Authors: Isha Pendem
Shree Samavedhi
Wesley Wu

I. Introduction

In this report, we will be covering our group's design and implementation of the mp4 machine problem, in which we were tasked with producing an in-order pipelined implementation of the RV32I processor. Throughout this project, we drew schematic diagrams and wrote Verilog code in order to have a basic pipeline that handles all of the RV32I instructions, hazard detection, and forwarding. We also attempted to implement an L2 cache– to keep our Fmax high while mitigating the effects of memory stalling– as well as a Local Branch Predictor to increase our static-NT predictor's accuracy and decrease code execution time.

While not all of our ambitions came to fruition, we gained a far better understanding of the way in which an in-order pipeline processor handles instructions and the complexity involved in adding additional functionality. This project also gave us a better understanding of the importance of thorough testing, and group-distributed work.

II. Project overview

The main goal of our project was to initially meet the pre-set deadlines for each major component (checkpoints) of the RV32I processor. As for the advanced design, we decided to wait until the completion of the previous checkpoints. We then choose advanced design components that would be realistic for our timeline or help our design's performance.

For the first two checkpoints, our group decided to work together on every aspect of implementation. While this worked for checkpoint 1, this strategy became less and less applicable as checkpoint 2 continued, since the timing for all members to work together grew highly scarce. Our group distribution for checkpoint 3, therefore, changed to accommodate the scheduling conflicts, allotting each member to a specific portion of work.

We utilized various branches, in Gitlab, for each major component of design, including testing. This also helped us distribute work, and eventually merge all changes to the working branch. Collaboration was conducted using VSCode Share, and other messaging tools.

III. Design Description

A. Overview

Each aspect of our design was done in small parts, delineated by the corresponding Figures seen below for each Milestone.

B. Milestones

1. *Checkpoint 1*

The progression of our in-order processor, for handling all RV32I instructions, is detailed below from Figures A through F. This includes all five stages of the pipeline (IF, ID, EX, MEM, WB), and a Control ROM for outputting a control word.

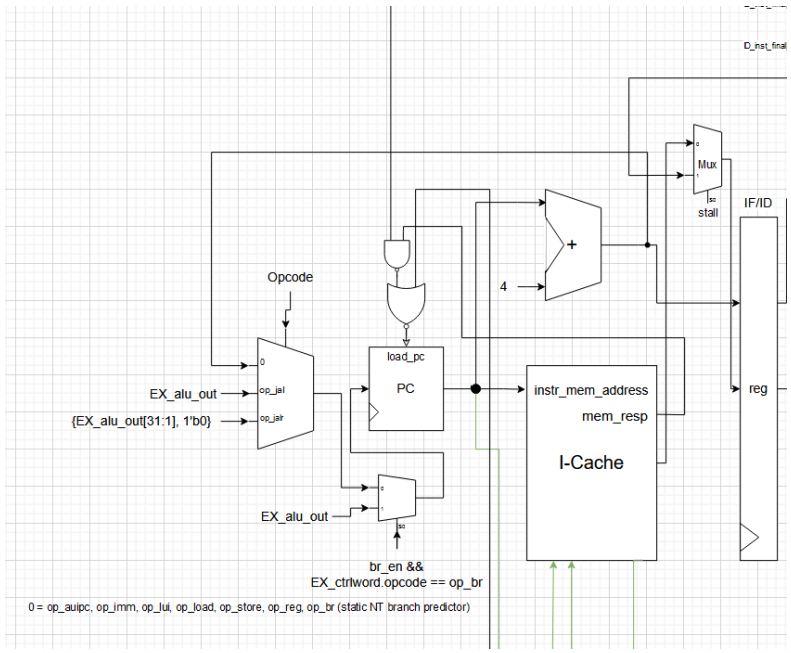

Figure A: IF Stage

The IF Stage (Figure A), is implemented using a PC Register, two Muxes, and an Adder. In checkpoint 1, our IF stage loaded in the next PC on every rising clock edge, and the PC would be decided by the opcode of the current instruction fetched so as to prevent any unnecessary stalling. There is a second mux preceding the opcode-based mux to decide whether to load the alu output for a branch if the instruction in execute stage is a branch and branch enable was set to high by the result of the comparator. Then, the PC in the PC register would get sent to the instruction magic mem, where the instruction would be

fetched in 1 instruction, and we send the instruction into the IF/ID pipeline register so it gets passed along to the ID stage for decoding.

