ECE 411: Computer Organization and Design

MP4 Final Report

Murat Altindag, Xinpei Jiang, Kelsey Chang

1. Introduction

This project utilizes information learned in the lecture to design a pipelined microprocessor to give students an understanding of the implementation details of a pipelined microprocessor and various associated features, as well as experience with evaluating the pros and cons of various design choices. This report is organized by checkpoint and records the progress, results, and designs associated with each checkpoint.

2. Project overview

The technical goal of the project is designing and then optimizing a pipelined microprocessor specifically implementing the RV32I instruction set, with the exception of FENCE*, ECALL, EBREAK, and CSRR instructions. This includes standard pipelining implementation challenges such as forwarding data to eliminate most data hazards and stalling and flushing mechanisms for control and data hazards. To then optimize our design, we implemented additional features to reduce latency, like a more complex memory hierarchy, branch predictor, and a prefetcher. We split the work to minimize the amount of merging conflicts we would have to resolve and meetings we need to have since our schedules often did not match well, and those with less work who completed it early helped others with theirs. As a result, our project is highly modular, which made testing different memory hierarchies very easy.

3. Design description

3.(a) Overview

For this project, we were given eight weeks to complete four checkpoints. We completed the base design in the first checkpoint, added caches in the second checkpoint, and implemented advanced features in the third checkpoint. The fourth checkpoint was the design competition, where we presented the final version of our project.

3.(b) Milestones

<u>Checkpoint 1:</u>

For the first checkpoint, we designed a 5-stage pipeline that would run the RV32I instruction set with no hazard detection or data forwarding. We designed our pipeline based on the course information (lecture slides, notes, etc.) with the stages of Fetch, Decode, Execute, Memory, and Write-Back. We were given a “Magic Memory” to use instead of caches for this checkpoint. We also made paper designs for the arbiter and the caches for the next checkpoint.

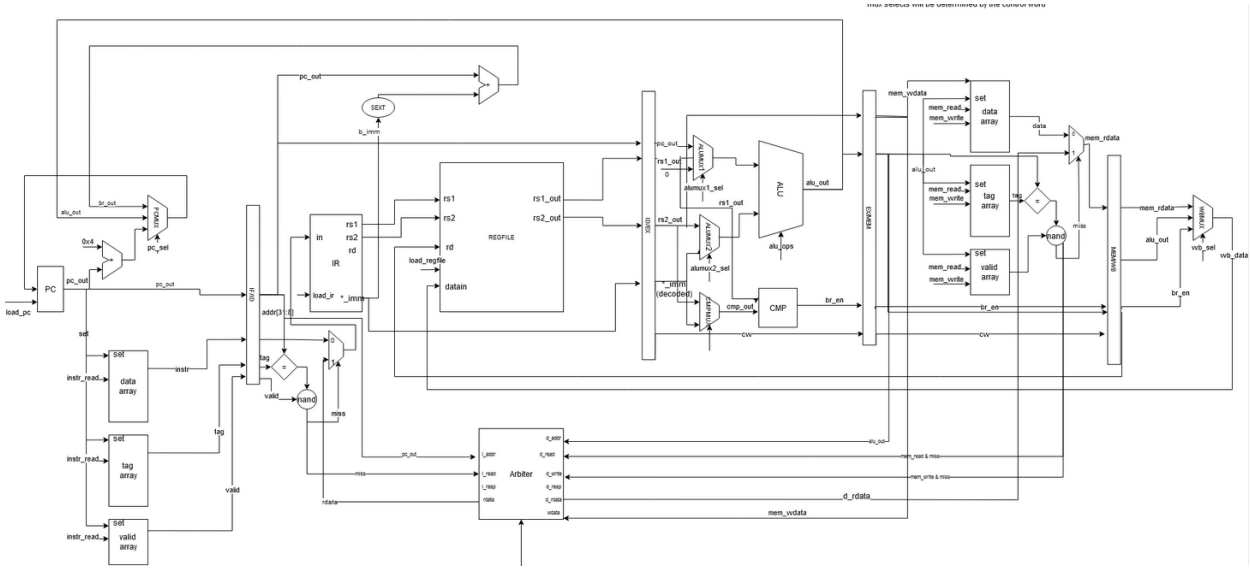


Figure 1: Checkpoint 1 Datapath (+caches and arbiter design)

- Fetch: Update the program counter and fetch the instruction from the instruction cache.
- Decode: Decode the instruction into opcode, rs1 (source register 1), rs2 (source register 2), rd (destination register), and imm (immediate value). Also, determine the potential next PC if the instruction is a branch.
- Execute: Execute arithmetic and logical operations and comparisons based on the inputs from the previous stage. If the instruction is a branch, determine if the branch is taken.
- Memory: Access the data cache to read or write data.
- Write-Back: Write the final result to the destination register.

We used the MP4 testcodes to test our design for correctness. We used waveforms that were generated to debug our code for this checkpoint.

Checkpoint 2:

For the second checkpoint, we implemented the instruction and data caches, and the arbiter into our design. We were given basic direct-mapped caches by the course staff. We decided to use these caches instead of the designs we had previously due to correctness concerns, as performance was not a priority. We made a state machine for our arbiter to serve the instruction cache and the data cache with equal priority.

