Flare-On 11 Challenge 6: bloke2

By Dave Riley (@6502_ftw)

Overview

The sample provided for the challenge is an archive full of Verilog files, a Makefile, and a README. The prompt in the README reads:

```
Unset
bloke2
======

One of our lab researchers has mysteriously disappeared.  He was working on the
prototype for a hashing IP block that worked very much like, but not
identically to, the common Blake2 hash family.  Last we heard from him, he was
working on the testbenches for the unit.  One of his labmates swears she knew
of a secret message that could be extracted with the testbenches, but she
couldn't quite recall how to trigger it.  Maybe you could help?


Details
-------

This source code is written in the Verilog hardware description language.  It
is written in the subset of SystemVerilog supported by the free and easily
available Icarus Verilog simulator (available under various package managers as
either `iverilog` or `icarus-verilog`).  It is not tested on, nor is it
guaranteed to run on, any other Verilog simulators (and in fact, I would not
recommend trying, as Icarus is... somewhat more permissive in some aspects than
a lot of commercial tooling).

The files can be built via GNU Make with the accompanying `Makefile`; simply
`make` to build the total source and check for errors, or `make tests` to build
and run the testbenches.  If you don't have GNU Make, you should be able to
build it with `iverilog -g2012 -s <top_module> *.v` followed by `vvp a.out` to
run the testbench.  For example:

```
iverilog -g2012 -s f_sched_tb *.v
vvp a.out
```

You should be able to get to the answer by modifying testbenches alone, though
```

```
there are some helpful diagnostics inside some of the code files which you
coulduncomment if you want a look at what's going on inside.  Brute-forcing
won't really help you here; some things have been changed from the true
implementation of Blake2 to discourage brute-force attempts.



Happy hardware hacking!

- Dave
```

Figure 1: README

This challenge is slightly outside the norm for reversing challenges; it's definitely not x86 (though we have a few of those every year), and it's not really even machine code. It's closer in some senses to some of the JavaScript or Python challenges (which we also have every year), not least because the full source code is included. I have done the courtesy of not obfuscating the code, which is definitely because I am nice and not because I was too lazy to look for or build a Verilog obfuscator.

Some notes:

- For the curious, I chose Verilog over VHDL (which I actually mostly like better) because Icarus Verilog is generally easier to install and use than any of the open-source VHDL options. Relatedly, this is mostly strict Verilog and not SystemVerilog, largely because Icarus Verilog doesn't truly support SystemVerilog beyond a few of the simpler constructs, but I did use a few SystemVerilog constructs especially in the testbenches. Purists, beware.
- As hinted in the README, this does not actually implement BLAKE2, though it comes pretty close structurally. There are a few things in it that are switched around from the true implementation, partly because I didn't want people to be tempted to brute-force the answer (which would be painful for all involved) and partly because it was not really worth the work to get it fully working for a CTF challenge where strict accuracy didn't matter.
- I did not run this through any actual FPGA synthesis tools; there's a good chance that there are some constructs in there that won't synthesize clearly, since Icarus is sometimes more permissive than others. If you wanted to try to run this in actual hardware, you may need to do some cleanup as well.

The Challenge

Okay, let's dive in. We have some instructions, and I'm going to assume you're running on a system with GNU Make (which should be most Unix/Linux based systems and a number of Windows systems as well). Per the README, we'll run `make` and see what we get.

```
Unset
$ make
iverilog -g2012 -o bloke2.out bloke2.v f_sched.v f_unit.v g_over_2.v g.v g_unit.v data_mgr.v
bloke2s.v bloke2b.v
```

