ECE 411

Spring 2021

MP4 Report

RISC-V 5-Stage Pipelined CPU

Implementing the RV-32I ISA

Team PCB

Prateek Tenkale, Chris Shannon, Brendan Callas

Introduction

This report summarizes the work that Team PCB has done for the implementation of our RISC-V pipeline processor as part of ECE411. In this report, we will give a general overview of the project, explore the milestones along the way, analyze advanced features implemented, and evaluate our final success and performance.

Before starting on this project, we have learned about, analyzed, and implemented a non-pipelined RISC-V CPU architecture in ECE411 to understand the fundamentals of the RISC-V architecture. The concept of pipelining is the next step in gaining an understanding of modern computer architecture for CPU design, but the approach of pipelining extends far beyond CPUs to memory systems, ADCs, and much more. The work in designing, implementing, and verifying this pipelined processor design is both a culmination of our previous studies in many other classes and is the introduction to modern methods of designing, analyzing, verifying, and further engineering of computer systems.

Project Overview

The purpose of this project was to design a RISC-V 5-stage pipelined processor using the RV-32I instruction set architecture, implement our design using SystemVerilog, and of course test and verify our design to ensure correctness and efficiency. We started with a base design, and through several checkpoints added more features to improve performance and energy efficiency. The goal of this project is for self-learning and advancing in the field of computer architecture- although we have a basis for our design, most of the project included independent implementation and testing, as well as design decisions which we had to consider for tradeoffs in complexity, performance, and efficiency. It was critical to ensure correctness at every checkpoint, and developing our own verification methods was essential. Finally, the addition of advanced features is perhaps the most important aspect of this project. We as a group had to discuss which features should be added to our processor, again with the important consideration of design tradeoffs; how long it might take for designing and testing, potential improvement of each feature, and possible drawbacks all needed examination. Further, we were to aim for a frequency of 100MHz in our design, so this required some additional consideration and optimization. Finally, our processor is entered into a competition with other groups, with scoring determined performance (timing on a few competition programs) and energy efficiency of the processor.

Our group was organized in a completely online setting. We used Discord in order to talk with each other, share code and help each other through live streaming, and to communicate with our mentor TA. Although the fully online setting can be somewhat difficult, we were able to effectively communicate with each other by organizing the plan for each checkpoint at the beginning and splitting up work appropriately, helping one another when needed. Organization and communication was particularly important for implementing advanced features, and we used individual branches on Github for each advanced feature to avoid conflicts and to ensure

that only fully working code was added to the main project. As will be discussed later, we were able to complete the project with full correctness and the addition of several important advanced features which helped with performance, and although results remain to be seen at this time, we are optimistic for the competition.

Milestones

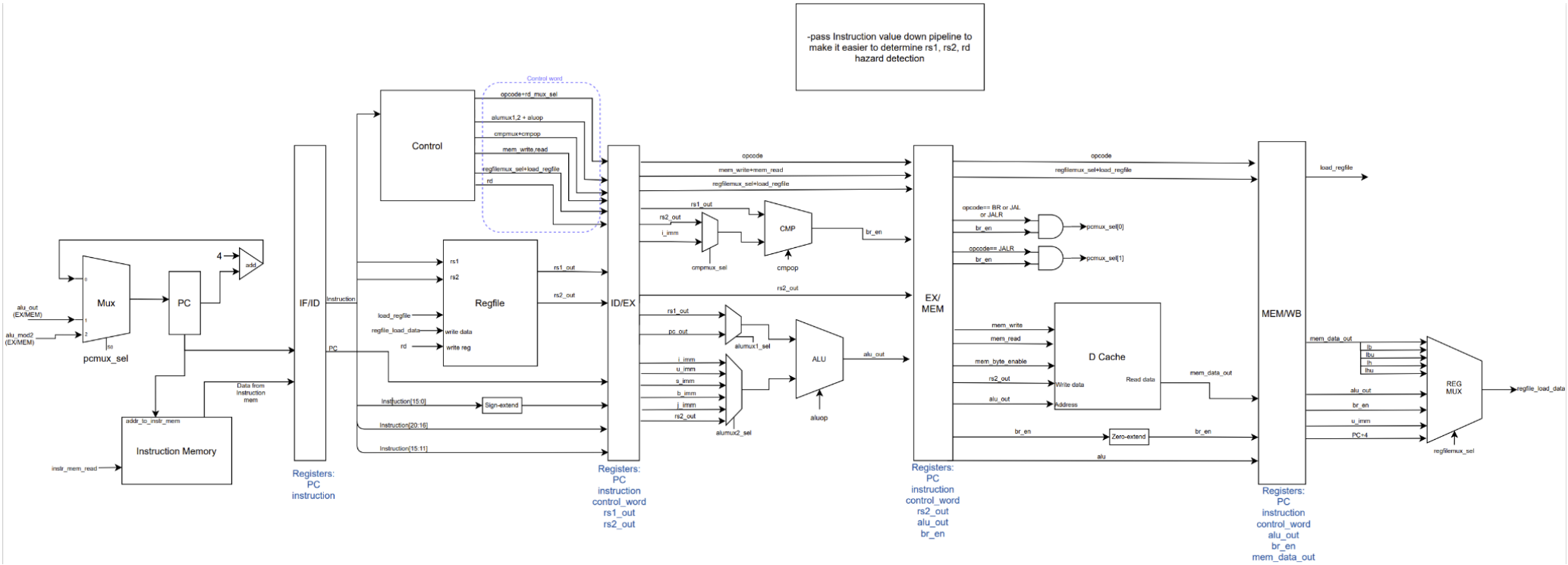
Checkpoint 1

To begin our project, we first designed the basic RISC-V pipeline processor for the RV-32I IAS (with the exception of the instructions FENCE, ECALL, EBREAK, and CSRR) that was able to run code without control or data hazards. Additionally, used a “magic memory” module instead of data cache and instruction cache, which would instantly respond to any memory request in one cycle, so that the pipeline could be tested without worrying about stalls due to memory or multiple memory requests to the main memory at the same time from data and instruction cache. We did include both caches in the initial design, but simply replaced them with magic memory during implementation.

For this initial design, we largely used diagrams and schematics given in the textbook *Computer Organization and Design* by David Patterson and John Hennessy. The textbook provided the overall description of a 5-stage pipelined processor, but not a comprehensive diagram, so we had to ensure our design was fully complete. For this part of the design, we did verification by manually tracing every instruction through the datapath to make sure every instruction worked in isolation. One part of the design worth mentioning is the Control ROM module; this part of the design was a major factor both for design and testing, as the Control ROM sets all of the required control signals that are used along each stage of the pipeline, and extra care was taken to check correctness on this module.

We then implemented our design in System Verilog, again substituting the caches for the magic memory module that was provided. Once this was complete, we did our initial testing and verification. Although some test code was provided, the code was not an exhaustive test so we again used our own code to individually test every instruction to assure correctness, at least in isolation. Our test code along with ModelSim was used for debugging purposes when we found an error, and carrying the complete instruction down each stage of the pipeline allowed easier debugging.

The initial datapath and the relevant diagrams for the Control ROM are shown below.


-pass Instruction value down pipeline to make it easier to determine rs1, rs2, rd hazard detection
Control word
Control
opcode+rd_mux_sel
alumux1,2 + aluop
cmpmux+cmpop
mem_write,read
regfilemux_sel+load_regfile
rd
Mux
PC
4
add
pcmux_sel
alu_out (EX/MEM)
alu_mod2 (EX/MEM)
IF/ID
Instruction
PC
Data from Instruction mem
addr_to_instr_mem
Instruction Memory
instr_mem_read
Regfile
rs1
rs2
load_regfile
regfile_load_data
rd
write data
write reg
rs1_out
rs2_out
Instruction[15:0]
Sign-extend
Instruction[20:16]
Instruction[15:11]
Registers: PC instruction
ID/EX
opcode
mem_write+mem_read
regfilemux_sel+load_regfile
rs1_out
rs2_out
i_imm
cmpmux_sel
CMP
cmpop
br_en
pc_out
alumux1_sel
u_imm
s_imm
b_imm
j_imm
alumux2_sel
ALU
aluop
alu_out
Registers: PC instruction control_word rs1_out rs2_out
EX/ MEM
opcode== BR or JAL
opcode== JALR
pcmux_sel[0]
pcmux_sel[1]
mem_write
mem_read
mem_byte_enable
D Cache
Write data
Address
Read data
mem_data_out
Zero-extend
alu
Registers: PC instruction control_word alu_out br_en mem_data_out
MEM/WB
load_regfile
lb
lbu
lh
lhu
REG MUX
regfile_load_data
regfilemux_sel

```
typedef struct packed {
    logic [31:0] instruction,
    rv32i_opcode opcode,

    alu_ops aluop,
    cmpop,

    alumux1_sel,
    alumux2_sel,
    regfilemux_sel,
    cmpmux_sel,
    rdmux_sel,

    logic load_regfile,

    logic data_mem_read,
    logic data_mem_write,
    logic [3:0] mem_byte_enable

} rv32i_control_word;
```

