ECE 411 MP3 Final Report

Group: Latecomers

Yuqi Xue / yuqixue2
Rizhao (Roger) Qiu / rizhaoq2
Zuodong (Nickel) Liang / zuodong2

University of Illinois at Urbana-Champaign

Table of Contents

Introduction

This project is the MP3 assignment of ECE 411 in Fall 2019. The expected output of this project is a 5-stage-pipelined RISC-V processor with split L1 caches and L2 cache. The group members are expected to have a better understanding of computer architecture after completing this project. The project report is divided into several sections, including an overview, deliverables at each project milestone, advanced features of the processor design, and additional contents.

Project Overview

The objective of this project is to build a high-performance RV32i compatible processor. The project is organized into 5 checkpoints. The first 3 checkpoints build a basic 5-stage pipelined processor with all necessary forwarding paths and L1 and L2 caches. Checkpoint 4 implements selected advanced features. Checkpoint 5 optimizes the design for running the provided competition codes. In the last checkpoint, we will observe the trade-offs of different design choices and implement optimizations according to our observations. It is important to acknowledge that several design choices made exclusively for the competition codes are not always the best choices when designing a processor; we will mention this again in the corresponding sections.

Design

Overview

This section includes documentation on the evolution of the processor at each checkpoint, the detailed design of the processor, and notable observations we have made during the development.

Milestones

Checkpoint 1

Overview

In this checkpoint, we developed a basic pipeline structure that supports all basic RV32i instructions without handling any control hazards or data hazards. In addition, the design includes no stall logic so that it can only run with the "magic memory" which has one read-only port and one read/write port that can be accessed simultaneously and always responses in one cycle.

Basic Pipeline Datapath

The basic CPU datapath is shown in the figure below.

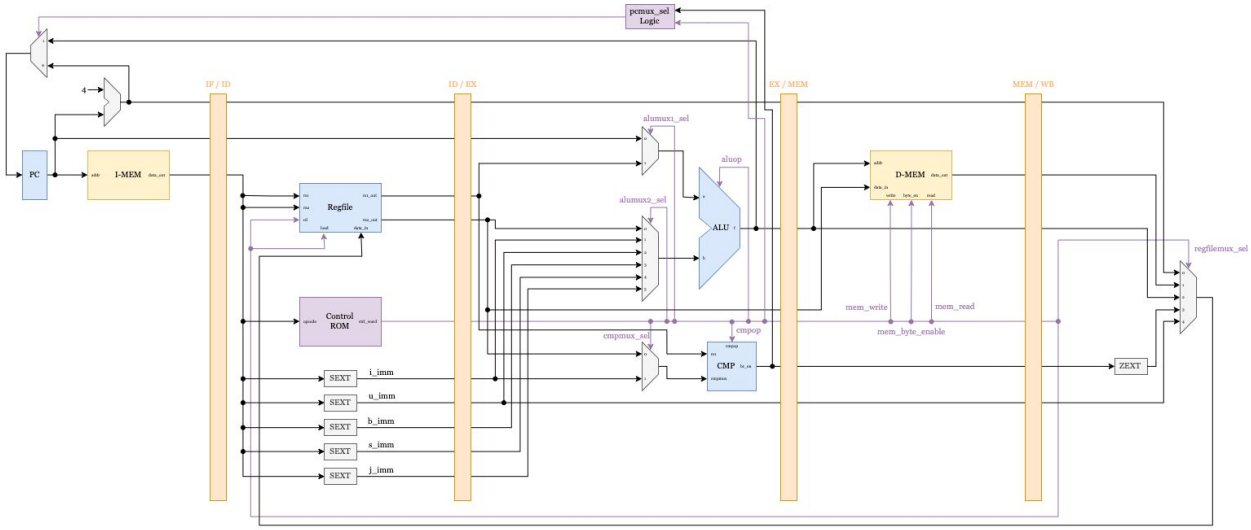

Figure 1. Checkpoint 1 Datapath

By convention, we assume that any program that will run on our processor starts at memory address $0x60$. Hence, the IF stage fetches instructions starting from this address. The output of the instruction memory (later this becomes the I-cache) is then stored in the IF/ID pipeline register. In the next cycle, the fetched instruction is decoded into a control word, register numbers, and immediate values according to the RV32i ISA specifications and these values are placed into the ID/EX pipeline register. The EX stage then performs necessary arithmetic operations and passes the results into the EX/MEM pipeline register.

Control ROM

The control ROM takes opcode, funct3, and func7 as inputs and outputs a control word, which is a set of signals that are immutable for all pipeline stages (and thus can be determined purely by the instruction word itself) for this instruction.

The control word contains the following fields:

```
typedef struct packed {
    // opcodes          # of bits; description
    rv32i_opcode opcode;     // 7; opcode
    alu_ops aluop;           // 3; select ALU operation
    mul_ops mulop;           // 3; select multiplier operation
    branch_funct3_t cmpop;   // 3; select comparator operation

    // muxes signal
    alumux1_sel_t alumux1_sel;  // 1; select between PC or rs1_out
    alumux2_sel_t alumux2_sel;  // 3; select between *_imm or rs2_out
    cmpmux_sel_t cmpmux_sel;    // 1; select between i_imm or rs2_out
    mulmux_sel_t mulmux_sel;    // 1; select input of multiplier
    regfilemux_sel_t regfilemux_sel;    // 4; select write back value
```

```
    // load signal
    logic load_regfile; // 1; load regfile

    // memory signal
    logic dmem_read;     // 1; data memory read
    logic dmem_write;    // 1; data memory write
    rv32i_mem_wmask dmem_wmask; // 4; data memory write mask
} rv32i_control_word;
```

Checkpoint 2

Overview

In this checkpoint, we added the L1 caches, i.e. the instruction cache and the data cache to the processor. We also implemented an arbiter that decides which of the I-cache and the D-cache can have access to the physical memory. In addition, the CPU pipeline now correctly stalls for memory accesses.

CPU Datapath

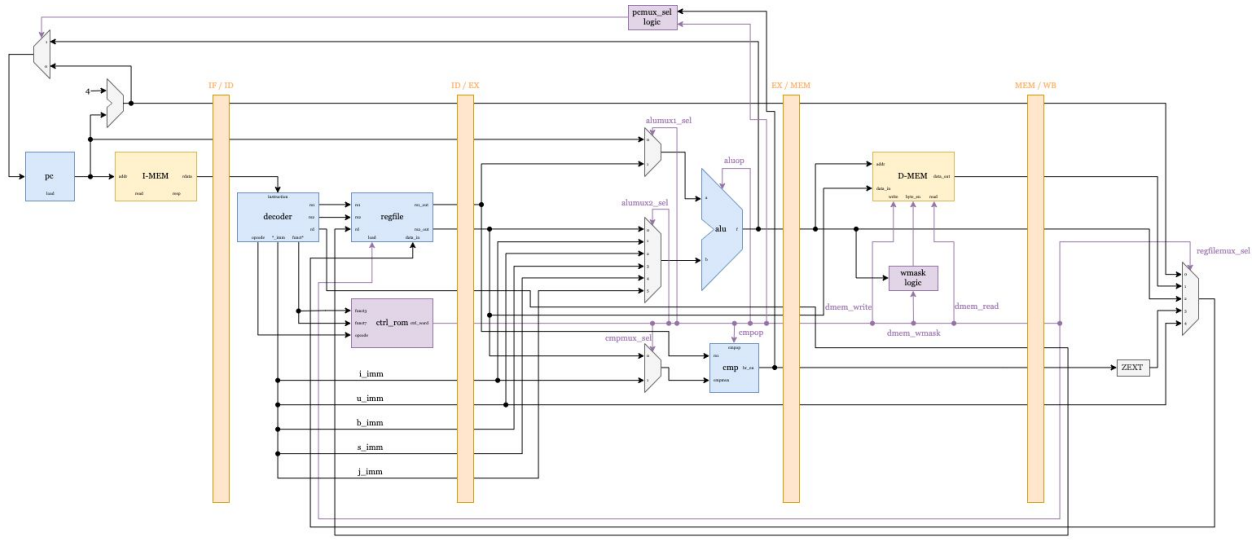

Figure 2. Checkpoint 2 Datapath

Memory Interfaces

In order to easily connect CPU, cache, arbiter, and memory together, we designed two different memory interfaces (implemented as SystemVerilog interfaces). In each interface, there's one server and one requestor. Server provides data when requester requests data. Server and requester have the same data signal, except that the directions are opposite. For read-only interfaces (mem_itf_ro), there are only four signals: read, resp, rdata, and addr. For read-write interfaces (mem_itf_rw), there are three more signals: write, wmask, and wdata.

