John Pohovey
Arjun Ray
Gally Huang

ECE 411 Final Paper

Intro:

Computer Engineering describes the study of the intersection between Electrical Engineering and Computer Science. To truly understand and appreciate the craft of a computer engineer, it is important to comprehend how computers function and how they are built. In ECE411, students are provided an opportunity to do so by working their way up from a variant of a simple computer that has been introduced to them in prior classes, to understanding how memory hierarchies work through the implementation of a cache, and finally, a fully-fledged five-stage pipelined processor.

In today's increasingly digital world, processors play a critical role in powering a wide range of devices, from smartphones and laptops to servers and supercomputers. As such, it is important to have a deep understanding of how processors work and how to design them to be efficient and performant for the task(s) at hand. This project provides an opportunity to learn about the design and optimization of a (somewhat basic) modern processor at the architectural level and to see the impact of various optimization techniques on its performance, area, and power.

Put simply, processors can be thought of as cities. The control unit can be thought of as the city's government, responsible for coordinating and directing the actions of the other components. The memory unit can be thought of as the city's storage facility, where data and instructions are kept for later use. The arithmetic logic unit (ALU) can be thought of as the city's factories, where data is processed, and calculations are performed. The input/output (I/O) devices can be thought of as the city's transportation systems, allowing data to be sent to and received from external devices. Just as a city relies on the coordination and cooperation of its various components to function effectively, a CPU relies on the interaction and integration of its various components to perform tasks and execute instructions.

The project aims to design and implement a 5-stage in-order pipeline RISC-V processor that includes various performance optimization techniques such as branch prediction, 8-way parameterized set-associative L2 cache, hardware prefetching, clock gating, and multiplication. This project is motivated by the importance of understanding computer architecture and the role that such optimization techniques play in improving the performance of a processor. By completing this design and implementation, we gained a deeper understanding of computer architecture.

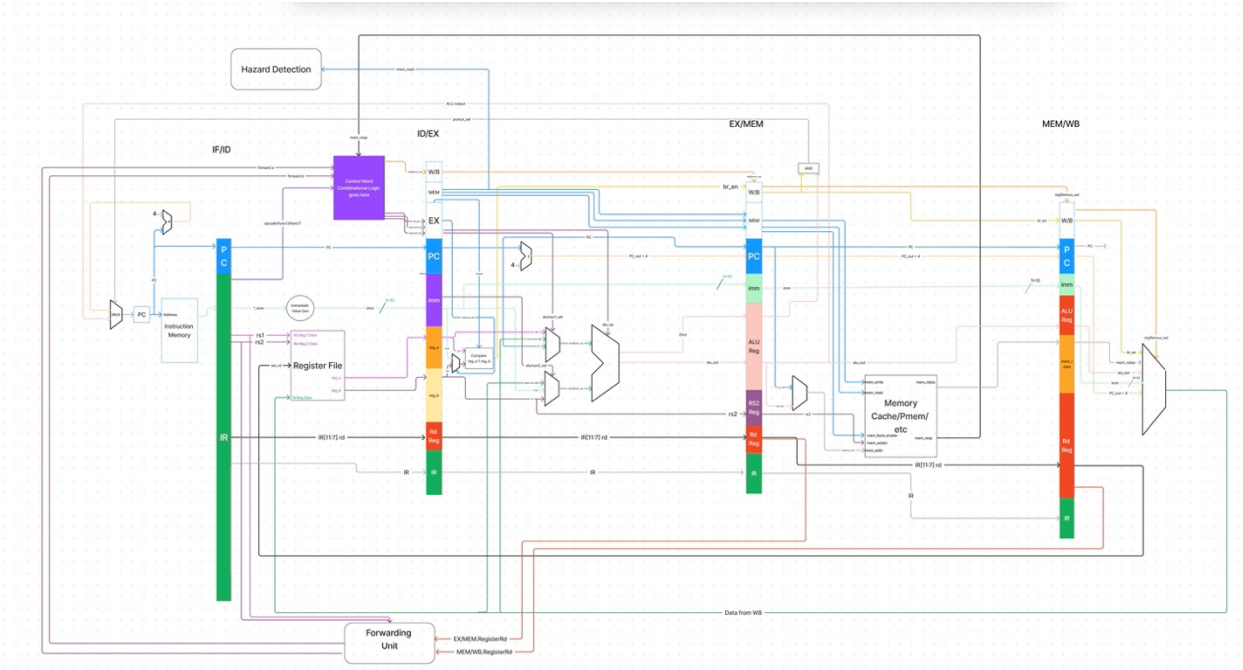
Overview:

Design Description:

Before starting checkpoint 1, we created a data path diagram that would serve as the starting point for our design. A well-thought-out design would prove to save many headaches further down the line when trying to implement the basic pipeline. After getting positive feedback from our TA Yian, we began the actual implementation of our basic pipelined design.

Checkpoint 1

For checkpoint 1, we were required to have a functional design that did not have to deal with any control hazards or data hazards. We built and implemented a 5-stage pipeline, complete with instruction fetch (IF), instruction decode (ID), execute (EX), memory access (MEM), and writeback (WB) stages, alongside the 4 pipeline registers that separated each two of the stages. Furthermore, the control word ROM was implemented in the ID stage of the pipeline and was responsible for generating the various control signals needed for that instruction as it flowed down the different stages of the pipeline. The control word itself was a struct consisting of each of the necessary signals. The diagram below shows our design for the 5-stage in-order pipeline, including what was at the time a preliminary forwarding and hazard detection units design.



When it came to testing this checkpoint, we first had to verify that our processor could correctly handle a single instruction. To do this, we mainly used the cp1 code as well as our test code with the appropriate nops to verify functionality. We also used waveforms to observe how instructions

got propagated one at a time through the various stages, so we knew the expected behavior when multiple instructions were in the pipeline at a time.

By the end of this checkpoint, we also did some preparation for the next checkpoint by submitting our detailed designs for data forwarding and hazard detection, as well as the design for our arbiter which would interface our instruction and data cache with the main memory.

Checkpoint 2

It was during this checkpoint that we added support for data forwarding and hazard detection which allowed us to appropriately handle multiple instructions in the datapath.

Since we no longer made use of nops in our code, we had to make sure that the instructions in our pipeline were executed correctly concerning the data dependencies they may have had with instructions further down the pipeline. More specifically, in our in-order execution, we had to handle RAW errors where incorrect register values are read before a previous instruction was able to write to the register.

The logic behind data forwarding was simple and ultimately came down to using combinational logic to check if either of the operands of the instruction in the ID stage was the destination register of the instruction currently in the EX stage or the instruction currently in the MEM stage. Using MUXes, the correct operands were forwarded to the input of the ALU, with preference given to any dependencies with the instruction that was in the EX stage as it would be the more recent instruction. Additionally, we had to modify the reg file to write in the first half of the clock cycle and read in the second half so that instructions were able to write back to register before an instruction in ID read that register.

Another major component of this checkpoint was switching from using magic memory to adding separate instruction and data caches that interface with physical memory via the arbiter. The arbiter served as the middleman between the caches and physical memory. Since main memory only has a single port, the arbiter determines the priority on which cache request will be served first in the case when both caches miss and need to access memory on the same cycle. Because of the multiple cycle cache misses, we also had to introduce stalling logic into our pipeline. We decided that for our design, we wanted to only stall the necessary stages as opposed to stalling the whole pipeline. This meant that instructions in front of where a stall was occurring were allowed to commit and nops were inserted in the pipeline until instructions were done stalling.

