Final Report for Cache Hierarchy and Dual Core Processor Design

ECE 437: Introduction to Digital Computer Design and Prototyping

Jason Holmes

12/07/12

Scott Stack and Chris Cange

4.2 Executive Overview

A pipeline processor with both instruction and data caches as well as a dual core processor were designed for this exercise. In the end both designs were successfully implemented, but there were issues at some of the checkpoints. Specifically, there were a couple corner cases that caused our pipeline design with caches to fail when there was delay in the RAM. In addition, our coherence controller for the dual core design was not implemented correctly and would only work in source when it was submitted. However it has since been fixed and functions properly now.

To implement the pipeline design with caches, an instruction cache was provided and integrated into the existing pipeline processor design. In addition, a two way set associative data cache was designed and added to the existing pipeline processor. A new main memory module was also provided to simplify the cache design. This ram module had a 64 bit data bus instead of the previous 32 bit one. This was done because the cache blocks for both the instruction and data caches were two words or 64 bits wide.

After both caches were integrated into the pipeline, a dual core processor was implemented using the cached pipeline design for both the cores. A coherence controller was designed to ensure that the data in both cores caches stayed coherent. This controller made sure that only one copy of data at an address was written to at any one time. Additionally, hardware had to be added to implement the instructions ‘load link’ (LL) and ‘store conditional’ (SC). These instructions were built in to aid with thread synchronization between the two cores.

The rest of this document will first discuss the overall design of both of the processors, demonstrate our debugging skills, and present the final results of both of the processor designs. Overall, this exercise was successful in the end even though the final version of the dual core processor was not finished on time.

4.3 Processor and Cache Design

Pipelined Processor:

The pipelined processor first started out with the basic design of a single cycle processor but had to be expanded on with registers to store the data between each of the different stages of the pipe. Since our pipelined processor is split into five stages each of the registers was placed after the stage that correlates to one of the stages. All of the signals had to be sent into the pipeline registers before being able to get to the next set of stages. Since the first stage is the fetch stage everything that dealt with the PC and fetching the instruction from memory was located there. After this the next phase is the decode phase. Everything related to decoding the instruction in the control unit and getting the data out from the registers is located their. Third is the execution stage. This has everything related the ALU and its controller which further decodes the instruction to allow the correct ALU operation to happen. This stage selects which values will be sent to the ALU via the forwarding unit and the immediate instruction mux. The fourth stage is the MEM stage and it relates to both the loading and storing of values from or into mem. Finally the last stage in the pipeline deals with the writeback of the values to the register.

Since multiple instructions are in the pipe at a time one thing that has to be dealt with in pipelining and not a single cycle processor is forwarding. This unit is needed to forward values from a stage that is already completed back to the execution stage. The forwarding unit would have to check if the value from either the MEM or writeback stage used the one of the registers that was supposed to be written in either of those stages. If this value was indeed ahead of the pipe but not yet written back it would need to be selected from the forwarding selecting muxes. The forwarding unit specifically checks to see if the mem stage has the same register usage as one of the two going into the ALU and forwards the data value if so. The same goes with the writeback stage after the MEM stage takes precedence since it would have been previously updated again. Another block located in the pipelined processor is a arbiter. This block was needed because in a pipelined processor both the fetch and the MEM stages happen at the same time and therefore both can have the need to access memory at the same time. The arbiter looks to see if a memory access is needed in the MEM stage and then gives precedence to the MEM stage load or store. A final unit that had to be added was the hazard detection unit. It detects and resolves load dependencies branch mispredictions, and other control hazards. If a branch is discovered and the condition is true, it will flush the two instructions preceding it and set the PC to the appropriate value. If a load dependency is detected it will insert a nop bubble into the pipe to resolve the issue. If the instruction is a jump or jump through register or jump and link, it will clear the IF/DEC register to clear the previously fetched instruction and let the next PC logic take over to set the PC to the right value.

