The packet router

This page is the specification of the packet router, which is the design you verify in this course. It describes everything the router does at its pins, cycle by cycle, so you shouldn't need to read the RTL to write a testbench for it. From Lab 02 on, your tests run on a faulty build of the router with bugs planted in its RTL, so the RTL wouldn't be a reliable guide to what the router should do anyway. If this page and the router ever disagree, one of them has a bug, and we'd like you to tell us which.

The router takes packets from one input stream and sends each one to one of two output streams, using a routing table that software writes through an APB register port. Each output has its own buffer and its own `ready` signal, so one output can stall without stopping the other. A packet that can't be delivered (its route is disabled, or its output is too full) is dropped whole, and counted.

The RTL is in `labs/common/rtl/`. The top module is `router`.

Sizes

Quantity	Value
Data width of a beat	8 bits
Routes (routing table entries)	4, picked by a 2-bit field in the header
Outputs	2
Buffer per output	16 beats
Longest packet	8 beats, header included
Admission threshold	an output takes a new packet only if it holds 8 beats or fewer
Drop counters	two, 8 bits each, saturating at 255
APB data width	32 bits
APB address width	8 bits (byte addresses), 10 registers

We kept the sizes small on purpose, so that you can reach every corner of the design in a short simulation:

- A beat is one byte, so the header fits in one beat, and you can read a packet straight off a waveform in hex.
- With four routes and two outputs you can cover the routing table completely, because there are only four entries, and each one does one of three things: drop the packet, send it to output 0, or send it to output 1.
- A legal packet is 1 to 8 beats long, so packet length is a coverpoint with eight bins, and the shortest oversize packet is only one beat longer than the longest legal one.
- Each output's buffer holds exactly two maximum-length packets. A packet is admitted only while its output holds 8 beats or fewer, so when an output is stalled and packets keep arriving for it, the first packet to arrive after it holds more than 8 beats is dropped. Getting there takes between 9 beats of traffic (nine 1-beat packets) and 16 (two 8-beat packets).
- The drop counters saturate at 255. A stream of 1-beat packets for a disabled route drops one packet per cycle, so you can saturate the NOROUTE counter in 255 cycles.
- The APB address space has 256 byte addresses, and 246 of them are bad (not in the register map). A read takes 3 cycles, so a test can read all 256 in 768 cycles.

The directed test we wrote for the router covers every section of this page and runs for 6,852 clock cycles.

Ports

Port	Direction	Width	What it is
`clk`	in	1	The clock for the whole design, the APB port included (APB calls it PCLK). Everything happens on its rising edge.
`rst_n`	in	1	Reset, active low and asynchronous (APB calls it PRESETn). See Reset.
`psel`	in	1	APB PSEL
`penable`	in	1	APB PENABLE
`pwrite`	in	1	APB PWRITE: 1 for a write, 0 for a read
`paddr`	in	8	APB PADDR, a byte address
`pwdata`	in	32	APB PWDATA
`prdata`	out	32	APB PRDATA
`pready`	out	1	APB PREADY
`pslverr`	out	1	APB PSLVERR
`irq`	out	1	Interrupt, active high. A level, not a pulse.
`in_valid`	in	1	Input stream: the source has a beat
`in_ready`	out	1	Input stream: the router will take it
`in_data`	in	8	Input stream: the beat
`in_last`	in	1	Input stream: this beat is the last of its packet
`out0_valid`	out	1	Output 0: the router has a beat
`out0_ready`	in	1	Output 0: the sink will take it
`out0_data`	out	8	Output 0: the beat
`out0_last`	out	1	Output 0: this beat is the last of its packet
`out1_valid`, `out1_ready`, `out1_data`, `out1_last`			Output 1, the same as output 0

The APB port has the APB3 signal set and a single slave select. There's no PSTRB (that came with APB4), so every write writes the whole register.

The course simulates the router with Verilator, which has only two states, 0 and 1, and so has no X to mark a value as unknown. A data signal whose `valid` is low still holds an ordinary-looking value, usually the last one it carried, so a testbench must never compare `out0_data` or `out0_last` in a cycle where `out0_valid` is low.

Cycles and handshakes

The tables and rules on this page refer to clock cycles by number. **Cycle n** is the clock period that ends at rising edge n, and the value of a signal "in cycle n" is the value it holds just before that edge. The router samples its inputs at rising edges. Most of its outputs depend only on the router's own flip-flops, so they change just after a rising edge or when `rst_n` falls (see Reset). The exceptions are `pready`, which also depends on `psel`, `penable` and `pwrite` in the same cycle, and `pslverr`, which depends on those three and on `paddr`. Sample those two at the end of the cycle, as you would for any APB slave.

