Out-of-Order CPU Pipeline Simulator Report

Xin Jin

August 27, 2025

1 Background Introduction

This project implements a C++17-based CPU simulator that models the internal operations of an out-of-order execution pipeline. It demonstrates modern CPU architecture concepts such as register renaming, Tomasulo's algorithm, branch prediction, and reservation stations. The simulator allows step-by-step analysis of instruction flow, with detailed debug logging of each pipeline stage per cycle.

2 System Design

The simulator models a 6-stage CPU pipeline:

- **Fetch:** Reads instructions from memory, uses BTB and branch prediction.
- **Decode & Rename:** Performs instruction decoding and register renaming.
- **Issue:** Allocates instructions into reservation stations and ROB.
- **Execute:** Executes instructions using appropriate functional units.
- **Write Back:** Broadcasts execution results and updates register files.
- **Commit:** Commits results in program order using the ROB.

The simulator supports instructions: fld, fsd, add, addi, slt, fadd, fsub, fmul, fdiv, bne. It supports multiple functional units (INT, FPADD, FPMUL, LOAD, STORE, BRANCH) and maintains global state for memory, registers, and pipeline buffers.

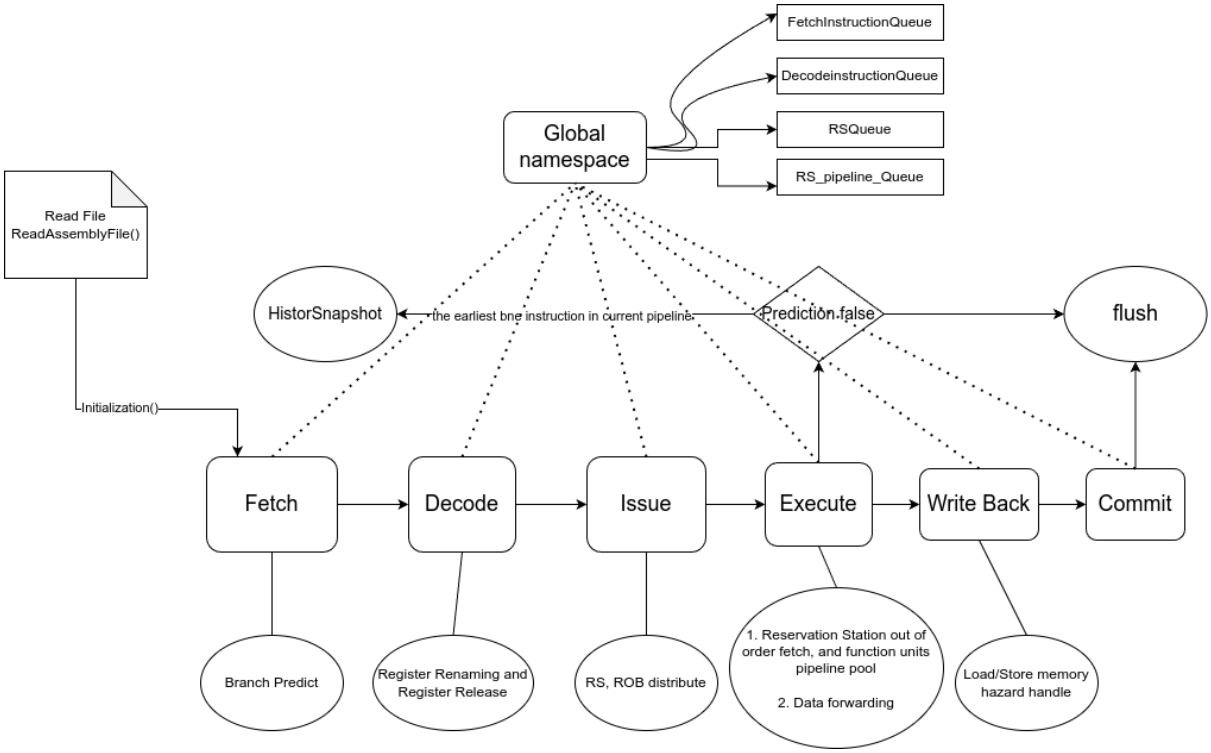

Figure 1: System design Diagram

2.1 Run instruction

- Enter Simulator root directory
- command line: make
- command line: ./myapp

The assembly code awaiting to be read is at assembaly.dat. You can also use command: ./myapp -p "XXX.dat" to change the file to read.