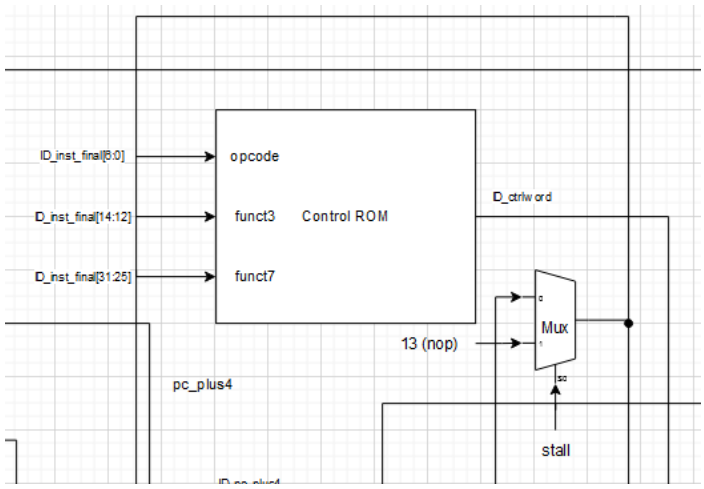

Figure B: Control ROM

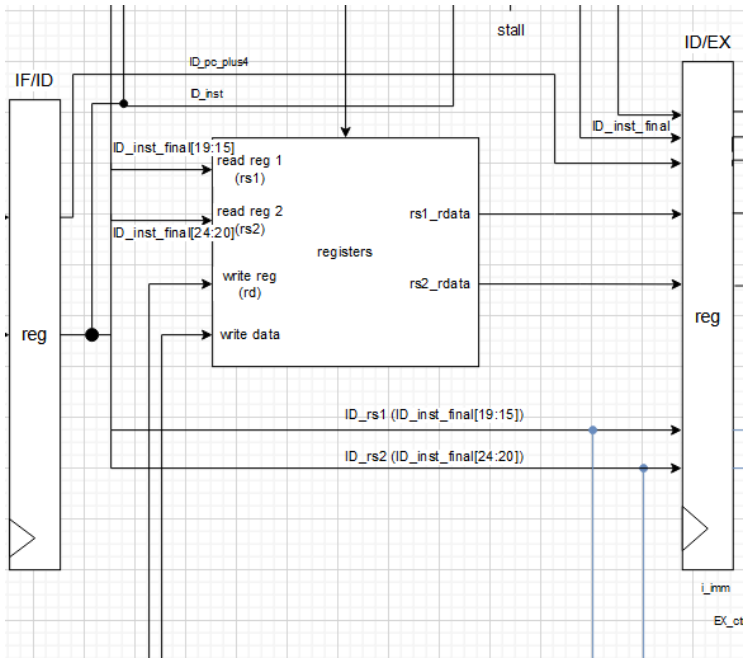

Figure C: ID Stage

The ID Stage (Figure C), is implemented using a Register File and Control ROM (Figure B). In the ID stage, the processor decodes the source registers from the instruction fetched in the IF stage and also sends out the opcode, funct3, and funct7 into the control ROM, which then generates a control word

with the necessary MUX signals for the EX and WB stage, and appropriate memory signals for the MEM stage.

