

hi :P

goals

my mango (my main goal) is $O(n+L)$ where L is some initial (constant) latency
I want the cycles required to get the solution to be \sim =number of inputs

I'm not super worried about area or resource usage, since my solution sort of trades these for latency

theory

```
im not restating the problem :P
```

```
It's day1
```

first we consider our input

my puzzle input seemed to be within this range:

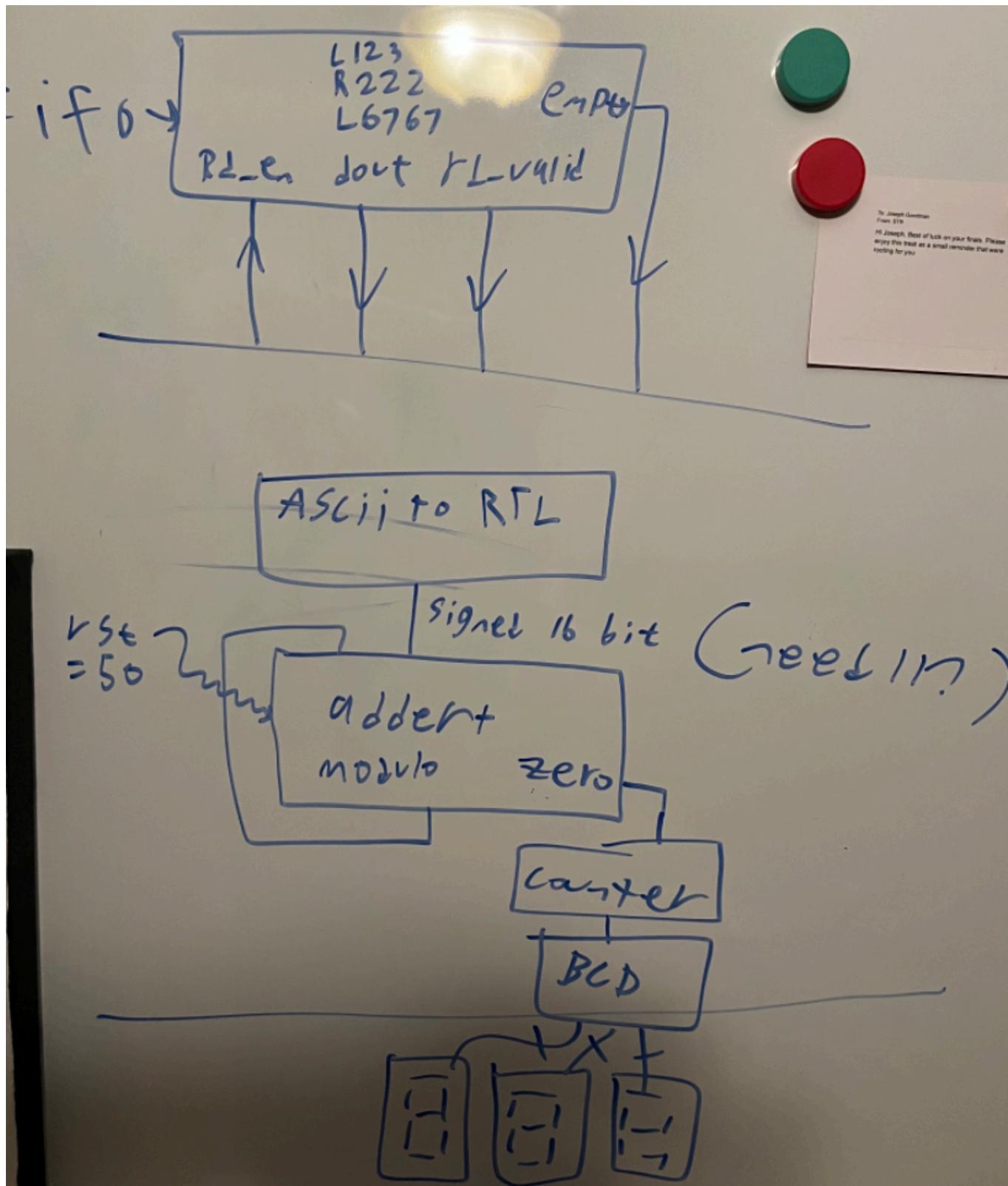
R | 2 3

L 2 4 8

R | ... R 9 9 9
L | ... L 9 9 9

making our pipeline

annoyingly, this problem isn't super suited to FPGA speed-ups. We need to know where the dial is in order to rotate it to its next position; I thought of this simple, two stage pipeline.



ground breaking stuff, I know

What is this art piece trying to convey?

Data comes from memory and there is a FIFO in between the memory and our pipeline (usually we'd have the same on the output side, but for this problem the output is one unsigned number)

Once we read the data from the FIFO, it goes through a two stage pipeline:

1. convert the ascii representation of our data into a signed 16-bit number (this will involve some super "fun" bcd conversions)
2. next we have an adder+modulo component
3. count the number of zeros, display on 7segment displays

in order to make modulo you must first invent the universe

Unfortunately, it's not as simple as saying "we have modulo."

For this particular problem we need to compute

```
acc = (acc + current_input) % 100 # for each input
```

In other words, to achieve the goal of ~1 cycle per input, we have to be able to compute $a \% 100$ for (basically) any a in 1 cycle.

can't be that hard, right?

VHDL literally has a keyword for this, how hard can it be??

Let's ask CHAT-GPT

Is it possible to do `mod` in one cycle?

Short answer

Yes, but with important caveats.

More precise answer

Situation	One-cycle possible?	Notes
<code>mod 2ⁿ</code>	<input checked="" type="checkbox"/> Always	Just bit slicing
Constant modulus	<input checked="" type="checkbox"/> Usually	Timing may be bad
Variable modulus	<input type="checkbox"/> ! Technically yes	Often impractical
High-speed designs	<input checked="" type="checkbox"/> X Usually	Needs pipelining

What does *not* happen automatically

VHDL will **not**:

- automatically pipeline the operation
- automatically make it multi-cycle
- guarantee timing

This is your job as the designer.

"un-rolling" mod operator

Before I said "this problem isn't super suited to FPGA speed-ups," but that isn't 100% true. We can break the mod operation up into a bunch of smaller cases. Based on these cases, we can simplify the operation.

