5.9 Advanced Material: Implementing Cache Controllers

The section starts with the SystemVerilog of the cache controller from Section 5.7 in eight figures. It then goes into details of an example cache coherency protocol and the difficulties in implementing such a protocol.

SystemVerilog of a Simple Cache Controller

The hardware description language we are using in this section is SystemVerilog. The biggest change from prior versions of Verilog is that it borrows structures from C to make the code easier to read. Figures 5.9.1 through 5.9.8 show the SystemVerilog description of the cache controller.

```
package cache_def;
  // data structures for cache tag & data

  parameter int TAGMSB = 31;    //tag msb
  parameter int TAGLSB = 14;    //tag lsb

  //data structure for cache tag
  typedef struct packed {
    bit    valid;               //valid bit
    bit    dirty;               //dirty bit
    bit [TAGMSB:TAGLSB]tag;     //tag bits
  }cache_tag_type;

  //data structure for cache memory request
  typedef struct {
    bit [9:0]index;             //10-bit index
    bit    we;                  //write enable
  }cache_req_type;

  //128-bit cache line data
  typedef bit [127:0]cache_data_type;
```

FIGURE 5.9.1 Type declarations in SystemVerilog for the cache tags and data. The tag field is 18 bits wide and the index field is 10 bits wide, while a 2-bit field (bits 3–2) is used to index the block and select the word from the block. The rest of the type declaration is found in the following figure.

```
    // data structures for CPU<->Cache controller interface

    // CPU request (CPU->cache controller)
    typedef struct {
      bit [31:0]addr;             //32-bit request addr
      bit [31:0]data;             //32-bit request data (used when write)
      bit rw;                     //request type : 0 = read, 1 = write
      bit valid;                  //request is valid
    }cpu_req_type;

    // Cache result (cache controller->cpu)
    typedef struct {
      bit [31:0]data;             //32-bit data
      bit ready;                  //result is ready
    }cpu_result_type;

    //----------------------------------------------------------------------
    // data structures for cache controller<->memory interface

    // memory request (cache controller->memory)
    typedef struct {
      bit [31:0]addr;             //request byte addr
      bit [127:0]data;            //128-bit request data (used when write)
      bit rw;                     //request type : 0 = read, 1 = write
      bit valid;                  //request is valid
    }mem_req_type;

    // memory controller response (memory -> cache controller)
    typedef struct {
      cache_data_type data;       //128-bit read back data
      bit    ready;               //data is ready
    }mem_data_type;

  endpackage
```

FIGURE 5.9.2 Type declarations in SystemVerilog for the CPU-cache and cache-memory interfaces. These are nearly identical except that the data is 32 bits wide between the CPU and cache and is 128 bits wide between the cache and memory.

Figures 5.9.1 and 5.9.2 declare the structures that are used in the definition of the cache in the following figures. For example, the cache tag structure (`cache_tag_type`) contains a valid bit (`valid`), a dirty bit (`dirty`), and an 18-bit tag field (`[TAGMSB:TAGLSB] tag`). Figure 5.9.3 shows the block diagram of the cache using the names from the Verilog description.

