The pyuvm register API

Use this page to look things up while you work through Lab 06. It gives the arguments of each `configure()` and what `predict()` does under each access policy. It also shows how `read()` and `write()` reach the bus, and what SystemVerilog UVM has that pyuvm 4.0.1 doesn't. The router's own register map is in the specification, with each field's policy and whether the field is volatile.

Every fact here about pyuvm comes from pyuvm 4.0.1 in the course image or from a run of the six tests in `guide-examples/pyuvm-register-api`. pyuvm's source is in `/usr/local/lib/python3.12/dist-packages/pyuvm/`, and a reference like `s19:70` means line 70 of `s19_uvm_reg_field.py` in that folder. The one quotation, under What pyuvm 4.0.1 lacks, is from pyuvm's README. The six tests run on the golden router, with the reference environment's APB agent, register model and adapter (`labs/ref/tb`) on the bus side. They print every result on a line that starts with `REGAPI`, and the outputs below are those lines, as `grep REGAPI` prints them.

The classes

You import all of these from `pyuvm`.

Class	File	What it is
`uvm_reg_block`	`s18_uvm_reg_block.py`	the register model: it holds the registers and the map
`uvm_reg`	`s20_uvm_reg.py`	one register: its fields, its address, and `read()`, `write()` and `predict()`
`uvm_reg_field`	`s19_uvm_reg_field.py`	one field: its width, position, access policy, volatility and reset value, and the model's copy of its value
`uvm_reg_map`	`s21_uvm_reg_map.py`	the registers by address, and the sequencer and adapter that their operations go through
`uvm_reg_adapter`	`s25_uvm_adapter.py`	the base class of your adapter, whose `reg2bus()` and `bus2reg()` you write
`uvm_reg_bus_op`	`s24_uvm_reg_includes.py`	the operation that the map passes to `reg2bus()` and `bus2reg()`
`uvm_reg_predictor`	`s26_uvm_predictor.py`	a class with nothing in it but a constructor (see What pyuvm 4.0.1 lacks)
`uvm_mem`	`s22_uvm_mem.py`	raises `UVMNotImplemented` as soon as you make one

The enumerations you pass and get back:

Enumeration	Values	Where it's used
`path_t`	`FRONTDOOR`, `BACKDOOR`, `USER_FRONTDOOR`	the `path` argument of `read()` and `write()`

Enumeration	Values	Where it's used
`check_t`	`CHECK`, `NO_CHECK`	the `check` argument of `read()` and `write()`
`uvm_resp_t`	`PASS_RESP`, `ERROR_RESP`	the status that `read()` and `write()` return, and that your `bus2reg()` sets
`access_e`	`UVM_READ`, `UVM_WRITE`	the `kind` of a `uvm_reg_bus_op`
`uvm_predict_e`	`UVM_PREDICT_DIRECT`, `UVM_PREDICT_READ`, `UVM_PREDICT_WRITE`	the `kind` argument of `predict()`
`uvm_door_e`	`UVM_FRONTDOOR`, `UVM_BACKDOOR`, `UVM_PREDICT`, `UVM_DEFAULT_DOOR`	the `path` argument of `predict()`

`path_t` and `uvm_door_e` both have a front door, but the two aren't interchangeable. The sections on predict() and read() and write() say what happens when you pass the wrong one.

Configuring a field

A register's `build()` makes each field and configures it (`s19:70-78`):

```
self.NOROUTE = uvm_reg_field("NOROUTE")
self.NOROUTE.configure(self, 1, 0, "W1C", True, 0)
```

Argument	Meaning
`parent`	the register the field belongs to, which is `self` in the register's `build()`
`size`	the field's width in bits
`lsb_pos`	the position of its lowest bit in the register
`access`	the access policy, one of the 26 names under Access policies; the router's fields use `"RO"`, `"RW"`, `"W1C"` and `"WC"`
`is_volatile`	`True` if the design can change the field on its own
`reset`	its value after reset

SystemVerilog UVM's `configure()` has nine arguments, and DD105-A's register layer page lists them. pyuvm's has the first six. It has nothing in place of the other three, `has_reset`, `is_rand` and `individually_accessible`.