A **handshake** on a stream happens in a cycle where `valid` and `ready` are both 1. The beat moves at the rising edge at the end of that cycle. This course's Python testbenches change inputs right after a rising edge and look at outputs at the falling edge in the middle of the cycle, so no signal is ever sampled at the moment it changes.

Packets

A legal packet is 1 to 8 beats on the input stream. Its first beat is the **header**, and the beat with `in_last` = 1 ends it, so a 1-beat packet is a header with `in_last` = 1.

The header carries the route:

Header bits	Name	Meaning
7:6	`DST`	Which routing table entry to use: 0 to 3
5:0	(tag)	Ignored by the router and passed through unchanged. Testbenches usually put a sequence number here.

Apart from `DST`, the router doesn't look at what any beat contains; it counts the beats and watches `in_last`. Every beat of a delivered packet, the header included, goes to the output unchanged and in the order it arrived, so the packet that comes out is identical to the one that went in. The one exception is an oversize packet, which is cut (see Oversize packets).

Routing

Each of the four routing table registers, `ROUTE0` to `ROUTE3`, has two fields: `EN`, which turns the route on, and `PORT`, which picks output 0 or 1. All four reset to 0, so every route starts disabled.

The router decides what to do with a whole packet once, in the cycle its header is accepted. It reads `ROUTE[DST]` and the level of the output that the entry's `PORT` names, both as they are in that cycle, and then:

`ROUTE[DST].EN`	Level of output `PORT`	The packet is
0	anything	dropped, and counted as NOROUTE
1	9 to 16	dropped, and counted as FULL
1	0 to 8	written, beat by beat, into the buffer of output `PORT`

The **level** of an output in cycle t is the number of beats written into it in input handshake cycles before t, minus the number taken from it in output handshake cycles before t, counting both from the last reset. The level ranges from 0 to 16, and it's what the `LEVEL` register reads. It includes the beat on the output's pins as well as the ones waiting behind it. In example E2, for instance, the level in cycle 4 is 2, because `05` is on `out0_data` and `11` is behind it.

A testbench can track both levels exactly, and that's enough to predict every FULL drop. The beats written into an output are the input beats of the packets that your testbench predicts are routed there (only the first 8 beats of an oversize packet), and the beats taken from it are the handshakes on that output's pins. The decision uses the level in the header's own cycle, so it counts neither the header itself nor a handshake on the output in that same cycle.

Because a packet is admitted only at a level of 8 or less, and no packet writes more than 8 beats, an admitted packet always fits. For example, with output 0 stalled, an 8-beat packet raises its level from 0 to 8, a second 8-beat packet is still admitted (at exactly 8) and fills it to 16, and the next packet for output 0 is dropped. With 1-beat packets, nine are admitted at levels 0 to 8, and the tenth is dropped.

A packet for a disabled route counts as NOROUTE whatever the level of the output its `PORT` names, and it's never counted as FULL as well.

The decision holds for the whole packet. If software writes `ROUTE[DST]` or `CTRL` while a packet is coming in, or either output's level changes, only later packets are affected. A register write applies to headers accepted from the cycle after the write completes; a header accepted in the same cycle that the write completes still uses the old value.

The router still takes a dropped packet from the input, with `in_ready` at 1 for all of its beats, and throws the beats away. Because the input never waits for a full output, a stalled output doesn't hold up packets for the other one.

Dropped packets

When a packet is dropped, at the edge that ends its header's cycle the router

- adds one to `DROP_CNT.NOROUTE_CNT` or `DROP_CNT.FULL_CNT`, unless that counter is already 255, and
- sets `STATUS.NOROUTE` or `STATUS.FULL`.

Both changes are visible from the next cycle on. Example E4 below shows a NOROUTE drop and E5 a FULL drop.

Oversize packets

A packet is **oversize** if its 8th beat doesn't have `in_last` = 1. In the cycle the 8th beat is accepted, the router sets `STATUS.OVERSIZE` (visible from the next cycle), and

- if the packet is being forwarded, it writes the 8th beat with `out_last` = 1, so the output sees an 8-beat packet;
- then it takes and discards every remaining beat of the input packet, up to and including the one with `in_last` = 1, with `in_ready` held at 1.

An oversize packet sets `STATUS.OVERSIZE` once, however long it is. Cutting a packet isn't a drop, so a forwarded oversize packet isn't counted in `DROP_CNT`. A packet that is dropped and is also oversize is counted once, as a drop, and sets `STATUS.OVERSIZE` as well. The next packet after an oversize one is handled normally. Example E6 shows a 10-beat packet cut to 8.

