CSE 293 Bluespec Lab 2

Resources:

Git Repository:

https://github.com/markzakharov/CSE-293-BlueSpec-Tutorial

This Lab depends on Lab 1, it is expected that the student does them in order, as there are intermodule dependencies. Similar to Lab 1, it is also expected that the Parts and subparts also be done in the order specified.

Part 1: Processor front-end and back-end (50%)

Regfile: Using the Vector instantiation syntax of Vector #(size, #type), you must create a RISC-V compliant register file (hint: look at how many bits are used to represent rd/rs1/rs2). This will act as your end-result CPU's register file, aka the writeback stage. There are three access functions to implement - write(), which is a single ported value write.

The other two functions, read() and probe(), are dual ported and single ported respectively. The dual port is implemented through the Rs_vals struct, as it contains two fields.

Don't forget that RISC-V x0 is hardwired to zero!

Fetch: Instruction fetch is steered by its own stream, which means branches and jumps can steer fetch off of its usual course. Typically the pointer to instruction memory, the program counter (PC), is incremented by 4 each clock cycle, as typical RISC-V instructions are 4-bytes. When a jump instruction is decoded, the fetch stage must instead increment by the jump instructions offset. Given this, you must first update the Decoder.bsv file from the Lab1/Part1 and include a decode for the JAL instruction, as well as give it an OP and OP_TYPE of J_TYPE.

Once that is done, the methods in Fetch.bsv must be implemented, to reflect a controllable instruction memory. Init(), is_init(), and st_insn() are all wrappers around the Mem.bsv module previously implemented. This means another pointer must be maintained (this is the PC previously mentioned). Therefore get_pc() and inc_pc() are dependent upon the presence of a JAL instruction downstream.

Get_insn() must return the instruction, the same way a C_read() would return data, but making it an ActionValue is unnecessary, so convert the data to a RawInstruction, to stay type-safe. (Hint, as a setup measure, it is ok to inc_pc() once to step into the first instruction, meaning PC = 0 does not need to have any content, PC = 4 is an acceptable starting address for programs)

Part 2: Pipelining and CPU implementation (50%)

Pipeline: In order to maintain multiple “in-flight” instructions, a processor must have pipeline stages, as this allows it to save results from small calculations. This also allows it to run at a faster speed, as it does not need to wait a long time for signals coming from far away. Pipeline stages are effectively large registers that carry large amounts of information necessary to fully describe the in-flight instructions.

As rs1, rs2, and rd all will have their own values, a Values struct is defined to represent each value from a performed operation. The pipeline stages will pass along Values, along with an Instruction, and an Address representing the PC for that instruction. Access functions are to be implemented.

Helper functions are also necessary, as they allow simple comparisons between Instructions and Values, as well as providing empty versions of these structs in case a pipeline stage needs to be flushed in case of some failure/exception.

Cpu: This is where you will combine the Fetch, Decode, Execute, Memory, and Writeback modules and methods you have written to build a small RISC-V core. Pipeline stages are instantiated to go between each stage, eg: fD is between Fetch and Decode, dE is between Decode and Execute, etc… Small wrapper access methods are already implemented for the testbench.

To model a running, clocked processor, modules and methods must be placed inside of “rule” blocks, which represent the logical activity of the processor. The fetch_stage rule must stream its fetched instructions to the fD pipeline stage, and must be steered by the decode_stage if a JAL instruction is detected. The decode_stage streams the decoded Instruction to the dE pipeline stage, as well as reading the register file to send values down stream. As it sits before the Execute stage, the eM pipeline will have results from operations ready, and therefore can “forward” them back to the decode_stage, to avoid having to wait for them to be written back to the register file. The memory_stage must check for ST/LD instructions and the writeback_stage is where operations’ results are finally committed.

Grading:

This lab will be graded on the following criteria

1. Demonstration: Was the code demonstrated to a TA or instructor?
2. Correctness: Does the code pass the provided testbench?