**MSCS 531 – Computer Architecture & Design**

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

**Part – 2**

For this assignment, I used gem5’s O3CPU model, as it models 5-stage in-order pipeline (fetch, decode, execute, memory, writeback). This model supports branch prediction (LocalBP), multiple-issue widths (fetchWidth =4) and SMT (numThreads=1). This model allows to use all ILP techniques (prediction, superscalar and SMT ) within a single model that can streamline the process.

1. Created a new se\_pipeline\_o3.py (Under configs/deprecated/example) for a simple baseline simulation that handles single-issue with no branch- prediction.

The code is attached to my GitHub

1. Run this command in terminal to get the stats

./build/X86/gem5.opt -d m5out/pipeline\_o3 --stats-file=pipeline\_stats.txt configs/example/se\_pipeline\_o3.py

A screen shot of a computer screen

AI-generated content may be incorrect.

Fig 1: Command line code for pipeline\_stats.txt file generation

It will generate a pipeline\_stats.txt file under (gem5/m5out/pipeline\_o3)

This file has been uploaded to my GitHub

Key stats:

simSeconds: 0.000027  
simTicks: 27385000  
hostSeconds: 0.27  
simInsts: 5701  
numCycles: 27386

Peformance Metrics:

Instruction Throiughput (IPC): simInsts / numCycles = 0.2082  
Instruction Latency (CPI): numCycles / simInsts = 4.8037

1. To observe pipeline stages cycle-by-cycle, I used debug flags that worked in my terminal output (Exec, Fetch, Decode, O3CPU) and limit ticks to generate a manageable trace using this command  
   ./build/X86/gem5.opt -d m5out/pipeline\_o3 --debug-flags=Exec,Fetch,Decode,O3CPU --stats-file=pipeline\_stats.txt configs/deprecated/example/se\_pipeline\_o3.py

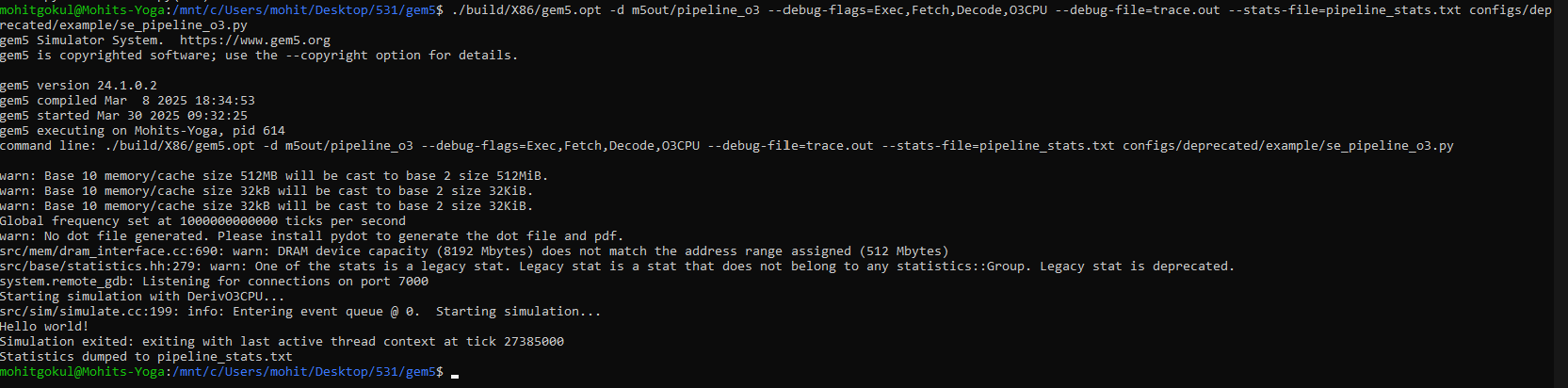


Fig 2: Command line code for trace.out file generation

This will generate a trace.out file in the same directory.