Enabling the router

`CTRL.EN` starts and stops the input, but only at packet boundaries. Between packets, `in_ready` equals `CTRL.EN`. In the middle of a packet `in_ready` is 1, whatever `CTRL.EN` is, so clearing `CTRL.EN` while a packet is coming in lets that packet finish and stops the input after it. Example E7 shows this.

`CTRL.EN` doesn't affect the outputs, so beats already in the buffers keep flowing while it's 0. It resets to 0, so the router takes nothing from the input until software sets it.

The outputs

Each output has a buffer of 16 beats and drives its stream from a register, so `out0_valid`, `out0_data` and `out0_last` come straight from flip-flops.

To work out the cycle in which each beat reaches the pins, number the beats written into one output in order, and let t_k be the cycle of beat k's input handshake and h_k the cycle in which the sink takes it. Beat k is on the output (`out_valid` = 1, with beat k on `out_data` and `out_last`) from cycle

$$a_k = \max(t_k + 2, h_{k-1} + 1)$$

until cycle h_k, and `out_valid` is 0 in every cycle that no beat covers this way. For the first beat after reset, $a_k = t_k + 2$. In particular:

- A beat into an empty output appears two cycles after its input handshake. In example E2 the header is accepted in cycle 2 and is on the output from cycle 4.
- Once beats are waiting, a sink that is always ready takes one per cycle with no gaps, as in E2′s cycles 4 to 6.
- A beat that the sink doesn't take stays on the output until it's taken, and the next beat appears in the cycle after that (E3).

An output stalls only on its own `ready`. While `out0_ready` is low, output 1 carries on at full speed, and the input keeps accepting packets for both outputs. The router drops the ones for output 0 once output 0 is too full.

Valid/ready rules

The rules marked *source* are the testbench's job on the input stream; the router assumes them and doesn't check them. The rest are the router's promises.

A reset ends whatever was in progress. The router lowers `in_ready` and both `out_valid`s at once, with no handshake, and the source drops the beat it was offering (rule I3). So the rules about normal operation hold only while `rst_n` is 1, and an assertion of one of them should be disabled while `rst_n` is 0, with `disable iff (!rst_n)`. The rules about reset itself work the other way round. I7, the first half of O4, the first half of A4 and the reset values say what holds while `rst_n` is 0, so check them without that `disable iff`, sampled at each rising edge.

Most rules are properties over one or two cycles, and each of those becomes a plain assertion. The others need more than a plain assertion:

- O5, O6 and O8 need state kept for each output: the order of the beats, how many beats the current packet has had so far, and the t_k and h_k of each beat. They belong in a scoreboard, or in assertions with tracking code beside them.
- I4, O3 and O7 say what a signal does *not* depend on, and you can't see that in any single trace. A reference model that predicts `in_ready` and `out_valid` from this page checks them, and so does a test that changes an input in the middle of a cycle and confirms the output doesn't move until the next rising edge.
- I3 constrains your stimulus, so it belongs in the driver or the sequence, not in an assertion.

Input stream

- **I1** (source). Once `in_valid` is 1, it stays 1 until the cycle of the handshake, or until a reset.
- **I2** (source). While `in_valid` is 1 and `in_ready` is 0, `in_data` and `in_last` don't change.
- **I3** (source). A beat that the source was offering when `rst_n` fell is abandoned, and the first beat the source offers after the reset is the header of a new packet. (A source that went on offering the old beat would have it accepted as a header once `CTRL.EN` is set.)
- **I4**. `in_ready` doesn't depend on `in_valid`, `in_data` or `in_last` in the same cycle. (It comes from flip-flops.) A source may therefore wait for `in_ready` before raising `in_valid`, though it doesn't have to.
- **I5**. Between packets (after reset, and after a handshake with `in_last` = 1), `in_ready` equals `CTRL.EN`.

- **I6**. In every cycle from a packet's first handshake to its last, `in_ready` is 1, whatever `CTRL.EN`, the routing table and the outputs do, unless `rst_n` falls.
- **I7**. `in_ready` is 0 while `rst_n` is 0.

The source may leave `in_valid` at 0 for any number of cycles, between packets or between the beats of one packet, and packets may also follow each other with no idle cycle at all. The router looks at `in_data` and `in_last` only in cycles with a handshake, so it doesn't notice a source that breaks I1 or I2, and handles whichever beats it took.

Output streams

Each rule holds for output 0 and output 1 separately.