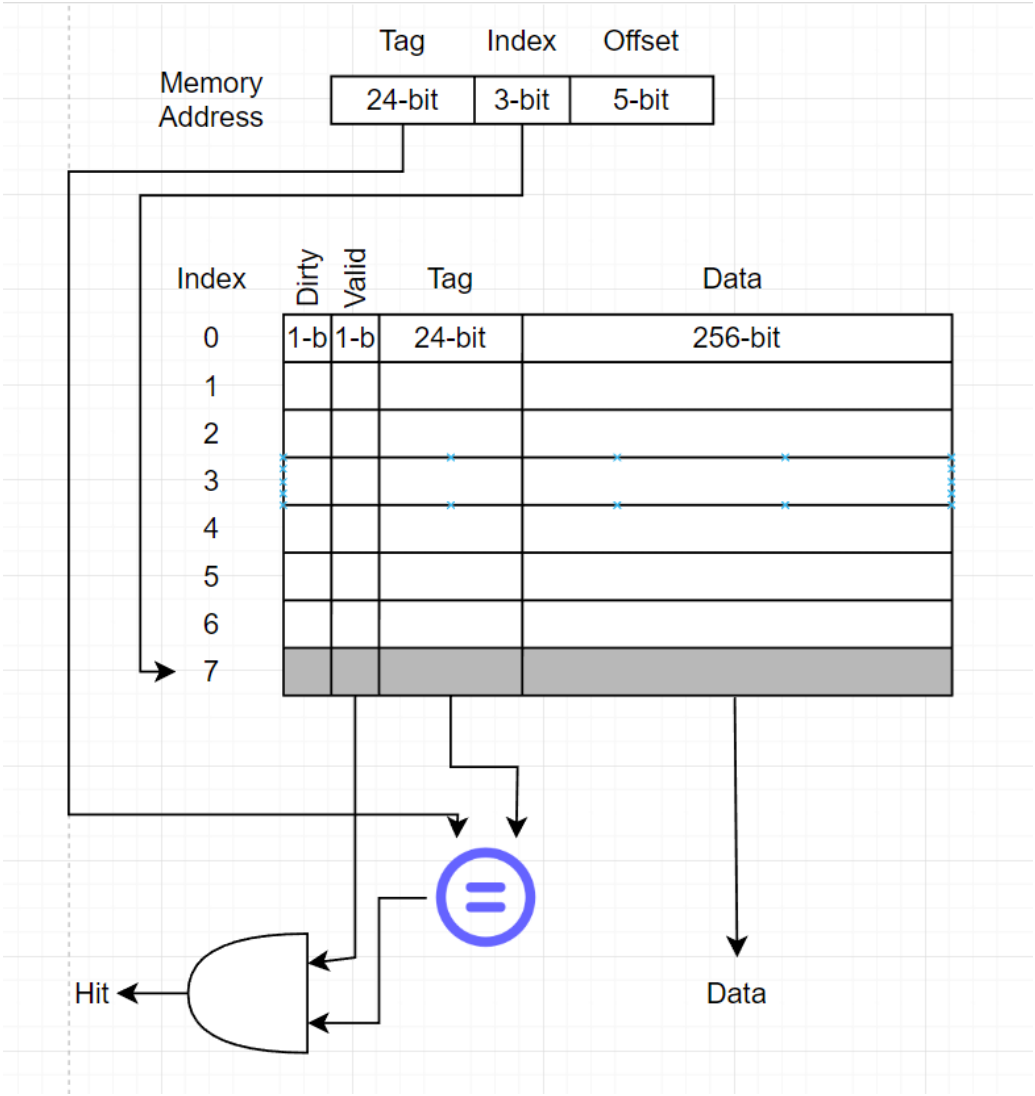


Figure 2: Given cache design

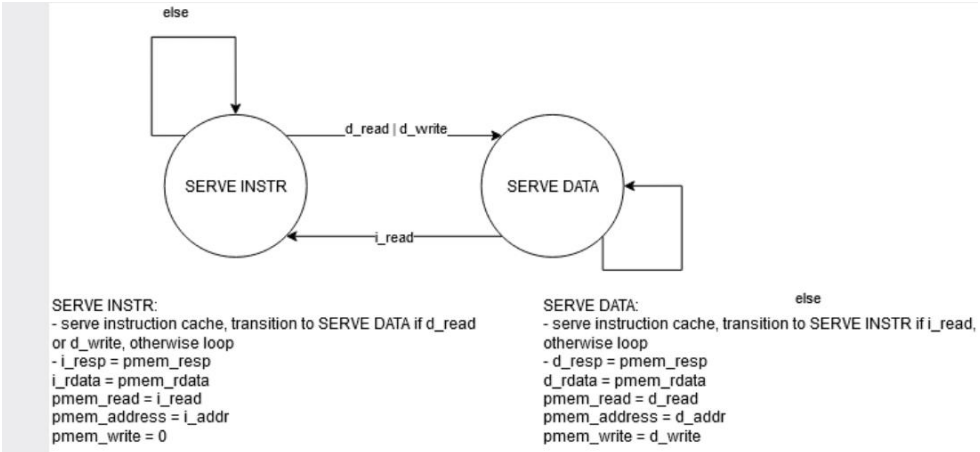


Figure 3: Arbiter state machine

We also implemented hazard detection, forwarding, and a static not-taken branch predictor.

```
Data Forwarding
    - EX Hazards (i.e. EX/MEM -> EX)
if (    EX/MEM.load_reg &
        (EX/MEM.rd = ID/EX.rs1) &
        EX/MEM.cw.opcode != op_load)
 alumux1_sel = f_rs1
 f_rs1 = ex_mem_data
if (    EX/MEM.load_reg &
        (EX/MEM.rd = ID/EX.rs2) &
        EX/MEM.cw.opcode != op_load)
 alumux2_sel = f_rs2
 f_rs2 = ex_mem_data
    - MEM Hazards (i.e. MEM/WB -> EX)
if (    MEM/WB.load_reg &
        ~(EX/MEM.load_reg & (EX/MEM.rd == ID/EX.rs1)) &
        (MEM/WB.rd == ID/EX.rs1)
  )
 alumux1_sel = f_rs1
 f_rs1 = mem_wb_data
if (    MEM/WB.load_reg &
        ~(EX/MEM.load_reg & (EX/MEM.rd == ID/EX.rs2)) &
        (MEM/WB.rd == ID/EX.rs2)
  )
alumux2_sel = f_rs2
f_rs2 = mem_wb_data
```

```
Hazard Detection
stall = {IF/ID, ID/EX, EX/MEM, MEM/WB} // 0 if not stalling
Flush:
    - for branch, flush IF (load_pc = 1, stall = 4'b1000)
    - for jmp, flush IF and ID (load_pc = 1, load_ir = 1, stall = 4'b1100)
Stall:
    - for memory stall in IF:
      while instr_mem_resp = 0,
      stall = 4'b1000
      load_pc = 0
      load_ir = 1
    - for memory stall in MEM:
      while data_mem_resp = 0,
      stall = 4'b1111
      load_pc = 0
      load_ir = 1
    - Read-after-Load:
      if ((ID.rs1 == EX/MEM.rd | ID.rs2 == EX/MEM.rd) &
          EX/MEM.cw.opcode = op_load
        )
      stall = 4'b1111
      load_pc = 0
      load_ir = 1
```

Figure 4: Pseudocode for Data Forwarding and Hazard Detection for CP2 (slightly modified later)

We used the MP4 testcodes, as well as the RVFI Monitor and the Shadow Memory, to test our design. These resources help us detect any memory issues at certain timestamps. We used these timestamps to check the waveforms and debug our code.

