Final Design Report

ECE437: Computer Design and Prototyping

Jevin Sweval

TA: Abhisek Pan

2010-04-30

1 Executive Overview

As part of the ECE437 course, I have realized a MIPS-subset CPU in single-cycle, pipelined, pipelined with cache, and dual-core pipelined with cache forms that are synthesizable for the Cyclone II FPGAs. The subset of MIPS implemented allows for general purpose computing while still keeping the design simple. The first design completed, the single-cycle implementation, introduced me to the major components of the CPU: memory, ALU, control logic, and the register file. The single-cycle CPU works correctly for all test cases, runs at 22 MHz, and has an average CPI of ≈ 1.2. The pipelined design required careful navigation of tricky data hazards (some of which were only unearthed during the last lab!) but paid off with a much faster clock rate. It runs correctly at 75 MHz with a CPI of ≈ 39 with maximum memory latency. Instruction and data caches were added to the pipelined design in order to decrease the average memory access time. The caches slowed the design down to 28 MHz but drastically decreases the CPI to ≈ 6.1 at maximum memory latency.

Using the cached, pipelined design, I created a dual core version of the CPU. This required a coherency controller between the data caches and an implementation of the MSI coherency protocol. In addition, the `LL/SC` instruction were implemented for locking. For the simple dual core design, I opted to include most of the coherency logic in the data caches themselves, making the coherency controller fairly trivial. This design decision would not scale to more cores but won in its simplicity for my purposes. The dual core design worked correctly for all test cases and ran at a glacial 7 MHz (due to long critical paths in the caches and a nearly full FPGA) with a CPI slightly higher (due to memory access contention between cores) than the cached pipelined design. Of course, with two cores, the throughput is essentially doubled for embarrassingly parallel workloads. The dual core design was difficult to implement but I am now confident in my understanding of CPU design.

2 Processor and Cache Design

All of the designs that I implemented closely followed the course textbook's designs. I feel that the most significant design choice was my coding methodology. All designs are coded almost entirely in behavioral VHDL. I find this type of code to be much easier to write and debug because it can be single-stepped in a debugger yet, if written carefully, can compile down to logic that is just as efficient as a structural design (excepting one case that I will talk about later). Also central to my coding methodology were VHDL records. The input and output signals to a block were wrapped in a record (similar to C `structs`). Using records, one can add and remove signals to blocks without the time-consuming and error-prone edits of port and entity declarations. The records also group the signals in the waveform viewer, allowing for a tidier waveform window. I extensively use `numeric_std` types and operators in my code to vastly cut down on code size (my ALU was two lines of code per operation) and to let the synthesizer automatically chose the best implementation in hardware. Most non-numeric types are declared as enumeration types. For example, the instructions are decoded into opcode enumerations. These enumerated types are human-readable in waveforms and source code instead of appearing as indecipherable bitstrings.

The pipelined design without cache was by far the fastest design in terms of clock speed. This is because the pipeline registers chopped up the critical path into much smaller parts. However, without a cache the performance was terrible with maximum memory latency (CPI $\approx$ 39). Adding a 16-instruction instruction cache and a 64-word, 2-way associative, LRU data cache to the design reduced the CPI to $\approx$ 6.1. The cost of this massive improvement to the CPI was a slower f_{max}. The cached design runs at 28 MHz but is about 2× as fast as the non-cached design in terms of wall time. This slowdown in clock speed was due to a very long critical path through the data cache. Examining the resulting FPGA resources indicated that the behavioral coding style utilized in the data cache created many unneeded, bloated muxes — some over 1000 bits wide! I contemplated a redesign in pure RTL but the effort would have been too great.