- **O1**. Once `out_valid` is 1, it stays 1 until the cycle of the handshake, or until a reset.
- **O2**. While `out_valid` is 1 and `out_ready` is 0, `out_data` and `out_last` don't change.
- **O3**. `out_valid`, `out_data` and `out_last` don't depend on `out_ready`, or on any other input, in the same cycle. (They come from flip-flops; only a falling `rst_n` changes them between edges.)
- **O4**. `out_valid` is 0 while `rst_n` is 0, and stays 0 after reset until a beat accepted after the reset reaches the output.
- **O5**. An output delivers the beats of the packets routed to it in the order that the router accepted them, unchanged except for the `out_last` of a cut packet. One packet's beats are never interleaved with another's.
- **O6**. Every packet on an output is 1 to 8 beats long and ends with the only beat that has `out_last` = 1, unless a reset cuts it short (see Reset).
- **O7**. Nothing about output 0 depends on `out1_ready`, and nothing about output 1 depends on `out0_ready`.
- **O8**. Beat k of an output is on the output from cycle $\max(t_k + 2, h_{k-1} + 1)$ until the cycle it's taken, and `out_valid` is 0 in every other cycle, as described under The outputs.

The sink has no rules to follow. Its `out_ready` may be high before `out_valid`, may change in any cycle, and may stay low for as long as the sink likes.

APB

A transfer is a **setup cycle** (`psel` = 1, `penable` = 0) followed by one or more **access cycles** (`psel` = 1, `penable` = 1). It completes in the access cycle where `pready` is 1. Writes take one access cycle and reads take two, so a write is 2 cycles long and a read is 3. Example E1 shows a write, a read and a read of a bad address.

In the first access cycle of a read, the router copies the register into an output register, and in the second it drives `prdata` from that copy. So a read returns the register's value as it is in the first access cycle. For a register that changes on its own, like `LEVEL`, this means that a beat written into a buffer at the edge ending the first access cycle isn't in the value you read.

A write takes effect at the rising edge that ends its access cycle. From the next cycle on, the router behaves according to the new value.

A **bad address** is one that isn't in the register map, which includes every address that isn't a multiple of 4. A transfer to a bad address completes normally, with the same timing as any other, but with `pslverr` = 1. A bad write changes nothing, and a bad read returns 0. Writing `ID` or `LEVEL`, or the reserved bits of any register, isn't an error. The transfer completes with `pslverr` = 0, and the router ignores the write. (`DROP_CNT` is read-only too, but a write to it isn't ignored, because any write clears the counters.)

APB rules

The rules marked *master* are the testbench's job; the router doesn't check them.

- **A1** (master). A transfer starts with one setup cycle, and its access cycles follow immediately.
- **A2** (master). `paddr` and `pwrite`, and `pwdata` for a write, stay the same from the setup cycle to the end of the transfer.

- **A3** (master). In the cycle after a transfer completes, `penable` is 0, so the bus is either idle (`psel` = 0) or in the setup cycle of the next transfer. Transfers may follow each other with no idle cycle between them.
- **A4** (master). `psel` is 0 while `rst_n` is 0. A transfer that is in progress when `rst_n` falls is abandoned and has no effect. The master may start it again after the reset, from its setup cycle, or drop it; to the router a restarted transfer is just the next transfer.
- **A5**. In the first access cycle of a write, `pready` is 1.
- **A6**. In the first access cycle of a read, `pready` is 0; in the second, it's 1. This is the only time `pready` is 0.
- **A7**. `pslverr` is 1 only in the cycle in which a transfer to a bad address completes, and 0 in every other cycle.
- **A8**. In the cycle a read completes, `prdata` is the value the register had in the first access cycle, or 0 for a bad address. `prdata` changes only at the edge that ends the first access cycle of a read, when it takes the value that read returns, and when `rst_n` falls, which sets it to 0. So between reads it holds the value of the last read, and writes don't change it.
- **A9**. A write to a good address takes effect at the edge that ends its access cycle, according to the register's access type: read-write fields take the value written, `STATUS` bits written as 1 clear, any write clears `DROP_CNT`, and `ID` and `LEVEL` ignore it. A write to a bad address changes nothing.

Register map

Addresses are byte addresses. Every register is 32 bits, and bits not listed read 0 and ignore writes.