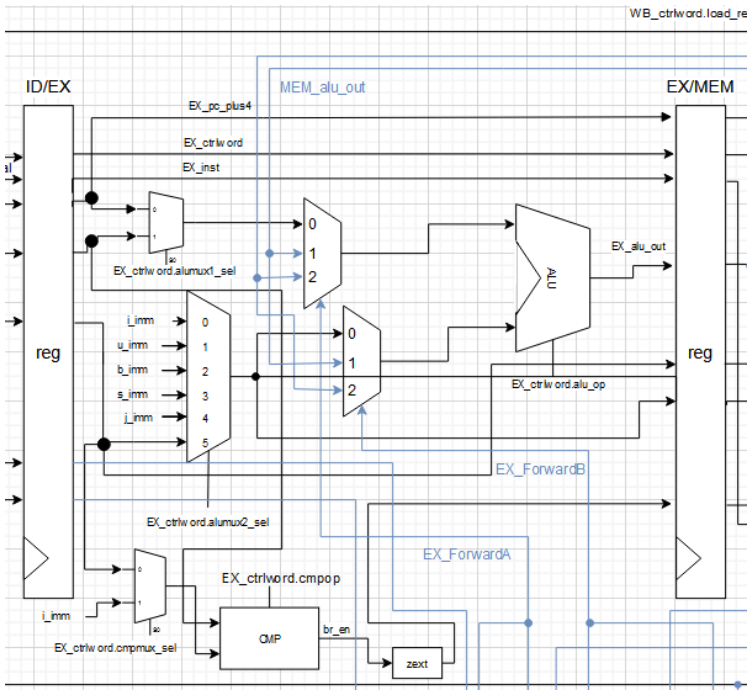

Figure D: EX Stage

The EX Stage (Figure D), is implemented using multiple Muxes, a CMP unit, and an ALU. In the EX stage, the computation for any reg-reg or reg-imm arithmetic instruction is done by the ALU, or the resolution of a branch is done by the CMP, and there are muxes to select what would've originally been inputted into the ALU/CMP unit without forwarding, and muxes to select what should be inputted into the ALU/CMP unit after forwarding.

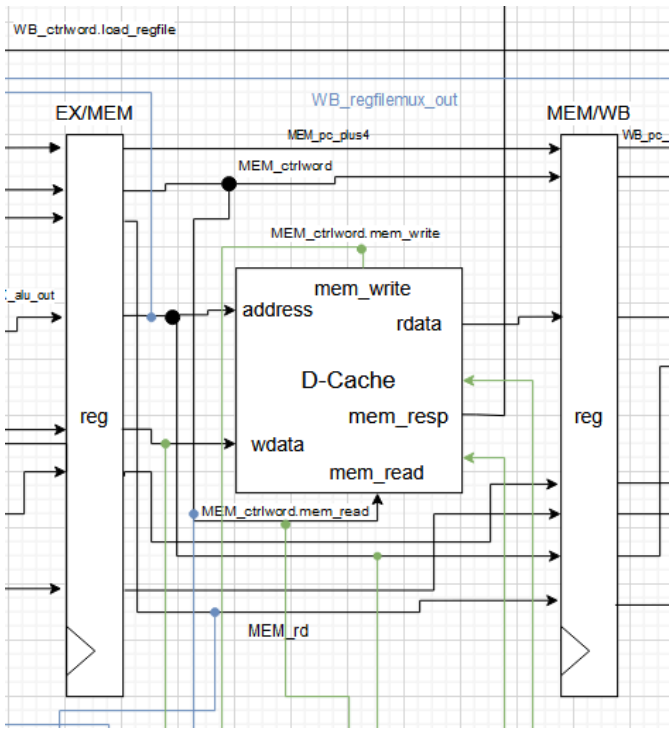

Figure E: MEM Stage

In the MEM stage, the processor communicates with the D-cache and physical memory if it sees that the control word has a mem_read or mem_write signal. If the read/write was a miss, this would be communicated to the arbiter, which would then serve the request (assuming an I-cache request wasn't previously generated) and send out the read data into the cache or write the write data to the cache and evict to main memory if necessary. In checkpoint 1, we simply had the mem_write and mem_read signals for the magic mem and the address from the EX stage ALU output. And, subsequently, this stage passes along other important information for the processor to commit the instruction in the WB stage, like the destination register and the incremented PC, etc.