The dual core design required more than simply copy-pasting two cores into a single VHDL file. I had to add a memory arbiter to arbitrate the RAM between the instruction caches and the coherency controller. This arbiter made sure to store the last serviced consumer so that no single consumer would be starved of data. The coherency controller sat between the two data caches and the memory arbiter Its purpose was to ensure data coherency thorough the MSI protocol. My implemented coherency controller was really just another memory arbiter that negotiated access to the RAM between the data caches, taking into account that one cache may need to perform a flush action on behalf of another. The majority of the coherency logic was placed into the data caches for simplicity. Each cache snoops the other cache's memory accesses. If both caches try to access the same line, one cache will wait for the other (the cache granted memory access via the coherency controller) to complete its access before continuing. To ensure consistency after processor writes to a shared line, a cache will direct its sibling to pause while it flushes the line to memory. The paused cache will then resume and read the written-back value from RAM. Each core has an `LL/SC` link register that is wired to the other core. Each core checks the other's link register during these atomic operations to ensure that they execute correctly. The dual core design, with two bloated data caches used $\approx 90\%$ of the FPGA resources (6 hour synthesis!). The fitter, faced with little breathing room was unable to keep the clock rate up. Thus, the final design runs at 7 MHz. The CPI is slightly increased over the single core design due to memory access contention. Especially during start-up, both cores are trying to fill their instruction and data caches and will often wait for the other memory consumers to finish before proceeding.

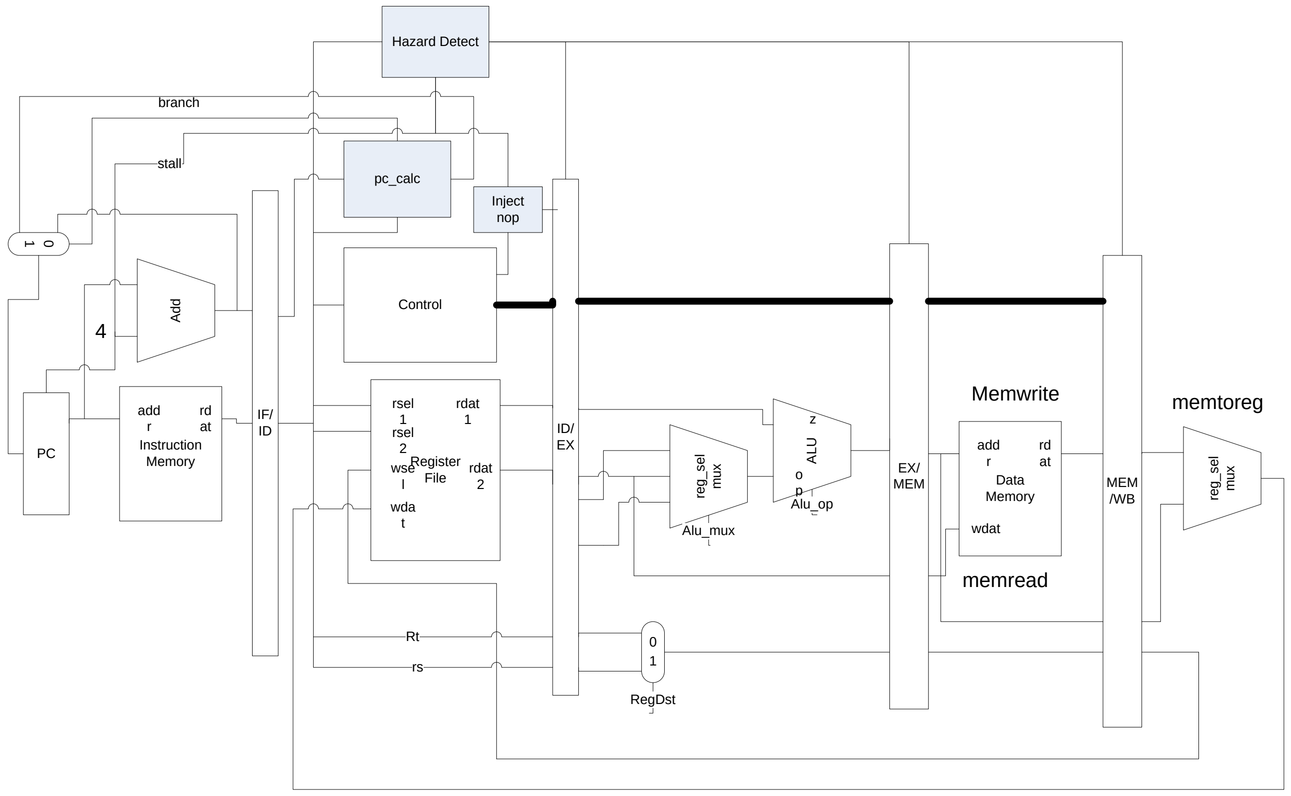


Figure 1: Pipeline Block Diagram

3 Processor Debug

The output should have this memory value instead of the one listed: `:0400900000000002xx`. It is likely that the coherency controller isn't working, leading to one core not correctly having its cached `res` value written back to memory before it is read by the other core.

