# Verification Environment

Jacob Halsey

### instruction.e

This file defines two structs `instruction_input_s` and `instruction_output_s`. I chose to seperate the input and output structs because otherwise it would not necessarily be clear to the user which fields should be set to change the instruction behaviour, versus those which are just used to return output, such as `port_number` which indicates what port the instruction was executed on, and `ticks` which counts how many cycles it took to complete. These additional fields are useful for debugging/logging purposes.

`instruction_output_s` keeps a reference to the input it was computed from allowing it to implement a `check_response()` function that validates the response code and value. On failure `check_response()` will cause a `dut_error` to be issued, but it also returns a boolean value depending on the status, allowing `driver.e` to keep its own success counters to aid in debugging.

When testing `SHL` and `SHR` instructions I have assumed that only the 5 LSBs of the second operand are respected by the calculator, since a shift of more than 32 bits on a 32 bit integer with a 32 bit result is not explicitly defined in the specification, and this seems a sensible way to handle it.

### driver.e

The `driver.e` contains the logic to interact with the DUT by defining the `hdl_paths` for the various wires/ports of the calculator. Since all 4 sets of inputs are intended to work the same, a unit `port_u` has been defined to represent the set of `cmd_in`, `data_in`, `out_resp` and `out_data`. This allows other functions to be passed a reference to a `port_u` but then access all the same field names regardless of which wires it is actually connected to.

The struct `test_group_s` is used to support batches of test instructions that can be executed on a specific port number or in parallel on all four ports. The group can be given a name and the driver will count the number of successful instructions in the batch to make debugging particular test cases easier. The calculator is reset between each group, and the reset is checked.

The logic of the `drive_parallel` function works by instantiating a `pending_task_s` for each port, containing an instuction input. Whilst there are active tasks, each task is sent a `tick()` for each cycle that is waited on, when a `pending_task_s` has completed (i.e. collected a response) it is then replaced with another task for that port with the next instruction. On each cycle the list of pending tasks is sent to the queue checker.

### queue_checker.e

The `queue_checker_s` is used to monitor the queues of `ADD/SUB` and `SHL/SHR` operations to ensure that they follow a first come first serve priority. The queue checker does so by being updated with the current pending tasks on each cycle, checking to see which instructions have been completed and can be removed from the queue.

### tests.e

The `tests.e` file uses the `e` language generation features to create various groups of test instructions. Some are general purpose containing a variety of opcodes and operands, and some more specific ones have been used to narrow down the bugs.

### calc1_sn_env.e

The specification does not define a response time for instructions to complete, but I have relied on the `e` runtime to set a maximum tick count, to catch the case where an instruction would appear to be running indefinitely.