Pipelining the ESP Last-Level Cache

COMS E6901 - Projects in Computer Science - Fall 2022

Kevin Yunchuan Jiang
kyj2112@columbia.edu

1. INTRODUCTION

Systems-on-chip (SoCs) have with increasingly complex integration primarily rely on on-chip shared memory for performance. Like caches on single-core processors, implementing cache-coherence via a cache hierarchy on a multi-core system can greatly improve performance and reduce energy consumption. Previous works have also shown that this can be extended to heterogeneous SoCs, which has the added complexity of managing accelerator memory accesses together with multi-core cache coherence.

ESP is an open-source platform for hetergeneous SoC design that streamlines the integration of heterogeneous components in SoC architecture. The ESP architecture utilizes an extended MESI directory-based cache coherence protocol to implement cache coherence across the processors and accelerators in the system. The ESP cache hierarchy consists of L1/L2 caches that are coupled to the processors and accelerators, as well as a last-level cache (LLC) that not only enables coherence between all L1/L2 caches and main memory, but also allows for accelerators without L1/L2 caches to have access to faster memory acceses directly from the LLC. As shown in a previous work by Giri et al., the ESP LLC is able to greatly reduce the number of accesses to main memory and improve the performance of such LLC-coherent accelerators.

In this work, we present an improved version of the ESP LLC. While the original version is implemented with a multi-cycle datapath, the improved version features a pipelined datapath, allowing for significantly greater throughput for high density work loads. In particular, large memory accesses by accelerators utilize the full capacity of the pipelined datapath and therefore experience particularly large imporvements to performance.

2. THE ESP CACHE HIERARCHY

The ESP cache hierarchy consists of L1/L2 caches and a LLC in order to achieve cache coherence on SoCs with multiple processors and accelerators. Figure 1 provides a high level functional view of each component of the hierarchy. The directory controller refers to the LLC. The LLC interfaces directly with main memory, and services requests from private cache controllers or DMA controllers. The private cache controllers refer to the L1/L2 caches processors (μP) or accelerators, which service memory requests from the processors and sends requests to the LLC if needed. The DMA controller belongs to accelerators without private caches. Rather than interfacing directly to main memory, these accelerators can receive data directly from the LLC if available.

Compared to the L1/L2 caches, the LLC contains additional metadata for the cache lines stored inside the L1/L2 caches at any point in time, such as which L1/L2 caches share the cache line or which L1/L2 owns the cache line. These metadata are necessary for the execution of the extended MESI directory-based cache coherence protocol. MESI stands for the four states that a cache line can be in at any point in time (Modified, Exclusive, Shared, Invalid). The ESP LLC also maintains cache lines that are no longer being used by any L1/L2 cache in a Valid state. These cache lines are the first choice for evictions but can also be sent directly to accelerators if requested through DMA, which is much faster compared to accelerators reading the data directly from main memory. Valid cache lines can also be sent directly to the L1/L2 caches if requested.

Figure 2 shows the components of the cache hierarchy in the context of an example ESP tile grid system. The LLC is located on memory tiles, while and the DMA controllers and L1/L2 caches are located on accelerator tiles and processor tiles. The LLC communicates with the private caches and DMA controller through the ESP Network on Chip (NoC), which reserve 5 separate planes for cache coherence requests/responses and DMA requests/responses.

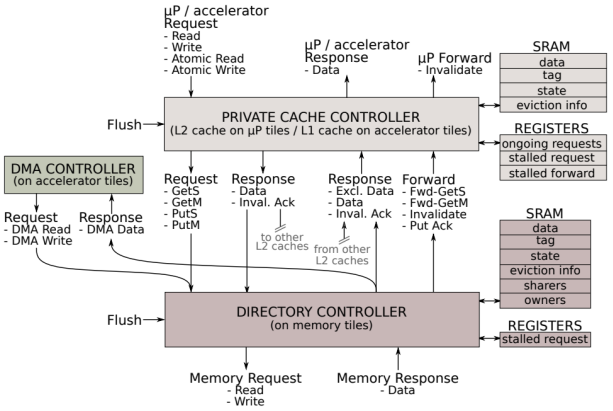

Figure 1: ESP Cache Hierarchy [1]

