Implementation and Evaluation of
MESI Cache Coherence Protocol

By
Vijaya Krishna Kasula

December 2012

In partial fulfillment of the requirements for the Masters in VLSI & CE at IIIT – Hyderabad

Abstract

Recent trends in processor technology use multiple cores on a single chip with dedicated instruction and data caches for each core. When programs running on different cores try to access shared memory locations, corresponding caches of the cores will be populated with redundant data from same location and coherency among the caches becomes a challenge.

Various protocols have been proposed to solve the coherency problem among the caches, MESI being the most widely used snooping based protocol.

Aim of the work is to implement Coherent Data Caches for a Quad-Core environment using MESI Bus Snooping Coherence Protocol and verify the implementation using Constrained Random Test-Bench and evaluate its performance based on the number of cycles spent in average for memory transactions.

Contents

List of Figures

List of Tables

1. Introduction

The performance of computing systems is becoming less scalable as the disparity between processor and DRAM memory continues to grow. It is referred to as the Memory Wall Problem (1). To bridge the memory discrepancy, a large number of on-die transistors are dedicated to increasingly larger caches and other memory related architectural features. Furthermore, many bus protocols are pipelined to improve the overall throughput. To further expedite data processing, applications are parallelized or multi-threaded on multiprocessor systems. Server-class multiprocessor systems are often based on shared-bus architecture, often referred to as symmetric multiprocessor (SMP). To maintain data consistency in SMP systems, multithreaded applications communicate among processors via cache coherence protocols. Due to the sharing of the memory bus, the communication becomes a limiting factor in performance as the number of processors increases.

Importance of effective utilization of shared bus grows with the number of caches using the shared memory. Apart form effective utilization of bus, maintaining coherence among the caches operating on shared memory is very much crucial for the multi threaded programs that run on multiple cores simultaneously. Many coherence protocols are proposed to solve the purpose, among them MESI is the first of its kind.

2. MESI Cache Coherence Protocol

The MESI protocol is a widely used snooping based cache coherency and memory coherence protocol that supports write-back cache, developed at the University of Illinois, Urbana-Champaign (2). Its use in personal computers became widespread with the introduction of Intel's Pentium processor to "*support the more efficient write-back cache in addition to the write-through cache previously used by the Intel 486 processor*" (3).

MESI is a snooping based protocol, which means that the caches monitor the bus activity and make necessary changes to the local line states. In the above context, the cache that “Snoops” the bus activity is generally termed as snooping-cache. A snooping cache also performs additional tasks under some circumstances which are discussed in detail in later parts of the chapter..

A Cache Line in a cache following MESI Cache Coherence protocol can have one of the four following states.

Modified:
The cache line is present only in the current cache, and is dirty (modified value different from that in memory). Value in memory is no longer valid and the cache is required to write the data back to main memory, before permitting any other read of the line by other caches. The write-back changes the line to the Exclusive state.

Exclusive:
The cache line is present only in the current cache, and is clean (matches main memory). It may be changed to the Shared state at any time, in response to a read request from any other cache. Alternatively, it may be changed to the Modified state when host cache writes to it.

Shared:
Indicates that this cache line may be stored in other caches of the machine and is clean (matches main memory).

Invalid:
Indicates that this cache line is invalid (unused).

2.1. Operation

In a typical system, several caches share a common bus to main memory. Each also has an attached CPU which issues read and write requests. The caches' collective goal is to minimize the use of the shared main memory.

Following explanation of MESI cache coherence protocol is based on the assumption that a cache doesn’t require an existing cache line to be flushed. However few more steps that are involved while flushing cache lines to make room for new data are elaborated later. Flow chart in Figure-1 describes the action sequence to serve Read and Write requests under all possible combinations of line states on local and remote caches.