Address	Register	Access	Reset value	Fields
`0x00`	`ID`	RO	`0xD105_0001`	[31:0] the design's ID: design 105, revision 1
`0x04`	`CTRL`	RW	`0x0000_0000`	[0] `EN`
`0x08`	`STATUS`	W1C	`0x0000_0000`	[0] `NOROUTE`, [1] `FULL`, [2] `OVERSIZE`
`0x0C`	`INT_EN`	RW	`0x0000_0000`	[0] `NOROUTE`, [1] `FULL`, [2] `OVERSIZE`
`0x10`	`DROP_CNT`	WC: read-only, any write clears	`0x0000_0000`	[7:0] `NOROUTE_CNT`, [15:8] `FULL_CNT`
`0x14`	`LEVEL`	RO	`0x0000_0000`	[4:0] `LEVEL0`, [12:8] `LEVEL1`
`0x20`	`ROUTE0`	RW	`0x0000_0000`	[0] `EN`, [1] `PORT`
`0x24`	`ROUTE1`	RW	`0x0000_0000`	[0] `EN`, [1] `PORT`
`0x28`	`ROUTE2`	RW	`0x0000_0000`	[0] `EN`, [1] `PORT`
`0x2C`	`ROUTE3`	RW	`0x0000_0000`	[0] `EN`, [1] `PORT`

Every other address is bad and gets `pslverr`: `0x18`, `0x1C`, everything from `0x30` to `0xFF`, and every address that isn't a multiple of 4. That's 246 of the 256.

When you model the registers in pyuvm's register layer in Lab 06, these are the access policies to give the fields:

Fields	UVM access	Volatile
`ID`	`RO`	no

Fields	UVM access	Volatile
CTRL.EN, INT_EN.*, ROUTEn.EN, ROUTEn.PORT	RW	no
STATUS.*	W1C	yes: hardware sets them
DROP_CNT.NOROUTE_CNT, DROP_CNT.FULL_CNT	WC	yes: hardware counts
LEVEL.LEVEL0, LEVEL.LEVEL1	RO	yes: they follow the buffers

ID

Reads 0xD105_0001. Reading it is a quick way to check that the APB port works at all.

CTRL

EN (bit 0) starts and stops the input at packet boundaries, as described in Enabling the router.

STATUS

Each bit records that an event has happened since software last cleared it:

Bit	Name	Set when
0	NOROUTE	a packet is dropped because its route is disabled
1	FULL	a packet is dropped because its output holds more than 8 beats
2	OVERSIZE	a packet's 8th beat isn't its last

A bit is set at the edge that ends the cycle of its event, and it stays set until software writes a 1 to it or the router is reset. Writing 0 to a bit leaves it alone, so writing 0x2 clears FULL and nothing else. If an event sets a bit in the same cycle that a write clears it, the event wins and the bit stays 1. INT_EN doesn't affect STATUS, so the bits record events whether or not their interrupts are enabled.

INT_EN

One enable per STATUS bit, in the same positions. The interrupt is

$$\texttt{irq} = \bigvee_{i=0}^{2} \left(\texttt{STATUS}[i] \wedge \texttt{INT_EN}[i] \right)$$

It's a plain combination of the two registers with no extra delay, so irq changes in the same cycle as the register bit that changes it. In example E4, a NOROUTE drop in cycle 2 makes STATUS.NOROUTE and irq 1 from cycle 3, and the write that clears the bit completes in cycle 7, so both are 0 from cycle 8. irq stays 1 for as long as an enabled bit is set; only clearing the bit, clearing its enable, or a reset lowers it.

DROP_CNT

DROP_CNT holds two counters. NOROUTE_CNT counts packets dropped for a disabled route, and FULL_CNT counts packets dropped for a full output. Each counts a dropped packet once, at its header. A counter that reaches 255 stays there rather than wrapping.

Any write to DROP_CNT clears both counters, whatever value it writes. If a drop happens in the same cycle as the clearing write, it's counted after the clear, so that counter reads 1 afterwards (example E9).

LEVEL

`LEVEL0` and `LEVEL1` are the number of beats that output 0's and output 1's buffers hold, 0 to 16, as described under Routing. Bits [7:5] and [31:13] read 0.

ROUTE0 to ROUTE3

Entry n of the routing table, used by packets whose `DST` is n. `EN` (bit 0) enables the route and `PORT` (bit 1) picks the output: 0 for output 0, 1 for output 1. `PORT` keeps its value while `EN` is 0, but it has no effect then, because a packet for a disabled route is a NOROUTE drop.

Reset

`rst_n` is active low and asynchronous. When it falls, the whole router resets at once, without waiting for a clock edge. In example E8, `rst_n` goes low just after the edge that starts cycle 3, and `in_ready`, `out0_valid` and `LEVEL0` are already 0 in cycle 3. The router does nothing at an edge where `rst_n` is 0, and runs normally from the first rising edge at which `rst_n` is 1.