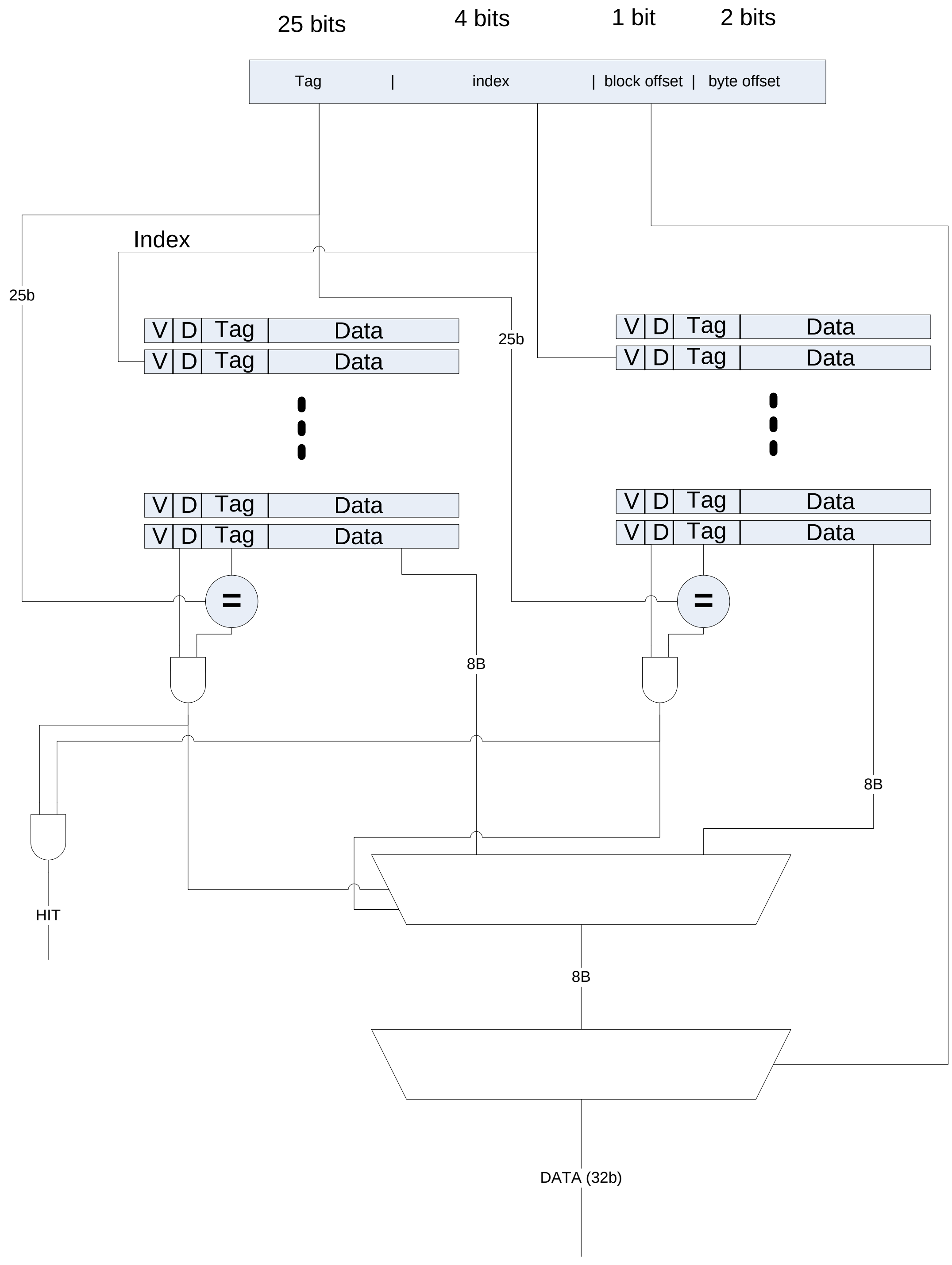


Figure 2: Data Cache Block Diagram

