

ECE 571

L3 Cache Simulator

Archit Datta Kunchaparthi
Bharadwaj Thandra
Vishwas Reddy Pulugu
Vasudev Rupanaguntla

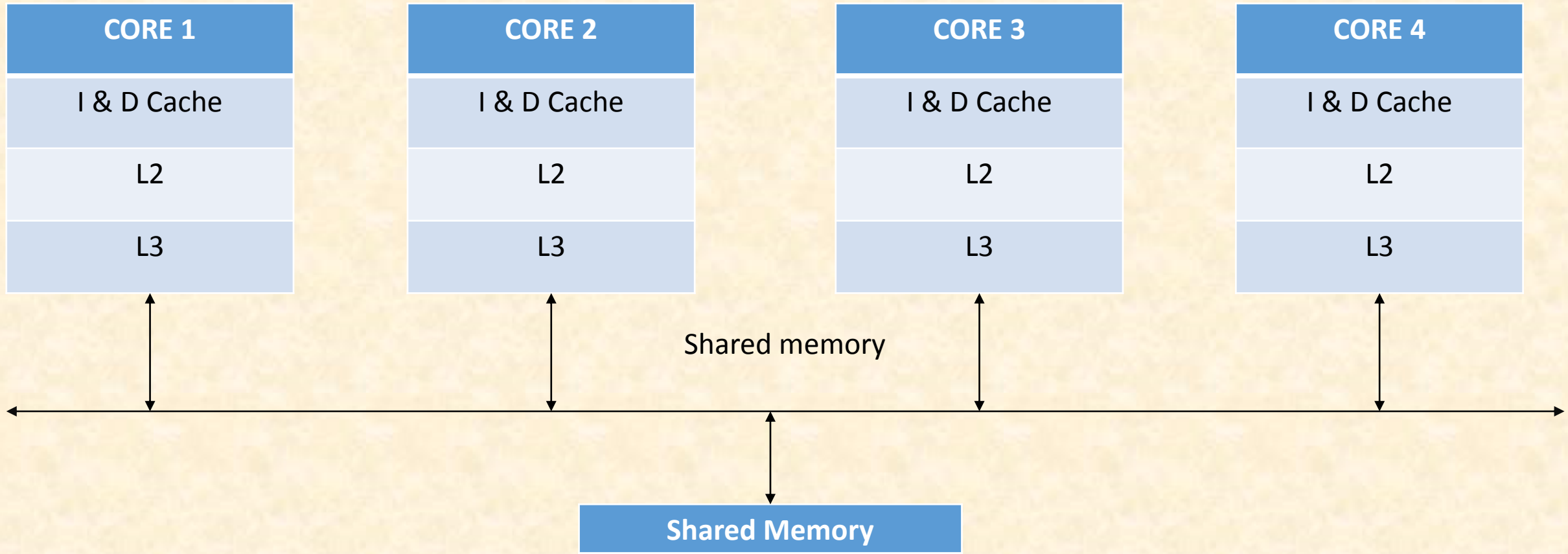
Introduction

- L3 Cache - We have designed the last level cache of a core in a multi-processor environment using a shared memory.
- Details of Cache –

Address Length	32bits(Configurable)
Memory	8MB(Configurable)
Line Size	64B(Configurable)
Associativity	16(Configurable)

- MESIF protocol has been implemented to ensure cache coherence.
- Cache employs inclusivity.

L3 Cache



Implementation of the Design

- Sets in Cache – $8K(\text{memory}/(\text{line_len} * \text{assoc}))$



categorized cache address bits

- Assumptions –
 - Contents of Cache are initially invalid.

L3 Cache Design

- There are 2 inputs – n and address.
 - ‘n’ – operation & ‘address’ – address on which the operation is performed.
- A Cache structure is declared with variable size & initialized to 0.
- Additional bits are allocated for MESIF & LRU.
- For READ or WRITE operations, we check whether the given address is already present in the cache or is to be fetched from memory.
 - In case of it being in cache, we increment HIT
 - Else we increment MISS
- If data is being fetched from cache, its MESIF and LRU bits are updated.