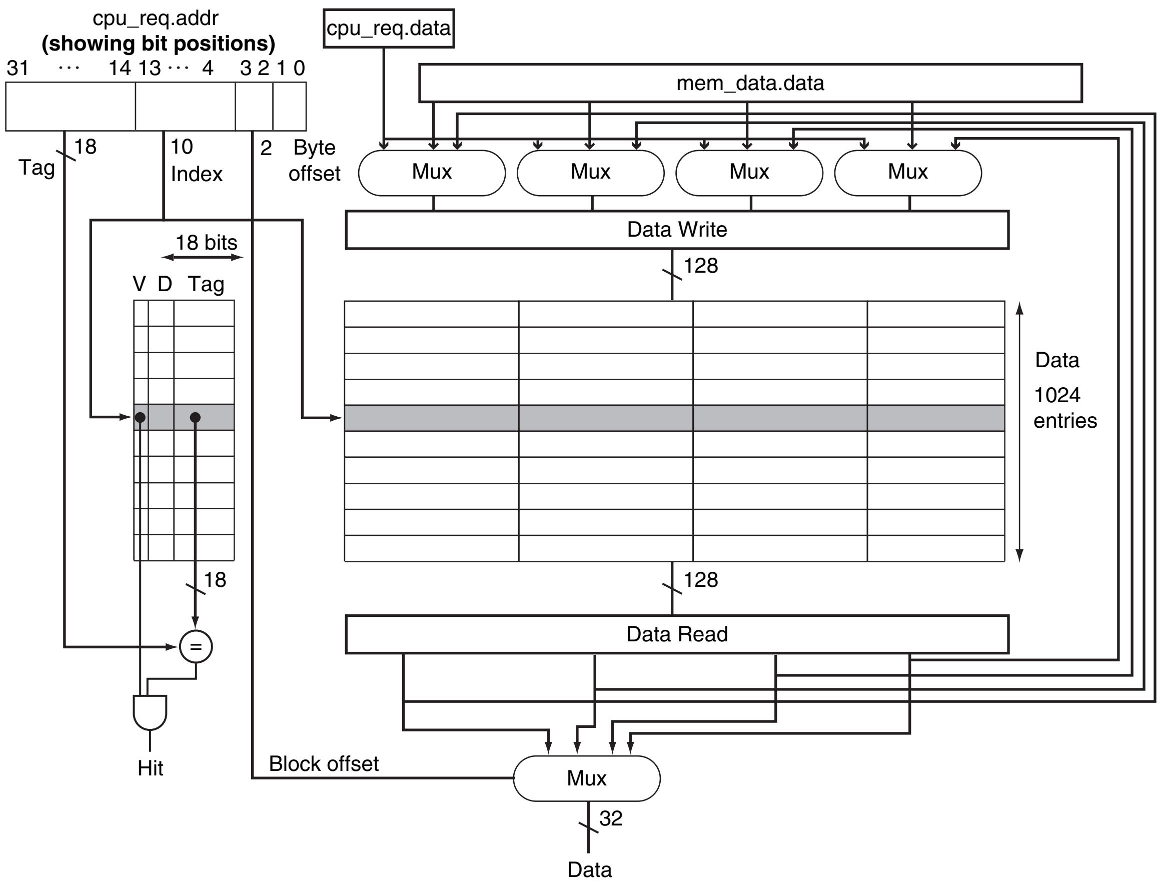

FIGURE 5.9.3 Block diagram of the simple cache using the Verilog names. Not shown are the write enables for the cache tag memory and for the cache data memory, or the control signals for multiplexors that supply data for the Data Write variable. Rather than have separate write enables on every word of the cache data block, the Verilog reads the old value of the block into Data Write and then updates the word in that variable on a write. It then writes the whole 128-bit block.

Figure 5.9.4 defines modules for the cache data (`dm_cache_data`) and cache tag (`dm_cache_tag`). These memories can be read at any time, but writes only occur on the positive clock edge (`posedge(clk)`) and only if write enable is a 1 (`data_req.we` or `tag_req.we`).

Figure 5.9.5 defines the inputs, outputs, and states of the FSM. The inputs are the requests from the CPU (cpu_req) and responses from memory (mem_data), and the outputs are responses to the CPU (cpu_res) and requests to memory (mem_req). The figure also declares the internal variables needed by the FSM. For example, the current state and next state registers of the FSM are rstate and vstate, respectively.

Figure 5.9.6 lists the default values of the control signals, including the word to be read or written from a block, setting the cache write enables to 0, and so on. These values are set every clock cycle, so the write enable for a portion of the cache—for example, tag_req.we—would be set to 1 for one clock cycle in the figures below and then would be reset to 0 according to the Verilog in this figure.

The last two figures show the FSM as a large case statement (case(rstate)), with the four states splits across the two figures. Figure 5.9.7 starts with the Idle state (idle), which simply goes to the Compare Tag state (compare_tag) if the CPU makes a valid request. It then describes most of the Compare Tag state. The Compare Tag state checks to see if the tags match and the entry is valid. If so, then it first sets the Cache Ready signal (v_cpu_res.ready). If the request is a write, it sets the tag field, the valid bit, and the dirty bit. The next state is Idle. If it is a miss, then the state prepares to change the tag entry and valid and dirty bits. If the block to be replaced is clean or the invalid, the next state is Allocate.

Figure 5.9.8 continues the Compare Tag state. If the block to be replaced is dirty, then the next state is Write-Back. The figure shows the Allocate state (allocate) next, which simply reads the new block. It keeps looping until the memory is ready; when it is, it goes to the Compare Tag state. This is followed in the figure by the Write-Back state (write_back). As the figure shows, the Write-Back state merely writes the dirty block to memory, once again looping until memory is ready. When memory is ready, indicating the write is complete, we go to the Allocate state.

