AG0/5 MP4 Final Report

Simon Ge, Ben Wu, Youyou Yu

Introduction

In the dynamic landscape of computer architecture, our collaborative project revolves around the design and implementation of a 5-stage in-order pipeline processor with hazard detection and data forwarding, enriched with advanced features, optimizing overall performance. The processor consists of a RISC-V based CPU, a unified set-associative cache and adapters for the interconnects. The pipelined CPU is enriched with a branch predictor and additional features to support RISC-V M-extension. The Cache is also parameterized to offer more flexibility for a variety of designs, and a victim cache is implemented to alleviate the cache misses. Our collaborative project in designing a multi-stage pipeline with advanced features holds profound importance in computer architecture. The specialized efforts of our team underscore the collective expertise required to navigate modern processor complexities. Integrating foundational elements and advanced functionalities such as hazard detection and cache optimization reflects our commitment to enhancing overall performance.

Project overview

The RISC-V based CPU is pipelined into five stages: instruction-fetch, instruction-decode, execution, memory, and write-back. This structure ensures a good instruction number per cycle (IPC). To improve on this architecture, our group set out to boost its performance and power efficiency in several aspects. Our ultimate goal was to increase the CPU's IPC and clock speed while reducing the overall power consumption. Obstacles on the road will be outlined in more detail below and our solutions will also be addressed. In order to keep track of the progress and achieve high efficiency, we broke tasks down and allocated them to different team members. By using git as the version control tool and sticking to the plan strictly, we were able to successfully meet every checkpoint on time. In the end, we were able to increase the IPC by implementing a branch predictor unit, victim cache, and the RISC-V M extension which supports additional multiplication, division, and remainder operations on hardware level. To lower the power consumption and total area of the CPU, a parametrized capacity cache was made to optimize the tradeoff between power and performance.

Design Description

Overview

The milestones of this project are broken down into individual checkpoints which are shown below. By the end of the first two checkpoints, a baseline pipelined CPU with a fixed sized cache and hazard detection unit was completed. After adding the aforementioned advanced features in checkpoint 3, performance was profiled against the baseline CPU.

Milestones

Checkpoint 1

In checkpoint 1, a basic pipeline processor that can handle all the RV32I instructions is designed and implemented. The pipelined CPU consists of instruction-fetch, instruction-decode, execution, memory, and write-back stages. Registers are inserted in-between two stages to store all the relevant data to the following stages and control signals for pipeline stalls. In the instruction-fetch stage, the address stored in the PC register is fed into the I-cache to fetch the instruction, which is then stored into the registers between IF and ID stages, along with the PC value. The PC is updated by adding 4, which indicates the length of the instructions, and the PC register is updated by selecting the branch target address and updated PC with a MUX. The instruction-decode stage mainly consists of a registerfile and a control unit. The operands decoded from the instruction stored in the IF-ID registers are fetched from the register file and then stored in the ID-EX registers. The control unit decodes the instructions and passes the corresponding control signals for the following states into the ID-EX register. In the execution stages, the input to the ALU and the comparator are selected using three MUXes with the control signals from the ID-EX registers. The outputs of the ALU and CMP, along with the immediate values from ID-EX registers are stored into the EX-MEM registers. The memory stage is mainly used for the interaction with the memory. Either a read or write is performed in the mem stage specified by the control signals in the EX-MEM registers. The result of memory read is passed to the MEM-WB registers. In the write-back stage, the result of the corresponding instruction is selected by a MUX and the result is written to the regfile. To test the processor, a comprehensive RISC-V assembly testbench is written, which covers all the RV32I ISA. NOPs are inserted between instructions to ensure no data hazard occurs in the testbench. The result from the RVFI is monitored and a SPIKE log is compared against the golden file to ensure the correctness of the design.