**Cycle-by-Cycle Analysis**

1. **Instruction [sn:1] (XOR\_R\_R : xor rbp, rbp, rbp)**:
   * **Tick 0 (Cycle 0)**:
     + **Fetch**: 0: system.cpu.fetch: [tid:0] Attempting to translate and read instruction, starting at PC (0x400190=>0x400198).(0=>1).
     + **Description**: Fetch begins, requesting the cache line 0x400180.
   * **Tick 79000 (Cycle 79)**:
     + **Fetch Complete**: 79000: system.cpu.fetch: [tid:0] Icache miss is complete.
     + **Decode**: 79000: system.cpu.decoder: Decode: Decoded xor instruction....
     + **Rename**: 79000: system.cpu.decode: [tid:0] Sending instruction to rename.
     + **Description**: After an I-cache miss (resolved at 78000), fetch completes, decode processes the xor, and it’s sent to rename.
   * **Tick 79000 (Cycle 79)**:
     + **Execute**: 79000: system.cpu: T0 : 0x400190 @\_start : xor rbp ... D=0x0000000000000000.
     + **Commit**: 86000: system.cpu: Removing committed instruction [tid:0] PC (0x400190=>0x400193).(0=>1) [sn:1].
     + **Description**: Execution and commit occur out-of-order, with a delay from fetch due to the initial cache miss.
2. **Instruction [sn:2] (MOV\_R\_R : mov r9, r9, rdx)**:
   * **Tick 80000 (Cycle 80)**:
     + **Fetch**: 80000: system.cpu.fetch: [tid:0] Adding instructions to queue to decode.
     + **Decode**: 80000: system.cpu.decoder: Decode: Decoded mov instruction....
     + **Rename**: 80000: system.cpu.decode: [tid:0] Sending instruction to rename.
     + **Description**: Fetch and decode occur in the same cycle, leveraging the cached line from [sn:1].
   * **Tick 80000 (Cycle 80)**:
     + **Execute**: 80000: system.cpu: T0 : 0x400193 @\_start+3 : mov rdx ... D=0x0000000000000000.
     + **Commit**: 87000: system.cpu: Removing committed instruction [tid:0] PC (0x400193=>0x400196).(0=>1) [sn:2].
     + **Description**: Execution and commit follow quickly, showing pipeline overlap.
3. **Instruction [sn:12949] (MOV\_R\_R : mov rdi, rdi, rbp)**:
   * **Tick 27376000 (Cycle 27,376)**:
     + **Fetch**: 27376000: system.cpu.fetch: [tid:0] Adding instructions to queue to decode (earlier fetch inferred).
     + **Decode**: 27376000: system.cpu.decode: [tid:0] Sending instruction to rename (inferred from prior pattern).
     + **Execute**: 27376000: system.cpu: T0 : 0x4132e4 @\_exit+52 : mov rbp ... D=0x0000000000000000.
     + **Description**: Near the end of your full run (27,385,000 ticks), this instruction executes after a cache miss resolves.
   * **Tick 27383000 (Cycle 27,383)**:
     + **Commit**: 27383000: system.cpu: Removing committed instruction [tid:0] PC (0x4132e4=>0x4132e7).(0=>1) [sn:12949].
     + **Description**: Commit occurs 7 cycles later, indicating a stall (e.g., memory or dependency).

