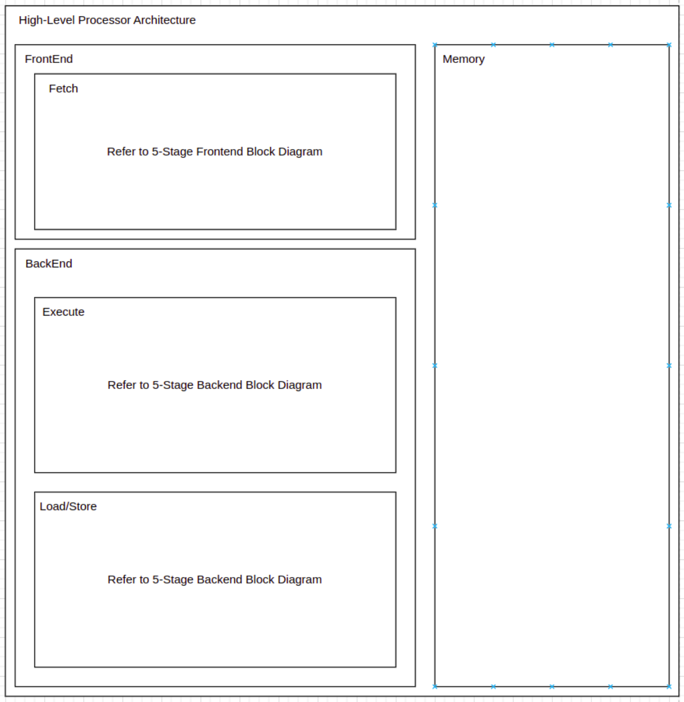
KuMarry Me Report

Max Ma | Joshua Yun | Larry Du

Introduction

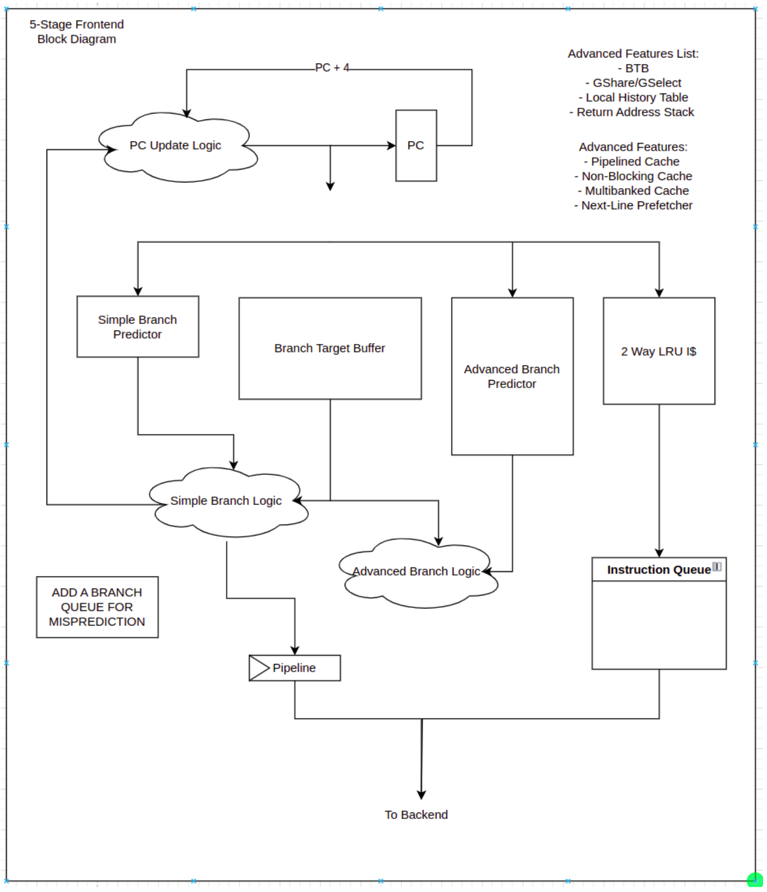
Computer architecture involves exploring an enormous design space filled with tradeoffs and bottlenecks at every turn. The goal of this project was to gain experience traversing this design space by targeting a large set of advanced features. Throughout the beginning weeks of the design, we focused on correctness in order to create a baseline out-of-order processor. After establishing this foundation, we continued to remove bottlenecks by completing various advanced features that are detailed later in the report. By the end of the project, we placed 3rd in the design competition, learning a lot about modern processor bottlenecks in area, power, and delay.