`configure()` checks the field's size and reset value itself (`s19:93-116`). Then it adds the field to its register, and the register checks that the widths of its fields add up to no more than its own width, and that the new field doesn't overlap the one configured before it (`s20:144-179`). `RegPolicyTest` makes five mistakes on purpose. The first, a policy name in lower case, passes without a message, and each of the other four raises an exception:

```
REGAPI  configure(...,  "rw",  ...):  get_access()  is  'NOACCESS';  after  predict(0x1,
kind=UVM_PREDICT_WRITE) the mirror is 0x0
REGAPI  a  1-bit  field  with  reset  value  2:  raised  UVMError:  F  --  Reset  value  for  REG
```

```
field : F is [2]                    is beyond the MAX valoue given                    by
the size [1]
REGAPI a field of size 0: raised UVMFatalError: F -- Size of a filed                  cannot
be 0 MINIMUM allowed is 1
REGAPI a 40-bit field in a 32-bit register: raised AssertionError: WIDE -- _add_field: Fields
F                      doesn't fit into a 32 bits register
REGAPI STATUS's fields configured from bit 1 down: raised AssertionError: STATUS -- _add_field:
Fields NOROUTE overlap                with field                FULL
```

- A policy name that isn't one of the 26, such as `"rw"` in lower case, becomes `"NOACCESS"` without a message (`s19:100-101`). A `NOACCESS` field ignores every predicted write, so the mirror of a field that you meant to be read-write doesn't change when a write is predicted.
- The overlap check compares a field's lowest bit with the highest bit of the field configured just before it (`s20:169-179`), so you have to configure a register's fields from bit 0 up. When `FULL` (bit 1) of `STATUS` is configured before `NOROUTE` (bit 0), pyuvm reports an overlap, although the two fields don't share a bit.
- The last two mistakes raise `AssertionError`. Inside a test, an `AssertionError` counts as a check that fired, so a register model that raises one would be graded as a caught bug. That's why the graders build your model on its own, outside the simulator, before every verdict. You run the same build with `make regcheck` in Lab 06, Part 2. In the simulation's log, each of the two `AssertionError`s has a second line, `assert False`, which `grep REGAPI` leaves out.

Configuring a register and adding it to the map

The signatures (`s20:27`, `s20:57-64`, `s21:53`, `s21:100`):

```
uvm_reg(name, reg_width=32)
reg.configure(parent,         address,         hdl_path,         throw_error_on_read=False,
throw_error_on_write=False)
reg_map.configure(parent, base_addr)
reg_map.add_reg(reg, offset="0x0", rights="RW")
```

Call and argument	Meaning
`uvm_reg(name, reg_width)`	your register class calls this from its `__init__()`; the width is in bits
`reg.configure(parent, ...)`	`parent` is the register block. `configure()` calls the register's `build()` and adds the register to the block (`s20:75-79`)
`address`	a string, written the way Python's `hex()` writes it (see Addresses are strings)
`hdl_path`	pyuvm stores it, and nothing in pyuvm reads it (pyuvm has no back door); Lab 06 passes `""`
`throw_error_on_read`, `throw_error_on_write`	copied into each field as it is configured; Lab 06 passes `False, False`
`reg_map.configure(parent, base_addr)`	`parent` is the block. pyuvm stores `base_addr`, but no register's address uses it

Call and argument	Meaning
`add_reg(reg, offset, rights)`	the map files the register under `hex(int(offset, 16) + int(address, 16))` (`s21:100-104`). `rights` must be one of the 26 policy names, or `add_reg()` raises `UVMFatalError`. pyuvm stores `rights`, and only `get_access_policy()` reads it back

```
REGAPI   add_reg(reg,   "0x0",   "XX"):   raised   UVMFatalError:   R   --   given   access
policy not correct: XX
```