Finally, flushing and branch prediction was added to this checkpoint. We would flush the pipeline (namely, the IF/ID pipeline register) if the branch predictor gave a taken prediction, but the actual behavior was not taken, and vice versa. To flush, we zeroed out the fields that the IF/ID pipeline register stored for the control word, ensuring that neither registers nor memory would be used and modified. An interesting optimization we made here was that we moved our

branch resolution from the MEM stage into the ID stage. This had two effects. One, when we need to flush the pipeline, we find out one cycle after the instruction was fetched in IF. If then we find that the branch prediction was wrong, we only lost one clock cycle from speculative execution. Leaving the branch resolution in the MEM stage would have meant that for every mispredicted branch instruction, we would have wasted work on three incorrect instructions and needed to flush the vast majority of the pipeline, instead of just the very front, wasting multiple clock cycles. However, as a caveat of moving forward the branch resolution into decode, without storing branch targets in something like a branch target buffer (BTB), it was not possible to predict that a branch was taken and compute the target all in one cycle (since this would have implied fetching and decode and computing the target all in one cycle, and also would have made the ID stage redundant and obsolete). The ultimate effect this had is that regardless of whether the branch predictor predicts that the branch instruction is taken or not taken, PC is always loaded with PC+4, because of the above-mentioned reason. Despite this, because we do resolve branches in ID, the pipeline still is much higher performant than the version where branches are resolved in MEM since the cost of flushes in the pipeline is 3x lower. To accommodate this move of branch resolution to ID, forwarding and hazard detection had to be vastly expanded (to account for new e.g., load-use hazards and dependencies). Branch predictors are a key asset in codes that include a decent amount of control flow. The backward-forwards predictor should perform quite well in codes that include many loops. The BHT should perform well in codes where there are many possible control flows and many of the same instructions are evaluated with the same result multiple times.

To test this checkpoint, we first incrementally tested our forwarding logic by creating our own test code that purposefully introduced data hazards to verify that the expected values were being forwarded properly. Only after this core logic did we introduce physical memory. In testing our physical memory, we painstakingly scrolled through waveforms checking for incorrect memory reads/writes. In the end, after fixing quite a few timing issues and stalling edge cases, we were convinced of our physical memory implementation after our design correctly ran cp2, cp3, and all the comp codes.

With our basic pipelined processor complete, it was time to differentiate our processor with the addition of advanced features which included an L2 cache, m-extension, an eviction write buffer, and a more accurate branch predictor.

Checkpoint 3

Checkpoint 3 was our opportunity to explore some of the advanced feature options that we thought would not only be interesting to implement but also those that we thought would most improve the performance of our design on the competition code. After doing some background research and consulting Yian, we implemented the advanced features and tuned our advanced features as described in the following section.