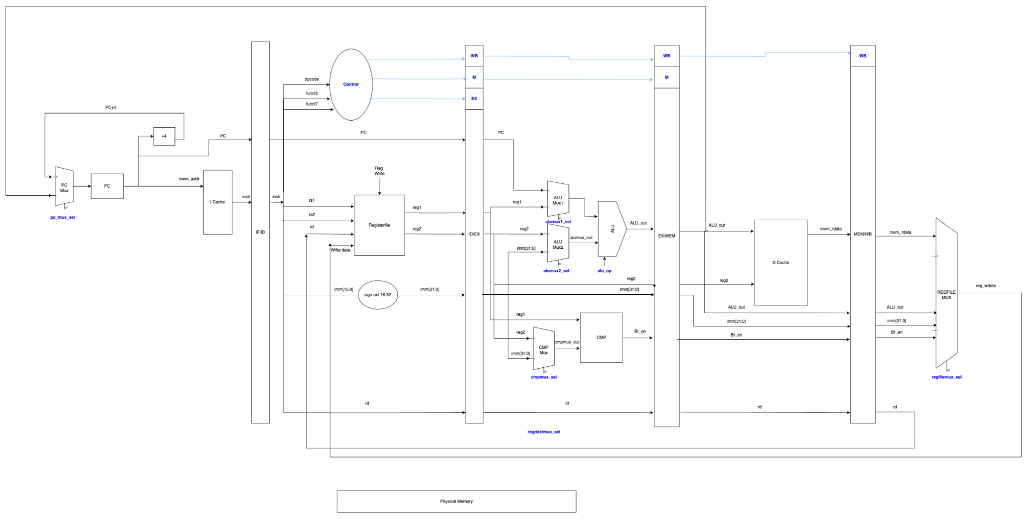


Figure 1: checkpoint1 diagram

Checkpoint 2

In checkpoint 2, several units were added to the checkpoint 1 CPU. An arbiter that communicates with a cache-line adapter and the two L1 caches were inserted into the pipeline. Its design is shown in figure 1. The two caches, instruction cache, and the data cache, replaced the magic memory used in the previous checkpoint. During the process, changes had to be made in order to address the difference between the one-cycle hit magic memory and the real memory system. To solve the cache miss latency problem, stalling the whole pipeline was necessary to ensure the correctness of computation. In addition to stalling the whole pipeline, an extra hazard detection unit was added so that necessary data can be forwarded to the place of consumption before actually writing it back to the register file which saves time by avoiding stalling. The design was first tested in a DUT manner which means that each individual component was tested alone before wiring them together in the datapath. After making sure each component works as expected, the whole CPU was tested on RISC-V assembly code that we came up with. Several aspects were focused, for example to test the stalling and forwarding functionality, we would write a load instruction and then an arithmetic operation that uses the same register for input. The RVFI monitor was useful for debugging and combined with Verdi waveform traces, we were able to solve problems related to memory access time and caches.