Addresses are strings

The map files each register under the `hex()` form of its address plus the offset, but `read()` and `write()` look the register up by the string that it was configured with (`s21:325`, `s21:379`). The two strings are the same only when the register's address is written the way `hex()` writes it and the offset is zero. That's why Lab 06 configures every register with `hex(addr)` and adds it with `"0x0"`. `RegAddrTest` configures three registers on a map whose base address is `0x200`:

```
REGAPI map.configure(block, 0x200); the map's keys: ['0x0', '0x104', '0x8']
REGAPI Z.read(...), Z configured at "0x00" and added at offset "0x0": raised KeyError: '0x00'
REGAPI O.read(...), O configured at "0x4" and added at offset "0x100": raised KeyError: '0x4'
REGAPI    reg2bus() got a uvm_reg_bus_op: kind UVM_READ, addr '0x8', data 0x0, n_bits 32,
byte_en True
REGAPI   bus, cycle 8: APB R 08 00000000
REGAPI S.read(...), S configured at "0x8" and added at offset "0x0": returned (PASS_RESP, 0x0)
REGAPI map.get_reg_by_offset("0x8").get_name(): S
REGAPI map.get_reg_by_offset("0x08").get_name(): raised KeyError: '0x08'
```

The read of `S` went to address `08` on the bus, so the map's base address, `0x200`, wasn't added to it. The address reaches your adapter as the same string, `'0x8'`, which is why Lab 06's `reg2bus()` turns it back into a number with `int(rw.addr, 16)`. `make regcheck`, which runs the same check that the graders run before every verdict, reports a register configured at an address that `hex()` would write differently, such as `"0x00"` (`labs/common/README.md`).

Access policies

pyuvm accepts 26 policy names (`s17:16-47`). `predict()` with `UVM_PREDICT_WRITE` applies the policy's write rule (`predict_write()`, `s19:421-452`), and with `UVM_PREDICT_READ` its read rule (`predict_read()`, `s19:486-512`). `RegPolicyTest` gives a 4-bit field each policy in turn, with a reset value of `0x5`. Before each prediction it puts the mirror at `0xc` with `UVM_PREDICT_DIRECT`. In the table, then, `0xc` means that the mirror didn't change, `0xa` that it took the value written, `0x3` that it took the value read, and `0x5` that it went back to the reset value:

```
REGAPI   policy    WRITE of 0xa    READ of 0x3
REGAPI   RO          0xc             0x3
REGAPI   RW          0xa             0x3
REGAPI   RC          0xc             0x0
REGAPI   RS          0xc             0xf
REGAPI   WRC         0xa             0x0
REGAPI   WRS         0xa             0xf
REGAPI   WC          0x0             0x5
```

```
REGAPI   WS          0xf           0x5
REGAPI   WSRC        0xf           0x0
REGAPI   WCRS        0x0           0xf
REGAPI   W1C         0x4           0x5
REGAPI   W1S         0xe           0x5
REGAPI   W1T         0x6           0x5
REGAPI   W0C         0x8           0x5
REGAPI   W0S         0xd           0x5
REGAPI   W0T         0x9           0x5
REGAPI   W1SRC       0xe           0x0
REGAPI   W1CRS       0x4           0xf
REGAPI   W0SRC       0xd           0x0
REGAPI   W0CRS       0x8           0xf
REGAPI   WO          0xa           0x5
REGAPI   WOC         0x0           0x5
REGAPI   WOS         0xf           0x5
REGAPI   W1          0x6           0x5
REGAPI   WO1         0xa           0x5
REGAPI   NOACCESS    0xc           0x5
```

The router's four policies behave like this:

Policy	Router fields	A predicted write	A predicted read
`RO`	`ID`, `LEVEL0`, `LEVEL1`	leaves the mirror as it was	takes the value read
`RW`	`CTRL.EN`, `INT_EN`'s three, `ROUTEn.EN` and `ROUTEn.PORT`	takes the value written	takes the value read
`W1C`	`STATUS`'s three	clears the bits written as 1 (`0xc` and `0xa` give `0x4`)	goes back to the reset value
`WC`	`NOROUTE_CNT`, `FULL_CNT`	clears every bit	goes back to the reset value

In two places these results differ from what SystemVerilog UVM does. For SystemVerilog UVM we read `uvm_reg_field.svh` in DD105-A's copy of the library, which is UVM 2020-3.2.

