MP4 Final Report

Group Name: Team ATM
Timothy Chan (tjchan2)
Matthew Nolan (mrnolan3)
Andrew Gacek (andrewg3)

Introduction

Throughout the course of these last few months, Team ATM has developed and implemented a RISC-V pipelined microprocessor. Our processor is designed to execute most of the RV32I instruction set, as well as handle data hazards as a result of instructions run according to the instruction set. In this report, we will outline the basic required features of the pipelined microprocessor, along with more in-depth descriptions of advanced features that were completed along the way. Over the time period, we have learned much more about the design flow and the process of creating a pipelined microprocessor, as well as optimizing the microprocessor to improve the overall performance of the design.

Project overview

The RISC-V pipelined microprocessor uses multiple stages in order to execute sections of instructions in parallel, which in turn speeds up the execution time of the instruction.

The processor was implemented in four different checkpoints, with the end goal of being able to handle the majority of the instructions within the RV32I instruction set. Our team first started by designing paper designs of the pipelined microprocessor, where we were able to trace the execution of the required RV32I instructions, then constructed the pipeline without the handling of control or data hazards by using NOPs. We then moved on to the next checkpoint, where we successfully designed and implemented data forwarding and hazard detection, as well as an arbiter to interface the L1 instruction and data caches to the cache line adaptor, which eventually connected to the main memory. We started on optimizations to the pipeline processor in checkpoint 3, where we were able to implement three advanced features, described later in the document.

Design description

Overview

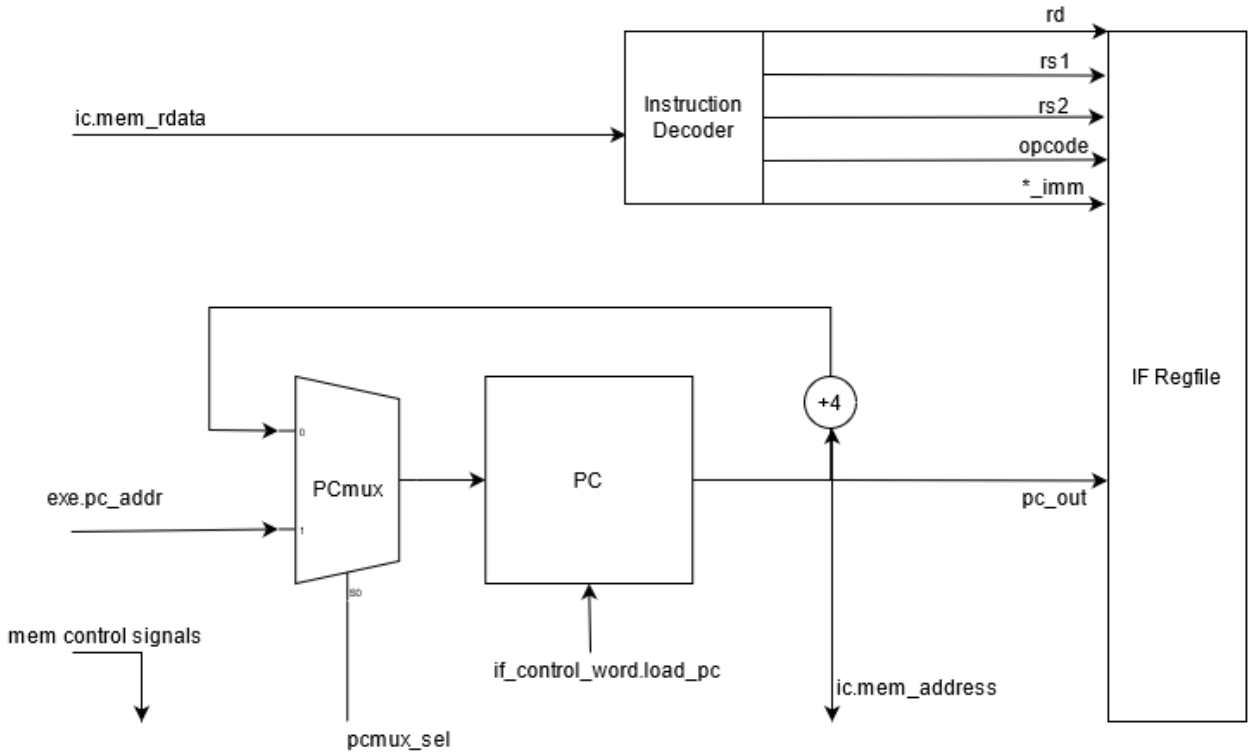
As described previously, the project included the basic necessities of a pipeline processor with the ability to handle data forwarding and hazard detection, as well as a dual L1 cache. After the foundation of the pipeline processor was laid out, the team implemented a parameterized L2 cache, a tournament branch predictor, and the RISC-V M Extension implemented using a Wallace Tree as our advanced features to improve the run time and overall performance of the processor. Our team was able to successfully implement each one of the three features, but fell short in the end in fully implementing these advanced features with the pipeline processor.

Milestones

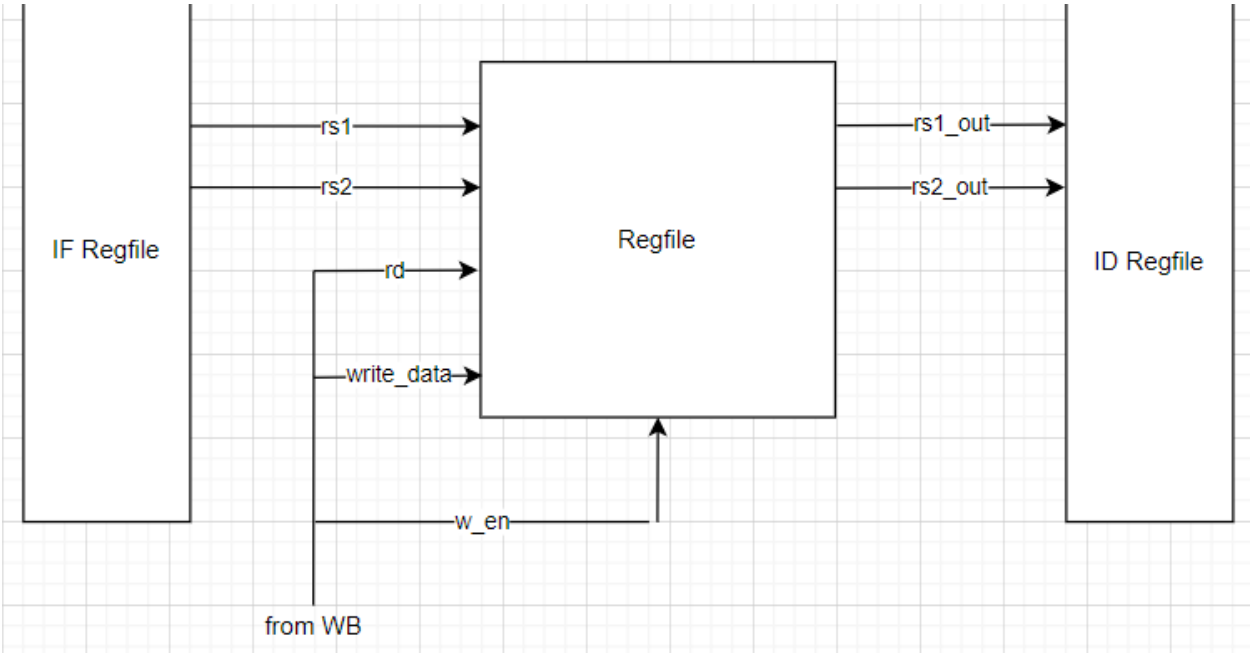
1. Checkpoint 1

For checkpoint 1, we designed and implemented a basic 5 stage pipeline that handled most of the RV32I instructions. We used knowledge from lectures and implemented the pipeline by using the 5 stages needed for a RISC machine, which are:

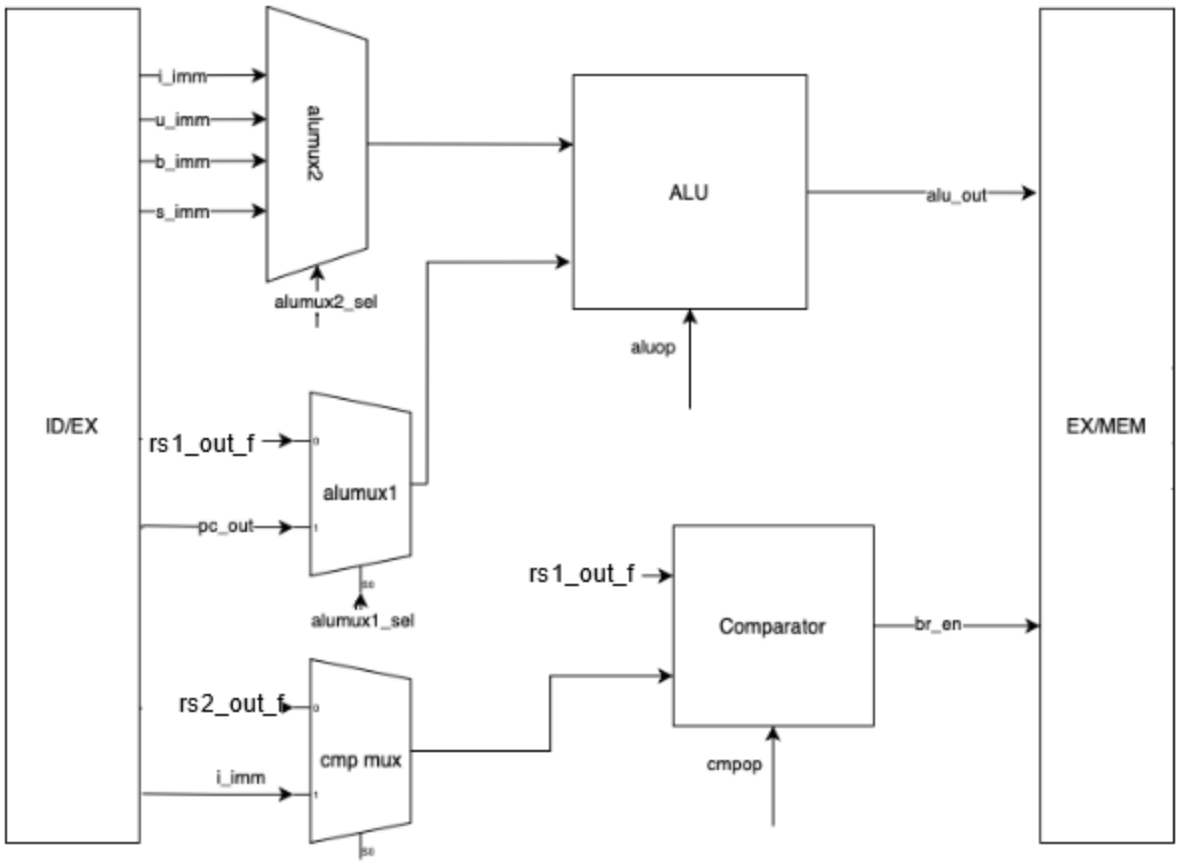
- IF - Instruction Fetch, which receives the instruction and updates the PC in one cycle



