CSEE4824 Final Project Report
Out-of-Order RISC-V P6 Processor

Tanvir's Angels: Joseph Han (JH4632), Evan Battaglia (EB3504), Yiyang Peng (YP2655), ,
Jiamiao He(JH4593), Rey Qin(RQ2190),

Abstract

This report provides a comprehensive analysis of the architectural design and performance enhancements of our latest processor. Initially, we delve into the fundamental design choices and present a detailed block diagram to outline the structural configuration of the processor. We further examine the baseline features which include Reservation Stations (RS), Reorder Buffer (ROB), the Mappable feature, Common Data Bus (CDB), Dispatch mechanism, and the Functional Units & Execution (FU & Ex) schema. Building upon the foundational design, we introduce advanced functionalities such as Early Branch Resolution and Branch Prediction (BP), supplemented by a Visual Debugger that facilitates real-time monitoring and debugging of the processor. The analysis section quantifies the performance improvements attributable to the early branch resolution and enhanced branch prediction mechanisms, providing empirical data and performance metrics to substantiate the enhancements. The conclusion synthesizes the findings, highlighting the processor's increased efficiency and predictive accuracy, marking significant advancements over traditional designs. This holistic review not only underscores the technical prowess of our processor design but also sets a precedent for future developments in processor architecture.

1 Overview

1.1 Features

- P6 Architecture, out-of-order processor with 2 ALUs, 2 4-cycle multipliers and 1 load and 1 store unit
- Global branch predictor
- Early branch resolution
- Both a basic p6 design where branches are resolved in the reorder buffer AND a design with early branch resolution
- Pipelined fetch to handle memory latency
- GUI debugger
- Passes all tests

1.2 Block Diagram

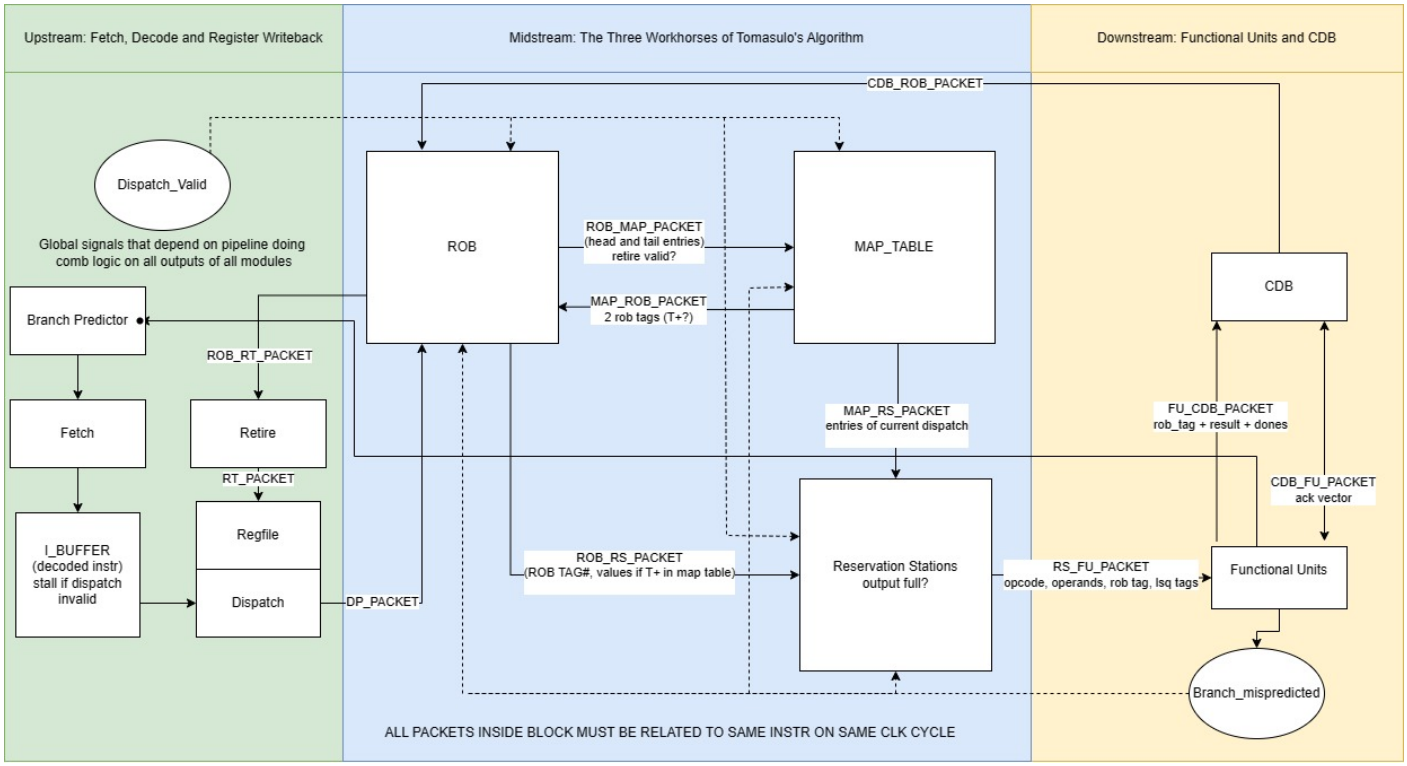

Figure 1: Block Diagram of our Design

1.3 CPI Analysis

Table 1: Baseline Version of OoO

Program Name	Output CPI
btest1	2.591304
btest2	2.583333
copy_long	1.908784
copy	3.939394
evens_long	1.860119
evens	2.969697
fib_long	1.691223
fib_rec	2.988375
fib	2.593333
haha	2.444444
halt	inf
insertion	2.671119
mult_lsq	3.733129
mult_no_lsq	2.816254
no_hazard	2.230769
parallel	3.270000
sampler	2.154545
saxpy	2.973262
alexnet	2.462364
backtrack	2.964038
basic_malloc	3.313559
bfs	2.816590
dft	2.708125
fc_forward	2.307875
graph	3.195039
insertionsort	2.804837
matrix_mult_rec	4.001522
mergesort	3.117591
omegalul	2.459459
outer_product	2.423888
priority_queue	3.365636
quicksort	2.524107
sort_search	2.711436

