Design and implementation of a shared memory dual-core processor on an FPGA

ECE 437

Computer Design and Prototyping

4/22/11

Alex Reyes (mg225)

Matthew Barga (mg217)

Executive Overview

The goal of these FPGA targeted designs was to implement functioning processors with support for execution of assembly programs written with a subset of the MIPS instruction set architecture. The two designs that are discussed in this report are a single-core pipelined processor with data and instruction caches, and a dual-core processor composed of two modified, pipelined cores. The dual-core processor design also required the integration of a memory coherence controller to facilitate communication and transfer of data between the caches and memory, and to maintain sequential consistency in the operation of both cores. All functional features of both designs were successfully implemented.

This report provides discussions on the design process followed for developing each processor and gives a performance analysis of each design including benchmarks based on execution of two separate sorting algorithms. Another purpose of the report is to present the two designs side-by-side to highlight the contrasts in their structure and performance. Perhaps the most beneficial insight into processor design was gained after contrasting the performance of both designs. Development of the memory coherence controller proved to be quite challenging, but the significant increase in performance of the dual-core design over the single-core design for the selected workload more than justified the extra design challenges encountered.

Pipelined Processor Design (FIG. 1)

The largest hurdles in the pipelined processor design were implementing hardware for allowing nops and bubbles in the pipeline, allowing for single memory read/write control and placement of branch control hardware. The problem of introducing bubbles and stalls in the pipeline was related to both of the other problems. The pipeline registers needed to be stalled if there was a forward dependency on a value that had yet to be loaded from memory. With memory latency enabled, an 'allwait' signal was sent to every pipeline register to stall them while waiting for a memory read to complete if needed. Nops also had to be introduced to one register after a branch was taken to flush the invalid instruction it contained. This was implemented using a nop enable signal that set all control signal values for that instruction to '0' when the nop enable was asserted.

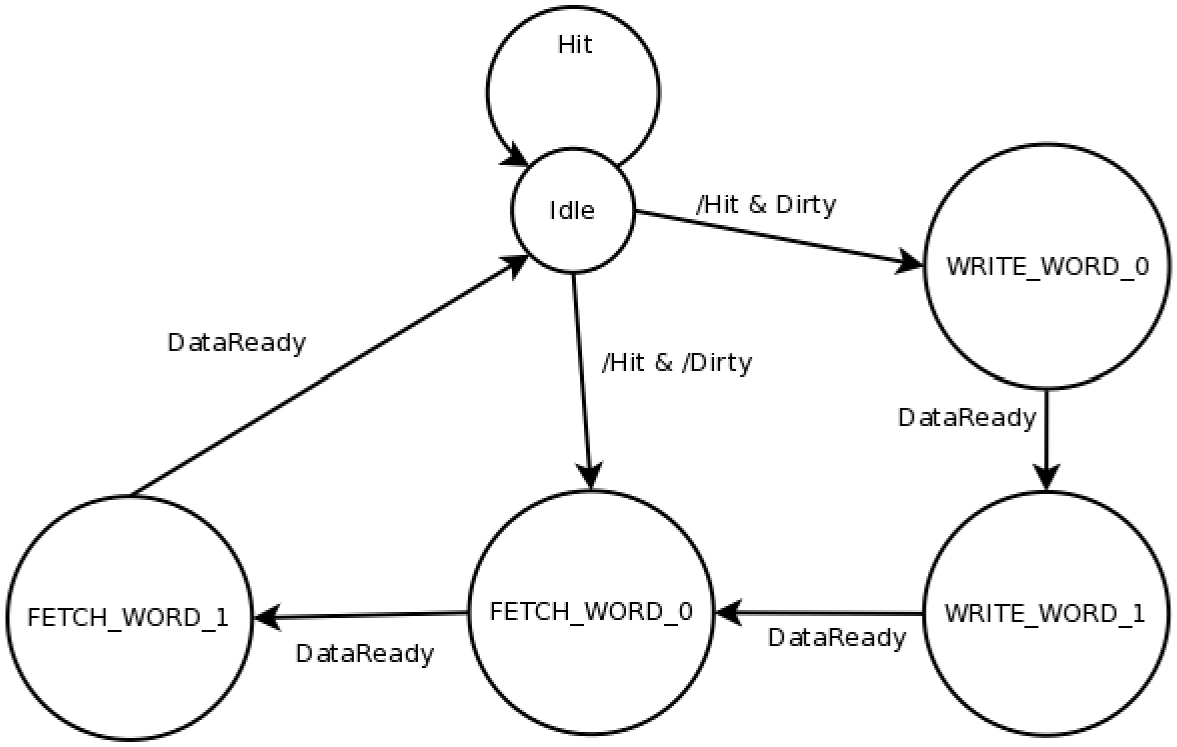
One last design decision was to move branch logic to the 2nd stage of the pipeline. This involved forwarding the ALU result signal to the second stage if the branch calculation depended on an ALU result that had not yet been written to a register. For the pipelined design, most of the code from the single cycle design was reused as the foundations of the designs were very similar. A consideration that was needed for the pipelined design that was not relevant in the single cycle design was how to divide up the stages. This came up in placing branch logic and some multiplexers selecting among calculated values to pass forward. Looking back, more care should have been taken during this process to reduce the critical path length.