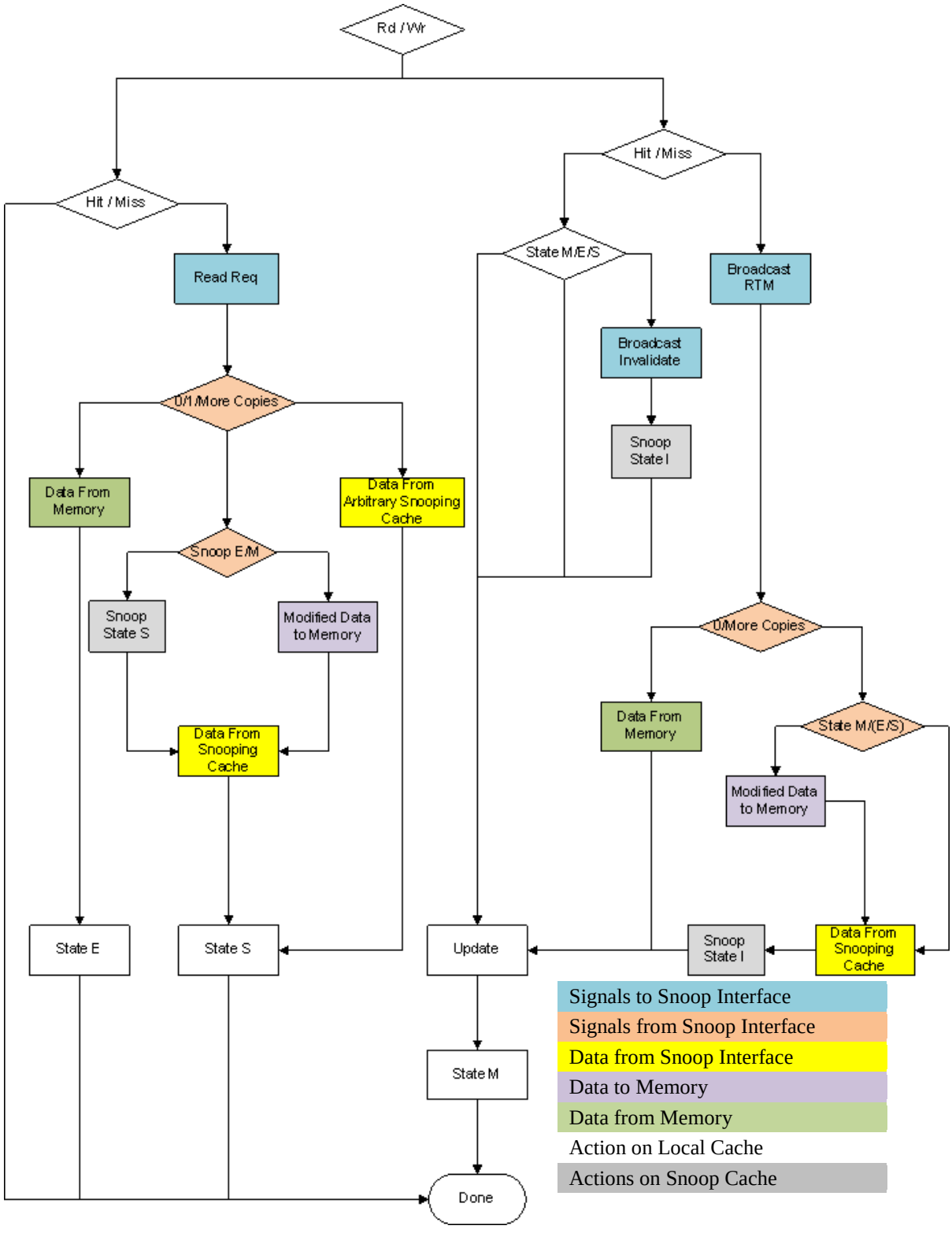

Figure 1 MESI Line State Change

A Cache line if found in a cache (State is not Invalid) can always satisfy a Read request.

When not found in cache, it has to be fetched from memory by placing a read request on bus. Other caches that are snooping the bus respond to the request if they hold the latest data, instead of having to wait on slower memory. The state changes are made accordingly on requesting and snooping caches. A Modified line on snooping cache is flushed to secondary memory before sending the data to the requesting cache.

If none of the caches have the requested data, the read request is finally serviced by the secondary memory.

For a write request, A Modified or Exclusive line on local cache can be directly modified changing the local state to Modified, this operation doesn't need any broadcasts.

When modifying a shared line, an Invalidate signal is broadcasted, to which all snooping caches respond by invalidating their corresponding shared lines.

If a write request generates a miss, RTM (Reading to Modify) is broadcasted on the bus, which informs the snooping caches to invalidate their copies after sending recent copy to requesting cache. If a snooping cache holds a Modified copy of the line, secondary memory is synced with the modified value. If none of the snooping caches are able to serve the request, the data is taken from secondary memory.

A write ends in Modified copy in local cache, which implies all other snooping caches finally invalidate the line.

Upon a Read or Write miss, if cache has no free locations for the new data, some victim line has to be flushed out to make room. An Exclusive victim can be flushed out without any broadcasts. A Modified victim should flush the data to secondary memory before invalidation. A victim can not be invalid by definition. A Shared victim is little tricky to handle, which depends on the number of remaining copies in other caches. Flushing a shared victim is achieved by broadcasting ISL (Invalidating Shared Line) followed by invalidating the shared line. If only one copy exists in snooping caches, the state is set to Exclusive on the remote cache that has the line. Otherwise statuses of all other shared copies remain untouched.