Different Operations

- Operation 1: ($n=0$) It indicates a read request from L2 cache. We check if it's a HIT or a miss.
 - If it's a HIT and the operation is a read, the MESIF and LRU bits are updated using their modules.
 - If it's a miss, only the MESIF bits are updated.
- Operation 2: ($n=1$) indicates write request from L2 cache, we do the same HIT or MISS check.
 - If it's a HIT, the MESIF and LRU bits are updated.
 - If it's a miss only the MESIF bits are updated.

Different Operations

- Operation 3: ($n=2$) It is similar to Operation 1 the only difference being that the request here is for an instruction.
- **Conditions when Cache is Snooping**
- Operation 4: ($n=3$) The cache snoops an invalidate command. If the cache has a HIT for the address provided in the invalidate command it checks its MESIF bits. If it's in either Forward or Shared state it responds by changing its MESIF to invalid state.

Different Operations

- Operation 5: (n=4) The cache snoops a read request and checks for the address.
 - Depending on the MESI state it either returns a HIT, HITM or NOHIT.
- Operation 6: (n=5) NO operation takes place.
- Operation 7: (n=6) If the cache snoops a RWIM on the bus and has the address, we invalidate the MESIF.
- Operation 8: (n=7) MESI is invalidated and cache is cleared.
- Operation 9: (n=8) We print the contents of the STATE of each valid line.

Protocols

- Cache Replacement Policy
 - True LRU algorithm
- Cache Coherence Protocol
 - MESIF
 - Modified
 - Exclusive
 - Shared
 - Invalid
 - Forward

Replacement Protocol

- Replacement (Eviction) happens whenever all cache lines in a set are filled and new address points to same index . Cache line has to be evicted so as to accommodate new entries.
- We have implemented 'True LRU' replacement algorithm.
- LRU – (Least Recently Used) In this algorithm, in case of eviction, the address location which was least recently accessed is replaced by the new entry.

Replacement Protocol

- In this algorithm, we used ordering of cache lines in each set according to the order of accessed addresses.
- We use 'first' and 'last' variables in set structure to keep track of least recently used cache line.
- The line pointed by 'last' variable is replaced and that address is now pointed by 'first' variable of set structure.