Opcode / Control Signals

	aluop	cmpop	alumux1_sel	alumux2_sel	regfilemux_sel	cmpmux_sel	load_regfile	data_mem_read	data_mem_write
op_lui	funct3	funct3	rs1_out	i_imm	u_imm	rs2_out	1	0	0
op_auipc	alu_add	funct3	pc_out	u_imm	alu_out	rs2_out	1	0	0
op_jal	alu_add	funct3	pc_out	j_imm	alu_out	rs2_out	0	0	0
op_jalr	alu_add	funct3	rs1_out	i_imm	pc_plus4	rs2_out	1	0	0
op_br	alu_add	funct3	pc_out	b_imm	alu_out	rs2_out	0	0	0
op_load	alu_add	funct3	rs1_out	i_imm	funct3*	rs2_out	1	1	0
op_store	funct3	funct3	rs1_out	s_imm	alu_out	rs2_out	0	0	1
op_imm	funct3*	funct3*	rs1_out	i_imm	alu_out	funct3*	1	0	0
op_reg	funct3*	funct3*	rs1_out	rs2_out	alu_out	rs2_out	1	0	0
op_csr	funct3	funct3	rs1_out	i_imm	alu_out	rs2_out	0	0	0

NOTE: many signals are filled in with the default value- this means some signals may be unused, but are here for completion

NOTE: some signals like funct3* means the signal is based off funct3. Often this means the default is a funct3 cast (like `cmpop = branch_funct3_t'(funct3)` and `aluop = alu_ops'(funct3)` but there are cases dependent on the funct3 which may change this value. The logic for this is shown below.

op_load

load_funct3	regfilemux_sel
lb	lb
lbu	lbu
lh	lh
lhu	lhu
lw	lw
default	lw

op_imm

funct3	aluop	cmpop	regfilemux_sel	cmpmux_sel
slt	funct3	blt	br_en	i_imm
sltu	funct3	bltu	br_en	i_imm
sr	if(funct7[5] == 1'b1) aluop = alu_sra;	funct3	alu_out	rs2_out
default	funct3	funct3	alu_out	rs2_out

op_reg

funct3	aluop	cmpop	regfilemux_sel

add	if(funct7[5] == 1'b1)
	    aluop = alu_sub;
	else
	    aluop = alu_add;
		funct3	alut_out

sr	if(funct7[5] == 1'b1)
	    aluop = alu_sra;
	else
	    aluop = alu_srl;
		funct3	alu_out

slt	funct3	blt	br_en
sltu	funct3	bltu	br_en
default	funct3	funct3	alu_out

Checkpoint 2

For Checkpoint 2, we expanded on our initial design by adding L1 caches, hazard detection, data forwarding, and static not-taken branch prediction.

L1 caches were relatively straightforward since we had designed a 2-cycle cache in the previous MP3. In order for these to be added to this processor design though, a few changes must be made. The cache implementation we had was changed to be a 1-cycle cache in order to work with the pipeline design; this was fairly simple to do, although it is worth noting that making 1-cycle caches adds a risk of a long critical path which may limit the max frequency achievable, and precludes BRAM for these caches. Additionally, since these are still caches, hazard detection is needed for a cache miss; when the cache does not respond in one cycle, the pipeline must be stalled until data is ready.

In order to have a fully working pipeline, we also added data hazard detection for when an instruction depends on a previous instruction which is still in the pipeline and has not been committed. There are several cases for forwarding with data hazards: (Note that in these descriptions, the “current stage” is the ID/EX stage registers)

1. Forwarding MEM to EX

   In this case, we check if the previous instruction (held in EX/MEM registers) is loading the regfile, and if that register being loaded is the same one as the current stage (the instruction held in ID/EX registers). If so, we must forward the data being loaded from the MEM stage to the EX stage, to the appropriate source register used.

2. Forwarding WB to EX

   Similar to the previous case, we must check if the second previous instruction is loading the regfile, and if the register being loaded is the same as one being used in the current stage. If so, we forward from the WB stage to EX.

3. Forwarding WB to MEM

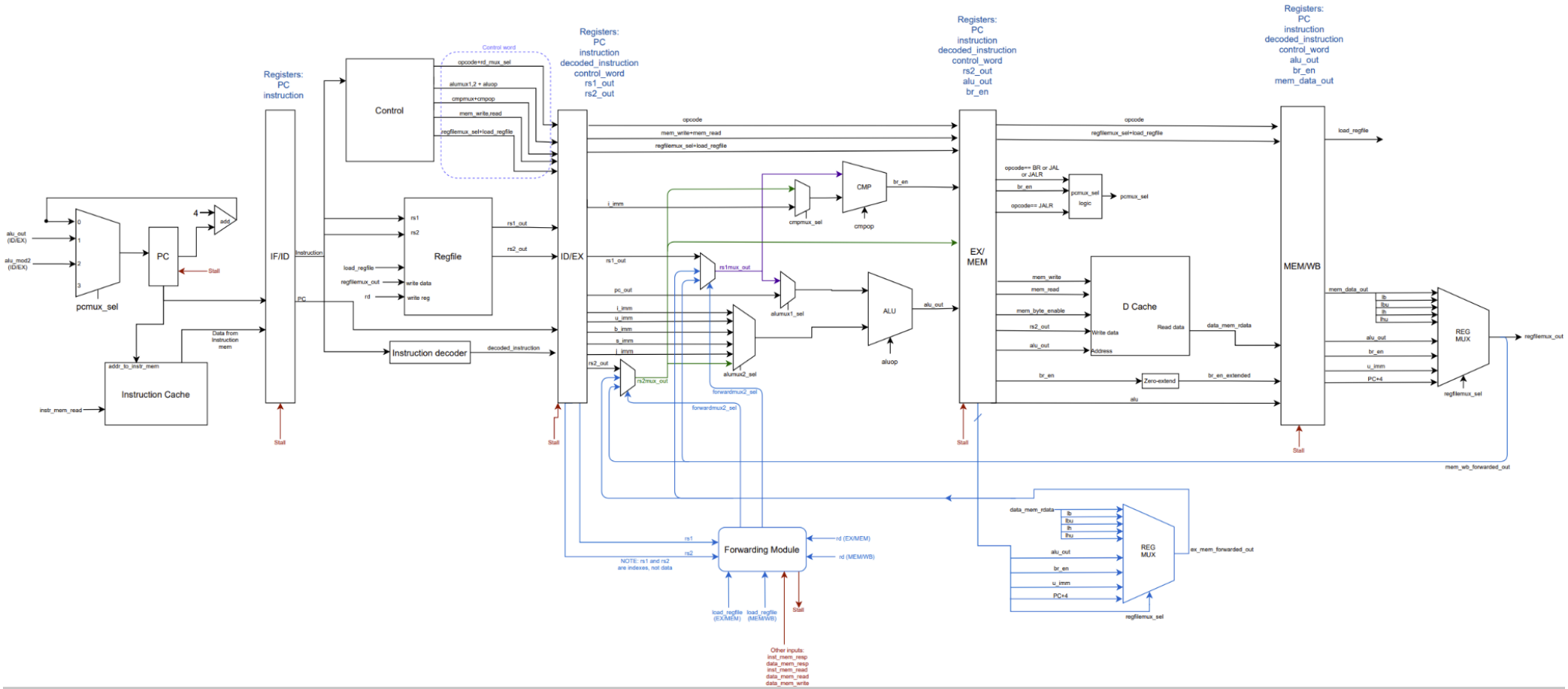
   Due to the placement of the data cache in the pipeline, there may also be a hazard if the WB stage is loading the regfile while the MEM stage is storing data into memory using the same register. In this case we must forward that register’s value from the WB stage to the MEM stage so it stores the correct value.

Also note that in all three cases here, the register being loaded must not be R0, which is hardwired to zero. If the register is R0, then no forwarding occurs.

There is one more hazard to account for, the “load-use” data hazard. When the previous instruction is a load and the current instruction uses the register that data is being loaded into, the pipeline must stall one cycle or “bubble” to avoid using incorrect data.

Finally for this checkpoint, we implemented static not-taken branch prediction. The branch prediction itself is simple, in fact the pipeline already “predicts” the branch is not taken by loading the next instruction using the PC+4 address. However, we must implement mechanics to squash instructions when the branch prediction is incorrect. This was done by resetting instruction stage registers when it is determined that a branch is taken. The control logic for hazard detection, forwarding, and stalling were all done in a single module in our design for simplicity. One can see the additions of the forwarding and hazard detection in the datapath below; the forwarding paths are color coded in blue, green, and purple for clarity.