Figure 2: Running make

Not much there, though it looks like it builds cleanly. How about the tests?

```
Unset
$ make tests
iverilog -g2012 -o f_sched.test.out f_sched.v f_sched_tb.v
vvp  f_sched.test.out
iverilog -g2012 -o bloke2b.test.out bloke2.v f_sched.v f_unit.v g_over_2.v g.v g_unit.v
data_mgr.v bloke2s.v bloke2b.v bloke2b_tb.v
vvp  bloke2b.test.out
Received message: 7g       ◆A◆&◆377◆◆S◆◆3◆◆
                                    ◆◆◆2◆◆◆&E◆}'<◆◆◆Y|N◆◆:'◆ĸ◆◆,◆◆    ◆6◆A◆◆C6
Received message: ◆q]y8◆◆B◆◆y◆◆#>[◆◆qi∓wRg◆◆:◆◆◆0◆◆◆◆$E◆◆◆w◆5CU◆◆M-◆◆◆
Received message: 0◆◆+(s3◆◆◆◆ɟ◆◆'pɣ◆$◆vg◆x◆Ж◆C◆_◆je◆◆◆◆◆◆0◆◆3ym◆&◆hA◆◆◆◆Q
iverilog -g2012 -o bloke2s.test.out bloke2.v f_sched.v f_unit.v g_over_2.v g.v g_unit.v
data_mgr.v bloke2s.v bloke2b.v bloke2s_tb.v
vvp  bloke2s.test.out
Received message:                                ◆◆F3◆◆p◆◆◆ʌ◆p◆{3xM◆%◆◆◆=W◆◆◆
Received message:                                n◆◆(◆     ◆ ◆F◆r◆c◆◆@lu◆s◆fFvr
Received message:
                                                 ◆◆a$◆a◆b◆◆}◆◆◆\◆H◆◆O?w◆`?εb
rm f_sched.test.out bloke2s.test.out bloke2b.test.out
```

Figure 3: Testing

Well, that's something. Looks like we build one quiet test, and then two very noisy ones. It might be worth investigating those two.

The bloke2[sb] testbenches

The two testbenches for `bloke2s` and `bloke2b` are nearly identical. The only difference (aside from the module names) is that one instantiates (and tests) the `bloke2s` module while the other one does the same for `bloke2b`. If you're curious, you can even do a diff of the two to show there's nothing sneaky going on:

```
Unset
$ diff -u bloke2s_tb.v bloke2b_tb.v
--- bloke2s_tb.v    2024-09-11 13:23:53.000000000 -0400
+++ bloke2b_tb.v    2024-09-11 13:23:07.000000000 -0400
@@ -1,6 +1,6 @@
 `timescale 1ns/1ps

-module bloke2s_tb;
+module bloke2b_tb;

       reg clk = 1'b0;
       reg rst = 1'b1;
@@ -23,7 +23,7 @@
       localparam CLK_PD = 10;
       always @clk #(CLK_PD/2) clk <= ~clk & ~done;

-      bloke2s uut (
+      bloke2b uut (
              .clk(clk),
              .rst(rst),
```

Figure 4: bloke2s and bloke2b diff

I'm not going into great detail on the actual test module here; those familiar with HDL testbenches will probably find much of it familiar. Some of the salient features for those who aren't, however:

```
Unset
       // Assert when done to finish the simulation.
       reg done = 1'b0;

       // Clock driver.
       localparam CLK_PD = 10;
       always @clk #(CLK_PD/2) clk <= ~clk & ~done;
```

Figure 5: Clock driver

Here we have a clock driver. There is a `localparam` (basically a constant) defining the clock period as 10 ticks (which are defined as 1ns in the `` `timescale `` directive at the beginning of the file), and the **`always`** block is a Verilog construct which basically means, "any time `clk` changes, do this". We are then telling it to wait half a `CLK_PD` and then toggle the `clk` line, unless the done line is asserted.

We do this because HDL simulators are actually queued event simulators; they don't just count up the ticks and do actions like you might see in an emulator, they actually have a queue of events to come at given timestamps, which then often generate more events for the queue, and so on. When there are no more events in the queue, the simulation ends. This can also be forced in Verilog with the `$finish` procedure, but

it's often more elegant to do it this way, as `$finish` is more of a forced exit. If you were piling multiple testbenches together in a bigger module, this would keep one testbench from ending the whole simulation.

```
Unset
bloke2s uut (
              .clk(clk),
              .rst(rst),

              .start(start),
              .finish(finish),

              .din(din),
              .din_valid(din_valid),
              .din_ready(din_ready),
              .din_end(din_end),

              .dout(dout),
              .dout_valid(dout_valid),
              .dout_end(dout_end)
       );
```

Figure 6: Instantiation of unit under test

Here we have the instantiation of the unit under test (uut). The various wires and registers that have been declared above are connected to ports on the module. In this way, HDLs are really more like textual descriptions of schematics than actual programming languages (though there are also behavioral blocks that are definitely more like programming languages, as we'll soon see). There's nothing particularly special about this instantiation, but it's a decently clean example.

