Out-of-Order RISC-V P6 Processor

Columbia University
Nicolas Alarcon, Donovan Sproule, Yun-Rong (Alice) Du

***Abstract*—This report presents a complete examination of the design, implementation, and evaluation of our pipelined out-of-order, P6 RISC-V processor.**

I. Introduction

This project implements an out-of-order (OoO) RISC-V processor based on a P6-style architecture. The design extends Tomasulo's dynamic scheduling algorithm by incorporating a Reorder Buffer and Load/Store Queue, enabling out-of-order execution while ensuring in-order memory and register writes. The processor features four main functional units, including an ALU/branch resolver, load unit, store unit, and a multiplication unit. The pipeline itself is equipped with both non-blocking, instruction and data caches. Additionally, it features a 2-bit saturating branch prediction unit to dynamically attempt to optimize branch resolution.

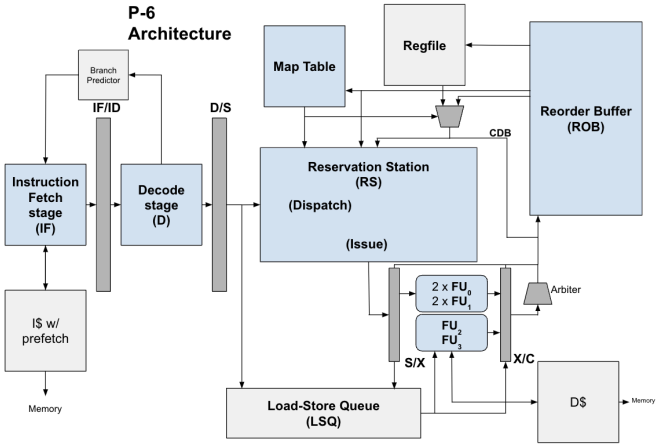

Fig. 1. A simplified implementation of our block diagram is featured above. The P-6 specific, essential structures are highlighted in blue.

II. Full Pipeline

We will begin by giving a high-level overview of the core.

The Instruction Fetch (**IF**) stage executes an instruction by making a memory request to the address corresponding to its program counter (**PC**), which is routed through the instruction cache (**icache**). If the instruction has been loaded before, the icache presents the data to the IF stage. Otherwise, it makes a memory request for the desired address to cache the data. While the icache waits for the main line, it prefetches the next cache slot to reduce the chance of a subsequent miss. The instruction is then mux'd for the 32 most/least significant bits so the address can be passed along to the decode stage (**D**).

The D stage parses the IF data, extracting information like *instruction_type, branching, validity,* etc. This packet is then marked with the Reservation Station (**RS**) index corresponding to the functional unit that will process the data. Simultaneously, the packet is passed to the branch predictor to determine whether we should make a preemptive branch jump within the IF stage.

At this point, the decoded instruction is presented to the rest of the pipeline. If it is a load or store instruction, an entry in the Load/Store Queue (**LSQ**) is allocated. All instructions present their destination tag to the map table (**MT**) and ReOrder Buffer (**ROB**), which responds with relevant tags/flags accordingly. If the value is present in the ROB/Regfile, it gets passed along with the instructions to the RS.

If the RS does not receive sufficient values for the operands, the entry will remain in the dispatch stage. Entries only enter the issue (**S**) stage if they are not waiting for values in dispatch. Entries waiting for tags in the dispatch can only proceed to the issue stage if they receive a Common Data Bus (**CBD**) broadcast with the corresponding tag. The entry will push itself to the functional units (**FU**) from the issue stage if they are not currently in use. When it allocates a functional unit, it clears the RS entry so another instruction can occupy its place.

Our pipeline features 6 FUs in the execute stage (**X**), with 2 being duplicates. We feature 2 ALU/branch resolution modules (**FU_0**), two pipelined multiplication modules (**FU_1**), a load unit (**FU_2**), and a store unit (**FU_3**). When these functional units have serviced their inputs, they place the results in the X/C register. From here, a 6-way rotating arbiter selects a result to place on the CDB, broadcasting to the commit (**C**) stage.

The LSQ consists of two circular queues, a Load Queue (**LQ**) and a Store Queue (**SQ**). Memory operations are allocated in the LQ or SQ when dispatched to keep instruction order. In the execute stage, the LSQ uses the S/X register to update the address and data (if applicable). If valid forwarding is possible from store to load, the load entry is updated and is sent to the X/C register once the load reaches the head of the LQ. The SQ head is sent to FU_3 once its address and data have been resolved and it has been retired by the ROB. The LQ head is sent to FU_2 if forwarding has not occurred, the address has been resolved, and there are no older stores present in the SQ. The older of the heads of the LQ and SQ is sent to their respective FUs, which handle the interaction between the data cache and LSQ. FU_2 then writes to the X/C register to resolve the load instruction.