3. Implementation

The environment to simulate MESI cache coherence protocol on a Quad core environment is implemented as shown in block diagram below.

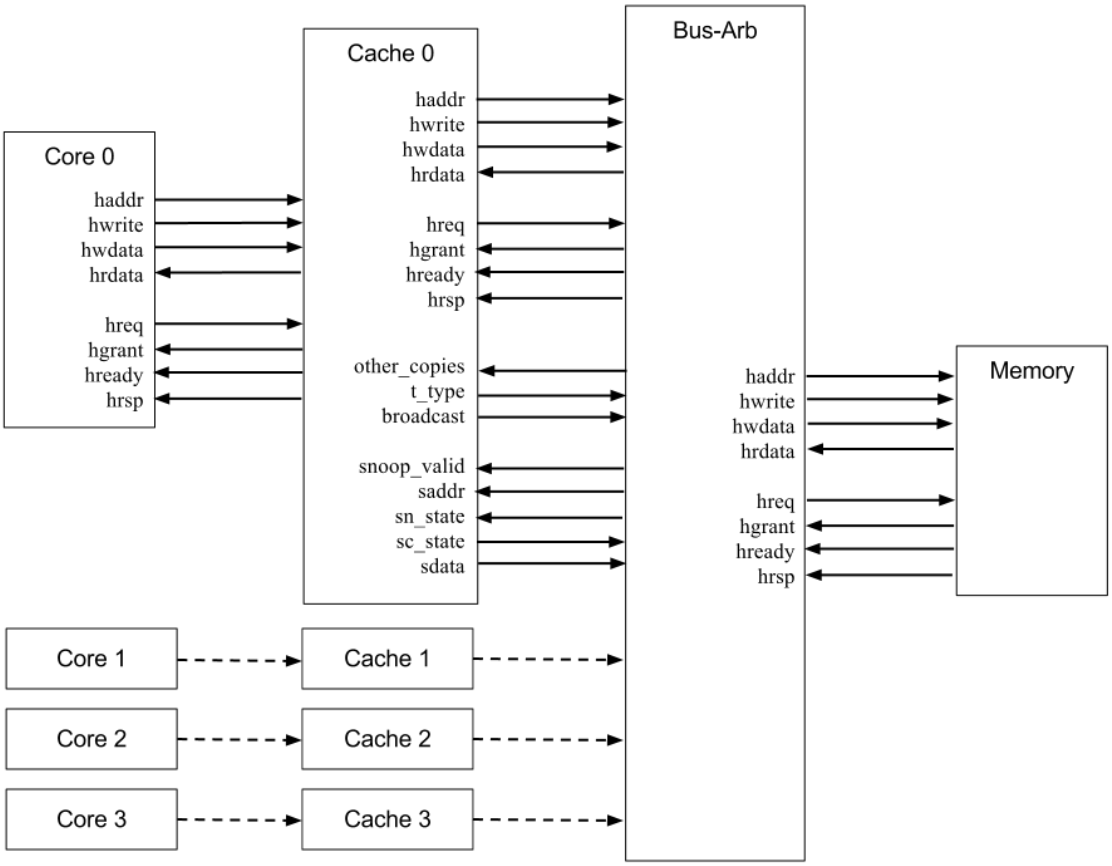

Figure 2 Block Diagram of Implementation

3.1. Specifications

A Symmetric Multiprocessing (SMP) (4) environment is considered for the implementation of MESI cache coherence protocol with four caches operating on single shared memory through AHB LITE bus protocol. An Arbiter manages the access to bus by assigning it to one cache requesting the bus at a given time. Specifications considered for the design of individual modules are as follows.

3.1.1. Core

All the four cores are identical to each other and initiate random memory transactions to which caches respond, imitating the behavior of real time general purpose processor cores. All transactions generated are one byte wide.

3.1.2. Memory

Slave Memory model responds to the requests from bus by serving the read and write requests. It also updates the secondary memory status as a result of the transactions made by the

Cores, which helps in debugging. Responses from memory slave model are associated with configurable delay to imitate the DRAMs in real world systems and the data size of responses is designed to be equal to cache line size, which gives a similar response time of caches that do not implement critical word first fetching (5).

3.1.3. Bus Protocol

All modules in the design communicate with each other fallowing a simplified version of AHB-LITE bus protocol (6), with varying data sizes depending on the modules interacting.

