Advanced Superscalar Execution R10K Architecture

Xiaomi Zhou, Yiting Lai, Yiwen Jiang, Tianrui Dang, Zhiheng Zhang, Yu Zhang

***Abstract*—This project presents an implementation of an out-of-order execution R10K architecture designed for two-way superscalar processing. The core optimizes instruction-level parallelism by dynamically reordering instructions through multiple functional units with varying latencies. Key features include early branch resolution via a robust branch prediction mechanism (utilizing a branch target buffer and a tournament predictor), efficient data forwarding through a Load-Store Queue, and enhanced fetch performance with instruction prefetching and non-blocking caches. The design includes separate 256-byte instruction and data caches with associative and non-blocking mechanisms to minimize memory access latency. The pipeline stages include Fetch, Dispatch, Schedule, Execute, Commit, and Retire, supporting efficient execution and fast recovery from branch mispredictions. Comprehensive unit-level and system-level testing verified the design's correctness, highlighting its efficiency and robustness for modern processor workloads.**

***Index Terms*—R10K, Two-way Superscalar, Early Branch Resolution, Data Forwarding, Load-Store Queue, Prefetching, Non-blocking Cache, Fetch Enhancements.**

I. Introduction

This report introduces a 2-way superscalar Out-of-Order (OoO) R10K computer architecture core, developed as part of the FA2024 EECS470 course project. The R10K architecture is designed to exploit instruction-level parallelism through dynamic reordering of instructions, enabling the processor to execute them out of order while maintaining program correctness. This dynamic scheduling optimizes performance by reducing stalls and ensuring that functional units remain active.

The advanced features of our R10K design include:

1. 2-way superscalar execution
2. Early branch resolution
3. Fetch enhancements:
    a. More sophisticated branch predictor (tournament branch predictor)
    b. Return address stack
4. Memory hierarchy improvements:
    a. Instruction prefetching
    b. Associative caches
    c. Non-blocking L1 data cache
5. Load-Store Queue
6. Data forwarding from stores to loads

The R10K architecture supports 2-way superscalar execution, allowing the processor to fetch, decode, and issue two instructions per cycle, significantly increasing throughput. Early branch resolution further enhances control flow efficiency by reducing the penalties associated with branch mispredictions, enabling faster recovery. Fetch stage enhancements feature a tournament branch predictor, combining global and local mechanisms for high prediction accuracy, and a return address stack that improves predictions for function returns, ensuring smoother control flow.

Memory hierarchy improvements are another key feature of the R10K architecture. It incorporates instruction prefetching, associative caches, and a non-blocking L1 data cache to minimize memory access delays and improve overall pipeline efficiency. The inclusion of a Load-Store Queue (LSQ) allows the processor to handle memory operations out of order while maintaining program correctness. The LSQ also facilitates data forwarding from stores to dependent loads, further optimizing memory operation throughput.

The core design features multiple functional units with varying latencies, which are tightly integrated with an advanced reservation station and a unified reorder buffer (ROB), ensuring seamless execution, dependency resolution, and precise exception handling. The pipeline supports speculative execution and aggressive resource utilization, further boosting throughput and addressing performance bottlenecks.

To mitigate memory bottlenecks, the architecture includes separate, highly optimized 256-byte instruction and data caches. These caches are designed with non-blocking and set-associative mechanisms to minimize latency and maximize data availability. A prefetching mechanism and a robust Miss Status History Register (MSHR) further enhance memory access efficiency, ensuring smooth operation even under heavy workloads.

Through its combination of dynamic instruction scheduling, optimized functional units, efficient memory systems, and advanced branch prediction, the R10K architecture delivers high performance, scalability, and efficiency. These features make it well-suited to address the demands of modern, complex workloads.

II. Implementation Of Basic Feature

The basic structure of the whole design is shown in Figure 1. The R10K works in terms of a pipeline with the stages F(Fetch), D(Dispatch), S(Schedule), X(Execute), C(Commit), and R(Retire). The processor fetches instructions from the instruction cache and then places them into the Instruction Buffer. Instructions in the buffer will be then dispatched to the Map Table and ROB. Also, Dispatch allocates entries in the RS and ROB for each instruction. Instructions wait in the RS at the Schedule stage. At the execution stage, the instructions are executed on the allocated functional units. At Commit, instructions complete execution and move the head of the Reorder buffer for in-order commit. At the Retire stage, when the instructions are committed, their results are finalized and then be used by the retired instructions for further use.

A. Instruction Fetch Unit

The Instruction Fetch Unit is responsible for retrieving instructions from memory, managing prefetching, and working with branch prediction to determine the program counter (PC) for the next fetch cycle. It integrates an Instruction Cache (Icache) and an Instruction Buffer to ensure efficient and non-blocking operation. Advanced features, including the prefetching mechanism, branch prediction logic, and their interactions, are discussed in detail in the advanced features section.

The number of instructions sent to the dispatch stage is determined by multiple factors, including whether there are more than four pending branches, the availability of free slots in the Reservation Stations (RS) and Reorder Buffer (ROB), the number of free slots in the Load/Store Queue, and whether the Instruction Buffer contains valid instructions ready to be dispatched. These conditions ensure that the dispatch stage functions efficiently while maintaining resource constraints and avoiding structural hazard.