The ROB works as a wraparound queue, with the head signifying the next tag/register to retire to the regfile/lsq and the tail being the following tag sent to the RS (referenced above). When the CDB broadcasts a tag, the value is stored in the ROB. This value and other pipeline control signals are pushed to the regfile/rest of the pipeline accordingly.

Various units can stall the pipeline if they are not done servicing their data or waiting for earlier instructions. Additionally, mispredicted branches trigger a full pipeline flush except for the LSQ and IF stage. The IF sets its PC to the branch target while the LQ clears itself, and the SQ marks all its valid/unretired transactions dirty.

III. Base Design Details

A. IF Stage: Instruction Fetch

The IF stage is a clocked module that takes as inputs a valid signal, a stall signal, branch redirection signals, and memory data. It outputs the instruction packet containing the 32-bit instruction, the PC, and the next PC. It interfaces directly with memory (or the instruction cache). Internally, it stores the current program counter. Functionally, all the IF stage does is keep track of where the program is in its execution and supply the rest of the pipeline with the instructions to process.

On a reset, the IF stage sets the PC to 0. If the *if_valid* signal is valid, it increments the PC by 4 (memory is stored at the granularity of bytes—each instruction is 4 bytes long). If the IF stage receives a valid *take branch*, the PC is set to the *branch target* instead. When the data corresponds to the PC address, it gets passed along with the rest of the packet to the IF/ID register. A stall triggers no change.

The memory controller is 8-byte aligned, which affects the design of the IF stage in two ways. Firstly, the data received from the memory controller is 64 bits instead of 32; when the IF stage processes the data, it must select the four most or least significant bytes depending on whether we are divisible by 4 (PC address[2] is high). The second implication of the 64-bit data response is that two instructions exist within the same cache line. Following a miss, the subsequent address will always be a hit, generating a default speedup with any cache.

B. D Stage: Decode / Dispatch

The decode stage is purely combinational. It extracts the 32-bit instruction into multiple flags that the rest of the processor core uses as control signals.

Only two changes were made to the standard 5 stage in-order decode stage. The corresponding FU / RS slot is designated here based on the instruction itself. Because we duplicated FU_0 and FU_1 modules, the RS index is rotated between 0/4 and 1/5 for ALU and multiplication instructions. Unlike before, the memory offsets for immediates of load/store instructions are also calculated in this stage.

D. Reservation Station (RS)

The Reservation Station consists of two stages— the allocation stage and the value stage. In the allocation stage, it receives the value from the D/S register, passes it to the intermediate registers, and then stores it in the rs_table. In the value stage, the ready signal in the rs_table, along with the ready signal from the functional unit, decides whether the entry can be passed to the functional unit for execution.

Each of the entries of the rs_table has the value of the destination tag (T), source tags T1 and T2, the value V1 and V2, the D_S_reg, and the ready signal. During allocation, the destination tag comes from the ROB, and the source tags come from the map table; the value is passed from either the regfile or the ROB, depending on the tag status. If the value is ready, the source tags will be wiped out, and the ready signal will be set to 1. They are all sent to the intermediate register first (for synthesizability) and an easier busy signal handle, and will be sent in the rs_table in the next cycle.

For the tag that is not resolved yet, the entry waits in the rs_table until it is broadcast from the CDB. The edge cases where CDB broadcasts in the same cycle as the allocation are handled properly.

E. Reorder Buffer (ROB)

The Reorder Buffer operates as a parameterizable circular queue allowing for out-of-order execution in the pipeline with in-order register writes. Every instruction dispatched to the ROB receives a unique tag corresponding to the index in the queue, which is used throughout the pipeline to track the instruction's progress and determine the relative age of instructions. Valid tags range from 1 to the ROB Size, as a tag value of 0 indicates the entry is ready in the register file.

Each entry in the ROB stores the destination register, the computed value received from the CDB, the next program counter, a ready-to-retire bit, and control flags.

When an instruction completes execution, its result is broadcast on the CDB. If a ROB entry is listening for that tag, it captures the value and marks itself as ready to retire. The ROB continuously checks the instruction at its head: if the entry is valid and marked done, it retires the instruction by writing the result to the register file (or signaling the LSQ in the case of store operations). After retiring, the ROB clears that entry and advances the head.

