Single Core Pipeline vs. Multicore Pipeline Report

ECE 437

Javier Gonzalez Souto & Cyrus Sutaria

Yuhao

December 14th, 2015

Overview:

The hardware that executes a program is called a processor; this report will contrast and compare the different types of processors that were implemented, pipelined single core with and without caches and pipelined multicore. Although, all of them have the same functionality, the main difference between them is noticeable in their performance, time taken to execute a program, and the complexity of each of them.

Processors fetch instructions from an executable program, they decode the information (read the registers and the data in them), execute basic operations (add, subtract, shift, and, or, etc), read from or write to memory, and lastly, write the computed value into registers so that other instructions can gather the information computed on a previous instruction.

Pipeline processors go through five instructions at a time, the first instruction is fetched, another may be on the decode stage, one in the execute stage, one in the memory and one can be on the write back stage. However, these types of processors have a few hazards that need to be taken care of. The only hazard taken care of is Read After Write (RAW), as the next instruction may not read a register when the previous one is writing to it. Other hazards are taken care of by the architectural designs. Therefore after the execute stage, the output from that stage becomes the input to the other. As stated above, both types of processors compared in this report have the same functionality; however, the multicore processor does all of the steps as the pipeline but with 2 cores at a time. It fetches the instruction, decodes it, executes it, if it needs to read data from memory or the other cache, it loads it or stores it, and lastly it writes the final result into the register destination. On one hand, the multiprocessor should be faster due to higher throughput on parallel programs. On the other hand, the pipeline processor ends up being much **faster** as either the programs are not parallelized well or there is more overhead in our multicore design. For testing we decided to use the mergesort algorithm for the singlecore and dual mergesort for the multicore which use most of the instructions available with and without dependencies which cause pipeline hazards. We chose a latency of 10 and a list size of 5 on the mergesort to ensure that even with many elements, the right computation takes place and it is loaded from and stored correctly in memory.

Furthermore, the processor with caches should be faster than without caches due to decreased memory latency once a block has already been brought to cache. As it turns out the processor without caches is much faster, again possibly due to un-optimized code or increased overhead in our design, than the single core uniprocessor with caches and the multicore.

Processor Designs:

Pipeline:

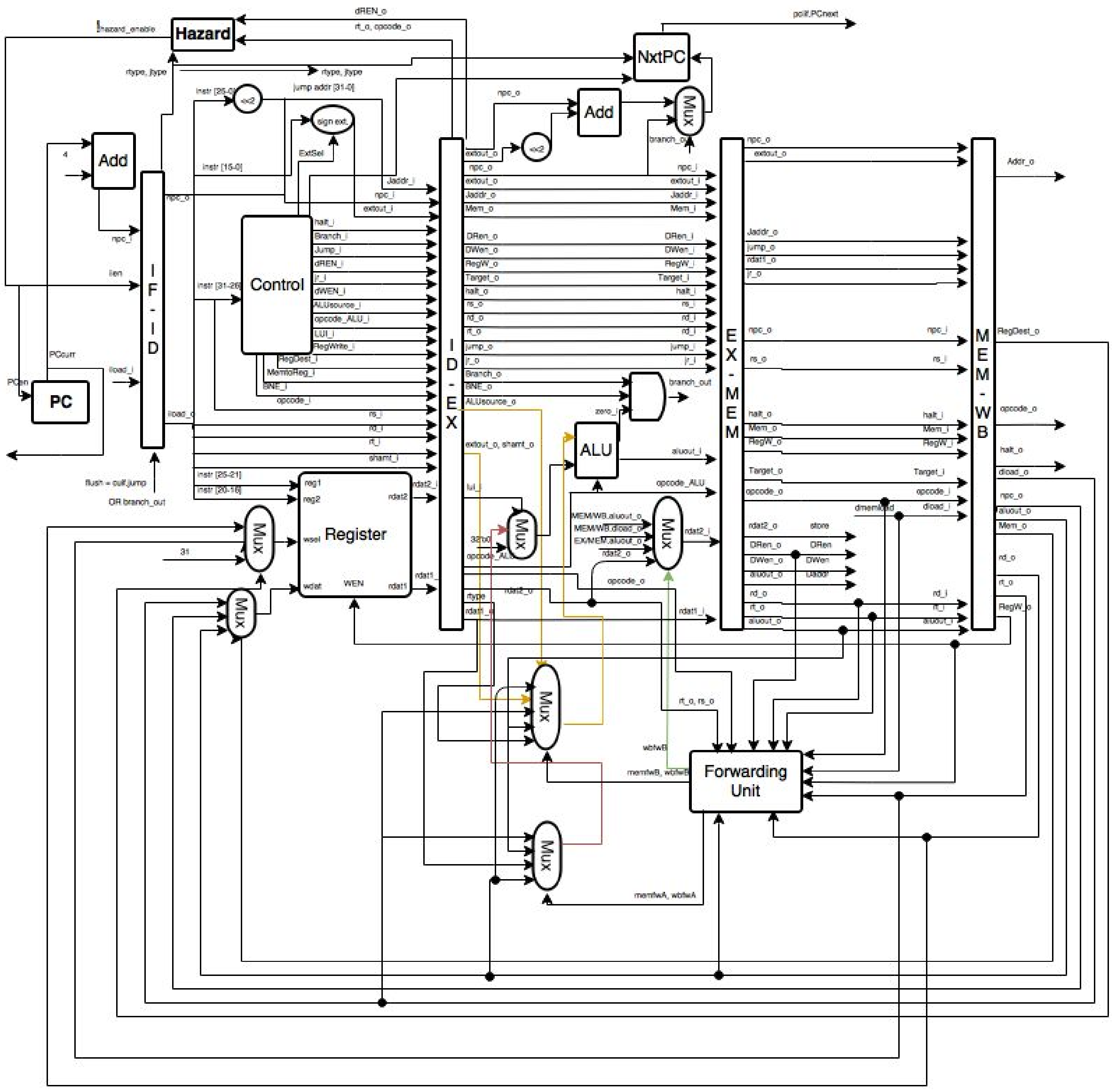

Fig A

Cache State Diagram:

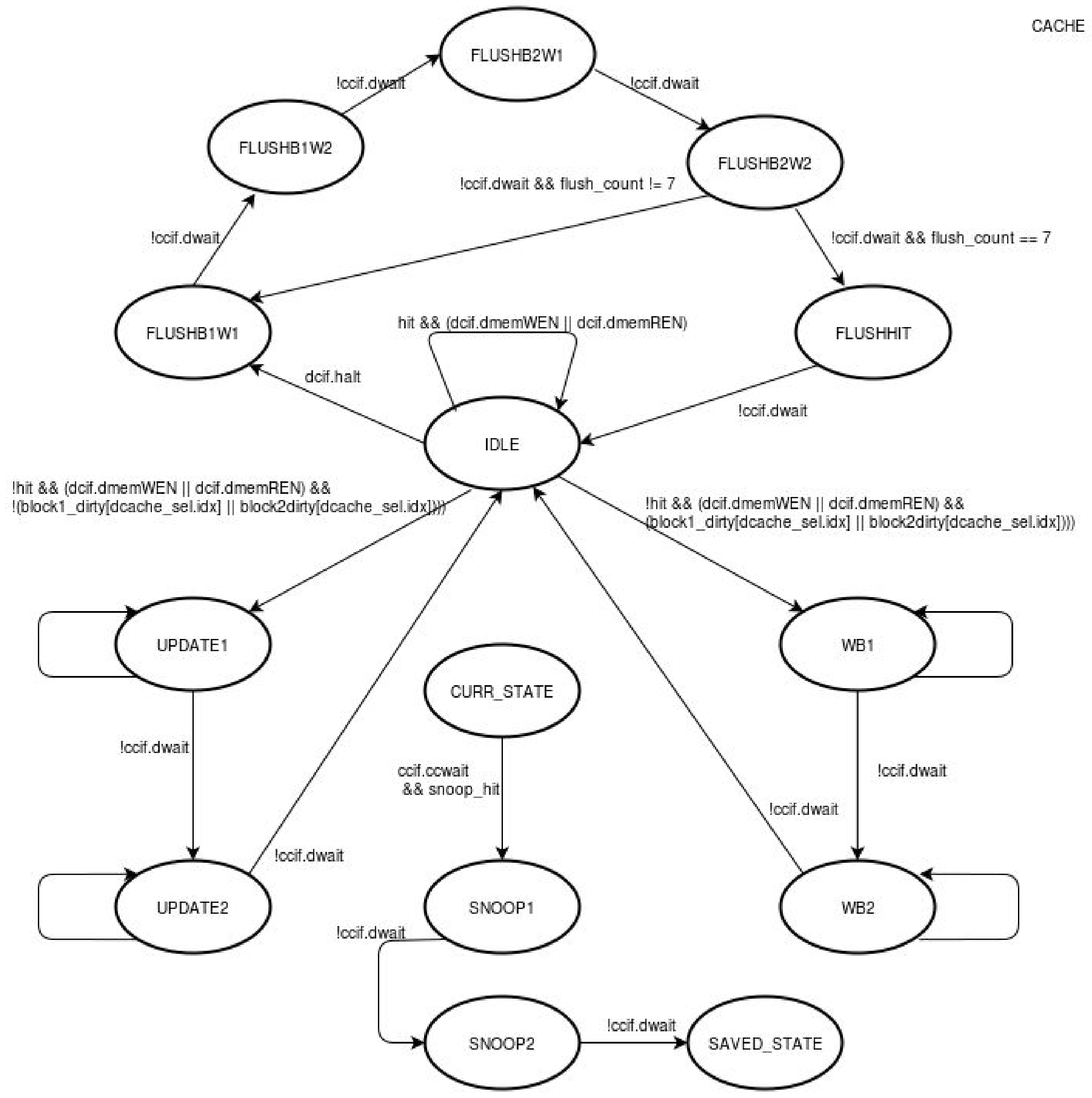
CACHE
FLUSHB2W1
!ccif.dwait
!ccif.dwait
FLUSHB1W2
FLUSHB2W2
!ccif.dwait && flush_count != 7
!ccif.dwait
!ccif.dwait && flush_count == 7
FLUSHB1W1
hit && (dcif.dmemWEN || dcif.dmemREN)
FLUSHHIT
dcif.halt
!ccif.dwait
IDLE
!hit && (dcif.dmemWEN || dcif.dmemREN) &&
!(block1_dirty[dcache_sel.idx] || block2dirty[dcache_sel.idx])))
!hit && (dcif.dmemWEN || dcif.dmemREN) &&
(block1_dirty[dcache_sel.idx] || block2dirty[dcache_sel.idx])))
UPDATE1
WB1
CURR_STATE
!ccif.dwait
ccif.ccwait
&& snoop_hit
!ccif.dwait
!ccif.dwait
!ccif.dwait
UPDATE2
SNOOP1
WB2
!ccif.dwait
SNOOP2
!ccif.dwait
SAVED_STATE