The code at the end sets the current state from the next state or resets the FSM to the Idle state on the next clock edge, depending on a reset signal (rst).

The CD includes a Test Case module that will be useful to check the code in these figures. This SystemVerilog could be used to create a cache and cache controller in an FPGA.

```
/*cache: data memory, single port, 1024 blocks*/
module dm_cache_data(input  bit clk,
    input  cache_req_type  data_req,//data request/command, e.g. RW, valid
    input  cache_data_type data_write, //write port (128-bit line)
    output cache_data_type data_read); //read port
  timeunit 1ns; timeprecision 1ps;

  cache_data_typedata_mem[0:1023];

  initial begin
    for (int i=0; i<1024; i++)
         data_mem[i] = ‘0;
  end

  assign data_read = data_mem[data_req.index];

  always_ff @(posedge(clk)) begin
    if (data_req.we)
      data_mem[data_req.index] <= data_write;
  end
endmodule

/*cache: tag memory, single port, 1024 blocks*/
module dm_cache_tag(input  bit clk, //write clock
    input  cache_req_type tag_req, //tag request/command, e.g. RW, valid
    input  cache_tag_type tag_write,//write port
    output cache_tag_type tag_read);//read port
  timeunit 1ns; timeprecision 1ps;

  cache_tag_typetag_mem[0:1023];

  initial begin
     for (int i=0; i<1024; i++)
     tag_mem[i] = ‘0;
  end

  assign tag_read = tag_mem[tag_req.index];

  always_ff @(posedge(clk)) begin
    if (tag_req.we)
      tag_mem[tag_req.index] <= tag_write;
  end

endmodule
```

FIGURE 5.9.4 Cache data and tag modules in SystemVerilog. These are nearly identical except that the data is 32 bits wide between the CPU and cache and is 128 bits wide between the cache and memory. Both only write on positive clock edges if the write enable is set.

```
/*cache finite state machine*/

module dm_cache_fsm(input  bit clk, input  bit rst,
       input  cpu_req_type   cpu_req,      //CPU request input (CPU->cache)
       input  mem_data_type  mem_data,     //memory response (memory->cache)
       output mem_req_type   mem_req,      //memory request (cache->memory)
       output cpu_result_type cpu_res      //cache result (cache->CPU)
    );

  timeunit 1ns;
  timeprecision 1ps;

  /*write clock*/
  typedef enum {idle, compare_tag, allocate, write_back} cache_state_type;

  /*FSM state register*/
  cache_state_typevstate, rstate;

  /*interface signals to tag memory*/
  cache_tag_typetag_read;                  //tag read result
  cache_tag_typetag_write;                 //tag write data
  cache_req_typetag_req;                   //tag request

  /*interface signals to cache data memory*/
  cache_data_typedata_read;                //cache line read data
  cache_data_typedata_write;               //cache line write data
  cache_req_typedata_req;                  //data req


  /*temporary variable for cache controller result*/
  cpu_result_typev_cpu_res;

  /*temporary variable for memory controller request*/
  mem_req_typev_mem_req;

  assign mem_req = v_mem_req;              //connect to output ports
  assign cpu_res = v_cpu_res;
```

FIGURE 5.9.5 FSM in SystemVerilog, part I. These modules instantiate the memories according to the type definitions in the previous figure.

```
always_comb begin

    /*-------------------------default values for all signals------------*/
    /*no state change by default*/
    vstate = rstate;
    v_cpu_res = '{0, 0}; tag_write = '{0, 0, 0};

    /*read tag by default*/
    tag_req.we = '0;
    /*direct map index for tag*/
    tag_req.index = cpu_req.addr[13:4];

    /*read current cache line by default*/
    data_req.we = '0;
    /*direct map index for cache data*/
    data_req.index = cpu_req.addr[13:4];

    /*modify correct word (32-bit) based on address*/
    data_write = data_read;
    case(cpu_req.addr[3:2])
    2'b00:data_write[31:0] = cpu_req.data;
    2'b01:data_write[63:32] = cpu_req.data;
    2'b10:data_write[95:64] = cpu_req.data;
    2'b11:data_write[127:96] = cpu_req.data;
    endcase

    /*read out correct word(32-bit) from cache (to CPU)*/
    case(cpu_req.addr[3:2])
    2'b00:v_cpu_res.data = data_read[31:0];
    2'b01:v_cpu_res.data = data_read[63:32];
    2'b10:v_cpu_res.data = data_read[95:64];
    2'b11:v_cpu_res.data = data_read[127:96];
    endcase

    /*memory request address (sampled from CPU request)*/
    v_mem_req.addr = cpu_req.addr;
    /*memory request data (used in write)*/
    v_mem_req.data = data_read;
    v_mem_req.rw = '0;
```