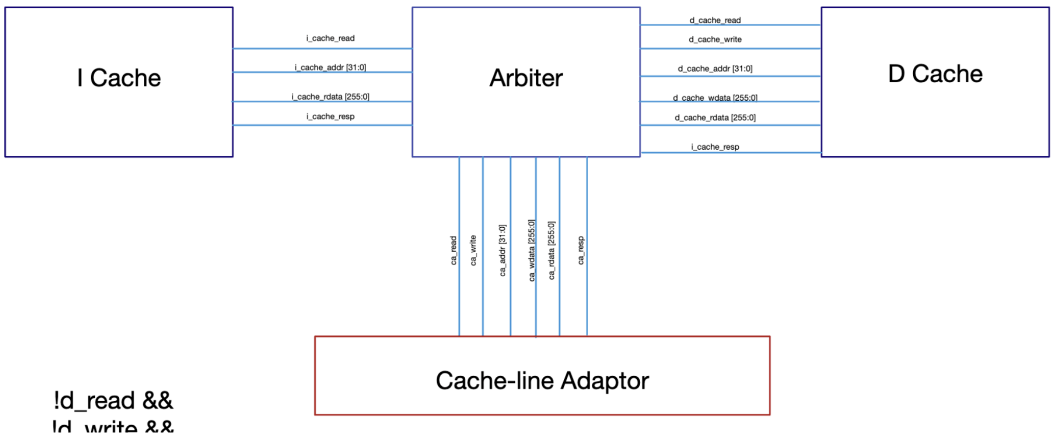


Figure 2: Arbiter design

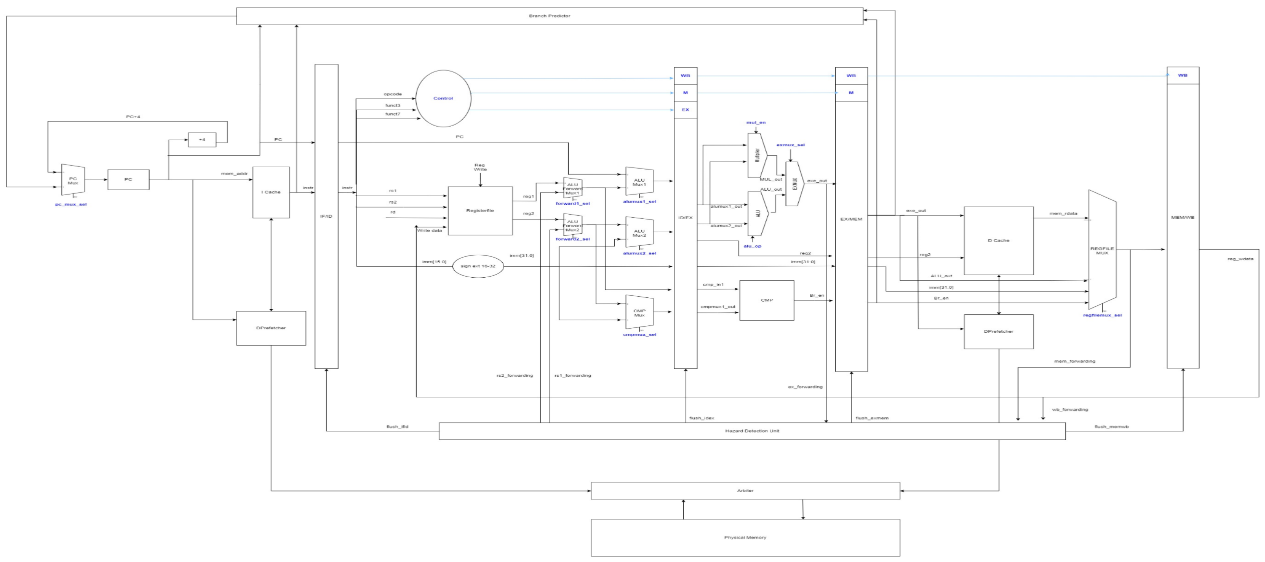


Figure 3: checkpoint2 diagram

Checkpoint 3

In checkpoint 3, we decided to split the task to “cache”, “predictor”, and “M-extension”.

Advanced design options

Option 1 - branch prediction

Design

The design of the branch predictor consists of a local branch predictor and a global branch predictor. The branch predictor includes a branch history table that stores the history of previous branches and a pattern history table that stores the state of a 2-bit counter that indicates the prediction of the corresponding branch. The difference between a local branch predictor and a global branch predictor is the history stored in the branch history table. The local branch history table stores the local history of the branches corresponding to a set of instructions specified by the PC, whereas the global branch predictor saves the global history of all branches occurring in the program. The PC value is used to determine the entree of the branch history table and the pattern history table, and a branch prediction is provided to fetch the next instruction. The entree and the prediction is passed in the pipeline, and when the actual branch is determined, the pipeline will either continue the execution or flush the pipeline based on whether the branch is mispredicted.

Testing

The design of the branch predictor is first verified by the DUT to ensure that the counter and the entree of the branch history table is updated correctly. A performance counter, including the number of correct predictions, is added to measure the performance of the branch prediction. Then a RISC-V assembly testbench is written to test all kinds of branches. Eventually, the branch predictor is integrated into the pipeline processor and the CoreMark testbench is run to ensure the correctness of the design.

Performance analysis

The performance of the branch predictor is measured on both the RV32I and the RV32IM CoreMark testbench. The branch prediction accuracy is recorded and compared against the static not-taken predictor.