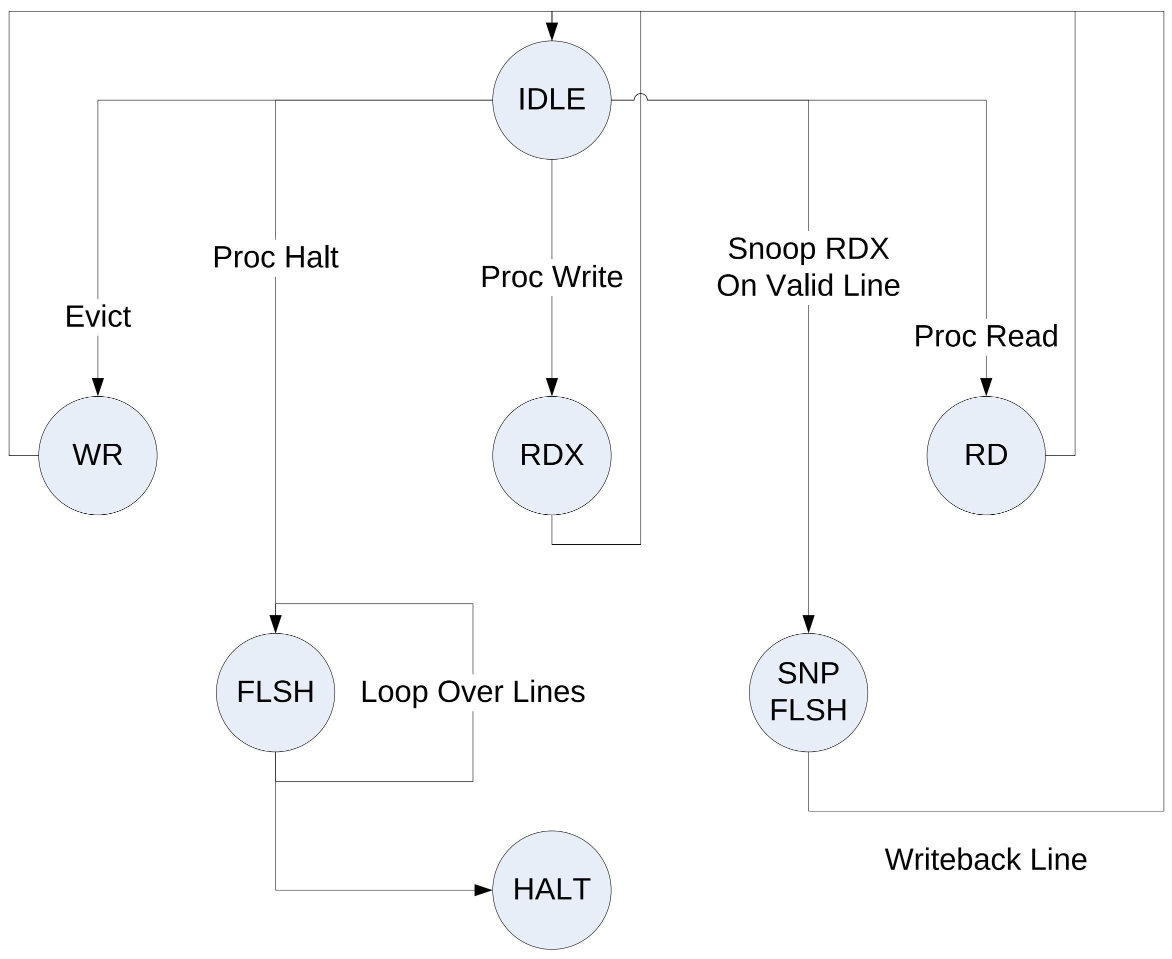
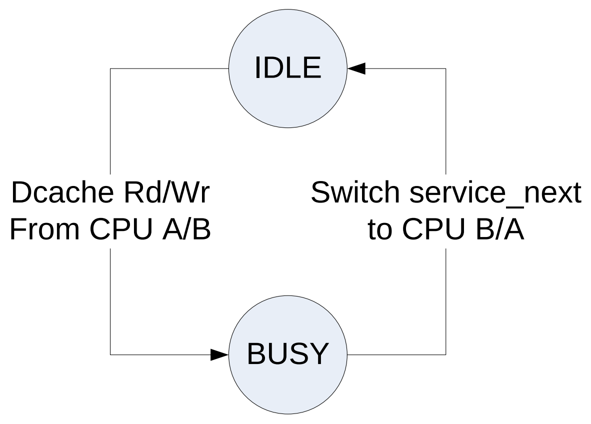


