Interfaces, clocking blocks and virtual interfaces

This page answers: *a class isn't part of the design. How does it drive the design's pins, and read them back, without racing the design's flip-flops?*

In Lab 01 you read a driver that wrote `vif.drv_cb.A <= req.A` and a monitor that read `vif.mon_cb.ALUout`. In Lab 02 you'll write a monitor for the router's valid/ready streams, and it will read each stream's signals through names like `vif.cb.valid`, `vif.cb.ready` and `vif.cb.data`. Each of these names has three parts. In `vif.drv_cb.A`, `A` is a signal in an **interface**, which bundles the design's signals. `vif` is a **virtual interface**, which is how a class gets hold of one particular instance of that interface. `drv_cb` is a **clocking block**, which says when the testbench samples and drives the signals. This page explains them in that order.

The example on this page drives the golden router from `labs/common/rtl`, and we ran it in the course image. Its source is in `guide-examples/interfaces-clocking-blocks-and-virtual-interfaces/`. The commands that produced each output are at the top of that folder's Makefile, and in the README one level up, `guide-examples/README.md`.

Interfaces

An interface is a named bundle of signals that you declare once and instantiate like a module. The example's interface, `rtr_pins_if`, holds the signals of the router's APB port, its input stream and both of its output streams. Shortened, it reads:

```
interface rtr_pins_if (input logic clk, input logic rst_n);
  logic        psel = 1'b0, penable = 1'b0, pwrite = 1'b0;
  logic [7:0]  paddr = 8'h00;
  logic [31:0] pwdata = 32'h0;
  logic        pready;
  logic        in_valid = 1'b0, in_last = 1'b0;
  logic [7:0]  in_data = 8'h00;
  logic        in_ready;
  logic        out0_valid;
  logic [7:0]  out0_data;
  ...
  clocking cb @(posedge clk);
    default input #1step output #1;
    output psel, penable, pwrite, paddr, pwdata, in_valid, in_data, in_last;
    input  pready, in_ready, out0_valid, out0_data, rst_n;
  endclocking
endinterface
```

The top module makes an instance of it called `pins`, with `rtr_pins_if pins (clk, rst_n);`, and connects the router's ports to the signals in that instance: `.in_valid(pins.in_valid)`, `.in_ready(pins.in_ready)` and so on. Any code that can see `pins` can reach every signal in it.

An interface can also say who drives what. Lab 01's `alu_if` does it with two modports, `DRV` and `MON`. `DRV` gives access only to the clocking block `drv_cb`, whose signals are all outputs, and `MON` only to `mon_cb`, whose signals are all inputs. The driver gets `alu_if.DRV` and the monitor gets `alu_if.MON`. Lab 02 does it for the router's streams with three interface types instead. In a `stream_src_if`, which carries the input stream, the testbench drives `valid`, `data` and `last`. In a `stream_snk_if`, which carries an output stream, the testbench drives only `ready`. Every signal of a `stream_mon_if` is an input. The top module makes one `stream_mon_if` for the input stream and another for output 0, and the monitor class holds a `stream_mon_if`, so the same code watches either stream and can't drive it by mistake.

Virtual interfaces

A class can't contain an interface instance, and the standard doesn't let code in a package name anything in the module hierarchy, so a class in a package can't reach `pins` by its name either. Instead it holds a **virtual interface**, which is a variable of type `virtual rtr_pins_if`. Its value is a handle to one particular instance of `rtr_pins_if`, and it's assigned while the simulation runs.

```
class edge_demo;
  virtual rtr_pins_if vif;

  function new(virtual rtr_pins_if vif);
    this.vif = vif;
  endfunction
  ...
endclass
```

The course hands each virtual interface to its classes in two steps. First the top module stores each instance in UVM's configuration database, before it starts the test:

```
uvm_config_db#(virtual rtr_pins_if)::set(null, "uvm_test_top", "pins_vif", pins);
run_test();
```

Then the test fetches it and passes it on to the classes that need it:

```
if (!uvm_config_db#(virtual rtr_pins_if)::get(this, "", "pins_vif", vif))
  `uvm_fatal("NOVIF", "the top module didn't set pins_vif")
d = new(vif);
```

Lab 02's `router_class_test` does this for each of its six interfaces and hands them to `env::new()`, which passes each one to the class that uses it. In Lab 01 the UVM driver and monitor each fetch their own interface from the database, in `build_phase`.