To maintain precise state, the ROB supports flushing on branch misprediction. In addition, to prevent halting the pipeline before stores are processed by the data cache, the ROB will not retire a halt instruction unless the store queue is empty.

F. Functional Units and Execute Stage

We have four kinds of functional units: FU_0 for ALU operations, FU_1 for pipelined multiplication, FU_2 for load instructions, and FU_3 for store instructions. To decrease the possibility of structural hazards of the functional units, we instantiate two FU_0s and two FU_1s.

1) *Functional Unit 0:* This is modified from the ALU from the execution stage used in the 5-stage in-order pipeline. It handles three types of instructions: arithmetic or logic operations, the conditional branch, and the unconditional branch. The first type is handled in the *ALU* submodule, and the second type is handled in the *conditional_branch* submodule.

Since the branch predictor is added, the prediction result is passed down the pipeline, and the flush signal is on only when there is a misprediction.

2) *Functional Unit 1:* This functional unit is a wrapper for a pipelined multiplier adapted for our processor. We used a four-stage pipeline and added other pipeline control signals to align with the output format of other functional units. The signed extension is also handled properly for the multiplication of data of different lengths.
3) *Functional Unit 2:* FU_2 combinatorially controls the response from the data cache for a load instruction. It receives the memory offset and memory size from the LQ in order to properly mask the data cache response. Once the data cache produces a valid response, the FU result is sent to the X/C register.
4) *Functional Unit 3:* FU_3 uses combinatorial logic to mask and shift the data already present in the memory before sending the data cache store request.

G. Load/Store Queue (LSQ)

Our design for an in-order LSQ functions to preserve correct memory ordering. It consists of two separate circular queues: the Load Queue (LQ) and the Store Queue (SQ). Each load or store instruction dispatched from the decode stage allocates an entry in the respective queue, maintaining the original program order.

During execution, memory addresses and store data are resolved and written into the LSQ. For store-to-load forwarding, the LSQ checks for matching addresses between younger loads and older stores. If a match is found and the store data is ready, the load receives the forwarded value directly from the SQ without accessing memory. As opposed to speculative load operation, our in-order design has two opportunities for forwarding: when a store address is updated or when a load address is updated.

The LQ head is eligible for execution when its address is resolved, there are no older unresolved stores in the SQ, and forwarding is either unnecessary or has already occurred.

If forwarding has not occurred, a memory request via the data cache is issued and FU_2 writes the result to the X/C register. If forwarding has occurred, the data and tag are sent to the X/C register to be broadcast by the CDB.

Stores retire in-order through the ROB. Once a store reaches the head of the SQ and the ROB signals it as ready, FU_3 performs a write to memory using the resolved address and data.

The LSQ also handles control flow changes. On branch misprediction or halt instructions, the LQ clears all entries, and the SQ marks unretired entries as invalid. This ensures stale memory operations do not execute, preserving correctness.

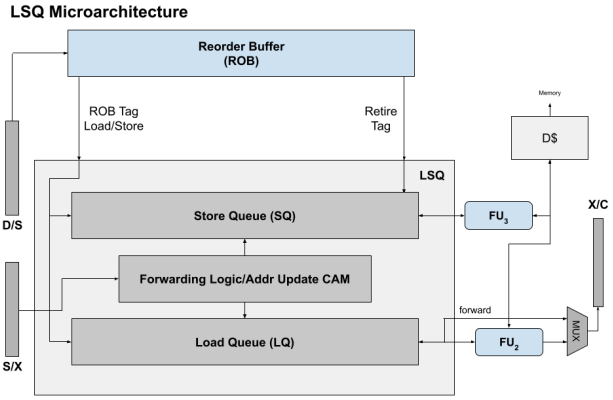

Fig. 2. A closer view of the LSQ microarchitecture and its interactions with other structures of the processor.

H. Branch predictor

The branch predictor consists of a PC-indexed 256-entry Branch History Table, and 2-bit saturating counters are used for branch predictor update. The branch predictor receives the packet from the decode stage and makes a prediction if it is a conditional branch instruction. The prediction result will be sent to the D_S_reg and passed down, and the target address is calculated immediately and sent back to the fetch stage.

The 2-bit saturating counter is updated after the execution stage. The result, along with the PC address, is sent back to the branch predictor for update.

