Final Report

ECE 437

Team: Mitchel Bouma, Alex Dunker

TA: Chuan Tan

Due Date: December 9, 2016

Overview

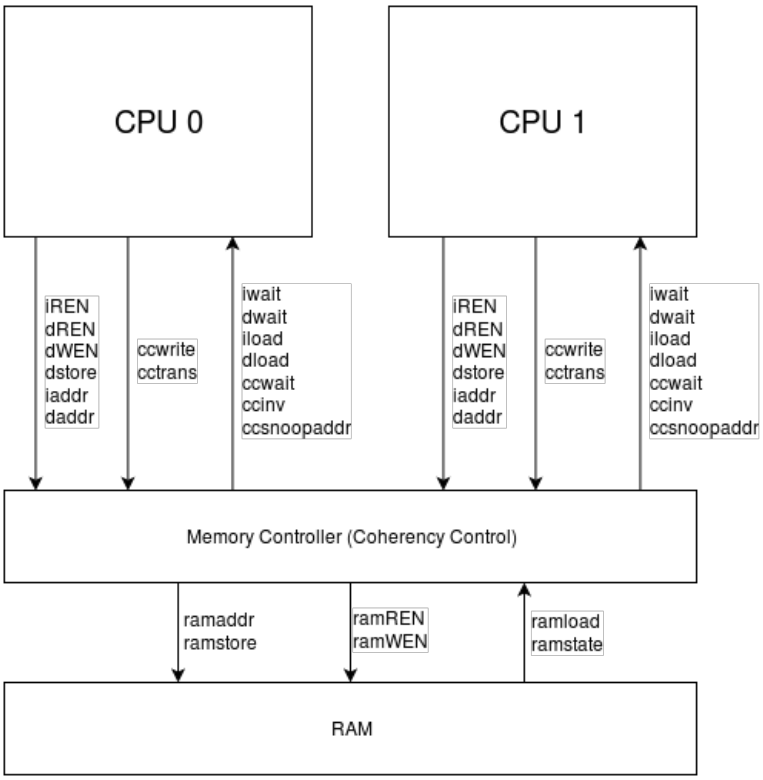
Our team designed and implemented a dual core processor and cache on top of our existing pipeline processor. The cache design implements a storage component that stores recently used data so it can be served to the processor faster. There are separate data caches and instruction caches. The instruction cache being a 512 bit direct mapped cache, and the data cache being a two way associative cache with 1 Kbits in size. These caches sit between the processor and RAM and are used to limit the number of times the processor is required to access RAM as this is a much more expensive operation than accessing data in the cache. The multi-core design allows the processor to take advantage of parallel programming. While this requires additional effort on the part of the program to take advantage of this, it allows each core to be executing different instructions concurrently. A coherency controller is used to prevent coherency issues, and the result is better CPI performance as it is leveraging both the cache and more instructions per cycle. In our multi-core implementation each core has its own cache and the MSI model is used to ensure coherency between caches, and allows each independent processor to be able to access and share data successfully. Both of the implementations however do require significantly higher resources and power requirements.