Luckily, we have some constraints on our input

smallest state: 0

biggest state: 99

most negative input = -999

most positive input = 999

thus

smallest state (pre mod) = -999

biggest state (pre mod) = 99+999 = 1098

positive case

if smaller than 100, do nothing

if smaller than 200, subtract 100

if smaller than 300, subtract 200

if small than $n \times 100$, subtract $(n-1) \times 100$

negative case

we could also use this:

$-a \% 100 = 100 - a \% 100$

^we already computed this

but then our worst case would look like this

1. take two's comp of some negative number
2. take mod of that
3. subtract from 100

i'd rather the positive and negative cases have ~same path length rather than making the negative case the critical path

if bigger than or equal to -100, add 100

if bigger than or equal to -n00, add n00

etc.

this approach also helps with part 2 of this problem, but we will get to that later

writing hard camel

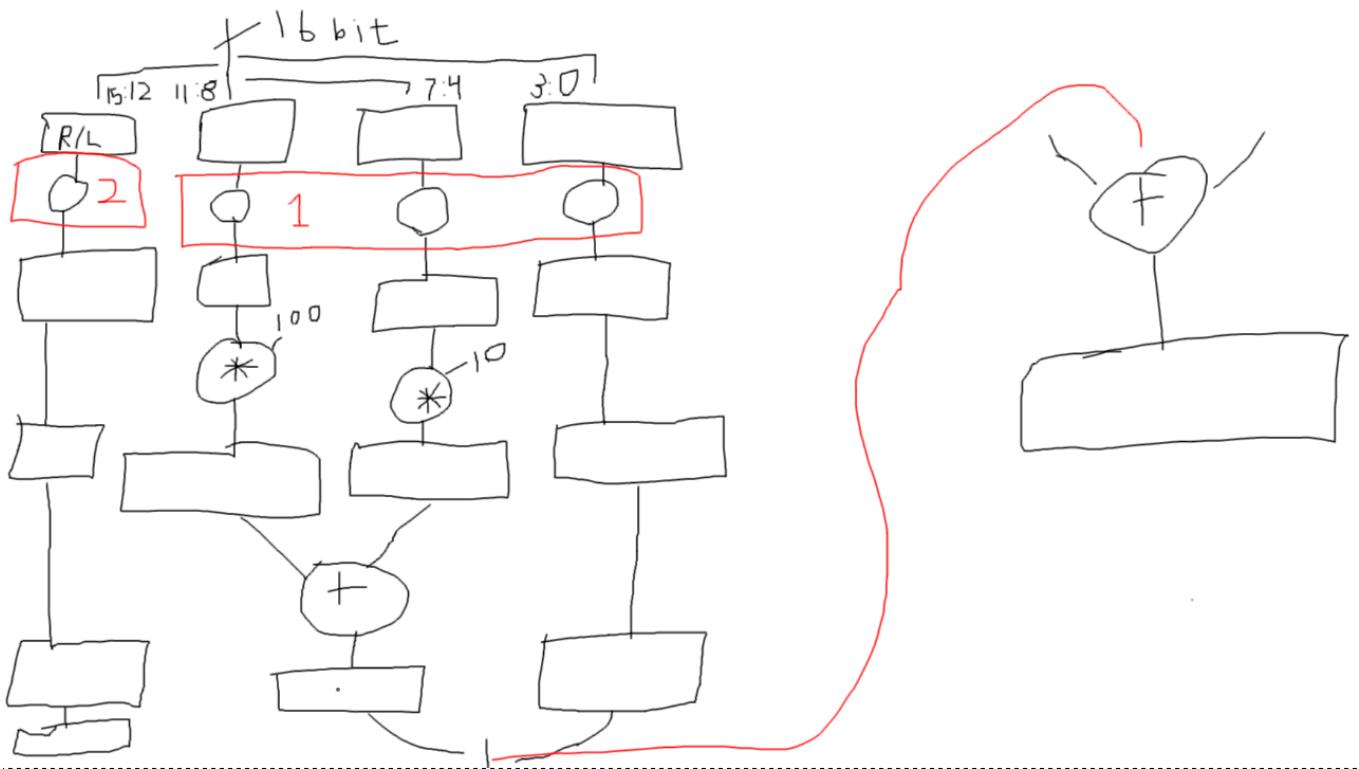
I have a lot of experience with VHDL, but a free t-shirt is ***way too enticing***.

Lets start with the ascii to readable input

I saw this https://www.janestreet.com/web-app/hardcaml-docs/more-on-circuit-design/binary_coded_decimal in the sort of awesome docs.

We will be copying as much of it as possible (I ended up copying none of it)

Here's what I'm thinking:



1. comb. logic that converts ascii digits into binary numbers (0-9)
2. comb. logic that converts ascii for R into '1' (and anything else into '0')

Sorry the design wraps like that, I spent 15 minutes trying to extend the canvas in GIMP before giving up...

There's a part of me that wants to just use an always block and manually declare each register and wire, but I feel like that wouldn't be idiomatic hardcaml... (I have no idea what idiomatic hardcaml is)

I read the docs (https://www.janestreet.com/web-app/hardcaml-docs/designing-circuits/sequential_logic) and have determined that I'd rather have a long line of
`let ... in ...` that define signals, registers, and outputs

This might come back to haunt me if I ever want to stall this pipeline (common when dealing with evil DRAM)

Also worth noting this assumes each input is the full 4 ascii characters (will fix later)

Here's my first iteration of the IO interface

```

let input_width  = 32
let output_width = 16

module I = struct
  type 'a t =

```

```

{ din : 'a [@bits input_width] (* 4*8=32 *)
; clock : 'a
; clear : 'a
}
[@@deriving hardcaml]
end

module O = struct
  type 'a t =
  { number : 'a [@bits output_width]
  ; was_r : 'a (* 1 for R, 0 for L*)
  }
[@@deriving hardcaml]
end

```

Here's my first iteration of the pipeline:

```

let create_scope ({ din } : _ I.t) : _ O.t =
  let spec = Signal.Reg_spec.create ~clock ~clear () in
  let chars = split_msb ~part_width:8 din in
  let num_chars = List.length chars in
  assert (num_chars = 4);

  (* stage 1 register raw ascii char inputs *)
  let chars_r = List.map (Signal.reg spec) chars in

  (* ideally we leave in list and keep doing operations
   on it, but char1 needs something different from char2
   so we have to pull them out *)
  let (c3_r, c2_r, c1_r, c0_r) =
    match chars_r with
    | [c3_r; c2_r; c1_r; c0_r] -> c3, c2, c1, c0
    | _ -> assert false
  in

  (* between stages 1 and 2, comb logic to
   convert ascii to binary, or R to 1 other to 0
   no registers here *)
  let c3_r_is_R = c3_r =: of_int ~width:8 (Char.code 'R') in
  (* we will need to resize later so no point compressing to 4 bits now *)
  let c2_r_as_bcd = c2_r -: of_int ~width:8 (Char.code '0') in
  let c1_r_as_bcd = c1_r -: of_int ~width:8 (Char.code '0') in
  let c0_r_as_bcd = c0_r -: of_int ~width:8 (Char.code '0') in

  (* stage 2: register the previous comb logic, and save some pain on R/L
  signal *)