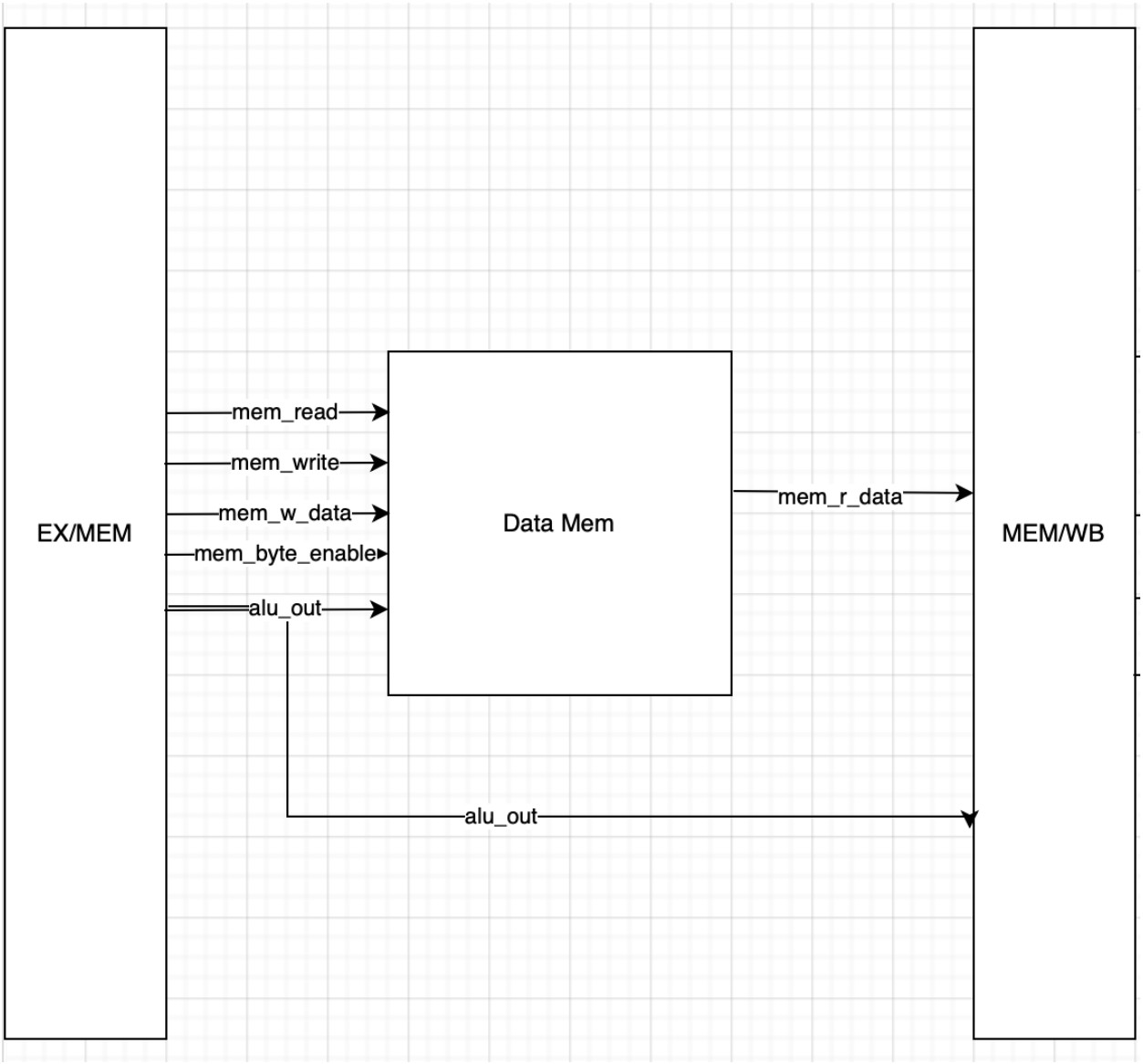
- ID - Instruction Decode, which deciphers thee instruction from the previous stage and reads from register.



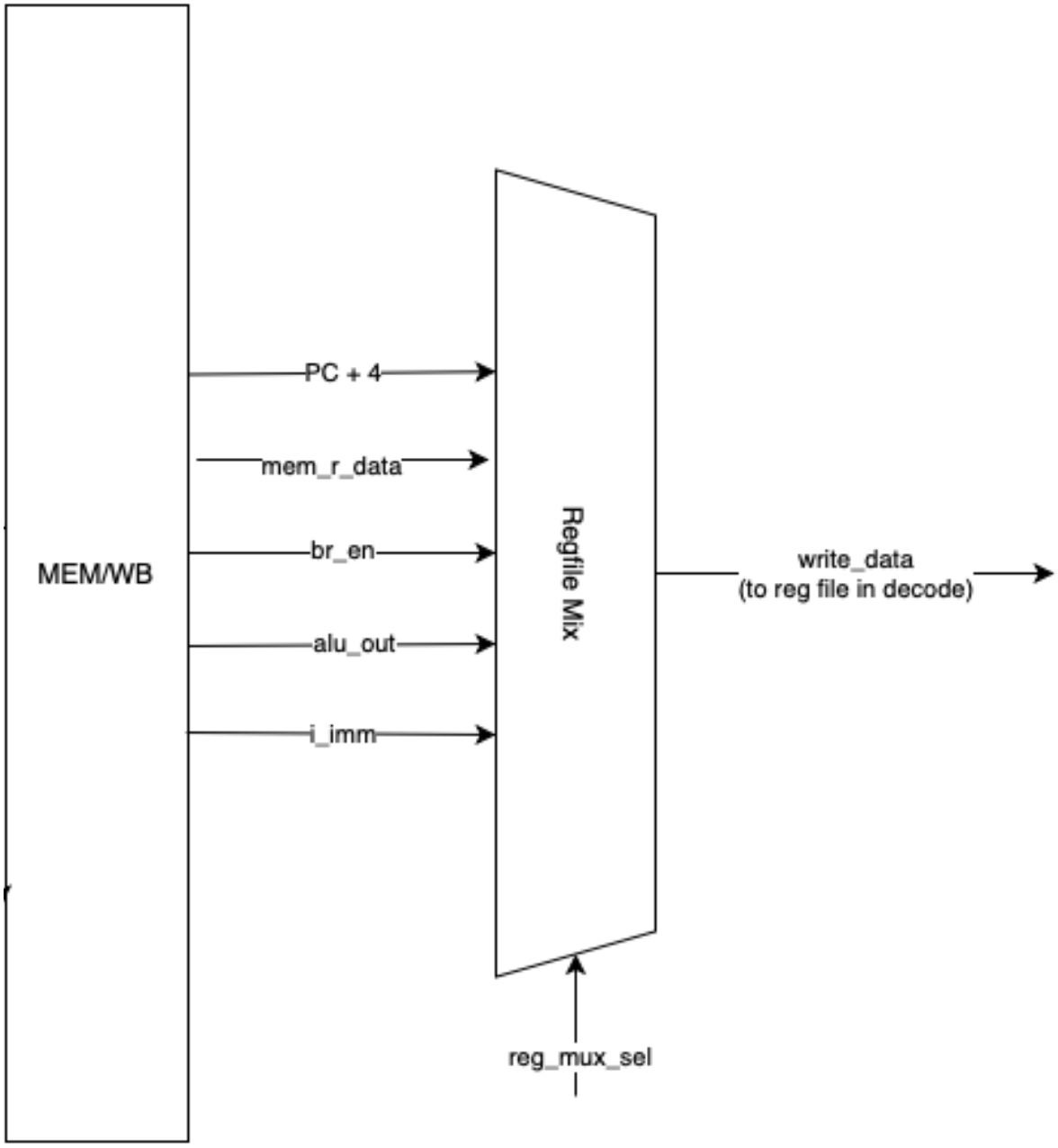
- EX - Execute, which executes the instruction/operation pr calculates the address



- MEM - Memory Access, which accesses the memory operand and caches if the instruction requires the use of memory (read or write accesses)



- WB - Write Back, which finishes the instruction by storing the data/writing the result back to register.

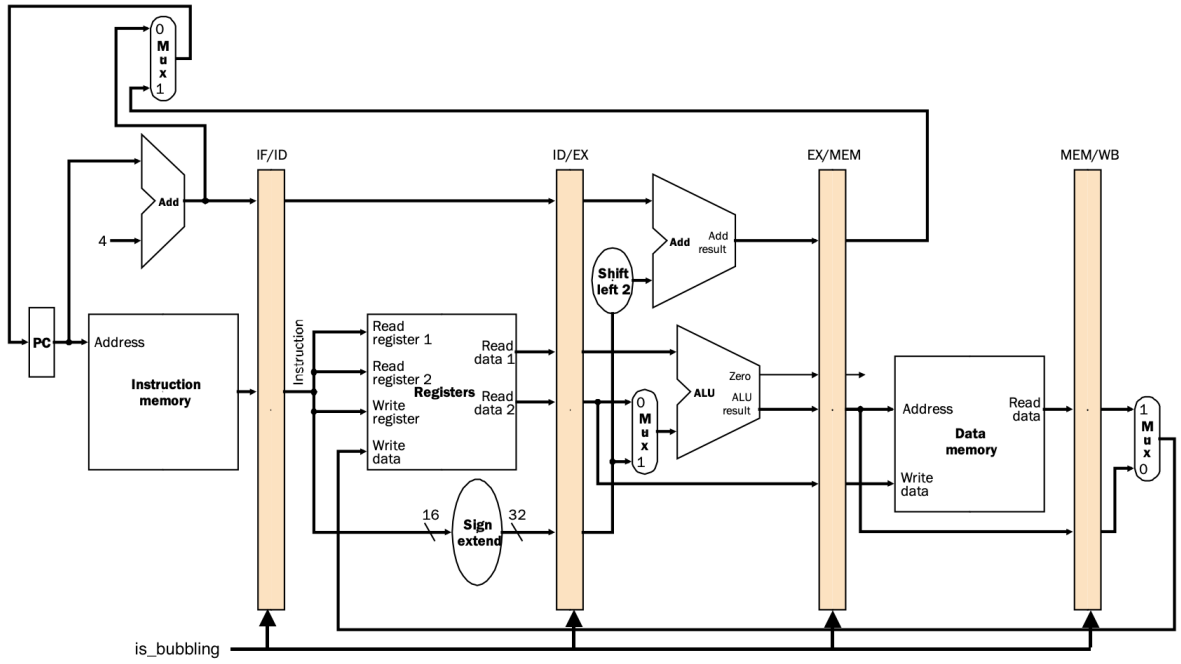


We used our processor from MP2 to implement the pipeline, as we split the original datapath into the 5 stages as shown above. We tested the pipeline by connecting magic memory and running test code at each stage of the pipeline, making sure the outputs at each stage of the pipeline was as expected before running the code given to us then checking the register in the ID stage to see if the output is correct.

2. Checkpoint 2

For Checkpoint 2, we added support for handling data hazards, forwarding, and L1 dual instruction and data cache system.