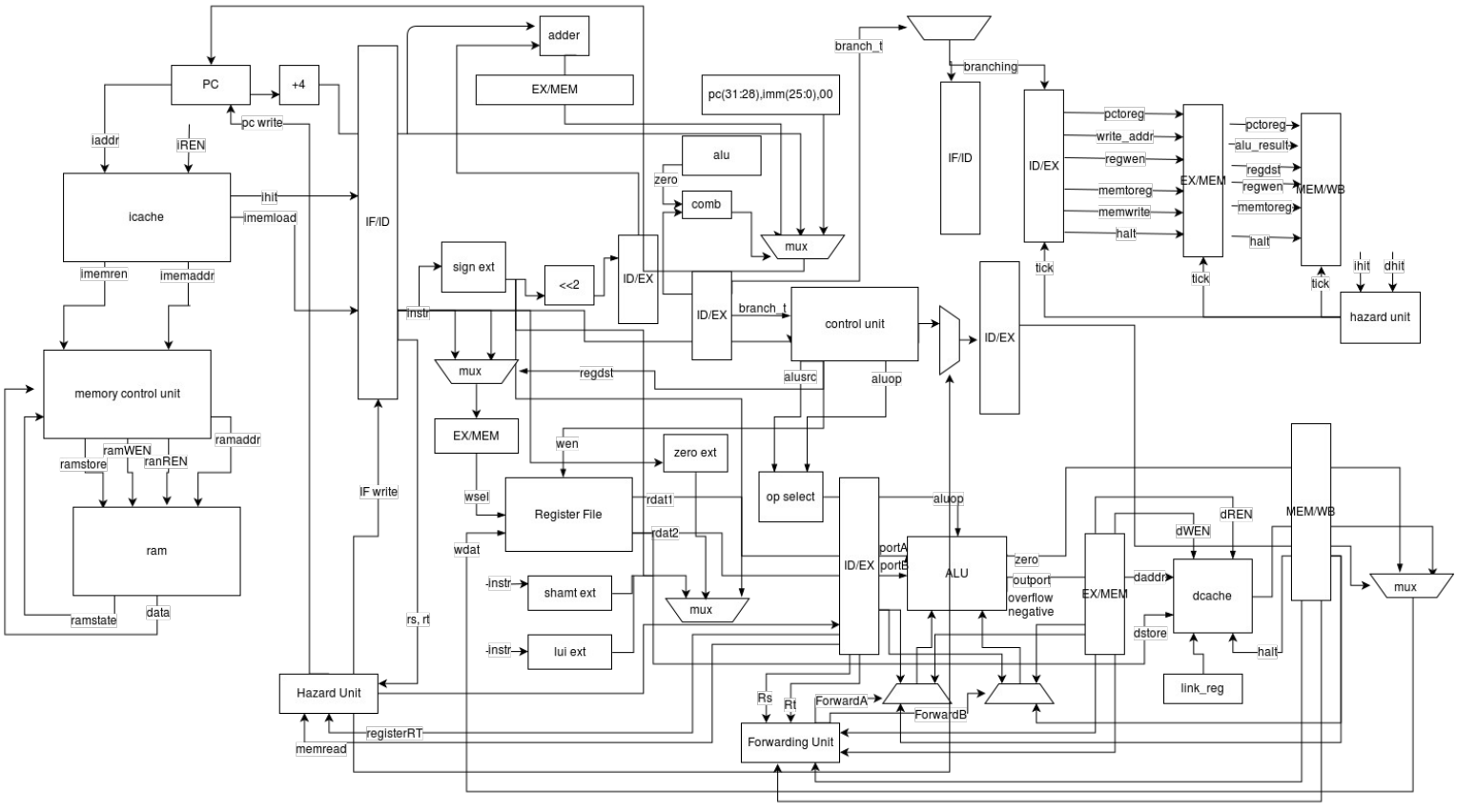
The test bench used to get metrics was mergesort.asm and dual.mergesort.asm. Mergesort has many branches, r-type instructions, and jumps, so it therefore tests the main functionality of the processor well. It is also very complex and has a relatively long execution time, so the metrics will be distinguishable between the designs. The main metrics used when comparing the designs are max frequency, mips (millions of instructions per second), and run time. However due to restrictions based on our multicore processor the algorithm will be run to sort 16 values. The comparisons made will bet between a pipelined processor, a pipelined processor with caches, and a dual core processor with caches.

The rest of our report will include the block diagrams for our processor, cache, and coherence control as well as corresponding state diagrams, the results from the mapped versions of our designs and the conclusions we have taken from the results.