```

```

(*we just need to delay the c3_r_is_R signal and then finally use it as
an output*)
let c3_r_is_R_delay_4 = Signal.pipeline spec ~enable:Signal.vdd ~n:4
c3_r_is_R in

(* same is true for the least sig char, which doesn't need bcd->binary
conversion
and isn't needed until the last stage of the pipeline*)
let c0_bcd_delay_3 = Signal.pipeline spec ~enable:Signal.vdd ~n:3
c0_r_as_bcd in

(* rest just get another dff *)
let c2_r_as_bcd_r = Signal.reg spec c2_r_as_bcd in
let c1_r_as_bcd_r = Signal.reg spec c1_r_as_bcd in

(* between stages 2 and 3, comb logic for
BCD to binary, no need to worry about RL signal
multiply digits by respective powers of 10
*)

let c2_as_binary = uresize (c2_r_as_bcd *: (of_int ~width:10 100))
~width:10 in
let c1_as_binary = uresize (c1_r_as_bcd *: (of_int ~width:10 10))
~width:10 in

(* stage 3! almost done! register the three values :)*)
let c2_as_binary_r = Signal.reg spec c2_as_binary in
let c1_as_binary_r = Signal.reg spec c1_as_binary in

(* between stages 3-4! lets gooooo
just add the previous two reg's *)
let c2_plus_c1 = uresize (c2_as_binary_r +: c1_as_binary_r) ~width:11 in

(*stage 4!*)
let c2_plus_c1 = Signal.reg spec c2_plus_c1 in

(*between stages 4 and 5! one more sum to go!*)
let final_sum = uresize (c2_plus_c1 +: uresize c0_bcd_delay_3 ~width:11)
~width:output_width in

(*stage 5!*)
let final_sum_r = Signal.reg spec final_sum in
{
    number : final_sum_r;
}

```

```
    was_r : c3_r_is_R_delay_4;
};;
```

worth noting this code does not compile

list of bugs:

- I didn't fully extract the I module in create function signature
- I called List.map (function) list instead of ~f:(function) list
- I used the wrong variable names in the match block (c3 instead of c3_r)
- I used =: instead of ==:
 - this one pisses my off why would ocaml use = but hard caml use ==
- I used of_int instead of of_int_trunc
- I used Char.code but core makes this Char.to_int (i did this 4 times)
- I didn't do .value at the very end when defining outputs
 - And I didn't need to I was just confused
- I had no idea how records worked in ocaml and had to google it
- I exposed create instead of hierarchical in my mli and the signature didn't match
- I didn't use some of the registers on accident, thank god the compiler doesn't like un-used variables
- I didn't actually need the scope variable (but maybe will in the future) so i removed it

A good lsp would have prevented me from making a lot of these errors in the first place.

Maybe there is a good hardcaml lsp but I don't know about it and the default ocamllsp doesn't really do anything :(

Finally we have a pipeline that builds!

half way there :(

We have to make our test bench now :(

i copied a test bench for an and gate (which i copied from the template project) and changed it a little

```
let simple_testbench
  ~(inputs : Bits.t ref Parse_ascii.I.t)
  ~(outputs : Bits.t ref Parse_ascii.O.t)
  (sim : Harness.Sim.t)
=
```

```

let cycle () = Cyclesim.cycle sim in

let bits_of_ascii str =
  assert (String.length str = 4);
  let open Bits in
  concat_msb (List.init 4 ~f:(fun i -> of_int_trunc ~width:8 (Char.to_int str.[i]))) in
  let set raw_ascii =
    inputs.din := (bits_of_ascii raw_ascii);
    cycle ();
    cycle ();
    cycle ();
    cycle ();
    cycle ();
    cycle ();
    let r = Bits.to_int_trunc !(outputs.was_r) in
    let number = Bits.to_int_trunc !(outputs.number) in
    print_s [%message "pipeline result" (raw_ascii : string) (r : int)
  (number : int)]
  in

  set "R123";
;;

```

I have good news and bad news:

```

@@ -45,7 +45,7 @@ let simple_testbench
let%expect_test "AND gate truth table" =
  Harness.run ~create simple_testbench;
  [%expect
-  {||}]
+  {| ("pipeline result" (raw_ascii R123) (r 1) (number 123)) |}]
;;

```

Code I wrote worked first try! (ignore all the bugs and the issues i had writing the tb that i didn't even talk about)

```

@@ -69,6 +69,24 @@ let%expect_test "AND gate with printed waveforms" =
    simple_testbench;

[%expect
-  {||}
+  {
+    ("pipeline result" (raw_ascii R123) (r 1) (number 123))
+    Signals---Waves
+    clear
+    clock
+    din
+    number
+    was_r
+    gnd
+  }
];
;;

```

The wave form doesn't match :(

lets debug this waveform

First, is din actually the decimal representation of our ascii input?