3. THE ESP LLC IMPLEMENTATIONS

3.1 Original LLC

The original LLC implements a multi-cycle datapath for handling requests according to the extended MESI protocol. The multi-cycle datapath consists of six stages, and the LLC contains an FSM to control each stage of the datapath. The FSM transitions between six states, each state corresponding to a stage in the datapath. When servicing a requests from a private cache or a DMA controller, only one stage of the datapath is active during any cycle. The first stage is

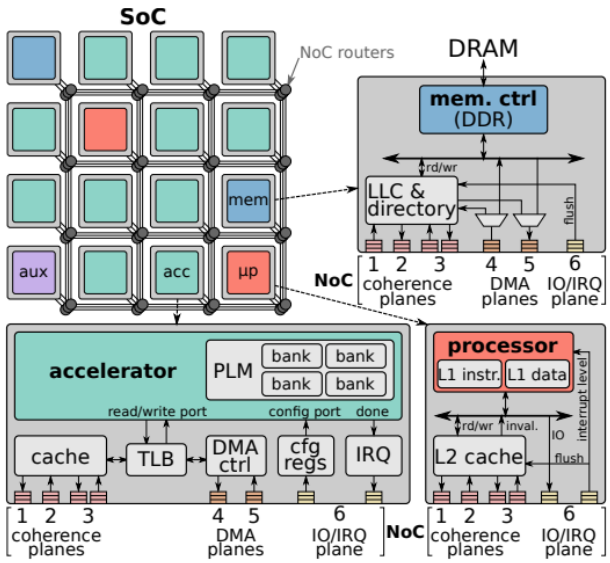

Figure 2: Detailed ESP Tile Grid [1]

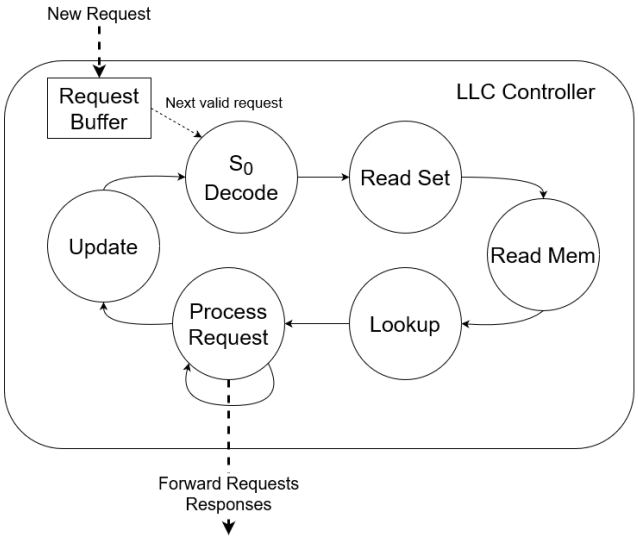

Figure 3: LLC State Machine

responsible for accepting new requests coming in from the NoC. However, during the servicing of a request, the earlier stages including the first stage will be idle, building up backpressure for later requests.

3.2 Pipelined LLC

The pipelined LLC implements six-stage pipeline datapath for handling the requests. Each stage is taken from the multi-cycle datapath and modified with additional logic that allows for backpressure between individual stages rather than just between the entire LLC and the NoC. Because there is no longer an FSM for controllnig the datapath, no stage is inactive at anytime, while the first stage still provides backpressure to the NoC if theres is backpressure from later stages.

The pipeline datapath eliminates idle times in earlier stages and allows multiple requests to enter the datapath at the same time. In the original LLC, large DMA requests require multiple iterations of the multi-cycle data path to complete. From a functional perspective, because the DMA request spans multiple addresses, each iteration is servicing a new "request" for a different address. In the pipelined LLC, these new "requests" can be generated each cycle by the LLC starting from the second stage, allowing the DMA request to fully utilize all stages of the pipeline. This provides a significant speedup if the data being requested exists in the LLC, because later stages do not need to request data from main memory and provide backpressure to earlier stages.

3.3 Unpipelined Implementation of LLC

The unpipelined implementation of the LLC utilizes state machine controller to move between 6 states of handling a request. Each request is handled according to the MESI Directory Protocol. The LLC controller is shown in Figure 3. Each state, with the exception of Process Request takes one cycle. Process Request may take one or more cycles, depending on the type of request.