3.1.4. Bus Arbiter

A Bus Arbiter takes care of granting the bus to one of the four caches, following a round robin selection scheme among the caches that have asserted request for bus. Arbiter also takes care passing the snooping information to all snooping caches, without any delay by using combinational logic.

3.1.5. Cache

All caches used are identical to each other. A direct mapped write-back (5) cache is considered for the work for its simpler design and fast response time. A direct mapped cache is also very easy to debug and gives enhanced control compared more complex implementations that give better performance.

Status-Ram of the cache is two bit wide to hold the four possible states associated with a line in MESI system. Widths of Tag and Data rams and Sizes of all rams are configurable based on the address size, cache line width and number of cache lines which can be varied at build time.

Each cache has an AHB-Lite interface to interact with Core that pushes memory transactions on to it and one more AHB-Lite interface to interact with Arbiter / Slave memory and a snooping interface to get snooping interface to snoop bus activities of cache holding the bus and to broadcast signals to other snooping caches while it holds the bus.

AHB-Lite signals serve their purpose aligning to the standard (6), while few other signals which have special purposes are explained below.

3.1.5.1. Port: saddr

Address on the bus is passed on to snooping caches through this port.

3.1.5.2. Port: sc_state

Each snooping cache reports the local state of the line corresponding to the address on the bus through this port. If the line entry corresponding to the address on bus is not found in the snooping cache, **sc_state** shows an INVALID state.

3.1.5.3. Port: sdata

Each snooping cache puts the local copy of data corresponding to the cache line of address on the bus through this port. If the line entry corresponding to the address on bus is not found in the snooping cache, a zero is kept on this bus. Signal **sc_state** validates the data on **sdata**.

Driving zero when a value is not found makes the design simpler as a simple “bit-wise-or” operation on “**sdata**” from all snooping caches gives the valid data simultaneously while other combinational logic decides if at least one of the copies has a valid value, saving a few clock cycles.

3.1.5.4. Port: other_copies

The cache holding the bus needs to know whether or not other caches have copies of the line currently being processed; many times this is to decide between the Exclusive and Shared status for the cache line. The signal goes high if at least one of the other caches reports its current state of line corresponding to the address on bus different from Invalid. This signal is driven combinational depending on other combinational signals, hence is not associated with any delay.

3.1.5.5. Port: broadcast

Caches that wish to broadcast signals to other caches about invalidation shared lines etc… will communicate through **broadcast** port. The broadcast is asserted for one clock cycle and goes low.

3.1.5.6. Port: sn_state

Information broadcasted by cache is passed to snooping caches through a combinational block which decodes the broadcast and drives the **sn_state** of snooping caches with the next state of line. For instance, a RTM (Reading to Modify) signal forces corresponding line on snooping caches into Invalid state.

3.1.5.7. Port: snoop_valid

Unlike the **sc_state** and **sdata** which just share the information, **sn_state** changes the local state of the cache line. In order to protect this operation from unintentional modifications due to persistent address information on AHB-Lite bus, a **snoop_valid** signal is used, which validates the information on port **sn_state.**

3.1.5.8. Port: t_type

Once the bus is granted to a cache, it may perform varying number of operations depending on the local line state and various other factors. For instance for a Write operation that generated a cache miss, the cache needs to perform a memory read. If the cache line is filled with a modified victim, the line has to be flushed out to secondary memory before attempting a read. It's the requesting cache which can decide what type of transaction is required, and the arbiter needs to know this information to decide how long the grant should be held to requesting cache. This is achieved through signaling on port **t_type.**

4. Conclusion & Future Scope

A set of four Direct Mapped Caches interacting with each other according to MESI Cache Coherence Protocol is implemented along with supporting modules (Cores to initiate memory transactions and Slave Memory model to service the requests). All the results appear to meet the design criteria. There were unpredictable design errors on the way, but none stopped the system from functioning normally.

The results are measured using linear random distribution to generate the memory transactions, which differ greatly from the memory traffic generated by real time applications. To get more accurate results, standard benchmarks from various vendors can be used which may require few modifications in the design.

The work can be easily extended to implement similar Snooping based Cache coherence protocols reusing other parts of environment. Thus allowing evaluation of Different Protocols in a identical environment.

Sample results are documented in Appendix – C.