You can see the overall layout of the module ports here, though: there's the typical clock and reset (hardware devices can come up with unpredictable internal states, so there is usually a reset line to allow them to be reset to a normal initial state, just like any chip on a board), a pair of start and finish strobes, a data in block (with valid, ready, and end indicators), and a data out block (with valid and end indicators; this indicates that the module assumes the receiver is always ready to receive, which might not be a valid assumption in the real world but works for us).

The testbench's job is to control these lines to drive the module as a test harness performing unit and sometimes integration tests, and is fairly analogous to test files in other languages like Java or Go (though the reporting mechanisms are often somewhat more DIY than in many modern programming languages unless you're using a formal framework like UVM or OVM, which are well beyond the scope of this project and not supported by Icarus Verilog anyway).

```
Unset
// Monitor for the message received from the UUT.
	reg [511:0] message;
	always @(posedge clk) begin
		// Very noisy diagnostic, but useful
		//$display("%t tb dv %b de %b m %h", $time, dout_valid, dout_end, message);

		// If we're starting, clear the message contents.
		if (start) message = "";

		// If we have valid data out, shift it in.
		if (dout_valid) message = {message[503:0], dout};

		// If the data finished, display what we received.  It's a hash, so it's
		// probably garbage, but you never know...
		if (dout_end) $display("Received message: %s", message);
	end
```

Figure 7

Here's the first one that's actually a bit interesting. We have another always block triggering on the positive edge of `clk` which does a few things:

- If the `start` strobe is asserted, it resets the message to an empty string (Verilog proper doesn't really have great string support, it's just a vector of bits; SystemVerilog has better strings but only newer versions of Icarus Verilog support them, and some older versions we tested had bugs, so we use the old style here).
- If the `dout_valid` strobe is asserted, we shift a byte from the module's output port into the `message` bit vector. This is a left shift; you can see we take the rightmost n-8 bits from the existing vector and concatenate them with the new data.
- If the `dout_end` strobe is asserted, we display our accumulated message. That's an interesting comment, isn't it? In any case, it's worth noting that we're displaying it as a text string rather than hex, which is what you'd normally expect from a hash.

Note that unlike some assignment statements, all of these use the = operator (blocking assignment) rather than the <= operator (non-blocking assignment); if you're curious what this means, feel free to look it up and figure out why the message output would be off by a byte if we used the non-blocking assignment.

Note also that there's a commented out $display line at the top. That can be a pretty useful diagnostic for seeing exactly what's going on at every clock, though it's also quite noisy. There are a few of those sprinkled throughout. There are other ways of getting detailed data out and viewing them with a waveform viewer, which I'll leave you to look up, but I was already requiring one additional application to be installed and didn't want to inflict GTKWave on anyone, so I made it text-friendly.

```
Unset
	// Run a string through the hash machine.
	task hash_message (
		input [1023:0] msg
	);
		integer i, len;
	begin
		// Ugh, working around a bug in old Icarus Verilog 10.x which
		// prevents the use of proper strings here.  Get the string
		// length because the string literal comes in backwards.
		for (i = 0; msg[8*i +: 8] != 0; i++); len = i;

		// Set our start and finish lines correctly.
		start <= 1'b1;
		finish <= 1'b0;
		@(posedge clk);
		start <= 1'b0;
		finish <= 1'b0;

		// Wait until it signals readiness.
		while (!din_ready) @(posedge clk);

		// Write it in byte by byte when drdy is asserted.
		for (i = len - 1; i >= 0; i--) begin
			din <= msg[8*i +: 8];
			din_valid <= 1'b1;
			do @(posedge clk); while (!din_ready);
		end

		// Deassert data valid, since we're done.
		din_valid <= 1'b0;

		// And assert finish for a clock.
		finish <= 1'b1;
		@(posedge clk);
		finish <= 1'b0;

		// Wait until the message processes.
		@(negedge dout_end);
	end endtask
```