The new pipelined processor design also employed the use of instruction and data caches (Figure 3). The data cache was a 256 byte, 2-way set associative cache. A least recently used policy was adopted for replacement. To implement this, an extra bit (LRU bit) was stored for each block in each way and whenever data was written to a block in one way, that bit was reset to '0' while the other way had its LRU bit set to '1'. Data was always written to the way that had its LRU bit set to '1', or the least recently used way. The data cache was also implemented as write allocate. This meant that whenever the CPU performed a write to the cache and caused a miss, both blocks of that entry were read in from memory before overwriting the entry.

To facilitate memory accesses from the two separate caches to the single external memory, a memory controller was developed. In the memory controller, precedence was given to data read and write operations over instruction fetch operations. However, to avoid depriving the CPU of instruction cache access, the instruction cache was given permission to fetch after 4 consecutive load or store instructions were observed. A state diagram describing the operation of the data cache controller is shown in Figure 4.

FIG. 4

Data Cache Controller State Diagram



Multicore Processor Design (FIG. 2)

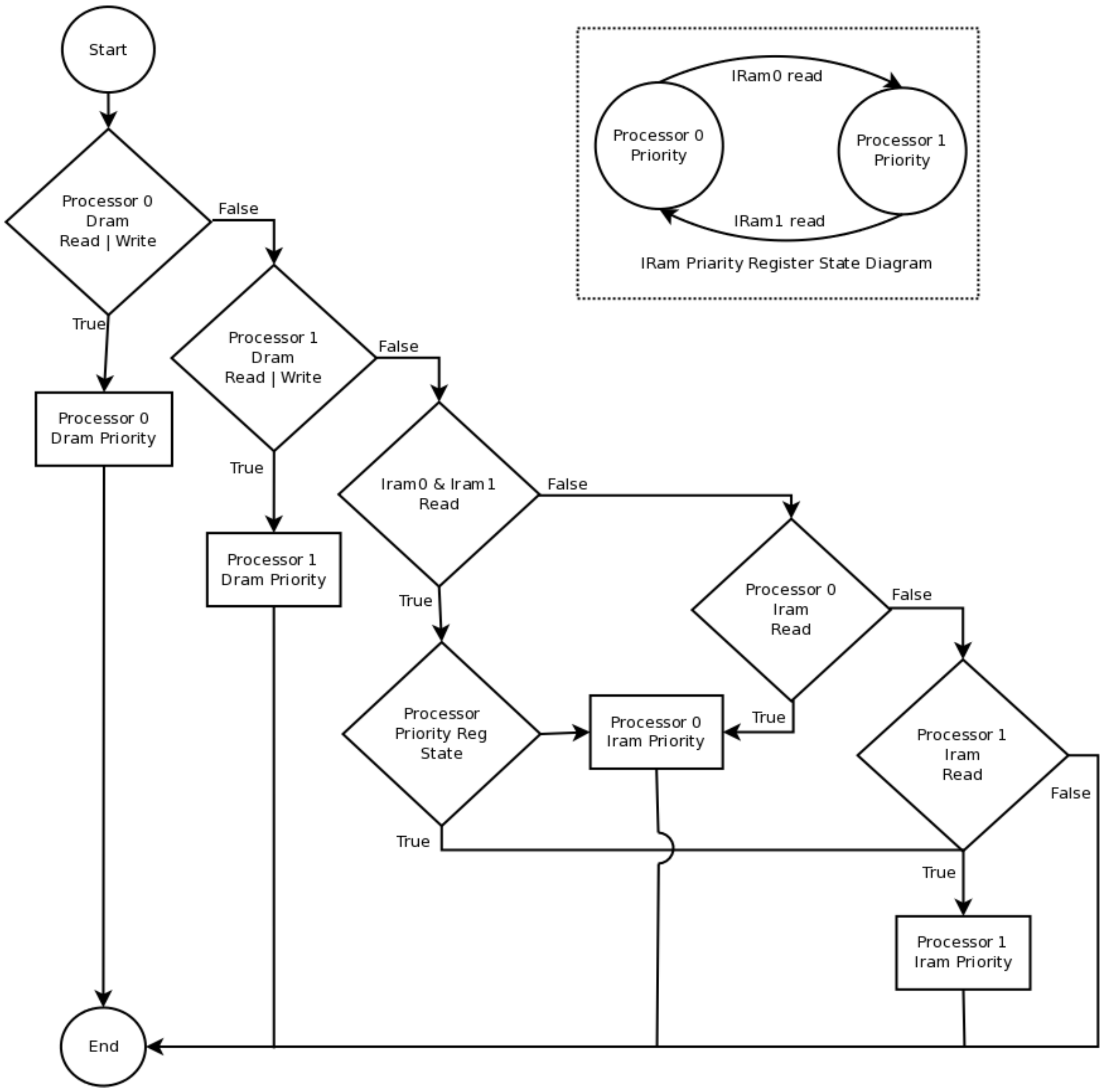
The multicore processor was implemented by connecting together two slightly modified pipelined processors with their own caches, through access to a common external memory by way of a coherence controller. These modified pipelined controllers also supported two extra MIPS instructions, ll and sc. These two instructions were implemented in the same way as a load and store, respectively, with the addition of two more control signals used to inform the data cache when one of these instructions was being executed.

The coherence controller was another major part of the multicore processor design. This controller had a dual purpose of arbitrating access to memory from the two instruction caches and the two data caches, while also monitoring coherence between the separate data caches in both processors. The only modifications to the cache hierarchy were the addition of signals to facilitate snooping actions between the data caches of both processor cores.

In order to coordinate the memory accesses of the different caches, the coherence controller contained a simple one bit 'last to access' register that kept track of the last processor that was given priority. Priority was then given to the caches according to the following rules. Data caches were given first priority, with processor '0's cache having priority over the cache of processor '1'. The reason behind this decision was simplicity of implementation. If neither data cache was requesting memory access in a clock cycle, then the instruction caches were given access. The question of which processor's instruction cache would be given priority was determined by the value in the 'last to access' register. This was done to allow both processors to progress at a similar rate. A flow chart and state diagram describing the coherence controller access priority algorithm is show in Figure 5.

