Implementing Tower of Hanoi simulation via Two Stage Pipelined RISC-V ISA Processor

Zhanhui Chen
500131284

I. Introduction

The basic RISC-V instruction consisted of mainly six different formats. The RISC-V micro-processor is capable of pipelining, which provides a considerable speedup to the processor in general when implemented. However, the pipelining strategy might also introduce the risk of data, control and structural hazards during instruction flows. Therefore, additional methods like Forwarding and Interlock need to be implemented to handle the hazards. While a two stage pipelined processor has been successfully designed using System Verilog in previous work, the goal of this project is to implement a puzzle called Tower of Hanoi using the working pipelined and produce correct results accordingly. In this project, the Tower of Hanoi is a mathematical puzzle or algorithm consisting of 3 rods and 4 disks of various diameters, which can slide onto any rod. The puzzle begins with the disks stacked on one rod in order of decreasing size, the smallest at the top, thus approximating a conical shape. The objective of the puzzle is to move the entire stack to the last rod. While the initial program is written in a C language, the implementation used the RISC-V 32 bit instruction set, was written in System Verilog and simulated in ModelSim. Although there are many possible ways to complete the puzzle, there are a few constraints in this project, such as potential hazards arising from the assembly languages and the cache organization of the program. The aim of the project is to also optimize the program such that the effect of constraints are minimized.

II. Background

An RISC-V instruction can be broken down into five successive steps: instruction fetch (IF), instruction decode and register file fetch (ID), execution (EX), memory allocation (MA) and write back (WB). In a single cycle processor, these steps will be done within one cycle.

In the previous project, I have designed a two staged pipeline processor that can run with no error. A two-stage pipeline processor allows two instructions to execute in parallel. It involves breaking the execution of an instruction across two clock cycles, or it could be multiple cycle if a more advance pipelined processor is used. With pipelining, the computer architecture allows the next instructions to be fetched while the processor is performing arithmetic operations, holding them in a buffer close to the processor until each instruction operation can be performed. The staging of instruction fetching is continuous. The result is an increase in the number of instructions that can be performed during a given time period. [1]As a result, the overall frequency is increased because the critical path of the processor has been divided. In a single cycle processor, the critical path is 5 because an instruction needs to complete all 5 steps before the next instruction starts. In an optimised pipelined processor, the pipeline registers are placed in the middle of the execution path, such that the critical path for the processor can be split into smaller fraction. In this implemented pipeline processor, the pipeline registers are placed before the ALU. By doing this, the data path is split into 2 parts, the first part includes Instruction Fetch and Decode, and the second part includes Execution, Memory Access and Write Back. Hence, the critical data path is now 3 out of 5, which is the second part, and theoretically the frequency of such pipeline processor should be faster than the single cycle processor.

This does seem more efficient than single cycle processor but implementing pipelining can also introduced many hazards. A data hazard occurs if an instruction reads a register that a previous instruction overwrites in the future cycle. Because the processor can only write the data to register file at the beginning of the cycle, the read instruction will fetch the value in the register file before the write starts in the current cycle. As a result, an old value will be read, and the computation is therefore incorrect. A good choice to use a Forwarding strategy. Forwarding is the concept of making data available to the input of the ALU for subsequent instructions, even though the generating instruction hasn't gotten to WB in order to write the memory or registers. This is also called bypassing.[2] A simple structure of Forwarding is displayed below in figure II-A-1.

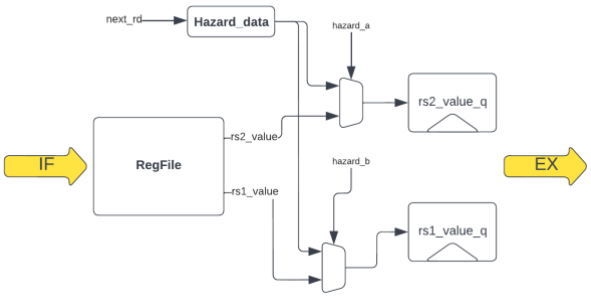


Figure II-1 Forwarding Strategy

Here the hazard_data will take the computed result, next_rd, at the positive edge of the cycle and points to the multiplexers inserted before the pipelined rs values. The hazard control signals are computed by comparing the current write back register address and the register read addresses that are going to execute in the next cycle. If the address is the same, then there is a data hazard. And if there is a hazard, then the hazard_data will be assigned to the particular rs value(s), since the hazard_data is the updated data value from the last computation, that just has not been overwritten in the register file. For example, here are two consecutive instruction:

```
addi x10, zero, 2
addi x10, x10, 3
```

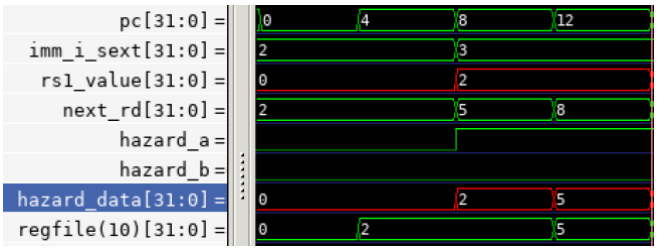


Figure II-2 Simulated result of the example 1

From these two line codes, we can see a hazard in the second line that the first operator is the same as the return address in the first line. So a data hazard will occur as the old value of x10 will be fetched by the second line. Hence, via bypassing the processor can fetch the calculated result to the second line before the result being written back. In the simulation figure II-2, we can see that the first line is executed in pc = 4, such that a "2" appears in "next_rd". Under the bypassing strategy, we see the "hazard_a" is high, indicating that the first operant is hazarded, so a hazard data will be fetched to "rs1_value" as desired.

However, this strategy does not deal with control hazards in this processor. And because the natural property of a branch instruction, it not only involves an offset to the program counter, but also need to compare two values, the branch instruction in pipeline design has potential data and control hazards. Hence a new method needs also to resolve the hazard problem. Control hazard occurs whenever the pipeline makes incorrect branch prediction decisions, resulting in instructions entering the pipeline that must be discarded. In the pipeline register, before any strategy is used to resolve hazards, if the current instruction is a branch or jump, the next instruction has been fetched and it is already waiting in the pipeline register. In that case, the next instruction will be executed and overwritten regardless of the branch condition. This is not wanted because in the case of a branch or jump, the next program counter is not necessarily PC + 4, so a PC + 4 instruction should not be executed sometimes. A solution to this problem is via stalling.

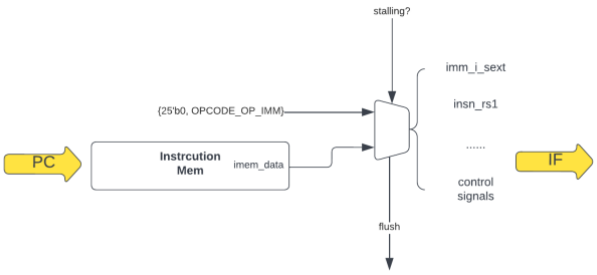


Figure II-3 Stalling Strategy

Figure II-A-3 shows the stalling implementation of this two- stage pipeline processor. Like a 'NOP', if a control hazard happened, the next cycle will be filled with a unaffected instruction and the program counter will freeze if necessary. The instruction {25'b0, OPCODE_OP_IMM} here will write 0 to the zero register, and because the zero register is hardwired to 0, it would not conflict with the other instructions. This strategy has been used to deal with the branch and unconditional jump instructions. Because the property of the pipeline registers, while executing the current instruction, the next instruction is already waiting in the pipeline registers. Hence stalling here is to replace the next instruction with a unaffected instruction to accomplish a 'NOP'. One drawback from the implementation is that this cannot reduce the number of cycles for the program. In the pipeline design, Forwarding is used instead of Branch Prediction, this is due to the consideration that Forwarding can result in a greater reduction in the number of cycles compared to branch prediction, as a prediction is not always true. For example, here are two consecutive instruction:

```
    addi x10, zero, 10
    addi x1, zero, 10
    beq  x10, x1, end
    addi x10, x10, 5
    ebreak
loop:
    add  x10, x10,  x1
```

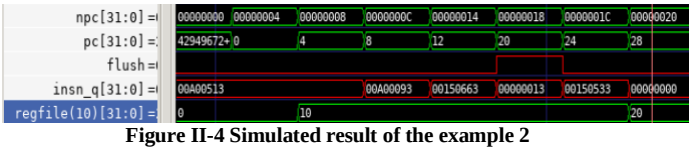


Figure II-4 Simulated result of the example 2

In the example above, we see that there is a branch instruction in the third line. This will introduce a control hazard such that if the branch is taken but the next code will still fetch to the processor and executed since there is a cycle delay in the processor. Hence, via stalling strategy, the processor will detect the branch instruction from its control signal and rise "flush" high, meaning that the next instruction is going to be the unaffected instruction, {25'b0, OPCODE_OP_IMM}. In the simulation result figure II-4, we can see that the branch is executed in pc = 12, and the flush is high in the next cycle, therefore the instruction fetched now becomes "00000013". In this way, even if the instruction is executed the program will not be affected. Toward the end, we see that the final result in register 10 is 20, as desired, instead of 15 if the branch is not taken.