Fig B

Cache Block Diagram:

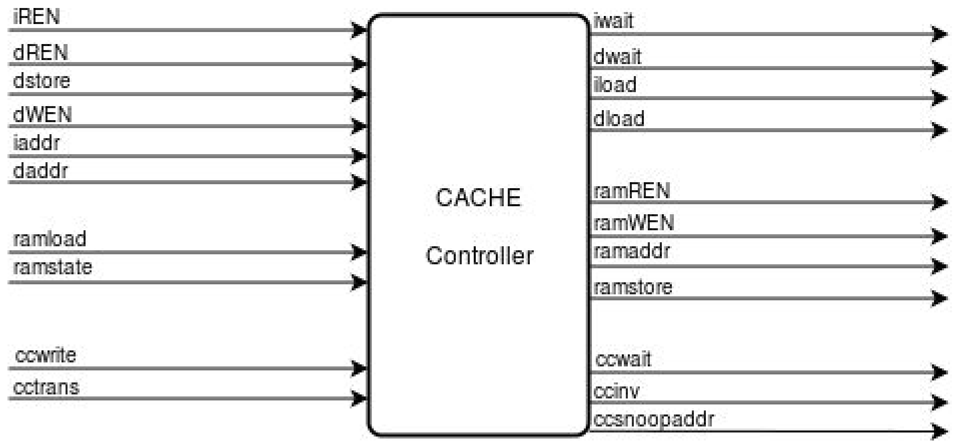

Fig C

Multicore State Diagram:

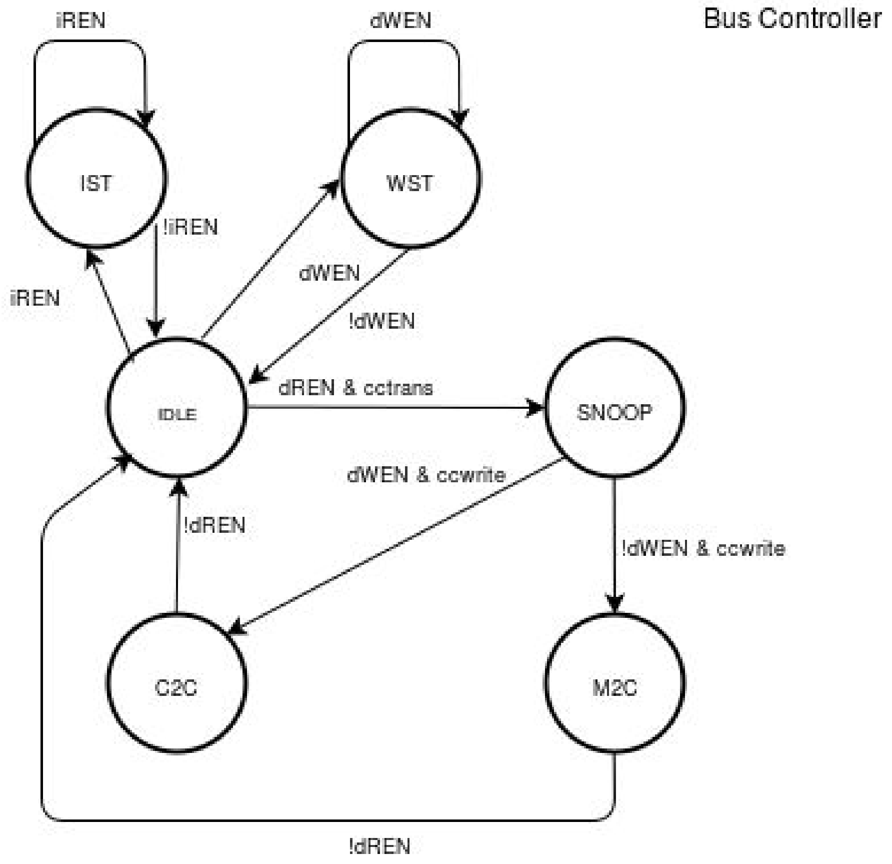

Fig D

Multicore Block Diagram:

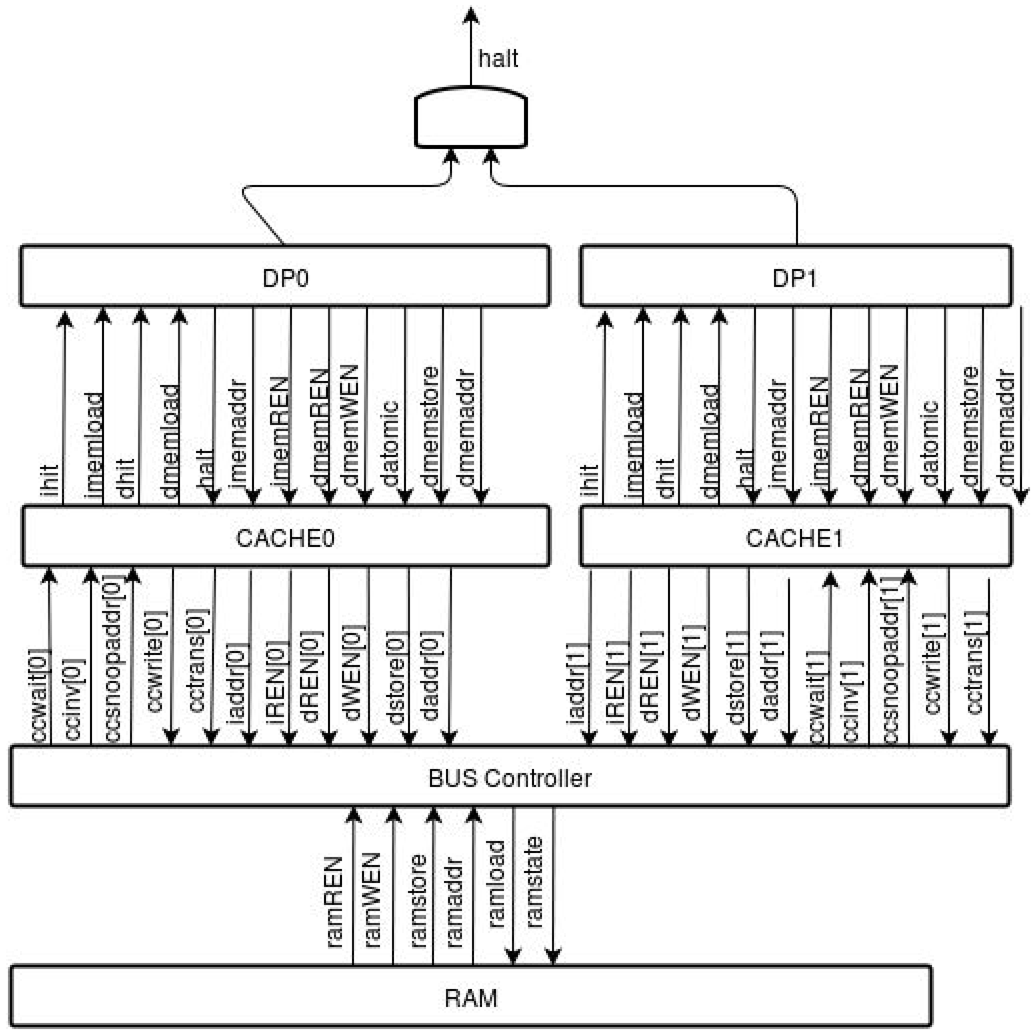

Fig E

Results:

Pipeline: CPUCLK Fmax = 47.82 MHz, RAMCLK Fmax = 74.07 MHz

Pipeline + Caches: CPUCLK Fmax = 43.73 MHz, RAMCLK Fmax = 98.93 MHz

Multicore Pipeline: CPUCLK Fmax = 26.84 MHz, RAMCLK Fmax = 51.9 MHz

	Pipeline design:	**Pipeline + Caches design:**	**Multicore pipeline design:**
Fmax	43.73 MHz	39.77 MHz	25.95 MHz
CPI	18.1	17.77	20.86
IPC	0.055	0.056	0.048
Latency	115 ns (calculated using CPUCLK Fmax)	130 ns (calculated using RAMCLK Fmax)	195 ns (calculated using RAMCLK Fmax)
Speed Up	N/A	0.27 (slowdown vs pipeline)	0.51 (slowdown vs pipelined with cache)
Critical Path Setup Slack Worst Case	21.337 ns from ram_block3a42~porta_we_reg to ide.halt_o	13.922 ns fromexme.alu_out_o to ram_block3a48~porta_we_reg	35.55 ns from flush_idx_count[1] to ram_block3a39-porta_we_reg
Design Resources Used	Total combinational functions: 3,390 Total registers: 1,723	Total combinational functions: 6,819 Total registers: 4,187	Total combinational functions: 14,163 Total registers: 8,211

The first result in the table above shows that for each improvement in the processing unit's design the synthesis frequency decreases. This may be surprising to some people. It is likely due to the increase in complexity between the designs. As complexity increases, the combinational logic paths may become fairly long. The delay in these paths must fit within a clock cycle and therefore the estimated frequency for each design actually increases.

As one can see in the results table, cycles per instruction and instructions per cycle are interrelated. One is the inverse of the other. One can see that the cycles per instruction decreased (or the instructions per cycle increased) slightly as expected from the uniprocessor without caches to the uniprocessor with caches. The instructions take less cycles to complete because there are many loads and stores and the point of the caches is to reduce the time spent for these instructions waiting for the memory operations to complete. Caches still do not always already have the memory block stored, so sometimes there are still high latency RAM accesses, explaining the modest improvement. We use a modest RAM latency of ten. If increased to a more realistic one the performance improvement may be more pronounced. From the uniprocessor with caches to the multicore with caches there was an increase in the cycles per instruction or a decrease in instructions per cycle. This counts as a performance decrease on a per-instruction basis. This may be to increased overhead of parallelization both in the software and hardware. The software used for the parallel mergesort may use more instructions to accomplish the same thing in parallel. Performance may improve past the offset by the overhead when a much larger input set size is used.

From single core without caches to single core with caches, the total combinational functions roughly doubled. The registers used roughly quadrupled. FPGA synthesis tools can allocate memory differently based on the design's HDL structure. The cache data storage might be using more registers instead of being allocated RAM due to this phenomenon. There are also many additional registers and a state machine required to implement caches. This increases complexity both in terms of registers and combinational logic. From the uniprocessor with caches to the multiprocessor with caches combinational complexity more than doubled. The number of registers roughly doubled as well. This is expected, and surprisingly it doesn't seem like there was too much extra resource usage other than just double of the caches. There is some extra due to the added complexity of coherence operations. These are necessary to provide the illusion of write atomicity.

Conclusions:

There are a few details we can observe by looking at the results. First let's look at the methods used to determine these results. All designs were synthesized with timing models generated. The mergesort algorithm was then run on each design. We then tested the memory dumps for correctness and recorded the run-time information. After synthesis the logs had been recorded to combine with the program run results to allow us to compute the values in the table above.

In our lab we have implemented multiple types of processors and learned the differences between these. There are the ideal differences between these types and then, as you can see with our results, there can be some more practical differences. The single core without caches turned out to be the fastest as far as program run time. The single core with caches did run individual instructions faster on average than without caches, but latency increased and run time suffered. This could be explained by a very low RAM latency as well. The multicore processor was larger and did run the program for a longer time, but this time the algorithm was a different parallelized program. It may be faster and overcome the overhead penalty with much larger instruction sets. Through this lab we learned what kinds of effects our design decisions can have on a completed design. As can be seen in our results, increased complexity can increase the latency of a processor. The type of test obviously can have an effect on the results as well. Performance metrics must be used carefully as well some can appear to be slower when that processor actually runs faster. These are all useful things to acknowledge.