Checkpoint 3:

In this checkpoint, we implemented advanced design features of our choice. We decided that branch prediction is one of the most important optimizations we could work on. Thus, we attempted to design a tournament branch predictor.

We considered many other options to add to our design. We eventually decided to choose to add three new components to the memory hierarchy since we were able to independently design and test those components and easily combine them. These components were a unified L2 cache, a prefetching buffer, and an eviction write buffer. The details of these advanced design features are in the next section.

Checkpoint 4:

The final checkpoint was the design competition. For the competition, we chose to keep the L2 cache system and the branch predictor, as the other features did not positively impact performance. We combined these features to make our final design. Here are the synthesis results:

comp1.s:

```
$finish called from file "/home/murata3/411-mp4/mp4/mp4/hvl/top.sv", line 215.
$finish at simulation time           395861480
           V C S   S i m u l a t i o n   R e p o r t
Time: 395861480 ps
CPU Time:     24.540 seconds;       Data structure size:   0.3Mb
Sun Dec 18 22:19:59 2022
```

Compile Time

```
  ---------------------------------------------------------------------------
  data required time                                                    9.87
  data arrival time                                                    -5.70
  ---------------------------------------------------------------------------
  slack (MET)                                                           4.16
```

Timing Report (fmax = 171.232877 MHz)

```
  Combinational area:              54732.427182
  Buf/Inv area:                     2237.857978
  Noncombinational area:          121894.761695
  Macro/Black Box area:                0.000000
  Net Interconnect area:      undefined  (Wire load has zero net area)

  Total cell area:                176627.188877
  Total area:                 undefined
```

Area Report

```
---------------------------------------------------------------------------
                                   Switch   Int      Leak     Total
Hierarchy                          Power    Power    Power    Power    %
---------------------------------------------------------------------------
mp4                                2.48e+03 5.90e+03 3.32e+06 1.17e+04 100.0
```

Energy Report

comp2_i.s:

```
$finish called from file "/home/murata3/411-mp4/mp4/mp4/hvl/top.sv", line 215.
$finish at simulation time           916736920
           V C S   S i m u l a t i o n   R e p o r t
Time: 916736920 ps
CPU Time:     58.450 seconds;       Data structure size:   0.3Mb
Sun Dec 18 22:43:01 2022
```

Timing Report

```
  ---------------------------------------------------------------------------
  data required time                                                    9.87
  data arrival time                                                    -5.70
  ---------------------------------------------------------------------------
  slack (MET)                                                           4.16
```

Timing Report (fmax = 171.232877 MHz)

```
Combinational area:              54732.427182
Buf/Inv area:                     2237.857978
Noncombinational area:          121894.761695
Macro/Black Box area:                0.000000
Net Interconnect area:      undefined  (Wire load has zero net area)

Total cell area:                176627.188877
Total area:                 undefined
```

Area Report

```
---------------------------------------------------------------------------
                                   Switch   Int      Leak     Total
Hierarchy                          Power    Power    Power    Power    %
---------------------------------------------------------------------------
mp4                                2.60e+03 6.43e+03 3.20e+06 1.22e+04 100.0
```

Energy Report

comp3.s:

```
$finish called from file "/home/murata3/411-mp4/mp4/mp4/hvl/top.sv", line 215.
$finish at simulation time           590567080
           V C S   S i m u l a t i o n   R e p o r t
Time: 590567080 ps
CPU Time:     30.740 seconds;       Data structure size:   0.3Mb
Sun Dec 18 22:48:23 2022
```

Compile Time

```
--------------------------------------------------------------------------
data required time                                                  9.87
data arrival time                                                  -5.70
--------------------------------------------------------------------------
slack (MET)                                                         4.16
```

Timing Report (fmax = 171.232877 MHz)

```
Combinational area:              54732.427182
Buf/Inv area:                     2237.857978
Noncombinational area:          121894.761695
Macro/Black Box area:                0.000000
Net Interconnect area:      undefined  (Wire load has zero net area)

Total cell area:                176627.188877
Total area:                 undefined
```

Area Report

```
--------------------------------------------------------------------------------
                                   Switch   Int      Leak     Total
Hierarchy                          Power    Power    Power    Power    %
--------------------------------------------------------------------------------
mp4                                2.01e+03 5.15e+03 3.29e+06 1.04e+04 100.0
```

Energy Report

3.(c) Advanced design options

I. Branch Prediction

I.(A) Design

We implemented a local history predictor in our CPU to predict the results of branch instructions and a direct-mapping BTB (Branch Target Buffer) to record the known branch instructions so that the next time it is executed; a prediction can directly be made at the IF stage.

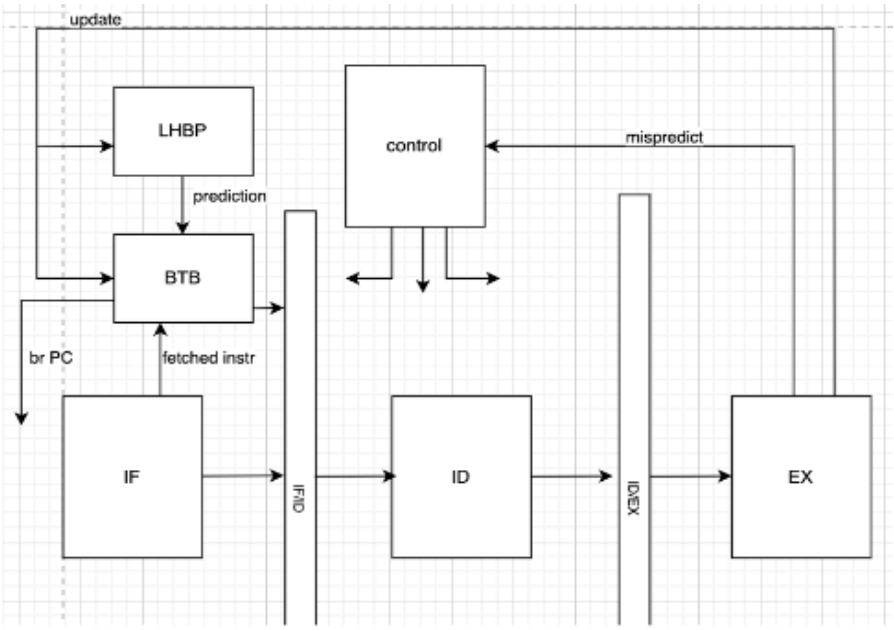


Figure 5: The general diagram for predictor and BTB units.