First we start with handling hazards. Before discussing hazards, it is worth talking about **stalls**. In order to stall a stage in the pipeline, we disabled the output signals to any of our memory devices. We already have a baseline for this in our "is_bubbling" signal. We also disabled all register load signals in that stage, so that the state of the stage does not change. This is done by also using our "is_bubbling" signal to prevent registers from loading



Control Hazards:

Detecting Control Hazards (occurs during branch instruction, when deciding whether to branch in the MEM stage)

1. We can always stall as soon as we see a branch/jump. (Not very good for timing as it takes 3 cycles per branch)
2. We can assume the branch is not taken, but if the branch is actually taken, we proceed to edit the following/flush the instruction by:
    a. IF: zero out instruction field in IF/ID
    b. ID: zero out control lines
    c. EX: zero out alu_out, rs1, rs2

    We are effectively turning these three instructions into nop's.
3. Finally, we can create a history table to predict whether or not a branch will be taken. DIfferent history tables for loops and jumps in order to predict the branch most accurately. If we are wrong, proceed to flush the next instructions like in method 2.

    The approach that we are using for CP2 is item #2, assuming the branch is not taken.

Data Hazards:

Detecting Data Hazards

One way to address hazards is stalling. This is not ideal, but explained here.

Between instruction i and i + 1:
ID/EX.write_reg == (IF/ID.rs1 || IF/ID.rs2) incorporate 3 bubbles
Between instruction i and i + 2:
EX/MEM.write_reg == (IF/ID.rs1 || IF/ID.rs2) incorporate 2 bubbles
Between instruction i and i + 3:
MEM/WB.write_reg == (IF/ID.rs1 || IF/ID.rs2) incorporate 1 bubbles
Bubbles/stalls stop instructions in the ID stage, which means we must stop fetching new instructions otherwise the PC and IF/ID register will be clobbered.

The better way to address data hazards is by using forwarding. It will allow the instructions to get the outputs of previous instructions before they are written to the regfile. This will allow us to remove many of the stalls that we would need above. The hazard has a reach of up to three instructions. We can effectively reduce this to 2 if we change the regfile to the 'transparent design', where the data that we read is the same as the data that we write on the event that we are writing into the regfile and rd==rs1/rs2. This is shown in the following change to the regfile code.

```
// Change to regfile
/* Regfile output logic OLD */
always_comb
begin
    reg_a = src_a ? data[src_a] : 0;
    reg_b = src_b ? data[src_b] : 0;
end


/* Regfile output logic NEW */

always_comb
begin
    reg_a = src_a ? ((load & (dest == src_a)) ? in : data[src_a]) : 0;
    reg_b = src_b ? ((load & (dest == src_b)) ? in : data[src_b]) : 0;
end
```

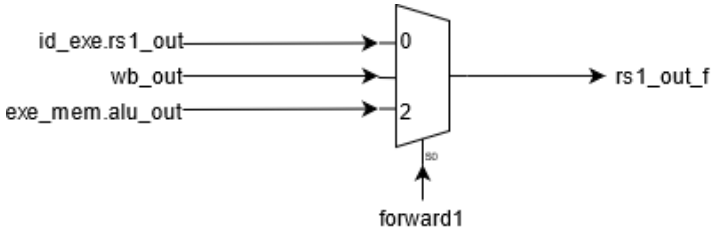
Data hazards are explained in more detail in the forwarding section.

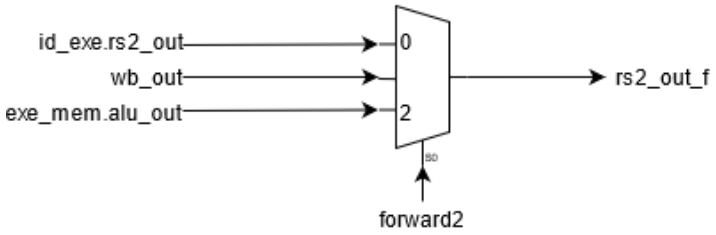
Structure Hazards:

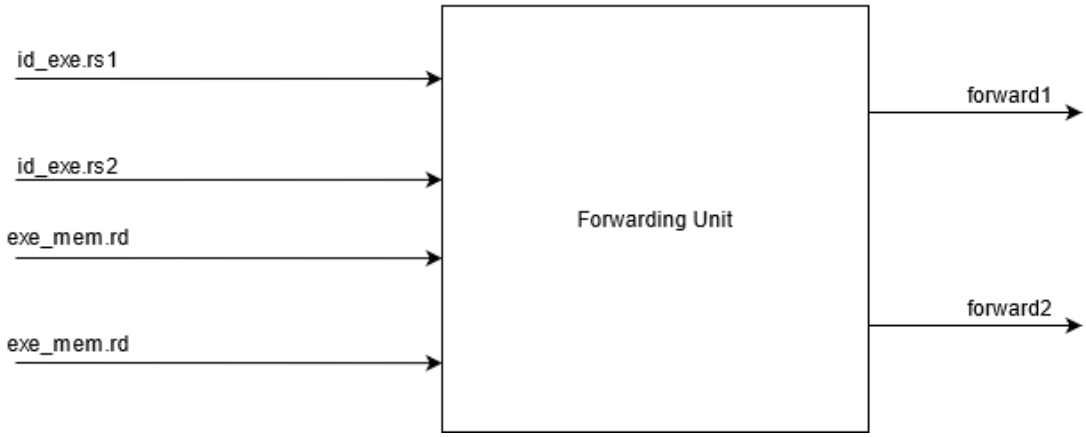
Structure hazards occur on cache misses. This will cause a very large delay on either the IF or MEM stage. In order to address this, all we need to do is stall the previous stages until the mem_resp is high. The way we stall is explained above.

If both caches miss at the same time, the data cache is served first. This will stall the previous stages (except IF). After MEM is served, the instructions in ID, EXE, and MEM will be allowed to complete while IF is stalled for the data read. Then the stages will again stall as they wait for new instructions.

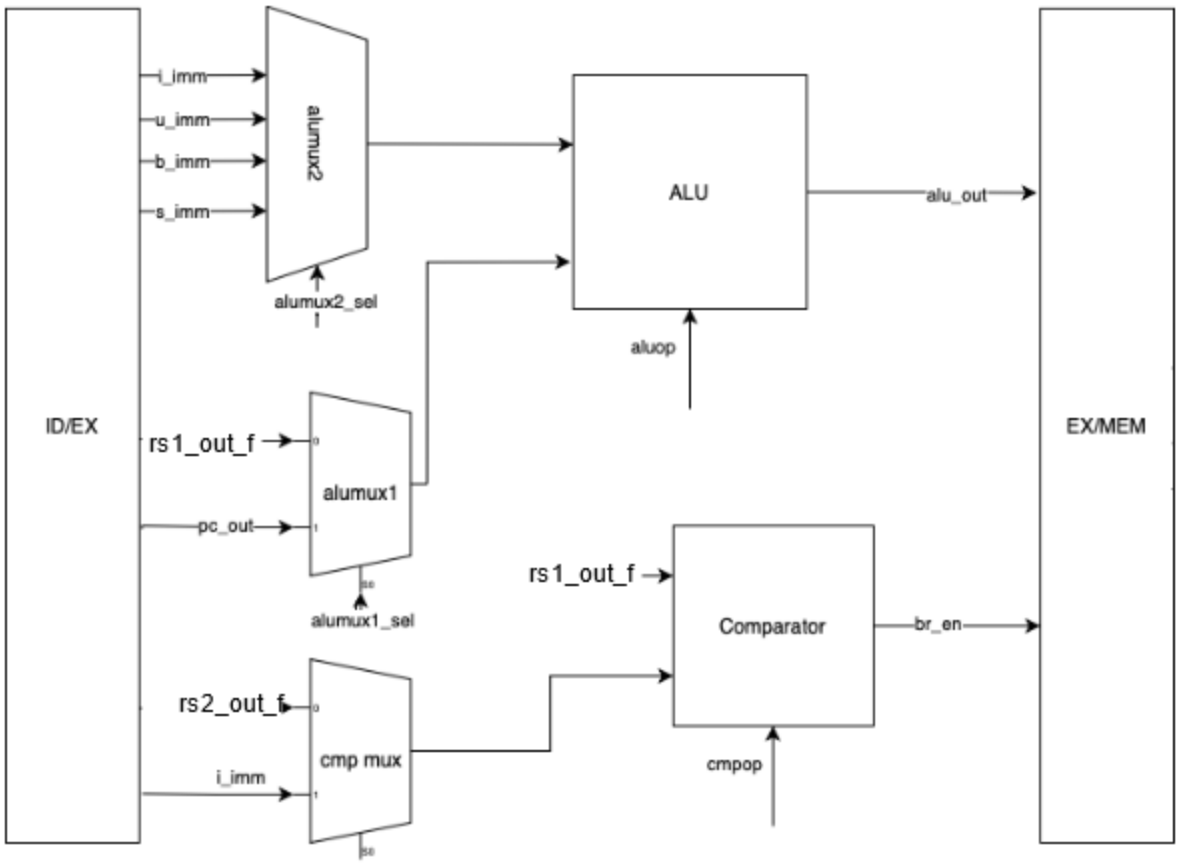
Forwarding Design:



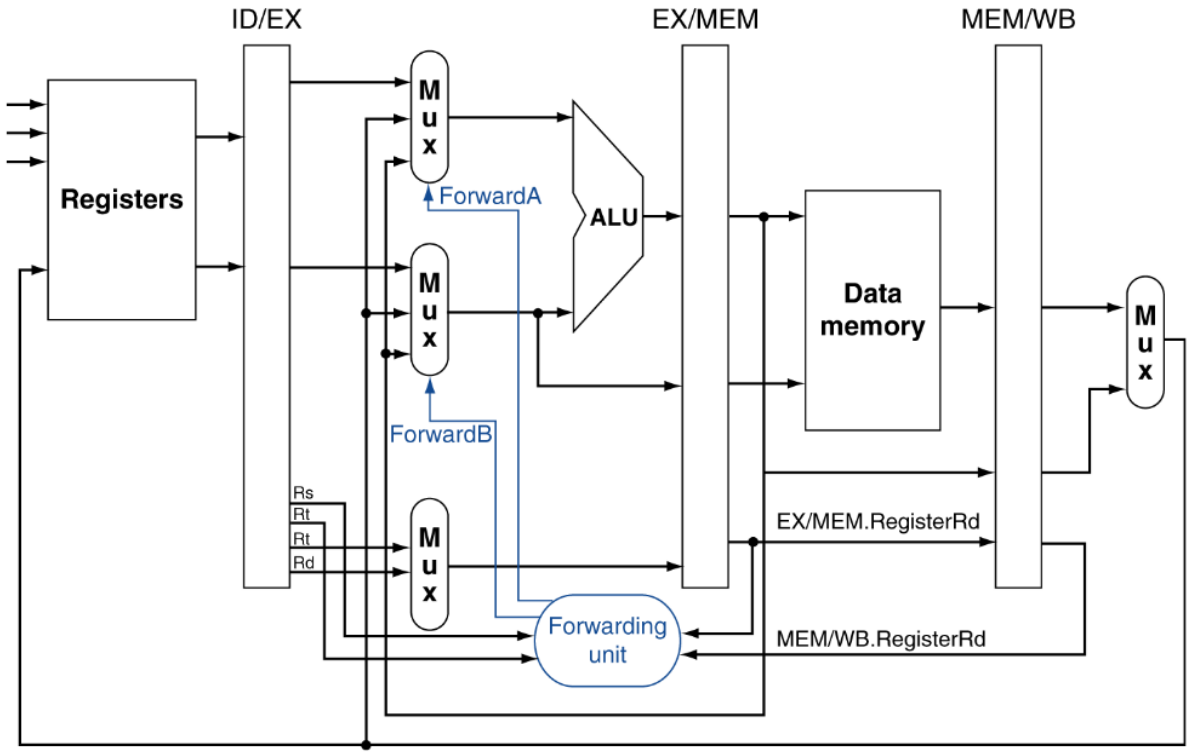




Above is the forwarding design. The signals are labeled with the name of the regfile that they come from. The outputs are ‘rs1_out_f’ and ‘rs2_out_f’. These signals permanently replace rs1_out and rs2_out in the MEM and WB stages.