A typedef of a virtual interface type

On Verilator 5.050, a class in a package reads wrong values from a clocking block if it reaches the interface through a `typedef` of the virtual interface type. The example's `typedef_test` waits for the end of reset and then sets `CTRL.EN`, while two monitor classes in one package print what their clocking block sampled at each rising edge. One declares its handle as `virtual rtr_mon_if`, and the other as `rtr_mon_vif_t`, after `typedef virtual rtr_mon_if rtr_mon_vif_t;`. The interface has two identical clocking blocks, `cb_a` and `cb_b`. The first class reads `cb_a` and the second reads `cb_b`, so each block is only ever read through one kind of handle. Apart from that, the two classes are the same. They printed:

```
45 ns: virtual rtr_mon_if: rst_n 1 in_ready 0 out0_valid 0
45 ns: rtr_mon_vif_t:      rst_n 0 in_ready 0 out0_valid 0
55 ns: virtual rtr_mon_if: rst_n 1 in_ready 0 out0_valid 0
55 ns: rtr_mon_vif_t:      rst_n 0 in_ready 0 out0_valid 0
65 ns: virtual rtr_mon_if: rst_n 1 in_ready 1 out0_valid 0
65 ns: rtr_mon_vif_t:      rst_n 0 in_ready 0 out0_valid 0
75 ns: virtual rtr_mon_if: rst_n 1 in_ready 1 out0_valid 0
75 ns: rtr_mon_vif_t:      rst_n 0 in_ready 0 out0_valid 0
```

The reset was over at all four edges. The class that uses `virtual rtr_mon_if` read `rst_n` as 1 at each of them, and `in_ready` as 1 from 65 ns on, once `CTRL.EN` was set. The class that uses the typedef read 0 from every clocking-block input at every edge. Verilator gave no warning at build time and no error at run time. That's why every class in the labs writes its virtual interface type out in full, as in `virtual stream_mon_if`, and the comment at the top of Lab 02's `driver.svh` gives this reason.

Clocking blocks

When a rising edge comes, the design's flip-flops take their new values and your testbench, which was waiting for that edge, wakes up. Both happen at the same moment of simulated time. If the testbench then reads a flip-flop's output straight from the pin, it can get either the old value or the new one. Which one it gets depends on when the testbench woke up within that moment, and that depends on what it was waiting for.

The example's `edge_test` shows what Verilator does. After it sets up a route, it writes 1 to `CTRL.EN`, which takes effect at the rising edge that ends the write and makes `in_ready` 1 from then on. Then it offers a two-beat packet. At every rising edge it prints `in_ready` and `out0_valid` twice, once as the clocking block sampled them (`cb`) and once read straight from the pins at the same moment (`pin`). It also prints the `in_valid` pin.

```
65 ns edge: in_ready cb 0 pin 0 | in_valid pin 0 | out0_valid cb 0 pin 0
75 ns: CTRL.EN written
75 ns edge: in_ready cb 0 pin 1 | in_valid pin 0 | out0_valid cb 0 pin 0
76 ns: in_valid pin changes to 1
85 ns: handshake on beat 00
85 ns edge: in_ready cb 1 pin 1 | in_valid pin 1 | out0_valid cb 0 pin 0
95 ns: handshake on beat 5a
95 ns edge: in_ready cb 1 pin 1 | in_valid pin 1 | out0_valid cb 0 pin 1
96 ns: in_valid pin changes to 0
105 ns edge: in_ready cb 1 pin 1 | in_valid pin 0 | out0_valid cb 1 pin 1
115 ns edge: in_ready cb 1 pin 1 | in_valid pin 0 | out0_valid cb 1 pin 0
```

At 75 ns the pin already reads 1, because the flip-flop had updated before the class read it. The clocking block reads 0, which is what `in_ready` was during the cycle that ended at 75 ns. The router's specification defines the value of a signal "in cycle n" the same way, as the value it holds just before the edge that ends cycle n. A monitor needs that value, because a handshake is a cycle in which `valid` and `ready` are both 1. A monitor that read `out0_valid` from the pin would get each value one cycle early (look at 95 ns and 105 ns), and it would pair each value with the `ready` of the wrong cycle.

The first two lines of the clocking block set this timing:

```
clocking cb @(posedge clk);
  default input #1step output #1;
```

- `@(posedge clk)` is the clocking event, and `@(vif.cb)` waits for it.
- `input #1step` makes every input in the block a sample taken just before the edge, so `vif.cb.in_ready` is its value in the cycle that the edge ends, whatever the flip-flops do at the edge itself.
- `output #1` makes every drive through the block take effect 1 ns after the edge. The class wrote `vif.cb.in_valid <= 1` right after the edge at 75 ns, and the pin changed at 76 ns. So nothing that the testbench drives through the block changes at the moment the block samples its inputs.

Here is the same run in Surfer, from 60 ns to 120 ns:



Figure 1: The router's APB write of CTRL.EN and a two-beat packet, driven through a clocking block, in Surfer

`in_ready` rises exactly at the 75 ns edge, because it comes from a flip-flop. `psel` and `penable` fall and `in_valid` rises at 76 ns, 1 ns after the edge, because the class drove them through the clocking block. The router takes `00` at 85 ns and `5a` at 95 ns, and `out0_valid` is high from 95 ns to 115 ns.

