Mor1kx Cappuccino's coherency implementation

Simone Disabato, Matteo Fusi, Stefano Longari

2016-12-4

1 Introduction

1.1 How to read the document

The document is divided in three main sections.
The first is a brief introduction about what is mor1kx and the aim of the project, the Implementation section is a conversational description of all the process that has been necessary to reach the goals while the last section is a more in depth and technical analysis of each core module that has been changed.
The reader is encouraged to read everything from start to finish only if he/she doesn't already have any informations about the project. If this is not the case he/she can avoid the introduction and, if the aim is to discover the meaning of some code lines the suggestion is to directly check the last section.

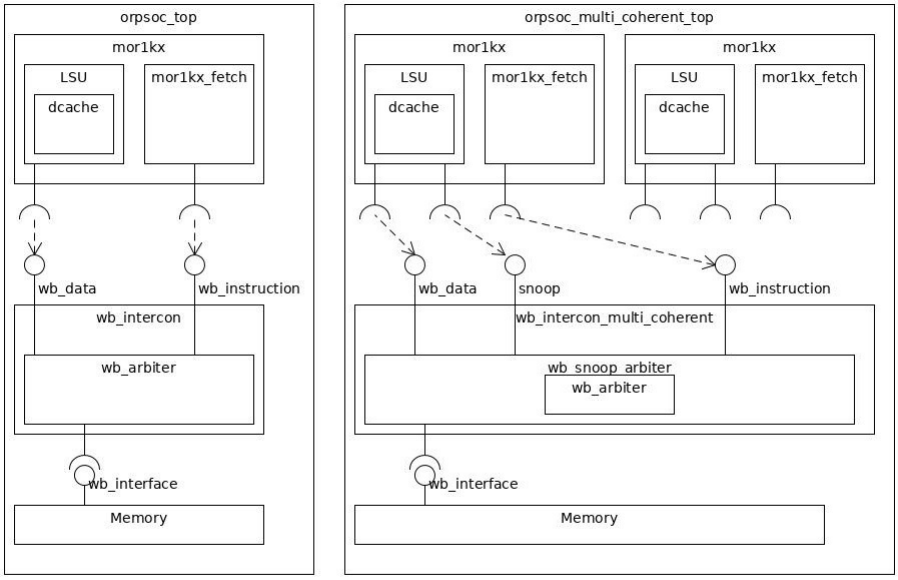
1.2 mor1kx

What we started from was the mor1kx version of the OpenRisc family of instruction set architectures. OpenRisc is a free and open source architecture with different implementations and the one we worked on, Mor1kx, is a core implementation that enables different variants in respect to the pipeline stages and gives the the user a vast choice of settings and possibilities. In particular Mor1kx gives the user the choice between three different cpu implementations: Cappuccino, Espresso and Prontoespresso.
We aimed to modify Cappuccino which is the six pipeline stages and supports data cache. More precisely there was already the possibility to use data caches for a single core cpu while if more than one core was implemented cache coherency was not fully implemented. There was only a simple cache for each core, with a write through implementation on the LSU that modified the data in memory as soon as it was modified in the core and the beginning of some kind of snoop implementation, with a snoop ram in the data cache to save snooped data addresses.
The LSU and the data cache were implemented with a finite state machine that enables jumps from a state to the other, usually passing through idle state, to generally enable a write, read or refill situation.

1.3 Goals

Aim of the project was to enable coherency between two or more processors so that caches could have been implemented even in a multi core situation.
The way we acted on the already existing implementation focused mostly on three elements, the LSU the data cache and a new coherence module created around the already existing bus arbiter.
In particular the new component was necessary as the bus as it was in the unmodified mor1kx version was not working as a controller of data, it was just sending every write and read request from the core to the memory and back.

