Final Report

ECE437 Computer Design and Prototyping

Matthew Hill and Austin Ketterer

Lab Section 3

GTA Nikkitha Subbiah

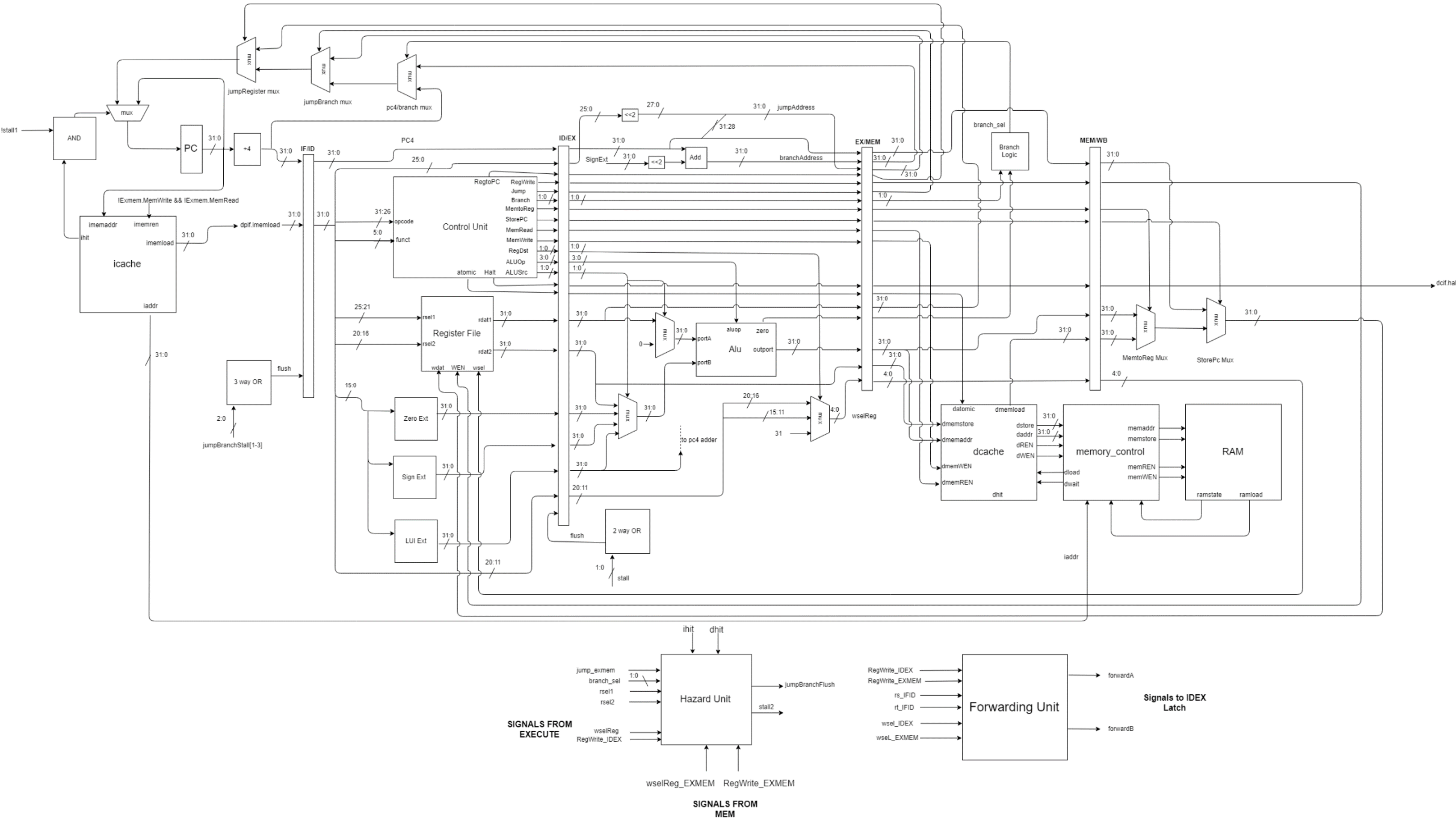
12/9/2018

Overview

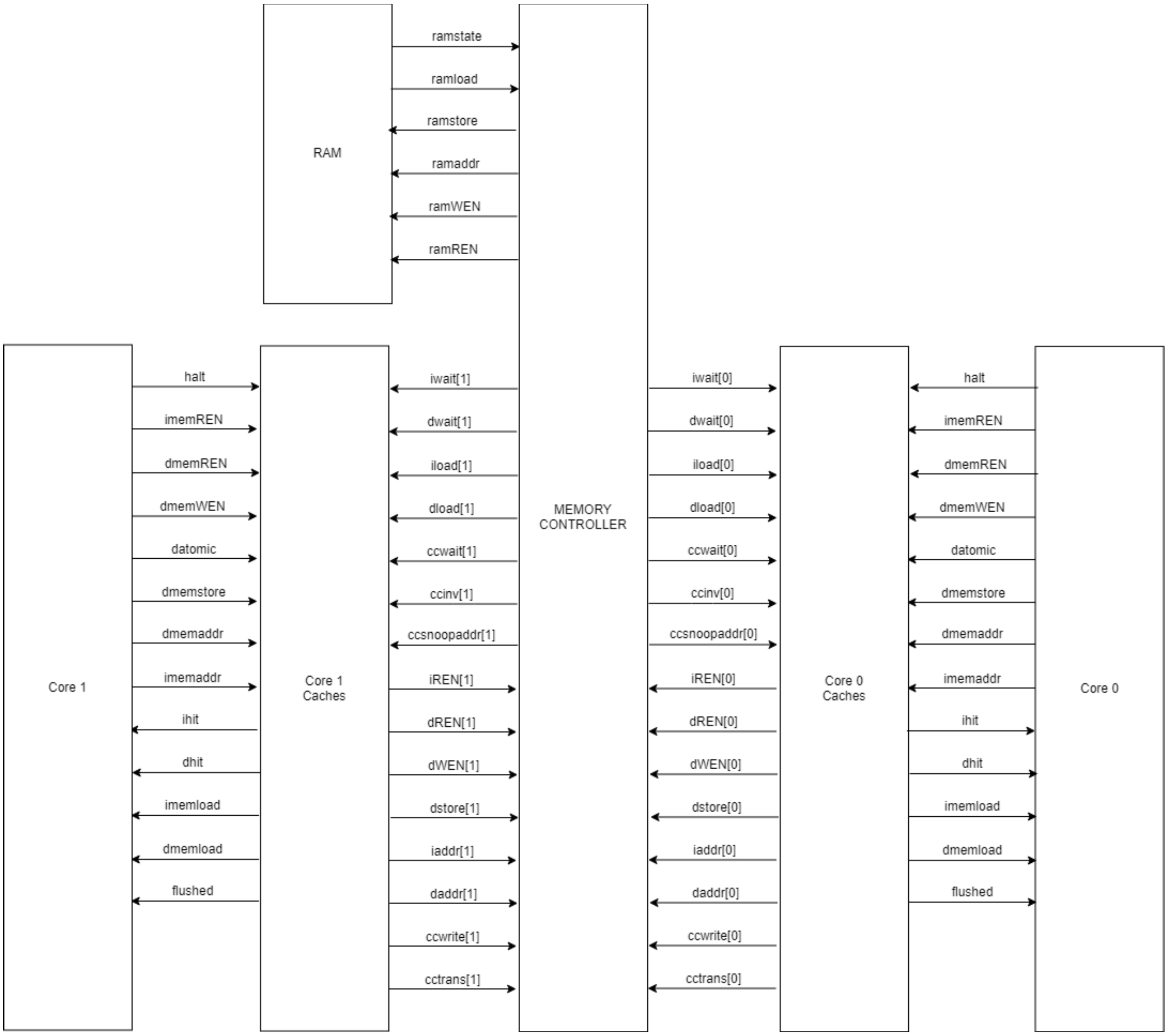
Five processor designs are being compared: a single cycle processor, a pipelined processor, a pipelined processor with caching, a multicore processor running a sequential program, and a multicore processor running a dual threaded program. The single cycle processor and pipelined processor were beneficial learning tools since their designs were simplistic compared to the later implementations (and both perform better than the other processors with 0 latency, but this is unrealistic). These processors also utilize less resources and have a smaller area budget compared to the later processors. They have a large downside however in that their performance suffers greatly with increasing RAM latency. The pipeline processor with caching has higher performance at the cost of more area for the caches. Less accesses to main memory allows it to run much faster. The multicore processor should have similar performance for a sequential program compared to the pipeline processor with caching, and better performance when running a parallel program at the cost of a much higher area.

Data was gathered from the synthesis log files of each processor and the execution of Merge Sort at varying latencies. The metrics being compared are number of FPGA registers and combinational logic gates, critical path time, execution time, number of cycles, CPI, and instruction latency. The data should validate that single cycle and pipeline without caches use less area than the pipeline with caches and multicore processor but have lower performance at higher latencies. The design for the multicore processor, caches, and memory controller are shown followed by the results and analysis of the data.