FIGURE 5.9.6 FSM in SystemVerilog, part II. This section describes the default value of all signals. The following figures will set these values for one clock cycle, and this Verilog will reset it to these values the following clock cycle.

```
//------------------------------------Cache FSM-------------------------
case(rstate)
/*idle state*/
idle : begin
    /*If there is a CPU request, then compare cache tag*/
    if (cpu_req.valid)
       vstate = compare_tag;
       end
/*compare_tag state*/
compare_tag : begin
          /*cache hit (tag match and cache entry is valid)*/
    if (cpu_req.addr[TAGMSB:TAGLSB] == tag_read.tag && tag_read.valid) begin
          v_cpu_res.ready = ‘1;

       /*write hit*/
       if (cpu_req.rw) begin
          /*read/modify cache line*/
          tag_req.we = ‘1; data_req.we = ‘1;

          /*no change in tag*/
          tag_write.tag = tag_read.tag;
          tag_write.valid = ‘1;
          /*cache line is dirty*/
          tag_write.dirty = ‘1;
       end

          /*xaction is finished*/
          vstate = idle;
    end
    /*cache miss*/
    else begin
      /*generate new tag*/
      tag_req.we = ‘1;
      tag_write.valid = ‘1;
      /*new tag*/
      tag_write.tag = cpu_req.addr[TAGMSB:TAGLSB];
      /*cache line is dirty if write*/
      tag_write.dirty = cpu_req.rw;

      /*generate memory request on miss*/
      v_mem_req.valid = ‘1;
      /*compulsory miss or miss with clean block*/
      if (tag_read.valid == 1’b0 || tag_read.dirty == 1’b0)
         /*wait till a new block is allocated*/
         vstate = allocate;
```

FIGURE 5.9.7 FSM in SystemVerilog, part III. Actual FSM states via case statement in this figure and the next. This figure has the Idle state and most of the Compare Tag state.

```
      else begin
      /*miss with dirty line*/
         /*write back address*/
         v_mem_req.addr = {tag_read.tag, cpu_req.addr[TAGLSB-1:0]};
         v_mem_req.rw = ‘1;
         /*wait till write is completed*/
         vstate = write_back;
      end
     end
 end
 /*wait for allocating a new cache line*/
 allocate: begin
     /*memory controller has responded*/
     if (mem_data.ready) begin
        /*re-compare tag for write miss (need modify correct word)*/
        vstate = compare_tag;
        data_write = mem_data.data;
        /*update cache line data*/
        data_req.we = ‘1;
     end
      end
 /*wait for writing back dirty cache line*/
 write_back : begin
     /*write back is completed*/
     if (mem_data.ready) begin
        /*issue new memory request (allocating a new line)*/
        v_mem_req.valid = ‘1;
        v_mem_req.rw = ‘0;

        vstate = allocate;
     end
   end
 endcase
end

always_ff @(posedge(clk)) begin
 if (rst)
   rstate <= idle;       //reset to idle state
 else
   rstate <= vstate;
end
/*connect cache tag/data memory*/
dm_cache_tag  ctag(.*);
dm_cache_data cdata(.*);
endmodule
```

FIGURE 5.9.8 FSM in SystemVerilog, part IV. Actual FSM states via case statement in prior figure and this one. This figure has last part of the Compare Tag state, plus Allocate and Write-Back states.

Basic Coherent Cache Implementation Techniques

The key to implementing an invalidate protocol is the use of the bus, or another broadcast medium, to perform invalidates. To invalidate, the processor simply acquires bus access and broadcasts the address to be invalidated on the bus. All processors continuously snoop on the bus, watching the addresses. The processors check whether the address on the bus is in their cache. If so, the corresponding data in the cache is invalidated.

When a write to a block that is shared occurs, the writing processor must acquire bus access to broadcast its invalidation. If two processors attempt to write shared blocks at the same time, their attempts to broadcast an invalidate operation will be serialized when they arbitrate for the bus. The first processor to obtain bus access will cause any other copies of the block it is writing to be invalidated. If the processors were attempting to write the same block, the serialization enforced by the bus also serializes their writes. One implication of this scheme is that a write to a shared data item cannot actually complete until it obtains bus access. All coherence schemes require some method of serializing accesses to the same cache block, either by serializing access to the communication medium or another shared structure.