- In pyuvm, `UVM_PREDICT_READ` puts a field back at its reset value, whatever was read, under 14 policies: `WO`, `WOC`, `WOS`, `WO1`, `NOACCESS`, `W1`, `W1T`, `W0T`, `WC`, `WS`, `W1C`, `W1S`, `W0C` and `W0S` (`s19:494-511`). Under the first five of those, SystemVerilog UVM leaves the mirror as it was, and under the other nine it takes the value read (`uvm_reg_field.svh`, lines 985 to 1031). The policies of the router's `STATUS` (`W1C`) and `DROP_CNT` (`WC`) are among the nine, so in pyuvm a predictor that predicts each read with `UVM_PREDICT_READ` leaves both registers at their reset value, 0, after every read. `UVM_PREDICT_DIRECT` takes the value read under every policy, and it's the kind that Lab 06′s predictor uses after a read. Part 3, step 1 shows the difference on your own model, with `make predict`.
- A `W1` field should take the first value written after a reset and ignore later writes. pyuvm's own comment on the policy says so (`s19:416-417`), and SystemVerilog UVM does it (`uvm_reg_field.svh`, line 893). In pyuvm, under `UVM_PREDICT_WRITE`, a `W1` field toggles the bits written as 1 instead, as `W1T` does, so `0xc` and `0xa` give `0x6` where they should give `0xa`. The test for `W1T` in `predict_write()` is `field_access in ("W1T")` (`s19:438`). Python

reads `("W1T")` as the string `"W1T"` rather than as a tuple, so the test is true for a `W1` field too, because `"W1"` is a substring of `"W1T"`. The router has no `W1` field.

predict()

```
reg.predict(value, be=-1, kind=uvm_predict_e.UVM_PREDICT_DIRECT,
            path=uvm_door_e.UVM_FRONTDOOR, map=None, fname="", lineno=0)
```

A register's `predict()` (`s20:193-212`) hands each field its own bits of `value` through the field's `do_predict()` (`s20:214-233`). The field works out the predicted value from `kind`, `path` and its policy, and sets both its mirror and its desired value to it (`s19:568-570`). A field has a `predict()` of its own, with the same arguments (`s19:575-594`). `predict()` returns `True` unless the operation's status is `UVM_NOT_OK`.

Pass `kind` and `path` by name. `kind` is the third parameter, after `be`, the byte enables, so `predict(value, kind)` passes the kind as `be`. The register then raises `TypeError` when it shifts `be` for its first field (`s20:232`), before any field's mirror has changed. `path` takes a `uvm_door_e`. If you pass `path_t.FRONTDOOR` instead, the field skips its policy (`s19:534-539`) and takes the value as it is, as if `kind` were `UVM_PREDICT_DIRECT`. `RegPolicyTest` predicts a write on `STATUS` each of the three ways:

```
REGAPI STATUS at 0x7, then predict(0x1, kind=UVM_PREDICT_WRITE): 0x6
REGAPI STATUS at 0x7, then predict(0x1, kind=UVM_PREDICT_WRITE, path=path_t.FRONTDOOR): 0x1
REGAPI STATUS at 0x7, then predict(0x1, UVM_PREDICT_WRITE): raised TypeError: unsupported
operand type(s) for >>: 'uvm_predict_e' and 'int'
REGAPI   STATUS's mirror after that: 0x7
```

The other methods that read or set the model's values:

Method	What it does
`reg.get_mirrored_value()`	the mirror: every field's mirror, shifted into its place (`s20:236-241`)
`reg.get_desired()`, `reg.set_desired(value)`	the desired value. `set_desired()` applies each field's write rule to it (`s20:139-141`, `s19:302-340`), and `write(None, ...)` writes it
`reg.reset()`, `reg_map.reset("HARD")`	every field's mirror back to its reset value (`s19:262-265`). The map takes `"HARD"` or `"SOFT"`, which do the same, and raises `UVMNotImplemented` for anything else
`field.get_value()`, `field.get()`, `field.get_reset()`	a field's mirror, its desired value and its reset value
`field.get_access()`, `field.is_volatile()`, `field.get_lsb_pos()`, `field.get_n_bits()`, `field.get_msb_pos()`	what `configure()` was given, except that `get_access()` returns `"NOACCESS"` for a policy name that isn't one of the 26, and `get_msb_pos()` works out the field's highest bit from its position and width
`reg.get_fields()`, `reg.get_address()`, `reg.get_reg_size()`	the register's fields in the order they were configured, which is from bit 0 up; its address string; its width
`reg_map.get_registers(as_dict=True)`	the map's registers, keyed by the `hex()` strings

```
REGAPI  map.reset("POWER"):  raised  UVMNotImplemented:  map  --  reset  --  POWER  is  not
mapped as type of reset                                  available values are ['SOFT', 'HARD']
```

