CSE 293 Bluespec Lab 1

Resources:

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

Before writing any code, be sure to follow the setup_instructions.txt to install the Bluespec compiler (BSC), and its dependent Haskell. The last command has you add BSC to your path, so you might need sudo privileges for a complete installation.

Part 1: An introduction to combinational logic and structs (50%)

Decoder: This module only has one method to implement, which is the RISC-V compliant ISA decoding. The logic is straightforward as it can all be completed in a single, large case() statement. There is a provided “Instruction” struct which is the designated return type of the decode() method.

Following the table on page 104 of https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf - you must be able to decode the enumerated operations defined by OP - both their R-type and I-type versions, as well as specify which one they are.

Bluespec maintains a strict type system, so for type safety one must adhere to these constraints. In the case of the decoder receiving any other instructions besides those mentioned, an OP_ERR and OP_TYPE_ERR must be returned.

Execution: The converse of the Decoder module, this specifies the expected execution of the decoded RISC-V operations. It is recommended that this is again done in case() statement fashion. Its operation otherwise is more generic than the Decoder, as it has no need to differentiate between R-type and I-type instructions, as it only receives the values necessary for the operation(s).

A single executeOP() method is defined that returns the result of the passed operation and its values.

Part 2: Memories and imports (50%)

Mem: As RAM-based memories are not completely accurately modeled by RTL (access times take more than the single assumed cycle), they will be modeled through a library of C functions. Found in the C_imports directory, there is a header file as well as a .bsv file that will be used to link the C code to our .bsv code.

The relevant functions defined are: C_init(), which malloc's a provided amount of bytes and returns a pointer. This pointer should be maintained by the Mem modules rg_ptr register, so that function calls to C_read() and C_write() are offsets from rg_ptr. These two functions accept the pointer, as well as an access size (and data for C_write()) and shall be used to implement the modules own read/write methods.

These methods have conditions, specified next to their definitions' declarations of "if (rg_initialized)", this guards us from accessing unallocated memory. Nevertheless, the implemented access methods should check for out of bounds accesses.

The read() method has a return type of ActionValue#(Bit#(64)), ActionValue is a monad (a type wrapped around another type) that specifies the behavior of the method. Looking at the write() method, it is an "Action" method, as it does not have a return type yet has an effect on the state of the module. Read() by the definition of the C library, both has a return value and has an effect on the state of the module, hence needing the monadic function. Values can be returned/unwrapped through the <- assignment, instead of a typical =. One can return the result of a C_read() using something like: "let x <- C_read(..); return x;"

Load_and_store: As the Mem module was just implemented, now the RISC-V decoder must be updated to accept LD/ST instructions (only the RV32I are necessary). Modify the original Decoder.bsv file from Part1 but run Make and the tests in this directory (you will be incrementally developing the decoder)

You must now add new OPs and a new OP_TYPE to Decoder.bsv to reflect the newly available functionality. Represent them as LD, LDU, ST for OP and add S_TYPE for the store type instruction format. They can again be found on the same page 104 of the RISC-V Specification.

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?