In addition to invalidating outstanding copies of a cache block that is being written into, we also need to locate a data item when a cache miss occurs. In a write-through cache, it is easy to find the recent value of a data item, since all written data are always sent to the memory, from which the most recent value of a data item can always be fetched. In a design with adequate memory bandwidth to support the write traffic from the processors, using write-through simplifies the implementation of cache coherence.

For a write-back cache, finding the most recent data value is more difficult, since the most recent value of a data item can be in a cache rather than in memory. Happily, write-back caches can use the same snooping scheme both for cache misses and for writes: each processor snoops all addresses placed on the bus. If a processor finds that it has a dirty copy of the requested cache block, it provides that cache block in response to the read request and causes the memory access to be aborted. The additional complexity comes from having to retrieve the cache block from a processor's cache, which can often take longer than retrieving it from the shared memory if the processors are in separate chips. Since write-back caches generate lower requirements for memory bandwidth, they can support larger numbers of faster processors and have been the approach chosen in most multiprocessors, despite the additional complexity of maintaining coherence. Therefore, we will examine the implementation of coherence with write-back caches.

The normal cache tags can be used to implement the process of snooping, and the valid bit for each block makes invalidation easy to implement. Read misses, whether generated by an invalidation or by some other event, are also straightforward, since they simply rely on the snooping capability. For writes, we'd like to know whether any other copies of the block are cached, because if there are no other cached copies, the write need not be placed on the bus in a write-back cache. Not sending the write reduces both the time taken by the write and the required bandwidth.

To track whether or not a cache block is shared, we can add an extra state bit associated with each cache block, just as we have a valid bit and a dirty bit. By adding a bit indicating whether the block is shared, we can decide whether a write must generate an invalidate. When a write to a block in the shared state occurs, the cache generates an invalidation on the bus and marks the block as *exclusive*. No further invalidations will be sent by that processor for that block. The processor with the sole copy of a cache block is normally called the *owner* of the cache block.

When an invalidation is sent, the state of the owner's cache block is changed from shared to unshared (or exclusive). If another processor later requests this cache block, the state must be made shared again. Since our snooping cache also sees any misses, it knows when the exclusive cache block has been requested by another processor and the state should be made shared.

Every bus transaction must check the cache-address tags, which could potentially interfere with processor cache accesses. One way to reduce this interference is to duplicate the tags. The interference can also be reduced in a multilevel cache by directing the snoop requests to the L2 cache, which the processor uses only when it has a miss in the L1 cache. For this scheme to work, every entry in the L1 cache must be present in the L2 cache, a property called the *inclusion property*. If the snoop gets a hit in the L2 cache, then it must arbitrate for the L1 cache to update the state and possibly retrieve the data, which usually requires a stall of the processor. Sometimes it may even be useful to duplicate the tags of the secondary cache to further decrease contention between the processor and the snooping activity.

An Example Cache Coherency Protocol

A snooping coherence protocol is usually implemented by incorporating a finite-state controller in each node. This controller responds to requests from the processor and from the bus (or other broadcast medium), changing the state of the selected cache block, as well as using the bus to access data or to invalidate it. Logically, you can think of a separate controller being associated with each block; that is, snooping operations or cache requests for different blocks can proceed independently. In actual implementations, a single controller allows multiple operations to distinct blocks to proceed in interleaved fashion (that is, one operation may be initiated before another is completed, even though only one cache access or one bus access is allowed at a time). Also, remember that although we refer to a bus in the following description, any interconnection network that supports a broadcast to all the coherence controllers and their associated caches can be used to implement snooping.

The simple protocol we consider has three states: invalid, shared, and modified. The shared state indicates that the block is potentially shared, while the modified state indicates that the block has been updated in the cache; note that the modified state *implies* that the block is exclusive. Figure 5.9.9 shows the requests generated by the processor-cache module in a node (in the top half of the table) as well as those coming from the bus (in the bottom half of the table). This protocol is for a write-back cache, but it can be easily changed to work for a write-through cache by reinterpreting the modified state as an exclusive state and updating the cache on writes in the normal fashion for a write-through cache. The most common extension of this basic protocol is the addition of an exclusive state, which describes a block that is unmodified but held in only one cache; the caption of Figure 5.9.9 describes this state and its addition in more detail.