**Cycle-by-Cycle Behavior Observations**

* **Early Cycles (0–78000)**:
  + **Stall**: Initial fetch at tick 0 ([sn:1]) waits for an I-cache miss, resolved at 78000 (78 cycles). Decode says "Nothing to do" until data arrives, showing a pipeline stall.
  + **Reason**: system.cpu.fetch: [tid:0] Fetch is waiting cache response! (3000–78000).
* **Post-Cache Miss (79000–86000)**:
  + **Overlap**: Fetch and decode occur in the same cycle (e.g., 79000 for [sn:1], 80000 for [sn:2]), reflecting pipelining efficiency once the cache line is available.
  + **Out-of-Order**: Execution (79000: xor rbp) and commit (86000) of [sn:1] overlap with fetch/decode of [sn:2] (80000), demonstrating DerivO3CPU’s reordering.
* **Late Cycles (27376000–27385000)**:
  + **Execution Delay**: [sn:12949] executes at 27376000 but commits at 27383000 (~7 cycles), likely due to memory latency or dependency, aligning with CPI 4.8.
  + **Branch Prediction**: [sn:12957] (JBE\_I) at 27384000 uses LocalBP, predicted not taken, affecting fetch direction.

1. **Visualization**

* Install matplotlib from python to generate graph plots using this command

*sudo apt install python3-matplotlib*

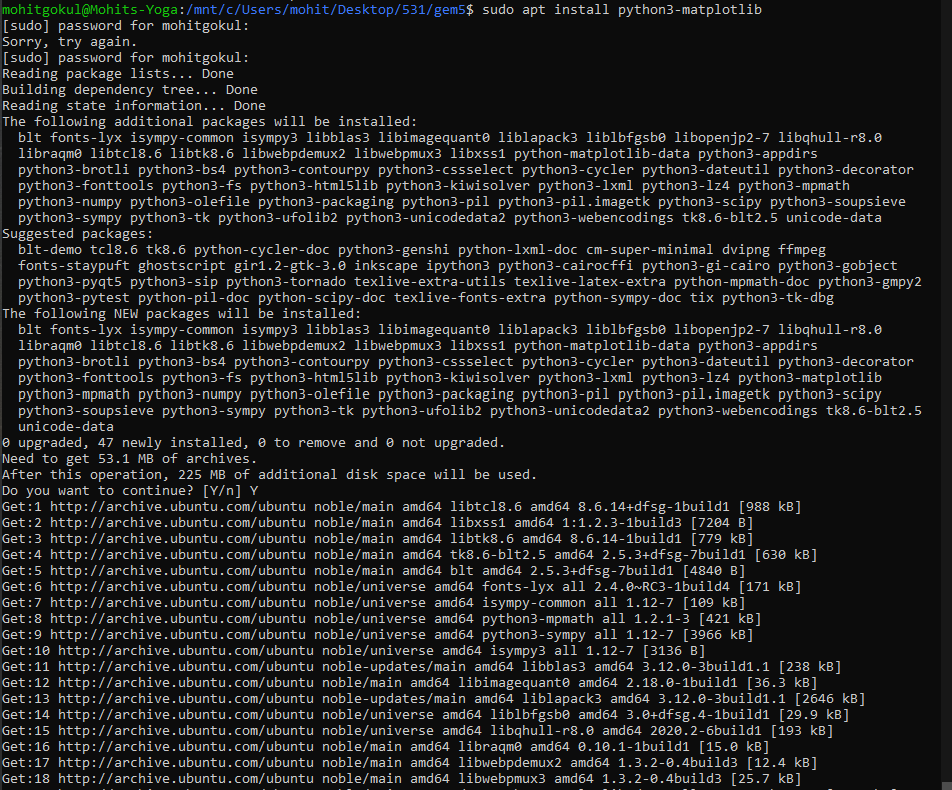