III. ARCHETECTURE

A. *Program Characterization*

The program is written in C language, it is the solution to the puzzle "Tower of Hanoi". The program file has used a recursion to solve the algorithm. First, the program has a "Hanoi" function that passes the N (current number of disk), "from_rod", "to_rod" and "aux_rod". Then the function will call itself with N-1 disk. The program will then store the move in an array using a "move" function and keep count of the current move. Again make a function call of "Hanoi" for N-1 disk. If we print the moves in the terminal, there will be 15 moves in total (see appendices for terminal output). Hence, in order to run this program using the pipeline processor, the program needs to be converted into assembly language. After removing all the "printf" in the c program, we generate a ".s" source code file that is written in assembly. This source code file has the same execution order and content compared to the c file, such that one can run the assembly program and get the same result as in c file simulation. In the c file, the main function is coded in the very last block of the program but the program will execute the main function first, however, when generating the source code file, the main block is also coded in the very last block. Unlike the C program, the assembly language

will execute the first instruction it reads on the top, in this case is the “move” function. This introduces problems when executing the source code file, because the “move” function is a function that needs to be called first and it will finish execution and return to the caller. Without a caller specified, or explicit without a right return address, a “ret” command inside the “move” function will restore the program counter back to 0, resulting in an infinite loop of the program. Hence, a good solution to the problem is to allow the program counter to jump to the main function by adding a few lines of code before “move”. Therefore, an adjustment is to first load a large and positive value to the stack pointer, such that when storing data value to the memory using the offset of stack pointer value, the memory address is valid. Then secondly, ask the program to jump to the main function using “call” and after executing the main function the program can stop using “ebreak”. In this way, the execution of the program is success. The modified source code file is renamed to test2.s and it will be sused to run the simulation instead of the original one. (see appendices for generated assembly code and modified version)

Using the two-stages pipeline processor, the source code file is executed in 1066 cycles. The execution will begin in a main function that will call the Hanoi function to run the recursive algorithm. Inside Hanoi, there is a branch condition, corresponding to the “if” statement inside the “Hanoi” function in c. If the branch is taken, the program will call “move” in “.L6”. Continuing from the branch condition in “Hanoi”, the function will call itself again, then call a “move” and finally call itself again. The program used register 10, 11, 12 and 14 to store the computed result and one can find out under the “move”. Recall that from the c file, the “move” function is used to store the result and the current number of moves. This is corresponding to the store instructions “sh” and “sw”. We can see that the a4 register, register 14, is incremented by 1 every time before it is stored in the data memory, so it is the number of moves of the algorithm. And register a0, a1 and a1, corresponding to register 10, 11 and 12, are stored in some offset address of the value in register 15, so they are disk number and rod numbers. It is not hard to find that out by printing their values in the terminal. Inside the “move”, there is a command called “ret”, which simply restores the program counter to the next instruction from its caller. Hence, here the “Hanoi” is called the “move” block, so the program can return to “Hanoi” correctly and continue execution.

Because we have removed the “printf” in the c program, in the case that one wants to print the result that is simulated using pipelined System Verilog, a display function is added to a sequential logic block. In every positive edge of the current circuit, if the current program counter is 52 (32'h00000034), that is the cycle after the result has been computed and updated, display the values inside those registers. This can achieve a print command to the terminal and verify the correctness of the result.

B. Architecture of Cache

Cache simulation in my RISC-V processor is to use spike under a virtual machine. Spike can simulata variety of cache configuration by changing the parameters of cache, such as the associativity level, cache size and the block size. To simulate the Hanoi program, we need to generate an ELF file from the original C program. Then attach the ELF file name to the end of the spike command. The cache has three important parameters, sets, ways and blocks. If one was to describe cache in a 2D table, set is corresponding to the number of rows in the cache. Way is corresponding to the number of columns in the cache. For each location in the cache organization, each cell contains a tag and the data content in a simple understanding. The number of blocks is equivalent to the number of cells available in the cache. The total block size is calculated using set*way*block. There are two memories in this program that need simulation, the instruction and data memory. The design of these two memories has caused them to have different configurations of cache. In this project, spike command is used to simulate a L1 cache. Simulation can reflect on the miss rate of the instruction and data memory. A cache's primary purpose is to increase data retrieval performance by reducing the need to access the underlying slower storage bank. Trading off capacity for speed, a cache typically stores a very small subset of data especially for a L1 cache, in contrast to databases whose data is usually complete and durable. Because there is a limited number of data that can be stored inside the cache, inevitably sometimes the data the program asks for is not available in the cache, and the program needs to fetch it inside the large storage bank, resulting in a significant time cost. This is called the miss rate, the total cache misses compared to the total number of memory requests expressed as a percentage over a time interval. Therefore, theoretically increase the size of cache can reduce the miss rate happens, because the cache can duplicate a large block of data from a location, assuming the case of spatial or temporal locality (spatial locality: all instructions which stored nearby to the recently executed instruction have high chances of execution; temporal locality: an instruction which is recently executed have high chances of execution again). However, the rule is that one cannot finitely increase the size of the cache as this will introduce growing cost on the time to hit in the cache (hit latency) due to its size and the penalty on cache misses. This cost, proportional to the size of the cache, refers to AMAT and it is calculated using hit latency plus the miss rate multiple by the cost. Several studies have shown that the performance of caches depends on the relationship between the cache block size and the granularity of sharing and locality exhibited by the program. Large cache blocks exploit processor and spatial locality, but may cause unnecessary cache invalidations due to false sharing. Small cache blocks can reduce the number of cache invalidations, but increase the number of bus or network transactions required to load data into the cache.[3] Hence it is crucial for designers to design a good cache configuration that both preserve an excellent hit rate and smaller cost or cache size.

IV. RESULTS

The whole program can be executed by “make” in the terminal, it will first produce a ELF file of the program.c and execute the modified version of Hanoi (test2.s) assembly code using the two-stages pipeline processor, written in System Verilog, and then run a spike command on the cache configuration that gives the lowest miss rate, finally another spike instruction will run and produce an optimized configuration that have a very good AMAT with acceptable

miss rate around or below 5%. The program result is shown below.

```
Move          0 disk          1 from    A to    B
Move          1 disk          2 from    A to    C
Move          2 disk          1 from    B to    C
Move          3 disk          3 from    A to    B
Move          4 disk          1 from    C to    A
Move          5 disk          2 from    C to    B
Move          6 disk          1 from    A to    B
Move          7 disk          4 from    A to    C
Move          8 disk          1 from    B to    C
Move          9 disk          2 from    B to    A
Move         10 disk          1 from    C to    A
Move         11 disk          3 from    B to    C
Move         12 disk          1 from    A to    B
Move         13 disk          2 from    A to    C
Move         14 disk          1 from    B to    C
x10=1, cycles=1066
```

Figure IV-1 Hanoi simulated result

All statements are output from the System Verilog, and the moves parameters all matches the correct output. The number of cycle executed is 1066 because stalling is used inside the processor, hence more cycles are need to complete the program.

To find the good cache configuration, I have tested all the combinations from 1 to 1024 for each parameter (steps in power of 2), and all the configurations are evaluated. (see appendices for all simulated results). One finding is that as the block increases, the overall miss rate will decrease dramatically and finally reach a minimum point, hence the block is inversely proportional to the miss rate. And the AMAT is very small at the beginning and experiences an expoentially increase above cache size 10 to the power of 5. Therefore, the best performance for instruction and data memory is 1 : 64 : 1024 (set : way : block) and 1 : 128 : 1024. And the simulated result is below:

```
D$ Bytes Read:            462132
D$ Bytes Written:         157282
D$ Read Accesses:         119199
D$ Write Accesses:        39972
D$ Read Misses:           8
D$ Write Misses:          90
D$ Writebacks:            0
D$ Miss Rate:             0.062%
I$ Bytes Read:            1714892
I$ Bytes Written:         0
I$ Read Accesses:         428723
I$ Write Accesses:        0
I$ Read Misses:           64
I$ Write Misses:          0
I$ Writebacks:            0
I$ Miss Rate:             0.015%
```

Figure IV-2 Best cache configuration

The miss rate of data memory is 0.062% and the instrction miss rate is 0.015%. But as mention, the totoal cache size is so large that the cost has reached to an untolerable level. So an optimised cache configuration is tested again, with instruction and data memory 1 : 2 : 128 and 2 : 2 : 64. The simulated result is shown below:

```
D$ Bytes Read:            462132
D$ Bytes Written:         157282
D$ Read Accesses:         119199
D$ Write Accesses:        39972
D$ Read Misses:           4796
D$ Write Misses:          3502
D$ Writebacks:            4767
D$ Miss Rate:             5.213%
I$ Bytes Read:            1714892
I$ Bytes Written:         0
I$ Read Accesses:         428723
I$ Write Accesses:        0
I$ Read Misses:           11731
I$ Write Misses:          0
I$ Writebacks:            0
I$ Miss Rate:             2.736%
```

Figure IV-3 Optimised cache configuration

The miss rate of data memory is 5.213 and the instrction miss rate is 2.736%. Although the data memory miss rate has reached over 5%, it has break even with the significant small cost, 0.16 and 0.19.

V. Discussion & Conclusion

Improvements can be made to the pipeline design in the future. The problem of the design is that too many assignments has increased the execution time since and the pipeline register might be inserted in a not-so-good position from coding that it increased the critical path of the processor, so changing how stalling and Forwarding can be implemented using fewer variables could help with the execution time. While this is not implemented, in the future study, the cache miss penalty can be reduced by proposing a new cache organization that adjusts the size of data blocks according to recent reference patterns. In this new cache organization, blocks are split in two when false sharing occurs, and merged back together to exploit spatial locality.[3] Nevertheless, this project has successfully demonstrated the use of assembly code and System Verilog to implement a harder program. Through spike simulation, an optimized cache configuration is able to be applied to the program, achieving a desired miss rate and penalty cost.