	# correct prediction	Prediction Accuracy	Speed up
Static predictor	79378 / 205516	0.385751	
Local branch predictor	154078 / 205516	0.749713	1.9694x
Global branch predictor	154390 / 205516	0.751231	1.9474x

Table1: Branch Prediction Performance on RV32I CoreMark、

	# correct prediction	Prediction Accuracy	Speed up
Static predictor	27448 / 59474	0.461513	
Local branch predictor	44973 / 59474	0.756179	1.6384x
Global branch predictor	48009 / 59474	0.807227	1.7490x

Table2: Branch Prediction Performance on RV32IM CoreMark

Option 2 - M extension

Design

The multiplier was designed based on the Dadda Tree structure. The whole multiplication was separated into three stages - partial product, reduction, and output. The design was chosen because of the performance and power advantage over the traditional add-shift multiplier. The whole multiplier uses 899 full adders and 31 half adders during the reduction stage. After calculating the partial product of two operands, each bit position has to be carefully wired into these adders so to help with coding, a python script was written to generate the hdl system Verilog code. It uses a deque data structure to hold the outputs from each adder from each bit position and determines the inputs for the following adders. During each of the 8 reduction stages, adders are performing the addition in parallel and outputting the sum or carried out for the next stage reduction. The signed multiplication is handled by negating the two's complement of the negative number so that a normal multiplication can be carried out using the same procedure discussed above and in the last stage, the product is properly negated based on the number of negative inputs. The division and remainder operation are handled by a basic subtraction based divider.

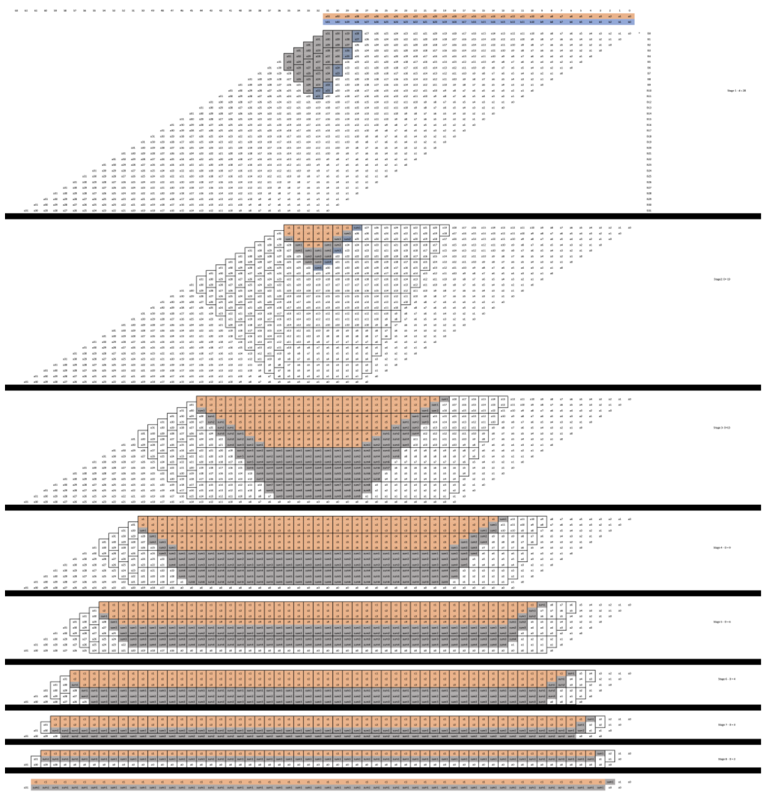


Figure 4: Dadda Tree for 32 bit multiplier

Testing

The multiplier and divider are initially tested in separate testbenches as dust. Randomization of input values is used to ensure it covers as many cases as possible. After passing all tests in dut testbenches, they are then integrated into the CPU datapath which is then tested against our customized RISC-V assembly code composed of every type of instructions in the M extension as well as other instructions like store and load. After passing all these tests, the CPU is then tested on running the coremark m version which is a very long program with the M extended opcodes.

Performance analysis

Due to the addition of hardware support for multiply and divide operations, the compiler no longer needs to separate a multiply instruction into multiple add and shift operations. The number of instructions are fewer with the M extension and thus, the same program will be able to finish in a shorter amount of time. If the same program (coremark in this case) is compiled into a M extension compatible version, the simulation run time is significantly less than the one without M extension which can be seen from table 3. The IPC is not significantly decreased either compared to the original design. Due to the addition of a multi-cycle execution unit, the IPC is expected to be lower but the tradeoff with the decreased number of instructions here results in an increase in performance overall. We are able to maintain a very similar clock speed compared to the original CPU thanks to the Dadda tree structure for the multiplier. The parallel addition in the reduction stages makes sure that the longest path in the multiplier is just 8 adders in serial.

