Lab 2: Caches

CPE 333 - F23

In this lab, you will design a cache module and integrate it with the your Pipeline Implementation.

Caching

Cache is the name given to the highest or first level of the memory hierarchy encountered once the address leaves the processor. Because the principle of locality applies at many levels, and taking advantage of locality to improve performance is popular, the term cache is now applied whenever buffering is employed to reuse commonly occurring items.

When the processor finds a requested data item in the cache, it is called a cache hit. When the processor does not find a data item it needs in the cache, a cache miss occurs. A fixed-size collection of data containing the requested word, called a block or cache line, is retrieved from the main memory and placed into the cache. Temporal locality tells us that we are likely to need this word again in the near future, so it is useful to place it in the cache where it can be accessed quickly. Because of spatial locality, there is a high probability that the other data in the block will be needed soon.

The time required for the cache miss depends on both the latency and bandwidth of the memory. Latency determines the time to retrieve the first word of the block, and bandwidth determines the time to retrieve the rest of this block. A cache miss is handled by hardware and causes processors using in-order execution to pause or stall until the data are available.

Overview Design Diagram

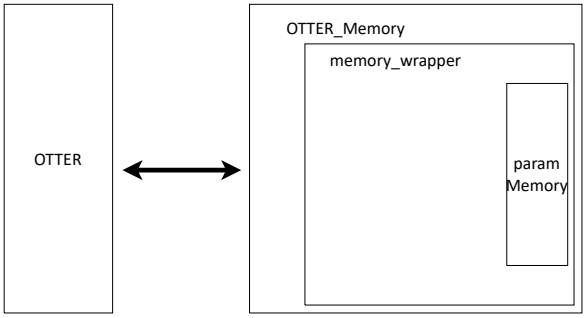

Figure 1: Otter + parameterized (variable delay) memory

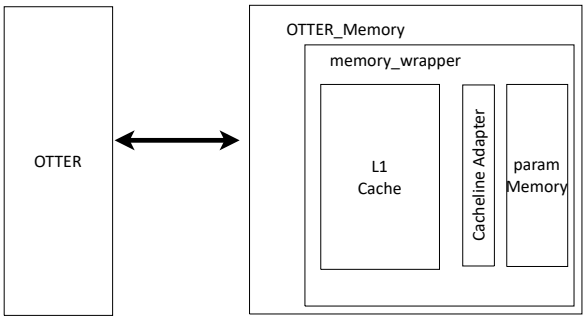

Figure 2: Otter + parameterized (variable delay) memory + Cache

Cache Specifications

You will need to design (and verify) a one-level, unified, 2-way set-associative cache [1].

Previously, the OTTER datapath was interacting with the main memory directly. Now, you will need to modify the interface to implement the memory hierarchy. That is, you will need to insert a cache between the OTTER's datapath and the main memory. You may NOT add additional signals between the cache and the OTTER datapath. Your cache must work with the same signals that OTTER main memory used to communicate with the OTTER; the datapath must have no knowledge of your memory hierarchy.

The main memory code will be provided as param_memory.sv. This memory module has a slightly increased delay. The memory bandwidth can also be increased to 256 bits, so that a single load will fill an entire cache line. Additionally, reads and writes occur in bursts over 8 cycles, requiring the use of a cache line adapter.

The cache can be constructed using the following components:

- Control unit (you must create a state diagram for this)
- Decoders, Comparators, Muxes, Logic gates, Registers
- Cacheline adaptor
- 2 data arrays (provided as data_array.sv)
- Metadata arrays (provided as array.sv):
- 2 tag arrays
- 2 valid bit arrays
- 2 dirty bit arrays
- LRU bit array

Read/Write hits MUST take at most two clock cycles to complete in this cache. Other operations may take multiple cycles, if necessary.

A cache with a two-cycle hit will follow the following diagrams:

[1] Cache Specifications:

- 8 sets with 2 ways per set
- Each way holds an 8-word cache line (32B)
- Uses a Write-back and write-allocate policy
- LRU replacement policy
- Read/Write hits must take at most 2 clock cycles to complete.