When an invalidate or a write miss is placed on the bus, any processors with copies of the cache block invalidate it. For a write-through cache, the data for a write miss can always be retrieved from the memory. For a write miss in a write-back cache, if the block is exclusive in just one cache, that cache also writes back the block; otherwise, the data can be read from memory.

Figure 5.9.10 shows a finite-state transition diagram for a single cache block using a write invalidation protocol and a write-back cache. For simplicity, the three states of the protocol are duplicated to represent transitions based on processor requests (on the left, which corresponds to the top half of the table in Figure 5.9.9), as opposed to transitions based on bus requests (on the right, which corresponds to the bottom half of the table in Figure 5.9.9). Boldface type is used to distinguish the bus actions, as opposed to the conditions on which a state transition depends. The state in each node represents the state of the selected cache block specified by the processor or bus request.

All of the states in this cache protocol would be needed in a uniprocessor cache, where they would correspond to the invalid, valid (and clean), and dirty states. Most of the state changes indicated by arcs in the left half of Figure 5.9.10 would be needed in a write-back uniprocessor cache, with the exception being the invalidate on a write hit to a shared block. The state changes represented by the arcs in the right half of Figure 5.9.10 are needed only for coherence and would not appear at all in a uniprocessor cache controller.

As mentioned earlier, there is only one finite-state machine per cache, with stimuli coming either from the attached processor or from the bus. Figure 5.9.11 shows how the state transitions in the right half of Figure 5.9.10 are combined with those in the left half of the figure to form a single state diagram for each cache block.

To understand why this protocol works, observe that any valid cache block is either in the shared state in one or more caches or in the exclusive state in exactly one cache. Any transition to the exclusive state (which is required for a processor to write to the block) requires an invalidate or write miss to be placed on the bus, causing all caches to make the block invalid. In addition, if some other cache had the block in exclusive state, that cache generates a write back, which supplies the block containing the desired address. Finally, if a read miss occurs on the bus to a block in the exclusive state, the cache with the exclusive copy changes its state to shared.

Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read hit	processor	shared or modified	normal hit	Read data in cache.
Read miss	processor	invalid	normal miss	Place read miss on bus.
Read miss	processor	shared	replacement	Address conflict miss: place read miss on bus.
Read miss	processor	modified	replacement	Address conflict miss: write-back block, then place read miss on bus.
Write hit	processor	modified	normal hit	Write data in cache.
Write hit	processor	shared	coherence	Place invalidate on bus. These operations are often called upgrade or *ownership* misses, since they do not fetch the data but only change the state.
Write miss	processor	invalid	normal miss	Place write miss on bus.
Write miss	processor	shared	replacement	Address conflict miss: place write miss on bus.
Write miss	processor	modified	replacement	Address conflict miss: write-back block, then place write miss on bus.
Read miss	bus	shared	no action	Allow memory to service read miss.
Read miss	bus	modified	coherence	Attempt to share data: place cache block on bus and change state to shared.
Invalidate	bus	shared	coherence	Attempt to write shared block; invalidate the block.
Write miss	bus	shared	coherence	Attempt to write block that is shared; invalidate the cache block.
Write miss	bus	modified	coherence	Attempt to write block that is exclusive elsewhere: write-back the cache block and make its state invalid.

FIGURE 5.9.9 The cache coherence mechanism receives requests from both the processor and the bus and responds to these based on the type of request, whether it hits or misses in the cache, and the state of the cache block specified in the request. The fourth column describes the type of cache action as normal hit or miss (the same as a uniprocessor cache would see), replacement (a uniprocessor cache replacement miss), or coherence (required to maintain cache coherence); a normal or replacement action may cause a coherence action depending on the state of the block in other caches. For read misses, write misses, or invalidates snooped from the bus, an action is required only if the read or write addresses match a block in the cache and the block is valid. Some protocols also introduce a state to designate when a block is exclusively in one cache but has not yet been written. This state can arise if a write access is broken into two pieces: getting the block exclusively in one cache and then subsequently updating it; in such a protocol this "exclusive unmodified state" is transient, ending as soon as the write is completed. Other protocols use and maintain an exclusive state for an unmodified block. In a snooping protocol, this state can be entered when a processor reads a block that is not resident in any other cache. Because all subsequent accesses are snooped, it is possible to maintain the accuracy of this state. In particular, if another processor issues a read miss, the state is changed from exclusive to shared. The advantage of adding this state is that a subsequent write to a block in the exclusive state by the same processor need not acquire bus access or generate an invalidate, since the block is known to be exclusively in this cache; the processor merely changes the state to modified. This state is easily added by using the bit that encodes the coherent state as an exclusive state and using the dirty bit to indicate that a block is modified. The popular MESI protocol, which is named for the four states it includes (modified, exclusive, shared, and invalid), uses this structure. The MOESI protocol introduces another extension: the "owned" state.