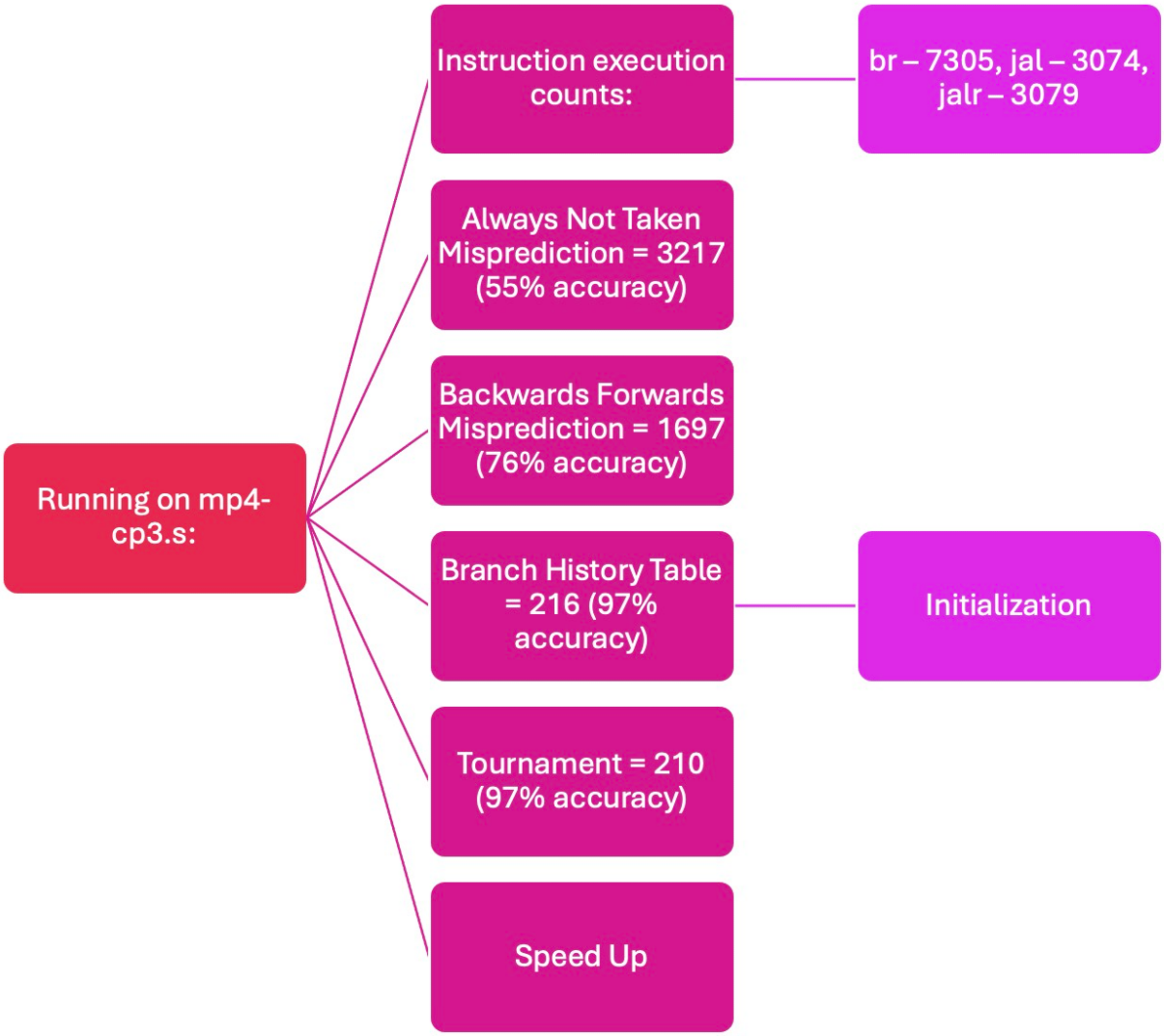
Advanced Features

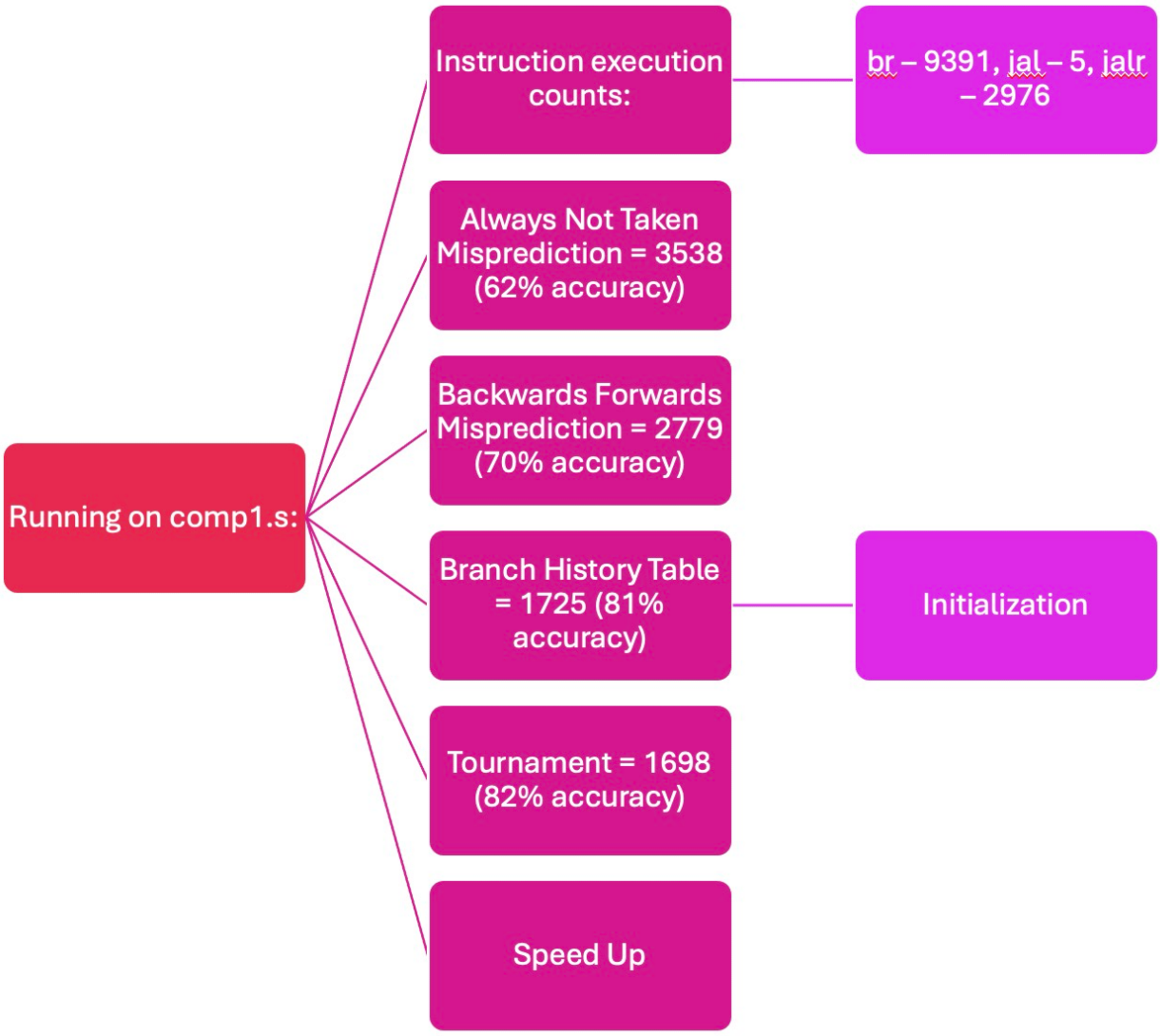
One set of advanced features we implemented for our project was branch prediction. All in all, four separate branch predictors were devised. A static always not taken predictor, a backward forwards predictor, a branch history table, and a tournament predictor. As the name implies, the always not taken predictor would predict that any branch instructions it saw flow through the ID stage of the pipeline would be not taken, and as such would ensure that the next instruction loaded would be the immediately following instruction, the program counter (PC) + 4, where the 4 is for 4 bytes to move to the next 32-bit aligned memory address, and therefore next consecutive instruction in memory. Branch instruction target addresses are calculated relative to where the branch instruction itself is located. The next branch predictor backwards-forwards relies on the observation that when looking at the target, targets that are negative, meaning branching to a previously executed piece of code, are on average taken about 90% of the time (such as what happens at the bottom of a for loop, before continuing to the next iteration). Contrastingly, branch instructions with a positive target, meaning to skip a set of instructions before resuming execution, are only taken about 50% of the time (consider if you have an if-else statement, depending on the condition, only one will be executed and the other will be skipped). A branch history table samples a subset of bits of the PC and uses these bits to index into a table. Each table entry stores a 2-bit entry, which is the output of a 2-bit saturating counter. Each value represented by the counter has a different meaning: 11 is strongly taken, 10 is weakly taken, 01 is weakly not taken, and 00 is strongly not taken. When the entry of a table that part of the PC indexes into is either 11 or 10, the branch is predicted to be taken. When the counter is 01 or 00, the branch is predicted to be not taken, and the PC is loaded with PC + 4 as normal. When the prediction of the BHT is wrong, the counter is decremented once, and when the prediction is correct, the counter is incremented. This gives the effect of "learning" which branches to take and which branches to not. By tuning the size of the BT and counter initializations, we were able to achieve very different results, as discussed further below. However, there exists a tradeoff between increased area and accuracy boost. We decided that accuracy boosting was more important than limiting the area in this regard. Finally, the tournament predictor selects between the aforementioned three predictors, and whichever predictor is performing the best at that moment, the tournament predictor will coopt that prediction, and this prediction will be the overall prediction for the processor.