FIG. 5

Coherence Controller Flowchart and State Diagram


IRam0 read
Processor 0 Priority
Processor 1 Priority
IRam1 read
IRam Priarity Register State Diagram
Start
Processor 0 Dram Read | Write
False
True
Processor 0 Dram Priority
Processor 1 Dram Read | Write
False
True
Processor 1 Dram Priority
Iram0 & Iram1 Read
False
True
Processor 0 Iram Read
False
True
Processor Priority Reg State
True
Processor 0 Iram Priority
Processor 1 Iram Read
False
True
Processor 1 Iram Priority
End

Another important purpose of the coherence controller was to manage snooping by forwarding data from one data cache to the other to maintain coherence. Keeping track of the state of each block, whether invalid, modified or shared, was the duty of each data cache. Signals indicating and controlling the state of each block in the cache were interfaced between the coherence controller and each one of the data caches. When a data cache requested an address, the coherence controller would forward the data from the other cache if there was a snoop hit, or proceed to get the data from memory if the address was not valid in the other data cache.

FIG. 1.0

Pipelined

Processor

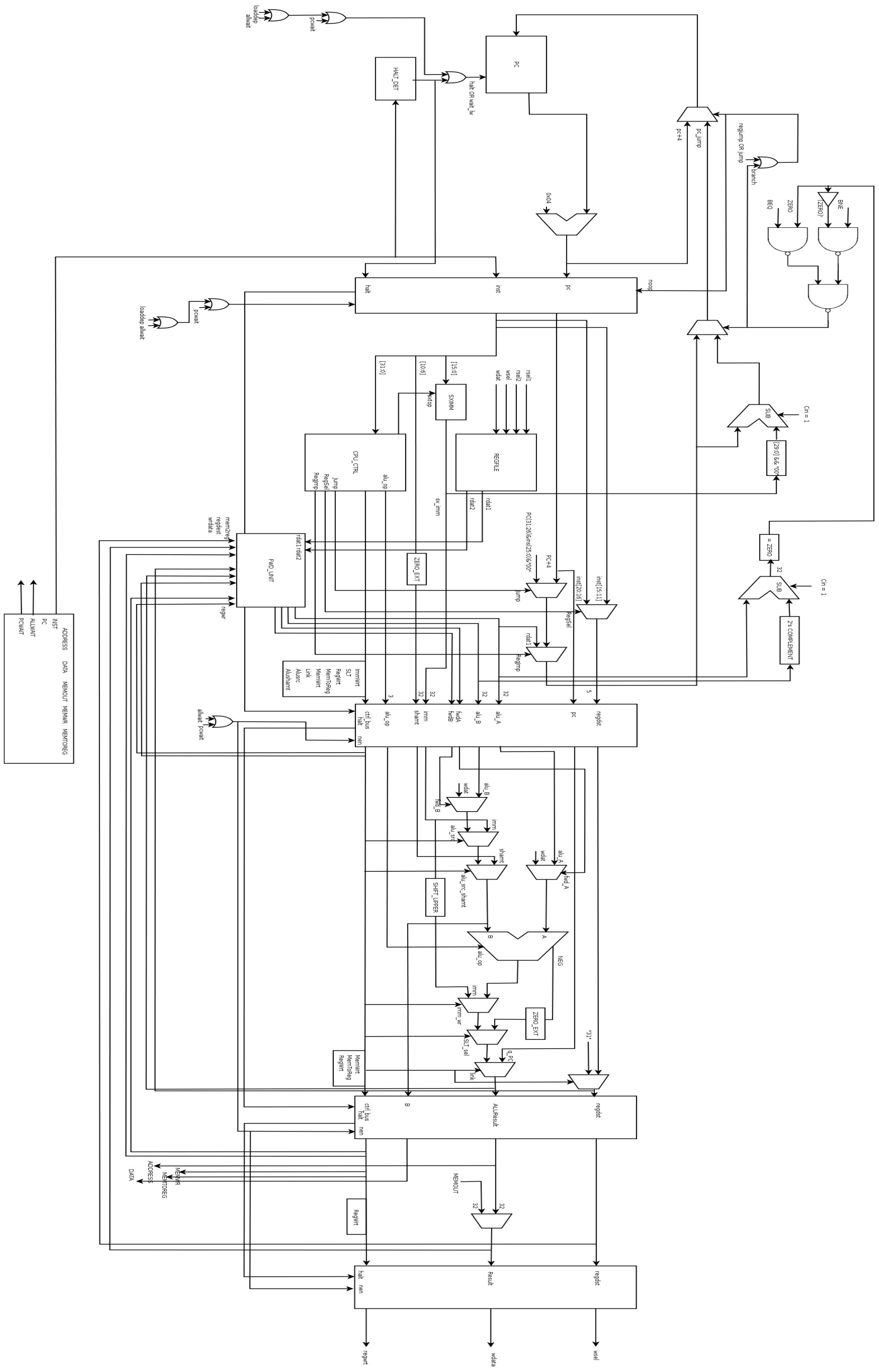


FIG. 2

Multicore

Processor

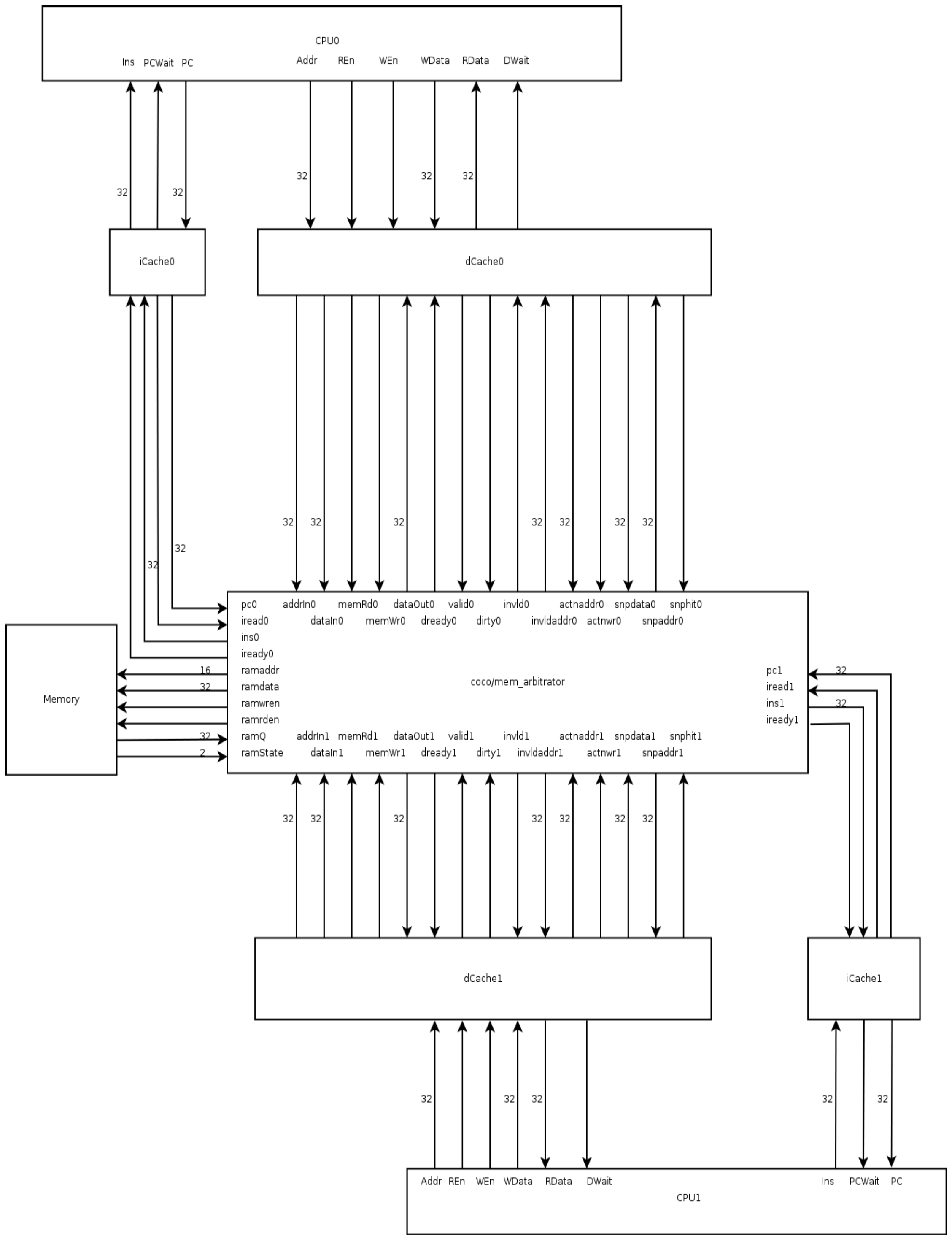
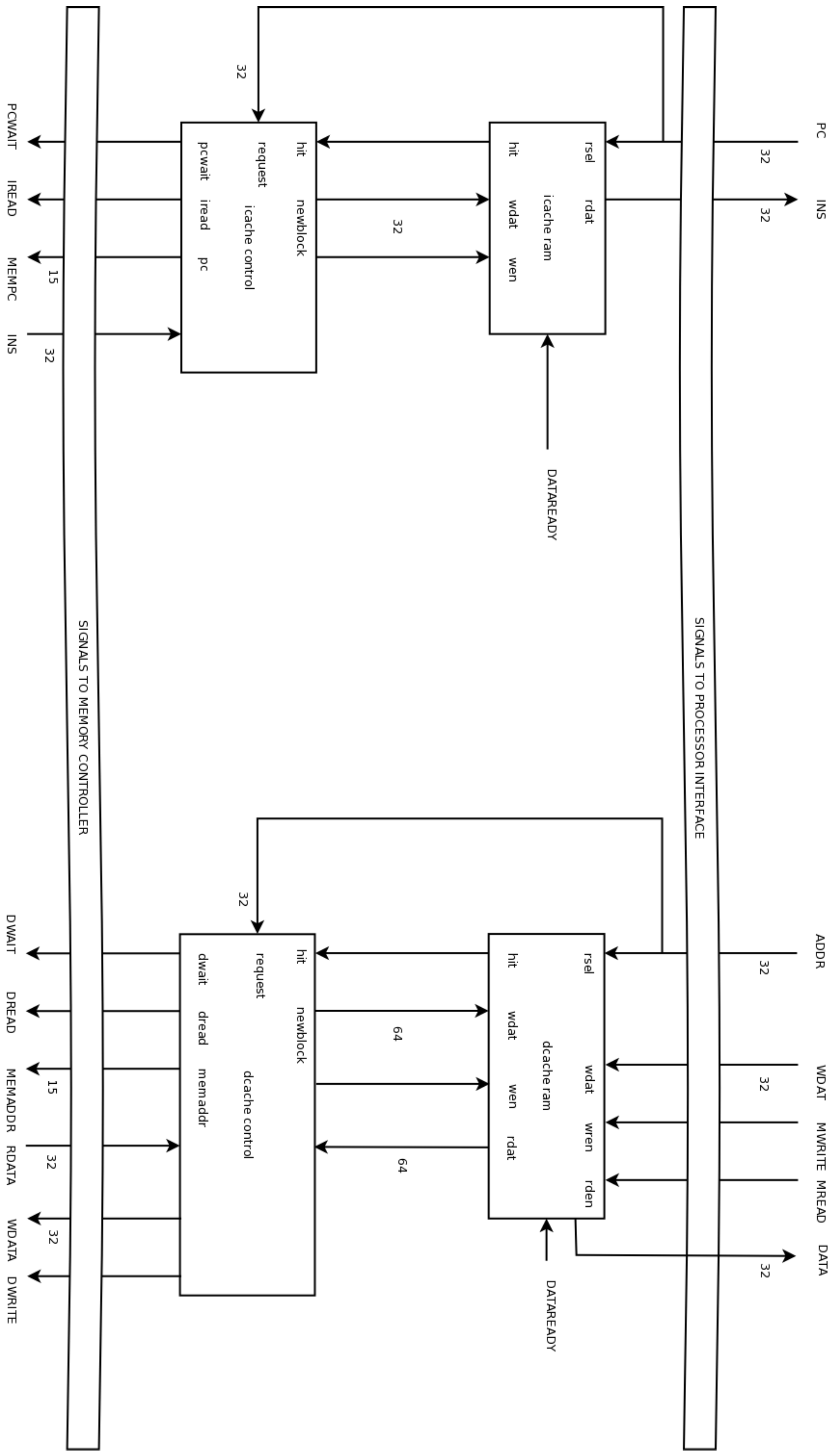