I checked, it was

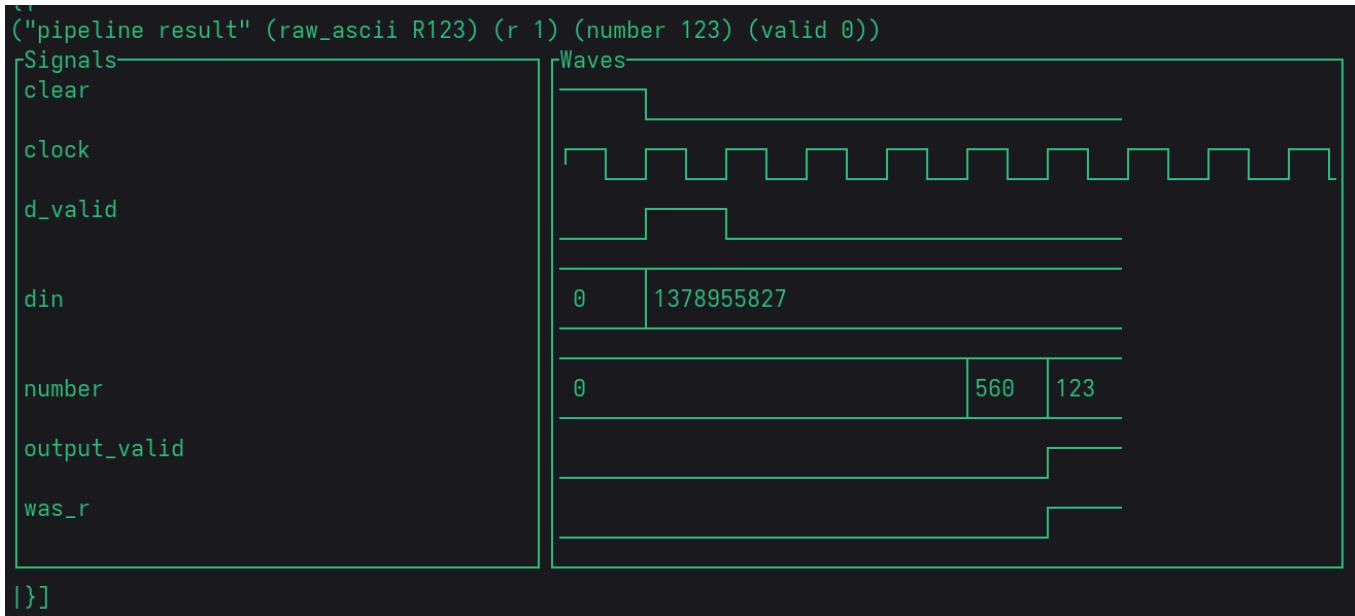
So why are we getting 560 instead of 123? :(

1. is 560 not in base 10?
 1. $123 = \text{hex } 291$
 2. $560 \text{ hex} = 1376$

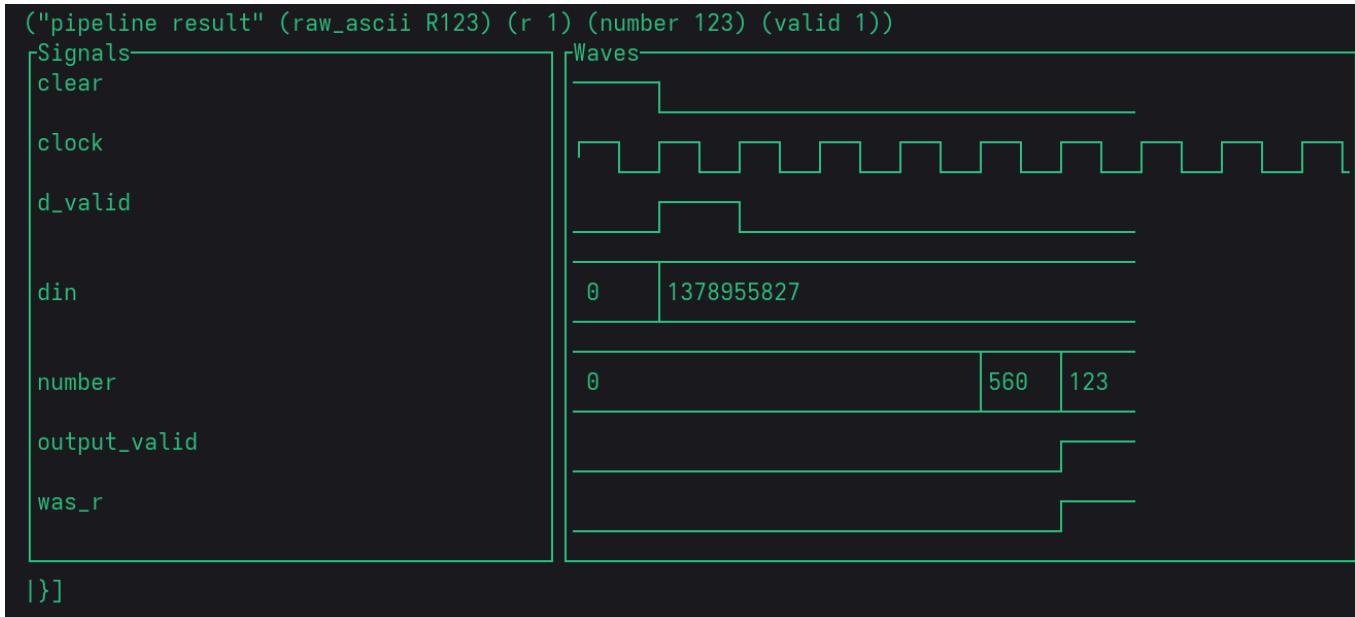
it wasn't a bug!

My code was correct but 560 is an intermediate that comes from the fact that $0b00000000 - '0'$ (the binary number 0 minus ascii 0) ends up wrapping around and producing a non-zero output. This cascades into the rest of my pipeline. It has become clear that I need to add a 'valid_in' signal to my data/pipeline so I know when the output is valid (before I assumed that a non-zero output implied a valid output)

we can't be serious:



now the waveform is correct but the printed result isn't?! Is this some kind of sick joke?



Fixed :P

better code and better tests

Now that we have a WIP design and test bench, we can start making it better.