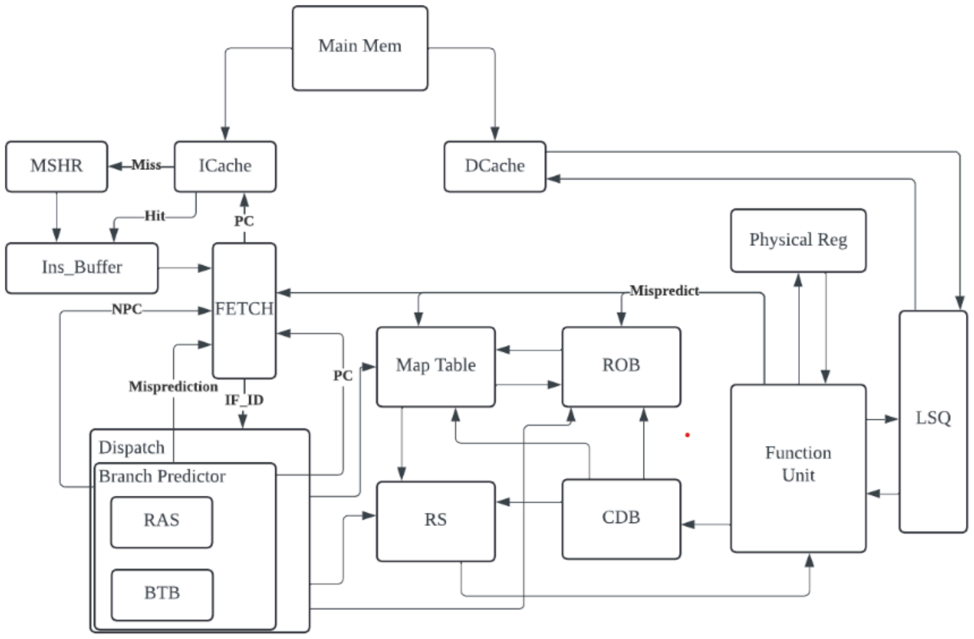


Figure 1. R10K overall structure

B. Dispatch Unit

The Dispatch Unit contains the Instruction decoder and the Roll_back logic. In this unit, the instruction package will be decoded according to its instruction features. The validity of the instruction will be first determined, then the valid instruction will have the categories combined in the package and be passed to the next related stages. To meet two-way superscalar execution, two decoders are placed in Dispatch stage.

Dispatch Unit is also controlled to dispatch a certain number of instruction packages. The number ranges from 0 to 2, which will invalidate the corresponding instruction packages and send Zero_reg to next stages when the number is 0 or 1. This number is also associated with the instruction buffer and fetch stages.

C. Map Table & Free List

The MapTable and FreeList are fundamental components of register renaming, ensuring efficient resource management and correct program execution. The MapTable maintains the mapping between architectural registers and physical registers by tracking the most recent physical register allocated to each architectural register. This mechanism resolves data dependencies, allowing instructions to execute out of order without conflicts. The MapTable is designed to process architectural register indices and return their corresponding physical register indices in a single cycle, enabling fast access to updated register mappings.

The FreeList, in contrast, manages the pool of available physical registers. When a new instruction is dispatched, the FreeList provides an unused physical register, which is then assigned to the architectural register through the MapTable. As instructions retire and the physical registers associated with them are no longer needed, these registers are returned to the FreeList, making them available for future allocations.

Together, the MapTable and FreeList work cohesively to ensure optimal register allocation, minimize resource contention, and enhance the overall performance and efficiency of the processor.

D.Reservation Station(RS)

Reservation Station is a hardware structure designed to dynamically schedule instructions based on data availability rather than program order. It plays a critical role in enabling the Tomasulo algorithm for out-of-order execution in the R10K microarchitecture. We implement a generic RS to issue instructions. The RS has two-way inputs (IFID packets), instructions dispatched from dispatch stage and outputs (IDEX packets), the instructions issued to the Function Unit. The RS holds an instruction if any operand value the instruction needs is not ready(still being calculated) and issues the instructions whose operands are ready. It supports up to four entries and receives physical register tags and ready status from the map table. It also updates the ready bits for all valid RS entries based on signals from the common data bus (CDB) when instructions complete execution. Furthermore, the RS takes branch misprediction rollback requests directly from the branch predictor.

To prioritize instruction issuance, we design priority logic for the RS. Branch instructions are issued first, followed by loads and stores, then multiplications, and finally, all other instructions. This ensures that performance-critical instructions are issued as early as possible once they are ready. Importantly, the RS uses the current ready bits to issue instructions within the same cycle, instead of relying on the next cycle's ready bits, effectively saving one cycle per instruction.

E. Reorder Buffer

The Reorder Buffer keeps the instructions in the processor and makes sure these instructions will be retired in order. Our ROB's capacity is 32. In every cycle, ROB receives the ID packets from Decoder as the same as the Reservation Station and stores two, one or zero instructions in the ROB entries sequentially. ROB updates its entries by receiving packets from CDBs. Each cycle, there will be two, one or zero instructions will be issued only when they are completed and reach the ROB head.

F. Function Unit

Our design includes two Arithmetic Logic Units (ALUs), one fully-pipelined multiplier, and one branch condition calculator. The ALUs also handle load/store address computation and branch address calculation, allowing reuse to minimize area and time overhead. By consolidating these functions, we reduce the number of ALU types to just three: general-purpose ALUs, the multiplier, and the branch condition calculator.

Each Functional Unit (FU) receives its input operands and control information via Reservation Station (RS) packets. Once computation is complete, the results are sent to the Common Data Bus (CDB) for broadcasting. To manage the CDB selection efficiently, we introduced a structure called pending_packet, which stores instructions that have been computed but are not yet broadcasted.

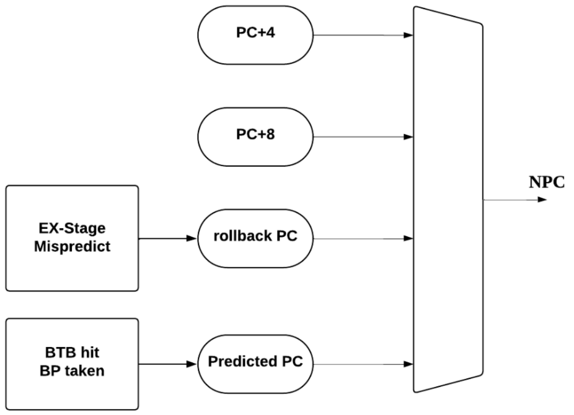
Branch instructions are prioritized in the CDB policy to minimize branch recovery latency. When a branch instruction completes in the execution (EX) stage, it is immediately broadcasted. This prioritization ensures that any misprediction can be promptly identified, allowing the pipeline to flush and recover with minimal delay.

For all other instructions, the pending_packet structure operates as a First-In-First-Out (FIFO) queue, ensuring that instructions are broadcasted in the order they were computed. This maintains fairness and prevents starvation of lower-priority instructions.

III. Implementation Of Advanced Features

A. Icache, Instruction Buffer and Prefetch

In the Fetch Stage, we integrate the Instruction Cache (Icache) and Instruction Buffer to accelerate the instruction fetch process. The Instruction Cache is designed as a non-blocking, dual-read-port, single-write-port, direct-mapped cache. This architecture minimizes the memory access latency for frequently accessed program counter (PC) addresses.