With both skews in place, a driver is a loop that does one clock cycle per pass. Each pass drives what the next cycle should hold, waits for the edge with `@(vif.cb)`, and then looks at what the clocking block sampled. A monitor's loop is the same without the drive. The example offers one beat with this task:

```
task send_beat(bit [7:0] data, bit last);
  vif.cb.in_valid <= 1'b1;
  vif.cb.in_data  <= data;
  vif.cb.in_last  <= last;
  do @(vif.cb); while (!vif.cb.in_ready);
endtask
```

Each pass of the `do` loop waits for an edge and then checks whether `in_ready` was 1 in the cycle that just ended. If it was, `in_valid` was 1 in that cycle too, so that cycle was the handshake and the beat moved at that edge. Lab 02's `driver::run()` offers each beat with the same loop, except that it stops with an error if `in_ready` stays 0 for 1,000 cycles.

Module code and nonblocking assignments

There's one more reason that the classes in the labs drive the design only through clocking blocks. On Verilator 5.050, a module task can lose assignments if it raises a signal for one cycle with nonblocking assignments and is called several times in a `repeat` loop. The example's top module raises one signal, `by_task`, three times by calling this task in a `repeat` loop:

```
task automatic pulse_by_task();
  by_task <= 1'b1;
  @(posedge clk);
  by_task <= 1'b0;
endtask
```

It raises a second signal three times through a clocking block, and then counts the rising edges at which each signal was high:

```
module task, <= on the signal:           high at 1 of 3 edges
module task, through a clocking block:   high at 3 of 3 edges
```

Each call's `by_task <= 1` runs after the previous call's `by_task <= 0`, in the same time step, so by the standard the signal stays high at all three edges. Verilator commits the two in source order instead, so `by_task <= 0` takes effect last, and two of the three pulses vanish. `labs/common/README.md` lists the shapes the course measured. Of

those, only the module task called in a `repeat` loop lost assignments. The same task with its three calls written out kept them all, and so did every class driver that was tried. A clocking block gave three of three everywhere, and its drives don't depend on the order in which Verilator commits two nonblocking assignments in one time step.

Check yourself

1. At the 75 ns edge, `edge_test` printed `in_ready cb 0 pin 1`. What was `in_ready` in the cycle that ended at 75 ns, and why would a monitor that used the pin value be wrong?

   Answer: A1, in Answers at the end.

2. A driver executes `vif.cb.in_valid <= 1` right after the edge at 75 ns, and `in_ready` is 1 from then on. When does the pin change, and at which edge does the beat move?

   Answer: A2, in Answers at the end.

3. Lab 02′s monitor holds a `virtual stream_mon_if` rather than the `stream_snk_if` that the sink holds. What does that buy?

   Answer: A3, in Answers at the end.

4. Your monitor reads 0 from every clocking-block input, though the waveform shows the signals going to 1. What's the first thing to look for in your code?

   Answer: A4, in Answers at the end.

Further reading

ChipVerify's pages on interfaces explain each construct at more length. Most of the logs on its SystemVerilog pages come from Cadence simulators rather than Verilator, and the course doesn't reuse its code.

- Interface: an overview of the whole chapter.
- Clocking Blocks Part 1 and Clocking Blocks Part 2: input and output skews, `#1step`, and the race that a clocking block removes. Part 2 has a log whose sample times match its skews.
- Modport: modports with a clocking block in them, as Lab 01′s `alu_if` uses.
- Virtual Interface: how a class gets its handle, and driving through `vif.cb`.
- Scheduling Semantics: the regions of a time step, which decide what "the same moment" means at an edge. Correction: it lists seven regions and leaves out the Re-Inactive and Re-NBA regions, which clocking-block drives depend on.

Next: Threads, mailboxes and events.

Answers

A1. Check yourself

It was 0, which is what the clocking block sampled just before the edge. The pin already showed the value that the flip-flop took at the edge, and that value belongs to the next cycle. A monitor that used it would pair this cycle's `valid` with the next cycle's `ready`, and would see handshakes that didn't happen.

A2. Check yourself

The pin changes at 76 ns, one output skew after the edge. The cycle from 75 ns to 85 ns then has `in_valid` and `in_ready` both 1, so the beat moves at the 85 ns edge, which is when `edge_test` printed `handshake on beat 00`.

A3. Check yourself

Every signal in `stream_mon_if` is an input, so the monitor can't drive the stream it watches. The top module also makes a `stream_mon_if` for the input stream and another for output 0, each wired to that stream's four signals, so one `monitor` class can watch either stream.

A4. Check yourself

A typedef of the virtual interface type. On Verilator 5.050 a class that reaches the interface through `typedef virtual ... t;` reads 0 from every clocking-block input, with no warning. Write the type out in full, as `virtual stream_mon_if`.