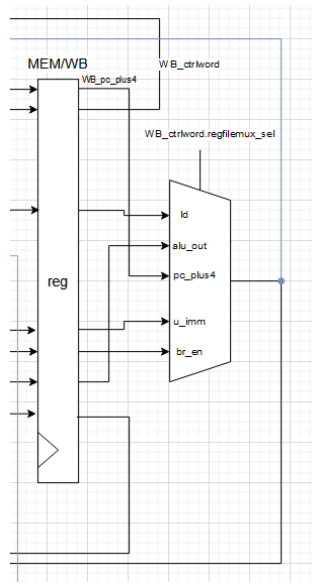

Figure F: WB Stage

The WB stage is implemented using a single mux. The mux select is passed in from the control word (which has been passed along the pipeline for every stage), and chooses what data gets loaded into the regfile (if necessary). From the control word, this would mean the load_regfile signal is 1, and this signal would get sent into the regfile in the ID stage, at which point the data gets written in the first half of the clock cycle (since we made our regfile transparent).

2. *Checkpoint 2*

The implementation of L1 caches, hazard detection, and static branch prediction is shown below in Figures G through K. These figures detail the addition and use of our arbiter and forwarding unit.

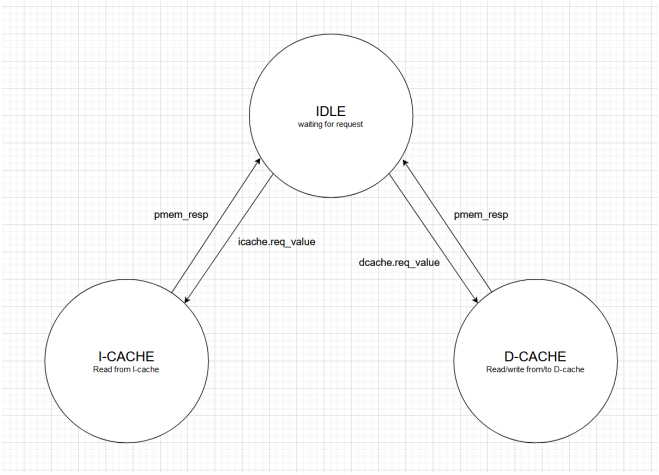

Figure G: Arbiter Schematic #1

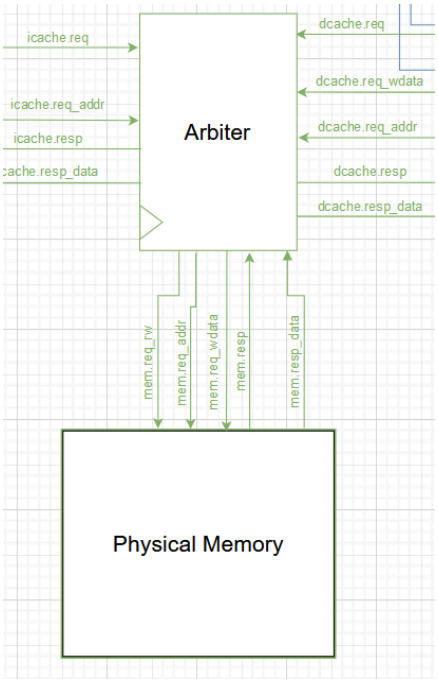

Figure H: Arbiter Schematic #2

The arbiter (Figure H) is implemented by intercepting signals from both the I-Cache and D-Cache, while also receiving data from physical memory. The state diagram for choosing which cache to serve is detailed in Figure G. The state machine has 3 states, IDLE, I-CACHE, and D-CACHE, accordingly for which request it is handling/not handling a request. In the I-CACHE state, it will intercept signals from the I-cache and send those out to physical memory,

whereby the cacheline will be read back to the arbiter, which passes the data into the I-cache and this causes a transition back to the IDLE state. In the D-cache state, there is similar logic to the I-cache state, except with support for writes since data memory can be modified by the program.
Our arbiter prioritizes I-cache requests since without the instructions, the program simply cannot run.