Appendixes

A – Code (Cache)

// Source code of direct mapped cache.

```
typedef enum bit [3:0] {
  IDLE,   // IDLE
  DELAY,  // Use to delay for one clk cycle
  C_A,    // Read Addr from Core
  C_RD,   // Core Reads Data from Cache
  C_WD,   // Core Writes Data into Cache
  ADDR,   // ADDR Phase
  DATA,   // DATA Phase
  FV_A,   // Flush Victim Addr
  FV_D,   // Flush Victim Data
  ISLP,   // Invalidating Shared Line
  W_BUS   // Wait For Bus
} cache_state_t;

module cache (input clk, input rst, ahblite_if.slave_p core_ifh,
ahblite_if.master_p bus_ifh, snoop_if.master_p snoopm_ifh,
snoop_if.slave_p snoops_ifh);

reg [`LINE_SIZE - 1 : 0] [7:0]  data_ram  [`NUM_LINES-1:0];
reg [`TAG_WIDTH - 1 : 0]  tag_ram   [`NUM_LINES-1:0];
line_state_t        status_ram      [`NUM_LINES-1:0];

cache_state_t CUR_STATE, NXT_STATE;
reg [`ADDR_BUS_WIDTH - 1:0] addr;
opr_t opr;
logic broadcasting;

// Extract tag
function automatic logic [`TAG_WIDTH - 1:0] tag (input reg
[`ADDR_BUS_WIDTH - 1:0] myAddr);
  tag = myAddr [`ADDR_BUS_WIDTH - 1:`LOG2_LINE_SIZE + `LOG2_NUM_LINES];
endfunction

// Extract line
function automatic logic [`LOG2_NUM_LINES - 1:0] line (input reg
[`ADDR_BUS_WIDTH - 1:0] myAddr);
  line = myAddr [`LOG2_LINE_SIZE + `LOG2_NUM_LINES -
1:`LOG2_LINE_SIZE];
endfunction

// Extract offset
function automatic logic [`LOG2_LINE_SIZE - 1:0] offset (input reg
[`ADDR_BUS_WIDTH - 1:0] myAddr);
  offset = myAddr [`LOG2_LINE_SIZE - 1:0];
endfunction

// Miss or Hit
function automatic logic miss (input reg [`ADDR_BUS_WIDTH - 1:0]
myAddr);
  miss = ( (status_ram [line (myAddr)] === INVALID) || (tag_ram [line
(myAddr)] !== tag (myAddr)) );
endfunction
```

```
// need_bus => if
//        not (Read Hit -or- Write to Modified or Exclusive line) -or-
//        Slave addr Matches Snoop Addr => wait for some time
function automatic logic need_bus (input reg [`ADDR_BUS_WIDTH - 1:0]
myAddr, input opr_t myOpr);
  need_bus = (
    (snoops_ifh.snoop_valid && snoops_ifh.saddr === myAddr) ||
    miss (myAddr) ||
    (myOpr === WRITE && status_ram [line (myAddr)] === SHARED)
  );
endfunction

always @ (posedge clk) if (rst) begin

  core_ifh.hgrant <= '1;
  core_ifh.hready <= '0;
  core_ifh.hrdata <= '0;
  core_ifh.hrsp   <= '0;

  bus_ifh.hreq   <= '0;
  bus_ifh.haddr  <= '0;
  bus_ifh.hwdata <= '0;
  bus_ifh.hwrite <= READ;

  snoopm_ifh.t_type <= NT;
  snoopm_ifh.broadcast <= NB;
  broadcasting <= '0;

  snoops_ifh.sc_state <= INVALID;
  snoops_ifh.sdata <= '0;

  CUR_STATE <= IDLE;
  opr <= READ;
  addr <= '0;

  for (int i=0; i<`NUM_LINES; i++) begin
    status_ram[i] <= INVALID;
  end

end


// Master FSM
always @ (posedge clk) if (!rst) begin

  if (CUR_STATE == IDLE) begin

    if (core_ifh.hreq) begin
      core_ifh.hready <= '1;
      NXT_STATE <= C_A;
      CUR_STATE <= DELAY;
    end

  end else if (CUR_STATE == DELAY) begin

    CUR_STATE <= NXT_STATE;

  end else if (CUR_STATE == C_A) begin
```

```
    addr <= core_ifh.haddr;
    opr  <= core_ifh.hwrite;

    if ( (snoops_ifh.snoop_valid && core_ifh.haddr == snoops_ifh.saddr)
||
      need_bus (core_ifh.haddr, core_ifh.hwrite) ) begin

      core_ifh.hready <= '0;
      bus_ifh.hreq <= '1;
      CUR_STATE <= W_BUS;

    end else if (core_ifh.hwrite) begin
      CUR_STATE <= C_WD;
    end else begin
      CUR_STATE <= C_RD;
    end

  end else if (CUR_STATE == C_WD) begin

