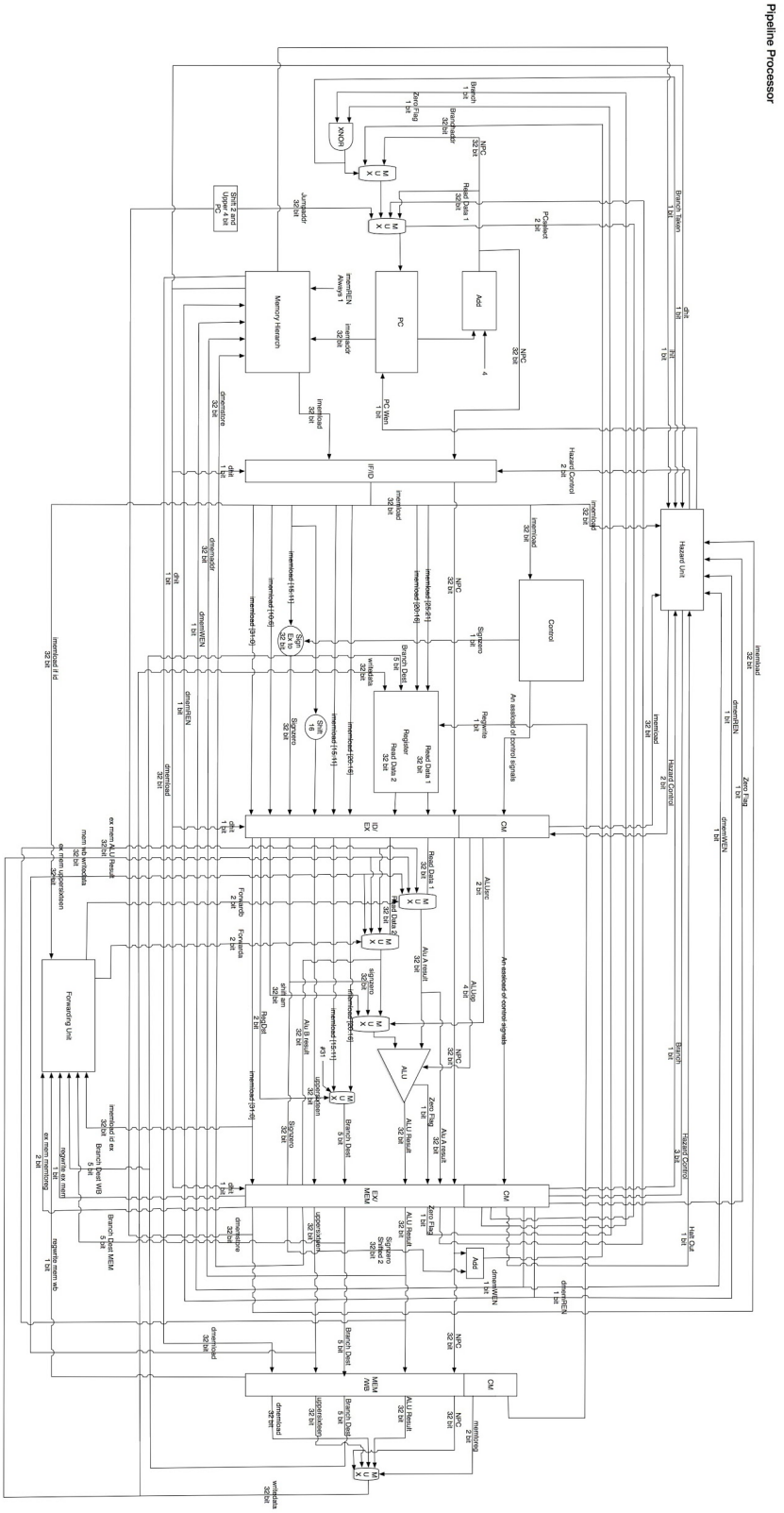
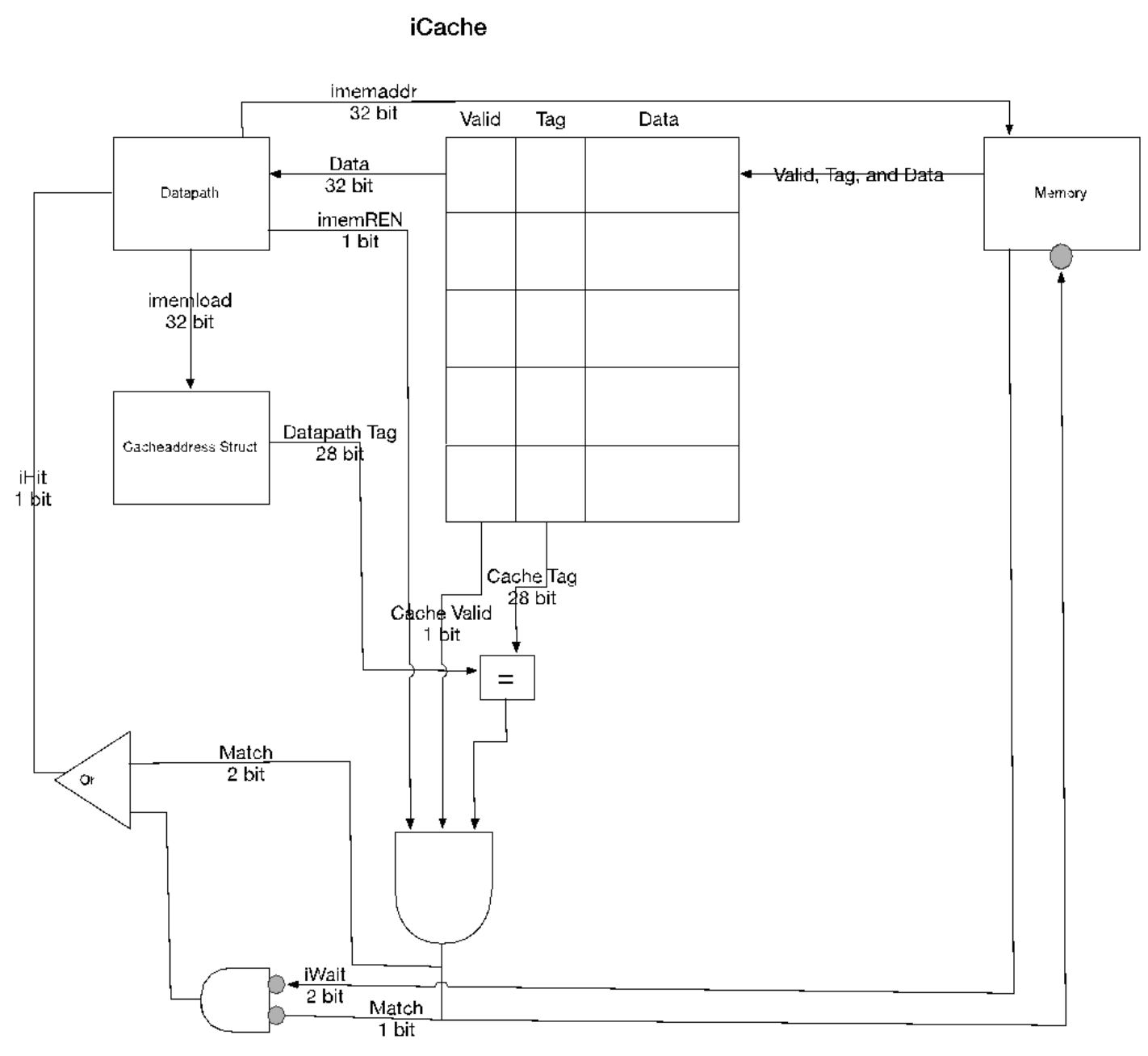
4.3 Cache and Coherence Design



4.3.1 iCache

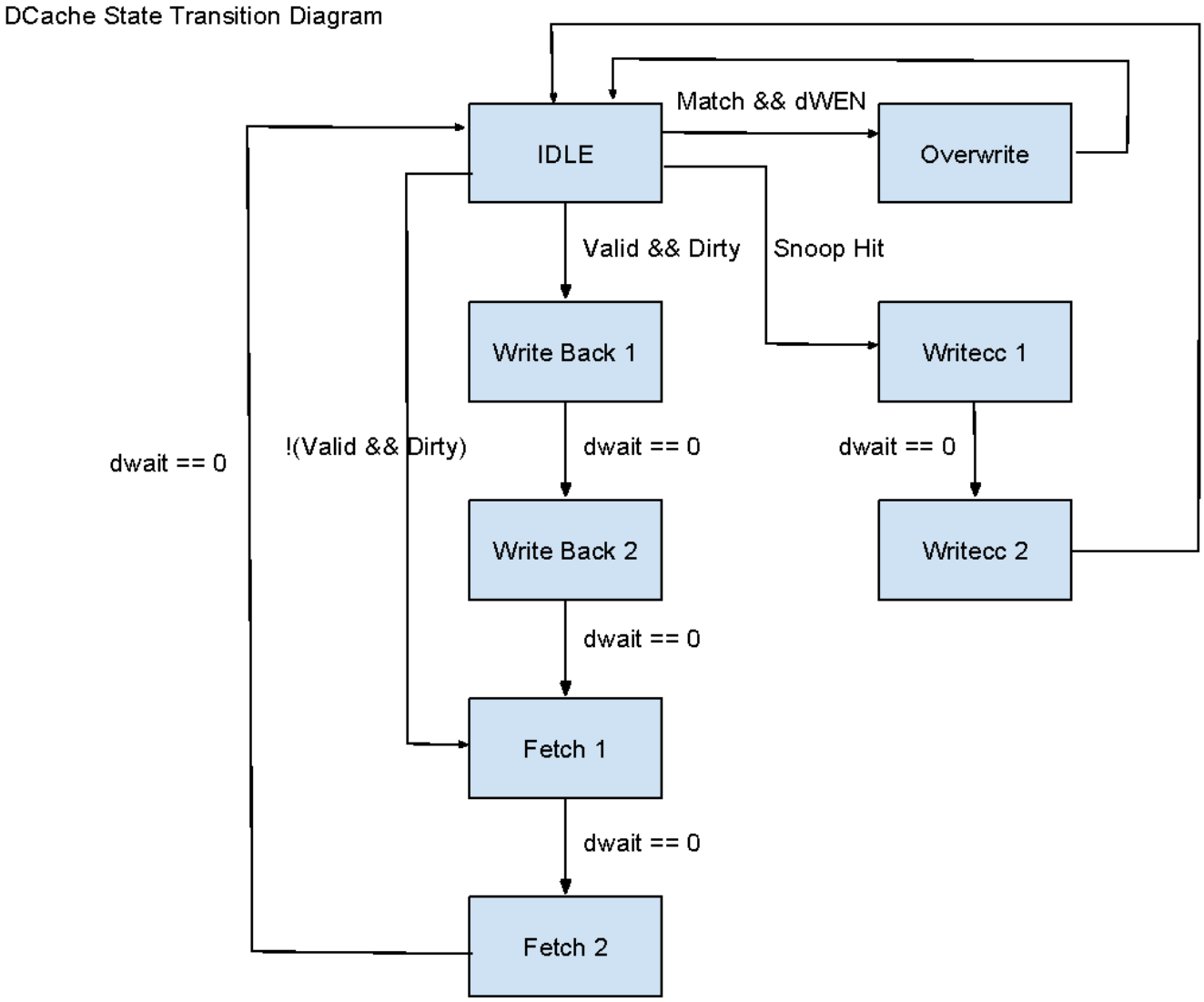

iCache
imemaddr
32 bit
Valid
Tag
Data
Datapath
Data
32 bit
Valid, Tag, and Data
Memory
imemREN
1 bit
imemload
32 bit
Cacheaddress Struct
Datapath Tag
28 bit
iHit
1 bit
Cache Tag
28 bit
Cache Valid
1 bit
=
Or
Match
2 bit
iWait
2 bit
Match
1 bit

The design of the iCache was relatively simple compared to the dcache. It began by creating a structure for a cache block. This structure included the data, tag, and valid signal. The data was the actual instruction that was pulled from memory, the tag is the identification value for the instruction, and finally the valid is used to mark the cache value as being correct. The signal match is the most important signal we had to add to the iCache design. The match signal was set when the tag from the incoming memory address matched the indexed cache block's tag and if there was an instruction read enable signal from the datapath. When the incoming address matches a cache block, then the datapath instruction data will simply be set to the data from the matched cache block. However, if a match is not detected, the icache will send signals to the memory to retrieve the correct instruction. This leads to the iwait signal to be asserted, and will only be de-asserted by the memory controller when the value is back from memory. On every rising clock edge, the icache will check to see if there's a miss and whether the data is back fro memory. When this is the case, the cache block will update itself with the new tag, data, valid bit, and the instruction will be sent to the datapath. The valid bit only gets changed to zero whenever the icache is reset. This is because there is never a situation where any instruction cache blocks need to be overwritten.

There were several tests done for the iCache. The first of them was to simply check to see if instructions were being stored in the iCache block. This was done by utilizing the system testbench and creating custom asm files. The asm for this test had to be rather short so that it will fit entirely in the icache, without any loops or branches. This way it is easier to confirm via the waveforms that the correct order and instructions were present. Since the cache was originally reset at the beginning of the test bench, all instructions will inherently miss and request the main memory to feed it the correct value. If this all works, then the memory retrieval and miss signals are properly working. The next phase of testing was to see if the icache can correctly detect matches and supply the correct cache data.

This can be done through an asm file that has a loop. The reason for the loops is that the first time through the loop, the cache will miss on all requests and pull data back into the cache blocks so that it is up to date. Thus, every future iteration in this loop will incur an iCache match. By following the waveforms and confirming that the match signal is high on every iteration of the loop except the first one, this would prove that the match function of the icache works properly.

4.3.2 dCache

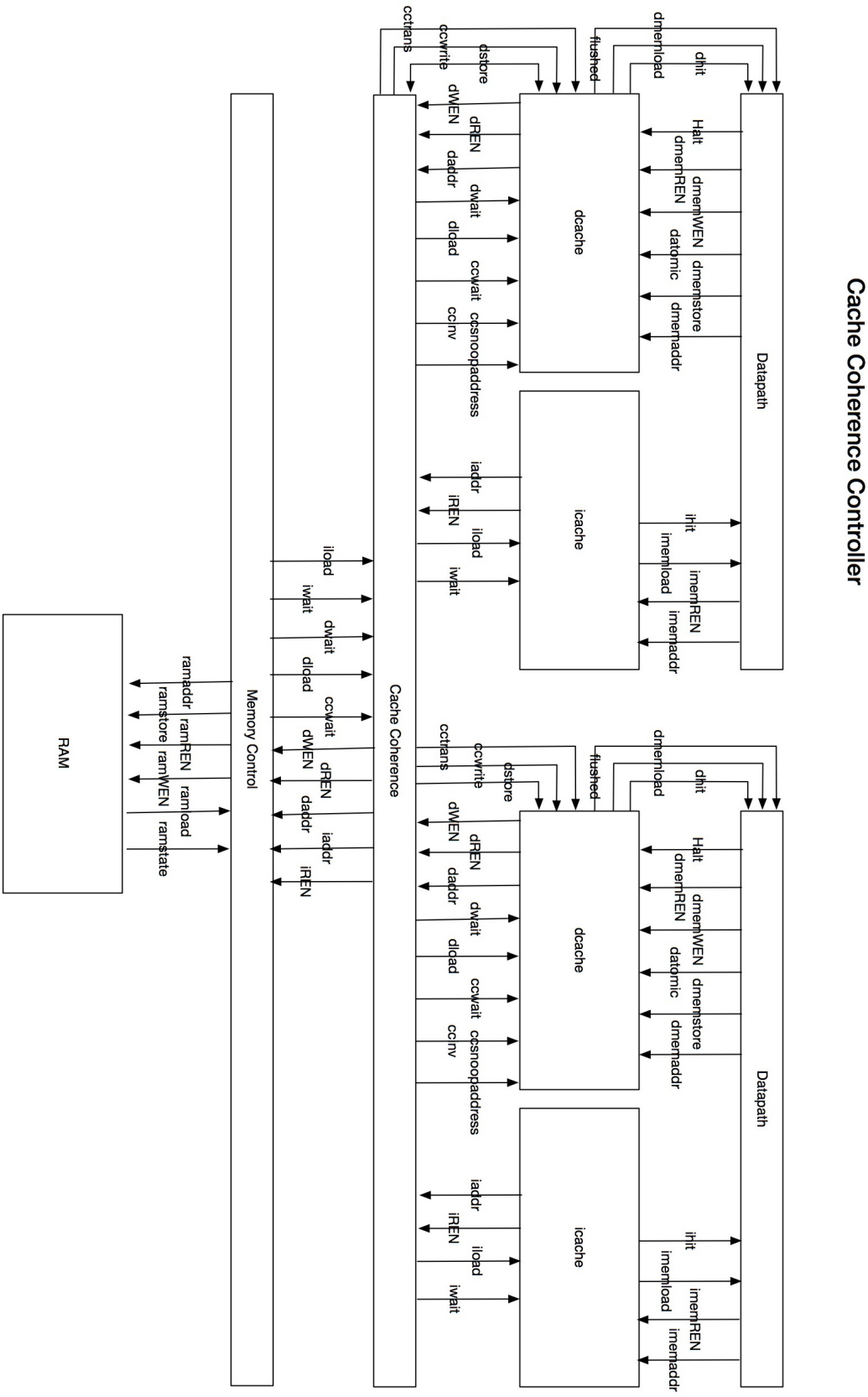

DCache State Transition Diagram
IDLE
Match && dWEN
Overwrite
Valid && Dirty
Snoop Hit
Write Back 1
Writecc 1
dwait == 0
!(Valid && Dirty)
dwait == 0
dwait == 0
Write Back 2
Writecc 2
dwait == 0
Fetch 1
dwait == 0
Fetch 2

The first step of designing the dCache was creating the state transition diagram. The above diagram shows the dCache state transition diagram with support for cache coherence. The main difference is the addition of the writecc1 and writecc2. These two changes will be explained in more detail in the coherence part of the preliminary report. The state machine transitioned based on a read or write request from the datapath. If it's a read request, then it checks to see if the there's a match in either of the ways. If it did match, then it would continue to check the valid and dirty bits. The dirty bit determines whether the dCache would need to write back a value that was "dirty" meaning it was modified in cache, but not written back to memory. The valid bit determines if the value was correctly pulled from memory at one point in time. If the there was a cache miss, asserted valid bit, and it's dirty, the dcache will proceed to the write back stage. This is due to the design choice of using the write-back method which only writes back to memory when needed due to the high clock penalty for accessing memory. There are two write back stages since each cache block in the dcache has two words, and due to the constraint of the memory only being able to handle one read or write operation at a time, the two words must be split into two states to store the entire block. After these values have been properly stored in memory, the next step is to actually fetch the correct value needed for the datapath from memory. The dcache may also enter this state when a cache miss occurs and the valid or dirty bit is zero. Similar to the write back state, fetch will also require two state due to the constraints of memory in the system. Once the memory transaction has completed, the values will be stored into the cache and passed on to the datapath and the state will transition to idle. The last state in the transition diagram is the overwrite state. When in idle, and the dcache receives a request for memory write and it matches one of the dcache ways, then it will proceed to overwrite the word in the set. If the dCache receives a request and there is a cache miss, but both ways are valid and dirty, eviction will occur. There will be a recent bit value attached to each block in a way, and from this bit you can choose which block to evict. At the same time of

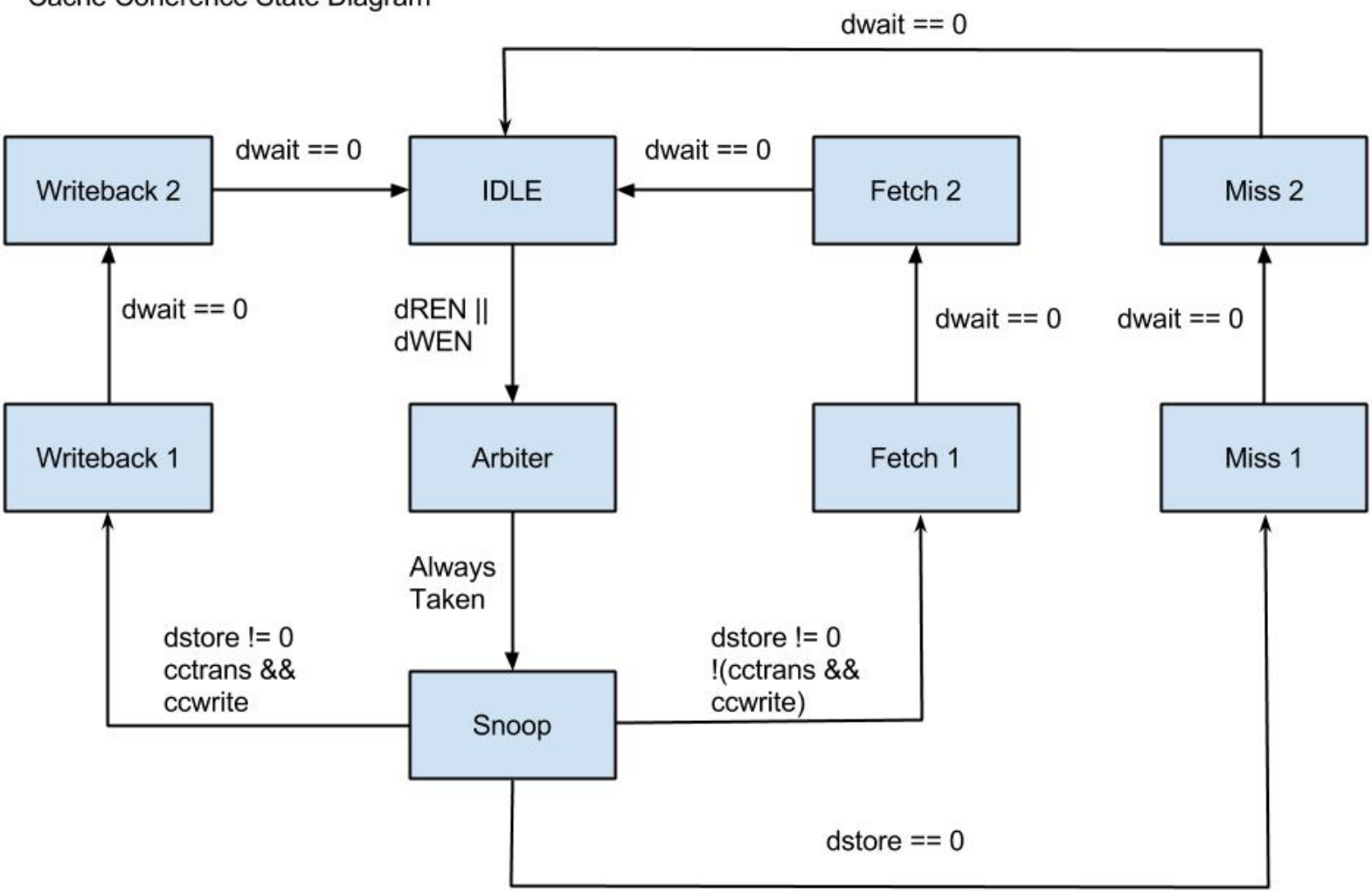
eviction, the dirty bit will be set, so that it may be written back to memory at a later time. FInally, when the dcache receives a halt, it will begin to flush out the entire cache by writing it back to memory if it's valid and dirty.

For testing purposes, a counter was added to the code and incremented every time there was a cache miss in either of the ways. The final value for the counter is stored at the memory location 0x3100 after flushing has finished. When simulating the asm file using the compiler, there is an option to enable simulating with cache hits which stores the ideal hit count at memory location 0x3100. The first asm file that was created, simply tested to see if dCache was able to store and load values into the cache. Once general functionality had been confirmed, asm files from previous labs were used to see if the hit count matched the simulated hit count. These asm files were much more complex than the initial tests, and they included many loops and read/write operations.

4.3.3 Coherence



Cache Coherence State Diagram



Coherence was the most difficult aspect of the memory hierarchy to design and build because it required changes to be made to pre-existing dCache and memory controller designs. Only small changes were required in the dCahce compared to the memory controller, which essentially had to get redesigned from the ground up.

Although the changes to the dCache were only slight, they were integral to the overall design. The first thing that needed to be added was the ability to snoop into the tag array. The logic for a snoop hit/miss was nearly identical to that of the pre-existing cache hit/miss signal. The only difference was where the tag in question was being pulled from (snoop address or cache address). The next addition to the dCache was to add states for cache to cache transfers. These states would be transitioned to if the cache received a ccwait signal from the memory controller a snoop hit in either one of the ways was found. Within these states, the cache would send out its data on the bus for a cache to cache transfer. Because the cache being snooped into doesn't have any knowledge of whether its data is being written to memory, the other cache or both; we were able to reuse these two states for every situation. After both words in the block are written, the cache will got into a state the will reset the valid and dirty bits if necessary based on ccinv signal from the memory controller. The final change required in the dCache was to handle the cctrans and ccwrite signals. These signals were used in different ways depending on what role the cache was playing (snooper, or snoopee). From the perspective of the snooper, these signals were used to tell the memory controller what, if any, state transition was occurring. See Appendix A for signal values and definitions. In order to determine the state transition, we used a combination of the cache hit/miss signals, the dirty bit, and dmemREN/dmemWEN. From the perspective of the snoopee, these signals were used in the memory controller to calculate the ccinv signal. See Appendix A for signal values and definitions.

There were several phases of testing for the coherence controller. We began with the simplest path through which consisted of a snoop miss on a data

read request. This caused the snooper to go directly to memory and avoid coherence completely. We tested this via a very simple asm file. One processor simply halted, and the other only had a load word. This allowed us to look at the waveforms and see the appropriate state transitions in the cache and the coherence controller. The next step in testing was to add in one type of cache to cache transfers. We decided that it would be easier to test the for a clean snoop hit first because we could avoid dealing with going to memory. In order to test this we had another asm file that had a load in one processor followed by nops, and nops followed by a load at the same address in the other processor. Once again this allowed us to go to the waveforms and see the appropriates state transitions occurring. For this test, we were also able to see data being transferred to the bus from the snoopee and going into the snooper. The next test was to do a cache to cache transfer on a dirty block of memory. For this test, we utilized an asm file that did a load and save word in one processor while the other was running nops. Following the save word, the other processor did a load word on the same address. This test not only does a cache to cache transfer, but also writes the dirty block to memory at the same time. The remaining tests that were run comprised of a combination of two or more of the other tests described above. We tested various combinations and in various orders to try and get as much test coverage as possible.

4.3.4 Synchronization

To implement synchronization between the two processors, we had to make changes in the datapath, caches and memory controller. The first change that needed to be made was to implement the datomic flag. This flag is set in the Mem stage in the datapath anytime an ll or sc instruction is being executed. That flag then goes down to the dCache where it sets or checks the link register and valid bit. If testing the valid bit succeeds, a one is returned to the datapath to notify that a write is possible. If the testing fails, a zero is returned and the program must try another time. The link register is invalidated in situations such as a matching coherent store and a match to the snoop address.

The testing for this was fairly straight forward. We had pre-existing asm files that incorporated ll and sc, so we utilized those to fully test our design.

Appendix A

State Transition	logic	cctrans	ccwrite
None	Match, doing a read	0	0
S -> M	Match, doing a write	1	1
I -> S	No match, doing a read	1	0
I -> M	No match, doing a write	1	1

Snooper Perspective

Logic	cctrans	ccwrite	ccinv
Snoop hit, dirty data	1	1	1
Snoop hit, clean data	1	0	0
Snoop miss, clean or dirty data	0	0	0

Snoopee Perspective