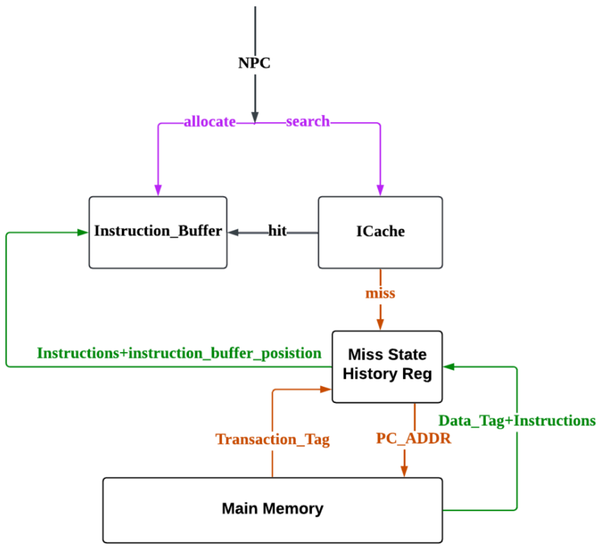


Figure 2. NPC select logic

Figure 3. Icache workflow

To achieve non-blocking behavior, the architecture utilizes a Miss Status History Register (MSHR), as shown in Figure 3. The MSHR enables the processor to continue fetching instructions without stalling, even if a previous fetch has not yet completed. The Next Program Counter (NPC) selection logic, shown in Figure 2, determines the program counter's value to ensure accurate and efficient instruction fetching. On reset, the program counter initializes to the program's starting address. If a branch misprediction occurs, the program counter updates to the correct branch target for recovery and pipeline flushing. In cases of a successful branch prediction, indicated by a Branch Target Buffer (BTB) hit or the use of a return address stack, the program counter is set to the predicted target address. When no branch-related conditions apply, and there are sufficient resources in the instruction buffer, no stalls, and no conflicts, the program counter increments to fetch the next sequential instructions. This logic ensures seamless handling of branch predictions, recovery, and sequential fetching to maintain high performance and correct program flow.

Once the NPC is calculated in the Instruction Fetch (IF) stage, an entry is allocated in the Instruction Buffer. The Instruction Buffer, designed as a First-in-First-out (FIFO) architecture, ensures that instructions are dispatched in the order they were fetched. If the Data Cache (Dcache) is in use, the IF stage does not update the NPC, and no additional requests are sent to the Instruction Cache (Icache), avoiding resource conflicts. This integration of the MSHR, NPC selection logic, and Instruction Buffer ensures that the fetch stage operates efficiently and maintains a steady flow of instructions into the pipeline.

Next, the Icache is checked for a hit or miss. If there is a hit, the instruction is filled into the Instruction Buffer. In case of a miss, a memory load (MEM_LOAD) command is sent to the main memory via the Dmem interface. The MSHR records the transaction tag associated with the memory request and the corresponding position in the Instruction Buffer. When the data tag returns, the corresponding entry in the Instruction Buffer, as recorded by the MSHR, is updated. The Instruction Buffer always outputs the first two stored instructions, ensuring that the fetch stage maintains in-order status.

Additionally, during this stage, the fetch logic identifies whether an instruction is a branch and signals the branch predictor to initiate prediction ahead of time.

In the event of a branch misprediction, the Instruction Buffer is cleared to prevent unnecessary instructions from entering the processor. Icache will stall if Dcache is in use.

B. Dcache

A processor interacts with memory to load and store data. However, memory latency is typically about seven times longer than the processor's clock cycle, requiring the processor to wait approximately seven cycles to retrieve the requested data. To address this delay and enhance efficiency, we implemented a non-blocking associative Dcache. Our Dcache processes one load or store instruction from the Load-Store Queue (LSQ) and outputs a 64-bit data block back to the LSQ.

When a load instruction enters the Dcache, it checks for a cache hit. If a hit occurs, the data is immediately sent to the LSQ. On a miss, a load request is sent to memory, and the load instruction is placed in the Miss Status Holding Register (MSHR) to await the response. For a store instruction, the Dcache similarly checks for a cache hit. If a hit occurs, a store request is sent directly to memory. On a miss, a load request is first sent to memory, and the store instruction is placed in the MSHR. Once the corresponding load response is received, and it matches the store instruction in the MSHR, a store request is sent to memory, merging
the loaded data and the store instruction's data.

