6.375 Final Project Report: RISC-V processor with Floating-Point instructions

Kathy Camenzind and Miguel Gomez
Mentor: Andy Wright

1. Objective

The current RISC-V processor that we've worked on in 6.375 uses the base integer instruction set. However, for many applications it is useful to perform floating-point operations, which is supported by various extensions to the RISC-V instruction set. In this project, we'd like to explore extending the current RISC-V processor to support the Single-Precision Floating-Point "F" standard extension. While it is possible to handle floating-point operations through software emulation, the goal of this project is to actually add an FPU to the processor that can handle the instructions as part of the processor pipeline.

There are several considerations when adding an FPU to the processor that will add considerable complexity. First, we will need to implement changes to the Decoder to correctly interpret this extended set of instructions and store information about whether operations are integer or floating-point. The development of a new register file will also be necessary, as it requires a separate set of floating point registers, as well as a third read port.

The primary changes, however, will be made by adding the FPU to the pipeline. This FPU will provide various functional units that perform math, comparisons, and other operations on floating-point values. We must determine the path of execution of the new floating-point instructions as well as the old integer instructions in the pipeline using the FPU.

Since floating-point operations are more computationally intensive than integer operations, the floating point computations that will be done in the FPU should be pipelined in order to reduce the combinational delay of the processor. This introduces the requirement that our processor pipeline now has to handle potentially multi-cycle execution of instructions. If the processor is scalar and in-order, then this will involve a relatively simple design that includes incorporating stall signals based on the type of the executing instruction. However, to increase our processor efficiency, we also plan to look into out of order implementations.

A basic out of order implementation scheme, which we will describe in further detail later, would be to allow for the dispatching of an integer instruction while the FPU is busy, and allowing for our FPU to handle calculating multiple floating-point operations simultaneously, given that the functional unit required is available. This will require scoreboard checks to ensure that there are no data conflicts between executing elements, and handling older instructions

finishing before newer ones if they are lower-latency operations. Reordering of these instructions that complete out of order will be explored.

Through this project, we hope to further explore more complex RISC-V architectures and gain an understanding of more sophisticated pipelining techniques, while keeping under consideration data and control hazards that might arise from out-of-order implementations of the processor.

2. High-Level Design

The high-level design of the processor will work off of a 4-stage RISC-V pipeline, similar to the 3-stage pipeline that we developed in Lab 5 with a bypassing register file. It will have stages for instruction fetch, decode, execute, and writeback, as well as a scoreboard to detect data hazards. For simplicity, and since they are not the focus of the project, we will not have data bypassing (other than the bypassing register file) or branch prediction. The increased number of stages as opposed to Lab 5 cuts down on the latency of each stage, which should allow us to run the processor at a higher clock speed. Most stages, other than instruction fetch, will have to be modified in order to incorporate floating point instructions, as described below and shown in Figure 1.

- **Decode***:* The decoder reads an instruction and converts it into information in a more usable format for the rest of the pipeline to handle. With the addition of floating point instructions, we will have to implement additional paths in the Decoder for these new instruction types, to teach it to handle their encodings. The data that is passed onto the next stage will need to be altered to include information about whether the operation uses floating-point registers, what floating-point operation it uses, etc.

  Decode will also read values from the register file. The RISC-V floating-point extension uses 32 additional 32-bit registers for floating-point operations, so we'll need to include a second register file for these floating-point registers. The register read stage will then have to distinguish between which of the two to read from by using a single additional bit, and insert this augmented register information into the scoreboard. Based on the data hazard present in the scoreboard and whether the corresponding execute pipeline is busy (described below), we will potentially stall the pipeline at this stage.

- **Execute***:* The execute stage will be broken into two parts: a single-cycle ALU for all operations that write back to the integer register file, and a multi-cycle FPU (with multiple multi-cycle units for different FPU operation types) that can complete floating-point operations out of order, and reorders operations before writing back to the floating-point register file. Each individual stage will be pipelined and operate in parallel.

They will be implemented with a get/put interface, so that traffic further down the pipeline will simply stall earlier in the pipeline, and data will not be lost.

The reordering of completed floating-point instructions will be done using a completion buffer, that is pushed to in order that instructions are dispatched, and are popped in the same order. When instructions complete out of order, the completion buffer will store the result until all earlier instructions complete and have been popped in order.