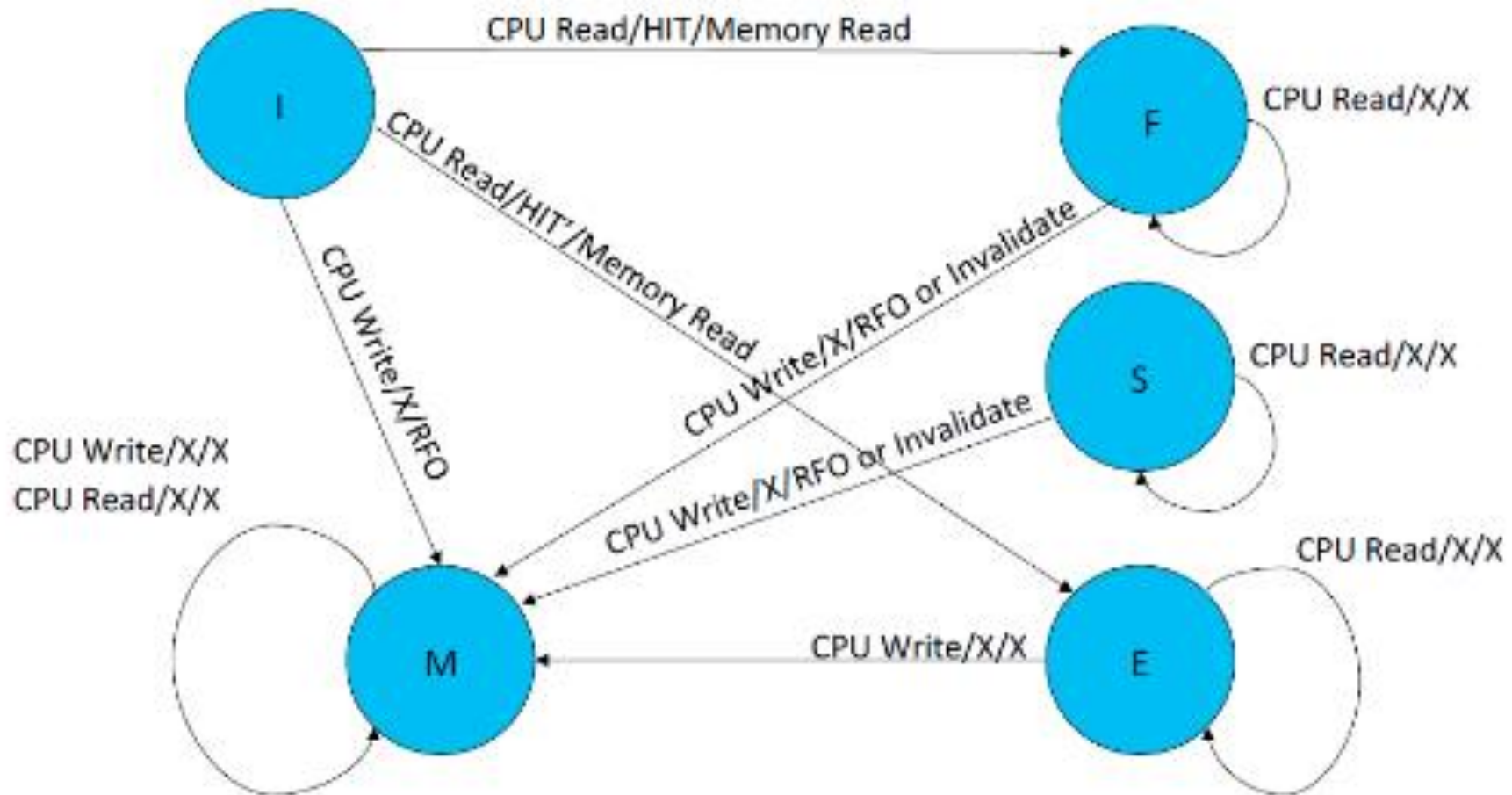
Cache Coherence Protocol

- A multi-processor system involves in referencing a particular data or instruction pertaining to the same address.
 - During this process, there will be inconsistency of the shared resource data that will create incoherence of the shared resource due to modifications done to the shared resource by different processors while they retain it in their cache.
- This inconsistency is eliminated by the help of various cache coherence protocols introduced. In this project we make use of the MESIF protocol in order to maintain coherence with the data.

