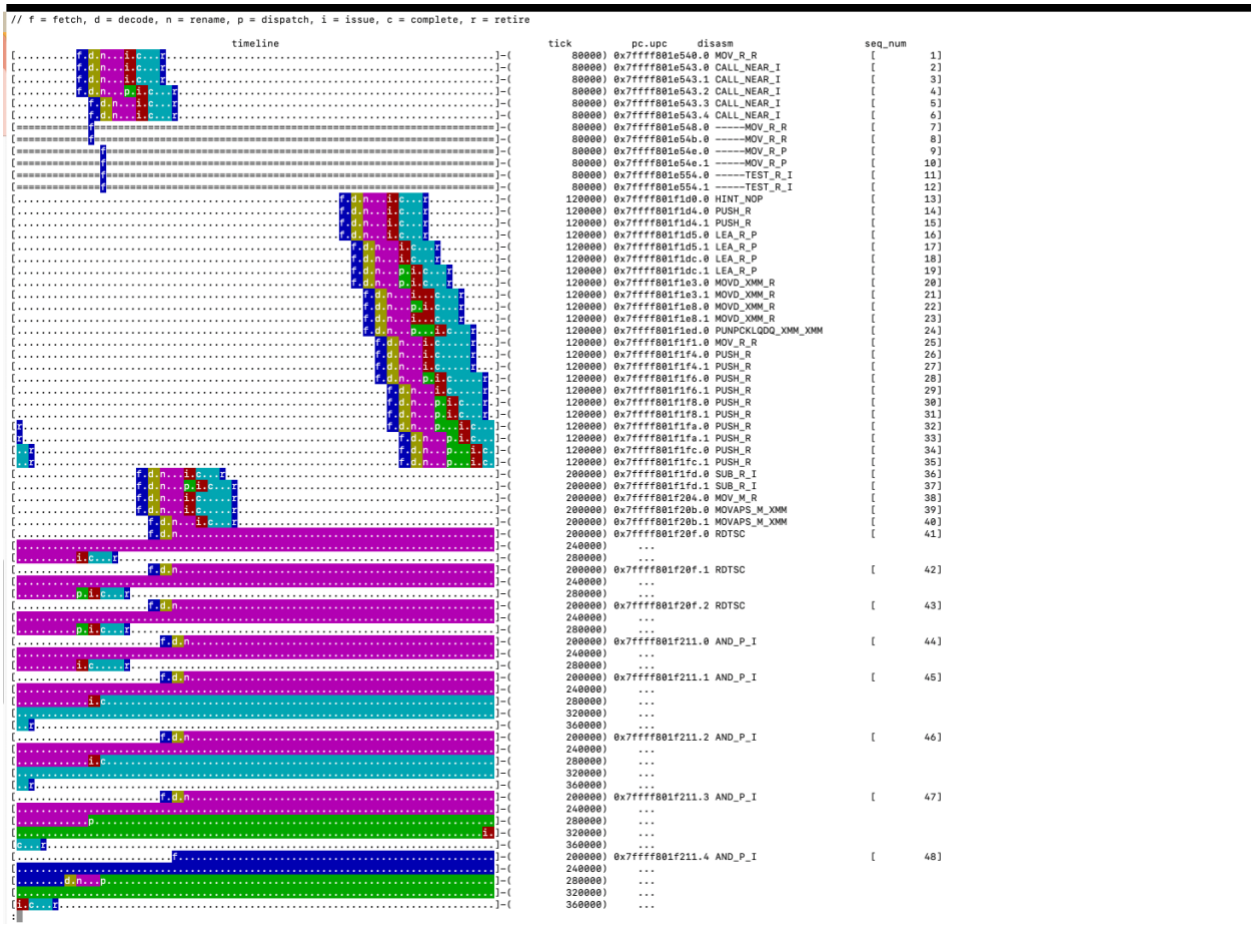
Run Simulation: I have created a simple binary hello program and ran the script as follows

../../build/x86/gem5.opt --debug-flags=O3PipeView --debug-file=trace.out out_of_order_cpu.py -c hello



Visualize:

I have used gem5 graphical pipeline viewer tool from utils class to visualize the output as follows



