ECE437L Final Report

TA: Eric Villasenor

Nabeel Zaim Steven Ellis

May 2, 2014

1 Executive Summary

We have designed several implementations of a MIPS CPU. Initially, a single-cycle CPU was designed that executes (roughly) a single instruction each clock cycle. That was built into a 5-stage pipelined CPU which allows the overlapping of several instructions in order to increase the clock rate. Afterwards, caches were added to reduce the average memory access time. Finally, the design was converted into a dual-core CPU, implementing a bus coherence controller connecting the cores. All of these designs have been successful.

During the process of creating these processors, many insights into processor design were gained. One of the most imporant of these is the importance of modularity, which is tremendously helpful when incrementally adding new features to a design by enabling large amounts of code reuse, assisted fully by SystemVerilog's parameterization feature. This was incredibly helpful in creating multiple pipeline latches, all of which were internally identical but had different bus widths. Another one was that making things into seperate Verilog modules was necessary but not sufficient for useful modularity. If a module made too many assumptions about the other pieces it is connected to (such as the latency of memory), then it becomes too specific to be easily reusable, largely defeating the point of the modularity. The other key importance of modularity besides code reuse was as an abstraction tool, helping to break a highly complex system into smaller, more managable pieces.

In the rest of the report, the design and verification process of both the caches and the cache coherence implementation will be discussed. There will also be a listing and analysis of the performance of the different designs.

2 Processor Design and Verification

2.1 Cache Design

A 2-way set associative cache was designed to hold 1KB of data. As specified, it was designed as a write-back cache, thereby allowing the regulation of writes and decreasing the chances of saturating memory bandwidth. A 64-byte direct-mapped cache was used as the instruction cache.

The data cache was implemented as a state machine since filling or flushing a multiple-word block required multiple memory cycles (the memory can only supply one word at a time). The state diagram shown in Figure 2 shows how the cache handles cache misses. Within the caches, each block holds an MSI state. It arbitrates with other caches by sending BusRd, BusRdX and BusCache signals onto the bus.

The MSI protocol was tested by running it through a parallel program (similar to *dual.llsc*) that would utilize the same address block. Only a single address was changed to keep it simple - the block in dcache was monitored for state transitions while one core would write and the other would read, and then vice versa.

Verification of dcache involved testing reading and writing. Data incurring cumpulsory misses (tests fetch states), conflict misses with no dirty block, conflict misses with dirty block, working on data with index match but different tag (tests set associativity), flushing.

The common case of dcache use is one where a block is brought into cache and used in the near future. Situations that would degrade the performance would be one such as two cores constantly writing to and reading from a

single address. Each cache would be constantly invalidating the other, thus saturating the bus and allowing few cache hits.

2.2 Multicore Design

The CPU designed utilizes a shared memory model architecture - one where multiple processors work on data in a shared address space, communicating over a single shared bus connecting the cores to each other and to main memory. Figure 5 shows the arrangement of the datapath and cache components of the CPU with respect to the memory.

Important design decisions taking in implementing the coherence controller included adding additional states to the data cache state machine. These included a state for invalidating a block when another cache wrote to it, and three states for writing back all words of a dirty block when another cache wanted it. If a cache had a dirty block and another cache performed a write to it, the former would simply invalidate itself and *not* write the old value to memory. This was done to save bus cycles.

In the process of implementing multiple cores, hardware was added to support write atomicity through **load-link/store-conditional** instructions. The address to work on is stored in a link register, which is a latch with two fields - a 32-bit address field and a 1-bit valid field. The register may be written to or read from but its result would only hold value (i.e. denote a true atomic operation) when paired with a high valid bit.