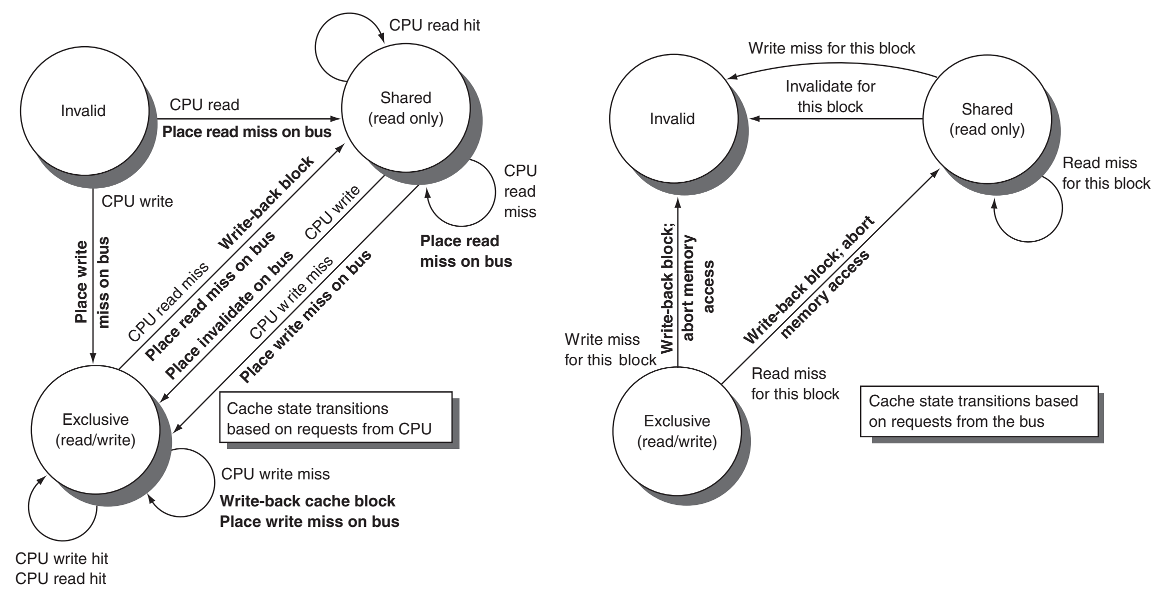

FIGURE 5.9.10 A write-invalidate, cache-coherence protocol for a write-back cache, showing the states and state transitions for each block in the cache. The cache states are shown in circles, with any access permitted by the processor without a state transition shown in parentheses under the name of the state. The stimulus causing a state change is shown on the transition arcs in regular type, and any bus actions generated as part of the state transition are shown on the transition arc in bold. The stimulus actions apply to a block in the cache, not to a specific address in the cache. Hence, a read miss to a block in the shared state is a miss for that cache block but for a different address. The left side of the diagram shows state transitions based on actions of the processor associated with this cache; the right side shows transitions based on operations on the bus. A read miss in the exclusive or shared state and a write miss in the exclusive state occur when the address requested by the processor does not match the address in the cache block. Such a miss is a standard cache replacement miss. An attempt to write a block in the shared state generates an invalidate. Whenever a bus transaction occurs, all caches that contain the cache block specified in the bus transaction take the action dictated by the right half of the diagram. The protocol assumes that memory provides data on a read miss for a block that is clean in all caches. In actual implementations, these two sets of state diagrams are combined. In practice, there are many subtle variations on invalidate protocols, including the introduction of the exclusive unmodified state, as to whether a processor or memory provides data on a miss.

The actions in gray in Figure 5.9.11, which handle read and write misses on the bus, are essentially the snooping component of the protocol. One other property that is preserved in this protocol, and in most other protocols, is that any memory block in the shared state is always up to date in the memory, which simplifies the implementation.