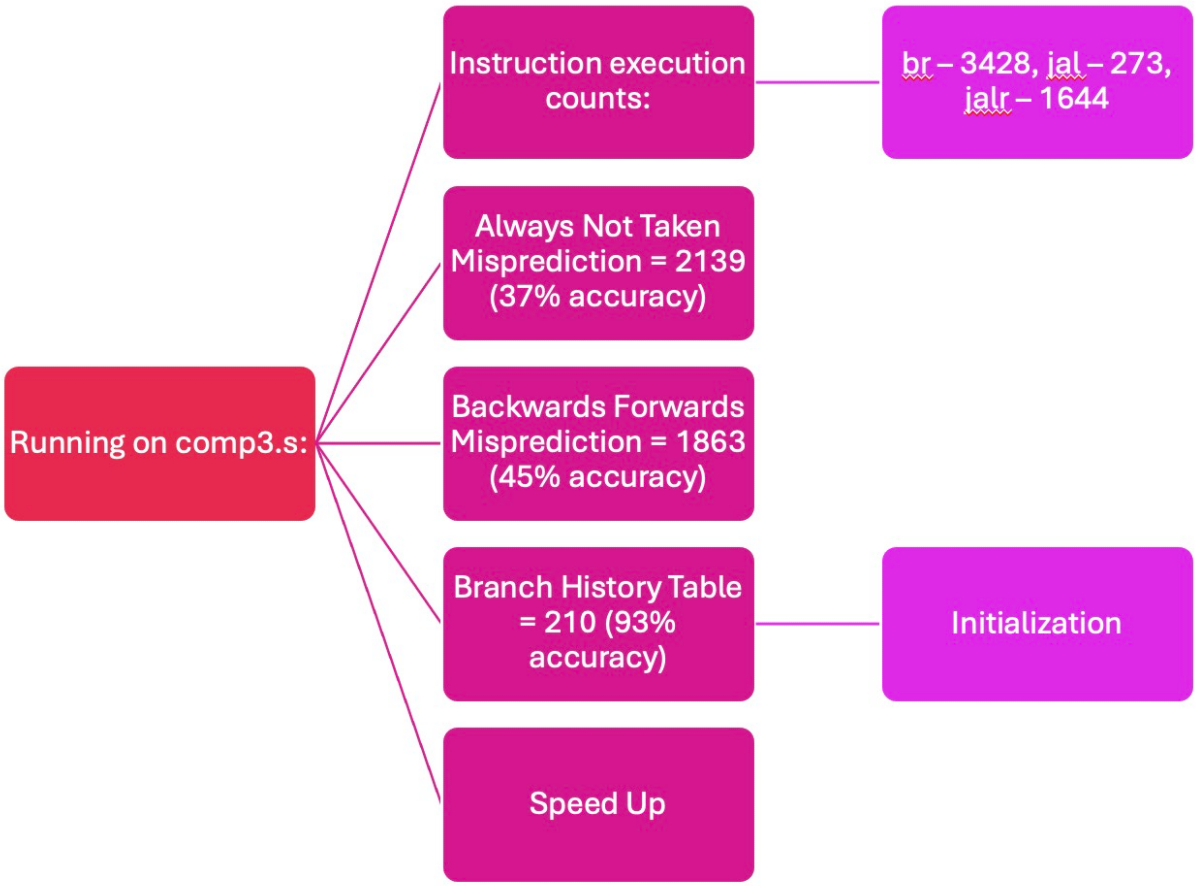
The testing of the predictors was fairly straightforward. A branch enable signal was calculated via outputs of the register file, and then the target was computed accordingly in ID, as discussed earlier. Testing was done to verify about the BHT for example when the branch enable signal was high and the BHT also predicted the branch to be taken, the 2-bit saturating counter at that index was incremented accordingly. A similar thing was done if branch enable was high but the prediction was not taken – the corresponding counter would be decremented, and was done similarly for the other two scenarios. This testing was facilitated by analyzing waveforms from different assembly codes. Testing of the backwards-forwards involved verifying the sign of the

target of the particular instruction corresponded to the correct prediction (negative for taken, positive for not taken). The always not taken predictor is a rather trivial case, as it is just assigned a constant low. The tournament predictor was tested via waveforms, looking to see that the individual predictor that had been performing the best up to that point was the prediction that the tournament predictor adopted.