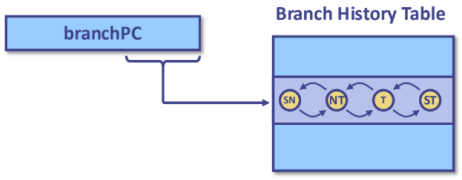

Fig. 3. Branch Predictor Structure [1]

I. Caches

Our design features two single-ported, direct-mapped, single-banked 256B caches. Each cache line is 8 bytes, and the caches can hold 32 lines simultaneously. Direct mapping associativity was selected because under sequential addressing with no branching, cache line collisions are infrequent. They are non-blocking designs featuring a parameterizable number of Miss-Status Handle Registers (**MSHR**) in an array. The icache and data cache (**dcache**) feature the same MSHR mechanism and fetch operation; the distinction between both modules is prefetching and storing, respectively. We will first explain the pipelined fetch system, which performs the loads, and then go into the extra features. As a note, the dcache and icache avoid structural hazards by giving the dcache priority whenever it attempts to perform an interaction.

On a load, the cache always returns valid_out along with the data if the requested address is stored in the cache; otherwise, the fetch process begins. The subsequent operations described are combinational logic and non-blocking, so the cache performs multiple or none of these operations

simultaneously for different addresses. First, the cache checks that the address is not in the MSHR and inserts it with an empty memory_tag if there is an available MSHR slot. If the cache detects a valid entry with a memory_tag == 0 in the MSHR, it flags a miss outstanding and attempts to make the memory request with the memory controller. While a miss is outstanding, the MSHR saves the corresponding address's memory tag with the response; if it is nonzero, the miss outstanding will not be triggered by this slot again. When an incoming tag matches an MSHR slot's saved memory tag, it updates the cache storage and clears the MSHR slot. This final step saves the data, and effectively, the valid_out will be true along with the proper data. The cache is non-blocking and can be pipelined because once an address enters the MSHR, a new request can be placed while the previous one gets serviced. Upon returning to the previous address, if it has been fetched a *valid out* is returned.

The icache and dcache use this system with extra functionality to serve the corresponding cache types.

The **icache** has a prefetching line, which is valid under the condition that:

A. IF stage address is in the cache

B. IF stage address is in the MSHR

The mechanisms from above hold, except the requested address switches to the IF stage address + 12. An offset of 12 was selected because the system is 8-byte aligned and the following reasoning:

Our request address will always be divisible by 8 or 4, so we will use the memory addresses 0, 4 to demonstrate the prefetching scenarios.

TABLE I

PREFETCHING OFFSET FOR A GIVEN MEMORY ADDRESS

	Memory Address		
Prefetch Offset	0	4	
+0	0 | 0 | 0	0 | 0 | 0	
+4	4 | 0 | 0	8 | 8 | 1	
+8	8 | 8 | 1	12 | 8 | 1	
+12	12 | 8 | 1	16 | 16 | 2	

Each slot shows:

mem address | aligned mem address | cache line

If we take the row with +0 offset to be the IF stage address being serviced, we can see that while we wait for the access to be completed the prefetch request is optimized for the next catch line most with +12. +16 would be too much as it would skip the next address and create a miss on the subsequent cache hit.

Our clock period results in a latency of about four cycles so this would create only a two cycle gap when we get to the next address.

The **dcache** supports writes, requiring only a slight addition to the cache system. The loads function as before without modification. If the store results in a cache miss, a load for the address is inserted into the MSHR; otherwise, the cache performs the following directly:

The cache always pushes the data of the current cache line out. This means that FU_3 is receiving the data and, after modification, can present it back. If the data is new, the cache updates its entry and marks the new *wr_cache* flag within the line. If the cache detects any entries awaiting a write, it loads them into the MSHR with the *BUS_STORE* memory command, which attempts to write to the controller. If the controller returns a valid response, the MSHR slot clears itself. During this process the *valid out* is low until the final MSHR clear.

IV. TESTING

All modules were initially tested using custom SystemVerilog testbenches for each unit. We created tasks to print internals and made assertion style checks. Through this we were able to visually and systematically confirm the modules worked as predicted.

Integration testing was then performed with a simplified version of the pipeline: featuring only the RS, MT, ROB, D/S registers and the CBD. The IF stage to D stage were also tested together in parallel with the other multi module testbench. Once these functioned the pipeline was constructed and verified for a single add instruction. We then wrote assembly files to force collisions we wanted to test and began debugging the pipeline using the program files. This way we could verify that our processor core both worked on the ISA as well as handled multiple instructions. We used the output files for register writes and memory writes (*.wb and *.out) from a verified in-order 5 stage pipeline in order to provide ourselves with ground truth comparisons.