Although our simple cache protocol is correct, it omits a number of complications that make the implementation much trickier. The most important of these is that the protocol assumes that operations are *atomic*—that is, an operation can be done in such a way that no intervening operation can occur. For example, the protocol described assumes that write misses can be detected, acquire the bus, and receive a response as a single atomic action. In reality, this is not true. Similarly, if we used

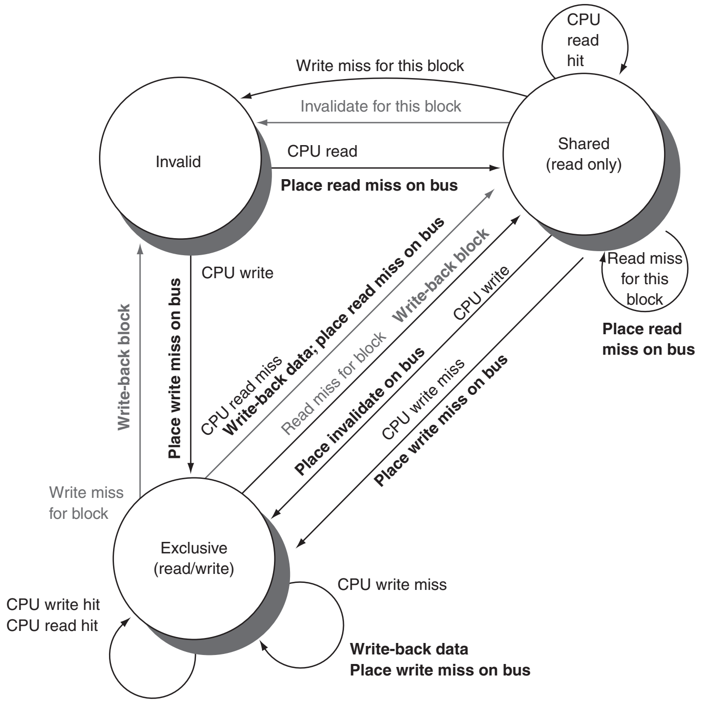

FIGURE 5.9.11 Cache coherence state diagram with the state transitions induced by the local processor shown in black and by the bus activities shown in gray. As in Figure 5.9.10, the activities on a transition are shown in bold.

a switch, as all recent multiprocessors do, then even read misses would also not be atomic.

Nonatomic actions introduce the possibility that the protocol can *deadlock*, meaning that it reaches a state where it cannot continue. We will explore how these protocols are implemented without a bus shortly.

Constructing small-scale (two to four processors) multiprocessors has become very easy. For example, the Intel Nehalem and AMD Opteron processors are designed for use in cache-coherent multiprocessors and have an external interface that supports snooping and allows two to four processors to be directly connected. They also have larger on-chip caches to reduce bus utilization. In the case of the Opteron processors, the support for interconnecting multiple processors is integrated onto the processor chip, as are the memory interfaces. In the case of the Intel design, a two-processor system can be built with only a few additional external chips to interface with the memory system and I/O. Although these designs cannot be easily scaled to larger processor counts, they offer an extremely cost-effective solution for two to four processors.

The devil is in the details.

Classic proverb

Implementing Snoopy Cache Coherence

As we said earlier, the major complication in actually implementing the snooping coherence protocol we have described is that write and upgrade misses are not atomic in any recent multiprocessor. The steps of detecting a write or upgrade miss; communicating with the other processors and memory; getting the most recent value for a write miss and ensuring that any invalidates are processed; and updating the cache cannot be done as if they took a single cycle.

In a simple single-bus system, these steps can be made effectively atomic by arbitrating for the bus first (before changing the cache state) and not releasing the bus until all actions are complete. How can the processor know when all the invalidates are complete? In most bus-based multiprocessors, a single line is used to signal when all necessary invalidates have been received and are being processed. Following that signal, the processor that generated the miss can release the bus, knowing that any required actions will be completed before any activity related to the next miss. By holding the bus exclusively during these steps, the processor effectively makes the individual steps atomic.

In a system without a bus, we must find some other method of making the steps in a miss atomic. In particular, we must ensure that two processors that attempt to write the same block at the same time, a situation which is called a *race*, are strictly ordered: one write is processed before the next is begun. It does not matter which of two writes in a race wins the race, just that there be only a single winner whose coherence actions are completed first. In a snoopy system, ensuring that a race has only one winner is accomplished by using broadcast for all misses, as well as some basic properties of the interconnection network. These properties, together with the ability to restart the miss handling of the loser in a race, are the keys to implementing snoopy cache coherence without a bus.