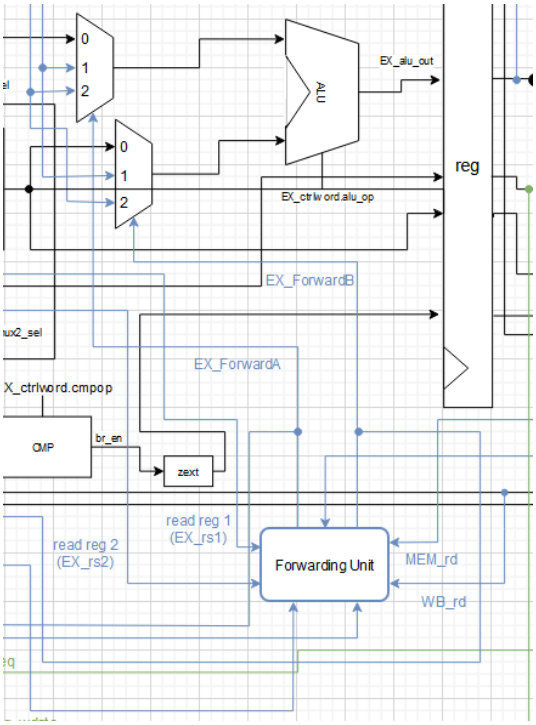

Figure I: Forwarding Schematic #1

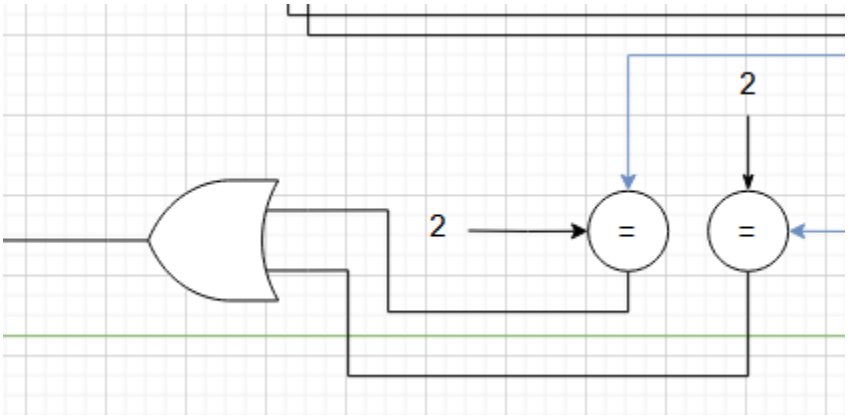

Figure J: Forwarding Schematic #2

The forwarding unit (Figure I, Figure J) lives in the EX stage, where it first does MEM->EX forwarding, checking if any of the source registers in the EX stage match the destination register in the MEM stage avoiding forwarding if the destination register is register x0 or the register index was formed due to an immediate. If it sees MEM->EX forwarding is necessary, it will send out 01 to the appropriate forwarding mux.

Then, the forwarding unit will check for WB->EX stage forwarding, with similar logic to that of the MEM->EX forwarding check, sending out a 10 to the appropriate forwarding mux if it deems WB->EX forwarding necessary, with this overriding the signal for MEM->EX forwarding.

Finally, the forwarding unit checks for a stall and sends out a signal to the IF stage if the D-cache is being read from, since we decided to forward the read data from the WB stage to prevent a long combinational path.