And the operations for binary runnable ./myapp is as follows:

- -f, NF,
- -i, NI,
- -w, NW,
- -b, NB,
- -r, NR

You can also manually change the above parameters in global.h. Read parameters as input:

```
1    $ ./myapp -p assembly.dat -f 2 -i 16 -w 4 -r 16 -b 4 > record.txt 2>&1
```

Listing 1: Read hyper parameters from input

3 Implementation Details

The project is modularly structured with classes and components such as:

- `Instruction`, `ROBEntry`, `ReservationStationEntry`, and `RegisterRenaming`.
- `ID_in_Queue` denotes the execution order of a instruction, is globally unique to maintain. I also maintain a global `InstructionQueue` to record all the instruction executing in order, each element in the `InstructionQueue` is globally unique.
- `Global` namespace: holds shared structures like queues, ROB, rename map, BTB, memory.
- Tomasulo's algorithm is implemented using physical register readiness tracking.
- Branch prediction uses 2-bit saturating counters with BTB.
- Mis-speculation recovery via register rename state snapshots.

Each pipeline stage (fetch, decode, etc.) is implemented in separate classes and executed sequentially per cycle.

3.1 Global namespace

All global variables are defined in global.h, to avoid complicated paratemer transfer:

```
1
2
3  namespace Global {
4      extern int current_cycle;
5      extern unordered_map<string, size_t> labelMap;
6      extern unordered_map<int, double> memory_value;
7      extern RegisterRenaming renaming_worker;
8      extern vector<Instruction> instructions;
9      extern deque<Instruction> instructionset;
10     //record the execution order
11     extern deque<Instruction> instructionQueue;//record the instruction status
12     extern deque<Instruction> fetchInstructionQueue;
13     extern int fetch_pointer;
14     extern BTB btb;
15     extern HistorySnapshot historySnapshot;
16     extern unordered_map<string, ArchitectureRegister> architectureRegisterFile
           ;
17     //decode
18     extern deque<Instruction> decodeInstructionQueue;
19     extern int renameStall;
20     //dependency relationship analysis, not used for result
21     extern map<int,DependencyList> dependency_map;
22
23     //ReservationStation
24     extern deque<ReservationStationEntry> RS_INT_Queue;
25     extern deque<ReservationStationEntry> RS_LOAD_Queue;
26     extern deque<ReservationStationEntry> RS_STORE_Queue;
27     extern deque<ReservationStationEntry> RS_FPadd_Queue;
28     extern deque<ReservationStationEntry> RS_FPmult_Queue;
29     extern deque<ReservationStationEntry> RS_FPdiv_Queue;
30     extern deque<ReservationStationEntry> RS_BU_Queue;
31
32     extern int rsFullNumber;//RS full stalls number
33     extern int stallCount_RSFull;
34     //ROB
35     extern int stallCount_ROBFull;
```

```
    extern int robHead;
    extern int robTail;
    extern vector<ROBEntry> ROBuffer;

    //unit pipeline
    extern deque<PipelineStage> INT_pipeline;
    extern deque<PipelineStage> LOAD_pipeline;
    extern deque<PipelineStage> STORE_pipeline;
    extern deque<PipelineStage> FPadd_pipeline;
    extern deque<PipelineStage> FPmult_pipeline;
    extern deque<PipelineStage> FPdiv_pipeline;
    extern deque<PipelineStage> BU_pipeline;
    //The instruction that have completed execution
    extern vector<ReservationStationEntry> completeRSQueue;
    extern vector<ReservationStationEntry> BUQueue;
    extern vector<LoadResult> LoadQueue;
    extern vector<PendingLoad> LoadHazardQueue;
    extern vector<ReadyStore> StoreQueue;
}
```

Listing 2: Global Name Space

Program Termination Detection

In a cycle-accurate out-of-order pipeline simulator, one important design consideration is when to stop the simulation. A naive approach would be to stop when all queues (fetch, decode, ROB) appear empty. However, this can lead to incorrect early termination.

For example, at the start of the simulation, all pipeline structures (ROB, reservation stations, etc.) are indeed empty — but the program hasn't started yet. Relying on emptiness alone could result in the simulator exiting before anything is fetched.

To address this, we introduce a flag `programStarted`, which becomes `true` once any instruction is in-flight (e.g., fetch or decode queue is non-empty, or ROB has active entries). After that point, we only allow the simulator to terminate if all relevant queues and buffers are truly empty.