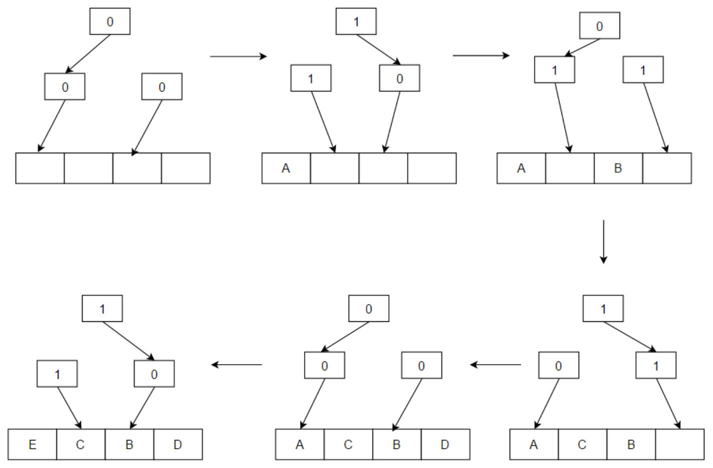


Figure 4. PLRU Algorithm

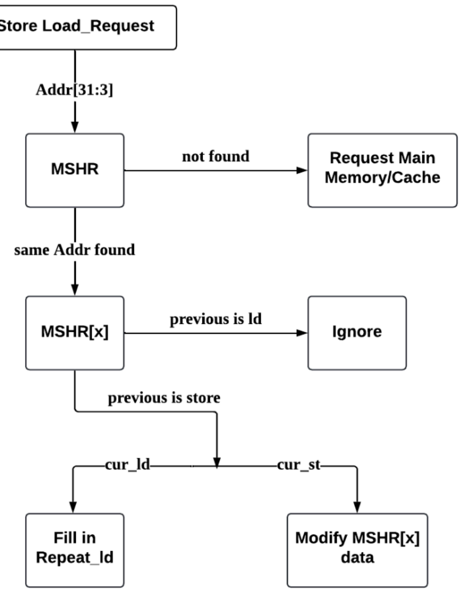


Figure 5. Request Merge Process

Since we are using a set-associative data cache, we employ a pseudo least-recently-used (PLRU) replacement algorithm, shown in Figure 4. Instead of recording the exact time of usage for each cache line, which would incur significant area and time overhead, PLRU uses a binary tree structure to approximate the least-recently-used line. Each node in the tree contains a bit indicating the direction (left or right) of the less-recently-used path, allowing the cache to quickly identify a candidate for eviction. In our design, the PLRU tree is updated whenever the cache is accessed to reflect the most recent usage pattern. The PLRU tree features two access ports: one for handling cache hits and updating the tree accordingly, and another for updating it when data is returned from the main memory. Additionally, the PLRU tree provides an output line to indicate the next candidate for allocation, ensuring efficient replacement decisions during cache misses. This dual-port configuration, combined with the output allocation line, guarantees that the replacement algorithm remains efficient and consistent for both hit and miss scenarios.

To improve efficiency further, the Dcache merges memory requests when multiple instructions access the same memory block. The process is shown in Figure 5. When an instruction enters the Dcache, it checks the current MSHR for merging opportunities. The process varies for loads and stores:

1. **Load-Load Merge**: If a load request for a load instruction sharing the same memory block has already been sent, the current load instruction is ignored.
2. **Load-Store Merge**: If a load request for a store instruction sharing the same memory block has already been sent, the current load instruction is placed in a "repeat load register." When the load response returns from memory and matches the load instruction in the repeat load register, the data, merged with the store instruction's data, is sent to the LSQ.
3. **Store-Store Merge**: If a load request for a store instruction sharing the same memory block has already been sent, the current store instruction combines its data with the previous store instruction's data stored in the MSHR. The merged data is then stored in the same MSHR entry.

By leveraging these mechanisms, the Dcache minimizes redundant memory operations and effectively reduces latency, ensuring efficient data handling and improved processor performance.

C. Branch Predictor

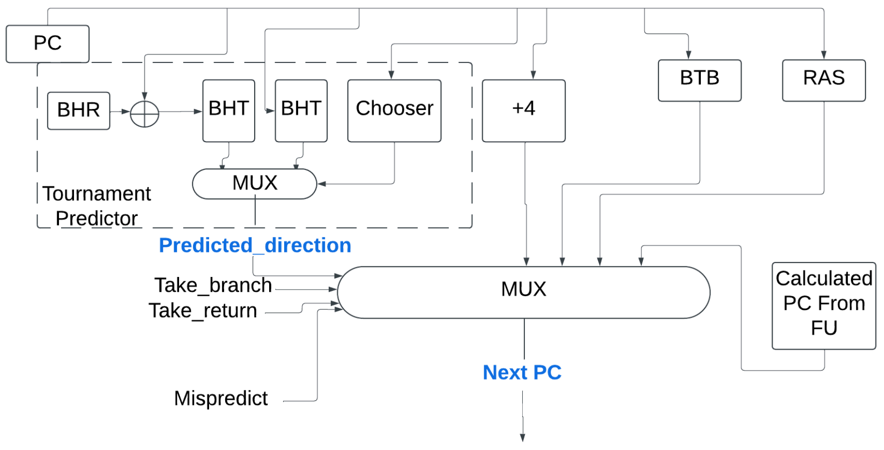


Figure 6. Branch predictor

To enhance the accuracy of branch prediction, we implemented a Tournament Branch Predictor for direction prediction, along with a Branch Target Buffer (BTB) and a Return Address Stack (RAS) for branch target prediction. The structure of our branch predictor is shown in Figure 6. During each cycle, only one branch is processed by the Branch Predictor. All branches, including both conditional and unconditional, will query the BTB for the target PC. The PC predicted by the BTB is then sent to the Function Unit along with the corresponding instruction. If the branch is conditional and the Function Unit predicts it as taken, or if the branch is unconditional, a misprediction signal is triggered if the target PC computed by the Function Unit differs from the predicted PC. When a branch resolves, the calculated target PC is written into the corresponding BTB entry. For function calls, the PC value (PC + 4) is pushed onto the RAS. When a function return occurs, the target PC is obtained by popping the PC from the RAS. For direction prediction of all conditional branches, we use the Tournament Branch Predictor. We implemented a meta predictor that uses the Two-Bit Saturating Counters to choose from the simple BHT and the correlated predictor. At the start of the prediction, a snapshot of the BHR will be taken. The direction predicted by the PHT is then shifted into the BHR for the next prediction. When a conditional branch resolves, both the simple BHT and the correlated predictor will be updated. The checkpointed BHR will be XORed with the PC of the resolved branch to identify the corresponding PHT entry in the correlated predictor that needs updating. The new BHR is generated by shifting the correct direction into the checkpointed BHR.

D. Load Store Queue

In the Load Store Queue, we separated load and store instructions from other instructions, managing the memory-interacting instructions independently. According to Figure 7, there are 4 load units and the store queue has 8 entries, but their sizes can be adjusted as needed. In order to manage the requests from the store queue and load units, we implement an interface between LSQ and Dcache. The interface will prioritize accepting requests from the store queue. If the store queue does not issue any request, the interface will use a priority selector to choose among the requests issued by the load units. Our LSQ has data forwarding from the store queue to load units to increase the efficiency of our processor.

Each cycle, zero, one, or two instructions may be dispatched. The LSQ will allocate store queue entries or load units to accommodate these instructions. LSQ updates its store queue entries and load units by receiving packets from CDBs. For every load instruction, as soon as it occupies a load unit, it receives a unique checklist indicating all older store instructions. Load units will forward data from the store queue according to the checklist. In order to manage the data size, we implemented data_mask to make sure load units will get forward the data correctly. If the data_mask is clean after forwarding, the load instruction will be set to CDB_ready and wait for CDBs to pick, if the data_mask is not clean, it will be sent to the interface.