Table 2: with Early Branch Resolution

Program Name	Output CPI
btest1	2.155844
btest2	2.153173
copy_long	1.493243
copy	3.575758
evens_long	1.294643
evens	2.565657
fib long	1.236677
fib rec	2.529436
fib	2.406667
haha	2.166667
halt	inf
insertion	2.168614
mult_lsq	3.334356
mult no lsq	2.505300
no hazard	1.714286
parallel	2.970000
sampler	1.836364
saxpy	2.475936
alexnet	2.281869
backtrack	2.840044
basic malloc	3.076271
bfs	2.572618
dft	2.462678
fc forward	1.897623
graph	2.949847
insertionsort	2.591571
matrix mult rec	3.587904
mergesort	2.947796
omegalul	2.067568
outer product	2.132559
priority queue	3.116838
quicksort	2.179751
sort search	2.544485

Table 3: Advanced Branch Predictor

Program Name	Output CPI
btest1	2.155844
btest2	2.153173
copy long	1.403716
copy	3.10687
evens long	1.190476
evens	2.565657
fib long	1.169279
fib rec	2.531945
fib	2.187919
haha	2.166667
halt	inf
insertion	2.133779
mult lsq	3.178462
mult no lsq	2.388693
no hazard	1.714286
parallel	2.738693
sampler	1.836364
saxpy	2.150538
alexnet	2.221415
backtrack	2.706054
basic malloc	3.048729
bfs	2.467566
dft	2.366037
fc forward	1.492868
graph	2.801007
insertionsort	2.114432
matrix mult rec	3.567007
mergesort	2.904451
omegalul	2.067568
outer product	1.85514
priority queue	3.068041
quicksort	1.815756
sort search	2.205622

2 Base Design Details

2.1 Upstream: Fetch, Branch Prediction, Decode, Dispatch

2.1.1 Instruction Fetch and Buffer

The instruction fetch stage provides a program counter and memory interface. The program counter is advanced when the memory is free to access and there is space in the instruction

buffer, and can be set directly by branch instructions. To account for memory latency, a system of memory tags should have been used. Instead, due to time constraints, a less flexible system was used that depends on a static, known memory latency. In this system, memory addresses are sent off identically to the case of no latency, but a history of memory responses and valid instruction bits is kept in a shift register. Instructions are read from the history index corresponding to the known memory latency to ensure that read instructions correspond to the correct memory address and valid bit.

The instruction buffer is a FIFO holding up to sixteen (16) valid instructions. This allows instructions to continue to be fetched regardless of valid dispatches. The buffer is considered full for the purposes of the fetch stage when it contains less free space than the necessary memory latency history, preventing the program counter from advancing in situations that could overflow the buffer.

2.1.2 Dispatch

This module is responsible for preparing instructions received from the instruction buffer and dispatching them to the appropriate execution units.

Register File Interaction
The module interfaces with the regfile submodule that handles register reads and writes. It fetches source operand values (rs1, rs2) based on indices provided by the instruction packet (ib dp packet). It writes back results to the register file as specified by the rt dp packet, enabling updates to register values after instruction execution (write-back stage).

Decoder
A decoder submodule is utilized to decode various fields of the instruction and set control signals accordingly. This includes:

- Operation types (arithmetic, memory access, branch, etc.).
- Operand sources and destinations.
- Special operations like memory read/write, and branching conditions.
- Setting flags for valid instruction and specific functionalities (e.g., has dest for destination register determination).

The always ff block manages the halted state, which is crucial for controlling the processor's operation in response to halt conditions or pipeline squash signals. The processor can halt if there's a valid halt condition detected by the decoder and dispatch valid is asserted. Squash or reset signals can reset the halted state, useful in flush or stall scenarios.

The module assigns various decoded and fetched information to the dp packet output, preparing it for subsequent stages in the pipeline. This includes: The instruction itself, next and current program counters (NPC, PC). Source and destination register indices and values. Predicted branch outcomes and other control signals necessary for execution. Ensuring that all relevant flags and fields are populated based on the decoded instruction and operational needs.

2.2 Midstream: Tomasulos Algorithm

2.2.1 Reservation Station

The reservation station keeps a record of the valid instructions dispatched that are currently in flight, and serves as the gatekeeper that prevents more instructions than the currently available number of functional units from being dispatched, while freeing up promptly so that the next valid instruction can be dispatched. In our implementation it also gate keeps around store instructions to prevent irrevocable memory writes and read-after-write hazards. For our implementation, there are 2 RS entries corresponding to the 2 ALU functional units, 1 load and 1 store entry, and 2 multiplier entries.

The reservation station functionality is divided into four actions: allocate, update, free and squash.

On allocation, the incoming instruction is only dispatched when there is a free RS entry that corresponds to the type of the functional unit associated with the instruction ready to be dispatched. The RS checks the MAP_RS_PACKET for 1) the validity 2) the ROB tag corresponding to the two operands of the instruction allocated. If the operand is invalid, i.e. not used for example in an addi instruction, the operand is ignored and the RS waits for the other operand to become ready to issue the instruction. If an operand is waiting for a specific ROB entry to complete, the RS records a non-zero ROB tag. When both operands are ready and both t1 and t2 are zero, the instruction is issued to the downstream. It is worth noting that on allocation, the register value for the two operands can come from 3 places: the regfile, the data being broadcast on the CDB, and the ROB keeping a temporary record of the completed, but not yet retired instruction. Each RS entry is associated with one ROB tag, which is address of the instruction in the ROB. At any time such ROB tags in the RS are unique, as the address of the same instruction in the ROB is unique. This will have important implications for freeing and squashing.

On update, the RS updates the operand value of all its entries, taking the value from the common data bus when such data are available. When both valid operands are ready (invalid operands are assumed to be ready), the instruction is issued to the downstream.