REFERENCES

1. Anonymous Contributor, Whatls.com, "***pipelining***", April 2005, https://www.techtarget.com/whatis/definition/pipelining
2. DR A.P.SHANTHI, Computer Architecture, "***Handling Data Hazards***", https://www.cs.umd.edu/~meesh/411/CA-online/chapter/handling-data-hazards/index.html
3. Cezary Dubnicki; Thomas J. LeBlanc, ResearchGate, "***The Effects of Block Size on the Performance of Coherent Caches in Shared-Memory Multiprocessors***", May 1993, https://www.researchgate.net/publication/48304603_The_Effects_of_Block_Size_on_the_Performance_of_Coherent_Caches_in_Shared-Memory_Multiprocessors

APPENDICES

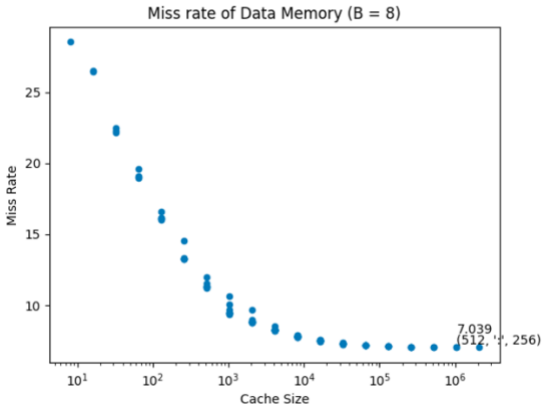
MAKEFILE CODE

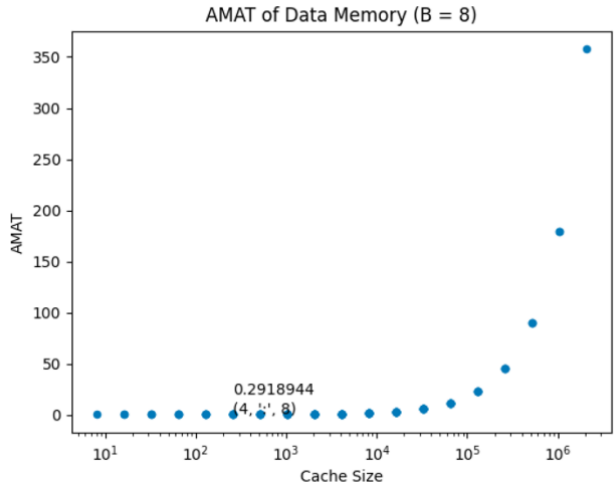
```
TOOLCHAIN_PREFIX?=riscv32-unknown-elf-
CC=$(TOOLCHAIN_PREFIX)gcc
AS=$(TOOLCHAIN_PREFIX)as
CFLAGS= -O1 -march=rv32i -mabi=ilp32
SOURCES = nerv.sv testbench.sv testbench.py
RDOCKER = docker run --platform linux/amd64 -it -e DISPLAY=host.docker.internal:0 -v `pwd`:/config phwl/elec3608-riscv:latest

OBJ=    program
all:    $(OBJ) test2.out Best_cache Optimised_cache

# produce an executable with the printf for execution in spike
$(OBJ): $(OBJ).o
    $(CC) $(CFLAGS) $(OBJ).o -o $(OBJ)

$(OBJ).s:   $(OBJ).c
    $(CC) -S $(CFLAGS) $(OBJ).c

$(OBJ).o:   $(OBJ).s
    $(AS) --traditional-format -march=rv32i -mabi=ilp32 -o $(OBJ).o $(OBJ).s

# execute $(OBJ) using spike simulator
test:
    -spike --isa=rv32i --ic=2:4:8 --dc=2:4:8  /opt/riscv/riscv32-unknown-elf/bin/pk $(OBJ)

Best_cache:
    -spike --isa=rv32i --ic=1:64:1024 --dc=1:128:1024 /opt/riscv/riscv32-unknown-elf/bin/pk $(OBJ)

Optimised_cache:
    -spike --isa=rv32i --ic=1:2:128 --dc=2:2:64 /opt/riscv/riscv32-unknown-elf/bin/pk $(OBJ)

# generate a listing in $(OBJ).lst (without debug)
list:
    $(CC) -S $(CFLAGS) $(OBJ).c
    $(AS) -al $(OBJ).s > $(OBJ).lst
    -rm -f a.out

clean:
    -rm -f $(OBJ).lst $(OBJ) $(OBJ).s $(OBJ).o $(OBJ) $(OBJ).o $(OBJ).s
    rm -rf firmware.elf firmware.hex testbench testbench.vcd gtkwave.log
    rm -rf disasm.o disasm.s checks/ cexdata/ obj_dir
    rm -rf *.log *.asc *.json *.result *.vcd firmware.s *.tgz *.out

%.out: %.hex $(SOURCES)
    cp $*.hex firmware.hex
    python testbench.py testbench.sv

%.json: %.sv
    yosys -s $*.yosys -l $*-yosys.log

%.elf: %.s
    $(TOOLCHAIN_PREFIX)gcc -march=rv32i -mabi=ilp32 -Os -Wall -Wextra -Wl,-Bstatic,-T,sections.lds,--strip-debug

%.hex: %.elf
    $(TOOLCHAIN_PREFIX)objcopy -O verilog $< $@

rundocker:
    $(RDOCKER)
```

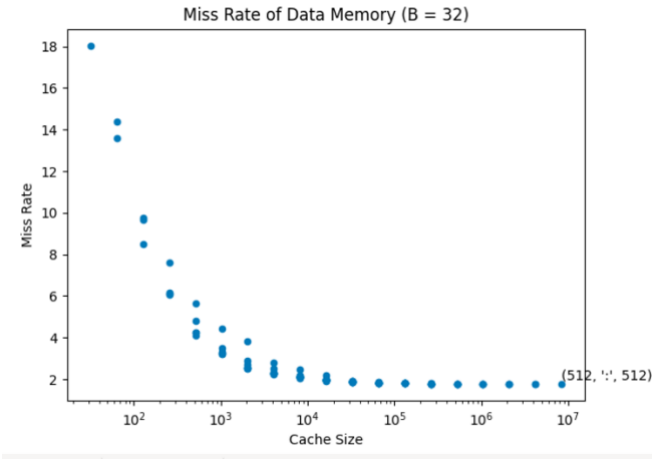
MAKE OUTPUT

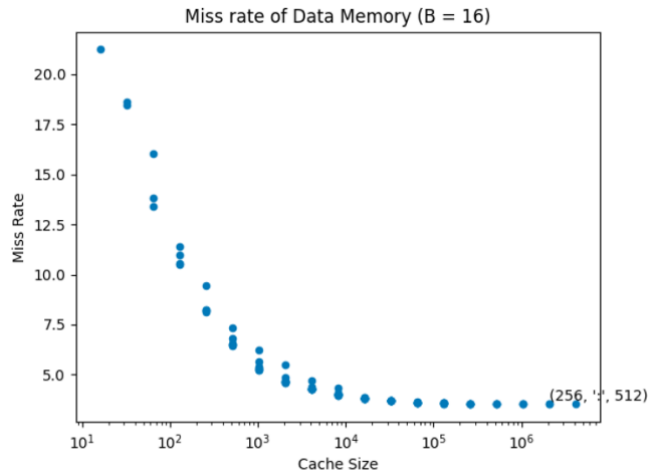
```
make[1]: Leaving directory '/config/obj_dir'
Move         0 disk         1 from    A to    B
Move         1 disk         2 from    A to    C
Move         2 disk         1 from    B to    C
Move         3 disk         3 from    A to    B
Move         4 disk         1 from    C to    A
Move         5 disk         2 from    C to    B
Move         6 disk         1 from    A to    B
Move         7 disk         4 from    A to    C
Move         8 disk         1 from    B to    C
Move         9 disk         2 from    B to    A
Move        10 disk         1 from    C to    A
Move        11 disk         3 from    B to    C
Move        12 disk         1 from    A to    B
Move        13 disk         2 from    A to    C
Move        14 disk         1 from    B to    C
x10=1, cycles=1066
spike --isa=rv32i --ic=1:64:1024 --dc=1:128:1024 /opt/riscv/riscv32-unknown-elf/bin/pk program
bbl loader
Move 0 disk 1 from A to B
Move 1 disk 2 from A to C
Move 2 disk 1 from B to C
Move 3 disk 3 from A to B
Move 4 disk 1 from C to A
Move 5 disk 2 from C to B
Move 6 disk 1 from A to B
Move 7 disk 4 from A to C
Move 8 disk 1 from B to C
Move 9 disk 2 from B to A
Move 10 disk 1 from C to A
Move 11 disk 3 from B to C
Move 12 disk 1 from A to B
Move 13 disk 2 from A to C
Move 14 disk 1 from B to C
D$ Bytes Read:            462132
D$ Bytes Written:         157282
D$ Read Accesses:         119199
D$ Write Accesses:        39972
D$ Read Misses:           4796
D$ Write Misses:          3502
D$ Writebacks:            4767
D$ Miss Rate:             5.213%
I$ Bytes Read:            1714892
I$ Bytes Written:         0
I$ Read Accesses:         428723
I$ Write Accesses:        0
I$ Read Misses:           11731
I$ Write Misses:          0
I$ Writebacks:            0
I$ Miss Rate:             2.736%
rm test2.hex test2.elf
```