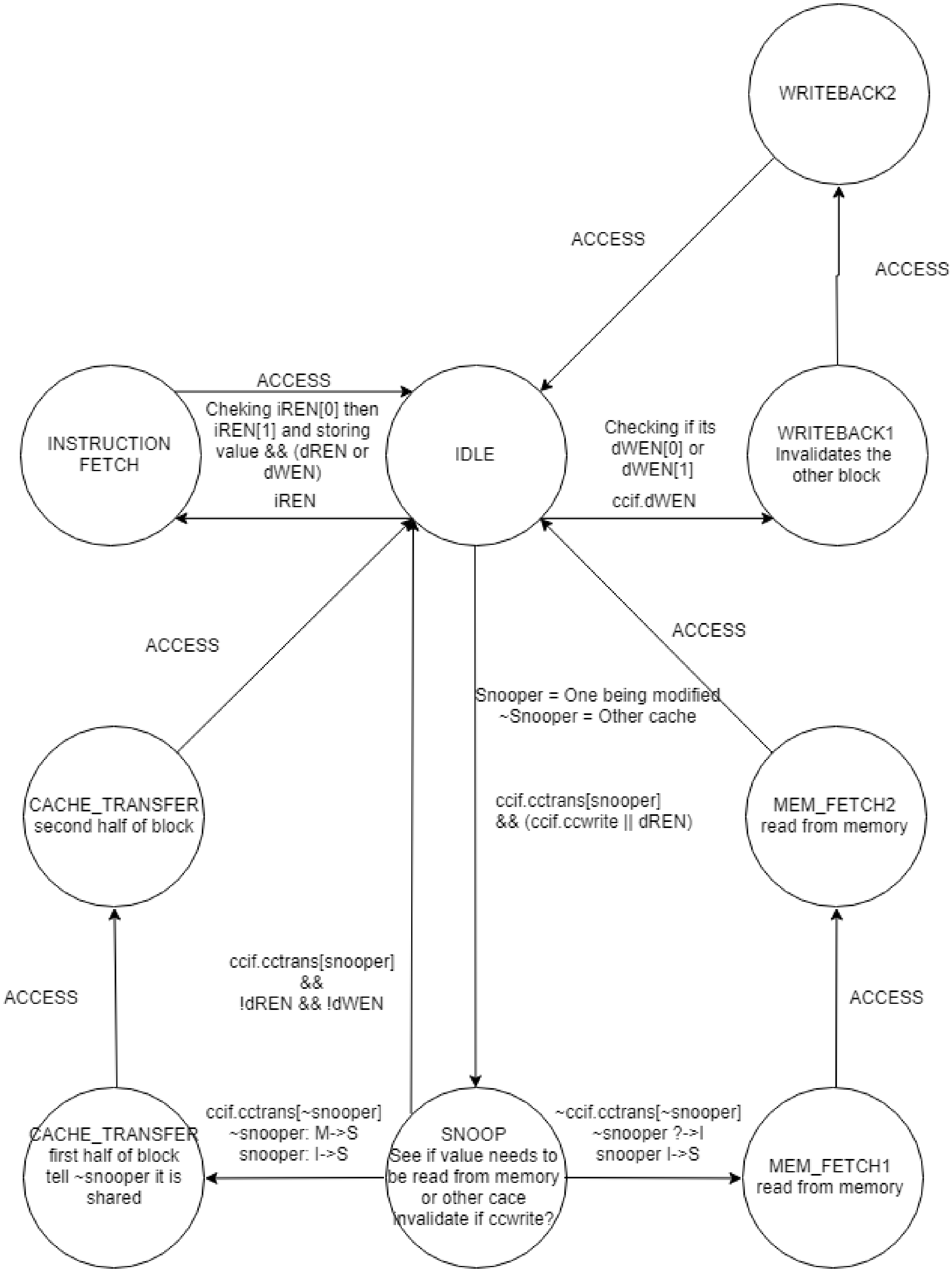
Processor Datapath with atomic signal added to control unit for LL/SC