With the inclusion of caches inside the pipeline processor many design decisions were required to help facilitate the correct working of the processor. One thing that had to be done was the arbiter had to be updated to wait on an access of ram from the icache and the dcache. This arbiter would allow dcache misses to access ram first while delaying the pipe and then allow the icache to access it next. This was so data in the dcache could be completed before the pipe continued to execute instructions. The icache was given to us and the dcache was fully designed by us in the processor. For the dcache we had a top level block that would take in signals both from the arbiter and the cpu. From the CPU a read, write, address and data information was needed to see if it was a store word or load word and also to see if the address was already located inside the cache with the address. This address was taken in and changed to a tag and index and stored inside the cache accordingly. If the correct tag at the index was already located in the cache a hit would occur and no memory access would be needed. If it was a store word it would make the bit dirty to signify that it was written too. If there was not a hit the dcache controller would begin to change to different states based on what type of transaction was needed.

For cache misses there are many different types of state changes that could happen. The first state change is a read miss with the data being clean. On this the state would change to a state that waits for ram to be accessed and retrieve the data at the address requested and then bring it back into the dcache at the correct location. After this the cache would return to the idle state. The next read miss is with the data being dirty. When the data is dirty at the location that wants to be stored at the a write back to ram needs to occur before a read from ram occurs. During this the first thing is that data from the dcache is written back and the wait for ram to be done happens. When the ram is finished writing the data it de asserts ramwait and the normal process for a read occurs as above. For a store word coming from the CPU the same sets of processes as above happen with an extra write back at the end. If the data is dirty and a miss then that data is written back to ram and the correct data at the address location is retrieved. This data is then written over and the dirty bit is set. If the data isn't dirty the correct data is retrieved from ram and then that data is written over at the correct location.

The actual data part of the cache was a complex set of logic equations that determined where the data was going to be placed and what was going to be stored with it. Also it had to determine what had to be updated. For hits there was a wire that did a comparison on both the left cache and right cache to see where it needed to go. The hit logic was to look at whether the tag bits and valid bits of each of the caches matched at the index of the incoming address. If it did it would select a certain select wire and tell the cache controller that a hit occurred. The next part of the hit inside the cache data file was to see if it was a write hit from the processor, write miss from the processor, or neither. If it was a write from the processor one of the select wires would be '1' and the CPU write would also be high. This would signal a one of the write enable wires for either of the caches in the 2 way associative to write the data currently on the line. If there wasn't a hit during a write the LRU bits would select which side of the cache would be written to. The LRU bits were the least recently used bits that determined which part of the cache was least recently used and should therefore be written to. A final thing needed in the pipelined portion of the dcache was the updating of the bits relating to the data. Anytime a cache was written to because of a read the data had to be valid. Anytime the cache was written to because of a CPU write the data had to be set as valid and dirty. These signals were used in the cache controller which was described above.

Multi Core Processor:

The multicore processor was two sets of pipeline processors put together with some changes in the instruction set and dcache. Also an additional arbiter and coherence controller was added to facilitate both cores being able to try to access memory. The first thing that was needed was to change the over hierarchy of the CPU. Each core has its own component that then connected to the overall larger arbiter and coherence controller.

The coherence controller needed to be added to make sure that the data between the caches was able to be coherent with each other. Many things were done by the coherence controller to make sure that this happened and some of them were because ease of design. One thing that we did to make the design easier and more straight forward was to not share data between the caches. Anytime another processor would need data that the other block had that was dirty we would write it back. Another thing that was done was if a store word was happening on one processor and the other hand the data in shared mode it would invalidate that data in P2 and then P1 would access ram and get the data then write to it and put it in modified state. The snooping into that caches was done asynchronously as the values of the addresses needed on one core were piped straight to the other core and the information came back instantaneously. This allowed for state transition diagram for the coherence controller to be simplified greatly. The coherence controller state diagram would change in many ways based on what was coming from the CPU. If the CPU was trying to write and needed to access memory it would generate a BusReadX and check the state of the other cores value at the address via the snoop address. If the state was just valid it would invalidate it and and the continue on with reading ram and then toggling the processor. One design idea we came up with is to always have the other processors wait high so if it comes to a spot where it needs ram access it won't interrupt an already acting process. After the ram is read this would toggle the correct core to continue on and then anything else waiting would go. If C2 was dirty then it would write back the data and then continue with the process above. The state diagram has these cases for both Core 1 and Core 2.

Another place the coherence controller would have to write back data to ram was when C1 executed a load word and core 2 had a dirty copy of the data. Core 2 would be signaled to write back that data and then the dirty would be disabled. Then it would toggle the processor to continue on and then to a simple read as above. The final needed states were writes when both were invalid. This would access ram to get the data and then return to the idle state. All of this can be seen below in the state transition diagram. The arbiter which controlled all of the access flow to memory was simply a priority mux that allowed the information from the coherence controller to come first followed by the information from core 1 then core 2. For ease of design we had a bit that would toggle every clock to allow core 2 to access ram even if core 1 didn't want something. This was needed because if core 1 didn't want something and it was core 1's turn to access ram it would be stuck waiting until core 1 accessed ram at some point in time. The coherence controller was given priority for the same reason that the pipeline processor allowed the dcache to go first. This was because the memory access of the dcache would need to finish before the pipe could continue on.

The dcache in the multi core design was relatively the same as the pipeline with a few changes. The state transition diagram had to be changed to allow for the writeback of a piece of data that was snooped and seen that it was dirty. If this happened a snoop writeback signal would be asserted and the core would write back its data so that the other processor could use it. Also added to the dcache was a way to invalidate. This was needed because invalidation is now part of the multi core processor that wasn't in the pipeline processor. If an invalidate signal was sent by the coherence controller the data matching the snoop addresses index and tag would then invalidate it at the point. Also added to the cache was a link register and and a link valid bit. These were used for the newly implemented LL/SC commands. These commands were new in the CPU portion.

Differences between Processors:

Above many differences are shown between the two different processor designs. One set of differences was in the dcache. The differences were that a link register and link valid register were needed for the multi core processor to enable locking and unlocking of a processor. Another change was that an invalidation had to occur if the other processor was writing to a shared block of data. This wasn't needed in the pipeline processor because the data was always valid if it had been read into the cache before. A final thing that was different was the state transition diagram of the multi core. A snoop writeback stage was needed to allow the processor to write back and break out of any of the states if the other processor was currently trying to do something.

Another major difference between something that was located in both processors was the arbiter. Although the arbiter did allow dcache to go first by way of the coherence controller it had to deal with multiple cores and the coherence controller in the multi core case as opposed to just the dcache and icache. The arbiter in multicore allowed the coherence controller to go first then icache1 then icache2.

The final thing discussed above is the addition of a coherence controller which isn't needed in the pipeline processor. This is discussed in detail in the multi core processor portion.

Figure 1: Pipeline Processor

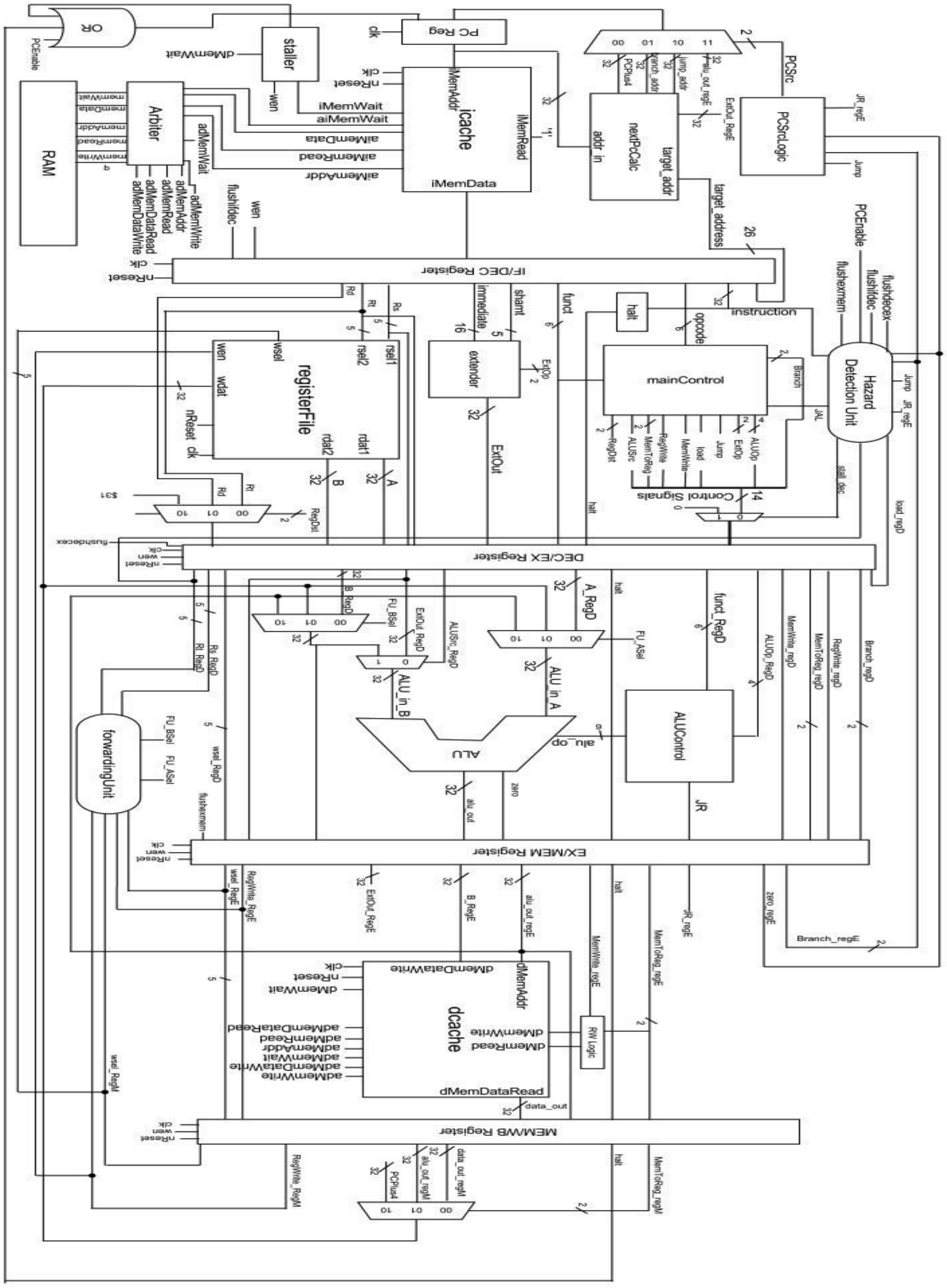

Figure 2: Cache Design

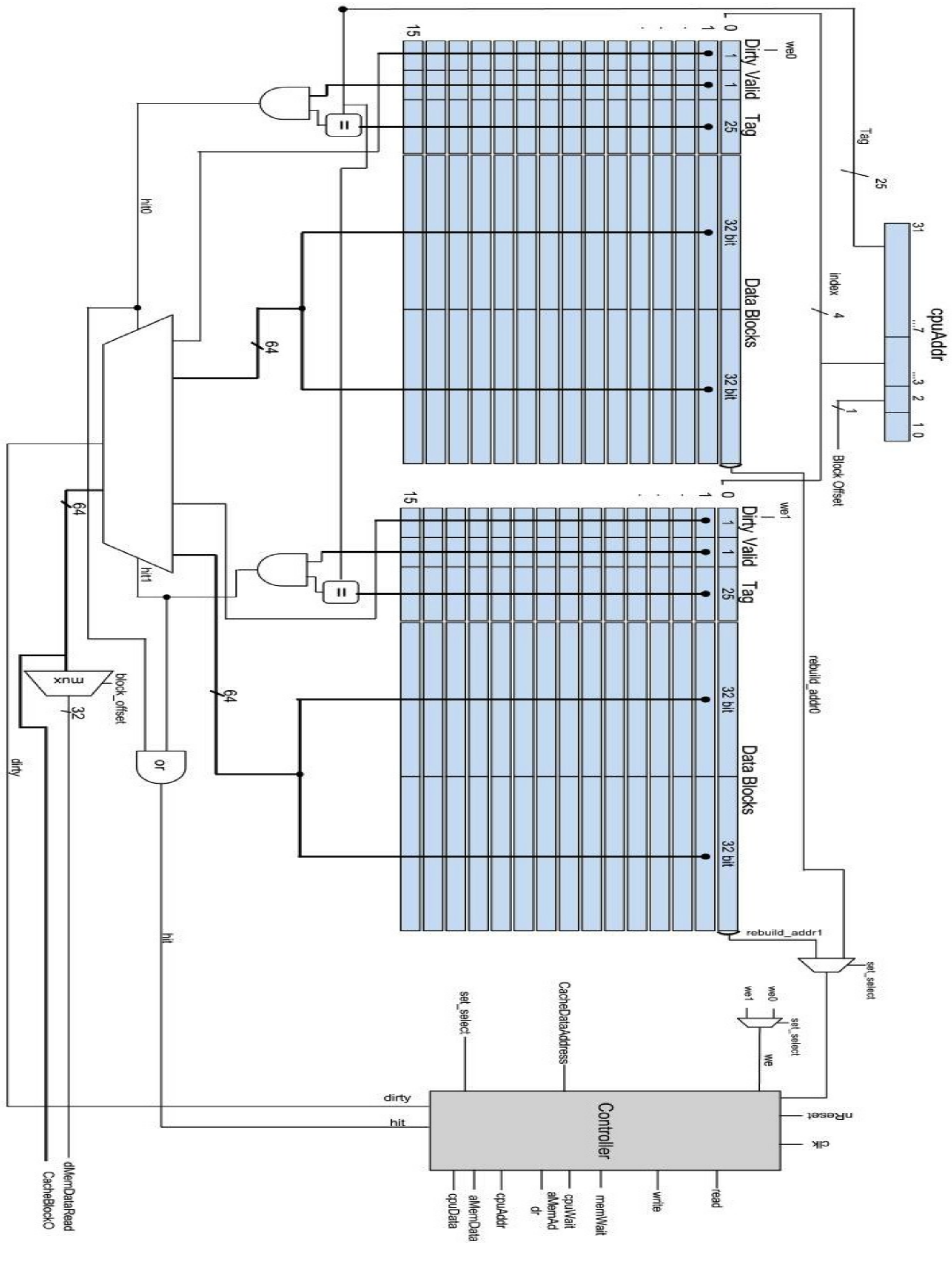

Figure 3: MultiCore Design

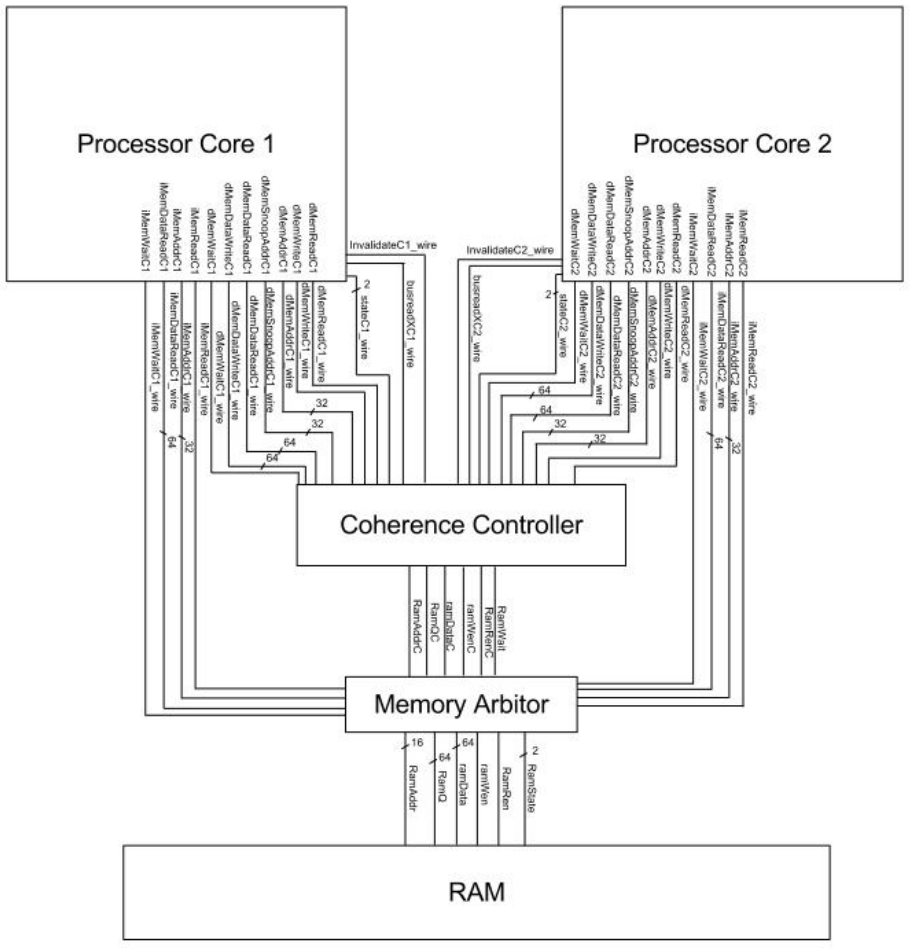

Figure 4: dcache state diagram

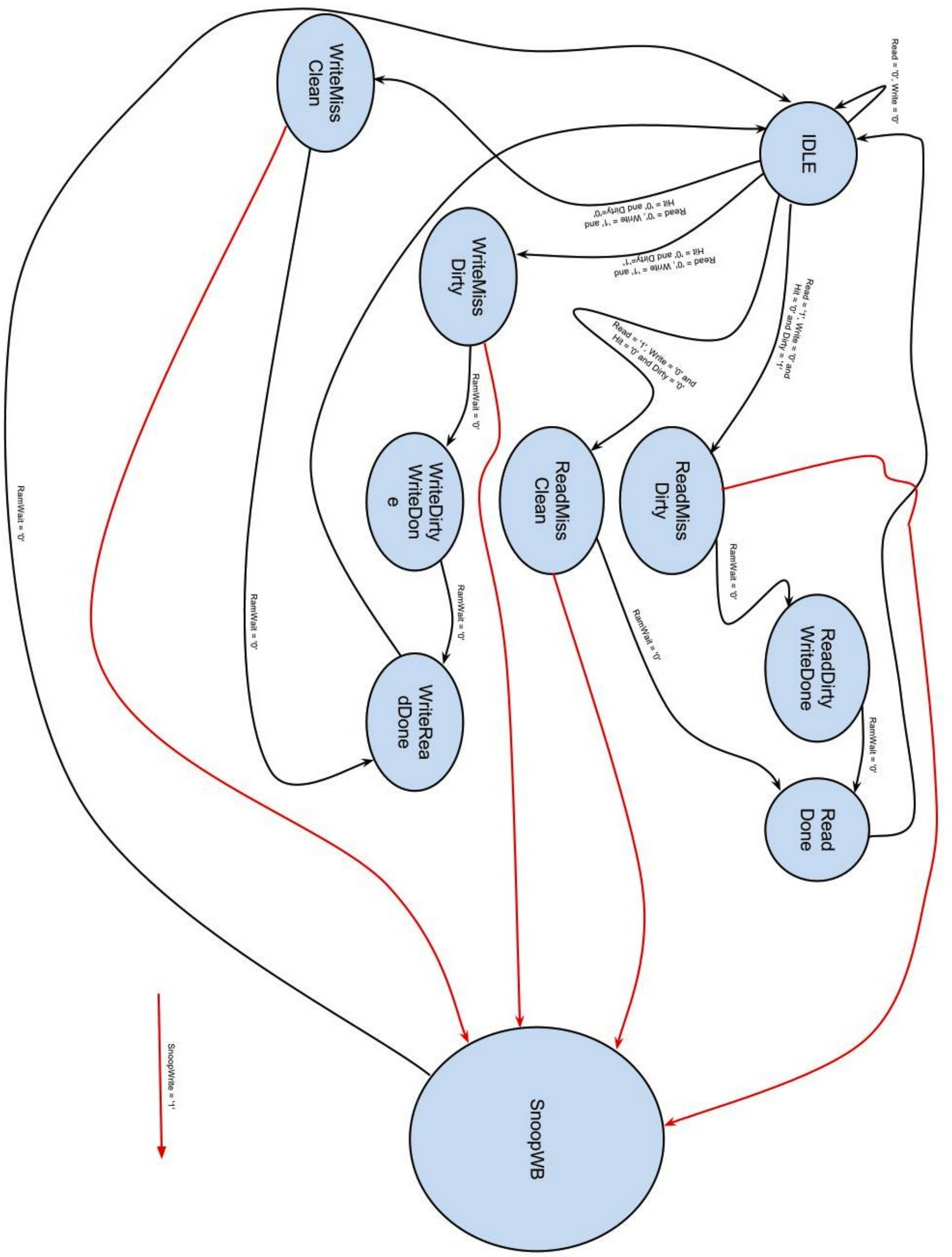

Figure 5: Coherence Controller State controller

4.4 Processor Debugging

In the memout file that results from execution of the code, the value at address 0x0090 should be 0xDEAD when it is actually 0xDEEF. There are several problems that could have caused this error. The first possibility is that the store conditional instruction does not invalidate the link register of the other core when it completes. This would cause both cores to acquire the lock at the same time and, in turn, cause a race condition where the processor that writes last, in this case core 2, will be the one whose data is stored as the result. To look for this error, one would look at both of the link registers in the data caches. If both are valid at the end of the lock routine and continue execution than this is the problem.

Another possible cause for this error is that dirty blocks are not being written back to main memory when the other core requests that address. For example, if the lock works as expected and processor one acquires the lock first, it will subtract 66 from res which will result in 0xBEAD. This will be stored in processor one's cache. Then when processor two tries to load res, it loads it from main memory instead of having processor one write its dirty value back. This will result in 0xDEEF in processor two's cache. Finally, when both caches dump at the end of execution or somewhere in between, processor two's cache gets dumped last and 0xDEEF is stored as the result in main memory. To distinguish between these two bugs the link registers would be looked at in the first case. If those are fine then the look inside the cache to see if when it loads a new value it loads the correctly updated value.

4.5 Results

Dual Core Processor:

The dual core processor took two copies of the pipeline with cache design and hooked them up together. A coherence controller and memory arbiter were placed in between them to preserve coherency and delegate who got access to RAM when.

Performance Results

	Results
Maximum Clock Speed	14.55 MHz
Average Instructions per Clock Cycle	~0.67
Instruction Latency	~ 5 * clock period
Number of FPGA Logic Blocks	17,759
Number of Registers	11,345

The critical path of the dual core processor was 71.826ns. It starts in the dcache state machine, goes through a combinational feedback loop in the dcache, and ends in dcache block memory registers. This was also not where the critical path was expected to be. It was expected to be between ram, the coherence controller, and either the icache or dcache.

Pipeline with Caches:

The pipeline processor with caches took the pipeline processor previously implemented and added an instruction and a data cache. The instruction cache was provided and only the data cache was designed. The data cache is a two way set associative cache with 64 bit blocks.

Performance Results

	Results
Maximum Clock Speed	38.74 MHz
Average Instructions per Clock Cycle	~0.37
Instruction Latency	~ 5 * clock period
Number of FPGA Logic Blocks	11,218
Number of Registers	5,601

The critical path for the pipeline design with caches was 17.636ns. It was from the 'EXMEM' pipeline latch through the dcache, and back into the 'MEMWB' register. This was not where the the critical path was expected to be. It was expected to be in the same place as it was for the pipeline processor: through the ALU barrel shifter.

4.6 Conclusions

During these labs both a pipeline processor with cache and dual core processor with caches were designed to operate correctly and run machine code and display the correct output. Both of the designs worked for every test program that was given and ones we created. For the pipeline processor we were able to test that the caches were able to load in data and store data correctly. When we ran the loaded files every output was the same as the simulator and we were able to verify that by looking at what was happening inside the processor. During this we were able to see that the cache was capable of loading in a 64 bit value and then having that hit twice in a row before a new value was needed to be loaded. When doing stores we were able to see that it would overwrite the correct data and then make that bit dirty and store it back when needed.

For the multicore processor we were able to test the the dual core designed programs and get the correct output. The main difference that had to be tested between the pipeline processor and the dual core processor was coherency between the two processors. We found that this worked also because the programs that had to share data and locks both worked and were able to get the correct output at the end. We were able to verify this by looking inside the cores and seeing if the data was written back when we needed it to be coherent.

The lessons learned from these labs are that there are many advantages and disadvantages to dual core processors and pipeline processors. The advantages of the pipeline processor with caches is that it was easier to implement than the multicore processor because only one set of data was being iterated on at a time. A disadvantage to this pipeline processor is that it is only capable of executing a commands in a set order and no parallelism can occur. The advantage of the multicore processor is that it can parallelize instructions and therefore run a program like a mergesort faster. The disadvantages of the multicore were that it was much more complex to design and build and also coherence between two fully working pipeline processors had to be managed. A final disadvantage was that one processor would have to stall for a longer time than in the pipelined case if both processors needed to access ram at the same time.

The uses for processors are many. These processors are probably not fast and efficient enough or fast enough to be used in real general purpose computers. However, they could possibly be used in an embedded application or as a peripheral controller in a modern computer.