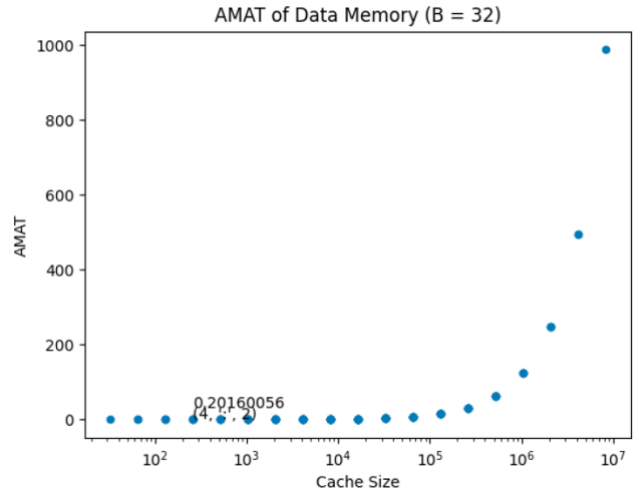
SPIKE SIMULATION

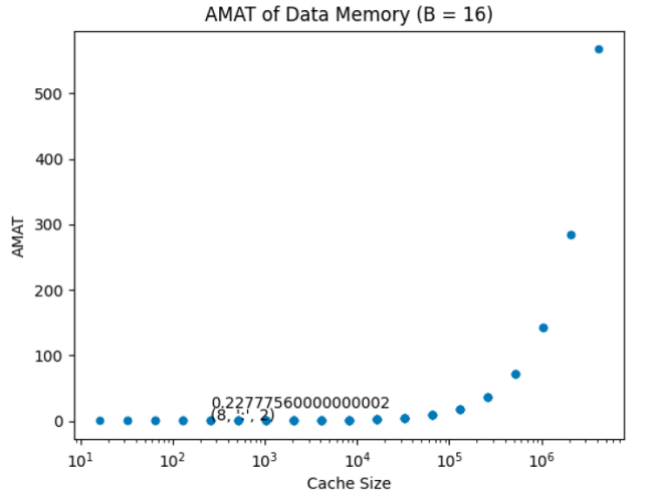


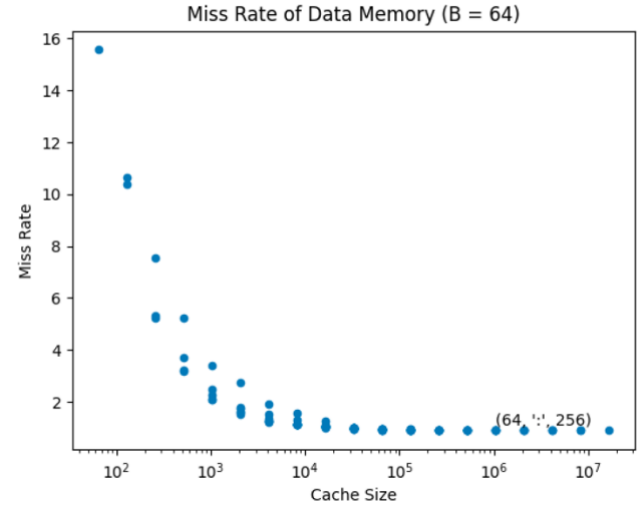

AMAT of Data Memory (B = 8)
AMAT
Cache Size
0.2918944
(4, ':', 8)

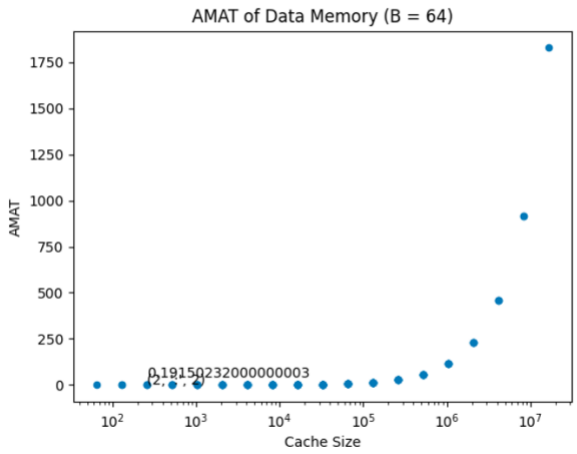

Miss Rate of Data Memory (B = 32)
Miss Rate
Cache Size
(512, ':', 512)

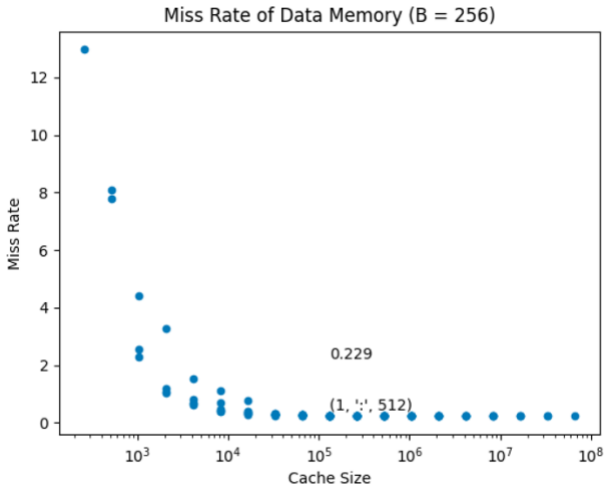

Miss rate of Data Memory (B = 16)
Miss Rate
Cache Size
(256, ':', 512)

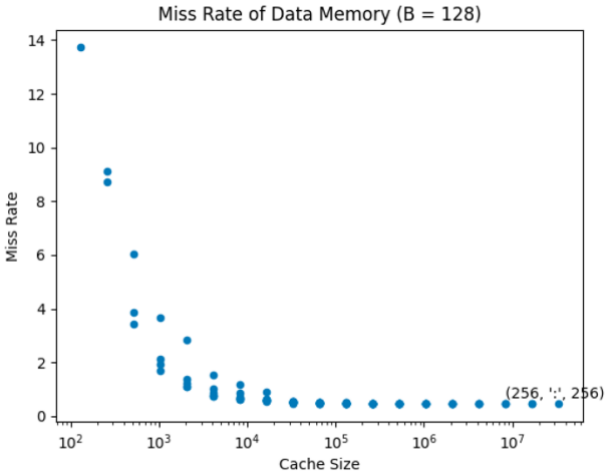

AMAT of Data Memory (B = 32)
AMAT
Cache Size
0.20160056
(4, ':', 2)

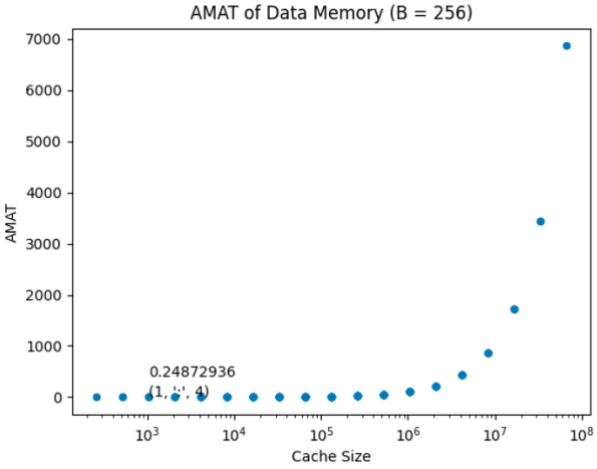

AMAT of Data Memory (B = 16)
AMAT
Cache Size
0.22777560000000002
(8, ':', 2)

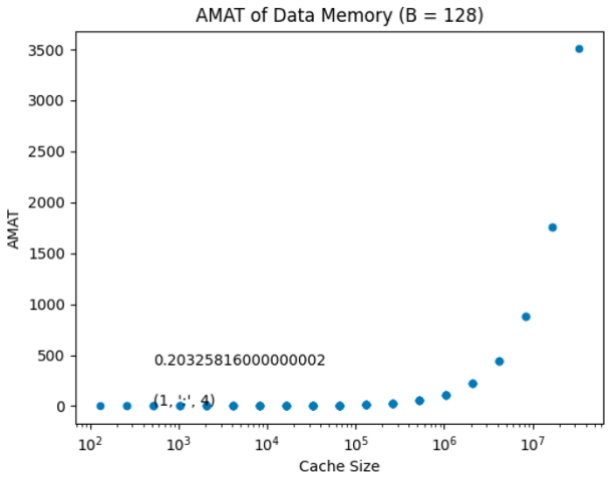

Miss Rate of Data Memory (B = 64)
Miss Rate
Cache Size
(64, ':', 256)

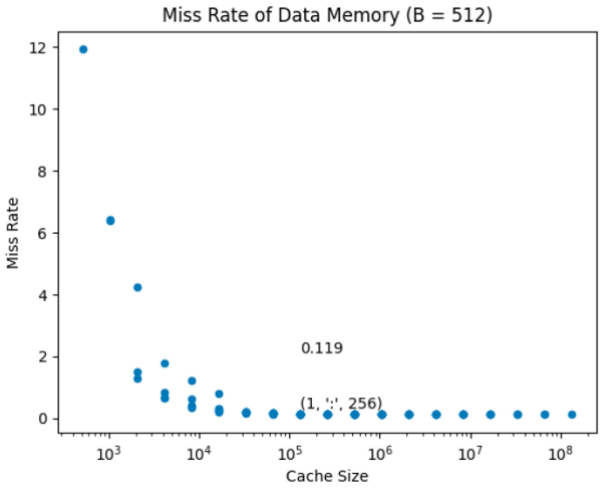

AMAT of Data Memory (B = 64)
AMAT
Cache Size
0.19150232000000003

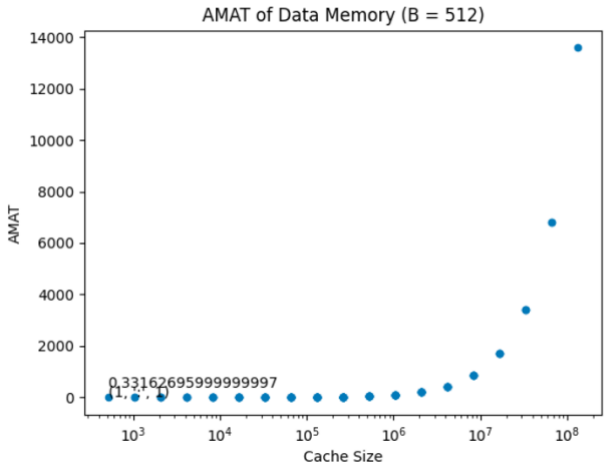

Miss Rate of Data Memory (B = 256)
Miss Rate
Cache Size
0.229
(1, ':', 512)