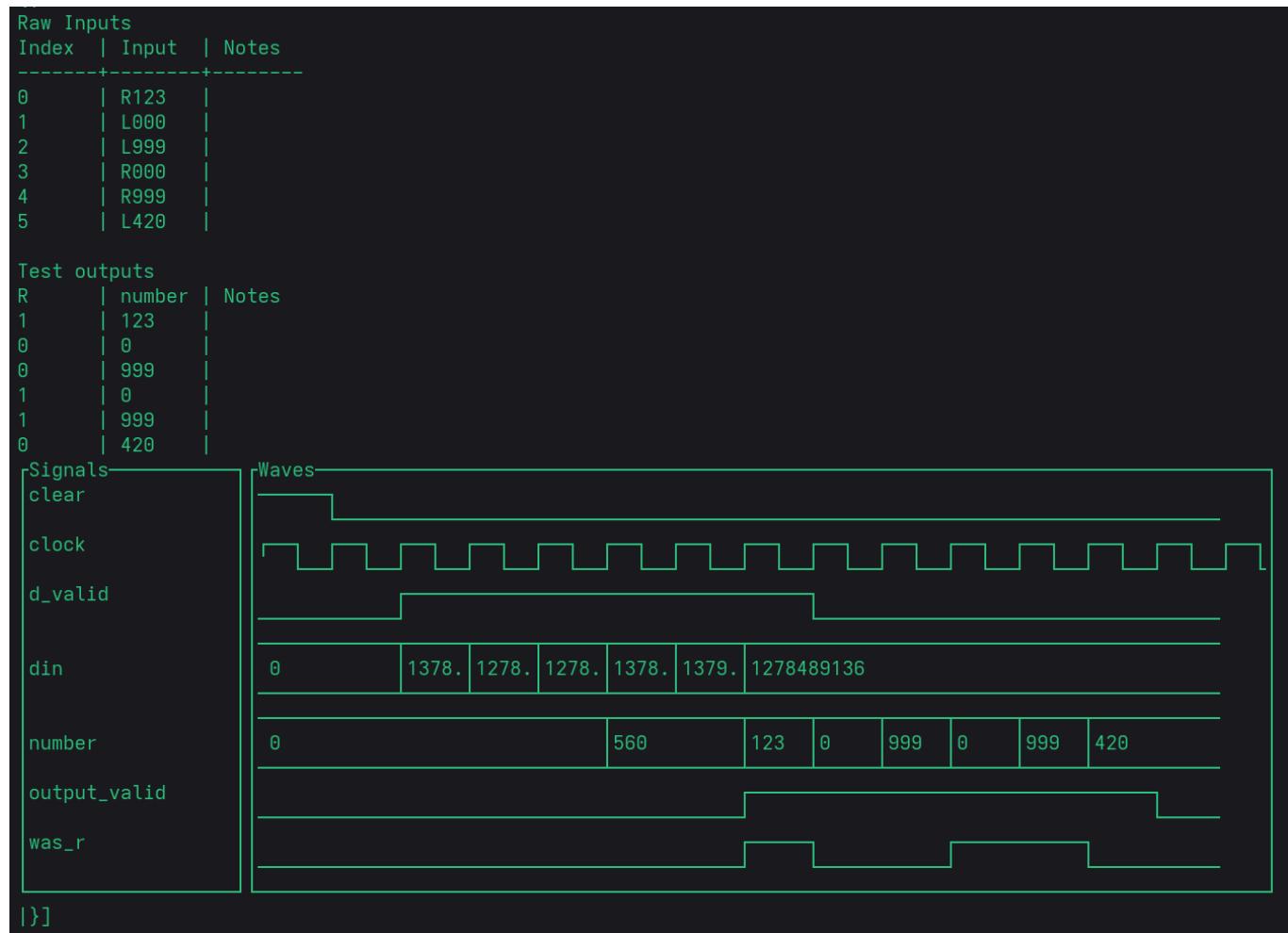
First lets beef up the tb with more edge cases and more than one input at a time

I added a list of inputs to test

```
let raw_inputs = ["R123"; "L000"; "L999"; "R000"; "R999"; "L420"];;
```

I also changed `cycle()` from the demo project to `step()` which, on top of incrementing by one clock cycle, also checks to see if output valid is true

Here's what the final "passing" test looked like:



Pretty awesome!

finally, we can handle the case where the input isn't 4 ascii characters

We would like to cover inputs such as:

- R1
- R12
- L1
- L10

We are working with this snippet of code

```
let spec = Signal.Reg_spec.create ~clock ~clear () in
let chars = split_msb ~part_width:8 din in
```

```
let num_chars = List.length chars in
assert (num_chars = 4); (*replace with code that ensures num_chars=4*)
```

I defined the inputs as coming from some kind of ROM or RAM so what would a 2 or 3 character input look like in memory?

lowkey I get to decide how it would look

I define the memory layout for 2 and 3 character inputs to be:

R1 = | 0b00000000 | |0b00000000| | 'R' | | '1' |

R11 = | 0b00000000 | | 'R'| | '1' | | '1' |

So from memory, the input is left padded with zeros

it could also have been right padded, I made this choice completely at random

we want to check the first two msb's and act based off them

this is a great time to use hardcamls cases keyword!

Here's what I came up with

```
let raw_chars = split_msb ~part_width:8 din in
let packed_chars =
  match raw_chars with
  | [a; b; c; d] ->
    let is_2_chars =
      (a ==: binary_zero) &:
      (b ==: binary_zero) in
    let is_3_chars =
      a ==: binary_zero in
    let sel = is_3_chars @: is_2_chars in
    cases sel
    ~default:(din)
    [
      of_string "10", concat_msb [b; ascii_zero; c; d]
      ; of_string "11", concat_msb [c; ascii_zero; ascii_zero;
d]
    ]
    | _ -> assert false in
let chars = split_msb ~part_width:8 packed_chars in
```

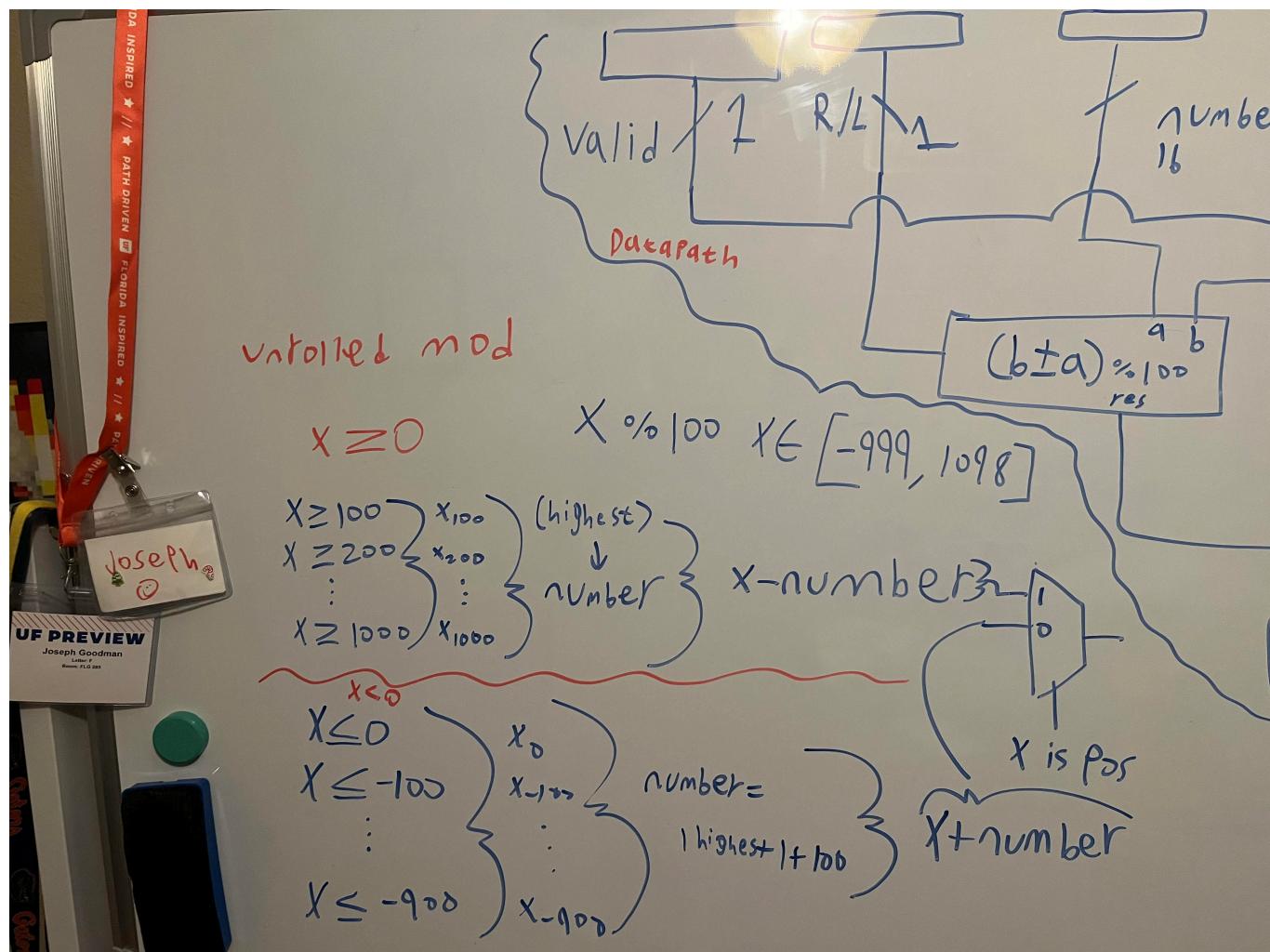
Note that I had to call concat_msb on my list of bit_vectors before returning it from the cases statement.

