CS 3220 - Project 4

Will Gulian **Yaotian Feng**

1 Overall Design

We designed a high performance processor implementing the CS 3220 ISA, taking ideas from past processors we've jointly designed. Our project 3 design completed fmedian2 in 35.4 s (93.7 s at 50MHz). Through a variety of optimizations, we were able nearly 2x (6x from 50MHz) performance with a new execution time of 15.3 s.

2 Performance Data

Optimization	No BTB	With BTB
Project 3	136.28MHz	116.96MHz
Project 4	150.02MHz	154.68MHz

Table 1: Fmax Matrix.

Optimization	No BTB	With BTB
Project 3 (no opt)	93.7s ($f = 50$)	83.5s ($f = 50$)
Project 3 (opt)	35.4s ($f = 133$)	37.4s ($f = 112$)
Project 4 (opt)	23.9s ($f = 150$)	**15.3s** ($f = 150$)

Table 2: fmedian2 execution time with corresponding clock frequency f (in MHz). 'opt' refers to results obtained after timing optimizations performed prior to submission for Project 3 and Project 4.

3 Branch Prediction

The first optimization we made was to implement a branch target buffer and branch predictor in our fetch stage. We use a 2-bit predictor and our branch target buffer is 256 entry direct mapped, from address$[9 : 2]$. This gives us optimal buffer efficiency such that tight loops like $SortDesc$ and $SortAsc$ will always fit in the BTB.

4 Register Forwarding

We implemented register forwarding in Project 3, improving MIPS at a slight cost to maximum clock frequency. All of our performance data is based on our processor performing register forwarding.

5 Timing Optimization

The first optimization we made (in lab 3) involved fixing a path where a conditional branch in the ALU would load a new value into the fetch buffer and retrieve the instruction by the new PC, in one cycle. Fixing this improved our Fmax from around 100 MHz to around 130 MHz, and this is the biggest improvement to the maximum frequency we've made.

The second big timing optimization we made was pipelining the execute stage to two total cycles (results are clocked into buffer on second cycle). This was not a simple change but it improved our Fmax to around 150 MHz. The reason is that certain operations in the ALU, mainly shift and comparisons, were significant contributors to our worst-case path timing delay. Since an instruction in execute is not complete until the second stage, instructions with a direct data dependency must wait one cycle. Fortunately this happens very rarely ($< 1\%$) so the frequency increase it allows leads to a significant improvement in execution time. We considered pipelining to 3 cycles as well but simulation data showed that this would cause significantly more stalls and may negate any improvement.

We made other optimizations as well such as using the opcode grouping to more quickly deduce instruction type, determining an inferred halt[1] one cycle later, and using an additional "forward valid" signal rather than comparing against zero in our register forwarding logic.

6 Memory Optimization

When we were originally designing our processor, we decided to use the Wishbone bus[2] to communicate with memory and IO. This allows us to easily add new memory mapped devices but incurs a cost since the memory stage must communicate over the Wishbone bus and will always stall multiple cycles due to this.

Performing some dynamic execution profiling, we determined that over 50% of the time for fmedian2 was spent on memory accesses. This makes sense because the majority of the workload is a memory IO heavy sorting algorithm. For performance, we moved the data memory from a device on the Wishbone bus to a special case

[1]Our inferred halt logic detects an unconditional jump that jumps to itself, since this is done at the end of fmedian2. This allows our performance counters to accurately stop when the test is done.

[2]https://opencores.org/cdn/downloads/wbspec_b4.pdf

within the memory stage (almost like a large L1 cache). This allows us to always perform a memory read and write in 2 and 1 cycles, respectively. IO operations still go over the Wishbone bus.

This change brought our execution time from around 30 s with just branch prediction and timing optimizations to 15.3 s.

7 Failed Optimizations

We tried other optimizations as well, but ended up removing them either because they didn't help or actually hurt our performance.

The way our IMEM is implemented, it cannot be inferred into M10K blocks due to an asynchronous read in our branch predictor. I fixed this issue so that Quartus could infer ROM, unfortunately, Quartus creates a full-size ROM for the 64k memory size out of M10K blocks and this ended up being much slower due to interconnect delays to all the M10K blocks. The original code forced Quartus to synthesize the IMEM into a large combinational lookup, but Quartus is capable of reducing this combinational logic so that our tests, which are very small, take up very little chip area. Since this change ended up hurting us significantly, we reverted it.

Before pipelining execute stage, we tried making the execute stage stall for one cycle on shifts so that shifts take two cycles. This worked but did not improve our worst case path much because comparisons had a similar delay. Since a large part of the workload is conditional jumps, we scrapped this idea and later implemented pipelining.

8 Future Optimizations

If we had more time, I would implement dynamic predication. This is where jumps that jump forward a small distance (less than 3 instructions) and have bad predictability are rewritten during the decode phase (along with the following instructions that may be jumped over) into a predication group. This way some branches that are hard to predict can be elided for better performance. A prime example of this is the compare and conditionally swap sequence in *SortAsc*. It is hard to predict because the data is not sorted so we will incur a branch miss around 50% of the time. I believe the mispredict cost for our processor is between 4 and 6 cycles so predicating the 2 instructions should save around 2-3 cycles per loop.

Although 2-3 cycles per loop is not a large amount, the loop is very hot so this would most likely make a noticeable difference in execution time.

9 Contribution

Yaotian did all of the branch prediction work, and I did most of the optimization work. Our contributions are about 50/50 each.