Cpu Instructions Per cycle can be visualized as

```
Hello MilanExiting @ tick 159645000 because exiting with last active thread context
I have no name!@950eff622438:/gem5/gem5/milan/assignment4$ cat m5out/stats.txt

---------- Begin Simulation Statistics ----------
simSeconds                                   0.000160                       # Number of seconds simulated (Second)
simTicks                                   159645000                       # Number of ticks simulated (Tick)
finalTick                                  159645000                       # Number of ticks from beginning of simulation (restored from checkpoints and never reset) (Tick)
simFreq                                 1000000000000                       # The number of ticks per simulated second ((Tick/Second))
hostSeconds                                    98.50                       # Real time elapsed on the host (Second)
hostTickRate                                 1620786                       # The number of ticks simulated per host second (ticks/s) ((Tick/Second))
hostMemory                                   8877660                       # Number of bytes of host memory used (Byte)
simInsts                                       96840                       # Number of instructions simulated (Count)
simOps                                        186471                       # Number of ops (including micro ops) simulated (Count)
hostInstRate                                     983                       # Simulator instruction rate (inst/s) ((Count/Second))
hostOpRate                                      1893                       # Simulator op (including micro ops) rate (op/s) ((Count/Second))
system.clk_domain.clock                         1000                       # Clock period in ticks (Tick)
system.clk_domain.voltage_domain.voltage           1                       # Voltage in Volts (Volt)
system.cpu.numCycles                           159646                       # Number of cpu cycles simulated (Cycle)
system.cpu.cpi                                1.648554                       # CPI: cycles per instruction (core level) ((Cycle/Count))
system.cpu.ipc                                0.606592                       # IPC: instructions per cycle (core level) ((Count/Cycle))
system.cpu.numWorkItemsStarted                     0                       # Number of work items this cpu started (Count)
system.cpu.numWorkItemsCompleted                   0                       # Number of work items this cpu completed (Count)
system.cpu.instsAdded                         245212                       # Number of instructions added to the IQ (excludes non-spec) (Count)
system.cpu.nonSpecInstsAdded                      68                       # Number of non-speculative instructions added to the IQ (Count)
system.cpu.instsIssued                        229073                       # Number of instructions issued (Count)
system.cpu.squashedInstsIssued                   932                       # Number of squashed instructions issued (Count)
system.cpu.squashedInstsExamined               58803                       # Number of squashed instructions iterated over during squash; mainly for profiling (Count)
system.cpu.squashedOperandsExamined            92622                       # Number of squashed operands that are examined and possibly removed from graph (Count)
system.cpu.squashedNonSpecRemoved                 32                       # Number of squashed non-spec instructions that were removed (Count)
system.cpu.numIssuedDist::samples             135035                       # Number of insts issued each cycle (Count)
system.cpu.numIssuedDist::mean                1.696397                       # Number of insts issued each cycle (Count)
system.cpu.numIssuedDist::stdev               1.614741                       # Number of insts issued each cycle (Count)
system.cpu.numIssuedDist::underflows               0      0.00%      0.00% # Number of insts issued each cycle (Count)
```

I have created a custom script to read stats.txt to analyze the metrics as

t_of_order_cpu.py 9+, U | analyze_stats.py U | hello.c .../assignment3 U | hello.c .../assignment4

gem5 > milan > assignment4 > analyze_stats.py > ...

```python
# File path to stats.txt
stats_file = "m5out/stats.txt"

ipc = 0
total_cycles = 0
total_insts = 0

with open(stats_file, "r") as file:
    for line in file:
        # Find the IPC
        if "system.cpu.ipc" in line:
            ipc = float(line.split()[1])
            print(f"Instruction  Per Cycle (IPC): {ipc}")

        # Find the total cycles
        if "system.cpu.numCycles" in line:
            total_cycles = int(line.split()[1])
            print(f"Number of Cycles: {total_cycles}")
        # Find the total instructions
        if "system.cpu.instsIssued" in line:
            total_insts = int(line.split()[1])
            print(f"Total Number of Instructions {total_insts}")

# Calculate latency in cycles
if total_insts > 0:
    latency = total_cycles / total_insts
else:
    latency = None

# Output results
print(f"Instruction Throughput (IPC): {ipc}")
print(f"Instruction Latency (Cycles per Instruction): {latency}")