Figure 8

The next bit is a `task`, which is like a function, but in Verilog functions are only pure functions (i.e. no side effects, including the fact that they can't take time, think of them more like lambdas or pre-processing functions) while tasks are more like traditional procedures (in fact, in VHDL, the analogous construct is called a procedure). This particular task does the job of performing a single test of the unit, running a given string

through the unit and waiting until it is done processing. A normal testbench will often have several of these for testing out different scenarios to avoid repetitive coding of line manipulations, etc.

Here you can see one bit vector as input which gets filled with the string. As noted in the comment, previously this used a string, which is one of the few SystemVerilog constructs Icarus supports (which makes testbenches like this much nicer), but there was a bug in all versions of 10.x, which some slightly older Linux distributions (looking at you, Ubuntu 20.x LTS) still have as their packaged version, so there is a workaround to make this behave on that. The first line gets the string length.

Next, the task sets the start and finish lines appropriately and waits a clock, then deasserts both. This gets the data in motion. It then waits for the din_ready line to be asserted, indicating that the unit is ready to receive data. After that, it shifts the data in one byte at a time from the string, waiting until a positive clock edge when the unit asserts it's ready for data before moving on to the next one.

Once the string is all shifted out, we assert the finish strobe, deassert din_valid, and then deassert finish after a clock and wait for the output data to fully shift out from the unit before ending the task. Thus the task pushes a string through the unit and waits until it's finished processing, allowing us to push all sorts of strings through the unit.

```
Unset
	initial begin
		// Initialize our start and finish lines.
		start <= 1'b0;
		finish <= 1'b0;

		// Kick off the clock, since it needs an event to start.
		clk <= 1'b1;

		// Wait for the next clock edge and pull the reset.
		@(posedge clk);
		rst <= 1'b0;

		// Run a few tests.
		hash_message("");
		hash_message("123");
		hash_message("abc");

		// And we're done, stop the clock.
		done <= 1'b1;
	end
```

Figure 9

The rest of the testbench is pretty short. The initial block in Verilog is like the always block, except it only runs once and is initiated by the start of simulation, not any other event (technically always blocks aren't actually initiated by events either, they just run on repeat, but they often have a statement saying to wait for an event at the beginning, so it's easy to think of them that way). They're often not really synthesizable into hardware (except for a few select constructs, like initializing memory from hex files), which is why we still have rst lines and the like. In the case of many testbenches, it's what contains the actual high-level test procedure, and this is no different.

First, this initializes the start and finish lines (if you don't do this, they start as x, which is the indeterminate state, which is an important concept for simulation but is not an especially helpful state for the lines to be in at startup!) and kicks off the `clk` line, which is important because the clock driver waits for an event to start! You'll note if you looked at the declarations that the `clk` and `rst` lines had initial values, which actually get set before the initial block begins, so `rst` is not initialized here, though in a lot of cases it could be.

After the first clock edge, we pull `rst` low to take the part out of reset, then feed the data through and set the `done` line to terminate the simulation. In a real testbench, we'd probably also provide expected data (for example, known hash values for the empty string and various test strings, etc.), but we don't here because this is a toy problem!

So anyway, having looked at this, we have a testbench which will probably output the answer once some condition is met (well, 2, actually), but we don't know how to trigger it. Let's look deeper.

