MP4 Checkpoint 2 Deliverables

Progress Report, Roadmap, Paper Designs

November 20th, 2020

Alex Vetsavong
Peter Kircher
Mohan Li

Responsibilities: What did people do for this checkpoint?

Arbiter and split-cache hierarchy:
Peter implemented the arbiter and attached the cache hierarchy to the design. The group debugged an issue with the same-cycle cache response.

Hazard Detection and Forwarding:
Mohan added the muxes, the basic structure of data forwarding, and detection of data hazards. Alex worked on branch predictions, added flushing and stalling signals, and made changes to the data forwarding code. The group as a whole worked on debugging the stalling and flushing process.

Testing and Debugging:
Alex attached the DUT to shadow memory, initially the RVFI monitor, and loaded the testcode. The group as a whole debugged waveforms and found solutions to our problems.

Changes from CP1:
Moved the regfilemux from WB to MEM stage to facilitate MEM -> EX forwarding in an easier way. Moved away from the magic memory interface to accommodate the cache line adaptor and its interaction with burst memory.

Roadmap: Who's doing what?

Alex:
Hardware Prefetching: will be working to implement a stream prefetcher with two buffers -- one for I-cache and one for D-cache.

Mohan:
RISC-V M-Extension: will implement a Wallace Tree Multiplier, and a division module. Will also make changes in datapath and control_rom to handle the 8 new operations.

Peter:
Peter will be working on a unified L2 cache, which will sit between the L1 caches and the cacheline adapter.
Peter will also be adding ways to the L1 caches, making them 4-way SA L1 caches for instructions and data.

RISC-V M-Extension

Problem Statement

4 Multiplication Operations:

- MUL: Performs 32-bit * 32bit multiplication, places the lower 32 bits of the result into rd.
- MULH: Signed-signed multiplication, stores the upper 32 bits of the result into rd.
- MULHU: Unsigned-unsigned multiplication, stores the upper 32 bits of the result into rd.
- MULHSU: Signed-unsigned multiplication, stores the upper 32 bits of the result into rd.

4 Division Operations:

- DIV: Signed integer division.
- DIVU: Unsigned integer division.
- REM: Remainder of the signed integer division.
- REMU: Remainder of the unsigned integer division.

RV32M Standard Extension

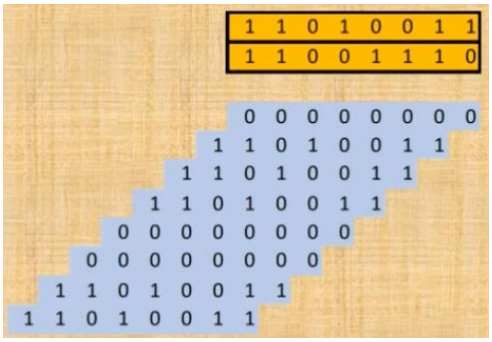
0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

How to perform multiplication?

We are going to add a new component named multiplier into the EX stage (keep it separate from ALU for now). But we need to keep in mind that data forwarding will affect the multiplier as well. That means that the multiplier may connect to ALUMUX3 and ALUMUX4.

Implementation of Wallace Multiplier

First, the multiplier is going to compute the **partial products** of the inputs using 32*32 AND gates. The example diagram below shows the result of stage 1 in 8-bit multiplication.



Next, the Wallace multiplier seeks to reduce any 3 rows of partial products into 2 rows. This is done by (1) partitioning the 32 rows into groups for 3, (2) in parallel, compute the sum of each group using full adders and half adders.
For 32-bit multiplications, initially there are 32 rows of partial products, which can be grouped to 10 * (group of 3 rows) + (group of 2 rows)
=> 22 rows after reduction => 7 * (group of 3 rows) + (group of 1 row)
=> 15 rows after reduction => 5 * (group of 3 rows)
=> 10 rows after reduction => 3 * (group of 3 rows) + (group of 1 row)
=> 7 rows after reduction => 2 * (group of 3 rows) + (group of 1 row)
=> 5 rows after reduction => 1 * (group of 3 rows) + (group of 2 rows)
=> 4 rows after reduction => 1 * (group of 3 rows) + (group of 1 row)
=> 3 rows after reduction => 1 * (group of 3 rows)
=> 2 rows after reduction => **Final addition!** (32-bit Carry Propagate Adder)
Therefore, the expected time delays for 32-bit multiplication using Wallace multiplier is (1 AND gate delay) + 8 * (Full Adder delay) + (Final addition delay == 32-bit Carry Propagate Adder delay).
Signals of multiplier:
Inputs: clk, rst, a[31:0], b[31:0]
Outputs: result[31:0], **mult_resp**(it may take many cycles to finish multiplication.)