In the actual RTL code, the LLC consists of several SystemVerilog modules. The modules are shown in detail in Figure 4. The LLC Core module contains the LLC Controller, which is responsible for activating enable signals to the Input Decoder, Address Decoder, Buffers, Lookup, Process Request, and Update modules. These six modules form the multi-cycle datapath for processing requests. Each of these modules correspond to one of the states in the state machine shown in Figure 3. The LLC Controller initially starts in Decode state. Requests from outside the LLC can only be accepted in this state. Requests that come into the LLC while the LLC controller is not in this state will cause the request to be stored in the Request Buffer.

- In the Decode state, the Input Decoder module is enabled by the LLC controller. During this stage, the LLC controller takes in a request from outside the LLC. At the end of this clock cycle, the Input Decoder outputs control signals according to request type.
- In the next clock cycle, the LLC controller moves into the Read Set state and enables the the Address Decoder module. The Address Decoder combinationally decodes the cache set and tag from the address specified in the request. During this state, the decoded cache set is also combinationally sent to the Local Memory module. At the end of the cycle, the Local Memory will output the data stored in the cache lines of the cache set.
- In the next clock cycle, in the Read Mem state, the Buffers module is enabled, and the requested data from Local Memory is stored into the Buffers.
- In the next clock cycle, in the Lookup state, the Lookup module is enabled. In this state, the data inside the Buffers is checked for a tag hit/miss.
- The next state, Process Request, may take multiple cycles. In this state, the LLC controller enables the Process Request module, which outputs the correct data responses, forward-requests, or memory requests according to the control signals. In certain cases, such as when a processing unit requests a piece of data that is currently in the Invalid state, the Process Request module must send out a request to Memory and wait

for a response, which will last several cycles. The Process Request module may also write new data into the Buffers module.

- The Update state takes any new data written to the Buffers during Process Request and writes it to Local Memory.
- Finally, the LLC controller returns to the Decode state and starts to takes in the next request.

In this implementation, only one module in the multi-cycle data path is enabled in any given clock cycle. The Input Decoder, Memory Buffer, and Lookup modules are all idle during the Process Request state. The LLC controller maintains a Request Buffer of size 1, but any more requests will result in backpressure signals from the LLC. This is inefficient because the Process Request state may last several clock cycles, and other processing units cannot issue any new memory requests during this time. Instead of keeping the earlier modules in an idle state, the next few requests can start being decoded, and cache hits/misses can start being checked. To achieve this, we attempt to decouple the earlier modules from the LLC Controller, implement pipeline buffers in between these modules, and add pipelining logic.

It is important to note that the outputs of each module in the multi-cycle data path are not only connected to the next module in the data path. For example, the output control signals from the Input Decoder module is recieved as inputs by the Address Decoder module, the Process Request module, and the Update module. This is not a problem in this implementation because the LLC controller de-enables the Input Decoder module and prevents its output form changing for the entire multi-cycle data path. However, as will be discussed in the following section, these kinds of signals are the main challenge in pipelining this cache design.

3.4 Pipelined Implementation of LLC

The most recent implementation of the Pipelined LLC is shown in Figure 5. This implementation builds open what we had accomplished in the Spring 2022 Semester. During the Spring 2022 semester, four major accomplishments were made:

- Pipeline buffers using FIFOs were inserted between each module. We chose to use FIFOs as our pipeline buffers because FIFOs have built-in valid and ready signals (full, empty, push, pop) that can be used for back pressure and other pipelining logic within the modules. Additionally, our implementation of the FIFO can take in custom data types. Custom data types are important because there are a large number of different signals going between each module, and every signal needs to be pipelined. Rather than allocating a register for each individual signal, we chose to pack the signals into custom data types to be passed into the FIFO.
- Custom data types were created for each set of signals between modules. We used SystemVerilog packed structs to combine multiple signals into a single data type. In total, we implemented five custom data types for each pipeline FIFO. Each FIFO handles different sets of signals, so five different data types were necessary. Additionally, we implemented logic for flattening 2D bit arrays into 1D bit arrays. SystemVerilog packed structs only accept 1D bit arrays as part of its constituent data, but some signals inside the LLC are implemented as 2D arrays. In order to pipeline these signals through the FIFO, they must be flatted and un-flatted at the input and output of the FIFO.
- The Input Decoder module was decoupled from the FSM controller. Instead of using an enable signal from the LLC Controller, the Input Decoder is now always active and communicates directly with the rest of the cache hierarchy. Upon receiving a new request from the outside, the Input Decoder will push the result control signals into the FIFO buffer between the Input Decoder and the Address Decoder. When the FIFO buffer is full, the Input Decoder will send backpressure to the outside. When the FIFO buffer is empty, the Input decoder will tell the outside that the LLC is reeady for a new request.
- In order to preserve the correctness of the LLC after pipelining, we re-routed important signals from the Input Decoder. As shown in Figure 4, the original implementation routes control signals directly from the Input Decoder directly to the Process Request and Update modules. This works when the FSM controller is in control of all modules, because the Input Decoder will not be active during later stages, and the control signals will stay the same. However, after decoupling the Input Decoder from the FSM controller, the Input Decoder can now handle a second request after a first request enters the FSM controller. When the Input Decoder handles the second request, the output control signals will change, and the Process Request module will now have the control signals corresponding to the second request instead of the first request. To solve this issue, we routed the control signals of the Input Decoder through FIFO buffers between each module, as shown Figure 5. The control signals enter the FSM controller together with the request data. At each state, the control signals propagate through one pipeline FIFO. When the control signals reach the FIFO before the Process Request module, they stay there until the Process Requests completes its task and sends a pop signal to the FIFO. This ensures that the correct Correct Signals will be inputted into the Process Request sub-module for the entirety of the Process Request state.

Our work in the Spring 2022 semester still left many bugs to be solved. The unpipelined implementation of the LLC came with a testbench to verify the funtionality of the RTL code. The implementation at the end of the Spring 2022 semester was unable to pass the unpipelined testbench. There would also be incorrect responses when this implementation received two consecutive requests.

This semester, three major accomplishments were made:

- We fixed the Spring 2022 implementation so that it could fully pass the unpipelined LLC testbench. This verifies the functional correctness of our code when used in contexts that do not require pipelining, and was an important first step. To acheive this, we needed to remove some LLC state variables and send them through the pipeline instead. The LLC State Variables module can be seen in Figure 4. These state variables