We use `Global::robHead == Global::robTail+1` as a reliable indicator that the Reorder Buffer is empty. This is because ROB is implemented as a circular buffer: when head equals tail, the buffer is logically empty — regardless of the content.

A robust termination condition is shown below:

```
    while (
    !Global::fetchInstructionQueue.empty() ||
    !Global::decodeInstructionQueue.empty() ||
    Global::robHead != Global::robTail+1 ||
    !Global::RS_INT_Queue.empty() ||
    !Global::RS_FPadd_Queue.empty() ||
    !Global::RS_FPmult_Queue.empty() ||
    !Global::RS_LOAD_Queue.empty() ||
    !Global::RS_STORE_Queue.empty() ||
    !Global::RS_BU_Queue.empty()|| global_cycle<10
    ){
                pipelineGlobalCycle();
                debugLogger();
    }
```

Listing 3: Simulation loop with safe termination condition

This approach guarantees correct completion: the simulation continues until all stages have drained — not just fetched nothing. It also avoids premature exit due to initial emptiness.

3.2 ROB and Reservation Station Stalls Count Method

At issue.h file:

```
        if (rsStallThisCycle || robStallThisCycle) {
            cout<<" full ----------------RS ROB--------"<<endl;
            if (rsStallThisCycle) Global::stallCount_RSFull++;
            if (robStallThisCycle) Global::stallCount_ROBFull++;
            break; // Insufficient resources in the current cycle, exiting the
                issue loop
        }
```

Listing 4: ROB and RS Stalls Count function

3.3 Debug Logger

```

void Simulator::debugLogger()
```

Listing 5: Debug Logger

The simulator outputs detailed logs per cycle showing:

- ROB contents and instruction status
- Reservation Station usage
- Fetch, Decode, Execute queues
- Physical register readiness and values

3.4 Branch Target Buffer and Predictor Design

The simulator includes a 16-entry Branch Target Buffer (BTB), indexed using bits [7:4] of the branch instruction's program counter (PC), following the project specification.

Each BTB entry stores:

- The target address of a previously seen branch instruction;
- An independent 2-bit dynamic branch predictor (i.e., a local predictor);

The branch predictor at each entry maintains one of four states:

- Strongly Taken (11), Weakly Taken (10), Weakly Not Taken (01), Strongly Not Taken (00)

and transitions between states based on actual branch outcomes using a 2-bit saturating counter scheme.

```
    class BTB{
    private:
        // const size_t capacity=MAX_BTB_SIZE;
        bool BTB_HIT[MAX_BTB_SIZE]={1};
    public:
       unordered_map<int,tuple<int, int, BranchPredict>> btbMap;
       BTB();
        ~BTB();
        int getTargetPosition(int instructionNumber);
        void update(int instructionNumber,bool taken);
        bool getPrediction(int instructionNumber);
};
```

```
13
14 class BranchPredict{
15     private: BranchPredictionStage stage=PREDICT_WEAK_TAKEN;
16     public:
17     BranchPredict(){
18         cout<<"BranchPredictor initialized"<<endl;
19     }
20     bool predict();
21     //update stage based on finally taken or not taken
22     void update(bool finalTaken);
23 };
```

Listing 6: BTB data structure

On instruction fetch, if a branch maps to a BTB entry, the corresponding predictor is queried to decide whether to predict "taken" or "not taken". If "taken", the BTB provides the predicted target address, and the fetch unit fetches from that target speculatively.

During execution or commit, if the branch outcome disagrees with the prediction, the predictor at the BTB entry is updated, and recovery may be triggered as described in the misprediction handling mechanism.

This per-entry predictor design allows learning per-branch behavior independently, improving prediction accuracy over a single global predictor.

3.5 Branch Misprediction Recovery Design

This simulator supports a dual-stage mechanism for handling branch misprediction, combining early recovery with conservative fallback to commit.