When an instruction is fetched from the I-cache, it is indexed into the BTB using [7:2] bits of its PC to check whether this instruction is a branch. Each entry of the BTB contains 3 parts: **pc**, **target_pc**, and a **predict** bit that indicates whether this instruction should be taken or not for this time. If it should be taken, **target_pc** will be used directly as the next pc in IF. This information, including whether this instruction was found in BTB and whether it was taken was stored in a two-bit variable and passed to the following stages through Control Words. Finally, in the EX stage, it will be compared to the result of the comparator to determine whether each branch instruction was mispredicted or not. Then an **update** signal will be generated and passed to the predictor and BTB to let them update accordingly.

For the predictor, we implemented it to be indexed by 5 bits of PC and then to generate the prediction decision based on a 2-bit state machine.

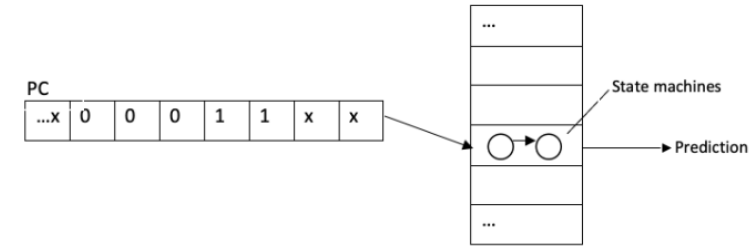


Figure 6: The predictor schema

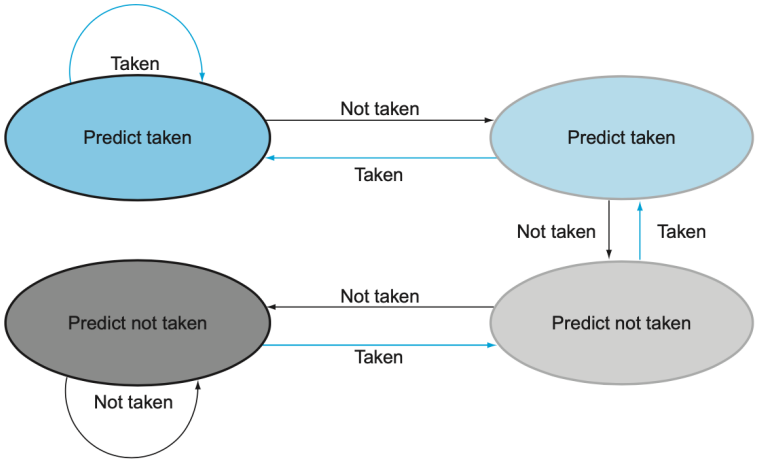


Figure 7: The states schema by Patterson & Hennessy (2014)

Every time a branch is executed in the EX stage, the state machine will be updated accordingly and send the prediction for the next time to the BTB.

I.(B) Testing

We implemented a predictor counter to count the total number of branches that happened in the program and the number of mispredictions by the predictor. Through these two data, we can calculate the accuracy of the predictor.

I.(C) Performance analysis

	comp1.s Time (ps)	comp2_i.s Time (ps)	comp3.s Time (ps)	Slack	Power	Total Cell Area
Static not taken	710585000	1700315000	960395000	2.86	7.37e+03	170369.805004
LHBP	395861480	916736920	590567080	4.16	1.04e+04	176627.188877

Table: Comparison between completion times, slack, power, and total cell area for the design with and without an LHBP.

In terms of completion time, the average speedup was $(\frac{710585000\ ps}{395861480\ ps} + \frac{1700315000\ ps}{916736920\ ps} + \frac{960395000\ ps}{590567080\ ps})/3 = \frac{1.79503+1.85475+1.62623}{3} = 1.758$. In return, there was a negative impact on the total cell area and power.

The average accuracy of the predictor, however, does not reach an ideal 80%. Analyzing the waveform of the Verdi report, we found that most of the branches do not stick to one selection (taken or not taken) for a long time. So our local history predictor, which uses a 2-bit state machine to update itself instead of

taking full advantage of different local branch histories, does not perform well in correctness.

Though the average accuracy does not reach an ideal 80%, the predictor does have a strong positive impact on the performance, so it is in the final design.

II. Hardware Prefetching

II.(A) Design

We implemented a sequential prefetching protocol between the L1 caches and the L2 unified cache. Instead of prefetching the data directly to the L1 cache, we designed an 8-entry tagged stream buffer to store the data temporarily. We designed the buffer by modifying the given cache to operate as a stream buffer based on Jouppi (1990). The state machine checked for read hits, sent the read and write signals from the L1 cache to the L2 cache, and sent a prefetch request to the L2 cache upon a read miss. The valid array was used to check if the prefetched data is still correct; the tag and data arrays stayed the same (behaved differently based on the new state machine). We used a stream buffer to prevent cache pollution by storing the data there that would have otherwise been in the cache.