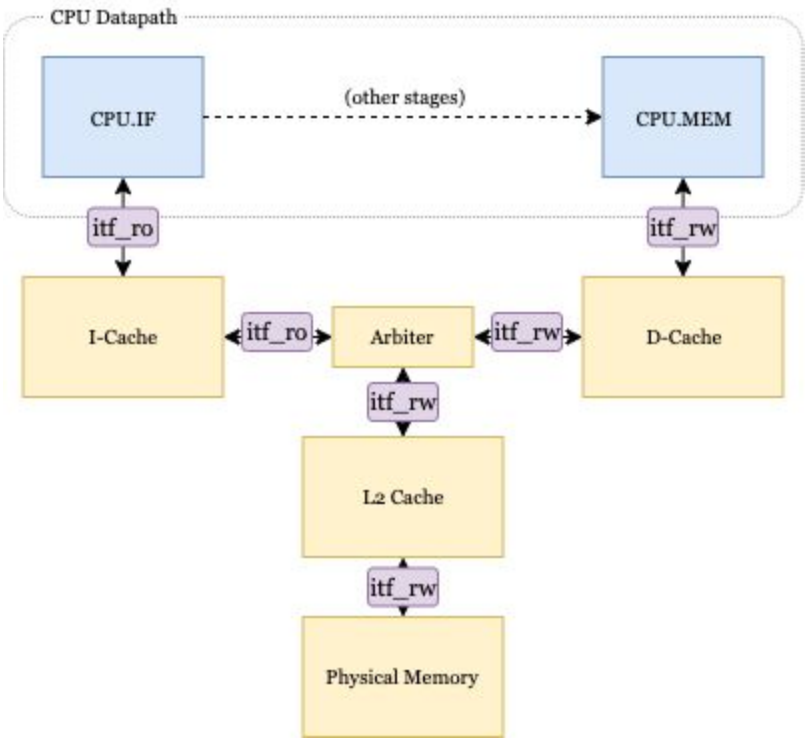

Figure 3. Memory Hierarchy

Data Cache

For the time being, the data cache largely resembles MP2 cache.

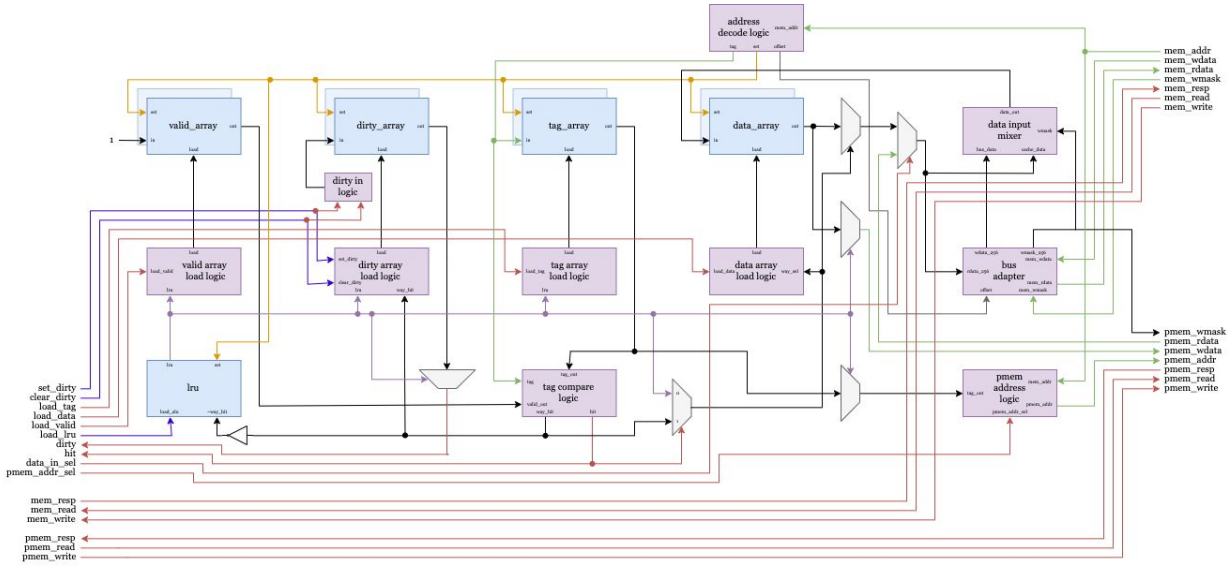

Figure 4. D Cache Datapath

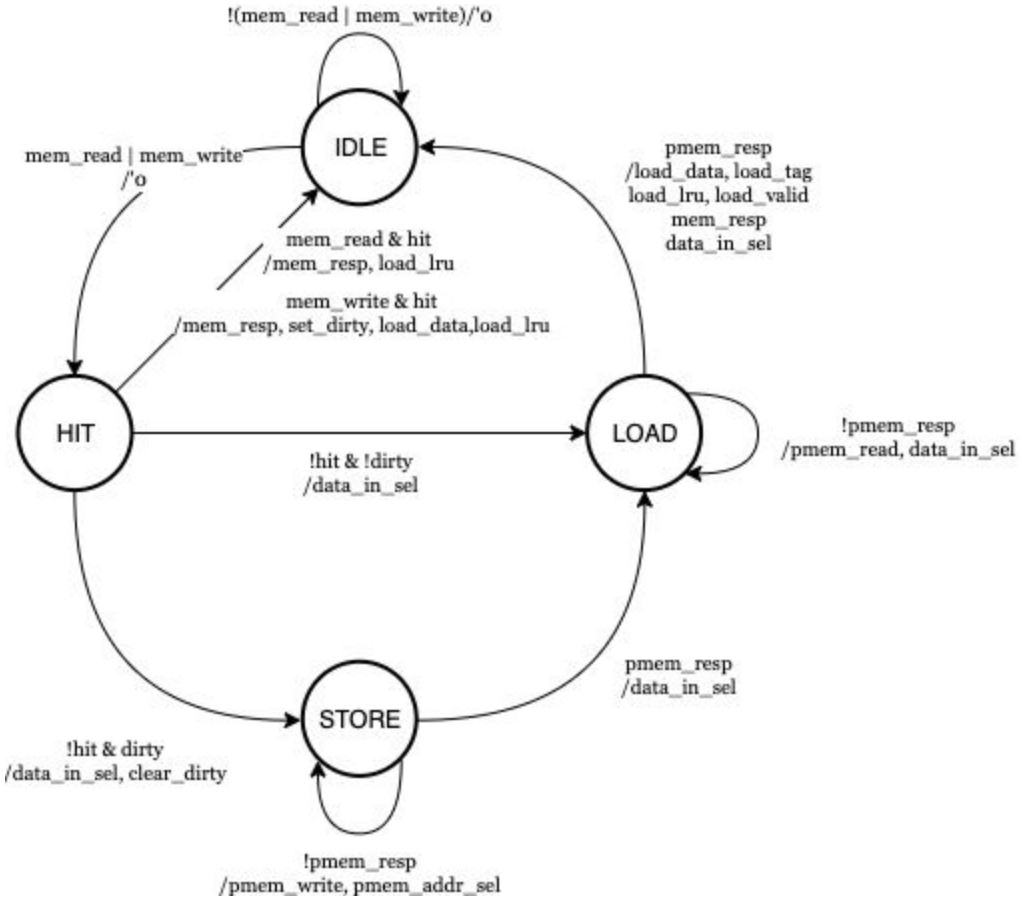

Figure 5. Data Cache State Diagram

Instruction Cache

Compared with the data cache, the instruction cache is read-only and will never write back anything. So the overall datapath and control logic for the instruction cache can be drastically simplified. In the datapath, there's no need for a dirty array to store the information of whether the content in data array has been modified or not. In the control logic, since we will never write back anything, there's no STORE state. Since instructions are usually accessed sequentially, we implemented a one-cycle read hit state machine. On a memory read, the state machine will stay in the HIT state until there is a miss. This prevents stalling the pipeline for one cycle doing tag compare for every instruction fetch.