Eventually we had all programs successfully passing the basic pipeline (highlighted in blue in Fig. 1). The original design did not feature the duplicated FUs. With the base pipeline completed we split design and verification of the advanced features with regular communication to ensure proper interfacing and expected behavior. The caches, branch predictor, and LSQ were created in parallel. They were individually tested through testbenches before pipeline integration. The advanced features were then combined and when all tests passed merged into the main branch.

We also created a visual debugger allowing the user to step through clock cycles and visualize the contents of the essential structures of the processor.

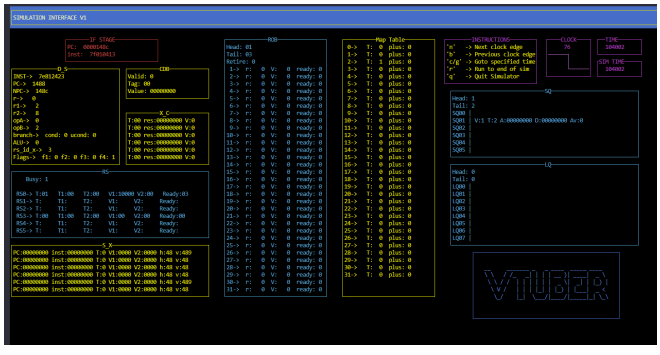

Fig. 4. Example execution of the visual debugger make alexnet.vis

V. Performance Analysis

The final performance of our out-of-order RISC-V P6 architecture is shown below, with an average **CPI of 3.99** and **clock period of 1.3 ns** or a **clock frequency of 769.2 MHz** passing every test program in synthesis and simulation, as well as our own additions for stress testing various instruction types and modules. The clock period was found by binary searching for the minimum clock period that did not cause timing violations. In addition, we did a parametric sweep of the ROB size, LQ size, and SQ size, and did not get a change in CPI for larger structure sizes. As a result we resulted in an ROB size of 32, LQ size of 8, and SQ size of 6.