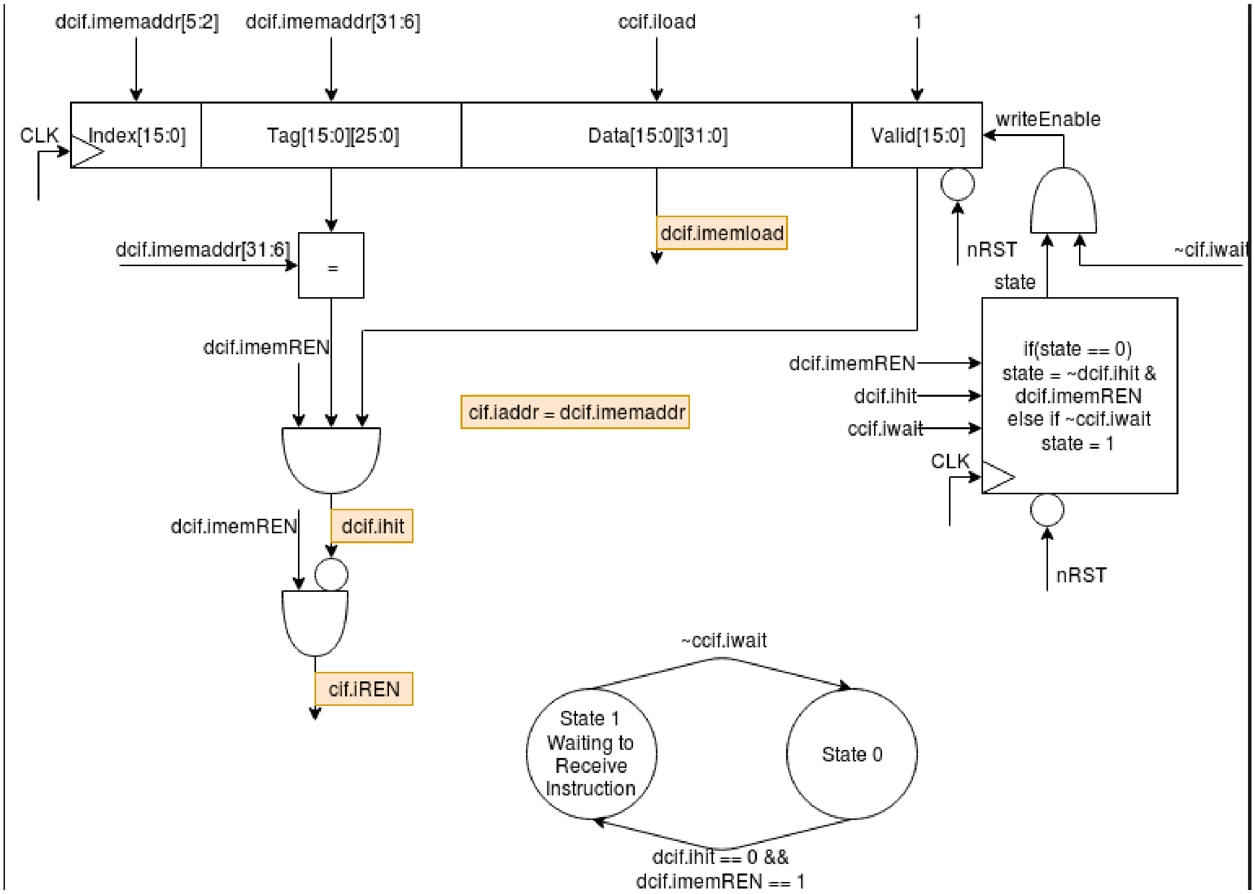
CPU Block Diagram with Memory Controller