The bloke2[sb] modules

The bloke2s module looks like this:

```
Unset
module bloke2s (
       input        clk,
       input        rst,

       input        start,
       input        finish,

       input  [7:0] din,
       input        din_valid,
       output       din_ready,
       input        din_end,

       output [7:0] dout,
       output       dout_valid,
       output       dout_end
```

```
);

        bloke2 #(
                .W(32),
                .R0(16),
                .R1(12),
                .R2(8),
                .R3(7),
                .ROUNDS(10),
                .IV(256'h6a09e667bb67ae853c6ef372a54ff53a510e527f9b05688c1f83d9ab5be0cd19)
        ) b2s (
                .clk(clk),
                .rst(rst),

                .start(start),
                .finish(finish),

                .din(din),
                .din_valid(din_valid),
                .din_ready(din_ready),
                .din_end(din_end),

                .dout(dout),
                .dout_valid(dout_valid),
                .dout_end(dout_end)
        );

endmodule
```

Figure 10: bloke2s module

Not much to see here, this just instantiates the `bloke2` module with some parameters (essentially pass-through constants used to customize a module) set to a 32-bit word size, and all the other pieces set pretty much the same as they are set for BLAKE2s. The `bloke2b` module is pretty similar, but with features corresponding to BLAKE2b.

The bloke2 module

I'm not going to paste the entire module here, but if you read along in the code, you'll see a few things:

- An `f_unit` module, `fu`, which contains the main hashing logic of the module
- A `data_mgr` module, `dmgr`, which manages the data flow between the outside and the hashing logic
- A state machine which looks like it manages various modes of the system based on the `start` and `finish` lines

Let's take a closer look at the state machine.

```
Unset
always @(posedge clk) begin
              if (rst)
                     state <= STATE_INIT;
              else begin
                     // Defaults
                     case ({start, finish})
                            2'b00:
                                   state <= STATE_DATA;
                            2'b01:
                                   state <= STATE_FINISH;
                            2'b10:
                                   state <= STATE_INIT;
                            2'b11:
                                   state <= STATE_INIT;
                     endcase

                     case(state)
                            STATE_FINISH:
                                   if (start) begin
                                          state <= STATE_INIT;
                                          if (finish)
                                                 state <= STATE_TEST;
                                   end
                     endcase
              end
       end
```

Figure 11

This is all very interesting, and it looks like if we were to assert start and finish at the same time when it's already in STATE_FINISH, we get to a state labeled STATE_TEST. However, there's a catch: the state register doesn't actually connect anywhere else in this module. It's a bit of a red herring.

(If you're curious why this is here, it's because I'd initially built this as a state machine for controlling the logic, but later moved to simpler logic in the data manager to control it, but I left this in because it hinted at the ultimate solution while still being misleading.)

Let's dig a little deeper.

The f_unit module

I'm not going deep into detail on this because this part is actually mostly pure BLAKE2 logic. It's nearly all combinational, meaning it is straight input to output with no clocking or registers in between, except at the

very top level where the hash state register keeps track of the actual hashing state. Data is presented as the full message block bit vector on start. The output is the state vector halves XORed together with the original message as per the specification. There are some counters to keep track of rounds and sub-rounds, which trigger strobes when they're counted down and also index various lookup tables for permutations, etc.

This is how you might make a first cut at the hashing unit if you were actually implementing it in hardware. It probably wouldn't be very fast, because the path through the functional units is somewhat long, which means there would be a long electrical delay between when the state register changes and when the output of all the hashing machinery is ready to be clocked back into the state register (which has to be ready a certain setup time before the rising clock edge; this is what typically sets the maximum clock speed for a given digital circuit). If you wanted to make it faster, you could pipeline some of the data (put intermediate registers in the path so the data gets worked on a chunk at a time like an assembly line), but that makes for more complicated logic and you have to make sure the data lines up properly at the end, and it's a lot more work.

In any case, no easter eggs here, but if you're curious how some elements of digital logic work in Verilog, you can read the code. There's also one testbench for the `f_sched` unit (which is responsible for scheduling the permutations for each round and sub-round of the algorithm), which doesn't have anything fun for us, but you can see an example of a lower-level unit test bench if you wanted to look at it.

The data_mgr unit

The data manager is responsible for all data flow in and out of the overall system. It adapts the byte-wide input and output ports to the message-block-wide inputs and outputs of the `f_unit` module, and provides data flow control signals for the external modules to manage the flow of data. It also happens to contain the easter egg we're looking for.