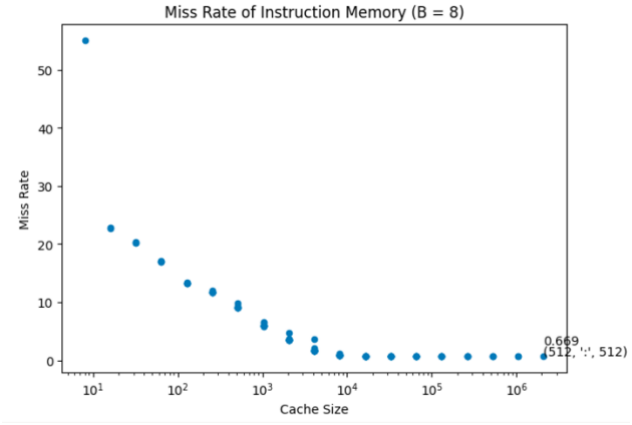

Miss Rate of Data Memory (B = 128)
Miss Rate
Cache Size
(256, ':', 256)

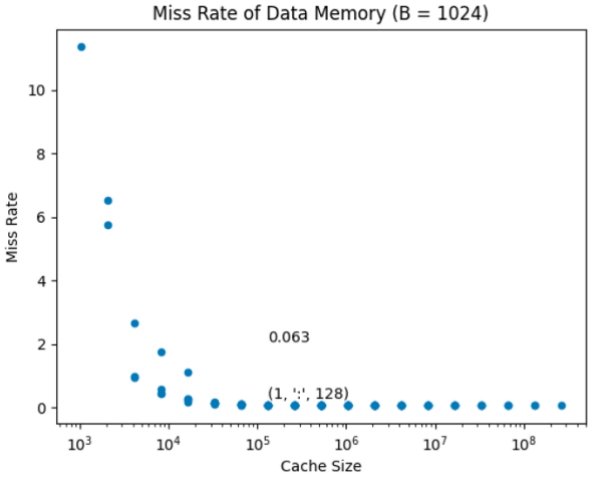

AMAT of Data Memory (B = 256)
AMAT
Cache Size
0.24872936
(1, ':', 4)

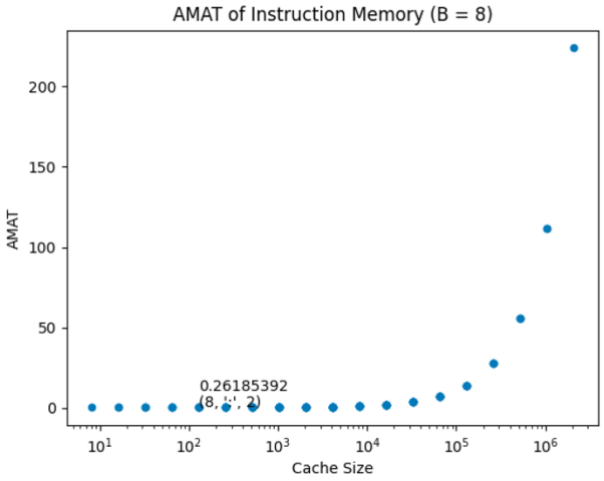

AMAT of Data Memory (B = 128)
AMAT
Cache Size
0.20325816000000002
(1, ':', 4)

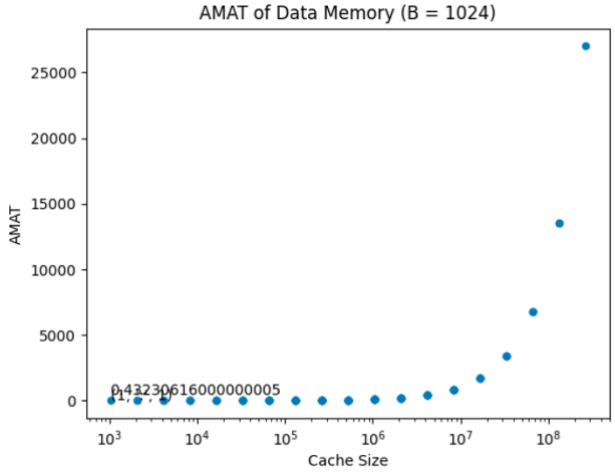

Miss Rate of Data Memory (B = 512)
Miss Rate
Cache Size
0.119
(1, ':', 256)

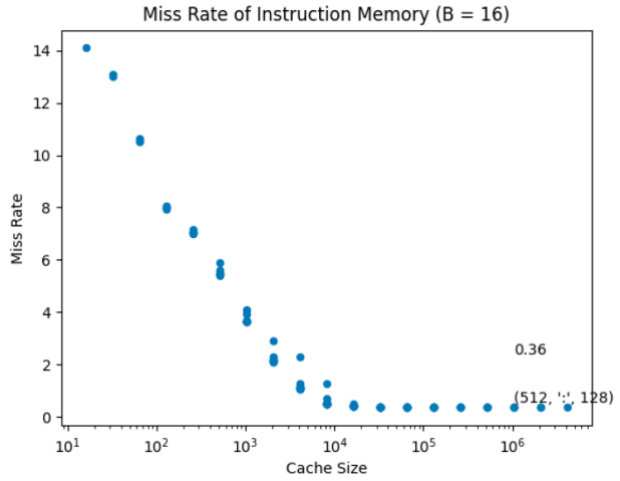

AMAT of Data Memory (B = 512)
AMAT
Cache Size
0.33162695999999997
(1, ':', 1)

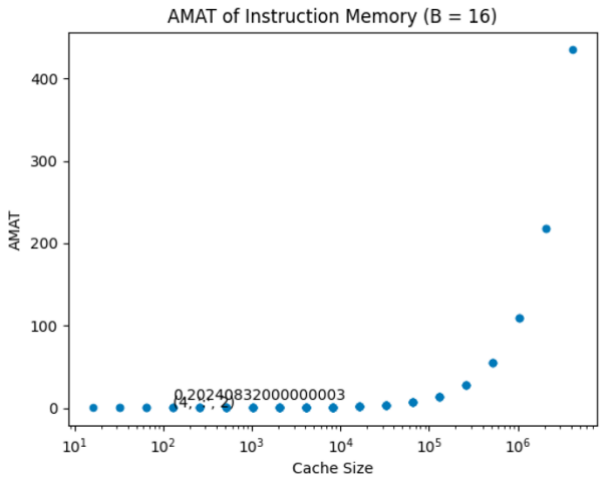

Miss Rate of Instruction Memory (B = 8)
Miss Rate
Cache Size
0.669
(512, ':', 512)

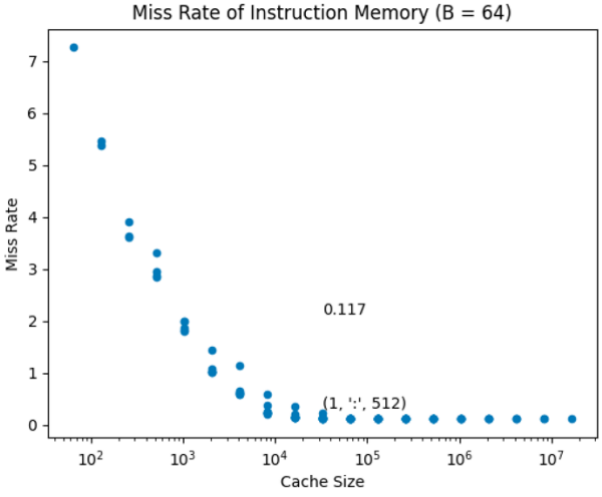

Miss Rate of Data Memory (B = 1024)
Miss Rate
Cache Size
0.063
(1, ':', 128)

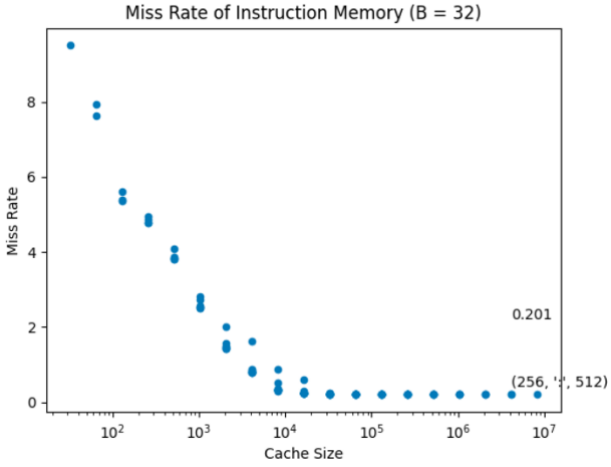

AMAT of Instruction Memory (B = 8)
AMAT
Cache Size
0.26185392
(8, ':', 2)

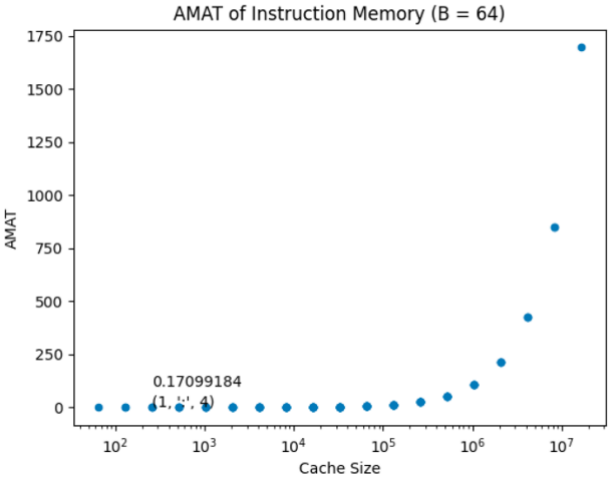

AMAT of Data Memory (B = 1024)
AMAT
Cache Size
0.43230616000000005
(1, ':', 1)