Project Overview


High-Level Processor Architecture
FrontEnd
Fetch
Refer to 5-Stage Frontend Block Diagram
BackEnd
Execute
Refer to 5-Stage Backend Block Diagram
Load/Store
Refer to 5-Stage Backend Block Diagram
Memory

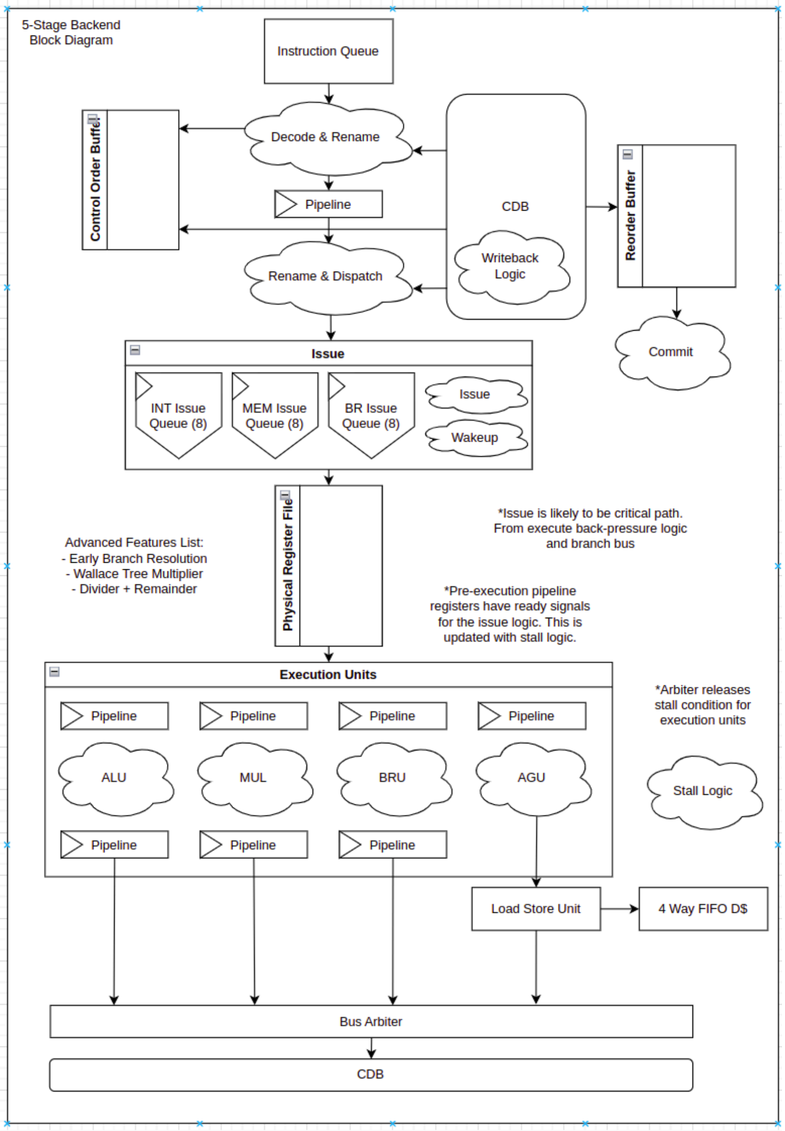
The project aims to construct an out-of-order processor integrated with a separate instruction and data cache hierarchy. We believed that instruction fetching would become a bottleneck very quickly. Thus, the frontend design includes a complex dynamic branch predictor with a pipelined instruction cache for quicker instruction fetch capabilities. The frontend interfaces with the backend via an instruction queue. The backend architecture will facilitate explicit register renaming alongside in-order commits. Because of the slow main memory interface, load and store dependencies will be managed through dedicated load and store queues. The load and store unit supports store to load forwarding in order to alleviate memory bottlenecks. To deal with the flush penalty, speculative branch execution will be addressed by tagging inflight instructions and flushing older instructions when necessary. With many features to implement, we divided the development of this project into three separate stages. The frontend, backend, and memory system can be divided into three distinct components with each group member handling a specific section, as shown above. This made the project manageable since each group member could handle tasks independently. However, this approach made final project integration difficult since each member made assumptions about how other components worked. In order to fix this, the team should have implemented the base processor together and split up for advanced features.