Of these four branch predictors, we suspected that the always-not-taken predictor would sit very roughly around 40-60% accuracy and be the worst predictor. Next, the backwards-forwards predictor was hypothecated to be an improvement over the always-not-taken predictor. Then the BHT was predicted to be another improvement over the prior. And finally, the tournament predictor was planned to be the most performant predictor, as it simply assumes the prediction of the best predictor of the former 3. In the images below, we see some stats from mp4-cp3.s, comp1.s, and comp3.s.




Running on comp1.s:
Instruction execution counts:
br – 9391, jal – 5, jalr – 2976
Always Not Taken Misprediction = 3538 (62% accuracy)
Backwards Forwards Misprediction = 2779 (70% accuracy)
Branch History Table = 1725 (81% accuracy)
Initialization
Tournament = 1698 (82% accuracy)
Speed Up



From this, we can see that our original hypothesis was correct in all three instances. Always-not-taken is outperformed by backwards-forwards outperformed by BHT outperformed by tournament predictor. Notably, in all instances, a prediction accuracy of over 80% was achieved by at least one of the predictors. Again, as a note on speed up – due to the branch resolution existing in ID, with an absence of something like a BTB, it is not possible to load the correct target address without decoding instructions and resolving branches in the IF stage itself. As such, all these predictors have no true speed up. The branch resolution within ID itself, however, has a large speed-up.

An advanced feature that we expected would have a major performance impact was adding an L2 cache. We observed that a lot of the cycles in the baseline implementation were spent stalling waiting on instructions/data from physical memory. Because we wanted the L2 cache to allow spillover from both the I-cache and D-cache, we had originally planned for a unified L2 cache but settled on split L2 caches due to design time restrictions. It was suggested to us that we use a parameterized cache which allowed us to experiment with the size of the cache and strike a balance between performance, power, and area. We settled on an 8-way set associative cache, which seemed to be the point at which a larger cache didn't have a significant impact on performance. As part of this advanced feature, we also implemented a FIFO cache replacement

policy which varies slightly from an LRU replacement policy as it doesn't consider the most recently used cache line but instead replaces the first cache line that was brought in. Finally, an interesting feature that we added as part of our memory hierarchy was an eviction write buffer which was meant to hold dirty evicted blocks between cache levels and allow the subsequently missed address to be processed first, and when the next level is free, proceed to write back the evicted block.

The testing for the L2 cache was mainly done by empirical analysis, observing that cache lines were being appropriately brought into the L2 cache and properly evicted from the L2 cache, and written back to memory if necessary. We used a performance counter to show that there was significantly fewer cache misses that resulted in needing to access memory with the L2 cache hit rates in the 99%-100% range.

We suspect that the L2 cache would perform well for most workloads because the end goal for performance improvement is just to have data in the L2 cache that can serve the data instead of memory. One disadvantage of the split L2 cache is that an instruction-heavy workload would not be able to make use of space in the L2 D-cache and vice versa. We overcame this by sacrificing some area to make the L2 caches themselves quite large.

The performance increase for the L2 cache is shown in the following table.

	mp4-cp2.s	**mp4-cp3.s**	**mp4-comp1.s**
L1 I Cache Hit Rate	85%	96.9	94.5%
L2 I Cache Hit Rate	99.9%	99.9%	98.75%
L1 D Cache Hit Rate	43.75%	52%	47.7%
L2 D Cache Hit Rate	99.9%	98.9%	98.4%
Speedup	1.0	3.49	3.4

*Note that performance improvements are measured only on fully functioning test code w/ L2 cache.