The above image shows the changes to the EXE stage in the datapath to include forwarding.

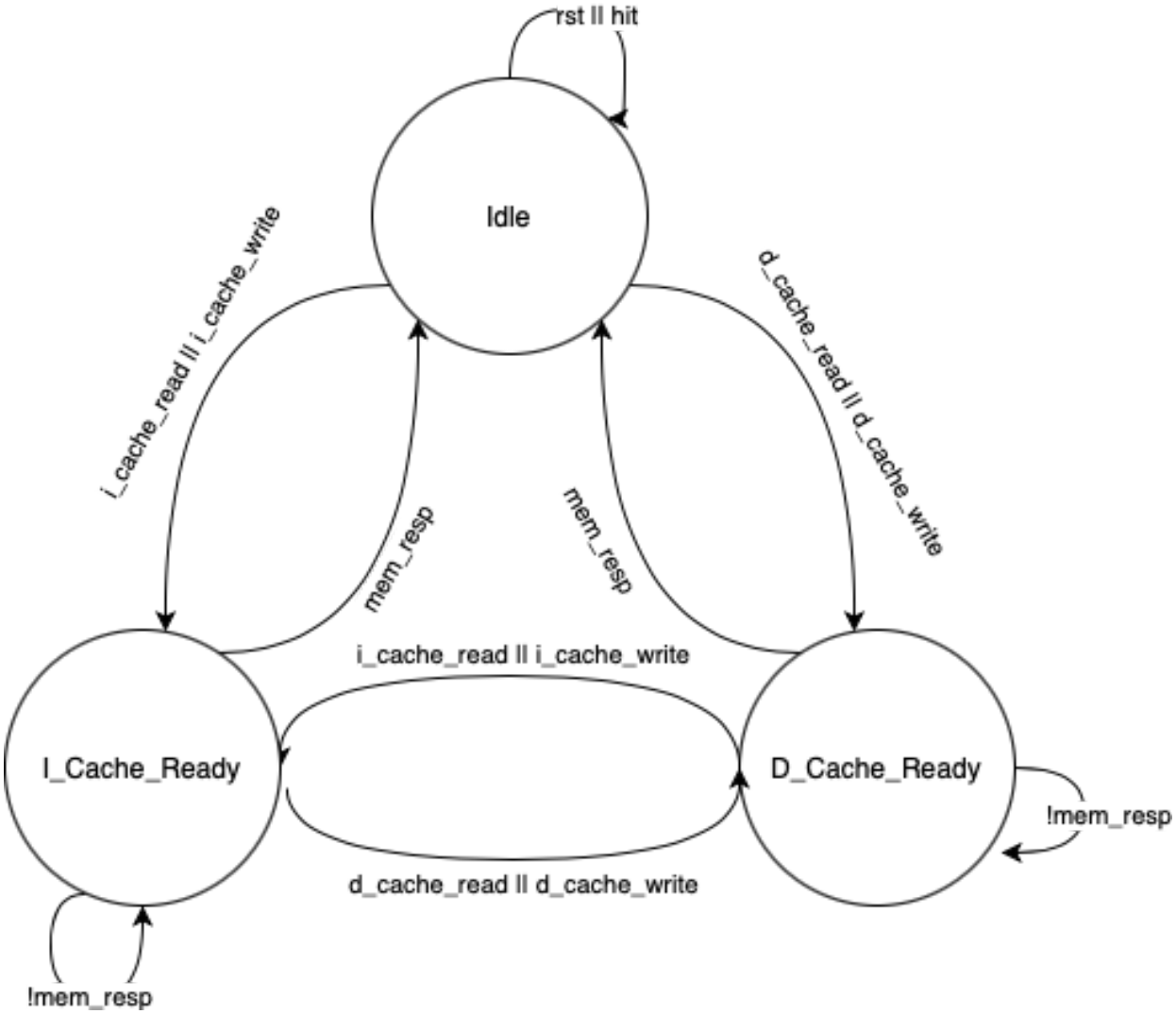


Above image is taken from lecture 5. While this diagram does not match our design exactly, it very clearly shows the connections into the forwarding unit. These are the same connections made in the previous two images.

```
module forwarding_unit
(
    input id_exe_regfile id_reg,
    input exe_mem_regfile exe_reg,
    input mem_wb_regfile mem_reg,

    input logic [4:0] mem_wb_reg,

    input rv32i_word wb_out,
    output rv32i_word rs1_out_f, rs2_out_f
);

logic [1:0] forward1, forward2;

always_comb begin : MUXES
    unique case (forward1)
        2'b00: rs1_out_f = id_reg.rs1_out;
        2'b01: rs1_out_f = wb_out;
        default: rs1_out_f = exe_reg.alu_out;
    endcase
    unique case (forward2)
        2'b00: rs2_out_f = id_reg.rs2_out;
        2'b01: rs2_out_f = wb_out;
        default: rs2_out_f = exe_reg.alu_out;
    endcase
end

always_comb begin

    // default
    forward1 = 2'b00;
    forward2 = 2'b00;

    // Check if EX hazard is true
    if (exe_reg.w_en & (exe_reg.rd != 0) & (exe_reg.rd == id_reg.rs1)) begin
        forward1 = 2'b10;
    end
    else if (mem_reg.w_en & (mem_reg.rd != 0) & (mem_reg.rd == id_reg.rs1)) begin // If EXE fails, then we can check mem condition
        forward1 = 2'b01;
    end

    // Check if EX hazard is true
    if (exe_reg.w_en & (exe_reg.rd != 0) & (exe_reg.rd == id_reg.rs2)) begin
        forward2 = 2'b10;
    end
    else if (mem_reg.w_en & (mem_reg.rd != 0) & (mem_reg.rd == id_reg.rs2)) begin // If EXE fails, then we can check mem condition
        forward2 = 2'b01;
    end

end
```

The above diagram and control address most of the concerns for data hazards. Forwarding allows us to execute consecutive operations with almost no issues. The updated data will be sent to a previous stage before it updates the regfile. One issue that this does not address perfectly is a load operation followed by an operation with the loaded data. In this case, we need to stall or bubble for one cycle before the arithmetic operation reaches the EXE stage. This will of course be longer on a cache miss.

Another part of Checkpoint 2 included the arbiter, which contains the logic to direct the cache data from the instruction cache and the data cache into the physical memory through the cache line adaptor. The arbiter is needed as only one of the caches can interact with the cache line adaptor, so providing a state machine, designed below, to direct the flow of traffic between the two caches and the adaptor is necessary in the design of the microprocessor system. Based on the read and write signals of the caches, we allocate interaction between the respective cache and memory. Testing was done similarly like the other parts of the microprocessor, by looking at waveforms of the signals to check for the correct change in read and write signals, as well as the correct data between the caches and the data



At State:

Idle: All signals set to 0

I_Cache_Ready:
i_cache data = data from cache adapter
i_cache resp = mem_resp
read_out = read signal from i_cache
write_out = write signal from i_cache
addr_out = address from the i_cache

D_Cache_Ready:
d_cache_data = data from cache adapter
d_cache_resp = mem_resp
read_out = read signal from d_cache
write_out = write signal from d_cache
addr_out = address from the d_cache

3. Checkpoint 3

Advanced design options

i. TAGE Branch Prediction

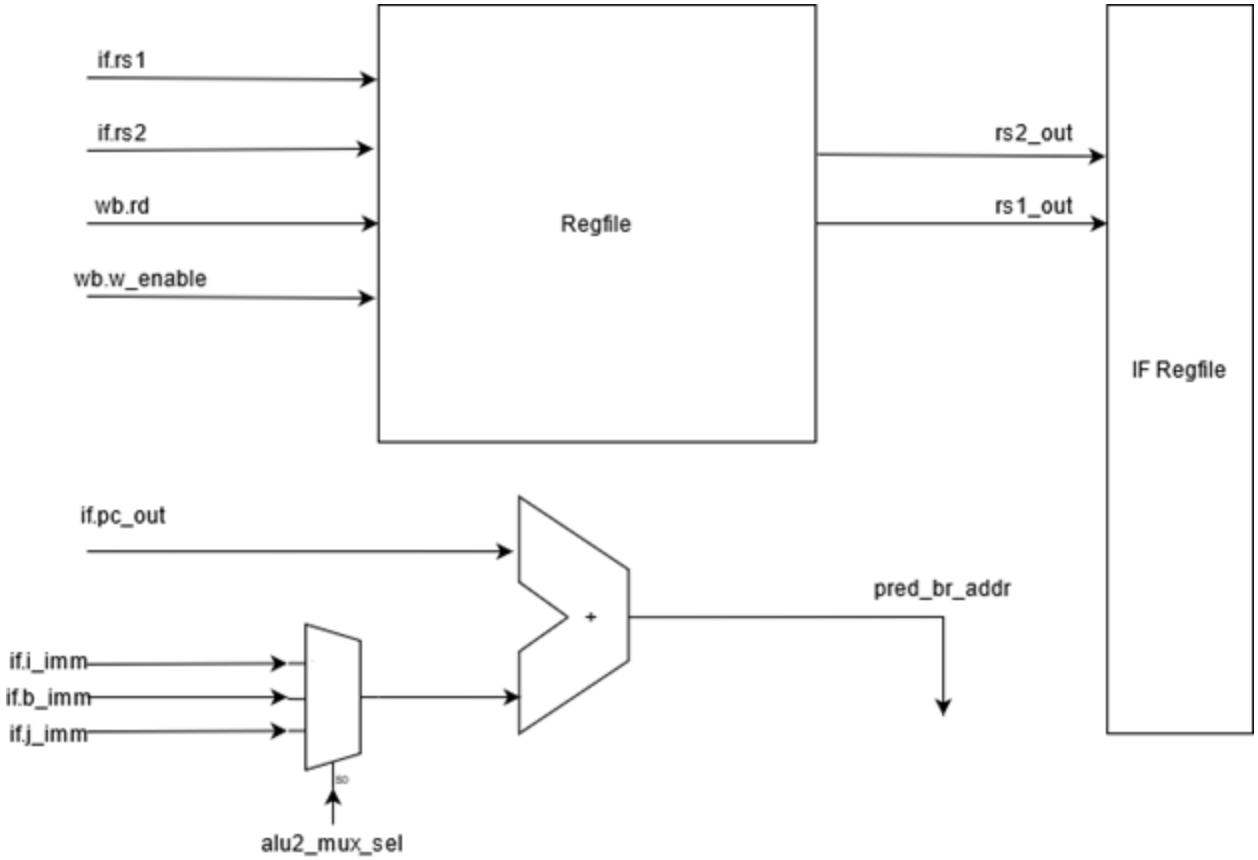
A. Design

We implemented a modified L-TAGE branch predictor in order to replace the static branch predictor used in CP2. Our branch predictor is ultimately an L-TAGE without the L (loop predictor) or the ‘useful bits’ used in the paper. It retains the idea of the address and history forming the tag, as well as the geometric series for history based predictors.