On free, the RS clears the entry corresponding to an instruction that is done at the Functional Units. Generally the RS waits for CDB acknowledgement of a finished instruction to free/clear its corresponding RS entry according to the ROB tag acknowledged by the CDB. The RS takes the ROB tag broadcast by the CDB, find its corresponding entry in the RS and frees that entry. As an optimization for ALU instructions, freeing happens when an instruction finishes computation and the result of the ALU computation is stored in a pipeline register waiting for CDB to acknowledge, not when the CDB acknowledges it some cycles afterwards.

On squash, the RS clears all entries that are allocated after a mispredicted branch instruction resolved in the functional units. This is done by comparing the ROB tag associated with each RS entry.

Another important optimization allows an RS slot to be freed and allocated on the same clock cycle. A previous ALU instruction that has been computed will be freed from the RS, and if at the same positive clock edge an ALU instruction is ready for dispatch, the RS allocates that later instruction at the same positive clock edge.

A design-specific implementation is that the RS turns the dispatch_valid signal low if there is a store instruction in flight in the RS. This is to ensure that any load from the same memory location as the store is dispatched after, not before the store is completed to prevent hazards in this out-of-order processor.

2.2.2 Reorder Buffer

The ROB facilitates the management of instruction execution, ensuring correct program order by temporarily storing instructions as they are issued and completed, but not yet committed. This module interfaces with several other components like the instruction buffer, dispatch unit, and completion data bus (CDB), handling dynamic scheduling complexities through a FIFO (First-In, First-Out) buffer mechanism.

Key Components and Functions ROB Entries (ROB Memory): rob memory: An array representing the FIFO structure of the ROB, with a size defined by ROB SZ. Each entry (ROB ENTRY) in rob memory includes details about the instruction, its destination, and its completion status.

Pointers: head and tail: These pointers manage the FIFO structure, indicating the next positions for retirement and entry insertion, respectively. Input/Output Packets: Inputs include packets from the mapping table (MAP ROB PACKET), dispatch packet (DP PACKET), and completion data from the CDB (CDB ROB PACKET). Outputs are packets directed to the reservation station and map table, as well as retired in- structions to the dispatch module. Operational Logic Combinational Logic: Default Settings: Initially sets all outputs to zero. New Entry Preparation: A new tail entry is prepared with data from the instruction buffer and updated to the tail of the ROB.

Map table and Reservation Station Updates: The ROB sends necessary data to the map table and reservation station based on dependencies and the availability of source operands. Sequential Logic: Reset: Clears all entries and resets pointers upon a reset signal. Branch Misprediction: In case of a misprediction, the ROB clears all entries past the mispre- dicted branch, adjusting the tail pointer accordingly. Retirement Logic: Instructions that are completed and at the head of the queue are retired and removed from the buffer. Dispatch Logic: If there is space (ROB not full) and dispatch signals availability, new instruc- tions are added to the ROB. Completion Data Bus Handling: Updates completion status of instructions based on feedback from the execution units through the CDB. 5

Flags and Status Checks Full and Empty Conditions: The full flag is set when the head equals the tail and the entry at the head is valid, or under specific conditions when handling store operations. The empty flag checks if the head equals the tail and the entry at the head is invalid, indicating no pending instructions

2.2.3 Map Table

The map table module maintains a reference of each register and the most recent ROB tag corresponding to the instruction that will require a write-back, as well as a tag indicating if the instruction has been computed and is thus available in the ROB, but has not yet been retired. These fields are cleared when that ROB tag has been retired, but can also be overwritten by

newly dispatched instructions at any time.

Each time a branch instruction is encountered, the map table generates a snapshot that continues to be updated and cleared, but will not be overwritten by new instructions. Upon a branch misprediction, the map table will revert to the corresponding snapshot.

2.3 Downstream: Functional Units, CDB

2.3.1 Functional Units

Four types of functional units are used in this design: arithmetic-logic-units (ALUs), stores, loads, and multipliers.

ALU

- Entry Points: Processes inputs via `FU_IN_PACKET` and `ALU_FUNC`, handling operational codes and data for computations.
- Data Structures: Utilizes logic and reg types for data storage and manipulation, including operational multiplexers and an ALU.
- Combination/Sequential Logic: Employs combinational logic for data manipulation and sequential logic tied to the clock for state updates and handling output packets.

Operation Logic

- Functionality: Core operations include arithmetic and logical functions, alongside integrated branch condition evaluations within the functional unit.
- Unique Features: Integrates conditional branch evaluations, reducing hardware requirements and streamlining data paths compared to standard designs.

Data Path

- Inter-module Communication: Interacts with the register file for data fetching and outputs results to the common data bus or subsequent pipeline stages.
- Control Flow: Manages data flow through control signals like `block` and `ack`, crucial for pipeline stalls and operation completions.

Control Flow

- Handling Squash: Designed to reset internal states and outputs effectively in response to pipeline flush or squash signals.
- Acknowledgment Handling: Uses the `ack` signal to confirm successful data reception by subsequent stages, ensuring proper operation completion.

Detailed Description of Key Components

- ALU Operations:
  - Supports various operations crucial for computation and control flow.
  - Dynamically selects operations based on the `func` signal, influencing processor state and branch logic.
- Conditional Branch Module:
  - Evaluates branch conditions and integrates within the ALU module to provide real-time decision capabilities.
  - Outputs affect branch decisions and are integral to pipeline control signals.
- Output Handling:
  - Outputs results and status flags via `FU_OUT_PACKET`, critical for downstream processing and state updates.
  - Manages outputs under normal and special conditions, including mispredictions or branch resolutions.

Module Specifications

- Global Signals: Utilizes `clock` and `reset` for timing and initialization.
- Control Signals: Includes `ack` for acknowledging completion to the Common Data Bus (CDB).

Functionality and Operation Logic

- Inputs and Outputs: Processes inputs through `FU_IN_PACKET` and outputs results via `FU_OUT_PACKET`.
- Multiplier Control Logic: Converts the `issue_valid` signal into a pulse to start the multiplication, ensuring that the operation starts only once per valid signal pulse.