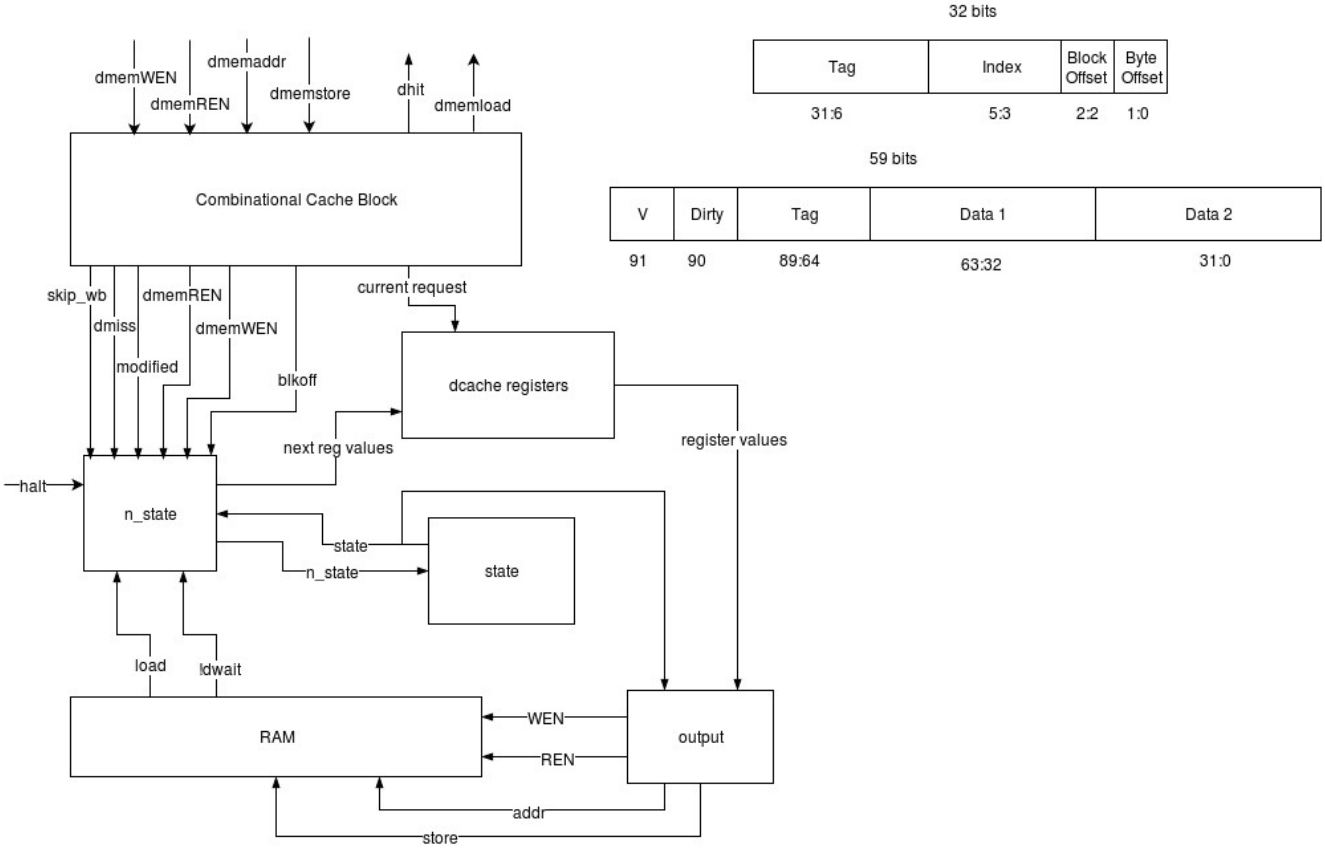
Processor Design

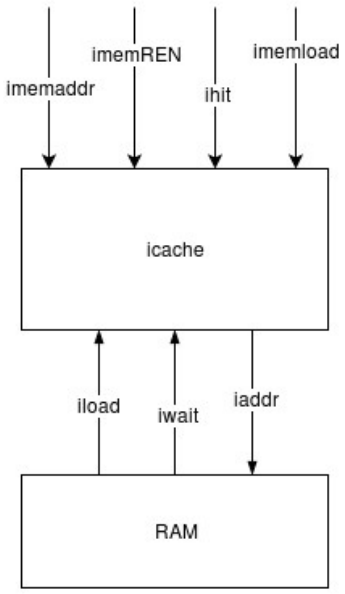
Processor Design

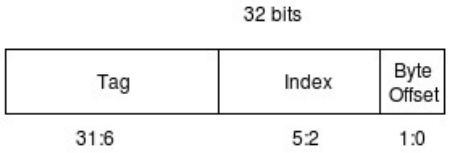


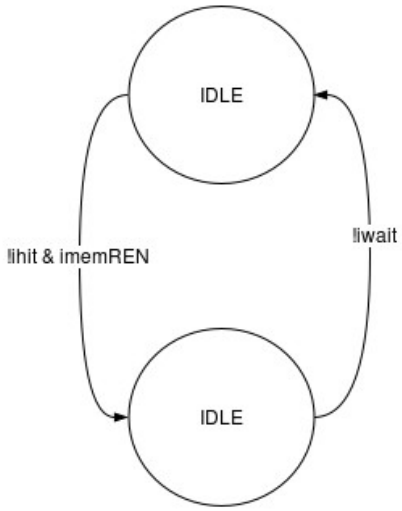


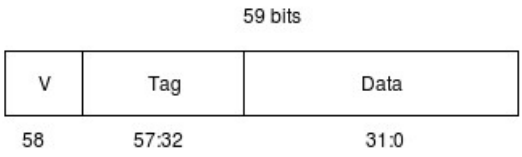
Cache Design



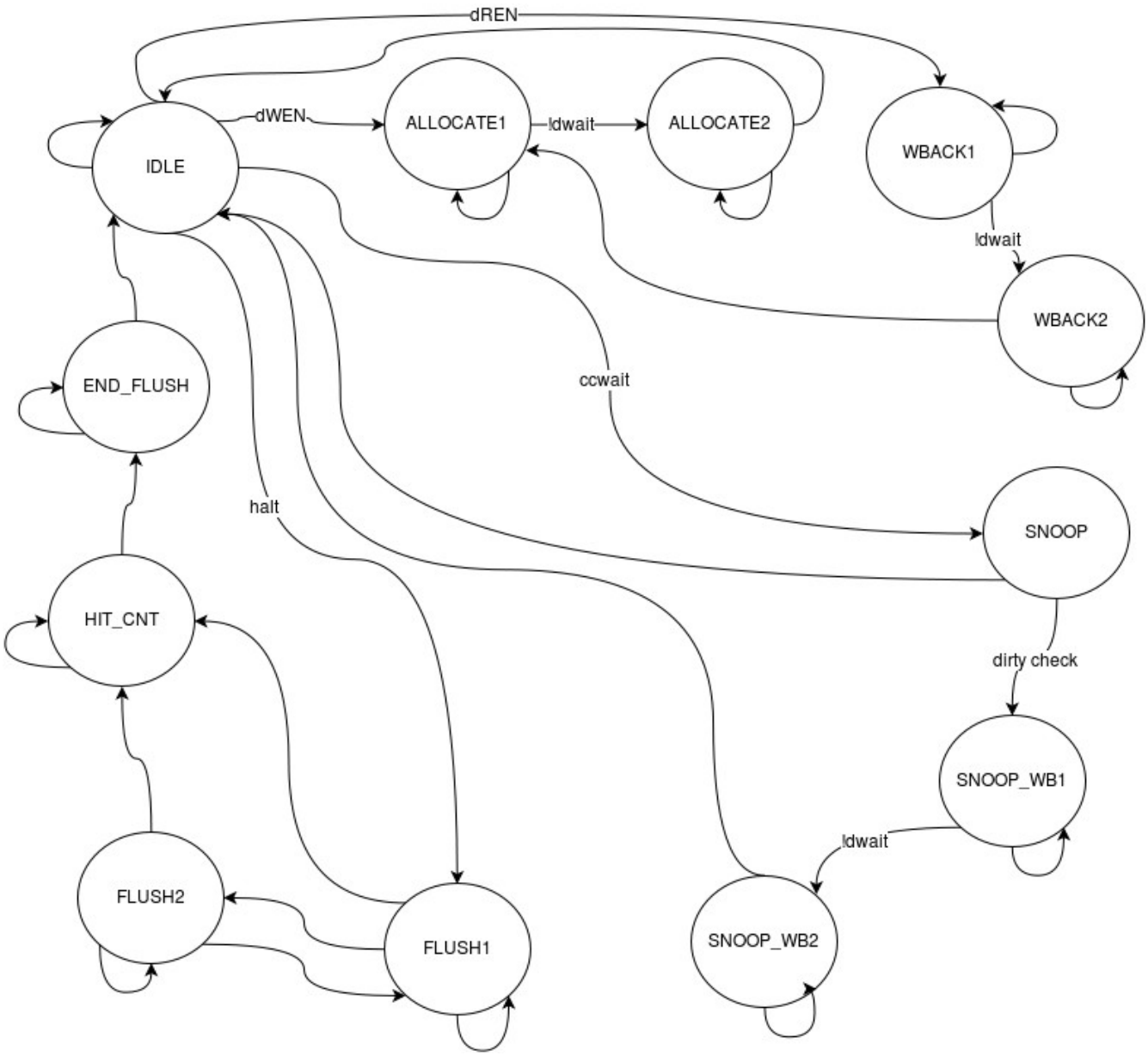

imemaddr
imemREN
ihit
imemload
icache
iload
iwait
iaddr
RAM


32 bits
Tag
Index
Byte Offset
31:6
5:2
1:0

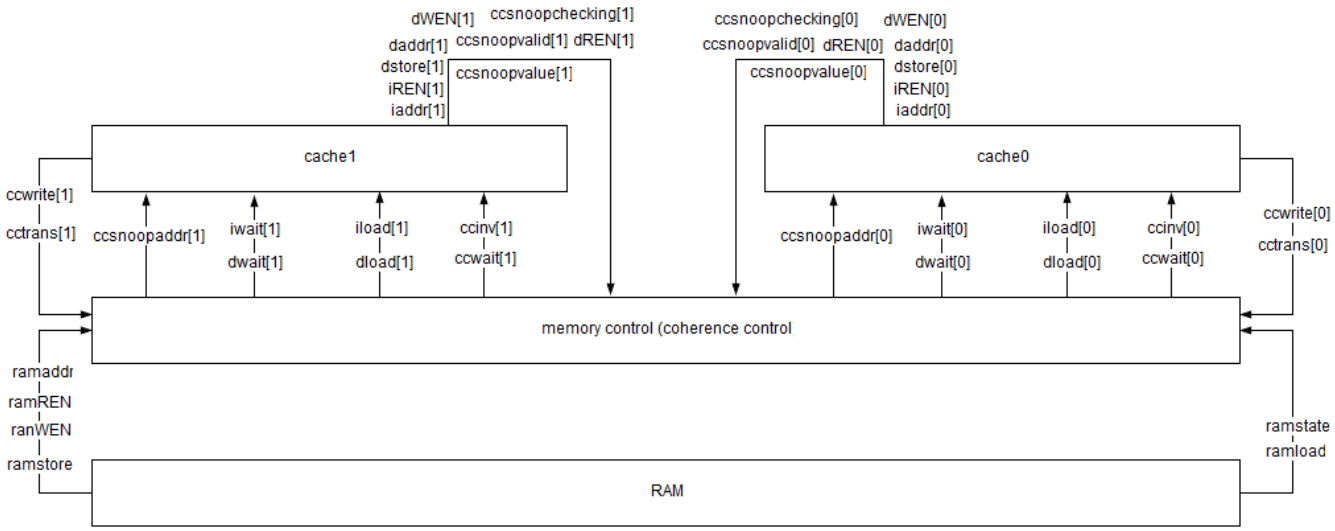

IDLE
!ihit & imemREN
!iwait
IDLE


59 bits
V
Tag
Data
58
57:32
31:0

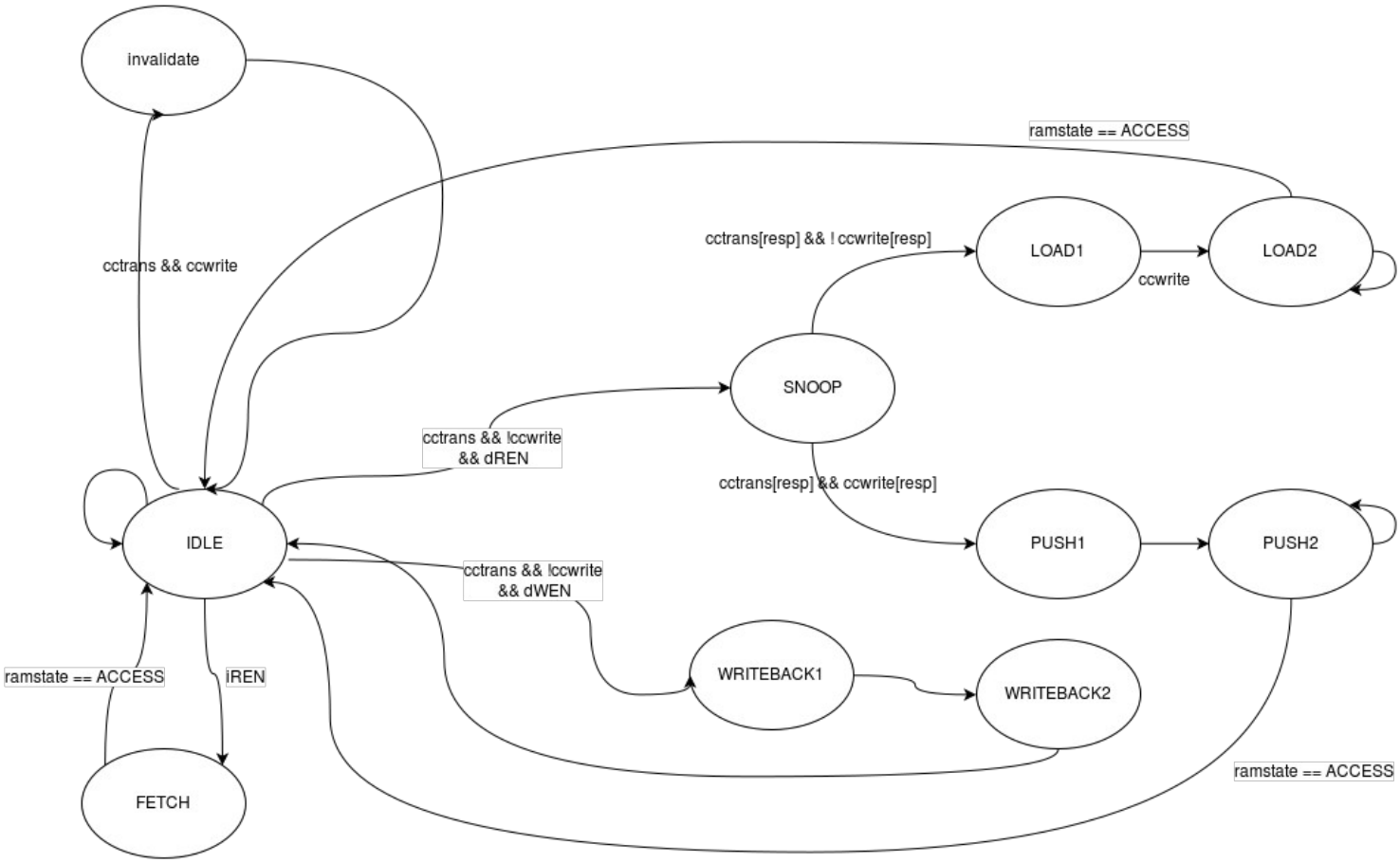
Cache State Diagram



Coherence Control



Memory Control State Diagram



Results

Test Case: mergesort.asm & dual.mergesort.asm