1. Early Recovery at Execute Stage:

When a `bne` instruction executes and detects that the prediction was incorrect, the simulator first checks whether this branch is the earliest active branch in the pipeline (i.e., the youngest instruction with branch behavior so far). If confirmed, it immediately triggers recovery via:

- Restoring register renaming and physical register state from the saved `HistorySnapshot`;
- Flushing all pipeline queues and reservation stations;
- Resetting ROB state;
- Redirecting the PC to the correct target;

This early recovery path minimizes the number of wasted cycles after a misprediction, but requires confidence that no older branch could interfere.

```
1  bool Execute::executeBU(int earliest_ID_in_Queue){
2      //It still starts with begin. According to the order of reading the pointer
           before, begin is the earliest. If begin is the smallest and true, then
          earliest_ID_in_Queue becomes the second smallest ID_in_Queue in RS BU
3      for(auto entry=Global::BU_pipeline.begin();entry!=Global::BU_pipeline.end()
           ;){
4          if(Global::BU_pipeline.empty()) {cout<<"No instruction in BU_pipeline"
               <<endl; break;}
5          entry->remaining_latency--;
6          if(entry->remaining_latency==0){
7              bool result=false;
8              //Operation completed
9              //Execute instruction
10             cout<<"R1 is: "<<entry->rs_entry.Vj<<endl;
11             if(entry->rs_entry.opcode==bne)result=(entry->rs_entry.Vj!=entry->
                   rs_entry.Vk);
12             else{
```

```
                throw runtime_error("opcode is not BU");
            }
            entry->rs_entry.result=result;
            //to get the snapshot
            Snapshot* snapshot=Global::historySnapshot.findMatchingSnapshot(
                entry->rs_entry.ID_in_Queue);
            //Get the previous branch prediction result in BTB
            bool predictTaken=snapshot->bne_instruction.bne_taken.value();
            bool predictTrueFalse=(predictTaken==result);
            //If the ID is the largest and the prediction is correct, the
                earliest_ID_in_Queue becomes the second smallest ID_in_Queue in
                the RS BU
            //The prediction value does not only come from the branch predictor
                . There is also the value set after rollback
            if(earliest_ID_in_Queue==entry->rs_entry.ID_in_Queue &&
                predictTrueFalse){
                earliest_ID_in_Queue=getEarliestIDIn_RS_BU_Queue();
                //Update predictor status
                entry->rs_entry.result = result;
                Global::btb.update(entry->rs_entry.ID_in_Queue,result);
                //Write into completeRSqueue, indicating the instruction of the
                     completed calculation
                entry->rs_entry.predictTrueFalse=predictTrueFalse;
                // entry->rs_entry.destPhysicalRegister=entry->rs_entry.
                    destPhysicalRegister;
                insertCompletedEntry(entry->rs_entry);
                entry = Global::BU_pipeline.erase(entry);
            }else if(earliest_ID_in_Queue==entry->rs_entry.ID_in_Queue && !
                predictTrueFalse){
                //Roll back immediately, and the entire execute pipeline ends.
                entry->rs_entry.result = result;
                entry->rs_entry.predictTrueFalse=predictTrueFalse;
                Global::btb.update(entry->rs_entry.ID_in_Queue,result);
                // entry->rs_entry.destPhysicalRegister=entry->rs_entry.
                    destPhysicalRegister;
                // insertCompletedEntry(entry->rs_entry);
                // entry = Global::BU_pipeline.erase(entry);
                cout<<"historySnapshot.predictionTrueFalseRecover"<<endl;
                Global::historySnapshot.predictionTrueFalseRecover(Global::
                    instructionQueue[entry->rs_entry.ID_in_Queue],predictTaken);
                return false;
            }else{//You cannot roll back directly in Execute. You have to pass
                it to ROB for decision. ROB also needs to make a decision on
                rollback.
                entry->rs_entry.result = result;
                entry->rs_entry.predictTrueFalse=predictTrueFalse;
                entry->rs_entry.destPhysicalRegister=entry->rs_entry.
                    destPhysicalRegister;
                Global::btb.update(entry->rs_entry.ID_in_Queue,result);
                insertBUEntry(entry->rs_entry);
                entry = Global::BU_pipeline.erase(entry);
            }
            //ROB and CDB are updated during the write back phase
        }
        cout<<"The size of BU_pipeline is: "<<Global::BU_pipeline.size()<<endl;
        cout<<"The size of BU_Queue is: "<<Global::BUQueue.size()<<endl;
        entry++;
    }
    return true;
}
```