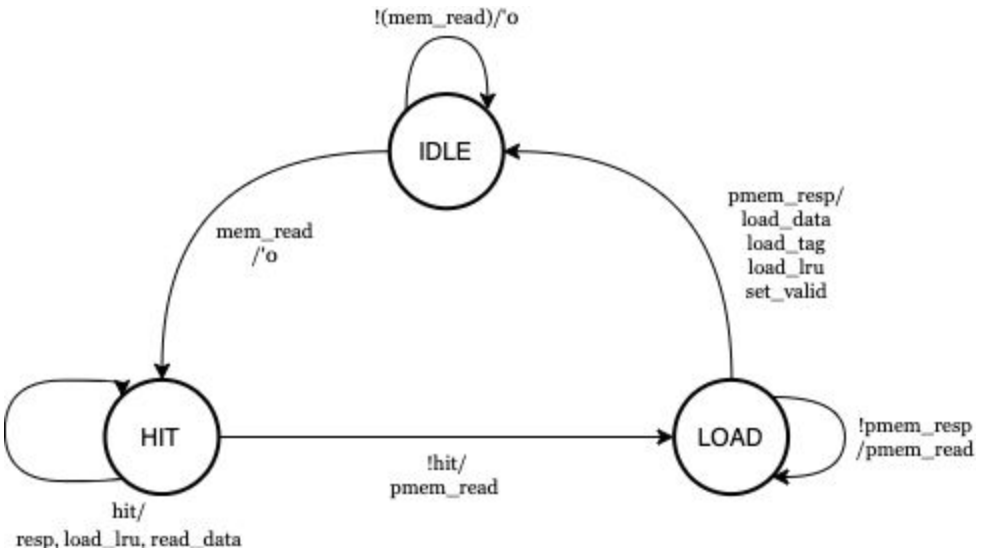

Figure 6. Instruction Cache State Diagram

Arbiter

The arbiter is an interface between L1 caches and L2 cache (anyway, the physical memory only has one data bus). Since we don't have a banked L2 cache, only one memory request can be served at a time by the L2 cache. Hence, we need to have the arbiter to decide which of I-cache and D-cache has access to L2 cache. Our design gives I-cache priority over D-cache in case we want to implement memory stage leapfrogging in the future (which did not happen).

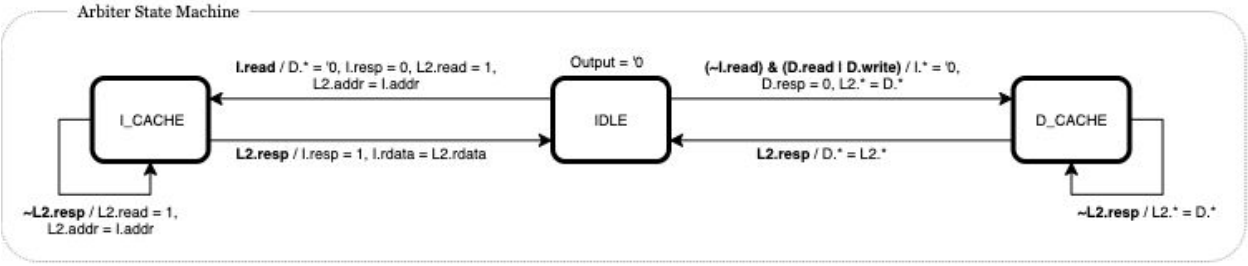

Figure 7. Arbiter State Diagram

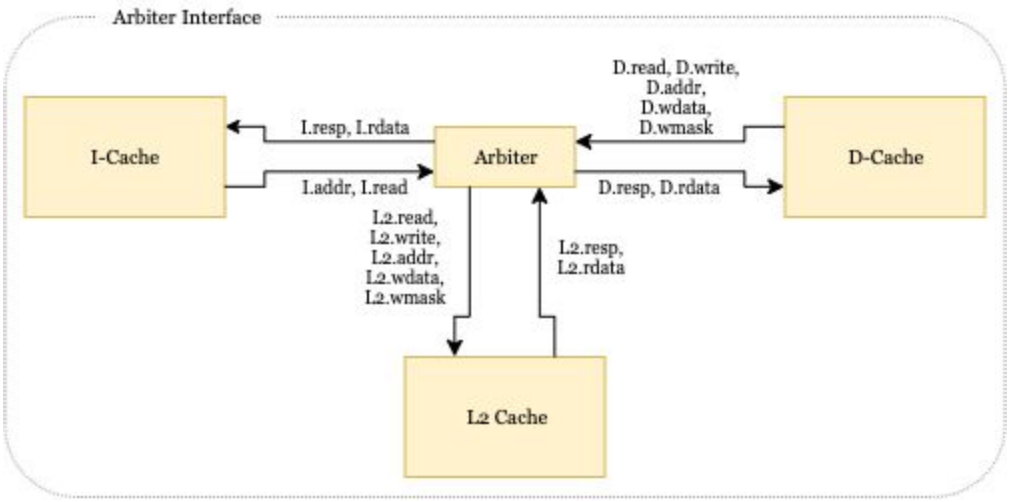

Figure 8. Arbiter Interface

Checkpoint 3

Overview

In this checkpoint, we implemented a fully functioning pipeline datapath with all necessary forwarding logic, flushing logic, stalling logic, and a static-not-taken branch predictor. In addition, we added an L2 cache into our design.

CPU Datapath

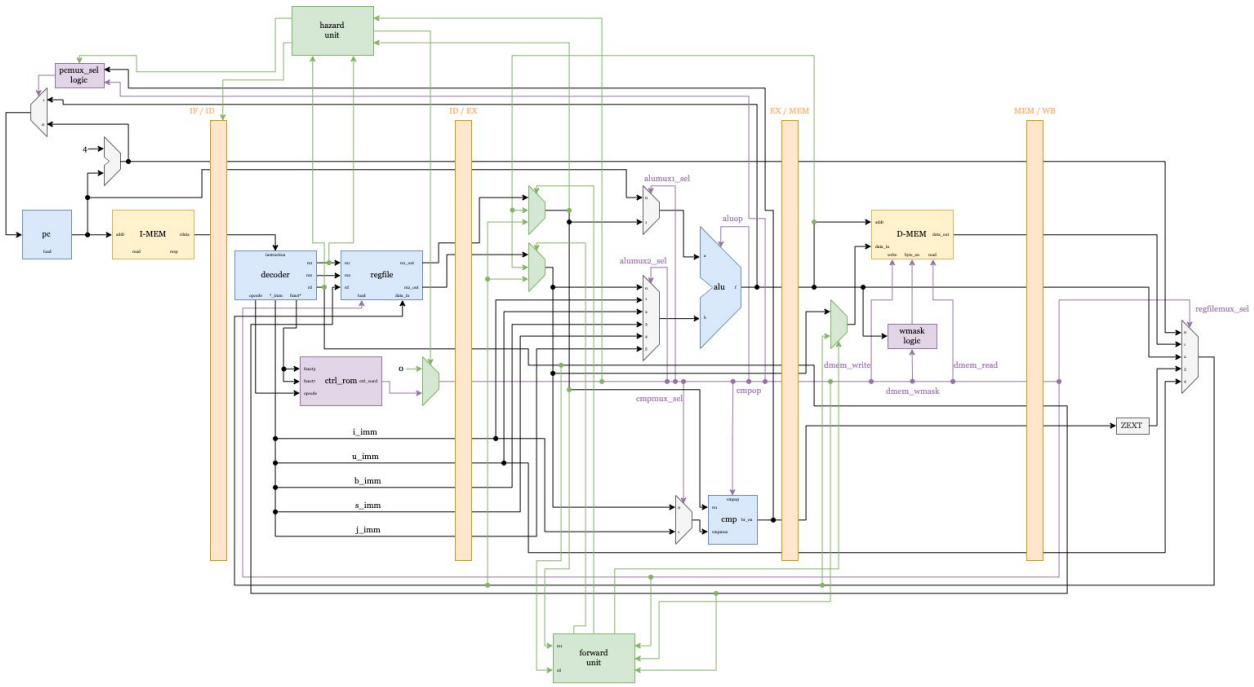

Figure 9. Checkpoint 3 Datapath

L2 Cache

For the time being, the L2 cache is a direct copy-and-paste of the L1 data cache. The only difference is that the data array in L2 cache does not support write mask since it doesn't need one. We would have a more sophisticated design in later checkpoints.