read() and write()

```
status = await reg.write(value, reg_map, path, check)
status, value = await reg.read(reg_map, path, check)
```

`write()` returns a `uvm_resp_t`, and `read()` returns a `uvm_resp_t` and the value read (`s20:276-358`). Neither has default arguments.

Argument	Meaning
`value`	the value to write, or `None` to write the register's desired value (`s20:304-307`)
`reg_map`	the map that holds the register; Lab 06's is `regs.map`
`path`	use `path_t.FRONTDOOR`. `path_t.BACKDOOR` and `path_t.USER_FRONTDOOR` raise `UVMNotImplemented`. Any other value, such as `uvm_door_e.UVM_FRONTDOOR`, matches none of the three, so `write()` returns `None` without a transfer on the bus, and `read()` raises `TypeError`
`check`	used only by auto-predict, which is off (see What pyuvm 4.0.1 lacks), so `read(..., check_t.CHECK)` compares nothing

For a front-door operation, the map builds a `uvm_reg_bus_op` and hands it to your adapter's `reg2bus()`. It starts the item that `reg2bus()` returns on the map's sequencer and waits for the driver to finish it. Then it calls `bus2reg(item, rw)` and returns the status that `bus2reg()` put in `rw.status`, with `rw.data` for a read (`s21:299-406`). A read whose status is `ERROR_RESP` returns 0 as its value (`s20:346-347`). Neither the map nor the register changes the mirror. `RegOpsTest` reads and writes registers through the map, on the golden router:

```
REGAPI ID's mirror before the read: 0xd1050001
REGAPI    reg2bus() got a uvm_reg_bus_op: kind UVM_READ, addr '0x0', data 0x0, n_bits 32,
byte_en True
REGAPI   bus, cycle 8: APB R 00 d1050001
REGAPI ID.read(map, path_t.FRONTDOOR, check_t.NO_CHECK): returned (PASS_RESP, 0xd1050001)
REGAPI    reg2bus() got a uvm_reg_bus_op: kind UVM_WRITE, addr '0x4', data 0x1, n_bits 32,
byte_en True
REGAPI   bus, cycle 11: APB W 04 00000001
REGAPI CTRL.write(0x1, map, path_t.FRONTDOOR, check_t.NO_CHECK): returned PASS_RESP
REGAPI CTRL's mirror after the write: 0x0
REGAPI    reg2bus() got a uvm_reg_bus_op: kind UVM_READ, addr '0x4', data 0x0, n_bits 32,
byte_en True
REGAPI   bus, cycle 14: APB R 04 00000001
REGAPI CTRL.read(map, path_t.FRONTDOOR, check_t.CHECK): returned (PASS_RESP, 0x1)
REGAPI CTRL's mirror after the read: 0x0; errors logged so far: 0
REGAPI INT_EN.set_desired(0x5): get_desired() is 0x5, get_mirrored_value() is 0x0
```

```
REGAPI    reg2bus() got a uvm_reg_bus_op: kind UVM_WRITE, addr '0xc', data 0x5, n_bits 32,
byte_en True
REGAPI   bus, cycle 17: APB W 0c 00000005
REGAPI INT_EN.write(None, map, path_t.FRONTDOOR, check_t.NO_CHECK): returned PASS_RESP
REGAPI CTRL.write(0x0, map, uvm_door_e.UVM_FRONTDOOR, check_t.NO_CHECK): returned None
REGAPI   transfers on the bus during that write: 0
REGAPI    reg2bus() got a uvm_reg_bus_op: kind UVM_READ, addr '0x4', data 0x0, n_bits 32,
byte_en True
REGAPI   bus, cycle 21: APB R 04 00000001
REGAPI CTRL.read(map, path_t.FRONTDOOR, check_t.NO_CHECK): returned (PASS_RESP, 0x1)
REGAPI ROUTE0.read(map, uvm_door_e.UVM_FRONTDOOR, check_t.NO_CHECK): raised TypeError: cannot
unpack non-iterable NoneType object
REGAPI ROUTE0.read(map, path_t.FRONTDOOR, check_t.NO_CHECK) after that: raised UVMFatalError:
ROUTE0 -- read cannot perform an operation while                    another is in progress
```