- **Writeback**: Writeback is largely the same as before, with the exception of having to decide which register file to write back to, either integer or floating point, depending on the instruction type.

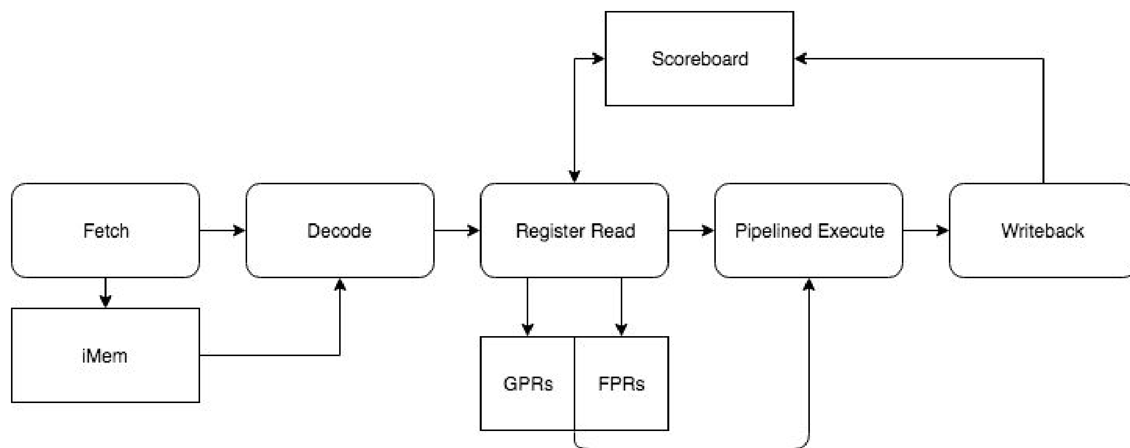

Figure 1: Instruction Pipeline for the processor with a multicycle FPU for floating-point operations.

There are several intermediates steps that we have written on the way to our eventual pipelined, out of order floating point processor. First, we ensure that our floating point library is functional by implementing the FPU as a single-cycle operation, and integrated it directly into the 3-stage processor from Lab 5 alongside the ALU. We additionally develop a

3. Testing Framework

The testing framework that we will be using is mostly the same as the one that was provided for the use of lab 5, with some key differences that will allow us to test it with the floating-point ISA. The basic structure of the test bench will remain the same as that of the lab; a Connectal wrapper around the processor in order to send and receive data using the Connectal main.cpp program. The instructions and data will be compiled to the riscv binary format using the Makefile, which will then be loaded onto the memory by main.cpp using the memInit method.

The processor will be started by using the hostToCPU method, after which the program will run until completion, and the final state will be returned with cpuToHost.