Then I had to call split_msb to unpack the single mega-bit-vector into a list of bit_vectos.

This is another thing that I sorta don't like about hardcaml, why must mux and cases only work on Signal.t and not also Signal.t list?

writing the adder + modulo thing

Here's the design I'm thinking



sorry I drew a register enabled by the valid signal and fed by the $(b+a)\%100$
the register also has a zero flag that drives a counter
all of this is in the same clock domain as the previous part

I'm sort of worried that the unrolled mod operator is going to ruin my clock frequency but in theory it's only 3 times as bad as anything I did in part 1

writing the mod operator

The unrolled mod thing is complicated enough that I'd like to write it as its own 'entity' with its own test bench

I'll skip the setup this time but we have a purely comb. logic circuit with a working test bench

```
%expect
-  (| |)
+  {}
+ Raw Inputs
+  Index | Input | Notes
+  -----+-----+
+  0     | 123   |
+  1     | 1      |
+  2     | -45   |
+  3     | 999   |
+  4     | 9      |
+  5     | -76   |
+  6     | -999  |
+  7     | 420   |
+
+ Test outputs
+  hi    | number | Notes
+  | 123   |
+  | 1      |
+  | 65491  |
+  | 999   |
+  | 9      |
+  | 65460  |
+  | 64537  |
+  | 420   |
+
Signals          Waves
+ din
+   123 | 1 | 65491 | 999 | 9 | 65460 | 64537 | 420
+
+ dout
+   123 | 1 | 65491 | 999 | 9 | 65460 | 64537 | 420
+
| []
;;
;
```

i left the Chat-gpt formatted tables, we start with just a wire,
input=output

Not the comically large outputs that are strangely close to a power of two, these are negative numbers that are cast back to ocaml as unsigned. I'm not fixing it since the result of the mod should never be negative (and also should always be strictly less than 99).

from here we can actually start designing

we follow the idomatic hardcaml design from the sort of awesome docs:

https://www.janestreet.com/web-app/hardcaml-docs/designing-circuits/priority_encoder

```

let create_scope ({ din } : _ I.t) : _ O.t =
  (*first we need some logic to *)
  let din_is_pos = din >=+ (of_int_trunc ~width:16 0) in (* i'd expect
this not to synthesize into a comparitor *)
  let pos_sel = List.map ~f:(of_int_trunc ~width:16) [100; 200; 300; 400;
500; 600; 700; 800; 900; 1000]
    |> List.map ~f:(fun num -> din >=: num)
    |> Signal.concat_msb in
  let pos_mod_result = priority_select_with_default
    ~default:din
    With_valid.
    [ { valid = pos_sel.:(0); value = din -: (of_int_trunc ~width:16
1000) }
      ; { valid = pos_sel.:(1); value = din -: (of_int_trunc ~width:16
900) }
      ; { valid = pos_sel.:(2); value = din -: (of_int_trunc ~width:16
800) }
      ; { valid = pos_sel.:(3); value = din -: (of_int_trunc ~width:16
700) }
      ; { valid = pos_sel.:(4); value = din -: (of_int_trunc ~width:16
600) }
      ; { valid = pos_sel.:(5); value = din -: (of_int_trunc ~width:16
500) }
      ; { valid = pos_sel.:(6); value = din -: (of_int_trunc ~width:16
400) }
      ; { valid = pos_sel.:(7); value = din -: (of_int_trunc ~width:16
300) }
      ; { valid = pos_sel.:(8); value = din -: (of_int_trunc ~width:16
200) }
      ; { valid = pos_sel.:(9); value = din -: (of_int_trunc ~width:16
100) }
    ] in
  let neg_sel = List.map ~f:(of_int_trunc ~width:16) [-100; -200; -300;
-400; -500; -600; -700; -800; -900]
    |> List.map ~f:(fun num -> din <=+ num)
    |> Signal.concat_msb in
  let neg_mod_result = priority_select_with_default
    ~default:(din +: (of_int_trunc ~width:16 100))
    With_valid.
    [ { valid = neg_sel.:(0); value = din -: (of_int_trunc ~width:16
1000) }
      ; { valid = neg_sel.:(1); value = din -: (of_int_trunc ~width:16
900) }
      ; { valid = neg_sel.:(2); value = din -: (of_int_trunc ~width:16
800) }
      ; { valid = neg_sel.:(3); value = din -: (of_int_trunc ~width:16