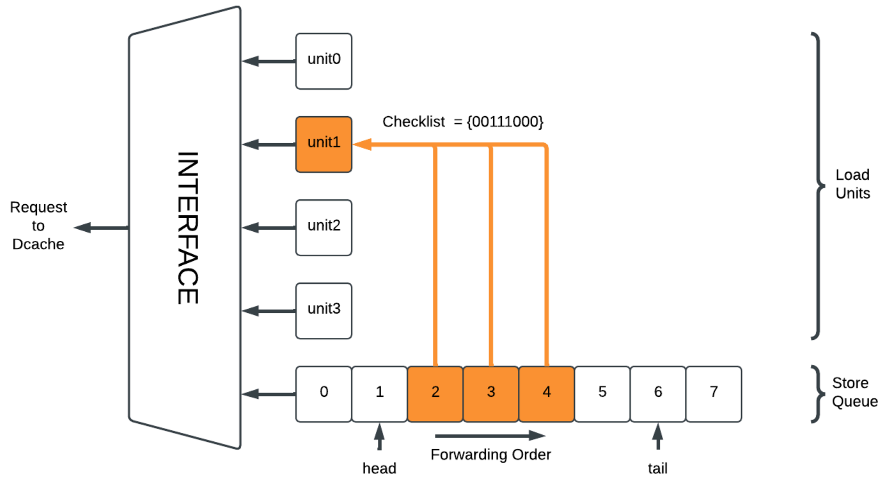


Figure 7: LSQ and interface

E. Fast Branch Recovery

When a misprediction is broadcast, all relevant modules must recover or flush their state to ensure correct execution. To facilitate this, both the mispredicted signal and the resolved_b_tag (the branch tag) are broadcast simultaneously. Each module maintains internal copies of its state associated with every branch, allowing efficient recovery when a misprediction occurs.

The Reorder Buffer (ROB) and Load/Store Queue (LSQ) save and restore their tail pointers to the state they held when the resolved branch was originally issued or resolved. This ensures that only instructions associated with the mispredicted branch and its successors are flushed.

The Freelist saves and restores its head pointer to the state corresponding to the resolved branch's issue or resolution. This prevents the premature reuse of physical registers associated with the mispredicted branch, maintaining register allocation integrity.

The Map Table saves its state in checkpoints at key stages, such as when a branch is issued. Upon a misprediction, it restores the saved state from the relevant checkpoint, ensuring correct mappings of architectural to physical registers.

The Reservation Stations (RS) and Functional Unit (FU) pipeline entries are squashed if the b-mask bit for the mispredicted branch is set to 1. Additionally, the branch stack entry and the associated b-mask bit are cleared to remove any traces of the mispredicted branch from the pipeline.

F. Early Branch Resolution

Early Branch Resolution allows a mispredicted branch to be recovered before it hits ROB head, reducing latency caused by branch misprediction. We implemented Early Branch Resolution using the checkpoint method on the basis of fast branch recovery. When a branch (either conditional or unconditional) is dispatched, a b_mask and a b_mask_mask (a one-hot signal) are assigned. RS will store the b_mask assigned to each instruction, and other modules will store their checkpoints in the assigned branch stack entry. Each asserted bit in the b_mask corresponds to an unresolved branch.

The data structure used to track active branches and their associated b_mask in module branch_int.sv is shown in Table 1. For clarity and better understanding, the variable names in this example differ from those used in branch_int.sv.

- active mask: Each bit corresponds to an in-flight branch.
- b_table (4 entries) : Each entry records the b_mask assigned to this instruction
- b_mask (1 entry with 4 bits)

instruction	active mask	b_table	b_mask
Branch1	1	0001	0001
Branch2	1	0010	0011
Branch3	1	0100	0111
NA	0		NA

Table 1. Fast Branch Recovery Checkpoint

When a branch resolves and it turns out to be a misprediction, the b_mask_mask assigned to the resolved branch will be sent to other modules, and they will restore to their checkpoints. The corresponding bit of the b_mask and all corresponding entries of the b_table will be cleared. When a branch resolves but it's not mispredicted, the corresponding bit in the b_mask and the corresponding bit in the corresponding entry of the b_table will be cleared.

IV. Analysis

At a later stage, we attempted to adjust the parameters of each module to achieve even better processor performance. We mainly focused on the size of the ROB and the instruction buffer. By increasing our Reorder Buffer (ROB) size to 32 entries, we observed a notable improvement in overall CPI within our 2-way superscalar processor.