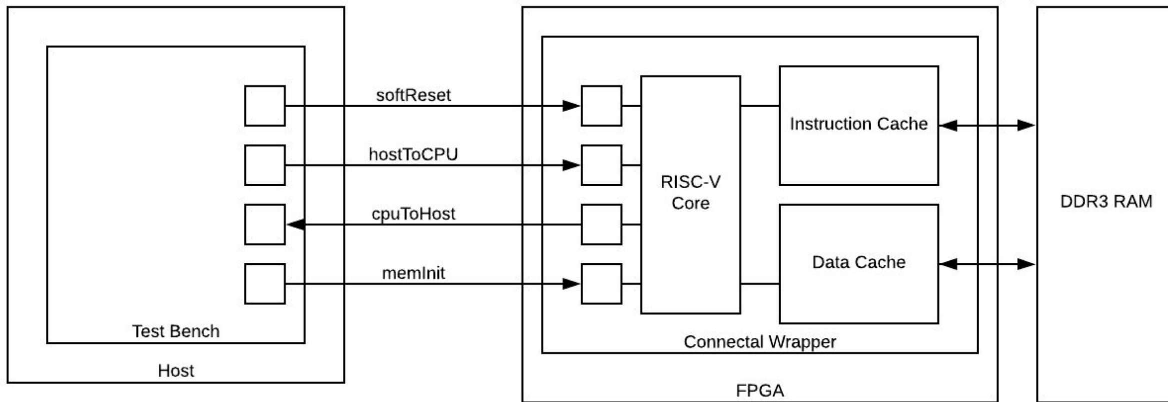

Figure 2: General Structure of Testing Framework with FPGA

To comprehensively test the processor, we will be using a combination of existing tests (which include most of the base RISC-V integer instructions) and new tests that we create, which will include all of the new floating-point instructions. To modify the instructions to function properly with the new floating-point architecture, all that is needed is to change the test configuration from RVTEST_RV32U to RVTEST_32UF, which will allow the use of all the new floating-point instructions in our test program. To aid us in debugging our design, apart from using the new floating-point instructions we will also use the test macros provided in "test_macros.h" to verify correct functionality. Table 1, listed below, includes a comprehensive list of all instructions that must be tested to ensure the correct implementation of the RISC-V Floating-Point processor.

FLW, FSW	Loads/Stores Floating-point data to/from rd
FMADD.S, FMSUB.S	Multiplies rs1, rs2, adds rs3, stores in rd
FNMADD.S, FNMSUB.S	Multiplies rs1, rs2, negates, adds rs3, stores in rd
FADD.S, FSUB.S, FMUL.S, FDIV.S	Adds/Subtracts/Multiplies/Divides rs1, rs2, stores in rd
FSQRT.S	Computes square root of rs1, stores in rd
FSGNJ.S, FSGNJN.S, FSGNJX.S	Takes all bits from rs1 except sign bit, which is determined by the sign of rs2, the opposite sign of rs2, or XOR of signs of rs1 and rs2, stores in rd
FMIN.S, FMAX.S	Takes min/max of rs1 and rs2, stores in rd
FCVT.W.S, FCVT.WU.S	Converts floating-point rs1 value to signed/unsigned integer value, stores in rd

FMV.X.W, FMV.W.X	Moves floating point value from rs1 to lower 32 bits of integer register rd, or vice versa
FEQ.S, FLT.S, FLE.S	Equality/Less than/Less than or equal to of rs1, rs2, stores in rd
FCLASS.S	Examines value in rs1, stores 10-bit mask in rd that indicates class of floating-point number
FCVT.S.W, FCVT.S.WU	Converts signed/unsigned rs1 value to floating-point value, stores in rd

Table 1: Descriptions of Floating-Point Instructions

4. Microarchitectural Description

In our pipeline, we will introduce several new modules. The first class of modules is adding the Floating Point execute units, that perform the new floating point operations. These are already implemented as multi-cycle units in built-in Bluespec library, in FloatingPoint.bsv. They have Server (request/response) interfaces, have inputs of 1-3 floating point operands and the rounding mode, and have outputs of the floating point result and any exceptions. We plan to use the following modules:

- **mkFloatingPointAdder**: Adds two floating point numbers. Takes 5 cycles.
- **mkFloatingPointMultiplier**: Multiplies two floating point numbers. Takes 5 cycles.
- **mkFloatingPointDivider**: Divides two floating point numbers in 5 cycles.
- **mkFloatingPointSquareRooter**: Takes the square root of a floating point number in 5 cycles. Only takes one floating point operand.
- **mkFloatingPointFusedMultiplyAccumulate**: Multiplies two operands and adds a third. Takes 9 cycles to complete, and uses 3 floating point operands.

The second class of modules that we plan to use are the already-existing modules from the processor implemented in class, that we will either instantiate or modify slightly. Any changes made are to support floating point registers and allow for out of order execution. The modules we will use and/or modify are as follows:

- **mkBypassingRFile**: The register file, which previously has two read ports and one write port, will now be used twice for the floating-point register file. To support the multiply-accumulate function, we must add a 3rd read port. Since we currently plan on

only implementing single-precision, there are no other changes to the register file, although if we were to extend to double precision, we could parameterize the data size.

- **mkCsrFile**: Modified to include the addresses to read FCSR and its fields. Since FCSR has to be read within the Decode stage to determine the rounding mode, we need to be able to handle 2 CSR reads in a cycle. Instead of implementing a second read port, we observed that one of the reads will always be of FRM, and that a more efficient implementation was to just add a method, getFRM, that returns the rounding mode directly.
- **mkBypassingScoreboard**: Our scoreboard will be augmented with a bit that distinguishes between floating point and integer registers. We also extend the scoreboard to a larger size to accommodate for the maximum number of instructions possible in Execute. To always be functionally correct, it has to be the size of the maximum number of instructions possible in the Execute stage, which includes up to 5 floating-point operations (our CompletionBuffer size, described below) and one single-cycle integer register-file operation.