In order to do this, the only change needed was to change the black box that produces the predictions and to calculate the address in ID. The rest of the control logic was already present in CP2.

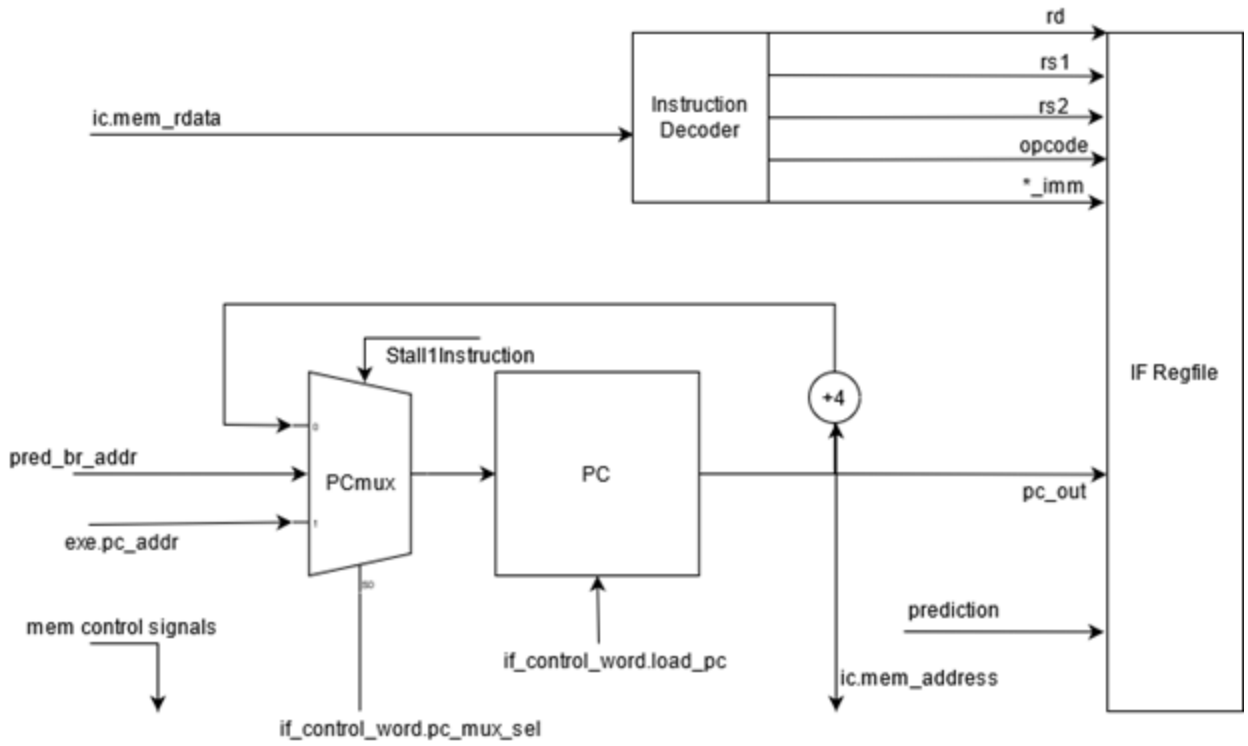
The previous black box for generating predictions always returned a 0. The new black box uses and updates a TAGE predictor. The arithmetic added to ID is very simple and shown below.

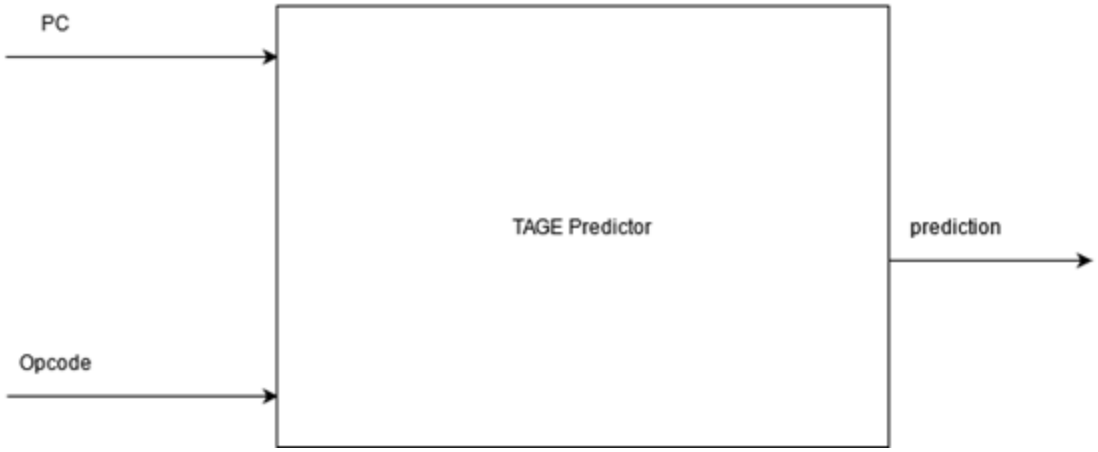
ID Stage



The ID stage now calculates the predicted next address on a branch and sends the value ‘pred_br_addr’ back to IF stage. This address is taken if the predicted direction is ‘1’. Note that this only works for BR and JAL instructions. Without a BTB, we cannot support JALR.

IF Stage





The IF stage is mostly unchanged from before. A new load signal for PC was added. This allows PC to stall exactly one instruction on a hit. Since we need to calculate the new address (we are not using a BTB), we need to wait one cycle to get the new address.

The pc_mux_select signal was expanded to allow a load of the input predicted address from ID.

The prediction still goes through a black box, which is discussed below.

The lack of a BTB does cause a 1 cycle stall, but allows

TAGE Predictor

The TAGE predictor is a type of tournament predictor that uses both local and correlated global predictors. The simple local predictor and the global predictor are unmodified from the lecture notes and are described in some detail here.

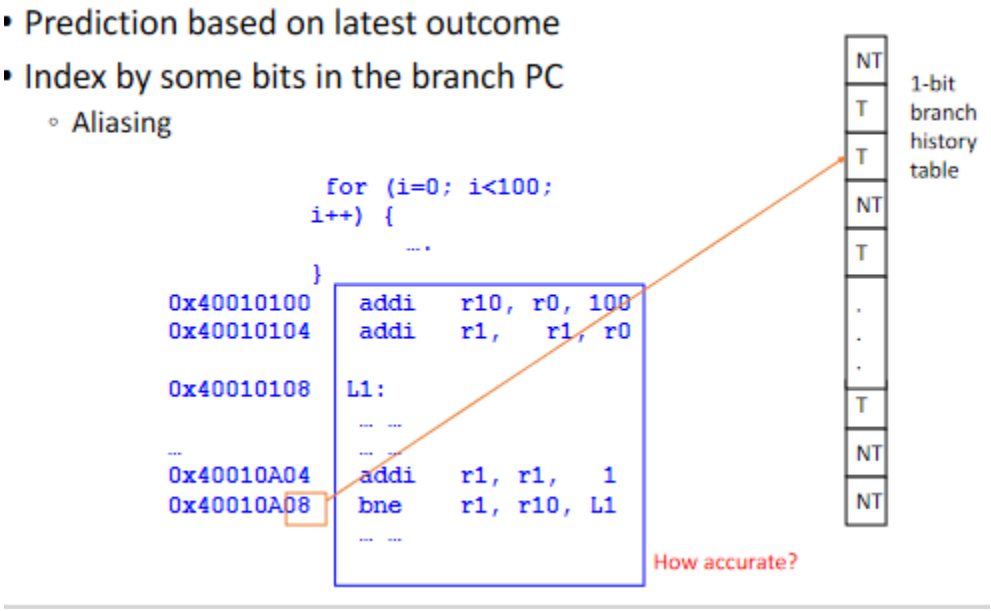
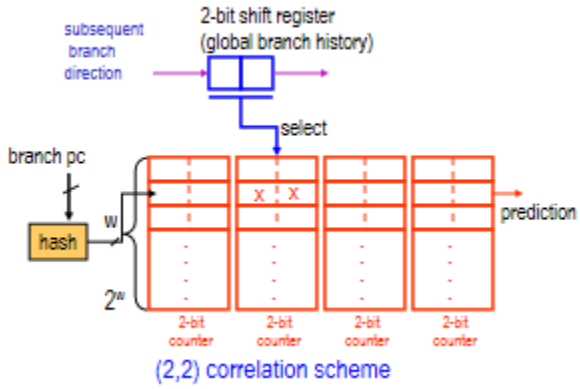
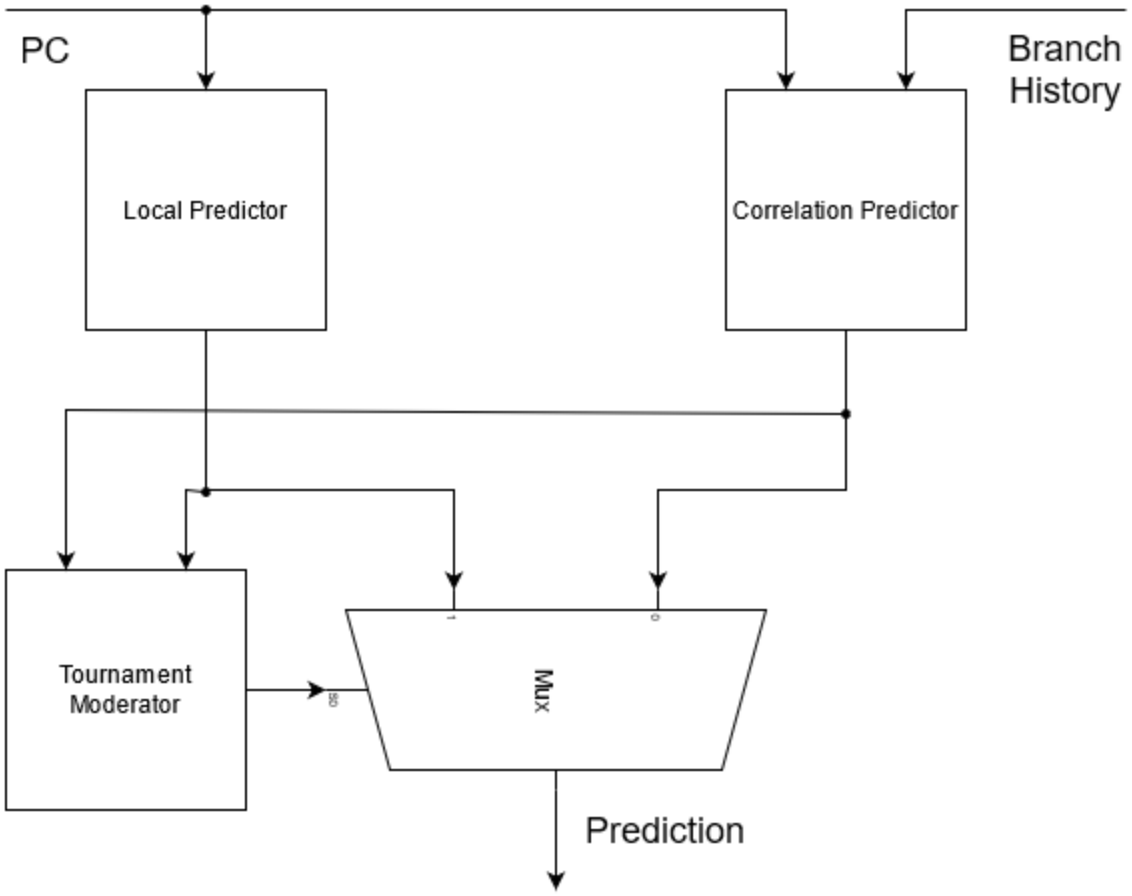


Image taken from the lecture notes. The simple predictor will follow this model. The size of the history table, as well as the hashing function have not yet been decided.