    data_ram [line (addr)] [offset (addr)] <= core_ifh.hwdata;
    status_ram [line (addr)] <= MODIFIED;
    CUR_STATE <= IDLE;

  end else if (CUR_STATE == C_RD) begin

    core_ifh.hrdata <= data_ram [line (addr)] [offset (addr)];
    CUR_STATE <= IDLE;

  end else if (CUR_STATE == W_BUS) begin

    if (! need_bus (addr, opr)) begin

      if (opr == WRITE) CUR_STATE <= C_WD; else CUR_STATE <= C_RD;

    end else if (bus_ifh.hready && bus_ifh.hgrant) begin

      if (miss (addr)) begin

        // We got a miss and need to Flush Victim
        if (status_ram [line (addr)] == MODIFIED) begin

          // Flush without any Broadcasts
          bus_ifh.haddr <= {tag_ram [line (addr)], line (addr)};
          bus_ifh.hwrite <= WRITE;
          CUR_STATE <= FV_A;

          snoopm_ifh.t_type <= opr === WRITE ? F_RB : F_R;

        end else if (status_ram [line (addr)] == SHARED) begin

          // Broadcasting Invalidating Shared line
          bus_ifh.haddr <= {tag_ram [line (addr)], line (addr)};
          snoopm_ifh.broadcast <= ISL;
          broadcasting <= '1;
          status_ram [line (addr)] <= INVALID;
          CUR_STATE <= ISLP;
```

```
        snoopm_ifh.t_type <= opr === WRITE ? B_RB : B_R;

      end else begin

        // Exclusive or Invalid or Shared do not need flushing
        status_ram [line (addr)] <= INVALID;

        bus_ifh.haddr <= {tag (addr), line (addr)};
        bus_ifh.hwrite <= READ;
        CUR_STATE <= ADDR;

        snoopm_ifh.t_type <= opr === WRITE ? RB : R;
        broadcasting <= '1;
        snoopm_ifh.broadcast <= opr == WRITE ? RTM : RD;

      end

    end else begin

      // Trying to Modify Shared line => Broadcast Invalidate
      snoopm_ifh.broadcast <= IVDATE;
      broadcasting <= '1;
      bus_ifh.haddr <= addr;

      data_ram [line (addr)] [offset (addr)] <= core_ifh.hwdata;
      status_ram [line (addr)] <= MODIFIED;
      CUR_STATE <= IDLE;

      snoopm_ifh.t_type <= B;

    end

  end

end else if (CUR_STATE == FV_A) begin

  bus_ifh.hwdata <= data_ram [line (addr)];
  status_ram [line (addr)] <= INVALID;
  CUR_STATE <= FV_D;

end else if (CUR_STATE == FV_D || CUR_STATE == ISLP) begin

  if (bus_ifh.hready) begin

    bus_ifh.haddr <= addr;
    bus_ifh.hwrite <= READ;
    CUR_STATE <= ADDR;

    broadcasting <= '1;
    snoopm_ifh.broadcast <= opr == WRITE ? RTM : RD;

  end

end else if (CUR_STATE == ADDR) begin

  CUR_STATE <= DATA;

end else if (CUR_STATE == DATA) begin
```

```
    if (bus_ifh.hready) begin

      data_ram [line (addr)]   <= bus_ifh.hrdata;
      status_ram [line (addr)] <= opr === WRITE ? MODIFIED :
(snoopm_ifh.other_copies ? SHARED : EXCLUSIVE);
      tag_ram [line (addr)]    <= tag (addr);
      CUR_STATE <= IDLE;

    end

  end

end


// Snooper COMBO
always @* if (!rst) begin

  if (miss (snoops_ifh.saddr << `LOG2_LINE_SIZE)) begin
    snoops_ifh.sc_state <= INVALID;
    snoops_ifh.sdata <= '0;
  end else begin
    snoops_ifh.sc_state <= status_ram [line (snoops_ifh.saddr <<
`LOG2_LINE_SIZE)];
    snoops_ifh.sdata  <= data_ram [line (snoops_ifh.saddr <<
`LOG2_LINE_SIZE)];
  end

end

// Snooper FSM
always @ (posedge clk) if (!rst && snoops_ifh.snoop_valid &&
!(miss(snoops_ifh.saddr))) status_ram [line (snoops_ifh.saddr)] <=
snoops_ifh.sn_state;

always @ (posedge clk) if (!rst && broadcasting) begin
  broadcasting <= '0;
  snoopm_ifh.broadcast <= NB;
end

endmodule : cache
```