Lastly, we will need to add a few modules to complete the out-of-order functionality. When dispatching instructions from Decode, we need to decide which functional unit to pass the instruction to. Similarly, when an instruction finishes executing, we will need a completion buffer to hold the result until it is ready to commit in-order (i.e. all previous instructions have committed).

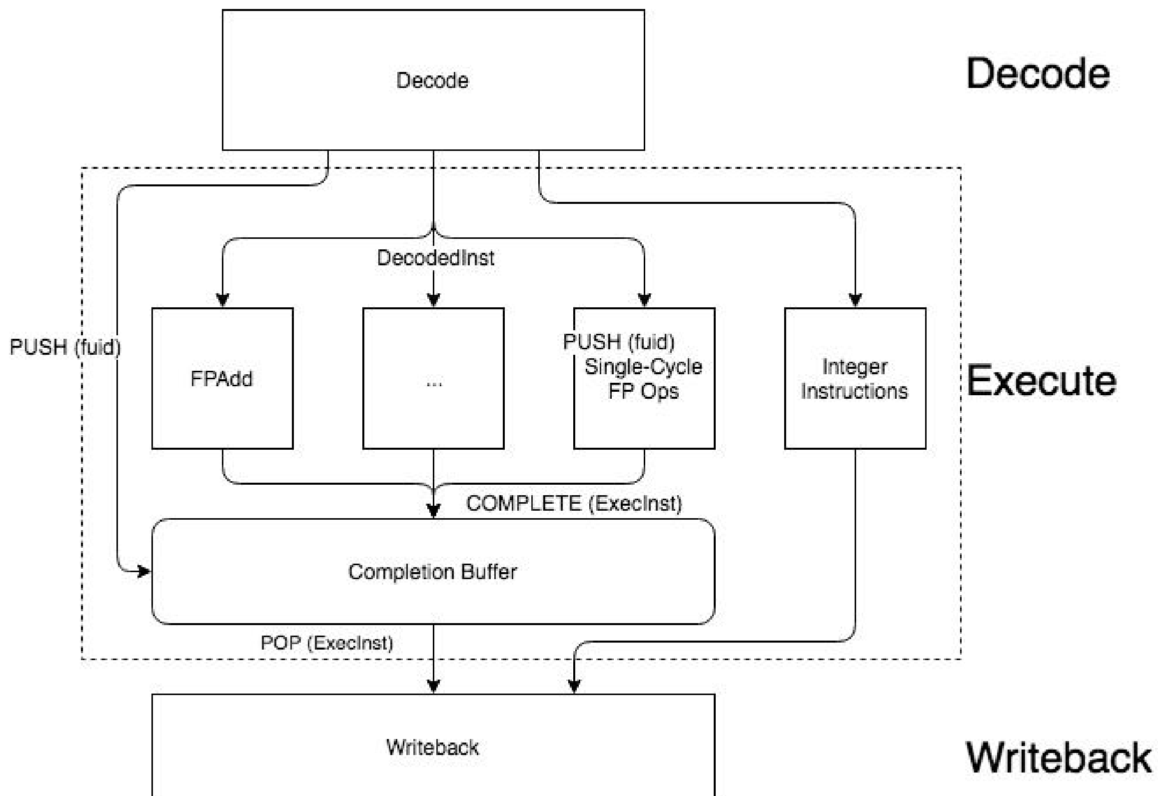

Figure 3: Microarchitecture of the modified pipelined execute stage for implementing floating-point instructions. Includes pipelined functional units for floating point operations, and a completion buffer to hold results until they're ready to commit.

- **mkCompletionBuffer**: Holds results from all completed functional units when they complete. Has a queue of the order than functional units were called, that acts as a FIFO, and only can *pop* a result when it is the first thing in the queue and has been completed. The interface of this module additionally includes a *complete* of a completed instruction with data, and a *push* method called by the reservation station to enqueue a functional unit to the ordered list of instructions to complete.

  Internally, this completion buffer can be any size, but making it larger is beneficial, since it can then hold many completed instructions that can pass an earlier instruction that uses a long-latency functional unit (likely multiply-accumulate, which takes 9 cycles). This avoids having the CompletionBuffer fills up, which would stall the rest of the pipeline.