The above figure is taken from the notes on correlation prediction schemes.



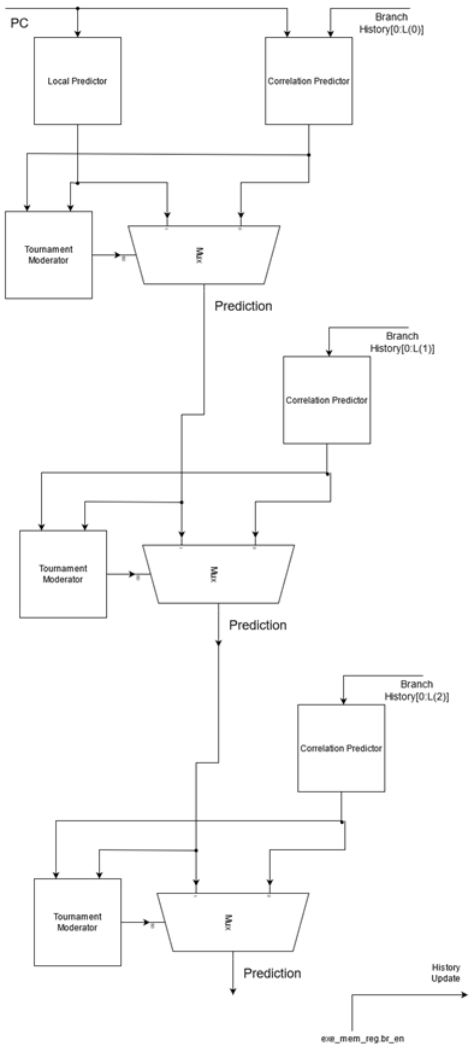
Short diagram showing the basic implementation of a tournament predictor. The final prediction is chosen from the two predictor results.

The local predictor and correlation predictor both take PC as input. The correlation predictors also take the branch history as an input, which is stored in a shift register. This branch history is variable and is updated whenever the opcode is BR.

The tournament moderator chooses the output prediction based on the input confidence. The confidence is based on the value of the input 2 bit predictions from the predictors. The rules for the moderation are simple.

- Pass through the two bit prediction value with the highest confidence
- A prediction of 11 or 00 have equal confidence and a higher confidence than 10 or 01, which are also equal.
- In the event of a tie, choose the output from the right side (correlation predictor).

The TAGE predictor takes the tournament scheme a bit further.



TAGE uses a series of correlation predictors that each use a different amount of history bits. In a similar way as the tournament predictor, the highest confidence prediction is passed through.

In order to make the TAGE predictor, we needed to create a parameterized correlation predictor. This means we need to have a variable number of registers to hold each of the prediction states, as well as an indexing scheme.

Below is a diagram of the TAGE predictor from the original paper by Andre Seznec. This does a better job of illustrating the TAGE and is included for clarity's sake. The TAGE I implemented is four components with equal spacing between each component of the L vector. No hash functions were used.

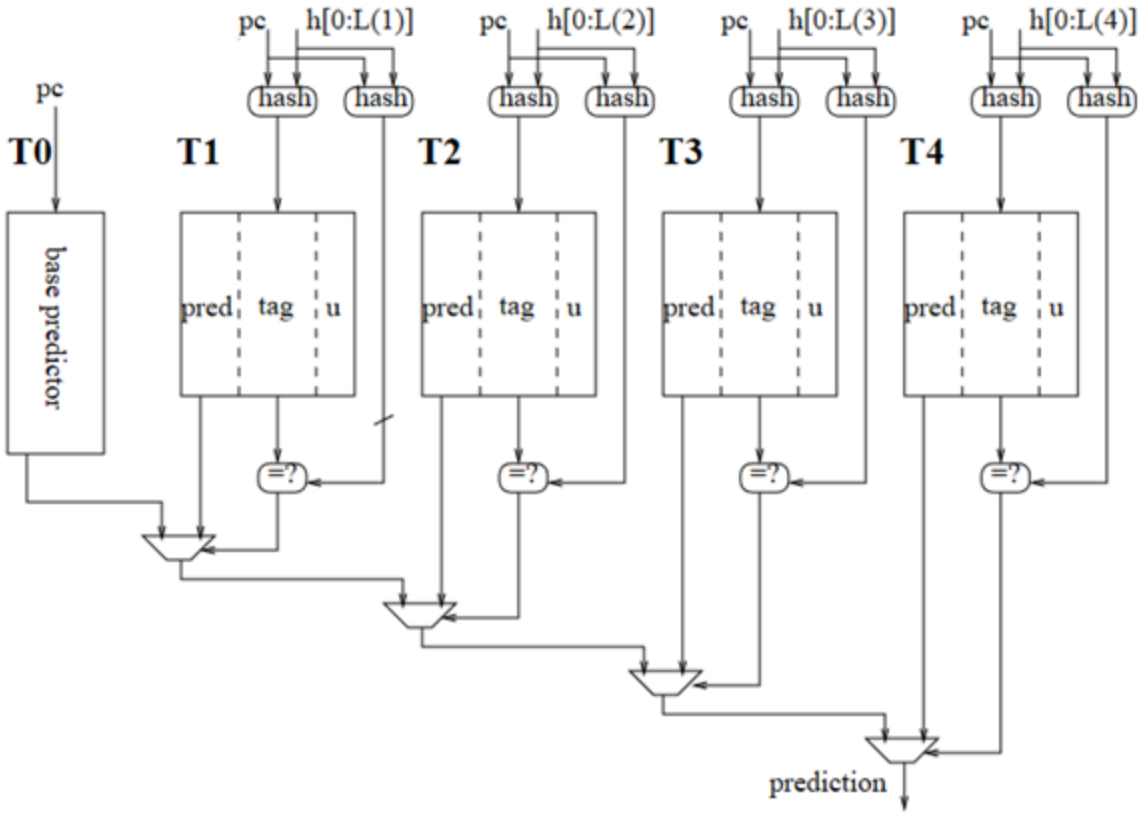


Figure 1: A 5-component TAGE predictor logical synopsis: a base predictor is backed with several tagged predictor components indexed with increasing history lengths. On an effective implementation, predictor selection would be performed through a tree of multiplexors.

Seznec , Andre. The L-TAGE Branch Predictor. In *Journal of Instruction Level Parallelism 9 (2007) 1-13*. 2007.

The L-TAGE branch predictor is very similar to a TAGE, except that it also includes a Loop predictor that stores the number of times a loop branches. We chose to not include this loop predictor as quite frankly, it does not seem to be worth the effort for the small performance boost we might experience.

TESTING

The tournament branch predictor was tested on the different provided test codes. The performance of the predictor does not have to be rigorously tested, as a failure in the prediction process only costs cycles. Regardless, we tested the predictor using different initial state values and tested the transition between states. The mp3-cp2.s code provided the best results, as the branches are almost always taken. The static branch predictor from CP2 served as the baseline.

Predictor	Hits	Misses
Static-0	6	581
TAGE Config 1	137	450
TAGE Config 2	581	6

The code was also profiled against a simple test code, provided below.

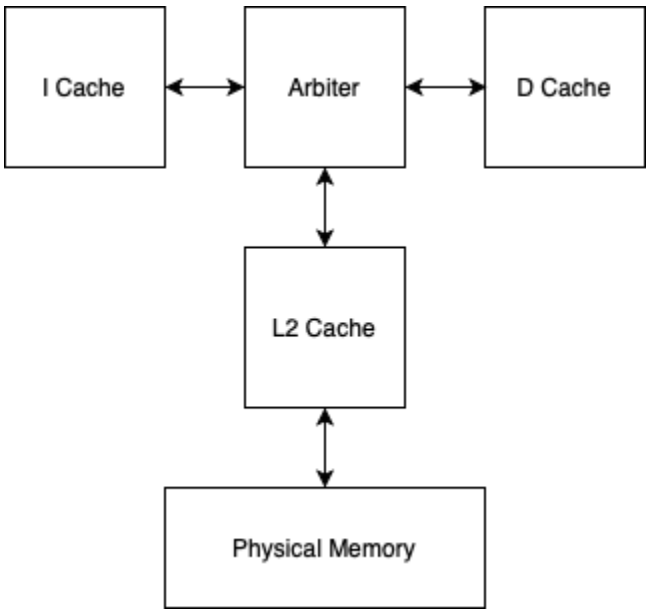
```
    li x1, 255

taken_branches:

    add x1, x1, -1
    bge x1, x0, taken_branches
    and x3, x3, 0
    and x2, x2, 0
    and x1, x1, 0
    li x3, 255
    li x2, 1
    li x1, -1

branch_two:
    bge x3, x2, positive
    ble x3, x1, negative

    beq x3, x0, done

positive:
    not x3, x3
    add x3, x3, 2
    j branch_two

negative:
    not x3, x3
    add x3, x3, 0
    j branch_two

done:
    j done
```

Predictor	# Taken	# Not Taken	# Correct	# Incorrect	Accuracy (%)
Local Predictor	510	130	545	87	86.2
Correlated-History = 1	510	130	545	87	86.2
Correlated-History = 2	508	132	632	8	99.0
Correlated-History = 3	508	132	632	8	99.0
Final Prediction	508	132	632	8	99.0

i. Parameterized L2 Cache

A. Design

With the second advanced feature we implemented a parameterized L2 Cache. The L2 Cache is parameterized by the ways and sets, tested with a combination of 1, 2, 4, 8 and 16 ways and 1, 2, 4, and 8 sets.



The above figure shows the location of the L2 Cache in respect to some of the other parts of the microprocessor

In our design document submitted before the implementation of the parameterized cache, we included a design for the 4 way set associative L2 cache, which helped assist the implementation of the parameterized cache. We first started with the cache implemented in MP3, or the 2-way set associative cache, and worked to expand the cache we created before into a 4 way set associative cache by first doubling the parameters of the 2-way cache into 4 arrays instead of 2. With the expansion from 2 arrays each to 4 arrays each, the signals controlling the loads, reads and writes signals of the arrays needed to be changed into 4 signals from the 2 signals controlling each of the arrays, and the muxes controlling the data_out, hit (which turned into USE), dirty_out, and tag_out signals required 4:1 muxes instead of 2:1 muxes.