If the Wallace multiplier does not work, we can integrate the Add-shift multiplier from MP1, which will give us 3 points.

How to perform division?
For division, we are going to add a division module that uses the binary version of the famous long division, which works as follows:

```
Q := 0                  -- Initialize quotient and remainder to zero
R := 0
for i = n − 1 .. 0 do           -- Where n is number of bits in N
        R = R << 1      -- Left-shift R by 1 bit
        R(0) := N(i)    -- Set the LSB of R equal to bit i of the numerator
        if R ≥ D then
                R = R − D
                Q(i) = 1
        End
End
```

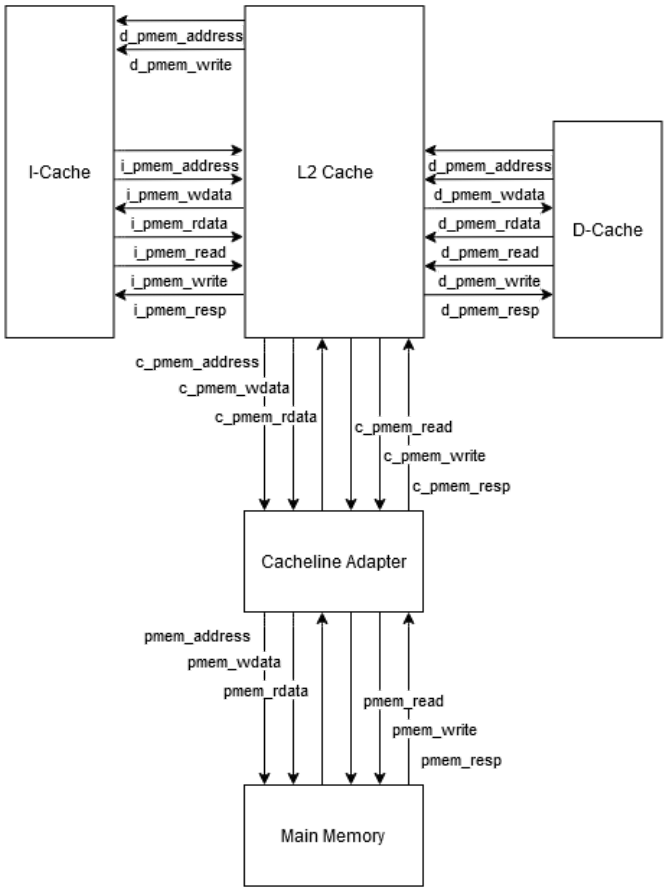
To do R = R - D, we simply take the 2's complement of D, then use an adder to compute the result. In the worst case, this can happen 32 times, which results in a delay of 32 * (delay of adder).

Unified L2 Cache

The L2 cache will sit between the cacheline adapter and the L1 caches. It will be a 2KB 2-way cache which holds 32 indices of 32B cachelines. The address will be broken up as follows:

22 bit tag	5 bit index	5 bit offset

With an attachment as such:

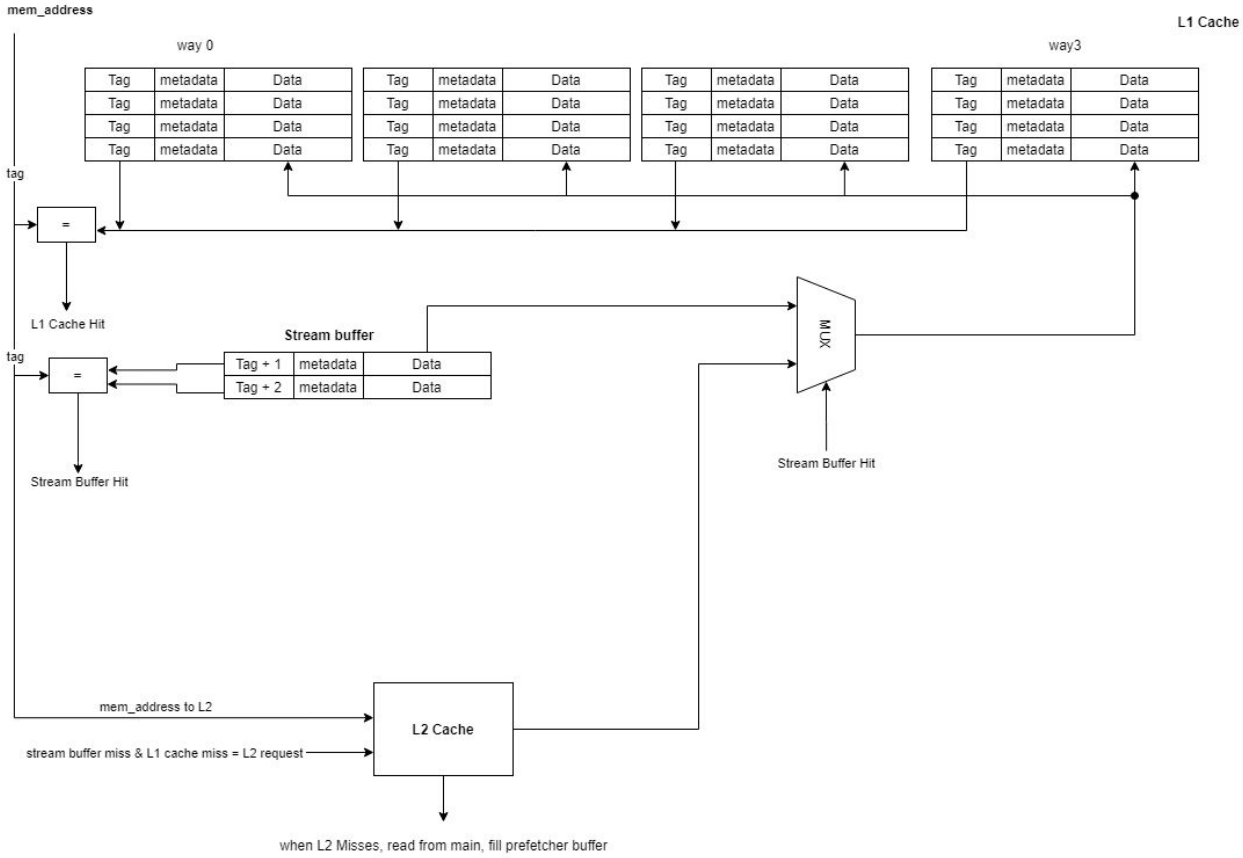


The L2 cache will have control logic similar to the L1 caches, but will also serve as an arbiter between them. In the event of a writeback from the D-cache, an invalidate will be sent to the I-cache on a match. This will enforce basic coherence where we assume that the I-cache cannot write to L2, so only the D-cache would cause such an event. The L2 cache will provide next cycle reads of the cacheline on a hit, which will greatly boost performance for regaining evicted blocks from the L1 caches.

4-way L1 Caches

Instead of 2 copies of every array in the L1 caches, we will have 4. This will double the storage, and widen the select signals by a single bit. Otherwise, the read and write timings will remain intact. The main source of complexity will be in the lru bits. This will be remedied through 32 sets of 3 bits which form a binary tree for the 4-ways that they represent. On each hit, the 3 bits will be updated according to which branch of the tree was used last. On a writeback, the 3 bits will determine which cacheline to replace.

Hardware Prefetching



Basic diagram for implementation of one stream buffer

Problem: There is large latency between a memory request and when the data is actually used, so we will try to implement a two-line stream buffer for both the I-cache and D-cache.

Implementation: A prefetch mechanism based on the L1 and L2 cache hit will be used to determine whether or not a prefetch is being requested: if both L1 and L2 caches miss on a CPU request, the prefetch request will signal to main memory that mem_address + 1 and mem_address + 2 are also needed in the stream buffer.

When L1 misses, but there is a match in the stream buffer, we choose to input data from the stream buffer instead of L2. If this happens, we evict the block from the stream buffer, and send a prefetch request for the next mem_address + n block. For this to work, we'll implement a counter to keep track of the stride at the tail end of the buffer.