`ID`'s mirror is already `0xd1050001` before the read, because the test's environment calls `map.reset("HARD")` when it builds the model. `CTRL`'s mirror stays at `0x0` after the write of `0x1` and after the read, and the read with `check_t.CHECK` logs no error, although it read `0x1` and the mirror held `0x0`. `CTRL` reads `0x1` after the write of `0x0` with `uvm_door_e.UVM_FRONTDOOR`, because that write never reached the bus. The last line comes from the rule in the next section.

One operation per register at a time

A register sets a flag, `_op_in_progress`, for the whole of a `read()` or `write()`, and an operation that starts while the flag is set raises `UVMFatalError` at once (`s20:296-316`, `s20:340-356`). SystemVerilog UVM would make the second operation wait. Operations on two different registers can overlap. `RegTwoOpsTest` overlaps two operations on `CTRL` twice, and then an operation on `CTRL` with one on `INT_EN`:

```
REGAPI    reg2bus() got a uvm_reg_bus_op: kind UVM_WRITE, addr '0x4', data 0x1, n_bits 32,
byte_en True
REGAPI CTRL.write(0x0, ...), started in the same time step: raised UVMFatalError: CTRL --
write cannot perform an operation while                    another is in progress
REGAPI   bus, cycle 6: APB W 04 00000001
REGAPI CTRL.write(0x1, ...), started first: returned PASS_RESP
REGAPI    reg2bus() got a uvm_reg_bus_op: kind UVM_WRITE, addr '0x4', data 0x0, n_bits 32,
byte_en True
REGAPI CTRL.read(...), one cycle into that write: raised UVMFatalError: CTRL -- read cannot
perform an operation while                    another is in progress
REGAPI   bus, cycle 8: APB W 04 00000000
REGAPI CTRL.write(0x0, ...): returned PASS_RESP
REGAPI    reg2bus() got a uvm_reg_bus_op: kind UVM_WRITE, addr '0x4', data 0x1, n_bits 32,
byte_en True
REGAPI    reg2bus() got a uvm_reg_bus_op: kind UVM_WRITE, addr '0xc', data 0x7, n_bits 32,
byte_en True
REGAPI   bus, cycle 11: APB W 04 00000001
REGAPI CTRL.write(0x1, ...): returned PASS_RESP
REGAPI   bus, cycle 14: APB W 0c 00000007
REGAPI INT_EN.write(0x7, ...), started in the same time step: returned PASS_RESP
REGAPI    reg2bus() got a uvm_reg_bus_op: kind UVM_READ, addr '0x4', data 0x0, n_bits 32,
byte_en True
```

```
REGAPI   bus, cycle 19: APB R 04 00000001
REGAPI CTRL.read(...) after all that: returned (PASS_RESP, 0x1)
```

In each pair on `CTRL`, the operation that started second raises, and the first one completes. The flag is cleared only when an operation returns normally, so an exception raised inside `read()` or `write()` leaves it set, and every later operation on that register raises `UVMFatalError` too. That happened to `ROUTE0` in `RegOpsTest` above, and to `CTRL` after a back-door read in `RegBackdoorTest`, below. Lab 06's register helpers are in `tb/router_vseq.py`. `wr_reg` and `rd_reg` hold a cocotb `Lock` for the register while the operation runs, and `check_reg` and `rd_all_regs` go through `rd_reg`, so two parts of a sequence that use one register take turns.