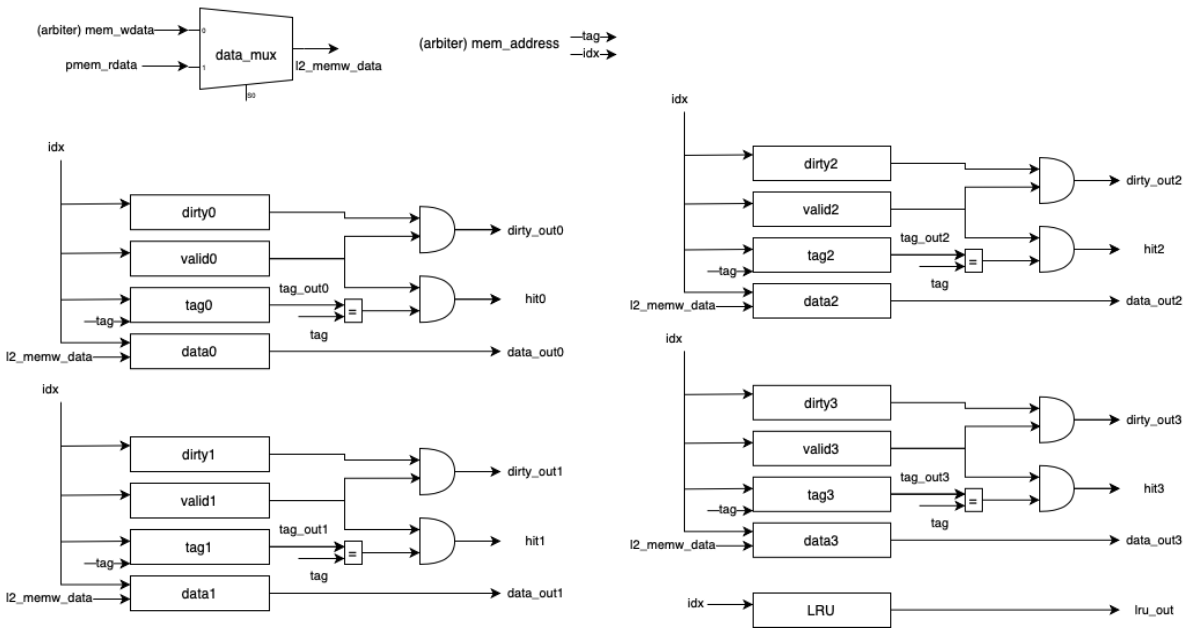


Figure: Arrays from the datapath of the 4 way set associative cache

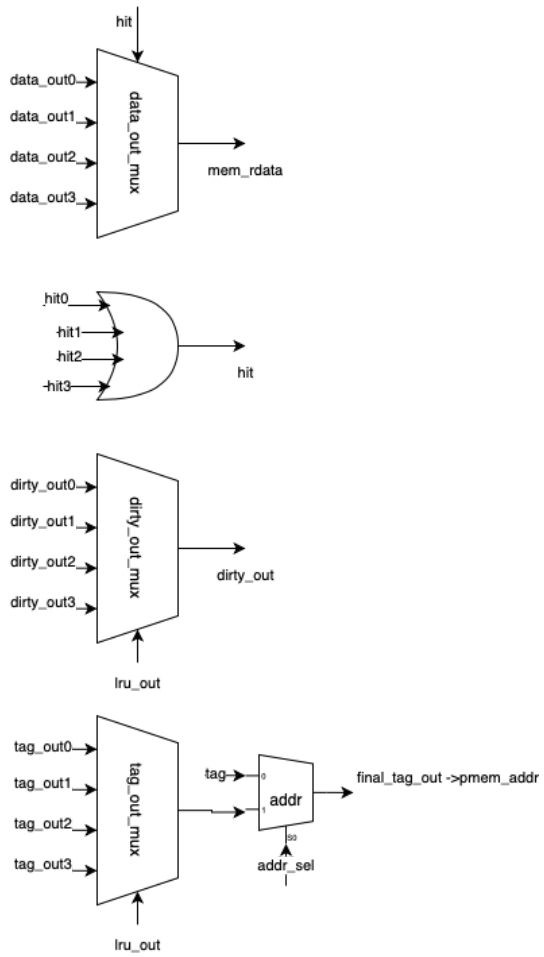


Figure: Muxes required in the datapath of the 4-way set associative cache.

A new feature in the 4 way set associative cache compared to the 2 way set associative cache is the implementation of the Pseudo LRU algorithm, which differs from the original LRU array that was used in the 2 way set associative cache. We used the LRU algorithm from lecture and emulated the pLRU tree using a series of two muxes to choose between the ways based on the LRU output and the USE signal.

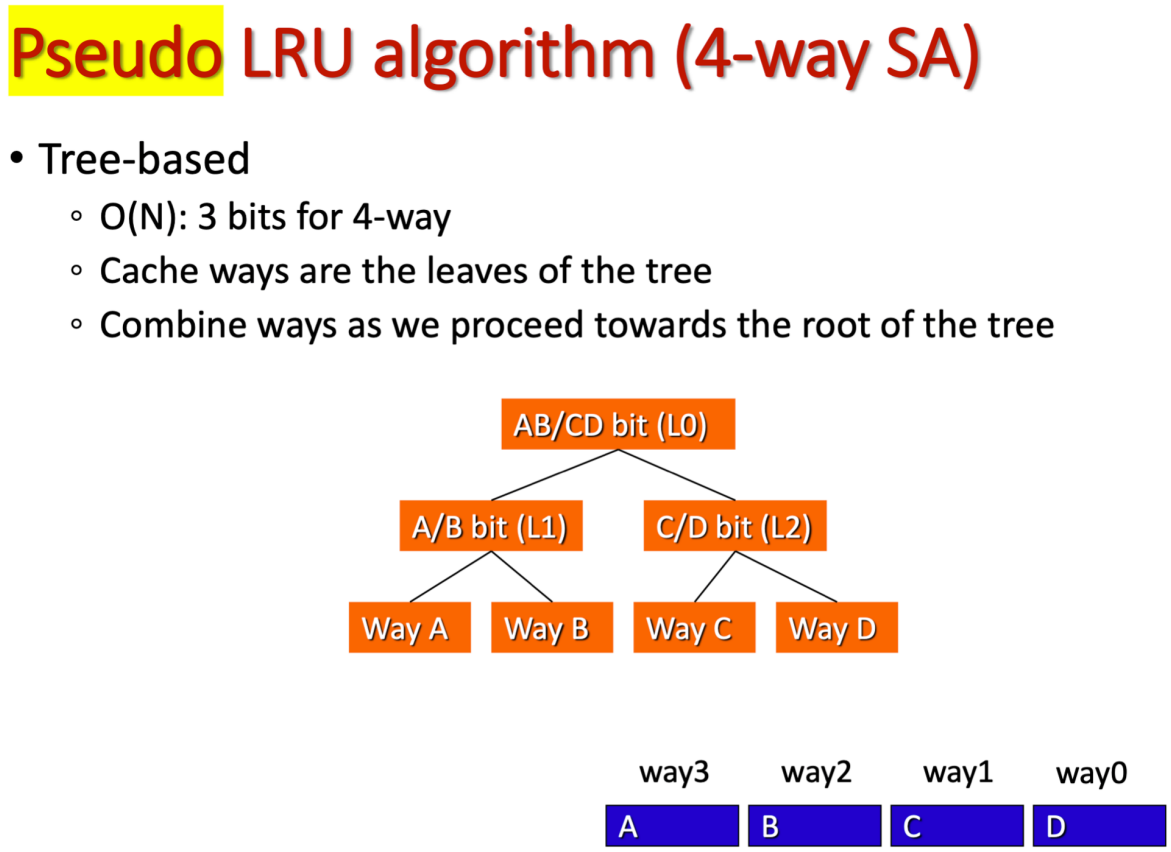


Figure: Pseudo LRU algorithm from lecture

After successfully implementing the 4 way set associative cache algorithm, we moved to working on the parameterized L2 cache, with the number of ways and sets being customizable. It was much easier to work on the parameterized L2 cache after the implementation of the 4 way cache as it was just continued expansion of the array signals, but most of the work was done with the parameterization of the pLRU tree. The parameterized pLRU tree was first built based on the number of ways and sets on every clock cycle, then based on the USE and write signals, a way is selected out of the parameterized number of ways for replacement.

B. Testing

Testing for the parameterized cache worked in parts. Since we knew that the L2-cache was just an extension of the L1 cache, except it connected the arbiter and the cache line adaptor instead of the CPU and arbiter, like the instruction and data caches did. We used the given cache, disabled the mem_byte_enable signal that was used in the original 1-way cache, and disconnected the customized given cache from the bus adapter. By disconnecting these signals in the datapath of the cache, we were able to get a working L2 cache relatively easily with just the provided cache with very little testing.

After we got a working L2 cache and confirmed that the removal of certain signals and the connection of the arbiter and the cache line adaptor to the L2 cache was all that was needed to make the L2 cache work, we then worked to test the 4 way cache using the MP3 code. In testing the new 4-way cache, we mostly looked at waveforms, making sure that the 4-way cache exhibited similar behavior to the working 2-way cache from MP3. Bugs we found with the 4-way cache mainly had to do with the pseudo LRU code we added to the 4-way cache system, as it was the only new code added to the cache (all of the other signals were just an expansion of previously declared signals from the 2-way cache). When testing the new pseudo LRU case statement, we looked at waveforms to check for the correct output of the case statement so the correct data array was being replaced properly. By reading the waveforms, we were able to diagnose our bugs within the pseudo LRU array, provide the correct bits that needed to be used in the case statement, then correctly run the 4 way set associative cache with the MP3 environment.

Testing the parameterized cache took more effort, as we had to ensure that all combinations of parameterized ways and sets were not only working within the MP3 environment, but within our MP4 environment. During testing, we did not run into many bugs involving other signals that interacted with the arrays, since we handled most of those signals when developing the 4 way cache, and the parameterized cache was a continued expansion of said signals, but some bugs that we encountered involved the pseudo LRU. The bugs within the new pLRU tree came mostly from the lower ways, as we did not have to cover a pLRU tree with the given cache nor the cache from MP3, so creating separate cases/using the original LRU array instead of a pLRU for the 1 and 2 way parameters helped fix our problems with the pLRU.

C. Performance analysis

With performance analysis, we added counters to produce the amount of times the L2 cache has been accessed, and how many times the L2 cache entered the miss state. Within the test bench, we took the ratio of misses with the total amount of accesses which gave us the percent chance of misses within the L2 cache compared to the total amount of hits. We proceeded to run the same test code (mp4-cp3.s) on four different combinations of ways and sets, recording the miss percentage of each of the combinations. We manually calculated the hit percentage (taking the complement), to gather the following data.