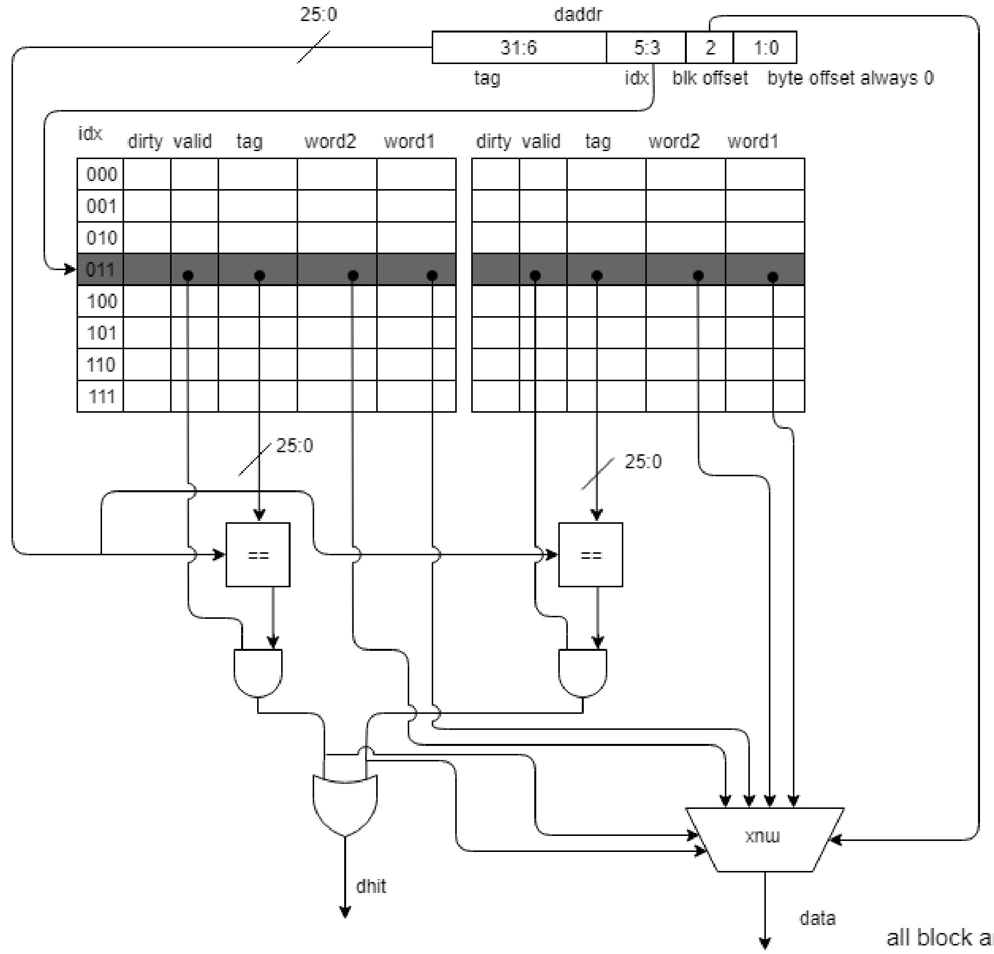
Memory Controller Coherence State Diagram


WRITEBACK2
ACCESS
ACCESS
ACCESS
INSTRUCTION FETCH
Cheking iREN[0] then iREN[1] and storing value && (dREN or dWEN)
iREN
IDLE
Checking if its dWEN[0] or dWEN[1]
ccif.dWEN
WRITEBACK1 Invalidates the other block
ACCESS
ACCESS
Snooper = One being modified
~Snooper = Other cache
ccif.cctrans[snooper] && (ccif.ccwrite || dREN)
CACHE_TRANSFER second half of block
MEM_FETCH2 read from memory
ccif.cctrans[snooper] && !dREN && !dWEN
ACCESS
ACCESS
ccif.cctrans[~snooper] ~snooper: M->S snooper: I->S
~ccif.cctrans[~snooper] ~snooper ?->I snooper I->S
CACHE_TRANSFER first half of block tell ~snooper it is shared
SNOOP See if value needs to be read from memory or other cace invalidate if ccwrite?
MEM_FETCH1 read from memory