2 Implementation



OLD VERSION **NEW VERSION**

As already mentioned the aim of the project is to implement a simple snooping process to enable coherency between different CPU's. The snooping process is a simple write invalidate protocol: if a core needs data it doesn't have in cache, it requests it to the arbiter, which sends the request to all the other cores. The cores will check if they have the data at the requested address and if they do, they will send it back to the arbiter that will then redirect the informations to the core which need them. The one that will send data back will also invalidate those data. If no CPU owns the data then central memory will have the most updated version of it, therefore the arbiter will ask memory to act like the other CPU's.

To implement the protocol first thing needed was to change the most external module, **orpsoc_top**, which was the module responsible of allocating all the other modules. A new external module, **orpsoc_multi_coherent_top**, has been created that allocates theorically as many cores as the user wants and the module has been created with the number of cores option as a parameter so that all the other settings that should be changed are done automatically in all the module allocations.

These changes affects amongst all **wb_intercon_multi_coherent** which is the new bus module. The old one, **wb_intercon** was created for a single core

and without the possibility to have coherency. The new one solves these problems and allocates a new kind of arbiter,**wb_snoop_arbiter**, which is wrapped around the old one, **wb_arbiter**. This new arbiter is the part of the bus that enables coherency and it does that by analyzing requests from the core before passing them to wb_arbiter. If the request is a write one, it simply let it pass through and the wb_arbiter will send it to memory and back.

Let's consider the new wb_snoop_arbiter as it is a completely new module created by scratch. It is composed by a finite state machine with four states:
IDLE is the initial state which sends directly data to the wb_arbiter after checking for a data read request and, in case, jumping to snoop_read.
SNOOP_READ is the state in which the snoop_arbiter waits the ack from all cores (apart the one requesting data) to know they checked for the snoop address. When this happens, if a core sent also a hit on the address it will have also have sent the data requested, so the arbiter simply redirects those informations to the core which requested them. If when all cores reply with the ack no one replies also with the hit it means that the wb_arbiter will ask to memory for those data and this will be done in mem_access.
MEM_ACCESS as previously said is the state in which the wb_snoop_arbiter lets the wb_arbiter handle the request to send it to memory as it was done in the old version of mor1kx. All this state does is reactivate the old arbiter so that it can take charge of the reuquests to memory.
SNOOP_WRITE at the moment is not used as the write is handled as it was before, therefore it doesn't need a new implementation with the wb_snoop_arbiter.

Now we'll get into what happens when the wb_snoop_arbiter sends snoop requests to the cores. The request is directly sent to the LSU in each core, without being modified or analyzed before by the CPU.
The LSU has been changed, as it was a finite state machine to which a new state has been added. **SNOOP_MANAGEMENT** is the new state which has the aim to analyze the answers sent from data cache and return them to the snoop_arbiter. Before going into more details let's consider where, if it is in data cache, the data requested could be. Obiouvsly if the core is not using them they could simply be in the data ram, but if the core was actually modifying them they could be found in the store buffer or they could already have been written into the bus. In both cases we can't just take these data from data ram, we need to get them in one case from the wire going in store buffer, **lsu_sdat** or from the one that will write them into the bus, **dbus_dat**.
At this point we can analyze what are the reactions of the LSU given the different answers by the data cache. if there is no hit in cache, the LSU immediately answers to the snoop_arbiter with the ack, while if the data have been found they are taken by the LSU (if we have conflicts, as said before, they're not taken from data cache) and the answer to the snoop_arbiter will be composed by the ack, the hit and the data.

The last module we need to consider is the **data_cache**. The most considerable change here is the addition of two new states in the finite state machine:
SNOOPHIT is the state in which if there is a snoop_hit (which will be automatically checked if the wire which could have the snoop address is not empty) the data cache waits for a clock cycle to ensure data have been sent to LSU before going back to idle.
SNOOPHIT_COUNTER is reached after the first clock on SNOOPHIT, it alerts the LSU that the data are ready and jumps back to IDLE.

3 Technical Informations

Here are the main lines of code that has been changed to implement the snooping protocol.

3.1 DATA CACHE

```
54      //SNOOP INTERFACE
55      // Snoop address
56      input  [31:0]              snoop_adr_i,
57      // Snoop event in this cycle
58      input                      snoop_valid_i,
59      // Whether the snoop hit. If so, there will be no tag memory write
60      // this cycle. The LSU may need to stall the pipeline.
61      output                     snoop_hit_o,
62      output [31:0]              snoop_dat_o,
63      output                     snoop_valid_dat_o,
64      //tells to dcache that outside has received the snooped datum. This should signal a
            state migration
65      //from SNOOPHIT to IDLE. master must deassert this signal.
66      input                      snoop_ack_i,
```

```
231     wire                    snoop_hit;
232     // Wheter the snooped data is retrieved.
233     reg  [OPTION_OPERAND_WIDTH-1:0] snoop_dat;
234     reg                     snoop_valid_dat;
235     // Snooped counter signal
236     reg         snoop_counter;
237
238     assign snoop_hit_o = (OPTION_DCACHE_SNOOP != "NONE") ? snoop_hit : 0;
239     assign snoop_dat_o = (OPTION_DCACHE_SNOOP != "NONE") ? snoop_dat : {
            OPTION_OPERAND_WIDTH{1'b0}};
240     assign snoop_valid_dat_o = (OPTION_DCACHE_SNOOP != "NONE") ?
            snoop_valid_dat : 0;
```

Snoop_dat_o, snoop_valid_dat_o and snoop_ack_i have been added to allow the cache to send informations about if the address requested by the snoop protocol have been found or not.

```
78    // States
79    localparam IDLE   = 7'b0000001;
80    localparam READ   = 7'b0000010;
81    localparam WRITE  = 7'b0000100;
82    localparam REFILL  = 7'b0001000;
83    localparam INVALIDATE = 7'b0010000;
84    localparam SNOOPHIT = 7'b0100000;
85    localparam SNOOPHIT_COUNTER = 7'b1000000;
```

```
340   assign snoop_hit_state = (state == SNOOPHIT);
341   assign snoop_hit_counter_state = (state == SNOOPHIT_COUNTER);
```

As said in the previous section, SNOOPHIT and SNOOPHIT_COUNTER are two new states, the first waits one clock cycle to enable, in the second one, to send back the informations about the address given by the snoop protocol.

```
174   wire [WAY_WIDTH-3:0] way_raddr_muxed[OPTION_DCACHE_WAYS-1:0];
```

```
260       assign way_raddr[i] = cpu_adr_i[WAY_WIDTH-1:2];
261       // Update the muxed wires
262       assign way_raddr_muxed[i] = (snoop_hit & (snoop_hit_state | snoop_hit_counter_state)
              ) ? snoop_adr_i[WAY_WIDTH-1:2] : way_raddr[i];
263       assign way_waddr[i] = write ? cpu_adr_match_i[WAY_WIDTH-1:2] :
264                   wradr_i[WAY_WIDTH-1:2];
265       assign way_din[i] = way_wr_dat;
```

Way_raddr_muxed has been added to enable the snooping protocol to switch the address to send to the ram to check for the data. When the data cache ends up in a snooping state the muxed wire receives the snooping address, while normally it simply receives a normal address to check for the cpu.

```
312     if (snoop_hit && (snoop_hit_state | snoop_hit_counter_state)) begin
313       for (w0 = 0; w0 < OPTION_DCACHE_WAYS; w0 = w0 + 1) begin
314         if (snoop_way_hit[w0]) begin
315         // In this situation the data_ram has on its output wire the snooped data.
316           snoop_dat = way_dout[w0];
317         end
318       end
319     end
```

Same thing as above but coming out of the data ram, the data need to be switched from the usual ones to the ones needed for the snooping protocol.

The finite state machine is composed of two parts, the first is activated only each clock posedge and its aim is only to switch from a state to the other, while the second is activated by any of the signals in the sensitivity list and is the one which actually completes the actions needed in each state.
Now let's start form IDLE state: only the first lines have been added, to reset snoop_valid_dat and to jump to SNOOPHIT if we have a snoop_hit. In READ and WRITE state nothing has been added, while in REFILL there's a check on snoop_hit to stop refilling if we have a hit.
SNOOPHIT_COUNTER is the only state in which we actually have to manage the snooping protocol (remember that SNOOPHIT is a state used only to wait for a clock cycle for the data coming from data ram).

```
514     SNOOPHIT_COUNTER: begin
515       // After one clock cycle in which we are in the snoop states, the snooped data
              has been retrieved
516       snoop_valid_dat <= 1'b1;
517       // Then we only wait for the end of the snooping operations, through an input
              signal
518       if (snoop_ack_i) begin
519         state <= IDLE;
520         // Reset the registers
521         snoop_valid_dat <= 1'b0;
522         snoop_dat <= {OPTION_OPERAND_WIDTH{1'b0}};
523       end
524     end
```

```
705     SNOOPHIT_COUNTER: begin
706       if (snoop_valid_dat) begin
707         // To be sure the write enable wire is zeroed
708         way_we = {(OPTION_DCACHE_WAYS){1'b0}};
709       end
710       // No operation needed since all the retrieving operations are handled outside,
711       // directly with the use of wires and regs.
712     end
```

Snoop_valid_dat is set to one after the cycle of SNOOPHIT as the data are surely available, then in the moment snoop_ack_i is raised the registers are reset

and the finite state machine goes back to idle. Also there's the necessity to ensure that the write enable signal is zeroed as if it gets raised in other parts of the finite state machine the snooping process cannot finish.

```
555         if (snoop_hit) begin
556             // This is the write access
557             tag_we = 1'b1;
558             tag_windex = snoop_windex;
559             for (w2 = 0; w2 < OPTION_DCACHE_WAYS; w2 = w2 + 1) begin
560                 if (snoop_way_hit[w2]) begin
561                     tag_way_in[w2][TAGMEM_WAY_VALID] = 1'b0;
562                 end else begin
563                     tag_way_in[w2] = snoop_way_out[w2];
564                 end
565             end
566         end else begin
```

The "lazy invalidation" that was implemented before has been modified as it was a loss of resources and time. It has been changed so that the only bit that now is zeroed is the validity bit, and not all the data line.

3.2 LSU

```
111     //snoop interface
112     output              snoop_response_ack_o, // When the response is available
113     output              snoop_response_hit_o,  // Snoop request's response.
114     output reg[OPTION_OPERAND_WIDTH-1:0] snoop_response_dat_o, // The eventual
            data
115     input               snoop_req_i,
116     input [OPTION_OPERAND_WIDTH-1:0] snoop_adr_i,
```

```
207     reg                 atomic_reserve;
208     wire                swa_success;
209
210     // Snoop operations
211     wire                snoop_valid;
212     wire                dc_snoop_hit;
213     reg             dc_snoop_ack;
214     wire              dc_snoop_valid_dat;
215
216     wire [OPTION_OPERAND_WIDTH-1:0] dc_snoop_dat_wire;
217     reg  [OPTION_OPERAND_WIDTH-1:0] snoop_dat;
218     reg               snoop_address_conflict ;
219     reg       snoop_on_refill ;
220     reg       snoop_on_write;
221     reg       snoop_req_delayed;
222
223     reg               snoop_response_hit;
224     reg               snoop_response_ack;
```

```
229     assign snoop_response_ack_o = snoop_response_ack;
230     assign snoop_response_hit_o = snoop_response_hit;
```

These are all the inputs, output, wires and registers that were necessary to enable snooping on LSU. The first ones are connected with the wb_snoop_arbiter while the others are for the data cache or for the lsu itself.

```
366     IDLE  = 3'd0,
367     READ = 3'd1,
368     WRITE  = 3'd2,
369     TLB_RELOAD = 3'd3,
370     DC_REFILL = 3'd4,
371     SNOOP_MANAGEMENT = 3'd5;
```

```
441       if (store_buffer_empty & snoop_req_i) begin
442         state <= SNOOP_MANAGEMENT;
```

```
499     end else if (snoop_req_delayed) begin
500       // Go into the snoop_management state with a special signal asserted
501       // In this way we do not block the refill in case of non snoop hits
502       // The request has been delayed of one clock cycle to allow the data cache
503       // to fullfil the snoop checkings.
504       snoop_on_refill <= 1;
505       state <= SNOOP_MANAGEMENT;
506     end
```

```
544     if (snoop_req_i) begin
545       snoop_on_write <= 1;
546       state <= SNOOP_MANAGEMENT;
547     end
```

From the IDLE state the SNOOP_MANAGEMENT state is reached if a snoop request is received and if the store buffer is empty. It is not reached if the store buffer is not empty because in that case the next state will be the WRITE state and only there, as it is visible in line 545 the SNOOP_MANAGEMENT state will be reached. This because if there is a write it might happen that the data that is being written is also the one needed by the snooping protocol. The jump to SNOOP_MANAGEMENT is done from the WRITE state to ensure that every action is completed and to avoid problems like writing the data and not having it for the snooping protocol. In a moment the process behind "snoop_on_write" will be analyzed.
The SNOOP_MANAGEMENT can be reached also from the DC_REFILL state. The reason behind this is that the DC_REFILL state has the necessity to use the bus, but if a snoop request arrives it means that the bus is busy. Therefore the core needs to go on and let the bus finish his snoop request before going on with the DC_REFILL work, or it might incur in a deadlock.

```
557     if (!snoop_address_conflict & !dc_snoop_hit) begin
558       // Write down the answers
559       snoop_response_ack <= 1;
560       snoop_response_hit <= 0;
561       snoop_response_dat_o <= {OPTION_OPERAND_WIDTH{1'b0}};
562     end else if (snoop_address_conflict | (!snoop_address_conflict & dc_snoop_valid_dat))
          begin
563       // Write down that we have had a hit, only when the data is available
564       snoop_response_ack <= 1;
565       snoop_response_hit <= 1;
566       snoop_response_dat_o <= snoop_address_conflict ? snoop_dat : dc_snoop_dat_wire;
567     end
```

```
589     if (!snoop_req_i) begin
590         // Unlock the dcache
591         dc_snoop_ack <= 1;
592         if ( snoop_on_refill ) begin
593           // Return to DC_REFILL state
594           state <= DC_REFILL;
595           // Reset the signal
596            snoop_on_refill <= 0;
597         end else if (snoop_on_write) begin
598           // Return to WRITE state
599           state <= WRITE;
600           // Reset the signal
601           snoop_on_write <= 0;
602         end else begin
603           // Return to IDLE state
604           state <= IDLE;
605       end
```

This is the inside of the SNOOP_MANAGEMENT state. As we can see if we have no conflicts (as seen before the data might not be in the data cache because it is being written in memory) and we have not a snoop_hit on the data cache it means that the core doesn't own the requested data, so the snoop_ack is asserted but the snoop_hit is not. If we instead have a conflict or the snooped data from the data cache is valid, this means that the core owns the requested data, and the data that is sent back to the wb_snoop_arbiter is either the one of the data cache, if there's no conflict, or the one coming from the conflict (we'll analyze that in a second).
The second part of the code is easier to understand and is simply the exiting process from SNOOP_MANAGEMENT: the snoop_req_i wire is the one that sent the LSU to SNOOP_MANAGEMENT, so when it is deasserted the snooping process is finished. The data cache is unlocked and the previous state (IDLE,WRITE OR DC_REFILL) is set again.

```
889 always @(snoop_req_i) begin : check_snoop_conflict_
890  // Posedge
891  if (snoop_req_i) begin
892     //set default snoop_dat
893     snoop_dat <= {OPTION_OPERAND_WIDTH{1'b0}};
894     //
895     // Here a snoop request has still been raised, thus we have to check for
896     // eventual conflicts :
897     //     - first, the address in which we are writing (thus the data coming
898     //  out from the store buffer) is the same of the snoop request.
899     //     - second, the address of the data we are saving into the store
900     //  buffer is the same of the snoop request.
901     //
902     //  Note that if we do not have a store buffer, only the first check makes
903     //  sense, since the data to be stored are directly written on the bus.
904     //
905     if(state == WRITE && snoop_adr_i == dbus_adr) begin
906       // We are writing the requested data on the bus, thus we have already
907       // the snooped data for handling the request in the FSM.
908       snoop_address_conflict <= 1;
909       snoop_dat <= dbus_dat;
910     end else
911     if (FEATURE_STORE_BUFFER != "NONE" && snoop_adr_i == store_buffer_wadr)
            begin
912       // We are writing the requested data into the store buffer, thus we
913       // have already the snooped data.
914       // WARNING: This kind of checks are formally correct ONLY under the
915       // current implementation of a write-trought cache, i.e., the store
916       // buffer will have at most one item.
917       snoop_address_conflict <= 1;
918       // The data we are storing into the store buffer is into the wire lsu_sdat
919       snoop_dat <= lsu_sdat;
920     end else begin
921       snoop_address_conflict <= 0;
922     end
923     // Finally the one cycle delayed request's signal is set to one
924     snoop_req_delayed <= 1;
925  end else begin
926   // Negedge logic
927   snoop_address_conflict <= 0;
928   snoop_dat <= {OPTION_OPERAND_WIDTH{1'b0}};
929   snoop_req_delayed <= 0;
930  end
931 end
```

As we said multiple times there is the chance to have a conflict where the data to send to the arbiter is not in the data cache because it is being written down in memory (remember that this is architecture is write_through). To handle this if a snoop_req_i is asserted (there's a request from the arbiter for a certain address) then if the state in which the LSU is is the WRITE one then the first thing to do before jumping to SNOOP_MANAGEMENT is to check that the data is not the one being written right now, and if that's the case the snoop_address_conflict is asserted and snoop_dat is connected directly to dbus_dat and we won't need the data cache to find something in its ram.

There's another case to take into consideration that is if the data is already in the store buffer but not yet written in the bus. So if there is a store buffer enabled and if the address requested by the snoop is the one im the store buffer again the snoop_address_conflict is raised but snoop_dat is connected with lsu_sdat to get it.
If neither the write nor the store buffer are having conflicts than the snoop_address_conflict is not asserted. In both the cases snoop_request_delayed is raised to leave a cycle to the data cache to check if it has the requested data.
When snoop_req_i is deasserted (Negedge) the necessary resets are handled.

3.3 WB_SNOOP_ARBITER

```
49    // Wishbone Master Interface
50    input [num_masters*aw-1:0] wbm_adr_i,
51    input [num_masters*dw-1:0] wbm_dat_i,
52    input [num_masters*4-1:0]  wbm_sel_i,
53    input [num_masters-1:0]    wbm_we_i,
54    input [num_masters-1:0]    wbm_cyc_i,
55    input [num_masters-1:0]    wbm_stb_i,
56    input [num_masters*3-1:0]  wbm_cti_i,
57    input [num_masters*2-1:0]  wbm_bte_i,
58    output [num_masters*dw-1:0] wbm_dat_o,
59    output [num_masters-1:0]   wbm_ack_o,
60    output [num_masters-1:0]   wbm_err_o,
61    output [num_masters-1:0]   wbm_rty_o,
62
63    // Wishbone Slave interface
64    output [aw-1:0]           wbs_adr_o,
65    output [dw-1:0]           wbs_dat_o,
66    output [3:0]              wbs_sel_o,
67    output                    wbs_we_o,
68    output                    wbs_cyc_o,
69    output                    wbs_stb_o,
70    output [2:0]              wbs_cti_o,
71    output [1:0]              wbs_bte_o,
72    input [dw-1:0]            wbs_dat_i,
73    input                     wbs_ack_i,
74    input                     wbs_err_i,
75    input                     wbs_rty_i,
76
77
78    //snoop interface
79    output [num_dbus*aw-1:0] snoop_adr_o,
80    output[num_dbus-1:0] snoop_type_o,
81    input [num_dbus-1:0] snoop_ack_i,
82    input [num_dbus-1:0] snoop_hit_i,
83    input [num_dbus*dw-1:0] snoop_dat_i
```

As the old arbiter has been wrapped inside the new snoop one all the inputs and outputs are mantained and connected to the old directly to let it work with no differences from before. The snoop interface has been added to connect the new arbiter with the cores. In particular snoop_adr_o is the wire that sends to the cores the address to check if they own it, ack, hit and dat inputs are going to be filled with the answers to the address by the cores.
We already explained briefly the states of the new arbiter, so let's get in depth with those:
IDLE:

```
195             IDLE:
196             begin
197                 if(active==1 && wbm_we_i[master_sel]==0 && wbm_cyc_i[master_sel]==1
                        && master_sel < num_dbus && wbm_ack_o[master_sel]==0)
198                 begin
199                     //$display("switch to SNOOP_READ due to active:%b, wbm_we_i:%b,
```

```
                    master_sel:%d, wbm_cyc_i: %b", active, wbm_we_i, master_sel,
                    wbm_cyc_i);
200                 next_state <= SNOOP_READ;
201               end
202               else
203               begin
204                 next_state <= IDLE;
205               end
206             end
```

```
251         IDLE:
252         begin
253           //don't interfere
254           snoop_adr_reg <= 0;
255         end
```

It's only objective is to check for possible snoop reads. This is done at line 197 where if wbm_cyc_i[master_sel] is raised (wishbone specification, wbm_cyc_i is raised by each core when it needs to use the bus. Master_sel represents the current bus user, so until it's wbm_cyc_i is raised, it hasn't finished to use the bus) and if wbm_we_i is not (wbm_we_i is raised by each core if it needs to make a write) and master_sel is lower than the number of data buses (remember that this is only a data snoop, and the masters are ordered with data buses first and address buses after) then it means that there is going to be a read and we need to check for snooping possibilities.
If that is not the case IDLE doesn't interfere with the messages sent to the arbiter underneath it and simply let informations pass.

```
207           SNOOP_READ:
208           begin
209             if(end_of_transaction)
210             begin
211               next_state <= IDLE;
212             end
213             else
214             begin
215               if( poll_response_flag == POLL_RESPONSE_NEGATIVE)
216               begin
217                 next_state <= MEM_ACCESS;
218               end
219               else
220               begin
221                 next_state <= SNOOP_READ;
222               end
223             end
224           end
```

```
256         SNOOP_READ:
257         begin
258           snoop_adr_reg <= wbm_adr_i[master_sel*aw+:aw];
259         end
```

SNOOP_READ is the state in which the arbiter stays while waiting for all the cores to check for the snoop. The possible answers from the cores and how they're handled is going to be explained in a second. If no core has the requested data (POLL_RESPONSE_NEGATIVE) then the arbiter switches to state MEM_ACCESS, while if not it can either go back to IDLE if the transaction has been done (end_of_transaction) or stay in SNOOP_READ until that point. The only action SNOOP_READ actually does is to fill the snoop_adr_reg wire with the address that has to be checked by the cores for the snoop.

```
149  assign wbs_adr_o = (state == SNOOP_READ) ? {aw{1'b0}} : wbm_adr_i[master_sel*aw
         +:aw];
150  assign wbs_dat_o = (state == SNOOP_READ) ? {dw{1'b0}} : wbm_dat_i[master_sel*dw
         +:dw];
151  assign wbs_sel_o = (state == SNOOP_READ) ? {dw{1'b0}} : wbm_sel_i[master_sel
         *4+:4];
152  assign wbs_we_o  = wbm_we_i [master_sel];
153  assign wbs_cyc_o = (state == SNOOP_READ) ? 1'b0 : wbm_cyc_i[master_sel] & active;
154  assign wbs_stb_o = wbm_stb_i[master_sel];
155  assign wbs_cti_o = wbm_cti_i[master_sel*3+:3];
156  assign wbs_bte_o = wbm_bte_i[master_sel*2+:2];
```

```
162  assign wbm_ack_o = (state == SNOOP_READ) ?
163                     ( poll_response_flag  == POLL_RESPONSE_POSITIVE) ?
164                       1'b1<<master_sel :
165                       {num_dbus{1'b0}} :
166                     ((wbs_ack_i & active) << master_sel);
```

Still in SNOOP_READ we have to conside that the underneath arbiter cannot send data to memory while we're waiting for the cores to answer the snoop requests. To avoid this the outputs that should send informations to the arbiter are zeroed when in SNOOP_READ, so that it cannot send requests to memory.

```
225          MEM_ACCESS:
226          begin
227           if(end_of_transaction==1)
228           begin
229              next_state <= IDLE;
230           end
231           else
232           begin
233              next_state <= MEM_ACCESS;
234           end
235          end
236          default:
237          begin
238              next_state <= IDLE;
239          end
```

```
260        MEM_ACCESS:
261        begin
262           //restore state before snoop read and access to memory
263           snoop_adr_reg <= 0;
```

```
264         end
```

MEM_ACCESS is called by the SNOOP_READ state in case the cores don't own the data. This means that these data are in memory and therefore the snoop_adr_reg is emptied, and the state will switch to IDLE only when data will be retrieved from memory.

```
273   always@(*)
274   begin
275      snooped_dat=0;
276      if(state==SNOOP_READ)
277      begin: snoop_read_active
278         integer k;
279         poll_response_flag = POLL_RESPONSE_UNDEFINED;
280         for(k=0; k<num_dbus; k = k+1)
281         begin
282            if(snoop_ack_i[k]==1)
283            begin
284               if(snoop_hit_i[k]==1)
285               begin
286                  poll_response_flag = POLL_RESPONSE_POSITIVE;
287                  snooped_dat = snoop_dat_i[dw*k+:dw];
288               end
289            end
290         end
291         if( ((snoop_ack_i | snoop_mask) == {num_dbus{1'b1}}) && snoop_hit_i=={
               num_dbus {1'b0} } && poll_response_flag ==
               POLL_RESPONSE_UNDEFINED)
292         begin
293            poll_response_flag = POLL_RESPONSE_NEGATIVE;
294         end
295      end
296      else
297      begin
298         snooped_dat = 0;
299         poll_response_flag = POLL_RESPONSE_UNDEFINED;
300      end
```

Last but not least this is how the WB_SNOOP_ARBITER handles the requests to the cores for data:
If the arbiter ends in the SNOOP_READ state then the variable poll_response_flag, which will be used to decide wether to go to MEM_ACCESS to retrieve the data, is set to undefined. This happens because there's the need to wait for all cores to answer before filling it with an actual response. then, each snoop_ack_i wire is checked to analyze which cores answered. If a core sends the ack also the hit is checked to see if the data have been found, and if that's the case poll_response_flag is changed to POLL_RESPONSE_POSITIVE and the data arriving from the core which answered positively are sent to snooped_dat, from which they will be sent to the core requesting them. How to realize if all cores answered negatively is handle at line 291, where the array snoop_ack_i is or'ed bitwise with a mask which is necessary as the address has not been sent to the core requesting it (as it wouldn't make sense) and therefore it will not answer.

If the or operation is equal to a sequence of ones as long as num_dbus dimension (or in simple words if every core apart the one which is masked has answered) and snoop_hit_i is equal to a sequence of zeros as long as num_dbus (no one answered positively) and the poll_response_flag has not already been changed to positive or negative, then it is set to negative and from SNOOP_READ the arbiter will switch to MEM_ACCESS.