Mult FU

- Signal Initialization and Management: Maintains internal signals to manage the state and progress of the multiplication operation, such as `start`, `mult_done`, and `fu_done`.
- Multiplier Instantiation: The module includes a multiplier unit (`u_mult`) that performs the multiplication operation. It handles both signed extensions of the operands and captures the full product of the operation.

Control Flow and Output Handling

- Handling of Start and Completion: Controls the start of multiplication based on input packet validation and manages the completion state to synchronize with the pipeline and CDB.
- Result Assignment: Conditions the output based on the function type specified in the instruction, selecting the appropriate half of the product result for output.

Acknowledgment and Reset Behavior

- Reset and Acknowledgment Logic: Implements robust control to reset internal states upon a reset signal and to handle acknowledgment signals appropriately to clear the done status.

2.3.2 Common Data Bus

The common data bus (CDB) is used to pass information between the various functional units and the upstream modules (reservation station, map table, and reorder buffer). The CDB includes a priority selector to choose between the various functional units that may be attempting to broadcast completed instructions. When the priority selector chooses a functional unit, it sends back an acknowledge pulse that clears the functional unit's done bit, allowing the functional unit to accept a new instruction. Simultaneously, the functional unit's output is broadcast to all upstream modules along with the corresponding ROB tag.

3 Advanced Features

3.1 Early Branch Resolution

The purpose of the early branch resolution is to resolve a mispredicted branch at the Functional Unit stage, rather than some clock cycles later when the branch instruction's corresponding ROB entry is getting retired. This leads to a CPI reduction especially in branch-heavy algorithms.

The challenge of implementing early branch resolution is when multi-cycle instructions that are still in flight are overwritten in the map table by a newly dispatched instruction. When the branch is resolved, the map table entry corresponding to the overwriting instruction is cleared, but the overwritten instruction is not restored in the map table, giving rise to a host of buggy behaviors. For one example, with the overwritten, still-in-flight mult instruction overwritten and never restored in the map table, the RS when allocating a new instruction and looking up the map table would potentially not find that the overwritten mult instruction is complete when it needs the result from the overwritten instruction for a newly allocated instruction in the RS, causing the whole program to grind to a halt.

The solution is to keep back-up snapshots of the map table whenever a new branch instruction is dispatched, in the event that such a branch instruction is mispredicted and we need to recover the state of the map table right before the branch is resolved. The snapshots of the map table can be updated (in the sense of turning T_plus high) but never overwritten, so that

any overwriting instruction that comes after the branch instruction will have no effect on the snapshot taken, and we can restore the state of the map table to the state right before the branch is resolved.

There may be multiple branch instructions so we created a heap of such snapshots. Each snapshot in the heap works independently - the snapshot corresponding to an earlier branch instruction restores the map table to an earlier state when the branch turns out to be mispredicted, and a later snapshot restores the map table to a later state. The design is fool-proof in the sense that an earlier snapshot always "squashes" map table entries of later instructions, but a later snapshot can never "corrupt" or "over-delete" map table entries allocated earlier in time, ensuring the correctness of the squashing mechanism on branch misprediction because we delete and restore just the right number of map table entries.

For our specific, optimized design we choose to keep at most 4 non-empty snapshots of the map table at any point in time. This is because at any time there can only be 2 ALU instructions in the RS and 2 ALU instructions in the post-functional unit pipeline registers waiting to be acknowledged, which is a maximum total of 4 branch instructions.

3.2 Global Branch Predictor

Branch Target Buffer
The BTB stores information about the branch target addresses so that the processor can predict the target of branch instructions before these are actually resolved. The BTB uses partial tagging and indexing to conserve space while trying to maintain accuracy by checking a portion of the PC and storing validity bits. The operation of this BTB heavily relies on proper synchronization with the pipeline stages and correct handling of control signals (wr en, reset) to ensure that it provides accurate predictions and updates based on dynamically changing branch behavior.

BTB Memory Structure
A memory array of 256 entries, where each entry is 22 bits wide. The BTB uses this memory to store partial information of both the index PC (ex pc) and its corresponding target PC (ex tg pc).

A validity array indicating whether a particular entry in the BTB is valid.

Operation
On reset, all entries in mem are set to zero and all valid bits are cleared, indicating that all BTB entries are initially invalid.

When wr en is high, the BTB entry corresponding to the index derived from ex pc[2 +: 8] is updated.

The memory stores 10 bits of the ex pc starting from bit 10 and 12 bits of the ex tg pc starting from bit 2. The index bits ex pc[2 +: 8] select which one of the 256 entries to update.

The corresponding valid bit for the entry is set to 1, marking it as valid.

For any given if pc, the BTB attempts to predict the target PC.

It uses the bits if pc[9 -: 8] to index into the BTB. This index is derived by using bits 9 through

0 of if pc but reversing the range to select bits from 9 to 2. The predicted PC (predict pc out) is constructed by concatenating the upper unused bits of if pc, the 12 lower bits of the BTB entry, and two zero bits (since the target addresses are word-aligned).

Hit Detection
A BTB hit occurs if the 10 bits of if pc starting from bit 10 match the corresponding 10 bits stored in the BTB entry and the valid bit for that entry is set.

This comparison ensures that the BTB prediction is for the correct instruction, not just any instruction that might have mapped to the same index due to the limited number of bits used for indexing.

Branch History Table
The BHT is used to track the behavior of branches (whether they were taken or not) to improve the prediction accuracy of branch instructions. This module implements a fundamental part of dynamic branch prediction mechanisms, helping the processor make educated guesses about branch behavior and thus improving the efficiency of the instruction pipeline by reducing stalls related to branch resolution. The design is efficient and straightforward, using minimal resources to provide a mechanism that can significantly impact the overall CPU performance in branch-intensive applications. This form of branch prediction table is particularly useful in scenarios where branches show consistent behavior over time.

BHT Array and Indexing
bht[255:0]: This is an array of 256 3-bit entries where each bit represents a recent history of branch taken or not taken decisions. Each entry corresponds to a particular branch instruction, indexed by bits of the PC. tail and head: These are 8-bit indices calculated from the lower bits of the execute and fetch stage PCs, respectively. The ex pc[2 +: 8] and if pc[2 +: 8] notation extracts 8 bits starting from bit 2 of ex pc and if pc, which forms an index into the BHT.