Icache Design

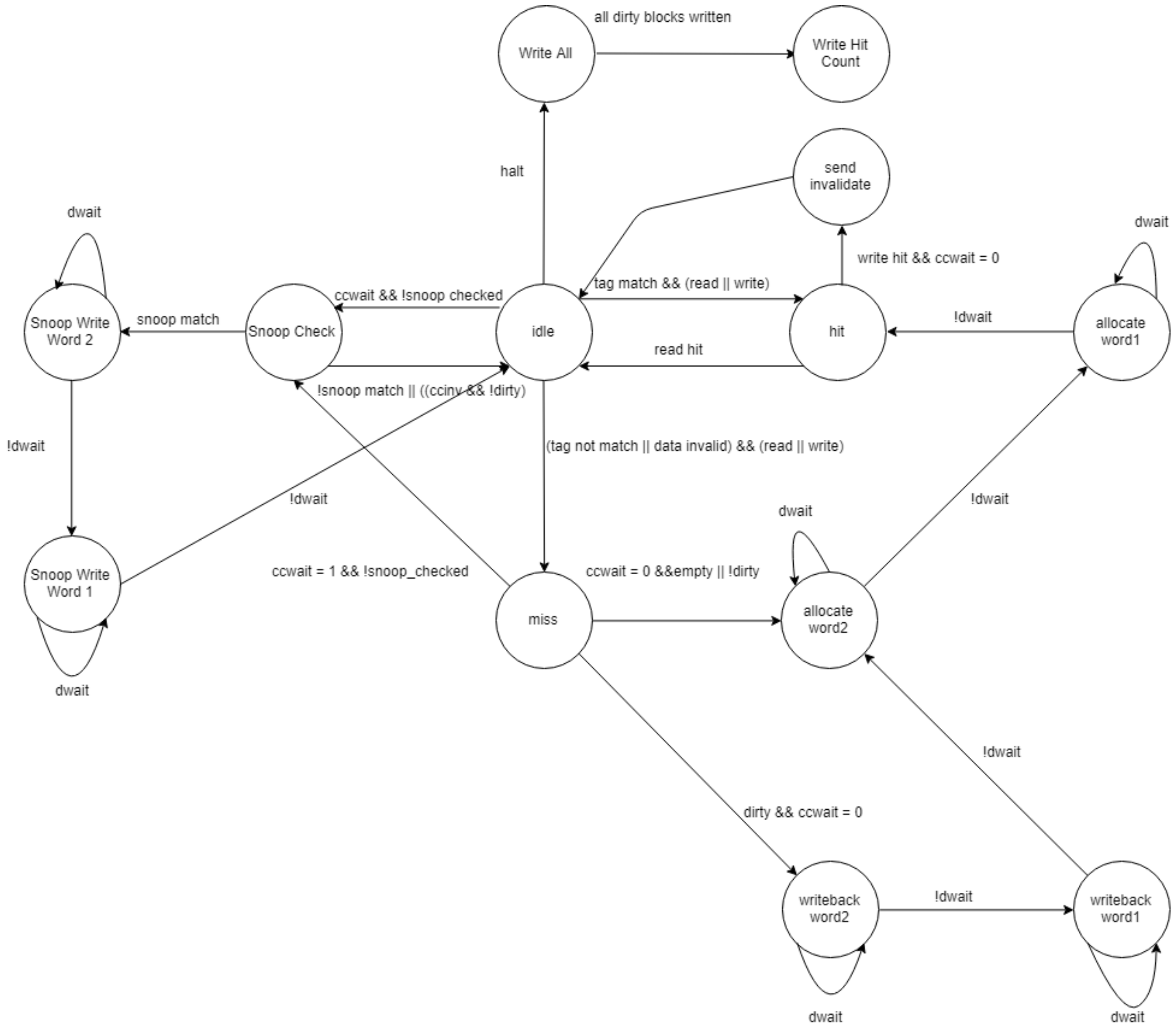


Dcache Design

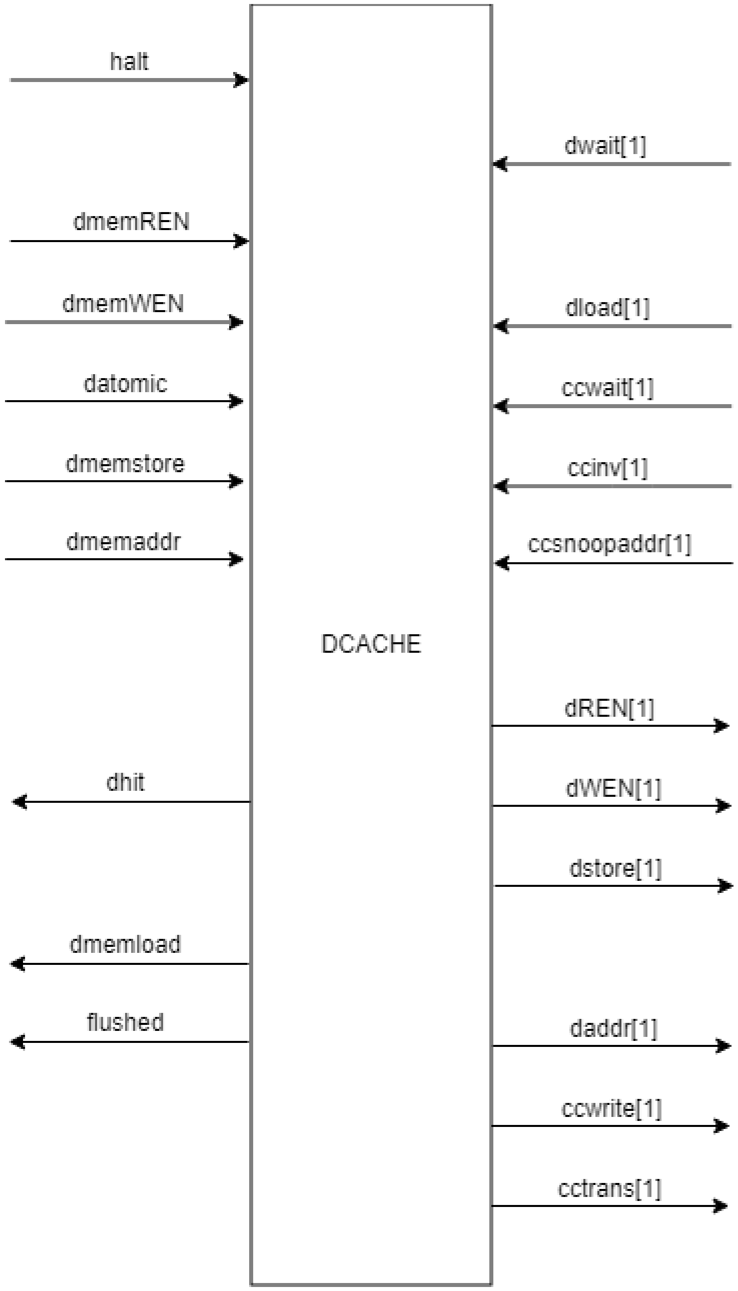


all block and data lines are 32 bit

DCache State Diagram



DCache Block Diagram



Results

Test Program Results at 6 latency

	Synthesis Frequency (MHz)	Critical Path (ns)	Average Single Instr Latency	FPGA Comb.	FPGA Registers
Single-Cycle	37.07	33.8	275.53	2863	1284
Pipe w/o Cache	68.61	16.0	217.98	3616	1839
Pipe w/ Cache	56.19	24.57	104.92	6965	4248
Multi w/ Cache Single-Threaded	32.58	37.62	201.70	14148	8331
Multi w/ Cache Dual-Threaded	32.58	37.62	138.23	14148	8331

Synthesis Frequency, Critical Path, and FPGA Resources

- All directly obtained from the synthesis log files.

Instruction Latency

- Calculated by dividing the number of instructions in mergesort by the execution time.

Speedup Compared to Sequential

- Calculated by comparing the execution time of sequential to dual threaded execution for the multicore processor.

Frequency of the Designs

The single cycle frequency serves as the baseline for all the other designs. By adding registers between each of the major sections of the design we were able to increase performance and increase the clock frequency in the pipelined design. Our design for adding caches affected the clock cycle but not by much, the design should largely run the same. The multicore frequency suffered due to the input signals not being latched into the snoopy bus which limited its overall performance.

Average Instructions per Clock Cycle w/ Caches and w/o

Pipeline w/o Caches @ Latency 6: 5404 Instructions / 80851 Cycles = 0.069 I/C

Pipeline w/ Caches @ Latency 6: 5404 Instructions / 31901 Cycles = 0.169 I/C

Demonstrated by comparing the two pipelines, the designs with caches were able to accomplish many more instructions per cycle. The reason behind this is that less time was wasted on interacting with the main memory. The designs seem to be more efficient with zero latency, but this situation does not exist and as soon as latency comes into the picture the cache designs fare far better.

The Latency of the Instructions

The pipeline had a lower latency due to the majorly improved clock speed. Then going to caches the average latency of each instruction was improved because the time taken for each of the memory related instructions was drastically reduced. Lastly in the multicore, even with a much lower frequency, the latency did not rise much because around twice any many instructions could be executed at the same time. If the bus clock were fixed this latency would have been much lower than the others designs.

FPGA Resources Required

From single cycle to pipelined little resources were required, and a large performance gain was obtained, making the upgrade very worthwhile. Adding caches added a lot of registers which had to store the values so that we could avoid using memory. This takes up a lot of space but is essential for a design with any amount of latency between the CPU and memory, which would be any realistic design. Lastly having two cores doubled the amount of resources required because we essentially doubled the whole design. If we had fixed our multicore frequency this upgrade would be worth it due to the lower CPI and number of cycles taken, if they user could make good use of both cores with their programs.

Speedup from Sequential to Parallel Program

Execution of Double Thread / Execution of the Single Thread = 138.23 /201.7 = 0.69

Using two cores reduced the original runtime of the program to about seven-tenths of what it originally was. Ideally the runtime would be half but this needs to account for all the time that one of the cores is waiting on the other or finishes before the other, which wastes time with one of the cores doing nothing.