Hazard Detection & Forwarding

Stalling

The stalling logic was already implemented in CP2. If the stall signal is high, then the load signals to pipeline registers will be turned off so that data is held in the datapath.

Flushing

To flush the pipeline, we implemented a few flushing MUXes. The flush signal is generated by a flushing detector in the EXE stage. When there is a branch misprediction (In CP3, a branch prediction simply means a taken branch), the detector issues the flush signal. This effectively changes the input to IF/ID register and ID/EXE register to be NOP so that no unwanted instructions would be executed.

Forwarding

Forwarding logic was implemented with forwarding MUXes and a forwarding detector. The forwarding detector sniffs ID, EXE and MEM stage to see if there is a data dependency. For example, if the instruction in the EXE stage reads from the x5 register and the instruction in the MEM stage is loading data into the x5 register, then the forwarding unit would forward the data from MEM stage to EXE stage to avoid stalling the pipeline.

An important change in design was introduced here - to make forwarding easier, we moved the regfilemux from the WB stage to the MEM stage so that we would not need any extra comparator in the forwarding detection unit.

Static Branch Prediction

In the CP3 design, we adopted a static-not-taken branch prediction strategy. This is done by flushing the pipeline if and only if there is a taken branch in the EXE stage.

Advanced Design Options

Branch Predictor

Overview

We implemented a GAs predictor. This is a 2-level branch predictor that consists of a Global Branch History Register and a Pattern History Table. The GBHR is essentially a shift register that holds

some past branching histories. The PHT is a table where each entry is a 2-bit saturating counter and is indexed by combining some bits from the PC register and some bits from GBHR. In this way, the predictor can learn correlations between branch instructions at different memory locations from recent global branching histories and local branching histories at each memory address.

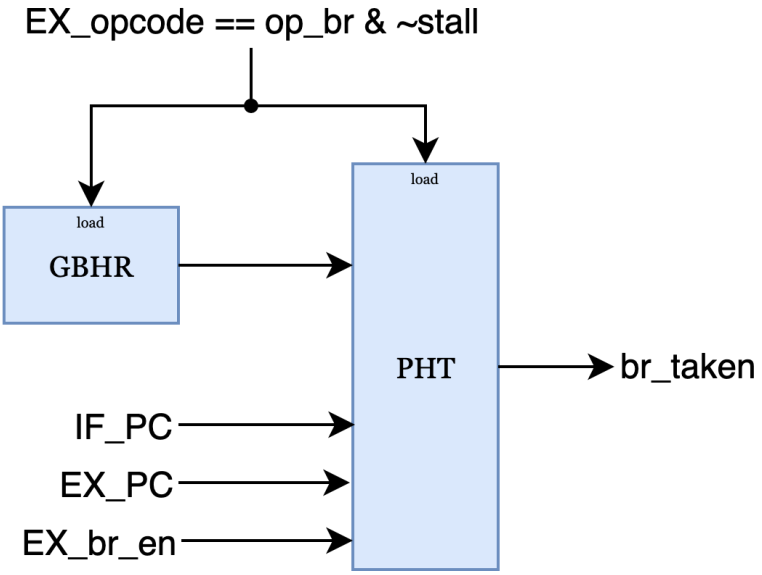

Figure 10. Datapath of the branch predictor.

Bimodal Predictor

Our 2-level GAs predictor is an extension of the simpler bimodal predictor, which is also called a 2-bit saturating counter that is equivalent to the following state machine:

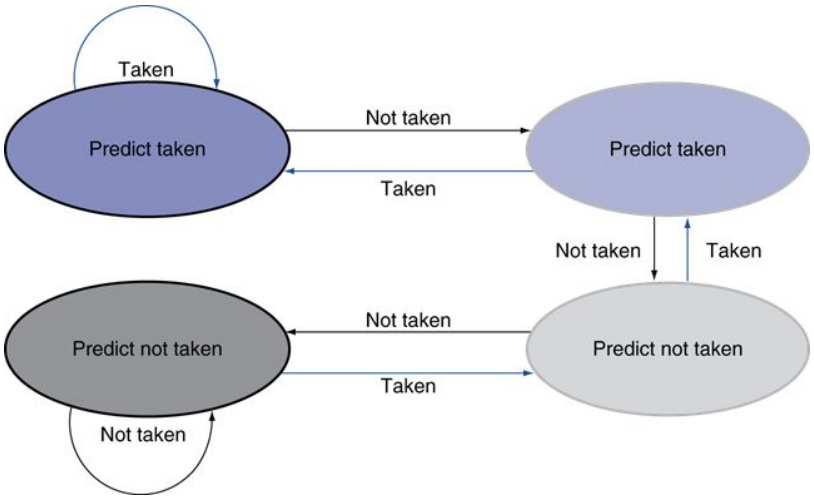

Figure 11. A 2-bit saturating counter.

Generally, we can consider an N-bit saturating counter as a state machine that has 2^N states with similar state transitions and outputs. If there are too few states - for example, a 1-bit saturating counter, - then the prediction might be inaccurate. If there are too many states, then it's hard to determine which initial state is the best and the predictor will need a lot of warm-up time. By some

heuristics and some experiments, we decided to use a 2-bit saturating counter with initial state “00” (strongly not taken).

Global Branch History Register

The GBHR is a shift register that shifts one bit to the left per clock cycle when the load signal is high. The serial input of this register is the actual br_en signal from the EX stage, and it is loaded if and only if there is a branch instruction in the EX stage. The output of this register is used as part of the index to the PHT.

Pattern History Table

The PHT is essentially an array of saturating counters. The array can be viewed as a 2-D table where each row corresponds to different memory locations and each column corresponds to different branching histories (i.e. “patterns”). In practice, some bits from PC are used as the row index and the value of GBHR is used as the column index to access the PHT. Therefore, an entry in the PHT gives the prediction of the output of a branch instruction at a certain memory location with certain patterns before this instruction.

Due to the limited size of the PHT, there are aliasing issues for branches at different memory locations. This problem can be eased by increasing the size of PHT but cannot be solved completely.

Module Parameterization

Our implementation can be easily parameterized to enhance development experience. Specifically, we parameterized the number of bits of the saturating counter, the number of bits taken from the PC register, and the number of bits in the GBHR. These parameters also determine the size of the PHT. However, we still need to configure the bits from PC manually every time we set a new parameter because we might want to take discontinuous sections of bits from PC.

In practice, we chose to use 3 bits for GBHR and 5 bits from PC (which are {PC[7:6], PC[4:2]}). Hence, there are 8 bits to index the PHT and the PHT contains 256 entries each of which is a 2-bit saturating counter. The overall storage needed by the branch predictor is thus about 512 bits (64 bytes).

Integration into the CPU Datapath

The branch predictor is placed in the IF stage in the CPU datapath. Upon a branch opcode detected in the IF stage, the branch predictor makes a prediction about that branch instruction. Meanwhile, the b_imm bits are extracted from the instruction and added to the PC value to produce the branch target address. If the prediction is correct, then there will be no penalty and the pipeline does not stall due to branching (except a possible miss in I-cache which induces a memory stall). If in the EX stage a misprediction is detected when resolving a branch instruction, the IF stage and the ID stage are flushed and the correct branch target address or the original pc_plus4 value will be loaded into PC. The misprediction penalty with this design is 2 cycles.

The GBHR and the PHT are updated during the EX stage by the true branching results (not the predicted results) and the corresponding PC value in the EX stage (which comes from the ID/EX pipeline register). They are updated per every branch instruction in the EX stage. The stall signal is given to ensure only one write operation is performed to the GBHR and the PHT when the pipeline stalls.