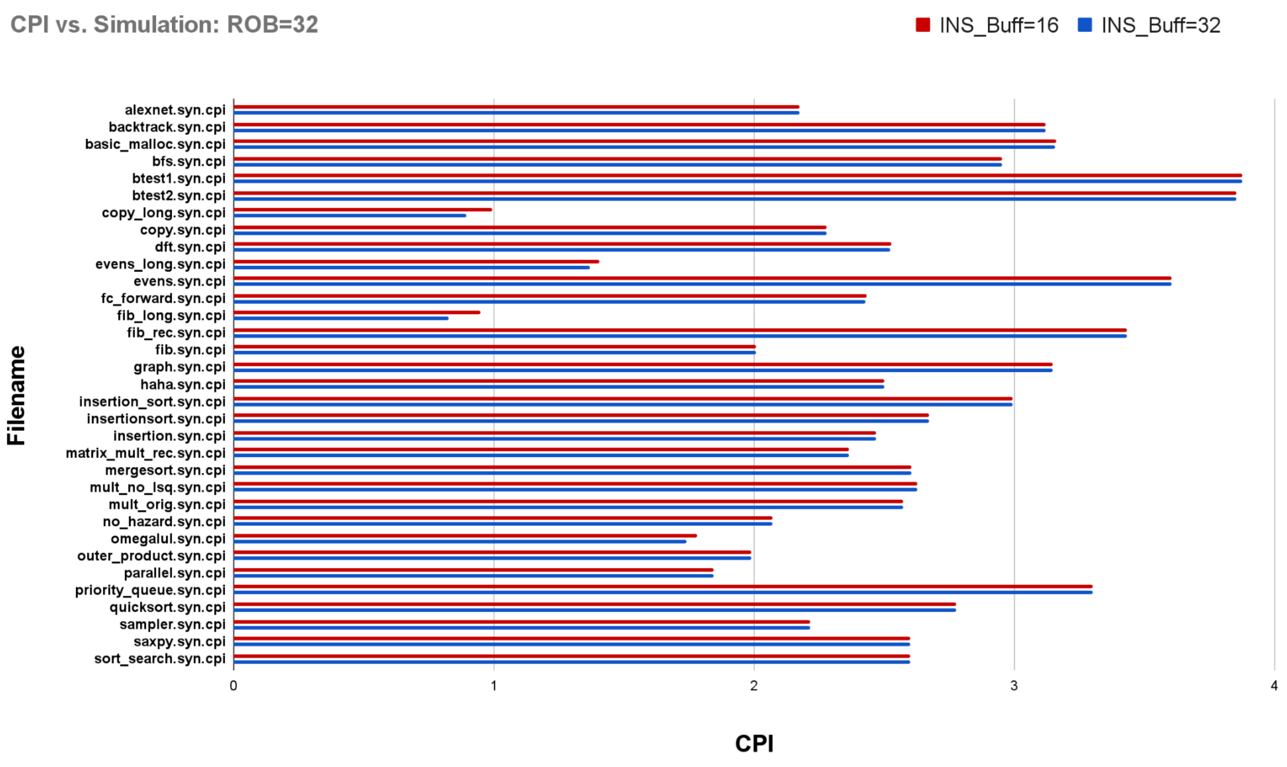


Figure 8. CPI vs INS_Buff_sz (ROB_sz = 32)

According to Figure 9, initially, with a smaller ROB (for example, 5 or 16 entries), we provided only limited instruction buffering, which constrained the out-of-order execution capability. This limitation was reflected in higher CPI test programs. For instance, in the "copy_long.syn.cpi" , a ROB of 5 entries resulted in a CPI of about 1.49, while increasing it to 16 entries reduced the CPI to roughly 0.99. Further increasing the ROB to 32 entries lowered the CPI to around 0.89, demonstrating a clear trend of performance improvement as the ROB size grew. In the "fib_long.syn.cpi" , the CPI dropped from approximately 1.37 at ROB=5, to 0.95 at ROB=16, and further down to about 0.83 at ROB=32.

In programs with stronger dependencies or more complex logic, having more instructions in flight (larger ROB) means more opportunities to effectively utilize the processor's resources. Of course, these advantages come at a certain cost. Increasing the ROB size may lead to a longer critical path and potentially affect the clock frequency. After careful adjusting, we found that while increasing the ROB to 32 entries yielded substantial benefits, going beyond 32 entries did not lead to noticeable performance improvements.

Beyond the ROB, we also investigated the effect of adjusting the instruction buffer size. According to Figure 8, the instruction buffer is critical at the frontend of the pipeline, temporarily holding fetched instructions before dispatch. Increasing the instruction buffer from 16 to 32 entries ensured a more continuous flow of instructions to the backend pipeline. According to our figure comparing CPI under these two configurations (with the ROB fixed at 32), program such as "copy_long.syn.cpi" showed a modest CPI reduction—approximately from 1 CPI at INS_Buff=16 down to about 0.9 CPI at INS_Buff=32. Similarly, "fib_long.syn.cpi" improved from around 0.95 CPI to roughly 0.83 CPI.

These reductions, though not drastic, indicate that having more instructions readily available can alleviate frontend stalls and slightly improve utilization. Just as enlarging the ROB introduces additional complexity, increasing the size of the instruction buffer can also lengthen the critical path and increase clock period. During our tuning process, we found that while a larger instruction buffer can help sustain a steady flow of instructions to the backend, the overall performance gains may be limited when facing modules with significant latency.

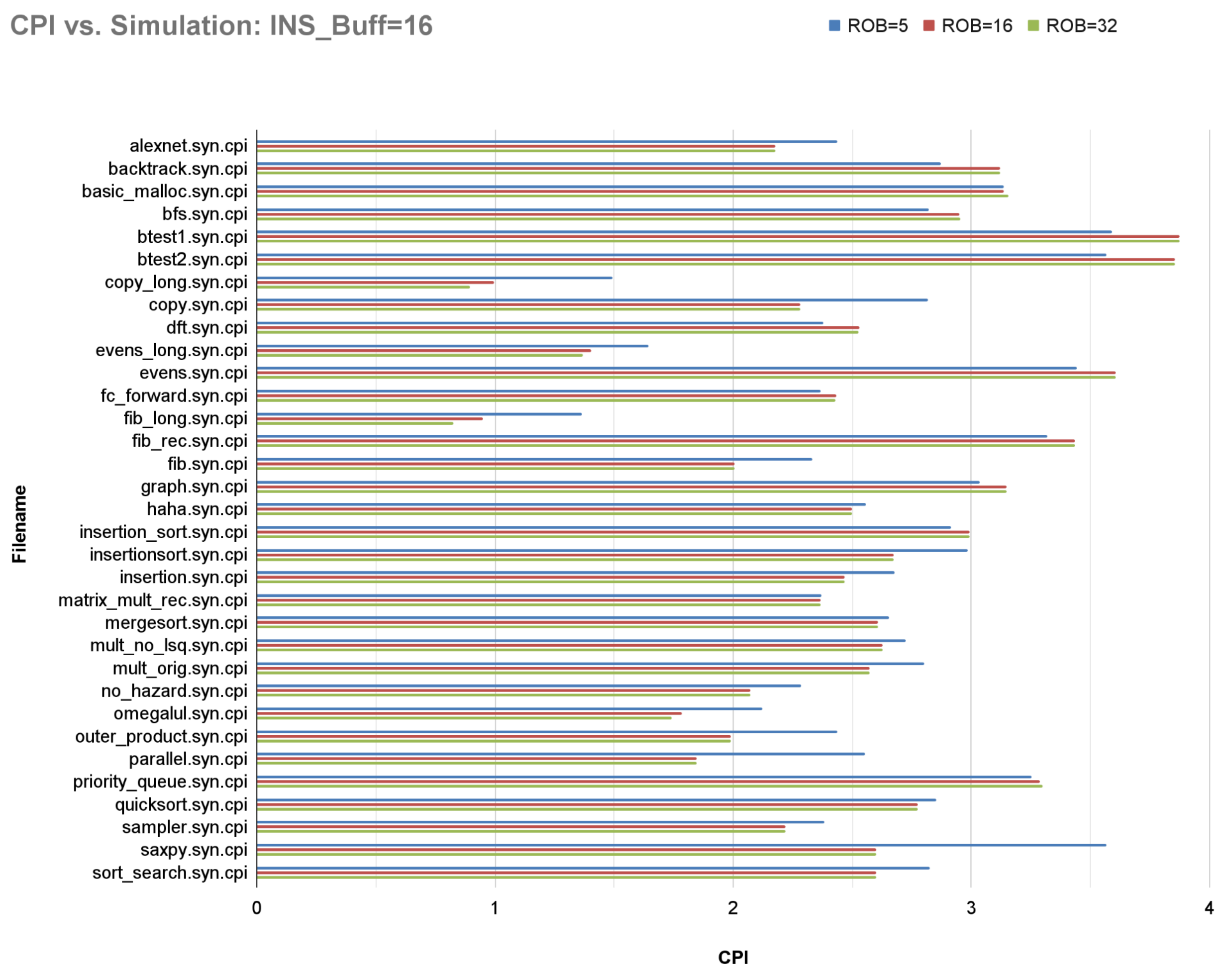


Figure 9. CPI vs ROB_sz (INS_Buff_sz = 32)

