The UVM register layer

This page describes the parts of UVM's register layer that Lab 06 uses, so that you can look up an argument or a message while you work through the lab. Everything here is as the course's UVM library (UVM 2020-3.2, in `third_party/uvm`) implements it. The router's own register map, with the UVM access policy of every field and whether the field is volatile, is in the specification.

Configuring a field

You create a field with the factory and then configure it:

```
f = uvm_reg_field::type_id::create("noroute");
f.configure(parent,  size,  lsb,  access,  volatile,  reset,  has_reset,  is_rand,
individually_accessible);
```

Argument	Meaning
`parent`	the register the field belongs to, which is `this` in the register's `build()`
`size`	the field's width in bits
`lsb`	the position of its lowest bit in the register
`access`	the UVM access policy; the router's fields use `"RW"`, `"RO"`, `"W1C"` and `"WC"`
`volatile`	1 if the design can change the field on its own
`reset`	its value after reset
`has_reset`	1 if the field has a reset value (every field on the router has one)
`is_rand`	1 if `randomize()` may pick a value for it. Pass 1 only for a field that software writes with a value of its own choosing, which on the router means an `RW` field
`individually_accessible`	1 if UVM may read or write the field without the rest of its register. Pass 1 only for a field that is alone in its register, which on the router means the fields of `ID` and `CTRL`

The register layer never compares a volatile field when it reads one, because the design may have changed the field since the model last looked.

Mirror, desired value and prediction

The model keeps two values for every field. The **mirror** is what the model believes the design holds. The **desired** value is what the testbench last asked for. A `write()` sets the desired value first and then starts the transfer.

Way the mirror changes	When
auto-predict (`set_auto_predict(1)` on the map)	after each operation that the model itself sends, but not after a transfer that goes around the model
a `uvm_reg_predictor` connected to a monitor's analysis port	after every transfer the monitor reports, whoever started it

Way the mirror changes	When
`peek()` and `poke()`	at once, since neither uses the bus. `peek()` sets the mirror to the value it found, and `poke()` sets it to the value it wrote, without applying the field's access policy

A prediction from a read sets both the mirror and the desired value to the value read.

Check	What it compares
`r.mirror(status, UVM_CHECK)`	the mirror with the value it reads, for every field that isn't volatile
check-on-read (`set_check_on_read(1)` on the map)	the mirror with the value of every read that the predictor sees, for the same fields

Either check reports a mismatch as

```
[RegModel] Register "regs.CTRL" value read from DUT (0x0000000000000001) does not match
mirrored value (0x0000000000000000) (valid bit mask = 0x0000000000000001)
```

where the valid bit mask has a 1 for every bit that was compared.

The built-in sequences

Neither sequence needs a sequencer of its own. To run either one, set its `model` to the register block and start it with `start(null, this)`. It sends its reads and writes through the model, and so through your adapter.

Sequence	What it does	What it skips
`uvm_reg_hw_reset_seq`	resets the model to its reset values. Then, for each register, it prints `Verifying reset value of register ...` and reads the register with `mirror(UVM_CHECK)`	a register in which no field would be compared, which on the router means one whose fields are all volatile
`uvm_reg_bit_bash_seq`	prints `Verifying bits in register ...` for each register. Then, for each bit it can check, it writes the register twice, each time with that bit the opposite of the desired value, reads the register back after each write, and compares the read with the desired value	the bits of volatile fields. It still tests a bit that belongs to no field, as a read-only bit that should read 0, so both of its writes set that bit to 1

A bit-bash failure is reported as

```
[uvm_reg_bit_bash_seq] Writing a 1 in bit #0 of register "regs.ROUTE3" with initial value
'h0000000000000000 yielded 'h0000000000000002 instead of 'h0000000000000001
```

The back door

A back-door access reads (`peek`) or writes (`poke`) a register through the name of the signal that holds it, with no bus transfer and no simulation time. The model finds the signal from HDL paths. `add_hdl_path()` on the block names the design instance that holds the registers, and `add_hdl_path_slice(signal, lsb, width)` on a register names the signal that holds each of its bit ranges. In Lab 06 the block's path is `router_tb_top.dut.u_regs`, and `CTRL` is held in `en_q` inside it.

UVM reaches the signal through VPI, and Verilator lets VPI see a signal only if it is marked public. The lab's `tb/router_regs.vlt` marks as public the five signals that hold `CTRL`, `STATUS`, `INT_EN` and `DROP_CNT` (`DROP_CNT`'s two counters are separate signals), with one line for each signal. This is the line for `CTRL`:

```
public_flat_rw -module "router_regs" -var "en_q"
```

The lab names the signals one at a time because on Verilator 5.050 the UVM build breaks with `--public-flat-rw`, the command-line flag that marks every signal public. Lab 06, Part 6, measures what `peek` and `poke` do on the router.