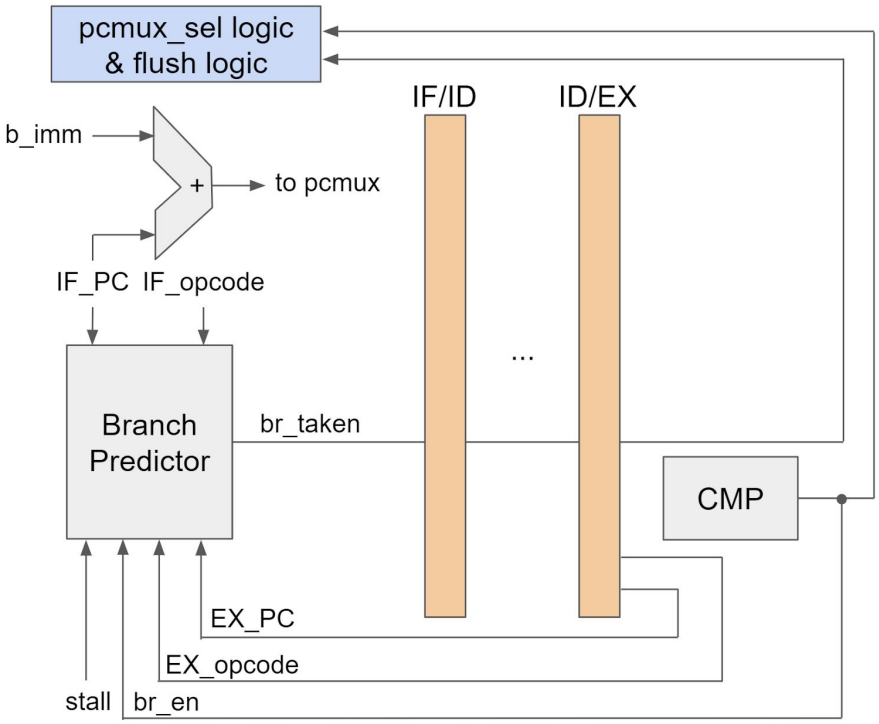

Figure 12. Branch predictor in the CPU datapath.

Performance

We measured the accuracy of our branch predictor running the competition codes against a static-not-taken predictor and a static-taken predictor ("hit" refers to correct prediction, and "miss" refers to misprediction):

	Program	**# of Hits**	**# of Misses**	**Accuracy**
Static Not Taken	comp1_im.s	287	827	25.76%
	comp2_im.s	118	410	22.35%
	comp3_im.s	39	1200	3.15%
Static Taken	comp1_im.s	827	287	74.24%
	comp2_im.s	410	118	77.65%

	comp3_im.s	1200	39	96.85%
GAs Predictor (Our predictor)	comp1_im.s	815	299	73.16%
	comp2_im.s	445	83	84.28%
	comp3_im.s	1178	61	95.08%

Table 1. Statistics about our branch predictor running the competition codes.

We observed that our implementation performs even worse than the static-taken predictor in some programs. After trying some other programs, we confirmed that this is an occasional phenomenon possibly caused by the particularity of competition codes. In other words, we did find many other programs for which our branch predictor has better performance (for example, "comp2_im.s" and "mp3-final.s").

Parameterized Cache

Overview

We implemented a parameterized L1 Data Cache, L1 Instruction Cache, and L2 Cache. In our parameterized cache, a single configuration file can change the number of sets and associativity of each cache. In order to achieve this, we designed a parameterized Binary Tree Pseudo Least Recently Used (LRU) unit and integrated it into all caches. Together with dedicated miss/hit counters in the testbench code and help from Makefile-based autotest, we were able to easily adjust parameters and get the benchmark of different configurations in the command line.

Parameterized Binary Tree Pseudo LRU

In order to track the usage of different cache ways and efficiently replace unused lines inside the cache, we choose the Least-Recently-Used (LRU) method as our replacement policy for all of our caches. However, precise LRU tacking will be increasingly expensive to implement with the increase of cache associativity. Hence, we decided to implement a binary tree pseudo LRU. With binary tree pseudo LRU, we can track S-way cache with only S-1 bits. Within these S-1 bits, data are hierarchically divided into a binary tree. At each level of the tree, one bit is used to track the least recently used way. On a cache hit, based on the most-recently-used (MRU) way, we can update the binary tree for a set. On a cache miss, we can use the bits stored in the binary tree to trace the least recently used way and replace it with new data.

As we want to design a parameterized cache, the LRU itself must be parameterized as well.
However, with binary tree pseudo LRU, we find it difficult to implement a non-power-of-2 way LRU. Hence, all our caches need to have an associativity of a power of 2.

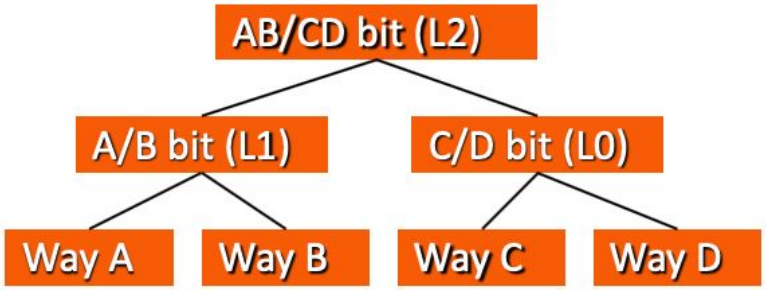

Figure 13. A Four-way Binary Tree Pseudo LRU Structure

In order to build this binary tree pseudo LRU, we designed it in a way that the input is the most-recently-used way, and the output is the least-recently-used way. Inside the LRU, it keeps the record of the binary tree structure. For any MRU input, it first converts the compact form LRU input into extended binary tree input format. Then, MRU generates a load signal that is also in extended binary tree format. On a clock rising edge, the internal binary tree record is updated based on the load signal and the new extended binary tree input. The last step for the LRU is to output the compact LRU ways based on the binary tree input format.

Parameterized Cache

With parameterized LRU ready, parameterizing the cache will be really simple. In our prior implementation, there were only two cache ways in each of I, D, and L2 caches. On a hit the cache will output data from the data array, the select signal can be easily parameterized with appropriate tag compare logic. On a cache miss, dirty signal and data array can also be easily selected with a tag compare logic.

Performance

We tested our parameterized cache and compared the finish time of each competition code to determine the final parameter. The baseline for our test is 2 ways, 8 sets for all caches.

L2 W/S	D W/S	I W/S	Comp1/ns	Comp2/ns	Comp3/ns
2/8	2/8	2/8	217665	76955	694965
4/8	2/8	2/8	0	-1870	0
2/16	2/8	2/8	0	-4320	-260
4/16	2/8	2/8	0	-4800	-260
2/8	2/16	2/16	0	-4800	0
2/8	4/8	4/8	0	-4800	-300

2/16	2/16	2/16	0	-4800	-260
2/16	4/8	4/8	0	-4800	-300
4/16	4/8	4/8	0	-4800	-300

Table 2. Competition Code Runtime with Different Cache Parameter. Less is better.

Eventually, we decided to use 2 ways, 8 sets for the L2 cache and 4 ways, 8 sets for both I and D cache. With more cache ways and sets the competition code performance did not have any improvement, so to save logic and area we go with the size described above.

Eviction Write Buffer

Overview

The EWB is a single-line cache placed between the L2 cache and the physical memory. It is meant to hold a dirty evicted block from the L2 cache and write back the evicted block to the physical memory and allow the subsequently missed address to be served first and, when the physical memory is free, perform the write-back of the dirty block. This allows the CPU to get the missed data faster instead of waiting for the dirty block to be written first.

The detailed design of EWB includes a sophisticated state machine and a simple datapath. Notably, this design requires the upper level (L2 cache) to hold the input memory address when asserting the read or write signal.

