CS7290 – Final Report – Cache System

Members: Syed Minhaj Hassan, Nickvash Kani

***Abstract*—The goal of this final project was to build a cache system using verilog. We implemented a 2-level, non-blocking, write-back cache. We had private L1 instruction and data caches with a shared L2 cache. The whole thing was implemented in Verilog and verified using custom, work-intensive testbenches.**

I. Section I: Overall Design and Target

THE goal of the final project was to implement a cache structure using Verilog. We created a 2-level caching system. The L1's are private, direct mapped, non-blocking, write-back caches. The L2 is shared, set-associative (4-way), non-blocking, write-back cache. Overall we succeeded in the creation of a functional cache that keeps resource consumption at a minimum. The interfaces of both the caches are same. The implementation inside is also very similar, except the ram itself is replicated to implement set-associativity. Figure below gives the block box view of the cache which will be described further in the next section.

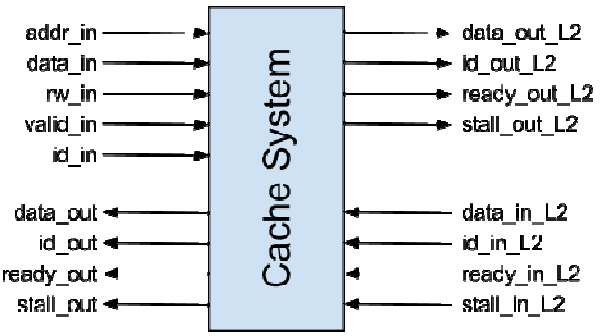

Figure 1: Black Box diagram of the cache system. All input/ouput ports are displayed.

II. Section – 2: Design And Methodology

A. *Interface to Processor*

Figure 2 gives an example showing the timing diagram of the interface to the processor. The initial condition in this example is a request with id 5 that happened to be a miss. Also, the cache stall_out signal is high which means the processor cannot send anymore request. Any change in data, addr_in or id_in bit is ignored during this time. In 3rd positive edge, stall_out signal goes down and ready_out goes high with id_out becoming 5. This indicate that a valid data for id 5 has arrived. The processor can read and process it.

A new request can be sent immediately as shown by request of address 63 with id 6. Note that the cache can handle a small delay from the clock edge for the valid_in signal. Since this is a read request, rw_in = 0, any value in data_in is ignored. This request happens to be an L1 hit. The ready_out signal become high again in the next cycle indicating valid output for id 6.

In the next cycle, a write request to address 69 is received with id 7. This is again a hit but since it is a write request, ready_out remains low. However, the data has been written in the cache.

The next request is read address c5 which is a miss. Furthermore, the request caused the stall_out signal to become high in the next cycle. This can be the result of many reason, the blocking cache, the MSHR getting full, or any other reason due to which the cache cannot handle the new request in the subsequent cycles. This stall_out indicates the processor that no further request should be sent. Any changes in the addr value or data_in value during this time when stall_out is high, will not affect the cache state. The ready_out signal will not get high until the cache request is being serviced and a valid data is available.

Also, it is important to note that as soon as the stall_out signal goes low, next request can be sent, regardless of whether the previous request has been serviced or not. It is important to understand that the cache is non-blocking, so the stall_out will not get high on each miss. id_out is an important signal since request will return to core out of order and id_out indicates to which request it actually belonged to.

The interface towards the memory side is similar. We tried the keep the interface exactly the same so that multiple cache level hierarchies are possible. The upper level cache will act as a processor core for the lower level of caches.

B. *Overall Microarchitecture*

Figure gives the overall micro-architecture of the direct-mapped cache. The cache has 2-ported tag and data ram. The tag ram also includes separate rams for valid and dirty bits. Port A handles the operation from the core