	Simulation Start Time	Simulation Stop Time	Simulation Duration	IPC
CPU with M extension	114,485,000	2,501,255,000	2,386,770,000	0.602762
CPU without M extension	96,776,000	4,149,420,000	4,052,644,000	0.721045

Table 3: M extension coremark run

Option 3 - Cache (Parameterization and Victim Cache)

Design

Design of Victim Cache

The victim cache is a small 1-way cache aside from the main cache, which will also respond hit/miss/valid/dirty to the cache control unit. The only definition of the victim cache is by defining its behavior:

> It acts like a buffer, such that whenever there's a read/write request, the cache will examine both the victim cache and the victim cache. If the tag exists within the victim cache, then the corresponding entry will switch with the PLRU entry in the main cache, and the main cache will update the PLRU state accordingly.
>
> If during the CPU read/write, both main cache and victim cache miss the tag, then the main cache will evict the PLRU entry to victim cache, while the original victim cache entry is being written-back to memory. The "fresh" data loaded from memory will be stored in the main cache.

The design of victim cache is based on the cache structure we've used in MP3, which uses 1-cycle-hit SRAM as the holder of tags and cachelines. The state diagram stays unchanged, but additional next-state logic and operation is added.

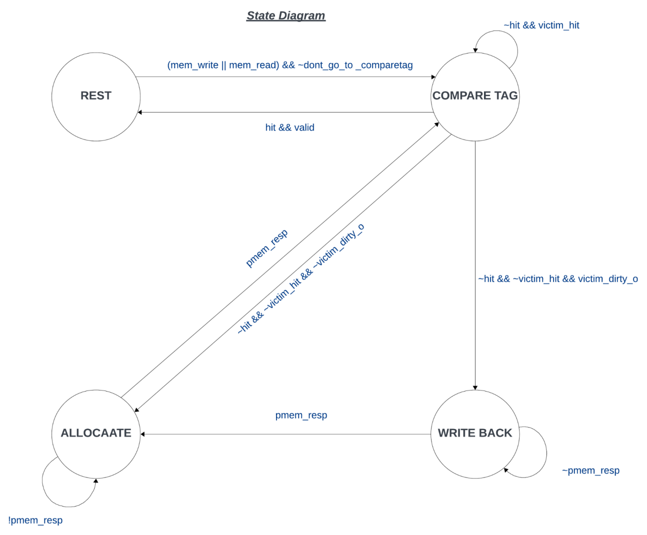

State Diagram
~hit && victim_hit
(mem_write || mem_read) && ~dont_go_to _comparetag
REST
COMPARE TAG
hit && valid
pmem_resp
~hit && ~victim_hit && ~victim_dirty_o
~hit && ~victim_hit && victim_dirty_o
pmem_resp
ALLOCAATE
WRITE BACK
~pmem_resp
!pmem_resp

Figure 5: Victim Cache State Diagram

During the state of "compare-tag", if the main cache misses but victim cache hits, there will be an additional cycle: the entry in the main cache and the victim cache will switch with each other. Then the main cache, after getting the right data and tag, will return a hit, letting the state go back to "idle".

If both caches miss, the state will let the victim cache to do the job of "write-back" and "allocate". Then, with the new data from memory, the victim cache will respond to a "victim-hit", leading the switch with the main cache. So in general, the main cache will no longer connect to the memory directly. The main cache will "pour the wastes" to the victim cache and let the victim cache takes the responsibility of all read/write with memory.

Design of Cache Parameterization

Cache parameterization contains the perspective of:

1. Number of ways (PLRU)
2. Number of sets (size of index)
3. Size of tags
4. Size of cache-lines

The most challenging one, math generalization of PLRU logic, can be demonstrated by the following chart:

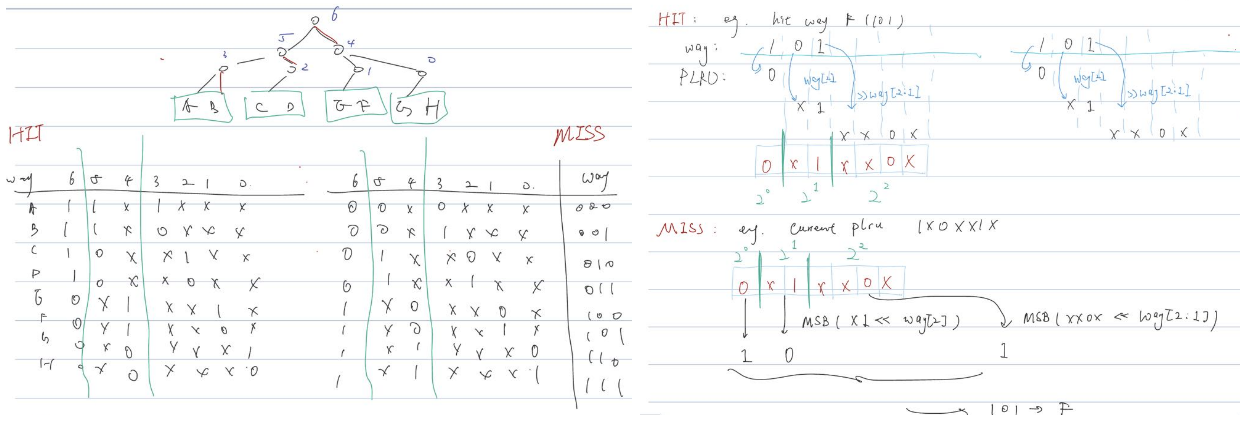


Figure 6.a, 6.b: Demonstration of Generalization of PLRU Logic

We can find a pattern between each digit of the hit/evict way index and slices of PLRU mask. We implemented this by applying for-loops in the cache control unit.

Another thing to notice is that, when we're trying to parameterize the size of cache-lines, we also need to modify the configuration of the SRAMS (to be specific, data-array). We have to re-make the SRAM module everytime we modify its property, and we also need to modify the burst-memory module, adjusting its number of bursts provided each time read/write is taking place.

For the rest of jobs, it will be purely "parameterization". We define the size of every signal by the parameters of the cache.

Testing

Since we remain a version of cache that does not have victim cache functionality, we can thus compare the performance of enabling/disabling victim cache. On the other hand, since the cache parameterization functionality mainly serves to fulfill the requirement of area, which limits us from wildly changing the parameter of cache, we didn't expect the performance improvement from this feature, and we didn't apply performance statistics to it.

We treat the base version of CoreMark as the comprehensive testing program, and we include the counter of cache hit/miss for both I-cache and D-cache; thus, we can observe the performance difference by enabling/disabling the victim cache functionality after the CoreMark finishes.

Performance analysis

	I-Cache Hit	I-Cache Miss	D-Cache Hit	D-Cache Miss	Total Run Time	IPC
Victim Cache Enabled	1196592	1165	73834	97	49678280000 ps	0.300709
Victim Cache Disabled	1196507	1250	73796	135	4968260000 ps	0.300691

Table 4: Stats of Enabling/Disabling the Victim Cache

By observing the table, we can find that we reduced 6.8% of I-cache miss and 28% of D-cache miss. The IPC improved by 0.005%. We can see that even the cache misses are noticeably reduced, the total run time and IPC didn't change too much. I suspect this is due to the reason that our cost of visiting the memory (burst-memory) is too low. If the punishment of cache miss is at a higher level, the significance of victim cache will become more obvious.