Performance Sweep Table using mergesort.asm (and dual.mergesort.asm when applicable)

Name - Latency	Execution Time	Number of Cycles	CPI	Frequency (MHz)	MIPS	Latency (ns)
Single Cycle - 0	0.372	13801	2.55	37.07	14.53	68.83
Single Cycle - 1	0.744	27603	5.1	37.07	7.26	137.67
Single Cycle - 2	0.744	27605	5.1	37.07	7.26	137.67
Single Cycle - 3	1.116	41405	7.66	37.07	4.83	206.51
Single Cycle - 4	1.116	41407	7.66	37.07	4.83	206.51
Single Cycle - 5	1.489	55207	10.21	37.07	3.63	275.53
Pipe w/o C - 0	0.296	20309	3.75	68.61	18.29	54.77
Pipe w/o C - 1	0.59	40489	7.49	68.61	9.16	109.17
Pipe w/o C - 2	0.59	40491	7.49	68.61	9.16	109.17
Pipe w/o C - 3	0.884	60669	11.22	68.61	6.11	163.58
Pipe w/o C - 4	0.884	60671	11.22	68.61	6.11	163.58
Pipe w/o C - 5	1.178	80849	14.96	68.61	4.58	217.98
Pipe w/ C - 0	0.479	26971	4.99	56.19	11.26	88.63
Pipe w/ C - 1	0.493	27711	5.12	56.19	10.97	91.22
Pipe w/ C - 2	0.493	27713	5.12	56.19	10.97	91.22
Pipe w/ C - 3	0.53	29805	5.51	56.19	10.19	98.07
Pipe w/ C - 4	0.53	29807	5.51	56.19	10.19	98.07
Pipe w/ C - 5	0.567	31899	5.9	56.19	9.52	104.92
MC w/ C Sgl - 0	0.902	29411	5.44	32.58	5.98	166.91
MC w/ C Sgl - 1	0.965	31455	5.82	32.58	5.59	178.57
MC w/ C Sgl - 2	0.965	31455	5.82	32.58	5.59	178.57
MC w/ C Sgl - 3	1.026	33449	6.18	32.58	5.27	189.85
MC w/ C Sgl - 4	1.028	33499	6.19	32.58	5.26	190.22

MC w/ C Sgl - 5	1.09	35543	6.57	32.58	4.95	201.7
MC w/ C Dbl - 0	0.579	18889	3.49	32.58	9.33	107.14
MC w/ C Dbl - 1	0.641	20909	3.86	32.58	8.44	118.61
MC w/ C Dbl - 2	0.641	20909	3.86	32.58	8.44	118.61
MC w/ C Dbl - 3	0.695	22669	4.19	32.58	7.77	128.6
MC w/ C Dbl - 4	0.695	22669	4.19	32.58	7.77	128.6
MC w/ C Dbl -5	0.747	24359	4.5	32.58	7.24	138.23

Conclusions

At higher memory latencies the pipelined processor with caches out performed the single cycle processor and the pipeline processor without caches, observable by its execution time being faster for all executions of mergesort that included latency. Having to access memory is a major setback, so any of the cache designs are a large improvement for any realistic amount of latency. The execution time for the multicore processor was faster at higher latencies than the single cycle and cacheless pipelined processor as well. However, since the snoopy bus clock was not fixed in the multicore implementation, the max frequency was inadequate and even worse than single cycle. If this issue was fixed, it is expected that the multicore would outperform the pipeline with caches when executing the dual threaded program since its CPI and number of cycles needed to execute are actually lower than the Pipeline with caches.

The area for both the pipeline with caches and multicore are both higher than the single cycle and pipeline with no caches. This is to be expected since these designs need area to implement the caches. While being double the size, the fixed multicore design would make up for it with a moderate increase to performance. It would be worthwhile to have if the user needed the extra performance and there was still room on the chip for having two cores. It is important with the two core model however to use programs which are able to make use of multiple cores. Running a single threaded program through the multicore design was slightly slower than if it had just been the original pipeline with caches due to the bus protocols added.

Contributions

Matthew Hill

- Wrote the testbench for the bus controller
- Assisted with the state diagram for the bus controller
- Completed the D-Cache block diagram and D-Cache Controller state transition diagram
- Updated the code for the D-Cache for multicore
- Added the atomic signal, implemented LL and SC, and updated all relevant parts
- Wrote overview and part of conclusion
- Worked on results section
- Added diagrams to design section

Austin Ketterer

- Wrote the code for the bus controller
- Completed the multicore block diagram and state diagram for bus controller
- Assisted in debugging the updated D-Cache code
- Wrote a testbench for the D-Cache design
- Wrote the parallel algorithm
- Worked on results section
- Worked on conclusion