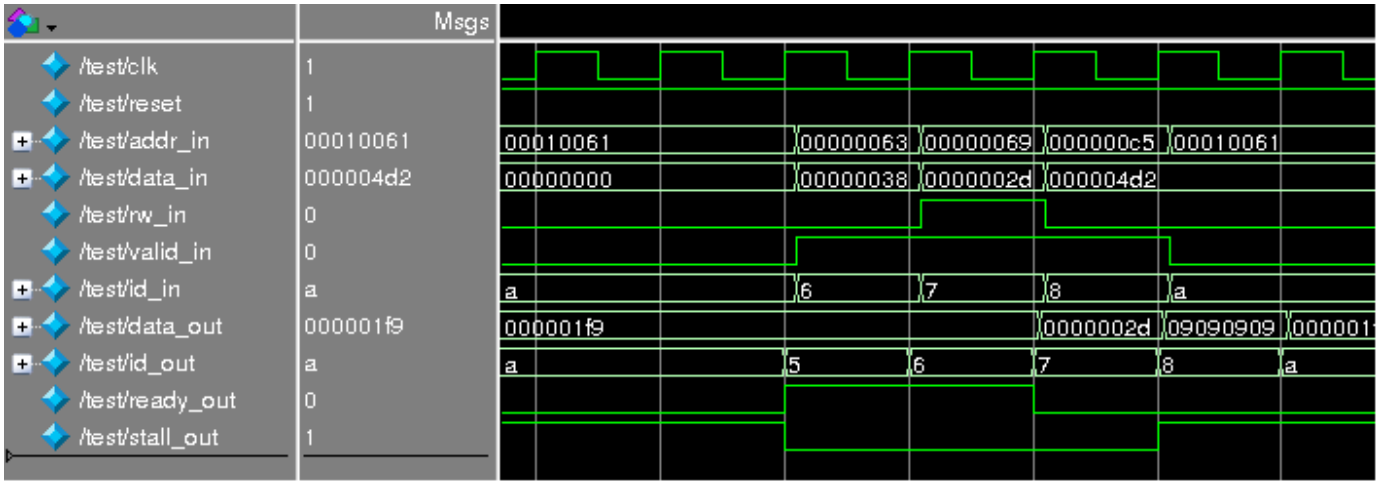

Figure 2: Timing diagram showing example of cache to core communication

side. Port B handles operations from the L2/memory side. A request enters the cache from the processor and reads the tag through its port A in the negative edge of the same cycle. If it's a hit, the request is read from / written to the ram during the positive edge of the same clock. Thus a single cycle operation is achieved in case of a hit. Since, a request is only 4 byte (32 bits), and the cache line is 32 byte long, we need a multiplexer/de-multiplexer for reading in / writing out of the data ram. The select lines for these are generated using the lower address bits. If the request is a miss, it is saved in the MSHR by generating the MSHR add command and its associated signal. The MSHR operations are discussed in detail later.

Port B handles the L2 side of the data and tag rams. Since L2 operates at cache line granularity, port B of the cache needs to be 32 bytes long. Thus the data ram not only has to be multi-ported, the widths of these ports have to be different as well. We implemented the data ram by implementing 8 smaller rams of 32 bits each. From port A, only 1 of these 8 is being selected. From port B, all of them are selected to generate a 256 bit output. This design makes it very easy to extend our cache model for SIMD execution where wider cache lines are required at the port A as well.

The saved miss requests in MSHR are read by the non-blocking fsm. This use MSHR's read next operation. The fsm handles this request and send it to the lower levels of cache. On return, the fsm again reads these requests by MSHR's get operation which saves the original request along with its data and id in the prev_reg data structure. This data structure is used to write the request back to the data and tag rams and recover the cpu_id of the original request to be sent to the processor.

If a write/read request is received for a line that is still being processed by the lower memory hierarchies, that request has to wait for that line to actually arrive in the cache. Note that the request will be a miss, since the reading of the line itself from L2 has not been completed. We cannot send this miss request to the lower level of cache hierarchies because that will overwrite the previous request with the effect of its update being nullified. We cannot save such a request in the MSHR with its current implementation. The reason is that, a miss can be read only once from the MSHR. (Note that MSHR keeps pointers to the next unread request only and does not read the requests that have already been read again). We save this request in a separate data structure, we call it same line buffer. When the earlier request returns and updates the line, servicing of the request saved in the same line buffer is initiated. This is explained in more detail in the non-blocking FSM section. We kept the same line buffer of size 1 only. We block the cache, when this buffer is occupied. A better non-blocking implementation of this buffer is left for future work.

Another problem that we faced during the implementation is reading/writing in the same cycle for L1-tag arrays. The L1 has to read the tag, check if the new request is a hit and write the line (if it's a write request) in the same cycle. Writing a line also includes writing to the tag (more specifically dirty bit). This means that the tag needs to read and write in a single cycle. As discussed earlier, the tag reads at the negative edge of the clock and writes at the positive edge. A straight forward code of a block ram in altera, with operations at both edges will not allow its synthesis tool to infer it as a ram. Since block rams can be triggered only at one edge of the clock. We implemented clock multiplication to generate a clock with twice the frequency and feed the block ram with this new clock. The write enable signal is anded with the actual clock to enable writing only after 2 cycles of this new faster clock. clock multiplication is done by xoring a delayed version of the clock with itself, a commonly used

clock multiplication technique which generates faster clock without 50% duty cycle.