Additional observations

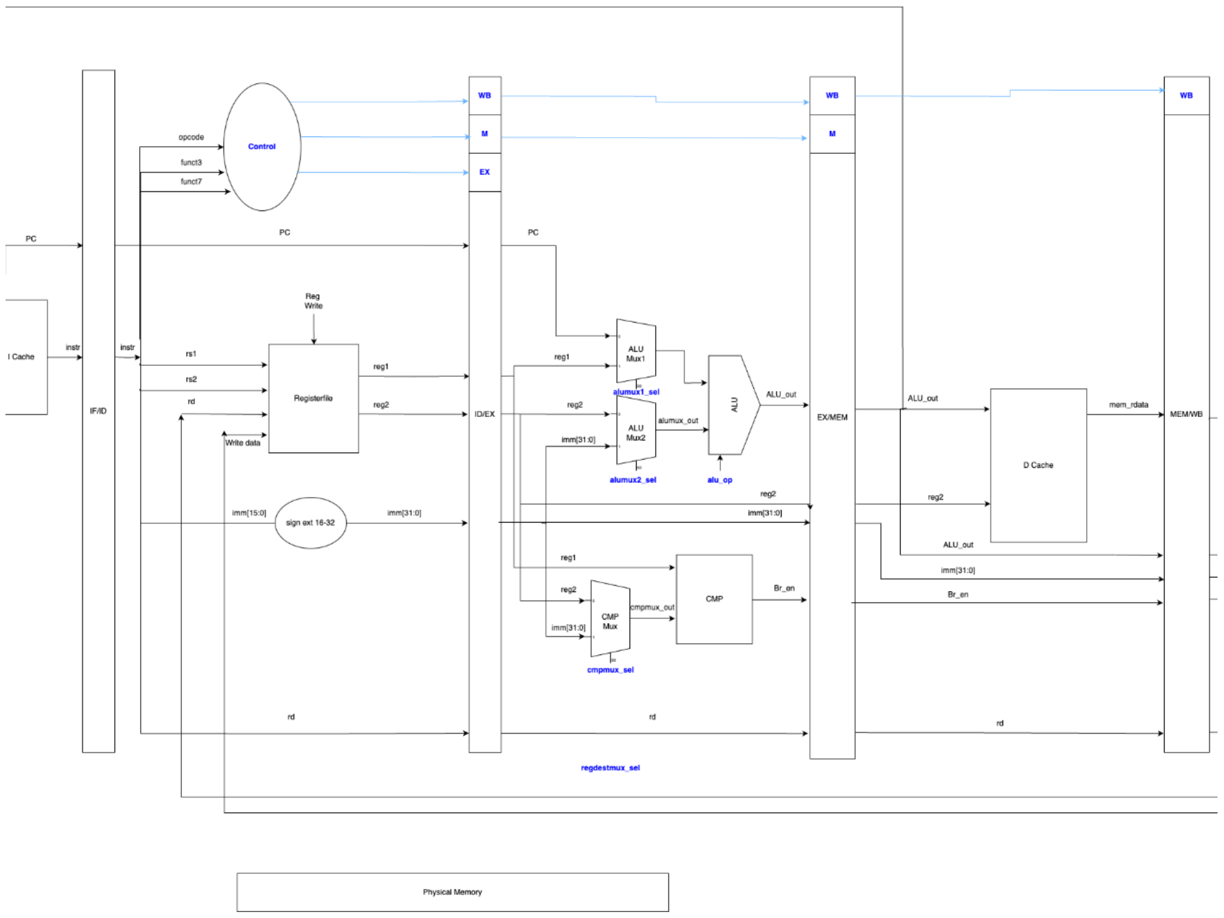
Cache Response vs Pipeline Proceeding Logic

There is an interesting bug when we integrate the 1-cycle-hit SRAM cache to our pipeline. Our pipeline will proceed only when I-cache and D-cache gives high cache response simultaneously; however, there might be a edge-case: if we start the request to them in sequence, there can be a chance that their states will loop between "idle" and "compare-tag", giving out high response value alternatively (180° phase difference). This infinite loop will cause the pipeline to stall forever.

In order to solve this problem, I introduced a "dont-go-to-compare-tag" signal in both cache's control module, such that whenever the cache detects consecutively request that is handling the same read/write task to the same address, we will disable the cache state from going to "compare-tag" state, and we will raise the response signal in "idle" state. Thus, we will prevent the state jumping back and forth between "idle" and "compare-tag". In this situation, it will stay in the "idle" and give out a high in response, waiting for another cache to give response. Once two responses are collected by the CPU control module, the pipeline will proceed.

Potential Reason of Low IPC

Even though we have a very low energy consumption, our IPC is still relatively low compared to other groups. One of the potential reasons for this is that with the implementation of M extension, the pipeline has to wait for the execution units to finish multiply or divide which stalls the commit speed.



Conclusion

In this MP, we successfully implemented a 5-stage pipelined RISC-V processor, with features of hazard detection, branch prediction, M-extension, victim cache, and cache parameterization. We thoroughly practiced the process of building and synthesizing a processor by SystemVerilog, Verdi, and OpenRAM. Special appreciation to our GOAT mentor Nick who gave us tons of help.