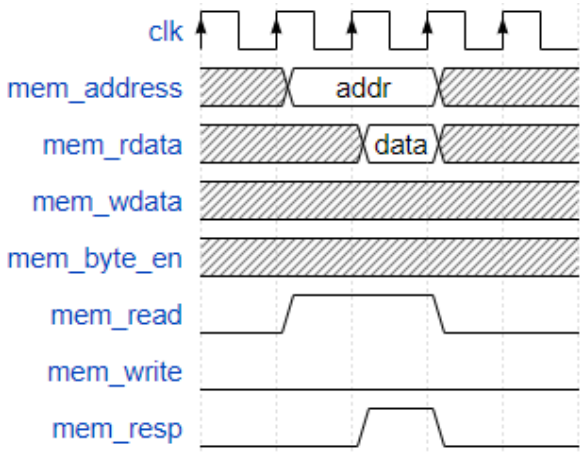

Figure 3: Example of 2-cycle read-hit

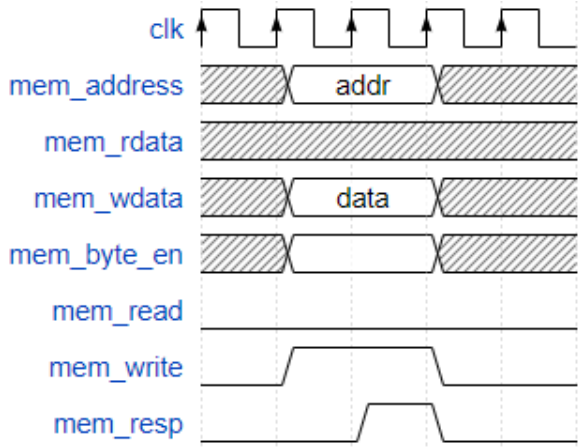

Figure 4: Example of 2-cycle write-hit

Cache Structure

Signals between the OTTER and param memory (No Cache).

- **mem_address[31:0]** The memory system is accessed using this 32 bit signal. It specifies the address that is to be read or written.
- **mem_rdata[31:0]** 32-bit data bus for receiving data from the memory system.
- **mem_wdata[31:0]** 32-bit data bus for sending data to the memory system.
- **mem_read** Active high signal that tells the memory system that the address is valid and the processor is trying to perform a memory read.
- **mem_write** Active high signal that tells the memory system that the address is valid and the processor is trying to perform a memory write.

- **strobe[3:0]** A mask describing which byte(s) of memory should be written on a memory write.
- **mem_valid** Active high signal generated by the memory system indicating that the memory has finished the requested operation.

Description of Provided Files. he following files are provided on canvas:

- **array.sv** A register array to be used for tag arrays, LRU array, etc.
- **cache.sv, cache_control.sv, cache_datapath.sv** Some blank modules to help you get started.
- **data_array.sv** A special register array specifically for your data arrays. This module supports a write mask to help you update the values in the array.
- **top.sv** Testbench to simulate your design.
- **param_memory.sv** The main memory module, with delay, which will be connected to your cache. This memory is different than that provided in 233 in that its access granularity is now 32-byte.
- **rvfimon.v** RVFI verification monitor.
- **shadow_memory.sv** Similar to the RVFI verification monitor, this module will help detect errors in your cache. The RVFI monitor aims to be synthesizable, which means it is impossible for it to keep track of memory state. This module does not aim to be synthesizable so it is able to maintain a copy of memory which updates every time the OTTER performs a write.
- **mem_itf.sv** The interface used to connect the memory_param to OTTER_memory.
- **cache_monitor_itf.sv** The interface used to connect the cache and DUT in the testbench.

Cache-line Adapter

When typical microprocessors load or store a byte of data, the memory controller interfacing with DRAM will typically request an entire "cacheline" of data. These cachelines are typically 32 or 64 bytes data in the address space contiguous with the requested byte, and aligned to 32 or 64 byte boundries. However, pin limitations on packages, as well as the design of DRAM DIMMs make it infeasible to send an entire cacheline concurrently. Instead, DRAMs support burst transmission modes, in which the cacheline is sent over several cycles.

You must design an adaptor which does two things:

- On loads, buffers data from memory until the burst is complete, and then responds to the lowest level cache (LLC) with the complete cache line.
- On stores, buffers a cacheline from the LLC, segments the data into appropriate sized blocks for burst transmission, and transmits the blocks to memory.

Changes to your OTTER from 233 for the variable latency memory

In 233, you likely used a memory module with a single cycle delay. The section describes the changes needed for making your OTTER compliant with the parameterized memory (which is more representative of larger scale off-chip memory technology, e.g. DRAM).

There are two new outputs from the OTTER_memory module to your CPU: **mem_valid1** and **mem_valid2**. These signals are high when DOUT1 and DOUT2 have the valid data requested from the OTTER respectively. Additionally, after a store instruction, mem_valid2 will go high when the write has completed. The following changes are necessary for your OTTER

- Add mem_valid1 and mem_valid2 wires between the OTTER_memory module and your control unit FSM (OTTER_CU_FSM). The control unit FSM uses these signals as inputs.
- Your control unit FSM should **stall** in the fetch state until mem_valid1 goes high (e.g. if(mem_valid1) state<=EXECUTE)
- Your control unit FSM should **stall** in the execute state if the instruction is a load until mem_valid2 goes high. It should also **stall** in the executes state for store instructions until mem_valid2 goes high.
- PCWRITE should only be set high in the WB state or in Execute state if a non-LD instruction. For store instructions, PCWRITE should be set high only when in the execute state and when mem_valid2 is high.