Your adapter

Your adapter derives from `uvm_reg_adapter`, whose `reg2bus()` and `bus2reg()` raise `UVMNotImplemented` until you override them (`s25:32-43`).

Method	The map calls it	Yours does
`reg2bus(rw)`	with each front-door operation, before anything reaches the bus	returns a new bus item (an `ApbItem` in Lab 06) made from `rw`
`bus2reg(item, rw)`	with the same item, once the driver has finished it	sets `rw.kind`, `rw.data` and `rw.status` from the item; `rw.status` is what `read()` or `write()` returns

`rw` is a `uvm_reg_bus_op` (`s24:162-174`):

Field	In `reg2bus()`
`kind`	`access_e.UVM_READ` or `access_e.UVM_WRITE`
`addr`	the register's address, as the string it was configured with (`'0x4'`)
`data`	the value to write, or 0 for a read
`n_bits`	the register's width
`byte_en`	the adapter's `byte_enable`, which is `True` unless you change it
`status`	`uvm_resp_t.PASS_RESP`, for `bus2reg()` to change

What pyuvm 4.0.1 lacks

pyuvm 4.0.1's `uvm_reg_predictor` has no `predict()`, and pyuvm has no reset or bit-bash sequence, so Lab 06 has you write all three. The first column of this table is SystemVerilog UVM's register layer as DD105-A's register layer page describes it, and the second is what pyuvm 4.0.1 has in its place.

SystemVerilog UVM	pyuvm 4.0.1	Shown by
`uvm_reg_predictor`, which updates the mirror from a bus monitor's transfers	a class of that name with nothing in it but a constructor, so it has no analysis export and no `predict()`. pyuvm's README describes it as	`s26:6-9`; `RegPolicyTest` and `RegAutoPredictTest`, below

SystemVerilog UVM	pyuvm 4.0.1	Shown by
	"Main register predictor, should be disabled if auto_prediction is not set"	
auto-predict, which updates the mirror after each operation the model sends	off, with no method to turn it on (see below)	`s24:125`, `s21:8`, `s21:345-347`; `RegAutoPredictTest`
check-on-read in the map, `mirror(UVM_CHECK)` on a register	neither; `read(..., check_t.CHECK)` compares nothing	`RegPolicyTest`; `RegOpsTest`, above
`mirror()`, `update()`, `needs_update()`, `peek()`, `poke()`, `set()`, `get()` and `get_reset()` on a register	none of them. A field has `get()` and `get_reset()`, and a register has `set_desired()` and `get_desired()`. The to-do list at the top of the block's file includes `needs_update`, `update` and `mirror`	`RegPolicyTest`; `s18:31-33`
`uvm_reg_hw_reset_seq`, `uvm_reg_bit_bash_seq`, `uvm_reg_access_seq`, `uvm_reg_mem_built_in_seq`	none of them. The enumeration `uvm_reg_mem_tests_e` has a value for running each of the first three (`UVM_DO_REG_HW_RESET`, `UVM_DO_REG_BIT_BASH`, `UVM_DO_REG_ACCESS`), and no file in pyuvm 4.0.1 defines the sequences	`RegPolicyTest`; `s17:149-164`
the back door, and HDL paths	`path_t.BACKDOOR` raises `UVMNotImplemented`, and so does `add_hdl_path()` on a block	`RegBackdoorTest`; `s21:316-317`, `s21:370-371`, `s18:77-78`
a user-defined front door	`path_t.USER_FRONTDOOR` raises `UVMNotImplemented`	`RegBackdoorTest`; `s21:318-319`, `s21:372-373`
`uvm_mem`	raises `UVMNotImplemented`	`RegPolicyTest`; `s22:7-10`
a second operation on a register waits for the first	it raises `UVMFatalError`	`RegTwoOpsTest`, above