	Pipeline	Pipeline w/ Caches	Multicore w/ Caches	Source
Max Clock Frequency	54.45 MHz	43.58 MHz	39.95 MHz	From log
Avg Instructions/Cycle	0.08	0.28	0.27	Instructions/cycles
Latency	91.8 ns	114.7 ns	166.9 ns	5*(1/frequency)
FPGA Resources	1629	4145	8125	Registers in log
Run Time	18 ns	23 ns	33 ns	instructions*(1/ipc) *((1/frequency)/cycles)
Cycles	7800	2145	2337	From system.sim
Instructions	610	610	625	From sim-m (core1+core2)

Conclusion

Our results show a great increase in the number of instructions per cycle with the implementation of the caches, which shows the amount of performance that can be given when it is not necessary to constantly access memory. While the clock frequency was decreasing, we can attribute this to the increasing complexity of our design, and time spent towards optimizations would help increase this. The multicore processor does not show any increase with terms of performance, but with larger and larger workloads the increased performance would become more apparent, as the number of instructions executed is still a rather small amount. The versions with the caches use almost a fourth of the cycles than the processor without the cache. These two processors do however use quite a bit more resources, which would be expected based on the extra items needed to implement each.

Contributions

Mitch can be credited with designing and creating the block diagrams for the caches as well as the data cache state diagram. Mitch also wrote the code for the data cache and did the majority of the debugging on it throughout the process. He also integrated the code needed to implement LL and SC instructions into the data cache.

Alex focus on the caches was creating the instruction cache diagram and code, as well as debugging it. He designed and wrote the coherency control inside of the memory controller. This included creating state diagrams as well as writing and debugging the code. He also assisted in the debugging of the data cache, but took a secondary role to Mitch. Alex wrote and implemented the LL SC instructions in the data-path and control unit as well as the parallel algorithm.