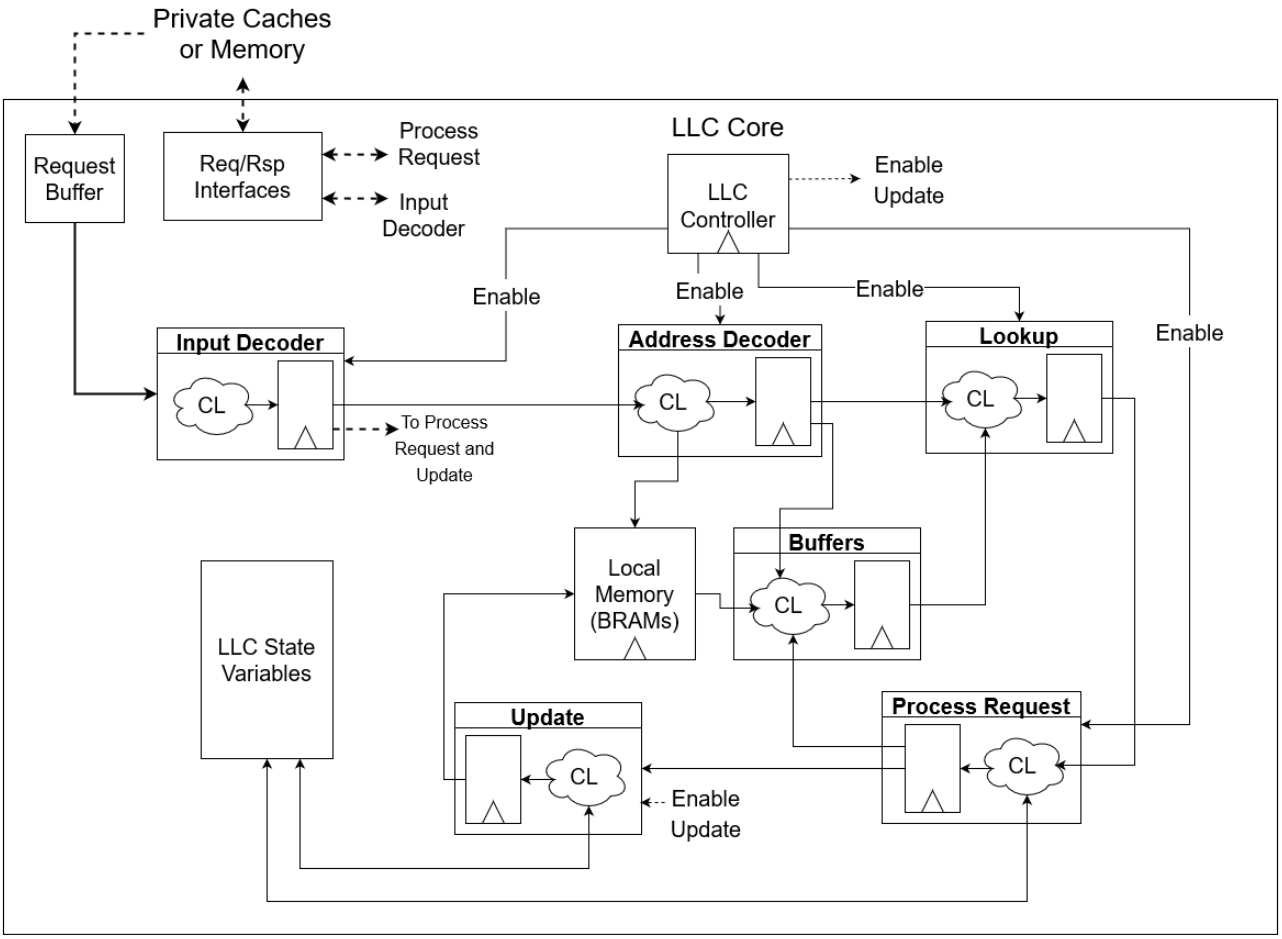

Figure 4: Unpipelined LLC RTL Modules

output to multiple modules within the datapath, and they are meant to stay constant during the multi-cycle datapath, similar to the control signals from the Input Decoder. However, multiple modules also modify these sate variables when they are active. The Input Decoder, now decoupled, would modify these state variables at the wrong time. While we did not completely remove the LLC State Variables module, we removed the module from Figure 5 to show the changes that we made.

- We decoupled the Address Decoder and Local Memory modules from the LLC Controller and passed the unpipelined LLC testbench. There were many challenges when achieving this. First, the cache set that is decoded by the Address Decoder needed to be pipelined so that the Update Module could use it a few cycles later. However, the cache set also needed to be sent combinationally to the Local Memory module in the current cycle. As as result, we needed to implement additional logic inside the Local Memory module to choose the right set (from Address Decoder or from Update). We also found out that the Process Request module was using data directly from the Req/Rsp Interfaces module, as shown in Figure 4. This module contains all the data for the current request coming into the LLC, such as the memory address being requested. However, since the current request coming in is different from the request already in the pipeline, the Process Request would end up using the wrong address. To solve this, we needed to create a new data structure to store all the fields of the request, because the Req/Rsp Interfaces module uses the SystemVerilog interface construct, which cannot be part of a SystemVerilog packed struct. This data structure is then used to send the request fields through the pipeline so that the Process Request module will have the correct request fields a few cycles later.

- We set up a pipelined LLC testbench which sends three consecutive requests into the LLC. The most recent LLC implementation is able to receive all three requests and send out the correct responses.

4. EVALUATION

4.1 Unpipelined LLC testbench

As described in the previous section, the unpipelined LLC testbench is used as the first test for the LLC. Whenever we decouple a module, we need to first pass this testbench. The