B - Sample Traffic

```
5         :                                C <- C#3
5         :                     C <- C#2
5         :           C <- C#1
5         : C <- C#0
85        :                     C -> C#2
85        :           C -> C#1
85        : C -> C#0
95        :                                C -> C#3
115       :                     C <- C#2
115       :           C <- C#1
115       : C <- C#0
125       :                                C <- C#3
155       :           C -> C#1
185       : C -> C#0
195       :                                C -> C#3
205       :                     C -> C#2
335       :           C <- C#1
415       :           C -> C#1
445       :                     C <- C#2
485       :                     C -> C#2
535       :                                C <- C#3
565       :                                C -> C#3
585       : C <- C#0
655       :           C <- C#1
675       :           C -> C#1
685       : C -> C#0
705       : C <- C#0
765       : C -> C#0
795       :                                C <- C#3
845       :                                C -> C#3
955       :           C <- C#1
1005      :           C -> C#1
1025      :                     C <- C#2
1075      :                     C -> C#2
1095      :                     C <- C#2
1115      :                                C <- C#3
1115      :                     C -> C#2
1185      : C <- C#0
1215      :                                C -> C#3
1255      : C -> C#0
1275      :           C <- C#1
1285      : C <- C#0
1295      :           C -> C#1
1315      :           C <- C#1
1335      :           C -> C#1
1335      : C -> C#0
1515      : C <- C#0
1565      : C -> C#0
1675      :                     C <- C#2
1745      :                                C <- C#3
1765      :                     C -> C#2
1785      :                     C <- C#2
1835      :                                C -> C#3
1845      :                     C -> C#2
1865      :                                C <- C#3
```

```
1865     : C <- C#0
1925     :                                C -> C#3
1945     :           C <- C#1
1945     : C -> C#0
2025     :                     C <- C#2
2045     :           C -> C#1
2075     :           C <- C#1
2105     :                     C -> C#2
2135     :           C -> C#1
2395     :                     C <- C#2
2425     :                     C -> C#2
2455     :                                C <- C#3
2455     :                     C <- C#2
2535     :                     C -> C#2
2535     : C <- C#0
2545     :                                C -> C#3
2575     :                                C <- C#3
2595     : C -> C#0
2615     :           C <- C#1
2625     :                                C -> C#3
2625     : C <- C#0
2655     :                                C <- C#3
2655     :           C -> C#1
2685     :           C <- C#1
2685     : C -> C#0
2705     :                                C -> C#3
2705     :                     C <- C#2
2705     :           C -> C#1
2725     :                                C <- C#3
2815     :                     C -> C#2
2835     :                                C -> C#3
2915     : C <- C#0
2945     : C -> C#0
2965     : C <- C#0
3025     : C -> C#0
3225     :           C <- C#1
3235     :           C -> C#1
3265     :           C <- C#1
3315     :                     C <- C#2
3335     :                     C -> C#2
3355     :                     C <- C#2
3355     :           C -> C#1
3415     :                                C <- C#3
3435     :                     C -> C#2
3495     : C <- C#0
3505     :                                C -> C#3
3535     :                                C <- C#3
3555     : C -> C#0
3565     :           C <- C#1
3575     :                                C -> C#3
3575     : C <- C#0
3645     :                     C <- C#2
3655     : C -> C#0
3675     :           C -> C#1
3675     : C <- C#0
3685     :                     C -> C#2
3745     : C -> C#0
3845     : C <- C#0
```

```
3925     : C -> C#0
4015     :                    C <- C#2
4035     :                              C <- C#3
4065     :                              C -> C#3
4065     :                    C -> C#2
4085     :                              C <- C#3
4145     :          C <- C#1
4175     :                              C -> C#3
4195     :                              C <- C#3
4215     :                              C -> C#3
4225     :          C -> C#1
4235     :                    C <- C#2
4245     :                              C <- C#3
4245     :          C <- C#1
4265     :                    C -> C#2
4265     :          C -> C#1
4315     : C <- C#0
4335     :                              C -> C#3
4345     : C -> C#0
4375     : C <- C#0
4385     : C -> C#0
4425     :          C <- C#1
4535     :          C -> C#1
4635     :          C <- C#1
4725     :                    C <- C#2
4725     :          C -> C#1
4785     :                    C -> C#2
4795     :                              C <- C#3
4805     :                    C <- C#2
4815     :                              C -> C#3
4835     :                              C <- C#3
4895     : C <- C#0
4915     :                              C -> C#3
4915     :                    C -> C#2
4945     : C -> C#0
4965     : C <- C#0
```