Controller

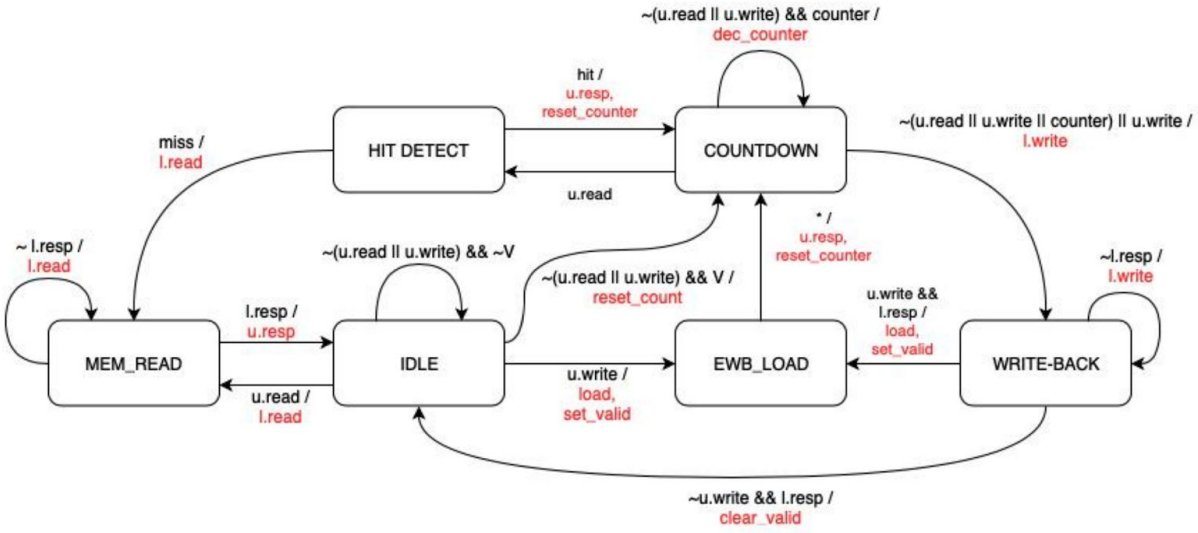

Figure 14. State machine of the EWB

Datapath

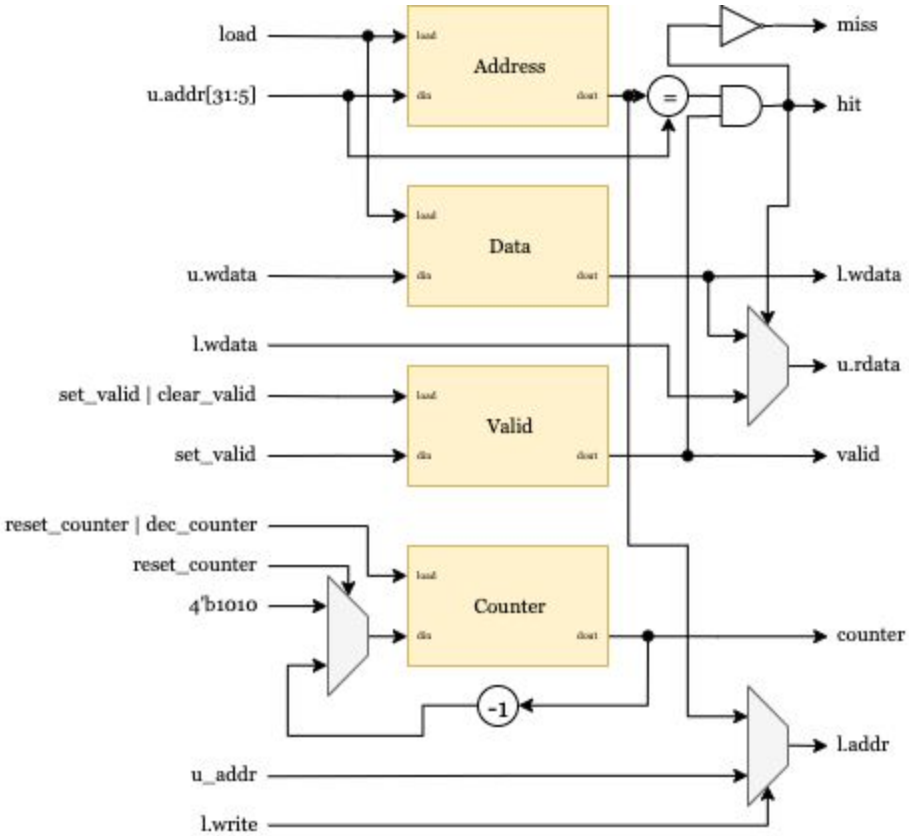

Figure 15. Datapath of the EWB

Performance

While the EWB provides a significant performance boost for some programs (for example, "mp3-final.s") that induce a lot of memory writebacks, we observed that it provides no benefits at all for competition codes after we tuned our L1 and L2 caches as described above. Particularly, within the L1 and L2 cache parameters we have set, there are no evictions in the L2 cache at all and hence no write operations to the physical memory at all! Hence, we decided to remove EWB in order to increase FMAX for running the competition code.

M-Extension

Overview

In order to support the RV32i "M" extension, we experimented with add-shift multiplier and Wallace tree multiplier for multiplication and implemented a restoring divider for division.

Multiplier - Fully Combinational Wallace Tree Multiplier

We modified the add-shift multiplier given in MP0 to be a 32-bit multiplier that runs in 32 states, but eventually we implemented a pipelined Wallace multiplier to increase the performance of the processor.

A lot of time was spent on optimizing the design with Wallace multiplier before the fifth checkpoint. Initially, we designed, implemented, and verified a fully functional combinational Wallace multiplier. Then we plugged it right next to the ALU and introduced a new MUX to select output from either the original ALU or the multiplier. However, it caused a tremendous amount of decrease in the FMAX of our design - we got an FMAX of about 12.95 MHz. We used the Quartus timing analyzer to analyze the failing path and multiplier was indeed the component causing most of the latency. Besides other optimizations that were done on other components, the most important performance gain came from pipelining the multiplier.

At first, we attempted to pipeline the Wallace multiplier into 4 stages by placing pipeline registers in between different layers of the Wallace multiplier. After integrating and verifying the multiplier into the CPU datapath, we found out that although the processor design was correct, the FMAX only increased by a diminishing amount. After a more careful critical path analysis, it turned out that the latency of intuitively pipelined Wallace multiplier is not roughly divided by the number of stages it's split into! Rather, the propagation delay of each stage is reduced only by a small constant amount (if it's split into n stages, the latency is reduced by the propagation delay of n full adders rather than divided by n).

Therefore, in our final design, we chose to simply add two pipeline registers for the input and output of the Wallace multiplier, respectively. This effectively isolates the propagation delay introduced by hundreds of adders in the Wallace multiplier from the rest of the processor datapath.