What `RegPolicyTest` found:

```
REGAPI add_hdl_path("dut.u_regs") on the block: raised UVMNotImplemented: router_regs --
REGAPI uvm_mem("mem"): raised UVMNotImplemented: mem uvm_mem not implemented
REGAPI uvm_reg_predictor derives from uvm_object and defines ['__init__']
REGAPI uvm_reg has read, write, predict, get_mirrored_value, get_desired, set_desired, reset;
it has no mirror, update, needs_update, peek, poke, get_reset, set, get
REGAPI uvm_reg_field has predict, get_value, get_reset, get, field_set, reset, is_volatile,
get_access; it has no set
REGAPI uvm_reg_map has set_sequencer, set_adapter, set_predictor, reset, get_registers,
get_reg_by_offset; it has no set_auto_predict, set_check_on_read
REGAPI uvm_reg_block has get_registers, get_reg_by_name; it has no mirror, update, reset
```

```
REGAPI in pyuvm: uvm_reg_predictor; not in pyuvm: uvm_reg_hw_reset_seq, uvm_reg_bit_bash_seq,
uvm_reg_access_seq, uvm_reg_mem_built_in_seq
REGAPI enable_auto_predict: False in s21_uvm_reg_map, False in s24_uvm_reg_includes
```

The back door, in `RegBackdoorTest`:

```
REGAPI CTRL.read(map, path_t.BACKDOOR, check_t.NO_CHECK): raised UVMNotImplemented: map --
BACKDOOR not implemented
REGAPI CTRL.read(map, path_t.FRONTDOOR, check_t.NO_CHECK) after that: raised UVMFatalError:
CTRL -- read cannot perform an operation while                    another is in progress
REGAPI INT_EN.write(0x1, map, path_t.BACKDOOR, check_t.NO_CHECK): raised UVMNotImplemented:
map -- BACKDOOR not implemented
REGAPI ROUTE0.read(map, path_t.USER_FRONTDOOR, check_t.NO_CHECK): raised UVMNotImplemented:
map -- USER_FRONTDOOR not implemented
REGAPI transfers on the bus: 0
```

Auto-predict

A module-level variable, `enable_auto_predict`, turns auto-predict on and off, and it's `False` in `s24_uvm_reg_includes` (`s24:125`). The map's module imports it by name (`s21:8`), so `s21_uvm_reg_map` gets its own copy of the value when pyuvm is imported, and setting the variable in `s24_uvm_reg_includes` afterwards doesn't change the map's copy. Set to `True` in `s21_uvm_reg_map`, it makes the map call `predict()` on its predictor after each operation (`s21:345-347`, `s21:398-400`). But `set_predictor()` accepts only a `uvm_reg_predictor` (`s21:122-130`), and that class has no `predict()`. `RegAutoPredictTest` sets the variable in each module in turn, and in `s21_uvm_reg_map` tries it without a predictor and then with one:

```
REGAPI    reg2bus() got a uvm_reg_bus_op: kind UVM_WRITE, addr '0x4', data 0x1, n_bits 32,
byte_en True
REGAPI   bus, cycle 5: APB W 04 00000001
REGAPI with s24_uvm_reg_includes.enable_auto_predict = True: CTRL.write(0x1, ...): returned
PASS_RESP
REGAPI   CTRL's mirror after it: 0x0
REGAPI    reg2bus() got a uvm_reg_bus_op: kind UVM_WRITE, addr '0xc', data 0x1, n_bits 32,
byte_en True
REGAPI   bus, cycle 8: APB W 0c 00000001
REGAPI with s21_uvm_reg_map.enable_auto_predict = True, no predictor: INT_EN.write(0x1, ...):
raised UVMError: map -- predictor Not set
REGAPI    reg2bus() got a uvm_reg_bus_op: kind UVM_WRITE, addr '0x20', data 0x1, n_bits 32,
byte_en True
REGAPI   bus, cycle 12: APB W 20 00000001
REGAPI      the      same,      with      map.set_predictor(uvm_reg_predictor("predictor")):
ROUTE0.write(0x1, ...): raised AttributeError: 'uvm_reg_predictor' object has no attribute
'predict'
REGAPI mirrors after the three writes: CTRL 0x0, INT_EN 0x0, ROUTE0 0x0; transfers on the
bus: 3
```

Each of the three writes reached the router, and none of them changed a mirror. In Lab 06 your predictor is a `uvm_subscriber` on the APB monitor, and it calls `predict()` itself. It sees every transfer on the bus, including the raw ones that go around the model.