Like the previous checkpoint, there was much testing and verification to complete. Although more test code was provided, we wrote and ran our own test code on our design to account for all possible forwarding cases, hazards, and stalls. Again using ModelSim, we were able to look at each individual instruction and make sure the correct control signals were set for forwarding, stalls, and bubbles in each test case.


Registers:
PC
instruction
Registers:
PC
instruction
decoded_instruction
control_word
rs1_out
rs2_out
Registers:
PC
instruction
decoded_instruction
control_word
rs2_out
alu_out
br_en
Registers:
PC
instruction
decoded_instruction
control_word
alu_out
br_en
mem_data_out
Control word
Control
opcode+rd_mux_sel
alumux1,2 + aluop
cmpmux+cmpop
mem_write,read
regfilemux_sel+load_regfile
alu_out
(ID/EX)
alu_mod2
(ID/EX)
0
1
2
3
pcmux_sel
PC
4
add
Stall
IF/ID
Instruction
PC
Data from
Instruction
mem
addr_to_instr_mem
Instruction Cache
instr_mem_read
Stall
load_regfile
regfilemux_out
rd
rs1
rs2
write data
write reg
Regfile
rs1_out
rs2_out
Instruction decoder
decoded_instruction
ID/EX
opcode
mem_write+mem_read
regfilemux_sel+load_regfile
i_imm
rs1_out
pc_out
i_imm
u_imm
b_imm
s_imm
j_imm
rs2_out
rs1mux_out
rs2mux_out
alumux1_sel
alumux2_sel
ALU
aluop
alu_out
forwardmux2_sel
forwardmux2_sel
Stall
CMP
cmpmux_sel
cmpop
br_en
EX/
MEM
opcode
regfilemux_sel+load_regfile
opcode== BR or JAL
or JALR
br_en
opcode== JALR
pcmux_sel
logic
pcmux_sel
mem_write
mem_read
mem_byte_enable
rs2_out
alu_out
D Cache
Write data
Address
Read data
data_mem_rdata
br_en
Zero-extend
br_en_extended
alu
Stall
MEM/WB
load_regfile
mem_data_out
lb
lbu
lh
lhu
alu_out
br_en
u_imm
PC+4
REG
MUX
regfilemux_sel
regfilemux_out
mem_wb_forwarded_out
Stall
Forwarding Module
rs1
rs2
NOTE: rs1 and rs2
are indexes, not data
rd (EX/MEM)
rd (MEM/WB)
load_regfile
(EX/MEM)
load_regfile
(MEM/WB)
Stall
Other inputs:
inst_mem_resp
data_mem_resp
inst_mem_read
data_mem_read
data_mem_write
data_mem_rdata
lb
lbu
lh
lhu
alu_out
br_en
u_imm
PC+4
REG
MUX
regfilemux_sel
ex_mem_forwarded_out

Checkpoint 3

This checkpoint is where our group decided on and implemented advanced features to create an individual design compared to other groups. We discussed the tradeoffs of many advanced features within our group and with our mentor TA to consider the features that could add the greatest benefits, while still considering the complexity, time effort, and potential drawbacks of each feature. We decided to implement an 8-way L2 cache, Eviction Write Buffer, OBL Prefetcher, and the RISC-V M-extension to support multiplication and division instructions for our advanced features. Testing for each feature was done in a similar manner to previous checkpoints; we were able to compare the data flow and final outputs of each competition program between our pipelined processor and our non-pipelined MP3 processor to ensure correctness, as well as writing individual tests for each advanced feature to test their complete functionalities. This report will go into more detail about each of these features in the Advanced Features section.

Checkpoint 4

The goal of the final checkpoint was to ready our processor design for competition. Competition takes into account three provided programs and uses the total time for each program as well as the power usage of the design for score. We analyzed each of our advanced features to check that each one was helping us for the competition, and decided to remove the OBL prefetcher since it increased times slightly for competition.

Additionally, we made several optimizations to our existing design. One advancement was implementing BRAM in the 8-way L2 cache since that cache was 2-cycle. Using BRAM instead of registers for this much larger cache allowed for better max frequency in our design and lower power usage. By this point in the design, using such a large L2 cache with registers also reached the limit of the FPGA's resources, so using BRAM allowed us to remove this flaw and not worry about its inability to compile with small additions. BRAM also allows for much larger caches, and it was further helpful as we considered increasing the L2 cache size to 16-way, but in the end this was not realized due to time limitations for testing.

One more advancement was improvement on our design's achievable frequency, or Fmax. With the additions of the advanced features, particularly the L2 cache, our Fmax had decreased significantly to around 70MHz. We used Quartus's built in analytics to find critical paths in our design and remove inefficiencies to improve Fmax. For example, we noticed one long critical path in our forwarding and realized that the data coming from the L1 cache was being unnecessarily forwarded in the MEM stage; removing this, along with several other changes, brought our FMax up to 96MHz. This was not perfect with an ideal target of 100MHz, but was very close nonetheless. This report will go into further detail of final performance in the conclusion.

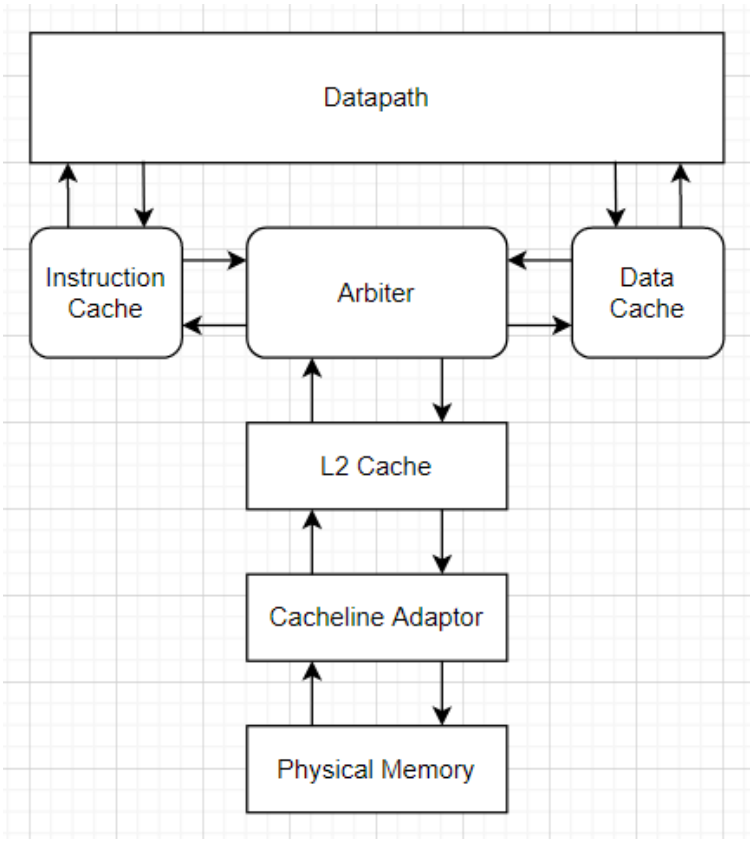
(Note: Our final competition code is on the Github's "comp" branch)