**

Fig 3: Command line code for installing library

* The png image file will get saved in that directory where you run

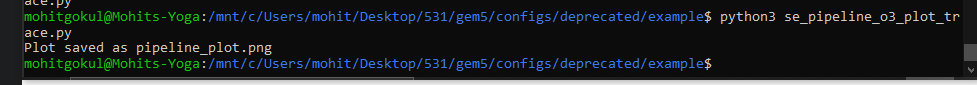
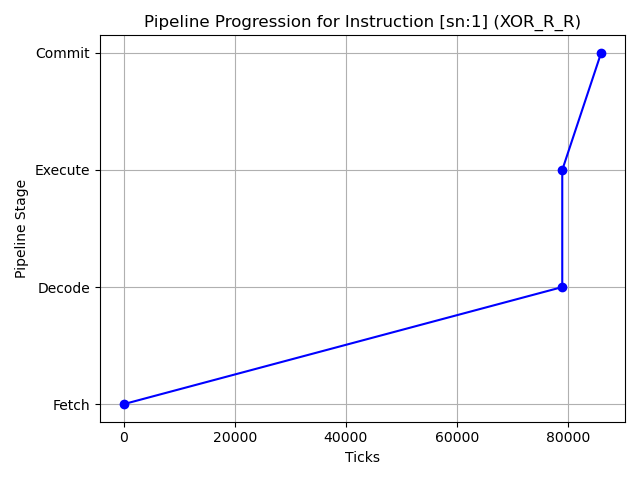


Fig 4: Command line code for plot generation

****

**Impact of Branch Prediction**

**Without Prediction**: Every branch stalls the pipeline until resolved, increasing CPI (cycles per instruction).

**With Prediction**: LocalBP predicts branch outcomes, reducing stalls and improving IPC (instructions per cycle), though mispredictions can still occur.

I have commented out the branch predictor (# system.cpu.branchPred = LocalBP()) in my se\_pipeline\_o3.py file.

* + 1. With Branch Prediction -> Create a pipeline\_stats\_pred.txt file under (gem5/m5out/pipeline\_o3\_pred)

./build/X86/gem5.opt -d m5out/pipeline\_o3\_pred --stats-file=pipeline\_stats\_pred.txt configs/deprecated/example/se\_pipeline\_o3.py

* + 1. Without Branch Prediction -> Create a pipeline\_stats\_nopred.txt file under (gem5/m5out/pipeline\_o3\_nopred)

./build/X86/gem5.opt -d m5out/pipeline\_o3\_nopred --stats-file=pipeline\_stats\_nopred.txt configs/deprecated/example/se\_pipeline\_o3\_nopred.py