After the various register and wire declarations, there are two always blocks performing two separate processes (roughly speaking, the data-in and data-out processes, as denoted in the diagnostic $display lines). Here's the din process:

```
Unset
	always @(posedge clk) begin
		if (rst | start) begin
			m   <= {MSG_BITS{1'b0}};
			cnt <= {CNT_BITS{1'b0}};
			t   <= {(W*2){1'b0}};
			f   <= 1'b0;
			tst <= finish;
		end else begin
			if (dv_in) begin
				m[((W-cnt)*8) +: 8] <= data_in;
```

```
                        cnt             <= next_cnt[CNT_BITS-1:0];
                        t               <= t + 1;
                        f               <= finish;
                        //$display("%t dmgr din d %h m %h c %h t %h f %b t %b", $time,
data_in, m, cnt, t, f, tst);
                    end else if (finish) begin
                        f <= 1'b1;
                    end
                end
        end
```

Figure 12

This is all pretty straightforward. Instead of a shift register, the message block is treated as an addressable array of bits (which would be a lot less efficient in hardware than a shift register, but the way the endianness meshed with the data alignment made this make more sense, and again, we're not synthesizing it, so it's fine). The t and f inputs to the BLAKE2 algorithm are also presented here. The only thing that's a bit out of place is the `tst` bit.

Between the two processes, we see this unusual thing:

```
Unset

        localparam TEST_VAL =
512'h3c9cf0addf2e45ef548b011f736cc99144bdfee0d69df4090c8a39c520e18ec3bdc1277aad1706f756affca4
1178dac066e4beb8ab7dd2d1402c4d624aaabe40;
```

Figure 13

That's probably related to what we're looking for. More on that in a bit.

The second process (the `dout` process) looks like this:

```
Unset
        always @(posedge clk) begin
            if (rst) begin
                out_cnt <= 0;
            end else begin
                //$display("%t dmgr dout oc %h", $time, out_cnt);
                if (h_rdy) begin
                    //$display("%t dmgr dout h %h t %b", $time, h_in, tst);
```

```
                        out_cnt <= W;
                        h <= h_in ^ (TEST_VAL & {(W*16){tst}});
                    end else if(out_cnt != 0) begin
                        //$display("%t dmgr dout d %h dv %b de %b oc %h", $time, data_out,
dv_out, data_end, out_cnt);
                        out_cnt <= out_cnt - 1;
                        h <= {8'b0, h[W*8-1:8]};
                    end
                end
            end
```

Figure 14

This, too, is mostly straightforward. The output is a plain right-shift register, and the output message is strobed in when h_rdy (the strobe from the f_unit module indicating that a new block is done) is asserted, resetting the output counter as well.

However, there's a little bonus on the message input: it's XORed with `(TEST_VAL & {(W*16){tst}})`. What exactly does that mean? `{(W*16){tst}}` means a vector of W*16 copies of the tst bit (why times 16 instead of 8? I think that was a typo, actually, but it didn't generate a warning and it doesn't break anything, so here we are). It's ANDed with TEST_VAL, so if `tst` is asserted, the vector is TEST_VAL and if not, it's zero. Recall that XORing with zero does nothing, so this is essentially a conditional XOR that alters the contents of the message if tst is asserted when `h_rdy` is asserted. This is probably the condition we're looking for.

So how do we get that bit asserted at the proper moment? Well, let's go back to the `din` process for a bit. Note that the `tst` bit is only ever set when `rst` or start is asserted, so it's set mainly at the beginning of a new message. What is it set to?

```
Unset
                    tst <= finish;
```

So the way to trigger the test mode is to simultaneously assert `start` and `finish` at the beginning of a message. Normally, this would be considered an invalid set of inputs, and a well-designed module would either have better designed inputs to avoid that possibility, or have the behavior specified when an invalid input combination is given (or the common cop-out, "undefined behavior", which ultimately leads to problems down the road when people find out that the undefined behavior does interesting things and start to count on it). If we look through the rest of the logic, there's no other effect from asserting `start` and `finish` on the same clock, so it seems like we should be able to try that.