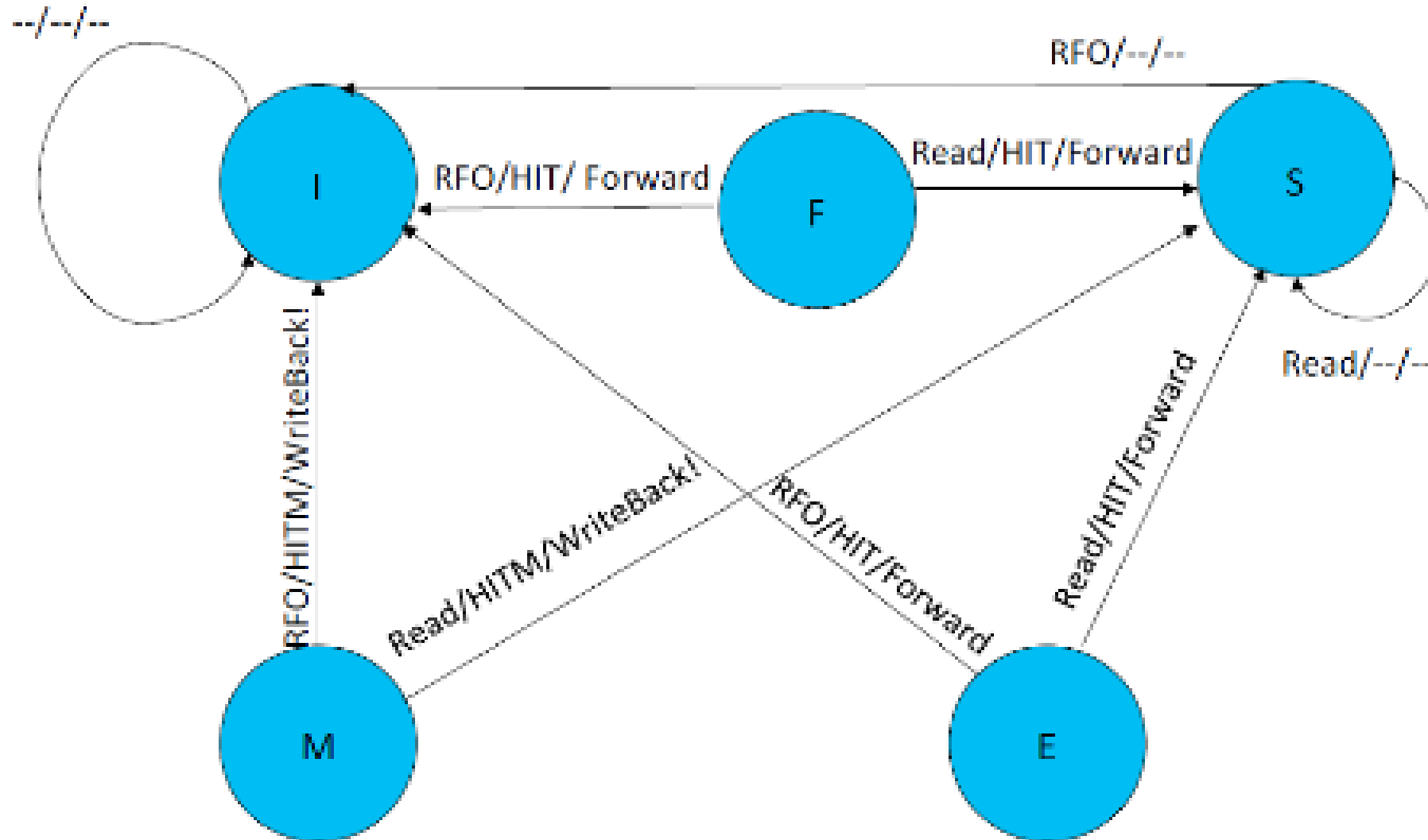
Cache Coherence Protocol

- The MESIF protocol consists of 5 states that represent the acronym of the name of protocol. The states are Modified, Exclusive, Shared, Invalid and Forward states. The definition of each of the states is as follows:
 - **Modified:** The cached copy of the only valid copy of the line and no other processor caches contain the line and the memory copy is out of date.
 - **Exclusive:** No other processor has a copy of the cached item and the processor's cache and memory are identical.
 - **Shared:** At least one other processor has a copy of the line, the cached copies and memory are identical.
 - **Invalid:** The cache entry is invalid i.e., Cache doesn't hold a copy of the line.
 - **Forward:** This is an extended state of the shared state as in the MESI protocol. The processor in forward state is responsible for providing the data if another processor does a read operation. This data is already in other caches but the cache in the forward state responds to any requests pertaining to this data. And this data is identical with the memory.

State dig. When cache controller of processor is accessing memory



State dig. Of snooping processor



System Verilog Constructs

- Interface
 - For cores
 - Bus Signals
- Packages
- Packed Structures
 - Address (Use of Logic)
 - MESIF State
 - Sets
 - Write Buffer

System Verilog Constructs

- Package
 - Parameters

```
`define address_length 32
`define associativity 8
`define line_length 8
`define memory_size 2**10

package l3_pkg;
    .....
    parameter buffer_len = 6;
    parameter adrs_len=`address_length;
    parameter assoc=`associativity;
    parameter line_len=`line_length;
    parameter memory= `memory_size;
    parameter set_len=memory/(line_len*assoc);    //
    parameter tag_len= adrs_len-$clog2(line_len)-$clog2(set_len);
```