Advanced Design Features

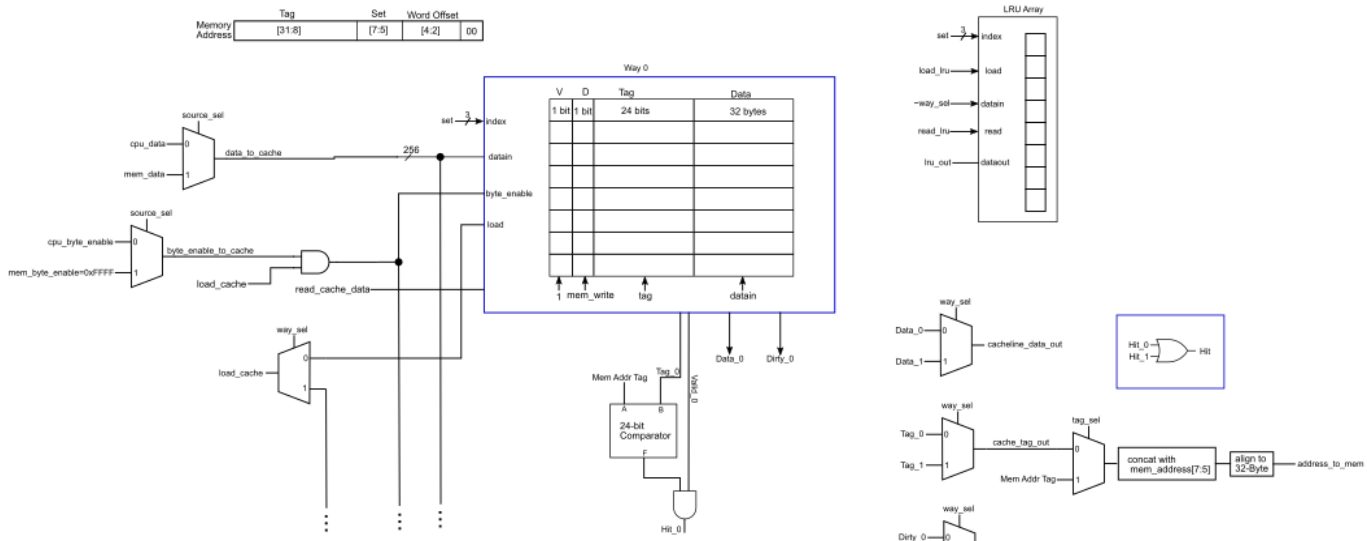
L2 Cache

One of the features we decided to implement was an 8-way set associative L2 cache. We predicted that a large L2 cache would have a great performance increase on all competition programs, since the L1 caches are relatively small and there were many memory operations in all three competition programs. Memory accesses also take significantly longer than any other instruction, so creating a large cache to vastly reduce the overall time of memory accesses would ideally have a huge benefit for our processor. We also had experience on caches already from the MP3 design, so we predicted this cache would be straightforward to implement. 8-way set associative was chosen to increase the size of the cache as much as possible without running into issues of the FPGA's resource limits, and frequency and timing constraints.

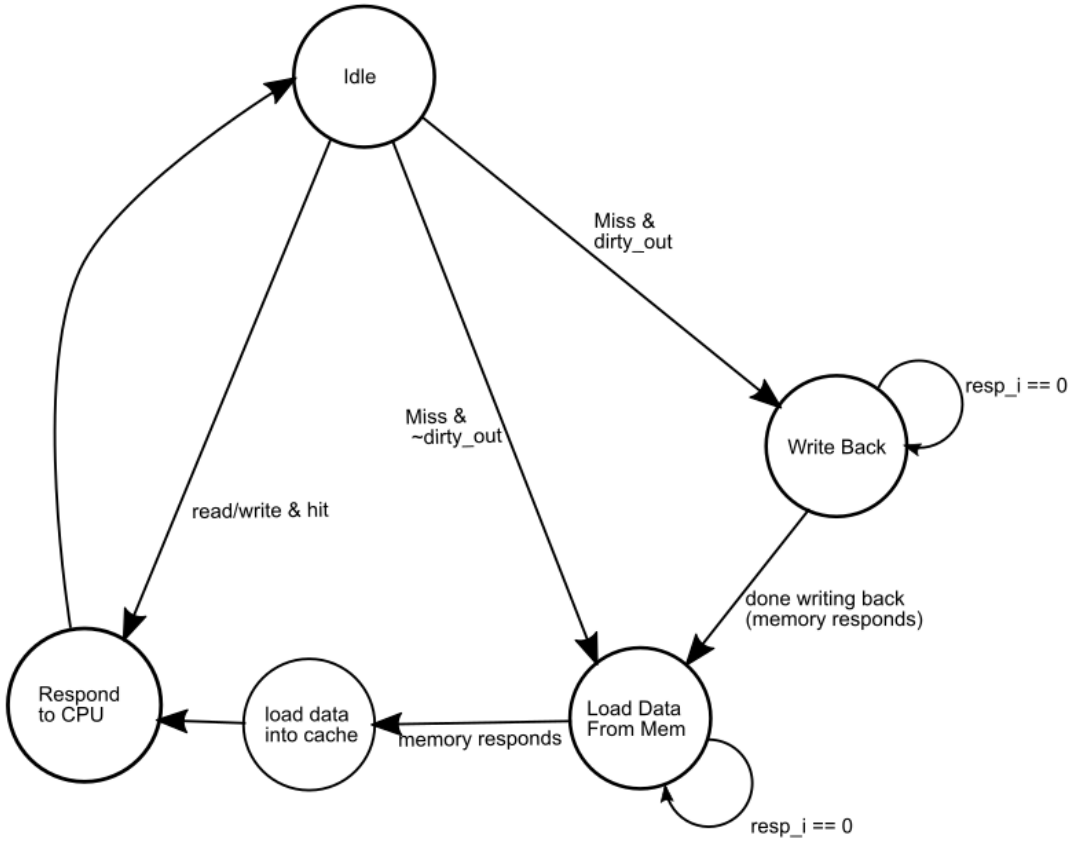
The L2 cache is a shared cache between the L1 data and instruction caches and main memory. A high-level diagram of the L2 cache placement is shown below.



As mentioned previously, the L2 cache is based on the 2-way set associative cache from MP3, but scaled to be 8-way and is maintained a 2-cycle cache. The datapath and state machine of the 8-way cache is shown below.


Memory Address
Tag
[31:8]
Set
[7:5]
Word Offset
[4:2]
00
Way 0
V
D
Tag
Data
1 bit
1 bit
24 bits
32 bytes
source_sel
cpu_data
mem_data
data_to_cache
256
set
index
datain
byte_enable
load
cpu_byte_enable
mem_byte_enable=0xFFFF
byte_enable_to_cache
load_cache
read_cache_data
mem_write
tag
datain
way_sel
Data_0
Dirty_0
Mem Addr Tag
Tag_0
24-bit Comparator
Hit_0
LRU Array
load_lru
way_sel
read_lru
lru_out
datain
read
dataout
cacheline_data_out
Hit_1
Hit
cache_tag_out
tag_sel
concat with mem_address[7:5]
align to 32-Byte
address_to_mem
Dirty_1
dirty_out

x8


Idle
Miss &
dirty_out
Miss &
~dirty_out
read/write & hit
Write Back
resp_i == 0
done writing back
(memory responds)
Respond
to CPU
load data
into cache
memory responds
Load Data
From Mem
resp_i == 0

Since the cache is extended to 8-way, a new scheme for evictions is required. For this L2 cache, we implemented a pseudo-LRU algorithm to quickly and efficiently handle evictions when the cache must read in new data and send changed data back to main memory. The pLRU method we chose is a bit-pLRU algorithm; this was chosen due to efficiency in implementation as well as the ease of scaling if we wanted to increase the cache size even further to be 16-way. The bit-pLRU algorithm is described here:

1. One bit is stored as a status bit for each cache line, called MRU-bits
2. Each access to a cache line sets its MRU-bit to a 1 to indicate it was recently used
3. When the final bit is set to a 1, all other bits are reset to a 0
4. On a cache miss, the cache line with the leftmost 0-bit is used to be replaced

The 8-way L2 cache was a success, greatly helping our CPU by significantly decreasing times for all three competition programs. The provided comp1 time decreased by 10% while comp2 and comp3 times decreased by 50%. Below is a table showing the L2 cache's performance when it was first implemented:

	L2 Hits	L2 Misses	L2 Writebacks
comp1	1	35	0
comp2_i	4772	1131	15
comp3	5341	1841	1

It is clear to see the great benefit from the L2 cache here. Averaging the total hits across all competition programs yields a 77% hit rate, with much less than one percent of the data that is written to cache being evicted. From this data (note the very few writebacks) we can also see that almost all misses in the L2 cache are compulsory misses, which are due to the empty cache at the beginning of the program so some misses will always be present. The very few writebacks also proves the effectiveness of the cache at holding all of the data necessary for the programs; this size appears to be optimal for the given programs, and increasing to 16-way would yield very diminishing returns at the expense of power and Fmax. By the end of this project it was clear to the team that the L2 cache had the greatest performance benefit of all the features, and this showed us the extreme importance of caches and other memory technology improvements for modern computing.