Listing 7: Execute Branch Unit False Measurement

```
1
2  void HistorySnapshot::predictionTrueFalseRecover(Instruction bne_instruction,
       bool predictTaken) {
3      //This function is called only if the prediction is wrong
4      if(bne_instruction.opcode != InstructionType::bne) {
5          cout<<"false ID_in_Queue: "<<bne_instruction.ID_in_Queue.value()<<endl;
6          throw runtime_error("historySnapshot::predictionTrueFalseRecover()
               called with non-BNE instruction");
7      }
8      Snapshot* snapshot = findMatchingSnapshot(bne_instruction.ID_in_Queue.value
           ());
9
10     //No matching snapshots found
11     if(snapshot == nullptr) {
12         cout<<"false ID_in_Queue: "<<bne_instruction.ID_in_Queue.value()<<endl;
13         cout<<"The history_snapshots size is: "<<history_snapshots.size()<<endl
               ;
14         throw runtime_error("snapshot not found in the history_snapshots");
15     }
16     //1. Restore memory_value
17     //2. Restore register rename
18     //3. Restore fetchInstructionQueue
19     //4. Restore instructionset
20     //5. Restore fetch_pointerb
21     //Get new fetch pointer
22     if(!predictTaken){
23         int target_position=Global::btb.getTargetPosition(bne_instruction.
               instructionNumber);
24         snapshot->bne_instruction.bne_taken=true;
25         Global::fetch_pointer = target_position;
26     }else{
27         Global::fetch_pointer=bne_instruction.instructionNumber+1;
28         snapshot->bne_instruction.bne_taken=false;
29     }
30     Global::instructionQueue=snapshot->instructionQueue;
31     Global::memory_value = snapshot->memory_value;
32     Global::renaming_worker = snapshot->renaming_worker;
33     Global::fetchInstructionQueue = snapshot->fetchInstructionQueue;
34     Global::decodeInstructionQueue=snapshot->decodeInstructionQueue;
35     Global::architectureRegisterFile=snapshot->architectureRegisterFile;
36     Global::renameStall=snapshot->renameStall;
37     //3. Recover ROB
38     Global::ROBuffer=snapshot->ROBuffer;
39     //4. Restore reservation station
40     Global::RS_INT_Queue=snapshot->RS_INT_Queue;
41     Global::RS_LOAD_Queue=snapshot->RS_LOAD_Queue;
42     Global::RS_STORE_Queue=snapshot->RS_STORE_Queue;
43     Global::RS_FPadd_Queue=snapshot->RS_FPadd_Queue;
44     Global::RS_FPmult_Queue=snapshot->RS_FPmult_Queue;
45     Global::RS_FPdiv_Queue=snapshot->RS_FPdiv_Queue;
46     Global::RS_BU_Queue=snapshot->RS_BU_Queue;
47     //5. Recovering the ROB
48     Global::stallCount_ROBFull=snapshot->stallCount_ROBFull;
49     Global::robHead=snapshot->robHead;
50     Global::robTail=snapshot->robTail;
51     Global::ROBuffer=snapshot->ROBuffer;
52     //unit pipeline
53     Global::INT_pipeline=snapshot->INT_pipeline;
54     Global::LOAD_pipeline=snapshot->LOAD_pipeline;
55     Global::STORE_pipeline=snapshot->STORE_pipeline;
56     Global::FPadd_pipeline=snapshot->FPadd_pipeline;
57     Global::FPmult_pipeline=snapshot->FPmult_pipeline;
58     Global::FPdiv_pipeline=snapshot->FPdiv_pipeline;
```

```
59      Global::BU_pipeline=snapshot->BU_pipeline;
60
61      Global::completeRSQueue=snapshot->completeRSQueue;
62      Global::BUQueue=snapshot->BUQueue;
63      Global::LoadQueue=snapshot->LoadQueue;
64      Global::LoadHazardQueue=snapshot->LoadHazardQueue;
65      Global::StoreQueue=snapshot->StoreQueue;
66      //5. Restore decode
67      //6. Restore issue
68      //7. Restore execute
69      //8. Restore write back
70      //9. Restore commit
71      //The snapshot is stored when the pointer points to bne. Restoring the
            snapshot will make the fetch pointer point to bne again. Here, the fetch
             pointer needs to be updated according to the actual branch result
72      //After the rollback is completed, the predictor state will also roll back
            to the original state, so the result of the bne prediction can be known
            based on the original branch predictor state
73      if(!predictTaken) {
74          //Actual taken
75          Global::fetch_pointer = snapshot->btb.getTargetPosition(bne_instruction
                .instructionNumber);//This is still retained, because it is agreed
                to index entry according to address4-7
76      } else {
77          //Actual not taken
78          Global::fetch_pointer++;
79      }
80
81      //Prediction error, the path after the current bne is wrong, the program
            sequence is no longer used, delete all snapshots after the current
            snapshot snap
82      clearHistoryAfter(bne_instruction.ID_in_Queue.value());
83      //      auto it = std::find_if(history_snapshots.begin(), history_snapshots.
            end(),
84      //          [snapshot](const Snapshot& snap) { return &snap == snapshot; });
85      //      if (it != history_snapshots.end()) {
86      //          history_snapshots.erase(it, history_snapshots.end());
87      //      }
88
89    }
90
91  }
```