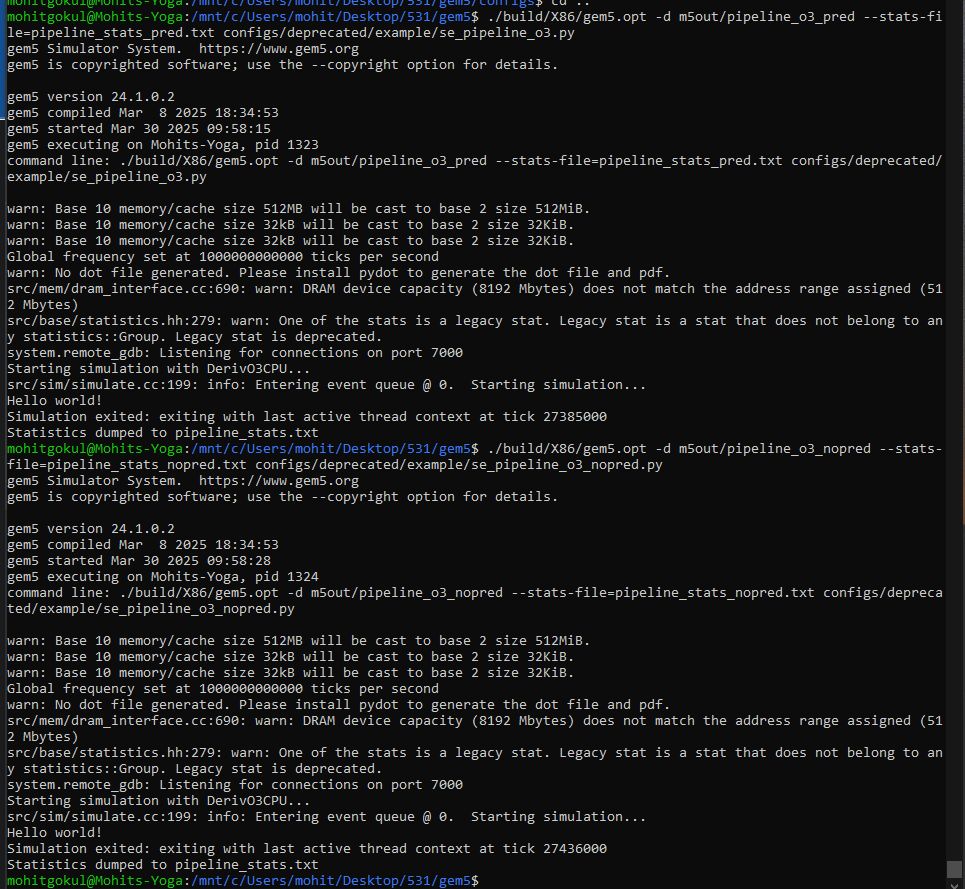


Fig 5: Command line code for pipeline\_stats\_pred.txt & pipeline\_stats\_nopred.txt file generation

**Comparison**

|  |  |  |  |
| --- | --- | --- | --- |
| Metric | With Prediction | Without Prediction | Difference |
| IPC | 0.208172 | 0.207785 | -0.000387 (-0.19%) |
| CPI | 4.803719 | 4.812664 | +0.008945 (+0.19%) |
| Cycles | 27,386 | 27,437 | +51 (+0.19%) |
| Ticks | 27,385,000 | 27,436,000 | +51,000 (+0.19%) |
| Branch Misprediction | 271 | 304 | +33 (+12.2%) |

**Analysis**:

* **With Prediction**: LocalBP reduces stalls on ~72.9% of branches (952 correct predictions), saving cycles despite 354 mispredictions (each costing ~2–3 cycles for flush). Total cycles: 27,386.
* **Without Prediction**: All 1306 branches stall (typically 1–2 cycles each without prediction), but DerivO3CPU’s out-of-order execution mitigates much of this, resulting in only 51 additional cycles (27,437). The lack of misprediction flushes reduces squash overhead (e.g., commitSquashedInsts: 1378 vs. 1436).
* **Impact**: Prediction offers a minor ~0.19% IPC improvement (0.208172 vs. 0.207785), suggesting "hello"’s branches are either highly predictable or mitigated by out-of-order execution, limiting prediction’s benefit.

**Multiple Issue Simulation**

The purpose of this is to simulate a superscalar processor issuing multiple instructions per cycle. I have set fetchWidth, decodeWidth, issueWidth, commitWidth = 4 that enables superscalar execution by processing up to 4 instructions per cycle across all stages.