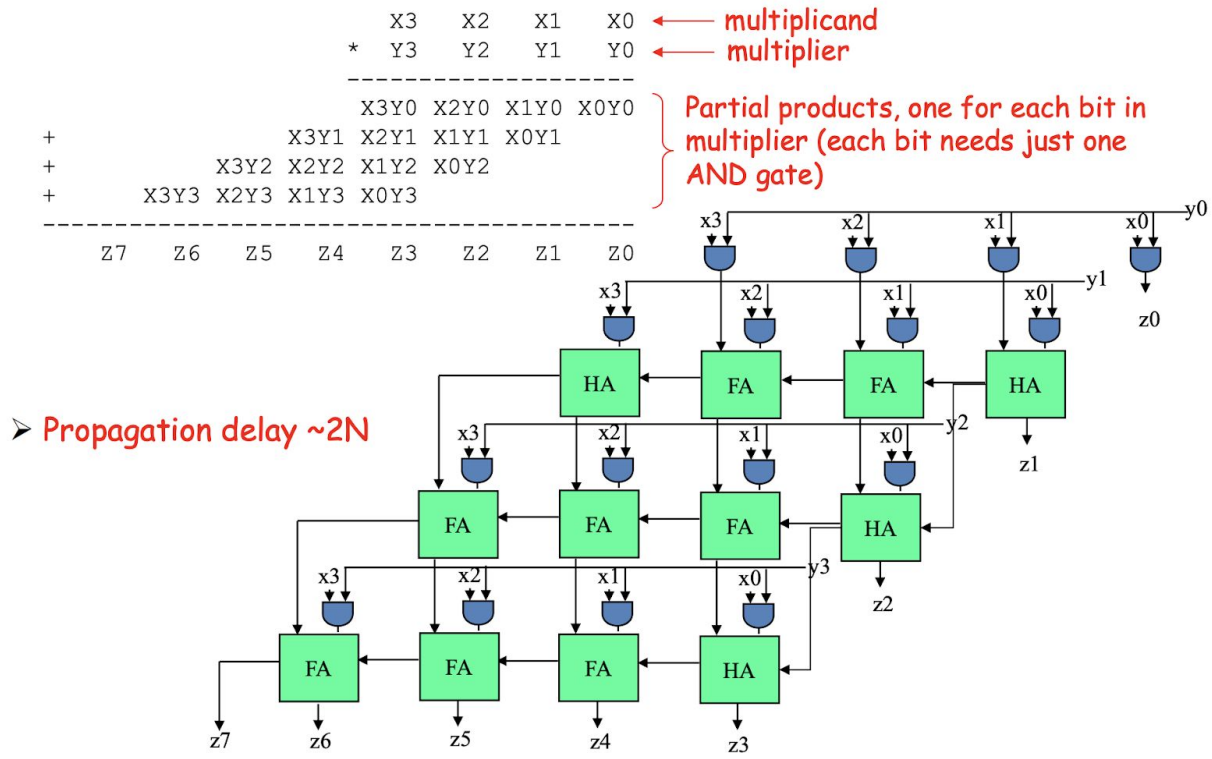

Figure 16. A 4-bit unsigned multiplier. Our design is a 32bit unsigned multiplier and is messy to draw.

Divider - Restoring Division

As for the divider, since there is no div or rem instruction in the competition code, we did not spend time on implementing an efficient divider. Instead, we designed a restoring divider. Restoring divider basically works by trial and error - it subtracts the current number (obtained by shifting bits from divisor) from the dividend, and then tests if the resulting number is negative to determine if the current bit should be included. The design and implementation of the restoring divider are very similar to that of an add-shift multiplier.

Test Environment

Verification with RISC-V Official Test Code

After forwarding and stalling are implemented, we adapted the test code provided by the RISC-V official repository (https://github.com/riscv/riscv-tests/tree/master/isa/rv32ui) to verify our design. The comprehensive test code is long, which enables us to test the stability of our design in a fashion similar to the MP2 long test code. Moreover, it covers almost all edge cases of the instruction. For example, it tests unaligned sb/sh instructions and tests the behavior of div/rem instructions when division by zero error is encountered.

The official test code is a part of the larger RISC-V toolchain. Therefore, in order to use it in our design, we spent some time to separate the useful part of the test code. The fence instruction is a good example of the test code that we stripped away. We also included a header file to pre-process the test code file so that it can be written in a more human-readable way. This made it easier to be maintained. Slight modifications were also done in order to make it compatible with the RISC-V tools we have so that it can be compiled and run with ModelSim.

Makefile

To facilitate the development of the processor, we also wrote a Makefile so that we can verify our design without opening up GUI. We took the .do script file generated by Quartus for Modelsim and modified it such that the script recompiles the design every time we run it. Also, we studied the command line interface of ModelSim so that the simulation process can be invoked from the command line automatically.

With the Makefile, the workflow of verification is as follows:

1. User types in 'make all' and hits enter
2. Makefile invokes a Python script we wrote to process the input file.
3. Then, the C preprocessor cpp is called to pre-process the provided file with the header.
4. After this, the file fully consists of RISC-V assembly and is fed to the load_memory.sh script to generate a ModelSim-compatible memory file.
5. The Makefile then copies the memory.lst file into the simulation folder and invokes Modelsim from the command line to run the simulation.
6. Our testbench detects if the CPU is in an infinite loop (by sniffing the PC reg and branching indicator). If so, the testbench would dump various numbers (if any, failing test case number, values in all registers, cache hit rate, branch accuracy and so on) to print them out to the command line.
7. If the user wants to run the code in Spike for debugging, he or she can simply type "make spike" after the Makefile terminates. The script will automatically compile the most recent assembly code and feed it to Spike.

```
# [Reg 29]: 0x00000000
# [Reg 30]: 0x00000000
# [Reg 31]: 0x000003c8
# Gathering branching accuracy
# Total number of OP_BR:        2015; MISSES:         676; HIT:        1339
# Branch predictor accuracy: 0.664516
# Total cycles elapsed:      50590
# L2 Cache total hit 1185, total miss 1185
# L2_Hit[0] = 592
# L2_Hit[1] = 593
# D Cache total hit 776, total miss 5
# D_Hit[0] = 0
# D_Hit[1] = 0
# D_Hit[2] = 0
# D_Hit[3] = 776
# I Cache total hit 14039, total miss 1180
# I_Hit[0] = 111
# I_Hit[1] = 6785
# I_Hit[2] = 3497
# I_Hit[3] = 3646
# All tests passed! Good job!
# ** Note: $finish    : /home/rizhaoq2/Latecomers/hvl/full_tb.sv(56)
#    Time: 506205 ns  Iteration: 1  Instance: /full_tb
# End time: 02:00:19 on Dec 11,2019, Elapsed time: 0:00:19
# Errors: 0, Warnings: 0
```

Figure 17. A screenshot of Our Makefile Command-Line Verification in Action.

Additional Remarks

Further Improvements

More Advanced Features

According to our observations on the competition codes, it would be beneficial to implement hardware prefetching in the L2 cache. This would also benefit other programs in general, since in most of the time the CPU pipeline is making requests to the L1 caches, creating an idle period for the L2 cache.

Pipelined Caches

The critical paths reported by the Quartus Timing Analyzer indicate that our caches are the bottleneck of achieving a higher FMAX. Moreover, while I-cache can respond to continuous hits without stalling, D-cache and L2 cache still needs a dedicated cycle to do tag compare. If we have all caches pipelined, not only our design runs at a higher frequency, but also our caches save a lot of extra cycles of doing tag compare and stalling the CPU pipeline.

Testing

Although we have a helpful Makefile and the RISC-V official test codes, it is still hard to find some very subtle errors. These errors may exist even though the program running on our processor has the correct outputs. To ease this pain, it will be extremely helpful to adopt the RISC-V monitor in

our testbenches. This monitor checks the internal states of the processor and compares them against the RV32i standard. This would save a lot of work for us in front of the waveform in Modelsim.

Conclusion

At the end of the semester, we were able to build a 5-stage-pipelined RISC-V processor with M extension, parameterized split L1 caches and L2 caches, branch predictor, and EWB. All of our group members have actively participated in the design and testing process of our processor, and all of us have a better understanding of computer architecture after completing this project.

Appendices

Complete Design Hierarchy

This is the overall design hierarchy of our processor.

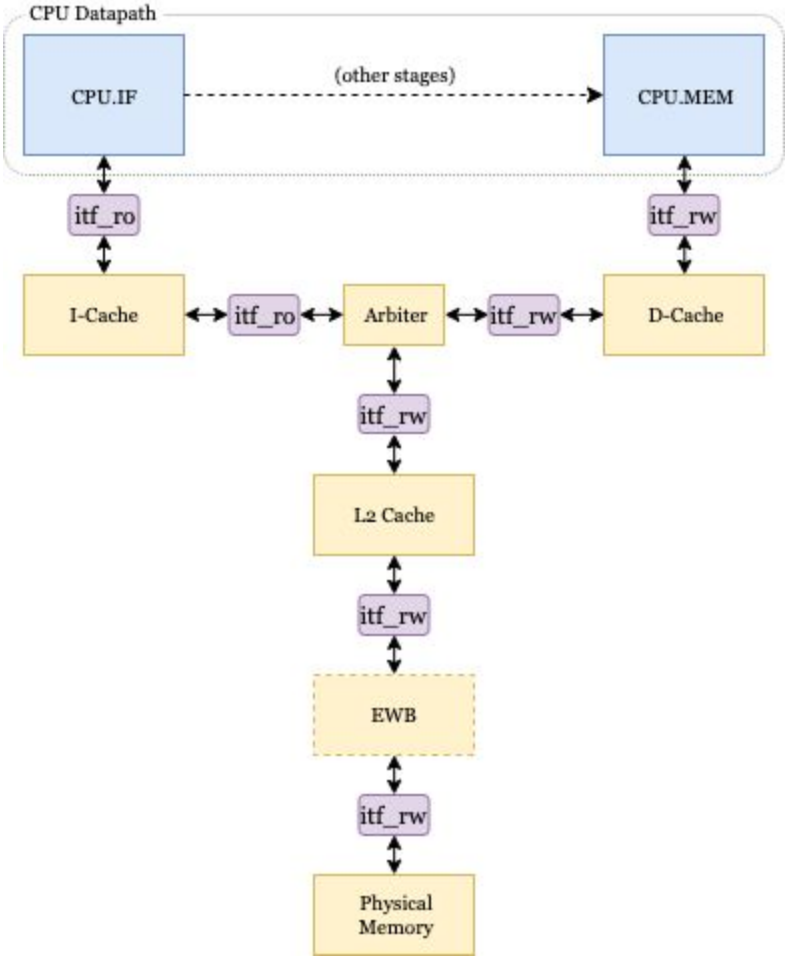

Figure A. Overall Design Hierarchy.