Listing 8: history Snapshot Recover

2. Conservative Recovery at Commit Stage:

If the branch is not the oldest in the ROB, or the simulator cannot safely determine this in Execute, the branch's actual outcome and prediction status are stored in its ROB entry.

When the branch instruction reaches the head of the ROB and is ready to commit, the simulator then:

- Checks whether the prediction was incorrect;
- If so, flushes the pipeline;
- Restores the state using `HistorySnapshot`;
- Resets the fetch pointer to the correct path.

This conservative commit-based flush ensures correctness even if multiple branches are in flight, at the cost of some execution cycles.

This two-tier recovery approach enables a balance between performance and safety in speculative execution.

```
void HistorySnapshot::flush(int ID_in_Queue, bool actucalPrediction){
    if(actucalPrediction){
        int instructionNumber=Global::instructionQueue[ID_in_Queue].
            instructionNumber;
        int target_position=Global::btb.getTargetPosition(instructionNumber);
        Global::fetch_pointer = target_position;
    }else{
            Global::fetch_pointer++;
    }
    Snapshot* snapshot = findMatchingSnapshot(ID_in_Queue);
    clearHistoryAfter(ID_in_Queue);
    Global::fetchInstructionQueue.clear();
    Global::ROBuffer.clear();
    Global::RS_INT_Queue.clear();
    Global::RS_LOAD_Queue.clear();
    Global::RS_STORE_Queue.clear();
    Global::RS_FPadd_Queue.clear();
    Global::RS_FPmult_Queue.clear();
    Global::RS_FPdiv_Queue.clear();
    Global::RS_BU_Queue.clear();
    Global::completeRSQueue.clear();
    Global::BUQueue.clear();
    Global::LoadQueue.clear();
    Global::LoadHazardQueue.clear();
    Global::LoadQueue.clear();
    Global::fetch_pointer=0;
    Global::robHead=0;
    Global::robTail=0;
    Global::ROBuffer.clear();
    Global::INT_pipeline.clear();
    Global::LOAD_pipeline.clear();
    Global::STORE_pipeline.clear();
    Global::FPadd_pipeline.clear();
    Global::FPmult_pipeline.clear();
    Global::FPdiv_pipeline.clear();
    Global::BU_pipeline.clear();
    Global::completeRSQueue.clear();
    Global::BUQueue.clear();
    Global::LoadQueue.clear();
    Global::LoadHazardQueue.clear();
    Global::StoreQueue.clear();
    //Roll back RAT, register value instructionQueue, set fetch pointer
    Global::memory_value = snapshot->memory_value;
    Global::renaming_worker = snapshot->renaming_worker;
    Global::rsFullNumber=snapshot->rsFullNumber;
    Global::stallCount_RSFull=snapshot->stallCount_RSFull;
    Global::stallCount_ROBFull=snapshot->stallCount_ROBFull;
    Global::instructionQueue=snapshot->instructionQueue;
    Global::stallCount_ROBFull=snapshot->stallCount_ROBFull;
    Global::renameStall=snapshot->renameStall;

}
```

Listing 9: Branch predict false flush

3.6 Load-Store Memory Hazard Handling

Load Store memory hazard can not be resolved by register renaming. To handle memory hazards between `fld` (load) and preceding `fsd` (store) instructions, the simulator implements a store-to-load forwarding mechanism with conservative checks to ensure memory consistency.