The bus controller state machine starts out in an idle state, which it remains in as long as there is no activity on the bus. Whenever one or both of the CPU cores initiates a bus transaction, the bus controller transitions

to a state giving one of the cores control over the bus. The arbitration gives control to whoever comes first, and in the case of both cores requesting at the same time, core 0 wins. The next state sends out signals for a snoop operation, where the requesting core will inform the other core if it needs to invalidate a block or perform a cache to cache transfer. After snooping, if the other core is forced to do a writeback by the coherence operation, a new state is entered for the writeback. If the writeback isn't needed or after the writeback completes, a memory access state is entered which allows the cache to access memory for either a read or a write. Once that core indicates that it has completed its memory transaction, the bus controller then returns to idle.

2.3 Diagrams

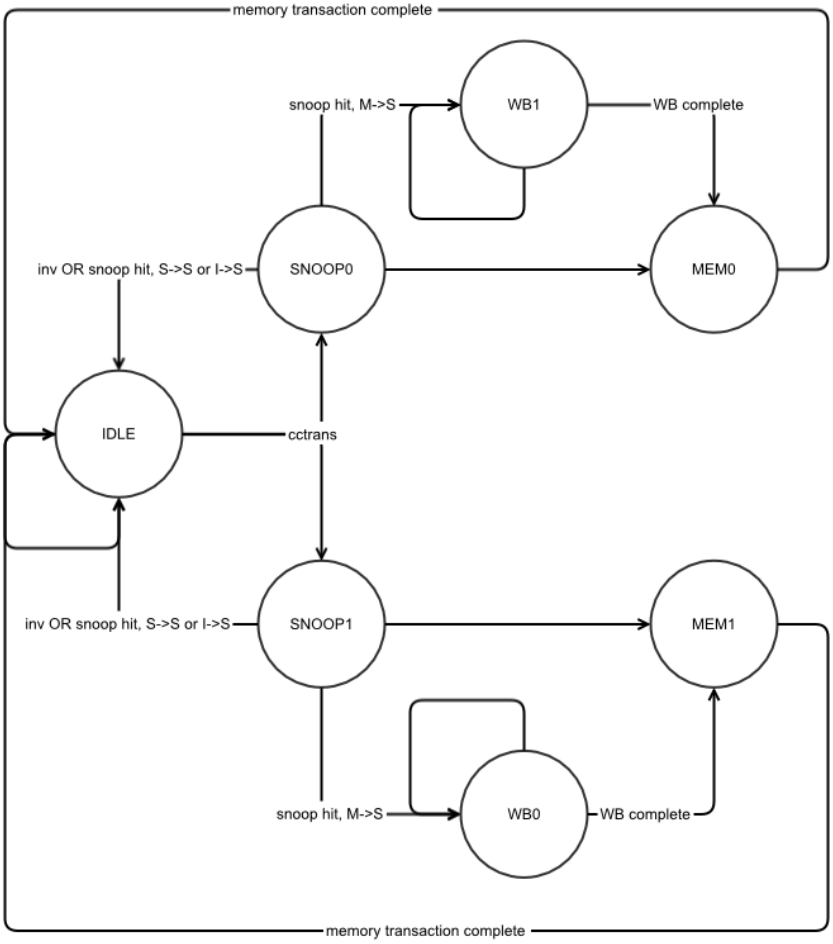

Figure 1: Coherence Controller State Diagram

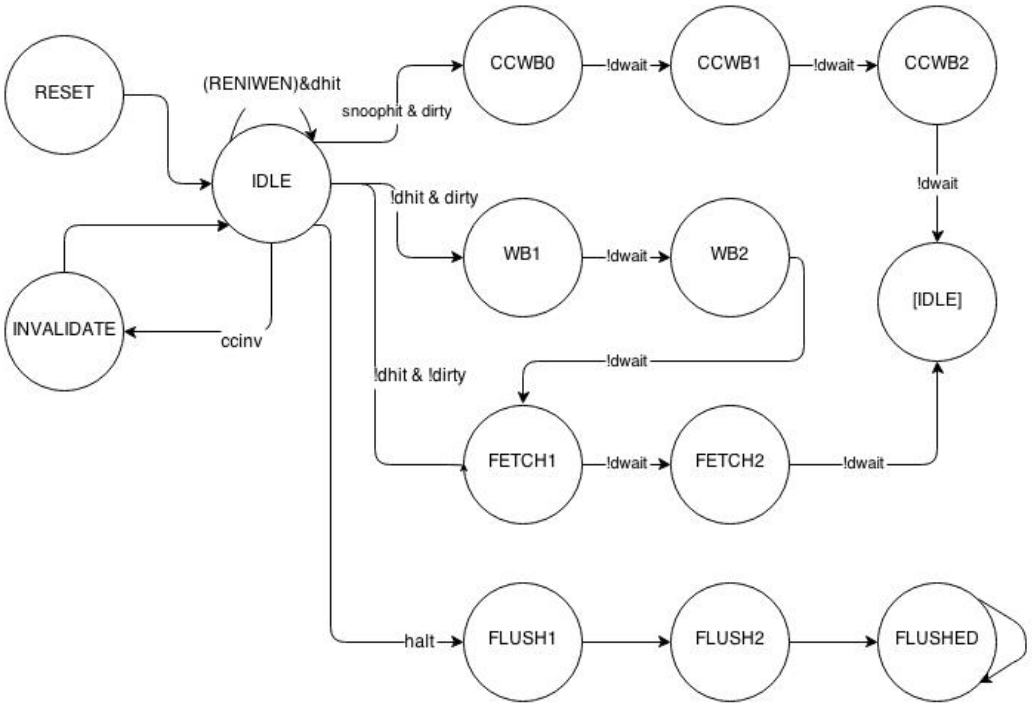

Figure 2: Data Cache State Diagram

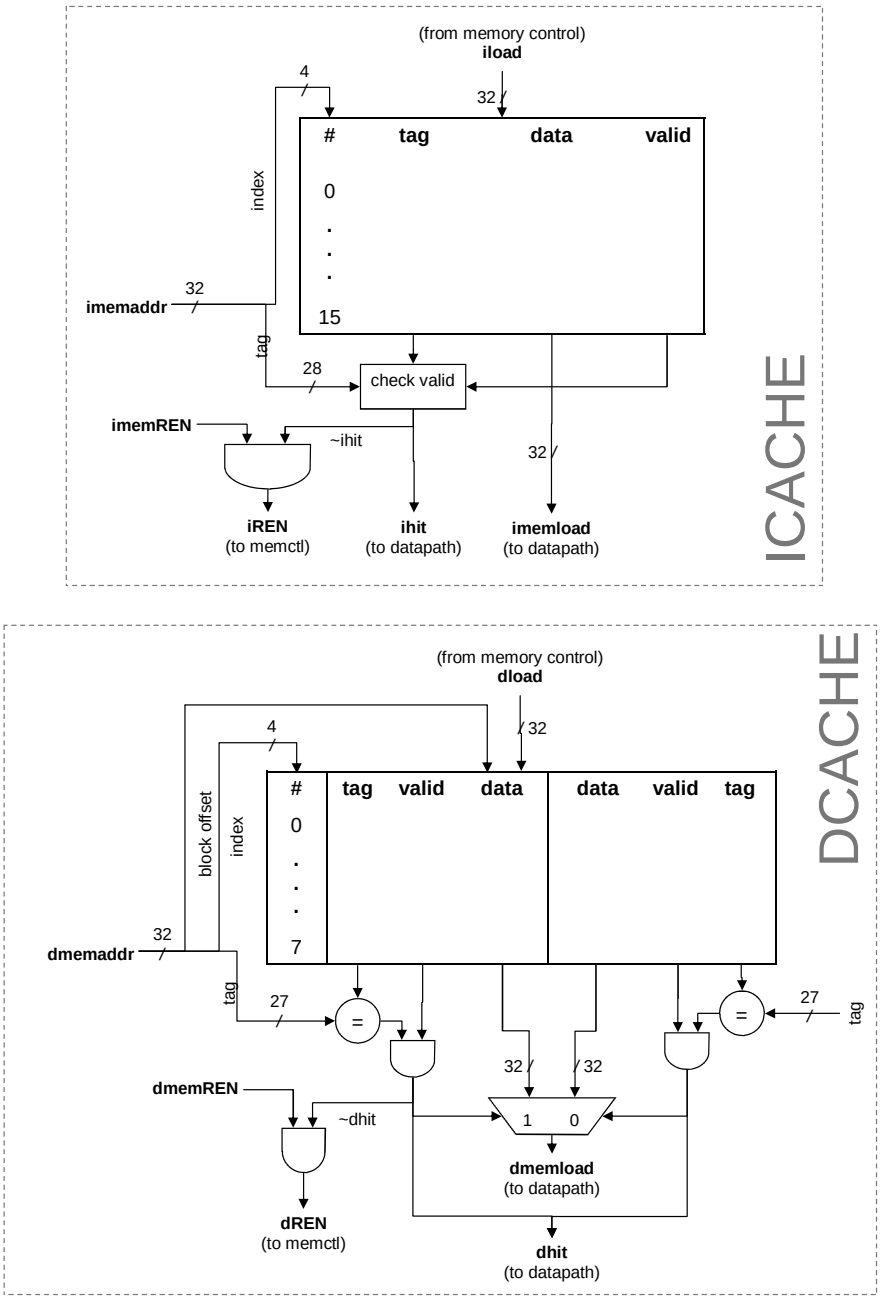

Figure 3: Data Cache Diagram

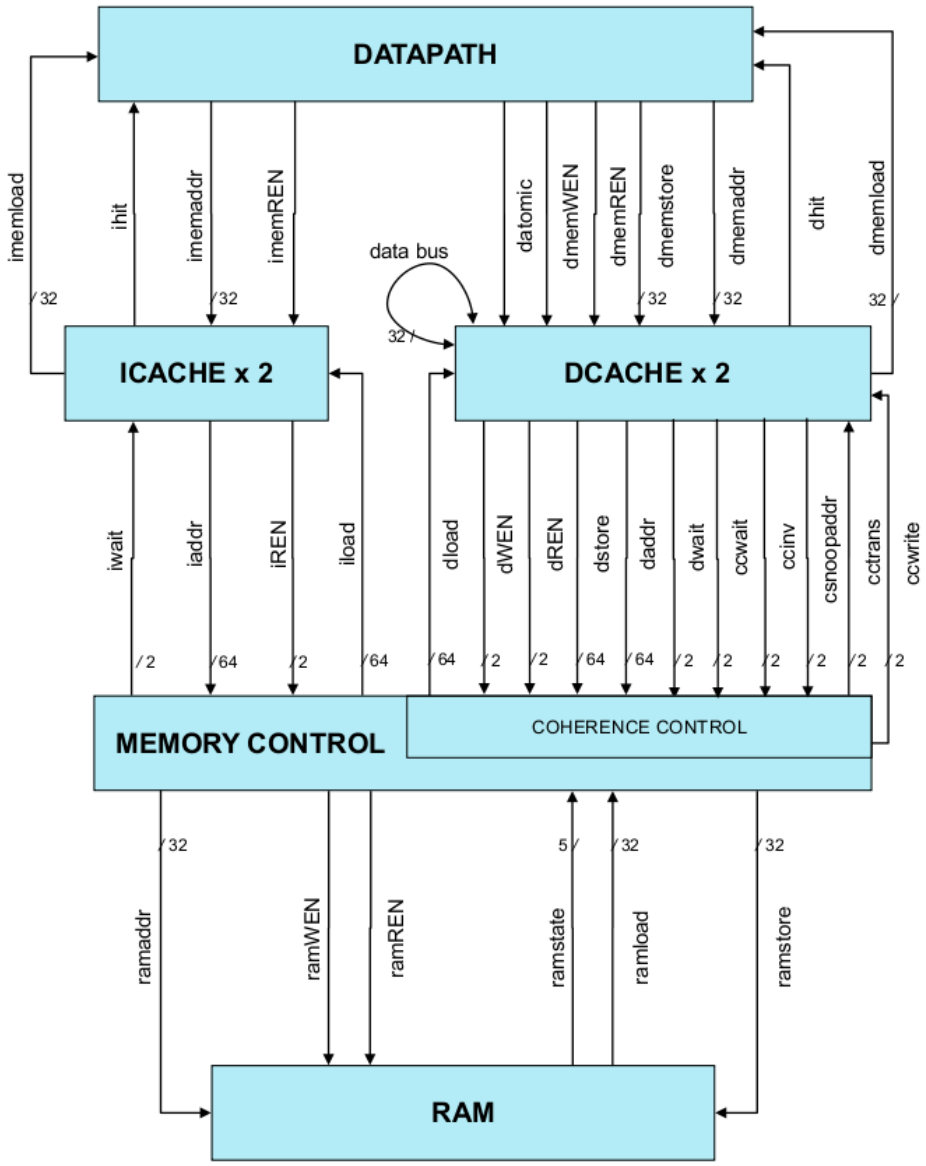

Figure 4: Multicore Block Diagram

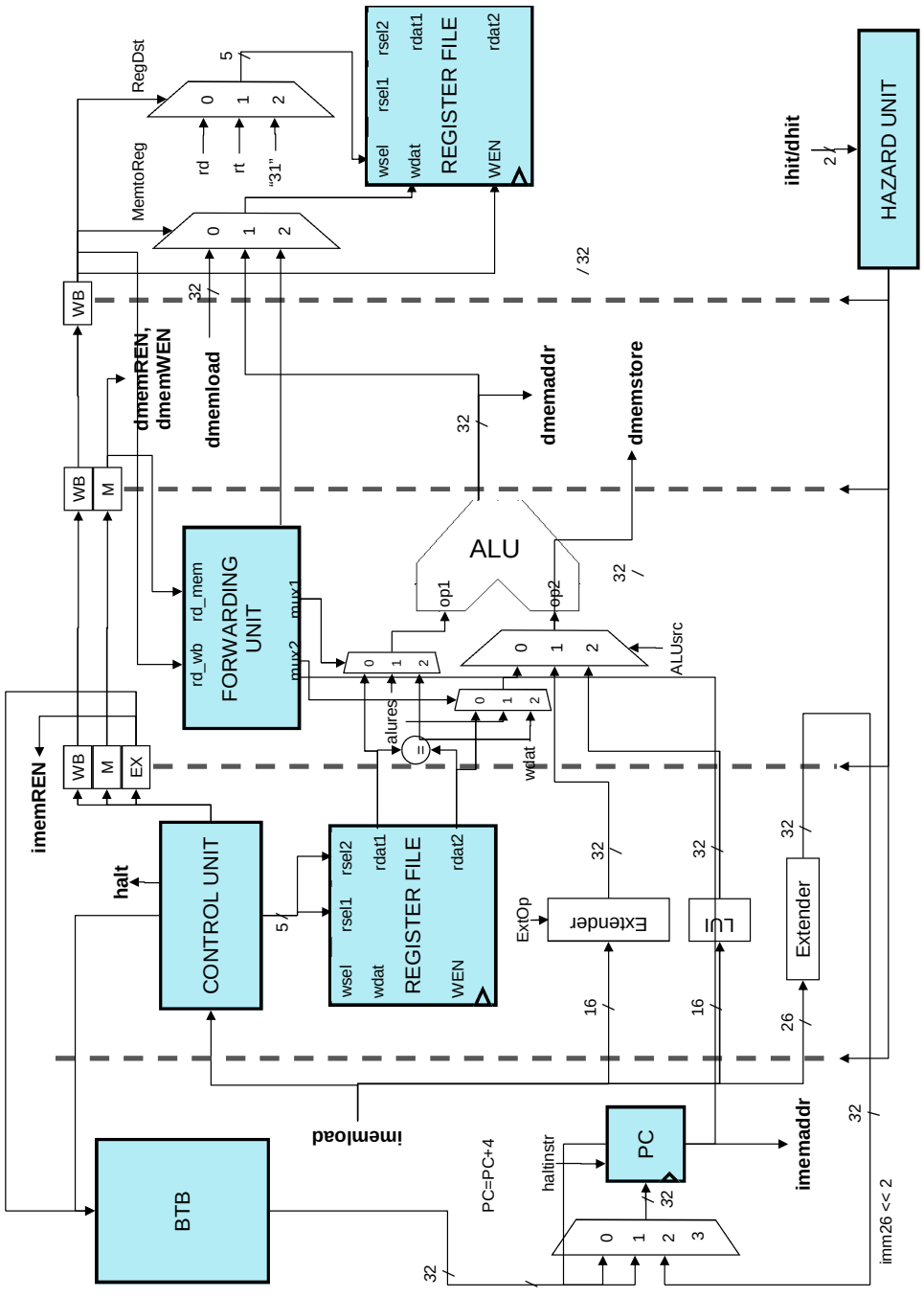

Figure 5: Pipelined Datapath Block Diagram

3 Results

	Singlecycle	Pipeline
Logic Elements Used	3,542	3,518
Registers Used	1252	1672
Minimum Clock Cycle	24.3ns	12.2ns
CPI without cache (1 cycle RAM)	1.4 cycles	1.8 cycles

Table 1: Synthesis results comparison using fibonacci

	Pipeline with cache	Multicore
Logic Elements	9,662	23,310
Registers Used	4135	8221
Minimum Clock Cycle	17.0ns	25.2ns

Table 2: Synthesis results comparison using parallel search