V. Testing

A. Testing Methodology

a. Unit-Level Testing

For individual modules, we developed dedicated test cases to cover a comprehensive range of scenarios, with a particular focus on edge cases to ensure robust functionality. For example, we tested the branch predictor across various patterns, including loops, alternating branches, and frequent mispredictions, to validate its accuracy and recovery mechanisms. For the load/store queue and data cache, we specifically tested scenarios where consecutive instructions accessed the same memory address to ensure proper data forwarding and consistency. For Icache and Dcache interactions, we verified that the Icache correctly stalls when the Dcache is in use, ensuring no conflicts or resource contention. In the fetch stage, we introduced structural hazards by setting other "full" signals to 1 (e.g., reservation station full or instruction buffer full) to observe and validate the system's behavior under such conditions. Each test was carefully designed to isolate and validate the functionality of the respective module independently. These tests were maintained and regularly rerun throughout the development cycle, ensuring their continued relevance and validating the design as it evolved.

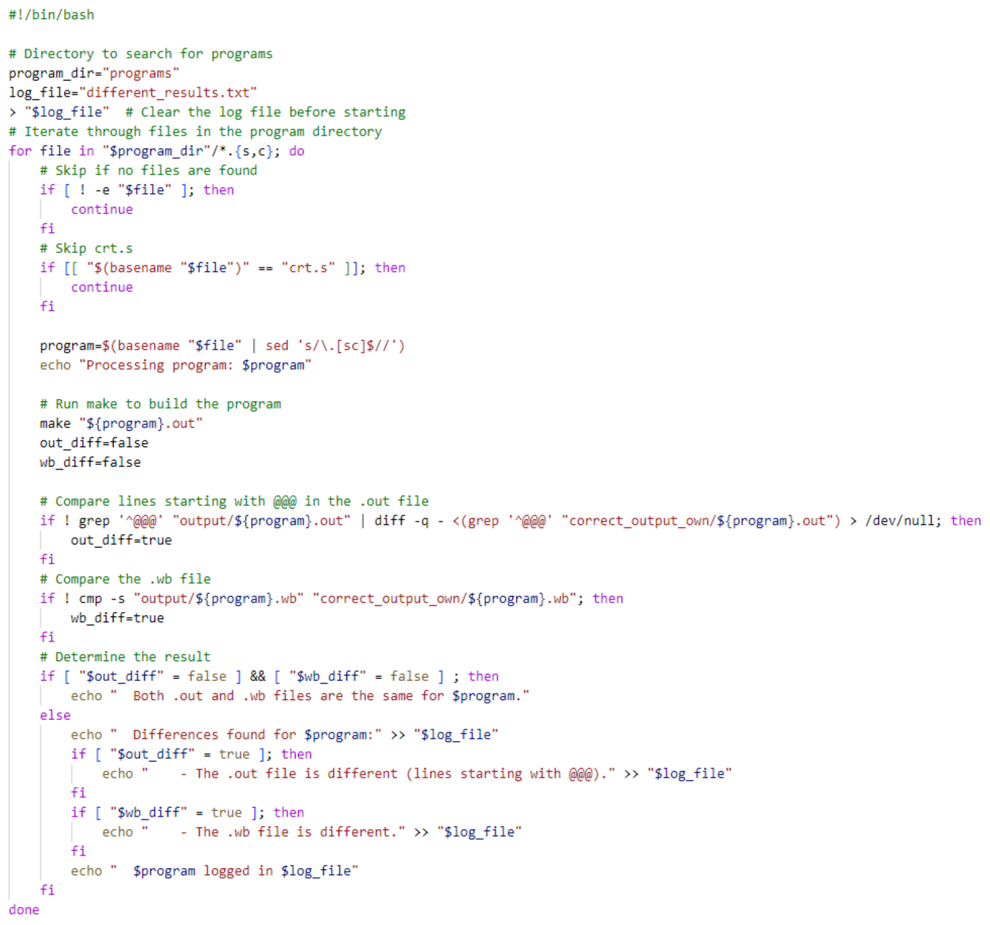
b. System-Level Testing

We used the provided test programs: alexnet, backtrack, basic_malloc, bfs, btest1, btest2, copy_long, copy, dft, evens_long, evens, fc_forwarding, fib_long, fib_rec, fib, graph, haha, insertionsort, insertion, matrix_mult_rec, mergesort, mult_no_lsq, mult_orig, no_hazard, omegalul, outer_product, parallel, priority_queue, quicksort, sampler, saxpy, and sort_search. Additionally, we wrote custom programs, including insertion_sort and simple_no, to test specific corner cases and debug our processor.

We simulated our processor with both OFLAG = 0 and OFLAG = 1 to validate correctness under different configurations. After verifying the processor's functionality through simulation, we synthesized it and reran the above programs to ensure its correctness in hardware. Synthesis testing helped us identify critical timing loops and resolve issues related to timing violations, ensuring the processor met timing constraints and functioned as intended in hardware.