For each `fld` instruction, before issuing the load operation, the simulator scans preceding

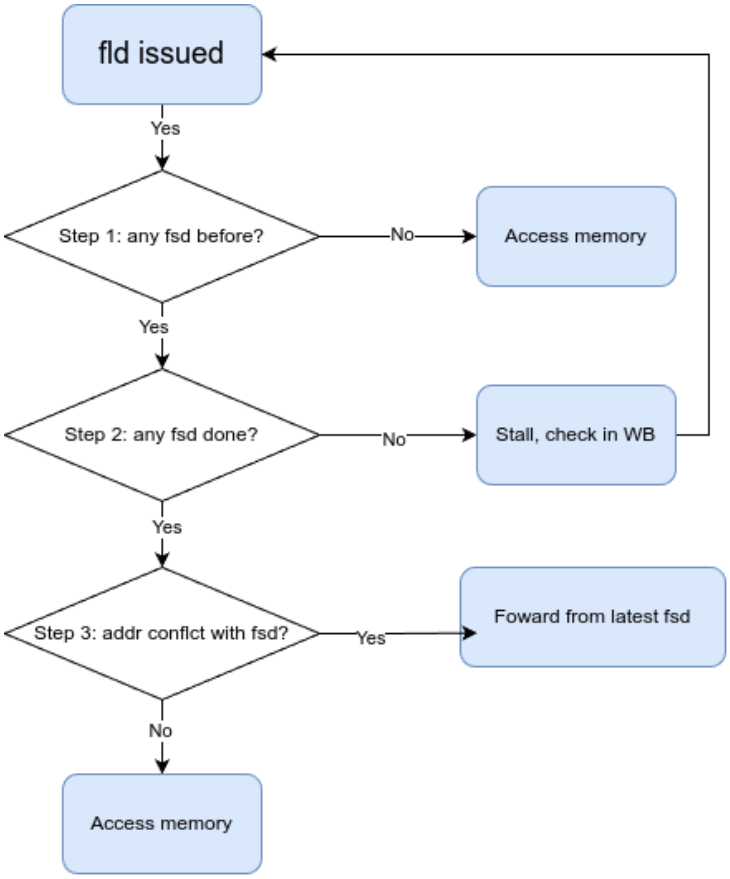

Figure 2: Load Store memory hazard handle

in-flight `fsd` instructions in program order (i.e., those with lower ROB index) and applies the following rules:

1. **No preceding store**: If there is no earlier `fsd` in the pipeline, the `fld` directly reads from memory and forwards the result to its destination physical register.

2. **Safe stores with different addresses**: If all preceding `fsd` instructions have completed their address computation, and none of them write to the same address as the `fld` is about to read, then the load proceeds to access memory and forward the result.

3. **Matching completed store**: If there exists one or more `fsd` instructions with matching addresses (same effective address as the `fld`) and all of them have completed address computation, the `fld` will forward the value directly from the most recent such `fsd`'s source register (bypassing memory).

4. **Unresolved address hazard**: If any preceding `fsd` has not yet completed address computation, and might alias with the `fld`, the load is stalled. Each cycle, during the Write-Back stage, the simulator checks again whether all earlier `fsd` instructions have resolved their addresses. Once resolved, the `fld` will follow one of the above rules accordingly.

This logic ensures that `fld` never reads stale data and that stores are not reordered before earlier conflicting stores complete. It mimics the behavior of a store buffer with conservative load speculation and forwarding. The design enables functional correctness while reducing unnecessary stalls through intelligent address-aware forwarding.

4 Result Presentation

A sample RISC-V-like program was simulated: a loop that loads, multiplies, and stores floating-point values. The simulator correctly tracks instruction execution, dependencies, branch decisions, and final memory updates, demonstrating its intended design.

4.1 Assembly Code

Assembly code used for the analysis is:

```
0, 111
8, 14
16, 5
24, 10
100, 2
108, 27
116, 3
124, 8
200, 12
addi R1, R0, 8
addi R2, R0, 124
fld F2, 200(R0)
loop: fld F0, 0(R1)
      fmul F0, F0, F2
      fld F4, 0(R2)
      fadd F0, F0, F4
      fsd F0, 0(R2)
      addi R1, R1, -8
      addi R2, R2, -8
      bne R1,$0, loop
```

Listing 10: Branch predict false flush

Tabulated Configuration and Performance Data

Configuration (nf, ni, nw, nr, nb)	Cycles	Fetch Stalls	Decode Stalls	ROB Stalls	RS Stalls
(4, 16, 4, 16, 4)	19	0	0	0	1
(4, 16, 2, 16, 2)	21	0	0	0	0
(2, 16, 4, 16, 4)	19	0	0	0	0
(4, 4, 4, 16, 4)	19	0	1	0	1
(4, 16, 4, 4, 4)	32	3	5	10	1
(4, 16, 4, 8, 4)	21	1	1	3	1
(4, 16, 4, 16, 4)	19	0	0	0	1
(4, 16, 4, 32, 4)	19	0	0	0	1