Change `rst_n` just after a rising edge, both when you pull it low and when you release it, as our testbenches do. Then every cycle is either wholly in reset or wholly out of it, and a monitor that samples at the falling edge never records a handshake that the reset cancels. (If `rst_n` falls in the middle of a cycle, nothing moves at the edge that ends it, whatever `valid` and `ready` were earlier in the cycle.)

Hold `rst_n` at 0 through at least one rising edge at the start of every simulation. Before its first reset the router's state is undefined. On Verilator it happens to start at all zeros, but don't rely on that.

While `rst_n` is 0, and right after it:

Signal or register	Value
`in_ready`	0, and it stays 0 after reset until software sets `CTRL.EN`
`out0_valid`, `out1_valid`	0
`irq`	0
`pready`	1, with the bus idle as rule A4 requires (it's 0 only in the first access cycle of a read)
`pslverr`	0
`prdata`	0
every register	its reset value from the register map; the routing table is all disabled
both buffers	empty: `LEVEL` reads 0

`out0_data`, `out0_last`, `out1_data` and `out1_last` aren't reset; they mean nothing while their `valid` is 0.

A reset with packets in flight loses them. Whatever was in the buffers is gone, including a beat that was waiting on a stalled output, and nothing from before the reset ever comes out afterwards.

- If the sink had already taken some beats of a packet, the rest of it never comes. A scoreboard should throw away everything it expected from before the reset, including the rest of a partly delivered packet, and a monitor that assembles beats into packets should drop a half-assembled one.
- If a packet was half accepted, the beats already taken are discarded with everything else, and the next beat that the router accepts is treated as a header. A source that was part way through a packet must start again with a header (rule I3).

- The configuration is gone too, so software must set up the routing table and `CTRL.EN` again before any packet gets through.

Timing examples

These tables come from runs of the RTL (`labs/common/smoke/tb_examples.sv` prints them). Each column is one cycle, numbered from 1 at the start of the example, and each value is what the signal holds in that cycle. A dash means that the value doesn't matter in that cycle, and the table shows a dash in place of whatever is left on the wires. The tables use a dash for

- `in_data` and `in_last` while `in_valid` is 0, and `out0_data` and `out0_last` while `out0_valid` is 0;
- `paddr` and `pwrite` while `psel` is 0, and `pwdata` outside a write.

Rows marked (reg) show a value inside the router, which is what a read whose first access cycle is that cycle would return. Hex values are written without `0x`. Each example starts from a reset, and its setup is given above its table.

E1: a write, a read and a bad read

Setup: nothing after the reset. The write of `0x1` to `CTRL` completes in cycle 2, so `CTRL.EN` and `in_ready` are 1 from cycle 3. The read of `ID` completes in cycle 5 and the read of the bad address `0x18` in cycle 8, with `pslverr` = 1 and `prdata` = 0. `prdata` changes only at the end of each read's first access cycle (rule A8), so it's `d1050001` from cycle 5 to cycle 7.

cycle	1	2	3	4	5	6	7	8	9
`psel`	1	1	1	1	1	1	1	1	0
`penable`	0	1	0	1	1	0	1	1	0
`pwrite`	1	1	0	0	0	0	0	0	-
`paddr`	04	04	00	00	00	18	18	18	-
`pwdata`	00000001	00000001	-	-	-	-	-	-	-
`pready`	1	1	1	0	1	1	0	1	1
`prdata`	00000000	00000000	00000000	00000000	d1050001	d1050001	d1050001	00000000	00000000
`pslverr`	0	0	0	0	0	0	0	1	0
`CTRL.EN` (reg)	0	0	1	1	1	1	1	1	1
`in_ready`	0	0	1	1	1	1	1	1	1

E2: a three-beat packet through an empty output

Setup: `ROUTE0` is `0x1` (enabled, output 0) and `CTRL.EN` is 1. The header `05` is accepted in cycle 2 and appears on output 0 in cycle 4.

cycle	1	2	3	4	5	6	7
`in_valid`	0	1	1	1	0	0	0
`in_ready`	1	1	1	1	1	1	1
`in_data`	-	05	11	22	-	-	-
`in_last`	-	0	0	1	-	-	-
`LEVEL0` (reg)	0	0	1	2	2	1	0

cycle	1	2	3	4	5	6	7
`out0_valid`	0	0	0	1	1	1	0
`out0_ready`	1	1	1	1	1	1	1
`out0_data`	-	-	-	05	11	22	-
`out0_last`	-	-	-	0	0	1	-