Back to the testbenches

Okay, we have a plan. Fortunately, the design of the testbench makes it extremely easy to try! Recall that the `hash_message` task starts like this:

```
Unset
            // Set our start and finish lines correctly.
            start <= 1'b1;
            finish <= 1'b0;
            @(posedge clk);
            start <= 1'b0;
            finish <= 1'b0;
```

Figure 15

Conveniently, there's already a finish assignment for us to alter. In some ways, this is redundant, because finish should theoretically already be 0 when this task starts, but it's not a bad safety margin to throw in in case a previous task didn't clean up properly, etc. The second finish <= 1'b0 is definitely redundant, but I figured an extra hint wouldn't hurt, plus it made things much easier to test. Let's alter it to assert start and finish on the same clock.

```
Unset
            // Set our start and finish lines correctly.
            start <= 1'b1;
            finish <= 1'b1;
            @(posedge clk);
            start <= 1'b0;
            finish <= 1'b0;
```

Figure 16

If you make this alteration on both testbenches, you'll have the following output when you run make tests:

```
Unset
$ make tests
iverilog -g2012 -o f_sched.test.out f_sched.v f_sched_tb.v
vvp  f_sched.test.out
iverilog -g2012 -o bloke2b.test.out bloke2.v f_sched.v f_unit.v g_over_2.v g.v g_unit.v
data_mgr.v bloke2s.v bloke2b.v bloke2b_tb.v
vvp  bloke2b.test.out
Received message: wh+C
                    fD J Th 1 Pv ~\b' 6 ` B 0 {*" [ o K

Received message: X mB:V i c![/ wK' ( <Q \
```

```
                                                    =ϕ.ϕϕW ϕϕJZ ϕϕϕh ϕϕϕB ϕ?\ϕ
Received message: please_send_help_i_am_trapped_in_a_ctf_flag_factory@flare-on.com
iverilog -g2012 -o bloke2s.test.out bloke2.v f_sched.v f_unit.v g_over_2.v g.v g_unit.v
data_mgr.v bloke2s.v bloke2b.v bloke2s_tb.v
vvp  bloke2s.test.out
Received message:                                  ϕJϕ
                                                      Q ϕϕϕ0]_ϕrD ϕϕϕ  \F.ϕ?ϕ*ϕϕϕ-ϕ
Received message:                                  .Tba ϕ2I(ϕxqϕ
ϕ;ϕϕϕف`u ϕϕ<)ϕϕ
Ṙeceived message:                                  L#ϕϕm,.d ϕϕϕϕϕϕlb$ ϕϕP ϕq ϕϕE ϕt ϕ
```

Figure 17

And there we have the flag we were looking for:
`please_send_help_i_am_trapped_in_a_ctf_flag_factory@flare-on.com`

Closing notes

If you look at the testbench, this is the output for bloke2b given the input ”123” as modified by the data manager unit. There were a few things that made this a little harder to find statically by just XORing the TEST_VAL with known strings:

- The value is XORed with the internal message representation, which is shifted out such that the string itself winds up byte-reversed
- The hash value produced isn’t actually the same as BLAKE2, so even though ”123” is given as a test vector value by the BLAKE2 RFC going through all the internal steps, you can’t just XOR the result even if you had the byte order correct

Now, you can bypass the need to modify the testbench by just setting tst to 1 in data_mgr.v. I considered strengthening things to break that as a bypass, but decided against it because I thought it might make the actual solution even more obvious, and if you figured out that you needed to do it, you were almost there anyway, so I’d consider that a mostly-valid solution as well, but know that you can get there solely by changing the inputs themselves instead of modifying the circuitry. I hope you enjoyed this challenge as much as I enjoyed writing it! If you’re ever interested in getting into FPGA work, learning VHDL and/or Verilog is a good step to start, and you’re now part of the way toward understanding it.

Final Flag

```
Unset
please_send_help_i_am_trapped_in_a_ctf_flag_factory@flare-on.com
```