FIG. 3

Cache

Diagram



Processor Debugging

After examining the assembly source file and memout.hex files in Table 1 of the handout, it appeared that the result stored in memory address 0x0240 of the memout.hex file was incorrect. If all instructions executed correctly, this register should have held the value 0xBEEF + 0x2000 - 66 = 0xDEAD. Instead, it held the value 0xDEEF = 0xBEEF + 0x2000. The problem appeared to be that one processor was overwriting the data written by the other processor. Instead of both processors operating on the same data sequentially, they appeared to both be using the initial value of 0xBEEF and then writing back their results without waiting for the other one to release the lock.

One possible cause for this error could have been an incorrectly handled sc race condition. If both processors happened to request a lock on the same address in the same clock cycle, it could have been that both were incorrectly acquiring the lock, when instead one should have been given priority and the other one should have had to retry acquiring the lock. If both processors acquired the lock, they would each run their arithmetic operations on the result variable and then store the result back. The value on the memout.hex file would have been the result stored by the processor that would have happened to finish last, instead of the combined result of each processor's arithmetic operation on the memory address. To identify this error, one could look at the lock signals to verify that only one processor was acquiring the lock at any given time. To verify if this is a possible cause, some code could be written that guarantees that the two processors reach the lock code sequentially one after the other, eliminating the possibility of a race condition. If the error persists, then a race condition would not be causing the error.

Another possible cause could be incorrectly implemented snooping. If one processor performed the arithmetic operation and stored the result back into the cache and the other processor then attempted to read that address, it should have snooped the data from the other processor's cache instead of reading it straight from memory. If a snoop hit was not being correctly identified, then one processor would have loaded an invalid value from memory instead of the modified value from the other processor's cache. The result in the memout.hex file would then be the result of only one processor's arithmetic operation. To identify this problem, one should verify the assertion of a snoop hit signal when the second processor tries to access the result variable after the first one has written to it. One test that could be run to verify if this could be causing the problem is to write a simple code where one processor loads and updates an address in memory and then the second processor reads from that address and copies the data to different address. The data in the addresses written to by both processors should match and be equal to the updated value from processor '1'.

Results

For the pipelined processor design (FIG. 1), benchmark calculations were based on run statistics for a sequential insertion sort algorithm sorting 100 random data samples. The performance metrics worth making note of are the IPC and instruction latency measurements. The benchmark performance is outlined in Table 1 below.

$$latency = Clock\ period \times CPI$$

The pipelined processor design required 7,423 logic elements and 4,913 registers. The critical path for the design was 38ns. This value was substituted in as the processor clock period value for all calculations requiring a clock period. The majority of this path is present in the ID stage where branch-taken control is generated, however the values used to determine the branch taken signal are forwarded from the ALU unit in the EXEC stage. The path begins at the ALU output in the EXEC stage, and propagates back to the program counter register. A path similar to this was hypothesized to be the critical path even before the analysis was completed as branch calculation signals propagate through two ripple-carry adders and other logical elements.