5. Implementation

The development of the processor was accomplished through several design stages of implementation, described below.

- **Combinational**: Uses the combinational versions of the floating point add, multiply, divide, and square root operations. Also, only one instruction is allowed to pass through the processor at a time, similar to the Multicycle implementation.
- **Multicycle**: Uses sequential versions of the floating point add, multiply, divide, and square root operations, decreasing the critical path delay of the design. One instruction can flow through the pipeline at a time.
- **Four Stage w/ Bypass:** Uses sequential versions of floating point add, multiply, divide, and square root operations. Multiple instructions can flow through the pipeline, and bypassing is achieved with EHRs in the register files and scoreboard.
- **Four Stage Superscalar w/ Bypass:** Final version of the floating point processor. Uses sequential versions of floating point add, multiply, divide, and square root operations. Multiple instructions can flow through the pipeline, as well as within the Execute stage using a completion buffer.

  All of the “F” extension instruction pass compliance tests, and our processor still passes all of the original microtests for integer instructions, including both small and large benchmarks from Lab 5.

  The Execute stage now has an implementation that includes out-of-order execution between floating-point and non-floating-point instructions. All instructions that write back to the integer register file still execute in one cycle. Floating point register instructions, on the other hand, are entered into the pipelined functional unit for the relevant floating point operation, which can take between 1 and 9 cycles to complete the operation. When a floating point instruction enters a functional unit, it is pushed into a completion buffer, and when a floating point instruction leaves a functional unit, it is marked as “complete” in the completion buffer. The writeback stage then pops instructions from the completion buffer in order, which reorders the writeback of floating point instructions.

6. Performance

The table below summarizes the synthesis results of each of these designs:

	Combinational	Multicycle	FourStageBypass	Out of Order
Area (μm2)	~190,000	~187,000	~368,000	~397,000

Critical Path (ps)	~10,200	1,147	2,018	1,656

By far, the greatest improvement in clock speed was between the first and second versions of the processor, which was due to the change from the combinational to the sequential versions of the FPU. The four-stage processor with bypassing sees an increase in latency due to the long-latency bypass register file, and the out of order processor is the largest processor, as it has not only the fully pipelined processor, but also the extra CompletionBuffer for reordering out-of-order instructions.

However, it was more challenging to analyze the actual performance of the different processor versions. We initially thought that the RISC-V compliance tests was an option; however, it doesn't properly illustrate the improvement in the final version, since due to the structure of the tests only one instruction would be present in the FPU at any given time. This is because all floating point math instructions were followed by an fmv instruction that directly depends on the result of the math operation, so the second floating point instruction that would in theory be able to pass the longer-latency math instruction, instead stalls due to the data dependency between the two instructions.

Writing C code compiled to F-extension RISC-V instructions was also unsuccessful, due to the nature of the compiler, and there are no available existing floating-point benchmark programs for RVI32UF (as double-precision is more common). We settled on using the small benchmark code for Lab 5; however, we are aware that it doesn't demonstrate the full capabilities of the processor since it only uses integer instructions. The benchmarks are shown below:

	Multicycle (Floating Point)	Four Stage Bypass (Floating Point)	Four Stage Out-of-Order Bypass (Floating Point)
towers	.1033	.1069	0.1069
median	.2768	.2872	0.2873
Multiply	.4763	.3838	0.3838
Qsort	.3817	n/a	0.3090
Vvadd	.2229	.2241	0.2242

7. Conclusions

Our greatest challenge by far was, surprisingly, debugging the functional correctness of our basic floating point operations. We found bugs in the FloatingPoint.bsv library, as well as in the compliance tests that we were using to check the correctness of our processor, which was difficult to debug since we had to manually check our design, the tests, and the floating point library all for correctness.

We were able to, in the end, design and build the processor that we set out to complete, that can run and pass all of the single-precision floating point RISC-V compliance tests, as well as still pass and run the benchmarks that use integer instructions. This design also successfully compiled and ran on the FPGA. However, we were unable to do significant performance testing, due to the lack of tests that we were able to run that used a significant number of floating point operations in a realistic manner.

We learned a lot from this project, particularly about putting in the time to develop a rigorous and useful debug framework, to save time later when debugging the processor. We also spent a lot of time fixing edge cases, and in the future would've liked to spend more time on the high-level design of the processor and performance testing.