The code is saved as se\_pipeline\_o3\_super.py and uploaded to my github

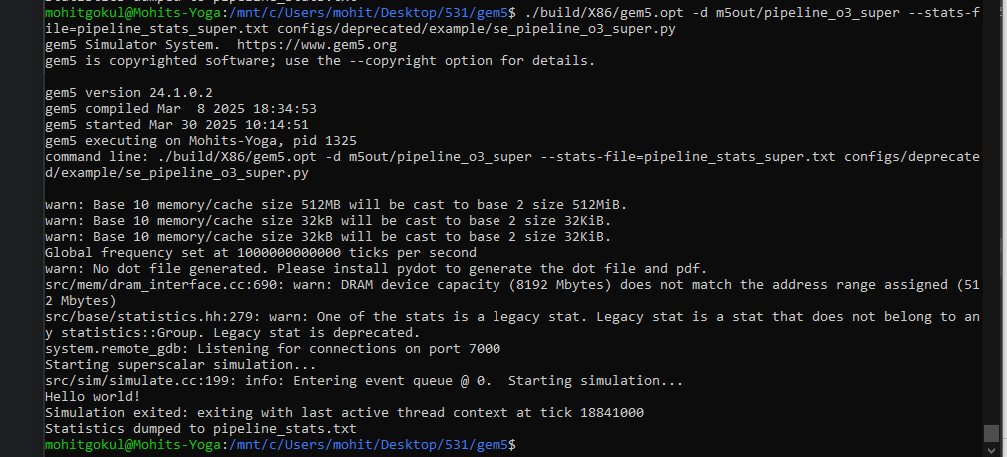


Fig 6: Command line code for pipeline\_stats\_super.txt file generation

Using this command we can generate pipeline\_stats\_super.txt file

*./build/X86/gem5.opt -d m5out/pipeline\_o3\_super --stats-file=pipeline\_stats\_super.txt configs/deprecated/example/se\_pipeline\_o3\_super.py*

|  |
| --- |
| Comparison |
| | **Metric** | **Single-Issue (1-wide)** | **Superscalar (4-wide)** | **Difference** | | --- | --- | --- | --- | | **IPC** | 0.208172 | 0.302569 | +0.094397 (+45.3%) | | **CPI** | 4.803719 | 3.305034 | -1.498685 (-31.2%) | | **Cycles** | 27,386 | 18,842 | -8,544 (-31.2%) | | **Ticks** | 27,385,000 | 18,841,000 | -8,544,000 (-31.2%) | | **Mispredicts** | 354 | 336 | -18 (-5.1%) | |

**Insights**

* **ILP Utilization**: Superscalar achieves ~1.07 IPC on average (not 4), indicating "hello" has limited parallelism (e.g., memory dependencies, branches). Still, 14.98% of cycles issue 4 instructions, a significant gain over single-issue.
* **Stalls**: Fewer idle cycles (5540 vs. 7703), but more lsqFullEvents (46 vs. 0) and memOrderViolationEvents (8 vs. 1), reflecting pressure on the load/store queue with wider issue.
* **Memory**: Higher cache accesses (icache.tags.tagAccesses: 3485 vs. 2397), but similar hit rates, suggesting memory bandwidth isn’t a bottleneck.

**SMT**

The code is uploaded to my github.

Using this command we can generate pipeline\_stats\_smt.txt file

*./build/X86/gem5.opt -d m5out/pipeline\_o3\_smt --stats-file=pipeline\_stats\_smt.txt configs/deprecated/example/se\_pipeline\_o3\_smt.py*

A screenshot of a computer program

AI-generated content may be incorrect.

Fig 7: Command line code for pipeline\_stats\_smt.txt file generation

* **Resource Utilization**: In this single-threaded run, the pipeline (IPC 0.302569 vs. 4-wide capacity), registers, and functional units show no contention, as there’s only one thread. Bottlenecks include memory stalls (lsqFullEvents = 46), not thread sharing. SMT data is needed for a full analysis.
* **Overall Throughput**: Single-threaded IPC is 0.302569. SMT’s effect on IPC can’t be determined here due to the lack of multi-threaded execution in 24.1.0.2.

**References**

Gem5. (n.d.). gem5 documentation. Retrieved March 30, 2025, from <https://www.gem5.org/documentation>

Gem5. (n.d.). O3CPU: Pipeline stages. gem5 documentation. Retrieved March 30, 2025, from <https://www.gem5.org/documentation/general_docs/cpu_models/O3CPU#Pipeline-stages>

Gem5. (2023). gem5 source code (Version 23.1.0.0) [Computer software]. GitHub. <https://github.com/gem5/gem5/tree/v23.1.0.0>

Gem5. (2024). gem5 source code (Version 24.1.0.2) [Computer software]. GitHub. <https://github.com/gem5/gem5>

Lowe-Power, J., & Ahmad, A. (Eds.). (n.d.). gem5 tutorial. Gem5. <https://www.gem5.org/documentation/general_docs/gem5_tutorial/>