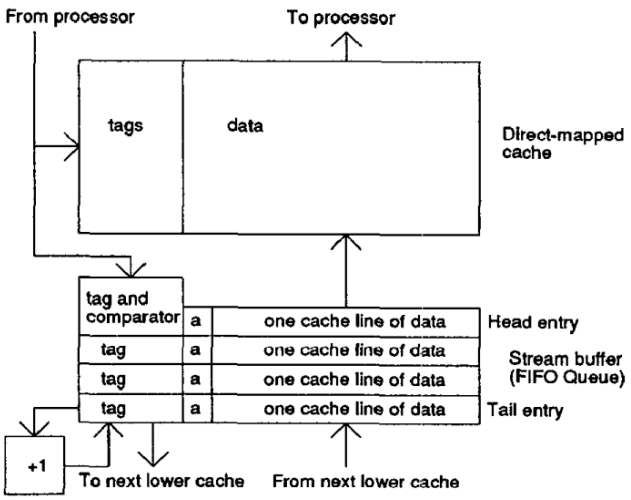


Figure 8: Sequential stream buffer design by Norman P. Jouppi (1990)

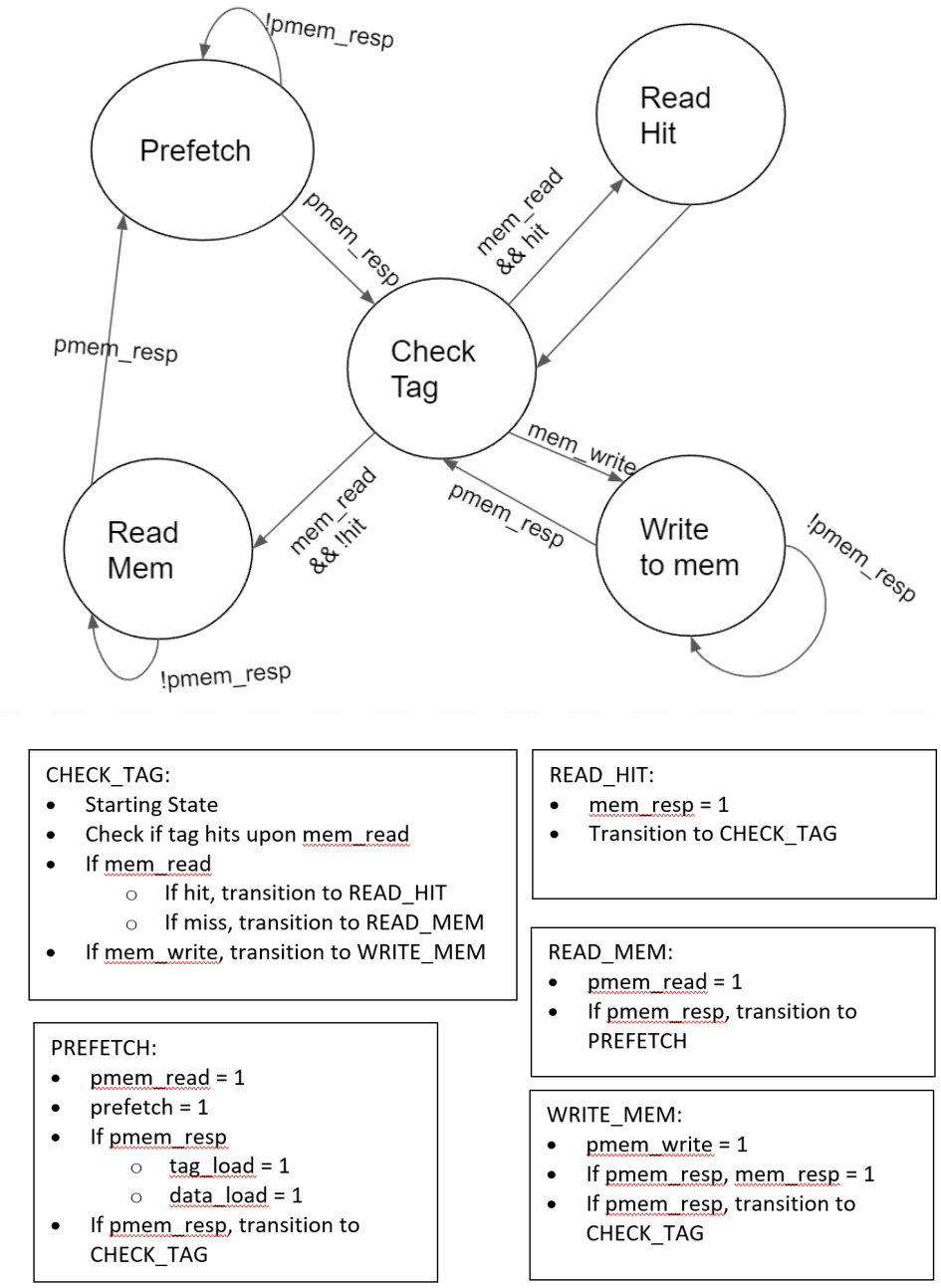


Figure 9: The stream buffer control unit.

II.(B) Testing

We used existing MP3 resources, like the Shadow Memory and the RVFI Monitor, as well as our MP4 HVL code that checks communication between caches to test hardware prefetching for correctness and performance.

II.(C) Performance analysis

Prior to the design competition, we received Shadow Memory errors when prefetching was activated.

We were only able to run prefetching on comp1.s correctly, as it caused issues on other testcodes. Since the structure of the stream buffer is very similar to the L1 cache, we expected speed-up but also increased area and power consumption.

Unfortunately, our design did not synthesize with prefetching, so we were not able to obtain detailed performance data. The speed-up based on compile time of comp1.s was **1.1%**, and cache misses were reduced from **32 to 25** due to prefetching (confirmed by the prefetching performance counter showing 7 read hits). Due to correctness issues and insignificant speedup, prefetching is not included in our final design.

```
Compilation Successful
Reset Memory
L2 CACHE MISS RATE:          32 /        3610
EVICTION WRITE BUFFER STATS:           0 read hits,           0 writebacks,          33 total accesses

$finish called from file "/home/murata3/411-mp4/mp4/mp4/hvl/top.sv", line 211.
$finish at simulation time           670555000
          V C S   S i m u l a t i o n   R e p o r t
Time: 670555000 ps
```

Performance counters **before** prefetching (comp1.s)

```
Compilation Successful
Reset Memory
L2 CACHE MISS RATE:          25 /        3603
EVICTION WRITE BUFFER STATS:           0 read hits,           0 writebacks,          26 total accesses

$finish called from file "/home/murata3/411-mp4/mp4/mp4/hvl/top.sv", line 211.
$finish at simulation time           667135000
          V C S   S i m u l a t i o n   R e p o r t
Time: 667135000 ps
```

Performance counters **after** prefetching (comp1.s)

III. L2+ Cache System

III.(A) Design

We designed an 8-way set associative cache with 8 sets. Our design re-used the MP3 cache design and modified it to parameterize the sets and ways. We chose to use an 8-way set associative L2 cache to alleviate the L1 instruction cache and data cache miss penalties. The L2 cache only becomes useful once data is evicted from the L1 cache i.e., once the L1 cache misses and the L2 cache hits. L2 caches greatly reduce the latency of memory accesses, with diminishing returns for each additional level of the memory hierarchy.