E3: a stalling sink

Setup as in E2, with the same packet, and `out0_ready` low in cycles 5 to 7 and in cycle 9. Beat `11` is on output 0 from cycle 5 to cycle 8 and is taken in cycle 8. Beat `22` is on it in cycles 9 and 10 and is taken in cycle 10. Neither beat changes while it waits (rules O1 and O2).

cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
`in_valid`	0	1	1	1	0	0	0	0	0	0	0	0	0
`in_ready`	1	1	1	1	1	1	1	1	1	1	1	1	1
`in_data`	-	05	11	22	-	-	-	-	-	-	-	-	-
`in_last`	-	0	0	1	-	-	-	-	-	-	-	-	-
`LEVEL0` (reg)	0	0	1	2	2	2	2	2	1	1	0	0	0
`out0_valid`	0	0	0	1	1	1	1	1	1	1	0	0	0
`out0_ready`	1	1	1	0	0	0	1	0	1	1	1	1	1
`out0_data`	-	-	05	11	11	11	11	22	22	-	-	-	-
`out0_last`	-	-	0	0	0	0	0	1	1	-	-	-	-

E4: a NOROUTE drop and a write that clears it

Setup: `INT_EN` is `0x1`, `CTRL.EN` is 1, and every route is disabled. The header `47` (`DST` = 1) is accepted in cycle 2, so `STATUS.NOROUTE`, `NOROUTE_CNT` and `irq` are 1 from cycle 3. The packet comes out of neither output. The write of `0x1` to `STATUS` completes in cycle 7.

cycle	1	2	3	4	5	6	7	8
`in_valid`	0	1	1	0	0	0	0	0
`in_ready`	1	1	1	1	1	1	1	1
`in_data`	-	47	99	-	-	-	-	-
`in_last`	-	0	1	-	-	-	-	-
`out0_valid`	0	0	0	0	0	0	0	0
`out1_valid`	0	0	0	0	0	0	0	0
`STATUS` (reg)	0	0	1	1	1	1	1	0
`NOROUTE_CNT` (reg)	0	0	1	1	1	1	1	1
`irq`	0	0	1	1	1	1	1	0
`psel`	0	0	0	0	0	1	1	0

cycle	1	2	3	4	5	6	7	8
`penable`	0	0	0	0	0	0	1	0
`pwrite`	-	-	-	-	-	1	1	-
`paddr`	-	-	-	-	-	08	08	-
`pwdata`	-	-	-	-	-	00000001	00000001	-
`pready`	1	1	1	1	1	1	1	1

E5: a FULL drop

Setup: `ROUTE0` is `0x1` and `CTRL.EN` is 1. Output 0 is stalled and holds nine 1-beat packets, `01` to `09`, so its level is 9. The packet `3a a0` for `DST` 0 is dropped in cycle 2. The sink starts taking beats in cycle 5.

cycle	1	2	3	4	5	6	7
`in_valid`	0	1	1	0	0	0	0
`in_ready`	1	1	1	1	1	1	1
`in_data`	-	3a	a0	-	-	-	-
`in_last`	-	0	1	-	-	-	-
`LEVEL0` (reg)	9	9	9	9	9	8	7
`out0_valid`	1	1	1	1	1	1	1
`out0_ready`	0	0	0	0	1	1	1
`out0_data`	01	01	01	01	01	02	03
`STATUS` (reg)	0	0	2	2	2	2	2
`FULL_CNT` (reg)	0	0	1	1	1	1	1

E6: a ten-beat packet

Setup: `ROUTE0` is `0x1` and `CTRL.EN` is 1. The 8th beat, `07`, is accepted in cycle 9 without `in_last`, so it goes out with `out0_last` = 1, and `STATUS.OVERSIZE` is set from cycle 10. The router takes beats `08` and `09` and discards them.

cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
`in_valid`	0	1	1	1	1	1	1	1	1	1	1	0	0
`in_ready`	1	1	1	1	1	1	1	1	1	1	1	1	1
`in_data`	-	00	01	02	03	04	05	06	07	08	09	-	-
`in_last`	-	0	0	0	0	0	0	0	0	0	1	-	-
`out0_valid`	0	0	0	1	1	1	1	1	1	1	1	0	0
`out0_ready`	1	1	1	1	1	1	1	1	1	1	1	1	1
`out0_data`	-	-	00	01	02	03	04	05	06	07	-	-	-
`out0_last`	-	-	0	0	0	0	0	0	0	1	-	-	-
`STATUS` (reg)	0	0	0	0	0	0	0	0	0	4	4	4	4

E7: CTRL.EN cleared in the middle of a packet

Setup: `ROUTE0` is `0x1` and `CTRL.EN` is 1. A write of 0 to `CTRL` completes in cycle 4, while the packet `01 11 12 13` is coming in. The packet finishes in cycle 5 with `in_ready` still 1. The next header, `02`, waits from cycle 6 with `in_ready` = 0.

cycle	1	2	3	4	5	6	7	8	9
`psel`	0	0	1	1	0	0	0	0	0
`penable`	0	0	0	1	0	0	0	0	0
`pwrite`	-	-	1	1	-	-	-	-	-
`paddr`	-	-	04	04	-	-	-	-	-
`pwdata`	-	-	00000000	00000000	-	-	-	-	-
`pready`	1	1	1	1	1	1	1	1	1
`CTRL.EN` (reg)	1	1	1	1	0	0	0	0	0
`in_valid`	0	1	1	1	1	1	1	1	1
`in_ready`	1	1	1	1	1	0	0	0	0
`in_data`	-	01	11	12	13	02	02	02	02
`in_last`	-	0	0	0	1	0	0	0	0
`out0_valid`	0	0	0	1	1	1	1	0	0
`out0_ready`	1	1	1	1	1	1	1	1	1
`out0_data`	-	-	-	01	11	12	13	-	-

E8: reset with a beat waiting on output 0

Setup: `ROUTE0` is `0x1` and `CTRL.EN` is 1. Output 0 is stalled with the packet `01 aa` in its buffer, and the packet `02 b0 b1 ...` has had two beats accepted when `rst_n` goes low for cycles 3 and 4. The reset clears everything at once, including the beat `01` waiting on output 0, so nothing comes out when the sink becomes ready in cycle 6. After the reset `CTRL.EN` is 0, so `in_ready` stays 0.

cycle	1	2	3	4	5	6	7
`rst_n`	1	1	0	0	1	1	1
`in_valid`	1	1	1	1	0	0	0
`in_ready`	1	1	0	0	0	0	0
`in_data`	02	b0	b1	b1	-	-	-
`in_last`	0	0	0	0	-	-	-
`LEVEL0` (reg)	2	3	0	0	0	0	0
`out0_valid`	1	1	0	0	0	0	0
`out0_ready`	0	0	0	0	0	1	1
`out0_data`	01	01	-	-	-	-	-
`CTRL.EN` (reg)	1	1	0	0	0	0	0

E9: a write to DROP_CNT in the same cycle as a drop

Setup: `CTRL.EN` is 1, every route is disabled, and two drops have made `NOROUTE_CNT` 2. A write to `DROP_CNT` completes in cycle 2, the same cycle in which a 1-beat packet for a disabled route is accepted. The counter is cleared and then counts the drop, so it reads 1.

cycle	1	2	3	4
`psel`	1	1	0	0
`penable`	0	1	0	0
`pwrite`	1	1	-	-
`paddr`	10	10	-	-
`pwdata`	00000000	00000000	-	-
`pready`	1	1	1	1
`in_valid`	0	1	0	0
`in_ready`	1	1	1	1
`in_data`	-	00	-	-
`in_last`	-	1	-	-
`NOROUTE_CNT` (reg)	2	2	1	1
`STATUS` (reg)	1	1	1	1

Building the router

The RTL is five files in `labs/common/rtl/`. `router.f` lists them in compile order, with the package first, and its paths are relative to its own folder, so pass it to Verilator with `-F` (not `-f`):

```
verilator --lint-only -Wall -F labs/common/rtl/router.f --top-module router
```

That lint passes with no warnings. The files have no `` `timescale `` and the design has no delays, so it runs at any clock period. Our builds pass `--timescale 1ns/1ps`, and our testbenches use a 10 ns clock.

File	What it holds
`router_pkg.sv`	Sizes, register offsets and the ingress state type
`router_fifo.sv`	One output buffer and the stream it drives
`router_ingress.sv`	The input stream: route lookup, admission, drops and the length limit
`router_regs.sv`	The APB port and every register
`router.sv`	The top module, `router`, which connects the four

The labs also use a **faulty build** of the router, made by `labs/common/inject.py --all --out DIR`. It compiles from `DIR/router.f` in exactly the same way and has the same ports, and with no bug switched on it behaves exactly like the RTL above. With `+BUG=Bnn` on the simulator's command line, it switches on one of the course's planted bugs at time 0, so you can change `+BUG` without rebuilding. An unknown value of `+BUG` stops the simulation with a fatal error. The faulty build prints nothing about which bug is on, so a log can't give it away. If you add `+BUG_ECHO`, it prints one line at time 0 that names the bug, or says that none is on.