Operations
On a reset, the BHT entries are all set to 0, clearing any previous history to start afresh.

When wr en is enabled, indicating that an update can be made based on the most recent branch outcome: The BHT uses tail as the index, which is derived from the execute stage PC (ex pc).

The least significant two bits of the selected BHT entry (bht[tail][1:0]) are shifted left, and the new take branch value is inserted as the least significant bit. This effectively records the most recent branch history, discarding the oldest record (the most significant bit before the shift).

For the instruction fetch stage (if pc), the module outputs the current state of the BHT entry indexed by head.

For the execute stage (ex pc), it outputs the state of the BHT entry indexed by tail.

Use in Branch Prediction
The 3-bit output (bht if out and bht ex out) typically represents a simple branch prediction history. The value could be used to determine the likelihood of a branch being taken based on its recent history. More bits allow finer-grained prediction strategies, such as 2-bit saturating counters per entry, which can better tolerate occasional mispredictions.

Pattern History Table
Our PHT is designed to enhance branch prediction accuracy in a processor by utilizing a two-

level adaptive predictor approach. The PHT works alongside a Branch History Table (BHT), taking advantage of history patterns to predict future branches more accurately.

The module operates as follows:

The PHT maintains states using a state table (state) defined by PHT STATE, which is not explicitly detailed in the code but typically represents a finite state machine for each PHT entry. Common states in a branch prediction context are NT STRONG (not taken, strong confidence), NT WEAK (not taken, weak confidence), T WEAK (taken, weak confidence), and T STRONG (taken, strong confidence).

tail and head are calculated from the lower bits of ex pc and if pc, respectively, using a specific range of bits ([2 +: 8]). These serve as indexes into the PHTs main array.

The n state array is a combinatorial intermediary that stores the next state of the PHT based on the current state and whether the branch was taken. The state transitions follow a typical two-bit saturating counter logic: Move towards stronger confidence in the direction indicated (taken or not taken). If the prediction was correct, strengthen the confidence; if incorrect, weaken it.

On the positive edge of the clock and when wr en is enabled, the state for the selected entry is updated from n state. The prediction output, predict taken, is determined by checking if the current state of the indexed entry (state[head][bht if in]) corresponds to either of the taken states (T WEAK or T STRONG).

Upon reset, all entries are initialized to NT WEAK, which implies a conservative start by assuming branches are not taken but with low confidence.

4 Development and Testing Strategies

4.1 Design compromises to get working base version

We adopted the following design compromises which may result in higher CPIs during the implementation process to get a base version of the processor working. We later unwound each of the compromises with significant CPI reduction.

- Enabling only 1 ALU and 1 RS corresponding to the ALU to avoid multiple branching instructions executed on the same clock cycle to avoid one major possible bug
- Stalling all instructions whenever there is a store instruction in the Reservation Station (to avoid read-after-write dependencies)
- Depending on CDB acknowledgement to free the corresponding RS entry, which causes the CPI for any instruction (including ALU instructions) to be at least 2 cycles, one cycle for computation, one cycle for being acknowledged by the CDB. The RS is not freed and therefore not available for the next instruction throughout these 2 cycles or potentially more.
- Always assume branch is not taken to avoid dealing with the specific implementation of branch predictor until everything else is working, which potentially resulted in a lot of

mispredictions and squashing.

4.2 Optimizations

- Enabled both ALUs after the base version of the processor works and after we figured out branch resolution logic at the ROB retirement stage.
- We optimized the RAW hazard elimination mechanism by only stalling the dispatch when 1) there is a store instruction in the Reservation Station and 2) the instruction being dispatched is a load instruction
- Optimized the ALU functional units such that a free reservation station signal is sent out to the RS whenever an ALU instruction finishes computation and is able to be stored in the pipeline reigster without overwriting some result yet to be acknowledged by the CDB. This in principle enables us to free the RS entry for an ALU instruction in 1 clock cycle after it's issued at the fastest, instead of 2.
- We wired in the branch predictor and made relevant changes to the control flow of related modules (namely, instruction buffer, dispatch, RS, ROB, Map Table, execution stage) to handle squash upon misprediction.

4.3 Testing Strategies

- When writing up individual modules for Tomasulo's algorithm, we started with the timing slides from the lecture as a standard benchmark and wrote individual testbenches for each of the major modules (ROB, RS, Map Table and CDB).
- We prioritized the development of a full pipeline over details of memory latency to get measurable results early on
- After the base pipeline (with design compromises) are written up, we created a visual debugger to track major signals from all modules on a cycle-to-cycle basis. The development of a visual debugger was a instrumental step in our development process which have saved us at least 50 hours of looking at text dump files to debug.
- With the visual debugger up, we developed a fool-proof workflow to handle every possible bug. The workflow goes as follows:
- Compile the test program and compare register writeback differences (diff correct_out/test.wb output/test.wb)
- Find line number of the first instance of difference, and plug that into the pipeline_test.sv file to generate the exact clock cycle number in the wrong program where this happens
- Using the visual debugger, go to the clock cycle number generated in the previous step and compare it with the visual debugger in the correct case with its clock cycle number generated in the same way.
- For programs that do not terminate, quit the hanging compilation and repeat the above 3 steps. The only way for the compilation to freeze is when the pipeline is "clogged" without a halt signal, where the cycle number keeps increasing but either fetch valid or dispatch valid stays low.