TABLE II

Synthesized Processor Performance for all tests

```
========================================== SCOREBOARD ==========================================
Test Run: Thu May  1 22:38:09 EDT 2025                    Clock Period: 1300 ps = 1.300000 µs
Program             | RegCheck   | MemCheck   | Cycles     | Time (µs)  | CPI        | System Halt
----------------------------------------------------------------------------------------------
evens_long          |    PASS    |    PASS    | 737        | 958.1      | 2.200000   | WFI instruction
load_tests          |    PASS    |    PASS    | 88         | 114.4      | 4.190476   | WFI instruction
matrix_mult_rec     |    PASS    |    PASS    | 70388      | 91504.4    | 3.247278   | WFI instruction
fc_forward          |    PASS    |    PASS    | 19924      | 25901.2    | 2.960915   | WFI instruction
parallel            |    PASS    |    PASS    | 423        | 549.9      | 2.125628   | WFI instruction
omegalul            |    PASS    |    PASS    | 216        | 280.8      | 2.958904   | WFI instruction
graph               |    PASS    |    PASS    | 39838      | 51789.4    | 3.580944   | WFI instruction
copy                |    PASS    |    PASS    | 385        | 500.5      | 2.938931   | WFI instruction
branch_test         |    PASS    |    PASS    | 54         | 70.2       | 5.400000   | WFI instruction
basic_malloc        |    PASS    |    PASS    | 3360       | 4368.0     | 3.563097   | WFI instruction
insertionsort       |    PASS    |    PASS    | 90568      | 117738.4   | 3.081592   | WFI instruction
bfs                 |    PASS    |    PASS    | 11363      | 14771.9    | 3.262417   | WFI instruction
btest2              |    PASS    |    PASS    | 2664       | 3463.2     | 5.842105   | WFI instruction
ppln_test           |    PASS    |    PASS    | 22         | 28.6       | 3.666667   | WFI instruction
backtrack           |    PASS    |    PASS    | 25087      | 32613.1    | 3.483822   | WFI instruction
insertion           |    PASS    |    PASS    | 1954       | 2540.2     | 3.267559   | WFI instruction
btest1              |    PASS    |    PASS    | 1344       | 1747.2     | 5.843478   | WFI instruction
halt                |    PASS    |    PASS    | 11         | 14.3       | N/A        | WFI instruction
link_list           |    PASS    |    PASS    | 6634       | 8624.2     | 3.733258   | WFI instruction
ppl_test            |    PASS    |    PASS    | 32         | 41.6       | 2.666667   | WFI instruction
sort_search         |    PASS    |    PASS    | 574439     | 746770.7   | 3.156380   | WFI instruction
priority_queue      |    PASS    |    PASS    | 5412       | 7035.6     | 3.722146   | WFI instruction
store_tests         |    PASS    |    PASS    | 161        | 209.3      | 3.500000   | WFI instruction
fib_long            |    PASS    |    PASS    | 1031       | 1340.3     | 1.618524   | WFI instruction
quicksort           |    PASS    |    PASS    | 299604     | 389485.2   | 3.138200   | WFI instruction
malloc_test         |    PASS    |    PASS    | 909        | 1181.7     | 3.710204   | WFI instruction
fib_rec             |    PASS    |    PASS    | 58508      | 76060.4    | 4.893201   | WFI instruction
mult_test           |    PASS    |    PASS    | 19         | 24.7       | 9.500000   | WFI instruction
mem_test            |    PASS    |    PASS    | 82         | 106.6      | 7.454545   | WFI instruction
fib                 |    PASS    |    PASS    | 390        | 507.0      | 2.617450   | WFI instruction
c_test              |    PASS    |    PASS    | 413        | 536.9      | 3.226562   | WFI instruction
no_hazard           |    PASS    |    PASS    | 12         | 15.6       | 12.000000  | WFI instruction
dft                 |    PASS    |    PASS    | 186611     | 242594.3   | 3.224492   | WFI instruction
dfs                 |    PASS    |    PASS    | 12099      | 15728.7    | 3.323901   | WFI instruction
raw_program         |    PASS    |    PASS    | 18         | 23.4       | 6.000000   | WFI instruction
if_stage_test       |    PASS    |    PASS    | 20         | 26.0       | 5.000000   | WFI instruction
outer_product       |    PASS    |    PASS    | 2194023    | 2852229.9  | 2.940490   | WFI instruction
evens               |    PASS    |    PASS    | 420        | 546.0      | 4.285714   | WFI instruction
easy_program        |    PASS    |    PASS    | 36         | 46.8       | 4.000000   | WFI instruction
more_mult           |    PASS    |    PASS    | 68         | 88.4       | 7.555556   | WFI instruction
haha                |    PASS    |    PASS    | 51         | 66.3       | 3.000000   | WFI instruction
alexnet             |    PASS    |    PASS    | 633700     | 823810.0   | 3.031071   | WFI instruction
saxpy               |    PASS    |    PASS    | 584        | 759.2      | 3.139785   | WFI instruction
mult                |    PASS    |    PASS    | 1283       | 1667.9     | 3.947692   | WFI instruction
sampler             |    PASS    |    PASS    | 379        | 492.7      | 3.477064   | WFI instruction
copy_long           |    PASS    |    PASS    | 1003       | 1303.9     | 1.697124   | WFI instruction
mergesort           |    PASS    |    PASS    | 34072      | 44293.6    | 3.593714   | WFI instruction
mult_no_lsq         |    PASS    |    PASS    | 898        | 1167.4     | 3.184397   | WFI instruction
Average CPI = 3.99        Average Time = 5.197400 µs
================================================================================
```

A. *Branch Predictor Analysis*

As shown in the table, our branch predictor has an average of 0.17 improvement compared to the original always-not-taken strategy. The below table implies that the always-taken strategy provides an improvement over the branch history table.

TABLE III

Branch Predictor Performance comparison for .c and .s

Branch Predictor Type	Average CPI
Always not taken	4.16
Always taken	3.95
Branch History Table	3.99

However, this average is determined from all of our test programs (.c and .s). After testing our .c files, which have more complex operations and longer execution time it is evident that our branch history table design provides a performance boost over the always taken strategy, with similar or lower CPI for 29/47 of our test programs.

TABLE IV

Branch Predictor Performance for .c files

Branch Predictor Type	Average CPI
Always not taken	3.61
Always taken	3.35
Branch History Table	3.31

B. *Memory Analysis*

Comparing CPI files between milestone branches for the same set of .s and .c programs results in the below table.

TABLE V

Memory Feature Performance Metrics for assume branch not taken (.s and .c files)

Caching	LSQ	Prefetch	Average CPI
			7.59
X			6.49
X	X		5.54
X	X	X	4.16

In particular, the cpi for copy.s improved from 7.84 to 4.8 with the addition of the LSQ and the forwarding it implements.

References

[1] Khan (2025). *Branch Prediction* [PowerPoint slides]. Columbia University.