On the more complicated test code that we were able to run, we observed a greater than 3.4x speedup in our execution time. And while we didn't have a baseline to compare its width, the FIFO replacement policy didn't seem to be doing any unnecessary evictions followed by a reading of the same line. The eviction writes buffer didn't have any notable effect on performance as the test code rarely required a dirty line to be written back to memory. In the end, we decided to stick with the 8-way set associative cache because, although it doubled our area and lead to an approximately 25% increase in power, this size cache seemed to hit the sweet spot of providing performance while not consuming substantial resources.

Another advanced feature that was introduced with our processor was the addition of the M-Extension for the RV32M set. Documentation was used to understand exactly how the extension would function as well as how it would integrate with the rest of our design. Two big operations

are being done in this extension – the multiplication and division operations. These operations were added to the control word in the control word module as well as the execute stage inside of the ALU. The multiplier was designed as combinational, by connecting a lot of 1-bit adders and shifting the outputs of these adders to be added to each other. The division module was designed by utilizing a shift subtraction algorithm that is like the one given to us in ECE 385. To fulfill the timing requirements demanded by each of these modules we needed to stall the entire pipeline while they were being run. This meant that a counter was enabled every time the pipeline detected an M-extension operation being decoded. This counter was 3 clock cycles for the multiplication unit since there is not much delay from the combinational circuit, but 64 clock cycles for the subtract shifted algorithm. 64 clock cycles were chosen because the division unit could have been subtracting and shifting every single bit that was inputted into the unit which takes a total of 64 clock cycles. Performance-wise, the M-extension performed considerably slower than the original comp2_i.s code since this meant that there was excessive stalling for the extension. The original time was 5844905000 ps, but the M-extension time was slower at 13975167000 ps – a 0.42 speedup.

While designing the multiplier and division unit, considerations were considered before settling on the current design. Instead of deciding to make the multiplier completely combinational, it is possible to just use a shift-add algorithm rather than rely on logical gates. This was done because the shift add algorithm for a multiplier would've been exceedingly slow in terms of clock cycles- 3 clock cycles vs 64 (same for the divider) while having a similar area. However, the division unit decided to use a state machine rather than using combinational logic due to the sheer size of a 32-bit divider once it has been synthesized. Thus, in this case, we chose to implement the divider in a state-machine fashion, which takes up less area but suffers a performance trade-off. To summarize, the multiplier was done in combinational logic because doing so did not take up too much area, but since the area for a combinational divider was too great (over the entire size of the entire project), we opted for a sequential slower divider.

Observations:

As we were putting together our competition code submission on our v2_cache_int branch, we observed that, at the expense of area, we could increase the ways in our direct mapped split L1 cache and achieve a phenomenal speed up over the base processor. On comp3.s code, for instance, increasing the size of the L1 cache, as well as increasing our clock frequency according to our calculated F-max, we were able to obtain a speedup of about 9x over the baseline, which was very interesting. Part of this is likely due to a reduction in conflict misses, predicated on the fact that we increased the size of the cache for both instruction and data.

Conclusion:

In this project, we created an initial base 5-stage RISC-V pipeline design. Then, we added a split direct mapped L1 cache, forwarding logic, and a hazard unit. Moving branch resolution into the

ID stage dictated a revamp of the forwarding and hazard detection logic. Next, we worked on advanced features, including various branch predictors, a parameterized 8-way set-associative cache with alternative replacement theory, and hardware acceleration of multiplication and division. Additionally, we included a healthy portion of novelty. The relocation of branch resolution from the MEM stage to the ID stage and the corresponding wide expansion and introduction of new forwarding and hazard logic. We chose a dynamic functionality, wherein we would stall only the front of the pipeline or at the back of the pipeline, depending on which L1 cache was accessing memory. To accompany this, we inserted nops by hijacking the control word of existing instructions in the pipeline and zeroing out memory access and register write signals when stalling. We are hopeful that our design is a “Ying”-er in the competition ☺.