5 Analysis

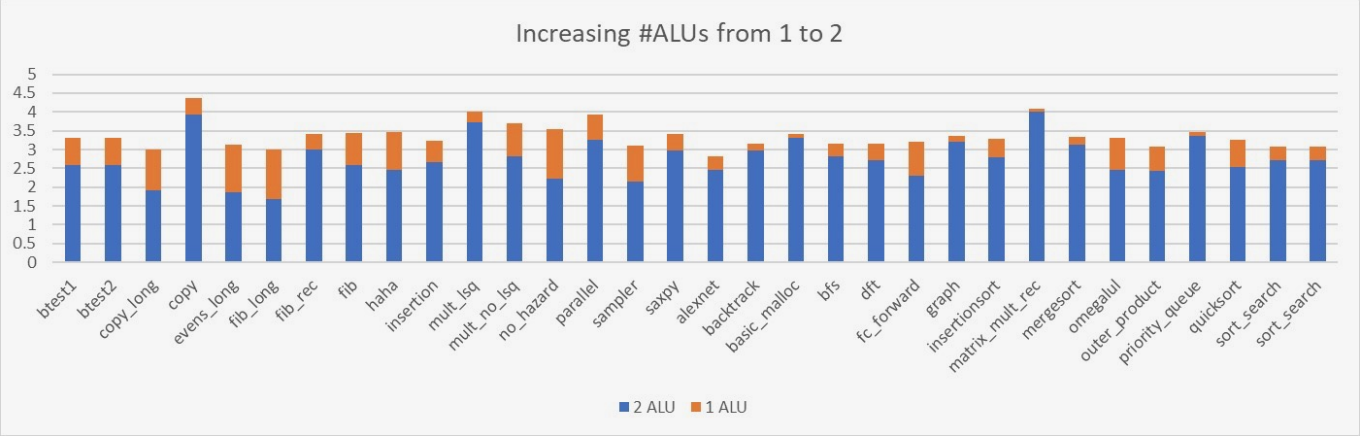

Figure 2: CPI change as #ALUs increase from 1 to 2

Increasing the number of ALUs had drastic impacts on CPI, in some cases improving CPI by almost 50% in the case where ALU instructions form the majority of the program, for example fibonacci algorithms. In memory manipulation programs like basic_malloc and multiply-dominated programs like matrix_mult_rec, the speed-up is not as significant. For realism of design, we did not increase the number of ALUs further, but with more advanced processes there is room to further increase the number of ALUs to bring down CPI.

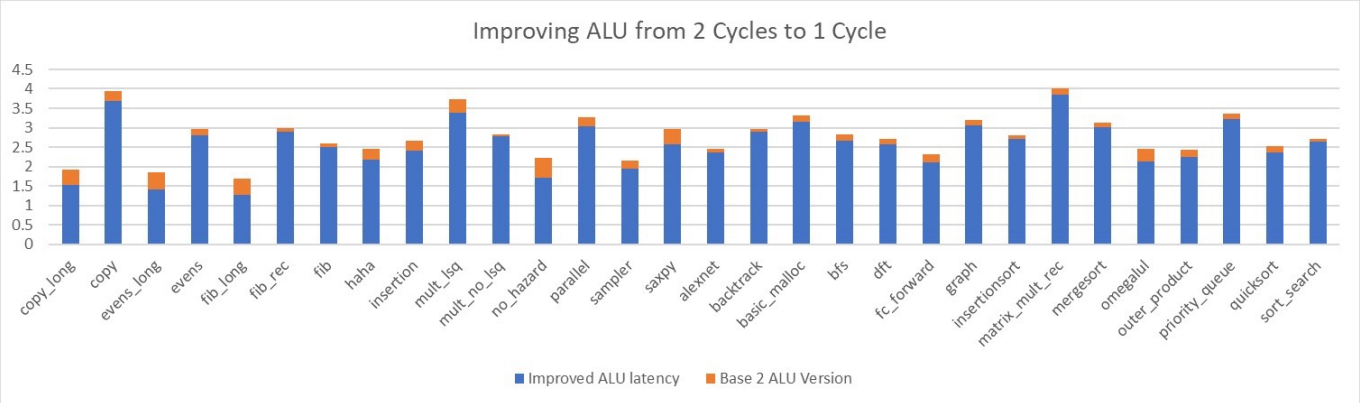

Figure 3: CPI change as ALU operational cycles decrease from 2 cycles to 1 cycle

Reducing the latency of ALUs had less than expected improvement to CPI, and the improvements are the most drastic in programs like no_hazard and fib_long where there are consecutive ALU instructions back-to-back. In programs "naturally pipelined" at the software level where there are not as many back-to-back ALU instructions, the speedup from better ALU latency is negligible, because there is not necessarily always an ALU instruction waiting on an ALU instruction to finish.

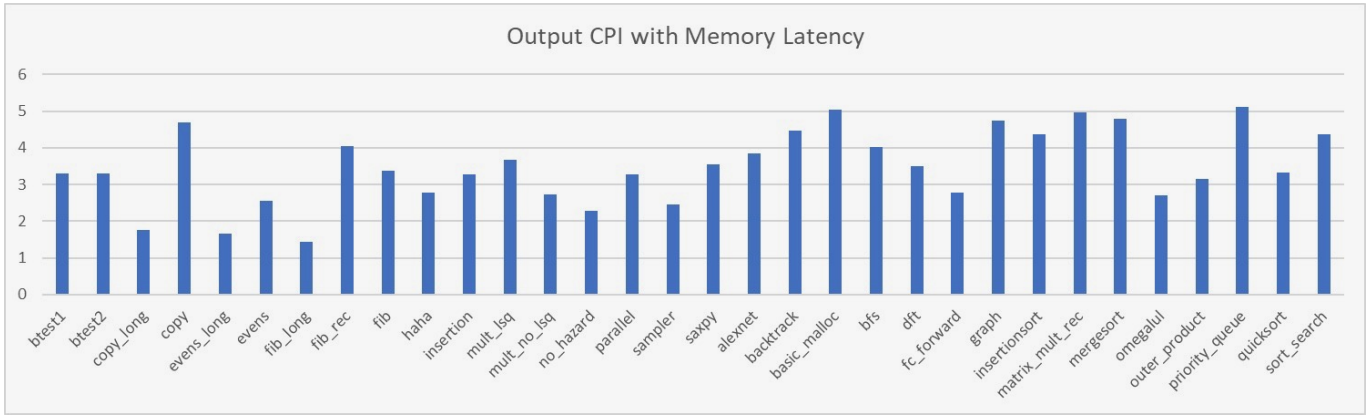

Figure 4: Output CPI with Memory Latency of 100ns

Including multi-cycle memory latency had the most significant impacts on programs with heavy memory accesses, most notably copy and basic_malloc. The fallout on CPI is contained because instruction fetch is pipelined so whatever increase in memory latency at the fetch stage is amortized over thousands of cycles. It is only in store and load functional units where instructions take the full multi-cycles to complete and the overhead cannot be amortized. In programs with less memory accesses such as fib_long, the slowdown is almost negligible.

