L-RU Ready ECE 411 Final Report

Advaith Bala, Jean-Luc Charlier, Varun Singhal

The ECE 411 Final project consists of building a RISC-V processor using SystemVerilog. RISC-V is an instruction set architecture (ISA) designed around a Reduced Instruction Set Computer model, where instructions are short, fixed-length, and perform only simple operations. In addition, only the basic RISC-V 32I Instruction set is required, which only enables addition, shifting, and comparison on 32-bit integer types (as well as control flow, and memory operations). The Final Project aims to build a multi-cycle pipelined processor, an associated cache, and several other advanced features, which are up to the choice of the group.

Our Final Project implements the pipelined processor, one low-level cache, and advanced features consisting of:

- The RISC-V M extension, enabling multiplication, division, and remainder operations in hardware,
- A parameterized cache, enabling greater configurability and compatibility with different design specifications,
- A write-back write-merging cache, which reduces memory overhead by minimally writing to it,
- A Return-Address Stack (RAS), which stores information from certain jump instructions to enable faster jumping in specific circumstances,
- A Branch-Table Buffer (BTB), which stores information from branch instructions to enable faster branching, combined with
- A Branch History Table, which stores information about whether recent branches were taken, to enable basic branch prediction.

Besides the implementation of various features, a principal motivator is to ensure good performance for the processor across several metrics. These include processor frequency, power usage, and delay time, as measured on a benchmark program. The design of the processor was spread across four checkpoints over the course of the assignment.

Checkpoint 1 required the implementation of basic pipelining. Neither detection nor handling of data or control hazards was required. In addition, no synthesizable cache was required. Instead, a “magic memory,” which always completed all operations in one cycle. This way, no stalling from any operations would be required. The pipeline in this stage, which served as the basis of the processor in all subsequent checkpoint designs, consisted of 5 stages: Fetch, Decode, Execute, Memory, and Write-Back. The Fetch stage,

worked on by Jean-Luc, involves maintaining the program counter (PC), which is the address of the current instruction, and also sending that PC to the memory to actually fetch the instruction word from memory. The Decode stage, worked on by Varun, consists of decoding the bits in the instruction word in accordance with the RISC-V spec, and turning it into a set of signals to be used by the processor in later stages to dictate the details of execution. The Execute stage, worked on by Jean-Luc, consists of an ALU, which can perform addition, subtraction, logical left and right shift, arithmetic right shift, and signed and unsigned comparisons. The Execute stage is also responsible for calculating branch destinations and conditions, and signaling the Fetch stage to update the PC as required. The Execute stage also reads from the register file ("regfile"), which is where register values are stored. The Memory stage, worked on by Advaith, is responsible for memory operations – loading to, or storing from, a register. In Checkpoint 1, due to the magic memory, this stage cannot stall, but its complexity increases in later checkpoints due to stalling and data hazard conditions. Finally, the Write-Back, worked on by Advaith, stage involves writing to the regfile when required by operations. This stage is the last in the pipeline and, as such, is also responsible for providing signals to the testing apparatus for the processor when instructions are completed. The testing apparatus consists of a RISC-V Formal Interface (RVFI) model for comparison of processor state to a verified model, and also of the Spike RISC-V simulator, which simulates memory better than RVFI. Testing for this checkpoint involved running programs which had no hazards, and verifying against RVFI and Spike.

Checkpoint 2 required the implementation of a proper cache system, divided into and Instruction cache (I-cache) and Data cache (D-cache), as well as handling of control and data hazards. The I-cache is read-only, and feeds the Fetch stage, while the D-cache enables reading and writing for the Memory stage. The cache system is required to be synthesizable, which imposes limitation on capacity and response time. Our group elected to make a set of small caches which could respond to read/write hits combinationally. This simplified logic for the processor pipeline, as decisions to stall or not could be made the same cycle as a request. In the case of a cache miss, however, the cache would have to query main memory, which would take several cycles and force the processor to stall stages until data was ready. Handling data hazards required implementing forwarding logic. If an instruction read from a register within 2 instructions of another which wrote to it, then the data would not be written to the regfile before the instruction would read the regfile in the Execute stage. Thus, the register's data would have to be "forwarded" directly from an earlier instruction further along the pipeline. Finally, control hazards, which arise from speculative execution of instructions behind a branch instruction which is mispredicted, require flushing certain stages of the pipeline so that those instructions are

not executed. In Checkpoint 2, no branch prediction is implemented (which is equivalent to a static “not taken” prediction). The testing for this checkpoint is successfully running the Coremark RISC-V 32I benchmark, as all features required for it should be implemented.