Table 1: Pipeline performance under different architectural configurations

Comparative Analysis

1. Baseline Configuration

The configuration (4, 16, 4, 16, 4) appears multiple times with consistent performance:

- Cycles = 19

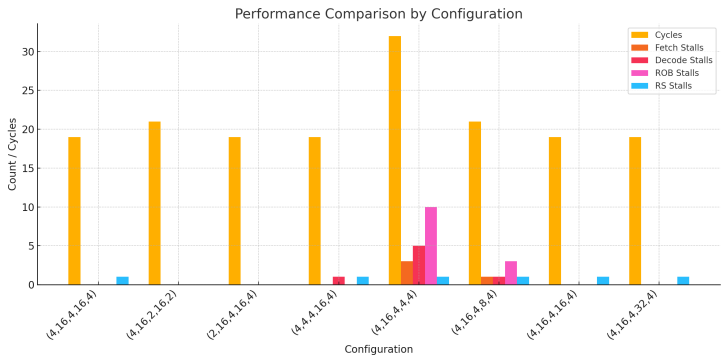

Figure 3: Load Store memory hazard handle

- No significant stalls except a minor RS stall (1)

It serves as the performance baseline for comparisons.

2. Impact of Writeback Bandwidth (`nw`)

Reducing writeback bandwidth from 4 to 2 increases Cycles from 19 to 21:

- Indicates that writeback throughput is performance-critical.

3. Effect of Fetch Units (`nf`)

Decreasing `nf` from 4 to 2 had negligible impact:

- Cycles remained at 19; all stalls eliminated.
- Suggests fetch bandwidth is not the bottleneck under this workload.

4. Issue Queue Size (`ni`)

Reducing issue queue from 16 to 4:

- Introduced decode stall (1)
- No change in cycles—indicating slight pressure on decode stage.

5. ROB Size (`nr`)

Critical to pipeline depth:

- At nr = 4: rob stalls = 10, Cycles = 32 (significant slowdown)
- At nr = 8: rob stalls = 3, Cycles = 21
- At nr = 16 or higher: no rob stalls, optimal performance.

6. Reservation Station Size (`nb`)

Effects are secondary:

- No rs stalls at nb = 2 in one config
- Some stalls appear at nb = 4, but no pattern implying strong dependency.

5 Conclusion and Insights

The results of the parameterized pipeline simulation strongly demonstrate how architectural resource allocation directly affects performance in an out-of-order CPU. Based on the comparative table and runtime metrics, we conclude:

- **Reorder Buffer (ROB) size** has the most profound effect on performance. With an insufficient ROB size (e.g., $nr = 4$), the system experiences severe stalling (10 rob stalls) and cycle inflation (32 cycles), highlighting the importance of adequate instruction window sizing for dynamic scheduling.

  csharp Copy Edit

- **Writeback bandwidth (nw)** is another critical factor. Reducing nw from 4 to 2 increased execution cycles, even though stalls remained low, indicating latent throughput saturation.

- **Fetch units (nf)** and **issue queue size (ni)** have moderate impacts. The former had negligible influence under this specific workload, while the latter introduced decode stalls but did not significantly slow down overall execution.

- **Reservation Station (RS) size** showed less consistent influence. Although RS stalls appear when $nb = 4$, the impact was not dominant, suggesting RS capacity in this workload was generally sufficient.

The simulator's robustness, including precise misprediction recovery (via `HistorySnapshot`) and correct pipeline draining on termination, allows reliable and accurate cycle-level architectural analysis. Furthermore, the design of the branch prediction subsystem—featuring local 2-bit dynamic predictors—improves branch accuracy and reduces speculative overhead.

Overall, the experiment validates the simulator's capacity to model real-world superscalar out-of-order CPUs. The stall behavior and performance patterns observed correlate well with theoretical expectations of Tomasulo-based scheduling and register renaming architectures.

6 Future Work

To further enhance this simulator, the following extensions are recommended:

- Implement **dynamic scheduling visualization tools** to show per-cycle pipeline activity.
- Add **cache and memory hierarchy models** to explore load/store latency impact.
- Introduce **multi-threaded simulation** support for SMT or multi-core evaluation.
- Extend instruction set coverage, including division, comparison, and atomic operations.

These additions would further solidify the simulator as a powerful research and teaching tool in modern CPU architecture.

References

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. Morgan Kaufmann, 2017.