Final Report: Dual core MIPS processor

ECE 437: Computer Design and Prototyping

TA: Adam Hendrickson

2012-12-07

by

Sambit K. Mishra (mg271)

And

Siddhesh Rajan Dhupe (mg216)

Executive overview

In this report we present the design, implementation and results of a cache coherent dual core MIPS processor. We implemented a one level split cache. The processors were implemented in VHDL and tested on ModelSim for source simulation and on Altera II cyclone FPGAs for hardware verification. The processors' designs were based on the designs provided in Chapters 4, 5 and 7 of the revised 4th edition of Computer Organization and Design by Hennessey and Patterson.

Both processors contain 32 32-bit registers. The pipelined processor has five stages viz. fetch, decode, execute, memory and writeback. The single cycle processor had the luxury of having separate memories for instruction and data. On the other hand the pipelined processor had to use a single memory and this warranted the use of a state machine to arbitrate between data and instruction fetch. Initially we developed the single cycle processor individually and then teamed up for the development of the pipelined processor. The multi core has a shared memory model and a coherence controller to maintain coherence in caches. It also supports the LL/SC instruction to provide RMW atomicity. Both the instruction and data caches have 16 frames.

During the design we faced many challenges, however we were able to implement a fully functional dual core processor with cache coherence. While implementing the multi core we had an especially tough time implementing the cache coherence. We developed the processor in a modular manner developing and testing the various modules e.g. caches, pipeline and coherence controller. Extreme care was taken in naming signals in a meaningful manner so as to ease debugging process..

The entire process demonstrated the advantages of multi core and caches over single core pipelined processors. It also brought into light the various issues involved with cache coherence and how they can be dealt with. In the following sections we describe the details of the implementation of both the processors and demonstrate a debugging example. We also present the results obtained by testing using MIPS assembly programs.

1. Processor Design

1.1 Pipelined processor and cache design

We implemented a 5 stage (instruction fetch, decode, execute, memory and writeback) pipelined processor having a single level split caches for instruction and data. We used Siddhesh's single cycle processor's modules to implement the pipelined processor. Though pipelining increased the latency of a single instruction, it improved overall throughput at increased clock frequency. Thus we have a faster processor. However pipelining gave rise to hazards. They were solved by forwarding and hazard detection logic.

We implemented a 16 set direct mapped icache and a 16 set two way associative dcache. Each block size was 2 words. We have a 32 bit address for accessing the cache.

No of bits for	ICache	DCache
Tag	25	26
Index	4	3
Block offset	1	1
Byte offset	2	2

The cache has two parts, the data and tag arrays and the controller. The controller services the cache on misses by loading data from the memory. It is implemented with a state machine. The state machine stays in the compare state during cache hits. When there is a miss it starts loading a new block into the cache. When the loading of the block into the cache is complete, the controller interface recognizes a hit and signals it to the processor. The cache controller state machines are shown in appendix.The cache used an LRU policy.

1.2 Multi core processor and cache coherence

We implemented a dual core processor with a snoop based MSI cache coherence protocol. We replicated the pipeline and caches and connected it to the bus controller. Each processor starts at a fixed PC. We modified the arbiter implemented for the pipeline to arbitrate between the two processors's instruction caches and bus controller. The appendix shows the connections between all modules. We implemented the LL/SC instructions which

are used for RMW atomicity. This is very important for shared memory model to work correctly. The earlier cache design had to be modified to include a 'link' register. Also cache controller's actions had to be modified.

Cache coherence refers to the consistency of data stored in local caches of a shared resource. In a shared memory multiprocessor system with a separate cache memory for each processor, it is possible to have many copies of any one instruction operand: one copy in the main memory and one in each cache memory. When one copy of an operand is changed, the other copies of the operand must be changed also. Cache coherence is the discipline that ensures that changes in the values of shared operands are propagated throughout the system in a timely fashion.

Our cache coherence mechanism was snoop based. Read or write misses are broadcast on a bus which snoops into all other caches. Cache controller updates state of the blocks in response to processor and snoop events and generates bus transactions. The different states of our bus and cache controllers and cache blocks are shown in appendix.

Since snooping occurs in parallel with processor read and write requests we had to have duplicate copies of tags otherwise this would have created a structural hazard. We could have had a dual ported cache tag array. However it is costlier in terms of area.

We reused the arbiters we had used for our previous labs. Arbiter 0 services requests from core 0's icache and bus controllers and arbiter 1 services requests from core 1's icache. The processor arbiter arbitrates between requests from the two arbiters. Arbiter 0 was given higher priority over arbiter 1 and arbiter 0 gave higher priority to requests from bus controllers. Thus our effective priority order was

Priority of Bus controller > Priority of icache 0 > Priority of icache1

This was done according to lab instructions and also because data fetches are given priority over instruction fetches.

In order to implement LL/SC instruction we modified the datapath such that our controller had two new signals viz. linked and st_condnal. The "linked" signal was asserted whenever there was ll instruction and the st_condnal was asserted for an sc instruction. The cache took both the linked and st_condnal signals. For a ll instruction the cache stores the address in a link register and sets its valid bit. The link register is invalidated whenever there is a write to that location. Thus whenever a processor tries to store (sc) to a linked address it first checks whether its link register has a valid value. If yes then it stores to that address and returns that the store conditional was successful. This is done a by a sc_success signal from the cache. Whenever sc_success is one i.e. a conditional store was successful, the write data is written back to the register else a zero is written back to the register. We then check the register to see if there was a successful store or not. We also had to modify the forwarding and hazard detection logic. The details of the ll/sc implementation are shown in Appendix.

2. Processor Debugging

Debug.asm	Expected memout.hex	Incorrect memout.hex
org 0x0000	:04000000341D3FFC70	:04000000341D3FFC70
ori $sp, $zero, 0x3ffc	:040001000C00000BE4	:040001000C00000BE4
jal mp0	:04000200FFFFFFFFFE	:04000200FFFFFFFFFE
halt	:04000300C0880000B1	:04000300C0880000B1
	:040004001408FFFEDF	:040004001408FFFEDF
lock:	:0400050021080001CD	:0400050021080001CD
ll $t0,0($a0)	:04000600E08800008E	:04000600E08800008E
bne $t0,$0,lock	:040007001008FFFBE3	:040007001008FFFBE3
addi $t0,$t0,1	:0400080003E0000809	:0400080003E0000809
sc $t0,0($a0)	:04000900AC800000C7	:04000900AC800000C7
beq $t0,$0,lock	:04000A0003E0000807	:04000A0003E0000807
jr $ra	:04000B0027BDFFFC12	:04000B0027BDFFFC12
	:04000C00AFBF000082	:04000C00AFBF000082
unlock:	:04000D003404006057	:04000D003404006057
sw $0,0($a0)	:04000E000C000003DF	:04000E000C000003DF
jr $ra	:04000F00340A02406D	:04000F00340A02406D
	:040010008D48000017	:040010008D48000017
mp0 :	:04001100250920009D	:04001100250920009D
push $ra	:04001200AD490000F4	:04001200AD490000F4
ori $a0,$zero,l1	:04001300340400 6051	:04001300340400 6051
jal lock	:040014000C000009D3	:040014000C000009D3
ori $t2,$0,res	:040015008FBF000099	:040015008FBF000099
lw $t0,0($t2)	:0400160027BD0004FE	:0400160027BD0004FE
addiu $t1, $t0, 0x2000	:0400170003E00008FA	:0400170003E00008FA
sw $t1,0($t2)	:04008000341D7FFCB0	:04008000341D7FFCB0
ori $a0,$0,l1	:040081000C000083EC	:040081000C000083EC
jal unlock	:04008200FFFFFFFF7E	:04008200FFFFFFFF7E
pop $ra	:0400830027BDFFFC9A	:0400830027BDFFFC9A
jr $ra	:04008400AFBF00000A	:04008400AFBF00000A
	:0400850034040060DF	:0400850034040060DF
l1:	:040086000C00000367	:040086000C00000367
cfw 0x0	:04008700340A0240F5	:04008700340A0240F5
org 0x200	:040088008D4800009F	:040088008D4800009F
ori $sp,$0,0x7ffc	:040089002509FFBE88	:040089002509FFBE88
jal mp1	:04008A00AD4900007C	:04008A00AD4900007C
halt	:04008B0034040060D9	:04008B0034040060D9
	:04008C000C0000095B	:04008C000C0000095B
mp1 :	:04008D008FBF000021	:04008D008FBF000021
push $ra	:04008E0027BD000486	:04008E0027BD000486
ori $a0,$zero,l1	:04008F0003E0000882	:04008F0003E0000882
jal lock	**:040090000000DEADE1**	**:040090000000DEEFE1**
ori $t2,$0,res	:040FFE0000000008E7	:040FFE0000000008E7
lw $t0,0($t2)	:041FFE0000000208D5	:041FFE0000000208D5

addiu $t1, $t0, -66
sw $t1,0($t2)
ori $a0,$0,l1
jal unlock
pop $ra
jr $ra
 	
res : cfw 0xBEEF
	:00000001FF	:00000001FF

As is evident from the differences between the correct and incorrect memout.hex the cache coherence didn't work. Two possible errors that could cause this output are:

1. cache 0 did not get the modified data from cache 1 and instead loaded it from memory. Thus we ended up getting BEEF+2000 = DEEF instead of BEEF+2000-42=DEAD. It would probably be because the cache 1's write was not broadcast or it didn't respond to cache 0's snooping.

2. Both processors got the lock simultaneously and read BEEF from memory. This might be because the LL/SC implementation has errors.

In order to debug the first case we just need to check what happens when we execute sw $t1,0($t2) and lw $t0,0($t2) for either processor. The write should be broadcasted and the read miss should be serviced by snooping into the other cache. For the second case we need to check the LL/SC control signals to see if we are getting a failure when we try locking an already locked location and if its getting forwarded to registers and ALU for branch condition check. These will help debug our issue.

3. Results

The results for the two designs are provided below.

Our critical path is determined from the timing analyser section in compile.log, generated during synthesis. The table below describes our findings.

Processor	Worst case tco	From	To
Pipelined processor with caches	12.233 ns	cpu:cpu_comp|mycpu:theCPU|cacheDump:cacheDumper|pc[5]	LEDG[8]
Dual core processor	13.342 ns	KEY[3]	cpu:cpu_comp|mycpu:theCPU|core:core:dataCache|cacheReg:way1|block64~159

Fig 3.1 Critical path description for each processor design

The critical path of the dual core processor originates in the data cache's state, goes through the block-select logic, through the cache array, out the data bus, snoops into the other cache, and returns to the requesting processor.
The critical path of the pipelined processor originates in the EX/MEM register, goes into the cache block select logic, through the hit/miss logic and returns to the requesting processor.
We used a weighted average to calculate the average IPC. The following formulae were used for MIPS and estimated frequency
MIPS = clk rate / (CPI * 10^6).

Estimated frequency = 1/ worst case tco

The following tables show our results for each of the processors we implemented.

Estimated frequency	81.7 MHz
Average IPC	0.41
Latency	61.165 = (5 * worst case tco)
MIPS number	34
FPGA resources	Total combinational functions: 4,558 / 33,216 (14 %) Dedicated logic registers: 3,264 / 33,216 (10 %)

Fig 3.2 Results for our pipelined processor with split caches

Estimated frequency	74.95 MHz
Average IPC	0.52
Latency	66.71 ns = (5 * worst case tco)
MIPS number	39
FPGA resources	Total combinational functions: 10,937 / 33,216 (33 %) Dedicated logic registers: 6,866 / 33,216 (21 %)

Fig 3.3 Results for our dual core processor

In order to compare CPIs of our processors we used two different programs. The dual core version of merge sort carries out an insertion sort on two data sets and the merges them to form a sorted output. The single core version does the same thing, except it runs the insertion sorts one after the other. The mult does a shift and add of partial products.

Table below outlines the CPI we got for each of the test programs on our pipelined processor.

Program	Instruction coun	Total cycles	CPI	IPC
MergeSort	80753	203909	2.52	0.40
Mult	29	71	2.45	0.41

Figure 3.4 CPI for merge sort and mult for pipelined processor with caches

Program	Instruction coun	Total cycles	CPI	IPC
Dual.mergeSort	53237	101558	1.91	0.52
Dual.Mult	52	96	1.84	0.54

Fig. 3.5 CPI for merge sort and mult programs for dual core processor

4. Conclusion

We started out with a single cycle processor and then successfully pipelined it. Pipelining exploited the Instruction Level Parallelism (ILP) and reduced the clock speed with of course some loss to CPI, and brought with it an overall reduction in execution time. It however introduced the possibility of hazards which required additional hardware, in terms of a detection unit and forwarding logic, to be dealt with. We then added caches to it. It improved CPI. We went further and replicated the cores. We then implemented snoop based cache coherence. Although our processor may have some minor bugs it was successfully able to execute the test programs provided. We could probably have implemented a 2-bit branch predictor with a branch prediction table to reduce our branch misprediction penalty if we had more time. We could have also implemented a write buffer for the writeback cache we had.

We could only go on pipelining as long as the penalties due to hazards do not offset the gain from improvement in clock speed. However we can further exploit ILP by designing multiple issue instruction execution pipelines. Of course we would require additional hardware but as Prof. Vijaykumar says, “We have got billions of transistors”. However even using superscalar and out of order processors can take us only so far. Ever since we hit the power wall we have been using multi cores. Multi cores consume less power and increase throughput. However parallel programming is tough and hence it becomes difficult to use them.

Our VHDL skills improved and we learnt quite a few concepts in vhdl programming and debugging. In conclusion we would say that our processor could be used as a small microcontroller if IO and interrupts are added.

5. References

[1] David A Patterson, John L Hennessy. Computer Organization and Design, Revised Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design) 2011

[2] Professor T.N. Vijaykumar's lecture notes

[3] ECE437 lab materials

[4] wiki-Cache_coherence

APPENDIX

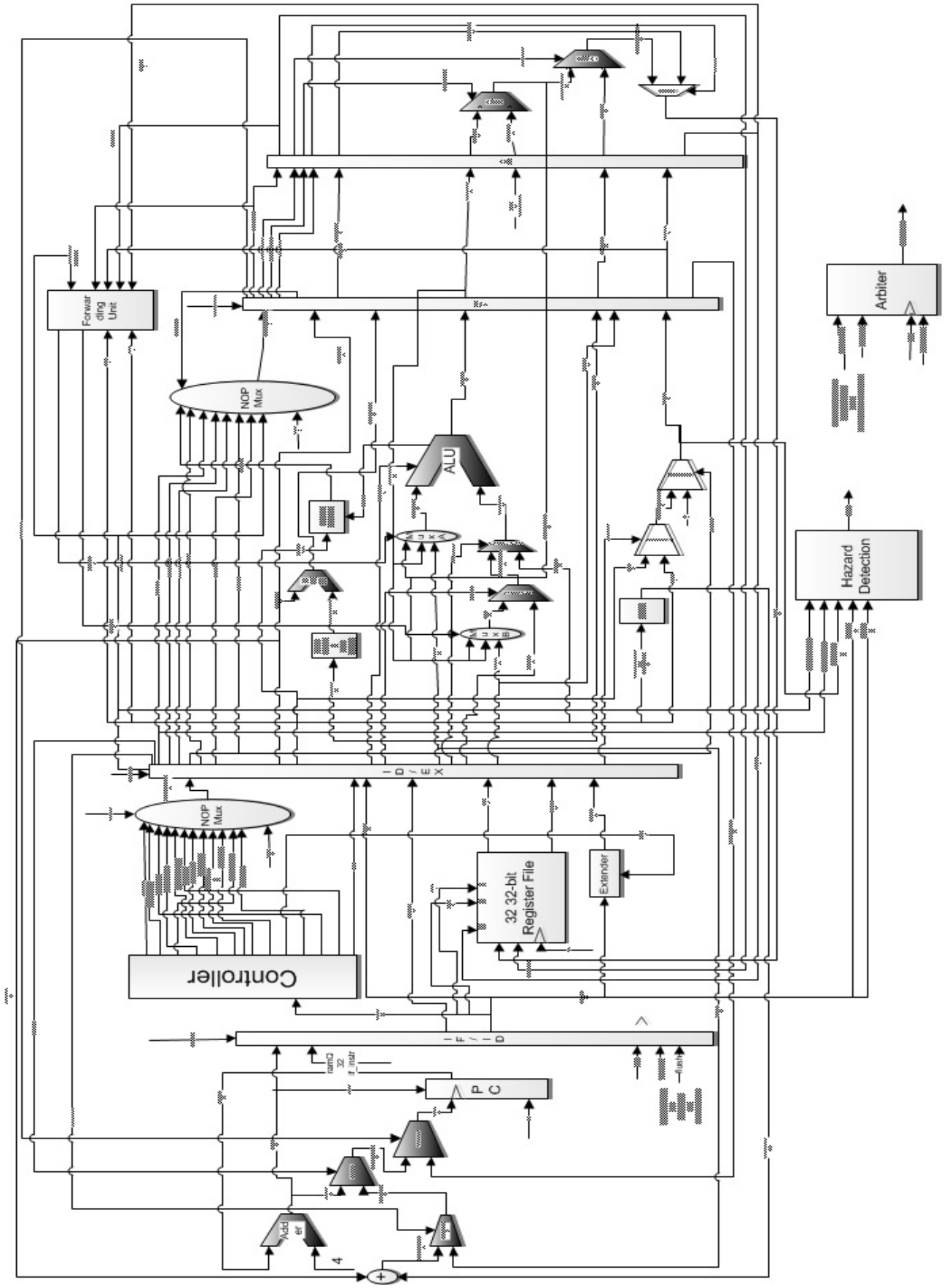

Figure A.1 : Pipelined processor diagram

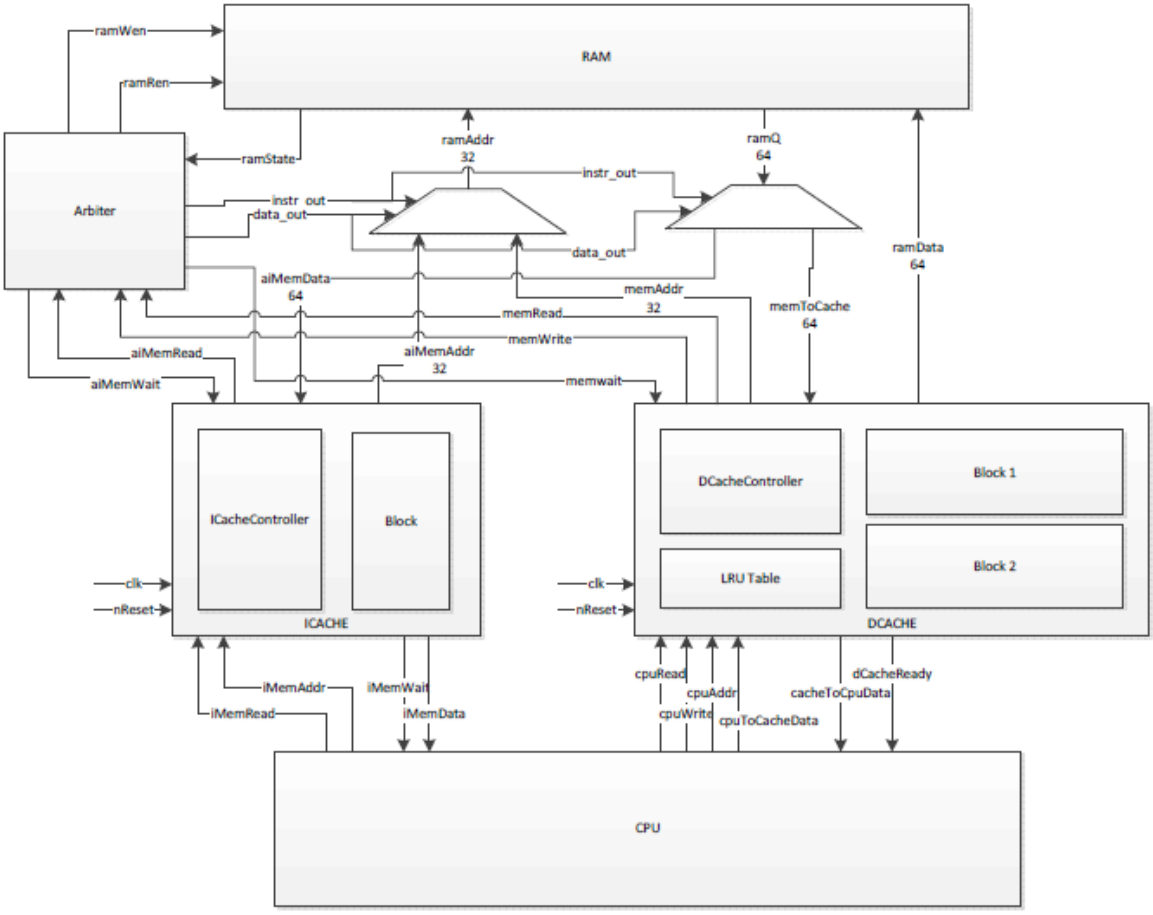

Figure A.2 : ICache and Dcache coupled with memory and processor

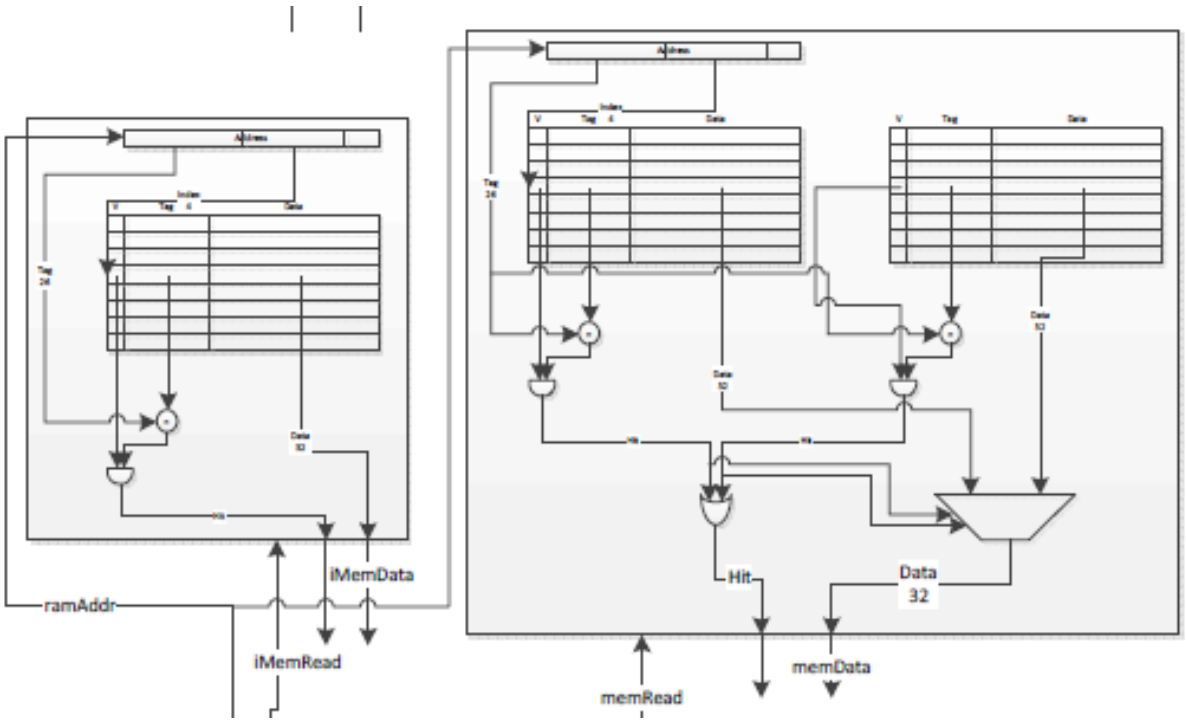

Figure A.3 : Internal logic of icache and dcache

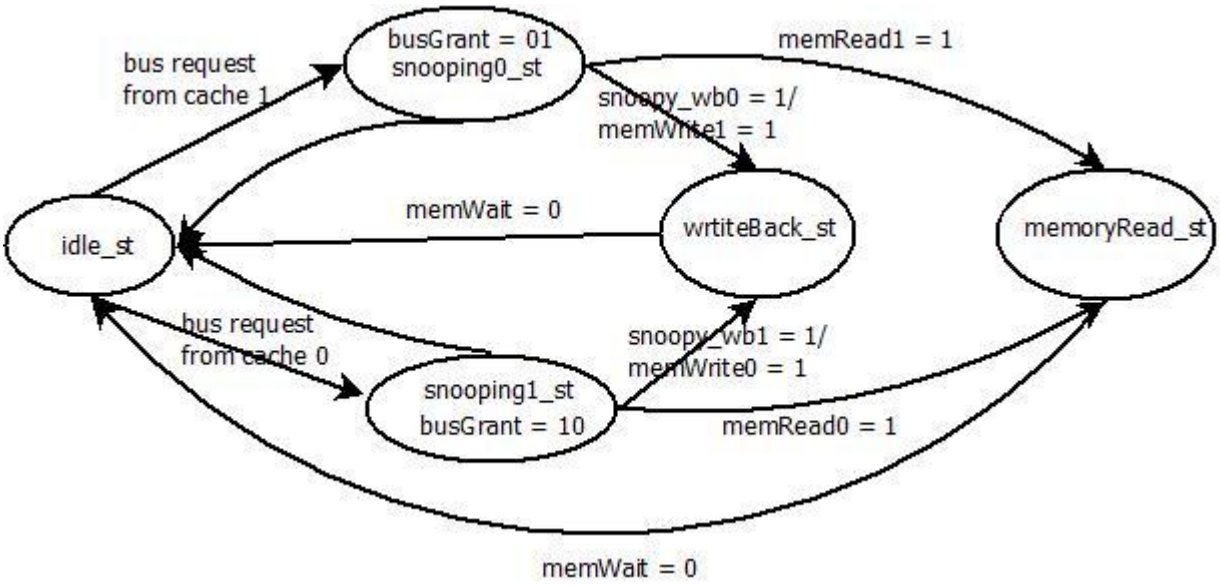

Figure A.4 : States of the bus controller

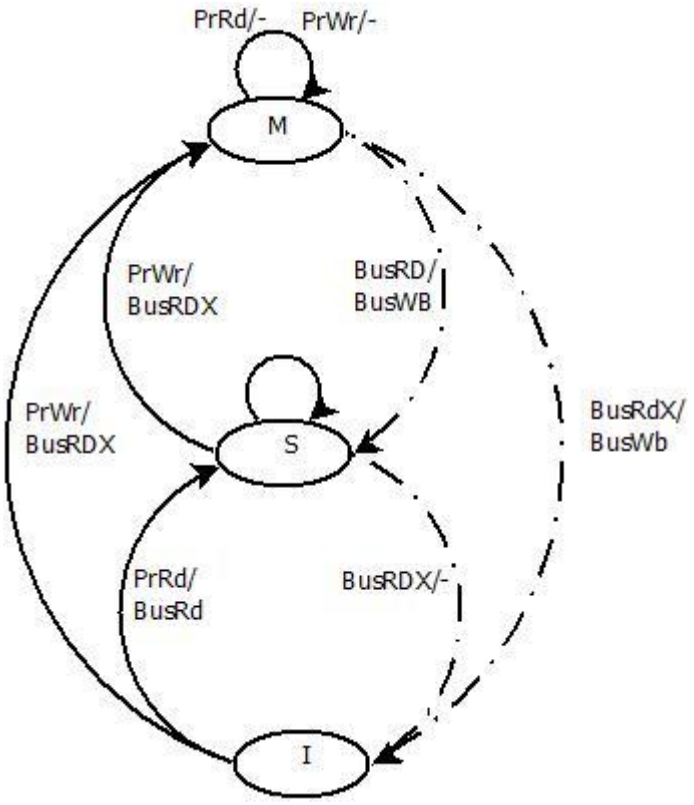

Figure A.5 : MSI states and transitions. On the right are transition due to processor requests and on the left are transitions due to bus snooping

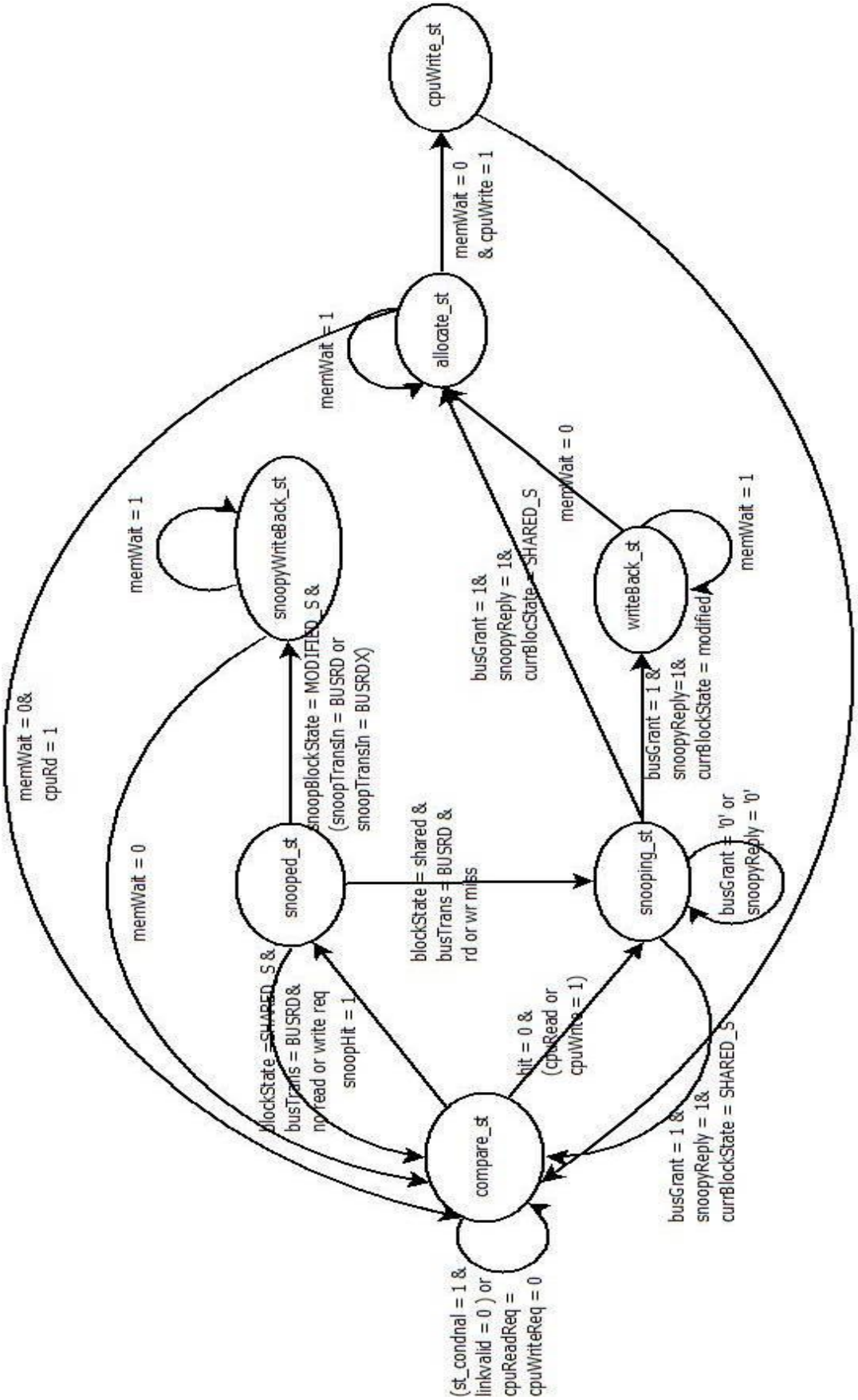

Figure A.5 : DCache controller states

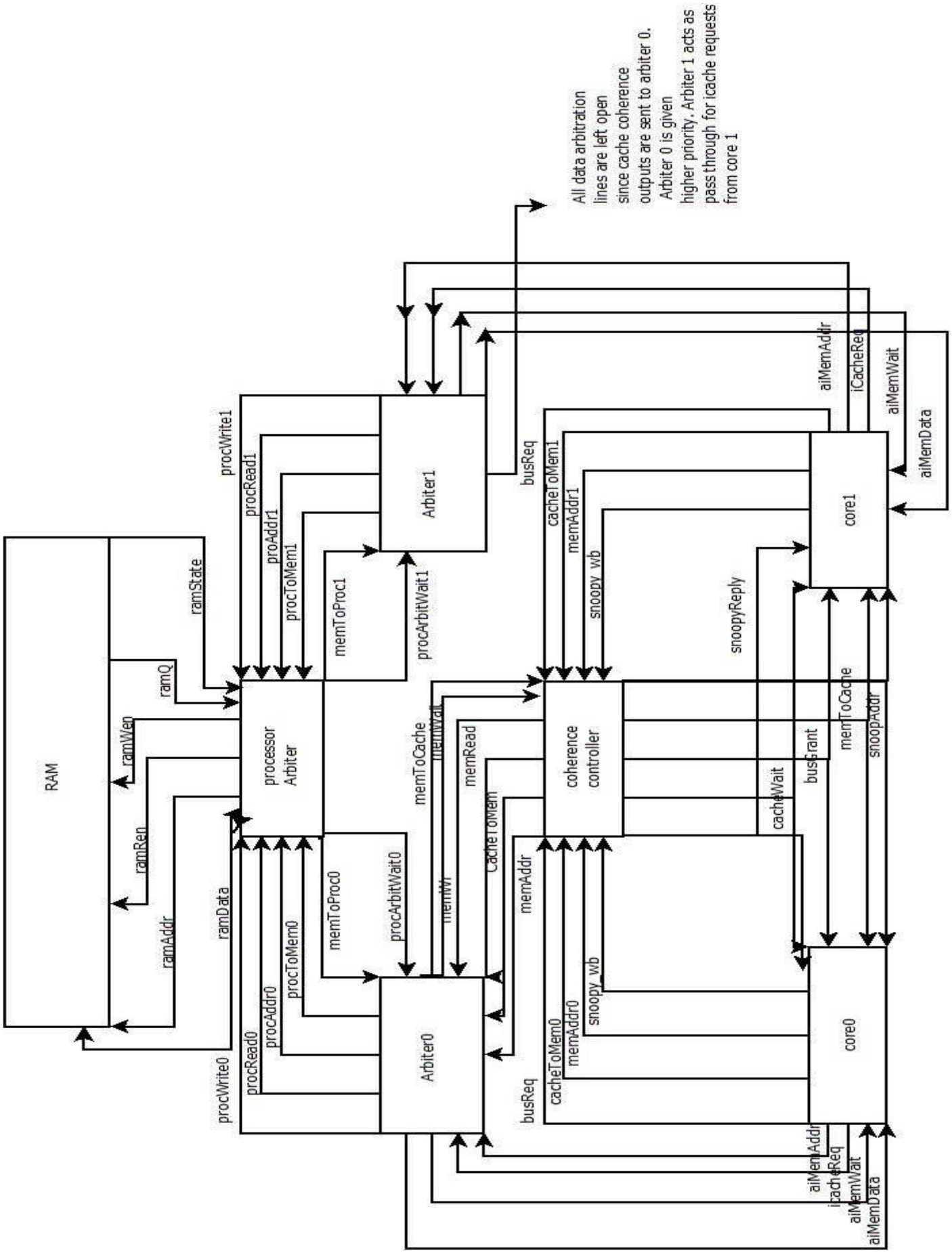

Figure A.6 : multi core with cache coherence controller

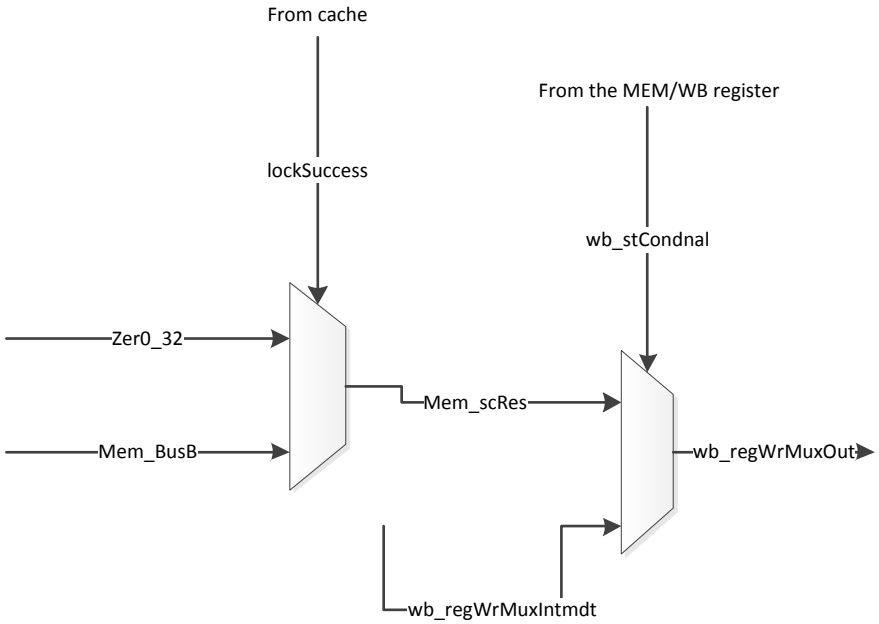

Figure A.7 : implementation for sc to write 0 or same value back to the register in case store conditional fails or passes respectively

Hazard detection Logic

```
if(IDEXMemRead = '1' and IDEXRegisterRd /= "00000") then
        if(IDEXRegisterRd = IFIDRegisterRs) then
            STALL <= '1';
        elsif(IDEXRegisterRd = IFIDRegisterRt) then
            STALL <= '1';
        else
            STALL <= '0';
        end if;
    elsif(EXMEMMemRead = '1' and EXMEMRegisterRd /= "00000") then
        if(EXMEMRegisterRd = IFIDRegisterRs) then
            STALL <= '1';
        elsif(EXMEMRegisterRd = IFIDRegisterRt) then
            STALL <= '1';
        else
            STALL <= '0';
        end if;
    elsif(IDEXStCondnal = '1' and IDEXRegisterRd /= "00000") then
        if(IDEXRegisterRd = IFIDRegisterRs) then
            STALL <= '1';
        elsif(IDEXRegisterRd = IFIDRegisterRt) then
            STALL <= '1';
        else
            STALL <= '0';
        end if;
    elsif(EXMEMStCondnal = '1' and EXMEMRegisterRd /= "00000")then
        if(EXMEMRegisterRd = IFIDRegisterRs) then
            STALL <= '1';
        elsif(EXMEMRegisterRd = IFIDRegisterRt) then
            STALL <= '1';
```

```
            else
                STALL <= '0';
            end if;
        else
            STALL <= '0';
        end if;
```

LL/SC implementation

```
    process(clk,linked,cpuAddr,cpuRead,nReset)
    begin
        if(nReset = '0')then
            linkedReg(31 downto 0) <= (others => '0');
        elsif(rising_edge(clk) and (cpuRead = '1') and (linked = '1'))then
            linkedReg(31 downto 0) <= cpuAddr;
        else
            --do nothing--
        end if;
    end process;
    process(nReset, snoopTransIn, snoopAddrIn,cpuRead, linked,linkInvalidate,cacheState,clk)
    begin
        if(nReset = '0')then
            linkedReg(32) <= '0';
        elsif((rising_edge(clk)) and ((snoopTransIn = BUSRDX and snoopAddrIn = linkedReg(31
downto 0) and cacheState = SNOOPED) or (linkInvalidate = '1')))then
            linkedReg(32) <= '0';
        elsif((rising_edge(clk)) and cpuRead = '1' and linked = '1')then
            linkedReg(32) <= '1';
        end if;
    end process;

-------------------------------------------------------------------------------------------------------------------------
if((readReq = '1' or writeReq = '1') and hit = '1') then

                    if(stCondnal = '1')then
                        if(linkValid = '0')then
                            scSuccess <= '0';
                        else
                            linkInvalidate <= '1';
                            scSuccess <= '1';
                            cacheDataWriteEn <= '1';
                            cacheTagWriteEn <= '0';
                        end if;
end if;
```