Checkpoint 3 required the implementation of the advanced design features chosen by our group. First, the M-extension requires hardware multiplication and division algorithms. Worked on by Jean-Luc, our design only uses basic long multiplication and division algorithms. The extension requires the ability to calculate the high and low 32 bits of a multiplication, which supports signed-signed, signed-unsigned, and unsigned-unsigned products, and the ability to calculate the quotient and remainder of signed and unsigned divisions. To save on hardware area, both multiplication and division extend to 33-bit operands as appropriate for signedness. Next, the parameterization of the cache, worked on by Varun, involves using SystemVerilog “genvars” and “generate” statements to have compile-time configurability of cache parameters, like the number of sets or data ways, by changing parameters in the instantiation of the cache module. Varun also worked on making the cache write-back with write merging. Write-back is a cache protocol which ensures validity of data between the cache and main memory by maintaining information about which cache lines have been written to (which lines are “dirty”) and writing to main memory only when such a line needs to be evicted from the cache. Next, the RAS, worked on by Jean-Luc, is a hardware stack which stores PC values from jump-and-link (jal) and jump-and-link-return (jalr) instructions. These instructions are best used for function calls, using specific registers – the x1 (link) and x5 (alternate link) registers – which store the PC address a jump occurs at in a jal instruction, and the PC address used to jump back to in a jalr instruction. By keeping these values in a stack (mimicking the call stack), the processor can determine return addresses earlier in the pipeline, minimizing the burden of a pipeline flush. Then, the BTB, worked on by Advaith, stores the PC addresses of previously taken branches in the program. This essentially functions as a miniature cache just for branch targets. Like a cache, capacity is limited, and like our cache design the BTB uses a pseudo-least-recently-used (PLRU) algorithm to determine the best branch addresses to evict. In this way, more frequently used branches, like those in common loop constructs, are retained in the BTB. Finally, the BHT, implemented by Advaith, stores a tag for the last several PC addresses for taken branch instructions. This allows basic branch predictions: If a PC address matches that in the BHT, then the branch is predicted taken and the next PC is retrieved from the BTB. Specifics for testing for this checkpoint varies based on the feature, but in general features can be verified as standalone modules, then integrated into the processor design and verified by running a Coremark benchmark and other testing programs.

The last checkpoint, Checkpoint 4, involves increasing the performance of the processor. The primary metric is the processor's frequency. The limit on the frequency is the length of the longest combinational path in the processor. Thus, reducing this path is the best, most straightforward way to improve frequency. Our longest path from Checkpoint 3 began in the Write-Back stage, passed through the large mux there to select the regfile input, then went through the Execute stage's forwarding logic, and finally into the divider, and through some set-up logic there before ending up in a register. Reducing this path involved moving some of the mux logic to the Memory stage, and moving some of the forwarding checking logic to the Decode stage. In total, we were able to achieve a frequency of 714.29 MHz, which was the 3rd highest among the ECE 411 groups. The next metric was the processor's power consumption. Power consumption chiefly came from register usage, which requires a static power to store bits, and from the dynamic switching power involved used by the simulated transistors. Our power usage was 14.8468 mW, which was among the highest in the competition. The final metric was the delay of our processor, which depended both on the frequency, and the underlying efficiency of the processor e.g. stall frequency, multiplication speed, etc. Our delay was 1189867 ns. This was low, and so in the overall competition rankings, by either power-delay^2 or power-delay^3, we placed 4th overall.

Ultimately, our design worked well within the bounds of the project. We were able to complete all of our objectives. Our main struggle was with Checkpoint 2, which took longer to complete than the allotted time. Starting earlier is a good lesson from this, as would be having more concrete design plans in place before beginning work – a reason we struggled with Checkpoint 2 was because we made an ad hoc cache design without considering all the features we might have wanted ahead of time. However, we were still able to place well in the final competition, because we committed to making up the issues that we had with Checkpoint 2.