```

gem5 — docker run --name gem5-container --platform linux/arm64 -u 501:20 --volume ~/Documents

system.cpu.ipc

```
I have no name!@950eff622438:/gem5/gem5/milan/assignment4$ python3 analyze_stats.py
Number of Cycles: 159646
Instruction  Per Cycle (IPC): 0.606592
Total Number of Instructions 229073
Instruction Throughput (IPC): 0.606592
Instruction Latency (Cycles per Instruction): 0.6969219419137131
I have no name!@950eff622438:/gem5/gem5/milan/assignment4$
```

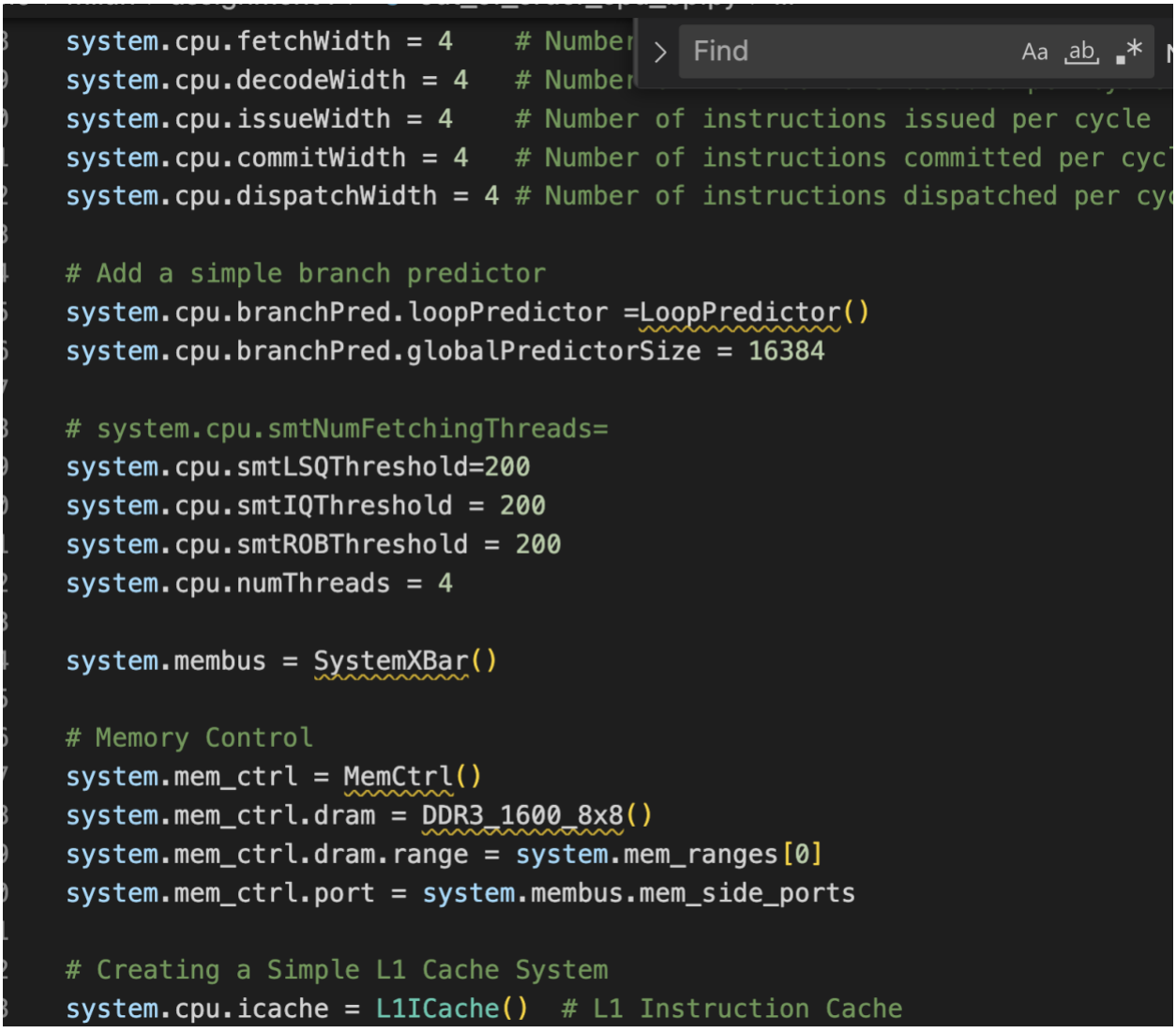
Impact of Branch Prediction

Now I am going to add a branch predictore to the above configuration

```
28  # Set the block size (cache line size) for the entire system
29  system.cache_line_size = 128
30
31  # Simple CPU and Memory Config
32  system.mem_mode = 'timing'
33  system.mem_ranges = [AddrRange('8192MB')]
34  system.cpu = DerivO3CPU()  # Out-of-order CPU model
35  system.cpu.numThreads = 1
36
37  # Configure ILP parameters for DerivO3CPU
38  system.cpu.fetchWidth = 4    # Number of instructions fetched per cyc
39  system.cpu.decodeWidth = 4   # Number of instructions decoded per cyc
40  system.cpu.issueWidth = 4    # Number of instructions issued per cycl
41  system.cpu.commitWidth = 4   # Number of instructions committed per cy
42  system.cpu.dispatchWidth = 4 # Number of instructions dispatched per
43
44  # Add a simple branch predictor
45  system.cpu.branchPred.loopPredictor =LoopPredictor()
46  system.cpu.branchPred.globalPredictorSize = 16384
47
48  system.membus = SystemXBar()
49
50  # Memory Control
51  system.mem_ctrl = MemCtrl()
52  system.mem_ctrl.dram = DDR3_1600_8x8()
53  system.mem_ctrl.dram.range = system.mem_ranges[0]
54  system.mem_ctrl.port = system.membus.mem_side_ports
55
56  # Creating a Simple L1 Cache System
57  system.cpu.icache = L1ICache()  # L1 Instruction Cache
58  system.cpu.dcache = L1DCache()  # L1 Data Cache
59
```