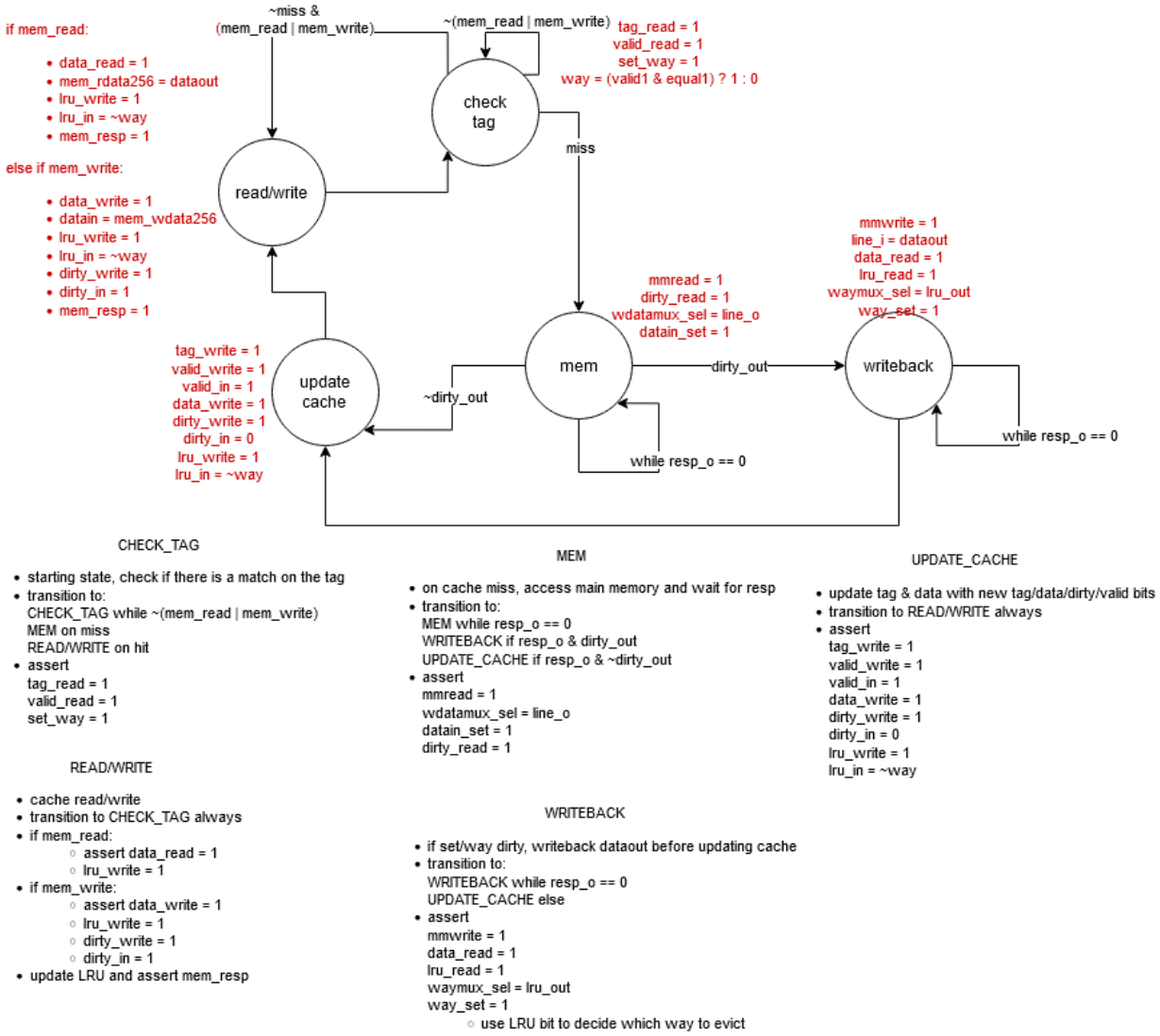


Figure 10: The MP3 cache control unit design, re-used for this project.