C – Results

The tables below give the average number of cycles taken for individual cores to complete memory transactions. Memory latency of 50 Cycles is emulated by the slave memory model, which is far less than the real time DRAM latencies. The figures also include the cycles spent for Handshaking of the AHB LITE protocol between Core-Cache and Cache-Bus-Memory, which are hidden in real processors by pipeline implementations. Entire data traffic is randomly generated following linear random distribution. Memory access from each core is controlled to eliminate the effect of Capacity misses on the performance evaluations. All the averages are calculated for about 7000 random memory transactions.

Table 1 gives the information of average number of cycles spent for transactions, when the Bus is reserved for Core 0. This gives an estimation of performance of identical cache in single core environment. Since Bus is reserved for Core 0, other cores can never complete any memory transactions and their caches are always empty (Do not contain any valid data). This ensures zero coherence traffic.

Core 0	Core 1	Core 2	Core 3
2.754205	-	-	-
2.76657	-	-	-
2.783867	-	-	-
2.75858	-	-	-
2.725893	-	-	-
2.714413	-	-	-
2.721796	-	-	-
2.781004	-	-	-
2.757105	-	-	-
2.717109	-	-	-

Table 1 Bus Grant only for Core-0 (Equivalent to Single Core Processor)

Table 2 tabulates the average number of cycles spent by each Core for memory transactions when all the cores are operating simultaneously on independent memories. This measures the performance of Caches in the absence of protocol traffic and in the presence of bus demands from other cores.

Core 0	Core 1	Core 2	Core 3
4.393331	4.317615	4.375273	4.359905
4.44578	4.357526	4.336783	4.373188
4.445622	4.357055	4.332029	4.340443
4.494352	4.427033	4.328861	4.40138
4.553196	4.449404	4.419367	4.369966
4.489198	4.389555	4.377677	4.340294
4.402842	4.374351	4.434646	4.39912
4.436949	4.352038	4.358871	4.358719

4.456125	4.327379	4.412269	4.351271
4.517159	4.301465	4.292144	4.365331

Table 2 Cores operating on independent memory (No Shared Data &No Protocol Traffic on bus)

Table 3 records the average cycles spent by enabling shared memory and by forcing the all the Cores to use the same memory region. The average number of cycles spent is reduced by around 1.2 cycles. This is due to data availability in other snooping caches removes the need to access main memory many times. However due to the additional memory transactions initiated while trying to maintain synchronization between caches, the average time spent is more than the first case (Single core operating).

Core 0	Core 1	Core 2	Core 3
3.173935	3.195465	3.175931	3.249395
3.179376	3.165291	3.188241	3.221095
3.146405	3.163443	3.149112	3.174747
3.164577	3.188275	3.272355	3.18127
3.231017	3.247837	3.250426	3.235735
3.22439	3.196449	3.249821	3.20452
3.169249	3.097354	3.176235	3.141675
3.079199	3.110741	3.098454	3.140193
3.229027	3.230289	3.215663	3.243619
3.139687	3.128817	3.205475	3.161095

Table 3 Cores operating on Shared Memory without Capacity Misses

References

1. **Wulf, W and McKee, S.** *Hitting the Memory Wall: Implications of the Obvious.* s.l. : ACM SIGArch Computer Architecture News, March 1995. vol. 23, pp. 20–24.

2. MESI protocol. *Wikipedia*. [Online] [Cited: 11 28, 2012.] http://en.wikipedia.org/wiki/MESI_protocol.

3. *A low-overhead coherence solution for multiprocessors with private cache memories.* **J. H, Patel and M. S, Papamarcos.** s.l. : Proc. 11th Annual Int. Symp. On Computer Architecture, June 1984. pp. 348-354.

4. **Lina J, Karam, et al.** TRENDS IN MULTI-CORE DSP PLATFORMS. *IEEE Signal Processing Magazine.* Nov - 2009, Special Issue on Signal Processing on Platforms with Multiple Cores.

5. **John L, Hennessy and David A, Patterson.** Computer Architecture: A Quantitative Approach. Fourth Edition. Pages 299-300.

6. AHB LITE Protocol. *University of Michigan.* [Online] [Cited: 12 02, 2012.] http://www.eecs.umich.edu/eecs/courses/eecs373/readings/ARM_IHI0033A_AMBA_AHB-Lite_SPEC.pdf.

7. **Adve, Saritha V. and Kourosh Fharachorloo.** *Shared Memory Consistency Models: A Tutorial.* s.l. : Rice University, September 1995. ECE Technical Report 9512.