```

```
700) }
      ; { valid = neg_sel.: (4); value = din -: (of_int_trunc ~width:16
600) }
      ; { valid = neg_sel.: (5); value = din -: (of_int_trunc ~width:16
500) }
      ; { valid = neg_sel.: (6); value = din -: (of_int_trunc ~width:16
400) }
      ; { valid = neg_sel.: (7); value = din -: (of_int_trunc ~width:16
300) }
      ; { valid = neg_sel.: (8); value = din -: (of_int_trunc ~width:16
200) }
    ] in
let mod_result = mux2 din_is_pos pos_mod_result neg_mod_result in

{
  dout = mod_result;
}
```

worth noting the above code has a typo

{	Raw Inputs	
Index	Input	Notes
-----+-----+-----		
0	123	
1	1	
2	-45	
3	999	
4	9	
5	-76	
6	-999	
7	420	
Test outputs		
hi	number	Notes
	23	
	1	
	55	
	99	
	9	
	24	
	1	
	20	
Signals		
din	Waves	
dout		
		123 1 65491 999 9 65460 64537 420
		23 1 55 99 9 24 1 20

now that we have the mod operator we can make the "data path"

this will be my first time using one circuit inside another so we'll see how well that goes

at this point I'm kinda wanting some kinda tooling where i could say

```
opam intantiate component --inputs {} --outputs {}
```

and it makes the files and a super basic test bench for me because I feel like the setup for making a new thing is creating a basic IO interface, copying code, and making a super basic test bench

it could even have a flag for if the component were combinatorial or sequential

We'll need a wire here because the adder/mod component takes input from the dial reg, but the dial reg takes input from the adder/mod component

```
(* we assume all inputs are 4 chars for now, we can handle other cases later *)
let create_scope ({ din; r; valid; clock; clear } : _ I.t) : _ O.t =
  let spec = Signal.Reg_spec.create ~clock ~clear () in
  let%hw dial_reg_out = Signal.wire 16 in
  let counter_reg_out = Signal.wire 16 in (* we don't need this *)

  let value_to_mod = mux2 r (dial_reg_out +: din) ( dial_reg_out -: din)
in
  let mod_100_scope = Scope.sub_scope scope "mod100" in
  let mod_out =
    Mod_hundred.hierarchical
      mod_100_scope
      { din = value_to_mod } in

  let dial_reg_zero = dial_reg_out ==: of_int_trunc ~width:16 0 in

  let dr = Signal.reg spec ~enable:valid ~clear_to:(of_int_trunc ~width:16
50) mod_out.dout in
    let cr = Signal.reg spec ~enable:dial_reg_zero (counter_reg_out +:
of_int_trunc ~width:16 1) in

    Signal.(dial_reg_out <-- dr);
    Signal.(counter_reg_out <-- cr);
  {
    result = counter_reg_out;
  }
```

even though it includes an unnecessary wire, I'm confident this is the best hard caml I've written so far

I can sort of see the light at the end of the tunnel, so I sort of skipped improving the test bench once I saw one input behave properly

whats left

integration hell and a little bit of test benching :)

Let's put everything together and write a testbench that simulates my actual puzzle input

ensuring I we meet the goal

remember at the beginning when I mentioned $O(n+L)$, this test bench will enforce that

```
inputs.clear <--. 1;
cycle ();
inputs.clear <--. 0;
cycle ();
List.iter ~f:(set) raw_inputs;

for _ = 1 to 7 do
  cycle ()
done;
let number = Bits.to_int_trunc !(outputs.number) in
Core.printf "%-6d | %-6d | %-6s\n" 0 number "";
```

Each call to `set` has 1 cycle and we include 7 extra cycles after that.

and

```
{ |
```

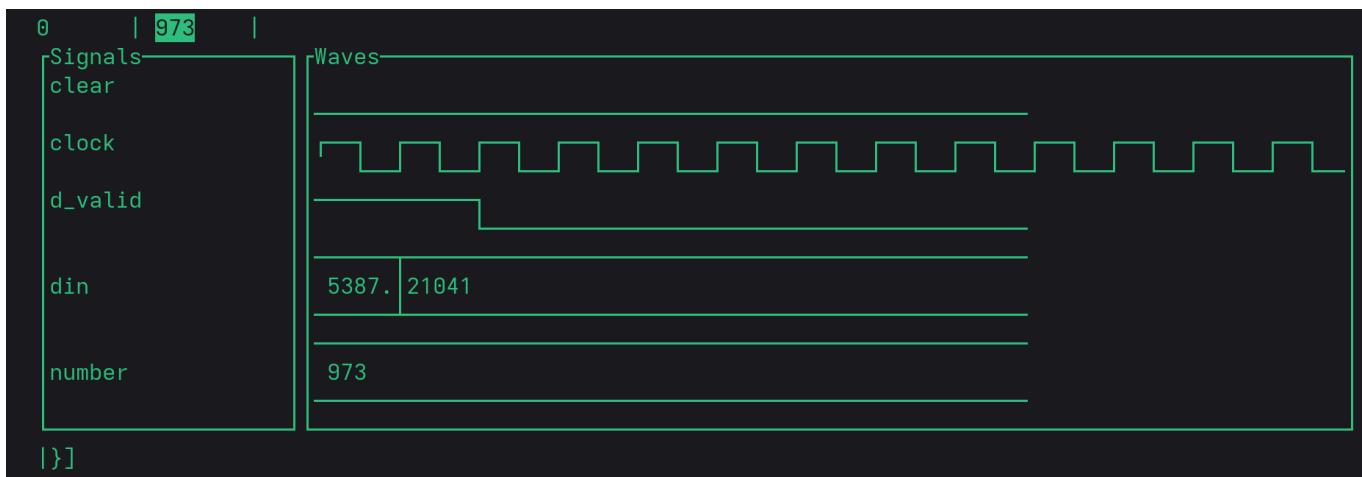
Raw Inputs

Index	Input	Notes
0	L68	
1	L30	
2	R48	
3	L5	
4	R60	
5	L55	
6	L1	
7	L99	
8	R14	
9	L82	
0	3	

boom! the example from aoc day 1 passes!

lets try with my real puzzle input!