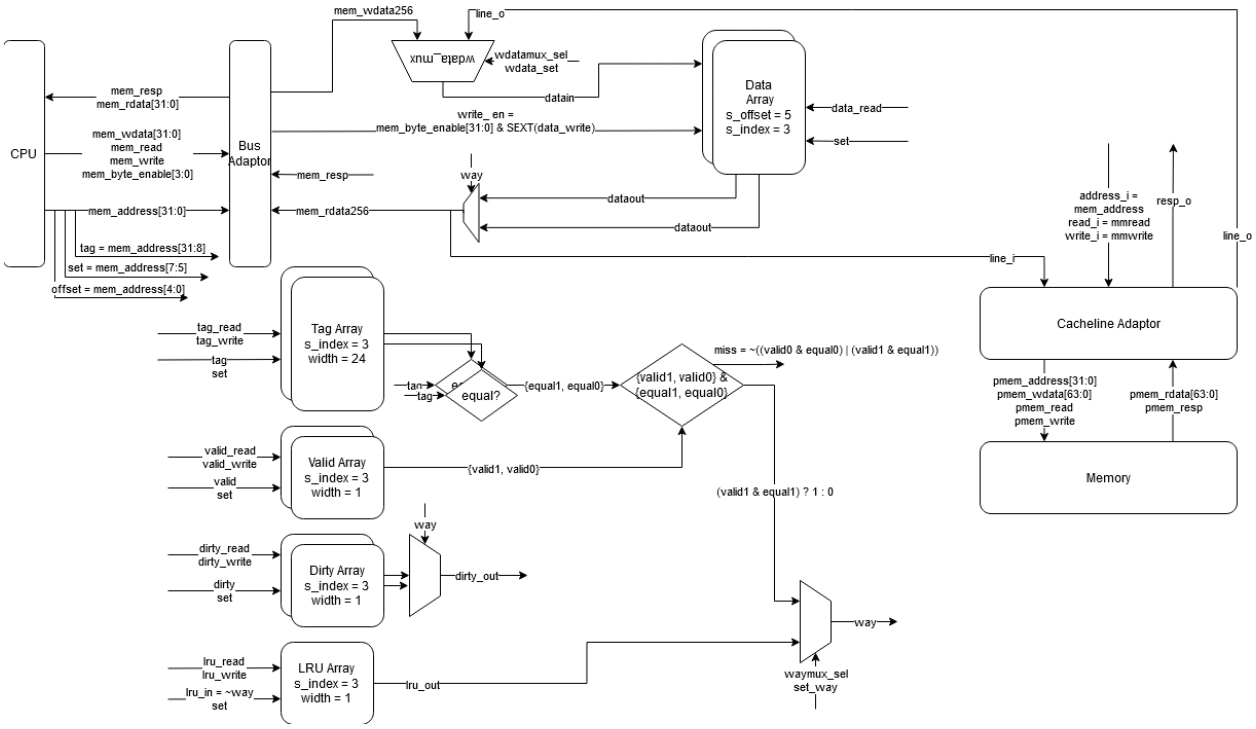


Figure 11: The MP3 cache datapath design, re-used for this project.

III.(B) Testing

We used existing MP3 resources, like the Shadow Memory and the RVFI Monitor, to test the L2 cache design with various parameters for the number of sets and ways to verify correctness. Additionally, we wrote a C script to check that the replacement policy in the cache functions correctly by filling one set and checking what order it fills and evicts entries in.

III.(C) Performance analysis

As a general rule, the larger the cache, the better the performance in terms of speed, but the trade-off is that it uses more power and area. Of course, there is a point where increasing the size of the L2 cache no longer increases the performance since it has eliminated all but the compulsory misses, but the L2 cache becomes so large that it does not seem worth the cost of power and area. Testing a system with a 4-way set associative L2 cache and an 8-way set associative L3 cache did not improve the performance.

	comp1.s Time (ps)	comp2_i.s Time (ps)	comp3.s Time (ps)	Slack	Power	Total Cell Area
No L2 Cache	1573048568	5020843880	3545448712	5.33	3.01e+03	50025.555066
L2 Cache	395861480	916736920	590567080	4.16	1.04e+04	176627.188877

Table: Comparison between completion times, slack, power, and total cell area for the design with and without an L2 cache.

In terms of completion time, the average speedup was $(\frac{1573048568\ ps}{395861480\ ps} + \frac{5020843880\ ps}{916736920\ ps} + \frac{3545448712\ ps}{590567080\ ps})/3 = \frac{3.97373487312+5.47686448583+6.00346485957}{3} = 5.15.$ In return, there was a negative impact on the fmax and a much larger negative impact on total cell area and power. The average miss rate was $(\frac{32}{3610} + \frac{65}{11077} + \frac{313}{8732})/3 = 0.0505774479345/3 = 1.69\%.$ The L2 had a strong positive impact on the performance, so it is in the final design.

IV. Eviction Write Buffer

IV.(A) Design

We designed a fully associative cache with four entries as advised by a TA for our eviction write buffer, abbreviated EVB. This buffer only caches writebacks from a higher memory level so that the higher memory level does not need to wait for the lower memory level to write then read new data. Our design re-used the MP3 cache components with a new control unit and datapath. It is useful on high-latency writebacks, so we placed it between the L2 cache and the main memory.

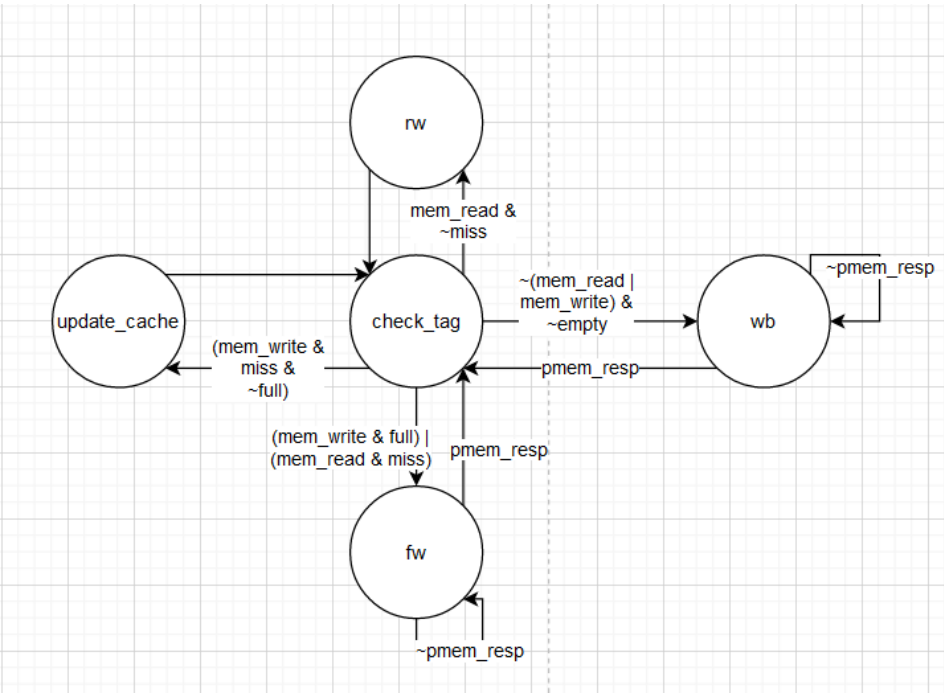


Figure 12: The EVB control unit.

IV.(B) Testing

We used existing MP3 resources, like the Shadow Memory and the RVFI Monitor, to test the eviction write buffer design to verify correctness. Additionally, we wrote a custom C script to check that the cache behaves correctly under various circumstances, such as during a read from a higher level, during a write from a higher level, as well as its behavior when full.

IV.(C) Performance analysis

Due to the behavior of this type of cache, the optimal size would depend on various design choices. Using fewer entries would've been more efficient for our EVB, but in an EVB with a counter between accesses, having more entries would be more optimal since it may receive more writeback requests

while it is waiting. The EVB worsened the performance. Testing the EVB in different locations in the memory hierarchy had the same result: either no speedup or slowdown.