We implemented the DerivO3CPU model in our gem5 simulation, which provided notable performance improvements. By leveraging its out-of-order execution capabilities, we observed enhanced instruction throughput and better utilization of resources, allowing for higher instruction-level parallelism (ILP) compared to simpler CPU models. The advanced features of DerivO3CPU, such as dynamic scheduling and branch prediction, contributed significantly to reducing execution time and improving overall simulation efficiency.

Performance can be improved by using Simultaneous Multithreading (SMT), which enables multiple threads to share the same processor resources. With SMT, the CPU can execute instructions from different threads concurrently, maximizing the utilization of functional units, registers, and pipeline stages. This approach reduces idle cycles and increases instruction throughput, allowing the processor to complete more instructions per cycle. As a result, SMT enhances overall system efficiency, particularly in workloads with parallel threads, by minimizing resource bottlenecks and improving execution speed.



Throughout these experiments, it's essential to consider how different Instruction-Level Parallelism (ILP) techniques, such as pipelining, out-of-order execution, and branch prediction, interact to improve performance. These techniques often complement each other by allowing

multiple instructions to be executed simultaneously and addressing data dependencies more efficiently. However, limitations arise due to factors like instruction dependencies, pipeline hazards, and resource contention, which can restrict the degree of achievable parallelism. Balancing the complexity of ILP techniques with performance benefits requires evaluating each technique's impact on execution time and hardware resources. Achieving optimal ILP often means selectively implementing techniques that maximize throughput without excessively increasing power consumption or design complexity.

References

- Hennessy, J. L., & Patterson, D. A. (2017). *Computer architecture: A quantitative approach* (6th ed.). Morgan Kaufmann.
- Stallings, W. (2021). *Computer organization and architecture* (11th ed.). Pearson.
- Rabaey, J. M., Chandrakasan, A. P., & Nikolic, B. (2022). *Digital integrated circuits: A design perspective* (3rd ed.). Pearson.