I'm not about to learn how to read an input from a text file so I'll just copy-paste the entire puzzle input into the .ml test file :)



this value is wrong :(

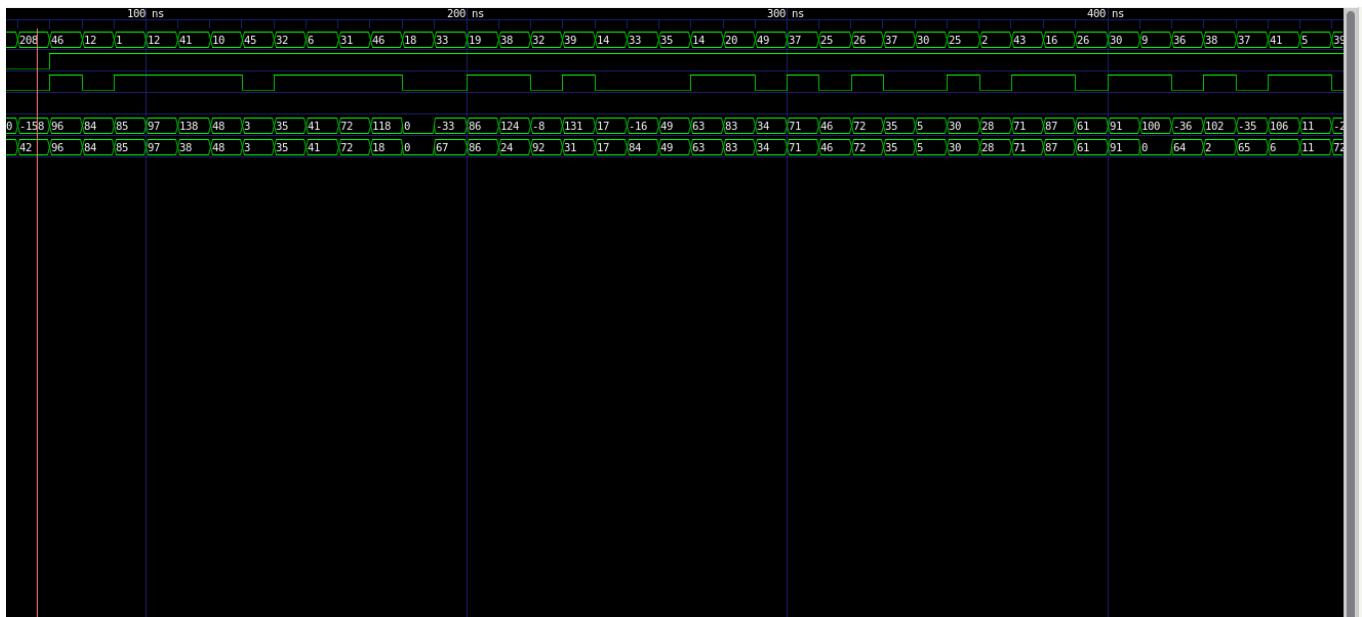
just think about the t-shirt

This is probably the hardest part of most engineering fields: idk how to debug this seeing as all the parts worked in isolation ;(

I'll have to export to a real waveform viewer to see what's wrong here...

learning new things is easy, actually

It was super easy to export the waveform to vcd and even easier to view the vcd :)



I'm looking at the outputs of my ascii pipeline as well as the outputs from my mod operator

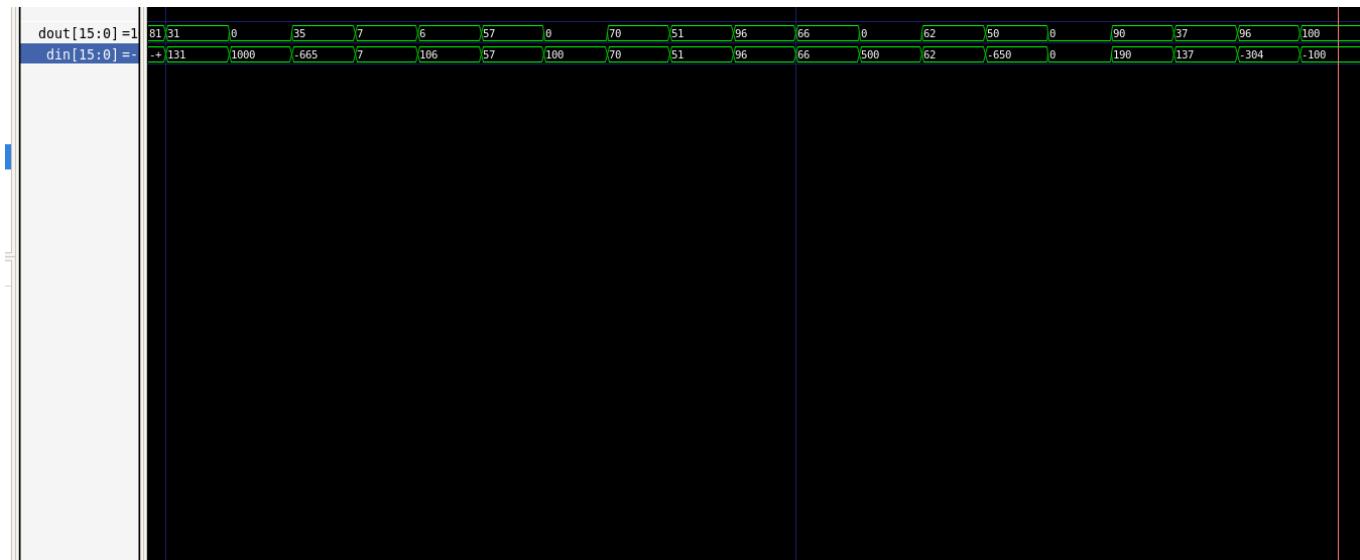
I can compare these to a working software solution and hopefully spot a difference

luckily I have already done day 1 in rust, just gotta go find my old laptop

I exported all of the mod results from software to one file `out.txt` and all of the mod results from hardware to another file `signal.txt`

```
$ diff out.txt ../ocaml/waveforms/signal.txt
0a1,3
> 50
> 90
> 42
294c297
< 0
---
> 100
332c335
< 0
---
> 100
340c343
< 0
---
> 100
399c402
< 0
...
```

The two files differ! We can debug this. This output is actually super telling since it's clear that we have the value 100 in our mod output, which we shouldn't



Here's part of the waveform where the input to the mod component is breaking some rules :(

The output is 100 when the input is -100, let's go to the test bench!

```
{
Raw Inputs
Index | Input | Notes
-----+-----+-----
0    | 123   |
1    | 1      |
2    | -45   |
3    | 999   |
4    | 9      |
5    | -100  |
6    | -200  |
7    | 420   |
}
```

```
Test outputs
hi  | number | Notes
    | 23     |
    | 1      |
    | 55     |
    | 99     |
    | 9      |
    | 100    |
    | 100    |
    | 20     |
[]]
```

Bug spotted!

```
|> List.map ~f:(fun num -> din <=+ num)
(*should be <+ not <=+*)
```

lets re-run my actual puzzle input:

```
{}
0      | 1084   |
Saved waves to /home/jh/Code/ocaml/waveforms/test_day_one_ml_AND_gate_truth_table.vcd
[]]
```

now that I'm 'done' I lose all motivation to do anything since I like hardware design, not submitting hardware design contests