System Verilog Constructs

- Package
- Type definitions

```
typedef struct packed{  
  logic [tag_len-1:0]tag;  
  logic [$clog2(set_len)-1:0]index  
  logic [$clog2(line_len)-1:0]byte  
} address_T;
```

```
typedef enum { NOHIT, HIT, HITM} snoop_T;  
typedef enum {READ =1, WRITE , INVALIDATE, RWIM, NOP} busop_T;  
typedef enum {INVALID, MODIFIED, EXCLUSIVE, SHARED, FORWARD} mesi_state_T;  
  
typedef enum {CACHE_HIT,CACHE_MISS} hit_miss_T;  
typedef enum {YES,NO} eviction_T;
```

System Verilog Constructs

- typedef struct packed for definition of cache.
- Package includes tasks for MESIF protocol, LRU algorithm.

```
//cache structure...typedef
typedef struct packed { //:
    bit [tag_len-1:0]tag;
    mesi_state_T mesif;
    int prev;
    int next;
    } lines_T;

typedef struct packed{
    lines_T [assoc:1]line; // [no
    int first;
    int last;
    } sets_T;

typedef struct {
    sets_T [set_len-1:0]set;
    }caches_T;
```

System Verilog Constructs

- Unique Cases
- always_comb
- Assertions

```
unique case (mesif )  
INVALID: begin  
:
```

```
always_comb  
begin  
    .....  
assert (p1.n<9 && p1.n >=0) ;  
    .....  
end
```

Interfaces

- Interfaces are used for Bus Operations and Cores.

- Bus Operation interface is used to Put Snoop Results and Get Snoop Results.
- Core interface is used to provide address and operation from core to the cache design.
- Values to the core interface are provided from test bench.

```
interface cores #(parameter adrs_len=32) ;  
import l3_pkg::*;  
int n;  
address_T address;  
endinterface
```

```
interface bus_signals;  
import l3_pkg::*;  
busop_T BusOp;  
snoop_T PutSnoopResult;  
snoop_T GetSnoopResult;  
  
endinterface
```

Verification Strategies

- The DUT was tested for READ/ WRITE operations
 - When the cache controller of processor is accessing the memory
 - When the cache is snooping
 - When the cache is full and replacement is required
 - To check the functionality of Write Buffer
- Directed Testing
 - Performed READS and WRITES to single index.
 - Transition from one MESIF state to other depending on the operation.
 - Invalidating addresses and read from the same.
 - Issuing reads and writes for same index with different tags.
 - Test case resulting in eviction for verifying replacement algorithm.

Verification Strategies

- Randomized Testing
 - Generated random traces and provide it to DUT and reference model using a random test case generator.
 - Used \$urandom_range to generate operation values
- Test Case Generator
 - Created a configurable test case generator which creates trace files for a range of specified addresses.

Verification Strategies

Reference Model:

- Using the input trace, reference model generates expected read_count, write_count, hit_count and miss_count
- It also maintains track of cache contents(only addresses) in cache.
- The results from DUT are compared to the expected values from reference model to verify the working of cache.

Assertions

- Assertions are used to ensure correct transition of MESIF states during the operations
- Assertions are used to check whether the input address has any unknown value and input operation has any out of range value.
- Assertions are used to check the process of eviction
 - When the line which has to be evicted is at top
 - When the line which has to be evicted is in the middle
 - When the line which has to be evicted is at the bottom

Emulation

- Code is synthesizable and ran on Emulator.
- Code isn't clock based so couldn't make it clock accurate.
- Hit count and miss count differs from expected values.
- Golden model cannot be implemented in emulation as it is not synthesizable.

Results

- Simulation successful on PureSim.
- Checked L3 cache on Stand Alone Mode and TBX mode.
- For directed testing, results are matching with golden model
- For random testing, there are few assertion errors.

References

- ECE 585 & ECE 571 Lecture Slides
- Code adopted from ECE 585 Final Project(Fall 2014)
- www.asic-world.com
- Testbench.in
- Sunburst Papers
- Stack Overflow

THANK YOU