The logic elements and registers used indicate how much of the FPGA is being used, these were taken directly from the Quartus log files. As expected, these increased with each new design's increased complexity. In particular, caches resulted in a huge number of registers because each bit of cache is one. And multicore approximately doubled each, as would be expected.

Minimum clock period gives the total delay through the system's critical path, and was found as the inverse of the max frequency (1200mV 0C Model) given in the log files. The pipeline decreased the length of the critical path, which is expected becaue that is the entire point of pipelining. Caches made the critical path longer, and multicore longer than that.

	Pipeline without cache	Pipeline with cache	Multicore
1 cycle RAM	1.8	2.0	2.3
5 cycle RAM	3.0*	2.8	3.1

Table 3: CPI results using parallel search

** = estimated result for non-working design*

For the CPI numbers, the parallel search program (provided in Appendix A) was run on each of the CPUs. To find the number of instructions, it was run in the MIPS simulator which reports the instruction count when it finishes executing, which was 160 instructions. Then when simulating in hardware, the testbench prints out the number of clock cycles. Since the CPU was running off of a clock divider at half the system clock frequency, the number of CPU cycles taken was half of what the testbench reported. Then the CPI calculation was a direct application of the definition, number of cycles/number of instructions. For the singlecycle, each instruction took 1 cycle unless it was a LW/SW, which took 2. With 40 percent of dynamic instructions in fib.asm being LW/SW, the CPI would be expected to be 1.4 which it was. Then the CPI got progressively worse in each implemtation, with 1 cycle RAM latency. First the pipeline added stalls, then the cache added miss penalties, then the multicore added coherence operations, all of which increases the CPI. However, with a higher RAM latency the cache decreates average memory access time and achieves better CPI, but coherence still brings it back down somewhat.

In the multicore CPU, the critical path is from the RAM address to the memory/writeback latch in the CPU. This implies that the critical path is

through the coherence controller and the data cache, which makes a lot of sense. First of all, the fact that it was slower than the single core pipeline with cache immediately points to the coherence controller and/or data caches being the critical path, because they are the only parts that changed in the multicore design, aside from some small additions to the datapath to add the LL and SC instructions. Also, the datapath of each CPU is pipelined to reduce its critical path but the coherence bus controller is a simple single cycle implentation. Obviously without pipelining we should expect it to have a fairly slow maximum clock.