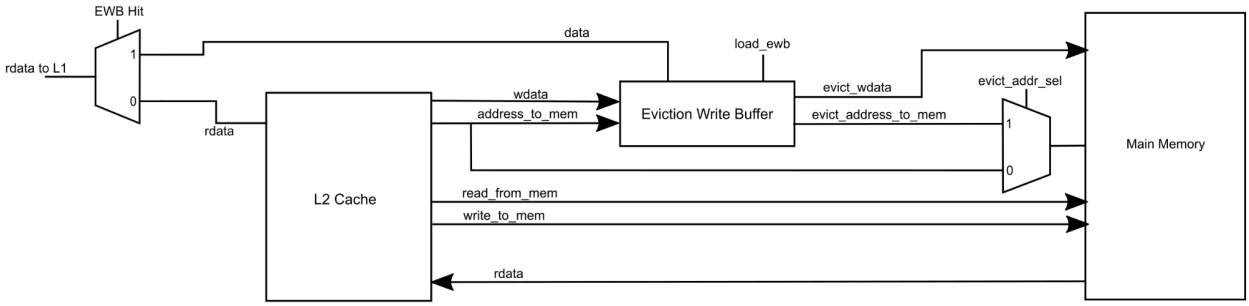
Of course, engineering is all about trade offs. With the addition of a relatively large cache, we had a significant decrease in Fmax, from about 100MHz to 70MHz. The cache originally used registers like our L1 caches, so we suspect the use of many registers had an impact on the achievable frequency of our design. We also ran into issues where the compilation of the design approached the limits of the FPGA's resources, causing difficulties when adding additional features afterwards. We also suspect the addition of this large cache greatly increased our power usage above the target baseline, as by the end we had approximately 800 mW power usage compared to the 400 mW targets. There also may be instances where the cache does

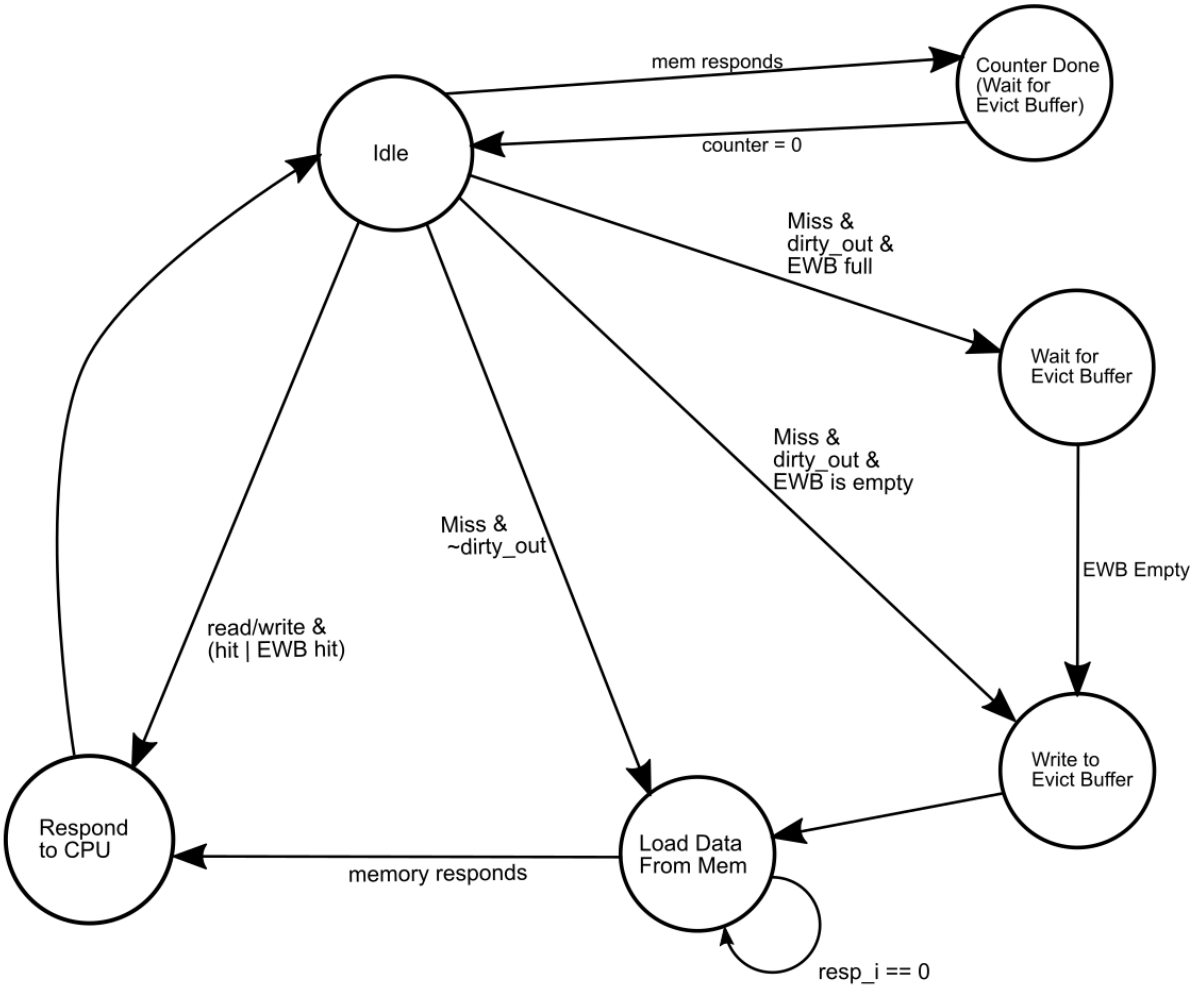
not provide a very great benefit. As one can see above, it does not provide a very large benefit for the comp1 program, presumably due to the low number of overall memory accesses. Any program which has very few memory accesses will not benefit much from the Large L2 cache, and the negative impacts may even outweigh its benefits for some cases. Further, even though the goal of this cache was to provide a large cache, if a program had many different memory accesses that all mapped to one set, or many accesses with very low locality, it is possible that there could be few to no cache hits and the additional overhead of the cache could have a slight negative impact. However, these cases are rare and the overhead of the cache is relatively small compared to the large benefit that is seen in the vast majority of programs as we see with the three competition programs above. Other programs with high spatial or temporal locality will highly benefit from the L2 cache, particularly considering the smaller L1 caches do not have the necessary capacity for even moderately sized programs.

Overall, as the above data shows, the L2 cache was a very beneficial addition to our processor and exemplifies the importance of faster memory improvements and technology in the larger scope of computer architecture.

Eviction Write Buffer

To expand on the cache system, we also added an eviction write buffer, or EWB, to our CPU design. The purpose of the EWB is to hide memory latency when the cache needs to writeback; under normal circumstances, if there is a read request with a full cache and a writeback needs to occur, the CPU must wait for the entire writeback plus the read latency, which can result in an extra long delay. In order to shorten this, the EWB is connected to the cache so the cache can write back to the EWB (which only takes one cycle) then read from memory, reducing latency in these cases. Further, if the evicted data is requested at a later time ,and if the data is still in the EWB, it essentially acts as another cache line. When the EWB is full and another writeback occurs, the EWB must write back to main memory. It also writes back to main memory after a set time under the assumption that the evicted data will not be used, so it can become ready for the next writeback. We decided to add the EWB to the L2 cache since the latency to write back to main memory from L2 is much greater than the time for L1 to write to L2, so it will hide more latency. The diagram and control flow of the eviction write buffer are shown below. Note that the control flow is combined with the state machine for the L2 cache.





Unfortunately, the addition of the eviction write buffer did not produce very impressive results. Although it was effective at hiding write back latency, the total performance benefit is underwhelming. Below are several performance metrics to express the effectiveness of the EWB. In this table, “hits” describes the number of times that there was a memory request and the data was present in the EWB to respond to the CPU with.

	EWB hits	EWB Misses	EWB Writebacks	Cycles Saved
comp1	1	35	0	0
comp2_i	0	5903	15	705
comp3	0	7182	1	57

From the above data one can see that while the EWB does technically improve performance, the increase in performance is extremely minimal. Even in the best case with 15 writebacks (the EWB is hiding latency each time) and 705 cycles saved- which translates to 7050ns saved- this

is quite minimal compared to the total runtime of approximately 1.86ms for comp2, less than one percent of the total runtime. This disappointing performance boost can be attributed to a few factors. The main reason is the large L2 cache described above. As one can recall from the data, there are only 16 writebacks total across all three competition programs. Since the EWB is only useful in the case of a writeback, the very low numbers of writebacks caused by the L2 cache means the EWB is used very infrequently and has a limited ability to perform in a meaningful way. This is also most likely due to the specific programs run on the processor. The programs have few enough memory accesses with good enough locality to be almost entirely optimized by the L2 cache. Given different programs with many more memory accesses and lower locality, we would expect to see many more writebacks and thus much more utilization of the EWB. Likewise, certain programs with more memory accesses would be more likely to have hits with the EWB as the L2 cache fills and needs to write back more frequently.

Fortunately, the eviction write buffer did not hurt the performance of any competition program. However, in hindsight it would have been much more beneficial for the team to focus our time and energy on other advanced features which could have given a greater performance advantage.