Out L2 caches implement associativity. They are run at a clock 2-times slower than the L1-clock. It does not require any clock multiplication, since we already have the faster clock available. We provide a very simple random replication policy for the L2 cache as follows. We keep a running counter modulo associativity. The value of this counter at the time a new request is received is considered the way of this request. If a valid line is already present in that way for the corresponding index, that line is replaced. The backside of L2 is connected with a dummy memory. This memory responds in variable cycles to model memory operations correctly. We initialized this memory with dummy values. Correct implementation of a memory controller with real programs and their associated data loaded on them is left as future work.

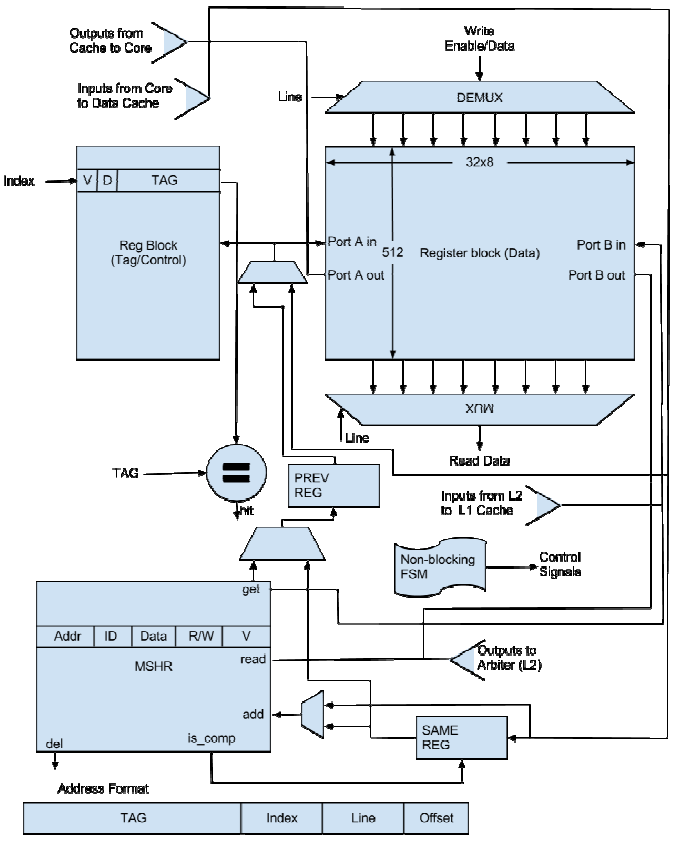

Figure 3: Overall cache microarchitecture

C. *MSHR Structure*

The concept of the MSHR is fairly simple but its implementation is somewhat tricky. Figure 4 shows the basic components of the MSHR. The basic idea behind our MSHR is that we have a large memory array which we use to store the packet request information. The

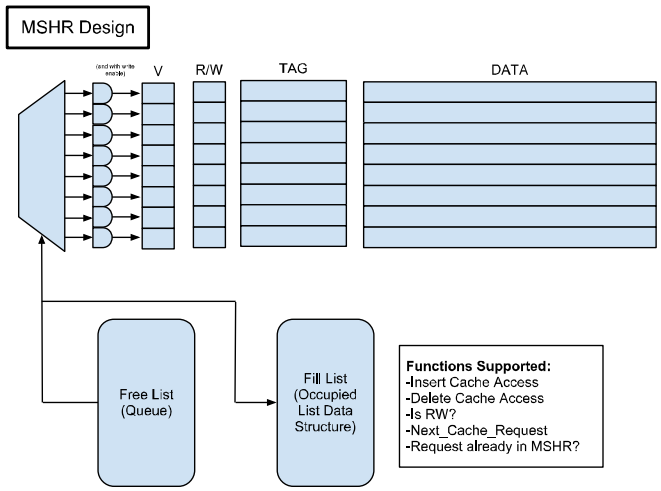

Figure 4: Overview of MSHR

information we stored are address, r/w, cpu_id, way information, data and dirty information. Our MSHR handles the following 5 operations 1) Add - which adds a new element in the MSHR. 2) Delete - which deletes the given element. 3) Get - which reads an entry whose tag is given as a request. 4) Read Next - which reads the next unread request from the MSHR 5) Is Compare - which compares if a request matches a request belonging to the same line as another entry in the MSHR.