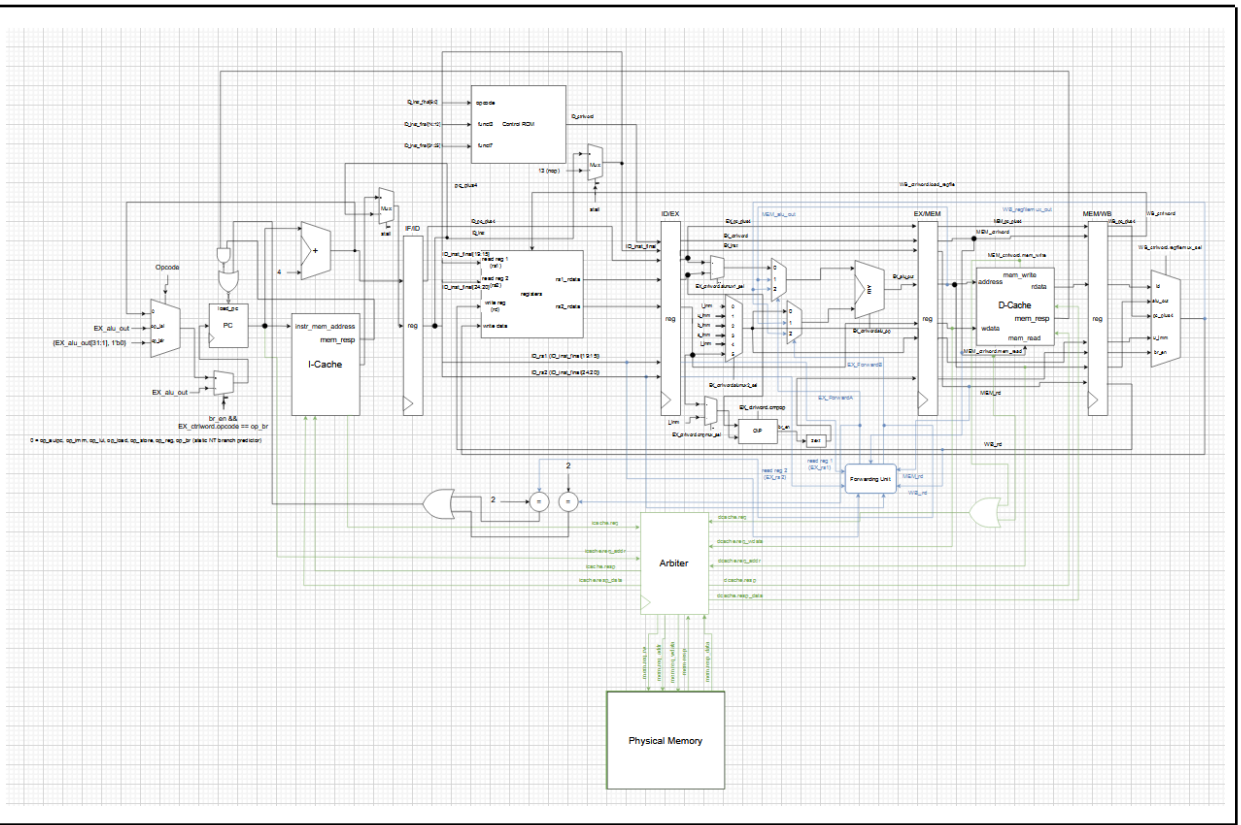

Figure K: Overall CP2 Design Schematic

For this checkpoint, we also attempted to implement RVFI integration. However, despite initially testing with an ASM file with only add instructions, we were unable to progress to testing with branches.

3. *Checkpoint 3*

For this checkpoint, we attempted to implement L2 Cache operations, as well as a Local Branch Predictor. Since we reached this checkpoint with only a week left until the project was due, we decided to implement easier components. The L2 Cache was also chosen for its performance boost– this could double as implementation for Checkpoint 4. More details for this checkpoint integration are listed below in "Advanced design options".

C. Advanced design options

1. *L2 Cache*

a) Design

The design, as shown in Figure L, made the most sense for our implementation since our arbiter already routed most of the work from the L1 caches. In order to expand this to include the L2 Cache, we switched the arbiter signals that were originally outputted to physical memory, to instead input into our L2 cache. This way, if there is a miss in both L1 caches, we route to the L2 cache. If there is a miss in the L2 cache, we continue to fetch the data from physical memory and populate the L2 cache appropriately.