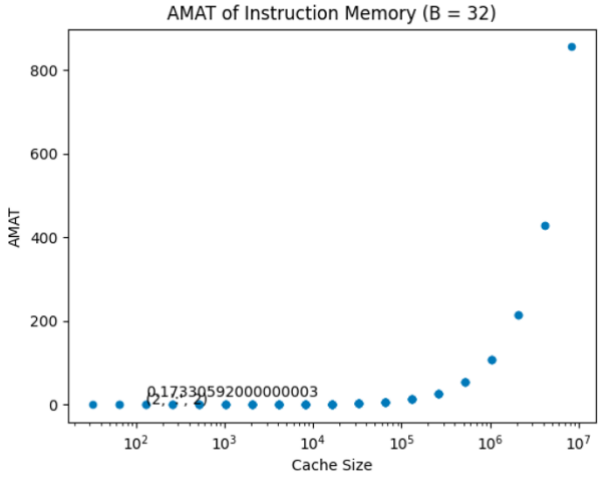

Miss Rate of Instruction Memory (B = 16)
Miss Rate
Cache Size
0.36
(512, ':', 128)

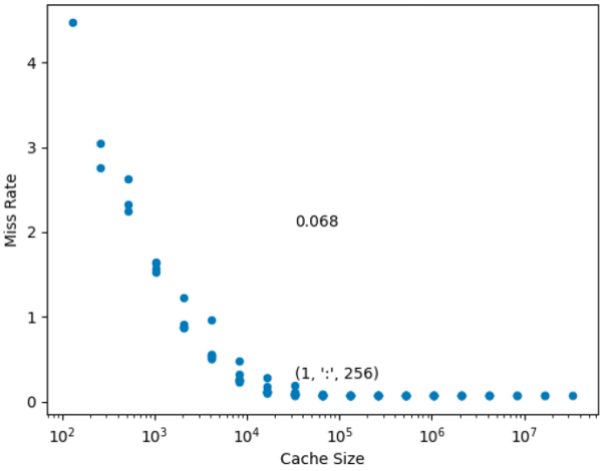

AMAT of Instruction Memory (B = 16)
AMAT
Cache Size
0.20240832000000003
(4, ':', 4)

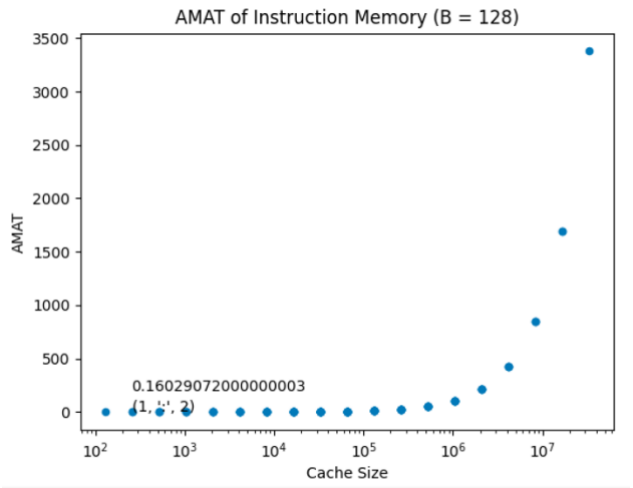

Miss Rate of Instruction Memory (B = 64)
Miss Rate
Cache Size
0.117
(1, ':', 512)

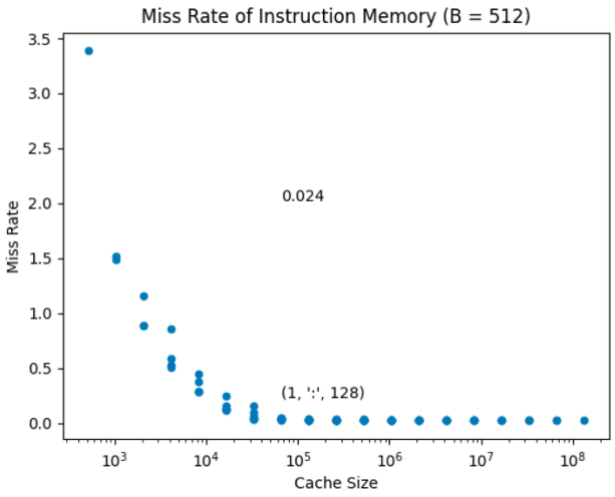

Miss Rate of Instruction Memory (B = 32)
Miss Rate
Cache Size
0.201
(256, ':', 512)

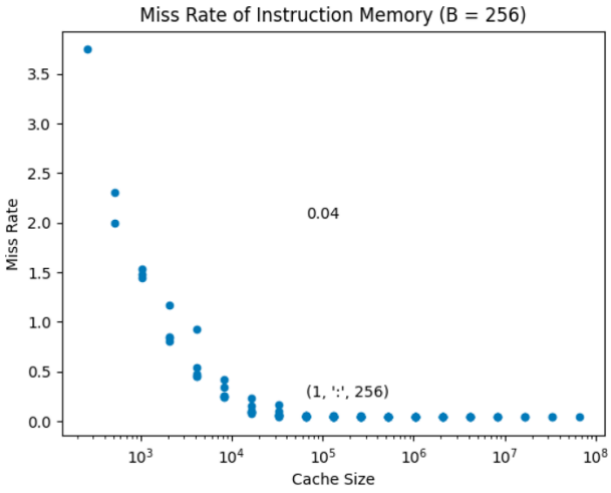

AMAT of Instruction Memory (B = 64)
AMAT
Cache Size
0.17099184
(1, ':', 4)

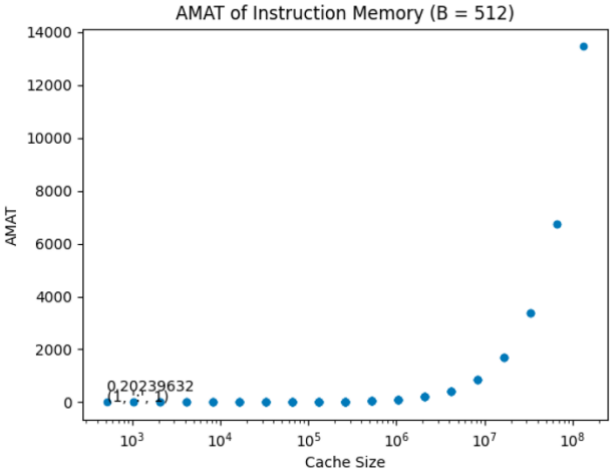

AMAT of Instruction Memory (B = 32)
AMAT
Cache Size
0.17330592000000003
(2, ':', 4)

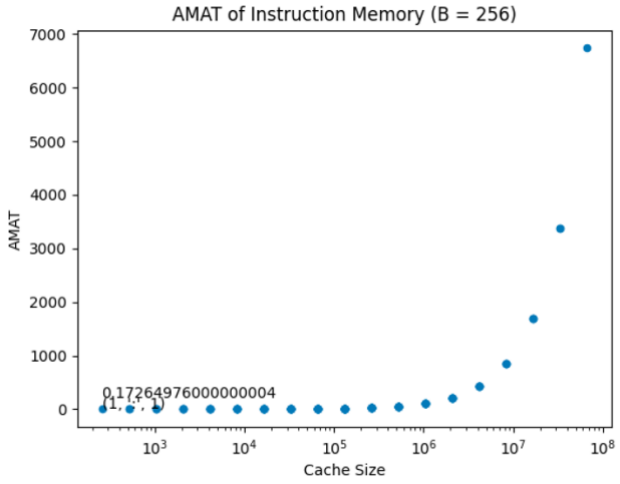

Miss Rate of Instruction Memory (B = 128)
Miss Rate
Cache Size
0.068
(1, ':', 256)

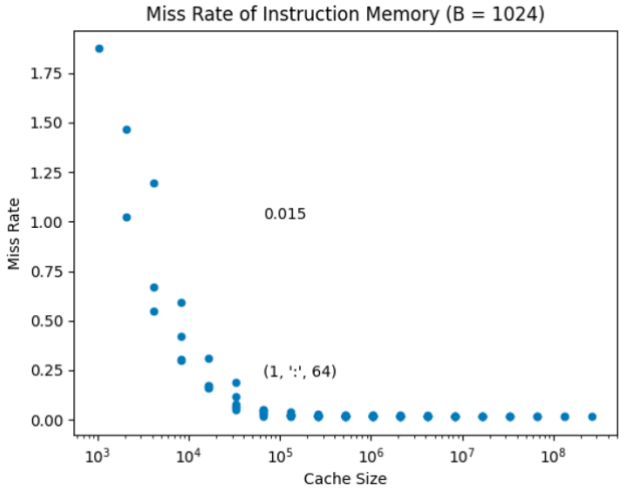

AMAT of Instruction Memory (B = 128)
AMAT
Cache Size
0.16029072000000003
(1, ':', 2)

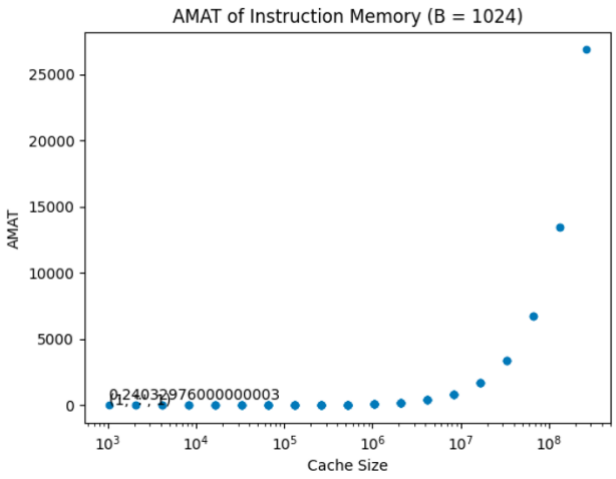

Miss Rate of Instruction Memory (B = 512)
Miss Rate
Cache Size
0.024
(1, ':', 128)


Miss Rate of Instruction Memory (B = 256)
Miss Rate
Cache Size
0.04
(1, ':', 256)


AMAT of Instruction Memory (B = 512)
AMAT
Cache Size
0.20239632
(1, ':', 1)