When we get a packet request we want to insert it into a

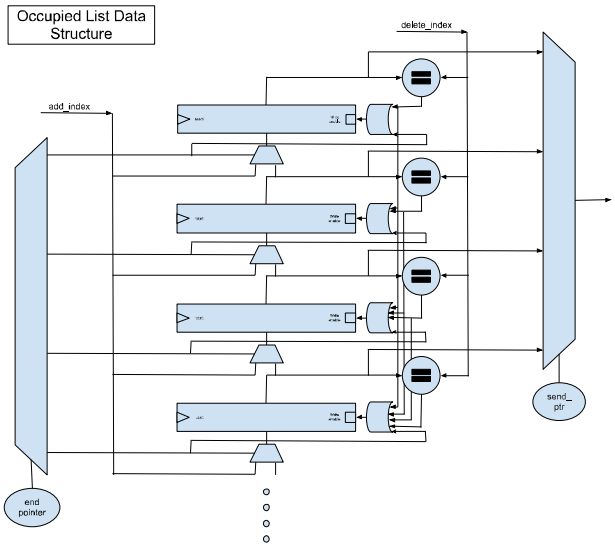

Figure 5: Fill List Data Structure

free spot in this memory array. For this functionality we have a FIFO structure referred to the free list which stores the tags of all the free request array locations. When a request wants to be added in the MSHR, it reads the head of this free list and inserts itself in that array element. The free list value is then popped.

The fill list is a special data structure shown in Figure 5. The basic idea behind this structure is that you want to

keep track of the array elements which are full. However you also want to send out requests in oldest-first order. Thus this data structure is similar to a FIFO in that it adds the consumed tag to the end of the list. However when a request is deleted from the MSHR we know that isn't necessarily the oldest request. Thus in order to keep the oldest first ordering, we delete the related tag from the fill list and shift all the full tags below it up. Hence, the MSHR knows that the oldest requests are kept near the head of the fill list. The fill list also keeps track of which cache requests were sent to the L2 and house an index pointer that outputs the oldest request not read for the MSHR read next function. The Is Compare compares all the entries with the new request. It also returns true, if the request does not belong to the same line but belong to the same set and same way. Thus it will be placed in the same location in the cache as the entry in the MSHR.

D. Non-blocking FSM

Figure 6 below is the state machine for the non-blocking cache. This controls the MSHR, port B of the ram and L2 side interface of the cache. It also generates an enable signal for port A to read/write to a newly brought in line. The general operation is as follows.

Initially, the MSHR is in idle state. When a new request, enters the MSHR, it triggers the MSHR non-empty signal. The FSM jumps to the Read_Next state. This basically means read from the MSHR. It waits for 1 cycle for the MSHR to return data which is sent to port B of the actual ram to read the corresponding line. If the line is dirty, it will be written back to the L2 (note the caches are write-back) and wait for few cycles. These cycles are required for caches with multiple clock domains. This basically is the max synchronization time between the faster and slower clock i.e. a new request to L2 should not be send within the same clock of L2. The read request for the new cache line is then immediately sent, and the cache goes back to idle mode. If the line is not dirty, the L2 write and wait states are bypassed. In the mean time, if a new request come to MSHR, it will serviced similarly. If the L2_stall signal becomes high, the fsm will stop after reaching the idle state. Thus, it is possible that at least 1 new request is being sent before the actual stopping of L1 due to the L2 stall signal. The L1-L2 interface needs to handle this request.

The L2 can send a reply anytime. The FSM will immediately move to the L2_complete state and handle the L2 reply operation. Handling L2 reply means reading the original request along with its address, way and cpu_id corresponding to that line from the MSHR, write the newly arrived cache line to the cache through port B and indicate port A to service the actual read/write request to the line pending in the MSHR. The cache also needs to send a stall signal to processor, since its portA is busy and cannot service a request during that time. If the L1 is in the middle of reading the next line from MSHR and the L2 reply arrived, it must save the L2 reply temporarily before the fsm gets ready to service it. After the L2 complete operations are done, the fsm moves back to the previous state from where it came from.