Milestones
The development of this project was split into three separate checkpoints with each checkpoint having specific development tasks. The tasks split, verification techniques, and specific issues are discussed for each checkpoint.



For checkpoint 1, the overall project planning was discussed. Max was tasked with implementing the out-of-order backend, which will be modeled by the explicit register renaming datapath. Max will also implement advanced features such as a pipelined dadda tree multiplier and early branch resolution. Larry was tasked with implementing the in-order frontend, which includes the fetch stage and instruction queue. Larry will also implement advanced features such as a complex branch predictor. Josh was tasked with implementing the memory hierarchy, which includes both instruction and data caches. Josh will also implement advanced features such as a pipelined cache and a complex store load unit. The main testing strategy involved using constrained random stimulus along with functional coverage. This is mainly achieved through UVM testbenches found in *hvl/modules*, organized into a *test_suite* in the provided Makefile. Directed tests and RVFI will also be utilized to verify individual modules and the entire processor design, respectively. For checkpoint 1, some specific issues that we wanted to address early were ensuring instructions of older/highest priority were issued first. Establishing a priority queue during the issue stage in the backend is necessary. Also, developing an effective branch predictor is essential to avoid the significant penalty of branch misprediction. This will likely involve a combination of BTB and various history/prediction structures. Finally, addressing memory bottlenecks is imperative. Implementing a pipelined read-only instruction cache will assist the fetch stage. Coupling this with prefetching will help reduce the number of stalls in the frontend.

For checkpoint 2, we completed a lot of the backend functionality, which includes all the backend data structures needed for explicit register renaming. This includes the RAT, Free List, RRF, Physical Register File, Reservation Station, and ROB. Proper rename, dispatch, and issue logic was developed and carried over to future development. The execution stage was designed to support pipelined execution units for future advanced multipliers or floating point units. A single CDB was implemented and proper arbitration and stall logic was designed. The backend was verified by using both a magic queue and magic frontend. The queue was used to manually check CDB commit signals and any miscellaneous logic. The frontend was hooked up to *cpu.sv* allowing the group to test with RVFI and run_top_tb. All the given test programs were able to run to completion passing both RVFI and Spike.



For checkpoint 3, we changed the memory model from a 1 cycle magic memory to a more realistic burstable banked memory model. The added functionality for this checkpoint includes running *coremark.elf* on bmem/competition memory. We also started the advanced features which we have listed below:

- Early Branch Resolution
- Branch Target Buffer (BTB)
- Return Address Stack (RAS)
- Two-bit Saturating Counter
- GShare
- Next-line Prefetcher
- Pipelined Cache
- Advanced Load/Store Unit

Advanced Design Features

After analysis of the baseline processor, we noticed that suboptimal execution units, control hazards, and memory bottlenecks would be the main source of slowdown. Thus, our advanced features targeted removing these bottlenecks and are discussed below in their respective sections.

Execution Units

The implementation of a dadda tree pipelined multiplier and Synopsys Divider IP addresses significant bottlenecks in arithmetic operations within our processor design. The dadda tree pipelined multiplier, although increasing hardware complexity, substantially reduces the latency per instruction compared to the provided multiplier, which took 64 cycles for a single multiply instruction. This enhancement is particularly advantageous for workloads with heavy arithmetic computations, such as physics simulations or encryption applications. Conversely, the integration of Synopsys Divider IP streamlines division operations by reducing the number of control and arithmetic instructions required, consequently enhancing the overall execution efficiency. However, this improvement introduces considerable chip area. The efficacy of these features is contingent upon workload characteristics, with arithmetic-intensive tasks benefitting significantly while workloads with infrequent arithmetic operations may see less impact.