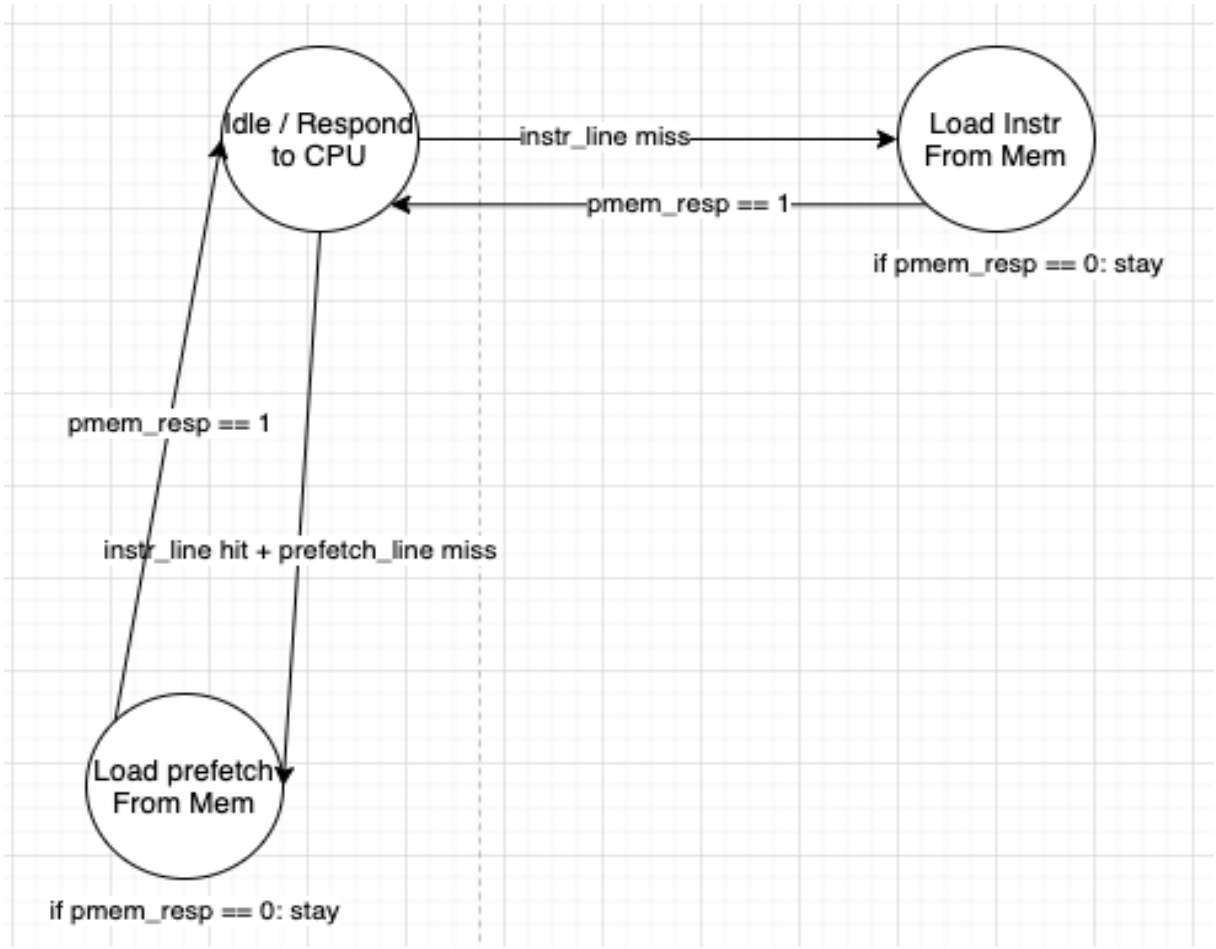
OBL Prefetching

In addition to an L2 Cache and Eviction Write Buffer, we also implemented One Block Lookahead (OBL) Prefetching. The idea behind prefetching is to have data available in the cache before it is requested in order to increase the hit rates for the cache. For OBL Prefetching specifically, you would like to initiate a memory access for cache line i+1 whenever line i is just referenced or accessed and results in a cache miss. We felt that implementing OBL Prefetching on the Instruction Cache would have the best performance benefit to our CPU because typically, the overall order of execution for the instructions is linear, and most of the instructions will be accessed at some point. We designed it so that the i+1 block was prefetched regardless of whether the i block resulted in a hit or miss because we believe there is a very high probability that instructions from the i+1 will be requested later in the program.

In order to get the best performance benefit out of prefetching, we needed to make sure that the prefetch request from memory was not stalling our entire pipeline. Our original thought was to have an i+1 cache line physical memory request in addition to the i cache line memory request that would load two blocks of memory at a time upon a cache miss. However, we quickly realized that this was not the most effective solution, as stalling the entire pipeline to fulfill the i+1 memory access (which may not even be accessed in the future) can only result in a performance delay. Instead, we changed it so that it stalls the entire pipeline when loading the i cache line from memory (as expected), but keeps the pipeline going while fetching the i+1 block from physical memory.

We created a new cache with re-written SystemVerilog modules to check whether the i block and i+1 block are cache hits in parallel. We also added a “prefetch buffer”, which is a register to store prefetch requests that need to be fetched from physical memory.

Below is a state diagram:



Below is a bullet-point summary of how OBL Prefetching works. ‘mem_address’ represents the input address that is being requested by the datapath (the address in PC), and ‘prefetch_address’ is the address stored in the prefetch buffer. Our Instruction cache starts out in the “Idle / Respond to CPU” state

During the “Idle / Respond to CPU” state:

- Check if both the i line (cache line that corresponds to mem_address) and i+1 line (cache line that corresponds to mem_address+32) result in cache hits.
- If i line is a miss:
    - Transition to “Load Instr From Mem”
- If i line is a hit and i+1 line is a miss:
    - Respond to CPU with i line request

        - Load i+1 line into Prefetch Buffer
        - Transition to “Load Prefetch from Mem”
- If both i line and i+1 lines are hits:
    - Respond to CPU with i line request
    - Stay in same state

During the “Load Instr From Mem” state:

- Set the “Physical Memory Read Request” signal on, and pass mem_address[31:5] as the address to physical memory
- Pipeline is stalled (including pc)
- If physical memory has not yet responded:
    - Stay in the same state
- If physical memory has responded:
    - Load data from physical memory into cache
    - Transition to “Idle / Respond to CPU” state

During the “Load prefetch From Mem”

- Set the “Physical Memory Read Request” signal on, and pass prefetch_address[31:5] as the address to physical memory. ‘prefetch_address’ is the address stored in the Prefetch buffer
- If physical memory has not yet responded:
    - If mem_address results in cache hit:
        - Respond to CPU
    - If mem_address results in cache miss:
        - Do nothing, wait for prefetch request to complete
        - Pipeline is stalled
    - Stay in same state
- If physical memory has responded:
    - Load data from physical memory into cache
    - Transition to “Idle / Respond to CPU” state

We implemented OBL Prefetching after the L2 cache and EWB were completed, so all of our performance metrics for OBL Prefetching were measured with L2 cache and EWB in use.

The hit rates across competition for our Instruction cache before/after we implemented OBL Prefetching was as follows:

	Instruction Cache Hit rate w/o OBL prefetching	Instruction Cache Hit rate w OBL prefetching	Difference
comp1.s	99.95%	99.58%	-0.37%
comp2_i.s	96.26%	98.16%	+1.9%
comp3.s	88.28%	98.27%	+9.99%

Even though in comp1.s the hit rate slightly decreased, there is a small improvement for comp2_i.s and a large improvement for comp3.s.

When looking at what portion of the cache hits were prefetched, this is what we found:

	Instruction Cache Hit rate w/ OBL prefetching	% of cache hits that were prefetched
comp1.s	99.58%	80.80%
comp2_i.s	98.16%	70.09%
comp3.s	98.27%	81.05%

While the % cache hits that were prefetched may seem unusually high, this actually makes sense considering that the program order of execution generally increases. As each instruction is executing, the next cache block is being fetched from physical memory, and once all the instructions in the current cache line are done, that next block of instructions is available in the cache. When the next cache block starts executing, the block after the next block starts getting fetched from physical memory, and the cycle continues. If the order of execution strictly increased, this would (in theory) result in a close-to-100 '% of cache hits that were prefetched'. However, because of branches and other jump instructions, the '% of cache hits that were prefetched' is a bit lower than that.

Despite the improvements in cache hit rate with OBL Prefetching, the overall execution time of each competition code actually increased quite a bit.