If the fsm observes that a new cache line arrived in the same line data structure, it immediately move to the blocking state. Thus, the stall signal for the processor gets high and the cache waits for the reply of the line that causes the enable of the same data structure. Once the cache line is written back, and block condition is resolved, the line in the same data structure will be serviced allowing the processor to generate new requests. The block signal stops the MSHR only when the conflicting line's request has been sent to the L2. i.e. the block signal will not stop the MSHR, if the same line request arrives for a request that is pending in MSHR but has not been serviced.

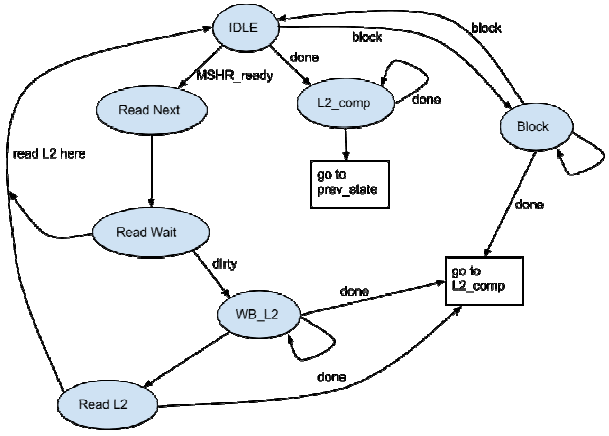

Figure 6: FSM Diagram

E. Arbiter Operation

We implemented shred L2 cache by the use of an arbiter module as shown in the figure below. The arbiter is a somewhat simple piece of hardware. The job of the arbiter is simply to store the cache transactions until the receiving cache is ready to accept them. Thus at the core the arbiter is simply a set of FIFO structures for each cache transaction (L1 to L2, and L2 to L1). Obviously these FIFO structures might get full so stall logic is also included in the arbiter.

The big difficulty with the arbiter structure is that it interacts with two structures which operate at different clock frequencies. This made sequential logic in the arbiter especially tricky. We set the FIFO structures to operate at the (faster) L1 clk. This means that all inputs to the FIFO from the L1 do not need to be

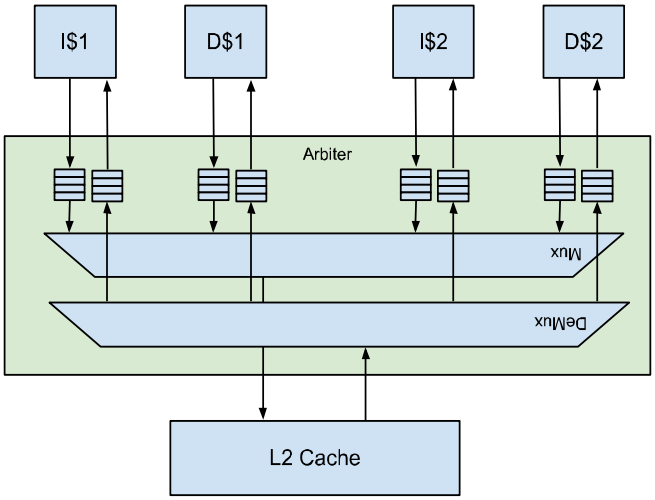

Figure 7: Shared Cache Diagram including arbiter.

masked/converted. However the L2 also has a set of controls going to the FIFO. So these controls must be converted to operate on L1 clk. Since the L2 clk is slower than the L1 clk, an L2 clk pulse needs to be converted to an L1 clk pulse. We do this by using a special module which takes in the signal and both clks. At every positive edge of the L1clk, we sample the L2 clk and when we see that the sample of the L2 clk goes from 0 to 1, we know that the L2 clk has had a positive edge and so we set the signal pulse to the input signal value for one L1 clk cycle. Other than this, there is some arbitration circuitry which determines which FIFO (and thus which L1) sends requests to the L2.

III. CONCLUSIONS

We have designed a 2-level non-blocking cache hierarchy with shared L2. We have thoroughly tested our design using modelsim with custom written test benches. We assumed a 128 KB multi-cycle memory to test our cache hierarchy. Our code is fully parameterized i.e. it can easily be extended to various cache sizes, line sizes, associativity, MSHR widths etc. by changing few parameters. Our code is synthesizable and can work with any fpga/asic. We did not use altera specific components to keep the code portable to other devices as well. We could not show timing/area results due to lack of availability of the board and the associated synthesis tool which will be done in future.