Figure 3: Data Cache State Diagram



Coherency Controller is just another memory arbiter to make sure only one cache does a memory operation at a time. The snooping logic in the caches ensures sharing coherency.

Figure 4: Coherency Controller State Diagram

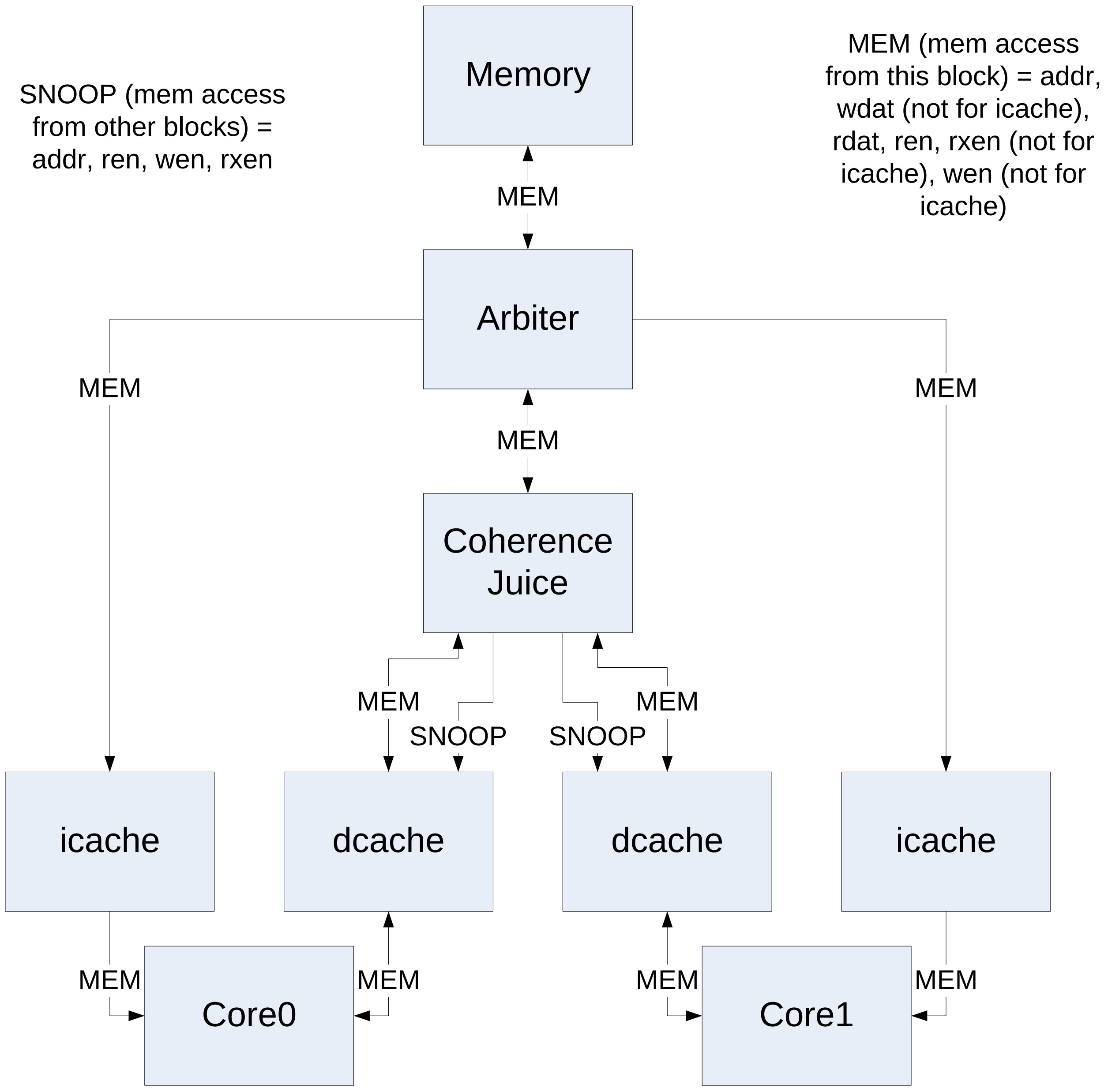

SNOOP (mem access from other blocks) = addr, ren, wen, rxen
Memory
MEM (mem access from this block) = addr, wdat (not for icache), rdat, ren, rxen (not for icache), wen (not for icache)
MEM
Arbiter
MEM
MEM
MEM
Coherence Juice
MEM
SNOOP
SNOOP
MEM
icache
dcache
dcache
icache
MEM
MEM
MEM
MEM
Core0
Core1

Figure 5: Multicore Block Diagram

4 Results

The performance results for the design are displayed in Tables **??**–**??**. The single core design was the simplest implementation of the course's instruction set. The pipeline CPU added pipeline registers to break up the critical path, tripling the speed. The cached, pipelined design added instruction and data caches to the CPU to reduce the average memory access time. It performed excellently in doing so — the CPI was reduced by $\approx 6.4\times$. The dual core, cached, pipelined design was the slowest of them all *but* it has theoretically twice the throughput (ignoring MIPS differences).

For all designs, the critical path is inside the MEM stage. The `vram` module does not have registered outputs so it is a big combinational path. In the cached, pipelined design the critical path was 35.7 ns. This path was through the data cache (during a miss), through `vram`, and terminating at the MEM/WB pipeline register. I expected this path because the combinational path through the huge data caches should be quite long. The dual core, cached, pipeline design had a critical path of 143 ns through the data cache, coherency controller, and terminating at the memory arbiter. I registered the output of `vram` in the memory arbiter to remove the phase shift (due to `vram`'s phase shifted clock) and fortuitously terminated the critical path before it got to `vram`. The fitter didn't have many spare FPGA resources to work with so I believe that it was not able to perform as aggressive speed optimizations as in the other designs. Again, this critical path was expected. The best way to improve the speed would be to rewrite the data cache with an attention to fewer and smaller muxes.