AMAT of Instruction Memory (B = 256)
AMAT
Cache Size
0.17264976000000004
(1, ':', 1)


Miss Rate of Instruction Memory (B = 1024)
Miss Rate
Cache Size
0.015
(1, ':', 64)



Final Spike Output

```
D$ Bytes Read:              462132
D$ Bytes Written:           157282
D$ Read Accesses:           119199
D$ Write Accesses:          39972
D$ Read Misses:             4796
D$ Write Misses:            3502
D$ Writebacks:              4767
D$ Miss Rate:               5.213%
I$ Bytes Read:              1714892
I$ Bytes Written:           0
I$ Read Accesses:           428723
I$ Write Accesses:          0
I$ Read Misses:             11731
I$ Write Misses:            0
I$ Writebacks:              0
I$ Miss Rate:               2.736%
```

Program Reusult

```
Move          0 disk          1 from    A to    B
Move          1 disk          2 from    A to    C
Move          2 disk          1 from    B to    C
Move          3 disk          3 from    A to    B
Move          4 disk          1 from    C to    A
Move          5 disk          2 from    C to    B
Move          6 disk          1 from    A to    B
Move          7 disk          4 from    A to    C
Move          8 disk          1 from    B to    C
Move          9 disk          2 from    B to    A
Move         10 disk          1 from    C to    A
Move         11 disk          3 from    B to    C
Move         12 disk          1 from    A to    B
Move         13 disk          2 from    A to    C
Move         14 disk          1 from    B to    C
x10=1, cycles=1066
```