c. Utilization of Script

We developed scripts to streamline and accelerate the testing process, allowing us to run all programs together in just one line and automatically compare their results, shown in Figure 10. By automating these steps, we eliminated the need for manual result comparisons, which reduced the risk of human error and saved significant time. The scripts also made it easier to quickly identify and pinpoint issues, enabling faster debugging and iteration. Additionally, they simplified command-line operations, reducing the overall time spent running commands and managing test cases. This automation not only enhanced the efficiency of our workflow but also ensured a more consistent and reliable testing process.



```
#!/bin/bash

# Directory to search for programs
program_dir="programs"
log_file="different_results.txt"
> "$log_file"  # Clear the log file before starting
# Iterate through files in the program directory
for file in "$program_dir"/*.{s,c}; do
    # Skip if no files are found
    if [ ! -e "$file" ]; then
        continue
    fi
    # Skip crt.s
    if [[ "$(basename "$file")" == "crt.s" ]]; then
        continue
    fi

    program=$(basename "$file" | sed 's/\.[sc]$//')
    echo "Processing program: $program"

    # Run make to build the program
    make "${program}.out"
    out_diff=false
    wb_diff=false

    # Compare lines starting with @@@ in the .out file
    if ! grep '^@@@' "output/${program}.out" | diff -q - <(grep '^@@@' "correct_output_own/${program}.out") > /dev/null; then
        out_diff=true
    fi
    # Compare the .wb file
    if ! cmp -s "output/${program}.wb" "correct_output_own/${program}.wb"; then
        wb_diff=true
    fi
    # Determine the result
    if [ "$out_diff" = false ] && [ "$wb_diff" = false ] ; then
        echo "  Both .out and .wb files are the same for $program."
    else
        echo "  Differences found for $program:" >> "$log_file"
        if [ "$out_diff" = true ]; then
            echo "    - The .out file is different (lines starting with @@@)." >> "$log_file"
        fi
        if [ "$wb_diff" = true ]; then
            echo "    - The .wb file is different." >> "$log_file"
        fi
        echo "  $program logged in $log_file"
    fi
done
```

Figure 10. Script for testing all programs

B. Results and Observations

The programs cover a wide range of workloads, including arithmetic operations, memory access patterns, and control flow. This ensures comprehensive coverage of the ISA. Our custom programs were written to address specific edge cases or scenarios not covered by the provided tests, such as alternating load and store operations to the same memory address but with different data.

All provided and custom programs passed both simulation and synthesized processor tests, demonstrating correctness. Synthesis testing further revealed and helped resolve timing loops that were not apparent during simulation, validating the hardware design's robustness. During testing, we identified and resolved several edge case bugs, such as incorrect data forwarding in the load/store queue and handling of certain branch misprediction scenarios. These fixes ensured the correctness and reliability of our design.

VII. Conclusion

The implementation of our two-way superscalar out-of-order execution R10K architecture has successfully demonstrated its ability to optimize instruction-level parallelism through dynamic scheduling, speculative execution, and efficient resource utilization. By incorporating advanced features such as early branch resolution, a robust branch prediction mechanism, associative and non-blocking caches, and an efficient Load-Store Queue with data forwarding, the architecture addresses key performance bottlenecks and achieves significant throughput improvements.

Through careful parameter tuning and testing, we identified the optimal configurations for critical modules, such as the Reorder Buffer and Instruction Buffer, to balance performance gains with hardware complexity and timing constraints. Systematic unit-level and system-level testing ensured the correctness and robustness of our design under a variety of workloads, including edge cases and timing-sensitive scenarios. The synthesis and simulation results validated the processor's functionality and highlighted its efficiency in managing diverse program demands.

This project underscores the importance of a holistic approach to processor design, where modular enhancements in branch prediction, memory hierarchy, and execution units work in unison to maximize performance. The final design showcases a high-performance, scalable architecture that is well-suited for modern workloads, achieving competitive CPI metrics across a range of test programs. These results not only reflect the successful realization of our design goals but also provide a strong foundation for further exploration of advanced microarchitectural features in future iterations.

VI. PROJECT MANAGEMENT

Our team maintained consistent communication and alignment through weekly meetings, where we reviewed individual contributions, resolved roadblocks, and ensured steady progress toward milestones. Each meeting included a review of completed tasks, identification of potential issues, and reallocation of responsibilities based on team members' strengths to maintain momentum.

Week Start with	Xiaomi Zhou	Yiting Lai	Yu Zhang	Yiwen Jiang	Zhiheng Zhang	Tianrui Dang
10/7	Maptable	LSQ + Execute	ROB	RS		Dcache + Icache
10/14	Execute Stage					Instruction buffer + Fetch + Dispatch
10/21		Execute Stage	Maptable + ROB Integration		Dispatch + RS Integration	
10/28	Integration					
11/4	Maptable Rollback +LSQ	Execute Stage Rollback +LSQ	ROB Rollback	RS Rollback		Dispatch_control + Rollback
11/11	prefetch +Icache +Integration		LSQ	Branch Predictor		victim cache
11/18	Dcache + Data forwarding		LSQ Rollback + Data forwarding			Synthesis
11/25	Integration + Debugging		LSQ + Integration	Branch Predictor Integration		Help to Syn
12/2	Integration + Debugging			Integration	Help to Syn	Report

Table 2. Task distribution in our team

Tasks were divided among the team to leverage individual expertise effectively. For instance, some members focused on the implementation of the pipeline stages, while others specialized in branch prediction or memory systems. This clear division of labor ensured that every aspect of the project was thoroughly addressed.

Figure 11 illustrates task distribution, highlighting each member's specific responsibilities and contributions, such as implementing functional units, optimizing the memory hierarchy, or designing the branch prediction mechanism. This collaborative approach allowed us to meet all required milestones on time while maintaining high quality.

Challenges, such as resolving timing violations and debugging complex interactions between modules, were promptly discussed and addressed during our meetings. Tools like Git for version control and Slack for real-time communication further streamlined our workflow, ensuring that the team remained aligned and productive throughout the project.

This structured and collaborative project management approach enabled us to achieve our objectives efficiently while fostering a strong team dynamic.

References

[1] University of Michigan, *EECS 470 References Page*. [Online]. Available: https://www.eecs.umich.edu/courses/eecs470/refs.html. [Accessed: Jun. 27, 2024].

[2] "Pseudo-LRU," *Wikipedia*, [Online]. Available: https://en.wikipedia.org/wiki/Pseudo-LRU. [Accessed: 8-Dec-2024].

[3] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. Cambridge, MA: Morgan Kaufmann, 2019.

[4] J. E. Smith, “A study of branch prediction strategies,” *Proceedings of the 8th Annual Symposium on Computer Architecture (ISCA)*, Minneapolis, MN, USA, 1981, pp. 135–148.

[5] M. S. Hsu et al., “The IBM R10K superscalar microprocessor,” in *IBM Journal of Research and Development*, vol. 41, no. 3, pp. 313–328, May 1997.