Branch Bottlenecks

Early Branch Resolution

A large feature we implemented was early branch resolution, which aims to address performance bottlenecks associated with branch instructions. This approach involves broadcasting the branch resolution in the execute stage, allowing subsequent instructions to continue execution without waiting for the branch outcome at the head of the ROB. Branches are tagged by the frontend to preserve program and speculation order, ensuring that instructions are killed in the correct sequence. By resolving branches selectively and early in the pipeline, the processor minimizes the flush penalty associated with control hazards and improves overall instruction throughput and energy efficiency. This feature is particularly advantageous for workloads with frequent branching, such as code with conditional statements or loops. It is also more advantageous without a complex dynamic branch predictor. However, its effectiveness may be limited in scenarios where branches have unpredictable outcomes and mispredictions are high. Early branch resolution also introduces additional complexity to the pipeline. Nonetheless, early branch resolution contributes to enhancing the processor's ability to exploit instruction-level parallelism and improve performance in branch-heavy workloads.

Branch Target Buffer

Branch prediction was a large focus on the frontend in order to reduce flushes as we paid a fairly high flush penalty before early branch resolution was implemented. We first began with an always taken branch predictor, as this required a branch target buffer to work properly. Our BTB is a direct mapped SRAM cache, taking bits [9:2] as the set index and the remaining bits as the tag. If the BTB misses, we consider the branch not taken. When the backend responds with information about a branch (taken or not, and its target address if taken), we update the BTB with that target address. We attempted to make the BTB set associative, but due to limitations in SRAM ports, it would require external control logic that we felt was not the best use of our time. Always taken + BTB brought our correct prediction rate from 34% to 67% on coremark, a very significant improvement. Area and power increased slightly, but thanks to SRAM these tradeoffs were minimal. Without the BTB, predicting the target address of a branch would be slow and difficult, so the BTB is almost essential for any sort of branch prediction aside from static not taken. Any workload with branches and a branch predictor should have a BTB for best performance.

Return Address Stack

The return address stack follows RISC V recommended control: push when the destination register of a JAL or JALR is x1 or x5 (return address register and temp register used for CALL and RET), and pop when the source register 1 of a JALR is x1 or x5. We implemented a naive flush method, where if a flush occurred at any time on the frontend due to branch mispredict, the entire RAS would be reset. We discussed implementing RAS recovery on mispredict, but determined that the area and power requirements were too heavy, as that would require keeping a snapshot of the RAS for every branch predicted. The RAS proved relatively inconsequential, improving our hit rate by about 0.5% on coremark with two bit saturating counters. It initially reduced our hit rate slightly when running with always taken due to the naive flushing. Area and power for the RAS are minimal, we settled on a depth of 8, meaning 8*32=256 bits of registers. The RAS would definitely show more significant improvement for function heavy code, as it can save and restore the return address regardless of where the function was called from.

Gshare

Before Gshare, we implemented two bit saturating counters, a necessary prerequisite. The counters were initially a direct mapped flip flop array cache with PC[9:2] as set index. They were set to weakly taken. Two bit saturating counters improved our hit rate from 67% to 76%, again very significant. Next, we implemented the global history register. On every branch we speculate, we shift our speculated prediction for that branch into the GHR, meaning the GHR holds 32 bits of the last recently speculated branch predictions. Each speculated branch stores a snapshot of the GHR at that time. When the backend responds, if we predict correctly, we do nothing to the GHR. But if we predict incorrectly, we restore the snapshot of the GHR taken by that mispredicted branch. In this way, branch history is updated speculatively, allowing Gshare to use the most current history (waiting for the backend to respond before updating the GHR would allow many branches to speculate on stale history). Finally, to implement Gshare, we XORed the PC with the GHR, and used bits [9:2] to index into the counter arrays. Ghare initially brought our hit rate down from 76% to 74% due to only using 8 bits of history. As we increased the set indexing bits ([9:2] up to [11:2] and even [15:2]), Gshare boasted significant improvement in hit rate, up to 82% on coremark, compared to two bit saturating which only reached 77%. However, such large set indexing exponentially increased our counter array size and power (imagine 2^16 * 2 bits), to the point where more than half of our area and an extra 10mW of power was being used for the counter arrays. To mitigate this, we turned our counter arrays from ff arrays to SRAM, which helped the problem but did not eliminate it. For competition, we opted to turn off Gshare and reduce our counter array sizing, as plain two bit saturating counters had already improved our hit rate to good levels and they even performed better than Gshare at smaller size. Gshare would perform best with a workload with significant history correlation (imagine repeated code where individual branches commonly switch between taken and not taken).