The cache hierarchy influences the instruction throughput because it determines the average memory access time. Because the cache in this design is quite small and with low associativity, the hit rate, hit penalty, and miss penalty will all be low.

The design was tested with a parallel search algorithm to measure differences in latency and throughput between the pipeline and the dual-core. The relevant portion of the test program can be found in **Appendix A**.

The program worked as expected for clock periods between **20ns** and **100ns**. It was tested on lower clock periods - **10ns**, **5ns**, and **2ns** - and the design did not execute anything beyond its first instruction cycle. The reason for this can be speculated to be the lack of delay time given for the combinational logic blocks between the pipeline latches, such that the clock signal would fall before the logic had time to complete its execution. Another reason could be the lack of delay time between the data cache and the *dmemload* port of the datapath, which is among the paths of our design with highest slack. However, this was not an issue in the test program.

4 Conclusion

The single cycle processor worked. The pipelined implementation worked, but broke with increased memory latency. The issues causing it to fail were remedied in the version with caches attached. The cached pipelined CPU worked for a reasonably wide range of memory latency (1 to 10 clock cycles). The dual core processor with cache coherence worked equally alike. While the singlecycle processor had the advantage of a low CPI, it suffered from a low clockspeed. The pipeline allowed us to speed up the clock, while paying with CPI, thus gaining some performance not withstanding a dramatically higher power consumption. Caches brought both the CPI and clock down, while taking up substantially more area. The multicore added the ability to run 2 programs simultaneously to potentially double the performance, but lost a large fraction of that performance gain to a combination of lower clock speed and higher CPI and, expectedly, doubled area and power consumption. The multicore design, however, could not be mapped onto an FPGA due to synthesis warnings discovered too late. This taught us how modelsim had synthesized our design leading to unintentional bidirectional ports being created. These went unnoticed in all synthesis tests but arose while trying to map to the DE-II. Other important lessons learned were about abstraction and the use of interfaces, tasks and parameters for elegant and clean Verilog code.

5 Appendix A - code listing

5.1 Parallel Search

```
#-------------------------------------------------------------------------
# Test a search algorithm in parallel:
# core1 searches half the list and core2 searches the latter half
# They store a found flag that is a shared address between them.
#-------------------------------------------------------------------------

/************************
 *     PROCESSOR 1
 ************************/
  org   0x0000
  ori   $sp, $zero, 0x80
startp1:
  ori   $1, $zero, 0x01
  ori   $2, $zero, 0x04

  sw    $0, 0($sp)
  lw    $3, 4($sp)
  lw    $4, 8($sp)
  addiu $5, $sp, 12

loopp1:
  lw    $6, 0($5)
  subu  $7, $6, $3
  beq   $7, $zero, foundp1
  addu  $5, $5, $2
  subu  $4, $4, $1
  beq   $4, $zero, notfoundp1
  beq   $0, $zero, loopp1
foundp1:
  sw    $5, 0($sp)

notfoundp1:
  halt
```

```
/************************
 *     PROCESSOR 2
 ************************/
  org   0x0200
  ori   $sp, $zero, 0x80
startp2:
  ori   $1, $zero, 0x01
  ori   $2, $zero, 0x04

  sw    $0, 0($sp)  # both storing 0
  lw    $3, 4($sp)
  lw    $4, 8($sp)
  ori   $5, $0, p2list

loopp2:
  lw    $6, 0($5)   # load from search ptr
  subu  $7, $6, $3
  beq   $7, $zero, foundp2
  addu  $5, $5, $2
  subu  $4, $4, $1
  beq   $4, $zero, notfoundp2
  beq   $0, $zero, loopp2
foundp2:
  sw    $5, 0($sp)  # only one of the
                    # cores will store
notfoundp2:
  halt
```