	2 way 4 set	**4 way 4 set**	**4 way 8 set**	**8 way 16 set**
Miss %	72.3	39.3	4.65	2.96
Hit %	27.7	60.7	95.35	97.04

Clearly from the data, as set associativity increases, miss percentage decreases dramatically, with ways and sets both having an effect on the amount of misses there were relative to the number of cache accesses. The parameterization had a slight effect on timing, as although there were less misses with the caches as set associativity increased there were much more accesses and much more resources that was needed in order to simulate a cache with higher set associativity, as well as the increased cycle time to access many more arrays. We found that higher set associativity theoretically decreased the amount of

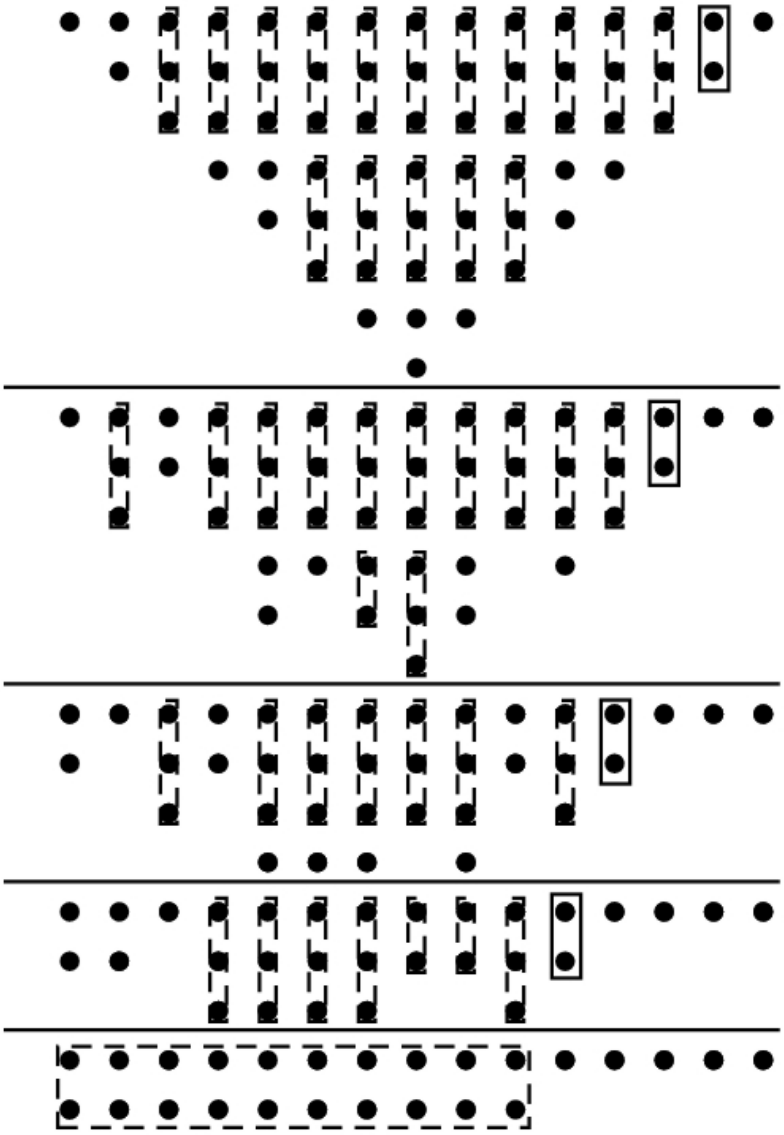
conflict misses, as shown in the following diagram, but we think that if the system can spare the cycle time (like with the L2 Cache) and the system spend the power/area to make the tag fetch and compare faster than data access, higher set associativity is worth the extra resources.

	Cache Parameter		
Miss Type	**Cache Size**	**Block Size**	**Associativity**
Compulsory	More misses	Less misses	0
Capacity	Less misses	0	0
Conflict	0	More misses	Less misses

In the table (source: https://courses.cs.washington.edu/courses/cse378/02sp/sections/section9-2.html) , we can see that conflict misses do decrease as associtivity increases. This explains our data we recorded above, which show the decrease in misses as associativity goes up.

Wallace Tree Multiplier

Our final advanced feature we decided to implement is as Wallace Tree multiplier, which went through many iterations during our development. A smaller (8-bit) version of our final implementation is shown is shown below, since the 32 bit variant would be far too large to display.



An 8-bit reduced complexity Wallace Tree multiplier. Dots represent the partial product tree, boxes surrounding 3 dots are full adders, while boxes surrounding 2 dots are half adders.

A. Design

A Wallace multiplier uses full adders and half adders to reduce the partial product tree to two rows, and then a final adder is used to add these two rows of partial products. A traditional Wallace multiplier performs its operation in three steps. (1) Generate all the partial products. (2) The partial product tree is reduced using full adders and half adders until it is reduced to two terms. (3) Finally, a fast adder is used to add these two terms. Initially, we decided to implement a traditional wallace multiplier, but it ended up being slightly too slow. Rather than change the multiplier to take two stages, we decided to implement a slightly different variation, the reduced complexity wallace tree multiplier. The main difference with this form of the multiplier is that only full adders are used outside of the final stage, except once each stage in

the rightmost column with more than one element (represented by the solid box in the above image). The result of this is that the multiplier has reduced area due to the greater number of full adders, and reduces the size of the final adder through the half adders.

The multiplier itself sits in the EXE stage. It uses the output of the forward muxes as inputs, similarly to the ALU. the result is passed into a mux alongside the output of the alu, so that during multiplication the output of the multiplier is passed into the alu_out register, while otherwise it is the output of the ALU.

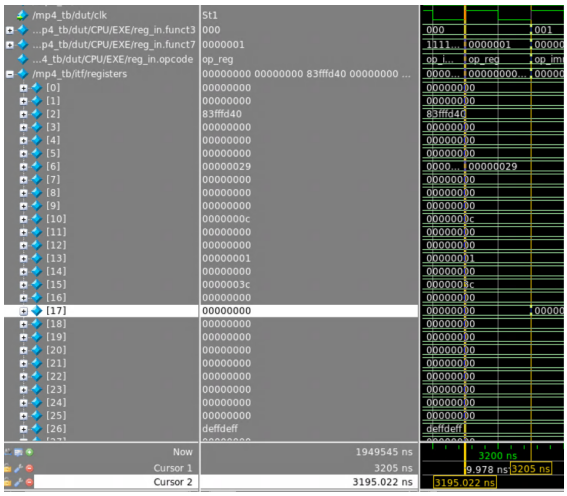
B. Testing

The multiplier was tested using a variety of methods. First it was tested with the comp2_m code, and when we confirmed that the results were correct, we moved to testing more thoroughly with assembly code, ensuring that mul, mulh, mulhu, and mulhsu all worked with positive*positive, positive*negative, negative*positive, and negative*negative multiplication. We also ensured that it encountered no bugs when using forwarding. However, this testing was not exhaustive, since we only did one multiplication of each type for every operation. The final phase of testing consisted of attaching the multiplier to a testbench directly, which allowed us to iterate through over 200,000 different operands, and when all of them produced the correct results we were confident that our multiplier worked properly. The testbench code is shown below

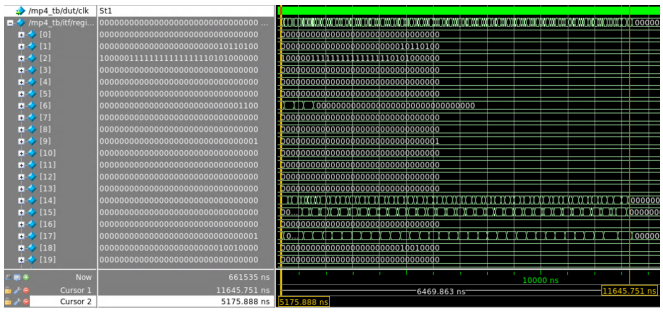
```
initial begin
        assign curr_product = w.curr_product;
        for(j = 0; j < 32'hFFFFFFFF; j = j+25418677) begin
                for(k = j%27382991; k < 32'hFFFFFFFF; k = k+27382991) begin
                        for(int i = 1; i < 4; i++) begin
                                mulop = i;
                                a = j[31:0];
                                b = k[31:0];
                                #1
                                // */
                                if(i == 1) begin
                                    if ($signed(f) != $signed(a) * $signed(b)) begin
                                        $error("Product Error! %d * %d != %d", $signed(a), $signed(b), $signed(f));
                                        $finish;
                                    end
                                    else $display("%d * %d = %d", $signed(a), $signed(b), $signed(f));
                                end
                                if(i == 2) begin
                                    if ($signed(f) != $signed(a) * $unsigned(b)) begin
                                        $error("Product Error! %d * %d != %d", $signed(a), $unsigned(b), $signed(f));
                                        $finish;
                                    end
                                    else $display("%d * %d = %d", $signed(a), $unsigned(b), $signed(f));
                                end
                                if(i == 3) begin
                                    if ($unsigned(f) != $unsigned(a) * $unsigned(b)) begin
                                        $error("Product Error! %d * %d != %d", $unsigned(a), $unsigned(b), $unsigned(f));
                                        $finish;
                                    end
                                    else $display("%d * %d = %d", $unsigned(a), $unsigned(b), $unsigned(f));
                                end
                                // */
                        end
                end
        end
        $finish;

end
```

C. Performance analysis

An example of multiplication being performed with and without the Wallace multiplier. Without the multiplier, the multiplication takes 656 cycles, while with the multiplier it only takes 1.



With Wallace multiplier using comp2_m code



Without Wallace multiplier using comp2_i

Conclusion

Going into this project, we had learned a bit about the pipeline and how important the pipeline microprocessor is to the world of computer organization and architecture. Over these past few months, we slowly designed and implemented the 5 Stage RISC-V pipelined processor from scratch, and week by week added additional new features to the processor. Some of these new features were for functional purposes and some features were for optimization purposes. We were able to achieve a functional RISC-V pipelined processor, but fell up short with the implementation of other optimizational features, as we could not successfully implement every advanced feature in one processor.

Nevertheless, we were able to implement three advanced features as standalone features, TAGE Branch Predictor, Parameterized L2 Cache, and Wallace Tree Multiplier, where we were able to analyze the performance and inner workings of each of the features. With the TAGE Branch Predictor, we were able to predicts the branch outcome for a particular branch scenario by looking up the history of what bits were chosen the last few times the same branch action occurred in a history table. By starting off with a simple branch predictor, we were able to fully create a full working TAGE Branch Predictor, and was able to compare the improved performance of the tournament branch predictor compared to the previous iterations of branch prediction implemented before it. With the Parameterized L2 Cache, we were able to expand on the dual L1 Cache system and provide a custom, set associative cache that improved hit rates and explored the tradeoffs between more cache lines and better performance. Finally, with the Wallace Tree Multiplier, we were able to start with a 8 bit design and expanded the design to fit 32 bits for our processor. In creating the multiplier, we were able to explore the trade-off between frequency and cycles, and was able to optimize the multiplier to be able to perform multiplication of 32 bit numbers within a certain frequency and cycle range.

Overall, the entire project was challenging but all three members were able to get hands-on experience with developing a pipeline microprocessor from scratch. With teamwork and the help of course staff and our mentor TA, we learned much more about the inner workings of the pipeline processor and how every part communicates with one another to create a fluid, adaptable machine. Throughout the few months of designing, implementing, and debugging, this project has had its ups and downs, but the processor will be an experience that none of us will never forget.