	Execution time w/o OBL Prefetching	Execution time w/ OBL Prefetching	% slowdown (relative to time w/o prefetching)
comp1.s	656146 ns	912935 ns	39.14%
comp2_i.s	2363225 ns	2648685 ns	12.08%
comp3.s	1867665 ns	2454495 ns	31.42%

There is a small slowdown in comp2_i.s and a significant slowdown in comp1.s and comp3.s. We think there are two potential reasons why the OBL Prefetcher is slowing down our execution times:

1. Data requests are being stalled while a prefetch request from physical memory is happening, and the entire pipeline is held up for more cycles than necessary
2. Instruction memory is taking up too much space on the L2 Cache and data requests are on average taking longer

With regard to the first potential reason, our cache arbiter is designed to prioritize data requests with instruction requests. If there is another data request happening while it is awaiting an instruction request, it will keep that data request on hold until the current instruction request is finished. Our reasoning is that since it does not know the difference between a regular instruction request vs. an instruction prefetch request, the cache arbiter is keeping the data request on hold instead of overriding the prefetch request with the data request

We decided to count exactly how many cycles are being wasted for this reason. Here are our findings:

	# of cycles where data request gets stalled bc of prefetch	Execution time conversion	% slowdown (relative to execution time without prefetching)
comp1.s	2040	20400 ns	3.11%
comp2_i.s	4657	46570 ns	1.97%
comp3.s	28368	283680 ns	15.19%

While there appears to be a decent amount of slowdown for comp3.s, the % slowdown for comp1.s and comp2_i.s is minimal. None of these values are close to the % slowdown that was measured in the overall execution time.

With regard to the second potential reason, the way we measured this was by comparing the average miss cycle time of the Data cache without OBL Prefetching vs. with OBL Prefetching. If there really is less space in the L2 Cache being allocated for data requests, it would show in a higher average cycle time for Data cache misses, as there would be a lower number of L2 Cache hits for data requests. Here are our findings:

	Data Cache hit rate	Avg # of cycles for Data Cache Miss without prefetching	Avg # of cycles for Data Cache Miss with prefetching	% diff (relative to avg without prefetching)
comp1.s	99.75%	295 / 7 = 42.14	455 / 7 = 65	54.25%
comp2_i.s	99.14%	2857 / 38 = 75.18	3112 / 38 = 81.89	8.93%

comp3.s	97.46%	16890 / 283 = 59.68	21136 / 283 = 74.69	25.15%

We can clearly see that the average miss cycle time increases across all three competition codes, which supports our hypothesis. However, even then, the hit rates for our Data cache are still extremely high, so the amount of cycles that are actually wasted from Data Cache misses is relatively low.

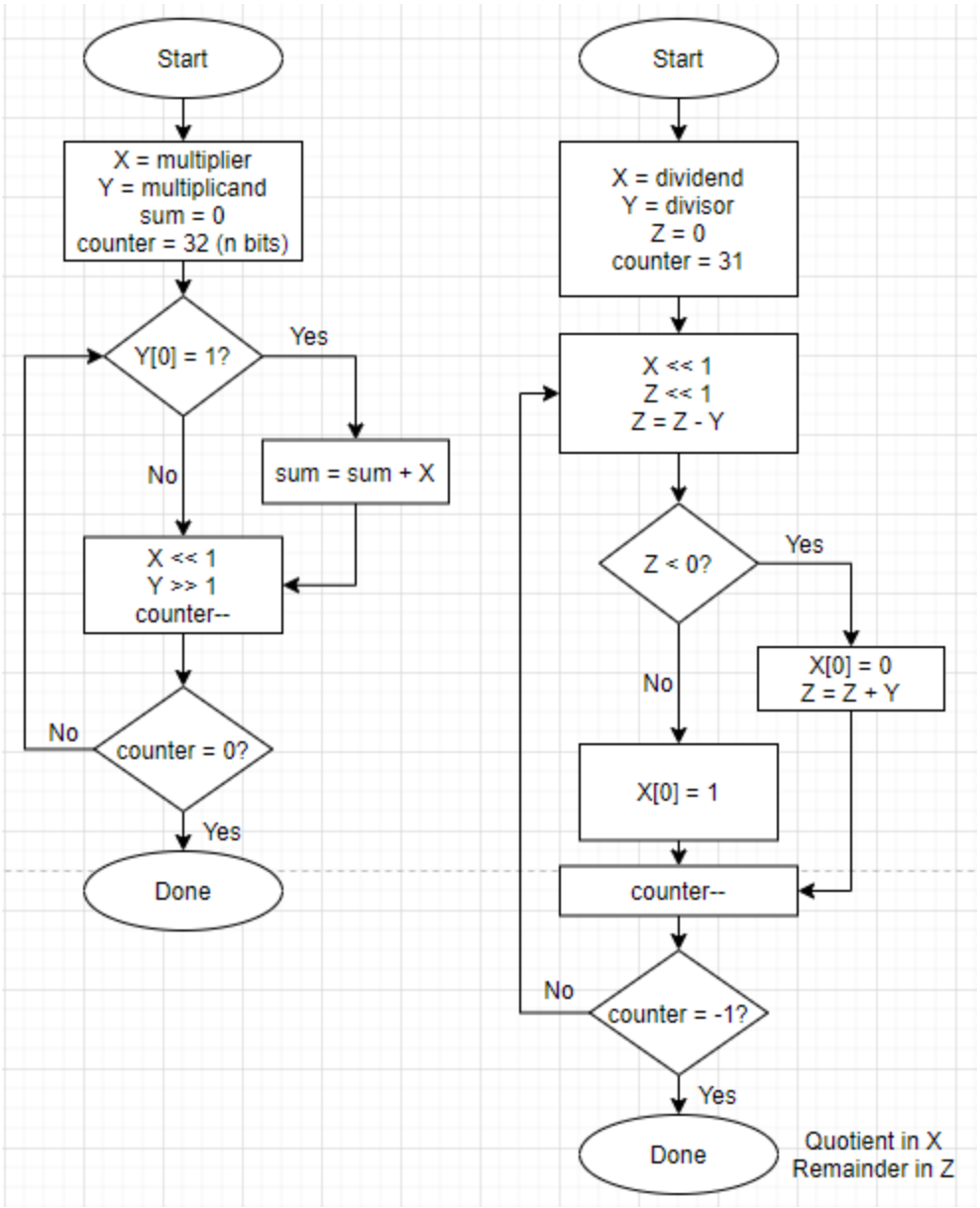
Even though we measured some slowdown from the OBL Prefetcher from our findings, it is still puzzling that OBL Prefetching increases the overall execution time by that much. We suspect these reasons for the slowdown, but there may be other factors we are unaware of. There also could be additional cycles wasted because instruction misses get held up for a prefetch request, but the high instruction cache hit rates would suggest that any slowdown due to this reason is minimal.

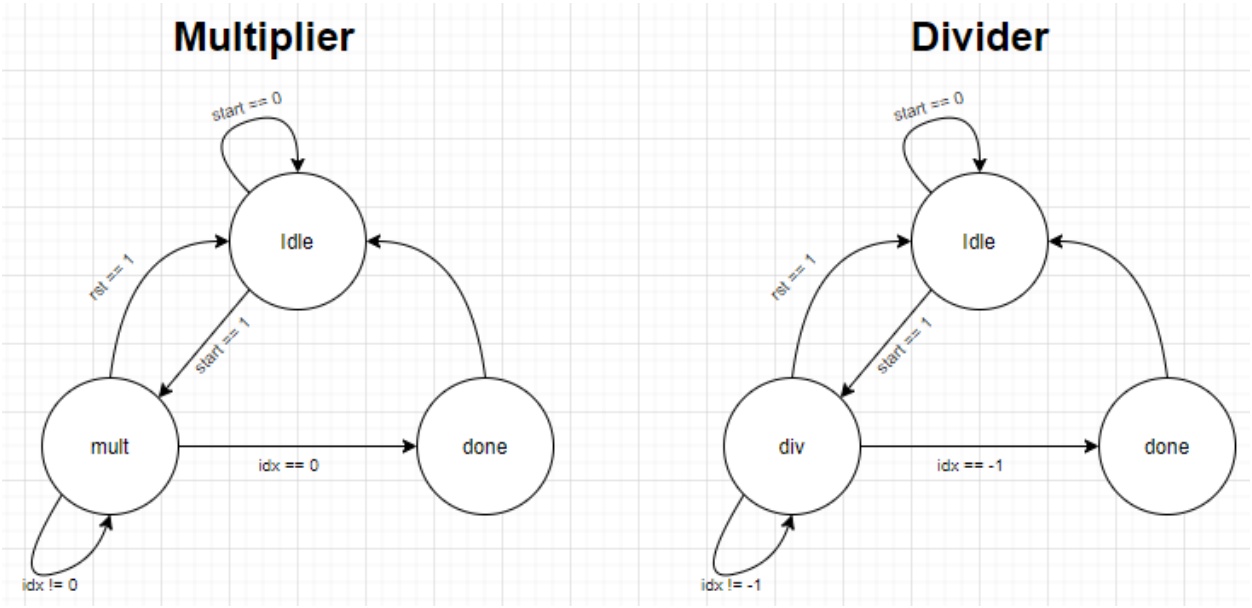
We were very surprised to see that the improvements in the Instruction cache hit rates still were not enough to overcome the negative factors in our Prefetcher. Based on the way we implemented OBL Prefetching, we believe that the best performance improvement would be seen in a program that has minimal data requests and follows a strictly increasing order of execution with minimal branches/jumps (like some sort of mathematical computation). Our implementation of OBL Prefetching would likely not have any performance improvement (perhaps even a performance decrease) if there are a significant amount of branches and data requests in the code, like there was with the competition code.