We originally decided to use the given "golden cache" to implement this design for the L2 cache. However, this turned out to be a problem for us later down the road, as the golden cache includes a line adapter (that we overlooked), meant to change the size of inputs for L1 caches. Since the L2 cache is transferring data from caches previously, this line adapter was not necessary. It was unfortunately too late for us to implement this design change in our L2 cache.

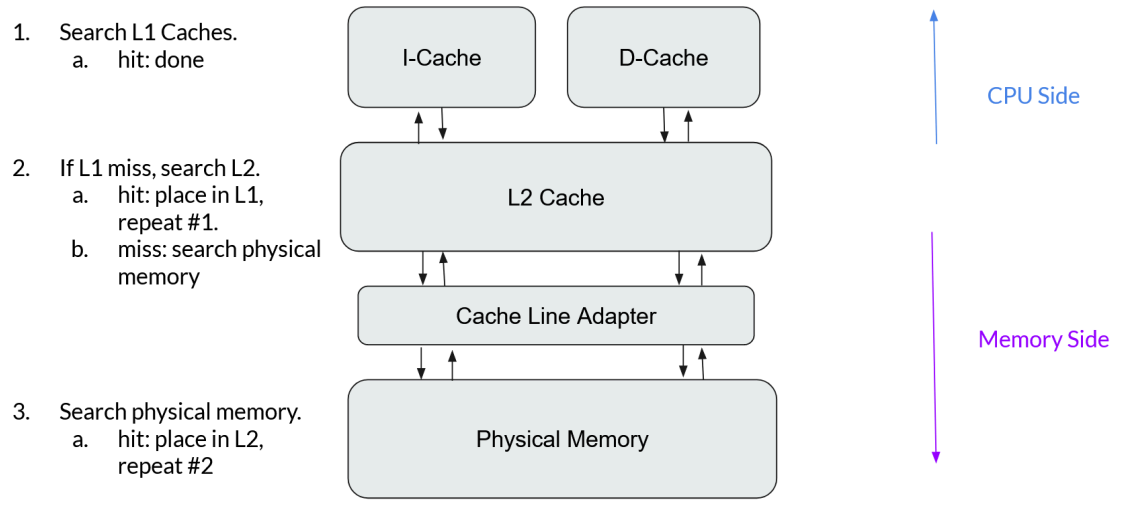

Figure L: L2 Cache Design

b) Testing

We began testing the L2 Cache by working off of previously tested/working code from just two L1 cache implementations. This included smaller test cases with only "add" instructions. The goal was to reach the same register output as the previously tested outcomes.

c) Performance analysis

Since we were not able to get any functionality out of our L2 Cache, we were not able to see the performance analysis. However, if the cache were to work, we would expect a higher Fmax, lending to higher data fetching speeds. Since there is a larger storage of caches, this performance boost would be natural as a result of the feature itself, rather than specific design decisions.

2. *Local Branch Predictor*

a) Design

As a baseline in CP2, our branch predictor was a static-NT branch predictor, and we chose to implement a local branch predictor to see if we could improve the branch prediction accuracy and speed up our execution time.

The design, as shown below in figure M, is fairly simple: the N least significant bits get passed in as index bits into the branch history table, which is essentially a table of N-bit history registers with the most recent branch resolution getting left shifted into the LSB of the entry indexed into. Then, this branch history register value is used as the index into the 2-bit saturating counter table. From the counter we indexed into, if it is either ST or WT (11 or 10), it is predicted taken.

Simultaneously, while the BHT/counter determine the prediction direction, the same N least significant bits of the PC get sent to the BTB, where a hit is checked: if the entry PC matches the current PC and there is valid data (i.e. not just null/zero data), then it is considered a hit.

If the predictor sees a BTB hit and a predict taken, it will send out this new PC into the PC register of the IF stage.