Updated Two-stages Pipeline Code

```
module nerv #(
    parameter [31:0] RESET_ADDR = 32'h 0000_0000,
    parameter integer NUMREGS = 32
) (
    input clock,
    input reset,
    output trap,
    output logic [31:0] x10,

    // we have 2 external memories
    // one is instruction memory
    output [31:0] imem_addr,
    input  [31:0] imem_data,

    // the other is data memory
    output        dmem_valid,
    output [31:0] dmem_addr,
    output [ 3:0] dmem_wstrb,
    output [31:0] dmem_wdata,
    input  [31:0] dmem_rdata
);
    logic mem_wr_enable;
    logic [31:0] mem_wr_addr;
    logic [31:0] mem_wr_data;
    logic [3:0] mem_wr_strb;

    logic mem_rd_enable;
    logic [31:0] mem_rd_addr;
    logic [4:0] mem_rd_reg;
    logic [4:0] mem_rd_func;

    logic mem_rd_enable_q;
    logic [4:0] mem_rd_reg_q;
    logic [4:0] mem_rd_func_q;
```

```
// delayed copies of mem_rd
always @(posedge clock) begin
    mem_rd_enable_q <= mem_rd_enable;
    mem_rd_reg_q <= mem_rd_reg;
    mem_rd_func_q <= mem_rd_func;
    if (reset) begin
        mem_rd_enable_q <= 0;
    end
end

// memory signals
assign dmem_valid = mem_wr_enable || mem_rd_enable;
assign dmem_addr  = mem_wr_enable ? mem_wr_addr : mem_rd_enable ? mem_rd_addr : 32'h x;
assign dmem_wstrb = mem_wr_enable ? mem_wr_strb : mem_rd_enable ? 4'h 0 : 4'h x;
assign dmem_wdata = mem_wr_enable ? mem_wr_data : 32'h x;

// registers, instruction reg, program counter, next pc
logic [31:0] regfile [0:NUMREGS-1];
logic [31:0] insn;
logic [31:0] insn_q;
logic [31:0] npc;
logic [31:0] pc;
logic [31:0] ppc;

logic [31:0] imem_addr_q;

// instruction memory pointer
assign imem_addr = (trap || mem_rd_enable_q) ? imem_addr_q : npc;
assign insn = imem_data;

// rs1 and rs2 are source for the instruction
logic [31:0] rs1_value;
logic [31:0] rs2_value;
logic [31:0] rs1_value_q;
logic [31:0] rs2_value_q;

always @(posedge clock) begin
    imem_addr_q <= imem_addr;
    ppc <= pc;
```

```
    if((npc != pc +4) || (insn_opcode_q == OPCODE_JAL) || insn_opcode_q == OPCODE_JALR) begin
        insn_q <= {25'b0, OPCODE_OP_IMM};
        flush <= 1;
    end else begin
        rs1_value <= !insn_rs1 ? 0 : regfile[insn_rs1];
        rs2_value <= !insn_rs2 ? 0 : regfile[insn_rs2];
        imm_i_sext <= $signed(imm_i);
        imm_s_sext <= $signed(imm_s);
        imm_b_sext <= $signed(imm_b);
        imm_j_sext <= $signed(imm_j);
        insn_q <= insn;
        flush <= 0;
    end
end
assign rs1_value_q = hazard_a? hazard_data : rs1_value;
assign rs2_value_q = hazard_b? hazard_data : rs2_value;

// components of the instruction
logic [6:0] insn_funct7;
logic [4:0] insn_rs2;
logic [4:0] insn_rs1;
logic [2:0] insn_funct3;
logic [4:0] insn_rd;
logic [6:0] insn_opcode;

// split R-type instruction - see section 2.2 of RiscV spec
assign {insn_funct7, insn_rs2, insn_rs1, insn_funct3, insn_rd, insn_opcode} = insn;

logic [6:0] insn_funct7_q;
logic [4:0] insn_rs2_q;
logic [4:0] insn_rs1_q;
logic [2:0] insn_funct3_q;
logic [4:0] insn_rd_q;
logic [6:0] insn_opcode_q;

// split R-type instruction - see section 2.2 of RiscV spec
assign {insn_funct7_q, insn_rs2_q, insn_rs1_q, insn_funct3_q, insn_rd_q, insn_opcode_q} = insn_q;
```

```
localparam OPCODE_LOAD_FP    = 7'b 00_001_11;
localparam OPCODE_STORE_FP   = 7'b 01_001_11;
localparam OPCODE_MSUB       = 7'b 10_001_11;
localparam OPCODE_JALR       = 7'b 11_001_11;

localparam OPCODE_CUSTOM_0   = 7'b 00_010_11;
localparam OPCODE_CUSTOM_1   = 7'b 01_010_11;
localparam OPCODE_NMSUB      = 7'b 10_010_11;
localparam OPCODE_RESERVED_0 = 7'b 11_010_11;

localparam OPCODE_MISC_MEM   = 7'b 00_011_11;
localparam OPCODE_AMO        = 7'b 01_011_11;
localparam OPCODE_NMADD      = 7'b 10_011_11;
localparam OPCODE_JAL        = 7'b 11_011_11;

localparam OPCODE_OP_IMM     = 7'b 00_100_11;
localparam OPCODE_OP         = 7'b 01_100_11;
localparam OPCODE_OP_FP      = 7'b 10_100_11;
localparam OPCODE_SYSTEM     = 7'b 11_100_11;

localparam OPCODE_AUIPC      = 7'b 00_101_11;
localparam OPCODE_LUI        = 7'b 01_101_11;
localparam OPCODE_RESERVED_1 = 7'b 10_101_11;
localparam OPCODE_RESERVED_2 = 7'b 11_101_11;

localparam OPCODE_OP_IMM_32  = 7'b 00_110_11;
localparam OPCODE_OP_32      = 7'b 01_110_11;
localparam OPCODE_CUSTOM_2   = 7'b 10_110_11;
localparam OPCODE_CUSTOM_3   = 7'b 11_110_11;

// next write, next destination (rd), illegal instruction registers
logic next_wr;
logic [31:0] next_rd;
logic illinsn;
```

```
// setup for I, S, B & J type instructions
// I - short immediates and loads
wire [11:0] imm_i;
assign imm_i = insn[31:20];

// S - stores
wire [11:0] imm_s;
assign imm_s[11:5] = insn_funct7, imm_s[4:0] = insn_rd;

// B - conditionals
wire [12:0] imm_b;
assign {imm_b[12], imm_b[10:5]} = insn_funct7, {imm_b[4:1], imm_b[11]} = insn_rd, imm_b[0] = 1'b0;

// J - unconditional jumps
wire [20:0] imm_j;
assign {imm_j[20], imm_j[10:1], imm_j[11], imm_j[19:12], imm_j[0]} = {insn[31:12], 1'b0};

logic [31:0] imm_i_sext_q;
logic [31:0] imm_s_sext_q;
logic [31:0] imm_b_sext_q;
logic [31:0] imm_j_sext_q;
logic [31:0] imm_i_sext;
logic [31:0] imm_s_sext;
logic [31:0] imm_b_sext;
logic [31:0] imm_j_sext;

assign imm_i_sext_q = imm_i_sext;
assign imm_s_sext_q = imm_s_sext;
assign imm_b_sext_q = imm_b_sext;
assign imm_j_sext_q = imm_j_sext;

// opcodes - see section 19 of RiscV spec
localparam OPCODE_LOAD       = 7'b 00_000_11;
localparam OPCODE_STORE      = 7'b 01_000_11;
localparam OPCODE_MADD       = 7'b 10_000_11;
localparam OPCODE_BRANCH     = 7'b 11_000_11;
```

```
// logic [5:0] prev_rd;
logic [31:0] hazard_data;
logic hazard_a;
logic hazard_b;
logic flush;

logic trapped;
logic trapped_q;
assign trap = trapped;

always_comb begin

    npc = pc + 4;
    // defaults for read, write
    next_wr = 0;
    next_rd = 0;
    illinsn = 0;

    mem_wr_enable = 0;
    mem_wr_addr = 'hx;
    mem_wr_data = 'hx;
    mem_wr_strb = 'hx;

    mem_rd_enable = 0;
    mem_rd_addr = 'hx;
    mem_rd_reg = 'hx;
    mem_rd_func = 'hx;

    // act on opcodes
    case (insn_opcode_q)
        // Load Upper Immediate
        OPCODE_LUI: begin
            next_wr = 1;
            next_rd = insn_q[31:12] << 12;
        end
        // Add Upper Immediate to Program Counter
```

```
    OPCODE_AUIPC: begin
        next_wr = 1;
        next_rd = (insn_q[31:12] << 12) + pc;
    end
    // Jump And Link (unconditional jump)
    OPCODE_JAL: begin
        next_wr = 1;
        next_rd = npc - 4;
        npc = ppc + imm_j_sext_q;
        if (npc & 32'b 11) begin
            illinsn = 1;
            npc = npc & ~32'b 11;
        end
    end
    // Jump And Link Register (indirect jump)
    OPCODE_JALR: begin
        case (insn_funct3_q)
            3'b 000 /* JALR */: begin
                next_wr = 1;
                next_rd = npc;
                npc = (rs1_value_q + imm_i_sext_q) & ~32'b 1;
            end
            default: illinsn = 1;
        endcase
        if (npc & 32'b 11) begin
            illinsn = 1;
            npc = npc & ~32'b 11;
        end
    end
```

```
OPCODE_OP_IMM: begin
    casez ({insn_funct7_q, insn_funct3_q})
        10'b zzzzzzz_000 /* ADDI  */: begin next_wr = 1; next_rd = rs1_value_q + imm_i_sext_q; end
        10'b zzzzzzz_010 /* SLTI  */: begin next_wr = 1; next_rd = $signed(rs1_value_q) < $signed(imm_i_sext_q); end
        10'b zzzzzzz_011 /* SLTIU */: begin next_wr = 1; next_rd = rs1_value_q < imm_i_sext_q; end
        10'b zzzzzzz_100 /* XORI  */: begin next_wr = 1; next_rd = rs1_value_q ^ imm_i_sext_q; end
        10'b zzzzzzz_110 /* ORI   */: begin next_wr = 1; next_rd = rs1_value_q | imm_i_sext_q; end
        10'b zzzzzzz_111 /* ANDI  */: begin next_wr = 1; next_rd = rs1_value_q & imm_i_sext_q; end
        10'b 0000000_001 /* SLLI  */: begin next_wr = 1; next_rd = rs1_value_q << insn_q[24:20]; end
        10'b 0000000_101 /* SRLI  */: begin next_wr = 1; next_rd = rs1_value_q >> insn_q[24:20]; end
        10'b 0100000_101 /* SRAI  */: begin next_wr = 1; next_rd = $signed(rs1_value_q) >>> insn_q[24:20]; end
        default: illinsn = 1;
    endcase
end
OPCODE_OP: begin
// ALU instructions: Add, Subtract, Shift Left Logical, Set Left Than, Set Less Than Unsigned, XOR, Shift Right Logical,
// Shift Right Arithmetic, OR, AND
    case ({insn_funct7_q, insn_funct3_q})
        10'b 0000000_000 /* ADD  */: begin next_wr = 1; next_rd = rs1_value_q + rs2_value_q; end
        10'b 0100000_000 /* SUB  */: begin next_wr = 1; next_rd = rs1_value_q - rs2_value_q; end
        10'b 0000000_001 /* SLL  */: begin next_wr = 1; next_rd = rs1_value_q << rs2_value_q[4:0]; end
        10'b 0000000_010 /* SLT  */: begin next_wr = 1; next_rd = $signed(rs1_value_q) < $signed(rs2_value_q); end
        10'b 0000000_011 /* SLTU */: begin next_wr = 1; next_rd = rs1_value_q < rs2_value_q; end
        10'b 0000000_100 /* XOR  */: begin next_wr = 1; next_rd = rs1_value_q ^ rs2_value_q; end
        10'b 0000000_101 /* SRL  */: begin next_wr = 1; next_rd = rs1_value_q >> rs2_value_q[4:0]; end
        10'b 0100000_101 /* SRA  */: begin next_wr = 1; next_rd = $signed(rs1_value_q) >>> rs2_value_q[4:0]; end
        10'b 0000000_110 /* OR   */: begin next_wr = 1; next_rd = rs1_value_q | rs2_value_q; end
        10'b 0000000_111 /* AND  */: begin next_wr = 1; next_rd = rs1_value_q & rs2_value_q; end
        default: illinsn = 1;
    endcase
end
default: illinsn = 1;
dcase
```

```
    // if last cycle was a memory read, then this cycle is the 2nd part of it and imem_data will not be a valid instruction
    if (mem_rd_enable_q) begin
        npc = pc;
        next_wr = 0;
        illinsn = 0;
        mem_rd_enable = 0;
        mem_wr_enable = 0;
    end

    // reset
    if (reset || reset_q) begin
        npc = RESET_ADDR;
        next_wr = 0;
        illinsn = 0;
        mem_rd_enable = 0;
        mem_wr_enable = 0;
    end
end

logic reset_q;
logic [31:0] mem_rdata;
```

```
OPCODE_BRANCH: begin
    case (insn_funct3_q)
        3'b 000 /* BEQ  */: begin if (rs1_value_q == rs2_value_q) npc = ppc + imm_b_sext_q; end
        3'b 001 /* BNE  */: begin if (rs1_value_q != rs2_value_q) npc = ppc + imm_b_sext_q; end
        3'b 100 /* BLT  */: begin if ($signed(rs1_value_q) < $signed(rs2_value_q)) npc = ppc + imm_b_sext_q; end
        3'b 101 /* BGE  */: begin if ($signed(rs1_value_q) >= $signed(rs2_value_q)) npc = ppc + imm_b_sext_q; end
        3'b 110 /* BLTU */: begin if (rs1_value_q < rs2_value_q) npc = ppc + imm_b_sext_q; end
        3'b 111 /* BGEU */: begin if (rs1_value_q >= rs2_value_q) npc = ppc + imm_b_sext_q; end
        default: illinsn = 1;
    endcase
    if (npc & 32'b 11) begin
        illinsn = 1;
        npc = npc & ~32'b 11;
    end
end
// load from memory into rd: Load Byte, Load Halfword, Load Word, Load Byte Unsigned, Load Halfword Unsigned
OPCODE_LOAD: begin
    mem_rd_addr = rs1_value_q + imm_i_sext_q;
    casez ({insn_funct3_q, mem_rd_addr[1:0]})
        5'b 000_zz /* LB  */,
        5'b 001_z0 /* LH  */,
        5'b 010_00 /* LW  */,
        5'b 100_zz /* LBU */,
        5'b 101_z0 /* LHU */: begin
            mem_rd_enable = 1;
            mem_rd_reg = insn_rd_q;
            mem_rd_func = {mem_rd_addr[1:0], insn_funct3_q};
            mem_rd_addr = {mem_rd_addr[31:2], 2'b 00};
        end
        default: illinsn = 1;
    endcase
    npc = pc;
end
// store to memory instructions: Store Byte, Store Halfword, Store Word
```

```
    // mem read functions: Lower and Upper Bytes, signed and unsigned
    always_comb begin
        mem_rdata = dmem_rdata >> (8*mem_rd_func_q[4:3]);
        case (mem_rd_func_q[2:0])
            3'b 000 /* LB  */: begin mem_rdata = $signed(mem_rdata[7:0]); end
            3'b 001 /* LH  */: begin mem_rdata = $signed(mem_rdata[15:0]); end
            3'b 100 /* LBU */: begin mem_rdata = mem_rdata[7:0]; end
            3'b 101 /* LHU */: begin mem_rdata = mem_rdata[15:0]; end
        endcase
    end

    // every cycle
    always @(posedge clock) begin

        reset_q <= reset;
        trapped_q <= trapped;

        if(ppc == 32'h00000034) begin
            $display("Move %d disk %d from %s to %s", regfile[14]-1, regfile[10], regfile[11], regfile[12]);
        end

        // increment pc if possible
        if (!trapped && !reset && !reset_q) begin
            if (illinsn)
                trapped <= 1;

            pc <= npc;

            if(!flush && insn_rd_q == insn_rs1) begin
                hazard_a <= 1;
                hazard_data <= next_rd;
            end else if (!flush && insn_rd_q == insn_rs2 && insn_opcode != OPCODE_OP_IMM) begin
                hazard_b <= 1;
                hazard_data <= next_rd;
            end else begin
                hazard_a <= 0;
                hazard_b <= 0;
            end
```

```
    OPCODE_STORE: begin
        mem_wr_addr = rs1_value_q + imm_s_sext_q;
        casez ({insn_funct3_q, mem_wr_addr[1:0]})
            5'b 000_zz /* SB */,
            5'b 001_z0 /* SH */,
            5'b 010_00 /* SW */: begin
                mem_wr_enable = 1;
                mem_wr_data = rs2_value_q;
                mem_wr_strb = 4'b 1111;
                case (insn_funct3_q)
                    3'b 000 /* SB  */: begin mem_wr_strb = 4'b 0001; end
                    3'b 001 /* SH  */: begin mem_wr_strb = 4'b 0011; end
                    3'b 010 /* SW  */: begin mem_wr_strb = 4'b 1111; end
                endcase
                mem_wr_data = mem_wr_data << (8*mem_wr_addr[1:0]);
                mem_wr_strb = mem_wr_strb << mem_wr_addr[1:0];
                mem_wr_addr = {mem_wr_addr[31:2], 2'b 00};
            end
            default: illinsn = 1;
        endcase
        npc = pc;
    end
```

```
        // update registers from memory or rd (destination)
        if (mem_rd_enable_q || next_wr) begin
            regfile[mem_rd_enable_q ? mem_rd_reg_q : insn_rd_q] <= mem_rd_enable_q ? mem_rdata : next_rd;
        end
        x10 <= regfile[10];
    end

    // reset
    if (reset || reset_q) begin
        pc <= RESET_ADDR - (reset ? 4 : 0);
        trapped <= 0;
    end
  end

ndmodule
```