M-Extension

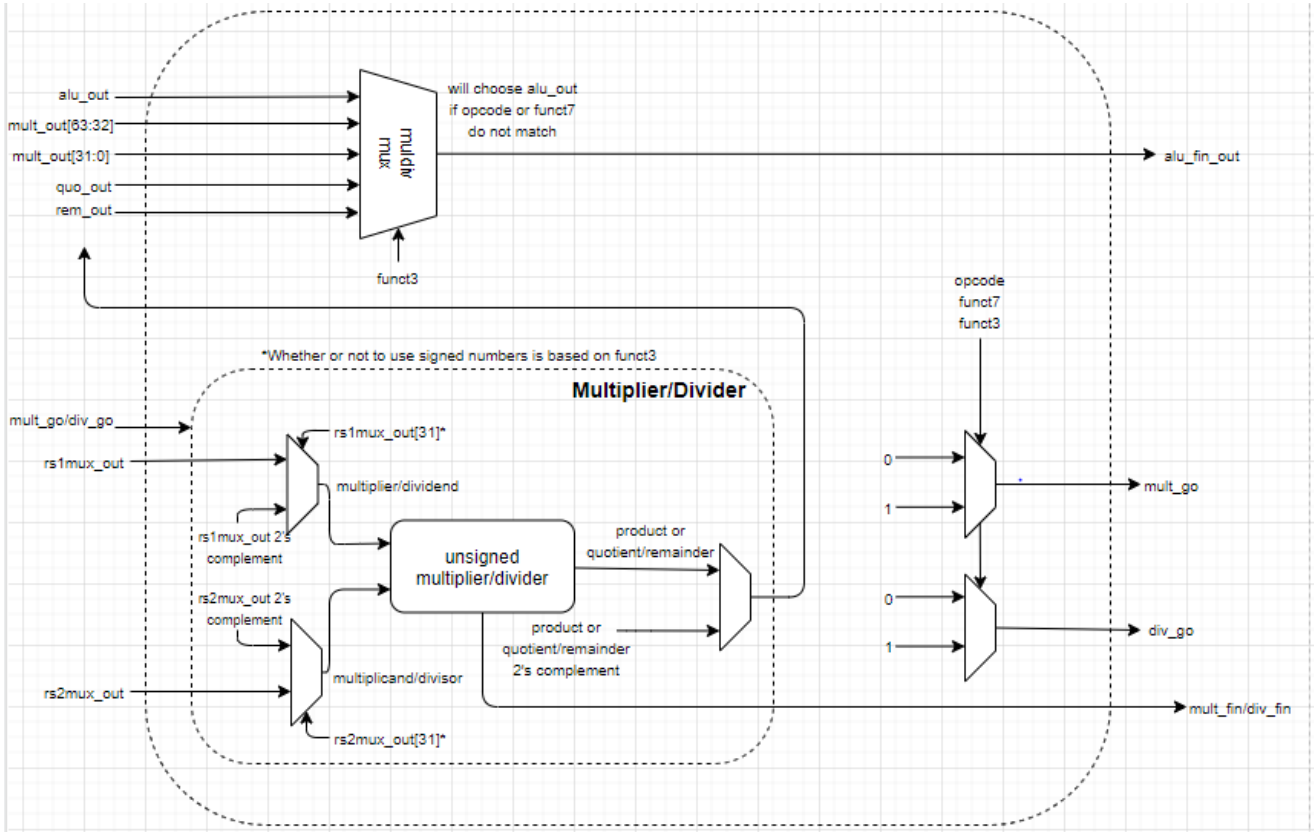
The final feature we aimed to implement was support for the standard “M” extension for the RISC-V, including operations that perform signed and unsigned 32-bit multiplication and division. These new commands are mainly for more intensive programs that include multiplying or dividing numbers, and can cut down runtimes decently effectively as the numbers get larger and larger. While we weren’t able to fully integrate this extension into our cache and get it returning the correct values in registers upon running competition code, thorough unit testing with seemingly perfect values leads us to believe that there may have been some error in the muldiv-datapath linkage or the stalling logic, rather than in the functionality of the modules themselves.

The flows of our M extension modules themselves aren’t overly complicated, and function similarly to one another. Multiplication uses a simple add-shift algorithm, and subtraction uses the standard shift-subtract algorithm. Both unsigned algorithms consist mainly of waiting for a start signal sent from our datapath, at which point the pipeline will stall and wait for a result. The multiplier/divider will then shift and add/subtract 32 times until they’ve calculated the correct value. From here, the module will send a “done” signal, indicating that the stall can be lifted and the output can be pushed through the pipeline. A rough flowchart of both the (unsigned) multiplier and divider can be seen below, as well as the control state diagrams for each.


Start
X = multiplier
Y = multiplicand
sum = 0
counter = 32 (n bits)
Y[0] = 1?
Yes
No
sum = sum + X
X << 1
Y >> 1
counter--
No
counter = 0?
Yes
Done
Start
X = dividend
Y = divisor
Z = 0
counter = 31
X << 1
Z << 1
Z = Z - Y
Z < 0?
Yes
No
X[0] = 0
Z = Z + Y
X[0] = 1
counter--
No
counter = -1?
Yes
Done
Quotient in X
Remainder in Z



Both the multiplier and divider contain a core module that performs their respective unsigned operations, with the main module keeping track of whether or not to account for signed factors and passing the correct value in. The diagram below illustrates this “wrapper” module. It also describes some muxes contained in our datapath to reroute the alu_out signal and overwrite it to push it through the pipeline.



Considering that the M-extension is more high-level than many of the other advanced features, the best way for us to evaluate the change in performance was to run two near identical programs, with one using adds and the other using muls, or with subs and divs. To realize this task, the team built two different factorial programs, as efficient as we could manage, and ran them with the same test values. We found that with lower numbers, the factorial program using add was faster than the mul factorial program. This is likely due to every mul operation taking a full 32 cycles to complete, whereas add operations only take 1. While there are obviously more adds than muls in their respective programs, smaller numbers tipped the balance of total cycles needed in favor of adds. Plugging in 12! (the highest factorial that will still fit in a 32-bit result) as a test value for each, however, we saw a 17% speedup when our mul operation was used over the add operation.

For more intensive and complex multiplications, the M-extension was definitely a step up from our basic processor's capabilities. However, with lesser, simpler programs, the M-extension as we implemented it isn't quite up to par, and using additions works more quickly.

Conclusion

In this project, we built a pipelined RISC-V CPU and implemented L2 Cache, Eviction Write Buffer, OBL Prefetching, and M-Extension for our Advanced Features. We were able to achieve full correctness as well as performance optimizations through our Advanced Features. Our large L2 Cache significantly reduced our execution times at the expense of power efficiency and Fmax, while our Eviction Write Buffer was not nearly as beneficial as we had hoped. The great benefit of the L2 cache showed us very directly the importance of memory technology and that there are many aspects of improvements across computer architecture, not all of which are obvious at first. We are also optimistic that the 8-way L2 cache's performance can set our design apart from the competition and give our processor an edge.

We were also surprised that our implementation of OBL Prefetching on Instruction Cache had an adverse effect on the execution times of our programs. We came up with two reasons related to the data requests that could explain the slowdown caused by OBL Prefetching, and sought additional performance metrics. Although the data supported our hypothesis, we found that the negative impact caused by the reasons did not match the degree of slowdown that we observed in execution times. We are still not sure what other factors could hurt performance, but we tried our best to figure it out.

Although the obvious goal of this project was to design, build, and verify a pipelined processor, it is clear to us that there are many slightly more subtle goals. We had to work collaboratively as a team, use our experience and resources available to plan advanced features, and often consider engineering trade offs of each aspect of the design, taking into account not only timing but also power and design complexity. We are now able to compare our predictions to actual results and gain a better understanding of the things that did not go as planned in order to improve in the future.

All in all, this project has been a very rewarding experience. We were given the opportunity to apply the concepts learned in lecture and gain a better understanding of how CPUs work, and we are pleased that we were able to successfully implement the original design and advanced features while maintaining close to the target Fmax. Although there are some drawbacks to our design, we are nonetheless hopeful for the results of the competition. The process of designing, implementing, and verifying our SystemVerilog code has greatly improved our problem-solving and debugging skills. In addition to all the technical skills learned, we gained experience working in a team remotely, similar to how we would be working in industry. As the great Immanuel Kant once said, “Experience without theory is blind, but theory without experience is mere intellectual play.” ECE 411 has been the ultimate combination of theory and practice, and we will carry the skills learned in this class throughout the rest of our careers.