Complete CPU Datapath

This is the complete CPU datapath that includes most of the details in the pipeline structure.

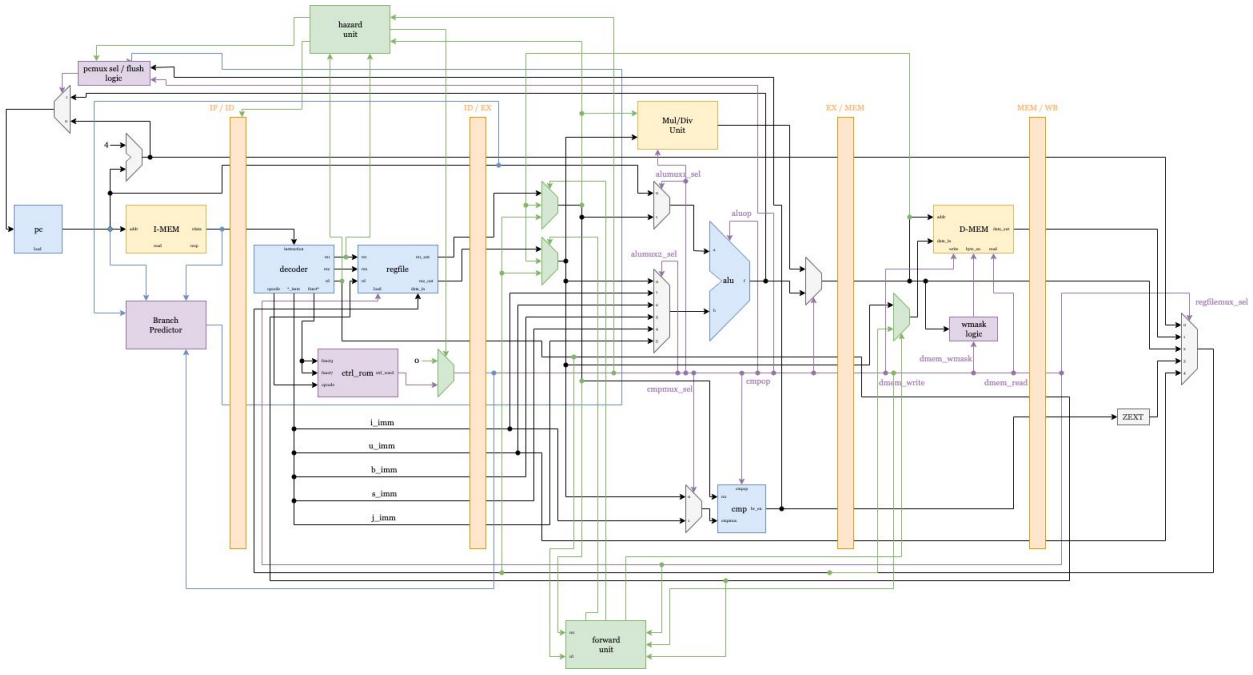

Figure B. Complete CPU Datapath.

Quartus Chip Planner Result

This is the output of the Quartus chip planner, which shows the physical layout of our design on the FPGA board. Green represent CPU datapath, black represent data cache, pink represent instruction cache, and blue represent L2 cache. This is the design used for the competition (which is why we have larger L1 caches than L2 cache).

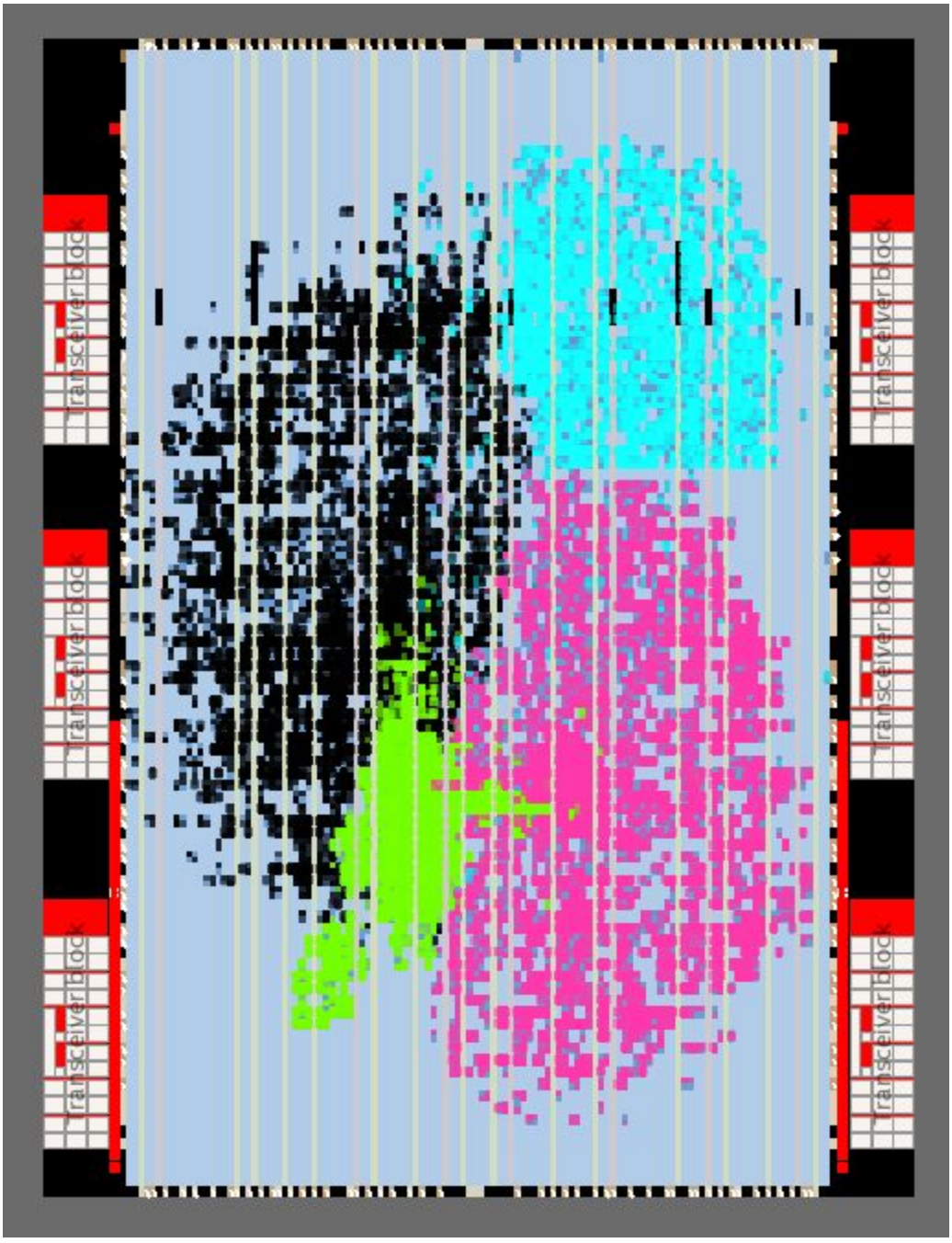

Figure C. Chip Planner Output.