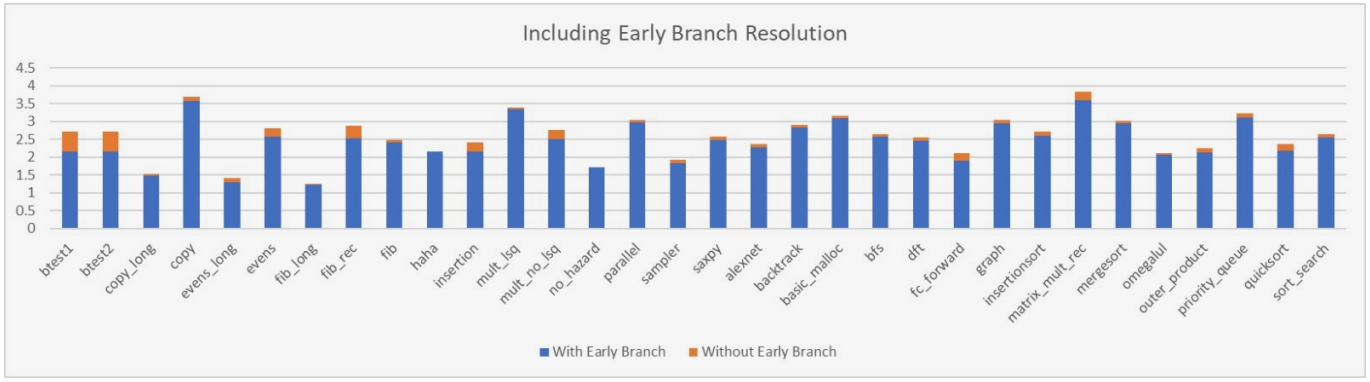

Figure 5: Output CPI with early branch resolution

Early branch resolution had the most impact on programs with large number of branches, namely btest1 and btest2. The speedup is as high as 20% in those cases. The reduction in number of cycles needed to complete the program is equal to number of mispredictions $\times$ average number of cycles between ALU resolves the branch mispredicted and the corresponding ROB entry getting retired. For programs with fewer branches (and hence fewer mispredictions) and fewer hazards (therefore shorter ROB queue), the speedup is most negligible, most notably in no_hazard where speedup is 0.

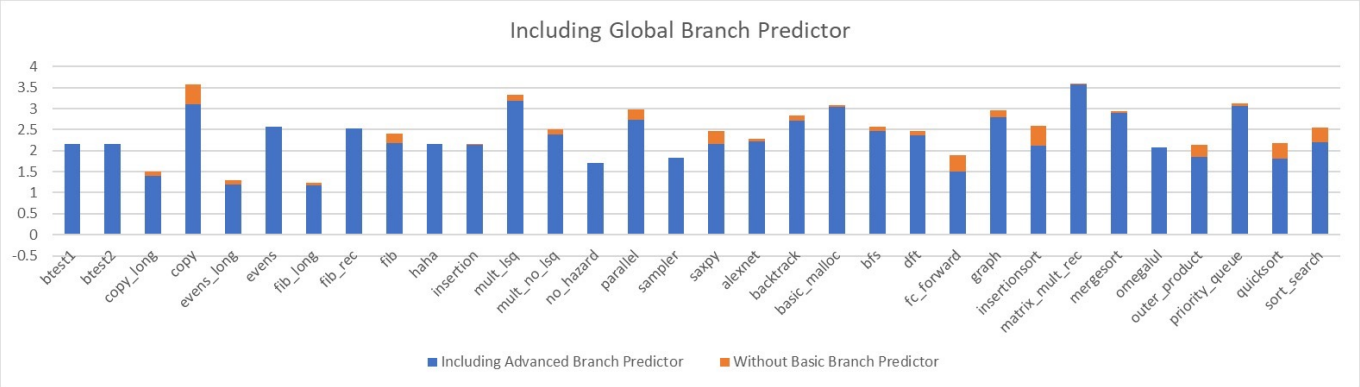

Figure 6: Output CPI Global Branch Predictor

Programs with significant improvements in CPI is due to the programs with frequent branching. The speedup is as high as 36%. The percentage reduction in number of cycles is a rough estimation of branch instruction to non branch instruction ratio. The branch predictor ultimately predicts correctly. For programs with invisible improvements show that either branch patterns are hard to train and observe before the program ended being randomized branching, or not enough instructions to create a reliable pattern for the branch predictor to follow. Otherwise, large improvements can show our branch predictor's accuracy.

6 Additional Notes

6.1 Memory Latency

Memory latency was initially ignored in favor of finishing a working base version of the processor for testing. Due to a lack of synthesis, a 30ns clock period was assumed as standard, which, when combined with the given 100ns of memory latency, results in a four cycle delay. The most correct way to handle this latency is to make use of the memory latency tag system, wherein each memory request is given a zero latency identifier that is used to track the request and match data from the memory to specific address requests. In addition, a cache system should have been used, wherein atomic memory accesses are forbidden, and all data must be read and written as doubles. All stores would thus be reads as well as writes.

Due to time constraints, we were unable to implement memory latency in this way. Instead, all memory accesses (fetches, loads, and stores) were parameterized to a static latency, and cache mode was not implemented. For loads and stores, this was simple to implement as it only involved sending a single-cycle memory access, then using a counter to wait for a static number of cycles before reading the response and allowing the functional unit to complete. Fetching was more complicated; as described in more detail above, we made use of a shift register to store a history of the program counter (PC) and the corresponding valid bits. Instructions were read from memory and paired to the corresponding PC and valid bits from four cycles earlier. Only valid instructions were passed on to the instruction buffer.