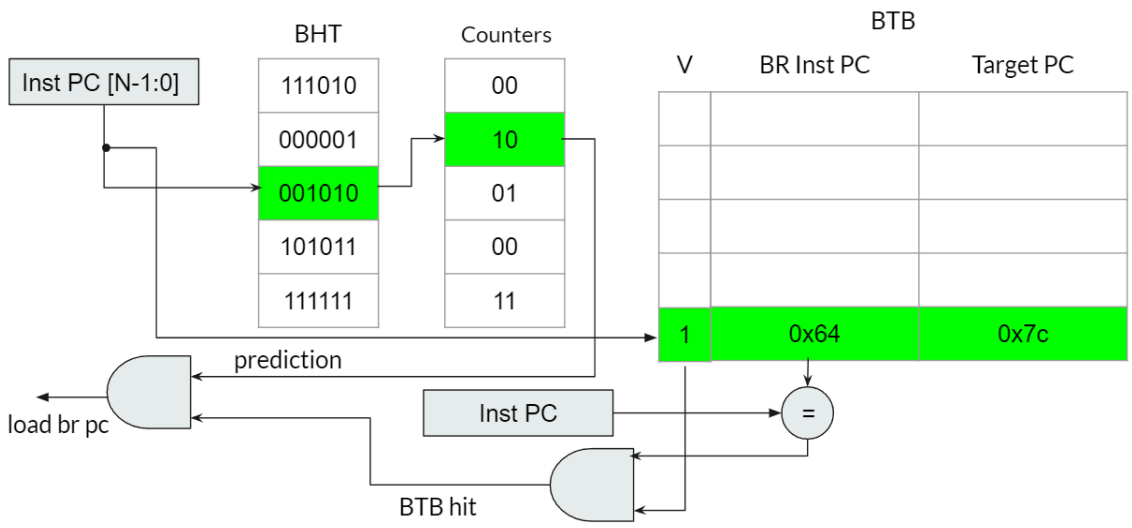

Figure M: Local Branch Predictor Design

b) Testing

We began testing on the mp4 checkpoint 2 code to make sure it produced the same output (i.e. same regfile output) as in checkpoint 2 as a baseline, but kept finding the simulation hanging or producing strange results. But, since we were low on time, we could not figure out the source of the bugs. And, on some runs, the sim would run to completion with correct regfile ouptut, but there would be negligible

difference in the execution time, and since we weren't able to setup RVFI monitor for the competition codes, we couldn't tell if the branch predictor actually helped our execution times.

A flaw of the design that we did not catch, as pointed out by the TAs during our presentation, is that using the last N bits would result in effectively losing a quarter of the indexing bits because the last 2 bits in a RV32I instruction are always multiples of 4 due to the instruction length of 4 bytes and possibly a reason for the predictor not working as expected.

c) Performance analysis

Since we weren't able to finish the local branch predictor, we weren't able to get prediction accuracy statistics from the predictor. However, if it were successful with a good prediction accuracy, we would expect to see a large decrease in execution time of the competition codes.

IV. Conclusion

In checkpoint 1, our goal was to implement a basic pipelined RV32I processor without hazard detection/forwarding or cache integration. After finalizing our paper design, we were able to complete this within one school week and demo the overall functionality over the weekend.

In checkpoint 2, our objectives became much more complicated, as we were required to implement cache integration and an arbiter, a static-NT branch predictor, as well as a forwarding unit with hazard detection.We were able to quickly finish the forwarding unit and static-NT branch predictor, finally finishing our caches + arbiter a while after.

In checkpoint 3, we attempted to implement a local branch predictor with a BTB as well as an L2 cache, but unfortunately since our checkpoint 2 was a major roadblock for us in terms of time, we were ultimately unable to implement these features. And, as a result, we weren't able to participate in the design competition for checkpoint 4 either.

In all, if there was one thing we would've done differently if we had to start over, we would certainly make sure to plan out times to work on the MP accordingly, and we would have split up the work starting from the very beginning rather than midway through checkpoint 2. Ultimately, if we had been able to juggle all our responsibilities the way our arbiter successfully handled requests between the two caches, we probably would have been able to meet the requirements set out for us in checkpoints 3 and 4.