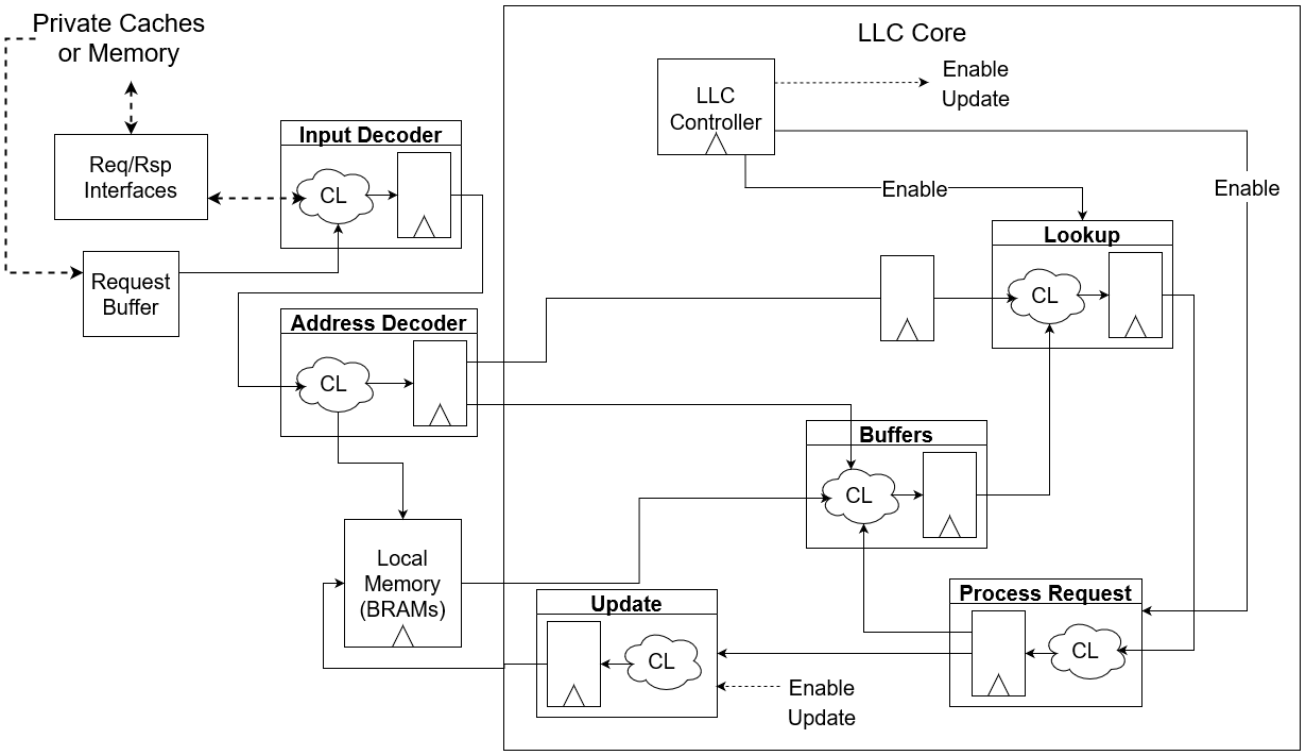

Figure 5: Pipelined LLC Block Diagram

current implementation with both the Input Decoder and Address Decoder decoupled is able to pass this testbench.

4.2 Pipelined LLC testbench

This testbench is a modified version of the unpipelined LLC testbench. Instead of sending one request and waiting for one response at a time, this testbench sends in three consecutive requests and waits for three consecutive responses. Currently, this testbench is extremely short and only tests one type of request at a time. However, the fact that our LLC implementation is able to pass this testbench means that the pipeline capability of the LLC is now functionally correct. The results of this testbench versus the results of a shortened unpipelined testbench can be seen in Figure 6 and Figure 7. The pipelined testbench ended three cycles earlier thanks to the pipelining.

4.3 ESP Full System simulation

The current LLC implementation was also tested in the ESP full system simulation, because it had already passed the unpipelined testbench. However, the current implementation only passes a portion of the simulation, and debugging is still in the process.

5. FUTURE WORK

- The remaining stages have yet to be decoupled. The Buffer module is the next module in the data path. Since it is used for both writing and reading to and from Local Memory, there will be additional challenges to implementing it.
- The current LLC does not yet pass the ESP full system simulation. Even without fully decoupling the stages, work can still be done in parallel to debug the LLC so that it passes the simulation. This would also be more productive in fully verifying the functoinality of the LLC. Once the LLC passes the simulation, we can get to FPGA testing.
- The pipelined LLC testbench needs more work so that it can test more types of requests for a longer period of time.