Due to the pipelining nature of these changes, despite the less than ideal implementation CPI only increased marginally. Loads and stores do not lock out other memory accesses (although

the primitive replacement of an LSQ prohibits the further improvement of CPI by increasing the number of functional units).

6.2 Icache

The `icache` module is an instruction cache designed to efficiently provide data to the processor's fetch stage. It uses the `icache_data` array, which contains `ICACHE_ENTRY` structures to store instruction blocks, tags, and validity bits. The address requested by the processor is split into tag and index values that map to specific cache lines. If the tags match and the data is valid, it's output through `Icache_data_out` and validated with `Icache_valid_out`. In the case of a cache miss, the cache interacts with memory via `proc2Imem_command` while monitoring responses using `mem2icache_response` and `mem2icache_tag`. Once data is received and the awaited tag matches, the relevant cache line is updated and marked as valid. Unfortunately, due to time constraints we will not be submitting a version of our design that includes an icache.

6.3 GUI debugger

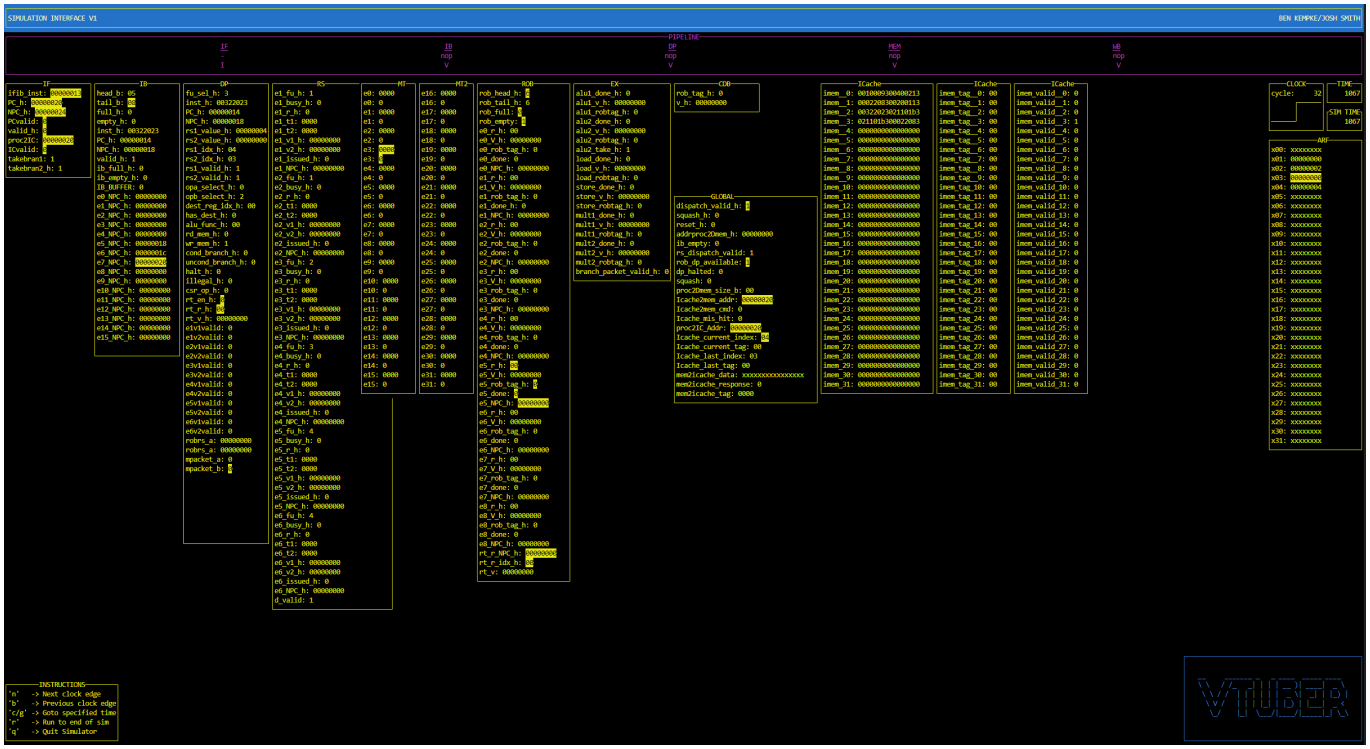

Figure 7: Our GUI Debugger

Our team has developed an advanced visual debugger for our custom processor architecture, designed to enhance the debugging process by providing a comprehensive, real-time visualization of all critical signals and states within the processor. This tool is crucial for efficient debugging, allowing engineers to observe interactions and dependencies between various components of the processor dynamically.

The visual debugger is structured to display detailed views of different processor blocks, including the execution unit, register file, and pipeline stages, among others. Each block within the debugger graphically represents the state and values of necessary signals—such as clock cycles, instruction decoding stages, register states, and data paths. This setup not only facilitates a deeper understanding of the processor's operational behavior but also significantly accelerates the identification and resolution of design issues.

By integrating this tool into our development environment, we aim to ensure robustness in our processor design and streamline the testing and optimization phases, ultimately leading to a more reliable and efficient hardware product.

7 Discussion

7.1 What Worked

- Packetization: We decided on datapaths early and put individual signals within packets, so that we did not have to change interfaces of modules over and over again while we put more signals into each individual packet.
- When in doubt, go back to the design: we spent non-trivial amounts of time dreaming up our own solutions to edge cases (for example, two ALUs both resolving mispredicted branches on the same clock cycle). We course corrected in time to realize that all strange edge cases should already have been captured by the P6 design and it is our job to understand the design thoroughly and implement it rather than coming up with non-standard designs.

7.2 What did not work

Starting the implementation process with scoreboarding algorithm is not a great idea. There is not that much overlap between scoreboarding and Tomasulo's algorithms. Future teams should consider going straight to the implementation of Tomasulo's algorithm, or risk being set back 3-4 days trying to modify interfaces to accommodate the intricacies of Tomasulo's algorithm on top of scoreboarding.