The instruction throughput for the cached designs was vastly ($\approx 6.1\times$) greater than for the uncached design. In the real world this is likely to be even greater because a memory access takes many more than 16 cycles as I simulated. If the application is even remotely typical, it will benefit greatly from a cache. Our moderate 64-word cache was more than adequate for the simple programs we ran but a more realistic application would likely benefit

Table 1: Single-cycle Performance (no memory latency)

f_{max} (MHz)	CPI	Latency (ns)	LEs	Registers
23	1.2	260	3,059	1,231

Table 2: Pipelined Processor Performance (max memory latency)

f_{max} (MHz)	CPI	Latency (ns)	LEs	Registers
75	39	2,600	3,045	1,475

from a larger cache.

5 Conclusion

In this course I have designed a single-cycle, pipelined, cached pipelined, and dual core cached pipelined MIPS-subset CPU. Each design was a trade-off between size, instruction latency, and instruction throughput. The single-cycle design was small yet it yearned for the parallelism that pipelining offers. The pipelined design was an almost total-win — the size was even *smaller* than the single-cycle design while it was about 3× faster. The only downside is increased complexity, especially with tricky data hazards. The cached designs were vastly faster than their non-cached brethren due to large memory latencies. Unfortunately, the data cache was written in an easy-to-develop but inefficient manner, greatly bloating the size of the CPU. The dual core design worked correctly although it nearly filled the FPGA and was rather slow. The benefits of a dual core design could be substantial in an embedded application where low latencies are critical. A dual core design could let a system thread

Table 3: Cached, Pipelined Processor Performance (max memory latency)

f_{max} (MHz)	CPI	Latency (ns)	LEs	Registers
28	6.1	1,090	13,186	5,359

Table 4: Dual Core, Cached, Pipelined Processor Performance (max memory latency)

f_{max} (MHz)	CPI	Latency (ns)	LEs	Registers
7	7.3	5,215	28,128	11,790

handle all realtime events while a worker thread could do long, uninterrupted calculations.