6. REFERENCES

[1] Davide Giri, Paolo Mantovani, and Luca P. Carloni. **NoC-Based Support of Heterogeneous Cache-Coherence Models for Accelerators**. In *2018 Twelfth IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 1–8, 2018.

```
Info:  llc_tb.   Reset LLC.
Info:  llc_tb.   Reset LLC.
Info:  llc_tb.   Reset LLC.
Info:  llc_tb.   Reset LLC.
Info:  llc_tb.   Reset LLC.
Info:  llc_tb.   T0) TEST OF MULTIPLE REQUESTS AT ONCE.
Info:  llc_tb.   T0.0) Invalid state. No eviction.
Info:  llc_tb.   @77250 ns : REQ_IN : (coh_msg: PUTS, hprot: 0, addr: 04f7d9de, req_id: 00, word_offset: 0, valid_words: 0, line: 000000000 000000000 000000000 000000000 )
Info:  llc_tb.   @77262500 ps : REQ_IN : (coh_msg: PUTS, hprot: 0, addr: 04f7dbde, req_id: 01, word_offset: 0, valid_words: 0, line: 000000000 000000000 000000000 000000000 )
Info:  llc_tb.   @77275 ns : REQ_IN : (coh_msg: PUTS, hprot: 0, addr: 04f7ddde, req_id: 02, word_offset: 0, valid_words: 0, line: 000000000 000000000 000000000 000000000 )
Info:  llc_tb.   @77312500 ps : RSP_OUT : (coh_msg: DATA_DMA, addr: 04f7d9de, line: 000000000 000000000 000000000 000000000 , invack_cnt: 00, req_id: 00, dest_id: 00, word_offset: 0)
Info:  llc_tb.   @77362500 ps : RSP_OUT : (coh_msg: DATA_DMA, addr: 04f7dbde, line: 000000000 000000000 000000000 000000000 , invack_cnt: 00, req_id: 01, dest_id: 01, word_offset: 0)
Info:  llc_tb.   @77412500 ps : RSP_OUT : (coh_msg: DATA_DMA, addr: 04f7ddde, line: 000000000 000000000 000000000 000000000 , invack_cnt: 00, req_id: 02, dest_id: 02, word_offset: 0)
Info:  llc_tb.   === Test completed ===
SystemC: simulation stopped by user.
```

Figure 6: Pipelined Testbench Results

```
Info:  llc_tb.   Reset LLC.
Info:  llc_tb.   Reset LLC.
Info:  llc_tb.   Reset LLC.
Info:  llc_tb.   Reset LLC.
Info:  llc_tb.   Reset LLC.
Info:  llc_tb.   T0) FULL TEST OF COHERENCE PROTOCOL.
Info:  llc_tb.   T0.0) Invalid state. No eviction.
Info:  llc_tb.   @77250 ns : REQ_IN : (coh_msg: PUTS, hprot: 0, addr: 001c95ce, req_id: 00, word_offset: 0, valid_words: 0, line: 000000000 000000000 000000000 000000000 )
Info:  llc_tb.   @77312500 ps : RSP_OUT : (coh_msg: DATA_DMA, addr: 001c95ce, line: 000000000 000000000 000000000 000000000 , invack_cnt: 00, req_id: 00, dest_id: 00, word_offset: 0)
Info:  llc_tb.   @77325 ns : REQ_IN : (coh_msg: PUTS, hprot: 0, addr: 001c97ce, req_id: 01, word_offset: 0, valid_words: 0, line: 000000000 000000000 000000000 000000000 )
Info:  llc_tb.   @77387500 ps : RSP_OUT : (coh_msg: DATA_DMA, addr: 001c97ce, line: 000000000 000000000 000000000 000000000 , invack_cnt: 00, req_id: 01, dest_id: 01, word_offset: 0)
Info:  llc_tb.   @77400 ns : REQ_IN : (coh_msg: PUTS, hprot: 0, addr: 001c99ce, req_id: 02, word_offset: 0, valid_words: 0, line: 000000000 000000000 000000000 000000000 )
Info:  llc_tb.   @77462500 ps : RSP_OUT : (coh_msg: DATA_DMA, addr: 001c99ce, line: 000000000 000000000 000000000 000000000 , invack_cnt: 00, req_id: 02, dest_id: 02, word_offset: 0)
Info:  llc_tb.   === Test completed ===
SystemC: simulation stopped by user.
```

Figure 7: Unpipelined Testbench Results