Performance Metrics for Pipelined Processor with Cache

Table 1

Clock Period (ns)	38.00
Clock Freq. (MHz)	26.32
IPC (1/CPI)	0.60
Avg. Ins. Latency(ns)	63.20
No. Logic Elements	7423
No. Registers	4913

The final design to be analyzed was the multicore processor. The block diagram for this design is FIG. 2 above. The program used for benchmark calculation was a parallel merge sort algorithm that was ran on the same data set that was tested with the insertion sort algorithm ran on the single core pipelined processor. The benchmark specs for this design are outlined in Table 2 below.

The final chip area for the multicore processor design required 17,515 logic elements and 9,757 registers. The critical path for this design was 38ns. Not surprisingly, this critical path for the multicore design matches the critical path outlined above for the single core pipelined processor. As the multicore is fundamentally two of the pipeline designs coupled together with a memory controller, if the critical path of the pipeline design is larger than any of the new signal paths added for the memory controller, then the critical path of the pipelined design and the multicore design should stay the same.

The caches comprised the bulk of the processor area in these designs. In the benchmarks discussed in this section, the caches would account for an apparent decrease in the calculated throughput when compared to the pipelined design without caches. This is because there is more cycle overhead used for coordination of memory accesses in the design using caches. This is misleading however, as there was no latency associated with RAM access for the prior pipeline design benchmark. For the designs with cache implementation, depending on the spatial and temporal locality of data accesses in the programs run, the majority of memory accesses could be associated with cache hits. These accesses would be serviced on local cache RAM vs. external RAM. Even with the overhead associated with access coordination, an access to cache RAM would have significantly lower latency compared with an access to external RAM. For the average program it would become evident that the caches would provide a significant boost to throughput.

Performance Metrics for Multicore Processor with Separate Caches

Table 2

Clock Period (ns)	38
Clock Freq. (MHz)	26.32
IPC (1/CPI)	1.21
Avg. Ins. Latency(ns)	31.34
No. Logic Elements	17515
No. Registers	9757

Conclusions

The two processor designs outlined in this report were able to execute all desired functionality without fault. The average CPI for the pipelined design was also slightly greater than 1 for the tested program binary, which was the targeted number as it means that on average one instruction completes in each cycle or stage of the pipeline. The critical path was however surprisingly large in the pipelined design due to the location of branch logic in the 2nd stage.

The multicore processor was able to significantly improve on throughput for the sort program tested. The average IPC increased from 0.6 for a sequential sort on the single core design to 1.21 for a parallel sort on the same data set on the multicore processor. This translates to a 2x increase in throughput. Although not observed in this analysis, the implementation of the caches in the design would also account for a further increase in throughput. If RAM latency was simulated and the pipelined design with caches was compared to a pipelined design without caches, a significant increase in CPI would be noticed for the design with no caches for most programs with any locality. This again is due to the very high latency of an access to external RAM compared to the relatively low latency associated with an access to cache RAM.

The main advantage of the pipelined processor design was the simplicity of it. There may be an increase in throughput for a multicore design vs. a pipelined design, but this is only realized when parallelism can be taken advantage of in the programs written for the multicore architecture. If a problem isn't parallelizable, then the overhead associated with keeping the caches in sync will actually decrease the throughput when compared to a single core pipelined design with only a single set of instruction and data caches. The obvious advantage of the multicore design is the significant increase in throughput that can be realized for a large set of problems where parallel algorithms can be applied. The multicore is also flexible in that if a program cannot be implemented in parallel, it can still be run on a single processor core. The implementation of the multicore design was however considerably more difficult to develop than the single core due to all the considerations needed for memory coherency.