Memory Bottlenecks
We integrated a next-line prefetcher for instruction memory to address bottlenecks within our processor design. By fetching data from the next sequential memory addresses ahead of time, the prefetcher aims to reduce the latency associated with memory accesses. This prefetching strategy anticipates the processor's next instruction cache line and brings the required data into the cache before it is actually needed. Thus, this mitigates the impact of memory access latencies on overall performance. The implementation of a next-line prefetcher offers notable benefits for workloads characterized by sequential program execution with minimal branches. However, the effectiveness of the prefetcher may be limited in workloads with irregular instruction memory access patterns.

Memory Hierarchy and Interactions
A pipelined parameterizable cache was used in the development of the instruction cache of the processor. Since all instructions needed to be fetched and thus required a memory operation, special attention was made to its latency when compared to that of the data cache. Furthermore, datacache's impact could be mitigated by a load store buffer, whilst without instructions to execute, the entire pipeline could not continue accomplishing work. To that end, a pipelined instruction cache was designed, such that with consistent cache hits, it could achieve a throughput of 1 instruction per cycle. Due to many benchmarks being very compact, it had a sizable impact on overall performance of about 0.1 to 0.15 IPC uplift on most work loads.

The load store unit was made to address the large memory bottlenecks that we foresaw in many of the load store heavy benchmarks. An out of order load and in order store architecture was used. Stores and loads were allocated during the dispatch step of the processor. During this step, the instructions were also assigned a ROB ID tag, which was used later during the address and data resolution step. When the AGU finished resolving its address, and results, it sent its findings to the LSU unit. If the instruction was a store, the AGU result was sent to the CDB to be retired from the ROB. During retirement from the ROB, the store with the ID in question was then marked as committed, and awaited the data cache before being removed from the queue. As soon as the store was retired from the queue, the ages of all the entries were decreased. When the load ages hit zero, loads were permitted to fire given that all the data was not forwarded. Forwarding was done using an age order lookup, with younger stores overwriting older stores. Doing this yielded an overall uplift of anywhere from 0.15 to 0.25 IPC in various workloads, due in large part to the previous system causing unnecessary stalls in the in order portion of the system.

Additional Observations
We observed that the benchmarks lacked significant instruction-level parallelism. As a result, the top-performing processors prioritized low power consumption over raw performance, resulting in designs with minimal advanced features. Consequently, the need for large reservation stations was diminished, given the limited instruction level parallelism. This made designs resemble

more closely to that of a pipelined processor. Moreover, the branch mispredict penalty for most designs was notably low, reducing the effectiveness of control hazard mitigation features and efficient branch prediction mechanisms. This observation calls into question the necessity of investing in such advanced features, considering their associated costs in chip area and power consumption.

Conclusion

Our project set out to navigate the complex design space of computer architecture, targeting a plethora of advanced features to address performance bottlenecks in execution units, control hazards, and memory access. Through a structured development process divided into three checkpoints, we successfully implemented features such as early branch resolution, branch prediction, and an advanced load/store unit, among others. A notable observation from our work was the limited instruction-level parallelism in benchmarks, prompting a focus on low-power designs with minimal advanced features. Overall, this finding underscores the importance of aligning processor design with workload characteristics.