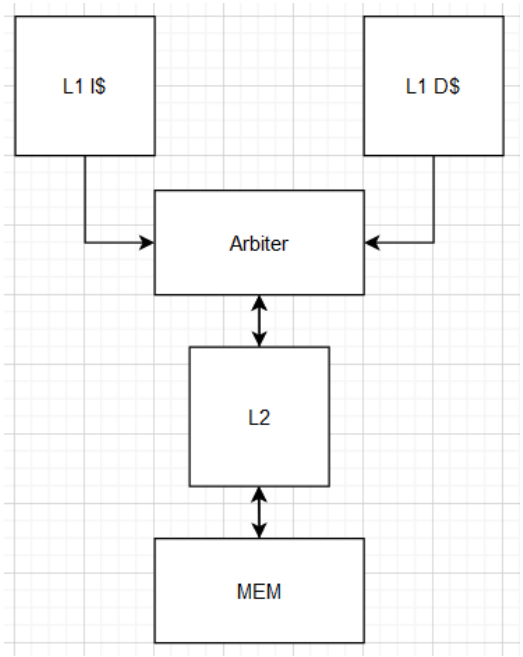
	comp1.s Time (ps)	comp2_i.s Time (ps)	comp3.s Time (ps)	Slack	Power	Total Cell Area
No EVB	406431000	941265000	604581000	4.16	1.04e+04	176627.188877
EVB	538756000	1246412000	778812000	2.14	1.03e+04	192291.396610

Table: Comparison between completion times, slack, power, and total cell area for the design with L2 and with and without an EVB.

In terms of completion time, the average speedup was $(\frac{406431000\ ps}{538756000\ ps} + \frac{941265000\ ps}{1246412000\ ps} + \frac{604581000\ ps}{778812000\ ps})/3 = 0.76$. The EVB had a negative impact on the completion times, decreased the slack, and increased the total cell area, therefore, we did not put it in the final design.

4. Additional observations (if any)

Our final memory hierarchy is as follows:



We considered a number of other hierarchies, but we found that the performance, mostly based on the completion time, either did not improve or got slightly worse. Unfortunately, we did not get the chance to test with different cacheline sizes, which we also believe was a factor in the reason that the prefetching was not as effective. Because each cacheline had 8 instructions in it, it was possible that there were multiple branches in a single cacheline.

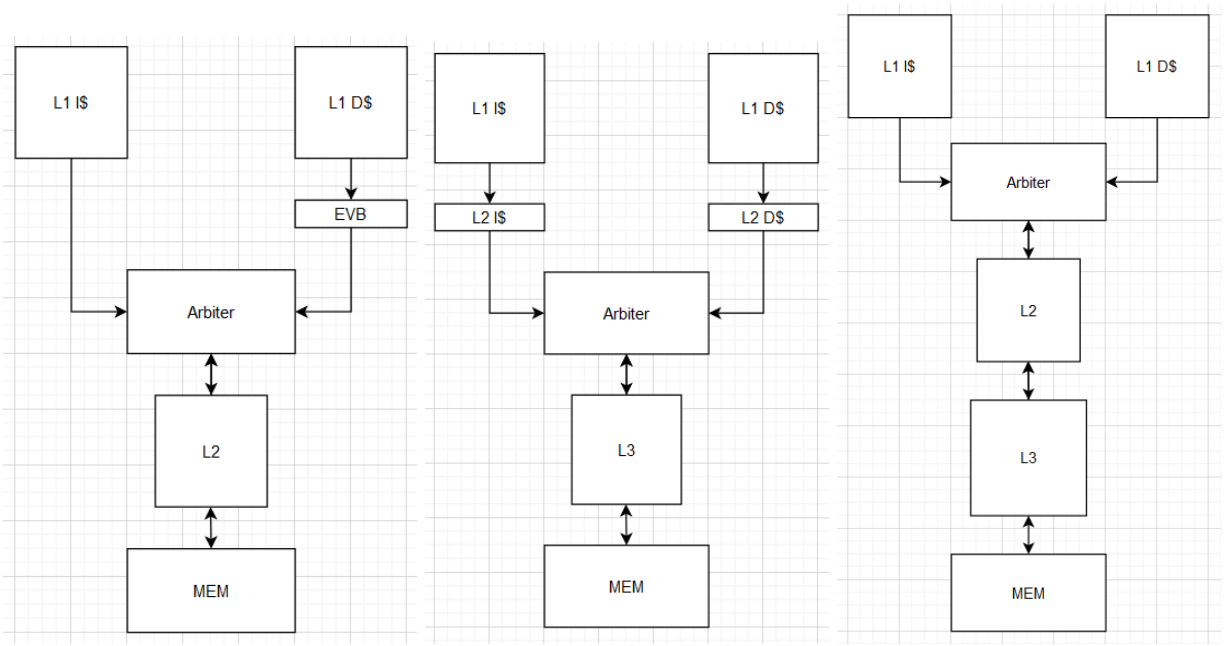


Figure 13: Considered memory hierarchies. (Left) The EVB was moved directly underneath the L1 D-cache. This did not have nearly as much effect on the slack, but the speedup was about the as above, so there was no speedup. (Middle) Smaller caches were placed under each of the L1 caches. This had very little improvement in performance. (Right) An L3 cache was placed under the L2 cache. The L2 cache was made smaller, and the L3 was made larger. This made very little difference in the performance, but it increased the area greatly.

5. Conclusion

Our main goal was to design a pipelined multiprocessor that runs the RV32I instruction set. And we achieved that goal. Even though we were not able to optimize to the extent we were aiming for; the project was an overall success. We believe what makes our processor stand out is the memory hierarchy. The L2 cache is parameterized, and we have multiple advanced features that increase cache performance. Additionally, most parts of the design can be modified without negatively affecting the rest of the processor, requiring a few to no changes to the other parts.

In the future, with the help of verification tools, the current advanced features can be heavily improved. Many other features can also be added thanks to the modularity of our design.

References

- Hennessy, J., Patterson, D. (2014). Computer Architecture: A Quantitative Approach.
- Jouppi, N. (1990). Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers.