Final Report: Dual-Core MIPS Microprocessor

ECE 437: Computer Design and Prototyping

TA: Eric Villasenor

December 9, 2011

Dan Ehrman and Brian Kelley

Executive Overview

In this report, we present the design, implementation, and results of a dual-core, 32-bit, pipelined, MIPS-based microprocessor with one level of split instruction and data cache. The processor was implemented in VHDL and tested both in ModelSim for source- and synthesis-simulation as well as on an Altera Cyclone II FPGA for hardware verification. In accordance with standard MIPS protocol, the processor is based on a five-stage pipeline (instruction fetch, decode, execute, memory, and write-back) and contains 32 32-bit registers. The multi-core processor implements the shared memory model, wherein a coherence controller maintains cache coherence between the two processors using cache invalidation. The architecture also provides support for LL/SC instructions to provide the programmer with a means of write atomicity.

Although there were numerous challenges in design, the team was successful in implementing a processor capable of meeting all of the target expectations. Throughout design, careful attention was paid to the modularization of code as well as the planning for future modifications. Furthermore, extensive testing was performed on the processor through each stage of the design process so as to ensure a working processor before moving on with additions and improvements. The process as a whole demonstrated the importance of various common optimizations, such as instruction and data caches for reducing average memory access time and multiple cores for allowing parallel computation, as well as the difficulties and tradeoffs involved in achieving this higher level of complexity. In the following report, we detail the creation of this processor from an earlier pipelined design, demonstrate an example of the kind of debugging involved in its design, as well as provide actual results when tested on a handful of MIPS assembly programs.

Processor and Cache Design

The processor as it exists in its final state--a dual-core microprocessor with fully functional cache coherence and an LRU replacement policy--is the result of a continuing evolution from an earlier five-stage pipelined uniprocessor design. In building on the earlier design, emphasis was placed on both improved functionality and performance as well as maintaining a means of modification with minimal time required for implementation. In doing so, the modularity of code was heavily enforced throughout every addition so that upon implementing future modifications, little would have to be rewritten or disassembled to make changes. A primary example of this is the frequent use of “wrappers” (or interfaces) to contain smaller blocks of code. By using a clearly defined interface and encapsulating all local functionality within their sub modules, new functional blocks could be easily implemented with the existing system, and often by another team member, with minimal effort or potential for mistakes.

In expanding on the previous pipelined processor design to create a dual-core processor, the team faced several challenges. First, as is the case with any shared-memory design, some level of cache coherence had to be developed to avoid either processor operating on stale data. Because this was of no concern in the uniprocessor, the original cache hierarchy involved both the instruction cache and the data cache communicating directly with the memory arbitrator, which acts as a priority multiplexer to selectively grant memory access to either the instruction-fetch (IF) or memory (MEM) stages of the pipeline. However, in the multiprocessor architecture, this design had to be modified so that the two data caches could communicate with each other. (Instruction data was assumed modifiable, and thus the instruction caches were left as independent entities, still communicating directly with the memory arbitrator.)

The solution to the cache coherence problem was to implement a coherence controller, which acts as a middleman between the two data caches as well as an interface with the memory arbitrator. Within the caches themselves, the modified-shared-invalid (MSI) protocol was used both as a means of keeping track of the state of each block of data as well as notifying the other processor(s) of any changes through the coherency controller. As states of individual blocks change, the activity is routed back to the coherence controller, where it takes the appropriate action in response. The

coherence controller was designed to exploit this data cache modification by using a snooping protocol to communicate applicable changes to each cache in the system. This “snooping” presented some challenges itself in that while the coherence controller attempts to gain access to a particular block in a cache, potentially changing its state as well, the corresponding processor may be accessing that block simultaneously. The chosen solution was to provide a second read and write port for use by the snoop controller and to arbitrate write access such that the snoop controller is always granted priority over any processor accesses to that location.

Inside the data caches, the basic array structure first implemented with the pipelined uniprocessor was left unchanged. Each data cache is comprised of 32 two-way associative sets, with each block containing two 32-bit words of data. The fact that there are two words per block (64 bits of data) did complicate design to some extent due to the fact that there is a maximum memory bandwidth of 32 bits. The solution, when either reading or writing back an entire block from or to the coherence controller, is to send and receive data one word at a time, back to back.

Another issue that arises from the implementation of a shared memory system is that of locking and unlocking data to provide write atomicity. For this processor, the MIPS-standard LL/SC write atomicity method was employed. Functionality was added both to the processor pipeline and to the data cache to accommodate this change whereby an LL (load linked) instruction stores the address of the loaded data in a special link register contained within the cache. Likewise, an SC (store conditional) instruction is implemented using a signal originating from the data cache, which is asserted when the address in the linked register matches that being written to (and has not been invalidated). This signal is used in a multiplexer in the MEM stage of the pipeline to determine whether or not a value should be written to memory as well as whether a value of 1 or 0 should be written to the destination register to indicate success or failure in completing the atomic store operation.

Processor Debugging

As a matter of exercise, the team was presented with a sample assembly file entitled “debug.asm” and its associated “memout.hex” file (generated from execution),

which is known to reveal at least two hardware design errors in a particular multicore processor. The problem description also indicates that the memory and pipeline components of the processor are in working order and that the cache hierarchy is the only subsystem in question.

This first step is to identify any abnormalities in the hex output, which are obtained simply by running the assembly program through a known working processor or simulator. After completing this operation, only one difference between the two hex outputs was made apparent, as is shown in highlighted text in Table 1.1 below:

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`
`lock:`	`:040004001408FFFEDF`	`:040004001408FFFEDF`
`ll  $t0, 0($a0)`	`:0400050021080001CD`	`:0400050021080001CD`
`bne  $t0, $0, lock`	`:04000600E08800008E`	`:04000600E08800008E`
`addi  $t0, $t0, 1`	`:040007001008FFFBE3`	`:040007001008FFFBE3`
`sc  $t0, 0($a0)`	`:0400080003E0000809`	`:0400080003E0000809`
`beq  $t0, $0, lock`	`:04000900AC800000C7`	`:04000900AC800000C7`
`jr  $ra`	`:04000A0003E0000807`	`:04000A0003E0000807`
`unlock:`	`:04000B0027BDFFFC12`	`:04000B0027BDFFFC12`
`sw  $0, 0($a0)`	`:04000C00AFBF000082`	`:04000C00AFBF000082`
`jr  $ra`	`:04000D003404006057`	`:04000D003404006057`
`mp0:`	`:04000E000C000003DF`	`:04000E000C000003DF`
`push  $ra`	`:04000F00340A02406D`	`:04000F00340A02406D`
`ori  $a0, $zero, l1`	`:040010008D48000017`	`:040010008D48000017`
`jal  lock`	`:04001100250920009D`	`:04001100250920009D`
`ori  $t2, $zero, res`	`:04001200AD490000F4`	`:04001200AD490000F4`
`lw  $t0, 0($t2)`	`:040013003404006051`	`:040013003404006051`
`addiu $t1, $t0, 0x2000`	`:040014000C000009D3`	`:040014000C000009D3`
`sw  $t1, 0($t2)`	`:040015008FBF000099`	`:040015008FBF000099`
`ori  $a0, $zero, l1`	`:0400160027BD0004FE`	`:0400160027BD0004FE`
`jal  unlock`	`:0400170003E00008FA`	`:0400170003E00008FA`
`pop  $ra`	`:04008000341D7FFCB0`	`:04008000341D7FFCB0`
`jr  $ra`	`:040081000C000083EC`	`:040081000C000083EC`
`l1:`	`:04008200FFFFFFFF7E`	`:04008200FFFFFFFF7E`
`cfw  0x0`	`:0400830027BDFFFC9A`	`:0400830027BDFFFC9A`
`org  0x0200`	`:04008400AFBF00000A`	`:04008400AFBF00000A`
`ori  $sp, $zero, 0x7FFC`	`:0400850034040060DF`	`:0400850034040060DF`
`jal  mp1`	`:040086000C00000367`	`:040086000C00000367`
`halt`	`:04008700340A0240F5`	`:04008700340A0240F5`
`mp1:`	`:040088008D4800009F`	`:040088008D4800009F`
`push  $ra`	`:0400890002509FFBE88`	`:0400890002509FFBE88`
`ori  $a0, $zero, l1`	`:04008A00AD4900007C`	`:04008A00AD4900007C`
`jal  lock`	`:04008B0034040060D9`	`:04008B0034040060D9`
`ori  $t2, $zero, res`	`:04008C000C0000095B`	`:04008C000C0000095B`

`lw $t0, 0($t2)`
`addiu $t1, $t0, -66`
`sw $t1, 0($t2)`
`ori $a0, $zero, 11`
`jal unlock`
`pop $ra`
`jr $ra`
`res:`
`cfw 0xBEEF`
	`:04008D008FBF000021`
	`:04008E0027BD000486`
	`:04008F0003E0000882`
	`:040090000000DEADE1`
	`:040FFE0000000008E7`
	`:041FFE0000000208D5`
	`:00000001FF`
		`:04008D008FBF000021`
		`:04008E0027BD000486`
		`:04008F0003E0000882`
		`:040090000000DEEFE1`
		`:040FFE0000000008E7`
		`:041FFE0000000208D5`
		`:00000001FF`

Table 1.1: Error-inducing assembly program along with its associated hex output files for both a working and non-working processor

As can be seen in the table, the bug(s) in the processor are manifested as erroneous data being stored at memory location 0x240 (represented in the table as 0x240/4=0x090), or simply “res” as in the assembly file.

Upon inspection of the assembly program, it can be seen that the intention of the program is to perform some operation on the data in a memory location shared between two processors and then write the data back to memory. When tracing back the expected output, it can be seen that the first processor (Processor 0) should gain precedence over the second processor (Processor 1) in acquiring the lock on the shared memory location. The proof is as follows: Processor 0 loads the word “0xBEEF” from “res” and adds 0x2000 to it before writing it back, thus changing the stored data to “0xDEEF.” Subsequently, Processor 1 loads “0xDEEF,” subtracts decimal 66 from it, thus changing its value to “0xDEAD,” and then writes it back to memory. This process could also be reversed (Processor 1 then Processor 0), but the two cannot be intertwined. Clearly there is a problem in the processor that is preventing this write sequence from occurring as expected.

One potential explanation for the error is that Processor 0’s write to the shared location is never broadcast to the rest of the system. This could occur for a variety of reasons, but clearly Processor 1’s cache is failing to see that the memory location has been modified by the other processor. Another potential source of the problem is that the lock on the memory location is never truly obtained. Had the lock been obtained by either processor, the other would detect that write atomicity on the lock location had been broken and thus would not read stale data from location “res.”

Upon further analysis, it becomes clear how the above errors might propagate to the resulting memout.hex. First, both processors actually obtain the lock, allowing each of them to “think” that they have exclusive control over that memory location, thus allowing them to operate on stale data without knowing it. Furthermore, each processor modifies the data locally in its cache without sending an appropriate signal to the coherence controller to invalidate the data in the other processor’s cache. Thus, upon writing back the data, it is simply a race to gain control over the memory first; in the case of the processor in question, Processor 1 writes its data back first, which is then overwritten by the data “0xDEEF” from Processor 0.

In searching for the true source of these errors, the first obvious place to look in the waveforms would be those corresponding to the data cache’s MSI state in each processor. Doing so would allow one to verify what data is actually contained in the processors’ caches at any given point in time (before, during, or after load and store operations). Next, one might look at the bus signals (as well as the data) flowing between the two caches through the coherence controller. This would show if there are any issues with either the coherence controller or the caches’ state machines themselves. After localizing the problem to any one of these various modules, further inspection would follow accordingly. To determine the specific source of the error, and potentially rule out one of the two theories, another good first step would be to develop a test whereby a lock is theoretically obtained by each processor, and then the data is written back to *different* locations. Doing so would avoid the problem of losing whatever data was overwritten by the most recent write operation.

Results

Tests

In order to compare our two processors, we used two different programs. The first was a merge sort program. This displays high cache usage and alternating loads and stores. The dual core version does an insertion sort on two 50-word datasets, and then synchronizes and the second processor does a merge sort on the two datasets. The single-core version is similar, but it runs on a single processor. There are still two insertion sorts, but they are performed one after the other.

The other test is a search program. There are 400 records to be search through for a pre-specified key. This program does mostly loads, then a store at the end of the program. With the high number of records, the cache is refilled with every other load. However, nothing is ever written to those locations so there is no write-back. The single-core version searches through all 400 records sequentially. The dual-core version splits the dataset into two and each processor searches its set. For both programs, the key is ten places from the end of the dataset.

Pipelined Processor with Instruction and Data Cache

Our pipelined processor with cache is an add-on to the pipelined processor we had already designed. We have added independent instruction and data caches. The instruction cache is a 16-set, one-word, direct-mapped cache.

The data cache is a 16-set, two-way, two-word, partly associative cache. The cache implements write-back with an LRU replacement policy. The cache supports 32-bit, word-granularity address (30-bit addresses). Cache hits take one cycle. Cache misses to a clean or unused cache position take 18 cycles (memory latency is eight cycles).

The critical path of the processor is contained within the data cache. It originates at the state register, goes through the data array, back to the controller to determine which block to select, and finally to the data output to the processor. This path is pretty much where we expected the critical path to occur. We actually could have reduced this path by half if we had clocked our cache on the same edge as the processor. It is currently clocked on the falling edge, which was necessary with the non-cached processor, but probably could be removed with a bit of work in the cached one.

Although Figures A.2 - A.5 are the diagrams of the pipeline for the dual-core processor, this processor is almost identical except for a few obvious changes. One such change is that the single-core processor does not have a snooping controller.

Est. Frequency	22.03 MHz

Avg. Instructions/Cycle	single.mergesort.asm: 0.479[1] single.search.asm: 0.316
Instruction Latency	1,089 ns (24 cycles)
Total FPGA Logic Blocks	6,897
Combinational	2,263
Register Only	4,634

Table 1.2: Pipelined Processor with Instruction and Data Cache Performance and Size Results

Dual-Core Processor

Our dual-core processor has two of the above processor cores with the data caches sharing a memory bus. The cache hierarchy is the same as the one implemented in the single-core processor with cache. The only difference is that the tag and status arrays are now dual-port arrays so that the new snoop controller can snoop into the arrays without stalling the processor.

One of the simplest coherence protocols, MSI, was used to ensure that coherence was followed. Each block in the cache has its own valid and dirty bits. Each of the three states in MSI (modified, shared, and invalid) can be encoded as a function of these two bits.

The coherence controller serves as the data cache's interface to the memory arbitrator. The coherence controller administrates bus operations (Rd, RdX, WB), provides a snooping bus into the opposite core, and has a state machine to control transfers to and from memory. To maintain an implementable design in the timeframe given, priority is always awarded to processor 0. The memory access times are similar for this new cache. A snoop that results in a cache-to-cache transfer only takes one cycle.

The memory arbitrator was moved out of the individual cores and placed between the core's instruction caches, the coherence controller, and memory. Priority is

[1] This result is very likely incorrect. The merge sort program was run and the processor seemed to be loading the data and computing the results correctly but they were not stored to RAM. Thus, there were about 200 less stores that the same program run on the multi-core processor.

given to the data port since the whole pipeline stalls on a data memory miss. If no data access is happening, priority is given to processor 0's instruction cache.

The critical path of the 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. Just like the single-core cached processor, this path is exactly what we expected. Again, the data cache is clocked on the falling edge, so this path could be cut down by half if we tweaked a few things to make it work on the rising edge.

The overall dual-core block diagram is shown in Figure A.1. The pipeline block diagrams are shown in Figures A.2 - A.4. The inside of the data cache block is shown in Figure A.5.

Est. Frequency	21.2 MHz
Avg. Instructions/Cycle	dual.mergesort.asm: 0.339 dual.search.asm: 0.461
Instruction Latency	991 ns (21 cycles)
Total FPGA Logic Blocks	19,406
Combinational	10,205
Register Only	9,201

Table 1.3: Dual-Core Processor Performance and Size Results

Conclusions

This semester, we implemented four different processor designs. We started with a single-cycle, pipelined it, added a cache, and finally just simply copied and pasted it (not really, of course). Our final design was a MIPS-based dual-core processor with instruction and data cache with MSI coherence and a snoopy bus. Although surely there are at least a few bugs left in our processor, it is able to pass all of the provided and self-written tests that we shoved in our computer's memory.

Our processor, though lacking in features, could perhaps be used as a cheap microcontroller if an I/O processor and interrupts were added. It would be very similar to some existing microcontrollers in the marketplace, such as Microchip's PIC32 (also MIPS based).

The only disappointment is that the cache is clocked on the falling edge, so our maximum frequency is about half as much as it could be. If we were to have a bit more time, we would have fixed that to improve the performance of our processor. Also, as development progressed, our core structural file became very messy, to the point where we both dreaded having to edit it. If we were to do it again, we would either break up that file more or come up with some sort of naming convention for the signals contained within.

Appendix

Figure A.1 – Dual-Core Processor Block Diagram

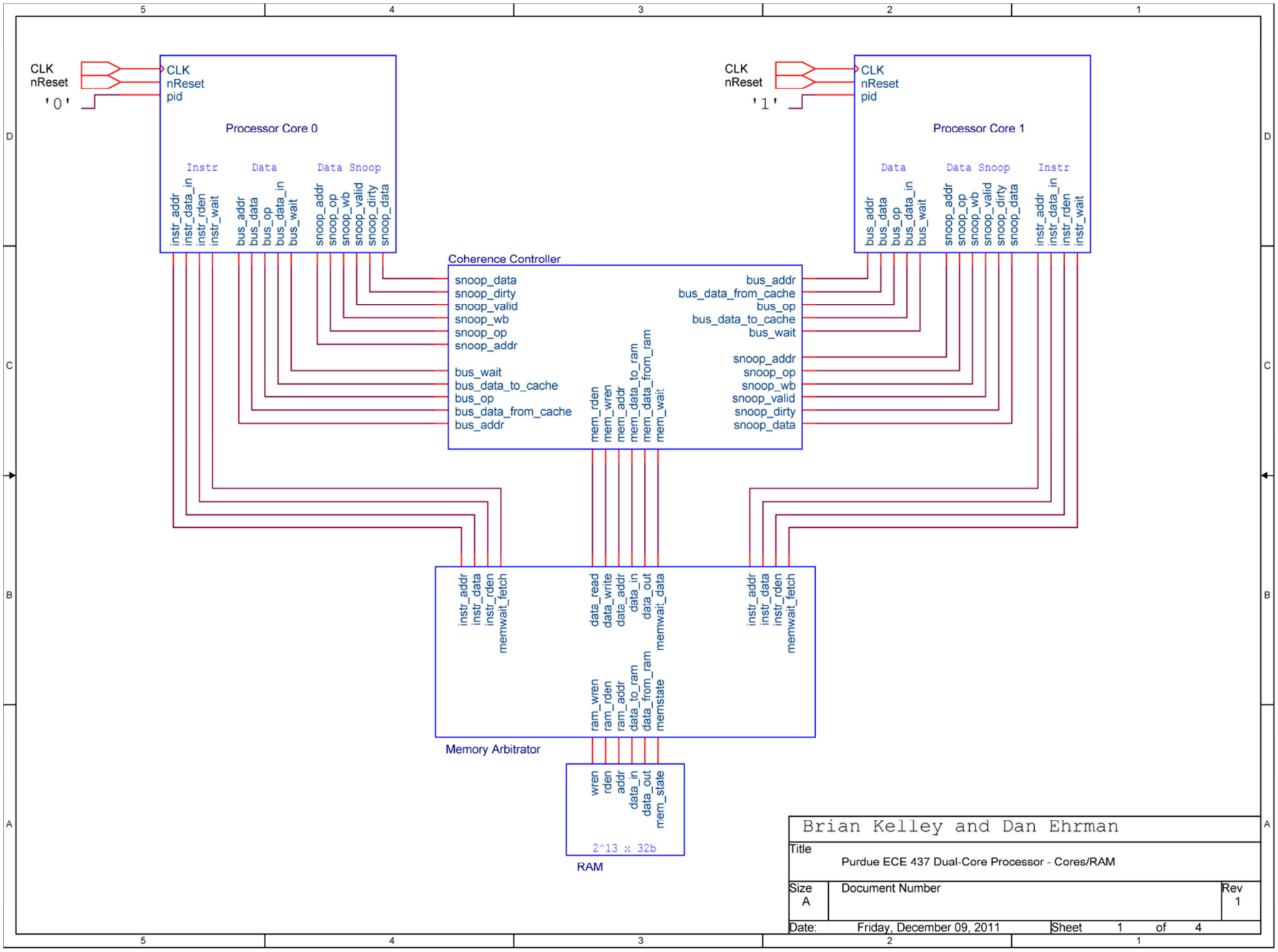

Figure A.2 – Instruction Fetch Stage

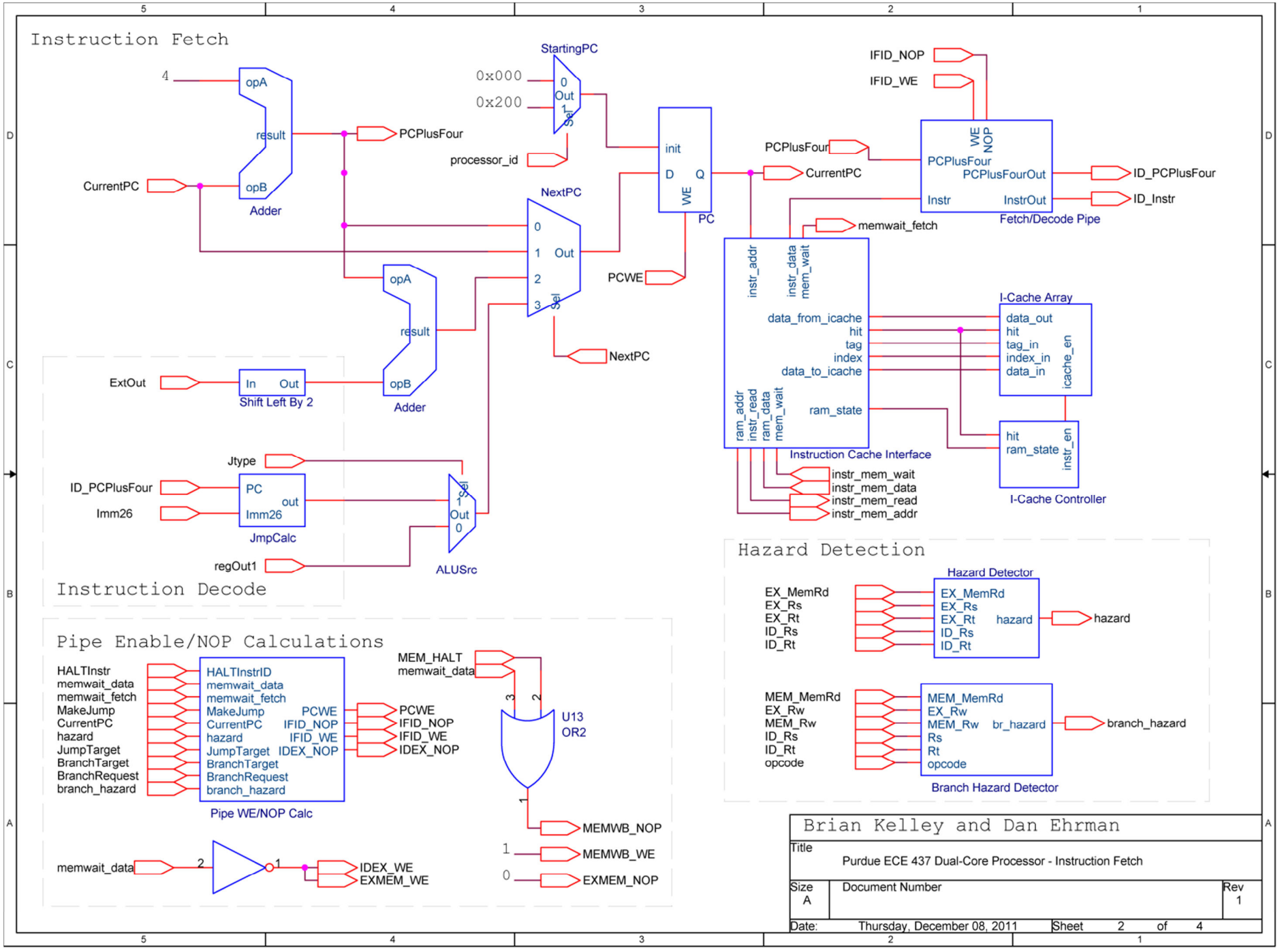

Figure A.3 – Instruction Decode Stage

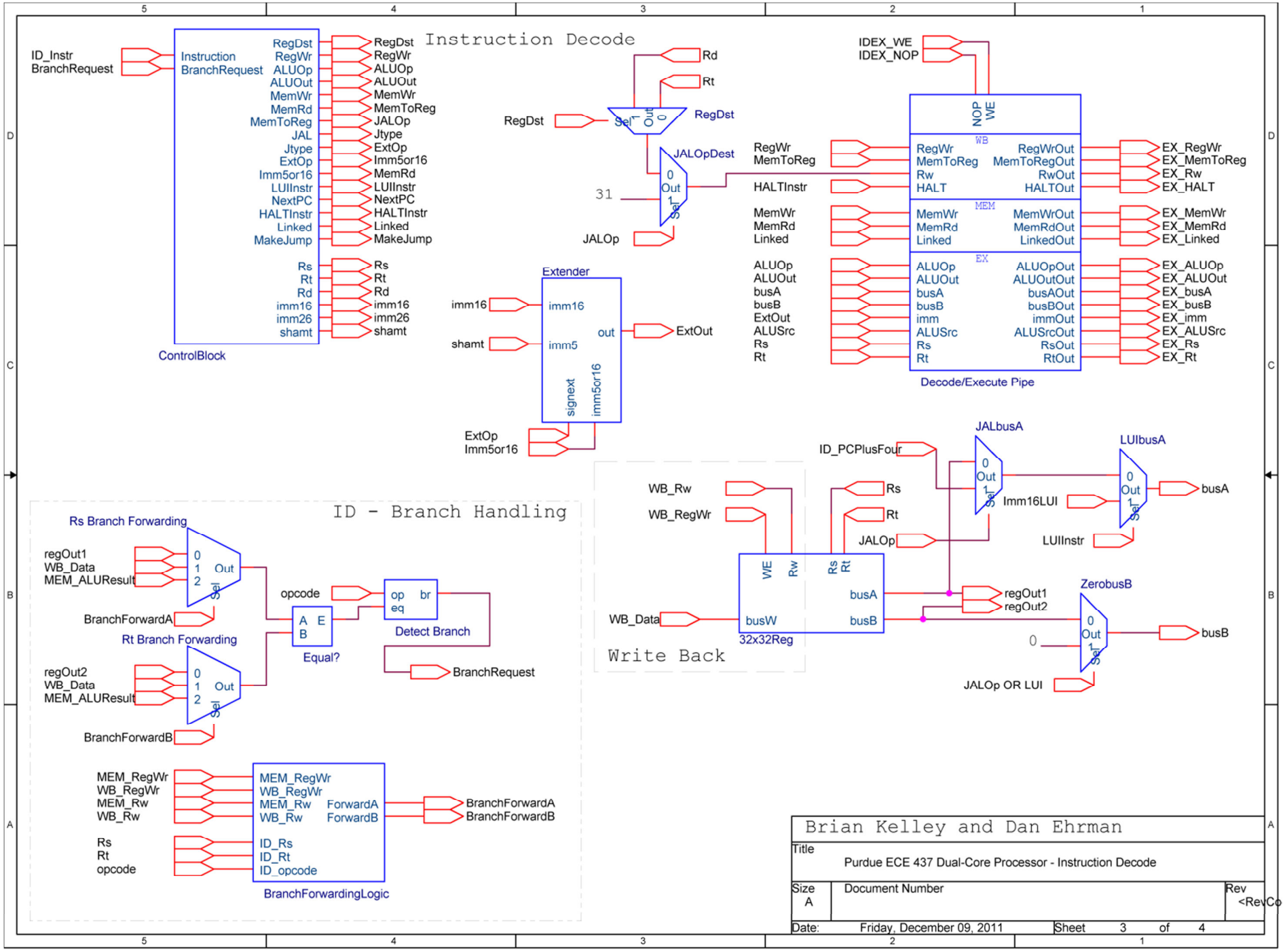

Figure A.4 – Execute, Memory, and Write Back Stages

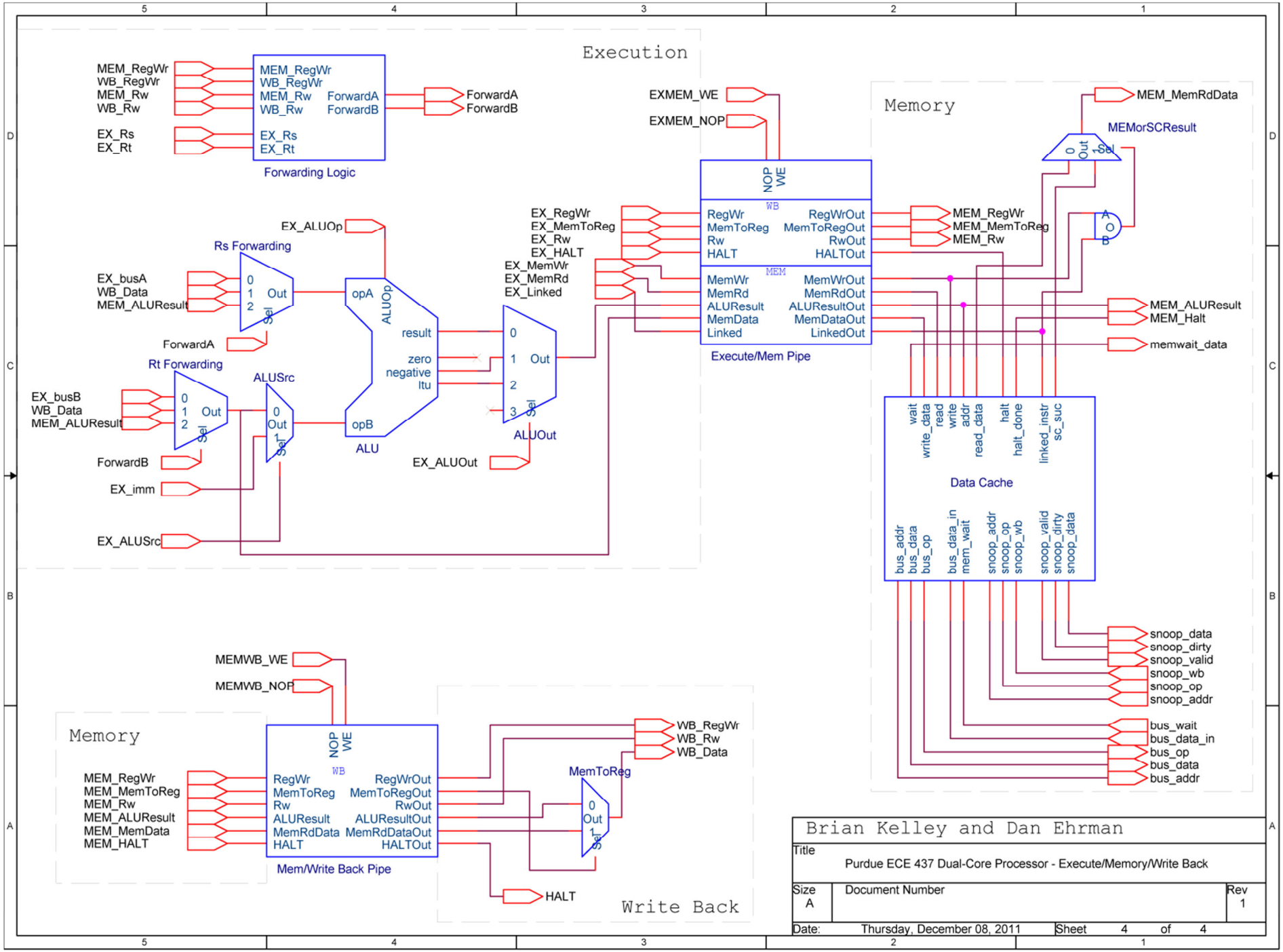

Figure A.5 – Data Cache Block Diagram

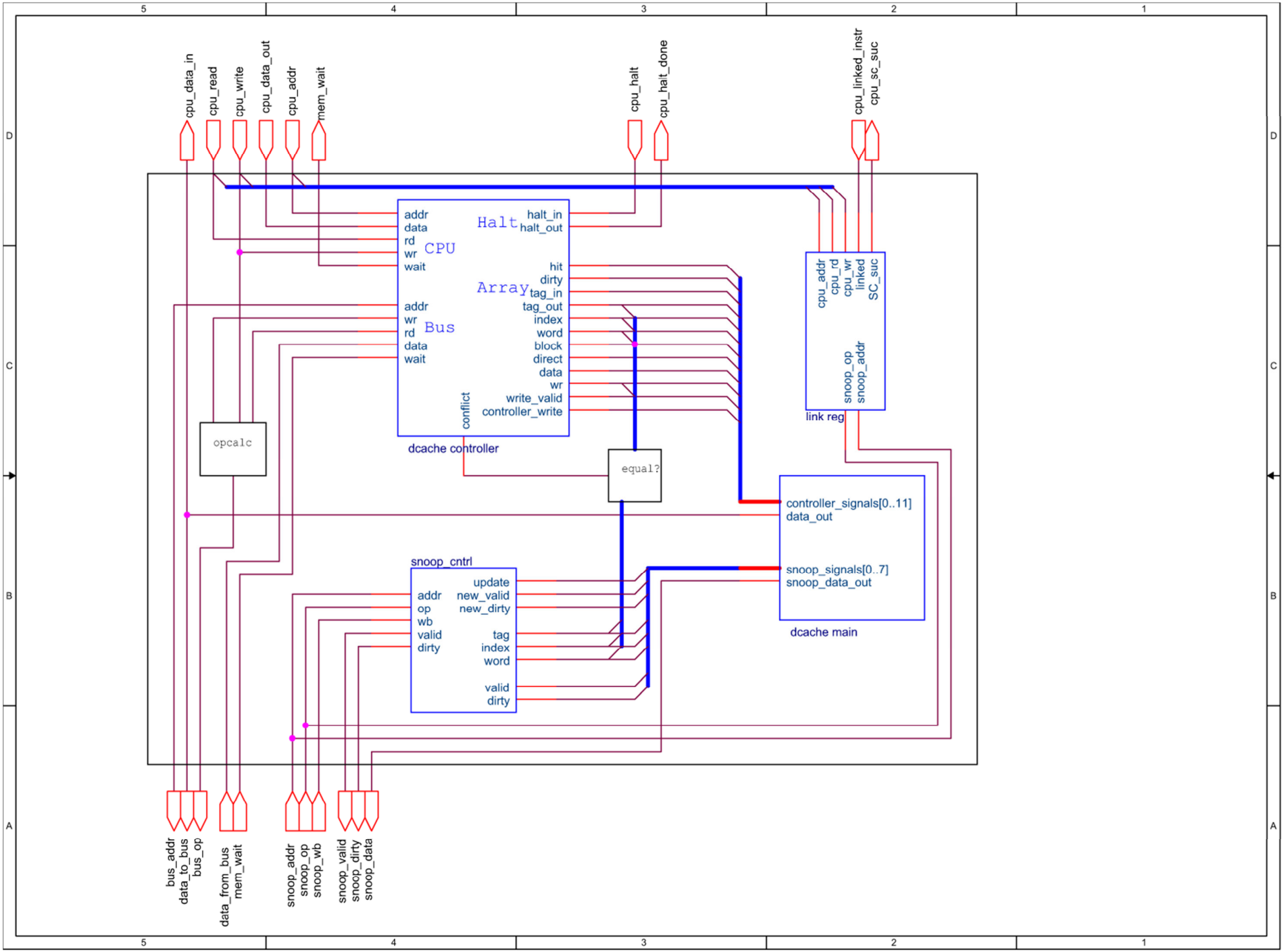

Figure A.6 – Data Cache Controller State Diagram

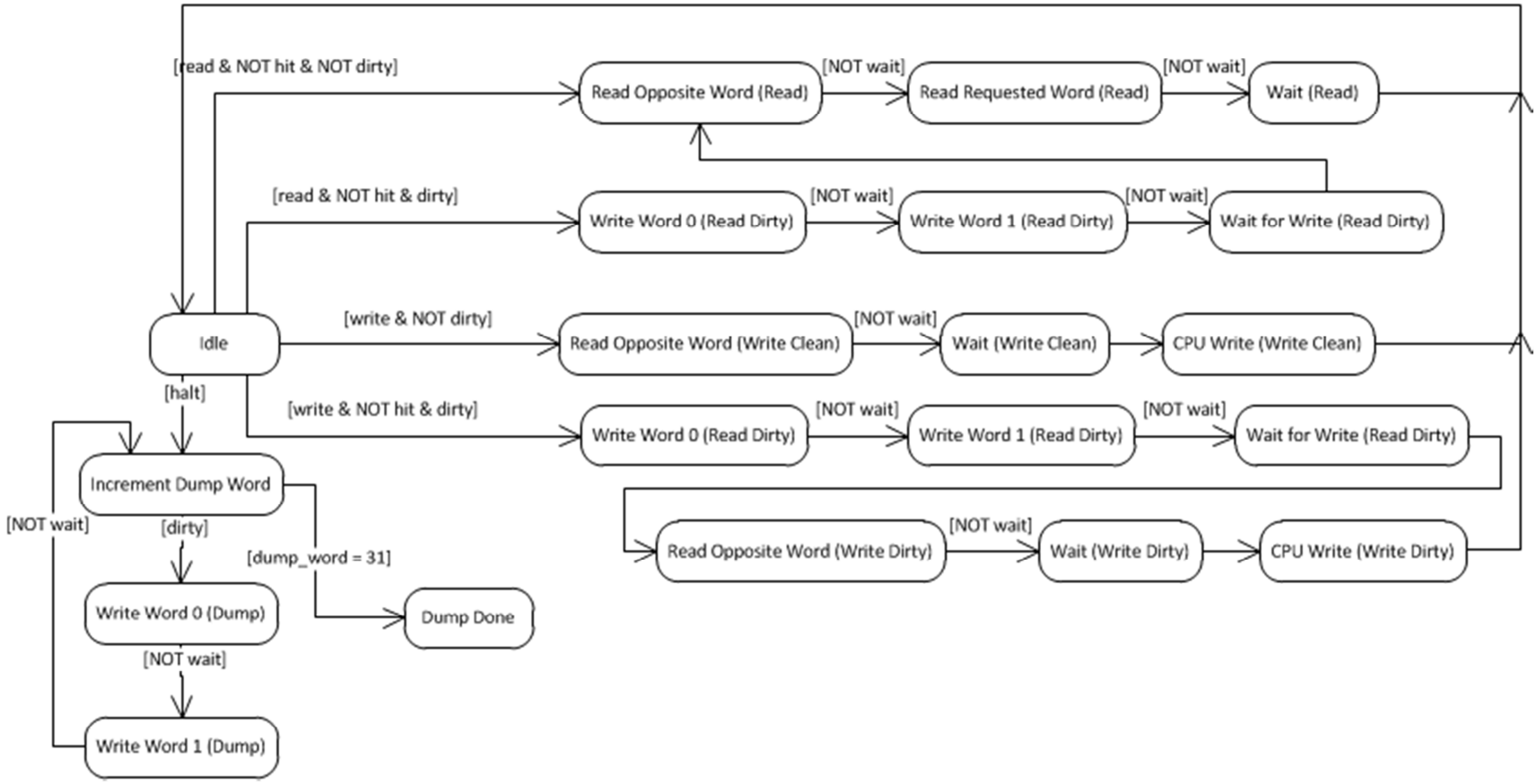

Figure A.7 – Coherence Controller State Diagram